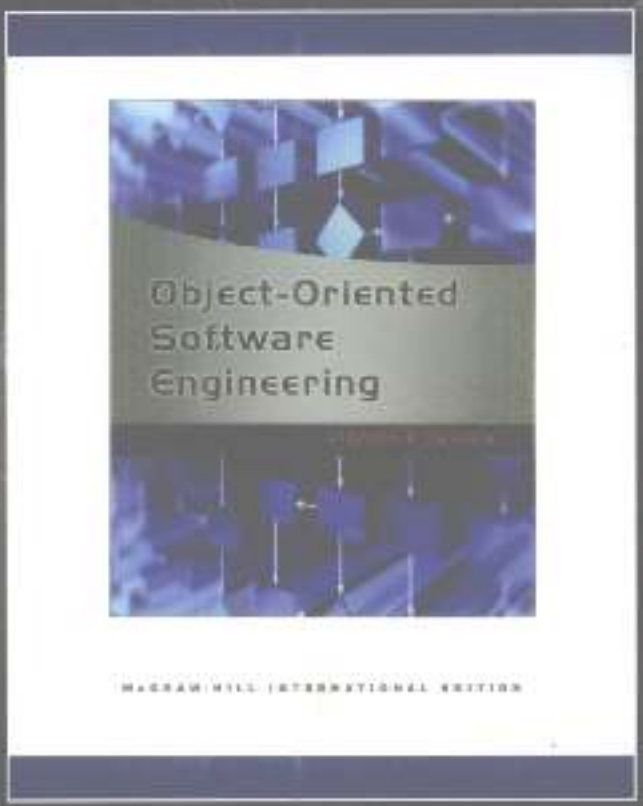
计 算 机 科 学 丛 书

面向对象软件工程

(美) Stephen R. Schach 范德比尔特大学 著
黄林鹏 徐小辉 伍建煜 上海交通大学 译



Object-Oriented Software Engineering

机械工业出版社
China Machine Press

面向对象软件工程

本书从面向对象范型出发对软件工程进行重新演绎，全面、系统、清晰地介绍了面向对象软件工程的基本概念、原理、方法和工具，通过实例说明了面向对象软件开发的整个过程。

本书分为两个部分：第一部分介绍了面向对象软件工程的基本理论；第二部分以工作流的形式介绍了软件生命周期。

本书特色

- 包括面向对象生命周期模型、面向对象分析、面向对象设计，以及面向对象软件的测试和维护。
- 讨论了文档、维护、复用、可修难性、测试和CASE工具等的重要性。
- 包括了能力成熟度模型（CMM）和人员能力成熟度模型（P-CMM）的内容。
- 与语言无关，实例代码对于C++和Java语言背景的读者同样清晰。
- 包括600余篇当前热点研究文章、经典文献和书籍的参考文献。
- 包含2个用于说明完整软件生命周期的运行实例，还有7个较小的实例，分别用于突出说明特定的主题。基于统一过程、Java和C++语言的完整源码可从作者网站（www.mhhe.com/schach）下载。
- 包括5种类型的习题：分别是概念理解、项目分析、课程设计、论文研读和实例修改。

作者简介

Stephen R. Schach



1972年获得兹曼科学院理科硕士学位，1973年获得宾夕法尼亚大学应用数学博士学位。目前任教于美国范德比尔特大学计算机科学系。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。他还在国际上广泛讲授软件工程方面的课程，包括复用、CASE和面向对象范型等。



软件工程：面向对象和传统的方法（原书第7版）
书号：978-7-111-21722-0
定价：48.00元

Mc Graw Hill Education

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

ISBN 978-7-111-25502-4

9 787111 255024

定价：48.00元

计 算 机 科 学 丛 书

面向对象软件工程

(美) Stephen R. Schach 范德比尔特大学 著 黄林鹏 徐小辉 伍建煜 上海交通大学 译



Object-Oriented Software Engineering

机械工业出版社
China Machine Press

本书从面向对象范型出发对软件工程进行重新演绎，全面、系统、清晰地介绍了面向对象软件工程的基本概念、原理、方法和工具，通过实例说明面向对象软件开发的整个过程。

本书分为两个部分：第一部分介绍面向对象软件工程的基本理论；第二部分以工作流的形式介绍软件生命周期。

本书可以作为计算机相关专业高年级本科生和研究生的教材，也可以作为软件工程领域专业人士的参考书。

Stephen R. Schach: Object-Oriented Software Engineering.

Original language copyright © 2009 by The McGraw-Hill Companies, Inc.

All rights reserved.

Simplified Chinese translation edition published by China Machine Press.

本书中文简体字版由美国麦格劳—希尔教育出版公司授权机械工业出版社出版，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：01-2008-1763

图书在版编目(CIP)数据

面向对象软件工程／(美) 沙赫查 (Schach, S. R.) 著；黄林鹏，徐小辉，伍建焜译. —北京：机械工业出版社，2008.12

(计算机科学丛书)

书名原文：Object-Oriented Software Engineering

ISBN 978-7-111-25502-4

Ⅰ. 面… Ⅱ. ①沙… ②黄… ③徐… ④伍… Ⅲ. 面向对象语言－程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字 (2008) 第 171861 号

机械工业出版社 (北京市西城区百万庄大街 22 号　邮政编码　100037)

责任编辑：王春华

北京诚信伟业印刷有限公司印刷

2009 年 2 月第 1 版第 1 次印刷

184mm×260mm · 23 印张

书号：ISBN 978-7-111-25502-4

定价：48.00 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绝对性的优势，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，不仅奠划了研究的范畴，还揭示了学术的源变，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和发展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到积极的推动作用，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分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Alfred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson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、研究及珍藏。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”已经出版了近两百个品种，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。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

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。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐步深化，教育界对外计算机教材的需求和应用都将步入一个新的阶段，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。华章分社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：

本章网址：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

华章教育

译者序

软件工程的目标是按时交付满足客户需要、未超出预算、无错误、能随需求变化易于修改的软件。

在计算机界，范型一词最早用于描述编程风格。编程范型可以看成是程序员对程序执行的看法，而一些语言是专门为某个特定的范型设计的，当然也有一些语言支持多种范型。由于编程语言和软件开发的密切关系，范型一词也被引申至软件工程领域。面向对象的软件工程（object-oriented software engineering）就是一门利用面向对象范型实现软件工程目标的学科。

本书的作者 Stephen R. Schach 博士编写了 14 本与软件工程相关的畅销书，他撰写的书籍深入浅出，被许多学校选为教材并翻译为多国文字。Stephen R. Schach 认为，当前传统范型的使用在很大程度上仅限于遗留系统的维护。学生所学的第一门面向对象程序语言是C++ 或 Java，他们毕业后一般也将工作于一个使用面向对象范型的公司，因此有必要从面向对象范型出发对软件工程进行重新演绎，本书正是这样一本教科书。它全面、系统、清晰地介绍了面向对象软件工程的基本概念、原理、方法和工具，并通过实例说明了面向对象软件开发的整个过程。

本书分为两个部分，第一部分介绍了面向对象软件工程的基本理论，第二部分以工作流的形式介绍了软件生命周期，并通过一个小规模和一个中等规模的软件开发实例进行阐述。按照作者建议的课程内容安排，本书既可作为一个学期也可作为两个学期的教科书使用。

本书的习题也很有特色，可分为 5 类：一是每章都包含的与知识点相关的练习；二是每一章都选择了一篇经典论文，要求学生阅读并对相关的问题展开讨论，此类题目对于研究性的学习或讨论班类的课程特别有助；三是针对需求、分析和设计工作流设计的面向对象的分析和设计的小项目，通过实践，学生可快速掌握相关的工具和技术；四是针对每章讨论的实例，要求学生按照需求变化对实例进行某种修改，修改一个现成的产品与从头开始开发一个产品相比，有时前者对于知识的掌握更加有效；五是 project，其是为 3 个人组成的团队设计的，目的是锻炼团队的协同软件开发能力。

本书的翻译工作主要由黄林鹏负责，参与本书翻译的还有徐小辉、王欣、陈俊清、任建焜、王德俊、孙俊、沈飞、徐成、黄冠、曾慧清和杜思奇等。其中第 2 章至第 4 章由徐小辉初译，第 5 章至第 7 章由王欣初译，第 8 章至第 11 章由陈俊清初译，第 12 章至第 15 章由伍建焜初译。陆朝俊副教授对本书的翻译提供了不少建设性的帮助。本书译稿由黄林鹏修改、整理和定稿，其对最终出现的问题负责，请将批评意见发至 lphuang@ sjtu. edu. cn，不胜感激。

黄林鹏

2008 年 11 月于上海交通大学

前 言

在1988年，我撰写了一本教科书名为《Software Engineering》。事实上，在那本书中唯一提到面向对象范型的只有一节，即面向对象设计。

直到1994年，面向对象范型开始得到软件业界的认同，因此，我撰写了《Classical and Object-Oriented Software Engineering》一书。6年后，面向对象范型变得比传统范型更加重要。为了反映这个变化，我在2000年撰写的《Object-Oriented and Classical Software Engineering》（软件工程：面向对象和传统的方法）[⊖]中更换了这两个主题的顺序。

今天，传统范型的使用仅限于对遗留系统的维护。学生所学的第一门面向对象的程序语言是C++或Java，而面向对象语言在后续的计算机科学和计算机工程课程中也常被使用。学生们期望，他们毕业后在一个使用面向对象范型的公司工作。面向对象范型已经完全挤压了传统范型的生存空间，这也是本书命名为《面向对象软件工程》的原因。

本书特点

- 统一过程仍是面向对象软件开发的首选，因此本书中，学生所学习的仍是统一过程的理论和实践。
- 第1章深入分析了面向对象范型的优势。
- 第2章引入了迭代-递增生命周期模型。此外，本章还讨论了敏捷过程。
- 第3章介绍了统一过程的各个阶段和工作流（活动），解释了为何需要二维生命周期模型。
- 第4章讨论了软件团队的多种组织方式，包括敏捷过程团队和开源软件开发团队。
- 第5章介绍了一些重要的CASE工具。
- 第6章强调了连续测试的重要性。
- 第7章关注的重点是对象。
- 第8章强调了设计模式。
- 第9章给出了软件项目管理计划的新IEEE标准。
- 第10章、第11章、第12章和第13章主要阐述统一过程中的工作流（活动）。
- 第13章清晰地阐述了实现和集成的区别。
- 第14章强调了交付后维护的重要性。
- 第15章提供了更多的关于UML的资料。这一章对于采用本书作为两学期的软件工程课程教材的教师尤其有用。在第二学期，除了开发基于团队的学期项目或者“封顶”项目外，通过学习本章，学生可以获得更多的UML知识。
- 本书包含两个使用统一过程的运行实例：MSG基金会实例和电梯问题。其Java和C++语言的开发源码可从www.mhhe.com/schach下载。
- 除了两个用于说明完整软件生命周期的运行实例外，还有7个较小的实例，分别用于突出说明特定的主题，如移动目标、逐步求精和交付后维护等。
- 本书强调文档、维护、重用、可移植性、测试和CASE工具。学生只有认识到这些面向对象软件工程的基础的重要性，才能更好地掌握最新的技术。

⊖ 该书的影印版和中文版，已由机械工业出版社出版，书号分别为ISBN 978-7-111-20822-8和ISBN 978-7-111-21722-0。

- 本书注重面向对象生命周期模型、面向对象分析、面向对象设计、面向对象范型的管理建议、面向对象软件的测试和维护，还包括了面向对象范型的度量。除此之外，许多地方或多或少地会提及对象，因为面向对象范型不仅与执行不同的工作流相关，而且已经渗透到我们对软件工程的思考之中。对象技术的使用遍及全书。
- 软件过程仍是本书的整体基础。为了控制这个过程，必须能够度量项目中所发生的一切。因此，有必要强调度量。对于过程改进，本书包括了能力成熟度模型（CMM）、ISO/IEC 15504（SPICE）和 ISO/IEC 12207。在第 4 章中包括了人员能力成熟度模型（P-CMM）的内容。
- 本书与语言无关。虽然书中少量代码是以C++ 或 Java 语言编写的，但我尽量消除与语言相关的细节，以确保代码例子对于C++ 或 Java 语言背景的读者同样清晰。例如，对于输出，我不使用C++ 语言的 cout，也不使用 Java 语言的 System. out. println，而是使用伪代码指令 print（唯一的例外是第二个研究实例，其完整实现分别以C++ 和 Java 语言给出）。
- 本书包括 600 余条参考文献条目，其中有当前的研究文章、一些内容仍新鲜且相关的经典文献以及书籍。毫无疑问，面向对象软件工程是一门快速发展的学科，学生需要了解最新的研究成果以及在哪里可以找到它们。同时，今天的前沿研究是在昨天的成果之上进行的，如果一篇文章的思想今天仍然实用，就没有理由将其排除在参考文献之外。
- 对于先修课程，假定读者熟悉一种高级面向对象程序设计语言（如C++ 或 Java 等），除此之外，希望读者学习过数据结构课程。

本书的结构

本书既可为传统的一个学期的软件工程课程使用，也同样适用于目前普遍流行的较为新颖的两个学期软件工程课程。在传统的一个学期（或者四分之一学年）的课程中，理论部分内容不得不一掠而过，这样，教师才有时间给学生讲授完成学期项目所需要的知识和技能。如此匆忙的目的是使学生能尽早开始项目并在学期结束前完成项目。为了满足一个学期的、基于项目的软件工程课程教学的需要，本书第二部分以一个工作流接着一个工作流的形式介绍了软件生命周期，而第一部分则提供了为理解第二部分所需要的理论基础。例如，第一部分介绍了CASE、度量和测试，而第二部分的每一章都包含一节讲授工作流的 CASE 工具、一节讲授工作流的度量和一节介绍工作流期间的测试。为了尽早讲授第二部分内容，第一部分的教学内容应保持简短。另外，第一部分最后两章（第 8 章和第 9 章）的内容的教学可以推迟，留待和第二部分的内容一起并行讲授。由此，可以尽快地开始学期项目。

现在来看一下两个学期的软件工程课程的教学情况。越来越多的计算机科学系和计算机工程系的教师认识到大部分毕业生受聘的职位是软件工程师，因此，许多大学都开设两个学期（或两个四分之一学年）的软件工程系列课程。第一学期的课程主要是理论性的（但通常也包括某种形式的小型项目），而第二学期的课程则是以基于团队的学期项目开发为主。学期项目通常是课程的最后一部分内容，因此，如果学期项目是第二学期的内容，教师的授课时间就不会显得捉襟见肘。

综上所述，使用本书作为一个学期（或四分之一学年）课程的教科书的教师，可首先讲授本书第 1 章至第 7 章的大部分内容，然后开始讲授第二部分（第 10 章至第 15 章）的内容。第 8 章和第 9 章的内容可以和第二部分的内容平行教授或在学期末（学生完成项目）时讲授。当本书作为两个学期使用的教材时，应当按顺序讲授各章节的内容。第一学期课程结束之后，学生就应该做好参与第二学期基于团队的学期项目的准备。

为了确保学生能够真正理解第二部分所介绍的一些关键软件工程技术，本书对每项技术都

介绍了两次。首先，通过电梯问题引入该技术并进行讲解。电梯问题大小适中，读者借此可以看到技术在一个完整问题上的应用，而且还包括了足够多的细节，可以凸显所教技术的优缺点。然后，给出 MSG 基金会案例中与该项技术相关的部分，借助案例的详细解决方案展开对应技术的第二次阐述。

习题类型

本书有 5 种类型的习题。第一，在第 10 章、第 11 章和第 12 章末尾都有连续的面向对象的分析和设计项目。因为只有通过实践才能掌握如何执行需求、分析和设计工作流。

第二，每一章的末尾都包含意在突出知识点的习题。这些习题是独立的，所有相关的技术信息都可以在本书中找到。

第三，有一个学期项目。这个项目是为 3 个人组成的团队设计的，3 是最小的不用通过标准电话交换意见的团队成员的数目。学期项目由 14 个独立的组件组成，每一组件都位于某一相关章节之后。例如，第 12 章中，学期项目的组件所涉及的就是软件设计。通过将一个大的项目分解为较小的、明确定义的若干部分，指导教师就能更密切地掌控学生的学习进度。学期项目的构造目的是使指导教师可以自由地将这 14 个组件应用于其他所选择的项目。

本书是为研究生和本科高年级学生编写的，因此，第 4 种类型的习题是基于软件工程领域的研究论文而设计的。每一章都选择一篇重要的论文。要求学生阅读文章并回答与其内容相关的问题。当然，教师也可自由选择其他的研究论文，每一章的“延伸阅读材料”中包含了多篇相关的论文。

第 5 种类型的习题和实例研究相关。这种类型的题目是应许多授课教师的要求加入的。许多教师感到学生通过修改一个现成的产品比从头开始开发一个产品所学的要多。许多业界的资深工程师也同意这个观点。由此，在每章给出实例研究之处都有需要学生对实例进行某种修改的问题。例如，其中一章就要求学生回答如下问题：若以不同的顺序执行面向对象分析的步骤，效果如何。为了使学生便于修改实例，在 www. mhhe. com/schach 上可获得实例的源代码。

该网站也给授课教师提供了 PowerPoint 形式的授课笔记以及包含学期项目在内的所有习题的详细解答。

UML 材料

本书大量使用统一建模语言（UML）。如果学生以前没有学过 UML，该方面的内容可以两种方式讲授。我倾向于在需要时才进行教授，即每个 UML 概念仅在需要时才介绍。下表给出了本书使用的 UML 结构所对应的章节。

结　构	介绍对应 UML 图的章节
类图、注解、继承（泛化）、聚合、关联、导航三角形	7.7 节
用例	10.4.3 节
用例图、用例描述	10.7 节
固定形式	11.4 节
状态图	11.9 节
交互图（顺序图、协作图）	11.18 节

另外一种讲授方式是整体教学。本书第 15 章介绍了 UML，包含上述概念以及以后学习所需的材料。第 15 章可以随时讲授，这章的内容不依赖于前面 14 章。第 15 章包含的主题见下表：

结　构	介绍对应 UML 图的小节
类图、聚合、多态、结合、泛化、关联	15.2 节
注解	15.3 节
用例图	15.4 节
固定形式	15.5 节
交互图	15.6 节
状态图	15.7 节
活动图	15.8 节
包	15.9 节
组件图	15.10 节
配置图	15.11 节

致谢

感谢本书的评审，他们是

Michael A. Aars（Baylor University）
Keith S. Decker（University of Delaware）
Xiaocong Fan（The Pennsylvania State University）
Adrian Fiech（Memorial University）
Sudipto Ghosh（Colorado State University）
David C. Rine（George Mason University）
Keng Siau（University of Nebraska-Lincoln）
Anita Kuchera（Rensselaer Polytechnic Institute）
Matthew R. McBride（Southern Methodist University）
Michael McCracken（Georgia Institute of Technology）
Rick Mercer（University of Arizona）
Richard J. Povinelli（Marquette University）
John Sturman（Rensselaer Polytechnic Institute）
Levent Yilmaz（Auburn University）

衷心地感谢对本书以及我先前书籍做出重要贡献的三位同仁，其一是 Kris Irwin，他提供了用 Java 和C++ 语言实现的学期项目的完整解答；其二是 Jeff Gray，他给出了 MSG 基金会案例的实现；其三是 Lauren Ryder，他和我一起完成了本书的教师指导手册和 PowerPoint 幻灯片。

接着要感谢的是出版商 McGraw-Hill。真诚地感谢资深出版编辑 Faye Schilling，她在出版中期主动承担了产品经理的工作，对于她在需要时能欣然应允修改日程安排表示感激。策划编辑 Lora Kalb，她自始至终都是本书出版的顶梁柱，非常高兴能再次和她一起工作。衷心感谢本书排版 Lucy Mullins、校对 Dorothy Wendell 和产品经理 Joyce Berendes。最后，感谢 Brenda Rolwes，在他的协调下，Studio Montage 公司的封面设计师 Jenny Hobein 将位于悉尼海港的桥梁变形为能在读者脑海产生深刻印象的封面图案。

感谢世界各地的教师，他们发来了关于我的其他书籍的意见。我期待着收到他们针对本书发来的反馈意见。我的 E-mail 地址是 srs@ vuse. vanderbilt. edu。

学生们对本书的出版也做出了巨大的贡献。再次感谢 Vanderbilt 大学的学生提出的来自课堂内外的挑战性问题和建设性意见。非常感谢来自世界各地的学生通过 E-mail 发给我的问题和建议。我真诚期望学生如对待我以前的书籍一样发来本书的反馈意见。

最后，感谢我的家人对我持之以恒的支持。和编写先前的书籍一样，我尽量确保家庭义务先于书籍写作。然而，当交稿临近时，有时这是不可能的。此时，他们总是能表示理解，为此我深深地感谢他们。

让我将这本我所撰写的第 14 本书籍和我的爱一起献给我的孙子 Jackson。

Stephen R. Schach

目 录

出版者的话
译者序
前言

第一部分 面向对象软件工程简介

第 1 章 面向对象软件工程的范畴 …… 3
1.1 历史方面 …… 4
1.2 经济方面 …… 6
1.3 维护方面 …… 6
1.3.1 现代软件维护观点 …… 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 …… 10
1.5 团队开发 …… 11
1.6 没有计划阶段的原因 …… 12
1.7 没有测试阶段的原因 …… 12
1.8 没有文档阶段的原因 …… 13
1.9 面向对象范型 …… 13
1.10 术语 …… 15
1.11 道德规范问题 …… 17
本章回顾 …… 18
延伸阅读材料 …… 18
习题 …… 19
参考文献 …… 20
第 2 章 软件生命周期模型 …… 23
2.1 理想软件开发 …… 23
2.2 Winburg 小型案例研究 …… 23
2.3 Winburg 小型案例研究经验 …… 25
2.4 Teal Tractors 公司小型案例研究 …… 25
2.5 迭代与增量 …… 26
2.6 Winburg 小型案例研究再探 …… 28
2.7 迭代和增量的风险及其他 …… 29
2.8 管理迭代与增量 …… 31
2.9 其他生命周期模型 …… 31
2.9.1 边写边改生命周期模型 …… 32
2.9.2 瀑布生命周期模型 …… 32
2.9.3 快速原型生命周期模型 …… 33
2.9.4 开源生命周期模型 …… 34
2.9.5 敏捷过程 …… 35
2.9.6 同步稳定生命周期模型 …… 37
2.9.7 螺旋生命周期模型 …… 38
2.10 生命周期模型的比较 …… 40
本章回顾 …… 41
延伸阅读材料 …… 41
习题 …… 42
参考文献 …… 43
第 3 章 软件过程 …… 46
3.1 统一过程 …… 47
3.2 迭代与增量 …… 48
3.3 需求工作流 …… 49
3.4 分析工作流 …… 50
3.5 设计工作流 …… 51
3.6 实现工作流 …… 52
3.7 测试工作流 …… 52
3.7.1 需求制品 …… 53
3.7.2 分析制品 …… 53
3.7.3 设计制品 …… 53
3.7.4 实现制品 …… 53
3.8 交付后维护 …… 54
3.9 退役 …… 55
3.10 统一过程的阶段 …… 55
3.10.1 初始阶段 …… 56
3.10.2 细化阶段 …… 57
3.10.3 构造阶段 …… 58
3.10.4 移交阶段 …… 58
3.11 一维与二维生命周期模型对比 …… 59
3.12 改进软件过程 …… 60
3.13 能力成熟度模型 …… 60
3.14 软件过程改进的其他方面 …… 62
3.15 软件过程改进的成本与收益 …… 62
本章回顾 …… 64
延伸阅读材料 …… 64
习题 …… 65
参考文献 …… 65
第 4 章 软件团队 …… 68
4.1 团队组织 …… 68
4.2 民主团队方式 …… 69
4.3 主程序员团队方式 …… 70
4.3.1 《纽约时报》项目 …… 71
4.3.2 主程序员团队方式的不切实际性 … 72
4.4 超越主程序员和民主团队 …… 72
4.5 同步 - 稳定团队 …… 73

4.6 敏捷过程团队 …… 74
4.7 开源编程团队 …… 74
4.8 人力资源能力成熟度模型 …… 75
4.9 选择合适的团队组织 …… 75
本章回顾 …… 76
延伸阅读材料 …… 76
习题 …… 77
参考文献 …… 77
第5章 软件工程工具 …… 79
5.1 逐步求精 …… 79
5.2 成本－效益分析法 …… 82
5.3 软件度量 …… 83
5.4 CASE …… 84
5.5 CASE 的分类 …… 85
5.6 CASE 的范围 …… 86
5.7 软件版本 …… 88
5.7.1 修订版 …… 89
5.7.2 变体 …… 89
5.8 配置控制 …… 89
5.8.1 交付后维护期间的配置控制 …… 91
5.8.2 基线 …… 91
5.8.3 产品开发过程中的配置控制 …… 91
5.9 建造工具 …… 92
5.10 使用 CASE 技术提高生产力 …… 93
本章回顾 …… 93
延伸阅读材料 …… 93
习题 …… 94
参考文献 …… 95
第6章 测试 …… 97
6.1 质量问题 …… 97
6.1.1 软件质量保证 …… 98
6.1.2 管理独立性 …… 98
6.2 基于非执行的测试 …… 99
6.2.1 走查 …… 99
6.2.2 管理走查 …… 100
6.2.3 审查 …… 100
6.2.4 走查和审查的对比 …… 102
6.2.5 评审的优缺点 …… 102
6.2.6 审查的度量方法 …… 102
6.3 基于执行的测试 …… 103
6.4 应该测试什么 …… 103
6.4.1 实用性 …… 104
6.4.2 可靠性 …… 104
6.4.3 健壮性 …… 104
6.4.4 性能 …… 105
6.4.5 正确性 …… 105
6.5 测试与正确性证明 …… 106
6.5.1 正确性证明的例子 …… 106
6.5.2 正确性证明小型实例研究 …… 108
6.5.3 正确性证明和软件工程 …… 109
6.6 由谁来完成基于执行的测试 …… 111
6.7 测试何时停止 …… 112
本章回顾 …… 112
延伸阅读材料 …… 112
习题 …… 113
参考文献 …… 114
第7章 从模块到对象 …… 117
7.1 什么是模块 …… 117
7.2 内聚 …… 119
7.2.1 偶然性内聚 …… 119
7.2.2 逻辑性内聚 …… 120
7.2.3 时间性内聚 …… 120
7.2.4 过程性内聚 …… 121
7.2.5 通信性内聚 …… 121
7.2.6 功能性内聚 …… 121
7.2.7 信息性内聚 …… 121
7.2.8 内聚示例 …… 122
7.3 耦合 …… 122
7.3.1 内容耦合 …… 122
7.3.2 公共耦合 …… 123
7.3.3 控制耦合 …… 124
7.3.4 印记耦合 …… 125
7.3.5 数据耦合 …… 125
7.3.6 耦合示例 …… 126
7.3.7 耦合的重要性 …… 126
7.4 数据封装 …… 127
7.4.1 数据封装和开发 …… 128
7.4.2 数据封装和维护 …… 129
7.5 抽象数据类型 …… 133
7.6 信息隐藏 …… 134
7.7 对象 …… 135
7.8 继承、多态和动态绑定 …… 137
7.9 面向对象范型 …… 139
本章回顾 …… 140
延伸阅读材料 …… 141
习题 …… 141
参考文献 …… 142
第8章 可复用性和可移植性 …… 144
8.1 复用的概念 …… 145
8.2 复用的障碍 …… 146
8.3 复用案例研究 …… 147
8.3.1 雷锡恩导弹系统部门 …… 147
8.3.2 欧洲航天局 …… 148
8.4 对象和复用 …… 149

8.5 在设计和实现过程中的复用 …… 149
8.5.1 设计复用 …… 149
8.5.2 应用架构 …… 150
8.5.3 设计模式 …… 151
8.5.4 软件体系结构 …… 152
8.5.5 基于组件的软件工程 …… 153
8.6 关于设计模式的更多内容 …… 153
8.6.1 FLIC 小型案例研究 …… 153
8.6.2 适配器设计模式 …… 154
8.6.3 桥接设计模式 …… 154
8.6.4 迭代器设计模式 …… 155
8.6.5 抽象工厂设计模式 …… 156
8.7 设计模式的范畴 …… 159
8.8 设计模式的优点和缺点 …… 159
8.9 复用和交付后的维护 …… 160
8.10 可移植性 …… 161
8.10.1 硬件的不兼容性 …… 161
8.10.2 操作系统的不兼容性 …… 162
8.10.3 数值计算软件的不兼容性 …… 162
8.10.4 编译器的不兼容性 …… 163
8.11 为什么需要可移植性 …… 165
8.12 实现可移植性的技术 …… 166
8.12.1 可移植的系统软件 …… 166
8.12.2 可移植的应用软件 …… 166
8.12.3 可移植数据 …… 167
8.12.4 基于 Web 的应用程序 …… 167
本章回顾 …… 168
延伸阅读材料 …… 168
习题 …… 169
参考文献 …… 170
第 9 章 计划与估算 …… 174
9.1 计划活动与软件过程 …… 174
9.2 估算项目周期和成本 …… 175
9.2.1 产品规模的衡量标准 …… 176
9.2.2 成本估算技术 …… 178
9.2.3 中级 COCOMO …… 180
9.2.4 COCOMO II …… 182
9.2.5 跟踪周期和成本估算 …… 183
9.3 估算探讨 …… 183
9.4 软件项目管理计划的组成 …… 184
9.5 软件项目管理计划框架 …… 185
9.6 IEEE 软件项目管理计划 …… 186
9.7 对测试进行计划 …… 188
9.8 培训需求 …… 188
9.9 文档标准 …… 189
9.10 计划和估算的 CASE 工具 …… 189
9.11 测试软件项目管理计划 …… 190
本章回顾 …… 190
延伸阅读材料 …… 190
习题 …… 191
参考文献 …… 192

第二部分 软件生命周期工作流

第 10 章 需求工作流 …… 196
10.1 确定什么是客户所需 …… 196
10.2 需求工作流概述 …… 197
10.3 域的理解 …… 197
10.4 业务模型 …… 198
10.4.1 访谈 …… 198
10.4.2 其他技术 …… 198
10.4.3 用例 …… 199
10.5 初始需求 …… 200
10.6 对应用域的初始理解：MSG 基金会实例研究 …… 200
10.7 初始业务模型：MSG 基金会实例研究 …… 202
10.8 初始需求：MSG 基金会实例研究 … 204
10.9 需求工作流继续：MSG 基金会实例研究 …… 205
10.10 修订需求：MSG 基金会实例研究 …… 206
10.11 测试工作流：MSG 基金会实例研究 …… 211
10.12 什么是面向对象的需求 …… 217
10.13 快速原型 …… 218
10.14 人为因素 …… 218
10.15 复用快速原型 …… 219
10.16 需求工作流的 CASE 工具 …… 220
10.17 需求工作流的度量 …… 220
10.18 需求工作流的挑战 …… 220
本章回顾 …… 221
延伸阅读材料 …… 222
习题 …… 222
参考文献 …… 223
第 11 章 分析工作流 …… 224
11.1 规格说明文档 …… 224
11.2 非形式化规格说明 …… 225
11.3 小型案例研究的正确性证明回顾 …… 226
11.4 分析工作流 …… 227
11.5 实体类的提取 …… 228
11.6 电梯问题 …… 228
11.7 功能建模：电梯问题案例研究 …… 229
11.8 实体类建模：电梯问题案例研究 …… 230
11.8.1 名词提取 …… 230

11.8.2 CRC 卡片 …… 232
11.9 动态建模：电梯问题案例研究 …… 233
11.10 测试工作流：电梯问题案例研究 … 235
11.11 提取边界类和控制类 …… 237
11.12 初始功能建模：MSG 基金会案例研究 …… 238
11.13 初始类图：MSG 基金会案例研究 …… 239
11.14 初始动态建模：MSG 基金会案例研究 …… 240
11.15 修订实体类：MSG 基金会案例研究 …… 242
11.16 提取边界类：MSG 基金会案例研究 …… 243
11.17 提取控制类：MSG 基金会案例研究 …… 243
11.18 用例实现：MSG 基金会案例研究 …… 243
11.18.1 Estimate Funds Available for Week 用例 …… 244
11.18.2 Manage an Asset 用例 …… 248
11.18.3 Update Estimated Annual Operating Expenses 用例 …… 251
11.18.4 Produce a Report 用例 …… 252
11.19 类图增量：MSG 基金会案例研究 …… 256
11.20 软件项目管理计划：MSG 基金会案例研究 …… 257
11.21 测试工作流：MSG 基金会案例研究 …… 257
11.22 统一过程中的规格说明文档 …… 257
11.23 更多关于参与者和用例的内容 …… 258
11.24 支持分析工作流的 CASE 工具 …… 259
11.25 分析工作流的挑战 …… 259
本章回顾 …… 259
延伸阅读材料 …… 260
习题 …… 260
参考文献 …… 262
第 12 章 设计工作流 …… 264
12.1 面向对象设计 …… 264
12.2 面向对象设计：电梯问题案例研究 …… 268
12.3 面向对象设计：MSG 基金会案例研究 …… 270
12.4 设计工作流 …… 272
12.5 测试工作流：设计 …… 273
12.6 测试工作流：MSG 基金会案例 …… 273
12.7 详细设计的形式化技术 …… 273
12.8 实时设计技术 …… 274
12.9 用于设计的 CASE 工具 …… 274
12.10 设计的度量 …… 275
12.11 设计工作流面临的挑战 …… 276
本章回顾 …… 277
延伸阅读材料 …… 277
习题 …… 277
参考文献 …… 278
第 13 章 实现工作流 …… 280
13.1 选择编程语言 …… 280
13.2 良好的编程实践 …… 282
13.2.1 使用一致和有意义的变量名 …… 282
13.2.2 自文档化代码的问题 …… 283
13.2.3 使用参数 …… 284
13.2.4 为增加可读性的代码编排 …… 284
13.2.5 嵌套的 if 语句 …… 285
13.3 编码标准 …… 286
13.4 代码复用 …… 286
13.5 集成 …… 286
13.5.1 自顶向下的集成 …… 287
13.5.2 自底向上的集成 …… 288
13.5.3 三明治集成 …… 288
13.5.4 集成技术 …… 289
13.5.5 集成管理 …… 290
13.6 实现工作流 …… 290
13.7 实现工作流：MSG 基金会案例研究 …… 290
13.8 测试工作流：实现 …… 290
13.9 测试用例的选择 …… 290
13.9.1 规格说明测试与代码测试 …… 291
13.9.2 规格说明测试的可行性 …… 291
13.9.3 代码测试的可行性 …… 291
13.10 黑盒单元测试技术 …… 293
13.10.1 等价测试和边界值分析 …… 293
13.10.2 功能测试 …… 294
13.11 黑盒测试用例：MSG 基金会案例研究 …… 294
13.12 玻璃盒单元测试技术 …… 296
13.12.1 结构测试：语句、分支和路径覆盖 …… 296
13.12.2 复杂性度量 …… 297
13.13 代码走查和审查 …… 298
13.14 单元测试技术的比较 …… 298

13.15 净室 …… 298
13.16 测试中的问题 …… 299
13.17 单元测试的管理方面内容 …… 301
13.18 何时重写而不是调试代码制品 …… 301
13.19 集成测试 …… 302
13.20 产品测试 …… 303
13.21 验收测试 …… 303
13.22 测试流：MSG 基金会案例研究 …… 304
13.23 用于实现的 CASE 工具 …… 304
13.23.1 软件开发全过程的 CASE 工具 …… 304
13.23.2 集成开发环境 …… 304
13.23.3 商业应用环境 …… 305
13.23.4 公共工具基础结构 …… 305
13.23.5 环境的潜在问题 …… 306
13.24 测试工作流的 CASE 工具 …… 306
13.25 实现工作流的度量 …… 306
13.26 实现工作流面临的挑战 …… 307
本章回顾 …… 307
延伸阅读材料 …… 308
习题 …… 309
参考文献 …… 310
第 14 章 交付后维护 …… 313
14.1 开发与维护 …… 313
14.2 为什么交付后维护是必要的 …… 314
14.3 交付后维护程序员要求具备什么 …… 314
14.4 交付后维护小型案例研究 …… 316
14.5 交付后维护的管理 …… 317
14.5.1 缺陷报告 …… 317
14.5.2 授权对产品的修改 …… 318
14.5.3 确保可维护性 …… 318
14.5.4 反复维护的问题 …… 319
14.6 维护问题 …… 319
14.7 交付后维护技能与开发技能 …… 321
14.8 逆向工程 …… 321
14.9 交付后维护期间的测试 …… 322
14.10 交付后维护的 CASE 工具 …… 323
14.11 交付后维护的度量 …… 323
14.12 交付后维护：MSG 基金会案例研究 …… 323
14.13 交付后维护面临的挑战 …… 323
本章回顾 …… 323
延伸阅读材料 …… 324
习题 …… 324
参考文献 …… 325
第 15 章 UML 的进一步讨论 …… 327
15.1 UML 不是一种方法学 …… 327
15.2 类图 …… 327
15.2.1 聚合 …… 328
15.2.2 多重性 …… 329
15.2.3 组合 …… 329
15.2.4 泛化 …… 330
15.2.5 关联 …… 330
15.3 注释 …… 330
15.4 用例图 …… 330
15.5 构造型 …… 331
15.6 交互图 …… 331
15.7 状态图 …… 333
15.8 活动图 …… 335
15.9 包 …… 335
15.10 组件图 …… 336
15.11 部署图 …… 336
15.12 UML 图回顾 …… 336
15.13 UML 和迭代 …… 336
本章回顾 …… 337
延伸阅读材料 …… 337
习题 …… 337
参考文献 …… 337
附 录 …… 338
附录 A 学期项目：Osric 办公用品和装饰公司项目 …… 338
附录 B 软件工程资源 …… 340
附录 C 需求工作流：MSG 基金会案例研究 …… 341
附录 D 分析工作流：MSG 基金会案例研究 …… 341
附录 E 软件工程管理计划：MSG 基金会案例研究 …… 341
附录 F 设计工作流：MSG 基金会案例研究 …… 345
附录 G 实现工作流：MSG 基金会案例研究（C++ 版） …… 349
附录 H 实现工作流：MSG 基金会案例研究（Java 版） …… 349
附录 I 测试工作流：MSG 基金会案例研究 …… 350

第一部分
面向对象软件工程简介

本书前9章的作用有二：一是向读者介绍面向对象软件过程；二是作为本书后半部分内容的基础，后半部分描述的是面向对象软件开发的工作流（活动）。

软件过程是生产软件的方式，它开始于概念探究，结束于产品退役。在这期间，产品将经历一系列步骤，包括需求、分析（规格说明）、设计、实现、集成、交付后维护和最终退役。软件过程不仅包括开发和维护软件所用的工具和技术，还涉及参与的软件工程人员。

第1章指出软件生产技术必须是有成本效益的，并且能促进软件生产团队成员之间的相互沟通。从这章开始，全书自始至终强这个目标的重要性。

第2章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、同步－稳定模型、开源模型、敏捷过程模型、螺旋模型以及最重要的迭代－递增模型（该模型是许多面向对象软件工程的基础）。为了使读者能够对具体的项目选用合适的生命周期模型，这章对各种生命周期模型进行了比较和对比。

第3章重点介绍了统一过程这一目前最有前景的软件开发方法。详细论述了敏捷过程这一流行的软件开发方法并介绍了开源软件。在本章结尾论述了软件过程的改进。

目前，大型软件项目仅凭个人力量是很难在给定的时间内完成的，这种项目通常由一组软件工程人员合作开发。第4章主要论述了团队该如何组织才能使其成员能富有成效地合作。本章论述各种不同团队，包括民主型团队、主程序员团队、同步－稳定团队、开源团队和敏捷过程团队的组织方法。

软件工程师要求能使用大量不同类型的工具，包括分析型工具和应用型工具。第5章将介绍不同的软件工程工具，其中之一是逐步求精，这是一种把大问题分解成较小但容易处理的问题的技术。另一种工具是成本－效益分析，这是一种判断软件项目经济可行性的工具。接着要描述的是计算机辅助软件工程（CASE）工具，这是一种用于辅助软件工程师开发和维护软件产品的工具。最后，为了管理软件过程，必须度量软件过程的不同指标，以判断该软件是否偏离了正常轨道。这些测量（度量）工具对一个项目的成功是至关重要的。

第10～13章详细介绍了第5章后面的两个主题——CASE工具和度量。这几章对支持每种工作流的CASE工具进行了讨论，同时也给出了恰当的工作流管理所需的度量。

第6章讨论了与测试有关的一些基本概念。对软件生命周期的每种工作流所使用的具体软件测试技术将在第10～14章中介绍。

第7章详细解释了类和对象，证明了为什么面向对象范型比传统范型更成功。本书的其余部分都会用到这一章介绍的概念，特别是在第10章、第11章和第12章。

第7章的思想在第8章中得到了扩展，编写可移植到不同硬件平台的可重用软件是非常重要的。这一章的第一部分讲的是重用，包括多种重用实例的研究以及一些重用策略（如模式和框架）。可移植性是这一章第二部分的主题，这部分对可移植性策略进行了比较深入的介绍。本章反复介绍了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。在软件设计项目开始之前，一个基本的要求是做一份详细的整体计划。项目一旦开始，管理者必须密切监督进度，注意项目是否偏离计划，并在必要时采取正确的行动。同样，向客户提供关于项目的时间和经费开销的准确估算也是很重要的。这一章描述了不同的估算技术，包括功能点技术和COCOMO II技术；给出了软件项目管理计划的详细描述。由于主要的计划活动都发生在分析工作流结束之后，所以在第11章将使用本章给出的材料。

第 1 章　面向对象软件工程的范畴

学习目标

通过本章学习，读者应能：

- 了解面向对象软件工程的定义。
- 解释现在面向对象范型被广泛接受的原因。
- 论述软件工程各方面的含义。
- 描述现代维护观点。
- 论述持续计划、测试和编制文档的重要性。
- 认识遵守伦理规范的重要性。

这是一个众所周知的故事，有一个公司的主管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，他与朋友一起尽情地讥讽了“愚蠢的计算机”一番后将账单扔掉了，一个月以后，他收到了一份标记过期 30 天的类似账单，接着，第 3 张账单也来了。又一个月之后，第 4 张账单来了，同时附有一份通知，提示如果不及时付清这个 0.00 美元的账单将可能采取法律行动。

第 5 张账单，上面标记过期 120 天，没有任何提示，直白而粗鲁地威胁道，如果不立即付清账单，将采取所有必须的法律手段。这位主管担心自己公司的信用会受到这个疯狂机器的影响，于是找了一位软件工程师朋友，跟他讲了这件恼人的事情。软件工程师忍住笑，让主管邮寄去一张 0.00 美元的支票。这产生了期望的结果，几天后一张 0.00 美元的收据寄来了，主管小心翼翼地收好这张收据，以防将来计算机宣称那张 0.00 美元的账单他还没有支付。

这个故事有一个不太为人知晓的结局。几天后，银行经理召见了这位主管。银行经理拿着一张 0.00 美元的支票问他，“这是你的支票吗?”

这位主管回答：“是的”。

“那你能告诉我为什么要签署一张 0.00 美元的支票吗?”银行经理问道。

于是，整个故事被重新讲述了一遍。当主管讲完时，经理盯住他，温和地问道“你付 0.00 美元对我们计算机系统会造成什么后果，你想过吗?”

计算机专业人员虽然会觉得这个故事可笑，但是也会感到一些窘迫。毕竟，任何一个人所设计或完成的产品，在其原型阶段，都有可能出现类似寄送催讨 0.00 美元信件这种问题。目前，虽然在测试中总能发现此类错误，但是计算机专业人员的笑声会包含一种恐惧感，他们担心这种错误没有在产品交付给顾客前被检测出来。

1979 年 11 月 9 日检测到的一个软件错误，却绝对称不上幽默。这天，美国战略空军司令部收到全球军事指挥和控制系统（WWMCCS）计算机网络发出的报告，报告称前苏联向美国发射了导弹，这引起了警报混乱［Neumann，1980］。而实际发生的情况和 5 年后放映的电影《War Games》中的剧情一样，是误把模拟演习当成了真实发生的战事。虽然出于可以理解的原因，美国国防部未能详细说明把实验数据当成真实数据的确切过程，但有理由把这个问题归于软件错误。整个系统在设计时就无法区分模拟和真实情景；或者是用户接口没有包括必要的检查，以确保系统终端用户能分辨真伪。换句话说，如果这个问题确实是由软件引起的，软件错误可能会给文明社会带来不愉快和灾难性的后果（有关由软件错误所导致的灾难方面的信息，请参见备忘录 1.1）。

无论涉及的是账单还是防空项目，许多软件都会推迟其交付时间，原因可能是预算超出、

带有残存错误，或者是不能满足用户的要求。软件工程（Software Engineering）是解决这些问题的一种尝试，换言之，软件工程是一门以生产出没有错误、按时并且可在预算内交付的、能满足用户需求的软件为目的的学科。而且当用户需求改变时，生产的软件必须易于修改。

软件工程的范畴非常广，某些方面可能归为数学或计算机科学，另一些方面则可能属于经济学、管理学或心理学的范畴。为了展示软件工程所涉及的广阔领域，本书将从5个方面进行考查。

备忘录 1.1

在WWMCCS网络案例中，灾难在最后一分钟得以避免。然而，有些软件错误却会导致悲剧的发生。例如，在1985年和1987年之间，至少有两位病人之死是由Therac-25医用线性加速器所产生的严重过量的辐射导致的［Leveson and Turner，1993］，而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导弹防御系统，击中了沙特阿拉伯达兰市附近的一个兵营，造成了28位美国人死亡，98人受伤。爱国者导弹的软件中有一个积累计时法（cumulative timing）错误。爱国者导弹的设计是每次只能工作几个小时，过了这个时间，时钟就会重置。这个错误未曾有过明显影响，也从未被检测到。然而，在海湾战争期间，在达兰市的爱国者导弹系统已经连续工作了100多个小时，其累积时间误差已经大得足以使系统变得不精确了。

在海湾战争中，美国使用船只运送爱国者导弹至以色列，以保护其免受飞毛腿导弹的攻击，以色列军队仅在8个小时后就察觉到了这个计时问题，并立刻向美国的制造商报告了这个情况，制造商也尽快纠正了这个错误，但悲惨的是，新软件在飞毛腿导弹直接击中兵营的第二天才送达以色列［Mellor，1994］。

幸运的是，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人的生活带来巨大的影响。例如，2003年2月，一个软件错误导致美国财政部发出了50 000张没有受益人名字的社会保险支票，由于没有名字，这些支票是无法储蓄或兑现的［St. Petersburg Times Online，2003］。2003年4月，贷款者被SLM公司（通常称为Sallie Mae）告知，由于计算机软件的问题，他们的助学贷款从1992年起就计算错了，但错误直至2002年底才被检测出来。大约有100万借款人被告知他们需要支付更多的还款，形式上可采用每月支付更多的还款额，或者在当初的10年还款期限过后，再继续支付额外的还款利息［GJSentinel. com，2003］。这两个错误虽然很快被纠正，但近百万人的经济受到了很大的影响。

一个带有广泛负面影响的失败软件项目是美国联邦调查局（FBI）在2000年至2005年所开发的虚拟案例档案（VCF）系统。当项目最终成为烂摊子被放弃的时候，已经浪费了纳税人1亿7千万美元的钱。而FBI也被迫继续使用过时的自动案例支持（ACS）系统，这妨碍了FBI的活动。

1.1　历史方面

发电机会出现问题，这是事实，但是，它比工资报表出问题的几率小得多；桥梁有时会倒塌，但是，它比操作系统崩溃的可能性小得多。软件的设计、实现和维护应当与传统的工程学科具有同等地位，在这一观念的驱使下，NATO研究小组在1967年创造了软件工程这一术语。软件开发应当与其他工程任务相类似，1968年，这一声明在德国Garmisch召开的NATO软件工程会议上得到了认可［Naur，Randell，and Buxton，1976］。这一认可并不令人太感到吃惊，会议本身的名字就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（参见备忘录

1.2)。会议得出结论：软件工程应当使用业已建立的工程类学科的基本原理和范型来解决所谓的软件危机（software crisis）问题，即软件产品的质量太差，并且交付日期和预算限制也无法满足。

尽管有许多成功的软件案例，但是延期交付、超出预算、存在错误的软件仍是不可接受的。例如，Standish Group是一家分析软件开发项目的研究机构，2004年，它完成了对9 236个软件开发项目的研究，研究结果可概括表示为图1-1［Hayes，2004］。从图中可以看出，仅有29%的项目是成功完成的，有18%的项目在完成之前被取消或者根本没有实现，其余53%的项目虽然得以完成并安装在客户的计算机上，但这些项目不是超出预算就是延期交付，或者是比最初确定的少了一些特性和功能。换句话说，在2004年里，这家公司成功的项目不足1/3，一半以上的项目呈现出软件危机的一个或多个征兆。

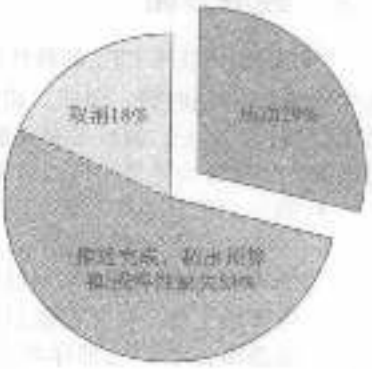


图1-1 对2004年完成的9 000多个开发项目的统计结果

软件危机带来的经济上的影响是非常可怕的。由Cutter Consortium［2002］所做的统计调查报告显示：

- 78%的信息技术机构卷入纠纷并以诉讼结束。
- 67%交付的软件产品没有达到软件开发者所声称的性能或功能。
- 56%承诺的交付日期被数次推迟。
- 45%的产品错误非常严重以致软件无法使用。

显然，只有很少的软件产品能够按时交付，不超出预算、没有差错且满足客户需求。为了达到这些目标，软件工程师需要掌握广泛的技巧，有技术上的也有管理上的。这些技巧不仅应用于编程上，还应该应用于软件生产的每一个步骤，从需求分析到交付后的维护。

40年之后，软件危机仍然伴随着我们，这个事实说明了两件事：第一，软件生产过程虽然在许多方面与传统工程相似，但有自己的特性和问题；第二，考虑到软件危机持续时间长且难以预测，软件危机应该重新命名为软件萧条（software depression）。

下面来看软件工程的经济方面。

备忘录1.2

如1.1节所说，Garmisch会议的目标是使软件开发如传统工程的实施一样得到成功。但并不是所有的传统工程项目都是成功的。下面以桥梁建筑为例来加以说明。

1940年7月，一座横跨华盛顿州塔科马纳罗斯的悬索桥建成了，但之后不久，大家发现，在大风下，桥会摇摆不定并出现危险的变形，上桥的汽车会消失在低谷，然后随着一部分桥面的升起而再现，因此，该桥被称为“飞跑的格蒂”。最后，1940年11月7日，该桥在时速42英里的大风中坍塌。幸运的是，在发生坍塌的几小时前，该桥已封。桥的最后15分钟被记录了下来并保存在美国国家电影档案馆。

2004年1月则发生了一件有点可笑的桥梁建筑事故。在德国Laufenberg镇附近的莱茵河上游，工程师欲建造一座连接德国和瑞士的新桥。桥靠近德国的一半由一个德国工程小组设计并建造，而瑞士那一半则由瑞士工程小组设计并建造，当桥的两部分合拢时，可明显看出，德国这一半比瑞士那一半高出21英寸（54厘米）。要解决此问题，桥的大部分都需要重建。问题产生的原因是瑞士工程师以地中海的平均高度作为海平面基准，而德国工程师使用的是北海作为海平面的基准。而为补偿海平面的差别，瑞士方建造的桥面本应升高10.5英寸，但他们却降低了10.5英寸，由此造成了21英寸的差距［Spiegel Online，2004］。

1.2　经济方面

使用旧编码技术 CT_{old} 的软件组织发现，使用新编码技术 CT_{new} 编写代码可比使用旧编码技术少花费 1/10 的时间，因此，花费 9/10 的时间。从常识上看，大家都会认为使用新技术 CT_{new} 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首选，但是，从软件工程经济学的角度看，有时会得出相反的结论。

- 第一个原因是将新技术引入一个软件组织的花费。使用 CT_{new} 技术后，编码速度可提高 10%，但这与将新技术引入开发组织的花费相比，可能就没有那么重要了。培训费用可能需要完成两到三个项目来弥补，另外，参加新技术 CT_{new} 培训的软件人员不能够从事生产性工作，甚至当他们培训结束后，还需要有一个艰难的学习过程，要使软件专家像熟悉旧技术 CT_{old} 那样熟悉新技术 CT_{new}，可能需要几个月的实践时间。使用新技术 CT_{new} 开发项目所花费的时间要比继续使用旧技术 CT_{old} 的时间长得多。在决定是否使用新技术 CT_{new} 时，所有这些花费都需要考虑到。
- 第二个重要原因是维护问题。新技术 CT_{new} 可能确实比旧技术 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} 的使用会导致代码很难维护，因而从整个产品的生命期来看，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 CT_{new} 是非常有吸引力的。毕竟，使用新技术 CT_{new} 的花费要少 10%。客户应当坚持使用旧技术 CT_{old}，付较高的前期投入并希望由此可减少软件的整个生命周期的花费。遗憾是的，客户和软件提供者的唯一目标是尽可能快地生产代码，他们在考虑使用一种专门的技术的短期效应时，通常会忽略它的长期效应。将经济学原理应用于软件工程，则要求客户选择能够降低长期成本的技术。

上述例子从对软件开发效果的贡献小于 10% 的编码技术进行了讨论。然而，经济学原理也适用于软件生产的其他方面。

下面讨论维护的重要性。

1.3　维护方面

本节在软件生命周期的环境下对维护进行描述。生命周期模型（life-cycle model）是对开发一个软件产品所需步骤的描述。在业界已有许多不同的生命周期模型，其中的一些在第 2 章进行描述。由于执行一系列较小的任务几乎总是比完成一个大的任务容易，因此可将整个生命周期模型划分为一系列较小的步骤，这些步骤称为阶段（phase）。阶段的数目因模型的不同而异，少则 4 个，多则 8 个。生命周期模型是应当完成工作的理论描述，与此相对照，对某个具体的软件产品，从概念开发到最终退役，所需完成的一系列实际步骤，则称为该产品的生命周期（life cycle）。在实践中，一个软件产品生命周期的若干阶段可能无法严格按照生命周期模型所规定的那样去实施，特别是时间和花费超支时，尤其如此。据称，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［Books，1975］。

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。这个模型的变体有许多，但大体上说，使用这种传统生命周期模型开发软件都将经历图 1-2 所示的 6 个阶段。这些阶段可能跟某个特定软件组织的规定不完全吻合，但是就本书目的而言，它们与大多数具体的开发实践非常接近。类似地，对于每个开发阶段的名称，不同软件组织之间也有较大的差异。对不同阶段的名字的选

1. 需求阶段
2. 分析（规格说明）阶段
3. 设计阶段
4. 实现阶段
5. 交付后维护
6. 退役

图 1-2　瀑布生命周期的 6 个阶段

择，本书尽量做到尽可能通用，希望读者能够感到便利。

1）需求阶段。需求阶段（requirement phase）探究并细化概念，提取客户的需求。

2）分析（规格说明）阶段。分析客户需求并以“期望产品做什么”的形式在规格说明文档（specification document）中给出，分析阶段（analysis phase）有时也称为规格说明阶段（specification phase）。该阶段结束时产生的详细描述所提议的软件开发计划也称为软件项目管理计划（software project management plan）。

3）设计阶段。在设计阶段（design phase），规格说明经历两个连续的设计。首先是体系结构设计（architectural design），它将作为整体的产品分解成若干称为模块（module）的组件；然后是详细设计（detailed design），它设计每个模块。阶段结束时得到了两个描述“产品是如何做的”的设计文档（design document）。

4）实现阶段。对各个组件独立进行编码（coding）和测试（单元测试，unit design），然后，将产品的各个组件组合起来并作为整体进行测试，这称为集成（integration）。当开发人员对产品功能感到满意时，由客户对该产品进行测试（验收测试，acceptance testing）。当客户接受产品并且产品已安装至客户的计算机时，实现阶段（implementation phase）结束（在第 13 章将叙述编码和集成应该并行进行）。

5）交付后维护。产品是用于完成开发时所赋予的使命的。在这期间，需要对其进行维护。一旦软件通过了验收测试并安装到客户的计算机上，交付后维护（postdelivery maintenance）就包括了该产品的所有改变。交付后维护包括纠错性维护（corrective maintenance，或软件修复，software repair）和增强性维护（enhancement maintenance，或软件更新）。纠错性维护主要是去掉残留的错误，它不对规格说明文档进行修改，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。增强性维护有两种类型：第一种是完善维护（perfective maintenance），对产品所做的改变将提高产品的性能，如增加功能或减少响应时间；第二种是适应性维护（adaptive maintenance），对程序进行修改以适应程序运行环境的变化，例如，适应新的硬件/操作系统或应对新的政府规定。（要深入了解这三种类型的交付后维护，请参见备忘录 1.3。）

6）退役。退役（retirement）是指产品退出服务。当产品所提供的功能对客户组织不再有任何用处时，该产品退役。

下面更详细地讨论一下维护的定义。

备忘录 1.3

在软件工程领域，最广泛引用的结论之一是：大约有 17.4% 的交付后维护是纠错维护；大约 18.2% 是适应性维护；大约 60.3% 是完善性维护；而 4.1% 可归为“其他”。这个结果来自 1978 年发表的一篇论文［Lientz，Swanson，and Tompkins，1978］。

然而，论文所得的结论不是来源于维护数据的测量结果。作者采取的是对维护经理进行调查，请他们估计一下，在其工作的组织内部，每种类型的维护各花费多少时间，并请他们对自己的估计给出一个确认度。更特别的是，当参加该项调查的软件维护经理被问及他们的答案所根据的是相当精确的数据、少量的数据还是没有数据依据时，49.3% 的经理是基于相当精确的数据的，37.7% 的经理是基于少量的数据的，而 8.7% 的经理是没有任何数据依据的。

事实上，就调查中有关用于各类维护的时间百分比问题，被问及人应当如实回答他的答案是否以“相应精确的数据”为依据。实际上，他们中的多数人都不是握有“较少数量的数据”。该项调查要求参加者说明在诸如“紧急修补”或“常规调试”等单项维护的百分比组成，而调查者则从这些原始数据推断出适应性维护、纠错性维护和完善性维护各自所占的百分比。但毫无疑问，在 1978 年，各个软件组织实质上还处在 CMM 的第一个阶段（参见 3.13 节）。

因此，有充足的理由提出质疑，1978 年之后的交付后维护活动的数据分布是否是那时参加调查经理的估计？与现在维护活动的分布毫无相似之处。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% 交付后维护是纠错性的，这与调查报告中所提到的 17.4% 不符。

1.3.1　现代软件维护观点

在 20 世纪 70 年代，人们认为软件生产包含了两个完全不同的活动：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并通过验收之后，对软件所做的任何改动（不管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［IEEE 610.12，1990］。因此，传统软件开发方法可以被描述为开发 - 维护模型（development-then-maintenance model）。

这是一个时间性定义（temporal definition），也就是说，活动按照其进行的时间归类为是开发还是维护。假设在软件安装后发现并纠正了一个软件错误，根据定义，该软件活动属于维护。但是，如果该错误是在软件安装之前发现并纠正的，那么根据定义，这样的软件活动属于软件开发。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，这应当认为是“完善性维护”。然而，如果客户在软件产品安装之前想要进行同一改变，这应当是开发活动。可见，虽然这两个活动在本质上没有任何不同，但一个被认为是开发，一个却被认为是完善性维护。

除了这种不一致外，下面两个原因进一步解释了现在开发 - 维护模型是不实际的：

1）现在开发一个软件产品，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，客户的需求很可能发生改变。例如，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。或者开发正在进行的时候，客户已将业务扩展到了比利时，所以需要对产品进行修改，以使其能处理在比利时的销售数据。为了看清此类需求变化对软件生命周期产生的影响，假设正处在设计阶段时，客户的需求发生了变化，这时，软件工程团队必须停止开发，修改规格说明文档来反映需求的变化，并且，还必须对设计进行修改。对规格说明的修改会迫使已经完成的部分设计也要进行相应的修改，只有完成这些修改，开发才能继续进行。换句话说，开发人员必须在产品安装前的很长时间就开始对产品进行“维护”工作。

2）开发 - 维护模型的第二个问题是，30 年前，软件开发团队是从零开始开发目标产品的。而在今天代价昂贵的软件生产中，软件开发者在产品构建中会尽量复用已经存在的软件产品的部分代码（第 8 章将详细讨论“复用”）。因而，在广泛重视复用的今天，开发 - 维护模型已经变得不适合了。

一个更现实的维护的定义是在国际标准化组织（ISO）和国际电子技术委员会（IEC）发布的生命周期过程标准中给出的，即维护是一个过程，是软件因存在问题或者为改进或适应需要而对代码及相应文档所进行的修改过程［ISO/IEC 12207，1995］。按照这个操作性定义（operational definition），不管是在产品安装前还是安装后，只要发生错误改正或需求变化，就是维护。国际电气和电子工程师协会（IEEE）与电子工业联合会（EIA）随后认可了这个定义，其中 IEEE 为了与 ISO/IEC 12207 相符，对 IEEE 标准作了修改［IEEE/EIA 12207.0—1996，1998］。（关于 ISO 的更多情况，参见备忘录 1.4。）

在本书中，交付后维护是指 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之后对软件所做的任何改变。而现代维护或维护（maintenance）的含义则是 1995 年 ISO/IEC 定义的，即任何时候所进行的纠错性、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。

备忘录 1.4

ISO 是由 147 个国家标准化机构组成的网络，中央秘书处在瑞士日内瓦。ISO 已经发布了 135 000 多个国际认可的标准，包括从照相胶片速度（“ISO 数字”）到本书给出的许多标准，例如，第 3 章讨论的 ISO 9000。

ISO 并不是一个首字母缩写词，它来源于希腊字 ισος，意思是“相等”，大家可以在一些词（如 isotope、isobar 和 isosceles）中看到英语的构词法前缀 iso-。国际标准化组织选择 ISO 作为其名称的缩写形式，是为了避免“International Organization for Standardization”翻译成不同成员国的文字后会产生多个缩写。为了达到国际标准化，才选择了这样一个通用的缩写。

1.3.2　交付后维护的重要性

有时人们会说只有坏的软件产品才需要交付后维护。但实际上，情况恰恰相反，坏的软件通常会被扔掉，而对好的软件，人们会在 10 年、15 年甚至 20 的时间范围内对其进行改进和提高。进一步来说，软件产品是为现实世界所建立的模型，而现实世界处在不断变化之中，因而，必须不断对软件进行维护以保证其能准确、持续地反映现实的客观世界。

假如，如果营业税率从 6% 增长到 7%，几乎每个涉及商品销售的软件产品都要进行修改。假设该产品包含下述C++ 声明语句：

```
const float SALES_TAX = 6.0;
```

或等价的 Java 声明语句：

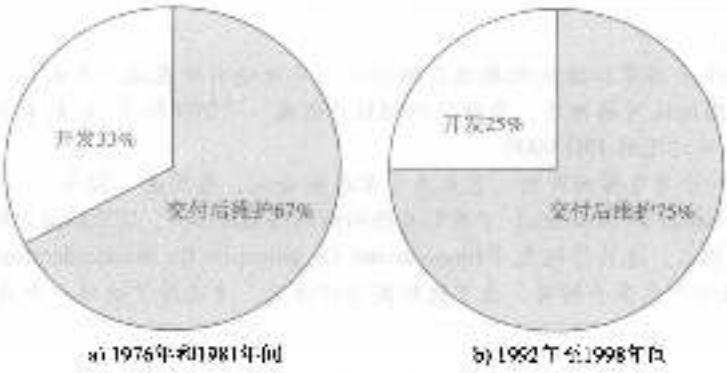
```
public static final float SALES_TAX = (float) 6.0;
```

语句声明 SALES_ TAX 为浮点常量，并初始化为 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，接着对代码进行重新编译和链接即可。然而，如果产品中不是使用变量名 SALES_ TAX，而是在使用营业税的地方直接使用值 6.0，那么对这样的产品进行改动会非常困难。例如，在原代码中，忽略了一些应该修改为 7.0 的 6.0 的出现，或者将那些不代表营业税的 6.0 错误地改成 7.0。找出这些错误很困难，也很费时间。事实上，对于某些软件，从长期效益来说，放弃并重新开发的花费要比找出需要改动的常量并探究如何修改的花费小。

现实世界也在不断变化之中。喷气战斗机所装配的导弹可能被新型号的武器所取代，这可能引发对机载电子系统的武器控制部分进行修改。给普通的 4 缸汽车提供 6 缸引擎，则需要修改控制燃料注入和调速功能的车载计算机系统软件。

但是对于交付后维护，应当投入多少时间（即金钱）呢？图 1-3a 的饼状图显示，30 年前，大概有 2/3 的软件花费用于交付后维护，统计数据取自不同的来源并进行了平均处理，包括 [Elshoff, 1976; Daly, 1977; Zelkowitz Shaw, and Gannon, 1979; Boehm, 1981]。较新的数据表明，交付后维护的花费所占的比例变得更大。许多组织将其软件预算的 70% ~80% 甚至更多用于交付后维护 [Yourdon, 1992; Hatton, 1998]，见图 1-3b。

现在再考虑一下当前正在使用 CT_{old} 编码技术的某个软件组织，他们得知使用 CT_{new}（新的软件开发技术）将减少 10% 的编码时间。即使 CT_{new} 在维护上没有什么不好的地方，一个聪明的软件管理者在改变编码技术前也会三思。全部软件开发人员都需要重新培训，新的软件开发工具需要购买，有可能需要额外雇用精通新技术的雇员。然而，编码和单元测试平均只占软件开发成本的 34% [Grady, 1994]，而开发成本仅占软件成本的 25%（如图 1-3b 所示）。因此，所有的付出和中断开发造成的损失的代价最多只能换来降低软件成本 25% 的 34% 的 10%（即 0.85%）的结果。



a) 1976年和1981年间　　b) 1992年至1998年间

图 1-3　开发和交付后维护平均成本的近似百分比

现在假设一种新技术能减少10%的维护成本，这种新技术则可能会被立即引入，因为它平均能减少整体成本的7.5%，引入这种技术所需要的耗费和它能节省的成本相比要小得多。

交付后维护非常重要，因而软件工程的一个重要方面就是可降低软件交付后维护成本的技术、工具和实践。

1.4　需求、分析和设计方面

软件开发人员也是人，因此在开发产品时，偶尔也会犯一些错误。因而，软件会存在缺陷。如果错误是在导出需求时所犯，发生的错误也有可能出现在规格说明、设计和编码之中。显然，错误越早纠正越好。

在软件生命周期的不同阶段纠正错误的相对成本情况参见图1-4［Boehm，1981］，图中数据来自IBM［Fagan，1974］、GTE［Daly，1977］和Safeguard［Stephenson，1976］，以及一些较小的TRW项目［Boehm，1980］。图1-4中的实线段是与大型项目相关的数据相拟合而成的曲线，而虚线段则是和小型项目相关的数据相拟合所得的结果。相应于软件生命周期的每个阶段，检测和纠正错误的相对耗费可参见图1-5。图1-5中每一实线段梯级都是取图1-4实线段上的相应点，并将点所连成实线段而成的。

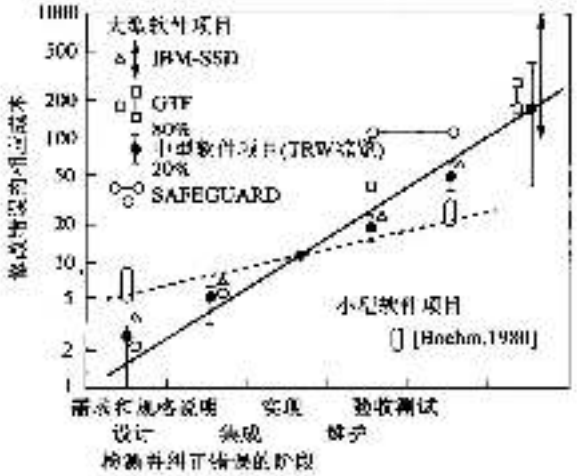


图 1-4　在软件生命周期的每个阶段解决错误的相对成本。实线段是与大型项目相关的数据相拟合，虚线段是和小型项目相关的数据相拟合

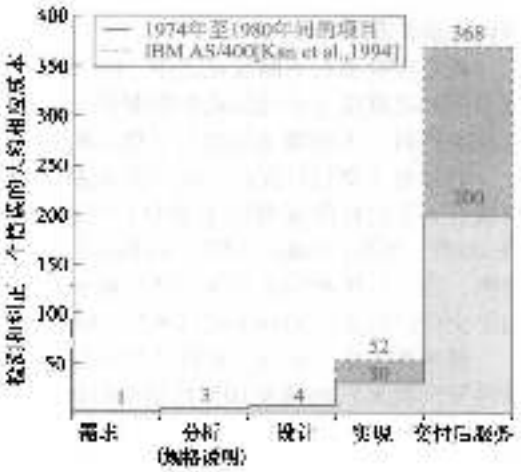


图 1-5　实线段描绘图1-4实线段上的点（延续成实线段），虚线段描绘较新的数据

假设在设计阶段检测和纠正一个错误需要花费 40 美元。从图 1-5 中的实线段（1974 年至 1980 年间的项目）可以看出，在分析阶段纠正相同的错误仅需要 30 美元，但是在交付后维护阶段检测和纠正相同的错误却需要花费 2 000 美元。更新的数据表明，现在，在较早阶段检测出错误变得更加重要。图 1-5 中的虚线段说明在 IBM AS/400［Kan et al.，1994］的系统软件的开发过程中检测和纠正一个错误的成本。平均来说，在 AS/400 软件交付后维护阶段，相同错误的纠正将花费 3 680 美元。

纠正一个错误的成本增长得如此之快，这主要和纠正一个错误需要完成的工作量有关。在软件开发生命周期的初期，产品仅存在于纸上，纠正错误可能就是仅仅修改一下文档。而对应的另一极端则是产品已经交付给客户。这时纠正错误，意味着编辑代码、重新编译和链接，以及仔细检查问题是否得到解决，接着，更关键的是，检查所进行的修改是否在产品的其他地方引发新的问题。所有相关的文档（包括手册）都要进行更新。最后，交付修改好的产品并重新安装。这说明应尽快尽早地检测错误，否则会付出巨大的代价。因此，在需求阶段和分析（规格说明）阶段，应当使用各种技术检测错误。

人们对于错误检测技术有进一步的需求。研究表明［Boehm，1979］，在较大的开发项目所发现的错误中，有 60% ~70% 的错误都是需求、分析或设计错误。最新的审查结果也表明：需求、分析或设计错误在错误比例中占的比例较大（审查是由一个小组对文档所做的详细检查，见 6.2.3 节）。喷气推动实验室是为 NASA（美国国家航空航天局）无人太空计划开发的软件，在对其进行的 203 次检查结果进行统计后发现，平均每页规格说明文档有 1.9 个错误，每页设计文档大概有 0.9 个错误，每页代码中只有约 0.3 个错误［Kelly，Sherif，and Hops，1992］。

而且，提高需求、分析和设计技术水平是非常重要的，这不仅是因为可以尽可能早地发现错误，还因为需求、分析和设计的错误占整个错误很大的比例。如 1.3 节中的例子所示，降低 10% 的交付后维护费用，就可以减少整个成本的 7.5%；而减少 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之多的错误，这提示了软件工程另一个重要的方面：高水平的技术能产生出更好的需求、规格说明和设计。对于大多数软件来说，是软件工程师团队而不是个人负责开发和维护软件生命周期方方面面的事情。下面来讨论团队开发。

1.5　团队开发

硬件的成本在不断下降，一台价格超过通胀前百万美元的 20 世纪 50 年代的大型计算机，今日其各方面的性能都不如一台价格低于 1 000 美元的笔记本电脑。因此，各种机构都买得起运行大型软件的计算机，而大型软件是无法由一个人在许可的时间内编写完成的。例如，如果一个软件需要在 18 个月内交付，而完成它需要花费一个软件专业开发人员 15 年的时间，那么产品就必须由一个团队来开发。然而，团队开发存在着代码组件的接口问题以及团队成员间的沟通问题。

例如，假定 Jeff 和 Juliet 分别编写方法 p 和 q，其中 p 调用 q。当 Jeff 编写 p 时，他使用了含有 5 个参数的参数表去调用 q。虽然 Juliet 编写的 q 也有 5 个参数，但其顺序和 Jeff 的不同。一些软件（如 Java 语言的解释器和装载器）仅在交换的参数具有不同类型时才能检测出这种类型的错误，如果参数的对应类型一致，那么问题可能很长时间也无法被检测到。也许有人会争辩说，这是一个设计问题，如果能更加细心地设计方法，这个问题就不会发生。这也许是对的，但实际上，编码开始之后设计也常改变，而发生改变的通知可能不会分发给团队的每个成员。因此，如果一个设计上的变化影响到了两个或更多的程序员，不良的沟通就会导致 Jeff 和 Juliet 所经历的情况。在由个人负责一个软件产品的各个方面之时，这种类型的问题是很少发生的，

在能购买得起运行大型软件的计算机之前也是如此。

在软件由一个团队开发的情况下，接口问题仅仅是冰上一角。除非团队组织良好，否则，大量的时间将浪费于团队成员间的会议之上。假设有一个产品需要一个程序员用一年的时间完成，若将任务指派给一个有6个程序员的团队，完成该工作常常也是一年而不是预期的2个月，而代码质量可能比整个工作由一个人完成的还要低（参阅4.1节）。由于今日的绝大多数软件是由一个团队开发和维护的，软件工程必须包括确保对团队进行良好管理和组织的技术。

如前面各节所述，软件工程的范畴是很宽广的，它包括了软件生命周期（从需求到交付后退役）的每个步骤，还包括了人为因素（如团队的组织），甚至涉及经济和法律（如版权等）。本章开始给出的软件工程定义包括了所有这些因素，即软件工程是一门学科，其目的是生产出满足客户需要的、未超出预算的、按时交付的且无错误的软件。

现在回到图1-2的瀑布模型，问一下为何没有计划、测试和文档阶段。

1.6　没有计划阶段的原因

显然，要开发一个软件产品，没有计划是不可能的。因此，在项目的初期有一个计划阶段（planning phase）似乎是基本的要求。

关键问题是，在确切知道开发什么之前，详尽的计划是无法描绘的。因此，使用瀑布模型开发软件产品时会有3种类型的计划活动：

1）项目开始时，对管理需求和分析阶段进行初步的计划。

2）一旦明确了到底要开发什么，就制定出软件项目管理计划（Software Project Management Plan，SPMP）。它包括预算、人事需求和详尽的日程安排。可以制定项目管理计划的最早时间是规格说明文档已经得到客户的认可，即分析阶段结束的时候。在此以前，计划都是初步和部分的。

3）在整个项目进行过程中，管理人员都需要监控SPMP并关注是否有偏离计划的情况发生。

例如，假定一个特定项目的SPMP指出整个项目将花费16个月的时间，而设计阶段将花费4个月的时间。一年之后，管理人员注意到项目整体比预期的进展要慢。经详细调查，到目前为止，设计已经花费了8个月的时间，但还远未结束。由此，项目几乎肯定要被取消，而到此为止花费的资金全浪费了。其实，管理人员早就应该按阶段跟踪项目的进展，这样最迟在项目开始2个月之后就能注意到设计阶段的一个重要问题。那时，就可以作出该如何以最好的方式继续项目的决定。通常，在这种情况下，第一个步骤是召开一个咨询会，确定项目是否可行，设计小组是否有能力完成此项任务，继续下去的风险是否太大。根据咨询报告，考虑各种备选方案，包括缩小目标产品的范围、设计并实现一个要求没有那么高的产品等。只有在所有的备选方案都被认为是不可行的时候，才取消项目。对于上述特定项目，如果管理人员密切监控项目，6个月以前就可以取消该项目了，由此可节省相当客观的开销。

总之，没有独立的计划阶段。相反，计划活动贯穿整个生命周期。然而，有时计划活动会占据主导地位。这包括项目开始的时候（最初计划）以及客户刚刚签署了规格说明文档（软件项目管理计划）之后。

1.7　没有测试阶段的原因

一个软件开发出来以后，对其一丝不苟地进行检查是绝对必要的。因此，有理由问一下为什么在产品实现之后没有测试阶段。

遗憾的是，在准备将一个产品交给用户的时候进行测试实在太迟了。例如，如果在规格说明文档中有一个错误，这个错误可能已经延伸到设计和实现中了。在开发过程之中，有时测试的执行几乎是和其他活动一起进行的。这常发生在每个阶段（验证，verification）接近结束的时

候，特别是产品提交给客户（确认，validation）时尤其如此。尽管测试有时占用过多的比重，但绝对不能不进行测试。如果将测试看成是一个独立的测试阶段（testing phase），那么就有可能存在不将测试持续贯穿于产品开发的每个阶段和维护过程的危险。

但是，这还不够。应该对一个软件产品进行持续检查。一丝不苟的检查应该自动伴随着软件开发和维护活动的每一个阶段。相反，一个独立的测试阶段和保证一个软件产品一直都不出现差错的目标是不立的。

每一软件开发组织都应该包括一个独立的小组，其主要职责是保证交付的产品就是客户所需要的，产品一直是以正确的方式打造的。该小组称为软件质量保证（Software Quality Assurance，SQA）小组。软件的质量（quality）是软件满足规格声明的程度。第 6 章将详细介绍质量和软件质量保证，以及 SQA 在设置和执行标准方面的作用。

1.8　没有文档阶段的原因

和不应该有独立的计划阶段和测试阶段一样，也不应该有独立的文档阶段（document phase）。在任何时候，一个软件产品的文档都必须是完整、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，其他阶段也是类似的。

1）确保文档是最新的，其基本原因是软件行业中从业人员的流动性。例如，假定设计文档没有保持更新，而主设计师离开去从事另一项工作，现在就很难更新设计文档以反映系统设计以来所作的所有改动。

2）除非前一个阶段的文档是完整、正确和最新的，否则执行下一个阶段的工作步骤几乎是不可能的。例如，一个不完整的规格说明文档不可避免会导致不完整的设计，继而产生不完整的实现。

3）除非有陈述软件产品的期望性能的文档，否则测试一个软件产品是否能正确工作实质上也是不可能的。

4）除非有一套描述当前版本的产品行为的完整、正确的精确文档，否则维护也几乎是不可能的。

因此，如同没有独立的设计或测试阶段一样，也没有独立的文档阶段。计划、测试和文档活动应该和软件产品构造的所有其他活动一起进行。

下面考察面向对象范型。

1.9　面向对象范型

1975 年之前，大多数软件组织都不使用专门的技术，每个人以其自己的方式工作。随着所谓的结构化（structured）或传统范型（classical paradigm）的发展，1975 年至 1985 年间，这种情况发生了突破性的变化。组成传统范型的技术包括结构化系统分析［Gane and Sarsen，1979］、结构化程序设计和结构化测试（13.12.2 节）。这些技术在首次使用时，似乎有较好的前景，然而，随着时间的推移，它们被证明在下面两个方面没有那么成功：

1）这些技术有时无法应付规模逐渐增大的软件产品。传统的技术适合于处理较小规模的软件（代码规模一般为 5 000 行）或代码长度为 50 000 行的中等规模的一般性软件。然而，现在有 500 000 行代码的软件相对来说是很普遍的，一个软件包括 500 万行甚至更多行的代码也是司空见惯的。但传统技术常常无法有效地按比例倍增其功能，以处理现在的大型软件开发。

2）传统范型无法满足用户对软件交付后维护的期望。30 年前传统范型发展的驱动力之一是，平均来说，软件交付后维护仅占 2/3 的软件预算（参阅图 1-3a）。遗憾的是，传统范型没有解决好这个问题，正如 1.3.2 节所指出的那样，许多组织仍然花费 70% ~80% 甚至更多比例的

时间和精力于交付后维护［Yourdon，1992；Hatton，1998］。

传统范型成功受限的主要原因是，传统技术要么是面向操作的，要么是面向属性（数据）的，而不是面向两者。相反，面向对象范型（object-oriented paradigm）则将属性和操作视为同等重要的实体。看待一个对象的简单方法是，将其视为一个包括属性以及对这些属性进行处理的操作的统一的制品（artifact，制品是软件产品的一个组成部分，如一份规格说明文档、一个代码模块或一份手册）。对象的这个定义并不完整，在详细考虑继承这个概念（7.8 节）之后，本书将充实该定义。尽管如此，这个定义还是抓住了对象的主要本质。

面向对象范型的主要优势包括：

1）面向对象范型支持信息隐藏，这是一个保证实现细节局部于一个对象的机制（更多细节请参阅 7.6 节）。因此，维护时，如果一个对象内的实现细节发生变化，信息隐藏保证无需对产品的其他部分进行修改就可保证一致性。相应地，面向对象范型使维护变得迅速而容易，极大地减少了回归错误（regression fault，指修改软件某部分时给明显无关的另一部分带来的错误）的出现。

2）除了维护之外，面向对象范型也使软件开发变得更容易。在许多情况下，对象在物理客观世界中都有对照之物。例如，银行软件产品中的一个银行账户对象可与采用该产品的实际账户相对应。本书第二部分将说明建模在面向对象范型中所起的重要作用。将软件产品中的对象与现实世界中的对照物紧密对应会产生出更高质量的软件。

3）有着良好设计的对象是一个独立的单元。如本节开始处所述，对象由属性和属性上的操作共同组成。若对一个对象的属性进行的所有操作都包含在该对象之中，就可把该对象视为概念上独立的实体。在一个产品中，若使用对象对现实世界建模，与现实世界相关的部分都可在对象本身上找到，这种概念独立性有时称为封装（encapsulation，参见 7.4 节）。还有一种形式独立性，即物理上独立。在一个设计良好的对象中，信息隐藏确保实现细节对对象外部的每一事物隐藏。外界唯一允许的与之通信的方式就是给对象发送消息，使对象执行特定的操作，而如何执行操作则完全是对象自己的职责。因此，面向对象设计有时也称为职责驱动设计（responsibility-driven design）［Wirfs-Brock，Wilkerson，and Winner，1990］或契约式设计（design by contract）［Meyer，1992］。（职责驱动设计的另一个观点可参阅备忘录 1.5，该例子取自［Budd，2002］。）

4）使用传统范型建造的产品有时包含一组模块，但在概念上，其实质还是一个单一的单元。这也是传统范型应用于建立大型产品不甚成功的原因之一。相反，如果正确使用面向对象范型，得到的产品会包括一些较小的，但独立性较高的单元。面向对象范型降低了软件产品的复杂度，从而简化了开发和维护过程。

5）面向对象范型提倡重用，因为对象是独立的实体，可用于未来的产品。对象复用减少了开发和维护的时间和费用，第 8 章将对此进行详细说明。

然而，面向对象范型也不是包治百病的灵药：

1）和所有的软件生产方法一样，面向对象范型也必须正确地使用；与其他范型一样，面向对象范型也很容易被误用。

2）当正确使用时，面向对象范型能解决一些（但不是所有）传统范型的问题。

3）面向对象范型也有自己的一些问题，这些问题将在 7.9 节介绍。

4）面向对象范型是目前最好的方法，然而，和其他方法一样，在未来其肯定也会被其他更优秀的方法所替代。

本书在讨论特定主题时，会相应指出面向对象范型的优势和缺点。即分析面向对象范型的材料不会出现在单一地方，而是散列于整本书籍。

现在来定义一些软件工程的术语。

备忘录 1.5

假定住在新奥尔良的你想给住在芝加哥的妈妈送上一束母亲节的鲜花。一种方法是（在万维网上）查阅芝加哥黄页，确定哪一家花店离你母亲的公寓最近，然后向其订购；另一种更便利的方法是在 1-800-flowers. com 网站上进行订购，将送交鲜花的所有责任托付给公司。1-800-flowers. com 的具体地址以及是哪家花店递送你订购的鲜花都与你无关。公司绝不会公布任何信息，这个例子说明的就是信息隐藏。

与此相同，当向一个对象发送消息时，不仅如何执行请求与此完全无关，发送消息的单元也不允许知道对象的内部结构。对象自身对执行消息的所有具体细节负完全责任。

1.10　术语

客户（client）是想雇人建造（开发）某一产品的个体。开发者（developer）是负责建造该产品的团队成员。开发者可能从需求开始，负责开发过程的每个方面，也可能只是负责一个已经设计好的产品的实现。

客户和开发者可能属于同一组织，例如，客户是一保险公司的首席精算师，而开发者是保险公司副总裁领导的负责软件开发的团队。这称为*内部软件开发*（internal software development）。另一方面，对于合同软件（contract software），客户和开发者则可能是完全独立的组织中的成员。例如，客户是国防部的一名高级官员，开发者是专门从事武器系统软件开发的国防合同承包商的雇员。一个更小的例子是，客户是一个单独从业的会计，开发者是一个利用课余时间编写软件赚钱的学生。

涉及软件生产的第 3 方是*用户*（user）。用户的行为由客户授权，他们使用所开发的软件。在保险公司的例子中，用户可以是保险代理人，他使用软件选择最合适的保单。在某些情况下，客户和用户也可能是同一个人（例如，前面讨论过的会计）。

与昂贵的、为特定用户所编写的定制软件相对应，市场上众多的软件拷贝（如文字处理软件和表格软件）是以低廉的价格出售给大量购买者的。即软件制造商（如微软或 Borland）是通过大规模销售来分摊产品开发成本的。这种类型的软件通常称为*商用现货*（COTS）*软件*（commercial off-the-shelf software），由于它们早期包含 CD 或软盘、手册以及产品许可证的盒子是使用收缩性薄膜塑料包装的，因此也称为*用收缩性薄膜包装的软件*（shrink-wrapped software）。现在，COTS 软件常常可通过万维网下载，这时就不再需要使用收缩性薄膜包装的盒子，因此，COTS 有时也称为*点击软件*（clickware）。COTS 软件是为“市场”开发的，在开发和销售之前，是没有客户和用户的。

开源软件（open-source software）目前也变得极为流行。开源软件产品由志愿者开发和维护，任何人都可以下载并免费使用。广泛使用的开源软件有 Linux 操作系统、Firefox Web 浏览器和 Apache Web 服务器等。开源是指任何人都可以获得源代码，和大多数仅销售可执行版本的商用软件不同。由于任何开源软件的使用者都可以查看源代码并向开源软件开发者报告软件错误，因此，许多开源软件的质量较高。开源软件错误公开化所期望的结果是由 Raymond 在《Cathedral and the Bazaar》一书中作为“Linus 法则”（Linus's Law，以 Linux 创始人 Linus Torvalds 的名字命名）提出的［Raymond，2000］。该法则指出，“如果有足够多的关注，所有错误将一目了然”。换句话说，如果有很多人查看一个开源软件的源代码，应该有人会确定错误所在并提出修改的建议。一个相关的原则是“及早发布，频繁发布”［Raymond，2000］。即开源软件的开发者趋向于比非开源软件的开发者花费较少的时间于软件测试，他们更喜欢在新版本刚完成后就及时发布，而将发现错误的责任更多地留给用户。

本书几乎每页都会出现的一个词——*软件*。软件不仅包括机器可读的代码，而且包括是每

个项目固有组成成分的所有文档。软件包括规格说明文档、设计文档、各种法律文书和财会文件、软件项目管理规划、其他管理文档以及各种类型的手册。

自20世纪70年代后，*程序*（program）和*系统*（system）之间的界限逐渐变得模糊。而当年，它们之间的区别是清晰的。通常以一堆穿孔卡片形式出现的程序是自治的、可以执行的代码段，而系统是一组相关的程序。例如，一个系统可能包括程序P、Q、R和S。安装上磁带T1，程序P开始运行。它读取卡片并将结果输出到磁带T2和T3。接着T2倒带，再执行程序Q，将结果输出到T4。再运行程序R，将磁带T3和T4的结果合并为T5。最后，T5作为程序S的输入，打印出一系列报告。

将该场景和一个运行在一台机器（它有一个前端通信处理器和一个后端数据库管理程序）上的钢铁厂实时控制软件产品相比较。该钢铁厂控制软件的功能远远超过老式系统，但就传统的程序和系统的定义而言，该软件无疑是一个程序。更加迷惑的是，*系统*一词目前也用来表示硬件和软件的组合体。例如，飞行器的飞行控制系统是由飞行中的计算机和运行在其上的软件组成。但也可认为飞行控制系统包括了将命令送往计算机的控制器（如操纵杆）和由计算机控制的飞机部件（如机翼等）。进一步来说，在传统的软件开发范畴内，术语*系统分析*（system analysis）是指软件开发的前两个阶段（需求和分析阶段），而*系统设计*（system design）指第三个阶段（设计阶段）。

为了减少困惑，本书使用*产品*（product）一词来表示有意义的软件。使用这种约定有两个原因：一是使用第三个术语可以回避程序和软件两个术语之间的混淆；二则更加重要，本书涉及的是软件的*生产过程*（process），而生产的最终结果就是*产品*。最后，术语*系统*取其现代含义（即软件和硬件的结合体），或作为被普遍接受的词语的一部分（如操作系统和管理信息系统）。

在软件工程领域，广泛使用的两个词是*方法学*（methodology）和*范型*。在20世纪70年代，*方法学*一词用于表示“开发软件产品的方式”，而该词实际上指的是“方法的科学”。在20世纪80年代，*范型*一词成了商业界的时髦词语，如“这是一种全新的范型。”很快，软件界也开始使用这个词，如面向对象范型和传统（或结构）范型等。这是术语选择中另外一个令人遗憾的例子，因为一个范型就是一个模式或一个模型。

方法学或范型将应用于整个软件过程。与之相反，*技术*只应用于软件过程的一个部分。例如，编码技术、文档技术和规划技术。

若一个程序员有了*过错*（mistake），该错误的结果就称为*差错*（fault）。执行有差错的软件产品，就会产生*故障*（failure），即差错的结果可导致可观察到的不正确的产品行为。*错误*（error）是使结果不正确的差错量。*过错*、*差错*、*故障*和*错误*这些词的定义见IEEE标准610.12“软件工程术语表”［IEEE 610.12，1990］，这些定义在2002年进行了修订［IEEE标准，2003］。*缺陷*（defect）是一个一般性的词，可指差错、故障或错误。为了精确性，*缺陷*一词本书将尽量少用。

一个大家尽可能要避免的词是*臭虫*（bug），关于该词的历史见备忘录1.6。今天，bug一词只是*差错*的委婉说法，尽管委婉叙述通常没有实际害处，但这种暗示对建造一个好的软件产品没有好处。特别地，当一个程序员犯过错时，本来他会说“我犯了一个错误”，而现在他会说“一个臭虫爬进了那段代码”（不说*我的代码*而说*那段代码*），从而将发生过错的责任从程序员推到了臭虫身上。没有人会责怪一个由于流感而躺下的程序员，因为流感是由流感病毒引起的。将过错指向臭虫是推卸责任的一种方法。相反，声明“我犯了一个过错”，则是一个负责任的专业人员的行为。

面向对象领域也存在许多令人混淆的术语。例如，*属性*（attribute）一词用于表示一个对象的数据成员之外，*状态变量*（state variable）一词有时也在面向对象的文献中使用；在Java中，相关的术语是*实例变量*（instance variable）；在C++中，是*域*（field）；在Visual Basic .NET中，则称为*属性*（property）。至于对象操作的实现，通常使用的是*方法*（method）一词，然而

在C++ 中，则被称为成员函数（member function）。实际上，在C++ 中，一个对象的成员（member）既可指属性（域）也可指方法。在 Java 中，术语域既可指属性（“实例变量”）也可指方法。为了避免混淆，本书使用一般性的术语属性和方法。

幸运的是，有一些术语已被广泛接受。例如，调用一个对象的方法几乎都被称为向对象发送一条消息（sending a message）。

最后来定义本书的主题。本章开始将软件工程定义为一个学科，其目的是生产出满足客户需要的、未超出预算的、按时交付的且无错误的软件。进一步，当用户需要改变时，软件必须易于修改。面向对象软件工程是一门利用面向对象范型实现软件工程目标的学科。

备忘录 1.6

bug 一词最早的使用者是计算机先驱、美国海军少将 Grace Murray Hopper（1906—1992）。1945 年 9 月 9 日，一只飞蛾飞进了 Hopper 和其同事正在使用的位于美国哈佛大学的 Mark II 计算机，并寄宿在一个继电器的两个触点之间。从而，系统出现了真正的 bug。Hopper 将出现 bug 一事记入日志，他写道“首次在计算机中发现了一只真实的 bug”。这本仍附有飞蛾尸体的日志目前保存在位于维吉尼亚 Dahlgren 的海军水面武器中心的海军博物馆。

尽管这可能是计算机界首次使用 bug 一词，但该词早在 19 世纪就曾作为工程行话使用[Shapiro，1994]。例如，发明家爱迪生在 1878 年 11 月 18 日就曾写道“事情变得不寻常了，接着，称为 Bug 的小差错和小麻烦……”[Josephon，1992]。1934 年版的《Webster's New English Dictionary》中 bug 一词的定义之一是“仪器或仪器操作上的缺陷”。从 Hopper 的备注可以清楚看出，她非常熟悉该词在计算机领域中的使用，否则，她会解释其含义的。

1.11　道德规范问题

本章以一些告诫来结束。软件产品由人开发和维护。如果这些人勤劳、聪慧、明智、现代，更重要的是有职业道德，那么软件开发和维护的方式会令人满意。遗憾的是，相反的情况也同样存在。

大多数专业团体都有一套其成员必须遵循的职业道德规范。两个主要的计算机专业团体——计算机器联合会（ACM）和电气电子工程师学会计算机协会（IEEE），联合通过了软件工程道德和从业规范，以作为软件工程教学和实践的标准[IEEE/ACM，1999]。该标准条文冗长，但也有一个由前言和 8 条原则组成的简本：

软件工程道德和从业规范（5.2 版）

IEEE-CS/ACM 软件工程道德和从业规范联合工作组推荐

简本

前言

简本在较高的层次上概括了规范的愿望；完全版中的条款则给出了细节和例子，指出这些愿望是如何影响软件工程专业人员的行为的。没有愿望，细节将变得空洞和教条；没有细节，愿望就是空洞的口号；愿望和细节一起构成了富有凝聚力的规范。

软件工程师将从事的是使软件的分析、规格说明、设计、开发、测试和维护成为一项有益的和令人尊敬的职业。与从事公共健康、安全、社会福利的职业人士一样，软件工程师应该遵循下面 8 条基本原则：

1）公众——软件工程师应始终如一以公众利益为重。

2）顾客及雇主——软件工程师应在最大程度上使顾客及雇主利益与公众利益相一致。

3）产品——软件工程师应确保他们的产品和相关修正尽可能符合最高级别的专业标准。

4）评判——软件工程师应在专业评判中保持诚实和独立。

5）管理——软件工程的管理者和领导者应该赞成并提倡在软件开发和维护过程进行具有职业道德的管理。

6）专业——软件工程师应提升与公众利益相符的专业诚信和声誉。

7）同事——软件工程师应和同事公平相待并互相帮助。

8）自身——软件工程师应在专业实践中终身学习并提升专业实践中的职业道德。

其他计算机专业团体的道德准则也有类似的观点。严格遵循道德条款对于软件工程师的职业未来是至关重要的。

第2章将考察和面向对象范型相关的各种生命周期模型。

本章回顾

软件工程定义（1.1节）为一门学科，其目的是生产出满足客户需求的、未超出预算的、按时交付的且无错误的软件。为了实现这个目标，需要在软件生产的各个阶段使用合适的技术，这些阶段包括分析（规格说明）和设计（1.4节）、交付后维护（1.3节）。软件工程涉及软件生命周期的各个步骤，结合了人类许多不同方面的知识，包括经济学（1.2节）和社会科学（1.5节）等。不存在独立的计划阶段（1.6节）、也没有独立的测试阶段（1.7节）和独立的文档阶段（1.8节）。1.9节讨论了面向对象范型。然后，在1.10节解释了本书使用的术语。最后，在1.11节讨论了道德规范。

延伸阅读材料

最早提及软件工程范畴的是［Boehm，1976］。关于软件工程在何种程度上可以被认为是一门真正的工程学科的分析，见［Wasserman，1996］和［Ebert，Matsubara，Pezzé，and Bertelsen，1997］。关于软件工程未来的讨论见［Brereton et al.，1999；Kroeker et al.，1999；and Finkelstein，2000］。2003年11/12月《IEEE Software》杂志中有多篇论文讨论了目前软件工程的实践状态。

关于交付后维护在软件工程中的重要性，以及如何为此制定计划的文章，见［Parnas，1994］。关于软件的不可靠性以及所产生的风险（特别是在安全关键型系统）的讨论见［Mellor，1994］和［Neumann，1995］。基于COTS产品的软件开发是［Brownsword，Oberndorf，and Sledge，2000］一文的主题。［Ulkuniemi and Seppanen，2004］和［Keil and Tiwana，2005］则描述了如何获得COTS组件。

［Scott and Vessey，2002］讨论了企业系统开发的风险，而一般信息系统的开发风险可见［Longstaff，Chittister，Pethia，and Haimes，2000］。有关软件危机的现代观点见［Glass，1998］。Zvegintzov［1998］解释了为何难以获得有关软件工程实践的精确数据的问题。

［Devlin，2001］强调了数学是软件工程的基础这一事实。经济学在软件工程中的重要性在［Boehm，1981；Baetjer，1996；and Boehm and Huang，2003］中讨论。2002年11/12月的《IEEE Software》杂志包含了一些有关软件工程经济学的文章。

［Weinberg，1971］和［Shneiderman，1980］是论述社会科学和软件工程关系的两本标准书籍。阅读这两本书通常不需要心理学或行为科学的预备知识。较新的讨论社会科学和软件工程关系的书是［DeMarco and Lister，1987］。

Brooks［1975］的不朽著作《The Mythical Man-Month》是一本备受推崇的介绍软件工程现实的书籍。该书包含了本章提到的所有主题的章节。

一篇杰出的介绍开源软件的文章是［Raymond，2000］。Paulsen、Succi和Eberlein［2004］以经验为根据给出了开源和非开源软件产品的比较研究。［Madanmohan and De'，2004］描述了

开源组件的复用。2004 年 1/2 月的《IEEE Software》杂志和 2005 年第 2 期的《IBM Systems Journal》杂志包含了一些开源软件的文章。

优秀的面向对象范型的介绍材料有［Meyer，1997］和［Budd，2002］。关于范型的中性观点可见［Radin，1996］。Khan、Al-A'ali 和 Girgis［1995］说明了经典范型和面向对象范型的不同。［Capper，Colgate，Hunter，and James，1994］描述了 3 个使用面向对象范型开发的成功项目，并给出了详细分析。［Johnson，2000］报告了就面向对象范型的态度对 150 位有经验的软件开发人员进行调查的结果。［Maring，1996］和［Fichman and Kemerer，1997］给出了开发大型面向对象产品的经验教训。［Webster，1995］描述了面向对象范型可能存在的缺陷。

习题

1.1　假定要为一个大型面包店设计一个自动化系统。估计用于软件开发的费用是 425 000 美元。问软件交付后维护工作的费用大概是多少？

1.2　对于维护，是否可在原来的时间性定义和现在正在使用的操作性定义间重新建立密切的关系？解释你的答案。

1.3　假定你是某公司的一个软件工程顾问。一个地区汽油销售公司的首席信息官希望你们公司为其开发一个软件产品，该产品能够执行公司所有的核算业务，并能在线为总店的工作人员提供关于订单和公司不同的存储仓库的存货信息。假定需要给 21 位负责核算的职员、15 位负责订单的职员、37 位负责库存的职员配备计算机。另外，还有 14 个管理人员需要存取数据。汽油销售公司可为该产品投入资金 30 000 美元，包括硬件和软件，并且希望你在 4 周之内完成该软件产品。你会如何答复该首席信息官？切记，无论他的要求有多么不合理，你的公司需要得到这个合同。

1.4　假定你是 Veloria 国海军的舰队副司令，现决定征召一个软件开发公司为新一代的舰对舰导弹开发控制软件，而你负责监督整个项目。为了保护 Veloria 政府的利益，在与软件开发者制定的合同中应该包含哪些条款？

1.5　假定你是一位软件工程师，工作是监督习题 1.4 中的软件的开发，列出你的公司可能在哪些方面不能履行与海军的合同。造成这些问题的可能原因是什么？

1.6　在交付产品 7 个月之后，使用 Stein-Röntgen 试剂分析 DNA 的产品的软件中发现了一个问题。纠正这个错误需要花费 16 700 美元。错误的起因是规格说明文档中的一条模糊语句。在分析阶段纠正该错误大约需要花费多少钱？

1.7　假设习题 1.6 中的错误是在实现阶段发现的。纠正该错误大约需要花费多少钱？

1.8　假定你是一个构建大型软件的组织的总裁。你将图 1-5 展示给雇员，要求他们在软件生命周期的早期发现错误。有人认为，期望在错误还没有混入产品之前就消除错误是不合理的。例如，如何能在产生设计的时候纠正一个编码的错误呢？对这个问题你该如何回答？

1.9　描述客户、开发者和用户都是同一个人的情形。

1.10　若客户、开发者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、开发者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，看看它有多少种不同的定义？写出那些在软件工程背景下有意义的定义。

1.13　在你得到第一份工作的第一天，经理给你一份程序清单并且说，“看看能否把其中的 bug 找出来。”你将如何回答。

1.14　假定由你负责开发习题 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　一个软件产品的开发者不想实现其中编号为 c9 的组件，而决定购买一个与组件 c9 有相同规格说明的 COTS 组件。这样做的利弊是什么？

1.16　一个软件产品的开发者不想实现 c37 组件，而决定利用一个与 c37 组件有相同规格说明的开源组件。这样做的利弊是什么？

1.17　（学期项目）假定附录 A 中的 Osric 办公用品和装饰公司的产品已经如所描述的那般实现。现在 Osric 想调整该产品，以便可以人工修改队列中顾客的优先级。该用什么方法修改现在的产品？抛弃一切，从头开始是不是更好？

1.18 （软件工程读物）指导老师提供文章［Schach et al.，2003b］的复印件。对于基于管理者的估计的结果和基于实际数据进行计算的结果，你认为相应的价值各是什么？

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), pp. 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 Society Press, Los Alamitos, CA, 2000.

