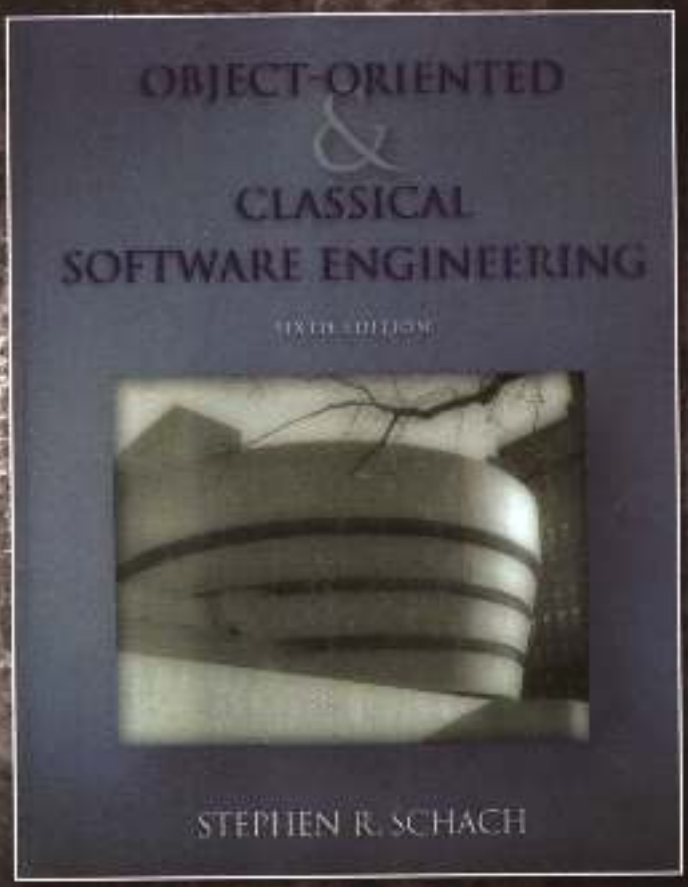
Education 计 算 机 科 学 丛 书

原书第6版

# 面向对象与传统软件工程

## 统一过程的理论与实践

(美) Stephen R. Schach 著 韩松 邓迎春 译




Object-Oriented & Classical Software Engineering

Sixth Edition

机械工业出版社
China Machine Press

本书是软件工程领域的经典著作，被国外多所大学选用为教材，至今已是第6版。从Java到面向对象，再到统一过程的软件开发方法，面对软件工程发展中的重大方法上的改变，本书作者都以严谨的态度对其软件工程专著进行修订，使软件工程在理论上与新的方法和实践更加紧密结合。

全书共分两部分：第一部分介绍了软件工程的理论基础；第二部分讲述了软件生命周期的各个阶段。采用这种独特的、极具可读性的组织方式，学生可以更加系统地学习软件工程。

本书特点

- 包含两个大型的运行实例，使学生能更好地学习和实践书中的内容。
- 全面包含统一过程的内容，并用专门的一章来介绍UML，帮助学生提前应对软件业的职场考验。
- 增加新的分析和设计项目，鼓励学生通过动手来掌握知识。
- 囊括600多条参考材料，其中包括许多新材料，便于读者进一步阅读。
- 配套站点www.mhhe.com/schach包含习题解答、教学幻灯片、大型实例的完整实现等。

作者简介

## Stephen R. Schach



1972年获瑞兹曼科学院物理学理科硕士学位，1973年获开普敦大学应用数学博士学位。Schach在软件工程领域有广泛研究，特别是对软件维护与开放源代码软件的实验分析有深入研究。他著有多部有关软件工程、面向对象软件工程、面向对象系统分析与设计的教材，署名出版物（包括录像带）达100部之多。他还在国际上广泛讲授软件工程方面的课程，包括重用、CASE和面向对象范型等。

ISBN 7-111-17963-3

9 787111 179634

HZ BOOKS 华章图书

上架指导：计算机科学/软件工程

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

ISBN 7-111-17963-3/TP · 4572

定价：55.00 元

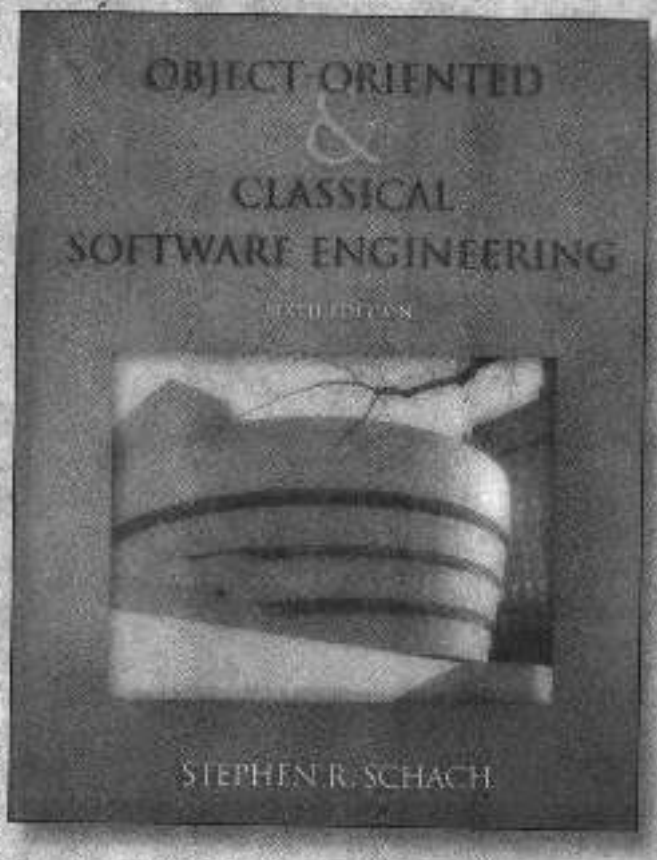
计 算 机 科 学 丛 书

原书第6版

# 面向对象与传统软件工程

## 统一过程的理论与实践

(美) Stephen R. Schach 著　韩松　邓迎春　译




**Object-Oriented & Classical Software Engineering**

**Sixth Edition**

机械工业出版社
China Machine Press

本书是一本经典的软件工程教科书，自1990年首次出版以来，这已是第6次修订出版。全书共分为两部分：第一部分介绍了以项目开发为基础的软件工程的理论基础；第二部分讲述了软件生命周期的各个阶段。

第6版的亮点在于结合了统一过程，并深入介绍了UML，使得内容更具实用性和时效性。此外，书中还包含两个大型的运行实例、大量的参考文献及习题集，使得读者能更好地学习和实践书中的内容。

本书是高等院校软件工程课程的理想教材，对于专业软件开发人员，本书也是一个很好的参考。

Stephen R. Schach: Object-Oriented and Classical Software Engineering, Sixth Edition (ISBN 0-07-111191-3).

Copyright © 2005,2002,1999,1996,1993,1990 by The McGraw-Hill Companies, Inc.

Original English edition published by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition jointly published by McGraw-Hill Education (Asia) Co. and China Machine Press.

本书中文简体字翻译版由机械工业出版社和美国麦格劳－希尔教育(亚洲)出版公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

**版权所有，侵权必究。**

**本书法律顾问　北京市展达律师事务所**

**本书版权登记号：图字：01-2005-1185**

**图书在版编目（CIP）数据**

面向对象与传统软件工程：统一过程的理论与实践（原书第6版）/（美）斯凯奇（Schach, S.R.）著；韩松等译. －北京：机械工业出版社，2006.1

（计算机科学丛书）

书名原文：Object-Oriented and Classical Software Engineering

ISBN 7-111-17963-3

Ⅰ.面…　Ⅱ.①斯…　②韩…　Ⅲ.软件工程　Ⅳ.TP311.5

中国版本图书馆CIP数据核字(2005)第142109号

机械工业出版社(北京市西城区百万庄大街22号　邮政编码　100037)

责任编辑：朱起飞

北京慧美印刷有限公司印刷 · 新华书店北京发行所发行

2006年2月第1版第1次印刷

787mm×1092mm 1/16 · 28印张

印数：0 001-4 000册

定价：55.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：(010)68326294

# 出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzjsj@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

# 专家指导委员会

（按姓氏笔画顺序）

| | | | | |
|---|---|---|---|---|
| 尤晋元 | 王　珊 | 冯博琴 | 史忠植 | 史美林 |
| 石教英 | 吕　建 | 孙玉芳 | 吴世忠 | 吴时霖 |
| 张立昂 | 李伟琴 | 李师贤 | 李建中 | 杨冬青 |
| 邵维忠 | 陆丽娜 | 陆鑫达 | 陈向群 | 周伯生 |
| 周克定 | 周傲英 | 孟小峰 | 岳丽华 | 范　明 |
| 郑国梁 | 施伯乐 | 钟玉琢 | 唐世渭 | 袁崇义 |
| 高传善 | 梅　宏 | 程　旭 | 程时端 | 谢希仁 |
| 裘宗燕 | 戴　葵 | | | |

# 译 者 序

Stephen R. Schach 先生所著的软件工程书籍是这一领域的经典著作，本书已是第 6 版。从 Java 到面向对象，再到统一过程的软件开发方法，每次面对软件工程发展中的重大方法上的改变，作者都以严谨的态度对其软件工程专著进行新的修改和补充，使软件工程在理论上与新的方法和实践更加紧密结合。

为了适应面向对象方法的深化与统一软件开发过程的广泛采用，本书第 6 版对第 5 版做了较为全面的修改，重新编写和调整的内容占一半以上。可以说，本书是软件工程专著中最能反映该学科领域最新变化的一本。

本书有两个主要特点：

其一，本书将当前流行的面向对象内容与传统软件工程很好地融合在一起，尤其突出了面向对象和 UML 方面的内容。这在同类书籍中实为少见，由此也显得弥足珍贵。

其二，本书包含了两个贯穿全文的极具代表性的运行实例，作者借助实例细致而巧妙地演示了各种相关概念，使得整个学习过程水到渠成。

本书主要由韩松和邓迎春翻译，其中第 1～8 章由韩松翻译，余下部分由邓迎春翻译。最后由韩松校对并审核全书。其他参加翻译和录入工作的同志还有赵玉亮、徐燕、张小明、傅兵、王勇、徐志东、刘云杭等，在此对他们的工作表示感谢。此外，感谢出版社编辑的大量具体指导和耐心细致的帮助，他们对保证本书的翻译质量以及尽快出版做出了很大贡献。同时也感谢我们的家人对我们的支持。

本书翻译中难免有差错，有不当之处欢迎广大读者批评指正。

韩 松

2005 年 11 月

# 前　言

自从3年前本书第5版出版以来，统一过程（Unified Process）已经得到广泛认可，被选为面向对象软件开发的方法。统一过程基于Booch的方法、对象体（Objectory）以及OMT，这三个较早的面向对象方法的提出者们现在已不再支持它们。因此，第6版新增的主要特性是采纳了统一过程。特别地，书中使用统一过程开发了两个运行实例研究，因此学生们既是统一过程的理论学习者，也是该理论的实践者。

**第6版的其他关键特色**

第1章经过了全面的修订。特别是增强了面向对象范型（paradigm）的内容，并进行了深入的分析。此外，书中引入了新的可维护性的定义，它们被ISO、IEC、IEEE和EIA所采纳。

第2章和第3章的顺序互换了，这样可以尽早介绍进化树生命周期模型和迭代－递增生命周期模型。但是，我仍然像本书的前几版一样，给出了大量其他的生命周期模型，并对它们进行了比较和对照。

在第3章“软件过程”中，介绍了工作流（活动）和统一过程的各个阶段，还解释了对二维生命周期模型的需求。

第4章～第9章都进行了更新。例如，第8章中介绍了基于组件（component）的软件工程，第9章中给出了软件项目管理计划的新的IEEE标准。符合该新标准的一个计划的例子见附录F。

有关互操作的内容已从第8章中删掉了。在第4版和第5版中，这部分内容在该书出版后不久都不可避免地过时了。在我看来，这个领域进展太快了，无法包含在一本教材中；想要了解这方面内容的教师应当从因特网上获取最新的材料。

第10章“需求”、第12章“面向对象分析”和第13章“设计”经过较大的修改，在其中加入了统一过程的工作流（活动）。出于众所周知的原因，第11章“传统的分析”未做改变。

有关实现和集成（integration）（第5版的第14章和第15章）的内容已经融入了新的第14章，但是，在实现和集成之间仍然有一个明显的区分。

第15章现在是关于交付后维护的。

第16章是全新的一章。第4版是第一本利用统一建模语言（Unified Modeling Language，UML）的软件工程教科书，UML是在该版书出版前不久推出的。在时隔3年之后，UML已经成为正式的标准并广为使用，因此在第5版中，我继续利用UML描述面向对象分析和面向对象设计，并用示意图描绘对象和它们之间的关系。在第4版和第5版中，我加入了大量有关UML的内容，以使学生们能够做完全部的练习，并进行基于小组的学期设计项目。然而，UML现在通常是软件工程的一个组成部分（特别是统一过程的一个组成部分），因此，我加入了最后一章，“UML的进一步讨论”。第16章的目的是提供有关UML的更多材料，为学生们就职于软件行业做进一步的准备。这一章对于使用本书作为连续两学

期软件工程课程的教师特别有用。在第二学期，除了开发基于小组的学期设计项目或一个毕业课题项目之外，学生能够获得额外的有关 UML 的知识，它们已超出本书的范围。

除了两个运行实例研究之外——它们用于说明完整的生命周期，本书还使用 7 个小型实例研究强调特定主题，如移动目标问题、渐进精化以及交付后维护。

有关极限编程（extreme programming，XP）的内容做了扩充。此外，现在在敏捷编程的范围内描述 XP。XP 仍然是充满争议的，但我感到学生们需要理解这个话题，这样，他们可以自己判断是否 XP 只是目前的一个时尚，还是软件工程中一个真正的巨大突破。

**从第 5 版中保留的特色**

这一版保留了第 5 版中的全部主要特色。

- 在先前的所有版本中，我强调文档、维护、重用、可移植性、测试和 CASE 工具的重要性。在这一版中，所有这些概念都依然被着重强调。如果学生不理解软件工程这些基础的重要性，教授他们最新的思想就没有用处。
- 如同第 5 版一样，本书对以下几方面给予了特殊的重视：面向对象生命周期模型、面向对象分析、面向对象设计、面向对象范型管理的含义、面向对象软件的测试和维护。其中还包括了面向对象范型的度量。此外，对对象做了许多简明的注解，有的长达一个段落，有的甚至只是一句话。这样做是因为面向对象范型不仅与各种阶段是如何执行的有关，也与它是如何渗透到我们思考软件工程的方式之中有关。对象技术再次贯穿本书始终。
- 软件过程仍然是贯穿本书的一个整体概念。为了控制这个过程，我们必须能够测量项目中发生了什么。相应地，保留了对度量的强调。考虑到软件过程的进展情况，保留了有关能力成熟度模型（CMM）、ISO/IEC 15504（SPICE）以及 ISO/IEC 12207 的内容，在有关软件小组的章节中加入了人员能力成熟度模型（P-CMM）。
- 本书仍然是与计算机语言无关的，少量代码实例是用C++ 或 Java 表示的，而且我尽量减少与语言有关的一些细节，确保代码实例对于 C++ 和 Java 用户同样清晰。例如，我没有使用 cout 表示C++ 的输出，也没有使用 System.out.printIn 表示 Java 的输出，我使用了伪码指令 print。（一个例外是新的实例研究，在这里完整的实现细节是同时用C++ 和 Java 给出的。）
- 同以前一样，本书有两个运行实例研究。一个是来自第 4 版的 Osbert Oglesby 实例研究，一个是电梯问题实例研究（来自前一版），已经用统一过程对它们进行了重新开发。同样，Java 和 C++ 实现都可在线从 www.mhhe.com/engcs/compsci/schach 处得到。
- 像在第 5 版中一样，本书包含 600 多个参考文献。我选择了目前的研究文章，也选择了一些仍保持新意和用途的经典的文章和书籍。毫无疑问，软件工程是一个快速发展的领域，学生们因此需要知道最新的成果以及在哪些文献里可以找到它们。与此同时，今天的前沿研究是在昨天的事实基础上进行的，我们没有理由将一个较早的参考文献排除在外，如果它的思想在今天如它最初一样仍是在应用着的话。
- 作为一个先决条件，我们假设读者熟悉一种高级编程语言，如 C、C++、Ada 或 Java。此外，读者最好应上过“数据结构”这门课。

### 为什么仍然包括了传统范型

当我开始编写第 6 版时，我决定摒弃谈论传统（结构化的）范型。毕竟，现在大家几乎一致认为面向对象范型比传统范型优越。但是，不久我就发现，试图完全不提及传统范型肯定是不明智的。

首先，若没有完全理解传统范型而且不知道它与面向对象方法的区别，就不可能明白为什么面向对象技术比传统技术优越。例如，面向对象范型使用迭代－递增的生命周期模型，要解释为什么需要这样一个生命周期模型，最根本的是要详细解释在经典的如瀑布模型这样的生命周期模型和面向对象的迭代－递增生命周期模型之间的区别。因此，本书中也包含了有关传统范型的材料，以使学生能够清楚把握传统范型和面向对象范型之间的不同。

我在书中包含两种范型的第二个原因是：技术的转变是一个缓慢的过程。要不是经受不住千年虫（Y2K）问题的影响从而加速了向面向对象范型的转变，大多数软件组织至今还不会采纳面向对象范型。因此，使用本书的许多学生有可能会受雇于那些仍然使用传统软件工程技术的组织。进一步说，即使一些技术组织使用面向对象方法开发新的软件，但现有的软件仍需要维护，而这个延续下来的软件不是面向对象的。因此，排除传统内容对于许多使用这本教材的学生并不合适。

包括两种范型的第三个原因是：受雇于一个正考虑向面向对象技术转变的组织的学生，将能够对该组织认识新范型的长处和短处提出自己的忠告。因此，如同前一版中一样，对传统和面向对象方法进行了比较、对照和分析。

### 第 6 版是如何安排的

就像本书第 5 版一样，第 6 版是为传统的一学期和新的两学期软件工程课程而编写的。在传统的一学期（或 1/4 学年）课程中，教师不得不把理论部分内容匆忙一带而过，以便尽早提供给学生们进行学期设计所必需的知识和技能。为使学生们能够及早开始学期设计，从而得以在学期结束前完成设计，这样的匆忙安排是必要的。为了满足一个学期的基于设计的软件工程课程的要求，本书的第二部分涵盖了按工作流进行的软件生命周期，第一部分包含了理解第二部分所需的理论知识。例如，第一部分介绍了 CASE、度量和测试；第二部分的每一章都有一节讲述了该工作流的 CASE 工具，有一节讲述了该工作流的度量，有一节讲述了在该工作流期间的测试。第一部分编写得简短一些，以便教师能够在一学期里较早开始第二部分内容的讲授。而且，第一部分的最后两章（第 8 章和第 9 章）可以留待与第二部分同时讲授。这样，本课程可以在尽早的时候开始进行学期设计。

我们再来看看两学期软件工程课程。越来越多的计算机科学和计算机工程系认识到它们的毕业生绝大多数将找软件工程师的工作，因而，许多学院和大学采用了一个连续两学期（或两个 1/4 学年）的软件工程课程安排。第一学期课程理论内容占多数（但也有少部分的某种设计实践内容）。第二学期课程主要是以小组为基础的学期设计，通常它是一个前沿课题。当学期设计安排在第二学期课程内时，指导教师就没有必要匆忙开始第二部分内容了。

因此，使用本教材教一学期（或 1/4 学年）课程的教师需要讲解第 1～7 章的大部分，然后开始第二部分（第 10～16 章）。第 8 章和第 9 章可以与第二部分并行讲授，或者在课程结束前当学生们正在进行学期设计时讲授。当教师进行连续两个学期的课程讲授时，应当顺

序讲授本书的各章，通过第一学期的学习，同学们就可以完全准备好进行第二学期的基于小组的学期设计了。

为了确保同学们能够真正理解第二部分一些关键的软件工程技术，每项技术都出现两次。首先，当介绍一项技术的时候，先通过电梯问题来讲解它。电梯问题的难易比较适中，它能够使读者明白应用于计算问题的那项技术，它有许多精妙之处来强调正被讲授的那项技术的优缺点。然后，给出与该项技术有关的 Osbert Oglesby 实例研究的相关部分。这个详细解决方案从另一个角度阐述了每一项技术。

**习题集**

前几版有四种类型的习题。本版有五种类型的习题。第一，新类型的习题是在第 10、12 和 13 章末给出的面向对象分析和设计项目。包含这些项目是因为，若要学习如何执行需求、分析和设计工作流，惟一的途径是进行大量的实习。

第二，每一章结尾包含一些意在突出重点的练习。这些练习是独立的，全部练习的技术信息都可以在本书中找到。

第三，有一个软件学期设计项目。它设计成由三个人组成的小组协作完成，而不是通过常规的电话协商完成，三个人是最少的小组成员数。学期设计项目由 16 个独立的组件组成，每个组件都附在相应的一章之后。例如，设计是第 13 章的主题，因此在该章中此学期设计的组件与软件设计有关。通过将一个大的项目分解为小的、明确定义的几个部分，教师能够更密切地掌控学生的学习进度。学期设计项目的结构很灵活，指导教师能够自由地将这 16 个组件应用于任何其他的项目。

由于本书是为研究生和大学高年级学生编写的，第四种类型的习题是根据软件工程文献中的研究报告拟制的。在每一章中，都选择了一篇重要的文章，尽可能选择一篇与面向对象软件工程有关的文章。要求学生阅读该文章并回答与它的内容有关的一个问题。当然，教师可以自由安排任何其他研究文献，在每一章后的“进一步阅读”部分中包含了各种相关论文。

第五种类型的习题与 Osbert Oglesby 实例研究有关。这类习题首先在第 3 版中引入，它是为响应许多教师的要求而加入的，教师们感觉到：学生通过修改一个现成的产品而不是凑合着开发一个新产品可以学到更多的东西。业界的许多高级软件工程师同意这个观点。基于此，给出实例研究的每一章至少有 3 个问题需要学生在某种程度上修改该实例。例如，在某一章中，要求学生使用一项与该实例研究中不同的设计技术重新设计该实例。在另一章中，要求回答以不同的顺序执行面向对象分析的步骤会产生什么不同的效果。为了方便修改实例研究的源代码，我们在万维网上提供了这些源代码，网址见：www.mhhe.com/engcs/comp-sci/schach。该网址上还有一个完整的 PowerPoint 课件。

“教师答案手册”（Instructor's Solution Manual）中包含了所有习题以及学期项目的详细答案。“教师答案手册”可从 McGraw-Hill 出版公司得到。

**致谢**

非常感谢前 5 版书的审阅者所给予的建设性批评和良好建议，他们是：

**Arvin Agah**
*University of Kansas*

**Kiumi Akingbehin**
*University of Michigan, Dearborn*

**Phil Bernhard**
*Clemson University*

**Dan Berry**
*The Technion*

**Betty Cheng**
*Michigan State University*

**David Cheriton**
*Stanford University*

**Thaddeus R. Crews, Jr.**
*Western Kentucky University*

**Buster Dunsmore**
*Purdue University*

**Eduardo B. Fernandez**
*Florida Atlantic University*

**Michael Godfrey**
*Cornell University*

**Bob Goldberg**
*IBM*

**Donald Gotterbarn**
*East Tennessee State University*

**Scott Hawker**
*University of Alabama*

**Thomas B. Horton**
*Florida Atlantic University*

**Greg Jones**
*Utah State University*

**Peter E. Jones**
*University of Western Australia*

**Gail Kaiser**
*Columbia University*

**Laxmikant V. Kale**
*University of Illinois*

**Helene Kershner**
*University of Buffalo*

**Chung Lee**
*California State Polytechnic, Pomona*

**Richar A. Lejk**
*University of North Carolina, Chapel Hill*

**Bill McCracken**
*Georgia Institute of Technology*

**Don Bickerstaff**
*Eastern Washington University*

**Richard J. Botting**
*California State University, San Bernardino*

**James Cardow**
*Air Force Institute of Technology*

**Susan Mengel**
*Texas Tech University*

**Everald E. Mills**
*Seattle University*

**Fred Mowle**
*Purdue University*

**Ron New**
*Johns Hopkins University*

**David Notkin**
*University of Washington*

**Hal Render**
*University of Colorado, Colorado Springs*

**David S. Rosenblum**
*University of California, Irvine*

**Shmuel Rotenstreich**
*George Washington University*

**Wendel Scarborough**
*Azusa Pacific University*

**Bob Schuerman**
*State College, Pennsylvania*

**Gerald B. Sheble**
*Iowa State*

**K. C. Tai**
*North Carolina State University*

**Toby Teorey**
*University of Michigan*

**Jie We**
*City University of New York*

**Laurie Werth**
*University of Texas, Austin*

**Lee White**
*Case Western Reserve University*

**David Workman**
*University of Central Florida*

**George W. Zobrist**
*University of Missouri, Rolla*

此外，还要特别感谢这一版书的审阅者，他们是：

**Michael Buckley**
*State University New York, Buffalo*
**Catherine Lowry Campbell**
*New Jersey Institute of Technology*
**Werner Krandick**
*Drexel University*
**Owen Lavin**
*DePaul University*
**Donald Needham**
*United States Naval Academy*
**Andy Podgurski**
*Case Western Reserve University*
**Frances Grodzinsky**
*Sacred Heart University*
**Jim Han**
*Florida Atlantic University*
**David C. Rine**
*George Mason University*
**Mansur Samadzadeh**
*Oklahoma State University*
**John H. Sayler**
*University of Michigan*
**Fred Strauss**
*Polytechnic University*

上述审阅者，无一例外地都对本书做出了很大的贡献。尽管如此，我还是要特别感谢DePaul大学的Owen Lavin先生，他提出的许多建议对于改进本书帮助极大。

我要衷心感谢三位先生，他们也对本书的较早版本做出了贡献。第一，Jeff Grey实现了Osbert Oglesby实例研究。第二，Kris Irwin提供了一个完整的学期设计的解决方案，包括用C++和Java将其实现。第三，我的女儿Lauren又一次成为我的“教师答案手册”的合作者，并且她完成了PowerPoint幻灯片的制作。

现在该轮到出版商McGraw-Hill了。如往常一样，我最深切地感谢我的出版人Betsy Jones和策划编辑Emily Lupash，他们自始至终对我提供了帮助和指导。我要感谢排版编辑Gnomi Schrift Gouldin，他又一次向我提出了许多得以实施的建议。与市场经理Dawn Bercier、兼职的高级设计Michelle Whitaker以及媒体项目经理Audrey Reiter一同工作是一件令人高兴的事。最后，我非常希望感谢高级项目经理Jane Mohr，她提供了数不尽的帮助和支持。她解决起问题来总是得心应手。

一如既往地，我乐于与Interactive Composition公司的排版人员们一同工作。Amy Rose是一位非常有能力的项目经理。

我还想感谢来自世界各地的许多教师们，他们就第5版的内容给我发来电子邮件。我十分感谢他们的建议、意见和批评。我也期望着收到教师们有关这一版的意见反馈。我的e-mail地址是srs@vuse.vanderbilt.edu。

学生也是非常有帮助的。我感谢我在Vanderbilt大学的学生们，他们在课堂内外提出了无数的问题和意见。特别地，我想对最近在Vanderbilt大学上我的“统一过程”研究生课程的下列学生表示感谢：Paul Bielaczyc, Zhihong Ding, Shaivya Easwaren, Kenon Ewing, Sarita Gupta, Julia Irani, Andrews Jebasingh, Pavil Jose, Anantha Narayanan, Joshua Phillips, Adam Loeb Small, Larry Thomas, Haripriya Venkatesan, Bin Zhou。我真的对他们的真知和创见表示谢意。

我也对来自世界各地的学生们的有启发性的问题和建设性的建议表示感谢，与前几版一样，我时刻期待着他们关于这一版的反馈。

最后，同往常一样，我感谢我的家人给予我的持续不断的支持。当我早先开始写书时，我不得不在我年幼的孩子和我当下的著书项目之间分配有限的自由时间。现在我的孩子已经长大成人了，他们同我一起在为我的著作忙碌着，写作已经成为我们家庭的活动。请再次允许我把这本书献给我亲爱的妻子Sharon和我亲爱的孩子，David和Lauren。

Stephen R. Schach

# 目　录

出版者的话
专家指导委员会
译者序
前言

**第一部分　软件工程概述**

第 1 章　软件工程的范畴 …… 3
1.1　历史方面 …… 4
1.2　经济方面 …… 5
1.3　维护性方面 …… 6
1.3.1　维护的传统和现代观点 …… 8
1.3.2　交付后维护的重要性 …… 9
1.4　需求、分析和设计方面 …… 11
1.5　小组编程方面 …… 13
1.6　为什么没有计划阶段 …… 13
1.7　为什么没有测试阶段 …… 14
1.8　为什么没有文档阶段 …… 14
1.9　面向对象范型 …… 15
1.10　正确看待面向对象范型 …… 19
1.11　术语 …… 19
1.12　道德问题 …… 22
本章回顾 …… 23
进一步阅读 …… 23
习题 …… 24
参考文献 …… 25
第 2 章　软件生命周期模型 …… 28
2.1　理论上的软件开发 …… 28
2.2　Winburg 小型实例研究 …… 28
2.3　Winburg 小型实例研究心得 …… 31
2.4　野鸭拖拉机公司小型实例研究 …… 31
2.5　迭代和递增 …… 32
2.6　修订的 Winburg 小型实例研究 …… 34
2.7　迭代和递增的风险和其他方面 …… 35
2.8　迭代和递增的控制 …… 37
2.9　其他生命周期模型 …… 37
2.9.1　编码－修补生命周期模型 …… 37
2.9.2　瀑布生命周期模型 …… 38
2.9.3　快速原型开发生命周期模型 …… 39
2.9.4　极限编程 …… 40
2.9.5　同步－稳定生命周期模型 …… 42
2.9.6　螺旋生命周期模型 …… 42
2.10　生命周期模型的比较 …… 45
本章回顾 …… 46
进一步阅读 …… 46
习题 …… 47
参考文献 …… 48
第 3 章　软件过程 …… 50
3.1　统一过程 …… 51
3.2　面向对象范型内的迭代和递增 …… 52
3.3　需求流 …… 53
3.4　分析流 …… 54
3.5　设计流 …… 56
3.6　实现流 …… 57
3.7　测试流 …… 58
3.7.1　需求制品 …… 58
3.7.2　分析制品 …… 58
3.7.3　设计制品 …… 58
3.7.4　实现制品 …… 59
3.8　交付后维护 …… 60
3.9　退役 …… 61
3.10　统一过程的各阶段 …… 61
3.10.1　开始阶段 …… 62
3.10.2　细化阶段 …… 63
3.10.3　构建阶段 …… 64
3.10.4　转换阶段 …… 64
3.11　一维与二维生命周期模型 …… 65
3.12　改进软件过程 …… 66
3.13　能力成熟度模型 …… 66
3.14　软件过程改进方面的其他努力 …… 69
3.15　软件过程改进的代价和收益 …… 70
本章回顾 …… 71
进一步阅读 …… 72
习题 …… 72

参考文献 …… 73
第4章 软件小组 …… 76
4.1 小组组织 …… 76
4.2 民主小组方法 …… 77
4.3 传统的主程序员小组方法 …… 78
4.3.1 纽约时报项目 …… 80
4.3.2 传统的主程序员小组方法的不实用性 …… 80
4.4 主程序员小组和民主小组之外的编程小组 …… 81
4.5 同步－稳定小组 …… 82
4.6 极限编程小组 …… 83
4.7 人员能力成熟度模型 …… 84
4.8 选择合适的小组组织 …… 84
本章回顾 …… 85
进一步阅读 …… 85
习题 …… 85
参考文献 …… 86
第5章 软件工程工具 …… 88
5.1 逐步求精法 …… 88
5.2 成本－效益分析法 …… 92
5.3 软件度量 …… 94
5.4 CASE …… 95
5.5 CASE的分类 …… 95
5.6 CASE的范围 …… 97
5.7 软件版本 …… 100
5.7.1 修订版 …… 100
5.7.2 变种版 …… 100
5.8 配置控制 …… 101
5.8.1 交付后维护期间的配置控制 …… 102
5.8.2 基准 …… 103
5.8.3 产品开发过程中的配置控制 …… 103
5.9 建造工具 …… 104
5.10 使用CASE技术提高生产力 …… 104
本章回顾 …… 105
进一步阅读 …… 105
习题 …… 106
参考文献 …… 107
第6章 测试 …… 109
6.1 质量问题 …… 110
6.1.1 软件质量保证 …… 110
6.1.2 管理独立 …… 111
6.2 非执行测试 …… 111
6.2.1 走查 …… 111
6.2.2 管理走查 …… 112
6.2.3 审查 …… 113
6.2.4 审查与走查的对比 …… 114
6.2.5 评审的优缺点 …… 115
6.2.6 审查的度量 …… 115
6.3 执行测试 …… 115
6.4 应该测试什么 …… 115
6.4.1 实用性 …… 116
6.4.2 可靠性 …… 117
6.4.3 健壮性 …… 117
6.4.4 性能 …… 117
6.4.5 正确性 …… 118
6.5 测试与正确性证明 …… 119
6.5.1 正确性证明的例子 …… 119
6.5.2 正确性证明小型实例研究 …… 121
6.5.3 正确性证明和软件工程 …… 122
6.6 谁应当完成执行测试 …… 124
6.7 测试什么时候停止 …… 125
本章回顾 …… 126
进一步阅读 …… 126
习题 …… 126
参考文献 …… 128
第7章 从模块到对象 …… 130
7.1 什么是模块 …… 130
7.2 内聚 …… 133
7.2.1 偶然性内聚 …… 133
7.2.2 逻辑性内聚 …… 133
7.2.3 时间性内聚 …… 134
7.2.4 过程性内聚 …… 135
7.2.5 通信性内聚 …… 135
7.2.6 功能性内聚 …… 135
7.2.7 信息性内聚 …… 136
7.2.8 内聚示例 …… 136
7.3 耦合 …… 137
7.3.1 内容耦合 …… 137
7.3.2 共用耦合 …… 138
7.3.3 控制耦合 …… 139
7.3.4 印记耦合 …… 140
7.3.5 数据耦合 …… 141
7.3.6 耦合示例 …… 141
7.3.7 耦合的重要性 …… 142

7.4 数据封装 …… 143
7.4.1 数据封装和产品开发 …… 144
7.4.2 数据封装和产品维护 …… 146
7.5 抽象数据类型 …… 150
7.6 信息隐藏 …… 152
7.7 对象 …… 154
7.8 继承、多态和动态绑定 …… 156
7.9 面向对象范型 …… 158
本章回顾 …… 161
进一步阅读 …… 161
习题 …… 161
参考文献 …… 162
第 8 章 可重用性和可移植性 …… 165
8.1 重用的概念 …… 165
8.2 重用的障碍 …… 167
8.3 重用实例研究 …… 168
8.3.1 Raytheon 导弹系统部 …… 168
8.3.2 欧洲航天局 …… 169
8.4 对象和重用 …… 170
8.5 设计和实现期间的重用 …… 170
8.5.1 设计重用 …… 170
8.5.2 应用框架 …… 171
8.5.3 设计模式 …… 172
8.5.4 软件体系结构 …… 175
8.5.5 基于组件的软件工程 …… 176
8.6 重用和交付后维护 …… 176
8.7 可移植性 …… 177
8.7.1 硬件的不兼容性 …… 177
8.7.2 操作系统的不兼容性 …… 178
8.7.3 数值计算软件的不兼容性 …… 178
8.7.4 编译器的不兼容性 …… 180
8.8 为什么需要可移植性 …… 182
8.9 实现可移植性的技术 …… 183
8.9.1 可移植的系统软件 …… 183
8.9.2 可移植的应用软件 …… 183
8.9.3 可移植的数据 …… 184
本章回顾 …… 185
进一步阅读 …… 185
习题 …… 186
参考文献 …… 188
第 9 章 计划和估算 …… 192
9.1 计划和软件过程 …… 192
9.2 周期和成本估算 …… 193
9.2.1 产品规模的度量 …… 194
9.2.2 成本估算技术 …… 197
9.2.3 中间 COCOMO …… 199
9.2.4 COCOMO II …… 202
9.2.5 跟踪周期和成本估算 …… 202
9.3 软件项目管理计划的组成 …… 203
9.4 软件项目管理计划框架 …… 204
9.5 IEEE 软件项目管理计划 …… 205
9.6 计划测试 …… 207
9.7 计划面向对象的项目 …… 208
9.8 培训需求 …… 208
9.9 文档标准 …… 209
9.10 用于计划和估算的 CASE 工具 …… 209
9.11 测试软件项目管理计划 …… 210
本章回顾 …… 210
进一步阅读 …… 210
习题 …… 211
参考文献 …… 212

## 第二部分 软件生命周期的各个阶段

第 10 章 需求 …… 216
10.1 确定客户需要什么 …… 216
10.2 需求阶段概述 …… 217
10.3 理解应用域 …… 217
10.4 商业模型 …… 218
10.4.1 访谈 …… 218
10.4.2 其他技术 …… 219
10.4.3 用例 …… 219
10.5 初始需求 …… 221
10.6 对应用领域的初始理解：Osbert Oglesby 实例研究 …… 221
10.7 初始商业模型：Osbert Oglesby 实例研究 …… 222
10.8 初始需求：Osbert Oglesby 实例研究 …… 224
10.9 继续需求阶段：Osbert Oglesby 实例研究 …… 225
10.10 测试阶段：Osbert Oglesby 实例研究 …… 230
10.11 传统的需求阶段 …… 231
10.12 快速原型开发 …… 231
10.13 人的因素 …… 233

10.14 重用快速原型 …… 234
10.15 需求流的 CASE 工具 …… 235
10.16 需求阶段的度量 …… 235
10.17 需求阶段面临的挑战 …… 235
本章回顾 …… 237
进一步阅读 …… 237
习题 …… 237
参考文献 …… 238
第 11 章 传统的分析 …… 240
11.1 规格说明文档 …… 240
11.2 非形式化规格说明 …… 241
11.3 结构化系统分析 …… 243
11.4 结构化系统分析：Osbert Oglesby 实例研究 …… 248
11.5 其他半形式化技术 …… 249
11.6 建造实体－关系模型 …… 250
11.7 有穷状态机 …… 251
11.8 Petri 网 …… 257
11.9 Z …… 260
11.9.1 Z：电梯问题实例研究 …… 261
11.9.2 Z 的分析 …… 262
11.10 其他的形式化技术 …… 263
11.11 传统分析技术的比较 …… 264
11.12 在传统分析阶段测试 …… 265
11.13 传统分析阶段的 CASE 工具 …… 265
11.14 传统分析阶段的度量 …… 266
11.15 软件项目管理计划：Osbert Oglesby 实例研究 …… 266
11.16 传统分析阶段面临的挑战 …… 267
本章回顾 …… 267
进一步阅读 …… 267
习题 …… 268
参考文献 …… 270
第 12 章 面向对象分析 …… 274
12.1 分析流 …… 275
12.2 抽象实体类 …… 275
12.3 面向对象分析：电梯问题实例研究 …… 275
12.4 功能建模：电梯问题实例研究 …… 276
12.5 实体类建模：电梯问题实例研究 …… 277
12.5.1 名词抽象 …… 278
12.5.2 CRC 卡片 …… 279
12.6 动态建模：电梯问题实例研究 …… 280
12.7 测试流：面向对象分析 …… 282
12.8 抽象边界类和控制类 …… 285
12.9 初始功能模型：Osbert Oglesby 实例研究 …… 285
12.10 初始类图：Osbert Oglesby 实例研究 …… 287
12.11 初始动态模型：Osbert Oglesby 实例研究 …… 292
12.12 抽象边界类：Osbert Oglesby 实例研究 …… 293
12.13 抽象控制类：Osbert Oglesby 实例研究 …… 294
12.14 求精用例：Osbert Oglesby 实例研究 …… 294
12.15 用例实现：Osbert Oglesby 实例研究 …… 296
12.15.1 Buy a Masterpiece 用例 …… 296
12.15.2 Buy a Masterwork 用例 …… 300
12.15.3 Buy Other Painting 用例 …… 301
12.15.4 其余 5 个用例 …… 302
12.16 类图递增：Osbert Oglesby 实例研究 …… 304
12.17 测试流：Osbert Oglesby 实例研究 …… 305
12.18 统一过程中的规格说明文档 …… 305
12.19 关于动作者和用例更详细的内容 …… 306
12.20 用于面向对象分析阶段的 CASE 工具 …… 307
12.21 面向对象分析阶段所面临的问题 …… 307
本章回顾 …… 308
进一步阅读 …… 308
习题 …… 309
参考文献 …… 310
第 13 章 设计 …… 312
13.1 设计和抽象 …… 312
13.2 面向操作设计 …… 313
13.3 数据流分析 …… 313
13.3.1 小型实例研究：字数统计 …… 314
13.3.2 数据流分析扩展 …… 318

13.4 事务分析 …… 319
13.5 面向数据设计 …… 320
13.6 面向对象设计 …… 321
13.7 面向对象设计：电梯问题实例研究 …… 322
13.8 面向对象设计：Osbert Oglesby实例研究 …… 324
13.9 设计流 …… 328
13.10 测试流：设计 …… 329
13.11 测试流：Osbert Oglesby实例研究 …… 329
13.12 详细设计的形式化技术 …… 329
13.13 实时设计技术 …… 330
13.14 设计阶段的 CASE 工具 …… 331
13.15 设计阶段的度量 …… 331
13.16 设计阶段面临的挑战 …… 332
本章回顾 …… 333
进一步阅读 …… 333
习题 …… 334
参考文献 …… 334
第 14 章 实现 …… 337
14.1 编程语言的选择 …… 337
14.2 第四代语言 …… 339
14.3 良好的编程实践 …… 341
14.3.1 使用一致和有意义的变量名 … 341
14.3.2 自文档代码的问题 …… 342
14.3.3 使用参数 …… 344
14.3.4 为增加可读性的代码编排 …… 344
14.3.5 嵌套的 if 语句 …… 344
14.4 编码标准 …… 345
14.5 代码重用 …… 346
14.6 集成 …… 346
14.6.1 自顶向下的集成 …… 347
14.6.2 自底向上的集成 …… 348
14.6.3 三明治集成 …… 349
14.6.4 面向对象产品的集成 …… 350
14.6.5 集成的管理 …… 350
14.7 实现阶段 …… 351
14.8 实现阶段：Osbert Oglesby实例研究 …… 351
14.9 测试阶段：实现 …… 351
14.10 测试用例选择 …… 351
14.10.1 规格说明测试与代码测试 …… 352
14.10.2 规格说明测试的可行性 …… 352
14.10.3 代码测试的可行性 …… 352
14.11 黑盒单元测试技术 …… 354
14.11.1 等价测试和边界值测试 …… 355
14.11.2 功能测试 …… 356
14.12 黑盒测试用例：Osbert Oglesby实例研究 …… 357
14.13 玻璃盒单元测试技术 …… 360
14.13.1 结构测试：语句、分支和路径覆盖 …… 360
14.13.2 复杂性度量 …… 361
14.14 代码走查和审查 …… 362
14.15 单元测试技术的比较 …… 362
14.16 净室 …… 362
14.17 测试对象时潜在的问题 …… 363
14.18 单元测试的管理方面 …… 365
14.19 何时该重写而不是调试一个代码模块 …… 366
14.20 集成测试 …… 367
14.21 产品测试 …… 367
14.22 验收测试 …… 368
14.23 测试流：Osbert Oglesby 实例研究 …… 368
14.24 实现的 CASE 工具 …… 369
14.24.1 软件开发全过程的 CASE 工具 …… 369
14.24.2 集成化开发环境 …… 369
14.24.3 商业应用软件开发环境 …… 370
14.24.4 公共工具基础结构 …… 370
14.24.5 开发环境的潜在问题 …… 371
14.25 实现流的度量 …… 371
14.26 实现流面临的挑战 …… 372
本章回顾 …… 372
进一步阅读 …… 373
习题 …… 374
参考文献 …… 376
第 15 章 交付后维护 …… 379
15.1 为什么交付后维护是必要的 …… 379
15.2 对交付后维护程序员的要求是什么 …… 379
15.3 交付后维护小型实例研究 …… 381
15.4 交付后维护的管理 …… 382
15.4.1 缺陷报告 …… 382

15.4.2 批准对产品的修改 …… 383
15.4.3 确保可维护性 …… 384
15.4.4 反复维护造成的问题 …… 384
15.5 面向对象软件的维护 …… 384
15.6 交付后维护技能与开发技能 …… 387
15.7 逆向工程 …… 387
15.8 交付后维护阶段的测试 …… 388
15.9 交付后维护的 CASE 工具 …… 388
15.10 交付后维护的度量 …… 389
15.11 交付后维护：Osbert Oglesby 实例研究 …… 389
15.12 交付后维护阶段面临的挑战 …… 389
本章小结 …… 389
进一步阅读 …… 390
习题 …… 390
参考文献 …… 391
第 16 章 UML 的进一步讨论 …… 393
16.1 UML 不是一种方法 …… 393
16.2 类图 …… 394
16.2.1 聚合 …… 395
16.2.2 多态 …… 395
16.2.3 组合 …… 396
16.2.4 泛化 …… 397
16.2.5 关联 …… 397
16.3 注解 …… 398
16.4 用例图 …… 398
16.5 模板 …… 398
16.6 交互图 …… 399
16.7 状态图 …… 401
16.8 活动图 …… 403
16.9 包 …… 404
16.10 组件图 …… 404
16.11 配置图 …… 404
16.12 UML 图回顾 …… 404
16.13 UML 和迭代 …… 405
本章回顾 …… 405
进一步阅读 …… 405
习题 …… 405
参考文献 …… 406

## 附 录

附录 A 学期项目：Amlet 沙漠中 Ophelia 的 Oasis …… 407
附录 B 软件工程资源 …… 409
附录 C 需求流：Osbert Oglesby 实例研究 …… 411
附录 D 结构化系统分析：Osbert Oglesby 实例研究 …… 412
附录 E 分析流：Osbert Oglesby 实例研究 …… 415
附录 F 软件项目管理计划：Osbert Oglesby 实例研究 …… 416
附录 G 设计流：Osbert Oglesby 实例研究 …… 419
附录 H 实现流：Osbert Oglesby 实例研究(C++ 版本) …… 425
附录 I 实现流：Osbert Oglesby 实例研究(Java 版本) …… 426
附录 J 测试流：Osbert Oglesby 实例研究 …… 427

# 第一部分　软件工程概述

本书的前9章担负着双重任务：向读者介绍软件过程，并为本书后半部分的内容打下基础。本书后半部分描述了软件开发的工作流（活动）。

软件过程是我们生产软件的方式。它开始于对概念的探讨，结束于产品最终退役的时候。在这段时间里，产品会经历一系列步骤，例如需求、分析（规格说明）、设计、实现、集成、交付后维护和最终的退役。软件过程不仅包括我们用来开发和维护软件所使用的工具和技术，也包括投入的软件专业人员。

第1章“软件工程的范畴”指出，软件生产所使用的技术必须提高成本效率，并且能够使软件生产小组成员间相互促进。从本章开始，全书始终强调贯彻这个目标的重要性。

第2章“软件生命周期模型”中详细讨论了各种不同的软件生命周期模型。包括：进化树模型，瀑布模型，快速原型开发模型，极限编程，同步－稳定模型，螺旋模型，以及最重要的迭代－递增模型。为了使读者能够针对某一具体设计选用合适的生命周期模型，该章对各种生命周期模型进行了对比。

“软件过程”是本书第3章的标题。这一章重点强调的是统一过程这一目前最有前途的软件开发方法。这一章以软件过程改进方面的内容结束。

第4章标题为“软件小组”。现在的设计项目很大，仅仅凭借一个人的力量是很难在有限的时间内完成的，取而代之的是一组软件专业人员的精诚合作。这一章的主题是讨论应当如何组织一个小组，以便小组成员一同高效协作。该章讨论了各种不同的组织团队的方法，包括民主小组、主程序员小组、同步－稳定小组和极限编程小组。

第5章讨论了“软件工程工具”，软件工程师需要使用不同的工具，包括分析工具和实用工具。该章介绍了不同的软件工程工具，其中一个工具是逐步求精（stepwise refinement），它是一种把大的问题分解成小的、容易处理的问题的技术。另一个工具是成本－效益分析，它是一种判断软件项目经济可行性的工具。然后，该章对计算机辅助软件工程（computer-aided software engineering，CASE）工具进行了描述，CASE工具是一种软件产品，辅助软件工程师开发和维护软件。最后，为了管理软件过程，对软件过程进行量化测量从而判断该软件是否偏离了正常轨道很必要。这些测量（度量）对项目的成功很关键。

在第10～15章中，对第5章的最后两个主题——CASE工具和度量进行了深入的讨论，对软件生命周期具体的工作流进行了详细描述，对支持每个工作流的CASE工具进行了讨论，同时也对充分管理该工作流所需的度量做了描述。

第6章“测试”对测试中蕴含的概念进行了讨论。对软件生命周期的每个工作流所使用的具体软件测试技术的考虑留在第10～15章讨论。

第7章“从模块到对象”详细解释了类和对象，并说明为什么面向对象范型比结构化范型证明是更成功的。本书的其余部分都用到了这一章中的概念，特别是第10章“需求”、第

12 章“面向对象分析”和第 13 章“设计”（这一章中给出了面向对象设计）。

第 7 章的思想在第 8 章“可重用性和可移植性”中得到了扩展，编写可移植到各种不同的硬件的可重用软件是非常重要的。该章的第一部分讲的是重用，内容包括多种重用实例研究，以及一些重用策略，如面向对象模式和框架；可移植性是第二个主要内容，这一章在某种程度上深入地介绍了可移植性策略。本章反复出现的一个主题是对象在获取可重用性和可移植性时所起的作用。

第一部分的最后一章是第 9 章“计划和估算”。在开始一个软件设计项目之前，对整个行动做一个详细计划是最基本的。项目一旦开始，管理者必须紧密监督进度，注意是否偏离了计划并且在必要的时候采取行动纠正。同样，向客户提供项目时间和金钱耗费的准确估算也是很重要的。这里对不同的估算技术进行了描述，包括功能点（function point）和 COCOMO II。这里给出了软件项目管理计划的详细描述，第 11 章和第 12 章使用了该章的材料。当使用传统范型时，主要的计划和估算活动发生在传统分析阶段结束的时候，第 11 章中对此有说明。当使用面向对象范型开发软件时，这个计划发生在面向对象分析工作流（第 12 章）结束时。

# 第 1 章　软件工程的范畴

**学习目标**

学完本章之后，你应当能够：

- 明确软件工程意味着什么；
- 描述传统软件工程生命周期模型；
- 解释为什么面向对象范型现在被广泛接受；
- 讨论软件工程的各个方面的含义；
- 区分维护的传统和现代的观点；
- 讨论连续的计划、测试和文档编制的重要性；
- 感受遵守道德规范的重要性。

一个有名的故事讲的是一个主管有一天收到了一份计算机生成的账单，账单的金额为 0.00 美元，这人与朋友一起大大地嘲笑了一下“愚蠢的计算机”，然后将账单扔掉了。一个月之后，收到了一份同样的账单，上面标示着已过了 30 天。然后，第三张账单来了。又一个月之后，第四张账单也来了，同时带来一个通知，暗示说如果不能及时付清这个 0.00 美元的账单将可能会采取法律行动。

第五张账单来的时候标示着时间已过了 120 天，这张账单没有任何暗示，它粗鲁、直白地威胁道：这张账单如不能立刻付清将要采取所有法律手段。这个主管担心自己公司的贷款（信用）利率会受到这个疯狂的机器的影响，于是找了一个软件工程师朋友，跟他讲了这件恼人的事情。这位工程师克制住不让自己发笑，他让主管邮了一张 0.00 美元的支票。该做法取得了预期的效果，几天后一张 0.00 美元的收据寄到了，这个主管小心翼翼地收好了这张收据以防将来计算机声称那张 0.00 美元的账单还没有支付。

这个有名的故事有一个不太为人知晓的结局，几天后这个主管被他的财务经理叫去了，这个财务经理拿着一张 0.00 美元的支票问“这是你的支票吗?”

这个主管回答说：“是。”

“那你能告诉我为什么要写一张 0.00 美元的支票吗?”财务经理问。

这样，整个故事又被重复一遍。当这个主管说完的时候，经理转向他，安静地问“你能说说你付 0.00 美元的账单对我们的计算机系统有什么影响吗?”

计算机专业人士虽然会感到一点点窘迫，但还是会对此故事感到可笑。毕竟在我们任何一个人设计或完成一个产品的初样阶段，是可能出现类似催讨 0.00 美元账单的情况的。到目前为止，我们在测试中总是能够发现此类错误。但是我们的笑声中有一种恐惧感，因为我们内心深处担心的是，有一天，在我们已经将产品交付给顾客的时候还没有发现这些问题。

1979 年 11 月 9 日，检测到一个绝对称不上幽默的软件错误。美国战略防空司令部收到由全球军事指挥控制系统（WWMCCS）计算机网络发出的警报，引起了混乱，警报报告苏联已经向美国发射导弹［Neumann 1980］。这实际上是模拟演习被当成了真的，就像 5 年后电影《War Games》里所演的那样。虽然美国国防部可以理解地没有详细说明把实验数据当成真实数据的确切理由，但看起来很可能是软件错误导致的；或者是系统作为一个整体在设

计上没有区分模拟和真实；或者是用户界面没有设置必要的检查来确保系统的终端用户从虚假中分辨出事实。换句话说，软件错误，如果该问题确实是由软件引起的，可能会给我们的文明社会带来不愉快和灾难性的结局（有关由软件错误所导致的灾难方面的信息，请见“如果你想知道”部分）。

无论我们正在处理的是账单还是防空任务，我们的许多软件都推迟交付时间、超出预算、带有残存的错误，并且不满足用户要求。软件工程试图解决这些问题，换言之，**软件工程**是一门学科，它的目的是生产出没有错误的软件，按时并且在预算内交付，而且能够满足用户的需求。更进一步，当用户的需求改变时，软件必须易于修改。

软件工程的范畴非常广。软件工程的某些方面可以归入数学或计算机科学；其他方面可以落入经济学、管理学或哲学的范畴。为了展示软件工程所触及的宽广领域，我们将从五个方面来进行考查。

**如果你想知道**

在WWMCCS的网络的情形下，灾难在最后几分钟内得以避免。然而，有些软件错误却会导致悲剧。例如，在1985年和1987年之间，至少有两个病人死于由Therac-25医用线性加速器产生的严重过量辐射[Leveson and Turner，1993]，原因是控制软件中的一个错误。

在1991年的海湾战争中，一枚飞毛腿导弹穿越了爱国者反导弹防御措施，击中了沙特阿拉伯的Dhahran附近的一个兵营，导致28名美国人死亡，98人受伤。爱国者导弹的软件中包含了一个累积的定时错误，爱国者导弹设计成每次只能工作几个小时，过了这个时间之后，时钟就要复位。结果是，这个错误从未有过明显的影响，从而就没有被检测到。然而，在海湾战争中，爱国者导弹的电池在Dhahran连续工作了100小时以上，这引起的累积时间误差大得足以导致系统的不精确。

在海湾战争中，美国用船运送爱国者导弹到以色列，以保护其免受飞毛腿导弹的攻击。以色列的军队仅在8个小时后就察觉到了这个定时问题，并立刻向美国的制造商报告了这个问题，制造商尽快地纠正了这个问题。然而悲惨的是，新软件在飞毛腿导弹直接击中兵营的第二天才到达[Mellor，1994]。

幸运的是，由软件错误造成的死亡或严重伤害非常少。然而，一个错误可能会给成千上万的人带来重大影响。例如，在2003年2月，一个软件错误使美国财政部发出50 000张没有受益者名字的社会保险支票，因此这些支票无法储蓄或兑现[St. Petersburg Times Online，2003]。在2003年4月，借款者被SLM公司（通常称为Sallie Mae公司）告知，他们的助学贷款由于计算机软件问题从1992年起就被计算错了，该问题直至2002年底才被检测出来。大约100万借款人被告知他们需要支付更多的还款，从而在形式上，或者每月支付更多的还款额，或者在当初10年的还款期限之外，再继续花时间支付多出来的还款利息[GJSentinel.com，2003]。这两个错误都很快被纠正了，但是它们对大约100万人产生了很大的经济上的影响。

## 1.1　历史方面

发电机会出现问题，这是事实，但是它比工资报表软件产品出问题的几率小得多；桥梁有时候会倒塌，但是比操作系统崩溃的可能性小得多。在软件设计、实现和维护应当与传统

的工程学科具有同等地位的观念驱使下，一个 NATO 研究小组在 1967 年创造了软件工程（software engineering）的概念。软件的开发应当同其他工程任务的开发相类似，这一声明在 1968 年于德国的 Garmisch 召开的 NATO 软件工程会议上得到了签署［Naur，Randell，and Buxton，1976］。这项声明的签署并不令人太感到吃惊，会议本身的名字反映了这样一种看法：软件生产应当是一项类似工程的活动。与会人员得出结论：软件工程应当使用已建立的工程学科的基本原理和范型（paradigm，即方法示例——译者注）来解决所谓的“软件危机”（software crisis）；顾名思义，“软件危机”指软件产品的质量低得通常不能接受，并且不能满足交付日期和预算限制。

尽管有许多成功软件的故事，但仍有相当数量的软件产品延期交付、超出预算，并且在其中存在错误。例如，Standish Group 是一个分析软件开发项目的研究机构。它们对 2000 年完成的 280 000 个软件开发项目进行了分析，由 Johnson、Boucher、Connors 和 Robinson［2001］给出了报告，这个报告概括在图 1-1 中。从图中可看出，仅有 28% 的项目是成功完成的；有 23% 的项目在完成之前被取消了或者永远不会实现；余下的 49% 的项目得以完成并安装在客户计算机上，然而，这些项目超出了预算、延期交付或比最初规定的少了一些特性和功能。换句话说，在 2000 年里，只有 1/4 的软件开发项目是成功的，3/4 的软件开发项目显示出软件危机的一个或多个征兆。

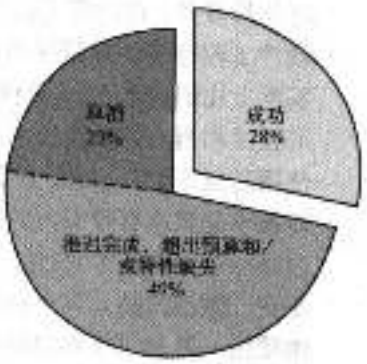



图 1-1 2000 年完成的 280 000 个软件开发项目的结果统计［Johnson et al.，2001］

软件危机带来的经济上的影响非常可怕。在一份由 Cutter Consortium［2002］做出的统计调查报告中，报告了以下情况：

- 有令人吃惊的 78% 的信息技术组织卷入到纠纷中并最终以诉讼方式终止。
- 在 67% 的这些事例中，交付的软件产品的性能或功能没有达到软件开发者所声称的程度。
- 在 56% 的这些事例中，承诺的交付日期几次推迟。
- 在 45% 的这些事例中，软件中的差错非常严重，以致软件产品无法使用。

显然，只有很少的软件产品能够及时、不超出预算、无差错地交付，并且满足客户的需求。为了达到这些目标，软件工程师需要获得广泛的技巧，既有技术的也有管理的。这些技巧不只要应用到编程，还要应用到软件生产的每一个步骤，从需求分析到交付后的产品维护。

在 35 年之后软件危机仍然伴随着我们，这告诉我们两件事情：首先，软件生产过程虽然在许多方面与传统工程是相似的，但仍然有自己的属性和问题；第二，考虑到软件危机的周期长且难预测，可能应当将软件危机重新命名为**软件萧条**（software depression）。

我们现在来看软件工程的经济方面。

## 1.2 经济方面

使用旧编码技术 $CT_{old}$ 的软件组织发现使用新的编码技术 $CT_{new}$ 后，编写代码的时间比使

用旧的编码技术要少花 1/10 的时间，因此，花费也要少 1/10。通常大家会认为使用新技术比较恰当。实际上，虽然大家普遍认为速度快的技术应当成为技术的首选，但是从软件工程的经济观点看却得出了相反的结论。

- 原因之一是将新技术引入到一个软件组织的花费。使用 $CT_{new}$ 技术后编码速度提高了 10%，这与将新技术引入到开发组织中的花费相比，前者不如后者那么大。培训费用可能需要用完成两到三个项目来弥补。并且，参加新技术 $CT_{new}$ 培训的阶段，软件人员不能够从事生产工作，甚至当他们培训结束后，还要有一个艰难的学习过程。要使软件专家们像熟悉旧技术 $CT_{old}$ 那样熟悉新技术 $CT_{new}$，需要花费几个月的时间去实践。这样，从一开始就用新技术 $CT_{new}$ 去开发项目所花的时间，比继续使用旧技术 $CT_{old}$ 的时间长得多。在决定是否需要使用新技术 $CT_{new}$ 的时候，所有这些花费都得考虑进去。
- 软件工程经济学建议保留旧技术 $CT_{old}$ 的第二个重要原因是维护问题。新技术 $CT_{new}$ 实际上比旧技术 $CT_{old}$ 要快 10%，并且从满足用户当前需求的角度来说，代码质量与旧技术相当。但是新技术 $CT_{new}$ 的使用导致代码很难维护，从整个产品的周期来看，使用新技术 $CT_{new}$ 的耗费要大一些。当然，如果软件开发者不用负责维护，那么使用新技术是非常有吸引力的一种建议。毕竟使用新技术的花费要少 10%。软件客户应当坚持使用旧技术 $CT_{old}$，并给予较高的前期投入，希望减少软件的整个生命周期的花费。遗憾的是，客户和软件提供者的惟一目标是尽可能快地生产代码，他们在考虑使用一种专门技术的短期效应时通常会忽略它的长期效应。将经济性原理应用于软件工程要求客户选择能够产生长期效应的技术。

这个例子从新代码对软件开发效果的贡献只有 10% 的角度讨论了编码的问题。可是，经济性原则也适用于软件生产的其他方面。

现在，我们来讨论可维护性的重要性。

## 1.3　维护性方面

在这一节中，我们在软件生命周期的范畴内描述维护性。**生命周期模型**是对在构建一个软件产品时应当完成的步骤的描述。已经提出许多不同的生命周期模型，第 2 章中描述了其中的几个。因为执行一系列较小的任务总是比执行一个大的任务要容易，因而将整个生命周期模型划分为一系列较小的步骤，称为**阶段**。阶段的数量因模型的不同而不同——从少的只有 4 个到多的有 8 个。生命周期模型，是一个应当做什么的理论上的描述；与此相对照的是，对某一个具体的软件产品所做的一系列实际步骤，从概念开发到最终退役，称为该产品的**生命周期**。在实践中，一个软件产品的生命周期的各阶段无法严格如生命周期模型中规定的那样完成，特别是当碰上时间和花费超支情况时。有人指出，软件项目由于时间原因出现问题的情况比所有其他原因加起来还要多［Brooks，1975］。



图 1-2　传统生命周期模型的 6 个阶段

直到 20 世纪 70 年代末，大多数软件组织都使用一种称为**瀑布模型**（waterfall model）的生命周期模型进行软件开发。这个模型的变种有许多，但使用这种传统的生命周期模型开发的软件基本上经历如图 1-2 所示的 6 个阶段。这些阶段可能跟某一个特定软件组织的开发阶段不完

全吻合，但是从本书的目的而言，它们与大多数具体开发实践非常接近。同样地，每个开发阶段的准确名称在各个软件组织之间有很大的不同。本书为不同阶段所选择的名字做到了尽量具有通用性，以使读者能够感到比较方便。

1）需求阶段。对概念进行研究和细化，提取客户的需求。

2）分析（规格说明）阶段。分析客户需求并以**规格说明文档**——“期望产品做什么”的形式给出，分析阶段有时称为**规格说明阶段**（specification phase）。在该阶段结束的时候，制定出一个计划，称为**软件项目管理计划**（software project management plan），该计划详细描述期望的软件开发。

3）设计阶段。在设计阶段，规格说明经过两个连续的设计过程。第一个是**结构设计**，将作为一个整体的产品分解成各个部分，称为**模块**，然后再对每个模块进行设计，这个过程称为**详细设计**；得到的两个**设计文档**描述了“产品是如何做的”。

4）实现阶段。对各个部分独立地进行代码编写和测试（**单元测试**），然后，将该产品的各部分组合起来并作为一个整体进行测试，这称为**集成**。当开发人员对该产品正确地完成功能感到满意时，由客户对该产品进行测试（**验收测试**）。当客户接受该产品并将它安装到客户的计算机中时，实现阶段就结束了（我们将会在第 14 章看到编码和集成应当并行进行）。

5）交付后维护。产品是用来完成设计它所要完成的任务的，在这期间，需要对其进行维护。交付后维护包括在产品交付并安装到客户计算机中并通过验收测试后对产品所做的全部改动。维护包括**纠错性维护**（或**软件修复**）和**增强性维护**（或软件改进）。纠错性维护主要是去掉残存错误，它不对规格说明文档做修改；增强性维护则是在对规格说明文档进行修改的同时，实现这些修改。有两种类型的增强性维护：第一种是**完善性维护**，客户对产品所做的改变将提高产品的性能，如附加功能或减少响应时间；第二种是**适应性维护**，这种维护对程序进行修改以响应程序运行环境的变化，比如新的硬件/操作系统或新的政府制度规定。（要深入理解这三种类型的交付后维护，请见“如果你想知道”部分。）

6）退役。退役发生在当产品退出服务的时候，当产品功能不再对客户组织有用的时候，就不再使用该产品。

**如果你想知道**

软件工程领域最广泛引用的一个研究结果是，实际上，维护人员大约花 17.4% 的交付后维护时间在纠错性维护上，大约 18.2% 的时间在适应性维护上，大约 60.3% 的时间在完善性维护上，另有 4.1% 的时间可归为“其他”类。这个结果来自 1978 年出版的一篇论文[Lientz，Swanson，and Tompkins，1978]。

然而，该篇文章的结果不是来自对维护数据的测算，作者进行的是对维护经理的调查，请他们估计在他们的组织内总共有多少时间用在每一个维护分类上，并请他们对自己的估计给出一个确认度。更特别的是，参加该项调查的软件维护经理们被问及是否他们的回答所根据的是相当精确的数据、较少数据或无数据。49.3% 的经理人声明他们的回答是基于相当精确的数据，37.7% 的经理人认为是基于较少数据，8.7% 的经理人承认没有数据依据。

事实上，就调查中有关用于各类维护的时间的百分比问题，被问及人应当认真地回答是否他的回答有精确的数据依据，多数人甚至都不应当有较少的数据依据。在该调查中，要求参加者说明在诸如“紧急修补”或“常规调试”等单项中维护的百分比组成是怎样的，从这个原始的信息中，可以推断出适应性维护、纠错性维护和完善性维护各自所占的百分比。在

1978年，软件工程刚刚开始作为一个学科出现，对于接受调查的软件维护管理者来说，搜集调查所需要的详细信息是一件很意外的事。确实，用现代的术语来说，在1978年实际上每个组织仍然处在CMM的第一个阶段（见3.13节）。

因此，我们有充足的理由提出疑问：是否1978年以前实际交付后维护活动的分布数据在某种程度上是参加调查的管理者的估计？现在，维护活动的分布当然不是那种状态了。例如，有关Linux内核［Schach et al.，2002］和gcc编译器［Schach et al.，2003b］的实际维护数据显示：至少50%的交付后维护是纠错性的，这一点与调查中提到的17.4%的数据是不符的。

现在，我们来更详细地讨论一下维护的定义。

### 1.3.1　维护的传统和现代观点

在20世纪70年代，人们将软件生产看作是由两个完全不同的活动顺序组成：首先是开发，然后是维护。人们从最初的构思开始，开发软件产品，然后将它们安装到客户的计算机上。在软件安装在客户的计算机上并验收之后，对软件所做的任何改动，无论是解决一个残留的错误还是扩展软件的功能，都属于传统的维护概念范围内的活动［IEEE 610.12，1990］。因此，传统软件开发方法可以描述为**开发－维护模型**。

这是一个**时间上的定义**，就是说，根据某个软件开发活动在什么时候进行，将其归类为开发还是维护。假设在软件安装后的某一天发现并纠正了一个软件错误，则该软件活动很显然属于维护的范畴，但是，如果该错误是在软件安装之前发现并纠正的，那么根据传统软件工程的定义，这样的软件活动应当属于软件开发范畴。现在假设一个软件产品刚刚安装，但客户想要增加该软件产品的功能。传统上，那应当认为是“完善性维护”，然而，如果客户想要在软件产品安装之前做同一改变，这应当是传统上的开发范畴。由此，这两个活动在本质上没有任何不同，但传统上一个被认为是开发，一个被认为是完善性维护。

除了这种不一致外，还有两个其他原因解释了为什么开发－维护模型在今天是不现实的：

1）现在开发一个软件所使用的时间为期一年或超过一年是一件非常正常的事情，在这个较长的开发期内，客户的需求是很可能发生改变的。例如，某个客户可能会要求在一个刚刚投入使用的速度更快的微处理器上实现该软件产品。或者，在开发正在进行的时候，客户将业务扩展到了比利时，所以就需要对产品进行修改，以使产品能处理在比利时的销售。为了看到此类需求变化如何影响软件生命周期，我们假设在设计正在进行阶段，客户的要求发生了变化。这时，软件工程小组必须停止开发，并修改规格说明文档来反映需求的变化。并且，也必须对设计进行修改。对规格说明的修改使得已经完成的设计部分也相应要进行修改，只有当这些改变都完成了之后，才能继续进行开发。换句话说，开发人员必须在产品安装前很长时间就开始对产品进行“维护”工作。

2）传统开发－维护模型中存在着第二个问题，该问题来源于我们目前所用的软件构造方法。传统软件工程中，软件开发的一个特征是开发小组从零开始开发目标产品。相反，作为今天代价昂贵的软件生产的结果，软件开发者在将要开发的软件中尽量重用已经存在的软件的各个部分（“重用”在第8章详细讨论）。因而，在“重用”被广泛使用的今天，开发－

维护模型已经变得不适用了。

一个更现实的看待维护的方法是在由国际标准化组织（International Organization for Standardization，ISO）和国际电子技术委员会（International Electrotechnical Commission，IEC）发布的生命周期过程标准中给出的，即，维护是一个过程，一个在下述时机所进行的过程："软件因存在问题或因有改进或适应性需求时而对代码或相应文档所进行的修改"[ISO/IEC 12207，1995]。按照这个**操作性定义**，当改正错误或需求变化时，对软件的维护就发生了，而不管它是发生在软件安装前还是在软件安装后。国际电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）和电子工业联合会（Electronic Industries Alliance，EIA）随后认可了这个定义［IEEE/EIA 12207.0-1996，1998］，同时 IEEE 标准做了修改，以与 ISO/IEC 12207 相符合。（见"如果你想知道"部分以了解更多有关 ISO 的情况。）

**如果你想知道**

ISO 是一个由 147 个国家的国家标准协会组成的网络，它的中心秘书处位于瑞士首都日内瓦。ISO 已经发布了超过 13 500 个国际上认可的标准，其标准范围包括从照相胶片速度（"ISO 数字号"）到本书中给出的许多标准。例如，在第 3 章中讨论的 ISO 9000。

ISO 不是一个取首字母的缩写词，它取自希腊字 ισος，意思是"相等"，我们可以在一些词如 isotope，isobar 和 isosceles 中发现英语的构词法前缀 iso-。国际标准化组织选择 ISO 作为它的名字的缩写形式，是为了避免由于其名字"International Organization for Standardization"翻译成不同成员国的文字而产生多个缩写词。为了取得国际上的统一标准，选择了一个它的通用的缩写形式。

在本书中，"交付后维护"一词参照的是 1990 年 IEEE 对于维护的定义，即在软件产品交付给客户并安装在其计算机上之后对软件所做的任何改变。而"现代维护"或仅仅"维护"，参照 1995 年 ISO/IEC 定义的在任何时候进行的纠错性、完善性或适应性活动。因此，交付后维护是（现代）维护的一个子集。

### 1.3.2　交付后维护的重要性

有时人们认为只有坏的软件产品才需要维护，实际上恰恰相反：人们通常会将坏的软件扔掉，而对好的软件在 10 年、15 年甚至 20 年的时间范围内进行改进和提高。进一步来说，软件产品是现实世界的模型，而现实世界是在不断变化着的。从而，必须不断对软件进行维护以保证它能精确地反映现实世界的变化。

例如，如果营业税率从 6%增长到 7%，几乎每个涉及买和卖的软件产品都会发生变化。假设该产品包含下述 C++ 声明语句：

```
const float salesTax = 6.0 ;
```

或同等的 Java 声明语句：

```
public static final float salesTax = (float) 6.0 ;
```

这里声明 salesTax 为浮点常量，并被初始化为 6.0。在这种情况下，维护相对比较简单。可以利用文本编辑器将常量 6.0 替换为常量 7.0，并且对代码进行重新编译和重新链

接。然而，如果产品中不是使用变量名 salesTax，而是在调用营业税值的地方使用实际值 6.0，那么这样一个产品就会非常难于修改。例如，源代码中，应将 6.0 换成 7.0 的地方被忽略了，或者是将不代表营业税的有 6.0 的地方错误地换成了 7.0。找出这些错误很困难并且很耗费时间。事实上，对某些软件来说，放弃该产品并对其重新编码比找出需要变动的常量并对其进行变动的花费要小一些。

实时的现实世界也在不停地变化。喷气式战机装备的导弹可能被一种新的型号取代，它要求对相关航空电子系统的武器控制部分进行某种改变。将提供八缸引擎作为对普通汽车的四缸引擎的选配，这需要改变控制燃料注入系统、定时系统等的车载计算机。

但是应当投入多少时间（=金钱）进行交付后维护呢？图 1-3a 的饼状图显示，大约 30 年前，接近 2/3 的整体软件花费用在了交付后维护上。统计数据取自不同来源的信息，包括 [Elshoff，1976；Daly，1977；Zelkowitz，Shaw，and Gannon，1979；and Boehm，1981]。较新的数据显示，更大的花费比例用于交付后维护。许多组织将其软件预算的 70%～80% 或更多用于交付后维护 [Yourdon，1992；Hatton，1998]，见图 1-3b。

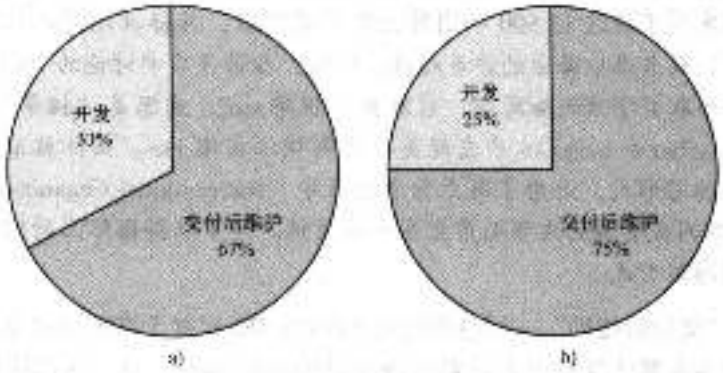



图 1-3　开发和交付后维护平均成本的近似百分比：a) 在 1976 年到 1981 年间；b) 在 1992 年到 1998 年间

令人吃惊的是，传统开发阶段的平均成本百分比已经大大改变了。从表 1-1 可以看出这一点，表中将用来得出图 1-3a 的数据与更新的惠普公司（Hewlett-Packard）的 132 个项目的数据进行了比较 [Grady，1994]。

表 1-1　1976 年到 1981 年间各种项目以及惠普公司 132 个更新的项目在传统开发阶段的近似平均成本百分比比较

| | 1976 年到 1981 年间各种项目 | 惠普公司 132 个更新的项目 |
|---|---|---|
| 需求和分析（规格说明）阶段 | 21% | 18% |
| 设计阶段 | 18% | 19% |
| 实现阶段 | | |
| 编码（包括单元测试） | 36% | 34% |
| 集成阶段 | 24% | 29% |

现在再考虑一下当前正在使用 $CT_{old}$（旧的）编码技术的某个软件组织，该软件组织得知使用 $CT_{new}$（新的软件开发技术）将减少 10% 的编码时间。即使 $CT_{new}$ 对维护并没有什么不好的作用，一个聪明的软件管理者在改变编码技术前也会三思而行的。整个软件开发人员

需重新培训，要购买新的软件开发工具，并且可能要另外雇用精通新技术的雇员。所有这些花费和中断最多只能减少软件成本的 0.85%，因为如图 1-3b 和表 1-1 所示，编码与单元测试平均只占整个软件成本的 25%中的 34%，即约占 8.5%。

现在假设开发出了一种新技术能够减少约 10%的维护成本。这种新技术可能就会被立刻引入，因为，它能减少整体成本的 7.5%。引入这种技术所需要的花费与它所能减少的花费相比要小得多。

交付后维护非常重要，因而它是软件工程的一个重要方面，它由那些可降低软件交付后维护成本的技术、工具和实践组成。

## 1.4　需求、分析和设计方面

软件专业人员也是人，所以他们在开发产品时有时可能会犯一些错误，这样在软件中就会出现一些缺陷。如果这个错误是在需求阶段出现的，那么这个错误会延续到规格说明阶段、设计阶段和编码阶段。很明显，越早纠正错误越好。

在传统的软件生命周期的不同阶段纠正一个错误的相对成本如图 1-4 所示［Boehm 1981］。该图的数据来自 IBM［Fagan，1974］、GTE［Daly，1977］、Safeguard 项目［Stephenson，1976］，还有一些较小的 TRW 项目［Boehm，1980］。图 1-4 中的实线是由与大项目有关的数据拟合而成的曲线，虚线是较小项目的数据拟合曲线。图 1-5 对在传统软件生命周期的每个阶段，发现和纠正一个错误的相对耗费进行了描绘。图 1-5 中的每一级实线都是依据图 1-4 中的每一点而来的，将点延续成实线。

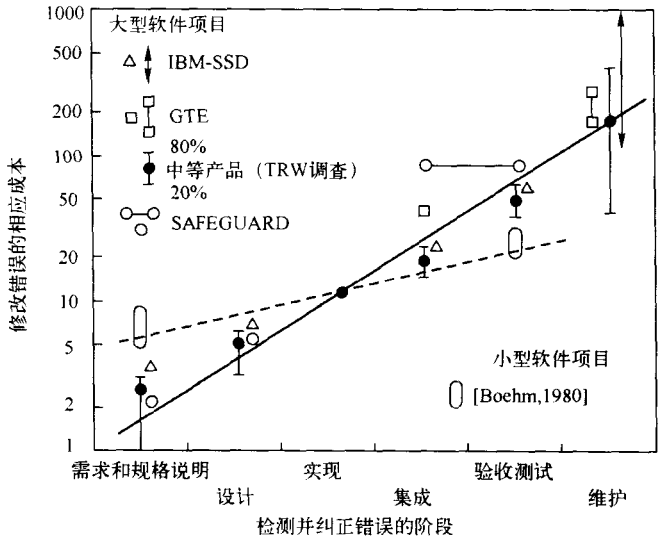



图 1-4　在传统软件生命周期的每个阶段解决一个错误的相对成本。实线是与大项目有关的数据的拟合，虚线是小项目数据的拟合（Barry Boehm，*Software Engineering Economics*，© 1981，p.40. 此次改编得到新泽西州 Prentice-Hall 公司的许可。）

假设在设计阶段发现和纠正一个错误需要花费 40 美元。由图 1-5 中的实线（1974 年到 1980 年间的项目）我们可以看出，在分析阶段纠正相同的错误我们仅需要花费 30 美元。但

是在交付后维护阶段纠正相同的错误却需要花费 2000 美元。新的数据说明现在较早地纠正错误更加重要。图 1-5 中的虚线说明了在 IBM AS/400 项目的系统软件开发中［Kan et al., 1994］，发现和纠正一个错误的成本。平均来说，在 AS/400 软件中，在交付后维护阶段纠正一个错误需要花费 3680 美元。

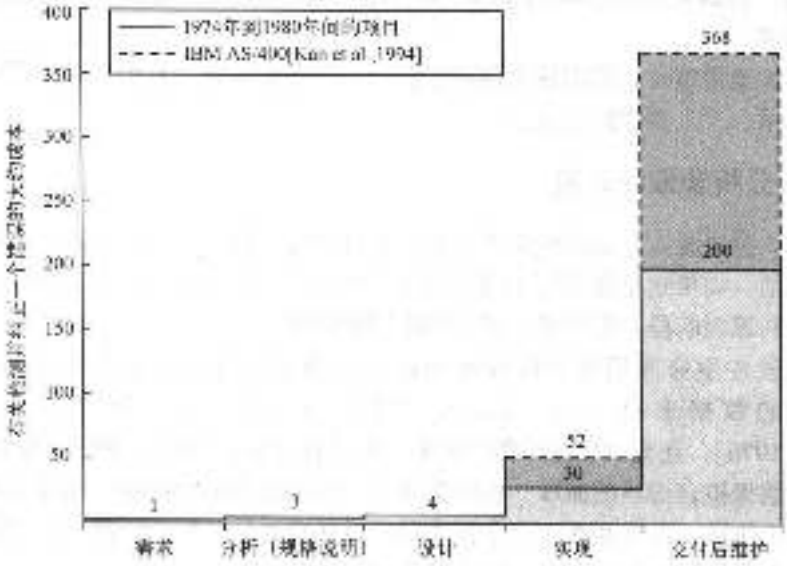



图 1-5　实线描绘了图 1-4 中实线上的点（以线性比例绘制），虚线描绘了较新的数据

纠正一个错误的花费为什么增长得如此迅速，这与纠正错误需要做的事情有关。在开发周期的早期，产品仅仅存在于纸上，在这个阶段纠正错误意味着仅仅对文档进行修改就可以了。另一个极端是产品已经交付给了客户，在这时纠正错误则意味着编辑代码、重新编译和链接，并且仔细验证问题是否得到解决。然后，还有更关键的问题是，检查所做的修改没有在产品中的其他部分产生新的问题。所有的相关文档，包括手册都需要更新。最后还要对改好的产品进行交付和重新安装。这个故事说明我们应当尽早发现错误，否则会付出很大的代价。所以我们应当使用各种技术，在需求阶段和分析（规格说明）阶段发现错误。

人们对于这种发现错误的技术有进一步的需求。研究表明［Boehm, 1979］，在较大开发项目所发现的错误中，60%－70%的错误都是需求、分析或设计错误。最新审查结果也证实：需求、分析或设计错误在错误中占的比例较大（审查是由一个小组对文档所做的仔细检查，见 6.2.3 节）。在对喷气机推动实验室为 NASA（美国国家航空航天局）无人太空计划开发的软件进行的 203 次审查中，发现平均每页规格说明文档有大概 1.9 个错误；每页设计文档中有大概 0.9 个错误；每页代码中只有约 0.3 个错误［Kelly, Sherif, and Hops, 1992］。

因而，提高我们的需求、分析和设计技术是非常重要的。不仅仅因为这样可以尽早发现错误，而且因为需求、分析和设计错误在整个错误数中占了很大比例。恰如前面的例子所说的，降低 10%的交付后维护费用，就可以减少整个成本的 7.5%；减少 10%的需求、分析和设计错误，就可以减少整个错误总数的 6%－7%。

在软件生命周期的早期产生如此多的错误，说明了软件工程的另外一个重要方面：技术

能够产生出更好的需求、规格说明和设计。大多数软件是由一个软件工程师小组开发的，而不是个人来负责开发和维护生命周期的各个方面。现在，我们来讨论这方面的含义。

## 1.5　小组编程方面

硬件的成本在不断地迅速下降。一台 1950 年代的大型计算机耗资超过通货膨胀前的 100 万美元，它在每个方面的性能还远不如今天低于 1000 美元的台式机强大。因此，各个组织都能够买得起运行较大软件产品的硬件，即是说，产品太大了无法由一个人在规定时间内编写完成。例如，一个需要在 18 个月内交付的产品需要耗费一个程序员 15 年的时间去完成，这样，这种产品就需要一个小组去完成。然而，小组编程导致了代码模块间的接口问题和小组成员间的沟通问题。

例如，Jeff 和 Juliet 分别编码模块 p 和 q，模块 p 调用模块 q。当 Jeff 编写模块 p 时，他用含有 5 个参数的参数表调用模块 q。Juliet 也用 5 个参数编写模块 q，但是这 5 个参数的顺序与 Jeff 的顺序不同。一些软件工具，例如 Java 解释器和加载器，或者 C 的 lint（8.7.4 节），仅仅当交换的参数具有不同类型的时候才能发现这样的类型不一致问题。如果参数的类型相同，那么这个问题要很长时间才能够发现。或许有人会认为这是设计问题，如果人们在设计的时候再仔细一点，这种问题或许不会发生。这可能是对的，但是实际上，在编码开始后，设计也常常会发生改变，但是这个改变可能不会通知给开发小组的各个成员。这样，当某个设计变化影响到两个或更多编程人员的时候，如果没有通知到相关人员，就会出现 Jeff 和 Juliet 遇到的问题。当仅由一个人负责某一软件产品的各个方面时，这类问题不太会出现，在能运行大型软件的高速计算机出现之前，也是这种情况。

接口问题仅仅是小组开发软件所出现问题中的一个很小的方面。如果不能对开发小组进行恰当的组织和安排，那么将有大量的时间浪费在小组成员之间的协调上。假设某产品需要一个程序员用一年时间完成，同样的任务分配给一个有三个程序员的开发小组，那么完成该工作的时间常常接近一年，而不是大家所期望的 4 个月。代码完成的质量可能比一个人完成的要低一些（见 4.1 节）。因为现在的软件绝大多数要由小组来开发和维护，所以软件工程的范畴必须包括确保小组恰当管理和组织方面的技术。

如前面各节所述，软件工程的范畴是很大的。它包括软件生命周期的每个步骤，从需求开始一直到交付后退役。它也包括人的方面，例如小组的组织，经济方面，法律方面（例如版权法）。所有这些都隐含在本章开始的软件工程的定义里；即，软件工程是一门学科，该学科的目标是生产出能如期交付、在预算范围内、满足用户需求、没有错误的软件产品。

我们现在回到图 1-2 的传统阶段来，问问它为什么没有计划、测试或文档阶段。

## 1.6　为什么没有计划阶段

显然，没有计划是不可能开发出一个软件产品的。因此，在每个项目的一开始就有一个**计划阶段**显然是最基本的。

关键在于，在明确知道将要开发什么之前，无法得出一个准确详尽的计划。因此，在使用传统范型开发一个软件产品时常常要进行三类计划活动：

1）在项目的开始，对管理需求和分析阶段进行初步计划。

2）一旦明确知道了将要开发什么，就制定出软件项目管理计划（SPMP）。这包括预算、

人员需求以及详细的日程安排。我们最早可以制定出项目管理计划是在规格说明文档已经得到客户批准的时候，也就是在分析阶段结束的时候。在此之前，计划还只是初步的和部分的。

3）在整个项目过程中，管理者需要监督 SPMP 的执行情况，并且注意是否有偏离计划的情况发生。

例如，假设某一项目的 SPMP 指出该项目整体需要用 16 个月，设计阶段将花费其中的 4 个月。一年以后，管理层注意到该项目整个比预想的进展要慢得多。详细调查显示，迄今为止，已有 8 个月用于设计阶段，它还远没有完成。该项目已经几乎可以肯定不得不放弃，到此为止花费的资金都浪费了。其实，管理层应当按阶段跟踪项目的进展，早在项目开始 2 个月后就应当注意到设计阶段的一个严重问题。在那时候，就应当决定如何按照最好的结果往下走。在这种情形下通常的第一个步骤是召开一个咨询会，以决定该项目是否是可行的，以及该设计小组是否有能力完成这项任务，或者继续下去的风险是大还是小。根据咨询报告，考虑各种候选措施，包括缩小目标产品的范围，然后设计并实现一个最现实的产品。仅当各种其他候选措施都认为不可行之后，该项目才可以取消。在这个具体的项目上，如果管理层密切监督这个计划，这个取消操作早在 6 个月之前就应当发生了，这将节省大量经费。

概括地说，不存在独立的计划阶段。反之，计划活动贯穿于软件生命周期的始终。然而，在有的时候计划活动占主导地位。这包括在项目的开始（最初计划）以及紧接着客户签署了规格说明文档之后（软件项目管理计划）。

## 1.7　为什么没有测试阶段

在一个软件产品开发出来后，非常仔细地检查它是非常重要的。因此，有必要问为什么在产品实现后没有测试阶段。

遗憾的是，在一个软件产品准备好交付给客户时才检查它实在是太晚了。例如，如果在规格说明文档中发现有一个错误，这个错误很可能已被带到设计和实现中了。在软件开发过程中，常常有几乎将其他活动排除在外而进行测试的时候，这发生在每个阶段接近结束的时候（**验证**），在产品提交客户之前尤其如此（**确认**）。尽管测试常常占有过重的分量，但永远也不要不进行测试。如果测试被看作一个独立的（**测试**）**阶段**，那么就会有不将测试连续贯穿于产品开发和维护的每个阶段的实际危险。

但是，即使这样也是不够的。需要的是连续检查一个软件产品。仔细检查应当自动伴随着每个软件开发和维护活动。与此相反，一个独立的测试阶段与确保一个软件产品尽可能在所有时候都无差错的目标是不一致的。

每个软件开发组织应当包含一个独立的组，它的主要职责是确保交付的产品是客户要求的，并且该产品在各方面都一直正确地建造。这个小组称为“软件质量保证”（Software Quality Assurance，SQA）组。软件的**质量**是指它满足规格说明的程度。第 6 章将更深入地介绍质量和软件质量，在该章中还将介绍 SQA 在设定和强化标准方面的作用。

## 1.8　为什么没有文档阶段

就像不应当有一个独立的计划阶段或测试阶段一样，也不应当有一个独立的**文档阶段**。与此相反，在任何时候，一个软件产品的文档必须是完整、正确和最新的。例如，在分析阶

段，规格说明文档必须反映规格说明的当前版本，对于其他阶段也一样。

1）为什么确保文档最新很重要，一个原因是软件行业中人员的流动性较大。例如，假设设计文档没有保持更新而主管设计师离开去做另一个工作了，现在就很难更新文档以反映系统设计过程中做出的所有变化。

2）如果前一个阶段的文档不是完整、正确和最新的，就几乎不可能执行下一阶段的步骤。例如，一个不完整的规格说明文档一定会不可避免地造成不完整的设计，然后造成不完整的实现。

3）除非提供文档来说明对一个软件产品期望的行为，否则便不可能测试该软件产品是否正确工作。

4）如果没有一套完整、正确的文档精确描述一个产品的当前版本做些什么，维护几乎是不可能的。

因此，如同没有独立的计划或测试阶段一样，也没有独立的文档阶段。相反，计划、测试和文档活动应当伴随着建造一个软件产品的其他活动进行。

现在我们考察面向对象范型。

## 1.9　面向对象范型

在 1975 年以前，大多数软件组织没有使用专门的开发技术，每一个人以他或她自己的方式工作着。在大约 1975 年到 1985 年之间，所谓**传统**或**结构化范型**（structured paradigm）的发展使这种情况有了突破性进展。组成传统范型的技术包括：结构化系统分析（11.3 节），数据流分析（13.3 节），结构化编程和结构化测试（14.13.2 节）。在最初使用时，这些技术似乎有极好的前景。然而，随着时间的推移，它们被证明存在两个方面的缺陷：

1）这些技术有时不能解决软件产品的规模越变越大的问题。即，传统技术在处理较小规模的产品（典型为 5000 行代码），或者甚至 50 000 行代码的中等规模的产品时是能够解决问题的。然而今天，500 000 行代码的大型产品是很普遍的，甚至 500 万行或更多程序行的产品也是司空见惯的，但是传统技术常常不具备有效的扩展能力以应对今天大型产品的开发。

2）交付后维护方面是传统范型不能满足人们最初期望的第二点。30 年前研究传统范型的主要驱动力是，平均来说，软件预算的 2/3 用于交付后维护（见图 1-3）。遗憾的是，传统范型并没能解决这个问题；如 1.3 节指出的那样，许多组织仍然花费 70％～80％的时间和精力用于交付后维护［Yourdon，1992；Hatton，1998］。

传统范型不完全成功的原因在于，传统技术要么面向操作，要么面向属性（数据），但却没有同时面向这两者。软件产品的基本组成是产品的行为和这些行为在其上进行操作的属性。例如，determine _ average _ height[⊖]是一种对一组高度值（属性）进行计算的操作，它返回这

⊖ 在本书中，传统软件产品中的变量名用下划线分隔变量名各部分的传统习惯书写，例如，this _ is _ a _ classical _ variable。面向对象软件产品的变量名使用面向对象的习惯书写，即用大写字母表示一个变量名中的新的部分的开始，例如，thisIsAnObjectOrientedVariable。

组高度值（属性）的平均值。某些传统技术，例如数据流分析（13.3 节），是面向操作的。即该技术将重点放在产品的操作上，属性则成了次要的。相反，像 Jackson 系统开发（13.5 节）这样的技术是面向属性的。这里，重点放在属性上，对属性的操作则是次要的。

相反，面向对象范型将属性和操作看作是同样重要的东西。一个简单的看待对象的方法就是将它看作是一个统一的**软件制品**（artifact，软件制品是软件产品的组成部分，如一个规格说明文档、一个代码模块或一本手册），它结合了属性和在该属性上施加的操作。这个对对象的定义是不完整的，将在本书的后面在定义继承（inheritance）这个概念之后（7.8 节），再对这个定义进行充实。尽管如此，这个定义还是抓住了“对象”这个概念的主要本质。

银行账户是对象的一个例子（见图 1-6）。该对象的属性成员是 accountBalance，能够对该账户余额实施的操作包括：向账户中存钱、从账户中取钱和计算余额。从面向对象的角度来看，银行账户是一个对象。这个对象在一个软件制品中将一个属性以及对该属性进行的三个操作结合在一起了。从传统范型的角度来看，它是一个涉及储蓄的产品，应当包括一个属性 account _ balance（账户余额）和三个操作（存钱、取钱和计算余额）。

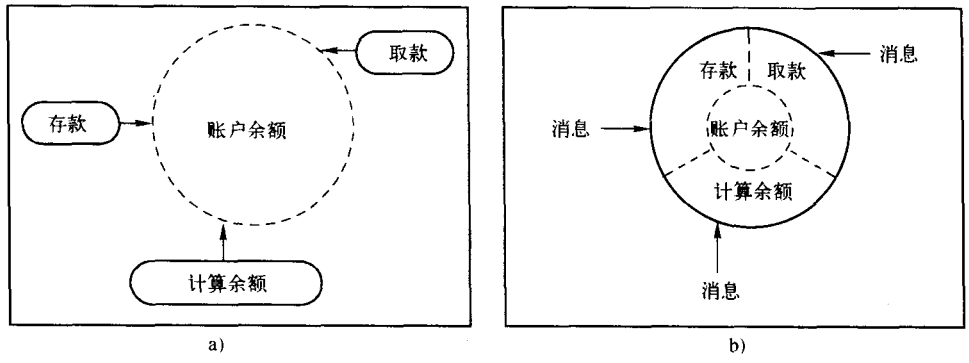



图 1-6　使用 a）传统范型和 b）面向对象范型进行的银行账户实现比较。对象周围的实线说明对象外部不知道 accountBalance（账户余额）这样的实现细节

到目前为止，这两种解决方法似乎并没有太大的不同。然而，关键问题在于实现对象的方法。特别是，像如何存储一个对象的属性这样的细节对象外部是不知道的。这是一个“信息隐藏”的例子，这方面的内容将在 7.6 节详细讨论。在图 1-6b 所示的银行账户对象的情形中可看到，软件产品的其余部分仅仅知道在银行账户对象中存在着一个叫余额的东西，但是它们对 accountBalance 的格式就一无所知了。这就是说，从银行账户对象外边不知道账户余额的实现方式是整型的、浮点型的，还是某种较大结构的一个域（组件）。对象周围的这道信息屏障在图 1-6b 中用实线说明，它描述了使用面向对象范型的一个实现。相反，图 1-6a 中 account _ balance 周围是虚线，因为，account _ balance 的所有细节对使用传统范型实现的各个模块来说，都是可见的，所以，任何模块都能改变 account _ balance 的值。

返回到图 1-6b 这个面向对象实现。如果客户在一个账户中存了 10 美元，那么就会有一个**消息**发给有关对象的“存钱”方法（**方法**是某个操作的一个实现），告诉它将 accountBalance 属性的值增加 10 美元。“存钱”方法在银行账户对象内，知道 accountBalance 是如何实现的，这一点在图 1-6b 中用对象内的虚线圆表示。但是对象外部的实体并不需要知道这方面的信息。图 1-6b 中的三种方法将 accountBalance 与产品的其余部分屏蔽起来，这标

志着这种信息的局部性。实现细节局限于对象内部带来了面向对象范型的众多优势，下面列出其中的一些：

1）关于交付后维护。假设银行储蓄软件产品使用传统范型进行设计，如果 account _ balance 的表示方法从比如说整型变成了某种结构的域，这样，软件产品中所有与 account _ balance 有关的地方都要发生变化，并且这些变化必须一致。相反，如果使用了面向对象范型，则仅仅需要在银行账户对象自身内部做相应改变。软件产品的其余部分并不知道 accountBalance 是怎样实现的，所以软件的其余部分不能访问 accountBalance。从而，储蓄产品的其余部分不需要改变。这样，面向对象范型使维护变得更迅速和更容易，出现**退化错误**（regression fault，意思为，对软件某处进行修改时，不小心在与该处明显没有关联的另一处造成了一个新的错误）的机会也极大地减少了。

2）除了维护之外，面向对象范型也使软件开发变得更容易。在许多情形下，对象在现实物理世界中都存在对应物。例如，银行产品中的银行账户对象就与采用该产品的银行的实际银行账户相对应。如同我们在第 2 部分将要看到的那样，建模在面向对象范型中起着非常重要的作用。软件产品中的对象与现实世界中对应实物的紧密对应关系将会导致更高质量软件的出现。

3）设计良好的对象是一个独立的单元。像我们曾经说明的那样，对象由属性和在属性上施加的操作共同组成。如果对对象中各属性执行的所有操作都包含在该对象的内部，那么就可以把该对象看成一个概念上独立的实体。产品中与现实世界有关的、被该对象模拟的部分都可以在对象中找到。有时将这种概念上的独立称为**封装**（encapsulation）（7.4 节）。独立还有另外一种形式，物理上独立。在一个设计良好的对象中，信息隐藏确保实现细节与对象外部的一切事物完全隔离。与外界通信的惟一形式就是给对象发消息，由对象执行特定操作，怎样执行操作完全成了对象自身的职责。由于这个原因，人们有时也将面向对象设计称为**职责驱动设计**（responsibility-driven design）[Wirfs-Brock，Wilkerson，and Wiener，1990] 或**按合同设计**（design by contract）[Meyer，1992]。（有关职责驱动设计的另一种观点，见下面的“如果你想知道”部分，它是从 Budd [2002] 处得来的一个例子。）

4）使用传统范型设计的产品是作为一组模块实现的，但是概念上它实质是一个单元。这也是传统范型应用于大型产品时不甚成功的一个原因。相反，当正确使用面向对象范型时，生产出的产品是由许多较小的、较大程度上独立的单元组成的。面向对象范型降低了软件产品的复杂度，从而简化了软件开发与维护。

5）面向对象范型提高了重用度。因为对象是独立的实体，它们可以用于将来的产品中。这种对象的重用减少了开发和维护所需的时间和费用，在第 8 章中将对此详细解释。

**如果你想知道**

*假设你住在加利弗尼亚的 Sacramento，你想为你住在芝加哥的妈妈送一束母亲节的鲜花。一个方法是到万维网上查芝加哥的黄页，看哪一个卖花人离你妈妈家最近，然后向该家花店订购鲜花。还有一个更简单的方法是在 1 800 flowers.com 网址上订购鲜花，将交付鲜花的所有事情都交给那个公司负责。这样，1 800 flowers.com 物理上位于哪儿以及哪家花店将递送你订购的鲜花都与你无关了。在任何情况下，该公司都不会泄露那个信息，这就是一个信息隐藏的例子。*

与上述方法完全相同，当向某个对象发出消息时，不仅请求是如何实现的无关紧要，并且发送消息的单元也不允许知道该对象的内部结构。对象本身完全负责消息执行的所有细节。

当使用面向对象范型时，需要对图 1-2 的传统软件生命周期略做修改。表 1-2 比较了传统范型和面向对象范型的软件生命周期。

**表 1-2　传统范型和面向对象范型的生命周期模型比较**

| 传统范型 | 面向对象范型 |
| --- | --- |
| 1. 需求阶段 | 1. 需求工作流 |
| 2. 分析（规格说明）阶段 | 2′. 面向对象分析工作流 |
| 3. 设计阶段 | 3′. 面向对象设计工作流 |
| 4. 实现阶段 | 4′. 面向对象实现工作流 |
| 5. 交付后维护阶段 | 5. 交付后维护 |
| 6. 退役 | 6. 退役 |

第一个区别显然是纯粹语义上的，阶段（phase）这个词用于传统范型，而工作流（workflow）一词用于面向对象范型。事实上，如我们将会在下一章指出的那样，在阶段和工作流之间没有对应关系。相反，这两个术语是完全不同的，这个区别体现了两种范型的生命周期模型的不同。

在本章中，我们考察这两个范型间的另一个不同，模块（在传统范型中）起的作用与对象（在面向对象范型中）起的作用的不同。我们首先考虑传统范型的设计阶段。如 1.3 节所述，这个设计阶段分成两个子阶段：结构化设计，然后是详细设计。在结构化设计子阶段，人们将产品分解成不同的部分（称为模块）。然后，在详细设计子阶段，再对每个模块的数据结构和算法依次进行设计。最后，在实现阶段，将这些模块最终实现。

如果采用面向对象范型，面向对象分析工作流的一个步骤是确定类。因为类是一种模块，因此结构化设计是在面向对象分析工作流中进行的。这样，面向对象分析比传统范型相应的分析（规格说明）阶段更进了一步。如表 1-3 所示。

**表 1-3　传统范型和面向对象范型的区别**

| 传统范型 | 面向对象范型 |
| --- | --- |
| 2. 分析（规格说明）阶段<br>· 确定产品要做什么 | 2′. 面向对象分析工作流<br>· 确定产品要做什么<br>· 提取类 |
| 3. 设计阶段<br>· 结构化设计（提取模块）<br>· 详细设计 | 3′. 面向对象设计工作流<br>· 详细设计 |
| 4. 实现阶段<br>· 用恰当的编程语言编码模块<br>· 集成 | 4′. 面向对象实现工作流<br>· 用恰当的面向对象编程语言编码类<br>· 集成 |

这两种范型间的差别带来了重要的结果。使用传统范型时，在分析（规格说明）阶段和设计阶段之间总是有一个很大的转变。毕竟，分析阶段的目的是确定产品做什么，而设计阶

段的目的是确定怎么做。相反，使用面向对象分析时，对象从一开始就进入了软件生命周期。人们在分析工作流中将对象提取出来，在设计工作流中对其进行设计，在实现工作流中对其进行编码。这样，面向对象范型是一个集成的方法；工作流与工作流之间的转变比传统范型要平缓得多，从而减少了开发中引入的错误数量。

像前面提到的，仅仅将对象定义成封装了属性和操作并实现了信息隐藏的一个软件制品是不充分的。在第 7 章中，我们将给出对象的更完整的定义，对其进行更深入的研究。

## 1.10　正确看待面向对象范型

图 1-1 是传统（结构化）范型的许多缺点的一些显现。然而，面向对象范型也绝不是医治这些顽症的灵丹妙药：

- 像所有软件生产的方法一样，也必须正确使用面向对象范型；像其他范型一样，很容易错误使用面向对象范型。
- 当正确使用的时候，面向对象范型可以解决一些（不是全部）传统范型遇到的问题。
- 面向对象范型有它自己的问题，见 7.9 节。
- 面向对象范型是目前可用的最好的方法。然而，像所有技术一样，将来它一定会被更先进的技术所取代。

在本书中，在讨论具体的话题的时候，将指出传统和面向对象范型的优缺点。这样，两个范型的比较就不仅在书中的某一处出现，而是散布在整本书中。

我们现在定义一些软件工程的术语。

## 1.11　术语

**客户**是想要建造（开发）某一产品的个体。**开发者**是小组的成员，他们负责建造该产品。开发者可能负责过程的每个方面，从需求开始，或者他们也可能只负责一个已经设计好的产品的实现。

客户和开发者可能是同一组织的一部分，例如，客户可能是保险公司的首席计算师，而开发者可能是一个由该保险公司负责软件开发的副总裁领导的一个小组。这称为**内部软件开发**。另一方面，对于**合同软件**，客户和开发者是完全独立的组织。例如，客户可能是国防部的一个高级官员，而开发者可能是专门从事武器系统软件的国防合同承包商的一个雇员。在一个更小的规模上，客户可能是一个单独从业的会计，而开发者可能是一个通过在业余时间编写软件赚取收入的学生。

涉及软件生产的第三方是**用户**。用户是客户委托产品所代表利益的那些人，他（们）将使用所开发的软件。在保险公司的例子中，用户可以是保险代理人，他将用软件选择最合适的保险单。在某些例子中，客户和用户是同一个人（例如，前面谈到的会计）。

与为一个客户编写昂贵的定制软件对应的是，软件的多个拷贝（如字处理软件或电子制表软件）被以较低的价格卖给大量的买者。即，这样软件的制造商（如 Microsoft 或 Borland）通过大量的卖出来填补产品开发的费用。这种类型的软件通常称为**商用现货**（commercial off-the-shelf，COTS）**软件**。这类软件早期称为**用收缩薄膜包装的软件**（shrink-wrapped software），因为装 CD、磁盘、手册以及产品许可的盒子几乎总是用收缩薄膜包装的。现在，COTS 软件常常可以通过万维网（World Wide Web）下载——不再有用收缩薄膜包装的盒子了。基于这个原

因，COTS软件现在有时被称为**点击软件**（clickware）。COTS软件是为“市场”开发的，就是说，在该软件开发出来并可以购买之前，没有特定的客户或用户。

**源码公开软件**现在变得极为普遍。一个源码公开软件产品是由一组自愿者开发和维护的，任何人都可以下载并免费使用。广泛使用的源码公开软件包括Linux操作系统和Apache Web服务器软件。“源码公开”一词指所有人都可得到源代码，不像大多数商业软件产品那样仅销售可执行版本。由于源码公开软件的任何使用者都可以浏览源代码并向开发者报告软件的错误，因此许多源码公开软件产品具有较高的质量。源码公开软件中的错误公开特性带来了很大的成果，它是由Raymond在《The Cathedral and the Bazaar》一书中以Linus法则的形式提出的，Linus法则是以Linux软件的创建人Linus Torvalds的名字命名的［Raymond，2000］。**Linus法则**指出，“如果足够多人给予关注，所有的软件错误都将一目了然。”换句话说，如果有很多人细察一个源码公开软件产品的源代码，应当有人能够定位每个错误并提议如何修改它。一个相关的法则是“尽早发布、经常发布”［Raymond，2000］。即，源码公开软件的开发者试图比源码封闭软件的开发者花较少时间用于测试，宁愿在一个新的软件版本刚刚完成就发布它，将更多的发现错误的职责交给用户。

本书中每页都使用到的一个词是**软件**。软件不仅由机器可识别的代码组成，而且包括作为每个项目固有组成部分的所有文档。软件包括规格说明文档、设计文档、各种法律和财务文档、软件项目管理计划，以及其他管理文档和各种类型的手册。

从20世纪70年代开始，**程序**和**系统**之间的界限开始变得模糊。在“过去的好时光”里，二者之间的区别是非常清楚的。一个程序是一个自治的代码段，通常以一堆打孔卡片的形式出现，它能够被执行。一个系统由许多这样的相关程序组成。比如，一个系统可能由程序P、Q、R和S组成，装入磁带$T_1$，程序P开始运行，机器开始读取一堆卡片然后送出磁带$T_2$和$T_3$作为输出，再将磁带$T_2$重新卷上，程序Q开始运行，送出磁带$T_4$作为输出。程序R将磁带$T_3$和$T_4$合并成磁带$T_5$；$T_5$作为程序S的输入，由程序S最后打印出一系列报表。

将上述情形与某一软件产品的情况进行比较，该产品在某一台机器上运行，它有一个前端通信处理器和一个后端数据库管理器，该软件实时控制一个炼钢厂。控制炼钢厂的这一软件与过去的系统相比做的要多得多，但如果依照程序和系统的传统定义来看，这个软件毫无疑问是一个程序。更使人迷惑的是，*系统*一词现在也用来表示硬件和软件的结合。例如，飞机的飞行控制系统由空中计算机和运行在计算机上的软件组成。视使用这个词汇的人而定，不同的人用这个概念有不同的涵义，飞行控制系统也可以包括控制部分，如操纵杆，它向计算机和飞机的各个部分（如计算机控制的机翼）发送命令。进一步地，在传统的软件开发范畴内，**系统分析**一词指软件开发的前两个阶段（需求和分析阶段），而**系统设计**指第三个阶段（设计阶段）。

为了减少这种迷惑，本书用**产品**这个概念来表示一段重要的软件。这个约定有两个原因。首先，是通过使用第三个概念以简单地避免程序和系统这两个概念的混乱。第二个原因更重要，本书涉及软件生产的**过程**，并且过程的最终结果是*产品*。最后，*系统*这个词使用它较现代的含义，即，软件和硬件的结合，或者作为那些被普遍接受的短语的一部分，例如，操作系统和管理信息系统等。

在软件工程范畴内得到广泛使用的两个单词是*方法*（methodology）和*范型*（paradigm）。

在 20 世纪 70 年代，**方法**一词开始用于“一种开发软件产品的方法”的含义。这个词实际上指有关方法的科学。然后，在 20 世纪 80 年代，就像在“这是一个全新的范型”这句话中所说的一样，**范型**一词成为工商界一个主要的时髦词。软件行业不久就开始在**面向对象范型**和**经典**（或**传统**）**范型**短语中使用范型一词，它表示“一种软件开发风格”。这是另一个令人遗憾的词汇选择，因为范型是一个模型或模式。我诚邀对词汇混用不满的知识渊博的读者们加入这场净化语言的斗争，我已经疲倦了这种唐吉诃德式的风车大战。

方法或范型应用于整个软件过程。相反，**技术**适用于软件过程的某个部分。例如，编码技术，文档技术，以及计划技术。

当一个程序员犯了**过错**，该过错的结果是造成代码中的**差错**。执行软件产品时就产生**故障**，即可观察到的产品的不正确行为，它是代码中的差错造成的。**错误**是差错的累积，造成结果的不正确。这些术语：过错（mistake）、差错（fault）、故障（failure）和错误（error）在 IEEE 标准 610.12“软件工程术语表”［IEEE 610.12，1990］中有定义，在 2002 年又对其重新确认［IEEE Standards，2003］。**缺陷**（defect）是一个通用词汇，它指一个差错、故障或错误。在本书中为精确描述考虑，我们因此减少通用术语“缺陷”的使用。

一个要尽量避免使用的词是 bug（**臭虫**）（该词的历史在下面的“如果你想知道”部分里）。bug 这个概念现在简单地是差错的委婉的说法。虽然使用这种说法并没有什么实际的坏处，但是 bug 这个词的暗示意义对好的软件产品并没有什么好处。特别是当一个程序员犯了过错时，他原来会说“我犯了一个过错”，他使用 bug 这个词后，会说“一个 bug 爬进了这些代码”（不是我的代码而是这些代码），从而将犯过错的责任从程序员身上转嫁给了 bug。一个程序员因流行感冒而病倒了没有人会责备他，因为流感是因为流感病毒引起的。将过错说成是 bug 是一种推卸责任的方式。相反，那些说“我犯了一个过错”的程序员，是一个对他或她的行为负责的计算机专业人员。

**如果你想知道**

首次用 bug 来说明一个差错，这个用法要归功于已故的美国海军少将 Grace Murray Hopper，COBOL 的设计者之一。在 1945 年 9 月 9 日，一只飞蛾飞进了 Hopper 和她的同事们在哈佛使用的 Mark Ⅱ计算机，并且在继电器的触点间居住下来。这样，系统中有了一只真正的 bug。Hopper 将这只 bug 记入了日志，日志写道，“首次发现了一只真正的 bug。”这本与飞蛾有关的日志，现在保存在位于维吉尼亚州 Dahlgren 的海军水面武器中心的海军博物馆里。

尽管这可能是计算机领域里首次使用 bug 这个词，但这个词在 19 世纪也作为工程行话使用过［Shapiro，1994］。例如，Thomas Alva Edison 在 1878 年 11 月 18 日写道，“这个事情发表了，然后 bug（指所谓的小差错和小问题）……”［Josephson，1992］。在 1934 年出版的《Webster 新英语字典》中有关于 bug 的一个定义“在仪器或其操作中的缺陷。”这个意思显然来自于 Hopper 的评论，她非常熟悉该词在其领域的用法；否则，她可能就会解释她要表达的意思了。

面向对象的术语方面也存在着相当多的混乱。例如，**属性**这个术语用于表示一个对象的数据成员，除了这个术语之外，**状态变量**这个术语有时也在面向对象的文献里表示相同的意思。在 Java 里，同样意思的术语是**实例变量**；在 C++ 里，使用**域**这个术语。关于对象操作的实现，

通常使用**方法**这个术语；然而，在C++里，这个术语是**成员函数**。在C++里，对象的一个**成员**既指它的一个“属性”（“域”），也指一个方法。在Java里，域这个术语用来表示一个属性（“实例变量”）或一个方法。为了避免混淆，本书使用一般的术语*属性*和*方法*。

幸运的是，一些术语已经被人们广泛接受。例如，当调用对象内的一个方法时，这已被普遍称为向对象**发送一个消息**。

## 1.12　道德问题

在本章快要结束的时候，我们给出一个警告性的说明。软件产品是由人开发和维护的，如果这些人勤劳、聪明、明智、现代，而且最重要的是要有道德，那么软件开发和维护的方式就极有可能是令人满意的。遗憾的是，相反的情况同样存在。

大多数专业人员团体都有一套它的成员必须遵守的道德规范。两个主要的计算机专业人员团体是计算机器联合会（Association for Computing Machinery，ACM）和电气电子工程师协会计算机分会（Computer Society of the Institute of Electrical and Electronics Engineers，IEEE-CS），它们联合批准通过了软件工程道德和从业规范，将其作为教学和软件工程实践的标准［IEEE/ACM，1999］。该标准有些过于冗长，因此产生了一个由导言和8条原则组成的简短版本：

**软件工程道德和从业规范**[⊖]**（5.2版）**

**（IEEE-CS/ACM联合工作组关于软件工程道德和从业规范，简版）**

**前　言**

规范的简版在一个较高层次抽象上概括了我们的期望，在完全版中包括的条款给出了一些例子和细节，显示了这些期望是如何改变我们作为软件工程专业人员的行为模式的。没有期望，细节将变得空洞和教条；没有细节，期望就只能是响亮的口号，期望和细节一同构成了密切关联的规范。

软件工程师将从事的是使软件的分析、规格说明、设计、开发、测试和维护成为一个有益并令人尊敬的职业。与从事公共健康、安全、福利等职业一样，软件工程师应当遵守下面8个准则：

1. *公众*——软件工程师应当始终如一地为公众的利益行事。
2. *客户和雇主*——软件工程师应当以一种最大程度上使客户和雇主的利益与公众利益一致的方式行事。
3. *产品*——软件工程师应当确保他们的产品和相关修改尽可能满足最高的专业标准。
4. *评判*——软件工程师应当维护专业评判标准的诚实性和独立性。
5. *管理*——软件工程管理者和领导者应当赞成并促进软件开发和维护管理的道德方法。
6. *专业*——软件工程师应当增进符合公众利益的专业的诚信和名誉。
7. *同事*——软件工程师应当对同事一视同仁并相互支持。
8. *自我*——软件工程师应当在他们的专业实践中终身学习并促进专业实践中的道德实践。

在其他计算机专业团体的道德准则中，表达了类似的想法。在我们的专业道路上严格遵守道德准则是至关重要的。

在下一章中，我们将考察各种生命周期模型，以便进一步阐述传统和面向对象范型的相异之处。

---

⊖ 1999年由电气电子工程师协会和计算机器联合会批准通过。

## 本章回顾

软件工程被定义为一门学科（1.1 节），它的目的是生产出满足客户要求的、未超出预算的、按时交付的没有错误的软件。为了实现这个目的，需要在软件生产的各个阶段使用恰当的技术，包括何时进行分析（规格说明）、设计（1.4 节）和交付后维护（1.3 节）。软件工程涉及软件生命周期的各个方面，结合了人类各个领域的许多知识，包括经济（1.2 节）和社会科学（1.5 节）。没有单独的计划阶段（1.6 节），没有测试阶段（1.7 节），也没有文档阶段（1.8 节）。在 1.9 节引入了对象的概念，将传统范型和面向对象范型做了一个简洁的比较。然后，对面向对象范型做了评价（1.10 节）。接下来，在 1.11 节，对本书中使用的术语做了一个解释。最后，在 1.12 节中讨论了道德问题。

## 进一步阅读

有关软件工程领域的最早的信息来源是［Boehm，1976］。对于在多大程度上可将软件工程看成一门真正的工程学科的分析，请见［Wasserman，1996］和［Ebert，Matsubara，Pezzé，and Bertelsen，1997］。在 Brereton 等人的著作中［1999］，Kroeker 等人的著作中［1999］，以及 Finkelstein 的著作中［2000］，对软件工程的未来进行了讨论。《IEEE Software》杂志 1999 年 5/6 月刊，特别是［Reel，1999］，对软件开发的关键问题进行了讨论。

关于交付后维护在软件工程中的重要性的观点，以及怎样为此制定计划，请见［Parnas，1994］。在 Mellor［1994］和 Neumann［1995］中，对软件的不可靠性和可能导致的风险（特别是在安全攸关的系统里）进行了分析。对于基于 COTS 产品的软件开发是［Brownsword，Oberndorf，and Sledge，2000］的主题。在［Scott and Vessey，2002］中讨论了企业系统中的风险，在［Longstaff，Chittister，Pethia，and Haimes，2000］中一般性地讨论了信息系统中的风险。有关软件危机的现代观点出现在 Glass［1998］中。Zvegintzov［1998］说明了软件工程实践中的精确统计数据实际上是非常难获得的。

数学是软件工程的支撑这个事实在 Devlin［2001］中得到了强调。经济学在软件工程中的重要性在 Boehm［1981］、Baetjer［1996］、Boehm and Huang［2003］中进行了讨论。《IEEE Software》杂志 2002 年 11/12 月刊包含了一些有关软件工程经济学的文章。

［Weinberg，1971］和［Shneiderman，1980］是关于社会科学和软件工程的两本经典书籍。阅读这两本书通常不需要心理学或行为科学的预备知识。社会科学和软件工程方面较新的一本书是［DeMarco and Lister，1987］。

Brooks的不朽著作，《The Mythical Man-Month》（中译本名为《人月神话》）［Brooks，1975］，是一本很受推崇的介绍软件工程现实的书籍。该书包括本章所有小节讨论的主题。

最好的公开源码软件的介绍是［Raymond，2000］。

［Budd，2002］和［Meyer，1997］对面向对象方法做了非常好的介绍。［Radin，1996］对范型给出了不偏不倚的看法。［Khan，Al-A'ali，and Girgis，1995］给出了传统范型和面向对象范型之间的区别。［Capper，Colgate，Hunter，and James，1994］对三个成功使用了面向对象范型的项目进行了描述，同时给出了详尽的分析。［Johnson，2000］报告了对 150 个有经验的软件开发人员对面向对象范型的态度的调查。［Maring，1996］和［Fichman and Kemerer，1997］给出了从大型面向对象产品开发中吸取的经验。［Webster，1995］对面向

对象范型的潜在缺陷进行了描述。

## 习题

1.1　你负责为一家白银提炼公司设计提取处理的计算软件。你的开发预算是 370 000 美元。交付后维护工作大概需要多少额外资金?

1.2　有一种方法能调和维护的传统定义和我们现在使用的定义吗?

1.3　你是一个软件工程顾问。一个图书发行商的执行副总裁希望你开发一个软件产品，该产品能够执行公司所有的会计核算功能，并能为总店的工作人员在线提供关于各个子公司订单和存货清单的信息，需要 18 个结算终端，27 个订货终端和 37 个库存终端。另外，有 16 个管理人员能够存取数据。总裁为该产品投入 25 000 美元资金，包括硬件和软件，并且希望 4 周内该产品全部完成。你将告诉他什么? 记住，作为一个顾问，无论客户的要求有多么不合理，你需要得到这份生意。

1.4　你是 Transmyopia 共和国的陆军准将，已决定召集一个软件开发组织为新一代激光制导反坦克导弹开发控制软件，你负责对整个项目进行监督。为了保护 Transmyopia 政府，在与软件开发商制定的开发条款中，你将包含哪些条款?

1.5　你是一个软件工程师，你的工作是监督习题 1.4 中的软件开发，列出你的公司可能在哪些方面不能满足与陆军的合同。造成这些问题的可能原因是什么?

1.6　交付产品 15 个月后，在一个跟踪 50 个最大城市的比萨饼消费情况的软件产品中发现了一个差错。纠正这个差错需要花费 14 594 美元。规格说明文档中的一个模糊语句导致了这个差错。估计一下，在分析阶段纠正该差错需要花费多少钱?

1.7　假设习题 1.6 中的差错在实现阶段发现。纠正该差错需要花费多少钱?

1.8　你是一个构建大型软件的组织的总裁。你将图 1-5 展示给你的雇员，要求他们在软件生命周期的早期发现错误。有人认为，在产品还没有开始生产前就开始纠正错误，对任何人来说都是不合理的。例如，如何在设计正在进行之时纠正一个编码错误? 对这个问题你怎样回答?

1.9　描述一下客户、开发商和用户都是同一个人的情形。

1.10　如果客户、开发商和用户都是同一个人，会出现什么问题? 如何解决这些问题?

1.11　对于客户、开发商和用户都是同一个人的情况，会有什么潜在的优势?

1.12　在字典中查询“系统”这个词。它有多少种不同的定义? 写出那些在软件工程背景下可用的定义。

1.13　在你第一份工作的第一天，经理给你一份程序清单，并说，“看看你能否把其中的 bug 找出来。”你将如何回答这个问题?

1.14　你负责开发习题 1.1 中的产品。你将使用面向对象范型还是传统范型? 给出你的理由。

1.15　(学期项目) 假定附录 A 中的 Ophelia’s Oasis 的产品已经像描述的那样正确实现。现要在建筑物中加入额外的 12 个房间，应该用什么方法改变现在的产品? 抛弃一切从头开始好不好?

1.16　(软件工程读物) 你的教师将提供 [Schach et al., 2003b] 的拷贝。对于基于管理者的估计和基于来自于实际数据的计算，你认为得到的结果各自的优缺点是什么?

## 参考文献

[Baetjer, 1996] H. BAETJER, *Software as Capital: An Economic Perspective on Software Engineering,* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Boehm, 1976] B. W. BOEHM, "Software Engineering," *IEEE Transactions on Computers* **C-25** (December 1976), pp. 1226–41.

[Boehm, 1979] B. W. BOEHM, "Software Engineering, R & D Trends and Defense Needs," in: *Research Directions in Software Technology,* P. Wegner (Editor), The MIT Press, Cambridge, MA, 1979.

[Boehm, 1980] B. W. BOEHM, "Developing Small-Scale Application Software Products: Some Experimental Results," *Proceedings of the Eighth IFIP World Computer Congress,* October 1980, pp. 321–26.

[Boehm, 1981] B. W. BOEHM, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Boehm and Huang, 2003] B. BOEHM AND L. G. HUANG, "Value-Based Software Engineering: A Case Study," *IEEE Computer* **36** (March 2003), pp. 33–41.

[Brereton et al., 1999] P. BRERETON, D. BUDGEN, K. BENNETT, M. MUNRO, P. LAYZELL, L. MACAULAY, D. GRIFFITHS, AND C. STANNETT, "The Future of Software," *Communications of the ACM* **42** (December 1999), pp. 78–84.

[Brooks, 1975] F. P. BROOKS, JR., *The Mythical Man-Month: Essays on Software Engineering,* Addison Wesley, Reading, MA, 1975; Twentieth Anniversary Edition, Addison Wesley, Reading, MA, 1995.

[Brownsword, Oberndorf, and Sledge, 2000] L. BROWNSWORD, T. OBERNDORF, AND C. A. SLEDGE, "Developing New Process for COTS-Based Systems," *IEEE Software* **17** (July/August 2000), pp. 40–47.

[Budd, 2002] T. A. BUDD, *An Introduction to Object-Oriented Programming,* 3rd ed., Addison Wesley, Reading, MA, 2002.

[Capper, Colgate, Hunter, and James, 1994] N. P. CAPPER, R. J. COLGATE, J. C. HUNTER, AND M. F. JAMES, "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories," *IBM Systems Journal* **33** (No. 1, 1994), pp. 131–57.

[Cutter Consortium, 2002] Cutter Consortium, "78% of IT Organizations Have Litigated," *The Cutter Edge,* **www.cutter.com/research/2002/edge020409.html**,[⊖] April 09, 2002.

[Daly, 1977] E. B. DALY, "Management of Software Development," *IEEE Transactions on Software Engineering* **SE-3** (May 1977), pp. 229–42.

[DeMarco and Lister, 1987] T. DEMARCO AND T. LISTER, *Peopleware: Productive Projects and Teams,* Dorset House, New York, 1987.

[Devlin, 2001] K. DEVLIN, "The Real Reason Why Software Engineers Need Math," *Communications of the ACM* **44** (October 2001), pp. 21–22.

[Ebert, Matsubara, Pezzé, and Bertelsen, 1997] C. EBERT, T. MATSUBARA, M. PEZZÉ, AND O. W. BERTELSEN, "The Road to Maturity: Navigating between Craft and Science," *IEEE Software* **14** (November/December 1997), pp. 77–88.

[Elshoff, 1976] J. L. ELSHOFF, "An Analysis of Some Commercial PL/I Programs," *IEEE Transactions on Software Engineering* **SE-2** (June 1976), 113–20.

[Fagan, 1974] M. E. FAGAN, "Design and Code Inspections and Process Control in the Development of Programs," Technical Report IBM-SSD TR 21.572, IBM Corporation, December 1974.

[Fichman and Kemerer, 1997] R. G. FICHMAN AND C. F. KEMERER, "Object Technology and Reuse: Lessons from Early Adopters," *IEEE Computer* **30** (July 1997), pp. 47–57.

[Finkelstein, 2000] A. FINKELSTEIN (Editor), *The Future of Software Engineering,* IEEE Computer

⊖ 本书中的这些 URL 在本书付印时是正确的，然而，互联网地址经常会改变，却没有事先或事后的声明。如果遇到这样的情况，读者应使用搜索引擎来定位新的 URL。

Society Press, Los Alamitos, CA, 2000.

[GJSentinel.com, 2003] "Sallie Mae's Errors Double Some Bills," **www.gjsentinel.com/news/content/coxnet/headlines/0522_salliemae.html**, May 22, 2003.

[Glass, 1998] R. L. GLASS, "Is There Really a Software Crisis?" *IEEE Software* **15** (January/February 1998), pp. 104–5.

[Grady, 1994] R. B. GRADY, "Successfully Applying Software Metrics," *IEEE Computer* **27** (September 1994), pp. 18–25.

[Hatton, 1998] L. HATTON, "Does OO Sync with How We Think?" *IEEE Software* **15** (May/June 1998), pp. 46–54.

[IEEE 610.12, 1990] *A Glossary of Software Engineering Terminology,* IEEE 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., 1990.

[IEEE Standards, 2003] "Products and Projects Status Report," **standards.ieee.org/db/status/status.txt**, June 3, 2003.

[IEEE/ACM, 1999] "Software Engineering Code of Ethics and Professional Practice, Version 5.2, as recommended by the IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practice," **www.computer.org/tab/seprof/code.htm**, 1999.

[IEEE/EIA 12207.0-1996, 1998] "IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995," Institute of Electrical and Electronic Engineers, Electronic Industries Alliance, New York, 1998.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, Switzerland, 1995.

[Johnson, 2000] R. A. JOHNSON, "The Ups and Downs of Object-Oriented System Development," *Communications of the ACM* **43** (October 2000), pp. 69–73.

[Johnson et al., 2001] J. JOHNSON, K. D. BOUCHER, K. CONNORS, AND J. ROBINSON, "Project Management: The Criteria for Success," **www.softwaremag.com/archive/2001feb/CollaborativeMgt.html**, February/March 2001.

[Josephson, 1992] M. JOSEPHSON, *Edison, a Biography,* John Wiley and Sons, New York, 1992.

[Kan et al., 1994] S. H. KAN, S. D. DULL, D. N. AMUNDSON, R. J. LINDNER, AND R. J. HEDGER, "AS/400 Software Quality Management," *IBM Systems Journal* **33** (No. 1, 1994), pp. 62–88.

[Kelly, Sherif, and Hops, 1992] J. C. KELLY, J. S. SHERIF, AND J. HOPS, "An Analysis of Defect Densities Found during Software Inspections," *Journal of Systems and Software* **17** (January 1992), pp. 111–17.

[Khan, Al-A'ali, and Girgis, 1995] E. H. KHAN, M. AL-A'ALI, AND M. R. GIRGIS, "Object-Oriented Programming for Structured Procedural Programming," *IEEE Computer* **28** (October 1995), pp. 48–57.

[Kroeker et al., 1999] K. K. KROEKER, L. WALL, D. A. TAYLOR, C. HORN, P. BASSETT, J. K. OUSTERHOUT, M. L. GRISS, R. M. SOLEY, J. WALDO, AND C. SIMONYI, "Software [R]evolution: A Roundtable," *IEEE Computer* **32** (May 1999), pp. 48–57.

[Leveson and Turner, 1993] N. G. LEVESON AND C. S. TURNER, "An Investigation of the Therac-25 Accidents," *IEEE Computer* **26** (July 1993), pp. 18–41.

[Lientz, Swanson, and Tompkins, 1978] B. P. LIENTZ, E. B. SWANSON, AND G. E. TOMPKINS, "Characteristics of Application Software Maintenance," *Communications of the ACM* **21** (June 1978), pp. 466–71.

[Longstaff, Chittister, Pethia, and Haimes, 2000] T. A. LONGSTAFF, C. CHITTISTER, R. PETHIA, AND Y. Y. HAIMES, "Are We Forgetting the Risks of Information Technology?" *IEEE Computer* **33** (December 2000), pp. 43–51.

[Maring, 1996] B. MARING, "Object-Oriented Development of Large Applications," *IEEE Software* **13** (May 1996), pp. 33–40.

[Mellor, 1994] P. MELLOR, "CAD: Computer-Aided Disaster," Technical Report, Centre for Software Reliability, City University, London, July 1994.

[Meyer, 1992] B. MEYER, "Applying 'Design by Contract'," *IEEE Computer* **25** (October 1992), pp. 40–51.

[Meyer, 1997] B. MEYER, *Object-Oriented Software Construction*, 2nd ed., Prentice-Hall, Upper Saddle River, NJ, 1997.

[Naur, Randell, and Buxton, 1976] P. NAUR, B. RANDELL, AND J. N. BUXTON (Editors), *Software Engineering: Concepts and Techniques: Proceedings of the NATO Conferences*, Petrocelli-Charter, New York, 1976.

[Neumann, 1980] P. G. NEUMANN, Letter from the Editor, *ACM SIGSOFT Software Engineering Notes* **5** (July 1980), p. 2.

[Neumann, 1995] P. G. NEUMANN, *Computer-Related Risks*, Addison Wesley, Reading, MA, 1995.

[Parnas, 1994] D. L. PARNAS, "Software Aging," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 279–87.

[Radin, 1996] G. RADIN, "Object Technology in Perspective," *IBM Systems Journal* **35** (No. 2, 1996), pp. 124–26.

[Raymond, 2000] E. S. RAYMOND, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates, Sebastopol, CA, 2000; also available at **www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/**.

[Reel, 1999] J. S. REEL, "Critical Success Factors in Software Projects," *IEEE Software* **16** (May/June 1999), pp. 18–23.

[Schach et al., 2002] S. R. SCHACH, B. JIN, D. R. WRIGHT, G. Z. HELLER, AND A. J. OFFUTT, "Maintainability of the Linux Kernel," *IEE Proceedings—Software* **149** (February 2002), pp. 18–23.

[Schach et al., 2003b] S. R. SCHACH, B. JIN, G. Z. HELLER, L. YU, AND J. OFFUTT, "Determining the Distribution of Maintenance Categories: Survey versus Measurement," *Empirical Software Engineering* **8** (December 2003), pp. 357–66.

[Scott and Vessey, 2002] J. E. SCOTT AND I. VESSEY, "Managing Risks in Enterprise Systems Implementations," *Communications of the ACM* **45** (April 2002), pp. 74–81.

[Shapiro, 1994] F. R. SHAPIRO, "The First Bug," *Byte* **19** (April 1994), p. 308.

[Shneiderman, 1980] B. SHNEIDERMAN, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, Cambridge, MA, 1980.

[St. Petersburg Times Online, 2003] "Thousands of Federal Checks Uncashable," **www.sptimes.com/2003/02/07/Worldandnation/Thousands_of_federal_.shtml**, February 7, 2003.

[Stephenson, 1976] W. E. STEPHENSON, "An Analysis of the Resources Used in Safeguard System Software Development," Bell Laboratories, Draft Paper, August 1976.

[Wasserman, 1996] A. I. WASSERMAN, "Toward a Discipline of Software Engineering," *IEEE Software* **13** (November/December 1996), pp. 23–31.

[Webster, 1995] B. F. WEBSTER, *Pitfalls of Object-Oriented Development*, M&T Books, New York, 1995.

[Weinberg, 1971] G. M. WEINBERG, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.

[Wirfs-Brock, Wilkerson, and Wiener, 1990] R. WIRFS-BROCK, B. WILKERSON, AND L. WIENER, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Yourdon, 1992] E. YOURDON, *The Decline and Fall of the American Programmer*, Yourdon Press, Upper Saddle River, NJ, 1992.

[Zelkowitz, Shaw, and Gannon, 1979] M. V. ZELKOWITZ, A. C. SHAW, AND J. D. GANNON, *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Zvegintzov, 1998] N. ZVEGINTZOV, "Frequently Begged Questions and How to Answer Them," *IEEE Software* **15** (January/February 1998), pp. 93–96.

# 第 2 章　软件生命周期模型

**学习目标**

学完本章之后，你应当能够：

- 描述在实践中信息系统是如何开发的；
- 理解进化树生命周期模型；
- 意识到软件产品改变造成的负面影响；
- 使用递增生命周期模型；
- 了解米勒法则对软件生产的影响；
- 描述迭代-递增生命周期模型的优点；
- 意识到及早降低风险的重要性；
- 对比其他各种生命周期模型。

第 1 章描述在理想世界中如何开发软件产品。本章的主题是在实践中会发生些什么。如我们将描述的那样，在理论和实践之间有巨大的差别。

## 2.1　理论上的软件开发

在理想世界中，软件产品是像第 1 章所描述的那样开发的。从图 2-1 中的按步骤描述可以看出，一个系统是从零开始开发的。∅表示空集（如果你想知道词汇"从零开始"的来历，请见"如果你想知道"部分）。首先明确客户需求，然后进行分析。当分析制品完成后，就从事设计，然后是整个软件产品的实现，最后将该软件安装在客户的计算机上。

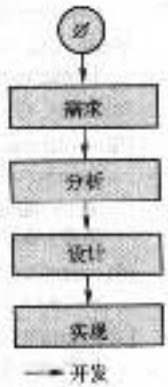



图 2-1　理想的软件开发

然而，软件开发在实践中有很大程度的不同，有两个原因：首先，软件专业人员是人，因此他们会犯过错。第二，当软件正在开发时客户的需求会发生变化。这一章将深入地讨论这两个问题，但是首先我们给出一个小型实例研究来说明涉及的问题，该实例参考了［Tomer and Schach，2000］中的实例研究。

**如果你想知道**

词汇"从零开始"（from scratch）意思是"从无开始"，它来自于 19 世纪的体育词汇。在道路（和跑道）铺就之前，赛跑只能在开阔地上举行。在许多情况下，起跑线是沙地上用木棍划（scratch）的一条线，即是"从划线开始"。

现在"划线"（scratch）一词有了一个不同的体育用法。"scratch golfer"在高尔夫运动中是指一个零差点球员，零差点球员的打球成绩是评定高尔夫球场难度值的基础。

## 2.2　Winburg 小型实例研究

为了缓解印第安那州 Winburg 市区交通拥塞的状况，市长说服市政府建立一个公共交通系统。将建立公共交通专用通道，鼓励通勤人员"停车换乘"，即将他们的小汽车停在郊区的停车场然后从该处转乘公共汽车上下班，每乘一次花费 1 美元，每辆公共汽车将设一个

自动收款机，它只接受 1 美元钞票。乘客进入公共汽车时将 1 美元塞入进钞孔，自动收款机中的传感器扫描该钞票，然后机器中的软件使用一个图像识别算法，以确定该乘客是否确实在进钞孔中插入了一张真正的 1 美元钞票。重要的是自动收款机要非常准确，因为一旦有随便一张纸就可骗过自动收款机的新闻传出，车费收入将直线下降甚至为零。相反，如果机器经常拒绝 1 美元真钞，乘客将不愿意乘坐公共汽车。另外，收款机必须速度快。如果机器花费 15 秒钟才确认 1 张 1 美元钞票是有效的——这将使好几分钟内只有少数乘客能登上汽车，乘客同样不愿意乘坐公共汽车。因此，对收款机软件的需求包括平均响应时间少于 1 秒钟并且平均准确度至少为 98%。

第 1 幕：实现了该软件的第 1 版。

第 2 幕：测试显示，要求的确定 1 美元钞票有效的平均 1 秒钟的响应时间没有达到。事实上，平均用了 10 秒钟得到响应。高级管理人员找到了原因。看起来是为了达到要求的 98%的准确度，编程人员曾被他的经理要求对所有的数学计算使用双精度数字。结果是，每个操作数都比通常的单精度数字花费至少 2 倍的时间。结果造成程序慢了许多，导致较长的响应时间。随后的计算显示，尽管经理告诉程序员那样做，规定的 98%的准确度即使使用单精度数字也可以达到。程序员开始对实现做必要的改变。

第 3 幕：在该程序员完成工作之前，对系统的进一步测试显示，即使对实现做了上面指出的改变，系统仍然有平均 4.5 秒的响应时间，没有接近规定的 1 秒的时间。问题在于复杂的图像识别算法。幸运的是，刚刚发明了一个快速算法，因此使用新的算法重新编写了收款机软件。结果是平均响应时间成功达到了。

第 4 幕：到现在，这个项目进度已经大大落后了并且超出了预算。这位市长是一个出色的企业家，他有了一个好主意，请求软件开发小组试着尽量提高系统中美钞识别组件的准确度，好将生成的软件包卖给自动售货机公司。开发组采纳了一个新的设计，改进了平均准确度，达到 99.5%。管理者决定在收款机上安装这个版本的软件。此时，软件的开发就完成了。这个城市后来将这个系统卖给两家小型自动售货机公司，补偿了 1/3 的项目超支。

尾声：几年后，收款机中的传感器变得陈旧了，需要用一个较新的模块替换它。管理人员建议利用这个改变来同时更新硬件。软件专业人员指出，硬件的改变意味着也需要新的软件。他们建议用一个新的编程语言重写软件。在编写程序期间，这个项目比原计划落后 6 个月，还超出预算 25%。然而，参加项目的每个人都确信新系统将比原系统更可靠并且质量更高，但还是“尽量减小改变”以满足它的响应时间和准确度要求。

图 2-2 描述了该小型实例研究的**进化树生命周期模型**。最左边的方框代表第 1 幕。如图所示，该系统是从零（∅）开始开发的。然后，依次是需求（需求$_1$）、分析（分析$_1$）、设计（设计$_1$）和实现（实现$_1$）。接下来，如前面所述，对该软件第 1 版的试验显示 1 秒钟的响应时间达不到，因而必须对实现做出修改。修改的实现在图 2-2 中作为实现$_2$ 出现。然而，实现$_2$ 从未完成。这就是表示实现$_2$ 的方框用虚线画的缘故。

在第 3 幕中，需求不得不改变。特别是使用了一个更快的图像识别算法。修改的需求（需求$_3$）造成了修改的规格说明（分析$_3$）、修改的设计（设计$_3$）以及修改的实现（实现$_3$）。

最后，在第 4 幕中，改变了设计（设计$_4$）以增加准确度。然后，实现必须改变（实现$_4$）以达到设计上的变化。

在图 2-2 中，实线箭头表示开发，虚线箭头表示维护。例如，当在第 4 幕中改变了设计时，设计$_4$代替了设计$_3$作为分析$_3$的设计。

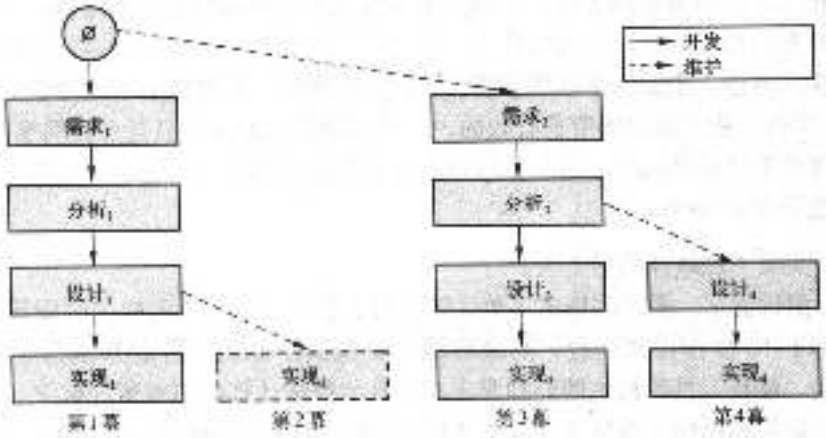



图 2-2　Winburg 小型实例研究的进化树生命周期模型，虚线画的方框表示该实现没有完成

进化树模型是**生命周期模型**（或简称**模型**）的一个例子，即在软件产品开发和维护过程中的一系列要执行的步骤。另一个可以用于此小型实例研究的生命周期模型是**瀑布生命周期模型**［Royce，1970］。一个简化版的瀑布模型见图 2-3。这个传统生命周期模型可以看作是带反馈环的图 2-1 的线性模型。然后，如果在设计期间发现了一个由需求中的差错引起的差错，顺着虚线向上的箭头，软件开发人员可以回溯设计到分析并由此到需求，并在那里做必要的改正。然后，他们向下移到分析，改正规格说明文档以反映对需求的改动，并顺次纠正设计文档。设计工作现在就可以在原来发现差错的地方继续进行了。同样，实线箭头表示开发，虚线箭头表示维护。

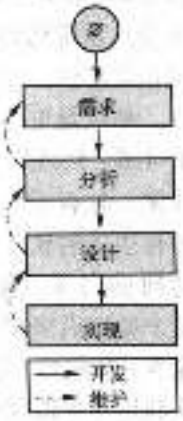



图 2-3　瀑布生命周期模型的简化版

瀑布模型当然能够用于表示 Winburg 小型实例研究，但是与图 2-2 的进化树模型不同，它无法显示事件的顺序。进化树模型比瀑布模型有一个进一步的优势。在每一幕的结尾，我们有一个**基准**，那就是，一套完整的软件制品（**制品**是一个软件产品的一个组成部分）。在图 2-2 中有 4 个基准，它们是：

- 在第 1 幕的结尾：需求$_1$、分析$_1$、设计$_1$、实现$_1$
- 在第 2 幕的结尾：需求$_1$、分析$_1$、设计$_1$、实现$_2$
- 在第 3 幕的结尾：需求$_3$、分析$_3$、设计$_3$、实现$_3$
- 在第 4 幕的结尾：需求$_3$、分析$_3$、设计$_4$、实现$_4$

第一个基准是最初的一套软件制品；第二个基准反映了第 2 幕的修改后的实现（但是从未完成），同时还有未修改的第 1 幕的需求、分析和设计。第三个基准是图 2-2 所示的一套完整的新的软件制品。第四个基准与第三个基准相同，只是设计和实现改变了。我们将在第 5 章和第 15 章再一次接触基准的概念。

## 2.3　Winburg 小型实例研究心得

Winburg 小型实例研究描述了软件产品的开发由于一些互不相干的原因而出错的情况，这些原因包括较差的实现策略（使用不必要的双精度数）和使用较慢的算法。最终，该项目是成功了。但是，显而易见的问题是，软件开发在实践中真是混乱无序吗？事实上，这个小型实例研究中的问题比起许多软件开发项目（虽然不是大多数）来说要小得多。在 Winburg 小型实例研究中，仅有一个新版本软件是由差错（误用双精度数）造成的，另两个新版本软件的出现是由于客户做出的改变（转换到更快的算法以及需要增加准确度）。

为什么需要对一个软件产品做出这么多的改变呢？首先，如前所述，软件专业人员是人，因此会出错；第二，一个软件产品是某个现实世界的一个模型，而现实世界是不断改变的。在 2.4 节中我们将深入讨论这个问题。

## 2.4　野鸭拖拉机公司小型实例研究

野鸭拖拉机公司在全美国的大部分地区销售拖拉机。该公司曾经要求它的软件部门开发一个新的软件产品，能够处理它的业务的各个方面。例如，该产品必须能够处理销售、库存以及向销售人员支付佣金，还能提供所有必需的财会功能。当这个软件产品正在实现的时候，野鸭拖拉机公司买下了加拿大的一家拖拉机公司。野鸭拖拉机公司的管理层决定，为了省钱，加拿大的业务要并到美国的业务中去。这意味着该软件完成前要进行修改：

1）必须修改它来处理增加的销售地区。
2）必须扩展它来处理那些在加拿大有所不同的业务方面，如税费。
3）它必须扩展以处理两种不同的货币，美元和加元。

野鸭拖拉机公司是一个快速增长的公司，它具有良好的业务前景。接管加拿大拖拉机公司是一个积极的进展，这很可能导致未来几年更大的效益。但是，从软件部门的观点来看，购买这家加拿大的拖拉机公司可能是一场灾难。要不是进行需求、分析和设计的时候考虑到加入未来可能的扩展，加入加拿大销售地区的工作可能非常大，可能抛弃迄今做的每件事而从零做起会更高效。原因是改变这个阶段的产品与在该产品的生命周期的后期（见图 1-6）修改它是相似的。扩展软件处理与加拿大市场有关的方面以及加拿大货币可能同样困难。

即使软件是经过深思熟虑的并且最初的设计确实是可扩展的，设计出的拼补在一起的产品不可能如最初就设计成满足美国和加拿大业务那样结合得好。这会给将来的维护带来严重的连带问题。

野鸭拖拉机公司软件部是**移动目标问题**的牺牲品。就是说，在软件正在开发的时候，需求改变了。它与改变的原因是非常值得的并没有关系，事实是，接管加拿大公司对于正在开发的软件的质量是非常有害的。

在某些情况下，移动目标的原因很少是仁慈的。有时一个组织内有权利的高层管理人会不断地改变他或她关于正在开发的软件产品的功能的想法。在另一种情形下，是**特性的蔓延**，即连续地向需求中加入小的甚至是琐碎的特性。但是，不管是什么原因，频繁的改变，不管它们看起来多么微不足道，对于一个软件产品的健康状况是有害的。重要的是一个软件产品设计成一套尽可能独立的组件，以使对于该软件某个部分的改变不在代码明显无关的部

分引入差错，称为**退化（性）差错**（regression fault）。当做大量改变的时候，结果是在软件内部引起连动，最后，存在许多的连动从而实际上任何改变都会引入一个或多个退化差错。在这种时候，惟一能做的事就是重新设计整个软件产品并重新实现它。

遗憾的是，对移动目标问题目前还没有解决办法。对需求的积极性改变而言，业务不断增长的公司总是要改变的，这些改变必须反映在公司的重要任务软件产品中。对于消极性改变，如果个人要求做这些改变的决定有较大来头，那便无法阻止对正在进行的实现做修改，也无法阻止对该软件产品将来的维护性的损害。

## 2.5　迭代和递增

由于移动目标问题和需要纠正在软件产品开发过程中不可避免的错误，实际软件产品的生命周期类似于图 2-2 的进化树模型或图 2-3 的瀑布模型，而不像图 2-1 的理想化的过程链。这种现实情况的结果是谈论“分析阶段”没有太多的意义，相反，分析阶段的操作散布在生命周期的各个阶段。同样，图 2-2 显示了实现阶段的四个不同版本，其中的一个版本（实现$_2$）由于移动目标问题而从未实现。

考察一个软件制品的后续版本，如规格说明文档或一个编码模块。从这个观点看，基本的过程是迭代的。即，我们制作制品的第一个版本，然后修改它并制作第二个版本，如此进行。我们的目的是每个版本比它的前一个版本离我们的目标更新一步，最终构建一个满意的版本。**迭代**是软件工程的一个固有特性，迭代生命周期模型已经使用了 30 多年［Larman and Basili，2003］。例如，瀑布模型是在 1970 年首次提出的，它是迭代的（但不是递增的）。

开发现实世界软件的第二个方面是**米勒法则**施加给我们的限制。在 1956 年，乔治·米勒，一位心理学教授，指出：在任何时候，人类最多只能够将精力集中在 7 桩事情（桩：信息的单位）上［Miller，1956］。然而，一个典型的软件制品远不止有 7 桩。例如，一个编码制品很可能有远远超过 7 个变量，一个需求文档很可能有远远多于 7 个需求。我们人类处理米勒法则的限制的一个办法是使用**逐步求精**方法。即，我们集中精力于事情目前最重要的那些方面，把那些不那么紧急的方面向后拖延。换句话说，事情的每个方面最终都要处理，但是要按照目前的重要性依次进行。这意味着我们开始建造一个软件制品仅解决我们要达到目标的一小部分。然后，进一步考虑问题的其他方面，并向已有的软件制品中加入生成的新的片断。例如，我们可能通过考虑我们认为最重要的 7 个需求来建造一个需求文档。然后，我们将考虑 7 个次重要的需求，如此下去。这是一个递增过程。**递增**也是软件工程的一个固有特性，递增软件开发有超过 45 年的历史了［Larman and Basili，2003］。

实践中，迭代和递增相互结合使用。即，一个软件制品是一块一块制造的（递增），每个增加经过多个版本（迭代）。这些思想在图 2-2 中解释，它表示了 Winburg 小型实例研究（2.2 节和 2.3 节）的生命周期。如图 2-2 中所示，没有单独的“需求阶段”。客户的需求被提取和分析了两次，分别是初始需求（需求$_1$）和修改的需求（需求$_3$）。同样地，没有单独的“实现阶段”，只有四个独立的片断，在其中编制代码并对它进行修改。

这些观点概括于图 2-4 中，它反映了迭代－递增生命周期模型所蕴含的基本概念［Jacobson，Booch，and Rumbaugh，1999］。该图用四个递增显示了软件产品的开发，分别标注为递增 A、递增 B、递增 C 和递增 D。水平坐标轴是时间，垂直坐标轴是人时（1 人时是一个人在 1 个小时内所能做的工作量），因此在每个曲线下的阴影区是那个增量的总的工作量。

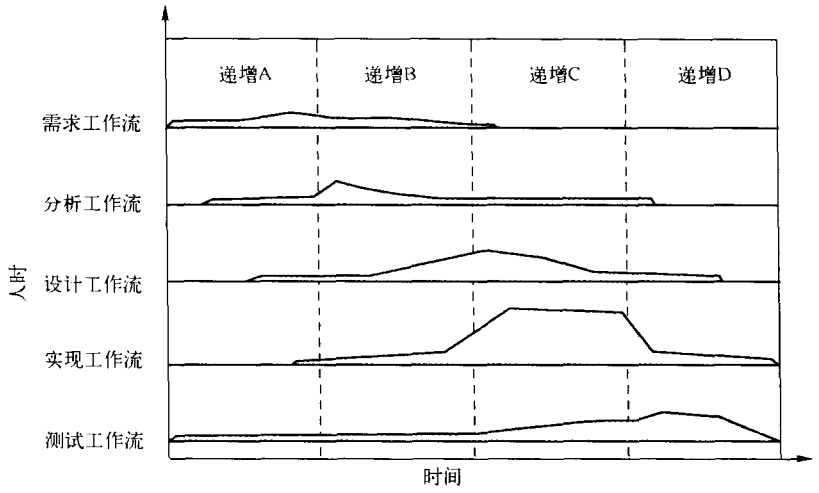



图 2-4　建造一个有四个增量的软件产品

重要的是要理解图 2-4 只描述了一个软件产品被分解为增量的一种可能的方法。另一个软件产品可以只用 2 个增量来建造，而第三个软件产品可能需要 13 个增量。进一步地，该图并不打算精确地描述一个软件产品是如何开发的。相反，它显示从迭代到迭代强调的重点是如何变化的。

图 2-1 的顺序阶段是制品构建阶段。相反，如图 2-4 中明确反映的那样，我们必须知道在整个生命周期中进行的不同的**工作流**（workflow）（活动）。有五个**核心工作流：需求工作流**（活动）、**分析工作流、设计工作流、实现工作流**以及**测试工作流**。如上面所说，全部五个工作流是在一个软件产品的整个生命周期进行的。然而，有时一个工作流比其他工作流更重要。

例如，在生命周期的开始，软件开发人员提取一套最初的需求。换句话说，在迭代-递增生命周期的开始，需求工作流占主导地位。在剩下的生命周期中扩展和修改这些需求制品。在那期间，其他四个工作流（分析流、设计流、实现流和测试流）占主导地位。换句话说，在生命周期的开始，需求流是主要工作流，但是它的相对重要性随后就降低了。相反，在临近软件生命周期结束的时候，实现流和测试流占用了软件开发小组成员的大部分时间。

计划和文档活动贯穿整个迭代－递增生命周期进行。进一步地，测试在每次迭代期间，特别是在每次迭代结束的时候，是一个主要活动。此外，软件一旦完成后要进行整体上的测试。在那时，测试然后按照测试结果修改实现实际上是软件小组惟一的活动。图 2-4 的测试流中反映出这一点。

图 2-4 显示四个递增。考虑左栏描述的递增 A，在这个递增开始的时候，需求小组的成员明确客户的需求。一旦多数需求明确了之后，分析部分的第一版可以开始了。当分析方面已经有了明显的进展之后，可以开始第一版的设计了。甚至某些编码工作常常在这第一个递增阶段就做了，它可能是以概念证明原型的形式来测试提议的软件产品的部分的可行性。最后，如前面所提到的，计划、测试和文档活动从第一天就开始并一直持续着，直至软件产品最终交付用户。

类似地，在递增 B 期间，初始的工作集中在需求和分析工作流上，然后是设计流。递

增 C 期间的重点首先是在设计流上，然后在实现流和测试流上。最后，在递增 D 期间，实现流和测试流占主要成分。

像表 1-1 中所反映出的那样，大约 20% 的全部工作量用于需求和分析流（总共），另外 20% 用于设计流，大约 60% 用于实现流。图 2-4 中阴影面积大小的相对比较说明了这一点。

在图 2-4 中的每个递增期间有一个迭代，见图 2-5，它描述了在递增 B 期间的三个迭代（图 2-5 是对图 2-4 的第二栏的一个放大的观察）。如图 2-5 所示，每个迭代包括全部五个工作流，但面积比例改变了。

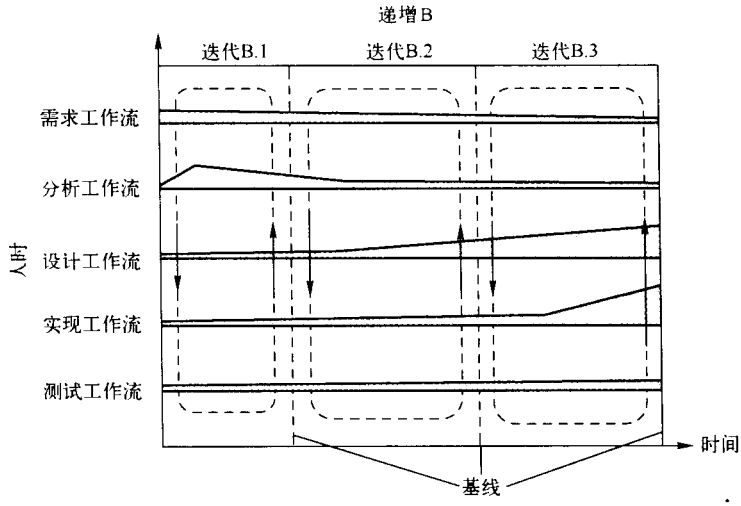



图 2-5　图 2-4 迭代－递增生命周期模型的递增 B 的三个迭代

必须再一次强调，图 2-5 不是想显示每个增量严格地包括 3 个迭代，迭代数因递增的不同而不同。图 2-5 想显示的是，在每个递增内的迭代以及在几乎每个迭代期间进行的对全部五个工作流（需求、分析、设计、实现和测试，连同计划和文档一道）的重复，尽管每次的比例有所改变。

如前面解释的那样，图 2-4 反映递增是每个软件产品开发所固有的特性。图 2-5 清楚地显示了在每个递增内所包含的迭代。特别是，图 2-5 描述了与一个大的递增相对应的三个连续的迭代步骤。更详细地说，迭代 B.1 由需求流、分析流、设计流、实现流和测试流组成，由最左边带圆角的虚线方框表示。迭代持续下去直至五个工作流的每个软件制品都令人满意为止。

接下来，全部五套软件制品都在迭代 B.2 中迭代进行。这第二个迭代与第一个性质相似。即，需求制品改进了，它接下来导致分析制品的改进，如此下去，如图 2-5 中第二个迭代所示，对于第三个迭代也同样。

迭代和递增的过程始于递增 A 的开始，持续下去直至递增 D 的结束。完成的软件产品然后安装在客户的计算机上。

## 2.6　修订的 Winburg 小型实例研究

图 2-6 显示了 Winburg 小型实例研究（图 2-2）添加在迭代－递增生命周期模型之上的进化树模型（没有显示测试流，因为进化树模型假定连续测试，如 1.7 节所解释的那样）。

图2-6揭示了递增的另外的含义：

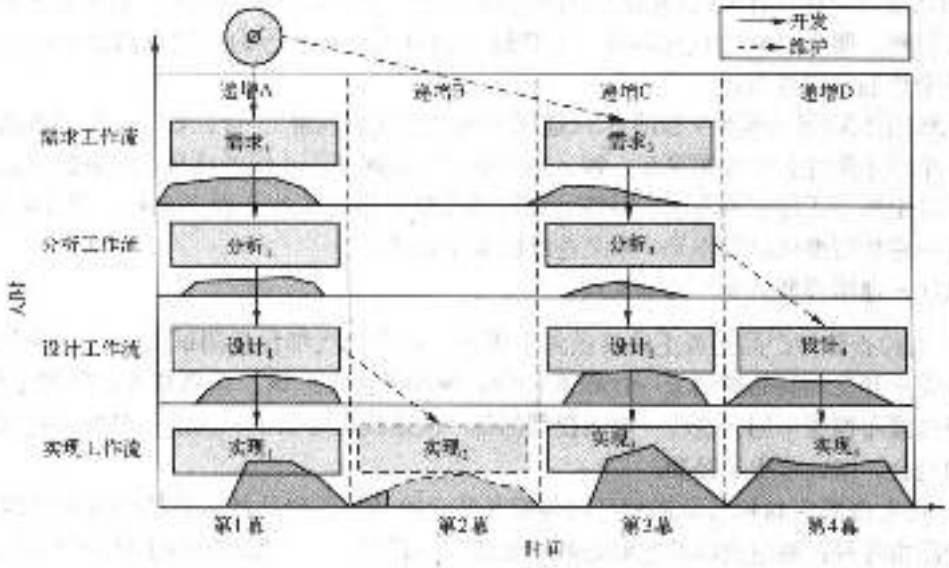



图2-6 Winburg小型实例研究（图2-2）添加在迭代-递增生命周期模型之上的进化树模型

- 递增A对应于第1幕，递增B对应于第2幕，如此等等。
- 从迭代-递增模型的观点来看，两个递增没有包含全部四个工作流。具体地说，递增B（第2幕）仅包括实现流，递增D（第4幕）仅包括设计流和实现流。迭代-递增模型不需要在每个递增内执行每个工作流。
- 进一步说，在图2-4中需求流的大部分在递增A和递增B中执行了，而在图2-6中它在递增A和递增C中执行。还有，在图2-4中，多数分析是在递增B中执行的，而在图2-6中，分析流是在递增A和递增C中执行的。这强调指出，图2-4和图2-6都没有代表每一个软件产品建造的方式。每个图显示了一个特定软件产品的建造方式，强调其中蕴含的迭代和递增。
- 图2-6的递增B（第2幕）期间实现流的较小规模和突然终止显示实现$_2$没有完成。图中更深的阴影块反映出那部分实现流没有完成。
- 进化树模型的三个虚线箭头显示每个递增形成前一个递增的维护。更准确地说，在这个例子中，每个递增是纠正的维护的一个实例。就是说，每个递增纠正了前一个递增中的差错。如前面解释的那样，递增B（第2幕）通过用普通单精度变量替代双精度变量纠正了实现流。递增C（第3幕）通过规定使用一个更快的图像识别算法纠正了需求流。然后对分析流、设计流、实现流做出相应的改变。最后，在递增D（第4幕）中改变设计从而改进了整个软件的准确度。必须对实现流做出相应的改变。

## 2.7 迭代和递增的风险和其他方面

另一种考察迭代和递增的方式是项目作为一个整体可以划分为更小的小项目（或增量）。每个小项目扩展出需求、分析、设计、实现和测试制品。最后，得到的一套制品构成了完整的软件产品。

实际上，每个小项目不只是由扩展了的制品组成。根本上是要检查每个软件制品是正确的（测试流）并且对有关的制品做必要的修改。这个检查并修改、然后是再检查再修改如此下去的过程，明显具有迭代的特征。它持续下去直至开发小组的成员对当前小项目（或增量）的全部制品满意为止。

比较图 2-3（瀑布模型）和图 2-5（递增 B 内对迭代的观察），可以看出，每个迭代可以看作是一个较小但完整的瀑布模型。即，在每个迭代期间开发小组的成员在该软件制品的某一特定部分上经历了传统的需求、分析、设计和实现阶段。从这个观点来看，图 2-4 和图 2-5 的迭代－递增模型可以看作是一系列连续的瀑布模型。

迭代－递增模型有如下几个优点：

1）为检查软件产品是否正确提供多个机会。每个迭代都包括测试流，因而每个迭代都为迄今为止开发的制品提供了一次检查机会。越晚检查出差错，花费越大，如图 1-5 所示。与传统的瀑布模型不同，迭代－递增模型的每一次迭代都提供了一次进一步发现差错并纠正它们的机会，因此节省了经费。

2）在生命周期的相当早期就可以确定其蕴含的结构的健壮性。软件产品的**结构**指出该软件产品由各种制品组成以及它们是如何组装在一起的。一个类似的例子是一个大教堂的结构，它可能被描述成罗马式、哥特式或巴洛克式，等等。同样，一个软件产品的结构可能被描述成面向对象（第 7 章）、管道和滤波（UNIX 或 Linux 组成）或客户－服务器（带有一个中央服务器，为客户计算机网络提供文件存储）。使用迭代－递增模型开发的软件产品的结构必须具有可以连续扩展以包含下一次递增的属性（如果有必要，容易改变）。能够处理这样的扩展和改变，同时却没有支离破碎，这称为**健壮性**。健壮性是开发软件产品期间的重要特性，在交付后维护期间它尤其重要。因此，如果一个软件产品要持续交付后 12 年、15 年或更长的维护期，其隐含的结构必须是健壮的。当使用一个迭代－递增模型，不久就可清楚看出软件结构是否健壮了。如果（比如说）在合并第三个递增的过程中，到现在为止开发的软件还不得不很大程度上重新组织和重新编写，那么显然这个结构不是十分健壮。客户必须决定是否放弃这个项目从头开始。另一个可能是重新设计结构使它更健壮，然后在进行到下一递增前尽可能重用当前的制品。健壮非常重要的另外一个原因是移动目标问题（2.4 节）。几乎可以肯定客户的需求将变化，这既因为客户组织内的业务不断增长，也因为客户对于目标软件将要做什么的想法在不断改变。软件结构越健壮，软件改变起来越有弹性。设计一个能够适应大量改变的结构是不可能的，但是，如果要求的改变在一定程度上是合理的，一个健壮的结构应当能够适应这些改变，而不用大量重新构建。

3）迭代－递增模型使我们能够较早地减轻风险。在软件开发和维护中，**风险**是不可避免的。例如，在 Winburg 小型实例研究中，最初的图像识别算法不够快：曾经出现一个完成的软件产品将无法满足时间限制的风险。递增开发一个软件产品使我们能够在生命周期的早期减少这种风险。例如，假设正在开发一个新的局域网（LAN），开发者担心当前的网络硬件不适合新的软件产品。然后，要求第一个或前两个迭代朝着建造与网络硬件接口的那部分软件进行。如果事实证明该网络具有必要的能力，担心是多余的，开发者便能够继续进行这个项目，相信这个风险已经减少了。另一方面，如果该网络确实不能应付新的 LAN 生成的附加业务，将情况在生命周期早些时候报告给客户，这时只花费了预算的很小一部分。客户现在可以决定是否取消这个项目，扩展现有网络的能力，购买新的功能更强大的网络，或

采取其他行动。

4）我们总是有该软件的一个工作版。假定使用图 2-1 的传统生命周期模型开发一个软件产品，仅在项目结束的时候才有一个该软件产品的工作版。相反，当使用迭代－递增生命周期模型时，在每个迭代的末尾有一个部分的目标软件产品的工作版。客户和目标用户可以试验该版本，然后决定需要做什么改变以确保将来完全的实现能够满足他们的要求。这些改变可以在随后的递增中做出，客户和用户然后能够确定是否需要做进一步的改变。关于这方面的一个变种是交付软件产品的部分版，不仅用作试验，也用于顺利地在客户组织内推广新的软件产品。改变几乎总是被视为一种威胁。用户担心，在工作岗位上太频繁地引入新的软件会导致他们让位于计算机。然而，逐渐地引入新的软件产品能够带来两个好处。其一，可以理解的担心会被计算机替代的恐惧消除了。其二，如果该功能在几个月内逐渐地介绍，而不是一下子整体介绍进来，则学习一个复杂的计算机软件产品的功能并不难。

## 2.8　迭代和递增的控制

乍一看，图 2-4 和图 2-5 的迭代－递增模型显得完全杂乱无序。与瀑布模型（图 2-3）从需求到实现的顺序进行不同，开发者显得随心所欲做自己喜欢的事，也许上午做一些编码，午饭后做一个或两个小时的设计，然后在回家前做半个小时的规格说明。事实并非如此。相反，迭代－递增模型与瀑布模型一样是受严格控制的，因为如前面所指，使用迭代－递增模型开发一个软件产品与开发一系列较小的软件产品（都使用瀑布模型）相比没有什么不同。

更详细地说，如图 2-3 所示，使用瀑布模型开发一个软件产品意味着将软件产品作为一个整体，对其顺序执行需求、分析、设计和实现阶段。如果碰到一个问题，执行图 2-3 的反馈环（虚线箭头），即进行迭代（维护）。然而，如果使用迭代－递增模型开发相同的软件产品，该软件产品被当作一套递增看待。对于每个递增，在该递增内依次重复执行需求、分析、设计和实现阶段，直至明确地不再需要进一步的迭代。换句话说，该项目整体上分割为一系列瀑布式小项目。在每个小项目期间，根据需要执行迭代，如图 2-5 所示。因此，前面之所以说迭代－递增模型与瀑布模型都是严格控制的，是因为迭代－递增模型就是瀑布模型，是成功应用的瀑布模型。

## 2.9　其他生命周期模型

我们现在考察一些其他生命周期模型。包括螺旋模型和同步－稳定模型。我们从不太有名的编码－修补模型开始。

### 2.9.1　编码－修补生命周期模型

令人遗憾的是，许多产品都是用所谓的**编码－修补生命周期模型**（code-and-fix life-cycle model）开发的。实现产品时没有需求或规格说明，也没有进行设计方面的尝试。开发者只是简单地将代码拼凑在一起，为满足客户的要求，多次改写该软件。这个方法如图 2-7 所示，它清楚地显示了缺乏需求、规格说明和设计的情形。尽管这种方法对于 100 行或 200 行长的短程序而言可以操作得很好，但对于任何合理规模的软件产品来说，编码－修补模型则完全不能令人满意。图 1-5 显示，如果修改是在需求、规格说明和设计阶段进行，修改软件产品的花费相对会小些。但如果在产品已经编出代码或者更糟地，产品已经交付并安装在客

户计算机上，则修改软件产品的代价会大得令人不能接受。因此，实际上编码－修补方法的花费远远大于有正确的规格说明、经过仔细设计的产品所需的花费。另外，对没有规格说明和设计文档的产品进行维护相当困难，而且发生退化错误的机会也相当大。要想取代编码－修补方法，重要的是在产品的开发过程开始之前，选择一个合适的生命周期模型（life-cycle model）。

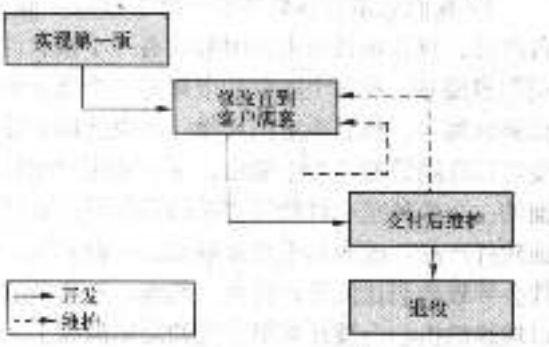



图 2-7　编码－修补生命周期模型

可悲的是，太多的项目使用编码－修补模型了。在那些用代码行惟一地度量项目进展的组织内，这个问题尤其突出，因此软件开发小组的成员们被迫从项目开始的第一天起尽可能多地辛辛苦苦编出一行行代码。编码－修补模型是最简单的软件开发方式，也是迄今为止最糟糕的方式。

2.2 节给出了瀑布模型的一个简化版。现在我们深入考察这个模型。

### 2.9.2　瀑布生命周期模型

**瀑布生命周期模型**最初是由 Royce［Royce，1970］提出的，图 2-8 显示了当产品正在开发时用于维护的反馈环，图 2-3 的简化瀑布模型中反映了这一点。然而，图 2-8 也显示出对于交付后维护的反馈环。

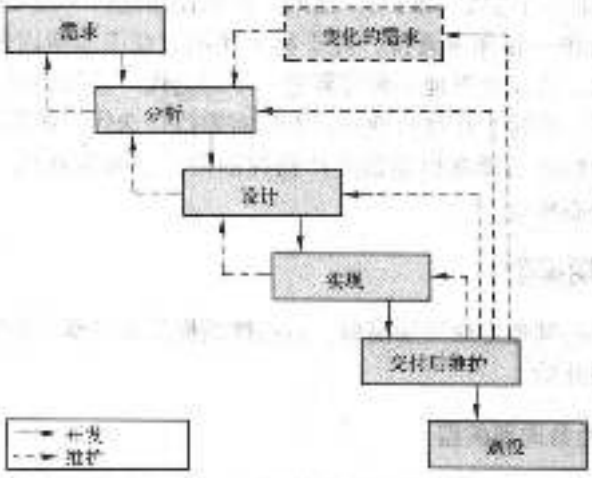



图 2-8　完整的瀑布生命周期模型

关于瀑布模型的一个关键点是，在任何阶段的文档完成并且该阶段的产品被客户和软件质量保证（software quality assurance，SQA）小组认可之前，该阶段是没有完成的。这种情况也存在于修改中：如果一个早期阶段的产品作为执行反馈环的结果不得不做出改变，早期阶段的产品必须要等到该阶段的文档修改过了并且修改被 SQA 小组认可后才告结束。

瀑布模型的每个阶段都内在地包含测试。测试不是一个仅在产品建造完成后才要进行的

独立的阶段，也不是仅在每个阶段结尾时进行的。相反，如 1.7 节所述，测试应当在软件过程中连续进行。特别是，在维护期间，必须要确保修改的产品版本不仅仍能够做先前版本所做的——仍然正确执行（回归测试），而且它还要满足客户提出的任何新的需求。

瀑布模型有许多优点，包括强制训练方法——规定在每个阶段提供文档，以及要求每阶段的所有产品（包括文档）需经 SQA 仔细检查。然而，瀑布模型是文档驱动的事实也是它的一个弱点。要明白这一点，看看下面两个有些奇怪的想像情形：

第一个情形是，琼和简·约翰逊夫妇决定建一幢房子，他们向一个建筑师请教。建筑师没有给他们看草图、规划或模型，而是交给他们一份用专业术语描述的满满 20 页的只有文字的文档。虽然两人都没有建筑学的经验，很难看懂这文档，但还是头脑发热地签署了合同并说“就这样建吧。”

下面是另一个情形，马克·马布雷邮购了一套衣服。公司没有给他寄衣服的图片和布料的样本，却寄来一个文件描述布料和裁剪方式，马克完全根据这份书面描述订购了一套衣服。

前面的两种情况各不相同，然而它们准确地代表了通常使用瀑布模型建造软件的方式。建造过程开始于规格说明，通常规格说明文档很长、强调细节并很枯燥，很难阅读。客户通常没有阅读软件规格说明的经验，另外，因为规格说明文档通常是以一种客户所不熟悉的风格写成的，因而就难上加难。如果规格说明文档以像 Z 语言［Spivey，1992］（11.9 节）这样形式化的规格说明语言写成，那困难就更大了。不过，客户不管是否真正明白，都会签署规格说明文档的。在很大程度上，这就如同琼和简·约翰逊夫妇签署并不完全理解的合同盖房子。

马克·马布雷和他邮购的衣服可能看起来非常奇怪，但它却准确地描述了在软件开发中使用瀑布模型的情形。客户只能在整个产品完成编程之后才首次看到能够工作的产品。因此难免软件开发者会害怕客户这样说：“我知道这是按我要求做的软件，但它真的不是我想要的那样。”

问题出在哪里？客户按照规格说明文档的描述所理解的产品与实际的产品有很大的不同。规格说明只存在于纸面上；客户因而不能真正理解产品本身会是什么样。瀑布模型如此依赖于文字上的规格说明，会直接导致建造出的产品不符合客户的真正需求。

公平地说，就像建筑师能够通过模型、草图和规划帮助客户了解房子会建成什么样子一样，软件工程师可以使用图形技术，例如数据流图（11.3.1 节）来与客户进行交流。问题是这些图形没有描述出完成后的产品是如何工作的。例如，一个流程图（产品的图形化描述）与工作着的产品存在着很大的差异。本书提出两个办法来解决下述问题：即规格说明文档通常未能使客户确定是否提出的产品满足他或她的需求。第 10 章和第 12 章介绍面向对象解决办法。下一节描述传统解决办法，即快速原型开发模型。

### 2.9.3　快速原型开发生命周期模型

**快速原型**（rapid prototype）是一个与产品子集功能相同的工作模型。例如，如果目标产品是处理账目支出、账目收入和库存，则快速原型组成的产品可能完成用于数据捕获的屏幕处理以及打印报表，但不进行文件更新和错误处理。一个目标产品用于确定溶液中酶的浓度，它的快速原型可能进行计算并显示答案，但不对输入数据进行合理性检查和确认。

图 2-9 所示的**快速原型开发生命周期模型**的第一步是建造一个快速原型，并让客户和未

来的用户试用快速原型。一旦客户认为快速原型确实满足了大多数要求，开发者就可以拟制规格说明文档，并对产品将能够满足客户的实际要求满怀信心。

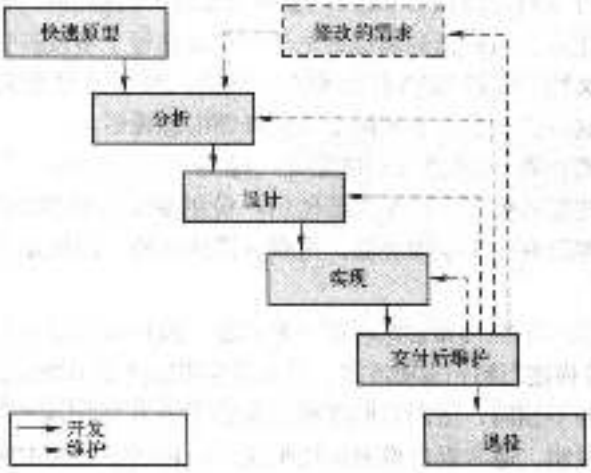



图 2-9　快速原型开发生命周期模型

建立快速原型后，软件过程按图 2-9 所示继续进行。快速原型开发模型的一个主要优点是，产品的开发从快速原型到交付的产品基本上是线性的；瀑布模型的反馈环（图 2-8 所示）在快速原型开发模型中不太需要。这里有许多原因：第一，开发小组使用快速原型来创建规格说明文档，因为通过与客户的交互，运行的快速原型已经得到了确认，所以由此得来的规格说明文档会是正确的；第二，考察一下设计阶段，尽管（非常恰当地）匆忙组装出快速原型，设计组成员还是可以从中获得深刻的理解——最起码，也能从中看出“不能做哪些”。再重复一遍，瀑布模型的反馈环在这里并不太需要。

接下来是实现阶段。在瀑布模型中，设计的实现有时会使设计上的错误显现出来。在快速原型开发模型中，由于初步的工作模型已经建造出来，这会减少在实现阶段中或实现阶段后修改设计的要求。原型给设计小组以启迪，尽管它可能只反映了完整的目标产品的部分功能。

一旦客户接受了产品并安装了它，交付后维护就开始了。根据所进行的具体的维护内容，软件周期重新进入需求、分析、设计或实现阶段。

快速原型的基本特性是体现一个“快”字。开发者应该尽可能快地建造原型，以加快软件开发进程。毕竟快速原型的惟一用途是确定客户真正的需要是什么；一旦将它确定下来，将丢弃快速原型的实现，但从中所学到的东西保留下来并应用到接下来的开发阶段中。为此，快速原型的内部结构无关紧要，最重要的是快速建造原型并快速修改以反映客户的需求。所以，速度是关键。

第 10 章将更深入地讨论快速原型开发。

### 2.9.4　极限编程

**极限编程**（eXtreme Programming，XP）[Beck，2000] 是在迭代-递增模型基础上发展起来的一种颇有争议的新的软件开发方法。第一步是软件开发小组确定客户希望产品支持的

各种特性（**情节**），对于每个特性，开发小组向客户通报实现这个特性需要的时间和花费。第一步相当于迭代－递增模型的需求和分析工作流（参见图 2-4）。

客户使用成本－效益分析方法（5.2 节）选择每个后续的构件所应包含的特性，也就是说，根据开发小组提供的时间、经费估计和该特性给客户带来的潜在收益来进行选择。提议的构件被分成更小的部分，称为**任务**（task）。一个程序员首先制定出一个任务的测试用例，这称为**测试驱动开发**（test-driven development，TDD）。然后与一个工作伙伴在一台计算机前一起工作（**结对编程**，pair programming）［Williams，Kessler，Cunningham and Jeffries，2000］，该程序员实现这个任务，确保全部测试用例正确工作。之后把这个任务集成到产品的当前版本中。理想情况下，实现和集成一个任务只需要几个小时。通常许多对程序员并行地实现任务，以便集成可以连续地进行。各任务所使用的 TDD 测试用例保留下来并应用到所有进一步的集成测试中。

极限编程（XP）有许多不同寻常的特性：

- XP 小组的计算机设在一个大房间的中心，大房间中有许多彼此相连的小隔间。
- 一个客户代表一直与 XP 小组一起工作。
- 没有一个人能够连续两周超时工作。
- 没有规格说明，而是 XP 小组的所有成员一同完成规格说明、设计、编码和测试。
- 在建造出各种构件之前没有概要设计步骤。相反，建造产品的过程中设计在不断地调整，这个过程被称为**重组**（refactoring）。只要有测试用例无法运行，就重新组织代码，直到小组满意地认为，这个设计是简单、易懂的并能正确地运行所有测试用例。

XP 已经在一些小规模的项目上取得成功。然而，XP 还没有得到充分的应用，无法确定这种生命周期模型是否能够兑现它早先的承诺。进一步说，即使 XP 可用于小规模软件产品，但这并不意味着它能够用于中规模或大规模软件产品，这一点将在后面解释。

极限编程是一些新的统称为**敏捷过程**（agile process）的范型中的一个［Beck et al.，2001］。比起其他生命周期模型来说，它们的特点是不怎么强调分析和设计，在生命周期中实现开始得较早，因为人们认为运行的软件比详细的文档更重要。对于变化的响应是敏捷过程的另一个主要目标，与客户的沟通也同样重要。

为了理解为什么那么多软件专业人员曾对使用敏捷过程来开发中等规模和大规模软件产品表示过怀疑，我们考察 Grady Booch［2000］做出的如下类推。任何人都能够成功地将几块木板钉成一个狗窝，但是，没有详细计划就建造一个三居室的房屋则是愚蠢之极。此外，还需要一些管道铺设、布线和铺顶的技术来建造三居室的房屋，而且检查也是必要的。（也就是说，能够建造小型软件产品不一定有能力建造中型或大型软件产品。）进一步地，一栋摩天大楼有 1000 个狗窝那么高的事实并不意味着一个人能够通过一层层地堆积 1000 个狗窝来建造摩天大楼。换句话说，建造大型软件产品比起将小型软件产品修补、堆砌在一起，需要更专业和复杂的技术。

在决定是否敏捷过程确实是软件工程中的一个主要突破时，一个关键的决定性因素是看将来交付后维护的成本（1.3.2 节）。即，如果使用敏捷过程造成交付后维护成本的降低，XP 和其他敏捷过程将被广泛接受。另一方面，重组是敏捷过程的一个固有组成部分。如前面解释的那样，产品不是整体上一起设计，设计其实是递增开发的，只要因为某种原因目前

的设计不令人满意，就要重新安排编码。此重组随后在交付后维护期间连续进行。如果一个通过验收测试的产品的设计是可修整的和灵活的，那么维护将易于完成且成本较低。然而，如果当额外功能加入时设计必须得重组，那么产品的交付后维护的成本将高得令人难以接受。由于这个方法还较新，仅在开发方面有使用敏捷过程的少量试验数据，几乎没有维护方面的数据。然而，原始数据显示重组会消耗很大百分比的总成本［Li and Alshayeb，2002］。

概括地说，敏捷过程看来也只是在客户需求模糊的情况下，一种非传统的构建小规模软件产品的方法。然而，实验显示敏捷过程有一些特性工作得很好。例如，Williams，Kessler，Cunningham 和 Jeffries［2000］显示，结对编程在短期内可开发较高质量的代码，带来较高的工作满意度。相应地，即使敏捷过程整体上证明是令人失望的，敏捷过程的某些特性在将来也可能被软件工程实践的主流所采纳。

### 2.9.5　同步－稳定生命周期模型

Microsoft 公司是世界上最大的 COTS 软件生产商。它的大多数软件包使用迭代－递增模型的一种版本建造，这种版本的模型叫做**同步－稳定**（synchronize-and-stabilize）**生命周期模型**［Cusumano and Selby，1997］。

在需求分析阶段的一开始，需要访问软件包的很多潜在顾客，提取出对顾客具有最高优先级的特性列表。现在就可以拟制规格说明文档了。接下来，将工作分为 3 或 4 个构件（builds），第一个构件包含最重要的特性，第二个构件包含次重要的特性，如此等等。每个构件都由一些小组并行地完成。每天工作结束前，所有小组进行工作**同步**（synchronize），也就是把部分完成的组件放在一起，测试和调试得到的产品。在每个构件结束时进行**稳定化**（stabilization）工作。修补至今检测到的遗留差错，并且现在将该构件**冻结**（freeze），也就是说，规格说明不会再修改。

重复的同步步骤保证各个组件总能一起工作。部分完成产品的这种定期执行的另一个优点是，开发者能早些深入了解产品的工作状态，而且必要时在组件生成的过程中修改规格说明。甚至在最初的规格说明未完成时都可使用这个模型。同步－稳定模型在 4.5 节详细讨论小组组织时会进一步涉及。

最后讨论的是螺旋模型，它结合了 2.9 节描述的所有其他模型的诸多特性。

### 2.9.6　螺旋生命周期模型

如 2.5 节中所述，软件开发中几乎总会有风险，例如，在产品充分归档之前，关键人物可能会离职；产品主要依赖的硬件生产商可能会破产；在测试和质量保证中的投入不是太多就是太少；在投入了成千上万美元来开发一个主要的软件产品之后，技术上的突破可能使整个产品价值全无；一个公司研究并开发了一个数据库管理系统，但在产品面市前，竞争对手可能抢先上市了价格更便宜、功能相当的软件包；在产品集成时发现产品的组件可能无法组合在一起。出于明显的原因，软件开发者都尽力将这种风险减到最小。

构建原型是减小某些类型风险的一个途径。2.9.3 节讲到，要减小交付产品不符合客户需求的风险，最佳方法是在需求阶段创建一个快速原型。在下一阶段，其他种类的原型可能更合适。例如，一个电话公司可能设计了一种新型高效的算法为远程网络呼叫进行路由选择。如果该产品实现了，但没有如期望的那样工作，电话公司可能就浪费了开发这项产品的

费用。另外，愤怒或感觉不方便的顾客将不再愿顾该公司，这种情形是可以避免的，可以创建一个**概念证明原型**，只处理呼叫的路由选择并在一个仿真器上进行测试。通过这种方式，实际的系统并未受到影响，实现的费用只限于路由选择算法，电话公司可决定是否值得用新的算法改进整个网络控制器。

概念证明原型不是一个如2.9.3节那样确信需求已经完全清楚了从而构建的快速原型。它更像是一个工程样机，即建造一个按比例缩小的模型以测试建造的可行性。如果开发小组关心提议的软件的某一部分是否可以建造，就可以建造一个概念证明原型。例如，开发者可能关心某一个计算是否能执行得足够快。在这种情况下，他们建造一个原型只测试该计算的时间。或者他们可能担心用于所有屏幕的字体对于大多数用户来说太小了，可能引起视觉疲劳。在这个事例中，他们构建一个原型，显示一些不同的屏幕并通过试验决定是否用户觉得字体太小令他们不舒服。

通过使用原型或其他方法减小风险是**螺旋**（spiral）**生命周期模型**中蕴含的概念[Boehm，1988]。可以简单地将这个生命周期模型看作是每个阶段之前带有风险分析的瀑布模型，如图2-10所示。在开始每个阶段前，努力**减小**（控制）**风险**，如果不能减小该阶段所有重大的风险，则该项目立即停止。

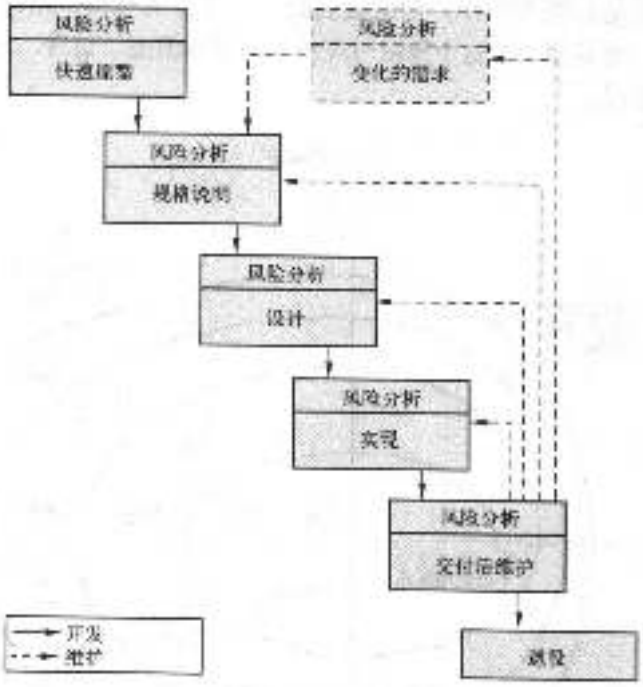



图2-10 螺旋生命周期模型的简版

用原型提供关于某类风险的信息非常有效。例如，时间限制通常可通过构建一个原型并计算原型是否能达到必需的性能来进行测试。如果原型是产品相关特性的准确的功能体现，则在原型上进行的测量应当能够清楚地告诉开发者时间要求能否达到。

其他领域的风险不是原型可以解决的。例如，未招聘到建造产品必需的软件人员，或者项目结束前那个关键人物可能要离职。另一个潜在的风险是个别的开发小组不能胜任开发特

定的大型产品。能够成功建造单个家庭房屋的承包商未必能建造一座复杂的高层写字楼。同样道理，小型软件和大型软件之间有很大的不同，而且原型在测试软件开发小组的能力方面用处也不大。通过在较小的原型上测试开发小组的能力并不能排除这方面的风险，因为在这个原型上显示不出大型软件所特有的小组组织问题。不能利用原型的另一个风险领域是评估硬件供应商的交付承诺。开发者可采取的策略是确定硬件供应商怎样对待以前的客户，但过去的表现并不一定能代表将来。交付合同中的罚款条款是确保主要硬件按时到货的一个途径，但如果硬件供应商拒绝签署带有罚款条款的合同怎么办？即便是有罚款条款，推迟交货还有可能发生，并且最终成为可能拖延数年的法律诉讼行为。在这同时，软件开发者可能因为承诺的硬件未到位造成承诺的软件不能交付，最终破产。总之，尽管原型可以在一些方面帮助降低风险，但在其他方面它可能只能解决部分问题，甚至根本不能解决问题。

图 2-11 是完整的螺旋模型，径坐标代表迄今累积的成本，角坐标代表螺旋形的进展，螺旋的每一圈对应一个阶段。每个阶段开始于（左上的第一象限）确定该阶段的目标、实现这些目标可供选择的办法以及对这些办法的限制条件，这个过程产生达到这些目标的策略。接下来，从风险角度分析这些策略，试图减小每个潜在的风险，有时通过建造原型来减小。如果某个风险不能减小，则项目应立即停止；然而在某些情况下，可以做出继续进行项目的决定，但项目的规模必须明显减小。如果成功减小了所有的风险，则进入下一个开发阶段（右下的第四象限），螺旋模型的这个象限对应传统的瀑布模型。最后，这个阶段的成果经过评估，并计划下一阶段。

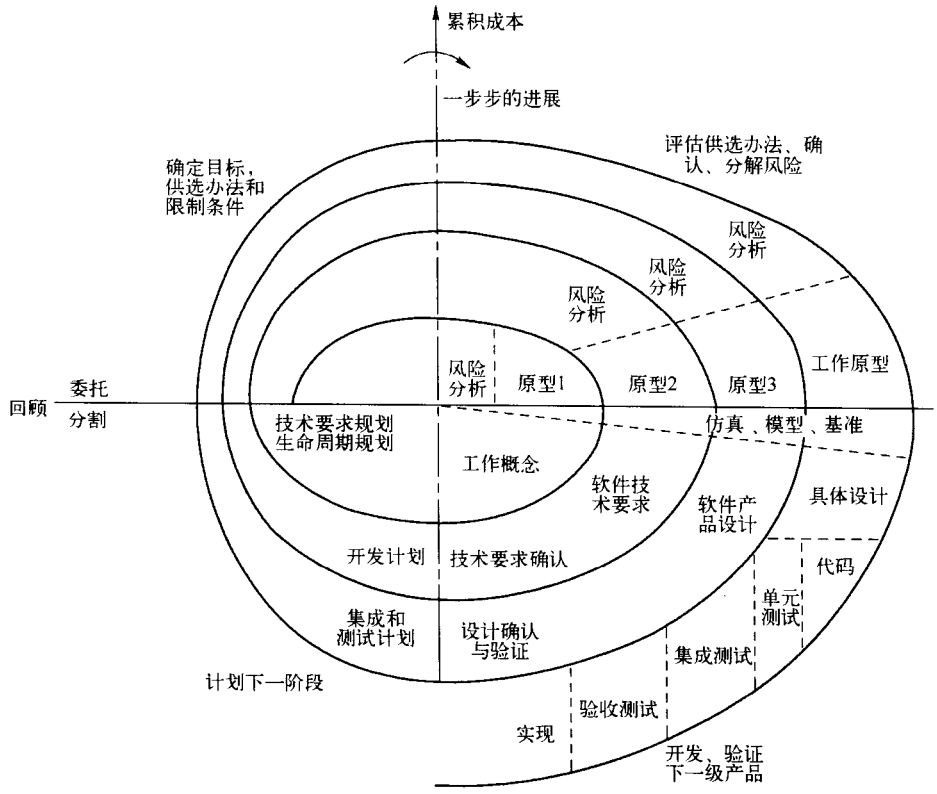



图 2-11　完整的螺旋生命周期模型［Boehm，1988］（©1988，IEEE）

已经使用螺旋模型成功开发出许多种产品，在一批 25 个项目中，螺旋模型与其他提高软件生产率的方法结合使用，与先前的生产效率相比，每个项目的生产率至少提高了 50%，在大部分项目中提高了 100% [Boehm，1988]。为了确定螺旋模型是否适用于某个项目，下面评价螺旋模型的优缺点。

螺旋模型有许多长处，它对选择和限制的强调支持重用现有软件（8.1 节）和把软件质量作为特定的目标结合在其中。另外，软件开发的一个常见问题是确定什么时候已经对某一阶段的产品充分测试完毕。在测试上花费太多时间是一种金钱上的浪费，而且会造成产品交付的过度推迟。相反地，如果测试的时间太少，则交付的软件可能存在残留的错误，给开发者带来不愉快的后果。螺旋模型能根据测试时间太多或太少带来的风险来回答这个问题。在螺旋模型的结构中可能最重要的是，交付后维护只是另一个螺旋周期而已，交付后维护和开发之间没有什么本质上的差别。这样，交付后维护有时受到无知的软件人员轻视的问题不会再出现了，因为交付后维护受到和开发同样方式的对待。

螺旋模型的应用有一些限制，特别地，以它目前的形式，螺旋模型专门用于大型软件的内部开发 [Boehm，1988]。我们来看看一个内部项目，它的开发者和客户都是同一个组织的成员，如果风险分析得出停止该项目的结论，则可以给内部的软件人员重新分配另一个项目。然而，一旦在开发组织和外部的客户之间签署了合同，任何一方试图终止合同都将面临违约诉讼。所以，在签合同软件的情况下，风险分析必须在签署合同前由客户和开发商共同进行，而不是像螺旋模型中那样。

螺旋模型的第二个限制与项目的规模有关，螺旋模型只适用于大型软件。如果风险分析的成本与整个项目成本相当，或者进行风险分析会大大影响潜在的收益，则没有必要进行风险分析。所以，开发者应该确定到底有多少风险和进行多少风险分析。

螺旋模型的一个主要长处是风险驱动，但这也可能是一个短处。除非软件开发者善于指出可能的风险和准确分析风险，否则真正的危险是：某个时候项目小组的人可能认为所有都很正常，而实际上项目正在走向灾难。只有开发小组的成员能够胜任风险分析时，管理者才能决定使用螺旋模型。

但是总的说来，螺旋模型的主要缺点与瀑布模型和快速原型开发模型一样，在于它假定软件是在各个分离的阶段开发的。然而在现实中，软件开发是迭代和递增的过程，这一点可见进化树模型（2.2 节）或迭代－递增模型（2.5 节）。

## 2.10　生命周期模型的比较

前面已经讨论了 8 种不同类型的软件生命周期模型，特别考察了它们的长处和短处。应当避免编码－修补模型（2.9.1 节）。瀑布模型（2.9.2 节）是个已知量，人们了解它的优点，也了解它的缺点。快速原型开发模型（2.9.3 节）是为解决瀑布模型中交付的产品不符合客户的真正要求这个缺点而开发的。然而，这个方法是否在其他方面比瀑布模型优越，人们还知之不多。极限编程（2.9.4 节）是敏捷过程的一个实例，它是有争议的新方法，它只适用于小型软件。同步－稳定模型（2.9.5 节）已被 Microsoft 公司成功地运用过，但还没有其他公司的成功范例。另一种可选方法是使用螺旋模型（2.9.6 节），但条件是开发者在风险分析和风险降低方面受过良好训练。进化树模型（2.2 节）和迭代－递增模型（2.5 节）与现实世界中的软件开发方法最为接近。全面的比较见表 2-1。

表 2-1　本章讨论的各种生命周期模型

| 生命周期模型 | 长　处 | 短　处 |
| --- | --- | --- |
| 进化树模型（2.2 节） | 与现实世界软件开发最接近的模型，与迭代－递增模型类似 | |
| 迭代－递增模型（2.5 节） | 与现实世界软件开发最接近的模型，蕴含统一过程方法 | |
| 编码－修补模型（2.9.1 节） | 适用于不需要任何维护的小程序 | 总的来说不适合重要的程序 |
| 瀑布模型（2.9.2 节） | 纪律性强制的方法，文档驱动 | 交付的产品可能不符合客户的要求 |
| 快速原型开发模型（2.9.3 节） | 确保交付的产品符合客户的要求 | 还没有证明无懈可击 |
| 极限编程（2.9.4 节） | 客户的需求模糊时能很好地工作 | 似乎只适合小规模项目 |
| 同步－稳定模型（2.9.5 节） | 能满足未来用户的要求，确保各组件能够成功集成 | 除了在 Microsoft 公司，还没有被广泛地应用 |
| 螺旋模型（2.9.6 节） | 风险驱动 | 只能用于大型的内部软件产品，开发者必须精通风险分析和风险排除 |

每个软件开发组织都需要为它的组织、它的管理、它的雇员和它的软件过程确定合适的生命周期模型，而且根据当前开发的具体产品的特性改变模型。这样的模型将结合各种生命周期模型的适当特点，扬长避短。

## 本章回顾

理论上的软件开发方式和实践上的软件开发方式有很大不同（2.1 节）。Winburg 小型实例研究用来介绍进化树模型（2.2 节），在 2.3 节中给出了这个小型实例研究的一些教训，特别是需求的变化。在 2.4 节中更深入地讨论了变化，在那里使用野鸭拖拉机小型实例研究给出了移动目标的问题。在 2.5 节中，强调了在现实世界软件开发中迭代－递增模型的重要性，并且给出了迭代－递增模型。在 2.6 节中又重新考察了 Winburg 小型实例研究，说明了进化树模型和迭代－递增模型的相似之处。在 2.7 节中，给出了迭代－递增模型的优点，特别是它能够使我们解决早期面临的风险。然后介绍了一些不同的生命周期模型，包括编码－修补模型（2.9.1 节）、瀑布模型（2.9.2 节）、快速原型开发模型（2.9.3 节）、极限编程和敏捷过程（2.9.4 节）、同步－稳定模型（2.9.5 节）、螺旋模型（2.9.6 节）。在 2.10 节中对这些模型进行了比较，并对具体项目选择生命周期模型提出了建议。

## 进一步阅读

在［Royce，1970］中首次提出瀑布模型，在［Royce，1998］的第 1 章有一段瀑布模型的分析。

关于快速原型开发介绍，推荐的一本书是［Connell and Shafer，1989］。

同步－稳定模型在［Cusumano and Selby，1997］中略有叙述，并在［Cusumano and Selby，1995］中有详细介绍。欲深入了解同步－稳定模型可参考［McConnell，1996］。［Boehm，1988］解释了螺旋模型，而且它在 TRW 软件生产系统中的应用见于［Boehm et al.，1984］。

极限编程在［Beck，2000］中有描述，重组是［Fowler et al.，1999］的主题。《IEEE Software》杂志的 2001 年 11/12 月刊上有一些关于极限编程的文章，特别是［Grenning，2001］和［Paulk，2001］。在［Beck et al，2001］上可以找到《The Manifesto for Agile Software Development》。其他关于敏捷方法的一些看法请见：Highsmith 和 Cockburn［2001］，Boehm［2002］，以及［DeMarco and Boehm，2002］。

在 Ropponen 和 Lyttinen［2000］，Longstaff，Chittister，Pethia 和 Haimes［2000］，以及 Scott 和 Vessey［2002］中描述了风险分析；《IEEE Software》杂志的 1997 年 5/6 月刊上有 10 篇风险管理的文章。

在 Jacobson，Booch 和 Rumbaugh［1999］中详细介绍了一种主要的迭代－递增模型。然而在过去的 30 年里，提出了许多其他的迭代－递增模型，在 Larman 和 Basili［2003］中对此作了统计。在 Goth［2000］中讨论了使用递增模型建造一个空中交通控制系统。在 Bianchi，Caivano，Marengo 和 Visaggio［2003］中给出了一个用迭代方法再工程遗产系统的例子。

许多其他的生命周期模型也已经提出，例如，［Rajlich and Bennett，2000］描述了一个面向维护的生命周期模型。《IEEE Software》杂志的 2000 年 7/8 月刊上有各种关于软件生命周期的文章，包括［Williams，Kessler，Cunningham，and Jeffries，2000］，它描述了一个关于结对编程的试验，一个极限编程的组件。一些关于极限编程的文章出现在《IEEE Software》杂志的 2003 年 5/6 月刊上，包括［Murru，Deias，and Mugheddu，2003］以及［Rasmusson，2003］，它们都描述了使用极限编程开发项目的成功例子。国际软件过程工作室（International Software Process Workshop）的学报是有关生命周期模型的有用的信息来源。ISO/IEC 12207［1995］是软件生命周期过程的一个标准。

## 习题

2.1　使用瀑布模型重新表示 Winburg 小型实例研究。这是否比进化树模型更有效？解释原因。

2.2　假定在 Winburg 小型实例研究中程序员从一开始就使用了单精度数，画出得到的进化树。

2.3　在米勒法则和逐步求精方法之间的联系是什么？

2.4　逐步求精方法对应于迭代或递增方法吗？

2.5　工作流、制品和基准之间是如何关联的？

2.6　在瀑布模型和迭代－递增模型间的联系是什么？

2.7　假设你要建造一个产品来确定 173．847562 的平方根，精确到 5 个小数位。一旦实现并测试了这个产品，就可以弃之不用了。你将使用哪一个生命周期模型？说出理由。

2.8　你是一个软件工程顾问，生产并向餐馆出售各种甜点心的 Deplorably Decadent Desserts 公司的财务副总裁召集你，她想让你的组织建造一个货物控制软件产品，它可以监视公司的生产，这个产品从购买各种配料开始，并跟踪甜点心的加工和向各种餐馆送货。为这个项目选择生命周期模型时你应遵循哪个准则？

2.9　列出开发习题 2.8 中的软件会遇到的风险。你将怎样试图降低每个风险？

2.10　你为 Deplorably Decadent Desserts 公司开发的货物控制软件非常成功，因而该公司想把该产品重编写为 COTS 包，卖给其他准备向餐馆出售食品的公司。所以新产品必须

是可移植的，容易适应新的硬件和操作系统。在为这个项目选择生命周期模型时，你是怎样选择与习题 2.8 不同的标准的？

2.11　描述什么样的产品是螺旋模型的理想应用。

2.12　描述什么情况下螺旋模型不适用。

2.13　（学期项目）你会为附录 A 中描述的 Ophelia's Oasis 的产品使用哪一种软件生命周期模型？请说明理由。

2.14　（软件工程读物）你的导师将分发［Larman and Basili，2003］的复印件。你同意本章中所说的迭代－递增模型是正在进行的大趋势的一部分而不是一个独特的建议的说法吗？

## 参考文献


[Beck, 2000] K. BECK, *Extreme Programming Explained: Embrace Change,* Addison Wesley Longman, Reading, MA, 2000.

[Beck et al., 2001] K. BECK, M. BEEDLE, A. COCKBURN, W. CUNNINGHAM, M. FOWLER, J. GRENNING, J. HIGHSMITH, A. HUNT, R. JEFFRIES, J. KERN, B. MARICK, R. C. MARTIN, S. MELLOR, K. SCHWABER, J. SUTHERLAND, D. THOMAS, AND A. VAN BENNEKUM, "Manifesto for Agile Software Development," agilemanifesto.org, 2001.

[Bianchi, Caivano, Marengo, and Visaggio, 2003] A. BIANCHI, D. CAIVANO, V. MARENGO, AND G. VISAGGIO, "Iterative Reengineering of Legacy Systems," *IEEE Transactions on Software Engineering* **29** (March 2003), pp. 225–41.

[Boehm, 1988] B. W. BOEHM, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* **21** (May 1988), pp. 61–72.

[Boehm, 2002] B. W. BOEHM, "Get Ready for Agile Methods, with Care," *IEEE Computer* **35** (January 2002), pp. 64–69.

[Boehm et al., 1984] B. W. BOEHM, M. H. PENEDO, E. D. STUCKLE, R. D. WILLIAMS, AND A. B. PYSTER, "A Software Development Environment for Improving Productivity," *IEEE Computer* **17** (June 1984), pp. 30–44.

[Booch, 2000] G. BOOCH, "The Future of Software Engineering," keynote address, International Conference on Software Engineering, Limerick, Ireland, May 2000.

[Connell and Shafer, 1989] J. L. CONNELL AND L. SHAFER, *Structured Rapid Prototyping: An Evolutionary Approach to Software Development,* Yourdon Press, Englewood Cliffs, NJ, 1989.

[Cusumano and Selby, 1995] M. A. CUSUMANO AND R. W. SELBY, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People,* The Free Press/Simon and Schuster, New York, 1995.

[Cusumano and Selby, 1997] M. A. CUSUMANO AND R. W. SELBY, "How Microsoft Builds Software," *Communications of the ACM* **40** (June 1997), pp. 53–61.

[DeMarco and Boehm, 2002] T. DEMARCO AND B. BOEHM, "The Agile Methods Fray," *IEEE Computer* **35** (June 2002), pp. 90–92.

[Fowler et al., 1999] M. FOWLER WITH K. BECK, J. BRANT, W. OPDYKE, AND D. ROBERTS, *Refactoring: Improving the Design of Existing Code,* Addison Wesley, Reading, MA, 1999.

[Goth, 2000] G. GOTH, "New Air Traffic Control Software Takes an Incremental Approach," *IEEE Software* **17** (July/August, 2000), pp. 108–11.

[Grenning, 2001] J. GRENNING, "Launching Extreme Programming at a Process-Intensive Company," *IEEE Software* **18** (November/December 2001), pp. 27–33.

[Highsmith and Coburn, 2001] J. HIGHSMITH AND A. COCKBURN, "Agile Software Development: The Business of Innovation," *IEEE Computer* **34** (September 2001), pp. 120–22.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle Processes," International Organization for Standardization, International Electrotechnical

Commission, Geneva, Switzerland, 1995.

[Jacobson, Booch, and Rumbaugh, 1999] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Information System Development Process,* Addison Wesley, Reading, MA, 1999.

[Larman and Basili, 2003] C. LARMAN AND V. R. BASILI, "Iterative and Incremental Development: A Brief History," *IEEE Computer* **36** (June 2003), pp. 47–56.

[Li and Alshayeb, 2002] W. LI AND M. ALSHAYEB, "An Empirical Study of XP Effort," *Proceedings of the 17th International Forum on COCOMO and Software Cost Modeling,* Los Angeles, October 2002.

[Longstaff, Chittister, Pethia, and Haimes, 2000] T. A. LONGSTAFF, C. CHITTISTER, R. PETHIA, AND Y. Y. HAIMES, "Are We Forgetting the Risks of Information Technology?" *IEEE Computer* **33** (December 2000), pp. 43–51.

[McConnell, 1996] S. MCCONNELL, "Daily Build and Smoke Test," *IEEE Software* **13** (July/August 1996), pp. 144, 143.

[Miller, 1956] G. A. MILLER, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review* **63** (March 1956), pp. 81–97; reprinted in: www.well.com/user/smalin/miller.html.

[Murru, Deias, and Mugheddu, 2003] O. MURRU, R. DEIAS, AND G. MUGHEDDU, "Assessing XP at a European Internet Company," *IEEE Software* **20** (May/June, 2003), pp. 37–43.

[Paulk, 2001] M. C. PAULK, "Extreme Programming from a CMM Perspective," *IEEE Software* **18** (November/December 2001), pp. 19–26.

[Rajlich and Bennett, 2000] V. RAJLICH AND K. H. BENNETT, "A Staged Model for the Software Life Cycle," *IEEE Computer* **33** (July 2000), pp. 66–71.

[Rasmusson, 2003] J. RASMUSSON, "Introducing XP into Greenfield Projects: Lessons Learned," *IEEE Software* **20** (May/June 2003), pp. 21–29.

[Ropponen and Lyttinen, 2000] J. ROPPONEN AND K. LYTTINEN, "Components of Software Development Risk: How to Address Them? A Project Manager Survey," *IEEE Transactions on Software Engineering* **26** (February 2000), pp. 96–111.

[Royce, 1970] W. W. ROYCE, "Managing the Development of Large Software Systems: Concepts and Techniques," *1970 WESCON Technical Papers, Western Electronic Show and Convention,* Los Angeles, August 1970, pp. A/1-1–A/1-9; reprinted in: *Proceedings of the 11th International Conference on Software Engineering* (Pittsburgh, May 1989), IEEE Computer Society Press, Los Alamitos, CA, pp. 328–38.

[Royce, 1998] W. ROYCE, *Software Project Management: A Unified Framework,* Addison Wesley, Reading, MA, 1998.

[Scott and Vessey, 2002] J. E. SCOTT AND I. VESSEY, "Managing Risks in Enterprise Systems Implementations," *Communications of the ACM* **45** (April 2002), pp. 74–81.

[Spivey, 1992] J. M. SPIVEY, *The Z Notation: A Reference Manual,* Prentice-Hall, New York, 1992.

[Tomer and Schach, 2000] A. TOMER AND S. R. SCHACH, "The Evolution Tree: A Maintenance-Oriented Software Development Model," in: *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering* (CSMR 2000), Zürich, Switzerland, February/March 2000, pp. 209–14.

[Williams, Kessler, Cunningham, and Jeffries, 2000] L. WILLIAMS, R. R. KESSLER, W. CUNNINGHAM, AND R. JEFFRIES, "Strengthening the Case for Pair Programming," *IEEE Software* **17** (July/August 2000), pp. 19–25.

# 第3章 软件过程

**学习目标**

学完本章之后，你应当能够：

- 解释为什么二维生命周期模型是重要的；
- 描述统一过程的五个核心工作流；
- 列出在测试工作流中测试的制品；
- 描述统一过程的四个阶段；
- 解释统一过程的工作流和阶段之间的差别；
- 理解软件过程改进的重要性；
- 描述能力成熟度模型 CMM。

软件过程是我们生产软件的方式。它包括方法学（参见 1.11 节）和它背后隐含着的生命周期模型（第 2 章）、技术、我们所使用的工具（参见 5.4 节至 5.10 节），以及所有这些因素中最重要的因素：建造这些软件的人。

不同的软件组织有不同的软件生产过程。以文档的问题为例，一些软件组织认为他们生产的软件本身就是文档；即他们认为仅仅通过阅读源代码，人们就能了解产品。而有的组织的文档特别多，他们按时编制规格说明文件，并对文件进行系统的审查；然后，他们不辞辛苦地进行设计活动，在开始编码前对设计一遍又一遍地检查，并为程序员提供每一个代码制品的详细说明；测试用例是预先计划好的，将测试结果记录成测试日志，并将测试数据仔细归档；产品交付并安装在客户的计算机上后，在做任何改变之前都要提交书面报告，并且详细列出改变的原因。要实行这些改变必须得到书面授权，直到文档更新过了，以及对文档的修改获得批准后，修改才能集成到产品中去。

测试的强度是软件组织之间进行比较的另一个度量。有些组织将全部软件经费的一半用于软件测试。但是有的人却认为只有用户才能对软件进行完全的测试，所以，有的公司在产品测试上投入很少的时间和精力，而在解决客户发现的产品问题上花费了大量的时间。

交付后维护是许多软件组织的首要任务。有些软件使用了 5 年、10 年甚至 20 年，但人们仍在对它们进行不断地修改、提高以满足客户的需求。此外，有的软件在成功地维护了若干年之后，还会有残留的错误。几乎所有的软件组织每隔 3 到 5 年都会将其软件移植到较新的硬件上，这也是维护的一部分。

相反，有的软件组织在本质上对研究非常关心，而把开发交给别人去做，更不要说维护了。这种情况特别适合大学的计算机科学系，在那里，研究生们通过开发软件来证明某种设计和技术是可行的，然后将商业开发留给其他软件组织。（关于不同的软件组织开发软件的各种方法，请见下面的“如果你想知道”部分）

然而，不管确切的程序如何，软件开发过程围绕着图 2-4 所示的五个工作流构建：需求流、分析流（规格说明）、设计流、实现流和测试流。本章描述了这些工作流，同时还阐述在每个工作流期间可能产生的潜在的挑战。对这些挑战的解决以及软件生产本身通常事关重要。本书余下的部分致力于描述适合的技术。在本章的第一部分，只强调了挑战，但引导读

者到相应的章节去寻求解决办法。因此，本章的这一部分不仅是软件过程的一个概述，也是学习本书后文的一个指导。在这一章结束的时候，会介绍国内外在改进软件过程方面的一些创见。

**如果你想知道**

为什么不同的软件开发组织之间的软件开发过程会有如此大的差异？主要的原因是由于软件工程技能的缺乏。有太多软件专业人员没有跟上时代的要求，他们不懂得别的方法，继续使用古老的方法进行软件开发。

软件开发过程有很大差异的另外一个原因是，许多软件管理人员虽然是优秀的管理人员，但是他们对软件开发和维护却一窍不通。技术知识的缺乏使得项目开发一再延期，造成项目不能继续进行。这些常常是许多软件项目不能完成的原因。

开发过程中存在差异的另一个原因是管理观点。例如，有的组织认为按期交付产品比较重要，即使不对产品进行充分测试也没有关系。但同样情况下，另外一个组织就会认为不对产品进行充分测试的风险要大于不按期交付产品的风险，即这个组织认为即使推迟交付产品，也要对产品进行充分测试。

我们现在考察统一过程。

## 3.1 统一过程

如这一部分开始所说，方法是软件过程的一个组成部分。今天最主要的面向对象方法是**统一过程**，就像在下面的“如果你想知道”部分中所说的那样，统一的“过程”实际上是一种方法，但是“统一方法”这个名字已经被用作**统一建模语言**（Unified Modeling Language，UML）的第一版的名字。目前已不再支持统一过程的三个先驱（OMT，Booch 的方法以及对象）了，并且其他的面向对象方法曾经有很少或没有后继者。结果，统一过程今天通常是面向对象软件最主要的选择。幸运的是，如本书第 2 部分将要说明的那样，统一过程几乎在所有的方面都是一个极好的面向对象方法。

**如果你想知道**

直至最近，最流行的面向对象软件开发方法是对象建模技术（object modeling technique，OMT）[Rumbaugh et，al.，1991] 和 Grady Booch 的方法 [Booch，1994]。OMT 是由位于纽约 Schenectady 的通用电器研发中心的 Jim Rumbaugh 和他的小组开发的；Grady Booch 在加利弗尼亚 Santa Clara 的 Rational 公司开发了他的方法。全部面向对象软件开发方法本质上是相同的，因此 OMT 和 Booch 方法之间的差别并不大。尽管如此，在这两个阵营的支持者之间总有一些友好的竞争。

在 1994 年 10 月，这种情况改变了，当时 Rumbaugh 加入了 Booch 在 Rational 的小组。两个方法的创始者随即开始一同工作，开发了一种将 OMT 和 Booch 方法结合在一起的方法。在他们发表第一阶段的工作成果时，他们指出并没有开发一种方法，只是提出一个表示面向对象产品的符号。“统一方法”的名字很快变成“统一建模语言”（UML）。在 1995 年，面向对象方法的创始人 Ivar Jacobson 又加入了 Rational 的工作组。人们将 Booch、Jacobson 和 Rumbaugh 亲切地称为“三位老友”（Three Amigos）（取自 1986 年 John Landis 的电影名），这三人立即联合工作，于 1997 年推出 UML 的第一版，在软件工程领域刮起一股风暴。在那之前，还没有开发出软件

产品所一致接受的符号。几乎一夜之间，全世界都在使用 UML。对象管理组（Object Management Group，OMG）是一个对象领域世界上顶尖公司的联合会，负责制定 UML 国际标准，以使每个软件专业人员能够使用同一版本的 UML，从而推动一个组织内的专业人员之间，以及全世界的公司间的交流和沟通。今天，UML［Booch，Rumbaugh，and Jacobson，1999］已成为无可争辩的表示面向对象软件产品的国际标准符号。

管弦乐队乐谱显示需要用哪些乐器演奏一个乐章、每一件乐器要演奏的音符以及什么时候演奏它，同时还有整个技术信息的载体，如音调符号、拍子、响度等。这个信息能用自然语言而不用五线谱的形式给出吗？也许可以，但不可能根据这样的描述演奏出音乐来。例如，钢琴家和小提琴家无法演奏一个如下描述的乐章："音乐是进行曲式，在 B 小调的键盘中。第一个把位从小提琴中央 C 上的 A 开始（四分音符）。当这个音符开始演奏的时候，钢琴家弹一个由 7 个音符组成的和弦。右手弹奏下面 4 个音符：中央 C 后面的 E……"

显然，在某些领域，文本描述不能简单代替图示。音乐就是一个这样的领域，软件开发是另一个。而且对于软件开发，在今天，最好的建模语言显然是 UML。

用 UML 给软件工程界带来风暴对于这三剑客来说还不够。他们下一个努力的目标是提出一个完整的软件开发方法，该方法将他们三人的三个独立的方法统一起来。这个统一的方法最初称为 Rational 统一过程（Rational Unified Process，RUP），该方法取名字 Rational，是因为这三个人在当时都是 Rational 公司的高级管理人员（Rational 在 2003 年被 IBM 买下）。在他们有关 RUP 的书中［Jacobson，Booch，and Rumbaugh，1999］，使用了统一软件开发过程这个名字（Unified Software Development Process，USDP）。为简便起见，今天通常使用术语"统一过程"。

统一过程并不是具体的一系列步骤，无法做到如果按此操作就可以构建一个软件产品。事实上，不存在一个这样的"普遍适用"的方法，因为软件产品的类别具有广泛的多样性。例如，有许多不同的应用域，如保险、航空以及制造业。而且，赶在竞争对手之前匆忙将一个 COTS 软件包推向市场的方法与用来构建一个高度安全的电子资金转账网络的方法是不同的。此外，软件专业人员的技能又是千差万别的。

因此，统一过程应被视为一种自适应的方法学。也就是说，它要根据具体要开发的软件产品进行修改。如我们将在第 2 部分要看到的那样，统一过程的某些特性不适合小型甚至中型软件。然而，统一过程的大部分可用于各种规模的软件产品。本书的重点放在统一过程的这个公共子集上，但是也讨论统一过程适用于大型产品的特性，以确保当构建大型软件产品时涉及的问题能够得到透彻理解。

## 3.2　面向对象范型内的迭代和递增

面向对象范型到处使用模型。**模型**是一套 UML 图表，它表示要开发的软件产品的一个或多个特性（UML 图表在第 7 章中介绍）。我们知道，UML 代表统一建模语言，即 UML 是我们用来表示（模拟）目标软件产品的工具。使用像 UML 图形表示的原因如古老的中国谚语所说：百闻不如一见。UML 图表使软件专业人员相互之间能够更快速和准确地沟通。

面向对象范型是一个迭代和递增方法。每个工作流由一些步骤组成，为了完成该工作流，重复执行工作流的步骤直至开发小组成员满意地认为，他们已经有了一个软件产品的精确的 UML 模型。也就是说，即使是最有经验的软件人员也要不断迭代直至他们满意地认为

UML图表是正确的。这其中的含意是，无论多么优秀的软件工程师，几乎没有第一次就将各种工作产品都顺利通过的。怎么会这样呢？

软件产品的特性是实际上每件产品都要迭代和递增地开发。毕竟软件工程师是人，因而服从米勒法则（2.5节）。即，不可能在同一时间考虑所有事情，因此开始只能处理7个左右的程序块（信息的单位）。然后，当考虑下一组程序块时，获得了更多的关于目标软件产品的知识，根据这个增加的信息UML图表得到修改。这个过程以这种方式继续下去，直至最终软件工程师满意地认为用于给定工作流的全部模型都是正确的。换句话说，根据在工作流开始得到的知识画出初始最可能的UML图表。然后，随着更多有关被建模的现实世界系统的知识的获得，图表做得更准确（迭代）并得到扩展（增量）。这样，不管一个软件工程师如何经验丰富和技术熟练，他（她）重复地迭代和递增，直至满意地认为UML图表是要开发的软件产品的准确表示。

理想状态下，在本书结束的时候，读者会学到为构建大型、复杂软件产品而进行的统一过程开发所必需的软件工程技巧。遗憾的是，由于以下几个原因这恐怕难以实现：

1）就像不可能通过一门课程成为微积分或一门外语的专家一样，精通统一过程需要额外的研究以及更重要的，在面向对象软件工程方面无止境的实践。

2）统一过程最初是为开发大型、复杂软件产品而开发的。为了能够处理这样的软件产品的诸多细节，统一过程本身较庞大。很难在一本这样篇幅的书中覆盖统一过程的每个方面。

3）为了讲授统一过程，必须要给出一个实例研究，解释说明在应用到大型软件产品时的特性，这样的实例研究必须是较大的。例如，仅仅是规格说明通常就需要1000页。

出于这三个原因，本书给出大多数而不是全部的统一过程。

下面讨论统一过程的五个**核心工作流**（需求流、分析流、设计流、实现流以及测试流）以及存在于其中的挑战。

## 3.3 需求流

软件开发是一个昂贵的过程。当客户认为一个软件能使他（她）的企业获利或认为该项目经济上是划算的，为该软件产品找到一个开发组织，开发过程通常就开始了。**需求流**的目标是让开发组织确定客户的需求。

开发小组的第一个任务是对**应用领域**（简称为**域**）获得一个基本的了解，即，将要运行目标软件的特定的环境。这个域可以是储蓄、汽车制造或核物理。

在过程的任何阶段，如果客户不再相信该软件产品的代价是合理的，开发将会立即停止。在本章中我们假定客户认为代价是合理的，因此，软件开发的一个关键特性是**商业模型**，它是一个说明目标产品代价合理的文档。（事实上，代价不总是经济上的，例如，军事软件经常是为了战略或战术目的而建造的，这里，软件的代价是不开发该型武器带来的潜在的危险。）

在客户和开发者之间举行的初次会谈中，客户按照他们头脑中的概念描述出产品。从开发者的观点来看，客户对产品的描述可能是模糊的、不合理的、矛盾的或干脆只是不可能实现的。在这个阶段，开发小组的任务是准确确定客户的需求并从客户的角度找出存在的限制

条件。

- 一个主要的限制几乎总是**最终期限**。例如，客户可能规定最终产品必须在 14 个月内完成。在几乎每个应用域，目标软件产品都是任务紧急的，这已司空见惯了。这也就是说，客户将在他们的核心活动中使用该软件产品，在交付目标产品中的任何拖延都将对客户造成损害。
- 经常有各种其他限制，如**可靠性**（例如，产品必须在 99% 的时间里是可工作的，或者平均故障间隔时间最小为 4 个月）。另一个常见的限制是可执行载入映像的大小（例如，它必须在客户的个人计算机上或在卫星上的硬件中运行）。
- **成本**几乎总是一个重要的限制条件。然而，客户很少告诉开发者有多少钱可用于建造该产品。而常见的做法是，一旦规格说明定下来了，客户让开发者给出完成该产品的价格。客户根据这个进行投标程序，希望开发者的要价低于客户已经为该项目留出的预算。

最初对客户需求的调研有时称为**概念探究**（concept exploration）。在客户小组和开发小组随后的会谈中，对提出的软件产品的功能不断提炼并分析技术上的可行性和经济上的合理性。

到目前为止，每件事情看来都很简单。遗憾的是，需求流经常完成得不好。当产品最终交付给用户时，可能是在客户签署了规格说明文档之后的一年或两年了，客户可能对开发者说，“我知道这是我要求的，但它真的不是我想要的。”客户所要求的并不是客户真正需要的。造成这种困境的原因有很多。首先，客户可能不真正了解在他（她）的组织内部正在进行的事情。例如，如果当前缓慢运转的原因是数据库设计得较差，那么向软件开发者要求一个更快的操作系统是无济于事的。或者，如果客户运营的是一家不盈利的零售连锁店，客户想要一个财务管理信息系统，能够反映出诸如销售、工资、财务支出和财务收入等情况。如果亏损的真正原因是货品丢失（被雇员偷拿或被顾客偷拿），那么这样一个产品将不会有什么用处。如果是这种情况，需要的是一个货物控制系统而不是财务管理信息系统。

客户常常提出错误的产品需求，产生这种情况的主要原因是软件的复杂性。如果软件人员难于将一部软件以及它的功能形象化，对于具有较少计算机知识的客户来说，问题可能要更糟。我们将在第 10 章看到，统一过程在这方面会有所帮助，许多统一过程的 UML 图表可帮助客户深入理解他需要的软件产品。

## 3.4　分析流

**分析流**的目标是分析和提取需求，以获得正确开发一个软件产品和易于维护它所必需的需求。然而，乍一看，分析工作流好像不是很必要。一个明显简单的过程是通过连续进行需求流的交互直至获得对目标产品的必要理解来开发一个软件产品。

关键是需求流的输出必须能够完全被客户所理解。换句话说，需求流的制品必须用客户的语言表达，即用像英语、阿米尼语或祖鲁语这样的自然语言表示。但是所有的自然语言，都毫无例外地在某种程度上不精确，会引起人们的误解。例如，看看下面这段话：

从数据库中读零件记录和设备记录，如果它包含其后紧随字母 Q 的字母 A，则计算将那个零件转移到那个设备的成本。

初看起来，这个需求似乎非常清楚。但有一点，“它”指什么：零件记录，设备记录，

或者数据库?

如果这个需求是用（比如说）一种数学符号表示的，这种模糊就不会产生。然而，如果将数学符号用于需求分析，那么客户就很可能会不理解大部分需求。结果是，在客户和开发者间对需求存在很大的理解差异，因而，开发出的软件产品满足了那些需求，但却不是客户想要的。

解决的办法是有两个独立的工作流。需求流用客户的语言表达，分析流则用更精确的语言，以确保设计流和实现流正确地完成。此外，在分析流期间加入更多的细节，细节与客户对目标软件产品的理解不直接相关，但对于软件开发者来说却至关重要。例如，状态图（12.6节）的初始状态与客户没有一点关系，但是如果开发者要正确建造目标产品的话，必须将它包含在规格说明中。

产品的规格说明文档构成了产品的合同。软件开发人员交付软件产品时，如果该产品满足规格说明文档所述的产品验收标准，则认为软件开发人员完成了产品合同。由于这个原因，软件规格说明文档不应当包含那些不严密的词语，比如“合适”、“方便”、“充分”或“足够”等类似词语，这些词语听起来精确，但实际上是模糊的，就像“最优”或“98%完成”一样。与此同时，签订软件开发合同会导致法律诉讼，当客户和开发人员属于同一组织的时候，规格说明文档是不能成为法律诉讼的依据的。但是即便是在内部软件开发的情形下，也应当编写规格说明文档，以便将来出现麻烦时用作证据。

更重要的是，规格说明文档对于测试和维护来说都是必需的。如果规格说明文档不精确，我们就不能确定规格说明是否正确，更不用说产品的实现是否满足规格说明的要求了。如果没有一个文档准确地告诉我们当时的规格说明是什么，那么在维护阶段修改规格说明是件非常困难的事情。

当使用统一过程的时候，没有通常意义上的规格说明文档。而是向客户展示一组UML制品，像第12章中所描述的那样。这些UML图表及其描述能够避免许多（但不是全部）传统规格说明文档的问题。

传统分析小组可能犯的一个错误是规格说明模糊，像前面解释的那样，**模糊**是自然语言固有的特性。**不完备**是规格说明的另一个问题，即，可能忽略了一些相关的事实或需求。例如，如果输入数据中有错，那么规格说明文档可能不会指明要采取什么行动。更有甚者，规格说明文档可能是**矛盾的**。例如，有一个控制发酵过程的产品，在该产品的规格说明文档里的某一个地方指出，如果压力超过35psi，则必须立即关闭阀门M17。然而，在文档的另外一处指出，如果压力超过35psi，则立刻向操作员报警，仅仅当操作员在30秒钟内没有采取补救行动时，阀门M17才自动关闭。在上述规格说明方面存在的问题得到纠正之前，软件开发过程不能够继续。如前面段落中指出的那样，可以通过使用统一过程减少这些问题的发生。这是因为UML图表与对这些图表的描述不太可能产生模糊、不完备和矛盾。

当客户批准了规格说明之后，就要开始进行详细计划和评估。没有客户会在事先不知道软件开发进度和软件成本的情况下就授权进行软件开发的。从开发人员的角度来看，上述两个问题也是同样重要的。如果软件开发人员低估了项目开发的成本，那么客户只会支付他们同意支付的费用，客户支付的费用可能比软件开发实际需要的费用要低很多。相反，如果开发人员过高估计了软件项目的成本，那么客户就会推翻这个项目，或者让其他估价更合理的开发者来做这个项目。在项目的开发周期的估计上也有相似的问题。如果开发人员低估了完

成一个项目所需的时间，那么产品不能按时交付，这样最起码会导致客户降低对开发人员的信任。最糟糕的情况是，产品不能按时交付将致使客户援引合同中的延期惩罚条款，这样开发人员需要为此付出经济上的代价。如果开发人员高估了软件开发所需的时间，客户同样会把项目交给那些承诺能够用更短时间开发出软件的团队。

对开发人员来说，仅仅对软件总的成本和开发时间进行评估是远远不够的，开发人员需要将不同的人分配到开发过程的不同工作流中去。例如，在 SQA 小组通过有关的设计制品之前，实现小组不能够开始工作；同样，在分析小组完成任务之前，设计小组是不需要开始工作的。换句话说，开发者必须事前计划。必须制定软件项目管理计划（software project management plan，SPMP），它反映开发过程各个独立的工作流并显示在每个任务中开发组织的哪些人员需要介入，同时还要规定每项任务的完成时间。

人们可以开始制定详细计划的最早时间是在规格说明文档完成的时候。在这之前，与项目有关的各个方面还没有完全定下来，所以还不能够开始制定详细计划。对项目的某些方面，必须从一开始就对它们正确地计划，但在开发人员确切地知道他们要建造什么之前，他们不可能制定项目各个方面的所有计划。

因此，当客户批准了规格说明文档之后，就可以开始准备制定软件项目管理计划了。该计划的主要组成部分有：**可交付的东西**（客户将要得到什么），**里程碑**（客户什么时候可以得到它们），以及**预算**（它要花费多少钱）。

计划尽可能详细地描述了整个软件过程。它包括下述方面：要使用的生命周期模型，开发组织的组织结构，项目职责，管理目标和优先权，使用的技术和 CASE 工具，以及详细时间表，预算和资源分配。整个计划的实质是对开发周期和成本的估计，9.2 节对进行上述评估的技术进行了阐述。

第 11 章和第 12 章阐述了分析流：第 11 章对传统分析技术进行了阐述，第 12 章的主题是面向对象分析。分析流中的一个主要制品是软件项目管理计划。在 9.3 节到 9.5 节中给出了对于如何制定 SPMP 的解释。

现在考察设计流。

## 3.5　设计流

产品的规格说明清楚地指出了产品要做什么。设计指示产品如何做。更准确地说，**设计流**的目标是细化分析流的制品，直至材料处于程序员可实现的形式。

如 1.3 节所说，在传统设计阶段，设计小组确定产品的内部结构。设计人员将产品分解成**模块**，它是与产品其他部分有明确定义的接口的独立代码段。必须详细定义每个模块的接口（即传递给模块的参数和从模块返回的参数）。例如，一个模块可能测量一个原子反应堆中的水平面，如果水平面太低则会报警；一个航空电子产品中的一个模块，可能将来袭的敌方导弹的两组或多组坐标数值作为输入，计算它的轨迹，然后调用另一个模块，建议飞行员采取可能的避让行动。设计小组完成模块化分解之后（**结构化设计**），开始实施**详细设计**，为每个模块选择相应的算法和数据结构。

现在转向面向对象范型，范型的基础是**类**，类是一种特殊类型的模块。在分析流期间提取类并在设计流期间设计它。因而，与结构化设计对应的面向对象的对应物是作为面向对象分析流的一部分来执行的，详细设计的面向对象对应物是面向对象设计流的一部分。

设计小组必须详细记录他们所做的每个设计决定。做这样的记录是基本的要求，有两个原因：

1）在进行产品设计时，有时会走到死胡同，这样设计小组需要原路返回重新进行设计。书面记录下做出具体决定的原因，在这种情况发生时能够对设计小组有帮助，帮助他们原路返回。

2）理想的产品设计应当是可修整的（open-ended），“可修整”的意思是将来可通过添加新的类或取代已存在的类来提高产品的性能（交付后维护），同时在整体上不影响设计。当然，这个理想状态在实践中很难实现。在现实世界中，设计人员争分夺秒，赶在任务期限到来之前完成满足最初的规格说明要求的产品设计，这时设计人员并不关心以后的产品改进问题。如果将来的产品改进问题（在产品交付给客户后加入）包含在规格说明中，那么在设计阶段就必须对这个问题进行考虑，但是这种情况很少见。通常，规格说明文档以及由此产生的设计，仅仅涉及当前的需求。另外，当产品仅仅处于设计阶段时，没法确定将来所有可能的改进。最后，如果在设计中把将来所有的可能性考虑进去，好的话，这个设计会很不实用；糟的话，该设计会由于过于复杂而不能实现。所以，设计人员应当对设计进行折衷，使其既能在合理的方面扩展又不需要全部重新设计。但是，在一个经受大的改进的产品中，其设计尚不能处理进一步的改变而时间已经到了。当到这一步时，则整个产品必须进行重新设计。进行重新设计的小组如果有原先做设计决策时的记录，他们的工作会更容易。

## 3.6 实现流

**实现流**的目标是用选择的实现语言实现目标软件产品。小型软件产品有时由设计者实现。而大型软件产品被划分为较小的子系统，它们由多个编码小组并行实现。相应地，子系统由**组件**或**代码制品**组成，它们是分别由单个程序员实现的。

通常，交给一个程序员仅有的文档是有关的设计制品。例如，在传统范型的情况下，交给程序员的是他（她）要实现的模块的详细设计。详细设计通常提供程序员实现代码制品所需的详细信息。如果有什么问题，他们可以通过询问负责的设计人员从而迅速弄清楚问题。然而，单个程序员无法知道结构化设计是否正确。仅当开始集成各个代码制品时，设计的缺陷才开始作为一个整体显现出来。

假定已经实现和集成了一些代码制品，并且到现在为止集成的产品部分似乎在正常工作。进一步假定一个程序员已经正确地实现了制品 a45，但是当这个制品与其他现存软件制品集成时，该产品不能正常工作。失败的原因不在 a45 本身，而在于制品 a45 与产品的其他部分的接口，该接口是在结构化设计中规定的。尽管如此，编码了制品 a45 的程序员将为此失败受到指责。这是不公平的，因为该程序员只是简单地遵照设计员提供的指导，如该制品的详细设计所描述的那样实现了制品。该编程小组的成员很少得知“大图”，即结构化设计，更不要说征求对它的意见了。尽管期望一个程序员意识到某个制品作为一个整体对产品的影响是非常不公平的，但不幸的是这种情况在实际中太常发生了。这也是为什么重要的是设计在每个方面都要正确的另一个原因。

设计（以及其他制品）的正确性被作为测试流的一部分进行检查。

## 3.7　测试流

如图 2-4 所示，在统一过程中，从始至终测试是与其他工作流并行进行的。测试的主要特性有两方面：

1）每个开发者和维护者都要负责确保他（她）的工作是正确的。因此，软件人员要对他（她）开发或维护的每个软件制品进行测试或再测试。

2）一旦软件人员确信一个制品是正确的，就将它交给软件质量保证小组进行独立测试，如第 6 章所述。

**测试流**的性质随着被测试的制品的不同而不同。然而，对所有制品都至关重要的一个特性是可追踪性（traceability）。

### 3.7.1　需求制品

如果需求制品在软件产品的整个生命周期将是可测试的，那么它们必须具有的一个属性是**可追踪性**。例如，必须能够对分析制品中的每一项追踪到需求制品，对于设计制品和实现制品也一样。如果需求说明有章可循、编号正确、前后参照并带有索引，那么，开发者应当能轻松地通过后来的制品向前追踪并确保它们是客户需求的真实反映。当 SQA 小组检查需求小组成员的工作时，可追踪性也简化了他们的工作。

### 3.7.2　分析制品

如第 1 章中指出的那样，交付软件中一个主要的错误来源是规格说明中的错误，直至软件产品安装到客户的计算机上，并且由客户所在组织依照自己的用途使用该产品时才发现它们。因此，分析小组和 SQA 小组必须一丝不苟地检查分析制品。此外，他们必须保证该规格说明是可行的。例如，所有规定的硬件组件的速度要足够快，或者客户当前的联机硬盘存储容量要能够适合新产品的运行。一个极好的检查分析制品的方法是通过评审（review）。评审会中，分析小组和客户双方的代表都到场，会议通常由 SQA 小组的成员主持。评审的目的是确定分析制品是否正确，评审人员审查分析制品，检查其中是否存在错误。走查（walkthrough）和审查（inspection）是两种不同的评审方式，6.2 节对此进行了阐述。

现在考虑对详细计划和估算的检查，它发生在客户签署了规格说明之后。尽管先由开发小组然后再由 SQA 小组仔细地检查 SPMP 计划的各个方面是最基本的，必须对计划的周期估算和成本估算给予特别的关注。这样做的一个办法是让管理者在详细的计划开始之前得到两个（或多个）独立的关于时间和成本的估算，然后调和它们之间明显的不同。与检查分析制品一样，检查 SPMP 文档的最好方法是评审。如果产品完成时间估算和产品成本估算令人满意，那么客户将允许项目继续下去。

### 3.7.3　设计制品

如 3.7.1 节提到的那样，可测试性的一个关键方面是可追踪性。在设计的情况下，这意味着设计的每个部分都可以与分析制品联系起来。一个前后有适当参照的设计为开发者和 SQA 小组提供了有力工具，使他们能够检查产品设计是否与规格说明吻合，以及规格说明

中的每一部分是否能在产品设计中有所反映。

设计评审与规格说明的评审相似。然而，考虑到大多数设计具有的技术特性，客户通常不参加评审。设计小组和SQA小组成员从整体上走查设计，同时也走查每个独立的设计制品，以确保设计是正确的。查找的错误类型包括逻辑错误、接口错误、缺少异常处理（处理错误情况），以及最重要的，不符合规格说明。另外，评审小组应当时刻注意某些可能在前面工作流中没有查出来的分析错误。6.2节详细描述了评审过程。

### 3.7.4 实现制品

每个组件在实现的同时应当对它们进行测试（桌面测试），并且在实现之后要对它们运行测试用例。这个非正式的测试是由程序员做的，在此之后由质量保证小组对组件进行系统测试，这称为**单元测试**。第14章阐述了各种单元测试技术。

除了运行测试用例，代码评审也是检查编程错误的一个强有力的、成功的技术。这里，程序员引导评审小组成员检查组件清单。评审小组成员必须包括SQA小组的代表。此评审过程与前面讲述的规格说明和设计的评审相似，像所有其他工作流中一样，要保存SQA小组的活动记录，作为测试流的一部分。

对一个组件进行编码后，必须将它与其他编码后的组件组合起来，以便SQA小组能够确定是否该（部分）产品整体上功能是正确的。组件集成的方式（一次全部或一次一个）以及具体的顺序（在组件互连图或类层次中的自顶向下或自底向上）会对最终产品的质量有重要影响。例如，假设产品自底向上集成。如果产品中有一个重要的设计错误，那么该错误由于这种自底向上的集成方式而会推迟出现，这样造成的重新设计会付出比较昂贵的代价。相反，如果组件自顶向下集成，那么底层组件通常不会像在自底向上方式中那样得到完全测试。第14章中对这些问题和其他问题进行了详细阐述。第14章对诸如为什么编码和集成要并行地进行等问题也给出了详细说明。

**集成测试**的目的是检查组件是否正确组合在一起，以实现满足规格说明要求的产品。在集成测试期间，对组件接口的测试必须格外小心。形参的数量、顺序和类型必须与实参的数量、顺序和类型相匹配，这一点很重要。编译器和链接器能很好地进行这种强制类型检查[van Wijngaarden et al., 1975]，然而，许多语言不进行强制类型检查，使用这种语言时，接口检查必须由SQA小组来做。

当集成测试完成时（即当全部组件都编码和集成完毕），SQA小组进行**产品测试**。依照规格说明对产品功能进行整体测试。特别是，要对规格说明中列出的约束条件进行测试。一个典型的例子是判断产品响应时间是否合适。因为产品测试的目的是确定是否规格说明正确实现了，因而在规格说明完成后，就可以开始编制大多数测试用例了。

不仅必须测试产品的正确性，还需要测试产品的健壮性。即，故意将错误的输入数据提供给产品，来确定产品是否会崩溃，或者是否产品的错误处理能力足够应付这些有问题的数据。如果产品将要与客户当前已经安装了的软件一起运行，那么必须测试新产品对客户计算机已有软件是否有不良影响。最后，必须检查源代码和所有其他类型的文档是否全部完成，并且是否具有内在一致性。14.21节对产品测试进行了讨论。根据产品测试的结果，开发组织的高层管理者决定是否准备将产品交付客户。

集成测试的最后一个方面是**验收测试**。软件交付给客户，客户使用与测试数据不同的真

实数据，在实际的硬件上对产品进行测试。无论开发小组或 SQA 小组测试得多么系统全面，测试数据和真实数据之间仍存在很大的不同，毕竟测试数据本质上是人工编制的。软件产品如果不通过验收测试，则不能称其满足了规格说明文档的要求。14.22 节给出了与验收测试有关的更多细节。

在 COTS 软件（1.11 节）的开发中，产品测试一经完成，该版本的产品就提供给那些挑选出来的潜在客户，由他们对其进行现场测试。第一个版本通常称为**α 版**，经过修改的 α 版通常称为**β 版**，通常 β 版最接近此类软件产品的最后版本。（术语 α 版和 β 版通常用于所有类型的软件产品，不只是 COTS。）

COTS 软件中存在错误通常导致的结果是：软件的销量很低，从而给开发公司造成了极大的损失。为了尽可能早并尽可能多地发现错误，COTS 软件开发人员常常将 α 版和 β 版软件交给挑选出来的公司，以此希望在现场测试中发现软件的一些潜在错误。作为报答，开发商常常向测试站点许诺为其提供交付版的免费拷贝。这对参加 α 版或 β 版测试的公司来说存在一定风险，特别是，α 测试版中会有许多错误，从而影响工作、浪费时间，以及可能毁坏数据库。然而，对于首先使用新的 COTS 软件的公司来说，这样做会使得该公司比他们的竞争对手在使用该软件方面有更大的优势。当软件开发组织用那些潜在客户提供的 α 版测试代替 SQA 小组提供的产品完全测试时，有时会出现问题。虽然在大量不同现场进行的 α 版测试通常会暴露出很多问题，但是，α 版测试并不能取代 SQA 小组提供的系统测试。

## 3.8　交付后维护

交付后维护并非是当产品交付并安装到客户的计算机上之后才勉强进行的一项活动。相反，它是软件过程整体的一个组成部分，必须从软件开发的一开始就对它进行计划。像 3.5 节所说的那样，可行的设计应当把将来对产品的改进考虑进去。进行编码的时候也应当把将来对产品的维护考虑进去。如 1.3 节所指出的那样，毕竟交付后维护要比其他的软件活动加起来花费的成本还要多，因而维护是软件生产的一个重要方面。人们不应当事后才想起维护。正确的是，在进行整个软件开发工作时，应当时刻遵循一个原则：尽量减小将来不可避免要产生的维护工作带来的影响。

交付后维护中存在的普遍问题是文档问题，或者说是文档的缺乏。在依照最后交付时间进行的软件开发过程中，最初的分析和设计制品常常没有更新，因而它们对维护小组来说几乎没有用途。别的文档，例如数据库手册和操作手册则可能从来就没有编写，因为管理层认为按时向客户交付产品比同时进行文档开发要重要得多。在许多例子中，源代码是维护人员手中惟一的文档。软件产业人员的高速流动使得软件维护的状况更加糟糕，在需要对产品进行维护的时候，某组织的早期开发人员已经全部都不为该组织工作了。由于上述原因，交付后维护常常是软件生产中最具挑战性的阶段，第 15 章将介绍其他一些原因。

现在来看测试，当执行交付后维护时，测试对产品所做的改变有两个方面。第一个是检查要求的改变已经正确实现了。第二个是确保在对产品做要求的改变时，不做其他无意识的改变。因此，在程序员确定要求的改变已经完成后，必须用各种测试用例对产品进行测试，以确保产品其他的功能不受影响。这个步骤称为**回归测试**。为了做好回归测试，先前所有的测试以及运行这些测试用例的结果有必要保留。第 15 章中将深入讨论交付后维护期间的测试。

交付后维护的一个主要方面是记录所做的全部改变，同时还有做这些改变的原因。当软

件改变时，必须对它进行回归测试。因此，回归测试用例是文档的主要内容。

## 3.9 退役

软件生命周期的最后一个阶段是**退役**。在软件使用了若干年之后，当进一步的交付后维护已经不值得时，软件就到达了退役阶段。

- 有时，对产品要做的改变太大，以至于产品的整体设计都需要变化。这种情况下，对整个产品重新设计和重新编码的费用可能比改变它的费用要小一些。
- 对初始设计进行了如此多的改变，以至于无意中在产品内部构成了相互依赖性，这种相互依赖甚至使得对产品最小模块的较小改变都会对产品的整体功能有重大影响。
- 文档维护工作做得不充分，增加了退化错误的风险，以至于对产品重新编码比对产品进行维护要更加安全。
- 产品运行所需的硬件（和操作系统）要更新换代；重新编写该软件比对软件进行修改要更经济。

在上述任何一种情况下，新版软件将替代旧版软件，并且软件过程将继续下去。

从另一方面说，真正的退役，即产品失去了它的可用性，这是一种极少发生的事件。客户组织不再需要产品所提供的功能，最终将其从计算机中清除掉。

## 3.10 统一过程的各阶段

图 3-1 与图 2-4 的不同之处在于递增的标签改变了。它们不是标为递增 A、递增 B 等，而是将四个递增标为：开始阶段、细化阶段、构建阶段和转换阶段。换句话说，统一过程的阶段与递增的各个阶段相对应。

尽管理论上开发一个软件产品可能会经过许多次递增，实际上的开发看来通常由四个递增构成。下面四个小节将描述递增或阶段，以及每个阶段可交付的东西，即在每个阶段结束时应当完成的制品。

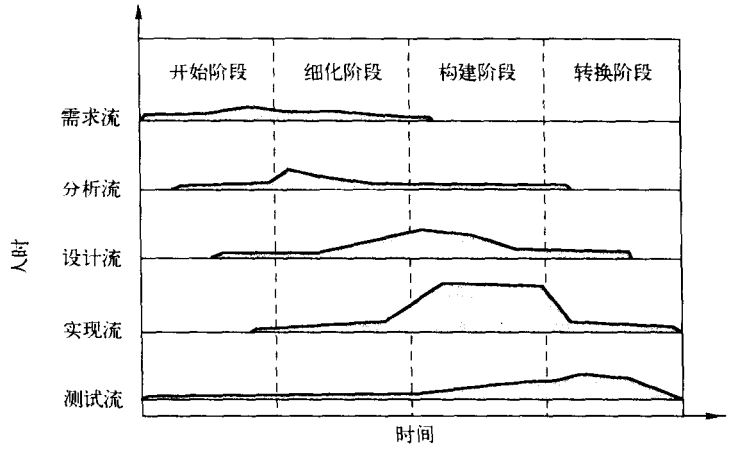



图 3-1 核心工作流和统一过程的各个阶段

统一过程中执行的每一个步骤属于五个核心工作流之一，也属于四个阶段之一，这四个阶段是：开始阶段，细化阶段，构建阶段和转换阶段。3.3 节到 3.7 节已经描述了这四个阶

段的各个步骤。例如，建造一个商业模型是需求流的一部分（3.3 节），它也是开始阶段的一部分。尽管如此，如我们将要解释的那样，每个步骤还是要再一次讨论。

就拿需求流来说，为了确定客户的需求，其中的一个步骤是建造商业模型。换句话说，在需求流的框架内，在技术范畴内提出建造一个商业模型。在下一小节，在开始阶段的框架内，给出建造一个商业模型的描述，在该阶段中，管理者决定是否开发提议中的软件产品。即是说，在经济范畴内，立即建造出商业模型（1.2 节）。

与此同时，再次在同一个细节层次上介绍每个步骤没有什么意义。因此，下面深入地描述了开始阶段，以便强调各工作流的技术层面之间，以及各阶段的经济层面之间的不同。而其他三个阶段则只做简要介绍。

### 3.10.1　开始阶段

**开始阶段**的目标是决定是否值得去开发目标软件产品。换句话说，这个阶段最主要的目标是明确提出的软件产品是否经济上是可行的。

需求流的两个步骤是理解问题域并建造一个商业模型。显然，如果软件开发人员不首先对要开发的软件产品的领域有一个理解，他们是无法就未来可能的软件产品给出任何意见的。不管涉及的领域是电信网络、机器工具公司还是专门治疗肝病的医院，如果开发人员没有完全理解该领域，就不会对他们要开发的产品有多大的把握。因此，第一个步骤是获得该领域的知识。在开发者对该领域有了完全的理解之后，第二个步骤就是建造一个商业模型。换句话说，首先需要的是理解问题域本身，第二个需要的是清楚地理解客户组织是如何在该领域中运作的。

现在必须限定目标产品的范围。例如，考虑为全国范围内的连锁银行建造一个用于新的高度安全的 ATM 网络的软件产品。连锁银行的商业模型整体看来规模非常大，要确定目标软件产品中应当包含些什么，开发者应当仅集中精力于商业模型中的一个子集，称为提议中的软件产品涵盖的子集。因此，限定提出项目的范围是第三个步骤。

现在开发者可以开始制定最初的商业案例了。在开始进行项目前需要回答的问题包括[Jacobson, Booch, and Rumbaugh, 1999]：

- 建议的软件产品会有经济效益吗？也就是说，从开发软件产品中获取的经济利益是否会超过软件开发中消耗的成本？开发建议的软件产品所进行的投资多长时间会见到效益？或者说，如果客户决定不开发建议的软件产品，其代价是什么？如果软件产品将要在市场上销售，进行必要的市场调查了吗？
- 建议的软件产品能够按时交付吗？即，如果该软件产品晚交付市场，该组织是否仍会得到收益，或者一个竞争的软件产品是否会占领市场的大部分份额？或者说，如果要开发该软件产品支持客户组织自身的活动（可能包括一些任务紧急的活动），如果建议的软件产品晚交付，影响会怎样？
- 开发软件产品会带来哪些风险，如何降低这些风险？将要开发提议中的软件产品的小组成员具有必需的经历吗？软件产品需要新的硬件吗？如果需要，在交付时间上有风险吗？如果有风险，有没有办法可能通过从另一个供货商处订购备份硬件来降低这些风险？需要软件工具（第 5 章）吗？目前能得到它们吗？它们有全部需要的功能吗？可能在该项目正在开发时，一个具有全部（或几乎全部）提议的客户软件

产品功能的COTS软件包（1.11节）将投放市场，这该如何处理？

在开始阶段的结尾，开发人员需要回答这些问题，以便能够制定最初的商业案例。

下一个步骤是明确风险。有三类主要的风险：

1）技术风险。技术风险的例子刚才已经列出。

2）没有得到正确的需求。可以通过正确地执行需求流降低这个风险。

3）没有得到正确的体系结构。体系结构可能不够健壮（从2.7节中我们知道，软件产品的体系结构由各种组件组成，它的健壮性是指如何将它们组装到一起以及能够处理意外和变化而不至于瘫痪）。换句话说，当软件产品正在开发时，试图向已经开发的软件中加入新的代码块，可能需要从头重新设计整个体系结构，因而存在风险。一个类似的情况是搭建一个纸牌房子，当整个房子搭完后，再另外加入一张纸牌只能造成房子摇晃倒地。

风险需要分等级，以便首先降低重要的风险。

如图3-1所示，在开始阶段进行少量的分析流的工作。通常做的是提取设计体系结构所需的信息。这个设计工作也反映在图3-1中。

现在再转到实现流，在开始阶段，常常不进行任何编码工作，然而，偶尔有必要建立一个概念证明原型，用来测试提议的软件产品的某些部分的可行性，见2.9.6节的介绍。

测试流始于开始阶段之初。这里主要的目标是保证准确地确定需求。

计划是每个阶段的一个基本组成部分。在开始阶段的情况下，开发人员在该阶段的开始没有足够的信息来计划整个开发，因此在项目开始仅有的计划是计划开始阶段本身。基于同样的原因，由于信息的缺乏，在开始阶段结尾可以做的有意义的计划是仅仅为下一阶段做计划，即细化阶段。

文档也是每个阶段的一个基本组成部分。开始阶段可交付的内容有［Jacobson，Booch，and Rumbaugh，1999］：

- 域模型的初始版本；
- 商业模型的初始版本；
- 需求制品的初始版本；
- 分析制品的初步版本；
- 体系结构的初步版本；
- 风险的初始清单；
- 初始用例（见第10章）；
- 对细化阶段的计划；
- 商业案例的初始版本。

获得最后一项，商业案例的初始版本，是开始阶段全部的目标。这个初始版本结合了软件产品范围的描述，以及经济上的细节。如果提出的软件产品将要投放市场，商业案例应包括收入预计、市场估计以及初步的成本估计。如果该软件产品要内部使用，商业案例应包括初步的成本效益分析（见5.2节）。

### 3.10.2 细化阶段

**细化阶段**的目标是细化最初的需求，细化体系结构，监视风险和细化它们的属性，细化

商业案例，以及生成软件项目管理计划。命名为“细化阶段”的原因很明显，这个阶段的主要活动是对前一阶段工作的细化。

图 3-1 显示这些任务几乎与以下各阶段密切相关：完成需求流（第 10 章）、实质上执行整个分析流（第 12 章）以及然后开始结构设计（8.5 节）。

细化阶段的可交付内容包括 [Jacobson, Booch, and Rumbaugh, 1999]：

- 完成的问题域模型；
- 完成的商业模型；
- 完成的需求制品；
- 完成的分析制品；
- 体系结构的更新版本；
- 风险的更新清单；
- 软件项目管理计划（对于项目的余下部分）；
- 完成的商业案例。

### 3.10.3 构建阶段

**构建阶段**的目标是产生软件产品的第一个可工作版本，也称为测试版（β 版）（3.7.4 节）。再看一下图 3-1，虽然该图只是各阶段的符号表示，显然在这个阶段强调的是实现和测试此软件产品。即，编码各组件并进行单元测试。然后编译代码制品并对其进行链接，从而构成子系统，对它进行集成测试。最后，将子系统组合成整个系统，对它进行产品测试。在 3.7.4 节中讨论了这方面内容。

构建阶段的可交付产品包括 [Jacobson, Booch, and Rumbaugh, 1999]：

- 初始用户手册和其他相关手册；
- 全部制品（测试版）；
- 完成的体系结构；
- 更新的风险清单；
- 软件项目管理计划（用于该项目的其余部分）；
- 必要时，更新商业案例。

### 3.10.4 转换阶段

**转换阶段**的目标是确保客户的需求切实得到满足。这个阶段由来自安装了软件测试版的各现场的反馈来驱动（对于为特定用户开发客户软件产品的情形，只有一个这样的现场）。在这个阶段中，软件产品中的错误得到纠正。而且，完成全部的手册。在这个阶段中，重要的是发现全部先前没有认识到的风险（即使在转换阶段也要揭示风险的重要性在“如果你想知道”部分中得到强调）。

**如果你想知道**

实时系统常常比多数人甚至是它的设计者想像的要复杂得多。结果是，有时在各软件组件间发生的微小的交互，即使是最有经验的测试者也很难察觉到。一个看起来微小的改变可能会带来严重的后果。

这方面一个有名的例子是由于一个差错而延迟了 1981 年 4 月的第一次空间飞行器在轨飞

行 [German, 1981]。空间飞行器电子系统由4台完全同步的计算机控制。另外，在4台计算机组成的计算机组出现故障时，独立的第5台计算机可以备份工作。在此前两年，对电子系统计算机同步执行初始化的一个软件模块做了改变，这个改变带来的一个负面影响是包含时间的记录稍微比当前时间晚了一点，该记录被错误地发送到用于对航空电子计算机初始化的数据区。发送的时间与实际时间非常接近，这个差错没有被检测到。大约1年以后，时间差稍有增加，但出错概率只有1/67。然后，在第一次空间飞行器发射的当天，在全世界上亿人通过电视观看的情况下，同步错误出现了，4台完全一致的计算机中的3台比另一台同步时间晚了一个时间周期。一台在不一致情况下阻止第5台计算机接收来自其他4台计算机的信息错误保护设备工作了，它出人意料地阻止第5台计算机同步，发射被迫推迟了。这个事件的主要差错出在初始化模块，这个模块表面上看来与同步子程序无论如何也不会有联系。

遗憾的是，这并非影响空间飞行器发射的最后一次差错。例如，在1999年4月，一颗Milstar军事通信卫星发射落入一条无用的低轨道，造成12亿美元的损失。事故的原因是大力神4系列火箭上升段的一个软件差错 [Florida Today, 1999]。

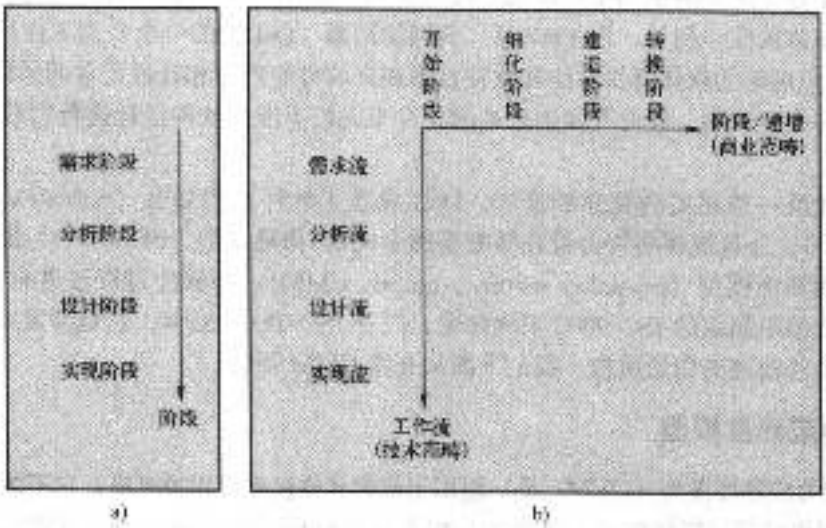
不仅仅是太空发射受实时系统差错的影响，航天器着陆也同样会受到影响。在2003年5月，从国际空间站发射Soyuz TMA－1宇宙飞船，经弹道轨迹降落后，偏离300英里返回哈萨克斯坦。着陆问题出现的原因再一次是由于实时软件的差错 [CNN. com, 2003]。

转换阶段可交付的产品包括 [Jacobson, Booch, and Rumbaugh, 1999]：

- 全部制品（最终版）；
- 完成的手册。

## 3.11 一维与二维生命周期模型

传统生命周期模型（如2.9.2节的瀑布模型）是一维模型，用图3-2a中的单轴坐标表示。统一过程涵盖的是一个二维生命周期模型，如图3-2b中的两轴坐标表示。




a) b)

图3-2 比较a）传统一维生命周期模型和b）二维统一过程生命周期模型

瀑布模型的一维特性在图 2-3 中清楚地表现出来。相反，图 2-2 显示 Winburg 小型实例研究的进化树模型。这个模型是一个二维模型，应当将其与图 3-2b 做一个比较。

二维模型带来的额外复杂因素有必要吗？答案在第 2 章中给出，但这是一个重要的问题，有必要在这里重复一下。当开发一个软件产品时，在理想情况下，需求流应当在着手进行分析流之前完成。同样，分析流应当在开始设计流之前完成，如此等等。然而在现实情况下，几乎最微不足道的软件产品也是非常大的，不能当作一个单个的单元处理。相反，必须将任务划分为递增（阶段），在每个递增内开发者需要不断迭代，直至完成构建的任务。作为人类，我们受到米勒法则［Miller，1956］的限制，该法则声明我们每次只能有效地处理 7 个概念。我们因此无法将软件产品作为一个整体来处理，但是我们却能够把那些系统划分为一些子系统。有时甚至子系统也过于庞大——只有在对软件整体有了完全的理解之后，才能处理软件的某些组件。

统一过程是迄今为止将大型问题作为一些较小、具有较独立子问题的子集解决的最好办法。它提供了递增和迭代的一个框架，这个机制用于解决大型软件产品的复杂性。

统一过程较好地处理的另一个挑战是无法避免的改变。这个挑战的一个方面是在产品开发过程中客户需求的改变，它被称为移动目标问题（2.4 节）。

由于所有这些原因，统一过程是当前可用的最好方法，然而，将来，统一过程将无可置疑地被一些后来的新方法所取代。今天的软件专业人员正在统一过程之外寻找下一个重大突破。毕竟，在人类所为之努力的每一个领域，今天的发现常常超越先前提出的所有发现。统一过程同样也注定要被未来的方法所超越。重要的是要记住：基于今天的理解，统一过程似乎比当前可用的其他方法要好。

本章余下的部分介绍着眼于过程改进的一些新的创见。

## 3.12　改进软件过程

我们的全球经济严重依赖于计算机，进而依赖于软件。因为这个原因，许多国家的政府对软件过程非常关注。例如，在 1987 年，美国国防部（DoD）的一个特别工作组报告，“过去 20 年，在应用新的软件开发方法和软件技术提高软件生产力和软件质量的承诺落空之后，工业和政府部门认识到，软件开发的根本问题在于人们不能对软件过程进行管理”。［Brooks et al.，1987］

作为对此及一些相关的关注的反应，DoD 成立了软件工程协会（Software Engineering Institute，SEI），并且选择将协会设在匹兹堡的卡内基·梅隆大学。SEI 的一个重要的成功是建立了能力成熟度模型（capability maturity model，CMM）。与软件过程改进有关的努力包括国际标准化组织制定的 ISO 9000 系列标准，以及 ISO/IEC 15504，它是涉及 40 多个国家的一个国际软件改进的创始机构。我们下面从介绍 CMM 开始。

## 3.13　能力成熟度模型

SEI 的能力成熟度模型（CMM）是一组用于改进软件过程的相关策略，它不考虑实际使用的软件生命周期模型（**成熟度**这个概念是过程本身良好程度的度量）。SEI 开发的 CMM 模型有，用于软件的（SW－CMM）；用于人力资源管理的（P－CMM；P 代表“人”）；用于系统工程的（SE－CMM）；用于集成产品开发的（IPD－CMM）；用于软件获取的（SA－CMM）。在

不同的模型间有不一致的地方，并且不可避免地还有某种程度上的冗余。因而，在1997年，人们决定开发一个集成的成熟度模型框架，能力成熟度模型集成（CMMI）。CMMI将5个现存能力成熟度模型集成进了一个模型。将来可以向CMMI中加入另外的一些约束［SEI，2002］。

因为篇幅原因，这里我们仅仅研究SW－CMM成熟度模型。SW－CMM模型在1986年由Watts Humphrey［Humphrey，1989］提出。回忆一下软件过程所包含的用于软件生产的各种活动、技术和工具。我们知道，软件过程所包含的内容包括技术方面和管理方面的。SW－CMM模型基于如下观点而来：新的软件技术本身并不能导致产量和利润的增加，我们的问题主要出现在软件过程管理上。SW－CMM模型的策略是改进软件过程的管理，相信技术的提高则是一个自然的结果。软件过程作为一个整体所获得的改进将导致产生较高质量的软件，并且将会有较少的软件项目超时或超支。

记住，软件过程改进不可能在一夜之间实现。SW－CMM促使变化不断增加。更特别的是，定义了5个成熟度级别，一个软件组织通过每一步的细微演变，将自己的成熟度级别提高到更高一级上［Paulk，Weber，Curtis，and Chrissis，1995］。为了对模型有更好的理解，下面分别对每个级别进行描述。

**成熟度级别1：初始级** 这是最低级别，在这样的组织里，有效的软件过程管理方法本质上没有获得使用。取而代之的是，每件事都在一个特别的基础上进行。若由有竞争力的软件管理人员和一个优秀的软件开发小组来开发某个具体的项目，则项目可能会成功。然而，通常的情况是由于有效管理和特殊计划的缺乏，造成软件开发的超时和超支。因而，许多措施都是在软件开发遇到困难的时候采取的，而不是事先计划好的。在成熟度级别为1的组织里，软件开发过程是不可预计的，因为这样的软件开发过程完全依赖于当前的软件开发人员，当开发人员调动或离开时，软件过程也跟着发生变化。这样的结果使得，对软件开发所需要花费的时间和金钱进行精确估计成为一件不可能的事情。

遗憾的是，世界上大多数的软件组织的软件开发能力成熟度级别仍处于1级。

**成熟度级别2：可重复级** 这个级别使用了基本的软件项目管理措施。根据类似产品的经验对新的产品进行计划和管理。从而这个级别的名字是：可重复级。在级别2，要进行测量工作，它是将一个过程充分实现的第一步。典型的测量包括对花费和工作进度表的仔细追踪。与级别1仅仅在软件开发过程出现问题的时候才采取措施相反，管理人员能及时发现问题，并立刻采取纠正措施阻止这些问题演化成大的危机。问题的关键在于，如果不进行测量工作，在问题变得不可控制之前我们不可能发现这些问题。并且，一个项目中的测量工作能为以后项目的时间和费用表的制定提供现实依据。

**成熟度级别3：定义级** 级别3的过程有充分的软件生产文档。软件开发过程在所有的管理和技术方面都有明确定义，并在任何可能的地方不断努力改进软件开发过程，用评审的（6.2节）方式来保证软件质量。在这个级别，人们引进像CASE（5.6节）这样的新技术来进一步提高软件质量和软件生产力。相反，在危机驱动的级别1，“高技术”只能引起更大的混乱。

虽然有些软件开发组织的级别达到了级别2和级别3，但是达到级别4和级别5的几乎没有。后两个最高的级别成了软件组织未来的努力目标。

**成熟度级别4：可管理级** 级别4的组织为每个项目设计了质量目标和生产目标。在软件开发过程中对这两项指标不断测量，当与目标有不可接受的偏离时，则采取措施对其进行纠正。设立统计质量控制［Deming，1986；Juran，1988］，确保管理者能够对质量和生产标

准的随机偏离及有意图的违背有所判断（一个统计质量控制测量的简单例子是每1000行代码检测出的错误数，相应目标是经过一段时间减少其数量）。

**成熟度级别5：最优级**　5级组织的目标是持续改进软件过程。人们用统计质量和过程控制技术对软件组织进行指引。在每个项目中获取的知识在以后的项目中得到利用。开发过程形成了一个反馈性的良性循环，从而使软件生产和软件质量获得不断提高。

图3-3对这5个级别进行了总结。软件组织为了提高自己的软件过程，首先需要努力对组织当前的软件过程有一个理解。然后对想要获得的软件过程进行明确地阐述。接着确定为实现过程提高要采取的措施，并确定实行措施的先后顺利。最后，制定一个实现过程提高的计划，并实施该计划。随着组织软件过程的不断提高，对这一系列步骤不断进行重复。图3-3反映了级别与级别之间的进步。能力成熟度模型的经验表明，提高一个完整的成熟度级别需要花费18个月到3年的时间。但是从1级到2级有时候需要花费3—5年的时间。这说明，如果一个组织到目前为止，其软件过程仍是很低级的，那么在此基础上要向该组织灌输一个科学的方法是多么困难。

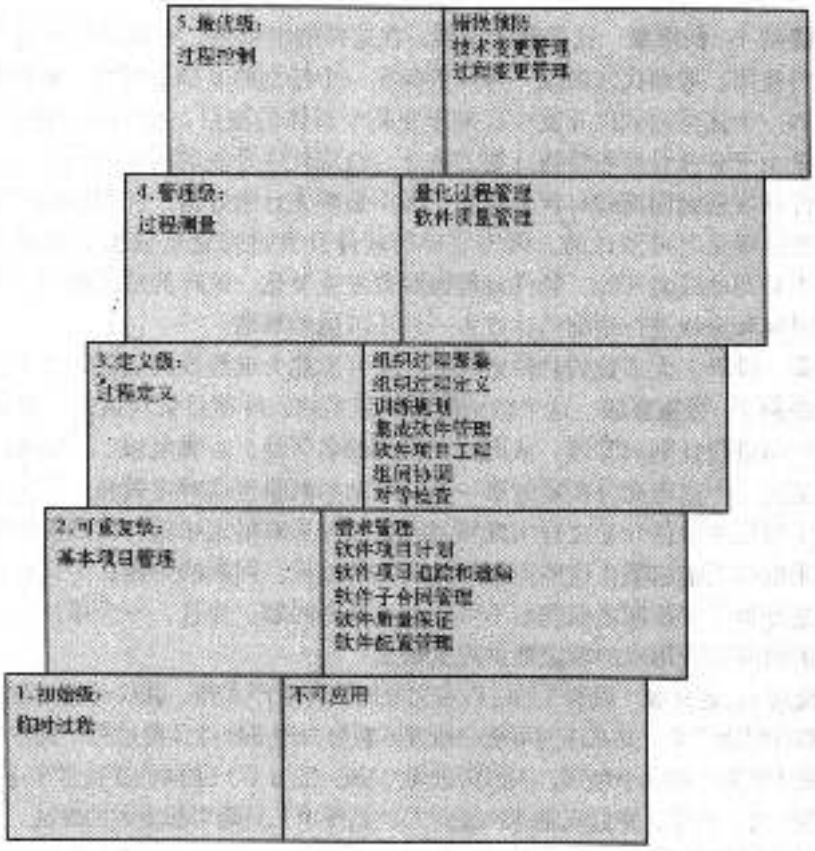



图3-3　5级能力成熟度模型以及它们的关键过程区

SEI为每个成熟度级别示出了一系列的**关键过程区**（key process areas，KPA）。关键过程区是一个组织在迈向下一个级别时要努力实现的目标。例如，级别2（可重复级）的KPA包括配置管理（5.8节）、软件质量保证（6.1.1节）、项目计划（第9章）、项目追踪

(9.2.5节)和需求管理(第10章)。以下内容是项目管理的基础要素:确定客户需要(需求管理),制定计划(项目计划),监督与计划不一致的地方(项目追踪),控制构成软件产品的不同部分(配置管理),以及确保产品没有错误(质量保证)。每个KPA里是一组2到4个相关目标,如果实现了这些目标,也就实现了下一个成熟度级别。例如,一个项目计划目标,就是一个开发计划,该计划真实并恰当地描述了软件开发的一切活动。

在最高级,成熟度5级,KPA包括错误预防、技术变更和过程变更管理。与2级的KPA相比,5级明显地比2级的要高级。例如,2级组织的软件质量保证是,发现并纠正错误(软件质量在第6章详细讨论)。相比之下,5级组织的软件开发过程包括错误预防,这样首先就确保了软件没有错误。为了帮助软件组织达到更好的成熟度级别,SEI开发出了一系列调查表,该调查表是SEI小组的评估基础。评估的目的是指出软件组织当前软件过程中的不足,并且指明软件组织改进其过程应采用的方法。

软件工程协会的CMM计划是由美国国防部发起的。CMM计划最初的目标之一是,对为美国国防部生产软件的承包商的软件过程进行评估,并与那些开发过程较成熟的承包商签订合同,以此提高国防软件的质量。美国空军曾规定,到1998年为止,任何想与空军签合同的软件开发组织必须达到SW-CMM模型3级水平,美国国防部接着就发布了一个类似的命令。这给软件组织提高软件过程成熟度增加了压力。然而,SW-CMM模型已经远远超出了提高国防部软件过程的有限目的,现在许多软件组织执行该模型,希望以此提高自己的软件质量和生产力。

## 3.14 软件过程改进方面的其他努力

**国际标准化组织**(ISO)的9000系列标准是用于提高软件质量的另一种尝试,它是应用于广泛的工业活动的5个相关标准,这些活动包括设计、开发、生产、安装和服务。ISO 9000当然不是一个软件标准,在ISO 9000系列中,为质量系统设立的ISO 9001标准[ISO 9001,1987]是最适合于软件开发的标准。因为ISO 9001内容较多,ISO发布了具体条款,辅助ISO 9001应用于软件:ISO 9000-3[1991]。(有关ISO的更多信息,请参见1.3.1节中的"如果你想知道"部分。)

ISO 9000有许多特征与CMM模型不同[Dawood,1994]。ISO 9000强调使用文字和图片对整个过程进行记录,以确保整个过程的一致性和易理解性。并且,ISO 9000的原则是,与该标准保持一致并不能保证生产出高质量的产品,但是这样可以减少生产出质量低劣的产品的可能。ISO 9000仅仅是质量体系的一部分。它在下述方面也有要求:质量委托管理,工人集中培训,设立持续质量提高的实现目标。

ISO 9000系列标准在20多个国家获得应用,包括美国、日本、加拿大和欧盟(EU)。这意味着,例如,如果美国的软件组织希望与欧洲客户签订合同,美国的组织必须首先通过ISO 9000的认证。一个有资格的注册员(审核师)必须对公司的软件生产过程进行检查,并且证明其软件生产过程符合ISO标准。

紧随欧洲同行其后,越来越多的美国公司开始要求ISO 9000认证。例如,通用电器公司的塑料部坚持要求其340个供货商在1993年6月之前通过该标准[Dawood,1994]。美国政府不会效仿欧共体的做法,对那些希望与美国公司做生意的非美国公司要求ISO 9000认证。尽管如此,来自美国内部及其主要贸易伙伴的压力将最终会导致世界范围的认可并执行ISO 9000。

像 ISO 9000 一样，ISO/IEC 15504 是一个国际性软件过程提高的先驱。这个方法先前称为 SPICE，它是 Software Process Improvement Capability dEtermination（软件过程提高能力测定）的缩写，有 40 多个国家实际使用了 SPICE。SPICE 由英国国防部（Ministry of Defence，MOD）提出，其长远目标是将 SPICE 建立成一个国际标准（MOD 是英国的一个部门，其作用与美国国防部类似，后者发明了 CMM）。SPICE 的第一个版本在 1995 年完成，在 1997 年 7 月，SPICE 由国际标准化组织委员会和国际电工学会（ISO/IEC）接手。由于这个原因，SPICE 改名为 ISO/IEC 15504，或简称 15504。

## 3.15　软件过程改进的代价和收益

实现了软件过程改进就一定能创造出更大的效益吗？初步结果表明，事实上是这样。例如，位于加利福尼亚 Fullerton 的休斯飞机公司的软件工程部门，在 1987 年到 1990 年间为过程评估和提高程序共投入了 50 万美元［Humphrey，Snider，and Willis，1991］。在这 3 年的时间内，休斯飞机公司的成熟度级别从 2 级升到了 3 级，并希望将来升到 4 级甚至 5 级。作为过程改进的结果，休斯飞机公司估计每年能节省 200 万美元。成本的节省可在各个方面体现出来，包括不断减少的超时时间，更少出现的软件危机，提高了的雇员士气，以及软件专业人员流动性的降低。

别的组织也有类似可比结果的报道。例如，位于 Raytheon 的装备部从 1988 年的级别 1 增加到了 1993 年的级别 3，生产力增加了 2 倍，用于过程改进的每一美元能获得 7.7 美元的回报［Dion，1993］。这样的结果造成，能力成熟度模型在全球软件工业中获得了相对来说非常广泛的应用。

例如，印度的 Tata 咨询服务机构使用 ISO 9000 框架和 CMM 来改进它的过程［Keeni，2000］。在 1996 年至 2000 年之间，估计软件工作中的差错从 50% 下降到 15%。检查的效果（即在检查期间发现错误的百分比）从 40% 增加到 80%。用于返工项目的工作百分比从接近 12% 下降到小于 6%。

摩托罗拉的政府电子部（GED）从 1992 年开始，积极地参加了 SEI 的软件过程改进计划［Diaz and Sligo，1997］。表 3-1 描述了 34 个 GED 项目，这些项目都是根据每个项目开发小组的能力成熟度模型来进行分类的。从图中可以看到，软件项目的开发时间（与 1992 年前项目开发的最长时间限制相比）随着成熟度级别的增加而减少。采用 MEASL（Million Equivalent Assembler Source Lines 每百万行等价汇编源代码）对软件中的错误数进行度量。为了对不同语言完成的项目进行比较，源代码的行数被转化成了等价的汇编程序的行数［Jones，1996］。如表 3-1 所示，产品质量随着能力成熟度的增加而增加。最后，用每人时的 MEASL（每人每小时编写的等价汇编源代码的百万行数）对生产力进行度量。为保密起见，摩托罗拉没有发布实际的生产数据，因而，表 3-1 表明的是与成熟度为 2 级的项目有关的生产情况。（因为级别 1 的小组没有进行定量测量，因此级别 1 的项目没有质量和生产数据可提供。）

**表 3-1　34 个摩托罗拉 GED 项目的结果［Diaz and Sligo，1997］（©1997，IEEE）**

| CMM 级别 | 项　目　数 | 相对开发时间减少 | 开发中每 MEASL 的错误数目 | 相对产量 |
|---|---|---|---|---|
| 1 级 | 3 | 1.0 | — | — |

（续）

| CMM级别 | 项 目 数 | 相对开发时间减少 | 开发中每MEASL的错误数目 | 相对产量 |
|---|---|---|---|---|
| 2级 | 9 | 3.2 | 890 | 1.0 |
| 3级 | 5 | 2.7 | 411 | 0.8 |
| 4级 | 8 | 5.0 | 205 | 2.3 |
| 5级 | 9 | 7.8 | 126 | 2.8 |

由这一节提供的研究数据和本章的“进一步阅读”中列出的资料可以发现，世界范围内越来越多的组织认识到，在过程改进方面投资是非常有效的。

过程改进运动非常有意思的一个影响就是：它使软件过程改进的尝试和软件工程标准之间发生相互作用。例如，1995年，国际标准化组织发布ISO/IEC 12207，一个完整的软件生命周期标准［ISO/IEC 12207，1995］。3年后，美国电气电子工程师学会和电子工业联盟发布了美国版标准［IEEE/EIA 12207 1998］。这个版本包括美国软件开发方法中的“最佳实践”，这些实践中的许多都可以追溯到CMM。要想达到IEEE/EIA 12207的标准，一个组织必须达到CMM3级标准［Ferguson and Sheard，1998］。并且，ISO 9000-3现在包括ISO/IEC 12207的部分内容。这种软件工程标准化组织和软件过程改进尝试之间的相互影响导致了真正的软件过程改进。

软件过程改进的另一种观点见“如果你想知道”部分。

**如果你想知道**

*硬件的速度有许多限制，因为电子无法比光速更快。在题为“没有银弹”（No Silver Bullet）［Brooks，1986］的文章中，Brooks揭示了在软件生产中存在着本质性的问题，这些问题由于软件上的类似的限制而永远无法解决。Brooks认为软件的内在特性，如它的复杂性、无形性和不可见性，以及软件在它的生命周期过程中经受的无数改变，使得在软件过程改进上不可能存在着一个巨大的跨越（或“银弹”）。*

## 本章回顾

在经过一些基本的定义之后，在3.1节介绍了统一过程。在3.2节描述了面向对象范型内的迭代和递增。随后详细解释了统一过程的核心工作流：需求流（3.3节），分析流（3.4节），设计流（3.5节），实现流（3.6节）和测试流（3.7节）。在3.7.1节到3.7.4节中，讲述了测试流期间的各种被测试的制品。在3.8节讨论了交付后维护，并且在3.9节中讨论了退役。在3.10节中分析了统一过程的各个阶段和各个工作流之间的关系，统一过程的四个阶段的详细描述在以下各小节中给出：初始阶段（3.10.1节）、细化阶段（3.10.2节）、构建阶段（3.10.3节）、转换阶段（3.10.4节）。二维生命周期模型的重要性在3.11节中讨论。

本章的最后一部分描述了软件过程改进（3.12节）。对全球软件过程改进方面的尝试进行了详细描述，包括能力成熟度模型（3.13节）、ISO 9000和ISO/IEC 15504（3.14节）。3.15节对软件过程改进的成本效率进行了讨论。

## 进一步阅读

第1章中“进一步阅读”一节中的综述性文章也强调了与软件过程有关的问题[Brooks, 1975; Boehm, 1976; Wasserman, 1996; and Ebert, Matsubara, Pezzé, and Bertelsen, 1997]。《IEEE Software》2003年3/4月刊包含了一些软件过程方面的文章，包括[Eickelmann and Anant, 2003]，其中讨论了统计过程控制。在Weller[2000]和Florac, Carleton, Barnard[2000]中讨论了统计过程控制的实践应用。

对于每个工作流期间的测试，一个较好的常用资料来源是[Beizer, 1990]。本书的第6章，以及该章末尾的“进一步阅读”一节有更多具体的参考资料。

[Humphrey, 1989]对最初的SEI能力成熟度模型进行了详细描述。[SEI, 2002]对能力成熟度模型集成作了阐述。[Humphrey, 1996]描述了个人软件过程（PSP）；PSP的应用结果出现在[Ferguson et al., 1997]中。[Johnson and Disney, 1998]对PSP的潜在问题进行了讨论。[Humphrey, 1999]对PSP和团队软件过程（TSP）进行了描述。度量PSP训练效果的实验结果在[Prechelt and Unger, 2000]中给出。对统一过程进行扩展以使其符合CMM级别2和级别3的一些工作在[Manzoni and Price, 2003]中给出。《IEEE Software》2000年7/8月刊有3篇软件过程成熟度方面的文章，并且在《IEEE Software》2000年11/12月刊中有4篇PSP方面的文章。

[Herbsleb et al., 1997]包含了对SEI软件过程评估的一个概述。也有人从特定公司的角度写了一些关于工业界经验的文章，这些文章介绍了有些公司引入SEI软件过程改进了工作进度；典型的例子包括Shlumberger[Wohlwend and Rosenbaum, 1993]和Raytheon[Haley, 1996]。[Saiedian and Kuzara, 1995]和[Johnson and Brodman, 2000]讨论了SEI对软件工业的影响。[Bamberger, 1997]给出了对CMM的一个有趣的看法。[Bamford and Deibler, 1993b]对ISO 9000和CMM进行了详细的比较；[Bamford and Deibler, 1993a]对上述比较作了一个概述。[Paulk, 1995]对ISO 9000和CMM进行了另一个详细的比较。Pitterman[2000]描述了Telecordia Technologies的一个小组是如何达到成熟度级别5的；[McGarry and Decker, 2002]中给出了一个计算机科学联合攻关小组是如何达到成熟度级别5的一个研究。

软件生产过程改进的问题出现在[Conradi and Fuggetta, 2002]中。大量有关CMM的文章见SEI CMM的网站：www. sei. cmu. edu。ISO/IEC 15504（SPICE）的主页位于：www. sei. cmu. edu/technology/process/spice/。

[Ferguson and Sheard, 1998]对CMM和IEEE/EIA 12207作了一个比较。[Murugappan and Keeni, 2003]对CMM和六西格玛（另一个过程改进解决办法）作了一个比较。[Blanco, Gutiérrez, and Satriani, 2001]中描述了一个信息库，其中包含了大约400个软件改进试验的结果。

## 习题

3.1　考察需求流和分析流。将这两个工作流结合为一个工作流比分别对待它们更有意义吗？

3.2　在实现流中比在别的工作流中需要进行更多的测试，将此工作流分成两个独立的工作流是否会更好？一个包含不需测试的方面，另一个包含其他所有的测试。

3.3 维护是软件生产最重要并且最困难的阶段。尽管如此，许多软件专业人员仍然对它非常轻视，并且维护人员的报酬少于开发人员。你认为这样合理吗？如果不合理，那么你将怎样试着去改变它？

3.4 为什么你认为，如3.9节所述，真正的退役是很罕见的情况？

3.5 如果Elmer的软件因为一场大火，在产品交付给客户前烧掉了它所有的文档。那么文档缺乏导致的影响是什么？

3.6 你刚刚购买了Antedeluvian Software Developers，一个快破产的组织，因为该公司的成熟度级别为1级。让该组织盈利所需采取的第一步是什么？

3.7 3.13节指出，在具有成熟度级别1和级别2的组织内部引入CASE环境没有多大意义。请解释为什么？

3.8 在具有较低成熟度级别的组织内引入CASE工具（与环境相对应）有什么效果？

3.9 （学期项目）如果附录A的Ophelia’s Oasis产品是由一个在CMM级别1的组织开发的，与一个级别5的组织相比，你预计会发现什么不同？

3.10 （软件工程读物）你的导师向你发放［Manzoni and Price，2003］的一些复印件。为了与CMM级别2和级别3相符合，请为统一过程需要加入的特性分类。

## 参考文献


[Bamberger, 1997] J. BAMBERGER, "Essence of the Capability Maturity Model," *IEEE Computer* **30** (June 1997), pp. 112–14.

[Bamford and Deibler, 1993a] R. C. BAMFORD AND W. J. DEIBLER, II, "Comparing, Contrasting ISO 9001 and the SEI Capability Maturity Model," *IEEE Computer* **26** (October 1993), pp. 68–70.

[Bamford and Deibler, 1993b] R. C. BAMFORD AND W. J. DEIBLER, II, "A Detailed Comparison of the SEI Software Maturity Levels and Technology Stages to the Requirements for ISO 9001 Registration," Software Systems Quality Consulting, San Jose, CA, 1993.

[Beizer, 1990] B. BEIZER, *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, New York, 1990.

[Blanco, Gutiérrez, and Satriani, 2001] M. BLANCO, P. GUTIÉRREZ, AND G. SATRIANI, "SPI Patterns: Learning from Experience," *IEEE Software* **18** (May/June 2001), pp. 28–35.

[Boehm, 1976] B. W. BOEHM, "Software Engineering," *IEEE Transactions on Computers* **C-25** (December 1976), pp. 1226–41.

[Booch, 1994] G. BOOCH, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.

[Booch, Rumbaugh, and Jacobson, 1999] G. BOOCH, J. RUMBAUGH, AND I. JACOBSON, *The UML Users Guide*, Addison Wesley, Reading, MA, 1999.

[Brooks, 1975] F. P. BROOKS, JR., *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, Reading, MA, 1975; Twentieth Anniversary Edition, Addison Wesley, Reading, MA, 1995.

[Brooks, 1986] F. P. BROOKS, JR., "No Silver Bullet," in: *Information Processing '86*, H.-J. Kugler (Editor), Elsevier North-Holland, New York, 1986; reprinted in *IEEE Computer* **20** (April 1987), pp. 10–19.

[Brooks et al., 1987] F. P. BROOKS, V. BASILI, B. BOEHM, E. BOND, N. EASTMAN, D. L. EVANS, A. K. JONES, M. SHAW, AND C. A. ZRAKET, "Report of the Defense Science Board Task Force on Military Software," Department of Defense, Office of the Under Secretary of Defense for Acquisition, Washington, DC, September 1987.

[CNN.com, 2003] "Russia: Software Bug Made Soyuz Stray," edition.cnn.com/2003/TECH/space/05/06/soyuz.landing.ap/.

[Conradi and Fuggetta, 2002] R. CONRADI AND A. FUGGETTA, "Improving Software Process Improvement," *IEEE Software* **19** (July/August 2002), pp. 92–99.

[Dawood, 1994] M. DAWOOD, "It's Time for ISO 9000," *CrossTalk* (March 1994) pp. 26–28.

[Deming, 1986] W. E. DEMING, *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge, MA, 1986.

[Diaz and Sligo, 1997] M. DIAZ AND J. SLIGO, "How Software Process Improvement Helped Motorola," *IEEE Software* **14** (September/October 1997), pp. 75–81.

[Dion, 1993] R. DION, "Process Improvement and the Corporate Balance Sheet," *IEEE Software* **10** (July 1993), pp. 28–35.

[Ebert, Matsubara, Pezzè, and Bertelsen, 1997] C. EBERT, T. MATSUBARA, M. PEZZÈ, AND O. W. BERTELSEN, "The Road to Maturity: Navigating between Craft and Science," *IEEE Software* **14** (November/December 1997), pp. 77–88.

[Eickelmann and Anant, 2003] N. EICKELMANN AND A. ANANT, "Statistical Process Control: What You Don't Know Can Hurt You!" *IEEE Software* **20** (March/April 2003), pp. 49–51.

[Ferguson and Sheard, 1998] J. FERGUSON AND S. SHEARD, "Leveraging Your CMM Efforts for IEEE/EIA 12207," *IEEE Software* **15** (September/October 1998), pp. 23–28.

[Ferguson et al., 1997] P. FERGUSON, W. S. HUMPHREY, S. KHAJENOORI, S. MACKE, AND A. MATVYA, "Results of Applying the Personal Software Process," *IEEE Computer* **30** (May 1997), pp. 24–31.

[Florac, Carleton, and Barnard, 2000] W. A. FLORAC, A. D. CARLETON, AND J. BARNARD, "Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process," *IEEE Software* **17** (July/August 2000), pp. 97–106.

[*Florida Today*, 1999] "Milstar Satellite Lost during Air Force Titan 4b Launch from Cape," www.floridatoday.com/space/explore/uselv/titan/b32/, June 5, 1999.

[Garman, 1981] J. R. GARMAN, "The 'Bug' Heard 'Round the World," *ACM SIGSOFT Software Engineering Notes* **6** (October 1981), pp. 3–10.

[Haley, 1996] T. J. HALEY, "Raytheon's Experience in Software Process Improvement," *IEEE Software* **13** (November 1996), pp. 33–41.

[Herbsleb et al., 1997] J. HERBSLEB, D. ZUBROW, D. GOLDENSON, W. HAYES, AND M. PAULK, "Software Quality and the Capability Maturity Model," *Communications of the ACM* **40** (June 1997), pp. 30–40.

[Humphrey, 1989] W. S. HUMPHREY, *Managing the Software Process*, Addison Wesley, Reading, MA, 1989.

[Humphrey, 1996] W. S. HUMPHREY, "Using a Defined and Measured Personal Software Process," *IEEE Software* **13** (May 1996), pp. 77–88.

[Humphrey, 1999] W. S. HUMPHREY, "Pathways to Process Maturity: The Personal Software Process and Team Software Process," *SEI Interactive* **2** (No. 4, December 1999), interactive.sei.cmu.edu/Features/1999/June/Background/Background.jun99.htm.

[Humphrey, Snider, and Willis, 1991] W. S. HUMPHREY, T. R. SNIDER, AND R. R. WILLIS, "Software Process Improvement at Hughes Aircraft," *IEEE Software* **8** (July 1991), pp. 11–23.

[IEEE/EIA 12207.0-1996, 1998] "IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995," Institute of Electrical and Electronic Engineers, Electronic Industries Alliance, New York, 1998.

[ISO 9000-3, 1991] "ISO 9000-3, Guidelines for the Application of ISO 9001 to the Development, Supply, and Maintenance of Software," International Organization for Standardization, Geneva, Switzerland, 1991.

[ISO 9001, 1987] "ISO 9001, Quality Systems—Model for Quality Assurance in Design/Development, Production, Installation, and Servicing," International Organization for Standardization, Geneva, Switzerland, 1987.

[ISO/IEC 12207, 1995] "ISO/IEC 12207:1995, Information Technology—Software Life-Cycle

Processes," International Organization for Standardization, International Electrotechnical Commission, Geneva, Switzerland, 1995.

[Jacobson, Booch, and Rumbaugh, 1999] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Software Development Process*, Addison Wesley, Reading, MA, 1999.

[Johnson and Brodman, 2000] D. JOHNSON AND J. G. BRODMAN, "Applying CMM Project Planning Practices to Diverse Environments," *IEEE Software* **17** (July/August 2000), pp. 40–47.

[Johnson and Disney, 1998] P. M. JOHNSON AND A. M. DISNEY, "The Personal Software Process: A Cautionary Tale," *IEEE Software* **15** (November/December 1998), pp. 85–88.

[Jones, 1996] C. JONES, *Applied Software Measurement*, McGraw-Hill, New York, 1996.

[Juran, 1988] J. M. JURAN, *Juran on Planning for Quality*, Macmillan, New York, 1988.

[Keeni, 2000] G. KEENI, "The Evolution of Quality Processes at Tata Consultancy Services," *IEEE Software* **17** (July/August 2000), pp. 79–88.

[Manzoni and Price, 2003] L. V. MANZONI AND R. T. PRICE, "Identifying Extensions Required by RUP (Rational Unified Process) to Comply with CMM (Capability Maturity Model) Levels 2 and 3," *IEEE Transactions on Software Engineering* **29** (February 2003), pp. 181–92.

[McGarry and Decker, 2002] F. MCGARRY AND B. DECKER, "Attaining Level 5 in CMM Process Maturity," *IEEE Software* **19** (2002), pp. 87–96.

[Miller, 1956] G. A. MILLER, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review* **63** (March 1956), pp. 81–97; reprinted at www.well.com/user/smalin/miller.html.

[Murugappan and Keeni, 2003] M. MURUGAPPAN AND G. KEENI, "Blending CMM and Six Sigma to Meet Business Goals," *IEEE Software* **20** (March/April 2003), pp. 42–48.

[Paulk, 1995] M. C. PAULK, "How ISO 9001 Compares with the CMM," *IEEE Software* **12** (January 1995), pp. 74–83.

[Paulk, Weber, Curtis, and Chrissis, 1995] M. C. PAULK, C. V. WEBER, B. CURTIS, AND M. B. CHRISSIS, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison Wesley, Reading, MA, 1995.

[Pitterman, 2000] B. PITTERMAN, "Telecordia Technologies: The Journey to High Maturity," *IEEE Software* **17** (July/August 2000), pp. 89–96.

[Prechelt and Unger, 2000] L. PRECHELT AND B. UNGER, "An Experiment Measuring the Effects of Personal Software Process (PSP) Training," *IEEE Transactions on Software Engineering* **27** (May 2000), pp. 465–72.

[Rumbaugh et al., 1991] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Saiedian and Kuzara, 1995] H. SAIEDIAN AND R. KUZARA, "SEI Capability Maturity Model's Impact on Contractors," *IEEE Computer* **28** (January 1995), pp. 16–26.

[SEI, 2002] "CMMI Frequently Asked Questions (FAQ)," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, June 2002.

[van Wijngaarden et al., 1975] A. VAN WIJNGAARDEN, B. J. MAILLOUX, J. E. L. PECK, C. H. A. KOSTER, M. SINTZOFF, C. H. LINDSEY, L. G. L. T. MEERTENS, AND R. G. FISKER, "Revised Report on the Algorithmic Language ALGOL 68," *Acta Informatica* **5** (1975), pp. 1–236.

[Wasserman, 1996] A. I. WASSERMAN, "Toward a Discipline of Software Engineering," *IEEE Software* **13** (November/December 1996), pp. 23–31.

[Weller, 2000] E. F. WELLER, "Practical Applications of Statistical Process Control," *IEEE Software* **18** (May/June 2000), pp. 48–55.

[Wohlwend and Rosenbaum, 1993] H. WOHLWEND AND S. ROSENBAUM, "Software Improvements in an International Company," *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, May 1993, pp. 212–20.

# 第4章 软件小组

**学习目标**

学完本章之后，你应当能够：

- 解释一个安排良好的软件小组的重要性；
- 描述现代分级小组是如何安排的；
- 分析各种不同小组组织的优缺点；
- 理解当选择一个合适的小组组织时会产生的问题。

若没有称职的、经过良好训练的软件工程师，一个软件项目注定要失败。然而有合适的人选并不够，软件小组必须有效地组织起来，使小组成员能够与其他人合作，提高生产力。小组组织是本章的主题。

## 4.1 小组组织

大多数的软件产品都很大，一个软件专业人员不可能在有限时间内单独完成。因而，产品必须分配给一组专业人员，形成一个**小组**。例如，在规格说明阶段，要在2个月之内确定目标产品的规格说明，可能需要把此项工作分配给三个专业人员，成立一个由规格说明经理领导的小组。同样地，设计任务可能需要设计小组的成员来共同完成。

假设一个涉及到1人年代码工作量（1人年指的是一个人干1年的工作量）的产品需要在3个月内编出代码，解决方案看似很简单：如果一个程序员可在1年内完成产品，则4个程序员在3个月内即可完成它。

这当然行不通，实践中，4个程序员大概要花将近1年的时间，而且编出的产品质量很可能会低于1个程序员对整个产品进行编程的质量。原因是有一些任务可以分担，但另一些必须单独完成。例如，如果一个农夫可在10天里采摘完一块草莓地，同样大小的草莓地可以由10个农夫在1天里采摘完。另一方面，一个大象可在22个月里生育一个小象崽，但决不可能由22个大象在1个月里生育出一个小象崽。

换句话说，像采摘草莓这样的任务完全可以分担，但像大象生育这样的任务是不能分担的。与生育小象崽不同，在小组成员之间通过分开编写代码，可以分担软件实现的任务。然而小组编程又不像采摘草莓，小组成员之间需要以一种富有意义并且卓有成效的方式进行交互。例如，假设赛拉和哈里要对两个模块m1和m2编写代码，许多事情都可能会弄错。比如，赛拉和哈里可能都对m1模块编写代码，而忽略了m2模块；或者赛拉编m1模块的代码，哈里编m2模块，但当m1调用m2时，传递了4个参数，而哈里给m2编码时却要求5个参数；或者m1和m2中参数的顺序可能不同；也有可能参数顺序相同了，而数据类型有所不同。这样的问题经常是由设计阶段做出的某一个决定没有在开发组中一一传达到而引起的。这种事情丝毫与程序员的技术能力无关，开发小组的组织是一个管理性的问题，管理人员必须对开发小组进行组织管理，保证每个小组工作的高效率。

图4-1显示了软件的小组开发会产生的另一种不同的困难。例如，开发某项目的3个计算机专业人员之间有3个沟通渠道，现在假设工作有点拖进度了，完成任务的最后期限很快

就要到了，而任务还未完成。最明显的办法是向小组中增加第4个专业人员，但第4个专业人员加入小组后，需解决的头一件事情是其他3个人需详细解释迄今为止完成了哪些工作，还有什么没有完成。换句话说，向一个已经延期的软件项目增加人员会使该项目完成得更晚。这个规律被称为**布鲁克斯法则**，这是Fred Brooks进行OS/360的开发时发现的［Brooks，1975］。

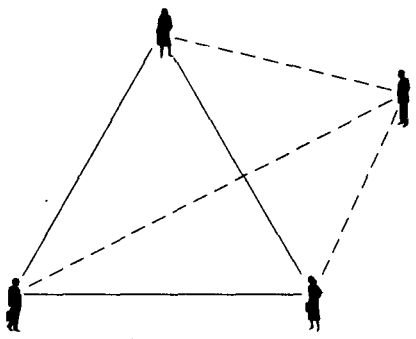


图4-1 3个计算机专业人员之间的沟通渠道（实线）以及加入第4个人后的沟通渠道（虚线）

在一个大型组织里，在软件生产的每个工作流都使用小组，尤其是在执行实现流的过程中。在该工作流中，程序员独立地实现不同的代码制品，因而主要在实现流中在几个计算机专业人员之间分担任务。在一些小的组织里，一个人就可以负责需求、规格说明和设计，之后的实现可以由2～3个程序员的小组完成。由于在实现流中最常使用小组，因而在实现期间感觉到的小组组织管理的问题就越尖锐。所以在本章的余下部分里，我们在软件实现的范畴内提出小组的组织管理问题，尽管这些问题和相应的解决方案同样适用于所有其他工作流。

编程小组的组织有两种极端方法：民主小组和主程序员小组。这里将描述每种方法，突出它的优缺点，然后提出结合了这两种极端方法的优点的其他组织编程小组的方法。

## 4.2 民主小组方法

1971年Weinberg首次描述了民主小组的组织［Weinberg，1971］。民主小组的最基本概念是**无我编程**（egoless programming）。Weinberg指出，程序员特别喜爱他们编写的代码，有时甚至以他们的名字命名模块，所以他们视这些模块为他们自己的一种扩展。这样做的一个问题是，程序员如果把自己开发的模块看成“他”或“她”自我的一种扩展，则不愿意找出“他”或“她”的代码中的错误。如果有错误，则称它为“bug”，就像某些爬行在代码中的不受欢迎的虫子，如果编写代码时对入侵更加重视的话，就可避免出现这样的“bug”（见下面的“如果你想知道”部分）。

**如果你想知道**

*大约40年前，那时软件还是在穿孔卡上输入，那时绝大多数程序员都对软件中的bug抱着同样的看法，将它们看成是若不加阻止就会入侵他们的读卡机的昆虫。这个态度又被投入市场的一种名为Shoo-bug的喷雾剂有趣地讽刺了一番。该产品标签上的使用说明郑重其事地说明，用Shoo-bug喷洒读卡机可以确保代码中不会滋生bug。当然，该喷雾剂除了空气之外不会包含其他任何物质。*

Weinberg解决程序员太看重自己代码这个问题的方法是无我编程。社会化的环境必须重新建立起来，程序员的价值也同样必须重新建立。每个程序员必须鼓励小组的其他成员在他或她的代码中找寻错误。一个错误的存在并不是什么坏事，应该看成是一个正常并可接受的事实。评审者应当对受邀提意见心存感激，而不应当嘲笑程序员犯了错误。小组作为一个整体形成了一种群体精神、一种群体特征，而且软件模块属于小组这个整体而不是任何个人。

一个多达 10 个无我编程员的小组组成了一个**民主小组**（democratic team）。Weinberg 警告管理者与这样的小组工作会有困难，毕竟要考虑到管理的职业方法问题。当一个程序员被提升到管理者的位置，他或她的程序员同事没有被提升，他们一定会努力争取在下一轮提升时获得更高的职位。相反，一个民主小组是为一个共同的事业而工作的一个团体，没有单独的领导，没有程序员试图提升到更高的职位，最重要的是小组的群体特征和相互尊重。

Weinberg 谈到一个开发出优质产品的民主小组，经理决定给这个小组名义上的领导（根据定义，一个民主小组没有领导）一笔现金奖励，他拒绝个人接受这个奖励，说应该平均分给小组内的所有成员。经理以为他想要更多的钱，并且认为该小组（特别是它名义上的这个领导）有一些怪异的想法。经理强迫这个名义上的领导接受这笔钱，之后他便将这笔钱均分给组员，然后全体组员集体辞职并仍作为一个小组加入到另一个公司。

下面讨论民主小组的优缺点。

**民主小组方法的分析**

民主小组方法的一个主要优点是对查找错误的一种积极的态度。找到的错误越多，民主小组的成员越高兴。这种积极的态度会导致错误更快地被发现，因而代码质量越高。但也有一些问题，就像前面指出的，经理们可能难以接受无我编程。另外，一个具有 15 年经验的程序员可能不愿意把他或她的代码交给别的程序员，尤其是交给初来乍到者评判。

Weinberg 认为无我小组是自发形成的，不能从外界强加。在民主小组上很少进行实验性的研究，但 Weinberg 的经验是民主小组的工作效率很高。Mantei 使用基于一般意义上的群体组织理论和实践的论据，分析了民主小组组织，而不是专门分析特定的编程小组［Mantei，1981］。她指出非集中式的小组在问题很困难时工作得很好，并提出民主小组在一个研究的环境下会更好地发挥作用。我曾经的经验是：在一个工业化的环境下如果有很难的问题需要解决，民主小组也可以工作得很好。在许多场合下，我曾是民主小组的成员，这些民主小组是在有研究经验的计算机专家之间自然形成的。但一旦任务简化为一个难得的解决方案的实现时，该小组一定得以更等级化的方式重新组合，例如，下节将要叙述的主程序员小组方法。

## 4.3　传统的主程序员小组方法

考虑图 4-2 所示的 6 人小组，2 人之间的沟通渠道有 15 条，事实上，2 人之间、3 人之间、4 人之间、5 人之间和 6 人之间的沟通渠道的总数为 57 条。这种沟通渠道的多样性是如图 4-2 这样的 6 人小组不可能在 6 个月内完成 36 人月的工作量的原因，许多时间都浪费在与两个或更多的小组成员间的交互上。

现在考虑图 4-3 所示的 6 人小组。这里还是有 6 个程序员，但只有 5 条沟通渠道，这就是现在称为**主程序员**（chief programmer）**小组**的基本概念。与此相关的想法是由 Brooks 提出的，他用主持手术的主外科医生进行了类比分析［Brooks，1975］。这个外科医生由其他外科医生、麻醉医生和各种护士辅助，另外在必要时这个小组还使用其他领域的专家，例如心脏学专家和肾脏学专家。这个类比突出了主程序员小组的两个关键特性，第一个是**专业化**（specialization）：小组的每个成员只承担他为之培训过的那部分工作；第二个特性是**等级性**（hierarchy）：主外科医生指导小组所有其他成员的行动，并且对该手术的每个方面负责。

主程序员小组的概念是由 Mills 形成的[Baker, 1972]。由 Baker 在 30 年前所描述的一个传统的主程序员小组如图 4-3 所示，它由主程序员、做其助手的备程序员、编程秘书和 1 到 3 个程序员组成。必要时，其他领域例如法律事务、金融事务或工作控制语言（job control language, JCL）语句（用来给那个时期的主机下操作系统命令）方面的专家也给予支持。**主程序员**既是一个成功的管理者，也是一个训练有素的程序员，他完成结构化设计以及代码中的任何关键和复杂的部分。其他的小组成员在主程序员的指导下进行具体的细节设计和编写代码。如图 4-3 所示，在程序员之间没有沟通的渠道，所有的接口问题都由主程序员解决。最后，主程序员审查其他小组成员的工作，因为主程序员个人要对每行代码负责。

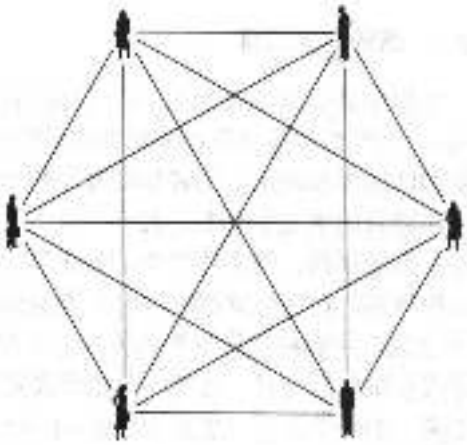


图 4-2 6 个计算机专业人员之间的沟通渠道

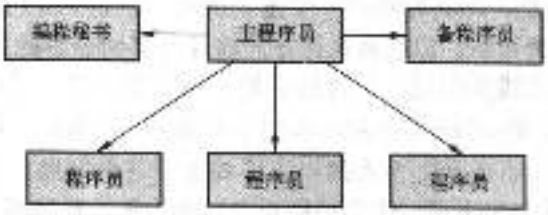



图 4-3 传统的主程序员小组的结构

**备程序员**（backup programmer）的位置之所以必要，是因为主程序员是人，他有可能会生病、迟到或调换工作。所以备程序员应该在各个方面与主程序员一样有能力，并且需要与主程序员一样深入了解这个项目。另外，为让主程序员集中精力进行结构化设计，备程序员应进行黑盒测试的用例规划（见 14.11 节），并承担其他与设计过程独立的任务。

秘书（secretary）一词有许多含意。一方面，秘书帮助繁忙的主管接听电话、打印信件等。但当我们谈论到美国国务卿或英国外交秘书时，我们指的是内阁中最主要的成员之一。**编程秘书**（programming secretary）不是一个兼职的办公助手，却是主程序员小组中一个精通专业、收入颇丰的核心人物。编程秘书负责维护项目产品库，即项目的文档、这包括源程序清单、JCL 和测试数据。程序员将他们的源程序交给编程秘书，由编程秘书负责将它们转换为机器可识别的形式，编译、链接、装载、执行并运行测试用例。**程序员**（programmer）因此除了编程什么也不做，其他的工作都交由编程秘书来做。（因为编程秘书维护项目的产品库，一些组织曾使用**资料管理员**一词。）

我们不要忘记前面描述的是 Mills 和 Baker 最初的想法，可追溯到 1971 年，那时键控穿孔机仍广泛使用。现在再也不那样编写代码了。程序员现在拥有自己的终端或工作站，在那上面他们输入自己的代码，编辑、测试它们，如此等等。4.4 节将叙述传统的主程序员小组的现代版。

### 4.3.1　纽约时报项目

主程序员小组的概念在1971年由IBM公司首次用于自动处理《纽约时报》（New York Times）的剪辑文件（报社编辑部收集的参考文件）。剪辑文件包含来自《纽约时报》和其他出版物的摘要和全文。记者和编辑部的其他成员使用这个信息库作为参考源。

该项目的事实情况令人大吃一惊。例如，83 000行代码写了22个月，耗费了11人年的努力。第1年后，仅文件维护系统就包含了写出的12 000行代码，大部分的代码在最后的6个月里写完。在验收测试的最初5周里只检测出21个错误，在第1年的运行中又仅检测出25个错误。主要的程序员平均每人每年编写一个错误和10 000行代码。文件维护系统在编码完成一星期后交付，在第一个错误检测出来之前该系统已运行了20个月。在首次编译时，差不多一半的子程序（通常是200～400行PL/I程序）是正确的［Baker，1972］。

然而，在这个巨大的成功之后，没有人就此声称制造了主程序员小组的概念。是的，许多成功的项目是使用主程序员小组完成的，但报道的数字尽管令人满意，但都不及《纽约时报》的项目那样令人印象深刻。为什么《纽约时报》项目如此成功，为什么在其他项目上没有得到类似的结果？

一个可能的解释是这是IBM公司的一个重要项目，它是PL/I的首次实践，PL/I是由IBM公司开发出来的一种语言。以拥有极优秀的软件专家而著称的IBM公司，建立了一个小组，若要形容该小组的成员组成，只能称它们是一个部门的尖子中的尖子。第二个解释是，技术支持极为强大。PL/I编译器的作者们随叫随到，以他们能够做到的各种方式，帮助小组中的程序员，而且还有JCL专家提供工作控制语言的帮助。第三个可能的解释是主程序员F.Terry Baker的专家水平。他是那种现在被称为**天才程序员**（superprogrammer）的人，他写出的程序量是一个一般程序员能写出的4或5倍。另外，Baker是一个极好的管理者和领导者，可以说，他的技能、热情和个性是这个项目成功背后的原因。

如果主程序员是胜任的，则主程序员小组组织就会工作得很好。尽管《纽约时报》项目的巨大成功没有再现，许多成功的项目还是使用了主程序员方法的各种变种。使用变种一词的原因是［Baker，1972］中描述的传统的主程序员小组在许多方面都不实用。

### 4.3.2　传统的主程序员小组方法的不实用性

让我们来看看主程序员，他是一个高水平程序员和成功的管理者的结合。这样的人很难找到，不仅高水平的程序员缺乏，成功的管理者也同样缺乏，而主程序员的工作要求是两者兼备。人们认为，一个高水平的程序员所需具备的品质与成功管理者所需具备的品质有所不同，所以按此要求找到一个主程序员的机率不大。

但如果主程序员很难找到，备程序员也同样很难找到。毕竟备程序员要与主程序员一样优秀，而且要处于替补位置，待遇比主程序员低，等待主程序员出什么问题时才能替代他。很少有顶尖的程序员或顶尖的管理者愿意担当这样的角色。

编程秘书也很难寻找。软件专业人员以讨厌进行文字工作而著称，但编程秘书却要整天只做文字工作。

这样，主程序员小组，至少像Baker所提出的主程序员小组，很难付诸实现。民主小组也不实用，只是原因不同。进一步地说，好像这两种技术都不能够处理在实现阶段需要20

个程序员（更不要说120个）的产品。因此，需要一种能够组织起编程小组的方法，它采用民主小组和主程序员小组的长处，并且能够扩展到更大型产品的实现中。

## 4.4 主程序员小组和民主小组之外的编程小组

民主小组有一个主要的长处：对查找错误的一个积极的态度。许多组织在使用主程序员小组的同时，结合使用代码评审（见6.2节），这造成一个潜在的缺陷。主程序员对每行代码是个人负责的，所以他必须出席全部的代码评审。然而，一个主程序员也是管理者，如第6章所讲，评审不应用作任何种类的能力评价。所以，因为主程序员也是管理者，负责对小组成员作出首要评价，则让他个人出席代码评审是非常失策的。

避免这种矛盾的办法是去掉主程序员的大部分管理职能，前面已经提到找到一个既精通业务，又擅长管理的人是多么困难。因而，主程序员应当由两个人替代：一个是**小组领导**（team leader），他负责小组活动中的技术方面事务；另一个是**小组经理**（team manager），他负责所有非技术性的管理事务，得出的小组的结构如图4-4所示。重要的是要意识到，这样的组织结构没有违反"雇员不应当向多个管理者汇报"这个基本的管理原则。职责的范围已经明确地描绘了：小组领导只负责技术性管理，这样，财务和法律上的事务不由小组领导负责，他也不负责对小组成员的表现进行评价。另一方面，小组领导是技术事务惟一的负责人，小组经理因而没有权利去许诺比如说该产品将在四周内交付，这样的许诺只能由小组领导来做出。小组领导自然要参与所有的代码评审，毕竟他或她对代码的每个方面都是个人负责的。同时，小组经理不允许参加评审，因为对程序员的表现进行鉴定是小组经理的职责，小组经理在日常安排的小组会议中了解小组中每个程序员的技术水平情况。

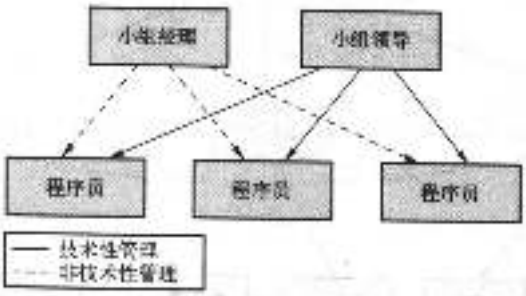



图4-4 现代编程小组的结构

在实现过程开始前，划清小组经理和小组领导好像都负责的那些区域非常重要。例如，年度休假的事情。会出现这种情况，小组经理批准了一个休假请求，因为休假是一个非技术性问题，然而却被小组领导因为产品交付的最后期限将至而否决了。这样的事情及类似的事情的解决，需要更高一层的管理者制定一项政策，该项政策事关小组领导和小组经理都认为对某一范围内的事务负责时的解决办法。

那么大型项目怎么办？这种方法可以如图4-5所示按比例扩展，图中显示了一个技术性管理的组织结构，非技术性方面也同样地组织。产品的整体实现是在项目领导的指导下进行的，程序员向他们的小组领导报告，小组领导向项目领导报告。对于更大型的软件产品，可以向这个分层结构中增加额外的层次。

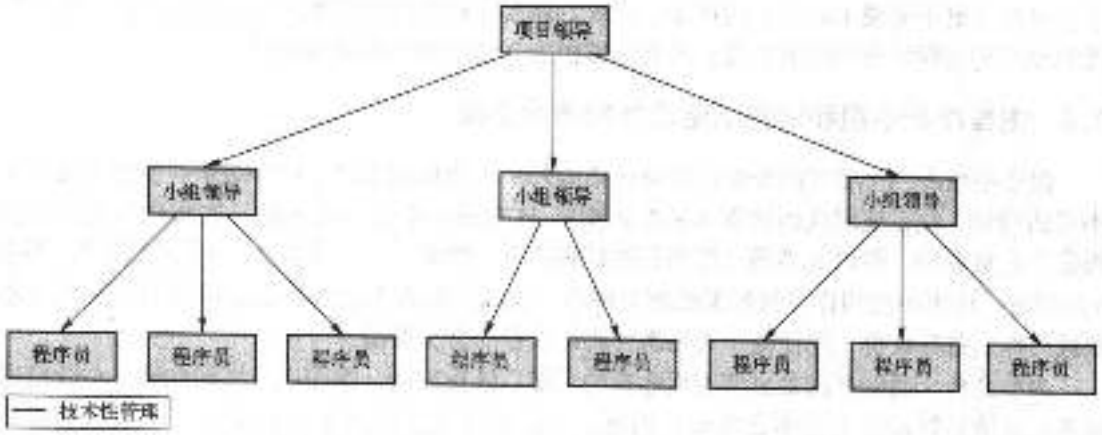



图 4.5　大型项目的技术性管理组织结构

从民主小组和主程序员小组的优点中得出的另一种方法是在合适的地方分散决策过程，这样得出的沟通渠道如图 4-6 所示。这种安排对适用于民主小组方法的各种问题非常有用，也就是说，在一个研究环境里，或者在一个难题需要小组交互来相互促进地解决的情况下。尽管使用分散的方法，各层之间的箭头仍旧向下指，允许程序员命令项目领导只能导致混乱。

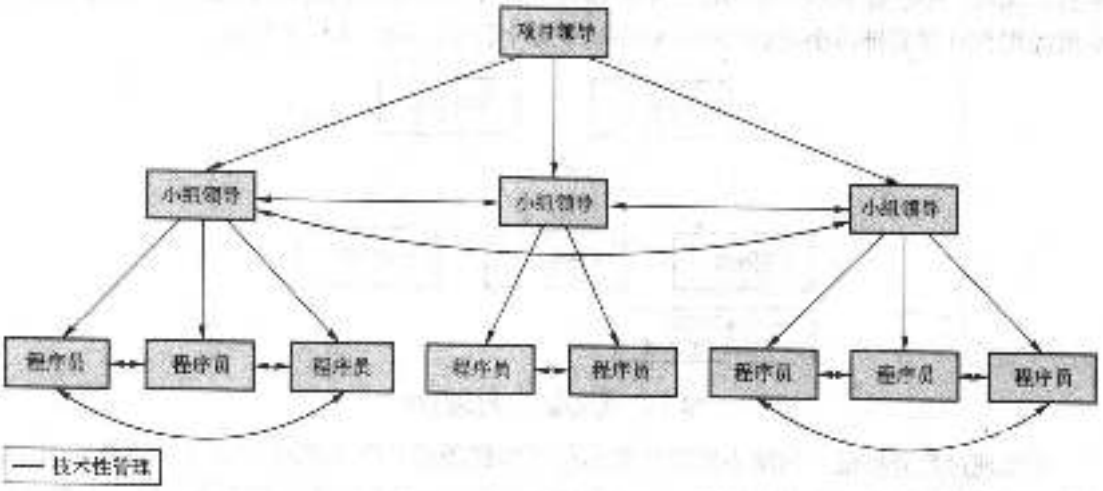



图 4-6　图 4-5 所示的小组结构的分散决策形式下的技术管理沟通渠道

## 4.5　同步－稳定小组

另一种供选择的小组组织方法是 Microsoft 公司使用的同步－稳定小组［Cusumano and Selby，1997］。Microsoft 公司创建了一些大型的软件产品，例如，Windows 2000 包含 3000 多万行代码，由 3000 多个程序员和测试者共同建造［Business Week Online，1999］。小组组织是成功建造这样大规模的软件产品的一个关键因素。

2.9.5节中描述了同步－稳定生命周期模型，这种模型的成功很大程度上是小组的组织方法的结果。同步－稳定模型的每3个或4个连续构件都是由一些小的并行小组建造的，这些小组由一个程序管理者领导，它由3～8个开发者和同开发者一对一工作的3～8个测试者组成。提供给该小组整个任务的规格说明，每个小组成员根据自己的意愿自由设计并实现任务中自己的那部分。这不会引起混乱的原因是每天进行的同步步骤：每天都对部分完成的组件测试并调试，这样即便培育了个人的创造性和自主性，个人的组件也总是要协同工作的。

这种方法的长处是，一方面鼓励单个的程序员进行创新，这是一个民主小组的标志；另一方面，每天的同步步骤确保上百个程序员能够朝着一个共同的目标一同工作，而不需要主程序员小组的沟通和协作特性（参见图4-3）。

Microsoft的开发者必须遵循的规则并不多，但其中一个必须严格遵守的是，他们每天都要将他们的代码输入到产品数据库中，以便于同步该天的工作。Cusumano和Selby把这比作告诉孩子他们可以整天做他们想做的，但到晚上9点时必须上床睡觉［Cusumano and Selby，1997］。另一个规则是，如果一个开发者的代码阻碍了该产品那天的同步编译，则问题必须立刻解决，这样小组的其他人才能测试并调试那天的工作。

使用同步－稳定生命周期模型和相关的小组组织一定能确保其他软件组织像Microsoft一样成功吗？这极不可能。Microsoft公司不只是同步－稳定模型的体现，它是一个由一群才华横溢的管理者和软件开发者组成的组织，他们具有高度发展的群体精神。仅仅使用同步－稳定模型并不能奇迹般地将一个公司变成另一个Microsoft。但是，在其他组织中使用该模型的多种特性可以促使开发过程的改进。另一方面，同步－稳定模型只是一种允许一组计算机天才建造一个大型产品的方式，并且Microsoft公司的成功是由于极好的市场开发，而不仅仅是高质量的软件。

## 4.6 极限编程小组

在2.9.4节中，我们对极限编程进行了概述［Beck，2000］，它是一个灵活的过程［Beck，et al.，2001］。在本节中，我们介绍使用极限编程（extreme programming，XP）时如何组织小组。

XP的一个有点超乎寻常的特点是小组由一对一对的程序员组成，每对程序员共享一台计算机编写代码，这指的是**结对编程**（pair programming）［Williams，Kessler，Cunningham，and Jeffries，2000］。这种方法的一些特点如下：

1）如2.9.4节所述，结对的程序员们首先写出测试用例，然后实现这块代码（**任务**）。如6.6节所述，让程序员自己来测试他或她的代码是很不妥的。XP采取的办法是，让小组中的一个程序员为一个任务编写测试用例，而让另一个程序员使用那些测试用例共同实现该代码。

2）在更常规的生命周期模型中，当开发者离开了一个项目，所有那个开发者积累的知识也离开了。特别是若开发者开发的软件还未形成文档，可能要从头开始重新开发。相反，如果一个结对编程小组中的一个成员离开了，另一个完全可以继续和一个新伙伴完成相同部分的软件开发。进一步说，如果新的小组成员偶然误修改造成软件毁坏，则测试用例的存在有助于揭示错误。

3）两人紧密地一同工作可以使经验不太丰富的软件专业人员从经验丰富的伙伴那里学

到更多的技艺。

4）2.9.4 节中曾提到，多个 XP 结对组员使用的所有计算机一起放置在一个大房间中，这增进了代码的小组所有权，是无我小组的一个积极的特性（见 4.2 节）。

这样，尽管两个程序员在同一台计算机上工作的想法有点不寻常，实践中，它还是有独特的优点的。

## 4.7　人员能力成熟度模型

人员能力成熟度模型（P－CMM）描述了管理和开发一个组织的人力资源的最佳实践[Curtis, Helfey, and Miller, 2002]。至于软件能力成熟度模型 SW－CMM（3.13 节），它是组织安排上的一个进步，它有 5 个成熟度级别，目标是不断改进个人的技能并形成高效的团队组织。

每个成熟度级别都有它自己的**关键过程区**（KPA），在确定一个组织获得了该成熟度级别前，需要完满地完成该区的任务。例如，对于级别 2（管理级），KPA 是指：安置员工、沟通和协调、工作环境、性能管理、训练和开发以及补偿。与此相对应，最优级即第 5 级的 KPA 是指：连续能力提高、组织的能力联合，以及连续人力资源改革。

SW－CMM 是提高一个组织的软件过程的框架，没有建议特定的过程或方法。同样，P－CMM是提高一个组织的管理和开发它的人力资源的框架，没有提出具体的小组组织的方法。

## 4.8　选择合适的小组组织

表 4-1 中给出了各种类型的小组组织的比较，它也给出了介绍每个小组组织的具体章节。遗憾的是，没有一个解决方案可以解决编程小组组织的所有问题，或者通过扩展，可以解决所有其他工作流的组织小组的问题。组织一个小组的最佳办法是依据要建造的产品本身、先前对于各种小组结构的经验、以及该软件组织的文化（这一点很重要）。例如，如果高层管理人员对于分散式地做决定感到不快，那么它就不会实现。

实践中，目前多数小组都如 4.4 节中所描述的那样来组织，即采用主程序员小组形式的某种变种。

软件开发小组组织方面所做的研究工作还不多，并且许多一般为人所接受的原则是基于通常的分组动力学，而不是基于软件开发小组。如果不在软件工业内获得有关小组组织的实验数据，就不容易为某个具体产品确定最优的小组组织。

**表 4-1　小组组织方法的比较以及各种方法所处的小节**

| 小组组织 | 优　点 | 缺　点 |
| --- | --- | --- |
| 民主小组（4.2 节） | 由于积极地寻找错误，因而代码质量高，特别适用于解决难的问题 | 不能从外部强加 |
| 传统的主程序员小组（4.3 节） | 《纽约时报》项目的主要成功之处 | 不实用 |
| 修改的主程序员小组（4.3.1 节） | 有许多成功的范例 | 没有与《纽约时报》项目可比拟的成功范例 |

（续）

| 小组组织 | 优 点 | 缺 点 |
| --- | --- | --- |
| 现代分级编程小组（4.4节） | 小组经理/小组领导结构避免了对主程序员需求的缺乏，可扩展，必要时支持分权管理 | 除非明确小组经理和小组领导间的负责范围，否则容易产生问题 |
| 同步－稳定小组（4.5节） | 鼓励创造性，确保大量开发者朝同一目标工作 | 在Microsoft公司之外还未有该方法应用的实例 |
| 极限编程小组（4.6节） | 程序员不测试自己的代码，如果一个程序员离开不会有损失，经验欠缺的程序员可以向其他人学习，代码具有小组所有权 | 还没有更多的实例证实它的功效 |

## 本章回顾

首先讨论的小组组织（4.1节）是民主小组（4.2节）和传统的主程序员小组（4.3节）。《纽约时报》项目的成功（4.3.1节）与不实用的传统主程序员小组（4.3.2节）形成对比。然后讨论了一种利用两种方法优点的小组组织（4.4节），在4.5节中讨论了（Microsoft公司使用的）同步－稳定小组，然后在4.6节讨论了极限编程（XP）小组，并在4.7节讨论了人员能力成熟度模型（P－CMM）。最后，在4.8节描述了为某一给定项目选择最优小组组织所涉及的因素。

## 进一步阅读

小组组织的经典著作是［Weinberg，1971；Baker，1972；and Brooks，1975］。此话题的较新的著作是［DeMarco and Lister，1987］和［Cusumano and Selby，1995］。关于小组交互进展方面的有趣描述可以在［Mackey，1999］中找到。小组组织和管理的文章可在《Communication of the ACM》杂志的1993年10月刊中找到。［Royce，1998］的第11章包含了一些小组成员担当的角色的有用信息。

同步－稳定小组在［Cusumano and Selby，1997］中有概述，并在［Cusumano and Selby，1995］中有详细描述。从［McConnell，1996］中可以深入了解同步－稳定小组。在［Beck，2000］中描述了极限编程。在《IEEE Software》2001年11/12月刊中有一些有关极限编程方面的文章，特别是［Grenning，2001，and Paulk，2001］。［Boehm，2002］和［DeMarco and Boehm，2002］中介绍了敏捷方法的其他一些观点。［Williams，Kessler，Cunningham，and Jeffries，2000］中描述了结对编程的试验，它是极限编程的一个组成部分。P－CMM在［Curtis，Hefley，and Miller，2002］中有介绍。

## 习题

4.1 要开发一个工资单项目，你如何组织小组？如果开发一个最新的军用通信软件，你如

何组织小组？解释你的答案。

4.2　你刚组建了一个新的软件公司，所有雇员都是刚毕业的大学生；这是他们的第一次编程工作。在你的组织中使用民主小组可能吗，如果可能，怎样实现？

4.3　以民主小组的形式组织一个学生编程小组，关于小组中的学生会有什么样的推断？

4.4　以主程序员小组的形式组织一个学生编程小组，关于小组中的学生会有什么样的推断？

4.5　要比较一个大型软件公司里两个不同的小组组织——$TO_1$ 和 $TO_2$，提出下面的试验：两个不同的小组建造相同的软件产品，一个按照 $TO_1$ 来组织，另一个按照 $TO_2$ 来组织。公司估计每组需要大约 18 个月完成产品。请列出三条理由来说明这个试验是不可行的，并且不会产生任何有意义的结果。

4.6　为什么极限编程小组需要共享一台计算机？

4.7　你愿意工作在一个使用同步－稳定小组的组织里吗？说明你的理由。

4.8　（学期项目）开发附录 A 中描述的 Ophelia’s Oasis 产品最宜采用什么类型的小组组织？

4.9　（软件工程读物）你的导师将分发 Grenning［2001］的复印件，你愿意成为极限编程小组的成员吗？

## 参考文献


[Baker, 1972] F. T. BAKER, “Chief Programmer Team Management of Production Programming,” *IBM Systems Journal* **11** (No. 1, 1972), pp. 56–73.

[Beck, 2000] K. BECK, *Extreme Programming Explained: Embrace Change,* Addison Wesley Longman, Reading, MA, 2000.

[Beck et al., 2001] K. BECK, M. BEEDLE, A. COCKBURN, W. CUNNINGHAM, M. FOWLER, J. GRENNING, J. HIGHSMITH, A. HUNT, R. JEFFRIES, J. KERN, B. MARICK, R. C. MARTIN, S. MELLOR, K. SCHWABER, J. SUTHERLAND, D. THOMAS, AND A. VAN BENNEKUM, “Manifesto for Agile Software Development,” agilemanifesto.org, 2001.

[Boehm, 2002] B. W. BOEHM, “Get Ready for Agile Methods, with Care,” *IEEE Computer* **35** (January 2002), pp. 64–69.

[Brooks, 1975] F. P. BROOKS, JR., *The Mythical Man-Month: Essays in Software Engineering,* Addison Wesley, Reading, MA, 1975; Twentieth Anniversary Edition, Addison Wesley, Reading, MA, 1995.

[Business Week Online, 1999] *Business Week Online,* www.businessweek.com/1999/99_08/b3617025.htm, February 2, 1999.

[Curtis, Hefley, and Miller, 2002] B. CURTIS, W. E. HEFLEY, AND S. A. MILLER, *The People Capability Maturity Model: Guidelines for Improving the Workforce,* Addison Wesley, Reading, MA, 2002.

[Cusumano and Selby, 1995] M. A. CUSUMANO AND R. W. SELBY, *Microsoft Secrets: How the World’s Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People,* The Free Press/Simon and Schuster, New York, 1995.

[Cusumano and Selby, 1997] M. A. CUSUMANO AND R. W. SELBY, “How Microsoft Builds Software,” *Communications of the ACM* **40** (June 1997), pp. 53–61.

[DeMarco and Boehm, 2002] T. DEMARCO AND B. BOEHM, “The Agile Methods Fray,” *IEEE Computer* **35** (June 2002), pp. 90–92.

[DeMarco and Lister, 1987] T. DEMARCO AND T. LISTER, *Peopleware: Productive Projects and Teams,* Dorset House, New York, 1987.

[Grenning, 2001] J. GRENNING, “Launching Extreme Programming at a Process-Intensive Company,” *IEEE Software* **18** (November/December 2001), pp. 27–33.

[Mackey, 1999] K. MACKEY, “Stages of Team Development,” *IEEE Software* **16** (July/August 1999),

pp. 90–91.

[Mantei, 1981] M. MANTEI, "The Effect of Programming Team Structures on Programming Tasks," *Communications of the ACM* **24** (March 1981), pp. 106–13.

[McConnell, 1996] S. MCCONNELL, "Daily Build and Smoke Test," *IEEE Software* **13** (July/August 1996), pp. 144, 143.

[Paulk, 2001] M. C. PAULK, "Extreme Programming from a CMM Perspective," *IEEE Software* **18** (November/December 2001), pp. 19–26.

[Royce, 1998] W. ROYCE, *Software Project Management: A Unified Framework,* Addison Wesley, Reading, MA, 1998.

[Weinberg, 1971] G. M. WEINBERG, *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.

[Williams, Kessler, Cunningham, and Jeffries, 2000] L. WILLIAMS, R. R. KESSLER, W. CUNNINGHAM, AND R. JEFFRIES, "Strengthening the Case for Pair Programming," *IEEE Software* **17** (July/August 2000), pp. 19–25.