[Gane and Sarsen, 1979] C. GANE AND T. SARSEN, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, Englewood Cliffs, NJ, 1979.

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

⊖ 本书中给出的网址在出版此书时都是正确的，但是网址通常会频繁地更改，而且不会有事先或持续的通知。如果发生这种情况，读者可以使用搜索引擎找到新的网址。

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

[Goldstein, 2005] H. GOLDSTEIN, "Who Killed the Virtual Case File?" *IEEE Spectrum* **43** (September 2005), pp. 24–35.

[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.

[Hayes, 2004] F. HAYES, "Chaos is Back," *Computerworld*, www.computerworld.com/managementtopics/management/project/story/0,10801,97283,00.html, November 8, 2004.

[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, 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.

[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.

[Keil and Tiwana, 2005] M. KEIL AND A. TIWANA, "Beyond Cost: The Drivers of COTS Application Value," IEEE Software 22 (May/June 2005), pp. 64–69.

[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.

[Madanmohan and De', 2004] T. R. MADANMOHAN AND R. DE', "Open Source Reuse in Commercial Firms," *IEEE Software* **21** (November/December 2004), pp. 62–69.

[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 En-*

gineering: 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.

[Paulson, Succi, and Eberlein, 2004] J. W. PAULSON, G. SUCCI, AND A. EBERLEIN, "An Empirical Study of Open-Source and Closed-Source Software Products," *IEEE Transactions on Software Engineering* **30** (April 2004), pp. 246–56.

[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/.

[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. 351–66.

[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.

[Spiegel Online, 2004] "Rheinbrücke mit Treppe—54 Zentimeter Höhenunterschied," www.spiegel.de/panorama/0,1518,281837,00.html.

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

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

[Ulkuniemi and Seppanen, 2004] P. ULKUNIEMI, AND V. SEPPANEN, "COTS Component Acquisition in an Emerging Market," *IEEE Software* **21** (November/December 2004), pp. 76–82.

[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

通过本章学习，读者应能：

- 描述软件产品的实际开发方式。
- 理解进化树生命周期模型。
- 认识软件产品因改动而产生的负面影响。
- 应用迭代 - 增量生命周期模型。
- 理解米勒法则对软件生产的影响。
- 描述迭代 - 增量生命周期模型的优点。
- 领悟尽早降低风险的重要性。
- 描述敏捷过程，包括极限编程。
- 比较并分析各种生命周期模型。

第 1 章描述了理想情况下软件产品的开发过程。本章则主要介绍实际的软件开发过程。从后文我们会看到，理论与实践之间存在着巨大的差异。

2.1　理想软件开发

理想情况下，软件产品的开发过程如第 1 章所述。图 2-1 概略描述了从零开始开发系统的过程，其中，∅代表空集。（如果想要了解术语“从零开始”（from scratch）的来历，请参阅备忘录 2.1。）首先明确客户需求，再进行分析，在得到完整的分析结果之后，进入设计阶段。紧接着的是整个软件产品的实现，以及将产品安装到客户的计算机之上。

然而，实际的软件开发与此相差甚大，原因有二：其一，软件专业人员是人，因此可能犯错；其二，在软件开发过程中，客户需求可能发生改变。本章将深入讨论这两个问题，这里先给出一个小型的研究案例以说明相关的论点，该案例是以［Tomer and Schach，2000］中的研究案例为基础的。

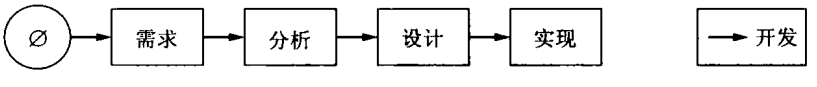


图 2-1　理想的软件开发过程

备忘录 2.1

术语“from scratch”来源于 19 世纪的运动学术语，意指“从零开始”（starting with nothing）。在正式道路（和跑道）铺就之前，赛跑只能在开阔地上举行。大多数情况下，起跑线就是沙地上的一条划线。没有优势（advantage）或差点（handicap）的赛跑运动员，必须从起跑线处起跑，即“从划线开始”（from［the］scratch）。

现在，术语“scratch”有着不同的运动学内涵。如“scratch golfer”指的是高尔夫运动中高尔夫差点（golfing handicap）为零的球员。

2.2　Winburg 小型案例研究

为缓解印第安纳州 Winburg 市的交通拥堵状况，市长说服市政府建立一个公共交通系统，

设立公交车专用通道，鼓励通勤者“停车换乘”，即在郊区停车场泊车，然后乘坐公交车上班和返回，每次换乘花费 1 美元。假定每辆公交车都配备自动收款机，仅接收 1 美元的钞票。当乘客上车时，将钞票投入进钞口。收款机中配置的传感器对钞票进行扫描，机器中的软件使用图像识别算法来判定乘客投入的钞票是否有效。重要的是收款机必须精确，因为一旦有消息称使用任何纸张都可以骗过机器，那么车费收入将很快变为 0。反之，如果机器总是拒绝合法的钞票，乘客将不愿再搭乘公交车。此外，收款机处理速度必须要快，如果机器需要 15 秒来判定钞票的有效性，即使只有几个乘客，有的市民也需要耗费几分钟才能上车，在这种情况下，乘客也不会再愿意乘坐公交车。因此，自动收款机软件的需求就是，必须保证平均响应时间少于 1 秒，且精确度至少要达到 98%。

阶段 1：实现软件的第 1 个版本。

阶段 2：测试结果表明，在判定美元钞票有效性时，时长 1 秒的平均响应时间需求未能达到。事实上，需要 10 秒才能得到响应。高层管理者找到了其中的原因。好像是，为达到 98% 的精确度，程序员曾应其经理的指示，在所有的数值计算中都使用双精度数。其结果就是，相对于单精度数，每个操作都需要至少 2 倍的耗时。因此，程序运行缓慢，响应时间很长。而后进行的计算表明，不管之前经理跟程序员作了怎样的指示，即使使用单精度数，也能达到 98% 精确度。为此，程序员开始着手对实现进行必要的修改。

阶段 3：在程序员结束其工作之前，对系统作了进一步的测试。结果表明，即使对实现进行了前述修改，系统的平均响应时间仍超过 4.5 秒，难以接近规定的 1 秒。主要原因是图像识别算法的复杂性。幸运的是，这时找到了更快的算法。因此利用新算法，重新设计并编写了自动收款机软件。最终，平均响应时间得以满足。

阶段 4：迄今为止，项目已大大滞后于计划进度，并且还可能超出预算。市长可谓是具有精明头脑的成功企业家，他要求软件开发团队尽可能加强系统中美钞识别组件的精确度，以便将开发的软件打包出售给自动售货机公司。为满足这一新的需求，采用了新的设计，使得平均精确度提高到了 99.5% 以上。管理层终于决定将这一版本的软件安装到自动收款机。至此，完成了相关软件的开发。后来，系统卖给了两家小型自动售货机公司，弥补了约 1/3 的成本超支。

尾声：几年后，自动收款机中的传感器变得陈旧，需要更换新的模块。管理层建议，利用这次修改机会同时更新硬件。软件专业人员则指出，硬件改变也意味着需要新的软件。他们建议，用一种程序语言重写软件。在程序编写期间，项目比计划进度晚了 6 个月，并且预算已超支 25%。不过，即使在满足响应时间和精确度需求时，存在“些许差异”，项目组的每个成员仍相信新系统会更为可靠，且更加优秀。

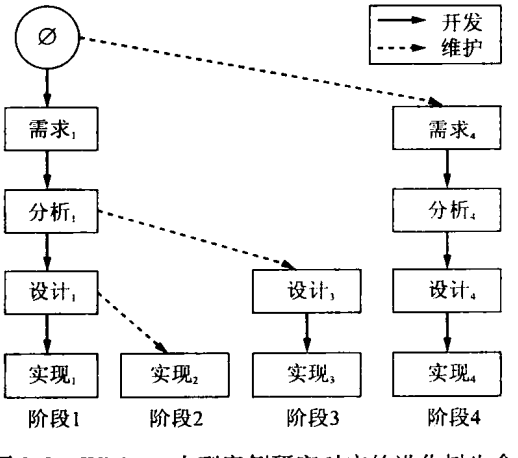


图 2-2　Winburg 小型案例研究对应的进化树生命周期模型（用虚线画的矩形框表示未完成的实现）

图 2-2 给出了这一小型案例研究的进化树生命周期模型（evolution-tree life-cycle model）。最左侧的方框代表阶段 1。如图 2-2 所示，系统开发从零开始（∅）。紧接着，依次是需求（需求$_1$）、分析（分析$_1$）、设计（设计$_1$）和实现（实现$_1$）。然后，如前所述，软件第一版的实验表明，1 秒的平均响应时间未能满足，必须对实现进行修改。在图 2-2 中，修改后的实现以实现$_2$给出。然而，实现$_2$始终没能完成，因此，实现$_2$的矩形表示采用的是虚线。

在阶段 3，必须修改原有设计。特别地，需要使用更快的图像识别算法。设计的改变（设

计$_3$）得到了对应实现的修改（实现$_3$）。

最后，在阶段4，为提高精确度，对需求也作了修改（需求$_4$）。由此得到修改后的规格说明（分析$_4$）、设计（设计$_4$）和实现（实现$_4$）。

在图2-2中，实线箭头表示开发，短划箭头表示维护。例如，在阶段3修改设计时，作为分析$_1$的设计的设计$_3$取代了设计$_1$。

进化树模型是生命周期模型（或简称模型）的一个实例，即在软件产品开发和维护过程中执行的步骤序列。另一个可用于此小型案例研究的生命周期模型是瀑布生命周期模型［Royce，1970］。图2-3给出了瀑布模型的一个简化版本。瀑布生命周期模型可以看作是带有反馈循环的图2-1所示的线性模型。如果在设计阶段出现错误，该错误是由需求阶段中的错误所致，则沿着虚线的向上箭头，软件开发人员可以从设计回溯到分析，并进而回溯到需求，以作出必要的修改。然后，再下移到分析，对应于需求修正规格说明文档，并依次修正设计文档。此时，可以从发现错误时被挂起的位置恢复设计活动。同样，实线箭头代表开发，短线箭头代表维护。

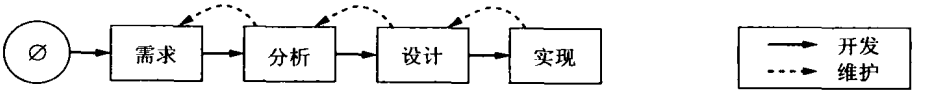


图2-3　瀑布生命周期模型的简化版本

毫无疑问，瀑布模型可以用来表示Winburg小型案例研究，但是与图2-2中进化树模型不同，它不能给出事件的顺序。与瀑布模型相比，进化树模型有更多的优点。它在每个阶段结束时，都有一条基线（baseline），即一个完整的软件制品（即软件产品的一个组成部件）集合。在图2-2中存在4组基线，分别是：

阶段1的基线：需求$_1$、分析$_1$、设计$_1$、实现$_1$。

阶段2的基线：需求$_1$、分析$_1$、设计$_1$、实现$_2$。

阶段3的基线：需求$_1$、分析$_1$、设计$_3$、实现$_3$。

阶段4的基线：需求$_4$、分析$_4$、设计$_4$、实现$_4$。

第一条基线为初始的软件制品集合；第二条基线对应于阶段2中修改后（但从未完成）的实现，即实现$_2$，连同未改变的阶段1中的需求、分析和设计；第三条基线与第一条基线相同，不过设计和实现都发生了变化；第四条基线中所包含的是全新制品的完整集合，如图2-2所示。在第5章和第14章中，我们将再次提及制品的概念。

2.3　Winburg小型案例研究经验

Winburg小型案例研究描绘了一个含有错误的软件产品的开发过程，这类错误多源于诸多互不相关的因素，如拙劣的实现策略（使用不必要的双精度数）以及过慢的算法等。虽然最终项目是成功的。然而，一个显而易见的问题是：实际的软件开发真的是这么杂乱无章吗？事实上，该小型案例研究的创伤程度远不如许多其他的（即使不是大多数的）软件项目。在Winburg小型案例研究中，由于错误（误用双精度数以及使用不满足响应时间需求的算法）仅产生了两个新的版本，由于客户所作修改（需要提高精确度）也仅产生了一个新的版本。

对于一个软件产品而言，为何需要如此频繁的修改？首先，如前所述，软件专业人员是人，难免犯错；其次，软件产品是对现实世界建模，而现实世界是不断变化的。在2.4节将更为深入地讨论这类问题。

2.4　Teal Tractors公司小型案例研究

Teal Tractors公司在全美大部分地区销售拖拉机产品。公司曾要求软件部门开发一个新的软件产品，以处理各个方面的业务。例如，销售、库存、支付佣金给销售人员，以及进行必要的

财务处理等。在软件产品的实现阶段，Teal Tractors 公司收购了一家加拿大拖拉机公司。为节省开支，公司管理层决定，将加拿大公司的业务合并到美国公司的业务中。这也意味着，在软件实际完成之前，必须对其改造：

1）必须进行修改以处理所增加的销售区域。

2）必须进行扩展以使其能处理因地处加拿大而有所不同的业务形态，如税收等。

3）必须进行扩展以使其能处理美元与加元两种不同的货币。

Teal Tractors 公司是一家快速成长的、具有良好发展前景的公司。接管加拿大拖拉机公司是一项积极的发展项目，这将在未来几年为企业带来更为丰厚的利润。但是，对软件部门而言，对加拿大公司的收购是一场灾难。除非原有的需求、分析和设计都考虑到了未来可能的扩展，否则，因增加加拿大销售区域而产生的软件修改量会很大，甚至于还不如放弃迄今所做的所有工作，从零开始。原因在于，在实现阶段对产品进行修改等同于在生命周期的末端（见图 1-5）进行修改。为处理特定于加拿大市场和加拿大货币的各方面因素而对软件进行扩展，这可能同从零开始开发软件一样困难。

即使软件开发经过深思熟虑，且初始设计也确实是可扩展的，但由于所引入的多个补丁，最终产品的内聚性也难以匹敌一开始就综合考虑美国和加拿大业务而开发的产品。因此，后续维护可能会有严重的隐患。

Teal Tractors 公司的软件部门成了*移动目标问题*（moving-target problem）的受害者。也就是说，在开发期间，软件需求发生了改变引起的问题。不管改变的理由有多重要，事实是，接管加拿大公司的决策对于正在开发的软件质量而言可能非常不利。

在某些情形下，移动目标的原因并不是良性的。有时，一个组织内部握有实权的高层管理人员会想在开发过程中改变软件产品的功能需求。另一类情形是*功能蔓延*（feature creep），即添加细小甚至是琐碎的需求。但是，不论何种原因，也不管多么细微，频繁的改变都会危及软件产品的健壮性。因此将软件产品设计为一组尽可能独立的组件是非常重要的，这样软件某一部分的改变不会在其他无关代码中引入错误，即所谓的*回归错误*（regression fault）。在进行代码修改时，可能会在代码间引入依赖关系。最终，由于存在太多的依赖关系，接下来的任何修改都会引入一个或多个回归错误。此时，唯一可行的就是重新设计并实现整个软件产品。

遗憾的是，对于移动目标问题，没有已知的解决方案。需求修改有积极的一面，成长型公司总在不断变化，这类变化必将反映到公司的关键软件产品中。同时，需求修改也有不利的一面，如果有足够影响力的个人要求进行修改，就不可避免地要在实现阶段进行修改，此举将危害软件产品的未来维护。

2.5　迭代与增量

在软件产品开发过程中，由于移动目标问题，以及难以避免的错误纠正，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，而不是图 2-1 中理想的开发过程。由此造成的一个结果就是，单独谈论“分析阶段本身”意义不大。实际上，分析阶段涉及整个生命周期。类似地，图 2-2 给出了实现的 4 个不同版本，由于移动目标问题，实现始终未完成。

考察某个软件制品的后继版本，例如，规格说明文档或代码模块。按此观点，其基本过程是迭代。即先开发该制品的第一个版本，然后修改并得到第二个版本，依此类推。目标是，较之于前驱版本，每个版本都更为接近目标，并最终构造一个满足条件的版本。*迭代*（Iteration）是软件工程的一个内在特性，并且迭代生命周期模型已经使用了 30 多年［Larman and Basili, 2003］。例如，在 1970 年提出的瀑布模型，就是迭代（但不是增量）的模型。

实际软件开发的第二个方面受*米勒法则*（Miller's Law）的约束。1956 年，心理学教授 George Miller 指出，任何时候，人们仅可能专注于大约 7 个信息块（chunk，信息单位）［Mill-

er, 1956]。但是，一个典型的软件制品，其程序块数远大于 7。例如，一个代码制品可能远不止 7 个变量，一个需求文档可能不止包含 7 个需求。也就是说，某个时刻我们可处理的信息数是有限制的，一种解决方法是采用*逐步求精*（stepwise refinement）。即先关注当前最为重要的方面，稍后考虑当前并不那么重要的其他方面。换言之，最终会处理每个方面，但要按照各个方面当前的重要性进行排序。这意味着，初期仅开发可解决目标一小部分的制品。然后，进一步考虑问题的其他方面，增加新的片断到已有制品。例如，在构造需求文档时，先考虑最为重要的 7 个需求，接下来考虑另外 7 个次重要的需求，依此类推。这是一个增量过程。*增量*（Incrementation）也是软件工程的一个内在特性，增量软件开发已有 45 年以上的历史了 [Larman and Basili, 2003]。

实际上，迭代与增量总是交互使用的。也就是说，一个制品的构造是逐片的（增量），并且每次增量都要经历多个版本（迭代）。这一思想如图 2-2 所示，它演示了 Winburg 小型案例研究（见 2.2 节和 2.3 节）的生命周期。如该图所示，不存在独立的“需求阶段”，客户的需求经由两次提取和分析，生成初始需求（需求$_1$）和修改后的需求（需求$_4$）。类似地，也不存在独立的“实现阶段”，而是包含 4 个不同的阶段，这 4 个阶段中的代码分别生成并修改。

图 2-4 概括了这些思想，明确了*迭代 - 增量生命周期模型*（iteration-and-incremental life-cycle model）[Jacobson, Booch, and Rumbaugh, 1999] 的基本概念。该图给出了一个软件产品开发过程中的 4 次增量，分别为增量 A、增量 B、增量 C 和增量 D。水平轴表示时间，垂直轴表示人时（1 人时表示一个人在一个小时内所能完成的工作量），每条曲线下方的阴影区域表示该次增量的总工作量。

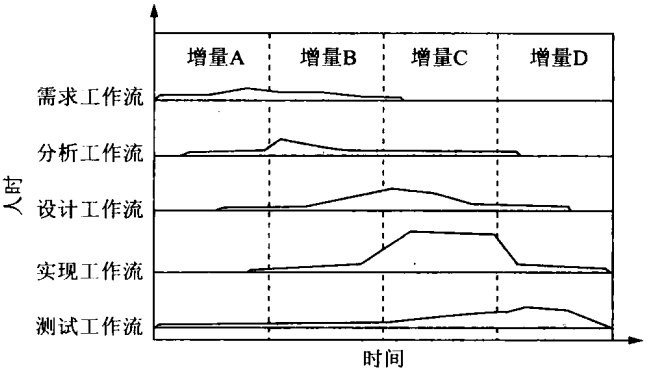


图 2-4　包含 4 次增量的软件产品开发

重要的是，要认识到图 2-4 给出的仅仅是软件产品分解为增量的一种可能方式。另一个软件产品可能仅通过两次增量来开发，而第三个产品则可能需要 13 次增量。并且，该图并未给出实际软件产品开发的精确表示，它只是演示了逐次迭代中重要的变化。

图 2-1 的顺序过程是假想的开发过程。实际上，正如图 2-4 所反映的那样，必须承认在软件整个生命周期中实施着不同的*工作流*（workflow，或活动）。实际上，*核心工作流*（core workflow）有 5 个：*需求工作流*（requirements workflow）、*分析工作流*（analysis workflow）、*设计工作流*（design workflow）、*实现工作流*（implementation workflow）和*测试工作流*（test workflow），并且，如上面所述，整个软件生命周期都执行着所有这 5 个工作流。但是，有时可能其中某个工作流支配着其余的 4 个。

例如，在软件生命周期开始阶段，软件开发人员的工作是提取初始需求。换言之，在迭代增量生命周期初期，需求工作流占据了主导地位。在生命周期的余下阶段，对这些需求制品进行扩展和修改。在此期间，其余 4 个工作流（分析、设计、实现和测试）依次占主导地位。也就是说，需求工作流是生命周期初期的主要工作流，不过，其所对应的重要性将逐步减弱。相反，较之于初始阶段，越趋向生命周期末端，实现和测试工作流将越多地占据软件开发团队成员的时间。

在整个迭代 - 增量生命周期中，计划和文档活动贯穿始终。并且，测试是每次迭代的主要活动，尤其是在每次迭代的后期。此外，一旦开发完成，软件就将作为一个整体来进行测试。此时，测试以及根据各种测试结果对实现进行的修改成为软件团队事实上的唯一活动。图 2-4

的测试工作流反映了这一点。

图 2-4 给出了 4 次增量。先考虑位于最左一栏的增量 A。在此次增量的起始，需求团队成员确定好客户的需求。一旦大多数需求被确定，就可以开始第一次分析。当分析已有足够的进展后，就可以启动第一次设计。虽然在第一次增量期间经常会进行一些编码工作，但主要还是使用概念验证原型（proof-of-concept prototype）来测试所提出的部分软件产品的可行性。最后，如前所述，在第一天就启动计划、测试和文档活动，并且长此以往，直到最终将软件产品交付给客户。

与此类似，增量 B 中所主要关注的是需求和分析工作流，然后是设计工作流。增量 C 则首先关注设计工作流，其后关注实现工作流和测试工作流。最后，增量 D 则主要集中在实现工作流和测试工作流。

正如［Grady，1994］所揭示的那样，需求和分析工作流（一起）大约占据了总工作量的 1/5，设计工作流占 1/5，其余3/5则用于实现工作流。图 2-4 中阴影部分的相对大小即反映了这一比例。

图 2-4 中的每次增量都包含迭代。如图 2-5 所示，在增量 B 中含有 3 次迭代。（图 2-5 是对图 2-4 中第二列的放大。）图 2-5 表明，每次迭代都包含所有 5 个工作流，但所占比例各不相同。

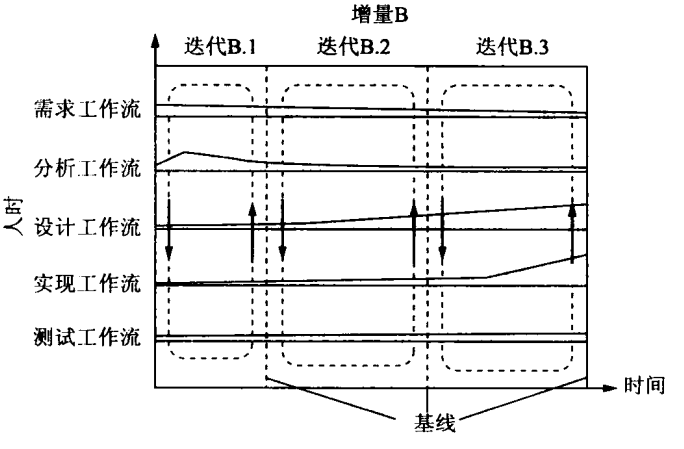


图 2-5　图 2-4 中迭代 - 增量生命周期模型增量 B 的 3 次迭代

再次强调，图 2-5 并未试图说明每次增量确切包含 3 次迭代。迭代的次数因增量而异。图 2-5 的目的是给出每次增量迭代，并重申每次迭代都会涉及几乎所有的 5 个工作流（需求工作流、分析工作流、设计工作流、实现工作流和测试工作流，包括相关的计划和文档），当然每个工作流在每次迭代被涉及的比例会有所不同。

如前所释，图 2-4 反映的是每个软件产品开发过程中内在的增量。图 2-5 则具体给出每次增量中所包含的迭代。特别地，相应于一个大的实现，图 2-5 明确了其所包含的 3 个连续迭代步骤。具体而言，迭代 B.1（如最左边的圆角短划线框所示）包含需求、分析、设计、实现和测试工作流。迭代过程一直持续，直到与 5 个工作流对应的制品令人满意为止。

接下来，迭代 B.2 中给出了所有 5 个制品的迭代过程。如图 2-5 所示，第二次迭代在本质上与第一次迭代相同，只不过需求制品得以改进，并依次激发分析制品及其他制品的改良。第三次迭代也与此类似。

在增量 A 初期启动迭代和增量过程，并持续下去，直到增量 D 结束。而后，在客户计算机中安装完整的软件产品。

2.6　Winburg 小型案例研究再探

对应于 Winburg 小型案例研究（图 2-2），结合考虑迭代 - 增量模型，图 2-6 给出其进化树模型（其中没有给出测试工作流，因为进化树模型假定测试是一个持续性过程（见 1.7 节））。图 2-6更多侧重于理清增量的本质：

- 增量 A 相应阶段 1，增量 B 相应于阶段 2，依此类推。
- 从迭代 - 增量模型的角度来看，其中的两次增量并未包含所有 4 个工作流。具体而言，增量 B（阶段 2）仅包含实现工作流，增量 C（阶段 3）仅包含设计工作流和实现工作

流。迭代－增量模型并不要求在每次增量期间执行所有的工作流。

- 进一步，对于图2-4，需求工作流的绝大部分在增量A和增量B中执行，但在图2-6中，需求工作流主要位于增量A和增量D。图2-4中，分析工作流主要位于增量B，然而在图2-6中，分析工作流主要在增量A和增量D中执行。由此说明，图2-4和图2-6都没有表示出每个软件产品的开发方式，而只是针对特定软件产品给出开发方法，以强调基本的迭代和增量。
- 至2-6的增量B（阶段2）中，实现工作流的小份额及突然终止表明，实现未被完成。虚线所表示的片断指明尚未完善的实现工作流部分。
- 进化树类型中的3条规划线箭头表明，每次增量都包含了对前一次增量的维护。在此例中，第二次和第三次增量进行的是纠错维护，即每次增量都纠正了前一次增量中的错误。据此，通过用单精度变量取代双精度变量，增量B（阶段2）纠正了实现工作流。通过使用更为快速的图像识别算法，增量C（阶段3）纠正了设计工作流。由此，相应时间需求得到满足。在实现工作流中也必须对此作出相应的修改。最后，增量D（阶段4）中需求发生改变，以保证改进的总精确度，实现圆满维护。然后依此，在分析工作流、设计工作流和实现工作流中作出相应的修改。

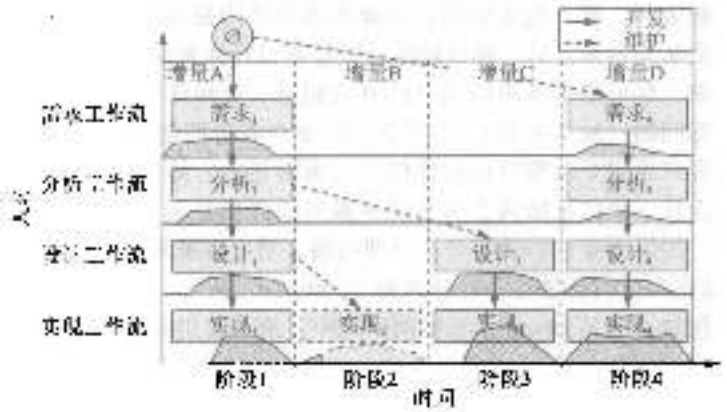


图2-6 结合考虑迭代－增量模型的Winburg小型案例研究进化树生命周期模型

2.7 迭代和增量的风险及其他

迭代和增量的另一种处理方式是将项目整体分解为较小的小型项目（或增量），每个小型项目都对需求、分析、设计、实现和测试制品进行扩展。最终得到的制品一起组成了完整的软件产品。

实际上，每个小型项目所包含的并不仅仅是对制品的扩展，还必须得验证每个制品的正确性（测试工作流），并对相应制品作出任何必要的修改。验证和修改的过程还需要进一步的验证和修改，依此类推，其本身就是一个迭代过程。这一过程持续下去，直到开发团队得到所有当前小型项目（或增量）所包含的制品。此时，再进一步处理下一次增量。

比较图2-3（瀑布模型）和图2-5（增量B内部的迭代）可知，每次迭代都可以看作是一个小而完整的瀑布模型。也就是说，在每次迭代期间，开发团队成员都会就软件产品的某一特定部分执行需求、分析、设计和实现过程。从这一角度看，图2-4和图2-5的迭代－增量模型可以看作是瀑布模型的连续序列。

迭代－增量模型具有诸多优点：

1）可多次验证软件产品的正确性。每次迭代都包含了测试工作流，因此，每次迭代都为验证迄今已开发的所有制品提供另一次机会。如图 1-5 所示，错误检测及纠正的时间越晚，成本就越高。与瀑布模型不同，迭代－增量模型中多次迭代的每一次都进一步提供机会，来找出错误并进行纠正，从而也节省了费用。

2）在生命周期中，能相对早地确定底层体系结构的鲁棒性。软件产品的*体系结构*（architecture）包含各类组件制品，以及将它们组装在一起的方式。这与一座教堂的结构相类似，可以描述为罗马式、哥特式或巴洛克式等。类似地，软件产品的体系结构可以是面向对象的（第 7 章）、管道和过滤器模式的（UNIX 或 Linux 组件）或是客户－服务器模式（包含一个为客户计算机网络提供文档存储的中心服务器）的。通过迭代－增量模型开发的软件产品，其体系结构具有不断扩展（以及如有必要，易于改变）以适应进一步增量的特性。处理这类扩展和改变而不致溃败的特性称为*鲁棒性*（robustness）。鲁棒性是软件产品开发期间的一个重要特性，在交付后维护期间，这一属性至关重要。因此，如果某个软件产品交付后要经受长达 12 年、15 年或更多年限的维护期，其底层体系结构务必是鲁棒的。当采用迭代－增量模型时，体系结构是否鲁棒很快就能显现出来。例如，在集成第三次增量时，如果已开发得到的软件显然必须彻底重组，并且其大部分要重新编写，那么毫无疑问，这样的体系结构是不够鲁棒的。客户必须确定，是取消这一项目，还是从头再来。另一种可能性就是重新设计体系结构，使其具备更好的鲁棒性，而后在下一次增量前，尽可能多地重用当前已有的制品。鲁棒体系结构之所以如此重要，另一个原因在于移动目标问题（见 2.4 节）。几乎所有的客户需求都会发生改变，要么因为客户组织的成长，要么因为客户总想要改变目标软件应该完成的任务。体系结构越鲁棒，软件改变的弹性就越大。当然，设计一个体系结构去应对过多激烈的变化是不太可能的。但是，对于一个鲁棒的体系结构而言，如果所要求的改变处在合理范围，则可以被集成进来而不需要彻底重构。

3）迭代－增量模型使得能够*尽早降低风险*（mitigate risk）。在软件开发与维护过程中，风险是难以避免的。例如，在 Winburg 小型案例研究中，最初使用的图像识别算法不够快速。这往往存在风险，致使最终的软件产品不能满足时间上的约束。增量开发软件产品，使得能够在生命周期早期就降低这类风险。例如，假设要重新构建一个局域网（LAN），而已有的网络硬件已不能适应新的软件产品，那么开始的一两次迭代应当侧重于开发软件对外的接口部分，以便与网络硬件相交互。如果解除了开发人员的忧虑，网络具备了必要的能力，则开发人员可以继续这一项目，并相信风险已经降低。另一方面，如果网络真的不能处理新 LAN 所产生的额外负载，则会在生命周期早期就向客户汇报，而此时仅仅花费了很少的预算。由此，客户可以决定，是取消该项目、扩展已有网络的能力、购买新的且功能更为强大的网络设备，还是采取其他措施。

4）任何时候都能获得软件的工作版本。假设采用图 2-1 中的理想生命周期模型来开发软件产品，则仅能在项目末期得到软件产品的工作版本。相反，当采用迭代增量生命周期模型时，在每次迭代结束后，都会存在一个工作版本，与完整目标软件产品的某些部分相对应。客户和有意向的用户可以用此版本进行实验，以确定需要作何种修改，确保未来完整的实现能满足他们的需求。在下一次增量时就可以施行这些改变，然后客户和用户可以再次决定是否需要进一步修改。与此相应的另一种做法是，交付软件产品的不完全版本，用于实验，而且也便于在客户组织内部对新软件产品进行推介。改变几乎总会被认为是一种威胁。通常，很多用户都担心，在工作场所引入新软件产品会使他们让位于计算机，而导致失业。因而，如果引入软件产品的过程足够缓慢，则能带来两个好处：其一，消除将被计算机取代的恐慌，当然，这种恐慌是可以理解的；其二，如果通过几个月时间来逐步介绍复杂软件产品的功能，而不是一次性引入，

则能很容易地学会这些功能。

5）有采用迭代、增量生命周期模型的实验证据。对于 2004 年所完成的项目[Hayes，2004]，图 1-1 中的饼形统计图给出了来自 Standish Group 研究报告的结果。实际上，每两年就会出一份这样的报告（即所谓的 CHAOS 报告，详见备忘录 2.2）。图 2-7 给出了从 1994 年到 2004 年的结果。成功产品的百分率从 1994 年的 16% 稳步增加到 2002 年的 34%，但 2004 年又降低到了 29%。在 2002 年[Software mag.com，2004]和 2004 年[Hayes，2004]的报告中，项目成功的因素之一是使用了迭代过程。（2004 年成功项目百分比下降的原因包括项目规模大于 2002 年的项目、使用瀑布模型、用户参与度低，高级执行官的支持不足[Hayes，2004]。）

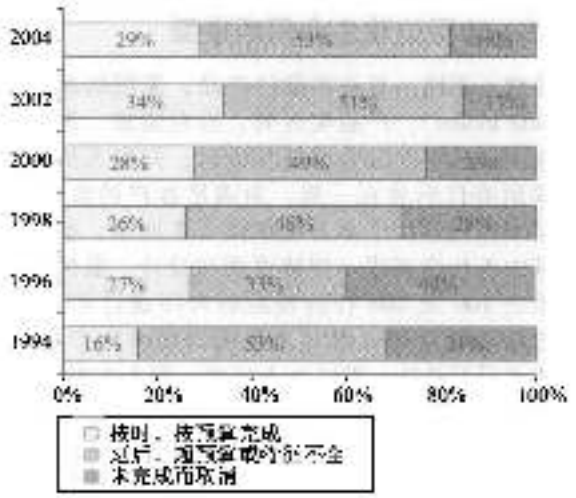


图 2-7　1994 年至 2004 年 Standish Group 研究报告的结果

备忘录 2.2

术语"CHAOS"为首字母缩写词。由于某些未知的原因，Standish Group 将这一首字母缩写词列为高度机密。他们声称[Standish，2003]：

知道 CHAOS 字母含义的仅包括 Standish Group 中的少数几个人，以及参与调查的 360 个人中部分收到 T 恤并加以保存的人员。

2.8　管理迭代与增量

初看起来，图 2-4 和图 2-5 的迭代－增量模型完全混乱无序，与瀑布模型（图 2-3）从需求到实现的有序过程不同，在迭代－增量模型中，开发者好像可以随心所欲，例如，在上午编码，午饭后花 1～2 小时设计，然后在下班之前花半个小时进行规格说明。实际并非如此，迭代－增量模型与瀑布模型一样受到严格的监控。如前所述，使用迭代－增量模型开发软件产品等同于开发一组更小的软件产品，而这些小软件产品都采用瀑布模型开发。

具体来说，如图 2-3 所示，采用瀑布模型开发软件产品意味着从整体上对软件产品逐步（按序）实施需求、分析、设计和实现阶段。如果出现问题，则依从图 2-3 中的反馈循环（虚线箭头），即执行认证（维护）。然而，如果用迭代－增量模型开发同样的软件产品，则产品被看成是增量的集合。对于每次增量，依次（按序）重复实施需求、分析、设计和实现阶段，直至不再需要进一步迭代为止。换言之，项目作为整体被分解为一组瀑布型小项目。在每个小项目中，按需实施迭代，如图 2-5 所示。因此，先前提及迭代－增量模型与瀑布模型一样严格受控，其原因在于，迭代增量模型就是连续多次运用瀑布模型。

2.9　其他生命周期模型

现在考虑与面向对象范型一起使用的众多其他的生命周期模型。首先从臭名昭著的边写边改模型开始。

2.9.1　边写边改生命周期模型

非常不幸地，有许多软件产品，采用的就是所谓的边写边改生命周期模型（code-and-fix life-cycle model）。产品实现时，没有需求，没有规格说明，也不尝试进行任何设计，开发人员只是简单地将代码放在一起。为满足客户的需求，工作必须重做多次。该方法如图 2-8 所示，很明显，其中不包含需求、规格说明和设计。虽然对于长度在 100 或 200 行的较短的程序设计而言，这一方法运转良好，但对于有相当规模的产品，边写边改模型总体上难以令人满意。图 1-5 表明，如果是在需求、分析或设计阶段对软件产品进行修改，则成本较低；但如果产品已经编码，或是更为糟糕地，产品已经交付并安装到客户计算机中，此时再做改变，则高昂的成本将令人难以接受。因此，与通过合理的规格说明和细致的设计来生产产品相比，边写边改方法的成本实际上要大得多。此外，由于没有规格说明或设计文档，产品维护变得非常困难，回归错误出现的可能性也大大增加。为不致身陷边写边改方法的囹圄，在产品开发启动前，选择一个合适的生命周期模型是必要的。

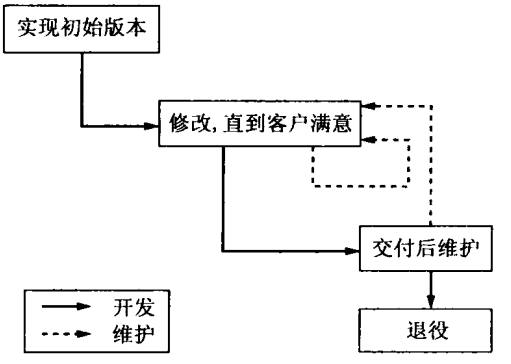


图 2-8　边写边改生命周期模型

遗憾的是，有太多的项目采用的是边写边改模型。在单纯通过代码行来衡量项目进展的组织中，这一问题尤为突出，因为软件开发团队成员从项目第一天开始，就被要求编写尽可能多的代码。边写边改模型是开发软件最为容易的方法，但也是迄今最为糟糕的方法。

2.2 节曾给出一个简化的瀑布模型，现在来更为详细地研究这类模型。

2.9.2　瀑布生命周期模型

瀑布生命周期模型首先由 Royce［1970］提出。与图 2-3 简化的瀑布模型一样，图 2-9 给出了产品开发时用于维护的反馈循环，以及与交付后维护相对应的反馈循环。

关于瀑布模型，重要的一点是，在完成了阶段的完整文档并且软件质量保证（SQA）小组验收了该阶段的产品之后，一个阶段才算完成。对于修改，如果因反馈循环，必须修改先前阶段的产品，则只有在阶段文档已作修改并且经由 SQA 小组验证通过后，先前阶段才被视为完成。

瀑布模型中的每个阶段都隐含着测试。测试并未作为只在产品构建之后才执行的独立阶段，也并非仅在每个阶段末期才执行。相反，如 1.7 节所述，测试应该在整个软件过程中持续执行。尤其是在维护阶段，必须确保产品的修正版本仍能处理前期版本所做的工作（同时保证处理的正确性（回归测试）），并且还得满足由客户所提出的任意新的需求。

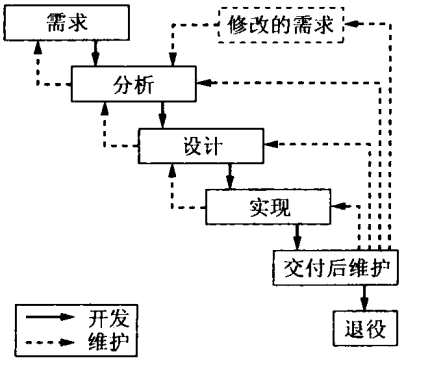


图 2-9　完整的瀑布生命周期模型

瀑布模型有诸多优点，其中包括强制约束方法，即约定每个阶段都得提供文档，并要求每个阶段（包括文档阶段）的所有产品都由 SQA 仔细检测。然而，瀑布模型是文档驱动的，这实际上也是一个弱点。为便于理解，考虑如下两个稍显奇特的场景。

第一个场景，Joe 和 Jane Johnson 夫妇决定修建一座房子。他们去咨询一位建筑师。建筑师没有给他们提供草图、规划以及可能的比例模型，而是给了他们一份长达 20 页的文档，通过非

常专业的术语对房子进行了描述。Joe 和 Jane 先前都没有过任何建筑经验，并且难以看懂这一文档，不过，他们还是非常热切地签收了文档，并表态："就这样，开始建房子吧！"

第二个场景，Mark Marberry 通过邮购方式购买衣服。公司没有发给 Mark 衣服的图片和可用布料的样本，而是寄过来关于产品裁减和布料信息的书面描述。而后，Mark 仅基于此书面描述，订购了一套衣服。

上述两个场景虽然不同，却与通过瀑布模型开发软件的本质相似。开发过程起始于规格说明。通常，规格说明文档很长、很详尽并且阅读起来相当乏味。一般客户往往不具备阅读软件规格说明的经验。不仅如此，规格说明文档还经常会采用客户所不熟知的风格编写。如果采用形式化（数学）规格语言（如 Z［Spivey，1992］）来编写规格说明，则阅读起来困难会更大。但是，不管是否真的理解规格说明文档，客户都会签署文件。在许多方面，Joe 和 Jane Johnson 夫妇同意按照不能完全了解的书面描述来建造房子和客户同意根据仅部分是可理解的规格说明文档来开发软件，这两者之间几乎没什么差别。

Mark Marberry 和他邮购衣服的方式看起来好像相当奇特，但这正是采用瀑布模型时，在软件开发过程中所可能发生的事情。仅当整个产品的编码完成之后，客户才能初次看到某个工作版本。不可避免地，软件开发者总是害怕客户这样说："我清楚这是按我的要求开发的，但它并非是我真正所需要的。"

问题出在哪儿？客户依据规格说明文档的描述所理解的产品，与实际得到的产品之间，存在着相当大的差异。规格说明仅存在于纸面上，因而，客户并不能真正理解产品本身可能会是什么样的。严格依据规格说明书面文档的瀑布模型方法可能导致所开发的产品并不能直接满足客户的真实需求。

应当指出，正如建筑师可以通过提供比例模型、草图和规划，来帮助客户去理解将要建造的房屋一样，软件工程师可以采用图例技术（如 UML 图（第 15 章）），与客户进行交流。例如，流图（产品的图形化描述）与工作版本之间往往会存在较大差异。规格说明文档在描述产品的时候，通常无法使客户作出判定：是否所提议的产品满足了他或她的需求，对于这一问题，将在第 10 章和第 11 章中给出解决方案。

现在来研究在面向对象软件工程中所用到的快速原型方法，在 10.14 节将会再次提及。

2.9.3　快速原型生命周期模型

快速原型（rapid prototype）作为一种工作模型，其在功能上等价于产品的某个子集。例如，如果目标产品需要处理应付、应收和库存数据，则快速原型法开发的产品可能可以从屏幕获取数据、打印报告，但不能更新文档，也不能处理错误。对于要在某个方案中判定酶浓度的目标产品而言，快速原型法可能可以执行计算，显示结果，但不能校验输入数据的有效性和合理性。

图 2-10 表明，快速原型生命周期模型的第一步是构建一个快速原型，让客户和未来用户接触快速原型并进行实验。只要客户对快速原型所实现的大多数需求感到满意，开发人员就可以编制规格说明文档，以在一定程度上保证产品满足客户的真实需求。

如图 2-10 所示，当构建好快速原型后，软件过程继续。快速原型模型的一个主要优点在于，产品开发过程中从快速原型到可交付产品的处理过程必定是线性的；在快速原型中不太可能需要瀑布模型（图 2-9）中的反馈循环。对于这一点，存在诸多原因。首先，开发团队成员使用快速原型来编制规格说明文档。由于通过与客户进行交互，已经验证了开发过程中的快速原型，可以合理地期望最终得到的规格说明

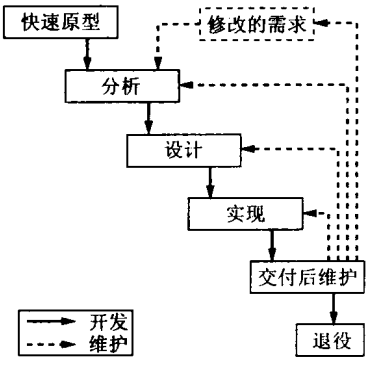


图 2-10　快速原型生命周期模型

文档是正确的；其次，考察一下设计阶段，尽管快速原型（非常恰当地说）是匆忙组装得到的，但设计团队能够从中有所洞悉，至少能知道一些“不能那样做”。再次重申，这里将不太可能用到瀑布模型中的反馈循环。

接下来要考虑实现。在瀑布模型中，对设计的实现有时会导致设计错误的呈现。实际上，在快速原型模型中，在开发软件产品的最初工作版本时就已经注意要减少在实现阶段或在实现之后对设计进行修改。尽管原型可能仅反映出完整目标产品的部分功能，但它已经为设计团队提供了某些启示。

如果客户已经验收了产品并且进行了安装，则启动交付后维护。取决于必须实施的特定维护任务，生命周期会再次进入需求、分析、设计或实现阶段。

快速原型的本质体现在“快速”上。开发人员应该尽可能快速地构建快速原型，以提高软件开发的整体速度。毕竟，快速原型的唯一作用是确定客户的真实需求；一旦确定，就丢弃快速原型实现，保留得到的经验，并在后续开发阶段加以利用。因此，快速原型的内部结构无关紧要。重要之处在于，快速构建原型，并进行快速修改，以反映客户需求。因此，速度是根本。

在统一过程中，将用到快速原型，第 10 章将对此加以介绍。

2.9.4　开源生命周期模型

近乎所有成功的*开源软件*项目都要经历两个非正式阶段。阶段一，某个人想要编写一个程序，例如，一个操作系统（Linux），一个网络浏览器（Firefox），或是一个 Web 服务器（Apache）。他（或她）首先开发一个初始版本，然后免费发布给任何想要的人。如今，这一过程经由 Internet 来完成，例如，SourceForge. net 和 FreshMeat. net 网站。如果有人下载了程序的初始版本，并认为该程序满足某种需要，他（或她）就会开始使用这个程序。

如果对编程非常感兴趣，项目就会逐渐进展到非正式阶段二。用户成为协同开发者，这包括一些用户报告缺陷，而另一些则给出修改这些缺陷的建议。随着程序功能的扩展，其他用户也会移植程序，以使它能够在其他操作系统或硬件平台上运行。关键在于，对于开源项目，每个人所投入的是他们的业余时间，并且是自愿的；他们并未从中获取报酬。

现在进一步研究第二个非正式阶段的 3 项活动：

1）报告并纠正缺陷的纠错性维护。

2）添加更多功能的完善性维护。

3）移植程序到新环境的适应性维护。

换言之，开源生命周期模型中的第二个非正式阶段仅包含交付后维护，如图 2-11 所示。实际上，本节第 2 段中的术语“协同开发者”相应地也应该是“协同维护者”。

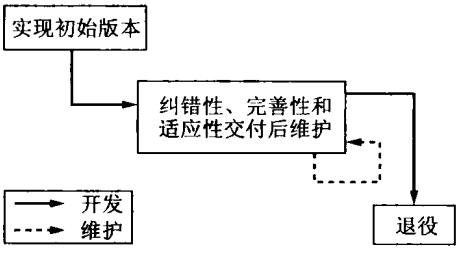


图 2-11　开源生命周期模型

在闭源与开源生命周期模型之间，存在着许多关键性差异：

- 闭源软件的维护和测试工作都由持有软件所有权的组织的雇员承担。有时，用户会提交缺陷报告。然而，这仅限于*故障报告*（报告观察到的不正确行为）。用户无权访问源代码，因此，他们不可能提交*差错报告*（报告源代码中的不正确之处以及纠正方法）。

  相反，开源软件的维护工作通常由没有报酬的志愿者承担。虽然鼓励用户提交缺陷报告，并且所有用户都可以访问源代码，但仅少部分人有此爱好同时具有必要的技能，他们可以投入时间去细读源代码并提交缺陷报告（“修复”）。因此，大多数缺陷报告实际上是故障报告。通常，有一个由专门维护人员组成的*核心小组*，负责管理开源项目。由非核心小组成员的用户组成*外围小组*，其中的一些成员会不时地提交缺陷报告。核心小组成员负责确保这些缺陷得以纠正。具体来说，当一份差错报告提交后，核心小组的某位成员会验证是否修复方案确实解决了问题，并且在源代码中进行适当的修改。当一

份故障报告提交后，核心小组的某位成员可以选择独自确定修复方案，或者选择将这一任务分派给其他志愿者，往往会分配给渴望更多参与到该开源项目中来的外围小组成员。不过，将修复安装到软件中的任务则严格地仅由核心小组成员来完成。

- 闭源软件往往一年只发布大约一个新版本。每个新版本发布之前，都经由软件质量保证小组仔细检查，并运行各种类型的测试用例。

  相反，开源运动的格言是“尽早发布，经常发布”［Raymond，2002］。也就是说，只要准备妥当，核心小组就会发布开源项目的新版本，可能仅与前一版本的发布相差一个月，甚至仅仅一天。新版本的发布经由最小限度的测试，并假定外围小组成员将会执行更多扩展性测试。在某个新版本发布的一两天之内，可能就会有成百上千用户安装这个新版本。这些用户不都会运行测试用例。他们在计算机上使用新版本的过程中，会碰到失败，这时他们可以通过电子邮件进行报告。通过这种方式，新版本中的错误（以及前一版本中隐藏更深的错误）将逐步呈现，并得到纠正。

比较图 2-8、图 2-10 和图 2-11 可以看到，开源生命周期模型与边写边改模型和快速原型模型有相同的特征。在所有这三种生命周期模型中，都开发了初始工作版本。对于快速原型模型，初始版本将被丢弃，并在编码前编写目标产品的规格说明，并进行设计。而对于边写边改和开源生命周期模型，初始版本一直被重写，直到最终形成目标产品。相应地，在开源项目中，通常不存在规格说明或设计阶段。

不要忘记，规格说明和设计文档都是非常重要的，但是某些开源项目怎么又会如此成功呢？在闭源开发环境下，通常某些软件专业人员具备优异的技能，虽然也有一些稍显逊色（参见 9.2 节）。但开发开源软件的挑战性，也已经吸引了一些出色的软件专家。也就是说，尽管缺乏规格说明或设计，但开源软件仍然能够成功，只要开发项目的那些人技能超群，以至无须规格说明和设计就能有效地实现功能。然而，不管核心小组成员具备怎样的能力，开源产品最终的下场都将是成为不再可维护的产品［Yu，Schach，Chen，and Offutt，2004］。

开源生命周期模型的实用性是有限的。一方面，开源模型在某些基础性软件项目中使用得非常成功，例如，操作系统（Linux、OpenBSD、Mach、Darwin）、Web 浏览器（Firefox、Netscape）、编译器（gcc）、Web 服务器（Apache）或数据库管理系统（MySQL）。另一方面，要在某个商业组织中使用开源开发，其结果难以想像。开源软件开发的关键之一在于，核心小组和外围小组的成员都应是所开发软件的用户。最终的结果就是，除非大量用户都认为目标产品对他们有用，否则就难以在产品开发中应用开源生命周期模型。

编写这本书时，在 SourceForge. net 和 FreshMeat. net 中包含 175 000 多个开源项目。大约有一半的项目从来没有吸引任何团队为之工作。当然，那些已经启动的项目，绝大多数都已不再可能完成，并且也不可能有进一步的进展。但是，采用开源模型，有时却能取得难以置信的成功。上一段所列的开源产品，使用就非常广泛，它们中大多数都经常被使用，并且大约有几百万用户。

第 4 章将介绍开源软件项目团队组织方面的内容，其中将解释开源生命周期模型成功的一面。

2.9.5　敏捷过程

基于迭代 - 增量模型的极限编程（extreme programming）［Beck，2000］在某种程度上是一种颇受争议的新的软件开发方法。首先，软件开发团队确定客户期望产品能支持哪些特征（情况，story）。对于每个特征，开发团队会告知客户实现该特征需要耗费的时间及成本。这一步骤相应于迭代 - 增量生命周期模型（图 2-4）中的需求和分析工作流。

在每次后续开发中，客户选定需要包含的特征，其依据是成本效益分析（参见 5.2 节），即基于开发团队所估算的时间和成本，以及该项特征为客户业务带来的潜在效益。所提议的开发将被分解成更小的部分，称之为任务。程序员首先为某一任务编制测试用例，这便是所谓的测试驱动开发（Test-Driven Development，TDD）。两名程序员在同一台计算机上协同工作（结对编程，pair

programming）[Williams, Kessler, Cunningham and Jeffries, 2000]，实现这一任务，并确保所有测试用例正确运行。两名程序员每 15 或 20 分钟就交换着编码；未在进行编码的程序员则在同伴输入时，仔细检查代码。然后将任务集成到产品的当前版本。理想情况下，实现并集成一个任务所耗费的时间应该不超过几个小时。通常，由多个结对并行实现任务，因此，任务的集成必定是不断进行的。团队成员尽可能每天更换一起编码的同伴，向其他团队成员学习，以此提高各自的技能水平。保存任务所采用的 TDD 测试用例，并在每次后续集成测试中加以利用。

通过实践，发现结对编程存在着某些缺点 [Drobka, Noftz and Raghu, 2004]。例如，结对编程要求时间上不能中断，但对于一些软件专业人员来说，要找到连续 3 ~ 4 个小时的时间有点困难。此外，结对编程并不能总是运作良好，如一些程序员比较自卑或是自负，或者两个程序员都经验不足。

某种程度上，极限编程（XP）有许多不同于通常软件开发方式的特性：

- XP 团队的计算机位于某个被隔断分割的大房间的中央。
- 客户代表一直与极限编程团队一起工作。
- 没有程序员能够连续工作两个星期。
- 不需要规格说明。但是，所有 XP 团队成员都负责需求、分析、设计、编码和测试。
- 在开发出各个模块之前，不存在总体设计。相反，在产品开发过程中，就对设计进行修改。这一过程称为*重构*（refactoring）。只要测试用例尚未运行，就重组代码，直到团队满意地认为设计简洁、直观，并能正确运行所有测试用例为止。

现今，与极限编程相关的两个首字母缩写词是 YAGNI（“你不会需要它”）和 DTSTTCPW（“做可能起作用的最简单的事情”）。换言之，极限编程的原则就是使特征数目最小化，不让开发的产品超出客户实际所需。

极限编程是众多新型范型中的一种，这类新范型统称为*敏捷过程*（agile processes）。2001 年 2 月，17 位软件开发人员（后来称为敏捷联盟）聚集在犹他州的滑雪胜地，发布了《敏捷软件开发宣言》（Manifesto for Agile Software Development）[Beck et al., 2001]。许多参加人员在此之前都发表了各自主张的软件开发方法，包括极限编程 [Beck, 2000]、Crystal [Cockburn, 2001] 和 Scrum [Schwaber, 2001] 等。因此，敏捷联盟没有规定某一特定的生命周期模型，而只是给出一组各种软件开发方法共通的基本原则。

与几乎所有其他的现代生命周期模型相比，敏捷过程的特点是极少强调分析和设计。在生命周期模型中，实现阶段开始得更早，因为开发工作软件被认为要比编制详细文档更为重要。敏捷过程的另一主要目标是对需求修改作出快速响应，因此，与客户之间的协作显得尤为重要。

宣言中的原则之一是要经常交付工作软件，理想情况下，每 2 ~ 3 周提交一次。为此，采用*时间定量*（timeboxing）[Jalote, Palit, Kurien and Peethamber, 2004] 法，它是一种已使用多年的时间管理技术。对于某个任务，给出指定的时间，而后，团队成员在规定时间内尽最大努力完成任务。在敏捷过程中，通常为每次迭代设置 3 周的时间。一方面，这可以使客户相信，每 3 周就可以给新软件版本增加功能。另一方面，使开发人员知道他们将有 3 周（仅此而已）时间交付新的迭代，在此期间不会有客户干扰；一旦客户选定了某次迭代所要完成的工作，就不能再改动或增加。但是，如果在定量时间内不可能完成整个任务，则可能要减少工作量（缩小范围）。换句话说，敏捷过程要求确定的时间，而非确定的特征。

敏捷过程的另一个特征就是要在每天固定的时间开一个短会。所有团队成员都必须参加会议。所有参与人员站着围成一圈，而不是围着桌子就坐，这有助于保证会议持续时间不会超过约定的 15 分钟。每个团队成员依次回答 5 个问题：

- 从昨天会议到现在我做了些什么？
- 今天我正在做什么？
- 要顺利完成任务，存在什么问题？

- 我们忽视了什么？
- 我学到的了什么，以及可以同团队分享些什么？

站立会议是为了发现问题，而不是解决问题；后续会议才是解决问题之道，而且最好是在站立会议之后随即举行。与时间定量法一样，站立会议是目前用于敏捷过程中的一种成功管理技术。所有敏捷过程都具有两个基本原则：交流和尽可能快地满足客户需求，而时间定量迭代和站立会议则是这两个原则的具体实现。

敏捷过程已经成功运用于许多规模较小的项目。然而，敏捷过程的运用还不够广泛，还不足以确定这一方法是否符合早期所作出的承诺。而且，即使能够证明敏捷过程对小型软件产品有诸多好处，但这也不一定意味着它适用于中型或大型规模的软件产品，以下对此进行解释。

许多软件专业人员对敏捷过程用于中型或大型软件产品持怀疑态度［Reifer，Maurer and Erdogmus，2003］，为便于理解，考虑下述由 Grady Booch［2000］进行的比拟。任何人都可以用几块木板搭建一个犬舍，但如果没有详细规划，要建造包含三间卧室的房子就会显得有勇无谋。毕竟，建造含有三间卧室的房子需要垂直测量、布线和吊顶等相关技能，并且还得进行勘察。（也就是说，能够开发小型软件产品并不必定意味着就有能力去开发中型规模的软件产品。）进一步讲，摩天大楼与 1 000 个犬舍的总高度一样，但这并不意味着就能够通过逐个叠加 1 000 个犬舍来建造摩天大楼。换句话说，较之于将小型软件产品拼在一起，大型软件产品的开发需要更为专业和更为娴熟的技能。

要判定敏捷过程是否真正成为软件工程中的一项重要突破，其关键取决于交付后维护的未来成本（参见 1.3.2 节）。也就是说，如果敏捷过程减少了交付后维护的成本，则 XP 和其他敏捷过程将会被广泛采用。另一方面，重构是敏捷过程的一个内在环节。如先前所释，软件产品不是以一个整体来进行设计的，而是逐步开发得到的，并且不管任何原因不满意当前设计，都会对代码进行改造。这一重构过程，在交付后维护阶段仍将继续。当验收测试时，如果软件产品的设计是可扩展的并且易于修改，则以低成本就可轻易完成完善性维护。但是，如果是为了添加功能而重构设计，则软件产品的交付后维护成本将高得令人难以接受。敏捷过程开发方法比较新，仅有少量实验数据，并且几乎没有维护方面的数据。但是，初步的数据表明，重构会消耗总成本的较大份额［Li and Alshayeb，2002］。

尽管如此，实验结果表明，敏捷过程的某些特征具有较好的可行性。例如，Williams、Kessler、Cunningham 和 Jeffries［2000］已经证明，结对编程可以在更短的时间内开发得到更高质量的代码，以及更高的工作效率。因此，即使从整体而言敏捷过程是令人失望的，但其中的一些特征还是可能为未来的主流软件工程实践所采纳。

《敏捷软件开发宣言》主要宣称，敏捷过程优于更为规范化的过程（如统一过程，参见第 3 章）。持怀疑态度的人认为敏捷过程的支持者与计算机黑客无异。但是，也存在中间派别。实际上，两种方法并非不可兼容；将敏捷过程中已证实可行的特征融合到规范过程的框架中，这是可能的。有些书（如 Boehm 和 Turner［2000］）描述了两类方法的集成。

总之，当客户需求尚无法确定时，敏捷过程是小型软件产品开发的有效方法，另外，敏捷过程的某些特征可以有效地运用于其他生命周期模型中。

2.9.6　同步稳定生命周期模型

微软公司是世界上最大的 COTS 软件开发商。大部分软件包都是采用某种迭代 - 增量模型来开发，这类模型称为同步稳定生命周期模型（synchronize-and-stabilize life-cycle model）［Cusumano and Selby，1997］。

在需求分析阶段，与软件包的众多潜在客户会谈，并从中抽取出对客户而言优先级最高的特征列表。随即草拟规格说明文档。然后将工作划分为 3 ~ 4 个模块。第一模块由最重要的特征组成，第二模块包含次重要的特征，依此类推。每个模块都由许多小型团队并行开发。在每天

工作行将结束时，所有团队相互同步（synchronize），即他们将部分完成的组件集成在一起，并对得到的产品进行测试和调试。在每个模块完成时，实施稳定（stabilization）操作。任何迄今为止被检测到的剩余错误都会被修正，然后冻结（freeze）模块，也就是说，不会对规格说明作进一步的修改。

重复同步步骤，确保各个组件都总能集成在一起。定期执行部分开发得到的产品，其另一个优点在于，开发人员能尽早了解软件产品的运作，而且如果必要的话，还可以在模块开发过程中修改需求。即使初始规格说明还不完整，也可以采用这种生命周期模型。在 4.5 节讨论团队组织细节时，将进一步研究同步稳定模型。

最后讨论螺旋模型，它是风险驱动模型。控制风险是统一过程的一个必要环节。

2.9.7　螺旋生命周期模型

正如 2.5 节所述，在软件开发过程中总会包含风险。例如，在软件产品文档化完成前，关键人员可能辞职。软件产品所严重依赖的硬件制造商可能破产。在测试和质量保证方面，也或多或少存在风险。在已经花费成百上千美元开发一个重要软件产品之后，新技术的突破可能致使整个产品毫无价值。某个组织可能研究并开发了一个数据库管理系统，但是，在产品推向市场之前，竞争对手却发布了价格更为低廉且功能相同的产品。当实施集成时，软件产品的组件之间可能无法兼容。出于种种原因，软件开发人员总是尽可能最小化风险。

最小化某类风险的方法之一是开发一个原型。如 2.9.3 节所述，为减少所交付的产品不满足客户真实需求的风险，一种方法就是在需求阶段，开发一个快速原型。在后续其他阶段，则采用其他种类的原型。例如，为了能通过长途网络进行路由呼叫，电话公司可能会设计一个效率明显更高的新算法。如果产品已经实现，但未达到预期效果，那么电话公司浪费了产品开发的费用。此外，客户可能感到生气或觉得不便，而将业务交给其他公司。避免这一结果的方法是，开发一个概念验证原型，仅处理呼叫路由，并在仿真器上进行测试。据此，不会干扰实际系统，并且实现成本也仅限于路由算法，电话公司可以决定是否需要开发一个完整的网络控制器，以结合采用新算法。

概念验证原型并非快速原型，不能如 2.9.3 节所描述的那样，确定是否已经完全搞清楚了需求。相反，它更像是一个工程原型，即一个用于测试可行性的比例模型。如果开发团队关心所提议的软件产品的某个特定部分，则可开发一个概念验证原型。例如，开发人员可能会关心某一特定部分的计算是否执行得足够快。此时，开发一个原型，以测试该项计算所需的时间。或者，开发人员担心屏幕上的字体对一般用户而言太小了，容易导致眼部疲劳。在这种情况下，则开发一个原型，以测试不同屏幕显示，并通过实验来判定用户是否会因字体较小而感到不太舒服。

通过使用原型和其他方法来最小化风险的思想，同时也是螺旋生命周期模型（spiral life-cycle model）的基本思想［Boehm，1988］。该生命周期模型可以简单地看作为，在瀑布模型中的每个阶段之前，先进行风险分析，如图 2-12 所示。在每个阶段开始前，先试图降低（控制）

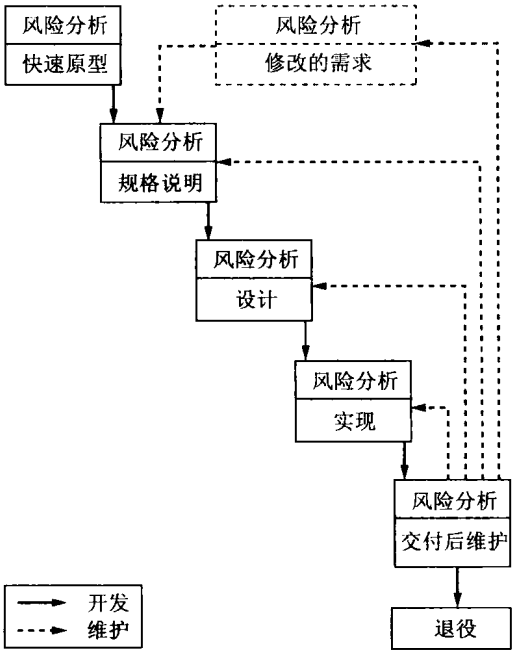


图 2-12　螺旋生命周期模型的简化版本

风险。如果在那个阶段不能降低所有主要的风险，则立即终止项目。

有效使用原型可以提供与某类风险相关的信息。例如，通过开发原型，并评测该原型能否具备必要的性能，可测试时间约束条件是否满足。如果原型给出了产品相关特征所对应功能的精确表示，那么通过原型评测，开发人员应该能及时获知时间约束是否满足。

一些其他方面的风险则无法通过原型进行评测，例如，未能雇用到产品开发所必需的软件人员，或者在项目完成前关键人员辞职等。另一潜在的风险是，某个团队可能无法胜任特定的大型产品的开发。这好比成功建造了家用住宅的承建商，可能无法胜任建造高层综合办公大楼。同样，小型与大型软件之间必然存在着本质差异，原型法对大型软件几乎没有用处。了解一个团队具备开发较小原型的能力，并不能降低风险，因为与开发大型软件相关的团队组织问题并未凸显。原型法无法用于评估的另一风险领域是硬件供应商的交付承诺。开发方所能够采用的策略就是判定该供应商的先前客户对其如何评价，但是，用过去的成绩评估将来的表现是毫无意义的。交付契约中必须设置惩罚条款，以确保供应商尽力按时交付重要硬件，但是，要是供应商拒绝签署包含这一条款的协议，该怎么办？即使有惩罚条款，也可能出现延迟交付的情况，并最终导致可能持续多年的法律诉讼。与此同时，所承诺的硬件无法送到将导致软件无法交付，最终可能是软件开发商破产。总之，原型法有助于减少某些方面的风险，但在另一些方面则只能部分缓解，而在其他方面则完全无助。

图 2-13 给出了完整的螺旋模型。径向表示迄今的累积成本，旋转角表示螺旋进展。螺旋的每一圈对应于一个阶段。阶段的起始（左上象限）确定该阶段的目标，选择是否达成目标，并对选择加以约束。经由这一过程，得到达成目标的策略。接着，从风险角度来分析这一策略。在某些情况下，借助原型开发，尽力降低每个潜在的风险。如果无法降低某些风险，则立即终止项目；然而，在某些情况下，也可能决定继续实施该项目，但规模上会大大缩小。如果已成功降低所有风险，则启动下一开发步骤（右下象限）。螺旋模型的象限与瀑布模型对应。最后，评估该阶段的结果，并计划下一阶段。

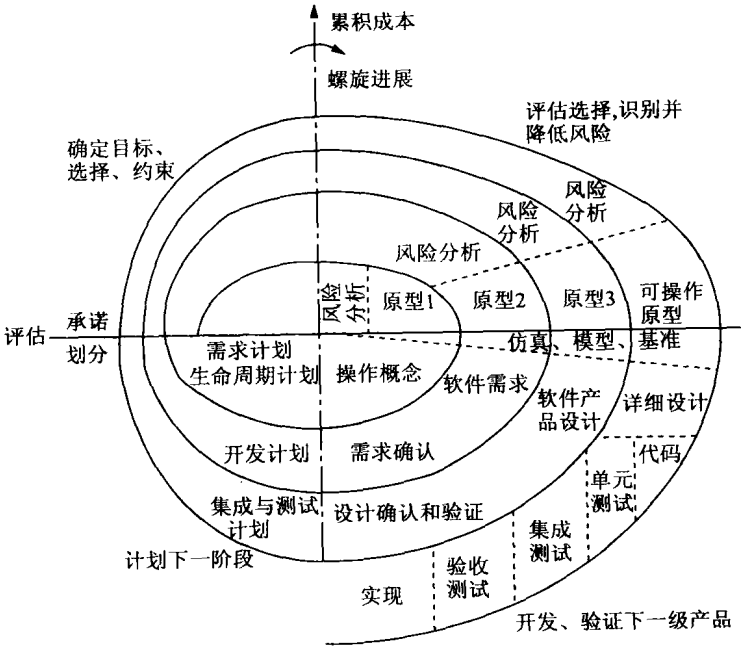


图 2-13　完整的螺旋生命周期模型

注：资料来源于［Boehm，1998］（©1988 IEEE）。

目前，螺旋模型已经成功应用于各类软件产品的开发。对一组25个使用螺旋模型并结合提高生产力的其他方法的项目开发结果的评估表明，每个项目的生产力水平较之先前至少提高50%，多数项目的生产力水平提高达100%［Boehm，1988］。为了判定在给定项目中是否应该使用螺旋模型，现在来评估一下螺旋模型的优点和缺点。

螺旋模型有许多优点。强调目标选择和条件约束，可支持已有软件的重用（参见8.1节），并将软件质量作为特定目标。此外，软件开发中的一个普遍问题在于，要确定何时对特定工作流的产品进行充分测试。在测试上花费过多时间就是浪费金钱，产品的交付也会被过度延迟。相反，如果测试不足，则交付的软件可能含有残留错误，这对开发人员而言，可不是愉快的事。测试不足或测试过量，都会招致风险，为解决这类问题，螺旋模型应运而生。在螺旋模型结构中，可能最为重要的就在于，交付后维护只是螺旋的另一个周期；交付后维护和开发之间不存在明显的界线。因此，即使再无知的软件人员也不会轻视交付后维护，因为它与开发具有相同的地位。

在应用螺旋模型时，存在一些限制。依目前形式，螺旋模型专门用于内部开发大型软件［Boehm，1988］。考虑一个内部项目，即项目开发方和客户同为组织成员。如果风险分析结果认为，项目应当终止，则可以很简单地把内部软件人员重新安排到不同的项目中。但是，一旦开发组织与外部客户签订了契约，则任何一方想终止契约都将引来违约诉讼。因此，只要存在关于软件的契约，在契约签订之前，客户方和开发方必须进行所有的风险分析，而在螺旋模型中并未对此作出要求。

螺旋模型的另一个限制与项目规模相关。特别地，螺旋模型仅适用于大型软件。如果执行风险分析的代价相当于整个工程项目的成本，或者执行风险分析将严重影响潜在利润，那么执行风险分析显得毫无意义。相反，开发方应当先判断风险有多大，如果要执行风险分析，其成本几何。

螺旋模型的最主要优点在于，它是风险驱动的，但是这也可能是它的缺点。除非软件开发方精确查明了可能的风险，并进行正确分析，否则就会陷入真正的危险之中，即在某个时候，开发团队可能认为一切顺利，而事实上项目正濒临灾难。仅当开发团队的成员都是能干的风险分析家时，管理层方能决定采用螺旋模型。

然而，与瀑布模型和快速原型模型一样，螺旋模型的主要弱点大致就在于，它假定软件开发的各个阶段是相互独立的。实际上，正如进化树模型（参见2.2节）或迭代－增量模型（参见2.5节）所演示的那样，软件开发是迭代和增量的过程。

2.10　生命周期模型的比较

至此，已经研究分析了9种不同的软件生命周期模型，并专门给出了它们各自的一些优点和缺点。边写边改模型应该避免使用（参见2.9.1节）。对于瀑布模型（参见2.9.2节），人们所知甚多，不单它的优点，其缺点也是众所周知的。瀑布模型存在特有的缺点，即交付后的产品可能并非客户真实所需，由此出现了快速原型模型（参见2.9.3节）。然而，仍然没有充分证据表明，在其他方面这种方法能胜过瀑布模型。在部分案例中，例如，在开发基础性软件时，开源生命周期模型（参见2.9.4节）不可置信地取得了成功。敏捷过程（参见2.9.5节）包容了诸多颇受争议的新方法，迄今，这些方法看似可行，但只限于小型软件。微软公司所采用的同步稳定模型（参见2.9.6节）取得巨大成功，但没有在其他公司中成功使用的案例可循。另一可选方案则是螺旋模型（参见2.9.7节），但是仅当开发人员在风险分析和风险处置方面训练有素时，才适宜采用。在实际软件开发中，进化树模型（参见2.2节）与迭代－增量模型（参见2.5节）是采用最多的。图2-14给出了这些生命周期模型的总体比较。

生命周期模型	优　点	缺　点
进化树模型（2.2 节）	贴近现实软件开发模型 等价于迭代 - 增量模型	
迭代 - 增量生命周期模型（2.5 节）	贴近现实软件开发模型 统一过程的基础	
边写边改生命周期模型（2.9.1 节）	适于不需维护的小程序	完全不适于重要程序
瀑布生命周期模型（2.9.2 节）	方法严谨 文档驱动	交付后产品可能不满足客户需求
快速原型生命周期模型（2.9.3 节）	确保交付后产品满足客户需求	尚未消除所有疑虑
开源生命周期模型（2.9.4 节）	在少部分案例中效果极好	适用性受限 通常不可行
敏捷过程（2.9.5 节）	当客户需求不明时效果良好	似乎仅适用于小型项目
同步 - 稳定生命周期模型（2.9.6 节）	能满足用户未来的需求 确保组件能成功集成	除微软公司外尚未推广
螺旋生命周期模型（2.9.7 节）	风险驱动	仅适于大型的内部产品 开发人员必须在风险分析和风险处置方面训练有素

图 2-14　本章所述生命周期模型的比较（含章节标识）

每个软件开发组织都应该依据组织结构、管理、员工和软件过程，选定一个生命周期模型，并根据当前所开发特定产品的特征适时地进行修改。这样一个模型要结合各种生命周期模型的适用部分，扬长避短。

本章回顾

对于软件开发的方式，理论（2.1 节）与实践上存在着重要差异。本章首先通过 Winburg 小型案例研究介绍进化树模型（2.2 节）。2.3 节给出了这一案例研究的实践经验，尤其是存在需求修改时。2.4 节进一步详细讨论与修改相关的问题，通过 Teal Tractors 小型案例研究，提出了移动目标问题。2.5 节特别强调迭代与增量在实际软件工程中的重要性，并给出了迭代增量模型。随后，2.6 节又重新分析了 Winburg 小型案例研究，阐明了进化树模型与迭代 - 增量模型的等价性。2.7 节给出迭代 - 增量模型的优点，特别是它能够尽早处置风险。2.8 节则讨论了迭代 - 增量模型的管理。2.9 节描述了众多现存的生命周期模型，包括边写边改生命周期模型（2.9.1 节）、瀑布生命周期模型（2.9.2 节）、快速原型生命周期模型（2.9.3 节）、开源生命周期模型（2.9.4 节）、敏捷过程（2.9.5 节）、同步 - 稳定生命周期模型（2.9.6 节）以及螺旋生命周期模型（2.9.7 节）。2.10 节比较了这些生命周期模型，并对特定项目给出了生命周期模型的选择建议。

延伸阅读材料

瀑布模型最早是在［Royce，1970］中提出的。［Royce，1998］一书的第 1 章再次对瀑布模型进行了分析。

关于快速原型的介绍，推荐的参考书是［Connell and Shafer，1989］。

同步稳定模型可参见［Cusumano and Selby，1997］的概述，以及［Cusumano and Selby，1995］给出的详尽描述。还可以从［McConnell，1996］中深入了解同步稳定模型。螺旋模型的解释可参阅［Boehm，1988］，将其应用于 TRW 软件生产力系统的实践则出现在［Boehm et al.，1984］。

［Beck，2000］介绍了极限编程，［Fowler et al.，1999］则详细描述了重构。《敏捷软件开

发宣言》可参阅［Beck et al.，2001］。对于各类敏捷方法，都有相应的书籍出版，包括［Cockburn，2001］和［Schwaber，2001］。敏捷方法的提倡者包括［Highsmith and Cockburn，2001；Boehm，2002；DeMarco and Boehm，2002；and Boehm and Turner，2003］，而［Stephens and Rosenberg，2003］则提出了反对敏捷方法的案例。［Mens and Tourwe，2004］对重构进行了综述。［Drobka，Noftz and Raghu，2004］描述了 XP 在 4 个任务关键型项目中的应用。将敏捷过程引入到当前仍使用传统方法的组织中可能会产生问题，这在［Nerur，Mahapatra and Mangalaraj，2005］中进行了讨论。2003 年 5/6 月的《IEEE Software》期刊上发表了数篇关于极限编程的论文，包括［Murru，Deias and Mugheddu，2003］和［Rasmusson，2003］，它们都对使用极限编程成功开发的项目进行了描述。［Erdogmus，Morisio and Torchiano，2005］则讨论了测试驱动开发的效果。2003 年 6 月的《IEEE Computer》期刊发表了几篇关于敏捷过程的论文；2003 年 5/6 月的《IEEE Software》期刊也发表了 4 篇关于敏捷过程的论文，特别是［Ceschi，Sillitti，Succi and De Panfilis，2005］和［Karlström and Runeson，2005］。

风险分析方面的描述参见［Ropponen and Lyttinen，2000；Longstaff，Chittister，Pethia，and Haimes，2000；和 Scott and Vessey，2002］。1997 年 5/6 月的《IEEE Software》期刊包含 10 篇关于风险管理的文章。

在［Jacobson，Booch and Rumbaugh，1999］中详细介绍了一种主流的迭代和增量模型。但是，在过去 30 年间，也相继提出过许多其他的迭代和增量模型，这在［Larman and Basili，2003］进行了详细列举。［Goth，2000］讨论了如何使用增量模型开发航空交通控制系统。［Bianchi，Caivano，Marengo and Visaggio，2003］则给出在重建遗留系统时迭代方法的使用。

目前已经提出了许多其他的生命周期模型。例如，Rajlich 和 Bennet［2000］提出了一个面向维护的生命周期模型。在 2000 年 7/8 月的《IEEE Software》期刊发表了许多关于软件生命周期模型的论文，其中，［Williams，Kessler，Cunningham and Jeffries，2000］给出一个实验，描述了作为敏捷方法组成之一的结对编程。

国际软件过程专题讨论会的论文集提供了许多关于生命周期模型的有用信息。［ISO/IEC 12207，1995］是广为接受的软件生命周期过程标准。

习题

2.1　用瀑布模型来表示 2.3 节中的 Winburg 小型案例研究。相较于进化树模型，其效果会更好还是更差？请进行解释。

2.2　在 Winburg 小型案例研究中，假设程序员一开始就采用了单精度数，请画出最终的进化树。

2.3　米勒法则与逐步求精之间有什么联系？

2.4　逐步求精与迭代或增量有对应关系吗？

2.5　工作流、制品、基线之间有何关联？

2.6　瀑布模型与迭代 - 增量模型之间有什么联系？

2.7　假设开发一个软件产品，确定 653.8231 的倒数并精确到小数点后 5 位。只要产品实现并通过测试，则可将其丢弃。该采用哪种生命周期模型？请给出理由。

2.8　假设你是一位软件工程顾问，一家皮靴制造与销售公司的财务副总来访。她想要你所在的组织为其开发一个产品来监控公司仓储状况，从皮革的购买开始，跟踪靴子生产、配货到各个店铺、销售的整个过程。对于此项目，你将依据什么标准来选定生命周期模型？

2.9　请列出习题 2.8 的软件开发过程中所存在的风险。你打算如何降低各个风险？

2.10　假设所开发的鞋类供应链仓储控制产品非常成功，你所在的组织决定将其重新编写成软件包，以便卖给那些制造并通过自有零售商销售产品的各类公司组织。因此，新产品必须具有可移植性，并且易于适应新硬件和/或操作系统。你为此项目选择生命周期模型采取的标准，与习题 2.7 中的标准

有何不同？

2.11　描述理想的可应用开源软件开发方法的产品种类。

2.12　描述在哪类情况下，不宜采用开源软件开发。

2.13　描述理想的可应用敏捷过程方法的产品种类。

2.14　描述在哪些类型情况下，不宜采用敏捷过程。

2.15　描述理想的可应用螺旋模型方法的产品种类。

2.16　描述在哪类情况下，不宜采用螺旋模型。

2.17　（学期项目）对于附录 A 中的 Osric 办公用品和装饰产品，你打算采用哪种软件生命周期模型？请给出理由。

2.18　（软件工程读物）由教师分发［Mens and Tourwe，2004］论文的复印件。对于敏捷过程，这篇论文给出了什么建议？

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 and Turner, 2003] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed,* Addison-Wesley Professional, Boston, MA, 2003.

[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.

[Ceschi, Sillitti, Succi, and De Panfilis, 2005] M. Ceschi, A. Sillitti, G. Succi, and S. De Panfilis, "Project Management in Plan-Based and Agile Companies," *IEEE Software* **22** (May/June 2005), pp. 21–27.

[Cockburn, 2001] A. Cockburn, *Agile Software Development,* Addison-Wesley Professional, Reading, MA, 2001.

[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.

[Drobka, Noftz, and Raghu, 2004] J. Drobka, D. Noftz, and R. Raghu, "Piloting XP on Four Mission-Critical Projects," *IEEE Software* **21** (November/December 2004), pp. 70–75.

[Erdogmus, Morisio, and Torchiano, 2005] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Transactions on Software Engineering* **31** (March 2005), pp. 226–237.

[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. 108–111.

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

[Hayes, 2004] F. HAYES, "Chaos Is Back," *Computerworld*, www.computerworld.com/managementtopics/management/project/story/0,10801,97283,00.html, November 8, 2004.

[Highsmith and Cockburn, 2001] J. HIGHSMITH AND A. COCKBURN, "Agile Software Development: The Business of Innovation," *IEEE Computer* **34** (September 2001), pp. 120–122.

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

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

[Jalote, Palit, Kurien, and Peethamber, 2004] P. JALOTE, A. PALIT, P. KURIEN AND V. T. PEETHAMBER, "Timeboxing: A Process Model for Iterative Software Development," *Journal of Systems and Software* **70** (February 2004), pp. 117–27.

[Karlström and Runeson, 2005] D. KARLSTRÖM AND P. RUNESON, "Combining Agile Methods with Stage-Gate Project Management," *IEEE Software* **22** (May/June 2005), pp. 43–49.

[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.

[Mens and Tourwe, 2004] T. MENS AND T. TOURWE, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering* **30** (February 2004), pp. 126–39.

[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.

[Nerur, Mahapatra, and Mangalaraj, 2005] S. NERUR, R. MAHAPATRA, AND G. MANGALARAJ, "Challenges of Migrating to Agile Methodologies," *Communications of the ACM* **48** (May 2005), pp. 72–78.

[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.

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

[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/.

[Reifer, Maurer, and Erdogmus, 2003] D. REIFER, F. MAURER, AND H. ERDOGMUS, "Scaling Agile Methods," *IEEE Software* **20** (July/August 2004), pp. 12–14.

[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, pp. 328–38.

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

Reading, MA, 1998.

[Schwaber, 2001] K. SCHWABER, *Agile Software Development with Scrum*, Prentice Hall, Upper Saddle River, NJ, 2001.

[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.

[Softwaremag.com, 2004] "Standish: Project Success Rates Improved Over 10 Years," www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish, January 15, 2004.

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

[Standish, 2003] STANDISH GROUP INTERNATIONAL, "Introduction," www.standishgroup.com/chaos/introduction.pdf, 2003.

[Stephens and Rosenberg, 2003] M. STEPHENS AND D. ROSENBERG, *Extreme Programming Refactored: The Case against XP*, Apress, Berkeley, CA, 2003.

[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.

[Yu, Schach, Chen, and Offutt, 2004] L. YU, S. R. SCHACH, K. CHEN, AND J. OFFUTT, "Categorization of Common Coupling and Its Application to the Maintainability of the Linux Kernel," *IEEE Transactions on Software Engineering* **30** (October 2004), pp. 694–706.

第3章 软件过程

学习目标

通过本章学习，读者应能：

- 解释二维生命周期模型的重要性。
- 描述统一过程中的5个核心工作流。
- 列举测试工作流所测试的制品。
- 描述统一过程的4个阶段。
- 解释工作流与统一过程阶段之间的差异。
- 认识到软件过程改进的重要性。
- 描述能力成熟度模型（CMM）。

软件过程是软件开发的方式，它结合了方法学（1.11节）和软件生命周期模型(第2章)与技术、所采用的工具（5.4～5.10节），以及其中最重要的因素（即开发软件的个体）。

不同的组织有着不同的软件过程。例如，考虑文档问题。有些组织认为他们所开发的软件是自为文档的，即通过读取源代码就能理解产品。而其他组织则强调文档的重要性，他们谨慎地编制规格说明，并系统地进行验证。然后努力实施设计活动，在开始编码前对设计进行多次验证，并向程序员提供每个代码制品的详尽描述。预先计划好测试用例，记录每次测试运行的结果，仔细归档测试数据。一旦产品已经交付并安装到客户计算机上，任何修改建议都必须以书面形式提交，并给出作此修改的详细理由，仅当文档已经更新，并且文档的修改已经得到许可，方能将修改集成到产品中。

测试强度是对组织进行比较的另一个尺度。一些组织将近半的软件预算用于测试软件，而另一些组织则认为仅有用户才能完全测试产品。因此，一些公司花费极少的时间和努力去测试产品，而将大量的时间用于修正用户所报告的问题。

交付后维护是许多软件组织的重中之重。使用了10年、15年甚至20年的软件仍然会继续进行改进以满足需求。此外，即使软件被成功维护多年之后，遗留问题仍会不断出现。近乎所有组织都会每隔3～5年就将软件移动到更新的硬件平台上，这也属于交付后维护的范畴。

相反，也有其他一些组织主要关心研究，而将开发（不考虑维护）留给他人。大学的计算科学系尤为如此，研究生们开发软件是为了证明某一设计和技术的可行性。已验证特征的商用开发则留给其他组织来完成。(关于不同组织软件开发方法上的广泛差异，请参阅备忘录3.1。)

备忘录3.1

为何不同组织间的软件过程的差异如此之大？主要原因之一在于缺乏软件工程技能。有太多的软件专业人员无法跟上时代的脚步，由于所知无他，他们一直采用古老陈旧的方式来开发软件。

软件过程存在差异的另一个原因是，许多软件经理虽然是出色的管理者，但对软件开发或维护却知之甚少。由于技术知识的欠缺，往往导致项目滞后于计划进度，难以为续。通常，这正是许多软件项目总不能完成的原因。

软件过程间存在差异还有一个原因是管理层的态度。例如，某个组织可能认为哪怕未能进行充分测试，也最好能按期交付产品；相同情况之下，另一组织则可能认为，相较于花时间去完整测试产品而导致交付延迟，未充分测试就发布产品的风险会更大。

然而，如果不考虑确切进程，则软件开发过程可分解为如图2-4所示的5个工作流：需求、分析（规格说明）、设计、实现和测试（参见备忘录3.2）。本章将描述这些工作流，并给出每个工作流中可能出现的潜在挑战。与软件开发相关的挑战性问题，其解决方案通常是至关重要的，本书余下部分将致力于描述合适可用的技术。本章第一部分仅提出有关的挑战，但会给出相应解决方案所在的章节。因此，这部分不单是软件过程的概述，而且也是本书余下大部分的导引。本章结论部分将给出软件过程改进方面的国内外发展。

首先来研究统一过程。

备忘录3.2

为了与对象管理组织（OMG）的软件过程工程元模型（SPEM，一种过程定义的标准[OMG，2005]）相一致，统一过程的术语发生了变化。原来使用的术语“工作流”已改为新的术语“规程”（discipline）。术语“工作流”的含义已稍有变化，如今它表示的是一个特定的活动序列。因此，每个统一过程规程都包含一个（新形式的）工作流。

术语的改动产生了广泛的混淆。因此，多数软件工程师继续像往常一样使用术语“工作流”。在本书中也是这么做的。

另一变化则是将原先的需求工作流分解为两个工作流：业务模型工作流和（新的）需求工作流。这同样会令人感到混乱，因此，本书仍然将业务模型作为需求工作流的一部分。

3.1 统一过程

正如本章开篇所述，方法学是软件过程的一个组成部分。如今的主流面向对象方法学是统一过程。就像备忘录3.3所解释的，统一“过程”实际上是一种方法学，但是“统一方法学”已被用作为“统一建模语言”（Unified Modeling Language，UML）初始版本的名字。统一过程的三个先驱（OMT、Booch方法和Objectory）已被淘汰，而其他面向对象方法学也很少再被提及。因此，统一过程已经成了目前面向对象软件开发的首选。所幸，正如本书第二部分所述，统一过程是在各个方面都表现出色的面向对象方法学。

备忘录3.3

直至大约10年前，普遍使用的面向对象软件开发方法学还是对象模型技术（Object Modeling Technique，OMT）[Rumbaugh et al.，1991]以及Grady Booch提出的方法[Booch，1994]。在纽约斯卡奈塔第市的通用电气研发中心，Jim Rumbaugh及其团队开发了OMT，而Grady Booch则在加利福尼亚州圣克拉拉市的Rational公司提出了Booch方法。所有的面向对象软件开发方法学必定都是等价的，因此，OMT和Booch方法之间的差异也极小。尽管如此，两大阵营的支持者之间还是存在着友好的竞争。

在1994年10月，当Rumbaugh加入Rational公司Booch所在的团队之后，这种情况发生了改变。两种方法学组合在一起，产生了一种新的方法学。当这一工作的初始版本发布之后，有人指出他们并没有开发出一种方法学，而仅仅只是给出了面向对象软件开发的一种表示法。很快，名字“统一方法学”（Unified Methodology）被改为“统一建模语言”。1995年，Ivar Jacobson（对象工厂方法学（Objectory methodology）的发起者）加入了Rational公司。Booch、Jacobson和Rumbaugh（被亲切称为“三剑客”，名字源自1986年John Landis与Chevy Chase、Steve Martin一起主演的电影《神勇三剑客》(Three Amigos!)）开始一起工作。1997年出版了UML的1.0版本，此举在软件工程界引起了轰动。因为直到那时，才有了被统一接受的软件开发表示法。几乎一夜之间，UML就在全世界广为应用。作为对象技术方面

的世界主流企业联盟，对象管理组织（Object Management Group，OMG）负责为 UML 制定国际标准，以便每个软件专业人员都可以使用相同版本的 UML，从而促进组织内部个体之间以及世界上各家公司之间的交流。今天，在面向对象软件产品表示方面，UML［Booch，Rumbaugh，and Jacobson，1999］已经毫无争议地成为国际标准表示法。

管弦乐的乐谱能够标示出在演奏某一乐章时需要哪些乐器，每一乐器要演奏的音符以及何时演奏，还有各种其他的技术细节（如主音调、拍子、响度等）。这类信息可以通过自然语言而不是乐谱来表示吗？或许可以，但不可能依据这样的描述来进行演奏。例如，钢琴家和小提琴家无法演奏如下描述的乐章："音乐为进行曲，主音调为 B 小调。第一小节起始于提琴中 C 之上的 A（四分音符）。当演奏此音符时，钢琴家弹奏含有 7 个音符的和弦。右手弹奏下列 4 个音符：中 C 之上的 E 升符……"

显然，在某些领域，文本描述不能简单地取代图示。音乐就是这样一个领域；软件开发则是另外一个领域。并且，对于软件开发而言，现今可用的最佳建模语言即为 UML。

"三剑客"不单通过 UML 给软件工程界带来了轰动，他们接下来又出版了一套完整的软件开发方法学，统一了他们三人各自的方法学。这个统一方法学最初称为"Rational 统一过程"（Rational Unified Process，RUP）。名字中的 Rational（合理）不是说他们认为所有其他方法都是不合理的，而是因为当时他们三人都是 Rational 公司的高级经理（2003 年，Rational 公司为 IBM 所收购）。在他们关于 RUP 的书［Jacobson，Booch，and Rumbaugh，1999］中，使用了名字"统一软件开发过程"（Unified Software Development Process，USDP）。为简便起见，今天所普遍采用的术语是"统一过程"。

统一过程不是软件产品开发所可以依赖的一系列具体步骤。实际上，由于软件产品类型的多样性，并不存在这样"一刀切"的方法学。软件产品的应用领域各不相同，例如，保险、航空、制造业等。先于竞争者快速向市场发布一个 COTS 包所用的方法学，与开发高安全性电子资金交易网络所采用的方法学是不同的。此外，软件专业人员的技能也各不相同。

相反，应该将统一过程看作是一种适应性方法，即随所开发的特定软件产品而改变。正如第二部分将提到的，统一过程的某些特征不适用于小型甚至是中型软件。然而，在各类大小的软件产品中，统一过程都被大量采用。本书所侧重的是统一过程的通用部分，不过也会讨论统一过程中仅适用于大型软件的某些方面，以确保充分认识在开发大型软件产品时所需要解决的问题。

3.2　迭代与增量

面向对象范型自始至终使用建模方法。模型是一组 UML 图，表示待开发软件产品的一个或多个方面（第七章将介绍 UML 图）。前面介绍过，UML 代表统一建模语言，它是用于表示（模拟）目标软件产品的工具。使用 UML 图形化表示方法的一个主要原因可以通过一句古老的谚语来说明：百闻不如一见（a picture is worth a thousand words）。较之于口头描述，UML 图使软件专业人员之间的交流更为快捷、更为精确。

面向对象范型是一种迭代增量方法学。每个工作流都包含许多步骤，并且为实现工作流，要重复执行各个步骤，直到开发团队成员满意地认为，已经有了一个精确的 UML 模型来模拟他们想要开发的软件产品了。也就是说，即使最有经验的软件专业人员也会反复迭代，以便最终能够满意地得到他们认为正确的 UML 图。其隐含的意思就是，不论软件工程师多么出色，他们也几乎不能第一次就得到正确的工作产品。这到底是怎么回事？

软件产品的本质在于，实际上的每件产品都必须迭代并增量地进行开发。毕竟，软件工程师是人，因此要遵从米勒法则（2.5 节）。也就是说，不可能同时考虑好所有的事情，初始时仅能处理大约 7 个信息块。而后，当考虑下一组信息块时，可以获得更多关于目标软件产品的知识，通过

这些增加的信息，对 UML 图进行修改。过程会依此持续下去，直到最终软件工程师满意地认为所有给定工作流的模型都是正确的。换句话说，最初是根据工作流起始时可获取的知识来绘制最可能的 UML 图。然后，随着更多与正在模拟的现实世界系统有关知识的获取，图会做得更精确（迭代），并且得到扩展（增量）。因此，不管一个软件工程师经验如何丰富和技能如何娴熟，都得重复地进行迭代与增量，直到满意地认为 UML 图精确地表示了将要开发的软件产品为止。

理想情况下，学完本书之后，读者将具备必要的软件工程技能，能够为构建大型复杂的软件产品开发统一过程。遗憾的是，有三个原因阻碍这一理想成为现实：

1）正如仅通过一门课不可能就成为微积分或某门外语方面的专家一样，要精通统一过程需要更为广泛的学习，更为重要的是，需要在面向对象软件工程方面不断地进行实践。

2）统一过程开发的最主要目的是开发大型复杂的软件产品。为了能够处理这类软件产品中诸多错综复杂的细节，统一过程本身就很庞大。由于本书篇幅有限，难以面面俱到。

3）要讲授统一过程，必须提供案例研究以阐明统一过程的特征。而要阐明应用于大型软件产品中的这些特征，这类案例研究的规模必须足够大。例如，仅仅规格说明就要达到 1 000 多页。

出于这三方面考虑，本书给出统一过程的大部分特征，而非全部细节。

现在来讨论统一过程的 5 个*核心工作流*（需求工作流、分析工作流、设计工作流、实现工作流和测试工作流）及其相关的挑战。

3.3 需求工作流

软件开发是一个昂贵的过程。如果软件产品被认为必能使企业获利，或至少从经济角度看是合理的，则客户会向开发组织提议这个软件产品，通常，由此便开启了开发过程。*需求工作流*的目标是为了让开发组织确定客户的需求。

开发团队的首要任务就是从根本上理解*应用领域*（简称*领域*），即目标软件产品即将要运行的特定环境。这类领域可以是银行、汽车制造公司或者核物理部门。

在过程的任意阶段，如果客户不再认为软件的成本效果合算，开发将立即终止。本章始终假设用户觉得成本是合理的。因此，软件开发的一个重要方面是*业务模型*（business model）文档，以证明目标产品的成本效果合算。（实际上，“成本”并非单指资金。例如，军用软件开发通常出于战略或战术方面的原因，此时，软件成本专指不开发某种武器时所可能造成的潜在危险。）

在客户与开发方的首次会议中，客户按其概念上的理解概略描述所期望的产品。从开发方角度来看，客户对预期产品的描述可能非常模糊、不合理、自相矛盾，或者根本就不可能实现。此时，开发人员的任务就是要明确客户的需求，并从客户描述中找到存在的约束条件。

- 主要约束几乎总是*最后期限*（deadline）。例如，客户可能规定完整产品必须在 14 个月内完成。在近乎所有应用领域，目标软件普遍都是任务关键性的。也就是说，客户需要的软件产品和所在组织的核心任务的运行相关，目标产品交付的任何延期都会对该组织不利。
- 通常还存在着一些其他约束，如*可靠性*（例如，产品必须 99% 的时间保持运行，或者失败之间的平均时间至少达到 4 个月）。另一个常见约束是可执行的目标程序的大小的限制（例如，软件必须运行于客户的个人计算机上，或者运行于卫星硬件之中）。
- *成本*几乎总是一个重要的约束。然而，客户很少告诉开发方会在产品开发上投入多少资金。通常的做法是，当完成规格说明后，客户会要求开发方给出完成项目开发的价格。依据投标规程，客户期望的是开发者的标低于项目原有的预算。

有时，把对客户需求的初步调查称为*概念设计*（concept exploration）。随后，开发团队成员与客户团队成员之间再次会面，继续精化并分析所提议产品的功能，以使其技术上可行、经济上合理。

到目前为止，一切看起来都非常直观。但遗憾的是，需求工作流的实施往往并不是充分的。当产品最终交付给用户时，或许客户对规格说明的签署已经过去了一两年，客户可能会对开发

人员说："我清楚这是按我的要求来开发的，但它并非是我真正所需要的。"因此，客户所要求的与开发方认为客户想要的，都不是客户真正需要的。导致这一困境的原因很多。首先，客户并不真正理解其组织内部将要发生的事情。例如，如果系统当前缓慢运转的原因是糟糕的数据库设计，则要求软件开发方提供一个操作更快的系统也无济于事。或者，如果客户运营的是无利可图的连锁经营，则客户可能想要开发一个财务管理信息系统，以反映销售、工资、财务支出、财务收入等。如果亏损的真正原因是货品亏缺（因员工偷盗或顾客扒窃），那么这样一个产品将几无用处。如果真是这样，所需要的应该是一个库存控制系统而非财务管理信息系统。

然而，客户所要求产品经常错误的主要原因是软件的复杂性。如果软件专业人员都难以清楚地呈现一个软件及其功能，那么对于并不精通计算机的客户而言，问题会更糟。正如第 10 章所述，统一过程在这方面会有所助益。统一过程包含许多 UML 图，这必然有助于客户更为深入地理解所需开发的产品。

3.4　分析工作流

分析工作流的目标是分析并精化需求，以理解软件正确开发和易于维护所必需的需求细节。然而，初看起来开发软件产品好像并不需要分析工作流。毕竟，更为简单的处理方式是，持续不断地向前迭代需求工作流，直到已获得目标软件产品的必要理解，依此来开发一个软件产品。

问题的关键是，需求工作流的输出必须为客户所完全理解。换句话说，必须把需求工作流的制品表达成客户方的语言，即自然（人类）语言，如英语、亚美尼亚语或祖鲁族语。但是，无一例外，所有自然语言都或多或少地不够严密，容易引起误解。例如，考虑下面这段话：

从数据库中读取一个零件记录和一个设备记录。如果它包含字母 A，且其后紧随字母 Q，则计算将该零件运输到工厂的成本。

初看起来，似乎需求已经非常明显。但是，第二句中的"它"指代什么：零件记录？设备记录？还是数据库？

如果用数学符号来表达（陈述）需求，就不会产生这种混淆。然而，使用数学符号会妨碍客户地理解需求。因此，很可能在客户与开发者之间造成对需求的误解，最终，为满足这些需求而开发的软件产品不可能是客户所需要的。

解决方法是采用两个独立的工作流。需求工作流按客户的语言来表达；分析工作流则采用更为精确的语言，以确保设计和实现工作流的正确实施。另外，在实施分析工作流期间，添加更多的细节，这些细节可能与客户理解目标软件产品无关，但是对于开发该软件产品的软件专业人员而言是必要的。例如，状态图（11.9 节）的初始状态肯定不为客户所关注，但是如果开发者想要正确开发目标产品，则必须将其包含在规格说明中。

产品规格说明等同于契约。当软件开发方交付一个满足规格说明验收标准的产品时，被视为已经履行完契约。因此，规格说明不能包含不严密的用语，如合适的、方便的、充分的或足够的等，或者听起来明确但实际上含糊的术语，如最优的、98% 完成等。虽然软件开发契约可能引起法律诉讼，但是当客户和开发者属于相同组织时，则不会存在因规格说明而导致的法律行为。然而，即使在内部软件开发的情况下，也总是应该编写规格说明，似乎它们可以成为将来出现麻烦时的证据。

更重要的是，对于测试和维护，规格说明是必要的。除非规格说明足够精确，否则难以判定它们的正确性，姑且不论实现是否满足规格说明。如果不存在一些精确表述当前规格说明的文档，就难以修改规格说明。

当使用统一过程时，不存在通常意义下的规格说明文档。取而代之，展现给客户的是一组 UML 制品，如第 11 章所述。较之于用自然语言来编写的规格说明，这些 UML 制品形成了对目标产品更为精确的描述。

在分析工作流中，已经确定产品的体系结构。也就是说，产品已分解为相对独立的组件（类），每一个都包含自己的数据（属性）和操作（方法）。在分析阶段就已确定这些属性。然而，出于11.25节所说明的原因，方法需要在设计阶段才加入到类中。

一旦客户批准了规格说明，就开始进行详细计划和评估。如果事先不知晓项目所需时间和所耗成本，客户是不会授权软件项目开发的。从开发者的角度来看，这两方面也同样非常重要。如果开发者低估了项目成本，那么客户只会支付所同意的费用，这可能远低于开发者的实际成本。相反，如果开发者高估了软件成本，客户可能会取消项目或转包给其他价格更为合理的开发者。在时间估计方面，情况也类似。如果开发者少估了完成项目所需的持续时间，产品就会延迟交付。最好情况下，也会失去对客户的信誉。最为糟糕地，契约中的延期补偿条款会生效，导致开发者经济上的损失。再者，如果开发者多估了所需时间，用户可能将产品开发交给能承诺更快交付的开发者。

对于开发者而言，不能仅评估持续时间和成本，还需要安排合适的人员到开发过程的各个工作流中。例如，仅当等到软件质量保证（SQA）小组已经通过相关设计制品，实现团队才能启动，而只有分析团队完成任务，才需要设计团队。换句话说，开发者必须预先规划。相应于开发过程的各个工作流，必须拟定软件项目管理计划（SPMP），并明示在每个任务中包含开发组织中的哪些成员，以及每个任务完成的截止时间。

最早可以在规格说明拟定时，制定这种详细计划。在此之前，对完成计划而言，项目还未定形。当然，项目的一些方面必须在一开始就计划好，但是只有当开发人员确切知道所要开发的产品时，才能指定开发计划的所有方面。

因此，一旦客户批准了规格说明，开发者就要开始准备软件项目管理计划。计划的主要部分是*交付能力*（deliverables，客户将得到的产品）、*里程碑*（milestones，客户得到产品的时间）和预算（budget，将花费的成本）。

计划最为详尽地描述软件过程。包含的项目有：将采用的生命周期模型，开发组织的组织结构，项目责任，管理目标和优先级，将使用到的技术和CASE工具，以及详细的进度、预算和资源分配。整个计划的基础是持续时间和成本估算。得到这些估算结果的技术将在9.2节中描述。

第11章描述了分析工作流。分析工作流的主要制品是软件项目管理计划。9.4～9.6节说明了如何拟定SPMP。

下面研究设计工作流。

3.5 设计工作流

产品的规格说明详细说明了产品的功能；设计则给出了如何实现产品的这些功能。更准确地说，*设计工作流*的目标是精化分析工作流的制品，直到程序员能够以某种形式实现。

如3.4节所述，在分析工作流中，目标产品被分解为类，并且提取出每个类的属性。在设计工作流中，设计团队确定产品的内在结构，提取出方法，并分配给相应的类。特别地，必须详细制定每个方法的接口（即传递给方法的参数以及从方法返回的参数）。例如，某种方法可能用于测量核反应堆的水位，并在水位过低时发出警报。航空电子产品中的一种方法可能需要输入两个甚至更多即将来袭的敌方导弹坐标值，计算其轨迹，并调用另一方法提醒飞行员采取可能的避让措施。设计工作流的另一个重要方面是为每种方法选择合适的算法。

现在考虑数据方面，属性的提取是在分析工作流（3.4节）中完成。然而，属性的格式则是在设计工作流中确定的。例如，假设在分析工作流中提取了属性grossProfit（总利润），在设计工作流期间，设计团队确定该属性应该表示为8位整数，因为产品必须能够处理的最大可能总利润为99 999 999美元。类似地，属性StatusOfPressureValve（压力阀状态）的值可以为open（打开）、closed（关闭）、disconnected（分离）或inoperable（不宜操作）。

设计团队必须精确记录所做的设计决定，这类信息是必需的，有如下两方面原因：

1）在设计产品时，有时可能会走进死胡同，设计团队必须回溯，并对某些模块进行重新设计。如果对于特定决定保有书面记录，则在发生这类情况时，能够帮助团队，使其返回到正常轨道。

2）第二是为了有助于未来的改进（交付后维护）。理想的，产品设计应当是可扩展的（open-ended），可以在未来加以改进，即在不影响整体设计的情况下，增加新类或取代现有类。当然，实际上难以达成这一理想。因为最后期限约束往往使得设计人员必须与时间赛跑，以完成满足初始规格说明的设计，而不会去考虑任何后期的改进。如果在产品规格说明中包含（在产品交付给客户后再添加进来的）未来改进，那么必须在设计中加以考虑，但这种情况非常少见。一般而言，规格说明以及随后的设计都只处理现有的需求。此外，当产品还处在设计阶段时，也无法确定所有可能的未来改进。最终，如果设计真的必须考虑所有的未来可能性，则其结构至好可能只是显得难以操纵，至坏则可能由于非常复杂而不可实现。因此，设计人员必须作一些折衷，将能够以多种合理方式进行扩展的方面放在一个设计中，而不致以后需要整体上的重新设计。然而，需要进行较大改进的产品，最终将导致设计再也无法处理进一步的改变。此时，必须从整体上重新设计产品。如果实施重新设计的团队成员能得到一份记录，包含所有最初设计决定的理由，那么团队的任务将相对更为容易。

3.6　实现工作流

*实现工作流*的目标是用所选定的实现语言实现目标软件产品。有时，一个小型软件产品直接就由设计人员来实现。相反，一个大型软件产品则会划分为更小的子系统，然后由多个编码团队并行实现。每个子系统都由*代码制品*（code artifact）组成，分别由各个程序员来实现。

通常，交给程序员的唯一文档就是相关的设计制品，包含所需要实现的类的设计。设计通常会给程序员提供足够信息，以便在实现代码制品时，不会有太多的困难。如果存在任何问题，可以通过咨询有关的设计负责人员，快速搞清有关问题。然而，各个程序员并无从知晓整体体系结构（2.7 节）的正确性。仅当开始集成各个代码制品时，整体设计的缺陷才能显现出来。

假定已经实现并集成好了多个代码制品，并且集成好的部分产品迄今看起来能正确运行。进一步假设某个程序员已经正确实现制品 a45，但是，当该制品与其他已有制品集成时，产品却失败了。失败的原因不在制品 a45 本身，而在于制品 a45 与产品其他部分集成的方式。即便如此，在这类情况下，仅编写制品 a45 的程序员也会因失败而受到指责。这实际上很不公平，因为程序员只是遵从设计人员提供的指导说明，并严格按照制品设计中的描述加以实现。程序设计团队的成员很少能够看到“大蓝图”，即整体体系结构，更不用说对它进行评价了。要求程序员能清楚特定制品从整体上对产品的影响，虽然这非常不公平，但实际上这类不幸时有发生。这就是为什么在各个方面进行正确设计显得如此重要。

测试工作流的部分任务就是验证设计（及其他制品）的正确性。

3.7　测试工作流

如图 2-4 所示，在统一过程中，自始至终，测试与其他工作流并行实施。两个需要测试的主要方面是：

1）每个开发人员和维护人员都有责任确保各自工作的正确性。因此，软件专业人员必须反复测试自己所开发或维护的每个制品。

2）一旦软件专业人员确信某一制品的正确性，就将其提交给软件质量保证小组进行独立测试（参见第 6 章）。

*测试工作流*的性质依测试制品而改变。然而，对于所有制品而言，很重要的一个特性是可追溯性（traceability）。

3.7.1 需求制品

如果在软件产品的整个生命周期中，可以对需求制品进行测试，则它们必须包含的一个属性就是可追溯性。例如，必须能将分析制品中的每一项都回溯到需求制品，并且对于设计制品和实现制品也能进行相似回溯。如果系统地给出了需求，并且包含合理编号、交互引用和索引，则开发人员可以轻易地通过随后的制品进行回溯，并确保它们确实正确地反应了客户需求。在随后检查需求团队成员的工作时，可追溯性也简化了 SQA 小组的任务。

3.7.2 分析制品

如第 1 章所指出的，已提交软件中错误的主要来源是规格说明错误，直到在客户计算机中已安装软件，并且客户组织按预设意图使用软件时，这些规格说明错误才能被检测出来。因此，分析团队和 SQA 小组必须努力校验分析制品。此外，他们还必须确保规格说明的可行性。例如，确保特定硬件组件速度足够快，或者用户当前所使用磁盘的存储能力足以支持新产品的运行等。校验分析制品的一种极佳方式是评审会议，由分析团队和用户方的代表出席。会议通常由 SQA 小组成员主持。评审的目的是判定分析制品的正确性。由评审员从头至尾审查分析制品，以验证是否存在任何错误。走查（walkthrough）和检查（inspection）是评审的两种类型，将在 6.2 节加以介绍。

现在来校验客户签署规格说明之后的详细计划和估算。虽然开发团队以及 SQA 小组有必要依次对 SPMP 的每个方面进行仔细检查，但必须特别关注的是计划持续时间和成本估算。要达成这一目标，管理层必须在开始详细计划前得到两个（或多个）对持续时间和成本的独立估算，然后协调其间的明显差异。对于 SPMP 文档的校验，一个最好的办法就是类似于分析制品的评审那样进行评审。如果持续时间和成本估算令人满意，客户将会同意项目继续进行。

3.7.3 设计制品

如 3.7.1 节所提及的，可测试性的一个重要方面是可追溯性。在设计阶段，这意味着每部分的设计都与分析制品相联系。一个合理的交互引用设计能提供一个有力的工具，使开发人员和 SQA 小组能够校验是否设计与规格说明相一致，是否规格说明的每部分都与设计的某些部分相对应。

设计评审与对规格说明的评审相似。然而，从大多数设计的技术性质角度来看，客户通常被排除在外。在整个设计过程中，对于每个独立的设计制品，设计团队的成员和 SQA 小组一起工作，以确保设计的正确性。需要关注的错误类型包括：逻辑错误、接口错误、缺少异常处理（错误情况的处理），以及与规格说明的不一致（最为重要）。此外，评审小组应该清楚，可能存在某些在前一工作流中没有检测出的分析错误。评审过程的详细介绍参见 6.2 节。

3.7.4 实现制品

每个组件在实现时都应该进行测试（桌面检查）；且在实现后，要运行测试用例进行测试。由程序员来完成这种非正式的测试。而后，质量保证小组系统地测试各个组件，这称为单元测试。第 13 章介绍了各种单元测试技术。

除运行测试用例外，代码评审也是用于检测程序设计错误的一个很有力且很成功的技术。这里，程序员引导评审团队成员检测组件清单。评审团队必须包含一名 SQA 代表。这一过程类似于先前所描述的规格说明评审和设计评审。与所有其他工作流一样，需要保存一份关于 SQA 小组活动情况的记录，并作为测试工作流的一部分。

对某一组件编码完毕后，必须将其与其他已编码的组件组合起来，以便 SQA 小组能够判定（部分）产品整体上的功能正确性。组件集成的方式（一次所有或一次一个）以及特定的顺序（依据组件互联图或类图，自顶向下或自底向上）对最终产品的质量具有重要影响。例如，假设产品

经由自底向上集成，要是存在重要设计错误，它会延迟显现，这就将导致重写，其代价高昂。相反，如果组件自顶向下集成，那么底层组件通常不能像在自底向上集成中那样，得到完整测试。第13章将详细讨论这类问题及其他问题，并详细说明编码与集成必须并行执行的理由。

集成测试的目标是检测组件是否正确组合而得到满足规格说明的产品。在集成测试时，要格外注意测试组件接口。形参的个数、顺序和类型必须与实参匹配，这一点尤为重要。编译器和链接器能很好地执行这种强类型检查［van Wijngaarden et al.，1975］。但是，许多语言都不是强类型的；当使用这类语言时，接口检查必须由SQA小组成员完成。

当完成集成测试后（即当已编码并集成好所有组件时），SQA小组开始产品测试。依据规格说明，检测产品整体上的功能性。特别地，要测试规格说明中所列出的约束条件。一个典型的例子就是，看是否已满足响应时间需求。产品测试的目标是判定规格说明是否已经正确实现，因此，只要规格说明是完整的，就可以拟定许多测试用例。

不仅要测试产品的正确性，而且还要测试其鲁棒性。也就是说，故意输入错误数据，以判定产品是否会崩溃、是否其错误处理能力足以应付有害数据。如果产品要与客户当前已安装的软件一起运行，那么还得运行测试，以确保新产品不会对客户已有的计算机操作产生不利影响。最后，还必须验证源代码和所有其他类型文档的完整性，以及内在一致性。产品测试的相关讨论见13.20节。基于产品测试的结果，开发组织的高级管理者决定是否已准备好将产品交付给客户。

集成测试的最后一个方面是验收测试。此时，软件已交付给客户，他们将使用实际数据而非测试数据在实际硬件上对软件进行测试。不论开发团队或SQA小组多么系统全面，测试用例与实际数据之间仍然存在重要差别，毕竟，测试数据本质上是人工编制的。仅当软件产品通过了验收测试，才被认为满足了规格说明。13.21节将给出验收测试的更多细节。

对于COTS软件（1.10节）而言，产品测试一旦完成，产品的完整版本就将提供给选定的潜在未来客户，进行现场测试。第一个这类版本称为α版（alpha release）。α版经修正后，被称为β版（beta release）；通常，β版将最接近最终版本。（不只COTS软件，在所有软件产品中通常都会应用术语α版与β版。）

COTS软件中的错误经常会影响产品的销量，并导致开发公司的巨大损失。因此，为了尽早发现尽可能多的错误，COTS软件开发人员不断将α或β版提供给选定的公司，期待现场测试能揭示任何潜在的错误。作为回报，一般会许诺给参与α或β版测试的公司提供软件最终交付版本的免费拷贝。对于参与α或β版测试的公司而言，也存在一定的风险。特别地，α版可能充满了错误，这将可能导致系统死机、时间浪费，而且还可能导致数据库的毁坏。然而，通过使用新的COTS软件，公司在此方面将获得竞争优势，领先于其他竞争对手。当软件组织选择由潜在客户进行α测试，而不是由SQA小组进行通盘的产品测试时，有时会出现问题。虽然在大量不同场所进行α测试能够显现大量不同类型的错误，但还是不能替代SQA小组所能提供的系统性测试。

3.8　交付后维护

交付后维护并非产品被交付并安装到客户计算机之后才勉强实施的行为。相反，它是软件过程的完整部分，从一开始就要进行计划。就像3.5节中所解释的那样，只要是可行的设计，就应该考虑未来的改进。进行编码时，也必须考虑未来的维护。毕竟，就如1.3节所指出的那样，交付后维护上的花费要比所有其他软件活动加起来还要多。因此，这是软件产品开发的一个重要方面。交付后维护始终不应被看作是一种事后的追悔之举。相反，整个软件开发工作都应当遵循这样一种方式，以尽量减少对未来的交付后维护产生影响。

交付后维护的一个常见问题在于文档化，或者是缺少文档。在依照最后期限进行软件开发的过程中，经常不会去更新最初的分析和设计制品，因此，对维护团队而言，这些文档几乎毫

无用处。其他文档（如数据库手册或操作手册等）可能从来没有被编写，因为管理人员认为，较之于在软件开发过程中并行开发文档，按时将产品交付给客户更为重要。在许多情况下，对维护人员而言，源代码是唯一可用的文档。软件产业人员的高速流动使得软件维护的境况更加恶化，因为当需要实施维护时，可能最初的开发人员都已经离开了该组织。交付后维护经常是软件产品开发中最具挑战性的方面，除了上述这些原因，还有第14章所给出的其他一些原因。

现在来看看测试，当实施交付后维护时，需要对产品所作的修改进行两方面的测试。其一，校验所需修改是否已被正确实现。其二，在对产品进行所需修改的过程中，确保没有进行其他未注意到的修改。因此，一旦程序员判定预期修改已经实现，必须使用先前的测试用例对产品再次进行测试，以保证产品其他方面的功能性未受影响。这一过程称为*回归测试*。为便于进行回归测试，必须保存先前的所有测试用例，以及那些测试用例的运行结果。第14章将更为详细地讨论交付后维护其间进行的测试。

交付后维护的一个主要方面是记录所作的所有修改，以及每次修改的理由。当修改软件时，必须进行回归测试。因此，回归测试用例是文档的主要组成部分。

3.9 退役

软件生命周期的最后一个阶段是*退役*。在服务多年以后，当进行进一步交付后维护的成本效率不再合算时，软件就进入一个新的阶段。

- 有时，所提议的修改过大，以至必须从整体上来修改设计。在这种情况下，对整个产品进行重新设计并重新编码，会更为合算。
- 对原始设计的改变过多，不经意间就会在产品中引入相互依赖性。此时，即使对于一个小型组件进行细微的改变，也可能会对整个产品产生巨大的影响。
- 文档维护不够充分，回归错误带来的风险已达到一定程度，此时，重新编码比维护会更安全。
- 产品运行的硬件平台（和操作系统）需要更新换代，这时，从零开始重写软件比修改会更为经济。

在以上各种情况下，都采用新版本来取代旧版本，并且软件过程将继续进行。

另一方面，即使产品失去可用性，真正的退役阶段也极少经历。客户组织不再需要产品提供的功能时，就会将产品从计算机中删除。

3.10 统一过程的阶段

不同于图2-4，图3-1中修改了增量的标签，“增量A”、“增量B”等4个增量现在标记为初始阶段（inception phase）、细化阶段（elaboration phase）、构造阶段（construction phase）和移交阶段（transition phase）。换句话说，统一过程的各个阶段相应于各个增量。

虽然在理论上，软件产品的开发经历多次增量，但实际看起来，开发通常仅包括4次增量。这些增量或阶段将在3.10.1～3.10.4节进行介绍，并给出每个阶段的成果，即每个阶段最后应当完成的制品。

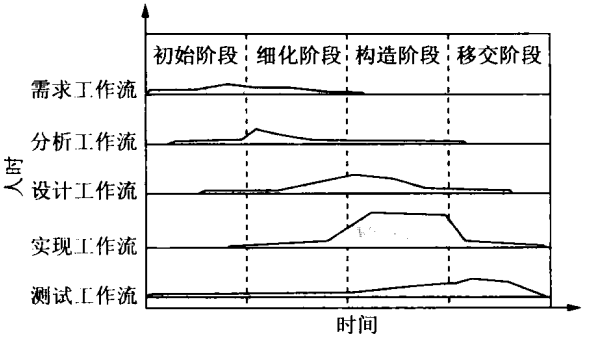


图3-1 统一过程中的核心工作流和阶段

统一过程中执行的每一步骤都分属于5个核心工作流，也分属于4个阶段：初始阶段、细化阶段、构造阶段和移交阶

段。3.3～3.7节已经描述了这4个阶段中的各个步骤。例如，构建业务模型属于需求工作流（3.3节）的一部分，同时也是初始阶段的一部分。不过，如下面所说明的，每一步骤将讨论两次。

考虑需求工作流。为确定客户需求，就像刚才所提到的，其中一步是构建业务模型。换句话说，在需求工作流的框架下，业务模型的构建属于技术范畴。3.10.1节将描述在初始阶段框架下构建业务模型，在此阶段，管理者将决定是否要开发所提议的软件产品。也就是说，业务模型的构建是从经济角度（1.2节）来考虑。

与此同时，以相同的详细度两次介绍同一步骤并没有多大意义。因此，深入地描述初始阶段，以强调各工作流的技术层面与各阶段的经济层面之间的差异，而其他三个阶段则仅简略地进行概述。

3.10.1　初始阶段

*初始阶段*的目标是判定是否值得开发目标软件产品。换句话说，这一阶段的主要目标是判定所提议的软件产品在经济上是否可行。

需求工作流的两个步骤包括理解问题领域和构建业务模型。显然，除非事先已经了解目标软件产品开发所涉及的问题领域，否则，开发人员无从就未来可能的软件产品提供任何意见。不管问题领域是网络电视、机械公司或者肝病专科医院，如果开发人员未充分了解相关领域，则最终所开发的产品将缺乏可信性。因此，第一步应该是获得领域知识。如果开发人员已经完全理解问题领域，则第二步就是构建业务模型。换句话说，第一需要是理解领域本身，第二需要是准确理解客户组织如何在领域内运作。

现在，要来限定目标项目的范围。例如，考虑一个提议用于银行全国分支机构的高安全性ATM网络的软件产品。从整体上看，银行分支机构的业务模型规模可能非常大。为判定目标软件应该包含的内容，开发人员必须仅集中于业务模型的某个子集，即所提议软件产品所包含的子集。因此，限定所提议项目的范围是第三步。

此时，开发人员可以开始制定最初的业务用例。在继续项目之前，需要回答的问题包括[Jacobson, Booch, and Rumbaugh, 1999]：

- 所提议软件产品的成本效率是否合算？也就是说，软件产品所带来的利益是否将超过所投入的成本？开发所提议的软件产品需要多久才能获取投资的收益？特别地，如果客户决定不再开发所提议的软件产品，将损失多少成本？如果要将软件产品投放市场，是否已做过必要的市场研究？
- 所提议软件产品能及时交付吗？也就是说，如果软件延迟投放市场，公司仍将获利？还是会让一个竞争者的软件产品抢走大部分的市场份额？特别地，如果所开发的软件产品是为了支持客户组织的内部运转活动（也许是任务关键型活动），所提议软件产品的延迟交付会有什么影响？
- 软件产品开发中存在哪些风险？如何降低这些风险？将参与开发所提议软件产品的团队成员有必要的经验吗？这个软件产品需要新的硬件支持吗？如果需要，是否会存在不能及时交付的风险？如果会，有办法降低这些风险吗？能否从另一家供应商订购备用硬件？需要软件工具（第5章）吗？当前有这些工具吗？它们是否包含所有必要的功能？在项目进行过程中，将包含所提议客户软件产品所有功能的COTS包（1.11节）投放市场，这可能吗？这该如何决定？

在初始阶段结束时，开发人员必须回答这些问题，以便制定最初的业务用例。

下一个步骤是风险识别。包含3类主要的风险：

1）*技术风险*。技术风险的例子刚才已经提到。

2）*缺少正确需求*。通过正确实施需求工作流，就能降低这类风险，参见第10章。

3）缺少正确体系结构。体系结构可能不够鲁棒。（由2.7节知道，软件产品的体系结构包含各种组件及其组装方式，以及可扩展、可修改而不影响鲁棒性的属性。）换句话说，当开发软件产品时，要往迄今已开发的产品中增加新模块，可能会存在从零开始重新设计整个体系结构的风险。这类似于，用纸牌搭建一个房子，当往其中增加一张牌时，却发现整个房子倒塌了。

风险必须分级，以便先降低最为严重的风险。

如图3-1所示，在初始阶段仅执行少数几个分析工作流。通常所做的是为体系结构设计提取必要的信息。图3-1反映了这一设计工作。

现在来考虑实现工作流，在初始阶段通常不执行编码。然而，有时需要开发概念验证原型，以测试所提议软件产品某些部分的可行性，如2.9.7节所述。

从初始阶段起始时，就开始实施测试工作流。主要目的是确保已准确确定了需求。

计划是每个阶段的基本组成部分。在初始阶段，阶段起始时，开发人员缺少充分信息而不能计划整体开发，因此，在项目开始时，唯一能做的计划就是规划初始阶段本身。同样由于缺少信息，在初始阶段结束时，唯一有意义的计划就是规划下一阶段，即细化阶段。

文档也是每个阶段的基本组成部分。初始阶段的成果包括［Jacobson，Booch，and Rumbaugh，1999］：

- 领域模型的初始版本。
- 业务模型的初始版本。
- 需求制品的初始版本。
- 分析制品的预备版本。
- 体系结构的预备版本。
- 风险的初始清单。
- 初始用例（见第10章）。
- 细化阶段的计划。
- 业务用例的初始版本。

得到最后一项（即业务用例的初始版本）是初始阶段的总体目标。这一初始版本包括软件产品的范围描述与财务细节。如果所提议软件产品将投放市场，则业务用例还包括收入预计、市场评估以及初步的成本估算。如果软件产品仅是内部使用，则业务用例包含初步的成本效益分析（5.2节）。

3.10.2 细化阶段

细化阶段的目标是精化最初的需求、精化体系结构、监控风险并精化风险的属性、精化业务用例，以及制定软件项目管理计划。命名为细化阶段的原因很明显，这一阶段的主要任务是对前一阶段的求精或细化。

图3-1表明，这些任务密切相关于完成需求工作流（第10章）、实际执行整个分析工作流（第11章）以及随后开始的体系结构设计（8.5.4节）。

细化阶段的成果包括［Jacobson，Booch，and Rumbaugh，1999］：

- 完全的领域模型。
- 完全的业务模型。
- 完全的需求制品。
- 完全的分析制品。
- 更新后的体系结构版本。
- 更新后的风险清单。
- 软件项目管理计划（对项目余下部分的计划）。
- 完全的业务用例。

3.10.3　构造阶段

构造阶段的目标是开发软件产品的第一个可操作版本，即所谓的β版（3.7.4节）。再次考虑图3-1，虽然该图只是各个阶段的象征性表示，但是很显然，构造阶段的重点是软件产品的实现和测试。也就是说，编码各个组件并进行单元测试。然后，对代码制品进行编译和链接（集成），以形成子系统，进行集成测试。最后，将子系统组合成完整的系统，进行产品测试。详细描述参见3.7.4节。

构造阶段的成果包括［Jacobson，Booch，and Rumbaugh，1999］：

- 适当的初始用户手册及其他手册。
- 所有制品（β版）。
- 完全的体系结构。
- 更新后的风险清单。
- 软件项目管理计划（对项目余下部分的计划）。
- 更新后的业务用例，如果有必要的话。

3.10.4　移交阶段

移交阶段的目标是确保客户需求已真正得到满足。这一阶段是由已安装β版进行测试的现场反馈所驱动的。（如果软件产品是为特定客户所定制，则仅存在一个测试现场。）纠正软件产品中的错误，并完成所有手册。在此阶段，很重要的一点是要尽力找出先前未曾识别的风险。（甚至在移交阶段也得找出风险，这一重要性在备忘录3.4中加以阐明。）

移交阶段的成果包括［Jacobson，Booch，and Rumbaugh，1999］：

- 所有制品（最终版本）。
- 完全手册。

备忘录3.4

实时系统通常比多数人（即使是开发人员）想像得更为复杂。因此，有时组件之间存在的细微交互，甚至连大多数有经验的测试人员也无法察觉。所以一个看起来细微的改变可能会产生严重的后果。

这方面著名的例子是1981年4月因差错而延迟的航天飞机的第一次轨道飞行［Garman，1981］。航天飞机的电子系统由4台完全同步的计算机控制。同时还有独立的第五台计算机，以便在4台计算机失败时进行后备支持。此前两年，修改了计算机同步之前所执行的初始化模块。不幸的是，这一改变带来了负面影响，一个记录上的时间略微晚于当前时间，而该记录被错误地写入航天计算机同步所使用的数据域。其时间与实际时间非常接近，以致未能检测到这一错误。大约1年后，累计增加的时间差导致系统失败的概率达到了1/67。而后，在发射航天飞机的那天，全世界数亿观众在电视机前翘首以盼之时，出现了同步失败，4台航天计算机中的3台比第一台计算机的时间晚了一个周期。

当4台计算机的时间不一致时，故障自动防护装置（用于防止第五台独立的计算机从其余4台计算机接收不一致的信息）产生了预期的结果，它阻止了第五台计算机的初始化，由此发射被迫延迟。这一事件中的主要错误出现在初始化模块，而这个模块看起来无论如何也不会与同步程序有任何关联。

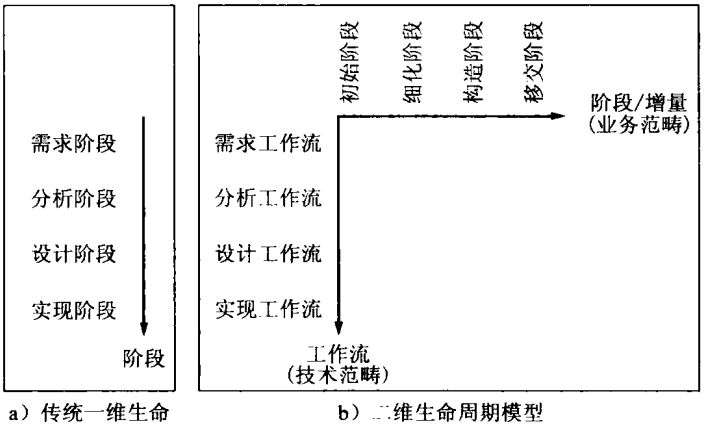
遗憾的是，这不是影响太空发射的最后一次软件错误。例如，1999年4月，一枚Milstar军事通信卫星被发射到一条无用的低轨，造成12亿美元的损失；其原因在于控制大力神4号火箭上升阶段的软件中的一个错误［《Florida Today》，1999］。

不仅太空发射受到实时错误的影响，而且航天器着陆也同样如此。2003 年 5 月，从国际空间站返回的一艘 Soyuz TMA-1 太空船经弹道轨迹降落后，偏离了哈萨克斯坦境内预定地点 300 英里。造成此着陆问题的原因，仍旧是实时软件错误［CNN. com，2003］。

3.11 一维与二维生命周期模型对比

传统的生命周期模型（如 2.9.2 节的瀑布模型）是一维模型，如图 3-2a 中单坐标轴所表示。而统一过程蕴涵的则是一个二维生命周期模型，如图 3-2b 中的两条坐标轴所示。

瀑布模型的一维特征在图 2-3中已有明确反映。相反，图 2-2 则给出了 Winburg 小型案例研究的进化树模型，这是一个二维模型，因此与图 3-2b 中的相类似。



a）传统一维生命周期模型　　b）二维生命周期模型

图 3-2 传统一维生命周期模型和二维生命周期模型的比较

使用二维模型将带来额外的复杂性，这有必要吗？第 2 章已给出了答案，但这个问题很重要，有必要在这里再重申一下。在实际开发软件产品时，在实施分析工作流之前，就将完成需求工作流。类似地，分析工作流应当在设计工作流之前完成，依此类推。然而，在实际中，即使最普通的软件产品也会很大而无法当作单个单元来处理。取而代之，任务必须划分成各个增量（阶段），并且在每个增量中开发人员都必须反复迭代，直到完成构造任务。作为人类，我们往往受限于米勒法则［Miller，1956］，同一时刻只能思考最多 7 个问题。因此，难以从整体上来处理软件产品，而必须将系统分解为子系统。有时甚至连子系统也显得过大，有时可能它包含了所有我们所能处理的组件，只有当我们从总体上更为完全地理解软件产品之后，才能处理这些组件。

在将一个大的问题分解为规模更小、独立程度更大的子问题时，统一过程是目前为止最佳的解决方案。它为增量和迭代提供框架，这些机制可用于处理大型软件的复杂性。

统一过程能较好处理的另一个挑战是不可避免的修改。这项挑战的一个方面是，在软件开发过程中修改客户需求，即所谓的移动目标问题（2.4 节）。

出于所有这些考虑，统一过程是当前可用的最优方法学。然而，毫无疑问，统一过程将会被未来一些新的方法学所超越。如今的软件专业人员正在统一过程之外寻找下一个重大的突破。毕竟，在现实中，每一个领域的人都在不断努力，当今的许多发现都要比过去所提出的优越得多。因此，统一过程也必将由未来的方法学所取代。重要之处在于，基于现有的认识，统一过程比当前其他可用的方法都要好。

对于国内外在过程改进方面所作的努力，将在本章剩余部分加以阐述。

3.12　改进软件过程

全球经济的发展很大程度上依赖于计算机，从而也就依赖于软件。因此，许多国家政府都很关心软件过程。例如，1987 年，美国国防部（DoD）的一个特别工作小组报告："过去 20 年，在应用新的软件方法学和技术来提高软件生产力和软件质量的承诺落空之后，工业和政府组织开始意识到，根本问题在于软件过程管理方面的能力不足"［Brooks et al.，1987］。

为了应对这类及与此相关的问题，基于竞争性采购过程原则，DoD 在美国宾夕法尼亚州匹兹堡市的卡内基·梅隆大学设立了软件工程协会（Software Engineering Institute，SEI）。SEI 的主要成功创举在于创立了能力成熟度模型（capability maturity model，CMM）。软件过程改进方面的相关成绩还包括国际标准化组织的 ISO 9000 系列标准，以及由 40 多个国家组成的国际软件改进组织标准 ISO/IEC 15504。下面将介绍 CMM。

3.13　能力成熟度模型

SEI 的*能力成熟度模型*是不考虑实际使用的生命周期模型而改进软件过程的一组相关策略（术语*成熟*(maturity)是过程良好程度的一个度量）。SEI 的 CMM 开发领域包括软件（SW-CMM）、人力资源管理（P-CMM；P 代表"people"）、系统工程（SE-CMM）。在这些模型之间，存在一些不一致性，以及不可避免的冗余。因此，在 1997 年，决定为成熟度模型开发单一的集成框架，即能力成熟度模型集成（capability maturity model integration，CMMI），以将现存的五个成熟度模型都组合起来。未来，也可以在 CMMI 中增加其他的约束条件［SEI，2002］。

由于篇幅限制，这里仅介绍一种能力成熟度模型 SW-CMM，4.8 节将对 P-CMM 进行介绍。SW-CMM 最早由 Watts Humphrey［1989］在 1986 年提出。前面曾介绍过，软件过程包括开发软件所包含的活动、所使用的技术和工具。因此，它包含软件生产的技术和管理方面。SW-CMM 基于这样的观点：新软件技术的使用本身并不能引起生产力和利润的提高，因为问题的本质在于如何管理软件过程。SW-CMM 的策略是改进软件过程的管理，而认为技术上的改良是理所应当的。软件过程整体改进的结果应当是软件质量更优，并且遭遇时间和成本超支的软件项目更少。

记住，软件过程的改进并非一时之功，SW-CMM 促进的是逐步改变。具体来说，它定义了五级成熟度，组织机构通过一系列小进步慢慢提高，以达到更高等级的过程成熟度［Paulk，Weber，Curtis，and Chrissis，1995］。为理解这一方法，现在对五个等级加以描述。

成熟度等级 1：初始级

初始级（initial level）是最低等级，处于这一等级的组织必然不具备良好的软件工程管理实践。而只能在一个特定基础上完成每一项工作。要是雇用到一位称职的经理，以及一个优秀的软件开发团队，则可能会成功开发某一特定项目。然而，由于缺乏良好的管理，特别是缺乏合理的计划，常常会导致时间和成本超预算。因此，大多数活动都是在冒险，而非事先就已计划好的。在等级为 1 的组织中，软件过程是不可预知的，因为这些完全依赖于当前的雇员状况；如果雇员发生变化，则过程也相应改变。最终，不可能精确预测一些重要方面，如产品开发所要花费的时间，或者产品的成本。

遗憾的是，世界上大多数软件组织都还只是成熟度等级为 1 的机构。

成熟度等级 2：可重复级

在可重复级（repeatable level），采用了基本的软件项目管理措施。计划和管理方面的技术来源于相似产品的开发经验，因此称为*可重复的*（repeatable）。在等级 2，采取了度量方法，这是软件过程得以充分实现的第一步。典型的度量包括对成本和进度的仔细追踪。在等级 1 中，当危机已经出现时才采取措施进行处理；在等级 2 中，管理人员则在问题刚一出现时就立即采取行动，避免其转化为风险。关键在于，通过度量方法，可以在问题出现之前就进行检测。而且，在一

个项目实施中采取的度量方法，可用于为将来项目拟定合理的持续时间进度和成本计划。

成熟度等级3：已定义级

已定义级（defined level），软件生产过程完全文档化。明确地定义过程的管理与技术方面，并不断作出努力以尽可能提高软件过程。通过评审（6.2节）来达到软件的质量目标。在这个等级中，引入新的技术（如CASE环境（5.5节））变得很有意义，可以进一步提高质量与生产力。而在风险驱动的等级1中，“高技术”会使过程变得更加混乱。

虽然许多组织已经达到了等级2或3，但很少有达到等级4或5的。因此，后两个最高等级是软件组织未来发展的目标。

成熟度等级4：已管理级

已管理级（managed-level）组织为每个项目设置质量与生产力目标。不断度量这两个属性，当与目标之间存在不可接受的偏差时，就采取措施进行修正。采取统计质量控制方法［Deming，1986；Juran，1988］，使管理者能够将质量或生产力标准中的随机偏差与有意违背区别开来。（统计质量控制度量法的一个简单例子是，每1 000行代码中所检测出的错误数量。相应目标是，随时间推移不断减少错误数量。）

成熟度等级5：优化级

优化级（optimizing-level）组织的目标是不断进行过程改进。采用统计定量与过程控制技术来对组织进行指导。把从每个已完成的项目中获取的知识运用到未来项目中。因此，过程包含一个正反馈循环，使得生产力与质量稳步提高。

图3-3总结了这五个成熟度等级，同时也给出了与每个成熟度等级相应的关键过程域。为改进软件过程，一个组织必须首先尝试理解当前过程，然后明确表达出想要实现的过程。接着，确定要实现这一过程改进应包含的活动，并按优先级排序。最后，拟定为实现这一改进而包含的计划，并加以执行。重复这一系列步骤，组织将不断改进其软件过程；这一逐级改进过程如图3-3所示。能力成熟度模型的实施经验表明，完全提高一个成熟度等级通常要经过18个月到3年的时间，而要从等级1提升到等级2，有时要耗费3年甚或5年的时间。这表明，对于一个迄今为止仍只具备纯粹特定行为能力和被动反应的组织而言，要向其灌输系统化方法有多么困难。

对于每个成熟度等级，SEI都阐明了一系列的关键过程域（key process areas，KPAs），以作为组织在努力迈向下一等级时应当瞄准的目标。例如，如图3-3所示，等级2（可重复级）的KPAs包含配置管理（5.8节）、软件质量保证（6.1.1节）、项目规划（第9章）、项目跟踪（9.2.5节），以及需求管理（第10章）。这些区域覆盖了软件管理的基本要素：确定客户需求（需求管理）、拟定计划（项目规划）、监控与计划之间的偏差（项目跟踪）、控制组成软件产品的各个模块（配置管理），以及确保产品没有错误（质量保证）。每个KPA都包含2～4个相关目标，如果能成就这些目标，则将达到对应的成熟度等级。例如，一个项目计划的目标是要开发一个计划，该

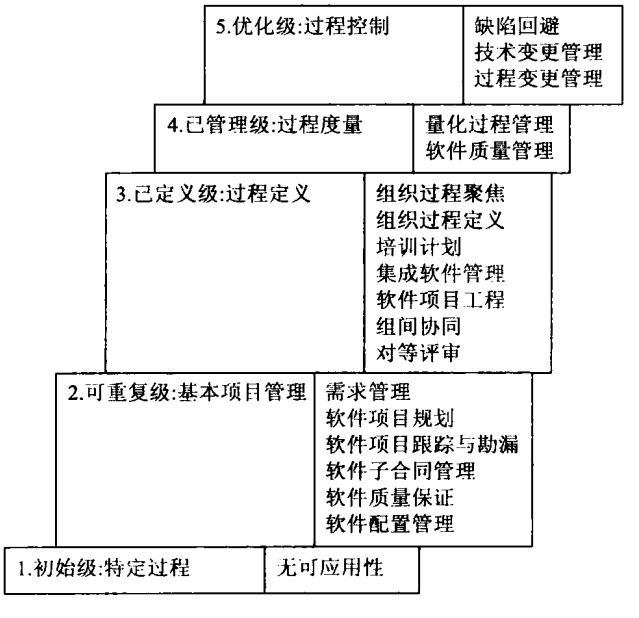


图3-3 能力成熟度模型的5个等级及其关键过程域（KPAs）

计划要能恰当并切合实际地包含所有软件开发活动。

在最高的成熟度等级 5 中，KPAs 包含错误规避、技术变更管理以及过程变更管理。对不同等级的 KPAs 对比发现，等级 5 的组织比等级 2 的组织明显要先进许多。例如，一个等级 2 的组织会关注软件的质量保证，即检测并纠正错误（第 6 章将更为详细地讨论软件质量）。相反，对于一个等级 5 的组织，其软件过程包含错误规避，即尽力确保一开始就不会有软件错误。为了有助于组织达到更高的成熟度等级，SEI 开发了一系列调查问卷，这些问卷成为 SEI 团队进行评估的基础。评估的目的是揭示组织的软件过程中存在的不足，并为组织指明过程改进的方法。

软件工程协会的 CMM 计划由美国国防部倡导。CMM 计划的最初目标之一是要提高国防软件的质量，方法是对那些为 DoD 生产软件的承包商进行评估，并与那些被证明拥有成熟过程的承包商签订合同。1998 年，美国空军规定，任何想要成为空军软件承包商的开发组织都必须达到 SW-CMM 等级 3，DoD 随后也提出相类似的要求。结果，这促使组织提高它们的软件过程成熟度。但是，SW-CMM 计划已经远远超越了 DoD 软件改进的有限目标，它正被各类软件组织所实现，以提高软件质量和生产力。

3.14　软件过程改进的其他方面

软件质量改进方面的一个不同尝试是基于国际标准化组织（International Organization for Standardization，ISO）的 9000-系列标准的，它包含应用于各种工业活动的 5 个相关标准，包括设计、开发、生产、安装和服务；显然，ISO 9000 不仅是一个软件标准。在 ISO 9000 系列中，质量体系标准 ISO 9001［1987］是最适用于软件开发的标准。由于 ISO 9001 的广泛性，ISO 为其出版了专门的指南：ISO9000-3［1991］，以帮助将 ISO9001 应用于软件。（有关 ISO 的更多信息，请参看备忘录 1.4。）

ISO 9000 存在许多不同于 CMM 的特征［Dawood，1994］。ISO 9000 强调通过文字和图片对过程进行文档化，以确保一致性和可理解性。此外，ISO 9000 的原则是与标准相符，不保证产品高质量，而是侧重于降低生产出低质量产品的风险。ISO 9000 仅是质量体系的一部分。质量体系中还对下述方面作出要求：质量委托管理、员工集中培训，以及设置和完成持续质量改进的目标。ISO 9000 系列标准已被 60 多个国家所采用，包括美国、日本、加拿大和一些欧盟（EU）国家。例如，这也就意味着，如果一家美国软件组织希望与欧盟国家的客户开展业务，则该美国组织必须先通过 ISO 9000 认证。由一位授权注册员（审核师）检查公司的软件过程，并确保其符合 ISO 标准。

紧随欧洲同行之后，越来越多的美国组织申请 ISO 9000 认证。例如，1993 年 6 月，通用电气公司的塑料部门要求其 340 个供应商达到这一标准［Dawood，1994］。对于美国政府而言，不太可能遵从欧盟的步调，要求想要与美国组织开展业务的非美国公司去遵从 ISO 9000 标准。然而，来自美国国内及其主要贸易伙伴的压力，最终可能使得全世界范围都遵从 ISO 9000 标准。

与 ISO 9000 一样，**ISO/IEC 15504** 也是一个国际性的过程改进标准。这一标准的前身为 **SPICE**，它是“Software Process Improvement Capability dEtermination”（软件过程改进能力测定）的首字母缩写词。40 多个国家实际参与了 SPICE 标准的制定。SPICE 首先由英国国防部（MOD）发起，其长远目标是将 SPICE 建立为国际标准（英国 MOD 与提出了 CMM 的美国 DoD 是对等的组织）。1995 年，SPICE 第一个版本完成。1997 年 7 月，SPICE 被国际标准化组织和国际电子技术协会（International Electrotechnical Commission）的一个联合委员会所接管。因此，名字 SPICE 被改为 ISO/IEC15504，或简称 15504。

3.15　软件过程改进的成本与收益

软件过程的改进带来了收益的提高吗？结果表明，事实正是如此。例如，在加利福尼亚州 Fullerton 市的 Hughes Aircraft 公司，软件工程部门从 1987 到 1990 年间花费了大约 50 万美元用于项目的评估和改进［Humphrey，Snider，and Willis，1991］。3 年期间，Hughes Aircraft 公司从

成熟度等级 2 提升到等级 3，而且将来还期望提升到等级 4 甚至 5。过程改进带来的结果是，Hughes Aircraft 估计每年能节省 200 万美元。节省的费用来源于多个方面，包括减少超时时间、降低风险、提高员工士气以及减少软件专业人员的流动。

其他组织也给出了对比结果。例如，Raytheon 公司的装备部门从 1988 年的等级 1 提升到 1993 年的等级 3。其结果是生产能力提高 2 倍，过程改进中每一美元的投资得到了 7.70 美元的回报。类似于这样的结果，使得能力成熟度模型在美国软件业界及全世界得到了相当广泛的应用。

例如，印度的 Tata 咨询服务公司使用 ISO 9000 框架和 CMM 来改进过程［Keeni，2000］。在 1996 年到 2000 年间，工作中估算的差错从约 50% 降低到仅 15%。评审的效用（即评审期间发现错误的百分率）从 40% 提高到 80%。而项目返工比率也从近 12% 降低到 6% 以下。

摩托罗拉的政府电子部门（GED）从 1992 年开始，就加入了 SEI 的软件过程改进计划。图 3-4 描述了 34 个 GED 项目，根据每个项目小组的成熟度等级进行分类。从图中可以看出，随着成熟度等级的提高，相应的项目持续时间（与 1992 年之前完成的基线项目相比，项目的持续时间）减少。质量借助于每一百万行等价汇编源程序（MEASL）中的错误数来度量的，这样就比较用不同语言完成的项目，其中，源代码的行数被转换为等价汇编代码的行数［Jones，1996］。如图 3-4 中所示，随着成熟度等级的提高，质量也相应提高。最终，生产力水平通过每人时的 MEASL 值来度量。为保密起见，摩托罗拉没有发布实际的生产力水平图表，因此，图 3-4 给出的是与一个等级为 2 的项目相应的生产力数据。（对于等级 1 的项目，没有可用的质量或生产力数据图表，因为当团队处在等级 1 时，还无法测量这些定量指标。）

CMM 等级	项目数	持续时间相对减量	开发期间检测到的 MEASL 错误数	相对生产力水平
1 级	3	1.0	—	—
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

图 3-4 34 个摩托罗拉 GED 项目的统计结果（MEASL 代表“一百万行等价汇编源程序”）

注：资料来源于［Diaz and Sligo，1997］（©1997，IEEE）。

依据本节已经描述的各类研究，以及本章“进一步阅读材料”中列出的研究成果，可以知道，世界上越来越多的组织正逐步认识到软件过程改进是成本效率合算的。

过程改进行动中一个有趣的影响是，在软件过程改进的研究与软件工程标准之间形成一种交互作用。例如，1995 年，国际标准化组织发布了 ISO/IEC12207，这是一个完整的生命周期软件标准［ISO/IEC 12207，1995］。三年后，电子与电气工程师协会（IEEE）和电气工业联盟（EIA）一起发布了该标准的美国版。这个版本包容了美国软件业的“最佳实践”，其中许多地方可追溯到 CMM。为达到 IEEE/EIA12207 标准，一个组织必须处于或接近能力成熟度等级 3［Ferguson and Sheard，1998］。ISO 9000-3 现在包含 ISO/IEC12207 的部分内容。软件工程标准组织与软件过程改进研究之间的这种相互影响，必将导致更好的软件过程。

软件过程改进的另一个方面可参阅备忘录 3.5。

备忘录 3.5

硬件的运行速度存在着限制，因为电磁场的传播速度不可能比光速快。在一篇题为（No Silver Bullet）《没有灵丹妙药》的著名文章中，Brooks［1986］揭示了软件生产所存在的本质问题，并指出由于软件间存在着相似的约束，这些问题可能永远得不到解决。Brooks 认为，软件的内在属性，如复杂性、无形性、不可见性，以及在整个生命周期期间经历的各类变更，软件过程改进是不可能存在一个大数量级的提升（或“灵丹妙药”）的。

本章回顾

在给出一些基本定义之后，3.1 节开始介绍统一过程。3.2 节描述了迭代与增量的重要性。接着详细解释了统一过程的核心工作流，包括需求工作流（3.3 节）、分析工作流（3.4 节），设计工作流（3.5 节）、实现工作流（3.6 节）和测试工作流（3.7 节）。在测试工作流期间进行测试的各种制品在 3.7.1 ~ 3.7.4 节中分别作了描述。3.8 节讨论了交付后维护，3.9 节则描述了软件退役。统一过程中各工作流与阶段之间的关系在 3.10 节中进行了分析，其中详细描述了统一过程的 4 个阶段：初始阶段（3.10.1 节）、细化阶段（3.10.2 节）、构造阶段（3.10.3 节）以及移交阶段（3.10.4 节）。3.11 节讨论了二维生命周期模型的重要性。

本章最后部分主要讲述了软件过程的改进（3.12 节）。随后描述了国内外软件过程改进研究的各个方面，包括能力成熟度模型（3.13 节）、ISO 9000 和 ISO/IEC 15504 标准（3.14 节）。最后在 3.15 节中讨论了软件过程改进的效用。

延伸阅读材料

第 1 章“进一步阅读材料”中的评论文章[Brook, 1975; Boehm, 1976; Wasserman, 1996; and Ebert, Matsubara, Pezze, and Bertelsen, 1997]阐述了与软件过程相关的一些问题。2003 年 3/4 月的《IEEE Software》期刊发表了几篇关于软件过程的论文，包括讨论统计过程控制的[Eickelmann and Anant, 2003]。[Weller, 2000]和[Florac, Carleton, and Barnard]描述了统计过程控制的实际应用。

关于每个工作流中的测试问题，较为常用的资源是[Beizer, 1990]。本书第 6 章以及该章的“进一步阅读材料”部分将给出更多具体的参考资料。

[Humphrey, 1998]详细描述了最初的 SEI 能力成熟度模型。能力成熟度模型的集成方面可参阅[SEI, 2002]。[Humphrey, 1996]描述了一个个体软件过程（PSP），PSP 的应用结果则出现在[Ferguson et al., 1997]中。[Johnson and Disney, 1998]也讨论了 PSP 的一些潜在问题。[Humphrey, 1999]则同时描述了 PSP 和群体软件过程（TSP）。[Prechelt and Unger, 2000]对 PSP 培训效能进行了评测实验，并给出了相应结果。要使统一过程遵从 CMM 等级 2 或 3，需要进行一些必要的扩展，这在[Manzoni and Price, 2003]进行了讨论。[Guerrero and Eterovic, 2004]描述了在一个小型组织中如何实现 SW-CMM。2000 年 7/8 月的《IEEE Software》杂志包含 3 篇关于软件过程成熟度的论文，而 2000 年 11/12 月的《IEEE Software》期刊上则发表了 4 篇关于 PSP 的论文。

[Herbsleb et al., 1997]对 SEI 软件过程评估给出了综述。这方面也存在一些关于工业界经验的文章，它们从已引入 SEI 过程改进计划的特定公司角度加以描述；典型的例子包括 Shlumberger [Wohlwend and Rosenbaum, 1993]和 Raytheon[Haley, 1996]。SEI 对软件工业的影响在[Saiedian and Kuzara, 1995]和[Johnson and Brodman, 2000]进行了讨论。对于 CMM，[Bamberger, 1997]提出了一种有趣的观点。[Bamford and Deibler, 1993b]对 ISO 9000 和 CMM 进行了详细比较；而[Bamford and Deibler, 1993a]则给出了有关方面的综述。Paulk[1995]详细给出了另一种比较。Pitterman[2000]描述了在 Telecordia Technologies 公司的一个小组是如何达到等级 5 的；对于 Computer Sciences Corporation 公司的小组如何达到等级 5，有关的研究则出现在[McGarry and Decker, 2002]中。对于等级 5 的组织的本质，[Eickelmann, 2003]进行了深入讨论。[van Solingen, 2004]对软件过程改进的成本效率进行了分析。[Dybå, 2005]则从经验角度研究软件过程改进成功的关键因素。

软件产品改进的有关问题可以在[Conradi and Fuggetta, 2002]中找到。[Borjesson and Mathiassen, 2004]列举并分析了爱立信公司所倡导的 18 个不同软件过程改进研究成果。关于 CMM，更为丰富的信息可从 CMM 网站 www. sei. cmu. edu 得到。ISO/IEC 15504（SPICE）主页是 www. sei. cmu. edu/technology/process/spice/。

[Ferguson and Sheard, 1998]对 CMM 和 IEEE/EIA 12207 进行了比较，而[Murugappan and Ke-

eni,2003]则比较了 CMM 和六西格玛(Six Sigma,过程改进的另一方法)。[Blanco,Gutiérrez,and Satriani,2001]描述了一个库,包含大约 400 个软件改进实验的结果。

习题

3.1 考虑需求工作流与分析工作流。将它们合二为一会不会比分别处理更有意义?

3.2 实现工作流比其他任何工作流所执行的测试都多。将这个工作流分解为两个独立的工作流,一个不涉及测试,另一个处理所有测试,会不会更好?

3.3 维护是软件生产中最为重要,也是最难以实现的活动。然而,许多软件工程师都很轻视维护,并且维护人员的报酬通常低于开发人员。你认为这合理吗?如果不合理,你想如何进行改进?

3.4 如 3.9 节所述,为什么会认为真正的退役很少发生?

3.5 由于 Elmer Software 公司的一场火灾,在交付产品给客户前,该产品的所有文档恰好遭到毁坏。缺少文档会产生什么影响?

3.6 假设你刚刚购买了濒临破产的 Antedeluvian 软件开发公司,该公司的成熟度等级还处于 1。要使公司赢利,第一步应采取什么措施?

3.7 3.13 节指出,将 CASE 环境引入成熟度等级为 1 或 2 的组织,几乎没有什么意义。请解释为何如此。

3.8 在成熟度等级低的组织引入 CASE 工具(相对于环境),有何效果?

3.9 成熟度基本等级 1,指的是缺乏良好的软件工程管理实践。对 SEI 而言,把成熟度基本等级标记为 0,岂不是更好?

3.10 (学期项目)如果附录 A 中的 Osric 办公用品和装饰产品是由 CMM 等级为 1 的组织开发的,相对于由 CMM 等级为 5 的组织开发,你期望能找到什么不同?

3.11 (软件工程读物)由教师分发[Eickelmann,2003]论文的复印件。你会选择到等级 5 的组织中工作吗?请解释原因。

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.

[Borjesson and Mathiassen, 2004] A. Borjesson and L. Mathiassen, "Successful Process Implementation," *IEEE Software* **21** (July/August 2004), pp. 36–44.

[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.

[Dybå, 2005] T. DYBÅ, "An Empirical Investigation of the Key Factors for Success in Software Process Improvement," *IEEE Transactions in Software Engineering* **31** (May 2005), pp. 410–24.

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

[Eickelmann, 2003] N. EICKELMANN, "An Insider's View of CMM Level 5," *IEEE Software* **20** (July/August 2003), pp. 79–81.

[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," *Florida Today*, 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.

[Guerrero and Eterovic, 2004] F. GUERRERO AND Y. ETEROVIC, "Adopting the SW-CMM in a Small IT Organization," *IEEE Software* **21** (July/August 2004), pp. 29–35.

[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, 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, 1991.

[ISO 9001, 1987] "ISO 9001, Quality Systems—Model for Quality Assurance in Design/Develop-

ment, Production, Installation, and Servicing," International Organization for Standardization, Geneva, 1987.

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

[OMG, 2005] "Software Process Engineering Metamodel Specification," Version 2.0, August 2005, www.omg.org/cgi-bin/doc?formal/05-07-04.

[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 Solingen, 2004] R. VAN SOLINGEN, "Measuring the ROI of Software Process Improvement," *IEEE Software* **21** (May/June 2004), pp. 32–38.

[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 软件团队

学习目标

通过本章学习，读者应能：

- 解释组织一个良好的团队的重要性。
- 描述如何组织现代层级团队。
- 分析各种不同团队组织的优缺点。
- 认识选择合适的团队组织时要考虑的问题。

如果没有称职的、良好受训的软件工程师，那么软件项目注定将走向失败。然而，有了合适的人选还不够；必须组织团队，以使团队成员能够卓有成效地工作并彼此协同。本章将详细介绍团队组织。

4.1 团队组织

由于大多数产品都很庞大，一般单个软件专业人员难以在规定时间内完成，因此，产品必须委派给由一组专业人员组成的团队（team）开发。以分析工作流为例，为在2个月内明确目标产品，可能需要把这个任务分配给三个分析专家，而他们组成了一个由分析经理领导的团队。类似地，设计任务必须由设计团队的成员共同承担。

假设有一个产品，需要在3个月内完成编码工作，但它的编码量为1人年（1人年是指一个人一年可以完成的工作流）。解决方法似乎显而易见：如果一个程序员在一年内能够完成整个产品的编码任务，那么4个程序员就可以在3个月内完成。

当然，这并不可行。实际上，4个程序员可能也需要大约一年的时间，而且，相较于一个程序员编写完整个产品，他们最终完成产品的质量可能要更低。原因在于，有些任务可以协作实现，而有些则必须独立完成。例如，如果一个农夫可以在10天内采摘完一块草莓地，则10个农夫可以在1天内采摘完同一块草莓地。另一方面，一头母象能用22个月孕育一头小象，但这项工作不可能由22头母象在1个月内完成。

换句话说，类似采草莓的工作可以完全通过协作完成；而类似孕育小象的工作则根本无法协作。与孕育小象不同，通过在团队成员间分配编码工作，协同完成实现任务是可能的。然而，团队编程也不像采草莓，因为成员间需要合理并且高效的彼此交互。例如，假设Sheila和Harry需要编写两个模块m1和m2，这可能涌现不少错误。例如，Sheila和Harry都编写了m1而忽略了m2。或者Sheila编写了m1，而Harry编写了m2。但是当m1调用m2时，需要传递4个参数；而Harry编写的m2需要5个参数。或者m1和m2中参数的顺序不同。或者顺序相同但是数据类型稍有差异。当设计工作流已经实施，但还未将任务完全分发给整个开发组织时，所作的决定经常会导致这类问题的发生。无论如何，这一问题与程序员的技术能力无关。团队组织是一个管理问题，管理者必须组织好编程团队，以使每个团队都能高效运作。

图4-1显示了团队软件开发时遇到的另一类困难。在项目中，三个计算机专业人员之间存在三条沟通路径。现在假设随着工作的进行，截至期限很快将到达，但任务尚未完成。显然，需要给团队增加第四个专业人员。但是，当第四人加入团队后，对于其他三人而言，首先要做的事情就是，详细解释迄今为止哪些任务已经完成，而哪些则尚未完成。换句话说，给一个进度已经落后的团队增加成员，可能会使进度更加滞后。这个法则就是Brook法则（Brooks's Law），是Fred Brooks在管理IBM 360上的OS/360操作系统开发时观测得到的［Brooks，1975］。

在一个大型组织中，软件生产的每个工作流程都有相应的团队，尤其是在实施实现工作流时。在实现工作流执行期间，程序员独立工作而开发单独的代码制品。因此，实现工作流是几个计算机专业人员共享任务的首选。在一些更小的组织中，可能由个人负责需求、分析和设计，而后的实现则由两或三个程序员所组成的团队来完成。因为在执行实现工作流时，会更多地使用团队，所以团队组织的问题在实现时就显得更为尖锐。因此，在本章剩余部分，将从实现角度来考虑团队组织问题，这些问题及其解决方案同样适用于所有其他工作流。

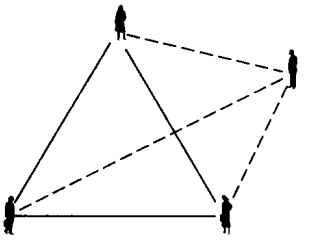


图4-1 三名计算机专业人员（如图中实线所示）以及第四人加入后（如图中虚线所示）的沟通路径

对于编程团队的组织，存在两种极端的解决方式：民主团队和主程序员团队。这里将分别描述这两种方式，阐明其优缺点，然后提出结合了两种方式优点的其他编程团队组织方式。

4.2 民主团队方式

民主团队组织最先由 Weinberg 在 1971 年提出［Weinberg，1971］。民主团队的基本原则是*无我编程*（egoless programming）。Weinberg 认为程序员可能非常沉迷于自己编写的代码。有时，甚至用自己的名字命名所编写的模块：他们因此将模块看作自我的延续。由此产生的问题是，把模块看作是自我延续的程序员，会不情愿去找出代码中存在的所有错误。如果程序中有错误，则被称为 bug，就像爬行在代码中不受欢迎的一种虫子。代码如果更为积极地被保护起来而免遭入侵，似乎就能避免这种 bug（参阅备忘录 4.1）。

备忘录 4.1

大约 40 年前，当软件还必须借助穿孔卡片输入计算机时，绝大多数程序员认为软件中的“bug”就像小虫子一样，如果不加阻止就会入侵他们的卡片。这种想法被市场上一种名为 Shoo-Bug 的烟雾喷射剂有趣地嘲讽了一番。该产品标签上的使用说明郑重其事地解释，对卡片喷射 Shoo-Bug 烟雾喷射剂，就能确保代码中不会大量滋生 bug。当然，喷雾中除了空气什么也没有。

对于程序员太过沉迷于自己编写的代码这一问题，Weinberg 的解决方法是无我编程。社会环境必须重构，程序员的价值也如此。每个程序员都必须鼓励团队的其他成员为其找出代码中的错误。不要认为出现错误是很糟糕的事情，而应当把它视为是正常并可接受的。在被征询建议时，评审人的态度应当正确客观，而不能嘲笑犯了错误的程序员。因此，团队作为一个整体形成一种风气，即群体认同；软件模块属于整个团队而非个人。

由多达 10 个无我程序员组成的小组即组成一个*民主团队*（democratic team）。Weinberg 警告说，管理这样一个团队可能有些困难。毕竟，需要考虑到管理的职业方法。当一个程序员被晋升到管理层时，其程序员同事没有得到晋升，他们必定会努力在下一轮晋升中得到更高的等级。而民主团队是一个为了共同目标而工作的小组，没有单一的领导者，不会有程序员试图晋升到下一个等级。因此，最重要的是团队认同和相互尊重。

Weinberg 谈到一个开发了优质产品的民主团队。管理层决定要给团队中名义上的管理者（根据定义，民主团队没有领导者）现金奖励。他拒绝以个人名义接受，认为应当由团队全体成员共同分享。管理层认为他是想要更多的奖励，并且认为这个团队（尤其是其名义上的管理者）想法非常怪异。管理层强迫名义上的管理者接受奖金，而他将这笔钱平均分配给团队成员。接

下来，整个团队集体辞职并加入了其他公司。

现在来讨论民主团队的优点和缺点。

民主团队方式的主要优点是对于查找错误持积极态度。找出错误越多，团队成员就越高兴。这种积极态度使得能更快检测到错误，并因此而提高软件质量。但是，存在一些主要问题。正如前面所指出的，管理层可能难以接受无我编程方式。此外，对于一个程序员，假设其拥有 15 年经验，他可能会不愿意将自己的代码交给同事（尤其是初来者）去评价。

Weinberg 觉得无我团队应当自发产生，而不能是外界强加。关于民主编程团队，很少有实验研究，但是 Weinberg 的经验证明民主团队是非常高效的。基于一般性的团队组织而非特定的编程团队的理论和所进行的实验，Mantei［1981］对民主团队组织进行了分析。她指出非集中式小组在遇到难题时效果最好，并提出民主团队在研究环境中应当能更好地发挥作用。就我个人经验而言，当有难题需要解决时，民主团队在工业场景中也能很好工作。在很多情形下，我都曾是民主团队的一员，这在有研究经历的计算机专业人员中很自然地就会形成。但是，一旦任务解决，进程进展到来之不易的实现阶段，团队就必须以更加层次化的方式重新组织，如 4.3 节将要描述的主程序员团队方式。

4.3　主程序员团队方式

考虑如图 4-2 所示的 6 人团队，它有 15 条 2 人沟通路径。实际上，2 人、3 人、4 人、5 人和 6 人小组的总数是 57。如图 4-2 所组织的 6 人团队不可能在 6 个月内完成 36 人月的工作量，其主要原因就在于沟通路径的多样性。很多时间浪费在每次只有 2 名或多名团队成员的会议上。

现在考虑如图 4-3 所示的 6 人团队。同样包含 6 个程序员，却只有 5 条联络线路。这就是现在称为*主程序员团队*（chief programmer team）的基本思想。相关思想由 Brooks［1975］提出，他用主外科医生指导手术这件事来打比方。主外科医生由其他外科医生、麻醉医师和各种护士协助。而且，如果需要的话，这个团队会使用来自其他领域的专家，例如，心脏病专家或者肾脏学专家。这个类比阐明了主程序员团队的两个主要方面。其一是*专门化*（specification）：每个团队成员仅承担其所擅长的那些任务。其二是*层级性*（hierarchy）：主外科医生指挥团队中所有其他成员的工作，并对手术的所有方面负责。

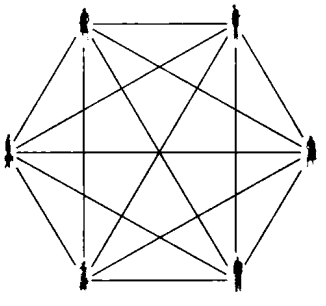


图 4-2　6 名计算机专业人员的沟通路径

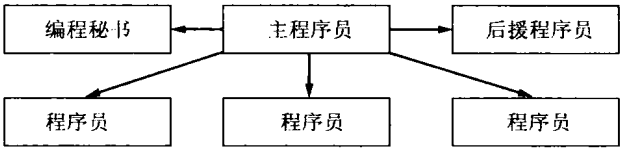


图 4-3　主程序员团队的结构

主程序员团队概念由 Mills［Baker，1972］提出。大约 30 年前，Baker 描述了一个如图 4-3 所示的主程序员团队，该团队由主程序员和协助主程序员的后援程序员、编程秘书以及 1～3 个一般程序员组成。如果需要，团队可包含其他领域的专家，例如，法律、金融事务方面的专家，或者是将操作系统指令发送给当时的大型计算机的作业控制语言（Job Control Language，JCL）方面的专家。*主程序员*（chief programmer）既是一位成功的管理者，也是一位娴熟的程序员，负责设计体系结构以及程序中任何重要或复杂的部分。在主程序员指导下，其他团队成员负责详细的设计及

编码工作。如图4-3所示，程序员之间不存在沟通路径，所有沟通事务由主程序员处理。最后，由主程序员评审其他团队成员的工作，因为主程序员个人要对程序中每一行代码负责。

后援程序员（backup programmer）是必须的，因为主程序员是一个普通人，会有生病、从公车上摔下，或者跳槽等状况。因此，后援程序员必须像主程序员一样，在每个方面都能胜任，而且对项目的了解必须跟主程序员一样多。并且，为了让主程序员集中于体系结构设计，后援程序员要完成黑盒测试用例计划（13.10节），以及其他独立于设计过程的任务。

*秘书*这个词有多种含义。一方面，秘书通过接听电话、打印信件等协助忙碌的主管人员。但是，当提到美国国务卿（American Secretary of State）或者英国外务大臣（Britich Foreign Secretary）时，指的是最高级别的内阁成员。*编程秘书*（programming secretary）不是兼职的文书助理，而是主程序员团队中娴熟、高薪的重要成员。编程秘书负责维护项目产品库以及项目文档。这包括源代码清单、JCL以及测试数据。程序员提交他们的源代码给编程秘书，编程秘书负责把代码转换成机器可读的形式，编译、链接、加载、执行以及运行测试用例。而*程序员*除了编程，不需要做其他事情，工作外的所有其他方面都由编程秘书处理（因为编程秘书维护项目产品库，有些组织也称其为*库管员*（librarian）。

记住，这里描述的是Mills和Baker的最初思想，追溯到1971年，那时还在广泛使用打孔机。现在已经不再以那种方式编程了，程序员有自己的终端或工作站，可自行输入代码、进行编辑和测试等。主程序员团队的现代形式将在4.4节中介绍。

4.3.1 《纽约时报》项目

初次使用主程序员团队概念是在1971年，当时IBM想要自动剪辑（morgue）《纽约时报》（New York Times）上的文件。剪辑的文件包括《纽约时报》和其他出版物上的摘要和全文。记者及编辑部的其他成员将使用这个信息银行作为资料索引源。

项目的实际情况很令人吃惊。例如，在22个月内写出83 000行代码（LOC），这是11人年的工作量。项目开始一年后，仅写出了包含12 000 LOC的文件维护系统。大多数代码是在后6个月编写的。在前5周的验收测试中，仅发现了21个错误；在第一年运行中，仅检测到其他的25个错误。主要编程人员平均有一个错误被检测到，并且每人的工作量为10 000LOC/人年。文件维护系统在编程结束后一个星期就被交付，在运行20个月后才发现了唯一的一个错误。几乎一半程序，在第一次编译就是正确的［Baker，1972］，它们通常包含200～400行PL/I（一种由IBM开发的语言）代码。

然而，在取得这个巨大的成功之后，却再也未见主程序员团队概念运用的类似报道。许多成功项目的运作是基于主程序员团队，虽然满意，但其所报告的数据远不如《纽约时报》项目那样令人印象深刻。《纽约时报》项目为何如此成功？而其他项目为何没有取得类似的成果？

第一，可能因为这对IBM来说是一个赢取声望的项目，它是PL/I的第一次真正亮相。从这个角度来说，IBM组织了一个“精英”的团队。第二，可能因为技术支持非常强大。PL/I编译器开发者以他们能做到的各种方式手把手地协助程序员，而且JCL专家也可以在作业控制语言方面提供帮助。第三，可能因为主程序员F. Terry Baker的专业技能。现在他被称为*超级程序员*（superprogrammer），即一个程序员的产出是优秀程序员平均产出的4～5倍。而且，Baker是一个优秀的管理者和领导者，他的技术、热情和人格可能是项目成功的潜在原因。

如果主程序员胜任，那么主程序员团队组织运行就会良好。虽然《纽约时报》项目的显著成功后无来者，但是有许多成功项目是受益于主程序员方式的变体。*用方式的变体*（variants of the approach）这个短语，原因在于，［Baker，1972］中所描述的纯的主程序员团队方式在许多方面是不切实际的。

4.3.2　主程序员团队方式的不切实际性

考虑一下主程序员，他既是一个技术娴熟的程序员，又是一个成功的管理者。这种人才非常难求，因为技术娴熟的程序员与成功的管理者都非常短缺，而主程序员的工作需要这两种才能。而且，成为技术娴熟的程序员与成为成功管理者，它们所需要的品质不同。因此，找到主程序员的机会很小。

不仅主程序员难于找到，而且后援程序员也非常少。毕竟，后援程序员被期望能像主程序员一样优秀，但他只是在等待主程序员发生事情时作为后备，而且薪水较低。几乎没有顶级程序员或者顶级管理者会接受这种角色。

编程秘书也难以找到。软件专业人员对于案头工作的厌恶是众所周知的，而编程秘书每天除了案头工作没有别的工作。

因此，主程序员团队，至少如 Baker 所提出的那种团队，是不切实际的。民主团队也是不切实际的，但其原因不同。而且，这两种技术看起来都不能处理需要 20 个（更不用说需要 120 个）程序员来执行实现工作流的产品。此时所需要的是，利用民主团队和主程序员团队的优势来组织编程团队，而且能够加以扩展而用于大型产品的开发。

4.4　超越主程序员和民主团队

民主团队的主要优势在于：查找错误时的积极态度。大量组织将主程序员团队和代码评审（6.2 节）相结合，由此导致一个潜在的缺陷。主程序员是个人对每一行代码负责，因此，必须在所有代码评审时在场。然而，主程序员也是一个管理者，而且，正如第 6 章中所说明的，评审不能用于作为任何种类的绩效评估。这样由于主程序员也是管理者，负责团队成员的主要评估工作，因此，强烈建议主程序员不要出现在代码评审现场。

这个矛盾的解决方法是移除主程序员的大量管理角色。毕竟，前面已经指出过，要找到既是技术娴熟的程序员又是成功的管理者的人才有多么的困难。取而代之，由两个人代替主程序员：管理团队技术方面工作的团队主管（team leader）和负责所有非技术管理决策的团队经理（team manager）。最终的团队结构如图 4-4 所示。很重要的一点是要认识到，这种组织结构没有违反基本管理原则，雇员不需要向多于一个管理者汇报。它清楚地描绘了不同的责任范围。团队主管只负责技术方面的管理，因此，预算和法律事务以及绩效评估也不需要团队主管处理。另一方面，团队主管对技术事务具有独立责任，因此，团队经理无权承诺 4 周内交付程序，而必须由团队主管来做类似承诺。自然地，团队主管参与所有的代码审查；毕竟，他（或她）个人负责代码的所有方面。同时，不允许团队经理参与代码评审，因为程序员绩效评估是团队经理的工作。相反，团队经理需要在定期的团队例会中了解团队每个程序员的技术技能。

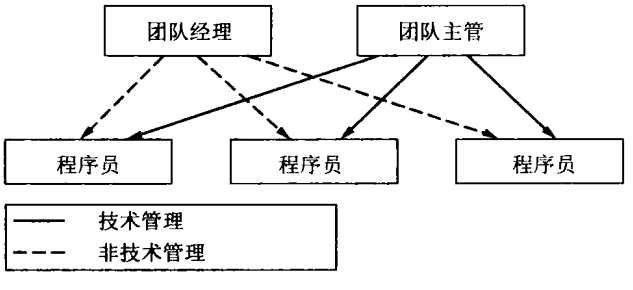


图 4-4　现代编程团队结构

在开始执行之前，清楚划分团队经理和团队主管的责任范围是很重要的。例如，年假问题，团队经理因为假期与技术无关而批准请假申请，团队主管却因为项目截至日期临近而不批准请假，这时问题就会产生。这个问题及类似问题的解决方法是在高层管理中拟定关于团队经理和团队主管共同负责领域的方针。

大型项目怎么办？如图 4-5 所示，这种方法能按比例扩展，图 4-5 给出了技术管理组织结

构。非技术方面也可类似地组织。由项目主管指导整体的产品实现。程序员向团队主管汇报，团队主管向项目主管汇报。对于更大型的产品，可以在层级中添加其他等级。

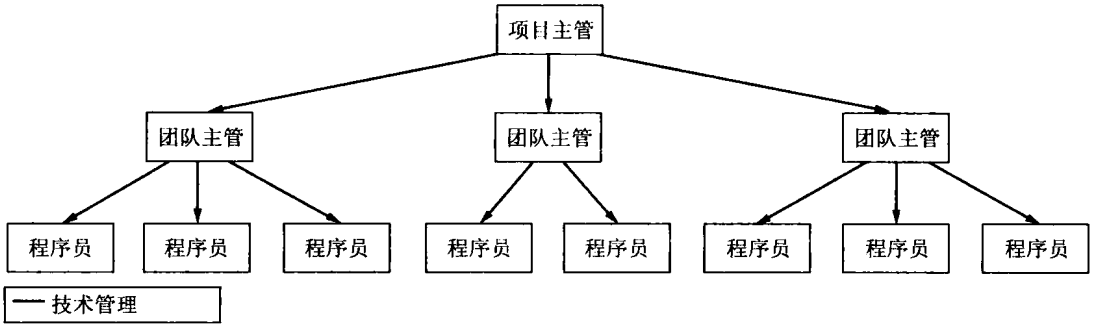


图4-5 大型项目的技术管理组织结构

另一个采用民主团队和主程序员团队优点的方式是适当分散决策过程。最终沟通路径如图4-6所示。这种方案适用于可采用民主方式的各种问题，也就是说，在研究环境下或需要团队交互协同解决难题时。虽然分散，但箭头始终是逐层向下；允许程序员听命于项目主管将只会导致混乱。

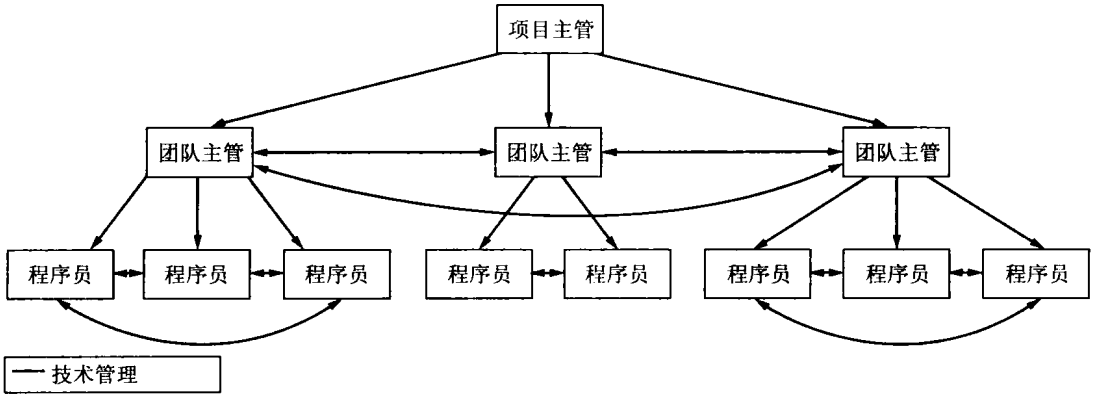


图4-6 图4-5团队组织的分散决策版本（给出技术管理的沟通路径）

4.5 同步－稳定团队

团队组织的一种可选方式是微软公司所采用的同步－稳定团队（synchronize-and-stabilize team）[Cusumano and Selby, 1997]。微软开发大型产品，例如，Windows 2000包含3 000多万行代码，参与开发的程序员和测试员超过3 000人，复用了大量Window NT 4.0的模块[Business Week Online, 1999]。对于这种规模的产品而言，团队组织是成功开发的一个非常重要的方面。

2.9.6节介绍了同步－稳定生命周期模型。该模型的成功，极大程度上是团队组织方式的结果。同步－稳定模型中3或4个顺序块中的每个都由一定数量的小型并行团队开发，每个团队由一个经理带领，包括3～8个开发人员和3～8个测试人员（与开发人员一一对应）。任务总体的规格说明会提供给团队；每个团队成员有随意设计和实现各自所负责工作部分的自由。这没有迅速导致混乱的原因是每天执行同步措施：在每天的基础上测试和调试部分完成的组件。因此，即使个人的创造性和自主性余地很大，各组件也总能协作运行。

这种方式的优点是，一方面，鼓励单个程序员的创造性和创新性，这是民主团队的一个特点。另一方面，每天的同步措施确保了几百个开发人员为一个共同的目标而合作，而不需要主

程序员团队的沟通与协同特性。

微软开发人员必须遵守的规定很少，但是为了当天的同步，他们必须严格遵守下达的时间，把程序输入产品数据库中。Cusumano 和 Selby［1997］把这比喻为：告诉孩子们，他们可以一整天做他们喜欢做的事情，但是必须晚上 9 点上床睡觉。另一个规则是，如果一个开发人员的程序在当天的同步处理中使得程序不能编译，则程序必须马上修复，以便团队的其他人可以测试和调试那天的工作。

使用同步 - 稳定模型以及相应团队组织方式，能保证每个其他软件组织也像微软一样成功吗？这是不可能的。微软公司并不只是同步 - 稳定模型。它是一个包括才华横溢的管理者和软件开发人员的组织，并且具有积极向上的组织风气。虽然其他组织使用该模型的大量特征确实可以使过程得到改进，但仅仅使用同步 - 稳定模型并不能神奇地使一个组织变成另一个微软。另一方面也表明，同步 - 稳定模型只是一个让一群计算机天才开发大型产品的方式，微软的成功是由于出色的市场而非高质量的软件。

4.6　敏捷过程团队

2.9.5 节给出了敏捷过程的概述［Bech et al.，2001］。本节描述当使用敏捷过程时，如何组织团队。

敏捷过程与众不同的特性是所有代码都由团队完成，而组成团队的每两个程序员共享一台计算机。这就是所谓的*结对编程*［Williams，Kessler，Cunningham，and Jeffries，2000］。使用这种方法的原因包括：

- 如 2.9.5 节所述，结对程序员首先拟定测试用例，然后实现对应部分的代码（任务）。在 6.6 节将解释，为什么强烈建议程序员不要测试自己的代码。敏捷过程通过团队中一个程序员拟定任务的测试用例，而另外一个结对程序员将这些测试用例应用到实现代码，来解决这一问题。
- 在更为传统的生命周期模型中，当一个程序员离开了一个项目，所有这个程序员所积累的知识也会失去。特别地，该程序员所开发的软件可能还没有归档，这时可能需要重新开发。相反地，如果结对编程团队的一个成员离开了，另一个程序员有足够的知识与一个新的结对程序员继续同一部分软件的工作。而且，新的团队如果因为做了一个错误的修改而不小心毁掉软件，测试用例的存在可以帮助找到错误。
- 通过结对而密切合作，可以使不太有经验的软件专业人员向有经验的团队成员学习技能。
- 如 2.9.5 节中所提到的，不同结对团队所使用的所有计算机都放置在一起，在一个大房间的中间。这种方式提高了团队对代码的归属感，这是无我团队的一个积极特性。

因此，即使两个程序员使用同一台计算机合作的想法有些与众不同，实践中还是有其独到的优势。

4.7　开源编程团队

令人惊讶的是，任何开源项目都取得了成功，更不用说一些最为成功的软件产品，它们曾经也是经由开源生命周期模型开发出来的。终究，开源项目的团队成员通常是一些无偿志愿者。他们异步交流（通过电子邮件），不需参与团队会议，并且也没有管理者，即各个方面都不太正式。而且，不存在规格说明或设计；实际上，即使是成熟项目，也很少存在任何形式的文档。尽管存在这种实际上难以逾越的障碍，但是像 Linux 和 Apache 等一小部分开源项目还是取得了巨大的成功。

个人志愿者参与开源项目主要有两个原因：完成有价值工作时的绝对愉悦，以及增加经验。

- 为吸引志愿者到一个开源项目并保持他们的兴趣，必须让他们一直觉得项目是“值得做

的”。除非真正相信项目一定会成功，且产品将被广泛应用，否则，个人不太可能会为一个项目投入大量业余时间。要是参与者开始认识到项目没有多少价值，他们就会慢慢离开。

- 对于第二个原因，许多软件专业人员参与到一个开源项目中，是为了获取技术方面的新技巧。例如，一种时新的编程语言，或是他们不熟悉的操作系统。他们可以利用获取的知识在自己的组织中获得晋升，或者在其他组织谋得更高的职位。毕竟，较之于通过学校学习获得的其他学历，雇主通常还是更为青睐在成功的大型开源项目中所获得的经验。但是，如果项目最终走向了失败，则投入的几个月辛苦工作也将付诸东流。

换句话说，除非项目一直被认为是成功的，否则，它将难以吸引并维持志愿者为其工作。而且，开源团队成员必须始终感到他们是在有所贡献。因此，发起开源项目的关键人物必须是一名出色的策动者，否则，项目将注定失败。

成功开源开发的另一个先决条件是团队成员的技能。正如9.2节将详细解释的那样，程序员之间存在着技能水平的巨大差异。回想一下本节第一段列出的开源软件产品走向成功的障碍，事实上，除非核心小组（2.9.4节）成员非常优秀且具有精雕细琢的高超技能，否则一个开源项目难以成功。这种顶级人才在任何环境下都能产生，包括在开源团队那样无组织的团队中。

总之，一个开源项目的成功取决于目标产品的本质、策动者的人格魅力以及核心小组成员的才能。本质上，它与开源团队的组织方式成功与否无关。

4.8 人力资源能力成熟度模型

人力资源能力成熟度模型（P-CMM）刻画了一个组织中人力资源管理和开发的最佳实践[Curtis, Helfey and Miller, 2002]。正如软件能力成熟度模型SW-CMM（3.13节）那样，为了不断提高员工技能并造就高效团队，一个组织需要通过5个成熟度等级逐级进步。

每个成熟度等级都有自己的关键过程域，一个组织想要达到相应的成熟度等级，必须满足每个关键过程域。例如，对于等级2（已管理级），KPAs包括人员配置、沟通与协同、工作环境、绩效管理、培训与发展，以及补偿。相反，等级5（优化级）的KPAs则是持续能力提升、组织绩效调整以及持续的人力资源革新。

SW-CMM是改进组织软件过程的一个框架，它没有提供具体的过程或方法学。同样，P-CMM是提高组织人力资源管理和开发的一个框架，它也没有为团队组织提供具体方法。

4.9 选择合适的团队组织

各种团队组织方式的比较如图4-7所示，图中给出了描述每种团队组织方式所在的章节。遗憾的是，任何解决方式都未能解决编程团队组织的问题，或者扩展开去，也没能解决所有其他工作流中团队组织的问题。组织一个团队的最佳方式依赖于要开发的产品、对不同团队结构的先验知识，并且最为重要的是，依赖于组织的文化氛围。例如，如果高层管理不适合分散决策，那么就不要使用相关方式。

实际上，现在的大多数团队都是按4.4节描述的那样进行组织。也就是说，主程序员团队的某些变体是通用的方法。

关于软件开发团队的组织形式，没有太多的研究，多数通常所接受的原则都是基于群体动力学研究，而不是软件开发团队研究。即使已经开始基于软件团队的研究，其样本规模通常也很小，因此，结果并不令人信服。

如果不能得到软件产业中团队组织的实验结果，那么针对特定产品，就不容易确定最优团队组织方式。

团队组织方式	优　点	缺　点
民主团队（4.2 节）	积极查找错误而得到高质量代码 特别适用于解决难问题	有经验成员反感新手的评价 不可外部强加
经典主程序员团队（4.3 节） 改进的主程序员团队（4.3.1 节）	《纽约时报》项目的巨大成功之处 许多成功案例	不切实际 没有可比于《纽约时报》项目的成功案例
现代层级式编程团队（4.4 节）	采用团队经理/团队主管结构，不需要主程序员 可扩展 必要时支持分散决策	团队经理和团队主管的责任不明确时，容易产生问题
同步－稳定团队（4.5 节）	鼓励创造性 确保大量开发人员为共同目标协作	微软公司以外尚无该方法的应用案例
敏捷过程团队（4.6 节）	程序员不需要测试自己的代码 如果一个程序员离开不会有知识损失 缺少经验的程序员可以向他人学习 代码归小组所有	几乎没有关于绩效的依据
开源团队（4.7 节）	少数几个项目非常成功	应用范围很窄 必须有一个出色的策动者 需要高水平参与者

图 4-7　团队组织方式比较及每种方式对应的章节

本章回顾

在考虑团队组织（4.1 节）问题时，首先讨论的是民主团队（4.2 节）和主程序员团队（4.3 节）。《纽约时报》项目（4.3.1 节）的成功与主程序员团队的不切实际性（4.3.2 节）形成了对比。4.4 节建议采用一种结合两种方式优点的团队组织方式。而 4.5 节则描述了同步－稳定团队（由微软采用）。敏捷过程团队和开源软件团队分别在 4.6 节和 4.7 节进行了讨论。4.8 节描述了人力资源能力成熟度模型（P-CMM）。最后，针对给定项目，4.9 节描述了选择最优团队组织方式所包含的因素。

延伸阅读材料

团队组织方面的三个经典著作是［Weinberg，1971；Baker，1972；和 Brooks，1975］。这一领域较新的著作包括［DeMacrco and Lister，1987］和［Cusumano and Selby，1995］。关于团队交互如何演化的有趣描述可以在［Mackey，1999］中找到。1993 年 10 月出版的《Communications of the ACM》杂志包含关于团队组织和管理方面的文章。［Royce，1998］的第 11 章给出了团队成员角色承担方面的有用信息。一种值得称道的方法是，使用人格类型分析方法来选择团队成员，参见［Gorla and Lam，2004］。

在［Cusumano and Selby，1997］中给出了同步－稳定模型的概述，并在［Cusumano and Selby，1995］加以详细描述。［McConnell，1996］也深入分析了同步－稳定模型。极限编程团队的描述可参见［Bech，2000］。2003 年 5/6 月出版的《IEEE Software》期刊发表了一些关于极限编程的论文，特别是［Reifer，2003］和［Murru，Deias and Mugheddue，2003］。其他关于敏捷过程的观点，可以在［Boehm，2002］和［DeMarco and Beohm，2002］，以及 2005 年 5/6 月《IEEE Software》期刊论文中找到。William、Kessler、Cunningham 和 Jeffries［2000］描述了一个结对编程实验，结对编程是极限编程的一部分。［Drobka，Noftz，and Raghu，2004］总结

了结对编程的优缺点。[Curtis, Hefley and Miller, 2002] 对 P-CMM 进行了分析。

习题

4.1 要开发一个工资单项目，如何组织一个团队？请加以解释。

4.2 要开发当前的军事通信软件，如何组织一个团队？请加以解释。

4.3 假设你刚创办一家新的软件公司。所有雇员都是新来的研究生；这是他们的第一份编程工作。在这样的公司中，民主团队方法可能实现吗？如果可行，如何实现？

4.4 采用民主方式组织一个学生编程团队。关于团队中的学生，你能得出什么结论？

4.5 采用主程序员方式组织一个学生编程团队。关于团队中的学生，你能得出什么结论？

4.6 在一个大型软件公司中，为了比较两种不同的团队组织方式 TO_1 和 TO_2，给出以下实验。由两个不同的团队开发同样的软件产品，一个团队的组织方式是 TO_1，另一个是 TO_2。公司预计，每个团队将用 18 个月开发产品。给出三个理由，阐述为什么这个实验是不切实际的，并且不可能得到有意义的结果。

4.7 为什么敏捷过程中的结对编程团队需要共享一台计算机？

4.8 民主团队和开放源码团队之间有何差异？

4.9 你愿意工作在以同步 - 稳定方式组织的团队中吗？请给出解释。

4.10 （学期项目）要开发附录 A 中所述的 Osric 办公用品和装饰产品，采用什么类型的团队组织方式更为合适？

4.11 （软件工程读物）教师分发 [Drobka, Noftz, and Raghu, 2004] 论文的复印件。讨论话题：你想成为极限编程团队的一员吗？请解释原因。

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.