# 第 5 章　软件工程工具

**学习目标**

学完本章之后，你应当能够：

- 理解逐步求精法和在实践中应用它的重要性；
- 应用成本效益分析；
- 选择适当的软件度量；
- 讨论 CASE 的范围和分类法；
- 描述版本控制工具、配置控制工具以及构建工具；
- 理解 CASE 的重要性。

软件工程师需要两种类型的工具。首先是用于软件开发的分析工具，例如逐步求精法和成本效益分析。然后还有软件工具，也就是帮助软件工程师小组开发和维护软件的产品。这些通常被称为 CASE 工具（CASE 是计算机辅助软件开发工程的缩写）。本章重点讨论这两种类型的软件工程工具，首先是理论上的（分析）工具，然后是软件（CASE）工具。首先从逐步求精法开始。

## 5.1　逐步求精法

逐步求精（在 2.5 节中介绍过）是一个解决问题的技术，是许多软件工程技术的基础。**逐步求精**可定义为“尽可能将细节的定义推延到最后，以便集中精力在重要的事项上”的一种方法。作为米勒法则（见 2.5 节）的结果，一次一个人最多只能集中精力于 7 桩（信息单位）事情。因此，我们使用逐步求精来推迟到后来做出不必要的决定，而将注意力集中在关键的事情上。

在本书中你会看到，逐步求精法潜在地支撑着许多规格说明技术、设计和实现技术，甚至测试和集成技术。逐步求精在面向对象范型内具有重要意义，因为它背后的生命周期模型是迭代和递增的。

下面的小型实例研究说明了如何在产品的设计中使用逐步求精法。

### 逐步求精法小型实例研究

本节的小型实例研究看起来很不起眼，它用来更新一个顺序主文件，一个在许多应用领域中常见的操作。选择这个熟悉的例子是经过考虑的，它能使你关注逐步求精法本身，而不是应用领域的问题。

设计一个产品，更新包含有《True Life Software Disasters》月刊订户的人名和地址数据的顺序主文件。共有三种类型的事务：插入、修改和删除，分别对应事务代码 1、2 和 3。因此，事务类型是

- 类型 1：INSERT（插入一个新订户到主文件中）。
- 类型 2：MODIFY（修改一个已经存在的订户记录）。

• 类型 3：DELETE（删除一个已经存在的订户记录）。

事务是按订户人名的字母顺序排序的。如果对一个指定的订户有两个以上的事务，则已对那个用户的事务重新排序了，以便插入发生在修改之前，而修改发生在删除之前。

设计解决方案的第一步是建立输入事务的典型文件，如表 5-1 所示。该文件包含 5 个记录：DELETE Brown、INSERT Harris、MODIFY Jones、DELETE Jones 和 INSERT Smith。（在一次运行中对相同的订户进行修改和删除并不奇怪。）

表 5-1　为顺序主文件的更新输入事务记录

| 事务类型 | 姓　名 | 地　址 |
|---|---|---|
| 3 | Brown | |
| 1 | Harris | 2 Oak Lane, Townsville |
| 2 | Jones | Box 345, Tarrytown |
| 3 | Jones | |
| 1 | Smith | 1304 Elm Avenue, Oak City |

问题可以如图 5-1 所示，有两个输入文件：

1）旧的主文件人名和地址记录。

2）事务文件。

还有三个输出文件：

3）新的主文件人名和地址记录。

4）异常报告。

5）概要和工作结束消息。

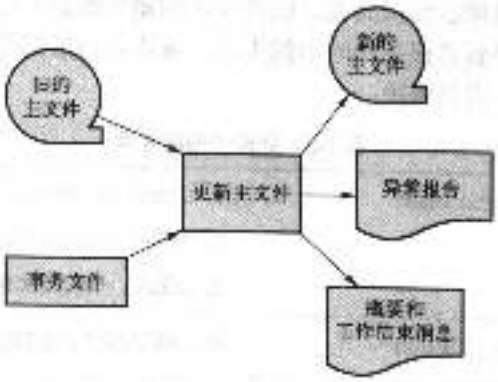



图 5-1　顺序主文件更新的表示

为开始设计过程，开始点是图 5-2 所示的更新主文件方框，这个方框可以分解为三个方框，分别称为输入、处理和输出。我们的假设是，当处理需要一个记录，我们能够做到在那个时间里产生正确的记录。同样，我们能够在当时把正确的记录写入正确的文件中。所以，该技术是把输入和输出两方面问题分离出去，集中精力在处理上。这个处理是什么样的？要

确定它做什么，考虑图 5-3 所示的例子。第一个事务记录（Brown）的关键词与第一个旧主文件记录（Abel）的关键词进行对比。因为 Brown 在 Abel 后面，将 Abel 记录写入新的主文件，读取下一个旧主文件记录（Brown）。在这种情况下，事务记录的关键词与旧主文件记录的关键词相匹配，又因为事务的类型是 3（DELETE），所以必须删除 Brown 记录，这通过不拷贝 Brown 记录到新主文件中来实现。读取下一个事务记录（Harris）和旧的主文件记录（James），在相应的缓冲区里覆盖 Brown 记录，Harris 记录在 James 记录之前，因此将其写入新的主文件中；读取下一个事务记录（Jones），因为 Jones 在 James 之后，将 James 记录写入新主文件，读取下一个旧主文件记录；就是 Jones。就像从事务文件中看到的，先修改 Jones 记录，然后将它删除，以便读取下一个事务记录（Smith）和下一个旧主文件记录（也是 Smith）。遗憾的是，事务类型是 1（INSERT），但 Smith 已经在主文件中，所以在数据中有某种错误，将 Smith 记录写入异常报告中。更准确地说，将 Smith 事务记录写入了异常报告，而将 Smith 旧的主文件记录写入新的主文件。

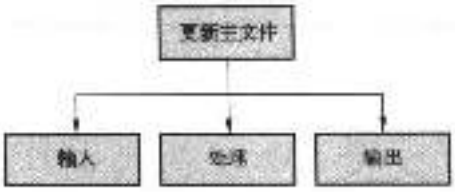



图 5-2　设计的第一次求精




图 5-3　事务文件、旧的主文件、新的主文件和异常报告

既然已经明白了处理过程（可以如表 5-2 那样表示它），接下来，可以对图 5-2 中的处理方框提炼，形成如图 5-4 所示的第二次求精。到输入和输出方框的虚线表示关于如何处理输入和输出的决定推迟至较晚的求精处理，该图的余下部分是处理的流程图，或者说对该流程图进行较早的求精。就像已经提到的，已将输入和输出推迟。还有，在此没有规定文件结束的条件，也没有规定遇到错误条件时如何去做。逐步求精法的优点就是这些和类似的问题可以在后面的求精过程中得以解决。

表 5-2　处理的图解表示

| | |
|---|---|
| 事务记录关键词=旧的主文件记录关键词 | 1. INSERT：打印错误信息<br>2. MODIFY：修改主文件记录<br>3. DELETE：*删除主文件记录 |
| 事务记录关键词>旧的主文件记录关键词 | 拷贝旧的主文件记录到新的主文件中 |
| 事务记录关键词<旧的主文件记录关键词 | 1. INSERT：将事务记录写入新的主文件<br>2. MODIFY：打印错误信息<br>3. DELETE：打印错误信息 |

*主文件记录的删除通过不向新的主文件拷贝记录来实现。

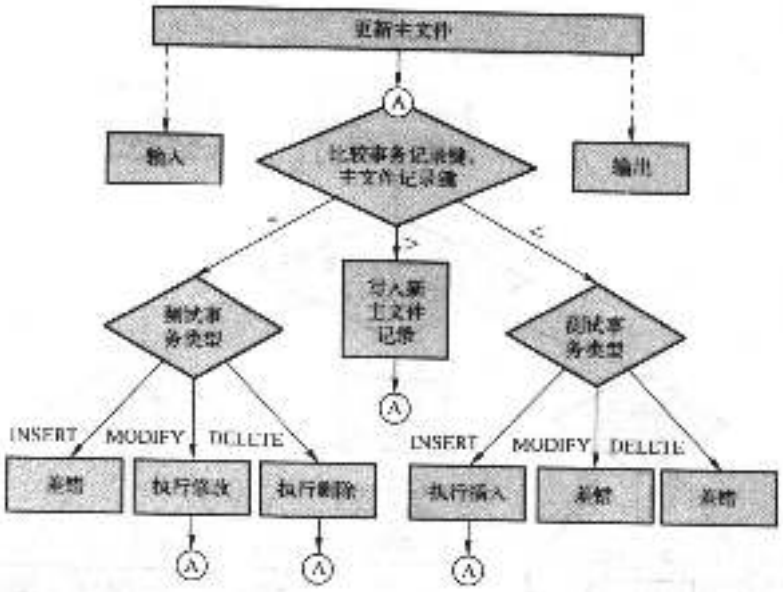



图 5-4 设计的第二次求精

下一步是求精图 5-4 中的输入和输出两个方框，形成图 5-5。还没有处理到文件结束的条件，也没有写入工作结束消息。这些仍然可在后面的迭代中完成。然而关键的是图 5-5 的设计有一个主要的错误。为理解这一点，考虑如图 5-3 中的数据的情形，当前的事务是 2 Jones，也就是修改 Jones，并且当前的旧主文件记录是 Jones。在图 5-5 的设计中，因为事务记录的关键词与旧主文件记录的关键词相同，最左边的路径到达测试事务类型判决框。因为当前的事务类型是 MODIFY，修改旧的主文件记录并写入新的主文件，然后读取下一个事务记录，该记录是 3 Jones，也就是删除 Jones。但已将修改后的 Jones 记录写入了新的主文件。

读者也许想知道为什么这里故意给出不正确的求精，原因在于使用逐步求精法时，有必要在进入下一次求精过程之前桌面检查随后的每个求精过程。如果某次求精过程出现了错误，不必从头开始重做一遍，只需回到前一次求精的位置，从那里开始。在本例中，第二次求精（图 5-4）是正确的，所以它可作为第三次求精的基础。这次，设计使用 1 级前瞻（lookahead），也就是只有在分析了一个事务记录的下一个事务记录之后，才能处理该事务记录。具体的细节留做练习，参见习题 5.1。

在第四次求精过程中，迄今为止被忽略的像打开和关闭文件这样的细节必须处理。对于逐步求精法，这样的细节是在设计的逻辑完全开发出来后，最后处理的。显然，不打开和关闭文件，产品是不可能执行的，但重要的是应当在设计过程中的这个阶段，处理打开和关闭文件。当进行设计时，设计者集中精力关注的 7 个左右的程序块是不应该包含打开和关闭文件这样的细节的；打开和关闭文件与设计本身无关，它们只是作为任何设计的一部分的实现细节。然而在后来的求精阶段中，打开和关闭文件变得重要起来。换句话说，逐步求精法可看成是建立某个阶段内需解决的各种问题的优先级的一种技术，逐步求精法能确保每个问题都得到解决，并且在合适的时间解决，不需要同时处理超过 7±2 个程序块。

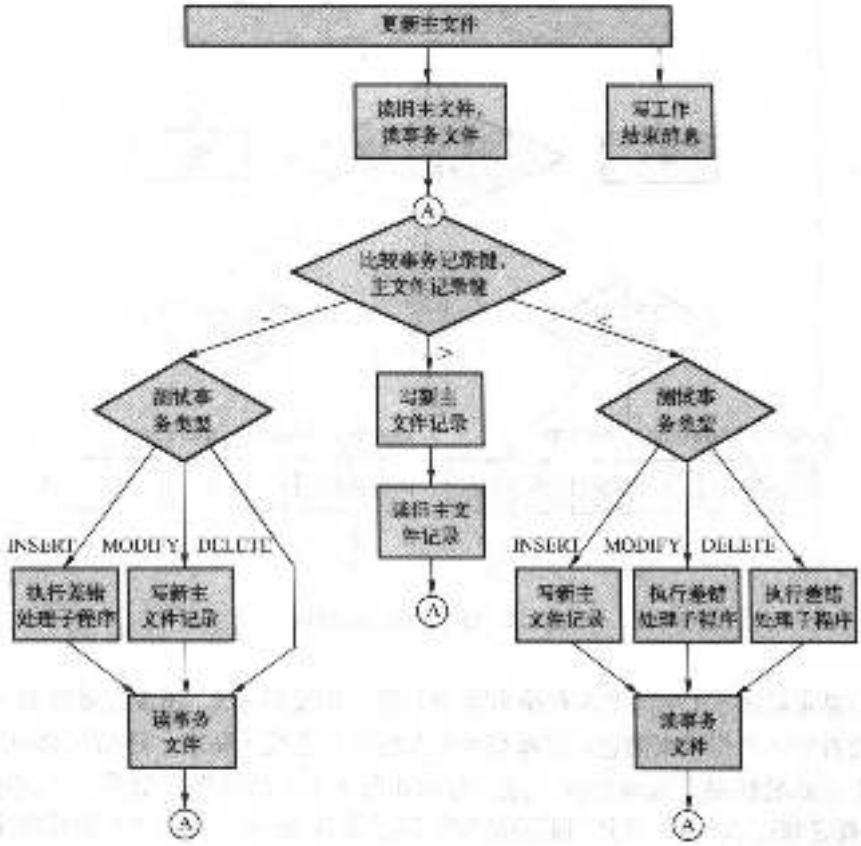



图 5-5　设计的第三次求精（设计有主要错误）

逐步求精法一词最早由 Wirth 引入［Wirth，1971］。在前面的例子中，逐步求精法用于流程图，而 Wirth 把这项技术用于伪代码。逐步求精应用的具体表现形式并不重要，逐步求精法是一项通用技术，可用于软件开发的每个阶段，表现形式也可以多种多样。

米勒法则是人类精神能力的一个基本限制，因为我们不能超出自然，我们必须适应自然，接受这个限制并在这个条件下尽可能做得更好。

逐步求精法的强大之处在于它能帮助软件工程师集中精力于当前开发阶段的相关方面（前面的例子是设计阶段），并忽略一些细节，尽管这些细节在总体方案中是必要的，也不能考虑它们，事实上在后面才考虑它们。不同于分而治之（divide-and-conquer）技术，把整个问题分解为重要程度相同的子问题；在逐步求精法中，问题的某个特定方面的重要性在一次又一次的求精中是变化的。最初，某个问题可能无关紧要，但后来同样的问题会变得相当重要。使用逐步求精法的难点在于确定当前的求精中必须处理的重要事项，以及哪些事项需推迟到后面的求精中解决。

与逐步求精法一样，成本－效益分析法是另一种贯穿于软件生命周期的基础理论性的软件工程技术。下一节将讨论这项技术。

## 5.2　成本－效益分析法

要确定一个可能的行为过程是否有利可图，一种方法是对比估计的未来收益和预测的未

来成本，这被称为**成本－效益分析法**（cost-benefit analysis）。下面举一个计算机行业里的成本－效益分析法的例子。1965 年 Krag 中心电子公司（Krag Central Electric Company，KCEC）要确定是否用计算机处理账单系统。当时账单由 80 个职员人工处理，每人平均每两个月向 KCEC 公司的客户邮寄一次账单。计算机化将要求 KCEC 公司购买或租用必需的软件和硬件，包括在打孔卡片或磁带上录入输入数据的数据收集设备。

计算机化的一个好处是账单可以每月邮寄一次，而不是每两个月邮寄一次，因而可以很大程度上加快公司的现金流通。进一步地，80 个记录账单的职员也可以由 11 个数据收集职员代替。如表 5-3 所示，在接下来的 7 年里，工资上的节余估计有 1 575 000 美元，而且现金流通快预计也可以带来 875 000 美元的收益，所以总的收益将是 245 万美元。另一方面，需要建立一个完整的数据处理部门，配备工资待遇优厚的计算机专业人员。在 7 年的时间里，成本可按如下估算：硬件和软件（包括维护）的成本估计为 125 万美元，在第一年里，将有约 35 万美元的转型成本，向客户解释新系统的额外成本约为 12.5 万美元，总的成本估算下来有 172.5 万美元，比预计的收益约少 75 万美元。KCEC 公司立即决定用计算机处理。

**表 5-3 KCEC 的成本效益分析数据**

| 效　益 | | 成　本 | |
|---|---|---|---|
| 工资节省（7 年） | 157.5 万美元 | 硬件和软件（7 年） | 125 万美元 |
| 现金流改善（7 年） | 87.5 万美元 | 转型成本（只是第一年） | 35 万美元 |
| | | 向客户解释（只是第一年） | 12.5 万美元 |
| 总收益 | 245 万美元 | 总成本 | 172.5 万美元 |

成本－效益分析法并不总是这样直截了当。一方面，管理顾问可能会估算工资节余，而会计可能计划加快现金流通，净现值（net present value，NPV）可用来处理资金成本中的变化，而软件工程顾问可以估算硬件、软件和转型的成本。但是我们如何确定与客户调整到计算机化有关的成本？或者我们如何计算整个人口接种麻疹疫苗所带来的收益？考虑市场窗口效应时，我们如何估计收益？或者说，第一个将新产品投入市场时所带来的收益，以及不是第一带来的代价（因此可能丢掉客户）。

关键是有形的收益容易计算，但无形的收益很难直接量化。给无形的收益确定金钱数量的一个实际的办法是做**假设**，这些假设必须与得到的收益估算值结合起来。毕竟管理者要做出决策，如果得不到数据，那么通过假设确定数据通常是这种情况下最好的选择。这种方法还有进一步的好处，如果其他什么人审核这组数据，并且据此数据中潜在的假设提出更好的假设，那么就能产生更好的数据，有关的无形收益也可以计算得更精确。对于无形的成本计算可以使用相同的技术。

成本－效益分析法是确定客户是否应当将他或她的业务计算机化的基本技术，以及如果确定使用计算机处理业务，应用何种方式来比较各种可选方案的成本和收益。例如，存储药品试验结果的软件产品可以有多种方式来实现，包括直接的文件和各种数据库管理系统。对于每一个可能的方案，都要计算成本和收益，收益和成本之间差异最大的方案将为最佳方案。

下面叙述本章最后一个理论工具——软件度量。

## 5.3 软件度量

3.13 节中讲到，没有度量（或**测度**，metrics）是不可能在软件开发过程的早期，在问题暴露之前检测到该问题的，这样度量可作为潜在问题的一个早期预警系统。有很多种度量可以使用，例如，代码行是度量产品规模的一种方法（参见 9.2.1 节）。如果定期进行 LOC 测量，则它们可提供项目进展快慢的度量。另外，每千行代码检测出的错误数是软件质量的一种度量，毕竟如果程序员一个月内连续写出 2000 行代码，却有一半因为不可用而被丢弃，那么这是没有用的。所以孤立地进行 LOC 不是一个非常有意义的度量。

一旦把产品安装到客户的计算机上，像平均故障间隔时间这样的度量可以表示产品的可靠性。如果某产品每隔一天出现一次故障，与类似的产品平均运行 9 个月不出一个故障相比，它的质量显然很低。

在整个软件开发过程中可能使用某种度量，例如，在每个阶段我们可以以人月为单位测量工作量（1 人月指的是一个人一个月的工作量）。职员的流动性是另一个重要的量度。高的流动性会影响目前的项目，因为一个新雇员需要时间学习与项目相关的内容（参见 4.1 节）。另外，新雇员需要在软件开发过程的某些方面进行培训，如果新雇员比他所替代的人缺乏软件工程方面的知识，则整个过程将受影响。当然，在整个开发过程中成本也是一个需要时常监测的重要度量。

本书讲述了许多不同的度量，一些是**产品度量**，也就是它们测量产品本身的一些特性，例如规模和可靠性。相反，另一些是**过程度量**，开发者使用这些度量推断有关软件开发过程的信息。这种类型的度量的一个典型例子是开发过程中错误检测的有效性，也就是开发过程中检测到的错误数量与产品整个生命周期期间检测到的错误数量之比。

许多度量对某给定的阶段来说是特定的，例如，代码行度量不能用在实现过程开始之前，而在审核规格说明过程中，每小时检测到的错误数只与规格说明阶段有关。在后续讲述软件开发过程的各阶段的章节中，我们将讨论与该阶段相关的度量。

成本与计算度量值所需搜集的数据相关，即使数据搜集是全自动的，积累所需信息的 CASE 工具（5.4 节）也不是免费的，而且解释该工具的输出结果需要耗费人力资源。要知道现在已有上百种（如果不是上千种的话）不同的度量，一个明显的问题是，一个软件组织应该测量什么？有 5 种主要的基本度量：

1）规模（以代码行或以更好的、更有意义的比如 9.2.1 节中介绍的那些度量计）。
2）成本（以美元计）。
3）持续时间（以月计）。
4）工作量（以人月计）。
5）质量（以检测到的错误数计）。

这些度量中的每一个都必须按阶段测量，根据从这些基本度量获得的数据，管理者可以发现软件组织内部的问题，例如，在设计阶段的高错误率或代码产量远低于行业平均水平。一旦发现了问题，就可以考虑解决方案。为监测方案是否成功，可以引入更具体的度量。例如，可以搜集每个程序员的错误率数据或进行一次用户满意度调查。这样，除了那 5 种基本的度量之外，为了一个特定目标可以进行更具体的数据搜集和分析。

最后，度量的一个特点仍是相当有争议。人们已经提出许多普遍使用的度量是否具有合理性？这样的一些问题将在 14.13.2 节讨论。尽管普遍认为只有度量了软件过程，才能控制软件过程，但在明确应该度量什么的问题上仍有一些不同的看法［Fenton and Pfleeger, 1997］。

下面我们从讨论理论工具转向讨论软件（CASE）工具。

## 5.4　CASE

在软件产品的开发过程中，需要进行许多不同的操作。典型的活动包括评估资源要求、写出规格说明文档、进行集成测试以及编写用户操作指南。遗憾的是，这些操作和软件开发过程中的其他操作不能完全由计算机自动化地执行，而需要人的参与。

然而，计算机可以辅助开发的每一步，本节的题目，CASE 代表计算机辅助软件工程（Computer-Aided（or Assisted）Software Engineering）（另外的含意也可见下面的“如果你想知道”部分）。计算机可以完成与软件开发有关的大部分繁重工作，包括创建并组织所有诸如计划、合同、规格说明、设计、源代码和管理信息这样的人工制品。文档对于软件的开发和维护是至关重要的，但大多数从事软件开发的个人并不喜欢生成或更新文档。维护计算机上的图表特别有用，因为允许轻松地修改它。

**如果你想知道**

如 1.11 节所解释的那样，对于软件工程师来说，术语“系统”经常用来指一个软件和硬件的混合体。**系统工程**领域的活动范围十分广泛，它从定义客户的需求和要求开始，一直到它们实现于构建的系统中。随后，在系统已经交付给客户后，在经过成功验收测试之后，它在整个生命周期过程中经受额外的修改，从而去除缺陷，或者加入一些改进需求，或者进行一些适应性改进［Tomer and Schach, 2002］。

这样，系统工程和软件工程之间有很大的相似之处。因此对于系统工程师，缩略词 CASE 代表“计算机辅助系统设计”也就不足为怪了。因为软件经常在系统工程中发挥主要作用，在系统工程范畴内，有时很难知道 CASE 代表哪个版本的缩写。

但 CASE 并不只限于对文档的帮助，特别地，计算机可以帮助软件工程师应付软件开发的复杂性，特别是在处理所有的细节上。CASE 包含了计算机支持软件工程的所有方面。同时，要牢记 CASE 代表计算机辅助软件工程，不是计算机自动化的软件工程——还没有一台计算机在软件的开发或维护上可以完全替代人，至少在可预知的未来，计算机仍是软件专业人员的一个工具。

## 5.5　CASE 的分类

CASE 的最简形式是软件**工具**，只在软件生产的某一方面起帮助作用的软件产品。目前 CASE 工具用于软件生命周期的每一个阶段。例如，市场上有许多种工具，大部分用于个人计算机，它帮助构建软件产品的图形表示，诸如流程图和 UML 图。在软件开发过程的较早阶段（指的是需求阶段、分析阶段和设计阶段）帮助开发者的 CASE 工具有时被称为**高端 CASE** 或**前端**（front-end）**工具**，而那些帮助实现、集成和维护的 CASE 工具被称为**低端 CASE** 或**后端**（back-end）**工具**。图 5-6a 表示了一个在需求阶段起辅助作用的 CASE 工具。

一类重要的CASE工具是**数据字典**（data dictionary），它是在产品中定义的所有数据的计算机化列表。一个大型的产品可能包含几万（或几十万）的数据词条，计算机最适于存储像变量名、变量类型、定义的每个变量的存储位置、过程名和参数以及它们的类型这样的信息。每个数据字典记录的最重要的部分是对该词条的描述，例如，“输入新生婴儿体重的过程”、“计算药物的适当剂量的过程”或“列出按先后时间排序的飞机到达时刻表”的过程。

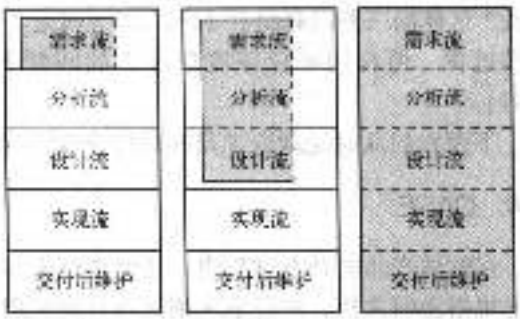



图5.6　工具、工作平台和环境的表示

**数据字典与一致性检查器**（consistency checker）结合在一起会增强功能，一致性检查器作为一种工具，它检验规格说明文档中的每一个数据项是否反映在设计中，反过来也检验设计中的每一项在规格说明文档中是否有定义。

数据字典的另一个用处是为报表生成器和屏幕生成器提供数据。**报表生成器**（report generator）产生生成报表所需的代码；**屏幕生成器**（screen generator）能够帮助软件开发者产生用于数据捕获屏幕的代码。假设需要设计一个屏幕，用于输入连锁书店各分店的每周销量。分店的号码是一个四位的整数，范围从1000到4500或者从8000到8999。从屏幕上方的第三行输入。这个信息传给屏幕生成器，然后屏幕生成器在屏幕上方的第三行自动生成代码以显示字符串 BRANCH NUMBER_ _ _ _，并把光标置于第一个下划线字符上。用户每输入一个数字，都将显示它，同时光标移向下一个下划线字符。屏幕生成器还生成代码来检验用户输入的数字，确认输入的四位数在特定的范围内。如果用户的输入无效，或者用户按下“?”键，则显示帮助信息。

使用这样的生成器可以使快速原型很快地建造起来，进一步说，图形表示工具与数据字典一道，一致性检查器、报表生成器和屏幕生成器一起构成了需求、分析和设计的**工作平台**（workbench），它支持快速原型开发。Software through Pictures就是结合了所有这些特性的商用工作平台的例子[○]。

另一类工作平台是需求管理工作平台，这类工作平台允许系统分析员组织和跟踪软件开发项目的需求。RequisitePro是这类工作平台的商用例子。

因而，一个CASE工作平台是一些工具的集合，它们共同支持一个或两个行为，这里一个行为是一个相关任务的集合。例如，编写代码行为包括编辑、编译、链接、测试和调试。一个行为与生命周期模型的一个阶段不能等同，事实上，一个行为的任务甚至可以跨阶段。例如，一个项目管理工作平台用于项目的每个阶段，一个编码工作平台可用于快速原型，也可用于实现、集成和维护阶段。图5.6b表示一个高端CASE工具的工作平台，该工作平台包括图5-6a的需求阶段工具，也包括部分分析和设计阶段的工具。

将CASE技术从工具到工作平台的发展而继续下去，下一个内容就是CASE环境。与支

○　本书引用某一CASE工具绝不意味着该CASE工具为本书作者或出版商所采用，本书提到的每一种CASE工具均是CASE工具类的典型示例。

持一个或两个行为的工作平台不同，一个**环境**支持整个软件开发过程，或者至少是软件开发过程的大部分［Fuggetta，1993］。图 5-6c 表示一个支持生命周期的各个阶段的所有方面的环境。第 14 章将详细讨论环境。

我们已经建立了 CASE 分类（它们命名为：工具、工作平台和环境），下面讨论 CASE 的范围。

## 5.6　CASE 的范围

前面提到，使用 CASE 技术的一个主要原因是总是需要有精确、最新和可用的文档。例如，假设人工生成规格说明，开发小组的成员无法确认某个规格说明文档是当前的版本还是旧版本，也无法知道文档上手写的修改之处是当前规格说明的一部分呢，还是只不过是一个后来被否决了的建议。另一方面，如果产品的规格说明是使用 CASE 工具生成的，那么在任何时候都只有一份规格说明，即通过 CASE 工具访问的在线版本。那么，如果规格说明改变了，开发小组的成员能够很容易访问该文档，并确信他们看到的是当前版本。另外，无需对照规格说明文档的修改，一致性检查器就会标志出任何设计上的改变。

程序员也需要**在线文档**。例如，必须提供关于操作系统、编辑器、编程语言等的在线帮助信息。另外，程序员还要查阅许多种手册，比如编辑器手册和编程手册。只要可能，程序员最希望能够在线得到这些手册。抛开任何东西都能在手边的便利不说，通常通过计算机查询要比找到合适的手册并翻到所需的那一项快得多。另外，更新在线手册比在组织内找到所有版本手册的硬拷贝并做必要的修改更容易。所以，在线文档比相同内容的硬拷贝看来更准确，这也是给程序员提供在线文档的另一个原因。UNIX 手册页便是这样的在线文档的一个例子［Sobell，1995］。CASE 还有助于小组成员间的沟通。**电子邮件**（e-mail）已成为一般办公室的一个重要部分，就像计算机或传真机一样。电子邮件有许多好处，从软件生产的角度看，如果与某项目有关的所有电子邮件存储在一个指定的邮箱里，那将是项目期间做出的决定的书面记录，这可防止以后出现争执。现在，许多 CASE 环境和一些 CASE 工作平台加入了电子邮件系统。在其他的组织里，通过诸如 Netscape 这样的万维网浏览器实现电子邮件系统。其他同样重要的工具是**电子数据表格**（spreadsheet）和**文字处理器**。

**编程工具**（coding tools）一词指的是诸如文本编辑器、调试器和灵巧打印机这样的 CASE 工具，它们可简化程序员的任务，减少许多程序员在工作中会遇到的挫折，并提高产品效率。在讨论这些工具之前，要先明确三个定义。**小编程**（programming-in-the-small）指的是在单个制品的代码级进行的软件开发，而**大编程**（programming-in-the-large）指的是在模块级进行的软件开发［DeRemer and Kron，1976］，后者包含着结构化设计和集成等方面。**多编程**（programming-in-the-many）指的是由一个小组进行的软件开发。有时，小组工作在模块级；有时，小组工作在代码级。因此，多编程结合了小编程和大编程两方面。

一个**结构编辑器**（structure editor）是理解实现语言的文本编辑器，也就是说，一个结构编辑器可以随时检测到程序员键入的语法错误，这加快了实现阶段，因为时间没有浪费在琐碎的编辑上。结构编辑器适用于多种语言、操作系统和硬件。因为结构编辑器具有编程语言的知识，容易把**灵巧打印机**（或者**格式器**）并入编辑器，以确保代码总是具有良好的视觉外观。例如，支持 C++ 的灵巧打印机能确保每一个“}”都能与对应的“{”缩进相同的格数。保留字会自动以粗体形式表现，以便突出显示；设计上仔细考虑了缩进，提高了可读

性。现在，这种结构化编辑器构成了无数编程平台的一部分，如 Visual C++ 和 JBuilder。

现在看看在代码内调用一个方法（method）存在的问题，它只能在链接时才发现该方法不存在或以某种方式被错误地定义过。为此结构编辑器需要支持**在线接口检查**，也就是说，就像结构编辑器知道程序员声明的每个变量的名称一样，它也必须知道产品内定义的每个方法。例如，如果程序员输入一个调用：

```
average = dataArray. computeAverage (numberOfValues);
```

但 computeAverage 方法还没有定义，则编辑器立即响应出一条信息：

```
Method computeAverage not known
```

这时，程序员有两种选择，要么纠正方法的名称，要么声明一个新的方法，命名为 computeAverage。如果选择了第二个选项，程序员还必须指定新方法的参数。声明新方法时，必须提供参数类型，因为进行在线接口检查必须检验全部接口信息，而不只是方法的名称。通常会有这样的错误：方法 p 调用方法 q 比如说传递 4 个参数，而方法 q 却规定有 5 个参数。当调用正确地使用了 4 个参数，但两个参数的顺序颠倒了，这种错误就更难检测到。例如，方法 q 的声明可能是：

```
void q (float floatVar, int intVar, string s1, string s2)
```

而调用是：

```
q (intVar, floatVar, s1, s2);
```

在调用语句中前面的两个参数顺序颠倒了，Java 编译器和链接器能检测到这个错误，但只在后面调用它们时才发现。相反，在线接口检查器能够立即检测到这个和类似的错误。另外，如果编辑器有一个帮助工具，程序员可以在编调用方法 q 的代码之前，请求获得关于方法 q 的准确参数的在线信息。更好的是，编辑器应生成该调用的一个模板，显示每个参数的类型。程序员只要用正确类型的实参替代每个形参即可。

在线接口检查的一个主要优点是，可以立即标志出由调用参数个数错或参数类型错的方法所引起的难于检测的错误。在线接口信息对于高效高质量的软件生产很重要，特别是当软件由小组进行开发时（多编程）。考虑到在任何时候所有代码制品对所有编程小组成员都是可用的，在线接口信息就很必要，进一步说，如果一个程序员修改了 vaporCheck 方法的接口，也许是修改了一个参数的类型，由 **int** 改为 **float**，或者增加了一个参数，那么调用 vaporCheck的每个组件一定自动地失效，直到修改相关的调用语句以反映事件的新状态。

即使有一个和在线接口检查器结合在一起的结构化编辑器，程序员还是需要从编辑器中退出，然后调用编译器和链接器。很明显不会有编辑错误，但还是要调用编译器以生成代码，然后调用链接器。由于在线接口检查器的存在，程序员可再次确信所有的外部参考都是正确的，但是仍需要链接器来链接产品。解决的办法可以是在编辑器内加入**操作系统前端**（工具），即程序员应当能够从编辑器内给出操作系统命令。为了让编辑器调用编译器、链接器、装载器以及其他任何需要使代码制品执行的系统软件，程序员应当能够键入一个简单的 go 或 run 命令，或使用鼠标选择合适的图标或菜单选项。在 UNIX 系统中，可使用 make 命令（5.9 节）或通过调用外壳（shell）脚本做到这一点［Sobell，1995］。这样的前端工具也

可用于其他操作系统。

最让人伤心的编程经历是产品执行了一秒左右，突然停止，打印出像这样的一条消息：

```
Overflow at 506
```

程序员通常使用诸如 Java 或 C++ 这样的高级语言编程，而不是用像汇编程序或机器代码这样的低级语言进行编程。但当调试支持信息为 "Overflow at 506" 之类时，程序员不得不检查机器代码核心转储、汇编程序清单和许多类似的低级语言文档，因而破坏了高级语言编程的整个好处。当出现了 "Core dumped" 或者 "Segmentation fault" 这样的 UNIX 信息时也会有同样的问题，用户同样需要检查低级语言信息。

在出现错误时，如图 5-7 所示的信息与前面的简短的错误信息相比是个很大的进步，程序员可立即明白方法出错是因为试图除以 0。更有用的是，操作系统输入编辑模式并自动显示检出错误的所在行，这里是第 6 行，也同时显示出这行前后的 4～5 行。程序员可能就会明白什么引起了错误并做必要的修改。




图 5-7 源代码级调试器的输出

另一种类型的源代码级的调试是跟踪。在有 CASE 工具之前，程序员需要在代码中的合适位置人工插入打印语句，以便在执行的时候显示行号和相关变量的值。现在可以给**源代码级调试器**下一道命令，它自动产生跟踪输出。更好的是交互式**源代码级调试器**。假设变量 escapeVelocity 的值看起来不对，方法 computeTrajectory 看起来也有错，程序员使用交互式源代码级调试器可以在代码中建立断点。当执行到断点时，程序将停下来并进入调试状态。程序员此时让调试器跟踪变量 escapeVelocity 和方法 computeTrajectory。也就是说，接下来每次使用 escapeVelocity 的值或改变它时，执行将再次停下来。程序员可以选择输入更进一步的调试命令，例如，要求显示某个变量的值。当然，程序员也可以选择继续在调试状态下运行或回到正常的执行状态。程序员也可以在进入或退出方法 computeTrajectory 时与调试器进行类似的交互。这样的交互式源代码级调试器在软件产品出故障时向程序员提供了几乎每一种可能想到的帮助类型。UNIX 调试器 dbx 也是这样的一种 CASE 工具。

前面多次提到过，在线得到各种文档是非常必要的。程序员需要在编辑器内能够访问到所有他们可能需要的文档。

现在我们所讨论的结构化编辑器，具有在线接口检查能力、操作系统前端、源代码级调试器和在线文档，这些组成了一个完备而高效的编程工作平台。

这种类型的工作平台没什么新意，早在 1980 年，FLOW 软件开发工作平台就支持前面提到的所有特性 [Dooley and Schach, 1985]。所以，在暂时生成一个原型之前，提出最小但基本的编程工作平台并不困难。恰恰相反，必要的技术已经存在了 20 年，有些令人诧异的是，有些程序员还在使用 "老式的方法" 编程，而不是使用像 Sun ONE Studio 这样的工作平台。

版本控制工具是一个基本工具，特别是当由小组来开发软件时。

## 5.7　软件版本

无论何时维护产品，都至少会有两个**版本**的产品：老版本和新版本。因为产品是由代码制品组成的，修改过的每个组件制品也会有两个或更多的版本。

下面首先讨论交付后维护范围内的版本控制，然后再扩展到该过程的较早阶段。

### 5.7.1　修订版

假设已在许多不同的地点安装了产品，如果在一个制品中发现了错误，之后修复了该制品。经过适当的修改后，该制品将会有两个版本，老版本和将要替代老版本的新版本，新版本被称为**修订版**（revision）。显然，解决多个版本并存的问题很容易——抛弃所有的老版本，只留正确的那一个。但那样做不明智。假设该制品的前一个版本是版本 $n$，新版本是版本 $n+1$。首先，并不能保证版本 $n+1$ 比版本 $n$ 都正确，尽管软件质量保证（SQA）小组已经全面地测试过了，但当用户在实际数据上运行产品的新版本时，无论是孤立的还是与产品的其余部分链接，还可能出现灾难性的后果。版本 $n$ 需要保留的第二个原因是，产品可能已经分发到许多地方，而这些地方并没有都安装了版本 $n+1$。如果从仍使用版本 $n$ 的地方传来一个错误报告，那么为分析这个新的错误，有必要将产品配置成与用户相同的配置，也就是使用该制品的版本 $n$。所以有必要保留每个制品的每一个版本。

1.3 节中讲到，好的维护扩展了产品的功能。在一些情况下写出了新的制品，在另一些情况下，修改现有制品从而加入这个新增的功能，这些新版本也是现有制品的修订版。进行适应性维护时制品也会被修改，也就是说，产品为适应变化的环境也要有所变化。对于纠错性维护，必须保留所有以前的版本，因为问题并不只在维护阶段产生，也会产生于前面的实现阶段。毕竟，一旦编完了一个制品，作为检错和纠错的结果，这个制品要经受连续不断的修改。结果，每个制品都有许多版本，一定要对这些版本保持某些控制，以确保开发小组的每个成员都知道哪一个版本是给定制品的当前版本。在我们给出这个问题的解决方案之前，必须要考虑得周到些。

### 5.7.2　变种版

考虑下面的例子，大多数计算机支持多种打印机，例如，个人计算机会支持喷墨打印机和激光打印机，所以操作系统必须有打印机驱动程序的两种**变种版**，每个对应一种打印机。与专门为了替代前面版本的修订版不同，变种版是为共存而设计的。需要变种版的另一个情况是产品要与多种操作系统或硬件接口，可能需要为每一种操作系统或硬件生成多个制品的不同变种版。图 5.8 示意了修订版和变种版。为进一步考虑复杂因素，每个变种版通常又有多个版本。对一个软件组织来说，为避免陷入多个版本的泥潭，需要使用 CASE 工具。

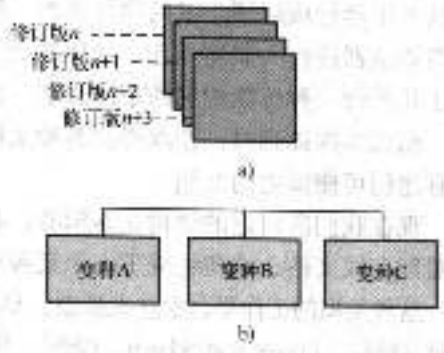



图 5.8　制品的多个版本示意图，a）表示修订版，b）表示变种版

## 5.8 配置控制

每个制品的代码以二种形式存在，第一种是源代码，现在通常使用像C++或Java这样的高级语言编写；然后是目标代码，通过编译源代码生成；在本书中，我们也将目标代码称为编译代码。最后每个制品的编译代码与运行时例程结合形成一个可执行载入映像。如图5-9所示，程序员可以使用每个制品的多种不同的版本，某个（完成的）产品的给定版本所赖以建造的每个制品的特定版本称为该产品那个版本的配置。

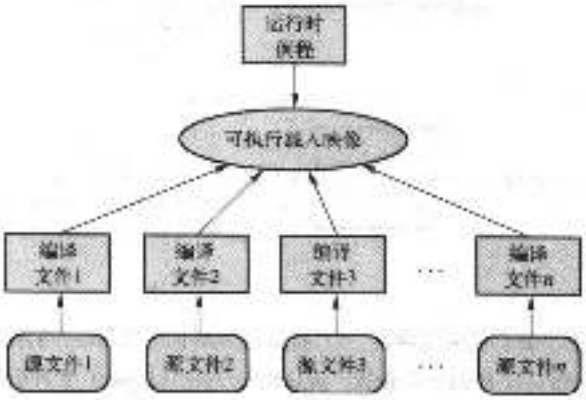



图5-9 可执行载入映像的组件

假设SQA小组交给程序员一个测试报告，证明一个制品在某组测试数据上有问题。首先要做的是试图重现问题，但程序员如何确定制品哪个变种版的哪个修订版进入了出现问题的那个产品版本？除非使用一个配置控制工具（下面将详细讨论），否则需要查看八进制或二进制的可执行载入映像才能指出错误源头。特别是，还得编译源代码的多个版本并与生成可执行载入映像的那个编译代码对比。尽管这可以做到，却需要很长时间，特别是当产品有几十个（或是几百个）制品而每个制品又都有多个版本的时候。所以，处理多个版本时必须解决两个问题。第一，有必要区分版本，以使将每个制品的正确版本编译并链接到产品中；第二，存在相反的问题：给定一个可执行载入映像，确定每个组件的哪个版本进入它了。

解决这个问题首先要有版本控制工具。许多操作系统，特别是用于大型计算机的操作系统，支持版本控制。但也有许多操作系统不支持版本控制，在这种情况下，需要一个单独的版本控制工具。版本控制中使用的一个常用技术是使每个文件的名称包含两部分，文件名本身和修订版本号。这样一个确认收到消息的制品就会有acknowledgeMessage/1、acknowledge Message/2等修订版，如图5-10a所示。程序员可以准确指明为完成某任务需要哪一个修订版。

关于多个变种版（不同情形下完成相同功能但稍有改变的版本），有一个有用的记法，即一个基本的文件名后跟着一个带圆括号的变种名[Babich, 1986]。这样两个打印机驱动程序被命名为printerDriver（inkJet）和printerDriver（laser）。

当然每个变种版会有多个修订版，如图5-10b所示的printerDriver（laser）/12、printerDriver（laser）/13和printerDriver（laser）/14。

版本控制工具是管理多个版本的第一步，一旦它就绪了，必须保留产品每个版本的细节

记录（或出处）。出处包含每个源代码组成部分的名称（包括变种版和修订版），使用的多种编译器和链接器的版本，创建产品的人的名字，当然还有生成产品的日期和时间。

版本控制在管理制品的多个版本和作为整体的产品方面有很大帮助，但由于与维护多个变种版有关的额外一些问题，版本控制还不够。

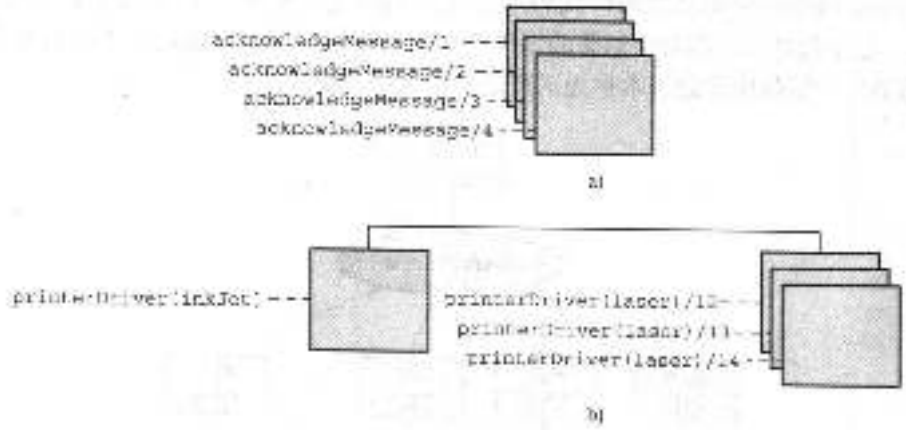



图 5-10　多个修订版和变种版。a）制品 acknowledgeMessage 的四个修订版。b）制品 printerDriver 的两个变种版，printerDriver（laser）变种版有三个修订版

考虑两个变种版 printerDriver（inkJet）和 printerDriver（laser）。假设在 printerDriver（inkJet）中发现了一个错误，并且猜想该错误发生在两个变种版都使用的那个制品中。那么不仅要修复 printerDriver（inkJet），还要修复 printerDriver（laser）。通常，一个制品如果有 $v$ 个变种版，则全部 $v$ 个变种版都需要修复。不仅如此，还应该以严格相同的方式修复它们。

解决这个问题的办法是只存储一个变种版，也就是 printerDriver（inkJet），然后其他任何变种版都根据从最初的版本到那个变种版本所做改变的列表存储。这个差异的列表被称为增量（delta）。这样，存储的是一个变种版和 $v-1$ 个增量。访问 printerDriver（inkJet）并应用增量就可以恢复 printerDriver（laser）变种版，通过改变适当的增量就可以改变 printerDriver（laser）。然而，对最初的变种版 printerDriver（inkJet）的任何修改都会自动地应用到所有其他的变种版上。

**配置控制工具**可以自动地管理多个变种版，但**配置控制**的作用不仅限于多个变种版。配置控制工具还能处理小组开发和维护时出现的问题，下一节将讨论它。

### 5.8.1　交付后维护期间的配置控制

如果多个程序员同时维护一个产品，会出现各种各样的麻烦事。例如，假设星期一早晨两个程序员被分别安排处理一份不同的故障报告，碰巧他们把错误定位在同一制品 mDual 的不同部分，准备去修复它。每个程序员都拷贝了一份该制品的当前版本 mDual/16，开始修复错误。第一个程序员修复了第一个错误，修改得到批准并替代了该制品，现在叫 mDual/17。一天后第二个程序员修复了第二个错误，修改也得到批准并安装了制品 mDual/18。遗憾的是第 17 修订版只包含了第一个程序员做出的修改，而第 18 版只包含第二个程序员做出

的修改，这样第一个程序员做出的修改丢掉了。

尽管让每个程序员生成制品的单独拷贝的想法比共同处理软件的同一部分要好，但对于团队维护来说这显然不合适，需要一种机制，每次只允许一个用户修改制品。

### 5.8.2 基准

维护的管理者必须建立一个**基准**，它是产品中所有制品的配置（版本集）。当要找寻错误时，维护程序员把任何所需的制品拷贝到他或她的**个人工作台**中。在这个个人工作台里，程序员可以做任何修改，对其他程序员决不会有任何影响，因为修改的只是该程序员的个人拷贝，基准版还保持未动。

一旦决定修改哪个制品以修复错误，程序员将“冻结”他或她要修改的那个制品的当前版本，其他程序员不能修改被冻结的制品。该维护程序员做完修改并经过测试后，就安装该制品的这个新版本，从而修改基准版。前一个被冻结的版本，仍保留以备后用（原因前面已解释过），但不能改变它。一旦安装了新版本，任何其他程序员都可以冻结这个新版本并修改它，结果产生的制品又变成下一个基准版。如果两个或更多的制品同时需要修改，可以进行类似的过程。

这个办法解决了制品 mDual 的问题。两个程序员制作 mDual/16 的个人拷贝，并使用它们来分析各自要修复的错误。第一个程序员确定了要做什么修改，冻结了 mDual/16，并修改它以便排除第一个错误。修改被测试通过后，产生的 mDual/17 版本成为基准版。同时，第二个程序员也已通过分析 mDual/16 的个人拷贝找到了第二个错误。然而不能对 mDual/16 进行修改，因为它被第一个程序员冻结了。一旦 mDual/17 成为基准版，第二个程序员就冻结它，并修改它。产生的制品现在作为 mDual/18 来安装，该制品体现了两个程序员的修改。mDual/16 和 mDual/17 修订版被保留，以便以后参考，但不能再修改它们。

### 5.8.3 产品开发过程中的配置控制

当一个制品正处在编码过程中时，软件版本变化太快，配置控制不太有用。然而，一旦该制品的编码已经完成，应立即由编写它的程序员非正式地测试（如 6.6 节所述）。在非正式测试的过程中，该制品又会经历多个版本。程序员满意了的时候，就把它交给 SQA 小组进行系统测试。该制品一经 SQA 小组测试通过，就准备集成到产品中。从那时起，它就应该经过同样的配置控制过程，就像维护阶段一样。对集成了的制品进行任何修改都会对整个产品产生影响，就像维护阶段做出修改一样。所以，配置控制不仅在维护阶段需要，在实现和集成阶段也需要。进一步说，管理者无法充分地监控开发过程，除非只要配置控制是合理的时候（也就是通过 SQA 小组以后），就将每个制品都处在它的控制下。如果正确地应用配置控制，管理者就能够知晓每个制品的状态，如果项目的最终期限看来要超出，就能提早做补救工作。

三个主要的 UNIX 版本控制工具是 sccs（source code control system，源代码控制系统）[Rochkind，1975]、rcs（revision control system，修订版控制系统）[Tichy，1985] 和 cvs（concurrent versions system，并行版本系统）[Loukides and Oram，1997]。PVCS 是一个广泛应用的商用配置控制工具。Microsoft SourceSafe 是用于个人计算机的配置控制工具。CVS 是一个公开源码的配置控制工具（在 1.11 节中描述了公开源码软件）。

## 5.9　建造工具

如果一个软件组织不希望购置全部的配置控制工具，那么至少要与版本控制工具一同使用**建造工具**，即帮助选择要链接的每个编译代码制品的正确版本，从而形成该产品的一个特定版本。在任何时候，每个制品的多个变种版和修订版都将在产品库中。任何版本控制工具都将帮助用户区分源代码制品的不同版本。但跟踪编译代码要更难，因为有些版本控制工具不把编译后版本的修改版号附上。

为解决这个问题，一些组织每天晚上自动编译每个制品的最新版本，从而确保所有的编译代码总是最新的。尽管这项技术可行，但也相当浪费计算机时间，因为要频繁地进行大量不必要的编译。UNIX 工具 make 可以解决这个问题［Feldman，1979］。对于每个可执行载入映像，程序员建立一个 Makefile，指明源文件和编译文件进入某一配置的层次。图 5-9 显示了这样的一个层次。像 C 或 C++ 中的包含文件这样更复杂的相关性也可由 make 处理。当程序员调用它时，该工具这样工作：UNIX（像其他操作系统一样）给每个文件标附上日期和时间标记。假设一个源文件上的标记是“6 月 6 日，星期五，上午 11：24”，而对应的编译文件的标记是“6 月 6 日，星期五，上午 11：40”。那么很明显在编译器生成编译文件之后没有修改源文件。另一方面，如果源文件上的日期和时间标记比编译文件上的晚，则 make 调用合适的编译器或汇编器生成与当前源文件版本对应的编译文件版本。

接下来，将可执行载入映像上的日期和时间标记与那个配置中的每个编译文件上的标记相比较。如果可执行载入映像比所有的编译文件创建得晚，则不必重新链接。但如果有一个编译文件的标记比可执行载入映像的晚，则该可执行载入映像没有体现那个编译文件的最新版。在这种情况下，make 调用链接器生成一个更新的载入映像。

换句话说，make 检验载入映像是否体现每个制品的当前版本。如果是，就不用再做什么，也不会在不需要的编译和链接上浪费 CPU 时间。但如果不是，则 make 调用相关的系统软件生成产品的最新版。

另外，make 简化了建造编译文件的任务。不需要用户每次指明需使用哪些制品以及如何链接它们，因为这个信息已经在 Makefile 中。所以，一个 make 命令就是建造一个有几百个制品的产品所需要的全部工具，它能确保完成的产品能正确地组合在一起。

像 make 这样的工具已经被加到不断变化的编程环境中，包括 Visual Java 和 Visual C++。make 的一个公开源码版本是 Ant（Apache 项目的一个产品）。

## 5.10　使用 CASE 技术提高生产力

Reifer（［Myers，1992］中报道过）进行了一项使用 CASE 技术提高生产力的调查，他从 10 个行业的 45 家公司收集数据。其中有半数的公司属于信息系统领域，25% 的公司属于科学领域，另外 25% 的公司属于实时的航天领域。平均每年提高生产力从 9%（实时的航天领域）到 12%（信息系统领域）不等。如果只考虑提高生产力，那么这些数字不能证明使用 CASE 技术的每个用户所花的 125 000 美元的成本是值得的。然而，被调查的公司感到使用 CASE 的好处不只体现在提高生产力，还体现在缩短开发时间以及提高软件质量。换句话说，尽管 CASE 技术的一些支持者宣称的有些夸张，但引入 CASE 环境确实提高了生产力。然而，还有其他同样重要的理由，说明应将 CASE 技术引入软件组织中，例如更快的开发、

更少的错误、更好的可用性、更容易的维护以及提高士气。

从 15 家财富 500 强公司的 100 多个开发项目中，CASE 技术有效性的较新结果反映出训练和软件过程的重要性［Guinan，Cooprider，and Sawyer，1997］。当使用 CASE 的小组接受通常的应用开发培训以及特定工具培训时，用户满意度增加了，开发计划安排也得以顺利实现。然而，当不提供培训时，软件交付会推迟并且用户也不太满意。还有，开发小组在同时使用 CASE 工具和结构化方法时，工作效率将提高 50%。这些结果支持 2.11 节中的主张，开发团队不应在成熟度等级 1 或 2 时使用 CASE 环境。说的难听点，一个用工具的傻瓜仍是个傻瓜［Guinan，Cooprider，and Sawyer，1997］。本章的最后一个图——图 5-11 是本章所讨论的理论工具和 CASE 工具的按字母排序表，同时给出各工具具体在哪一节介绍。

**理论工具**

成本－效益分析法（5.2 节）
度量（5.3 节）
逐步求精（5.1 节）

**CASE 分类**

环境（5.5 节）
低端 CASE 工具（5.5 节）
高端 CASE 工具（5.5 节）
工作平台（5.5 节）

**CASE 工具**

建造工具（5.9 节）
编程工具（5.6 节）
配置控制工具（5.8 节）
一致性检查器（5.5 节）
数据字典（5.5 节）
E-mail（5.6 节）
接口检查器（5.6 节）
在线文档（5.6 节）
操作系统前端（5.6 节）
灵巧打印机（5.6 节）
报表生成器（5.5 节）
屏幕生成器（5.5 节）
源代码级调试器（5.6 节）
电子数据表格（5.6 节）
结构化编辑器（5.6 节）
版本控制工具（5.7 节）
字处理器（5.6 节）
万维网浏览器（5.6 节）

图 5-11　本章讨论的理论（分析）工具和软件（CASE）工具概要以及所在的小节

## 本章回顾

本章首先讨论了一些理论工具，5.1 节描述并说明了基于米勒法则的逐步求精法，并在 5.1.1 节中通过例子对它进行了解释。另一个分析工具——成本－效益分析法在 5.2 节中介绍，软件度量在 5.3 节中介绍。

5.4 节定义了计算机辅助软件工程（CASE）工具，在 5.5 节和 5.6 节分别介绍了 CASE 工具的分类和范围。接下来描述了各种 CASE 工具。构建大型产品时，版本控制、配置控制和建造工具是最基本的，这些在 5.7 节到 5.9 节中有所描述。提高生产力作为 CASE 技术的一个结果，在 5.10 节中描述。

## 进一步阅读

要进一步了解米勒法则和他的有关大脑如何按块工作的理论，读者应该参考［Tracz，1979，and Moran，1981］和 Miller 的最初论文［Miller，1956］。

Wirth 关于逐步求精的论文［Wirth，1971］是经典之作，值得仔细研读。从逐步求精的观点看，同样精彩的书籍有 Dijkstra 的［Dijkstra，1976］和 Wirth 的［Wirth，1975］。实时系统的逐步设计在［Kurki-Suonio，1993］中有描述。

在［Kitchenham，Pickard，and Pfleeger，1995］中有对工具评估的实例研究。CASE 在软件工业中的适用程度在［Sharma and Rai，2000］中有介绍。

本书中，软件过程的每个工作流所用的 CASE 工具在讨论各工作流的章节中描述，有关工作平台或 CASE 环境的信息可参考第 14 章的“进一步阅读”。

[Whitgift, 1991] 很好地介绍了配置管理。较新的文章包括 [van der Hoek, Carzaniga, Heimbigner, and Wolf, 2002; Mens, 2002; and Walrad and Strom, 2002]。《International Workshops on Software Configuration Management》学报是一个有用的信息源。

在成本－效益分析方面有许多优秀的书籍，其中包括 [Gramlich, 1997]。

软件度量方面的重要书籍包括 [Shepperd, 1996, and Fenton and Pfleeger, 1997]，Jones [1994] 强调不可用和无效度量，但却在文献中接连被提及。在 [El Emam, Benlarbi, Goel, and Rai, 2001] 中讨论了面向对象度量的有效性。《IEEE Software》杂志的 1997 年 3 月/4 月刊中有一些关于度量的论文，包括 [Pfleeger, Jeffrey, Curtis, and Kitchenham, 1997]，它涉及软件度量的评估。Kilpi [2001] 描述了 Nokia 是如何实现一个度量程序的。在 [Sedigh-Ali and Paul, 2001] 中给出了用于基于 COTS 的系统的度量。

来自第 7 次国际软件度量论坛的大量有关度量的文章出现在《IEEE Transactions on Software Engineering》杂志的 2001 年 11 月刊中，其中特别令人感兴趣的是 [Briand and Wüst, 2001]。

## 习题

5.1　考虑将“前瞻”(lookahead) 引入到顺序主文件更新问题纠正的第三次求精设计中的效果，也就是说，在处理一个事务之前，必须读取下一个事务。如果两个事务应用于同一个主文件记录，那么根据下一个事务的类型决定对当前事务的处理。画一个 3×3 的表，行表示当前事务的类型，列表示下一个事务的类型，填入每种情况下所采取的动作。例如，对一个相同的记录进行两次连续插入明显是一个错误，但进行两次修改却很正常。例如，一个订户在某月里可以多次改变地址。现在画一个加入前瞻的第三次求精的流程图。

5.2　检查你对习题 5.1 的回答是否能正确地处理一个修改事务，紧接着处理一个删除事务，这两个事务应用于同一个主文件记录。如果不能，请修改你的答案。

5.3　检查你对习题 5.1 的回答是否也能先后、正确地处理一个插入、一个修改和一个删除事务，这三个事务均应用于同一个主文件记录。如果不能，请修改你的答案。

5.4　检查你的回答是否也能正确地处理 $n$ 次插入、修改和删除事务，$n>2$，所有事务均应用于同一个主文件记录。如果不能，请修改你的答案。

5.5　最后一个事务记录没有后继者，检查你对习题 5.1 的流程图是否能考虑到这一点，并正确地处理最后一个事务记录。如果不能，请修改你的答案。

5.6　在一些应用中，取代前瞻的一种做法是仔细地对事务进行排序。例如，原来由于修改后删除同一个主文件记录引起的问题可通过在修改前处理删除事务得到解决，这会造成主文件正确地写入了，而在异常报告中出现一个错误信息。调查一下是否存在一个事务的顺序，它可以解决习题 5.2、5.3 和 5.4 中列出的所有困难。

5.7　一种新型的胃肠疾病正在侵袭 Concordia 地区。像网状内皮细胞真菌病一样，它随空气传播。尽管这种病不致命，但打击还是很严重，受害者大约 2 周不能工作。Concordia 地方当局希望确定，如果试图根除该疾病需要花费多少钱。负责给公共卫生部提出建议的委员会正考虑该问题的四个方面：健康护理成本（Concordia 对所有居民提供免费的健康护理）、收入的损失（因而也损失了税收）、痛苦和不适、以及对政府的态度。请说明成本－效益分析法如何能帮助委员会，对于每项收益或成本，建议如何得到该

收益或成本的美元估计。

5.8　一个人完成软件产品需要版本控制工具吗？如果需要，为什么？

5.9　一个人完成软件产品需要配置控制工具吗？如果需要，为什么？

5.10　你是负责控制小型潜艇导航系统的软件的经理，需要修复三个用户报告的不同的错误，你把它们分配给 Paul、Quentin 和 Rachel。一天后你了解到，为实现这三个修复，必须修改同样的四个制品。然而你的配置控制工具不起作用，你不得不自己管理这些修改，你将如何做？

5.11　（学期项目）开发附录 A 中描述的 Ophelia's Oasis 产品，最适合使用什么类型的 CASE 工具？

5.12　（软件工程读物）你的导师将分发一些［Wirth，1971］的复印件，列出 Wirth 的方法与本章讲到的逐步求精法有什么区别。

## 参考文献


[Babich, 1986] W. A. BABICH, *Software Configuration Management: Coordination for Team Productivity,* Addison Wesley, Reading, MA, 1986.

[Briand and Wüst, 2001] L. C. BRIAND AND J. WÜST, "Modeling Development Effort in Object-Oriented Systems Using Design Properties," *IEEE Transactions on Software Engineering* **27** (November 2001), pp. 963–86.

[DeRemer and Kron, 1976] F. DEREMER AND H. H. KRON, "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Transactions on Software Engineering* **SE-2** (June 1976), pp. 80–86.

[Dijkstra, 1976] E. W. DIJKSTRA, *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dooley and Schach, 1985] J. W. M. DOOLEY AND S. R. SCHACH, "FLOW: A Software Development Environment Using Diagrams," *Journal of Systems and Software* **5** (August 1985), pp. 203–19.

[El Emam, Benlarbi, Goel, and Rai, 2001] K. EL EMAM, S. BENLARBI, N. GOEL, AND S. N. RAI, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering* **27** (July 2001), pp. 630–50.

[Feldman, 1979] S. I. FELDMAN, "Make—A Program for Maintaining Computer Programs," *Software—Practice and Experience* **9** (April 1979), pp. 225–65.

[Fenton and Pfleeger, 1997] N. E. FENTON AND S. L. PFLEEGER, *Software Metrics: A Rigorous and Practical Approach,* 2nd ed., IEEE Computer Society, Los Alamitos, CA, 1997.

[Fuggetta, 1993] A. FUGGETTA, "A Classification of CASE Technology," *IEEE Computer* **26** (December 1993), pp. 25–38.

[Gramlich, 1997] E. M. GRAMLICH, *A Guide to Benefit–Cost Analysis,* 2nd ed., Waveland Books, Prospect Heights, IL, 1997.

[Guinan, Cooprider, and Sawyer, 1997] P. J. GUINAN, J. G. COOPRIDER, AND S. SAWYER, "The Effective Use of Automated Application Development Tools," *IBM Systems Journal* **36** (No. 1, 1997), pp. 124–39.

[Jones, 1994] C. JONES, "Software Metrics: Good, Bad, and Missing," *IEEE Computer* **27** (September 1994), pp. 98–100.

[Kilpi, 2001] T. KILPI, "Implementing a Software Metrics Program at Nokia," *IEEE Software* **18** (November/December 2001), pp. 72–76.

[Kitchenham, Pickard, and Pfleeger, 1995] B. KITCHENHAM, L. PICKARD, AND S. L. PFLEEGER. "Case Studies for Method and Tool Evaluation," *IEEE Software* **12** (July 1995), pp. 52–62.

[Kurki-Suonio, 1993] R. KURKI-SUONIO, "Stepwise Design of Real-Time Systems," *IEEE Transactions on Software Engineering* **19** (January 1993), pp. 56–69.

[Loukides and Oram, 1997] M. K. LOUKIDES AND A. ORAM, *Programming with GNU Software,* O'Reilly and Associates, Sebastopol, CA, 1997.

[Mens, 2002] T. MENS, "A State-of-the-Art Survey on Software Merging," *IEEE Transactions on Software Engineering* **28** (May 2002), pp. 449–62.

[Miller, 1956] G. A. MILLER, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review* **63** (March 1956), pp. 81–97; reprinted at www.well.com/user/smalin/miller.html.

[Moran, 1981] T. P. MORAN (Editor), Special Issue: The Psychology of Human-Computer Interaction, *ACM Computing Surveys* **13** (March 1981).

[Myers, 1992] W. MYERS, "Good Software Practices Pay off—or Do They?" *IEEE Software* **9** (March 1992), pp. 96–97.

[Pfleeger, Jeffrey, Curtis, and Kitchenham, 1997] S. L. PFLEEGER, R. JEFFREY, B. CURTIS, AND B. KITCHENHAM, "Status Report on Software Measurement," *IEEE Software* **14** (March/April 1997), pp. 33–44.

[Rochkind, 1975] M. J. ROCHKIND, "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1** (October 1975), pp. 255–65.

[Sedigh-Ali and Paul, 2001] S. SEDIGH-ALI AND R. A. PAUL, "Software Engineering Metrics for COTS-Based Systems," *IEEE Computer* **34** (May 2001), pp. 44–50.

[Sharma and Rai, 2000] S. SHARMA AND A. RAI, "CASE Deployment in IS Organizations," *Communications of the ACM* **43** (January 2000), pp. 80–88.

[Shepperd, 1996] M. SHEPPERD, *Foundations of Software Measurement,* Prentice Hall, Upper Saddle River, NJ, 1996.

[Sobell, 1995] M. G. SOBELL, *A Practical Guide to the UNIX System,* 3rd ed., Benjamin/Cummings, Menlo Park, CA, 1995.

[Tichy, 1985] W. F. TICHY, "RCS—A System for Version Control," *Software—Practice and Experience* **15** (July 1985), pp. 637–54.

[Tomer and Schach, 2002] A. TOMER AND S. R. SCHACH, "A Three-Dimensional Model for System Design Evolution," *Systems Engineering* **5** (No. 4, 2002), pp. 264–73.

[Tracz, 1979] W. J. TRACZ, "Computer Programming and the Human Thought Process," *Software—Practice and Experience* **9** (February 1979), pp. 127–37.

[van der Hoek, Carzaniga, Heimbigner, and Wolf, 2002] A. VAN DER HOEK, A. CARZANIGA, D. HEIMBIGNER, AND A. L. WOLF, "A Testbed for Configuration Management Policy Programming," *IEEE Transactions on Software Engineering* **28** (January 2002), pp. 79–99.

[Walrad and Strom, 2002] C. WALRAD AND D. STROM, "The Importance of Branching Models in SCM," *IEEE Computer* **35** (September 2002), pp. 31–38.

[Whitgift, 1991] D. WHITGIFT, *Methods and Tools for Software Configuration Management,* John Wiley and Sons, New York, 1991.

[Wirth, 1971] N. WIRTH, "Program Development by Stepwise Refinement," *Communications of the ACM* **14** (April 1971), pp. 221–27.

[Wirth, 1975] N. WIRTH, *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1975.

# 第6章 测 试

**学习目标**

学完本章之后，你应当能够：

- 描述质量保证问题；
- 描述如何对制品进行基于非执行的测试（审查）；
- 描述基于执行的测试的原则；
- 解释需要测试什么。

传统软件生命周期模型在集成之后、交付后维护之前都常常包含一个单独的测试阶段。从获得高质量的软件角度讲，这个阶段是最关键的。测试是软件过程中一个完整的组成部分，是软件生命周期从始至终必须进行的行为。在需求流期间，需要检查需求；在分析流期间，需要检查规格说明；软件产品管理计划也必须经过类似的详细审查。设计流要求在每一步仔细检查，在实现流当然需要测试每个代码制品，并且产品作为一个整体在集成时需要进行测试。通过了验收测试后，安装产品，交付后维护就开始了，联合维护产品变成对产品修改版的反复检查。

换句话说，只在一个工作流结束时才测试该工作流的产品是远远不够的。例如对于设计流，设计小组的成员必须在开发它时不断检查设计。若小组在开发完成全部设计制品几星期或几个月后才发现过程早期犯的错误，则必须重新设计几乎整个制品。所以，开发小组需要在每个工作流进行的时候不断地测试，而不仅仅是在每个工作流结束时进行较为系统的测试。

1.7节中介绍了术语验证（verification）和确认（validation）。**验证**指的是确定某个工作流是否正确完成的过程，这在每个工作流结束时进行。另一方面，**确认**是在产品交付用户之前进行的深入细致的评定，它的目的是确定整个产品是否满足规格说明。尽管这两个术语在IEEE软件工程术语表［IEEE 610.12，1990］中是这样定义的，而且尽管术语V&V普遍用于表示测试，但本书尽可能少用验证和确认。一个原因就像6.5节中解释的，验证一词在测试的范围里有另一个含义。另一个原因是验证和确认（或V&V）这一词组意味着检查某一工作流的过程可以等到该阶段结束时。相反，重要的是这个检查应该与所有的开发和维护行为并行进行，所以，为避免曲解V&V词组的含意，我们使用**测试**（testing）一词。我们使用测试一词的第二个重要原因是：它是统一过程的术语。例如，第5个核心工作流是**测试流**。

基本上，有两种类型的测试：基于执行的测试和基于非执行的测试。例如，不可能执行一个书面的规格说明文档，只能尽可能地仔细审查它或对它进行某种形式的分析。然而，一旦有可执行的代码，就可以运行测试用例，也就是进行基于执行的测试。不过代码的存在不能排除做非执行测试的可能性，因为就像下面要解释的，仔细审查代码将会发现和运行测试用例时一样多的错误。本章将介绍执行测试和非执行测试的原则，第10～16章将应用这些原则，这些章节将描述过程模型的每个工作流和适用于这些工作流的专门的测试实践。在1.1节之前的“如果你想知道”部分中描述的前两个错误导致了致命的结果，所幸大多数情

况下，交付带有残留错误的软件都不会引起灾难性的结果，不过强调测试的重要性并不为过。

## 6.1 质量问题

我们在本节一开始先扩展 1.11 节与测试有关的定义。**差错**（fault）是当一个人犯了**过错**（mistake）时加到软件中的［IEEE 610.12，1990］。软件专业人员犯的一个过错可能会造成几个差错，反过来，几个过错可能会导致同一个差错。**故障**（failure）是观察到的软件产品的不正确行为，它是差错的结果［IEEE 610.12，1990］；**错误**（error）是不正确的结果的累积［IEEE 610.12，1990］。某个故障可能是由几个差错引起的，而有些差错可能永远不会引起故障。**缺陷**（defect）是一个通用词汇，它泛指差错、故障或错误。

现在我们转到质量问题。在软件范围内使用质量一词时经常会引起误解，毕竟质量意味着某种优秀，但这恰巧不是软件工程师想要的意思。生硬点说，软件开发的艺术只是让软件运行正确即可——优秀远远超过正常状态许多数量级，它对于在 CMM 级别 1 的软件组织而言是可望不可及的（见 3.13 节）。

软件的**质量**是产品满足它的规格说明的程度（参见下面的“如果你想知道”部分）。然而，这还不够。例如，为了确保产品易于维护，该产品必须仔细设计并编码。因此，软件具有较高质量是必须的，但是这还不够。

**如果你想知道**

使用“质量”一词表示“符合规格说明”（与“优秀”或“精美”相对），是工程和加工领域中的实际情况。例如，考虑可口可乐瓶生产厂的质量控制管理员，他的工作就是确保每一个离开生产线的瓶子或罐在任何方面都能满足可口可乐的规格说明，不需要生产什么“优秀”的可口可乐或“精美”的可口可乐。基本目标就是确认每个瓶或罐严格地符合公司充气饮料的规则（规格说明）。

“质量”一词同样适用于汽车工业。“质量第一”是福特汽车公司从前的口号，换句话说，福特的目标是确保每辆从福特生产线上下来的汽车严格符合该车的规格说明。按通常的软件工程说法，该汽车必须在任何方面都是“免调试”的。

每个软件专业人员的任务是随时保证高质量的软件，就是说，每个开发者和维护者个人应对于检查他（或她）的工作是正确的负责。质量不是由 SQA 小组后来加上的东西，它必须从一开始由开发者建立起来。SQA 小组的一个作用是确保开发者确实做出高质量的工作。SQA 小组还有另外的职责，下面小节中将对此进行说明。

### 6.1.1 软件质量保证

如前所述，SQA 小组作用的一个方面是确保产品是正确的，更简单地说，一旦开发者完成了一个工作流并仔细地检查了他们的工作，SQA 小组的成员就需要检验那个工作流正确地完成了。还有，当产品完成并且开发人员认为该产品整体上是正确时，SQA 小组必须确保产品是这样的。然而，软件质量保证不只在工作流结束或开发过程结束时测试（或 V&V），SQA 应当应用于软件过程本身。例如，SQA 小组的职责包括开发各种软件必须遵循的标准，以及建立确保符合这些标准的监督过程。简单地说，SQA 小组的原则是确保软

件过程的质量，从而确保软件产品的质量。

### 6.1.2 管理独立

在开发小组和SQA小组之间保持**管理独立**很重要，也就是说，开发处于一个管理者领导之下，SAQ处于另一个不同的管理者领导之下，哪一个管理者也不能替代另一个。这样做的原因是，常常在交付期限快到时发现严重的错误。软件组织现在必须在两个不完满的选项中进行选择。要么产品按时交付，但充满错误，让客户自己去应付满是错误的软件；要么开发者修复软件但推迟交付。不管怎样，客户都可能对该软件组织失去信心。负责开发的管理者不应做出按时交付有错误的软件这个属于开发问题的决定，SQA管理者也不能做出进行进一步测试并推迟交付软件的决定。两个管理者应该把情况报告给更高级的管理者，他可以决定两个选择中的哪一个对软件开发组织和客户都最好。

乍一看，有一个单独的SQA小组看起来可能会增加软件开发的成本，但事实并不是这样。额外的成本与它所带来的收益，也就是高质量的软件相比是相当小的。没有SQA小组，软件开发组织的每个成员将不得不在某种程度上涉及质量保证方面的事情。假设一个组织有100个软件专业人员，每人大约需要花费30%的时间处理质量保证方面的事情。而如果将100个人分成两组，其中70人进行软件开发，另外30人负责SAQ。相同数量的时间用于SQA，惟一增加的费用是需要一个管理者领导SQA小组。质量保证现在可以由一个独立的专家小组完成，这将使得产品质量比SQA行为由全体组织成员完成时更高。

在软件公司非常小（4个雇员或更少）的情况下，建立一个单独的SQA小组可能不那么经济。在这种情况下，最好是确保分析制品由不负责生产这些制品的某个人进行检查，对于设计制品、编码制品等也要类似处理。下一节解释这样做的原因。

## 6.2 非执行测试

测试软件而不运行测试用例称为**基于非执行的测试**（简称非执行测试）。非执行测试方法的例子包括评审（review）软件（仔细阅读它）以及用数学方法分析软件（见6.5节）。

负责撰写文档的人成为对文档进行评审的惟一的人不是一个好主意。几乎每个人都有盲点，这使得差错在文档中蔓延，而那些相同的盲点使错误在文档评审时不能被发现。所以，评审的任务必须交给其他人，而不是文档的原始作者。另外，只有一个评审者是不合适的，我们都曾有过这样的经历：多次阅读一篇文档却不能发现一个明显的拼写错误，而第二个阅读者可能立即就能发现它。这是像走查或审查这样的评审技术的基本原则之一。在这两种类型的评审中，一个文档（例如一个规格说明文档或设计文档）由一组技能全面的软件专业人员进行仔细地检查。由专家小组评审的好处是参加者不同的技能增加了找到错误的机会。另外，一组有经验的人在一起工作通常会产生相互促进的效果。

走查和审查是两类评审，两者之间的基本不同是走查比审查的步骤少且不那么正式。

### 6.2.1 走查

一个走查（walkthrough）小组应包含4～6个人。一个分析走查小组至少应包含一个负责撰写规格说明的小组的代表、负责分析流的管理者、一个客户代表、一个即将进行下一个开发工作流的小组（在本例中是设计小组）的代表和一个软件质量保证小组的代表。由于下

一节将要说明的一些原因，SQA 小组成员应该主持走查。

走查小组的成员应尽可能是有经验的高级技术人员，因为他们可能会查找到重要的差错，也就是说，他们要查到可能对项目有主要负面影响的差错［R. New，personal communication，1992］。

走查用的材料应提前分发给各参加者，允许他们充分地准备。每个评审者应研究该材料并写出两个清单：一个是评审者不明白的事项清单，另一个是评审者认为不正确的事项清单。

### 6.2.2 管理走查

走查应由 SQA 代表主持，因为如果走查完成得不好，频繁地漏掉差错，那么 SQA 代表损失最大。相反，负责分析流的代表可能会希望规格说明文档尽快得到批准以开始其他的任务；客户代表则会认为评审中未发现的错误会在验收测试中显现出来并得到修复，因而对客户组织来说不会有额外的花销；但对 SQA 代表来说最为意义重大：产品的质量是 SQA 小组的专业能力的直接体现。

走查的主持人引导走查小组的其他成员走查文档以发现差错。改正差错不是小组的任务，只需记录它们以备以后修改，这主要有 4 个原因：

1）在走查的时间限制内由委员会（也就是走查小组）进行修改，在质量上可能不如由受过必要技术训练的个人进行修改。

2）由 5 个人组成的走查小组进行修改需要的时间与一个人进行修改需要的时间相当，因而，考虑这 5 个人的报酬时，将花费 5 倍的成本。

3）并不是所有标为错误的事项一定不正确。依照一句格言，“如果它没断，就不要修复它”，最好仔细分析错误，然后只在它确实是一个问题时才修正它，而不是按照小组的意愿去“修复”一个完全正确的东西。

4）走查时没有足够的时间去检测和纠正错误。一般走查不超过 2 个小时，时间应用于检测并记录错误，而不是去纠正它们。

有两种方式来实施走查，第一个是参加者驱动。参加者拿出不清楚的事项和他们认为不对的事项清单，分析小组的代表必须回答每一个询问，说清楚评审者不清楚的地方，或者承认确实有错误，或者解释评审者为何错了。

实施走查的第二个方式是文档驱动。一个负责该文档的人（或者是个人，或者是作为小组的一部分）带领参加者阅读文档，评审者就他们事先准备的意见或现场引发的意见，随时打断并提问。这种方法看起来更彻底，另外，它通常导致发现更多的错误，因为文档驱动走查中的大多数错误是由介绍者自发地发现的。不只一次地，介绍者在叙述中间停下来，他或她的脸会红起来，那个多次阅读文档也没有发现的潜伏着的错误，突然变得很明显。心理学家研究的一个卓有成果的领域就是，确定为什么在各种走查期间言语表达常常导致错误的检出，各种走查包括分析走查、设计走查、规划走查和代码走查。毫不奇怪，更彻底的文档驱动评审是 IEEE 软件评审和审计标准［IEEE 1028，1997］中规定的技术。

走查主持人的主要作用是引出问题并促进讨论。走查是一个交互的过程，不是介绍者的一面之词。走查不能用作评估参加者的一种方法，这也是很重要的。如果是这样，走查将退

化成一个打分会议，无论这个会议的主持人如何运作，也不能检测到错误。负责被评审文档的管理者应该是走查小组的成员，这在前面已经建议过。如果这个管理者也负责走查小组成员（特别是介绍者）的年度评估，小组检测到错误的能力将大打折扣，因为介绍者的原始动机是将暴露的错误减到最少。为避免这种利益上的矛盾，负责某个工作流的人不应是直接负责评估那个工作流走查小组的成员。

### 6.2.3 审查

审查（inspection）最初是由 Fagan 为测试设计和代码而提出的［Fagan，1976］。审查远比走查更深入，它有 5 个正式的步骤：

1）由一个负责生成文档的人提供被审查的文档（需求、规格说明、设计、代码或规划）的**概要**。在概要部分结束时，将文档分发给参加者。

2）处在**准备**中。参加者设法详细理解文档。在最近的审查中发现的错误类型（按出现频度排列）的列表是最好的帮助。这些列表有助于小组成员集中精力在错误发生最多的区域。

3）开始审查。开始时一个参加者与审查小组一起浏览文档，确保覆盖每个事项，而且每个分支都至少经过一次。然后开始查找错误。像走查一样，目的是发现和证实错误，而不是修改它们。在一天里，审查小组的领导（**主持者**）必须写出一个审查报告以确保审查小心细致地完成。

4）处于**修订**中。负责该文档的个人改正审查报告中列出的所有错误和问题。

5）处于**跟踪**状态。主持者必须确认提出的每个事项都得到满意的解决，或者修改文档，或者澄清被误当成错误的事项。所有的修改都必须经过检查，以确保不会产生新的错误［Fagan，1986］。如果送审材料的 5%需要修订，那么必须重新召集审查小组进行100%的重新审查。

审查应当由四人小组领导。例如，在设计审查的情况下，审查小组应包含主持者、设计者、实现者和测试者。主持者是审查小组的管理者和领导，必须有一个负责当前阶段的小组的代表，还应有一个负责下一阶段的小组的代表。设计者是生成设计的小组的成员，而实现者是负责（作为个人或作为小组的一部分）将设计转化为代码的小组成员。Fagan 建议测试者是负责建立测试用例的任意一个程序员，当然最好该测试者是 SQA 小组的成员。IEEE 标准建议 3～6 个人参加审查小组［IEEE 1028，1997］。“主持者”担任特别的角色，既是**朗读者**带着小组浏览设计，又是**记录员**负责生成检测到的错误的书面报告。

审查的一个基本组成部分是潜在错误的一览表。例如，一个设计审查的一览表应包括如下事项：是否充分并正确地解决规格说明文档的每个问题？对于每一个接口，实参和形参是否对应？错误处理机制是否已完全确定？该设计与硬件资源兼容吗？或者是否对硬件的要求比实际可得到的多？该设计是否与软件资源兼容？例如，设计制品中规定的操作系统是否具有设计所要求的功能？

审查的一个重要组成部分是错误统计记录。错误必须按严重程度（重要的或次要的，重要错误的例子是引起过早结束或破坏数据库的错误）和错误类型记录。在设计审查的情况下，典型的错误类型包括接口错误和逻辑错误。这些信息可以用在许多方面：

- 一个给定产品中的错误数可与可比产品的同一开发阶段检测到的平均错误数进行比

较，它能向管理者提前发出警告，提醒有些事情出了问题，允许及时采取纠正措施。

- 如果审查两个或三个代码制品，发现一种特定类型的错误数量不成比例，那么应该开始检查其他代码制品并采取纠正措施。
- 如果在一个特定代码制品的审查中发现了比产品中其他代码制品远远多的错误，那么通常应从头重新设计该制品并实现新的设计。
- 在设计制品审查中检测到的错误数量和错误类型信息将有助于小组在下一个阶段对该制品的实现进行代码审查。

Fagan［1976］的第一个试验是在一个系统软件产品上进行的。审查投入了100人时，进度是四人小组每天进行2个小时的审查。在产品开发过程中发现的所有错误中，67%是由单元测试开始之前的审查找到的。进一步地，产品安装后的头7个月里，在审查过的产品中检测到的错误比使用非正式的走查进行评审的可比产品少38%。

Fagan［1976］在另一个应用软件产品上做了试验，发现所有检测到的错误的82%是在设计和代码审查中发现的。审查的一个有用的副作用是程序员的生产力提高了，因为在单元测试上花费了更少的时间。使用一个自动评估模型，Fagan确定，不算花费在审查上的时间，审查过程的结果是使程序员资源节省了25%。在另一个不同的试验中，Jones发现超过70%的错误可通过设计和代码审查检测出来［Jones，1978］。

后来的研究结果同样给人深刻印象。在一个6000行的商业数据处理应用程序中，93%的错误是在审查中发现的［Fagan，1986］。［Ackerman，Buchwald，and Lewski，1989］中还报道了在一个操作系统产品的开发过程中，审查（而不是测试）的使用使检测错误的成本减少了85%，在一个交换系统产品中减少了90%［Fowler，1986］。在美国喷气推进实验室（Jet Propulsion Laboratory，JPL），平均每2小时的审查暴露4个主要错误和14个次要错误［Bush，1990］。转化成美元计算，这意味着“每次审查”大约节省25 000美元。另一个JPL研究［Kelly，Sherif，and Hops，1992］显示，检测到的错误数随传统上阶段的进展呈指数递减。换句话说，在审查的帮助下，可在软件过程中及早发现错误。早发现错误的重要性反映在图1-5中。

代码审查较之运行测试用例（基于执行的测试）的一个优势是测试者不需要解决故障。经常发生的是，当一个产品在接受测试时，它出现故障了。造成故障的差错现在必须定位并修复，这样，基于执行的测试才能继续。相反，在非执行测试期间发现的代码中的错误可以记录下来，并继续向下进行审查。

审查过程的一个风险是，与走查一样，它可能被用于评估能力表现。在审查的情况下这种危险就更突出，因为可以得到详细的错误信息。Fagan消除了这种恐惧，他指出，经过3年多的时间，他了解到没有一个IBM管理者使用这样的信息来评估程序员，或者如他所提出的，没有一个管理者试图“杀死一只能下金蛋的鹅”［Fagan，1976］。然而，如果没有正确地实施审查，它们将不会像在IBM那样取得广泛的成功。除非高层管理者认识到这潜在的问题，否则错误使用审查信息是非常可能的。

### 6.2.4　审查与走查的对比

表面上看，审查与走查之间的区别是审查小组使用询问一览表来帮助找到错误。但区别远不止如此，走查的过程有两步：准备、随后小组对文档进行分析。而审查过程有五步：概

要、准备、审查、修订和跟踪，而且在这些步骤中，每一个接下来的过程都是形式化的。这种形式化的例子是：仔细对错误归类，并在后续工作流文档的审查以及将来产品的审查中利用该信息。

审查过程比走查花更多的时间，值得在审查上花费额外的时间和努力吗？前一节的数据清楚地表明审查是检测错误的一种强有力的划算的工具。

### 6.2.5 评审的优缺点

评审（走查或审查）有两个主要的优点。首先，评审是检测错误的一个有效途径；其次，在软件过程的早期发现错误，也就是在它们的修复变得昂贵前发现它们。例如，在实现开始之前发现设计错误，以及在制品集成到产品中之前发现编程错误。

然而，如果软件过程不合适，则评审的有效性会减小。首先，大型软件相当难于评审，除非它包含更小的很大程度上独立的组件。面向对象的范型的优点之一是，如果正确地实现，生成的产品真正包含很大程度上独立的小块；其次，设计评审小组有时需要参考分析制品，代码评审小组经常需要查看设计文档。除非前一工作流的文档是完整的、更新过的，能够反映项目的当前版本，而且在线可用，否则会严重地妨碍评审小组发挥作用。

### 6.2.6 审查的度量

为确定审查的效果，可以使用一些不同的度量。第一个度量是**审查速率**。当审查规格说明和设计文档时，可以测量每小时检查的页数。对于代码审查，一个合适的度量是每小时检查的代码行数；第二个度量是**错误密度**，它用每页检查的错误数或每千行代码（1000 lines of code，KLOC）检查的错误数来计算。这些度量可分解为每单元材料的主要错误数和每单元材料的最小错误数；另一个有用的度量是**错误检测率**，也就是每小时检测到的主要和最小错误数。第四个度量是**错误检测效率**，也就是每人时检测到的主要和最小错误数。

尽管这些度量的目的是测量审查过程的效果，结果却可能反映出开发小组的不足。例如，如果错误检测率从每千行代码 20 个错误突然升到 30 个，这不一定意味着审查小组突然效率提高 50%。另一种解释是代码质量下降了，仅仅是检测出更多的错误而已。

讨论完非执行测试后，下面将讨论执行测试。

## 6.3 执行测试

已经讲过，测试证实了差错（bug）的不存在。尽管一些组织将软件预算的 50% 花费在测试上，交付的“测试过的”软件还是不可靠。

产生这个矛盾的原因很简单，就像 Dijkstra 形容的，“程序测试可以是显示 bug 存在的非常有效的方式，但显示它们的不存在却是绝对不充分的”[Dijkstra，1972]。Dijkstra 要说明的是，如果产品使用测试数据运行而输出是错误的，那么该产品肯定有错误。但如果输出是正确的，那么产品仍可能含有错误，能从这个特定的测试中得到的惟一信息是，产品在特定的这组测试数据上运行正确。

## 6.4 应该测试什么

要描述应该测试什么特性，首先有必要给执行测试一个精确的定义。根据 Goodenough

的描述，**执行测试**是一个推断某一产品的特定行为特性的过程，它基于或部分基于在一个已知环境下用经过选择的输入执行产品得到的结果［Goodenough，1979］。这个定义有三个令人困扰的含义：

1）该定义说明了测试是一个推断的过程。测试者拿到产品，用已知的输入数据运行它并检查输出。测试者推断产品如果有错，错在哪里。从这个角度看，测试无异于试图在一个黑房间里寻找一只黑猫，但首先却不知道猫是不是在房间里。测试者几乎没有线索来帮助找到错误：也许 10 或 20 组输入和对应的输出，可能是一份用户错误报告和数千行代码。测试者不得不由此推断是否有错误，如果有，是什么错误。

2）该定义产生的一个问题来自“在一个已知环境”这个词组。我们永远不会真正知道我们的环境，不管是硬件还是软件；我们不能确定操作系统在正确地运行或运行时例程是正确的；计算机主存里也许有间歇的硬件错误。所以观察到的产品的行为实际上可能是：正确的产品正与错误的编译器或错误的硬件或环境中一些其他的错误组件相互作用。

3）执行测试定义的另一个困扰人的地方是“用经过选择的输入”这个词组。在一个实时系统的情况下，对系统的输入施加控制常常是不可能的。考虑一个航空软件。飞行控制系统有两种类型的输入。第一类输入是飞行员要飞机做什么。这样，如果飞行员拉回操纵杆爬升或打开油门提高飞机速度，这些机械的动作被转化为数字信号送给飞行控制计算机；第二类输入是飞机当前的物理状态，例如它的高度、速度和机翼的仰角。飞行控制软件使用这样的量值来计算应给飞机的组件（例如机翼和发动机）传送什么信号，以实现飞行员的指示。尽管只需通过适当地设置飞机的控制器，就可以将飞行员的输入很容易地设置成任何想要的值，而对应飞机当前物理状态的输入却不容易被操纵。事实上，没有什么方法能够强迫飞机提供“经过选择的输入”。

那么如何测试一个实时系统？答案是使用一个仿真器。**仿真器**是产品（在本例中是飞行控制软件）运行环境的一个工作模型。可以通过让仿真器向飞行控制软件发送经选择的输入来测试飞行控制软件。仿真器具有控制装置，允许操作者将一个输入变量设置成任何选择的值。若测试的目的是确定如果一个发动机起火，飞行控制软件将怎样做，那么就设置仿真器的控制装置，使送到飞行控制软件的输入与实际发动机起火时的输入没有区别。通过检查从飞行控制软件发送到仿真器的输出信号，对输出进行分析。最好的仿真器可以是系统某些方面忠实的模型的很好近似，但它不可能是系统本身。使用仿真器意味着确实有一个“已知的环境”，但这个环境不可能在各个方面都与产品安装的实际环境相同。

前面的测试定义提到“行为特性”，那么必须测试什么行为特性？一个明显的答案是测试产品功能是否正确。但是，下面将看到，正确性既不是必要的，也不是充分的。讨论正确性之前，要考虑 4 个其他的行为特性：实用性、可靠性、健壮性和性能［Goodenough，1979］。

### 6.4.1　实用性

**实用性**（utility）是在规格说明允许的条件下使用正确的产品时，满足用户需求的程度。换句话说，正确运行的产品离不开用规格说明衡量是有效的输入。例如，用户可以测试产品如何易于使用，产品能否执行有用的功能，以及与竞争产品相比该产品的成本是否划算。不管产品是否正确，必须测试这些重要的事项。如果产品的成本不划算，那么就没有必要购

买。除非产品易于使用，否则用户根本不会去使用它或不能正确地使用它。所以，考虑购买一个已存在的产品时，首先需要测试产品的实用性，如果该产品在这项测试中失败，那么测试应该结束。

### 6.4.2 可靠性

必须测试产品的另一个方面是它的可靠性。**可靠性**（reliability）是对产品故障的出现频率和严重性进行的测量，我们还记得，故障是在允许的操作条件下，一个不可接受的结果或行为，它是由一个差错造成的。换句话说，有必要知道产品隔多久出现故障（**平均故障间隔时间**），以及该故障造成的影响有多严重。当一个产品出现故障时，一个重要的问题是平均修复故障需要多长时间（**平均修复时间**）。但是，常常更重要的是修复故障的结果用了多长时间，这常被忽视。假设运行于通信前端的软件平均，每6个月出现一次故障，但出现故障时，它彻底毁坏了数据库。数据库最多能重新恢复到最后一个测试点转储后的状态，可以使用审计跟踪使数据库处于实际上是最新的状态。但是，如果这个恢复过程需要2天的时间，在此期间数据库和通信前端不能工作，那么该产品的可靠性很低，尽管它的平均故障间隔时间为6个月。

### 6.4.3 健壮性

每个产品的另一个需要测试的方面是它的**健壮性**（robustness）。尽管它很难有一个准确的定义，但健壮性基本上是一些因素的函数，如运行条件的范围、有效输入带来不可接受的结果的可能性，以及产品的输入无效时结果的可接受性。一个运行条件很宽的产品比运行条件限制多的产品更具有健壮性。当输入满足规格说明时，一个健壮的产品不应产生不可接受的结果。例如，一个有效的命令不应造成灾难性的后果；当产品在不允许的运行条件下使用时，一个健壮的产品不应崩溃。为测试健壮性的这个方面，测试者故意输入不满足输入规格说明的测试数据，以确定产品的反应有多糟糕。例如，当产品要求输入一个名称时，测试者用一串不可接受的字符应答，比如“control-A escape-% ? $#@”。如果计算机以一条信息来响应，如“Incorrect data, Try again”，或者更好地，提示用户为何这些数据不符合要求，那么它比那些输入的数据不符合要求时就崩溃的产品要更健壮。

### 6.4.4 性能

**性能**是产品必须测试的另一个方面，例如，根据时间或空间要求，知道产品所受限制的程度很重要。对于一个嵌入式计算机系统，诸如防空导弹上携带的计算机，系统空间上的限制要求软件只能使用128 MB的主存。不论软件多么优秀，如果它需要256 MB的主存，那么它根本不能使用。（要进一步了解嵌入式软件，参见下面的“如果你想知道”部分。）

**如果你想知道**

嵌入式计算机是一个较大系统的一个完整的部分，它的主要目的不是计算。嵌入式软件的功能是控制计算机嵌入其中的设备。军事应用的例子包括战斗机上的航空控制计算机网络或洲际弹道导弹内部的计算机。位于导弹弹头位置的嵌入式计算机只控制导弹，它不能被导弹基地的战士用于（比如说）打印薪水册。

更常见的例子是电子表或洗衣机中的计算机芯片。同样，洗衣机里的芯片只可用来控制

洗衣机，洗衣机的主人不可能利用该芯片来结算支票本。

实时软件的特征是硬件时间限制严格，即如果不能满足该限制，就会丢失信息。例如，一个核反应控制系统需要采样内核的温度，并每 1/10 秒处理该数据。如果系统不够快，不能处理每 1/10 秒从温度传感器传过来的中断，那么将丢失数据，且没有办法来恢复该数据。系统下一次接收到的温度数据将会是当前温度，而不是丢失的那次读数。如果反应堆正处于溶化的关键点上，则如规格说明中所规定的，收到并处理所有相关的信息是至关重要的。对于所有的实时系统，其性能必须每次都满足规格说明中列出的时间限制。

### 6.4.5　正确性

最后，可以给出**正确性**的定义了。如果产品在允许的条件下运行，能够满足它的输出规格说明，并与它使用的计算资源无关，则该产品是正确的 [Goodenough，1979]。换句话说，如果提供了满足输入规格说明的输入，而且给产品提供它所需的所有资源，那么如果产品的输出满足输出规格说明，则它是正确的。

正确性的这个定义像测试的定义一样，有令人困惑的含义。假设已用广泛的测试数据成功测试了一个产品，这能意味着该产品是可接受的吗？遗憾的是，不能。如果一个产品是正确的，意味着产品满足规格说明。但如果规格说明本身就是不正确的呢？为了说明这个难题，考虑图 6-1 所示的规格说明。该规格说明标明输入是有 $n$ 个整数的数组 $p$，而输出是另一个数组 $q$，按非降顺序排列。表面上看，规格说明似乎完全正确，但考虑图 6-2 所示的方法 trickSort，在这个方法里，数组 $q$ 的所有 $n$ 个元素被置为 0，该方法满足图 6-1 所示的规格说明，所以是正确的。




图 6-1　不正确的排序规格说明

```
void trickSort (int p[ ], int q[ ])
{
    int i;
    for (i = 0; i < n; i++)
        q[i] = 0;
}
```

图 6-2　满足图 6-1 的规格说明的方法 trickSort

发生了什么呢？遗憾的是，图 6-1 所示的规格说明是错误的，它忽视了输出数组 $q$ 的各元素的状态是输入数组 $p$ 各元素的一个置换（重新排列）。排序的一个本质方面在于它是一个重新排列的过程。图 6-2 所示的方法利用了这个规格说明的错误，换句话说，trickSort 是正确的，但图 6-1 的规格说明是错误的。改正后的规格说明如图 6-3 所示。




图 6-3　改正后的排序规格说明

从这个例子可以清楚地看到，规格说明错误的结果是严重的，毕竟，如果规格说明是不正确的，产品的正确性就无从谈起。

产品是正确的这一事实并不是充分的，因为看起来正确的规格说明本身可能是错的。但它是必要的吗？考虑下面的例子：一个软件组织获得了一个新的极好的C++编译器，新的编译器每分钟编译的源代码行数是旧编译器的两倍，目标代码运行几乎快45%，而目标代码的规模减小了大约20%。另外，错误提示信息更明确，每年的维护和更新费用比旧编译器少了一半。然而存在一个问题：**for**语句第一次出现在任何类里时，编译器打印出一个假的错误信息，所以该编译器不是正确的，因为编译器的规格说明或隐含或明确地要求，当且仅当源代码中有错误时才打印出错误信息。当然可以使用这个编译器——事实上，除了这个问题它在各方面都是非常理想的，进而有理由期望这个次要的错误会在下一版中改正。与此同时，程序员认识到可以忽略这个假的错误信息，不但该组织可以使用这个不正确的编译器，而且如果有人建议替换回原来旧的但正确的编译器，必将遭到强烈抗议。所以产品的正确性既不是必要的，也不是充分的。

诚然，前面的两个例子有点人为因素，但它们却切中要点，即正确性只意味着产品是它的规格说明的一个正确实现。换句话说，除了显示产品是正确的之外，还有许多需要测试。

对于所有与执行测试有关的难点，计算机科学家已设法提出其他办法来确保产品按期望的那样运行。一个这样的非执行测试选择已受到40多年的广泛关注，它就是正确性证明。

## 6.5 测试与正确性证明

**正确性证明**是显示产品正确的一种数学技术，换句话说，显示产品满足规格说明。该技术有时被称为验证，然而就像前面指出的，验证一词通常用于表示所有的非执行测试技术，不止是正确性证明。为明确起见，将这个数学技术称为正确性证明，提醒读者它是一个数学证明过程。

### 6.5.1 正确性证明的例子

为了明白如何证明正确性，考虑图6-4所示的代码段。与代码等效的流程图如图6-5所示。我们现在来表明该代码段是正确的——执行该代码之后，变量s将包含数组y的n个元素的和。在图6-6中，在每个语句之前和之后，在标注上字母A到H的地方，放置了一个断言（assertion），也就是在每个拥有某个数学属性的地方做出一个声明。现在证明每个断言的正确性。

```
int k, s;
int y[n];
k = 0;
s = 0;
while (k < n)
{
    s = s + y[k];
    k = k + 1;
}
```

图6-4 要证明是正确的代码段

输入规格说明——代码执行前在$A$处具有的条件是，变量n是一个正整数，也就是：

$$A:\quad n\in\{1,2,3,\ldots\} \tag{6.1}$$

明显的输出规格说明是，如果控制到达$H$点，s的值包含存储在数组y中的n个值的和，也就是：

$$H:\quad s=y[0]+y[1]+\ldots+y[n-1] \tag{6.2}$$

事实上，对于较强的输出规格说明，可以证明该代码段是正确的，这个输出规格说明是：

$$H:\quad k=n \text{ 且 } s=y[0]+y[1]+\ldots+y[n-1] \tag{6.3}$$

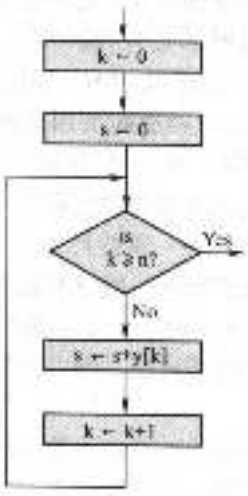



图 6-5　图 6-4 的流程图

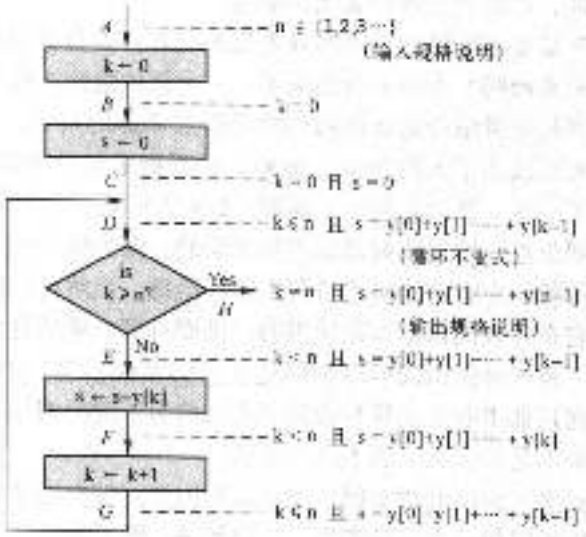



图 6-6　带有输入规格说明、输出规格说明、循环不变式和断言的图 6-5

你可能会询问：输出规格说明（6.3）从哪里得来？在本证明的最后，我们希望你能回答这个问题，也可参见习题 6.10 和 6.11。

除了输入和输出规格说明，本证明过程的第三个方面是提供一个循环不变式，也就是必须在 $D$ 点上提供一个数学表达式，不管该循环被执行了 0 次、1 次或许多次。要证明持有的循环不变式是：

$$D:\quad k\leqslant n \text{ 且 } s=y[0]+y[1]+\ldots+y[k-1] \tag{6.4}$$

下面将表明，如果在 $A$ 点持有输入规格说明（6.1），那么将在 $H$ 点持有输出规格说明（6.3），也就是证明该代码段是正确的。

首先，执行赋值语句 k←0，现在控制停留在 B 点，那里持有如下的断言：

$$B:\quad k=0 \tag{6.5}$$

为了更准确，在 $B$ 点断言应该读做 $k=0$ 且 $n\in\{1,2,3,\ldots\}$。然而，流程图中的所有点都持有输入规格说明（6.1）。为简便起见，以下“且 $n\in\{1,2,3,\ldots\}$”被省略。

在 $C$ 点，作为第二个赋值语句 s←0 的结果，下面的断言是真的：

$$C:\quad k=0 \text{ 且 } s=0 \tag{6.6}$$

现在进入了循环，这里将通过推导证明循环不变式（6.4）确实是正确的。在该循环第一次执行之前，有一个断言（6.6），也就是 $k=0$ 且 $s=0$。现在看循环不变式（6.4）。因为断言（6.6）可确定 $k=0$，而且从输入规格说明（6.1）可确定 $n\geqslant 1$，如要求的那样它遵从 $k\leqslant n$。进一步地说，因为 $k=0$，它遵从 $k-1=-1$，所以（6.4）中的和是空的，而且要求 $s=0$。循环不变式（6.4）因此在第一次进入循环之前是真的。

现在进行归纳假设的步骤。假定在该代码段执行期间的某些阶段，该循环不变式有效，也就是 k 等于一些值 $k_0$，$0\leqslant k_0\leqslant n$，执行到 $D$ 点，有下面的断言：

$$D:\quad k_0\leqslant n \text{ 且 } s=y[0]+y[1]+\ldots+y[k_0-1] \tag{6.7}$$

控制现在经过测试框。如果 $k_0 \geqslant n$，那么因为假设 $k_0 \leqslant n$，它遵从 $k_0 = n$。按照归纳假设 (6.7)，这意味着：

$$H: \quad k_0 = n \text{ 且 } s = y[0] + y[1] + \ldots + y[n-1] \tag{6.8}$$

这正好是输出规格说明 (6.3)。

另一方面，如果测试 $k_0 \geqslant n$?结果为否，那么控制从 $D$ 点转向 $E$ 点。因为 $k_0$ 不大于或等于 n，$k_0 < n$ 则 (6.7) 就变成：

$$E: \quad k_0 < n \text{ 且 } s = y[0] + y[1] + \ldots + y[k_0 - 1] \tag{6.9}$$

现在执行语句 $s \leftarrow s + y[k_0]$，因此由于有断言 (6.9)，在 $F$ 点一定有下列的断言：

$$\begin{aligned} F: \quad & k_0 < n \text{ 且 } s = y[0] + y[1] + \ldots + y[k_0 - 1] + y[k_0] \\ & = y[0] + y[1] + \ldots + y[k_0] \end{aligned} \tag{6.10}$$

下一个要执行的语句是 $k_0 \leftarrow k_0 + 1$，为了看到这个语句的结果，假定执行该语句前 $k_0$ 的值是 17，那么 (6.10) 中和的最后一项是 $y[17]$。现在 $k_0$ 的值增加 1 成为 18，和 s 没有变，所以和中最后一项仍是 $y[17]$，它现在是 $y[k_0 - 1]$。同样，在 $F$ 点，$k_0 < n$，$k_0$ 的值增加 1 意味着，如果在 $G$ 点有不等式，那么 $k_0 \leqslant n$。这样，$k_0$ 增加 1 的结果是在 $G$ 点有下面的断言：

$$G: \quad k_0 \leqslant n \text{ 且 } s = y[0] + y[1] + \ldots + y[k_0 - 1] \tag{6.11}$$

在 $G$ 点的断言 (6.11) 与在 $D$ 点假定的断言 (6.7) 相同，但 $D$ 点在拓扑上与 $G$ 点相同，换句话说，对于 $k = k_0$，如果在 $D$ 点有 (6.7)，那么对于 $k = k_0 + 1$，它仍在 $D$ 点成立。前面已经显示了 $k = 0$ 时循环不变式成立，经过推导，对于所有的 k，$0 \leqslant k \leqslant n$，都有循环不变式 (6.4)。

剩下的就是证明循环终止。最初由断言 (6.6)，k 的值等于 0。循环每次迭代时执行 $k \leftarrow k + 1$，每次使 k 值增加 1，最后，k 一定会达到 n 值，那时将退出循环，并且断言 (6.8) 给出 s 的值，这样就满足了输出规格说明 (6.3)。

回顾一下，给定输入规格说明 (6.1)，可以证明不论该循环执行 0 次、1 次或更多次，循环不变式 (6.4) 成立。进一步地，可以证明经过 n 次迭代后，循环终止，而且这时 k 和 s 的值满足输出规格说明 (6.3)。换句话说，图 6-4 的代码段经过数学证明是正确的。

### 6.5.2 正确性证明小型实例研究

正确性证明的一个重要方面是它们应与设计和编程结合进行。Dijkstra 把它表达为“程序员应让程序证明和程序一起发展”[Dijkstra，1972]。例如，当在设计中应用循环时，会提出循环不变式；而且当对设计逐步求精时，不变式也被逐步求精。以这种方式开发产品给程序员以信心，相信产品是正确的，并趋向于减少错误的数量。再次引用 Dijkstra 的话，“提高程序的信心程度的惟一有效方式是对它的正确性给出有说服力的证明”[Dijkstra，1972]。但即使证明产品是正确的，它也必须再进行全面的测试。为说明测试结合正确性证明的必要性，考虑下面的例子。

1969 年，Naur 报告了一种构造和证明产品正确的技术 [Naur，1969]。Naur 用行编辑问题对该技术进行了阐述，今天这可看成是文本处理问题。它如下叙述：

给定一个文本，它包含有以“空格”符或以“新行”符分开的单词，依照下列原则转化

为一行接一行的格式：

1．只在包含空格或新行的地方才能断行；

2．只要可能，尽可能填充每一行；

3．每行不会包含超过“最长”符的字符。

Naur 使用他的技术构造了一个过程，并非形式地证明了它的正确性。该过程包含大约 25 行 ALGOL 60。然后论文由《Computing Reviews》的 Leavenworth 进行评审［Leavenworth，1970］。评审者指出，在 Naur 过程的输出中，第一行的第一个词前面有一个空格，除非第一个词正好有“最长”符那么长。尽管这看起来是个小错误，但它肯定会在测试过程时检测出来，也就是说，用测试数据运行，而不只是证明是正确的。但更糟的还在后面，London 在 Naur 的过程里检测到 3 个额外的错误［London，1971］。一个是该过程不能终止，除非有一个词的长度比“最长”符还要长。再有，如果该过程经过测试，这个问题很可能会测试出来。然后 London 提出了该过程的一个修正版本，并形式地证明它是正确的，而 Naur 使用的只是非形式的证明技术。

这个故事中的下一个插曲是 Goodenough 和 Gerhart［1975］发现了 3 个 London 没有发现的错误，尽管有他的形式“证明”。这些错误包括最后一个词不会输出，除非它后面有“空格”或“新行”字符。同样，合理选择测试数据将很容易检测到这个错误。事实上，由 Leavenworth、London、Goodenough 和 Gerhart 发现的总共这 7 个错误中，只要用测试数据运行该过程就能检测到 4 个错误，就像 Naur 的原始论文中给出的阐述一样。从这个故事中得到的教训很明显，即使产品证明是正确的，还是需要对它进行全面的测试。

6.5.1 节中的例子显示出，甚至证明一个小代码段的正确性过程也很长，进一步说，本节研究的实例显示出证明正确性是一个困难和有可能出错的过程，即使是对一个只有 25 行代码的过程。所以必须提出如下问题：正确性证明只是一个有趣的研究想法呢，还是它是一个强有力的软件工程技术，它的时代已经到来？下一节将回答这个问题。

### 6.5.3　正确性证明和软件工程

许多软件工程实践者提出，正确性证明不能被看成是标准的软件工程技术。首先，他们声称软件工程师缺乏充分的数学培训；其次，认为证明太昂贵，没有实用性；第三，证明太难了。下面表明这些原因都过于简化了：

1）尽管 6.5.1 节给出的证明并不比高中代数难于理解，但此类证明需要在第一或第二次谓词计算或类似计算中表示输入规格说明、输出规格说明和循环不变式。这不仅使数学家证明过程更容易些，还允许由计算机进行正确性证明。由于使事情进一步复杂化，现在谓词计算有点过时了，为证明并发产品的正确性，要求使用时间的或其他形式逻辑的技术［Manna and Pnueli，1992］。毫无疑问，正确性证明要求在数学逻辑上有所培训，所幸，今天大多数计算机专业的学生要么选修了必需的课程，要么有过学习正确性证明技术的工作经历。所以，现在大学输送出的计算机专业毕业生，具有的数学技能足以进行正确性证明。声称软件工程实践者缺乏必要的数学培训在过去可能是这样，但随着每年该行业加入数以千计的计算机专业学生，情况不再是这样。

2）声称在软件开发中使用证明太昂贵，这也不对。相反，可以在项目到项目的基础上应用成本－效益分析法（5.2 节），确定正确性证明的经济性。例如，考虑为 NASA 空间站

开发的软件，如果一旦出错，航天飞机的救援行动不能按时到达，人的生命则处在危险之中。证明生命攸关的空间站软件的正确性成本是很巨大，但如果没有进行正确性证明，可能被忽视的软件错误造成的潜在成本会更巨大。

3）第三个声称是正确性证明太难，尽管这样，还是有许多重要的产品被成功地证明是正确的，包括操作系统内核、编译器和通信系统［Landwehr，1983；Berry and Wing，1985］。进一步说，许多像定理证明器这样的工具有助于正确性证明。定理证明器将产品、产品的输入和输出规格说明以及循环不变式作为它的输入，然后试图从数学角度证明，当给定的输入数据满足输入规格说明时，产品将产生满足输出规格说明的输出数据。

与此同时，正确性证明也面临一些困难：

- 例如，我们如何确认定理证明器是正确的？如果定理证明器打印出“该产品正确”的信息，我们能相信它吗？作为极端的情况，考虑图 6-7 所示的所谓的定理证明器，无论给这个定理证明器提交什么代码，它都将打印“该产品正确”的信息。换句话说，定理证明器的输出应具有怎样的可靠性？一个建议是将定理证明器本身提交给它，看它是否正确。除了哲学含义，看这样做是否行得通的一个简单方法是，如果将图 6-7 所示的定理证明器提交给它本身来证明，看会发生什么。一如往常，它会打印出“该产品正确”的信息，因而“证明”了它自己的正确性。

```
void theoremProver ( )
{
   print "This product is correct";
}
```

图 6-7 定理证明器

- 更进一步的困难是找出输入和输出规格说明，特别是循环不变式或处于其他逻辑状态（诸如形式逻辑）的同等物。假设一个产品是正确的，除非能为每个循环找到一个合适的不变式，否则没有办法证明产品是正确的。诚然，是有工具帮助做这项工作，但即便是使用目前最先进的工具，软件工程师也还是不能简单地得出正确性证明。解决这个问题的一个方案是，如 6.5.2 节所提倡的那样，开发产品与证明并行完成。在设计一个循环的同时，也确定了该循环的不变式。应用这个方法，证明一个代码模块的正确性会容易些。
- 比不能找出循环不变式更难的是，如果规格说明本身不正确怎么办？这样的一个例子是方法 `trickSort`（6.2 节）。当给出图 6-1 所示的不正确的规格说明时，一个好的定理证明器毫无疑问会断定图 6-2 所示的方法是正确的。

Manna 和 Waldinger 声明说，“我们不能确认规格说明是正确的”以及“我们不能确认一个验证系统是正确的”［Manna and Waldinger，1978］。由该领域里两个顶尖的专家做出的声明浓缩了先前说明的各种观点。

所有这些意味着软件工程中没有正确性证明的位置吗？恰恰相反。证明产品正确是一个重要的，有时是至关重要的软件工程工具。在人命关天的场合或成本－效益分析法指出有必要证明的场合，应当采用正确性证明。如果证明软件正确性的花费比产品出现故障时可能的花费少，那么应当证明产品。然而，就像文本处理实例研究所显示的，单单证明是不够的，正确性证明应该看作是综合验证产品正确性的整套技术中的一个重要组成部分。因为软件工程的目标是软件的高质量，正确性证明确实是一个重要的软件工程技术。

即便一个完全正规的证明没有证明产品正确，软件的质量仍可通过使用非形式的证明来

得到明显提高。例如，一个与6.5.1节相类似的证明将有助于检验一个循环正确执行的次数。提高软件质量的第二个方式是插入像图6-6中的那些断言到代码中，然后，如果在执行时没有证实某断言，则产品应被停止运行，软件开发小组查清终止执行的断言是不正确的呢，还是代码中真的有错误，通过触发断言而被检测到。像Java（1.4版）这样的语言支持通过assert语句直接进行断言。假设一个非形式的证明要求代码中特定点上的变量xxx的值是正的，即使设计小组可确信变量xxx不可能是负的，为了更可靠，他们可以指定下面的语句出现在代码中的那一点上：

**assert**（xxx>0）

如果xxx小于或等于0，程序将停止运行，然后软件小组可以查明这种情况。遗憾的是，在C++中assert是一个调试状态，与C中的assert相似，它不是该语言的一部分。Ada 95通过**pragma**支持断言。

一旦用户有信心认为产品工作正确，他们可以关闭断言检验，这将加快运行速度，但如果关闭了断言检验，将不会找到能由断言检测出的错误。所以，在运行效率和产品安装到客户计算机上之后仍继续断言检验之间有一个权衡。（下面的“如果你想知道”部分在这个问题上给出了一个有趣的深入剖析。）

执行测试的一个基本问题是软件开发小组的哪一个成员应该负责实现它。

**如果你想知道**

像Java（但不是C或C++）这样的语言的一个特性是边界检验。边界检验的一个例子是在执行时检查每个数组索引以确保它在声明的范围内。

Hoare建议在开发产品时使用边界检验，一旦产品运行正确就停止使用它，这有些像在陆地上穿着救生衣学习航海，然后真正在大海上时脱掉救生衣。在他的图灵奖讲稿中，Hoare描述了一种他在1961年开发的编译器［Hoare，1981］。当后来给用户提供在编译器的最后版本安装后，关掉边界检验的机会时，他们一致拒绝了，因为他们在该编译器的先前版本的测试运行期间，已经遇到了许多变量值超出范围的事件。

边界检验可看做是更普遍的概念——断言检验的一个特例。Hoare的救生衣比喻与一旦安装了最后版本就可以去掉断言检验一样。

Hoare的评论不幸被言中了。今天，黑客们用来渗透到计算机中的一项主要技术就是向操作系统发送一个长数据流，故意造成缓冲区溢出，并且用恶意的可执行代码覆盖一部分操作系统。这项技术仅在向用C或C++编写的操作系统的缓冲区中读数据的情况下，程序员忽略了在代码中包含边界检验或关闭了边界检验时才能够工作。

## 6.6　谁应当完成执行测试

假设要求一个程序员测试他或她写出的一个代码制品，Myers已经描述了测试是一个执行产品的过程，目的是找到错误［Myers，1979］。所以测试是一个破坏性过程。另一方面，进行测试的程序员通常不希望破坏他或她的工作成果。如果程序员对代码的基本态度是保护性的，那么程序员使用测试数据发现错误的机会很有可能比主要动机是真正的破坏性时要少。一个成功的测试是能够发现错误的测试。这也造成一个问题，它意味着如果该代码制品通过了测试，那么测试是失败的。相反，如果该代码制品没有按照规格说明执行，则测试是成功的。要求一个程序员测试他或她自己写的模块，就是要求他或她以一种错误（不正确的

行为）接着发生的方式来执行模块，这与程序员的创造天性是背道而驰的。

毫无疑问，程序员不应测试他们自己的模块。在程序员建设性地构造了一个模块之后，测试那个模块要求创建者进行一个破坏性的行动，并试图破坏构造的东西。执行测试应该由其他人完成的第二个原因是：程序员可能会误解设计或规格说明的某些方面，如果测试由其他人完成，将会发现这样的错误。不过，调试（发现故障的原因并改正错误）最好由最初的程序员做，他是对代码最熟悉的人。

程序员不应测试他或她自己的代码这种论断一定不要太过绝对。考虑编程过程，程序员开始时阅读该模块的详细设计，这可能是流程图的形式，或者更可能是伪码。但无论使用什么技术，程序员必须在输入到计算机之前进行**桌面检查**。也就是说，程序员必须用多种测试用例考验流程图或伪码，跟踪详细的设计来检验每个测试用例都能正确执行。只有当程序员认为详细设计正确时，才调用文本编辑器并对该模块进行编程。

一旦模块成为机器可读的形式，它将接受一系列测试。测试数据用于确定模块是否能正常工作，可能在桌面检查详细设计时会用到同样的测试数据。接下来，如果使用正确的测试数据时，模块运行正常，那么程序员使用不正确的数据来测试该模块的健壮性。当程序员满意地认为该模块运行正确时，系统的测试开始了。**系统的测试**不应由程序员进行。

如果程序员不能进行系统的测试，谁应去做它？如6.1.2节所述，独立的测试必须由SQA小组进行，这里的关键词是*独立的*。只有SQA小组真正地与开发小组独立，SQA的成员才能履行确保产品真正满足规格说明的任务，没有软件开发管理者施加诸如产品的最后期限将妨碍工作这样的压力。SQA组的个人必须给他们的管理者递交报告，这样来保护他们的独立性。

系统的测试如何进行？一个测试用例的基本部分是在测试开始执行前声明期望的输出。测试者坐在终端前，执行模块，输入任意的测试数据，然后扫一眼屏幕，说“我猜这看起来是对的”，这完全是在浪费时间。同样无益的是测试者很认真地准备测试数据，依次执行每个测试用例，查看输出，并说“是的，这看起来当然是对的。”人们很容易被似是而非的结果所欺骗，如果允许程序员测试他们的代码，那么总会有这样的危险，程序员将看到他或她想要看到的。即便测试由其他人完成，也会发生同样的问题。解决的办法是在管理上坚持要求在测试开始前记录测试数据和期望的测试结果。在测试完成后，应记录实际的结果，并与期望的结果进行对比。

即便在小的组织里开发小的产品，以机器可读的形式进行这种记录也很重要，因为测试用例不应抛弃。这样做的原因在于维护，当维护产品时，必须进行回归测试。产品先前正确执行过的测试用例必须重新运行，以确保对产品增加了新功能后没有破坏产品现有的功能，这将在第15章进一步讨论。

## 6.7 测试什么时候停止

产品成功地维护许多年之后，它将渐渐地失去作用并被一个完全不同的产品所取代，基本上与晶体管取代电子管相同。或者，产品仍旧有用，但使它与新的硬件接口或在一个新的操作系统下运行它，所需的成本远远大于建造一个新产品，这时可以将旧的产品作为原型。因此，最后软件产品会退役，不再服务。只有在义无反顾地废除软件时，才是停止测试的时候。

既然已经介绍了所有必须的背景材料，就可以更深入地考察对象了。这是下一章的内容。

## 本章回顾

本章的一个核心主题是测试必须与软件过程的所有活动并行进行。本章一开始描述了质量问题（6.1 节），随后，描述了非执行测试（6.2 节），其中详细讨论了走查和审查。在这之后定义了执行测试的概念（6.3 节和 6.4 节），并讨论了必须对其进行测试的一个产品的行为属性，包括实用性、可靠性、健壮性、性能和正确性（6.4.1 节至 6.4.5 节）。在 6.5 节中介绍了正确性证明，并在 6.5.1 节中给出了一个这样的例子，然后分析了软件工程中正确性证明的角色（6.5.2 节和 6.5.3 节）。另一个重要的事项是系统的执行测试必须由独立的 SQA 小组来完成，而不是由程序员来完成（6.6 节）。最后，在 6.7 节中讨论了何时测试最终结束。

## 进一步阅读

经过多年以后，软件开发者对测试过程的态度已在改变，从先前的把测试看作是显示产品运行正常的一种方法，到现代的认为应用测试来防止需求、分析、设计和实现出现错误。这个发展过程在［Gelperin and Hetzel，1988］中有所描述，软件测试的本质和它如此之难的原因在［Whittaker，2000］中有讨论。在［Lieberman and Fry，2001］中描述了错误存在的普遍和深入性。降低错误数量的方法出现在［Boehm and Basili，2001］中。

关于软件质量的一些神话般的故事在［Voas，1999］中有所讨论。［Whittaker and Voas，2000］中给出了一个有关可靠性的新的理论。

［Baber，1987］很好地介绍了有关证明程序是正确的内容。正确性证明的一个标准技术是使用所谓的 Hoare 逻辑，在［Hoare，1969］中描述了它。另一个确保产品满足规格说明的可选方法是建造产品的分步阶段，检验每个步骤都保持正确性，这在［Dijkstra，1968］和［Wirth，1971］中有所描述。关于软件工程界接受的正确性证明方面的重要文章是［DeMillo，Lipton，and Perlis，1979］。

IEEE的“软件评审标准”［IEEE 1028，1997］是有关非执行测试的非常好的信息源。在［Perry et al.，2002］中描述了检查评估大型软件产品的一些试验。［Vitharana and Ramamurthy，2003］建议审查应当匿名进行并通过计算机中介。小组处理支持对于审查的影响在［Tyran and George，2002］中给出。

执行测试方面的经典著作是［Myers，1979］，它在测试方面有很大的影响。［DeMillo，Lipton，and Sayward，1978］中含有关于选择测试数据的非常有用的信息。［Beizer，1990］是关于测试的概要，是该课题的真正有用的参考书。另一个类似的著作是［Hetzel，1988］。

说到面向对象的范型，［Kung，Hsia，and Gao，1998］是关于面向对象测试方面的著作，［Sykes and McGregor，2000］也是。

国际软件测试与分析论坛（International Symposium on Software Testing and Analysis）学报包含了类似的测试方面的文章。《IEEE Transactions on Software Engineering》2002 年 2 月刊包含各种来自 2000 论坛的文章；［Elbaum，Malishevsky，and Rothermel，2002］是其中特别有意思的一篇。

## 习题

6.1　本书中是如何使用“正确性证明”、“验证”和“确认”这些术语的？

6.2 一个软件开发组织目前雇用了94个软件专业人员，包括19个管理者，所有的人进行软件的开发和测试，最新的数字表明他们的27%时间消耗在测试活动上。公司中一个管理者平均每年的成本是144 000美元，而非管理性专业人员的成本每年平均为107 000美元，这两个数据都包含加班的成本。请使用成本－效益分析法来确定是否应在组织内部建立一个单独的SQA小组。

6.3 假设习题6.2的情况只有7个软件专业人员，包括2个管理者，其他的数据均保持不变，重复使用成本－效益分析法来确定是否应在组织内部建立一个单独的SQA小组。

6.4 假设你已经持续11天测试一个代码制品，并发现2个错误，请问这能预示其他错误的存在吗？

6.5 走查和审查之间有什么相似之处？又有什么不同之处？

6.6 假设你是Ye Olde Fashioned Software公司SQA小组的成员，你建议管理者引入审查机制，他的反应是没有必要浪费4个人的时间来找寻错误，只需编写代码的人运行测试用例即可。你如何回答他？

6.7 假设你是Hardy Hardware公司的SQA管理者，该公司是有754个分支机构的硬件代理商。公司正考虑购买一种库存控制软件包，以便在整个组织内使用。在批准购买该软件包之前，你决定对它进行全面的测试，你将会调查它的什么特性？

6.8 Hardy Hardware公司的所有754个分支现在要用通信网络连接起来，通信软件包的销售代表给你提供4周免费试用期，为此你将进行什么样的软件测试，为什么？

6.9 你是Transmyopia共和国的陆军准将，负责开发控制激光制导反坦克导弹的软件（习题1.4），该软件已经交付给你，等待验收测试。你将测试软件的哪些属性？

6.10 如果使用循环不变式

```
s = y[0] + y[1] + ... + y[k-1]
```

而不使用（6.4）式，6.5.1节的正确性证明将会出现什么情况？

6.11 假定你有过一些循环不变式的经验，你知道（6.4）式是图6-6循环的正确不变式，请说明输出规格说明（6.3）是该循环不变式的正常结果。

6.12 考虑下面这段代码：

```
k = 1;
g = 2;
while (k < n)
{
    k = k + 1;
    g = g * 2;
}
```

证明：如果n是一个正整数，这段代码正确地计算了$g = 2^n$。

6.13 正确性证明能够解决交付给客户的产品可能不是客户真正需要的这类问题吗？请回答并解释原因。

6.14 应当怎样改变Dijkstra的陈述（6.3节），将它用于正确性证明而不是测试？别忘了6.5.2节中的小型实例研究。

6.15 使用由你的导师规定的语言设计并实现Naur文本处理问题（6.5.2节）的一个解决方案。用测试数据执行它并记录下发现的错误数和每个错误的原因（例如，逻辑错误，循环计数器错误）。不要纠正你查出的任何错误，现在与同学交换产品，看看你在其他人的产品中发现多少错误以及它们是否是新的错误，再一次记录下每个错误的

原因并且比较每个人发现的错误类型。将班上的结果统一列表。

6.16 （学期项目）解释你将如何测试附录 A 中 Ophelia's Oasis 产品的实用性、可靠性、健壮性、性能和正确性。

6.17 （软件工程读物）你的导师将向你们分发［Lieberman and Fry，2001］的复印件。你同意作者悲观的观点吗？

## 参考文献


[Ackerman, Buchwald, and Lewski, 1989] A. F. ACKERMAN, L. S. BUCHWALD, AND F. H. LEWSKI, "Software Inspections: An Effective Verification Process," *IEEE Software* **6** (May 1989), pp. 31–36.

[Baber, 1987] R. L. BABER, *The Spine of Software: Designing Provably Correct Software: Theory and Practice*, John Wiley and Sons, New York, 1987.

[Beizer, 1990] B. BEIZER, *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, New York, 1990.

[Berry and Wing, 1985] D. M. BERRY AND J. M. WING, "Specifying and Prototyping: Some Thoughts on Why They Are Successful," in: *Formal Methods and Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Vol. 2, Springer-Verlag, Berlin, 1985, pp. 117–28.

[Boehm and Basili, 2001] B. BOEHM AND V. R. BASILI, "Software Defect Reduction Top Ten List," *IEEE Computer* **34** (January 2001), pp. 135–37.

[Bush, 1990] M. BUSH, "Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory," *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, March 1990, pp. 196–99.

[DeMillo, Lipton, and Perlis, 1979] R. A. DEMILLO, R. J. LIPTON, AND A. J. PERLIS, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM* **22** (May 1979), pp. 271–80.

[DeMillo, Lipton, and Sayward, 1978] R. A. DEMILLO, R. J. LIPTON, AND F. G. SAYWARD, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer* **11** (April 1978), pp. 34–43.

[Dijkstra, 1968] E. W. DIJKSTRA, "A Constructive Approach to the Problem of Program Correctness," *BIT* **8** (No. 3, 1968), pp. 174–86.

[Dijkstra, 1972] E. W. DIJKSTRA, "The Humble Programmer," *Communications of the ACM* **15** (October 1972), pp. 859–66.

[Elbaum, Malishevsky, and Rothermel, 2002] A. ELBAUM, A. G. MALISHEVSKY, AND G. ROTHERMEL, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Transactions on Software Engineering* **28** (2002), pp. 159–82.

[Fagan, 1976] M. E. FAGAN, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* **15** (No. 3, 1976), pp. 182–211.

[Fagan, 1986] M. E. FAGAN, "Advances in Software Inspections," *IEEE Transactions on Software Engineering* **SE-12** (July 1986), pp. 744–51.

[Fowler, 1986] P. J. FOWLER, "In-Process Inspections of Workproducts at AT&T," *AT&T Technical Journal* **65** (March/April 1986), pp. 102–12.

[Gelperin and Hetzel, 1988] D. GELPERIN AND B. HETZEL, "The Growth of Software Testing," *Communications of the ACM* **31** (June 1988), pp. 687–95.

[Goodenough, 1979] J. B. GOODENOUGH, "A Survey of Program Testing Issues," in: *Research Directions in Software Technology*, P. Wegner (Editor), The MIT Press, Cambridge, MA, 1979, pp. 316–40.

[Goodenough and Gerhart, 1975] J. B. GOODENOUGH AND S. L. GERHART, "Toward a Theory of Test Data Selection," *Proceedings of the Third International Conference on Reliable Software*, Los Angeles, 1975, pp. 493–510; also published in *IEEE Transactions on Software Engineering* **SE-1** (June 1975), pp. 156–73. Revised version: J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection: Data Selection Criteria," in: *Current Trends in Programming Methodology*,

Vol. 2, R. T. Yeh (Editor), Prentice-Hall, Englewood Cliffs, NJ, 1977, pp. 44–79.

[Hetzel, 1988] W. HETZEL, *The Complete Guide to Software Testing*, 2nd ed., QED Information Systems, Wellesley, MA, 1988.

[Hoare, 1969] C. A. R. HOARE, "An Axiomatic Basis for Computer Programming," *Communications of the ACM* **12** (October 1969), pp. 576–83.

[Hoare, 1981] C. A. R. HOARE, "The Emperor's Old Clothes," *Communications of the ACM* **24** (February 1981), pp. 75–83.

[IEEE 610.12, 1990] *A Glossary of Software Engineering Terminology*, IEEE 610.12-1990, Institute of Electrical and Electronic Engineers, New York, 1990.

[IEEE 1028, 1997] *Standard for Software Reviews*, IEEE 1028, Institute of Electrical and Electronic Engineers, New York, 1997.

[Jones, 1978] T. C. JONES, "Measuring Programming Quality and Productivity," *IBM Systems Journal* **17** (No. 1, 1978), pp. 39–63.

[Kelly, Sherif, and Hops, 1992] J. C. KELLY, J. S. SHERIF, AND J. HOPS, "An Analysis of Defect Densities Found during Software Inspections," *Journal of Systems and Software* **17** (January 1992), pp. 111–17.

[Kung, Hsia, and Gao, 1998] D. C. KUNG, P. HSIA, AND J. GAO, *Testing Object-Oriented Software*, IEEE Computer Society Press, Los Alamitos, CA, 1998.

[Landwehr, 1983] C. E. LANDWEHR, "The Best Available Technologies for Computer Security," *IEEE Computer* **16** (July 1983), pp. 86–100.

[Leavenworth, 1970] B. LEAVENWORTH, Review #19420, *Computing Reviews* **11** (July 1970), pp. 396–97.

[Lieberman and Fry, 2001] H. LIEBERMAN AND C. FRY, "Will Software Ever Work?" *Communications of the ACM* **44** (March 2001), pp. 122–24.

[London, 1971] R. L. LONDON, "Software Reliability through Proving Programs Correct," *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, Pasadena, CA, March 1971.

[Manna and Pnueli, 1992] Z. MANNA AND A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York, 1992.

[Manna and Waldinger, 1978] Z. MANNA AND R. WALDINGER, "The Logic of Computer Programming," *IEEE Transactions on Software Engineering* **SE-4** (1978), pp. 199–229.

[Myers, 1979] G. J. MYERS, *The Art of Software Testing*, John Wiley and Sons, New York, 1979.

[Naur, 1969] P. NAUR, "Programming by Action Clusters," *BIT* **9** (No. 3, 1969), pp. 250–58.

[Perry et al., 2002] D. E. PERRY, A. PORTER, M. W. WADE, L. G. VOTTA, AND J. PERPICH, "Reducing Inspection Interval in Large-Scale Software Development," *IEEE Transactions on Software Engineering* **28** (July 2002), pp. 695–705.

[Sykes and McGregor, 2000] D. A. SYKES AND J. D. MCGREGOR, *Practical Guide to Testing Object-Oriented Software*, Addison Wesley, Reading, MA, 2000.

[Tyran and George, 2002] C. K. TYRAN AND J. F. GEORGE, "Improving Software Inspections with Group Process Support," *Communications of the ACM* **45** (September 2002), pp. 87–92.

[Vitharana and Ramamurthy, 2003] P. VITHARANA AND K. RAMAMURTHY, "Computer-Mediated Group Support, Anonymity and the Software Inspection Process: An Empirical Investigation," *IEEE Transactions on Software Engineering* **29** (March 2003), pp. 167–80.

[Voas, 1999] J. VOAS, "Software Quality's Eight Greatest Myths," *IEEE Software* **16** (September/October 1999), pp. 118–20.

[Whittaker, 2000] J. A. WHITTAKER, "What Is Software Testing? And Why Is It So Hard?" *IEEE Software* **17** (January/February 2000), pp. 70–79.

[Whittaker and Voas, 2000] J. A. WHITTAKER AND J. VOAS, "Toward a More Reliable Theory of Software Reliability," *IEEE Computer* **33** (December 2000), pp. 36–42.

[Wirth, 1971] N. WIRTH, "Program Development by Stepwise Refinement," *Communications of the ACM* **14** (April 1971), pp. 221–27.

# 第 7 章　从模块到对象

**学习目标**

学完本章后，你应当能够：

- 设计带有高内聚和低耦合的模块与类；
- 理解信息隐藏的必要性；
- 描述继承、多态和动态绑定的软件工程含义；
- 区分泛化、聚合和关联的不同；
- 较以往更深入地讨论面向对象范型。

一些较为耸人听闻的计算机杂志似乎认为，面向对象范型是 20 世纪 80 年代中期突然、戏剧性出现的新发现，是替代当时流行的传统范型的变革。实际并不是这样，在 20 世纪 70 年代和 80 年代期间，模块化理论经历了稳步的发展，而对象只是模块化理论中的一个逻辑发展（参见下面的“如果你想知道”部分）。本章在模块化的范畴内描述对象。

**如果你想知道**

面向对象的概念最早是在 1966 年的仿真语言 Simula 67 中出现的［Dahl and Nygaard, 1966］。然而在那时，该项技术太超前，不实用，所以一直处于搁置状态，直到 20 世纪 80 年代初在模块化理论的内容中重新使用它。

本章还有其他有关前沿技术搁置着，直到世界做好准备来利用它的例子。如 Parnas 于 1971 年在软件著作中首次提出的信息隐藏（7.6 节）［Parnas, 1971］，该技术大约在 10 年后，封装和抽象数据类型成为软件工程的一部分时，才被广泛地接受。

好像人类总是要在准备好时才能接受新事物，而并不一定是在它们刚出现时就接受它。

采用这种方法是因为，如果不理解为什么面向对象范型优于传统范型，那么很难正确使用对象，而且为了这样做，有必要明白：对象只是始于模块概念的知识体系的下一个逻辑步。

## 7.1　什么是模块

当大型产品由单个的代码块整体组成时，维护将是一件可怕的事情。即使是这种产品的作者，试图调试代码也相当困难，而让另一个程序员去读懂它几乎不可能。解决办法是将该产品分成较小的块，称之为模块。什么是模块呢？是将产品分解成模块的方式本身重要呢，还是只是将大型产品分为小的代码块重要呢？

最早试图描述模块的是 Stevens、Myers 和 Constantine［1974］，他们把**模块**定义为“一个或多个邻接的程序语句的集合，它有一个名称以便系统的其他部分调用它，并且最好具有自己专用的变量名集。”换句话说，一个模块由一个单独的代码块组成，它可像过程、函数或方法一样被调用。这个定义似乎相当宽泛，它包含各种过程和函数，不管它们是内部编译还是单独编译；它也包括 COBOL 段落和节，尽管它们不具备自己的变量，因为该定义声明的只是“最好”拥有专门的变量名集的特性；它还包含嵌套在其他模块内部的模块。但尽管该定义很宽泛，却还是不够，例如，不能调用汇编器宏，因此根据前面的定义，它不是一个模块。在 C 和 C++ 中，包含在产品中的由声明组成的头文件，同样没有被调用。一句话，这个定义太严格了。

Yourdon 和 Constantine [1979] 给出了一个更宽泛的定义："模块是一个词汇上邻接的程序语句序列，由边界元素限制范围，有一个聚合标识符。"边界元素的例子是像 Pascal 这样的块结构化语言中的 begin...end 对，或者 C++ 或 Java 中的{...}对。这个定义不仅包含由前面的定义所排除的所有情形，它的宽泛性使它还能完全适用于本书。特别地，传统范型的过程和函数都是模块。在面向对象范型中，一个对象是模块，对象内的方法也是模块。

为了理解模块化的重要意义，考虑下面这个略带想像的例子。John Fence 是一个极不称职的计算机体系结构设计师，他一直未发现 NAND（与非）门和 NOR（或非）门是完备的，即每个电路都可以仅用 NAND 门或仅用 NOR 门建造。因此 John 决定使用 AND（与门）、OR（或门）和 NOT（非门）来建造一个 ALU、移存器和 16 位寄存器。得到的计算机如图 7-1 所示，三个组件以简单的方式连接在一起。现在我们的设计师朋友决定该电路应建造于三个硅芯片上。这样他设计了如图 7-2 所示的三个芯片。一个芯片上有 ALU 的所有门，第二个芯片上有移存器的所有门，第三个芯片上有寄存器的所有门。这时 John 依稀想起酒吧里的什么人告诉过他最好建造只有一种门的芯片，所以他重新设计了芯片。在芯片 1 上他放置了所有的与门，在芯片 2 上他放置了所有的或门，而芯片 3 上放置了所有的非门。得到的"艺术作品"如图 7-3 所示。

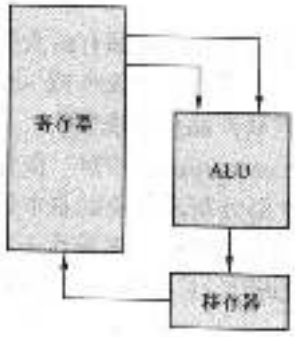



图 7-1　一个计算机的设计

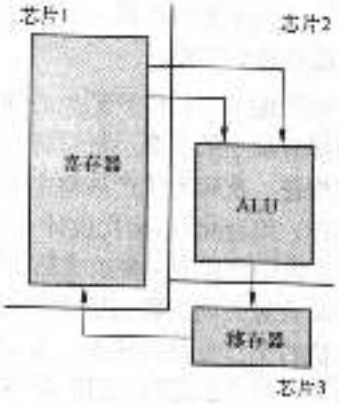



图 7-2　图 7-1 的计算机建造在三个芯片上

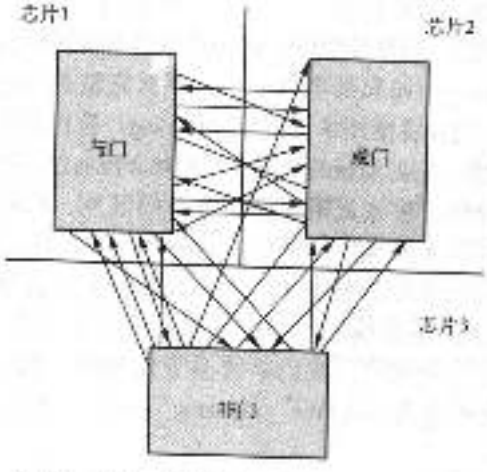



图 7-3　图 7-1 的计算机建造在为三个芯片上

图 7-2 和图 7-3 的功能是一样的，也就是它们做相同的事情。但这两种设计显然有不同的特性：

1）图 7-3 比图 7-2 更难“理解”，几乎任何懂点数字逻辑的人都会立即明白图 7-2 的芯片构成了一个 ALU、一个移存器和一组寄存器。然而，即便是一个顶尖的硬件专家也很难明白图 7-3 中这众多的与门、或门和非门的作用。

2）图 7-3 所示的电路很难进行*纠错性维护*。要是计算机中有一个设计错误的话（任何能够得出图 7-3 的人都毫无疑问地会出现很多错误），这将难于确定错误的位置。另一方面，如果图 7-2 中的计算机有一个设计错误，则可通过确定它是以 ALU 工作的方式出现、以移存器工作的方式出现，还是以寄存器工作的方式出现来定位错误。类似地，如果图 7-2 中的计算机瘫痪了，当然很容易确定应替换哪一个芯片，而如果图 7-3 中的计算机瘫痪了，解决的办法可能最好是替换所有三个芯片。

3）图 7-3 的计算机难于*扩展或提高*。如果需要一种新型的 ALU 或更快的寄存器，则必须从设计阶段重新开始设计，但图 7-2 所示的计算机的设计就容易替换芯片。可能最糟糕的是，图 7-3 的芯片在任何新产品中都不能*重用*，专门为该产品设计的由与门、或门和非门构成的组合不能用于其他产品，而图 7-2 的三个芯片很可能在其他要求有 ALU、移存器或寄存器的产品中得到重用。

这里的关键点是软件产品也必须设计成像图 7-2 一样，每个芯片内部有最大的关联，而芯片之间却有最小的关联。可以将一个模块比作一个芯片，它完成一个操作或一组操作并与其他模块相连。在确定了产品整个的功能后，需要确定如何将产品分割成模块。如第 1 章所指出的那样，组合化/结构化设计［Stevens，Myers，and Constantine，1974］作为降低维护成本（整个软件预算的主要组成部分）的途径，提供了将产品分割成模块的基本原理。当每个模块内部有最大的关联而模块之间有最小的关联时，不管是纠错性、完善性还是适应性的维护，维护的工作量都会减少。换句话说，组合化/结构化设计（C/SD）的目的是确保组成产品的模块与图 7-2 类似，而不是与图 7-3 类似。

Myers［1978b］量化了模块**内聚**（cohesion，即一个模块内部交互的程度）和模块**耦合**（coupling，即两个模块之间交互的程度）的思想。更准确地说，Myers 使用的是术语**强度**（strength），而不是内聚。然而，内聚更可取，因为模块可有高强度或低强度之分，而在低强度的形容里有一些固有的矛盾——如果某事物不强，那么它就是弱的。为防止术语的不准确，C/SD 现在使用术语内聚。一些作者使用术语**绑定**（binding）替代耦合，然而遗憾的是，绑定还可用于计算机科学的其他方面，例如变量的绑定值。但耦合没有这些隐含意义，因而最合适。

此时有必要明确模块操作、模块逻辑和模块背景的区别，模块**操作**指的是模块做什么，也就是它的行为。例如，模块 m 的操作是计算它的参数的平方根。模块**逻辑**指的是模块如何完成它的操作，在上述的模块 m 的情况中，计算平方根的特定方法是牛顿方法［Gerald and Wheatley，1999］。模块**背景**指的是模块的特定用途，例如使用模块 m 来计算一个双精度整数的平方根。C/SD 的关键是分配给模块的名称是它的操作，而不是它的逻辑或背景。所以，在 C/SD 中模块 m 的名称应是 `compute _ square _ root`[○]（计算平方根），它的名称与它的逻辑和背景是不相关的。

○ 为了增加清晰度，在像 compute _ square _ root 这样的函数名中使用了下划线，以强调在这里以及下面各节中使用了结构化范型。当使用面向对象范型时（从 7.4.2 节往后），相应的方法应当命名为 computeSquareRoot。

## 7.2 内聚

Myers [1978b] 定义了内聚的7个分类或级别。根据现代计算机科学理论，接下来将看到，Myers的前两个级别应该交换，因为信息性内聚比功能性内聚更加支持重用，后面将会讲到。得到的各分类的排列顺序如图7-4所示，这不是任何类型的线性度量，只是一种相关联的分级顺序，用于确定哪种类型的内聚高（好的），哪种类型的内聚低（坏的）。




图7-4 内聚级别

为了理解什么构成了一个高内聚的模块，有必要从另一方面，从较低的内聚级开始讨论。

### 7.2.1 偶然性内聚

如果一个模块执行多个完全不相关的操作，则具有**偶然性内聚**（coincidental cohesion）。具有偶然性内聚的模块例子是具有如下名称的模块：print_the_next_line，reverse_the_string_of_characters_comprising_the_second_argument，add_7_to_the_fifth_argument，convert_the_fourth_argument_to_floating_point（打印下一行，转换组成第二个参数的字符串，给第五个参数加上7，把第四个参数转换为浮点）。一个明显的问题是，这样的模块怎么会在实际中出现？最通常的理由是作为严格强制的规则（例如"每个模块应由35—50个可执行语句构成"）的结果。如果软件组织坚持认为模块必须是既不能太大，也不能太小，那么将发生两件令人不快的事情。首先，必须将两个或更多不那么完美的小模块集中在一起，形成具有偶然性内聚的一个较大模块。其次，从设计良好的模块（管理者认为该模块过大）中删减下来的片断组合在一起，又构成了具有偶然性内聚的模块。

为什么偶然性内聚如此不好？具有偶然性内聚的模块有两个严重的缺点。第一，这样的模块使产品的可维护性（纠错性维护和增强性维护）产生退化，从试图理解产品的角度看，具有偶然性内聚的模块化过程比根本不模块化更糟糕［Shneiderman and Mayer，1975］。第二，这些模块是不可重用的，上一段中提到的具有偶然性内聚的模块根本不可能在其他任何产品中重用。

不可重用是一个严重的缺点，建造软件的成本很巨大，因此要尽可能地试图重用模块。设计、编程、编写文档，尤其是测试模块是很耗费时间的，因而该过程成本很高。如果一个经过了良好设计和全面测试，并且文档齐备的模块可用于另一个产品，那么应该坚持重用该模块。但是，具有偶然性内聚的模块是不可能重用的，而且开发它所耗费的钱决不会有任何补偿。（第8章将详细讨论重用。）

通常很容易矫正具有偶然性内聚的模块——因为它执行多个操作，可将模块分成更小的模块，每个小模块执行一个操作。

### 7.2.2 逻辑性内聚

当一个模块进行一系列相关的操作，每个操作由调用模块来选择时，该模块就具有**逻辑性内聚**（logical cohesion）。下面是一些具有逻辑性内聚的例子。

**例子 1**　调用模块 new_operation 如下：

```
function_code = 7;
new_operation(function_code,dummy_1,dummy_2,dummy_3);
//dummy_1,dummy_2 和 dummy_3 是伪变量,
//如果 function_code 等于 7 则不使用它们。
```

在这个例子中，调用 new_operation 时有四个参数，但如注释行所声明的，如果 function_code 等于 7 则不使用它们中的三个参数。通常对于纠错性和增强性维护来说，这降低了可读性。

**例子 2**　一个执行所有输入和输出的对象。

**例子 3**　一个对主文件记录进行插入、删除和修改的编辑模块。

**例子 4**　OS/VS2 的早期版本中具有逻辑性内聚的一个模块进行 13 个不同的操作，它的接口包含 21 块数据 [Myers, 1978b]。




图 7-5　执行所有输入和输出操作的模块

如果模块是逻辑性内聚的，会有两个问题。第一，接口难于理解，例子 1 就是有关这方面的情形，而且这造成了模块整体上的不易理解。第二，完成多个操作的代码互相纠缠在一起，导致严重的维护问题。例如，执行所有输入和输出的模块可以如图 7-5 所示那样构造。如果安装了一个新的磁带单元，将有必要修改序号为 1、2、3、4、6、9 和 10 的各部分。这些修改可能会反过来影响输入－输出的其他形式（例如激光打印机输出），因为第 1 和第 3 部分的修改将影响激光打印机。这种纠缠的特性是具有逻辑性内聚的模块的特征，它的进一步影响是在其他产品中很难重用这样的模块。

### 7.2.3　时间性内聚

当模块执行一系列与及时性有关的操作时，该模块具有**时间性内聚**（temporal cohesion）。时间性内聚模块的例子是具有如下名称的模块：open_old_master_file，new_master_file，transaction_file 以及 print_file；initialize_sales_region_table；read_first_transaction_record_and_first_old_master_file_record（打开旧的主文件、新的主文件、事务文件以及打印文件、初始化销售地区表、读取第一个事务记录和第一个旧的主文件记录）。在 CASD 出现之前，这样的模块被称为 perform_initialization（执行初始化）。

这个模块的操作之间的关联很弱，但与其他模块的操作却有很强的关联。例如，考虑 sales_region_table（销售地区表）的操作，它在本模块中初始化，但像 update_sales_region_table（更新销售地区表）和 print_sales_region_table（打印销售地区表）这样的操作却位于其他模块。所以，如果修改了 sales_region_table 的结构（也许因为公司将业务扩展到原来没有业务的乡村地区），因而需要修改许多模块。这不仅可能会产生退化错误（由于修改产品中明显不相关的部分而引起的错误），而且如果受影响的模块数很大，

则很可能会忽略一个或两个模块。最好是与 sales _ region _ table 有关的操作都集中在一个模块中，如 7.2.7 节所述。然后在必要时由其他模块调用这些操作，另外，具有时间性内聚的模块也不太可能在另一个不同的产品中被重用。

### 7.2.4　过程性内聚

如果一个模块执行一系列与产品要遵循的步骤顺序有关的操作，则该模块具有**过程性内聚**（procedural cohesion）。例如，名称为 read _ part _ number _ from _ database _ and _ update _ repair _ record _ on _ maintenance _ file（从数据库读取零件编号并在维护文件中更新修复记录）的模块具有过程性内聚。

很明显这比时间性内聚好——至少操作之间是按过程关联的。尽管这样，操作之间仍是弱连接，而且在其他产品中重用该模块也不太可能。解决方案是把具有过程性内聚的模块分割为单独的模块，每个模块执行一个操作。

### 7.2.5　通信性内聚

如果一个模块执行一系列与产品要遵循的步骤顺序有关的操作，并且如果所有操作都在相同的数据上进行，则该模块具有**通信性内聚**（communicational cohesion）。例如，名称为 update _ record _ in _ database _ and _ write _ it _ to _ the _ audit _ trail（更新数据库中的记录并将它写入审计追踪中）的模块和名称为 calculate _ new _ trajectory _ and _ send _ it _ to _ the _ printer（计算新弹道并把它传送给打印机）的模块具有通信性内聚。因为模块中的各操作是紧密相连的，所以通信性内聚比过程性内聚更好，但它与偶然性、逻辑性、时间性和过程性内聚有相同的缺点，也就是不能重用该模块。解决方案还是将一个模块分成多个模块，每个模块执行一个操作。

顺便提一下，有趣的是，Dan Berry［personal communication，1978］使用术语**流程图内聚**来形容时间性、过程性和通信性内聚，因为这样的模块执行的操作在产品流程图中是相邻的。对于时间性内聚，因为各操作是同时执行的，因而它们是相邻的；对于过程性内聚，各操作的算法要求按顺序进行各操作，因而也是相邻的；对于通信性内聚，各操作除了按顺序执行外，均基于相同数据进行，因而也是相邻的。所以很自然这些操作在流程图中是相邻的。

### 7.2.6　功能性内聚

只执行一个操作或只达到一个单一目标的模块具有**功能性内聚**（functional cohesion）。这样的模块的例子有 get _ temperature _ of _ furnace（取炉子的温度）、compute _ orbital _ of _ electron（计算电子轨道）、write _ to _ diskette（写入磁盘）和 calculate _ sales _ commission（计算销售佣金）。

通常，具有功能性内聚的模块可重用，因为它进行的这个操作通常也在其他产品上需要。一个经过良好设计、完全测试和文档齐备的、具有功能性内聚的模块对于任何软件组织来说都是很有价值的（经济上和技术上），应尽可能地重用（参见 8.4 节）。

在具有功能性内聚的模块上进行维护将更容易。首先，功能性内聚可隔离错误。如果确定了不能正确读取炉子温度，那么模块 get _ temperature _ of _ furnace 中肯定也有这个错误。类似地，如果电子轨道计算得不正确，则首先应查看 compute _ orbital _ of _ electron 模块。

一旦在单个的模块中定位了该错误，下一步就是进行修改。因为具有功能性内聚的模块只进行一个操作，这样的模块通常比低内聚的模块更容易理解和维护。最后，修改之后，该修改对其他模块的影响将很小，特别是当模块间的耦合很低时（7.3 节）。

扩充产品功能时功能性内聚也很有价值。例如，假设一个计算机具有 10 GB 的硬盘驱动，但制造商现在希望订购具有 30 GB 硬盘驱动的功能更强大的计算机。通过浏览模块清单，维护程序员找到了名称为 write_to_hard_drive（写入硬盘驱动）的模块。很明显，需要用一个新的名称为 write_to_larger_hard_drive（写入更大的硬盘驱动）的模块来替换该模块。

顺便提一下，应该指出图 7-2 中的三个“模块”具有功能性内聚，而且 7.1 节中为帮助图 7-2 超越图 7-3 的设计所生成的参数，恰恰是前面讨论中为支持功能性内聚所生成的参数。

### 7.2.7　信息性内聚

如果模块进行许多操作，每个都有各自的入口点，每个操作的代码相对独立，而且所有操作都在相同的数据结构上完成，则该模块具有**信息性内聚**（informational cohesion）。图7-6 给出了一个这样的例子。这没有违反结构化编程的原则，每块代码都有一个入口点和一个出口点。逻辑性内聚和信息性内聚之间的主要区别是：逻辑性内聚模块的各个操作是互相纠缠的，而信息性内聚模块的各操作的代码是完全独立的。




图 7-6　具有信息性内聚的模块

具有信息性内聚的模块主要用来实现一种抽象的数据类型，如 7.5 节所述，使用具有信息性内聚的模块时，可以得到使用抽象数据类型的所有好处。因为一个对象基本上是一个抽象数据类型的一个例子（实例）（7.7 节），而对象也是一个具有信息性内聚的模块。[⊖]

### 7.2.8　内聚示例

为进一步理解内聚，考虑图 7-7 所示的例子，有两个模块特别值得讨论。你可能有点奇怪，模块 initialize_sums_and_open_files（初始化和并打开文件）和模块 close_files_and_print_average_temperatures（关闭文件并打印平均温度）都标注着具有偶然

⊖ 这一段的讨论假定抽象数据类型或对象是经过良好设计的，如果一个对象的方法执行完全不相关的操作，那么该对象具有偶然性内聚。

性内聚，而不是时间性内聚。首先，考虑模块 initialize_sums_and_open_files，它执行两个与及时性有关的操作，在进行任何计算之前这两个操作必须完成，所以看起来该模块具有时间性内聚。尽管“初始化和并打开文件”这两个操作实际上是在计算的开始阶段完成的，但这里涉及另一个因素。初始化和与该问题有关；但打开文件是一个硬件问题，与该问题本身并不相关。当可以分配两个或更多的不同级别的内聚给一个模块时，规则是分配最低级别的内聚给该模块。这样，因为 initialize_sums_and_open_files 模块既可以是时间性内聚的，也可以是偶然性内聚的，那么两个级别中的低者——偶然性内聚将分配给该模块。这也是 close_files_and_print_average_temperatures 模块具有偶然性内聚的原因。

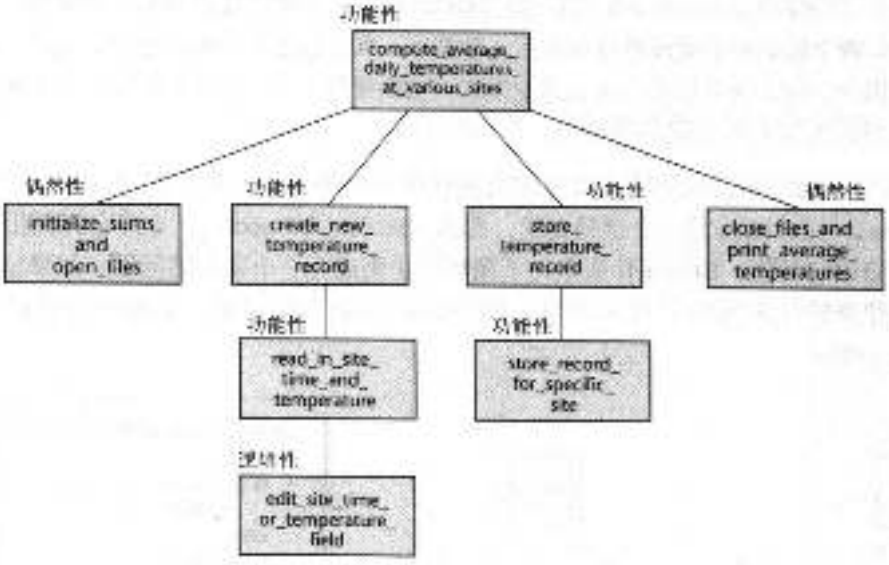



图 7-7　表示每个模块的内聚的模块层次图

## 7.3　耦合

我们还记得，内聚是模块内部的交互程度，而“耦合”是两个模块之间的交互程度。同前面一样，耦合可分为几个级别，如图 7-8 所示。为突出好的耦合，将按顺序描述各个级别，从最坏的到最好的。




图 7-8　耦合级别

### 7.3.1　内容耦合

如果两个模块中的一个直接引用了另一个模块的内容，则它们之间是**内容耦合**。以下是内容耦合的例子：

**例子 1**　模块 p 修改模块 q 的一个语句。这个练习不仅限于汇编语言编程，现在，COBOL 中已宽容地去掉了 alter 动词，该动词表示：它修改了另一个语句。

**例子 2**　根据模块 q 内部用数字表示的转移，模块 p 引用模块 q 的局部数据。

**例子3**　模块p分支转移到模块q的一个局部标号。

假设模块p和模块q之间内容耦合，许多危险之一是几乎对q的任何修改，甚至用一个新的编译器或汇编器重新编译q，也要求对p进行修改。进一步说，在一个新产品中，如果不重用模块q，则不可能重用模块p。两个模块内容耦合时，它们不可避免地相互连接在一起。

### 7.3.2　共用耦合

如果两个模块都可存取相同的全局数据，则它们之间是**共用耦合**。如图7-9所示，模块oca和ocb可以存取和修改global_variable（全局变量）的值，而不是通过传递参数来互相通信。最常见的情况是oca和ocb都存取相同的数据库，并能够读取和写入相同的记录。对于共用耦合，两个模块有必要能够读取和写入数据库，如果数据库存取状态是只读的，那么这就不是共用耦合。但实现共用耦合还有其他方式，包括使用C++或Java修饰符**public**。

这种形式的耦合不是我们想要的，原因如下：

1）它与结构化编程相矛盾，因为生成的代码完全不可读。考虑图7-10所示的伪码段，如果global_variable是一个全局变量，那么module_3、module_4或它们调用的任何模块都可能修改它的值。确定在什么条件下循环终止则成为一个重要的问题，如果出现运行时故障，将很难推测出发生了什么事情，因为这些模块中的任意一个都可能修改global_variable的值。




图7-9　共用耦合示例

```
while (global_variable == 0)
{
    if (argument_xyz > 25)
        module_3 ();
    else
        module_4 ();
}
```

图7-10　反映共用耦合的伪码段

2）考虑调用edit　this_transaction（record_7）（编辑这个事务（记录7））。如果有共用耦合，这个调用将不仅修改record_7的值，还将修改该模块能够存取的任何全局变量。简单地说，必须阅读整个模块才能准确找出它做什么。

3）如果在一个模块中对一个全局变量的声明进行了维护性修改，那么必须修改能够访问该全局变量的每一个模块。进一步说，所有的修改必须是一致的。

4）另一个问题是共用耦合的模块难于重用，因为每次重用该模块时必须提供同一个全局变量的清单。

5）共用耦合拥有令人遗憾的属性，那就是，即使模块p本身从不改变，模块p和产品中的其他模块之间共用耦合的实例数也会变化非常大。这称为**秘密共用耦合**［Schach et al., 2003a］。例如，如果模块p和模块q可以修改全局变量g_v，那么在模块p和产品中的其他模块之间有一个共用耦合的实例。但是，如果设计并实现了10个新的模块，它们都可以修改全局变量g_v，那么在模块p和产品中的其他模块之间的共用耦合的实例数增加到11个，

虽然模块 p 本身没有任何变化。秘密共用耦合会产生令人吃惊的结果，例如，在 1993 年到 2000 年间，有接近 400 次 Linux 版本发布，17 个 Linux 内核模块的 5332 个版本在后续的发布间没有改变。虽然内核模块本身没有改变，5332 个版本中有一半以上的版本，每个内核模块和其他 Linux 模块之间的共用耦合实例数增加或减少了。相当多的模块在朝上升（2482）而不是下降（379）的方向上展示了秘密共用耦合［Schach et al.，2003a］。Linux 内核中的代码行数随版本号线性增长，但共用耦合的实例数则呈指数增长［Schach et al.，2002］。看来不可避免的是，在将来的某个时候，由秘密共用耦合引起的模块间的依赖性将使 Linux 极难维护。那样将非常难以在产品中其他部分不引起退化错误（一个显然不相关的错误）的情况下，改变 Linux 的某一部分。

6）这个问题的潜在危险更大。作为共用耦合的结果，模块会比需要的暴露出更多的数据，这使得试图控制数据存取的努力付之东流，而且完全会导致计算机犯罪。许多类型的计算机犯罪需要一些形式的共谋。设计良好的软件不应允许一个程序员能够访问犯罪所需的所有数据或模块。例如，写出工资报表产品的校验打印部分的程序员需要访问雇员记录，但在经过良好设计的产品中，这样的访问应严格控制在只读状态下，以防止程序员对他或她的月工资进行非授权的修改。若程序员还想做这样的修改，则必须找到另一个不诚实的雇员，他能在更新模式下访问相关的记录。但如果产品设计得不好，每个模块都可在更新状态下访问工资报表数据库，那么毫无道德的程序员则可单独未经授权就修改数据库的记录。

尽管前面的讨论意在说服所有读者（最大胆的读者除外）避免使用共用耦合，但有些情况下共用耦合看起来更好。例如，考虑完成储油罐计算机辅助设计的产品［Schach and Stevens-Guille，1979］。一个罐可由大量的描述信息来确定，例如高度、直径、储油罐可承受的最大风速和绝缘层厚度。需要初始化这些描述信息，但其后不能修改这些值，而且产品中的大部分模块需要访问描述信息的这些值。假设有 55 个油罐描述信息，如果所有这些描述信息作为每个模块的参数进行传递，那么每个模块的接口都将包含至少 55 个参数，而且潜在的错误也很巨大。即使在像 Ada 这样的要求对参数进行严格类型检查的语言中，仍可能颠倒互换了相同类型的两个参数，而类型检查器将检测不到这样的错误。

一个解决方案是把所有的油罐描述信息放入数据库中，并按下面的方式设计产品：一个模块初始化所有描述信息的值，而所有其他的模块严格地在只读状态下访问该数据库。然而，如果数据库的解决方案不切实际，也许因为特定的实现语言不能与可用的数据库管理系统进行接口，那么另一个办法是在受控状态下使用共用耦合。也就是说，产品应设计成一个模块初始化 55 个描述信息，而其他的所有模块不能修改描述信息的值。这种编程风格必须严格管理，不像数据库解决方案由软件进行这种严格控制。所以，在没有比使用共用耦合更好的方案时，经过严格的管理可以减小一些风险。然而，还有一个更好的解决方案是通过使用信息隐藏来避免共用耦合，如 7.6 节所述。

### 7.3.3　控制耦合

如果两个模块中的一个模块给另一个模块传递控制要素，则它们具有**控制耦合**，也就是说，一个模块明确地控制另一个模块的逻辑。例如，当给具有逻辑性内聚的模块（7.2.2 节）传递一个函数代码时就传递了一个控制。控制耦合的另一个例子是控制开关作为一个参

数进行传递时的情况。

如果模块 p 调用模块 q，而模块 q 给模块 p 传回一个标志，表明“不能完成任务”，那么模块 q 就是在传递数据。但如果该标志是“不能完成任务，因而写出错误消息 ABC123”，那么模块 p 和模块 q 是控制耦合。换句话说，如果模块 q 给模块 p 传回信息，而且模块 p 决定收到信息后进行什么操作，那么 q 在传递数据。但如果模块 q 不仅传回信息，还传回模块 p 应执行什么操作的指示，那么二者之间存在控制耦合。

控制耦合的一个主要难点是两个模块是非独立的，被调用的模块（模块 q）需要知道模块 p 的内部结构和逻辑，因此降低了重用的可能性。另外，通常控制耦合与具有逻辑性内聚的模块有关联，因而也包含了与逻辑性内聚有关的困难。

### 7.3.4　印记耦合

在一些编程语言中，只有像 part _ number（零件编号）、satellite _ altitude（卫星高度）或 degree _ of _ multiprogramming（多道程序设计的程度）这样的简单变量可作为参数进行传递。但许多语言还支持传递数据结构，例如把记录或数组作为参数。在这样的语言中，可用的参数包括 part _ record（零件记录）、satellite _ coordinates（卫星坐标）或 segment _ table（程序段表）。如果把数据结构作为参数进行传递，两个模块是**印记耦合**，但被调用的模块只在该数据结构的一些个别组件上进行操作。

例如，考虑调用 calculate _ withholding（employee _ record）（计算扣除的工资（雇员记录））。不阅读整个 calculate _ withholding 不能明确该模块访问或修改的是 employee _ record 的哪个字段。传递雇员的工资显然对于计算工资扣除是必需的，但很难明白为何需要雇员的家庭电话号码。只有计算扣除的工资真正需要的那些字段应传递给模块 calculate _ withholding。不仅是得到的模块，特别是它的接口，很容易理解的是，在各种其他也需要计算扣除工资的产品中，很可能都可以重用该模块和接口。（关于此问题的另一个观点请参见下面的“如果你想知道”部分。）

**如果你想知道**

传递给模块 4～5 个不同的字段将比传递整个记录慢，这导致一个更大的问题：当最优化问题（例如响应时间或空间限制）与通常认为是好的软件工程实践冲突时应怎么办？

以我的经验，解决这个问题是无意义的。使用推荐的方法可以降低响应时间，但只有 1 ms 左右，用户不能感觉得到。所以，根据 Don Knuth［1974］的最优化第一定律：不要！——几乎没有任何类型的最优化，包括由于性能方面的原因。

但如果真需要最优化时怎么办？在这种情况下，可应用 Don Knuth 的最优化第二定律。第二定律（只对专家）是：还没有！换句话说，首先使用合适的软件工程技术完成整个产品，然后，如果真需要最优化，只做必要的修改。仔细用文字描述要修改什么及为什么。如果有可能，应该由经验丰富的软件工程师进行这个最优化工作。

可能更重要的是，因为上述的调用 calculate _ withholding（employee _ record）比需要的传递了更多的数据，对数据访问无法控制的问题和接踵而来的计算机犯罪将再一次地出现，7.3.2 节讨论了这个问题。

假若数据结构的全部组件由被调用模块使用，把数据结构作为参数进行传递根本没有

错。例如，像 invert_matrix (original_matrix, inverted_matrix) 或 print_inventory_record (warehouse_record) 这样的调用把数据结构作为参数进行传递，但被调用的模块在数据结构的所有组件上进行操作。当把数据结构作为一个参数进行传递，但被调用的模块只使用一部分组件时就出现了印记耦合。

在像 C 或 C++ 这样的语言中，把指向记录的指针作为参数传递时会出现一种微妙形式的印记耦合。考虑调用 check_altitude (pointer_to_position_record)（检查高度（位置记录的指针））。乍一看以为正在传递的是一个简单的变量，但被调用的模块可访问由 pointer_to_position_record 指向的 position_record 中的所有字段。由于这个潜在的问题，建议无论何时把指针作为参数进行传递，都要仔细检查该耦合。

### 7.3.5 数据耦合

如果两模块的所有参数是同类数据项，则它们具有**数据耦合**。也就是说，每个参数或者是简单参数，或者是数据结构（其中的所有元素为被调用的模块所使用）。数据耦合的例子有 display_time_of_arrival (flight_number)、compute_product (first_number, second_number, result) 和 determine_job_with_highest_priority (job_queue)。

数据耦合是理想的目标。首先从反面想，如果一个产品只显示出数据耦合，那么内容耦合、共用耦合、控制耦合和印记耦合的那些困难将不会存在。从更积极的角度看，如果两个模块是数据耦合的，那么维护会很容易，因为对一个模块的修改几乎不会使另一个模块产生连带的错误。下面的例子阐明了耦合的某些特征。

### 7.3.6 耦合示例

考虑图 7-11 所示的例子。连线上的数字代表接口，它在表 7-1 中有详细的定义。例如，当模块 p 调用模块 q 时（接口 1），它传递了一个参数——飞机类型。当 q 返回控制给 p 时，它传回了一个状态标志。使用图 7-11 和表 7-1 中的信息，每对模块之间的耦合可以推导出来，结果如表 7-2 所示。

表 7-2 中的一些项是显然的，例如，模块 p 和 q 之间的数据耦合（图 7-11 中的接口 1）、模块 r 和 t 之间的数据耦合（接口 5）及模块 s 和 u 之间的数据耦合（接口 6）是在每个方向上传递一个简单变量的直接结果。如果使用或更新从模块 p 传递给模块 s 的零件清单的所有元素，那么模块 p 和 s 之间的耦合（接口 2）是数据耦合，但如果模块 s 只对该清单中的某个元素进行操作，则模块 p 和 s 之间的耦合是印记耦合。模块 q 和 s 之间的耦合（接口 4）情况类似。因为图 7-11 和表 7-1 中的信息不能完全描述这各个模块的功能，所以没有办法确定该耦合是数据耦合还是印记耦合。模块 q 和 r 之间（接口 3）是控制耦合，因为从模块 q 传递给 r 的是一个功能代码。

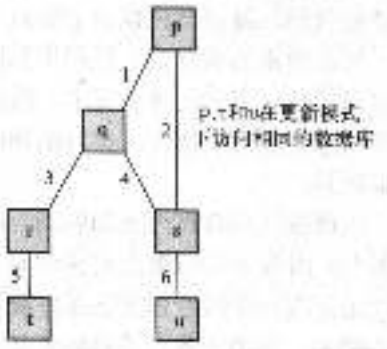



图 7-11 耦合例子中的模块互连图

表 7-1　图 7-11 的接口描述

| 号码 | 输入 | 输出 |
|---|---|---|
| 1 | 飞机类型 | 状态标志 |
| 2 | 飞机的零件清单 | — |
| 3 | 功能代码 | — |
| 4 | 飞机的零件清单 | — |
| 5 | 零件编号 | 零件制造商 |
| 6 | 零件编号 | 零件名称 |

表 7-2　图 7-11 中各对模块间的耦合

| | q | r | s | t | u |
|---|---|---|---|---|---|
| p | 数据 | — | 数据或印记 | 共用 | 共用 |
| q | | 控制 | 数据或印记 | — | — |
| r | | | — | 数据 | — |
| s | | | | — | 数据 |
| t | | | | | 共用 |

有点奇怪的是，表 7-2 中标明有三个共用耦合。这三个模块对是图 7-11 中最远的连接——模块p 和 t、模块 p 和 u 以及模块 t 和 u——开始好像它们之间并没有任何耦合。毕竟它们之间没有接口，所以有必要解释为什么它们之间具有耦合，更不要说是共用耦合了。答案在图 7-11 的右侧有所暗示，即模块 p、t 和 u 在更新模式下都可访问相同的数据库。结果所有这三个模块均可修改一些全局变量，因而它们两两之间具有共用耦合。

### 7.3.7　耦合的重要性

耦合是一个重要的度量。如果模块 p 与模块 q 之间耦合紧密，那么对模块 p 进行了修改，也要求对模块 q 进行相应的修改。如果在集成或交付后维护阶段按要求进行了修改，那么产品的功能将是正确的，然而那个阶段的进展将比松耦合的情况要慢。另一方面，如果当时没有对模块 q 进行相应的修改，那么错误会在晚些时候自行表现出来。在最好的情况下，编译器或链接器测试出模块 p 的这个修改时，将立即提醒小组漏掉了什么东西或将会出现故障。然而通常的情况是，在接下来的集成测试或产品已经安装到客户的计算机上之后，产品才出现故障。在这两种情况下，故障是在完成对模块 p 的修改之后出现的，这时，对模块 p 进行的修改和对模块 q 应进行的相应修改（但没有进行）之间不再有明显的联系，所以错误很难找到。

假设设计具有高内聚和低耦合的模块是一个好设计，那么明显的问题是，这样的设计如何实现？因为本章的重点是围绕设计讨论理论概念，所以此问题的答案在第 13 章中讨论，同时还会进一步讨论和细化良好设计的那些品质。为方便起见，本章中出现的关键定义如图 7-12 所示，其中还提示了这些定义所在的章节。

抽象数据类型：一个数据类型连同在那个数据类型的实例上进行的操作（7.5节）
抽象：通过抑制不必要的细节并强调相关的细节达到理想要求结果的一种方法（7.4.1节）
类：支持继承的一种抽象数据类型（7.7节）
内聚：模块内部的交互程度（7.1节）
耦合：两个模块之间的交互程度（7.1节）
数据封装：一个数据结构连同在该数据结构上进行的操作（7.4节）
封装：把真实世界中的实体的各方面集中在一个对该实体建模的单元中（7.4.1节）
信息隐蔽：建立一个设计，使得到的实现细节对其他模块是隐藏的（7.6节）
对象：类的一个实例（7.7节）

图 7-12　本章的关键定义及所在的章节

## 7.4　数据封装

考虑为一个大型计算机设计操作系统的问题。根据规格说明，任何提交给计算机的作业都分类为高优先级、中优先级或低优先级。操作系统的任务是决定下一个载入内存的是哪一个作业，内存中的那一个作业得到下一个时间片，时间片的长度是多少，以及哪个作业要求磁盘访问的优先级最高。在执行这些任务中，操作系统必须考虑每个作业的优先级，优先级越高，把计算机资源分配给该作业就越快。实现这一点的方式是取得按作业优先级排列的一个独立的作业队列。需要初始化该作业队列，并且能够在作业要求内存、CPU 时间或磁盘访问时，添加一项作业到作业队列中。当操作系统决定给某作业分配它所要求的资源时，从队列中删除该作业。

为简化问题，考虑批作业在排队等候内存访问的问题。进来的批作业有三个队列，每个队列对应一个优先级。当用户提交作业时，将在合适的队列中增加一项作业，当操作系统决定已准备好运行某作业时，该作业将从它的队列中退出，并得到分配的内存。

产品的这个部分可以用许多方式建造。一个可能的设计如图 7-13 所示，描绘了操作三个作业队列中的一个队列的模块。一个类似 C 的伪码用来突出显示这个传统范型中可能出现的一些问题。本章的后面将应用面向对象范型来解决这些问题。

考虑图 7-13，模块 m_1 中的函数 initialize_job_queue 负责初始化作业队列，而模块 m_2 和模块 m_3 中的函数 add_job_to_queue 和 remove_job_from_queue 分别负责增加和删除作业。模块 m_123 包含对所有这三个函数的调用，以便操作该作业队列。为集中精力于数据封装上，像下溢（试图从空队列中删除作业）和上溢（试图在已满的队列中增加作业）这样的事项在这里不讨论，它们将在本章的其余部分讨论。

图 7-13 的设计中的模块具有低内聚，因为作业队列方面的操作在整个产品中分散开来。如果决定修改实现 job_queue 的方式（例如以链接的记录列表实现，而不是以线性的记录列表实现），那么必须彻底地修改模块 m_1、m_2 和 m_3。还必须修改模块 m_123，至少要修改数据结构定义。

现在假设选择了图 7-14 的设计，图中右边的模块具有信息性内聚（7.2.7 节），它在相同的数据结构上执行许多操作。每个操作都有自己的入口点、出口点及独立的代码。图 7-14 中的模块 m_encapsulation 是数据封装的一种实现，也就是说，在这种作业队列的情况

中，一个数据结构中含有在这个数据结构之上执行的操作。

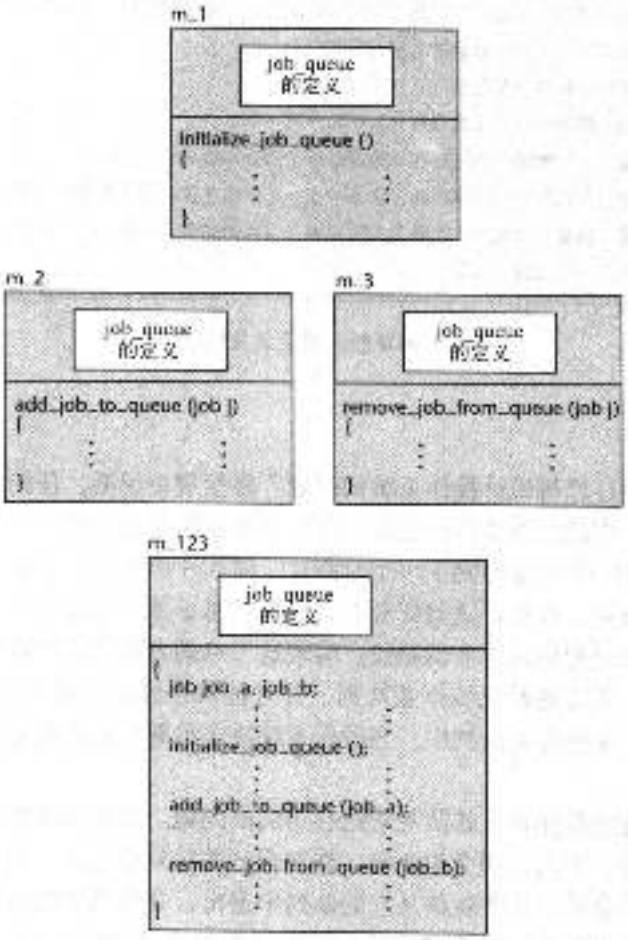



图 7-13　操作系统中作业队列部分的一种设计

此时一个明显的问题是，使用数据封装设计产品的好处是什么？下面将从开发的角度和维护的角度两方面回答这个问题。

### 7.4.1　数据封装和产品开发

数据封装是**抽象**的一个例子。还是回到作业队列的例子，已经定义了一个数据结构（作业队列），带有三个相关的操作（初始化作业队列、增加作业到队列和从队列删除作业）。开发者可以在更高的层次（作业级别和作业队列）上构思这个问题，而不是在记录或数组这样的低层次上构思。

抽象背后的基本理论概念仍然是逐步求精法。首先，产品的设计以高层概念（例如作业、作业队列和在作业队列上执行的操作）为基础，在这个阶段，如何实现作业队列与设计毫不相关。一旦得到了完全高层次的设计，第二步是设计低层次的组件，即数据结构和在这个数据结构上将要实现的操作。例如在 C 中，数据结构（作业队列）将以记录（结构）或数组的形式实现，三个操作（初始化作业队列、增加作业到队列和从队列删除作业）将以函

数形式实现。关键点是设计低层次时，设计者完全忽视了作业、作业队列和操作的扩展用途，所以在第一步里，假定了低层次的存在（即使还并未考虑该层次）；而在第二步（低层次的设计），忽视了高层次的存在。在高层次里，主要关心的是数据结构（即作业队列）的行为；在低层次里，主要考虑这些行为的实现。当然，更大的产品将有许多层次的抽象。

抽象有许多不同的类型，考虑图 7-14，图中有两种类型的抽象。数据封装（也就是数据结构连同在该数据结构上进行的操作）是**数据抽象**的一个例子；C 函数本身是**过程抽象**的一个例子。概括地说，“抽象”只是通过抑制不必要的细节并强调有关的细节达到逐步求精的一种方法。现在可以定义**封装**为把真实世界中的实体的各方面集中在一个对该实体进行建模的单元中，在 1.9 节中把这称之为概念上独立。