[Drobka, Noftz, and Raghu, 2004] J. Drobka, D. Noftz, and R. Raghu, "Piloting XP on Four Mission-

Critical Projects," *IEEE Software* **21** (November/December 2004), pp. 70–75.

[Gorla and Lam, 2004] N. GORLA AND Y. W. LAM, "Who Should Work with Whom?" *Communications of the ACM* **47** (June 2004), pp. 79–82.

[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.

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

[Reifer, 2003] D. REIFER, "XP and the CMM," *IEEE Software* **20** (May/June 2003), pp. 14–15.

[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

通过本章学习，读者应能：

- 认识逐步求精的重要性，并能够在实际中使用。
- 应用成本 - 效益分析。
- 选择合适的软件度量。
- 讨论 CASE 的范围和分类。
- 描述版本控制工具、配置控制工具和构建工具。
- 理解 CASE 的重要性。

软件工程师需要两种工具：一种是软件开发过程中应用的分析工具，例如，逐步求精和成本 - 效益分析；另一种是帮助软件工程师团队开发和维护软件的工具，通常称为 CASE 工具（CASE 是 computer-aided software engineering 的首字母缩写）。本章主要讨论这两种工具：理论（分析）工具和软件（CASE）工具。先从逐步求精开始。

5.1　逐步求精

2.5 节介绍的逐步求精是一种解决问题的技术，是许多软件工程技术的基础。*逐步求精*可以定义为一种方式：尽量延迟细节的确定，以便集中精力考虑重要问题。依照米勒法则（2.5 节），人们一次大约只能关注 7 件（信息单位）事情。因此，应使用逐步求精技术来推迟一些不太重要的决定，而将注意力集中在关键问题上。

从本书可以看到，逐步求精是分析技术、设计和实现技术，甚至是测试和集成技术的基础。逐步求精在面向对象范型的研究范围内是极其重要的，因为潜在的生命周期模型是迭代和增量的。

下面的小型案例研究阐述了如何在产品设计中使用逐步求精。

这一节的小型案例研究似很平凡只涉及一个顺序主文件的更新，但这是一个在许多应用领域通用的操作。选择这个简单而熟悉的例子是为了让你关注逐步求精而不是应用领域。

设计一个产品，它为月刊《True life Software Disasters》更新包含订阅者姓名和地址的顺序主文件。有三种事务：插入、修改和删除，分别记以 1、2、3。事务类型有：

类型 1：插入（一个新的订阅者到主文件）。

类型 2：修改（一个存在的订阅者记录）。

类型 3：删除（一个存在的订阅者记录）。

事务按订阅者名字的字母顺序排序。如果一个给定的订阅者有多于一个事务，则需要对该订阅者的事务重新排序，以便使插入发生在修改前、修改发生在删除前。

设计解决方案的第一步是建立一个有代表性的输入事务文件，如图 5-1 所示。这个文件包含 5 条记录：删除 Brown、插入 Harris、修改 Jones、删除 Jones 和插入 Smith（其实在一次运行中对同一个订阅者执行修改和删除并不奇怪）。

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

图 5-1　顺序主文件更新对应的输入事务记录

这个问题如图 5-2 所示，有两个输入文件：

1）旧的主文件名和地址记录。

2）事务文件。

还有三个输出文件：

3）新的主文件名和地址记录。

4）异常报告。

5）摘要和工作结束信息。

现在开始设计过程，出发点选择为图 5-3 中的“更新主文件”方框。将这个方框分解成三个方框：输入、处理和输出。假设，当处理需要一条记录时，在该事件执行时就能产生正确的记录。类似地，还要能够在正确的时间将正确的记录写进正确的文件。因此，这个技术是为了能将输入和输出分离，并且把主要精力放在处理上。处理是什么？为了认识它是做什么的，考虑一下图 5-4 中的例子。第一条事务记录（Brown）的关键词与第一条旧的主文件记录（Abel）的关键词作比较。因为 Brown 位于 Abel 后，因此就将 Abel 记录写进新的主文件，并且读取下一条旧的主文件记录（Brown）。在这条例子中，这条事务记录的关键词与旧的主文件记录的关键词匹配，而且因为事务类型是 3（删除），所以必须删除 Brown 记录。这可通过不将 Brown 记录复制到新的主文件加以实现。读取下一条事务记录（Harris）和旧的主文件记录（James），在它们各自的缓冲中重写 Brown 记录。Harris 位于 James 之前，因此，将其插入到新的主文件；同时读取下一条事务记录（Jones）。因为 Jones 在 James 之后，将 James 记录写入新的主文件，并且读取下一条旧的主文件记录，即 Jones。就像在事务文件中看到的一样，过程将会修改 Jones 记录，然后删除它，并读取下一条事务记录（Smith）和下一条旧的主文件记录（也是 Smith）。遗憾的是，这个事务类型是 1（插入），但是 Smith 已经在主文件中了。因此数据中会有某些类型错误，并且将 Smith 记录写入异常报告。更精确点地说，将 Smith 事务记录写入异常报告，并且将 Smith 旧的主文件记录写入新的主文件。

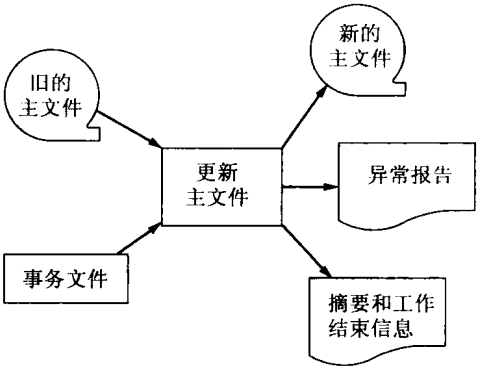


图 5-2　顺序主文件更新描述

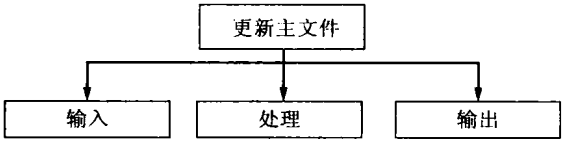


图 5-3　设计的第一次求精

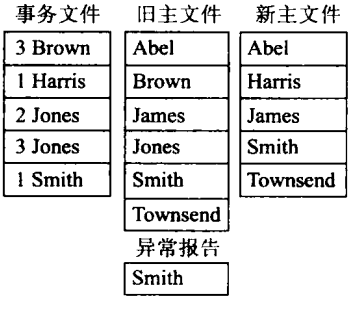


图 5-4　事务文件、旧主文件、新主文件和异常报告

既然这个过程已经为大家所了解，它可以用图 5-5 来表示。接下来，可以对图 5-3 的处理方框求精，产生出图 5-6 所示的第二步求精。连到输入和输出方框的虚线表示对输入和输出的处理将在后面的求精过程进行。这个图的剩余部分是处理

事务记录关键词=旧的主文件记录关键词	1.插入:打印错误消息 2.修改:改变主文件记录 3.删除:*删除主文件记录
事务记录关键词>旧的主文件记录关键词	将旧的主文件记录复制到新的主文件
事务记录关键词<旧的主文件记录关键词	1.插入:将事务记录写进新的主文件 2.修改:打印错误消息 3.删除:删除主文件

图 5-5　处理过程描述

* 删除一条主文件记录是通过不将这条记录复制到新的主文件来实现的

过程的流程图，或者说，流程图的一个早期的求精。就像早已指出的那样，输入和输出都推迟了。而且，对于一个文件的结束条件没有任何规定，当遇到一个错误条件时也没有规定要如何处理。逐步求精的优点是这些问题和类似的问题能够在将来的求精过程中解决。

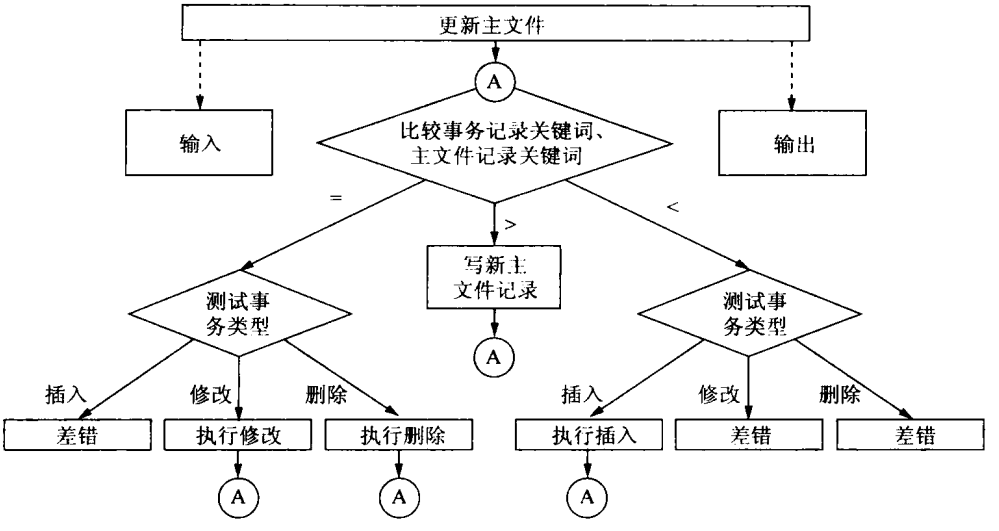


图 5-6　设计的第 2 次求精

下一步是求精图 5-6 的输入和输出方框，产生图 5-7。文件结束条件和任务结束消息的条件仍然没有得到处理。这些任务仍然可以在以后的迭代中完成。然而，关键点是图 5-7 的设计有一个大的错误。为了更好地理解这一点，考虑当前的事务是 2 Jones 时的情况，也就是说，修改 Jones，而且当前的旧的主文件记录是 Jones。在图 5-7 的设计中，因为这条事务记录关键词和旧的主文件记录关键词相同，最左边的路径到达“测试事务类型”判断方框。因为当前的事务类型是修改，修改旧的主文件记录并写入新的主文件，并且读取下一条事务记录。这条记录是 3 Jones，也就是说，删除 Jones。但是已将修改后的 Jones 记录写入了新的主文件中。

读者可能会奇怪为什么故意给出一个不正确的求精。答案是当使用逐步求精时，在进行下一步前，必须检查每一个阶段性求精的正确性。如果某次求精出错，则不必从头开始重做，而只需要回到先前的求精，然后从那里开始即可。在这个实例中，第 2 次求精（图 5-6）是正确的，因此可以将它作为再次进行第 3 次求精的基础。这一次，设计使用 1 级前瞻（lookahead），也就是说，只有分析了一个事务的下一条事务记录后，才能处理该事务记录。具体细节留做练习，见习题 5.1。

在第 4 次求精时，必须要处理一些一直忽略的细节（如打开和关闭文件）。在逐步求精的过程中，最后处理这些细节，即在设计的逻辑完全开发后才处理。显然，不可能在没有打开和关闭文件的情况下执行产品。然而，这里重要的是设计过程中的这个阶段才来处理类似文件打开和关闭这些细节。当处于设计过程中时，设计者能够立刻注意的大约 7 个程序块是不应该包括类似打开和关闭文件这些细节的。文件的打开和关闭与设计本身没有什么关系，它们只是任何设计部分的实现细节。然而在后来的求精中，打开和关闭文件变得非常关键。换句话说，可以认为逐步求精是一种技术，用来设置各种工作流程中需要解决的问题的优先级。逐步求精保证每个问题都能得到解决，而且在合适的时间解决，不需要一次处理超过 7 ± 2 个程序块。

*逐步求精*最早是由 Wirth［1971］引入的。在前面的小型案例研究中，将逐步求精应用于流程图，而 Wirth 将这项技术应用于伪代码。应用逐步求精的具体形式并不重要，逐步求精是一种通用的技术，可以用在每一个工作流程中，表现形式也可以多种多样。

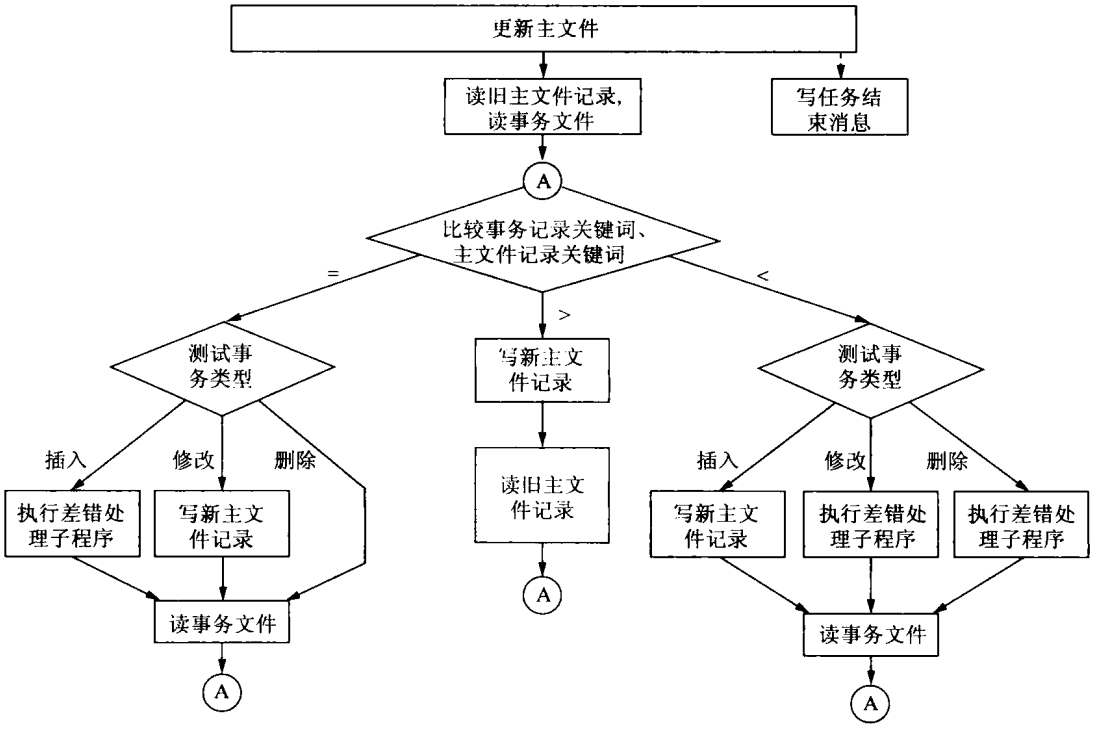


图 5-7　设计的第 3 次求精（设计有一个主要错误）

米勒法则是人类精神力量有限性的描述。人类无法抗拒大自然法则，我们必须适应大自然，接受人的能力有限的观念，并在这种限制下做到最好。

逐步求精的力量在于，它帮助软件工程师关注当前开发任务的相关方面，忽略那些尽管在整体规划中必要，但现阶段不需考虑的问题。不同于分而治之技术——将问题作为整体分解成必要的同等重要的子问题，在逐步求精中，问题的特定方面的重要性在求精过程中不停地改变。最初，一个特定的要素可能不相关，但是后来却变得相当重要。逐步求精的挑战在于，决定哪些问题在当前求精过程中必须解决，哪些应该延迟到以后的求精过程中解决。

与逐步求精一样，成本 - 效益分析法是另一种贯穿在软件生命周期的基本的理论软件工程技术。这项技术将在 5.2 节介绍。

5.2　成本 - 效益分析法

确定一个可能的行为过程是否会有盈利，一种方法是比较未来收益和预期花费，这称为*成本 - 效益分析法*（cost-benefit analysis）。举一个计算机领域中的成本 - 效益分析法的例子，1965 年，Krag 中央电子公司（KCEC）决定是否应将它的付费系统计算机化。当时，账单由 80 名职员手工处理，每隔两个月将帐单寄给 KCEC 公司的客户。计算机化将要求 KCEC 购买或者租用必要的硬件和软件，包括在打孔卡和磁带上记录输入数据的数据采集设备。

计算机化的一个优势是账单能够每月一寄，而不是每两个月，从而显著提高了公司的资金周转速度。而且，80 名处理账单的职员将由 11 名数据采集职员代替。就像图 5-8 中所示，在接下来的 7 年中，工资结余估计有 1 575 000 美元，并且改进的资金流动带来的收益预计为 875 000 美元。因此，总的收益估计有 2 450 000 美元。另一方面，要高薪雇用一些计算机专家成立一个完整的数据处理部门。在 7 年中，成本估计如下：硬件和软件的成本，包括交付后的维护开销，估计有 1 250 000 美元。在第一年中，将会有 350 000 美元的转变耗费，并且向顾客解释新系统

的额外成本估计有125 000美元。总的花费估计有1 725 000美元，大约比预计收益少750 000美元。于是KCEC立即决定计算机化。

收益(美元)		成本(美元)	
薪水结余(7年)	1 575 000	硬件和软件(7年)	1 250 000
改进的资金流动(7年)	875 000	转换成本(仅第一年)	350 000
		向顾客解释(仅第一年)	125 000
总收益	2 450 000	总成本	1 725 000

图5-8 KCEC的成本－效益分析数据

成本－效益分析并不总是这样直接的。一方面，管理顾问能够估计薪水结余，会计能预计资金流动的改进，净现值（net present value，NPV）能用来处理资金成本中的变化，软件工程顾问能估算硬件、软件和转型的成本。但是如何确定让顾客适应计算机化所需要的相关成本呢？而且倘若考虑市场因素，又该如何估计收益？也就是说，第一个将新产品推向市场所带来的收益，或者不是第一个的代价（因为失去了顾客）？

问题是有形的收益容易衡量，无形的收益很难直接计算。一种实际的用来确定无形收益的方法是*假设*（assumption）。这些假设必须与得到的收益估算结合起来。毕竟，管理者不得不做决定，而在没有实际数据可用的情况下，通过假设确定数据通常是最好的选择。这种方法的另一个优点是，如果其他人重新审视这些数据，并在潜在的假设基础上提出更好的假设，那么便能够产生更好的数据，这样，相关的无形收益就能够计算得更加精确。对于无形成本的计算可以使用同样的技术。

成本－收益分析法是确定一个顾客是否应该将其业务计算机化的基本技术，另外，如果需要计算机化，应该采用何种方式对可选策略的成本和收益进行比较。例如，存储药品实验结果的软件产品可以有多种，包括直接文件存储和各种数据库管理系统。对于每个可能的策略，都要计算成本和收益，并且选择收益和成本的差值最大的作为最优策略。

这章介绍的最后一个理论工具是软件度量。

5.3 软件度量

正如在3.13节所介绍的，如果没有*度量*，我们不可能在软件过程的早期，在问题暴露之前就发现它们。如此，度量就可以作为对潜在问题的早期警告系统。有多种度量可以使用。例如，代码行数（LOC）是一种度量产品规模的方法（9.2.1节）。如果定期进行LOC度量，则可给出工程的进展速度。另外，每1 000行代码的错误数是软件质量的度量。然而，如果一个程序员一个月内连续写出2 000行代码，但是其中的一半因为不可用而必须丢弃，那么这是没有用的。因此，孤立状态下的LOC不是一个很有用的度量。

一旦将产品安装在用户的计算机上，像平均故障间隔时间这样的度量就可以提供给管理者一个可靠性的指示。如果一个特定的产品每隔一天出现一次故障，它的质量肯定比同类的平均运行9个月而不出错的产品差。

某种度量可能应用于整个软件过程。例如，对于每个工作流，可以以每人月（一每人月是一个人在一个月的工作量）来度量效率。职员的流动性是另一个重要的度量。高流动性对当前工程有消极影响，因为新的雇员需要时间来了解学习工程的相关情况（4.1节）。另外，新的雇员可能需要软件工程相关方面的培训；如果新的雇员比其所替代的人受的软件工程方面的教育少，那么整个过程都可能受到影响。当然，成本也是一个必要的度量，也必须在整个过程中持续受到监控。

本书描述了几种不同的度量。一些是*产品度量*（product metric），用来描述产品本身的某些

方面，例如，它的大小和可靠性；另外一些是过程度量（process metric），开发者用这些度量来推断关于软件开发过程的信息。这种度量的一个典型例子是开发过程中错误探测的有效性，也就是说，开发过程中的错误检测数与产品的生命周期中的总错误检测数之比。

许多度量对于给定的工作流是特定的。例如，代码行不能用在实现工作流之前，审核规格说明的每小时错误检测数仅仅与分析工作流相关。在接下来的章节中将讨论软件过程中不同的工作流，以及与工作流相关的度量。

收集数据来计算度量值是需要成本的。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，并且解释这个工具的输出也消耗人力资源。在目前提出的数百种（如果不是上千种的话）度量中，一个明显的问题是，软件组织应该衡量什么？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（如那些在 9.2.1 节中介绍的）度量来计算）。
2）成本（以美元为单位计算）。
3）持续时间（以月为单位计算）。
4）工作量（以人月为单位计算）。
5）质量（以探测到的错误数量来计算）。

这些度量中的每一种都必须以工作流来衡量。以来自这些基本度量的数据为基础，管理者可以发现软件组织内部的问题，例如，设计工作流中的高错误率或者远低于行业平均水平的代码输出。一旦发现问题，就可以考虑解决问题的方法。为了监测这种策略的正确性，可以引入更详细的度量。例如，收集关于每个程序员的错误率数据或者进行用户满意度调查。因此，除了 5 种基本的度量外，针对一个特定的目标，还要进行更详细的数据采集和分析。

最后要指出的是，度量仍有一个饱受争议的方面，那就是关于一些普遍使用的度量的正确性问题。这些话题将在 13.12.2 节讨论。尽管普遍认为只有度量软件过程，才能控制软件过程。但在应该度量什么的问题上仍然存有争议［Fenton and Pfleeger，1997］。

现在从理论工具转向软件工具（CASE）。

5.4　CASE

在软件产品的开发过程中，会执行几种不同的操作。典型的操作包括估计资源需求、制作规格说明文档、执行集成测试，以及写用户手册。遗憾的是，这些操作和软件过程中其他操作都不是全自动的，需要人的参与。

然而，计算机能够为开发的每一步提供帮助。这节的标题“CASE”代表计算机辅助软件工程（见备忘录 5.1）。计算机能够帮助完成许多与软件开发有关的繁重工作，包括产生和组织各种人造产品，例如，计划、合同、规格说明、设计、源代码，以及管理信息。文档对于软件开发和维护是必要的，但是软件开发中的大部分人都不喜欢制作和更新文档。在计算机上维护图表尤其有用，因为它可以很方便地修改。

但是，CASE 并不只限定于帮助编写文档。特别地计算机能帮助软件工程师处理软件开发的复杂性，尤其是管理所有的细节。CASE 包括计算机支持的软件工程的所有方面。同时，更重要的是，CASE 代表计算机辅助软件工程，而不是计算机自动化软件工程，现阶段没有一台计算机能够在软件开发和维护方面代替人类。至少在可预见的将来，计算机肯定仍是软件专业人员的工具。

备忘录 5.1

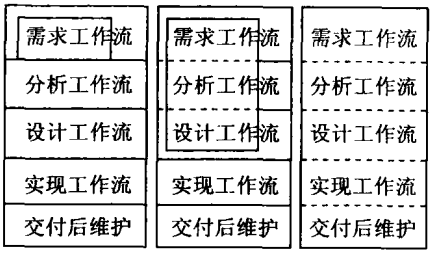
正如 1.11 节中介绍的，对于软件工程师，系统这个名词经常用来表示软件和硬件的结合体。系统工程包括的领域很广泛，由定义顾客的需要和要求开始，直到在构造的系统中充分实现它们。接着，把系统交付给顾客，在经过成功测试之后，它在整个生命周期中经过大

量的修改，从而去除缺陷，或者加入一些需要的改进，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便有很大的相似之处。因此对于软件工程师，CASE是“计算机辅助系统工程”的首字母缩写也就不足为怪了。由于软件经常在系统工程中起着主要作用，在系统工程领域中，有时候很难知道CASE是哪个版本的缩写。

5.5 CASE的分类

CASE最简单的形式就是软件工具，它是有助于软件生产的某一个方面的软件产品。目前，CASE工具被用于软件生命周期的每一个工作流。例如，市场上有多种工具可以用于个人电脑，以帮助构造软件产品的图形（如流程图和UML图等）。那些在早期工作流（需求、分析、设计工作流）中帮助开发者的CASE工具有时候称为*高端CASE*或者*前端*（front-end）*工具*，而那些有助于实现工作流和交付后维护的称为*低端CASE*或者*后端*（back-end）*工具*。图5-9a表示一个在需求工作流中起辅助作用的CASE工具。



a) 工具　b) 工作平台　c) 环境描述

图5-9 工具、工作平台和环境描述

CASE工具中重要的一类是*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。一个大的产品包括数以万计的数据项，且计算机很适合存储像变量名、类型、每个定义的方位、过程名、参数及其类型这样的信息。每个数据字典记录的重要部分是对该项目的描述。例如，“该方法以新生儿体重为输入，计算的是正确的药物剂量”或者“以时间先后顺序列出各飞机到达的时刻列表”。

数据字典与*一致性检查器*（consistency checker）结合在一起会增强其功能。一致性检查器是一个工具，用来检查规格说明文档中的每个数据项是否都反映在设计中，相反，检查是否设计中的每个项在说明文档中都有所定义。

数据字典的另一个用途是为*报表生成器*（report generator）和*屏幕生成器*（screen generator）提供数据。报表生成器是用来产生生成报表所需的代码。屏幕生成器用来帮助软件开发者生成用于屏幕上数据定位所需的代码。假设要设计一个屏幕，用来输入连锁书店每个分店的周销售情况。分店编号是介于1000～4500或8000～8999的四位整数，在离屏幕上方三行的地方输入。将这个信息送给屏幕生成器，屏幕生成器就会自动生成代码，以在离屏幕上方三行的地方显示字符串BRANCH NUMBER_ _ _ _，并使光标定位于第一个下划线处。用户输入每个数字，系统都将先显示，然后将光标移动到下一个下划线处。屏幕生成器也会生成代码，以检测用户输入的是数字且确认输入的四位整数在规定的范围之内。如果输入的数字不符合要求或者用户按下了“?”键，则显示帮助信息。

使用生成器能够快速构建实现。进一步说，结合图形表示工具、数据字典、一致性检查器、报表生成器、屏幕生成器可以构建出需求、分析和设计的*工作平台*（workbench），以支持前三个核心工作流。Software through Pictures[⊖]就是一个结合了所有这些特性的商用工作平台。

工作平台的另一个例子是需求管理工作平台。该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。例如，RequisitePro就是这样的一个商用工作平台。

⊖ 本书有时会提到特定的CASE工具，但这并不意味着该工具为作者或者出版商所采用。本书提到的每一种CASE工具均是CASE工具的某种典型示例。

因此，一个 CASE 工作平台是一个工具的集合，它支持一个或者两个*活动*（activity），活动是指相关任务的集合。例如，编码活动包括编辑、编译、链接、测试和调试。活动与生命周期模型的工作流不一样。事实上，一个活动的任务甚至能跨越工作流的界线。例如，工程管理工作平台用于工程的每个工作流，编程工作平台能够用于概念验证原型，也可用于实现工作流和交付后维护。图 5-9b 代表一个高端 CASE 工具的工作平台。该工作平台包括图 5-9a 的需求工作流工具，还包括用于分析和设计工作流部分的工具。

将 CASE 技术从工具到工作平台的发展再继续下去，下一项就是 CASE *环境*（environment）。与支持一两个活动的工作平台不同，*环境*支持整个软件过程，或者至少支持该软件过程的大部分［Fuggetta，1993］。图 5-9c 描绘了一个支持软件生命周期的所有工作流的各个方面的环境。第 13 章将会更详细地讨论环境。

在建立了 CASE 的分类后（工具、工作平台和环境），下面讨论 CASE 的范围。

5.6　CASE 的范围

正如前面所提到的，使用 CASE 技术的一个主要原因就是能够随时拥有精确的和实时更新的文档。例如，假设规格说明是手动产生的，开发团队的成员没有办法区分一份特定的规格说明文档是当前版本还是更旧一些的版本；也没有办法知道文档的手动修改是当前规格说明的一部分，还是仅仅是后来被否决的建议。另一方面，如果产品的规格说明是使用 CASE 工具产生的，那么在任何时间都只有规格说明的一个副本，团队成员可以通过 CASE 工具访问在线版本。由此，如果规格说明被修改，开发团队的成员就能够轻松地获得这些文档，并且肯定他们看到的是当前的版本。另外，一致性检查器会标记任何和规格说明文档不一致的改变。

程序员也需要在线文档（online documentation）。例如，操作系统、编辑器、编程语言等必须提供在线帮助信息。另外，程序员必须查询多种手册（如编辑器手册和编程手册）。不论在哪里，这些手册必须是在线的。除非所有东西都在手边，否则，用计算机进行查询比试图找到合适的手册并翻到需要的页面要快得多。另外，通常对在线文档进行修改要比先找出手册的所有硬拷贝再对需要的页面进行替换容易得多。因此，在线文档似乎比同样材料的硬拷贝文档更加精确，这是为程序员提供在线文档的另一个原因。在线文档的一个例子是 UNIX 手册［Sobell，1995］。CASE 也有助于团队成员之间的交流。*电子邮件*就像电脑或者传真机一样成为了办公室的一部分。使用电子邮件有很多优点。从软件生产的观点来看，在特定的邮箱里存储的所有与项目有关的邮件的拷贝提供了在工程进行期间所做决定的一份书面记录。它能够解决将来可能发生的冲突。许多 CASE 环境和一些 CASE 工作平台现在都集成了电子邮件系统。在其他的组织中，电子邮件系统是通过万维网浏览器（如 Netscape 或者 Firefox）来实现的。另外一些必要的工具是*电子数据表*（spreadsheet）和*文字处理器*（word processor）。

编程工具（coding tool）指的是诸如文字编辑器、调试器和灵巧打印机等 CASE 工具，用来简化程序员的任务，减少程序员在工作中所受的失败，并且提高程序员的效率。在讨论这些工具前，需要明确三个定义：*小型编程*（programming-in-the-small）指的是单一模块代码级的软件开发；而*大型编程*（programming-in-the-large）指的是模块级的软件开发［DeRemer and Kron，1976］，它包括如设计架构和集成的一些方面；*多人编程*（programming-in-the-many）指的是以团队进行的软件生产。团队工作有时在模块级，有时在代码级。相应地，多人编程集成了大型编程和小型编程的各个方面。

结构编辑器（structure editor）是一个“懂得”实现语言的文字编辑器。也就是说，一旦程序员输入字符串，结构编辑器就能探测出是否有语法错误，由于减少了浪费在琐碎的编译上的时间，因此可加速程序的实现。结构编辑器存在于各种语言、操作系统和硬件上。因为结构编辑器能理解编程语言，因此可将一个*灵巧打印器*（pretty printer 或*格式器*（formatter））集成到

编辑器，以保证代码有一个良好的视觉外观。例如，C++ 的灵巧打印器可保证每一个“{”都和对应的“}”缩进相同的列数，并将保留字自动改为粗体，以便突出显示，而缩进有助于可读性。如今，这种类型的结构编辑器构成了大量编程工作平台的一部分，例如，Visual C++ 和 JBuilder。

现在来考虑在代码中调用一个方法的问题，只有在链接时候才能发现方法不存在，或者被错误定义了。因此需要结构编辑器支持*在线接口检查*（online interface checking）。也就是说，结构编辑器知道程序员声明的每一个变量的名字，也必须知道产品中定义的每一个方法的名字。例如，如果程序员输入一个调用：

```
average = dataArray.computeAverage (numberOfValues);
```

但是方法 computeAverage 还没有被定义，那么编辑器立刻返回一个消息：

```
Method computeAverage not known
```

这时，程序员有两种选择，要么改正方法的名字，要么声明一个名叫 computeAverage 的新方法。如果选择后者，程序员必须说明新方法的参数。当声明新方法的时候，参数类型必须提供，因为拥有在线接口检查的主要原因是能够检查完整的接口信息，而不仅仅是方法名。一个常见的错误是方法 p 使用 4 个参数调用方法 q，而方法 q 按规定需要 5 个参数。当调用正确地使用了 4 个参数，但其中 2 个参数位置错位时，错误会更难发现。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);
```

前两个参数在调用语句中错误地更换了位置。Java 编译器和链接器只有在以后调用它们的时候才会探测到这个错误。相反，一个在线接口检查器可以立即探测到这个错误或类似的错误。另外，如果编辑器有帮助工具，在尝试编写对 q 的调用前，程序员能够请求在线信息获取方法 q 的精确参数。更好的是，编辑器应该产生一个调用的模板，显示每个参数的类型。程序员仅仅需要把每个形参用正确类型的实参替代。

在线接口检查的一个主要优点是，可以立即标志出由于调用方法时错误的参数数目或者错误的参数类型而导致的难以检查出的错误。在线接口信息对高质量软件的高效生产是重要的，尤其是当软件是由团队生产时（多人编程）。关于所有代码制品的在线接口信息在任何时候对所有编程团队成员都可得是十分有用的。进一步，如果一个程序员改变了 vaporCheck 方法的接口，也许是由 **int** 到 **float** 改变了一个参数的类型，或者是增加了一个参数，那么每一个调用 vaporCheck 的组件必须自动成为不可用的，直到相关反映事务新状态的调用语句被改变为止。

即使有一个*语法制导的编辑器*（syntax-directed editor）被集成到在线接口检查器，程序员仍然需要先退出编辑器，然后调用编译器和链接器。显然，这可能没有编译错误，但仍必须调用编译器以生成执行代码。然后调用链接器。接下来，由于在线接口检查器的存在，程序员可以肯定所有的外部引用都可以满足，但是链接器仍然需要链接到产品。对这个问题的解决方法是集成一个*操作系统前端*（operating system front end）到编辑器。也就是说，程序员应该能在编辑器的内部给出操作系统命令。为了引起编辑器调用编译器、链接器、加载器和其他引起代码制品执行的系统软件，程序员应该能够输入一句简单的命令，名为 go 或者 run，或者使用鼠标来选择合适的标签或者菜单选择。在 UNIX 中，这可以通过使用*make* 命令（5.9 节）或者通过调用一个命令解释程序的脚本来实现［Sobell，1995］。这样的前端也能用于其他的操作系统。

最令人感到有挫败感的编程经验是，一个产品执行大约 1 秒，然后突然中断，打印出如下信息：

```
Overflow at 506
```

程序员工作在高级语言（如 Java 或 C++）环境，而不是低级语言（如汇编语言或机器代码）环境。但是当调试报告是 Overflow at 506 之类时，程序员不得不检查机器码核心转储、汇编程序列表、链接器列表和各种类似的低级文档，从而破坏了用高级语言编程的整个优点。

当出现

Core dumped

或者

Segmentation fault

这样的 UNIX 信息时也会有同样的问题，用户也需要进行低级语言检查。

在出现错误时，如图 5-10 所示的消息是对前面的简要错误消息的重大改进。程序员立即能够看到方法失败是因为尝试被 0 除。甚至更有用的是让操作系统进入编辑模式并且自动显示发现的错误代码行，即第 6 行，同时显示前面或者接下来的 4、5 行。然后程序员可能看到是什么引起这些错误，然后做出必要的修改。

```
OVERFLOW ERROR
 Class: cyclotronEnergy
Method: performComputation
Line 6: newValue=(oldValue+tempValue)/tempValue;
          oldValue=3.9583     tempValue=0.0000
```

图 5-10　源代码级调试器的输出

另一种源代码级调试是跟踪。在 CASE 工具出现之前，程序员不得不在执行时手工插入合适的打印语句到代码之中，以显示行数和相关变量的值。现在这可以通过能自动产生跟踪输出的*源代码级调试器*（source-level debugger）来完成。*交互式源代码级调试器*（interactive source-level debugger）则更好。假设变量 escapeVelocity 的值看起来不正确，并且方法 computeTrajectory 看起来也有错。这时通过使用交互式源代码级调试器，程序员能在代码中设置断点。当断点到达时，执行被暂停，进入到调试模式。程序员现在可要求调试器跟踪变量 escapeVelocity 和方法 computeTrajectory。也就是说，接下来每一次使用或修改 escapeVelocity 的值时，执行就会又一次终止。这样程序员就有输入进一步调试命令的选择权利，例如，要求显示特定变量的值。程序员可以选择在调试模式下继续执行，或者返回到正常的执行模式。不论何时进入或者退出方法 computeTrajectory，程序员同样能与调试器交互。当软件产品出现故障时，一个交互式源代码级调试器可以向程序员提供几乎任何可以想像得到的帮助类型。UNIX 调试器 **dbx** 就是这样的一个 CASE 工具。

前面多次提到，在线得到各种文档是十分必要的。程序员需要在编辑器内得到他们需要的所有文档。

正如前面介绍过的，具有在线接口检查能力的结构化编辑器、操作系统前端、源代码级调试器和在线文档，一起构成了完备而高效的编程工作平台。

这种类型的工作平台没有什么新意，所有这些特点自从 1980 年就被 FLOW 软件开发工作平台所支持［Dooley and Schah，1985］。因此，所提出的最小但基本的编程工作平台并未需要多年的研究，就产生了一个试验性的原型。与此相反，必要技术已经存在了 20 多年，有些程序员仍然按照旧的方式编码，而不使用像 Sun Java Studio（能够免费下载）或者使用开源的开发环境（如 Eclipse），这让人有些惊讶。

版本控制是一个必要的基本工具，尤其在基于团队进行软件开发时。

5.7　软件版本

无论何时维护一个产品，都至少有两个版本的产品：旧版本和新版本。因为产品由代码制品构成，那么每一个被修改过的组件制品中也会有两个或者更多的版本。

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早期阶段。

5.7.1 修订版

假设在几个不同的地点安装了产品。如果在一个制品中找到一个错误，那么需要修复那个制品。修改实施后，将会有两个版本的制品，旧版本和用来代替它的新版本。新版本定义为修订版（revision）。多个版本的存在很容易解决——任何旧版本都应该扔掉，只留下正确的。但是这样做是最不明智的。假设这个制品的先前版本是修订版 n，而新版本是修订版 $n+1$. 首先，无法保证修订版 $n+1$ 比修订版 n 更加正确。即使修订版 $n+1$ 可能已经由软件质量保证小组完全地测试过了，包括隔离测试和链接到产品的其他部分，当新版本的产品由使用者用实际数据来运行时，还是可能会有灾难性的后果。其次，必须保存修订版 n，因为产品可能分发到许多地点，而它们中不是所有的都能安装修订版 $n+1$. 如果在一个仍然使用修订版 n 的地点收到一个新错误报告，那么要分析这个新的错误，就有必要调整产品的配置，使其与实际使用场所相符，也就是说，使用制品的修订版 n。因此有必要保留每个制品的每一个修订版的拷贝。

正如在1.3节中所介绍的，完美的维护扩展了产品的功能。在一些情况下写出新的制品，在另外一些情况下，修改现有制品，从而加入这个新的功能，这些版本也是现有制品的修订版。因此，当执行适应性维护时制品也会被改变，也就是说，产品工作环境的改变也会引起对产品的改变。对于纠错性维护，所有先前的版本必须保留，因为问题不仅会产生于交付后维护时，还会产生于前面的实现阶段。毕竟，一旦一个制品编写完成，因为错误被发现和改正，它将持续地改变。因此，每一个制品都有很多版本，一定要对这些版本作出某些控制，以保证开发团队的每个成员知道那个版本是给定制品的当前版本。在给出该问题的解决方案前，必须要考虑周到些。

5.7.2 变体

考虑下面的例子。大部分计算机支持多种打印机。例如，一台计算机可能支持喷墨打印机和激光打印机。因此，操作系统必须包括两个打印驱动的变体（variation），每个对应一种打印机。每一个修订版都是来代替前面的版本，与修订版不一样，变体是用来共存。需要变体的另一种情况是产品被接入到许多种不同的操作系统和硬件中。许多制品的一个不同变体可能是针对每一操作系统和硬件组合而产生的。

图5-11简略描述了版本，它同时表示了修订版和变体。进一步考虑更复杂的情况，每个变体可能存在多个修订版，所以对于一个软件组织，为了避免陷入多个版本的泥潭，CASE工具是必需的。

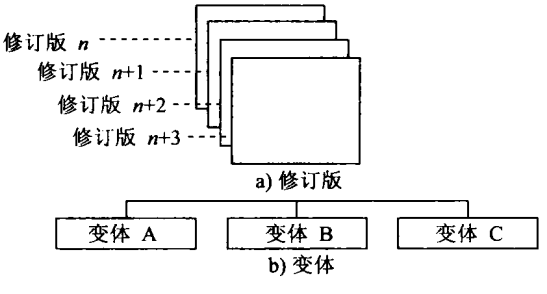


图5-11 制品多个版本的示意图

5.8 配置控制

每个制品的代码以三种形式存在。第一种是源代码，如今基本上都是用高级语言（如C++和Java）编写的。接下来是目标代码，由编译源代码得到。本书为避免单词“object”造成的混乱，把目标代码视为编译后代码。最终，每个制品的编译后代码与运行时例程结合，产生出一个可执行的加载镜像，如图5-12所示。程序员能够使用各个制品的各种不同版本。从各个制品的特定版本建立给定版本的完整产品叫做这个产品该版本的配置（configuration）。

假设SQA小组交给程序员一个测试报告，报告说一个制品在特定数据的测试下失败。首先

要做的一件事情是尝试问题再现。但是程序员怎样才能决定哪个变体的哪个修订版是崩溃的版本呢？除非使用一个配置控制工具（接下来将会讨论），指出错误原因的唯一方法是以八进制或者十六进制的形式查看可执行的加载镜像，并且将它与编译后代码（也是以八进制或者十六进制的形式）比较。具体而言，各种版本的源代码必须编译并和可执行加载镜像的编译后代码相比较。尽管这种办法能行，但是费时太长，尤其是当产品有几十个代码制品，而每个又有多个版本时。因此，当处理多版本时，有两个问题必须解决。首先，必须区分出不同的版本，以便每个代码制品的正确版本被编译和链接到产品。其次，有一个反问题：给定一个可执行加载镜像，决定要加载每一个组件的哪个版本。

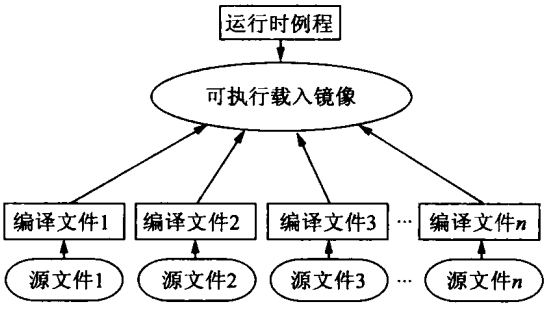


图 5-12　可执行载入镜像的组件

解决这个问题首先需要版本控制工具。许多操作系统（尤其是针对大型机的）支持版本控制。但也有许多不支持，在这种情况下就需要一个单独的版本控制工具了。版本控制中使用的一个通用技术是每个文件的名字都是由两部分组成：文件名本身和修订版号码。例如，一个制品有修订版 acknowledge/1、acknowledge/2 等，就像在图 5-13a 中描述的那样。程序员随后就能够明确对于给定的任务到底需要哪个修订版。

关于多个变体（不同的场合执行同样的任务但功能略微改变的版本），一种标志方法是用基本的文件名，后面括号里接变体名字［Babich，1986］。例如，两个打印机驱动命名为 printDriver(inkJet)和 printDriver(laser)。

当然，每个变体将会出现多个修订版，例如，printerDriver(laser)/12、printerDriver(laser)/13、printDriver(laser)/14，如图 5-13b。

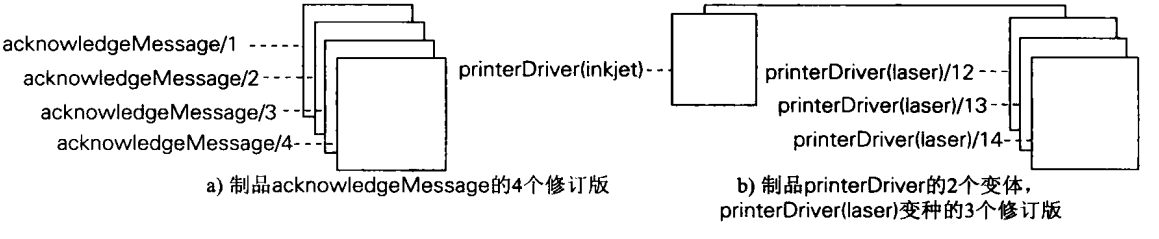


图 5-13　多重修订版和变种

版本控制工具是能够管理多个版本的第一步。一旦它就位了，产品的每个版本的详细记录（或者出处（derivation））必须保留。出处包含每个源代码元素的名字——包括修订版和变体；使用的各种编译器和连接器的版本；构建产品的人的名字；当然，还有它被构建的日期和时间。

版本控制对于将制品的多个版本和产品作为一个整体来管理有很大的帮助。但是由于与维护多个变体相关的其他问题，需要的不仅仅是版本控制。

考虑两个变体 printerDriver(inkJet)和 printerDriver(laser)。假设在 printDriver(inkJet)中找到一个错误，并且假设错误发生在两个变体都使用的制品中。那么必须同时修正 printerDriver(inkJet)和 printerDriver(laser)。一般地，如果有一个制品的 v 个变体，那么它们所有都必须修正。不仅如此，它们还必须以完全相同的方式修正。

这个问题的一个解决方式是只存储一个变体，如 printerDriver(inkJet)。任何其他的变体都根据从最初版本到那个变体所做改变的存储列表生成。这个差异列表称为增量（delta）。存储的

是一个变体和 $v-1$ 个增量。访问 printerDriver(inkJet)并应用增量就可以恢复 printerDriver(laser)变体，通过改变适当的增量就可以改变 printerDriver(laser)。然而，任何对 printerDriver(inkJet)做的改变，都会自动地应用到所有其他的变体中。

一个配置控制工具(configuration-control tool)能够自动地管理多个变体，但是配置控制(configuration control)的作用不局限于多个变体。控制配置工具也能处理团队开发和维护时遇到的问题，将在5.8.1节中详细描述。

5.8.1 交付后维护期间的配置控制

当一个以上的程序员同时维护一个产品时，会出现各种各样的问题。例如，假设两个程序员都在星期一早上分别处理一份不同的错误报告，恰巧两人都将他们将要修复的错误定位在同一个制品 mDual 的不同部分。每个程序员都制作制品的当前版本的一个拷贝 mDual/16，并且开始处理这些错误。第一个程序员修复第一个错误，修改得到了批准并替换当前这个制品，现在叫做 mDual /17。一天后第二个程序员修复了第二个错误，修改也得到了批准，并得到了制品 mDual /18。遗憾的是，17 版只包含了第一个程序员做出的修改，而 18 版只包含了第二个程序员做出的修改。mDual /18 中没有第一个程序员的修改，因为第二个程序员是针对 mDual/16 进行的修改而非针对 mDual /17。

尽管每个程序员生成制品的单独副本的想法远胜于一起处理软件的同一部分，但对于团队维护来说，这显然是不合适的，团队需要的是一种一次只允许一个使用者改变制品的机制。

5.8.2 基线

维护管理者必须设置一条基线，即产品中所有制品的配置（版本集）。当尝试找到错误时，维护程序员将任何需要的制品的拷贝放到其个人工作空间（private workspace）。在这样的一个私人工作区间中，程序员可以任意修改，而不会对其他程序员产生任何影响，因为所有的改变是针对程序员的个人拷贝，基线是没有改变的。

一旦决定了修改哪个制品以修正错误，程序员便冻结（freez）其将要改变的制品的当前版本。其他的程序员不能对任何冻结的版本作出修改。在维护程序员作出修改并且修改得到测试后，安装制品的新版本，从而修改基线。由于以后有可能还会需要，因此先前被冻结的版本仍然需要保留，但不可对其进行修改，其原因在前面已经解释过了。一旦安装了新版本，任何其他的维护程序员都能够冻结新版本，并作出改变。结果这个制品便成为下一个基线。如果两个或多个制品必须同时修改，那么将会接着进行一个相似的过程。

这种方法解决了制品 mDual 的问题。两个程序员都制作 mDual/16 的个人副本，并且使用这些副本来分析那些他们需要修改的错误。第一个程序员确定了要做什么修改，便冻结 mDual/16，然后修正第一个错误。在修改测试通过后，产生的版本 mDual/17 变为基线版本。同时，第二个程序员通过用 mDual/16 的个人副本测试，找到了第二个错误。然而，mDual/16 现在已经不能修改，因为它被第一个程序员冻结了。一旦 mDual/17 成为基线，它将被第二程序员冻结，他的改变是针对 mDual/17。产生的制品 mDual/18 现在作为产品安装，一个集成两个程序员的修改的版本就产生了。保留修订版 mDual/16 和 mDual/17 作为将来可能需要的资料，但是不能修改它们，因为 mDual/18 已成为基线。

5.8.3 产品开发过程中的配置控制

当一个制品正处于编码的过程中，版本的改变是非常快速的，以至于配置工具不能有效发挥作用。一旦一个制品编码完成，它应该立即由其程序员来进行非正规测试，参见6.6节中的介绍。

在非正式测试过程中，制品又一次经过许多版本。当程序员满意的时候，把制品递交给 SQA 小组来进行有条理的测试。一旦制品经过了 SQA 小组，就准备把它集成到产品中了。从那时开始，像交付后维护一样，它应该从属于同样的配置控制过程。对一个集成制品的任何改变都能影响产品整体，就像在交付后维护过程中的改变一样。因此，配置控制不只在交付后维护中需要，在实现中也需要。进一步来说，管理者无法完全监视开发进程，除非每个制品尽可能合理地、快速地从属于配置控制（也就是说，在它通过 SQA 组后）。当恰当地应用配置控制时，管理者就能够知晓每个制品的状态，如果看出项目要超出工程结束期限，就能够提早采取补救措施。

PVCS 是一个流行的商用配置工具。Microsoft SourceSafe 是一个用于个人电脑的配置工具。三个主要的 UNIX 版本控制工具是 sccs（源代码控制系统）［Rochkind，1975］、rcs（修订版控制系统）［Tichy，1985］和 cvs（并行版本系统）［Loukides and Oram，1997］。cvs 是一个开源的配置管理工具（开源工具在 1.10 节中有过介绍）。

5.9　建造工具

如果一个软件组织不打算购买全套的配置控制工具，那么至少要同时使用一个版本控制工具和一个*建造工具*（build tool），后者可帮助选择每个要链接的编译后代码制品的正确版本，从而形成产品的一个特定版本。在任何时候，每个制品的多个修订版和变种都在产品库里。所有的版本控制工具帮助使用者区分源代码制品的不同版本。但跟踪编译后代码更加困难，因为一些版本控制工具不能在编译后的版本后附加修订号。

为了处理这个问题，一些组织每晚自动编译每个制品的最新版本，从而保证所有的编译后代码是最新的。尽管这种方法可行，但是它极其浪费计算机资源，因为经常会有不必要的编译执行。UNIX 工具 make 可以解决这个问题［Feldman，1979］。对于每个可执行加载镜像，程序员设置一个 Makefile 文件来说明源文件在编译时采用何种特定的配置层次，这样一个层次如图 5-12 所示。例如，C++ 这种包含更加复杂的依赖关系的文件，也可以通过 make 来处理。当一个程序员调用它时，工具按如下方式运行：像其他虚拟操作系统一样，UNIX 将日期和时间戳关联到每个文件。假设源文件上的时间戳是星期五，6 月 6 号，上午 11: 24，而其编译后文件上的时间戳是星期五，6 月 6 号，上午 11: 40。那么很明显，编译器编译文件后并没有修改源文件。另一方面，如果源文件上的日期和时间戳比在编译后文件上的要迟，那么 make 调用了合适的编译器或者汇编程序来产生与源文件版本对应的编译后文件的一个版本。

接下来，可执行加载镜像上的日期和时间戳与配置中的每一个编译后文件相比较。假如可执行加载镜像比所有的编译后文件产生得更迟，那么没有再链接的必要。但是假如一个编译后文件有比加载镜像更迟的时间戳，那么加载镜像没有集成编译后文件的最新版本。在这种情况下，make 调用链接器，生成一个更新的加载镜像。

换句话说，make 检验是否加载镜像体现每个制品的当前版本。如果是，那么无需再做什么，并且不会有 CPU 时间浪费在不需要的编译和链接上。如果不是，那么 make 调用相关的系统软件产生产品的最新版本。

另外，make 简化了构建编译后文件的任务。使用者不需要在每次都说明什么制品将会被使用，以及它们怎样链接起来，因为这个信息已经在 **Makefile** 里面了。因此，一个简单的 make 命令就可以用几百个制品来构建产品，并确保完成的产品是正确地组合在一起的。

像 make 这样的工具被集成到了许多种编程环境中，包括 JBuilder 和 Visual C++。make 的开源版本是 Ant（Apache 项目的一个产品）。

5.10 使用CASE技术提高生产力

Reifer（在［Myers，1992］中报导过）组织了一项调查——引入CASE技术后对生产率产生的影响。他收集了10个行业45家公司的数据。一半的公司从事信息系统领域，25%从事科学领域，25%从事实时航天领域。平均年度生产率的提高各不相同，由9%（实时航天）到12%（信息系统）不等。如果仅考虑生产率的提高，那么这些数字不能证明使用CASE技术的每个用户花费的125 000美元的成本是值得的。然而，参与调查的公司均认为CASE的意义不仅在于提高生产率，还缩短了开发周期并且改进了软件质量。换句话说，尽管不如一些CASE技术的拥护者宣称的那么大，CASE环境的引入确实提高了生产率。此外，引入CASE技术到软件组织还有着其他同样重要的原因，例如，更快的开发、更少的错误、更好的可用性、更简单的维护以及更易于提高士气。

从15个500强公司的100多个开发项目中，关于CASE技术有效性的最新结果反映出培训和软件过程的重要性［Guinan，Cooprider，and Sawyer，1997］。当使用CASE的小组进行应用开发的培训和特定工具培训时，用户满意度增长了，开发计划也得以顺利完成。然而，当没有培训时，软件完成时间会推迟且用户也不太满意。同样，当小组使用CASE工具和结构化方法时，性能提高50个百分点。这些结果都表明了3.13节的断言，即CASE环境不应该由处于成熟级别1或2的团队使用。坦率地说，用工具的傻瓜仍然是傻瓜［Guinan，Cooprider，and Sawyer，1997］。本章最后的图5-14是本章描述的理论工具和CASE工具的字母顺序表，同时包括各工具出现的章节。

分析工具
- 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)
- 电子邮件(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)
- 灵巧打印器(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)
- 文字处理器(5.6节)
- 万维网浏览器(5.6节)

图5-14 理论工具和CASE工具的字母顺序表及出现章节

本章回顾

首先，本章介绍了几个分析工具。在5.1节描述了建立在米勒定律的基础上的逐步求精，并在5.1.1节通过一个例子进一步详细分析。5.2节介绍了另一个分析工具——成本-效益分析法。5.3节介绍了软件度量。

5.4节定义了计算机辅助软件工程（CASE），5.5节和5.6节分别介绍了它的分类和范围。接下来的几节介绍了几种CASE工具。当构建大型产品时，版本控制工具、配置控制工具和建造工具都很必要，它们在5.7～5.9节都作了介绍。生产率的提高，作为使用CASE技术的结果，在5.10节有所描述。

延伸阅读材料

为了进一步认识米勒定律和有关大脑如何按块工作，可以查阅［Tracz，1979］、［Moran，1981］和米勒的最初论文［Miller，1956］。

Wirth［1971］关于逐步求精方面的论文是经典之作，值得详细研究。关于逐步求精的书籍，Dijkstra［1976］和Wirth［1975］同样重要。实时系统的逐步设计在［Kurki-Suonio，1993］中有所介绍。

［Kitchenham，Pickard，and Pfleeger，1995］表述了工具评价的研究。［Sharma and Rai，2000］描述了CASE在软件工程中使用的范围。

本书中，软件过程的不同工作流用到的 CASE 工具在本章的每个工作流都有介绍。要想进一步了解关于工作平台或者 CASE 环境的信息，可查找第 13 章进一步阅读材料的相关部分。

［Whitgift，1991］详细介绍了配置管理。更新一些的文章参见［van der Hoek，Carzaniga，Heimbigner，and Wolf，2002；Mens，2002；and Walrad and Strom，2002］。《软件配置管理国际会议论文集》是一个有用的信息来源。

有关成本－效益分析有非常多的优秀书籍，包括［Gramlich，1997］。软件产品线（8.5.4 节）的成本－效益分析在［Bockle et al.，2004］中有所讨论。Van Solingen［2004］描述了软件过程改进的成本－效益分析。

度量方面的重要书籍有［Shepperd，1996］和［Fenton and Pfleeger，1997］。Jones［1994］强调了不可用和无效度量，不过仍然在文献中提到。面向对象度量的实用性在［El Emam，Benlarbi，Goel，and Rai，2001］和［Alshayeb and Li，2003］中有所讨论。1997 年 3/4 日期《IEEE Software》包括了度量方面的一系列论文，包括［Pfleeger，Jeffrey，Curtis，and Kitchenham，1997］，以及软件度量的估计。Kilpi［2001］描述了度量计划如何在 Nokia 中实现。基于 COTS 系统的度量在［Sedigh-Ali and Paul，2001］中有所表述。

第 7 届国际软件度量会议的一些文章发表在 2001 年 11 月的《IEEE Transactions on Software Engineering》；特别感兴趣的读者可以阅读［Briand and Wüst，2001］。

习题

5.1　现考虑将前瞻操作引入顺序主文件更新问题的第 3 次求精所带来的效果。也就是说，在执行一个事务前，必须读到下一个事务。如果两个事务应用于同一个主文件记录，那么当前事务的处理决定依赖于下一个事务的类型。画一个 3×3 的表格，行表示当前事务的类型，列表示下一个事务的类型，并且填入每种情况下将采取的行动。例如，两次连续对同一条记录的插入是一个错误。但是两个修改可能完全有效，例如，一个订阅者能在某月多次改变地址。现在开发一个考虑了前瞻的第三次求精的流程图。

5.2　检验你的习题 5.1 的答案是否能正确处理一个修改事务，然后是删除事务，这两个事务都应用到同一个主文件记录。如果不能，修改你的答案。

5.3　检验你的习题 5.1 的答案是否能正确处理一个插入，然后是修改和删除，所有事务都应用到同一个主文件记录。如果不能，修改你的答案。

5.4　检验你的习题 5.1 的答案是否能正确处理 n（$n>2$）次插入、修改或者删除，所有事务都应用到同一个主文件记录。如果不能，修改你的答案。

5.5　最后一条事务记录没有后继者。检验你的习题 5.1 的流程图是否将这个情况考虑进去，并且正确地处理最后一条事务记录。如果不能，修改你的答案。

5.6　在一些应用中，取代前瞻的一种方法是仔细地对事务进行排序。例如，对同一个主文件记录进行修改，然后进行删除，这样产生的问题能够通过在修改前执行删除来解决。这样可能会产生主文件正确写入而在异常报告里出现一个错误信息的结果。查找是否存在一个事务的顺序能解决习题 5.2～5.4 中列出的所有困难。

5.7　一种新型的肠胃疾病正肆虐 Concordia 地区。就像网状内皮细胞真菌病一样，它也是以空气中真菌的形式传播。尽管这种疾病几乎从不致命，但是会带来很大的痛苦，而且患者有两周不能工作。Concordia 的地方当局希望确定根治这种病需要多少钱。负责给公共卫生部建议的委员会正考虑四个方面的问题：健康护理成本（Concordia 给所有居民提供免费的健康护理）、收入损失（因此而减少的税收）、痛苦和不舒服，以及对政府的态度。请解释成本－效益分析法将如何帮助委员会。对于每项效益或者成本，给出如何估计收益或成本的美元值的建议。

5.8　单人的软件生产组织需要版本控制工具吗？如果需要，为什么？

5.9　单人的软件生产组织需要配置控制工具吗？如果需要，为什么？

5.10　你是一个负责控制小型潜水艇航行系统的软件的管理者。必须修正三个用户报告的错误，而你把它们分别分配给 Paul、Quentin 和 Rachel。一天以后，你知道为了修改这三个错误，必须修改同样的 4 个制品。然而，你的配置控制工具不能工作，因此，你不得不自己去管理这些修改。你将如何做？

5.11　图 5-14 列出的 CASE 工具中，哪些可以促进软件开发中的逐步求精？证明你的回答。

5.12 （学期项目）哪种类型的 CASE 工具将会适合开发附录 A 中介绍的 Osric 的办公用品和装饰产品？

5.13 （软件工程读物）教师分发［Wirth，1971］的复印件。列出 Wirth 的方法和本章所述的逐步求精方法的不同点。

参考文献

[Alshayeb and Li, 2003] M. ALSHAYEB AND W. LI, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes," *IEEE Transactions on Software Engineering* **29** (November 2003), pp. 1043–49.

[Babich, 1986] W. A. BABICH, *Software Configuration Management: Coordination for Team Productivity*, Addison-Wesley, Reading, MA, 1986.

[Bockle et al., 2004] G. BOCKLE, P. CLEMENTS, J. D. MCGREGOR, D. MUTHIG, AND K. SCHMID, "Calculating ROI for Software Product Lines," *IEEE Software* **21** (May/June 2004), pp. 23–31.

[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.

[van Solingen, 2004] R. VAN SOLINGEN, "Measuring the ROI of Software Process Improvement," *IEEE Software* **21** (May/June 2004), pp. 32–38.

[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

通过本章学习，读者应能：

- 描述质量保证问题。
- 描述如何进行产品的基于非执行的测试（审查）。
- 描述基于执行测试的基本原则。
- 能够解释在什么情况下需要测试。

传统的软件生命周期模型在集成之后、交付维护之前通常有一个独立的测试阶段。从获得高质量软件的角度来看，这一阶段十分重要。测试是软件过程中一个完整的组成部分，是应该贯穿整个软件生命周期的行为。例如，在需求工作流阶段应该检查需求；在分析工作流阶段，应该检查规格说明；而软件产品管理计划则应该进行类似的详细审查。设计工作流中需要小心翼翼地检验每一个阶段；在软件的实现工作流中当然需要测试每个代码制品；产品完全集成时，需要将产品作为一个整体进行测试。在通过验收测试后，安装产品，并开始交付后的维护。而联合维护也就变成了对产品修改版的迭代检查。

换句话说，仅仅在工作流结束后才测试该工作流的产品是远远不够的。例如，考虑设计工作流，开发小组的成员必须在开发的同时不断检验设计。若在开发完全结束后的几周或几个月后才发现之前的设计存在重大问题，则必然要重新设计整个产品。所以，开发小组必须在每个工作流中不断进行测试，而不仅仅是在每个工作流的结束阶段进行较为系统的测试。

我们在1.7节介绍了术语“验证”（verification）和“确认”（validation）。*验证*是确定一个工作流是否正确执行的过程，这个过程在每个工作流结束时发生。*确认*是在产品交付用户之前进行的深入细致的评估过程，其目的在于确定整个产品是否满足规格说明。尽管这两个术语在IEEE软件工程词汇表［IEEE 610.12，1990］中是这样定义的，而且普遍使用**V&V**来指代测试，但本书中将尽量避免使用*验证*和*确认*。一个原因是，就像在6.5节解释的那样，*验证*在软件的测试范围里有另外的含义；另一个原因是*验证*和*确认*（也可以称为V&V）暗示着可以等到某一个工作流完全结束后再对其进行检查。相反，这个检查与软件开发和维护活动并行是非常重要的。因此，为了避免曲解V&V的含义，本书只使用*测试*（testing）一词。使用*测试*一词的第二个原因在于，它是统一过程中的术语。例如，第5个核心工作流是*测试工作流*（test workflow）。

实质上，测试类型有两种：基于执行的测试和基于非执行的测试。例如，书面的规格说明文档是不能够执行的，只能对它进行尽可能细致的检查，或者对它进行某种形式的分析。然而一旦有可执行代码，就可以运行测试用例，也就是说，进行基于执行的测试。当然，有些情况即便有可执行代码也可以进行非执行测试，因为就像下面将要提到的，仔细审查代码会发现和运行测试用例时同样多的错误。本章将向读者介绍基于执行的测试和基于非执行的测试的主要原则，第10～14章将应用这些原则，届时将描述过程模型的每个工作流以及适用于这些工作流的特定测试惯例。在备忘录1.1中描述的前两个错误导致了严重的后果，幸运的是，在大多数情况下就算交付了含有错误的软件也不会引起太严重的后果，然而，不管如何强调测试工作的重要性都不为过。

6.1 质量问题

本节首先扩展1.10节中与测试有关的概念。软件系统中的*差错*（fault）是由开发人员的过

错（mistake）导致的［IEEE 610.12，1990］。软件人员的一个过错可能会导致软件中出现多个差错，当然，几个过错也可能造成同一个差错。故障（failure）是在软件运行时观察到的不正确的行为，它是差错的结果。错误（error）是不正确结果的累积［IEEE 610.12，1990］。某个故障可能是由数个差错引起的，而某些差错可能永远不会引起故障。缺陷（defect）是一个通用词汇，泛指差错、故障或者错误。

现在来介绍质量问题的相关知识。术语质量在软件工程的范畴里常常被人误解。毕竟，质量这个词暗示着某种完美的含义，但是这种含义通常不在软件工程的范畴里使用。坦率点说，多数软件开发组织只是让软件运行正确罢了，“完美”这个词所暗示的程度远远超过正常状态，位于CMM级别1的软件组织是难以望其项背的。

软件产品的质量是指软件产品符合其规格说明的程度（参考备忘录6.1）。然而，仅仅考虑这一点是不够的。例如，为了保证软件产品的可维护性，该产品必须经过小心的设计和仔细的编码。因此，对于软件产品来说，质量是必要的但不是充分的。

备忘录6.1

“质量”一词指“符合规格说明”（而不是“完美”或者“精美”）的程度，反映了工程和制造领域中的实际情况。以可口可乐公司为例，其灌装厂质量管理经理的职责主要是确保每个离开生产线的瓶子或罐子在任何方面都能满足可口可乐公司的规格说明，而并不是试图去制作“完美”或者“精美”的可口可乐，他们只关注是否每瓶（或每罐）可口可乐都严格地符合了公司对于这种碳酸饮料的质量标准（规格说明）。

“质量”一词在汽车工业中也同样适用。“质量第一”是福特汽车公司以前的一句口号。换句话说，福特公司的目标在于让每辆从他们生产线上制造出来的汽车都能严格符合该车的规格说明。按照通常软件工程的说法，该汽车必须在任何方面都是“免检的”。

专业软件工程人员的任务是随时保证高质量的软件产品，换句话说，每一个参与项目的开发人员和维护人员都应该对他们应完成的任务负有检查和核对的责任。软件的质量并非是软件质量保证（SQA）小组后来加入的东西，而应该在一开始就由开发人员建立。SQA小组的职责之一是确保每个开发人员确实在进行高质量的开发工作。SQA小组当然还有其他职责，6.1.1节中将对此进行说明。

6.1.1　软件质量保证

如前所述，SQA小组的职责之一是确保开发者的产品是正确的，简单来说，一旦开发者完成了一个工作流并且已经仔细检查了工作，SQA小组的成员就要确保这个工作流确实正确地完成了各项工作。此外，当整个产品完成并且开发人员认为该产品整体上正确时，SQA小组也需要再次检查整个产品以确定其正确性。然而，SQA小组的职责并不仅限于在每个工作流或者整个项目结束时进行测试（或V&V），SQA在软件开发的整个流程中都应该发挥作用。例如，SQA小组的职责还包括制定一系列软件产品必须遵守的规范及标准，并建立一套督察机制来保证产品确实符合这些标准。简而言之，SQA小组的职责就是通过保证软件开发过程的质量来确保软件产品的质量。

6.1.2　管理独立性

在开发小组和SQA小组之间保持管理独立性（managerial independence）是非常重要的。换句话说，开发小组和SQA小组应该隶属于不同的管理者，任意一方的管理者都不能支配另一方的管理者。原因在于，时常在软件产品即将发布时可能还存在着某些重大的缺陷。软件开发组织必须决定究竟是按时发布一个带有缺陷的软件，还是让开发人员修正缺陷延期发布产品。前

者可能意味着用户将会使用一个含有错误和漏洞的产品，而后者则意味着用户不能及时用上他们想要的产品。无论作出哪一个选择，软件产品的用户都会对产品的开发组织失去信心。负责软件开发工作的管理者不应该作出按时发布有漏洞的产品的决定，而负责 SQA 的管理者不应该作出采取更多测试而延期发布的决定。事实上，两位管理者都应该向更高级领导汇报，由他来决定采用哪一种策略对客户和软件开发组织都最好。

表面上看起来，设置一个管理独立的 SQA 小组会增加软件开发的成本，然而事实并非如此。额外的成本与它带来的收益——高质量的产品相比是微不足道的。如果没有 SQA 小组，软件开发组织中的每个开发人员就都必须或多或少地参与到质量保证工作中来。假设一个软件开发组织有 100 位专业的软件开发人员并且他们每人投入 30% 的时间和精力用于软件质量保证工作，而一种更好的人员配置将是把这 100 名开发人员划分为两个小组，70 人从事开发工作而 30 人专注于提高软件质量。这样，投入到 SQA 工作中的总时间保持不变，唯一需要增加的成本就是必须为那 30 位专职测试人员再雇用一名管理者。这样一来，质量保证的工作就可以由一个独立的小组来完成，而这种配置相较于上文那种人人参与 SQA 的情况来说，将产生更高质量的软件产品。

在软件公司非常小的情况下（例如，雇员只有 4 个或者更少），成立一个独立的 SQA 小组可能不那么经济。在这种情况下，最好的做法就是确保分析产品由不负责生产这些产品的某个人来进行检查，对于设计产品、编码产品等也要进行类似处理。6.2 节将给出这样做的原因。

6.2 基于非执行的测试

测试软件而不运行测试用例称为*基于非执行的测试*（non-execution-based testing）。基于非执行的测试方法的例子包括评审软件（仔细地读软件代码）和用数学方法分析软件（见 6.5 节）。

让撰写文档的人员自行检查自己的文档不是一种好的做法。几乎所有人对于文档中的错误都有其自身的盲点，而如果撰写文档和检查文档的人的这种盲点相同，很多错误可能就发现不了。因此，检查文档的工作应该由某位不是文档原作者的人来完成。另外，只有一位检查人员可能还不够。大家可能都有这样的经历：多次阅读一篇文档却不能发现文章里面明显的拼写错误，而换一个人则能够立刻挑出这样的错误。这就牵涉到走查或审查这样的评审技术中的一个重要原则。在这两种类型的评审中，一个文档（如一个规格说明文档或设计文档）应该由一组具有不同技能背景的软件专业人员来共同检验，因为这些专家具有的不同知识背景，这将会极大地增加发现错误的概率。此外，一组有经验的人在一起工作通常会产生一种互相促进的效果。

走查和审查是两种类型的评审。两者之间的主要区别在于，走查相比审查来说步骤较少，而且也不那么正式。

6.2.1 走查

一个走查（walkthrough）小组应由 4 ~ 6 名开发人员组成。一个分析走查小组应该至少包含以下人员：一位负责撰写规格说明的小组代表、一位负责分析工作流的管理者、一位客户代表、一位即将进行下一个开发流程的小组的代表（在这个例子中是指设计小组）以及一位 SQA 小组的代表。SQA 小组的代表应该负责整个走查流程，6.2.2 节将对此加以解释。

走查小组的成员应该尽可能是高级软件技术人员，因为这些有经验的技术人员能够发现更多的严重错误。也就是说，他们能够首先找出那些对于软件产品有重大负面影响的错误[R. New，personal communication，1992]。

走查所需的相关材料应该尽可能提前发给各参与者，以便于每个人都能有充足的准备时间。每个评审者都应该仔细阅读材料，并且制作两个清单：一份是评审者不明白的事项列表，另一份是这位审查者认为有错误的地方的清单。

6.2.2　管理走查

走查应由 SQA 小组的代表负责，因为如果走查没有进行好，SQA 小组损失最大。相对而言，负责撰写分析工作流的代表可能非常希望规格说明文档尽快得到批准以开始其他的工作流；客户代表可能会认为走查没有找出的错误将会在验收测试中显现出来进而得到修复，修正它们也不需要客户额外的投入。然而，SQA 小组的成员在这里承担着最大的风险：产品的质量直接反映 SQA 小组的专业能力。

走查的组织者负责引导走查小组的其他成员通读分析文档来找出错误。改正错误不是走查小组的任务，他们只是把错误记录下来以备修改。这样做的原因主要有 4 点：

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）并不是所有标记为错误的地方都有真正的错误。依照格言："没断，就不要修复"。因此，最好对那些标示出的错误进行系统的分析，并且只在确定它确实有错时才去修正它；而不是组织一个小组去修正那些根本就是正确的东西。

4）走查没有足够的时间来检测和修正错误。一般走查不应该超过 2 小时，而这些时间应该用来找出并记录那些错误，而不是修正它们。

实施走查的方法有两种。第一种方法是参与者驱动的方法。在这种方法中，参与者列出不清楚的事项和认为有错误的地方，而撰写分析文档小组的代表必须解释参与者提到的每个问题，说明评审者看不懂的地方，或者认同那确实是一个错误，或者指出评审者为何错了。

第二种方法是文档驱动的方法。负责该文档的个人或小组的一部分成员带领参与者通读文档，而所有评审者就事先准备的意见或现场引发的意见，随时打断通读并提出问题。这种方法看起来更加彻底一些，并且会发现更多的问题，因为文档驱动走查中的大多数错误是由介绍者自发发现的。在会议中，介绍者的通读会一次又一次被打断，介绍者会脸红，多次阅读也未被发现的潜伏错误会突然变得明显。心理学家们研究的一个卓有成效的领域就是，确定为什么在各种走查（包括需求走查、分析走查、设计走查、规划走查和代码走查）期间，言语表达常常会导致错误的检出。毫不奇怪，更加彻底的文档驱动检查方法是 IEEE 软件审查标准［IEEE 1028，1997］中规定的技术。

走查工作负责人的主要任务是引出问题和鼓励讨论。走查是一个互动的过程，而不是负责人的一面之词。同时，走查也不能作为对于参加者的评估方法。如果是那样的话，走查就变成了一次给参与者打分的过程；无论负责人多么努力地鼓励讨论，这样的会议也不能发现任何错误。曾经有人建议负责被审读文档的管理者应该是走查小组成员，然而如果这名管理者同时也负责走查小组（特别是介绍者）的年终审核的话，走查小组的查错能力将会大打折扣，因为介绍者的主要动机将是使暴露的错误减到最少。为了防止这种利益冲突的出现，负责某个工作流的人员不应该出现在负责管理评价这个工作流的走查小组成员中。

6.2.3　审查

审查这种方法首先由 Fagan［1976］提出的，主要用于测试设计和代码。审查（inspection）比走查更加深入，它有 5 个正式的步骤：

1）由负责生成文档的人提供待审查的文档（如需求、规格说明、设计、代码或者规划）的概要（overview）。在概要部分结束时，将文档分发放参加者。

2）在准备阶段，每位参与者都试图去详细理解发放的文档。这时，在最近的审查中发现的错误类型（按照出现频率排列）的列表对参与者是最有帮助的。这些列表能够帮助与会者关注

那些最容易出现错误的部分。

3）当审查工作开始时，首先由一位参与者与审查小组的所有人一起通读整个文档，并且保证涉及了文档中的每个细节和分支。然后开始查找错误。与走查一样，审查的主要目的也仅仅是查找错误，而并非修正它们。在一天以内，审查工作的组织者（主持者）必须提交一份书面的会议报告，以确保审查小心细致地完成。

4）在修订（rework）阶段，负责文档的人员修正所有审查报告中提到的问题和错误。

5）在跟踪（follow-up）阶段，主持者必须保证所有提到的问题都已经很好地解决了，或者修正原先的文档，或者进一步澄清被误当成错误的事项。所有改动过的内容都应该再次检查，避免引入新的错误［Fagan，1986］。如果送审材料中5%以上的篇幅重新修改了，那么就必须要重新召集审查小组进行审查。

审查工作会议应该由一个4人小组召开。例如，在设计审查会议中，工作小组包括一位主持者、一位设计者、一位实现者和一位测试人员。主持者在整个审查工作中负责领导和组织工作。参加工作的人员中必须包含当前工作流小组的代表，以及将会负责下一流程小组的代表。上面提到的设计者对应于生成设计的小组成员代表，而实现者对应于将设计转换为代码的小组成员代表。Fagan建议测试者就是那些负责编写测试用例的任意一个程序员，当然该测试者最好是SQA小组成员。IEEE标准建议3～6人组成审查小组［IEEE 1028，1997］。其中，主持者要担当一些特殊的角色，例如，既作为*朗读者*带领大家通读文档，又作为*记录者*负责撰写关于所发现的错误的书面报告。

审查工作的一个重要的组成部分是一份潜在错误的清单。例如，设计审查的检查清单应该包括以下内容：是否已经清晰和充分地说明了设计文档中的每项内容？对于每个接口来说，是否实参和形参均能够匹配？是否错误处理机制都已经设计完成？该设计是否与硬件资源兼容？或者它需要更加先进的硬件？这个设计是否能在现有的软件环境下运行？例如，是否设计阶段指定的那个操作系统有足够的功能来支持这样一个软件设计？

审查工作的另一个重要成果是错误统计记录。在记录时，应该将错误按照严重程度（主要错误或次要错误；例如，导致程序过早中止或者损害数据库的错误是主要错误）和错误类型进行分类。在设计审查中，诸如接口错误或者逻辑错误都是典型的错误类型。这些信息可以用在许多方面：

- 发现的错误的个数可以与同期其他可比软件开发项目的错误平均数进行比较。这样的比较可以较早地提醒管理人员软件开发过程中存在的某些问题，从而使得整个开发小组有较充裕的时间来进行修正工作。
- 如果检查了几个代码片段后发现某种类型的错误数量特别多，管理人员就可以针对其他代码片段展开类似的检查和修正工作。
- 如果对于某个代码片段的检查发现了它比其他部分多很多的错误，通常就说明必须对这个部分的代码彻底进行重新设计，并实现新的设计。
- 对于早期设计文档进行的审查所得出的错误统计，可以帮助整个小组在后期进行产品实现的代码审查工作。

Fagan［1976］的第一个实验是针对一个系统软件产品进行的。在这个实验中，审查共投入了大约100人时，平均大约是一个4人小组每天进行2小时的审查工作。在所有于开发过程期间发现的问题中，有67%是在单元测试开始之前的审查工作中找到的。此外，在产品发布的最初7个月内，在审查过的产品中发现的错误比采用非正式的走查方法检查过的产品中发现的错误少了38%。

Fagan［1976］对于另一个应用软件项目进行的实验显示，82%的错误是在对设计和代码实现进行审查的过程中发现的。审查工作的一个明显的好处就是可以缩短开发时间，因为审查后的项目需要的单元测试时间更少。Fagan通过一个自动评估模型，得出结论如下：不考虑审查所花费的时间，审查工作使程序员资源节省了25%。在另一个不同的实验中，Jones［1978］发现超过70%的错误能够通过设计和代码审查检测出来。

随后的实验也得出了同样令人印象深刻的结果。在一个含有6 000行代码的商务数据处理程序中，大约93%的错误是在审查阶段发现的［Fagan，1986］。在［Ackerman，Buchwald，and Lewski，1989］中还公布了这样一个结论：使用审查方法而不是测试方法来开发一个操作系统，检测错误的成本在这个项目中下降了85%；而在另一个交换机系统中下降了90%［Folwer，1986］。在美国喷气推进实验室（JPL）的一个实验中，平均每2小时的审查工作能够发现4个主要错误和14个次要错误［Bush，1990］。如果把这种节省换算成美元的话，大约每次审查工作就可以节省25 000美元。另外一个JPL的研究显示［Kelly，Sherif，and Hops，1992］，检测到的错误数量随着传统的软件开发阶段的推进成指数减小。换句话说，借助审查，能够在软件开发的较早阶段发现错误。图1-5反映了在较早的开发阶段发现错误的重要性。

代码审查相对于基于执行的测试的另一个优势在于测试者不必处理故障。当进行基于执行的测试时，如果发生了一个运行故障，就必须开始查找导致这个故障的缺陷并改正，然后测试才能继续。而在基于非执行的测试中只需记录发现的代码错误，然后可以继续审查。

进行审查工作的风险与走查相同，即它有可能会用作对员工能力的评估。这个风险在审查工作中更明显一些，因为审查工作会产生详细的错误信息。Fagan消除了这种恐惧，他指出，在3年内并没有一位IBM公司的管理者使用这些信息来评估程序员；或者如他所说，并没有管理者试图“杀死一只能下金蛋的鹅”［Fagan，1976］。然而，如果审查工作没有正确展开，这种方法可能就不会像它在IBM一样取得广泛成功。除非高层管理者意识到滥用审查工作信息所造成的潜在的问题，不然就很可能出现审查信息的滥用。

6.2.4　走查和审查的对比

表面上看，走查和审查工作的区别在于，审查小组会使用一个问题列表来帮助发现问题。其实区别远不止如此。走查的过程有两步：准备，以及随后整个小组对文档进行分析。而审查的过程有5步：概要、准备、审查、修订和跟踪；而且对于每一个步骤都有形式化的、严格的规定。例如，在审查中，将发现的错误系统地归类，并且将其用于后续的工作流和未来的产品。

审查要比走查花更多的时间，那么值得在审查上花费额外的时间和努力吗？6.2.3节的数据清楚地表明审查是一种强有力的、划算的查错工具。

6.2.5　评审的优缺点

评审（review）是上面提到的走查和审查方法的总称，并且它有两个明显的优点。首先，评审是找出错误的一个有效途径；其次，在软件开发的早期发现错误，就避免了在后期修正错误将要花费的昂贵成本。例如，在实现开始之前发现设计错误或者在产品集成之前发现编程错误。

然而，如果软件开发不是按照规范的流程进行的话，评审方法的有效性就会下降。首先，大型软件必须要能够被分割成很多个相对独立的小部分，否则要对它进行评审就很困难。面向对象的方法的一个巨大优点就是它能够把软件分割为若干个相对独立的部分。其次，设计评审小组常常需要参考分析产品，代码评审小组常常需要参考设计文档。除非前面工作流的文档是完整而且更能反映当前情况的，否则评审工作就会严重受阻。

6.2.6　审查的度量方法

为确定审查的效果，可以采用一系列不同的度量方法。首先是*审查速率*（inspection rate）。在审查规格说明和设计文档时，可以记录下每小时审查的页数；当审查代码时，可以记录下来每小时审阅的行数。另一个标准是*差错密度*（fault density），这个指标是指检查每页或者每千行代码（KLOC）能够发现多少错误。这个标准也可以细化为主要错误密度和次要错误密度两种。另一个常用的标准是*差错检测率*（fault detection rate），即每小时的审查工作检测到的主要和次要错误数。第4个标准是*差错检测效率*（fault detection efficiency），即每个人每小时检测到的主

要和次要错误数。

虽然这些标准的目的是测量审查工作的效果，但结果却能够反映开发小组的不足。例如，如果错误检测效率突然由每千行20个错误上升到每千行30个错误，这可能并不意味着审查小组的工作效率突然提高了50%，而可能是因为代码的质量有所下降，才发现了更多的错误。

上面讨论了基于非执行的测试，接下来将讨论基于执行的测试。

6.3 基于执行的测试

上文提到，测试最多只能证明某些差错（bugs）不存在而已。尽管有些软件开发组织把软件开发预算的一半都投入测试，然而他们发布的“经过测试”的软件仍然相当的不稳定。

出现这种矛盾的原因很简单。就像Dijkstra所说的，“程序测试可能是显示bug存在的非常有效的方式，但说明它们的不存在却是绝对不充分的”［Dijkstra，1972］。Dijkstra的意思是，如果在某些测试数据下软件产品给出了错误的结果，那么该软件一定存在漏洞。但是如果输出正确，软件产品仍然有可能在其他方面存在漏洞。从一个特定的测试试验中能够得出的唯一结论就是，软件产品在该测试数据的输入下运行正确而已。

6.4 应该测试什么

在阐述需要测试哪些属性前，首先给基于执行的测试下个精确的定义。Goodenough［1979］指出，*基于执行的测试*是，基于或部分基于在已知环境下用经过选择的输入执行产品得到的结果推断某产品的特定行为特性的过程。这个定义有三个令人困扰的含义：

1）首先，这个定义把测试看成一种推断的过程。测试员用给定的数据去运行软件产品，并且检查软件输出。如果有错，测试员就必须推断该软件产品什么地方有问题。从这个观点来看，测试工作无疑就像是在黑屋子里找黑猫一般，而且还事先不知道黑屋子里究竟有没有黑猫。测试员几乎没有线索来找到错误：也许有10组或20组测试数据及其输出结果，或者一份用户的错误报告，再就是海量的代码，凭这些信息，测试者要从中找出软件中是否存在错误，如果有，还要弄清是什么错误。

2）这个定义在谈到“已知环境”时也存在问题。环境中的许多因素是不可确定的，软件环境和硬件环境都是如此。操作系统不一定总是能正确运行，而且运行时环境也经常出问题；计算机的内存也可能存在间歇性硬件故障。在这些环境下，某些软件产品表现出的行为可能并非由产品本身引起，而有可能是一个正确的产品和一个有问题的编译器、硬件或者其他什么东西相互作用的结果。

3）另一个有问题的地方是这个定义所提到的“用经过选择的输入”。如果是要开发一个实时系统的话，对于输入数据就几乎不能加什么限制。以航空电子设备软件为例，这种飞行控制系统通常有两类输入数据。第一类是飞行员想要飞机做的动作，例如，如果飞行员把操纵杆向后拉来让飞机爬升，或者打开节流阀让飞机加速的话，这类动作都会转化为数字信号送入飞行控制计算机。第二类数据是飞机的当前状态，例如，它的高度、速度和机翼仰角。飞行控制软件用这些数据来计算采取怎样的动作，例如，调整机翼仰角或者改变引擎功率才能实现飞行员的意图。虽然通过设置飞机的操纵系统，上述飞行员的输入指令可以采取任何想要的数据，但是有关飞机当前状态的数据就没有办法限定到一个给定的值。事实上，目前还没有办法能让一架飞行的飞机去提供“经过选择的输入”。

那么，要如何测试这些实时系统呢？答案是使用仿真器。*仿真器*（simulator）是产品即将投放的运行环境的一个工作模型。在上例中，飞行控制软件的测试工作可以由仿真器向飞行控制软件发送经过选择的输入信号来完成。仿真器拥有一套控制系统，能够让测试人员把被测试软件的输入调整到选择的任何值。如果测试工作的目标是测试飞行控制系统要如何处理飞机引

擎着火的状况，那么仿真器就可以向飞行控制系统输入引擎着火的仿真信号，而这种信号对于被测试的飞行控制系统来说，与真实的飞机着火信号是没有区别的。随后，测试人员将着手分析飞行控制系统给出的输出信号。最好的仿真器最多也就只能作为一个实际系统的运行环境的很好近似：它可以在许多方面表征环境的特性，却决不能完全仿真一个真实的环境状况。使用仿真器进行测试意味着虽然我们可以人工创造一个“已知环境”，然而这种已知环境永远不能准确地切合真实的软件运行环境。

前面关于软件测试的定义侧重于“行为特性”，然而哪些行为特性需要测试呢？显然应该测试软件的功能是否正确。但是，下面将要看到，正确性对于软件来说是既不充分也不必要的。在我们讨论正确性以前，先讨论另外4个行为特性：实用性、可靠性、健壮性和性能［Goodenough，1979］。

6.4.1　实用性

实用性（Utility）是在规格说明许可的范围内使用正确的产品时，用户需求满足的程度。换句话说，上文提到的软件正确性现在正被实用性所取代：实用性指出了软件在规格说明许可的输入下能否得到正确的输出。例如，用户可能会去尝试软件是否易于使用，或者软件是否提供实用的功能，以及软件相对于其他同类软件来说是否成本划算。不管产品是否正确，都必须测试这些重要问题。如果软件并非物有所值，就没人会购买它。而如果软件不方便使用，就不会有人使用它，或者不能正确地使用它。因此，在考虑要购买哪一种产品时，应该首先测试该产品的实用性；如果该软件连这一条标准都不能达到，后面的测试就没有必要进行了。

6.4.2　可靠性

软件产品必须测试的另一个方面是它的可靠性。可靠性（reliability）是对软件出现故障的频率和严重程度的度量。正如上文所说，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，有必要知道产品每隔多长时间出现故障（平均故障间隔时间）和故障造成了多么严重的后果。当一个产品出现故障时，一个重要问题是修复故障平均需要多少时间（平均修复时间）。但是，更加重要的是要花多少时间去修复产品故障所造成的后果，而这一点常常都被忽视了。例如，某种在通信前端运行的软件系统平均每6个月才会出现一次故障，但是当这种故障出现的时候，它会把后端的整个数据库系统都毁掉。数据库最多可以恢复到最后一次备份的时候，使用审计跟踪可以使数据库处于实际上最新的状态。然而，这个恢复过程至少需要两天时间，在此期间数据库系统和通信前后端都处于非工作状态。因此，哪怕该产品6个月才出会现一次故障，其可靠性也是非常低的。

6.4.3　健壮性

软件产品另一个需要测试的方面是它的健壮性（robustness）。虽然很难给健壮性下个准确的定义，但是健壮性基本上是一系列因素（如运行条件的范围、有效输入带来错误输出的可能性以及产品的输入无效时结果的可接受性）的函数。如果一个软件能够支持多种不同的操作环境，它就比那种只能接受一种单一操作环境的软件更加强壮。一个健壮性较好的软件不应该在输入数据符合要求时却给出错误的输出，例如，输入任何一个合法的命令都不应导致灾难性的后果。当产品在不允许的运行条件下使用时，一个健壮产品不应崩溃。为了测试健壮性，通常需要给软件系统提供一些输入规格以外的输入数据，并且检验软件的输出结果。例如，当一个软件系统请求用户输入一个名称的时候，软件测试员可能会输出一串不可接受的字符序列（如“ctrl-A Esc-% ? $ #@ ”）作为应答。如果计算机能够给出提示信息“输入不正确，请重新输入”，或者更好地，提示用户这些数据为何不符合要求，其健壮性就比那些输入的数据不符合要求时就崩溃的产品要好得多。

6.4.4 性能

性能（performance）是软件测试中所必须关注的另外一个方面。例如，必须了解软件是否符合时间和空间上的要求。对于一个嵌入式计算机系统（如地对空导弹的控制系统）来说，系统空间上的限制要求软件只能够使用的主存大小为128MB。无论导弹控制软件多么优秀，如果它需要256MB主存的话，就根本无法使用。（要进一步了解嵌入式软件，请参阅备忘录6.2）。

实时软件的特征是硬件时间限制严格，因为在实时系统中，如果不能满足该限制，信息就有可能会丢失。例如，核反应炉的控制系统需要采样内核的温度，每0.1秒处理该数据。如果这个系统的速度不够快，不能处理每0.1秒从温度传感器传过来的中断的话，就会丢失相关的温度信息，并且这些丢失的信息是没有办法找回来的，下一次系统收到温度读数已经是0.1秒以后了，而非先前丢掉的数据。如果反应堆正处在融化的关键点上，那么接收和处理所有的温度读数就显得非常重要。对于所有的实时系统来说，软件在时间方面都必须严格符合规格说明中列出的时间限制。

备忘录6.2

嵌入式电脑是某些非用于计算目的的大型系统的组成部分，嵌入式软件的作用是控制含有嵌入式电脑的设备。例如，军事中用在战斗机上的飞行控制系统，或者在洲际导弹尖端的控制设备。洲际导弹中的控制电脑仅仅用于控制这颗导弹的飞行，它们不能用于其他方面，比如说给导弹基地的士兵打印工资单。

更加为人所知的例子是电子手表或者洗衣机中的电脑芯片。洗衣机中的芯片只能用于控制洗衣机清洗衣服，没办法用它来核对家庭账本。

6.4.5 正确性

最后，要给出的是正确性（correctness）的定义。产品的正确性是指当软件在规格说明许可的条件下运行时，其输出结果符合规格说明的要求，而与使用的计算资源无关［Goodenough，1979］。换句话说，如果输入数据满足规格说明的规定，并且给产品提供其所需要的所有资源，那么一个正确的产品应该能给出符合规格说明的输出结果。

正确性的定义就如同上文提到的有关测试的定义一样，有令人困惑的地方。假设一个产品已经通过了大量的不同测试，这是否就意味着该产品可以为用户所接受？遗憾的是，不能。因为软件在测试数据上运行正确，只是说明软件产品符合规格说明，但是如果规格说明本身不正确呢？为了说明这个问题，请看图6-1，图中的规格说明标明了输入数据p是由n个整数构成的数组，而输出数据q是按非降序排列的另一个数组。虽然从表面上看，这个设定很好地描述了一个排序程序，但是考虑图6-2所示的trickSort方法，在这个方法中，数组q里的n元素都为0。按照上述定义，这是符合规格说明的，是正确的，但这其中显然存在问题。

```
输入规格说明:  p:array of n integers,n>0.