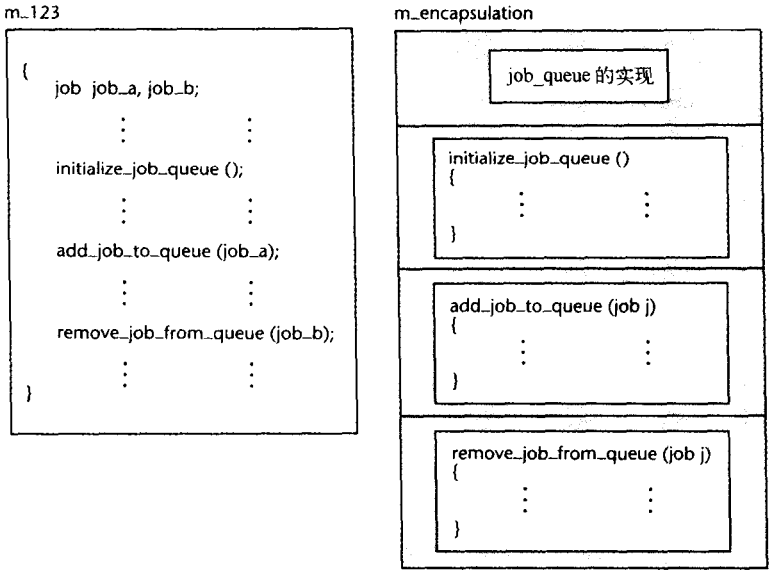



图 7-14　使用数据封装的操作系统中作业队列部分的设计

数据抽象允许设计者在数据结构和在其上进行的操作的层次上思考问题，并且随后才考虑如何实现数据结构和这些细节。现在转向过程抽象，考虑定义一个 C 函数 initialize_job_queue 的结果。这么做的效果是通过给开发者提供另一个函数来扩充语言，该函数不是这种语言原来定义过的部分。开发者可以像使用 sqrt 或 abs 一样来使用 initialize_job_queue。

过程抽象与数据抽象一样对设计意义重大。设计者可以从高层次行为上构思产品，这些行为以低层次操作的形式来定义，直到达到最低的层次。在这个最低的层次上，操作以编程语言中预先定义好的结构来表达。在每个层次上，设计者都只考虑以与本层次相符的操作来表达产品，还可以忽视下面的层次，因为它们将在下一个抽象层次中得到处理，也就是在下一个求精步骤中得到处理。设计者也可以忽视上面的层次，因为上面的层次与设计当前层次不相关。

### 7.4.2 数据封装和产品维护

从维护的角度考虑数据封装，基本的问题是识别产品可能需要改变的方面并且设计产品，以使将来的改变的影响最小化。像这样的数据结构不太可能改变，例如，如果产品包含作业队列，那么未来的版本将很可能结合它们，同时，实现作业队列的特定的方式很可能会改变，而数据封装提供了适应改变的一种方法。

图 7-15 显示了以 C++ 实现的作业队列数据结构，即类 **JobQueue**，图 7-16 是对应的 Java

```
//
// 警告：
// 本代码的形式适合不太精通C++的读者阅读，没有使用良好的C++
// 风格。还有为简化起见，忽略了像对上溢和下溢进行检查这样的基
// 本特性，详见下面的“如果你想知道”部分。
//
class JobQueue
{
  // 属性
  public:
    int queueLength;        // 作业队列的长度
    int queue[25];          // 队列可包含量多25个作业

  // 方法
  public:
    void initializeJobQueue ()
    /*
     * 空的作业队列的长度为0
     */
    {
       queueLength = 0;
    }

    void addJobToQueue (int jobNumber)
    /*
     * 在作业队列的最后增加作业
     */
    {
       queue[queueLength] = jobNumber;
       queueLength = queueLength + 1;
    }

    int removeJobFromQueue ()
    /*
     * 设置jobNumber等于存储在队列头的那个作业号，
     * 在作业队列头删除那个作业，上移剩余的作业，
     * 并返回jobNumber。
     */
    {
       int jobNumber = queue[0];
       queueLength = queueLength - 1;
       for (int k = 0; k < queueLength; k++)
          queue[k] = queue[k + 1];
       return jobNumber;
    }
}// class JobQueue
```

图 7-15　类 JobQueue 的 C++ 实现（由 public 属性引起的问题将在 7.7 节中解决）

实现。(下面的“如果你想知道”部分中包含有介绍图 7-15 和图 7-16 以及本章后续的代码例子的编程风格的内容)。在图 7-15 或图 7-16 中，队列以最多到 25 个作业号的数组实现，第 1 个元素是 queue [0]，第 25 个元素是 queue [24]。每个作业号以一个整数来表示。保留字 **public** 允许 queueLength 和 queue 在操作系统中的任何地方都是可见的。这样得到的共用耦合相当不实用，将在 7.6 节中得以修正。

```
//
// 警告：
//本代码的形式适合不太精通Java的读者阅读，没有使用良好的Java
//风格。还有为简化起见，忽略了像对上溢和下溢进行检查这样的基
//本特性，详见下面的“如果你想知道”部分。
//
class JobQueue
{
   // 属性
   public int     queueLength;              // 作业队列的长度
   public int     queue[ ] = new int[25];   // 队列可包含最多25个作业

   //方法
   public void initializeJobQueue ()
   /*
    * 空的作业队列的长度为0
    */
   {
      queueLength = 0;
   }



   public void addJobToQueue (int jobNumber)
   /*
    * 在作业队列的最后增加作业
    */
   {
      queue[queueLength] = jobNumber;
      queueLength = queueLength + 1;
   }


   public int removeJobFromQueue ()
   /*
    * 设置jobNumber等于存储在队列头的那个作业号，
    * 在作业队列头删除那个作业，上移剩余的作业，
    * 并返回jobNumber。
    */
   {
      int jobNumber = queue[0];
      queueLength = queueLength - 1;
      for (int k = 0; k < queueLength; k++)
         queue[k] = queue[k + 1];
      return jobNumber;
   }
}// class JobQueue
```

图 7-16　类 **JobQueue** 的 Java 实现（由 **public** 属性引起的问题将在 7.7 节中解决）

**如果你想知道**

我特意采用突出数据抽象事项的方式，以好的编程实践为代价编写了图 7-15 和图 7-16 以及其后的代码例子。例如，图 7-15 和图 7-16 中 JobQueue 的定义里的数值 25 当然应该作为参数来编码，也就是在 C++ 中作为一个 const 或在 Java 中作为一个 public static final 变量。还有为简便起见，我忽略了诸如下溢（试图从一个空队列中删除记录项）或上溢（试图向一个满队列中增加记录项）这样的条件检查。在任何实际的产品中，包含这样的检查非常重要。

另外，还最小化了语言特有的特性。通常一个 C++ 程序员把 queueLength 的值增加 1 写成：

```
queueLength++;
```

而不是写成：

```
queueLength = queueLength + 1;
```

类似地，还最小程度地使用构造器和析构器。

概括地说，我只是从教学的角度写出这一章的代码，不应将这些代码用于任何其他目的。

因为它们是 **public**（公有）的，所以操作系统中的任何地方都可调用类 **JobQueue** 中的方法。特别地，图 7-17 显示了在 C++ 中，queueHandler 方法如何使用类 **JobQueue**，而图 7-18 是对应的 Java 实现。queueHandler 方法调用了 **JobQueue** 中的 initializeJobQueue、addJobToQueue 和 removeJobFromQueue 方法，而不需要知道这个作业队列是如何实现的。使用类 **JobQueue** 惟一需要知道的信息是与这三个方法相关的接口信息。

```
class Scheduler
{
  ...
  public:
    void queueHandler ()
    {
      int              jobA, jobB;
      JobQueue         jobQueueJ;

        // 各种语句
      jobQueueJ.initializeJobQueue ();
        // 更多语句
      jobQueueJ.addJobToQueue (jobA);
        // 仍是更多语句
      jobB = jobQueueJ.removeJobFromQueue ();
        // 进一步的语句
    }// queueHandler
  ...
}// class Scheduler
```

图 7-17　queueHandler 的 C++ 实现

现在假设作业队列以作业号的线性列表形式得以实现，但之后需要将它作为作业记录的双向链表重新实现。每个作业记录将有三个组件：与前面所述相同的作业号、在链表中位于该作业记录之前的作业记录指针，以及位于作业记录之后的作业记录指针。图 7-19 给出了

用 C++ 实现的代码，而图 7-20 给出了用 Java 实现的代码。现在，为适应实现作业队列方式的调整，需要对整个软件产品进行什么样的修改？事实上，只有 **JobQueue** 本身需要修改。图 7-21 给出了图 7-19 使用双向链表以 C++ 实现 **JobQueue** 的大概轮廓，省略了实现的细节，以突出表示出 **JobQueue** 和产品其他部分（包括 queueHandler 方法）之间的接口并没有任何改变（参见习题 7.12）。也就是说，调用三个方法 initializeJobQueue、addJobToQueue 和 removeJobFromQueue 的方式没有改变，特别是调用 addJobToQueue 方法时，它仍给 **JobQueue** 传递一个整数值，而 removeJobFromQueue 方法仍旧从 **JobQueue** 返回一个整数值，尽管作业队列本身的实现方式完全改变了。结果，queueHandler 方法的源代码（图 7-17）根本不需要修改。由此，数据封装以简化产品维护的方式支持了数据抽象的实现，从而减小了出现退化错误的可能性。

```
class Scheduler
{
   ...
   public void queueHandler ()
   {
        int              jobA, jobB;
        JobQueue         jobQueueJ = new JobQueue ();

           // 各种语句
        jobQueueJ.initializeJobQueue ();
           // 更多语句
        jobQueueJ.addJobToQueue (jobA);
           // 仍是更多语句
        jobB = jobQueueJ.removeJobFromQueue ();
           // 进一步的语句
   }// queueHandler
   ...
}// class Scheduler
```

图 7-18　queueHandler 的 Java 实现

```
class JobRecord
{
  public:
      int              jobNo;          // 作业号（整数）
      JobRecord        *inFront;       // 在作业记录前面的指针
      JobRecord        *inRear;        // 在作业记录后面的指针
}// class JobRecord
```

图 7-19　双向链接的类 **JobRecord** 的 C++ 实现（由 **public** 属性引起的问题将在 7.6 节中解决）

```
class JobRecord
{
   public int              jobNo;      // 作业号（整数）
   public JobRecord        inFront;    // 在作业记录前面的指针
   public JobRecord        inRear;     // 在作业记录后面的指针
} // class JobRecord
```

图 7-20　双向链接的类 **JobRecord** 的 Java 实现（由 **public** 属性引起的问题将在 7.6 节中解决）

```
class JobQueue
{
  public:
    JobRecord      *frontOfQueue;      // 队列前面的指针
    JobRecord      *rearOfQueue;       // 队列后面的指针

    void initializeJobQueue ()
    {
      /*
       * 通过把frontOfQueue和rearOfQueue设置为NULL（空）来初始化作业队列
       */
    }

    void addJobToQueue (int JobNumber)
    {
      /*
       * 创建新的工作记录，把jobNumber 放到它的jobNo字段中，
       * 设置inFront字段指向当前的rearOfQueue (因而将新记录链接到队列的后面)，
       * 并把inRear字段设置为NULL(空)。
       * 设置当前的rearOfQueue指向的记录的inRear字段指向新的记录
       * （因而建立了一个双向链接），
       * 最后设置rearOfQueue指向这个新记录。
       */
    }

    int removeJobFromQueue ()
    {
      /*
       * 设置jobNumber等于队列前面的那个记录的jobNo字段，
       * 更新frontOfQueue指向队列中的下一项，
       * 把现在成为队列头的记录的inFront字段设置为NULL（空），
       * 并返回jobNumber。
       */
    }
}// class JobQueue
```

图 7-21　使用双向链表的类 JobQueue 的 C++ 实现的轮廓

对比图 7-15 和图 7-16 与图 7-17 和图 7-18，很明显在这些实例中，C++ 和 Java 实现之间的区别主要是语法上的区别。在本章的其余部分，我们只给出一种实现，并会说明在另一种实现中语法上的差异。特别地，这个作业队列代码的其余部分以 C++ 写成，而所有其他的代码例子则由 Java 写成。

## 7.5　抽象数据类型

图 7-15（或图 7-16）是作业队列类的一个实现，也就是说，一个数据类型连同在该数据类型的实例上进行的操作。这样的构造称为**抽象数据类型**。

图 7-22 显示了如何用 C++ 语言实现这种抽象数据类型，用于操作系统的三个作业队列中。这三个作业队列用具体的例子来说明：highPriorityQueue、mediumPriorityQueue 和 lowPriorityQueue。（Java 版本只是在三个作业队列的数据声明的语法上有所不同。）语句 highPriorityQueue.initializeJobQueue（）的意思是“把 initializeJobQueue 方法应用到数据结构 highPriorityQueue 中”，另两个语句与之类似。

```
class Scheduler
{
  ...
  public:
    void queueHandler ()
    {
      int              job1, job2;
      JobQueue         highPriorityQueue;
      JobQueue         mediumPriorityQueue;
      JobQueue         lowPriorityQueue;

        // 一些语句
      highPriorityQueue.initializeJobQueue ();
        // 更多的一些语句
      mediumPriorityQueue.addJobToQueue (job1);
        // 仍是更多的语句
      job2 = lowPriorityQueue.removeJobFromQueue ();
        // 甚至更多的语句
    }// queueHandler
  ...
}// class Scheduler
```

图 7-22　使用图 7-15 的抽象数据类型实现的 C++ 方法 queueHandler

抽象数据类型是一个有广泛用途的设计工具，例如，假设一个产品有许多操作需要在有理数上完成，有理数即可用 n/d 的形式代表的数，其中 n 和 d 是整数，d≠0。可以有许多方式来表示有理数，例如一个一维整型数组的两个元素或一个类的两个属性。为了以抽象数据类型的形式实现有理数，可以为这个数据结构选择一个合适的表示法。在 Java 中，可以如图 7-23 所示来定义它，带有在有理数上完成的各种操作，例如由两个整数创建一个有理数、两个有理数相加或两个有理数相乘。（由图 7-23 中诸如 numerator 和 denominator 这样的 **public** 属性引起的问题将在 7.6 节中得到修正。）对应的 C++ 实现的不同之处是保留字 **public** 的位置不同，还有，通过引用传递参数时需要 & 符号。

```
class Rational
{
  public int       numerator;
  public int       denominator;

  public void sameDenominator (Rational r, Rational s)
  {
    // 用相同的分母约分r和s的代码
  }

  public boolean equal (Rational t, Rational u)
  {
    Rational       v, w;
    v = t;
    w = u;
    sameDenominator (v, w);
    return (v.numerator == w.numerator);
  }

  // 加、减、乘、除两个有理数的方法

}// class Rational
```

图 7-23　有理数的 Java 抽象数据类型的实现（由 **public** 属性引起的问题在 7.6 节中解决）

抽象数据类型支持数据抽象和过程抽象（7.4.1 节），另外，当修改产品时，通常不会修改抽象数据类型，最坏的情况下，可能需要增加额外的操作到抽象数据类型中。所以，从产品开发和产品维护两个角度看，抽象数据类型是对软件制造者很有吸引力的一个工具。

## 7.6 信息隐藏

7.4.1 节中所讨论的两种类型的抽象（数据抽象和过程抽象）依次是 Parnas 提出的更通用的设计概念**信息隐藏**的例子［Parnas，1971，1972a，1972b］。Parnas 的意图是面向未来的维护。在设计产品之前，应列出一个未来可能修改的实现决定的清单。然后设计模块，对其他模块隐藏本模块设计的实现细节。这样，每个未来的修改都可以定位到一个特定的模块。因为原来的实现决定的细节对其他模块不可见，那么对该设计进行修改显然不会影响其他任何模块。（需进一步了解信息隐藏，请参见下面的“如果你想知道”部分。）

**如果你想知道**

*术语“信息隐藏”有点用词不当，更准确的描述应是“细节隐藏”。因为隐藏的不是信息，而是实现的细节。*

为了明白这些思想如何在实际中得到应用，请考虑图 7-22，它使用了图 7-15 中的抽象数据类型实现。使用抽象数据类型的一个主要原因是，确保只有调用图 7-15 中的三个方法之一才能修改作业队列的内容。遗憾的是，前面所述的实现可以通过其他途径来修改作业队列。图 7-15 中的属性 queueLength 和 queue 都声明为 **public** 的，因此在 queueHandler 内部即可访问它们。结果，在图 7-22 中 queueHandler 的任何地方，使用如下非常合法的 C++（或 Java）赋值语句可以修改 highPriorityQueue：

```
highPriorityQueue.queue [7] =-5678;
```

换句话说，不使用抽象数据类型的三个操作也能修改作业队列的内容。除了需要降低内聚和提高耦合之外，管理者必须意识到该产品易受计算机犯罪的攻击，如 7.3.2 节所述。

所幸有一个摆脱困境的办法，C++ 和 Java 的设计者在类的规格说明内部提供了信息隐藏。C++ 的情况如图 7-24 所示（Java 的语法上的区别如前所述）。除了将属性由 **public** 改变为 **private** 调整了可见性之外，图 7-24 与图 7-15 是一样的。现在对其他模块来说，惟一可见的是 **JobQueue** 是一个类，以及可以对这个生成的作业队列进行操作的带有特定接口的三个操作。但实现作业队列的真正方式是 **private** 的，也就是说，对外部是不可见的。图 7-25 显示了带有 **private** 属性的类如何能使 C++ 或 Java 用户实现完全隐藏信息的抽象数据类型。

信息隐藏技术还可用于防止共用耦合，如 7.3.2 节最后所提到的。再次考虑那一节中描述的产品，一个储油罐的计算机辅助设计工具，有 55 个描述符对它进行规范。如果该产品的实现方式是，用 **private** 操作初始化描述符，而用 **public** 操作取得一个描述符的值，那么就不会有共用耦合。这种解决方案是面向对象范型的特征，因为下一节将讲到，对象支持信息隐藏，这是使用对象技术的另一个好处。

```
class jobQueue
{
  // 属性
  private:
    int     queueLength;      // 作业队列的长度
    int     queue[25];        // 队列可包含最多25个作业


  // 方法
  public:
    void initializeJobQueue ()
    {
      //与图7-15中对应位置的方法体相同，没有修改
    }


    void addJobToQueue (int jobNumber)
    {
      //与图7-15中对应位置的方法体相同，没有修改
    }


    int removeJobFromQueue ()
    {
      //与图7-15中对应位置的方法体相同，没有修改
    }
}// class jobQueue
```

图 7-24　具有信息隐藏的 C++ 抽象数据类型实现，解决了图 7-15、图 7-16、图 7-19、图 7-20 和图 7-23 中的问题

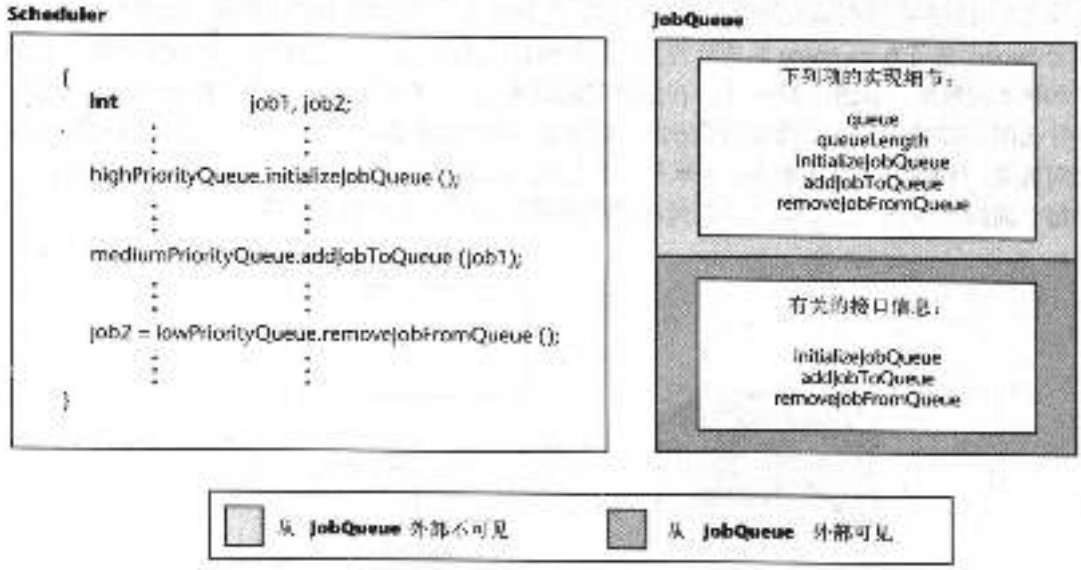



图 7-25　通过 private 属性实现信息隐藏的抽象数据类型的图示法（图 7-24 与图 7-22）

## 7.7　对象

本章开始时已讲过，对象只是图 7-26 所示的发展的下一步骤。关于对象没有什么特别的，它们与抽象数据类型或带有信息性内聚的模块一样普通。对象的重要性在于它们具有图 7-26 中它们的前辈所拥有的所有特性，还有它们自己的一些额外特性。

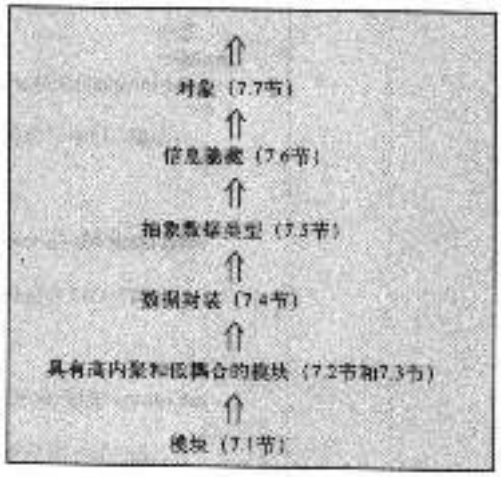


图 7-26　第 7 章的主要概念及其所在章节

"对象"的一个不完全的定义是，对象是抽象数据类型的一个具体例子（实例）。也就是说，产品根据抽象数据类型进行设计，产品的变量（对象）是抽象数据类型的实例。但用抽象数据类型的实例来定义一个对象太简单化了，还需要更多的一些东西，那就是**继承**（inheritance），它是最早在 Simula 67 中引入的一个概念［Dahl and Nygaard，1966］。所有的面向对象编程语言都支持继承，例如 Smalltalk［Goldberg and Robson，1989］、C++［Stroustrup，2000］和 Java［Flanagan，2002］。继承背后的基本概念是新的数据类型可定义为先前定义过的类型的扩展，而不是从头开始定义［Meyer，1986］。

在面向对象的语言中，可把类定义为支持继承的抽象数据类型，那么对象就是类的实例。为了明白如何使用类，考虑下面的例子。定义 **HumanBeing** 为一个类，Joe 是一个对象，它是那个类的一个实例。每个 **HumanBeing** 都有诸如年龄和身高这样的某些属性，以及描述对象 Joe 时分配给那些属性的值。现在假设定义 **Parent** 为 **HumanBeing** 的**子类**（或派生类），这意味着 **Parent** 具备 **HumanBeing** 的所有特性，另外还有他或她自己的一些属性，例如最大的孩子的姓名和孩子的数量，如图 7-27 所示。在面向对象的术语中，**Parent** "是一个" **HumanBeing**，这就是为什么图 7-27 中的箭头好像标错了方向。事实上，那个箭头表示"是一个"关系，因此从派生类指向基类。（使用开放箭头来表示继承是一个 UML 惯例，另一个惯例是类名称以粗体字表示，其中每个词的第一个字母大写。UML 将在第 2 部分特别是第 16 章详细讨论。）

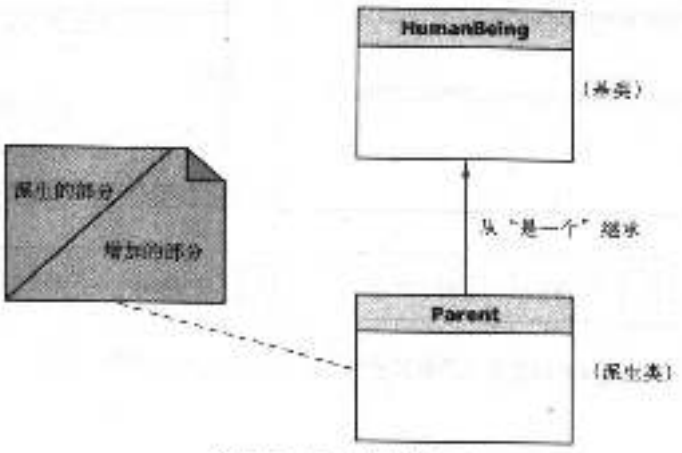


图 7-27　派生类型和继承

Parent 类继承了 HumanBeing 的所有属性，因为 Parent 类是 HumanBeing 基类的一个派生类（或子类）。如果 Fred 是一个对象，并且是 Parent 类的一个实例，那么 Fred 具有 Parent 的所有属性，还继承了 HumanBeing 的所有属性。图 7-28 给出了它的 Java 实现，C++ 版本在 private 和 public 修饰符的位置上有所不同。还有本例中 Java 的 extends 语法在 C++ 中以：public 代替。

```
class HumanBeing
{
   private int     age;
   private float   height;

   // 对 HumanBeing 进行操作的 public 声明

}// class HumanBeing


class Parent extends HumanBeing
{
   private String  nameOfOldestChild;
   private int     numberOfChildren;

   // 对 Parent 进行操作的 public 声明

}// class Parent
```

图 7-28 图 7-27 的 Java 实现

继承特性是所有面向对象编程语言的主要特性，然而，传统语言（例如 C、COBOL 或 FORTRAN）既不支持继承，也不支持类的概念。所以，面向对象范型不能在这些语言中直接实现（进一步的讨论见 8.7.4 节）。

在面向对象范型的术语中，看待图 7-27 中 Parent 和 HumanBeing 之间的关系另有两种方式。我们可以说 Parent 是 HumanBeing 的一个**特化**，或者 HumanBeing 是 Parent 的**泛化**。除了特化和泛化，类还有另外两个基本关系［Blaha，Premerlani，and Rumbaugh，1988］：聚合和关联。**聚合**指的是类的组件。例如 PersonalComputer 类可能包含 CPU、Monitor、Keyboard 和 Printer 组件，如图 7-29 所示（使用菱形表示聚合是另一种 UML 惯例）。关于这一点没有什么新意，它存在于任何支持记录的语言中，例如 C 中的 struct。然而，在面向对象的环境里，它用来组合相关的项，产生一个可重用的类（8.1 节）。

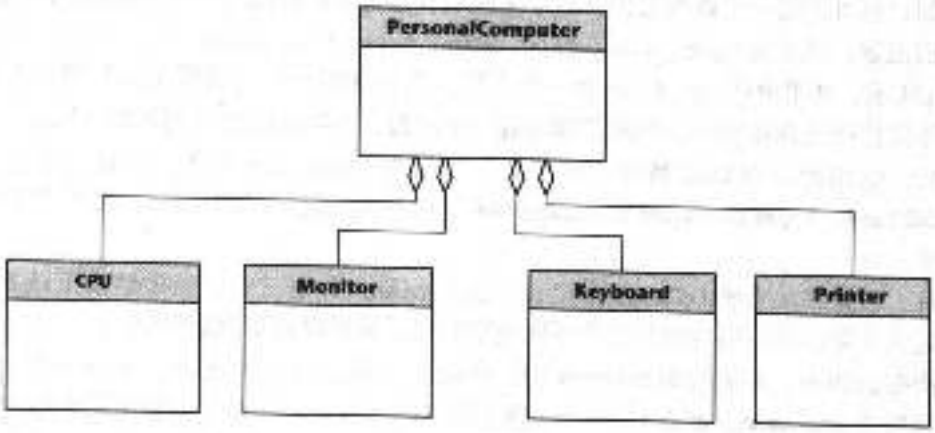



图 7-29 聚合的例子

**关联**指的是两个明显不相关的类之间的某种关系。例如，放射学家和艺术家之间看起来没有什么联系，但放射学家可以请教艺术家关于为一本描述 MRI 机器原理的书画插图的事情。图 7-30 给出了关联的图示，在这个例子中关联的本质通过 consults（请教）一词表达出来。另外，实心三角形表示关联的方向，毕竟艺术家脚踝骨折时可能会请教放射学家。

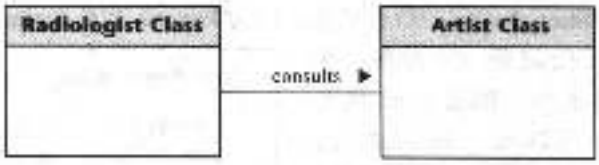



图 7-30　关联的例子

顺便提一下，像其他的面向对象的语言一样，Java 和 C++ 表示法的一个特征是，明确地反映了操作和数据的同等重要性。考虑支持记录的传统语言，例如 C。假设 record_1 是一个 struct（记录），field_2 是类内部的一个字段，那么该字段可表示为 record_1.field_2，也就是说，点（.）表示记录内部的成员关系。如果 function_3 是 C 模块内部的一个函数，那么 function_3() 表示那个函数的一个调用。

相反，假设 **ClassA** 是一个类，只有属性 attributeB 和方法 methodC。假设 ourObject 是 **ClassA** 的一个实例，那么字段指的是 ourObject.attributeB。进一步说，ourObject.methodC() 表示对该方法的调用。这样，点（.）用来表示对象内部的成员关系，而成员是属性或方法。

使用对象（或者说是类）的好处恰恰就是那些使用抽象数据类型的好处，包括数据抽象和过程抽象。另外，类的继承特征提供了更深层次的数据抽象，使产品开发更容易，错误倾向更少。还有另一个好处来自于继承、多态与动态绑定的结合，这是下一节的主题。

## 7.8　继承、多态和动态绑定

假设需要调用计算机的操作系统来打开一个文件，该文件可能存储在许多不同的介质上。例如，它可能是一个磁盘文件、磁带文件或软盘文件。使用传统范型的话，将有三个名称不同的函数，分别是 open_disk_file、open_tape_file 和 open_diskette_file，如图 7-31a 所示。如果声明 my_file 为一个文件，那么在运行时有必要测试它是一个磁盘文件、磁带文件还是软盘文件，以确定调用哪一个函数。相应的传统代码见图 7-32a。

相反，使用面向对象范型时，可定义一个名为 **FileClass** 的类，带有三个派生的类 **DiskFileClass**、**TapeFileClass** 和 **DisketteFileClass**，如图 7-31b 所示，其中开放的箭头表示继承。

现在，假设在父类 **FileClass** 中定义了方法 open，并由三个派生类继承了该方法。遗憾的是这并不起作用，因为打开三个不同类型的文件需要完成不同的操作。

解决办法如下，在父类 **FileClass** 中，声明一个虚拟的方法 open。在 Java 中，这样的方法被声明为 **abstract**，而在 C++ 中，使用保留字 **virtual**。在每个派生类中都有该方法的特定实现，而且每个方法都具有相同的名称，也就是 open，如图 7-31b 所示。再次假定声明 myFile 为一个文件，运行时发送了 myFile.open() 消息。面向对象的系统现在确定 myFile 是一个磁盘文件、磁带文件还是软盘文件，并调用相应的 open。也就是说，系统在运行时确定对象 myFile 是 **DiskFileClass** 类的实例、**TapeFileClass** 类的实例还是 **DisketteFileClass** 类的实例，并自动调用合适的方法。因为这是在运行时（动态）完成的，而不是在编译时（静态）完成的，因此这种把对象与合适的方法连接起来的行为被称为**动态绑定**。进一步说，因为方法 open 可应用于不同类的对象，它被称为**多态**（polymorphic），该术语的

意思是“许多形态”。就好像碳晶体可表现为许多不同的形态，包括钻石和软石墨一样，方法 open 可表现为三种不同的版本。在 Java 中，这些版本表示为 DiskFileClass.open、TapeFileClass.open 和 DisketteFileClass.open。（在 C++ 中，用两个冒号来替代点，这些文件被表示为 DiskFileClass:: open、TapeFileClass:: open 和 DisketteFileClass:: open。）然而，正是由于动态绑定，没有必要去确定调用哪一个方法来打开文件，而是在运行时只需发送 myFile.open（）消息，系统将确定 myFile 的类型（类）并调用正确的方法，见图 7-32b。

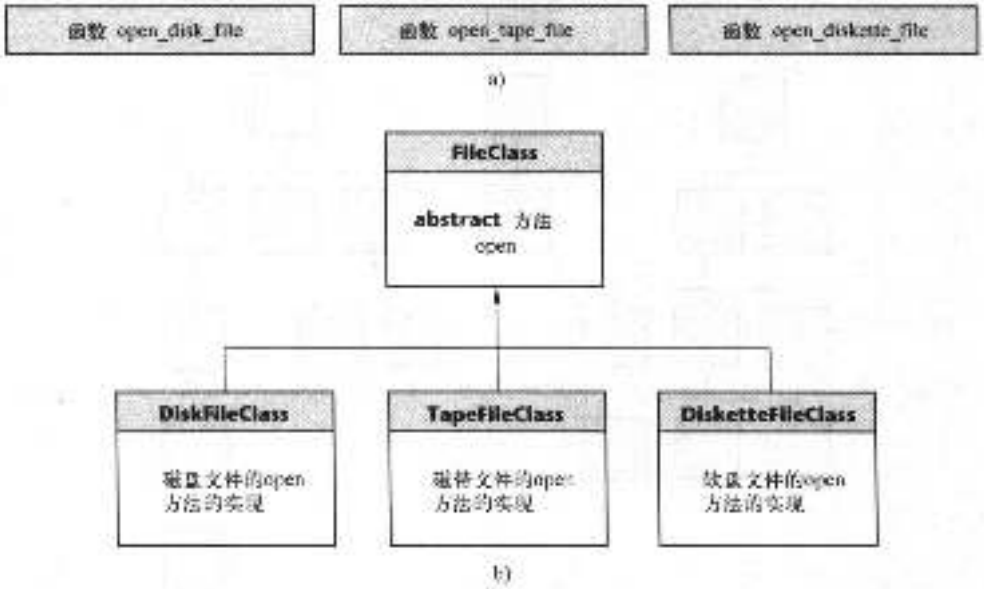



图 7-31　打开文件需要的操作。a）传统实现。b）面向对象的文件类层次（使用 Java 来表示）

```
switch (file_type)
{
    case 1:
        open_disk_file ( );              // file_type 1 对应磁盘文件
        break;
    case 2:
        open_tape_file ( );              // file_type 2 对应磁带文件
        break;
    case 3:
        open_diskette_file ( );          // file_type 3 对应软盘文件
        break;
}
```

a)

```
myFile.open ( )
```

b)

图 7-32　a）打开文件的传统代码，对应图 7-31a。
b）打开文件的面向对象代码，对应图 7-31b。

这些思路不仅适用于 abstract（virtual）方法。考虑如图 7-33 所示的类的层次，通过从 Base 类继承而派生出所有的类。假设方法 checkOrder（b: Base）把 Base 类的一个实例作为参数，那么由于继承、多态和动态绑定，不只是使用 Base 类的实例能调用 checkOrder，使用 Base 类的任何子类（也就是从 Base 类派生出的任何类）的实例也能调用它。需要做的只是调用 checkOrder，则运行时一切都会被照料得很好。这个技术的功能相当强大，应用它软件专业人员不用关心发送消息时参数的准确类型。

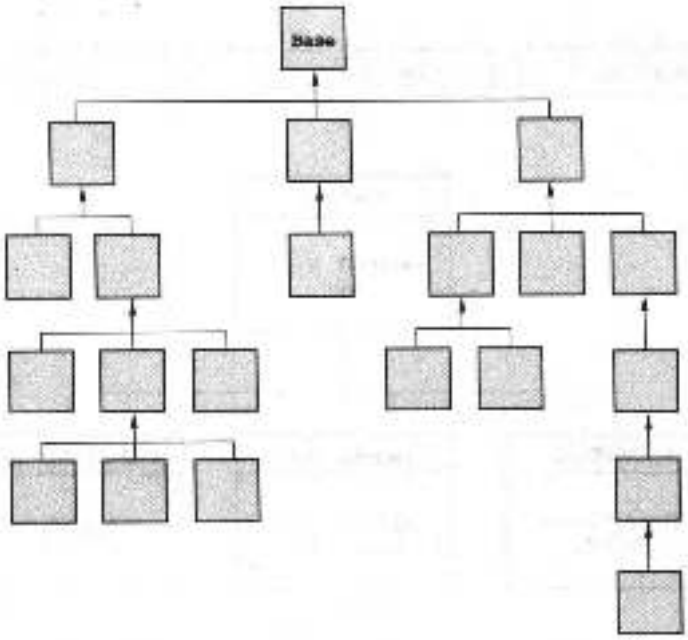



图 7-33　类的层次

然而，多态和动态绑定也有重大的缺点：

1）通常不太可能在编译时确定运行时将调用哪种特定的多态方法，因而很难确定引起故障的原因。

2）多态和动态绑定会对维护产生负面的影响。维护程序员的第一个任务是理解产品（如第 15 章所述，维护者很少会是开发产品的人），然而如果一个特定的方法有多个可能性，那么理解产品将会费尽心思。在代码中的一个特定位置上，程序员必须考虑可动态调用的所有可能的方法，这是一个耗费时间的任务。

因此，对于面向对象范型而言，多态和动态绑定既有优点，也有缺点。

在本章的最后我们来讨论面向对象范型。

## 7.9　面向对象范型

看待每个软件产品有两种方式。一种方式是只考虑数据，包括局部和全局的变量、参数、动态数据结构、文件等。另一种方式是只考虑对数据进行的操作，也就是过程和函数。按照把软件分为数据和操作这种分法，传统技术主要分为两组。面向操作的技术主要考虑产

品的操作，那么数据的重要性就位于第二位，只在深入分析了产品的操作之后才考虑它。相反，面向数据的技术强调产品的数据，只在数据的框架内考察操作。

面向数据和面向操作的方法的基本缺点是，数据和操作是同一事物相互依存的两个方面，数据项不能改变，除非对它进行操作；而没有相关数据的操作同样毫无意义。所以，需要同等对待数据和操作的技术。面向对象的技术能够做到这一点并不奇怪，毕竟对象是由数据和操作组成的。回忆一下，对象是抽象数据类型的实例（或者更确切地说，对象是类的实例），因此它合并了数据和对那些数据进行的操作，数据和操作在对象中作为同等重要的部分存在。类似地，在所有面向对象的技术中，考虑数据和考虑操作同样重要，哪一个也没有得到优待。

声称在面向对象范型的技术中同时考虑数据和操作是不准确的。从逐步求精法（5.1 节）的材料中可以清楚地看到，有很多次强调数据，也有很多次强调操作。然而总的来说，在面向对象范型的各阶段，数据和操作同等重要。

第 1 章和本章给出了许多原因来说明为什么面向对象范型比传统范型优越。根据所有这些原因可以得出结论，经过良好设计的对象，也就是具有高内聚和低耦合的对象可以对一个物理实体的所有方面进行建模。就是说，在一个现实世界实体和模拟它的对象之间有一个清楚的映像。

如何实现这些的细节被隐藏了，与一个对象进行通信的惟一途径是给对象发送消息。因此，对象基本上是具有良好定义接口的独立单元，并且容易维护，比较安全，发生连带错误的机会也减少了。此外，后面将在第 8 章中讲到，对象是可重用的，这种可重用的能力通过继承的特性得以增强。现在回到使用对象开发的问题上，通过把软件的这些基本构件结合起来建造一个大型产品是安全的，而且对这个开发的管理也容易些，因而更可能减少错误。

面向对象范型的所有这些优势引发出一个问题：如果传统范型与面向对象范型相比如此低下，为什么传统范型有这么多成功的实例？原因是在软件工程没有得到广泛的实践时，采用的普遍是传统范型，而那时，只是“编写”软件。对于管理者来说，最重要的事情是程序员写出程序代码行，付给产品的需求和规格说明（系统分析）的经费还没有小费服务的高，而且几乎没有什么设计。编码－修补模型（2.9.1 节）是 20 世纪 70 年代这项技术的典型。所以，开始时传统范型是软件开发者使用的主要方法性技术，那时没有人怀疑传统范型的结构化技术是全球范围内的软件行业的重要进步。然而，随着软件产品的规模增长，结构化技术的不足日益显露出来，而面向对象范型成为更好的替代物。

接下来这又产生另一个问题：我们怎么知道面向对象范型比其他所有今天存在的技术都要优越？没有什么数据能用来证明面向对象技术比其他任何当前的技术更好，而且很难想像如何得到这样的数据。我们能做的是根据已应用了面向对象范型的组织的经验来得出结论。尽管不是所有的报告都赞成，但如果不是压倒性的多数，至少大多数都证明了使用面向对象范型是明智的决策。

例如，IBM 公司曾报道过三个完全不同的项目，均使用面向对象技术［Capper，Colgate，Hunter，and James，1994］。几乎在每一个方面，面向对象范型都胜过传统范型，特别是在检测到的错误数量上有显著的减少；而且在开发和交付后维护期间，除了由于不可预见的商业变更而要求进行修改以外，其他情况的修改申请更少了；还有在适应和完善维护能力上有明显的提高。另外在实用性方面也有提高，尽管没有前面几项那么大，以及在性能上

没有什么实质的区别。

有 150 个富有经验的美国软件开发者接受了一项调查，以确定他们对面向对象范型的态度［Johnson，2000］。其中有 96 个开发者使用过面向对象范型，还有 54 个开发者仍使用传统范型来开发软件。两组人员都感到面向对象范型更优越，尽管使用面向对象范型的小组的积极态度更强些。两组人员基本上都不计较面向对象范型的各种缺点。

尽管面向对象范型有许多优点，但还是有一些真正的短处及问题被报道出来。一个频繁报道的问题与开发工作量和规模有关。第一次做一件新事情时，都会比以后的场合花费更长的时间，这个初始阶段有时被称为**学习曲线**（learning curve）。但当一个组织第一次使用面向对象范型时，通常比预计的需要更长的时间，即便已经考虑了学习曲线。这是因为产品的规模比使用结构化技术时大，特别是在产品有图形用户界面（GUI）时更是如此（参见 10.13 节）。在这之后，情况会大大改善。首先，交付后维护的花费更少了，从而减少了产品整个生命周期的成本。第二，下一次开发新产品时，从前面的项目中可以重用一些类，更进一步地降低了软件成本。第一次使用 GUI 时特别明显，花在 GUI 上的大部分工作量在后续的产品中均会得到补偿。

继承的问题更难解决一些，使用继承的一个主要原因是，新的子类与它的父类区别不大，却不会影响到它的父类或继承树中的其他祖先。然而反过来说，一旦实现了一个产品，那么对已存在的类进行修改会直接影响继承树中它的所有子孙，这通常被称为**脆弱的基类问题**。至少受影响的部分需要重新编译，在一些情况下，相关对象（受影响的子类的实例）的方法需要重新编程，这将不是一个小任务。为了最小化这个问题，在开发阶段仔细设计所有的类非常重要，这将减小对已存在的类进行修改所带来的影响。

不加约束地使用继承会带来第二个问题，除非明确地禁止，否则子类将继承它的父类（们）的所有属性。通常子类还具有它们自己的属性，结果在继承树低层的对象很快变得巨大起来，因而引起存储问题［Bruegge，Blythe，Jackson，and Shufelt，1992］。避免出现这个问题的一个方式是将“尽可能地使用继承”那句格言修改为“适当时使用继承”。另外，如果后继的类不需要祖先的某个属性，那么应该明确地排除这个属性。

第三组问题来自多态和动态绑定，在 7.8 节中已讨论过。

第四，用任何语言都有可能写出坏的代码。然而，用面向对象语言比使用传统语言更容易写出坏的代码，这是因为面向对象语言支持各种构造，在使用不恰当时，会给软件产品增加不必要的复杂性。因此，当使用面向对象范型时，必须多加小心以确保该代码总具有最高的质量。

最后一个问题：将来会有比面向对象范型更好的技术吗？也就是说，在将来是否会有一个新的技术出现在图 7-26 中最上部箭头的位置？即使是强烈鼓吹者，也没有宣称面向对象范型是解决所有软件工程问题的终极答案。进一步地，今天的软件工程已超越对象，瞄向了下一个重大突破，毕竟，在人类努力的领域中，很少有过去的发现超出今天提出的任何事情的。未来的方法肯定会取代面向对象范型。已有建议认为**面向问题方面的编程**（aspect-oriented programming，AOP）可能会发挥重要作用［Murphy et al，2001］。AOP 能否确实成为图 7-26 的未来版本的下一个主要概念，或者是否某些其他的技术将被作为面向对象范型的后继者而广泛使用，这些都有待进一步观察。重要的经验是，基于目前的知识水平，面向对象范型看来是最好的了。

## 本章回顾

本章开始描述了模块（7.1 节），接下来的两节从模块内聚和模块耦合两方面分析了什么构成了良好设计的模块（7.2 节和 7.3 节）。特别地，模块应具有高内聚和低耦合。这里给出了各种类型的内聚和耦合的描述。在 7.4 节至 7.7 节中提出了各种类型的抽象。在数据封装（7.4 节）中，一个模块包含一个数据“结构”和在这个数据结构上进行的操作。一个抽象数据类型（7.5 节）是一个数据“类型”，连同在这种类型的实例上进行的操作。信息隐藏（7.6 节）包含对模块这样的设计：其中实现的细节对其他模块是隐藏的。对类的描述使增加抽象性发展到顶点，类是支持继承的一种抽象数据类型（7.7 节）。对象是类的一个实例，多态和动态绑定是 7.8 节的主题。本章结束于对面向对象范型的讨论（7.9 节）。

## 进一步阅读

[Dahl and Nygaard, 1966] 中首次描述了对象，本章中的许多思想是由 Parnas 最早提出来的 [Parnas, 1971, 1972a, 1972b]。在软件开发中使用抽象数据类型是在 [Liskov and Zilles, 1974] 中提出的，另一个重要的早期论文是 [Guttag, 1977]。

内聚和耦合方面的文章主要来源于 [Stevens, Myers, and Constantine, 1974]。组合化/结构化设计扩展成对象的思想在 [Binkley and Schach, 1997] 中有描述。

关于对象的介绍性材料可在 [Meyer, 1997] 中找到。在 [Meyer, 1996b] 中描述了不同类型的继承。有关面向对象范型的许多小文章可在 [El-Rewini et al., 1995] 中找到。面向对象编程系统、语言和应用（OOPSLA）年会的学报包含了广泛的研究论文，还包含了描述成功的面向对象项目的非正式报告。在 [Capper, Colgate, Hunter, and James, 1994] 中描述了三个 IBM 项目成功地使用了面向对象范型。对面向对象范型的态度的调查出现在 [Johnson, 2000] 中。[Fayad, Tsai, and Fulghum, 1996] 描述了如何转向面向对象的技术，其中还包含了许多给管理者的建议。

《IEEE Computer》杂志 1992 年 10 月刊包含了一些关于对象的重要文章，包括描述“按合同设计”的 [Meyer, 1992]。关于对象的各种文章还可在《IEEE Software》杂志 1993 年 1 月刊中找到，Snyder 的论文仔细地定义了该领域中的关键术语 [Snyder, 1993]，非常有用。多态可能存在的缺陷在 [Ponder and Bush, 1994] 中有所描述。《Communications of the ACM》杂志 1995 年 10 月刊中包含有关于对象技术的文章，《IBM Systems Journal》杂志的 1996 年第 2 期中也有类似的文章。

在《Communications of the ACM》的 2001 年 10 月刊上出现了 11 篇关于面向方面编程的文章，其中 [Elrad et al., 2001] 和 [Murphy et al., 2001] 特别令人感兴趣。在 [Cartwright and Shepperd, 2000] 中出现了一个有关继承对差错密度影响的调查。

## 习题

7.1　选择一种你熟悉的编程语言，考虑 7.1 节给出的模块化的两个定义，确定这两个定义中的哪一个包含了你能直观理解的东西，使你能用你所选择的语言建造一个模块。

7.2　确定以下模块的内聚类型：

`edit_profit_and_tax_record`（编辑利润和税款记录）

edit_profit_record_and_tax_record（编辑利润记录和税款记录）

read_delivery_record_and_check_salary_payments（读取交付记录并检查工资的支付）

compute_the_optimal_cost_using_Aksen's_algorithm（使用 Aksen 算法计算最优成本）

measure_vapor_pressure_and_sound_alarm_if_necessary（测量蒸汽压力并在必要时报警）

7.3　假设你是一个负责产品开发的软件工程师，你的管理者要求你研究一下确保你所在小组设计出的模块能够尽可能重用的途径，你将如何回答她？

7.4　现在你的管理者要求你确定如何使已存在的模块能够被重用。你的第一个建议是把每个具有偶然性内聚的模块分割成具有功能性内聚的单个模块。而你的管理者正确地指出了这单个的模块还没有经过测试，也没有建立规范的文档，现在你将如何回答？

7.5　维护时内聚的影响是什么？

7.6　维护时耦合的影响是什么？

7.7　请仔细区分数据封装和抽象数据类型。

7.8　请仔细区分抽象和信息隐藏。

7.9　请仔细区分多态和动态绑定。

7.10　如果我们使用多态而不使用动态绑定会发生什么？

7.11　如果我们使用动态绑定而不使用多态会发生什么？

7.12　根据你的导师的要求，把图 7-21 中的注释转化为 C++ 或 Java，要确保完成的模块能够正确执行。

7.13　有人说，C++ 和 Java 支持对抽象数据类型的实现，但却以放弃信息隐藏为代价。请讨论这个观点。

7.14　本章开始时的“如果你想知道”部分中指出过，对象是在 1966 年首次提出的。但在将近 20 年之后，对象才被重新使用并得到广泛接受，你能解释这种现象吗？

7.15　你的导师将发布一个传统的软件产品，从信息隐藏、抽象级别、耦合和内聚的角度分析一下该产品的模块。

7.16　你的导师将发布一个面向对象的软件产品，从信息隐藏、抽象级别、耦合和内聚的角度分析一下该产品的模块。将答案与习题 7.15 的答案进行对比。

7.17　（学期项目）假设附录 A 中 Ophelia 的 Oasis 产品是使用传统范型开发的。举出你希望能找到的功能性内聚的模块的例子。现在假设该产品是使用面向对象范型开发的，举出你希望能找到的类的例子。

7.18　（软件工程读物）你的导师将分发［Johnson，2000］的复印件。为什么接受调查者会不在意面向对象范型的缺点？谈谈你的看法。

## 参考文献

[Binkley and Schach, 1997] A. B. BINKLEY AND S. R. SCHACH, "Toward a Unified Approach to Object-Oriented Coupling," *Proceedings of the 35th Annual ACM Southeast Conference,* Murfreesboro, TN, April 2–4, 1997, pp. 91–97.

[Blaha, Premerlani, and Rumbaugh, 1988] M. R. BLAHA, W. J. PREMERLANI, AND J. E. RUMBAUGH, "Relational Database Design Using an Object-Oriented Methodology," *Communications of the ACM* **31** (April 1988), pp. 414–27.

[Bruegge, Blythe, Jackson, and Shufelt, 1992] B. BRUEGGE, J. BLYTHE, J. JACKSON, AND J. SHUFELT, "Object-Oriented Modeling with OMT," Proceedings of the Conference on Object-Oriented Programming, Languages, and Systems, OOPSLA '92, *ACM SIGPLAN Notices* **27** (October 1992), pp. 359–76.

[Capper, Colgate, Hunter, and James, 1994] N. P. CAPPER, R. J. COLGATE, J. C. HUNTER, AND M. F. JAMES, "The Impact of Object-Oriented Technology on Software Quality: Three Case Histories," *IBM Systems Journal* **33** (No. 1, 1994), pp. 131–57.

[Cartwright and Shepperd, 2000] M. CARTWRIGHT AND M. SHEPPERD, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Transactions on Software Engineering* **26** (August 2000), pp. 786–95.

[Dahl and Nygaard, 1966] O.-J. DAHL AND K. NYGAARD, "SIMULA—An ALGOL-Based Simulation Language," *Communications of the ACM* **9** (September 1966), pp. 671–78.

[El-Rewini et al., 1995] H. EL-REWINI, S. HAMILTON, Y.-P. SHAN, R. EARLE, S. MCGAUGHEY, A. HELAL, R. BADRACHALAM, A. CHIEN, A. GRIMSHAW, B. LEE, A. WADE, D. MORSE, A. ELMAGRAMID, E. PITOURA, R. BINDER, AND P. WEGNER, "Object Technology," *IEEE Computer* **28** (October 1995), pp. 58–72.

[Elrad et al., 2001] T. ELRAD, M. AKSIT, G. KICZALES, K. LIEBERHERR, AND H. OSSHER, "Discussing Aspects of AOP," *Communications of the ACM* **44** (October 2001), pp. 33–38.

[Fayad, Tsai, and Fulghum, 1996] M. E. FAYAD, W.-T. TSAI, AND M. L. FULGHUM, "Transition to Object-Oriented Software Development," *Communications of the ACM* **39** (February 1996), pp. 108–21.

[Flanagan, 2002] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference,* 4th ed., O'Reilly and Associates, Sebastopol, CA, 2002.

[Gerald and Wheatley, 1999] C. F. GERALD AND P. O. WHEATLEY, *Applied Numerical Analysis,* 6th ed., Addison Wesley, Reading, MA, 1999.

[Goldberg and Robson, 1989] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language,* Addison Wesley, Reading, MA, 1989.

[Guttag, 1977] J. GUTTAG, "Abstract Data Types and the Development of Data Structures," *Communications of the ACM* **20** (June 1977), pp. 396–404.

[Johnson, 2000] R. A. JOHNSON, "The Ups and Downs of Object-Oriented System Development," *Communications of the ACM* **43** (October 2000), pp. 69–73.

[Knuth, 1974] D. E. KNUTH, "Structured Programming with `go to` Statements," *ACM Computing Surveys* **6** (December 1974), pp. 261–301.

[Liskov and Zilles, 1974] B. LISKOV AND S. ZILLES, "Programming with Abstract Data Types," *ACM SIGPLAN Notices* **9** (April 1974), pp. 50–59.

[Meyer, 1986] B. MEYER, "Genericity versus Inheritance," Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, *ACM SIGPLAN Notices* **21** (November 1986), pp. 391–405.

[Meyer, 1992] B. MEYER, "Applying 'Design by Contract'," *IEEE Computer* **25** (October 1992), pp. 40–51.

[Meyer, 1996b] B. MEYER, "The Many Faces of Inheritance: A Taxonomy of Taxonomy," *IEEE Computer* **29** (May 1996), pp. 105–8.

[Meyer, 1997] B. MEYER, *Object-Oriented Software Construction,* 2nd ed., Prentice-Hall, Upper Saddle River, NJ, 1997.

[Murphy et al., 2001] G. C. MURPHY, R. J. WALKER, E. L. A. BANNIASSAD, M. P. ROBILLARD, A. LIA, AND M. A. KERSTEN, "Does Aspect-Oriented Programming Work?" *Communications of the ACM* **44** (October 2001), pp. 75–78.

[Myers, 1978b] G. J. MYERS, *Composite/Structured Design,* Van Nostrand Reinhold, New York, 1978.

[Parnas, 1971] D. L. PARNAS, "Information Distribution Aspects of Design Methodology," *Proceedings of the IFIP Congress,* Ljubljana, Yugoslavia, 1971, pp. 339–44.

[Parnas, 1972a] D. L. PARNAS, "A Technique for Software Module Specification with Examples," *Communications of the ACM* **15** (May 1972), pp. 330–36.

[Parnas, 1972b] D. L. PARNAS, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* **15** (December 1972), pp. 1053–58.

[Ponder and Bush, 1994] C. PONDER AND B. BUSH, "Polymorphism Considered Harmful," *ACM SIGSOFT Software Engineering Notes* **19** (April, 1994), pp. 35–38.

[Schach et al., 2002] S. R. SCHACH, B. JIN, D. R. WRIGHT, G. Z. HELLER, AND A. J. OFFUTT, "Maintainability of the Linux Kernel," *IEE Proceedings—Software* **149** (February 2002), pp. 18–23.

[Schach et al., 2003a] S. R. SCHACH, B. JIN, DAVID R. WRIGHT, G. Z. HELLER, AND J. OFFUTT, "Quality Impacts of Clandestine Common Coupling," *Software Quality Journal* **11** (July 2003), pp. 211–18.

[Schach and Stevens-Guille, 1979] S. R. SCHACH AND P. D. STEVENS-GUILLE, "Two Aspects of Computer-Aided Design," *Transactions of the Royal Society of South Africa* **44** (Part 1, 1979), pp. 123–26.

[Shneiderman and Mayer, 1975] B. SHNEIDERMAN AND R. MAYER, "Towards a Cognitive Model of Programmer Behavior," Technical Report TR-37, Indiana University, Bloomington, 1975.

[Snyder, 1993] A. SNYDER, "The Essence of Objects: Concepts and Terms," *IEEE Software* **10** (January 1993), pp. 31–42.

[Stevens, Myers, and Constantine, 1974] W. P. STEVENS, G. J. MYERS, AND L. L. CONSTANTINE, "Structured Design," *IBM Systems Journal* **13** (No. 2, 1974), pp. 115–39.

[Stroustrup, 2000] B. STROUSTRUP, *The C++ Programming Language,* Special 3rd ed., Addison Wesley, Reading, MA, 2000.

[Yourdon and Constantine, 1979] E. YOURDON AND L. L. CONSTANTINE, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Prentice-Hall, Englewood Cliffs, NJ, 1979.

# 第 8 章　可重用性和可移植性

**学习目标**

学完本章之后，你应当能够：

- 解释为什么重用如此重要；
- 理解重用的障碍；
- 描述在各工作流期间获得重用的技术；
- 讨论重用对可维护性的影响；
- 解释为什么可移植性是重要的；
- 理解获得可移植性的障碍；
- 开发可移植软件。

如果重复开发是一项犯罪，那么许多软件专业人员都将被投入监狱。例如，如果没有几十万的话，也有好几万个不同的 COBOL 工资单程序都在做相同的事情。肯定地说，全世界只需要一个工资单程序，它能在各种硬件上运行，并可在必要时根据单个组织的专门需要进行裁剪。然而，全世界的无数个组织不是使用前面所述的工资单程序，而是从头开始建造它们自己的工资单程序。本章中，我们研究为什么软件工程师们愿意不断地重复开发，以及如何使用可重用的组件建造可移植的软件。下面首先讨论可移植性和重用性之间的区别。

## 8.1　重用的概念

如果比起从头开始编程，很容易修改整个产品使其在另一个编译器－硬件－操作系统配置上运行，那么该产品是**可移植的**。相反，**重用**指的是使用一个产品中的组件来简化另一个功能不同的产品的开发。一个可重用的组件不一定是一个模块或代码段——它可以是一个设计、一个用户手册的一部分、一组测试数据或一个周期和成本估算。（有关重用的另一个不同的观点，见下面的“如果你想知道”部分。）

**如果你想知道**

重用不仅仅限于软件。例如，律师现在很少从头拟制遗嘱。他们使用字处理器存储他们先前拟制的遗嘱，然后做适当修改并形成一份新的遗嘱。其他法律文本如合同，通常也是以相同的方式从已有文档做起。

古典作曲家常常重用他们自己的音乐。例如，在 1823 年弗朗兹·舒伯特为 Helmina von Chezy 的剧本写了一个幕间小品《Rosamunde，Princess of Cyprus》，过了几年他在他的弦乐四重奏 13 号的慢板作品中又重用了该材料。贝多芬也在他的作品中重用了另一个伟大作曲家莫扎特的音乐。他从莫扎特的歌剧《魔笛》第 22 场中简单地借用了咏叹调“一个女朋友或小妇人”，然后对该咏叹调中为钢琴配乐的大提琴演奏写了一连串 7 个变奏。

在我看来，有史以来最伟大的重用者是莎士比亚。他的天才就在于重用他人的情节上——我无法想起哪一行故事是他自己的。例如，他的历史剧很大程度上重用了 Raphael Holinshed 写于 1577 年的作品《英格兰、苏格兰和爱尔兰编年史》。然后，莎士比亚的《罗密欧和朱丽叶》（1594）几乎整行整行地借用了 Arthur Brooke 出版于 1562 年的长诗《The

Tragicall Historye of Romeus and Iuliet》。该书出版那年是莎士比亚出生的前两年。

但是这个重用传奇并不是始于此。事实上，已知该故事的最早重用的版本出现在公元200年，Ephesus城的古希腊小说家Xenophon写的《Ephesiaka》（Ephesus城传说）。在1476年，Tommaso Guardati（更为人熟悉的名字是Masuccio Salernitano）在他的50篇小说选集《Ⅱ Novellino》中，第33篇小说中重用了Xenophon的传说。在1530年，Luigi da Porto在《Historia Novellamente Ritrovata di Due Nobili Amanti》（一个新编写的有关两个贵族恋人的故事）里重用了那个故事。他将这个故事第一次安排在了意大利的Verona。Brooke的诗重用了Matteo Bandello写的《Giulietta e Romeo》（1554）的一部分，它是da Porto版本的重用。

而这个重用传奇并未结束于“罗密欧和朱丽叶”。在1957年，“西边故事”在百老汇公演。这个音乐剧由Arthur Laurents编写剧本，Stephen Sondheim填写歌词，由Leonard Bernstein谱曲，该剧重用了莎士比亚的故事版本。该百老汇音乐剧后来被一部好莱坞影片所重用，该影片在1961年获得10项奥斯卡奖。

有两种类型的重用，偶然重用和有意重用。如果一个新产品的开发者意识到，以前设计的产品的一个组件可在这个新产品中重用，那么这是**偶然重用**或**机会重用**。另一方面，使用专门为未来可能的重用而建造的软件组件则是**有意重用**或**有计划重用**。有意重用比偶然重用的一个潜在好处是：为未来可能的重用专门建造的组件会更容易重用，也更安全，通常这样的组件是健壮的、文档完善并经过全面测试的。另外，它们通常显示出风格的一致性，使维护更容易。但是从另一个角度讲，在公司内部实现有意重用可能会很昂贵，它需要时间来规定、设计、实现、测试一个软件组件，并形成软件组件的文档。然而不保证这样一个组件一定会被重用，从而补偿在开发这个潜在的可重用组件上的投资。

首次建造出计算机时，没有什么是可重用的。每次开发产品时，诸如乘法程序、输入-输出程序或计算正弦和余弦的程序等等均是从头开始建造的。然而人们很快意识到这是一个极大的浪费，因而建造了子程序库。程序员只需调用均方根或正弦函数即可。这些子程序库逐渐变得越来越复杂，并开发成运行时支持程序。所以，当程序员调用一个C++或Java方法时，不需要写出代码来管理堆栈或直接传递参数，可以通过调用合适的运行时支持程序来自动进行处理。子程序库的概念已经扩展到大型的静态库，如SPSS [Norusis，2000]，以及数值分析库，如NAG [2003]。类库在帮助用户使用面向对象的语言时也起到重要的作用。例如，Smalltalk的成功至少部分归功于Smalltalk库中项目的广泛多样，以及浏览器（一种可帮助用户扫描类库的CASE工具）的出现。在C++中，有大量的不同类型的库可用，许多库是在公共域中，例如C++标准模板库（Standard Template Library，STL）[Musser and Saini，1996]。

应用编程接口（API）通常是一组有助于编程的操作系统调用，例如，Win32是用于微软操作系统（例如Windows 2000和Windows XP）的API，而Macintosh Toolbox是用于Mac OS（Macintosh操作系统）的API。尽管通常情况下，API是作为一组操作系统的调用来实现的，但对于建造API程序的程序员来说，可把API看作一个子程序库。例如，Java应用编程接口包含有许多软件包（库）。

无论软件产品的质量有多高，如果它花费了2年的时间才投入市场，它将卖不出去，而与之竞争的产品可能只需要1年就推向市场。开发过程的长短在市场经济中尤其重要。如果

产品不具有时间上的竞争性，那么其他构成“好”产品的标准都无关紧要了。对于向市场推出产品屡次失败的公司来说，软件重用提供了一项吸引人的技术，毕竟，如果重用已存在的组件，那么不需要再去设计、实现、测试和归档该组件。关键是，平均来说软件产品只有大约 15%真正符合最初的意图［Jones，1984］。产品的另外 85%理论上是可以标准化的，并可在未来的产品中重用。

85%的数值基本上是重用率的一个理论的上限，尽管如此，实际中只能实现 40%左右的重用率。这导致一个明显的问题：如果实际中可实现这样的重用率，而且重用不是什么新思想，为什么很少有组织使用重用来缩短开发过程？

## 8.2　重用的障碍

重用会面临这样一些障碍：

- 太多的软件专业人员宁愿从头编写一个程序，也不愿重用一个别人编写的程序。障碍是：一个程序只有是自己编的才是好的，否则称为“不在此开发”（not invented here，NIH）综合症［Griss，1993］。NIH 是一个管理方面的事情，如果管理者意识到这个问题，就可解决它，通常可通过提供财政上的激励来促进重用。
- 许多开发者更愿意重用一个他们确信不会给产品带来错误的程序，这种注重软件质量的态度很容易理解，毕竟每个软件专业人员都看过别人写出的有错误的程序。解决办法是在重用这些程序前，让这些潜在可重用的程序经受详尽的测试。
- 一个大型的组织可能有几十万个潜在有用的组件，如何存储这些组件以备日后有效地检索？例如，一个可重用的组件数据库可能包含 20 000 项，其中 125 个是排序程序。那么必须规划该数据库，使新产品的设计者能够很快地确定这 125 个排序程序中的哪一个对新产品更合适。解决存储－检索问题是一个技术事项，已经提出了大量的解决方案［e.g.，Meyer，1987 or Prieto-Díaz，1991］。
- 重用会很昂贵。Tracz［1994］已经说明了需要考虑三项成本：建造可重用组件的成本、重用它的成本以及定义和实现一个重用过程的成本。他估算，仅仅是建造可重用的组件就将增加至少 60%的成本，一些组织还报道过，成本将增加 200%，甚至 480%。而在惠普公司的一个重用项目中，建造一个可重用组件的成本只占 11%［Lim，1994］。
- 对于合同软件会产生司法问题。通常按照在软件开发组织和客户之间签订的合同，软件产品是属于客户的。因此，如果软件开发者在给另一个客户开发的新产品中重用客户产品的一个组件，这实质上构成了对第一个客户的侵权。对于内部软件，即当开发者和客户是同一组织的成员时，这个问题不存在。
- 另一个阻力来自于当重用商业现货（commercial off-the-shelf，COTS）组件时，通常开发者很难得到 COTS 组件的源代码，因此，重用 COTS 组件的软件限制了可扩展性和可修改性。

前面的四个障碍至少在原则上是可克服的。因此，除了某些法律事务和 COTS 组件的问题，基本上没有什么障碍能阻止在一个软件组织内部实现重用（但是见下面的“如果你想知道”部分）。

**如果你想知道**

万维网（World Wide Web）是“城市神话”的巨大来源，也就是说，显然真实的故事

在被仔细研究时却有点站不住脚，代码重用就是这样的一个城市神话。

故事说的是澳大利亚空军为训练飞行员的作战能力而建立了一个虚拟现实的训练仿真器。为使情况尽可能地真实，程序中加进了具体的风景和（北方领土中的）袋鼠群。毕竟飞机掠过惊动动物群时，扬起的尘土会将该飞机的位置暴露给敌人。

这里要求程序员对袋鼠的移动和它们对飞机的反应进行建模。为节省时间，程序员重用了原来用来模拟被飞机攻击的步兵团反应的代码，只进行了两个修改：他们把战士的图标换成了袋鼠的图标，并提高了图片移动速度。

一天，一组澳大利亚飞行员需要通过飞行仿真器向来访的美国飞行员展示他们的威力。他们“低空掠过”（飞得非常低）虚拟的袋鼠，像预料中的一样，袋鼠四下散开了，然后袋鼠又从一个小山后面出现，并向飞机发射了毒刺导弹。当重用那个虚拟步兵实现时，程序员忘记了去掉步兵发射导弹的那部分代码。

然而，正如《The Risks Digest》（军事文摘）中所报道的，这个故事好像不完全是一个城市神话——这里的大部分情节是真正发生的［Green，2000］。Anne－Marie Grisogono 博士，澳大利亚国防科技部的陆军仿真师师长，于 1999 年 5 月 6 日在澳大利亚的堪培拉讲述了这个故事。尽管该仿真器尽可能地模拟现实（它甚至包含了超过 200 万个虚拟树木，像在航空照片上看到的一样），加进袋鼠只是为了好玩。程序员实际上重用了毒刺导弹分离程序，这样袋鼠们可以检测到飞机的到来，但袋鼠的特性被设置为“撤退”，所以如果飞机来了，袋鼠应能逃跑。然而当软件小组在实验室里测试他们的仿真器时（而不是在参观者面前），发现他们忘记去掉武器和“开火”的特性了，还有，他们也没有规定被仿真的事物使用什么样的武器，因此当袋鼠向飞机开火时，使用了默认的武器，碰巧是大型的彩色充气球。

Grisogono 证实了袋鼠们立即缴械了，因此现在飞过澳大利亚是安全的。虽然结局是好的，但软件专业人员仍需要特别注意，重用代码时不要重用得过度。

## 8.3　重用实例研究

许多实例研究显示了在实践中是如何成功地进行重用的。具有重大影响的重用实例研究包括：［Matsumoto，1984，1987］，［Selby，1989］，［Prieto-Díaz，1991］和［Lim，1994］。在此我们将描述两个实例研究。第一个描述了发生在 1976 年到 1982 年之间的重用项目。它之所以重要，是因为其中用于 COBOL 设计的重用机制，与今天在面向对象框架（8.5.2 节）下应用的重用机制是相同的。这个实例研究因此用于阐明现代的重用实践。

### 8.3.1　Raytheon 导弹系统部

1976 年，在美国 Raytheon 公司的导弹系统部进行了一项研究，确定设计和代码的有意重用是否可行［Lanergan and Grasso，1984］。期间分析了 5000 多个使用中的 COBOL 产品，并对它们进行了分类。研究者确定，在一个商业应用产品中，只完成 6 个基本操作，40%～60%的商业应用设计和模块可被标准化和重用。基本的操作包括数据排序、编辑或处理数据、合并数据、分解数据、更新数据和报告数据。在接下来的 6 年里，Raytheon 公司集中精力试图在任何可能的地方重用设计和代码。

Raytheon 方法以两种方式使用了重用：研究者所谓的功能模块和 COBOL 程序逻辑结构。在 Raytheon 的术语中，**功能模块**是为一个专门的目的而设计和编写的一个 COBOL 代

码段，例如一个编辑程序、数据库过程的部分调用、税款计算程序或可接受账户的日期累计程序。使用了 3200 个可重用的模块产生的应用，平均来说包含 60% 的可重用代码。功能模块被仔细地设计、测试并形成文档。使用这些功能模块的产品被证明更灵活，而且需要更少的产品整体测试。

这些模块存储在一个标准的复制库中，可使用动词 **copy** 来得到它。也就是说，在应用产品内部并不物理地存在这些代码，只是由 COBOL 编译器在编译时包含进来，该机制与 C 或 C++ 中的 **#include** 相类似。因而生成的源代码比这些代码物理地存在时要短得多，维护起来也更容易。

Raytheon 研究者们还使用了他们所说的 **COBOL 程序逻辑结构**，一个必须充实成为完整产品的框架。逻辑结构的一个例子是更新逻辑结构，它可用于完成按顺序的更新，例如 5.1.1 节中的实例研究。其中创建了错误处理，就像顺序检查一样，该逻辑结构在长度上有 22 个段落（COBOL 程序的单元）。可以使用诸如 get-transaction、print-page-headings 和 print-control-totals 这样的功能模块填充许多段落。图 8-1 展示了一个 COBOL 程序逻辑结构的框架，其中的段落中填进了功能模块。

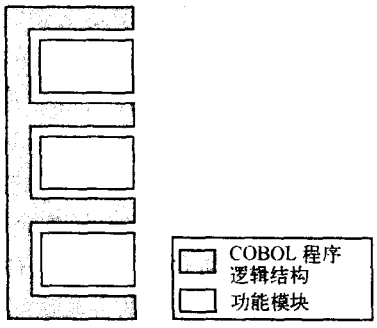



图 8-1　Raytheon 导弹系统部重用机制的图示

使用这样的模板有许多好处，它可使产品的设计和编码更快且更容易，因为产品的框架已经存在，只需填进细节。易出错的区域，例如文件结束的条件句已经经过测试，事实上，整个产品的测试将更容易。但 Raytheon 认为，主要的好处在于用户申请进行修改或增强功能时，一旦维护程序员熟悉了相关的逻辑结构，就几乎可以认为他就是原始开发小组的一员了。

到 1983 年，在开发新产品中使用逻辑结构已经超过了 5500 次，大约 60% 的代码包含有功能模块，也就是可重用的代码。这意味着设计、编码、模块测试和归档时间也节省了 60%，给软件产品开发带来了大约 50% 的生产力增长。但是，对于 Raytheon 来说，该项技术的真正好处在于，相同的风格带来的可读性和可理解性将减少 60% ～80% 的维护成本。遗憾的是，在取得必要的维护数据之前，Raytheon 关闭了该部门。

第二个重用实例研究是一个告诫性的故事，而不是一个成功的故事。

### 8.3.2　欧洲航天局

1996 年 6 月 4 日，欧洲航天局第一次发射了阿丽亚娜 5 号火箭，由于一个软件错误，火箭升空后大约 37 秒便爆炸了。火箭和有效载体的成本大约为 5 亿美元 [Jézéquel and Meyer，1997]。

引起这次故障的主要原因是试图将一个 64 位的整数转换为一个 16 位的无符号整数。被转换的数比 $2^{16}$ 还要大，因此产生了一个 Ada 异常（运行时故障）。遗憾的是，在代码中对此没有设置明确的异常处理过程，因此软件死机了。这引起火箭上的计算机崩溃，因而造成了阿丽亚娜 5 号火箭的爆炸。

具有讽刺意味的是，引起故障的那个转换是不必要的。在火箭升空前进行某种计算以校准惯性参考系统，这些计算应在升空前 9 秒时停止。然而，如果在倒数计秒中有一个随后的暂

停，那么在倒数计秒重新开始后重置惯性参考系统需要几个小时。为避免发生这样的事情，在开始进入飞行状态（也就是完全飞行）后的 50 秒里仍继续进行计算（尽管这样，一旦开始升空，就没有办法校准惯性参考系统了）。这个无用的继续校准过程的计算导致了该故障。

欧洲航天局使用了一个谨慎的软件开发过程，它包含了有效的软件质量保证成份。那么为什么在 Ada 代码中没有处理这种可能发生的溢出错误的异常处理程序呢？为了不加重计算机的负担，那些不可能导致溢出的转换被保留下来，没有采取任何保护措施。有问题的代码来源于控制阿丽亚娜 4 号火箭（阿丽亚娜 5 号的先驱）的软件，已经存在了 10 年，它在被重用时，没有做任何修改，也没有进一步测试。数学分析已经证明了这段有问题的计算对于阿丽亚娜 4 号火箭来说是安全的，然而，该分析是在某种假设的前提下进行的，该假设对于阿丽亚娜 4 号火箭来说是正确的，但对于阿丽亚娜 5 号火箭来说却不是这样。所以，该分析不再适用，需要有异常处理程序的保护来避免这种可能的溢出。如果不是因为性能的限制，肯定会在阿丽亚娜 5 号的 Ada 代码中设置异常处理程序。换句话说，在测试中或者安装了产品之后（6.5.3 节），如果相关的模块中包含了一个声明，要求被转换的数小于 $2^{16}$，那么使用 **assert** 编译指示可以避免阿丽亚娜 5 号火箭的爆炸［Jézéquel and Meyer，1997］。

这个重用的主要教训是在一种环境下开发出的软件用于另一个环境时，必须重新进行测试，也就是说，重用的软件模块不需要自己进行重新测试，但在它被集成到新产品中之后，必须进行重新测试。另一个教训如 6.5.2 节所述，完全依赖数学证明的结果是不明智的。

下面我们考察一下面向对象范型对重用的影响。

## 8.4　对象和重用

在大约 30 年前首次提出组合化/结构化设计（C/SD）理论时，人们认为理想的模块是具有功能性内聚的模块（7.2.6 节）。也就是说，如果一个模块只完成一个动作，就可认为该模块是重用的典型候选者，对这种模块的维护也将是容易的。这个推理中的一个缺陷是，具有功能性内聚的模块不是自包含和独立的，相反，它必须对数据进行操作。如果重用这样的模块，那么它所操作的数据也必须被重用。如果新产品中的数据与原产品中的数据不同，那么要么修改数据，要么修改具有功能性内聚的模块。所以，与我们以前相信的相反，功能性内聚不适于重用。

根据传统的 C/SD，下一个最好的模块类型是具有信息性内聚的模块（7.2.7 节）。现在我们理解了这样的模块基本是一个对象，也就是一个类的实例。一个经过良好设计的对象是软件的基本建造块，因为它对真实世界中的实体（概念独立或封装）的所有特征都进行了建模，但隐蔽了它的数据和对数据进行的操作的实现（物理独立或信息隐藏）。这样，当正确使用了面向对象范型时，得到的模块（对象）具有信息性内聚，这促进了重用。

## 8.5　设计和实现期间的重用

在设计期间可能会有明显不同的重用类型，重用的内容会从一个或两个制品到整个软件产品的体系结构。现在我们考察一下设计重用的各种类型，其中一些将带到实现中。

### 8.5.1　设计重用

当设计一个产品时，设计小组的成员可能会发现从早先的设计中得到的模块或类，经过

一些小的修改或不做修改，可在目前的项目中重用。这种类型的重用对于在某些特定的应用领域开发软件的组织相当普遍，例如银行储蓄或空中交通控制系统。这些组织可通过建立未来可能重用的设计组件储存库并鼓励设计者去重用它们，也许可以每重用一次奖励一元钱，从而促进这种类型的重用。这种类型的重用有两个好处，尽管它也可能受到限制。首先，被测试模块的设计被结合到产品中，因此整个产品的设计可以进行得更快，也可能比从头开始设计整个产品有更高的质量。其次，如果一个模块的设计可以重用，那么该模块的实现很可能也可以重用，即便不是重用实际的代码，也至少是重用概念性的东西。

这种方法可扩展为库重用，如图 8-2a 所示。库是一组相关的可重用程序的集合，例如，科学计算软件的开发者很少自己编写程序来完成像矩阵变换或找到特征值这样的通用任务，而是购买像 LAPACK 3.0 这样的科学计算程序库［Anderson et al., 1999］。然后只要可能就在未来的软件中使用该科学计算程序库中的程序。随着面向对象范型的日益普及，科学计算软件的类库也已经开发出来，例如 LAPACK++［2000］。

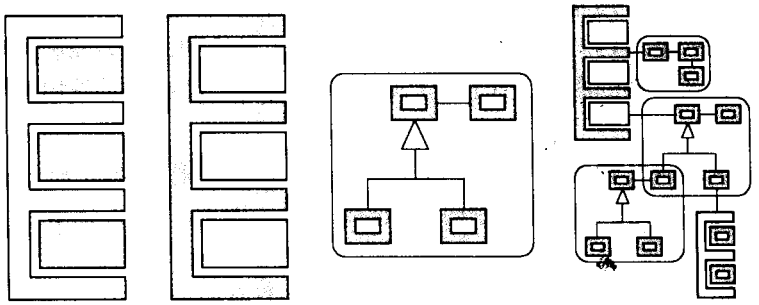



图 8-2　四种类型的设计重用的图示，阴影表示下列范围内的设计重用：a）一个库或工具包；b）一个框架；c）一个设计模式；d）包含框架、工具包和设计模式的软件体系结构

另一个例子是图形用户界面库。不需要从头开始写 GUI 方法，使用 GUI 类库或**工具包**（toolkit）将更方便，工具包是能处理 GUI 的每个特征的一组类。有许多这种类型的 GUI 工具包，包括 Java Abstract Windowing Toolkit［Flanagan, 2002］。

库重用的问题是库通常以一组可重用的子程序的形式存在，而不是可重用的设计。工具包通常也只是促进代码重用；而不是设计重用。然而，当使用面向对象范型时，可通过浏览器（也就是显示继承树的 CASE 工具）的帮助来缓解这个问题。设计者可遍历库的继承树，检查各种类的域，并确定哪个类适合于当前的设计。

库和工具包重用的关键特征是，如图 8-2a 所示，设计者负责整个产品的控制逻辑。库或工具包通过提供结合进产品的特定操作的部分设计，对软件开发过程提供帮助。

另一方面，应用框架与库或工具包相反，它提供控制逻辑，开发者负责特定操作的设计，这将在下一节中讨论。

### 8.5.2　应用框架

图 8-2b 所示的**应用框架**结合了一个设计的控制逻辑。当重用框架时，开发者需要设计所建产品的特定应用操作。特定应用操作插入的地方通常称为**热点**（hot spots）。

今天，**框架**（framework）一词通常指的是面向对象的应用框架。例如，在［Gamma, Helm, Johnson, and Vlissides, 1995］中，框架被定义为“一组互操作类，它们构成了一类特定软件的可重用设计。”然而请注意8.3.1节中图8-1所示的Raytheon导弹系统部的实例研究与图8-2b相同，换句话说，20世纪70年代的Raytheon COBOL程序逻辑结构是今天的面向对象应用框架的传统先驱。

应用框架的一个例子是用于编译器设计的一组类，设计小组只需提供适应特定语言的类、以及想要的目标机器，然后这些类被插入到框架中，如图8-2b中的白色方框所示。框架的另一个例子是用于软件控制ATM的一组类，其中设计者需要提供由银行网络的ATM机所提供的、用于特定银行服务的类。

重用一个框架比重用工具包更能加快产品的开发，原因有两个。首先，多数设计随着框架一起被重用，因而需要从头开始设计的部分就更少。其次，随着框架（控制逻辑）重用的那部分设计通常比操作更难设计，因此生成的设计质量也很可能比重用工具包时更高。就像库或工具包的重用一样，通常框架的实现也可重用。开发者需要使用名称和调用框架的协定，但这是很小的一笔花费。还有，生成的产品会很容易维护，因为控制逻辑已经在其他重用了应用框架的产品中得到测试，而且先前的维护者可能已经维护了重用相同框架的另一个产品。

IBM的WebSphere（较早前称为e-Components，最开始称为San Francisco）是一个用Java构建在线信息系统的框架。它利用了Enterprise JavaBeans，即为分布于网上的客户提供服务的类。

除了应用框架，还可以使用许多代码框架。第一个商用的成功的代码框架是MacApp，它是在Macintosh上编写应用软件的框架。微软的基本类库（Microsoft Foundation Class Library，MFC）是在基于Windows的应用程序中创建GUI框架的大型集合。MFC应用可以完成标准的视窗操作，例如移动窗口和调整窗口大小、通过对话框处理输入以及处理像鼠标点击或菜单选择这样的事件。Borland公司的可视化组件库（Visual Component Library，VCL）是对ObjectWindows Library（OWL）的更新，与MFC的功能类似。然而，VCL是完全面向对象的。

下面我们讨论设计模式。

### 8.5.3　设计模式

Christopher Alexander（参见下面的“如果你想知道”部分）说过，“每个模式描述一个在我们的环境中反复出现的问题，并描述该问题的解决方案的核心内容，以这种方式你可以使用这个解决方案无数次，却不用两次以同样的方式做这件事。”［Alexander et al., 1977］。尽管他是在构件和其他结构化对象的范畴内描述模式的，但他的观点同样适用于设计模式。

**如果你想知道**

在面向对象的软件工程领域中，最有影响的个人之一是Christopher Alexander，他是世界闻名的建筑师，但他坦言不懂对象或软件工程。在他的书中，特别是在［Alexander et al., 1977］中，他描述了一种建筑的模式语言，也就是为了描述城镇、建筑物、房间、花园等而使用的语言。他的想法被面向对象软件工程师们采用，并加以改造，特别是被称为

“四人组”的一组软件工程师（Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides）所采用。他们在设计模式方面的畅销书［Gamma、Helm、Johnson and Vlissides, 1995］使 Alexander 的想法被面向对象的业内人士广泛接受。

模式也在其他情况下出现，例如，到达一个机场时，飞行员需要知道合适的降落模式，也就是有关方向、高度以及将飞机停在正确跑道上所需旋转角度的顺序。还有，制衣模式是生产一件特定服装时可重复使用的样式组。模式的概念本身绝不新奇，新奇的是模式在软件开发，特别是设计上的应用。

设计模式是通常的设计问题的解决方案，这类问题以一组交互类的形式出现，需要由用户根据需要定制这些交互类以形成专门的设计，如图 8-2c 所示。带阴影的方框通过线连接起来，代表交互类。阴影方框内的白色方框代表为进行专门的设计而必须定制的那些类。

为理解模式如何有助于软件开发，请考虑下面的例子。假设一个软件组织希望建造一个窗口小部件（widget）生成器，它可帮助开发者建造一个图形用户界面，而不用从头开始开发各种**窗口小部件**（例如窗口、按钮、菜单、滑块和滚动条），开发者可以使用由窗口小部件生成器（它定义了要在应用程序中应用的窗口小部件）创建的这组类。

问题是应用程序（以及窗口小部件）可能会在许多不同的操作系统（包括 Linux、Mac OS 和 Windows）下运行，窗口小部件生成器需要支持所有这三个操作系统。然而，如果窗口小部件生成器刻板地将一个特定系统下运行的例程编进一个应用程序中，那么未来将很难修改该应用程序，只能用另一个在不同的操作系统下运行的例程来代替它。例如，假设应用程序是在 Linux 系统下运行的，如果现在需要在 Mac OS 系统下运行这个应用程序，则 create Linux menu 的每个实例都必须由 create Mac OS menu 来代替。对于一个大型的应用程序，从 Linux 到 Mac OS 这样的转换将是艰巨的，并容易出错。

解决办法是以下面的方式设计窗口小部件生成器，使应用程序与特定的操作系统之间去耦，这可以通过使用 Abstract Factory（抽象工厂）设计模式来实现［Gamma, Helm, Johnson, and Vlissides, 1995］。图 8-3 表示了这样的设计，在图中，抽象类和它们的抽象（虚拟）方法的名称以斜体表示。（抽象类是不能用具体例子说明的类，尽管可以使用它作为一个基类。通常情况下，它至少包含一个抽象方法。）在图 8-3 的顶端是抽象类 ***Abstract Widget Factory***（如 7.7 节所讲，UML 惯例是类以粗体字形式出现，每个单词的第一个字母大写）。这个抽象类包含大量的抽象方法，为明确起见，这里只显示了两个：*create menu* 和 *create window*。从图中再往下看，**Linux Widget Factory**、**Mac OS Widget Factory** 和 **Windows Widget Factory** 是 ***Abstract Widget Factory*** 的具体子类。每个类包含特定的方法，用来创建在给定操作系统下运行的窗口小部件。例如，**Linux Widget Factory** 内部的 create menu 将产生一个在 Linux 下运行的菜单对象。

对于每个窗口小部件还有抽象类，这里给出了两个，***Abstract Menu*** 和 ***Abstract Window***。每一个都有具体子类，每个子类对应上述三种操作系统中的一种。例如，**Linux Menu** 是 ***Abstract Menu*** 的一个具体子类，具体子类 **Linux Widget Factory** 内部的 create menu 方法能够生成一个类型为 **Linux Menu** 的对象。

为创建一个窗口，应用程序内部的 **Client** 对象不仅需要发送一个消息给 ***Abstract Widget Factory*** 的 *create window* 方法，而且多态能够确保生成正确的窗口小部件。假设应用程序需要在 Linux 下运行，首先，生成一个类型（或类）为 **Linux Widget Factory** 的 Widget

Factory对象。然后把一个将Widget Factory作为参数传递的、送给**Abstract Widget Factory**的create window抽象方法的消息，解释为一个送给具体的子类**Linux Widget Factory**内部的create window方法的消息。create window方法按顺序发送一个消息来创建一个**Linux Window**，如图8-3最左边的垂直虚线所示。

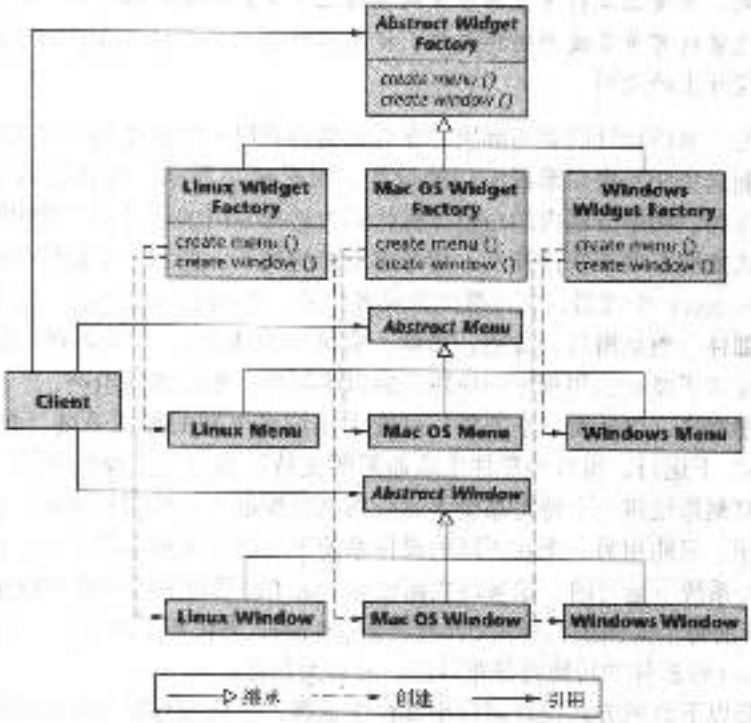



图8-3　图形用户界面工具包的设计。抽象类和它们的虚拟函数的名称以粗体字表示

这个图的重要方面是应用程序内部的**Client**和窗口小部件生成器之间的三个接口，**Abstract Widget Factory**类、**Abstract Menu**类和**Abstract Window**类都是抽象类。这些接口都不是为那一个操作系统而专门设计的，因为抽象类的方法是抽象的（在C++中是虚拟的）。这样，图8-3的设计实际上是将操作系统与应用程序去耦了。图8-3的设计是图8-4所示的**Abstract Factory**模式的一个实例。为应用这种模式，专门的类替代了像**Concrete Factory 2**和**Product B3**这样的普通的名称。这就是为什么在图8-2c中，设计模式的图示在阴影方框内部包含有白色方框，这白色的方框代表了在设计中重用该模式时需要提供的细节。

模式之间可以进行交互，如图8-2d所示，中间的模式的左下模块也是一个模式。[Gamma, Helm, Johnson, and Vlissides, 1995]中对文档编辑器的实例研究包含8个不同的交互模式，这是实际中的情况，很少有产品的设计只包含一个模式。

与工具包和框架一样，如果重用设计模式，那么该模式的实现也可能被重用。另外，模式分析有助于面向对象的分析[Fowler, 1997]。最后，除了模式，还有反模式，下面的"如果你想知道"部分将对此进行描述。

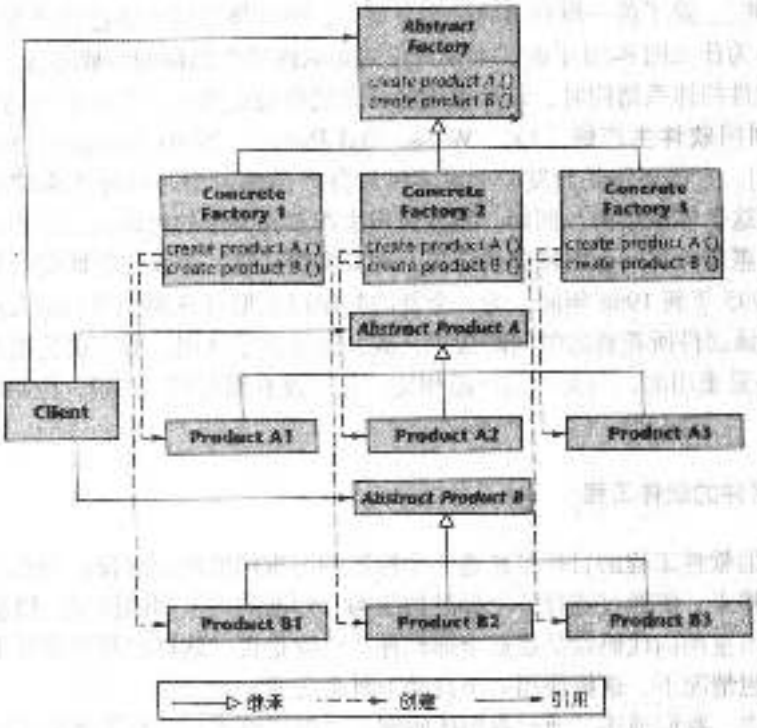



图 8-4 Abstract Factory 模式。抽象类和它们的虚拟函数的名称以斜体字表示

**如果你想知道**

反模式是一种能够引起项目失败的做法，例如“分析瘫痪”（在分析阶段花费了太多的时间和努力）或设计一个面向对象的产品时只使用一个对象做几乎全部的工作。写第一本反模式的书的主要动机是，大约有 1/3 的软件项目被取消了，2/3 的软件项目成本超出了 200%，80%以上的软件项目注定会失败 [Brown et al., 1998]

### 8.5.4 软件体系结构

一个大教堂的结构可以描述为罗马式的、哥特式的或巴洛克式的。类似地，软件产品的结构也可以描述为面向对象的、管道和过滤器（UNIX 组件）的或客户－服务器的（有一个中心服务器为网络或客户计算机提供文件存储和计算设施）。图 8-2d 所示的结构包含有一个工具包、一个框架和三个设计模式。

因为应用于产品整体上的设计，因此**软件体系结构**领域面临各类设计事项，包括根据它的组件进行的产品的组织、产品级的控制结构、通信和同步问题、数据库和数据访问、组件的物理分发、性能及设计替代的选择 [Shaw and Garlan, 1996]。这样，软件体系结构比起设计模式来说，是范围更广泛的概念。

事实上，Shaw 和 Garlan 指出 [1996]，“抽象地说，软件体系结构涉及用来建造系统的要素的描述，还涉及这些要素之间的相互作用、指导它们的组合的模式和对这些模式的限

制”［重点强调］。除了前一段列出的许多事项外，软件体系结构还把模式作为一个子域包含进来，这就是为什么图 8-2d 中的三个设计模式是软件体系结构的组成部分。

当重用软件的体系结构时，设计重用的许多优势会更强了。实际中实现体系结构重用的一种方式是利用**软件生产线**［Lai，Weiss，and Parnas，1999；Jazayeri，Ran，and van der Linden，2000］。它的思路是开发一个对多种软件产品都通用的软件体系结构，并在开发新产品时实例化这个体系结构。例如，惠普公司生产多种类型的打印机，并不断有新产品开发出来。现在，惠普具有一个固件体系结构，用其实例化每一个新的打印机机型，结果十分有效。例如在 1995 年到 1998 年间，为一个新的打印机机型开发固件所需的人小时数下降了四成，而且开发该固件所花费的时间降低了三成，还增进了重用。对于更近期的打印机，固件中 70％的组件是重用的，与前面的产品相比，几乎没有做修改［Toft，Coleman，and Ohta，2000］。

### 8.5.5　基于组件的软件工程

**基于组件的软件工程**的目标是建造一个标准的可重用组件的集合。那么，不是每次再去从头做起，在将来，将通过选择一个标准的结构和标准的可重用的框架，以及在框架的热点处插入标准的可重用的代码制品建造全部软件。也就是说，软件产品将通过组合可重用的组件来建造。理想情况下，这将使用一个自动工具来完成。

在这一章中，我们描述了通过重用代码制品、设计模式以及软件体系结构增加的许多优势。因此，实现基于组件的软件工程将能够解决软件开发中的许多问题。特别地，它将导致软件生产力和产品质量的成倍增加，并且减少投放市场的时间以及维护工作。

遗憾的是，目前重用的水平还远没有达到这个雄心勃勃的目标。另外，基于组件的软件构建也还面临许多挑战，包括定义、标准和组件的修补。然而，许多软件中心的研究人员正在积极工作，试图尽早达成此目标［Heineman and Councill，2001］。

## 8.6　重用和交付后维护

促进重用的传统原因是它可以缩短开发过程，例如，一些主要的软件组织正试图将开发一个新产品的时间减半，那么在众多的努力中，重用是一个主要策略。然而，正像图 1-3 所反映的，在开发产品上每花费 1 美元，那么在维护该产品上将花费 2 美元或更多。因此，重用很重要的第二个原因是它可减少维护的时间和花费。事实上，重用对交付后维护的影响比对开发的影响大。

现在假设产品的 40％包含有从以前的产品中重用的组件，而且这种重用贯穿于整个产品中，也就是说，规格说明文档的 40％、设计的 40％、代码制品的 40％、用户手册的 40％等由重用的组件组成。遗憾的是，这并不意味着开发整个产品的时间将是不重用时所需时间的 40％。首先，这些重用的组件中有一些需要加以裁剪以适应新产品，假设重用组件的 1/4 进行了修改，如果需要修改一个组件，那么那个组件的文档也需要修改，进一步地，修改了的组件需要测试。其次，如果一个代码制品没有经过修改就重用，那么不需要对该模块进行单元测试，但仍需要对该模块进行集成测试。所以，即使产品的 30％包含有未经修改的重用组件，另有产品的 10％ 经过修改后重用，那么在最好的情况下，开发整个产品所需的时间只减少了大约 27％［Schach，1992］。假定，如图 1-3a 所示，软件预算的 33％用于开发，

那么如果重用降低了 27% 的开发成本，则产品的整个成本在其 12 到 15 年的生命周期里由于重用也只减少了 9%，表 8-1 中反映了这一点。

**表 8-1　假设新产品的 40% 包含有重用的组件，其中 3/4 的组件未经修改即被重用时，成本节省的平均百分比**

| 活　动 | 在生命周期里占产品整个成本的百分比 | 因为重用，在生命周期里节省的百分比 |
|---|---|---|
| 开发 | 33% | 9.3% |
| 交付后维护 | 67% | 17.9% |

类似但冗长的论证可用于软件过程的维护部分［Schach，1994］。在上一段的假设下，重用对维护的影响是使整个成本大约节省了 18%，如表 8-1 所示。很明显，重用的主要影响是对维护，而不是开发，原因在于重用的组件通常经过了良好设计、全面测试并全面地形成文档，因此简化了所有三种类型的交付后维护。

如果给定产品的实际重用率比本节中假设的低（或高），那么重用的效益也将不同。但整体结果仍是相同的：重用对维护的影响多于对开发的影响。

下面我们讨论可移植性。

## 8.7　可移植性

软件不断增长的成本要求找到一些方法来节约成本，一种方法是确保整个产品能够容易地在各种不同的硬件和操作系统上运行。写出此类产品的一部分成本可通过出售能在其他计算机上运行的版本得到补偿。但编写可移植软件的最重要的原因是，大约每 4 年客户组织将购买新硬件，它所有的软件将被转换到新硬件上运行。与从头开始编写一个新产品相比，如果一个产品能够很便宜地适应在新的计算机上运行，则该产品是“可移植的”［Mooney，1990］。

可移植性可以更精确地定义如下：假设产品 $P$ 由编译器 $C$ 进行编译，然后运行在**源计算机**上，源计算机的硬件配置为 $H$，操作系统为 $O$。产品 $P'$ 与产品 $P$ 的功能相同，但必须由编译器 $C'$ 进行编译，并运行在**目标计算机**上，目标计算机的硬件配置为 $H'$，操作系统为 $O'$。如果把 $P$ 转换为 $P'$ 的成本比从头开始编写 $P'$ 的成本少很多，则产品 $P$ 是可移植的。

总的来说，由于不同的硬件配置、操作系统和编译器之间的不兼容性，移植软件的问题就显得尤为重要，下面依次讨论这些不兼容性的各方面。

### 8.7.1　硬件的不兼容性

目前运行于硬件配置 $H$ 上的产品 $P$ 将要安装在硬件配置 $H'$ 上，表面上看这很简单，从 $H$ 的硬盘驱动器中拷贝 $P$ 到 DAT 磁带上，并传送给 $H'$ 即可，然而如果 $H'$ 使用 Zip 驱动器来备份，那么这将行不通，在 Zip 驱动器上不能读取 DAT 磁带。

现在假设产品 $P$ 的源代码物理拷贝的问题已经解决，也不能保证 $H'$ 可以解释 $H$ 创建的比特模式。现有许多不同的字符代码，最常用的是扩展二进制编码的十进制交换码（Extended Binary Coded Decimal Interchange Code，EBCDIC）和美国信息交换标准码（American Standard Code for Information Interchange，ASCII），7 比特 ISO 代码的美国版［Mackenzie，1980］。如果 $H$ 使用 EBCDIC，而 $H'$ 使用 ASCII，那么 $H'$ 将认为 $P$ 是无用信息。类似地，

通常 PC 机不能读取 Macintosh 格式的数据，反之亦然。

尽管造成这些不同最初是由于历史原因（也就是说，为不同的制造商独立工作的研究者们以不同的方式开发相同的东西），但保持这种区别也还存在着明确的经济原因。为明白这一点，请考虑下面的虚构情况。MCM 计算机制造商已经卖掉了上千台 MCM-1 型计算机，现在 MCM 公司希望设计、生产一种新计算机 MCM－2，并投入市场。MCM-2 型计算机比 MCM-1 在各方面都要强，成本却低很多。进一步假设 MCM-1 型计算机使用 ASCII 码，并有包含四个 9 比特字节的 36 比特字。现在 MCM 的主设计师决定 MCM-2 型计算机应使用 EBCDIC 码，并具有包含两个 8 比特字节的 16 比特字。那么销售人员只能告诉 MCM-1 的用户，MCM-2 型计算机比其他竞争者的相同款型机器便宜 35 000 美元，却需要花费 200 000 美元来转换现有的软件和数据，从 MCM-1 格式到 MCM-2 格式。从市场的角度考虑，需要确保新的计算机对于旧款型来说是可移植的，然后销售人员可以给现有的 MCM-1 客户指出，MCM-2 型计算机不仅比竞争者的相同款型机器便宜 35 000 美元，而且如果不听建议的顾客从其他生产商那里购买机器，除了多花费 35 000 美元以外，还将再花费 200 000 美元来将现有的软件和数据的格式转换为非 MCM 机器的格式。

从前面的虚构的情况转到现实情况中，至今为止最成功的计算机生产线是 IBM System/360－370 系列［Gifford and Spector，1987］。这个计算机生产线的成功很大程度上得益于机器间大规模地全兼容。运行在 1964 年生产的 IBM System/360 的 30 型计算机上的产品可以不经修改地运行在 2003 年生产的 IBM zSeries 990 计算机上。然而，在 OS/360 下，运行在 IBM System/360 的 30 型计算机上的产品如果需要在完全不同的 2003 年的机器（例如 Solaris 下的 Sun Fire 15K）上运行，则需要进行相应的调整，这种困难部分归结于硬件的不兼容性，但也部分归结于操作系统的不兼容性。

### 8.7.2　操作系统的不兼容性

任何两种计算机的作业控制语言（job control language，JCL）通常都有很大的不同，其中的一些区别是语法上的——例如执行可执行载入映像的命令在一种计算机上可能是@xeq，在另一种计算机上可能是//xqt，而在第三种计算机上可能是 .exc。当把一个产品向另一种不同的操作系统移植时，通过简单地从一种 JCL 转换到另一种 JCL，可以直接解决语法上的区别。但另一些区别则更严重些。例如，一些操作系统支持虚拟内存，假设某个操作系统允许产品的大小超过 1024 MB，但实际分配给特定产品的主存区域可能只有 64 MB，那么用户的产品被分区为大小为 2048 KB 的数页，这些页中只有 32 页可以同时存在于主存中，剩下的页则存储在磁盘中，根据需要由虚拟内存的操作系统将它们交换进来或交换出去。这样的结果是编写产品时对产品的大小没有实际的限制。但是，如果一个产品在虚拟内存的操作系统中成功实现，需要将它移植到另一个对产品的大小有物理限制的操作系统时，则需要重写整个产品，并使用覆盖技术重新链接以确保没有超过产品大小的限制。

### 8.7.3　数值计算软件的不兼容性

当产品由一台机器移植到另一台机器，或甚至是使用一个不同的编译器编译过时，执行运算的结果都可能不同。在一个 16 比特的机器上，也就是在字的大小为 16 比特的计算机上，一个整数通常以一个字（16 比特）来表示，一个双精度整数由两个相邻的字（32 比特）

表示。遗憾的是，一些语言实现不包含双精度整数。例如，标准的 Pascal 语言不包含双精度整数，因此在一个编译器－硬件－操作系统的配置上执行完好的产品中，使用 32 比特来表示 Pascal 整数，而当它被移植到只能以 16 比特来表示整数的计算机上时，程序可能会运行不正常。显然，用浮点数（**real** 类型）表示大于 $2^{16}$的整数的解决方案并不起作用，因为整数是精确表示的，而浮点数通常只是使用尾数和指数来近似得出的。

这个问题可以在 Java 中得到解决，因为它的 8 种基本数据类型都有详细的规定。例如，类型 **int** 总是实现为一个 32 比特有符号整数的补码，而类型 **float** 总是占据 32 比特，并符合 ANSI/IEEE 的有关浮点数的 754 标准［1985］。因此在 Java 中不会出现数值计算在每种目标硬件－操作系统上能否正确完成的问题（为进一步了解 Java 的设计，参见下面的“如果你想知道”部分）。然而，在除了 Java 以外的语言中完成数值计算时，重要的（通常也是困难的）是确保数值计算在目标硬件－操作系统上正确地完成。

**如果你想知道**

1991 年，Sun Microsystem 公司的 James Gosling 开发了 Java。在开发这种语言时，他不断地注视着办公室窗外的一棵巨大的橡树。事实上，他因为经常这样做，而想把这门新语言称为“橡树”。然而，Sun 公司无法接受这个名字，因为它不能作为注册商标，而如果没有注册商标，Sun 将失去对这种语言的控制。

为寻找能够作为商标的名称，还容易记住这名称，Gosling 的小组想到了 Java。在 18 世纪，大多数进口到英国的咖啡生长在 Java，Java 是荷属东印度群岛中人口最稠密的岛屿（现在的印度尼西亚）。结果现在 Java 成了咖啡的俚语，软件工程中第三流行的饮料。遗憾的是，两大碳酸盐可乐饮料的名称已经是注册商标了。

为了理解 Gosling 为何设计 Java，有必要了解他发现的 C++ 的弱点的根源，为此，我们回到 C 语言，C++ 的父语言。

1972 年，编程语言 C 由 Dennis Ritchie 在 AT&T 公司的贝尔实验室（现在是朗讯科技公司的）开发而成，在系统软件中应用。该语言相当灵活，例如，它允许在指针变量上进行运算，也就是说，在用来存储内存地址的变量上进行运算。在大多数程序员看来，这意味着相当大的危险，由此得来的程序将相当不安全，因为在计算机里可以向任何地方传递控制。还有，C 不以这样的方式包含数组，而是以一个指针来指向数组的第一个元素所在的地址。结果，数组下标超出范围的概念并不是 C 所固有的，这是不安全可能的进一步的根源。

这些和其他的不安全因素在贝尔实验室里却不成问题，毕竟 C 是由贝尔实验室的一个经验丰富的软件工程师设计的，而使用它的是另一些经验丰富的软件工程师。这些专业人员能够以安全的方式利用 C 的强大和灵活的特性。C 语言设计的一个基本理念是使用 C 的人清楚地知道他或她在做什么。不太熟悉 C 或经验不太丰富的程序员使用 C 时出现的软件故障不应归罪于 AT&T，那时还从来没有像现在这样把 C 作为通用编程语言而广泛使用的趋势。

随着面向对象范型的出现，发展出一些基于 C 的面向对象的编程语言，包括 Object C、Objective C 和 C++。这些语言背后的思想是在当时流行的编程语言 C 内部嵌入面向对象的结构。对于程序员来说，学习一种语言，它基于你所熟悉的语言，是否比学习一种全新的语法更容易些这个问题曾备受争论。然而，许多基于 C 的面向对象的语言中只有一种被广泛接受，它就是 C++，由也是 AT&T 公司贝尔实验室的 Bjarne Stroustrup 开发。

人们一度认为 C++ 成功背后的原因是 AT&T 的强大经济实力。然而，如果在宣传编程

语言中，公司的规模和商业实力是相关因素，那么我们今天使用的应是由 IBM 公司开发和宣传的 PL/I 语言，而实际上，尽管 IBM 的名气很大，但 PL/I 还是没有发展起来。

C++ 成功的真正原因是，C++ 是 C 语言的真正超集，也就是说，与其他基于 C 的面向对象的编程语言不同，几乎任何 C 程序都是可用的 C++ 程序，因此，各公司都意识到，它们不用对已有的 C 软件做任何修改就可以将 C 转成 C++，它们可以不破坏任何东西就将结构化范型升级为面向对象范型。在 Java 文献中经常遇到的声明是："Java 与 C++ 可类比。"它的含义是，如果 Stroustrup 当初像 Gosling 一样聪明，那么 C++ 将转变成 Java。相反地，如果 C++ 不是 C 的真正超集，它将步其他所有基于 C 的面向对象编程语言的后尘，基本上会消失。只有在 C++ 已成为流行的语言之后，才针对前面提到的 C++ 的弱点设计 Java。Java 不是 C 的超集，例如，Java 没有指针变量，因此，这样说可能更准确些："Java 能做到 C++ 不大可能做到的事。"

最后，像其他的每种编程语言一样，Java 也有它自己的弱点，认识到这一点很重要。另外，在一些领域里（例如存取规则），C++ 优于 Java [Schach, 1997]。在未来的几年里是 C++ 继续保持面向对象编程语言的主导地位，还是由 Java 或其他的什么语言来取代它，这将是一件引人注目的大事。

### 8.7.4　编译器的不兼容性

如果产品以一种很少有编译器可用的语言来实现，那么很难实现可移植性。如果产品以一种诸如 CLU [Liskov, Snyder, Atkinson, and Schaffert, 1977] 的专用语言实现，而目标计算机上没有该语言的编译器，那么就有必要用另一种不同的语言来重写它。另一方面，如果产品是以一种流行的语言实现的，例如 COBOL、FORTRAN、Lisp、Pascal、C、C++ 或 Java，则目标计算机上很可能有该语言的编译器或解释程序。

假设一个产品是以一种流行的高级语言（例如标准 FORTRAN 语言）写成的，从理论上说，把该产品从一个机器上移植到另一个机器上是没有问题的，毕竟 FORTRAN 语言是标准的。然而遗憾的是，事实并非如此。实际中并没有纯粹标准的 FORTRAN，尽管存在 ISO/IEC FORTRAN 标准，称为 Fortran 95 [ISO/IEC 1539-1, 1997]，编译器的编写者也不大可能完全依照它（要进一步了解 Fortran 95 的名称，参见下面的"如果你想知道"部分）。例如，可以做出决定支持在 FORTRAN 中通常找不到的附加特性，以便市场部推销一种"新型的扩展 FORTRAN 编译器"。相反地，一个微机编译器可能不是完全 FORTRAN 实现的。还有，如果完成一个编译器有最后期限，管理层可能会决定拿出一个非完全的实现，在以后的修订版中再试图支持全标准。假设源计算机上的编译器支持 Fortran 95 的一个超集，而目标编译器是一个标准 Fortran 95 的实现。当在源计算机上实现的产品移植到目标计算机时，产品中使用超集的非标准 Fortran 95 结构的任何部分都必须重新编写代码。因此，为确保可移植性，程序员应只能使用标准 FORTRAN 的语言特性。

**如果你想知道**

编程语言的名称以缩写形式表示时用大写字母拼写，例子包括 ALGOL（ALGOrithmic Language）、COBOL（COmmon Business Oriented Language）和 FORTRAN（FORmula TRANslator）。与之相对，所有其他的编程语言都以一个大写字母开始，名称中的其余字母以小写形式拼写，

例如 Ada、C、C++、Java 和 Pascal。Ada 不是缩写，该语言是以 Ada，Lovelace 伯爵夫人（1815—1852）的名字命名的，她是诗人 Lord Alfred Byron 的女儿。以她的名字命名是由于她是世界上第一个程序员，给 Charles Babbage 的差分计算机器编写程序。Pascal 也不是一个缩写词，该语言是以法国数学家和哲学家 Blaise Pascal（1623—1662）的名字而命名的。关于 Java 的名称由来，我相信你已经看过了 8.7.3 节的“如果你想知道”部分。

我非常报歉和非常伤心地告诉您，FORTRAN 标准委员会决定，从 1990 年版开始，该语言将写成 Fortran。由于我已经习惯于写成 FORTRAN，本书仍这样做，但这并不表示我有意不遵守标准规定。

早期的 COBOL 标准是由数据系统语言协会（COnference DAta SYstems Languages，CODASYL）开发的，该协会是由美国的计算机生产商、政府及个人用户组成的委员会。ISO/IEC 第 22 小组委员会的第 1 联合技术委员会现在负责 COBOL 标准［Schricker，2000］。然而 COBOL 标准不推崇可移植性。每个 COBOL 标准有 5 年的正式生命期，但每个后续的标准并不必是它先辈的超集。同样有点麻烦的是许多特性留给了单个的实现者，子集被称为“标准 COBOL”，而且在扩展该语言形成的超集上没有什么限制。

OO-COBOL，目前的 COBOL 标准语言草案，是完全面向对象的［ISO/IEC 1989，2002］。相反地，Fortran 95 只是**基于对象**的［Wegner，1992］，也就是说，在 Fortran 95 中不能实现继承和类。因此，Fortran 95 对象是自成一体的，它们不是类的实例。下一个 FORTRAN 标准，Fortran 2000 将是完全面向对象的。按照编写的时间安排，公布 Fortran 2000 标准的目标日期是 2004 年底。

美国国家标准学会（American National Standards Institute，ANSI）已经通过了编程语言 C 的一个标准［ANSI X3.159，1989］。ISO 于 1990 年通过了该标准。大多数 C 编译器完全符合原来的语言规格说明［Kernighan and Ritchie，1978］。这是因为几乎所有的 C 编译器的编写者都使用可移植 C 编译器的标准前端 pcc［Johnson，1979］，结果大多数编译器能接受的语言是相同的。通常，容易把 C 的产品从一个实现移植到另一个实现中。用于辅助实现 C 可移植性的一个工具是 `lint` 处理机，它可用于确定基于实现的特性和产品移植到目标计算机时可能会导致困难的结构。然而，`lint` 只检查语法和静态语义，因此它不是防止错误操作的，但是它有助于减少未来的问题。例如，在 C 中，把一个整数值分配给指针是合法的，反之亦然，但 `lint` 禁止如此。在一些实现中，整数的大小和指针的大小（比特数）是相同的，但在另一些实现中，它们的大小可能是不同的。这种在未来移植时可能出现的潜在问题可由 `lint` 进行标记，并通过记录冲突部分来避免。

各种国家标准委员会（包括 ANSI）在 1997 年 11 月都一致通过了 C++ 标准［ISO/IEC 14882，1998］，1998 年该标准得到了最后的批准。

到目前为止，真正成功的语言标准是 Ada 83 标准，收录在 Ada 参考手册中［ANSI/MIL-STD-1815A，1983］。（关于 Ada 的背景信息请参见上一个“如果你想知道”部分。）到 1987 年底，Ada 的名称已成为美国政府的 Ada 联合程序办公室（Ada Joint Program Office，AJPO）的注册商标。作为该商标的拥有者，AJPO 规定，Ada 的名称只可以在完全符合该标准的语言实现中使用，明确地禁止使用子集和超集。人们为验证 Ada 编译器建立了一种机制，而且只有成功地通过了验证过程的编译器才被称为 Ada 编译器。这样，该商标作为一种强制执行标准的方法而使用，因而具有可移植性。

现在 Ada 的名称不再是一个商标，该标准的强制性通过另一个不同的机制实现。没有经过验证的 Ada 编译器几乎没有市场。这样，强大的经济驱动力促使 Ada 编译器的开发者们尽力使他们的编译器能通过验证，因而也符合 Ada 标准。这已应用于 Ada 83［ANSI/MIL-STD-1815A，1983］和 Ada 95［ISO/IEC 8652，1995］两种编译器。

因为 Java 是完全可移植的语言，因此标准化该语言很重要，而且确保严格遵守该标准也很重要。Sun Microsystems 公司与 Ada 联合程序办公室一样，正使用法律系统以实现标准化。如 8.7.3 节中的“如果你想知道”部分所提到的，Sun 为它的新语言选择了一个具有版权的名字，看起来好像 Sun 将强制执行它们的版权，并以法律手段打击盗版者，毕竟，可移植性是 Java 最强大的特性之一。如果允许存在 Java 的多个版本，将破坏 Java 的可移植性，只有每个 Java 编译器同样处理每个 Java 程序，Java 才能实现真正的可移植性。为影响公众的观念，1997 年 Sun 开展了一项名为“纯粹 Java”的广告行动。

Java 的 1.0 版本在 1997 年初公布，作为对注释和评论的回应，之后出现了一系列修订版，目前最新的版本是 Java J2EE（Java 2 平台，企业版）。Java 的这种逐步求精的过程还将继续。当该语言逐渐稳定之后，很可能像 ANSI 或 ISO 这样的标准组织将公布一个标准草案，并从全世界征求反馈。这些反馈将用来形成正式的 Java 标准。

## 8.8　为什么需要可移植性

看到在移植软件上的这么多阻碍，读者很可能想知道是否值得去移植软件。8.7 节中关于可移植性的讨论说的是，把产品移植到另一个不同的硬件－操作系统配置上，软件的成本可能因此而得到部分补偿。然而，不太可能出售该软件的多个变种版。应用可能是非常专业化的，而且没有其他的客户会需要该软件。例如，为一个大型汽车出租公司编写的管理信息系统对其他的汽车出租公司的运营来说可能是不适用的。作为选择，软件本身可以给客户一个竞争优势，而且出售产品的副本无异于经济自杀。考虑所有这些因素，在设计产品时使产品具有可移植性是否是在浪费时间和金钱呢？

答案当然是“不”。可移植性重要的主要原因是，软件产品的生存期通常比第一次为之编写软件的硬件的生存期更长。好的软件产品可有 15 年或更长的生存期，而硬件每隔 4 年就将更新一次。所以，好的软件产品可在它的整个生存期间，在三个或更多的不同硬件配置上实现。

解决这个问题的一种方式是购买向上兼容的硬件。仅有的费用是硬件的成本，软件不需要变。然而在一些情况下，可能从经济的角度看将产品移植到完全不同的硬件上更合理，例如，产品的第一个版本可能 7 年前已经在大型主机上实现了，尽管有可能购买一个新的大型机，软件产品不加改变即可在其上运行，但在一个个人计算机网络上实现该产品的多个拷贝，每个用户的桌面上有一个，这样可能会更便宜些。在这个例子中，如果软件已经以某种支持可移植性的方式编写出来，那么将该产品移植到个人计算机网络中则更经济些。

但还有另外一些类型的软件，例如，许多给个人计算机编写软件的公司通过出售 COTS 软件的多个副本来赚钱。例如，电子数据表软件包的利润是很小的，不能填补开发的成本。为得到利润，可能需要出售 50 000（甚至是 500 000）个副本，超过了这个数字之后，额外的销售额就是纯利润了。所以，如果该产品可轻松地移植到其他类型的硬件上，则会赚更多的钱。

当然，对于所有软件来说，产品并不只是代码，还有文档，包括用户手册。把电子数据

表软件包移植到其他的硬件上意味着还要修改文档，这样，可移植性还意味着能够很容易地修改文档，以反映目标配置，而不是从头开始重写一个新的文档。如果一个被人熟悉的已存在的产品移植到新的计算机，它所需的训练自然比重新编写一个全新产品所需的训练要少得多。因此，可移植性是受到鼓励的。

下面描述实现可移植性的技术。

## 8.9　实现可移植性的技术

实现可移植性的一种方式是禁止程序员使用在移植到另一个计算机时可能引起问题的构造。例如，一个明显的原则看似如下：以一种高级编程语言的标准版本来编写所有的软件。但如何编写一个可移植的操作系统呢？毕竟，一个操作系统如果没有一些汇编代码是不可想像的。类似地，一个编译器需要为特定的计算机生成目标代码，因此，不可能完全避免所有依赖于实现的组件。

### 8.9.1　可移植的系统软件

一个更好的技术是隔离任何必须依赖于实现的程序段，而不是禁止所有依赖于实现的特性方面——尽管这可以避免重新编写几乎所有的系统软件。这项技术的一个例子是建造 UNIX 操作系统的方式［Johnson and Ritchie，1978］。该操作系统大约有 9000 行代码用 C 语言写成，剩下的 1000 行代码构成了内核，以汇编语言写成，因而对于每一个实现都需要重写。大约 1000 行 C 代码组成了设备驱动程序，这些代码也需要在每次移植时重写，然而其余的 8000 行 C 代码在从实现到实现的移植中基本保持不变。

增进系统软件的可移植性的另一个有用的技术是使用抽象层次（7.4.1 节）。例如，考虑工作站的图形显示程序。用户插入一个诸如 drawLine 这样的命令到他或她的源代码中，编译该源代码并将它链接到图形显示程序中。运行时，drawLine 可使工作站按用户要求在屏幕上画一条线。可以通过使用两级抽象来实现这一点。高一点的层次以高级语言写成，解释用户的命令并调用合适的低层模块来执行该命令。如果将该图形显示程序移植到一种新型的工作站上，则没有必要修改用户代码或该图形显示程序的高层代码。然而，需要重写该程序的低层模块，因为它们与实际的硬件接口，新工作站的硬件与原来实现软件的硬件不同。这项技术还成功地应用于移植符合 ISO－OSI 模型的七层抽象的通信软件［Tanenbaum，2002］。

### 8.9.2　可移植的应用软件

至于应用软件，与诸如操作系统和编译器这样的系统软件不同，它通常可以使用高级语言来编写。14.1 节指出，对实现的语言经常没有什么选择，如果可能选择语言的话，也应该基于成本－效益的分析来进行选择（5.2 节），在成本－效益分析中必须考虑的一个因素是对可移植性的影响。

在产品开发的每个阶段，可做出一些决策来确保产品更可移植。例如，一些编译器能够区分大写和小写字母，对于这样的编译器，This _ Is _ A _ Name 和 this _ is _ a _ name 是两个不同的变量，但其他的编译器却认为这两个名称是相同的。依赖于大写字母和小写字母来进行区分的产品，在进行移植时可能会产生不易察觉的错误。

就像对于编程语言没有什么选择一样，对于操作系统也没有什么选择。然而，如果可能的话，运行产品的操作系统应是主流系统之一，这对于 UNIX 操作系统是个有利的论据。UNIX 已在很大范围内的硬件上得到实现，另外，UNIX，或者更明确地说，类似 UNIX 的操作系统已在诸如 IBM VM/370 和 VAX/VMS 这样的大型机顶层操作系统中得以实现。对于个人计算机，Linux 是否能取代最普遍使用的 Windows 操作系统还有待观望。使用普遍实现的编程语言能促进可移植性，使用普遍实现的操作系统也能促进可移植性。

为实现将软件从基于 UNIX 的系统移植到另一个系统，开发了计算机环境的可移植操作系统接口（Portable Operating System Interface for Computer Environment，POSIX）[NIST 151，1988]。POSIX 还在大量非 UNIX 操作系统上得以实现，扩展了应用软件可正常移植的计算机数量。

语言标准在实现可移植性中发挥着重要作用，如果开发组织的编码标准规定只能使用标准构造，那么形成的产品将更具可移植性。为了这个最终目的，必须向程序员提供编译器支持的非标准特性的清单，并且指出谁的应用在没有得到上级管理层的批准时是禁止的。与其他合理的编码标准一样，可由机器检查这一点。

通过引入标准 GUI 语言，图形用户界面也同样变得可移植，相关例子包括 Motif 和 X11。GUI 语言的标准化是对 GUI 的重要性日益增加的响应，也是对人机界面可移植性的需求的结果。

此外，还应对构造产品所应用的操作系统与产品将移植到的任何未来的操作系统之间的不兼容性进行必要的规划。如果可能的话，应把操作系统调用局限在一个或两个代码制品上。在任何情形下，必须对每个操作系统调用仔细归档。操作系统调用的文档标准应假定下一个读代码的程序员对当前的操作系统不太熟悉，这通常是一个合理的假设。

提供安装手册形式的文档有助于未来的移植，该手册将指出产品的什么部分在移植时必须进行修改，哪些部分可能需要修改。在这两种情况下，必须提供详细的做什么及如何做的说明。最后，在其他的手册中已做的修改清单（例如用户手册或操作手册）也必须在安装手册中出现。

### 8.9.3　可移植的数据

数据可移植性的问题可能会很麻烦，8.7.1 节中已指出了硬件不兼容性的问题，但是，即使解决了这个问题，还存在着软件的不兼容性。例如，索引顺序文件的格式是由操作系统确定的，不同的操作系统通常使用不同的格式。许多文件的文件头要求包含诸如文件中的数据类型这样的信息。对于特定的编译器和创建文件的操作系统来说，文件头的格式几乎总是惟一的，当使用数据库管理系统时情况甚至会更糟。

移植数据最安全的方式是建造一个非结构化的（有序的）文件，然后可轻松地将该文件移植到目标计算机上。从这个非结构化的文件可以重新构建想要的结构化文件，但需要编写两个特定的转换程序，一个运行在源机器上，将原来的结构化文件转换成顺序文件形式，另一个运行在目标机器上，从移植过来的顺序文件重新构建该结构化的文件。尽管这个解决方案看起来相当简单，当需要在复杂的数据库模型之间进行转换时，这两个程序却并不简单。

在结束本章之前，我们总结一下重用和可移植性的长处及面临的障碍（表 8-2），并指出每个问题在哪个小节中讨论。

表 8-2　重用和可移植性的长处和障碍，以及讨论话题所在的小节

| 长　处 | 障　碍 |
| --- | --- |
| **重用** | |
| 较短开发时间（8.1 节） | NIH 综合症（8.2 节） |
| 较低开发成本（8.1 节） | 潜在质量问题（8.2 节） |
| 高质量软件（8.1 节） | 修补问题（8.2 节） |
| 较短维护时间（8.6 节） | 建造一个可重用组件的成本（机会重用）（8.2 节） |
| 较低维护成本（8.6 节） | 建造一个未来可重用组件的成本（系统重用）（8.2 节） |
| | 法律问题（仅合同软件）（8.2 节） |
| | 缺乏 COTS 组件的源代码（8.2 节） |
| **可移植性** | |
| 大约每 4 年软件必须移植到新的硬件（8.8 节） | 潜在的不兼容性： |
| 可以卖更多的 COTS 软件的拷贝（8.8 节） | 硬件（8.7.1 节） |
| | 操作系统（8.7.2 节） |
| | 数值计算软件（8.7.3 节） |
| | 编译器（8.7.4 节） |
| | 数据格式（8.9.3 节） |

## 本章回顾

8.1 节描述了重用，8.2 节描述了重用面临的各种障碍，8.3 节给出了 2 个重用实例研究，8.4 节分析了面向对象范型对重用的影响。设计和实现期间的重用是 8.5 节的主题，话题包括框架、模式、软件体系结构以及基于组件的软件工程。重用对交付后维护的影响在 8.6 节中讨论。

8.7 节讨论了可移植性，由硬件（8.7.1 节）、操作系统（8.7.2 节）、数值计算软件（8.7.3 节）或编译器（8.7.4 节）引起的不兼容会牵制可移植性。尽管这样，尽可能地使所有产品可移植还是非常重要的（8.8 节）。实现可移植性的方式包括使用流行的高级语言、隔离产品中不可移植的部分（8.9.1 节）和坚持语言标准（8.9.2 节）。

## 进一步阅读

本章中重用实例研究的进一步信息可在［Lanergan and Grasso，1984；Matsumoto，1984，1987；Selby，1989；Prieto-Díaz，1991；Lim，1994；Jézéquel and Meyer，1997；and Toft，Coleman，and Ohta，2000］中找到。在［Morisio，Tully，and Ezran，2000］中描述了在 4 个欧洲公司中成功地重用软件的例子。重用的管理在［Lim，1998］中有所描述。在［Isakowitz and Kauffman，1996］中描述了对象恢复和重用的一个研究计划。在［Barnes and Bollinger，1991］中描述了重用的成本效益，而且在［Caldiera and Basili，1991］中描述了确定未来重用的组件的方式。［Meyer，1996a］分析了面向对象范型如何促进重用，在［Fichman and Kemerer，1997］中有 4 个重用和对象技术的实例研究。重用的度量在［Poulin，1997］中有所描述。影响重用计划成功的因素在［Morisio，Ezran，and Tully，2002］中给出。在《IEEE Transactions on Software Engineering》杂志 2000 年 5 月刊中还可找到关于重用的更多的论文。

关于框架结构的好的信息源是［Lewis et al.，1995］。［D' Souza and Wills，1999］提供

了基于面向对象的框架结构和组件的开发方法。在［Fayad and Johnson，1999；Fayad and Schmidt，1999；and Fayad，Schmidt，and Johnson，1999］中可找到一系列优秀的关于面向对象的框架结构的文章。《Communications of the ACM》杂志2000年10月刊中有关于基于组件的框架结构的文章，包括［Fingar，2000，and Kobryn，2000］，它们描述了如何使用UML对组件和框架结构建模。在Fach［2001］中描述了通过框架和模式获得重用。

Alexander在［Alexander et al.，1977］中有关体系结构的内容中提出了设计模式的概念，模式理论的首次提出是在［Alexander，1999］中。软件设计模式的主要著作是［Gamma，Helm，Johnson，and Vlissides，1995］，较新的著作是［Vlissides，1998］。分析模式在［Fowler，1997］中有所描述。在［Prechelt，Unger-Lamprecht，Philippsen，and Tichy，2002］中介绍了评价设计模式文档对维护的影响的实验。［Brown et al.，1998］中描述了反模式。

关于软件结构的主要信息来源是［Shaw and Garlan，1996］。关于软件结构的更新的著作包括［Bosch，2000，and Bass，Clements，and Kazman，2003］。

［Jazayeri，Ran，and van der Linden，2000］，［Knauber，Muthig，Schmid，and Widen，2000］，［Donohoe，2000］以及［Clements and Northrop，2002］描述了软件生产线。在《IEEE Software》杂志2002年7/8月刊中包含了各种有关生产线的文章。

有关基于组件的软件工程的文章可以在《IEEE Software》杂志1998年9/10月刊中找到，包括［Weyuker，1998］，它讨论了基于组件的软件测试。［Brereton and Budgen，2000］讨论了基于组件的软件产品中的关键问题。关于基于组件的软件工程的经验的一些文章包括［Sparling，2000，and Baster，Konana，and Scott，2001］。［Heineman and Councill，2001］是一篇很受推荐的关于基于组件的软件工程的概述性文章。

［Mooney，1990］中介绍了实现可移植性的策略。［Johnson and Ritchie，1978］中讨论了C和UNIX的可移植性。

## 习题

8.1　请详细论述可重用性与可移植性之间的区别。

8.2　一个代码制品在新产品中未做修改即被重用。以何种方式能使这种重用降低产品的整体成本？以何种方式可使成本保持不变？

8.3　假设一个代码制品在重用时进行了一处修改，即将加法操作修改为减法操作。这微小的修改将会对习题8.2中的节省成本有何影响？

8.4　内聚对可重用性的影响是什么？

8.5　耦合对可重用性的影响是什么？

8.6　假设你刚加入了一个生产污染控制产品的大型公司，该公司有上百个软件产品，由95 000个不同的FORTRAN模块组成。公司雇用你来拟订一个规划，在未来的产品中尽可能多地重用这些模块，你将提出怎样的方案？

8.7　考虑一个图书自动循环系统。每本书有一个条形码，每个借书者有一张借书卡，上面也有一个条形码。当借书者想借走一本书时，图书管理员扫描该书和借书卡上的条形码，并在计算机终端上输入C。类似地，归还一本书时，图书管理员再次进行扫描，并输入R。图书管理员可以向书库中增加图书（+）或去掉图书（-）。借书者可以在

一台终端上确定出书库中特定作者的所有书籍（借书者输入 A= 之后，再输入作者的名字）、特定标题的所有书籍（输入 T= 之后，再输入标题）或者特定主题范围的所有书籍（输入 S= 之后，再输入主题范围）。最后，如果借书者想要一本目前已借出的书，图书管理员可以在该书上做个标记，当该书被归还时，将为申请过它的借书者保留起来（输入 H= 之后，再输入该书的书号）。请说明你如何确保可重用的代码制品比率高。

8.8　现在要求你建造一个产品来确定银行的储户报告书是否正确。需要的数据包括月初的余额、每张支票的号码、日期和数额、每笔储蓄的日期和数额及月末的余额。请说明你如何确保该产品中尽可能多的代码制品可在未来的产品中得到重用。

8.9　考虑一个自动柜员机（ATM）。用户将信用卡插入一个槽中，并输入 4 位数字的个人识别号（PIN）。如果 PIN 不正确，将弹出该信用卡。如果 PIN 正确，用户可以对最多四个不同的银行账号进行下面的操作：

a) 存钱，数额任意。将打印出一个收据，显示日期、存入的金额和账号。

b) 以 20 美元为单位，最多提取 200 美元（不能超过该数额）。除了现金，还将给用户打印出收据，显示日期、提取的金额、账号和提取后账户余额。

c) 确定账户余额。这将在屏幕上显示出来。

d) 在两个账户之间转移资金。被提取的账户中导出的金额还是不能超过最高限额。用户将得到一个收据，显示出日期、转移的金额和两个账号。

e) 退出。弹出信用卡。

请说明你如何确保该产品中尽可能多的代码制品可在未来的产品中得到重用。

8.10　开发者在软件生命周期中多早的时候能够发现阿丽亚娜 5 软件中的错误（8.3.2 节）？

8.11　8.5.2 节中说明了“20 世纪 70 年代的 Raytheon COBOL 程序逻辑结构是今天的面向对象应用框架的传统先驱”，对于技术发展来说这有什么含义？

8.12　请说明图 8-3 的设计模式中抽象类所扮演的角色？

8.13　请说明你如何确保图书自动循环系统（习题 8.7）尽可能地可移植。

8.14　请说明你如何确保检查银行储户报告书是否正确的产品（习题 8.8）尽可能地可移植。

8.15　请说明你如何确保习题 8.9 的自动柜员机（ATM）尽可能地可移植。

8.16　你的公司正在为治疗癌症所使用的一种新型激光开发一个实时控制系统，你负责编写两个汇编器模块。你将如何指导你的小组确保生成的代码尽可能地可移植？

8.17　你负责移植一个 750 000 行的 COBOL 产品到公司的新计算机中，你拷贝了源代码到新机器中，但当你试图编译它时，发现超过 15 000 个输入－输出语句都以非标准的 COBOL 语法写成，而这些非标准的 COBOL 语法在新的编译器中已被废弃，现在你将怎么办？

8.18　（学期项目）假设附录 A 中的 Ophelia's Oasis 产品是使用传统范型开发的，该产品的哪些部分可在未来的产品中得到重用？现在，假设该产品是使用面向对象范型开发的，则该产品的哪些部分可在未来的产品中得到重用？

8.19　（软件工程读物）你的导师将分发［Morisio，Ezran，and Tully，2002］的复印件，文章中列出的影响重用的因素在多大程度上反映在本章的重用实例研究中？

## 参考文献

[Alexander, 1999] C. ALEXANDER, "The Origins of Pattern Theory," *IEEE Software* **16** (September/October 1999), pp. 71–82.

[Alexander et al., 1977] C. ALEXANDER, S. ISHIKAWA, M. SILVERSTEIN, M. JACOBSON, I. FIKSDAHL-KING, AND S. ANGEL, *A Pattern Language*, Oxford University Press, New York, 1977.

[Anderson et al., 1999] E. ANDERSON, Z. BAI, C. BISCHOF, S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, 1999.

[ANSI X3.159, 1989] *The Programming Language C*, ANSI X3.159-1989, American National Standards Institute, New York, 1989.

[ANSI/IEEE 754, 1985] *Standard for Binary Floating Point Arithmetic*, ANSI/IEEE 754, American National Standards Institute, Institute of Electrical and Electronic Engineers, New York, 1985.

[ANSI/MIL-STD-1815A, 1983] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, American National Standards Institute, United States Department of Defense, Washington, DC, 1983.

[Barnes and Bollinger, 1991] B. H. BARNES AND T. B. BOLLINGER, "Making Reuse Cost-Effective," *IEEE Software* **8** (January 1991), pp. 13–24.

[Bass, Clements, and Kazman, 2003] L. BASS, P. CLEMENTS, AND R. KAZMAN, *Software Architecture in Practice*, 2nd ed., Addison Wesley, Reading, MA, 2003.

[Baster, Konana, and Scott, 2001] G. BASTER, P. KONANA, AND J. E. SCOTT, "Business Components: A Case Study of Bankers Trust Australia Limited," *Communications of the ACM* **44** (May 2001), pp. 92–98.

[Bosch, 2000] J. BOSCH, *Design and Use of Software Architectures*, Addison Wesley, Reading, MA, 2000.

[Brereton and Budgen, 2000] P. BRERETON AND D. BUDGEN, "Component-Based Systems: A Classification of Issues," *IEEE Computer* **33** (November 2000), pp. 54–62.

[Brown et al., 1998] W. J. BROWN, R. C. MALVEAU, W. H. BROWN, H. W. MCCORMICK, III, AND T. J. MOWBRAY, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley and Sons, New York, 1998.

[Caldiera and Basili, 1991] G. CALDIERA AND V. R. BASILI, "Identifying and Qualifying Reusable Software Components," *IEEE Computer* **24** (February 1991), pp. 61–70.

[Clements and Northrop, 2002] P. CLEMENTS AND L. NORTHROP, *Software Product Lines: Practices and Patterns*, Addison Wesley, Reading, MA, 2002.

[Donohoe, 2000] P. DONOHOE (EDITOR), *Software Product Lines: Experience and Research Directions*, Kluwer Academic Publishers, Boston, 2000.

[D'Souza and Wills, 1999] D. D'SOUZA AND A. WILLS, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley, Reading, MA, 1999.

[Fach, 2001] P. W. FACH, "Design Reuse through Frameworks and Patterns," *IEEE Software* **18** (September/October 2001), pp. 71–76.

[Fayad and Johnson, 1999] M. FAYAD AND R. JOHNSON, *Domain-Specific Application Frameworks: Frameworks Experience by Industry*, John Wiley and Sons, New York, 1999.

[Fayad and Schmidt, 1999] M. FAYAD AND D. C. SCHMIDT, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley and Sons, New York, 1999.

[Fayad, Schmidt, and Johnson, 1999] M. FAYAD, D. C. SCHMIDT, AND R. JOHNSON, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, John Wiley and Sons, New York, 1999.

[Fichman and Kemerer, 1997] R. G. FICHMAN AND C. F. KEMERER, "Object Technology and Reuse: Lessons from Early Adopters," *IEEE Computer* **30** (July 1997), pp. 47–57.

[Fingar, 2000] P. FINGAR, "Component-Based Frameworks for e-Commerce," *Communications of the ACM* **43** (October 2000), pp. 61–66.

[Flanagan, 2002] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference*, 4th ed., O'Reilly and Associates, Sebastopol, CA, 2002.

[Fowler, 1997] M. FOWLER, *Analysis Patterns: Reusable Object Models*, Addison Wesley, Reading, MA, 1997.

[Gamma, Helm, Johnson, and Vlissides, 1995] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, MA, 1995.

[Gifford and Spector, 1987] D. GIFFORD AND A. SPECTOR, "Case Study: IBM's System/360–370 Architecture," *Communications of the ACM* **30** (April 1987), pp. 292–307.

[Green, 2000] P. GREEN, "FW: Here's an Update to the Simulated Kangaroo Story," *The Risks Digest* **20** (January 23, 2000), catless.ncl.ac.uk/Risks/20.76.html.

[Griss, 1993] M. L. GRISS, "Software Reuse: From Library to Factory," *IBM Systems Journal* **32** (No. 4, 1993), pp. 548–66.

[Heineman and Councill, 2001] G. T. HEINEMAN AND W. T. COUNCILL, *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, Reading, MA, 2001.

[Isakowitz and Kauffman, 1996] T. ISAKOWITZ AND R. J. KAUFFMAN, "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering* **22** (June 1996), pp. 407–23.

[ISO/IEC 1539–1, 1997] *Information Technology—Programming Languages—Fortran—Part 1: Base Language*, ISO/IEC 1539–1, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1997.

[ISO/IEC 1989, 2002] *Information Technology—Programming Language COBOL*, ISO 1989:2002, International Organization for Standardization, International Electrotechnical Commission, Geneva, 2002.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries*, ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[ISO/IEC 14882, 1998] *Programming Language C++*, ISO/IEC 14882, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1998.

[Jazayeri, Ran, and van der Linden, 2000] M. JAZAYERI, A. RAN, AND F. VAN DER LINDEN, *Software Architecture for Product Families: Principles and Practice*, Addison Wesley, Reading, MA, 2000.

[Jézéquel and Meyer, 1997] J.-M. JÉZÉQUEL AND B. MEYER, "Put It in the Contract: The Lessons of Ariane," *IEEE Computer* **30** (January 1997), pp. 129–30.

[Johnson, 1979] S. C. JOHNSON, "A Tour through the Portable C Compiler," *UNIX Programmer's Manual*, 7th ed., Bell Laboratories, Murray Hill, NJ, January 1979.

[Johnson and Ritchie, 1978] S. C. JOHNSON AND D. M. RITCHIE, "Portability of C Programs and the UNIX System," *Bell System Technical Journal* **57** (No. 6, Part 2, 1978), pp. 2021–48.

[Jones, 1984] T. C. JONES, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering* **SE-10** (September 1984), pp. 488–94.

[Jones, 1996] C. JONES, *Applied Software Measurement*, McGraw-Hill, New York, 1996.

[Kernighan and Ritchie, 1978] B. W. KERNIGHAN AND D. M. RITCHIE, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Knauber, Muthig, Schmid, and Widen, 2000] P. KNAUBER, D. MUTHIG, K. SCHMID, AND T. WIDEN, "Applying Product Line Concepts in Small and Medium-Sized Companies," *IEEE Software* **17** (September/October 2000), pp. 88–95.

[Kobryn, 2000] C. KOBRYN, "Modeling Components and Frameworks with UML," *Communications of the ACM* **43** (October 2000), pp. 31–38.

[Lai, Weiss, and Parnas, 1999] C. T. R. LAI, D. M. WEISS, AND D. L. PARNAS, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison Wesley, Reading, MA,

1999.

[Lanergan and Grasso, 1984] R. G. LANERGAN AND C. A. GRASSO, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering* **SE-10** (September 1984), pp. 498–501.

[LAPACK++, 2000] "LAPACK++: Linear Algebra Package in C++," at math.nist.gov/lapack++, 2000.

[Lewis et al., 1995] T. LEWIS, L. ROSENSTEIN, W. PREE, A. WEINAND, E. GAMMA, P. CALDER, G. ANDERT, J. VLISSIDES, AND K. SCHMUCKER, *Object-Oriented Application Frameworks,* Manning, Greenwich, CT, 1995.

[Lim, 1994] W. C. LIM, "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software* **11** (September 1994), pp. 23–30.

[Lim, 1998] W. C. LIM, *Managing Software Reuse,* Prentice Hall, Upper Saddle River, NJ, 1998.

[Liskov, Snyder, Atkinson, and Schaffert, 1977] B. LISKOV, A. SNYDER, R. ATKINSON, AND C. SCHAFFERT, "Abstraction Mechanisms in CLU," *Communications of the ACM* **20** (August 1977), pp. 564–76.

[Mackenzie, 1980] C. E. MACKENZIE, *Coded Character Sets: History and Development,* Addison Wesley, Reading, MA, 1980.

[Matsumoto, 1984] Y. MATSUMOTO, "Management of Industrial Software Production," *IEEE Computer* **17** (February 1984), pp. 59–72.

[Matsumoto, 1987] Y. MATSUMOTO, "A Software Factory: An Overall Approach to Software Production," in: *Tutorial: Software Reusability,* P. Freeman (Editor), Computer Society Press, Washington, DC, 1987, pp. 155–78.

[Meyer, 1987] B. MEYER, "Reusability: The Case for Object-Oriented Design," *IEEE Software* **4** (March 1987), pp. 50–64.

[Meyer, 1996a] B. MEYER, "The Reusability Challenge," *IEEE Computer* **29** (February 1996), pp. 76–78.

[Mooney, 1990] J. D. MOONEY, "Strategies for Supporting Application Portability," *IEEE Computer* **23** (November 1990), pp. 59–70.

[Morisio, Ezran, and Tully, 2002] M. MORISIO, M. EZRAN, AND C. TULLY, "Success and Failure Factors in Software Reuse," *IEEE Transactions on Software Engineering* **28** (April 2002), pp. 340–57.

[Morisio, Tully, and Ezran, 2000] M. MORISIO, C. TULLY, AND M. EZRAN, "Diversity in Reuse Processes," *IEEE Software* **17** (July/August 2000), pp. 56–63.

[Musser and Saini, 1996] D. R. MUSSER AND A. SAINI, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library,* Addison Wesley, Reading, MA, 1996.

[NAG, 2003] "NAG The Numerical Algorithms Group Ltd," at www.nag.co.uk, 2003.

[NIST 151, 1988] *POSIX: Portable Operating System Interface for Computer Environments,* Federal Information Processing Standard 151, National Institute of Standards and Technology, Washington, DC, 1988.

[Norusis, 2000] M. J. NORUSIS, *SPSS 10.0 Guide to Data Analysis,* Prentice Hall, Upper Saddle Valley River, NJ, 2000.

[Poulin, 1997] J. S. POULIN, *Measuring Software Reuse: Principles, Practice, and Economic Models,* Addison Wesley, Reading, MA, 1997.

[Prechelt, Unger-Lamprecht, Philippsen, and Tichy, 2002] L. PRECHELT, B. UNGER-LAMPRECHT, M. PHILIPPSEN, AND W. F. TICHY, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance, *IEEE Transactions on Software Engineering* **28** (June 2002), pp. 595–606.

[Prieto-Díaz, 1991] R. PRIETO-DÍAZ, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM* **34** (May 1991), pp. 88–97.

[Schach, 1992] S. R. SCHACH, *Software Reuse: Past, Present, and Future,* videotape, 150 min, US-VHS format, IEEE Computer Society Press, Los Alamitos, CA, November 1992.

[Schach, 1994] S. R. SCHACH, "The Economic Impact of Software Reuse on Maintenance," *Journal of Software Maintenance—Research and Practice* **6** (July/August 1994), pp. 185–96.

[Schach, 1997] S. R. SCHACH, *Software Engineering with Java,* Richard D. Irwin, Chicago, 1997.

[Schricker, 2000] D. SCHRICKER, "Cobol for the Next Millennium," *IEEE Software* **17** (March/April 2000), pp. 48–52.

[Selby, 1989] R. W. SELBY, "Quantitative Studies of Software Reuse," in: *Software Reusability,* Vol. 2, *Applications and Experience,* T. J. Biggerstaff and A. J. Perlis (Editors), ACM Press, New York, 1989, pp. 213–33.

[Shaw and Garlan, 1996] M. SHAW AND D. GARLAN, *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, Upper Saddle Valley River, NJ, 1996.

[Sparling, 2000] M. SPARLING, "Lessons Learned through Six Years of Component-Based Development," *Communications of the ACM* **43** (October 2000), pp. 47–53.

[Tanenbaum, 2002] A. S. TANENBAUM, *Computer Networks,* 4th ed., Prentice Hall, Upper Saddle River, NJ, 2002.

[Toft, Coleman, and Ohta, 2000] P. TOFT, D. COLEMAN, AND J. OHTA, "A Cooperative Model for Cross-Divisional Product Development for a Software Product Line," in: *Software Product Lines: Experience and Research Directions,* P. Donohoe (Editor), Kluwer Academic Publishers, Boston, 2000, pp. 111–32.

[Tracz, 1994] W. TRACZ, "Software Reuse Myths Revisited," *Proceedings of the 16th International Conference on Software Engineering,* Sorrento, Italy, May 1994, pp. 271–72.

[Vlissides, 1998] J. VLISSIDES, *Pattern Hatching: Design Patterns Applied,* Addison Wesley, Reading, MA, 1998.

[Wegner, 1992] P. WEGNER, "Dimensions of Object-Oriented Modeling," *IEEE Computer* **25** (October 1992), pp. 12–20.

[Weyuker, 1998] E. J. WEYUKER, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software* **15** (September/October 1998), pp. 54–59.

# 第 9 章　计划和估算

**学习目标**[⊖]

学完本章之后，你应当能够：

- 说明计划的重要性；
- 估算构建一个软件产品的规模和成本；
- 领会更新和跟踪估算的重要性；
- 提出符合 IEEE 标准的项目管理计划。

建造一个软件产品并没有什么快捷之道。完整建造一个大型软件产品需要时间和资源。而且像其他大型建造项目一样，在项目开始阶段仔细地规划可能是决定成败的最重要的因素。然而这初始的计划无论如何是不够的。计划与测试一样，必须在软件开发和维护过程中不断地进行。尽管需要不断地计划，但在拟制规格说明之后，设计活动开始之前，这些计划活动达到了一个顶点。此时，需要计算有意义的周期和成本估算，并生成完成该项目的具体计划。

本章中，我们区别这两种类型的**计划**，一个是贯穿项目始终的计划，另一个是完成规格说明之后必须产生的详细计划。

## 9.1　计划和软件过程

理想情况下，我们希望在项目最开始时计划整个软件，然后按照计划实行，直到目标软件最终交付给客户。然而这是不可能的，因为在最初的工作流我们缺乏足够的信息来对整个项目提出一个有针对性的计划。例如，在需求工作流，任何类型的计划（除了针对需求工作流本身的计划）都是徒劳的。

开发者在需求工作流末期和分析工作流末期处理的信息相当不同，类似于草图与反映细节的蓝图之间的区别。在需求工作流末期，开发者最多能对客户需要什么有一个非正式的理解。相反，在分析工作流末期，客户签署了一个文档，明确说明了需要建造什么，开发者对目标产品的大部分特征（但通常还不是全部）有了具体的了解。这是建造软件过程中最早的一个关键点，可以确定准确的周期和成本估算。

尽管如此，在一些情况下，客户可能会要求公司在提出规格说明之前进行周期和成本估算。在最坏的情况下，客户可能会坚持在进行了一个或两个小时的预备性讨论后就进行投标。图 9-1 显示了这为何会问题多多。根据［Boehm et al.，2000］中的一个模型，它说明了生命周期的各个工作流成本估算的相对范围。例如，假设产品在实现工作流末期通过了验收测试，并交付给客户，此时发现成本为 100 万美元。如果在需求工作流中期进行了成本估算，很可能估算的成本将在 25 万美元到 400 万美元之间。类似地，如果在分析工作流的中期进行成本估算，则相应的估算范围将缩小为 50～200 万美元。进一步地，如果在分析工作流的末期，也就是合适的时候，进行成本估算，则结果可能仍将是 67～150 万美元的相对范围。换句话说，成本估

⊖ 如前言中所解释的，本章可与第二部分并行教学。（11.15 节和 12.16 节，习题 11.20 和 11.21）的 Osbert Oglesby 实例研究和（习题 11.16）的 Ophelia's Oasis 学期项目的软件项目管理计划要求使用第 9 章的材料。

算不是一项精确的科学，原因将在下一节中讨论。这个模型所依据的数据已经过时，包括提交给美国空军电子化系统部的五个提案［Devenny，1976］和那时经过验证的估算技术。尽管如此，图 9-1 中曲线的整个形状不会有太大的改变。因而，草率的周期或成本估算，也就是客户签署规格说明之前进行的估算，很有可能比收集了足够数据时进行的估算要缺乏准确性。

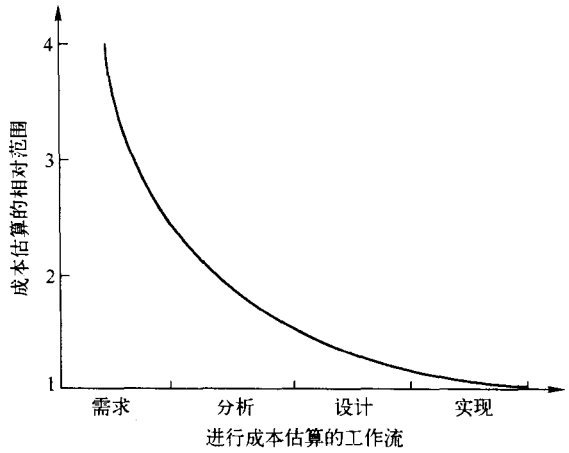



图 9-1　每个生命周期工作流的成本估算的相对范围的模型估算

我们现在考察估算周期和成本的技术。贯穿本章剩余部分的假设是已经完成了分析工作流，也就是说，现在可以产生出有意义的估算和计划了。

## 9.2　周期和成本估算

预算是任何软件项目管理计划中的主要部分。在进行开发之前，客户想要知道他或她将为该产品花费多少钱。如果开发小组对实际的成本估计不足，则开发公司将在该项目上有所损失。另一方面，如果开发小组对实际的成本估算过高，则客户可能会从成本－收益分析或投资回报的角度出发，决定是否有必要建造该产品。客户也可能将该工作转给另一家预算相对合理的公司来做。无论怎样，进行准确的成本估算显然是重要的。

事实上，与软件开发有关的成本有两种类型。第一个是**内部成本**，针对开发者的**成本**；第二个是**外部成本**，客户付的**价格**。内部成本包括开发小组、管理者及与项目相关的支持人员的工资；开发产品所需硬件和软件的成本；以及管理成本，例如租金、公共事业和高层管理者的工资。尽管价格通常是在内部成本的基础上加上利润率，但在一些情况下，经济和心理上的因素是重要的。例如，迫切需要该工作的开发者向客户的要价可能会等于或低于内部成本。而当根据投标的结果签订合同时，则会发生不同的情况。客户可能会放弃价格明显低于其他竞标者而产品质量也明显低于其他竞标者的投标者。开发小组因此会试图让自己的投标只是略微地（而不是明显地）低于那些他们认为有威胁的竞标。

计划的另一个重要部分是估算项目的周期。客户当然想知道何时能交付完成的产品。如果开发公司不能按时间表行事，那么该公司轻则失去信誉，重则被处以罚款。在所有情况下，负责软件项目管理计划的管理者都需要做许多解释工作。相反，如果开发公司将建造产品所需的时间估算得过长，则很有可能失去客户。

遗憾的是，准确地估算出成本和周期决不是件容易的事。要准确估算成本和周期，需要

考虑太多的变量。一个很大的困难是人的因素。35 年前，Sackman 和他的合作者观察到程序员对之间的差距最多可达到 28 比 1［Sackman，Erikson，and Grant，1968］。如果说有经验的程序员总是要胜过新手，因而无视他们的观察结果这很容易，但 Sackman 和他的同事对比了情况匹配的程序员对。例如，他们观察了两个具有 10 年类似项目经验的程序员，计算他们花费在像编码和调试这样的任务上的时间。然后，他们再观察两个入行时间相同并具有类似的教育背景的新手。对比最差的和最好的表现，他们发现，在产品规模上的差距为 6 比 1，在产品执行时间上的差距为 8 比 1，在开发时间上的差距为 9 比 1，在编码的时间上差距为 18 比 1，而在调试时间上的差距为 28 比 1。更给人警示的观察结果是对一个产品最好和最坏的表现是由两个都具有 11 年经验的程序员做出的。甚至从 Sackman 等人的样本中去掉最好和最坏的两种情况时，观察到的差距仍在 5 比 1 左右。根据这些结果，很明显，我们不能期望估算成本和周期会准确到何种程度（除非我们具有关于所有雇员的所有特长的详细信息，而这是不太可能的）。对于大型项目，现在还存在着争论，有人认为个体之间的差距可以忽略，但这也许只是美好的想法，一个或两个非常好（或非常差）的小组成员的存在会引起计划时间表的显著偏差，并严重影响预算。

另一个影响估算的人的因素是，在一个自由的国家里，无法确保关键的小组成员在项目期间不跳槽。之后需要花费时间和金钱来填补这个真空位置，并使替代者融合到小组中，或者重新组织剩余的小组成员来补偿损失。无论采用何种方法，时间计划表都会有偏差，估算都会失去控制。

成本估算问题的背后隐含着另一个问题：如何计算产品的规模？

### 9.2.1 产品规模的度量

最常用的产品规模度量是代码行数，通常使用两个单位：**代码行**（lines of code，LOC）和**千行交付源代码指令**（thousand delivered source instructions，KDSI）。许多问题与代码行的使用有关［van der Poel and Schach，1983］。

- 源代码的创建只是整个软件开发工作量的小部分，把需求、分析、设计、实现和测试工作流（包含计划和形成文档）所需的时间只表示为最终产品的代码行的函数看起来有点牵强。
- 用两种不同的语言实现相同的产品会导致生成具有不同代码行数的版本。还有使用诸如 Lisp 或许多非过程的 4GL（14.2 节）时，并没有定义代码行的概念。
- 通常没有非常明确的说法来说明如何计算代码行。是只计算可执行代码行，还是要考虑到数据的定义？而且是否应该计算注释的内容？如果不应计算注释的内容，则程序员将不愿意花费时间在他们认为是“非生产性”的注释上，但如果计入注释内容，那么程序员反过来会写出大量的注释，以试图拔高他们表面的生产力。还有，如何计算任务控制语言的语句？另一个问题是如何计算修改过的或删除了的行——在改进产品以提高性能的过程中，有时会减少代码行数。代码重用（8.1 节）也使行的计算复杂化。如果重用的代码需要修改，应如何计算？而如果代码是从父类中继承下来的（7.8 节），又将如何计算？一句话，代码行的度量是相当直观的，但如何计算它却并不直观。
- 不是所有编写出来的代码都要交付给客户，通常有一半的代码包含有支持开发工作所需的工具。
- 假设软件开发者使用诸如报表生成器、屏幕生成器或图形用户接口（GUI）生成器这样

的代码生成器，在该开发者进行了一些设计工作后，这些工具可能会产生数千行代码。

- 最终产品中的代码行数只能在产品全部完成之后才能确定出来，所以，根据代码行的成本估算具有双重危险。为开始估算过程，必须先估算出完成的产品中的代码行，然后，应用这个估算结果来估算产品的成本。不仅在每种计算成本的技术中存在有不确定性，而且如果不确定的成本估算器的输入（即还没建造出的产品的代码行数）就是不确定的，那么得出的成本估算结果的可信度不太可能很高。

因为代码行数如此不可靠，就必须考虑其他度量。另一个估算产品规模的方法是使用基于可测量量的度量，这些量可在软件开发过程的初期确定。例如，van der Poel 和 Schach [1983] 为中等规模的数据处理产品的成本估算提出了**FFP 度量**。数据处理产品的三个基本结构组成是文件（file）、信息流（flow）和过程（process）。名称 FFP 是这些要素的首字母组成的缩写。文件定义为永久驻留在产品中的、逻辑或物理上相关的记录的集合，不包括事务处理文件和临时文件。信息流是产品和环境（例如屏幕和报表）之间的数据接口。过程是功能上定义的对数据的逻辑或算术操作，例如排序、验证或更新。假设一个产品中的文件数为 $Fi$，信息流数为 $Fl$，过程数为 $Pr$，则产品的规模 $S$ 和成本 $C$ 为：

$$S = Fi + Fl + Pr \tag{9.1}$$

$$C = d \times S \tag{9.2}$$

其中 $d$ 是一个常数，各个公司之间的这个常数各不相同。常数 $d$ 是对公司内部软件开发过程的**效率（生产力）**的测量。产品的规模只是文件数、信息流数和处理数的简单求和，它是一旦结构设计完成之后即可确定的数量。成本则是与规模成比例的，比例常数 $d$ 由与那个公司先前开发的产品有关的成本数据的最小平方值确定。不像基于代码行数的度量，成本可在编写代码开始前估算。

通过使用一个专门选择的样本（该样本覆盖了中等规模的数据处理应用），说明了 FFP 度量的正确性和可靠性。遗憾的是，该度量从来没有扩展到包含数据库，而数据库是许多数据处理产品的基本组成部分。

一个类似的、但独立开发的产品规模的度量由 Albrecht [1979] 基于功能点[⊖]开发出来。Albrecht 的度量基于输入项数 $Inp$、输出项数 $Out$、查询数 $Inq$、主文件数 $Maf$ 和接口数 $Inf$。**功能点**数 $FP$ 可由下面最简形式的等式给出：

$$FP = 4 \times Inp + 5 \times Out + 4 \times Inq + 10 \times Maf + 7 \times Inf \tag{9.3}$$

因为这是对产品规模的度量，所以它可用于成本估算和生产力估算。

等式（9.3）是过于简化的三步计算过程。首先，计算了未经调整的功能点：

1）产品的这些组件（$Inp$、$Out$、$Inq$、$Maf$ 和 $Inf$）中的每一个必须归类为简单的、一般的或复杂的（参见表 9-1）。

2）根据每个组件的级别给每个组件分配一个功能点数。例如，如等式（9.3）所反映的，给一般的输入分配了 4 个功能点，但只给简单的输入分配了 3 个功能点，而给复杂的输入分配了 6 个功能点。这一步所需的数据如表 9-1 所示。

3）然后对分配给每个组件的功能点求和，则产生了**未经调整的功能点**（unadjusted func-

⊖ 原文为 function point，译为功能分数更准确些，因为它实际上是对功能的一种打分评估，但国内软件工程方面的书籍大多译为功能点，为保持一致起见，也译为功能点。——译者注

tion point，UFP)。

表 9-1　功能点取值表

| 系　统 | 复杂度级别 | | |
|---|---|---|---|
| | 简单 | 一般 | 复杂 |
| 输入项 | 3 | 4 | 6 |
| 输出项 | 4 | 5 | 7 |
| 查询 | 3 | 4 | 6 |
| 主文件 | 7 | 10 | 15 |
| 接口 | 5 | 7 | 10 |

其次，计算**技术复杂因子**（technical complexity factor，TCF)。这是对 14 个技术因子的影响进行的测量，诸如高事务处理率、性能准则（例如吞吐量或反应时间）和在线更新等，图 9-2 给出了这些因子的完整的集合。这 14 个因子中的每一个都分配了一个值，从 0（“不存在或没有影响”）到 5（“从始到终的影响都很强烈”)。然后把所产生的 14 个数相加，得到总的影响度（degree of influence，DI)，然后由下式给出 TCF：

$$TCF = 0.65 + 0.01 \times DI \tag{9.4}$$

因为 DI 可从 0 变化到 70，所以 TCF 可以从 0.65 变化到 1.35。

第三，功能点数 FP 可由下式给出：

$$FP = UFP \times TCF \tag{9.5}$$




图 9-2　功能点计算的技术因素

测量软件生产力的实践已经显示出使用功能点比使用 KDSI 更合适。例如，Jones [1987] 指出，他观察到计算 KDSI 有超过 800% 的错误，而计算功能点“只有” 200% 的错误，这是最能说明问题的解释。

表 9-2　汇编器和 Ada 产品的对比 [Jones，1987]（©1987 IEEE）

| | 汇编器版本 | Ada 版本 |
|---|---|---|
| 源代码规模 | 70 KDSI | 25 KDSI |
| 开发成本 | 1 043 000 美元 | 590 000 美元 |
| 每人月约 KDSI | 0.335 | 0.211 |

（续）

| | 汇编器版本 | Ada 版本 |
|---|---|---|
| 每个源语句成本 | 14.90 美元 | 23.60 美元 |
| 每人月的功能点 | 1.65 | 2.92 |
| 每个功能点的成本 | 3023 美元 | 1170 美元 |

为展示功能点比代码行优越，Jones［1987］引用了表 9-2 所示的例子。分别以汇编语言和 Ada 对相同的产品进行编程，并将结果进行对比。首先看每人月的 KDSI，这个度量告诉我们用汇编语言编程比用 Ada 编程的效率高出 60%，而这是一个明显错误的结论。像 Ada 这样的第三代语言取代汇编语言，就是因为第三代语言的代码效率更高。现在再看第二个度量，每个源语句成本。注意在这个产品中，一个 Ada 语句等于 2.8 个汇编语句。使用每个源语句成本作为效率的度量再次表明了汇编语言比 Ada 的效率更高，然而，当把每人月的功能点作为编程效率的度量时，Ada 比汇编的优越性则明显地反映出来。

另一方面，等式（9.1）和（9.2）的功能点和 FFP 度量也具有相同的缺点：产品维护通常很难准确测度。当对产品进行维护时，可以在不改变文件、信息流和过程数，或者不改变输入、输出、查询、主文件和接口数的情况下对产品进行主要修改。而这样的情况下代码行也不适用，举个极端的情况为例，可以把产品的每一行都替换成完全不同的代码，却不会改变整个代码行数。

至少有 40 个 Albrecht 功能点的变种和扩展已被提出［Maxwell and Forselius，2000］。Mk II 功能点由 Symons［1991］提出，它提供了一个计算未经调整的功能点（UFP）的更精确的方式。软件可分解成一系列组件事务，每一个组件事务包含一个输入、一个过程和一个输出，然后根据这些输入、过程和输出计算出 UFP 的值。Mk II 功能点在全世界被广泛应用［Boehm，1997］。

### 9.2.2　成本估算技术

尽管估算规模有难度，但重要的是软件开发者需要尽量准确地估算项目周期和项目成本，还要尽可能多地考虑到那些会影响估算的因素，包括个人的业务熟练程度、项目的复杂度、项目的规模（成本随规模的增加而增加，但却远大于线性地增加）、开发小组对应用领域的熟悉程度、运行产品的硬件以及 CASE 工具的可用度。另一个因素是所谓的最后期限的影响。如果项目需要在某一个时间前完成，则以人月计算的工作量会比没有完成时间限制时的工作量大很多。这表明周期和成本不是各自独立的，最后期限越短，工作量就越大，因而成本就越高。

从前面列出的还不全面的清单来看，显然估算是一个困难的问题，可以使用如下一些多多少少有些成功的方法。

**1. 用类推法进行专家判决**

在**用类推法进行专家判决**这项技术中，需要咨询一些专家。一个专家通过将目标产品与他积极参与完成的产品进行对比，并指出相似点和不同处，从而得出一个估算。例如，一个专家可以把目标产品与两年前开发的需要成批地输入数据的产品进行对比（尽管目标产品需要具有在线数据捕获能力）。因为公司对要开发的产品的类型很熟悉，专家可以将开发时间

和工作量减少15%。然而，图形用户界面（GUI）有点复杂，这增加了25%的时间和工作量。最后，目标产品需要以大多数小组成员不熟悉的语言进行开发，这样增加了15%的时间和20%的工作量。综合这三个数字，专家确定目标产品将比前面的产品多花费25%的时间和30%的工作量。因为完成前面的产品花费了12个月，并要求100人月，则目标产品将花费15个月，耗费130人月。

公司内的另外两个专家对比了相同的两个产品。一个专家断定目标产品将花费13.5个月以及140人月，而另一个专家的结论是16个月和95人月。如何协调这三个专家的预测？一项可用的技术是**Delphi技术**：它允许专家们不经过集体会议就达成一致意见，而集体会议可能有整个团体受某个善于游说的成员左右的负面影响。在这项技术中，专家们独立地进行工作。每个专家产生一个估算结果，并为该估算提供解释。然后这些估算和解释分发给所有的专家，而这些专家现在要进行第二次估算。这个估算和分发过程继续保持，直到专家们在一个可接受的公差内达成一致。在这个重复的过程中没有召开集体会议。

房产的评估常常通过类推法，在专家决策的基础上进行。评估人通过对比类似的最近售出的房子来得到评估值。假设要评估房子A，隔壁的房子B刚刚以205 000美元售出，而下一条街的房子C三个月前以218 000美元的价格售出。评估人可以进行如下推论：房子A比房子B多一个浴室，而院子比房子B大5000平方英尺，房子C的面积与房子A相同，但它的屋顶状况不好。另一方面，房子C有一个按摩浴缸。经过仔细的思考，评估人为房子A估价215 000美元。

在软件产品的情况下，用类推法进行专家决策没有房产估价那么精确。回想我们的第一个软件专家声称，使用不熟悉的语言将增加15%的时间和20%的工作量。除非该专家具有一些经过验证的数据，从中可以确定每个不同之处的影响（这是非常不可能的），由这种只能说是猜想而引发的错误肯定会导致出错误的成本估算。另外，除非专家们能保留整个回忆（或记住具体的记录），否则他们回忆的已完成的产品必然将是不准确的，以至于使他们的预测变得无效。最后，专家也是人，因此也会出现偏差，会影响他们的预测。同时，由一组专家所估算的结果应反映他们集体的经验，如果这些经验足够广泛，则结果将会很准确。

**2. 自底向上的方法**

要减少估算整个产品所引起的错误，方法之一是把产品分割成更小的组件。对每个组件单独地进行周期和成本的估算，然后结合起来形成一个整体的数字。这种**自底向上的方法**的好处在于，为多个更小的组件估算成本通常比为大的组件估算成本更快，也更准确。另外，估算过程也比大型的单个产品更具体。这种方法的缺点是产品不仅仅是各个组件的和。

对于面向对象的范型，各种类的独立性有助于采用这种自底向上的方法。然而，产品中各种对象之间的交互也使估算过程复杂化。

**3. 算法成本估算模型**

在这个方法中，诸如功能点或FFP这样的度量可作为确定产品成本的模型的输入。估算人计算度量的值，然后使用该模型计算周期和成本。从表面看，**算法的成本估算**模型优于专家们的观点，因为专家作为一个人，如前面所指出的，总会有偏差，可能会忽视已完成产品和目标产品的某个方面。相反，一个算法的成本估算模型是不会有偏差的，每个产品都是按相同的方式对待的。使用这种模型的问题在于它的估算只在隐含假设下才是好的。例如，隐含的功能点模型的假设是，产品的每个方面都具体化为等式（9.3）右边的5个量和14个

技术因素。进一步的问题是，通常在决定给模型的参数赋什么值时需要相当的主观判断。例如，估算人经常会不清楚功能点模型的某个技术因子是应该打分为3还是4。

目前已经提出许多算法的成本估算模型，一些是基于像软件如何开发这样的数学理论，另一些模型是基于统计值的，通过研究大量的项目，从数据中确定经验法则。混合模型结合了数学等式、统计模型和专家决策。最重要的混合模型是Boehm的COCOMO，将在下一节中详细描述。(要了解首字母缩写词COCOMO，参见下面的“如果你想知道”部分。)

**如果你想知道**

COCOMO是从COnstructive COst MOdel（构造性成本模型）中抽取每个单词的前两个字母而形成的首字母缩写词，它与印第安那州的科科莫（KoKomo，Indiana）的关联只是巧合而已。

COCOMO中的MO代表“模型”，因此不应该使用词组“COCOMO模型”，这与“ATM机器”和“PIN号码”的情况相同，都是冗余的用法。

### 9.2.3 中间COCOMO

**COCOMO**实际上是三个模型的系列，从把产品作为一个整体看待的宏观估算模型到具体化看待产品的微观估算模型。本节将给出对中间COCOMO的描述，它具有中等的复杂度和细节。Boehm［1981］中详细描述了COCOMO，Boehm［1984］中给出了一个概述。

使用中间COCOMO计算开发时间分两个阶段。首先，提出一个开发工作量的大致估算，这需要估算两个参数：产品以KDSI计算的长度和产品的开发模式——对开发该产品固有的困难程度的度量。有三种模式：有组织的（小而简单的）、半分离的（中等规模的）和嵌入式的（复杂的）。

从这两个参数可以计算**额定工作量**（nominal effort）。例如，如果判断一个项目基本上是简单的（有组织的），那么额定工作量（以人月为单位）由下面的等式得出：

$$\text{额定工作量} = 3.2 \times (\text{KDSI})^{1.05} \text{人月} \tag{9.6}$$

常数3.2和1.05是最合适的值，与Boehm为开发中间COCOMO而使用的组织模式产品中的数据最匹配。

例如，如果要建造的产品是组织模式的，经估算有12 000行交付的源语句（即12KDSI），那么额定工作量是

$$3.2 \times (12)^{1.05} = 43 \text{ 人月}$$

(但是请阅读下面的“如果你想知道”部分来了解这个值的含义)。

**如果你想知道**

你对额定工作量值的一个反应可能是，“如果生成12 000行的源代码指令需要43人月的工作量，那么平均每个程序员每个月可生成不到300行代码——而我在一个晚上即可写出不止300行代码。”

通常一个300行的产品只是：300行代码。相反地，一个可维护的12 000行产品需要经过生命周期的所有阶段。换句话说，43人月的总工作量包含有许多行为，包括编码（平均来说编码只占总工作量的15%～20%）。

其次，这个额定值必须乘以15个**软件开发工作量因子**。这些因子和它们的值在表9-3中

给出。每个因子最多有6个值，例如，根据开发者评定产品复杂度为非常低、低、额定（一般）、高、非常高或特别高，产品复杂度因子分别具有0.70、0.85、1.00、1.15、1.30或1.65这些值。从表9-3中可看出，所有的15个因子在对应的参数为额定时都取值为1.00。

**表9-3　中间COCOMO软件开发工作量因子**［Boehm，1984］（©1984 IEEE）

| 成本组件 | 复杂度级别 | | | | | |
|---|---|---|---|---|---|---|
| | 非常低 | 低 | 额定的 | 高 | 非常高 | 特别高 |
| 产品属性 | | | | | | |
| 要求的软件可靠性 | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 | |
| 数据库规模 | | 0.94 | 1.00 | 1.08 | 1.16 | |
| 产品复杂度 | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| 计算机属性 | | | | | | |
| 执行时间限制 | | | 1.00 | 1.11 | 1.30 | 1.66 |
| 主存限制 | | | 1.00 | 1.06 | 1.21 | 1.56 |
| 虚拟机的变更性* | | 0.87 | 1.00 | 1.15 | 1.30 | |
| 计算机周转时间 | | 0.87 | 1.00 | 1.07 | 1.15 | |
| 人员属性 | | | | | | |
| 分析员能力 | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 | |
| 应用经验 | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 | |
| 程序员能力 | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 | |
| 虚拟机经验* | 1.21 | 1.10 | 1.00 | 0.90 | | |
| 编程语言经验 | 1.14 | 1.07 | 1.00 | 0.95 | | |
| 项目属性 | | | | | | |
| 现代编程实践的使用 | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 | |
| 软件工具的使用 | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 | |
| 要求的开发时间表 | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 | |

*对于一个给定的软件产品，使用的虚拟机是硬件和它所调用以完成其任务的软件（操作系统、数据库管理系统）的复合体。

Boehm提出了一些准则，帮助开发者确定是否该参数确实应当定级为额定值，或者该定级是低了还是高了。例如，再看那个模块复杂度因子。如果模块的控制操作主要由一系列结构化编程的构造组成（例如**if-then-else**、**do-while**、**case**），那么这个复杂度可定级为“非常低”；如果这些操作是嵌套的，则复杂度可定级为“低”；加入模块间控制和判决表可使复杂度级别提高为“额定的”；如果这些操作是高度嵌套的，带有复合断言，而且有队列和堆栈，那么复杂度可定级为“高”；可重入和递归编程以及固定优先级中断处理的出现使复杂度达到了“非常高”；最后，用动态改变的优先级调度的多个资源和微代码级控制确保了复杂度定级为“特别高”。这些分级应用于控制操作。一个模块也需要从计算操作、基于设备的操作和数据管理操作的视角进行估计。有关计算这15种因子的标准详见［Boehm，1981］。

为明白这是如何工作的，［Boehm，1984］给出了基于微处理器的通信处理软件的例子，该软件是为一个高可靠性的新的电子资金传送网络而设计的，它带有性能、开发计划和接口

要求。这个产品符合嵌入式模式的描述，估计将有 10 000 行交付的源指令（10KDSI），所以额定开发工作量由下式给出：

$$\text{额定工作量} = 2.8 \times (\text{KDSI})^{1.20} \tag{9.7}$$

（同样，常数 2.8 和 1.20 是最符合嵌入式产品数据的值。）因为该项目估计长度有 10KDSI，则额定工作量是

$$2.8 \times (10)^{1.20} = 44 \text{ 人月}$$

估算的开发工作量通过将额定工作量乘以 15 个软件开发工作量因子得到。这些因子的复杂度级别和它们的值在表 9-4 中给出。使用这些值，得到这些因子的产品是 1.35，所以该项目的估算工作量为

$$1.35 \times 44 = 59 \text{ 人月}$$

然后将这个数用到另外的公式中，以确定美元成本、开发时间表、阶段和活动分布、计算机成本、每年的维护成本和其他的相关事项，要了解详细内容可参见［Boehm，1981］。中间 COCOMO 是一个完全算法的成本估算模型，可在项目计划中给用户提供实际上能想得到的每种帮助。

**表 9-4　微处理器通信软件的中间 COCOMO 工作量因子级别［Boehm，1984］（©1984 IEEE）**

| 成本组件 | 情　形 | 级　别 | 工作量因子 |
|---|---|---|---|
| 要求的软件可靠性 | 软件错误带来的严重经济后果 | 高 | 1.15 |
| 数据库规模 | 20 000 字节 | 低 | 0.94 |
| 产品复杂度 | 通信处理 | 非常高 | 1.30 |
| 执行时间限制 | 将使用 70% 的可用时间 | 高 | 1.11 |
| 主存限制 | 64K 主存中的 45K（70%） | 高 | 1.06 |
| 虚拟机的变更性 | 基于商用微处理器硬件 | 额定的 | 1.00 |
| 计算机周转时间 | 平均两小时周转时间 | 额定的 | 1.00 |
| 分析员能力 | 好的高级分析员 | 高 | 0.86 |
| 应用经验 | 3 年 | 额定的 | 1.00 |
| 程序员能力 | 好的高级程序员 | 高 | 0.86 |
| 虚拟机经验 | 6 个月 | 低 | 1.10 |
| 编程语言经验 | 12 个月 | 额定的 | 1.00 |
| 现代编程实践的使用 | 所用技术大多数超过了 1 年 | 高 | 0.91 |
| 软件工具的使用 | 处于基本的微机工具水平 | 低 | 1.10 |
| 要求的开发时间表 | 9 个月 | 额定的 | 1.00 |

中间 COCOMO 已经得到一个有 63 个项目的广泛抽样调查的验证，这个抽样调查覆盖了各种应用领域。将中间 COCOMO 应用于这个抽样调查的结果是，在大约 68% 的时间里，实际值在预测值的 20% 以内。试图提高这个准确度基本没有什么意义，因为在大多数公司里，中间 COCOMO 的输入数据的准确度通常仅在 20% 以内。尽管如此，在 20 世纪 80 年代，经验丰富的估算员所得到的准确度使得中间 COCOMO 处于成本估算研究的最前沿，没有其他技术能够始终这样准确。

中间 COCOMO 的主要问题是，它最重要的输入是目标产品中的代码行数。如果这个估算是不正确的，那么该模型的每个预测都将不正确。由于中间 COCOMO 的预测或任何其他的估算技术都有可能不准确，因此管理者必须在整个软件开发过程中监控所有的预测。

### 9.2.4 COCOMO II

COCOMO是在1981年提出的，那时使用的生命周期模型只有瀑布模型，大多数软件运行在大型机上，诸如客户－服务器和面向对象这样的技术基本上还是未知的。因而，COCOMO没有包含任何这些因素。然而，随着更新的技术开始成为广泛接受的软件工程实践，COCOMO开始变得不再准确。

**COCOMO II**［Bohem et al.，2000］是1981年的COCOMO的主要修订版。COCOMO II可以处理很宽范围的现代软件工程技术，包括面向对象、第2章中描述的各种生命周期模型、快速原型（10.12节）、第四代语言（14.2节）、重用（8.1节）和COTS软件（1.11节）。COCOMO II既灵活又完善。但遗憾的是，为实现这一目标，COCOMO II也比原来的COCOMO更复杂了，因而，希望使用COCOMO II的读者应该仔细研究［Boehm et al.，2000］，这里只概述了COCOMO II和中间COCOMO之间的主要区别。

首先，中间COCOMO包含一个基于代码行（KDSI）的总的模型，另一方面，COCOMO II包含三个不同的模型。**应用组合模型**基于对象点（类似于功能点），应用于最早的阶段，此时关于要建造的产品可用的信息很少。然后，随着可用的信息变得多起来，可以使用**早期设计模型**，这个模型基于功能点。最后，当开发者拥有了最多的信息，则可以使用**后结构模型**，这个模型使用功能点或代码行（KDSI）。中间COCOMO的输出是成本和规模的估算，COCOMO II的三个模型的输出均是成本和规模估算的范围。这样，如果工作量的最可能的估算是$E$，那么应用组合模型将返回一个范围（$0.50E$，$2.0E$），后结构模型返回范围（$0.80E$，$1.25E$）。这反映了COCOMO II的模型逐渐增加的准确度。

第二个区别存在于隐含在COCOMO中的工作量模型中：

$$\text{工作量} = a \times (\text{规模})^b \tag{9.8}$$

其中$a$和$b$是常数。在中间COCOMO中，指数$b$有三个不同的值，这取决于要建造的产品的模式是有组织的（$b=1.05$）、半分离的（$b=1.12$）还是嵌入式的（$b=1.20$）。在COCOMO II中，根据模型的各种参数，$b$的值在1.01和1.26之间变化。这些参数包含对那种类型产品的熟悉程度、过程成熟度级别（3.13节）、风险解决的程度（2.7节）和小组合作的程度（4.1节）。

第三个区别是与重用有关的假设。中间COCOMO假设重用带来的节省与重用的数量是直接成比例的。COCOMO II考虑到对重用软件进行的小的修改导致不成比例的巨大成本（因为甚至一个小的修改也需要详细理解代码，且测试一个修改过的模块的成本相当巨大）。

第四，目前有17种乘性的成本组件，而不是中间COCOMO中的15种。成本组件中的7个是新的，例如在未来产品中要求的重用能力、年均的人员调整和该产品是否在多个地点进行开发。

COCOMO II已经用从各种不同的领域选择的83个项目进行了检验，但该模型仍然太新，还不能有许多关于它的准确性的结果，特别是它比它的前辈（1981年的COCOMO）改善的程度。

### 9.2.5 跟踪周期和成本估算

当开发产品时，实际的开发工作量一定要与预测值进行比较。例如，假设软件开发者使用的预算度量预示了分析工作流的**周期**将持续3个月，并且要求有7人月的工作量。然而，实际持续了4个月，并扩充到10人月的工作量，仍旧不能完成规格说明的文档。这类偏差可作为一个有问题的早期警告，必须采取正确的行动。问题可能是大大低估了产品的规模，

或者开发小组并不像想像的那样胜任。无论是什么原因，都将会使周期和成本大大超出，而管理者必须采取合适的行动来使这种影响最小化。

必须在整个开发过程中对预测进行仔细地跟踪，不论预测应用了何种技术。偏差应归咎于低水平的预测者的度量、低效率的软件开发、上述两者的结合或者其他原因。重要的是尽早检测出偏差，并立即采取正确的行动。另外，根据已变成可用的额外信息持续更新预测很重要。

上面讨论了估算周期和成本的度量，下面讨论软件项目管理计划的组成。

## 9.3　软件项目管理计划的组成

一个软件项目管理计划有三个主要的组成部分：要做的工作、做这个工作所用的资源以及为此需要付出的金钱。在本节中将讨论这三个组成部分，术语取自［IEEE 1058，1998］，9.4 节中会深入地讨论这些术语。

软件开发需要资源，所需的主要**资源**是开发该软件的人、软件在其上运行的硬件和诸如操作系统、文本编辑器和版本控制软件（5.7 节）这样的支持软件。

使用人员这样的资源随时间而变化。Norden［1958］表示对于大型的项目，**瑞利分布**是资源消耗 $R_c$ 随时间 $t$ 变化的最好近似，也就是

$$R_c = \frac{t}{k^2} e^{-t^2/2k^2} \qquad 0 \leqslant t < \infty \tag{9.9}$$

参数 $k$ 是一个常数，表示消耗最大的时刻，而 e=2.71828...，是自然对数的底。一个典型的瑞利曲线如图 9-3 所示，资源消耗开始很小，很快爬到高峰点，然后以较慢的速度下降。Putnam［1978］调查了 Norden 结果对软件开发的适用性，并发现人员和其他资源的消耗由瑞利分布建模，具有一定程度的准确性。

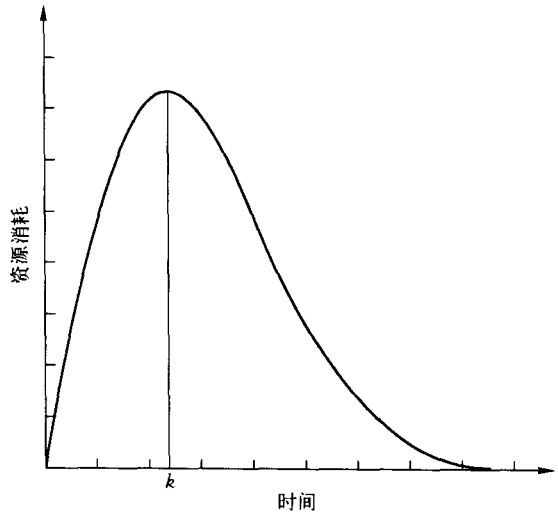



图 9-3　表明资源消耗如何随时间变化的瑞利曲线

因此在软件计划中认为只需要 3 个具有至少 5 年经验的高级程序员是不够的，还需要进行下述的一些事情：

在实时编程时需要 3 个具有至少 5 年经验的高级程序员，其中有 2 个程序员需要在项目开始后 3 个月开始工作，而 6 个月之后需要第三个程序员。产品测试开始后有 2 个程序员逐

渐退出，当维护开始时，第三个程序员退出编程。

资源需求依赖于时间的事实不仅适用于人员，还适用于计算机时间、支持软件、计算机硬件、办公室设备，甚至包括旅程。这样软件项目管理计划将是时间的一个函数。

需要做的工作分为两类。首先是在整个项目中持续进行与软件开发的任何特定阶段都不相关的工作。这样的工作被称为**项目函数**。例子是项目管理和质量控制。第二是与产品开发中的某个特定阶段相关的工作，这样的工作被称为“行为”或“任务”。**行为**是有明确的开始和结束日期的工作的主要单元，它消耗“资源”，例如计算机时间或人天，并形成**工作产品**，例如预算、设计文档、时间表、源代码或用户手册。反过来，行为包含一系列“任务”，一个**任务**是受管理责任控制的工作的最小单元。在软件项目管理计划中有三类工作：整个项目中实现的项目功能、行为（较大的工作单元）和任务（较小的工作单元）。

这个计划的一个重要方面涉及工作的产品的完成。确认工作产品是完成了的日期被称为**里程碑**（milestone）。为确定一个工作产品是否真正到达了一个里程碑，首先必须通过一系列的**评审**，这些评审由小组成员的同事、管理者或客户进行。一个典型的里程碑是设计完成并通过评审的日期。一旦一个工作产品经过了评审，并得到通过，它就成为一个**基准**（baseline），只能通过正式的程序步骤才能修改它，如 5.8.2 节所描述。

实际中，对于工作产品来说，不仅仅是产品本身。一个**工作包**（work package）不仅定义一个工作产品，还包括人员配备要求、周期、资源、责任人的姓名以及工作产品的验收标准。当然**钱**是计划的一个主要部分，必须具有一个详细的预算，并作为一个时间的函数，将这些钱分配给项目的功能和行为。

下面将讨论如何拟制软件产品计划的问题。

## 9.4　软件项目管理计划框架

提出项目管理计划有许多方法，最好的一个是 IEEE 标准 1058［1998］，该计划的组成如图 9-4 所示。

- 该标准是由许多从事软件开发的主要公司的代表提出的。输入信息来自工业界和大学，工作组和评审组的成员在提出项目管理计划方面都具有多年的实践经验。该标准结合了这些实践。
- IEEE 项目管理计划是为所有类型的软件产品而设计的，它不强加特定的生命周期模型或描述特定的方法学。该计划主要是一种体制，内容可由每个公司根据特定的应用领域、开发小组或技术进行剪裁。
- IEEE 项目管理计划体制支持过程提升，例如，该框架的许多章节反映了诸如配置管理和度量这样的 CMM 关键过程领域（3.13 节）。
- IEEE 项目管理计划体制适合于统一过程。例如，该计划的一节是关于需求控制的，而另一节是关于风险管理的，两者都是统一过程的主要方面。

另一方面，尽管 IEEE 标准 1058［1998］声明 IEEE 项目管理计划对所有规模的软件项目都适用，但其中的一些章节与小型软件无关。例如，计划体制的 7.7 节标题为“转包商管理计划”，但在小型软件中还没有听说过有转包商。

因此，我们现在分两种方式介绍该计划。首先，在 9.5 节描述全部的体制；其次，在附录 F 中介绍小型项目——Osbert Oglesby 用例研究的管理计划，它采用了该体制的简化版。

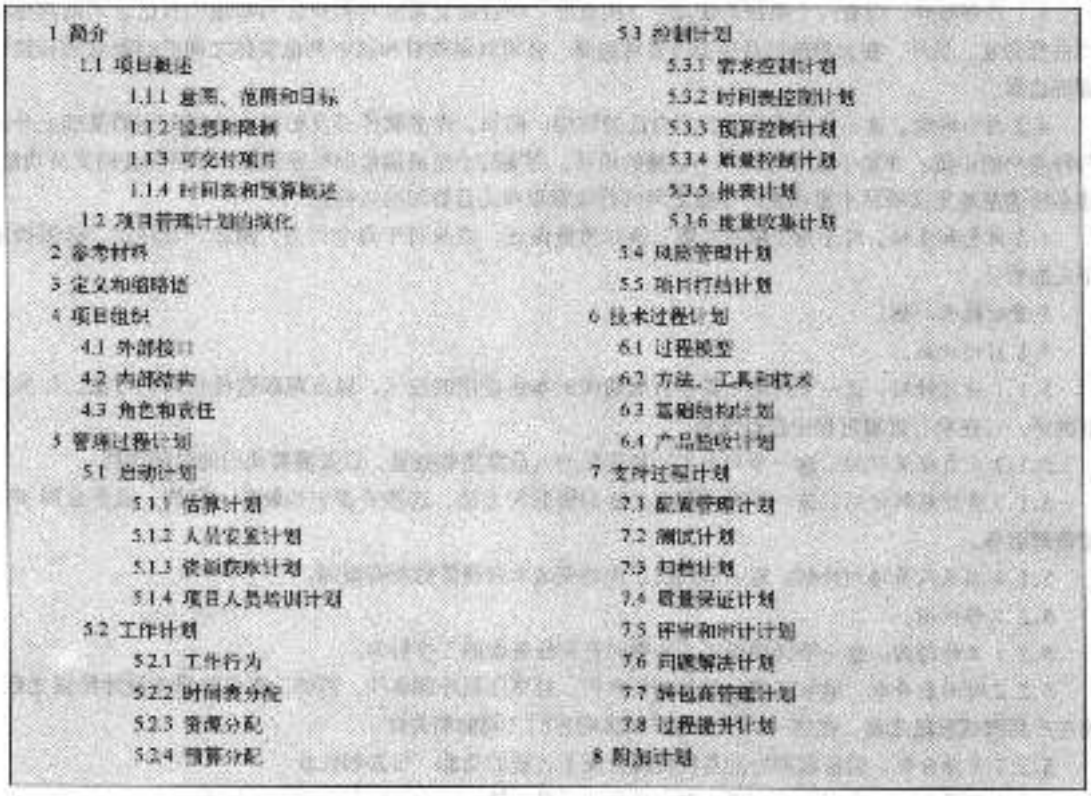



图 9-4 IEEE 项目管理计划框架

## 9.5 IEEE 软件项目管理计划

现在具体描述 IEEE 软件项目管理计划（SPMP）体制本身，文章中的编号和标题对应于图 9-4 中的条目，9.3 节中已经定义了所使用的各种术语。

1 简介。

1.1 项目概述。

1.1.1 意图、范围和目标。简要描述了要交付的软件产品的意图和范围，以及项目目标，商业需要也包含在这一小节中。

1.1.2 设想和限制。任何隐含在项目中的设想和限制都在这里说明，例如交付时间、预算、资源和要重用的产品。

1.1.3 可交付项目。该节列出了需要交付给客户的所有事项，包括交付时间。

1.1.4 时间表和预算概述。该节提供了整个时间表，包括整个预算。

1.2 项目管理计划的演化。计划不可能一下子塑造成形。项目管理计划与其他计划一样，要求在经验的启发和客户公司及软件开发公司共同修改的基础下进行不断地更新。这一节中描述了修改计划的正规过程和机制。包括把项目管理计划本身纳入配置控制的机制。

2 参考材料。这里列出了项目管理计划中的所有参考文档。

3 定义和缩略语。这个信息确保每个人都能以相同的方式理解项目管理计划。

4 项目组织。

4.1 外部接口。没有一个项目是在真空下构造的。项目成员需要与客户公司和他们自己公司的其他成员进行交互。另外，在大型的项目中会涉及转包商，必须制定项目和这些其他实体之间的行政管理和经营管理边界。

4.2 内部结构。这一节讨论开发公司自己的结构，例如，许多软件开发公司在公司范围的基础上分成两种类型的小组：开发小组致力于一个单独的项目，而支持小组提供诸如配置管理和质量保证的支持功能。还必须清楚地定义项目小组和支持小组之间的行政管理和经营管理的边界。

4.3 角色和责任。对于每个项目功能，例如质量保证，以及对于每个行为，例如产品测试，必须确定个人的责任。

5 管理过程计划。

5.1 启动计划。

5.1.1 估算计划。这一节讨论估算项目周期和成本所使用的技术，以及跟踪这些估算的方法，如果需要的话，应在项目进展过程中进行调整。

5.1.2 人员安置计划。这一节列出了项目需要的人员类型和数量，以及需要他们的时间周期。

5.1.3 资源获取计划。这一节讨论获取必要的资源的方法，这些资源包括硬件、软件、服务合同和行政管理服务。

5.1.4 项目人员培训计划。这一节列出了成功完成项目所需的所有培训。

5.2 工作计划。

5.2.1 工作行为。这一节详细说明了必要时直到任务级的工作行为。

5.2.2 时间表分配。通常工作包是互相依赖的，且更依赖外部事件，例如，实现阶段在设计阶段之后，而在产品测试阶段之前。在这一小节里将详细说明它们之间的相关性。

5.2.3 资源分配。给前面列出的各种资源分配了合适的功能、行为和任务。

5.2.4 预算分配。在这一小节里，整个预算在项目功能、行为和任务的等级上被分解。

5.3 控制计划。

5.3.1 需求控制计划。如本书第二部分所描述，当开发软件产品时，需要不断地更改需求。这一节描述了用来监视和控制需求变化的机制。

5.3.2 时间表控制计划。在这一小节里，列出了测量进展的机制，并描述了如果实际的进展落后于计划的进展时应采取的行动。

5.3.3 预算控制计划。花销不能超过预算的数额这很重要，本节描述了当实际的成本超过预算成本时监视的控制机制和所应采取的措施。

5.3.4 质量控制计划。这一节描述了测量和控制质量的方法。

5.3.5 报表计划。为监视需求、时间表、预算和质量，需要实行报表机制。这一节讨论这些机制。

5.3.6 度量收集计划。如5.3节所说明的，不测量相关的度量是不可能管理开发过程的，这一节列出了需要收集的度量。

5.4 风险管理计划。风险需要确定、分清优先次序、减轻和跟踪。这一节描述风险管理的所有方面。

5.5 项目打结计划。这一节描述了一旦项目完成需要采取的行为，包括人员的再分配和产品存档。

6 技术过程计划。

6.1 过程模型。这一节详细描述了所使用的生命周期模型。

6.2 方法、工具和技术。这一节描述了所使用的开发方法和编程语言。

6.3 基础结构计划。这一节详细描述了硬件和软件的技术方面，应包含的事项有开发软件产品用到的计算系统（硬件、操作系统、网络和软件），将要运行软件产品的目标计算系统和使用的CASE工具。

6.4 产品验收计划。为确保完成的软件产品通过验收测试，必须提出验收标准，客户必须书面同意该标准，然后开发者确保真正地达到这些标准。这一节将讨论验收过程的这三个阶段产生的方式。

7 支持过程计划。

7.1 配置管理计划。这一节具体描述将产品置于配置管理下的方法。

7.2 测试计划。测试与软件开发的其他方面一样需要仔细计划。

7.3 归档计划。这一节描述了所有种类的文档，不论在项目结束时是否交付给客户。

7.4 质量保证计划。本节围绕质量保证的所有方面，包括测试、标准和评审，进行讨论。

7.5 评审和审计计划。这一节描述了诸如如何进行评审这样的细节。

7.6 问题解决计划。在开发软件产品的过程中，问题肯定会出现。例如，一个设计评审可能会发现分析阶段的一个致命错误，要求更改几乎完成了的产品。这一节描述处理这些问题的方式。

7.7 转包商管理计划。这一节适用于需要转包商提供工作产品时，讨论了选择和管理转包商的途径。

7.8 过程提升计划。这一节包含了过程提升策略的内容。

8 附加计划。对某些项目，计划里需要出现附加的部分。根据 IEEE 框架，它们出现在计划的最后。附加的部分可能包括保证计划、安全计划、数据变换计划、安装计划和软件产品交付后的维护计划。

## 9.6 计划测试

SPMP 经常被忽视的一部分是**测试计划**。像软件开发的其他每个行为一样，测试必须经过计划。SPMP 必须包含测试的资源，而且在每个阶段中必须有明确的具体时间表来指示要进行的测试。

如果没有测试计划，一个项目可能会以多种方式出错。例如，在产品测试期间（3.7.4 节），SQA 小组必须检查客户签署过的规格说明文档中的每个方面是否都在已完成的产品中实现。在这项任务中，帮助 SQA 小组的一个好方法是要求开发是可跟踪的（3.7 节），也就是说，必须能够将规格说明文档中的每一条都联系到设计中的一部分，而设计中的每个部分必须在代码中有明确的对应。实现这一点的一个技术是对规格说明文档中的每一条进行编号，并确保这些编号在设计和生成的代码中有所对应。然而，如果测试计划不规定这是必做的，那么分析、设计和代码制品将不太可能会做适当的标注，结果，当产品最终进行测试时，SQA 小组要确定该产品是否是对规格说明的完全实现将非常困难。事实上，可跟踪性应该与需求阶段同步开始，需求文档中的每一条陈述（或快速原型中的每一部分）必须与分析阶段的规格说明文档中的部分相联系。

审查的一个强大方面是在审查期间检测到的错误的详细清单。假设一个小组正在审查产品的规格说明，如 6.2.3 节所述，错误清单有两种用途。首先，从审查中得到的错误统计必须与前面的规格说明审查中错误统计的累积平均值进行比较，偏离前面的平均数说明项目内部存在问题。其次，当前的规格说明审查中统计的错误必须提交给产品的设计和代码审查。毕竟，如果存在大量的某种类型的错误，那么在规格说明的审查期间很可能检测不出全部的这种错误，而设计和代码审查提供了额外的机会来定位这种类型的剩余错误。然而，除非测试计划中规定需要仔细地记录所有错误的细节，否则这项任务不太可能完成。

测试代码模块的一个重要方式是黑盒测试（14.11 节），它使用根据规格说明提出的测试用例来执行代码。SQA 小组的成员浏览整个规格说明，并提出测试用例来检验代码是否符合规格说明。提出黑盒测试用例的最好时间是在分析阶段的最后，此时规格说明文档的细节依然清晰地保留在审查它们的 SQA 小组成员的头脑中。然而，除非测试计划明确地规定黑盒测试用例需要在这个时候进行选择，否则极有可能在以后匆忙地一股脑提出一些黑盒测试用例。也就是说，只在编程小组感到有压力，需要 SQA 小组审查通过它的模块以便能够将这些模块集成为整个产品时，才快速地汇编出有限数量的测试实例，这样做只会损害产品

的整体质量。

所以，每个测试计划必须规定出测试要做什么、什么时候做以及如何做。这样的一个测试计划是 SPMP 的 7.2 节的主要部分，没有它，整个产品的质量无疑会受到损害。

## 9.7　计划面向对象的项目

假设使用传统的范型，从概念上看，生成的产品即使由分离的模块组成，通常也是一个大的单元。相反，使用面向对象的范型会使产品包含一些相对独立的较小组成部分，被称为类。这使计划相对更容易进行，对较小的单元进行成本和周期的估算将更容易和更准确。当然，估算必须考虑到产品不只是它的各部分的和，分离的部分不是完全独立的，它们可以调用另一个，而这些影响是一定不能忽视的。

本章中所描述的成本和周期估算技术能够用于面向对象范型吗？COCOMO II（9.2.4 节）是设计用于解决现代软件技术问题的，包括面向对象，但是像功能点（9.2.1 节）这样的早期度量和中间 COCOMO（9.2.3 节）怎么办？在中间 COCOMO 的情况下，需要对某些成本因子做较小的改变［Pittman，1993］。除此以外，传统范型的估算工具似乎在面向对象项目上工作得相当好——假如没有重用。重用以两种方式进入面向对象范型：在开发期间重用现有产品，以及预先考虑好的组件的生产（在当前项目期间），以便在将来的产品中重用。这两种形式的重用都影响估算过程。在开发期间的重用明显会降低成本和周期，已经发表的公式显示了该节省是这个重用的函数［Schach，1994］，但是这些结果与传统范型有关。在目前，还没有可用的信息表明在一个面向对象产品的开发中使用重用时，成本和周期是如何改变的。

我们现在转到重用当前项目的一部分内容的目标上来，与类似的非可重用组件相比，需要花费三倍的时间来设计、实现、测试和归档一个可重用组件［Pittman，1993］。必须修改成本和周期估算，以包括这个附加的工作，而 SPMP 作为一个整体必须调整，以包括重用努力的结果。因此，两个重用行为在两个相反的方向上起作用。重用现有组件可降低开发面向对象产品的总工作量，与此同时，设计组件以便在将来的产品中重用又增加了工作量。从长远来看，因重用类带来的节省优于原始开发的成本，已有一些证据证明了这个观点［Lim，1994］。

## 9.8　培训需求

当与客户讨论**培训**的主题时，通常的反应是“在产品完成前我们不需要为培训担心，然后我们可以培训用户。”这是有点令人遗憾的说法，它意指只有用户需要培训。事实上，开发小组的成员也需要培训，从软件计划和估算就开始培训。当使用诸如新的设计技术或测试过程这样的新软件开发技术时，必须给使用新技术的每个小组成员提供培训。

面向对象范型的引入有较大的培训意义，引入诸如工作站或集成环境（参见 14.24.2 节）这样的硬件或软件工具也需要培训。程序员可能需要在开发产品所使用的机器操作系统和实现语言方面进行培训。文档准备的培训经常被忽视，这已被这么多质量差的文档所证明了。计算机操作者当然需要某种能运行新产品的培训，如果使用新硬件，他们还可能需要额外的培训。

有许多方式获得所要求的培训，最容易和最具连续性的培训是由雇员中的同事或顾问进行的内部培训。许多公司提供各种培训课程，而且大学也在晚上提供培训课程。基于因特网

的课程是另一种获取培训的途径。

一旦确定了培训需求，并提出了培训计划，则该计划必须要与 SPMP 结合起来。

## 9.9　文档标准

软件产品的开发还伴随着各类**文档**。Jones 发现，规模大约为 50 KDSI 的 IBM 内部的商用产品中每 1000 行语句（KDSI）会产生 28 页文档，而相同规模的商用软件中每 KDSI 会产生 66 页文档。IMS/360 操作系统的版本 2.3 大约有 166 KDSI 大小，每 KDSI 产生 157 页文档。文档有各种类型，包括计划方面、控制方面、商业方面以及技术方面的［Jones，1986a］。除了这些文档类型，源代码本身也是一种文档形式，代码内的注释构成了更进一步的文档。

相当多的一部分软件开发工作量体现在文档上，一个对 63 个开发项目和 25 个交付后维护项目的调查表明，如果花费在与代码相关的行为上的时间为 100 小时，则有 150 小时的时间花费在与文档相关的行为上［Boehm，1981］。对于大型的 TRW 产品，如果花费在与代码相关的行为上的时间为 100 小时，与文档相关的行为所花费的时间则上升到 200 小时［Boehm et al.，1984］。

每种类型的文档都需要有标准。例如，设计文档的一致性能减少小组成员之间的误会，并能够帮助 SQA 小组。尽管新雇员仍需要培训文档标准，但已有的雇员在公司内部从一个项目转到另一个项目时不需要再进行培训。从交付后维护的角度看，一致的编码标准有助于维护程序员理解源代码。对于用户手册，标准化更为重要，因为会有各种人来阅读用户手册，而这些人中很少会有计算机专家。IEEE 开发了用户手册标准（软件用户文档的 IEEE 1063 标准）。

作为计划过程的一部分，在软件生产期间必须要为生成的所有文档建立标准，这些标准结合在 SPMP 中。那里使用的是诸如软件测试文档的 ANSI/IEEE 标准［ANSI/IEEE 829，1991］这样的已有标准，在 SPMP 的第 2 节（参考材料）中列出了这些标准。如果一个标准是专门为此次开发所写的，则它应出现在 6.2 节（方法、工具和技术）。

文档是软件生产工作的重要部分，从非常现实的意义来说，产品“就是”文档，因为如果没有文档，就不能维护产品。非常详细地计划文档，然后确保按照计划来实施，是成功的软件产品的一个重要部分。

## 9.10　用于计划和估算的 CASE 工具

有许多可用的工具来使中间 COCOMO 和 COCOMO II 自动化。为追求修改参数值时计算的速度，一些中间 COCOMO 的实现是以电子表格的语言写成的，例如 Lotus 1-2-3 或 Excel。为了开发和更新计划本身，文字处理器是必需的。

管理信息工具对于计划也很有用，例如，假设一个大型软件公司拥有 150 名程序员，那么制定计划工具可帮助计划者确定哪个程序员已经分配了特定的任务，而哪些程序员可以完成当前的任务。

还需要更通用的管理信息类型。一些市场上可买到的管理工具可用于协助计划和估算过程，并监视整个开发过程。这些工具包括 MacProject 和 Microsoft Project。

## 9.11　测试软件项目管理计划

正如本章开始时所指出的，软件项目管理计划中的错误对于开发者来说，会牵涉严重的商业问题。开发组织既不高估、也不低估项目的成本和周期很重要。因此，整个 SPMP 必须由 SQA 小组在估算提交给客户之前进行检查。测试计划的最好方式是进行计划审查。

计划审查小组必须仔细评审 SPMP，特别注意成本和周期的估算。为进一步减少风险，不论使用何种度量，计划小组一确定完估算，就应该由 SQA 小组的一个成员独立地计算周期和成本估算。

## 本章回顾

本章的主题是软件过程中计划的重要性（9.1 节）。任何软件项目管理计划的一个重要部分是估算周期和成本（9.2 节）。为估算产品的规模提出了几种度量，包括功能点（9.2.1 节）。接下来，描述了各种成本估算用到的度量，特别是中间 COCOMO（9.2.3 节）和 COCOMO II（9.2.4 节）。如 9.2.5 节所述，跟踪所有的估算非常重要。软件项目管理计划的三个主要部分——要做的工作、所用到的资源和实现项目所要花费的金钱——在 9.3 节中讲述。一种特别的 SPMP，IEEE 标准在 9.4 节中提出，并在 9.5 节中详细描述。接下来的各节是关于计划测试（9.6 节）、计划面向对象项目（9.7 节）以及培训需求、文档标准和它们对计划过程的影响（9.8 节和 9.9 节）。计划和估算的 CASE 工具在 9.10 节描述。最后本章以测试软件项目管理计划方面的材料结束（9.11 节）。

## 进一步阅读

Weinberg 的四卷本著作［Weinberg，1992，1993，1994，1997］提供了关于软件管理的许多方面的详尽信息，［Bennatan，2000，and Reifer，2000］也是一样。管理软件项目的度量在 Weller［1994］中有所描述。

关于面向对象范型的管理，应该参考 Pittman［1993］和 Nesi［1998］，要进一步了解软件项目管理计划方面的 IEEE 1058 标准，应该仔细阅读该标准［IEEE 1058，1998］。McConnell［2001］描述了详细计划的需求。

Sackman 的经典著作在［Sackman，Erikson and Grant，1968］中有所描述，［Sackman，1970］中有更详细的资源。

［Low and Jeffrey，1990］中有关于功能点的有用信息，对功能点的仔细分析和建议的改进措施出现在［Symons，1991］中，对功能点的评论出现在［Kitchenham，1997］中。

功能点的可靠性在［Kemerer and Porter，1992］和［Kemerer，1993］中讨论。在［Furey and Kitchenham，1997］中讨论了功能点的优缺点。有关功能点的所有方面的综合资源在［Boehm，1997］中可找到。

在［Boehm，1981］中包含有中间 COCOMO 的理论证明和实现它的全部细节，在［Boehm，1984］中可找到它的简本。COCOMO II 在［Boehm et al.，2000］中有所描述。加强 COCOMO 预测的方式在［Smith，Hale and Parrish，2001］中有所描述。

［Briand and Wüst，2001］描述了如何估算面向对象产品的开发工作量。［Cartwright and Shepperd，2000］描述了对面向对象软件产品的规模和缺点的估算。各种商业数据处理产品的软件生产率数据在［Maxwell and Forselius，2000］中提供，所使用的生产率单位是

每小时的功能点。以第四代语言编写的软件规模估算在［Dolado，2000］中提供。《IEEE Software》杂志 2000 年 11 月/12 月刊中有关于估算的各种文章。

## 习题

9.1　你认为为什么一些软件公司冷嘲热讽地把“milestone”（里程碑）称作“millstone”（磨石）（提示：请在字典里查找一下 millstone 的喻义）?

9.2　你是 Bantry Bay Software Developers 公司的软件工程师，一年前，你的经理宣布你的下一个产品将包含 10 个文件、53 个数据流和 81 个过程：

a）使用 FFP 度量确定它的规模。

b）对于 Bantry Bay Software Developers 公司来说，等式（9.2）中的常数 $d$ 确定为 904 美元，FFP 度量预测的成本估算是什么?

c）该产品最近以 123 500 美元的成本完成，那么你的开发小组的生产率是多少?

9.3　目标产品有 8 个简单的输入、1 个一般的输入和 9 个复杂的输入，有 41 个一般的输出、7 个简单的查询、16 个一般的主文件和 14 个复杂的接口。请确定未经调整的功能点（UFP)?

9.4　如果习题 9.3 中的产品的总影响度为 46，请确定功能点数。

9.5　为什么代码行（LOC 或 KDSI）尽管有缺点，还是作为产品规模的度量而被广泛应用?

9.6　你负责开发一个有 76 KDSI 的嵌入式产品，除了数据库规模非常高，以及软件工具的使用非常低以外，其他都是额定的。请问如果使用中间 COCOMO，估算的工作量以人月计算将是多少?

9.7　你负责开发两个 43 KDSI 的有组织模式的产品，除了产品 $P_1$ 具有特别高的复杂度，以及产品 $P_2$ 具有特别低的复杂度以外，两个产品在每个方面都是额定的。为开发产品，你可以支配两个小组。小组 A 具有非常高的分析能力、应用经验和编程能力，小组 A 还具有很高的虚拟机经验，而小组 B 在这五个属性上的级别都非常低。

a）如果小组 A 开发产品 $P_1$，小组 B 开发产品 $P_2$，总的工作量是多少（以人月为单位)?

b）如果小组 B 开发产品 $P_1$，小组 A 开发产品 $P_2$，总的工作量是多少（以人月为单位)?

c）前面的两种人员分配中哪一个更合理？中间 COCOMO 的预测支持你的直觉吗?

9.8　你负责开发一个 52 KDSI 的有组织模式的产品，该产品在每个方面都是额定的。

a）假设每人月的成本是 9700 美元，该项目的估算成本将是多少?

b）在项目开始时你的整个开发小组辞职了，但你很幸运，有机会以一个具有非常丰富经验和高能力的小组替代原来的小组，但是每人月的成本将上升到 12 600 美元。你认为人员这样变化后会赚得（或失去）多少钱?

9.9　你负责开发一个软件，它使用一系列新开发的算法来为大型的货车运输公司计算最划算的路线。通过使用中间 COCOMO，你确定该产品的成本将是 410 000 美元，然而，为了进行检验，你让小组中的一个成员使用功能点来计算工作量。她的报告说功能点量度预测的成本为 885 000 美元，是你的 COCOMO 预测的两倍多，此时你将怎样做?

9.10　证明当 $t = k$ 时，瑞利分布（等式（9.9））达到最大值，并找到对应的资源消耗。

9.11　一个产品的维护计划被认为是 IEEE 软件项目管理计划的“额外的部分”，但是要记住每个较大的产品都需要维护，而平均来说，交付后维护的成本大约是开发该产品的

成本的两倍或三倍，应该如何证明这个结论？

9.12　（学期项目）考虑附录 A 描述的 Ophelia 的 Oasis 项目，为什么不可能单纯根据附录 A 中的信息估算成本和周期？

9.13　（软件工程读物）你的导师将分发一些［Briand and Wüst，2001］的复印件，你同意“进行面向对象的分析和设计时成本估算将是准确的”这个观点吗？

## 参考文献


[Albrecht, 1979] A. J. ALBRECHT, "Measuring Application Development Productivity," *Proceedings of the IBM SHARE/GUIDE Applications Development Symposium,* Monterey, CA, October 1979, pp. 83–92.

[ANSI/IEEE 829, 1991] *Software Test Documentation,* ANSI/IEEE 829-1991, American National Standards Institute, Institute of Electrical and Electronic Engineers, New York, 1991.

[Bennatan, 2000] E. M. BENNATAN, *On Time within Budget: Software Project Management Practices and Techniques,* 3rd ed., John Wiley and Sons, New York, 2000.

[Boehm, 1981] B. W. BOEHM, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Boehm, 1984] B. W. BOEHM, "Software Engineering Economics," *IEEE Transactions on Software Engineering* **SE-10** (January 1984), pp. 4–21.

[Boehm, 1997] R. BOEHM (EDITOR), "Function Point FAQ," at ourworld.compuserve.com/homepages/softcomp/fpfaq.htm, June 25, 1997.

[Boehm et al., 1984] B. W. BOEHM, M. H. PENEDO, E. D. STUCKLE, R. D. WILLIAMS, AND A. B. PYSTER, "A Software Development Environment for Improving Productivity," *IEEE Computer* **17** (June 1984), pp. 30–44.

[Boehm et al., 2000] B. W. BOEHM, C. ABTS, A. W. BROWN, S. CHULANI, B. K. CLARK, E. HOROWITZ, R. MADACHY, D. REIFER, AND B. STEECE, *Software Cost Estimation with COCOMO II,* Prentice Hall, Upper Saddle River, NJ, 2000.

[Briand and Wüst, 2001] L. C. BRIAND AND J. WÜST, "Modeling Development Effort in Object-Oriented Systems Using Design Properties," *IEEE Transactions on Software Engineering* **27** (November 2001), pp. 963–86.

[Cartwright and Shepperd, 2000] M. CARTWRIGHT AND M. SHEPPERD, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Transactions on Software Engineering* **26** (August 2000), pp. 786–95.

[Devenny, 1976] T. DEVENNY, "An Exploratory Study of Software Cost Estimating at the Electronic Systems Division," Thesis No. GSM/SM/765–4, Air Force Institute of Technology, Dayton, OH, 1976.

[Dolado, 2000] J. J. DOLADO, "A Validation of the Component-Based Method for Software Size Estimation," *IEEE Transactions on Software Engineering* **26** (October 2000), pp. 1006–21.

[Furey and Kitchenham, 1997] S. FUREY AND B. KITCHENHAM, "Function Points," *IEEE Software* **14** (March/April 1997), pp. 28–32.

[IEEE 1058, 1998] *IEEE Standard for Software Project Management Plans,* IEEE Std. 1058-1998, Institute of Electrical and Electronic Engineers, New York, 1998.

[Jones, 1986a] C. JONES, *Programming Productivity,* McGraw-Hill, New York, 1986.

[Jones, 1987] C. JONES, Letter to the Editor, *IEEE Computer* **20** (December 1987), p. 4.

[Kemerer, 1993] C. F. KEMERER, "Reliability of Function Points Measurement: A Field Experiment," *Communications of the ACM* **36** (February 1993), pp. 85–97.

[Kemerer and Porter, 1992] C. F. KEMERER AND B. S. PORTER, "Improving the Reliability of Function Point Measurement: An Empirical Study," *IEEE Transactions on Software Engineering* **18** (November 1992), pp. 1011–24.

[Kitchenham, 1997] B. KITCHENHAM, "The Problem with Function Points," *IEEE Software* **14** (March/April 1997), pp. 29, 31.

[Lim, 1994] W. C. LIM, "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software* **11** (September 1994), pp. 23–30.

[Low and Jeffrey, 1990] G. C. LOW AND D. R. JEFFREY, "Function Points in the Estimation and Evaluation of the Software Process," *IEEE Transactions on Software Engineering* **16** (January 1990), pp. 64–71.

[Maxwell and Forselius, 2000] K. D. MAXWELL AND P. FORSELIUS, "Benchmarking Software Development Productivity," *IEEE Software* **17** (January/February 2000), pp. 80–88.

[McConnell, 2001] S. MCCONNELL, "The Nine Deadly Sins of Project Planning," *IEEE Software* **18** (November/December 2001), pp. 5–7.

[Nesi, 1998] P. NESI, "Managing OO Projects Better," *IEEE Software* **15** (July/August 1998), pp. 50–60.

[Norden, 1958] P. V. NORDEN, "Curve Fitting for a Model of Applied Research and Development Scheduling," *IBM Journal of Research and Development* **2** (July 1958), pp. 232–48.

[Pittman, 1993] M. PITTMAN, "Lessons Learned in Managing Object-Oriented Development," *IEEE Software* **10** (January 1993), pp. 43–53.

[Putnam, 1978] L. H. PUTNAM, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering* **SE-4** (July 1978), pp. 345–61.

[Reifer, 2000] D. J. REIFER, "Software Management: The Good, the Bad, and the Ugly," *IEEE Software* **17** (March/April 2000), pp. 73–75.

[Sackman, 1970] H. SACKMAN, *Man–Computer Problem Solving: Experimental Evaluation of Time-Sharing and Batch Processing*, Auerbach, Princeton, NJ, 1970.

[Sackman, Erikson, and Grant, 1968] H. SACKMAN, W. J. ERIKSON, AND E. E. GRANT, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM* **11** (January 1968), pp. 3–11.

[Schach, 1994] S. R. SCHACH, "The Economic Impact of Software Reuse on Maintenance," *Journal of Software Maintenance: Research and Practice* **6** (July/August 1994), pp. 185–96.

[Smith, Hale, and Parrish, 2001] R. K. SMITH, J. E. HALE, AND A. S. PARRISH, "An Empirical Study Using Task Assignment Patterns to Improve the Accuracy of Software Effort Estimation," *IEEE Transactions on Software Engineering* **27** (March 2001), pp. 264–71.

[Symons, 1991] C. R. SYMONS, *Software Sizing and Estimating: Mk II FPA*, John Wiley and Sons, Chichester, UK, 1991.

[van der Poel and Schach, 1983] K. G. VAN DER POEL AND S. R. SCHACH, "A Software Metric for Cost Estimation and Efficiency Measurement in Data Processing System Development," *Journal of Systems and Software* **3** (September 1983), pp. 187–91.

[Weinberg, 1992] G. M. WEINBERG, *Quality Software Management: Systems Thinking*, Vol. 1, Dorset House, New York, 1992.

[Weinberg, 1993] G. M. WEINBERG, *Quality Software Management: First-Order Measurement*, Vol. 2, Dorset House, New York, 1993.

[Weinberg, 1994] G. M. WEINBERG, *Quality Software Management: Congruent Action*, Vol. 3, Dorset House, New York, 1994.

[Weinberg, 1997] G. M. WEINBERG, *Quality Software Management: Anticipating Change*, Vol. 4, Dorset House, New York, 1997.

[Weller, 1994] E. F. WELLER, "Using Metrics to Manage Software Projects," *IEEE Computer* **27** (September 1994), pp. 27–34.

# 第二部分　软件生命周期的各个阶段

在第二部分中，将深入描述软件生命周期的各个阶段。对于每个阶段，给出了适合于该阶段的行为、CASE 工具、度量和测试技术，同时也指出了该阶段面临的挑战。

第 10 章“需求”研究需求阶段。这一阶段的目标是确定客户的真正要求，这一章研究了各种需求分析技术。

一旦明确了需求，下一步就是拟制出规格说明书。第 11 章“传统的分析”描述了传统的方法，这一章中给出了提出规格说明的三种基本方法：非正式的、半正式的和正式的，对每一种方法都给出了实例描述。深入描述和通过实例研究举例说明的技术包括结构化系统分析、有限状态机、Petri 网和 Z。在这一章还对各种技术进行了比较。

第 11 章中的所有分析技术都来自于传统范型。第 12 章“面向对象分析”描述了面向对象的方法，这个面向对象的技术是作为前一章的传统分析技术的替代而给出的。

在第 13 章“设计”中，对各种设计技术进行了比较，包括像数据流分析和事务处理分析这样的传统技术以及面向对象设计。这里，对面向对象设计给予了特别的关注，其中还包括了实例研究。同样，在这里将重点放在比较和对照上。

第 14 章“实现”讨论了实现问题，涉及的领域包括实现的事项、集成的事项、好的编程实践和编程标准。

第 15 章标题为“交付后维护”，讨论的主题包括交付后维护的重要性和面临的挑战，在某些细节上还考虑了交付后维护的管理。

第 16 章“UML 的进一步讨论”提供了关于统一建模语言的额外信息。

在第二部分的最后，你应能够清楚理解软件过程的所有阶段，各阶段面临的挑战和如何应对那些挑战。

# 第10章 需　求

**学习目标**

学完本章之后，你应当能够：

- 完成需求阶段的任务；
- 提出初始商业模型；
- 提出需求；
- 构造快速原型。

将一个软件产品按时而又不超出预算地开发出来的机会有时是很小的，除非软件开发小组的成员都对软件产品将做什么意见一致。达到这种全体一致的第一步是尽可能精确地分析客户当前的状态形势，例如，这样说是不合适的："因为他们报怨他们的人工设计系统很糟，所以他们需要一个计算机辅助设计系统。"除非软件开发小组明确地知道当前的人工系统有什么问题，否则，很有可能新的计算机系统将会在许多方面一样地"糟"。同样，如果一个个人计算机制造商正打算开发一个新的操作系统，第一步是评价企业当前的操作系统，并认真准确地分析为什么它不能令人满意。举个极端的例子，知道以下几方面至关重要：是否问题仅存在于销售商的头脑里？谁责怪操作系统销售不好？或者是否该操作系统的用户完全不认同它的功能和可靠性？仅当对于当前情形有了一个清晰的认识之后，软件开发小组才能够试图回答关键的问题，即：新的软件产品必须能够做什么？回答这个问题的过程是需求阶段的基本目标。

## 10.1 确定客户需要什么

人们通常持有的一个错误概念是，在需求阶段，开发者必须确定客户想"要"什么样的软件。与此相反，需求阶段真正的目标是确定客户"需要"什么样的软件。问题在于许多客户不知道他们需要什么，更进一步说，即使一个客户对什么是他们所需要的有一个好的想法，他可能难于准确地将这些想法传达给开发者，因为大多数客户的计算机知识比起软件开发小组成员来讲要少得多。（若要进一步了解这个问题，参见下面的"如果你想知道"部分。）

**如果你想知道**

来自加利福尼亚州的美国议员 S.I.Hayakawa（1906—1992）曾有一次告诉一群记者："我知道你自以为理解了我所说的，但是我不确定你能认识到你听到的并不是我所意指的。"这个托辞同样很适用于需求分析的问题。开发人员听到客户的要求，但他们所听到的不是客户所应当说的。

上述言论曾被错误地认为是前美国总统候选人 George Romney（1907—1995）所说的，他曾在一个记者会上说："我不是说我没说过它，我说的是我没有说我说了它。我想要把那分清楚。"Romney 的"澄清"强调了需求分析的另一个挑战——很容易误解客户所说的。

另一个问题是客户可能没有意识到他或她自己的公司正在做什么。例如，当设计很差的

数据库是客户当前软件产品具有如此长的响应时间的真正原因时，客户要求得到一个更快的软件产品是没有用的，需要做的是重新组织和加强数据存储在当前的软件产品中的方式。一个新的软件产品也将同样慢。或者，如果客户经营一个不赚钱的零售连锁店，客户可能要求研制一个商用管理信息系统，能够反映诸如销售、工资、应付款和应收款的这些事项。但如果不赚钱的真正原因是贬值（入店行窃和内盗），那么这样的信息系统将起不到任何作用，反而是库存控制系统比商用管理信息系统更为需要。

乍一看，确定客户需要什么是简单的——开发小组的成员只要询问他或她即可，然而，有两个原因说明了为什么这个直接的方法通常不太起作用。

首先，就像前面提到的，客户可能没有意识到他或她自己的公司正在做什么。但客户经常要求一个错误的软件产品的主要原因是软件很复杂。对于系统分析师而言，将软件产品和它的功能可视化已经相当难了，而这种问题对于通常不是软件工程专家的客户来说则更加困难了。

没有专业的软件开发小组的协助，客户很难了解到需要开发什么的相关信息。另一方面，除非与客户有面对面的交流，否则很难弄清楚他们真正需要什么。

解决这个问题的传统方法在10.11节中有描述，面向对象的方法是从客户和目标产品的未来用户那里得到原始的信息，并使用这些信息作为统一过程［Jacobson，Booch，and Rumbaugh，1999］需求阶段的输入，下一节将谈到这一点。

## 10.2 需求阶段概述

**需求阶段**的第一步是理解**应用域**（简称**域**），也就是目标产品应用的特定环境，这个域可以是银行、空间探索、汽车加工或遥感勘测。一旦开发小组成员达到了对该域的充分理解，就可以建造一个商业模型，也就是使用UML图来描述客户的商业过程。该商业模型用来确定客户的初始需求，然后应用迭代法。

换句话说，开始点是对域的初始理解，应用这个信息建造初始的商业模型，而这个初始的商业模型被用来提出一系列初始的客户需求。然后，在了解客户需求的基础上，再更深入地理解应用域，并回过头来推敲商业模型，并因此得到客户的需求。这个迭代过程持续进行着，直到开发小组对需求真正满意，这时迭代结束了。

发现客户需求的过程被称为**需求启发**（或**需求捕获**）。一旦提出了最初的需求，推敲和扩充的过程被称为**需求分析**。

现在我们具体讨论这些步骤。

## 10.3 理解应用域

为了启发出客户的要求，需求小组的成员必须熟悉应用领域，即目标软件产品通常要在哪些领域使用。例如，如果首先没有对银行业或护理专业有某种程度上的熟悉，就不太容易向一个银行家或护士问出有意义的问题。因此，每个需求分析小组成员最初的任务之一就是获得对应用领域的熟悉，除非他（或她）已经在那个领域有过一些经历。当与客户和目标软件的潜在用户交流时，特别重要的一点是使用正确的术语。毕竟，这一点很难引起工作在某一特定领域的人的重视，除非访谈者使用适于该领域的术语。更重要的是，使用一个不合适的术语可能会导致曲解，甚至会造成交付一个有错误的软件产品的结果。如果需求小组的成

员不理解该领域术语的细微差别，可能会产生同样的问题。例如，在一个外行人看来，brace（支柱）、beam（横梁）、girder（大梁）和 strut（撑杆）这些词看起来可能像是同义词，但是对于一个土木工程师而言，它们是截然不同的词汇。如果一个软件开发人员对于一个土木工程师正在以一种精确的方式使用这四个词感到不以为然，并且如果此土木工程师想当然地认为，该开发人员对这些语汇间的差别已很熟悉，而这个开发人员可能将这四个词等同对待，由此产生的计算机辅助桥梁设计软件可能包含错误，造成桥梁倒塌。职业的计算机人员希望在根据某一程序做决定前，每个程序的输出由人来仔细地检查，但是，对计算机越来越普遍的信任意味着依赖这类检查的似然性显然是不明智的。因此，对术语的误解会造成软件开发人员被控疏忽决不是危言耸听。

解决术语问题的一个办法是建立一个**术语表**——在该领域应用的技术词汇列表和对应的解释。当小组成员正忙于尽可能学习应用领域的相关知识时，就将初始的词条插入到术语表中。然后，需求小组成员一遇到新的术语就将该术语表更新。适当时候还可打印出该术语表并分发给小组成员或下载到 PDA（例如 Palm Pilot）。这样的术语表不仅减少了客户和系统分析师之间的误解，它在减少开发小组成员之间的误解方面也是很有用的。

一旦需求小组成员熟悉了该应用领域之后，下一步就是建立商业模型。

## 10.4　商业模型

**商业模型**是对公司的商业过程进行的描述。例如，银行的一些商业过程包括从客户那里接受存款、贷款给客户和进行投资。

建立商业模型的原因首先是商业模型提供了对客户整体商业行为的理解，通过这个理解，开发者可以向客户提出建议，需要对客户生意的哪个部分进行计算化。或者，如果任务是扩充一个已有的软件产品，开发者必须把已有的产品作为一个整体来理解，以确定如何加入扩充的部分，并知道已有产品的哪个部分需要调整，以加入新的部分。

为建立商业模型，系统分析师需要对各种商业过程有具体的理解，这些过程现在是经过提炼的，也就是说，经过了更仔细的分析。获得建立商业模型所需的信息可以使用许多不同的技术，主要是访谈。

### 10.4.1　访谈

需求小组的成员会见客户公司的成员，直到他们确信已经从客户和目标软件产品未来的用户处得到启发并获得了所有相关信息。

有两种基本类型的问题。受限回答的问题要求一个特定的答案，例如，客户可能被问到公司雇用了多少销售人员或要求的响应时间有多快。自由回答的问题则鼓励受访人畅所欲言，例如，这样向客户提问："为什么当前产品不令人满意？"从中可能会看出客户对业务的倾向的许多方面，而如果这个问题是受限回答的，则无法看清这些事实。

类似地，有两种基本类型的访谈——程式化的（structured）和非程式化的（unstructured）。在一个**程式化的访谈**中，提出特定的、预先计划好的、通常是受限回答的问题。在一个**非程式化的访谈**中，访问者可能先提出一个或两个事先准备好的受限回答的问题，但接下来的问题则根据受访者的回答而提出，大多数这些接下来问的问题是自由回答的，能够给访问者提供很宽范围的信息。

同时，如果访谈太过于非程式化也不好，如对客户说“请谈谈你的生意”，则不太可能产生多少有用的信息。换句话说，应当以一种这样的方式提出问题，它能够鼓励受访者给出范围广泛的回答，但该回答又不要超出访谈者需要信息的范围。

进行一个好的访谈有时并不容易。首先，访谈者必须熟悉应用领域；其次，如果访谈者已经决意尊重客户需求的话，访谈客户公司的成员时是没有访谈要点的。不管他或她先前被告知什么或通过其他方式了解过什么，访谈者必须带着认真倾听受访者的目的着手开始每一次访谈，与此同时，坚决地克制任何预先固有的成见，尊重客户公司的意见或客户和待开发产品的潜在用户的要求。

访谈结束后，访谈者必须准备一份书面报告，概要列出访谈的结果。非常有帮助的做法是将报告的一份拷贝送给受访者，他或她可能会想澄清某些陈述或增加一些被忽略的项目。

### 10.4.2 其他技术

访谈是获取商业模型信息的主要技术，本节描述了其他一些可以和访谈结合使用的技术。

获得关于客户公司行为信息的一种方式是给客户公司的相关人员发放**调查问卷**。当需要确定上百个人员的意见时，这项技术很有用。进一步说，一个经过客户公司的雇员认真思考后写出的书面答案可能比一个随口而出的回答更准确。然而，由一个很有经验的访谈者进行的非程式化的访谈，由于他能够认真倾听受访者并提出问题，从而将起初得到的响应大大扩展了，这样的访谈比一个经过深思熟虑的问卷调查表通常能揭示出更多的信息。因为问卷调查表是预先计划好的，所以就没有办法根据某一个回答再提出一个问题。

启发需求的另一种方法是检查客户在业务上使用的各种**表格**。例如，印刷厂的一张表格可能反映了出版号、纸张轧压尺寸、湿度、油墨温度、纸张张力等。这个表格中的各种字段显示了印刷工作的流程以及印刷过程中相关步骤的重要性。其他的文档，例如，操作流程和工作描述也是准确找出已做了什么和如何做的强有力工具。如果使用了软件产品，应该仔细地学习用户手册，这些不同类型的全面的数据反映了当前客户是如何从业的，这对确定客户需求相当有帮助。因此，一个好的软件专业人员要仔细学习客户文档，把它看作是有价值的信息源，引导出对客户需求的准确评估。

获取这样的信息的另一种方法是对用户**直接观察**，也就是由需求小组的成员观察和记下客户雇员工作的情况。这项技术的现代版是（在得到被观察对象书面允许的前提下）在工作场所安装摄像机来准确记录做了什么。这项技术的一个困难之处是需要花很长时间来分析录像带，通常需求小组的一个或多个成员需要花上一个小时回放摄像机录下的每个小时的磁带。这个时间对于评估所观察到的东西而言是额外增加的。更严重的是，这一技术据说已经引发了严重的适得其反的结果，因为雇员可能将摄像机看作是对他们个人隐私的一种不当的入侵。重要的是需求小组要与所有雇员进行全面的合作，如果人们感到受威胁或被打扰，他们就很难获得必要的信息。在引入摄像机，或者为此事采用任何其他有可能激怒雇员的行动前，必须仔细考虑到可能的风险。

### 10.4.3 用例

3.2 节中提到过，**模型**是代表要开发的软件产品的一个或多个方面的 UML 图（回想一

下“建模语言”的 UML 标准中的 ML）。在商业建模中最常用的 UML 图是用例。

用例为软件产品本身和软件产品的使用者（行动者）之间的交互建立模型。例如，图 10-1 描述了一个来自银行软件产品的用例，其中有两个行动者——顾客和出纳员，由 UML 线条画表示。椭圆内的标签描述了用例代表的商业行为，在这个实例中是 Withdraw Money。




图 10-1　银行软件产品的 Withdraw Money 用例

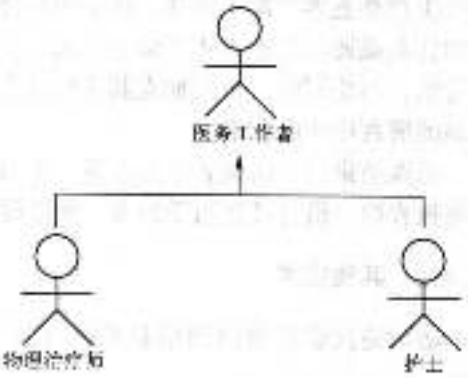



图 10-2　医务工作者的泛化

看待用例的另一种方式是，它体现了软件产品和软件产品运行环境之间的交互，也就是说，行动者是软件产品之外的一个成员，而用例中的矩形代表软件产品本身。

行动者通常很容易辨别，行动者经常会是软件产品的使用者。在银行软件产品的例子中，该软件产品的使用者是银行的顾客和银行的职员，包括出纳员、经理等。通常，行动者扮演一个与软件产品有关的角色，这个角色可能是软件产品的使用者。然而，用例的发起人或在用例中起到关键作用的某个人也正扮演一个角色，因而也被看作是一个行动者，不论这个人是否是软件产品的使用者。10.7 节给出了一个这样的例子。

系统的使用者可以扮演不止一个角色，例如，银行的顾客可以是一个“借钱者”（当他或她贷款时）或“借出者”（当他或她存款到银行里——银行通过把顾客存进的钱进行投资来获得利润）。相反地，一个行动者可以参加多个用例，例如，一个“借钱者”可以是 Borrow Money 用例、Pay Interest on Loan 用例和 Repay Loan Principal 用例里的行动者，另外，“借钱者”这个行动者代表了成百上千的银行顾客。

行动者不必一定是人。回想一下，行动者是软件产品的使用者，在许多情况下，另一个软件产品可以是使用者。例如，电子商务信息系统允许购买者用信用卡付款，就需要与信用卡公司的信息系统进行交互，也就是说，从电子商务信息系统的角度看，信用卡公司的信息系统就是行动者。类似地，从信用卡公司的信息系统角度看，电子商务信息系统也是行动者。

如前所述，行动者很容易分辨。通常情况下，在范型的这个部分会产生的惟一问题是过分热心的软件专业人员有时会把重叠的行动者看成一样。例如在一个医院软件产品中，有一个用例带有行动者“护士”，而另一个不同的用例带有行动者“医务工作者”就不太好，因为所有的护士都是医务工作者，但一些医务工作者（例如物理治疗师）不是护士，那么确定有行动者“物理治疗师”和“护士”就比较好。而行动者“医务工作者”可被定义为具有两个专业，“物理治疗师”和“护士”，如图 10-2 所示。在 7.7 节中指出过，继承是泛化的一个特例，泛化在 7.7 节中的类里有应用。图 10-2 还示意了泛化如何应用于行动者。

## 10.5 初始需求

为确定客户的需求，可基于初始的商业模型提出初始需求，然后经过与客户的进一步讨论，精炼了对应用域的理解和商业模型，从而需求也得到精炼。

需求是动态的，也就是说，不仅需求本身会经常变化，开发小组、客户和未来使用者对每个需求的态度也会变化。例如，呈现给开发小组的某项特定需求最初可能是可选项，经过进一步的分析，那个需求现在看来非常重要了，然而经过与客户讨论后，该项需求被放弃了。处理这些频繁变化的好方法是维护一个可能的需求表，带有需求的用例，这些用例得到了开发小组各成员和客户的认可。

记住面向对象范型是迭代的这一点很重要，因此术语表、商业模型或需求也会随时调整。特别是，很多种事件（从用户不经意的评论到在需求小组的系统分析师正式会议上客户提出的建议）都可能引发需求表的增加、对需求表中已存在的事项进行调整以及从需求表中去掉某些项。任何这样的变化都可能引起商业模型的相应变化。

需求分为两类，功能性需求和非功能性需求。**功能性需求**指定了目标产品必须能够执行的行为，通常用输入和输出的术语来表达功能性需求：假设一个指定的输入，功能性需求规定出必须有什么样的输出。相反地，**非功能性需求**指定了目标产品本身的属性，例如**平台限制**（“该软件产品应运行于 Linux 下”）、**响应时间**（“平均而言，3B 类型的队列应在 2.5 秒内得到应答”）或**可靠性**（“软件产品的可运行时间比率应在 99.95%以上”）。

在需求和分析阶段进行功能性需求的处理，而一些非功能性需求需要等到设计阶段才能处理，原因是要处理某些非功能性需求，需要了解目标产品的具体情况，而这些情况通常在需求和分析阶段结束以后才能清楚（参见习题 10.2 和 10.3）。然而一旦可能，应该在需求和分析阶段处理非功能性需求。

下面通过一个运行的实例研究阐述需求阶段。

## 10.6 对应用领域的初始理解：Osbert Oglesby 实例研究

Osbert Oglesby，著名的艺术品经销商，需要一个软件产品来协助他购买和销售绘画作品。Osbert 专门买卖法国印象主义画家的画作，然而，如果他认为可在他的画廊 Les Objets d’Orient 中卖出好价钱，他实际上也会购买任何画作。

很有必要对 Osbert 进行访谈，以得到关于他生意上每个方面的详细信息，但在这之前，获得应用领域的知识，也就是关于绘画和艺术市场的常识很重要。这个背景信息可使需求小组在会见 Osbert 时，理解 Osbert 的任何技术性术语。另外，如果他们在与 Osbert 会面之前知道艺术市场的常识，他们就可以标注出 Osbert 做生意时的一些明显不寻常的方面，然后询问这些事项。在一些情况下，他们可提出建议，通过了解他的主要对手在做什么，来提高他生意的收益率。在另一些情况下，Osbert 生意实践的独特方面使他处于竞争的边缘。在即将为他建立的软件产品中应考虑所有这些情况，这一点很重要。

为学习绘画和艺术市场的常识，需求小组访问了两个主要的博物馆和两个艺术品经销商的网站。他们发现在对绘画作品分类方面有许多不同的方法。一种是考虑**材质**，也就是艺术作品所采用的材料。最常见的材质是基于油的绘画（**油画**），但一些较小的绘画作品是基于水的绘画（**水彩画**）。有时也使用**其他介质**，包括铅笔、彩色蜡笔和彩色粉笔，但不太常用。

他们将这个信息放入他们的术语表里，如图 10-3 所示。

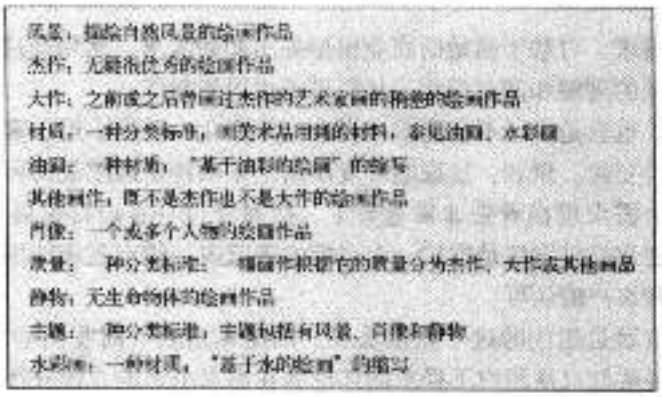



图 10-3　Osbert Oglesby 实例研究初始的术语表

对绘画作品分类的第二种方法是按**主题**分类，最常见的主题是人物（**肖像**）。其他常见的主题包括自然（**风景**）和诸如一瓶花、一盘水果等的静止物体（**静物**）。这个信息也被加入到术语表中。

第三种是按绘画作品的**质量**来分类。无疑优秀的绘画作品被称为**杰作**，**大作**是之前或之后曾画过杰作的艺术家画的稍差的绘画作品。然而，大多数的绘画作品既不是杰作，也不是大作（**其他画作**）。这个信息也被加入到术语表中，如图 10-3 所示。

现在要建立 Osbert Oglesby 实例研究的初始商业模型了。

## 10.7　初始商业模型：Osbert Oglesby 实例研究

Osbert Oglesby 的生意多年以来一直很成功，然而后来 Osbert 开始亏损。一个管理顾问分析了生意记录，结论是 Osbert 收购绘画作品的价格太高。顾问建议 Osbert 寻找一个软件产品，运行在手提电脑上，可用来确定一幅绘画作品的最高价格。Osbert 然后可将该软件用在他的画廊中，在顾客家里或办公室看画时带上这个电脑。（Osbert 称他的顾客为他的客户，然而在本书中，“客户”一词指的是需要待开发的软件产品的人，在本例中客户指的 Osbert。）

这个管理顾问推导出一个法则，该软件产品应该用来确定一幅绘画作品的最高价格，这时看这个公式还为时过早，毕竟 Miller 法则适用于所有人。在任一时刻我们受限于我们能够处理的大块信息。在此时，我们所要知道的全部信息是 Osbert 的生意过程，关于公式的细节可以等到后面的迭代时再了解。

Osbert 希望这个软件产品能够在购买画作时计算出所能提供的最高价格。与大多数商人不一样，Osbert 不赞成侃价。当他想要买一幅作品时，他开的价就是他的最终价格。类似地，当他卖一幅作品时，他也不讲价，他所要的价即是最终价格。

另外，Osbert 希望该软件产品能够尽可能地探测出艺术品市场的新趋势，他尤其关注用比预想高的价格购买某个特定画家的作品，以便能够在其他人注意到那个趋势之前就买新该画家的作品。

为了能在画作的销售价格高于 Osbert 在购入时的期望值时做出标志，该软件产品需要

保留所有购入和售出的记录。假设在他的计算机里这个信息可用，Osbert 可能想要利用它获得关于他的生意的额外信息。特别地，Osbert 目前手工生成两个报表，一个报表显示过去的一年里购入的所有画作，另一个报表显示过去的一年里售出的所有画作。只需要增加很少的额外成本，该软件产品就能够根据需要生成这两个报表。

换句话说，Osbert "想要" 的软件产品能够计算出他应为一幅画作付出的最高价格，还能探测出新的艺术趋势。而他 "需要" 的软件产品还额外能够提供购入和售出报表。如果开发者不预先确定这个事实，他们只能在软件产品交付后再补充额外的功能。记住图 1-5，进行过调整的软件产品的成本将肯定比原来系统谈好的成本要高。进一步说，事后进行调整也将使 Osbert 对开发者不满意，好的情况下他认为开发者们不胜任，坏的情况下他将把这额外的花费看作是开发者们不诚实的证据。作为选择，开发者可能会同意免费补充报表功能，然而这将意味着在给 Osbert 的软件产品上开发者有一定的损失。然而幸运的是，系统分析师通过确定 Osbert 的真正需要避免了这个问题。

现在已经得到了初始数据，可以建立初始商业模型了。Osbert 有三项商业行为：购买绘画作品、销售绘画作品以及手工生成报表。图 10-4、图 10-5 和图 10-6 显示了对应的 Buy a Painting、Sell a Painting 和 Produce a Report 用例。然而，如 10.4.3 节所讲，发起用例的人也是一个行动者，扮演着重要的角色。顾客可以发起 Buy a Painting 或 Sell a Painting 用例。当然在这两个用例中顾客扮演了重要的角色，因为他提供了可供 Osbert 输入软件产品的数据，因此顾客在这两个用例中是行动者，在接下来的迭代中加入其他的行动者是可能的。为简明起见，图 10-7 的用例图结合了这三个用例，用例图是反映了两个或多个用例的 UML 图。

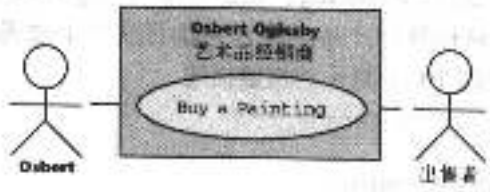



图 10-4 Osbert Oglesby 实例研究的初始商业模型的 Buy a Painting 用例




图 10-5 Osbert Oglesby 实例研究的初始商业模型的 Sell a Painting 用例

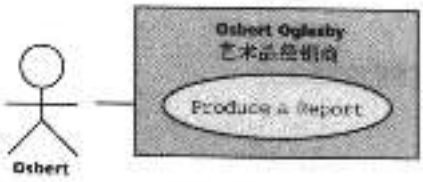



图 10-6 Osbert Oglesby 实例研究的初始商业模型的 Produce a Report 用例

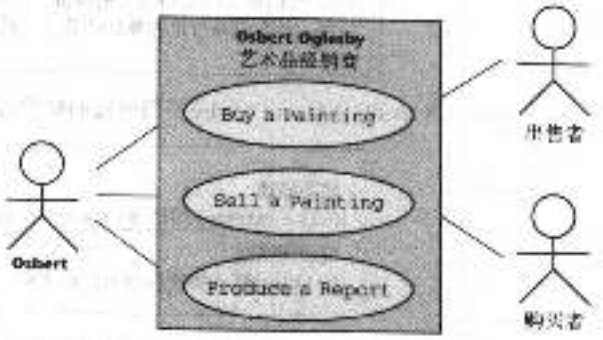



图 10-7 Osbert Oglesby 实例研究的初始商业模型的用例图

接下来需要注释用例，可以添加图 10-8 到图 10-10 显示的初始**用例描述**。这时可以提出初始的需求了。

| **简要描述** |
| --- |
| Buy a Painting 用例使 Osbert Oglesby 能够购买绘画作品。 |
| **按步骤描述** |
| 在初始阶段不适用。 |

图 10-8　Osbert Oglesby 实例研究的初始商业模型的 Buy a Painting 用例描述

| **简要描述** |
| --- |
| Sell a Painting 用例使 Osbert Oglesby 能够销售绘画作品。 |
| **按步骤描述** |
| 在初始阶段不适用。 |

图 10-9　Osbert Oglesby 实例研究的初始商业模型的 Sell a Painting 用例描述

| **简要描述** |
| --- |
| Produce a Report 用例使 Osbert Oglesby 能够获得过去一年里他买进和售出的画作的信息。 |
| **按步骤描述** |
| 在初始阶段不适用。 |

图 10-10　Osbert Oglesby 实例研究的初始商业模型的 Produce a Report 用例描述

## 10.8　初始需求：Osbert Oglesby 实例研究

初始商业模型（图 10-7 的三个用例）显示了 Osbert 的生意当前是如何进行的，下一步是确定这些用例中的哪些也是要建立的软件产品的需求。然后，由此得到的初始需求经过添加、调整和删除来得到精炼，直到 Osbert 和需求小组对完成的一整套需求满意为止。

初始商业模型的全部三个用例（图 10-7）——Buy a Painting、Sell a Painting 和 Produce a Report 将是初始的需求。进一步地说，在目标软件产品中将自动执行这三个过程，因此将用图 10-11 到图 10-13 的描述代替图 10-8 到图 10-10 显示的初始描述。

| **简要描述** |
| --- |
| Buy a Painting 用例使 Osbert Oglesby 能够购买绘画作品。 |
| **按步骤描述**<br>1. Osbert 输入他正考虑要购买的画作的细节。<br>2. 软件产品回答他应提供的最高购买价格。<br>3. 如果出售者接受 Osbert 的提价，Osbert 输入进一步的细节。<br>注释：确定最高价格的算法细节将在后面得到。 |

图 10-11　Osbert Oglesby 实例研究的初始需求的 Buy a Painting 用例描述

| **简要描述** |
| --- |
| Sell a Painting 用例使 Osbert Oglesby 能够销售绘画作品。 |
| **按步骤描述**<br>1. Osbert 输入他售出的画作的细节。 |

图 10-12　Osbert Oglesby 实例研究的初始需求的 Sell a Painting 用例描述

这些描述的不确定性是面向对象的范型的迭代特性的结果，例如，图 10-11 的第 1 条提到“Osbert 正考虑要购买的画作的细节”，但需求小组成员还不知道这些细节是什么，他们

也强调在这一阶段不希望知道。基本的原则是尽可能晚地提交所有细节，原因是在下一个迭代中不得不做一些修改时，需求调整的事项数量可以尽可能地少。

| **简要描述** |
| --- |
| Produce a Report 用例使 Osbert Oglesby 能够获得过去一年里他买进和售出的画作的信息，或者探测到艺术品市场的新趋势。 |
| **按步骤描述** |
| 1. Osbert 提出他所需要的报表类型，该报表将被打印出来。 |

图 10-13 Osbert Oglesby 实例研究的初始需求的 Produce a Report 用例描述

## 10.9 继续需求阶段：Osbert Oglesby 实例研究

考虑每个用例描述的当前版本（图 10-11 到图 10-13），可以很清楚地看到在下一个迭代中需要了解更多的细节。

首先考虑用例 Buy a Painting 和 Sell a Painting。为精炼描述，在买进画作和售出画作时需要输入的特征必须要确定。通过检查相关的文档，即 Osbert 当前的手工记录，需求小组了解到 Osbert 对每幅他买进的画作需要记录以下数据：

画作的描述：
- 画家的名（最多 20 个字符，如果不确定以？表示）
- 画家的姓（最多 20 个字符，如果不确定以？表示）
- 作品的标题（最多 40 个字符，如果不确定以？表示）
- 作品的年份（yyyy，如果不确定以？表示）
- 分类（杰作、大作、其他画品）
- 高度（cm）
- 宽度（cm）
- 材质（油画、水彩画、其他材质）
- 主题（肖像、静物、风景、其他主题）

购买时间（mm/dd/yyyy）
出售者的姓名（最多 30 个字符）
出售者的地址（最多 40 个字符）
通过算法确定的最高购买价格（最大到 99 999 999 美元）
实际购买价格（最大到 99 999 999 美元）
目标出售价格（购买价格的 2.15 倍）

当售出一幅作品时，Osbert 必须记录下列信息：

售出时间（mm/dd/yyyy）
购买者的姓名（最多 30 个字符）
购买者的地址（最多 40 个字符）
实际售出价格（最大到 99 999 999 美元）

（这些需求存在一个隐患，如下面的“如果你想知道”部分所述。）

**如果你想知道**

Osbert Oglesby 实例研究的需求表明，四个数据项（包括画作的标题）中的任何一个不确定时，在该项后需要跟一个“?”，然而画作的标题当然可能包含一个问号。例如，Paul Gauguin 1897 年居住于 Tahiti 时画了一幅恐怕是他最伟大的作品，标题为“我们从哪里来？我们是什么？我们将去何方?”

如果 Osbert Oglesby 实例研究的实现满足它的需求，那么 Osbert 将不能够正确地输入 Gauguin 的杰作的标题，尽管波士顿微观艺术博物馆极不可能将该画作卖给 Osbert（但请参见习题 15.15）。

为计算所要求的“由算法确定的最高购买价格”，需求小组现在开始考虑管理顾问的报告，该顾问开发的算法如下所述：

将绘画作品分为杰作、大作或其他画作。

- 杰作。扫描过去 25 年来同一个画家最相似作品在世界范围的拍卖记录，忽略画家的名或姓，或者标题和作品日期中的问号。（关于相似性的定义后面会讲到。）把最相似作品的拍卖价格作为基本价格，在拍卖以来的每一年里，最高购买价格是此基本价格之上每年增加 8%。
- 大作。首先，把它当作同一画家的杰作计算最高购买价格，然后如果该画是在 21 世纪画的，将该数字乘以 0.25，否则乘以 $(21-c)/(22-c)$，其中 $c$ 表示该作品是在 $c$ 世纪画的 $(12<c<21)$。
- 其他画作。测量画布的尺寸，最高购买价格由公式 $F\times A$ 给出，其中 $F$ 是该画家的常数（**流行系数**），$A$ 是以平方厘米计算的画布的面积。如果该画家没有流行系数，Osbert 将不会购买该画作。

在杰作和大作的情况下，两幅画作之间的**相似性**系数如下计算：

1）材质上相符则记 1 分，否则为 0。

2）主题上相符则记 1 分，否则为 0。

3）把这两个得数相加，乘以两幅画作中小的作品的面积，再除以两者中大的作品的面积，所得的结果即是相似性系数。

如果被考虑的画作与拍卖数据文件中的所有画作之间的相似性系数为 0，Osbert 将不会购买该大作或杰作。

这个软件产品必须包含一个艺术家列表及他们对应的 $F$ 值，$F$ 值可以根据画家的当前流行情况按月变化，因此必须以允许 Osbert 进行定期更新的方式来实现该列表。Osbert 基于他的常识和经验来确定 $F$ 值，并且如果他在报纸、杂志或互联网上的艺术品销售报告中观察到某个特定艺术家的作品在价格上有涨落，他会调整该系数。

该软件产品必须利用过去 25 年来世界范围的杰作拍卖信息，这个信息包括艺术家的姓名、画作的标题、画作的时间、拍卖的时间、售出价格和作品类型。作品类型包括三个部分：材质（油画、水彩画或其他材质）、尺寸（高度和宽度）及主题（肖像、静物、风景或其他主题）。每个月 Osbert 收到一张 CD，带有更新的世界范围的拍卖价格，Osbert 从来没有调整过这些价格。

需求小组更新用例描述以反映出这个信息，得到的 `Buy a Painting` 用例的描述如图 10-14 所示，`Sell a Painting` 用例的描述如图 10-15 所示。

**简要描述**
Buy a Painting 用例使 Osbert Oglesby 能够购买绘画作品。

**按步骤描述**
1. Osbert 输入他正考虑要购买的画作的细节，包括：
   画家的名
   画家的姓
   作品标题
   作品年份
   分类（杰作、大作、其他）
   高度
   宽度
   主题（肖像、静物、风景、其他）
2. 软件产品回答他应提供的最高购买价格。
   2.1 对于杰作，该软件产品计算出有该画家拍卖记录的每幅画作与正考虑要购买的画作之间的相似性系数，画家的姓名、作品标题或时间中的问号忽略不计。
   如果材质上相符，软件产品则记 1 分，否则记 0 分。
   如果主题上相符，软件产品则记 1 分，否则记 0 分。
   将这两个得数相加，乘以两幅画中较小的那幅画的面积，再除以两幅画中较大的那幅画的面积。得到的结果就是相似性系数。
   软件产品找寻具有最大的非零相似性系数的被拍卖的画作，如果没有这样的画作，Osbert 将不购买这幅考虑中的画作。
   软件产品把最相似作品的拍卖价格按每年增加 8.5%计算（从拍卖那年算起）得出最高购买价格。
   2.2 对于大作，该软件产品首先把画作当作同一画家的杰作计算最高购买价格，然后，如果该画作画于 21 世纪，将得数乘以 0.25，否则乘以 $(21-c)/(22-c)$，其中 $c$ 表示该作品是在 $c$ 世纪画的 $(12<c<21)$。
   2.3 对于其他画作，该软件产品按公式 $F \times A$ 来计算最高购买价格，其中 $F$ 是画家的一个常数（流行系数），$A$ 是画布的面积（以平方厘米为单位）。如果那个画家没有流行系数，Osbert 将不会购买这幅考虑中的画作。
3. 如果 Osbert 购买了该画作，他会输入进一步的细节，包括：
   购买日期
   出售者的姓名
   出售者的地址
   实际购买价格
4. 然后该软件产品记录下列信息：
   算法确定的最高购买价格
   目标售出价格

图 10-14 Osbert Oglesby 实例研究的修订需求的 Buy a Painting 用例描述

**简要描述**
Sell a Painting 用例使 Osbert Oglesby 能够销售绘画作品。

**按步骤描述**
1. Osbert 输入他售出的画作的细节，包括：
   售出日期
   购买者的姓名
   购买者的地址
   实际售出价格

图 10-15 Osbert Oglesby 实例研究的修订需求的 Sell a Painting 用例描述

现在他们转向报表。有三种报表：过去这一年的购买报表、过去这一年的销售报表和新趋势的探测报表。他们再一次查看 Osbert 的手工记录，包括他当前手工生成的报表。他们确定出他的需要如下所述（在所有报表中包含了艺术家姓名、作品标题或日期中的问号）：

需要一个报表来显示过去一年里购买的所有画作，应按下列的顺序输出：

- 分类（杰作、大作、其他画作）
- 购买日期
- 艺术家的姓
- 画作的标题
- 建议的最高购买价格
- 实际的购买价格

所有购买价格超过了经算法计算出的最高购买价格的画作必须进行标记（在分类前放置一个星号）。这个报表必须按分类来排序，在各分类里按购买日期排序。报表中所有画作的实际购买价格与算法建议的最高购买价格之间的比值应在报表结尾显示出来。图 10-16 显示了一个报表样本。




图 10-16　显示过去一年里 Osbert Oglesby 购买的画作的报表样本

第二个报表应显示在过去一年里售出的画作，按下列的顺序输出：

- 分类
- 售出日期
- 画家的姓
- 画作标题
- 目标售出价格
- 实际售出价格

任何以比目标售出价格低 5%或以上的价格售出的画作必须进行标记（在分类前放置一个星号）。这个报表必须按分类来排序，在同一类里按售出日期来排序。报表的最后要显示报表中所有画作的实际售出价格与目标售出价格之间的比值。图 10-17 显示了一个报表样本。

最后，Osbert 希望尽可能地探测出艺术品市场的新趋势，他对确定何时可以给某个特定画家以高于期望值的价格购买画作特别感兴趣，这样他可以在其他人注意到这个趋势前买断那个画家的画作。因此他要求生成一个报表，显示过去一年里每个交易中作品售出价格超过目标售出价格的画家，对于出现在这个报表中的画家，在那段时间里至少由 Osbert 售出两幅他或她的作品。相关画家（如果有的话）的姓名应按字母顺序排列，每个姓名应在屏幕上占一行，然后是那个画家的各种已售出作品，按售出日期排列。对于每幅作品，应包含下列信息：


报表日期：06/14/2004
Osbert Oglesby——油魂艺术收藏者
已售出的画作

分类：大作
姓：Hatsuyar
目标价格：264 450
售出日期：05/11/2004
标题：Table Mountain
售出价格：270 000

分类：大作
姓：Hatsuyar
目标价格：161 250
售出日期：06/12/2004
标题：Walker Bay
售出价格：205 000

分类：*其他
姓：Amman
目标价格：4 300
售出日期：07/13/2004
标题：Apples and Oranges
售出价格：300

平均比率：1.12

图 10-17 显示过去一年里 Osbert Oglesby 售出的画作的报表样本

- 分类
- 画作标题
- 售出日期
- 目标售出价格
- 实际售出价格

图 10-18 显示了一个报表样本。


报表日期：08/14/2004
Osbert Oglesby——油魂艺术收藏者
趋势
画家：David Hatsuyar

分类：大作
售出日期：05/11/2004
目标价格：264 450
标题：Table_Mountain
售出价格：270 000

分类：大作
售出日期：06/12/2004
目标价格：161 250
标题：Walker_Bay
售出价格：205 000

图 10-18 显示艺术品市场新趋势的报表样本

图 10-19 显示了 Produce a Report 用例的一个更新描述和这些细节。

**简要描述**

Produce a Report 用例使 Osbert Oglesby 能够获得过去一年里他买进和售出的画作的信息，或者探测到艺术品市场的新趋势。

**按步骤描述**

1. 下面的报表必须按要求生成（画家姓名、作品标题或日期中的问号必须打印出来）：
1.1 购买报表
系统显示过去一年里购买的所有画作，按下列顺序输出：
分类
购买日期
画家的姓
画作标题
建议的最高购买价格
实际的购买价格

图 10-19 Osbert Oglesby 实例研究的修订需求的 Produce a Report 用例描述

购买价格超过报算法计算的最高购买价格的画作需要做标记（在分类前放置一个星号）。
这个报表要按分类排序，同一分类中按购买日期排序。
报表结尾显示所有画作的实际购买价格与算法建议的最高购买价格之间的平均比值。

1.2 售出报表

系统显示过去一年里售出的所有画作，按下列顺序输出：

分类
售出日期
画家的姓
画作标题
目标售出价格
实际售出价格

售出价格低于目标售出价格5%或以上的画作需要做标记（在分类前放置一个星号）。
这个报表要按分类排序，同一分类中按售出日期排序。
报表结尾显示所有画作的实际售出价格与目标售出价格之间的平均比值。

1.3 未来趋势报表

系统显示过去一年里每个交易中作品售出价格超过目标售出价格的所有画家。
对于出现在这个报表中的画家，在那段时间里至少售出两幅性感裸体的作品。
相关画家（如果有的话）的姓名应按字母顺序排列。
每个姓名应在屏幕上占一行，在这下来的各行里是各种已售出作品，按售出日期排列。
对于每幅作品，以下列顺序输出：

分类
画作标题
售出日期
目标售出价格
实际售出价格

图 10-19（续）

## 10.10　测试阶段：Osbert Oglesby 实例研究

在进行需求阶段的工作期间，对于每个其他的软件开发和维护行为，需求小组的所有成员必须不断地检查他们的工作。现在需求阶段看起来要结束了，需求阶段的成果需要由SQA小组的成员进行检查。换句话说，现在要进行**测试阶段**了。

SQA小组发现了一个严重的疏忽——一个用例被忽视了，10.9节有如下的段落：

*软件产品必须包含一个艺术家列表及他们对应的F值，F值可以根据画家的当前流行情况按月变化，因此必须以允许Osbert进行定期更新的方式来实现该列表。*

换句话说，需求小组成员忘记包含了一个用例来更新流行系数，因此他们加上了图10-20所示的Update a Fashionability Coefficient用例以及它的描述（图10-21）。图10-22显示包含了这个漏掉的用例的用例图的第二个迭代。




图 10-20　Osbert Oglesby实例研究的修订需求的Update a Fashionability Coefficient用例

**简要描述**

Update a Fashionability Coefficient 用例使 Osbert Oglesby 能够修改一个画家的流行系数 F。

**按步骤描述**

1. Osbert 输入该画家新的流行系数。

图 10-21　Osbert Oglesby实例研究的修订需求的Update a Fashionability Coefficient用例描述

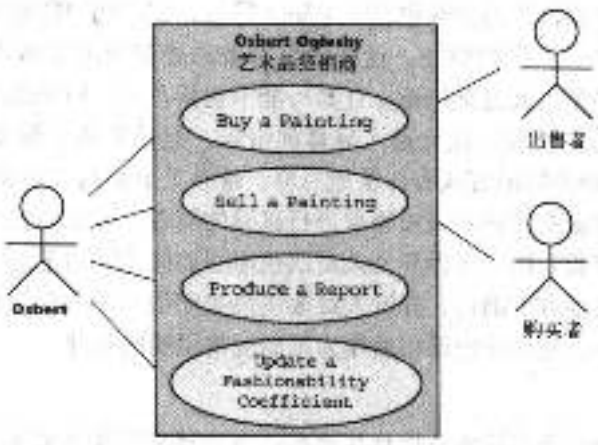



图 10.22 Osbert Oglesby 实例研究的用例图的第二次迭代

经过进一步的检查，好像每件事都让人满意了，至少是现在。然而，在进行面向对象的分析过程中，将会进一步发现错误，需要有额外的需求或现有的需求不再适用。面向对象的范型是迭代和递增的，开发小组的成员必须很清楚，对软件产品的当前版本的变化和扩充会在任何时候发生。

这便结束了 Osbert Oglesby 实例研究的需求阶段的描述。

## 10.11 传统的需求阶段

一方面，没有像"面向对象的需求"这样的事情，也不应该有这样的事情。需求阶段的目标是确定客户的需要，也就是说，目标系统应有什么功能。需求阶段不管产品是如何建造起来的，从这个角度看，在需求阶段无所谓什么传统的范型和面向对象的范型，也无所谓什么传统的或面向对象的用户手册。毕竟，用户手册描述了用户运行软件产品时应该遵循的步骤，也与产品是如何建造的没有关系。同样地，需求阶段的结果是形成产品做什么的声明，产品将如何建造不应包含在其中。

另一方面，10.2 节到 10.10 节的整个方法实质上是面向对象的，即面向模型。用例和它们的描述形成了需求阶段的基础。如本书的第二部分所描述的，建立模型是面向对象范型的根本。

然而，通常的建立模型（和特别的 UML 建模）不是传统范型的一部分。传统的需求阶段开始于需求启发，接下来是类似于面向对象范型的需求分析（10.3 节到 10.4.2 节）。但从那以后，两种范型开始脱离。不同于建立模型，在传统需求阶段的下一步是提出一个需求列表，之后通常是建立一个快速原型，按照那些需求来实现关键的功能，如 2.9.3 节所述。然后客户和目标产品的未来用户在快速原型上实验，直到需求小组的成员满意地认为，快速原型展示了客户需要的软件产品的关键功能。

为产品建立一个快速原型不是面向对象范型的一部分，原因如 12.17 节所述，然而，下面将要讨论到，我们强烈建议需要建立一个用户界面的快速原型。

## 10.12 快速原型开发

**快速原型**是仓促建立的软件，该软件展示了目标软件产品的主要功能。例如，一个帮助

管理公寓全套设备的软件产品必须包含一个输入屏幕，它使用户能够输入一个新房客的详细内容，并每月打印出一个居住报告。这些内容都被结合到快速原型中。然而，差错检查能力、文件更新例行事务以及复杂的税金计算可能不包括在内。问题的关键是，一个快速原型反映的是客户看得到的功能，比如输入屏幕和报告，但是省略了像文件更新这样的“隐藏的”方面。(想以一种不同的方式看待快速原型，请参见下面的“如果你想知道”部分。)

客户和该产品预定的用户对快速原型进行试验的同时，开发小组成员观察并做记录。根据他们丰富的实践经验，用户告诉开发人员快速原型如何能够满足他们的要求，而且更重要的是，指出需改进的地方。随后，开发人员改进快速原型，直至双方确认客户的要求已准确地包含在快速原型中，然后快速原型就作为拟制规格说明的基础。

**如果你想知道**

构建模型来显示一个产品的关键特性的想法可以追溯到很久以前。例如，一幅 1618 年由 Domenico Cresti（又称为“II Passignano”，因为他出生在意大利 Chianti 地区的 Passignano 镇）做的绘画，显示米开朗基罗向保罗四世教皇展示他设计的圣·彼得教堂（在罗马）的木制模型。这样的模型可能很大；建筑师 Bramante 的前期设计建议的模型，其每侧的长度都超过 20 英尺。

建筑模型有许多不同的用途。首先，如 Cresti 的绘画（现悬挂在佛罗伦萨的 Casa Buonarroti 博物馆）所描绘的，模型用于试图在找到一个项目的过程中，激起客户的兴趣。这与使用快速原型来明确客户的真正要求有点相像。第二，在建筑图纸出现之前的年代里，模型向建筑者展示建筑物的结构，并向石匠指示如何装修建筑物。这与我们现在在 10.13 节中描述的建立用户界面的快速原型有些相似。

然而，在这样的建筑模型和软件快速原型之间，做过于密切的并行对比不是太好的主意。快速原型用在传统的需求阶段，启发客户的要求。与建筑模型不同的是，它们不用于展示结构设计和细节设计。生成此设计是在两个阶段之后，即在传统的设计阶段。

快速原型开发模型的一个重要方面包含在“快速”一词中，全部的要旨是尽可能快速地建立原型。说到底，快速原型的目的是向客户提供对软件产品的理解，并且是尽可能快、尽可能好地理解。如果快速原型难以工作，如果它每隔几分钟就崩溃，或者如果屏幕布局不很完美，这些都不要紧。快速原型的意图是使客户和开发者能够在该产品做哪些事情方面尽可能快速地达成一致意见，因此，快速原型中的任何不完美之处都可以忽略不计，前提是它们没有严重损害快速原型的功能并因此误导了对产品的行为的印象。

快速原型开发模型的第二个主要方面是快速原型必须是可修改的。如果快速原型的第一版不是客户所要求的，那么该原型必须迅速改变为第二版，希望这一版能够较好地满足客户的需求。为了在快速原型开发过程中能够较快地进行开发工作，使用了第四代语言（4GL）以及像 Smalltalk、Prolog 和 Lisp 这样的解释性语言。今天流行的快速原型开发语言包括 HTML 和 Perl，还有 Visual C++ 和 Visual J++。人们已经表示了对特定的解释语言的可维护性的关注，但是从快速原型开发的观点来看，这是不相关的。所要考虑的是，给定的一种语言能够用于生成一个快速原型吗？以及快速原型能够迅速地修改吗？如果这两个问题的答案是肯定的，那么该语言对于快速原型开发就可能是一个好的候选者。

当为一个软件产品开发用户界面时，快速原型开发相当有效。在下一节中讨论这种用法。

## 10.13 人的因素

客户和该产品未来的用户与用户界面的快速原型交互是很重要的。鼓励用户试验使用人机界面（human-computer interface，HCI）大大降低了完成的产品将不得不改变的风险。特别是，这项试验有助于获得用户界面的友好性，这是所有软件产品的一个至关重要的目标。

**用户友好**一词指人类与软件产品沟通的容易性。如果用户对学习如何使用一个软件产品感到困难，或者发现屏幕令人困惑或激怒，那么他们将不使用该产品，或者不正确地使用它。为了试图解决这个问题，引入了菜单驱动的软件产品。与需要输入一个像 Perform computation（执行计算）或 Print service rate report（打印服务等级报告）这样的命令不同，用户只要从一系列可能的响应中进行选择即可，比如：

1. Perform computation（执行计算）
2. Print service rate report（打印服务等级报告）
3. Select view to be graphed（选择要图示的场景）

在这个例子中，用户输入 1、2 或 3，调用相应的命令。

现如今，HCI 不再是简单地显示文本行了，而是使用图形。窗口、光标和下拉式菜单是一个**图形用户界面**（Graphical user interface，GUI）的要素。由于存在大量的以窗口方式工作的系统，像 X Window 这样的标准已经大大地发展了。而且，**点击**选择正在成为常用手段，用户移动鼠标（它是一个手握的指向设备），将屏幕光标移到想要的响应（“点”）处，按一下鼠标按钮（“击”）来选择那个响应。

然而，即便是当目标产品采用现代技术的时候，设计者也必须永远不要忘记产品也是由人来使用的。换句话说，HCI 设计者必须考虑**人的因素**，如字母的大小、大小写、颜色、线长以及屏幕上的线数。

再举一个将人的因素应用于前述菜单的例子。如果用户选择选项 3：Select view to be graphed（选择要图示的场景），则出现带有另一个选择列表的另一个菜单。若非菜单驱动系统经过深思熟虑地设计，则可能存在着即便是一个相当简单的操作，用户也要面临着进入一长串菜单的情况。这种拖延会使用户感到气恼，有时会造成他们做出不正确的菜单选择，而且，HCI 必须允许用户不必返回到顶层菜单和重新开始就可以改变先前的某一个选择。这个问题甚至当使用了 GUI 时还会存在，因为许多图形用户界面本质上是以一个交互屏幕的格式显示一系列菜单。

有时，单个用户界面不可能满足所有用户的需要，例如，如果一个软件产品要由专业计算机人士和先前没有任何计算机经验的高中退学生使用，那么，最好设计两套不同的 HCI，每个都经过仔细裁剪，以适应它的预期用户的技能水平和心理状态。这个技术可以通过结合多套要求各种复杂等级的用户界面而加以扩展，如果该软件产品推断用户使用一个不那么复杂的用户界面将会更感方便，可能是因为用户正在不断出错或者正在连续调用帮助工具，那么，随着该用户对该产品的逐渐熟悉，软件将向用户显示提供更少信息的最新屏幕，这样可以使用户快速完成任务。这种自动化的方法减少了用户的困惑，提高了工作效率［Schach and Wood，1986］。

在设计一个 HCI 期间，当考虑了人的因素时自然可以增加许多好处，包括减少学习时间和降低差错率。尽管帮助工具总是要提供的，在一个仔细设计的 HCI 中，它们实际上很

少使用，这同样提高了工作效率。一个产品或一组产品的 HCI 外观的一致性使用户凭直觉就知道如何使用一个他们从未见过的屏幕，因为它与他们熟悉的其他屏幕相似。Macintosh 软件的设计者们已考虑了这个原则，这也是为什么 Macintosh 软件通常非常用户友好的众多原因之一。

有人说要求设计一个用户友好的 HCI 只是简单的常识而已，不管这种说法是否属实，每个软件产品都要建造其 HCI 的快速原型却是必须的。该产品预期的用户可以对 HCI 的快速原型进行试验，告知设计者是否目标产品确实是用户友好的，即是否设计者已经考虑了必要的人的因素。

在下一节中，将围绕快速原型讨论重用。

## 10.14　重用快速原型

建立了快速原型之后，在软件过程的前期就将它丢弃了，一种可选但不明智的处理方式是开发和精炼该快速原型，直到它成为产品。理论上，这种方法应当能加快软件开发过程，因为它毕竟没有抛弃构成快速原型的代码，而是与建立在其中的知识一道，将快速原型转变成最终的产品。然而，实际上这个过程与图 2-7 的编码－修补（code-and-fix）方法非常相似，与编码－修补模型一样，这种形式的快速原型开发模型的第一个问题是，在精炼快速原型的过程中，对一个工作着的产品必须进行修改。采取这种方法是一种代价昂贵的做法，如图1-5 所示。第二个问题是当构建一个快速原型时，由于最初的目标是快速建立，一个快速原型是（正确地）匆匆忙忙凑在一起的，未经过仔细定义、设计和实现。在缺乏规格说明和设计文档的情况下，生成的代码维护困难且花销大。建造一个快速原型然后扔掉它并从头再来设计产品，这看起来可能浪费，但无论是从长期还是从短期来看，这样做都远比试图将快速原型转变成产品级软件造价小得多［Brooks，1975］。

丢弃快速原型的另一个原因是性能问题，对于实时系统尤其是这样。为了确保满足时间限制，有必要认真仔细地设计产品。与此相反，构建快速原型为的是向客户显示关键功能，不处理性能问题。结果是，如果试图把一个快速原型制成一个交付的产品，响应时间和其他时间限制可能无法满足要求。

确保丢弃快速原型并正确地设计和实现一个软件产品的一种方法是，使用不同的语言建造快速原型与建造产品。例如，客户可能指定软件产品必须用 Java 编写，如果快速原型使用 HTML 实现，在快速原型实现后将不得不丢弃它。首先，这个快速原型是用 HTML 实现并完善的，直到客户对它所做的每件事情或几乎每件事情都感到满意，这些事情是目标产品要做到的；其次，设计这个产品依靠了在构建快速原型中所获得的知识和技能；最后，设计是用 Java 实现的，测试过的产品按通常的方式移交给用户。

尽管如此，存在一种情况，在此时允许重用一个快速原型，或特别重用该快速原型的某些部分。当部分快速原型是由计算机生成的时候，那些部分就可以用在最后的产品中。例如，用户界面经常是一个快速原型的一个重要方面，当用像屏幕生成器和报表生成器（见 5.5 节）这样的 CASE 工具生成用户界面时，该快速原型的那些部分确实可以用做产品级软件的一部分。

不“浪费”快速原型的愿望造成了某些组织采用了一个修正版本的快速原型开发模型，其中，开发者在快速原型之前做出管理上的决定，以便可以在最终的产品中利用该部分软

件，假定那部分软件像产品中的其他软件组件一样，通过了相同的质量保证测试。因此，在快速原型完成之后，那些开发者希望继续使用的部分必须通过设计和代码审查。这个方法超出了快速原型开发的范围。举个例子，通常在一个快速原型中找不到足够高质量的可以通过设计和代码审查的软件组件。进一步地，设计文档不是传统快速原型开发的一部分，尽管如此，这种混合的方法对于某些希望收回一些投入在快速原型中的时间和经费的组织来说是有吸引力的。然而，为了确保该代码质量足够高，必须把快速原型建造成在某种程度上比一个通常的“快速”原型慢一些。

## 10.15 需求流的 CASE 工具

本章的许多 UML 图反映出对于需求流起协助作用的图形工具的重要性，也就是说，需要的是一个画图工具，使用户容易地画出相关的 UML 图。这样的工具有两个长处。首先，对存储在此工具中的图进行修改远比手工重画该图容易得多。其次，使用这种 CASE 工具时，产品的细节存储于 CASE 工具本身里，因而，文档总是可用的及更新的。

这样的 CASE 工具的缺点是它们不总是用户友好的。一个强大的图形平台或环境有如此的功能，通常都有一个陡峭的学习曲线，甚至有时有经验的用户也很难记住如何实现一个特定的结果。第二个缺点是要求编程的计算机画出的 UML 图像如手工画出的图那样令人满意几乎不可能。一种选择是花相当的时间来调整由工具生成的图，然而，有时这种方法像手工画图一样慢。更糟糕的是，许多图形 CASE 工具的局限是，不论在一个图上花费多长的时间和努力，它也不可能看上去像手工画出的图一样完美。第三个问题是许多 CASE 工具很昂贵。要求每个用户付出 5000 美元或者更多钱购买复杂的 CASE 工具不太可能。另一方面，一些代码开放的此类 CASE 工具可以免费下载得到。总的来说，本节第一段中提到的 CASE 工具的两个优点可以弥补这些缺点。

许多传统的图形 CASE 工作平台和环境，例如 System Architect 和 Software through Pictures，已经扩充到可以支持 UML 图。另外，还有像 Rose 和 Together 这样的面向对象的 CASE 工作平台和环境，也有这种类型的开放代码的 CASE 工具，包括 ArgoUML。

## 10.16 需求阶段的度量

需求阶段的一个关键特性是需求小组如何能很快确定客户的真正需求。所以此阶段的一个有用的度量是需求变更率的测量。记录下需求阶段中需求变化的频度能够给管理者提供一种方式，来确定需求小组精力集中在产品实际需求上的速率。这个度量还有更进一步的好处，它能应用于任何需求启发技术，例如访谈或形式分析。

另一个测量需求小组工作效率的度量是软件开发过程的其余阶段中更改的需求数量。对于需求中的每一个这样的修改，都应该记录下该修改是由客户提出的还是由开发者提出的。如果在分析、设计和接下来的阶段中由开发者提出修改大量的需求，那么显然需要对小组在需求阶段的工作方式进行全面的复查。相反，如果客户在接下来的阶段对需求进行反复的修改，那这个度量可用来警示客户，变化中的目标问题可能对项目有不利的影响，今后的修改应该控制在最小值。

## 10.17 需求阶段面临的挑战

与软件开发过程的其他阶段一样，需求阶段也有一些潜在的问题和缺陷。首先，非常重

要的是，从过程的开始就要有目标产品的潜在用户的全心全意的合作。一些个人经常感到受到计算机化的威胁，害怕计算机将取代他们的工作。这种害怕有些道理，在过去的 30 多年里，计算机化的影响已经降低了对无技能的工人的需求，但也为有技能的工人创造了一些工作机会。总的来看，作为计算机化的直接结果，创造的收入丰厚的工作机会的数量，远远超过变为多余的相对无技能的工作的数量，这一点从减少的失业率和增加的平均报酬中可以看出。但是，作为所谓的计算机时代的直接或间接的后果，世界范围内众多国家的经济的空前发展，绝不能补偿对那些因为计算机化而失去工作的个人的消极影响。

重要的是，需求分析小组的每个成员随时要意识到，他们很可能与之交互的客户组织的成员，深切地关心着目标软件产品对他们工作的潜在影响。在最坏的情况下，员工可能故意给出误导或错误的信息，试图让产品不满足客户的要求，从而保护那些员工的工作。但是，即使没有这种故意破坏，客户组织的某些成员可能也不会提供多大帮助，这只是因为他们隐隐约约感到计算机化的威胁。

需求阶段的另一个挑战是**协商**的能力。例如，通常重要的是降低客户所想要达到的。几乎每个客户都希望有一个能够做到所能想到的每件事情的软件，这不足为怪。建造这样一个软件时间花费之长会让人无法接受，所需资金也远超过客户的想像。因此，常常有必要说服客户接受比他或她想要的少（有时少得多）的功能。计算争论中的每个需求的成本和收益（见 5.2 节）会有帮助。

另一个需要协商能力的例子是，就目标产品的功能与管理者之间达成一个妥协的能力。例如，一个狡猾的管理者可能试图通过将一个需求纳入以扩展他或她的权力，而这个需求的实现只需要通过将当前由另一个管理者负责的某一业务功能结合进他或她负责的领域。很显然，另一个管理者将对发现的要进行的事情强烈反对。需求小组必须与两个管理者协商，解决这个问题。

需求阶段的第三个挑战是，在许多组织中，拥有需求小组想要启发出的信息的个人没有时间与需求小组会面，进行深入的讨论。当出现这种情况时，小组必须告知客户，客户必须决定哪个更重要，是个人的当前工作职责，还是要建造的软件产品。如果客户未能坚持软件产品优先，开发者可能除了从这个项目中抽身外别无选择，因为这个项目几乎注定失败。

最后，灵活性和客观性对于需求启发是基本的。需求小组的成员不带任何先见参加每次访谈很重要，特别是，一个访谈者绝不要根据先前的访谈结果对需求做假设，然后在那些假设的框架内引导后续的访谈。相反，一个访谈者必须有意识地压制在先前访谈中收集的信息，并且以一种公正的方式引导后面的访谈。做出有关需求的不成熟的假设是危险的，在需求阶段做出任何关于要建造的软件的假设可能会造成惨重的损失。

本章结束于“如何完成”部分，它概括了需求阶段的各步骤。

**如何完成需求阶段的工作**

- 迭代

  获得对应用域的理解。

  提出商业模型。

  提出需求。

- 直到需求让人满意。

## 本章回顾

这一章开始时描述了确定客户需求的重要性（10.1 节），随后概述了需求阶段（10.2 节）。在 10.3 节描述了理解应用域的需要。在 10.4 节中讨论了如何提出商业模型。访谈和其他的需求提取技术在 10.4.1 节和 10.4.2 节中讨论，10.4.3 节介绍了通过用例建立商业模型。10.5 节描述了提出初始的需求，然后提供了 Osbert Oglesby 实例研究。10.6 节中描述了获得对应用域的初始理解；10.7 节和 10.8 节提供了相应的初始商业模型和初始需求，然后在 10.9 节精炼了需求，然后描述了 Osbert Oglesby 实例研究的测试阶段（10.10 节）。在 10.11 节中，对比了传统的需求阶段和统一过程的需求阶段。然后在 10.12 节和 10.13 节详细讨论了快速原型开发；在 10.13 节强调了为用户界面建造快速原型的重要性。在 10.14 节提出了不要重用快速原型的警示。然后讨论了需求阶段的 CASE 工具（10.15 节）和需求阶段的度量（10.16 节）。本章最后描述了需求阶段面临的挑战（10.17 节）。

## 进一步阅读

Jackson［1995］对需求分析做了精彩的介绍。［Thayer and Dorfman，1999］是有关需求分析论文的合集。Berry［2002］提出，对需求不可避免的修改带来的连锁影响是没有软件工程银弹（见 3.15 节中的“如果你想知道”部分）的原因。《IEEE Software》杂志 2003 年 1 月/2 月刊上有关于需求方面的文章。在需求中设置优先权的成本－利润分析的使用在［Karlsson and Ryan，1997］中有描述。

统一过程的需求阶段在［Jacobson，Booch，and Rumbaugh，1999］的第 6、7 章里有详细描述。误用之实例（软件应避免的模型交互用例）在［Alexander，2003］中有描述。

关于快速原型开发的介绍，建议阅读的书有［Connell and Shafer，1989；and Gane，1989］。快速原型开发模型是快速应用开发（rapid application development，RAD）的一个版本，各种有关 RAD 的文章可见《IEEE Software》杂志 1995 年 9 月刊。

人的因素在［Dix，Finlay，Abowd，and Beale，1993］、［Browne，1994］和［Preece，1994］中讨论。用户界面设计方面的经典著作是［Shneiderman，2003］。用户界面方面的文章可在《Communications of the ACM》杂志的 2000 年 9 月刊和 2003 年 3 月刊以及《IEEE Computer》杂志的 2002 年 3 月刊上找到。《Annual Conference on Human Factors in Computer Systems》学报（由 ACM SIGCHI 主办）是有关人的因素的各个方面的有价值的信息源。

## 习题

10.1 对于 Osbert Oglesby 实例研究，初始商业模型的用例图也是初始需求模型的用例图。这总是正确的吗？解释你的答案。

10.2 提出一个非功能性需求，不需要关于目标软件产品的详细信息就可以进行处理。

10.3 提出一个非功能性需求，必须在需求阶段完成之后才能处理。

10.4 区分用例（use case）和用例图（use-case diagram）。

10.5 区分使用者（user）和行动者（actor）。

10.6 画出代表需求阶段的流程图。

10.7 你刚刚作为一个软件经理加入 Victoria & Niagara 公司。Victoria & Niagara 公司多年

来一直为小型商店开发财务软件，它使用的是瀑布模型，常常很成功。根据你的经历，你认为统一过程是一个更先进的软件开发方法。就软件开发给副总裁写一份报告，解释你为什么相信公司应当转到统一过程上来。记住，副总裁不喜欢长度超过半页纸的报告。

10.8　你是 Victoria & Niagara 公司负责软件开发的副总裁，回答习题 10.7 的报告。

10.9　如果没有快速建造一个快速原型，其结果是什么？

10.10　（分析与设计项目）为习题 8.7 的图书自动循环系统完成需求阶段的流程。

10.11　（分析与设计项目）为习题 8.8 的确定银行状态是否正确的产品完成需求阶段的流程。

10.12　（分析与设计项目）为习题 8.9 的自动柜员机完成需求阶段的流程。

10.13　（学期项目）为附录 A 的 Ophelia 的 Oasis 项目完成需求阶段的流程。

10.14　（实例研究）Osbert Oglesby 决定通过附带销售来扩张他的生意，也就是说，他接受把画作悬挂在他的画廊来卖。如果 3 个月之内该画按照 Osbert 和画的主人之前商量好的价格售出了，则 Osbert 和画主平分所得的钱。如果没有售出，把该画返回画主，没有钱的交易。为这个附带销售画用例，并给出你所画的用例的描述，尽可能地提供细节。

10.15　（实例研究）需要生成过去一年里所有附带销售（习题 10.14）的报表，修改图 10-6 和图 10-19，恰当地合并这个额外的报表。

10.16　（实例研究）使用 10.6 节到 10.9 节的信息，为 Osbert Oglesby 实例研究建立一个快速原型。使用你的导师规定的软件及硬件。

10.17　（软件工程读物）你的导师将给你们分发［Alexander，2003］的复印件。你同意误用之实例是重要的吗？

## 参考文献

[Alexander, 2003] I. ALEXANDER, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software* **20** (January/February 2003), pp. 58–66.

[Berry, 2002] D. M. BERRY, "The Inevitable Pain of Software Development: Why There Is No Silver Bullet," Technical Report, University of Waterloo, School of Computer Science, Waterloo, Canada, November 2002.

[Brooks, 1975] F. P. BROOKS, JR., *The Mythical Man-Month: Essays on Software Engineering,* Addison Wesley, Reading, MA, 1975; Twentieth Anniversary Edition, Addison Wesley, Reading, MA, 1995.

[Browne, 1994] D. BROWNE, *STUDIO: STructured User-interface Design for Interaction Optimization,* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Connell and Shafer, 1989] J. L. CONNELL AND L. SHAFER, *Structured Rapid Prototyping: An Evolutionary Approach to Software Development,* Yourdon Press, Englewood Cliffs, NJ, 1989.

[Dix, Finlay, Abowd, and Beale, 1993] A. DIX, J. FINLAY, G. ABOWD, AND R. BEALE, *Human-Computer Interaction,* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Gane, 1989] C. GANE, *Rapid System Development: Using Structured Techniques and Relational Technology,* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Jackson, 1995] M. JACKSON, *Software Requirements and Specifications: A Lexicon of Practice,*

*Principles and Prejudices,* Addison Wesley Longman, Reading, MA, 1995.

[Jacobson, Booch, and Rumbaugh, 1999] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Software Development Process,* Addison Wesley, Reading, MA, 1999.

[Karlsson and Ryan, 1997] J. KARLSSON AND K. RYAN, "A Cost-Value Approach for Prioritizing Requirements," *IEEE Software* **14** (September/October 1997), pp. 67–74.

[Preece, 1994] J. PREECE, *Human-Computer Interaction,* Addison Wesley, Reading, MA, 1994.

[Schach and Wood, 1986] S. R. SCHACH AND P. T. WOOD, "An Almost Path-Free Very High-Level Interactive Data Manipulation Language for a Microcomputer-Based Database System," *Software—Practice and Experience* **16** (March 1986), pp. 243–68.

[Shneiderman, 2003] B. SHNEIDERMAN, *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* 4th ed., Addison Wesley Longman, Reading, MA, 2003.

[Thayer and Dorfman, 1999] R. H. THAYER AND M. DORFMAN, *Software Requirements Engineering,* revised 2nd ed., IEEE Computer Society Press, Los Alamitos, CA, 1999.

# 第 11 章　传统的分析

**学习目标**

学完本章之后，你应当能够：

- 完成结构化的系统分析；
- 使用有穷状态机（Petri 网和 Z）提出形式化的规格说明；
- 比较和对照传统的分析方法。

一个规格说明文档必须满足两个相互矛盾的需求。一方面，这个文档对于客户必须是清晰和可理解的，因为客户很可能不是一个计算机专家。毕竟，客户在为产品付钱，并且除非客户相信他或她真正理解了新产品将是什么样的，否则客户很可能决定不批准开发这个产品，也可能请其他软件公司建造它。

另一方面，规格说明文档必须完整而详细，因为这实际上是开展设计可得到的惟一的信息来源。即使客户承认需求阶段所有的要求都已经明确了，如果规格说明文档包含一些错误，如遗漏、矛盾或模糊，不可避免的结果是，设计中的错误将被带到实现中去。因此，我们需要一种以某种格式描述目标产品的技术，它既要是相当非技术性的，能够为客户所理解；又要足够准确，使得在开发周期结束时交付客户的产品是无错误的。这些分析（规格说明）技术是本章和下一章的主题。本章的重点是传统的（结构化的）分析技术，而第 12 章讨论面向对象分析。

## 11.1　规格说明文档

**规格说明文档**是客户和开发者之间的一种合同。它明确规定了产品必须做什么，以及对产品的**约束**。实质上，每个规格说明文档都包含产品必须满足的一些约束，并且几乎都规定了一个交付产品的最后期限。另一个常见的约定是，“这个产品应当以这样一种方式安装，以使它能够与现有的产品并行运行”，直到客户满意地认为新产品确实满足了规格说明文档的每个方面。其他约束可能包括可移植性：建造的产品能够在安装同一操作系统的其他硬件上运行，或者可能在各种不同的操作系统下运行。可靠性可能是另一个约束。如果某一产品要监视处在特别护理单元中的病人，那么它一天 24 小时不间断工作是至关重要的。快速响应时间可能是一个需求，这类约束中的一个典型情况可能是，“95% 的所有类型 4 的查询应在 0.25 秒内做出回答”。许多响应时间约束不得不用概率术语来表达，因为响应时间依赖于当前计算机的装入——把（数据）从存储装置调入计算机存储器（译者注）。相反，所谓的严格实时约束是用绝对术语来表示的。例如，开发一个软件，仅有 95% 的次数在导弹来袭的 0.25 秒内告知战机飞行员，这是毫无用途的，该产品必须 100% 地满足这个约束。

规格说明文档的一个至关重要的组成部分是验收标准集。从客户和开发者两方面的观点来看，重要的是清楚地给出一系列测试，可以用它向客户表明产品确实满足了它的规格说明，并且开发者的工作完成了。某些验收标准可能是约束的复述，而其他则提出不同的问题。例如，客户可能向开发者提供产品将要处理的数据的描述，那么，一个合适的验收标准将是，产品正确地处理这种类型的数据，并滤出不符合（即不正确）的数据。一旦开发小组

对问题完全理解了，就可以提出可能的解决策略。**解决策略**（solution strategy）是建造产品的一个大致的方法。例如，一个产品一种可能的解决策略是使用联机数据库；另一种可能的解决策略是直截了当地使用常规的文件，并用长时间的批运行提取出需要的信息。当确定解决策略时，在不考虑规格说明文档中的约束的情况下提出策略是一个好办法，然后就可以按照约束评估各种解决策略，并做必要的修改。有一些方法可以确定某个解决策略是否能满足客户的约束，一个明显的办法是原型开发，它是解决与用户界面和时间限制有关的问题的一项好技术，这已在第 10 章中讨论过了。其他确定是否将满足约束的技术包括仿真［Banks, Carson, Nelson, and Nichol, 2001］和解析网络模型［Kleinrock and Gail, 1996］。

在这个过程中，会提出一些解决策略然后又将它丢弃。保存所有丢弃的策略以及拒绝它们的原因的记录很重要，如果要求证明选择的策略是正确的，这将会帮助开发小组。但是，更重要的是，在交付后维护阶段经常存在着一种危险，在增进维护过程中经常伴随着提出新的不明智的策略的尝试，在交付后维护期间保留一个为什么在开发期间某些策略被拒绝的记录非常有帮助。

按照生命周期里的这个观点，开发小组将要确定满足约束的一个或多个可能的解决策略，现在需要分两步做出决定。首先，是否应建议客户进行计算机化的工作，如果是这样，他会采纳哪个可行的解决策略。第一个问题的答案完全可以在成本－效益分析（见 5.2 节）的基础上决定；其次，如果客户决定进行这个项目，那么客户必须告诉开发小组要使用的最优化准则，如使客户的成本最低或使投资的回报最大。开发者然后向客户建议最符合优化准则的可行的解决策略。

## 11.2　非形式化规格说明

在许多开发项目中，规格说明文档由一页页的英文或一些其他的**自然语言**如法语或科萨语组成。一段典型的**非形式化规格说明**文档如下：

BV.4.2.5. 如果当月销售额低于目标销售额，那么打印一份报表，除非目标销售额和实际销售额之差低于上月目标销售额和实际销售额之差的一半，或者除非当月目标销售额和实际销售额之差低于 5%。

该段规格说明的背景是这样的：零售连锁业的管理者每月为每个商店设定了一个目标销售额，而如果一个商店没有完成这个目标，就打印一份报表。考虑下面的情况：假定某一商店 1 月份的目标销售额是 10 万美元，但实际销售额仅为 6.4 万美元，即低于目标销售额 36%。在这种情况下，必须打印报表。现在进一步假定 2 月份的目标销售额是 12 万美元，而实际销售额仅为 10 万美元，低于目标 16.7%。尽管销售额低于目标数字，但对于 2 月份的百分比差 16.7%，低于上月百分比差（36%）的一半，管理者认为有进步，于是不用打印报表了。接下来假定 3 月份目标又是 10 万美元，但是，该商店完成 9.8 万美元，仅比目标低 2%。因为百分比差较小（低于 5%），不应当打印报表。

仔细重新阅读前段规格说明，可发现与零售连锁业的管理者实际要求的有一些分歧。B.V.4.2.5 段谈到“目标销售额和实际销售额之差”，没有提到百分比之差。1 月份的差额是 3.6 万美元，而 2 月份的差额是 2 万美元。管理者想要的百分比之差，从 1 月份的 36% 降至 2 月份的 16.7%，少于 1 月份百分比之差的一半。然而，实际差额从 3.6 万美元降至 2 万美元，而 2 万美元大于 3.6 万美元的一半。因此，如果开发小组忠实地实现规格说明文档，

将打印报表，但它却不是管理者想要的。那么最后一个条款说到“……之差低于5%”，当然它指的是5%的百分比差额，只是“百分比”一词没有在段落中出现罢了。

因此，该规格说明文档包含一些错误。首先，忽视了客户的愿望；其次，存在模糊性——最后一个条款说的是“差……5%”，还是“差额……5千美元”，或是其他什么？此外，风格也很差。该段中说“如果发生什么，则打印报表。然而，如果发生其他的某些事，就不打印报表。如果第三种情况发生，也不打印报表。”如果规格说明文档只简单地声明什么时候打印报表，应当会更清楚。总而言之，B.V.4.2.5段不是一个如何写规格说明文档的非常好的例子。

B.V.4.2.5段是假想的，但遗憾的是，它是许多规格说明文档的典型情况。你可能认为这个例子有些不公平，而如果规格说明文档由专业的规格说明文档编写者仔细来写，这类问题就不会发生。为了反驳这个观点，把第6章的小型实例研究在这里重述一下。

**正确性证明（再论）——小型实例研究**

我们还记得在6.5.2节中提到，在1969年Naur写了一篇有关正确性证明的论文[Naur，1969]。他通过文本处理问题阐述了他的技术。使用他的技术，Naur构建了一个ALGOL 60程序来解决问题，并且非形式化地证明了他的程序的正确性。Naur的论文的一个评论者[Leavenworth，1970]指出他的程序中的一个错误，London[1971]然后检测出Naur程序中的另外三个错误，给出了该程序的修正版本，并且形式化证明了它的正确性。Goodenough和Gerhart[1975]后来又发现了London没有检测出的三个错误。在由评论者Leavenworth、London、Goodenough和Gerhart共同纠正的7个错误中，有两个是分析错误。例如，Naur的规格说明没有声明如果输入包括两个连续、相邻的分隔符（空格或换行字符）会发生什么。为此，Goodenough和Gerhart提出一套新的规格说明，他们的规格说明比Naur的长四倍，参见6.5.2节。

1985年，Meyer写了一篇有关形式化规格说明技术的文章[Meyer，1985]。他的文章的要点是，用像英语这样的自然语言写的规格说明有出现矛盾、模糊和遗漏的倾向，他建议使用数学术语来形式化地表达规格说明。Meyer检测出Goodenough和Gerhart的规格说明中的12个错误，并编写了一套数学规格说明来纠正全部的错误。Meyer然后将他的数学规格说明意译并形成英语的规格说明。在我看来，Meyer的英语规格说明包含一个错误。Meyer在他的论文中指出，如果每行的最大字符数比如说是10，而输入比如说是“WHO WHAT WHEN”，那么，根据Naur的规格说明以及Goodenough和Gerhart的规格说明，有两个同样有效的输出：第一行上的WHO WHAT和第二行上的WHEN，或者第一行上的WHO和第二行上的WHAT WHEN。事实上，Meyer的意译的规格说明也包含这种模糊性。

关键是Goodenough和Gerhart的规格说明是非常认真地编写的。毕竟，编写它们是为了纠正Naur的规格说明。而且，Goodenough和Gerhart的论文有两个版本，第一个版本发表在一个权威会议的会议录上，第二个版本发表在一本权威杂志上[Goodenough and Gerhart，1975]。最后，Goodenough和Gerhart都是软件工程方面的专家，特别是规格说明方面的专家。因此，如果两个专家用这样多的时间认真编写的规格说明中，都包含有Meyer检出的12个错误，那么，想要一个普通的计算机专业人员在时间压力之下编写一个无错误的规格说明，这样可能吗？更糟糕的是，文本处理问题可以仅用25或30行代码写成，而现

实世界的产品可能由成千上万行甚至百万行源代码组成。

显然，自然语言不是一个规定产品的好方法。本章介绍了其他一些较好的替代方法，介绍的顺序是从非形式化技术到形式化技术。

## 11.3　结构化系统分析

将图形应用于软件的规格说明是 20 世纪 70 年代的一项重要技术。有三种使用图形的技术非常流行，分别是：DeMarco 的方法［1978］、Gane 和 Sarsen 的方法［1979］、以及 Yourdon 和 Constantine 的方法［1979］。这三种技术都很好并且基本上类同，这里给出 Gane 和 Sarsen 的方法，因为他们的符号表示是时下业界广泛使用的。

为便于理解这项技术，考虑下面的小型实例研究。

### Sally 的软件商店——小型实例研究

Sally 的软件商店从各供应商处买来软件，然后将它卖给大众。Sally 采购流行软件包，需要的话也订购其他的。Sally 给研究所、公司和一些个人提供信用贷款，她的软件商店办得相当好，以平均每个 250 美元的零售价每月周转 300 套软件包。尽管她的生意很成功，但有人建议她计算机化，她将如何做呢？

这样提出问题是不合适的。应当这样陈述问题：生意的哪部分（如果需要的话）——应付账款、应收账款和库存——应当计算机化？甚至这还不够，系统是批处理的还是联机的？使用的是内部的计算机还是需外购计算机？但是，即使进一步细化问题，它还是遗漏了根本的问题：Sally 将其生意计算机化的目的是什么？

仅当知道了 Sally 的目标后，才能继续分析下去。例如，如果她想计算机化仅仅是为了卖软件，那么她需要一个带有各种声、光效果的内部系统，以显示一个计算机的潜力。另一方面，如果她用她的生意洗“赃手的”钱，那么她需要一个产品，保留 4～5 套不同的账本，不给查账留下痕迹。

这个例子假定 Sally 想计算机化以“赚取更多的钱”。这没有多大帮助，但很清楚的是，成本－效益分析法可以确定是否将她生意的三部分中的每个或任一个计算机化。

许多标准方法的主要危险在于，它诱惑人们首先提出解决办法，例如，一台带有 50G 硬盘的 Lime Ⅲ计算机以及一台激光打印机，后来再查找问题是什么。与此相反，Gane 和 Sarsen［1979］使用**结构化系统分析**——一个 9 步骤的技术，分析客户的要求。重要的一点是，在这 9 个步骤中多次用到逐步求精，在展示该技术时将说明这点。

在确定了 Sally 的需求之后，结构化系统分析的第一步是确定**逻辑数据流**（logical data flow），与它相对立的是物理数据流（即，发生了“什么”与“如何”发生相对立）。这通过画**数据流图**（data flow diagram，DFD）完成。DFD 使用如图 11-1 所示的 4 种基本符号。（这些符号与 Gane 和 Sarsen 的相似，但与 DeMarco［1978］的以

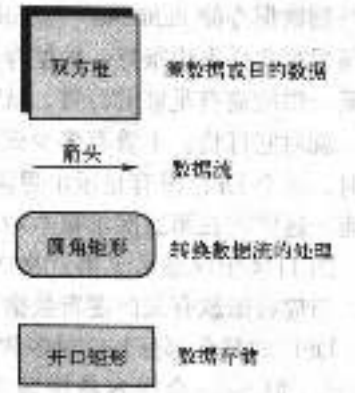



图 11-1　Gane 和 Sarsen 的结构化系统分析的符号

及 Yourdon 和 Constantine［1979］的不同）。

**步骤 1　画 DFD**

重要产品的 DFD 可能很大，DFD 是逻辑数据流各个方面的图形化表示，这样保证它包含多于 7±2 个要素，因此，DFD 必须由逐步求精法得出（见 5.1 节）。

数据流图通过识别需求文档或快速原型内的**数据流**来建造。每个单独的数据流，或者开始和结束于**源数据或目的数据**（用双方框表示），或者开始和结束于**数据存储**（用开口矩形表示）。数据由一个或多个**处理**（用圆角矩形表示）进行转换。在每个后续的求精中，或者将一个新的数据流加到 DFD，或者通过增加进一步的细节，求精现有的数据流。

回到例子中来，第一次求精如图 11-2 所示。这个逻辑数据流图可以有许多解释，下面是两个可能的实现。

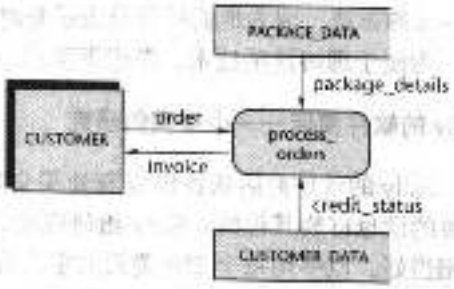



图 11-2　Sally 的软件商店的数据流图：第一次求精

在实现 1 中，数据存储 PACKAGE _ DATA 由大约 900 个紧缩包装盒组成，内有货架上展示的磁盘或 CD，还有一些在桌子抽屉里的目录册。数据存储 CUSTOMER _ DATA 是一摞 5×7 的卡片，由一根橡皮带捆在一起，加上一个到期未付款的顾客的清单。处理（行为）process _ orders 是 Sally 在货架上找适当的软件包，如果有必要，在目录册中查找，然后找到正确的 5×7 卡片，并且检查那个顾客的名字是否在拖欠货款者名单上。这个实现完全是人工的，符合 Sally 现在做生意的方式。

在实现 2 中，数据存储 PACKAGE _ DATA 和 CUSTOMER _ DATA 是计算机文件，process _ orders 是 Sally 在一个终端上输入顾客的名字和软件包的名字。这个实现相当于一个完全计算机化的解决方法，全部的信息都可联机得到。

图 11-2 的 DFD 不仅表现了前述两个实现，还描述了其他无穷的可能性。关键是 DFD 表示了一个信息流——Sally 的顾客想要的实际的软件包对这个信息流并不重要。

现在对 DFD 逐步求精。图 11-3 描述了第二次求精。在顾客请求一个软件包而 Sally 手边没有的时候，代表所发生事情的逻辑数据流被加到 DFD 中。特别是，将那个软件包的细节放到数据存储 PENDING _ ORDERS 中，它可能是一个计算机文件，但在这个阶段如果它是一个马尼拉文件夹也很好。数据存储 PENDING _ ORDERS 每天由计算机或由 Sally 检查，而如果对某一供应商有足够的订货，就安排一次批量订货。还有，如果一个订单已经等了 5 个工作日，就对它订货，不管有多少软件包正等着从相应的供应商处订货。当软件包从供应商处到达时，这个 DFD 没有显示出逻辑数据流，它也没有显示如应付账款和应收账款这样的财务功能。这些将在第三次求精中加入。

图 11-4 中仅显示了第三次求精的一部分，因为 DFD 开始变得大起来了。在这次求精中，与应收账款有关的逻辑数据流加到了 DFD 中。

DFD 的其余部分与应付账款和软件供应商有关。最后的 DFD 仍将很大，展开了可能有 6 页纸。但 Sally 会很容易理解它，她会签署它，确认它是她的生意中逻辑数据流的精确表示。

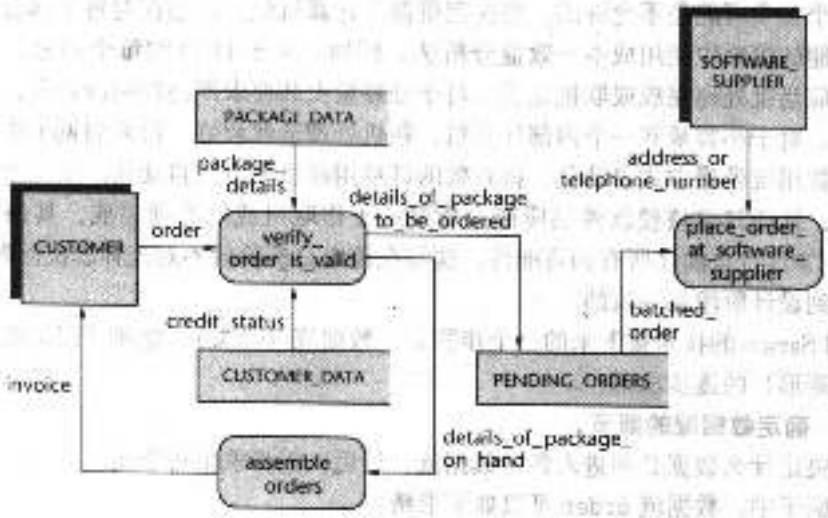



图 11-3 Sally 的软件商店的数据流图：第一次求精

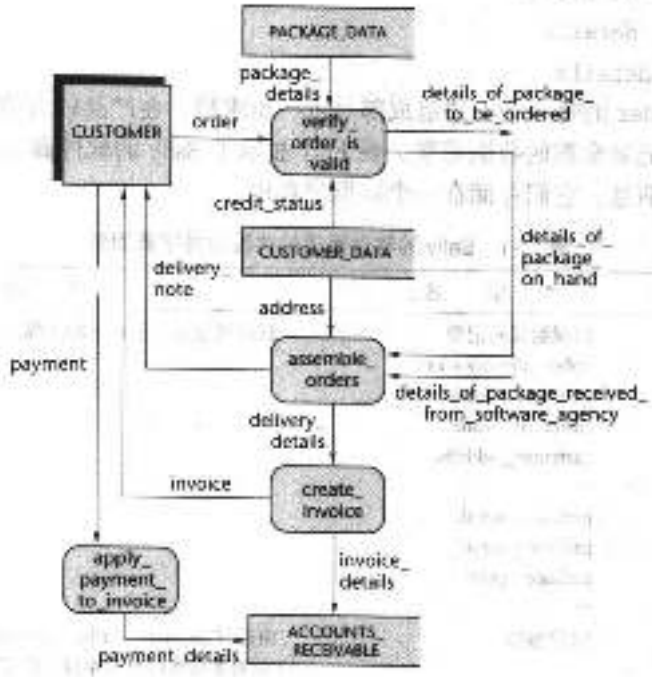



图 11-4 Sally 的软件商店的数据流图：第三次求精的一部分

对于一个较大的产品，DFD 将很大。在某一点后，只有一个 DFD 变得不切实际了，需要一组分等级的 DFD。某一个级别上的单个的方框，在较低级别上扩展为一个完整的 DFD。

在这一节里，我们为 Sally 的软件商店概述了 DFD 的结构。11.4 节中给出了一个数据流图结构的更详细的例子。

**步骤 2 决定哪部分计算机化以及如何计算机化（批处理或联机）**

选择对什么进行自动化常常取决于客户准备花多少钱。显然，最好将整个运转都自动

化，但是这个花费可能是不允许的。要决定哪部分计算机化，应当在对每个部分进行计算机化的各种可能的策略中使用成本－效益分析法。例如，对于 DFD 的每个部分，需要决定哪组操作是否应当批处理完成或联机完成。对于处理量大和要求严格控制的情况，经常选择批处理；但是，对于小容量和一个内部计算机，联机处理显然较好。再回到例子中，一个选择是将应付账款用批处理方式自动化，将有效的订单用联机的方式自动化。第二个选择是自动化每件事情，让凭订单核校软件供应商发货票的工作联机或批处理完成，其余操作联机完成。关键是 DFD 符合前述所有的可能性。这与在传统分析阶段不对怎样解决问题做出承诺，而是一直等到设计阶段是一致的。

Gane 和 Sarsen 的技术接下来的三个步骤是：数据流（箭头）、处理（圆角矩形）和数据存储（开口矩形）的逐步求精。

**步骤 3　确定数据流的细节**

首先，决定什么数据必须进入各种数据流，然后，逐步求精每个流。

在这个例子中，数据流 order 可以如下求精：

```
order:
    order _ identification
    customer _ details
    package _ details
```

接下来，对 order 的每一个上述组成部分进一步求精，在产品较大的情况下，可用数据字典（见 5.5 节）记录全部的数据元素。表 11-1 显示了 Sally 的软件商店的计算机化中有关的数据元素的典型信息，它们存储在一个数据字典中。

**表 11-1　Sally 的软件商店的典型数据字典词条**

| 数据元素名 | 描　述 | 叙　述 |
|---|---|---|
| order | 由域组成的记录<br>order _ identification<br>customer _ details<br>customer _ name<br>customer _ address<br>…<br>package _ details<br>package _ name<br>package _ price<br>… | 这些域包含一个订单的全部细节 |
| order _ identification | 12 位整数 | 由过程 generate _ order _ number 生成的惟一编号，前 10 位数字包含订单号本身，后 2 位是校验位 |
| verify _ order _ is _ valid | 过程：<br>输入参数：<br>order<br>输出参数：<br>number _ of _ errors | 这个处理把 order 作为输入并检查每个域的有效性，对于发现的每个错误，在屏幕上显示相应的消息（发现的总的错误数在参数 number _ of _ errors 中返回） |

**步骤 4　定义处理的逻辑**

既然已经确定了产品中的数据元素，现在可以研究在每个处理中发生了什么。假定这个例子有一个处理 give _ educational _ discount。Sally 必须向软件开发者提供有关她给教育

机构打折的细节信息，例如，4 个软件包以内折扣 10%，5 个软件包以上折扣 15%。要解决自然语言规格说明文档的难题，应当把这些从英语翻译成一个判决树，这样的一个树如图 11-5 所示。

一个判决树便于检查考虑的所有可能性，特别是在较复杂的情形下。图 11-6 给出了一个例子，从图 11-6 中可以很明显地看出，校友在端区的座位价格没有指定。

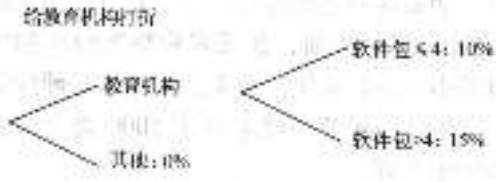



图 11-5　描述 Sally 的软件商店的教育折扣策略的判决树

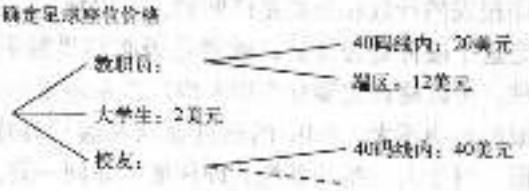



图 11-6　描述大学足球赛座位价格的判决树

**步骤 5　定义数据存储**

在这个阶段，有必要定义每个数据存储和它的表示（格式）的准确内容，因此这个产品如果要用 COBOL 实现，这个信息必须向下提供到 `pic` 级，如果要使用 Ada，必须定义 `digits` 或 `delta`。此外，还必须指定哪里要求实时访问。

实时访问的问题取决于对产品进行什么查询。例如，在这个例子中假设，决定采用联机方式验证订单，一个顾客可能按名称订购了一个软件包（“你有 Lotus 1-2-3 的现货吗?”），也可能按功能订购（“你有什么财务软件?”），或者按机器订购（“你有 786 机上用的什么软件吗?”），但是很少有按价格订购的（“你有售价 149.50 美元的什么东西吗?”）。因此，实时访问 PACKAGE_DATA 要通过名称、功能和机器类型进行，在图 11-7 的**数据实时访问图**（DLAD）中描述了这一点。

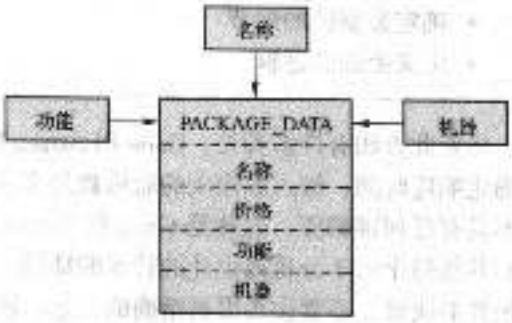



图 11-7　PACKAGE_DATA 的数据实时访问图

**步骤 6　定义物理资源**

既然开发者知道联机所想要的以及每个元素的表示（格式），就可以对组合因子做决定了。另外，对于每个文件，可以指定文件名、组织结构（排序、索引等）、存储介质以及向下到域一级的记录。如果要使用一个数据库管理系统（database management system，DBMS），那么要在这里指定每个表的相关信息。

**步骤 7　确定输入-输出规格说明**

必须指定输入格式，如果没有详细规划，最起码应明确有哪些组成部分。必须近似地确定输入屏幕，还必须规定要打印的输出格式，如果可能就详细些，否则至少要估计输出的长度。

**步骤 8　确定大小**

有必要计算数值数据，将在步骤 9 中用它来确定硬件要求。这包括输入量（每天或每小

时)、每个要打印报表的频率和最后期限、在 CPU 和大容量存储器间传递的每种类型的记录的大小和数量、每个文件的大小。

**步骤 9　确定硬件要求**

从步骤 8 中确定的磁盘文件大小的信息，可以计算出大容量存储器的要求。另外，可以确定用于备份的大容量存储器的要求。从输入容量的信息，可以发现这方面的要求。因为打印报表的行数和频率是已知的，可以指定输出设备。如果客户已经有硬件了，它可以用来确定这个硬件是否合适，或者是否必须得到附加的硬件。另一方面，如果客户缺乏合适的硬件，可以建议需要什么以及应当购买还是租用。对于小一些的系统，技术上的进步对硬件决策的影响不大，Sally 的软件商店在编写程序时，所需的全部硬件成本低于 1000 美元，然而，对于大一些的系统，硬件成本非同一般，需要仔细决策。

确定硬件需求是 Gane 和 Sarsen 的分析技术的最后一个步骤，经客户批准后，得到的规格说明文档交给设计小组，软件过程继续。

下面的“如何进行”部分总结了 Gane 和 Sarsen 的结构化系统分析的 9 个步骤。

**如何进行结构化系统分析**

- 画数据流图
- 决定哪部分计算机化以及如何计算机化（批处理或联机处理）
- 确定数据流的细节
- 定义处理的逻辑
- 定义数据存储
- 定义物理资源
- 确定输入－输出规格说明
- 确定大小
- 确定硬件要求

尽管此方法有许多长处，Gane 和 Sarsen 的技术并不能解决所有问题。例如，它不能用于确定响应时间；输入和输出的通道数最多只能粗略地测量；而且，估计 CPU 大小和定时也不具有任何准确度。这些是 Gane 和 Sarsen 的技术的明显缺陷，而且公平地说，这实际上也是其他每个用于分析或设计的技术的缺陷。尽管如此，在传统的分析阶段的最后，必须做出硬件的决策，不管能否得到精确的信息。这个情形比过去所做的要好得多了，在提出用系统的方法制定规格说明之前，有关硬件的决策在软件开发过程的开始就做出了。Gane 和 Sarsen 的技术在制定产品的规格说明的方法上有重大改进，虽然 Gane 和 Sarsen 以及其他最有竞争的技术的创立者们基本上忽略了时间作为一个变量的事实，但这并不影响这些技术曾给软件工业带来的好处。

## 11.4　结构化系统分析：Osbert Oglesby 实例研究

数据流图如图 11-8 所示。Osbert 初始化了三个处理：buy、sell 和 print_report。因此，三个处理矩形连接到 OSBERT_OGLESBY 数据源或目标。

首先考虑 buy 处理。一幅画作被分类为杰作/大作（masterwork）或其他画作。

- 如果该画作是杰作/大作，那么软件产品需要访问 AUCTION_PAINTING 存储来确定 Osbert 应开什么价。图 10-14 描述了这个计算过程，更详细地，buy 处理需要从 Osbert 那里获得 buy_painting_information 信息。根据 Osbert 提供的数据，提取出 artist_name (first_name_of_artist, last_name_of_artist) 传递给 AUC-

TION _ PAINTING 数据存储，返回 auction _ data 给 buy 处理，按名称讲，是那个画家的所有被拍卖的画作的 auction _ price（拍卖价格）、auction _ date（拍卖日期）、medium（介质）、subject（主题）、height（高度）和 width（宽度）。然后 buy 处理使用这些信息来计算该画家所有拍卖画作的相似系数，因此 Osbert 应提供 maximum _ purchase _ price（最高购买价格），这个值被返回给 Osbert。

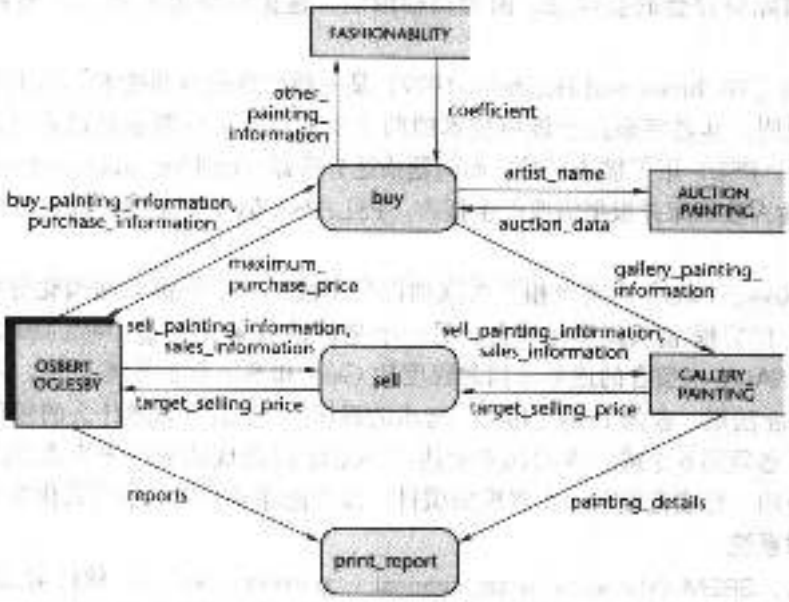



图 11.8　Osbert Oglesby 实例研究的数据流图

- 其他画作的价格取决于画家的流行系数。在这个例子中，other _ painting _ information 被传递给 FASHIONABILITY 存储，包括 first _ name _ of _ artist、last _ name _ of _ artist、height 和 width（流行系数的计算细节参见图 10-14）。
- 最后，不考虑画作的分类，如果 Osbert 的开价被接受，必须由 Osbert 输入 purchase _ information，并在 GALLERY _ PAINTING 数据存储中记录 gallery _ painting _ information 信息。

其次，如果 Osbert 希望售出一幅画作，他首先需要知道应为该画作标注什么价格，也就是说，需要由 sell 处理转发 sell _ painting _ information 给 GALLERY _ PAINTING，返回 target _ selling _ price。然后，一旦决定进行交易，使用 sales _ information 来更新 GALLERY _ PAINTING 数据存储。

第三，如果 Osbert 希望打印一个报表，print _ report 处理将需要报表的细节，用来从 GALLERY _ PAINTING 数据存储中获得必需的 painting _ details 信息。

Osbert Oglesby 实例研究的结构化系统分析的其余部分在附录 D 中叙述。

## 11.5　其他半形式化技术

Gane 和 Sarsen 的技术显然比用自然语言编写的规格说明文档形式化多了。与此同时，它又不如后面将讨论的许多技术那么形式化，如 Petri 网（11.8 节）和 Z（11.9 节）。Dart

和她的同事们将分析和设计技术划分为非形式化、半形式化和形式化三类［Dart，Ellison，Feiler，and Habermann，1987］。根据这个分类，Gane 和 Sarsen 的结构化系统分析属于**半形式化规格说明**技术，而本段中提到的另外两种技术是形式化规格说明技术。

结构化系统分析的应用很广泛，你很有可能受雇于一个使用结构化系统分析或它的某种变种的公司。然而，还有许多其他好的半形式化的规格说明技术，比如可以看各种软件规格说明和设计国际研讨会的会议录。因为篇幅限制，这里将给出少数几个著名技术的简要描述。

PSL/PSA［Teichroew and Hershey，1977］是一种计算机辅助技术，用于对信息处理产品进行规格说明。其名字来源于该项技术的两个组成部分：问题描述语言（problem statement language，PSL）用于描述产品，而问题描述分析器（problem statement analyzer，PSA）将 PSL 描述输入数据库并根据需要产生报告。PSL/PSA 仍在被应用，特别是用于文档处理产品。

SADT［Ross，1985］由两个相互关联的部分组成，一个是称为**结构化分析**（structural analysis，SA）的方框-箭头图形化语言，另一个是设计技术（design technique，DT），因此称为 SADT。SADT 中蕴含的逐步求精的程度比 Gane 和 Sarsen 的技术更深，人们有意识地努力拥护 Miller 法则。就像 Ross［1985］指出的那样，“对值得表述什么的任何事情所表述的每件事情，必须用 6 个或更少的词来表达。”SADT 已经成功地用于为范围广泛的多种产品规定规格说明，特别是复杂、大规模的项目。像其他许多类似的半形式化技术一样，它不太适用于实时系统。

另一方面，SREM（the software requirement engineering method，软件需求工程化方法）是专为规定一些条件而设计的，在这些条件下将发生某些动作［Alford，1985］。因为这个原因，SREM 对于实时系统的规格说明特别有用，也扩展到分布式系统。SREM 有一些组成部分：RSL 是一个规格说明语言；REVS 是一套工具集，完成各种与规格说明有关的任务，如把 RSL 规格说明翻译成一个自动的数据库，自动地检查数据流的一致性（确保一个数据项在赋值之前不被使用），并且从规格说明中产生仿真器，它可以用来确信规格说明是正确的。此外，SREM 有一项设计技术——DCDS（distributed computing design system，分布式计算设计系统）。

SREM 的强大功能来自于整个技术背后蕴含的模型，它是一个有穷状态机（finite state machine，FSM，见 11.7 节）。作为 SREM 中蕴含的这个形式化模型的结果，它可能完成前面提到的一致性检查，可用于验证在给出单个组件的性能的情况下，对产品整体上的性能约束。SREM 已被美国空军用于规定两个 $C^3I$ 软件（命令、控制、通信和智能，command，control，communication，and intelligence）系统［Scheffer，Stone，and Rzepka，1985］。尽管 SREM 被证明在规格说明阶段非常有用，但看起来人们认为 REVS 工具在开发周期的后期用处不大。

## 11.6　建造实体－关系模型

结构化系统分析的重点在于要建造的产品的行为而非数据。当然，也建造产品的数据模型，但数据较之行为是第二位的。相反，建造实体－关系模型（entity-relationship modeling，ERM）是制定产品规格说明的一项半形式化的面向数据技术，它 25 年来广泛用于制定数据

库规格说明。在那个应用领域，重点在于数据。当然，需要通过行为来访问数据，数据库的组织方式必须能够使访问时间最短。尽管如此，在数据上进行的行为仍是次要的。

一个简单的实体－关系图如图 11-9 所示，它为作者、小说和读者之间的关系建立模型。这里有三个实体：作者（Author）、小说（Novel）和读者（Reader）。顶层关系“写”反映了一个作者写一本小说，这是一个一对多的关系，因为一个作者能够写一本以上的小说，这通过“作者”后面的 1 以及“小说”后面的 n 反映出来。这个实体－关系图也显示了“小说”和“读者”之间的两个关系，它们两个都是一对多的关系，左侧的关系为一个读者可能读许多本小说的事实建模；同样，如右侧所示，一个读者可能拥有许多本小说。显示了两个独立的关系是因为，一个读者可能读了一本小说但不拥有它，一个读者也可能买了一本小说但却没有读它。

下一个例子取自供应商和他们提供的组件这一领域。图 11-10 显示了组件和供应商之间的多对多关系，即，一个供应商供应多个组件，反过来，某一组件可以从多个供应商处得到。这个多对多关系通过“供应商”实体后的 m 以及“组件”实体后的 n 表示。

也可以表示更复杂的关系。例如，如图 11-11 所示，可以将一个组件看作是由一些零部件组成。而且，多对多对多的关系是可能的。考虑该图中显示的三个实体“供应商”、“组件”和“项目”，某一组件可以由几个供应商提供，这要依项目而定；而且，为某一项目提供的各种组件也可以来自不同的供应商。要准确对这样的情形建立模型，一个多对多对多关系是必要的。

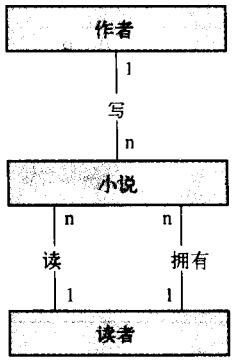



图 11-9　简单实体－关系图

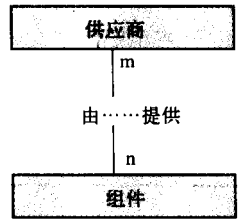



图 11-10　多对多实体－关系图

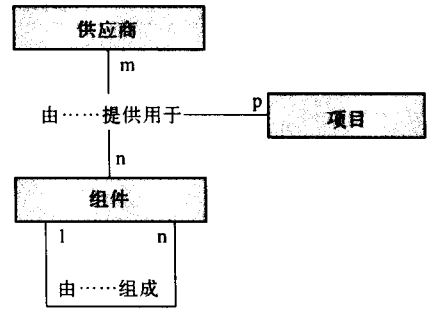



图 11-11　更复杂的实体－关系图

本章的下一个主题是形式化的规格说明技术，接下来的四节蕴含的主题是，使用形式化的规格说明技术比使用半形式化或非形式化技术更可能得到精确的规格说明。然而，使用形式化技术通常要求很长时间的训练，而且使用形式化技术的软件工程师需要有相关的数学训练。下面几节已经是用最少的数学概念编写了，进一步地，如果可能，数学公式用同样内容的非形式化表示来处理。尽管如此，11.7 节到 11.10 节仍比本书的其他部分要求的数学标准要高。

## 11.7　有穷状态机

考虑下面的例子，它最初是由英国 Open 大学的 M202 小组明确表述的［Brady，1977］。一个保险箱有一个号码锁，有三个位置，分别标为 1、2 和 3。转盘可以转向左或转向右（L

或R)。这样，在任何时候，可能有6种转盘动作：1L、1R、2L、2R、3L和3R。保险箱的号码组合是1L、3R、2L，任何其他的转盘动作将引起报警声。在图11-12中描述了这种情形，有一个初始状态"保险箱锁定"，如果输入是1L，那么下一状态是A，但是如果做出其他转盘动作，比如说1R或3L，那么下一个状态是"声音报警"——两个最终状态之一。如果选择了正确的号码组合，那么转换序列是从"保险箱锁定"到"A"到"B"到"保险箱解锁"——另一个最终状态。图11-12显示了有穷状态机的一个**状态转换图**（state transition diagram，STD）。不一定必须图形化描述一个STD，同样的信息以表格的形式显示在表11-2中。对除两个最终状态以外的每个状态，根据转盘的动作方式，指出了向下一个状态的转换。

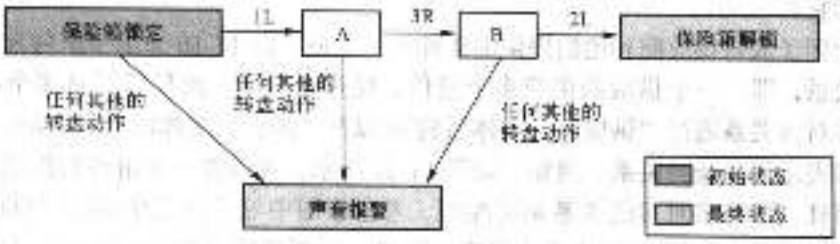



图11-12　一个组合密码箱的有穷状态机表示

表11-2　图11-12的有穷状态机的转换表

| 转盘动作 | 下一状态表 | | | |
|---|---|---|---|---|
| | 当前状态 | 保险箱锁定 | A | B |
| 1L | | A | 声音报警 | 声音报警 |
| 1R | | 声音报警 | 声音报警 | 声音报警 |
| 2L | | 声音报警 | 声音报警 | 保险箱解锁 |
| 2R | | 声音报警 | 声音报警 | 声音报警 |
| 3L | | 声音报警 | 声音报警 | 声音报警 |
| 3R | | 声音报警 | B | 声音报警 |

一个**有穷状态机**（finite state machine，FSM）由5部分组成：状态集J、输入集K、转换函数T（在给定当前状态和当前输入的情况下，它规定下一状态）、初始状态S和最终状态集F。在保险箱的组合密码锁情形下：

- 状态集J是{保险箱锁定，A，B，保险箱解锁，声音报警}；
- 输入集K是{1L，1R，2L，2R，3L，3R}；
- 转换函数T的描述见表11-2；
- 初始状态S是"保险箱锁定"；
- 最终状态集F是{保险箱解锁，声音报警}。

用更形式化的术语表达，一个有穷状态机是一个5-元组（J，K，T，S，F），其中：

J是一个有穷的非空**状态集**；

K是一个有穷的非空**输入集**；

T是一个从（J~F）×K转换到J的函数，称为**转换函数**；

S∈J 是**初始状态**；

F 是**最终状态集**，F⊆J。

在计算应用方面，有穷状态机的使用很广泛。例如，每个菜单驱动用户界面是一个有穷状态机的实现，一个菜单的显示相当于一个状态，而在键盘上键入一个输入或用鼠标选择一个图标是一个使产品进入某些其他状态的**事件**。例如，当主菜单出现在屏幕上时，输入 V 可能导致对当前数据集执行容量分析。然后出现一个新菜单，用户可能输入 G、P 或 R，选择 G 使计算结果用图形显示，P 使得打印结果，而 R 则返回主菜单。每个转换规则具有下列格式：

**当前状态**［菜单］　与　**事件**［选项选择］⇒**下一个状态**　　(11.1)

为了规定一个产品，一个有用的 FSM 扩展是将第 6 个组成元素，一个谓词（predicate）集 P，加到前面的 5 元组中。这里，每个谓词是产品的全局状态 Y 的函数［Kampen，1987］（一个**谓词**要么是真，要么是假）。更形式化地说，转换函数 T 现在是从（J～F）×K×P 转换为 J 的函数。**转换规则**现在具有如下形式：

**当前状态**　与　**事件**　与　**谓词**⇒**下一个状态**　　(11.2)

有穷状态机是制定产品规格说明的一个功能强大的形式化方法，可以根据状态和状态间的转换对产品进行建模。为了理解这个形式化方法在实际中是如何工作的，现将这项技术应用于一个所谓的“电梯问题”的修正版本，在下面的“如果你想知道”部分中，可以了解电梯问题的背景信息。

**如果你想知道**

电梯问题（elevator problem）确实是软件工程的经典问题。它 1968 年首次出现在 Don Knuth 的划时代著作《计算机程序设计艺术》（The Art of Computer Programming）的第一卷中，它是以加州理工学院数学楼的单个电梯为基础提出的。这个例子用来说明虚构的编程语言 MIX 中的协同程序。

到 20 世纪 80 年代中期，该电梯问题被推广成 $n$ 电梯问题，另外，答案的详细而精确的属性也得以证明，例如，一个电梯最终将在一个有限的时间内到达。就是现在，工作在形式化规格说明语言以及被提议的任何形式化规格说明语言领域的研究者们，都需要解决电梯问题。

这个问题在 1986 年得到广泛关注，当时它刊登在《ACM SIGSOFT Software Engineering Notes》上，征集参加第四次国际软件规格说明和设计研讨会的论文［IWSSD，1986］。电梯问题是研究者用作例子提交给会议的 5 个问题之一，会议于 1987 年 5 月在加利福尼亚的蒙特雷举行。在它以“征集论文”的形式出现时，它被称为升降机问题（lift problem），这个问题的出现归因于 STCIDEC（位于英国 Stevenage 的一个标准电信和电缆公司）的 N.（Neil）Davis。

从那时起，这个问题越来越著名，它常常被用来示范软件工程内的各种技术，不仅仅是形式化规格说明语言。本书中用它说明每项技术，因为就像你不久就会看到的那样，这个问题一点也不像它看起来的那样简单。

**有穷状态机：电梯问题实例研究**

这个问题关注的是，按照下面的约束，在 m 层楼之间移动 n 部电梯所需的逻辑：

1）每个电梯有一组 m 个按钮，每层对应一个按钮。当按下按钮并让电梯降临相应的层时，按钮灯亮。当电梯已降临相应层后灯灭。

2）除了第一层和顶层，每一层有两个按钮，一个要求电梯向上，一个要求电梯向下。当按下时，这些按钮灯亮，当电梯降临该层时，灯灭，然后向想要的方向移动。

3）当没有对电梯提出请求时，它仍停留在当前的楼层，门关着。

现在使用一个扩展的有穷状态机对该产品进行规格说明 [Kampen，1987]。在这个问题中有两个按钮集，在 n 个电梯的每个电梯中，有一组 m 个按钮，每层对应一个按钮。因为这些 n × m 按钮是在电梯里，称它们为**电梯按钮**（elevator buttons，下简称 EB）。另外，在每层楼有两个按钮，一个请求电梯向上，一个请求电梯向下，称它们为**楼层按钮**（floor buttons）。

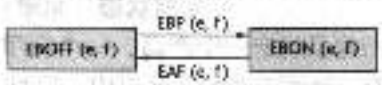



图 11-13　电梯按钮的 STD
[Kampen，1987]（©1987 IEEE）

一个电梯按钮的状态转换图示于图 11-13 中，让 EB（e，f）表示在电梯 e 里，被按下请求到楼层 f 的按钮。EB（e，f）可能处于两种状态，按钮灯开（灯亮）或关。更准确地说，状态是

EBON（e，f）：　电梯按钮（e，f）开启
EBOFF（e，f）：　电梯按钮（e，f）关闭　　(11.3)

如果按钮灯开而电梯到达 f 层，那么按钮灯关。反之，如果按钮灯关且将它按下，那么按钮灯开。这样，涉及两个事件：

EBP（e，f）：　按下电梯按钮（e，f）
EAF（e，f）：　电梯 e 到达楼层 f　　(11.4)

注：EBP——Elevator Button Pressed，按下电梯按钮。
EAF——Elevator Arrives at Floor，电梯到达楼层。

为了定义连接这些事件和状态的状态转换规则，需要一个谓词 V（e，f）：

V（e，f）：　电梯 e 到达（停在）楼层 f　　(11.5)

现在可以描述形式化转换规则。如果电梯按钮（e，f）为关（当前状态），按下（事件）电梯按钮（e，f），并且电梯 e 现在没有到达楼层 f（谓词），那么按钮开启。按照转换规则（11.2）的形式，这变成：

EBOFF（e，f）　与　EBP（e，f）　与非 V（e，f）⇒EBON（e，f）　　(11.6)

如果电梯现正到达楼层 f，则什么也不发生。在 Kampen 的形式化方法中，不触发转换的事件确实可能发生，但是如果它们发生了，则将它们忽略。

反过来，如果电梯到达楼层 f 且按钮处于开启状态，那么关闭电梯按钮，如下表达：

EBON（e，f）　与　EAF（e，f）⇒EBOFF（e，f）　　(11.7)

接下来考虑楼层按钮。FB（d，f）表示楼层 f 的按钮，它请求电梯向方向 d 移动。楼层按钮 FB（d，f）的 STD 如图 11-14 所示，更准确地说，状态是：

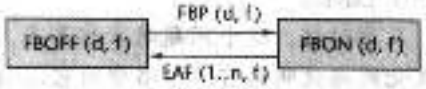



图 11-14　楼层按钮的 STD
[Kampen，1987]（©1987 IEEE）

FBON (d, f):　楼层按钮 (d, f) 开启
FBOFF (d, f):　楼层按钮 (d, f) 关闭　(11.8)

如果按钮开启且一个电梯向着正确的方向 d 移动，到达楼层 f，那么该按钮关闭。反过来，如果按钮关闭且将它按下，那么该按钮灯亮。同样，涉及两个事件：

FBP (d, f):　按下楼层按钮 (d, f)
EAF (1…n, f):　电梯 1 或……或 n 到达楼层 f　(11.9)

注意，使用 1…n 表示析取。在本节中表达式 P (a, 1…n, b) 表示：

P (a, 1, b) 或 P (a, 2, b) 或……或 P (a, n, b)　(11.10)

为了定义与这些事件和状态有关的状态转换规则，同样需要一个谓词。在这种情况下，它是 S (d, e, f)，如下定义：

S (d, e, f)：电梯 e 到达楼层 f 且它将要移动的方向是向上 (d=U)、向下 (d=D)，或者无请求时为停留状态 (d=N)　(11.11)

这个谓词实际上是一个状态，实际上，形式化方法允许将事件和状态都当作谓词对待。

使用 S (d, e, f)，那么形式转换是：

FBOFF (d, f) 与 FBP (d, f) 与非 S (d, 1…n, f) ⇒FBON (d, f)
FBON (d, f) 与 EAF (1…n, f) 与 S (d, 1…n, f) ⇒FBOFF (d, f)、d=U 或 D　(11.12)

也就是说，如果在楼层 f 向 d 方向移动的按钮关闭，按下该按钮并且当前没有向 d 方向移动的电梯到达楼层 f，则楼层按钮开启。反过来，如果该按钮开启，而且最少有一个电梯到达楼层 f 且该电梯将向 d 方向移动，那么该按钮关闭。在 S (d, 1…n, f) 和 EAF (1…n, f) 中的标记 1…n 的定义见式 (11.10)。式 (11.5) 的谓词 V (e, f) 可以根据 S (d, e, f) 如下定义：

V (e, f) =S (U, e, f) 或 S (D, e, f) 或 S (N, e, f)　(11.13)

定义电梯按钮和楼层按钮的状态都是很直观的。现在转到电梯上来，会出现一些复杂情况。一个电梯的状态本质上是由一些子状态组成的，Kampen [1987] 确定了几个，如电梯减速并停下来、电梯门开、带有定时器延时的电梯门开、或者电梯门在一个时延后关。他做出了合理的假设，**电梯控制器**（指导电梯移动的装置）初始化一个状态如 S (d, e, f)，然后该控制器通过子状态移动电梯。可以定义三个电梯状态，其中一个 S (d, e, f) 定义在式 (11.11) 中，但为了完整起见也将它列在下面：

M (d, e, f):　电梯 e 正向 d 方向移动（下一个到达的楼层是 f）
S (d, e, f):　电梯 e 停在楼层 f（要开往 d 方向）　(11.14)
W (e, f):　电梯 e 正在楼层 f 等待（门关闭）

这些状态如图 11-15 所示，注意将三个停止状态 S (U, e, f)、S (N, e, f) 和 S (D, e, f) 组合在一起成为一个更大的状态，这样简化了该示意图，减小了整个的状态数。

可以触发状态转换的事件是 DC (e, f) ——在楼层 f 关闭电梯 e 的门；ST (e, f) ——随着电梯接近楼层 f，电梯上的传感器被触发，电梯控制器必须决定是否将电梯停在该层；RL——当电梯按钮或楼层按钮按下时进入它的 ON 状态：

DC (e, f)：在楼层 f，电梯 e 的门关闭
ST (e, f)：随着电梯 e 接近楼层 f，触发传感器　(11.15)
RL：登录请求（按下按钮）

注：DC——Door Closed，关门
　　ST——Sensor Triggered，触发传感器
　　RL——Request Logged，登录请求

这些事件示于图 11-15 中。

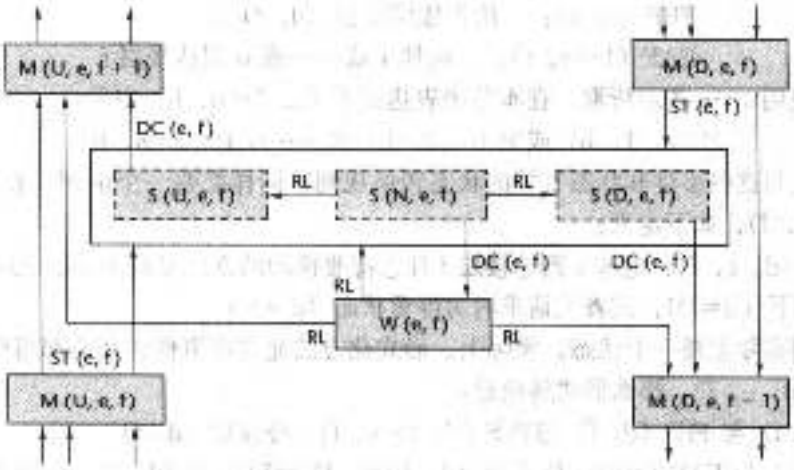



图 11-15　电梯问题实例研究的 STD［Kampen，1987］

最后，可以给出一个电梯的状态转换规则。从图 11-15 中可以推断出它来，但是在某些情况下需要有额外的谓词。如果要求更精确，图 11-15 就显得不太确定了，由于有其他原因，必须要有谓词以使 STD 是确定的。感兴趣的读者可以参考［Kampen，1987］，从中可得到完整的规则集。为了简明起见，这里所提出的规则只是发生在电梯门关闭之时。电梯向上、向下移动或进入一个等待状态，取决于当前的状态：

$$\begin{aligned} &S\ (U,\ e,\ f)\ 与\ DC\ (e,\ f) \Rightarrow M\ (U,\ e,\ f+1) \\ &S\ (D,\ e,\ f)\ 与\ DC\ (e,\ f) \Rightarrow M\ (D,\ e,\ f-1) \\ &S\ (N,\ e,\ f)\ 与\ DC\ (e,\ f) \Rightarrow W\ (e,\ f) \end{aligned} \tag{11.16}$$