输出规格说明:  q:array of n integers such that
                  q[0]≤q[1]≤…≤q[n-1]
```

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规格说明的trickSort方法

问题出在哪里？实际上，图6-1的规格说明本身就是不正确的。它忽视了输出数组q的各元素的状态是输入数组p的一个置换（重新排列）。排序的本质在于，它是一个数据重新安排的

过程，但如果不在设定中加以明确说明，就会导致图 6-2 的状况。换句话说，tricksort 方法是正确的，图 6-1 的设定是错误的。图 6-3 展示了修改后的规格说明。从这个例子可以看出，软件规格说明中的错误并不是一个小问题。毕竟，如果一个规格说明本身是错误的，那么产品的正确性也就无从谈起了。

输入规格说明：	p:array of n integers,n>0.
输出规格说明：	q:array of n integers such that q[0]≤q[1]≤…≤q[n-1] 数组q的各元素是数组p各元素的置换，不能改变

图 6-3　改正后的排序规格说明

如果软件的正确性是不充分的，那么它是否是必要的呢？例如，一个软件开发组织新得到了一款超级 C++ 编译器。这种编译器的编译速度是以前使用的编译器的 2 倍，其运行二进制代码的速度比原先快了 45%，而目标代码的规模却减小了大约 20%。另外，该编译器的错误提示更加简单易懂，每年维护和更新的费用比旧编译器少了一半。然而这个编译器存在一个问题，它对于每个类中出现的第一个 **for** 语句都会给出一个错误提示信息。这个编译器是不正确的，因为编译器的规格说明已经明确指出当且仅当代码中确实存在错误时，编译器才应该给出错误提示信息。然而，这个编译器仍然是可用的，事实上，除了这一点瑕疵以外，这个编译器的各个方面都很理想。甚至可以期待这个小错误在下一个发行版中予以更正。同时，使用这个编译器的程序员将逐渐学会忽视那个不正确的错误提示信息。因此，不但这个软件组织可以使用这个新编译器，而且甚至没有任何人会愿意回到以前那个老版本的编译器去。从以上例子可以看出，产品的正确性是既不充分也不必要的。

尽管上面举的例子都有些人为臆造，但它们却切中要点，那就是产品的正确性仅仅表达了软件能够符合它的规格说明而已。换句话说，除了显示产品是正确性的以外，还有许多其他测试要做。

对于所有与基于执行的测试有关的难点，计算机科学家已设法提出其他办法来确保产品按期望运行。一个这样的基于非执行的测试方法已经受到了 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（correctness proof）是证明产品正确的一种数学技术。该技术有时被称为*验证*。然而上文已经提到，验证这个词通常用于表示所有基于非执行的测试技术，而不只是正确性证明。因此，本书将一直使用正确性证明来指称这种数学证明手段。

6.5.1　正确性证明的例子

为了说明如何证明正确性，请看图 6-4 中的代码片断。图 6-5 是对应的程序流程图。下面将证明这个代码片断是正确的，在该代码运行后，变量 s 将包含数组 y 的 n 个元素的和。在图 6-6 中，每个语句前后都放置了一个*断言*（assertion），并且标上了字母 *A* 到 *H* 作为顺序标号。也就是说，在具有某种数学属性的地方作了一个声明。下面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

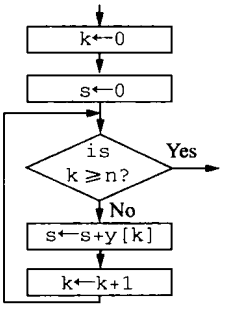


图 6-5　图 6-4 代码段的流程图

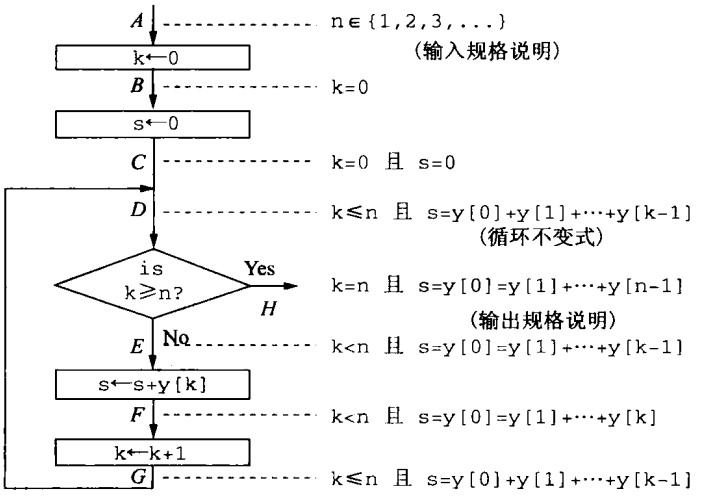


图 6-6 带有输入规格说明、输出规格说明、循环不变式和断言的流程图

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，即

$$A:\quad n \in \{1, 2, 3, \cdots\} \tag{6-1}$$

明显的输出规格说明是，如果控制达到了点 H，那么 s 的值就应该是 y 中存放的 n 个值的和：

$$H:\quad s = y[0] + y[1] + \cdots + y[n-1] \tag{6-2}$$

事实上，对于更强的输出规格说明，也可以证明这段代码是正确的：

$$H:\quad k = n \text{ 且 } s = y[0] + y[1] + \cdots + y[n-1] \tag{6-3}$$

大家自然会问，上面（6-3）的这个输出规格说明是从何而来的？在本证明的最后，希望你能回答这个问题；关于这个问题，也可以参照习题 6.10 和习题 6.11。

除了输入和输出规格说明，本证明过程的第 3 方面是要提供一个循环不变式，即无论循环运行了 0、1 或者很多次，在点 D 程序必须满足一个数学表达式。此处要证明的循环不变式（loop invariant）是：

$$D:\quad k \leq n \text{ 且 } s = y[0] + y[1] + \cdots + y[k-1] \tag{6-4}$$

下面将表明如果在 A 点的输入规格说明（6-1）成立，那么输出规格说明（6-3）在 H 点成立，也就是说，证明这个代码片断是正确的。

首先，执行赋值语句 k←0。现在控制在 B 点，下面的断言成立：

$$B:\quad k = 0 \tag{6-5}$$

为了更准确，在点 B 的断言是 k=0 且 n∈ {1, 2, 3, …}。然而，输入规格说明（6-1）在流程图中的所有点都成立。为了简略起见，下面将 n∈ {1, 2, 3, …} 这部分省略了。

在点 C 处，作为第二个赋值语句 s←0 的结果，下面的断言成立：

$$C:\quad k = 0 \text{ 且 } s = 0 \tag{6-6}$$

现在进入循环部分。现使用数学归纳法来证明循环不变式（6-4）是正确的。在开始执行第一次循环以前，断言（6-6）成立，即 k=0 并且 s=0。对于循环不变式（6-4）来说，由断言（6-6）知 k=0 且由输入规格说明（6-1）知 n≥1，因此 k≤n 成立（按照要求的那样）。进一步说，由于 k=0，故而 k-1=-1，所以（6-4）中的和为空，而且如所求 s=0。综上，循环不变式（6-4）在第一次进入循环前成立。

接下来进行归纳假设的步骤。假设在代码执行中的某个阶段，该循环不变式有效，即 k 等于某个值 k_0（$0 \leq k_0 \leq n$），执行到 D 点，下面断言成立：

$$D:\quad k_0 \leq n \text{ 且 } s = y[0] + y[1] + \cdots + y[k_0 - 1] \tag{6-7}$$

现在控制经过测试框。如果 $k_0 \geqslant n$，那么由于归纳假设 $k_0 \leqslant n$，必有 $k_0 = n$。按照归纳假设（6-7），这意味着：

$$H:\ k_0 = n \ \text{且}\ s = y[0] + y[1] + \cdots + y[n-1] \tag{6-8}$$

上式正好符合了前文提到的规格说明（6-3）。

另一方面，如果测试 $k_0 \geqslant n$ 结果为否定的，则控制越过 D 点到达 E 点。因为 k_0 不大于或者等于 n，所以 $k_0 < n$，而式（6-7）变为：

$$E:\ k_0 < n \ \text{且}\ s = y[0] + y[1] + \cdots + y[k_0 - 1] \tag{6-9}$$

现在执行语句 $s \leftarrow s + y[k_0]$。根据断言（6-9），在 F 点，下面的断言必然成立：

$$F:\ k_0 < n \ \text{且}\ s = y[0] + y[1] + \cdots + y[k_0 - 1] + y[k_0]$$

$$= y[0] + y[1] + \cdots + y[k_0] \tag{6-10}$$

下面要执行的语句是 $k_0 \leftarrow k_0 + 1$。为了说明这个语句执行的效果，假设在执行该语句前 k_0 的值是 17，那么式（6-10）中和的最后一项是 y [17]。现在 k_0 的值增加 1 变为 18。和 s 没有变，所以求和序列中的最后一项仍然是 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 点不等式成立，那么 $k_0 \leqslant n$。因此，把 k_0 的值增加 1 后，下面的断言在点 G 成立：

$$G:\ k_0 \leqslant n \ \text{且}\ s = y[0] + y[1] + \cdots + y[k_0 - 1] \tag{6-11}$$

在 G 点的断言（6-11）和在 D 点的断言（6-7）相同。但 D 点和 G 点在拓扑上是一致的。换言之，对于 $k = k_0$，如果式（6-7）在 D 点成立，那么对于 $k = k_0 + 1$，它在 D 点也成立。如前所示，循环不变式（6-4）在 k = 0 时成立。通过归纳，对所有的 k（$0 \leqslant k \leqslant n$）值，循环不变式（6-4）都成立。

剩下就是证明循环终止。最初，根据断言（6-6），k 的值等于 0。每次循环执行语句 $k \leftarrow k + 1$时，k 的值都会增加 1。最终 k 一定会达到值 n，那时将退出循环，并且 s 的值由断言（6-8）给出，这样就满足了输出规格说明式（6-3）。

对上述证明过程总结如下：根据给定的输入规格说明（6-1），证明循环不变式（6-4）在循环执行 0、1 或者更多次时都成立。随后，证明在 n 次迭代后，循环体能够终止。而当它结束时，k 和 s 的值满足输出规格说明（6-3）。换句话说，图 6-4 所示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应与设计和编程相伴而行。Dijkstra 称："程序员应该让程序的正确性证明和程序编写一起进行"［Dijkstra，1972］。例如，当设计中引入一个循环体时，要同步标出它的循环不变式。当这个设计逐步求精时，不变式也逐步求精。用这种方法开发软件产品能够增加程序员对产品正确性的信心，并且减少错误的数量。再次引用 Dijkstra 的话，"唯一能够增强程序员自信心的方法就是给他们提供具有说服力的程序正确性的证明"［Dijkstra，1972］。不过就算已经证明一个产品是正确的了，也必须再进行全面的测试。为了说明测试那些通过正确性证明的软件的重要性，请考虑下面的例子。

1969 年，Naur 提出了一种建造和证明产品正确性的方法［Naur，1969］。Naur 使用行编辑问题作为研究对象对该方法进行了阐述，这个问题今天可以被看作是文本处理问题。问题可叙述如下：

给出一段由空格和换行符隔开的单词所构成的文本，依照以下原则将它转化为行文本形式：

1）只能在现在是空格或者换行符的地方才能断行。

2）只要可能，尽可能填充每一行。

3）每行不会包含超过 Maxpos 个字符。

Naur 使用这种方法构造了一个程序，并非形式地证明了它的正确性。此程序由大约 25 行代

码构成。他的论文后来由《Computing Review》的 Leavenworth 进行评审［Leavenworth，1970］。评审者指出，在 Naur 的程序给出的输出数据中，除非第一个单词正好和 maxpos 一样长，否则第一个单词的前面总是会多出一个空格。虽然这看起来只是一个小错误，但是如果对代码用测试数据进行测试而不是仅仅进行证明的话，它肯定可以在测试过程中检测出来。不过，更糟糕的是，London［1971］在 Naur 的程序中又找到了另外 3 个错误。其中一个错误是，如果程序找不到一个比 maxpos 还要长的单词的话，它就永远停不下来。另外，如果已经测试过该程序，那么很可能检测出这个错误。London 随后提供了一个该程序的修正版本，并且形式化地证明它的正确性，而 Naur 使用的仅是非形式化证明技术。

此外，Goodenough 和 Gerhart［1975］又在程序中发现了 London 没有找到的另外 3 个错误。其中一个错误是，如果输入文件的最后没有空格或者换行的话，最后一个单词就不会输出。这也是一个非常容易通过执行测试找出的错误。事实上，Leavenworth、London、Goodenough 和 Gerhart 总共在 Naur 的程序中找出了 7 个错误，而这其中的至少 4 个错误可以通过运行测试数据很方便地找出来，并不需要如此大费周章地进行这么多次正确性证明。这个故事的启发是，就算一个产品已经通过了正确性证明，还是要对它进行全面的测试。

6.5.1 节中的例子表明，就算是要证明一个很小的程序模块，正确性证明也是一个漫长的过程。本节的实例研讨表明，正确性证明就算对于一个只有 25 行代码的程序来说，也是一个非常困难而且极容易犯错误的过程。那么，正确性证明是只具有理论上的研究价值，还是在真实的软件工程项目中是一项强有力的工程技术？这些问题将在 6.5.3 节中得到解答。

6.5.3 正确性证明和软件工程

许多软件工程人员提出，正确性证明不能成为标准软件工程技术。首先，有人认为软件工程师缺乏足够的进行正确性证明的数学训练。其次，由于进行软件正确性证明的代价过于昂贵，其往往被认为是没有实用价值的。最后，许多人认为进行正确性证明过于困难。然而，下面表明了以上提及的每个理由事实上都把问题过度单纯化了。

1）虽然 6.5.1 节的证明只使用到了高中的数学知识，但是实际的证明过程需要把输入规格说明、输出规格说明和循环不变式表达为一阶或者二阶谓词计算或等价计算。这样做的目的不仅仅在于方便数学家进行证明，而且还使得电脑能自动证明正确性。不过，对于更加复杂的正确性证明，谓词计算已经有点过时了。为了证明并发软件系统的正确性，必须使用时态逻辑或者模态逻辑［Manna and Pnueli，1992］。显然，正确性证明确实需要一些数学上的训练，幸运的是，目前高校计算机专业要么开设了与正确性证明相关的必修课，要么其学生有机会在工作背景中学习该技术。因此，现在高校培养的计算机系毕业生事实上已经具备了进行正确性证明的足够数学功底。在以前，说软件工程人员缺乏足够的数学能力也许是正确的，但是对于现在每年数千人的计算机系毕业生来说，情况已经不再是这样的了。

2）关于正确性证明的成本过于昂贵的说法也是不正确的。正确性证明的经济可行性应该根据每一个软件项目使用的成本－效益分析法（5.2 节）。以国际空间站的软件系统为例，在该系统中，当事故发生时，由于救援航天飞机可能不能及时赶到，从而使航天员濒临危险。虽然证明空间站的生命维持系统的正确性的代价是昂贵的，但是如果系统出错必将付出更加昂贵的代价。

3）虽然正确性证明看起来非常困难，但是事实上许多复杂的软件产品都成功地通过了正确性证明，包括操作系统内核、编译器和通信系统［Landwehr，1983；Berry and Wing，1985］。此外，还有许多诸如定理证明器之类的工具来帮助进行正确性证明。定理证明器将一个软件产品的输入、输出规格说明以及循环不变式作为输入，然后从数学上证明如果输入数据满足输入规格说明，那么该产品的输出数据也将满足输出规格说明。

与此同时，正确性证明同样也面临一些困难：

- 例如，如何确定定理证明器的正确性呢？如果定理证明器输出“This product is cor-

rect”，能相信吗？作为一种极端的情况，请看图 6-7 中的定理证明器。无论给它提供什么样的输入，它都会输出“This product is correct。”换句话说，对于定理证明器的输出，到底可以在什么程度上信任它？一种建议是把定理证明器提供给其自身作为输入，然后看看它是否正确。对于这种方法，除了一些哲学上的争论外，只要看看图 6-7 就知道它一定行不通。如果把图 6-7 中的程序提交给其自身的话，它依然会打印出“This product is correct”，从而“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

- 另一个困难是，如何寻找合适的输入和输出规格说明，特别是循环不变式或者以其他逻辑（如模态逻辑）所表示的等价式。假设一个产品是正确的，但除非能够顺利地找到每个循环体的循环不变式，否则要证明它的正确性也很困难。虽然在这方面也有一些辅助性工具，但是仅仅凭借今天已有的工具还不足以保证软件工程师们一定能够完成证明。解决这个问题的一种可能的方案是，像 6.5.2 节说的那样，在开发产品时同步完成正确性证明。在设计一个循环时，就同步写出它的循环不变式。如果使用这种方法，证明代码的正确性在某种程度上会简单一些。
- 除了输入和输出规格说明以及循环不变式方面的困难以外，还必须意识到规格说明本身就有可能不正确。图 6-2 的 trickSort 就是这样的一个例子。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［1978］指出，“永远无法证明软件规格说明的正确性”并且“永远无法证明软件验证机制的正确性”。这两位计算机科学领域的顶尖专家给出的综述在一定程度上概括了上文提及的各种观点。

这样是否就意味着正确性证明在软件工程中是没有位置的呢？恰好相反，正确性证明对软件工程来说非常重要，甚至在某些情况下是至关重要的。在人命关天的场合或者成本－效益分析法指出有必要证明正确性的场合，则应当进行证明。不过，正像上文实例研讨部分指出的那样，很多时候仅仅进行证明是不够的。一般来说，应该把正确性证明看做是综合验证产品正确性整套技术中的一个重要组成部分，它必须和其他方法配合使用才能达到良好的效果。因为软件工程的目标是生产高质量的软件产品，所以正确性证明确实是一种重要的软件工程技术。

即便一个完全形式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得到明显提高。例如，类似于 6.5.1 节的证明能够辅助检查循环执行的次数是否符合要求。另一种提高软件质量的方法是，在代码中插入一些类似于图 6-6 中的数学断言。如果这些断言语句不被满足的话，软件会自动中止运行，然后开发小组就可以来检查问题出在哪里：究竟是断言有问题，还是软件本身确实有错误。某些高级语言（如 Java（版本 1.4 或者更高））支持直接书写 **assert** 语句进行断言。例如，假设一个非形式证明指出代码中的某个环节，变量 xxx 的值必须大于 0。即使开发小组的人员认为变量 xxx 在此处不可能小于 0，但为了软件的可靠性，最好还是加上一个断言语句：

assert (xxx > 0)

如果系统在此检测到 xxx 的值小于等于 0，程序将会自动中止运行，并且提示软件开发小组来处理这个问题。不过，C++ 中 Assert 是个调试指令，与 C 中的 assert 相似，而不是语言本身的语句。Ada 95［ISO/IEC 8652，1995］可以通过 **pragma** 来支持断言。

当用户对于产品的正确运行有了足够的信心后，就可以关闭断言检查，这将加快软件运行速度，但是那些有机会被断言检查找到的错误将不会再被发现。因此，即使在产品的最终发布版已经交给用户后，还必须在运行效率和连续的断言检查间作出某种平衡（备忘录 6.3 对这个问题进行了深入剖析）。

另一个有关基于执行的测试的基本问题是软件开发小组的哪些人应该来负责实现它。这个问题将在以下的小节中讨论。

备忘录 6.3

Java 等（除 C 和 C++ 外）高级语言具有一种叫做边界检查的功能。例如，在运行时，Java 系统会检查每一个数组索引，以确保它们都在声明的范围内。

霍尔（Hoare）提出在开发软件时使用边界检查而在软件能够正常运行后就关闭它，就好像是在地面上穿着救生衣学航海，然后真正航海时把救生衣脱掉一样。在他的图灵奖讲稿中，霍尔描述了一个他在 1961 年开发的编译器［Hoare，1981］。当用户收到这个编译器的最终版本时，发现它可以把边界检查功能关掉，但所有的用户都拒绝这么做，因为在前期测试版本的编译器中，大家发现了很多数组索引越界的情况。

边界检查可以看做是断言检查的一种特殊情况。霍尔的关于救生衣的比喻与在软件最终版本完成后关闭所有的断言检查一样。

霍尔的评论不幸被言中了。当今一个主流的黑客手段就是向计算机发送一个非常长的二进制流。当这个二进制流超出缓冲区的限制而又没有进行边界检查的话，操作系统的一部分二进制代码就会被黑客的代码所改写。不过，如果在 C 或 C++ 的系统中引入人工的边界检查，或者使用更高级的语言来编写系统的话，这种状况还是可以避免的。

6.6 由谁来完成基于执行的测试

假定要求程序员测试其编写的代码。就像 Myers［1979］描述的那样，测试是带着找出错误的目的去执行产品的过程。因此，测试工作实质上是一个破坏性的工作。从另一个方面来说，程序员如果测试他自己写的代码的话，他通常并不想破坏掉他自己的工作成果。如果测试员对待程序的基本态度是带有一定的保护性的话，通常他找出的错误就不会有那种彻底以破坏为目的去测试的测试员来得多。由于一个成功的测试工作应该能找出错误，因此这里存在着一个悖论。如果程序通过了测试，那么就说明测试本身失败了。而如果程序没有通过基于规格说明编写的测试，那么说明测试成功了，但是程序本身存在问题。因此，如果让撰写代码的那个程序员去测试他自己的代码的话，那么不管怎样他都得面临一个失败的结果，而这种状况是和程序员的创造性本能背道而驰的。

因此，显然不应该指派程序员去测试他们自己编写的代码。由于编写程序是创造性的而测试是破坏性的，这两个工作显然不应该由同一个人来完成。此外，由于程序员可能会误解规格说明，因此测试工作也最好由其他的人来帮助评审。当然，找出错误的根源并且修正代码的工作最好还是由程序员自己来做，因为他对自己编写的代码最熟悉。

程序员不能测试他自己的代码这个结论也不能太过绝对。在撰写程序的过程中，程序员首先要阅读设计文档，而设计文档可能是一个流程图或者一段伪代码。无论程序员采用何种手段去编写程序，显然他必须先对设计蓝图作一番桌面检查（desk check）。也就是说，程序员必须先假想使用各种测试用例去测试考验程序流程图或伪代码，并且确定每个细节都正确。只有当整个设计都被详细地审查完之后，程序员才会开始实际的编程工作。

当代码成为可以运行的形式后，程序员自己将开始对它进行一系列的检查。测试数据用于确定代码制品是否能正常工作，在桌面检查详细设计时可能会用到同样的测试数据。接下来，如果程序对于正确数据的输出正确的话，程序员再使用非正确输入来测试软件的健壮性。如果程序员对自己的代码感到满意的话，就把程序提交到系统测试环节。系统测试（systematic testing）不应该由程序员自己来完成。

如果程序员自己不负责系统测试，那么谁来负责做这种测试呢？6.1.2 节已经提到，独立测试应该由 SQA 小组来完成。这里一个重要的概念是“独立性”。只有当 SQA 小组确实相对开发小组独立的情况下，他们才能确保产品符合其规格说明，而没有软件开发管理者施加诸如产品的最后期限这种妨碍工作的压力。SQA 小组的成员应该只对他们自己的管理者负责，从而保证他们工作的独立性。

那么，系统测试应该如何开展呢？测试用例的一个重要部分是必须明确指出期望得到的输出结果。如果测试员只是坐在电脑前面，运行程序代码，输出一堆测试数据，瞅一眼屏幕然后就说：“我想这结果可能是对的”，那就只是纯粹浪费时间而已。同样没有效果的事情还包括测试员花费大量的时间准备测试样例，然后逐一运行每个程序，但是不重视运行结果。测试员很容易被似是而非的结果所蒙蔽。如果程序员测试自己的代码的话，他们很可能只会看到自己想要的结果。就算是由别人来完成测试，这里也存在同样的问题。解决的方法就是，必须坚持测试样例和测试期望的结果必须在测试前都设计完毕。当测试工作结束时，测试员就可以用实际的输出结果和预先规划的测试结果来进行比较。

就算是小型软件组织开发出的小产品，把这些记录用机器可读的方式保存下来也是很重要的，因为测试数据不应该被丢弃掉。后续的售后服务中，可能会用到这些数据，因为那时必须进行*回归测试*（regression testing）。保存下来的以前的测试数据，在每次向产品加入新功能时都应该再测试一遍，以保证新加入的功能没有影响到产品的原有功能。这方面的问题将在第 14 章进一步加以讨论。

6.7　测试何时停止

当一个产品成功地运行了很多年以后，它可能渐渐失去作用，并且被一个全新的产品所替代，就像电子管被晶体管替代一样。或者某个产品虽然仍然有用，但是把它转移到新的平台上所花费的成本，要比完全开发一个全新产品还要高。在这些情况下，原有的产品将不再提供服务，并且从产品线中退役。只有这时，当一个软件产品被彻底废除的时候，对于它的测试工作才能够停止。

既然已经介绍了所有必须的背景知识，现在就可以更深入地考察对象了。第 7 章将详细论述这些话题。

本章回顾

本章的核心主题是测试工作必须随着软件的开发工作同步展开。本章开始时讨论了一些有关质量的问题（6.1 节），随后讨论了基于非执行的测试（6.2 节），其中详细讨论了走查和审查。接下来，定义了基于执行的测试这一概念（6.3 节和 6.4 节），并且讨论了软件产品必须被测试的行为属性，包括实用性、可靠性、健壮性、性能和正确性（6.4.1 ~ 6.4.5 节）。6.5 节介绍了正确性证明，并且在 6.5.1 节中给出了一个例子，然后在 6.5.2 节和 6.5.3 节分析了正确性证明在软件工程中的作用。另一个重要的问题是系统的基于执行的测试必须由独立的 SQA 小组来开展，而不是由程序员自己来完成（6.6 节）。最后，在 6.7 节中讨论了何时测试最终结束。

延伸阅读材料

软件工程人员对于测试工作的看法一直随着时代而变化。起初，大部分软件人员把测试工作看成表明软件能够正常工作的手段；而现在则认为测试工作主要是为了防止需求、分析、设计和实现出现错误。[Gelperin and Hetzel, 1988] 描述了这个发展过程。软件测试工作的本质和这种工作特别困难的原因在 [Whittaker, 2000] 中有所描述。软件错误的普遍性在 [Lieberman and Fry, 2001] 中有所讨论。削减错误的方法可以参阅 [Boehm and Basili, 2001]。

关于软件质量的一些传奇故事在［Voas，1999］中有所讨论。［Whittaker and Voas，2000］描述了一种有关可靠性的新理论。

［Baber，1987］很好地介绍了有关正确性证明的内容。［Hoare，1969］介绍了进行正确性证明的一种标准方法——霍尔逻辑。保证软件符合规格说明的另一种方法是在软件开发的整个过程中同步检查其正确性，［Dijkstra，1968］和［Wirth，1971］描述了这种方法。［DeMillo，Lipton，and Perlis，1979］是论述正确性证明在软件工程界被接受程度的一篇重要文章。

IEEE的“软件评审标准”［IEEE 1028，1997］是基于非执行的测试方面的重要文献。［Perry et al.，2002］中描述了检查评估大型软件产品的一些实验。［Vitharana and Ramamurthy，2003］认为审查工作应该是匿名的，而且要通过电脑中介。［Tyran and George，2002］讨论了团队的支持工作对于审查的影响。［Miller and Yin，2004］研究了应该如何选择审查小组成员的问题。对于审查工作的一系列总结见［Parnas and Lawford，2003］，而当前这方面实际工作开展的情况则在［Ciolkowski，Laitenberger，and Biffl，2003］中有所讨论。面向对象的代码审查工作参见［Dunsmore，Roper，and Wood，2003］。

关于基于执行的测试的一些经典著作请参考［Myers，1979］，该书对于测试领域的影响很大。［DeMillo，Lipton and Sayward，1978］是关于测试数据选择方面的一篇重要文章。［Beizer，1990］提纲挈领地描述了整个测试工作，是该课题的真正有用的参考书。还有［Hetzel，1988］与此书类似。

对于面向对象的软件工程中的测试工作，可以参阅［Kung，Hsia，and Gao，1998］和［Sykes and McGregor，2000］。

国际软件测试与分析论坛（International Symposium on Software Testing and Analysis）提供了丰富的关于测试工作的资料。2002年2月的《IEEE Transactions on Software Engineering》杂志包含了2000年软件工程会议的一些文章，［Elbaum，Malishevsky，and Rothermel，2002］是其中特别有意义的一篇。

习题

6.1 请说明“正确性证明”、“验证”和“确认”这几个术语在本书中是如何使用的？

6.2 一个软件开发组织现有96名软件工程师，包括19名管理者，所有人员既从事开发又从事测试。最新统计数据表明，这些员工大约27%的时间用在测试工作上。管理人员平均每年的成本是144 000美元，而普通员工平均每年的成本是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需要在这个公司中设立一个独立的SQA小组。

6.3 假设习题6.2的情况只有7名软件工程师，其中2名管理者，其他数据均保持不变，请分析此时是否值得建立一个独立的SQA小组。

6.4 如果你已经花了11天来测试某段代码，并且已经找到了2个错误，请问这能预示其他错误的存在吗？

6.5 走查与审查之间有什么相似之处？又有什么不同？

6.6 假设你是Ye Olde Fashioned Software公司的SQA小组的一员。现在你建议管理者引入审查机制，而管理者则说既然程序员能执行测试用例，就没有必要浪费4个人的时间去对产品进行审查。你要如何回答呢？

6.7 假设你是Hardy Hardware的SQA管理者，而Hardy Hardware是一个拥有754家连锁店的硬件销售商。现在公司决定购买一套库存管理系统并在全公司中使用，而你决定在购买这批产品前先对其进行彻底的测试。你应该测试这个产品的哪些属性？

6.8 假设Hardy Hardware的754家商店都需要用一种通信网络连接起来，通信软件包的销售代表试图向你推一种新型的通信网络，他承诺给你4周的免费试用期。现在你应该进行哪些方面的测试？为什么？

6.9 假设你是Valorian Navy海军少将，负责开发类似于习题1.4中的舰对舰导弹的控制程序。现在软件系统已经开发完成并且交给你来做验收测试。那么该软件的哪些属性应当被测试？

6.10 如果用以下循环不变式来取代6.5.1节中正确性证明用到的循环不变式会怎样？

$s = y[0] + y[1] + \cdots + y[k-1]$

6.11　假设你在设定循环不变式方面富有经验，并且已经得知（6.4）式就是图6-6中所示循环的正确的循环不变式。现在请证明输出规格说明式（6.3）是这种循环不变式的正常结果。

6.12　考察以下代码片段：

```
k = 0;
g = 1;
while (k < n)
{
  k = k + 1;
  g = g * k;
}
```

证明：如果n是一个正整数的话，该代码片段正确计算了g = n!。

6.13　正确性证明能不能解决“发布的产品可能不是用户真正想要的产品”的问题？请解释你的答案。

6.14　Dijkstra的理论（6.3节）如果不是用于测试而是用于正确性证明的话，应该作出何种改动？参考6.5.2节的实例研究。

6.15　根据指导老师指定的语言，撰写程序实现6.5.2节提到的Naur文本处理问题。用测试数据去测试你的程序，并且记录测试结果，以及每个错误的原因（例如，逻辑错误、循环变量错误等）。不要去修正你找到的错误，然后和其他同学交换作业，并且测试别人的程序。比较一下现在你找到的错误是否是新的错误，然后也记录下每个错误的原因。将班上的结果统一列表。

6.16　为什么要区分差错、故障和错误三个概念？使用缺陷来一言以蔽之真的能简化理解吗？

6.17　（学期项目）解释你将要如何测试附录A中Osric办公用品和装饰产品的实用性、可靠性、健壮性、性能和正确性。

6.18　（软件工程读物）教师分发论文［Miller and Yin，2004］的复印件。你会使用基于感知的审查小组选择机制吗？请证明你的答案是正确的。

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.

[Ciolkowski, Laitenberger, and Biffl, 2003] M. Ciolkowski, O. Laitenberger, and S. Biffl, "Software Reviews, the State of the Practice," *IEEE Software* **20** (November/December 2003), pp. 46–51.

[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.

[Dunsmore, Roper, and Wood, 2003] A. DUNSMORE, M. ROPER, AND M. WOOD, "The Development and Evaluation of Three Diverse Techniques for Object-Oriented Code Inspection," *IEEE Transactions on Software Engineering* **29** (August 2003), pp. 677–86.

[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.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries*, ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, 1995.

[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*, 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.

[Miller and Yin, 2004] J. MILLER AND Z. YIN, "A Cognitive-Based Mechanism for Constructing Software Inspection Teams," *IEEE Transactions on Software Engineering* **30** (November 30), pp. 811–25.

[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.

[Parnas and Lawford, 2003] D. L. PARNAS AND M. LAWFORD, "The Role of Inspection in Software Quality Assurance," *IEEE Transactions on Software Engineering* **29** (August 2003), pp. 674–76.

[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

通过本章学习，读者应能：

- 设计出高内聚低耦合的类。
- 理解信息隐藏的必要性。
- 描述出软件工程中继承性、多态性和动态绑定的含义。
- 区分泛化、聚合和关联。
- 更深层地讨论面向对象范型。

一些比较耸人听闻的计算机杂志似乎提出：面向对象范型是 20 世纪 80 年代中期的一项戏剧性的意外发现，是替代当时流行的传统范型的一种革命性选择。然而事实并非如此，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的简单进化（参见备忘录 7.1）。本章将在模块范畴中描述对象。

采用这种方法是因为，如果不理解为什么面向对象范型比传统范型优越，将很难正确使用对象。为此，有必要认识到：在始于模块概念的知识体中，对象只是下一个逻辑步骤。

> **备忘录 7.1**
>
> 早在 1966 年，仿真语言 Simula 67［Dahl and Nygaard，1966］就有了面向对象的概念。然而，在当时，这项技术对现实应用来说太超前了，它一直默默无闻，直到 20 世纪 80 年代初在模块化理论中才重新使用它。
>
> 本章还包括其他有关前沿技术的例子：这些技术一直搁置，直到世界能够接受它们为止。例如，信息隐藏（见 7.6 节）是 1971 年在软件领域中由 Parnas 第一次提出［Parnas，1971］，但是大约直到 10 年后，当封装和抽象数据类型成为软件工程学的一部分时，这项技术才被广泛采用。
>
> 人类似乎总在准备充分时，才采用新技术，而不是在第一次提出它时就采用。

7.1　什么是模块

当大型产品由单个代码块构成时，其维护将成为一个恶梦。即使是这种产品的开发者去调试代码，也相当困难，如果让其他的编程人员去理解它，几乎是不可能的。解决的方法就是把这个产品分成一些称为模块的小块。模块是什么？是把产品分成模块的方式本身重要，还是仅仅把大型产品分成小的代码块重要呢？

模块（module）是“词汇上相近，由边界元素界定且有着聚集标识符的程序语句序列”［Yourdon and Constantine，1979］。边界元素的例子是 C++ 或 Java 中的 {…} 对。在面向对象范型中，一个对象就是一个模块，对象中的一个方法也是一个模块。

为了理解模块化的重要性，考虑下面带些虚构色彩的例子。John Fence 是一个很不合格的计算机体系结构设计师。他一直没发现与非门和或非门是完备的，也就是说，每个电路都可以只由与非门或者或非门来构成。因此，John 决定使用与门、或门和非门构造算术逻辑单元（ALU）、移位器和 16 位寄存器。所得到的计算机如图 7-1 所示。三个组件以一种简单方式连接。现在，John 决定在三颗硅芯片上构造该电路，他设计了三颗芯片（如图 7-2 所示）。第一颗

芯片包含了 ALU 的所有门电路，第二颗芯片包含了移位器，第三颗则是寄存器。这时，John 想起似乎在酒吧里有人告诉过他，最好构造只含一种门电路的芯片，于是他又重新设计他的芯片。他在第一颗芯片上放置了所有的与门，在第二颗芯片上放置了所有的或门，第三颗芯片上放置了所有的非门。最后的“艺术作品”如图 7-3 所示。

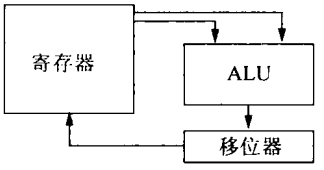


图 7-1　一个计算机的设计

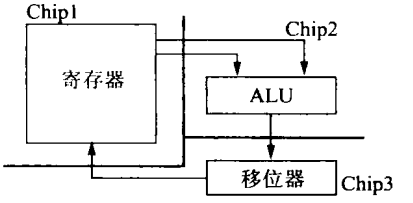


图 7-2　三颗芯片构建的图 7-1 的计算机

图 7-2 与图 7-3 在功能上相同，也就是说，它们做完全相同的工作。但是两种设计有截然不同的属性：

1）图 7-3 比图 7-2 难理解。几乎任何具有数字逻辑知识的人都能立刻明白图 7-2 中的芯片构成一个 ALU、一个移位器和一组寄存器。然而，即使是优秀的硬件专家也很难明白图 7-3 中各种与门、或门和非门的作用。

2）对图 7-3 中的电路进行纠错性维护很困难。如果计算机有一个设计错误，将很难确定错误的位置，而任何完成图 7-3 的人都毫无疑问会犯很多错误。另一方面，如果在图 7-2 中的计算机设计有一个错误，则可以通过错误是 ALU 工作时出现、是移位器工作时出现、还是寄存器工作时出现来进行定位。类似地，如果图 7-2 中的计算机崩溃，可以较容易地决定替换哪颗芯片；如果图 7-3 中的计算机崩溃，可能最好是替换全部三颗芯片。

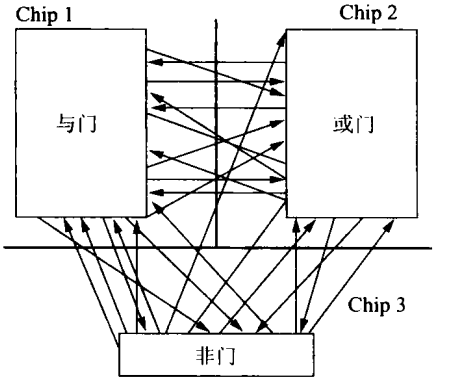


图 7-3　另外三颗芯片上构建的图 7-1 的计算机

3）图 7-3 中的计算机难以扩展或加强。如果需要一种新型的 ALU 或更快的寄存器，必须重新设计电路图。但是对图 7-2 中的计算机设计，要进行芯片替换则相对容易。也许最糟糕的是，图 7-3 中的芯片不能在任何新产品中复用。由与门、或门和非门电路专门组合起来的三颗芯片，除了能用于最初设计的产品外，不能用于任何其他产品。图 7-2 中的三颗芯片则很有可能在其他需要 ALU、移位器或寄存器的产品中复用。

这里的关键是：软件产品的设计要类似图 7-2，在每颗芯片内部有最大的关联，而在芯片之间则关联最小。一个模块与一颗芯片相似，因为它执行一个或一系列操作，并与其他模块相连。确定产品的整体功能后，需要确定如何把产品分成模块。如第 1 章所述，C/SD（Composite/structured design，组合化/结构化设计）[Stevens，Myers，and Constantine，1974] 提供了将产品分解成模块的基本方法，可作为一种降低维护成本和整个软件主要组件的预算成本的途径。当在每个模块内部存在最大关联而在模块之间存在最小关联时，不论纠错性、完善性还是适应性的维护，其维护工作量都会降低。换句话说，C/SD 的目标就是确保产品的模块分解类似于图 7-2 而不是图 7-3。

Myers [1978b] 量化了模块内聚（cohesion）的概念，即一个模块内部的交互程度；量化了模块耦合（coupling）的概念，即模块之间的交互程度。更准确地说，Myers 使用术语强度（strength）而不是内聚。但是内聚更合适，因为模块可能有高强度或低强度之分，而在低强度的表述里还有一些固有的矛盾——不强即弱。为了防止术语的不准确，C/SD 使用内聚这个术

语。一些作者使用术语“绑定”来代替“耦合”。遗憾的是，绑定（binding）已经用于计算机科学的其他方面，例如，值和变量的绑定。但是耦合不会有这样的歧义，因而较合适。

这里有必要区分模块的操作、模块的逻辑以及模块的上下文的概念。一个模块的操作（operation）是指该模块做什么，是指它的行为。例如，模块 m 的操作是计算参数的平方根。模块的逻辑（logic）是指该模块如何完成它的操作，以模块 m 为例，计算平方根的特定方法是牛顿方法［Gerald and wheatley，1999］。模块的上下文（context）是指模块的特定用途。例如，模块 m 用来计算双精度整数的平方根。C/SD 的关键是，应该按一个模块的操作而不是它的逻辑或背景来命名该模块。因此，在 C/SD 中，应该将模块 m 命名为 `computeSquareRoot`，而逻辑和上下文应该和名字无关。

7.2　内聚

Myers［1978b］定义了 7 种内聚范畴或级别。Myers 的上两个级别需要交换一下，因为根据现代计算机科学理论，信息性内聚比功能性内聚能更有力地支持复用，具体将在后面论述。最终的排列如图 7-4 所示，这不是依任何度量值进行线性排序的结果，而仅仅表示相对的分级，用来确定哪些类型内聚程度更高（好）以及哪些类型更低（差）。

7. 信息性内聚	（好）
6. 功能性内聚	
5. 通信性内聚	
4. 过程性内聚	
3. 时间性内聚	
2. 逻辑性内聚	
1. 偶然性内聚	（差）

图 7-4　内聚级别

为了理解高内聚模块由什么构成，必须从另一端（即低内聚级别）着手讨论。

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，就称该模块具有偶然性内聚（coincidental cohesion）。例如，一个命名为 `printTheNextLine`、`reverseTheStringOfCharactersComprisingTheSecondArgument`、`add7ToTheFifthArgument` 和 `convertTheFourthArgumentToFloatingPoint` 的模块就是一个具有偶然性内聚的模块。这里有一个明显的问题：在实际中这些模块是如何出现的呢？这通常是某种严格的强制性规则的结果，例如，“每个模块包含 35 ~ 50 条可执行语句”。如果软件组织坚持模块不能太大或太小，那么就会产生两件令人不愉快的事情。其一，两个或更多不太理想的小模块集中在一起，组成一个更大的具有偶然性内聚的模块；其二，从一些设计良好但管理者认为其太大的一些模块中删减下来的片段结合在一起，同样产生了具有偶然性内聚的模块。

为什么偶然性内聚如此之差？偶然性内聚模块具有两个严重的缺陷。第一，这些模块使产品的可维护性（包括纠错性维护和增强性维护）下降。从试图理解一个产品的角度来看，具有偶然性内聚的模块化过程比完全不进行模块化的情况更糟［Shneiderman and Mayer，1975］。第二，这些模块不可复用。如本节第一段所说的，具有偶然性内聚的模块决不可能在任何其他产品中复用。

不可复用是一个严重的缺陷。开发软件的花费很高，在任何可能的地方复用模块是非常必要的。设计、编码、编写文档以及模块测试都需要花费时间，因此该过程代价昂贵。如果有一个设计良好、经过全面测试并且文档齐全的模块能用于另一个产品中，那么管理者应坚持复用已有模块。但是没有办法复用一个具有偶然性内聚的模块，这样用于开发该模块的费用永远不会得到补偿（在第 8 章将详细讨论复用）。

通常，调整一个具有偶然性内聚的模块是容易的，因为它执行多个操作，把模块分成小的模块，每个小模块执行一个操作。

7.2.2 逻辑性内聚

当一个模块执行一系列相关操作，某个操作由调用模块选择时，称该模块具有逻辑性内聚（logical cohesion）。下面都是具有逻辑性内聚的模块例子。

例1 方法 newOperation 由下面代码调用：

```
functionCode = 7;
newOperation ( functionCode, dummy1, dummy2, dummy3);
//dummy1, dummy2 和 dummy3 是伪变量
//如果 functionCode 等于 7 将不使用它们
```

在这个例子中，被调用的 newOperation 具有 4 个参数，但是如注释中所述，functionCode 等于 7 时就不需要后 3 个参数。如此一来，对于通常的纠错性和增强性维护来说，可读性下降了。

例2 一个执行所有输入和输出的对象。

例3 一个对主文件记录进行插入、删除和修改等编辑操作的对象。

例4 在 OS/VS2 早期版本中，一个能执行 13 种不同操作的具有逻辑性内聚的模块，其接口包括 21 块数据［Myers，1978b］。

当一个模块具有逻辑性内聚时，会产生两个问题。第一，接口很难理解（例 1 是一个这样的例子），从而使得模块整体上不容易理解。第二，完成多个操作的代码可能会发生纠结，导致严重的维护问题。例如，一个执行所有输入和输出操作的对象可能如图 7-5 来构造。如果安装一个新的磁带单元，可能必须修改编号为 1、2、3、4、6、9 和 10 的各部分。这些改变可能反过来影响别的输入输出形式，比如激光打印机的输出，因为 1、3 部分的修改将影响激光打印机。这种纠结的性质是具有逻辑性内聚的模块特征。纠结所产生的更深的影响是难以在其他产品中复用该模块。

1. 处理所有输入和输出的代码
2. 只处理输入的代码
3. 只处理输出的代码
4. 处理磁盘和磁带输入/输出的代码
5. 处理磁盘输入/输出的代码
6. 处理磁带输入/输出的代码
7. 处理磁盘输入的代码
8. 处理磁盘输出的代码
9. 处理磁带输入的代码
10. 处理磁带输出的代码
⋮　　⋮　　⋮
37. 处理键盘输入的代码

图 7-5　执行所有输入和输出的对象

7.2.3 时间性内聚

当模块执行一系列与时间相关的操作时，称其具有时间性内聚（temporal cohesion）。例如，一个具有如下名称的对象就是一个具有时间性内聚的模块：openOldMasterFile，newMasterFile，transactionRecord，and printfile；initializeSalesRegionTable；readFirstTransactionRecordAndFirstOldMasterFileRecord。在过去没有 C/SD 的糟糕日子里，这样一个对象一般称为 performInitialization。

这个对象的操作之间具有很弱的关联，而与其他对象的操作的关联却更强。例如，考虑销售地区列表（salesRegionTable）。它在本对象中进行初始化，而诸如更新销售地区列表（updateSalesRegionTable）和打印销售地区列表（pritnSalesRegionTable）的方法却位于其他对象中。因此，如果销售地区列表的结构改变，可能因为公司业务扩展到原来没有业务的乡村地区，所以一些对象需要修改。不但产生回归错误的可能性更大（由于改变产品中明显不相关的部分所产生的错误），而且如果受影响的对象数量很多时，可能会遗漏一两个对象。如 7.2.7 节所述，最好一个对象拥有销售地区列表的所有操作。当需要时，这些方法能被其他对象调用。

另外，具有时间性内聚的模块不可能在不同的产品中复用。

7.2.4　过程性内聚

如果一个模块执行一系列与产品所遵循的步骤顺序相关的操作，那么这个模块具有过程性内聚（procedural cohesion）。例如，方法 readPartNumberFromDatabaseAndUpdateRepairRecordOnMaintenanceFile 就是一个过程性内聚模块。

过程性内聚明显比时间性内聚要好——至少操作之间在过程上相关。即使如此，这些操作之间仍然是弱连接的，模块仍然不能在其他产品中复用。解决方法是，把具有过程性内聚的模块分成几个模块，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所遵循的步骤顺序相关的操作，而且所有操作都对相同的数据进行，那么这个模块具有通信性内聚（communicational cohesion）。例如，方法 updateRecordInDatabaseAndWrite/*t*ToTheAuditTrail 和方法 calculateNewTrajectoryAndSend/*t*ToThePrinter 是两个具有通信性内聚的模块。由于方法的操作之间的联系更密切，因此通信性内聚比过程性内聚要好，但它仍然与偶然性、逻辑性、时间性以及过程性内聚有同样的缺点，也就是说，方法不能复用。解决方法还是把一个模块分成多个独立模块，每个模块执行一个操作。

顺便提一下，Dan Berry［personal communication，1978］使用术语流程图内聚（flowchart cohesion）来描述时间性、过程性以及通信性内聚，因为这些模块的操作在产品流程图中是邻近的。由于时间性内聚中的操作同时执行，所以是邻近的。过程性内聚中算法要求操作按顺序依次执行，所以是邻近的。在通信性内聚中操作除依次执行外，还对相同的数据进行操作，因此自然在流程图中这些操作是邻近的。

7.2.6　功能性内聚

精确执行一个操作或完成单个目标的模块具有功能性内聚（functional cohesion）。例如，方法 getTemperatureOfFurnace、方法 computeOrbitalOfElectron、方法 writeToHardDrive 以及方法 calculateSalesCommission 都是具有功能性内聚的模块。

具有功能性内聚的模块通常能复用，因为一个频繁执行的操作在其他产品中也经常会需要。一个设计良好、经过详细测试并且有详细文档说明、具有功能性内聚的模块，对任何软件组织来说都是有价值（经济和技术上）的资产，应该尽可能多地复用（参见 8.4 节）。

对一个具有功能性内聚的模块进行维护将更容易。首先，功能性内聚可以隔离错误。如果确定没有正确读取炉子温度，那么错误基本可以肯定在方法 getTemperatureOfFurnace 中。类似地，如果没有正确地计算出电子轨道，那么首先查看的是方法 computeOrbitalOfElectron。

一旦在单个模块中定位了错误，下一步就是根据所需进行修改。因为具有功能性内聚的一个模块只执行一个操作，这样的模块比低内聚模块更容易理解。在理解上的简易也简化了维护工作。最终，当作出修改时，该修改影响其他模块的机率很小，特别是在模块之间耦合度很小的时候。

当一个产品需要扩展时，功能性内聚也具有价值。例如，假设一台个人计算机有一个 100G 的硬盘驱动器，但是制造商现在想销售更强大的具有 300G 硬盘驱动器的计算机。维护程序员通过读取模块列表，发现一个叫做 writeToHardDrive 的方法。显而易见，所要做的就是用新方法 writeToLargeHardDrive 来取代它。

顺便指出，图 7-2 中的三个模块具有功能性内聚，而 7.1 节中图 7-2 的设计超越图 7-3 的设计的论据，正好是前面讨论中为支持功能性内聚所给出的论据。

7.2.7　信息性内聚

如果模块执行很多操作，每个操作都有自己的入口点和独立的代码，并且都对相同的数据

结构执行，那么这个模块具有信息性内聚(informational cohesion)。图 7-6 中给出了一个例子。信息性内聚没有违反结构化编程的原则，每段代码正好有一个入口点和一个出口点。逻辑性内聚和信息性内聚的主要不同之处在于，具有逻辑性内聚的模块的操作之间会纠结，而具有信息性内聚的模块中，每个操作的代码都是完全独立的。

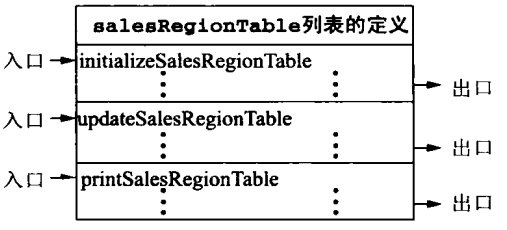


图 7-6　具有信息性内聚的模块

具有信息性内聚的模块本质上是用来实现一种抽象数据类型，如 7.5 节所述，使用一个具有信息性内聚的模块时，会得到使用抽象数据类型的所有优点。因为一个对象本质上就是一种抽象数据类型的实例（实例化）（见 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

为更加深入地领悟内聚，考虑如图 7-7 所示的例子。有两个方法值得特别讨论一下。当看到方法 initializeSumsAndOpenFiles（初始化和并打开文件）和方法 closeFilesAndPrintAverageTemperatures（关闭文件并打印平均温度）被标记为具有偶然性内聚而不是时间性内聚时，可能会有一些惊讶。首先，考虑 initializeSumsAndOpenFiles 方法，它执行两个与时间有关的操作，而这两个操作必须在执行任何计算之前完成，因此看起来它具有时间性内聚。尽管 initializeSumsAndOpenFiles 方法的两个操作确实在计算开始前被执行，但这里需要考虑另一个因素。初始化和与该问题相关，但是打开文件是一个与问题本身没有任何关系的硬件问题。当两个或更多不同层次的内聚能分配一个模块时，规则是分配给模块最低级别的内聚。因此，由于 initializeSumsAndOpenFiles 方法可以有时间性或偶然性内聚，两个内聚级别中较低级别的偶然性内聚将分配给该模块。同理，closeFilesAndPrintAverageTemperatures 方法具有偶然性内聚。

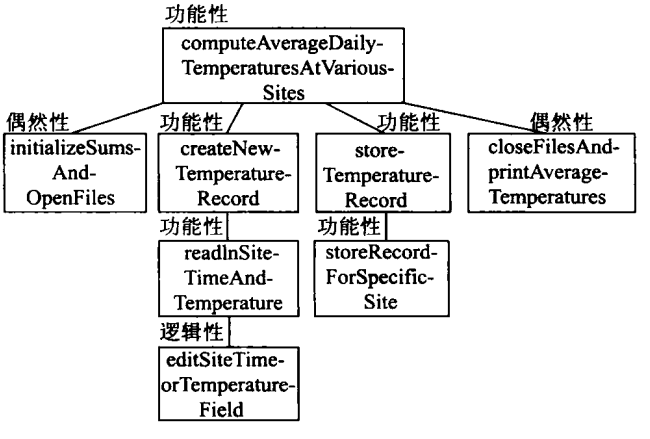


图 7-7　显示各个模块内聚性的模块互连图

7.3　耦合

前面介绍了内聚是指一个模块内部之间的交互程度；耦合则是两个模块间的交互程度。跟前面一样，耦合也有如图 7-8 所示的一些级别的区分。为了突出好的耦合，各个级别按照从差到好的顺序进行排列。

5. 数据耦合　(好)
4. 印记耦合
3. 控制耦合
2. 公共耦合
1. 内容耦合　(差)

图 7-8　耦合级别

7.3.1　内容耦合

如果两个模块中的一个模块直接引用另一个模块的内容，那么这两个模块之间是内容耦合

⊖ 在这一段讨论中，假设抽象数据类型或对象的设计良好。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

(content coupled)。下面是内容耦合的例子。

例 1　模块 p 修改模块 q 的一条语句。汇编语言不限制这种形式的编程。而 COBOL，现在也已经去掉了 **alter** 操作，alter 做的恰好是修改另一条语句。

例 2　模块 p 根据模块 q 内的数值位移来访问模块 q 的局部数据。

例 3　模块 p 跳转到模块 q 中的一个局部标签。

假设模块 p 和模块 q 是内容耦合的，危险之一是几乎任何对 q 的改变，即使是用一个新的编译器或汇编程序重新编译 q，都需要改变 p。进一步说，在新产品中不可能只复用模块 p 而不复用模块 q。当两个模块内容耦合时，它们无法避免相互间的连接。

7.3.2　公共耦合

如果两个模块存取相同的全局数据，那么这两个模块之间是公共耦合（common coupled），如图 7-9 所示。模块 cca 和 ccb 能存取和修改 globalVariable（全局变量）的值，而不是依靠传递参数来进行相互之间的通信。在通常情形下，当 cca 和 ccb 都存取相同的数据库并能读写相同的记录时会发生这种情况。对公共耦合来说，两个模块必须都能读写数据库，如果数据库的访问模式是只读，那么就不是公共耦合的。但是还有其他方法来实现公共耦合，包括使用 C++ 或 Java 修饰符 **public**。

这种形式的耦合不是我们所要的，原因如下：

1）生成代码不可读，这与结构化编程的思想相违背。考虑图 7-10 所示的代码段，如果 globalVariable 是一个全局变量，method3、method4 或这两个方法调用的任何方法都可能修改它的值。确定在什么条件下循环结束是一个重要问题，如果出现一个运行错误，由于那些模块中的任何一个都可能已经修改了 globalVariable 的值，这将很难去重现发生了什么。

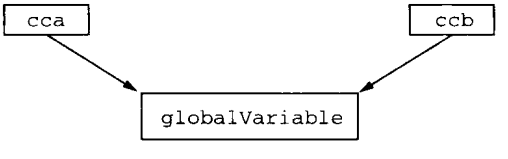


图 7-9　公共耦合的示例

```
while(globalVariable==0)
{
  if(xyz.argument>25)
    xyz.method3();
  else
    xyz.method4();
}
```

图 7-10　反映公共耦合的代码块

2）考虑调用 record7.editThisTransaction (changedData)。如果具有公共耦合，这个调用不仅可以修改 record7，而且可以修改该方法可以存取的任何全局变量。简而言之，必须读取整个方法才能准确地找出它做了什么。

3）如果模块中对一个全局变量的声明作了维护性修改，那么必须修改可以访问此全局变量的每个模块。进一步说，所有的改变必须是一致的。

4）另一个问题是难于复用一个公共耦合模块，因为每次模块复用时都必须提供同样的全局变量列表。

5）公共耦合有个令人遗憾的特性：即使模块 p 本身从不改变，模块 p 与产品中的其他模块之间的公共耦合的实例数量也会有非常大的变化，这用术语“秘密公共耦合”（Clandestine common coupling）[Schach et al.，2003a] 来描述。例如，如果模块 p 和模块 q 都能改变 globalVariable 的值，那么在模块 p 和软件产品的其他模块之间有一个公共耦合的实例。但是如果设计并实现的 10 个新模块都可以修改 globalVariable，那么即使模块 p 自身没有任何形式的改变，模块 p 和其他模块之间的公共耦合的数量会增加到 11 个。秘密公共耦合会产生令人惊讶的结果。例如，从 1993 ~ 2000 年有近 400 个 Linux 版本发布，17 个 Linux 内核模块的 5 332 个版

本在后续的发布期间没有改变。即使内核模块自身没有改变，5 332 个版本中半数以上每一个内核模块和其他 Linux 模块之间公共耦合的实例数量会增加或减少。值得考虑的是，模块显示秘密公共耦合数显上升方向（2 482）而非下降（379）[Schach et al.，2003a]。Linux 核心的代码行数随着版本号线性增长，但是公共耦合的实例数量则呈指数级增长 [Schach et al.，2002]。由秘密公共耦合所导致的模块间依赖，似乎不可避免地将使 Linux 难于维护。只改变 Linux 某一部分而不在产品的其他部分产生回归错误（一个显然不相关的错误）将非常困难。

6）这个问题的潜在危险可能最大：作为公共耦合产生的结果，模块可能暴露比需要更多的数据。这将导致无法控制数据存取，并且最终可能导致计算机犯罪。许多类型的计算机犯罪需要某种形式的共谋。设计良好的软件不应该允许任何程序员访问实行犯罪所需的所有数据和模块。例如，一个编写工资报表产品的校对打印部分的程序员需要访问员工记录，但是在一个设计良好的产品中，这种访问是只读模式下的排他访问，以免程序员对他（她）的工资做非授权改动。程序员为了做这种改动，就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记录。但是如果产品设计很差，每个模块都能在更新模式下访问工资报表数据库，那么一个不道德的程序员就可以独自对数据库中任何记录做非授权修改。

尽管我们希望上述论述能劝阻大多数读者（除了最大胆的读者外）避免使用公共耦合，但在某些情形下，公共耦合看起来似乎是可取的。例如，考虑执行储油罐计算机辅助设计的产品 [Schach and Stevens-Guille，1979]。可以用大量描述符（如高度、直径、容器可能遭遇的最大风速以及绝缘厚度）来详细描述一个储油罐。必须初始化这些描述符且其值以后不会再修改，产品中的大部分模块需要访问它们的值。假如有 55 个储油罐描述符。如果所有这些描述符作为参数传递给每个模块，那么每个模块的接口将至少包括 55 个参数，潜在出错的可能性将非常大。即使在像 Ada95 [ISO/IEC8652，1995] 这样对参数进行严格类型检查的面向对象语言中，两个同类型的参数仍可能被互换，而类型检查器将检测不到这样的错误。

一种解决方案是，把所有储油罐描述符放在数据库中，并且按以下方式来设计产品：一个模块初始化所有描述符的值，而所有其他模块都在只读状态下对数据库进行排他访问。然而，如果数据库解决方案不切实际，也许因为特定的实现语言不能与数据库管理系统进行接口，那么一种选择是在可控方式下使用公共耦合。也就是说，产品应该设计为一个模块初始化 55 个描述符，而其他任何模块都不能改变描述符的值。不像数据库解决方案中靠软件来加强控制，这种编程风格必须严格管理。因此，在没有更好选择的情形下使用公共耦合，通过管理的密切监督能降低一些风险。然而，一种更好的办法是用信息隐藏来避免公共耦合，见 7.6 节。

7.3.3　控制耦合

如果两个模块中一个模块传递控制元素给另一个模块，则这两个模块具有控制耦合（control coupled），即一个模块明确地控制另一个模块的逻辑。例如，当传递一段函数代码给具有逻辑性内聚的模块时，就是传递控制（见 7.2.2 节）。另一个控制耦合的例子是把一个控制开关作为参数传递时的情形。

如果模块 p 调用模块 q 并且 q 给 p 传回一个标志："我不能完成我的工作"，那么就是 q 在传递数据。但是如果标志是"我不能完成我的工作，相应地，显示出错消息 ABC123"，那么 p 和 q 是控制耦合的。换句话说，如果 q 给 p 传递回信息，并且 p 决定接收该信息后将采取什么行为，那么 q 正在传递数据。但是，如果 q 不仅给 p 传递回信息，同时通知模块 p 必须采取什么操作，那么这是控制耦合。

控制耦合产生的主要难点是两个模块不是独立的，被调用的模块 q 必须知道模块 p 的内部结构和逻辑。因此降低了复用的可能性。另外，控制耦合通常与具有逻辑性内聚的模块关联，所以它也有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，只有如 partNumber、satelliteAltitude 或 degreeOfMultiprogramming 这样的简单变量能作为参数传递。但是许多语言还支持将数据结构作为参数传递，例如，把记录或数组作为参数。在这些语言中，有效的参数包括 partRecord、satelliteCoordinates 或 segmentTable。如果把数据结构作为参数传递，则两个模块是印记耦合(stamp coupled)，但被调用模块仅对该数据结构一些个别组件进行操作。

例如，考虑消息 employeeRecord. calculateWithholding()。不了解整个 calculateWithholding 方法就不会清楚这个方法访问或修改的是 emplyeeRecord 的哪些字段。传递员工的工资显然对计算工资扣除是必须的，但是很难明白实现这个目的为什么需要员工的家庭电话号码。应该只传递那些计算扣除工资真正需要的字段给 calculateWithholding 方法。容易理解的是，不仅是得到的方法，特别是它的接口，都可能在需要计算工资扣除的各种产品中复用。(另一种观点参见备忘录7.2。)

备忘录7.2

传递4个或5个字段给一个模块可能比传递一个完整的记录要慢。这种情形导致了一个更大的问题：当优化问题（如响应时间或空间限制）与通常认为是好的软件工程实践相冲突时，应该怎么做？

根据个人经验，这个问题通常是无关紧要的。推荐的方法可能会降低响应时间，但是仅仅相差1毫秒左右，用户根本就不会觉察到。因此，按照 Knuth [1974] 优化第一定律：不做！几乎没有必要做任何形式的优化，包括性能上的原因。

但是如果确实需要最优化呢？在这种情况下，可应用 Knuth 优化第二定律。第二定律（仅适用于专家）是：仍然不做！换句话说，首先使用合适的软件工程技术完成整个产品。然后如果确实需要优化，只做必要的修改，认真记录修改了什么以及修改的原因。如果有可能，应该由一个资深软件工程师来完成优化。

可能更重要的是，由于上述调用 employeeRecord. calculateWithholding()所传递的数据比严格需要的数据多，对数据访问无法控制的问题和可以想像得出的计算机犯罪将再次出现。这个问题在7.3.2节中已经讨论了。

如果被调用模块使用了数据结构的全部组件，把数据结构作为参数传递根本就没有错误。例如，像 invertMatrix(originalMatrix,invertedMatrix)或 printInventoryRecord(warehouseRecord)调用中把数据结构作为一个参数传递，而被调用模块对数据结构的所有组件进行操作。当数据结构作为一个参数传递，而被调用模块只使用一部分组件时，就出现了印记耦合。

在类似C++这样的语言中，当指向记录的指针作为参数传递时，一个形式微妙的印记耦合就会产生。考虑调用 checkAltitude(pointerToPositionRecord)。初看起来，传递的是一个简单变量。但是被调用的模块访问了 pointerToPositionRecord 指向的 positionRecord 中的所有字段。因为这个潜在的问题，无论何时把指针作为参数传递，仔细检查该耦合都会是一个好主意。

7.3.5 数据耦合

当所有参数是同类的数据项时，两个模块是数据耦合(data coupled)。也就是说，每个参数或者是简单参数，或者是数据结构，其中被调用模块使用了该数据结构的所有元素。例如，dispalyTimeOfArrival(flightNumber)、computeProduct(firstNumber,secondNumber, result)以及 determineJobWithHighestPriority (jobQueue)。

数据耦合是一个理想的目标。从消极的方面看，如果一个产品只显示数据耦合，那么内容

耦合、公共耦合、控制耦合以及印记耦合中的难点就不会出现。从积极的方面来看，如果两个模块是数据耦合的，那么维护会更容易，因为对一个模块中的修改引起另一个模块发生回归错误的可能性更小。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。连线的数字代表接口，它们在图7-12中有更详细的定义。例如，当模块p调用模块q（接口1）时传递一个参数：飞机的类型。当q返回控制给p时，它传回一个状态标志。使用图7-11和图7-12中的信息，可以推导出每对模块间的耦合。结果显示见图7-13。

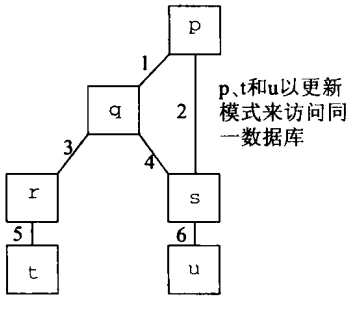


图7-11　耦合示例中的模块连接图

图7-13中的一些项是显然的。例如，p和q之间（图7-11中接口1）的数据耦合、r和t之间的数据耦合（接口5）以及s和u之间（接口6）的数据耦合，是向各个方向传递一个简单变量的直接结果。如果使用或更新从p传递到s的部分列表的所有元素，那么p和s之间的耦合（接口2）是数据耦合，但是如果s仅对列表的某些元素进行操作，那么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。因为在图7-11和图7-12中的信息不能完全描述各模块的功能，所以没有办法确定该耦合是数据耦合还是印记耦合。因为从q传递到r一个功能代码，所以在模块q和r之间的耦合（接口3）是控制耦合。

编号	输入	输出
1	aircraftType	statusFlag
2	listOfAircraftParts	—
3	functionCode	—
4	listOfAircraftParts	—
5	partNumber	partManufacturer
6	partNumber	partName

图7-12　图7-11的接口描述

	q	r	s	t	u
p	数据	—	{数据或者 印记	公共	公共
q		控制	{数据或者 印记	—	—
r			—	数据	—
s				—	数据
t					公共

图7-13　图7-11中各模块对间的耦合

在图7-13中标明有三个公共耦合的项或许有些奇怪。在图7-11中最远的三个模块对（p和t、p和u以及t和u）开始似乎没有任何形式的耦合。毕竟，没有接口连接它们，因此需要说明的是为什么它们之间存在耦合，更不用说公共耦合了。答案就在图7-11右边注释中，就是p、t和u都在更新模式下访问相同的数据库。结果是三个模块都能修改一些全局变量，因此它们之间两两构成公共耦合。

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耦合紧密，那么对模块p的修改就需要对模块q作相应的修改。如果在集成或交付后维护期间按照要求作出修改，那么最终产品的功能将是正确的。然而，那个阶段的进展将比松耦合情况下慢。另一方面，如果当时没有对模块q作出相应修改，那么错误随后会自己显现出来。最好的情况下，编译器或链接器测试出模块p处的修改时，立即通知团队有地方出错，或将发生错误。然而，通常的情况是，产品在接下来的集成测试中出错，或者在产品已经安装到用户计算机后出错。在这两种情况下，错误都是在对模块p的修改已经完成后出现的。这时，对模块p的修改和被忽视的模块q的相应修改之间，不再有任何明显的联系。因此将很难发现错误。

有迹象表明，耦合越强（越不合需要），发生错误的倾向越大［Briand，Daly，Porter，and

Wüst，1998］。这一现象背后的主要原因是代码中的依赖性导致回归错误。进一步来说，如果一个模块有出错倾向，那么必须对它进行反复维护，并且这些频繁的修改可能会损害它的可维护性。而且，这些频繁的修改不总是局限于有错误倾向的模块本身，通常为了修复一个错误需要修改多个模块。从而，一个模块的出错倾向会对一些其他模块的可维护性产生负面影响。换句话说，有理由相信强耦合对可维护性有负面的影响［Yu，Schach，Chen，and Offutt，2004］。

如果高内聚低耦合模块的设计是一个好的设计，一个显然的问题是，如何实现这样一个设计？由于本章重点讲述与设计相关的理论概念，第 12 章会给出问题的答案。同时，还会进一步讨论和精化良好的设计所具有的品质。为了方便起见，本章中的关键定义以及出现章节如图 7-14所示。

抽象数据类型：一种数据类型以及执行于该数据类型实例上的操作（7.5节）
抽象：通过抑制不必要的细节同时强调相关细节来达到逐步求精的一种方法（7.4.1节）
类：支持继承的一种数据抽象数据类型（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：一种数据结构以及在该数据结构上执行的操作（7.4节）
封装：现实世界实体的所有方面集中在一个对该实体进行建模的单元中（7.4.1节）
信息隐藏：构建一种设计，使其对其他模块隐藏了最终的实现细节（7.6节）
对象：类的一个实例（7.7节）

图 7-14　本章的关键定义以及所在节

7.4　数据封装

考虑为大型计算机设计操作系统的问题。按照规格说明，任何提交给计算机的工作会按高优先级、中优先级和低优先级进行分类。操作系统的任务是决定下一个载入内存的是哪个作业，内存中的哪个作业将得到下一个时间片以及时间片长度，哪个要求磁盘访问的作业优先级最高。在执行调度时，操作系统必须考虑每个作业的优先级，作业优先级越高，其获得计算机资源就越快。实现这一点的方式是为每个作业优先级维护独立的作业序列。必须初始化作业队列，当一个作业需要内存、CPU 时间或磁盘访问时，必须可以方便地添加一个作业到作业队列，当操作系统决定给一个作业分配所需资源时，也可以方便地从队列中删除该作业。

为了简化问题，把问题限制为批处理作业排队等候访问内存的问题。进来的批处理作业有三个队列，每个队列对应一个优先级。当用户提交后，添加一个作业到适当的队列；当操作系统决定运行一个就绪作业时，将该作业从队列中删除并且给它分配内存。

产品的这个部分可以用多种不同的方式来构建。一个可能的设计如图 7-15 所示，它描绘了用来处理三个作业队列中一个队列的模块。定义在模块 m1、m2 和 m3 中的操作是全局方法（C++ 中的静态成员函数，Java 中的静态方法或类方法）。

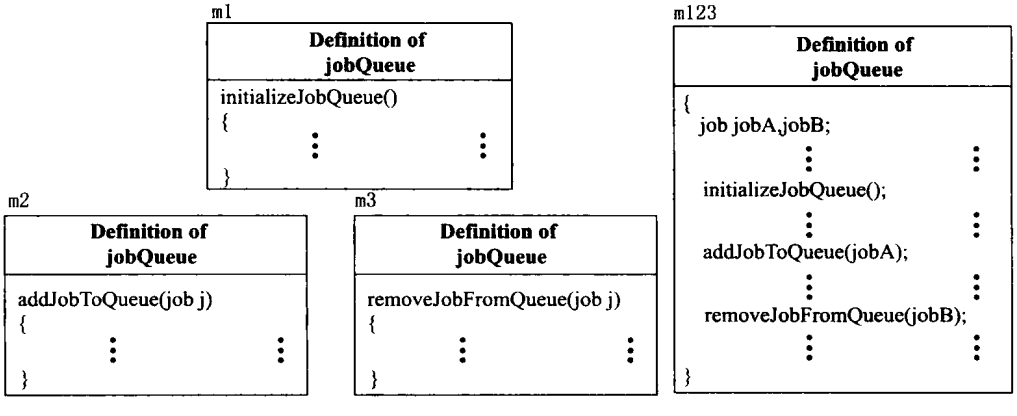


图 7-15　操作系统的作业队列部分的一种可能设计

考虑图7-15。模块m1中的方法initializeJobQueue负责初始化工作队列；模块m2和m3中的方法addJobToQueue和removeJobFromQueue分别负责增加和删除作业。模块m123包含所有三种方法的调用以便于处理作业队列。为了集中讨论数据封装，这里忽略了类似下溢（尝试从一个空队列中删除作业）和溢出（尝试向一个满队列中添加作业）的问题，本章的其余部分也作相同处理。

在图7-15中设计的模块具有低内聚性，因为对作业队列的操作遍布整个产品。如果决定修改jobQueue的实现方式（例如，以记录的链表来代替线性表），那么必须彻底修改模块m1、m2和m3。还必须修改模块m123，至少要修改数据结构定义。

现在假设选择图7-16的设计替代图7-15的设计。在图右边的模块具有信息性内聚（见7.2.7节），因为它对同一数据结构执行多个操作。每个操作都有自己的入口点、出口点和独立代码。图7-16中的模块mEncapsulation是数据封装的一种实现，也就是说，在这种作业队列情况下，一个数据结构与在该数据结构执行的操作。

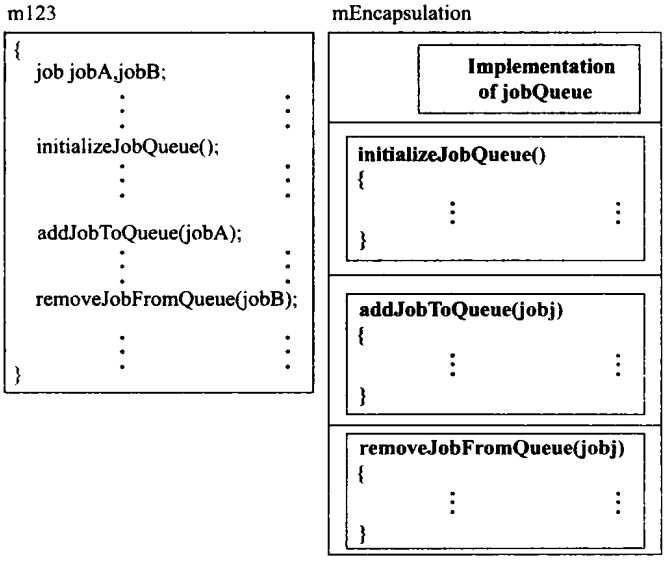


图7-16　使用数据封装的操作系统的作业队列部分的设计

一个明显的问题是：使用数据封装设计产品的优点是什么？可以以两种方式回答：一是从开发的角度回答，另一是从维护的角度回答。

7.4.1　数据封装和开发

数据封装是*抽象*（abstraction）的一个例子。回到作业队列的例子，已经定义了一个数据结构（作业队列）与三个相关的操作（初始化作业队列、添加作业到队列、从队列中删除作业）。开发者能在更高层次（作业级别和作业队列层次）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。

抽象后的基本理论概念仍然需要逐步求精。首先，产品的设计基于高层次概念，例如，作业、作业队列和对作业队列执行的操作。在这一阶段，如何实现作业队列是无关紧要的。一旦得到完整的高层次设计，第二步就是根据数据结构以及在数据结构上执行的操作来设计低层次的组件。例如，在C++中数据结构（作业队列）按照记录（结构）或数组来实现，三种操作（初始化作业队列、添加一个作业到队列以及从队列中删除一个作业）以方法的形式实现。这里关键是在低层次设计中，设计者完全忽略作业、作业队列以及操作的设计用途。因此，在第一步中，假设低层次已经存在，即使在这一阶段还没有关于低层次的任何思考；在第二步（低层次的设计）中，忽略高层次的存在。在高层次中，关注的是数据结构（即作业队列）的行为；在低层次中，主要考虑的则是行为的实现。当然，一个更大型的产品将有许多抽象层次。

存在着不同类型的抽象。考虑图7-16，图中有两种类型的抽象。数据封装（也就是数据结构与在该数据结构上执行的操作）是*数据抽象*（data abstraction）的一个例子；方法本身是*过程抽象*（procedural abstraction）的一个例子。简而言之，*抽象*是通过抑制不必要的细节同时强调相关细节来达到逐步求精的一种方法。现在可以把*封装*（encapsulation）定义为把现实世界实体

的所有方面集中在一个对该实体进行建模的单元中，在1.9节中称其为概念独立。

数据抽象允许设计者在数据结构以及在其上进行的操作的层次上来考虑问题，随后才考虑如何实现数据结构和操作。那么过程抽象呢？考虑定义一个方法 initializeJobQueue 的结果。这么做的效果是通过给开发者提供另一个方法来扩展语言，而这个方法不是最初语言定义中的一部分。开发者能像使用 sqrt 或 abs 一样来使用 initializeJobQueue。

对设计来说，过程抽象与设计抽象一样意义重大。设计者能基于高层次操作构思产品。这些操作基于低层次操作来定义，直到达到最低层次。在最低层次，操作按照编程语言预定义的结构予以表达。在每一层次，设计者只考虑按照适合该层次的操作来表示产品。设计者可以忽略下面的层次，因为它们将在下一抽象层次得到处理，即在下一个求精步骤中得到处理。设计者也可以忽略上面的层次，因为从设计角度来说，上面的层次与当前层次并不相关。

7.4.2 数据封装和维护

从维护的角度考虑数据封装，一个基本问题是确定产品的哪些方面可能需要修改并设计产品，使将来修改产品的影响最小化。例如，如果一个产品包括作业队列，那么未来版本可能合并它们，像这样的数据结构不太可能会修改。同时，实现工作队列的特定方式可能会改变，数据封装提供了处理这种改变的一种方法。

图7-17描述了作业队列数据结构 **JobQueueClass** 在C++中的实现；图7-18是相应的Java实现。（备忘录7.3介绍了图7-17和图7-18以及本章中的后续代码例子的编程风格。）在图7-17或图7-18中，队列由一个最多25个作业号的数组来实现，第一个元素是queue[0]，第25个元素是queue[24]。每个作业号用一个整数来表示。保留字 **public** 允许 queueLength 和 queue 在操作系统中任何地方都是可见的。这样得到的公共耦合非常不实用，这将在7.6中得到改正。

备忘录7.3

这里特意采用突出数据抽象的方式，以牺牲好的编程习惯为代价，编写了图7-17和图7-18以及本章后面的代码例子。例如，在图7-17和图7-18中 **JobQueueClass** 的定义里的数字25，应该作为参数编码，也就是说，在C++中作为一个 **const** 或者在Java中作为一个 **public static final** 变量。而且为了简便起见，这里忽略了诸如下溢（尝试从一个空队列中移除一个项）或上溢（尝试向一个满队列中添加一个项）这些情况的检查。在任何实际产品中，包含这些检查是非常重要的。

另外，最小化了语言的特有属性。例如，一个C++程序员通常用

```
queueLength ++;
```

将 queueLength 的值增加1，而不是用

```
queueLength = queueLength + 1;
```

同样，尽量不用构造器和析构器。

总之，本章中的代码只是出于教学目的，不应该将这些代码用于其他目的。

因为在 **JobQueueClass** 中的方法属性为 **public**（公共的），所以在操作系统中任何地方都可以调用它们。特别是图7-19显示了在C++中，方法 queueHandler 是如何调用 **JobQueueClass** 的，图7-20是相应的Java实现。方法 queueHandler 不知道作业队列如何实现，但仍能调用 **JobQueueClass** 中的 initializeJobQueue 方法、addJobToQueue 方法和 removeJobFromQueue 方法。使用 **JobQueueClass** 所需要的信息仅仅是关于三个方法的接口信息。

现在假设当前作业队列是用基于作业号的线性表形式实现的，但之后决定用作业记录的双向链表来重新实现作业队列。每个作业记录将有三个成分：如前所述的作业号、指向该作业之

前的作业记录的指针、指向其后作业的作业记录的指针。图 7-21 给出了基于 C++ 的代码实现，图 7-22 给出了基于 Java 的代码实现。由于作业队列实现方式发生了改变，那么整个软件需要做出什么样的调整呢？事实上，只有 **JobQueueClass** 本身需要修改。图 7-23 给出了使用图 7-21 所示的双向链表的 **JobQueueClass** 的 C++ 实现框架。为了强调 **JobQueueClass** 与产品中其他部分（包括 queueHandler 方法）的接口没有改变（参见习题 7.11），忽略了 JobQueueClass 的 C++ 实现细节。也就是说，调用三个方法 initializeJobQueue、addJobToQueue 和 removeJobFromQueue 的方式没有改变。需要明确的是，即使作业队列本身的实现方式完全不同，当调用 addJobToQueue 方法时，它仍传递一个整型值；removeJobFromQueue 方法仍旧返回一个整型值。因此，queueHandler 方法的源代码（图 7-19）完全不需要改变。相应地，数据封装通过简化维护、降低回归错误发生概率的方式来支持数据抽象的实现。