第一个规则表明，如果电梯 e 处于状态 S（U，e，f），即停在楼层 f，电梯门关，并且准备上升，则该电梯将向上一层移动。第二和第三个规则对应于电梯将下降或没有要处理的请求的情形。

这些规则的格式反映了有穷状态机说明复杂产品的强大功能。规格说明并非是列出一系列复杂的前置条件让产品做什么，然后列出产品处理后的全部状态；而是采用了一种简单的描述格式：

**当前状态　与　事件　与　谓词⇒下一个状态**

这种类型的规格说明易于编写，易于确认，并且易于转换成为设计和代码。事实上，很容易构建一个 CASE 工具，它将一个有穷状态机规格说明直接翻译成为源代码。维护通过重放（replay）得到，也就是，如果需要新的状态或事件，修改规格说明，一个新版本的产品直接从新的规格说明中产生。

FSM 方法比 11.3.1 节给出的 Gane 和 Sarson 的图形化技术更精确，而且和它一样易于理解。不过它有一个缺点，对于大的系统，（状态，事件，谓词）三元组的数量会迅速增长。而

且，像 Gane 和 Sarsen 的图形化技术一样，Kampen 的形式化技术对定时需求也未加以处理。

这些问题可以使用状态图（statechart）解决，它是 FSM 的一个扩展［Harel et al., 1990］。状态图功能非常强大，并且由一个 CASE 工作平台 Rhapsody 支持。这个方法已经成功地用于一些大型实时系统。

另一个可以处理定时问题的形式化技术是 Petri 网。

## 11.8　Petri 网

说明并发系统的一个主要困难是处理定时问题。这个困难通过许多途径表现出来，如同步问题、竞争条件以及死锁［Silberschatz, Galvin, and Gagne, 2002］。定时问题经常是由不好的设计或有错误的实现引起的，而这些设计和实现通常又是由不好的规格说明引起的。如果没有正确地拟制规格说明，就会引发实际危险：相应的设计和实现将是不适当的。用于说明隐含有定时问题的系统的一个功能强大的技术是 Petri 网，这项技术一个很大的优点是它也可以用于设计。

**Petri 网**是由 Carl Adam Petri 发明的［Petri, 1962］。最初它只引起自动控制理论工作者的兴趣，后来 Petri 网在计算机科学中得到广泛的应用，它用在像性能评估、操作系统以及软件工程这些领域。特别是，Petri 网被证明可有效地描述并发关系活动。但是，在使用 Petri 网做规格说明之前，需要向那些对它不熟悉的读者简单介绍一下 Petri 网。

Petri 网由四部分组成：位置集 P、转换集 T、输入函数 I 以及输出函数 O。看一下图 11-16 所示的 Petri 网。

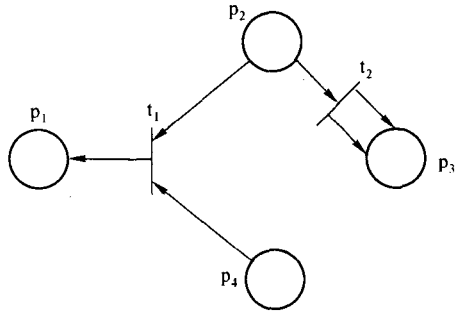



图 11-16　Petri 网

位置集 P 是 $\{p_1, p_2, p_3, p_4\}$。

转换集 T 是 $\{t_1, t_2\}$。

两个转换的输入函数由位置指向转换的箭头表示，它们是：

$$I(t_1) = \{p_2, p_4\}$$

$$I(t_2) = \{p_2\}$$

两个转换的输出函数由转换指向位置的箭头表示，它们是：

$$O(t_1) = \{p_1\}$$

$$O(t_2) = \{p_3, p_3\}$$

注意这里有两个 $p_3$，从 $t_2$ 到 $p_3$ 有两个箭头。

更形式化的 Petri 网结构［Peterson, 1981］是一个 4 元组，C=（P, T, I, O）。

P= $\{p_1, p_2, \cdots, p_n\}$ 是一个有穷**位置**集，$n \geqslant 0$。

T= $\{t_1, t_2, \cdots, t_m\}$ 是一个有穷**转换**集，$m \geqslant 0$，P 和 T 无交集。

I：$T \rightarrow P^{\infty}$是**输入函数**，从转换到位置无序单位组（bags）的映射。

O：$T \rightarrow P^{\infty}$是**输出函数**，从转换到位置无序单位组的映射。

（无序单位组（bag）或多重组（multiset）是允许一个元素有多个实例的广义集。）

**标记**（marking）一个 Petri 网是给那个 Petri 网分配**令牌**（token）。图 11-17 包含 4 个令牌：$p_1$ 中有一个，$p_2$ 中有两个，$p_3$ 中没有，$p_4$ 中有一个。标记可以用向量（1, 2, 0, 1）

表示。转换 $t_1$ **被允许**（准备激发），因为在 $p_2$ 和 $p_4$ 中有令牌。通常，如果每个输入位置中的令牌等于位置到转换的弧的数量时，就允许转换。如果 $t_1$ 被激发，将从 $p_2$ 和 $p_4$ 上分别移出一个令牌，并将一个新的令牌放入 $p_1$ 中。令牌数不守恒——移出两个令牌，但只将一个新令牌放入 $p_1$ 中。在图 11-17 中，转换 $t_2$ 也是被允许的，因为在 $p_2$ 中有令牌。如果 $t_2$ 被激发，将从 $p_2$ 中移出一个令牌，而将两个新令牌放入 $p_3$ 中。

Petri 网是非确定性的，即，如果能够激发多个转换，那么它们中的任一个都可以被激发。图 11-17 具有标记（1，2，0，1），$t_1$ 和 $t_2$ 都是被允许的。假定 $t_1$ 激发，得到的标记（2，1，0，0）显示在图 11-18 中，这里只有 $t_2$ 是被允许的。它激发，使令牌从 $p_2$ 中移出，两个新的令牌放入 $p_3$ 中。现在标记是（2，0，2，0），如图 11-19 所示。

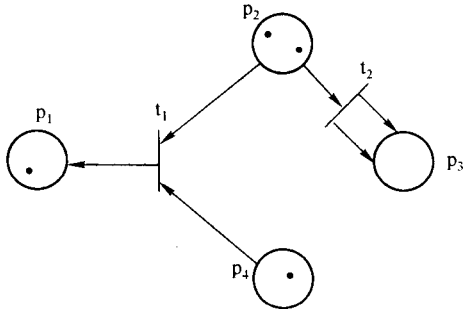



图 11-17　带标记的 Petri 网

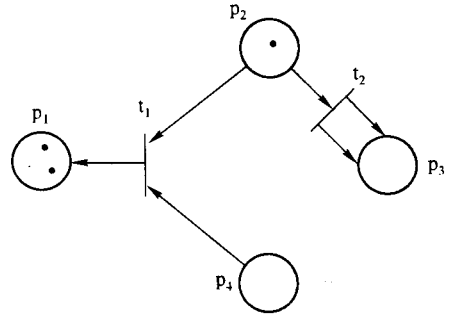



图 11-18　图 11-17 中的 Petri 网在转换 $t_1$ 激发后的情况

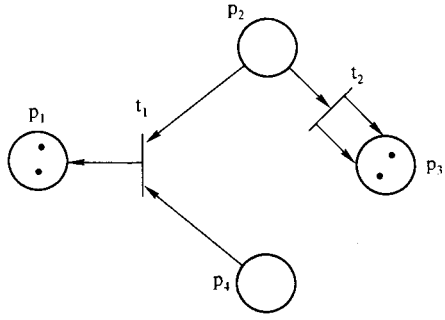



图 11-19　图 11-18 中的 Petri 网在转换 $t_2$ 激发后的情况

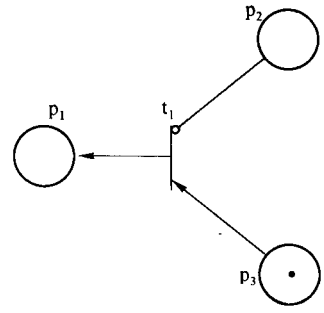



图 11-20　含禁止弧的 Petri 网

更形式化地［Peterson，1981］，一个 Petri 网 C=（P，T，I，O）的标记 M，是从位置集 P 到非负整数集的一个函数：

$$M: P \rightarrow \{0, 1, 2, \cdots\}$$

则一个**带标记的 Petri 网**是一个 5 元组（P，T，I，O，M）。

Petri 网的一个重要扩充是加入**禁止弧**。在图 11-20 中，禁止弧是用一个小圆圈而不是一个箭头标记的。转换 $t_1$ 被允许，因为在 $p_3$ 中有一个令牌，但在 $p_2$ 中没有。通常情况下，如果一个转换的每个（常规）输入弧上至少有一个令牌，而禁止输入弧上没有令牌，那么这个转换被允许。这个扩充被用在 11.7.1 节的电梯问题实例研究的 Petri 网规格说明中［Guha，Lang，and Bassiouni，1987］。

**Petri 网：电梯问题实例研究**

回想一下，将要在一个 m 层的楼内安装 n 个电梯的系统。在这个 Petri 网规格说明中，楼房内的每一层用 Petri 网中的位置 $F_f$ 表示，$1 \leqslant f \leqslant m$；一个电梯用一个令牌表示。在 $F_f$ 中的一个令牌表示一个电梯处在 f 层。

**第一个约束**

每个电梯有 m 个按钮，每个按钮代表一层。当按下时这些按钮灯亮，使电梯到达相应的楼层。当电梯到达相应的楼层时，按钮灯灭。

为了将其与规格说明相协调，需要附加的位置。楼层 f 的电梯按钮在 Petri 网中用位置 $EB_f$ 表示，$1 \leqslant f \leqslant m$。更准确地说，因为有 n 个电梯，位置应当表示为 $EB_{f,e}$，$1 \leqslant f \leqslant m$，$1 \leqslant e \leqslant n$，但是，为了简单起见，表示电梯的下标 e 被省去了。$EB_f$ 中的令牌表示楼层 f 的电梯按钮亮。因为按钮只有在第一次被按下时才亮，而后来再按也只会被忽略，这一点使用 Petri 网进行了规格说明，如图 11-21 所示。首先，假定按钮 $EB_f$ 没有亮，显然在位置上没有令牌，并且因为禁止弧的存在，转换“$EB_f$ 按下”被允许。该按钮现在被按下，该转换激发并且将一个新的令牌放入 $EB_f$，如图 11-21 所示。现在，不管该按钮被按下多少次，禁止弧和令牌的同时存在意味着那个转换“$EB_f$ 按下”不被允许。因此，在位置 $EB_f$ 的令牌数不会多于 1。

现在进一步假定电梯要从楼层 g 移动到楼层 f，因为电梯是在楼层 g，因此令牌是在位置 $F_g$ 中，如图 11-21 所示。转换“电梯在运行”被允许并激发，在 $EB_f$ 和 $F_g$ 中的令牌被清除，关闭按钮 $EB_f$ 并且在 $F_f$ 中出现一个新的令牌，这个转换的激发将电梯从楼层 g 移动到楼层 f。

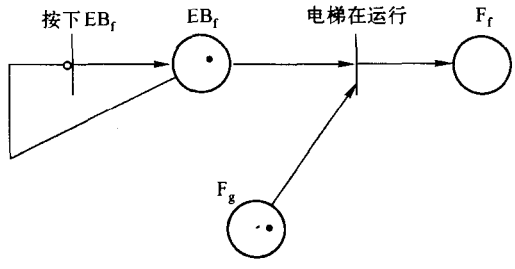



图 11-21　电梯按钮的 Petri 网表示
[Guha, Lang, and Bassiouni, 1987]

从楼层 g 到楼层 f 的运动不可能立刻发生，为了处理这个以及类似的问题，比如一个按钮在物理上不可能在按下它后非常短的瞬间亮起，必须向 Petri 网模型加入定时要求。也就是说，尽管在传统 Petri 网理论中，转换是立即发生的，而在实际情形中（如电梯问题），需要用定时的 Petri 网把一个非零时间与一个转换联系起来。

**第二个约束**

每层楼（除了第一层楼和顶层之外）都有两个按钮，一个请求电梯向上，一个请求电梯向下。当按下时这些按钮灯亮，当电梯到达该楼层然后向想要的方向移动时，按钮灯灭。

楼层按钮用位置 $FB_f^u$ 和 $FB_f^d$ 表示，它们分别代表请求电梯向上和向下的按钮。更精确地说，楼层 1 有一个按钮 $FB_1^u$，楼层 m 有一个按钮 $FB_m^d$，中间楼层每个都有两个按钮 $FB_f^u$ 和 $FB_f^d$，$1 < f < m$。当一个电梯从楼层 g 到达楼层 f 时，有一个按钮亮或两个按钮都亮的情形如图 11-22 所示。事实上，此图需要进一步求精，因为如果两个按钮都亮，根据不确定原理一个按钮被关闭。为了确保关闭正确的按钮，要求的 Petri 模型过于复杂，不便在这里给出，例子可见 [Ghezzi and Mandrioli, 1987]。

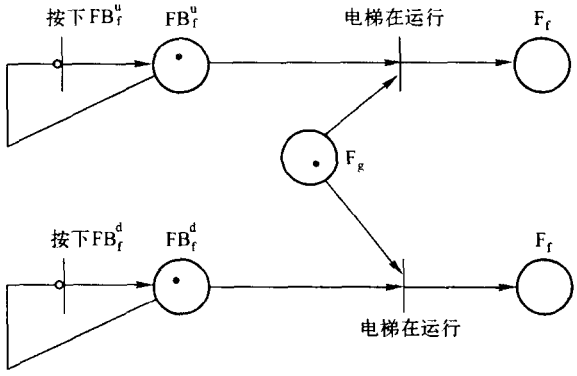



图 11-22　楼层按钮的 Petri 网表示［Guha，Lang，and Bassiouni，1987］

**第三个约束**

当电梯没有请求时，它停留在当前的楼层，电梯门关着。

这很容易做到：如果没有请求，没有“电梯在运行”转换被允许。

Petri 网不仅能用于表示规格说明，它们也可以用于设计［Guha，Lang，and Bassiouni，1987］。然而，即使在产品开发的这个阶段，Petri 网也具有确定并发系统同步方面的特性所必需的强大的表现力。

## 11.9　Z

获得广泛认可的一个形式化规格说明语言是 Z［Spivey，2001］（关于名称 Z 的正确读音参见下面的“如果你想知道”部分）。Z 的使用要求集合论、函数、离散数学（包括一阶逻辑）的知识，即使对于具有相当背景的用户（这包括多数计算机专业的学生），开头也难于理解 Z，因为除了通常的集合论的符号和逻辑符号，如∃、⊃和⇒之外，它还使用许多不常见的特别符号，如⊕、⩤、↦等。

**如果你想知道**

Z 是发明者 Jean-Raymond Abrial 给形式化规格说明语言命名的名称，其目的是为了纪念著名的集合论专家 Ernst Friedrich Ferdinand Zermelo（1871—1953）。因为它是在牛津大学发展起来的［Abrial，1980］，名称 Z 正确的发音为“zed”，英国人读字母表中第 26 个字母的方式。

但后来，人们认为 Z 是以一位德国数学家的名字命名的，将它按德国发音方式读为“tzet”。与此对应，亲法者指出 Abrial 是一个法国人，字母 Z 按法国读音方式也应读为“zed”。

完全不可接受的发音方式是美国式的，读音为“zee”。原因是 Z（读音为“zee”）是一个美国第四代语言的名字（见 14.2 节）。然而，我们不能为字母表的单个字母注册商标。进一步地说，我们可以自由地按我们的意愿读字母 Z。然而，在编程语言范围内，发音“zee”指 4GL，而不是形式化规格说明语言。

让我们对下一轮的 Z 读音大战拭目以待。

为了深入了解如何使用 Z 规定一个产品，再次考虑 11.7.1 节的电梯问题实例研究。

### 11.9.1　Z：电梯问题实例研究

按照它最简单的形式，一个 Z 规格说明由下面四部分组成：

1）给定的集合、数据类型和常数；

2）状态定义；

3）初始状态；

4）操作。

下面依次分析这些部分。

**1. 给定的集合**

一个 Z 规格说明从一个**给定的集合**（given set）的列表开始，即集合不需要详细定义。任何这样的集合的名字出现在方括号中。对于电梯问题，给定的集合称为 Button，它是所有按钮的集合。因此 Z 规格说明开始于：

```
[Button]
```

**2. 状态定义**

一个 Z 规格说明由一些语句集组成，每个**语句集**由一组变量声明和一系列限制变量的取值范围的谓词组成。语句集 S 的格式如图 11-23 所示。

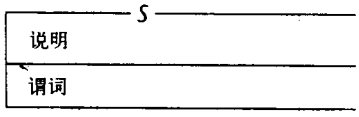



图 11-23　Z 语句集 S 的格式

在电梯问题实例研究中，有四个 Button 的子集：floor _ buttons（楼层按钮）、elevator _ buttons（电梯按钮）、buttons（电梯问题实例研究中所有按钮的集合）以及 pushed——那些被按下（因此开启）的按钮的集合。图 11-24 描述了语句集 Button _ State，一个**状态定义**。符号 **P** 表示幂（power）集（一个给定集的全部子集的集合）。约束，即在水平线以下的语句，表明 floor _ buttons 集合和 elevator _ buttons 集合没有交集，它们共同组成了 buttons 集合。（集合 floor _ buttons 和集合 elevator _ buttons 在接下来不需要，图 11-24 中包含它们只是为了说明 Z 的幂。）

*Button_State*

| | |
|---|---|
| floor_buttons, elevator_buttons | : **P** Button |
| buttons | : **P** Button |
| pushed | : **P** Button |

floor_buttons ∩ elevator_buttons = ∅
floor_buttons ∪ elevator_buttons = buttons

图 11-24　Z 语句集 Button _ State

**3. 初始状态**

**抽象初始状态**描述了系统最初开启时的状态。电梯问题实例研究的抽象初始状态是：

$$Button\_Init \triangleq [Button\_State' \mid pushed' = \Delta]$$

这个状态显示了**垂直语句集定义**，与**水平语句集定义**相反，如图 11-24 所示。垂直语句集表明，当电梯系统最初开启时，pushed 集合最初是空的，即全部的按钮关闭。

**4. 操作**

如果最先按下一个按钮，那么那个按钮开启，这个按钮加到 pushed 集合中。图 11-25 描述了这一点，在那里定义了 Push _ Button 操作。语句集第一行的△表示这个操作改变了

Button _ State 的状态，这个操作有一个输入变量 button?。像在各种其他语言中一样（如 CSP［Hoare，1985］），问号（?）表示一个输入变量，而惊叹号（!）表示一个输出变量。

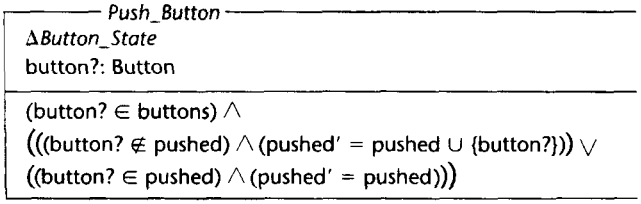



图 11-25　操作 Push _ Button 的 Z 规格说明

一个**操作**的谓词部分由一组操作前被调用的前置条件，以及操作完全结束后的后置条件组成。假定满足了前置条件，则执行完成后可得后置条件。然而，如果在前提条件不满足的情况下调用操作，可能产生未规定（因此是不可预测）的结果。

图 11-25 的第一个前置条件表明 button?必须是 buttons（在这个电梯系统中它是所有按钮的集合）的一个成员。如果满足了第二个前提条件，button?∉pushed（即如果该按钮没有开启），则更新 pushed 按钮集，使之包含 button?。在 Z 中，一个变量的新的值用符号“′”表示，因此后置条件是：在执行完操作 Push _ Button 后，button?按钮必须加到 pushed 集合中。不需要直接开启按钮，button?现在是 pushed 集合的一个元素已经足够了。

另一个可能性是一个已经按下的按钮又再一次被按下，因为 button?∈pushed，根据第三个前置条件⊖，将没有任何事发生。这可用 pushed′ = pushed 表示，pushed 的新状态与旧状态相同。

现在假定电梯到达某层楼，如果相应的楼层按钮开，那么必须关闭它，对于相应的电梯按钮也同样。即是说，如果 button?是 pushed 的一个元素，那么必须将它从这个集合中移出，如图 11-26 所示（符号 \ 表示集合相异）。然而，如果按钮没有开启，则集合 pushed 不变。

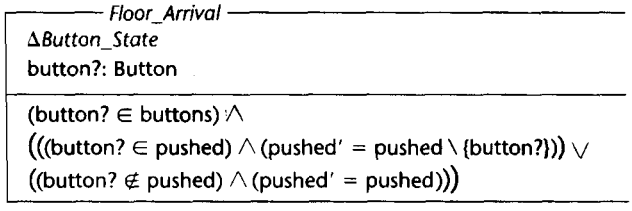



图 11-26

这一节中给出的解决方案过于简单，它没有区分向上和向下的楼层按钮。尽管如此，它还是给出了如何用 Z 来在电梯问题实例研究中说明按钮动作的。

### 11.9.2　Z 的分析

Z 已经在范围广泛的项目中使用，包括 CASE 工具［Hall，1990］、一个实时内核

⊖ 如果没有第三个前置条件，规格说明不会说明如果一个已按下的按钮再次被按下将会发生什么。那么结果就是未规定的。

[Spivey, 1990] 以及一个示波器 [Delisle and Garlan, 1990]。在 CICS（IBM 事务处理系统）的一个发布版中，有很大一部分是使用 Z 进行规格说明的 [Nix and Collins, 1988]。

这些成功应用也许有些令人吃惊，而我们看到的事实是，即使对于电梯问题实例研究的简化版；Z 也显然不那么容易使用。首先是由符号引起的问题，一个新用户在阅读 Z 规格说明以前，必须学习符号的集合以及它们的含义，更不要说编写它们了。其次，并非每个软件工程师都具有数学方面的知识来使用 Z（尽管现在大多数计算机专业毕业生或者知道了足够的使用 Z 的数学知识，或者能够不费劲地学习他们还需要知道的新知识）。

Z 可能是它这种类型的最广泛使用的形式化语言，为什么是这样？为什么 Z 这样成功，特别是在大型项目上？原因有以下几点：

- 用 Z 写的规格说明中的错误可以比较容易地被发现，特别是在规格说明自身检查期间，以及根据形式化规格说明检查设计或代码期间 [Nix and Collins, 1988; Hall, 1990]。
- 编写 Z 规格说明要求规格说明编写人非常精确，作为这个精确性要求的结果，显然比非形式化规格说明存在较少的模糊、矛盾以及遗漏。
- 作为一个形式化语言，Z 允许开发者在必要时证明规格说明是正确的。这样，尽管有些公司很少做 Z 的正确性证明，但这样的证明已经做过了，甚至连像 CICS 存储管理器这样的实际的规格说明都做了正确性证明 [Woodcock, 1989]。
- 有人曾建议仅具有高中数学知识的软件专业人员可以在相当短的时间内学写 Z 规格说明 [Hall, 1990]，显然这些人不能证明得到的规格说明是正确的，但那时没有必要证明形式化规格说明是正确的。
- 使用 Z 降低了软件开发的成本。毫无疑问，比起使用非形式化规格说明技术来，形式化规格说明本身要花费更多的时间，但是整个开发过程所用的总时间减少了。
- 客户不理解用 Z 写的规格说明这个问题已经有一些解决方法，包括用自然语言重写规格说明。人们发现，这样得到的自然语言规格说明比从头构建的非形式化规格说明更清晰。（这也是 Meyer 的经验，他对 11.2.1 节描述的 Naur 的文本处理问题的形式化规格说明进行了英语解释。）

最后，尽管有相反的观点，Z 在软件业界还是已经成功地用于一些大型项目。虽然大多数规格说明仍然是用远不如 Z 形式化的语言写的，但使用形式化规格说明是全球发展的总趋势。使用这样的形式化规格说明语言过去主要是欧洲的习惯，然而，越来越多的美国公司也开始使用一些形式化规格说明技术。Z 及类似的语言在将来的使用程度尚须拭目以待。

## 11.10　其他的形式化技术

许多其他的形式化技术也已被提出，这些技术差异很大，例如，**Anna** [Luckham and von Henke, 1985] 是 Ada 的形式化规格说明语言。某些规格说明语言是基于知识的，如 **Gist** [Balzer, 1985]。设计 Gist 是为了使用户能够以一种近似于思考过程的方式来描述处理过程，这通过对自然语言中使用的结构进行形式化来实现。实际上，Gist 规格说明与大多数其他的形式化规格说明一样难读，以至于有人曾编写了从 Gist 转换到英语的解释器。

**Vienna 定义方法**（Vienna definition method, VDM）[Jones, 1986b] 是一项基于符号语义学 [Gordon, 1979] 的技术，VDM 不仅可以用于规格说明，也可以应用于设计和实现。

VDM 曾经在一些项目中成功地使用，最引人注目的是在 DDC Ada 编译器系统的 Dansk Datamatik Center 开发项目中［Oest，1986］。

看待规格说明的另一种方式是根据事件顺序来描述，这里一个事件或者是一个简单的动作，或者是将数据移进或移出系统的一次通信。例如，在电梯问题实例研究中，一个事件由在电梯 e 按到楼层 f 的电梯按钮以及使该按钮的灯亮组成。另一个事件是电梯 e 朝着向下的方向离开楼层 f 以及熄灭相应楼层按钮的灯。由 Hoare［1985］发明的**通信顺序处理**（Communicating Sequential Processes，CSP）语言，就是基于以这样的事件描述系统特性的想法。在 CSP 中，根据事件顺序描述一个处理，在事件中该处理与它的环境相交互。处理之间通过发消息来交互，CSP 允许用各种方式将处理组合起来，如顺序地、并行地或非确定性地交织。

CSP 的强大功能体现在 CSP 规格说明的可执行特性［Delisle and Schwartz，1987］，结果是，可以检查它们内在的一致性。此外，CSP 提供一个框架，据此可以顺序地从规格说明进行到设计再到实现，其间保留各步骤的有效性。换句话说，如果该规格说明是正确的并且转换正确地完成了，那么设计和实现也将是正确的。如果实现语言是 Ada，从设计到实现就可直接进行。

然而，CSP 也有缺点。特别是，像 Z 一样，它不是一种容易学习的语言。本书尝试着介绍一个电梯问题实例研究的 CSP 规格说明［Schwartz and Delisle，1987］，但是适当描述每个 CSP 语句所需的预备材料和要解释的细节太多，因而难以容纳在书中。下一节介绍在规格说明语言的强大功能与使用它的困难程度之间的关系。

## 11.11　传统分析技术的比较

本章的主要内容是每个开发公司应当决定对于要开发的产品，哪种类型的规格说明语言是合适的。非形式化技术易于学习，但缺乏半形式化技术或形式化技术的强大功能。反过来，形式化技术支持多种特性，它可能包括可执行性、正确性证明，或者经过一系列保留正确性的步骤向设计和实现转换的能力。尽管通常越形式化的技术，其功能也越强大，然而形式化技术却难于学习和使用。而且，客户可能难于理解形式化规格说明，换句话说，需要有一种规格说明的应用性和功能间的权衡。

在某些环境下，规格说明语言类型的选择很容易。例如，如果开发小组的大多数成员在计算机科学方面没有经过训练，那么除了非形式化或半形式化规格说明技术之外，实际上别无选择。反之，如果正在一个研究实验室中建造一个任务重要的实时系统，几乎注定要利用形式化规格说明技术的强大功能。

另外一个复杂的因素是，许多新的形式化技术还不曾在实践环境中试验过，使用这样一种技术时包含着相当的危险。需要大量的经费来培训开发小组相关人员，当小组从在教室中使用该语言转向在实际项目中使用时，将花费更多的钱。进一步地，该语言的支持软件工具可能无法正常工作，就像 SREM 所发生的那样［Scheffer，Stone，and Rzepka，1985］，造成额外的金钱和时间浪费。但是如果每件事都正常运转，并且软件项目管理计划考虑了重要项目使用新技术所需的额外的时间和金钱，那么巨大的收益是可能的。

对于一个特定的项目应当使用哪一种分析技术？这取决于项目、开发小组、管理小组以及综合的其他因素，如客户坚持使用（或不使用）某一方法。由于还有软件工程的许多其他

方面，必须做出权衡。遗憾的是，不存在简单的规则来决定使用哪项分析技术。

表 11-3 总结了本节的观点。

**表 11-3　本章讨论的传统分析方法小结以及描述它们的章节**

| 传统分析方法 | 分　类 | 优　点 | 缺　点 |
|---|---|---|---|
| 自然语言［11.2 节］ | 非形式化 | 易于学习<br>易于使用<br>客户易于理解 | 不精确<br>规格说明可能是模糊、矛盾或不完整的 |
| 实体－关系模型（11.6 节）<br>PSL/PSA（11.5 节）<br>SADT（11.5 节）<br>SREM（11.5 节）<br>结构化系统分析（11.3 节） | 半形式化 | 可以为客户所理解<br>比非形式化方法更精确 | 不如形式化方法精确<br>通常不能处理定时问题 |
| Anna（11.10 节）<br>CSP（11.10 节）<br>扩展的有穷状态机（11.7 节）<br>Gist（11.10 节）<br>Petri 网（11.8 节）<br>VDM（11.10 节）<br>Z（11.9 节） | 形式化 | 非常精确<br>可以减少分析错误<br>可以减少开发成本和工作量<br>可以支持正确性证明 | 开发小组难于学习<br>难于使用<br>多数客户几乎不可能理解 |

## 11.12　在传统分析阶段测试

在传统分析阶段，所提议产品的功能精确地在规格说明文档中描述。检验该规格说明文档的正确性至关重要，做到这一点的一种方法是对规格说明文档进行走查（见 6.2.1 节）。

检测规格说明文档中的错误的一个强有力机制是审查（见 6.2.3 节）。审查员小组对照一览表评审规格说明，规格说明审查一览表上的典型项目包括：是否规定了所需的硬件资源？是否规定了验收准则？

审查最早是由 Fagan 在他的关于测试设计和代码的文章中提出的［Fagan，1976］。在 6.2.3 节中详细介绍了 Fagan 的工作，然而实践证明，审查在测试规格说明中也十分有用。例如，Doolan［1992］使用审查验证一个产品的规格说明，该产品在建造时由超过 200 万行 FORTRAN 语句组成。从有关修复产品错误所花成本的数据，他推断出在审查中的每小时投入可节省 30 小时的基于执行的错误检测和纠正。

当使用形式化技术拟制规格说明时，可以应用其他测试技术，例如，可以采用正确性证明方法（见 6.5 节）。即使没有进行形式化证明，像 6.5.1 节介绍的非形式化证明技术也是一种非常有用的凸显规格说明错误的方法。事实上，产品和它的证明应当并行开发，按照这种方法，可以快速检测出错误。

## 11.13　传统分析阶段的 CASE 工具

在传统分析阶段，有两类 CASE 工具特别有用。第一个是图形工具。不管一个产品是使用数据流图、Petri 网、实体－关系图还是其他的本书未介绍的表示法进行描述，手工描绘

整个产品是一个冗长的过程。另外，做出本质上的修改时，必须重新绘制每个部分。因此绘制工具将非常节省时间，这类工具适用于本章描述的分析技术，还有许多其他的图形表示法适用于规格说明。这个阶段所需的第二个工具是数据字典，像5.5节提到的那样，这个工具存储产品中每个数据项的每个组件的名字和表示（格式），包括数据流和它们的组件，数据存储和它们的组件，以及处理（动作）和它们的内部变量（表11-1显示了Sally的软件商店的数据字典中存储的典型信息）。再次声明，在各种硬件操作系统组合上有大量可选用的数据字典。

真正需要的不是单独的图形工具和数据字典，而是将这两个工具集成在一起，使对数据组件的任何修改都能自动反映到规格说明的相应部分中。这种类型的工具的例子有许多，比如Analyst/Designer、Software through Pictures以及System Architect。进一步说，许多这样的工具还结合了一个自动的一致性检查器，确保规格说明文档和相应设计文档之间的一致性。例如，能够检查规格说明文档中的每一项是否都转移到设计文档中，并且设计中提到的每一项是否都在数据字典中声明了。

一项规格说明技术不大可能得到广泛的接受，除非有一个工具丰富的CASE环境支持该项技术。例如，如果REVS（与SREM相关的CASE工具集）在美国空军的测试中能表现得更好的话，SREM（见11.5节）很可能在今天得到更广泛的应用［Scheffer，Stone，and Rzepka，1985］。正确地规定一个系统并不容易，即使对于经验丰富的软件专业人员来讲也是如此。惟一可行的是向规格说明编写者提供一套最先进的CASE工具，尽最大可能帮助他们。

## 11.14　传统分析阶段的度量

像在其他阶段中一样，在传统分析阶段必须测量5个基本的度量：规模、成本、周期、工作量和质量。规格说明规模的一个测度是规格说明文档的页数。如果使用相同的技术规定一些类似的产品，那么在规格说明规模方面的差异将最能预计建造这些产品所需的工作量。

再看质量，规格说明审查的一个最重要的方面是记录错误统计数。记下在审查期间发现的每类错误数是审查过程的一个不可缺少的部分，而且，错误检测的比率可以度量检查过程的效率。

预测目标产品规模的度量包括数据字典中的条目数。应当采用几个不同的计数，包括文件数、数据项数、处理（动作）数等，这个信息可以给管理者提供建造产品所需的工作量方面的一个初步估计。重要的是要认识到这个信息至多只是假设性的。毕竟，在传统设计阶段，一个DFD中的处理可能分解为一些不同的模块。相反，一些处理可能一起组成单个模块。不过，从数据字典得来的度量可给管理者提供目标产品最终规模的一个早期线索。

## 11.15　软件项目管理计划：Osbert Oglesby实例研究

现在规格说明已经完成了，软件项目管理计划就要拟制，它包括估算成本和周期（见第9章）。附录F中包含了由一个小的软件公司（3个人）开发的Osbert Oglesby产品的软件项目管理计划。这个计划符合IEEE SPMP格式（见9.5节）。

## 11.16　传统分析阶段面临的挑战

本章重复的主题是，一个规格说明文档必须既是非形式化的，足以让客户理解，又是形式化的，足以让开发小组将其作为要建造的产品的惟一描述。传统分析阶段面临的主要挑战是解决这个矛盾。不存在简单的答案，相反，在这两个竞争的目标之间存在持久的冲突，而开发小组只能尽其最大努力在进退两难中安全地掌舵。

传统分析阶段面临的第二个挑战是，分析（什么）和（如何）设计之间的界限太容易跨越。规格说明文档应当描述产品必须做什么，而不能说如何实现产品。例如，假定客户要求无论何时进行某一网络路由选择的计算，响应时间都要小于 0.05 秒。规格说明文档应当准确地表述这一点——仅此而已。特别是，规格说明文档不应当指出必须使用哪个算法得到这个响应时间，也就是说，规格说明文档需要列出全部的约束，但它一定不要规定如何达到这些约束。

这个潜在的缺陷的另一个例子来自于数据流图（见 11.3.1 节）。圆角矩形表示一个处理，它不表示一个模块。如 11.14 节所解释的那样，在一个 DFD 中的处理可以分解为一些不同的模块，反过来，一些处理也可以合成为单个模块。关键是处理的这个求精而形成模块必须发生在传统设计阶段，而不是发生在传统分析阶段。规格说明文档只是描述目标处理的操作，却一定不能规定这些操作是如何实现的，甚至也不要规定为每个操作分配哪些模块。设计小组的任务是从整体上研究规格说明并决定一个设计，该设计将产生那些规格说明的最佳实现，这在第 13 章中描述。在产品总体上已经分解成为模块之前，试图为具体模块安排操作为时过早，结果注定不令人满意。

### 本章回顾

规格说明（11.1 节）可以非形式化表述（11.2 节），也可以半形式化表述（11.3 节至 11.5 节），或者形式化表述（11.6 节至 11.10 节）。

本章的主题是，非形式化技术易于使用但不精确，通过一个小实例研究阐明了这一点（11.2.1 节）。相反，形式化技术功能强大，但是要求在训练时间上有不小的投入（11.11 节）。本章稍微详细地描述了一种半形式化技术——Gane 和 Sarsen 的结构化系统分析方法（11.3 节），并描述了它应用于 Osbert Oglesby 实例研究的情况（11.4 节）。然后描述了其他的半形式化技术（11.5 节），包括建造实体 - 关系模型（11.6 节）。本章给出的形式化技术包括有穷状态机（11.7 节）、Petri 网（11.8 节）和 Z（11.9 节）。其他的形式化技术在 11.10 节中进行了概述。有关规格说明评审的材料出现在 11.12 节中，接下来描述了传统分析阶段的 CASE 工具（11.13 节）和度量（11.14 节）。然后是 Osbert Oglesby 实例研究的软件项目管理计划（11.15 节），本章最后讨论了传统分析阶段面临的挑战（11.16 节）。

### 进一步阅读

有关结构化系统分析方面的经典著述来自［DeMarco，1978］、［Gane and Sarsen，1979］、［Yourdon and Constantine，1979］。这些思想已经在［Modell，1996］中进一步更新了。在 12.5 节中概要列出的面向数据设计，是集成了不同种类的半形式化规格说明技术的设计技术，有关细节见［Jackson，1975］、［Warnier，1976］和［Orr，1981］。SADT 在

[Ross，1985] 中描述，而 PSL/PSA 的描述见 [Teichroew and Hershey，1977]。两个有关 SREM 的信息源是 [Alford，1985] 和 [Scheffer，Stone，and Rzepka，1985]。

在 [Wing，1990] 中介绍了 6 种形式化技术。在以下杂志的 1990 年 9 月刊中可以找到一些有关形式化技术的著名论文：《IEEE Transactions on Software Engineering》、《IEEE Computer》、《IEEE Software》和《ACM SIGSOFT Software Engineering Notes》。其中最能引起人兴趣的是 [Hall，1990]，应当完整地阅读这篇文章。[Bowen and Hinchey，1995b] 是 Hall 的文章的续篇，而 [Bowen and Hinchey，1995a] 描述了使用形式化技术的指导原则。其他有关形式化技术的文章可以在《IEEE Transactions on Software Engineering》杂志 2000 年 8 月刊中找到。在 [Larsen，Fitzgerald，and Brooks，1996] 和 [Pfleeger and Hatton，1997] 中讨论了工业界应用形式化规格说明得到的经验教训。可以在 [Saiedian et al.，1996] 中找到各种关于形式化方法的观点。在 [Fraser and Vaishnavi，1997] 中给出了形式化规格说明的一个成熟模型。在 [Fraser and Clarkson，2002] 中对比了不同类型的形式化技术，并进行了完全根据经验的研究。[Haxthausen and Peleska，2000] 对一个分布式铁路控制系统应用了形式化验证。[Palshikar，2001] 描述了形式化规格说明在实际软件开发中的应用。[Hall and Chapman，2002] 描述了使用形式化技术的一个商业安全系统的结构。

关于有穷状态机方法的一个早期参考文献是 [Naur，1964]，遗憾的是在那里它被称为图灵机方法（Turing machine approach）。状态表是 FSM 的一个强有力的扩展，在 [Harel et al.，1990] 中介绍了它们。状态图面向对象的扩展出现在 [Harel and Gery，1997] 中。

[Peterson，1981] 是对 Petri 网及其应用的一个极好的介绍。Petri 网在原型开发中的使用参见 [Bruno and Marchetto，1986]。定时的 Petri 网在 [Coolahan and Roussopoulos，1983] 中有介绍。

关于 Z，[Diller，1994] 是一个很好的介绍性文章。关于规格说明语言的详细参考手册，请见 [Spivey，2001]。根据阅读 Z 规格说明的结果，[Finney，1996] 对 Z 规格说明是否像某些 Z 倡导者所宣称的那样易于阅读提出疑问。

“国际软件规格说明和设计研讨会”（International Workshops on Software Specification and Design）的会议录是研究有关规格说明思想的好来源。

## 习题

11.1　为什么下列约束不应当出现在规格说明文档中？

a）产品必须明显降低在加拿大西部分发我们的食品所带来的运输费用。

b）MRI 存储单元必须建立在合理的成本基础上。

11.2　考虑下面的烤 pockwester 鱼的菜谱。

配料：1 个大洋葱
1 罐冰冻橙汁
1 个柠檬压出的鲜汁
1 杯面包屑
面粉
牛奶
3 根中等大小的葱

2 个中等大小的茄子
1 条鲜 pockwester 鱼
1/2 杯 Pouilly-Fuissé 酒
1 头大蒜
Parmesan 干酪
4 个鸡蛋

前一晚，拿一个柠檬，压榨，滤汁，然后将其冷冻。拿一个大洋葱和 3 根葱，切成方块，在一个浅锅中烧它们，当开始冒黑烟时，加入两杯鲜橙汁。不停地搅动，将柠檬切成非常薄的薄片，加到混合物中。同时，给蘑菇挂上面粉，将它们在牛奶中醮一下，然后在一个纸袋中与面包屑一起摇动它们。在一个深锅中，加热 1/2 杯 Pouilly-Fuissé 酒，当它达到 170°时，加入糖并继续加热。当糖溶化成焦糖时，加入蘑菇。将混合物混合 10 分钟，或者等到所有的块状物都已消除，加入鸡蛋。现在拿来 pockwester 鱼，将它杀死。剥去 pockwester 鱼的皮，切成小块，将它加到混合物中。使其沸腾然后慢煮，不盖锅。鸡蛋先前应当用搅打器迅速搅动 5 分钟，当感觉 pockwester 软烂时，将它放到盘子上。在上面撒上 Parmesan 干酪，烤不超过 4 分钟。

确定上述规格说明中的模糊、遗漏和矛盾之处（注：pockwester 是一种假想的鱼）。

11.3　改正 11.2 节的规格说明段落，以更准确地反映客户的意愿。

11.4　使用数学公式表示 11.2 节的规格说明段落。将你的答案与习题 11.3 的答案进行比较。

11.5　给确定银行陈述是否正确的产品编写一个准确的英语规格说明（习题 8.8）。

11.6　为习题 11.5 提出的规格说明画数据流图，确保你的 DFD 只反映数据流，不做任何关于计算机化的假设。

11.7　考虑习题 8.7 的图书馆图书自动循环系统，为该系统编写精确的规格说明。

11.8　画一个数据流图，显示习题 8.7 的图书馆图书循环系统的操作。

11.9　使用 Gane 和 Sarsen 的技术完成习题 8.7 的图书馆图书循环系统的规格说明文档。这里数据未做规定（例如，每天图书返还和借出的总数），你自己做假设，但要确保清楚地指出它们。

11.10　一个浮点二进制数的组成是这样的：一个可选符号，后跟一个或多个比特位，其后跟字母 E，再跟另一个可选符号，后面是一个或多个比特位。浮点二进制数的例子有 11010E－1010，－100101E11101 和＋1E0。

更形式化地，这可以表述为：

＜浮点二进制数＞::＝［＜符号＞］＜比特串＞E［＜符号＞］＜比特串＞
＜符号＞::＝＋｜－
＜比特串＞::＝＜比特＞［＜比特串＞］
＜比特＞::＝0｜1

（符号［...］表示一个可选项，而 a｜b 表示 a 或 b）。

规定一个有穷状态机，它将一个字符串作为输入，确定该字符串是否构成一个有效的浮点二进制数。

11.11　使用有穷状态机方法规定习题 8.7 的图书馆图书循环系统。

11.12　说明习题 11.11 的解决方案如何用于设计和实现图书馆图书循环系统的菜单驱动产品（习题 8.7）。

11.13　使用一个 Petri 网规定习题 8.7 的单本书出入图书馆的循环，在你的规格说明中包括操作 $H$、$C$ 和 $R$。

11.14　你是一个软件工程师，为一个致力于图书馆系统计算机化的大公司工作。你的经理让你用 Z 规定习题 8.7 的整个图书馆图书循环系统，你的反应是什么？

11.15　（学期计划）使用你的导师规定的技术，为附录 A 描述的 Ophelia 的 Oasis 产品拟制一个规格说明文档。

11.16　（学期计划）为附录 A 描述的 Ophelia 的 Oasis 产品拟制一个软件项目管理计划。

11.17　（实例研究）使用有穷状态机方法提出 Osbert Oglesby 产品的需求。

11.18　（实例研究）使用 Petri 网技术规定 Osbert Oglesby 产品中的单幅画作经过的状态。

11.19　（实例研究）使用 11.9 节的 Z 构造规定 Osbert Oglesby 产品的一部分。

11.20　（实例研究）11.15 节的软件项目管理计划是针对由三个软件工程师组成的小软件工程公司的，修改这个计划，使它能够适用于具有 1000 名软件工程师的中等规模的公司。

11.21　（实例研究）如果 Osbert Oglesby 产品必须在 8 周内完成，11.15 节的软件项目管理计划将以什么方式修改？

11.22　（软件工程读物）你的导师会分发［Sobel and Clarkson，2002］的复印件，该文章将以什么方式影响你对形式化技术优缺点的看法。

## 参考文献


[Abrial, 1980] J.-R. ABRIAL, "The Specification Language Z: Syntax and Semantics," Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, April 1980.

[Alford, 1985] M. ALFORD, "SREM at the Age of Eight; The Distributed Computing Design System," *IEEE Computer* **18** (April 1985), pp. 36–46.

[Balzer, 1985] R. BALZER, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering* **SE-11** (November 1985), pp. 1257–68.

[Banks, Carson, Nelson, and Nichol, 2001] J. BANKS, J. S. CARSON, B. L. NELSON, AND D. M. NICHOL, *Discrete-Event System Simulation,* 3rd ed., Prentice-Hall, Upper Saddle River, NJ, 2001.

[Bowen and Hinchey, 1995a] J. P. BOWEN AND M. G. HINCHEY, "Ten Commandments of Formal Methods," *IEEE Computer* **28** (April 1995), pp. 56–63.

[Bowen and Hinchey, 1995b] J. P. BOWEN AND M. G. HINCHEY, "Seven More Myths of Formal Methods," *IEEE Software* **12** (July 1995), pp. 34–41.

[Brady, 1977] J. M. BRADY, *The Theory of Computer Science,* Chapman and Hall, London, 1977.

[Bruno and Marchetto, 1986] G. BRUNO AND G. MARCHETTO, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEE Transactions on Software Engineering* **SE-12** (February 1986), pp. 346–57.

[Chen, 1976] P. CHEN, "The Entity-Relationship Model—Towards a Unified View of Data," *ACM Transactions on Database Systems* **1** (March 1976), pp. 9–36.

[Coolahan and Roussopoulos, 1983] J. E. COOLAHAN, JR., AND N. ROUSSOPOULOS, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Transactions on Software Engineering* **SE-9** (September 1983), pp. 603–16.

[Dart, Ellison, Feiler, and Habermann, 1987] S. A. DART, R. J. ELLISON, P. H. FEILER, AND A. N.

HABERMANN, "Software Development Environments," *IEEE Computer* **20** (November 1987), pp. 18–28.

[Delisle and Garlan, 1990] N. DELISLE AND D. GARLAN, "A Formal Description of an Oscilloscope," *IEEE Software* **7** (September 1990), pp. 29–36.

[Delisle and Schwartz, 1987] N. DELISLE AND M. SCHWARTZ, "A Programming Environment for CSP," Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *ACM SIGPLAN Notices* **22** (January 1987), pp. 34–41.

[DeMarco, 1978] T. DEMARCO, *Structured Analysis and System Specification,* Yourdon Press, New York, 1978.

[Diller, 1994] A. DILLER, *Z: An Introduction to Formal Methods,* 2nd ed., John Wiley and Sons, Chichester, UK, 1994.

[Doolan, 1992] E. P. DOOLAN, "Experience with Fagan's Inspection Method," *Software—Practice and Experience* **22** (February 1992), pp. 173–82.

[Fagan, 1976] M. E. FAGAN, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* **15** (No. 3, 1976), pp. 182–211.

[Finney, 1996] K. FINNEY, "Mathematical Notation in Formal Specification: Too Difficult for the Masses?" *IEEE Transactions on Software Engineering* **22** (1996), pp. 158–59.

[Fraser and Vaishnavi, 1997] M. D. FRASER AND V. K. VAISHNAVI, "A Formal Specifications Maturity Model," *Communications of the ACM* **40** (May 1997), pp. 95–103.

[Gane and Sarsen, 1979] C. GANE AND T. SARSEN, *Structured Systems Analysis: Tools and Techniques,* Prentice Hall, Englewood Cliffs, NJ, 1979.

[Ghezzi and Mandrioli, 1987] C. GHEZZI AND D. MANDRIOLI, "On Eclecticism in Specifications: A Case Study Centered around Petri Nets," *Proceedings of the Fourth International Workshop on Software Specification and Design,* Monterey, CA, 1987, pp. 216–24.

[Goodenough and Gerhart, 1975] J. B. GOODENOUGH AND S. L. GERHART, "Toward a Theory of Test Data Selection," *Proceedings of the Third International Conference on Reliable Software,* Los Angeles, 1975, pp. 493–510; also published in *IEEE Transactions on Software Engineering* **SE-1** (June 1975), pp. 156–73. Revised version: J. B. Goodenough, and S. L. Gerhart, "Toward a Theory of Test Data Selection: Data Selection Criteria," in: *Current Trends in Programming Methodology,* Vol. 2, R. T. Yeh (Editor), Prentice-Hall, Englewood Cliffs, NJ, 1977, pp. 44–79.

[Gordon, 1979] M. J. C. GORDON, *The Denotational Description of Programming Languages: An Introduction,* Springer-Verlag, New York, 1979.

[Guha, Lang, and Bassiouni, 1987] R. K. GUHA, S. D. LANG, AND M. BASSIOUNI, "Software Specification and Design Using Petri Nets," *Proceedings of the Fourth International Workshop on Software Specification and Design,* Monterey, CA, April 1987, pp. 225–30.

[Hall, 1990] A. HALL, "Seven Myths of Formal Methods," *IEEE Software* **7** (September 1990), pp. 11–19.

[Hall and Chapman, 2002] A. HALL AND R. CHAPMAN, "Correctness by Construction: Developing a Commercial Secure System," *IEEE Software* **19** (January/February 2002), pp. 18–25.

[Harel and Gery, 1997] D. HAREL AND E. GERY, "Executable Object Modeling with Statecharts," *IEEE Computer* **30** (July 1997), pp. 31–42.

[Harel et al., 1990] D. HAREL, H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A. SHTULL-TRAURING, AND M. TRAKHTENBROT, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering* **16** (April 1990), pp. 403–14.

[Haxthausen and Peleska, 2000] A. E. HAXTHAUSEN AND J. PELESKA, "Formal Development and Verification of a Distributed Railway Control System," *IEEE Transactions on Software Engineering* **26** (August 2000), pp. 687–701.

[Hoare, 1985] C. A. R. HOARE, *Communicating Sequential Processes,* Prentice-Hall International,

Englewood Cliffs, NJ, 1985.

[IWSSD, 1986] Call for Papers, Fourth International Workshop on Software Specification and Design, *ACM SIGSOFT Software Engineering Notes* **11** (April 1986), pp. 94–96.

[Jackson, 1975] M. A. JACKSON, *Principles of Program Design*, Academic Press, New York, 1975.

[Jones, 1986b] C. B. JONES, *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Kampen, 1987] G. R. KAMPEN, "An Eclectic Approach to Specification," *Proceeding of the Fourth International Workshop on Software Specification and Design*, Monterey, CA, April 1987, pp. 178–82.

[Kleinrock and Gail, 1996] L. KLEINROCK AND R. GAIL, *Queuing Systems: Problems and Solutions*, John Wiley and Sons, New York, 1996.

[Knuth, 1968] D. E. KNUTH, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison Wesley, Reading, MA, 1968.

[Larsen, Fitzgerald, and Brookes, 1996] P. G. LARSEN, J. FITZGERALD, AND T. BROOKES, "Applying Formal Specification in Industry," *IEEE Software* **13** (May 1996), pp. 48–56.

[Leavenworth, 1970] B. LEAVENWORTH, Review #19420, *Computing Reviews* **11** (July 1970), pp. 396–97.

[London, 1971] R. L. LONDON, "Software Reliability through Proving Programs Correct," *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, Pasadena, CA, March 1971.

[Luckham and von Henke, 1985] D. C. LUCKHAM AND F. W. VON HENKE, "An Overview of Anna, a Specification Language for Ada," *IEEE Software* **2** (March 1985), pp. 9–22.

[Meyer, 1985] B. MEYER, "On Formalism in Specifications," *IEEE Software* **2** (January 1985), pp. 6–26.

[Modell, 1996] M. E. MODELL, *A Professional's Guide to Systems Analysis*, 2nd ed., McGraw-Hill, New York, 1996.

[Naur, 1964] P. NAUR, "The Design of the GIER ALGOL Compiler," in: *Annual Review in Automatic Programming*, Vol. 4, Pergamon Press, Oxford, UK, 1964, pp. 49–85.

[Naur, 1969] P. NAUR, "Programming by Action Clusters," *BIT* **9** (No. 3, 1969), pp. 250–58.

[Nix and Collins, 1988] C. J. NIX AND B. P. COLLINS, "The Use of Software Engineering, Including the Z Notation, in the Development of CICS," *Quality Assurance* **14** (September 1988), pp. 103–10.

[Oest, 1986] O. N. OEST, "VDM from Research to Practice," *Proceeding of the IFIP Congress, Information Processing '86*, 1986, pp. 527–33.

[Orr, 1981] K. ORR, *Structured Requirements Definition*, Ken Orr and Associates, Inc., Topeka, KS, 1981.

[Palshikar, 2001] G. K. PALSHIKAR, "Applying Formal Specifications to Real-World Software Development," *IEEE Software* **18** (November/December 2001), pp. 89–97.

[Peterson, 1981] J. L. PETERSON, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Petri, 1962] C. A. PETRI, "Kommunikation mit Automaten," Ph.D. Dissertation, University of Bonn, Germany, 1962. [In German.]

[Pfleeger and Hatton, 1997] S. L. PFLEEGER AND L. HATTON, "Investigating the Influence of Formal Methods," *IEEE Computer* **30** (February 1997), pp. 33–43.

[Ross, 1985] D. T. ROSS, "Applications and Extensions of SADT," *IEEE Computer* **18** (April 1985), pp. 25–34.

[Saiedian et. al., 1996] H. SAIEDIAN, J. P. BOWEN, R. W. BUTLER, D. L. DILL, R. L. GLASS, D. GRIES, A. HALL, M. G. HINCHEY, C. M. HOLLOWAY, D. JACKSON, C. B. JONES, M. J. LUTZ, D. L. PARNAS, J. RUSHBY, J. WING, AND P. ZAVE, "An Invitation to Formal Methods," *IEEE Computer* **29** (April 1996), pp. 16–30.

[Scheffer, Stone, and Rzepka, 1985] P. A. SCHEFFER, A. H. STONE, III, AND W. E. RZEPKA, "A Case Study of SREM," *IEEE Computer* **18** (April 1985), pp. 47–57.

[Schwartz and Delisle, 1987] M. D. SCHWARTZ AND N. M. DELISLE, "Specifying a Lift Control System with CSP," *Proceedings of the Fourth International Workshop on Software Specification and Design*, Monterey, CA, April 1987, pp. 21–27.

[Silberschatz, Galvin, and Gagne, 2002] A. SILBERSCHATZ, P. B. GALVIN, AND G. GAGNE, *Operating System Concepts*, 6th ed., Addison Wesley, Reading, MA, 2002.

[Sobel and Clarkson, 2002] A. E. K. SOBEL AND M. R. CLARKSON, "Formal Methods Application: An Empirical Tale of Software Development," *IEEE Transactions on Software Engineering* **28** (March 2002), pp. 308–20.

[Spivey, 1990] J. M. SPIVEY, "Specifying a Real-Time Kernel," *IEEE Software* **7** (September 1990), pp. 21–28.

[Spivey, 2001] J. M. SPIVEY, *The Z Notation: A Reference Manual*, 3rd ed., spivey.oriel.ox.ac.uk/~mike/zrm/, 2001.

[Teichroew and Hershey, 1977] D. TEICHROEW AND E. A. HERSHEY, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering* **SE-3** (January 1977), pp. 41–48.

[Warnier, 1976] J. D. WARNIER, *Logical Construction of Programs*, Van Nostrand Reinhold, New York, 1976.

[Wing, 1990] J. WING, "A Specifier's Introduction to Formal Methods," *IEEE Computer* **23** (September 1990), pp. 8–24.

[Woodcock, 1989] J. WOODCOCK, "Calculating Properties of Z Specifications," *ACM SIGSOFT Software Engineering Notes* **14** (July 1989), pp. 43–54.

[Yourdon and Constantine, 1979] E. YOURDON AND L. L. CONSTANTINE, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.

# 第 12 章　面向对象分析

**学习目标**

学完本章之后，你应当能够：

- 完成分析流；
- 抽象边界类、控制类和实体类；
- 完成功能建模；
- 完成类建模；
- 完成动态建模；
- 完成用例的实现。

第 11 章中讨论了各种传统的分析技术，本章是第 11 章的对应部分——面向对象的分析。

**面向对象分析**（object-oriented analysis，OOA）是面向对象范型的半形式化分析技术。在第 11 章中我们指出，结构化系统分析使用了一些不同的技术，但它们本质上是相当的。类似地，OOA 领域里已提出了超过 60 种不同的技术，这些技术很大程度上是相似的。本章的“进一步阅读”一节包含了各种技术的参考文献，以及出版的有关不同技术的比较。

然而，如 3.1 节所解释的，今天的统一过程（Unified Process）[Jacobson，Booch，and Rumbaugh，1999] 几乎已经成为面向对象的软件产品必选的方法论。因此，本章的第一节和最后一节研究统一过程的分析流。

面向对象分析是面向对象范型的关键部分，完成了这个工作流，即开始抽象类，用例和类是开发面向对象软件产品的基础。（要进一步深入了解面向对象范型，参见下面的“如果你想知道”部分。）

**如果你想知道**

面向对象范型的主要进展发生在 1990 年到 1995 年之间，因为新技术得到广泛接受通常需要 15 年，因此面向对象范型得到广泛接受至少要等到 2005 年。然而**千年虫**或 **Y2K 问题**改变了预计的时间表。

在 20 世纪 60 年代，当计算机在商业上开始被广泛使用时，硬件与今天相比要昂贵得多，结果那个时期的大多数软件产品只使用年份的后两个数字，前面的两个数字 19 是不言而喻的，这带来的问题是 00 年被解释为 1900 年，而不是 2000 年。

到 20 世纪 70 年代和 80 年代时，硬件变得更便宜了，很少有管理者愿意花费大量金钱为四位年份问题重写现有的软件产品，毕竟在 2000 年到来时，它将是别人的问题，结果**遗留系统**不适应 2000 年。然而随着最后期限 2000 年 1 月 1 日的到来，软件公司被迫争分夺秒地修复它们的软件产品，因为谁也不能推迟 Y2K 的到来。

维护程序员面临的问题包括缺乏许多遗留软件产品的文档，还有软件产品是由现在已过时的编程语言编写的。当不可能调整现有的软件产品时，惟一的选择是从头再开始。一些公司决定使用 COTS 技术（见 1.11 节），其他一些公司认为需要新的定制软件产品。很明显，管理者想要使用现代技术来开发这些软件产品，这些技术已经显示出高的性能价格比，即使用面向对象范型。因此 Y2K 问题成为面向对象范型得到广泛接受的最好的催化剂。

## 12.1　分析流

统一过程的**分析流**有两个目标。从需求流（前述的工作流）的角度看，分析流的目标是得到对需求更深的理解；相反地，从设计和实现流（分析流之后的工作流程）的角度看，分析流的目标是按设计和实现易于维护的思路描述那些需求。

统一过程是用例驱动的，在分析流期间，用例以软件产品的类来描述。统一过程有三种类：实体类、边界类和控制类。**实体类**为长期存在的信息建模，在银行软件产品用例中，**Account Class**（账户类）是一个实体类，因为账户信息需要保存在软件产品中。对于Osbert Oglesby 软件产品，**Painting Class**（画作类）是实体类，也是因为画作信息是长期存在的。

**边界类**为软件产品和它的操作者之间的交互行为建模。边界类通常与输入和输出相关。例如，在 Osbert Oglesby 软件产品中，需要打印前一年中买进和售出的画作报表，这意味着边界类 **Purchases Report Class**（购买报表类）和 **Sales Report Class**（售出报表类）是必需的。

**控制类**为复杂的计算和算法建模。在 Osbert Oglesby 软件产品中，计算某类画作价格的算法是控制类，类名为：**Compute Masterpiece Price Class**（计算杰作价格类）、**Compute Masterwork Price Class**（计算大作价格类）和 **Compute Other Painting Price Class**（计算其他画作价格类）。

这三种类的 UML 符号如图 12-1 所示，它们是**固定形式**，即 UML 扩展。UML 的优点是它允许定义额外的结构，该结构不是 UML 的一部分，但却是准确地为特定系统建立模型所必需的。




图 12-1　代表实体类、边界类和控制类的 UML 固定形式（UML 扩展）

如本节开始时所讲，在分析流，用例以软件产品的类来描述。统一过程本身不描述如何抽象类，因为统一过程的用户在面向对象的分析和设计方面有一定的背景知识，因此统一过程的这个讨论暂时推后，以便说明如何抽象类，我们将在 12.15 节再回来讨论统一过程。

下面首先讨论为长期存在的信息建模的类——实体类。

## 12.2　抽象实体类

实体类抽象包括三个迭代和递增地完成的步骤：

1. **功能建模**。提出所有用例的情景（**情景**是用例的一个实例）。

2. **实体类建模**。确定实体类和它们的属性，然后确定实体类之间的交互关系和交互行为，以类图的形式提供这个信息。

3. **动态建模**。确定每个实体类或子类执行的操作或对它们的操作，以状态表的形式提供这个信息。

然而，对于所有迭代和递增的处理，这三个步骤不总是以这个顺序进行，一个模型中的变化通常引发其他两个模型对应的修订。

为显示这是如何做的，我们现在抽象电梯问题实例研究的实体类。

## 12.3　面向对象分析：电梯问题实例研究

第 11 章描述了电梯问题实例研究，为便于参考，这里再将该问题重复一遍。

为控制m层楼房里的n个电梯，安装一个软件产品，这个问题关注在楼层间按照下列的约束移动电梯所要求的逻辑：

1）每个电梯有一组m个按钮，每层对应一个。当按下按钮时按钮灯亮，并让电梯向相应的层移动。当电梯降临相应层时按钮灯灭。

2）除了第一层和顶层，每层有两个按钮，一个请求电梯向上，一个请求电梯向下。当按下时，这些按钮灯亮，当一个电梯降临该层时，灯灭，然后向想要去的方向移动。

3）当没有对电梯提出请求时，它仍停留在当前的楼层，门关着。

OOA中的第一步是用例建模。

## 12.4　功能建模：电梯问题实例研究

用例描述了要建造的产品和它的操作者（即产品的外部用户）之间的交互行为，用户和电梯之间惟一可能的交互行为是用户按下电梯按钮调动电梯，或者用户按下楼层按钮要求电梯停在某一层，因而有两个用例：Press an Elevator和Press a Floor Button。这两个用例如图12.2的用例图（10.7节）所示。

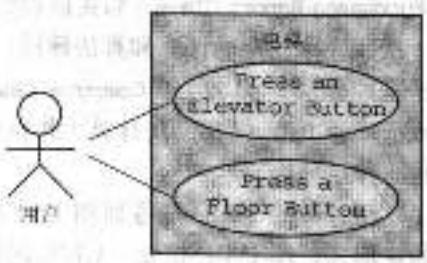



图12-2　电梯问题实例研究的用例图

用例提供了整个功能的一般描述，情景是用例的一个特定实例，就像对象是类的一个实例。通常情况下，有大量的情景，每个代表特定的交互行为组。本节中我们考虑图12-3所示的情景，该情景合并了两个用例的实例。

1. 用户A按下3层的向上楼层按钮，希望电梯去7层。
2. 向上的楼层按钮灯亮。
3. 电梯到达3层，用户B在电梯内，用户D从1层进入电梯，按下到9层的电梯按钮。
4. 向上的楼层按钮灯灭。
5. 电梯门开启。
6. 定时器开始计时，用户A进入电梯。
7. 用户A按下到7层的电梯按钮。
8. 7层的电梯按钮灯亮。
9. 定时时间到后电梯门关闭。
10. 电梯到达7层。
11. 7层的电梯按钮灯灭。
12. 电梯门开启以便用户A出电梯。
13. 定时器开始计时，用户A走出电梯。
14. 定时时间到后电梯门关闭。
15. 电梯载着用户B向9层移动。

图12-3　正常情景的第一次迭代

图12-3描述了一个**正常情景**，即用户和电梯之间的一组按正常想法使用电梯的交互行为。图12-3是经过仔细观察不同的用户与电梯（或更准确地说，与电梯按钮和楼层按钮）的交互后得到的，这15项事件详细描述了用户A和电梯系统的按钮之间的两个交互行为（事件1和事件7），以及电梯系统的组件完成的操作（事件2到事件6和事件8到事件15）。

两个事项——User A enters the elevator（用户 A 进入电梯）和 User A exits from the elevator（用户 A 走出电梯）没有单独列项，这样的事项其实是注释，用户 A 进入或走出电梯时不与电梯的组件进行交互。

相反，图 12-4 是一个**异常情景**，它描述了当用户在 3 层按下了向上按钮，但实际上想下到 1 层时发生的情况。这个情景也是通过观察许多用户在电梯里的行为构建的，从未用过电梯的人可能认为用户有时会按下错误的按钮，但这不太可能。

图 12-3 和图 12-4 的情景，加上其他无数的情景都是图 12-2 所示的用例的特定实例。为给 OOA 小组提供要建模系统特性的深入理解，应该充分研究情景，在下一步骤实体类建模中，会使用这个信息确定实体类。

1. 用户 A 按下 3 层的向上楼层按钮，希望电梯去 1 层。
2. 向上的楼层按钮灯亮。
3. 电梯到达 3 层，用户 B 在电梯内，用户 B 从 1 层进入电梯，按下到 9 层的电梯按钮。
4. 向上的楼层按钮灯灭。
5. 电梯门开启。
6. 定时器开始计时，用户 A 进入电梯。
7. 用户 A 按下到 1 层的电梯按钮。
8. 1 层的电梯按钮灯亮。
9. 定时时间到后电梯门关闭。
10. 电梯到达 9 层。
11. 9 层的电梯按钮灯灭。
12. 电梯门开启以便用户 B 出电梯。
13. 定时器开始计时，用户 B 走出电梯。
14. 定时时间到后电梯门关闭。
15. 电梯载着用户 A 向 1 层移动。

图 12-4　一个异常情景

## 12.5　实体类建模：电梯问题实例研究

在这个步骤中，抽象实体类和它们的属性并用一个 UML 类图表示。这时只确定实体类的属性，不确定方法，后者在面向对象设计（object-oriented design，OOD）流期间分配给这些实体类。

整个面向对象范型的特点是，各个步骤都不太易于实现。所幸的是，使用对象带来的好处值得付出这种努力，因此，分析流的第一部分——抽象实体类和它们的属性通常很难一次做好就不足为怪了。

确定实体类的一个方法是从用例推断它们，即开发者仔细研究全部情景，包括正常的和异常的，并且分辨在用例中发挥作用的组件。仅从图 12-3 和图 12-4 的情景中可以推断，候选的实体类有电梯按钮、楼层按钮、电梯、门和定时器，如我们所将要看到的，这些候选实体类与在实体类建模期间抽象的实际类相近。然而，通常有许多情景，而且随之有大量潜在的类。一个缺乏经验的开发者可能试图从情景中推断出太多的候选实体类，这对实体类建模会造成不利的影响，因为加入一个新实体类比移去一个不应当包含在内的候选实体类更容易。

当开发者具有某一领域的专门知识时，另一个确定有效实体类的方法是 CRC 卡片（见

12.5.2 节)。然而，如果开发者在应用领域内具有很少或没有经验时，那么建议使用下一节描述的名词抽象。

### 12.5.1 名词抽象

对于没有专业知识的开发者，一个继续下去的好办法是使用下面的两阶段**名词抽象法**抽象候选实体类，然后求精解决方案：

**阶段 1　用一个段落描述软件产品。**对于电梯问题实例研究，一种可能的方式如下所述：

电梯中和楼层上的按钮控制在 m 层楼中的 n 个电梯的移动。按钮被按下时灯亮请求电梯到达某一层；当请求被满足时，按钮灯亮被取消（即灯灭）。当电梯没有请求时，电梯保持在当前层，门关着。

**阶段 2　分辨名词。**分辨非形式化策略中的名词（不包含问题边界以外的那些名词），然后使用这些名词作为候选实体类。现在重新描述那个非形式化策略，但这一次分辨出的名词以黑体字印刷。

**电梯**中和**楼层**上的**按钮**控制在 m 层楼中的 n 个**电梯**的**移动**。**按钮**被按下时灯亮请求**电梯**到达某一层；当**请求**被满足时，按钮**灯亮**被取消（即灯灭）。当**电梯**没有**请求**时，电梯保持在当前层，门关着。

这里有 8 个不同的名词：**按钮**、**电梯**、**楼层**、**移动**、**楼房**、**灯亮**、**请求**和**门**。这些名词中的 3 个——**楼层**、**楼房**和**门**在问题边界之外，因此可以忽略。剩下名词中的 3 个——**移动**、**灯亮**和**请求**是**抽象名词**，即它们识别没有物理存在的事物。一个有用的经验性方法是，抽象名词很少与类一起结束。相反，它们经常是类的属性。例如，**灯亮**是**按钮**的一个**属性**。这样，剩下两个名词，因此也是两个候选类：**Elevator**（电梯）和 **Button**（按钮）。（UML 的约定是对类名称使用粗体且类名称的首字母大写。）

得到的类图如图 12-5 所示。类 **Button**（按钮）具有布尔属性 illuminated（灯亮），用来模拟图 12-3 和图 12-4 情景的模型事件 2、4、8 和 11。该问题规定了两种类型的按钮，因此定义了两个 **Button** 子类——**Elavator Button**（电梯按钮）和 **Floor Button**（楼层按钮）（空三角在 UML 中表示继承）。每个 **Elavator Button** 和 **Floor Button** 与 **Elevator**（电梯）通信，后一个类——**Elevator** 具有布尔属性 doors open（门开），用来模拟这两个情景的事件 5、9、12 和 14。

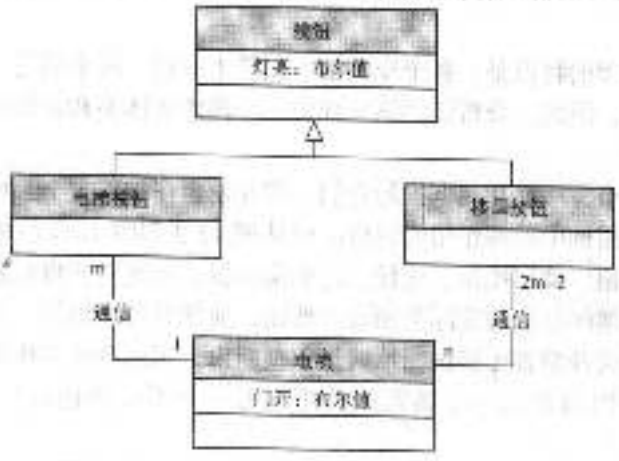



图 12-5　电梯问题实例研究中类图的第一次迭代

遗憾的是，这不是一个好的开始。在实际的电梯中，按钮不直接与电梯通信，如果仅决定分派哪部电梯来响应某个请求的话，则需要某种电梯控制器。然而，问题的陈述并没有提到控制器，因此，在名词抽象过程中没有把它选为一个实体类。换句话说，本节寻找候选实体类的技术提供了一个起始点，但是当然不能依靠它做过多的事情。

在图 12-5 中加入 **Elevator Controller**（电梯控制器）就得到图 12-6，这当然更有意义。进一步地说，在图 12-6 中现在有了一个一对多的关系，与图 12-5 中难于建模的多对多关系相反。因此看来有理由从这一点继续进行到步骤 3，记住，随时可能返回到实体类建模，即使是在实现流中也一样。然而，在进行动态建模之前，考虑一个不同的实体类建模技术。

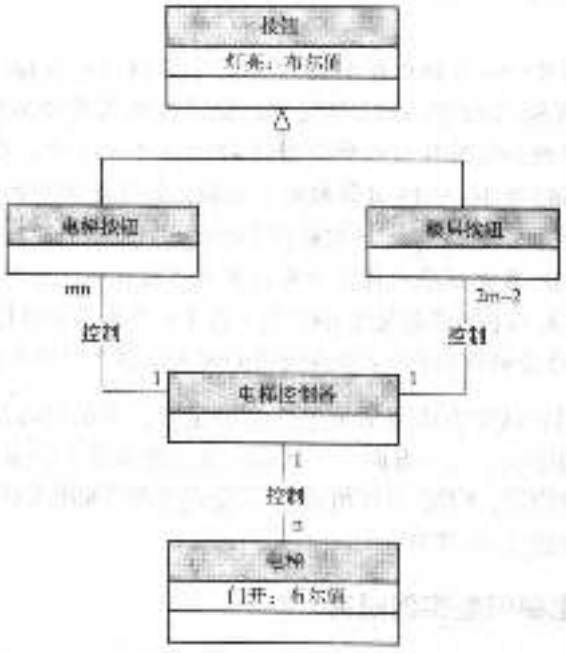



图 12-6　电梯问题实例研究中类图的第二次迭代

### 12.5.2　CRC 卡片

多年来，在面向对象分析流中一直使用**类职责协作**（class-responsibility-collaboration，CRC）卡片［Wirfs-Brock，Wilkerson，and Wiener，1990］。对于每个类，软件开发小组填写一个卡片，卡片上有类的名称、它的功能（职责）和它达成功能要调用的其他类的列表（协作）。

这个方法随后扩展了。首先，一个 CRC 卡片常常明确地含有类的属性和方法，而不仅仅是用某种自然语言描述的它的“职责”；其次，这项技术变化了，不是使用卡片，某些组织将类的名称放到 Post-it 便笺上，他们在一个白板上随处移动它，在 Post-it 便笺间画线表示协作。现在，整个过程都可以自动完成，像 System Architect 这样的 CASE 工具包含在屏幕上创建和更新 CRC“卡片”的模块。

CRC 卡片的优点是，当一个小组利用它时，在小组成员之间的交互可以突出显示一个类中遗漏的或不正确的字段，不管是属性还是方法。而且，当使用 CRC 卡片时，类之间的关系也得到了明确。一个特别强有力的技术是在小组成员之间分发卡片，然后小组成员将它们的类的职责付诸实施。这样，有些人可能会说，“我是 **Date** 类，我的职责是创建新的日期

对象。”另一个小组成员可能插嘴说他或她需要从 **Date** 类得到附加的功能，如将一个日期从常规格式转换成为一个整数——从 1900 年 1 月 1 日算起的天数，以便可以通过减去相应的两个整数得到任意两个日期间的天数（参见下面的“如果你想知道”部分）。因此，将 CRC 卡片的职责付诸实施是一个验证类图的完整性和正确性的有效方法。

**如果你想知道**

我们怎样算出在 1999 年 2 月 21 日和 2005 年 8 月 10 日之间的天数呢？在许多财务计算中需要做这样的减法，如计算偿还的利息或者确定未来资金周转的当前值。通常的做法是将每一天转换成为一个整数——从某一起始日期算起的天数，问题是我们不能就使用什么起始日期达成一致。

宇航员使用 Julian 日——从公元前 4713 年 1 月 1 日格林尼治标准时间正午算起的日期数，这个系统由 Joseph Scaliger 在 1582 年发明，他以他的父亲的名字 Julius Caesar Scaliger 为其命名（如果你实在想知道为什么选择公元前 4713 年 1 月 1 日，参见 [USNO, 2000]）。

Lilian 日期是从 1582 年 10 月 15 日算起的日期数，该日是罗马教皇历法的第一日，它是由教皇 Gregory 十三世引入的。Lilian 日期是以 Luigi Lilio，一个罗马教皇历法改革的主要倡导者的名字命名的。Lilio 负责推导出许多罗马教皇历法的算法，包括闰年的规则。

在软件方面，COBOL 内在的函数使用 1601 年 1 月 1 日作为整数日期的起始日期。然而，紧随 Lotus 1-2-3 其后，几乎全部的电子制表软件使用 1900 年 1 月 1 日作为整数日期的起始日期。

CRC 卡片的缺点是，这个方法通常不是一个确定实体类的好办法，除非小组成员在相关的应用领域有相当的经验。另一方面，一旦开发者已经确定了许多类，并且对于它们的职责和协作有了比较好的想法，CRC 卡片可以是完成该过程并确保每件事情是正确的一个绝好的方法，这方面的描述见 12.7 节。

## 12.6　动态建模：电梯问题实例研究

动态建模的目标是生成每个类的**状态图**——与有穷状态机相类似的对目标产品的描述。首先来看看 **Elevator Controller**（电梯控制器）类，为简单起见，仅考虑一个电梯，对应的 **Elevator Controller** 类的状态图如图 12-7 所示。

该表示法有些类似于 11.7 节的有穷状态机（FSM），但是有一个明显的不同。第 11 章中给出的 FSM 是形式化技术的一个例子，状态转换图本身并不能完全代表要建造的产品，相反，该模型由一套具有式（11.2）给出的形式的转换规则组成：

**当前状态　与　事件　与　谓词⇒下一个状态**

形式化是通过以一套数学规则的形式给出模型来获得的。

与此相反，一个 UML 状态图的表示有些不那么形式化，一个状态机的三个方面（状态、事件和谓词）分布在 UML 图中。例如，如果当前状态是 **Elevator Event Loop**（电梯事件循环），并且事件“电梯停止，无请求挂起”为真，那么进入图 12-7 中的状态 **Going into Wait State**（进入等待状态）。当进入状态 **Going into Wait State**（进入等待状态）后，将要执行操作“定时时间到后电梯门关”。当前的 OOA 版本是半形式化的（图形）技术，相应地，状态图所固有的对形式化的缺乏不是问题。然而，当面向对象范型成熟了，可能会开发出更形式化的版本，因而对应的动态模型将会在某种程度上更接近有穷状态机。

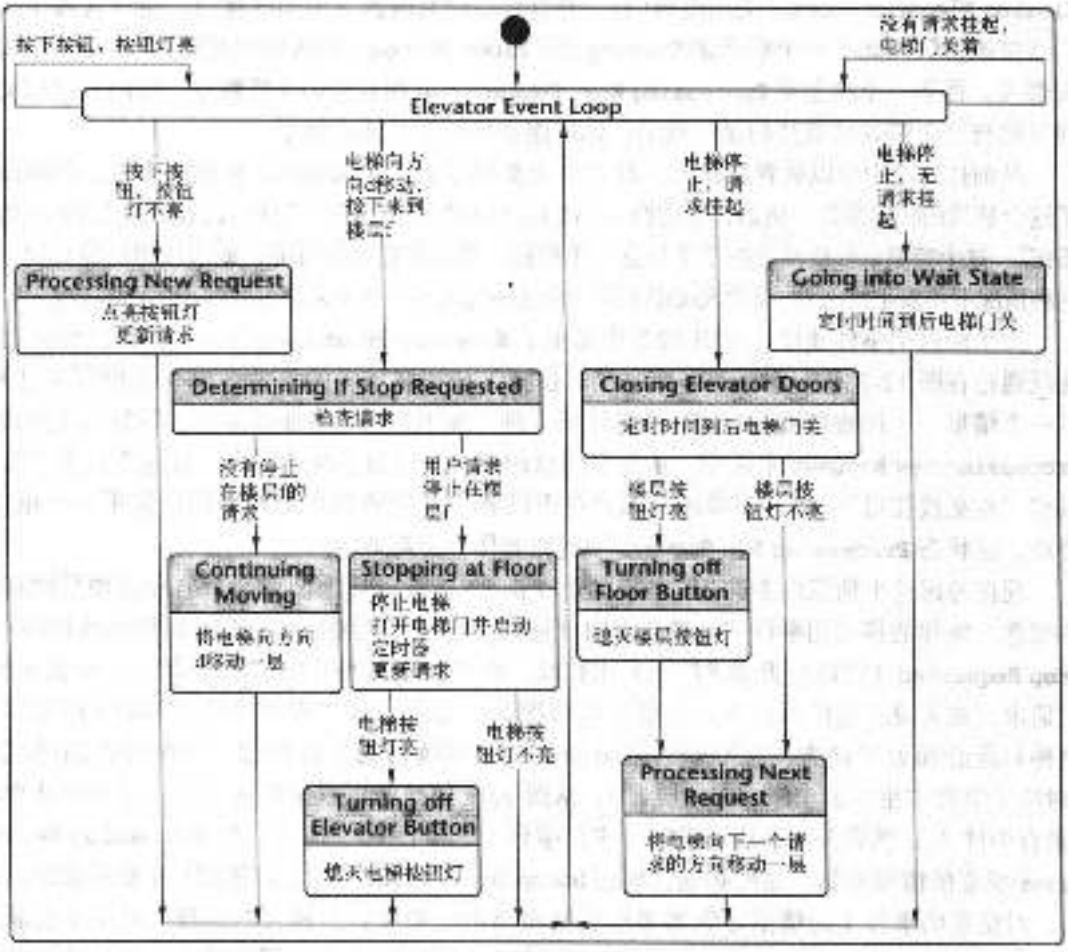



图 12-7　Elevator Controller 类状态图的第一次迭代

为了明白图 12-7 的状态图与图 11-13 至图 11-15 的 STD 的等效性，我们来看看各种情景。例如，考虑图 12-3 的情景的第一部分。首先，用户 A 在楼层 3 按下向上楼层按钮，如果该楼层按钮未点亮，那么，按照图 11-14 和图 12-7 的状态 **Processing New Request**（处理新请求），该按钮按要求被点亮。在状态图的情形中，下一个状态是 **Elevator Event Loop**。

接下来，电梯接近 3 楼，首先，考虑 STD 方法。在图 11-15 中，电梯进入状态 s（u，3），即它停在 3 楼，将要上升（因为已做出简化的假设，认为只有一部电梯，图 11-15 中的参数 e 在这里忽略）。从图 11-14 中看出，当电梯到达时，楼层按钮灯熄灭，现在（图 11-15）电梯门关闭，电梯开始向 4 楼移动。

回到图 12-7 的状态图，看看当电梯接近 3 楼时发生了什么。因为电梯在运动中，输入的下一个状态是 **Determining if Stop Requested**（确定是否请求停止）。检查该请求，并且因为用户 A 已请求电梯停在那里，下一个状态是 **Stopping at Floor**（停止在楼层）。电梯停在 3 楼，电梯门开。由于未曾按下 3 楼的电梯按钮，因此下一个状态是 **Elevator Event Loop**。

用户 A 进入并按下到 7 楼的电梯按钮，因此，下一个状态又是 **Processing New Request**，随后又是 **Elevator Event Loop**。电梯曾停下并且挂起 2 个请求，因此下一个状态是

**Closing Elevator Doors**（关闭电梯门），并且在定时时间到后关闭电梯门。用户 A 按下 3 楼的楼层按钮，因此下一个状态是 **Turning off Floor Button**（熄灭楼层按钮灯），楼层按钮灯被熄灭。再下一个状态是 **Processing Next Request**，电梯开始向 4 楼移动。对应图的相关方面显然与这个情景所期待的是一致的，你可能还想看看其他的情景。

从前述讨论中可以很容易看出，图 12-7 是根据这个情景构建的。更准确地说，归纳概括了这个情景的特定事件。例如，考虑图 12-3 的情景的第一个事件“用户 A 按下 3 层的向上楼层按钮”。这个特定事件概括为按下了任意一个按钮，那么存在两种可能，或者按钮已经亮起（在这种情况下不发生什么），或者该按钮不亮（在这种情况下，必须采取行动处理用户的请求）。

为了给这个事件建模，在图 12-7 中画出了 **Elevator Event Loop** 状态。按钮已经亮起的情况通过在图 12-7 左上角带有事件“按下按钮，按钮灯亮”的不做任何事的循环来建模。另一个情形——按钮灯不亮，用一个标注有事件“按下按钮，按钮灯不亮”的箭头指向状态 **Processing New Request** 来建模。从这个情景的事件 2 可以清楚地看出，在这个状态下需要操作“点亮按钮灯”。更进一步地，用户按下任意一个按钮的行为的意图是请求一个电梯，因此，在状态 **Processing New Request** 中进行操作“更新请求”。

现在考虑这个情景的事件 3“电梯到达 3 楼”，这被概括为任意一个电梯在楼层间移动的概念。电梯的移动用事件“电梯向方向 d 移动，接下来到楼层 f”和状态 **Determining if Stop Requested**（确定是否请求停止）来建模，但仍存在两种可能：或者有一个停在楼层 f 的请求，或者没有这样的请求。在前者的情况下，对应事件“没有停止在楼层 f 的请求”，电梯只是必须处于向方向 d **Continuing Moving**（继续移动）的状态。而在后者的情况下（对应于事件“用户请求停止在楼层 f”），从图 12-3 的情景可以清楚地看出，必须停止电梯（来自事件 3），然后开门并启动定时（来自事件 5 和事件 6）。而且，与 **Processing New Request** 状态的情形相似，显然必须在状态 **Stopping at Floor**（停止在楼层）中更新请求。最后，对情景的事件 4 的概括导致如果点亮则需要将电梯按钮灯熄灭的实现。这是通过状态 **Turning off Elevator Button**（熄灭电梯按钮灯）（在图 12-7 左下部的状态）和代表该状态的方框上的两个事件来建模的。

对图 12-3 的情景的事件 9 的概括生成了状态 **Closing Elevator Doors**，对事件 10 的概括生成了状态 **Processing Next Request**。然而，对状态 **Going into Wait State** 和事件“没有请求挂起，电梯门关着”的需要，是通过归纳概括另一个不同的情景的一个事件得出的，在该情景的那个事件中，用户离开电梯并且没有按钮灯亮着。

其他类的状态图相对容易理解，因此留做练习（习题 12.1）。

## 12.7　测试流：面向对象分析

现在看来已经完成了功能、实体类和动态模型，**测试流**开始了。面向对象分析过程的三个模型似乎已经完成，下一个步是评审到现在为止的分析流，这个评审的一个组成部分，如 12.5.2 节所建议的，是使用 CRC 卡片。

因此，对下面的每个实体类填写 CRC 卡片：**Button**、**Elevator Button**、**Floor Button**、**Elevator** 和 **Elevator Controller**。**Elevator Controller** 的 CRC 卡片（如图 12-8 所示），是从图 12-6 的类图和图 12-7 的状态图中演绎出来的。更详细地，**Elevator Controller** 的 RESPONSIBILITY（职责）是通过列出 **Elevator Controller** 状态图中的所有操作得出的（图 12-7），**Ele-**

vator Controller 的 COLLABORATION（协作）是通过检查图 12-6 的类图并注意到类 Elevator Button、Floor Button 和 Elevator 与类 Elevator Controller 的相互作用而确定的。

这个 CRC 卡片突出了面向对象分析的第一次迭代的两个主要问题。

1）考虑职责 1。开启电梯按钮，这个命令在面向对象范型中完全是不合适的，从职责驱动设计的观点来看（见 1.9 节），类 Elevator Button 的目标是负责将它们自己开启或关闭，而且，从信息隐藏的观点来看（见 7.6 节），Elevator Controller 不应当知道需要开启按钮的 Elevator Button 的内部情况。正确的职责是：发送一个消息给 Elevator Button，使它能将自己开启。对于图 12-8 中的职责 2 到 6 也需要类似的改变，这 6 个更正反映在图 12-9 中，该图是 Elevator Controller 的 CRC 卡片的第二次迭代。

| 类<br>Elevator Controller |
| --- |
| RESPONSIBILITY（职责）<br>1. 开启电梯按钮<br>2. 关闭电梯按钮<br>3. 开启楼层按钮<br>4. 关闭楼层按钮<br>5. 将电梯向上移动一层<br>6. 将电梯向下移动一层<br>7. 打开电梯门并启动定时器<br>8. 定时时间到后关闭电梯门<br>9. 检查请求<br>10. 更新请求 |
| COLLABORATION（协作）<br>1. 类 Elevator Button<br>2. 类 Floor Button<br>3. 类 Elevator |

图 12-8　类 Elevator Controller 的 CRC 卡片的第一次迭代

| 类<br>Elevator Controller |
| --- |
| RESPONSIBILITY（职责）<br>1. 给 Elevator Button 发送消息来开启电梯按钮<br>2. 给 Elevator Button 发送消息来关闭电梯按钮<br>3. 给 Floor Button 发送消息来开启楼层按钮<br>4. 给 Floor Button 发送消息来关闭楼层按钮<br>5. 给 Elevator 发送消息来将电梯向上移动一层<br>6. 给 Elevator 发送消息来将电梯向下移动一层<br>7. 给 Elevator Doors 发送消息来打开电梯门<br>8. 启动定时器<br>9. 定时时间到后给 Elevator Doors 发送消息来关闭电梯门<br>10. 检查请求<br>11. 更新请求 |
| COLLABORATION（协作）<br>1. 子类 Elevator Button<br>2. 子类 Floor Button<br>3. 类 Elevator Doors<br>4. 类 Elevator |

图 12-9　类 Elevator Controller 的 CRC 卡片的第二次迭代

2）忽略了一个类。看一下职责 7，打开电梯门并启动定时器，这里关键的概念是**状态**。一个类的属性有时被称为**状态变量**，使用这个术语的原因是，在大多数面向对象实现中，产品的状态由各种组件对象的属性的值决定。状态图与有穷状态机有许多相同的特性。因此，在面向对象范型中状态的概念扮演着重要的角色是不足为怪的。这个概念有助于确定一个组件是否应当作为一个类来建模，如果有疑问的组件拥有一个将在实现的执行期间改变的状态，那么，它很可能将被作为一个类来建模。显然，电梯的门拥有一个状态（开或关），因此 Elevator Doors 应当是一个类。

关于为什么 Elevator Doors 应当是一个类还有另外一个原因，面向对象范型允许状态隐藏在对象中，从而禁止未授权改变。如果有一个 Elevator Doors 对象，电梯的门能够打开或关闭的惟一方法是，向 Elevator Doors 对象发送一个消息。在错误的时间打开或关闭电梯门可能造成严重的意外事故，参见如下“如果你想知道”部分。因此，对于某些类型的产品，安全考虑也是对象的优势之一。

**如果你想知道**

若干年前，我在一栋建筑物的 10 层不耐烦地等电梯。电梯门开了，我开始向前迈步，

但那儿却没有电梯！救了我一命的是当我将要步入电梯升降机内时我看到的是漆黑一片，我本能地意识到出差错了。

如果该电梯控制系统是用面向对象范型开发的，在第 10 层不适当地打开电梯门的事就可能不会发生。

把 **Elevator Doors** 当成一个类意味着需要把图 12-8 中的职责 7 和 8 类似地改变为职责 1 到 6，即，应当将消息发送到 **Elevator Doors**，以便将它们自身打开或关闭。但是这样额外增加了一个复杂性，职责 7 是“打开电梯门并启动定时器”。

必须将这分成两个单独的职责。确实必须向 **Elevator Doors** 发送一个消息来开门，但是，定时器是 **Elevator Controller** 的一部分，因此启动定时器是 **Elevator Controller** 的职责。**Elevator Controller** 类的 CRC 卡片的第二次迭代（见图 12-9）显示出，这个职责的分割已顺利实现。

除了图 12-8 的 CRC 卡片强调的两个主要问题之外，Elevator Controller 的“检查请求”和“更新请求”职责要求将属性 requests（请求）加到类 **Elevator Controller** 中。在这个步骤中，简单地将 requests 定义为 requestType 的类型，requestType 是在设计流中选用的 requests 的一个数据类型。

改正的类图如图 12-10 所示，由于已经对类图做了修改，必须重新检查用例图和状态图，看看是否也需要对它们进一步求精。用例图显然仍是合适的，然而，图 12-7 的状态图中的操作必须修改，以反映图 12-9 的职责（CRC 卡片的第二次迭代）而不是图 12-8 的职责（第一次迭代）。而且，必须扩展这一套状态图以包括附加的类，需要更新情景来反映这些变化，图 12-11 显示了图 12-3 的情景的第二次迭代，即使在做出所有这些变化和检查（包括修改的 CRC 卡片）之后，在面向对象设计工作流仍有必要返回到面向对象分析工作流，并且修改一个或多个分析图。然而看起来在这个阶段，已正确地抽象了电梯问题实例研究的实体类。

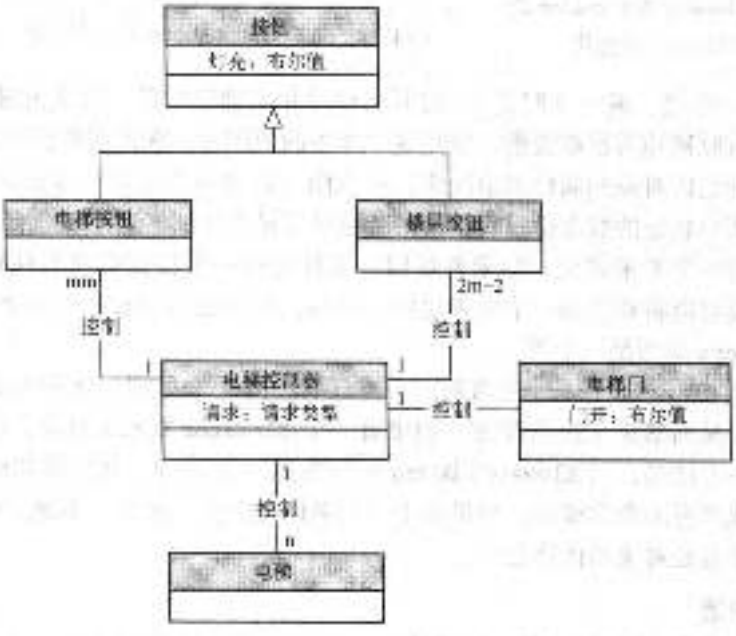



图 12-10　电梯问题实例研究的类图的第三次迭代

1. 用户 A 在 3 楼按下向上楼层按钮，请求一个电梯。用户 A 想到 7 楼。
2. 楼层按钮告知电梯控制器楼层按钮已被按下。
3. 电梯控制器给向上楼层按钮发送一个消息，让它开启自己。
4. 电梯控制器发送一串消息给电梯，使电梯自己到达 3 楼，电梯里有用户 B，他在 1 楼进入电梯并按下到 9 楼的电梯按钮。
5. 电梯控制器发送一个消息给向上楼层按钮，使它关闭自己。
6. 电梯控制器发送一个消息给电梯门，使它打开自己。
7. 电梯控制器启动定时器。用户 A 进入电梯。
8. 用户 A 按下去 7 楼的电梯按钮。
9. 电梯按钮告知电梯控制器电梯按钮已被按下。
10. 电梯控制器给到 7 层的电梯按钮发送一个消息，开启它自己。
11. 电梯控制器给电梯门发送一个消息，在定时时间到后关闭电梯门。
12. 电梯控制器给电梯发送一串消息，将它自己移向 7 楼。
13. 电梯控制器给到 7 层的电梯按钮发送一个消息，关闭它自己。
14. 电梯控制器给电梯门发送一个消息，使电梯门开启，让用户 A 从电梯中出来。
15. 电梯控制器启动定时器。用户 A 从电梯中出来。
16. 电梯控制器给电梯门发送一个消息、定时时间到后电梯门关。
17. 电梯控制器向电梯发送一串消息，使电梯载着用户 B 移向 9 楼。

图 12-11　电梯问题实例研究的一个正常情景的第二次迭代

## 12.8　抽象边界类和控制类

与实体类不同，边界类通常易于抽象。通常情况下，每个输入屏幕、输出屏幕和打印的报表由它自己的边界类建模。回想一下，类合并属性（数据）和操作，边界类建模（假设打印的报表）合并可包含在报表中的所有数据项及打印报表所涉及的各种操作。

控制类通常像边界类一样易于抽象，通常情况下，每个重要的计算由控制类进行建模。

现在我们通过抽象 Osbert Oglesby 实例研究中的类，说明实体类、边界类和控制类的抽象。

## 12.9　初始功能模型：Osbert Oglesby 实例研究

在功能建模中，确定用例的情景。回想一下，用例图描述软件产品和软件产品用户之间的交互行为。考虑 Osbert Oglesby 软件产品，尽管软件产品的惟一用户是 Osbert 自己，客户（即购买者或出售者）是两个为购买或售出画作建模的用例中的动作者。Osbert 可以用四种方式使用该软件产品，得到的四个用例在图 10-22 中显示，为方便起见，在这里重新画出它，如图 12-12 所示。

考虑用例 Buy a Painting，它描述了 Osbert、出售者和软件产品之间购买一幅画作时的交互行为。这个用例的一个可能的实例是 Osbert 使用计划中的软件产品帮助他购买一幅杰作（masterpiece）。这个实例显示在图 12-13 的情景中，即图 12-12（用例）描述了所有可能的购买交互行为，而图 12-13（情景）只描述了一个特定的购买交互行为。

在图 12-13 的情景的 6 句话中，只有 4 句话有编号，原因是两个未编号的句子与 Osbert

和软件产品之间的交互行为无关，主要是注释。

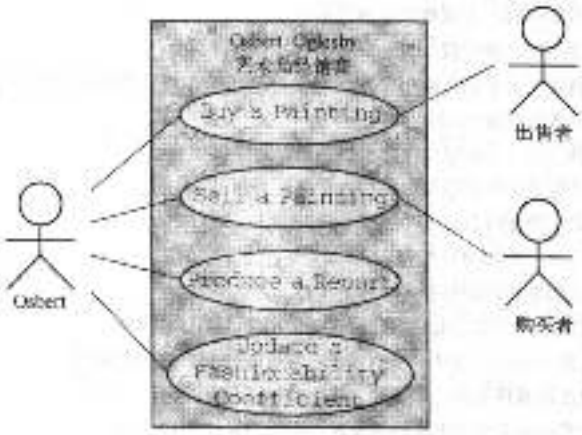



图 12-12　Osbert Oglesby 实例研究的用例图的第二次迭代

Osbert Oglesby 希望购买一幅杰作。
1. Osbert Oglesby 输入画作的描述。
2. 软件产品扫描拍卖记录，找寻同一画家最类似作品的售出价格和售出时间。
3. 软件产品计算最高购买价格，从最相似作品的拍卖价算起，每年增加 8.5%。
Osbert 开出一个价，低于最高购买价格，出售者接受这个开价。
4. Osbert 输入销售信息（出售者的名字和地址，购买价格）。

图 12-13　购买一幅杰作的一个可能的情景

图 12-14 和图 12-15 显示了购买一幅杰作的另两个情景。在图 12-14 的情景中，出售者拒绝 Osbert 的开价，而在图 12-15 的情景中，软件产品在拍卖文件中找不到该画家的类似画作。图 12-13 代表正常情景，而图 12-14 和图 12-15 代表异常情景。

Osbert Oglesby 希望购买一幅杰作。
1. Osbert Oglesby 输入画作的描述。
2. 软件产品扫描拍卖记录，找寻同一画家最类似作品的售出价格和售出时间。
3. 软件产品计算最高购买价格，从最相似作品的拍卖价算起，每年增加 8.5%。
Osbert 开出一个价，低于最高购买价格，出售者拒绝这个开价。

图 12-14　购买一幅杰作的第二个可能的情景

Osbert Oglesby 希望购买一幅杰作。
1. Osbert Oglesby 输入画作的描述。
2. 软件产品扫描拍卖记录，找寻同一画家最类似作品的售出价格和售出时间。
3. 软件产品报告说没有类似的作品。
Osbert 没有给该画作开价。

图 12-15　购买一幅杰作的第三个可能的情景

我们已经看到，用例可合并成一个用例图，类似地，正常和异常情景也可合并成一个扩展情景，如图 12-16 所示。

Osbert Oglesby 希望购买一幅杰作。
1. Osbert Oglesby 输入画作的描述。
2. 软件产品扫描拍卖记录，找寻同一画家最类似作品的售出价格和售出时间。
3. 软件产品计算最高购买价格，从最相似作品的拍卖算起，每年增加 8.5%。
Osbert 开出一个价，低于最高购买价格，出售者接受这个开价。
4. Osbert 输入销售信息（出售者的名字和地址、购买价格）。

可能的选择：
A. 出售者拒绝 Osbert 的开价。
B. 在拍卖文件里没有该画家的类似画作，因此 Osbert 没有给该画作开价。

图 12-16　购买一幅杰作的扩展情景

本节提供了图 12-12 中用例之一的情景，其他三个用例的扩展情景作为练习题（参见习题 12.8 到 12.10）。

情景不仅可用于功能建模步骤，对动态建模步骤也很重要，如 12.14 节所述。然而我们还是先看实体类建模步骤。

## 12.10　初始类图：Osbert Oglesby 实例研究

实体类建模步骤的目的是抽象出实体类，确定它们的交互关系，并且找出它们的属性。这是通过使用 12.5.1 节的两阶段名词抽象过程进行初始化，在阶段 1 中，用单独的一个段落描述 Osbert Oblesby 实例研究，这样做的一个可能的方式是：

为提高购买艺术作品的决策过程的有效性，生成报表。报表包含画作的购买和出售信息，画作被分类为杰作、大作和其他画作。

在阶段 2 中，分辨出这个段落中的名词，为清楚起见，用黑体表示这些名词：

为提高**购买艺术作品**的决策**过程**的**有效性**，生成**报表**。**报表**包含**画作**的**购买**和**出售信息**，画作被分类为**杰作**、**大作**和**其他画作**。

这些名词是报表、有效性、过程、购买、艺术作品、出售、信息、画作、杰作和大作。有效性、过程和信息是抽象名词，因此不会是实体类。名词“购买”和“出售”是从动词“买”和“卖”对应派生而来的，因此它们可能是一些类的操作。名词“报表”不太可能是实体类，因为报表不是长期存在的，报表更可能是一个边界类。艺术作品只是画作的同义词，这就留下了 4 个候选实体类：**Painting Class**（画作类）、**Masterpiece Class**（杰作类）、**Masterwork Class**（大作类）和 **Other Painting Class**（其他画作类），如图 12-17 所示。

现在考虑这四个实体类之间的交互关系，杰作是画作的一个特定类型，大作和其他画作也是一样。以 7.6 节的术语描述，**Painting Class** 是基类，**Masterpiece Class**、**Masterwork Class** 和 **Other Painting Class** 是这个基类的子类，这在图 12-18 中反映出来。回想一下，空三角是继承（泛化）的 UML 表示法。

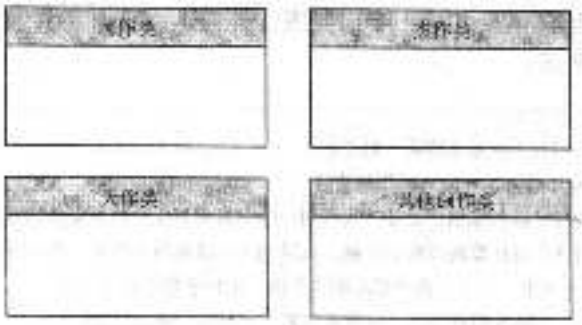


图 12-17　Osbert Oglesby 实例研究的初始类图的第一次迭代

很明显图 12-18 是图 12-17 的改进版，它反映了类之间的交互关系，然而还不够，至少需要不止一个交互行为。统一过程是用例驱动的，因此再次研究用例通常可以得到更深入的理解，而不是进行前一次迭代时开发者了解的知识层面。

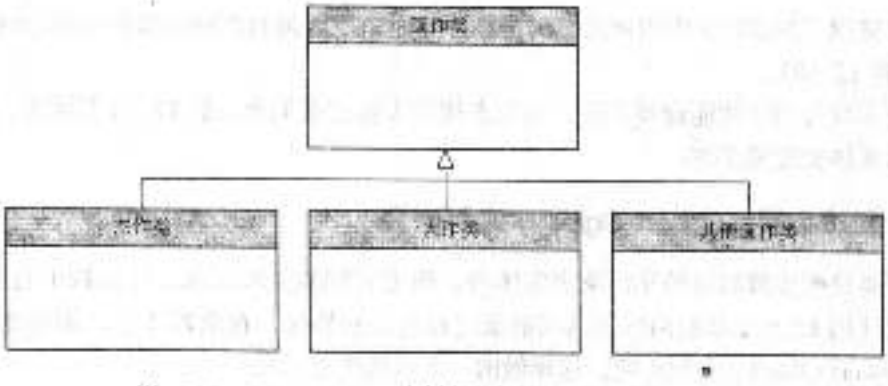


图 12-18　Osbert Oglesby 实例研究的初始类图的第二次迭代

在 Osbert Oglesby 软件产品的修订需求的 Buy a Painting 用例的描述中，确定 Osbert 将付给一幅画作的最高价格的算法如图 10-14 所示，从这个描述中可明显看到，该算法的一些关键方面还没有被准确地建模。

首先，根据图 12-18，一幅画作被分类为杰作、大作或其他画作。由于继承的本性，这意味着所有这三个类型继承了 `Painting Class` 的一个实例的所有属性，并且还具有它们自己的属性。另一方面，回想一下图 10-14 的段落 2.2：

2.2　对于大作，软件产品首先把画作当作同一画家的杰作来计算最高购买价格。

为满足这个段落的要求，大作应具有杰作的所有属性（这样可把它当作杰作一样计算最高购买价格）；另外，它可能还具有它自己的属性。换句话说，目前图 12-19 建模分辨出的实体类之间的关系比图 12-18 的建模要更精确。根据图 12-19，一幅画作首先被分类为杰作/大作或其他画作，如果它属于第一个分类，那么它还要进一步分类为杰作或大作。

最高价格算法的第二个关键方面是拍卖的画作还未被调整，图 10-14 的段落 2.1 这样开头：

2.1　对于杰作，软件产品计算此有该画家拍卖记录的每幅画作与正考虑要购买的画作之间的相似性系数。

很明显，需要有 **Auctioned Painting Class**（已拍卖的画作类），这样才能比较 Osbert 正考虑购买的画作和过去 25 年来世界范围内已拍卖的画作。问题是：**Auctioned Painting Class** 应该加到图 12-19 的哪里呢？已拍卖的画作是一幅画作，因此它应是 **Painting Class** 的子类。但先前在世界某地拍卖会上售出的画作与 Osbert 画廊里当前展出待售的画作没有关系，因此需要的是图 12-20 所示的类继承。

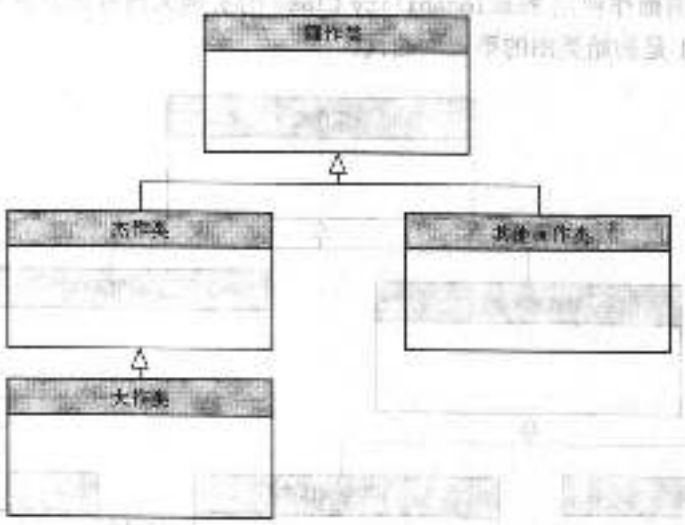


图 12-19　Osbert Oglesby 实例研究的初始类图的第三次迭代

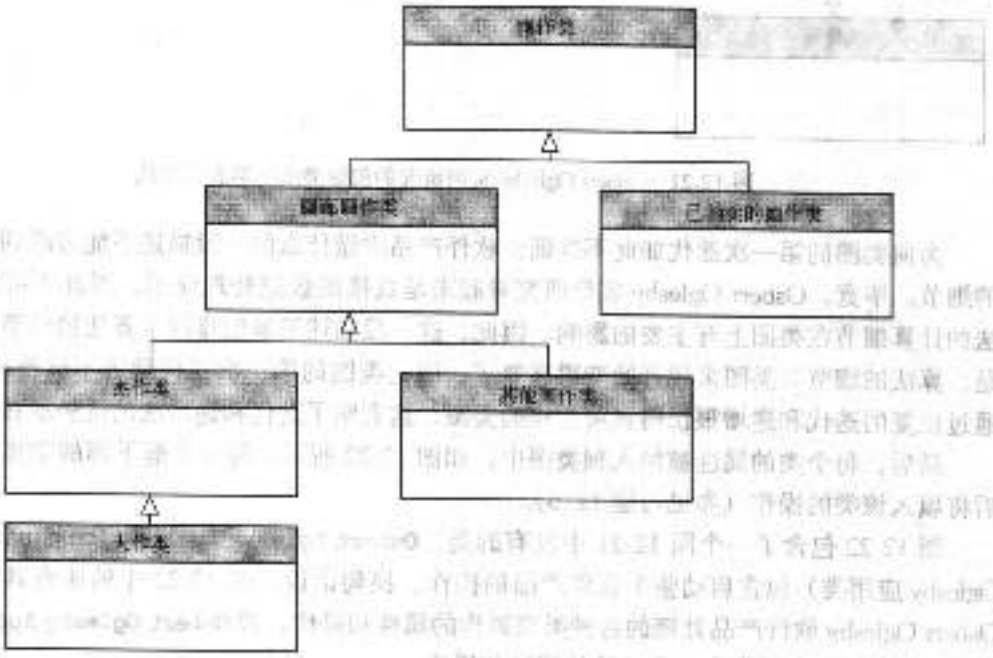


图 12-20　Osbert Oglesby 实例研究的初始类图的第四次迭代

最高价格算法的第三个关键方面是流行度还未被调整。图10-14的段落2.3这样开头：

2.3　对于其他画作，软件产品按公式$F \times A$计算最高价格，其中$F$是画家的一个常数（流行系数）……

需要有 **Fashionability Class**（流行度类），如图12-21的右下部所示。然后 **Other Painting Class** 的画作使用 **Fashionability Class** 的实例为该画家计算 Osbert 应开出的最高价格。图12-21是初始类图的第五次迭代。

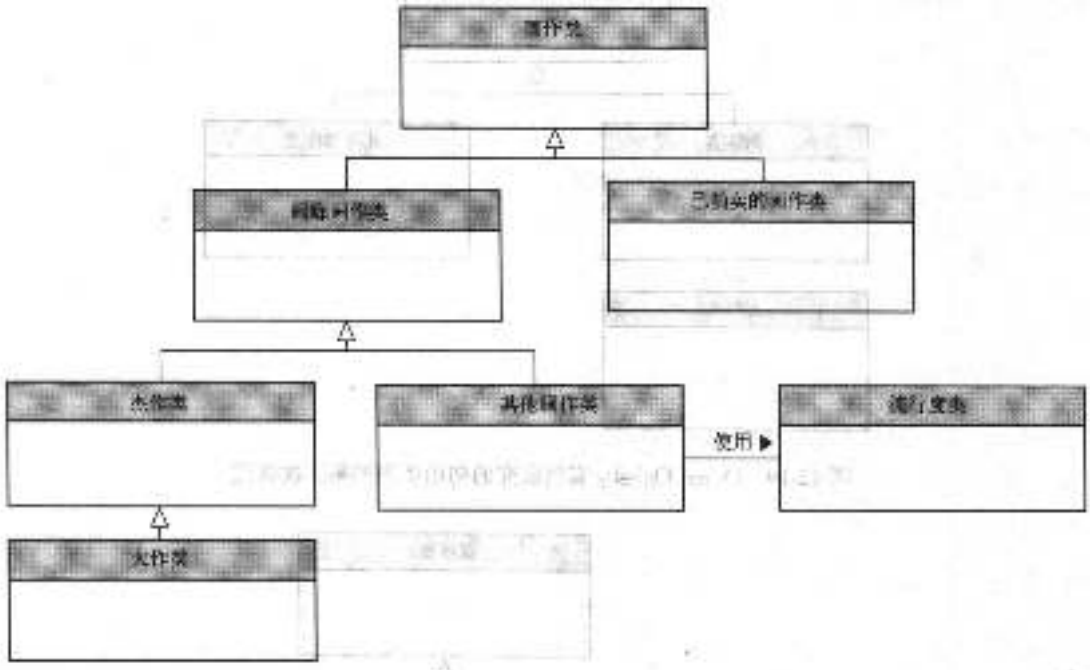



图12-21　Osbert Oglesby 实例研究的初始类图的第五次迭代

为何类图的第一次迭代如此不准确？软件产品应做什么的一段描述不能考虑到价格算法的细节。毕竟，Osbert Oglesby 实例研究看起来是直接的数据处理应用，因此不可能希望算法的计算细节在类图上有主要的影响。因此，这一段描述正确地排除了算法的细节。遗憾的是，算法的细节对类图来说开始变得重要了，因此类图的第一次迭代远远不够精确。然而，通过反复的迭代和递增很快得到可工作的类图，这表明了迭代和递增法的优势所在。

最后，每个类的属性被加入到类图中，如图12-22所示，每个方框下部的空矩形代表之后将填入该类的操作（参见习题12.6）。

图12-22包含了一个图12-21中没有的类，**Osbert Oglesby Application Class**（Osbert Oglesby 应用类）包含启动整个软件产品的操作，换句话说，图12-22中的所有其他类包含 Osbert Oglesby 软件产品处理的各种类型画作的属性和操作，而 **Osbert Oglesby Application Class** 包含软件产品作为一个整体的属性和操作。

最后，图12-22被重画，没有了属性但明确反映了每个类的固定形式（见12.1节），结果显示在图12-23中，图中所有的8个类是实体类，即它们为长期存在的信息建模。乍一看

图 12-23一点也不像图 12 22，然而两者实际都是类图。类图描述类和它们之间的相互关系，属性和操作是可选项（参见第 16 章）。

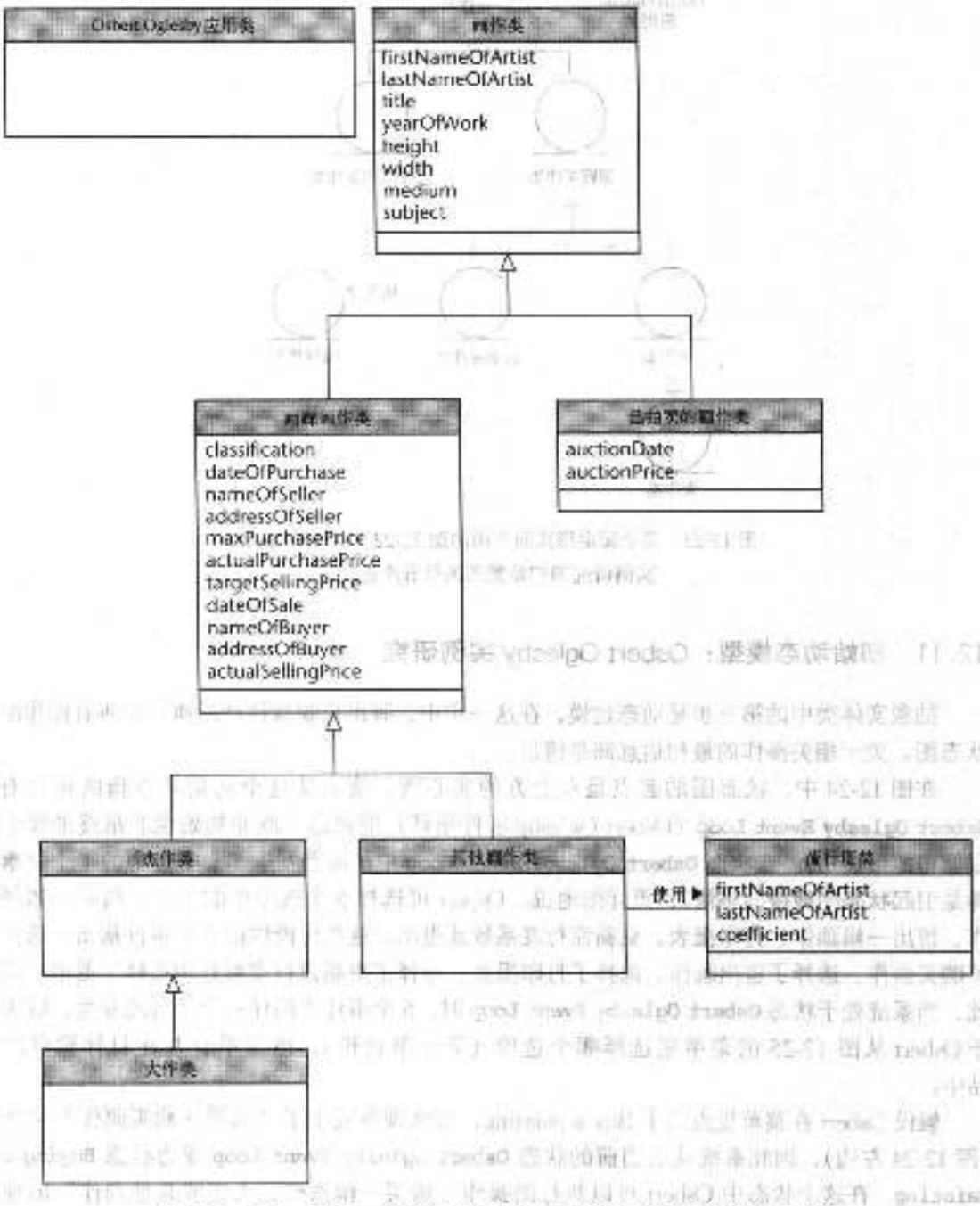



图 12-22 Odeot Ogleby 实例研究的初始类图的第五次迭代（添加了属性）

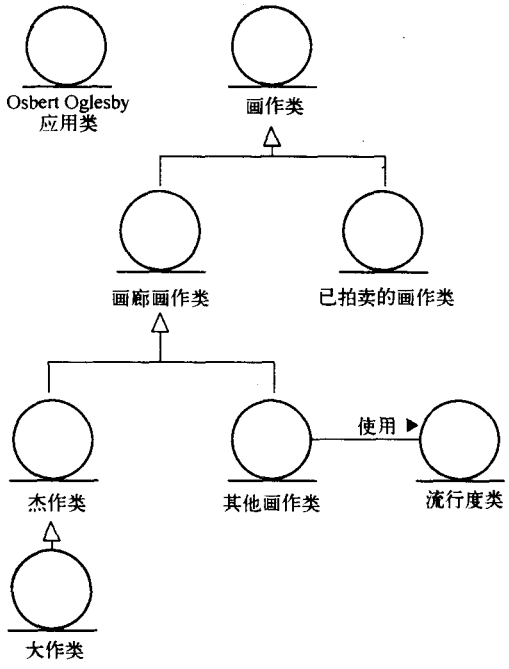



图 12-23　显示固定形式而重画的图 12-22（Osbert Oglesby 实例研究的初始类图的第五次迭代）

## 12.11　初始动态模型：Osbert Oglesby 实例研究

抽象实体类中的第三步是动态建模，在这一步中，画出反映软件产品执行的所有操作的状态图。关于相关操作的最初信息源是情景。

在图 12-24 中，状态图的起点是左上方的实心圆，箭头从这个初始状态指向标注有 **Osbert Oglesby Event Loop**（Osbert Oglesby 事件循环）的状态（既非初始也非最终的状态以圆角矩形表示）。在状态 **Osbert Oglesby Event Loop** 中，可能发生 5 个事件中的 1 个（**事件**是引起状态间**转换**的事情），更详细地说，Osbert 可选择 5 个选项中的 1 个：购买一幅画作、售出一幅画作、打印报表、更新流行度系数或退出。这些可能性由 5 个事件指出：选择了购买画作、选择了售出画作、选择了打印报表、选择了更新流行度系数和选择了退出。因此，当系统处于状态 **Osbert Oglesby Event Loop** 时，5 个事件中的任一个都可能发生，取决于 Osbert 从图 12-25 的菜单里选择哪个选项（下一节讨论），该菜单合并在目标软件产品中。

假设 Osbert 在菜单里点击了 Buy a painting，那么现在发生了“选择了购买画作”事件（图 12-24 左边），因此系统从它当前的状态 **Osbert Oglesby Event Loop** 变为状态 **Buying a Painting**。在这个状态中 Osbert 可以执行的操作“购买一幅杰作、大作或其他画作”出现在线下。从状态 **Buying a Painting**，箭头指回到状态 **Osbert Oglesby Event Loop**。状态图的其他部分的行为是类似的，特别地，如果 Osbert 点击了 Quit，系统变为它的最终状态，用一个白圈内的小黑圆表示。

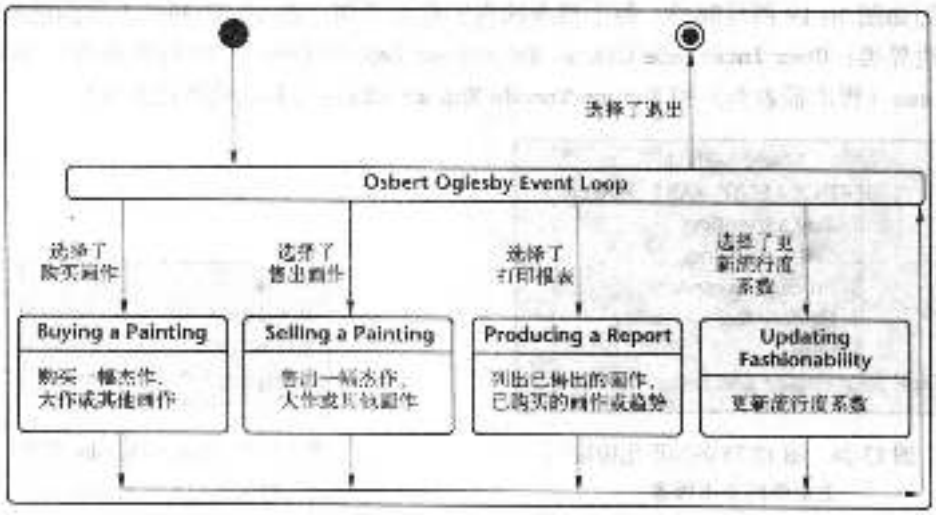



图 12-24　Osbert Oglesby 实例研究的初始状态图

在面向对象范型中，每个类都有一个动态模型，而不是系统作为一个整体有一个动态模型（如这个实例研究所示）。然而，在 Osbert Oglesby 实例研究内部，对象从不从一个类变为另一个类，因此软件产品作为一个整体的动态模型是合适的。

完成了初始功能性建模、实体类建模和动态建模后，检查所有这三种模型来确保实体类至少在现在看起来是正确的。

接下来，抽象边界类和控制类。




图 12-25　目标 Osbert Oglesby 软件产品中的初始主菜单

## 12.12　抽象边界类：Osbert Oglesby 实例研究

如 12.8 节所指出的，每个输入屏幕、输出屏幕和打印的报表通常由一个类建模。在 Osbert Oglesby 实例研究的情况中，从软件产品的相对专一性角度看，试图仅以一个屏幕让 Osbert使用所有的用例（购买一幅画作、售出一幅画作、打印报表和更新流行度系数）是合理的。在后来的迭代中，有必要将这一个屏幕求精为多个屏幕，然而同时，在初始类抽象中，只有一个屏幕类——`User Interface Class`（用户接口类）。

图 12-25 描述了用户接口屏幕的主菜单的第一次迭代。那里出现的 5 个命令对应前面图 12-24 中的 5 个事件。如果在后来的迭代中，额外的事件被加入到状态图中，对应的命令也应加入到这个菜单中。（附录 H（C++）和附录 I（Java）中给出的 Osbert Oglesby 实例研究的实现使用了文本的接口，而不是图形接口，即不是像图 12-25 中所示的点击方框，用户从键盘输入一个选择，如图 12-26 所示。例如，用户输入 1 来选择“Buy a painting”，输入 2 来选择“Sell a painting”，等等。附录 H 和 I 中的实现使用如图 12-26 的文本接口的原因是，文本接口可以在所有计算机上运行，而图形用户接口（GUI）通常需要特定的软件。）

有三种报表：购买报表、售出报表和未来趋势报表。每种报表由一个单独的边界类建

模，因为如图 10-19 所反映的，每个报表的内容截然不同。图 12-27 列出了得到的四个对应的初始边界类：**User Interface Class**、**Purchases Report Class**（购买报表类）、**Sales Report Class**（售出报表类）和 **Future Trends Report Class**（未来趋势报表类）。




图 12-26　图 12-25 的图形化初始主菜单的文本版本



图 12-27　Osbert Oglesby 实例研究中的初始边界类

## 12.13　抽象控制类：Osbert Oglesby 实例研究

如 12.8 节所说明的，每个重要的计算通常由一个控制类进行建模。在 Osbert Oglesby 实例研究中，有 4 个计算，分别是确定 Osbert 应为一幅杰作、大作或其他画作开出的最高价格，以及确定在艺术品买卖市场上是否有新趋势。这产生了 4 个初始控制类：**Compute Masterpiece Price Class**（计算杰作价格类）、**Compute Masterwork Price Class**（计算大作价格类）、**Compute Other Painting Price Class**（计算其他画作价格类）和 **Compute Future Trends Class**（计算未来趋势类）。它们列于图 12-28 中。



图 12-28　Osbert Oglesby 实例研究中的初始控制类

## 12.14　求精用例：Osbert Oglesby 实例研究

现在检查这三组类，这个检查显示出需要进一步的求精。回想一下图 12-18 的类图（第二次迭代）中，三种类型的艺术作品被作为 **Painting Class** 的子类，然而经过该类图的第 5 次迭代（图 12-21）后，显然作为价格算法的结果，这三种类型的艺术作品中的每个都非常不一样。

类似地，再看图 12-12 的用例，现在看起来用例 Buy a Painting 需要求精为三个单独的用例：Buy a Masterpiece（购买一幅杰作）、Buy a Masterwork（购买一幅大作）和 Buy Other Painting（购买其他画作），这在图 12-29——Osbert Oglesby 软件产品用例图的第三次迭代中反映出来。

另外，需要求精 Produce a Report（生成报表）用例，报表中的两个——购买报表和售出报表是数据抽象的直接例子。但未来趋势报表涉及一个计算，而且所有这三种报表使用它

们自己的边界类，如图 12-27 所示。由于这两个原因，Produce a Report 用例需要被求精为三个用例：Produce a Purchases Report（生成购买报表）、Produce a Sales Report（生成销售报表）和 Produce a Future Trends Report（生成未来趋势报表）。图 12-29 也反映了这个求精。

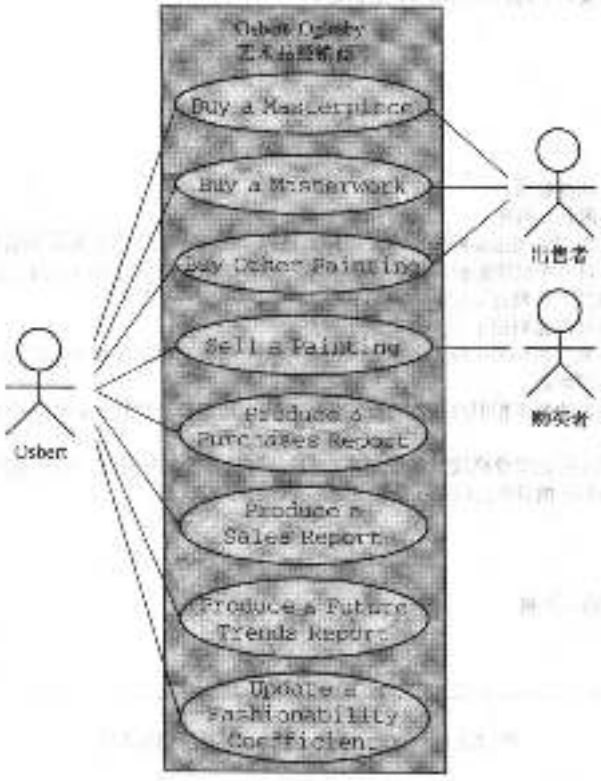



图 12-29　Osbert Oglesby 实例研究的用例图的第三次迭代

现在必须检查这些用例图求精的其他 UML 图的含义。首先，当 Osbert 希望购买一幅画作时，没有理由提供三个单独的用户接口。Osbert 使用这个软件产品购买一幅画作时必须提供的内容之一是画作的分类，然后调用相关的算法，因此作为求精 Buy a Painting 用例为三个单独的用例的结果，惟一增加的变化是将 Buy a Painting 用例（图 10-11）的描述分为三个单独的描述。Buy a Masterpiece 用例（图 12-30）的描述在图 12-31 中给出，其他两个描述留做练习（习题 12.11 和习题 12.12），另三个报表用例的描述也留做练习（习题 12.13 到习题 12.15）。

详细检查了类抽象后，现在我们转向统一过程［Jacobson, Booch, and Rumbaugh, 1999］。




图 12-30　Osbert Oglesby 实例研究的 Buy a Masterpiece 用例

**简要描述**

Buy a Masterpiece 用例使 Osbert Oglesby 能够购买一幅杰作。

**按步骤描述**

1. Osbert 输入他正考虑要购买的杰作的细节，它们是：
   画家的名
   画家的姓
   作品标题
   作品年份
   高度
   宽度
   材质（油画、水彩、其他材质）
   主题（肖像、静物、风景、其他）
2. 软件产品回答 Osbert 应提供的最高购买价格。更详细地说，软件产品计算有该画家拍卖记录的每幅画作与正考虑要购买的作品之间的相似性系数。画家的名或姓，或者作品标题或时间中的问号忽略不计。
   如果材质上相符，软件产品则记 1 分，否则记 0 分。
   如果主题上相符，软件产品则记 1 分，否则记 0 分。
   将这两个得数相加，乘以两幅画中较小的那幅画的面积，再除以两幅画中较大的那幅画的面积。
   得到的结果就是相似性系数。
   软件产品找寻具有最大的非零相似性系数的被拍卖的画作，如果没有这样的画作，Osbert 将不购买这幅考虑中的画作。
   软件产品把最相似作品的拍卖价格按每年增加 8.5% 计算（从拍卖那年算起）得出最高购买价格。
3. 如果出售者接受了 Osbert 的开价，Osbert 会输入进一步的细节，包括：
   购买时间
   出售者的姓名
   出售者的地址
   由该算法确定的最高购买价格
   实际购买价格
   目标售出价格

图 12-31　Buy a Masterpiece 用例的描述

## 12.15　用例实现：Osbert Oglesby 实例研究

用例是对动作者和软件产品之间的交互的描述，用例首先应用在软件生命周期的开始阶段，即在需求流。在分析和设计流，更多的细节被加入到每个用例中，包括实现用例所涉及的类的描述。扩展和求精用例的这个过程被称为**用例实现**（use-case realization）。最后，在实现流，用例被实现为代码。

这个术语有点让人迷惑，因为动词 realize 至少有三种不同的含义：

- 明白（“Harvey 开始慢慢意识到他进错教室了”）
- 收到（“Ingrid 将在股票交易中实现 45 000 美元的收益”）
- 完成（“Janet 希望实现开办计算机公司的梦想”）

在短语“realize a use case”（实现一个用例）中，“realize”（实现）一词具有最后一种含义，即它意味着完成（或达到）用例。

**交互图**（**顺序图**或**协作图**）描述了用例的一个特定情景的实现，为明白这是如何做的，我们回到 Osbert Oglesby 实例研究，考虑 Buy a Masterpiece 用例。

### 12.15.1　Buy a Masterpiece 用例

图 12-29 是 Osbert Oglesby 实例研究的当前用例图，即它是合并了该实例研究在这个阶

段分辨出的所有用例的一个图，包括图 12-30 单独显示的 Buy a Masterpiece 用例。图 12-31 是 Buy a Masterpiece 用例的描述——它规定了编写用例细节方面的内容。

如图 12-32 的类图中所示，进入这个用例的类包括 **User Interface Class**（为用户接口建模）和 **Compute Masterpiece Price Class**（为 Osbert 应开价的计算建模）。这个计算涉及比较考虑中的杰作（**Masterpiece Class** 的一个实例）和以前拍卖过的杰作（**Auctioned Painting Class** 的所有实例）。**Seller**（出售者）不直接与软件产品进行交互，只是给 Osbert 提供输入到软件产品中的数据，这在注释（右上角折叠的矩形）中指出。虚线从这个注释指向它所解释的事项（在本例中是 **Seller**）。

图 12-32 是一个类图，显示了实现用例的类，以及这些类之间的关系。图 12-32 是图12-23 风格的类图，而不是图 12-22 风格的类图，也就是说，图 12-32 描述了类和它们的交互关系，只是没有显示可选的属性而已。

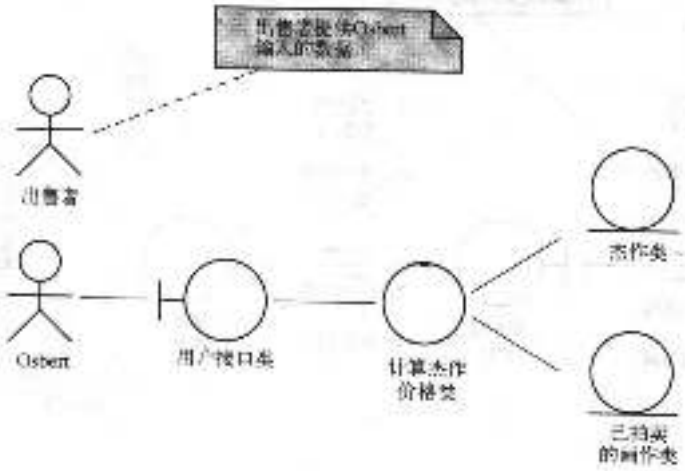



图 12-32　实现 Osbert Oglesby 实例研究的 Buy a Masterpiece 用例的类的类图

就像我们已经看到的，需求流中代表用例的制品是用例和它的描述。在分析流中，代表用例的是一些不同的制品，其中之一就是显示实现用例的类的类图。

回想一下，情景是用例的一个可能的实例，也就是说，用例为动作者和软件产品之间的所有交互行为集进行建模。每个交互行为是该用例的一个情景（实例）。例如，图 12-13 是 Buy a Masterpiece 用例的一个情景，在这里图 12-33 重新显示了它，这样在图 12-32 中的类就能够实现图 12-33 的情景。

Osbert Oglesby 希望购买一幅杰作。
1. Osbert 输入画作的描述。
2. 软件产品扫描拍卖记录找寻同一画家最相似作品的售出价格和年份。
3. 软件产品计算最高购买价格，从最相似作品的拍卖那年算起，每年增加 8.5%。Osbert 给出一个开价，低于最高购买价格，出售者接受了该价格。
4. Osbert 输入出售信息（出售者的姓名和地址、购买价格）。

图 12-33　购买一幅杰作的一个可能的情景

一个运转中的软件产品使用对象，而不是类。例如，Masterpiece Class 不能代表一个特定的杰作，而一个对象——Masterpiece Class 的一个特定实例却可以。我们将这样的一个对象记为：Masterpiece Class。（第 16 章解释使用冒号的原因。）诸如图 12-32 的类图显示了用例中的类和它们的关系，它既不显示对象，也不显示从对象发往对象的消息序列。我们需要的不止是这些。

图 12-34 是一个协作图，即它显示了对象和消息，消息按照在一个特定的情景（在这个例子中是图 12-33 的情景）中被发送的顺序编号。在这个情景的第 1 段里，Osbert 输入它正考虑购买的杰作的细节。在图 12-34 中，从 Osbert 到对象：User Interface Class 的消息“1：给出杰作细节”代表了这个过程，箭头的方向显示了信息流的方向。

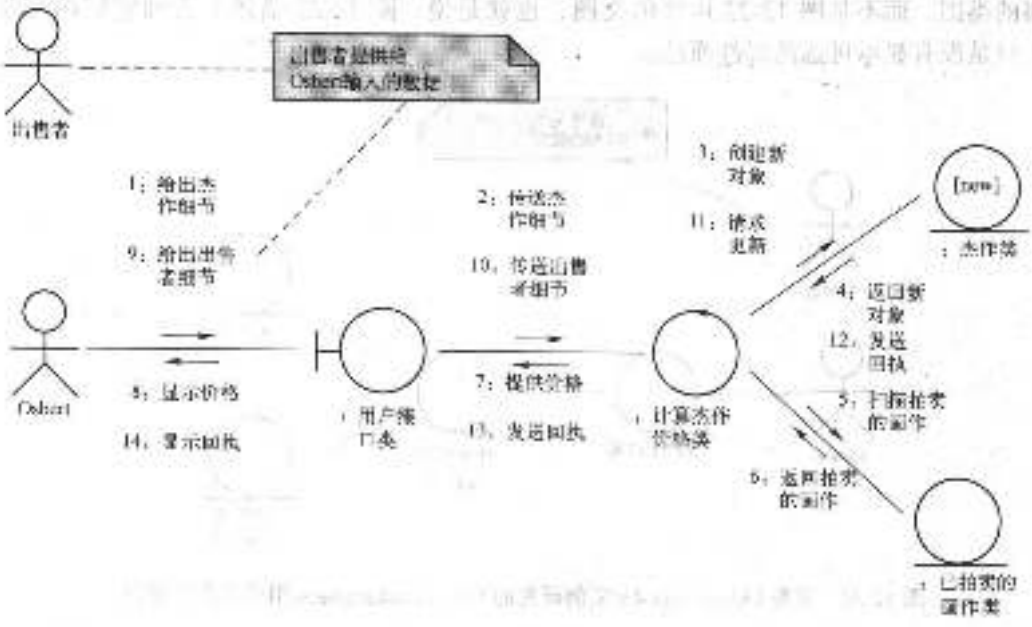



图 12-34　图 12-33 的情景的实现的协作图

在这个情景（图 12-33）的第 2 段和第 3 段，通过比较考虑中的杰作和之前拍卖的杰作来计算最高购买价格。这个计算通过控制类的一个实例——对象：Compute Masterpiece Price Class 来完成。为做到这一点，考虑购买的杰作的细节需要从对象：User Interface Class 传送到对象：Compute Masterpiece Price Class，消息“2：传送杰作细节”为此过程建模。

为了进行比较，对象：Compute Masterpiece Price Class 创建了一个杰作对象，它首先创建了 Masterpiece Class 的一个实例，从：Compute Masterpiece Price Class 到：Masterpiece Class 的消息“3：创建新对象”为此建模。该对象内部的［new］显示出它是作为该消息的结果创建的，这个新对象又被送回到：Compute Masterpiece Price Class（4：返回新对象）。

现在：Compute Masterpiece Price Class 比较考虑购买的杰作和之前拍卖过的每幅杰作，即它扫描 Auctioned Painting Class 的所有实例。这通过从：Compute Masterpiece Price Class 到：Auctioned Painting Class 的消息“5：扫描拍卖的画作”和反向的消息“6：返回拍卖的画作”来建模。

发现了最匹配的杰作后，**: Compute Masterpiece Price Class** 计算开出的最高价格，并通知 Osbert 该价格是多少。换句话说，从**: Compute Masterpiece Price Class** 向用户接口对象**: User Interface Class** 传送该价格，这样可以显示该价格，消息“7：提供价格”为此建模。接下来为 Osbert 显示该价格（消息“8：显示价格”）。

Osbert 给这幅杰作开价，图 12-33 的情景的第 3 段以下的内容表明该开价被接受。如该情景的第 4 段所讲，Osbert 现在需要输入出售者提供的细节，这由从 Osbert 传送给对象**: User Interface Class** 的消息“9：给出出售者细节”来建模。注释指出该数据是由出售者提供给 Osbert 的。

出售者的数据又传送给对象**: Compute Masterpiece Price Class**（消息“10：传送出售者细节”）来更新杰作对象**: Masterpiece Class** 的细节（消息“11：请求更新”）。然后发送回执给**: Compute Masterpiece Price Class**（消息“12：发送回执”）。这个回执之后被传送给用户接口对象**: User Interface Class**（消息“13：发送回执”），以显示给 Osbert（消息“14：显示回执”）。

除非 Osbert 准确理解了被提议的软件产品将做什么，否则他不会同意规格说明文档，因而需要协作图的文字描述，如图 12-35 的**事件流**所示。

> Osbert 输入他正考虑购买的杰作的细节（1）。软件产品然后查找所有已经被拍卖的杰作，找到与考虑中的杰作最接近的一幅（2-6），然后使用提供的公式计算 Osbert 应开出的最高价格（7-8）。
>
> Osbert 现在开出价，并且被接受，他提供出售者的细节（9），用于更新杰作数据（10-14）。

图 12-35　图 12-33 的情景的实现的协作图（图 12-34）的事件流

UML 支持两种不同类型的交互图，第一个是如图 12-34 所示的协作图，第二个是顺序图。协作图和顺序图包含绝对相同的信息，但以不同的方式显示。图 12-36 是顺序图，与图 12-34 的协作图是一样的。在协作图中的**: Masterpiece Class** 内部的［new］对象在顺序图中对应的对象被向下移动，这样它的**生命线**（被称作垂直线）开始于创建对象的地方。**激活框**（生命线上的窄矩形）显示何时相关的对象是激活的。

图 12-36 的顺序图显示图 12-33 的情景的每个消息或者与用户接口类**: User Interface Class** 的实例有关，或者与控制类**: Compute Masterpiece Price Class** 的实例有关。它还显示出每个从对象 A 到对象 B 的信息传送最终都跟着一个反方向的传送。在完全等价的协作图（图 12-34）中，这两种情况也是真的，但不那么明显。软件产品开发者可以选择使用每个情景的实现的顺序图或者协作图，或者两者都选择。

顺序图的优点是它明白地显示了消息流，消息的次序特别清楚，每个单独的消息的发送者和接收者也特别清楚。因此，当信息的传输是关注点时（完成分析流时多数是这种情况），顺序图好于协作图。另一方面，类图（如图 12-32）和实现相关情景（如图 12-34）的协作图之间的相似性很强。因而，在那些开发者关注于类的情况下，协作图通常比等价的顺序图更有用。

应记住的关键一点是，图 12-30 到图 12-36 都是用例，或者用例的实例，或者用例的实例的文字描述。更详细地，

- 图 12-30 描述了用例 Buy a Masterpiece。

- 图 12-31 是那个用例的描述，即它提供了图 12-30 的 Buy a Masterpiece 用例的细节的文字版。
- 图 12-32 是显示实现 Buy a Masterpiece 用例的类图，类图描述该用例所有可能的情景的类和它们的交互行为。
- 图 12-33 是一个情景，即图 12-30 的用例的一个实例。
- 图 12-34 是实现图 12-33 的情景的协作图，即它描述了实现情景中的对象和它们之间发送的消息。
- 图 12-35 是图 12-34 的协作图的事件流，即，就像图 12-31 是图 12-30 的 Buy a Masterpiece 用例的文字描述一样，图 12-35 是图 12-34 的协作图的文字描述。
- 图 12-36 是与图 12-34 的协作图完全等价的顺序图，因此图 12-35 也是图 12-36 的顺序图的事件流。

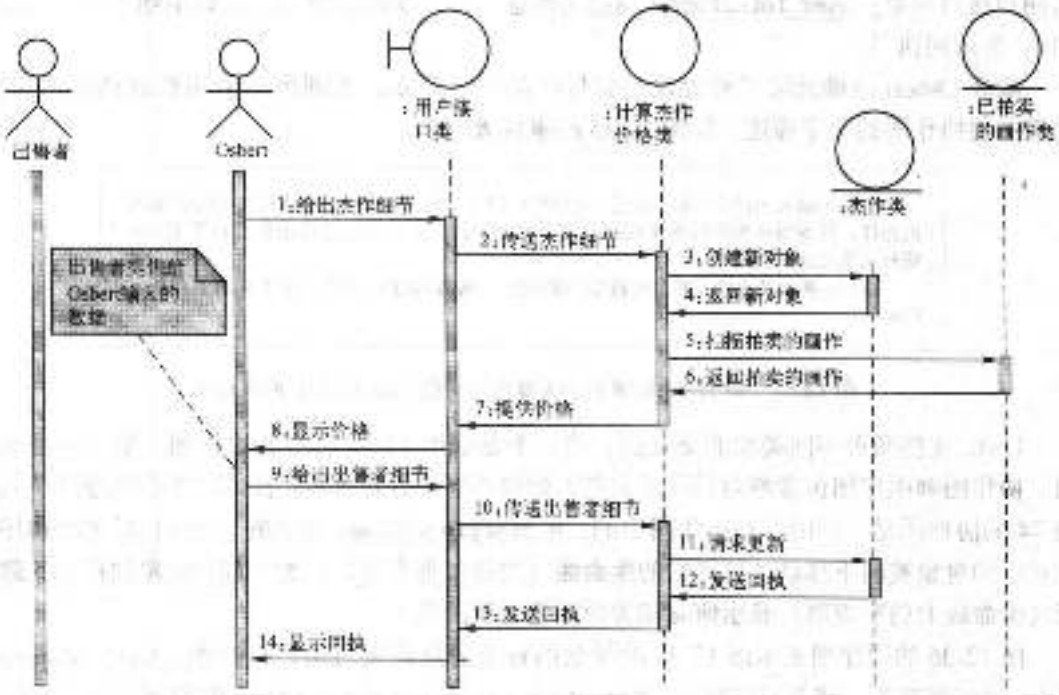



图 12-36　图 12-33 的情景的实现的顺序图，与协作图等价
（因此顺序图的事件流也如图 12-35 所示）

图 12-30 到图 12-36 使用不同的符号，提供同一行为（购买一幅画作）的不同级别的细节。我们构造这么多相关的事物的原因是我们从各个不同的方面检查这一行为，以便更了解它，确保分析流的正确性。

### 12.15.2　Buy a Masterwork 用例

考虑实现 Buy a Masterwork 用例的类的类图（图 12-37），回想一下如何确定一幅大作的最高价格，首先把考虑购买的大作当作一幅杰作来考虑，然后调整结果。换句话说，进入这个用例的类首先是 User Interface Class，它对用户接口建模，还有计算 Osbert 应开的价格

的 **Compute Masterwork Price Class**。它通过创建一个大作对象（**Masterwork Class** 的一个实例）来完成这个计算，并将这个对象传送给 **Compute Masterpiece Price Class**。再回想一下，如图 12-21 所示，一幅大作“是”一幅杰作（参见 7.7 节），因此软件产品中计算 Osbert 应为一幅杰作提出的最高价格的部分与为一幅大作的相应计算完全一样。因此，进入这个计算的其他类是 **Compute Masterpiece Price Class** 和 **Auctioned Painting Class**。图 12-38 显示了这个用例的一个情景，实现这个情景的其余 UML 图与用例 Buy a Masterpiece 的对应部分类似（图 12-34 到图 12-36）。

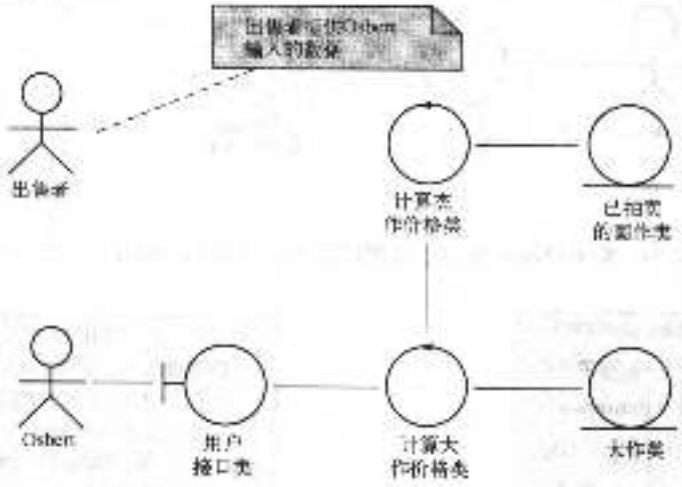



图 12-37 实现 Osbert Oglesby 实例研究的 Buy a Masterwork 用例的类的类图

Osbert Oglesby 希望购买一幅创作于 17 世纪的大作。

1. Osbert 输入该画作的描述。
2. 软件产品扫描拍卖记录，找寻同一画家的最相似作品的销售价格和年份。
3. 软件产品计算最高购买价格，把最相似作品的拍卖价格按每年增加 8.5% 计算，并将结果乘以（21－17）/（22－17）或 0.8。
   Osbert 开出一个价，低于最高购买价格——出售者接受了他的开价。
4. Osbert 输入销售信息（出售者的姓名和地址，购买价格）。

图 12-38 购买一幅大作的一个可能的情景

### 12.15.3 Buy Other Painting 用例

实现 Buy Other Painting 用例的类的类图如图 12-39 所示。情景和交互行为图（协作图、顺序图）和相关的事件流留做练习（习题 12.16 到 12.19）。

边界类 **User Interface Class** 还是出现了，这并不令人惊奇，因为像前面所解释的，购买所有这三种类型的画作（事实上是通过所有 8 个用例）使用相同的用户接口，因此图 12-25 的初始主菜单需要被调整以明确反映购买这三种不同类型的画作。这样，图 12-25 中的 Buy a painting 被 Buy a masterpiece、Buy a masterwork 和 Buy other painting 替代。这可以确保调用正确的算法计算 Osbert 应开出的最高价格。图 12-40 显示的修订后的屏幕由：**User Interface Class**

生成。(附录 H (C++) 或附录 I (Java) 中实现的对应文本接口如图 12-41 所示。)

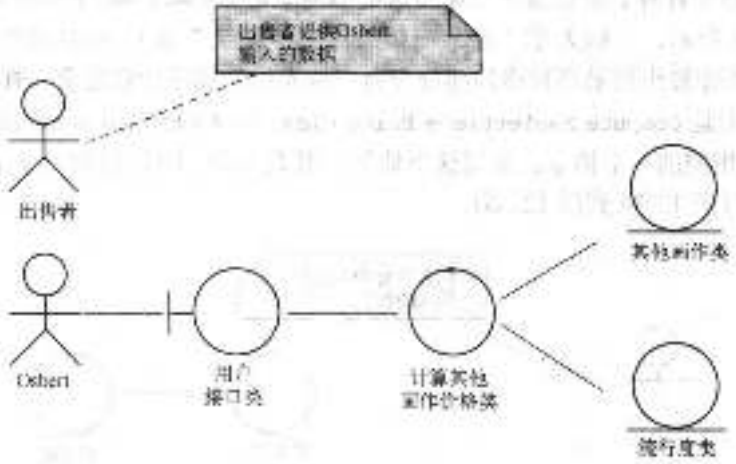



图 12-39　实现 Osbert Oglesby 实例研究 Buy Other Painting 用例的类的类图




图 12-40　目标 Osbert Oglesby 实例研究中主菜单的第二次迭代

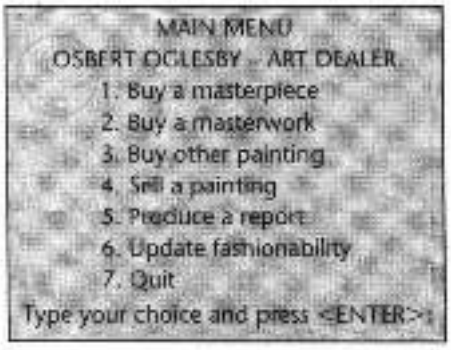



图 12-41　图 12-40 的图形化主菜单的文本版本

### 12.15.4　其余 5 个用例

图 12-42 描述了实现 Sell a Painting 用例的类的类图。很容易理解，这个实现是该用例（图 10-15）的简化版的结果。

现在来看三个报表用例，实现这些用例的类的类图如图 12-43 到图 12-45 所示。前两个类图只在输出接口类中不同，第三个用例（Produce a Future Trends Report）与另两个不同。用例 Produce a Purchases Report 和 Produce a Sales Report 只是浏览 **Gallery Painting Class**，即 Osbert 已经购买的画作（可能以后售出了）。但用例 Produce a Future Trends Report 需要完成图 10-19 中的段落 1.3 所解释的计算，这在图 12-45 的类图中有所反映。实现这个用例的细节留作练习（习题 12.20 到习题 12.23）。

最后，实现 Update a Fashionability Coefficient 用例的类的类图如图 12-46 所示，这个用例也很容易理解。

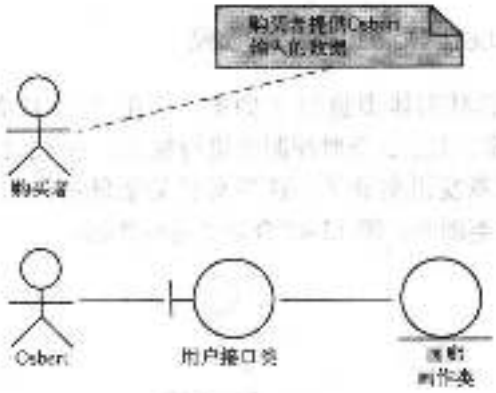



图 12-42　实现 Osbert Oglesby 实例研究 Sell a Painting 用例的类的类图




图 12-43　实现 Osbert Oglesby 实例研究 Produce a Purchases Report 用例的类的类图




图 12-44　实现 Osbert Oglesby 实例研究 Produce a Sales Report 用例的类的类图

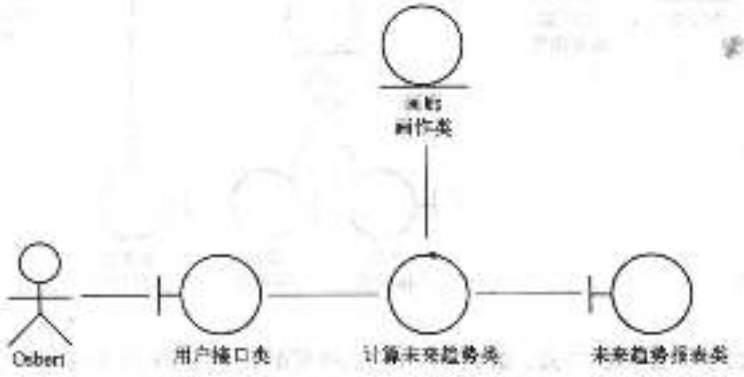



图 12-45　实现 Osbert Oglesby 实例研究 Produce a Future Trends Report 用例的类的类图




图 12-46　实现 Osbert Oglesby 实例研究 Update a Fashionability Coefficient 用例的类的类图

## 12.16　类图递增：Osbert Oglesby 实例研究

从12.9节到12.11节对实体类进行了抽象，产生了图12-23，显示出8个实体类。12.12节对边界类进行抽象，12.13节对控制类进行抽象。在12.15节实现各种用例的过程中，许多类之间的交互关系变得明显了，这些交互关系显示在图12-32、图12-37、图12-39、图12-42到图12-46的类图中。图12-47合并了这些类图。

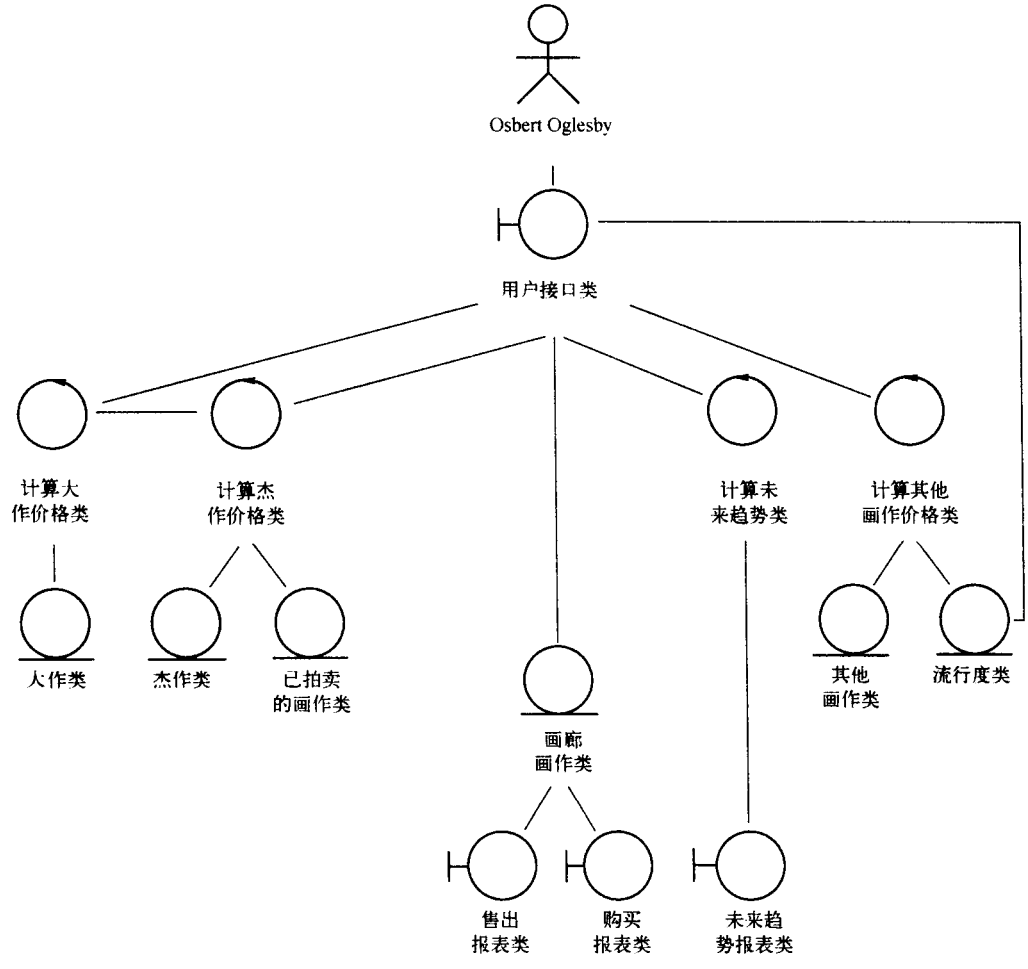



图12-47　合并图12-32、图12-37、图12-39和图12-42到图12-46中类图的类图

现在图12-23和图12-47的类图合并生成了Osbert Oglesby实例研究的类图的第六次迭代，如图12-48所示。更明确地，图12-23的**Osbert Oglesby Application Class**和**Painting Class**被加入到图12-47中，然后图12-23中的6个关系被画入，用虚线显示，与图12-47的关系相区分，结果生成的图12-48——类图的第六次迭代，是在分析流后期的类图。

就在Osbert Oglesby实例研究的传统分析完成时（11.15节），软件项目管理计划被提出。附录F包含由一个小的软件公司（3个人）开发的Osbert Oglesby产品的软件项目管理计划，这个计划符合IEEE SPMP格式。

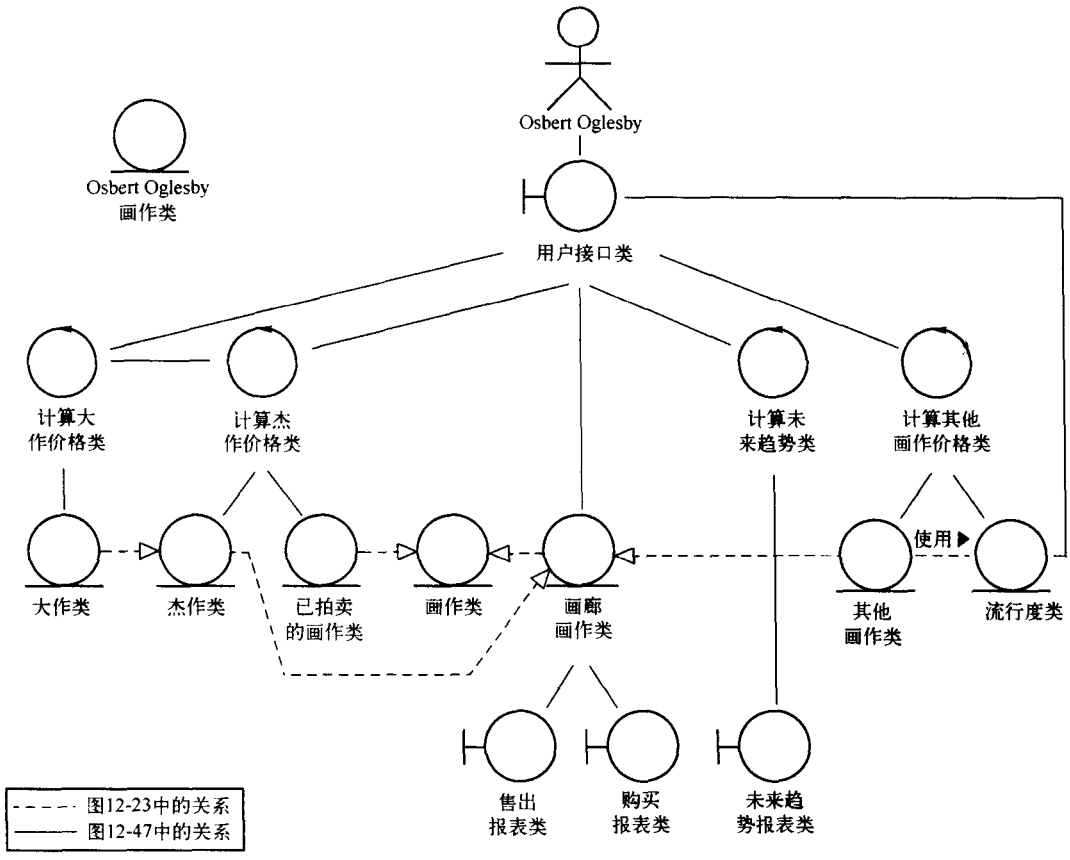



图 12-48　通过合并图 12-23 和图 12-47 的类图得到 Osbert Oglesby 实例研究的类图的第六次迭代

## 12.17　测试流：Osbert Oglesby 实例研究

两天后检查 Osbert Oglesby 实例研究的分析流。首先使用 CRC 卡片检查类，如 12.7 节所描述。然后审查分析流的所有制品（见 6.2.3 节）。

这就完成了 Osbert Oglesby 实例研究的分析流。

## 12.18　统一过程中的规格说明文档

分析流的基本目标是生成规格说明文档，但前一节的最后已经声明分析流完成了，明显的问题是，规格说明文档在哪里？

简单地说，统一过程是用例驱动的。更详细地说，用例和从中派生出来的制品包含了传统范型里文本形式的规格说明文档中的所有信息。

例如，考虑用例 Buy a Masterpiece。当需求流完成时，Buy a Masterpiece 用例（图 10-4）和它的描述（图 10-11）被展示给客户。系统分析员必须小心翼翼地确保他们的客户——Osbert Oglesby 完全理解这两样制品，并认为他们已准确地为他所需的软件产品进行建模。然后，在分析流期间，给 Osbert 展示购买一幅画作的情景（图 12-13 到图 12-15，或者图 12-16的扩展情景）、用例 Buy a Masterpiece（图 12-30）、它的描述（图 12-31）、实现

该用例的类的类图（图 12-32）、实现该用例的一个情景的交互图（图 12-34 和图 12-36）以及这些交互图的事件流（图 12-35）。

刚刚列出的这些东西只是属于用例 Buy a Masterpiece，如图 12-29 所示，一共有 8 个用例。对于其他的 7 个用例，每个都有相同的这些东西，其中一些是图形化的，一些是文本的，向客户传达的信息越多，就可能比传统范型的纯文本规格说明文档更准确。

传统的规格说明文档通常扮演合同的角色，即一旦它被开发者和客户签署了，它实质上构成了一个具有法律效力的文档。如果开发者研制了一个满足规格说明文档的软件产品，客户乐于为软件产品付款，相反如果产品不符合规格说明文档，开发者如果想得到付款，将被要求修改软件。在统一过程的情况里，整套的用例类似地形成了一个合同，因此，如上一节最后所讲，Osbert Oglesby 实例研究的分析流实际上已完成了。

如前面所说，统一过程是用例驱动的。当使用统一过程时，展示给客户的是用例，更准确地说，是反映实现用例情景的类的交互图，而不是建造一个快速原型。

客户通过交互图和它们的形成文字的事件流（如同通过快速原型一样），可以理解目标软件产品将如何运转。毕竟，一个情景是被提议的软件产品的一个特定的执行过程，如同快速原型的每个执行一样，区别在于，快速原型通常是抛弃型的，而用例则通过每次添加更多的信息而得到成功地求精。

然而，快速原型在用户接口方面优越于情景，这并不意味着建立快速原型只是为了客户和用户能够检查范例屏幕和报表。但是需要组织范例屏幕和报表，如图 10-13 所示，更适宜在 CASE 工具（例如屏幕生成器和报表生成器）的协助下完成（见 5.4 节）。

在下一节讨论确定动作者和用例的方法。

## 12.19　关于动作者和用例更详细的内容

如 10.4.3 节所讲，一个用例描述了软件产品本身和动作者（软件产品的使用者）之间的交互行为。下面提供了一些动作者和用例的例子，适于描述如何寻找动作者和用例。

为寻找动作者，我们考虑可与软件产品交互的个人的每个**角色**（role）。例如，考虑一对夫妇希望从银行得到抵押，当他们申请抵押时，他们是**申请者**，而当他们的申请被批准，并把买房子的钱借给他们后，他们变成了**贷款者**。换句话说，动作者不是这对夫妇，只是开始时他们扮演**申请者**的角色，然后扮演**贷款者**的角色。这意味着只列出使用软件产品的个人不足以发现动作者。我们需要找出每个使用者（或使用者组）扮演的所有角色，从这个角色列表中我们可以抽象动作者。

在统一过程的术语中，术语工作者（worker）代表个人扮演的特定角色，它有点不合宜，因为工作者这个词通常指一个雇员。按统一过程的术语，在申请抵押的夫妇的情况中，申请者和贷款者是两个不同的工作者，在本书中为清楚起见，以“角色”一词代替“工作者”。

在一个业务内部，找寻角色的任务通常很直截了当。用例的业务模型通常显示与业务有交互的个人所扮演的所有角色，这样突出了业务的动作者。然后我们找到用例业务模型的子集，对应需求的用例模型，更详细地说，

1）通过找寻与业务有交互的个人所扮演的所有角色来建造用例业务模型。

2）找到能够为我们希望开发的软件产品建模的业务模型的用例图的子集，即，只考虑与提议中的软件产品对应的业务模型中的那些部分。

3）在这个子集中的动作者就是我们找寻的动作者。

一旦确定了动作者，通常可直接找到用例。对于每个角色，有一个或多个用例。因此，找到需求的用例的开始点是找到动作者，如本节所述。

下面的“如何完成”部分总结了面向对象的分析。

**如何完成面向对象的分析**

- 迭代
  完成功能性建模。
  完成实体类建模。
  完成动态建模。
- 直到令人满意地抽象了实体类。
- 抽象边界类和控制类。
- 求精用例。
- 完成用例的实现。

## 12.20　用于面向对象分析阶段的 CASE 工具

由于我们已经知道了图在面向对象分析中所起的作用，就不会对开发出一些 CASE 工具来支持面向对象分析感到惊奇了。在它的基本形式中，这样一个工具本质上是一个画图工具，它使得完成每个建模步骤容易一些。更重要的是，修改一个用画图工具构建的图比试图改变一个手工画的图简单得多，因此，CASE 工具支持面向对象分析的图形方面的特性，此外，一些这种类型的工具不仅画出所有有关的图，同时还有 CRC 卡片。这些工具的一个优点是，对其中蕴含的模型的修改自动反映在全部受影响的图中，毕竟，各种图体现的仅是其背后的模型的不同视图而已。

另一方面，某些 CASE 工具不仅支持面向对象分析，还支持面向对象生命周期中许多其他的部分。现在实际上全部这些工具都支持 UML［Rumbaugh，Jacobson，and Booch，1999］，这类工具的例子包括 Rose 和 Together。ArgoUML 是一个这种类型的典型的公开源码 CASE 工具。

## 12.21　面向对象分析阶段所面临的问题

面向对象分析是一个特定的分析方法，因此 11.16 节所描述的传统分析所面临的问题也同样适用于面向对象分析。特别是那一节所列的第二个挑战，容易跨越规格说明（“什么”）和设计（“如何”）之间的边界线，这个危险对于面向对象分析的情形特别严重。

我们还记得，像 1.9 节所描述的那样，从面向对象分析向面向对象设计的转变，比在传统范型中从分析阶段向设计阶段的转变要平滑得多。在传统范型中，设计阶段的首要任务是将产品分解成为模块。相反，类——面向对象设计流的“模块”，是在面向对象分析流期间抽象的，准备在面向对象设计流期间求精。在 OOA 流的早期就给出类意味着，很晚才实现 OOA 的诱惑是很大的。

例如，考虑给类分配方法的问题。传统规格说明阶段的一个任务是确定目标产品的数据

和行为，然而，给某一模块分配各种行为应当推延至设计阶段进行，因为如 11.16 节中所指出的那样，我们首先必须决定产品从整体上如何拆解为模块。

然而，在面向对象范型中，这后一个任务是分析流的一部分。即，在面向对象分析流期间，我们确定模块（类）和它们的相互作用，结果在类图中描述。因此，显然我们没有明显的理由等到面向对象设计流才给类分配方法。

尽管如此，记住面向对象分析是一个迭代的过程仍很重要。在求精各种模型的过程中，常常需要重新组织大部分的类图，重新分配方法就会产生不必要的附加工作。

在 OOA 过程的每一步中，减少在迭代期间重新组织的信息是一个好主意。因此，不管在面向对象分析流期间向前迈出一小步的诱惑如何大，为类分配方法应当等到设计流再进行。

## 本章回顾

本章介绍了面向对象分析（见 12.1 节），在 12.2 节描述了抽象实体类。该技术然后应用于电梯问题实例研究（12.3 节）。在 12.4、12.5 和 12.6 节中分别介绍了功能建模、实体类建模和动态建模。接下来在 12.7 节中讲述了测试流的面向对象分析方面。边界抽象和控制类是 12.8 节的主题。Osbert Oglesby 实例研究的面向对象分析分别在以下几节中讲述：12.9 节（初始功能模型）、12.10 节（初始类模型）、12.11 节（初始动态模型）、12.12 节（边界类抽象）、12.13 节（控制类抽象）、12.14 节（用例求精）、12.15 节（用例实现）、12.16 节（类图递增）、12.17 节（测试流）。在 12.18 节中讨论了统一过程的规格说明图。与动作者和用例有关的额外信息见 12.19 节。在 12.20 节中描述了面向对象分析的 CASE 工具。本章结束时讨论了面向对象分析流面临的挑战（12.21 节）。

## 进一步阅读

早期描述各种面向对象分析方法的书包括［Coad and Yourdon，1991a；Rumbaugh et al.，1991；Shlaer and Mellor，1992；and Booch，1994］。就像本章中提到的那样，这些技术（以及其他这里没有列出的技术）基本上相似。

除了这类面向对象分析技术之外，融合（fusion）［Coleman et al.，1994］是一种第二代 OOA 技术，它结合（或融合）了一些第一代技术，包括 OMT［Rumbaugh et al.，1991］和 Objectory［Jacobson，Christerson，Jonsson，and Overgaard，1992］。统一软件开发过程（The Unified Software Development Process）结合了 Jacobson、Booch 和 Rumbaugh 的工作［Jacobson，Booch，and Rumbaugh，1999］，Catalysis 是另一个重要的面向对象方法［D'Souza and Mills，1999］。

ROOM 是一个用于实时软件的面向对象方法［Selic，Gullekson，and Ward，1995］。有关实时的面向对象技术的进一步信息可以在［Awad，Kuusela，and Ziegler，1996］中找到。

有关 UML 的进一步细节可见［Booch，Rumbaugh，and Jacobson，1999］。《Communications of the ACM》杂志 1999 年 10 月刊包含大量各种各样的有关 UML 使用的文章。UML 现在在对象管理小组（Object Management Group）的控制之下，在 OMG 的网站 www.omg.org 上可发现最新版本的 UML。

本章中用来抽象候选类的名词抽象技术在［Juristo，Moreno，and Lopez，2000］中形式化地给出，CRC 卡片在［Beck and Cunningham，1989］中第一次提出，［Wirfs-Brock，

Wilkerson, and Wiener, 1990] 是一个关于 CRC 卡片的较好的信息来源。

已经出版的一些文章对面向对象分析技术进行了比较，包括 [de Champeaux and Faure, 1992; Monarchi and Puhr, 1992; and Embley, Jackson, and Woodfield, 1995]。面向对象和结构化技术的比较出现在 [Fichman and Kemerer, 1992] 中。

[Williams, 1996] 描述了在面向对象项目中对迭代的管理，状态图在 [Harel and Gery, 1997] 中进行了描述。[Bellinzona, Fugini, and Pernici, 1995] 描述了面向对象范型中的规格说明的重用。[Kazman, Abowd, Bass, and Clements, 1996] 提出运用策略帮助进行面向对象分析。

各种有关面向对象软件的形式化技术的论文出现在《IEEE Transactions on Software Engineering》杂志 2000 年 7 月刊中。

## 习题[⊖]

12.1　通过为图 12-10 中所示的其他类开发状态图，完成电梯问题的实例研究。

12.2　为什么 11.6 节中的有穷状态机的形式不能不加改变地用在面向对象分析中？

12.3　在面向对象分析的过程中可以引入类而不会反过来影响项目，关于这个过程的最新的观点是什么？

12.4　在面向对象范型中可以有目的地引入类，关于它的最早的观点是什么？

12.5　使用一个形式而不是本章所描述的状态图来表示动态模型，这是否可能？解释你的答案。

12.6　为什么在面向对象分析期间，决定的是类的属性而不是类的方法？

12.7　在 12.5.1 节中描述了名词抽象过程，我们为什么不也抽象动词？对于句子的其他构成部分（形容词，副词，连接词，介词，前置词，代名词）如何？

12.8　给出图 12-12 的用例 Sell a Painting 的情景。

12.9　给出图 12-12 的用例 Produce a Report 的情景。

12.10　给出图 12-12 的用例 Update a Fashionability Coefficient 的扩展情景。

12.11　给出图 12-29 的用例 Buy a Masterwork 的描述，尽可能提供多的细节。

12.12　给出图 12-29 的用例 Buy Other Painting 的描述，尽可能提供多的细节。

12.13　给出图 12-29 的用例 Produce a Purchases Report 的描述，尽可能提供多的细节。

12.14　给出图 12-29 的用例 Produce a Sales Report 的描述，尽可能提供多的细节。

12.15　给出图 12-29 的用例 Produce a Future Trends Report 的描述，尽可能提供多的细节。

12.16　为习题 12.12 的用例 Buy Other Painting 给出一个扩展的情景。

12.17　为习题 12.16 你的扩展情景的通常部分的实现画出协作图。利用图 12-39 的类图。

12.18　为习题 12.16 你的扩展情景的通常部分的实现画出顺序图。利用图 12-39 的类图。

12.19　给出习题 12.17 和 12.18 的你的交互图的事件流。

12.20　给出图 12-29 中显示的 Osbert Oglesby 软件产品的用例 Produce a Future Trends Report 的扩展情景。

12.21　为习题 12.20 你的情景的实现画出协作图。利用图 12-45 的类图。

---

⊖ 习题 11.16（学期项目）和习题 11.20 以及习题 11.21（实例研究）可以在第 11 章或第 12 章末做。

12.22　为习题 12.20 你的情景的实现画出顺序图。利用图 12-45 的类图。

12.23　给出习题 12.21 和 12.22 的你的交互图的事件流。

12.24　（分析和设计项目）执行习题 8.8 的该产品的分析流，确定是否银行声明是正确的。

12.25　（分析和设计项目）执行习题 8.7 的图书馆软件产品的分析流。

12.26　（分析和设计项目）执行习题 8.9 的自动柜员机的分析流。不需要考虑构成的硬件组件的细节，如读卡器、打印机、点钞机等。但要简单假定当 ATM 向那些组件发送命令时，它们能够正确执行。

12.27　（学期项目）执行附录 A 中描述的 Ophelia's Oasis 产品的分析流。

12.28　（实例研究）向 Ophelia's Oasis 实例研究（12.9 节到 12.16 节）的分析流中加入类 Report。这是一项改进还是一项不必要的麻烦？

12.29　（实例研究）确定当面向对象分析始于动态建模时，会发生什么？从图 12-24 的状态图开始，完成 Osbert Oglesby 实例研究的面向对象分析过程。

12.30　（实例研究）将 11.14 节的结构化系统分析与 12.9 节到 12.16 节的分析流进行对比。

12.31　（软件工程读物）指导教师将发给你们［Juristo，Moreno，and Lopez，2000］的复印件，你对他们的面向对象分析方法有什么意见？

## 参考文献


[Awad, Kuusela, and Ziegler, 1996] M. AWAD, J. KUUSELA, AND J. ZIEGLER, *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion,* Prentice Hall, Upper Saddle River, NJ, 1996.