```
//
//警告：
//这里给出的代码是给那些不是C++专家的读者阅读的，为简单起见，程序段
//略去了一些重要的语句，如上溢和下溢的检查。细节请参阅备忘录7.3。
class JobQueueClass
{
  //属性
  public:
    int queueLength;          //作业队列的长度
    int queue[25];            //队列可最多包含25个作业

  //方法
  public:
    void initializeJobQueue ()
    /*
     * 空作业队列的长度为0
     */
    {
       queueLength = 0;
    }

    void addJobToQueue (int jobNumber)
    /*
     * 将作业添加到作业队列的尾部
     */
    {
       queue[queueLength] = jobNumber;
       queueLength = queueLength + 1;
    }

    int removeJobFromQueue ()
    /*
     * 令jobNumber等于存储在队列中的作业数，将队列头部的作业移去，
     * 移动剩余的作业并返回jobNumber的值
     */
    {
       int jobNumber = queue[0];
       queueLength = queueLength - 1;
       for (int k = 0; k < queueLength; k++)
          queue[k] = queue[k + 1];
       return jobNumber;
    }
}// class JobQueueClass
```

图 7-17　**JobQueueClass** 的 C++ 实现（由 **public** 属性引起的问题将在 7.7 节解决）

```
//
//警告：
//这里给出的代码是给那些不是C++专家的读者阅读的，为简单起见，程序段
//略去了一些重要的语句，如上溢和下溢的检查。细节请参阅备忘录7.3。
//
class JobQueueClass
{
  //属性
  public int     queueLength;              //作业队列的长度
  public int     queue[ ] = new int[25];   //队列可最多包含25个作业

  //方法
  public void initializeJobQueue ()
  /*
   * 空作业队列的长度为0
   */
  {
     queueLength = 0;
  }
  public void addJobToQueue (int jobNumber)
  /*
   * 将作业添加到作业队列的尾部
   */
  {
     queue[queueLength] = jobNumber;
     queueLength = queueLength + 1;
  }
  public int removeJobFromQueue ()
  /*
   * 令jobNumber等于存储在队列中的作业数，将队列头部的作业移去，
   * 移动剩余的作业并返回jobNumber的值
   */
  {
     int jobNumber = queue[0];
     queueLength = queueLength - 1;
     for (int k = 0; k < queueLength; k++)
        queue[k] = queue[k + 1];
     return jobNumber;
  }
}// class JobQueueClass
```

图 7-18　**JobQueueClass** 的 Java 实现（由 **public** 属性引起的问题将在 7.7 节解决）

```
class SchedulerClass
{
  ...
  public:
    void queueHandler ()
    {
      int                jobA, jobB;
      JobQueueClass      jobQueueJ;

        //若干语句
      jobQueueJ.initializeJobQueue ();
        //更多语句
      jobQueueJ.addJobToQueue (jobA);
        //其他更多的语句
      jobB = jobQueueJ.removeJobFromQueue ();
        //进一步的语句
    }// queueHandler
  ...
}// class SchedulerClass
```

图 7-19　queueHandler 的 C++ 实现

```
class SchedulerClass
{
  ...
  public void queueHandler ()
  {
      int                jobA, jobB;
      JobQueueClass      jobQueueJ = new JobQueueClass ();

        //若干语句
      jobQueueJ.initializeJobQueue ();
        //更多语句
      jobQueueJ.addJobToQueue (jobA);
        //其他更多的语句
      jobB = jobQueueJ.removeJobFromQueue ();
        //进一步的语句
  }// queueHandler
  ...
}// class SchedulerClass
```

图 7-20　queueHandler 的 Java 实现

```
class JobRecordClass
{
   public:
        int                 jobNo;      //作业数（整数）
        JobRecordClass      *inFront;   //指向作业队列头部的指针
        JobRecordClass      *inRear;    //指向作业队列尾部的指针
}// class JobRecordClass
```

图 7-21　双向链接类 JobRecordClass 的 C++ 实现（由 public 属性引起的问题将在 7.6 节解决）

```
class JobRecordClass
{
   public int                  jobNo;      //作业数（整数）
   public JobRecordClass       inFront;    //对作业队列头部的引用
   public JobRecordClass       inRear;     //对作业队列尾部的引用
} // class JobRecordClass
```

图 7-22　双向链接类 JobRecordClass 的 Java 实现（由 public 属性引起的问题将在 7.6 节解决）

```
class JobQueueClass
{
  public:
    JobRecordClass      *frontOfQueue;      //指向作业队列头部的指针
    JobRecordClass      *rearOfQueue;       //指向作业队列尾部的指针

    void initializeJobQueue ()
    {
      /*
       * 通过将frontOfQueue和rearOfQueue设置为NULL来对作业队列进行初始化
       */
    }

    void addJobToQueue (int jobNumber)
    {
      /*
       * 创建一个新的作业记录
       * 将jobNumber值放置在新记录的jobNo域上
       * 将新记录的inFront域设置为指向当前rearOfQueue所指向的记录
       * （由此将新的记录链接到队列的尾部）
       * 将新记录的inRear域设置为NULL
       * 将当前rearOfQueue所指向的记录的inRear域设置为指向新记录
       *（由此设置了双向链接），最后，令rearOfQueue指向新记录
       */
    }

    int removeJobFromQueue ()
    {
      /*
       * 设置jobNumber等于队列头部记录的jobNo域
       * 更新frontOfQueue指向队列中的下一条
       * 设置记录的inFront域，此时这条记录在队列的头部，为NULL
       * 并返回jobNumber
       */
    }
}// class JobQueueClass
```

图 7-23　使用双向链表的 JobQueueClass 的 C++ 实现框架

对比图 7-17、图 7-18 和图 7-19、图 7-20，显然在这些实例中，C++ 和 Java 实现的区别主要是语法的不同。本章剩余部分将只给出一种实现，并说明在另一种实现中语法上的差异。特别需要指出的是，作业队列代码的剩余部分是基于 C++ 实现的，其他所有代码例子都基于 Java 实现。

7.5　抽象数据类型

图 7-17（等价于图 7-18）是一个作业队列类（class）的实现，即一个数据类型以及在该数据类型实例上所进行的操作。这样的一个构造叫做*抽象数据类型*（abstract data type）。

```
class SchedulerClass
{
  ...
  public:
    void queueHandler ()
    {
      int              job1, job2;
      JobQueueClass    highPriorityQueue;
      JobQueueClass    mediumPriorityQueue;
      JobQueueClass    lowPriorityQueue;

        // 一些语句
      highPriorityQueue.initializeJobQueue ();
        // 更多语句
      mediumPriorityQueue.addJobToQueue (job1);
        // 其他更多的语句
      job2 = lowPriorityQueue.removeJobFromQueue ();
        // 更多的一些语句
    }// queueHandler
  ...
}// class SchedulerClass
```

图 7-24　使用图 7-17 的抽象数据类型来实现的 C++ 方法 queueHandler

图 7-24 显示了如何在 C++ 语言中使用这种抽象数据类型实现操作系统的三个作业队类。三个作业队列的具体例子是：highPriorityQueue、mediumPriorityQueue 和 lowPriorityQueue。（Java 版本仅在三个作业队列的数据声明的语法上有所不同。）语句 highPriorityQueue. initializeJobQueue() 表示“把 initializeJobQueue 方法应用到数据结构 highPriorityQueue 中”，其余两个语句与之类似。

抽象数据类型是一种应用广泛的设计工具。例如，假设要开发的一个产品中，需要对有理数进行大量操作，有理数就是形为 n/d 的数，这里 n 和 d 都是整数，d≠0。有理数的表示方式有很多种，比如使用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来实现有理数，可以为这个数据结构选择一个合适的表示。在 Java 中，可以由如图 7-25 所示那样定义，定义中包含有理数上的各种操作，例如，两个整数构造一个有理数、两个有理数相加、两个有理数相乘等。（由 **public** 属性所引起的问题，比如图 7-25 中 numerator、denominator 所引起的问题，将在 7.6 节中解决。）对应 C++ 实现的不同之处是保留字 **public** 的位置不同。还有当传递一个参数的引用时需要“&”符号。

```
class RationalClass
{
  public int          numerator;
  public int          denominator;

  public void sameDenominator (RationalClass r, RationalClass s)
  {
    // 将r和s约简为同一分母的代码
  }

  public boolean equal (RationalClass t, RationalClass u)
  {
    RationalClass     v, w;
    v = t;
    w = u;
    sameDenominator (v, w);
    return (v.numerator == w.numerator);
  }

  //对两个有理数进行加、减、乘、除的方法

}// class RationalClass
```

图 7-25　有理数的 Java 抽象数据类型的实现（由 **public** 属性引起的问题将在 7.6 节解决）

抽象数据类型同时支持数据抽象和过程抽象（见 7.4.1 节）。另外，当修改产品时，修改抽象数据类型的可能性不大；最坏情况下，需要增加额外的操作到抽象数据类型中。因此，从开发和维护的角度来看，抽象数据类型对软件制造者是很有吸引力的一个工具。

7.6 信息隐藏

在7.4.1节中讨论的两种抽象类型（数据抽象和过程抽象）是由Parnas提出的更通用的设计概念——信息隐藏（information hiding）的例子[Parnas，1971，1972a，1972b]。Parnas的想法是直接面向将来的维护。在设计产品前，应该列出一个清单记录将来可能发生修改的实现策略。然后，设计模块，这样最终设计的实现细节对其他模块来说是隐藏起来的。结果，每个未来的修改都可以定位在一个特定模块。因为最初实现决策的细节对其他模块是不可见的，所以改变设计显然不会影响其他模块。（阅读备忘录7.4以进一步了解信息隐藏。）

要了解这种思想在实际中如何应用，考虑图7-24，它使用了图7-17的抽象数据类型的实现。使用抽象数据类型的一个主要原因是，确保只有调用图7-17中三个方法之一时，才能修改作业队列的内容。遗憾的是，这种实现本身就可以通过其他方式修改作业队列。图7-17中的属性queueLength和queue均声明为**public**，因此可以在queueHandler内部访问它们。结果如图7-24所示，在queueHandler中的任何地方，只要是合法的C++（或Java）赋值语句，例如：

```
highPriorityQueue.queue[7] = -5678;
```

都可以修改highPriorityQueue。换句话说，不使用三种抽象数据类型操作中的任何一种也能修改作业队列的内容。另外，这暗示了，这种情况可能与降低内聚和提高耦合有关，管理者必须意识到该产品可能如7.3.2节中所述的那样，容易受到计算机犯罪的攻击。

幸运的是，有一种方法能够解决上面的问题。C++和Java的设计者在类的规格说明中提供了信息隐藏，基于C++的代码参见图7-26（Java语法的不同处与以前相同）。除了修改了属性修饰语的可见性——将属性由**public**改为**private**，图7-26与图7-17是一样的。现在对其他模块唯一可见的信息是**JobQueueClass**类以及可以对生成的作业队列进行操作的具有特定接口的三个操作。但是作业队列实现的真正方式是**private**，即对外部来说是不可见的。图7-27显示了一个具有**private**属性的类如何使一个C++或Java用户实现完全信息隐藏的抽象数据类型。

```
class JobQueueClass
{
  //属性
  private:
    int    queueLength;     //作业队列的长度
    int    queue[25];       //队列可最多包含25个作业

  //方法
  public:
    void initializeJobQueue ()
    {
      //方法体，同图7-17
    }

    void addJobToQueue (int jobNumber)
    {
      //方法体，同图7-17
    }

    int removeJobFromQueue ()
    {
      //方法体，同图7-17
    }
}// class JobQueueClass
```

图7-26　具有信息隐藏的C++抽象数据类型的实现，修正了图7-17、图7-18、图7-21、图7-22和图7-25中的问题

信息隐藏技术也能用来避免在7.3.2节末尾提到的公共耦合。再次考虑在7.3.2节所描述的产品，用55个描述符对一个储油罐的计算机辅助设计工具进行规范。如果该产品的实现方式是用**private**操作来初始化描述符，用**public**操作来获取一个描述符的值，那么将没有公共耦合。就如7.7节中所述的那样，因为对象支持信息隐藏，所以这种解决方案具有面向对象范型的特征。这是对象技术的另一个优点。

备忘录7.4

信息隐藏这一术语有些用词不当。更准确的描述应为“细节隐藏”，因为隐藏的不是信息而是实现的细节。

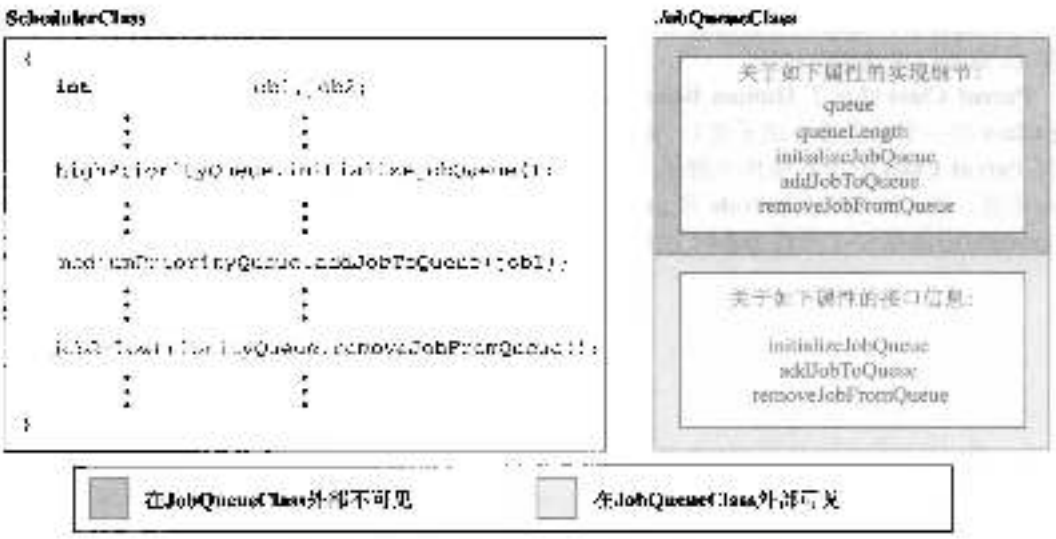


图7-27 具有信息隐藏的抽象数据类型的表达。信息隐藏通过 private 属性来实现（表7-26和表7-24）

7.7 对象

如本章开始所述，对象仅是图7-28中所示发展的下一步。对象没有什么特殊性，它们与抽象数据类型或具有信息性内聚的模块一样平常。对象的重要性在于其具有图7-28中它们前面所有的一切特性，还具有本身的一些额外特性。



图7-28 第7章的主要概念及其出现章节

对象的一个不完全定义是：对象是抽象数据类型的实例。也就是说，产品按照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。但是定义对象为抽象数据类型的实例远远不够，还需要的更多的东西，即继承（inheritance）。这一概念最早是在 Simula 67 中介绍的［Dahl and Nygaard, 1966］。所有面向对象编程语言都支持继承，例如，Smalltalk［Goldberg and Robson, 1989］、C++［Stroustrup, 2003］和 Java［Flanagan, 2005］。继承背后的基本思想是新的数据类型可以定义为先前定义类型的扩充，而不是从零开始定义［Meyer, 1986］。

在面向对象语言中，把类定义为一个支持继承的抽象数据类型，对象就是类的实例。考虑下面的例子，定义 **Human Being Class** 为一个类，Joe 为对象，是该类的一个实例。每个 **Human Being Class** 的实例具有某些属性（如年龄、身高以及描述对象 Joe 时赋给这些属性的值）。现在假设 **Parent Class** 定义为 **Human Being Class** 的子类（subclass，或派生类）。这就意味着 **Parent Class** 的实例具有 **Human Being Class** 的实例的所有属性，并且还可能具有其自身的属性（如孩子的姓名和孩子的个数），这在图7-29中进行了描述。用面向对象术语来说，Parent is a Human Being。这就是为什么图7-29中的箭头好像标错了方向。事实上，那个箭头描述表示 is A 关系，因此从派生类指向基类。使用空心箭头表示继承是 UML 的一种惯例；另一个惯例是类名字表示为首字

母大写的黑体字。最后，带折角的空心矩形是 UML 的注释（note）。UML 将在第二部分特别是在第 15 章进行更详细的讨论。

Parent Class 继承了 **Human Being Class** 的所有属性，因为 **Parent Class** 是基类 **Human Being Class** 的一个派生类（或子类）。如果 Fred 是 **Parent Class** 的一个对象（实例），那么 Fred 拥有 **Parent Class** 的所有属性并继承了 **Human Being Class** 的实例的所有属性。图 7-30 给出了 Java 实现，C++ 的实现在 **private** 和 **public** 修饰符的位置上有所不同。还有这个例子中，Java 中的 **extends** 语法在 C++ 中用 **public** 代替。

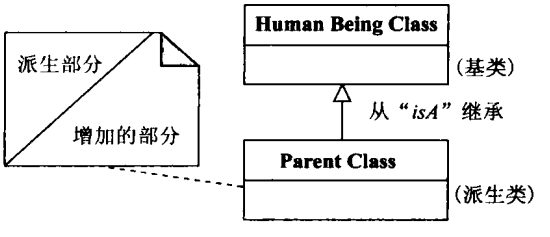


图 7-29　表示派生类型和继承的 UML 图

```
class HumanBeingClass
{
   private int          age;
   private float        height;

   //public HumanBeingClass上的操作的声明

}// class HumanBeingClass


class ParentClass extends HumanBeingClass
{
   private String       nameOfOldestChild;
   private int          numberOfChildren;

   //public ParontClass上的操作的声明

}// class ParentClass
```

图 7-30　图 7-29 的 Java 实现

继承性是所有面向对象编程语言的重要特性，但是，C 或 Lisp 之类的传统语言都不支持继承和类的概念。因此，面向对象范型不能用这些语言直接实现。

在面向对象范型的术语中，还有另外两种方式来看待图 7-29 中 **Parent Class** 和 **Human Being Class** 之间的关系。可以说 **Parent Class** 是 **Human Being Class** 的特化（specialization），或者说 **Human Being Class** 是 **Parent Class** 的泛化（generalization）。除了特化和泛化，类还有另外两个基本关系［Blaha，Premerlani，and Rumbaugh，1988］：聚集和关联。聚集（Aggregation）是指类的组件。例如，类 **Personal Computer Class** 可能由组件 **CPU Class**、**Monitor Class**、**Keyboard Class** 和 **Printer Class** 组成。如图 7-31 所示，使用菱形来标注聚集是 UML 中另一个惯例。聚集用于组合相关项，产生一个可复用的类（见 8.1 节。）

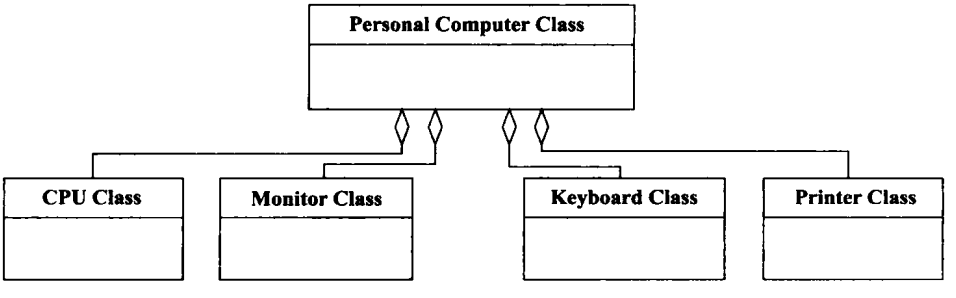


图 7-31　UML 聚集范例

关联（Association）是指两个明显不相关的类之间的某种关系。例如，放射学家与律师之间看起来没有任何关系，但是放射学家可能向律师咨询有关出租 MRI 设备合同方面的意见。图

7-32 中用 UML 给出了关联的图示。在这个实例中关联的本质可以用咨询（consult）一词来说明。另外，实心的三角形（在 UML 中称为导向三角形（navigation triangle））表示关联的方向，毕竟，律师脚踝骨折时也会向放射学家咨询。

顺便提一下，同其他面向对象语言一样，Java 和 C++ 符号表示明确地反应了操作与数据的等同性。首先，考虑支持记录的传统语言（如 C）。假设 record_ 1[⊖] 是一个 **struct**（记录），field_ 2 是类中的一个字段，那么字段可以表示为 record_ 1. field_ 2。也就是说，句点“.”表示记录内部的成员关系。如果说 function_ 3 是 C 模块内部的函数，那么 function_3() 表示了该函数的一个调用。

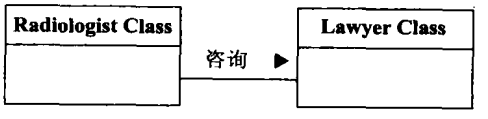


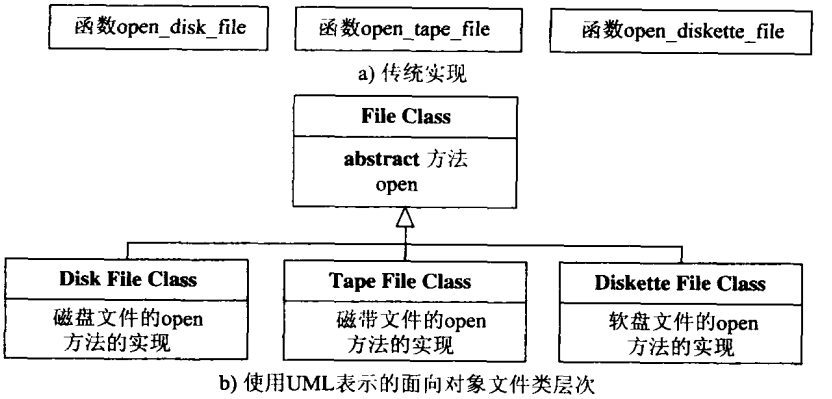
图 7-32　UML 关联范例

相反，假设 **AClass** 是一个有属性 attributeB 和方法 methodC 的类。进一步假设 ourObject 是 **AClass** 的一个实例。字段指的是 ourObject. attributeB。而且 ourObject. methodC() 表示对该方法的调用。因此，不管成员是属性还是方法，句点都用来表示对象中的成员关系。

使用对象（或类）的好处就是使用抽象数据类型的好处，包括数据抽象和过程抽象。另外，类的继承性提供更深层次的数据抽象，这会使产品开发更容易、更少出错。而另一个好处来自于将继承与多态性和动态绑定相结合，这是本书 7.8 节的主题。

7.8　继承、多态和动态绑定

假设调用计算机操作系统打开一个文件。该文件可能存储在许多不同介质上，例如，它可能是一个硬盘文件、一个磁带文件或一个软盘文件。使用传统范型，将会有三个不同名称的函数，如图 7-33a 所示：open_ disk_ file、open_ tape_ file 以及 open_ diskette_ file。如果声明 my_ file 是一个文件，那么在运行时，有必要测试它是一个硬盘文件、磁带文件还是一个软盘文件，以确定应该调用哪个函数。对应的传统代码如图 7-34a 所示。



a) 传统实现

b) 使用UML表示的面向对象文件类层次

图 7-33　打开一个文件所需的操作

相反，当使用面向对象范型时，定义一个叫做 **File Class** 的类，它有三个派生类：**Disk File Class**、**Tape File Class** 以及 **Diskette File Class**，如图 7-33b 所示。其中，UML 用空心箭头来表示继承。

⊖ 本书中，传统软件产品中变量的名字是按照传统惯例书写的，即用下划线来隔开变量名称的各个部分。例如，this_ is_ a_ classical_ variable（这与面向对象的惯例——“用一个大写字母标识一个变量名称中新的部分的开始”（如 thisIsAnObjectOrientedVariable）不同）。

现在，假设 open 方法定义在父类 **File Class** 中，并由三个派生类继承。遗憾的是，这并不起作用，因为打开这三种不同的文件需要执行不同的操作。

解决方法如下。在父类 **File Class** 中声明一个虚方法 open；在 Java 中，这样的方法声明为 **abstract**；在 C++ 中，使用保留字 **virtual**。如图 7-33b 所示，每个派生类都有该方法的特定实现，并且赋予每个方法的名字都相同，即 open。又假设将 myFile 声明为一个文件。在运行时，发送消息

```
myFile.open ();
```

面向对象系统现在需要确定 myFile 是一个硬盘文件、磁带文件还是一个软盘文件，并且调用相应的 open。也就是说，系统在运行时才确定对象 myFile 是 **Disk File Class** 的实例、**Tape File Class** 的实例还是 **Diskette File Class** 的实例，然后自动调用正确的方法。因为这项工作必须在运行时（动态）完成，而不是在编译时（静态）完成，因此把对象与合适的方法关联起来的行为称为*动态绑定*（dynamic binding）。进一步说，因为 open 方法能应用于不同类的对象，所以把它称为*多态*（polymorphic），即表示“多种形态”。就像碳晶体可以表现为多种不同的形态（包括坚硬的钻石和柔软的石墨）一样，open 方法可以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，用两个冒号代替句点，这些方法可以表示为 DiskFileClass::open、TapeFileClass::open 和 DisketteFileClass::open）。但是正是由于动态绑定，才不必确定调用哪个方法打开一个特定文件。如图 7-34b 所示，取而代之的是，需要在运行时发送消息 myFile.open()，然后由系统确定 myFile 的类型（类）并调用正确的方法。

这些思想不仅适用于 **abstract**（**virtual**）方法。考虑图 7-35 中类的层次。通过继承 **Base** 类来派生所有子类。假设方法 checkOrder（b: **Base**）把类 **Base** 的一个实例作为一个参数。那么，由于继承、多态以及动态绑定的特性，调用 checkOrder 时，checkOrder 的参数不仅可以是 Base 类还可以是 **Base** 类的任何子类，即 **Base** 的任意派生类以上调用都是正确的。所需要的只是调用 checkOrder，在运行时才对所有的事情进行处理。这种技术非常强大，因为在发送消息时，软件工程师不需要关注参数的准确类型。

```
switch (file_type)
{
    case 1:
       open_disk_file ( );          // file_type 1  对应硬盘文件
       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) 相应于图7-33a,打开文件的传统代码

```
myFile.open ( );
```

b) 相应于图7-33b,打开文件的面向对象代码

图 7-34　打开文件的代码

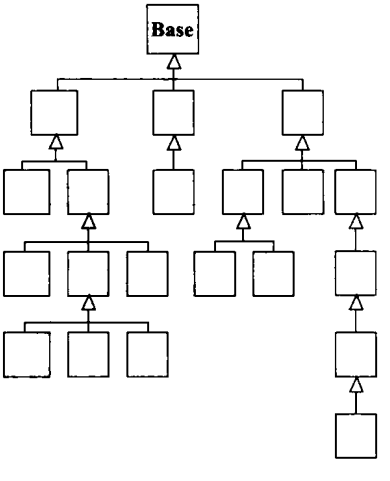


图 7-35　类的层次

然而，多态和动态绑定也有很大的缺点。

1）通常不太可能在编译阶段确定在运行时会调用哪个特定的多态方法。相应地，很难确定程序运行失败的原因。

2）多态和动态绑定对可维护性具有消极影响。维护程序员的首要任务通常是努力理解产品

（如第 14 章中所述，维护人员很少是代码的开发人员）。然而，如果一个特定的方法有多种可能性，那么理解产品的工作将非常费力。在代码中一个特定位置，程序员必须考虑动态调用的所有可能方法，这也是一个耗时的任务。

换句话说，多态和动态绑定为面向对象范型添加优点的同时也带来了缺点。

我们以对面向对象范型的讨论来结束本章。

7.9　面向对象范型

有两种方式来看待每个软件产品：一种是只考虑数据，包括局部和全局的变量、参数、动态数据结构和文件；另一种是只考虑在数据上执行的操作，即过程和函数。按照把软件分成数据和操作的分法，传统技术主要分成两组。面向操作的技术主要考虑产品的操作，数据是次重要的，只在已经深入分析了产品操作之后才会考虑数据。相反地，面向数据的技术强调产品的数据，只在数据框架内对操作进行检查。

面向数据和面向操作方法的基本缺点是：数据和操作是同一事物的两个方面，数据项不能改变，除非一个操作作用于该数据项，同样，与数据无关的操作毫无意义。因此需要同等对待所需的数据和操作的技术。面向对象技术做到了这点并不为奇。毕竟，对象是由数据和操作组成。前面介绍过，对象是抽象数据类型（更精确的说是类）的实例。因此它结合了数据和对该数据执行的操作，在对象中数据和操作地位相同。类似地，在所有面向对象技术中，数据和操作被认为是一样重要的，任何一个都不能优先于另一个。

声称在面向对象范型技术中会同时考虑数据和操作是不准确的。从逐步求精的资料中可以明显看出（见 5.1 节），有时候需要强调数据，而另一些时候可能操作更关键。然而，总的来说，在面向对象范型的工作流中数据和操作具有相同的重要性。

本书第 1 章和本章给出了很多理由说明面向对象范型优于传统范型。其根本原因在于，一个定义良好的对象，即一个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。也就是说，现实世界的实体和模拟该实体的对象之间有一个清晰的映射。

如何实现这些的细节被隐藏了，与对象进行通信的唯一方式是发送消息。因而，对象本质上是具有良好定义的接口的独立单元。因此，它们维护起来比较容易，而且安全，回归错误发生的概率也降低了。另外，对象是可复用的，这种可复用性通过继承得到增强，这将在第 8 章中进行阐述。现在回到使用对象进行开发的问题，通过组合这些软件的基本组件来构造一个大型产品，比使用传统范型更安全。由于对象本质上是一个产品的独立组件，产品开发和开发管理都更容易，因此引起错误的可能性更小。

面向对象范型的所有这些优势引发了一个问题：如果传统范型与面向对象范型相比如此之差，为什么传统范型会取得这么多的成功？意识到传统范型是在软件工程没有广泛实践的时期被采用的，这也就不难解释了。软件只是简单地“编写”，对管理员来说，最重要的是程序员写出程序代码，几乎没有产品的需求和分析（系统分析），设计也基本没有。边写边改模型（见 2.9.1 节）是 20 世纪 70 年代的典型技术。因此，开始时，传统范型是大多数软件开发者使用的方法和技术。那时，也很少有人怀疑传统范型的结构化技术引发了世界软件产业的较大进展。然而，随着软件产品规模的增大，结构化技术的不足开始显现出来，从而提出了面向对象范型。

这也引起了另一个问题：如何能确定面向对象范型比现在所有的技术都优越呢？没有数据能确切证明面向对象技术比当前其他任何技术更好，并且很难想像如何取得这些数据。所能做的就是依靠采用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，但大多数（如果不是绝大多数）报告都证明了使用面向对象范型是一个明智的选择。

例如，IBM 用面向对象技术开发的三个完全不同的工程［Capper，Colgate，Hunter，and James，1994］。几乎在每个方面，面向对象范型都比传统范型好得多，特别是检测到的错误数

量显著下降；在开发和交付后维护期间，只有很少的修改申请，这里的修改不是由不可预见的商业变更而产生的修改；在适应和完善维护能力上有显著提高。另外，可用性也有所提高，尽管不像前面四项改进那么大，而且可用性的提高对系统性能来说没有什么区别。

采用基于对 150 个有经验的美国软件开发人员的调查结果来确定他们对面向对象范型的态度［Johnson，2000］。样本包括 96 个使用面向对象范型的开发人员和 54 个仍然使用传统范型开发软件的人员组成。两组人员都感到面向对象范型更优越，而面向对象的小组的积极态度更强些。两个小组基本都不计较面向对象范型的各种缺点。

虽然面向对象范型有很多优点，但是一些真正难点和问题也被报道出来了。一个频繁报告的问题是与开发工作量和规模相关的。第一次做任何新事物花费的时间比后面的要多，有时称这个最初阶段为*学习曲线*（learning curve）。但是当一个组织第一次使用面向对象范型，即使考虑了学习曲线，通常花费的时间还是比预期的要多，因为产品的规模比使用结构化技术时的更大。特别是产品具有图形用户界面（GUI）时更是如此（详见 10.14 节）。在这之后，情况会改善很多。首先，交付后维护费用会更少，从而减少了产品整个生命周期的成本。其次，下次开发一个新产品时，通常可以重用前面项目的一些类，从而进一步降低了软件成本。当第一次使用一个 GUI 时这点非常重要，花在 GUI 上的大部分工作量能在后续的产品中得到补偿。

继承的问题更难解决一些。

1）使用继承的一个主要原因是为了创建一个新的子类，这个子类与它的父类差别很小，而且不会影响到它的父类或者继承层次结构中的其他祖先类。反之，一旦实现一个产品，任何对已存在类的改变都会直接影响到继承层次结构中它的所有子孙，这通常称为*脆弱的基类问题*（fragile base class problem）。至少受影响的单元必须重新编译。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，这不是一个小任务。为了最小化这个问题，在开发过程中对所有类进行精心设计是非常重要的。这将减少由对存在类的改变而引起的连锁反应。

2）第 2 个问题产生于对继承的随意使用。除非明确阻止，否则子类继承它的父类（们）的所有属性。通常，子类具有它们自身的额外属性。结果，在继承层次结构中较低层次的对象很快变得巨大起来，从而产生存储上的问题［Bruegge，Blythe，Jackson，and Shufelt，1992］。防止这种情况发生的一种方式是把格言“在任何可能的地方使用继承”改为“在任何适当的地方使用继承”。另外，如果后继的类不需要祖先类的某个属性，那么应该明确地排除这个属性。

3）第 3 组问题源自多态和动态绑定，这在 7.8 节中已经论述过了。

4）第 4 个问题是，使用任何语言都可能写出坏的代码。然而，因为面向对象语言支持各种构造，所以用面向对象语言比用传统语言更容易写出坏的代码，当使用不当时，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，需要格外的小心以确保代码总有最高的质量。

最后一个问题是，将来是否会有比面向对象范型更好的技术出现？也就是说，将来是否会有一个新的技术出现在图 7-28 最顶端箭头的上方？即使是最忠实的支持者也不会表示面向对象对象范型是解决所有软件工程问题的最终答案。另外，今天的软件工程已经超越对象，瞄准了下一个重要突破。毕竟，在人类奋斗的领域中，很少有过去的发现超越当今提出的任何事物，未来的方法肯定会取代面向对象范型。已经提出的*面向方面的编程*（aspect-oriented programming，AOP）可能占据一席之地［Murphy et al.，2001］。AOP 是否确实会成为图 7-28 的未来版本中的下一个重要概念，或者其他范型是否将作为面向对象范型的后继而得到广泛的采纳，还有待观察。重要的经验是，就目前而言，面向对象范型比其他技术都好。

本章回顾

本章首先描述了模块（7.1 节），对象和方法都是模块。接下来两节基于模块内聚和模块耦合分析了如何构建一个设计良好的模块（7.2 节和 7.3 节），特别地，模块应该具有高内聚和低

耦合，这里给出了不同类型的内聚和耦合的描述。7.4 ~ 7.7 节介绍了各种抽象类型。在数据封装（7.4 节）中，一个模块包括数据结构和在该数据结构上执行的操作。抽象数据类型（7.5 节）是一种数据类型以及在该类型实例上执行的操作。信息隐藏（7.6 节）包含这样的设计模块方式：对其他模块隐藏了实现细节。不断增强的抽象在类的描述中达到了顶点，类是支持继承的一种抽象数据类型（7.7 节）。对象是类的实例。继承、多态和动态绑定是 7.8 节的主要内容。本章最后讨论了面向对象范型（7.9 节）。

延伸阅读材料

在［Dahl and Nygaard，1996］中第一次描述了对象。本章中的许多思想最初都是由 Parnas 提出的［1971、1972a、1972b］。软件开发中抽象数据类型的使用在［Liskov and Zilles，1974］中提出；另外一篇比较重要的早期论文是［Guttag，1977］。

内聚和耦合的主要来源是［Stevens，Myers and Constantine，1974］。组合/结构化设计的思想已经扩展到对象［Binkley and Schach，1997］。

［Meyer，1997］中有关于对象的介绍性资料。［Meyer，1996b］中描述了不同类型的继承。［El-Rewini et al.，1995］中能找到一些关于面向对象范型的短文。面向对象编程系统、语言和应用年会学报（OOPSLA）包括大量的研究论文和描述成功的面向对象项目的报告。［Capper，Colgate，Hunter，and James，1994］中描述了 IBM 的三个成功使用面向对象范型的项目。［Johnson，2000］中描述了关于面向对象范型态度的一个调查。［Fayad，Tsai，and Fulghum，1996］描述了如何向面向对象技术转变，包含了许多给管理者的建议。

在 1992 年 10 月的《IEEE Computer》期刊中包含了一些关于对象的重要论文，特别是［Meyer，1992］，它描述了“契约式设计”。各种与对象相关的论文可以在 1993 年 1 月出版的《IEEE Software》中找到。Snyder 所写的论文［1993］准确定义了这一领域中的关键术语，这非常有用。多态可能存在的缺点描述于［Ponder and Bush，1994］。1995 年 10 月出版的《Communications of the ACM》包含了关于对象技术的论文，2006 年第 2 期的《IBM Systems Journal》也有相关的论文。

2001 年 10 月出版的《Communicatoins of the ACM》包含了 11 篇关于面向方面编程的论文，特别有意义的是［Elrad et al.，2001］和［Murphy et al.，2001］。

关于继承对缺陷密度影响的调查见论文［Cartwright and Shepperd，2000］。

习题

7.1　确定下面方法的内聚类型：

editProfitAndTaxRecord

editProfitRecordAndTaxRecord

readDeliveryRecordAndCheckSalaryPayments

computeTheOptimalCostUsingAksen'sAlgorithm

measureVaporPressureAndSoundAlarmIfNecessary

7.2　假定你是一个负责产品开发的软件工程师。管理者要求你研究如何确保你们小组所设计的模块能尽可能地被复用的方法。你会怎么回答？

7.3　现在管理者让你确定如何能够复用已有模块。你的第一个建议是把每个具有偶然性内聚的模块划分成一些具有功能性内聚的单个模块。你的管理者正确地指出，这些单个的模块既没有经过测试也没有建立文档。那么你会如何回答？

7.4　内聚对维护的影响是什么？

7.5　耦合对维护的影响是什么？

7.6　请区分数据封装和抽象数据类型。

7.7　请区分抽象和信息隐藏。

7.8　请区分多态和动态绑定。

7.9　如果使用无动态绑定的多态会如何？

7.10　如果使用动态绑定而不使用多态会如何？

7.11　按照指导教师要求，将图 7-23 中的注解转换为 C++ 或 Java，并确保最终生成的模块能够正确执行。

7.12　已经有人提出 C++ 和 Java 支持抽象数据类型的实现，但要以牺牲信息隐藏为代价。对这个观点进行讨论。

7.13　正如备忘录 7.1 中所指出的，在 1966 年就提出了对象的概念，但差不多 20 年后，对象概念才被重新发掘并被广泛接受。你能解释这个现象吗？

7.14　指导教师分发一个软件产品，大家分别从信息隐藏、抽象级别、耦合和内聚的角度对模块（对象和方法）进行分析。

7.15　继承的优点和缺点是什么？

7.16　（学期项目）给出附录 A 中 Osric 的办公用品和装饰产品的类的例子。

7.17　（软件工程读物）教师分发［R. Alexander，2003］的复印件。你认为面向方面编程会成为图 7-28 未来版本中的下一个主要概念吗？说明你的答案。

参考文献

[R. Alexander, 2003] R. Alexander, "The Real Costs of Aspect-Oriented Programming," *IEEE Software* **20** (November/December 2003), pp. 92–93.

[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.

[Briand, Daly, Porter, and Wüst, 1998] L. C. Briand, J. Daly, V. Porter, and J. Wüst, "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems," *Proceedings of the Fifth International Metrics Symposium*, Bethesda, MD, November 1998, pp. 246–57.

[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.

[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.

[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.

[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, 2005] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*, 5th ed., O'Reilly and Associates, Sebastopol, CA, 2005.

[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.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries*, ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, 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.

[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 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), 123–26.

[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.

[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, 2003] B. STROUSTRUP, *The C++ Standard: Incorporating Technical Corrigendum No. 1*, 2nd ed., John Wiley and Sons, New York, 2003.

[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.

[Yu, Schach, Chen, and Offutt, 2004] L. YU, S. R. SCHACH, K. CHEN, AND J. OFFUTT, "Categorization of Common Coupling and its Application to the Maintainability of the Linux Kernel," *IEEE Transactions on Software Engineering* **30** (October 2004), pp. 694–706.

第 8 章　可复用性和可移植性

学习目标

通过本章学习，读者应能：

- 解释复用的重要性。
- 理解复用的障碍。
- 描述在各种工作流中实现复用的技术。
- 理解设计模式的重要性。
- 讨论复用对可维护性的影响。
- 解释可移植性的重要性。
- 明白实现可移植性的障碍。
- 开发可移植的软件。

如果重复发明轮子是一种违法行为，那么今天许多软件专业人员都将被关入监狱。例如，如果没有成千上万，也有成百上千的 COBOL 工资报表程序本质上都在做着相同的工作，无疑，整个世界只需要一个能在各种硬件平台上运行的工资报表程序就可以了，如果必要，可以对其进行裁减以满足某个组织的专门需求。然而，世界各地的多数组织并没有使用先前开发的工资报表程序，而是从头建造自己的工资报表程序。本章将研究软件工程师乐于不断重复开发程序的原因，以及如何使用可复用的组件来建造可移植的软件[⊖]。首先介绍可移植性和可复用性之间的区别。

备忘录 8.1

复用的概念并不局限于软件。例如，现在很少有律师会从头开始起草一份遗嘱，他们往往使用文字处理器存储以前起草的遗嘱，然后对已有的遗嘱做一些适当的修改并形成一份新的遗嘱。其他的法律文件（如合同）也是以同样的方式完成的。

古典音乐作曲家经常复用他们自己的音乐。例如，在 1823 年，舒伯特为 Helmina von Chezy 的戏剧《Rosamunde, Princess of Cyprus》创作了一首幕间曲；次年，他在第 13 号弦乐四重奏的慢乐章中复用了这首曲子。贝多芬也在他的作品第 66 号中复用了另一位伟大作曲家莫扎特的作品，贝多芬简单地借用了莫扎特的歌剧《The Magic Hute》（魔笛）中第 22 幕的咏叹调“A Girlfriend or Little Wife”（女朋友或小妇人），然后根据这个咏叹调，为钢琴配乐的大提琴演奏谱写了一连串 7 个音符的变奏。

在我看来，有史以来最伟大的复用者是莎士比亚。他的天赋在于复用别人的情节，我想不出来哪一个故事情节是他自己撰写的。例如，他的历史剧大量复用了 Raphael Holinshed 在 1577 年发表的作品《Chronicles of England, Scotland and Ireland》（英格兰、苏格兰和爱尔兰编年史），还有《Romeo and Juliet》（罗密欧和朱丽叶，1594 年）几乎每一行都出于 Arthur Brooke 于 1562 年发表的长诗《The Tragicall Historye of Romeus and Juliet》。该书出版 2 年后莎士比亚才出生。

⊖ 如前言所述，本章内容可与第二部分并行教授。

但是这个复用的传奇并不始于此。事实上，最早知道的版本是古希腊小说家 Xenophon 在公元 200 年左右所创作的《Ephesiaka》（Ephesiaka 城的传说）。1476 年，Tommaso Guardati（叫 Masuccio Salernitano 可能大家会更熟悉）在其由 50 篇小说组成的选集《Il Novellino》中的第 33 篇中复述了 Xenophon 的故事。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）里再次复用了该故事。他第一次将这个故事的发生安排在意大利的维罗纳市。Brooke 的诗中复用了 Matteo Bandello 所作的《Giulietta e Romeo》（1554 年）中的一部分，复用的是 da Porto 版本。

而《罗密欧和朱丽叶》的复用传奇还没有结束。1957 年，《West Side Story》在百老汇公演。这个音乐剧的剧本由 Arthur Laurents 编写，乐谱由 Leonard Bernstein 所作，Stephen Sondheim 填词，该剧复用了莎士比亚的故事版本。随后，这部百老汇的音乐剧被一部好莱坞影片所复用，该影片在 1961 年赢得了 10 项奥斯卡奖。

8.1　复用的概念

如果从整体上修改一个产品，使其运行在另一个编译器 - 硬件 - 操作系统配置上，比从头开始编程要容易得多，那么称这种产品是可移植的（portable）。相反，复用（reuse）是指使用一个产品的组件来简化另一个功能不同的产品的开发。一个可复用组件不一定是一个模块或一个代码段，它也可以是一个设计、用户手册的一部分、一组测试数据或一个周期和成本估算。（对于复用的另一种观点，参见备忘录 8.1。）

复用有两种类型：机会复用和有意复用。如果新产品的开发者意识到，先前开发的产品中有一个组件能用在新产品中，那么这就是一个机会复用（opportunistic reuse），有时称为偶然复用（accidental reuse）。如果专门为将来的可能复用构造软件组件的话，就叫做系统性复用（systematic reuse）或有意复用（deliberate reuse）。与机会复用相比，有意复用的一个潜在优点是：专门为以后产品的使用所构造的组件的复用可能会更容易和安全，因为这些组件通常是健壮的、文档说明完善并且经过全面测试的。另外，它们通常风格一致，由此维护更加容易。但另一方面，在公司里实现有意复用的花费将是昂贵的，它需要花费时间去规定、设计、实现、测试一个软件组件，并形成软件组件的文档。但是，不能保证这个组件都能被复用，也就是说，投入到开发可复用的组件上的投资也许得不到回报。

首次建造计算机时，没有什么是可复用的。每次开发一个产品，一切都是从零开始构造，例如，乘法程序、输入输出程序、计算正弦和余弦的程序。然而很快大家就意识到有相当多的成果和努力是重复无益的，于是构造了子程序库。此后，程序员就可以随时简单地调用计算平方根或正弦的函数。这些子程序库逐渐变得越来越复杂，并被开发成运行时的支持程序。因此，当一个程序员调用一个C++ 或 Java 方法时，他不必编写代码去管理堆栈或参数传递，而是调用适当的运行时支持程序进行自动处理。子程序库的概念已经扩展成大规模统计库（如 SPSS［Norušis，2005］）以及数值分析库（如 NAG［2003］）。类库在帮助使用面向对象语言的用户方面也起着重要作用。例如，Smalltalk 的成功至少部分归功于 Smalltalk 库中广泛的项目种类以及浏览器（一个用来帮助用户浏览类库的 CASE 工具）的出现。关于C++，有大量不同的类库可用，包括很多位于公共域的类库，例如，C++ 标准模板库（STL）［Musser and Saini，1996］。

一个应用程序接口（API）通常是一组方便编程的操作系统调用。例如，Win32 是一个微软操作系统（如 Windows XP）；Cocoa 是 Mac OS X（苹果操作系统）的一个 API。尽管一个 API 通常作为一组操作系统调用来实现，但对程序员来说，程序组成的 API 可以看成一个子程序库。

例如，Java 应用编程接口就包含了很多包（库）。

无论一个软件产品的质量有多高，如果它花费两年时间才推向市场，而一个竞争产品的交付只用一年时间，那么它是不会被卖出的。在市场经济中，开发产品的时间是非常关键的。如果产品不能在时间方面取胜，那么其他所有标准（如“一个好的产品由什么组成”）都是无关紧要的。对于在把产品首次推向市场过程中屡遭失败的一个公司来说，软件复用提供了一项诱人的技术。毕竟，如果一个已有的组件能够复用，就不需要去规定、设计、实现、测试以及归档该组件。关键是，在平均情况下，任何软件产品中只有 15% 是真正面向创意的［Jones，1984］，而 85% 理论上是可标准化的，并可在未来产品中复用。

85% 基本上是一个复用率的理论上限，然而在实际中，只能实现 40% 左右的复用率。这导致了一个明显的问题：如果在实际中能达到这个复用率，并且复用也决不是一个新的思想，为什么如此少的组织使用复用来缩短开发过程呢？

8.2　复用的障碍

复用存在很多障碍：

- 太多软件专业人员宁愿从头开始编写一个组件，也不愿意复用一个别人所编写的组件，其中的含意是：除了自己编写的，其他的组件不会是好的。这就是所谓非我发明（not invented here，NIH，指某些人或企业，对自我创新的能力颇为自负，并拒绝采用或购买他人或别的公司所发明的技术）综合症［Griss，1993］。NIH 是一个管理方面的问题，如果管理部门意识到这个问题，通常可以通过经济激励的方式来促进复用。
- 假如能确定所复用的组件不会给产品引入错误，许多开发人员才愿意复用这个组件。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到过别人写出的有错误的软件。这里的解决方法是：在复用这些组件前，将这些潜在可复用的组件进行彻底的测试。
- 一个大型的机构可能有成千上万种潜在有用的组件。为了以后能有效地检索，这些组件应该如何存储？例如，一个可复用的组件数据库可能包含 20 000 项，其中 125 项是排序程序。那么必须规划该数据库，使得新产品的设计人员能很快地确定：125 项排序程序中的哪一项适用于新产品。解决存储/检索问题是一个技术问题，针对此问题已经提出了各种解决方法（例如，［Meyer，1987］或［Prieto－Díaz，1991］）。
- 复用是昂贵的。Tracz［1994］指出要考虑三种成本：建造可复用组件的成本、复用组件的成本以及定义和实现一个复用过程的成本。他估计仅建造可复用组件就要增加至少 60% 的成本。一些机构的报告指出，成本将增加 200%，甚至 480%。但是在惠普公司的一个复用工程项目中，建造可复用组件的成本仅增加 11%［Lim，1994］。
- 依照合同开发的软件会产生法律问题。按照客户与软件开发机构之间签订的合同，通常软件产品是属于客户的。因此，如果软件开发者在一个客户的新产品中复用另一个客户产品中的组件，这本质上构成了对第一个客户的版权侵犯。对于内部软件，即当开发人员与客户是同一机构的成员时，不会发生这个问题。
- 另一个障碍来自复用商业现货供应（COTS）组件时。一般，开发人员很难获得 COTS 组件的源代码，因此，复用 COTS 组件的软件限制了可扩展性和可修改性。

前 4 个障碍至少在原则上是能够克服的。因此，除了一些法律问题和 COTS 组件的问题，基本上没有什么严重障碍能阻止在一个软件机构内实现复用（可参见备忘录 8.2）。

备忘录 8.2

万维网是“都市神话”的巨大源泉，所谓神话，是指若经过仔细调查，故事的真实性大多会有点站不住脚。代码复用就是这样一个都市神话。

这个都市神话故事讲的是，澳大利亚空军为了直升机格斗训练的需要，建造了一个基于虚拟现实的训练仿真器，为了使情节尽可能真实，程序员放入了详细的地形和（在北方地区的）大量袋鼠。想法是，直升机掠过袋鼠群时，受惊吓的动物所扬起的尘土可能会把直升机的位置暴露给敌人。

程序员被指示要模仿袋鼠群的移动以及它们对直升机的反应。为了节省时间，程序员复用了原本用来模拟被直升机攻击的步兵团反应的代码。只做了两个改动：一是把士兵的图标换成了袋鼠的图标；二是提高了图形的移动速度。

在一个晴天，一组澳大利亚飞行员想通过飞行模拟器，向一些来访的美国飞行员证明他们的威力。他们“低空掠过”（飞得非常低）虚拟的袋鼠群。如期望的那样，袋鼠四下散开，但随后袋鼠又在一个小山后出现并向直升机发射“毒刺”导弹。原因是，当复用虚拟的步兵团实现时，程序员忘记了去掉步兵发射导弹的那部分代码。

然而，就如《风险文摘》杂志所报道的，似乎这不完全是一个都市神话，大部分情节实际上已经发生过［Green，2000］。澳大利亚国防科技部陆军仿真部门的领导人 Anne－Marie Grisogono 博士，于 1999 年 5 月 6 日在澳大利亚堪培拉的一次会议上讲述了这个故事。仿真器的设计很真实（它甚至包含了 200 多万棵如航空图片所示的虚拟树木），但添加袋鼠只是为了好玩。程序员确实复用了从“毒刺”导弹分离出的组件，这样袋鼠就能检测到直升机的到来，但是袋鼠的行为被设置为“撤退”，所以直升机接近时，袋鼠就会四散逃跑。然而，当软件小组在实验室测试他们的仿真器时（不是在参观者面前），他们发现忘却了去掉武器和开火的行为，也没有规定模拟袋鼠所使用的武器，所以当袋鼠向直升机开火时，它们使用的是默认武器，即大型海滩彩球。

Grisogono 确认袋鼠已当即被解除了武装，因此，大家现在可以安全地飞越澳大利亚领空了。尽管故事以戏剧结束了，但是软件专业人员复用代码时需小心，不要过度复用。

8.3　复用案例研究

许多已经出版的案例研究显示了在现实中应该如何成功地进行复用；具有重要影响的复用案例研究包括［Matsumoto，1984，1987；Selby，1989；Prieto－Díaz，1991；and Lim，1994］。这里将分析两个案例。首先描述的是发生在 1976 年到 1982 年之间的复用项目，这很重要，因为那时用于 COBOL 设计的复用机制和今天在面向对象应用架构中使用的复用机制（8.5.2 节）相同。这个案例研究的目的是阐明现代复用实践。

8.3.1　雷锡恩导弹系统部门

在 1976 年，雷锡恩导弹系统部进行了一项研究，确定系统的设计和代码复用是否可行［Lanergan and Grasso，1984］。期间分析和分类了 5 000 多个使用中的 COBOL 产品。研究人员确定，在商业应用产品中，只完成 6 种基本操作。有 40% ~60% 的商业应用研究设计和模块能被标准化和复用。发现的 6 种基本操作是：排列数据、编辑或处理数据、组合数据、分解数据、更新数据和报告数据。在随后的 6 年里，公司集中精力试图在任何可能的地方复用设计和代码。

雷锡恩方法以两种方式使用复用，研究人员将其称为功能模块和 COBOL 程序逻辑结构。功能模块（functional module）是为特殊目的而设计和编写的一个 COBOL 代码片段，例如，一个编辑程序、数据库过程调用程序、税款计算程序或者可接受账户的日期时效程序。使用 3 200 个

可复用的模块产生的应用程序平均包含60%的可复用代码。功能模块被细心地设计、测试以及形成文档说明。使用这些功能模块的产品更可靠，并且产品整体上需要更少的测试。

这些模块存储在一个标准的副本库中，可使用动词复制（copy）来获得。也就是说，代码物理上没有存在于应用产品中，而是由COBOL编译器在编译时包含进来的，该机制类似于C++中的**#include**机制。因此，生成的源代码长度要比被复制代码物理存在时的长度要短，相应地，维护也更容易。

雷锡恩研究人员也使用他们自行定义的术语**COBOL**程序逻辑结构（COBOL program logic structure）。这是一个必须被充实成为完整产品的架构。逻辑结构的一个例子是更新逻辑结构，其用来执行一个有序的更新，例如，5.1.1节中的小型案例研究。错误处理和顺序检查一样是内嵌的。该逻辑结构在长度上有22个段落（COBOL程序的单元）。许多段落可使用功能模块（如`get-transaction`、`print-page-headings`和`print-control-totals`等）进行填充。图8-1给出了一个COBOL程序逻辑结构架构符号描述，其中的段落已由功能模块填充。

使用这些模板有很多优点。它使一个产品的设计和编码更快更容易，因为该产品的架构已经存在，所需要的就是填充细节。易出错的区域（如文件结束的条件句）已经经过测试。事实上，整体测试也将更容易进行。雷锡恩认为，当用户要求进行系统修改或增加功能时，该技术的主要优点才会显现。一旦维护程序员熟悉了相关的逻辑结构，几乎可以认为他就是原来开发小组的一员。

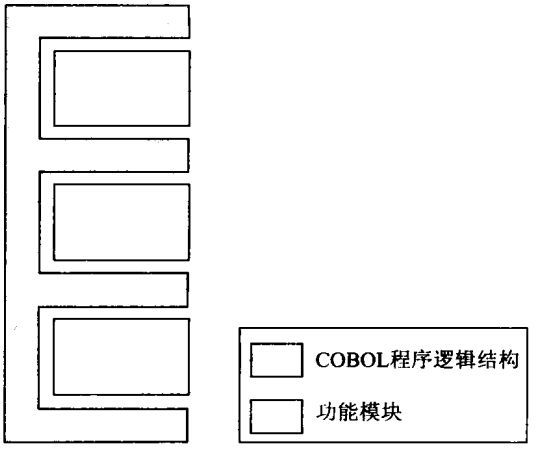


图8-1　雷锡恩导弹系统部复用机制的示意图

到1983年为止，在新产品开发中，使用逻辑结构已经超过了5 500次。大约60%的代码是由功能模块（即可复用的代码）组成的。这意味着设计、编码、模块测试以及归档的时间大约节省了60%，从而在软件产品开发中，估计生产力提高了50%。但是，对雷锡恩来说，真正的好处在于，一致风格所产生的代码的可读性和易懂性将减少60%～80%的维护成本。遗憾的是，在获得必需的维护数据之前，雷锡恩就关闭了该部门。

第二个复用案例研究是一个警示，而不是一个成功的故事。

8.3.2　欧洲航天局

在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，导致了火箭发射后37秒坠毁。火箭和载运物的造价大约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位整数转换为16位无符号整数。被转换的数字大于2^{16}，因此产生了一个Ada异常（exception，运行时故障）。不幸的是，代码中没有明确的异常处理程序来处理这个异常，因此软件崩溃了。这导致火箭上的计算机崩溃，因而导致了阿丽亚娜5号火箭的坠毁。

具有讽刺意义的是，导致失败的那个转换是不必要的。在火箭发射之前，会执行一些计算以校正惯性参照系统，这些计算应该在发射前9秒就停止。然而，如果在倒数计时中有一个暂停，那么在倒数计时重新恢复后，要重设惯性参照系统需要花费几个小时的时间。为了避免这种情况发生，在开始进入飞行状态（也就是完全飞行）后50秒内仍继续进行计算（尽管这样，一旦已经发射，就没有办法再校正惯性参照系统了）。这个无用的继续校准处理过程的计算导致了失败。

欧洲航天局使用了一个谨慎的软件开发过程，有一个有效的软件质量保证体系。那么在 Ada 代码中，为什么没有异常处理程序去处理这样的溢出错误呢？原来为了避免计算机超载，那些不会导致溢出的数据转换就没有采取任何保护措施。上面提到的代码已经存在了 10 年，它是对阿丽亚娜 4 号火箭（阿丽亚娜 5 号火箭的先驱）的软件的复用，未经任何修改，也没有做进一步测试。数学上的分析已经证明了该段代码对于阿丽亚娜 4 号火箭是完全正确的。然而，这种分析是在一些假设的基础上进行的，虽然这些假设对阿丽亚娜 4 号火箭来说是成立的，但对阿丽亚娜 5 号火箭来说却是不成立的。因此，该分析不再适用，代码需要异常处理的保护，以处理可能出现的溢出。如果不是由于性能的限制，整个阿丽亚娜 5 号的 Ada 代码中肯定会有异常处理程序。另一个选择是，如果相关模块已经包含了断言——要求被转换的数小于 2^{16}，那么在测试中以及产品安装后（见 6.5.3 节）都使用 **assert pragma** 编译指示，就可以避免阿丽亚娜 5 号火箭的坠毁［Jézéquel and Meyer，1997］。

这个复用经历的主要教训是，在一个背景中开发的软件，当在另一个背景中复用时，必须重新进行测试。也就是说，一个可复用的软件模块本身不需要重新测试，但在集成到新产品之后，它必须进行重新测试。另一个教训是，如 6.5.2 节所述，完全地依赖数学证明的结果是不明智的。

下面考察一下面向对象范型对复用的影响。

8.4　对象和复用

大约 30 年前，组合/结构设计理论被第一次提出时，其宣称一个理想的模块应该是具有功能性内聚的模块（见 7.2.6 节）。也就是说，如果一个模块只执行一个操作，那么就可认为它是一个复用的典型可选模块，并且对这种模块的维护也将是容易的。这个推论的缺点在于，一个具有功能性内聚的模块不是自主的和独立的。相反，它必须对数据进行操作。如果复用这样一个模块，那么必须复用它原来所执行的数据。如果每个新产品中的数据与原来产品中的数据不一致，那么，要么修改数据，要么修改具有功能性内聚的模块。因此，与先前的认识相反，功能性内聚对复用来说不是理想的。

按照 1974 年最初提出的 C/SD，下一个最好的模块类型是具有信息性内聚的模块（见 7.2.7 节）。现在，可领会到这样的一个模块本质上是一个对象，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，因为它是对一个特定的现实世界实体的模块化（概念上独立或封装），但隐藏了它的数据和对数据的操作的实现（物理上的独立或信息隐藏）。因此，当正确使用面向对象范型时，所产生的模块（对象）就具有信息性内聚，这也促进了复用。

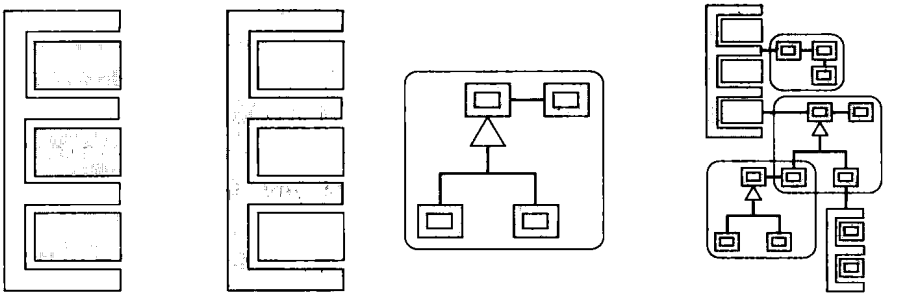
8.5　在设计和实现过程中的复用

在设计过程中，可能有着明显不同的复用类型。复用的内容也会有所不同，范围从一个或两个制品到整个软件产品的体系结构。现在来考察一下设计复用的各种类型，其中有一些会延续到实现阶段。

8.5.1　设计复用

当设计一个产品时，设计小组成员可能会意识到，从先前设计中得到的一个类，经过较小的修改或不做任何修改，就能复用于当前项目中。如果某个组织在一个特定领域（如银行业务或空中交通控制系统）中开发软件，那么这种复用会非常普遍。这些组织可以通过建造未来可能会复用的设计组件的知识库，并鼓励设计人员复用，来促进这种类型的复用。这样可更快地提出整体设计，另外与从头开始整个设计相比，其设计质量可能会更高。另外，如果一个类的设计能够复用，那么可能该类的实现也可复用，如果不是代码层次的，至少也是概念上的。

这种方法可扩展为库复用，如图 8-2a 中所描述的。一个库是一组相关的可复用程序的集合。例如，科学计算软件的开发者很少自己编写程序用于执行这些通用任务（如矩阵转置或查找特征值等），而是会购买一个类似 LAPACK++[2000] 这样的科学类库来实现。从而，在未来的软件中，在任何时候都可以使用科学类库中的类。



a) 一个库或工具包　b) 一个架构　c) 一个设计模式　d) 包含架构、工具包和3种设计模式的软件体系结构

图 8-2　4 种类型设计复用的示意图。阴影表示设计复用

另一个例子是图形用户接口（GUI）库。程序员不需要从头开始编写 GUI 方法，只需要使用一个 GUI 类库或工具包（Toolkit），工具包是能处理 GUI 的各个方面的一组类。有许多这种类型的 GUI 工具包可用，包括 Java 抽象窗口工具包（AWT）[Flanagan, 2005] 等。

库复用的一个问题是，库通常以一组可复用代码制品集合的形式存在，而不是可复用设计。工具包通常也只是促进代码复用，而不是设计复用。借助浏览器（即使用一个显示继承树的 CASE 工具）可以缓解这个问题。然后，设计人员可以遍历库的继承树，检查各种类的域，从而确定哪种类适用于当前设计。

库和工具包复用的一个关键方面是，设计人员负责整个产品的控制逻辑，如图 8-2a 所示。库或工具包支持软件开发过程的方式是，其提供了和特定操作相结合形成产品的部分设计。

此外，应用架构与库或工具包相反，它提供了控制逻辑，而由开发人员负责特定操作的设计，这将在 8.5.2 节中讨论。

8.5.2　应用架构

如图 8-2b 所示，一个应用架构（application framework）包含了设计的控制逻辑。当复用一个架构时，开发人员需要设计所构建产品的特定应用操作。特定应用操作插入的地方称为热点（hot spot）。

现在，架构（framework）一词通常指一个面向对象的应用架构。例如，在 [Gamma, Helm, Johnson, and Vlissides, 1995] 中，架构被定义为“一组协同类，它们构成软件的一个特定类的可复用设计”。然而，考虑 8.3.1 节中图 8-1 所示的雷锡恩导弹系统部门的案例研究，其与图 8-2b 是一致的。也就是说，20 世纪 70 年代的雷锡恩 COBOL 程序逻辑结构是今天面向对象应用架构的先驱。

例如，用于编译器设计的一组类是一个应用架构。设计小组只需要为语言和所需目标机器提供专门的类。然后，把这些类插入架构中，如图 8-2b 中的白色方框所描述。另一个架构的例子是用于控制 ATM 的软件中的一组类。那里，设计人员需要为特定的银行服务提供类，而这些服务是由银行网络的 ATM 机器供给的。

复用一个架构进行产品开发比复用一个工具包更快，原因有二：其一，多数设计随架构一

起复用，因此，需要从头开始设计的部分更少；其二，与操作相比，在架构中被复用的设计部分（控制逻辑）通常更难设计，因此最终设计的质量也可能比复用工具包要高。就像复用库或工具包一样，架构的实现也能复用。开发人员可能需要使用架构的名称和调用惯例，但这花费很小。由于控制逻辑已经在其他复用了应用架构的产品中得到测试，并且先前的维护人员可能已经维护过复用相同架构的其他产品，所以最终的产品更容易维护。

IBM 的 WebSphere（以前称为 e - Components，最初则是 San Francisco）是一个用 Java 构建的在线信息系统架构。它使用的 Enterprise JavaBeans 就是为分布于网络上的客户提供服务的类。

除了应用架构外，还有许多代码架构是可以使用的。第一个成功的商用代码架构是 MacApp——一个在苹果机上编写应用软件的架构。Borland 的可视组件库（VCL）是一个面向对象的架构集合，这些架构可用来构建基于 Windows 的应用程序的 GUI。VCL 应用程序能执行标准窗口操作，例如，移动窗口和重设窗口大小、处理对话框的输入、处理鼠标点击或菜单选择等事件。

下面介绍设计模式。

8.5.3　设计模式

Christopher Alexander（参见备忘录 8.3）说过，“每个模式都描述一个在我们周围反复发生的问题，并且描述了那个问题的核心解决方法，这样你就能无数次使用这个解决方案，而不需要两次寻求同一问题的答案”［Alexander et al.，1977］。虽然他是在房屋和其他建筑物的范畴内描述模式的，但是他的观点也同样适用于设计模式。

备忘录 8.3

Christopher Alexander 是面向对象软件工程领域中最有影响力的人物之一，他是一位世界著名的建筑设计师，但他却坦白地承认关于对象或软件工程，自己只知道一点皮毛或者根本不了解。在其著作，特别是［Alexander et al.，1977］中，他描述了一种体系结构模式语言，用来描述城镇、建筑、房间、公园之类的事物。他的思想被面向对象软件工程师采用，并加以改造，特别是被称为“四人帮”的 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 所采用。他们关于设计模式的畅销书籍［Gamma，Helm，Johnson and Vissides，1995］使 Alexander 的思想为面向对象团体所广泛接受。

在其他情况下也存在着模式。例如，当到达一个机场时，飞行员必须知道适当的着陆模式，即把飞机降落在正确跑道所需要的方向、高度和旋转度的序列。同样，一个制衣模式是制作一件特定的服装时可重复使用的一系列样式组。模式这一概念本身并不新奇，新奇的是模式应用于软件开发，特别是设计。

设计模式是一般的设计问题的一个解决方案，这类问题以一组交互类的形式出现，用户根据需要对这些交互类进行定制，以创建一个特定的设计。如图 8-2c 所示，阴影的方框用线连接起来，表示交互类。阴影方框中的白色方框表示为进行特定的设计所必须定制的类。

为了理解模式如何辅助软件开发，考虑下面的例子。假设一个软件工程师希望复用两个已存在的类 **P** 和 **Q**，但是它们的接口不兼容。例如，当 **P** 向 **Q** 发送消息时，传递了 4 个参数，而 **Q** 的接口只期望得到 3 个参数。改变 **P** 或 **Q** 的接口，将出现许多不兼容问题，涉及当前包含 **P** 或 **Q** 的所有应用程序。另一种方法是构造一个类 **A**，它从 **P** 接受带有 4 个参数的消息，再将只带 3 个参数的消息发给 **Q**。（这种类型的类有时称为包装（wrapper。）

上面描述的是一个更普遍问题（即让任意两个不兼容的类能一起工作）的一种特殊解决方案。另外一种解决方案是，设计一个模式：适配器模式。正如类的一个实例是一个对象，适配

器模式的一个实例是一个关于不兼容问题的解决方案，是为所涉及的两个类而定做的。这个模式将在 8.6 节中进一步讨论。

模式与其他模式之间会相互影响。如图 8-2d 所示，位于中间模式的左下方的方块也是一个模式。[Gamma，Helm，Johnson and Vlissides，1995] 中的一个文档编辑案例研究包含了 8 个相交互的模式。在现实中，一个产品的设计很少只包含一个模式。

与工具包和架构一样，如果一个设计模式被复用，那么该设计模式的实现可能也会被复用。另外，分析模式可以辅助分析工作流 [Fowler，1997]。最后，除了模式，还有反模式，在备忘录 8.4 中有相关的介绍。

由于设计模式的重要性，在介绍完设计和实现中的复用之后，8.6 节将再次对其进行讨论。

备忘录 8.4

反模式是一个造成项目失败的实践，例如“分析停滞”（在分析工作流上花费了太多时间和精力）或设计了一个面向对象软件产品，而其中的一个对象却几乎做了所有的工作。写第一本反模式的书的主要动机是，由于有将近 1/3 的软件工程项目被取消，2/3 的软件工程项目的成本超支了 200%，而 80% 以上的软件工程项目注定会以失败告终 [Brown et al.，1998]。

8.5.4　软件体系结构

一个教堂建筑的体系结构可能是罗马式的、哥特式的或巴洛克式的。同样，一个软件产品的体系结构可能是面向对象的、管道和过滤器（UNIX 组件）的或客户 - 服务器（有一个中心服务器，用来为客户机网络提供计算功能和文件存储）的。图 8-2d 描述了一个由一个工具包、一个架构和三个设计模式组成的体系结构。

由于应用于产品整体设计，因此*软件体系结构*（software architecture）领域面临着各种设计问题，包括基于组件的产品组织、产品级的控制结构、通信和同步问题、数据库和数据访问、组件的物理分布、性能以及设计选择等 [Shaw and Garlan，1996]。相应地，与设计模式相比，软件体系结构是一个范围更广泛的概念。

事实上，Shaw 和 Garlan [1996] 称，“抽象地说，软件体系结构包括建造系统的要素的描述及要素之间的相互作用、指导要素进行组合的模式以及对这些模式的约束”。从而，除了许多在前一段落中所列出的事项外，软件体系结构还把模式作为一个子域包含在内。这就是图 8-2d 中把三个设计模式作为一个软件体系结构的组件的原因之一。

当复用软件体系结构时，设计复用的许多优势就变得更突出。现实中实现体系结构复用的方法与软件产品线（software product Line）有关 [Lai，Weiss，and Parnas，1999；Jazayeri，Ran，and van der Linden，2000]，其思想是开发一个软件产品公共的软件体系结构，并在开发一个新产品时实例化这个体系结构。例如，惠普公司生产了很多类型打印机，并且不断开发出新的产品。现在，惠普有一个固件体系结构，对每一个新的打印机模型，都可进行实例化，其结果十分有效。例如，在 1995 年到 1998 年间，为新打印机机型开发固件所需的人时数减少了 1/4，并且开发固件的时间减少了 1/3，同时还增进了复用。对于近期的打印机，固件中 70% 以上的组件可以几乎未加改动地从早期产品复用 [Toft，Coleman，and Ohta，2000]。

体系结构模式（architecture pattern）是实现体系结构复用的另一种方法。一个流行的体系结构模式是模型 - 视图 - 控制器（Model - View - Controller，MVC）体系结构模式。如 5.1 节中所述，设计软件的传统方法是把它分解成三个部分：输入、处理和输出。MVC 模式可以看做是输入 - 处理 - 输出体系结构的一个 GUI 领域的扩展，如图 8-3 所示。视图和控制器提供了 GUI。体系结构可分解为模型、视图和控制器，它允许其中的一个组件改变而独立于其他两个组件，因此加强了可复用性。

MVC 组件	描　述	对 应 于
模块	核心功能、数据	处理
视图	显示信息	输出
控制器	处理用户输入	输入

图 8-3　MVC 模型和输入 - 处理 - 输出模型的构件之间的对应关系

另一个流行的体系结构模式是三层体系结构。*表示逻辑层*（presentation logic tier）接受用户的输入并产生输出，这一层与 GUI 相对应；*业务逻辑层*（business logic tier）包含业务规则处理；*数据访问逻辑层*（data access logic tier）与底层数据库进行通信。另外，这个体系结构模式也允许独立于其他两个组件而改变三个组件中的任何一个。这种独立性就是三层体系结构能够促进复用的一个主要原因。

8.5.5　基于组件的软件工程

基于组件的软件工程（component - based software engineering）的目标是构造一个可复用组件的集合。这样以后就不用每次从头开始，而是通过选择一个标准的体系结构和标准的可复用架构，并且在架构的热点处嵌入标准的可复用代码制品来构建所有软件。也就是说，软件产品是通过组合可复用组件来构造的。理想情况下，这将由一个自动工具来完成。

本章描述由复用代码制品、设计模式和软件体系结构所产生的诸多优点。因此，实现基于组件的软件工程将能够解决软件开发中的大量问题。特别地，它会导致软件生产力和质量以数量级提高，并减少推向市场所需的时间和维护工作。

遗憾的是，现在关于复用的技术描述与这一宏伟目标相距甚远。另外，基于组件的软件构造还有很多挑战，包括组件的定义、标准化和检索等。但是，许多中心的研究人员正在积极工作，努力实现基于组件软件工程所提出的目标［Heineman and Councill，2001］。

8.6　关于设计模式的更多内容

由于设计模式在面向对象软件工程中的重要性，现在更详细地来考察一下设计模式。先从一个小型案例研究开始来阐明适配器设计模式（见 8.5.3 节）。

8.6.1　FLIC 小型案例研究

之前，FLIC（Flintstock Life Insurance Company）公司是按照投保人的年龄和性别来计算保险费的。而最近，FLIC 决定采用不考虑性别的政策，也就是说，保险费仅取决于申请人的年龄。

目前，保险费的计算是通过向 **Applicant** 类的 `computePremium` 方法发送信息来完成的，需要传递的数据是申请人的年龄和性别。但是现在需要构造一个不同的方法，它只取决于申请人的年龄。于是编写了一个新的类 **Neutral Applicant**，保险费的计算通过向该类的 `computeNeutralPremium` 方法发送信息来完成。然而，没有足够的时间来改变整个系统。因此，此时的状况就如图 8-4 所示。

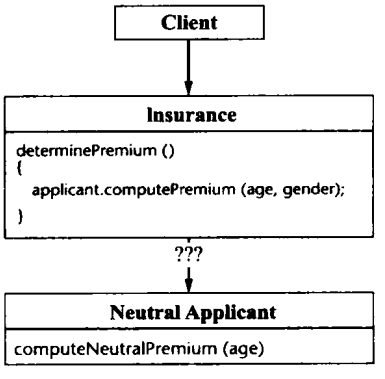


图 8-4　一个显示类之间的接口问题的 UML 图

这里存在着严重的接口问题。第一，一个 **Insurance** 对象传递一个消息给 **Applicant** 类型的一个对象，而不是 **Neutral Applicant** 类型的一个对象。第二，消息传递给 `computePremium` 方

法，而不是 computeNeutralPremium 方法。第三，传递的是年龄和性别参数，而不单是年龄。在图 8-4 中的一个箭头上所标的三个问号表示了这三个接口问题。

为了解决这些问题，需要插入如图 8-5 所示的 **Wrapper** 类。一个 **Insurance** 类的对象传递包含两个参数（年龄和性别）的同一消息给 computePremium，但是，现在消息是传递给 **Wrapper** 类型的一个对象，然后这个对象再传递消息 computeNeutralPremium 给 **NeutralApplicant** 类的一个对象，这时只传递了年龄参数，这样就解决了这个三个接口的问题。

8.6.2　适配器设计模式

通过归纳图 8-5 中的解决方案产生了图 8-6 中的适配器设计模式（Adapter design pattern）[Gamma, Helm, Johnson, and Vlissides, 1995]。在该图中，描述抽象类和抽象（虚拟）方法的名字所使用的字体是 *sans serif italics*（一个抽象类是一个不能被实例化的类，尽管它可用作基类。通常，一个抽象类至少包含一个抽象方法，所谓抽象方法是带有一个接口但没有实现的一个方法）。*request* 方法定义为 ***Abstract Target*** 类的一个抽象方法，然后，在 **Adapter** 类中实现（具体化），并给 **Adaptee** 类的一个对象传递消息 *specificRequest*。这解决了实现的不兼容性问题。**Adapter** 类是抽象类 ***Abstract Target*** 的一个具体子类，在图 8-6 中反映为一个表示继承的空箭头。

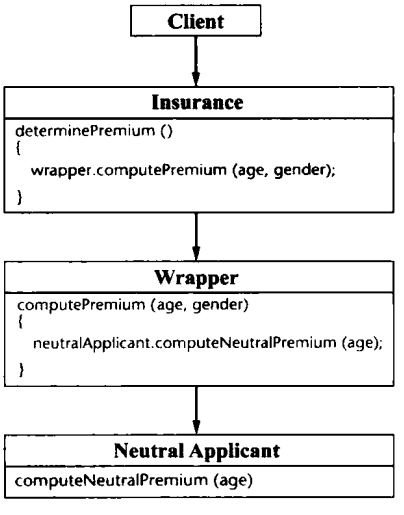


图 8-5　图 8-4 中接口问题的基于包装（Wrapper）的解决方法

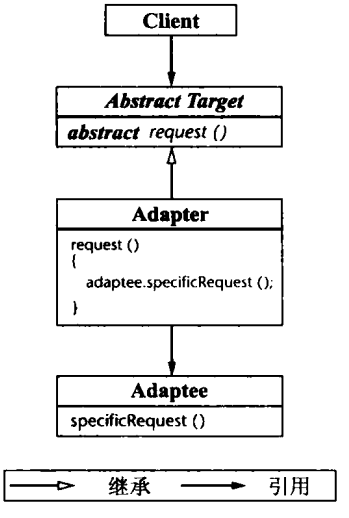


图 8-6　适配器设计模式

图 8-6 中描述了一个通用的使两个接口不兼容的对象进行通信的解决方案。事实上，适配器设计模式的功能更强大。它给对象提供了访问内部实现的方法，在这种方式下，客户就不必和对象的内部实现结构相连接。也就是说，其提供了信息隐藏的所有优点（7.6 节）却不必真正隐藏实现的细节。

现在来考察桥接设计模式。

8.6.3　桥接设计模式

桥接设计模式（Bridge design pattern）的目的是把抽象与实现相分离，使其中一个的改变独立于另一个。桥接模式有时称为驱动（driver）（例如，打印机驱动或视频驱动）。

假定设计的一部分依赖于硬件，而其他部分不是，那么设计可由两部分组成，把设计中依赖于硬件的部分放在桥接的一边，不依赖于硬件的一部分放在另一边。采用这种方法，可把抽

象操作中依赖于硬件的部分分离出来，分离的两部分间有一个“桥接”。现在，如果硬件改变了，对设计和代码的改动仅在于桥接的一边。因此，桥接设计模式可看做是通过封装来实现信息隐藏的一种方法。

如图 8-7 所示，实现独立的部分是在类 **Abstract Conceptualization** 和 **Refined Conceptualization** 中，实现依赖的部分则在类 **Abstract Implementation** 和 **Concrete Implementation** 中。

桥接设计模式对分离依赖于操作系统的部分或依赖于编译的部分也是有用的，因此，它支持多种实现，如图 8-8 所示。

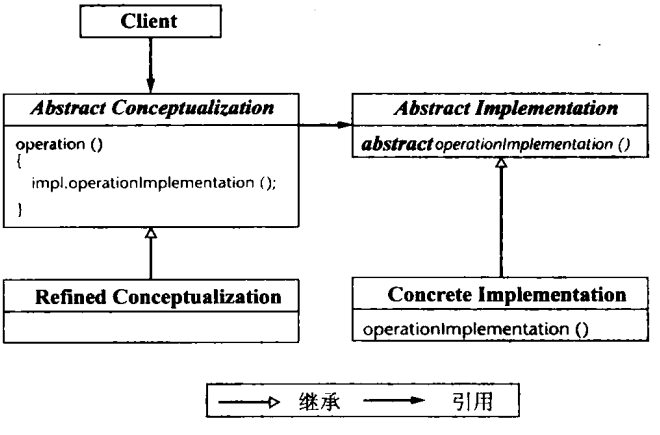


图 8-7　桥接设计模式

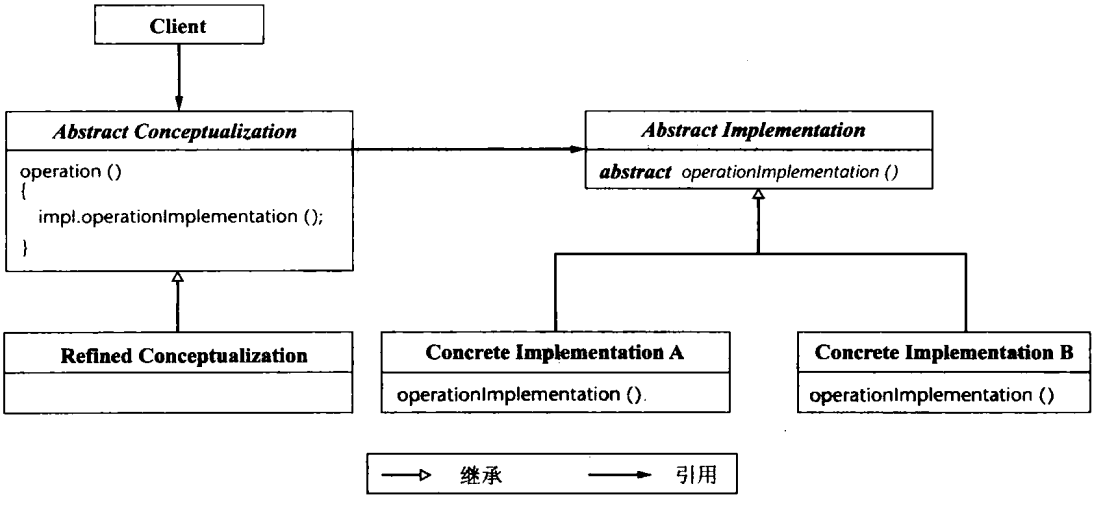


图 8-8　使用桥接设计模式来支持多种实现

8.6.4　迭代器设计模式

一个聚集（aggregate）对象（或容器（container）或聚积（collection））是包含由其他对象所组成的单元的对象，例如，一个连接表或一个哈希表。一个迭代器（iterator）是一个程序结构，它能在不暴露聚集实现的情况下，允许程序员遍历聚集对象的元素。一个迭代器经常作为一个指针（cursor），特别是在数据库环境中。

一个迭代器可以被看做是具有两个主要操作的指针：*元素访问*（element access），即引用聚集

中的一个特定元素；*元素遍历*（element traversal），即通过改变自身以指向聚集中的下一个元素。

迭代器的一个著名例子是电视遥控器。每个遥控器都有一个能使频道号增加 1 的键（通常标记为向上或▲），以及一个能使频道号减少 1 的键（通常标记为向下或▼）。遥控器可增加或减少频道号，电视观众不需指定（甚至不用知道）当前的频道号，更不用说当前频道所播放的节目。也就是说，设备实现了元素遍历而没有暴露聚集的实现。

迭代器设计模式（Iterator design pattern）如图 8-9 所示。一个 Client 对象仅处理 **Abstract Aggregate** 和 **Abstract Iterator**（本质上是一个接口）。Client 对象让 **Abstract Aggregate** 对象去创建一个 **Concrete Aggregate** 对象的迭代器，然后利用返回的 **Concrete Iterator** 去遍历聚集中的元素。**Abstract Aggregate** 对象必须有一个抽象方法 *createIterator*，作为给应用程序中的 Client 对象返回迭代器的一种方法，然而 **Abstract Iterator** 接口只需定义 4 种基本的遍历操作，它们是抽象方法 *first*、*next*、*isDone* 和 *currentItem*。这 5 种方法的实现在下一抽象层（即在 **Concrete Aggregate**(createIterator) 和 **Concrete Iterator**(first、next、isDone 和 currentItem)）中实现。

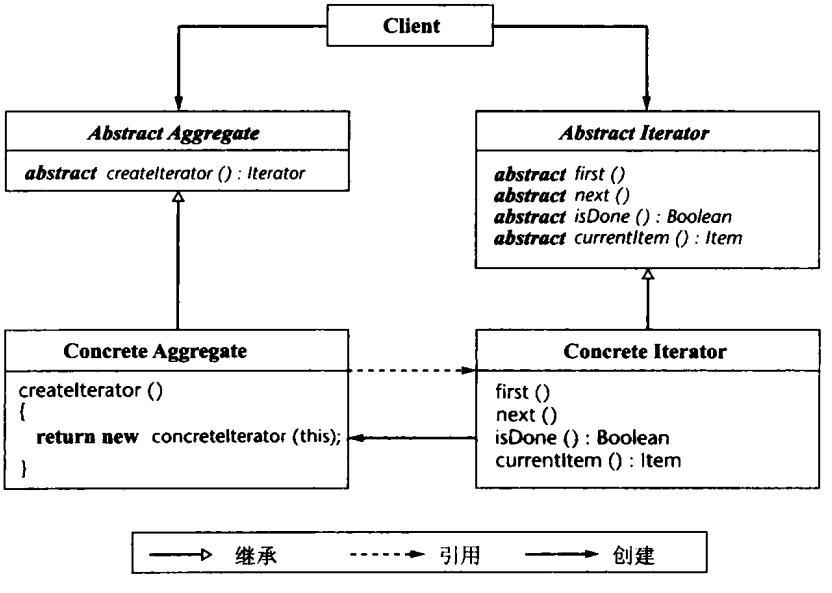


图 8-9　迭代器设计模式

迭代器设计模式的关键方面是迭代器本身能够隐藏元素的实现细节。相应地，可使用一个迭代器去处理聚集中的每个元素，而不依赖于元素容器的实现。

另外，模式允许不同的遍历方法。它甚至允许多种遍历并发执行，并且这些遍历不需要在接口中列出详细的操作，也能实现。这里有一个统一的接口，即 **Abstract Iterator** 中的 4 种抽象操作：*first*、*next*、*isDone* 和 *currentItem*，遍历方法的具体实现在 **Concrete Iterator** 中。

8.6.5　抽象工厂设计模式

假设一个软件组织机构希望建造一个窗口小部件（widget）生成器，即一个用来帮助开发人员开发图形用户接口的工具。开发人员可以使用窗口小部件生成器创造的类集合，此集合定义了应用程序中用到的窗口小部件，而不需从头开始开发各种窗口小部件（如窗口、按钮、菜单、滑动块和滚动条等）。

问题是应用程序（也包括窗口小部件）可能要在多个不同的操作系统上运行，包括 Linux、Mac OS 和 Windows。窗口小部件生成器应该支持所有这三个操作系统。然而，如果窗口小部件生成器中与硬件相关的例程被硬编码（hard - code）为一个应用程序，并运行在一个特定系统

之上，将来改变应用程序（即用运行在另一种操作系统上的不同程序来取代原有所生成的程序时）会很困难。例如，假设应用程序运行在 Linux 上，那么每次需要生成一个菜单来发送消息 `create Linux menu`。然而，如果现在应用程序需要运行在 Mac OS 上，必须用 `create Mac OS menu` 替代 `create Linux menu`。对于一个大型应用程序，从 Linux 转换到 Mac OS 将是既费力又容易出错的过程。

解决方法是使用应用程序与特定操作系统相分离的方式来设计窗口小部件生成器。这可以使用*抽象工厂设计模式*（Abstract Factory design pattern）实现［Gamma, Helm, Johnson, and Vlissides, 1995］。图 8-10 给出了图形用户接口工具包的最终设计。描述抽象类及其抽象（虚拟）方法名字的字体还是用 *sans serif italics*。图 8-10 的顶部是 ***Abstract Widget Factory*** 类。这个抽象类包含许多抽象方法，为简单起见，这里只给出两个抽象方法：*create menu* 和 *create window*。下面的 **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来产生能运行在特定操作系统上的窗口小部件。例如，**Linux Widget Factory** 中的 `create menu` 将创建一个运行在 Linux 上的菜单对象。

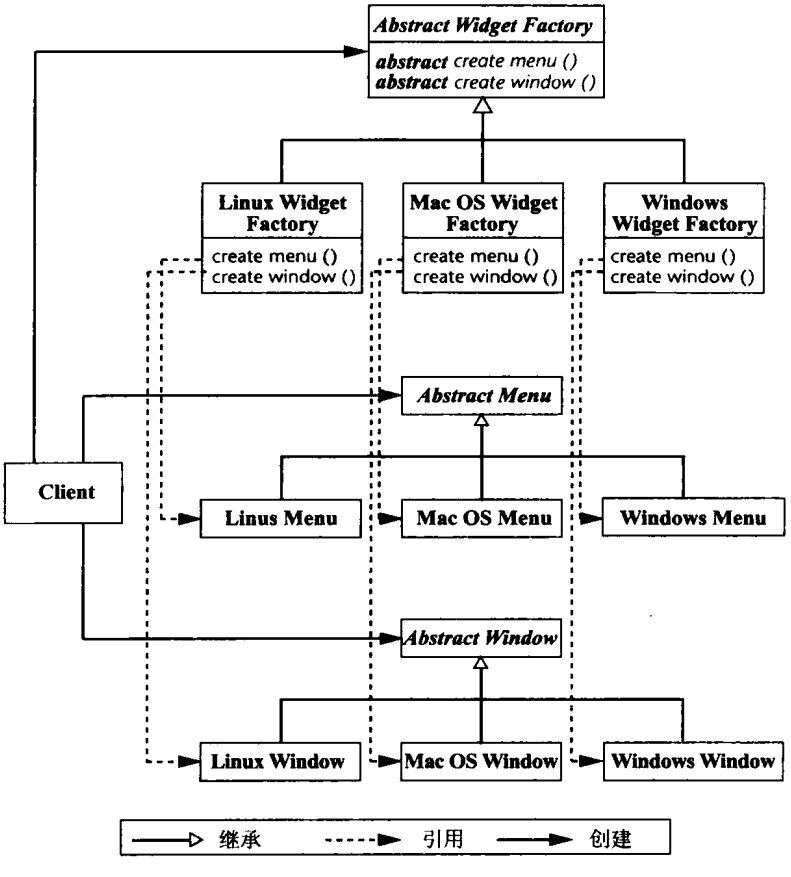


图 8-10　图形用户接口工具包的设计。抽象类的名字和功能用斜体来表示

每个窗口小部件也有抽象类。这里给出两个：***Abstract Menu*** 和 ***Abstract Window***。每个都有具体的子类，对应着三个操作系统中的一个。例如，**Linux Menu** 是 ***Abstract Menu*** 的一个具体子类。具体子类 **Linux Widget Factory** 中的方法 `create menu` 创建了 **Linux Menu** 类型的一个对象。

为了创建一个窗口，应用程序中的一个 **Client** 对象只需传递一个消息给 ***Abstract Widget Factory*** 的抽象方法 `create window`，并且多态性可保证创建正确的窗口小部件。假设应用程序必须运行在 Linux 上。首先，创建一个 **Linux Widget Factory** 类型的 `Widget Factory` 对象。然后把一条给 ***Abstract Widget Factory*** 的虚拟（抽象）方法 `create window` 的消息解释为一条到具体子类 **Linux Widget Factory** 的 `create window` 方法的消息，这个消息所传递的参数是 `Linux`。方法 `create window` 反过来发送一条消息以创建一个 **Linux Window**，这由图 8-10 中最左边的垂直虚线指出。

这个图的重要方面在于，应用程序中的 **Client** 与窗口小部件生成器——类 ***Abstract Widget Factory***、***Abstract Menu*** 和 ***Abstract Window*** 之间的三个接口都是抽象类。这些接口中的任何一个都不特定于任何操作系统，因为抽象类的方法是抽象的（在C++ 中称为虚拟的）。因此，图 8-10的设计确实使应用程序与操作系统相分离。

图 8-10 的设计是图 8-11 中的抽象工厂设计模式的一个实例。为了使用这个模式，特定类取代了如 **Concrete Factory 2** 和 **Product B3** 的一般名字。这就是为什么在图 8-2c 中，一个设计模式的符号表示包含了阴影矩形中的白色矩形，其中白色矩形表示在设计中要复用这个模式所必须提供的细节。

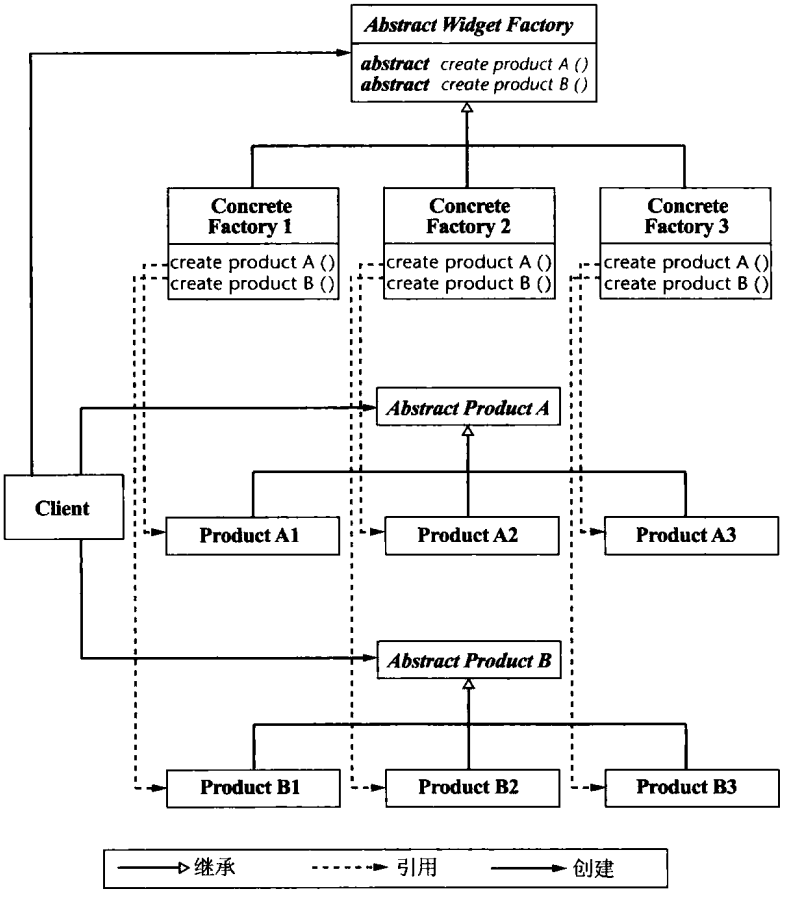


图 8-11　抽象工厂设计模式。抽象类的名字和功能用斜体来表示

8.7　设计模式的范畴

[Gamma, Helm, Johnson, and Vlissides, 1995] 一书明确列出了 23 种设计模式，如图 8-12所示。这些模式可分成三类：创建型模式、结构型模式和行为型模式。创建型设计模式（Creational design pattern）通过创建对象来解决设计问题，例如，抽象工厂模式（8.6.5 节）。结构型设计模式（Structural design pattern）通过确定一个简单方法去认识实体间的关系来解决设计问题，例如，适配器模式（8.6.2 节）和桥接模式（8.6.3 节）。最后，行为型设计模式（behavioral design pattern）通过确定对象间的公共通信模式来解决设计问题，例如，迭代器模式（8.6.4 节）。

创建型模式

抽象工厂：创建一个若干类族的实例（8.6.5 节）
建设者：允许相同的建造过程去创建不同的表达
工厂方法：创建一个若干可能的派生类的实例
原型：被克隆的类
单态：限制一个类的实例化只能为单一实例

结构型模式

适配器：匹配不同类的接口（8.6.2 节）
桥接：将一个抽象从其实现中分离（8.6.3 节）
组合：由相似的类所组成的一个类
装饰：允许另外的行为动态地加到一个类
门面：提供一个简化接口的单一类
共享元：以共享的方式高效地支持大量的细粒度类
代理：功能如同一个接口的一个类

行为型模式

责任链：用类的链来处理请求的一种方法
命令：把一个操作封装在一个类中
解释器：实现特定语言元素的一种方法
迭代器：顺序访问一个聚集中的元素（8.6.4 节）
中介：给一组接口提供一个统一接口
备忘录：捕获并且恢复一个对象的内部状态
观察者：允许运行时观察一个对象的状态
状态：允许一个对象运行中部分地改变它的类型
策略：允许在运行时动态选定一个算法
模板方法：推迟一个算法的实现到它的子类
访问者：向一个类增加新的操作却不改变这个类

图 8-12　在 [Gamma, Helm, Johnson, and Vlissides, 1995] 中列出的 23 种设计模式

目前已经提出了许多其他设计模式，并有着各种不同的分类。这些分类要么适用于一般的设计模式，要么适用于特定的领域，例如，网页或计算机游戏的设计模式。然而，这些可选的模式列表并没有被广泛接受。

8.8　设计模式的优点和缺点

设计模式具有很多优点：

1）如 8.5.3 节中所指出的，设计模式通过解决一个常规的设计问题来促进复用。通过对那

些能进一步加强复用的特征的结合，来加强一个设计模式的可复用性，例如继承。

2）一个设计模式能提供高水平的设计文档，因为模式对设计的抽象性进行详细说明。

3）存在许多设计模式的实现。在这种情况下，没必要对实现设计模式的那部分程序进行编码或归档（当然，程序中那些部分的测试仍然很重要）。

4）如果一个维护程序员对设计模式很熟悉，就会更容易理解一个结合了设计模式程序，即使他以前从来没有见过那个特定程序。

然而，设计模式也有很多缺点：

1）在开发软件产品中使用了［Gamma，Helm，Johnson，and Vlissides，1995］中的23种标准设计模式的程序员可能会得到暗示：正在使用的程序语言的功能不够强大。Norwig［1996］研究了那些模式的C++实现，并发现其中16种模式，至少在每个模式的某些用法上，采用Lisp或Dylan实现比采用C++更简单。

2）一个主要问题是还没有系统的方法来确定何时以及如何应用设计模式。设计模式仍然使用自然语言文本进行非形式化描述。相应地，必须手工确定何时应用模式，并且不能使用CASE工具（第5章）。

3）为了从设计模式中获得最大的好处，需要使用多种交互模式。如8.5.3节所述，［Gamma，Helm，Johnson，and Vlissides，1995］中的文档编辑器的案例研究包含了8种交互模式。正如前面所指出的，仍然没有一个系统的方法来确定何时以及如何使用一个模式，更不用说使用多种交互模式。

4）当对一个采用传统范型构造的软件产品进行维护时，不可能在本质上改进类和对象。一个已经存在的软件产品的模式是无法改进的，不论是传统范型还是面向对象范型，都是如此。

无论如何，采用设计模式总是利大于弊的。此外，一旦在形式化研究和自动设计模式上获得成功，模式就将比目前更容易得到使用。

8.9　复用和交付后的维护

促进使用复用的传统理由是它能缩短开发过程。例如，很多主要软件组织都在尝试把开发新产品所需的时间减半，在这些努力中，复用是一个主要的策略。然而，如图1-3中所反映的，在开发产品上每花费1美元，则在维护该产品上将花费2美元或更多。因此，复用的第二个重要理由就是减少维护产品的时间和费用。事实上，复用对交付后维护的影响比对开发的影响大。

假设现在一个产品40%是由早期产品中的复用组件来组成，并且这个复用均匀地分布在整个产品中。也就是说，规格说明文档的40%是由可复用组件组成的，设计制品的40%、代码制品的40%、用户手册的40%等也是如此。遗憾的是，这并不意味着，开发整个产品的时间将会比没有使用复用所需的时间减少40%。首先，一些组件必须经过处理才能适用于新产品。假设复用组件的1/4作了变动，如果一个组件改变了，那么该组件的文档也必须改变。另外，改变了的组件必须经过测试。其次，如果一个代码制品没有经过修改就复用，那么不需要对该代码制品进行单元测试，但是仍然需要对该代码制品进行集成测试。因此，即使一个产品的30%由不用修改的复用组件组成，另有产品的10%经过修改后复用，在最好情况下，开发整个产品所需的时间只节省27%［Schach，1992］。如图1－3a所示，假设一个软件预算的33%投入到开发中。那么，如果复用使开发成本减少了27%，则该产品在它12～15年的生命周期中由于复用，其整个成本也仅减少了9%，如图8-13所示。

活　动	在生命期中占产品整个成本的百分比（%）	在产品生命期中由于复用而节省的百分比（%）
开发	33	9.3
交付后维护	67	17.9

图 8-13　假设一个新产品的 40% 由可复用的组件组成，
其中 3/4 的复用没有修改，成本节省的平均百分比

类似但冗长的论证可应用于软件过程的交付后维护部分［Schach，1994］。在前一段的假设下，在交付后的维护上，复用使整个成本大约节省了 18%，如图 8-13 所示。显然，复用的主要影响是在交付后的维护上，而不是开发上。根本的原因是复用的组件通常是设计良好的、全面测试过的以及全面地形成文档的，因此简化了三种类型的交付后维护。

如果所给产品的实际复用率比本节中所假设的更低（或更高），复用的好处也会不同。但是整体结果仍然相同：复用对交付后维护的影响比对开发的影响更大。

现在来讨论可移植性。

8.10　可移植性

不断增长的软件开发花费需要寻找一些节约成本的方法。一种方法是确保整个产品容易被改编以运行于各种硬件和操作系统组合上。出售能在其他计算机上运行的产品版本，可以补偿编写产品的部分开销。但是编写可方便运行于其他计算机上的软件的最重要原因是，大约每过 4 年，客户机构都会购买新的硬件，然后所有软件都将转移到新硬件上运行。如果修改一个产品使其可以运行在一个新的计算机上的费用要比从头开始编写它的费用少很多，那么就认为该产品具有可移植性［Mooney，1990］。

可移植性的一个更精确定义是：假设一个产品 P 由编译器 C 进行编译，然后运行在*源计算机*（source computer，即操作系统 O 下的硬件配置 H）上。产品 P 与产品 P' 在功能上相同，但是必须由编译器 C' 进行编译，并且运行于*目标计算机*（target computer，即操作系统 O' 下的硬件配置 H'）上。如果把 P 转换到 P' 的成本比从头开始编写 P' 的成本要少得多，那么称 P 具有*可移植性*。

总体来说，由于不同的硬件配置、操作系统和编译器之间的不兼容性，移植软件的问题显得尤为重要。下面依次考察不兼容性的各个方面。

8.10.1　硬件的不兼容性

目前运行在硬件配置 H 上的产品 P 将安装在硬件配置 H' 上。表面看来这很简单，把 P 从 H 的硬件驱动器复制到 DAT 磁带上，并把它传送给 H' 即可。然而，如果 H' 使用 Zip 驱动器作为备份，这种方法就行不通，因为 Zip 驱动器不能读取 DAT 磁带。

现在假设解决了把产品 P 的源代码物理复制到计算机 H' 的问题。但不能保证 H' 能解释 H 所创建的位模式。目前有很多不同的字符编码，其中最流行的是 EBCDIC（Extended Binary Coded Decimal Interchange Code，扩展二进码编码的十进制交换码）和 ASCII（American Standard Code for Information Interchange，美国信息交换标准码），即 7 位 ISO 编码的美国版本［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。

尽管这些不同是历史造成的（即为不同厂商独立工作的研究人员以不同的开发方式完成相同的工作），但其长期存在却有着明显的经济方面的原因。为了说明这点，考虑下面的虚构情形。MCM 计算机制造商已经卖出成千上万台 MCM－1 型计算机。MCM 公司现在希望设计、制造并向市场推出一种新型计算机 MCM－2，在所有方面，它的功能都比 MCM－1 更强，但是价

格却更低。进一步假设 MCM－1 使用 ASCII 代码和由 4 个 9 位字节构成的 36 位字。现在，MCM 的首席计算机设计师决定 MCM－2 型计算机使用 EBCDIC 码和由 2 个 8 位字节构成的 16 位字。而销售人员不得不告诉当前 MCM－1 用户：MCM－2 比其他竞争者的同款机型便宜 35 000美元，但是把现有的 MCM－1 格式的软件和数据转换为 MCM－2 格式将花费 200 000 美元。不论重新设计 MCM－2 型的科学理由如何正确，推向市场就要考虑确保新型计算机与旧款型的相兼容。然后，销售人员才可以告诉现有的 MCM－1 用户：MCM－2 型计算机不仅比其他竞争者的同款机型便宜 35 000 美元，而且不听建议的顾客如果从其他厂商购买计算机，不仅需要多花费 35 000 美元，而且还需要再花费 200 000 美元来把已有的软件和数据转换成非 MCM 机型的格式。

从前面虚构的情况回到现实世界，目前为止，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在于机型之间的完全兼容性，一个运行在 1964 年所造的 IBM System/360 Model 30 上的产品，不需要改变仍然可以运行在 2007 年所造的 IBM eServer zSeries 990 上。然而，在 OS/360 下，运行于 IBM System/360 Model 30 的产品可能需要相当多的修改，才能运行在完全不同的 2007 机型上，如 Solaris 下的一个 Sun Fire E25K 服务器。这种困难一方面在于硬件的不兼容性，但另一方面可能是由操作系统的不兼容性引起的。

8.10.2　操作系统的不兼容性

任何两种计算机的 JCL（Job Control Language，作业控制语言）通常都有很大的不同。其中一些区别是语法上的，例如，执行载入映像的命令在一台计算机上可能是@ xeq，而在另一台上是//xqt，在第三台则是 . exc。当把一个产品移植到一个不同操作系统上时，语法上的区别可以相对直接地加以处理，将一种 JCL 简单地转换为另一种即可，而其他的区别则更严重。例如，一些操作系统支持虚拟内存。假设某一操作系统允许产品容量最大为1 024MB，但是实际分配给一个特定产品的主存只有 64MB。这时将发生如下情况：用户的产品被分割成大小为 2 048KB的页，但任何时候那些页中只有 32 页可以同时存在于主存中。其余的页则存储在磁盘上，当需要时由虚拟内存的操作系统将它们调进或换出。结果，产品编写时在大小上没有实际的约束。但是，如果一个已经在虚拟内存操作系统上成功实现的产品要转换到一个对产品大小有物理限制的操作系统上，那么可能必须重写整个产品，并且使用覆盖技术链接以确保没有超出产品大小的限制。

8.10.3　数值计算软件的不兼容性

当一个产品从一个机型移植到另一个机型，或者只是使用不同的编译器进行编译，执行算法的结果也可能不同。一个 16 位机型上，即在字的大小为 16 位的计算机上，一个整型通常用一个字（16 位）表示，一个双精度整型用两个相邻字（32 位）表示。遗憾的是，一些语言的实现不包括双精度整型，例如，标准 Pascal 就不包括双精度整型。因此，在用 32 位表示 Pascal 整型的编译器－硬件－操作系统的配置上功能良好的一个产品，在用 16 位表示整型的计算机上就可能不会正确运行。一个明显的解决方案是使用浮点数（**real** 类型）表示比 2^{16} 大的整型数，但这里行不通，因为整型是精确表示的，而浮点数通常只是用尾数（分数）和指数来近似表示一个数。

Java 解决了这个问题，因为 8 种基本数据类型在其中都有详细的定义。例如，**int** 类型总是用一个有符号 32 位整数的补码实现，**float** 类型总是占用 32 位并且符合 ANSI/IEEE（标准）754［1985］对浮点数所作的规定。因此，在 Java 中没有出现需要确保一个数值计算在每种目标硬件－操作系统上都能正确执行的问题。（如果想进一步了解 Java 的设计，参见备忘录 8.5）。然而，如果在 Java 以外的其他语言中执行数值计算，确保数值计算在目标硬件－操作系统上能正

确执行是重要的，并且通常也是困难的。

8.10.4　编译器的不兼容性

如果一个产品以一种几乎没有编译器可用的语言实现，那么很难实现可移植性。如果该产品已经在一个特定语言（如 CLU［Liskow，Snyder，Atkinson，and Schaffert，1977］）中实现，而目标计算机上没有该语言的编译器，那么可能必须用另一种语言重写它。另一方面，如果一个产品用一种流行的面向对象语言（如C++ 或 Java）实现，那么很可能在目标计算机上就可找到该语言的编译器或解释器。

假设一个产品用一种面向对象语言（如标准 Fortran 语言、Fortran 2003——关于 Fortran 2003 名字的起源，参见备忘录 8.6）来编写。理论上，把该产品从一个机型移植到另一机型上应该没有问题，毕竟标准 Fortran 是标准的。遗憾的是，事实并非如此，实际上，没有纯粹标准的 Fortran。即使有一个 Fortran 2003 的 ISO/IEC 标准［ISO/IEC 1539－1，2004］，一个编译器编写者也没有理由完全依照它进行编译器的开发。例如，他可能决定支持 Fortran 2003 中没有的额外特性，以便市场部门能推销一种“新的、扩展的 Fortran 编译器”。相反，一个小型嵌入式微处理器的编译器可能不是标准 Fortran 的完全实现。另外，如果生产编译器有一个最终期限，管理部门可能会决定推出一个不完整的实现，再在以后的修正版本中支持全部标准。假设源计算机上的编译器支持 Fortran 2003 的一个超集。进一步假设目标计算机上的编译器是一个标准 Fortran 2003 的实现。当一个在源计算机上实现的产品要移植到目标计算机上时，使用来自超集的非标准 Fortran 2003 结构的产品的任何部分都必须重编码。因此，为了确保可移植性，程序员应该只使用标准 Fortran 语言特性。

早期的 COBOL 标准是由 CODASYL（COnference on DAta SYstems Languages）开发的，该协会是由美国计算机生产厂商、政府及个人用户组成的。ISO/IEC 第 22 小组委员会的第 1 联合技术委员会现在负责 COBOL 标准的制定［Schricker，2000］。遗憾的是，COBOL 标准没有提倡可移植性。一个 COBOL 标准有 5 年的官方有效期，但是每个后续的标准不必是其先驱的超集。同样令人担忧的是，许多特性由个体自行实现，子集可以称为标准 COBOL，而且对把语言扩展成为超集没有进行限制。现在 COBOL 标准语言 OO－COBOL［ISO/IEC1989，2002］是面向对象的，Fortran 2003［ISO/IEC 1539－1，2004］也是面向对象的。

备忘录 8.5

在 1991 年，Sun Microsystems 的 James Gosling 开发了 Java。在开发该语言时，他经常注视着办公室窗外的一棵橡树。事实上，他经常这么做以至于决定命名他的新语言为 Oak。然而，因为不能注册商标，而没有商标意味着 Sun 将失去该语言的控制权，所以 Sun 不接受他所选择的名字。

为了寻找一个可注册并易于记住的名称，Gosling 的小组建议使用 Java。在 18 世纪，很多进口到英国的咖啡生长在 Java，它是荷属东印度群岛中人口最多的岛屿（现在属于印尼）。结果，Java 成为咖啡的一个俚语，也是软件工程师中第三种流行的饮料。遗憾的是，最受欢迎的前两种碳酸可乐饮料已经注册了商标。

为了理解 Gosling 为什么设计 Java，有必要理解一下他所察觉的 C++ 语言的缺点的来源。为此必须对 C 语言（C++ 的父语言）进行讨论。

在 1972 年，AT&T 贝尔实验室（现在 Lucent Technologies）的 Dennis Ritchie 为了设计系统软件而开发了编程语言 C。该语言的设计相当灵活，例如，其允许对指针变量进行操作，也就是对存储内存地址的变量进行操作。从一般编程人员的角度来看，这存在着一个明显的危险，得到的程序可能会非常不安全，因为计算机可以向任何地方传递它的控制权。C 也没

有具体的数组表示，取而代之的是使用一个指向数组开始地址的指针。因此，超出数组下标这种说法并不是C所固有的。这可能是带来不安全的更深层的原因。

这些以及其他不安全因素在贝尔实验室中并不是什么问题。毕竟，C是由一位资深软件工程师设计出来，并供贝尔实验室中的其他资深软件工程师使用。可以相信这些专家能安全地使用C的强大灵活特性。C的设计中一个基本原则是使用C的人确切地知道他（或她）在做什么。不太熟悉C或经验较少的程序员使用C所产生的软件错误不应该归罪于贝尔实验室。从来没有设想过C会像今天一样，作为一种通用编程语言而被广泛使用。

随着面向对象范型的兴起，基于C开发了大量面向对象的程序语言，包括Object C、Objective C和C++等。这些语言背后的思想是把面向对象构造嵌入C，而那时C已经是一种非常流行的编程语言了。对于编程人员是学习一种基于已熟悉语言的语言更容易，还是学习一种全新的语言更容易这个问题产生了争论。但是，只有一种基于C的面向对象语言被广泛使用，即是由同在AT&T贝尔实验室中的Bjarne Stroustrup所开发的C++。

已经有人提出C++成功的原因在于AT&T（现在是SBC通信的一部分）强大的经济支持。但是，如果公司规模和经济实力是促进一种编程语言发展的相关特点，那么今天我们将都使用PL/I——一种由IBM开发并大力推广的语言。事实上，尽管有IBM的支持，PL/I还是无人知晓。C++成功的真正原因是它是C的一个真正超集。也就是说，不像其他基于C的面向对象编程语言，近乎所有C程序在C++上都有效。因此，各公司意识到他们不用改变任何已有的C软件，就能从C转换到C++。没有任何中断就能从传统范型进步到面向对象范型。在Java著作中经常看到的一条评论是“C++本来应成为Java”。这句话的含义就是，只要Stroustrup像Gosling一样聪明，C++将会成为Java。然而，如果C++不是C的一个真正超集，它已经走上了与其他基于C语言的面向对象编程语言相同的道路，即它基本上已经消失了。在C++已成为一种流行语言后，针对C++的缺点才设计了Java。Java不是C的一个超集，例如，Java没有指针变量。因此，这么说会更确切：“Java能做的事是C++所不可能做到的事”。

最后，要意识到Java跟其他编程语言一样也有自己的缺点，这很重要。另外，在一些领域（如访问规则）中C++要优于Java[Schach,1997]。在未来的几年中，是否C++仍是主要的面向对象编程语言，还是会被Java或别的语言所代替，值得期待。

在1997年11月，各个国家标准委员会（包括ANSI）一致通过了C++标准［ISO/IEC 14882，1998］。该标准于1998年得到最后的批准。

迄今为止，唯一真正成功的语言标准是Ada 83标准，该标准收录在Ada参考手册［ANSI/MIL－STD－1815A，1983］中（关于Ada的背景信息，参见备忘录8.6）。直到1987年底，Ada的名字都是美国政府AJPO（Ada Joint Program Office）的一个注册商标。作为该商标的拥有者，AJPO规定名字Ada只能合法地用于严格遵守标准的语言实现，明确禁止使用语言的子集或超集。AJPO建立了一个验证Ada编译器的机制，一个编译器只有成功通过了验证过程，才称为是一个Ada编译器。因此，该商标作为一种强制遵守标准的手段，从而具有可移植性。

现在名字Ada不再是一个商标，标准的强制执行则通过另一种机制实现。没有经过验证的Ada编译器几乎没有市场。因此，强大的经济驱动迫使Ada编译器开发人员尽力使编译器通过验证，从而保证其符合Ada标准。这些手段已经应用于Ada 83［ANSI/MIL－STD－1815A，1983］和Ada 95［ISO/IEC 8652，1995］，后者是面向对象的。

因为Java是一个完全可移植的语言，所以语言的标准化以及确保标准能被严格遵守是很重要的。Sun Microsystems公司像AJPO一样，使用法律系统来实现标准化。如备忘录8.5所提到

的，Sun 为它的新语言选了一个受版权保护的名字，从而 Sun 能加强版权保护，对所谓的违反者采取法律措施（当微软开发非标准 Java 类时发生了这种情况）。毕竟，可移植性是 Java 当中最强大的特性之一。如果允许多种 Java 版本，Java 的可移植性就会受损；只有每个 Java 编译器对每个 Java 程序的处理完全相同时，Java 才能实现真正的可移植性。在 1997 年，Sun 曾发起“Pure Java”广告运动来影响公众的观念。

Java 1.0 版本最早于 1997 年发布。一系列修正版本随后发布来作为对建议和批评的响应。撰写本书之时最新的版本是 Java J2SE 5.0 版本（Java 2 Platform，Standard Edition）。Java 逐步求精的过程还将继续。当该语言最终稳定时，可能会有一个类似 ANSI 或 ISO 的标准化组织出版一个标准草案，并接收世界各地的反馈。这些反馈将用于整理出正式的 Java 标准。

备忘录 8.6

当程序语言的名称是取首字母的缩写词时，使用大写字符进行拼写。例如，ALGOL（ALGOrithmic Language）、COBOL（COmmon Business Oriented Language）以及 FORTRAN（FORmula TRANslator）。相对地，所有其他程序语言都以一个大写字母开始，名称中的其余字母（如果有）用小写。例如：Ada、C、C++、Java 和 Pascal。Ada 不是一个只取首字母的缩写词，它是以 Lovelace 伯爵夫人（1815—1852）、诗人 Alfred Byron 勋爵的女儿的名字 Ada 命名的。由于 Ada 曾给 Charles Babbage 的差分计算机器编写程序，这使她成为世界上第一个程序员。Pascal 也不是一个只取首字母的缩写词，该语言的名字取自法国数学家和哲学家 Blaise Pascal（1623—1662）。备忘录 8.5 中已经说明了 Java 名称的由来。

有一个例外：Fortran。FORTRAN 标准委员会决定，从 1990 版本开始，该语言的名字从那以后写成 Fortran。

8.11　为什么需要可移植性

看到移植软件上的诸多障碍，读者可能想知道是否值得移植软件。图 8-10 中一个有利于可移植性的论据是：通过将产品移植到一个不同的硬件 - 操作系统配置中，软件成本可能会获得部分补偿。然而出售一个软件的多个变种版本是不太可能的。应用可能非常专业化，并且也没有其他客户会需要该软件。例如，为一个大型汽车出租公司编写的管理信息系统可能完全不适用于其他汽车出租公司。另一方面，软件本身可能给客户提供一个竞争优势，出售产品的副本将等价于经济自杀。根据这些因素，就会考虑，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？

这个问题的答案显然是不。可移植性重要的主要原因是，通常一个软件产品的使用期要比第一次编写软件时所服务的硬件的使用期更长。好的软件产品能有 15 年或更长的使用期，而硬件经常每隔 4 年就更新一次。因此，好的软件在其使用期内会在 3 个或更多的不同硬件配置上实现。

解决这个问题的一种方法是购买向上兼容的硬件。仅有的开支只是硬件的成本，软件不需要改变。然而，在一些情况下，移植产品到完全不同的硬件上在经济上可能更合理。例如，一个产品的第一个版本可能 7 年前已经在一个大型主机上实现了。尽管购买一个新的大型机，软件产品可能不加修改仍可以在其上运行，但是在个人计算机（每个用户桌面上一台计算机）网络上实现产品的多种拷贝可能仍会便宜不少。在这个实例中，如果软件用一种能支持可移植性的方法来编写，那么移植该产品到个人计算机网络中则更经济些。

但是还有其他类型的软件。例如，很多为个人计算机编写软件的公司靠出售 COTS 软件的多个副本来赚钱。例如，一个电子表格软件包的利润很少，不可能弥补开发成本。为了获利，

可能需要卖出 50 000（甚至 500 000）个副本。超过了这个数字之后，额外的销售额就是净利润了。因此，如果产品能较容易地移植到其他类型的硬件上，甚至可以赢利更多。

当然，与所有软件一样，产品不只是代码，还有文档及用户手册。移植电子表格软件包到其他硬件意味着也要改变文档。因此，可移植性还意味着能很容易地改变文档以反映目标配置，而不是从头开始写一个新文档。与编写一个全新的产品相比，移植一个熟悉的已有产品所需要的训练要少很多。因此，应该鼓励可移植性。

现在来介绍实现可移植性的技术。

8.12　实现可移植性的技术

一种实现可移植性的方法是禁止程序员使用当移植到其他计算机上可能会引起问题的结构。例如，一个显然的规则似乎是：用高级编程语言的标准版本来编写所有软件。但是如何来编写一个可移植的操作系统呢？毕竟，编写一个操作系统要用到汇编语言代码。类似地，一个编译器必须为特定的计算机生成目标代码。因此，也不可能完全避免使用所有依赖于实现的组件。

8.12.1　可移植的系统软件

一种更好的技术是对任何依赖于实现的相关部分进行隔离，而不是禁止，这可以避免几乎所有系统软件的重写。这种技术的一个例子是最初的 UNIX 操作系统的构建方法［Johnson and Ritchie，1978］。该操作系统中大约有 9 000 行代码是用 C 编写的，其余 1 000 行代码构成的内核是用汇编语言编写的，后者对每种实现都必须重写。9 000 行 C 语言代码中大约 1 000 行是设备驱动程序，这部分代码每次也必须重写，其余 8 000 行在不同实现之间基本不用改变。

另一种提高操作系统的可移植性的有用技术是使用抽象层次（7.4.1 节）。例如，考虑一个工作站的图形显示程序。一个用户插入类似 `drawLine` 的一条命令到源代码中。编译器编译源代码并将其与图形显示程序连接。运行时，`drawLine` 使工作站按用户要求在屏幕上画出一条线。这可通过使用两层抽象来实现。在上面一层，用一种高级语言编写，解释用户的命令并调用适当的下面一层的代码来执行该命令。如果图形显示程序移植到一种新型工作站上，那么图形显示程序的用户代码或者说上面一层代码不需要改变。然而，下面一层的程序代码必须重写，因为它与实际的硬件接口有关，这里新型工作站的硬件与先前程序包实现所在的工作站是不同的。这种技术也成功地应用于符合 ISO - OSI 模型七层抽象的通信软件的移植上［Tanenbaum，2002］。

8.12.2　可移植的应用软件

应用软件，而不是类似操作系统的系统软件或编译程序，通常使用高级语言编写。13.1 节指出，关于实现语言通常没有什么选择，但是当有可能选择一种语言时，所做的选择应该以成本 - 效益分析为基础（5.2 节）。在成本 - 效益分析中必须考虑的一个因素是可移植性的效果。

在开发一个产品的每一阶段都可以决定生成一个可移植性更好的产品。例如，一些编译器区分大小写字母。对于这种编译器，`thisisAName` 和 `thisisaname` 是不同的变量。但是其他编译器会把它们看成一样的。一个依赖于大小写字母进行区分的产品，会导致在产品移植时出现很难察觉得到的错误。

就像通常对编程语言没有什么选择一样，对操作系统也没有什么选择。然而，如果完全可能，产品运行所在的操作系统应该是一个主流的操作系统。这是一个有利于 UNIX 操作系统的论据。UNIX 已经在很大范围的硬件上实现。此外，UNIX，更精确来说，类 UNIX 操作系统已经在许多大型机操作系统上实现。对于个人计算机，Linux 是否能超过 Windows 成为使用最广

的操作系统仍有待观察。如同使用一种广泛实现的编程语言促进了可移植性一样，使用一种广泛实现的操作系统也会如此。

为了有利于软件从一种基于 UNIX 的系统移植到另一种系统，出现了 POSIX（Portable Operating System Interface for Computer Environments）［NIST 151，1988］。POSIX 对应用程序和 UNIX 操作系统之间的接口进行了标准化。POSIX 也在很多非 UNIX 操作系统上得到实现，增加了应用软件能正常移植的计算机数量。

语言标准在可移植性的实现上发挥着主要作用。如果一个研发机构的编码标准规定只能使用标准结构，那么最终的产品可能更具有可移植性。为了达到这个目标，必须给程序员提供一张编译器所支持的非标准特性列表，而这些特性没有上级经理的批准是禁止使用的。像其他合理的编码标准一样，这种标准可由机器来检查。

通过采用标准 GUI 语言，图形用户界面同样具有可移植性。这样的例子包括 Motif 和 X11。GUI 语言的标准化是 GUI 重要性的不断增加和人机界面可移植性的最终需求所产生的结果。

另外，必须对建造产品所在的操作系统与产品移植的目标操作系统之间的不兼容性进行规划。如果完全可能，操作系统调用应该局限于 1 或 2 个代码制品。不管怎样，每个操作系统调用必须仔细归档。操作系统调用的文档标准应该假定下一个读代码的程序员对现有操作系统不太熟悉，这经常是一个合理的假设。

应该提供安装手册形式的文档以帮助将来的移植。手册指出移植产品时产品的哪些部分必须作变动，以及哪些部分可能作变动。在这两种情况下，必须提供详细的做什么以及如何去做的说明。最后，在其他手册（如用户手册或操作手册）中已做的变动列表也必须显示在安装手册中。

8.12.3　可移植数据

数据可移植性问题可能会很麻烦。硬件不兼容的问题在 8.10.1 节中已经指出。但是，即使解决了这些问题，软件的不兼容性仍然存在。例如，一个索引顺序文件的格式由操作系统确定，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类似文件数据格式之类的信息。在创建该文件所在的特定编译器和操作系统中，文件头格式几乎总是唯一的。当使用数据管理系统时这种情况更糟。

移植数据最安全的方法是构造一个非结构化（顺序的）文件，它最容易移植到目标机器。由这个非结构化文件，可以构造出所需的结构化文件，必须编写两个特定的转换程序，一个运行于源程序机器用来把最初的结构文件转换为顺序文件格式，另一个运行于目标机器用来把顺序文件重新构造成结构化的文件。虽然这个解决方法看起来相当简单，但是当需要完成复杂数据模型之间的转换时，这两个程序并不平凡。

8.12.4　基于 Web 的应用程序

万维网最大的优点之一在于，基于 Web 的应用程序能获得很高的可移植性。首先，利用类似 HTML（Hypertext Markup Language）［HTML，2006］或 XML（Extensible Markup Language）［XML，2003］的语言能使基于 Web 的应用程序具有可移植性，其中 XML 能被任何网络浏览器读取，并且采用 Java applets 后几乎能运行在任何一个客户端上。从程序的其余部分（特别是应用逻辑）分离出 HTML 或 XML 接口能实现一个更深层次的可移植性。然后，最终的应用程序将运行在服务器上，但实际上每个客户端都能通过网页浏览器来访问，包括个人数字助手（PDA）或蜂窝式手机。此外，这样一个应用程序不需要改变访问它的客户端，就能转移植到一个新服务器上。

在编写本书时，并不是所有应用程序都能运行在每个网页浏览器上。例如，一些运行在 In-

ternet Explorer 上的应用程序不能运行于 Firefox 上，因为 Firefox 遵从 W3C（World Wide Web Consortium）标准［W3C，2006］，而 Internet Explorer 没有［Computer Gripes，2004］。然而，随着网络技术的发展，将来有可能实现最高层次的可移植性。

本章最后总结一下复用和可移植性的优点和障碍（如图 8-14 所示），并列出了每一项在哪个小节进行过讨论。

优　点	障　碍
复用	
缩短开发时间（8.1 节）	NIH 综合症（8.2 节）
降低开发费用（8.1 节）	潜在质量问题（8.2 节）
高质量软件（8.1 节）	恢复问题（8.2 节）
缩短维护时间（8.6 节）	
降低维护费用（8.6 节）	使一个组件可复用的费用（机会复用）（8.2 节）
	使一个组件能以后复用的费用（系统复用）（8.2 节）
	法律问题（仅限合同软件）（8.2 节）
	COTS 组件缺少源代码（8.2 节）
可移植性	
大约每 4 年，软件必须转换到新硬件上（8.11 节）	可能的不相容性
	硬件（8.7.1 节）
能出售更多 COTS 软件的拷贝（8.11 节）	操作系统（8.7.2 节）
	数值软件（8.7.3 节）
	编译器（8.7.4 节）
	数据格式（8.9.3 节）

图 8-14　复用和可移植性的优点和障碍，以及相应主题讨论所在的小节

本章回顾

在 8.1 节中讨论了复用。在 8.2 节讨论了各种复用障碍。在 8.3 节提出了两个复用案例研究。在 8.4 节分析了面向对象范型对复用的影响。8.5 节的主题是设计和实现过程中的复用，主题涵盖了架构、模式、软件体系结构和基于组件的软件工程。在 8.6 节更详细地讨论了设计模式，在一个小型案例研究之后（8.6.1 节），8.6.2 节、8.6.3 节、8.6.4 节和 8.6.5 节分别论述了适配器、桥接、迭代器以及抽象工厂设计模式。8.7 节讨论了设计模式的种类。8.8 节讨论了设计模式的优点和缺点。8.9 节讨论了复用对交付后维护的影响。

8.10 节讨论了可移植性。可移植性能被硬件（8.10.1 节）、操作系统（8.10.2 节）、数值软件（8.10.3 节）或编译器（8.10.4 节）所引起的不兼容性所牵制。不过，使所有产品尽可能可移植是非常重要的（8.11 节）。促进可移植性的方法包括使用流行的高级语言、隔离产品的不可移植部分（8.12.1 节）、遵循语言标准（8.12.2 节）以及使用非结构化数据（8.12.3 节）。本章最后讨论了一个基于 Web 的应用程序。

延伸阅读材料

各种复用案例研究可参见［Lanergan and Grasso，1984；Matsumoto，1984，1987；Selby，1989；Prieto－Díaz，1991；Lim，1994；Jézéquel and Meyer，1997；and Toft，Coleman，and Ohta，2000］。［Morisio，Tully，and Ezran，2000］中描述了 4 个欧洲公司成功复用软件的例子。复用的管理在［Lim，1998］中有所描述。一个关于对象恢复和复用的研究计划可参见［Isakowitz and Kauffman，1996］。复用的成本－效益描述可参见［Barnes and Bollinger，1991］，为

将来复用而标识组件的方法可参见［Caldiera and Basili，1991］。Meyer［1996a］分析了面向对象范型如何促进了复用，在［Fichman and Kemerer，1997］中有 4 个复用和对象技术的案例研究。复用的度量在［Poulin，1997］中有所讨论。影响复用程序成功的因素可参见［Morisio，Ezran，and Tully，2002］。［Ravichandran and Rothenberger，2003］中讨论了复用策略。一个评估软件复用选择的综合模型可参见［Tomer et al.，2004］。更多关于复用的论文可参见 2000 年 5 月出版的《IEEE Transactions on Software Engineering》杂志。

关于架构的一个好的信息源是［Lewis et al.，1995］。D' Souza 和 Wills［1999］提出一个基于面向对象架构和组件的开发方法。一系列关于架构的文章参见［Fayad and Johnson，1999；Fayad and Schmidt，1999；and Fayad，Schmidt，and Johnson，1999］。2000 年 10 月出版的《Communications of the ACM》包含有关基于组件架构的文章，包括［Fingar，2000］和［Kobryn，2000］，后者描述了如何使用 UML 来对组件和架构进行建模。通过架构和模式实现复用的讨论可参见［Fach，2001］。

设计模式由 Alexander 在建筑设计中提出，参见［Alexander et al.，1997］。模式理论出现的直接原因可参见［Alexander，1999］。软件设计模式的主要工作可参见［Gamma，Helm，Johnson，and Vlissides，1995］，更新的书是［Vlissides，1998］。分析模式在［Fowler，1997］中有所描述。［Hagge and Lappe，2005］中描述了需求模式。

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。反模式可参见［Brown et al.，1998］。设计嵌入式系统的模式可参见［Pont and Banner，2004］。Vokac［2004］中描述了出错率模式对 500－KLOC 产品的影响。

关于软件体系结构的主要信息资源是［Shaw and Garlan，1996］。软件体系结构上更新的著作可参见［Bosch，2000］和［Bass，Clements，and Kazman，2003］。软件产品线的描述可参见［Jazayeri，Ran，and van der Linden，2000；Knauber Muthig，Schmid，and Widen，2000；Donohoe，2000；and Clements and Northrop，2002］。［Birk et al.，2003］中描述了软件产品线的实践。软件产品线的成本－效益分析可参见［Bockle et al.，2004］。2002 年 7/8 月出版的《IEEE Software》杂志中包含了产品线的文章。

有关基于组件的软件工程的论文在 1998 年 9/10 月出版的《IEEE Software》杂志中可以找到，包括［Weyuker，1998］，它讨论了基于组件的软件测试。Brereton 和 Budgen［2000］讨论了基于组件的软件产品中的关键问题。有关基于组件的软件工程的体验方面的文章包括［Sparling，2000］和［Baster，Konana，and Scott，2001］。基于组件的软件工程的优缺点可参见［Vitharana，2003］。［Heineman and Councill，2001］是一篇很受推荐的基于组件的软件工程的概述性文章。

实现可移植性的策略可参见［Mooney，1990］。UNIX 的可移植性可参见［Johnson and Ritchie，1978］。

习题

8.1　详细说明可复用性和可移植性之间的区别。

8.2　一个代码制品没有经过修改就被复用于一个新产品。这种复用以什么方式来降低产品的整体成本？以什么方式可使成本保持不变？

8.3　假设一个代码制品被复用时只作了一个改变，即将加法操作改为减法操作。这个微小的改变对习题 8.2 节省成本会有什么影响？

8.4　内聚对可复用性的影响是什么？

8.5　耦合对可复用性的影响是什么？

8.6　你刚加入了一个生产污染控制产品的大型公司。这个公司有成百上千由大约 8 000 种不同 Fortran 2003 的类所构成的软件产品。公司雇用你来拟定一个计划：在未来产品中尽可能多地复用这些类。你将给出什么建议？

8.7 考虑一个图书馆自动循环系统。每本书有一个条形码，每个借书者有一张带条形码的卡。当一个借书者想借书时，图书管理员扫描书上的条形码和借书者的卡，并在电脑终端键入C。类似地，当还书时，图书管理员再次进行扫描，并键入R。图书管理员可往书库中增加图书（+）或去掉图书（-）。借书者能到一台终端去确定书库中一个特定作者的所有图书（借书者键入A="作者"）、有一个特定标题的所有图书（T="标题"），或者一个特定主题范围的所有图书（S="主题"）。最后，如果一个借书者想要一本目前已借出的书，图书管理员能在该书上做个标记，这样，当那本书归还后，它将被保留给该借书者（H="书名"）。说明你如何确保可复用的代码制品比率高。

8.8 要求你构建一个产品，用来确定银行的储户报告书是否正确。所需数据包括月初的余额、每张支票的编号、日期和总额、每笔存款的日期和总额以及月末的余额。说明你如何确保本产品中的代码制品能尽可能多地复用到未来的产品中。

8.9 考虑一台自动取款机（ATM）。用户把一张卡放入插槽中，键入一个4位数个人识别号码（PIN）。如果PIN不正确，卡被退回；否则，用户最多能在4个不同银行账号上执行以下操作：

（i）存入任意金额。将打印一张显示日期、存款总额和账号的收据。

（ii）每次提款20美元，最多能提款200美元（账户不能透支）。提款后，除了现金，还将给用户提供一张收据，包含了日期、取款总额和账号。

（iii）确定账号余额。这显示在屏幕上。

（iv）在两个账户之间转账。从被提取的账户中导出的总额不能超出最高限额。用户将得到一张收据，包含了日期、转移的总额和两个账号。

（v）退出。弹出卡。

说明你如何确保本产品中的代码能尽可能多地复用到未来的产品中。

8.10 在软件生命周期中最早什么时候，开发人员能发觉Ariane 5软件（8.3.2节）中的错误？

8.11 在8.5.2节中陈述了"雷锡恩20世纪70年代的COCOL程序逻辑结构是今天面向对象应用架构的传统先驱。"这对技术发展来说意味着什么？

8.12 说明抽象类在图8-10的设计模式中所扮演的角色。

8.13 说明你如何确保图书馆自动循环系统（习题8.7）尽可能地可移植。

8.14 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 说明你如何确保习题8.9的ATM软件尽可能地可移植。

8.16 你所在的公司正在开发一种新型激光器的实时控制系统，这种激光器用于癌症治疗。你负责编写两个汇编器模块。你将如何指导你的小组来确保所产生的代码能尽可能地可移植？

8.17 你负责把一个750 000行的OO-COBOL产品移植到公司的新计算机上。你把源代码复制到新计算机上，当你试图编译它时，发现15 000多条输入/输出语句都用非标准OO-COBOL语法编写，这些在新编译器中已被废弃。现在你将如何处理？

8.18 面向对象范型用什么方法来促进可移植性和可复用性？

8.19 （学期项目）假设附录A中Osric的办公用品和装饰产品是使用面向对象范型开发的。产品的哪些部分能在未来的产品中得到复用？

8.20 （软件工程读物）教老师分发［Tomer et al.，2004］的复印件。为了使用该模型，你需要积累哪些数据？

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.

[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.

[Birk et al. 2003] A. BIRK, G. HELLER, I. JOHN, K. SCHMID, T. VON DER MASSEN, AND K. MULLER, "Product Line Engineering, the State of the Practice," *IEEE Software* **20** (November/December 2003), pp. 52–60.

[Bockle et al., 2004] G. BOCKLE, P. CLEMENTS, J. D. MCGREGOR, D. MUTHIG, AND K. SCHMID, "Calculating ROI for Software Product Lines," *IEEE Software* **21** (May/June 2004), pp. 23–31.

[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.

[Computer Gripes, 2004] "Gripes about Web Sites That Don't Work Well with Firefox," at: www.computergripes.com/firefoxsites.html, 2004.

[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, 2005] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference*, 5th ed., O'Reilly and Associates, Sebastopol, CA, 2005.

[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.

[Hagge and Lappe, 2005] L. HAGGE AND K. LAPPE, "Sharing Requirements Engineering Experience Using Patterns," *IEEE Software* **22** (January/February 2005), pp. 24–31.

[Heineman and Councill, 2001] G. T. HEINEMAN AND W. T. COUNCILL, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, MA, 2001.

[HTML, 2006] "W3C HTML Homepage," at www.w3.org/MarkUp, 2006.

[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, 2004] *Information Technology—Programming Languages—Fortran—Part 1: Base Language*, ISO/IEC 1539–1, International Organization for Standardization, International Electrotechnical Commission, Geneva, 2004.

[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 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.

[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.

[Norušis, 2005] M. J. Norušis, *SPSS 13.0 Guide to Data Analysis*, Prentice Hall, Upper Saddle Valley River, NJ, 2005.

[Norwig, 1996] P. Norwig, "Design Patterns in Dynamic Programming," norvig.com/design-patterns/ppframe.htm/, 1996.

[Pont and Banner, 2004] M. J. Pont and M. P. Banner, "Designing Embedded Systems Using Patterns: A Case Study," *Journal of Systems and Software* 71 (May 2004), pp. 201–13.

[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 in 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.

[Ravichandran and Rothenberger, 2003] T. Ravichandran and M. A. Rothenberger, "Software Reuse Strategies and Component Markets," *Communications of the ACM* **46** (August 2003), pp. 109–14

[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 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.

[Tomer et al., 2004] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi, and S. R. Schach, "Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study," *IEEE Transactions on Software Engineering* **30** (September 2004), pp. 601–12.

[Tracz, 1994] W. Tracz, "Software Reuse Myths Revisited," *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 271–72.

[Vitharana, 2003] P. Vitharana, "Risks and Challenges of Component-Based Software Development," *Communications of the ACM* **46** (August 2003), pp. 67–72.

[Vlissides, 1998] J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, Reading, MA, 1998.

[Vokac, 2004] M. Vokac, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 904–17.

[Weyuker, 1998] E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software* **15** (September/October 1998), pp. 54–59.

[XML, 2003] "Extensible Markup Language (XML)," at www.w3.org/XML/, 2003.

第9章　计划与估算

学习目标

通过本章学习，读者应能：

- 理解计划的重要性。
- 对软件产品制造的规模和成本进行估算。
- 理解对估算进行修正和跟踪的重要性。
- 拟定一份符合 IEEE 标准要求的项目管理计划。

建造合格软件产品并无捷径可循。集成和建造一个大型软件产品需要时间和资源。而且和其他大型建筑工程项目一样，在项目开始阶段就进行精心准备和计划往往是区分项目成功与失败的唯一决定性因素。然而，仅凭一开始的精心计划对于成功还是远远不够的。项目计划与软件测试一样，必须自始至终贯彻到项目的开发过程和维护过程当中。尽管保持项目计划的连续性是必要的，但是计划活动会在规格说明拟定之后、设计活动开始之前达到一个巅峰。此时，需要计算出合理的项目完成时间并进行成本的估算，以生成一份详尽的项目实现计划。

本章将把整个计划（planning）活动分为两类，即贯穿项目始终的计划活动和在规格说明完成之后就必须产生的详细计划。

注释　正如前言说明的，本章内容可以与本书第二部分的内容并行教授。在理解 MSG 基金会实例研究（参见 11.20 节，习题 11.26 和习题 11.27）以及 Osric 办公用品和装饰的学期项目（参见习题 11.23）的软件项目管理计划时，第 9 章的相关内容和知识是必要的。

9.1　计划活动与软件过程

理想情况下，在一个软件过程的最初就必须对整个软件项目进行计划，之后还需要始终遵循这个计划，直到软件产品最终交付给客户使用为止。然而，这一般是不可能的，因为从最初的工作流无法收集到足够的信息来给出完成整个软件项目的合理计划。例如，在需求工作流中，任何类型的计划活动（除了那些仅涉及需求阶段自身的计划）都是徒劳的。

在需求工作流程的末期和分析工作流程的末期，可供软件开发者支配的信息是完全不同的，两者区别类似于一个是草图，而另一个是详细的蓝图。在需求工作流程的末期，开发者最多对客户所需只有一个大概的认识。相反，在分析工作流的末期，客户已经签署了文件，其中精确定义了所要建造的是何物，而开发者也对目标产品的绝大多数方面（但通常还不是全部）有了详细的理解。此时也是整个软件过程中确定精确的项目周期和成本估算的最早时机。

然而，在某些情况下，在规约说明起草之前可能就要求定下项目周期和成本估算。在最坏的情况下，客户可能在 1 ~ 2 个小时的初步讨论之后就会坚持一个报价。图 9-1 说明了问题的严重性。根据［Boehm et al.，2000］所建立的模型，它描

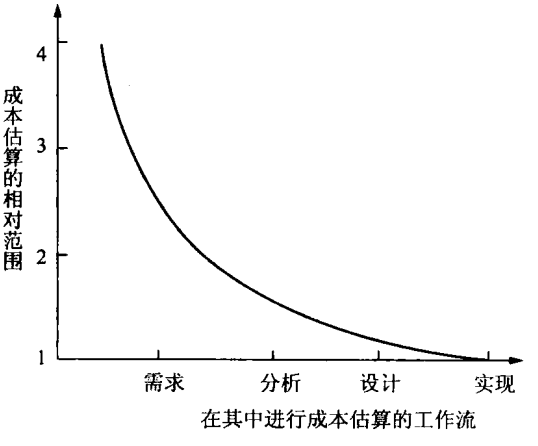


图 9-1　每个生命周期中工作流成本估算的相对范围的估算模型

述了软件生命周期的各个工作流成本估算的相对范围。例如，假设当一个软件产品在通过接受测试后提交给客户使用时，其总体成本是100万美元，如果在项目的需求工作流中进行成本估算，估算值大致在范围（25万美元，400万美元）当中，如图9-2所示。类似地，如果在分析工作流中进行本估算，类似的估算值可能缩小在范围（50万美元，200万美元）间。更进一步，如果在分析工作流末期即合适的时机），进行成本估算，则结果值仍会在一个相对较大的范围（67万美元，150万美元）中。上述4个时机都在图9-2的上、下界限线中标出，注意，图中的纵坐标轴是对数刻度。该模型称为*非确定性锥*（cone of uncertainty）。从图9-1和图9-2可以清楚看到，成本估算并不是一门精确的科学，9.2节会给出一些解释。

非确定性锥模型所依据的数据（包括提交给美国空军电子系统部门（U. S. Air Force Electronic Systems Division）的5个提案［Devenny，1976］以及那时被验证的估算技术）已经陈旧了。然而，图9-1所描述的曲线的整体形状并没有变化太大。因此，一个过早的对项目周期或成本的估算，也就是在客户正式签署规格说明之前进行估算，很可能要比积累了充分的数据后进行估算缺乏准确性。

下面将要介绍的是估算项目周期和成本的技术。本章剩下部分始终假设分析工作流已完成，也就是说，现在所进行的估算和计划工作都是有意义的。

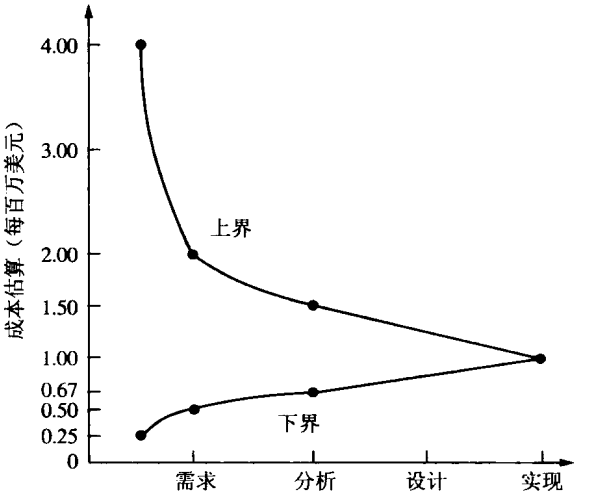


图9-2 花费100万美元建造的软件产品的成本估算范围

9.2 估算项目周期和成本

预算是任何软件项目管理计划中必不可少的组成部分。在进行设计之前，需要让客户知道将要为软件产品支付多少钱。如果开发团队过低估计了实际成本，那么开发公司就要在项目上赔钱。另一方面，如果开发团队过高估计了成本，那么从成本－收益分析或者投资回报的角度出发，客户可能认为没有必要开发这个软件产品，或者会把工作交给估价更合理的其他开发公司。无论如何，成本估算的准确性显然是极其重要的。

实际上，与软件开发有关的成本有两种。第一种是*内部成本*（internal cost），即开发成本；第二种是*外部成本*（external cost），即报给客户的价格（price）。内部成本包括开发团队、项目管理人员和项目相关的支持人员的薪水；开发产品的软硬件成本；以及一般的管理费用，如租金、水电费和高管的薪水等。虽然价格通常是在内部成本的基础上加上利润率，但在某些情况下，经济和心理因素也是很重要的。例如，急需项目的开发者可能会以内部成本向客户报价。而基于报价签订合同时，则会发生不同的情况。如果报价比所有其他公司的报价低但产品质量也明显低于其他公司时，投标可能会被客户拒绝。因此，这时开发团队应该试图给出比竞争对手可能的报价略低（而不是低很多）的一个报价。

对于任何计划，另一个重要部分是估算项目的周期。客户往往也很想知道完整的产品何时能交付使用。如果开发公司无法保证进度，那么轻则丧失信用，重则将被罚款。在任何情况下，负责软件项目计划的管理者都会有很多的解释工作要做。相反，如果开发公司对产品完成所需时间估计过长，那么将很可能失去客户。

遗憾的是，要精确地估算出成本和周期不是件容易的事。不论要精确的掌握成本还是周期，都要考虑太多的变数。一个很大的困难是人为因素。35年前，Sackman及其同事观察到成对程

序员之间的差距可高达 28 比 1 [Sackman, Erikson, and Grant, 1968]。那些认为有经验的程序员总比新手强的人，会轻易地无视这些观察数据，但 Sackman 及其同事对成对程序员进行了比较。他们观察了具有 10 年类似项目经验的两个程序员，并且测量了他们完成像编码和调试这样的任务所花费的时间。然后，他们又观察了两个入行时间差不多短且具有类似教育背景的新手。在比较了最好和最差的表现之后，他们发现，在产品规模上差距是 6 比 1、在产品执行时间上差距是 8 比 1、在开发时间上差距是 9 比 1、在编写代码时间上是 18 比 1 以及在调试时间上差距是 28 比 1。一个尤其给人警示的观察结果是，最好和最坏的表现分别是由两个都具有 11 年经验的程序员做出的。即便是去掉最好和最坏的情况，观察出的差距仍在 5 比 1 左右。显然，根据这些结果，不可能期望软件成本或周期（除非具有所有员工的全部技能的相关详细信息，但这是不现实的）能被估算到任意精度。目前还存在着这样的争论，认为在大型项目中，个体之间的差异可以忽略，但是这恐怕只是理想的假设。只要存在一或两个极为优秀（或者尤其差劲）的成员，就会导致明显的进度偏离，从而对项目预算造成巨大的影响。

另一个影响估算的人为因素是，在一个自由的国家里，不能排除关键的员工在项目期间跳槽。那么为了填补离职人员的空缺，促使新进人员融入团队或者调整剩下的人员来补位，需要花费时间和金钱。无论如何，这都会导致进度发生偏差，并使估算失去控制。

成本估算问题的背后隐含着另一个问题：如何度量一个产品的规模？

9.2.1　产品规模的衡量标准

最常见的产品规模的衡量标准是代码行数。通常用到两个单位：*代码行数*（lines of code, LOC）和*千行交付的源代码指令*（thousand delivered source instructions, KDSI）。但是使用代码行数却存在不少的问题 [van der Poel and Schach, 1983]。

- 编写代码只占整个软件开发工作量很小的一部分，把需求、分析、设计、实现和测试工作流（包括计划和形成文档）所需的时间只表示成一个最终产品源代码行数的函数，看起来显得有些牵强。
- 用两种不同的程序语言实现一个相同的产品会产生两个不同的代码行数。同样，使用 Lisp 这样的语言或者许多其他非过程式语言时，根本就没有代码行数这样的概念。
- 通常没有非常明确的方法来说明如何计算代码行数。是只计算执行代码呢，还是把数据定义的代码也一起算上，另外，注释的内容又该如何处理？如果不计算注释的内容，那么程序员就不愿意花时间去写那些被认为是“非生产性”的注释，但是如果计算注释的内容，那么程序员反过来会写下大量的注释以试图拔高他们表面上的生产力。此外，要如何对作业控制语言的语句进行计数呢？还有，如何对修改和删除的代码计数呢？有时为了提高产品的性能表现，会删去一些代码行。代码复用（8.1 节）也使得代码行计数问题变得更加复杂。如果复用的代码需进行修改，应如何计数？如果代码是从一个父类中继承下来的，又将如何计数呢（7.8 节）？简而言之，度量代码行数相当直观，但如何计算却不是件容易的事。
- 并不是把所有编写的代码都交付给客户。通常有一半的代码可能只是一些用来帮助完成最终产品的工具。
- 假设软件开发人员会使用一些代码生成器，例如，文档生成器、屏幕生成器或者图形用户接口（GUI）生成器等。在开发人员完成设计工作后的几分钟内，这些代码生成器就会自动生成数千行的源代码。
- 只有当软件产品最终完成后，其代码行数才能被确定下来。因此，基于代码行来进行成本估算尤其危险。为了开始估算过程，必须先估算产品完成后可能的代码行数。然后，使用估算出的代码行数再进行成本的估算。不仅成本估算技术本身就存在不确定性，而且如果不确定的成本估算本身的输入值也是不确定的（即还没建造出产品的代码行数），那么这样的估算结果的可靠性不可能太高。

由于把代码行数作为衡量标准非常不可靠，所以就必须考虑其他的衡量标准。一种方法是使用一些基于可度量的量进行衡量，并且这些量在软件过程初期就可以确定。例如，van der Poel 和 Schach［1983］为中等规模的数据处理软件产品的成本估算提出了 FFP 指标（FFP metric）。一个数据处理软件产品的 3 个基本结构要素是文件（files）、信息流（flows）和过程（processes），名称 FFP 就是取自这些要素的首字母。文件的定义是永久驻留在软件产品中的、逻辑上或物理上相关的记录的集合，但不包括事务处理文件和临时文件。信息流是指软件产品和环境（如显示器屏幕和报表）之间的数据接口。过程是功能上定义的对数据的逻辑操作或算术操作，例如，排序、验证及更新操作等。对一个产品，给定其文件数量 *Fi*、信息流数量 *Fl* 以及过程数量 *Pr*，则产品的规模 *S* 和成本 *C* 可表示为：

$$S = Fi + Fl + Pr \tag{9-1}$$

$$C = d \times S \tag{9-2}$$

其中 *d* 是一个常量，各软件公司之间的这个常量各不相同。常量 *d* 是机构内部软件开发过程中效率（efficiency）或生产率（productivity）的一个测量值。软件产品规模只是对文件数、信息流数和过程数量的简单求和，这在分析阶段结束时就可以确定了。而成本则与产品规模成比例，比例系数 *d* 是个常量，其值可由与该软件公司过去曾经开发过的那些产品相关的成本数据的最小平方值来确定。不像基于代码行数的那些衡量标准，使用这个指标，成本在代码编写开始前就可估算出来。

FFP 指标的正确性和可靠性可以用一个基于大量中等规模数据处理软件应用的特殊样本来验证。遗憾的是，这个指标没有把数据库考虑进去，而数据库是许多数据处理软件的基本组件。

一个比较相似的基于功能点（function point）的软件产品规模衡量指标是由 Albrecht［1979］独立提出的。Albrecht 的指标基于输入项的数量 *Inp*、输出项的数量 *Out*、查询的数量 *Inq*、主文件数量 *Maf* 和接口数量 *Inf*。最简形式的功能点 *FP* 由下面等式给出：

$$FP = 4 \times Inp + 5 \times Out + 4 \times Inq + 10 \times Maf + 7 \times Inf \tag{9-3}$$

因为这是产品规模的度量值，所以可以用于成本估算和生产力估算。

等式（9-3）是如下 3 步计算的简化表现形式。第一步，计算未经调整的功能点：

1）产品的每个组件（*Inp*、*Out*、*Inq*、*Maf* 和 *Inf*）按简单、一般和复杂分类（如图 9-3 所示）。

2）每个组件分配一个与其级别相关的功能点数，例如，给一般的输入分配了 4 个功能点，如等式（9-3）所示，但给一个简单的输入分配了 3 个，而给一个复杂的输入分配了 6 个。这步所需的数据如图 9-3 所示。

3）再对分配给每个组件的功能点数进行求和，这样就产生了所谓的未经调整的功能点（Unadjusted Function Point，UFP）。

第二步，计算技术复杂度因子（Technical Complexity Factor，TCF）。它是对 14 个技术因子效果的测量值，技术因子包括：高事务处理率、性能标准（如吞吐量或反应时间）以及在线更新等。全部的技术因子如图 9-4 中所示。这 14 个因子中每一个都分配一个值，从 0（表示“不存在或没有影响”）到 5（表示“自始至终都有重大的影响”）。然后，对这 14 个数求和得到总体影响度（Degree of Influence，DI）。由于 *DI* 的值域是 0 ~ 70，那么 *TCF* 的值域是 0.65 ~ 1.35，*TCF* 可以表示为：

$$TCF = 0.65 + 0.01 \times DI \tag{9-4}$$

第三步，计算功能点的数量 *FP*，其值为：

$$FP = UFP \times TCF \tag{9-5}$$

为测量软件生产力而作的一系列实验证明了用功能点比用 KDSI 更准确。例如，Jones［1987］曾指出，他观察到使用 KDSI 有超过 800% 的误差，而使用功能点仅为 200%，这是一个很能说明问题的解释。

为了展示功能点比代码行更优越，Jones［1987］采用了如图 9-5 所示的例子，分别使用汇编语言和 Ada 语言对同一个产品进行编程，并对结果进行对比。首先，看每人月的 KDSI，这个

指标说明了用汇编语言比用 Ada 语言编程效率高了60%，但这是一个明显错误的结论。要知道，Ada 这样的第三代语言取代汇编语言的原因就是，第三代语言的编码效率更高。现在再看第二个度量指标——每条源代码语句成本。注意到，在这个产品中，一条 Ada 语句相当于 2.8 条汇编语句。使用每条源代码语句成本作为效率的度量指标时，仍然表明汇编语言比 Ada 效率更高，但是，当把每人月功能点作为编程效率的度量指标时，Ada 比汇编的优越性就明显地反映出来了。

		复杂度级别	
组件	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

图 9-3　功能点值表

1. 数据通信	8. 在线更新
2. 分布式数据处理	9. 复杂计算
3. 性能标准	10. 复用性
4. 大量使用的硬件	11. 易安装
5. 高事务处理率	12. 易操作
6. 在线数据入口	13. 可移植性
7. 端用户效率	14. 可维护性

图 9-4　功能点计算的技术因子

	汇编器版本	Ada 版本
源代码规模（KDSI）	70	25
开发成本（美元）	1 043 000	590 000
每人月 KDSI	0.335	0.211
每条源代码语句成本（美元）	14.90	23.60
每人月功能点	1.65	2.92
每个功能点的成本（美元）	3 023	1 170

图 9-5　汇编器和 Ada 产品的比较

注：资料来源于［Jones，1987］（©1987 IEEE）。

从另一方面来说，等式（9-1）和等式（9-2）的功能点和 FFP 指标具有相同的缺点：产品维护通常很难准确测度。当对产品进行维护时，可以在不改变文件数、信息流数和过程数，或者不改变输入数、输出数、查询数、主文件数和接口数的前提下对产品进行主要修改。在这样的情况下，代码行也不适用，举个极端情况的例子，可以把产品中的每一行都换成完全不同的代码，而不会改变整个代码行数。

［Maxwell and Forselius，2000］中已提出至少 40 个 Albrecht 功能点的变种和扩展。Symons［1991］提出的 Mk II 功能点计算方法是一个更精确的计算未经调整的功能点（*UFP*）的方法。它可以把软件分解为一系列组件事务，每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、过程和输出来计算 *UFP* 的值。Mk II 功能点计算方法在全世界范围内被广泛应用［Boehm，1997］。

9.2.2　成本估算技术

尽管估算规模有难度，但是软件开发者必须尽量准确地对项目周期和成本进行估算，这点是至关重要的。同时，还要尽可能多地考虑到那些会影响估算的因素，包括个人的技术能力、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）、开发团队对应用领域的熟悉程度、产品运行的硬件平台以及 CASE 工具的可用性等。另一个因素是所谓的最后期限的影响。如果项目必须在一定时间内完成，那么与没有完成时间限制的情况相比，前者以人月计算的工作量要比后者大很多。这表明周期和成本不是相互独立的，最后期限越短，

工作量就越大，因而相应的成本也就越高。

从前面列出的还不全面的清单来看，估算显然是一个困难的问题，可以使用下面这些方法，它们都或多或少有成功的地方。

1. 基于类比的专家决策

在使用基于类比的专家决策（expert judgment by analogy）方法中，需要咨询一些专家。专家通过将目标产品与自己参与完成的项目进行对比，指出相似点和不同处，从中得出一个估算。例如，专家可以把目标产品与2年前开发的需以批处理方式输入数据的产品进行比较，然而目标产品必须要有在线捕获数据的能力。因为软件公司对要开发的产品的类型很熟悉，专家可以认为开发时间和工作量减少15%。然而，图形用户界面实现起来有点难度，专家认为这会增加25%的时间和工作量。最后，目标产品必须要使用一种团队内成员大都不很熟悉的语言来开发，这又会增加15%的时间和20%的工作量。因为之前的产品花费了12个月的时间和100个人月的工作量来完成，所以专家认为目标产品将要花费15个月和消耗130个人月。

软件机构内的另外两位专家比较了相同的两个产品。一个专家断定目标产品将花费13.5个月以及140人月，而另一位专家的结论是16个月和95人月。如何来协调这三位专家的预测呢？使用**Delphi**技术，不用召开专家集体会议就可达成一致意见，而召开集体会议则可能会产生整体被某个善于游说的成员所左右的负面结果。在该技术中，专家们独立地进行工作。每位专家产生一个估算结果，并给出该结果的理由。然后把这些估算结果和理由分发给所有的专家，让他们在此基础上再做第二次估算。估算和分发的过程一直持续，直到专家们在一个可接受的误差内达成一致。在这个迭代过程内不用召开集体会议。

房产的评估常常就是使用这种基于类比的专家决策进行的。评估人通过把目标房产与相似的最近售出的房子进行对比，得到目标房产的一个评估值。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在3个月前以218 000美元售出。评估人可以这样来推论：房子A比房子B多一个浴室，并且院子也比房子B大5 000平方英尺；房子C跟房子A差不多大，不过屋顶状况不大好，但是多了一个按摩浴缸。经过仔细的思考，评估人最终给房子A估价为215 000美元。

对于软件产品，使用类比法的专家决策就没有房产估价那么精确了。回想一下，前面第一个专家宣称，使用一个不熟悉的开发语言会增加15%的时间和20%的工作量。除非该专家拥有一些经过验证的数据可以确定每个不同之处造成的影响（这极不可能），否则，这种完全可以认为是臆测的结论所引发的错误将会直接导致成本估算的错误。此外，除非专家们有幸保留完整的记忆（或者保留了具体的记录），否则，他们所回忆的以前做过的产品必定是不准确的，以至于使他们的预测变得无效。最后，专家也是人，也会产生偏差，影响他们的预测。同时，一组专家做出的估算结果反映了他们集体的经验，只有这些经验足够的广泛，估算结果才会比较准确。

2. 自底向上的方法

要减少估算整个产品的误差，一种方法是把整个产品分成较小的组件。先对每个组件单独进行周期和成本的估算，再组合起来得到一个整体的数字。这种自底向上的方法（bottom - up approach）的好处在于，为多个较小的组件估算成本比为一个很大的组件估算成本更快，也更准确。此外，估算组件的过程也比大型的、集成的单个产品更具体。这种方法的缺点是一个产品的成本不仅仅是对其组件成本进行求和的结果。

类之间的独立性有助于实施这种自底向上的方法。不过，产品中各种对象之间的交互却使得估算过程变得复杂。

3. 算法化成本估算模型

在这种方法中，需要使用一个衡量指标（如功能点或者FFP指标）作为一个用来确定产品成本的模型的输入值。估算者计算度量指标的值，然后使用该模型计算出周期和成本。从表面

上看，算法化成本估算模型（algorithmic cost estimation model）优于专家们的观点，因为专家是人，正如前面所提到的，他们总会有偏差，可能会忽视已完成产品和目标产品中的某些方面。相反，算法成本估算模型是没有偏差的，因为每个产品都是按相同方式处理的。使用这种模型的危险性在于其估算只是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，在决定给模型的参数赋什么值时通常需要大量的主观判断。例如，估算者经常会不清楚功能点模型的某个技术因子是应评为3还是4。

目前已经提出了许多算法成本估算模型，一些是基于与软件如何开发相关的数学理论，另一些模型是基于统计值，通过研究大量的项目，从数据中确定经验规则。混合模型综合了数学等式、统计模型和专家决策。最重要的一组混合模型是将在9.2.3节中详细描述的Boehm的COCOMO模型。（关于首字母缩写词COCOMO的讨论，参见备忘录9.1。）

9.2.3　中级COCOMO

COCOMO事实上是一个包括三个模型的模型系列，从整体看待产品的宏观估算模型到具体化看待产品的微观估算模型。本节将描述中级COCOMO，其具有中等的复杂度和细节。COCOMO在［Boehm，1981］中有详细的描述，概述可见［Boehm，1984］。

使用中级COCOMO计算开发时间由两个阶段完成。首先，给出一个开发工作量的大致估算，这时必须先估算两个参数：以KDSI计的产品长度和产品的开发模式（对开发产品固有的难度级别的测量）。有三种模式：有组织的（小而直接的）、半分离的（中等规模的）和嵌入式的（复杂的）。

> **备忘录9.1**
>
> COCOMO是在COnstructive COst MOdel（构造性成本模型）中取每个单词的前两个字母而形成的首字母缩写词。与印第安那州的科科莫（KoKomo）的关联只是纯粹的巧合。
>
> COCOMO中的MO意为“模型”，因此不应该使用“COCOMO模型”的称呼，与“ATM机器”和“PIN号码”中的“机器”和“号码”类似，“模型”两字是冗余的。

由这两个参数就可以计算出额定工作量（nominal effort）。例如，如果判断一个项目基本上是直截了当的（有组织的），那么额定工作量（以人月为单位）由下面的等式给出：

$$\text{额定工作量} = 3.2 \times (\text{KDSI})^{1.05} \text{人月} \tag{9-6}$$

其中，常量3.2和1.05是与Boehm开发中级COCOMO使用的组织模式的产品数据最匹配的值。

例如，如果要建造的产品是组织模式的，经估算有12 000行交付的源代码语句（即12KDSI），那么额定工作量是

$$3.2 \times (12)^{1.05} = 43 \text{ 人月}$$

关于这个值的评论请参见备忘录9.2。

> **备忘录9.2**
>
> 对额定工作量值的一个可能反应会是，“如果生成12 000行的可交付源代码指令需要43人月的工作量，那么平均每个月每个程序员只需生成不到300行的代码——我一个晚上就可以写出300多行的代码。”
>
> 一个300行的产品通常只是：300行代码。相反，一个可维护的12 000行产品需要经过生命周期的所有工作流。换句话说，43人月的总工作量要分配到许多活动上，包括写代码。

接下来，这个额定值必须乘以15这个软件开发工作量因子（software development effort

multiplier)。这些因子和它们的值在图 9-6 中给出。每个因子的取值可多达 6 个，例如，根据开发者是否评定产品复杂度为非常低、低、额定（平均）、高、非常高或者特别高，给产品复杂度因子分配了 0.70、0.85、1.00、1.15、1.30 或 1.65 这些值。如图 9-6 所示，当对应的参数为额定时，所有 15 个因子都取值为 1.00。

	成本驱动					
复杂度级别	非常低	低	额定	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	
虚拟机知识 *	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	
要求的开发进度表	1.23	1.08	1.00	1.04	1.10	

* 对于一个给定的软件产品，其基础虚拟机是指硬件和所调用的用以完成任务的软件（如操作系统、数据库管理系统）的组合体

图 9-6　中级 COCOMO 软件开发工作量因子

注：资料来源于［Boehm，1984］（© 1984 IEEE）。

Boehm 给出了一些准则，来帮助开发者确定参数定级是否是额定的，定得低了或者高了。例如，再看一下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（如 **if - then - else**、**do - while**、**case**）组成，那么复杂度可定为“非常低”；如果是嵌套操作，则级别可定为“低”。模块间控制和决策表的加入可将级别提高为“额定”；如果这些操作是高度嵌套的，带有组合断言，包括队列和堆栈，那么级别可定为“高”；出现可重入和递归编程以及固定优先级中断处理会使级别定为“非常高”。最后，使用可动态改变优先级的多个资源的调度和微代码级的控制可确保级别定为“特别高”。这些定级可应用于控制操作。估计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角度来进行。关于这 15 种因子的细节请参考［Boehm，1981］。

为了解释这是如何工作的，［Boehm，1984］给出了一个基于微处理器的用于新的高可靠的电子资金传送网络的通信处理软件的例子，并带有性能、开发计划和接口要求。这个产品符合嵌入式模式的描述，估计长度为 10 000 行交付的源代码指令（10KDSI），所以额定开发工作量为：

$$\text{额定工作量} = 2.8 \times (\text{KDSI})^{1.20} \tag{9-7}$$

（同样，这里的常量 2.8 和 1.20 是符合嵌入式产品数据最好的值。）因为该项目的长度估计为 10KDSI，于是额定工作量是

$$2.8 \times (10)^{1.20} = 44 \text{ 人月}$$

通过将额定工作量乘以 15 个软件开发工作量因子得到估算的开发工作量。在图 9-7 中给出

这些因子的级别和它们的值。按照这些值，乘法的因子是1.35，所以该项目的估算工作量为

$$1.35 \times 44 = 59 \text{ 人月}$$

成本驱动	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存储中的45K（70%）	高	1.06
虚拟机的变更性	基于商用微处理器硬件	额定	1.00
计算机周转时间	2小时的平均周转时间	额定	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	1.00
程序员能力	好的高级程序员	高	0.86
虚拟机知识	6个月	低	1.10
编程语言经验	12个月	额定	1.00
使用现代编程实践	所用技术大多超过了1年	高	0.91
使用软件工具	处于基本的微机工具级别	低	1.10
要求的开发进度表	9个月	额定	1.00

图9-7　微处理器通信软件的中级COCOMO工作量因子级别

注：资料来源于［Boehm，1984］（©1984 IEEE）。

这个数之后会用于其他的公式中，以确定美元成本、开发进度表、阶段和活动分布、计算机成本、年维护成本和其他相关事项，详细内容参见［Boehm，1981］。中级COCOMO是一个完整的算法成本估算模型，在项目计划中给用户提供各种可能的帮助。

中级COCOMO已被一个有着63个项目的样本所验证，这些项目涵盖了广泛的应用领域。将中级COCOMO应用于这个样本的结果是，在差不多68%的时间里，实际值在预测值的20%的范围以内。试图提高这个准确度基本上没有什么意义，因为在大多数机构里，中级COCOMO的输入数据的准确度通常也仅在20%的范围以内。不过，在20世纪80年代，由经验丰富的估算者对模型准确度的验证结果确认了中级COCOMO处于成本估算研究的领先地位，没有其他技术能够像它一样如此一贯准确。

伴随中级COCOMO的主要问题是，其最重要的输入是目标产品的代码行数。如果这个估算不正确，那么该模型的所有的预测值都会不正确。由于中级COCOMO或其他任何的估算技术的预测都可能不准确，管理层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，唯一使用的生命周期模型是瀑布模型。大多数的软件运行于主框架上。像客户－服务器模式和面向对象技术当时还完全不为人所知。相应地，COCOMO也并没有包含这些因素。然而，当更新的技术开始成为普遍接受的软件工程实践时，COCOMO也开始变得不那么准确了。

COCOMO II［Boehm et al.，2000］是1981年的COCOMO的一个主要修订版本。COCOMO II可以处理各种各样的现代软件工程技术，包括面向对象、第2章里提到的各种生命周期模型、快速原型（10.13节）、复用（8.1节）和COTS软件（1.10节）。COCOMO II既富有弹性又成熟可靠。遗憾的是，为了达到这个目标，COCOMO II比原来的COCOMO复杂了很多。因此，想要使用COCOMO II的读者必须要认真地学习［Boehm et al.，2000］，其中给出了COCOMO II与中级COCOMO之间的主要区别的一个概述。

第 1 个区别是，中级 COCOMO 包含基于代码行数（KDSI）的一个总模型，而 COCOMO II 由 3 个不同的模型组成。*应用组合模型*（application composition model）基于对象点（与功能点相似），它应用于早期工作流，此时关于要制造的产品可用的信息很少。然后，当可用信息变得更多时，可以使用*早期设计模型*（early design model），它基于功能点。最后，当开发者得到最多的信息后，就可以使用*后架构模型*（postarchitecture model）。这个模型使用功能点或者代码行数（KDSI）。中级 COCOMO 的输出是成本和规模的估算，COCOMO II 的 3 个模型中每个的输出都是一个成本和规模估算的范围。这样，如果最有可能的工作量的估算值是 E，那么应用组合模型得出范围（$0.50E$，$2.0E$），后架构模型得出范围（$0.80E$，$1.25E$）。这反映出了随 COCOMO II 模型演进而带来的准确性的不断增加。

第 2 个区别在于隐含在 COCOMO 中的工作量模型：

$$\text{工作量} = a \times (\text{规模})^b \tag{9-8}$$

其中 a 和 b 是常量。在中级 COCOMO 中，指数 b 有 3 个不同的取值，依赖于制造产品的模型是有组织的（$b = 1.05$）、半分离的（$b = 1.12$）还是嵌入式的（$b = 1.20$）。在 COCOMO II 中，根据模型里的各种参数，b 的值在 1.01 ~ 1.26 之间变化。这些参数包括对该种产品的熟悉度、过程成熟度级别（3.13 节）、风险解决的程度（2.7 节）和团队的合作程度（4.1 节）。

第 3 个区别是关于复用的假设。中级 COCOMO 假设复用带来的节省与复用数量是直接成比例的。COCOMO II 则考虑到了对复用软件进行小修改会产生不成比例的大的开销的情况（有时，甚至一个很小的修改也需要详细地理解代码，并且改动后的模块的测试成本也相对要大）。

第 4 个区别是，有 17 种成本驱动的乘法因子，而不是中级 COCOMO 中的 15 种。其中的 7 种是全新的，例如，像未来产品中所要求的复用能力、年均人员调整和该产品是否在多个地点进行开发。

COCOMO II 已经在不同领域的 83 个项目中得到了检验。但是，这个模型仍然很新，还不能得出与它的准确性相关的结论，尤其是与其前辈（1981 年的原版 COCOMO）相比，所改善的程度。

9.2.5　跟踪周期和成本估算

产品开发过程中，实际的开发工作量必须持续与预测值进行比较。例如，假设软件开发者使用的预算度量预计分析工作流的*周期*（duration）将持续 3 个月，且需要 7 个人月的工作量。然而，实际上已经持续了 4 个月，并扩充到 10 人月的工作量，而规格说明文档仍旧不能完成。这类偏差可作为发生问题的一个早期警告，必须采取正确的行动。问题可能是大大低估了产品规模，或者开发团队并不如想像的那般胜任。不论是什么原因，都会使周期和成本大大超支，而管理层必须采取恰当的行动来最小化这种影响。

不论预测是使用哪种技术做出的，在整个开发过程中对预测都必须进行仔细跟踪。偏差应归咎于差劲的预测者的度量、低效的软件开发、两者的结合或一些其他的原因。关键的是要尽早检测出偏差，并马上采取正确的行动。此外，在额外信息变得可用时，根据其持续更新预测是很有必要的。

9.3　估算探讨

面向对象范型的运用可使一个产品由一系列相对独立的更小组件（即类）组合而成。这使计划变得更容易了，因为成本和周期的估算对于较小的单元来说计算起来更容易和精确。当然，估算也必须考虑到一个产品不仅仅是其各部分的简单相加。分离的组件之间并不是完全独立的，它们可能相互引用，而这些影响是不能忽视的。

9.2 节中提到的成本和周期的估算技术能否应用于面向对象范型中呢？COCOMO II（9.2.4 节）就是为了应对包括面向对象的现代软件技术而设计的，但是，更早些的度量指标（如功能

点（9.2.1 节）和中级 COCOMO（9.2.3 节））又怎么样呢？在中级 COCOMO 的情况下，需要对其一些成本因子作微小的修改［Pittman，1993］。此外，9.2 节的估算工具在面向对象工程中看起来相当好用（假设没有复用）。复用在面向对象范型中有两种形式：开发过程中已存在组件的复用和有意地对将在未来产品中要复用的组件进行生产（在当前项目过程中）。这两种复用形式都会影响估算过程。开发中的复用显然会减少成本和周期。有公式表明节省量是复用的一个函数［Schach，1994］，但是这些是基于传统范型的结果。目前，在一个面向对象产品中使用复用会对成本和周期造成怎样的影响还不得而知。

现在，转向要在当前项目中复用部件的目的上来。与一个不可复用的相似组件相比，设计、实现、测试和文档化一个可复用的组件可能会花费 3 倍的时间［Pittman，1993］。成本和周期估算必须进行修改以包含这些额外的工作量，并且整个软件项目管理计划也需要进行调整以容纳实现复用带来的影响。因此，两种复用活动的作用正好相反。已存在组件的复用减少开发面向对象产品的总工作量，而设计能在未来产品中复用的组件则增加工作量。我们期望，在长期开发中，类的复用带来的节省量将超过重新开发的成本，并且已经有一些证据支持它［Lim，1994］。

在讨论了估算周期和成本的度量指标之后，接下来讨论软件项目管理计划的组成。

9.4　软件项目管理计划的组成

一个软件项目管理计划（SPMP）包括 3 个主要的部分：要做的工作、做工作所需的资源以及为此支付的所有金钱。本节，将就这 3 个部分进行讨论，术语取自于［IEEE 1058，1998］，这些术语会在 9.5 节中详细讨论。

软件开发需要资源。需要的主要资源是开发软件的人、运行软件的硬件和支持软件（如操作系统、文本编辑器和版本控制软件（5.7 节）等）。

使用像人员这样的资源会随时间发生变化。Norden［1958］指出，对于大型项目，Rayleigh 分布（Rayleigh distribution）是资源消耗 R_c 随时间 t 变化的很好的近似，即

$$R_c = \frac{t}{k^2} e^{-t^2/2k^2} \quad 0 \leqslant t < \infty \qquad (9\text{-}9)$$

参数 k 是一个常量，是消耗达到峰值的时刻，而 e = 2.71828…，是自然对数的底。一个典型的 Rayleigh 曲线如图 9-8 所示，开始资源消耗很小，接着迅速爬升到峰值，然后再以较慢的速率减少。Putnam［1978］分析了 Norden 的研究结果在软件开发上的应用，他发现利用 Rayleigh 分布对人员和其他资源的耗费进行建模，具有一定的准确性。

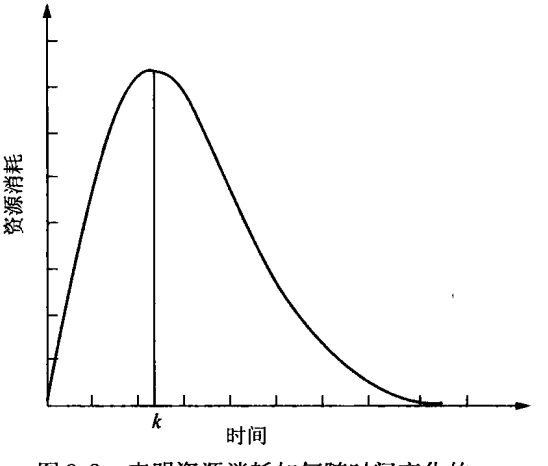


图 9-8　表明资源消耗如何随时间变化的 Rayleigh 曲线

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，还需要考虑下面所述的一些情况：

在实时编程中需要 3 个具有至少 5 年经验的高级程序员，在项目开始后 3 个月其中的 2 个程序员将着手工作，第三个程序员在 6 个月之后开始工作。有 2 个程序员在产品测试开始后逐渐退出，第 3 个程序员则是当交付后维护开始时退出。

资源需求依赖于时间的事实不仅对人员适用，还适用于计算机时间、支持软件、计算机硬件、办公设施，甚至出差旅行等。因此，软件项目管理计划是时间的一个函数。

要做的工作分为两类。第一类是在整个项目中持续进行，与软件开发的任何特定工作流都不相关。这类工作的术语是项目功能（project function），例如，项目管理和质量控制；第二类

是产品开发中与某个特定工作流相关的工作，这类工作的术语是“活动”或“任务”。*活动*（activity）是有明确的开始和结束日期的主要工作单元，它消耗诸如计算机时间或人日这样的“资源”，并产生*工作产品*（work product），例如，预算、设计文档、进度表、源代码或用户手册。接下来，活动由一系列“任务”组成，*任务*（task）是受治于管理职责的最小工作单元。因此，在软件项目管理计划中有3类工作：实行于整个项目中的项目功能、活动（较大的工作单元）和任务（较小的工作单元）。

计划的一个关键方面关系到工作产品的完成。确认工作产品完成的日期在术语里称为*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由团队成员、管理层或客户进行的评审。一个典型的里程碑是完成设计并通过评审的日期。一旦一个工作产品通过了评审，并得到认可，它就成为一个基线，并只能经由正式的程序步骤才能进行修改，如5.8.2节所述。

实际上，对于工作产品来说，不仅仅是产品本身。*工作包*（work package）定义的不仅仅是一个工作产品，还定义了包括人员配备要求、周期、资源、责任人的姓名以及工作产品的验收标准。资金当然是计划的一个至关重要的部分，必须做出一个详细的预算，并将资金作为一个时间的函数分配给项目功能和活动。

如何拟定软件产品计划将在下面进行讨论。

9.5 软件项目管理计划框架

拟定项目管理计划有多种方法，其中最好的一种是IEEE标准1058［1998］，该计划的结构如图9-9所示。



图9-9 IEEE项目管理计划框架

- 该标准是由许多与软件开发相关的主要机构的代表拟定的。输入来自工业界和大学，工作组和评审组的成员有多年拟定项目管理计划的经验。该标准整合了这些经验。
- IEEE项目管理计划是为适用于所有类型的软件产品而设计的，它并不强求特定的生命周

期模型或阐述特定的方法学。这个计划本质上是一种框架，内容可由每个机构根据特定的应用领域、开发团队或技术进行裁剪。

- IEEE 项目管理计划框架支持过程改进，例如，框架中的许多章节反映了像配置管理和度量之类的 CMM 关键过程域（3.13 节）。
- IEEE 项目管理计划框架对于统一过程很适合。例如，计划的其中一节是关于需求控制的，而另一节是关于风险管理的，它们两个都是统一过程的重要方面。

另一方面，尽管 IEEE 标准 1058［1998］宣称 IEEE 项目管理计划适用于所有规模的软件项目，但其中的部分章节与小型软件无关。例如，计划框架的 7.7 节标题为“外包管理计划”，但还从未听说会在小型项目中使用外包。

由此，下面使用两种不同的方式来介绍该计划。首先，9.6 节描述了完整的框架；其次，使用该框架的一个简化版本为附录 E 中的小型项目——MSG 基金会实例研究（10.6 节）做管理计划。

9.6　IEEE 软件项目管理计划

现在来详细描述 IEEE 软件项目管理计划（SPMP）框架，文中的编号和标题与图 9-9 中的条目相对应，使用的各种术语已在 9.4 节中定义。

1 综述。

1.1 项目概述。

1.1.1 意图、范围和目标。对要交付的软件产品的意图和范围以及项目目标，给出一个简要描述。业务需要也包含在这一小节中。

1.1.2 假设和约束。隐含在项目中的任何假设和诸如交付时间、预算、资源和要复用的制品这样的约束都在这里说明。

1.1.3 可交付项目。需要交付给客户的所有事项以及交付时间在这里列出。

1.1.4 进度表和预算概述。整个进度表以及整个预算在这里表述。

1.2 项目管理计划的演化。没有计划可以一次铸造成型。项目管理计划应和其他计划一样按照经验不断更新，并和客户机构及软件开发机构一起共同修改。这一节描述修改计划的正式过程和机制，包括把项目管理计划本身置于配置控制之下的机制。

2 参考材料。项目管理计划中的所有参考文档在此处列出。

3 定义和缩略语。这个信息确保项目管理计划可以被每个人以相同的方式理解。

4 项目组织。

4.1 外部接口。没有项目是在真空中构造的。项目组成员必须与客户机构以及自己所在机构的其他成员进行交互。此外，外包在大型的项目中会被涉及，必须设置项目和其他实体之间的行政和管理边界。

4.2 内部结构。这一节对开发机构本身的结构进行描述。例如，许多软件开发机构分成两类团队：致力于单一项目的开发团队和提供诸如配置管理和质量保证等功能的支持团队。项目团队和支持团队之间的行政和管理边界也必须有明确的定义。

4.3 角色和职责。对于每个诸如质量保证之类的项目功能，以及对于每个诸如软件测试这样的活动，必须标识个人的职责。

5 管理过程计划。

5.1 启动计划。

5.1.1 估算计划。这里列出用于估算项目周期和成本的技术，以及在项目进展过程中跟踪和调整（如果需要的话）这些估算的方法。

5.1.2 人员安置计划。这里列出了项目所需人员数量和类型，还有时间周期。

5.1.3 资源获取计划。这里给出了获取包括硬件、软件、服务合同和行政管理服务这些必

要资源的方法。

5.1.4 项目人员培训计划。成功完成项目所需的所有培训在这一节列出。

5.2 工作计划。

5.2.1 工作活动。在这一节中，详细说明了工作活动，必要时可深入到任务级。

5.2.2 进度分配。通常认为，工作包是相互依赖的，并且对外部事件的依赖更甚，例如，实现工作流紧随设计工作流之后，而在产品测试工作流之前。这一小节将详细说明它们之间的相关依赖。

5.2.3 资源分配。将列出的各种资源分配给合适的项目功能、活动和任务。

5.2.4 预算分配。在这一小节里，整个预算在项目功能、活动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2章所描述，当开发软件产品时，需求频繁的更改。监视和控制需求变更的机制在这一节给出。

5.3.2 进度表控制计划。这一小节列出了测量进度的机制，还描述了实际的进度落后于计划的进度这样的情况发生时所应采取的行动。

5.3.3 预算控制计划。很重要的一点是花费不能超过预算额，本节描述了对实际的成本超过预算成本进行监视的控制机制和情况发生时所应采取的措施。

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、进度表、预算和质量，需要实行报表机制。该机制在这一节中描述。

5.3.6 度量收集计划。如5.3节所述，不对相关的度量进行测量是不可能管理开发过程的，要收集的度量在这一节中列出。

5.4 风险管理计划。风险需要识别、定优先级、减轻和跟踪。风险管理的所有方面在这一节中描述。

5.5 项目结束计划。一旦项目完成，所要进行的活动（包括人员的再分配和制品存档）就要在这一节中描述。

6 技术过程计划。

6.1 过程模型。这一节给出所使用的生命周期模型的一个详细描述。

6.2 方法、工具和技术。所使用的开发方法和编程语言在这里描述。

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，应涵盖的事项包括开发软件产品用到的计算系统（硬件、操作系统、网络和软件），将要在其上运行软件产品的目标计算系统和将使用的CASE工具。

6.4 产品验收计划。为保证完成的软件产品通过其验收测试，必须拟定验收标准，客户必须签署并同意该标准，然后开发者必须保证确实达到了这些标准。这一节描述将要实施验收过程三个阶段的方式。

7 过程支持计划。

7.1 配置管理计划。这一节具体描述了将所有制品纳入配置管理中的方法。

7.2 测试计划。正如软件开发的其他方面一样，测试也需要仔细计划。

7.3 归档计划。这一节包括了对所有种类的文档的描述，不论这些文档是否在项目结束时交付给客户。

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、标准以及评审。

7.5 评审和审计计划。诸如如何进行评审这样的细节在这一节里描述。

7.6 问题解决计划。在开发软件产品的过程中，问题必然会出现。例如，一个设计评审可能会暴露分析工作流的一个致命错误，要求对几乎所有完成了的制品做重要的更改。这一节，对这些问题的处理方法进行了描述。

7.7 外包管理计划。这一节适用于由外包提供某个工作产品，这里描述了选择和管理外包的方法。

7.8 过程改进计划。这一节包括了过程改进策略。

8 附加计划。对于某些项目，附加的部分需要列在计划中。根据 IEEE 框架，它们出现在计划的结尾。附加的部分可以包括保密计划、安全计划、数据变更计划、安装计划和软件产品交付后的维护计划。

9.7 对测试进行计划

一个经常被忽视的 SPMP 组成部分是测试计划。像其他软件开发活动一样，测试必须要有计划。SPMP 必须包含测试的资源，而且在每个工作流中，详细的进度表必须明确指出将要进行的测试。

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 小组必须核查在已完成的产品中已经实现了客户签署的规格说明文档中的每个方面。在这项任务中，对 SQA 小组非常有帮助的方法是要求开发是可跟踪的（3.7 节），即必须有可能把规格说明文档中的每一条声明都联系到设计的一个部分，并且设计的每个部分必须明确地由代码反应出来。达到这一点的一个技术是把规格说明文档中的每个声明编号，并确保这些编号可以由设计和结果代码反应出来。然而，如果测试计划不指明将要这样做，就不大可能适当地对分析、设计和代码制品进行标注，因此，当最后进行产品测试时，确定该产品是一个规格说明的完全实现对于 SQA 小组来说将极其困难。事实上，可跟踪性应该与需求一起开始，需求制品中的每一条声明必须关联到分析制品中的一个部分。

审查的一个强大方面在于审查中所测出的错误的详细列表。假设一个小组正在审查产品的规格说明。如 6.2.3 节所述，使用错误列表的方法有两种。其一，来自该审查中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。偏离前面的准则暗示了项目中的问题。其二，来自当前的规格说明审查的错误统计必须传递到产品的设计和代码审查。毕竟，如果存在大量特定类型的错误，就很有可能无法在规格说明的审查中检测出全部的这种错误，而设计和代码审查则提供了更多的机会来定位残存的这种类型的错误。然而，除非测试计划声明所有错误的细节必须仔细地记录下来，否则这项任务不可能完成。

测试代码模块的一个重要的方法是所谓的黑盒测试（13.10 节），该方法按照基于规格说明的测试用例来运行代码。SQA 团队的成员浏览规格说明，并提出测试用例来检查代码是否符合规格说明文档。提出黑盒测试用例的最佳时机是在分析工作流的最后，此时规格说明文档中的细节依然清晰地保留在审查它的 SQA 团队成员的头脑中。然而，除非测试计划明确地声明黑盒测试用例将在这个时刻选出，否则以后可能只有匆忙给出的寥寥几个黑盒测试用例。也就是说，只有当来自编程小组的压力迫使 SQA 团队批准模块集成为整个产品时，SQA 团队才仓促编出有限数量的测试用例。结果就只会使产品的整体质量受损。

因此，每个测试计划必须指明要做什么测试、什么时候做测试以及如何做测试。这样的一个测试计划是 SPMP 的 7.2 节的必备部分。缺少它，整个产品的质量毫无疑问会受到损害。

9.8 培训需求

在与客户讨论中，涉及培训这一主题时，一个通常的反应是“无需担心培训的问题，产品完成后，才开始对用户进行培训。”这是个不太令人满意的说法，它暗示只有用户才需要培训。事实上，开发团队的成员也需要培训，从软件计划和估算开始就需要进行培训工作。当使用新的软件开发技术时，例如，使用新的设计技术或测试过程，必须对每个使用新技术的团队成员进行培训。

引入面向对象范型需要大量的培训，引入新的硬件或诸如工作站或集成环境（参见 13.23.2 节）之类的软件工具也需要培训。程序员可能需要有关用于开发产品的机器的操作系统和实现

语言方面的培训。文档准备的培训也往往被忽视，结果就是所产生的低质量的文档。计算机操作者当然需要接受某种培训以运行新的产品，如果使用新的硬件，可能还需要更多的培训。

所需的培训可以以多种方式进行。最容易也最不易打断的培训是由同事或顾问进行的内部培训。许多公司提供多种多样的培训课程，夜校也经常提供培训课程。另一个选择是基于万维网的课程学习。

一旦确定了培训需求，并且提出了培训计划，那么培训计划就必须被纳入 SPMP 中。

9.9 文档标准

软件产品的开发伴随着各种各样的文档。Jones 发现，对于一个规模大约为 50KDSI 的 IBM 内部商业产品来说，每 1 000 行代码指令（KDSI）会生成 28 页文档，而对于相同规模的商业软件产品而言，每 KDSI 会生成 66 页文档。2.3 版本的 IMS/360 操作系统的规模差不多有 166KDSI，其中每 KDSI 会生成 157 页文档。文档有多种类型，包括计划、控制、财务和技术［Jones，1986a］等。除了这些文档类型，源代码本身也是一种文档形式，代码中的注释组成了另外的文档。

文档占了软件开发工作量相当大的一部分。一个对 63 个开发项目和 25 个交付后维护项目的调查表明，每当花费 100 小时的时间于代码编写相关的活动，就有 150 小时的时间花费在编写文档相关的活动［Boehm，1981］。而对于大型的 TRW 产品，花费在文档相关活动上的时间与花费在代码相关活动上的时间的比例上升到了 200 小时比 100 小时［Boehm et al.，1984］。

每种类型的文档都需要标准。例如，设计文档的格式一致化能减少小组成员间的误解并有助于 SQA 团队的工作。尽管新员工必须就文档标准进行培训，但是现有员工在机构内部的项目间流动时不需要再次培训。从交付后维护的观点来看，格式一致的代码标准有助于维护程序员理解源代码。标准化对于用户手册显得尤为重要，因为用户手册会被各种人阅读，而其中只有少量是计算机专家。IEEE 业已提出了用户手册的编写标准（软件用户文档的 IEEE 标准 1063）。

作为计划过程的一部分，在软件生产过程中必须为所有要产生的文档建立标准。这些标准需要被纳入 SPMP 之中。

SPMP 中使用的是现存标准，例如，软件测试文档的 ANSI/IEEE 标准［ANSI/IEEE 829，1991］等，这些标准在 SPMP 的第 2 节（参考材料）中列出。若一个标准是为开发工作所特制的，则应放在 6.2 节（方法、工具和技术）中说明。

文档是软件生产工作的一个很重要的方面，从本质意义上来说，产品就是文档，因为没有文档，产品就无法维护。详细地进行文档计划，然后确保遵循计划，是成功进行软件生产至关重要的一个因素。

9.10 计划和估算的 CASE 工具

自动进行中级 COCOMO 和 COCOMO II 的工具有很多。为了加快对应于参数值修改的计算速度，一些中级 COCOMO 的实现是用诸如 Lotus1 -2 -3 或 Excel 之类的电子表格语言写的。而为了开发和更新计划本身，文字处理器是必备的。

信息管理工具对计划也很有帮助。例如，假设一个大型软件机构有 150 名程序员，那么调度工具可以帮助计划者追踪哪些程序员已经分配了特定的任务，哪些程序员相对当前项目是可用的。

还需要更一般的管理信息类型。许多商业上可用的管理工具既可用来协助计划和估算过程，又可用来监控整个开发过程。它们包括 MacProject 和 Microsoft Project。

现在回到文档编制，Javadoc 是一个广泛用于编制 Java 类文档的工具，它利用 Java 源代码中的注释生成 HTML 格式的文档。Doxygen 是一个更为强大的开源工具，它能为不同种类的编程

语言生成文档，并可以运行在许多不同的平台上。

9.11　测试软件项目管理计划

正如本章一开始所指出的那样，一个软件项目管理计划上的错误会给开发组织带来严重的财政问题。至关重要的是开发组织既不能高估、也不能低估项目的成本和周期。因此，在估算提交给客户之前，整个 SPMP 必须经由 SQA 团队核查。对计划进行测试的最好方法是通过计划审查。

计划审查小组必须详细审查 SPMP，加倍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，一旦计划小组成员确定了其估算，周期和成本的估算就应该立即由 SQA 小组的一个成员独立计算出来。

本章回顾

本章主要是讨论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关键组成部分是估算周期和成本（9.2 节）。本章提出几种估算产品规模的度量指标，包括功能点（9.2.1 节）。之后，描述了各种成本估算的度量指标，特别是中级 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组成部分（要做的工作、工作要用到的资源以及为工作支付的金钱）在 9.4 节中解释。一种特别的 SPMP（IEEE 标准）在 9.5 节中进行综述，并在 9.6 节中详述。接下来的章节是有关计划测试（9.7 节）、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。本章的最后介绍了测试软件项目管理计划（9.11 节）。

延伸阅读材料

Weinberg 的 4 卷著作［Weinberg，1992；1993；1994；1997］详细介绍了软件管理的诸多方面，此外，还有［Bennatan，2000］和［Reifer，2000］。管理软件项目的度量在［Weller，1994］中有所论述。

对于面向对象范型的管理，应参考［Pittman，1993］和［Nesi，1998］，需要关于软件项目管理计划的 IEEE1058 标准方面的进一步信息，应该仔细阅读标准［IEEE 1058，1998］。详细计划的需要在［McConnell，2001］中有所描述。

Sackman 的经典作品在［Sackman，Erikson，and Grant，1968］中有所描述，更详细的资料参见［Sackman，1970］。

关于功能点的有用信息可在［Low and Jeffrey，1990］中找到。对功能点的仔细分析以及推荐的改进出现在［Symons，1991］中。对功能点的评价出现在［Kitchenham，1997］中。

功能点的可靠性在［Kemerer and Porter，1992］和［Kemerer，1993］中有所讨论。功能点的优缺点在［Furey and Kitchenham，1997］中描述。功能点的所有方面的综合资料源是［Boehm，1997］。

中级 COCOMO 的理论证明和全部实现细节在［Boehm，1981］中描述，它的简本在［Boehm，1984］中可以找到。COCOMO II 在［Boehm et al.，2000］中描述。强化 COCOMO 预测的方式在［Smith，Hale and Parrish，2001］中描述。

Briand and Wüst［2001］描述了如何估算面向对象产品的开发工作量。面向对象软件产品的规模和缺点的估算在［Cartwright and Shepperd，2000］中有所描述。类点（一种拓展到类的功能点的扩展型）在［Costagliola，Ferrucci，Tortora，and Vitiello，2005］中有所介绍。

软件工作量估算的误差在［Jorgensen and Molokken－Ostvold，2004］中进行了分析。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。其他的生产率的测量在［Kitchenham and Mendes，2004］中讨论。

以第 4 代语言编写的软件规模估算在［Dolado，2000］中提供。《IEEE Software》杂志 2000 年 11/12 月刊包含多篇关于估算的文章。

习题

9.1　为什么一些软件机构会讽刺“milestones”为“millstones”？（提示：在字典里查找 millstone 的形象含义。）

9.2　假定你是 Bronkhorstspruit Software Developers 的一位软件工程师，1 年前，你的经理宣布下一个软件产品将包含 9 个文件、49 个信息流和 92 个过程：

（i）使用 FFP 指标确定产品的规模。

（ii）假定对于 Bronkhorstspruit Software Developers 公司来说，等式（9-2）里的常量 d 被定为 1 003 美元，那么根据 FFP 指标给出的成本估算是多少？

（iii）该产品最近以 123 800 美元的成本完成，那么这意味着你的开发团队的生产率是多少？

9.3　假定目标产品有 7 个简单的输入、2 个一般的输入和 10 个复杂的输入。有 56 个一般的输出、8 个简单的查询、12 个一般的主文件和 17 个复杂的接口。确定未经调整的功能点（UFP）的数量？

9.4　如果习题 9.3 中的产品的总影响度为 49，请确定功能点的数量。

9.5　在你看来，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标准而被广泛应用？

9.6　假定由你负责开发一个有 67KDSI 的嵌入式产品，除了数据库的规模级别非常高，以及软件工具的使用非常低以外，其他都是额定的。请问使用中级 COCOMO，以人月计算的估算工作量将是多少？

9.7　假定由你负责开发两个 33KDSI 的有组织模式的产品，除了产品 P1 具有特别高的复杂度，以及产品 P2 具有特别低的复杂度外，其他都是额定的。为开发产品，你可以指派两个团队。团队 A 具有非常高的分析能力、应用经验和编程能力，此外，团队 A 还具有很高的虚拟机经验和编程语言经验，而团队 B 在这 5 个属性上的级别都很低。

（i）如果团队 A 开发产品 P1，团队 B 开发产品 P2，总的工作量是多少（以人月为单位）？

（ii）如果团队 B 开发产品 P1，团队 A 开发产品 P2，总的工作量是多少（以人月为单位）？

（iii）前面的两种人员分配中哪一个更合理？中级 COCOMO 的预测与你的直觉一致吗？

9.8　假定由你负责开发一个 49KDSI 的有组织模式的产品，并且每个方面都是额定的。

（i）假设成本是每人月 9 900 美元，该项目的估算成本将是多少？

（ii）在项目开始时你的整个开发团队都辞职了，但是幸运的是你有一个具有非常丰富经验和高能力的团队来替代原团队，但是成本将上升到每人月 12 900 美元。你认为人员变化后会赚得（或失去）多少钱？

9.9　假定由你负责开发一个软件，该软件使用一系列新开发的算法为大型货车运输公司计算最划算的路线。通过使用中级 COCOMO，你确定该产品的成本将是 470 000 美元，然而，作为一个检查，你让团队中的一个成员使用功能点来计算工作量。她的报告称，功能点度量预测的成本为 985 000 美元，是你的 COCOMO 预测的 2 倍还多。此时你如何做才好？

9.10　证明：Rayleigh 分布（等式(9-9)）在 $t = k$ 时达到最大值，并求出对应的资源消耗。

9.11　一个产品的交付后维护计划被认为是 IEEE 软件项目管理计划的“额外部分”，但是切记每个重要的产品都需要维护，平均而言，交付后维护的成本大约是开发该产品成本的 2 ~ 3 倍，应该如何证明这个结论？

9.12　为什么软件开发项目会生成那么多文档？

9.13　（学期项目）考虑附录 A 中的 Osric 办公用品和装饰的工程项目，为什么不能单纯地依据附录 A 中的信息估算成本和周期？

9.14　（软件工程读物）教师将分发［Costagliola，Ferrucci，Tortora，and Vitiello，2005］的复印件，讨论你是否认同凭经验确认的类点？

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.