[Beck and Cunningham, 1989] K. BECK AND W. CUNNINGHAM, "A Laboratory for Teaching Object-Oriented Thinking," Proceedings of OOPSLA '89, *ACM SIGPLAN Notices* **24** (October 1989), pp. 1–6.

[Bellinzona, Fugini, and Pernici, 1995] R. BELLINZONA, M. G. FUGINI, AND B. PERNICI, "Reusing Specifications in OO Applications," *IEEE Software* **12** (March 1995), pp. 656–75.

[Booch, 1994] G. BOOCH, *Object-Oriented Analysis and Design with Applications,* 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.

[Booch, Rumbaugh, and Jacobson, 1999] G. BOOCH, J. RUMBAUGH, AND I. JACOBSON, The UML Users Guide, Addison Wesley, Reading, MA, 1999.

[Coad and Yourdon, 1991a] P. COAD AND E. YOURDON, *Object-Oriented Analysis,* 2nd ed., Yourdon Press, Englewood Cliffs, NJ, 1991.

[Coleman et al., 1994] D. COLEMAN, P. ARNOLD, S. BODOFF, C. DOLLIN, H. GILCHRIST, F. HAYES, AND P. JEREMAES, *Object-Oriented Development: The Fusion Method,* Prentice-Hall, Englewood Cliffs, NJ, 1994.

[D'Souza and Wills, 1999] D. F. D'SOUZA AND C. WILLS, *Objects, Components, and Frameworks with UML: The Catalysis Approach,* Addison Wesley, Reading, MA, 1999.

[de Champeaux and Faure, 1992] D. DE CHAMPEAUX AND P. FAURE, "A Comparative Study of Object-Oriented Analysis Methods," *Journal of Object-Oriented Programming* **5** (March/April 1992), pp. 21–33.

[Embley, Jackson, and Woodfield, 1995] D. W. EMBLEY, R. B. JACKSON, AND S. N. WOODFIELD, "OO Systems Analysis: Is It or Isn't It?" *IEEE Software* **12** (July 1995), pp. 18–33.

[Fichman and Kemerer, 1992] R. G. FICHMAN AND C. F. KEMERER, "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique," *IEEE Computer* **25** (October 1992), pp. 22–39.

[Harel and Gery, 1997] D. HAREL AND E. GERY, "Executable Object Modeling with Statecharts,"

*IEEE Computer* **30** (July 1997), pp. 31–42.

[Jacobson, Booch, and Rumbaugh, 1999] J. RUMBAUGH, G. BOOCH, AND I. JACOBSON, *The Unified Software Development Process,* Addison Wesley, Reading, MA, 1999.

[Jacobson, Christerson, Jonsson, and Overgaard, 1992] I. JACOBSON, M. CHRISTERSON, P. JONSSON, AND G. OVERGAARD, *Object-Oriented Software Engineering: A Use Case Driven Approach,* ACM Press, New York, 1992.

[Juristo, Moreno, and López, 2000] N. JURISTO, A. M. MORENO, AND M. LÓPEZ, "How to Use Linguistic Instruments for Object-Oriented Analysis," *IEEE Software* **17** (May/June 2000), pp. 80–89.

[Kazman, Abowd, Bass, and Clements, 1996] R. KAZMAN, G. ABOWD, L. BASS, AND P. CLEMENTS, "Scenario-Based Analysis of Software Architecture," *IEEE Software* **13** (November/December 1996), pp. 47–55.

[Monarchi and Puhr, 1992] D. E. MONARCHI AND G. I. PUHR, "A Research Typology for Object-Oriented Analysis and Design," *Communications of the ACM* **35** (September 1992), pp. 35–47.

[Rumbaugh et al., 1991] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN, *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Rumbaugh, Jacobson, and Booch, 1999] J. RUMBAUGH, I. JACOBSON, AND G. BOOCH, *The Unified Modeling Language Reference Manual,* Addison Wesley, Reading, MA, 1999.

[Selic, Gullekson, and Ward, 1995] B. SELIC, G. GULLEKSON, AND P. T. WARD, *Real-Time Object-Oriented Modeling,* John Wiley and Sons, New York, 1995.

[Shlaer and Mellor, 1992] S. SHLAER AND S. MELLOR, *Object Lifecycles: Modeling the World in States,* Yourdon Press, Englewood Cliffs, NJ, 1992.

[USNO, 2000] "The 21st Century and the Third Millennium—When Will They Begin?" U.S. Naval Observatory, Astronomical Applications Department, at aa.usno.navy.mil/AA/faq/docs/millennium.html, February 22, 2000.

[Williams, 1996] J. D. WILLIAMS, "Managing Iteration in OO Projects," *IEEE Computer* **29** (September 1996), pp. 39–43.

[Wirfs-Brock, Wilkerson, and Wiener, 1990] R. WIRFS-BROCK, B. WILKERSON, AND L. WIENER, *Designing Object-Oriented Software,* Prentice-Hall, Englewood Cliffs, NJ, 1990.

# 第13章 设 计

**学习目标**

学完本章之后，你应当能够：

- 完成设计流；
- 完成面向对象设计；
- 完成数据流分析和事务分析。

在过去的40年左右时间里，已经提出了几百种设计技术。某些是对现存技术的变种，另一些则与任何先前提出的技术完全不同。一些设计技术已为成千上万的软件工程师所用，许多技术仅被它们的作者用过。某些设计策略，特别是那些由学院派开发出来的，具有坚实的理论基础。其他的设计策略，包括许多由学院派拟制的，更注重实际，因为它们的作者发现它们在实际工作中运行得很好。大多数设计技术是人工的，但是自动化正在逐渐成为设计的一个重要方面，只要它在文档管理中有帮助。

尽管设计技术很多，但其背后存在一定的模式。本书的一个主题是：一个产品有两个基本方面，一个是它的操作，一个是操作作用在其上的数据。因此，设计一个产品的两种基本方法是面向操作设计和面向数据设计。在**面向操作设计**（operation-oriented design）中，强调的是操作，例如数据流分析（13.3节），这里的目标是设计具有高内聚性的模块（7.2节）。在**面向数据设计**（data-oriented design）中，首先考虑的是数据。例如，在Jackson的技术（13.5节）中，首先确定数据结构，然后设计符合数据结构的过程。

面向操作设计技术的一个缺点是它们集中在操作方面，数据仅是次要的。同样，面向数据设计技术强调的是数据，低估了操作的作用。解决办法是使用面向对象技术，它对操作和数据给予同样的重视。本章首先描述面向操作和面向数据的设计，然后描述面向对象设计，恰恰因为对象将操作和数据结合在一起，因此面向对象设计结合了面向操作和面向数据设计的特性。因而，为了全面理解面向对象设计，需要对面向操作设计和面向数据设计有一个基本的认识。

在研究具体的设计技术之前，必须简要地谈及设计。

## 13.1 设计和抽象

传统的设计阶段由三个活动组成：结构化设计、详细设计和设计测试。设计过程的输入是规格说明文档，它描述了产品将要做“什么”。输出是设计文档，它描述了产品“如何”做才能达到这一点。

在**结构化设计**（又称为**概要设计**、**逻辑设计**或**高层设计**）期间，对产品进行模块化分解，即，仔细分析规格说明，产生具有需要的功能的模块结构。这个行为的输出是一个模块的列表，以及一个对于它们是如何相互连接的说明。从抽象的观点来看，在结构化设计期间，假定某些模块存在，然后根据那些模块开展设计。

然而当使用面向对象范型时，如1.9节所解释的那样，结构化设计行为是在面向对象分析阶段进行的（第12章）。这是因为分析流的第一步是明确类，因为类是模块的一种，某些

模块化的分解已经在分析阶段进行了。

传统设计阶段的下一个活动和面向对象设计工作流的主要行为是**详细设计**，也称为**模块化设计**、**物理设计**或**低层设计**，在此期间对每个模块（或类）进行详细设计。例如，选择特定的算法和数据结构。同样，从抽象的观点来看，在这个活动期间，将模块（或类）互连构成一个完整产品的事实被忽略了。

前面提到，设计阶段有三个活动，第三个活动是测试。使用“活动”（activity）一词而不是“阶段”（stage）或“步骤”（step），是为了强调测试是设计整体上的一个部分，就像它是整个软件开发和维护过程的一个完整部分一样。测试并不是仅在结构化设计和详细设计已经完成后进行的某项工作。类似地，在面向对象设计的情况下，测试流是与设计流同步完成的。

接下来介绍各种不同的设计技术，首先是面向操作技术，然后是面向数据技术，最后是面向对象技术。

## 13.2 面向操作设计

7.2节和7.3节举了一个理论上的实例，将一个产品分解成为具有高内聚和低耦合的模块。我们现在介绍达到这个设计目标的两个实用的传统技术：数据流分析（13.3节）和事务分析（13.4节）。理论上，只要规格说明可以用一个数据流图表示，就可以应用数据流分析，而且因为每个产品可以用一个DFD表示（至少在理论上是如此），数据流分析就普遍适用。然而实际上，在许多情形下，存在更合适的设计技术，特别是在数据流较之其他考虑是第二位时的产品设计中，情况更是如此。举出的其他设计技术的例子包括基于规则的系统（专家系统）、数据库以及事务处理产品。（13.4节中描述的事务分析，是一个将事务处理产品分解成模块的好方法。）

## 13.3 数据流分析

**数据流分析**（data flow analysis，DFA）是一项得到具有高内聚模块的传统设计技术。它可以和多数规格说明技术一同使用，这里，DFA与结构化系统分析一同给出（11.3节），该技术的输入是一个数据流图。关键是，一旦完成了DFD，软件设计者就有了关于产品的输入和输出的精确和完整的信息。

考虑一下图13-1的DFD表示的产品中的数据流。产品在某种程度上将输入转变为输出，在DFD中的某些点上，输入停止作为输入并且成为某种内部数据。然后，在接下来的某些点上，这些内部数据具有输出的性质。图13-2中进一步显示了这些细节，将那些输入失去作为输入的性质并且简单地变为由产品操作的内部数据的点，称为**输入的最高抽象点**（point of highest abstraction of input）。输出的最高抽象点类似，即是数据流图中输出可以被如此识别的第一点，而不是被识别为某种内部数据。




图13-1 显示产品的数据流和操作的数据流图

通过使用输入和输出的最高抽象点，将产品分解成为三个模块：input_module（输入模

块）、transform _ module（转换模块）和 output _ module（输出模块）。现在依次获得每个模块，找到它的最高抽象点，然后对模块再进行分解。对这个过程连续进行求精，直到每个模块执行单个操作，即该设计由具有高内聚的模块组成。这样，逐步求精（作为如此之多的其他软件工程技术的基础）也蕴含在数据流分析中。

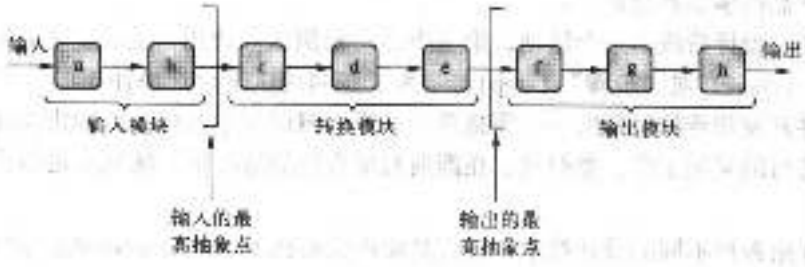



图 13-2　输入和输出的最高抽象点

应当公正地指出，可能需要对分解做出微小的修改，以获得最小可能的耦合。数据流分析是一个获得高内聚的方法，复合/结构化设计的目标是高内聚但同时还要低耦合。为了达到后者，有时有必要对设计做稍小的修改。例如，因为 DFA 没有将耦合考虑在内，在一个用 DFA 构建的设计中，稍不注意就会出现控制耦合。在这种情况下，所需要做的就是修改所涉及的两个模块，以便在它们之间传递数据而不是控制。

### 13.3.1　小型实例研究：字数统计

考虑设计一个产品的问题，它将一个文件名作为输入，并返回那个文件中的字数，就像 UNIX 中的 wc 实用程序。

图 13-3 描绘了数据流图，有 5 个模块。模块 read _ file _ name 读取文件名，然后通过 validate _ file _ name 对文件名进行确认，确认后的文件名送至 count _ number _ of _ words 模块，它精确地计算文件中的字数。字数统计送给 format _ word _ count 模块，格式化后的字数统计最后送到 display _ word _ count 模块来输出。

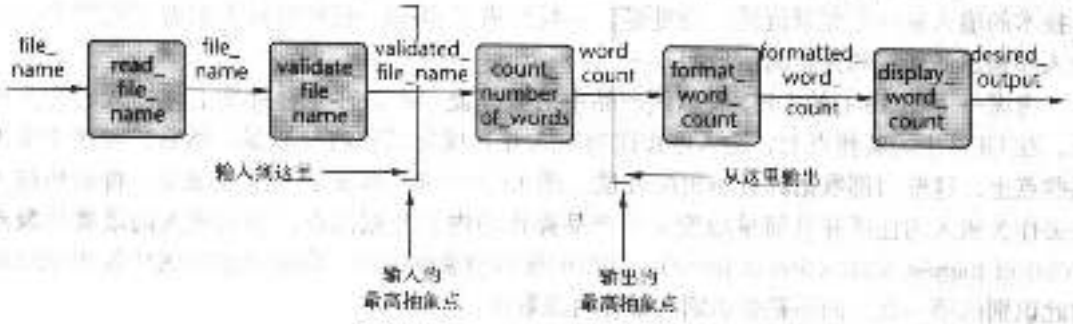



图 13-3　数据流图的第一次求精

再看数据流，初始输入是 file _ name，当这变成 validate _ file _ name 时，它仍是一个文件名，因此没有失去它作为输入数据的性质。但是考虑模块 count _ number _ of _ words，它的输入是 validated _ file _ name，它的输出是 word _ count。这个模块的输出在性质上完全不

同于整个产品的输入，显然输入的最高抽象点如图 13-3 所示。同样地，即使 count _ number _ of _ words 的输出经过某种格式化，它基本上是从模块 count _ number _ of _ words 中出现的输出。因此输出的最高抽象点示于图 13-3 中。

使用这两个最高抽象点分解产品的结果示于图 13-4 的结构图中。图 13-4 也揭示出图 13-3的数据流图在某种程度上过于简单了。如果由用户规定的文件不存在，DFD 没有显示出对应于所发生的事情的逻辑流。模块 read _ and _ validate _ file _ name 必须向 perform _ word _ count 模块返回一个 status _ flag。如果该名字无效，那么 perform _ word _ count 忽略它并打印一个错误消息，但是，如果该名字有效，将它传给 count _ number _ of _ words 模块。通常，在有条件数据流的地方，就需要有一个相应的控制流。

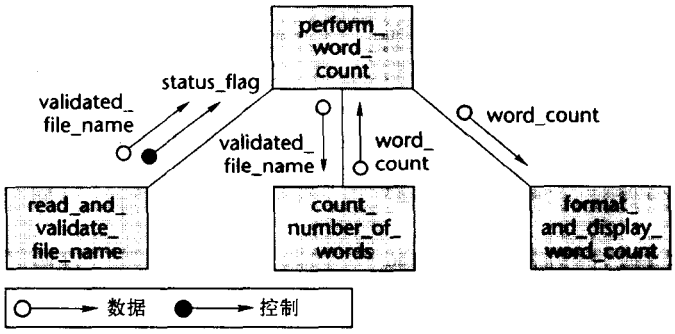



图 13-4 结构图的第一次求精

在图 13-4 中，两个模块具有通信性内聚（见 7.2.5 节），read _ and _ validate _ file _ name 和 format _ and _ display _ word _ count。必须对这些进一步分解，最后的结果示于图 13-5中。全部 8 个模块拥有功能性内聚，在它们之间或者有数据耦合，或者没有数据耦合。

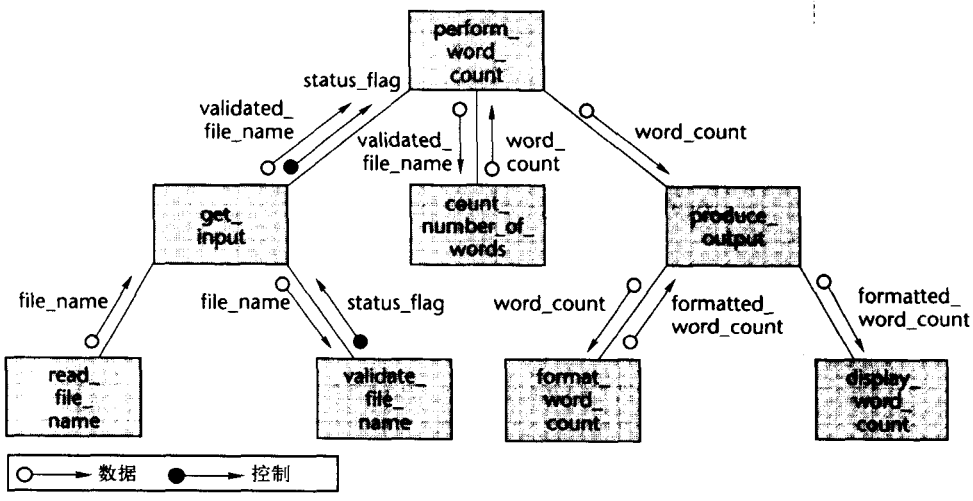



图 13-5 结构图的第二次求精

既然结构化设计已经完成，下一步就是详细设计。在这里选择数据结构和算法，然后每

个模块的详细设计交给程序员实现。与实际软件生产中的每个其他阶段一样，由于时间限制通常要求由一个小组而不是由单个程序员负责编写所有模块的代码。由于这个原因，必须给出每个模块的详细设计，使得不用参考其他模块就可以理解每个模块。图 13-6 中显示了 8 个模块中的 4 个模块的详细设计，其他 4 个模块以不同的形式给出。

| | |
|---|---|
| 模块名称 | read_file_name |
| 模块类型 | 函数 |
| 返回类型 | string |
| 输入参数 | 无 |
| 输出参数 | 无 |
| 错误消息 | 无 |
| 文件存取 | 无 |
| 文件修改 | 无 |
| 模块调用 | 无 |
| 描述 | 用户通过命令字符串 word_count <file_name> 调用本产品，通过使用一个操作系统调用，这个模块访问用户输入的命令字符串的内容，提取出<file_name>，然后作为该模块的值返回它。 |

a)

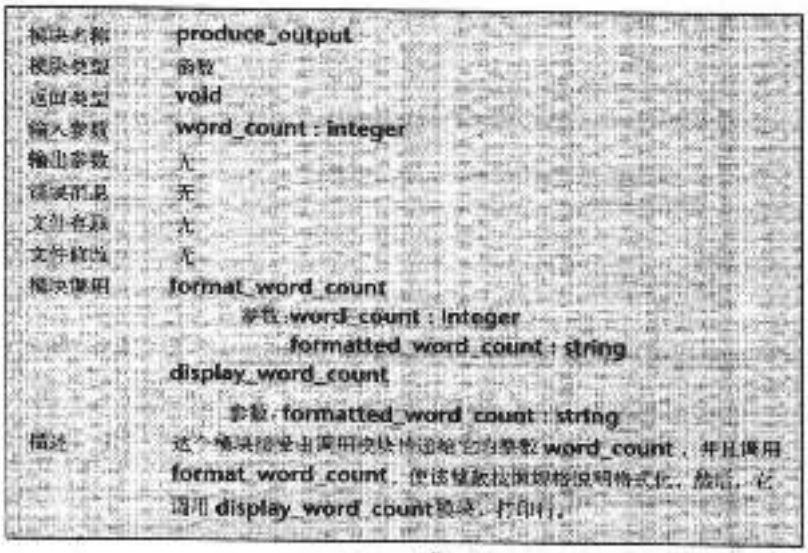
| | |
|---|---|
| 模块名称 | validate_file_name |
| 模块类型 | 函数 |
| 返回类型 | Boolean |
| 输入参数 | file_name : string |
| 输出参数 | 无 |
| 错误消息 | 无 |
| 文件存取 | 无 |
| 文件修改 | 无 |
| 模块调用 | 无 |
| 描述 | 这个模块产生一个操作系统调用，以确定文件 file_name 是否存在。如果该文件存在，模块返回 true，否则返回 false。 |

b)

| | |
|---|---|
| 模块名称 | count_number_of_words |
| 模块类型 | 函数 |
| 返回类型 | integer |
| 输入参数 | validated_file_name : string |
| 输出参数 | 无 |
| 错误消息 | 无 |
| 文件存取 | 无 |
| 文件修改 | 无 |
| 模块调用 | 无 |
| 描述 | 这个模块确定 validated_file_name 是否为一个文本文件，即，是分为字符行。如果是，模块返回文本文件中的字数，如果不是，模块返回 −1。 |

c)

图 13-6　例子的四个模块的详细设计




d)

图 13-6 （续）

图 13-6 的设计独立于编程语言，然而，如果管理者在详细设计开始前决定采用某一实现语言，使用**程序描述语言**（program description language，PDL）给出详细设计是一个吸引人的选择（**伪码**是 PDL 以前的名字）。PDL 本质上是由所选的实现语言的控制语句连接起来的注释组成的。图 13-7 显示了该产品的其余 4 个模块的详细设计，它是用带有 C++ 或 Java 风格的 PDL 编写的。PDL 的优点在于它通常是清晰和准确的，实现步骤常常仅由少数的从注释到相应的编程语言的翻译组成。缺点是有时存在这样一种倾向，设计员过分关注细节，生成了模块的完整代码实现，而不是进行一个 PDL 详细设计。

```
void perform_word_count()
{
  String          validate_file_name;
  int             word_count;

  if (get_input (validate_file_name) = false)
    print "error 1: file does not exist";
  else
  {
    set word_count equal to count_number_of_words (validate_file_name);
    if (word_count is equal to -1)
      print "error 2  file is not a text file";
    else
      produce_output (word_count);
  }
}

Boolean get_input (String validate_file_name)
{
```

图 13-7 例子的 4 个方法的详细设计的 PDL（伪码）表示

```
  String                file_name;

  file_name = read_file_name ();
  if (validate_file_name (file_name) is true)
  {
     set validate_file_ name equal to file_name;
     return true;
  }
  else
    return false;
}

void display_word_count (String formatted_word_count)
{
  print formatted_word_count, left justified;
}

String format_word_count (int word_count);
{
  return "File contains" word_count "words";
}
```

图 13-7　（续）

在将详细设计完全编成文档并成功测试之后，把它提交给实现小组进行编码，然后产品进入传统软件生命周期的其他阶段。

### 13.3.2　数据流分析扩展

读者可能会感到这个例子在某种程度上具有人为的成分，在那个数据流图中（图 13-3）只有一个输入流和一个输出流。为了理解在更复杂的情形下会发生什么情况，来看一下图 13-8，其中有 4 个输入流和 5 个输出流，这种情形与实际情况更相近。

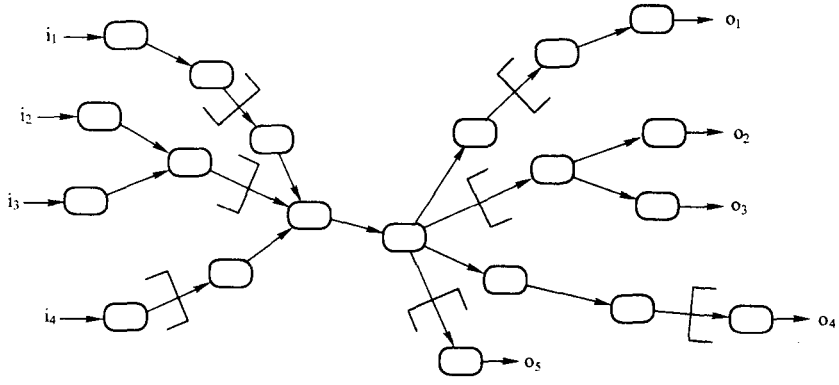



图 13-8　具有多个输入流和输出流的数据流图

当有多个输入和输出流时，处理的方法是对每个输入流找到输入的最高抽象点，对每个输出流找到输出的最高抽象点。通过使用这些点，用比原始状态少的输入－输出流将给定的数据流图分解成为模块。连续使用这种方法，直到得到的每个模块具有较高的内聚。最后，确定每对模块之间的耦合，并做必要的修改。

数据流分析概括于如下的“如何完成”部分中。

**如何完成数据流分析**

- **迭代**
  找到每个输入流的输入最高抽象点。
  找到每个输出流的输出最高抽象点。
  使用这些最高抽象点分解数据流图。
- **直到**得到的模块具有高内聚。
- 如果得到的耦合太紧，调整设计。

## 13.4 事务分析

**事务**（transaction）是从产品用户的观点来看的一个操作，如“处理一个请求”或“打印一份今天的订单的列表”。数据流分析对于事务处理类产品是不合适的，因为事务处理类产品必须完成一些相关的操作，它们大概相似但细节上不同。一个典型的例子是软件控制一台自动柜员机。顾客在槽中插入一张磁卡，键入一个口令字，然后执行动作，如向支票、存折或信用卡账户存款，或从账户提款，或查询账户的收支平衡等。这种类型的产品描述于图13.9中，设计这样的产品的一个好办法是将它分成两部分：分析器和分配器。分析器确定事务类型并将这个信息送到分配器，由分配器进行事务处理。

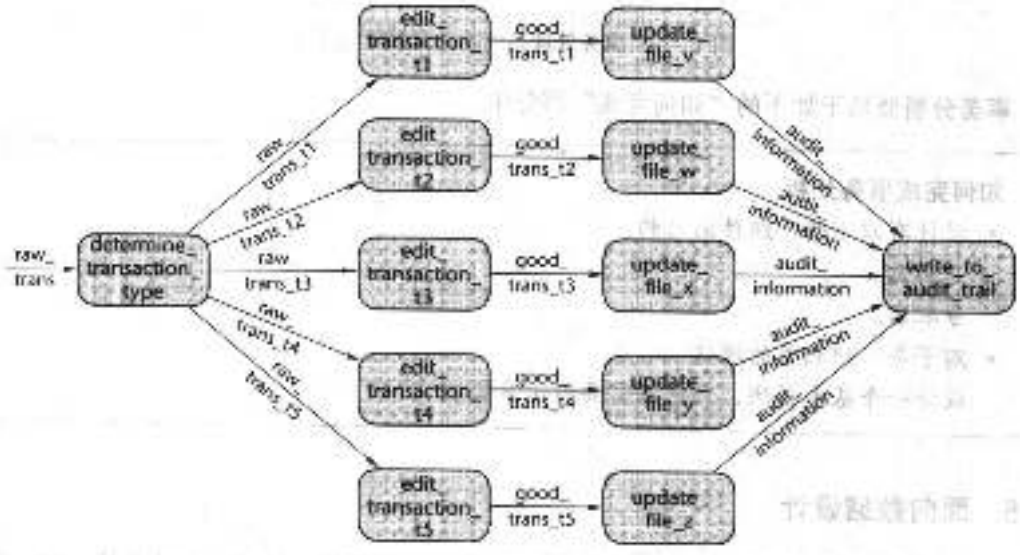



图13-9 典型的事务处理系统

图13-10中显示了这种类型的一个较差的设计，它有两个带逻辑性内聚的模块：edit_any_transaction和update_any_file。另一方面，需要5个非常相似的编辑模块和5个非常相似的更新模块，似乎是一种浪费。解决的办法是软件重用（8.1节）：设计、编码、编写文档并且测试一个基本的编辑模块，然后例示5次。每个版本会略有不同，但是差异很小，

值得使用这种方法。同样地，将基本的更新模块例示5次并略做修改，以适合5种不同的更新类型，得到的这个设计将是高内聚和低耦合的。

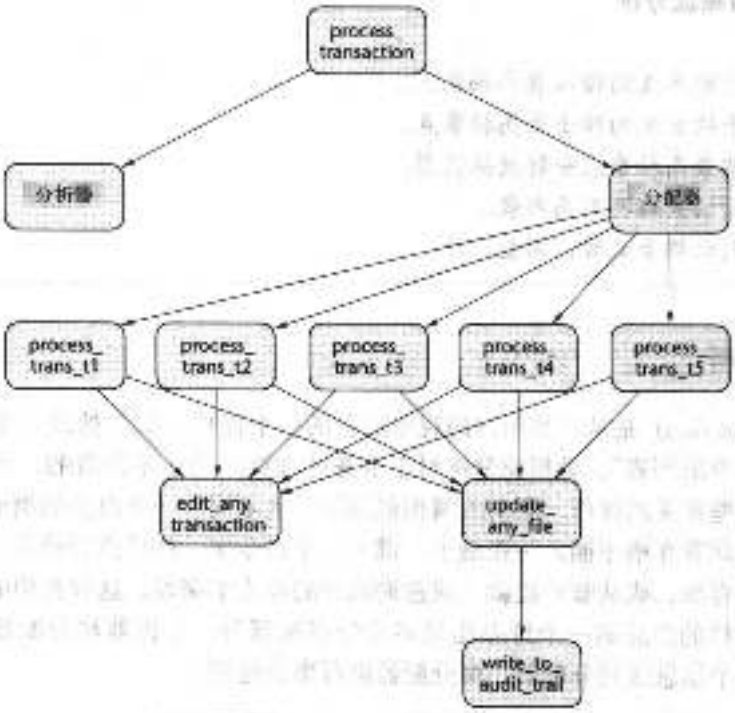



图13-10　事务处理系统的一个较差的设计

事务分析概括于如下的“如何完成”部分中。

**如何完成事务分析**

- 设计有以下两个组件的结构：
  分析器。
  分配器。
- 对于每一组相关的操作
  设计一个基本模块，并根据需要例示多次。

## 13.5　面向数据设计

面向数据设计背后的基本原则是根据在它上面运行的数据的结构来设计产品，即，首先确定数据的结构，然后赋予每个过程与它所操作的数据相同的结构。有许多这种类型的面向数据技术，最著名的是Michael Jackson［1975］、Warnier［1976］和Orr［1981］的技术，这三种技术有许多相似之处。

面向数据设计从来没有像面向操作设计那样流行过，而且随着面向对象范型的出现，它基本上已经落伍了。基于篇幅的原因，本书不对面向数据设计做进一步讨论，感兴趣的读者

可以参考前面段落中引用的文献。

## 13.6 面向对象设计

如前所述，统一过程的前提是知道面向对象设计（object-oriented design，OOD），因而我们现在描述 OOD，然后在 13.9 节中讨论统一过程的设计流。

OOD 的目标是按照对象设计产品，对象指的是，在面向对象分析期间提取的类和子类的实例。传统的语言像 C、COBOL 和 FORTRAN 不支持这样的对象，这看起来可能意味着 OOD 仅对于面向对象语言的用户可用，如 Smalltalk［Goldberg and Robson，1989］、C++［Stroustrup，2000］、Ada 95［ISO/IEC 8652，1995］和 Java［Flanagan，2002］。

但是情况并非如此，尽管传统语言不支持 OOD，却可以使用 OOD 的一个大子集。如 7.7 节所解释的那样，类是一个带有继承的抽象数据类型，而对象是类的一个实例。当使用一个不支持继承的实现语言时，解决办法是利用项目所使用的编程语言中能够得到的 OOD 的某些方面，即使用**抽象数据类型设计**。抽象数据类型实际上可以在支持 **type** 语句的任何语言中实现，即使在不支持这样的类型声明的传统语言中（因此它不支持抽象数据类型），仍有可能实现数据封装。图 7-26 描述了从模块开始、结束于对象的一个设计概念的层次，在完全 QOD 不可能的那些情况下，开发者应当努力确保他们的设计尽可能使用图 7-26 的层次中最高可能的概念，他们的实现语言支持这些概念。

OOD 的两个关键步骤是完成类图和进行详细设计。关于第一步——完成类图，需要确定属性的格式，给相关的类分配方法。通常情况下，从分析可以直接推导出属性的格式。例如，在美国规格说明可能声明像 1947 年 12 月 3 日这样的日期用 12/03/1947（mm/dd/yyyy 格式）表示，或者在欧洲用 03/12/1947（dd/mm/yyyy 格式）表示。但是不管使用哪一个日期协定，都需要 10 个字符。

在分析流期间可以得到确定格式的信息，因此这些格式当然可以在那里加入到类图中。然而，面向对象的范型是迭代的，每次迭代都给已经完成的部分带来一些修改。从实际角度出发，应尽可能晚地将信息加入到 UML 模型中。例如考虑图 12-17 到图 12-21，它们显示了 Osbert Oglesby 实例研究的类图的前 5 次迭代，这 5 个迭代都没有显示出类的属性。如果这些属性更早些被确定，可能必须调整它们，还有可能从类移动类，直到分析小组对类图满意为止。实际情况是需要调整的是类本身。通常情况下，在完全需要这样做之前向类图（或其他的 UML 图）添加一项没有什么意义，因为添加这项内容将成为下一次迭代不必要的累赘，特别是，在确实需要某些格式之前就规定这些格式完全没有必要。

OOD 的第一步骤的其他主要部分是给类分配方法（操作的实现）。对产品所有操作的确定通过检查每个情景的交互图来完成，这很容易理解。难于理解的部分是确定如何决定哪一个方法应与每个类相关。

方法可以分配给类或者客户，该客户给那个类的一个对象发送消息。（对象的客户是给那个对象发送消息的一个程序单元。）用来帮助确定如何分配操作的一个原则是信息隐藏（7.6 节），即，类的状态变量应声明为 **private**（只在那个类的一个对象内部可访问）或者 **protected**（只在那个类的一个对象或子类内部可访问）。因而运行在状态变量上的操作对于那个类而言必须是局部的。

第二个原则是，如果对象的一些不同的客户调用一个特定的操作，则应该把那个操作的

单个拷贝作为该对象的一个方法来实现，而不是在那个对象的每个客户里都拥有一个拷贝。

用来帮助确定方法所在位置的第三个原则是使用职责驱动设计。如1.9节所述，**职责驱动设计**是面向对象范型的关键方面。如果客户给对象发送消息，那么对象负责实现客户请求的每个方面。客户不知道请求是被如何实现的，也不允许客户知道。一旦实现了该请求，控制返回给客户。在那一点上，所有客户知道请求已被实现，但仍不知晓这是如何实现的。

为了明白如何应用这些原则，我们现在通过两个例子来阐述OOD。和以前一样，为简便起见，电梯问题实例研究只提供一个电梯的情形。然后我们再看Osbert Oglesby实例研究。通过使用相同的例子，你可以对比不同的方法，而不用担心问题本身的结果。

## 13.7　面向对象设计：电梯问题实例研究

**步骤1. 完成类图**

通过向图12-9的类图添加操作（方法）来得到一个设计流的类图（图13-11）。（在Java实现的情形下，需要两个额外的类，**ElevatorApplication**对应于C++的main函数，而**ElevatorUtilities**包含一些Java例程，它们与在C++类外部声明的C++函数相对应。）

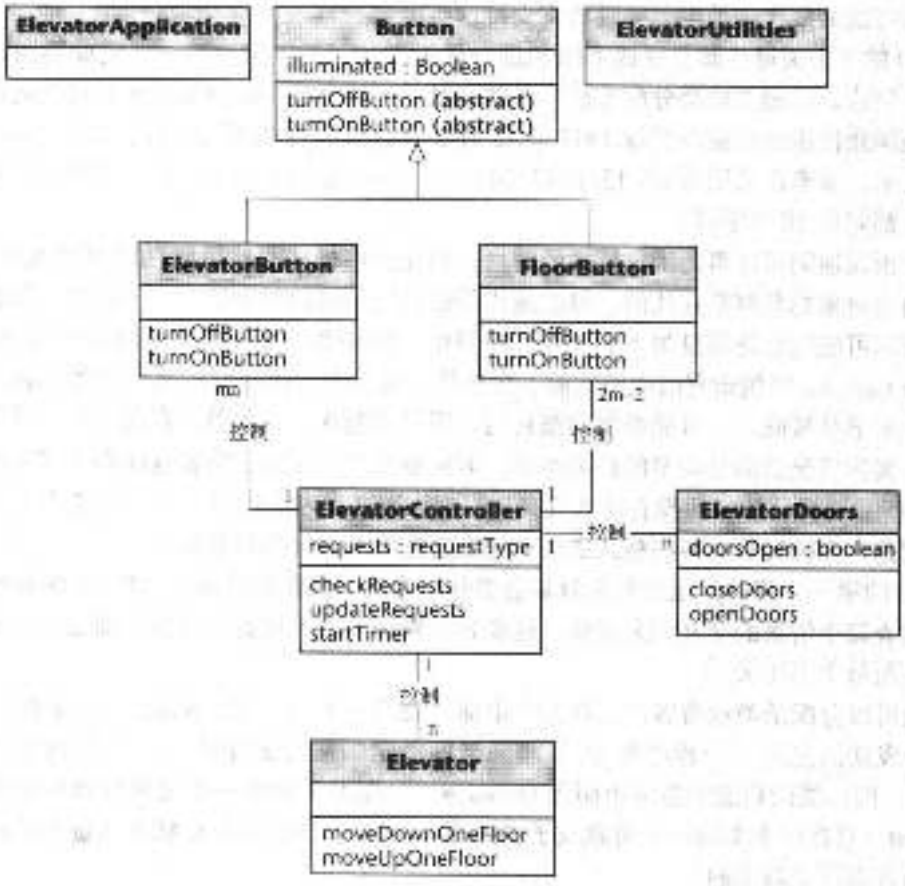



图13-11　电梯问题实例研究的详细类图

考虑电梯控制器的 CRC 卡片的第二次迭代（图 12-9），职责分为两组。在职责驱动设计的基础上，将三个职责“8. 启动定时器”、“10. 检查请求”和“11. 更新请求”分配给电梯控制器，由电梯控制器自己完成这些任务。

另一方面，剩下的 8 个职责（事件 1 到事件 7，以及事件 9）具有“向另一个类发送消息告诉它做某事”的形式，这意味着在给类安排相关的方法时，同样应当使用职责驱动设计的原则。此外，出于安全方面的考虑，信息隐藏的原则同样应用于全部 8 个事例中。

因为这两个原因，给类 **Elevator Doors** 分配了方法 close doors 和 open doors，即 **Elevator Doors** 的客户（在这种情况下，它是类 **Elevator Doors** 的一个实例）向类 **Elevator Doors** 的对象发送消息，来关闭或打开电梯的门，然后相关的方法完成那个请求。那两个方法的每个方面被封装在类 **Elevator Doors** 内部。此外，信息隐藏产生了一个真正独立的 **Elevator Doors** 类，它的实例能够独立地经历详细设计和实现，并且以后能够在其他产品中重用。

同样的两个原则应用于方法 move down one floor 和 move up one floor，并且将它们分配给类 **Elevator**。不需要一个明确的指令让电梯停下，如果不调用这两个方法中的任意一个，电梯不会移动。除了调用这两个方法中的某一个，没有其他改变电梯状态的方法。

最后，给 **Elevator Button** 和 **Floor Button** 分配方法 turn off button 和 turn on button，这里的推理与给类 **Elevator Doors** 和类 **Elevator** 分配方法一样。首先，职责驱动设计的原则要求按钮对它们是开还是关具有完全的控制；其次，信息隐藏的原则要求将按钮的内部状态隐藏起来。因此，打开或关闭一个电梯按钮的方法对于类 **Elevator Button** 必须是局部的，对于类 **Floor Button** 也是一样。为了利用多态（polymorphism）和动态绑定，由于 7.8 节所说的原因，在基类 **Button** 中，将方法 turn on button 和 turn off button 声明为抽象（虚拟）的。然后在运行期间就会调用方法 turn on button 的正确版本。

**步骤 2. 进行详细设计**

现在为所有的类进行详细设计。任何合适的技术都可以使用，如第 5 章中描述的逐步求精。方法 elevator event loop 的详细设计如图 13-12 所示，这里使用了 PDL（伪码），但是表格化的表示（如图 13-6 的表示）也同样有效。

```
void elevatorEventLoop (void)
{
  while (TRUE)
  {
    if (a button has been pressed)
      if (button is not on)
      {
        updateRequests;
        button::turnOnButton;
      }
    else if (elevator is moving up)
    {
      if (there is no request to stop at floor f)
        elevator::moveUpOneFloor;
      else
```

图 13-12 方法 elevator Event Loop 的详细设计

```
        {
          stop elevator by not sending a message to move;
          elevatorDoors::openDoors;
          startTimer;
          if (elevatorButton is on)
            elevatorButton::turnOffButton;
          updateRequests;
        }
      }
      else if (elevator is moving down)
        [similar to up case]
      else if (elevator is stopped and request is pending)
      {
        elevatorDoors::closeDoors;
        if (floorButton is on)
          floorButton::turnOffButton;
        determine direction of next request;
        elevator::moveUp/DownOneFloor;
      }
      else if (elevator is at rest and not (request is pending))
        elevatorDoors::closeDoors;
      else
        there are no requests, elevator is stopped with elevatorDoors closed, so do nothing;
    }
  }
```

图 13-12　（续）

图 13-12 是从图 12-6 的状态图得来的，例如，事件“按下按钮，按钮灯不亮”是在图 13-12 的开始用两个嵌套的 **if** 语句实现的，然后是状态 **Process Requests** 的两个操作，**else-if** 条件对应于 **Elevator Event Loop** 的下一个事件——电梯向方向 d 移动，楼层 f 是下一层。剩下的详细设计同样容易理解。

现在我们考虑 Osbert Oglesby 实例研究的面向对象设计。

## 13.8　面向对象设计：Osbert Oglesby 实例研究

如 13.6 节所描述的，面向对象的设计包括两个步骤。

**步骤 1　完成类图**

Osbert Oglesby 实例研究的最终类图如图 13-13 所示，用户定义的 **Date Class** 被画成虚线表示仅需要一个 C++ 实现；Java 具有内嵌类来处理日期，包括 **java.text.Dateformat** 和 **java.util.Calendar**。

接下来，从 10.9 节的需求中推导出类属性的格式，结果示于图 13-14 中。

产品的方法出现在各种交互图中，设计者的任务是决定每个方法应被分配给哪个类。例如，按面向对象的软件产品的惯例，与类的每个属性相关的是方法 setAttribute（用来给那个属性分配一个特定值）和方法 getAttribute（返回属性的当前值）。

例如，考虑用来给一个画作对象分配画作名称的方法 setTitle。在传统的范型中，我们需要函数 set _ masterpiece _ title、set _ masterwork _ title 和 set _ other _ painting _ title，然而面向对象的范型支持继承。所以，方法 setTitle 应分配给 **Painting Class**。然后如图 12-22

的继承树所示，该方法之后不仅可以用于类 **Painting Class** 的实例，还可以作为 **Painting Class** 的每个子类的实例，即作为类 **Gallery Painting Class**、**Auctioned Painting Class**、**Masterpiece Class**、**Other Painting Class** 和 **Masterwork Class** 的实例继承的结果。类似地，方法 getTitle 也应分配给超类 **Painting Class**。

分配其他的方法给合适的类同样容易理解，结果显示在附录 G 中。

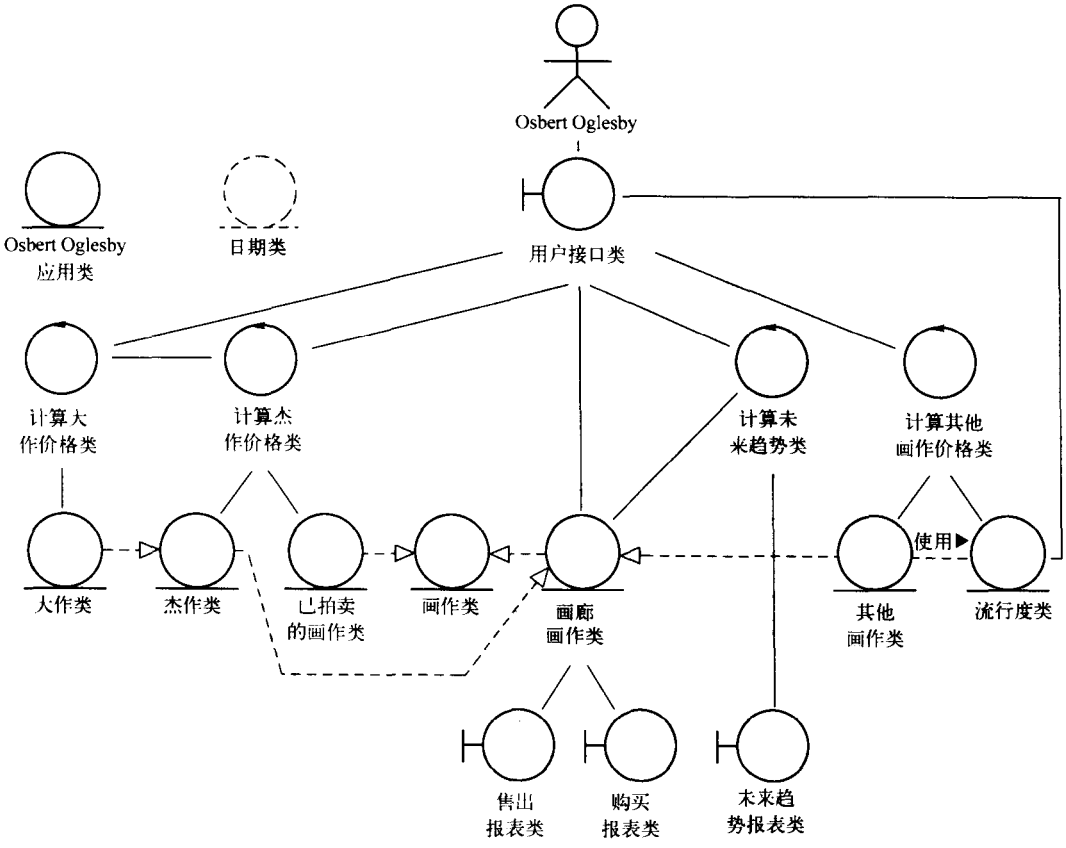



图 13-13 Osbert Oglesby 实例研究的整个类图

**步骤 2 进行详细设计**

接下来，通过确定每个方法做什么来进行详细设计。图 13-15 和图 13-16 显示了 Osbert Oglesby 实例研究的两个方法的详细设计（以 C++ 的 PDL 实现）。图 13-15 显示了类 **ComputeMasterpiecePrice** 的方法 getAlgorithmPrice 的详细设计，图 13-16 显示了类 **ComputeMasterworkPrice** 的方法 getAlgorithmPrice 的详细设计。

下面的“如何完成”部分总结了面向对象设计的步骤。

**如何完成面向对象设计**

- 完成类图
- 进行详细设计

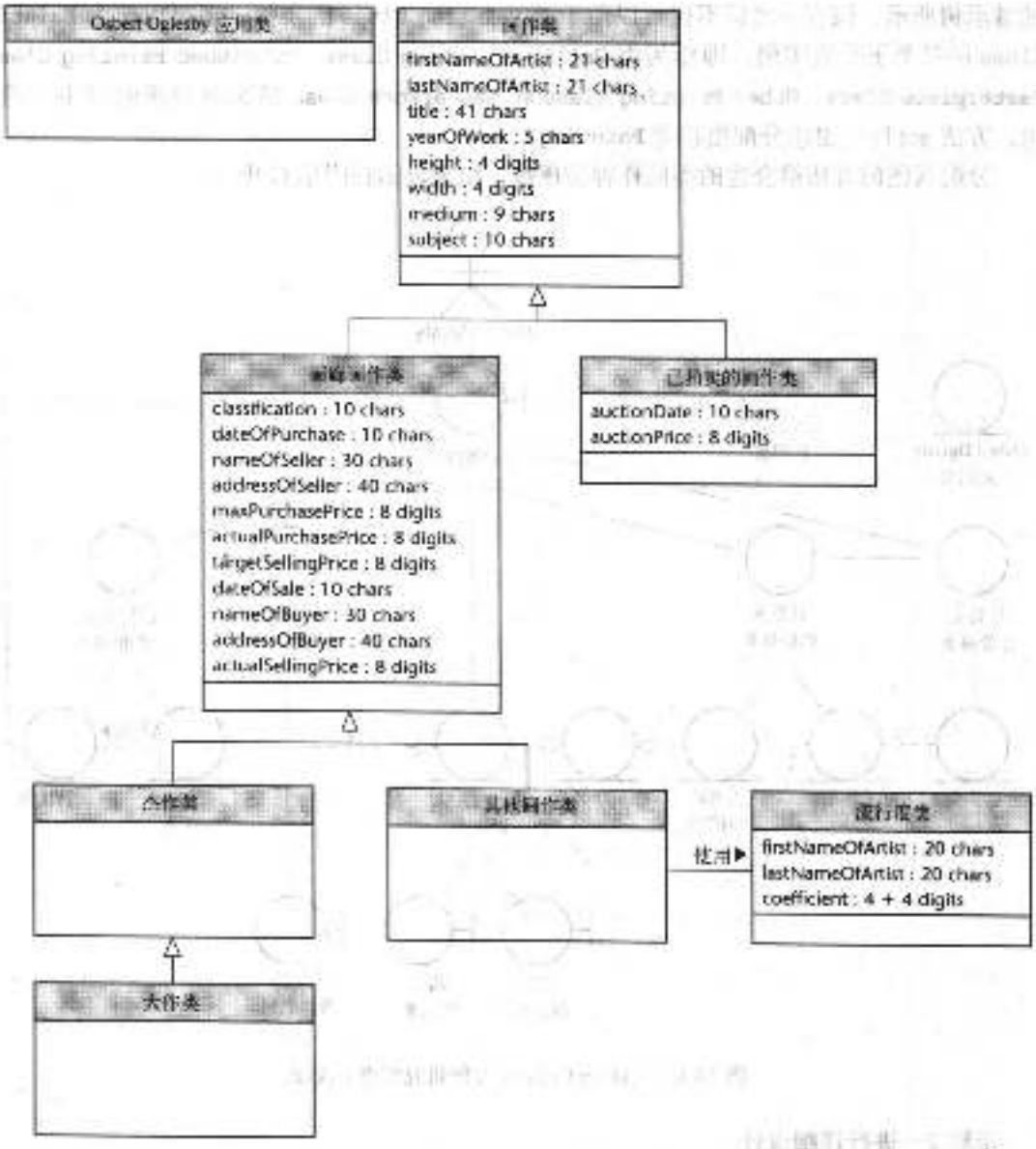



图 13-14　增加了属性格式的 Osbert Oglesby 实例研究的整个类图

```
float ComputeMasterpiecePrice::getAlgorithmPrice (GalleryPainting* const masterpiece)
determine the maximum price to be offered for a masterpiece as follows:
{
  float high;          (highest similarity so far)
  float algHigh;       (price of most similar work)
  float highDate;      (date of auction of most similar work)
  float temp;          (number of matches on medium/subject)
```

图 13-15　Osbert Oglesby 实例研究的类 `ComputeMasterpiecePrice` 的方法 getAlgorithmPrice 的详细设计

```
set high equal to 0.0;
set algHigh equal to 0.0;
set highDate equal to current date;

loop through all of the auction objects and find the most similar work as follows:
{
   for each auction object
   {
      set temp equal to 0.0;
      if (there is a match on firstName and lastName)
         compute the similarity as follows:
         {
            if (there is a match on medium)
               add 1 to temp;
            if (there is a match on subject)
               add 1 to temp;
            get the area of each painting by multiplying its height by its width;
            multiply temp by area of larger painting, divide by area of smaller painting;
            if (temp is greater than high)
            {
               set temp equal to high;
               set algHigh equal to auction price of this painting;
               set highDate equal to date of auction of this painting;
            }
         }
   }
}
adjust price for time since auction as follows:
   for each year since highDate.getYear()
      multiply algHigh by 1.085;
return (algHigh);
}//ComputeMasterpiecePrice::getAlgorithmPrice
```

图 13-15 （续）

```
float ComputeMasterworkPrice::getAlgorithmPrice (GalleryPainting* const masterwork)
determine the maximum price to be offered for a masterwork as follows:
{
  int century;                    (century in which painting was created)
  float masterpiecePrice;         (price if the painting were a masterpiece)
  first compute the price of the painting as if it were a masterpiece:
    masterpiecePrice = ComputeMasterpiecePrice::getAlgorithmPrice (masterwork);
  now obtain the century in which the painting was created and adjust the price
  based on the century, as follows:
    if (century is equal to 21)
      return masterpiecePrice * 0.25;
    else
      return masterpiecePrice * (21 – century)/(22 – century);
}//ComputeMasterworkPrice::getAlgorithmPrice
```

图 13-16 Osbert Oglesby 实例研究的类 **ComputeMasterworkPrice** 的方法 getAlgorithmPrice 的详细设计

## 13.9　设计流

**设计流**的输入是分析流的制品（第12章），在设计流期间，这些制品经过迭代和递增，直到它们的格式可以被程序员应用。

这种迭代和递增的一个方面是确认方法和给合适的类分配这些方法，另一个方面是进行详细设计。这两个步骤建立了设计流的面向对象设计组件。

除了进行面向对象设计，还需要在设计流中做出许多决策。一个决策是选择实现软件产品的编程语言，这个过程在第14章中进行了详细的讨论。另一个决策是待开发的新软件产品中能够重用多少现有的软件产品，第8章中描述了重用。可移植性是另一个重要的设计决策，这个话题在第8章中也有所描述。还有，大型的软件产品通常在计算机网络上实现，另一个设计决策是将每个软件组件分配给运行该软件的硬件组件。

开发统一过程背后的主要动机是提供一种方法论，用于开发大型软件产品，典型的情况是500 000行代码或更多。另一方面，附录H和附录I中的Osbert Oglesby实例研究的实现分别以C++和Java编写，程序少于5000行。换句话说，统一过程主要针对比本书中提供的Osbert Oglesby实例研究大至少100倍的软件产品，因此统一过程的许多方面不适用于这个实例研究。例如，分析流的一个重要部分是将软件产品分割成分析包，每个**包**由一组相关类组成，通常是一组行为者的小子集，可作为单个单元来实现。例如应付款、应收款和总账是典型的分析包。分析包暗示了开发小一点的软件比开发大一点的软件要更容易，因此，如果大型软件产品可以分解为相对独立的软件包，应更容易开发。

分解大型工作流为相对独立的小工作流的理念发扬到设计流中，这里，目标是将即将面临的实现流分成可管理的小块，称为**子系统**。当然，没有必要将Osbert Oglesby实例研究分成子系统，该实例太小了。

大型工作流分解为子系统有以下两个原因：

1）如前所述，实现一些更小的子系统比实现一个大系统要容易得多。

2）如果要实现的子系统是真正相对独立的，那么可以由编程小组并行地实现它们，这使得软件整体能够很快交付。

回想一下8.5.4节，软件产品的体系结构包括各种组件和这些组件如何配合在一起。给子系统分配组件是结构化任务的主要部分，对软件产品的体系结构做出决定无疑是容易的，并且在所有情况（除了最小的软件产品）中，都是由专家（软件设计师）来完成。

设计师除了是技术专家外，还需要知道如何提出**折衷办法**。软件产品需要满足功能要求，包括可移植性（第8章）、可靠性（6.4.2节）、健壮性（6.4.3节）、可维护性、安全性等。但这一切需要在预算和时间限制内进行，几乎不可能开发出满足所有（功能性的和非功能性的）需求并在成本和时间限制内完成的软件产品，总是要做出妥协——客户放宽一些需求、增加预算或者延迟交付期限，或者客户做这些妥协中的多个妥协。设计师必须通过清楚地勾画出折衷办法来帮助客户进行决策。

在一些情况下折衷办法是显而易见的，例如，设计师可能指出，一系列的符合新的高安全性标准的安全性需求要花费超过3个月和350 000美元才能体现在软件产品中。如果产品是一个国际银行网络，这个事情是没有实际意义的——绝没有用户可能同意在安全性方面做出妥协。然而，在其他的实例中，客户需要对有关折衷办法做出重要确认，并且依靠设计师

的技术经验来做出正确的商业决策。例如，设计师可能提出延缓一个特定需求，直到软件产品可被交付，而且维护起来现在可能会节省 150 000 美元，但以后合并起来需要花费 300 000美元（参见图 1-5）。是否延缓一个需求的决策只可由客户做出，但他或她需要设计师的专业意见来帮助做出正确的决策。

软件产品的体系结构是决定交付的产品成功与否的重要因素，而且关于体系结构的关键决策必须在进行设计流时做出。如果需求流完成得不好，假如在分析流时投入额外的时间和资金，仍有可能使项目成功。类似地，如果分析流完成得不充分，也有可能通过在设计流中做额外的努力来得到恢复。但是，如果体系结构是未达到最佳标准的，则没有办法恢复；体系结构必须立即重新设计，因此开发小组（包括设计师）具备必要的专业经验非常重要。

## 13.10 测试流：设计

测试设计的目标是验证规格说明已准确和完整地体现到设计中，并确保设计本身的正确性。例如，设计必须没有逻辑错误，必须正确定义所有接口。在代码生成之前检测出设计中的任何错误非常重要，否则，修复这些错误的成本会更高，如图 1-5 所示。设计错误可通过设计审查和设计走查的方法进行检测。设计审查在本节的剩余部分进行讨论，但其中的注解同样可应用于设计走查。

当产品是面向事务的（13.4 节），设计审查应反映出这一点［Beizer，1990］。应计划出包含所有可能的事务类型的审查，评审者应使设计中的每个事务与规格说明发生联系，显示出事务是如何从规格说明文档中生成的。例如，如果应用是自动柜员机，事务则对应于顾客可能进行的每个操作，如向信用卡账户存款或取款。在其他实例中，规格说明和事务之间的对应关系没有必要是一对一的，例如在交通灯控制系统里，如果一辆驶过传感器板的汽车导致系统决定在 15 秒后将红灯变为绿灯，那么来自传感器的更多的脉冲将被忽略。相反，为了加速交通流，一个脉冲就可能引起交通灯从红灯变为绿灯。

将评审限于**事务驱动的审查**将无法检测出下列情况：设计者忽略规格说明要求的事务实例。举个极端的例子，交通灯控制器的规格说明可能规定，在晚上 11:00 到早上 6:00 之间，一个方向上的所有灯为黄色，另一方向上的所有灯为红色。如果设计者忽略这个约定，那么在晚上 11:00 和早上 6:00 由时钟生成的事务将不会被包含在设计中；如果这些事务被忽略，它们则不能够在基于事务的设计审查中被检测到。因此，只安排事务驱动的设计审查是不够的，规格说明驱动的审查也很重要，它可确保规格说明文档中没有任何语句被忽视或误解。

## 13.11 测试流：Osbert Oglesby 实例研究

现在显然设计已经完成，Osbert Oglesby 实例研究的设计的所有方面必须通过设计审查（6.2.3 节）来进行检查。特别地，必须检查每个设计制品，即使没有找到错误，也可能在实现 Osbert Oglesby 实例研究时再次修改设计，有可能是根本性的修改。

## 13.12 详细设计的形式化技术

前面已经给出了详细设计的一个技术，在 5.1 节中，给出了逐步求精的描述，然后使用流程图（flowchart），将它应用于详细设计。除了逐步求精，还可以使用形式化技术来优化

详细设计。第6章指出，实现一个完整的产品，然后再证明它的正确性可能达不到预期目标，然而，证明和详细设计并行进行，同时也仔细测试代码则完全是另一回事。将形式化技术应用于详细设计在三个方面大有帮助：

1）正确性证明的最新成果是，尽管正确性证明通常不能应用于整个产品，它可以应用于产品的模块级。

2）与详细设计一同开发正确性证明，比起不使用正确性证明来，会使设计中出现的错误减少。

3）通常，如果同一程序员既负责详细设计又负责实现，那么那个程序员会十分自信地认为详细设计是正确的，这种对设计的积极的态度会使代码中出现较少的错误。

## 13.13　实时设计技术

如6.4.4节所解释的那样，**实时软件**的特点是有严格的时间限制，即，如果这样一个时间受限的特性没有满足，那么信息就会丢失。特别是，每一个输入必须在下一个输入到达前处理完毕。这类系统的例子是一个计算机控制的核反应堆。像内核温度和反应舱中水的高度等输入连续不断地送到计算机中，计算机读取每个输入的值并在下一个输入到达前进行必要的处理。另一个例子是一间计算机控制的特护病房。有两类病人数据：一类是例行信息，如每个病人的心跳速率、体温和血压；另一类是紧急信息，它是在系统推断出一个病人的状况变得严重的时候输出的信息。当出现紧急情况时，软件必须处理从一个或多个病人来的例行信息输入和与紧急情况有关的信息输入。

许多实时系统的一个特点是它们在分布式硬件上实现。例如，控制战斗机的软件可能在5台计算机上实现：一台掌管航行，另一台控制武器系统，第三台用于电子对抗，第四台用于控制像机翼的襟翼和发动机这样的飞行硬件，第五台用于在战斗中进行战术安排。因为硬件不是完全可依赖的，因此必须有额外的备份计算机自动替代一个故障单元。这样一个系统的设计不仅有主要的通信含意，也有时间问题，在前面描述的那些类型的问题之上，作为系统的分布式特性的结果，还会产生一些其他问题。例如，在作战条件下，战术计算机可能建议飞行员爬升，而武器计算机可能建议飞行员俯冲，以便某一武器可以在最佳状态下发射。但是，飞行员本人决定向右拉操纵杆，因此向航行硬件计算机发送一个信号做必要的调整，以使飞机向指示的方向倾斜。必须以这样一种方式谨慎地管理全部这些信息，以使飞机的实际动作在各个方面优先于建议的动作。进一步地，实际的动作必须传达给战术和武器计算机，使其能够根据实际情况形成新的建议。

实时系统更进一步的困难在于同步。假定将在分布式硬件上实现一个实时系统，当两个动作都对一个数据项独占使用，并且每个请求都独占地使用另外一个数据项的时候，就会发生像死锁（或死环绕）这样的情形。当然，死锁不仅仅发生于在分布式硬件上实现的实时系统中，但是，在实时系统中它特别棘手，在那里对输入的顺序或定时没有控制，而且硬件的分布式特性也会使问题复杂化。除了死锁，也可能出现其他同步问题，包括竞争条件。有关细节读者可以参考［Silberschatz，Galvin，and Gagne，2002］或其他操作系统教科书。

从这些例子中可以清楚地看出，关于实时系统设计的主要困难是确保设计满足时间限制。即设计技术应当提供一个检查机制，当实现时，设计能够以要求的速度读取和处理输入数据。进一步地，它应当能够显示设计中的同步问题也已得到正确的解决。

自打计算机时代一开始，几乎在每个方面，计算机硬件技术的发展步伐都远远超过计算机软件。因此，尽管硬件需要去处理前面提到的实时系统的每个方面，软件设计技术已经大大地拖后了。在实时软件工程的某些领域，已经取得了重大的进步。例如，第11章和第12章的许多规格说明技术可以用于规定实时系统，遗憾的是，软件设计还远未达到同样的复杂水平。目前确实有了大幅度的进步，但是它的最新发展水平与规格说明技术已达到的状态相比，还差很多。因为实时系统的任何设计技术几乎都比根本没有技术更可取，因此在实际中还是应用了一些实时设计技术。但是，在能够设计像前面所描述的实时系统之前还有很长的路要走，我们暂时还不能确保在系统实现之前，每个实时限制都会满足，并且不会出现同步问题。

早期实时设计技术是非实时技术在实时领域的扩展。例如，实时系统的结构化开发（structured development for real-time systems，SDRTS）［Ward and Mellor，1985］本质上是结构化系统分析（11.3节）、数据流分析（11.3节）和事务分析（13.4节）扩展到实时软件。在软件开发技术中，实时设计是作为它的一个组成部分包括在其中的。更新的技术在Liu［2000］和Gomaa［2000］中有描述。

如前所述，很遗憾，实时设计的技术状态没有达到期望的进展程度。尽管如此，人们仍采取了很大的努力来改进现状。

## 13.14 设计阶段的CASE工具

如13.10节所述，设计阶段的一个重要方面是测试设计成果是否精确地体现了规格说明文档的各个方面。因此需要一个CASE工具，它既可用于规格说明文档，又可用于设计，称其为前端或高端CASE工具（与有助于实现的后端或低端CASE工具相对应）。

市场上售有一些高端的CASE工具，一些较流行的工具有Analyst/Designer、Software through Pictures和System Architect。高端CASE工具通常围绕数据字典建立，CASE工具可以检查字典中每个记录的每个字段在设计文档中的某处被提及，它还可以检查设计文档中的每一项反映在数据流图中。此外，许多高端CASE工具结合了一个一致性检查器，它使用数据字典确定设计中的每一项已经在规格说明文档中声明过，同时，确定规格说明中的每一项出现在设计中。

进一步地，许多高端CASE工具结合了屏幕和报表生成器，即客户能够规定哪些项出现在报表或出现在输入屏上，以及每一项在哪里和怎样出现。因为关于每一项的全部细节在数据字典中，按照客户的意愿，CASE工具能够很容易地生成用于打印报表或显示输入屏幕的代码。某些高端CASE产品还结合了用于估算和计划的管理工具。

关于面向对象设计，Together、Rose和Software through Pictures提供了在完整的面向对象生命周期的范围内，对这个工作流的支持。这种类型的开放源代码的CASE工具包括ArgoUML。

## 13.15 设计阶段的度量

有各种度量可以用于描述设计的各方面特性。例如，代码制品（模块或类）数是对目标产品大小的一个粗略的度量；模块内聚和耦合像错误统计一样，是对设计质量的度量。对于所有其他类型的审查，至关重要的是，保存在设计审查期间检测出的设计错误的数量和类

型，这个信息在产品的代码审查期间和后续产品的设计审查期间要用到。

一个详细设计的**秩复杂度** $M$ 是二元判定（谓词）数加1［McCabe，1976］，或者，等效于模块中的分支数。有人曾建议将秩复杂度作为设计质量的度量：$M$ 的值越小，设计质量越好。这个度量的一个优点是它容易计算，然而，它有一个固有的问题，秩复杂度只是对控制复杂度的测量，忽略了数据复杂度，即 $M$ 没有测量出像表中的值这样的数据驱动的模块的复杂度。例如，假定一个设计者不知道 C++ 库函数 toascii，他从头开始设计了一个这样的模块——读取用户输入的字符并返回相应的 ASCII 码（在 0 和 127 之间的整数）。设计方法之一是使用一个依靠 **switch** 语句实现的 128 路分支，第二个方法是用一个包含按 ASCII 代码顺序排列的 128 个字符的数组，并利用一个循环来将用户输入的字符与字符数组的每一元素进行比较，当得到一个匹配时就退出循环。那么循环变量的当前值就是相应的 ASCII 码。这两个设计在功能上是等效的，但是，分别具有的秩复杂度却是 128 和 1。

当使用传统范型时，设计阶段的一个相关类的度量是以将结构化设计表示为一个有向图为基础的，它用结点表示模块，用弧线表示模块间的流（过程和函数调用）。一个模块的**扇入**（fan-in）可以定义为流入模块的流数，加上被模块访问的全局数据结构的数量；类似地，**扇出**（fan-out）是流出模块的流数，加上被模块更新的全局数据结构的数量。那么模块的复杂性度量就由“长度×（扇入×扇出）$^2$”给出［Henry and Kafura，1981］，这里**长度**是模块的大小的测量（9.2.1 节）。因为扇入和扇出的定义包含了全局数据，这个度量具有与数据相关的组成部分。然而，试验显示，这个度量比起像秩复杂度这样简单的度量来，并不是对复杂度的一个更好的度量［Kitchenham，Pickard，and Linkman，1990；Shepperd，1990］。

当使用面向对象范型时，设计度量的问题甚至更加复杂。例如，一个类的秩复杂度通常较低，因为许多类典型地包含了大量的小而简单的方法。更进一步地，如前面指出的，秩复杂度忽略了数据复杂度，因为数据和操作在面向对象范型中具有同等重要的地位，秩复杂度忽略了能够对对象的复杂度做出贡献的一个重要组成部分，因此，结合了秩复杂度的类的度量通常没有什么用。

人们已经提出了一些面向对象设计的度量，例如，在［Chidamber and Kemerer，1994］中。［Binkley and Schach，1996；1997；1998］在理论和实践范围内都对这些和其他度量提出了一些质疑。

## 13.16　设计阶段面临的挑战

如在 11.16 节和 12.9 节中所指出的那样，重要的是在规格说明阶段不要做得太多，即规格说明小组必须不要过早地开始设计阶段的部分。在设计阶段，设计小组可能在两个方向误入歧途：做得过多和做得过少。

考虑图 13-7 的 PDL（伪码）详细设计。对于喜爱编程的设计者来说，用 C++ 或 Java 而不是 PDL 编写详细设计的诱惑是很大的，即不是用伪码拟制详细设计，设计者几乎是在编写模块的代码。这比仅仅对模块进行概要设计花费的时间要多，而且如果在设计中检查出错误（见图 1-5），花费的时间会更多。像规格说明小组一样，设计小组的成员必须牢牢克制自己，不要比要求他们的做得更多。

与此同时，设计小组必须注意不要做得太少。考虑图 13-6 的表格形式的详细设计，如果设计小组匆忙行事，它可能决定将详细设计缩减为仅仅是叙述性的方框图，它甚至可能决

定程序员应当自己完成详细设计。这些决定都是错误的，进行详细设计的主要原因是为了确保全部的接口正确，叙述性方框图对于这个目的是不合适的，完全没有详细设计显然更无助于达到目的。因此，设计阶段面临的一个挑战是，设计者做的工作量要恰到好处。

此外，还有其他更多更严重的挑战。Brooks［1986］在他的“No Silver Bullet”文章中（参见3.15节中的“如果你想知道”部分），谴责他所说的“伟大的设计者”的缺乏，即指的是，那些比设计小组的其他成员杰出得多的设计者。按照Brooks的意见，一个软件项目的成功很大程度上依赖于设计小组是否由一个伟大的设计者领导。好的设计是可以学习的，伟大的设计只可能由伟大的设计者产生，而他们是“非常稀罕的”。

那么挑战就是，培养伟大的设计者。应当尽可能早地识别出他们（最好的设计者不一定是最有经验的），给他们安排导师，向他们提供正式的教育，以及成为伟大的设计者所需的学徒期，并且允许他们与其他设计者交流。这些设计者应当能够进入专门的职业生涯，他们得到的薪水应当与伟大的设计者对软件项目做出的贡献相称。

## 本章回顾

设计流在13.1节中介绍。有三个基本的设计方法：面向操作设计（13.2节）、面向数据设计（13.5节）和面向对象设计（13.6节）。讲述了两个面向操作设计的实例，它们是数据流分析（13.3节）和事务分析（13.4节）。在13.7节中将面向对象设计应用于电梯问题实例研究。在13.8节中提出了详细设计技术。13.9节提出了设计流，测试流的设计方面在13.10节中描述，13.11节描述了它应用于Osbert Oglesby实例研究的情况。13.12节讨论了详细设计的形式化技术，实时系统设计在13.13节中介绍。在13.14节和13.15节中，分别给出了设计阶段的CASE工具和度量。本章结束时讨论了设计阶段面临的挑战（13.16节）。

## 进一步阅读

讨论了数据流分析和事务分析的书籍有［Gane and Sarsen，1979］和［Yourdon and Constantine，1979］。Jackson技术的描述见Jackson［1975］。对Warnier的工作感兴趣的读者，可参考的最原始的信息源是［Warnier，1976］，Orr的方法可以在［Orr，1981］中找到。

现在转到面向对象设计，有关信息可以从下面获得：［Wirfs-Brock，Wilkerson，and Wiener，1990；Coad and Yourdon，1991b；Shlaer and Mellor，1992；and Jacobson，Booch，and Rumbaugh，1999］。各种面向对象设计技术的比较可见［Monarchi and Puhr，1992］和［Walker，1992］。［Briand，Bunse，and Daly，2001］讨论面向对象设计的可维护性。面向对象和传统设计技术的比较出现在［Fichman and Kemerer，1992］中。［Jackson and Chapin，2000］描述了空中交通控制系统的再设计，［Stolper，1999］中给出了高性能、可靠系统的设计技术。

形式化设计技术的描述见［Hoare，1987］。

关于在设计过程期间的评审，最初有关设计审查的文章是［Fagan，1976］，详细信息可以从那篇文章中得到。后来有关评审技术的进展在［Fagan，1986］中描述，使用走查来测试用户接口设计的描述见［Bias，1991］。

关于实时设计，具体技术可以在［Liu，2000］和［Gomaa，2000］中找到。可以在

[Kelly and Sherif，1992] 中可找到四种实时设计技术的比较，分布式系统的设计在 [Kramer，1994] 中有介绍。《IEEE Software》的 1999 年 9/10 月刊中包含了一些有关结构化设计的文章，特别是 [Herbsleb and Grinter，1999]。

在 [Henry and Kafura，1981] 和 [Zage and Zage，1993] 中描述了设计阶段的度量，面向对象设计的度量的讨论见 [Chidamber and Kemerer，1994] 和 [Binkley and Schach，1996]。面向对象性质的模型在 [Bansiya and Davis，2002] 中给出。

软件规格说明和设计国际研讨会的会议录是有关设计技术的详尽的信息来源。

## 习题

13.1　从习题 11.6 的 DFD 开始，用数据流分析设计一个产品，确定一个银行陈述报告书是否正确。

13.2　使用事务分析设计一个控制 ATM（习题 8.9）的软件，在这个阶段忽略错误处理能力。

13.3　现在利用你对习题 13.2 的设计，向其中加入执行错误处理的模块。认真检查得到的设计，并确定模块的内聚和耦合。注意图 13-10 中所描述的情形。

13.4　在 13.3.1 节中（图 13-6 和图 13-7）给出了描述详细设计的两种不同技术，对照比较这两种技术。

13.5　从你的自动化图书馆循环系统的数据流图（习题 11.8）开始，使用数据流分析设计该循环系统。

13.6　使用事务分析重复习题 13.5。你发现这两种技术中的哪一种更合适？

13.7　（分析和设计项目）从确定银行声明是否正确的产品（习题 12.24）的面向对象分析开始，使用面向对象设计来设计该软件产品。

13.8　（分析和设计项目）从自动化图书馆循环系统的面向对象分析（习题 12.25）开始，使用面向对象设计来设计该图书馆系统。

13.9　（分析和设计项目）从 ATM 软件（习题 12.26）的面向对象分析开始，使用面向对象设计来设计该 ATM 软件。

13.10　（学期项目）从习题 11.15 或习题 12.27 的规格说明开始，设计 Ophelia 的 Oasis 产品（附录 A），使用由你的老师规定的设计技术。

13.11　（实例研究）使用数据流分析重新设计 Osbert Oglesby 产品。

13.12　（实例研究）使用事务分析重新设计 Osbert Oglesby 产品。

13.13　（实例研究）图 13-15 和图 13-16 的详细设计以 PDL 的形式给出，请使用表格的形式再次给出该设计。哪种表示更好？给出你的理由。

13.14　（软件工程读物）你的老师将发给你们 [Bansiya and Davis，2002] 的复印件，对于评定面向对象设计质量的模型的可用性，你有何看法？

## 参考文献

[Bansiya and Davis, 2002] J. BANSIYA AND C. G. DAVIS, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering* **28** (January 2002), pp. 4–17.

[Beizer, 1990] B. BEIZER, *Software Testing Techniques*, 2nd ed., Van Nostrand Reinhold, New York,

1990.

[Bias, 1991] R. BIAS, "Walkthroughs: Efficient Collaborative Testing," *IEEE Software* **8** (September 1991), pp. 94–95.

[Binkley and Schach, 1996] A. B. BINKLEY AND S. R. SCHACH, "A Comparison of Sixteen Quality Metrics for Object-Oriented Design," *Information Processing Letters* **57** (No. 6, June 1996), pp. 271–75.

[Binkley and Schach, 1997] A. B. BINKLEY AND S. R. SCHACH, "Toward a Unified Approach to Object-Oriented Coupling," *Proceedings of the 35th Annual ACM Southeast Conference,* Murfreesboro, TN, April 2–4, 1997, pp. 91–97.

[Binkley and Schach, 1998] A. B. BINKLEY AND S. R. SCHACH, "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures," *Proceedings of the 20th International Conference on Software Engineering,* Kyoto, Japan, April 1988, pp. 542–55.

[Briand, Bunse, and Daly, 2001] L. C. BRIAND, C. BUNSE, AND J. W. DALY, "A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs," *IEEE Transactions on Software Engineering* **27** (June 2001), pp. 513–30.

[Brooks, 1986] F. P. BROOKS, JR., "No Silver Bullet," in: *Information Processing '86,* H.-J. Kugler (Editor), Elsevier North-Holland, New York, 1986; reprinted in: *IEEE Computer* **20** (April 1987), pp. 10–19.

[Chidamber and Kemerer, 1994] S. R. CHIDAMBER AND C. F. KEMERER, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* **20** (June 1994), pp. 476–93.

[Coad and Yourdon, 1991b] P. COAD AND E. YOURDON, *Object-Oriented Design,* Yourdon Press, Englewood Cliffs, NJ, 1991.

[Fagan, 1976] M. E. FAGAN, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* **15** (No. 3, 1976), pp. 182–211.

[Fagan, 1986] M. E. FAGAN, "Advances in Software Inspections," *IEEE Transactions on Software Engineering* **SE-12** (July 1986), pp. 744–51.

[Fichman and Kemerer, 1992] R. G. FICHMAN AND C. F. KEMERER, "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique," *IEEE Computer* **25** (October 1992), pp. 22–39.

[Flanagan, 2002] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference,* 4th ed., O'Reilly and Associates, Sebastopol, CA, 2002.

[Gane and Sarsen, 1979] C. GANE AND T. SARSEN, *Structured Systems Analysis: Tools and Techniques,* Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Goldberg and Robson, 1989] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language,* Addison Wesley, Reading, MA, 1989.

[Gomaa, 2000] H. GOMAA, *Designing Concurrent, Distributed, and Real-Time Applications with UML,* Addison Wesley, Reading, MA, 2000.

[Henry and Kafura, 1981] S. M. HENRY AND D. KAFURA, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering* **SE-7** (September 1981), pp. 510–18.

[Herbsleb and Grinter, 1999] J. D. HERBSLEB AND R. E. GRINTER, "Architectures, Coordination, and Distance: Conway's Law and Beyond," *IEEE Software* **16** (September/October 1999), pp. 63–70.

[Hoare, 1987] C. A. R. HOARE, "An Overview of Some Formal Methods for Program Design," *IEEE Computer* **20** (September 1987), pp. 85–91.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries,* ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, Switzerland, 1995.

[Jackson, 1975] M. A. JACKSON, *Principles of Program Design,* Academic Press, New York, 1975.

[Jackson and Chapin, 2000] D. JACKSON AND J. CHAPIN, "Redesigning Air Traffic Control: An Exercise in Software Design," *IEEE Software* **17** (May/June 2000), pp. 63–70.

[Jacobson, Booch, and Rumbaugh, 1999] J. RUMBAUGH, G. BOOCH, AND I. JACOBSON, *The Unified Software Development Process,* Addison Wesley, Reading, MA, 1999.

[Kelly and Sherif, 1992] J. C. KELLY AND J. S. SHERIF, "A Comparison of Four Design Methods for Real-Time Software Development," *Information and Software Technology* **34** (February 1992), pp. 74–82.

[Kitchenham, Pickard, and Linkman, 1990] B. A. KITCHENHAM, L. M. PICKARD, AND S. J. LINKMAN, "An Evaluation of Some Design Metrics," *Sofware Engineering Journal* **5** (January 1990), pp. 50–58.

[Kramer, 1994] J. KRAMER, "Distributed Software Engineering," *Proceedings of the 16th International Conference on Software Engineering,* Sorrento, Italy, May 1994, pp. 253–63.

[Liu, 2000] J. W. S. LIU, *Real Time Systems,* Prentice-Hall, Upper Saddle River, NJ, 2000.

[McCabe, 1976] T. J. MCCABE, "A Complexity Measure," *IEEE Transactions on Software Engineering* **SE-2** (December 1976), pp. 308–20.

[Monarchi and Puhr, 1992] D. E. MONARCHI AND G. I. PUHR, "A Research Typology for Object-Oriented Analysis and Design," *Communications of the ACM* **35** (September 1992), pp. 35–47.

[Orr, 1981] K. ORR, *Structured Requirements Definition,* Ken Orr and Associates, Topeka, KS, 1981.

[Shepperd, 1990] M. SHEPPERD, "Design Metrics: An Empirical Analysis," *Software Engineering Journal* **5** (January 1990), pp. 3–10.

[Shlaer and Mellor, 1992] S. SHLAER AND S. MELLOR, *Object Lifecycles: Modeling the World in States,* Yourdon Press, Englewood Cliffs, NJ, 1992.

[Silberschatz, Galvin, and Gagne, 2002] A. SILBERSCHATZ, P. B. GALVIN, AND G. GAGNE, *Operating System Concepts,* 6th ed., Addison Wesley, Reading, MA, 2002.

[Stolper, 1999] S. A. STOLPER, "Streamlined Design Approach Lands Mars Pathfinder," *IEEE Software* **16** (September/October 1999), pp. 52–62.

[Stroustrup, 2000] B. STROUSTRUP, *The C++ Programming Language,* special 3rd ed., AddisonWesley, Reading, MA, 2000.

[Walker, 1992] I. J. WALKER, "Requirements of an Object-Oriented Design Method," *Software Engineering Journal* **7** (March 1992), pp. 102–13.

[Ward and Mellor, 1985] P. T. WARD AND S. MELLOR, *Structured Development for Real-Time Systems,* Vols. 1, 2 and 3, Yourdon Press, New York, 1985.

[Warnier, 1976] J. D. WARNIER, *Logical Construction of Programs,* Van Nostrand Reinhold, New York, 1976.

[Wirfs-Brock, Wilkerson, and Wiener, 1990] R. WIRFS-BROCK, B. WILKERSON, AND L. WIENER, *Designing Object-Oriented Software,* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Yourdon and Constantine, 1979] E. YOURDON AND L. L. CONSTANTINE, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Prentice-Hall, Englewood Cliffs, NJ, 1979.

[Zage and Zage, 1993] W. M. ZAGE AND D. M. ZAGE, "Evaluating Design Metrics on Large-Scale Software," *IEEE Software* **10** (July 1993), pp. 75–81.

# 第14章 实　现

**学习目标**

学完本章之后，你应当能够：

- 完成实现流；
- 完成黑盒测试、白盒测试和基于非执行代码的单元测试；
- 完成综合测试、产品测试和验收测试；
- 意识到需要有好的编程实践和编程标准。

实现是将详细设计翻译成为代码的过程，当这由一个人完成时，这个过程相当好理解，但是今天大多数实际产品太大了，不可能由一个程序员在给定的时间限制内实现。这个产品需要由一个小组来实现，小组内的成员在同一时间内对该产品的不同组件做着工作，这称为**多人编程**（programming-in-the-many）。本章讨论与多人编程有关的问题。

## 14.1 编程语言的选择

在大多数情况下，完全不存在选择哪种编程语言实现的问题。假定客户想用比如说 Smalltalk 来编写产品，也许按照开发小组的观点，Smalltalk 完全不适合该产品。这样一种观点与客户完全无关，开发组织的管理者只有两个选择：用 Smalltalk 实现该产品或者拒绝这个工作。

同样，如果该产品必须在一台特定的计算机上实现，而该计算机上可用的语言仅是汇编语言，那么同样不存在任何选择。如果没有其他语言可用，这或者是因为还没有为该台计算机上的高层语言编写编译器，或者管理者不准备为这台计算机新的 C++ 编译器付费，那么同样，编程语言的选择问题没有任何意义。

一个更有趣的问题是：一个合同规定产品要用“最合适的”编程语言实现，应当选择什么样的语言？要回答这个问题，来看看下面的情景。QQQ 公司一直在编写 COBOL 软件产品，已经有 30 多年了。QQQ 公司的全部 200 名软件职员，从最低层的职员到负责软件的副总裁，都有 COBOL 的编程经历，为什么最适合的编程语言不应当是 COBOL 而是别的呢？引入一种新语言，比如说 Java，将意味着必须雇用新的程序员，或者最起码，现有的职员需要再次接受强化培训。既然在 Java 培训上花费了很多的金钱和努力，管理者可能会决定将来的产品应当用 Java 来编写，然而，所有现有的 COBOL 产品将不得不对其进行维护，那么就将存在两类程序员，COBOL 维护程序员和编写新的应用程序的 Java 程序员。很不应当的是，维护工作总是被认为比开发新的应用程序低下，因此在那些 COBOL 程序员中间将存在明显的不快情绪，这种不快还会因 Java 程序员通常比 COBOL 程序员得到较高薪水的事实而加重，因为 Java 程序员现在短缺。尽管 QQQ 公司有极好的 COBOL 开发工具，将不得不购买一个 Java 编译器，同样还要购买适当的 Java CASE 工具。还要购买或租用额外的硬件，以使新的软件得以运行。所有这一切中也许最严重的是，QQQ 已经积累了几百年的 COBOL 专长，这种专长只能通过亲身实践才能获得，比如说，当某个含义模糊的错误消息出现在屏幕上时做什么，或者如何处理编译器的某些特性。简而言之，看起来似乎“最适合的”编程语言只能是 COBOL——任何其他的选择将是经济上的自杀，无论是从投入的成本来看，还

是从加重职员的精神负担从而导致较差的代码质量的结果来看，都是如此。

可是，QQQ公司最近接手的项目最适合的编程语言可能确实是除COBOL之外的某种语言，尽管COBOL具有世界上最广泛使用的编程语言的地位（参见如下的“如果你想知道”部分）。COBOL仅适合数据处理应用一类的软件产品，但是如果QQQ公司有了这种类型之外的软件需求，那么COBOL立即就失去了它的吸引力。例如，如果QQQ公司想要使用人工智能（artificial intelligence，AI）技术建造一个基于知识的产品，那么可能使用像Lisp这样的AI语言。对于AI应用，COBOL是完全不合适的。如果要建造一个大型通信软件，可能因为QQQ需要卫星链路连接世界各地的几百个分支机构，那么像Java这样的语言可能远比COBOL合适。如果QQQ公司将要涉足编写像操作系统、编译器和链接器这样的系统软件的商业领域，那么COBOL绝对是不适合的。而且如果QQQ公司决定进入国防项目合同，那么公司的管理者不久就会发现，COBOL根本不能用于实时嵌入式软件。

**如果你想知道**

用COBOL编写的代码比用所有其他编程语言编写的代码总和多得多，COBOL是最广泛使用的语言，这主要是因为COBOL是美国国防部（Department of Defense，DoD）的一个产品，在已故的美国海军少将Grace Murray Hopper的指导下开发的COBOL，在1960年得到DoD的认可。从那以后，除非硬件有COBOL编译器，否则DoD不会购买该硬件来运行数据处理应用［Sammet，1978］。DoD曾是而且现在仍是世界上最大的计算机采购者，而且在20世纪60年代，编写的很大一部分DoD软件是用于数据处理的。结果是，作为对实际上每台计算机的强制性要求，编写了COBOL编译器。COBOL的广泛的可用性，加上在当时可选的语言通常只有汇编语言，这些使得COBOL成为世界上最流行的编程语言。

对于新的应用，像C、C++、Java和4GL这样的语言毫无疑问地正在流行，然而，交付后的维护仍是主要的软件活动，而且这个维护是在现存的COBOL软件上进行。简言之，DoD通过它的第一个主要的编程语言COBOL，在全世界的软件上打下了它的印记。

COBOL流行的另一个原因是它常常是实现数据处理产品的最好的语言，特别是，当涉及钱时，COBOL通常是所选择的语言。财务账簿需要平衡，这样就不允许舍入误差混进来，因此，全部的计算都必须使用整数算法完成，COBOL支持对每个较大的数的整数算法（即几十亿美元）。此外，COBOL可以处理非常小的数，比如，分币的小数。银行业的规则要求利息最少计算到分币的小数点后4位，而COBOL可以轻松地完成这个算法。最后，COBOL在任何第三代语言（或高级语言，见14.2节）中，可能具有最好的格式化、排序以及报表生成辅助工具。所有这些原因曾使COBOL对于实现数据处理产品来说，成为一个极好的选择。

如8.7.4节提到的那样，新的COBOL语言标准是用于面向对象语言的。这个标准确实将进一步促进COBOL的流行。

使用哪种编程语言的问题通常可通过使用成本－效益分析法（5.2节）来决定，也就是说，管理者必须计算一个COBOL实现的美元成本，以及使用COBOL的现在和将来的美元收益。这个计算必须对每个在考虑之列的语言重复进行，具有最大期望收益的语言，即估计效益和估计成本之差最大的语言，是合适的实现语言。另一种办法是使用风险分析，对于所考虑的每种语言，列出潜在的风险和解决办法的清单，整个风险最小的语言就是应选择的语言。

当前，软件公司受到压力，要用一种面向对象语言开发新的软件——任何的面向对象语

言。出现的问题是这样：哪种是适当的面向对象语言？20 年前，只存在一种选择，Smalltalk。可是今天，广泛应用的面向对象编程语言是 C++ [Borland，2002]，Java 占第二位。C++ 的流行有一些原因，原因之一是 C++ 编译器的广泛可用性，实际上，一些 C++ 编译器简单地将源代码从 C++ 翻译成为 C，然后调用 C 编译器，因此，任何带有 C 编译器的计算机本质上都可以处理 C++ 程序。

但是对 C++ 的普及的真正解释是它与 C 明显的相似性，这有些令人遗憾，表现在一些管理者将 C++ 简单地看作是 C 的一个扩展集，因此推断任何了解 C 的程序员能够迅速地掌握增加的部分。确实，若只是从句法的角度来看，C++ 本质上是 C 的一个扩展集，不管怎么说，实际上所有的 C 程序都能够用一个 C++ 编译器来编译。然而，从概念上讲，C++ 与 C 完全不同，C 是一个结构化范型的产品，而 C++ 是用于面向对象范型的。只有在已经使用了面向对象技术，并且产品是围绕着对象和类而不是模块来安排的时候，使用 C++ 才有意义。

因此，在一个公司采用 C++ 之前，对有关软件专业人员进行面向对象范型方面的培训很重要，特别重要的是要教给他们第 7 章的知识。除非所有涉及的人（特别是管理者）清楚了面向对象范型是一种不同的软件开发方法，并深知它与结构化范型的不同点是什么，否则只能继续将结构化范型用于用 C++ 而不是用 C 编写的代码。当一些公司对从 C 转换到 C++ 的结果感到失望时，一个主要因素是缺乏在面向对象范型方面的训练。

假定一个组织决定采用 Java，这时便不可能从结构化范型渐进地转移到面向对象范型。Java 是一种纯粹的面向对象编程语言，它不支持结构化范型的函数和过程。与像 C++ 这样的混合型的面向对象语言不同，Java 程序员必须从一开始就使用面向对象范型（且仅使用面向对象范型）。由于必须突然从一个范型转变到另一个范型，比起该公司转到一个像 C++ 或 OO-COBOL 这样的混合型的面向对象语言，采用 Java（或其他像 Smalltalk 这样纯粹的面向对象语言）时的培训显得更重要。

## 14.2 第四代语言

第一台计算机既没有解释器也没有编译器，它是用二进制码编程的，或是用电路板电路连接实现，或是通过设置开关实现，这样一个二进制机器码是**第一代语言**。**第二代语言**是在 20 世纪 40 年代末和 50 年代初开发的汇编语言，与不得不用二进制码编程不同，它的指令可以用下面这样的符号来表示：

```
mov $17, next
```

通常，每个汇编指令都被翻译成为一个机器码指令，因此，尽管汇编程序比机器码指令易于编写，而且便于维护程序员理解，汇编源代码与机器码一样长。

像 C、C++、Pascal 或 Java 等**第三代语言**（或高级语言）中蕴含的思想是，将一个高级语言的语句编译成为多达 5 或 10 个机器码指令（这是抽象的另一个例子，见 7.4.1 节）。高级语言代码由此比同等的汇编代码短很多，它也简单并易于理解，而且，因此比汇编代码容易维护。通常高级语言代码可能不如同等的汇编代码那样效率高，但由于易于交付后维护，因此还是更具优势。

这个概念在 20 世纪 70 年代后期又更进了一步，设计**第四代语言**（4GL）的一个主要目标是：每个 4GL 语句应当等效于 30 或 50 条机器代码指令。用像 Focus 或 Natural 这样的第四代语言开发的产品规模缩小了，并且因此开发较快和易于维护。

用机器码编程很困难，在某种程度上用汇编语言编程会容易一些，使用高级语言会更容易。4GL的第二个主要的目标是易于编程，特别地，许多4GL是**非过程的**（nonprocedural，参见如下的“如果你想知道”部分，可以得到对这个术语的进一步理解）。例如，考虑下面的命令：

```
for every surveyor
  if rating is excellent
    add 6500 to salary
```

直到4GL编译器才可将这个非过程的指令翻译成为一系列可以程序化执行的机器码指令。

**如果你想知道**

若干年前，我招计程车去纽约市的Grand中心车站，我对司机说：“请带我到林肯中心。”这就是一个非过程的请求，因为我只是表达了我想要的结果，至于如何达到想要的结果，留给司机去决定。然而司机是一名刚从中欧来的移民，他到美国还不到2个月，事实上对纽约的地理或英语了解甚少。因此，我赶快用过程式请求代替了先前的非过程式请求，“直行，直行，下一个红绿灯右转。对，这里右转，对，右转！现在直行。请慢下来，慢下来。看在上帝的份上，请慢一些！”等等，直到我终于到达林肯中心。

在转向4GL的公司中，存在大量的成功故事，几个先前使用COBOL的公司曾报告通过使用4GL，他们的生产率增长了10倍。许多公司发现通过使用4GL他们的生产率确实提高了，但是没有那么大。其他公司试过4GL，但是对结果十分失望。

出现这种不一致情况的一个原因是，一个4GL不会对所有的产品都合适，正相反，重要的是为具体的产品选择合适的4GL。例如，Playtex使用IBM的Application Development Facility（ADF），并报告生产率比使用COBOL增长了80倍，尽管有这个惊人的结果，Playtex后来继续对产品使用COBOL，因为管理者认为它们不太适合ADF［Martin，1985］。

产生这些不一致结果的第二个原因是，许多4GL由功能强大的CASE工作台和环境来支持（见5.5节）。CASE工作台和环境既有优点也有缺点，如3.13节解释的那样，在一个成熟程度低的公司里，不建议引入大规模的CASE，原因是CASE工作台或环境的目的是支持软件过程。一个级别1的软件公司还没有一个合适的软件过程，如果在这个水平上将CASE作为向4GL转换的一部分而使用，这将在一个还没有对任何过程做好准备的软件公司上施加一个过程。通常最好的结果也不能令人满意，并且结果可能是灾难性的。事实上，一些报道过的4GL失败可以归因于与此有关的CASE环境的结果，而不是4GL本身。

在［Guimaraes，1985］中报道了43个公司对4GL的态度，发现4GL的使用提升了用户的工作效率，因为当一个用户需要从公司的数据库中提取信息时，数据处理部门可能响应得更快。然而，也存在一些问题，某些4GL被证明是缓慢和低效的，响应时间很长。一个在IBM 4331主机上消耗60%CPU周期的产品，虽然支持，但最多只有12个并发的用户。总的来说，已经使用3年多4GL的28个公司感到，收益大于成本。

没有一个4GL在软件市场上占有绝对优势地位，相反，有几百个4GL，其中包括DB2、Oracle和PowerBuilder，并拥有一定规模的用户群。4GL的这种大面积扩散进一步证明了，在选择合适的4GL时要仔细。当然，很少有公司能够支付支持多于一个4GL的费用，一旦选择并使用了一个4GL，该公司必须在后续的产品中使用那个4GL，或者回到在引入4GL之前使用的那种语言上。

尽管存在潜在的生产率增长，不恰当地使用 4GL 却有潜在的危险。许多公司当前有大量要开发的产品的订货，还有一长串列表的交付后维护任务要完成。许多 4GL 的一个设计目标是**最终用户编程**（end-user programming），即由使用产品的人编程。例如，在 4GL 出现前，保险公司的投资管理者可能会向产品的数据处理管理者请求，显示有关其订单数量的某些信息。投资管理者然后会等一年左右，让数据处理小组有时间开发产品。人们希望得到一个简单易用的 4GL，它能使未受过编程训练的投资管理者能独立编写出想要的产品。最终用户编程的目的是帮助减少开发工作量，留下专业人员去维护现有的产品。

实践中，最终用户编程是很危险的。首先，考虑所有产品开发都由计算机专业人员完成的情形，计算机专业人员被训练得不相信计算机的输出，毕竟，在产品开发期间所有输出中正确的可能还不到 1%。另一方面，用户被告知可以信任全部的计算机输出，因为在无差错前不应将产品交付用户。现在考虑鼓励最终用户编程的情形，当一个无编程经验的用户用一个用户界面友好的、非过程的 4GL 编写代码时，自然的趋向是让用户相信输出，毕竟多年来一直让用户相信计算机的输出。结果是，许多商业决策是根据由根本不可能正确的最终用户代码生成的数据做出的。在某些情形下，某些 4GL 的易使用性曾经导致经济上的灾难。

在这种趋向中存在另一种潜在的危险，在某些公司中，允许用户编写更新公司数据库的 4GL 产品。由用户制造的一个编程错误最终可能造成整个数据库的混乱。教训是显然的：由无经验或未受过适当训练的用户编程可能是极其危险的，即使不是致命的，它也会对公司的经济造成极大损害。

对 4GL 的最终选择是由管理者做出的，在做这样的决定时，管理者应当看懂许多成功使用 4GL 的故事，与此同时，管理者应当认真分析由使用一个不适当 4GL 导致的失败，以及由过早引入一个 CASE 环境或者由开发过程的较差管理所导致的失败。例如，失败的一个共同原因是忽视了在 4GL 的各个方面全面训练开发小组，包括合适的关系型数据库理论［Date, 1999］。管理者应当研究在规格说明领域成功和失败两个方面的经验教训，并且从过去的错误中吸取教训。选择合适的 4GL 可能意味着重大的成功，也可能意味着灾难性的错误。

在决定了实现语言之后，下一个问题就是如何应用软件工程原理来构建高质量的代码。

## 14.3 良好的编程实践

许多关于好的编程风格的建议是与特定语言有关的。例如，关于在 Lisp 中使用 COBOL 88 级入口或圆括号的建议，对于一个正在用 Java 实现产品的编程者而言意义并不大。相反，我们现在给出一些与语言无关的**良好编程实践**的建议。

### 14.3.1 使用一致和有意义的变量名

如第 1 章中所述，平均 2/3 的软件支出用于交付后维护。这意味着开发一个模块的程序员只是许多将要在该模块上工作的人中的第一人。程序员给出一个仅对该程序员有意义的变量名是达不到预期目标的，在软件工程的范畴内，术语**有意义的变量名**意味着“从将来维护程序员的角度来看是有意义的”。这个观点在如下的“如果你想知道”部分中有补充说明。

**如果你想知道**

20 世纪 70 年代后期，在南非的约翰内斯堡，有一个软件公司由两个编程小组组成。小

组A由从莫桑比克来的移民组成，他们具有葡萄牙血统，它们的母语是葡萄牙语，他们的代码写得好，变量名是有意义的，但遗憾的是仅对说葡萄牙语的人有意义。小组B由以色列移民组成，他们的母语是希伯来语，他们的代码写得同样好，他们选择的变量名的名称同样是有意义的，但是仅对讲希伯来语的人有意义。

一天，小组A连同他们的小组负责人一同辞职了。小组B完全无法维护小组A曾经编写的优秀的代码，因为他们不会讲葡萄牙语，对讲葡萄牙语人有意义的变量名，对于以色列人是完全不可理解的，这些以色列人的语言能力仅限于希伯来语和英语。软件公司的拥有人无法雇用足够的会说葡萄牙语的程序员代替小组A，不久公司在受到大量不满客户的法律诉讼的情况下破产了。因为这些客户的代码现在基本上是不可维护的了。

这种情形很容易避免，公司的领导应当在一开始就坚持让全部的变量名用英语表示，它是每个南非计算机从业人员都理解的语言。变量名从而对每个维护程序员就是有意义的了。

除了使用有意义的变量名之外，选择**一致的变量名**同样很重要。例如，在一个模块内声明了下面四个变量：averageFreq，frequencyMaximum，minFr和frqncyTotl。一个试图理解这段代码的维护程序员必须知道freq，frequency，fr和frqncy是否都指同一个东西。如果是这样，那么应当使用同样的词，建议使用frequency，当然freq或frqncy也在接受之列，不建议使用fr。但是如果一个或多个变量名指的是不同的量，那么应当使用一个完全不同的名字，如用rate表示速率，用frequency表示频率。相反，不要使用两个不同的名字表示同一个概念，例如，average和mean不应当在同一个程序中使用。

一致性的第二个方面是变量名的组件的顺序。例如，如果一个变量被命名为frequencyMaximum，那么名字minimumFrequency可能会使人迷惑，它应当是frequencyMininum。为了使代码易于被将来的维护程序员所理解，前面列出的四个变量应当分别命名为frequencyAverage，frequencyMaximum，frequencyMinimum和frequencyTotal。当然，frequency组件也可以出现在全部四个变量名的末尾，得到变量名averageFrequency，maximumFrequency，minimumFrequency和totalFrequency。显然选择两组中的哪一个没有关系，重要的是名字全部来自一个组或另一个组。

目前已经提出一些不同的命名约定，以使代码易于理解。其想法是一个变量名中应当体现类型信息。例如，ptrChTmp可能表示一个指向字符（Ch）指针类型（ptr）的临时变量（Tmp）。其中最著名的方案是匈牙利命名约定［Klunder，1988］（如果你想知道为什么称它们为匈牙利，参见如下的“如果你想知道”部分）。这样的许多方案的一个缺点是，当参加者不能拼读出变量名时，代码审查（参见14.14节）的效果将降低。不得不逐个字母地读出变量名，这非常不便。

**如果你想知道**

对于术语**匈牙利命名约定**有两种解释。第一，这些约定是由Charles Simonyi提出的，他出生于匈牙利；第二，人们普遍承认，对于没有经验的人来讲，程序带有符合该约定的变量名，阅读起来将像阅读匈牙利文一样容易。无论如何，使用它们的公司（如Microsoft）声称，对于那些具有匈牙利命名约定经验的人来说，它们增强了代码的可读性。

### 14.3.2　自文档代码的问题

当问起为什么他们的代码中不包含注释时，程序员们常常自豪地说“我编写的是**自文档**

代码（self documenting code）”，意思是他们的变量名经过认真选择，他们的代码编制得很精巧，以至于没有必要编写注释。自文档代码确实存在，但是非常少。其实，通常的情景是程序员在编写模块时，认真考虑代码中的每个名词，可以想见，程序员会对每个模块使用同样的风格，而且即使是在很长一段时间之后，最初的程序员对编写的代码也会非常清楚。很遗憾，这并不起作用，关键是模块是否能够容易和清楚地被所有其他必须读此代码的程序员所理解——从软件质量保证小组成员开始，包括一些不同的交付后维护程序员。当把交付后维护的任务安排给缺乏经验的程序员并且没有认真指导他们时，这些自以为是的做法会使问题变得更加尖锐。未加注释的模块代码对于一个有经验的程序员可能仅仅部分可懂，那么当维护程序员经验不丰富时，情形会更糟糕。

要了解为何会产生这类问题，考虑变量 xCoordinateOfPositionOfRobotArm，无论从哪种意义上说这样一个变量名无疑是自文档的，但是很少有程序员愿意使用一个 31 个字符的变量名，特别是如果这个名字使用频繁的话。相反，应使用一个短名字，如 xCoord。理由是，如果整个模块处理一个机器人手臂的动作，xCoord 仅能够表示机器人手臂的 x 坐标动作。尽管该看法在开发过程中说得通，它对于交付后维护不一定正确。维护程序员对该产品可能没有整体上足够的认识，认识到在这个模块内，xCoord 指的是机器人手臂的动作，或者可能没有必需的注释来理解模块的工作。避免这类问题的方法是坚持在每个模块的开始，即在**序言注释**（prologue comments）中对变量名做解释。如果遵循这个规则，维护程序员就会很快理解变量 xCoord 是用于机器人手臂位置的 x 坐标。

序言注释在每个模块中是必不可少的。在每个模块的顶部必须提供的最基本信息在图 14-1 中列出。

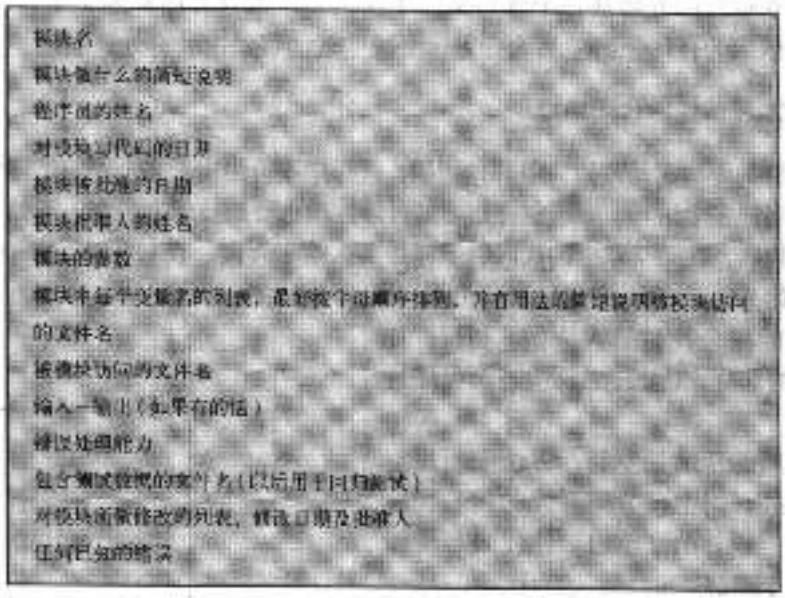



图 14-1 最基本的模块序言注释

即使清楚地编写了模块，也没有理由期望某人阅读每一行代码，理解该模块做什么和是如何做的。序言注释使其他人易于理解关键点，只有 SQA 小组的成员或修改某一模块的程

序员才应当阅读该模块的每一行。

除了序言注释之外，应当在代码中插入行内注释，以帮助维护程序员理解那行代码。人们已经建议，行内注释应当仅用在代码是用一种不显而易见的方式编写，或者使用了该语言中某些难理解的方面的时候。与此同时，令人迷惑的代码应当以清楚的方式重新编写。行内注释是帮助维护程序员的一种手段，不应当用来助长较差的编程实践或为其寻找借口。

### 14.3.3　使用参数

很少有真正的常量，即那些值“永远”不变的变量。例如，卫星图片引起对潜艇导航系统的改变，要在其中加进夏威夷的珍珠港的纬度和经度，以反映出关于珍珠港的精确位置的更准确的地理数据。另外举一个例子，销售税不是一个真正的常量，立法者有时倾向于改变销售税率，假定当前销售税率是6.0%，如果已经在一个产品的一些模块中对值6.0进行了确定的编码，那么改变该产品将是一个较大的动作，而这可能导致“常量”6.0的一个或两个实例被忽视，而且也许错误地改变了一个不相关的6.0。一个较好的解决办法是像下面这样的C++声明：

```
const float salesTaxRate = 6.0;
```

或者在Java中，为：

```
public static final float salesTaxRate = (float) 6.0;
```

那么，无论在哪里需要销售税率的值，应当使用常量salesTaxRate而不是数6.0。如果改变了销售税率，那么只需使用一个编辑器改变包含值salesTaxRate的行。更好的是，销售税率的值应当在运行的开始从一个参数文件中读入，全部这样明显的常量应当作为参数处理。如果一个值因为任何原因应当改变，这个改变就能够快速和有效地实现。

### 14.3.4　为增加可读性的代码编排

使一个模块易于阅读是相当简单的，例如，一行中不应当出现多个语句，即使许多编程语言允许一行中出现多个语句。缩进也许是增加可读性最重要的技术，想像一下如果没有使用缩进来帮助理解代码，在第7章中的代码例子将是多么难于阅读。在C++或Java中，缩进可以用来连接相应的{...}对。它也显示哪个语句属于某个给定的代码块，事实上，正确的缩进太重要了，不能完全托付给人来做。相反，如5.6节所描述的那样，CASE工具应当确保缩进正确完成。

另一个有用的帮助是空行。方法应当用空行隔离开，此外，用空行分隔开大的代码块常常是有帮助的。额外的“空白”使代码容易阅读，因而也易于理解。

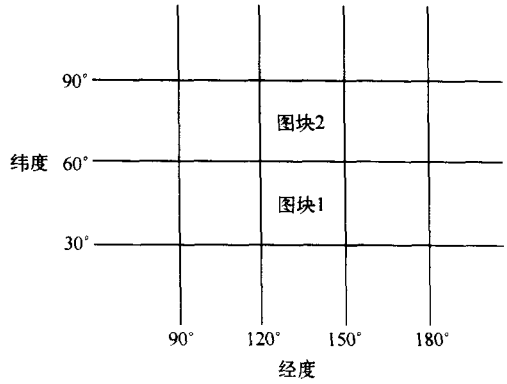



图14-2　图的坐标

### 14.3.5　嵌套的if语句

考虑下面的例子，一个图由两个方块组成，如图14-2所示。要求编写代码确定地球表面的

一个点是否位于图块 1 或图块 2 中，或者根本不在图中。图 14-3 的解决方法格式太差，它很难理解。适当进行格式编排的版本示于图14-4 中，尽管这样，**if-if** 和 **if-else-if** 构造的混合很复杂，难于检查代码段是否正确。在图14-5 中对此进行了整理。

```
if (latitude > 30 && longitude > 120){if (latitude <= 60 && longitude <= 150)
mapSquareNo =1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";}else print "Not on the map";
```

图 14-3　较差格式的嵌套 **if** 语句

```
if(latitude > 30 && longitude > 120)
{
  if(latitude <= 60 && longitude <= 150)
    mapSquareNo = 1;
  else
    if(latitude <= 90 && longitude<=150)
      mapSquareNo = 2;
    else
      print"Not on the map";
}
else
  print "Not on the map";
```

图 14-4　格式好但构造差的嵌套 **if** 语句

```
if (longitude > 120 && longitude <= 150 && latitude > 30 && latitude <= 60)
  mapSquareNo = 1;
else
  if (longitude > 120 && longitude <= 150 && latitude > 60 && latitude <= 90)
    mapSquareNo = 2;
  else
    print "Not on the map";
```

图 14-5　可接受的嵌套 **if** 语句

在面对包含 **if-if** 构造的复杂的代码时，一个简化的方法是利用下面这个事实，即 **if-if** 组合

**if** <条件 1>
　**if**<条件 2>

与单个条件

**if**<条件 1> **and** <条件 2>

是等效的，假如即使不持有<条件 1>，也定义<条件 2>。例如，<条件 1>可能检测到一个指针不是空的，并且如果是这样，那么<条件 2>可以使用那个指针。（此问题在 Java 或 C++ 中不出现，&& 操作符定义为，如果<条件 1>为 false，则不计算<条件 2>。）

**if-if** 构造的另一个问题是嵌套 **if** 语句过深会导致代码难以阅读，单从经验上看，嵌套 **if** 语句超过 3 级是一个较差的编程习惯，应当避免。

## 14.4　编码标准

**编码标准**既是好事也是坏事。7.2.1 节指出过，带有偶然性内聚的模块通常是作为一些

规则的结果出现的，这些规则如“每个模块将由 35～50 个可执行语句组成”。与提出这样一个教条的方法不同，更好的明确表达是，“程序员在构造一个少于 35 个或多于 50 个可执行语句的模块之前，应当征求管理者的意见。”关键要注意的是，没有编码标准可以适用于所有可能的情形。

上面施加的强制性编码标准常常被忽视，像前面提出的那样，一个有用的凭经验的方法是，**if** 语句嵌套深度不应当超过 3 级。如果向程序员展示因 **if** 嵌套深度过多导致的不可读代码的例子，那么很可能他们将遵守这样的规则。但是，他们很可能不会恪守一长串施加给他们、不经讨论和解释的规则。进一步地，这样的标准可能导致程序员和他们的管理者之间产生冲突。

此外，除非一个编码标准可以由机器检验，它或者将浪费 SQA 小组大量的时间，或者简单地为程序员和 SQA 小组所忽视。另一方面，考虑下面的规则：

- **if** 语句的嵌套不应当超过 3 级的深度，除非得到小组领导的特许。
- 模块应当由 35～50 个语句组成，除非得到小组领导的特许。
- 应当避免使用 **goto** 语句，然而，在得到小组领导的特许的情况下，向前的 **goto** 语句可以用于错误处理。

假如建立某种机制，捕捉涉及允许偏离标准的数据，这样的规则就可以用机器检测到。

编码标准的目标是使得维护更容易，然而，如果一个标准的效果是给软件开发者带来了不便，那么，该标准应当修改，即使是在项目的中期。过于严格的编码标准是达不到预期目标的，如果程序员不得不在这样一个框架下开发软件，软件产品的质量必然会受损失。另一方面，像前面列出的关于 **if** 语句嵌套、模块大小以及 **goto** 语句的那些标准，与一个偏离那些标准的检验机制结合起来，可以引起软件质量的提高，它毕竟是软件工程的一个主要目标。

## 14.5　代码重用

第 8 章中详细介绍过重用，事实上，有关重用的内容已经在本书中各处出现了，因为来自软件过程的各个阶段的组件都重用过，包括部分的规格说明、合同、计划、设计和模块。这就是为什么有关重用的内容被放在本书的第一部分，而不是试图放在一个或另一个特定的阶段。特别重要的是，有关重用的内容不放在这一章是为强调这样的事实：即使代码的重用是迄今为止最普遍的重用形式，重用也不仅仅是代码的重用。

## 14.6　集成

考虑图 14-6 中描述的产品，产品**集成**的一个方法是独立地对每个模块编写代码和测试，再连接所有 13 个模块，并作为一个整体测试产品。这种方法中有两个难题。第一：考虑 a 模块，它不能依靠自身进行测试，因为它调用模块 b、c 和 d。因此，要测试 a 模块，模块 b、c、d 必须被当成**存根**（stub）编码。在最简单的形式下，一个存根就是一个空的模块。一个更有效的存根是打印一条消息，例如模块 displayRadarPatterncalled。最好的情况下，一个存根应该能够返回与预先计划的测试用例相符合的值。

现在，考虑模块 h。测试模块 h 需要一个能调用它一次或多次的**驱动**（它是一个模块），如果可能，要检查接受测试的模块的返回值。同样地，测试模块 d 需要一个驱动与两个存根。因此，随着相互分离的实现与集成，第一个难题出现了，这就是：精力都放在构建存根与驱动上了，而所有这些存根与驱动在模块测试完成后都将被抛弃。

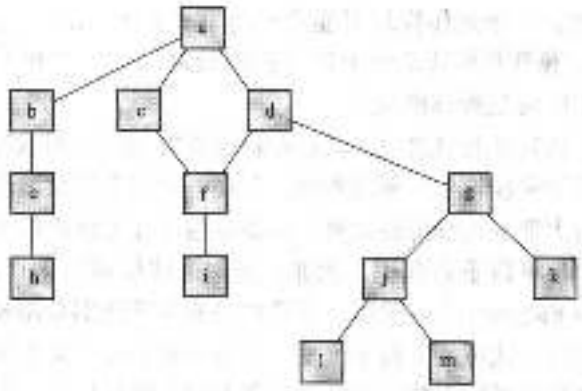



图 14-6 一种典型的互连图

随着实现阶段的完成，在集成工作开始之前，第二个更为重要的难题又出现了，这就是：缺乏错误隔离的手段。如果产品作为一个整体测试，而在特定的测试用例下产品失败了，那么错误就会存在于 13 个模块或者 13 个接口的任何一个或多个中。换另一个大型的软件来说，可能是 103 个模块和 108 个接口，那么可能隐藏错误的地点就会多达 211 个。

解决这两个问题的办法是将单元测试与集成测试结合起来。

### 14.6.1 自顶向下的集成

**自顶向下的集成**中，若模块 mAbove 给模块 mBelow 发消息，那么模块 mAbove 在模块 mBelow 之前实现与集成。假想一下，若图 14-6 中的产品是自顶向下实现与集成的。一种可能的自顶向下的次序是 a、b、c、d、e、f、g、h、i、j、k、l 和 m。首先，对模块 a 编码，并用作为存根实现的 b、c、d 测试 a。接下来存根 b 扩展为模块 b，与模块 a 连接，同时用作为一个存根实现的模块 e 对 b 进行测试。实现与集成按照这种方式进行下去，直至所有模块都集成到这个产品中去。另一种自顶向下的可能次序是 a、b、e、h、c、d、f、i、g、j、k、l 和 m。在这种次序下，实现与集成的一部分工作可以按如下方式并行进行。在模块 a 编码与测试结束之后，一个程序员可能利用模块 a 来实现与集成模块 b、c、h，同时另一个程序员可能利用模块 a 并行工作于模块 c、d、f 和 i。一旦模块 d 与 f 完成，第三个程序员可以开始进行模块 g、j、k、l 和 m 的工作。

设想一下：若模块 a 在一个特定的测试用例上正确执行，然而，当模块 b 在完成编码并集成到产品中后，此时的软件由模块 a 与模块 b 连接而成，用同样的测试数据再进行测试，测试失败了。这个错误可能存在于两个地点中的一个：模块 b 或者是模块 a 与模块 b 的接口。通常，无论什么时候一个新模块 mNew 加入一个已经过成功测试的产品后，运行测试用例失败了。错误几乎毫无疑问地存在于模块 mNew 或者模块 mNew 与产品的其他部分之间的接口中。因此自顶向下的实现与集成支持错误的隔离。

自顶向下的实现与集成的另一个优点是主要的错误能够较早地得以发现。一个软件的所有模块可以分为两组：逻辑模块与操作模块。**逻辑模块**本质上组成软件产品控制方面的决策流，逻辑模块通常是那些在模块相互关系图中离“根”较近的模块。例如，在图 14-6 中，认为模块 a、b、c、d 或许还有 g、j 是逻辑模块是合理的。另一方面，**操作模块**履行软件产

品中的实际操作。例如，一个操作模块可能命名为 getLineFromTerminal 或者 measureTemperatureOfReactorCore。操作模块通常位于互连图的较低层，离“叶”较近。在图 14-6 中，模块 e、f、h、i、k、l 和 m 是操作模块。

在对操作模块进行编码及测试之前，进行逻辑模块的编码与测试通常是非常重要的。这可以确保任何主要的设计失误较早地露出水面。若在一个主要错误被发现之前整个产品就完成了，那么整个软件的大部分代码都需重写，特别是包含控制流程的逻辑模块。许多操作模块在产品的重建过程中都可以重新利用，例如，类似上述提到的 getLineFromTerminal 模块或 measureTemperatureOfReactorCore 模块，不管产品如何重建都是需要的。然而，操作模块连接产品中的其他模块的方式可能不得不改变，这会导致不必要的工作。因此，设计上的错误发现得越早，修改产品与返回到软件的开发计划上的花费就越小，而且速度也会越快。采用模块自顶向下的实现与集成策略，需确保逻辑模块在操作模块之前实现与集成。因为在模块的互连关系图中，逻辑模块是操作模块的祖先。这是自顶向下集成的一个主要特点。

不过，自顶向下的集成方法也有一个弱点：潜在的可重用模块可能会测试不充分。重用被误认为经过充分测试的模块通常比重写代码方便得多，但是当一个产品失败时，则可能导致错误的结论。测试者可能考虑错误隐藏在另外的地方，而不是怀疑重用模块的不充分测试，由此造成工作的浪费。

逻辑模块大多适用于某些特定的问题，因此在另外一种环境下是不可重用的。然而，操作模块，特别是如果它们有信息性内聚（7.2.7 节），在将来的产品中有可能是能重用的，因此需要彻底地测试。遗憾的是，操作模块在模块关系表中通常在较低层，因此，不能像上层模块那样得到充分地测试。例如，若有 184 个模块，“根”模块可能会被测试 184 次，然而，集成在产品中的最后一个模块可能只被测试 1 次。由于操作模块的不充分测试，自顶向下的集成方法造成了重用是一件危险的事情。

如果产品设计得很好，那么情况可能会变得更坏；事实上，一个产品设计得越好，模块的测试可能就会越不彻底。为明白这一点，考虑模块 computeSquareRoot。这个模块带 2 个参数，第一个是要计算出其平方根的浮点型数“x”；第二个是“errorFlag”，如果 x 是负数，将置为“true”。深入考虑一下，模块 computeSquareRoot 被模块 a3 调用，并且模块 a3 包含以下语句：

```
if (X> = 0)
    y = computeSquareRoot(x, errorFlag);
```

换句话说，除非 x 的值是非负数，否则 computeSquareRoot 模块不会被调用，因此，此模块从来不会测试 x 为负值时模块运行是否正确。这种在调用模块之前进行安全检查的设计方式一般称为**保守编程**（defensive programming）。作为保守编程，如果自顶向下地集成，次要的操作模块不可能会被彻底地测试。与保守编程相对的另一种可选择的方法是职责驱动设计（见 1.9 节），在这种方法中，必要的安全性检查连入被调用的模块，而不是调用者。另一种办法是在被调用模块中加入声明（见 6.5.3 节）。

### 14.6.2　自底向上的集成

在**自底向上集成**中，如果模块 mAbove 发送一个消息给模块 mBelow，那么模块 mBelow

在模块 mAlxive 之前实现与集成。在图 14-6 中，一个可能的自底向上的次序是 l、m、h、i、j、k、e、f、g、b、c、d 和 a。为了便于一个编程小组开发产品，一个更好的自底向上的次序是：h、e 和 b 给一个程序员；i、f 和 c 给另一个程序员；第三个程序员负责 l、m、j、k 和 g，然后实现 d，并将自己的工作与第二个程序员的工作集成。最后，当 b、c 和 d 已成功集成之后，可以实现与集成 a。

当使用自底向上的策略时，操作模块能被充分地测试。另外，测试是通过驱动的帮助来完成的，而不是通过屏蔽错误、保守编程的模块来完成。尽管自底向上的集成解决了自顶向下集成的主要问题，并且与自顶向下的集成一样具有错误隔离的优点，但遗憾的是它自身也有一个麻烦。特别是在实现阶段后期检测主要的设计错误，逻辑模块被最后集成，因此，如果有重大的设计错误，在实现阶段的后期将不得不重新整理，需要花费巨大的精力来重新设计与编写大部分的产品代码。

因此，自顶向下与自底向上的集成各具优势与不足，产品开发的解决办法就是结合这两种策略，利用它们的优点去弥补不足，这就带来了三明治（sandwich）集成的理念。

### 14.6.3 三明治集成

考虑图 14-7 所示的互连图。a、b、c、d、g 和 j 这 6 个模块为逻辑模块，因此应自顶向下地集成。e、f、h、i、k、l 和 m 这 7 个操作模块应自底向上地集成。也就是说，因为自顶向下、自底向上这两种方法，无论哪一种也不能全部适用于所有模块，所以将它们分开处理。6 个逻辑模块用自顶向下的方法集成，从而能够尽早地发现主要问题。7 个操作模块自底向上集成，通过保守编程调用模块来屏蔽错误，它们能够受到彻底地测试，因此可以在其他产品中放心地重用。当所有模块都被正确地集成时，两组模块之间的接口一个一个地测试，整个过程中均有错误隔离的手段，这就称为**三明治集成**（参见下面的“如果你想知道”部分）。

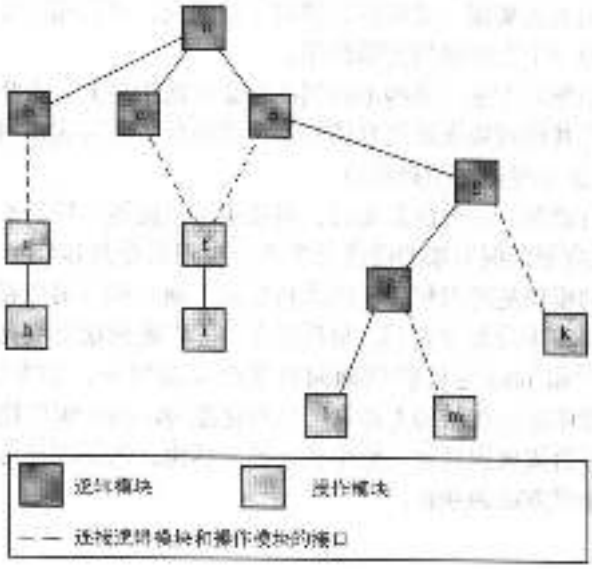



图 14-7 使用三明治集成开发的图 14-6 的产品

**如果你想知道**

**三明治集成**［Myers，1979］这个术语来源于可以认为逻辑模块与操作模块就似三明治的顶层与底层，而连接两种模块之间的接口就似三明治的填充物，如图 14-7 所示。

表 14-1 总结了三明治集成和本章中前面讨论过的其他集成技术的优缺点。

**表 14-1　本章给出的集成方法的总结及对应章节**

| 方　法 | 优　点 | 缺　点 |
| --- | --- | --- |
| 实现然后集成（14.6 节） | | 没有错误隔离手段，主要设计错误发现迟，潜在可重用代码模块不能被充分地测试 |
| 自顶向下的集成（14.6.1 节） | 具有错误隔离手段，主要设计错误发现早 | 潜在可重用代码模块不能被充分地测试 |
| 自底向上的集成（14.6.2 节） | 具有错误隔离手段，潜在可重用代码模块能被充分地测试 | 主要设计错误发现迟 |
| 三明治集成（14.6.3 节） | 具有错误隔离手段，主要设计错误发现早，潜在可重用代码模块能被充分地测试 | |

下面的“如何完成”部分中总结了三明治集成。

**如何完成三明治集成**

- **并行地**

  自顶向下地实现和集成逻辑模块。

  自底向上地实现和集成操作模块。

- 测试逻辑模块和操作模块之间的接口。

### 14.6.4　面向对象产品的集成

对象既可以自底向上地集成，又可以自顶向下地集成，若采用自顶向下的集成方法，与传统的模块一样，为每一个方法使用存根程序。

若采用自底向上的集成方法，那些不向别的对象发送消息的对象将首先实现与集成，然后，实现与集成那些向其他对象发送消息的对象，就这样，直到实现与集成完产品中的所有对象。（如果有递归，这个过程必须修正。）

由于自顶向下与自底向上的方法都支持，同样也可以使用三明治集成。如果一个产品由一种像 C++ 这样的混合型面向对象的语言来实现，类通常是操作模块，因此对其自底向上地集成。许多不是类的模块是逻辑模块，因此自顶向下地实现与集成它们。其他模块是操作性的，因此自底向上地实现与集成它们。最后将所有非对象模块集成到对象中。

即使当产品由一种如 Java 这样的纯面向对象语言实现时，类方法（有时也称**静态方法**），如 main 及实用程序方法在结构上通常与结构化范型中的逻辑模块类似。因此类方法也是自顶向下地实现，然后集成到其他对象中去。换句话说，当实现与集成一个面向对象的产品时，会用到三明治集成方法的变种。

### 14.6.5　集成的管理

在集成阶段会暴露管理的问题，即代码模块不能简单地连接在一起。例如，程序员 1 编

写对象 o1 的代码，程序员 2 编写对象 o2 的代码。在程序员 1 使用的设计文档中，对象 o1 向对象 o2 发送带有 4 个参数的消息，但程序员 2 在自己的设计文档中却清楚地写明对象 o2 只收到 3 个参数。如果没有通知开发小组的全体人员，就只对设计文档的一份拷贝进行修改时，会出现这样的问题。两个程序员都认为自己是正确的，谁也不愿意妥协，因为妥协的程序员将不得不重写产品的大部分代码。

为了解决这些以及类似的不相容的问题，整个集成阶段必须在 SQA 小组的管理下进行，而且，在其他阶段的测试中，如果集成测试工作进行得不正确，则 SQA 小组没有完成他们的主要工作。因此，SQA 小组更应确保测试工作进行得彻底。SQA 小组的负责人应为集成测试的各方面负责，他或她必须决定哪些模块应自顶向下地实现与集成，哪些模块应自底向上地实现与集成，并且决定分配合适的小组与个人进行实现与集成的测试任务。SQA 小组制定了软件项目管理计划中的集成测试计划，同时负责测试计划的实现。

在集成阶段的后期，应将所有代码模块测试过并合成为一个产品。

## 14.7 实现阶段

**实现阶段**的目标是用所选的实现语言实现目标软件产品。更准确地，如 13.9 节所讲，大型的软件产品被分成小一些的子系统，然后由编码小组并行地进行实现，同时，这些子系统由**组件**或**代码模块**组成。

只要代码模块完成编码，程序员就对其进行测试，这被称为**单元测试**。一旦程序员确认代码模块是正确的，就上交给质量保证小组做进一步的测试。质量保证小组所做的这种测试是测试阶段的一部分，本章后面将讨论。

## 14.8 实现阶段：Osbert Oglesby 实例研究

Osbert Oglesby 产品的 C++ 和 Java 版本的完整实现可从 www.mhhe.com/engcs/compsci/schach 下载。程序员在其中包含了各种注释来帮助交付后维护程序员。

接下来介绍实现流期间的测试。

## 14.9 测试阶段：实现

在实现流需要进行一些不同类型的测试，包括单元测试、集成测试、产品测试和验收测试。这些类型的测试将在后面的章节中讨论。

如 6.6 节所指出的，代码制品（模块、类）接受两种类型的测试：在开发代码制品时由程序员进行的非形式化单元测试，以及当程序员对制品功能正常显现感到满意后，由 SQA 小组进行的系统的单元测试。系统测试在 14.10 节到 14.14 节中描述，依次介绍的有两类基本的系统测试：**基于非执行的测试**，在其中制品由一个小组评审；**基于执行的测试**，在其中对照着测试用例运行制品。现在介绍选择这些测试用例的技术。

## 14.10 测试用例选择

测试一个制品最差的方法是使用随意的测试数据，测试者坐在键盘前，只要制品要求输入，测试者就用任意的数据来响应。如将要看到的那样，这样只能测试所有可能测试用例的最小一部分，再多时间就不允许了，因为它轻易就可以达到比 $10^{100}$ 还多。能够运行的很少

的测试用例（可能在1000这个量级上）非常宝贵，不能将它浪费在随意的数据上。更糟的是，机器经常请求输入，对同样的数据响应多次，这样浪费了更多的测试用例。显然**测试用例选择**必须系统地进行。

### 14.10.1　规格说明测试与代码测试

单元测试的测试数据可以用两个基本的方法系统地构建。第一个是**规格说明测试**，这个技术也称为**黑盒测试、行为测试、数据驱动测试、功能测试**以及**输入/输出驱动测试**。在这个方法中，不考虑代码本身，在拟制测试用例中使用的仅有的信息是规格说明文档。另一个极端是**代码测试**，它在选择测试用例时不理会规格说明文档。这个技术的其他名字有**玻璃盒测试、白盒测试、结构测试、逻辑驱动测试**以及**面向路径测试**（有关为什么有这么多不同的术语的解释，请见下面的“如果你想知道”部分）。

我们现在考虑这两个技术的可行性，从规格说明测试开始。

**如果你想知道**

*你当然好奇为什么给相同的测试概念这么多不同的名称。像在软件工程中常发生的一样，相同的概念由一些不同的研究者独立地发现了，他们每个人都发明了他或她自己的术语。当软件工程界认识到这些是同一概念的不同名字时，已经太晚了——各种名称已经渗透到软件工程的词汇表中来了。*

*在这本书中，我使用术语“黑盒测试”和“玻璃盒测试”。这些术语很具有描述性，当我们测试规格说明时，我们把代码当作一个完全不透明的黑盒。相反，当我们测试代码时，我们需要能够看到盒子内部，因此使用术语“玻璃盒测试”。我避免使用术语“白盒测试”，因为它多少有些令人误解，毕竟，一个涂成白色的盒子与涂成黑色的盒子一样是不透明的。*

### 14.10.2　规格说明测试的可行性

考虑下面的例子。假定某个数据处理产品的规格说明指出，必须包含5类佣金和7类折扣。仅测试佣金和折扣的每个可能的组合就需要35个测试用例，说佣金和折扣是在两个完全独立的模块中，因而可以独立地测试是没有用的——在黑盒测试中将产品当作黑盒对待，它的内部结构因此也是完全无关的。

这个例子仅包含两个要素，佣金和折扣，它们分别取5个和7个不同的值。任何现实的产品如果没有几千个也有几百个不同的要素，即使仅有20个要素，每个仅取4个不同的值，也必须检查总共$4^{20}$或$1.1\times10^{12}$个不同的测试用例。

要看超过1万亿个测试用例的实现，可以考虑一下全部测试它们要花多少时间。如果可以找到一个程序员小组，他们以平均每30秒一个的速率生成、运行和检查测试用例，那么将花费一百多万年彻底地测试该产品。

因此，彻底的规格说明测试在实际中是不可能的，因为它的组合方式会爆炸性地增长。有太多的测试用例要考虑，现在来看代码测试。

### 14.10.3　代码测试的可行性

代码测试最常见的形式要求对模块通过的每条路径最少执行一次。

- 要看这种可能性，考虑图14-8的代码段，对应的流程图如图14-9所示。即使这个流

程图看起来很小，它也有超过 $10^{12}$ 个不同的路径。有 5 个可能的路径穿过中央 6 个带阴影的方框，穿过流程图的可能的路径总数因此是：

$$5^1 + 5^2 + 5^3 + \cdots + 5^{18} = \frac{5 \times (5^{18} - 1)}{(5 - 1)} = 4.77 \times 10^{12}$$

```
read (kmax)                          // kmax is an integer between 1 and 18
for (k = 0; k < kmax; k++) do
{
   read (myChar)                     // myChar is the character A, B, or C
   switch (myChar)
   {
     case 'A':
       blockA;
       if (cond1) blockC;
       break;
     case 'B':
       blockB;
       if (cond2) blockC;
       break;
     case 'C':
       blockC;
       break;
   }
   blockD;
}
```

图 14-8 一个代码段

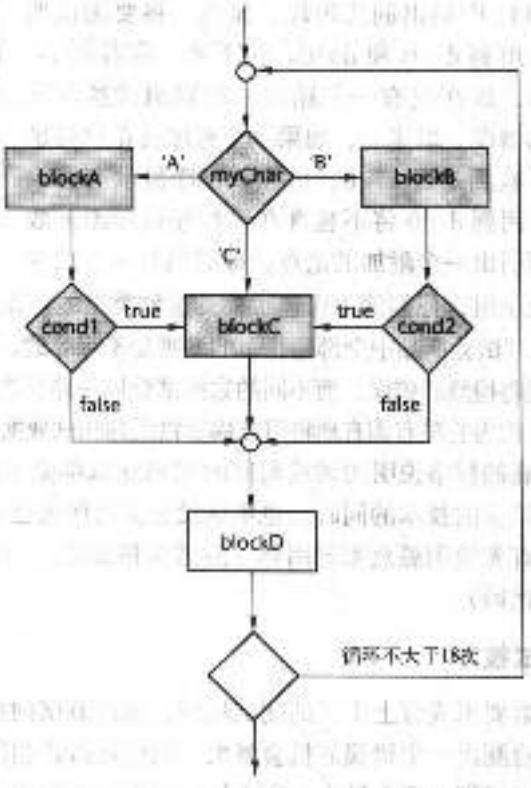



图 14-9 带有通过 $10^{12}$ 个可能路径的流程图

如果一个包含单个循环的简单流程图就有这么多的路径，不难想像在一个有相当规模和复杂度的模块中的不同路径总数，更不要说带有许多循环的大模块了。简而言之，可能路径的庞大数量致使彻底的代码测试是不可行的，就像彻底的规格说明测试是不可行的一样。

- 进一步地，代码测试要求测试者试验每条路径，有可能试验每条路径但没有检测出产品中的每个错误，即代码测试是不可靠的。要明白这一点，考虑图14-10中显示的代码段［Myers，1976］，编写这个代码段是为了测试三个整数x、y和z是否相等，它使用了完全不合理的假定，即如果三个数的平均数等于第一个数，那么这三个数相等。图14-10中显示了两个测试用例，在第一个测试用例中，这三个数的平均值是6/3或2，它不等于1，这个产品因此正确地告诉测试者x、y和z不相等。在第二个测试用例中，整数x、y和z都等于2，该产品计算它们的平均数是2，它等于x的值，因此该产品正确地推算出这三个数相等。这样，经过这个产品的两个路径都检查到了，却没有检测出错误。当然，如果使用像x=2、y=1、z=3这样的测试数据，错误就会显现出来。

```
if ((x + y + z)/3 == x)
  print"x, y, z are equal in value";
else
  print"x, y, z are unequal";

Test case 1:    x = 1, y = 2, z = 3
Test case 2:    x = y = z = 2
```

图14-10　用两个测试用例测试判定三个整数是否相等的一个不正确的代码段

- 路径测试的第三个困难是，一个路径仅在出现时才能测试它。考虑图14-11a中给出的代码段，显然，将要测试两个路径，相应于用例d=0和d≠0。接下来，看看图14-11b的单条语句，现在仅有一条路径，能够测试这条路径，却检测不出错误。事实上，如果一个程序员在他或她的代码中省略了检测是否d=0，可能该程序员没有意识到潜在的危险，用例d=0将不包含在该程序员的测试数据中。这个问题引出一个附加的论点，即应当有一个独立的软件质量保证小组，他们的工作包括检测这种类型的错误。

```
if (d == 0)
  zeroDivisionRoutine ();
else
  x = n/d;
```

a)

```
x = n/d;
```

b)

图14-11　计算商的两个代码段

这些例子最后显示，“试验产品中全部路径”的准则是**不可靠的**，因为存在这样的产品，某些数据试验一个给定路径将检测到错误，而不同的数据试验同一路径将不会检出错误。然而，面向路径的测试是**有效的**，因为它没有固有地将可能揭示错误的测试数据的选择排除在外。

因为组合爆炸，彻底的规格说明测试或彻底的代码测试都是不可行的。为此需要妥协，在使用将尽可能多揭示错误的技术的同时，也承认没有方法保证已检测出全部错误，一个继续下去的合理的办法是首先使用黑盒测试用例（测试规格说明），然后使用玻璃盒测试开发额外的测试用例（测试代码）。

## 14.11　黑盒单元测试技术

彻底的黑盒测试通常要求成百上千亿的测试用例，因此测试的技巧是设计一个较小、可管理的测试用例集，使检测出一个错误的机会最大，同时通过让相同的错误由多个测试用例检出从而使浪费一个测试用例的机会最小。所选择的每个测试用例必须能够检出一个先前未

检出的错误，一个这样的黑盒技术是结合了边界值分析的等价测试。

### 14.11.1 等价测试和边界值测试

假定一个数据库产品的规格说明指出，该产品必须能够处理任何从 1 到 16 383（$2^{14}-1$）个记录，如果该产品能够处理 34 个记录和 14 870 个记录，那么它在比如说 8 252 个记录时工作良好的可能性很大。事实上，如果选择任何从 1 到 16 383 个记录的测试用例，检测出一个错误的机会（如果出现的话）很可能同样地大。相反，如果该产品对于在 1 到 16 383 的范围内的任何一个测试用例工作正常，那么它可能对范围内的任何其他测试用例也工作正常。从 1 到 16 383 的范围构成了一个**等价类**，即一个这样的测试用例集，该类的任何一个成员与任何其他成员是同样好的测试用例。为了更准确，该产品必须能够处理的记录数的规定范围定义了三个等价类：

等价类 1。比 1 个记录小。

等价类 2。从 1 到 16 383 个记录。

等价类 3。多于 16 383 个记录。

那么使用等价类技术测试数据库产品要求从每个等价类中选择一个测试用例，应当正确处理来自等价类 2 的测试用例，而对来自等价类 1 和等价类 3 的测试用例应当打印错误消息。

一个成功的测试用例能检测出先前未检测到的错误，为了使发现这样的错误的机会最大，一个高回报的技术是**边界值分析**。

经验表明，当选择一个处在或接近一个等价类的边界的测试用例时，检测出一个错误的可能性增大，因此，当测试这个数据库产品的时候，应当选择 7 个测试用例：

测试用例 1。0 个记录：等价类 1 的成员，临近边界值。

测试用例 2。1 个记录：边界值。

测试用例 3。2 个记录：临近边界值。

测试用例 4。723 个记录：等价类 2 的成员。

测试用例 5。16 382 个记录：临近边界值。

测试用例 6。16 383 个记录：边界值。

测试用例 7。16 384 个记录：等价类 3 的成员，临近边界值。

这个例子应用于输入规格说明，一个同样功能强大的技术是检查输出规格说明。例如，在 2003 年，美国税收规则允许从个人薪水中扣除的最少社会保险减扣，或者更准确地说，最少老龄、幸存和残疾保险（Old-Age, Survivors, and Disability Insurance, OASDI）减扣是 0.00 美元，最多是 5 394.00 美元，后者相应于 87 000 美元的总收入。因此，当测试一个工资单产品时，工资单中的社会保险减扣的测试用例，应当包括期望正好产生 0.00 美元和 5 394.00 美元减扣的输入数据。此外，建立的测试数据应当能产生比 0.00 美元小和比 5 394.00 美元多的减扣。

一般来说，对于列在输入或输出规格说明中的每个范围（$R_1$，$R_2$），应当选择 5 个测试用例，对应于比 $R_1$ 小的值、等于 $R_1$ 的值、比 $R_1$ 大的值但小于 $R_2$ 的值、等于 $R_2$ 的值、比 $R_2$ 大的值。无论在哪里定义，一个数据项必须是某一集合的元素（例如，输入必须是一个字母），必须测试两个等价类：规定的集合的元素和不是该集合的元素。无论规格说明在哪里拟定一个精确的值（例如，响应必须后面跟一个 # 标记），则同样还有两个等价类，规

定值和其他任意值。

与边界值分析一道，使用等价类测试输入规格说明和输出规格说明，对于产生相当小的测试数据集，揭示可能隐藏着的一些未发现的错误来说，比起使用的用于测试数据选择的不很有力的技术，它是一个有价值的技术。

等价测试的过程概括在下面的“如何完成”部分中。

**如何完成等价测试**

- **对于**输入和输出规格说明

  **对于**每个范围（$L$，$U$）

  选择5个测试用例：小于 $L$、等于 $L$、比 $L$ 大但比 $U$ 小、等于 $U$ 以及大于 $U$。

  **对于**每个集合 $S$

  选择2个测试用例：一个 $S$ 的元素和一个非 $S$ 的元素。

  **对于**每个精确值 $P$

  选择2个用例：$P$ 和其他任何值。

### 14.11.2　功能测试

一个可选的黑盒测试形式是根据模块的功能选择测试数据。在功能测试中［Howden，1987］，要区别每个功能项或在模块中实现的功能。在一个计算机化仓库软件产品的模块中，典型的功能可能有 get _ next _ database _ record（取下一个数据库记录）或 determine _ whether _ quantity _ on _ hand _ is _ below _ the _ recorder point（决定是否库存的数量低于再订购点）。在一个武器控制系统中，一个模块可能包括 compute _ trajectory（计算弹道）功能。在一个操作系统的模块中，一个功能可能是 determine _ whether _ file _ is _ empty（确定文件是否是空的）。

在确定代码模块所有的功能后，设计测试数据来分别测试每个功能。现在对功能测试采取进一步的步骤，如果该代码模块由低级功能按层次组成，这些低层功能由结构化编程的控制结构连接起来，那么功能测试递归进行。例如，如果一个高层功能具有如下形式：

```
<高层功能> :: = if <条件表达式>
                    <低层功能 1>;
                else
                    <低层功能 2>;
```

那么，因为<条件表达式>、<低层功能1>和<低层功能2>已经过功能测试，可以使用分支覆盖对<高层功能>进行测试，它是一种在14.13.1节中讲述的玻璃盒测试技术。注意这种形式的结构化测试是一种混合技术——低层功能使用黑盒技术测试，但高层功能使用玻璃盒技术测试。

然而在实际中，高层功能不是使用这样的结构化方式从低层功能构建而来。相反，低层功能常常以某种方式相互绞在一起。为了确定这种情形下的错误，需要进行**功能分析**，有关一个有些复杂的过程的细节，请见［Howden，1987］。一个更加复杂的因素是功能经常与模块边界不一致，因此，单元测试和集成测试之间的区别变得不明显，一个代码模块不可能在测试的同时，不测试它使用的其他模块的功能。这个问题也出现在面向对象范型中，当一个

对象的方法向另一个对象的方法发送消息（调用它）时。

从功能测试的角度看，模块间随意的相互关系对于管理者来说可能是不可接受的结果。例如，里程碑和最终期限会变得不清楚，使得难于确定关于软件项目管理计划的产品状态。

## 14.12 黑盒测试用例：Osbert Oglesby 实例研究

图 14-12 和图 14-13 包含 Osbert Oglesby 实例研究的黑盒测试用例。首先考虑从等价类和边界值分析派生出来的测试用例。图 14-12 中的第一个测试用例集测试如果画家的 first name（名）不以字母开始时，产品是否能检测出错误。这产生了两个测试用例，分别是词首非字母的字符（应作为输入错误标记出来）和词首为字母的字符（可接受的）。接下来的 5 个测试用例检查一个画家的 first name，它包含 1 到 21 个字符。为画家的 last name（姓）建立同样的测试用例集。类似的测试用例检查规格说明中的其他语句，如图 14-12 所示。

现在看功能测试，在规格说明文档中列出了 17 项功能，如图 14-13 所示，另外 3 个测试用例对应这些功能的错误使用。

**画作数据**

| | |
|---|---|
| first name（名）和 last name（姓）的等价类 | |
| 1. 第一个字符不是字母 | 错误 |
| 2. 第一个字符是字母 | 可接受 |
| 3. 小于 1 个字符 | 错误 |
| 4. 1 个字符 | 可接受 |
| 5. 1 个和 21 个字符之间 | 可接受 |
| 6. 21 个字符 | 可接受 |
| 7. 大于 21 个字符 | 可接受（截短到 21 个字符） |
| title（标题）的等价类 | |
| 1. 小于 1 个字符 | 错误 |
| 2. 1 个字符 | 可接受 |
| 3. 1 个和 41 个字符之间 | 可接受 |
| 4. 41 个字符 | 可接受 |
| 5. 大于 41 个字符 | 可接受（截短到 41 个字符） |
| painting date（画作日期）的等价类 | |
| 1. 形式为 mm/dd/yyyy 的有效日期 | 可接受 |
| 2. 在合适的位置遗漏“/” | 错误 |
| 3. 在日或月里遗漏前导的 0 | 错误 |
| 4. 月的组成部分<1 | 错误 |
| 5. 月的组成部分>12 | 错误 |
| 6. 日的组成部分<1 | 错误 |
| 7. 日的组成部分>31 | 错误 |
| 8. 年的组成部分<0 | 错误 |
| 9. 年的组成部分>99 | 错误 |
| （额外的测试可用来检查天数对应月份是有效的，例如二月不应有 31 天）。 | |
| medium（材质）的等价类 | |
| 1.“油画” | 可接受 |
| 2.“水彩” | 可接受 |

图 14-12 从等价类和边界值分析派生出来的 Osbert Oglesby 实例研究的黑盒测试用例

| | |
|---|---|
| 3. “其他” | 可接受 |
| 4. 其他任何字符串 | 错误（无效值） |
| subject（主题）的等价类 | |
| 1. “肖像” | 可接受 |
| 2. “静物” | 可接受 |
| 3. “风景” | 可接受 |
| 4. “其他” | 可接受 |
| 5. 其他任何字符串 | 错误（无效值） |
| height（高）和 width（宽）的等价类 | |
| 1. <0.00 | 错误 |
| 2. 0.00 | 错误 |
| 3. 0.01 | 可接受 |
| 4. 0.01 和 9 999.99 之间 | 可接受 |
| 5. 10 000.00 | 错误 |
| 6. >10 000.00 | 错误 |
| 7. 不是整数的字符 | 错误（不是一个数） |
| **画廊数据** | |
| （属性分类、购买日期、售出日期、按算法算出的价格和目标价格由系统确定，不由用户输入） | |
| seller name（出售者姓名）和 buyer name（购买者姓名）的等价类 | |
| 1. 第一个字符不是字母 | 错误 |
| 2. 小于 1 个字符 | 错误 |
| 3. 1 个字符 | 可接受 |
| 4. 1 个和 21 个字符之间 | 可接受 |
| 5. 21 个字符 | 可接受 |
| 6. 大于 21 个字符 | 可接受（截短到 21 个字符） |
| seller address（出售者地址）和 buyer address（购买者地址）的等价类 | |
| 1. 小于 1 个字符 | 错误 |
| 2. 1 个字符 | 可接受 |
| 3. 1 个和 25 个字符之间 | 可接受 |
| 4. 25 个字符 | 可接受 |
| 5. 大于 25 个字符 | 可接受（截短到 25 个字符） |
| purchase price（购买价格）和 selling price（售出价格）的等价类 | |
| 1. <0.00 | 错误 |
| 2. 0.00 | 可接受 |
| 3. 0.01 | 可接受 |
| 4. 0.01 和 999.99 之间 | 可接受 |
| 5. 1000.00 | 错误 |
| 6. >1000.00 | 错误 |
| 7. 不是整数的字符 | 错误（不是一个数） |
| **流行度数据** | |
| first name（名）和 last name（姓）的等价类 | |
| 1. 第一个字符不是字母 | 错误 |
| 2. 小于 1 个字符 | 错误 |

图 14-12（续）

| | |
|---|---|
| 3. 1 个字符 | 可接受 |
| 4. 1 个和 21 个字符之间 | 可接受 |
| 5. 21 个字符 | 可接受 |
| 6. 大于 21 个字符 | 可接受（截短到 21 个字符） |
| coefficient（系数）的等价类 | |
| 1. <0.00 | 错误 |
| 2. 0.00 | 错误 |
| 3. 0.01 | 可接受 |
| 4. 0.01 和 9 999.99 | 可接受 |
| 5. 10 000.00 | 错误 |
| 6. >10 000.00 | 错误 |
| 7. 不是整数的字符 | 错误（不是一个数） |

图 14-12（续）

规格说明文档中列出的功能用来创建测试用例。

1. 购买一幅杰作，在拍卖记录中找不到该画家。
2. 购买一幅杰作，在拍卖记录中可找到该画家。
3. 购买一幅大作，在拍卖记录中找不到该画家。
4. 购买一幅大作，在拍卖记录中可找到该画家。
5. 更新几个画家的流行度系数。
6. 购买一幅“其他”类型的画作，在流行记录中找不到该画家。
7. 购买一幅“其他”类型的画作，在流行记录中可找到该画家。
8. 购买一幅画作，购买价格低于算法建议的价格。
9. 购买一幅画作，购买价格等于算法建议的价格。
10. 购买一幅画作，购买价格高于算法建议的价格。
11. 售出一幅画作，售出价格低于目标价格。
12. 售出一幅画作，售出价格等于目标价格。
13. 售出一幅画作，售出价格高于目标价格。
14. 对于一个或多个画家，售出至少两幅画，每幅画的售出价格均高于目标价格（测试用例 17 中使用）。
15. 显示已买画作的报表。
16. 显示售出画作的报表。
17. 显示流行趋势的报表。

除了这些直接测试，有必要进行下面的进一步测试：

18. 尝试购买画廊里已经有的一幅画。
19. 尝试售出一幅画廊里没有的画。
20. 尝试卖出一幅已售出的画作。

图 14-13　Osbert Oglesby 实例研究的功能分析

只要分析流完成了，应可以开发这些测试用例，知道这一点很重要，它们出现在这里的惟一原因是测试用例选择是本章的一个主题，而不是在前面的章节。每个测试计划的主要部分应是一个约定，只要批准了分析成果，就应提出黑盒测试用例，供 SQA 小组在实现流期间使用。

## 14.13　玻璃盒单元测试技术

在玻璃盒测试技术中，基于代码的检查，而不是规格说明的检查来选择测试用例。有一些不同形式的玻璃盒测试，包括语句、分支以及路径覆盖。

### 14.13.1　结构测试：语句、分支和路径覆盖

最简单形式的玻璃盒测试是**语句覆盖**（statement coverage），即运行一系列测试用例，在运行期间每个语句最少执行一次。为了跟踪哪个语句仍在执行，一个 CASE 工具保留有在一系列测试中每个语句被执行了多少次的记录，PureCoverage 是一个这种工具的例子。

这个方法的一个缺点是不能保证对分支的所有输出都充分地测试了。为了说明这一点，考虑图 14-14 的代码段，这个程序员犯了一个错误，复合条件 s>1 && t ==0 应当是 s>1 || t ==0，该图中显示的测试数据能够执行语句 x = 9，但没有发现错误。

```
If (s > 1 && t == 0)
  x = 9;

Test case:    s = 2, t = 0.
```

图 14-14　带测试数据的代码段

语句覆盖的一个改进是**分支覆盖**（branch coverage），即运行一系列测试，确保所有的分支最少测试一次。同样，通常需要一个工具帮助测试者知道哪条分支已经或还未测试，Generic Coverage Tool（*gct*）是 C 程序的一个分支覆盖的例子。像语句或分支覆盖的技术称为**结构测试**（structural test）。

功能最强大的结构测试的形式是**路径覆盖**（path coverage），即测试所有的路径。如前面所示，在一个带循环的产品中，路径数确实会非常大，结果，研究人员一直在研究降低需检查的路径数量的方法，虽然还是比使用分支覆盖找到了更多的错误。选择路径的一个原则是将测试用例局限于**线性代码序列**［Woodward，Hedley，and Hennell，1980］，为此，首先标识出控制流可以跳出的点的集合 L，集合 L 包括入口和出口点，以及分支语句，如一个 **if** 或 **goto** 语句。那么线性代码序列是那些始于 L 的一个元素，并且终止于 L 的一个元素的路径。这项技术一直很成功，它曾经发现许多错误，而不必测试每条路径。

另一个减小测试路径数的方法是**完全定义－使用路径覆盖**（all-definition-use-path coverage）［Rapps and Weyuker，1985］。在这项技术中，源代码中变量（比如说变量 pqr）的每次出现，要么是变量的定义，如 pqr = 1 或 read (pqr)；要么是变量的使用，如 y = pqr + 3 或 **if**（pqr<9）errorB ()。在变量定义和定义的使用之间的全部路径都被标出，现在是通过一个自动工具来完成。最后，为每个这样的路径建立一个测试用例。完全定义－使用路径覆盖是一个优秀的测试技术，通过相当少的测试用例可以频繁检测出大量错误。然而，完全定义－使用路径覆盖也有缺点，它的路径数的上边界是 $2^d$，这里 $d$ 是产品中判决语句（分支）的个数。可以构建例子展示上边界，然而，对于实际的产品已经显示出，与人工的例子不同的是，这个上边界是达不到的，实际的路径数是与 $d$ 成比例的［Weyuker，1988a］。换句话说，完全定义－使用路径覆盖需要的测试用例数通常比理论的上边界小很多。因而，完全定义－使用路径覆盖是一个实用的测试用例选择技术。

当使用结构测试时，测试者可能只是没有提出一个检查某一语句、分支或路径的测试用例，可能发生的是，一个不可能路径（“死代码”）存在于模块中，即对任何输入数据不可能执行那个路径。图 14-15 显示两个不可能路径的例子，在图 14-15a 中，程序员遗漏了一个减

号，如果 k 小于 2，那么 k 不可能比 3 大，因此，语句 x = x * k 不可能执行到。同样，在图 14-15b 中，j 永远不会比 0 小，因此语句 total = total + value [j] 可能永远执行不到，程序员已经认识到了测试是 j < 10，但是犯了一个书写错误。一个使用语句覆盖的测试者不久就会意识到两个语句都执行不到，从而可以找到错误。

```
if (k < 2)
{
   if (k > 3)          [should be k > -3]
      x = x * k;
}
```

a)

```
for (j = 0; j < 0; j++) [should be j < 10]
   total = total + value[j];
```

b)

图 14-15　不可能路径的两个例子

## 14.13.2　复杂性度量

质量保证观点提供另一个玻璃盒单元测试的方法。假定一个管理者被告知代码模块 m1 比代码模块 m2 更复杂，且不管术语“复杂”是如何准确定义的，管理者直觉上相信 m1 可能比 m2 有更多的错误。沿着这条思路，计算机科学家已经开发出一些软件**复杂性**度量，以帮助确定哪个代码模块更可能有错误。如果发现一个代码模块的复杂度不合理地高，管理者可能直接要求对它重新设计和重新实现，与试图调试一个有错的代码模块相比，可能从头开始的代价更小，速度更快。

预报错误数的一个简单的度量是代码行数。隐含的假定是一行代码包含一个错误，它存在一个恒定概率 $p$。如果一个测试者相信，平均而言，一行代码有 2% 的机会包含一个错误，接受测试的模块是 100 行，那么这意味着此模块预计包含 2 个错误；而一个比它长两倍的模块可能有 4 个错误。[Basili and Hutchens，1983] 和 [Takahashi and Kamayachi，1985] 指出，错误数确实与整体上软件产品的大小有关。

人们已经努力寻找更复杂的基于产品复杂性度量的错误预报器，一个典型的竞争产品是 McCabe [1976] 的**秩复杂性**度量，它是二进制判决（预计）数加 1。如 13.15 节中所述，秩复杂度本质上是代码模块中的分支数，因此，秩复杂度可以用作一个代码模块的分支覆盖所需的测试用例数的度量，这是所谓的**结构测试**的基础 [Watson and McCabe，1996]。

McCabe 的度量几乎可以像代码行一样容易计算。在某些情况下已经显示出，它是预报错误的一个很好的度量，$M$ 值越高，一个代码模块包含错误的可能性越大。例如，Walsh [1979] 分析了 Aegis 系统（一个舰只海战系统）中的 276 个模块，Walsh 测量了秩复杂度 $M$，发现 23% 的 $M$ 大于或等于 10 的模块含有 53% 的检出错误。另外，$M$ 大于或等于 10 的模块与 $M$ 小于 10 的模块相比，每行代码含有的错误多 21%。然而，在 [Shepperd，1988b] 和 [Shepperd and Ince，1994] 中，对于 McCabe 的度量的有效性，无论对其理论根据还是对其许多试验的基础都提出了严厉的质疑。

Musa、Iannino 和 Okumoto [1987] 分析了有关错误密度的可用数据。他们得出结论，多数复杂性度量包括 McCabe 的度量，显示出与代码行数很高的相关性，或者更准确地说，与交付的可执行的源代码指令数有很高的相关性。换句话说，当研究者测量一个代码模块或产品的复杂度的时候，他们得到的结果很大程度上可能是代码行数的反映，它又与错误数有很强的相关性。此外，复杂性度量对通过代码行数预报错误几乎没有什么改进，其他复杂性的问题在 [Shepperd and Ince，1994] 中讨论。

## 14.14　代码走查和审查

6.2 节举了一个通常使用走查和审查的极端的情况，对于代码走查和审查也有同样的争论。简单地说，这两个基于非执行技术的错误检测功能导致了快速、彻底和较早的错误检测，用于代码走查或审查的额外的时间，由于在集成阶段较少错误的出现而增加了生产率，这样它就不仅仅是快速了。进一步说，代码审查可降低高达 95% 的纠错性维护成本［Crossman，1982］。

有关为什么应当进行代码审查的另一个原因是，基于执行的测试（测试用例）会在两个方面代价非常大。第一，它消耗大量时间；第二，与基于执行的测试相比，审查可以使错误在软件生命周期的早期得到检测和纠正。如图 1-5 所示，越早地检测和纠正一个错误，它花费的成本越少。运行测试用例的一个极端的高成本的例子是 NASA 的阿波罗计划，软件开支的 80% 消耗在测试上［Dunn，1984］。

支持走查和审查的进一步的观点在下一节中给出。

## 14.15　单元测试技术的比较

曾有一些研究对单元测试策略进行了比较。Myers［1978a］比较了黑盒测试、黑盒和玻璃盒测试的结合、第三方代码走查，试验由 59 个经验丰富的程序员进行，他们测试相同的产品。这三项技术在发现错误方面同样有效，但是代码走查比其他两项技术成本低。Hwang 比较了黑盒测试、玻璃盒测试和由一个人所做的代码阅读［Hwang，1981］。发现所有这三项技术同样有效，每项技术都有其优缺点。

一项主要试验是由 Basili 和 Selby 完成的［1987］。比较的技术与 Hwang 的试验中的一样：黑盒测试、玻璃盒测试和一人代码阅读。试验者是 32 个专业程序员和 42 个高年级的学生，每个人每次使用一项测试技术测试三个产品，使用分数因子设计［Basili and Weiss，1984］补偿由不同的参加者使用不同的方法对产品进行的测试，没有参加者用一种以上的方法测试同一个产品。最后从两组参加者中得到了不同的结果。专业程序员用代码阅读比用其他两种技术检测出更多的错误，错误检测速率较快。两组高年级学生参加了试验，在一个组中，在三项技术中没有发现明显的差别；在另一个组中，代码阅读和黑盒测试同样地好，它们都比玻璃盒测试性能好。然而，学生们检测错误的速率对于所有这三项技术都是一样的。总的来说，代码阅读比其他两项技术检测出更多的接口错误，而黑盒测试在发现控制错误方面更成功。从这个试验中可以得出的主要结论是，代码审查在检测错误方面最起码与玻璃盒测试和黑盒测试一样成功。

很好地利用这个结论的一项开发技术是净室软件开发技术。

## 14.16　净室

**净室**（Cleanroom）技术［Linger，1994］是一些不同的软件开发技术的组合，包括一个递增生命周期模型、分析和设计的形式化技术，以及像代码阅读［Mills，Dyer，and Linger，1987］、代码走查和审查这样的基于非执行的单元测试技术。净室的一个重要特性是一个代码模块必须通过审查后才编译它，即一个代码模块仅在基于非执行测试成功完成后才应当进行编译。

这项技术已有一些很成功的应用例子。例如，使用净室为美军海军水下系统中心开发了一个原型自动化文档系统［Trammel，Binder，and Snyder，1992］。尽管在设计阶段接受“功能检验”——一个采用正确性证明技术的评审过程（6.5 节）时，检测出 18 个错误，还

是尽可能采用了一个如6.5.1节所给出的非形式化证明，仅在审查者不完全相信受审查设计部分的正确性时，才给出完全的数学证明。在对1820行的FoxBASE代码进行走查期间，又检测出19个错误，当编译代码时没有任何编译错误。进一步地说，在执行期间没有任何故障。这是展现基于非执行测试技术能力的另一个示例。

这当然是给人留下深刻印象的结果。尽管前面指出过，应用于小规模软件产品的结果不一定能推广到大规模软件，但在净室的情形下，用于大规模产品的结果也是很有效的。相关的度量是**测试错误率**，即每KLOC（thousand lines of code，千行代码）检测出的总错误数，它是软件业界一个相当通用的度量。然而，当净室技术与传统开发技术对比使用时，这个技术的度量方法有很大的不同。

如6.6节所指出的那样，当使用传统的开发技术时，一个代码模块在开发时由它的编程者非正式地测试，然后该模块由SQA小组系统地进行测试。开发代码时由程序员检测出的错误不进行记录，然而，从模块离开编程者的个人工作间到递交给SQA小组接受基于执行的和基于非执行的测试的这段时间里，检测出的错误总数是要记录的。与此相反，当使用净室技术时，测试错误是从编译时算起的。错误计数则延续到基于执行的测试。换句话说，当使用传统开发技术时，由程序员所做的非正式的错误检测不计入测试错误率；当使用净室技术时，在编译之前的检查和其他基于非执行测试期间的测试过程是记录的，但是它们不计入测试错误率。

关于17个净室产品的报告在［Linger，1994］中可以找到。例如，采用净室技术开发350 000行爱立信Telecom OS32操作系统，该产品由一个70人小组在18个月内开发完成，测试错误率仅为每千行代码1.0个错误。另一个产品是前面介绍的原型自动化文档系统，其测试结果是对于1820行程序来说，每千行代码0.0个错误。17个产品总共将近100万行代码，加权平均测试错误率是每千行代码2.3个错误，Linger认为这是一个了不起的质量成就，这种表扬并不为过。

## 14.17 测试对象时潜在的问题

提出使用面向对象范型的众多原因之一是它降低了对测试的要求。通过继承的重用是该范型的一个主要的长处，一旦一个类已经测试了，变量传送了，就没有必要再测试它了。进一步地，在这样一个测试类的子类内定义的新的方法必须经过测试，但继承的方法不需要进一步地测试。

事实上，这两个说法都只是部分正确。此外，对象的测试提出某些特定于面向对象的新问题，这些问题在这里讨论。

首先，有必要澄清一个与类和对象的测试有关的问题。如7.7节所解释的，类是一个抽象的数据类型，它支持继承，而对象是类的一个实例，即类没有具体的实现，而对象是在一个特定环境内执行的代码的物理块。因此，不可能在一个类上进行基于执行的测试，仅能进行基于非执行的测试，如可以做审查。

信息隐藏和许多方法由相当少的代码行组成的事实，会对测试有相当重要的影响。首先考虑一个使用结构化范型开发的产品，现在，这样一个产品通常由具有大约50条可执行指令的模块组成。在一个模块和该产品的其余部分之间的接口是参数列表，参数有两种：当模块被调用时提供给模块的输入参数，以及当一个模块将控制返回给调用模块时，该模块返回的输出参数。测试一个模块由以下几个步骤组成：向输入参数提供数值，以及调用该模块然后将输出参数值与预期的测试结果做比较。

相反，一个“典型的”对象可能包含 30 个方法，它们中的许多是相当小的，常常只有两个或三个可执行语句［Wilde，Matthews，and Huitt，1993］。这些方法不向调用者返回一个值，但却改变对象的状态，即这些方法修改对象的属性（状态变量）。这里的困难在于，要测试已经正确地进行了状态的改变，有必要向对象发送额外的消息。例如，考虑 1.9 节描述的银行账户对象，方法 deposit 的作用是增加状态变量 accountBalance 的值，然而，作为信息隐藏的结果，测试某一 deposit 方法是否已经正确执行的方式是，在调用方法 deposit 之前和之后，调用方法 determineBalance 并且看储蓄余额是如何变化的。

如果该对象不包括可以被调用以确定所有状态变量值的方法，情形就更糟。一个替代的方法是为此目的包括另外的方法，然后使用条件编译以确保除了测试用途之外它们是不可用的（在 C++ 中，用 #ifdef 达到这个目的）。测试计划（9.6 节）应当规定每个状态变量的值在测试期间是可访问的，为了满足这个需求，在设计阶段，需要把返回状态变量值的附加方法加到有关的类中。结果是，有可能通过查询应用状态变量的值，来测试调用一个对象的特定方法的效果。

非常令人不解的是，仍需测试一个继承的方法，也就是说，即使一个方法已经充分地测试了，当被一个子类不加改变地继承时，同样的方法可能需要完全地测试。要明白这后一个观点，考虑图 14-16 中显示的类层次。在基类 **RootedTree** 中定义了两个方法 displayNodeContents 和 printRoutine，这里方法 displayNodeContents 使用方法 printRoutine。

```
class RootedTree
{
  ...
  void displayNodeContents (Node a);
  void printRoutine (Node b);
//
// method displayNodeContents uses method printRoutine
//
  ...
}

class BinaryTree extends RootedTree
{
  ...
  void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from class RootedTree
//
  ...
}

class BalancedBinaryTree extends BinaryTree
{
  ...
  void printRoutine (Node b);
//
// method displayNodeContents (inherited from BinaryTree) uses this
// local version of printRoutine within class BalancedBinaryTree
//
  ...
}
```

图 14-16　树层次的 Java 实现

接下来考虑子类 **BinaryTree**，这个子类从它的基类 **RootedTree** 继承了方法 printRoutine。此外，定义了新的方法 displayNodeContents，覆盖在 **RootedTree** 中定义的同名的方法。这个新的方法仍使用 printRoutine。在 Java 表示中，BinaryTree.displayNodeContents 使用 RootedTree.printRoutine。

现在考虑子类 **BalancedBinaryTree**。这个子类从它的超类 **BinaryTree** 继承了方法 displayNodeContents；然而，定义了一个新的方法 printRoutine，它覆盖了在 **RootedTree** 中定义的那个方法。当 displayNodeContents 在 **BalancedBinaryTree** 环境中使用 printRoutine 时，C++ 和 Java 中的作用域规则指定使用 printRoutine 的局部形式。在 Java 表示中：当在 **BalancedBinaryTree** 的词法范围内调用方法 BinaryTree.displayNodeContents 时，它使用方法 BalancedBinaryTree. printRoutine。

因此，当在 **BinaryTree** 的实例内调用 displayNodeContents 时，实际执行的代码（方法 printRoutine）与在 **BalancedBinaryTree** 的实例内调用 displayNodeContents 时执行的代码不同。这种状态保持有效，尽管方法 displayNodeContents 本身被 **BalancedBinaryTree** 从 **BinaryTree** 处不加改变地继承。因此，即使方法 displayNodeContents 已在 **BinaryTree** 对象内完全测试过了，当在 **BalancedBinaryTree** 环境内重新使用该方法时，必须从头开始再测试。更深入一些的话，[Perry and Kaiser，1990] 中解释了为什么需要对不同的测试用例重新测试的理论依据。

必须立即指出，这些复杂性不是摒弃面向对象范型的原因。第一，它们只有在方法（在例子中是 displayNodeContents 和 printRoutine）交互作用时才出现；第二，确定什么时候需要这种重新测试是有可能的 [Harrold，McGregor，and Fitzpatrick，1992]。

假定已经对一个类的实例完全测试过了，那么需要测试一个子类的任何新的或重新定义的方法，同时要测试的还有那些标记了要重新测试的方法，因为它们与其他方法有相互作用。简而言之，声称使用面向对象范型很大程度上降低了对测试的需求是真实的。

现在考虑单元测试的某些管理含义。

## 14.18 单元测试的管理方面

在每个代码模块的开发期间必须做出的一个重要的决定是，多少时间以及因此多少金钱要花费在那个模块的测试上。由于在软件工程上有许多其他的经济因素，成本-效益分析（见 5.2 节）可以发挥有用的作用。例如，根据成本-效益分析可以做出以下决定：正确性证明的成本是否超出某一产品满足其规格说明所保证的效益。成本-效益分析也可以用于下列比较：将运行附加的测试用例的成本与由不适当的测试引起的可交付产品故障的成本进行比较。

还有另一个方法用于决定，是应当继续测试某一代码模块，还是似乎全部的错误已经排除了。可靠性分析技术可以提供剩下的错误数的统计估计。已经提出各种不同的技术，用于确定遗留错误数的统计估计。这些技术蕴含的基本思想是这样的，假定一个代码模块测试一周，在星期一发现了 23 个错误，在星期二又发现了 7 个错误，在星期三又发现了 5 个错误，星期四又发现了 2 个错误，星期五没发现错误。因为错误检测率从 23 个错误开始每天稳步减少一直到无，看起来多数错误已经找到了，测试该模块可以停止了。确定代码中不再有错误出现的概率需要一定程度的数学统计知识，这超出了对本书读者所要求的，因此这里不给出有关细节，对可靠性分析感兴趣的读者可以参考 [Grady，1992]。

## 14.19　何时该重写而不是调试一个代码模块

当SQA小组的成员检测出故障（有错的输出）时，如前面所说，该模块必须返回给最初的程序员进行**调试**，也就是说，进行错误的检测和代码的改正。在某些场合下，宁肯扔掉该模块而从头开始重新设计和重新编码——这或者由最初的程序员完成，或者由另一个可能更高级的开发小组的成员完成。

要明白为什么这可能是必要的，考虑图14-17，该图显示了一个违反直觉的概念，一个模块中存在更多错误的概率与该模块中已经发现的错误数成正比［Myers，1979］。要明白为什么会是这样，考虑两个代码模块——a1和a2。假定这两个代码模块长度接近，都已经过相同小时数的测试。进一步假定在a1中仅检测到2个错误，但在a2中检测到48个错误，很可能在a2中比在a1中仍然存在更多的错误。而且，对a2进行的额外的测试和调试的过程可能更长，而且还存在对a2仍不完善的怀疑。无论从短期看还是从长期看，最好选择丢弃a2，重新设计和重新编码它。

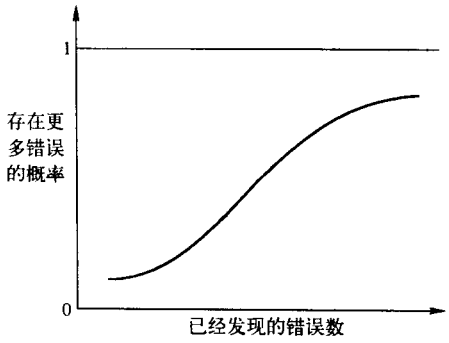



图14-17　仍将发现错误的可能性与已检测出的错误数成比例

模块中的错误分布当然不是一致的。Myers［1979］引用了在OS/370中由用户发现的错误的例子，发现47%的错误仅与4%的模块有关。由Endres［1975］所做的一个早期研究是关于在德国Bblingen的IBM实验室的DOS/VS（版本28）的内部测试，该研究显示了同样的非一致性。在202个模块中检测到的总共512个错误中，仅有一个错误在112个模块中的每个都检测出，另一方面，发现某些模块分别有14、15、19和28个错误。Endres指出后三个模块是产品中最大的三个模块，每个都由超过3000行DOS宏汇编语言组成。而带有14个错误的模块是一个先前被认为非常不稳定的小模块，这种类型的模块是被丢弃和重新编码的主要对象。

管理者解决这种情形的办法是，预先确定在一个给定代码模块的开发期间可允许的最大错误数量，当达到那个最大数值时，必须舍弃该代码模块，然后重新设计和重新编码。有经验的软件专业人员更喜欢选择这种方式。这个最大值将因应用领域的不同而不同，还因模块的不同而不同。毕竟，在一个读取数据库记录并检查该部分数据有效性的模块中，最大允许的检测错误数，应当比一个坦克武器控制系统中的复杂的模块的错误数要小得多，后者必须整理来自各种传感器的数据，并将主要武器的目标指向想要打击的目标。确定某个模块最大错误数的一个办法是检查某个类似的已得到纠错性维护的模块。但是，不管使用什么样的估计技术，如果超出预计的错误数的话，管理者必须保证丢弃该模块（但是请参见下面的“如果你想知道”部分）。

**如果你想知道**

*在一个模块开发期间的最大允许检测错误数，准确地说是指：在开发期间允许的最大数。在产品已经交付给用户后的最大允许检测错误数，对于全部产品的全部模块都应当是零。也就是说，向客户交付无错误代码应当是每个软件工程师的目标。*

## 14.20 集成测试

每个新的代码模块被加入到已集成的模块中时都必须进行测试，这称为**集成测试**。这里的关键点是首先测试新的代码模块，如 14.10 节到 14.14 节所述（单元测试），然后像这个新代码模块集成进来之前一样检查该部分产品的其余功能。

当产品带有图形用户接口时，集成测试会出现新的情况。通常，通过将测试用例的输入数据存入一个文件中可以简化产品的测试。然后执行该产品，相应的数据提交给它。借助基本的相关 CASE 工具，整个过程可自动进行，即设置一组测试用例，同时还有每种情况期望的输出。CASE 工具运行每个测试用例，比较期望的结果与实际结果，并向用户报告每个测试用例的情况。然后保存测试用例，以便当修改产品时用它做回归测试。SilkTest 就是这种测试工具的一个例子。

然而，当产品合并图形用户接口时，这种方法就不起作用了。特别是下拉菜单及点击鼠标按钮的测试数据不能像通常的测试数据一样保存到文件中。但是，手工测试 GUI 是非常繁琐的。解决这个问题的办法是利用特殊的 CASE 工具，它能保存点击鼠标、按键等的记录。手工测试 GUI 一次，以使 CASE 工具能够建立测试文件。其后，将这个文件用于随后的测试中。有许多 CASE 工具支持 GUI 的测试，包括 QAPartner 及 XRunner。

当集成过程结束时，软件产品作为一个整体进行测试，这称为**产品测试**（product testing）。当开发者能确保软件产品各方面正确时，软件产品将交给客户进行**验收测试**（acceptance testing）。这两种形式的测试下面会更深入地讨论。

## 14.21 产品测试

最后一个模块成功地集成到产品中后，并不代表开发人员的任务已经结束了。SQA 小组仍将进行许多测试任务以确保产品是成功的。有两种主要类型的软件：商用现货（COTS）软件（见 1.1 节）及定制软件。COTS 产品测试的目标是确保产品整体上没有错误。当产品测试完成时，对产品进行 α 测试和 β 测试，如 3.7 节所述，即将产品的最初版有选择地送给预期的用户，从他们那里得到反馈意见，特别要关注 SQA 小组没有注意到的残留错误。

而定制软件的产品测试有所不同。SQA 小组进行许多测试任务以确保产品不会在进行验收测试时失败，这是定制软件开发小组必须克服的最后一道障碍。定制软件未能通过其验收测试是开发小组组织管理失败的一个反映。客户会认为开发小组是不胜任的，这会导致客户尽量避免再雇用这些开发者。更糟糕的是，客户会认为这些开发者不诚实，故意移交不合格的产品，以便结束合同更快地拿到他们的酬金。如果客户真的这样认为，并告知其他潜在客户，那么开发者便会面临一个巨大的现实问题。所以说 SQA 小组必须确保产品成功地通过验收测试。

为了确保成功地通过验收测试，SQA 小组必须使用自认为最接近即将到来的验收测试的方式进行产品测试。

- 必须对产品进行黑盒测试。到目前为止，已进行了模块到模块、类到类的基本测试，以确保每个代码模块和类分别满足它们的规格说明。
- 必须对产品进行健壮性测试。在集成期间已进行单个代码模块与类的健壮性测试，现在必须对整个产品的健壮性进行测试。另外产品必须经受**强度测试**（stress test-

ing)，也就是说，当在最大负荷下操作时，产品能正确运行。例如，所有终端在同一时刻都在进行登录或所有客户都同时在操作自动柜员机。同时产品也要经受**容量测试**（volume testing)。例如，确保它可以处理大的输入文件。

- SQA 小组必须检查产品是否满足它的所有限制条件。例如，如果规格说明规定产品在满负荷下工作时对 95% 的查询工作响应时间必须小于 3 秒，那么确保产品满足这种要求是 SQA 的责职。客户在验收测试期间将进行限制条件的测试是毫无疑问的，如果在一个限制条件上未能达到要求，客户将对开发组失去相当大的信任。同样，存储性限制条件与安全性限制条件也必须检查。
- SQA 小组必须检查所有的随代码一起交给客户的文档。SQA 小组必须遵照 SPMP 中的标准检查文档。另外，文档必须与产品吻合。例如，SQA 小组必须保证用户手册能真正指明产品的正确使用方法，并且产品的功能与用户手册中所描述的一样。

一旦 SQA 小组确认产品可以经受验收测试者的任何验收测试，产品（代码及文档）就可以交给客户组织进行验收测试。

## 14.22　验收测试

客户进行验收测试的目的是判定产品是否确实满足开发者所声称的特性。验收测试可以由客户组织进行，也可以由 SQA 小组在有客户代表参加情况下进行，或者由客户雇用的独立 SQA 小组进行。验收测试当然包括正确性测试，但除此之外还有必要包括性能测试和健壮性测试。验收测试的四个主要元素（正确性测试、稳定性测试、性能测试、文档测试）实际上是由开发者在产品测试期间进行的；这并不令人惊奇，因为产品测试就是验收测试的全面预演。

验收测试中关键的一面是，进行验收测试所使用的必须是真实数据而不是测试数据。无论测试用例如何建立，究其本质来讲，它们是人工的。更重要的是，测试数据应该是相应真实数据的真实反映，但实际情况并不总是这样。例如，负责确定实际数据特征的规格说明小组成员可能没有很好地完成任务；或者，即使数据正确地规定了，SQA 小组中使用这些数据规定的成员也可能错误地理解或解释这些数据。结果所产生的测试用例并不能正确反映真实数据，从而导致产生未经充分测试的产品。因此，验收测试必须建立在真实数据的基础上。

当新产品要取代现有产品的时候，规格说明文档中几乎总是包括这样一条，即新产品必须在与现有产品并存的情况下安装使用。其原因是新产品很有可能在某些方面存在错误，而现有产品工作正常但在某些方面有不足之处。如果现有产品被工作不正常的新产品所取代，那么就会给用户带来麻烦。因此，两代产品必须同时存在，直到客户能满意地用新产品代替现用产品的功能。能成功地并行运行时就可以结束验收测试，现有产品就可以退役了。

当产品通过验收测试后，开发者的工作就完成了。对产品所做的任何更改都属于维护工作。

## 14.23　测试流：Osbert Oglesby 实例研究

Osbert Oglesby 产品的 C++ 和 Java 实现（可从 www.mhhe.com/engcs/compsci/schach 下载得到）经过图 14-12 和图 14-13 的黑盒测试用例进行测试，并经过习题 14.30 到习题 14.34 的玻璃盒测试用例进行测试。

## 14.24 实现的CASE工具

在第5章中我们已经对支持代码模块实现的CASE工具进行了详细阐述。对于集成，需要有版本控制工具、创建工具及配置管理工具（第5章）。因为，随着一系列的错误被发现并更正，处在测试中的模块将不断变化，而这些CASE工具对于保证编译及连接每一模块的适当版本是必要的。商业上可用的配置控制平台包括PVCS和SourceSafe。一个流行的开放源代码的配置控制工具是CVS。

到目前的每章里，我们已经对针对每一阶段的CASE工具及平台进行了阐述。既然已经对开发过程的所有阶段进行了阐述，那么现在我们就可以对开发过程的CASE工具进行整体研究了。

### 14.24.1 软件开发全过程的CASE工具

CASE工具有一个自然发展的过程。如5.5节中所描述的，最简单的CASE设备是一个单独的**工具**，比如说一个在线接口检查器或一个创建工具。接下来，工具可以进行组合，由此可以产生一个在软件开发过程中支持一个或两个活动的**工作平台**，比如说配置控制或编码。然而，这样的工作平台甚至于不能为软件开发过程中有限的部分提供可用的管理信息，更不用说为整个项目了。最终发展而成的**开发环境**则可以为开发过程提供（即使不是全部）大部分的计算机辅助支持。

理想情况下，每一个软件开发公司应该使用一种开发环境。但使用开发环境可能耗资巨大，当然并不仅仅是软件包自身，还有运行这种软件开发环境的硬件设备。对于一个相对较小的公司来讲，一个工作平台或者可能仅仅是一套工具就能满足要求了。但是，如果可能的话，应该使用**集成的开发环境**来支持开发和维护工作。

### 14.24.2 集成化开发环境

在CASE环境中，集成化的最普通的含义是**用户接口集成**。也就是说，开发环境中的所有工具共享一个通用的用户接口。这背后所隐藏的思想是，如果所有的工具都有相同的可视界面，那么使用开发环境中某个工具的用户可以毫不费力地学会其中的其他工具。这一思想在Macintosh中已经得到了成功实现，Macintosh中的大部分应用软件都有相似的“外观”。当然这是最普遍应用的含义，也有其他类型的集成。

术语**工具集成**的含义是指所有的工具通过相同的数据格式进行通信。例如在UNIX程序员的工作平台中，UNIX管道格式对所有数据采用ASCII码流的形式。因此通过将一工具的输出流指向另一工具的输入流，可以很容易地将两个工具综合起来。

**过程集成**指的是支持一个具体软件开发过程的开发环境。这一类开发环境的一个分支是**基于技术的开发环境**（technique-based environment）（参见下面的“如果你想知道”部分）。这一类型的开发环境只支持开发软件的某一具体技术，而不是支持全过程。本书中所讨论的各种技术都应用于开发环境中，如Gane和Sarsen的结构化系统分析（见11.3节）、Jackson系统开发（见13.5节）以及Petri网（见11.8节）。这些开发环境主要为分析和设计阶段提供图形支持并集成了数据词典，此外还提供了一些相容性检查。开发环境中还集成了对开发过程管理的支持。有许多这种类型的商业开发环境，包括Analyst/Designer及Rhapsody。Analyst/Designer主

要针对 Yourdon 方法［Yourdon，1989］，Rhapsody 则支持状态图［Harel et al.，1990］。至于面向对象的技术，Rose 支持统一过程［Jacobson，Booch，and Rumbaugh，1999］。除此之外，一些较老的开发环境已经得到了扩展，可以支持面向对象的范型，Software through Pictures 就是这种类型的例子。几乎所有面向对象的开发环境现在都支持 UML。

**如果你想知道**

文献中经常把“基于技术的开发环境”称为**基于方法的开发环境**。面向对象的出现，给了方法这一术语第二种含义（在软件工程环境中），其原始的含义是一种技术或一种途径——这就是“方法”这个词在“基于方法的开发环境”中的含义。面向对象的含义是一个对象或一个类中的动作。遗憾的是，有时它并不像其在上下文中想表达的那样清楚。

因此，本书在面向对象范型环境中不使用“方法”这个词，而采用“技术”或“途径”这种表述。例如，这就是为什么第 11 章从不用“形式化方法”而采用“形式化技术”这一术语的原因。同理，在本章中使用“基于技术的开发环境”这一术语。

大多数基于技术的开发环境都重点强调对由这种技术制定的软件开发人工操作的支持和形式化。也就是说，这些环境迫使用户按照环境开发者所预定的方式一步一步地使用这种技术，同时通过提供图形工具、数据字典及相容性检查来帮助用户。这种计算机化的框架工作加强了基于技术的开发环境，它迫使用户使用并正确地使用指定的技术。但同时这也可能是一个弱点。除非公司的软件开发过程集成了这项指定的技术，否则使用基于技术的开发环境可能达不到预期的目的。

### 14.24.3　商业应用软件开发环境

另一类重要的开发环境用来开发面向商业应用的软件。它强调的是易于使用，以及应该能通过许多方法实现。特别是其中包含着一些标准界面，用户可以通过用户界面友好的 GUI 对其进行各种修改。这种开发环境的一个普遍特征就是代码生成器，产品的最底层抽象由此得到了详细的设计。用户只需对代码生成器进行输入，代码生成器会自动生成基于某种编程语言（例如 C、C++ 或 Java）的代码。用户只需对这些自动生成的代码进行编译，无需对其进行任何编程工作。

规定详细设计的语言将是未来的编程语言。抽象级编程语言已经从物理机（physical machine）级的第一代和第二代编程语言上升到了抽象机（abstract machine）级的第三代和第四代编程语言。到今天，这一类开发环境的抽象级已经是详细设计级——一个可移植级。14.2 节中已经指出，使用第四代语言的目的是代码更简洁，开发更快捷，维护更容易。使用代码生成器则可以更好地实现这些目标，与 4GL 的解释器或编译器不同，程序员只需为代码生成器提供很少的细节即可。因此，支持代码生成器的面向商业应用的开发环境将提高生产率。

当前有许多这类开发环境可用，包括 Oracle Developer Suite。切记面向商用 CASE 开发环境的市场规模，在未来几年中很可能会开发出更多这种类型的开发环境。

### 14.24.4　公共工具基础结构

“欧洲信息技术研究战略计划”（European Strategic Programme for Research in Informa-

tion Technology，ESPRIT）开发了一种支持 CASE 工具的基础结构。尽管拥有**可移植公共工具环境**（portable common tool environment，PCTE）这样的名字［Thomas，1989，Long，Morris，1993］，但它不是一种开发环境。相反，它只是提供 CASE 工具所需服务的一种基础结构，这一点很类似于 UNIX 为其用户产品提供所需的操作系统服务。（PCTE 中“公共”的含义就是“公开的”或“非版权的”意思。）

PCTE 已经得到了广泛接受。例如，PCTE 及其与 C 和 Ada 的接口已经在 1995 年被采用为 ISO/IEC 13719 标准。PCTE 的实现包含了 Emeraude 与 IBM 实现。

我们希望，将来会有更多的 CASE 工具遵守 PCTE 标准，而 PCTE 自身也能在更多种类的计算机上得到应用实现。遵守 PCTE 标准的工具可以在任何支持 PCTE 的计算机上运行，由此也就会有更多的 CASE 工具产生，反过来，这又会带来更好的软件开发过程和更高质量的软件。

### 14.24.5 开发环境的潜在问题

没有任何一种开发环境对所有的产品和所有的公司都是最理想适用的，也没有任何一种编程语言是“最好的”。每一个开发环境都有其优点和缺点，选用一种不合适的环境可能比不使用开发环境还要糟糕。例如，如 14.24.2 节中所解释的，一种基于技术的开发环境本质上是使人工开发过程自动化。如果一个公司选用了一种强制采用某种技术的开发环境，而这种技术从整体上对公司是不合适的，或者对于正在开发的软件产品是不合适的，那么使用这种 CASE 开发环境就达不到预期目的。

更糟糕的情况发生在如果公司忽视 5.10 节中所提出的建议的情况下，即应该严格避免使用 CASE 开发环境，除非公司达到 CMM 3 级。当然，每一个公司都应该使用 CASE 工具，使用工作平台一般情况下不会带来害处。然而，开发环境将自动化的软件开发过程施加于使用它的公司。如果要使用一个好的开发过程，也就是说，该公司为 3 级或更高，那么通过使开发过程自动化，开发环境可以在软件生产的各个方面起到帮助作用。但如果公司只处于危机驱动的 1 级或 2 级，则没有这样的开发过程。对一个不存在的开发过程进行自动化，即引入 CASE 环境（与 CASE 工具或 CASE 工作平台相对应），可能只会带来混乱。

## 14.25 实现流的度量

在 14.13.2 节中我们已经讨论了一些不同的实现与集成阶段的复杂性度量，包括代码行数和 McCabe 的秩复杂性。

从测试的观点来看，相关的度量包括测试用例的数目及导致失败的测试用例的数目。代码审查中必须要统计通常的错误。错误的总体数据很重要，因为如果在一个代码模块中所检测到的错误数超过预定的最大数量后，如 14.19 节中所述，那么这个代码模块就必须重新设计或重新编码。除此之外，还须对所检测到的错误的种类进行详细统计。典型的错误类型包括对设计的错误理解、缺乏初始化及变量使用前后不一致。在未来产品的代码审查过程中，可以将错误数据包含进将要使用的检查表中。

专门针对面向对象的范型的一些度量已经被提出，例如，继承树的高度［Chidamber and Kemerer，1994］。许多这样的度量在理论和实践中都被提出质疑［Binkley and Schach，1996，1997］。它还显示出有对面向对象度量的需求，对应于传统的度量可同样应用于面向对象软件。

## 14.26　实现流面临的挑战

自相矛盾的是，实现流面临的主要问题在实现流之前就已经遇到。如第 8 章中解释的那样，代码重用是降低软件开发成本和交付时间的一个有效途径，然而，如果晚到实现流才做这方面的工作，就很难达到代码重用。

例如，假定决定用 L 语言实现一个产品。现在，在半数的代码模块已经实现和测试后，管理者决定将软件包 P 用于软件产品的图形用户界面。不管例程 P 的功能有多么强大，如果它们是用一种难于与 L 接口的语言编写的，那么它们就不能重用在软件产品中。

即使语言的互操作不成问题，如果重用的东西不能很好地适合设计，那么试图重用一个现有模块也没有什么意义。修改现有的模块可能比从头建立新的模块所做的工作量更大。

代码重用因此必须从一开始就成为软件产品的一个组成部分。重用应当既是用户需求，同时也是规格说明文档的强制要求。软件项目管理计划（9.4 节）必须包含重用。而且，设计文档必须声明将要实现哪个模块，以及将要重用哪个模块。

因此，如本节开始所指出的那样，虽然模块重用是实现流面临的一个重要问题，但模块重用还必须结合在需求、规格说明和设计流中。

从纯技术的观点来看，实现流相对简单。如果需求、分析及设计流令人满意地完成了的话，实现的任务应该不会给有能力的程序员带来什么问题。然而集成的管理特别重要，实现流面临的挑战就在于此。

典型的显著成功或彻底失败的议题包括：使用适当的 CASE 工具（14.24 节），客户签订规格说明后的测试计划（9.6 节），保证设计的改变能够通知所有相关的人员（14.6.5 节），决定何时停止测试并将产品交付给客户（6.1.2 节）。

### 本章回顾

本章给出了与由一个小组完成产品的实现有关的各种问题。它们包括编程语言的选择（14.1 节），第四代语言的问题在 14.2 节中一定程度上深入地进行了讨论，在 14.3 节中介绍了好的编程实践，对实用编码标准的需求在 14.4 节中给出。然后，对有关代码重用做了说明（14.5 节）。实现和集成行为必须并行地实现（14.6 节），描述并对比了自顶向下集成、自底向上集成和三明治集成（14.6.1 节到 14.6.3 节）。面向对象产品的集成在 14.6.4 节中讨论，集成的管理在 14.6.5 节中讨论。实现流在 14.7 节给出，并在 14.8 节应用于 Osbert Oglesby 实例研究。接下来是测试流的实现方面的问题（14.9 节）。测试用例必须系统地选择（14.10 节），对各种黑盒测试、玻璃盒测试以及基于非执行单元的测试技术做了介绍（分别在 14.11 节、14.13 节和 14.14 节中），然后做了比较（14.15 节）。14.12 节给出了 Osbert Oglesby 实例研究的黑盒测试。在 14.16 节中描述了净室技术。在 14.17 节中讨论了对象的测试，随后讨论了单元测试的管理上的实现（14.18 节）。另一个问题是何时重写而不是调试一个模块(14.19 节)，集成测试在 14.20 节描述，产品测试在 14.21 节描述，验收测试在 14.22 节描述。Osbert Oglesby 实例研究的测试流在 14.23 节中列出。在 14.24 节描述了实现流的 CASE 工具。更具体地，完整过程的 CASE 工具在 14.24.1 节中讨论，集成开发环境在14.24.2 节中讨论，商业应用软件的开发环境在 14.24.3 节中给出，14.24.4 节讨论了公共工具基础结构，接下来讨论了开发环境的潜在问题（14.24.5 节）。实现流的度量

在 14.25 节中讨论。最后以对实现阶段面临的挑战的分析结束本章（14.26 节）。

## 进一步阅读

有关 4GL 的广泛的信息源是［Martin，1985］，在［Guimaraes，1985］中报告了 43 家公司对 4GL 的态度。［Klepper and Bock，1995］描述了 McDonnell Douglas 是如何使用 4GL 得到比使用 3GL 更高的生产率。

关于好的编程实践的优秀著作有［Kernighan and Plauger，1974，and McConnell，1993］。

关于基于执行的测试，最重要的早期著作可能是［Myers，1979］。有关一般测试的全面信息源是［Beizer，1990］。功能测试在［Howden，1987］中有描述，结构化技术在［Clarke，Podgurski，Richardson，and Zeil，1989］中做了比较。黑盒测试在［Beizer，1995］中有深入的描述，黑盒测试用例的设计在［Yamaura，1998］中给出。各种结构化测试的覆盖度量和软件质量之间的关系在［Horgan，London，and Lyu，1994］中进行了讨论。在［Stocks and Carrington，1996］中介绍了玻璃盒测试的形式化方法。［Elbaum，Malishevsky，and Rothermel，2002］讨论测试用例优先权的设置问题。

净室在［Linger，1994］中有介绍，在［Sherer，Kouchakdjian，and Arnold，1996］中给出了交付后维护阶段净室的使用，在［Beizer，1997］中给出了净室的准则。

有关软件可靠性的一个很好的参考资料是［Musa and Everett，1990］。此外，每年的“软件可靠性工程国际讨论会”（International Symposium on Software Reliability Engineering）会议录中包含涉及范围广泛的各种有关软件可靠性的文章。

“软件测试和分析国际讨论会”（International Symposia on Software Testing and Analysis）会议录中包含了相当广泛的测试问题。2000 年国际讨论会选用的文章收录在《IEEE Transactions on Software Engineering》杂志 2002 年 2 月刊上。

［Turner，1994］中有关于对象测试的不同方法的调查，关于这个主题的两篇重要的文章是［Perry and Kaiser，1990，and Harrold，McGregor，and Fitzpatrick，1992］。前面提到的［Beizer，1995］中也有面向对象软件的黑盒测试方面的介绍，关于面向对象范型，［Jorgensen and Erickson，1994］描述了面向对象软件的集成测试。

关于实现阶段的度量，McCabe 的秩复杂性是第一次在［McCabe，1976］中提出的，设计阶段度量的扩展出现在［McCabe and Butler，1989］中。对秩复杂性的有效性提出质疑的包括［Shepperd，1988；Weyuker，1988b；and Shepperd and Ince，1994］。［Barnard and Price，1994］中描述了管理代码审查的度量。

集成测试中测试数据的选择在［Harrold and Soffa，1991］中有描述，测试 GUI 的测试用例的生成在［Memon，Pollack，and Soffa，2001］中有描述。

每 2 年或每 3 年，ACM SIGSOFT 和 SIGPLAN 会发起一个有关实用软件开发环境的讨论会，会议录提供了广泛的工具包和开发环境的信息。每年的“计算机辅助软件工程国际工作室”（International Workshops on Computer-Aided Software Engineering）的会议录也非常有用。

关于 PCTE，［Long and Morris，1993］包含了许多相关的信息来源。

## 习题

14.1　你的老师让你实现 Ophelia 的 Oasis 产品（附录 A），你想选择哪种语言实现该产品，为什么？在可使用的各种语言中，列出它们的效益和成本，不要试图给你的答案附上美元值。

14.2　对电梯问题实例研究（11.7.1 节）重复习题 14.1。

14.3　对图书自动循环系统（习题 8.7）重复习题 14.1。

14.4　对确定银行储户报告书是否正确的软件产品（习题 8.8）重复习题 14.1。

14.5　对自动柜员机（习题 8.9）重复习题 14.1。

14.6　给你最近编写的代码模块增加序言注释。

14.7　个人软件生产公司与拥有 300 个软件专业人员的公司的编码标准有什么不同？

14.8　对于开发和维护特别护理单元软件的软件公司，和那些开发和维护财务产品的组织，编码标准有什么不同？

14.9　为 Naur 的文本处理问题（6.5.2 节）建立黑盒测试用例，对于每个测试用例，说明正在测试什么，对该测试用例期望的输出是什么。

14.10　利用你对习题 6.15 的答案（或者你的老师发给你的代码），建立语句覆盖测试用例，对每个测试用例，说明正在测试什么，对该测试用例期望的输出是什么。

14.11　对分支覆盖，重复习题 14.10。

14.12　对完全定义－使用路径覆盖，重复习题 14.10。

14.13　对路径覆盖，重复习题 14.10。

14.14　对线性代码序列，重复习题 14.10。

14.15　画一个你的习题 6.15 的答案（或者你的老师发给你的代码）的流程图。确定它的秩复杂性，如果你不能确定分支数，把流程图当为一个有向图考虑，确定边数 $e$，结点数 $n$，以及连接分量 $c$ 的数量（每个方法组成一个连接分量），秩复杂性 $V$ 由下式给出［McCabe，1976］：

$$V = e - n + 2c$$

14.16　你是一个软件公司惟一的雇主和雇员，你买了 5.6 节描述的编程工具包，按照它对你的重要程度级别，列出它的 5 种能力，给出原因。

14.17　你现在是 Very Big Software 公司的软件技术副总裁，该公司有 17 500 名员工。你如何按级排列 5.6 节描述的编程工具包的能力？解释你对这个问题的答案和对上一题的答案的所有不同之处。

14.18　作为一个软件开发公司的 SQA 管理者，你负责确定测试期间在一个给定模块中可以发现的最大错误数，如果超出这个最大值，那么该模块必须重新设计和重新编码。你将用什么准则确定一个给定模块允许的最大错误数？

14.19　请解释逻辑模块与操作模块间的区别。

14.20　保守编程是一种好的软件工程实践。但同时，在重用时它可能会妨碍对操作模块进行充分完全的测试，那么如何解决这一明显的矛盾呢？

14.21　产品测试与验收测试间的相似之处是什么？其主要区别是什么？

14.22　在实现阶段 SQA 小组的主要职能是什么？

14.23 假定你是某单人软件公司的拥有者和惟一员工。为了使公司更具竞争力，你决定购买 CASE 工具。为此你需要从银行贷款 15 000 美元。银行管理人员要求你出示一份长度不多于一页（越短越好）的说明，以解释为什么你需要这些 CASE 工具。请写出这份说明。

14.24 Ye Olde Fashioned 软件公司新任命的软件开发副总经理雇用你帮助她改变公司开发软件的方式。该公司共有 650 名员工，都在没有任何 CASE 工具的帮助下编写 COBOL 源代码。请为副总经理写一份备忘录，说明公司应该购买什么样的 CASE 工具。仔细斟酌你的选择。

14.25 你和一个朋友决定开始编写个人计算机软件程序“R” Us，即在个人计算机上开发个人计算机应用软件。这时一个远房表兄去世了，给你留下了 100 万美元，要求你将钱花在面向商业的开发环境及此环境需要的硬件上，要求你保持这个环境至少 5 年，你怎么做？为什么？

14.26 你是优秀小型人文艺术学院的一名计算机科学教授。计算机科学课程通过网络编程进行分配（网络由 35 台个人计算机构成）。系主任问你是否需要用有限的软件预算来购买 CASE 工具，请记住，除非可以得到某种站点许可，否则每一种 CASE 工具都需购买 35 份拷贝。对此你有何建议？

14.27 你刚刚被选为一个大城市的市长。你发现在为城市进行的软件开发中没有使用任何 CASE 工具，你怎么办？

14.28 （学期项目）为你在习题 11.15 或习题 12.27 中规定的产品拟制黑盒测试用例。对于每个测试用例，说明正在测试什么，对该测试用例期望的输出是什么。

14.29 （学期项目）实现并集成 Ophelia 的 Oasis 产品（附录 A）。使用指导教师指定的编程语言。指导教师将告诉你是否构造基于网络的用户接口、图形用户接口或基于文本的用户接口。请使用你在习题 14.28 中所开发黑盒测试用例来进行代码测试工作。

14.30 （实例研究）下载一份 14.8 节中描述的 Osbert Oglesby 产品的实现的拷贝，为该产品拟制语句覆盖测试用例。对于每个测试用例，说明正在测试什么，对该测试用例期望的输出是什么。

14.31 （实例研究）对于分支覆盖，重复习题 14.30。

14.32 （实例研究）对于完全定义－使用路径覆盖，重复习题 14.30。

14.33 （实例研究）对于路径覆盖，重复习题 14.30。

14.34 （实例研究）对于线性代码序列，重复习题 14.30。

14.35 （实例研究）从 13.8 节的详细设计开始，使用除 C++ 和 Java 之外的面向对象语言对 Osbert Oglesby 实例研究进行编码。

14.36 （实例研究）用 C 对 Osbert Oglesby 实例研究（14.8 节）重新进行编码，不要使用任何 C++ 特性。尽管 C 代码不支持继承，但类似封装和信息隐藏这样的面向对象的概念也可以很容易地实现。那么你如何实现多态和动态绑定呢？

14.37 （实例研究）对于 14.7 节中实现代码的文档，什么长度是不合适的？请做必要的补充。

14.38 （软件工程读物）你的老师将发给你们［Elbaum，Malishevsky，and Rothermel，2002］的复印件，你认为测试用例应有优先级吗？

## 参考文献

[Barnard and Price, 1994] J. BARNARD AND A. PRICE, "Managing Code Inspection Information," *IEEE Software* **11** (March 1994), pp. 59–69.

[Basili and Hutchens, 1983] V. R. BASILI AND D. H. HUTCHENS, "An Empirical Study of a Syntactic Complexity Family," *IEEE Transactions on Software Engineering* **SE-9** (November 1983), pp. 664–72.

[Basili and Selby, 1987] V. R. BASILI AND R. W. SELBY, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering* **SE-13** (December 1987), pp. 1278–96.

[Basili and Weiss, 1984] V. R. BASILI AND D. M. WEISS, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering* **SE-10** (November 1984), pp. 728–38.

[Beizer, 1990] B. BEIZER, *Software Testing Techniques,* 2nd ed., Van Nostrand Reinhold, New York, 1990.

[Beizer, 1995] B. BEIZER, *Black-Box Testing: Techniques for Functional Testing of Software and Systems,* John Wiley and Sons, New York, 1995.

[Beizer, 1997] B. BEIZER, "Cleanroom Process Model: A Critical Examination," *IEEE Software* **14** (March/April 1997), pp. 14–16.

[Binkley and Schach, 1996] A. B. BINKLEY AND S. R. SCHACH, "A Comparison of Sixteen Quality Metrics for Object-Oriented Design," *Information Processing Letters* **57** (No. 6, June 1996), pp. 271–75.

[Binkley and Schach, 1997] A. B. BINKLEY AND S. R. SCHACH, "Toward a Unified Approach to Object-Oriented Coupling," *Proceedings of the 35th Annual ACM Southeast Conference,* Murfreesboro, TN, April 2–4, 1997, pp. 91–97.

[Borland, 2002] BORLAND, "Press Release: Borland Unveils C++ Application Development Strategy for 2002," www.borland.com/news/press_releases/2002/01_28_02_cpp.strategy.html, January 28, 2002.

[Chidamber and Kemerer, 1994] S. R. CHIDAMBER AND C. F. KEMERER, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* **20** (June 1994), pp. 476–93.

[Clarke, Podgurski, Richardson, and Zeil, 1989] L. A. CLARKE, A. PODGURSKI, D. J. RICHARDSON, AND S. J. ZEIL, "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering* **15** (November 1989), pp. 1318–32.

[Crossman, 1982] T. D. CROSSMAN, "Inspection Teams, Are They Worth It?" *Proceedings of the Second National Symposium on EDP Quality Assurance,* Chicago, November 1982.

[Date, 1999] C. J. DATE, *An Introduction to Database Systems,* 7th ed., Addison Wesley, Reading, MA, 1999.

[Dunn, 1984] R. H. DUNN, *Software Defect Removal,* McGraw-Hill, New York, 1984.

[Elbaum, Malishevsky, and Rothermel, 2002] S. ELBAUM, A. G. MALISHEVSKY, AND G. ROTHERMEL, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Transactions on Software Engineering* **28** (February 2002), pp. 159–82.

[Endres, 1975] A. ENDRES, "An Analysis of Errors and Their Causes in System Programs," *IEEE Transactions on Software Engineering* **SE-1** (June 1975), pp. 140–49.

[Grady, 1992] R. B. GRADY, *Practical Software Metrics for Project Management and Process Improvement,* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Guimaraes, 1985] T. GUIMARAES, "A Study of Application Program Development Techniques," *Communications of the ACM* **28** (May 1985), pp. 494–99.

[Harel et al., 1990] D. HAREL, H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A. SHTULL-TRAURING, AND M. TRAKHTENBROT, "STATEMATE: A Working Environment for the

Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering* **16** (April 1990), pp. 403–14.

[Harrold, McGregor, and Fitzpatrick, 1992] M. J. HARROLD, J. D. MCGREGOR, AND K. J. FITZPATRICK, "Incremental Testing of Object-Oriented Class Structures," *Proceedings of the 14th International Conference on Software Engineering,* Melbourne, Australia, May 1992, pp. 68–80.

[Harrold and Soffa, 1991] M. J. HARROLD AND M. L. SOFFA, "Selecting and Using Data for Integration Testing," *IEEE Software* **8** (1991), pp. 58–65.

[Horgan, London, and Lyu, 1994] J. R. HORGAN, S. LONDON, AND M. R. LYU, "Achieving Software Quality with Testing Coverage Measures," *IEEE Computer* **27** (1994), pp. 60–69.

[Howden, 1987] W. E. HOWDEN, *Functional Program Testing and Analysis,* McGraw-Hill, New York, 1987.

[Hwang, 1981] S.-S. V. HWANG, "An Empirical Study in Functional Testing, Structural Testing, and Code Reading Inspection," Scholarly Paper 362, Department of Computer Science, University of Maryland, College Park, 1981.

[Jacobson, Booch, and Rumbaugh, 1999] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *The Unified Software Development Process,* Addison Wesley, Reading, MA, 1999.

[Jorgensen and Erickson, 1994] P. C. JORGENSEN AND C. ERICKSON, "Object-Oriented Integration Testing," *Communications of the ACM* **37** (September 1994), pp. 30–38.

[Kernighan and Plauger, 1974] B. W. KERNIGHAN AND P. J. PLAUGER, *The Elements of Programming Style,* McGraw-Hill, New York, 1974.

[Klepper and Bock, 1995] R. KLEPPER AND D. BOCK, "Third and Fourth Generation Productivity Differences," *Communications of the ACM* **38** (September, 1995), pp. 69–79.

[Klunder, 1988] D. KLUNDER, "Hungarian Naming Conventions," Technical Report, Microsoft Corporation, Redmond, WA, January 1988.

[Linger, 1994] R. C. LINGER, "Cleanroom Process Model," *IEEE Software* **11** (March 1994), pp. 50–58.

[Long and Morris, 1993] F. LONG AND E. MORRIS, "An Overview of PCTE: A Basis for a Portable Common Tool Environment," Technical Report CMU/SEI-93-TR-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, January 1993.

[Martin, 1985] J. MARTIN, *Fourth-Generation Languages,* Vols. 1, 2, and 3, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[McCabe, 1976] T. J. MCCABE, "A Complexity Measure," *IEEE Transactions on Software Engineering* **SE-2** (December 1976), pp. 308–20.

[McCabe and Butler, 1989] T. J. MCCABE AND C. W. BUTLER, "Design Complexity Measurement and Testing," *Communications of the ACM* **32** (December 1989), pp. 1415–25.

[McConnell, 1993] S. MCCONNELL, *Code Complete: A Practical Handbook of Software Construction,* Microsoft Press, Redmond, WA, 1993.

[Memon, Pollack, and Soffa, 2001] A. M. MEMON, M. E. POLLACK, AND M. L. SOFFA, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Transactions on Software Engineering* **27** (February 2001), pp. 144–55.

[Mills, Dyer, and Linger, 1987] H. D. MILLS, M. DYER, AND R. C. LINGER, "Cleanroom Software Engineering," *IEEE Software* **4** (September 1987), pp. 19–25.

[Musa and Everett, 1990] J. D. MUSA AND W. W. EVERETT, "Software-Reliability Engineering: Technology for the 1990s," *IEEE Software* **7** (November 1990), pp. 36–43.

[Musa, Iannino, and Okumoto, 1987] J. D. MUSA, A. IANNINO, AND K. OKUMOTO, *Software Reliability: Measurement, Prediction, Application,* McGraw-Hill, New York, 1987.

[Myers, 1976] G. J. MYERS, *Software Reliability: Principles and Practices,* Wiley-Interscience, New York, 1976.

[Myers, 1978a] G. J. MYERS, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Communications of the ACM* **21** (September 1978), pp. 760–68.

[Myers, 1979] G. J. MYERS, *The Art of Software Testing,* John Wiley and Sons, New York, 1979.

[Perry and Kaiser, 1990] D. E. PERRY AND G. E. KAISER, "Adequate Testing and Object-Oriented Programming," *Journal of Object-Oriented Programming* **2** (January/February 1990), pp. 13–19.

[Rapps and Weyuker, 1985] S. RAPPS AND E. J. WEYUKER, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering* **SE-11** (April 1985), pp. 367–75.

[Sammet, 1978] J. E. SAMMET, "The Early History of COBOL," *Proceedings of the History of Programming Languages Conference,* Los Angeles, 1978, pp. 199–276.

[Shepperd, 1988] M. SHEPPERD, "A Critique of Cyclomatic Complexity as a Software Metric," *Software Engineering Journal* **3** (March 1988), pp. 30–36.

[Shepperd and Ince, 1994] M. SHEPPERD AND D. C. INCE, "A Critique of Three Metrics," *Journal of Systems and Software* **26** (September 1994), pp. 197–210.

[Sherer, Kouchakdjian, and Arnold, 1996] S. W. SHERER, A. KOUCHAKDJIAN, AND P. G. ARNOLD, "Experience Using Cleanroom Software Engineering," *IEEE Software* **13** (May 1996), pp. 69–76.

[Stocks and Carrington, 1996] P. STOCKS AND D. CARRINGTON, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering* **22** (November 1996), pp. 777–93.

[Takahashi and Kamayachi, 1985] M. TAKAHASHI AND Y. KAMAYACHI, "An Empirical Study of a Model for Program Error Prediction," *Proceedings of the Eighth International Conference on Software Engineering,* London, 1985, pp. 330–36.

[Trammel, Binder, and Snyder, 1992] C. J. TRAMMEL, L. H. BINDER, AND C. E. SNYDER, "The Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering," *ACM Transactions on Software Engineering and Methodology* **1** (January 1992), pp. 81–94.

[Turner, 1994] C. D. TURNER, "State-Based Testing: A New Method for the Testing of Object-Oriented Programs," Ph.D. thesis, Computer Science Division, University of Durham, Durham, UK, November, 1994.

[Walsh, 1979] T. J. WALSH, "A Software Reliability Study Using a Complexity Measure," *Proceedings of the AFIPS National Computer Conference,* New York, 1979, pp. 761–68.

[Watson and McCabe, 1996] A. H. WATSON and T. J. MCCABE, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," NIST Special Publication 500-235, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 1996.

[Weyuker, 1988a] E. J. WEYUKER, "An Empirical Study of the Complexity of Data Flow Testing," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis,* Banff, Canada, July 1988, pp. 188–95.

[Weyuker, 1988b] E. WEYUKER, "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering* **14** (September 1988), pp. 1357–65.

[Wilde, Matthews, and Huitt, 1993] N. WILDE, P. MATTHEWS, AND R. HUITT, "Maintaining Object-Oriented Software," *IEEE Software* **10** (January 1993), pp. 75–80.

[Woodward, Hedley, and Hennell, 1980] M. R. WOODWARD, D. HEDLEY, AND M. A. HENNELL, "Experience with Path Analysis and Testing of Programs," *IEEE Transactions on Software Engineering* **SE-6** (May 1980), pp. 278–86.

[Yamaura, 1998] T. YAMAURA, "How to Design Practical Test Cases," *IEEE Software* **15** (November/December 1998), pp. 30–36.

[Yourdon, 1989] E. YOURDON, *Modern Structured Analysis,* Yourdon Press, Englewood Cliffs, NJ, 1989.

# 第 15 章　交付后维护

**学习目标**

学完本章之后，你应当能够：

- 完成交付后维护；
- 理解交付后维护的重要性；
- 描述交付后维护面临的挑战；
- 描述面向对象范型的维护含义；
- 描述维护所需的技巧。

一旦产品经过了验收测试，就可以移交给客户，然后安装产品并按照建造它的目的使用它。然而，任何实用的产品几乎一定要进行**交付后维护**——修正错误（纠错性维护）或者扩展产品功能（改进）。

由于产品不仅仅包括源代码，所以在产品移交客户后对文档、手册或其他任何部分的改动均属于交付后维护。有些计算机科学家不喜欢用维护，而更喜欢用**演变**这个词来说明产品随着时间的推移有了改进。事实上，有些人把软件从开始到结束的整个生命周期看作一个逐渐演变的过程。

本书的一个重要主题就是讨论软件维护的极端重要性。因此，你可能会对本章篇幅相对较短而惊讶。这样安排的原因是，产品从一开始就应该具备可维护性，并且在开发过程中的任何时候都不能遭到削弱。因此，前面所有的章节实际上已经在讨论交付后维护这个主题。本章要讨论的是如何在交付后维护阶段确保产品的可维护性。

## 15.1　为什么交付后维护是必要的

对产品进行修改有三方面的原因：

1）需要纠正错误，包括分析错误、设计缺陷、编码错误、文档错误以及其他任何错误，这称为**纠错性维护**。

2）在**完善性维护**中，修改源代码是为了提高产品的有效性。例如，客户可能希望给产品增加功能，或对产品进行修改，以使其运行速度更快。提高产品可维护性是完善性维护的又一例子。

3）在**适应性维护**中，为适应产品运行环境的变化需要对产品进行修改。例如，如果一个产品需要移植到新的编译器、操作系统或硬件平台，那么它几乎一定需要修改。例如，免税代码每改动一次，生成纳税申报单的软件就要相应做出修改。美国邮政部门于 1981 年引入 9 位邮政编码后，原来只能使用 5 位编码的产品就不得不进行修改。适应性维护并不是客户要求进行的，而是由外界给客户造成的。

## 15.2　对交付后维护程序员的要求是什么

在软件生命周期中，交付后维护工作所占的时间比其他任何阶段都多。实际上，平均来说，产品总成本中至少有 67% 的支出用于交付后维护，如图 1-3 所示。但直到今天，许多单

位仍然把交付后维护工作分配给刚刚入门或能力不强的程序员，而把产品开发中“闪光”的部分留给更加出色或更具经验的程序员。

事实上，交付后维护是软件产品开发所有工作中最困难的部分。一个主要原因是，交付后维护工作涵盖了软件开发过程所有其他阶段的各个方面。让我们想像一下一份缺陷报告送到维护程序员手上时所发生的情况吧（回想一下 1.11 节，**缺陷**是差错、故障或错误）。如果用户认为产品没有按用户手册上的说明运行，他就会提交缺陷报告。这可能由几种原因造成。首先，软件本身根本没错，只是用户误解了用户手册或者没有正确使用该产品；或者如果产品中确实有差错，也可能只是用户手册阐述不当，而代码本身没有任何错误。然而，通常情况是代码中有错误。可是在做出任何修改之前，维护程序员必须依据用户填写的缺陷报告和源代码（通常不会有其他可依据的了）来准确判断错误所在。因此，维护程序员需要有非同一般的排错能力，因为错误可能存在于产品的任何地方。而且缺陷可能是由现在还不存在的分析或设计成果带来的。

如果维护程序员已经找出了错误，那么他必须在纠正该错误的同时防止无意中在产品其他地方引入另一个错误，即**退化错误**。如果希望将退化错误减少到最低程度，就需要整个产品以及产品中每一模块的详细文档。然而，软件专业人员是以讨厌一切形式的书面工作（特别是建立文档）而著称的。文档不完整、存在错误或根本找不到等都是相当常见的情况。在这些情况下，维护程序员必须通过能够得到的惟一有效的文档——源代码——来推测避免引入退化错误所需的一切信息。

在判断出可能存在的错误并设法将其纠正后，维护程序员必须测试所做的修改能否正确运行，以及是否引入了退化错误。为了检查修改本身，维护程序员必须建造专门的测试用例；检查退化错误是通过使用为进行**退化测试**而明确存储的测试数据集来完成的（见 3.8 节）。然后，为检查修改而建造的测试用例必须加入到存储测试用例集中，以便供将来的调整后产品做退化测试用。另外，如果为纠正错误需要对分析或设计进行修改，那么这些修改也必须检查。因此，测试方面的专业知识是交付后维护工作的另一先决条件。最后，维护程序员必须为每一处修改建立文档。前面讨论的是纠错性维护。这时，维护程序员首先必须是一个出色的诊断专家，判断是否存在错误。如果存在错误，他还必须是一名熟练的纠错技师。

其他的主要维护任务是适应性维护和完善性维护。为完成这些维护任务，维护程序员必须将已存在的产品作为起始点，完成需求流、分析流、设计流和实现流。对于某些类型的修改，需要设计和实现额外的代码模块。在其他情况中，需要对已有的代码模块的设计和实现进行修改。因此，尽管说规格说明经常是由分析专家完成的，设计是由设计专家完成的，代码是由编程专家完成的，但维护程序员则需要是所有这三方面的专家。与纠错性维护一样，完善性维护和适应性维护面临的不利影响是适当文档的缺乏。而且，正如纠错性维护中一样，在完善性维护和适应性维护中，需要一种能力，即设计合适的测试用例及编写好的文档。因此，除非有最优秀的计算机专家监督维护过程，否则缺乏经验的程序员无法做好任何形式的维护工作。

从前面的讨论可以清楚地看出，维护程序员几乎必须掌握软件专业人员所应有的全部技能。但他们得到了什么回报呢？

- 无论从哪方面看，交付后维护都是一项吃力不讨好的工作。维护人员要与心存不满的用户打交道；如果用户对产品满意，就不需要维护了。

- 用户遇到的问题经常是由产品开发人员，而不是维护人员造成的。
- 代码本身可能写得不好，这加重了维护人员的挫折感。
- 许多软件开发人员看不起交付后维护，他们认为开发是一项“闪光”的工作，而维护则只是适合初级程序员或能力不强者的苦工。

可以把交付后维护看作售后服务。产品已经交给客户，但现在客户提出不满，要么是因为产品运行不正常，要么是产品不能满足客户现在所有的要求，或者产品开发时的环境现在发生了某种变化。如果软件公司不能够提供良好的维护服务，客户将来就会选择其他软件开发公司。当客户和软件开发部门同处一个单位时，双方不可避免地都要考虑将来，如果客户不满意，他会使尽一切好的或者坏的办法，损毁开发部门的信誉。这反过来又会导致开发单位信心下降，使开发单位退出开发工作，或者客户不再雇用开发单位。通过提供优秀的交付后维护服务令客户满意对每个软件单位都是重要的。因此，对于一件又一件的产品，交付后维护阶段是软件生产过程中最富挑战性的阶段，并且经常是吃力不讨好的。

这种情况怎样才能改变呢？管理者必须把交付后维护工作交给那些掌握维护所需的全部技能的程序员。管理者必须让其他人知道，只有本单位一流的专业人员才有资格做维护，同时要向维护程序员支付相应的报酬。如果管理层认为维护工作是一项挑战，良好的维护对本单位的成功至关重要，那么人们对待交付后维护工作的态度将慢慢改善（参见如下的“如果你想知道”部分）。

**如果你想知道**

在《Practical Software Maintenance》一书中，Tom Pigoski［1996］介绍了他是如何在佛罗里达州 Pensacola 建立美国海军软件维护机构的。他的想法是，如果提前告诉未来的雇员，他们的工作是担任维护程序员，他们就会对维护工作持肯定的态度。另外，他通过确保员工能够受到大量训练，并有机会在工作过程中全世界旅行，来保持他们高昂的士气。单位附近美丽的海滩以及他们使用的新办公楼在这方面也发挥了作用。

然而，维护工作开始后的 6 个月里，每名员工都在询问自己什么时候才能参加一些开发工作。看来改变人们对待维护工作的态度是极其困难的。

现在可以通过小型实例研究，对维护程序员遇到的有些问题加以强调。

## 15.3　交付后维护小型实例研究

在经济集中化的国家，政府控制着农产品的分配和交易。有这样一个国家，它的桃、苹果和梨等温带水果均由温带水果委员会（Temperate Fruit Committee，TFC）负责。有一天，TFC 主任要求一名政府计算机顾问将 TFC 的工作实现计算机管理。主任通知计算机顾问，共有 7 种温带水果——苹果、杏、樱桃、油桃、桃子和李子。数据库的设计应恰好容纳这 7 种水果，不多不少。毕竟，过去的情况就是这样的，顾问不能浪费时间和金钱考虑任何扩展性。

产品按时交给了 TFC。大约 1 年后，主任把负责该产品维护的程序员召集到一起。主任问：“你们对猕猴桃了解多少?”程序员们迷惑不解地回答说：“一无所知。”主任说：“好，看来猕猴桃是一种在我们国家刚刚开始种植的水果，TFC 会对此负责。请你们对软件做出相应修改。”

维护程序员幸运地发现，那位计算机顾问没有一字不差地按照主任原来的指示开发这个软件。考虑产品将来的扩展性这一良好习惯在计算机顾问心中根深蒂固，所以他在相关数据

库记录中预留了许多空字段。通过对数据库中的某些项目稍加重新安排，维护程序员就把第8种水果——猕猴桃——加入到产品中。

时间又过了1年，产品运行良好。后来，维护程序员又被叫到主任办公室。主任心情不错。他通知程序员，政府对农产品的分配与交易政策做了重新调整。他的委员会现在负责本国所有的水果，而不只是温带水果，所以软件必须加以修改，以容纳他交给程序员的水果清单上的另外26种水果。程序员们提出了抗议，指出这样做的工作量无异于从头重写这个软件。主任回答说："胡说！你们增加猕猴桃时不是没有问题吗？按同样的方法把这26项加进去就行了！"

从这一事例可以总结出许多重要的教训：

- 产品本身存在的、不考虑将来扩展的问题是由开发人员，而不是维护程序员造成的。开发人员在考虑软件未来扩展性方面错误地服从了主任的指示，但吃苦头的却是维护程序员。实际上，如果开发这个产品的那位计算机顾问不读一下这本书，她可能永远也意识不到她的产品根本算不上成功。维护工作中这些方面的问题更令人恼火，因为维护程序员是在负责纠正别人的错误。造成问题的人可能另有公干，或者离开了所在单位，但他们造成的后果却要由维护程序员来承担。
- 维护是困难的，在有些情况下甚至是不可能的，但客户对此并不理解。维护程序员以前可能成功地完成了完善性和适应性维护，却突然提出新任务无法完成，虽然这些任务表面上与以前毫不费力完成的任务没有什么不同。此时，这个问题就更突出了。
- 所有软件开发都必须考虑到将来的维护。如果那位计算机顾问在设计软件时考虑到可以添加任意种类的水果，那么后来增加猕猴桃和另外26种水果时就没有困难了。

正如我们多次强调的，维护是软件生产中最重要的阶段，也是最消耗资源的阶段。在产品开发过程中，重要的一点是开发人员不要忘记了维护程序员，后者将在产品安装后对产品负责。

## 15.4　交付后维护的管理

现在考虑一下有关交付后维护的管理问题。

### 15.4.1　缺陷报告

在维护产品时首先需要的是对产品进行修改的机制。对于纠错性维护，也就是在产品运行不正确而要剔除残存的错误时，用户必须提交**缺陷报告**。缺陷报告必须包括足够的信息，使维护程序员能够再现那个问题——通常是某种类型的软件故障。另外，维护程序员必须指出缺陷的严重性，典型的严重性类别包括致命的、主要的、通常的、较小的和微不足道的。

理想情况下，用户提出的每个缺陷都应立即纠正。而实际上，程序开发公司通常人力不足，开发和维护工作都会滞后。如果缺陷是致命的，比如工资发放软件在发工资的前一天或有员工增减工资的前一天崩溃了，那么必须立即采取纠正措施。其他情况下，必须立即对每一份缺陷报告进行初步的调查。

维护程序员应该首先参考缺陷报告文件。缺陷报告包括了已经发现但尚未纠正的所有缺陷，以及关于在缺陷得到纠正之前用户如何绕过它们的建议。如果缺陷以前已经有人报告过，缺陷报告中的任何信息都应传递给用户。但如果用户报告的是新缺陷，那么维护程序员应该对问题加以研究并设法找出原因和解决问题。另外，应该设法找到绕过问题的办法，因

为开发单位有可能需要 6～9 个月的时间才能分配人力对软件做出必要的修改。考虑到程序员，特别是能够胜任维护工作的优秀程序员的短缺，对于那些不十分紧急的缺陷报告，只能建议用户通过某种方法继续使用带有缺陷的软件，直到缺陷可以得到解决。

然后，维护程序员的结论要连同所有支持其结论的文档——用以得出上述结论的清单、设计、手册等——一同加入缺陷报告文件中。负责交付后维护的管理员应该定期阅读该报告，确定各种纠错任务的优先次序。该文件还应包括客户在完善性维护和适应性维护等方面的要求。下一次将纠正优先级最高的缺陷。

如果有若干套产品发售到许多单位，那么必须向该产品的所有用户递交缺陷报告，以及纠正这些缺陷的预期日期。然后，如果相同的问题出现在另一个单位，用户可以参考相关的缺陷报告来确定是否可能绕过缺陷以及何时能纠正该缺陷。当然，最理想的是立即纠正所有错误并向所有用户单位发送产品的新版本。鉴于目前世界范围内优秀程序员的短缺，以及交付后软件维护的现实情况，发布缺陷报告也许是能够采取的最佳措施。

错误通常不能立即纠正还有另外一个原因。同时做出大量修改，对全部修改同时进行测试，改编文档并安装产品的新版本，比单独纠正每个错误、进行测试、归档并安装新产品，然后再对下一处错误重复整个周期更经济。当新版本必须安装在大量计算机上（比如在客户/服务器模式网络上的大量客户端）或软件运行在不同单位时，情况尤其如此。结果，开发单位更乐于将非关键性的维护任务积累起来，批量修改。

### 15.4.2　批准对产品的修改

一旦决定进行纠错性维护，维护程序员就查找软件运行失败的原因，并承担起修正该错误的任务。代码改变后，必须像对整个产品进行测试一样，对所做修改进行测试（退化测试）。然后必须更新文档，以反映出所做的修改。特别是对改变后的模块，要在其序言注释中加入关于进行了哪些修改、为什么修改、由谁做的修改，以及何时进行修改等方面的信息（见图 14-1）。如果有必要，分析或设计模块也需要修改。在完善性维护或适应性维护之后，也要采取类似的步骤。惟一的区别是，完善性维护和适应性维护是应客户要求进行的，不是由缺陷报告引发的。

接下来看起来要做的是把新版本发布给用户。但如果维护程序员对所做的修改测试不充分该怎么办呢？产品在发布前，要通过一个独立的小组进行软件质量保证，即维护 SQA 小组的成员一定不能作为维护程序员给相同的管理者提供报告。SQA 保持管理上的独立很重要（见 6.1.2 节）。

前面讨论了为什么交付后维护工作是困难的。同理，维护工作也是容易出错的。交付后维护阶段的测试是困难的，也是消耗时间的，SQA 小组不应低估测试对软件维护的影响。一旦新版本得到 SQA 小组的批准之后，它就可以发布了。

管理层需保证工作程序得到严格遵守的另外一种情况是在使用软件基线技术或私人拷贝的时候（5.8.2 节）。假设一名程序员想修改 **TaxProvision** 类，他拷贝了 **TaxProvision** 类和维护工作所需的其他所有类，通常包括产品中的所有其他类。程序员对 **TaxProvision** 做了必要的修改并进行了测试。现在，**TaxProvision** 的前一个版本已被冻结，修改后的 **TaxProvision** 安装在软件基线上。但是，当修改后的产品交到用户手上后却立刻崩溃了。问题出在维护程序员对修改后的 **TaxProvision** 类进行测试时使用的是私人工作平台的拷贝，也就

是刚开始对 **TaxProvision** 进行维护的时候软件基线上的其他类。但与此同时，某个其他类被开发同一产品的其他程序员更新了。这个教训是明显的，在安装某一模块之前，必须利用所有其他模块的当前软件基线对该模块进行测试，而不能用程序员的私人版本进行测试。这也是建立独立的 SQA 的另一个原因，SQA 小组的成员根本看不到程序员的私人工作平台。第三个原因是，对某个错误的初次纠正有 70% 是不正确的 [Parnas，1999]。

### 15.4.3　确保可维护性

交付后维护不是一劳永逸的。一件出色的产品在其生命周期中要经过一系列修改，所以，在整个软件生产过程中都必须考虑交付后维护问题。比如在设计阶段，应采用信息隐藏技术（见 7.6 节）；在实现阶段，变量名的选择要让将来的维护程序员容易看懂（见 14.3 节）。文档要完整、正确，并能够反映出产品每一模块的当前版本。

在交付后维护阶段，重要的一点是不要使从一开始就建立起来的软件的可维护性遭到削弱。换句话说，正如软件开发人员应该始终明白软件将来需要维护一样，维护程序员也应始终清楚软件将来同样不可避免地需要维护。开发阶段确立的保证维护性原则同样适用于交付后维护阶段。

### 15.4.4　反复维护造成的问题

产品开发中一个更令人恼火的问题是**移动目标问题**（见 2.4 节）。客户改变需求的速度与开发人员完成产品的速度一样快，这个问题不仅困扰开发小组，频繁的变化还造成产品的结构性差异，另外，这样的变化也增加了产品的成本。

在交付后维护阶段，这个问题会恶化。完成的产品修改得越多，它就越偏离原来的设计，将来的修改也就越困难。经过反复维护后，文档可能会比通常更不可靠，退化测试文件也可能得不到更新。如果再做更多的维护，整个产品可能就需要全部重写。

移动目标问题明显是个管理问题。理论上讲，如果管理部门对待用户态度坚决，并在项目的开始就讲明问题，那么从签署规格说明起，需求就可以冻结，直到产品交付。还有，在提出每次完善性维护需求后，需求也可冻结 3 个月或 1 年。而实际上，这种方法是行不通的。例如，客户恰好是某公司的总裁，而开发部门恰好是该公司的软件分部，那么总裁实际上可以时常要求对软件进行修改，并且落实。那句古老的谚语“出钱的人说了算”形容这种情况很合适。也许软件副总裁能做的最好的事情就是设法向总裁解释反复维护对产品的影响，然后在进一步的修改会对产品的完整性造成损害时干脆重写整个产品。

通过拖延修改的时间表示反对是行不通的，这只能使开发人员被那些乐于以更快的速度完成任务的人替换掉。简言之，如果要求反复修改的人有足够的权力，移动目标问题是无法解决的。

## 15.5　面向对象软件的维护

采用面向对象范型的原因之一是它提高了可维护性。毕竟对象是独立的程序单元。更明确地说，设计良好的对象表现了概念上的独立性，也称作**封装**（7.4 节）。产品中由对象模拟的现实世界相关的部分均被定位在对象本身。另外，对象也表现出物理的独立性，信息隐藏技术可以保证对象的实现细节在对象外不可见（7.6 节）。与对象通信的惟一方式是向对

象发送消息调用某一具体方法。

两方面的原因决定了维护对象是容易的。首先，概念独立性意味着容易判断出产品的哪一部分必须进行修改，以达到某一明确的维护目标——无论是增强性维护还是纠错性维护。其次，信息隐藏确保了对象本身的修改不会在该对象以外产生影响，因此可以大大降低退化错误的数量。

然而，实际情况并不像这样简单和美好。实际上，有三个问题是专门针对维护面向对象软件的。其中一个问题可以通过使用 CASE 工具来解决，而其他问题则不容易处理：

1）考虑图 15-1 所示的 C++ 类层次结构。方法 displayNode 是在 **UndirectedTree** 类中定义的，由 **DirectedTree** 类继承，然后在 **RootedTree** 类中重新定义。这个重定义的版本由 **BinaryTree** 类和 **BalancedBinaryTree** 类继承，并在 **BalancedBinaryTree** 类中使用。因此，维护程序员必须研究整个继承结构才能理解 **BalancedBinaryTree** 类。更糟的是，继承结构通常不是图 15-1 所展示的那种线型结构，而是涵盖了整个产品，所以，为了理解 displayNode 方法在 **BalancedBinaryTree** 类里的行为，维护程序员必须仔细阅读产品的大部分代码。这与本节一开始描述的对象“独立性”相去甚远。这个问题的解决方案很简单：使用合适的 CASE 工具。正如 C++ 编译器可以在 **BalancedBinaryTree** 类实例的内部准确判断 displayNode 方法的版本那样，维护程序员使用的工作平台可以提供一个类的“展开版本”，即类的定义要明确给出该类直接或间接继承的全部特征，包括任何重命名或重定义。图 15-1 中 **BalancedBinaryTree** 类的展开形式包括 **RootedTree** 类中对 displayNode 方法的定义。

```
class UndirectedTree
{
  ...
  void displayNode (Node a);
  ...
}// class UndirectedTree

class DirectedTree : public UndirectedTree
{
  ...
}// class DirectedTree

class RootedTree : public DirectedTree
{
  ...
  void displayNode (Node a);
  ...
}// class RootedTree

class BinaryTree : public RootedTree
{
  ...
}// class BinaryTree

class BalancedBinaryTree : public BinaryTree
{
  Node        hhh;
  displayNode (hhh);
}// class BalancedBinaryTree
```

图 15-1　类层次结构的 C++ 实现

2）对使用面向对象语言实现的产品进行维护时遇到的第二个问题不容易解决。这是由7.8节介绍的多态性和动态绑定这两个概念造成的。在那一节中我们给出了一个例子：1个基类名为 **FileClass**，带有3个子类 **DiskFileClass**、**TapeFileClass** 和 **Diskette FileClass**。图7.31显示了这几个类，这里为了阅读方便我们把它复制到图15-2。在基类 **FileClass** 里，声明了一个哑（**abstract** 或 **virtual**）方法 open。然后3个子类分别实现了该方法，每个方法的名称都是相同的 open，见图15-2。假设 myFile 作为 **FileClass** 类的实例被声明为一个对象，那么要维护的代码中就包括了 myFile.open（）方法。由于多态性和动态绑定的原因，myFile 在运行时可能是 **FileClass** 类的3个子类（硬盘文件、磁带文件或软盘文件）中任何一个类的成员。一旦运行时系统确定了 myFile 所属的类，就会调用相应的 open 方法。这样的结果是不利于维护工作的。如果维护程序员在代码中遇到了 myFile.open（）调用，为了理解这部分产品，他必须分别考虑 myFile 是3个子类中任意一个类的实例时所发生的情况。CASE 工具在这种情况下也无能为力，因为一般无法用静态工具解决动态绑定问题。判断大量绑定中究竟哪一个会发生，惟一的方法是对代码进行跟踪，要么在计算机上运行代码，要么手工跟踪。多态性和动态绑定实际上是面向对象技术中非常强大的技术，它利于面向对象产品的开发。然而，它们对维护工作却是有害的。维护程序员不得不研究运行时可能发生的各种绑定，然后在大量方法中判断出哪一个方法会在代码的这一点被调用。

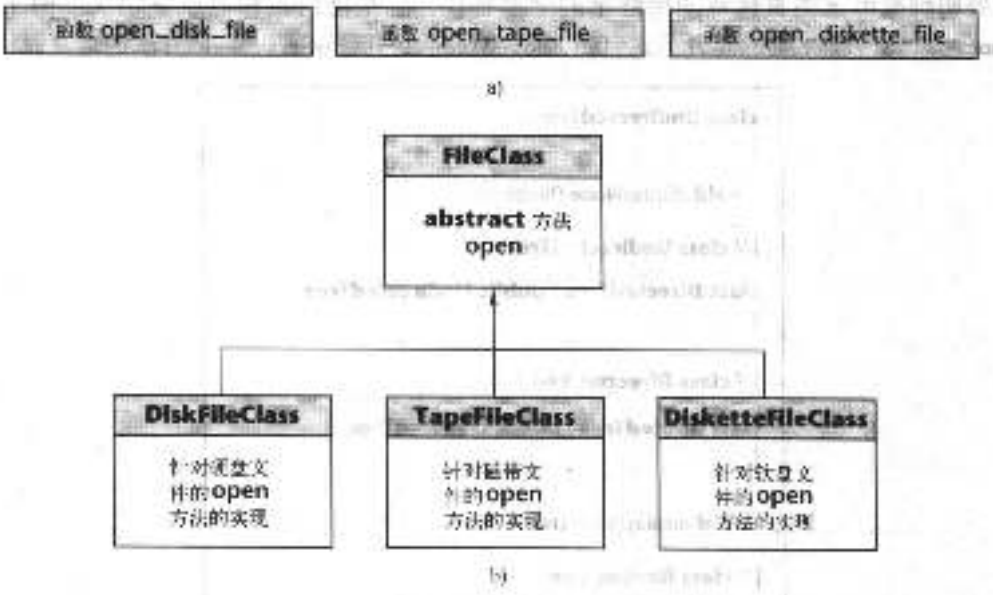



图15-2　基类 FileClass 及其3个子类 DiskFileClass、TapeFileClass、DisketteFileClass 的定义

3）最后一个问题是由**继承**造成的。假设某一基类满足了一件新产品设计中多数但不是全部要求。现在定义一个子类，这个子类在许多方面与基类相同，但可能加入了新特征，原有的特征可能被重命名、重新实现、取消或做出其他改动。另外，这些修改可能不会对基类或该基类的其他子类产生影响。然而，假设现在基类本身改变了。如果真是这样，所有子类均会产生相同的变化。换句话说，继承的优势在于可以在不改变继承树（如果实现时使用的是C++这种支持多继承的语言，那么也可以是继承图）中其他类的情况下，给继承树（继

承图）加入新叶子。但是，如果继承树的内部节点发生了某种变化，那么这种变化将传递给它的子节点（**脆弱基类问题**）。

这样，继承就成为面向对象技术的另一特征，它对开发有重大的积极影响，但对维护却存在负面影响。

## 15.6　交付后维护技能与开发技能

本章前面大部分讨论了维护工作所需的技能。

- 对于纠错性维护，判断造成大型产品运行故障的原因被认为是关键技能。但这一技能并不是仅仅在产品交付后维护中才需要，它在集成与产品测试中同样需要。
- 另外一项重要的技能是，在没有充足文档的情况下有效工作的能力。再次说明，在产品处于集成和测试阶段时，文档很少是完整的。
- 另外要强调的是，与分析、设计、实现和测试有关的技能对于适应性维护和完善性维护是必需的。这些活动在开发阶段也要进行，而其中每一项活动都需要特别的技能。

换句话说，交付后维护程序员所需的技能与软件开发其他阶段的专业人员所需掌握的技能是没有任何区别的。关键的一点是，维护程序员不能仅掌握各个领域的技能，而是要高度熟练地掌握所有那些领域的技能。虽然一般的软件开发人员可以专攻软件开发中的某一领域，如设计或测试，但软件维护程序员必须是软件生产中每个领域的专家。毕竟，维护与开发一样重要，并且是有过之而无不及。

## 15.7　逆向工程

正如已经指出的那样，有时候交付后维护所能依据的惟一的文档就是源代码本身。（这在维护那些**遗留系统**时经常发生；遗留系统指至少是 15 年或 20 年前开发的、目前仍在使用的软件。）这种情况下，代码的维护极其困难。处理这类问题的一种方法是从源代码入手，设法重新设计文档甚至产品规格说明。这一过程称作**逆向工程**。

CASE 工具可以协助完成这一过程。最简单的一个例子就是小型打印机（5.6 节），它可以帮助维护程序员更清楚地浏览代码。其他工具可以用来直接从源代码给出流程图或 UML 图，这些可视化工具可以辅助恢复产品的设计方案。

一旦维护小组完成对设计的重构后，有两种方案可以选择。一是重新制定产品规格说明，对重新制定的规格说明加以修改，以反映出必要的变化，并按通常的方法重新实现该新产品。（在逆向工程领域，通常的开发过程即从产品规格说明到设计再到编码的过程，这称为**正向工程**。正向工程之后的逆向工程有时称作**再工程**。）实际上重新制定产品规格说明是件极其困难的事情。更普遍的情况是，重新制定的设计方案要经过修改，然后利用修改后的设计方案实施正向工程。

维护阶段经常要进行的一项相关活动是“重构”。逆向工程使产品从低层次抽象发展到高层次抽象，例如，从代码到设计。正向工程是由高层次抽象到低层次抽象。然而，**重构**是在同一个抽象层次上进行的。它是在不改变产品功能的前提下，对产品加以改进的过程。实现精美的打印是重构的一种形式，将代码由非结构化转化为结构化也是一种重构。总的来讲，重构是使源代码（或设计方案，甚至数据库）更容易维护。当使用了像极限编程（见 2.9.4 节）这样灵活的过程时，设计调整被称为**重分解**，这是重构的另一个例子。

如果源代码丢失了，只剩下一份可执行程序，情况就更糟了。乍看起来，惟一可能重建源代码的方法就是利用反汇编程序得到汇编代码，然后建立一种工具（可以称作反编译器）设法重建产品的高级语言代码。这种方法会带来大量难以解决的问题：

- 经过原来的编译，变量名丢失了。
- 许多编译器会以某种方式对代码进行优化，使重建源代码变得极其困难。
- 汇编程序中的一个结构（如循环）可能与源代码中许多不同的结构相对应。

因此，现有产品成了一个黑盒子，需要采用逆向工程的方法，依据现有产品的行为来推测产品原来的规格说明。重构的规格说明要根据需要进行修改，并以这些规格说明为基础，通过正向工程开发产品的新版本。

## 15.8　交付后维护阶段的测试

在产品开发阶段，许多开发人员对整个产品有很好的宏观了解。但由于计算机行业人员流动迅速，维护程序员不大可能是原来参加过开发的人员。因此维护程序员会把产品看成是一系列松散耦合的模块，他们不清楚一个模块的改变是否会严重影响一个或更多的其他模块乃至整个产品。即使维护程序员希望理解产品的方方面面，但修正和扩展产品的压力也使他们没有时间进行仔细研究。另外，在许多情况下，很少或根本没有文档来帮助他们详细理解产品。降低这种困难的一种方法是采用退化测试，即用以前的测试用例对改变的部分进行测试，确保产品正常运转。

由于这个原因，以机器可以识别的形式保留所有测试用例及预期的测试结果是十分重要的。由于产品的变化，有些保留的测试用例必须加以修正。例如，如果由于税法的修订引起征收所得税的变化，那么与所得税有关的、用于测试工资发放软件的测试用例也要改变。同样，如果通过卫星观察数据发现需要对某一个岛的经纬度进行修正，那么通过该岛坐标计算飞机位置的软件，输出结果也应进行相应调整。由于维护内容的不同，有些过去有效的测试用例将变得无效。实际上，更正保留测试用例时所需的计算与为验证维护正确性而建立新测试用例所需的计算是相同的。因此，维护测试用例文件及其预期结果不需要做其他工作。

有人认为退化测试是在浪费时间，因为退化测试要求利用大量测试用例对整个产品进行重新测试，而测试用例中的大多数，表面上都与在产品维护过程中修改过的模块无关。这种说法是值得商榷的。在前面那句话里，“表面上”这个词很关键。维护工作中存在着意识不到的副作用（即引入退化错误），这就使上面的论点难以成立。退化测试是各种维护中不可缺少的一个方面。

## 15.9　交付后维护的 CASE 工具

我们没有理由指望维护程序员手工跟踪各种版本号，并在模块每次更新后为其指定下一个版本号。除非操作系统包含版本控制功能，否则就需要一个版本控制工具，如 UNIX 工具中的 sccs（源代码控制系统）[Rochkind，1975] 和 rcs（版本控制系统）[Tichy，1985]，或开放源代码的产品 CVS（并行版本系统）[Loukides and Oram，1997]。同样也不能指望手工控制第 5 章讨论的冻结技术，以及其他确保版本号能够相应更新的方法。这就需要一种配置控制工具。一种典型的商业软件是 CCC（修改和配置控制）。即使软件公司不愿意购买一整套配置控制工具，也应至少连同版本控制工具购买一套建造工具。在交付后维护阶段必需的

另一类 CASE 工具是缺陷跟踪工具——用于记录已经报告但尚未纠正的缺陷。

15.7 节描述了一些有助于逆向工程和再工程的 CASE 工具。这类工具能够以可视化方式显示产品结构，如 Rational Rose 和 Together。Doxygen 是开放源代码的这类工具。

缺陷跟踪是交付后维护的一个重要方面，确定每个已报告的缺陷的当前状态很重要。Rational ClearQuest 是商用的**缺陷跟踪工具**，Bugzilla 是时下流行的开放源代码的工具，这样的工具可用于记录缺陷的严重程度（15.4.1 节）和它的状态（基本上不管该缺陷是否被修复）。另外，一些缺陷跟踪工具可以链接配置管理工具和缺陷报告，这样当建立了新版本后，维护程序员可以选择包括在新版本中特定的缺陷修复报告。

交付后维护是困难和恼人的，管理部门至少要给维护小组提供必需的工具，以保证产品维护的效率和效力。

## 15.10　交付后维护的度量

交付后维护阶段的活动基本上是分析、设计、实现、测试及修订文档。因此，用于评价这些活动的度量同样适用于维护阶段。例如，14.13.2 节讨论的复杂性度量同样适用于交付后维护，因为高度复杂的代码模块有可能引入退化错误。在修改这类代码模块时应特别注意。

另外，专门适用于交付后维护的度量包括了与软件缺陷报告（如所报告缺陷的总数以及按严重程度和类型划分的缺陷）相关的各种度量。另外，还需要与缺陷报告当前状态相关的信息。例如，在 2005 年报告并纠正了 13 个关键缺陷，与在同一年只报告了 2 个关键缺陷但一个也未纠正这两种情况之间有着天壤之别。

## 15.11　交付后维护：Osbert Oglesby 实例研究

在 Osbert Oglesby 实例研究的源代码中已查到一些错误，另外，必须进行完善性维护。这 6 项维护任务留做练习（习题 15.11 到习题 15.16）。

## 15.12　交付后维护阶段面临的挑战

本章描述了交付后维护阶段面临的许多挑战，最困难的挑战之一是，维护通常比开发更难，然而维护程序员又通常被开发程序员看低，还经常比开发程序员收入低。

## 本章小结

交付后维护是一项重要并且颇具挑战性的软件活动（15.1 节及 15.2 节）。这一点通过 15.3 节的小型实例研究可以看得出来。本章还讨论了与交付后维护管理有关的问题（见 15.4 节），包括反复维护的问题（15.4.4 节）。15.5 节讨论了面向对象软件的交付后维护问题。维护程序员所需的技能与开发人员是相同的，二者的差别在于开发人员可以专注于软件开发过程的某一环节，而维护程序员必须熟练掌握软件生产过程的所有方面（见 15.6 节）。15.7 节描述了逆向工程。接下来我们讨论了交付后维护阶段的测试问题（见 15.8 节）及交付后维护中使用的 CASE 工具（见 15.9 节）。15.10 节讨论了交付后维护的度量。15.11 节讨论的 Osbert Oglesby 实例研究的交付后维护留作练习。在本章的结束讨论了交付后维护面临的挑战（见 15.12 节）。

## 进一步阅读

与交付后维护有关的传统信息来源是［Lientz，Swanson，and Tompkins，1978］，尽管一些结果现在有疑问（参见第 1 章 1.3 节下的第一个“如果你想知道”部分）。在［Harrold，Rosenblum，Rothermel and Weyuker，2001］中讨论了退化测试用例选择，设置退化测试用例的优先级在［Rothermel，Untch，Chu，and Harrold，2001］中有讨论。［Onoma，Tsai，Poonawala，and Suganuma，1998］中讨论了工业环境下的退化测试。

［Sneed，1995］讨论了规划再工程，收录在《IEEE Software》杂志 1995 年 1 月刊的一些有关再工程的文章中。［Adolph，1996］讨论了再工程的成本和收益。［Charette，Adams，and White，1997］描述了交付后维护框架内的风险管理。［von Mayrhauser and Vans，1996］讨论了大规模软件产品交付后维护的几个不同的程序理解机制。

［Teng，Jeond，and Grover，1998］给出了几个成功的再工程项目的分析。［Bisbal，Lawless，Wu，and Grimson，1999］描述了遗留软件产品的维护。［Fioravanti and Nesi，2001］给出了模拟适应性维护工作量的度量。［Rajlich，Wilde，Buckellew，and Page，2001］中讨论了遗留系统的理解问题。再工程领域内可跟踪性的重要性是［Ebner and Kaindl，2002］的主题。［Bandi，Vaishnavi，and Turk，2003］中讨论了可维护性领域内度量的应用。

［Henry and Humphrey，1990］和［Mancl and Havanas，1990］描述了对象的使用对交付后维护的影响。［Lejter，Meyers，and Reiss，1992］和［Wilde，Matthews，and Huitt，1993］对具体面向对象产品的交付后维护进行了讨论。［Briand，Bunse，and Daly，2001］讨论了面向对象设计的可维护性。评估设计模式文档对交付后维护的影响的相关内容在［Prechelt，Unger-Lamprecht，Philippsen，and Tichy，2002］中有描述。

《Communications of the ACM》的 1994 年 5 月刊有关于软件维护方面的文章。《IEEE Software》1998 年 7、8 月刊有几篇遗留系统方面的文章，特别是［Rugaber and White，1998］。软件维护年会的会议录有大量的各种各样与维护有关的材料。

## 习题

15.1　为什么人们经常错误地认为软件维护比不上软件开发？

15.2　假设有一个产品的作用是判断一台计算机是否感染了病毒。阐述为什么这类产品的许多代码模块可能有多个变种。这种情况对交付后维护有哪些影响？由此出现的问题如何解决？

15.3　针对习题 8.7 中的图书自动循环系统，重新回答习题 15.2。

15.4　针对习题 8.8 中提到的用于检查银行报告书是否正确的软件，重新回答习题 15.2。

15.5　针对习题 8.9 中提到的自动柜员机重新回答习题 15.2。

15.6　假设你是一家大型软件公司负责交付后维护工作的经理。在雇用新员工的时候，你希望他具备哪些方面的素质？

15.7　交付后维护工作对于由 1 个人组成的软件产品公司有哪些意义？

15.8　如果要求你建立一份计算机化的缺陷报告文件，你准备在文件中保存哪些数据？你的工具能够满足哪些方面的查询？不能满足哪些方面的查询？

15.9　假设你收到一份来自 Ye Olde Fashioned 软件公司（习题 14.24）的备忘录，指出在可

预见的将来，Ye Olde Fashioned 公司将需要维护数千万行 COBOL 程序代码，他们向你咨询使用哪些 CASE 工具进行这些维护。你该如何回答？

15.10 （学期项目）Ophelia 的 Oasis 需要加入另外 12 个房间，习题 14.30 的产品将以什么方式进行修改？放弃现有的，从头开始是否更好？把你的答案与习题 1.11 的解答进行比较。

15.11 （实例研究）对 14.8 节中产品的实现加以纠正，使该实现能够避免用户输入负的购买价格、售出价格或流行度系数。

15.12 （实例研究）在当前的实现中，每类报表的默认行为是报表中的每页显示三幅画作。修改 14.8 节的实现，使 Osbert 能够设置显示在报表页的画作数量。

15.13 在当前的实现中，用户正确地要求提供画作的长和宽（以厘米为单位），假设你希望将你的软件出售给美国的艺术商，他更习惯于以英寸为单位，而不是以厘米为单位。正确地修改 14.8 节中的实现。

15.14 （实例研究）对于 14.8 节中产品的实现中的所有报表，通过调整各种控件的对齐方式改进其外观。

15.15 （实例研究）在 14.8 节的实现中用图形用户界面取代菜单驱动的输入例程。

15.16 （实例研究）在当前的实现中，画作标题最后的符号“?”表明关于标题还有不确定的内容。纠正 14.8 节的实现，使它能够处理画作标题中的问题。（如果你想知道为什么需要修改，请参见 10.9 节中的“如果你想知道”部分。）

15.17 （软件工程读物）你的导师将发给你［Harrold，Rosenblum，Rothermel and Weyuker，2001］的复印件。你对 Rosenblum-Weyuker 预测器有多大的信心？

## 参考文献


[Adolph, 1996] W. S. ADOLPH, “Cash Cow in the Tar Pit: Reengineering a Legacy System,” *IEEE Software* **13** (May 1996), pp. 41–47.

[Bandi, Vaishnavi, and Turk, 2003] R. K. BANDI, V. K. VAISHNAVI, AND D. E. TURK, “Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics,” *IEEE Transactions on Software Engineering* **29** (January 2003), pp. 77–87.

[Bisbal, Lawless, Wu, and Grimson, 1999] J. BISBAL, D. LAWLESS, B. WU, J. GRIMSON, “Legacy Software Products: Issues and Directions,” *IEEE Software* **16** (September/October 1999), pp. 103–11.

[Briand, Bunse, and Daly, 2001] L. C. BRIAND, C. BUNSE, AND J. W. DALY, “A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs,” *IEEE Transactions on Software Engineering* **27** (June 2001), pp. 513–30.

[Charette, Adams, and White, 1997] R. N. CHARETTE, K. M. ADAMS, AND M. B. WHITE, “Managing Risk in Software Maintenance,” *IEEE Software* **14** (May/June 1997), pp. 43–50.

[Ebner and Kaindl, 2002] G. EBNER AND H. KAINDL, “Tracing All Around in Reengineering,” *IEEE Software* **19** (May/June 2002), pp. 70–77.

[Fioravanti and Nesi, 2001] F. FIORAVANTI AND P. NESI, “Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems,” *IEEE Transactions on Software Engineering* **27** (December 2001), pp. 1062–84.

[Harrold, Rosenblum, Rothermel, and Weyuker, 2001] M. J. HARROLD, D. ROSENBLUM, G. ROTHERMEL, AND E. WEYUKER, “Empirical Studies of a Prediction Model for Regression Test Selection,” *IEEE Transactions on Software Engineering* **27** (March 2001), pp. 248–63.

[Henry and Humphrey, 1990] S. M. HENRY AND M. HUMPHREY, "A Controlled Experiment to Evaluate Maintainability of Object-Oriented Software," *Proceedings of the IEEE Conference on Software Maintenance*, San Diego, CA, November 1990, pp. 258–65.

[Lejter, Meyers, and Reiss, 1992] M. LEJTER, S. MEYERS, AND S. P. REISS, "Support for Maintaining Object-Oriented Programs," *IEEE Transactions on Software Engineering* **18** (December 1992), pp. 1045–52.

[Lientz, Swanson, and Tompkins, 1978] B. P. LIENTZ, E. B. SWANSON, AND G. E. TOMPKINS, "Characteristics of Application Software Maintenance," *Communications of the ACM* **21** (June 1978), pp. 466–71.

[Loukides and Oram, 1997] M. K. LOUKIDES AND A. ORAM, *Programming with GNU Software*, O'Reilly and Associates, Sebastopol, CA, 1997.

[Mancl and Havanas, 1990] D. MANCL AND W. HAVANAS, "A Study of the Impact of C++ on Software Maintenance," *Proceedings of the IEEE Conference on Software Maintenance*, San Diego, CA, November 1990, pp. 63–69.

[Onoma, Tsai, Poonawala, and Suganuma, 1998] A. K. ONOMA, W.-T. TSAI, M. H. POONAWALA, AND H. SUGANUMA, "Regression Testing in an Industrial Environment," *Communications of the ACM* **42** (May 1998), pp. 81–86.

[Parnas, 1999] D. L. PARNAS, "Ten Myths about Y2K Inspections," *Communications of the ACM* **42** (May 1999), p. 128.

[Pigoski, 1996] T. M. PIGOSKI, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, John Wiley and Sons, New York, 1996.

[Prechelt, Unger-Lamprecht, Philippsen, and Tichy, 2002] L. PRECHELT, B. UNGER-LAMPRECHT, M. PHILIPPSEN, AND W. F. TICHY, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance," *IEEE Transactions on Software Engineering* **28** (June 2002), pp. 595–606.

[Rajlich, Wilde, Buckellew, and Page, 2001] V. RAJLICH, N. WILDE, M. BUCKELLEW, AND H. PAGE, "Software Cultures and Evolution," *IEEE Computer* **34** (September 2001), pp. 24–28.

[Rochkind, 1975] M. J. ROCHKIND, "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1** (October 1975), pp. 255–65.

[Rothermel, Untch, Chu, and Harrold, 2001] G. ROTHERMEL, R. H. UNTCH, C. CHU, AND M. J. HARROLD, "Prioritizing Test Cases for Regression Test Cases," *IEEE Transactions on Software Engineering* **27** (October 2001), pp. 929–48.

[Rugaber and White, 1998] S. RUGABER AND J. WHITE, "Restoring a Legacy: Lessons Learned," *IEEE Software* **15** (July/August 1998), pp. 28–33.

[Sneed, 1995] H. M. SNEED, "Planning the Reengineering of Legacy Systems," *IEEE Software* **12** (January 1995), pp. 24–34.

[Teng, Jeong, and Grover, 1998] J. T. C. TENG, S. R. JEONG, AND V. GROVER, "Profiling Successful Reengineering Projects," *Communications of the ACM* **41** (June 1998), pp. 96–102.

[Tichy, 1985] W. F. TICHY, "RCS—A System for Version Control," *Software—Practice and Experience* **15** (July 1985), pp. 637–54.

[von Mayrhauser and Vans, 1996] A. VON MAYRHAUSER AND A. M. VANS, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Transactions on Software Engineering* **22** (June 1996), pp. 424–37.

[Wilde, Matthews, and Huitt, 1993] N. WILDE, P. MATTHEWS, AND R. HUITT, "Maintaining Object-Oriented Software," *IEEE Software* **10** (January 1993), pp. 75–80.

# 第 16 章　UML 的进一步讨论

**学习目标**

学完本章之后，你应当能够：

- 使用 UML 用例、类图、注解、用例图、交互图、状态图、活动图、包、组件图以及配置图来建模软件；
- 理解 UML 是一种语言，而不是一种方法。

在本书前面，已经提出了 UML 的各种要素［Booch，Rumbaugh，and Jacobson，1999］。特别是，在第 7 章中已经介绍了表示继承、聚合及关联等符号；在第 10 章中，介绍了用例、用例图以及注解；在第 12 章，增加了类图、状态图、协作图以及序列图。

这个 UML 子集对于理解本书和做全部习题以及附录 A 的学期项目是足够了。然而，遗憾的是，现实世界的软件产品比 Osbert Oglesby 实例研究或附录 A 的学期项目更大也更复杂。因此，在本章中给出了更多有关 UML 的材料，作为进入软件行业的准备。

在阅读本章之前，必须要认识到 UML 像其他所有最新的计算机语言一样，是不断变化的。在写这本书的时候，UML 的最新版本是 1.4 版。然而在本书出版的时候，UML 的有些方面已经变化了。像 3.1 节中的“如果你想知道”部分中所解释的那样，UML 现在已经在对象管理小组（Object Management Group）的控制之下。在往下进行之前，最好到 OMG 网站 www.omg.org 检查一下对 UML 的更新。

## 16.1　UML 不是一种方法

在详细察看 UML 之前，最基本的是要澄清 UML 是什么，以及更重要的是，UML 不是什么？UML 是 Unified Modeling Language（统一建模语言）的缩写，也就是说，UML 是一种语言。让我们来看一下像英语这样的语言，英语可以用来写小说、百科全书、诗、祷词、新闻报导，甚至是关于软件工程的课本。就是说，语言只是表达思想的工具。一种特定的语言不会限制该语言可以描述的思想的类别，或者它们被描述的方式。

作为一种语言，UML 能够用来描述使用传统范型或任何版本的许多面向对象范型开发的软件，包括统一过程（Unified Process）。换个说法，UML 是一种符号，不是一个方法。它是一种可以与任何其他方法结合使用的符号。

事实上，UML 不仅是一般的符号，它就是我们所需要的符号。很难想像，一本现代的关于软件工程的书不使用 UML 来描述软件。UML 已经成为一个世界标准，它是如此的普及，以至于不熟悉 UML 的软件专业人员将难于履行职责。

本章的标题是“UML 的进一步讨论”。记住 UML 所起的核心作用，这里给出的所有 UML 内容看起来都是最基本的。然而，UML 版本 1.4 的用户手册将近有 600 页长，因此，对其内容全部覆盖似乎不是一个好主意。但是，不知道 UML 的每个方面能成为一个熟练的软件专业人员吗？

关键是 UML 是一种语言。英语有超过 100 000 个单词，但是，几乎每个讲英语的人只用完整英语词汇的一个子集就能对英语运用自如。同样，在这一章中描述了全部的 UML 图

表，同时还介绍了每个图表的许多（但不是全部）选项。在第 7、10、12 和 13 章中给出的 UML 的较小子集对于学习本书是足够了，同样，本章中给出的 UML 的较大子集对于开发和维护多数软件产品也是足够了。

## 16.2　类图

最简单的可能的类图见图 16-1。它描述了 Bank Account class 类。更多有关 Bank Account Class 类的细节见图 16-2 中的类图。UML 的一个关键方面是图 16-1 和图 16-2 都是有效的类图。换句话说，如果认为对于当前的迭代和递增合适，可以向 UML 中加入或多或少的细节。

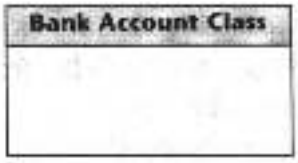



图 16-1　最简单的可能的类图

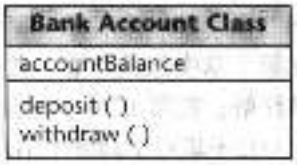



图 16-2　在图 16-1 的类图中加入一个属性和两个操作

将这个表示扩展到对象，bank account 可以非正式地用于这个类的某一个特定的对象。完整的 UML 注释是：

bank account：Bank Account Class

即，bank account 是一个对象，是类 Bank Account Class 的一个实例。详细地说，用下划线表示一个对象，冒号表示“一个实例”，Bank Account Class 中的黑体字和首字母大写表示这是一个类。然而，当不会产生混淆时，UML 允许我们使用一个较短的表示：bank account。

现在假定我们要建模任意银行账户的概念，即我们不希望指定 Bank Account Class 的某一个特定对象。对此的 UML 表示是：

：Bank Account Class

如刚刚指出的，冒号表示“一个实例”，因此：Bank Account Class 表示“类 Bank Account Class 的一个实例”，它正是我们想要建模的。在第 12 章中使用了这个表示。与此不同，在图 12-34 中，为实现 Osbert Oglesby 软件产品中的一个用例 Buy a Masterwork 的策略，给出了协作图。行动者被标注为 Osbert 而不是：Osbert（该图中有许多其他类似标注），这只是因为 Osbert 表示 Osbert 是一个行动者，而：Osbert 表示“（不存在的）Osbert Class 类的一个实例”。

7.6 节引入了信息隐藏的概念。在 UML 中，前缀“+”表示一个属性或操作是 public（公共的），类似地，前缀“-”表示属性或操作是 private（私有的）。图 16-3 中使用了这个表示，在其中声明了 Bank Account Class 的属性为私有的（以使我们能够获得信息隐藏），而两个操作是公有的，以便能够从软件产品中的任何地方调用它们。第三个标准的可视类型是 protected（受保护的），它使用前缀“#”表示。如果一个属性是公共的，它在任何地方都是可见的；如果它是私有的，它仅在定义它的类内可视；如果它是受保护的，它就在定义它的类内或者该类的子类内可视。

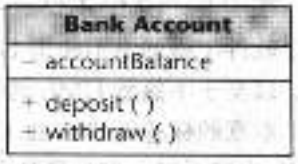



图 16-3　带有可视性前缀的图 16-2 的类图

本章到现在为止，类图仅包含已经给出的一个类。下面考虑具有更多类的类图。

### 16.2.1　聚合

考察图 16-4，它对语句“汽车由底盘、发动机、车轮和座椅组成”进行了建模。我们还记得空菱形表示聚合（aggregation）。**聚合**是 UML 中表示**局部-整体关系**的术语。汽车的局部有：底盘、发动机、车轮和座椅，菱形放在“整体”（汽车）的一端，而不是放在连接局部和整体的线的“局部”（底盘、发动机、车轮和座椅）的一端。

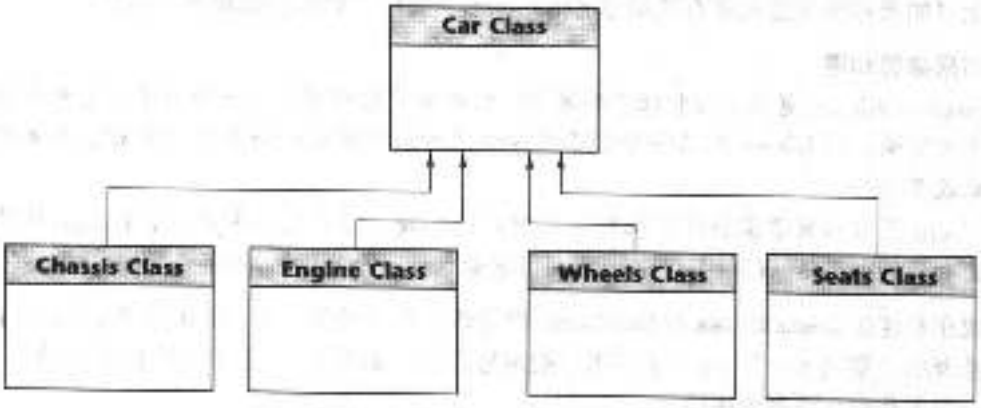



图 16-4　一个聚合的例子

### 16.2.2　多态

现在假定我们使用 UML 建模语句“一辆汽车由一个底盘、一台发动机、4 或 5 个车轮、一可选的天窗、0 或多个挂在后视镜上的放大小镜、以及 2 个或更多座椅组成。”如图 16-5 所示。线的端点处的数字表示**多态**（multiplicity），它表示一个类与其他类**关联**的次数。

先考虑连接 `Chassis Class` 与 `Car Class` 的连线。在连线的“局部”端的 1 表示在这个关系中涉及一个底盘，在“整体”端的 1 表示其中涉及一辆汽车，也就是说，每辆车有一个底盘。同样可以观察连接 `Engine Class` 与 `Car Class` 的连线。

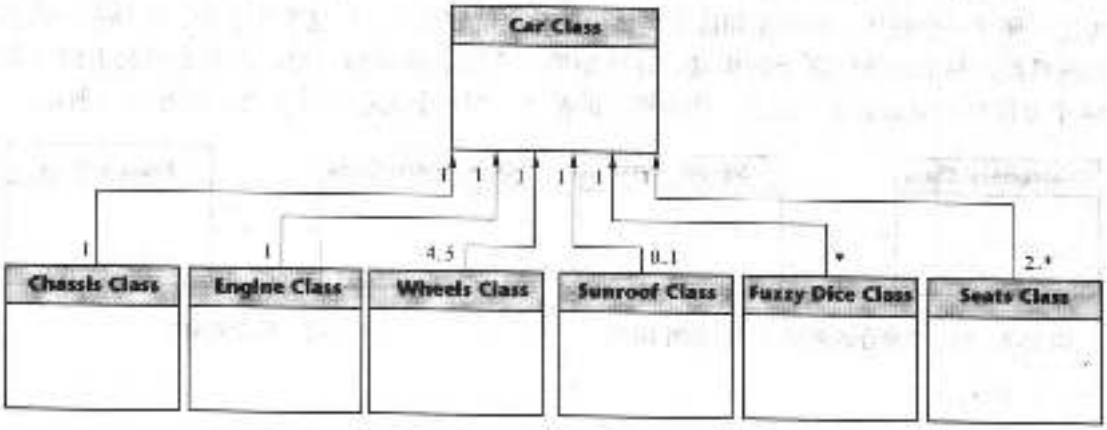



图 16-5　带有多态的聚合的例子

现在考虑连接 Wheels Class 与 Car Class 的连线。在“局部”端的 4..5 和在“整体”端的 1 一道，表示每辆车有 4 到 5 个轮子（第 5 个轮子是备胎）。因为类的实例只能取列出的全部整数，这意味着 UML 图如要求的那样，建模了“一辆车有 4 或 5 个轮子”的语句。

一般地，两个点 .. 表示范围。这样，0..1 的意思是 0 或 1，它是 UML 表示“可选”的方法。这就是为什么连接 Sunroof Class 与 Car Class 的连线的旁边是 0..1 的原因。

现在看连接 Fuzzy Dice Class 与 Car Class 的连线。在“局部”端，标记是 *。一个单独的星号表示“零或多个”。这样，图 16-5 中的 * 意味着一辆车有 0 或多个放大小镜挂在后视镜上（如果你想知道更多有关星号的信息，见下面的“如果你想知道”部分）。

**如果你想知道**

Stephen Kleene 奠定了递归理论的基础，该理论是数理逻辑的一个分支，它对计算机科学有重大影响。以 Kleene 的名字命名了 Kleene 星号（这种星号如图 16-5 所示，在类图中表示“零或多个”）。

Kleene 星号在数学家和计算机科学家中广为知晓。不广为人知的是，Kleene 将他的姓名字发音为如书面语“Clay knee”的形式（重音在第一个音节），而不是“Clean knee”。

现在看连接 Seats Class 与 Car Class 的连线。在“局部”端，标记是 2..*，一个单独的星号表示“零或多个”，在一个范围后的星号表示“或多个”。这样，图 16-5 中的 2..* 意思是一辆车有两个或多个座椅。

因此，在 UML 中，如果知道准确的多态，就使用该数值。一个例子是 1 出现在图 16-5 中的 8 个位置。如果知道了范围，就使用范围标注，就像图 16-5 中的 0..1 或 4..5 那样。如果没有指定数值，就使用星号。如果一个范围中的上限未指定，范围标注就与星号标注混合使用，就像图 16-5 中的 2..* 那样。顺便说一下，UML 的多态标注是基于传统数据库理论中的实体－关系图的（见 11.6 节）。

### 16.2.3　组合

另一个聚合的例子见图 16-6，它对象棋盘和它的方格之间的关系进行了建模，每个象棋盘由 64 个方格组成。事实上，这个关系又进了一步；它是一个有关组合的例子，一个更强形式的聚合。如前所述，聚合建模了部分－整体的关系。当存在组合的时候，则每个部分可能仅属于一个整体，而如果删除了整体，部分也就删除了。在这个例子中，如果有一些不同的棋盘，每个方格仅属于该棋盘，并且如果一个棋盘被扔到一边，在那个棋盘上的全部 64 个方格也一同失去了。组合，作为聚合的扩展，用一个实心菱形表示，如图 16-7 所示。

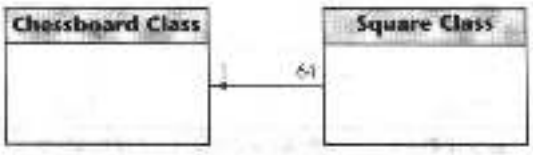



图 16-6　另一个聚合的例子（但请见图 16-7）

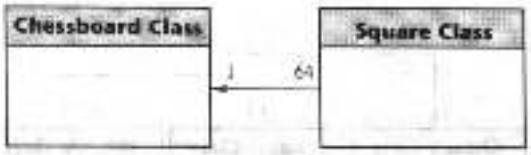



图 16-7　组合的例子

### 16.2.4　泛化

继承是面向对象要求的一个特性。继承是泛化（generalization）的一个特例。UML 用于标注泛化的是一个三角形，有时我们用一个带区别符的空心三角表示它。考虑图 16-8，它建模了两类投资：债券和股票。紧邻空心三角的标注 investmentType 表示 Investment Class 的每个实例或它的两个子类有一个属性 investmentType，并且这个属性可以用来区分债券的实例和股票的实例。

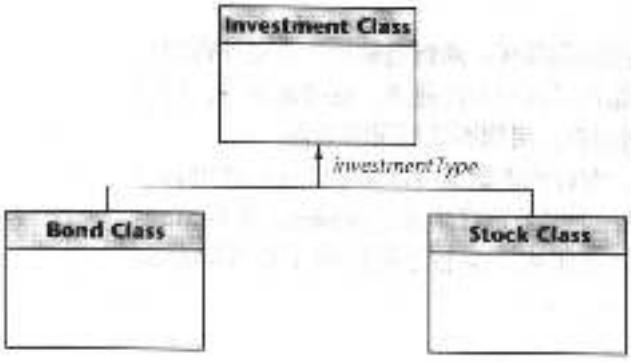



图 16-8　带有明确区别符的泛化（继承）的例子

### 16.2.5　关联

在 7.7 节，给出了一个涉及两个类的关联的例子，在其中，关联的方向需由一个实心三角形式的箭头来明确。图 7-30 在这里重新以图 16-9 画出。

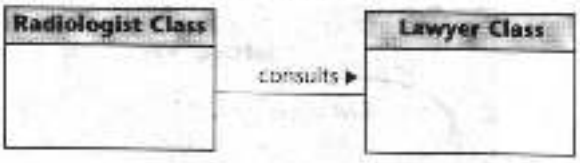



图 16-9　一个关联的例子

在某些情形下，两个类间的关联本身可能需要建模为一个类。例如，假定图 16-9 中的放射线学者在许多不同的场合下向律师咨询，每个场合都用了不同的时间长度。为了使律师能够正确地向放射线学者收费，需要一个如图 16-10 所示的类图。现在“咨询”已经成了一个类，Consults Class，称它为一个关联类（因为它既是一个关联也是一个类）。

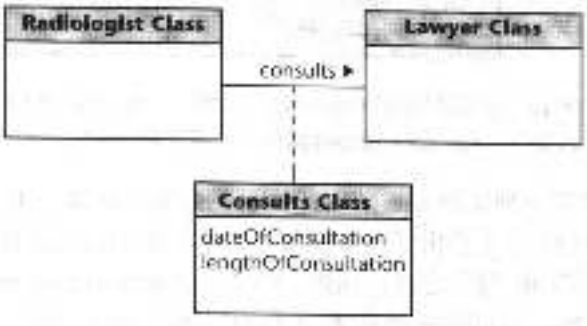



图 16-10　一个关联类

## 16.3　注解

当我们想要在UML图中包含一个注释的时候，我们把它放在一个“**注解**”（一个右上角折起的矩形框）中。然后从该注解到这个注解涉及的事项画一条虚线。图12.32显示一个注解。

## 16.4　用例图

如10.4.3节所描述的那样，**用例**是软件产品的外部用户（行动者）和软件产品自身的交互的述模。更准确地说，行动者是担负特殊任务的用户。**用例图**是用例的集合。

在10.4.3节中，在行动者范围内描述了泛化，如图10-2所示。图16-11是另一个例子，它显示了**Manager**是**Employee**的一个特殊实例。至于类，空心三角指向了更普遍的实例。




图16-11　一个行动者的泛化

## 16.5　模板

美国个人所得税的三个最主要税收形式是：Form 1040，Form 1040A以及Form 1040EZ。图16-12显示用例Prepare Form 1040，Prepare Form 1040A以及Prepare Form 1040EZ都包含了用例Print Tax Form，从由一个模板（Stereotype）给出的include关系可看出这一点。

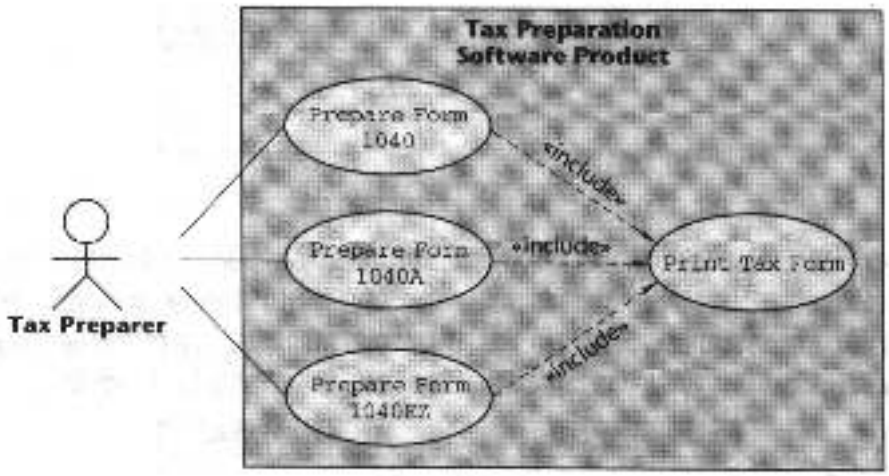



图16-12　全部的用例Prepare Form 1040，Prepare Form 1040A以及Prepare Form 1040EZ都包含了用例Print Tax Form

在UML中，模板是一种扩展的方法。即如果我们需要定义一个不在UML中的构造，我们能够做到它。在第12章中给出了三个模板：边界、控制以及实体类。一般地，模板的名字出现在书名号“《”和“》”之间，例如，《This is my own construct》。这样，不是使用特殊的符号表示边界类，而是可能使用表示类的标准矩形符号在矩形内标注《boundary

class》，对于控制类和实体类也类似。

图 16-12 中显示的包含关系在 UML 中被当作模板对待，因此，在该图中的标注《include》表示公用功能，在这个实例中，是用例 Print Tax Form。另一个关系是扩展关系，在那里一个用例是标准用例的变体。例如，我们可能想有一个独立的用例来建模一个潜在的艺术品出售者拒绝 Osbert 的出价的情形。标注《extend》同样用于这个目的，如图 16-13 所示。然而，对于这个关系，开口箭头指向另一个方向。

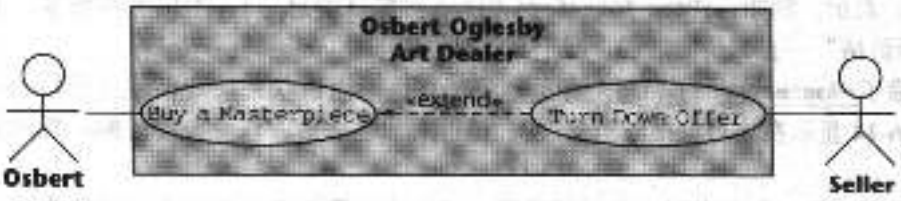



图 16-13 当客户拒绝 Osbert 的出价时，用户实例 Buy a Masterpiece 显示其变化

## 16.6 交互图

交互图显示软件产品中的对象与另一个对象交互的方式。在第 12 章中介绍了两种类型 UML 支持的交互图：顺序图和协作图。

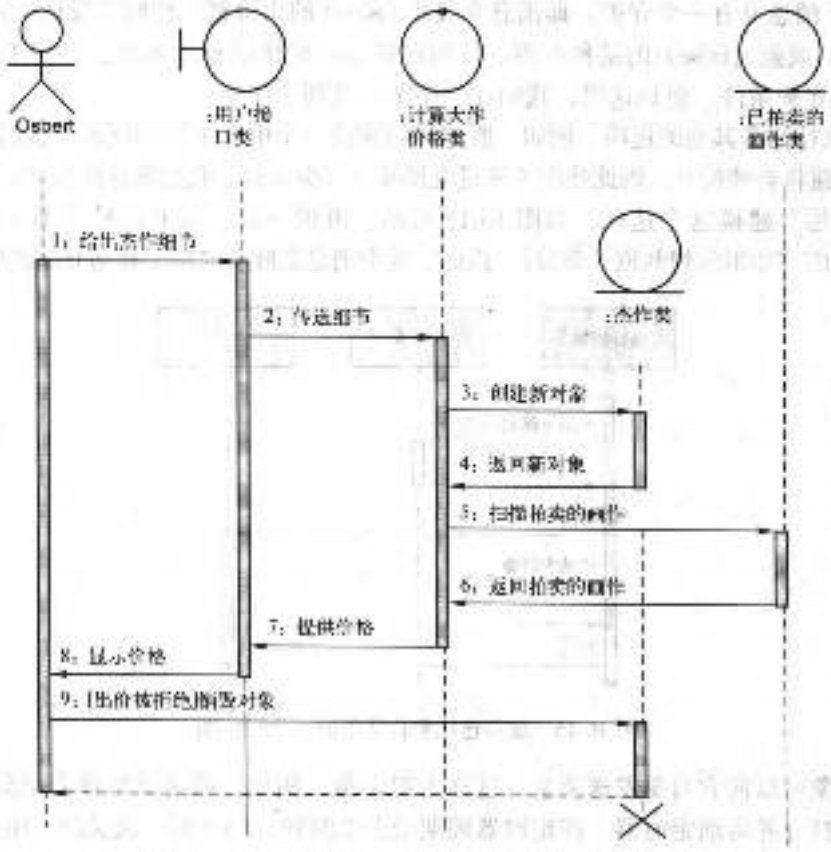



图 16-14 顺序图显示一个对象的动态构建和析构，返回，以及显式的激活

首先，考虑顺序图。图 12-36 显示当动态构建一个对象的时候，一个顺序图是什么样子的。就像对象可以构建一样，它们也可以被销毁。例如，如果 Osbert 为一幅杰作出价，但该出价被拒绝了，那就不需要保留这个为了对 Osbert 的出价进行最高价格计算而创建的杰作对象。图 16-14 描述了一个杰作对象的创建以及随后的删除。这个顺序图是松散地建立在图 12-36 的基础上的，但是它包含一些可选特性：

1）考虑图 16-14 中的生命线。当一个对象激活时，这一点由虚线位置的一个窄矩形（激活框）表示。例如，:User Interface Class 对象从消息“1：给出杰作细节”直至消息“8：显示价格”一直都是激活的，对于其他对象也如此。

2）除了:Masterpiece Class 对象的创建外，这一点由仅在动态创建点开始的生命线表示，图 16-14 显示在该对象接收消息“9：销毁对象”之后，该对象的析构。析构由大大的×表示。

3）这个析构发生在出现一个返回之后，这是由在事件 9 下面的水平虚线表示的，它终止于一个开口箭头。在前面全部的顺序图中，每个消息最终都跟随着一个消息，该消息是由发送最初消息的对象发回的。事实上，这个交互作用是可选的，发送一个消息而最终没有收到任何类型的应答是非常可能的。即使返回了一个应答，也没有必要发回特定的新的消息。相反，画出了一个终止于开口箭头的虚线（返回一个应答），以指示从最初消息处的一个返回，对应于一个新的消息。

4）对于消息 9 有一个守护，即消息 9 仅当 Osbert 的出价被拒绝时才发送。守护（条件）是真（true）或假（false）的某种东西，仅当它是 true 时该消息才发送。我们不久将会在状态图中遇到守护条件，但是这里，我们在顺序图中见到了守护。

UML 支持许多其他的选项。例如，假定我们建模一个电梯上升的问题。我们事先不知道哪个电梯按钮将会被按下，因此我们不知道电梯要上多少层楼。我们通过标注相关的消息“* 向上移动一层”建模这个迭代，如图 16-15 所示。再说一次，这个星号是 Kleene 星号（见 16.2.2 节中的“如果你想知道”部分）。因此，这个消息意味着“向上移动 0 层或更多层”。

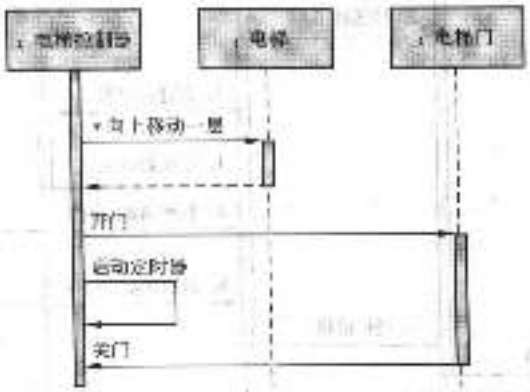



图 16-15　显示迭代和自调用的一个顺序图

一个对象可以向它自身发送消息，这称为**自调用**。例如，假定该电梯已经到达某层楼。电梯门现在打开并启动定时器。在定时器周期结束的时候，门又再一次关闭。电梯控制器向它自己发送一个消息来启动它的定时器，这个自调用也显示在图 16-15 中。

现在转到协作图（collaboration diagram）上来，在 12.5.1 节指出过，协作图等效于顺序图。因此，这一节中给出的顺序图的全部特性同样适用于协作图。

## 16.7　状态图

考虑图 16-16 的状态图。这与图 12-24 的状态图类似，建模使用了守护而不是事件。它显示带有未标注“转移”的状态图（实心圆）指向状态 Osbert Oglesby Event Loop。从该状态引出 5 个转移，每个带有一个守护，即一个是 true 或 false 的条件。当其中一个守护成为 true 时，发生相应的转移。

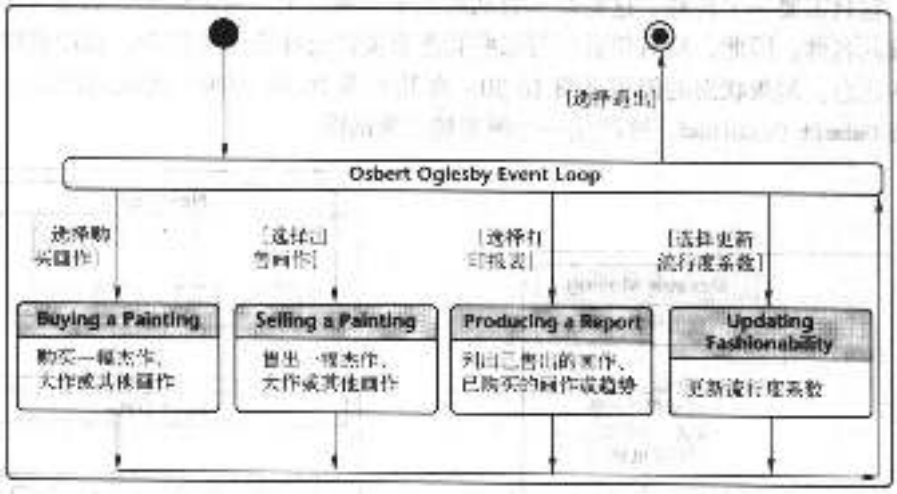



图 16-16　Osbert Oglesby 实例研究的状态图

**事件**也在状态间引起转移。一个常见的事件是接收到一个消息。考虑图 16-17，它描述了电梯状态图的一部分。电梯处在状态 **Elevator Moving**，它处于移动过程中，执行操作“向上移动一层”，此时守护［还没有收到消息］保持为 true，直至它收到消息“电梯已经到达楼层”。收到这个消息（事件）导致守护为 false，并且也启动到状态 **Stopped at Floor** 的转移。在这个状态，执行活动“打开电梯门”。




图 16-17　电梯状态图的一部分

至此，转移的标志一直是［守护］或事件的形式。事实上，转移标志最通用的形式是：

事件［守护］/动作

即如果事件发生了并且［守护］是 true，那么出现转移，并且当它出现时，执行动作（action）。这样一个转移标志的例子见图 16-18，它与图 16-17 等效。转移标志是“电梯已经到达楼层［已经收到一个消息］/打开电梯门”。当事件“电梯已经到达楼层”发生时，守护［已经收到一个消息］为真，结果就发送了一个消息，执行了动作，它由斜线/后的指示“打开电梯门”说明。

比较图 16-17 和图 16-18，我们看到，在一个状态图中有两个地方可以执行一个动作。首先，如图 16-17 中状态 **Stopped at Floor** 所反映出来的，一个动作可以在进入某个状态时执行。这样一个动作在 UML 中称为**活动**（activity）。其次，如图 16-18 所示，一个动作可

以作为转移的一部分发生。(技术上讲，在动作和活动间略微有些不同，动作是假定本来要在瞬间发生的，但活动可能不那么快发生，也许要在几秒后发生。)

UML 支持状态图中各种不同类型的动作和事件。例如，可以用词语如 when 或 after 规定一个事件。因此，一个事件可能在 when (cost>1000) 或 after (2.5seconds) 时激发。

带有大量状态的状态图会有大量的转移。代表这些转移的箭头不久就会使得状态图看起来像是一大碗意大利面条，非常复杂。解决这个问题的一个技术是使用**超级状态** (superstate)。考虑图 16-19a 的状态图，四个状态 A、B、C、D 都有到 Next State 的转移。图 16-19b 显示这四个状态如何可以合成为一个超级状态 ABCD Combined，与图 16-19a 的四个转移不同，它只需要一个转移。这将箭头数从四个减少到一个。与此同时，状态 A、B、C 和 D 仍然保留其名称，因此，所有现有的与那些状态有关的动作既不受影响，也没有现有的转移进入那些状态。超级状态的例子见图 16-20，在其中图 16-16 的四个较低级状态合成为一个超级状态 Osbert Combined，这产生一个越来越清晰的图。




图 16-18　等效于图 16-17 的状态图

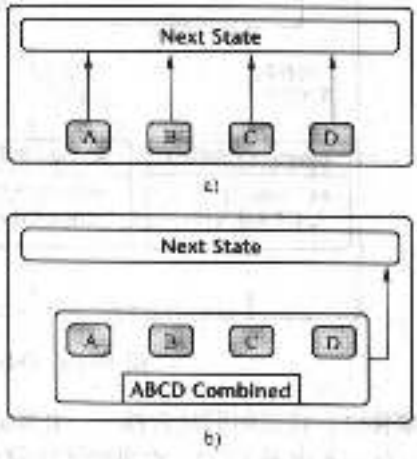



图 16-19　状态图：a) 没有超级状态　b) 带有超级状态

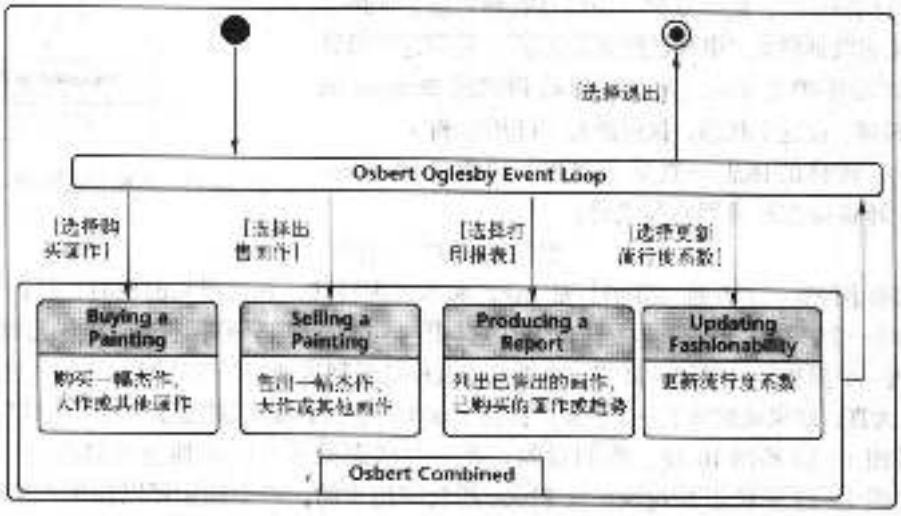



图 16-20　带有四个状态的图 16-16 合并成为一个超级状态 Osbert Combined

## 16.8　活动图

活动图显示各种事件是如何协调的，因此，当活动并行进行时使用它。

假定坐在餐馆的一对情侣开始点餐，一个人点了一盘鸡肉，另一个人点了鱼。侍者写下他们点的菜，将订单递给厨师，使她知道准备什么菜。哪道菜先完成并没有关系，因为仅当两盘菜都准备好了之后，送餐服务才开始。这情形见图 16-21。上面的加粗水平线称为交叉，下面的称为结合。一般地，交叉有一个输入转移和多个输出转移，其中的每个都开始一个要与其他活动并行执行的活动。当全部并行活动完成时，开始一个输出转移。

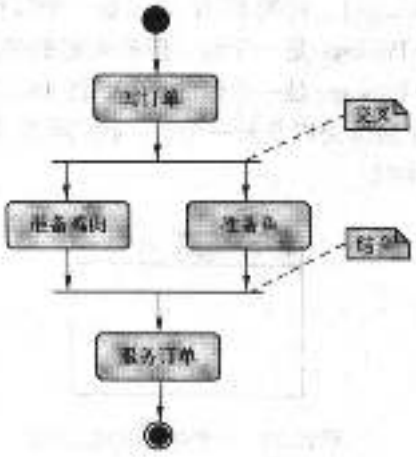



图 16-21　餐馆为两个进餐者订餐的活动图

活动图对于建模一个大量活动并行进行的商业活动非常有用。例如，考虑一个组装客户指定（配置）的计算机的公司，如图 16-22 的活动图所示，当收到一个订单时，它被传递到组装部门（Assembly Department）；它也被传递到结算部门（Accounts Receivable Department）。当计算机组装好并交付，并且客户的支付处理完后，该订单完成。涉及的三个部门中的每一个——组装部门、订货部门（Order Department）、结算部门，都在它自己的泳道内。通常，交叉、结合、泳道清楚地显示了每个特定活动涉及一个组织的哪个分支，哪些任务是并行完成的，以及哪些任务在下一个任务可以开始前需要并行完成。

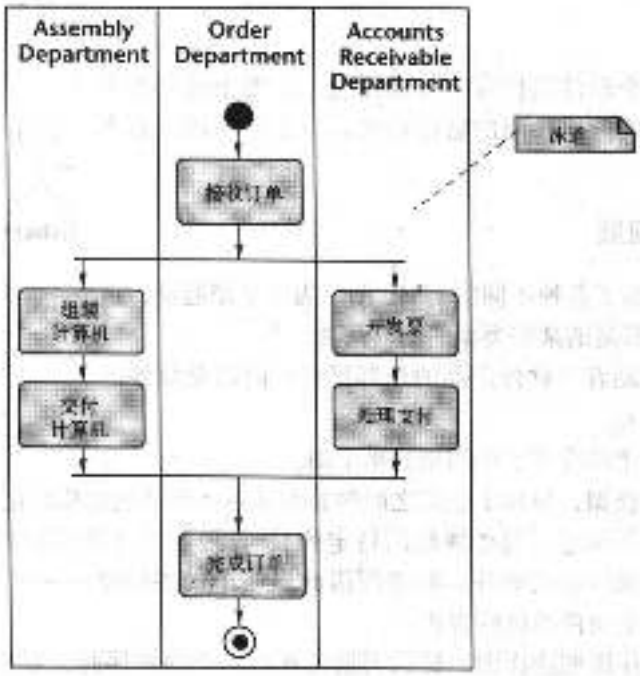



图 16-22　一个计算机组装公司的活动图

## 16.9　包

如13.9节所解释的那样，处理大型软件产品的方法是将它分解成为相对独立的包（package）。包的UML表示是一个带有名字标签的矩形框，如图16-23所示。这个图显示了My Package是一个包，但是矩形框是空的。这是一个有效的UML图——该图简单建模了My Package是一个包的事实。图16-24更有趣，它显示了My Package的内容，包括一个类、一个实体类以及另一个包。我们可以连续提供多个细节，直至该包对于目前的迭代和递增足够详细。

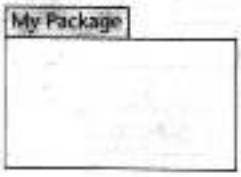



图16-23　一个包的UML表示

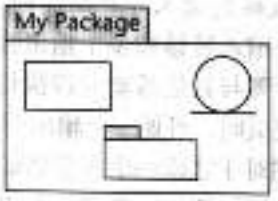



图16-24　图16-23带有多个细节显示的包

## 16.10　组件图

**组件图**显示了软件组件之间的依存度，软件组件包括源代码、编译代码以及可执行载入映像。例如，图16-25的组件图显示源代码（由一个便笺代表）和由源代码产生的可执行载入映像。

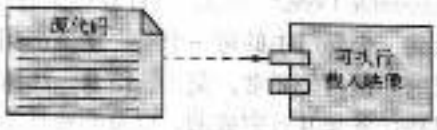


图16-25　组件图

## 16.11　配置图

**配置图**显示每个软件组件安装（或配置）在哪个硬件组件上。它也显示了在硬件组件间的通信链接。一个简单的配置图见图16-26。

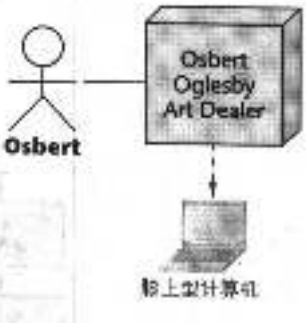



图16-26　配置图

## 16.12　UML图回顾

在这一章中介绍了各种不同的UML图。为了清楚起见，这里给出可能会引起混淆的某些类型的图的列表：

- 用例建模行动者（软件产品的外部用户）间以及软件产品本身的交互。
- 用例图是一个结合了一些用例的单个图。
- 类图是类的模型，显示了它们之间的静态关系，包括关联和泛化。
- 状态图显示了状态（对象属性的特定值）、导致状态（受守护约束）之间转移的事件，以及对象采取的动作。状态图因此是一个动态模型——它反映了对象的行为，即它们对特定事件的反应方式。
- 交互图（顺序图或协作图）显示当消息在对象之间传递时，对象相互之间交互的方式。这是另一种动态模型，即它也显示了对象的行为。

- 活动图显示了发生在同一时间的事件是如何协调的，这仍然是另一种动态模型。

## 16.13　UML 和迭代

来看一下状态图。转移可以标注为一个守护、一个事件、一个行动或者它们三者。现在再看顺序图。生命线可以包括也可以不包括激活框，可以有返回也可以没有返回，并且可以有也可以没有对消息的守护。

对于每个 UML 图有广泛的选项可用。即一个有效的 UML 图由一个小的必须的部分再加上任何数量的选项组成。UML 图有如此多的选项有两个原因：首先，不是每个 UML 特性都可应用于每个软件产品，因此必须有选择的自由。第二，除非允许我们逐步向图中加入特性，否则就无法执行统一过程的迭代和递增，更不要说在一开始就创建完整的最后的图了。即是说，UML 允许我们从一个基本的图开始，然后我们可以加入想要的选项，记住，在任何时候，得到的 UML 图仍然是有效的。这就是为什么 UML 非常适合统一过程的众多原因之一。

## 本章回顾

在 16.1 节解释了 UML 是一种语言，而不是一种方法。在 16.2 节描述了类图，讨论了类图的特定方面，包括聚合（16.2.1 节）、多态（16.2.2 节）、组合（16.2.3 节）、泛化（16.2.4 节）以及关联（16.2.5 节）。接下来，给出了各种 UML 图，包括注解（16.3 节）、用例图（16.4 节）、模板（16.5 节）、交互图（包括顺序图和协作图，见 16.6 节）、状态图（16.7 节）、活动图（16.8 节）、包（16.9 节）、组件图（16.10 节），以及配置图（16.11 节）。本章结尾处回顾了 UML 图（16.12 节），并且讨论了 UML 非常适合统一过程的原因（16.13 节）。

## 进一步阅读

没有什么能够比阅读当前版本的 UML 手册更好了，可以在 OMG 网站 www.omg.org 找到它。两个有关 UML 介绍的好文章是［Fowler and Scott，2000，and Stevens and Pooley，2000］。

## 习题

16.1　使用 UML 建模股票市场。（提示：不要给出超出回答问题所需要的过多细节。）

16.2　使用 UML 建模巧克力蛋糕。巧克力蛋糕用鸡蛋、面粉、糖、烘焙粉、牛奶和可可制成。一个巧克力蛋糕经过混合、烘制、着粉，然后被吃掉。为了禁止未经授权的个人烘制巧克力蛋糕，配方是不公开的，其他操作过程也如此，最后一道操作过程除外。

16.3　使用 UML 建模一个饭厅。每个饭厅必须有一张桌子、四把或更多的椅子，以及一个餐具柜。可选地，它也可以有一个壁炉。

16.4　结合使用聚合和组合，建模习题 16.3 的饭厅。

16.5　修改习题 16.3 的 UML 模型，以反映饭厅是一种特殊类型的房间。

16.6　给你的习题 16.2 的图加上一个注解，指出你建模的蛋糕是一个巧克力蛋糕。

16.7　John Cage 于 1952 年创作了一首有些争议的标题为 4′33″的钢琴作品，用 UML 建模该

作品。乐曲包含了三个无声乐章，长度分别为：30 秒、2 分 23 秒、1 分 40 秒（作品的标题来自于它的总长度）。钢琴家走上舞台，手持一个秒表和乐谱（使用常规的音乐记号，但是各小节却是空白的）。钢琴家坐在琴凳上，把乐谱和秒表放在钢琴上，打开乐谱，启动秒表，然后通过放低钢琴的盖子示意第一个乐章的开始。在第一个乐章结束时（即在 30 秒的静默后，这期间钢琴家小心地跟随着空白乐谱，必要时翻动乐谱页），升高钢琴的盖子以示意第一乐章的结束。在第二个乐章（2 分 23 秒）和第三个乐章（1 分 40 秒）重复这些动作。钢琴家然后合上乐谱，收起乐谱和秒表，起身并离开舞台。

## 参考文献

[Fowler and Scott, 2000] M. FOWLER WITH K. SCOTT, *UML Distilled*, 2nd ed., Addison Wesley, Upper Saddle River, NJ, 2000.

[Stevens and Pooley, 2000] P. STEVENS WITH R. POOLEY, *Using UML: Software Engineering with Objects and Components*, updated edition, Addison Wesley, Upper Saddle River, NJ, 2000.

# 附　录

## 附录 A　学期项目：Amlet 沙漠中 Ophelia 的 Oasis

Ophelia 的 Oasis 是位于澳大利亚 Amlet 沙漠一座山顶的拥有 45 个房间的宾馆，可以俯瞰 Amlet 沙漠最优美的风景。每个房间都可一眼看到沙漠，并且视线可延伸到水平方向 15 到 20 英里以外。

在宾馆入住的花费取决于时间和预订的类型。管理层至少提前一年就确定了所有 45 个房间在当天的收费价格，这个价格被称为“基价”。冬天（高峰季节）和假日时这个基价更高一些。

有四种类型的预订：

1）预付金预订。客人将预订的入住时间的全部花费一次预付清。预付价格是基价的 75%。预付金预订需要提前至少 90 天，如果取消预付金预订将不退款，但可以更改预订（条件是能提供要更改的房间），此时需要为更改的预订额外支付的金额是：当前基价的 110%减去原来的预付金。（如果更改后的预订比原来的预订便宜，也不退款。）

2）提前 60 天预订。如果进行提前 60 天预订，客人可提供 e-mail 地址，在他们入住前 45 天开始，将有一封 e-mail 发给客人，通知客人在 15 天内支付全款或者取消预订。每天早晨，一位雇员必须生成这些 e-mail 信息。提前 60 天预订的价格是基价的 85%。如果取消提前 60 天预订将不退款，但可以更改预订（条件是能提供要更改的房间），此时需要为更改的预订额外支付的金额是：当前基价的 110%减去原来的预付金。（如果更改后的预订比原来的预订便宜，也不退款。）

3）常规预订。对于常规预订，客人在停留的最后一天付款。然而，在预订时客人需要提供信用卡号码，如果信用卡没有得到确认或者入住前 3 天内取消预订，需要支付第一天的房费。可以在任何时候进行常规预订，并随时都可以更改（需得到确认）。

4）奖励预订。该宾馆距离最近的城镇有 125 英里，因此管理层在高峰期雇用临时雇员很困难，实际上管理层应确保所有时候都有足够的雇员来应付客满的情况。然而，为支付由此带来的高额劳动成本，管理层不得不力争每晚都客满，特别是，对于提前 30 天预订的情况，如果平均的客人住房率为 60%或更少，报出的价格将降低 20%。例如，假设在提前 25 天时进行了预订，预订从周日住到周四（四个晚上），那四个晚上的平均预订房间数为当前的 22.5 个房间，那么那个时候的平均期望住房率为 50%，因此那四晚中的每一晚上的价格将为基价的 80%。客人在最后一天为奖励预订付款，与常规预订一样。奖励预订与常规预订具有相同的取消和更改规定。

所有的付款都必须使用信用卡，不管是在入住第一天付款，还是在最后一天付款。预订时必须提供相关的信用卡信息，除了提前 60 天预订，提前 60 天预订的信用卡信息必须在到达日前 30 天之前提供。

管理层可以在任一天的任何时候更改基价，因此，预期的客人预订房间时，可以告知他们每晚的价格是不同的，同时，基价的更改一定不能影响任何现有的预订。

不能为某个特定房间提出预订，房间是在预期到达的那天的早晨才分配的，客人登记时才告知客人房间号。没有登记或预订的第一天下午 11:59 才打电话来确认房间预订的客人将失去预订的房间，并需要支付罚款。每天早晨都有一个雇员来生成罚单。

将保留所有记录，例如，如果取消了一个预订，该预订将被修改以反映这个取消，进一步地说，该软件产品必须在每个事务的最后将数据写入磁盘。应在每个工作日的最后生成所有文件的备份拷贝，并存储在另外的地点。

预订系统必须能够打印许多报表，管理层每天需要生成三种报表：

- “预计入住的报表”是一页纸的管理报表，显示未来 30 天中每个晚上在当前已被预订的房间数。报表的每一行显示日期，预付金预订、提前 60 天预订、常规预订和奖励预订的数量，以及已被预订的房间总数。报表的最后一行显示那个时期的平均预计入住率。
- “预计房间收入报表”是一页纸的管理报表，显示未来 30 天中每个晚上从房间出租上预计得到的收入。报表的每一行显示日期和那个晚上预计的收入。报表的最后两行是那个时期的总收入和平均收入。
- “奖励报表”是一页纸的管理报表，显示 20% 的奖励折扣给未来 30 天带来多少金钱上的损失。报表的每一行显示日期和那个晚上总的奖励折扣。报表的最后两行是那个时期总的奖励折扣和平均奖励折扣。

由雇员每天生成的两种报表是操作报表：

- “每日到达的报表”是预计在那一天到达的客人的列表。报表的每一行显示客人姓名、预订类型、预分配的房间号和离开日期，该报表按客人的姓名排序。
- “每日入住的报表”是当前还停留在宾馆的客人的列表。报表的每一行显示房间号、客人姓名和离开日期（如果该房间在前一晚还有客人住）。如果客人当天离开，姓名前加注星号。该报表按房间号排序。

另外，产品必须能够打印“发票”，能够反映该票据打印的日期、客人姓名、房间号、到达日期、离开日期、入住的天数和总金额。对于预付金预订和提前 60 天预订的情况，该票据还必须能够反映提前支付的日期和金额。在客人结账时应把发票交给客人。

# 附录 B　软件工程资源

有两种好办法来获得软件工程方面的更多信息：阅读期刊和会议文献汇编，并借助于因特网。

有一些软件工程方面的专刊，例如《IEEE Transactions on Software Engineering》，还有一些更通用的刊物，例如《Communications of the ACM》，里面刊有关于软件工程方面的重要文章。由于篇幅的限制，下面只选择了这两种类型的一些刊物，这些刊物都是我自认为最有用的。

《ACM Computing Reviews》

《ACM Computing Surveys》

《ACM SIGSOFT Software Engineering Notes》

《ACM Transactions on Computer Systems》

《ACM Transactions on Programming Languages and Systems》

《ACM Transactions on Software Enginneering and Methodology》

《Communications of the ACM》

《Computer Journal》

《Empirical Software Engineering》

《IBM Systems Journal》

《IEEE Computer》

《IEEE Software》

《IEEE Transactions on Software Engineering》

《Journal of Systems and Software》

《Software Engineering Journal》

《Software—Practice and Experience》

此外，许多会议的文献汇编包含有软件工程方面的重要文章，下面是凭主观选择的会议，大多数的会议用它们的缩写或主办公司的名称表示，显示在圆括号里。

ACM SIGPLAN Annual Conference（SIGPLAN）

ACM SIGSOFT Symposium on the Foundations of Software Engineering（FSE）

Conference on Human Factors in Computing Systems（CHI）

Conference on Object-Oriented Programming Systems, Languages, and Applications（OOPSLA）

International Computer Software and Applications Conference（COMPSAC）

International Conference on Software Engineering（ICSE）

International Conference on Software Maintenance（ICSM）

International Conference on Software Reuse（ICSR）

International Conference on the Software Process（ICSP）

International Software Architecture Workshop（ISAW）

International Symposium on Software Testing and Analysis（ISSTA）

International Workshop on Software Configuration Management（SCM）

International Workshop on Software Specification and Design（IWSSD）

因特网是获得有关软件工程方面信息资源的另一个有效途径，关于网络新闻组，下面是对我来说相当有用的两个：

comp.object

comp.software-eng

另外一些新闻组有时也会有相关的信息，包括：

comp.lang.c++.moderated

comp.lang.java.programmer

comp.risks

comp.software.config-mgmt

# 附录 C　需求流：Osbert Oglesby 实例研究

第 10 章描述了 Osbert Oglesby 实例研究的需求流。

# 附录 D　结构化系统分析：Osbert Oglesby 实例研究

**第 1 步：画出数据流图，**参见图 11-8。

**第 2 步：决定计算机化什么部分，及如何去做。**因为客户将在远程的笔记本电脑上使用该产品，确定预期的买卖，整个产品应被计算机化并能够在线操作。

**第 3 步：给数据流图加入细节。**

| | |
|---|---|
| buy_painting_information | |
| first_name_of_artist | (21 个字符) |
| last_name_of_artist | (21 个字符) |
| title | (41 个字符) |
| year_of_work | (5 个字符) |
| medium | (9 个字符) |
| subject | (10 个字符) |
| height | (4 个数字) |
| width | (4 个数字) |
| classification | (10 个字符) |
| | |
| purchase_information | |
| date_of_purchase | (10 个字符) |
| name_of_seller | (30 个字符) |
| address_of_seller | (40 个字符) |
| maximum_purchase_price | (8 个数字) |
| actual_purchase_price | (8 个数字) |
| | |
| gallery_painting_information | |
| buy_painting_information | (参见上面) |
| buying_information | (参见上面) |
| | |
| sell_painting_information | |
| first_name_of_artist | (21 个字符) |
| last_name_of_artist | (21 个字符) |
| title | (41 个字符) |
| | |
| sale_information | |
| target_selling_price | (8 个数字) |
| date_of_sale | (10 个字符) |
| name_of_buyer | (30 个字符) |
| address_of_buyer | (40 个字符) |
| actual_selling_price | (8 个数字) |

```
painting_details
    gallery_painting_information        (参见上面)
    selling_information                 (参见上面)

reports
    report_type                         (12 个字符)
```

**第 4 步：定义过程的逻辑。**

```
buy
      如果画作已经购买过,打印出错信息,
      否则,
            计算 maximum_purchase_price。
            如果 maximum_purchase_price>0
                取 purchase_information。
                把 gallery_painting_information 加到 GALLERY_PAINTING 中。
sell
      取 sell_painting_information。
      如果画作已经购买过,打印出错信息,
      否则,
        把 sale_information 加到 GALLERY_PAINTING 中。
print_report
      对于去年处理过的每个 GALLERY_PAINTING 对象,
        如果该对象满足报表类型的要求,打印 painting_details。
```

**第 5 步：定义数据存储。**

```
AUCTION_PAINTING
    first_name_of_artist            (21 个字符)
    last_name_of_artist             (21 个字符)
    title                           (41 个字符)
    year_of_work                    (5 个字符)
    medium                          (9 个字符)
    subject                         (10 个字符)
    height                          (4 个数字)
    width                           (4 个数字)
    auction_date                    (10 个字符)
    auction_price                   (8 个数字)

FASHIONABILITY
    first_name_of_artist            (21 个字符)
    last_name_of_artist             (21 个字符)
    coefficient                     (4 + 4 个数字)

GALLERY_PAINTING
    first_name_of_artist            (21 个字符)
```

| | |
|---|---|
| last _ name _ of _ artist | (21 个字符) |
| title | (41 个字符) |
| year _ of _ work | (5 个字符) |
| medium | (9 个字符) |
| subject | (10 个字符) |
| height | (4 个数字) |
| width | (4 个数字) |
| classification | (10 个字符) |
| date _ of _ purchase | (10 个字符) |
| name _ of _ seller | (30 个字符) |
| address _ of _ seller | (40 个字符) |
| maximum _ purchase _ price | (8 个数字) |
| actual _ purchase _ price | (8 个数字) |
| target _ selling _ price | (8 个数字) |
| date _ of _ sale | (10 个字符) |
| name _ of _ buyer | (30 个字符) |
| address _ of _ buyer | (40 个字符) |
| actual _ selling _ price | (8 个数字) |

所有文件都是有顺序的，因而没有 DIAD。

**第 6 步：定义物理资源。**

AUCTION _ PAINTING
　　顺序文件
　　存储在磁盘上
FASHIONABILITY
　　顺序文件
　　存储在磁盘上
GALLERY _ PAINTING
　　顺序文件
　　存储在磁盘上

**第 7 步：确定输入/输出的规格说明。**为下列过程设计输入屏幕：get _ gallery _ description，buy _ painting，sell _ painting，update _ fashionability _ coefficient。显示下列的报表：list _ of _ bought _ painting，list _ of _ sold _ paintings，current _ fashion _ trends。快速原型的屏幕和报表作为这些屏幕和报表的基础。

**第 8 步：进行规模估算。**该软件大约需要 4 MB 的存储器，每个画廊画作对象需要大约 500 字节的存储器。因此，每 1000 幅画作存储于硬盘中大约需要 0.5 MB 的辅助存储器。

**第 9 步：确定硬件需求。**

带有硬盘的笔记本电脑，运行 Linux。

用于备份的 Zip 驱动器。

用于打印报表的激光打印机。

下载拍卖数据的光盘驱动器。

# 附录 E　分析流：Osbert Oglesby 实例研究

分析流在第 12 章中给出。

# 附录 F　软件项目管理计划：Osbert Oglesby 实例研究

这份开发 Osbert Oglesby 软件产品的计划是由包含三个人的开发公司拟制的，这三个人分别是：Pat（公司老板）和两个软件专业人员，Robin 和 Dale。

**1　简介**

**1.1　项目概述。**

1.1.1　**意图、范畴和目标。**这个项目的目标是开发一个软件产品，能够帮助 Osbert Oglesby，艺术品经销商（OOAD）在他的画廊内展出和售出的画作买卖中做出决策。该产品允许客户购买和售出杰作、大作和其他画作。该软件产品将完成所要求的计算，并跟踪记录这些画作，生成报表列出已买进的画作、售出的画作和当前的流行趋势。

1.1.2　**假设和限制。**包含下列的限制：

必须满足最后期限。

必须满足预算限制。

产品必须是可靠的。

结构必须是开放的，以便将来增加额外的功能。

产品必须是对用户友好的。

1.1.3　**项目可交付使用。**整个产品，包含用户手册，将在项目开始后 10 个星期可交付使用。

1.1.4　**时间表和预算概述。**每个工作流的持续时间、人员要求和预算如下所示：

需求流（1 个星期，2 个小组成员，3380 美元）

分析流（2 个星期，2 个小组成员，6760 美元）

设计流（2 个星期，2 个小组成员，6760 美元）

实现流（3 个星期，3 个小组成员，15 210 美元）

测试流（2 个星期，3 个小组成员，10 140 美元）

总的开发时间是 10 个星期，总的内部成本为 42 250 美元。

1.2　**项目管理计划的演变。**项目管理计划中的所有修改在被实施前必须经过 Pat 同意。所有修改都必须形成文档以保持项目管理计划是正确的和最新的。

**2　参考材料**　所有模块都将符合公司的编码、编制文档和测试标准。

**3　定义和术语**　OOAD——Osbert Oglesby，艺术品经销商；Oglesby 先生是我们的客户。

**4　项目组织**

4.1　**外部接口。**这个项目的所有工作都由 Pat、Robin 和 Dale 完成。Pat 每周与客户见一次面，报告进展情况，并讨论可能的修改和调整。

4.2　**内部结构。**开发小组包含 Pat（老板）、Robin 和 Dale。

4.3　**角色和职责。**Robin 和 Dale 将完成设计流程，Pat 将实现类定义和报表模块，Robin 将构建处理已买入画作的模块，Dale 开发处理已卖出画作的模块。每个成员负责他或她所生成模块的质量。Pat 将监视集成和软件产品的整体质量，并与客户保持联络。

5　**管理过程计划**

5.1　**启动计划。**

5.1.1　**估算计划。**如前所述，整个开发时间估计为 10 周，整个内部成本为 42 250 美元。这些数字是通过类推的专家判决得到的，即通过与类似的项目对比而得到的。

5.1.2　**人员计划。**整个 10 周都需要有 Pat 参加，前 5 周只是管理层面的内容，而第二个 5 周则既有管理者，也有程序员的内容。Robin 和 Dale 在整个 10 周内也都需要参与，前 5 周作为系统分析员和设计者，而第二个 5 周则作为程序员和测试者。

5.1.3　**资源获取计划。**该项目所有必需的硬件、软件和 CASE 工具已经具备，该产品将交付给 Osbert Oglesby，安装在可从通常的供应商那租借的笔记本电脑上。

5.1.4　**项目人员培训计划。**这个项目不需要额外的人员培训。

5.2　**工作计划。**

5.2.1-2　**工作活动和时间表分配。**

第 1 周　　与客户见面，确定需求模块。审查需求模块。

第 2、3 周　生成分析模块，审查分析模块，给客户展示分析模块，征得客户的同意。生成项目管理计划，审查项目管理计划。

第 4、5 周　生成设计模块，审查设计模块。

第 6～10 周　实现并审查每个类、单元测试和文档。对每个类进行集成，进行集成测试、产品测试，审查文档。

5.2.3　**资源分配。**三个小组成员将在他们负责的模块上单独工作，Pat 的职责是监控另两个人的每日进展，监视实现过程，负责整个产品的质量，并与客户交互。小组成员在每天工作结束时会面，讨论问题和进展。与客户正式的会议将在每周结束时召开，报告进展情况并确定是否需要修改。Pat 将确保时间表和预算需求与计划相符合，风险管理也是 Pat 的职责。

使错误最少，使用户友好程度最大化是 Pat 的首要任务。Pat 负责整个文档的质量，并确保这些文档是最新的。

5.2.4　**预算分配。**每个工作流的预算如下所示：

| | |
|---|---|
| 需求流 | 3380 美元 |
| 分析流 | 6760 美元 |
| 设计流 | 6760 美元 |
| 实现流 | 15 210 美元 |
| 测试流 | 10 140 美元 |
| 总计 | 42 250 美元 |

5.3　**控制计划。**任何影响里程碑或预算的主要修改必须得到 Pat 的同意，并形成文档，这里不涉及外部的质量保证人员。每个人测试其他人的工作成果，通过这种方式确保测试的公正性。

Pat 将负责确保该项目按时完成，并不超出预算，这通过每天与小组成员的例会实现。在每次会议上，Robin 和 Dale 提交当天的进展情况和问题，Pat 将确定他们是否像所期望的那样有进展，以及他们是否按照规格说明文档和项目管理计划行事。小组成员遇到的任何主要问题都将立即报告给 Pat。

5.4 **风险管理计划**。风险因素和跟踪机制如下所述：

这个新产品没有已存在的软件可以进行对比，因而，该产品不能与已存在的软件并行运行。因此，该产品应该进行广泛的测试。

客户被假设为对计算机不熟悉，因此在分析流和与客户交流时需要予以特别的注意。该产品应尽可能地做到对用户友好。

由于总是可能出现一个主要的设计错误，因此在设计流应进行广泛的测试。还有，每个小组成员先测试他或她自己的代码，然后测试其他成员的代码。Pat 负责集成测试和产品测试。

产品必须符合特定的存储要求和响应时间。由于产品的规模小，这不应是主要问题，但 Pat 必须在整个开发期间进行监控。

硬件故障发生的机会很小，在本例中将租用另一台机器。如果在编译器里有一个错误，应该替换掉这个编译器。这些均包含在供应硬件和编译器的厂商的质量保证中。

5.5 **项目停止计划**。这里不适用。

6 **技术过程计划**

6.1 **过程模型**。使用统一过程。

6.2 **方法、工具和技术**。该流程将依照统一过程进行。该产品将用 Java 实现。

6.3 **基础设施计划**。该产品将使用运行在个人电脑上的 Linux 下的 ArgoUML 进行开发。

6.4 **产品验收计划**。由客户进行的产品验收按照统一过程的步骤进行。

7 **支持过程计划**

7.1 **配置和管理计划**。对于所有模块将全程使用 CVS。

7.2 **测试计划**。执行统一过程的测试流。

7.3 **文档计划**。按照统一过程的规定生成文档。

7.4-5 **质量保证计划及评审和审计计划**。Robin 和 Dale 将互相测试代码，Pat 进行集成测试，然后三个人共同进行扩展的产品测试。

7.6 **问题解决计划**。如 5.3 节所述，小组成员面临的任何主要问题都将立即报告 Pat。

7.7 **次承包商管理计划**。这里不适用。

7.8 **过程改进计划**。所有活动都将按照公司在 2 年内从 CMM 2 级升到 3 级的计划进行。

8 **附加计划** 附加的部分包括：

安全性——使用该产品需要一个口令。

培训——交付时由 Pat 实施培训。因为该产品便于使用，1 天的时间用于培训足够了。Pat 将在使用的第一年内免费进行咨询。

维护——在 12 个月的时间里小组成员免费进行正确性维护。有关增强产品功能，另外签署一份单独的合同。

# 附录 G　设计流：Osbert Oglesby 实例研究

这个附录包含了 Osbert Oglesby 实例研究的类图的最后一次迭代（图 G-1）和 17 个组成类的 UML 图（按字母顺序），这些 UML 图显示了属性和方法。如 16.2 节所解释的，UML 可视化的前辍符号是：“-”代表 **private**，“+”代表 **public**，“#”代表 **protected**。

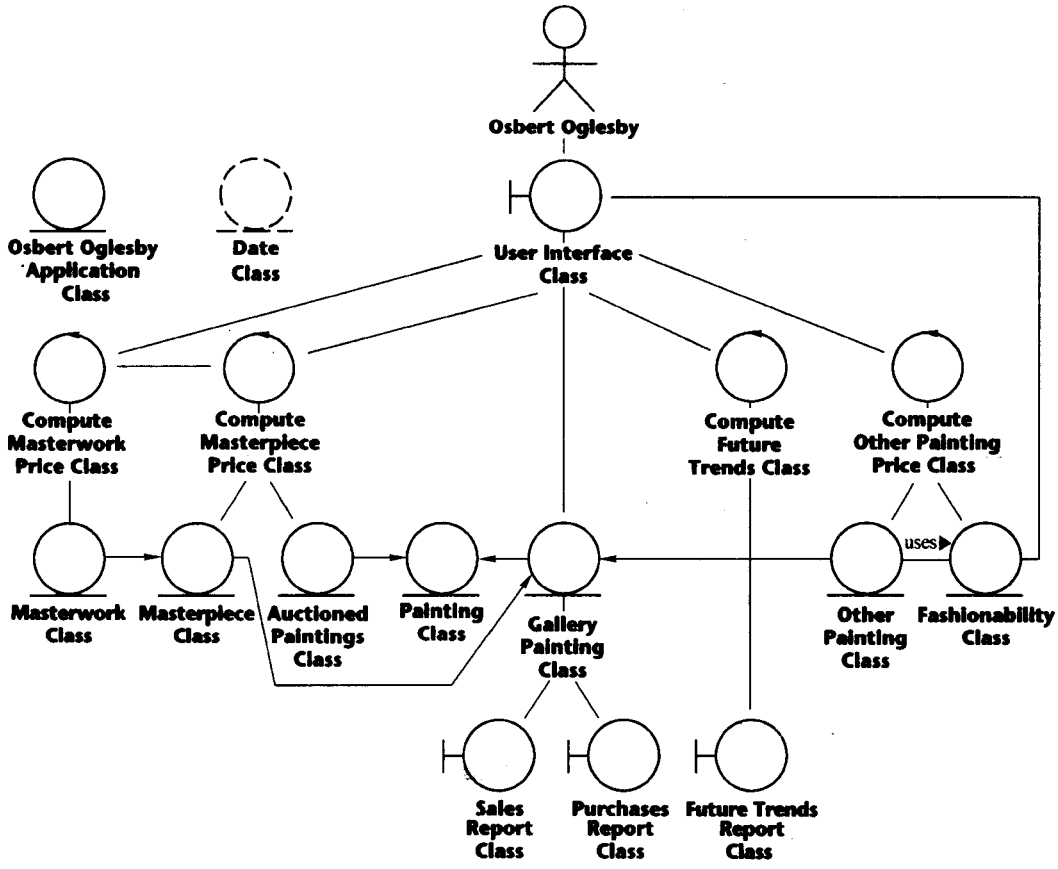
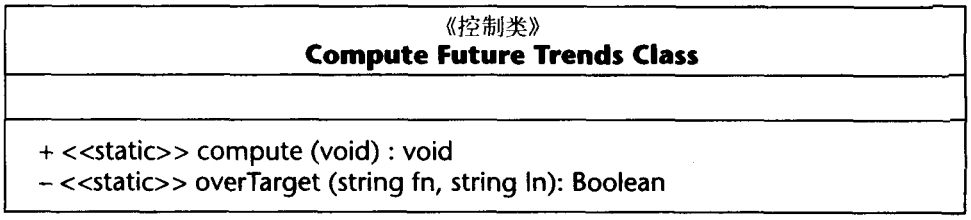
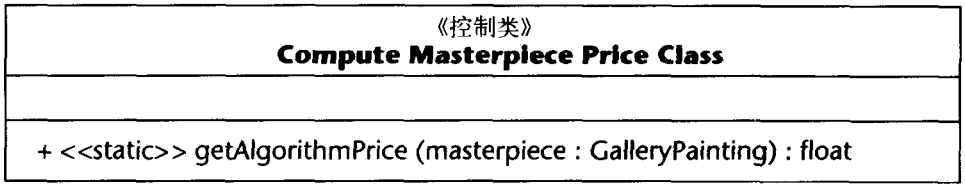
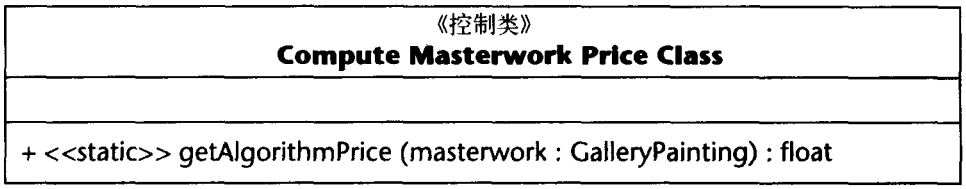


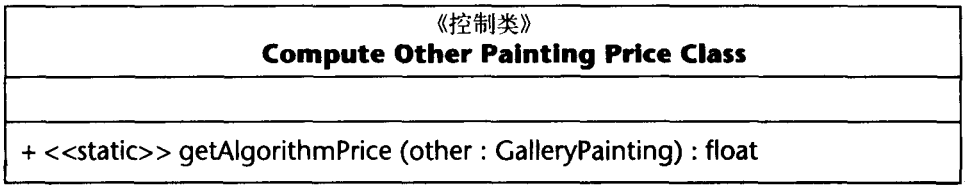

图 G-1　Osbert Oglesby 实例研究的最终类图（**Date Class** 为虚线表示这个类只在 C++ 实现中使用，在 Java 中，使用内部的日期类。）

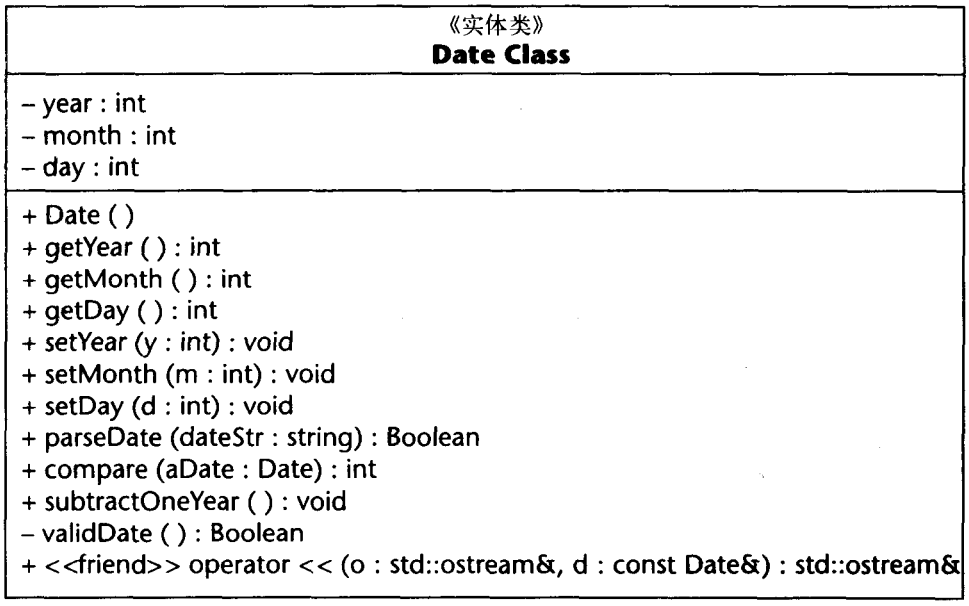
| 《实体类》<br>**Auctioned Paintings Class** |
| --- |
| # auctionPrice : float<br># auctionDate : Date |
| + getAuctionDate ( ) : Date<br>+ setAuctionDate (d : Date) : void<br>+ getAuctionPrice ( ) : float<br>+ setAuctionPrice (p : float) : void<br>+ readAuctionData (fileName : ifstream&) : void |


《控制类》
Compute Future Trends Class
+ <<static>> compute (void) : void
- <<static>> overTarget (string fn, string ln): Boolean


《控制类》
Compute Masterpiece Price Class
+ <<static>> getAlgorithmPrice (masterpiece : GalleryPainting) : float


《控制类》
Compute Masterwork Price Class
+ <<static>> getAlgorithmPrice (masterwork : GalleryPainting) : float


《控制类》
Compute Other Painting Price Class
+ <<static>> getAlgorithmPrice (other : GalleryPainting) : float


《实体类》
Date Class
- year : int
- month : int
- day : int
+ Date ( )
+ getYear ( ) : int
+ getMonth ( ) : int
+ getDay ( ) : int
+ setYear (y : int) : void
+ setMonth (m : int) : void
+ setDay (d : int) : void
+ parseDate (dateStr : string) : Boolean
+ compare (aDate : Date) : int
+ subtractOneYear ( ) : void
- validDate ( ) : Boolean
+ <<friend>> operator << (o : std::ostream&, d : const Date&) : std::ostream&

| 《实体类》<br>**Fashionability Class** |
| --- |
| # coefficient : float<br># firstName : string<br># lastName : string |
| + getFirstName ( ) : string<br>+ setFirstName (n : string) : void<br>+ getLastName ( ) : string<br>+ setLastName (n : string) : void<br>+ getCoefficient ( ) : float<br>+ setCoefficient (c : float) : void<br>+ getDescription ( ) : void<br>+ addNewFash ( ) : void<br>+ readFash (fileName : ifstream&) : void<br>+ writeFash (fileName : ofstream&) : void |

| 《实体类》<br>**Future Trends Report Class** |
| --- |
| |
| + <<static>> printReport (string tempFn, string tempLn) : void |

| 《实体类》<br>**Gallery Painting Class** |
| --- |
| # algPrice : float<br># purchasePrice : float<br># targetPrice : float<br># sellPrice : float<br># classification : string<br># purchaseDate : Date<br># saleDate : Date<br># sellerName : string<br># buyerName : string<br># sellerAddr : string<br># buyerAddr : string |
| + getClassification ( ) : string<br>+ setClassification (c : string) : void<br>+ getPurchaseDate ( ) : Date<br>+ setPurchaseDate (d : Date) : void<br>+ getSaleDate ( ) : Date<br>+ setSaleDate (d : Date) : void<br>+ getSellerName ( ) : string<br>+ setSellerName (n : string) : void<br>+ getBuyerName ( ) : string |

| |
|---|
| + setBuyerName (n : string) : void<br>+ getSellerAddr ( ) : string<br>+ setSellerAddr (a : string) : void<br>+ getBuyerAddr ( ) : string<br>+ setBuyerAddr (a : string) : void<br>+ getAlgPrice ( ) : float<br>+ setAlgPrice (p : float) : void<br>+ getPurchasePrice ( ) : float<br>+ setPurchasePrice (p : float) : void<br>+ getTargetPrice ( ) : float<br>+ setTargetPrice (p : float) : void<br>+ getSellPrice ( ) : float<br>+ setSellPrice (p : float) : void<br>+ getGalleryInformation ( : void) : void<br>+ addNewPainting ( : void) : void<br>+ buy ( : void) : void<br>+ readBought (fileName : ifstream&) : void<br>+ writeBought (fileName : ofstream&) : void<br>+ addNewSale ( : void) : void<br>+ sell ( : void) : void<br>+ readSold (fileName : ifstream&) : void<br>+ writeSold (fileName : ofstream&) : void |

| 《实体类》<br>**Masterwork Class** |
|---|
| |
| + Masterwork ( ) |

| 《实体类》<br>**Masterpiece Class** |
|---|
| |
| + Masterpiece ( ) |

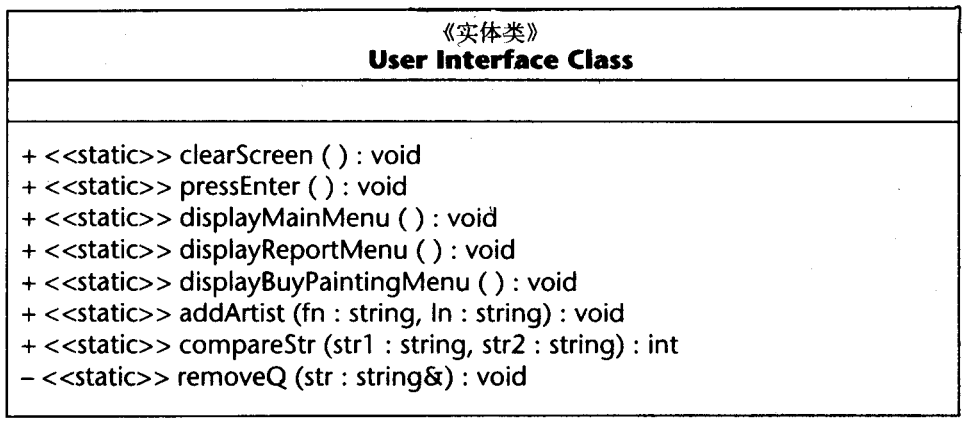
| 《实体类》<br>**Osbert Oglesby Application Class** |
|---|
| |
| main ( ) : void |

| 《实体类》<br>**Other Painting Class** |
|---|
| |
| + Other ( ) |

| 《实体类》<br>**Painting Class** |
| --- |
| # height : float<br># width : float<br># firstName : string<br># lastName : string<br># title : string<br># paintingDate : Date<br># saleDate : Date<br># medium : string<br># subject : string |
| + getFirstName ( ) : string<br>+ setFirstName (fn : string) : void<br>+ getLastName ( ) : string<br>+ setLastName (ln : string) : void<br>+ getTitle ( ) : string<br>+ setTitle (t : string) : void<br>+ getPaintDate ( ) : Date<br>+ setSaleDate (t : string) : Date<br>+ getSaleDate (d : Date) : void<br>+ getMedium ( ) : string<br>+ setMedium (m : string) : void<br>+ getSubject ( ) : string<br>+ setSubject (s : string) : void<br>+ getHeight ( ) : float<br>+ setHeight (h : float) : void<br>+ getWidth ( ) : float<br>+ setWidth (w : float) : void<br>+ getDescription ( ) : void |

| 《边界类》<br>**Purchases Report Class** |
| --- |
| |
| + <<static>> printReport ( ) : void |

| 《边界类》<br>**Sales Report Class** |
| --- |
| |
| + <<static>> printReport ( ) : void |


《实体类》
User Interface Class
+ <<static>> clearScreen ( ) : void
+ <<static>> pressEnter ( ) : void
+ <<static>> displayMainMenu ( ) : void
+ <<static>> displayReportMenu ( ) : void
+ <<static>> displayBuyPaintingMenu ( ) : void
+ <<static>> addArtist (fn : string, ln : string) : void
+ <<static>> compareStr (str1 : string, str2 : string) : int
– <<static>> removeQ (str : string&) : void

# 附录 H　实现流：Osbert Oglesby 实例研究（C++ 版本）

Osbert Oglesby 产品的完整 C++ 源代码可从 www.mhhe.com/engcs/compsci/schach 下载到。

# 附录 I　实现流：Osbert Oglesby 实例研究（Java 版本）

Osbert Oglesby 产品的完整 Java 源代码可从 www.mhhe.com/engcs/compsci/schach 下载到。

# 附录J　测试流：Osbert Oglesby实例研究

在下面四节中给出了Osbert Oglesby实例研究的测试流：

10.10节（需求）

12.17节（分析）

13.11节（设计）

14.23节（实现）