[Costagliola, Ferrucci, Tortora, and Vitiello, 2005] G. COSTAGLIOLA, F. FERRUCCI, G. TORTORA, AND G. VITIELLO, "Class Point: An Approach for the Size Estimation of Object-Oriented Systems," *IEEE Transactions on Software Engineering* **31** (January 2005), pp. 52–74.

[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.

[Jorgensen and Molokken-Ostvold, 2004] M. JORGENSEN AND K. MOLOKKEN-OSTVOLD, "Reasons for Software Effort Estimation Error: Impact of Respondent Role, Information Collection Approach, and Data Analysis Method," *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 993–1007.

[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.

[Kitchenham and Mendes, 2004] B. KITCHENHAM AND E. MENDES, "Software Productivity Measurement Using Multiple Size Measures," *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 1023–35.

[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
软件生命周期工作流

第二部分将深入描述软件生命周期的各个工作流。对每个工作流，将给出与之相适应的活动、CASE 工具、度量标准、测试技术以及该工作流所面临的挑战。

第 10 章将考察需求，该工作流的目的是确定客户真正的需要。这章还将给出各种需求分析技术。

需求一旦确定，下一步就是拟定规格说明文档，这是第 11 章要介绍的内容。这章使用的是面向对象的分析方法。

面向对象设计是第 12 章的主题。

第 13 章讨论实现，涉及的方面包括实现、集成、好的编程实践和编程标准。

第 14 章的标题是“交付后维护”。这章涵盖的主题包括交付后维护的重要性和挑战，并讨论交付后维护管理的一些细节。

第 15 章将提供关于统一建模语言（UML）的更多资料。

学习完第二部分，你应该对软件过程的所有工作流、与每个工作流相关的挑战以及如何应对这些挑战有一个清晰的了解。

第 10 章　需求工作流

学习目标

通过本章学习，读者应能：

- 执行需求工作流。
- 拟订出初始业务模型。
- 拟订出需求。

按时且不超预算地开发出一个产品的机会是很微小的，除非软件开发团队的每个成员都对该软件产品要做什么达成一致的意见。达到这种全体一致性的第一步就是要尽可能准确地分析客户当前的情况。例如，“客户抱怨人工设计系统极其不适，所以他们需要一个计算机辅助设计系统”，这种陈述是不充分的。除非开发团队确实了解现有的人工设计系统存在什么问题，否则，极有可能新的计算机系统的各个方面也同样会“极其不适”。类似地，如果一个个人计算机制造商试图开发一个新的操作系统，第一步应评价公司现有的操作系统，并且认真分析令人不满意的确切原因。举个极端的例子，弄清楚问题是仅存在于那些由于销售业绩差而责怪操作系统的销售经理的脑子里呢，还是确实用户完全不相信现有操作系统的功能性和可靠性，这是非常重要的。只有对当前的状况有了一个清晰的认识，开发团队才能试图回答关键性问题，即新产品究竟能干什么？回答这个问题的过程正是需求工作流的主要目标。

10.1　确定什么是客户所需

一个常见的误解是，在需求工作流中，开发者必须确定什么样的软件是客户想要的。相反，需求工作流的真正目标是确定什么样的软件是客户所需的。问题是许多客户不知道他们需要什么。进一步来讲，即便客户真正了解他们需要什么，也有可能很难精确地把这些想法传达给开发者，因为大多数客户的计算机知识不如开发团队的成员。（想要进一步了解该问题，请参考备忘录 10.1。）

> **备忘录 10.1**
>
> S. I. Hayakawa（1906—1992）是一位来自加利福尼亚的美国参议员，他曾经对一群记者说道：“我知道你们相信自己理解了我所说的事情，但是我不确定你们是否意识到，你们所听到的其实并不是我要表达的真正的意思。”这条辩解也同样适用于需求分析问题。软件工程师听取了他们客户的要求，但是他们所听到的并不是客户所要表达的。
>
> 上述引语曾错误地被认为是美国前总统候选人 George Romney（1907—1995）说的，他曾经在一个新闻发布会上说过：“我并没有说过我没说过它，我只是说我没说我说了它。我希望能澄清这一点。”Romney 的“澄清”突出了需求分析的另一个挑战，客户所说的话很容易被误解。

另一个问题是客户可能并不了解自己的企业正在做什么。例如，如果现有软件系统响应时间过长的真正原因是数据库设计得太差，那么客户要求开发一个运行速度更快的软件是毫无用处的。真正需要做的是在目前的软件产品中重新组织和改善数据的存储方式。重新开发一个软件产品，运行效果还是会和原来一样慢。又如，如果客户经营的是亏损的零售连锁店，那么客户可能会要求一个财务管理信息系统来反映诸如销售量、工资、应付账目和应收账目之类的项

目。但如果亏损的真正原因是商品的损耗（或行窃和雇员监守自盗），信息系统就几乎毫无用武之处。如果真是这样的情况，那么所需的其实是一个库存控制系统而不是一个财务管理信息系统。

乍一看，确定客户需要什么是挺简单的事情，只要开发团队的成员简单询问客户就行了。然而，有两个原因可以解释为什么这个直接的方法通常并不奏效。

首先，正如前面所说的，客户可能并不了解自己的企业正在做什么。不过，客户经常要求得到一个错误的软件产品的主要原因是软件很复杂。对一个软件工程师来说，对一个软件产品及其功能进行形象化描述已经很难了，而这对于并不精通软件工程的客户来说则更加困难。

没有专业的软件开发团队的协助，客户很难了解到需要开发些什么。另一方面，除非能与客户面对面交流，否则专业的软件开发团队也无法找出究竟客户需要什么。

面向对象的方法是从客户和目标产品的未来用户那里获取初始信息，并将这些初始信息作为统一过程的需求工作流的输入［Jacobson，Booch，and Rumbaugh，1999］，这将在 10.2 节中描述。

10.2　需求工作流概述

*需求工作流*的第一步是理解*应用域*（或者简称域），即目标产品进行操作的特定环境。域可以是银行、太空探索、汽车制造或者遥感勘测。一旦开发团队的成员对域理解到一定深度，他们就可以构造出一个业务模型，即使用 UML 图来描述客户的业务过程。业务模型用来确定客户的初始需求是什么，然后就可应用迭代方法。

换句话说，起点是对域的初始了解。用这个信息来构造初始的业务模型，而初始业务模型则用来拟订客户需求的初始集合。然后，根据已知的客户需求，获得一个对于域的更深层次的理解；使用这个知识来精化业务模型，并因此得到客户需求。该迭代一直持续到开发团队对需求集合感到满意为止。此时，迭代就可以停止了。

发现客户的需求的过程叫做*需求引出*（requirement elicitation）（或者叫做*需求获取*（requirement capture））。一旦拟订初始需求集合，对其进行精化和扩展的过程叫做*需求分析*（requirement analysis）。

现在来详细考察这些步骤。

10.3　域的理解

为了引出客户的需求，需求小组的成员必须对应用域（即目标产品被使用的一般领域）十分熟悉。例如，如果不事先熟悉银行业和神经外科，则很难向一个银行家和神经外科医生提出有意义的问题。因此，除非对产品被使用的领域已有经验，需求分析小组每位成员的最初任务就是熟悉应用域。特别重要的一点是，在与客户和目标软件的潜在用户交流时要使用正确的术语。毕竟，除非访问者使用与该域相符的术语，否则很难引起工作在一个特定领域的人的注意。更为重要的是，使用不合适的词语会产生误解，最终导致错误产品的交付。如果需求小组的成员不理解域中术语的细微差别，同样的问题也会发生。例如，对外行来说，*支柱*、*桁条*、*钢桁*和*支撑柱*这些词语看起来是同义的，但对于一个土木工程师来说，它们是不同的术语。如果开发者没有理解土木工程师是如何精确使用这 4 个术语，而土木工程师又假设开发者对这些术语间的区别十分熟悉的话，开发者可能会认为这 4 个术语是等价的，结果就是计算机辅助桥梁设计软件可能包含错误，最后造成桥梁的坍塌。计算机专业人员希望在基于程序的决策作出之前，能有专人仔细察看程序的每个输出。但是，对计算机系统的信任不断增加意味着依赖于这种检查是不明智的。所以，软件开发者若由于术语误解而最后受到玩忽职守的控告并不是牵强附会。

处理术语问题的一种方法是建立一张*术语表*（glossary），这是一张在域中使用的技术词语

列表以及相应的解释。当小组成员致力于尽可能多地学习应用域的相关知识时，可把初次接触到的词条插入术语表中。然后，每当需求小组成员遇到新的术语就更新术语表。通常，可打印出术语表并分发给团队成员或者让他们下载到PDA（如Palm Pilot）上。使用术语表不仅可以减少客户与开发者之间的误解，并且还有助于减少开发团队成员之间的误解。

一旦需求小组熟悉了应用域，下一步就是构建业务模型。

10.4　业务模型

业务模型（business model）是对一个机构的业务过程的描述。例如，银行的业务过程包括收受客户的存款、贷款给客户以及投资等。

构建业务模型的原因之一是业务模型可提供对客户业务的整体了解。通过这个了解，开发者才能建议客户将业务中的哪些部分计算机化。其二，如果任务是要扩展一个已存在的软件产品，那么开发者必须先从整体上理解现存的业务，才能决定如何扩展现存业务，现存产品的哪些部分（如果存在的话）需要进行修改，以及需要添加哪些新的部分。

为了构建业务模型，开发者需要详细了解各种业务过程。这些过程要进行精化，即进行更加详细的分析。有许多不同的技术可以用来获取构建业务模型所需的信息，但主要还是通过访谈。

10.4.1　访谈

需求小组成员需要和客户企业成员一直会见，直到他们确信可以提取出来自客户和目标软件产品未来用户的所有相关信息为止。

问题有两种基本类型。封闭式问题，其要求一个特定的答案。例如，客户可能会被问到公司里有多少销售人员或者响应时间能有多快。开放式问题则用来鼓励访谈的对象畅所欲言，例如，“为什么当前的软件产品不令人满意?”。其可以解释客户的业务手段的诸多方面。如果这个问题是封闭式的，那么可能就无法看清这部分的真相。

类似地，访谈有两种基本类型：程式化的和非程式化的。在*程式化访谈*（structured interview）中，会提出一些特定的、预先计划好的问题，这些问题一般是封闭式的。在*非程式化访谈*（unstructured interview）中，访谈可能从1～2个准备好的封闭式问题开始，但是后续的问题要根据受访谈的对象的回答而提出。许多这样的后续问题可能在实质上是开放式的，以便给进行访谈的人员提供广泛的信息。

同时，访谈过于非程式化也不是一个好的主意。例如，对客户说，“请谈谈你的业务”，就不太可能得出很多相关的信息。换句话说，问题应该以这样一种方式进行：既能鼓励受访谈者给出范围广泛的回答，但又总是在访谈者所需的特定信息的范围内。

做好一个访谈并不是那么容易。首先，访谈者必须熟悉该应用域。其次，若访谈者已经决定遵循客户的需求，那么对客户公司成员进行访谈是没有要点的。无论访谈者先前被告知什么或者通过其他途径知道些什么，每次进行访谈都必须认真听取受访者所说的内容，同时，要坚决克制任何与客户公司或客户及要开发目标产品的潜在用户的需求相关的固有成见。

访谈结束后，访谈者必须准备一份访谈的书面总结报告。明智的做法是发放报告的副本给每一位受访者，他们可能会澄清某些陈述或添加一些被忽视的条目。

10.4.2　其他技术

访谈是获取业务模型信息的主要技术，本节描述了一些其他的可以和访谈结合使用的技术。

一种获得关于客户公司活动信息的方式是给客户公司相关成员发放*调查问卷*（questionnaire）。这项技术在需要确定数百人的意见时很有用。进一步，一个经过客户公司雇员认真思考

后写出的书面答案可能比一个随口说出的答案更准确。不过，由于一个有技巧的访谈者能够认真倾听受访者并提出问题，他所做的非程式化访谈将大大拓展初始的应答，这样的访谈通常比一个经过深思熟虑的问卷调查表能揭示出更多的信息。因为问卷调查表是预先计划好的，于是就无法根据某一个回答，而再提出一个问题。

另一种启发需求的方法是检查客户在业务上使用的各种表格（form）。例如，印刷厂的一张表格可能反映出出版号、纸张轧压尺寸、湿度、油墨温度、纸张张力等信息。这个表格中的各种字段显示了印刷工作的流程及印刷工作中相关步骤的重要性。其他文档（如操作步骤和工作描述）也是准确找出做了什么和如何做的强有力工具。如果使用了软件产品，应仔细学习用户手册，一系列关于当前客户如何从业的不同类型的数据，对确定客户需求相当有帮助。因此，一个优秀的软件专业人员要仔细学习客户文档，视其为有价值的信息源，并导出对客户需求的准确评估。

获取这些信息的另一种方法是通过对用户的*直接观察*，即由需求小组成员观察和记录客户雇员工作时的情况。这个技术的现代应用方式是在工作场所安装*视频磁带摄像机*来记录（在得到观察对象书面同意的前提下）确切发生的事情。它的一个难点在于需要花很长的时间来分析录像带，通常需求小组的一个或多个成员需要花上 1h 回放视频磁带摄像机录下的每小时的录像带。这个时间对于评估观察到的东西来说是额外的。更糟糕的是，这个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视为一种对其个人隐私的不正当侵犯。需求小组要与所有雇员进行全面合作是非常重要的，如果他们感到受威胁或被打扰，就很难获取必要的信息。在引入视频磁带摄像机，或者为此采取其他任何有可能冒犯雇员的行动前，必须仔细考虑可能的风险。

10.4.3　用例

如 3.2 节所述，模型是一系列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，UML 中的 ML 代表“建模语言”）。用于业务建模的最主要的 UML 图是用例图。

用例（use case）是对软件产品本身和软件产品的使用者（*参与者*，actor）之间的交互进行建模。例如，图 10-1 描述了一个来自银行软件产品的用例，其中的两个参与者——*顾客*（Customer）和*出纳员*（Teller）由 UML 的线条图表示。椭圆里的标签描述了该用例所代表的业务行为，在这个实例中是 `Withdraw Money`。

用例的另一种方式是它显示了软件产品与其运行环境之间的交互，换句话说，参与者是软件产品外部世界的一个成员，而用例中的矩形代表了产品本身。

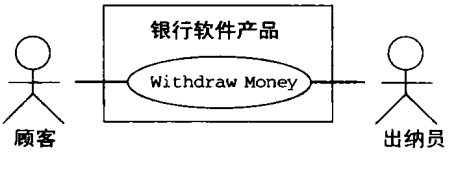


图 10-1　银行软件产品的 `Withdraw Money` 用例

通常，识别参与者很简单。

- 参与者经常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职员，也包括出纳员和经理。
- 通常，参与者扮演的是与软件产品有关的一个角色，如软件产品的使用者。然而，用例的发起者或在用例中起关键作用的某个人也正在扮演一个角色，所以，不论他们是否是软件产品的使用者，他们也被视为一个参与者。10.7 节给出了一个这样的例子。

系统的使用者可以扮演一个以上的角色，例如，银行的顾客可以是*贷款者*（Borrower，当他或她贷款时）或者*出借者*（Lender，当他或她到银行存款时，银行利用顾客存入的钱款进行投资而获利）。反过来说，一个参与者也可以参加到多个用例，例如，一个贷款者可以是 `Borrow Money` 用例、`Pay Interest on Loan` 用例和 `Repay Loan Principal` 用例中的参与者，此外，贷款者这个参与者也代表了成千上万的银行顾客。

参与者不一定是人。回想一下，参与者是软件产品的使用者，在很多情况下，另一个软件

产品也可以是使用者。例如，允许购买者用信用卡付费的电子商务信息系统需要与信用卡公司的信息系统进行交互。也就是说，在电子商务信息系统看来，信用卡公司的信息系统就是一个参与者。类似地，在信用卡公司的信息系统看来，电子商务信息系统也是一个参与者。

正如前面所说的，参与者很容易识别。通常，在面向对象范型的这个部分会产生的唯一困难是过分热心的软件专业人员有时会识别出重叠的参与者。例如，在一个医院软件产品中，一个用例中有*护士*这个参与者，而另一个不同的用例中有*医务人员*这个参与者就不是个好主意，因为所有的护士都是医务人员，而一些医务人员（如理疗师）不是护士。这时定义两个参与者——理疗师和护士会好些，或者，把医务人员这个参与者定义成两个专业：理疗师和护士，如图 10-2 所示。在 7.7 节曾指出，继承是泛化的一个特例，泛化在 7.7 节的类里有应用。图 10-2 也示意了如何把泛化应用到参与者上。

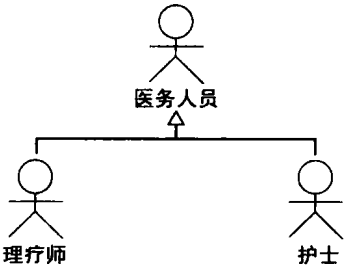


图 10-2　医务人员的泛化

10.5　初始需求

为确定客户的需求，先基于初始业务模型拟订初始需求，然后，在同客户进一步讨论的基础上，精化对于域的理解和业务模型，同时对需求进行精化。

需求是动态的。也就是说，不仅需求本身会多变，开发团队、客户和未来用户对于需求的态度也是多变的。例如，出现在开发团队面前的某项特定需求最初可能是可选的。经进一步分析，这个需求现在看上去可能变得非常重要了。但是，经过同客户的讨论后，这个需求被舍弃了。为处理这些频繁的变化，最好是维护一张可能的需求列表，再附上已经得到开发团队成员一致同意并且经客户认可的需求用例。

很重要的一点是，牢记面向对象范型是迭代的，因此，术语表、业务模型或者需求也可能要随时进行调整。尤其是，各种事件（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）都可能引发对需求列表的增加、对需求列表中已存在事项的修改以及从需求列表中删除某些事项。任何这样的改变都可能引起对业务模型进行相应的改变。

需求可分为两类：功能性需求和非功能性需求。*功能性需求*（functional requirement）指定了目标产品必须能够执行的行为。功能性需求通常用输入和输出来表示：给定一个特定的输入，功能性需求规定输出必须是什么样的。相反地，*非功能性需求*（nonfunctional requirement）指定了目标产品本身的特性，如*平台约束*（platfom contraint，“该软件产品应运行在 Linux 下”）、*响应时间*（reponse time，“平均情况下，3B 类型队列应在 2.5 秒内收到应答”）或*可靠性*（“软件产品应在 99.5% 以上的时间里可运行”）。

功能性需求在需求工作流和分析工作流进行时进行处理，而一些非功能性需求需要等到设计工作流才能处理。原因在于，为了处理某些非功能性需求，需要详细了解目标产品的具体情况，而这些通常要在需求工作流和分析工作流结束之后才能获得（参见习题 10.1 和习题 10.2）。但是，只要有可能，非功能性需求也应该在需求工作流和分析工作流阶段进行处理。

现在用一个运行实例来阐明需求工作流。

10.6　对应用域的初始理解：MSG 基金会实例研究

当 Martha Stockton Greengage 女士 87 岁去世时，她把全部 23 亿美元的遗产捐赠给慈善机构。她希望建立 Martha Stockton Greengage（MSG）基金，通过提供低息贷款来帮助年轻的夫妇购置自己的房产。

为了减少运作费用，MSG 基金会的理事们正为基金会的计算机化进行调查。由于没有一位

理事有计算机方面的经验，他们决定委任一个小型软件开发机构实现一个试验性项目——一个可以通过计算来确定每周有多少钱款可用于购买房产的软件产品。

按照惯例，第一步是理解应用域，在这个实例中是房产抵押贷款。没有多少人承担得起一次性支付现金来购买房子，一般是使用储蓄支付购买价格的一小部分，然后通过贷款支付剩下的钱。这类使用真实的房产做抵押以保证贷款安全性的借贷叫做抵押贷款（mortgage）（参见备忘录 10.2）。

备忘录 10.2

你是否曾经为 mortgage 这个词的发音是“more gidge”，重音在第一个音节上，而感到奇怪？这个词最早出现在 14 世纪的中世纪英语中，来源于古法语的“mort”（意为“死亡”）和德语的“gage”（意为“一份抵押”，即一个如果贷款无法偿还就可以没收财产的保证）。奇怪的是，mortgage 是一个具有两个不同意义的“死亡抵押”。如果贷款无法还清，那么财产就会被没收，或者说，财产对借款人而言永远“死亡”了。但是如果偿还了贷款，那么为了偿还而作的保证就死亡了。这个二义的解释最早由英国法官 Edward Coke 爵士（1552—1634）给出。

那么这个奇怪的发音是什么回事呢？法语中像 mort 这样的词的最后一个字母是不发音的，因此就读作“more”。而后缀-age 在英语里常常读作“idge”，例如，单词 carriage、marriage、disparage 和 encourage 等。

例如，假设某人希望用 100 000 美元买一幢房子。（现在许多房子的价格都远高于此，特别是在大城市，不过使用整数进行计算可以简单点。）买房子的人（假设）要付 10% 的保证金（deposit），也就是 10 000 美元，再从金融机构（如银行或者储蓄借贷公司）以抵押贷款的形式借到剩下的 90 000 美元。因此，借贷的本金（principal 或者资本，capital）就是 90 000 美元。

再假设抵押贷款的期限是 30 年，每月分期偿还，每年利率是 7.5%（或者说每月利率 0.625%）。每个月，贷款人要付给金融公司 629.30 美元。这个数额的一部分是未偿还余额的利息，其余的用来偿还本金。因此，月付款额常称为 **P&I**（本金（principal）和利息（interest））。例如，第 1 个月的时候，未偿还余额是 90 000 美元。月息是 90 000 美元的 0.625%，即 562.50 美元。支付了 629.30 美元的 P&I 的剩余部分（即 66.80 美元）用来偿还本金。因此，在第 1 个月结束时，第一次支付后，只欠金融公司 89 933.20 美元了。

第 2 个月的利息是 89 933.20 美元的 0.625%，即 562.08 美元。P&I 的支付额是 629.30 美元，与以前一样，P&I 的余额（这次是 67.22 美元）再次被用于偿还本金，这次之后就只欠 89 865.98美元了。

15 年（180 个月）之后，每月的 P&I 支付额仍是 629.30 美元，不过现在的本金已被减少到了 67 881.61 美元。67 881.61 美元的月息是 424.26 美元，所以 P&I 中余下的 205.04 美元被用来偿还本金。在 30 年（360 个月）之后，整个贷款将会还清。

金融公司希望能确保 90 000 美元的欠款包括利息能如数还清。以下一些不同的方法可以确保这一点。

- 第一，贷款人签署一份法律文件（抵押贷款契约）声明，如果每月的分期付款无法支付，金融公司可以出售该房子，并用得到的收益支付贷款的未偿还余额。
- 第二，金融公司要求借款人投保房子，为的是如果（假设）房子被烧毁了，保险公司可以弥补损失，来自保险公司的支票会被用来偿还贷款。保险费一般是由金融公司每年支付。为了从贷款人那里得到保险金的钱，金融公司要求贷款人每月分期支付保险金。每笔分期付款金额将会存储在一个第三方托管账户（escrow account）之内，但必须是由该金融公司管理的储蓄账户。当应付每年的保险费时，钱从第三方账户内取出。房子的房

产税也是同样处理的，也就是说，每月的分期付款会保存在第三方账户内，而每年的房产税由这个账户支付。

- 第三，金融公司希望确保贷款人承担得起抵押贷款。典型的情况是，如果月付款总数（P&I 加保险费及房产税）超过贷款人总收入的 28%，就不会批准抵押贷款。

除了月付款项，作为借钱给贷款人的回报，金融公司几乎总是希望贷款人前期一次付清一笔钱（首付）。典型情况下，金融公司会要本金的 2%（返点 2）。对于 90 000 美元的贷款，这个量是 1 800 美元。

最后，买房子时还会有其他费用，比如诉讼费和各种的税。因此，当签署 100 000 美元的买房合同时（就是当交易“完结”的时候），手续费（包括诉讼费、税等）加上返点 2 很容易就有 7 000 美元之多。

MSG 基金会应用域的初始术语表如图 10-3 所示。

余额（balance）：仍然欠着的贷款额
资本（capital）：本金的同义词
手续费（closing cost）：买房相关的其他成本，如诉讼费和各种各样的税
保证金（deposit）：整个买房费用的首付款项
第三方托管账户（escrow account）：一个由金融公司管理的储蓄账户，每年的保险费和房产税的每周分期付款会打入其中，每年的保险费和房产税由这个账户来支付
利息（interest）：借钱的成本，按照所欠贷款额的百分比进行计算
抵押贷款（mortgage）：以房产作保的贷款
P&I：“本金和利息”的缩写
返点（points）：借钱费用，按照贷款总额的百分比进行计算
本金（principal）：一次性的贷款额
本金和利息（principal and interest）：一次分期付款额，由利息加上当次付款时本金的一个分数组成

图 10-3　MSG 基金会实例研究的初始术语表

现在来构建 MSG 基金会实例研究的初始业务模型。

10.7　初始业务模型：MSG 基金会实例研究

开发机构的成员对 MSG 基金会的多个经理和员工进行了访谈，并了解了基金会的运作方式。在每周开始时，MSG 基金会估算本周将有多少资金可以用来资助抵押贷款。因收入过低而无法负担标准抵押贷款来买房的夫妇可以随时向 MSG 基金会申请抵押贷款。MSG 基金会员工首先确定这对夫妇是否有资格享受 MSG 抵押贷款，然后再确定 MSG 基金会本周手头上是否有足够的资金来买房。如果可以的话，就批准抵押贷款，并且根据 MSG 基金会规则，计算每周的分期偿还金额。还款金额可能每周各异，这取决于这对夫妇当前的收入。

业务模型的对应部分由 3 个用例来组成：Estimate Funds Available for Week、Apply for an MSG Mortgage 和 Compute Weekly Repayment Amount。这些用例如图 10-4、图 10-5 和图 10-6 所示，对应的初始用例描述（use-case description）分别如图 10-7、图 10-8 和图 10-9 所示。

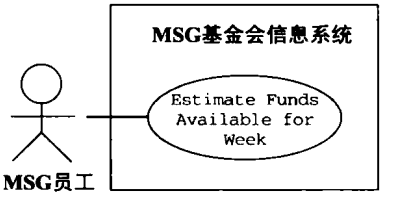


图 10-4　MSG 基金会实例研究的初始业务模型的 Estimate Funds Available for Week 用例

考虑用例 Apply for an MSG Mortgage（图 10-5），右边的参与者是申请人（applicant）。但申请人真是一个参与者吗？回想 10.4.3 节中的描述，参与者是软件产品的使用者。然而，申请人并不使用软件产品，他们只是填写表格。他们的应答由 MSG 员工输入软件产品中。此外，他们可能会问 MSG 员工问题或者回答员工提出的

问题。但是除了他们与 MSG 员工的交互，申请人并不与软件产品发生交互[⊖]。



图 10-5　MSG 基金会实例研究的初始业务模型的 Apply for an MSG Mortgage 用例



图 10-6　MSG 基金会实例研究的初始业务模型的 Compute Weekly Repayment Amount 用例

简述
Estimate Funds Available for Week 用例能够使 MSG 基金会员工估算出基金会本周有多少资金可以用来资助抵押贷款
步骤描述
在初始阶段尚不可用

图 10-7　MSG 基金会实例研究的初始业务模型的 Estimate Funds Available for Week 用例描述

简述
当一对夫妇申请抵押贷款的时候，Apply for an MSG Mortgage 用例用来使 MSG 基金员工能够确定他们是否有资格享受 MSG 抵押贷款，如果可以，那么当前是否还有可用资金用来资助该抵押贷款
步骤描述
在初始阶段尚不可用

图 10-8　MSG 基金会实例研究的初始业务模型的 Apply for an MSG Mortgage 用例描述

简述
Compute Weekly Repayment Amount 用例用来使 MSG 基金会员工能够计算出贷款人每周需偿还的金额
步骤描述
在初始阶段尚不可用

图 10-9　MSG 基金会实例研究的初始业务模型的 Compute Weekly Repayment Amount 用例描述

然而：

- 第一，申请人引发了用例。也就是说，如果一对夫妇不提出抵押贷款的申请，这个用例将不会发生。
- 第二，MSG 员工（MSG Staff Member）输入软件产品的信息由申请人提供。
- 第三，从某种含义上，真正的参与者是申请人；MSG 员工只是申请人的代理。

基于上述理由，申请人的确是一个参与者。

现在考虑图 10-6，它描述了 Compute Weekly Repayment Amount 用例。右边的参与者现在是贷款者。一旦批准了申请，申请抵押贷款的这对夫妇（申请人）就变成了贷款者。但即便是贷款者，他们也不与软件产品发生交互。如前所述，只有 MSG 员工才能向软件产品输入信息。不过，这次又是贷款者这个参与者初始化了用例并且 MSG 员工输入的信息也是由贷款者提供的。因此，贷款者也的确是图 10-6 所示用例的一个参与者。

MSG 基金会业务模型还涉及 MSG 基金会的投资。在这个初始阶段，关于投资的买和卖或者投资收入如何用来做抵押贷款这些细节问题都还不得而知，不过相当清楚的一点是图 10-10 所示的 Manage an Investment 用例是初始业务模型的基本部分。在图 10-11 中给出了初始描

⊖ 如果 MSG 基金会决定接受网上申请的话，问题就会发生变化。特别地，申请人将成为图 10-6 的唯一参与者；而 MSG 员工将不再扮演任何参与者。

述。在以后的迭代中，将加入投资如何处理的细节。

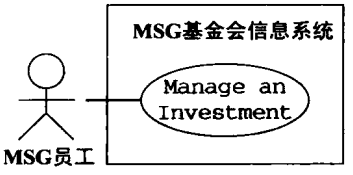


图 10-10　MSG 基金会实例研究的初始业务模型的 Manage an Investment 用例

简述
Manage an Investment 用例用来使 MSG 基金会员工能够做买和卖的投资并管理投资业务总量
步骤描述
在初始阶段尚不可用

图 10-11　MSG 基金会实例研究的初始业务模型的 Manage an Investment 用例描述

为了简化起见，将图 10-4、图 10-5、图 10-6 和图 10-10 的 4 个用例合并成为图 10-12 的用例图（use－case diagram）。

接下来拟订初始需求。

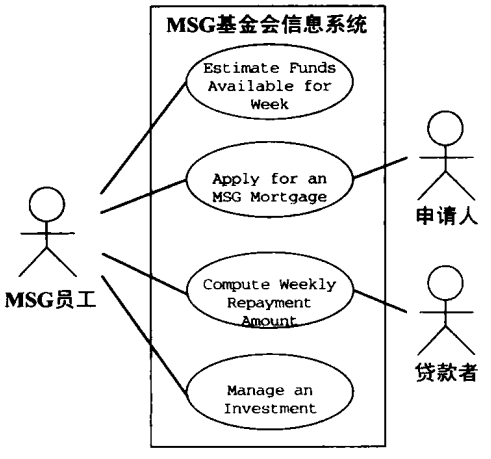


图 10-12　MSG 基金会实例研究的初始业务模型的用例图

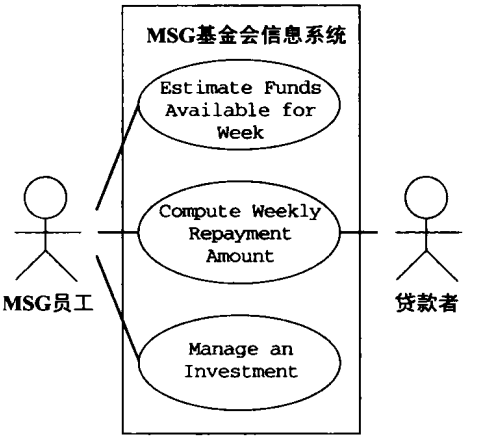


图 10-13　MSG 基金会实例研究的初始需求的用例图

10.8　初始需求：MSG 基金会实例研究

图 10-12 的 4 个用例构成了 MSG 基金会的业务模型。然而，它们是否就是所需开发的 MSG 基金会软件产品的全部需求，仍然不是很清楚。回忆一下，客户所想要的是“一个试验性项目，即一个能够进行必要的计算以确定每周有多少资金可以用于购买房子的软件产品。”与往常一样，开发者的任务是在客户的协助下确定什么是客户所需的。但是，在早期阶段，没有可供分析员支配的足够的信息来决定是否所需的就恰是这个“试验性项目”。在这种情况下，继续下去的最好方法就是基于客户之所想拟订初始需求，然后进行迭代。

相应地，依次考虑图 10-12 中的每个用例。Estimate Funds Available for Week 用例显然是初始需求的一部分。另一方面，Apply for an MSG Mortgage 用例似乎与这个试验性项目没什么关系，所以把它排除在初始需求外。第 3 个用例（Compute Weekly Repayment Amount）看起来似乎也与这个试验性项目不相关。然而，这个试验性项目要处理“每周可用于买房的钱”。而这些钱的一部分必然是来自于现存的抵押贷款的周偿还金，所以第 3 个用例事实上的确是初始需求的一部分。同理，第 4 个用例（Manage an Investment）也是初始需求的一部分——投资的收入也必须用于新的抵押贷款上。

初始需求于是由 3 个用例及其描述组成，即 Estimate Funds Available for Week（图 10-4

和图 10-7）、Compute Weekly Repayment Amount（图 10-6 和图 10-9）和 Manage an Investment（图 10-10 和图10-11）。这 3 个用例如图 10-13 所示。

下一步进行需求工作流的迭代，也就是说，重复需求过程的步骤以得到一个更好的客户需求模型。

10.9　需求工作流继续：MSG 基金会实例研究

有了应用域知识并对初始业务模型熟悉后，开发团队的成员现在对 MSG 基金会的管理者和员工们进行了更深入的访谈工作。他们发现了如下信息。

只有在下列情况下，MSG 基金会才会批准一个 100% 的购房抵押贷款：

- 一对夫妇结婚至少一年，但不满 10 年。
- 夫妇双方都有正式的工作。尤其，必须出示双方在过去的一年中至少全职工作了 48 周的证明。
- 房子的价格必须低于过去 12 个月公开发布的该地区房屋的价格。
- 按照固定汇率、30 年、90% 抵押贷款的分期偿还金超过了该夫妇俩联合收入的 28%，并且/或者这对夫妇没有足够的积蓄支付房费的 10% 再加 7 000 美元（这个 7 000 美元是相关的额外成本，包括手续费和返点）。
- 基金会有足够的经费用来购房，这在之后会详细描述。

如果申请获得了许可，那么这对夫妇在未来的 30 年中每周需要支付给 MSG 基金会的数额是 P&I 款项（在抵押贷款的生命期中其数额保持不变），加上第三方款项（是年房产税与年户主保险金之和的 1/52）。如果这个总数超过了这对夫妇每周联合收入的 28%，那么 MSG 基金会就会以补助的形式支付这个差额。这样，抵押贷款每周都会全额付清，但这对夫妇不需要支付超过他们联合收入的 28% 的部分。

每对夫妇必须提供他们每年收入税单回执的复印件，以便 MSG 基金会有关于他们过去一年收入的证明。此外，每对夫妇可以提交他们工资单的副本作为他们当前总收入的证明。因此，一对夫妇用来支付抵押贷款的金额可能每周都会不同。

MSG 基金会使用如下的算法来确定是否还有经费来许可一个抵押贷款的申请：

1）在每周的开始，计算基金会投资年收入额的估计值，再除以 52。

2）MSG 基金会年运作费用的估计值除以 52。

3）计算本周抵押贷款偿还金的估计值的总数。

4）计算本周拨款的估计值的总数。

5）在本周开始时，可用的金额是（第 1 项）－（第 2 项）＋（第 3 项）－（第 4 项）。

6）在本周内，如果一个房屋的价格低于用于抵押贷款的可用金额，那么 MSG 基金会就认为有购买该房屋所需的经费。然后，在抵押贷款的可用金额里减去该房屋的价格。

7）在每周末，MSG 基金会投资顾问把所有没有用掉的经费用于投资。

为了保证试验性项目的成本尽可能的低，软件产品应该只包含那些计算每周经费才需要的数据项。如果 MSG 基金会决定将其运作的所有方面计算机化，那么以后可以再添加剩下的数据项。因此，只需要 3 种类型的数据：投资数据、运作费用数据和抵押贷款数据。

关于投资，需要下面的数据：

- 项目号。
- 项目名。
- 估计的年利润。（每当有新的可用信息的时候，就会更新这个数值。平均来说，每年更新 4 次。）
- 年利润估计值最后一次更新的日期。

关于运作费用，需要下面的数据：

- 估计的年运作费用。（这个数值每年确定 4 次。）

- 年运作费用估计值最后一次更新的日期。

对每个抵押贷款，需要下面的数据：

- 账户号。
- 抵押贷款者的姓。
- 房屋的原始购买价格。
- 发布抵押贷款的日期。
- 周本金和利息偿还金（即周 P&I 金额）。
- 当前夫妇的周联合总收入。
- 周联合总收入最后一次更新的日期。
- 年房产税。
- 年房产税最后一次更新的日期。
- 年户主保险金。
- 年户主保险金最后一次更新的日期。

在与 MSG 管理人员进一步的讨论中，开发者了解到需要三种报表：

- 本周经费计算结果。
- 全部投资列表（需要时可打印）。
- 全部抵押贷款列表（需要时可打印）。

10.10　修订需求：MSG 基金会实例研究

回想一下，初始需求模型（10.8 节）包括三个用例，即 Estimate Funds Available for Week、Compute Weekly Repayment Amount 和 Manage an Investment。这些用例如图 10-13 所示。鉴于已获得的额外信息，可对初始需求进行修订。

10.9 节给出的用来确定每周开始时可用金额数量的公式如下：

1）计算来自投资的年收入估计值，再除以 52。

2）MSG 基金会年运作费用的估计值除以 52。

3）计算本周总抵押贷款偿还金的估计值。

4）计算本周总拨款的估计值。

5）可用金额数量为（第 1 项）－（第 2 项）＋（第 3 项）－（第 4 项）。

依次考虑下面这些条款：

1）投资的年收入估计值。对于每项投资，依次累加估计每项带来的年度回报，求取的结果除以 52。为了做到这一点，需要额外的一个用例，即 Estimate Investment Income for Week。（仍然需要用例 Manage an Investment 来增加、删除和修改投资项。）这个新用例如图 10-14 所示，并在图 10-15 中给出其用例描述。在图 10-14 中，标注《include》带箭头的虚线表示用例 Estimate Investment Income for Week 是用例 Estimate Funds Available for Week 的一部分。图 10-16 给出了经过第一次迭代修订后的用例图，其中新的用例以阴影标记。

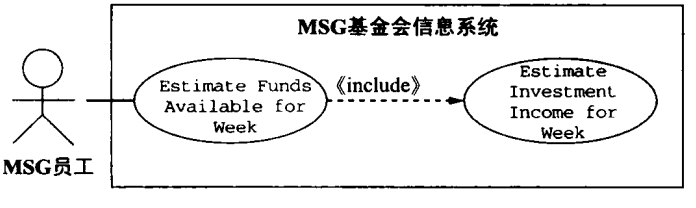


图 10-14　MSG 基金会实例研究的修订需求的 Estimate Investment Income for Week 用例

简述
Estimate Investment Income for Week 用例使 Estimate Funds Available for Week 用例能够估计出本周有多少可用投资收入
步骤描述 1. 对每项投资，得出该项投资带来的年利润的估计值 2. 对步骤 1 中得出的值求和并把结果除以 52

图 10-15　MSG 基金会实例研究的修订需求的 Estimate Investment Income for Week 用例描述

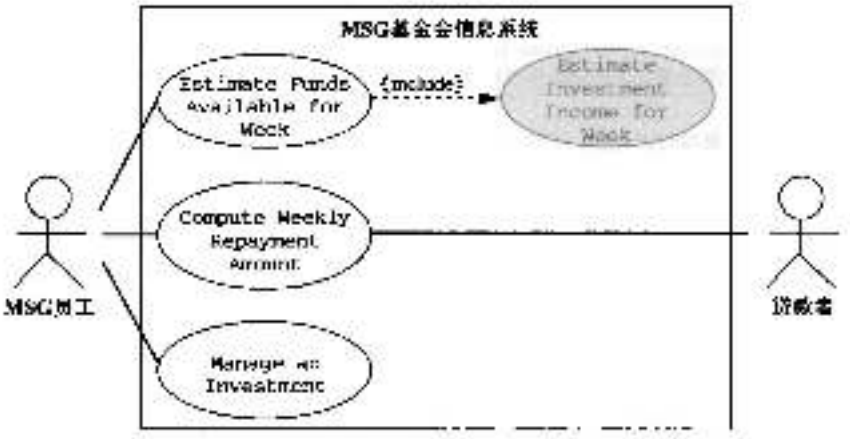


图 10-16　MSG 基金会实例研究的修订需求的用例图的第一次迭代。新增用例标记为阴影

2）年运作费用的估计值。到目前为止，还没有考虑年运作费用的估计值。为包括这些费用，需要两个额外的用例。用例 Update Estimated Annual Operating Expenses 建模了年运作费用的估计值的调整，而用例 Estimate Operating Expenses for Week 提供了所需运作费用的估计值。这些用例如图 10-17 到图 10-20 所示。在图 10-19 中，类似地，用例 Estimate Operating Expenses for Week 也是用例 Estimate Funds Available for Week 的一部分，正如图中标注《include》带箭头的虚线所示。修订用例图的第二次迭代结果如图 10-21 所示，其中两个新的用例（Update Estimated Annual Operating Expenses 和 Estimate Operating Expenses for Week）以阴影标记。



图 10-17　MSG 基金会实例研究的修订需求的 Update Estimated Annual Operating Expenses 用例

3）本周总投种贷款偿还金的估计值。参见第 4 项。

简述
Update Estimated Annual Operating Expenses 用例使一个 MSG 基金会员工能够更新年运作费用的估计值
步骤描述 1. 更新年运作费用的估计值

图 10-18　MSG 基金会实例研究的修订需求的 Update Estimated Annual Operating Expenses 用例描述



图 10-19　MSG 基金会实例研究的修订需求的 Estimate Operating Expenses for Week 用例

简述
Estimate Operating Expenses for Week 主例处 Estimate Funds Available for Week 用例能够估算出本周的运作费用
步骤描述
1. 把年运作费用的估计值除以 52

图 10-20　MSG 基金会实例研究的修订需求的 Estimate Operating Expenses for Week 用例描述

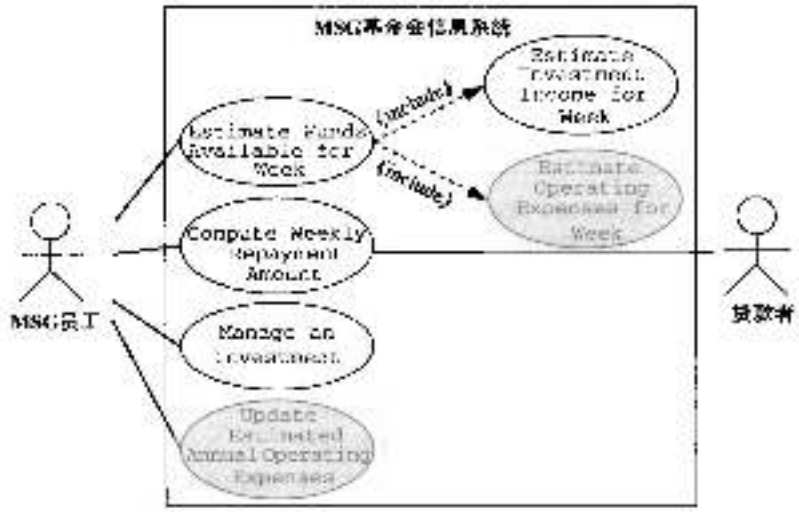


图 10-21　MSG 基金会实例研究的修订需求的用例图的第二次迭代。两个新的用例 Estimate Operating Expenses for Week 和 Update Estimated Annual Operating Expenses 标记为阴影

4）本周总拨款的估计值。由用例 Compute Weekly Repayment Amount 得出的周偿还金额是由总抵押贷款偿还金的估计值减去总拨款的估计值后得到的。换句话说，用例 Compute Weekly Repayment Amount 建模了为每个抵押贷款独立计算其抵押贷款偿还金的估计值和本周拨款的估计值的过程。将这些单独的数值求和，就得到了本周的总抵押贷款偿还金的估计值以及本周总拨款的估计值。然而，Compute Weekly Repayment Amount 也建模了贷款者修改他们周收入数额。相应地，Compute Weekly Repayment Amount 需要分成两个独立的用例，即 Estimate Payments and Grants for Week 和 Update Borrowers' Weekly Income。这两个新的用例如图 10-22 至图 10-25 所示。以一个新的用例（Estimate Payments and Grants for Week）是用例 Estimate Funds Available for Week 的一部分，正如图 10-22 中标注《include》带箭头的虚线所示。修订用例图的第二次迭代结果如图 10-26 所示，其中两个新的用例是从用例 Compute Weekly Repayment Amount 中派生出来的，以阴影标记。

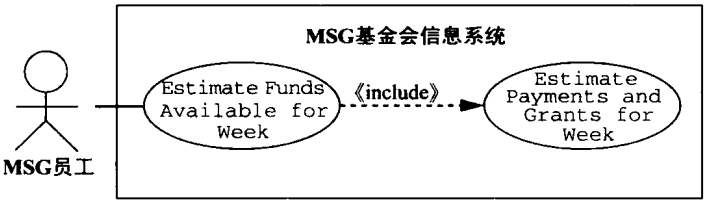


图 10-22　MSG 基金会实例研究的修订需求的 Estimate Payments and Grants for Week 用例

简述

Estimate Payments and Grants for week 用例使 Estimate Funds Available for week 用例能够估算出本周由贷款者付给 MSG 基金会的总抵押贷款偿还金的估计值和本周由 MSG 基金会支付的总拨款额的估计值

步骤描述

1. 对每个抵押贷款：
   - 1.1　本周需支付的数额是本金和利息偿还金（P&I）与年房产税和年户主保险金之和的 1/52 的总和
   - 1.2　计算出该对夫妇当前周总收入的 28%
   - 1.3　如果步骤 1.1 的值比步骤 1.2 的值大，那么本周的抵押贷款偿还金额就是步骤 1.2 的结果，并且本周的拨款额就是步骤 1.1 的结果与步骤 1.2 的结果之差
   - 1.4　否则，本周的抵押贷款偿还金就是步骤 1.1 的结果，并且本周没有拨款
2. 对步骤 1.3 和步骤 1.4 的抵押贷款偿还金求和就产生了本周的抵押贷款偿还金的估计值
3. 对步骤 1.3 的拨款额求和就产生了本周的拨款额的估计值

图 10-23　MSG 基金会实例研究的修订需求的 Estimate Payments and Grants for Week 用例描述

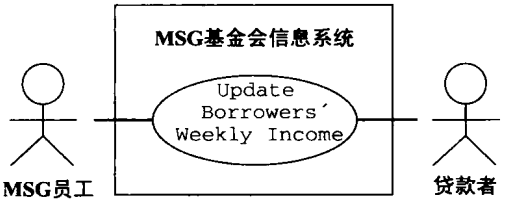


图 10-24　MSG 基金会实例研究的修订需求的 Update Borrowers' Weekly Income 用例

简述

Update Borrowers'Weekly Income 用例使一位 MSG 基金会员工能够更新一对从基金会贷款的夫妇的周收入额

步骤描述

1. 更新贷款者的周收入

图 10-25　MSG 基金会实例研究的修订需求的 Update Borrowers' Weekly Income 用例描述

再考虑图 10-26。用例 Estimate Funds Available for Week 建模了使用从另外 3 个用例中得出的数据进行计算的过程，这 3 个用例是 Estimate Investment Income for Week、Estimate Operating Expenses for Week 和 Estimate Payments and Grants for Week。这如图 10-27 所示，它是用例 Estimate Funds Available for Week 的第二次迭代版本。图 10-27 是从图 10-26 中的用例图中抽取出的。图 10-28 是该用例的对应描述。

为什么在 UML 图中指明**《include》关系**如此重要？例如，图 10-29 显示了图 10-26 的两个版本，图 10-29a 是正确的版本而图 10-29b 是错误的。图 10-29a 正确地将用例 Estimate Funds Available for Week 建模为用例 Estimate Payments and Grants for Week 的一部分。图 10-29b 将用例 Estimate Funds Available for Week 和 Estimate Payments and Grants for Week 建模为两个独立的用例。然而，正如 10.4.3 节所述，一个用例建模了软件产品本身和软件产品的使用者（参与者）之间的一种交互。这对用例 Estimate Funds

Available for Week 而言是对的。但是，用例 Estimate Payments and Grants for Week 并没有与一个参与者交互，因此，它未获独立形成一个用例。相反，它只是用例 Estimate Funds Available for Week 的一部分，正如图 10-29a 所示。

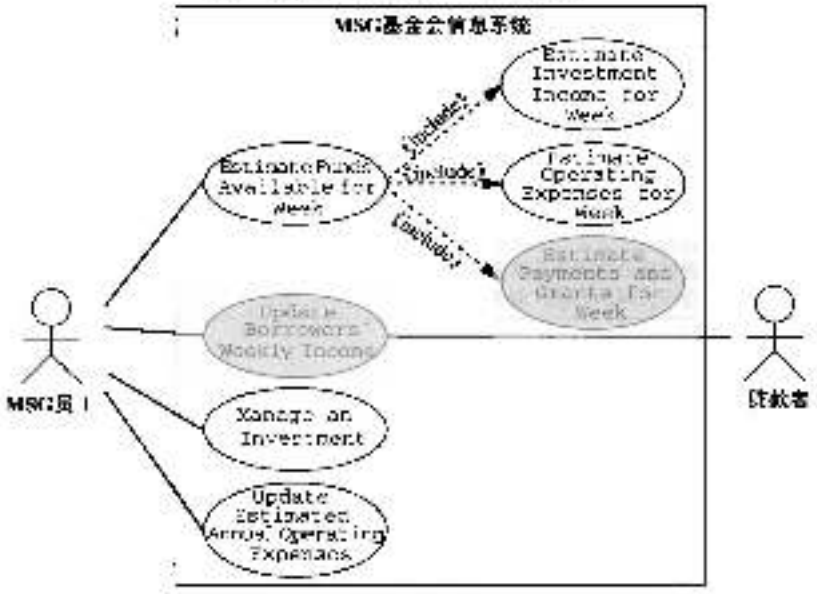


图 10-26　MSG 基金会实例研究的修订需求的用例图的第二次迭代。由用例 Estimate Weekly Repayment Amount 衍生出的两个新的用例以阴影标记

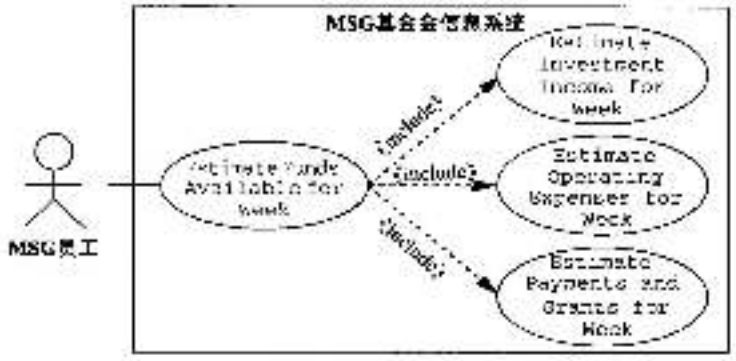


图 10-27　MSG 基金会实例研究的修订需求的 Estimate Funds Available for Week 用例的第二次迭代

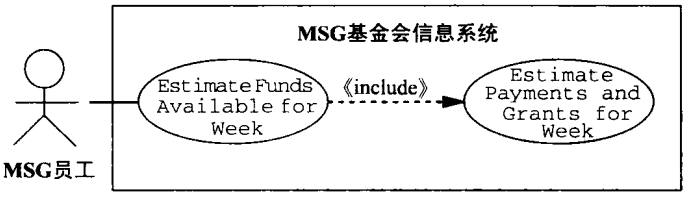
描述

Estimate Funds Available for Week 用例使 MSG 基金会的工作人员能够估算出在周基金会有多少资金可以用来资助抵押贷款

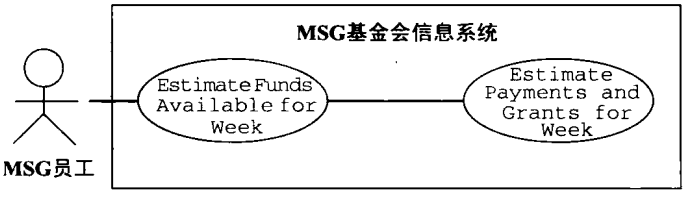
步骤描述

1. 使用用例 Estimate Investment Income for Week 确定本周来自投资的收入估计值
2. 使用用例 Estimate Operating Expenses for Week 确定本周运作费用的估计值
3. 使用用例 Estimate Payments and Grants for Week 确定本周总抵押贷款资金的估计值
4. 使用用例 Estimate Payments and Grants for Week 确定本周总拨款额的估计值
5. 把步骤 1 和步骤 2 的结果相加并减去步骤 3 和步骤 4 的结果，这就是当前这周可用于抵押贷款的总金额

图 10-28　MSG 基金会实例研究的修订需求的 Estimate Funds Available for Week 用例描述的第二次迭代



a）正确版本



b）错误版本

图 10-29　图 10-22 的正确版本和错误版本

10.11　测试工作流：MSG 基金会实例研究

迭代和增量的生命周期模型的一个常见的副作用是本来被延期的正确的细节被遗忘了。这就是持续测试至关重要的众多原因之一。在本例中，用例 Manage an Investment 的细节被忽视了。图 10-30 和图 10-31 修正了这一点。

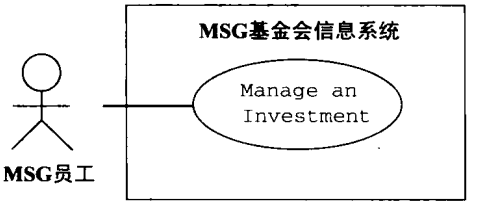


图 10-30　MSG 基金会实例研究修订了需求的 Manage an Investment 用例

进一步的访谈发现用例 Manage a Mortgage 被忽视了，该用例建模了增加一个新抵押贷款、修改一个现存的抵押贷款或删除一个现存的抵押贷款的情况，这跟用例 Manage an Investment 相类似。图 10-32 和图 10-33 纠正了这一疏忽，修订的用例图的第 4 次迭代结果如图10-34所示，其中新的用例（Manage a Mortgage）以阴影标记。

简述
Manage an Investment 用例使一位 MSG 基金会员工能够增加和删除投资项并管理投资总额
步骤描述
1. 增加、修改或删除一项投资

图 10-31　MSG 基金会实例研究修订了需求的 Manage an Investment 用例描述

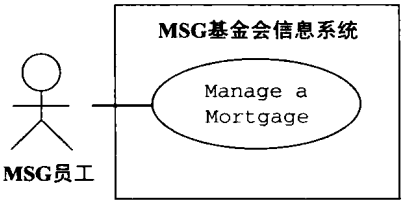


图 10-32　MSG 基金会实例研究修订了需求的 Manage a Mortgage 用例

简述
Manage a Mortgage 用例使一位 MSG 基金会员工能够增加和删除抵押贷款项并管理抵押贷款总额
步骤描述
1. 增加、修改或删除一项抵押贷款

图 10-33　MSG 基金会实例研究修订了需求的 Manage a Mortgage 用例描述

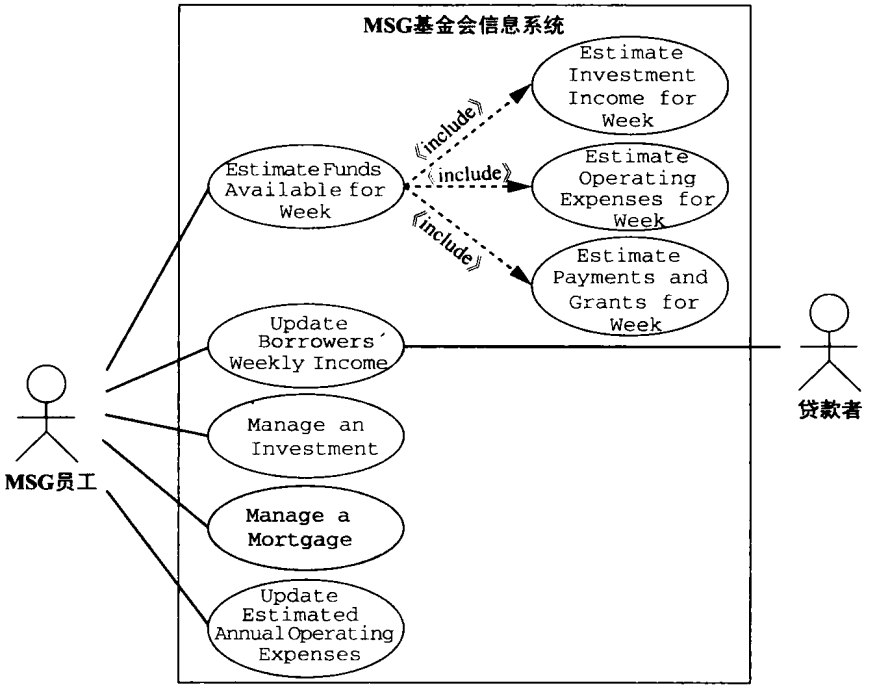


图 10-34　MSG 基金会实例研究修订了需求的用例图的第 4 次迭代。新的用例 Manage a Mortgage 以阴影标记

进一步，打印各种报表的用例也被忽视了。相应地，新增用例 Produce a Report，它建模了打印三种报表的情况。该用例的细节部分如图 10-35 和图 10-36 所示。修订用例图的第 5 次迭代结果如图 10-37 所示，其中新的用例 Produce a Report 以阴影标记。

再次检查修订的需求，又发现了两个新问题。首先，一个用例有部分冗余。其次，有 2 个用例需要重新组织。

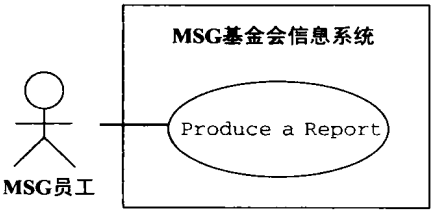


图 10-35　MSG 基金会实例研究修订了需求的 Produce a Report 用例

第一个要做的改动是除去部分冗余的用例。考虑用例 Manage a Mortgage（图 10-32 和图 10-33）。如图 10-33 所述，该用例的一个行为是修改一项抵押贷款。再看用例 Update Borrowers' Weekly Income（图 10-24 和图 10-25）。该用例（图 10-25）的唯一目的是更新贷款者的周收入。但是贷款者的周收入是抵押贷款的一个属性。也就是说，用例 Manage a Mortgage 已经包括了用例 Update Borrowers' Weekly Income，相应地，用例 Update Borrowers' Weekly Income 是多余的，应该被删除。结果如图 10-38 所示，它是修订的用例图的第 6 次迭代，其中被修改的用例 Manage a Mortgage 以阴影标记。

简述

Produce a Report 用例使 MSG 基金会员工能够打印出每周可用于新抵押贷款的经费的计算结果或者打印出一份全部投资项或全部抵押贷款项的表单。

步骤描述

1. 必须生成如下的报表：
   1.1　投资报表——需要时可打印：
   信息系统打印一份全部投资项的表单。对每一项投资，下面的属性要打印出来：
   项目号
   项目名
   估计的年利润
   年利润估计值最后一次更新的日期
   1.2　抵押贷款报表——需要时可打印：
   信息系统打印一份全部抵押贷款项的表单。对每一项抵押贷款，下面的属性要打印出来：
   账户号
   抵押贷款者的姓名
   房屋的原始购买价格
   发布抵押贷款的日期
   本金和利润（P&I）偿还金
   当前的周联合总收入
   周合并总收入最后一次更新的日期
   年房产税
   年房产税最后一次更新的日期
   年户主保险金
   年户主保险金最后一次更新的日期
   1.3　每周计算结果——每周都要打印：
   信息系统打印出本周内可用于新的抵押贷款的总金额

图 10-36　MSG 基金会实例研究修订了需求的 Produce a Report 用例描述

这是第一次导致减量而非增量的迭代过程。也就是说，这是本书第一次发生迭代删除了一个制品（Update Borrowers' Weekly Income 用例）的情况。事实上，当错误发生时，经常会出现删除。关键是，当发现一个错误的时候，没有必要放弃掉到目前为止所做的一切，而从头开始再重做整个需求过程。应该是，尝试修正当前进行的迭代，如这个实例研究中所做的那样。如果这个策略失败（原因是错误确实相当严重），就回溯到上一次迭代，并试着从那里再找一条更好的路线进行。

第二个为改进需求而必须做的改动是重新组织两个用例。考虑 Estimate Funds Available for Week（图 10-28）和 Produce a Report（图 10-36）的用例描述。假设有一个 MSG 员工想确定本周可用的经费。用例 Estimate Funds Available for Week 执行计算，再由用例 Produce a Report 的步骤 1.3 打印出计算的结果。这有点可笑，但毕竟，除非结果需要打印，否则无法估计可用经费。

换言之，Produce a Report 的步骤 1.3 需要从该用例的用例描述中移动到用例 Estimate Funds Available for Week 的用例描述的末尾。这并不改变两个用例本身（图 10-27 和图 10-35）或者当前的用例图（图 10-38），但是两个用例的描述（图 10-28 和图 10-36）需要修改。用例描述的修改结果如图 10-39 和图 10-40 所示。

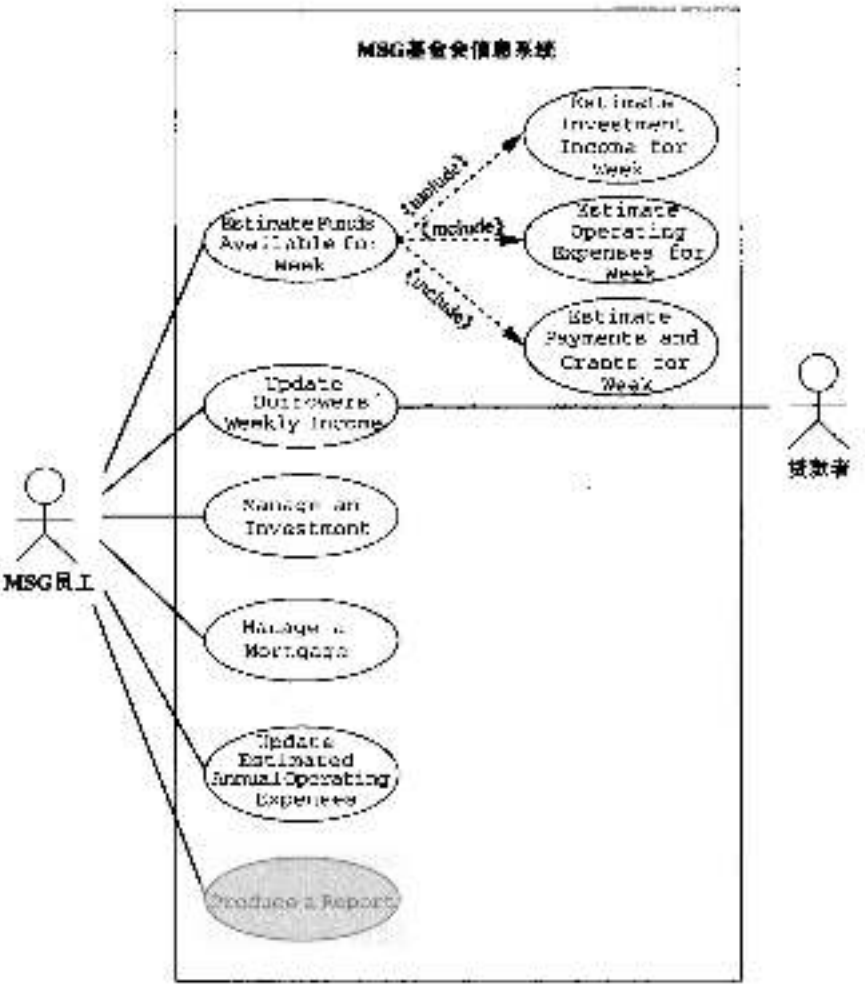


图 10-37　MSG 基金会实例研究修订了需求的用例图的第 5 次迭代，新的用例 Produce a Report 以阴影标记

现在，可以再进一步改进用例图。考虑图 10-38 最上面的 4 个用例。在右边的三个用例，即 Estimate Investment Income for Week、Estimate Operating Expenses for Week 和 Estimate Payments and Grants for Week 是用例 Estimate Funds Available for Week 的组成部分。通常在一个用例是两个或更多的其他用例的一部分的情况下，使用《include》关系。例如，如图 10-41 所示，用例 Print Tax Form 是用例 Prepare Form 1040，Prepare Form 1040A 和 Prepare Form 1040EZ（这是美国最主要的三种个人税单）的一部分。在这种情况下，保留 Print Tax Form 为一个单独的用例是有意义的。将 Print Tax Form 的操作包含到其他三个用例中去意味着该用例重复了三次。

然而，就图 10-38 而言，所有包括的用例都只是一个用例（即 Estimate Funds Available for Week）的一部分。其中并无冗余。相应地，将这三个《include》用例并入 Estimate Funds Available for Week 是有意义的，正如图 10-42 所示的用例图的第 7 次迭代。Estimate Funds Available for Week 用例描述的第 4 次迭代结果如图 10-43 所示。

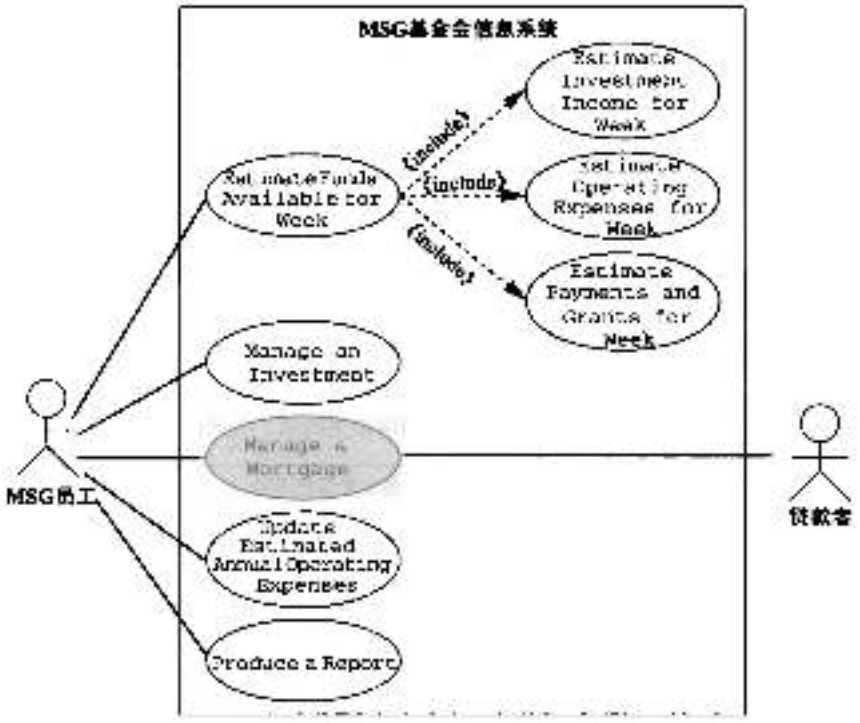


图 10-38　MSG 基金会实例研究修订了需求的用例图的第 6 次迭代：
经修改的用例 Manage a Mortgage 以阴影标记

简述
Produce a Report 用例使 MSG 基金会员 L 能够打印出一份全部投资项或全部抵押贷款项的表单
步骤描述 1. 必须生成如下的报表： 1.1　投资报表——需要时可打印： 信息系统打印一份全部投资项的表单，对每一项投资，下面的属性要打印出来： 项目号 项目名 估计的年利润 年利润估计值最后一次更新的日期 1.2　抵押贷款报表——需要时可打印： 信息系统打印一份全部抵押贷款项的表单，对每一项抵押贷款，下面的属性要打印出来： 账户号 抵押贷款者的姓名 房屋的原始购买价格 发布抵押贷款的日期 本金和利润（P&I）偿还金 当前的周联合总收入 周联合总收入最后一次更新的日期 年房产税 年房产税的最后一次更新的日期 年广主保险金 年广主保险金最后一次更新的日期

图 10-39　MSG 基金会实例研究修订了需求的 Produce a Report 用例描述的第 2 次迭代

简述
Estimate Funds Available for Week 用例使 MSG 基金会员工能够估算出本周基金会有多资金可以用来资助抵押贷款
步骤描述 1. 使用用例 Estimate Investment Income for Week 确定本周来自投资的收入的估计值 2. 使用用例 Estimate Operating Expenses for Week 确定本周运作费用的估计值 3. 使用用例 Extimate Payments and Grants for Week 确定本周总抵押贷款偿还金的估计值 4. 使用用例 Extimate Payments and Grants for Week 确定本周总拨款额的估计值 5. 把步骤 1 和步骤 3 的结果相加并减去步骤 2 和步骤 4 的结果。这就是当前这周可用于抵押贷款的总金额 6. 打印出本周内可用于新的抵押贷款的总金额

图 10-40　MSG 基金会实例研究修订了需求的 Estimate Funds Available for Week 用例描述的第 3 次迭代

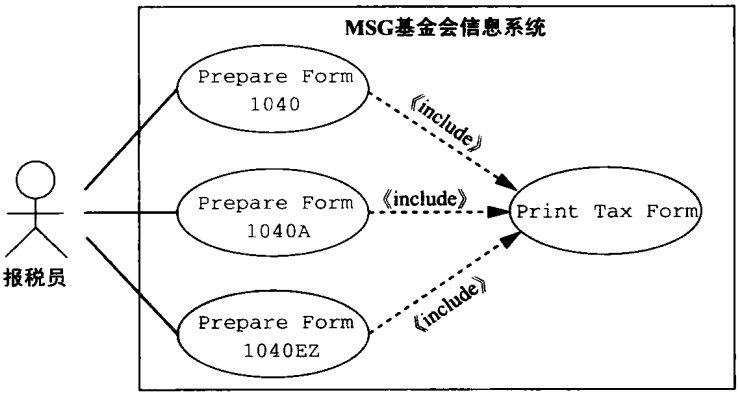


图 10-41　用例 Print Tax Form 是其他三个用例的一部分

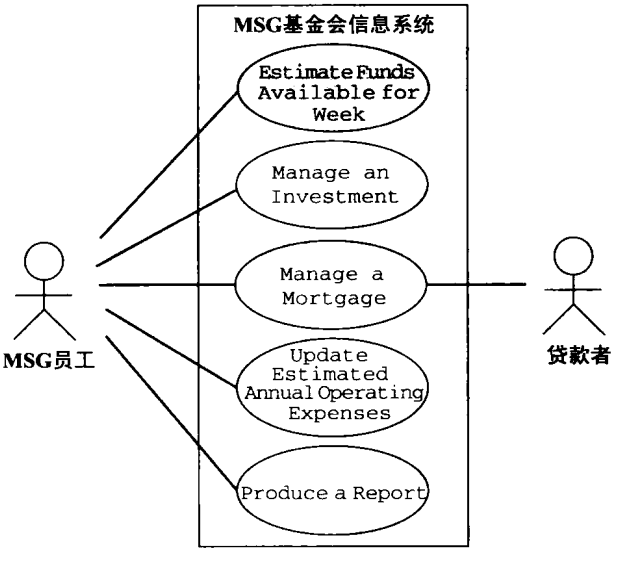


图 10-42　MSG 基金会实例研究修订了需求的用例图的第 7 次迭代。经修改的用例 Estimate Funds Available for Week 标记为阴影

简述

Estimate Funds Available for Week 用例使 MSG 基金会员工能够估算出本周基金会有多少资金可以用来资助抵押贷款

步骤描述

1. 对每项投资，得出该项投资带来的年利润的估计值。对单独的每个值求和并把结果除以 52，这样就得到了本周投资收入的估计值
2. 通过取得年运作费用的估计值并除以 52 来确定 MSG 基金会本周的运作费用的估计值
3. 对每个抵押贷款：
   - 3.1　本周需支付的数额是本金和利息偿还金（P&I）与年房产税和年户主保险金之和的 1/52 的总和
   - 3.2　计算出该对夫妇当前周总收入的 28%
   - 3.3　如果步骤 3.1 的值比步骤 3.2 的值大，那么本周的抵押贷款偿还金额就是步骤 3.2 的结果，并且本周的拨款额就是步骤 3.1 的结果与步骤 3.2 的结果之差
   - 3.4　否则，本周的抵押贷款偿还金就是步骤 3.1 的结果，并且本周没有拨款
4. 将步骤 3.3 和步骤 3.4 中抵押贷款偿还金额的结果求和就产生了本周总抵押贷款偿还金额的估计值
5. 将步骤 3.3 中拨款额的结果求和就产生了本周总拨款额的估计值
6. 把步骤 1 和步骤 4 的结果相加并减去步骤 2 和步骤 5 的结果。这就是当前这周可用于抵押贷款的总金额
7. 打印出本周内可用于新的抵押贷款的总金额

图 10-43　MSG 基金会实例研究修订了需求的 Estimate Funds Available for Week 用例描述的第 4 次迭代

现在的需求看来是正确的。

- 首先，这些需求与客户所要求的相符。
- 其次，看起来没什么错误。
- 最后，从现阶段来看，客户所想要的恰好就是客户所需要的。

因此，就目前而言，需求工作流似乎是完成了。不过，在随后的工作流中，很有可能会出现额外的需求。同样，也可能必须把 5 个用例中的一个或多个用例分裂为新的用例。例如，在以后的迭代中，图 10-36 所示的 Produce a Report 用例会分裂成两个独立的用例，一个用来做投资报表，另一个用来做抵押贷款报表。但到目前为止，一切都令人满意。

MSG 基金会实例研究需求工作流的描述到此为止。

10.12　什么是面向对象的需求

一方面，没有所谓“面向对象的需求”这样的东西存在，而且它也不应该存在。需求工作流的目标是确定客户的需求，即目标系统的功能应该是什么。需求工作流与如何制造产品无关。按照这一观点，在需求工作流的范畴内提及面向对象范型就像提出一份面向对象的用户手册一样，都是毫无意义的。毕竟，用户手册描述的是运行软件产品时用户所需遵循的步骤，而与如何制造产品毫无关系。同样的道理，需求工作流产生要做什么样的产品的说明，制造产品的方式并不包含在内。

另一方面，从 10.2 节至 10.11 节的整个方法实质上是面向对象的，也就是面向模型的。用例以及用例描述形成了需求工作流的基础。如本书的整个第二部分所述，建模是面向对象范型的精髓。

基于 11.22 节给出的理由，为产品的整体建造一个快速原型并不是面向对象范型的一部分。

然而，当使用面向对象范型时，强烈建议建造一个用户界面（UI）的快速原型，这点接下来将会介绍。

10.13　快速原型

一个快速原型（rapid prototype）是一个展现目标产品关键功能的快速建立的软件。例如，一个协助管理一栋复合公寓的产品必须包括一个输入屏幕，以允许用户输入新增房客的详细信息，并每月打印出一份房间占有情况报表。这些方面都需要包含到快速原型当中。然而，查错能力、文件更新程序和复杂税金计算可能就不包括在内了。关键是一个快速原型要反应客户看得见的功能（如输入屏幕和报表），而省略掉“隐藏”的方面，如文件更新等（关于看待快速原型的不同方式，请参见备忘录 10.3）。

备忘录 10.3

建造模型以展示产品关键特征的想法由来已久。例如，一幅 Domenico Cresti（又名“II Passignano”，因其出生于意大利 Chianti 地区的 Passignano 小镇）于 1618 年绘制的油画描绘了米开朗基罗向保罗四世展示一具由其设计的圣·彼得大教堂（在罗马）木制模型的场面。这个建筑模型十分巨大，建筑师 Bramante 前期设计所建议的圣·彼得大教堂模型，其每边长度都超过 20 英尺。

建筑模型有许多不同的用途。首先，就像 Cresti 油画（现悬挂于佛罗伦萨的 Casa Buonarroti 博物馆里）中所描绘的，模型用于激起客户投资项目的兴趣。这与在传统范型中使用快速原型确定客户的真正需求是类似的。其次，在建筑设计图纸出现之前的年代里，模型向建造者展示了建筑的结构，并且指示了石匠如何装修建筑。这与我们现在建立用户界面的快速原型是相似的，这将在 10.14 节中描述。

客户和该产品的预期用户试用快速原型的同时，开发团队的成员观察并作记录。用户基于自己亲自的实践经历，告诉开发者快速原型如何才能满足他们的需要，更重要的是，指出需要改进的地方。开发者修改快速原型，直到双方都确信客户的需求已精确地封装在了快速原型内为止。

在传统范型中，产品关键特征的快速原型用于作为规格说明书提出的基础。在面向对象范型中，快速原型是产品用户界面开发的必要组件，这将在 10.14 节进行讨论。

10.14　人为因素

让客户和产品未来的用户都与用户界面的快速原型进行交互是很重要的。鼓励用户试用人机界面（Human - Computer Interface，HCI）可以大大减少已完成产品还需改动的风险。特别是，试用还有助于获得用户友好性，这是所有软件产品的一个至关重要的目标。

用户友好性（user friendliness）是指人与软件产品沟通的容易性。如果用户在学习如何使用产品时有困难或者发现屏幕令人感到混淆或不愉快，那么他们或者不使用这个产品或者会不正确地使用产品。为了解决这个问题，引入了菜单驱动的产品。用户不用输入一条像 `Perform computation`（执行计算）或 `Print service rate report`（打印服务等级报表）这样的命令，而仅需从一系列可能的响应中选择一个，例如：

1. `Perform computation`
2. `Print service rate report`
3. `Select view to be graphed`

在这个例子中，用户输入 1、2 或 3 来调用相应的命令。

现今的 HCI 往往采用图形界面，而不是简单地显示文本行。窗口、图标和下拉式菜单是一个图形用户界面（Graphical User Interface，GUI）的组件。由于窗口系统繁多，于是就发展出了像 X Window 这样的标准。同样，点击（point - and - click）选择现在也很平常。用户移动鼠标（即一种手持的点击设备），把屏幕上的指针移动到想要的响应（“点”）上，然后按一下鼠标的按键（“击”）选择那个响应。

然而，即便目标产品采用的是现代技术，设计者也必须牢记产品是给人用的。换言之，HCI 的设计者必须考虑人为因素（human factor），例如，字母大小、大小写、颜色、行长度和屏幕上的行数等。

人为因素的另一个例子应用于前述的菜单。如果用户选择选项“3. Select view to be graphed”，会出现带有其他一些选择的列表的另一个菜单。除非菜单驱动系统的设计考虑周全，否则用户可能冒着遭遇冗长的菜单序列的危险来完成一次相对简单的操作。这样的耽搁会激怒用户，有时会造成用户做出不合适的菜单选择。同样，HCI 必须允许用户改变上一次的选择，而无需回到顶层菜单和重新开始。即使使用 GUI，这样的问题也会出现，因为许多图形用户界面本质上就是以吸引人的屏幕格式来显示一系列菜单。

有时单一的用户界面无法满足所有用户的需要。例如，如果产品既需要给计算机专业人员使用，也需要给过去没有计算机经验的高中辍学者使用，那么设计两套不同的 HCI 更为可取，每套都根据其期望用户的技能等级和心理状态进行精心裁剪。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来加以扩展，如果该软件产品推断用户使用一个不太复杂的用户界面将会感觉到更方便，那么可能是因为该用户正在不断出错或者正在连续调用帮助工具，但是，随着该用户对产品的逐渐熟悉，软件将向用户显示提供具有较少信息的简单屏幕视图，这样可使用户迅速完成任务。这种自动化方法减少了用户的困惑，提高了生产率［Schach and Wood，1986］。

在 HCI 设计过程中将人为因素纳入考虑会产生不少好处，包括减少学习时间和降低出错率。尽管帮助工具总是要提供的，但是在一个精心设计的 HCI 中它们很少被使用。这也提高了工作效率。一个产品或一组产品的 HCI 外观的一致性使用户凭直觉就可以知道如何使用一个他们从未见到过的屏幕，因为它与他们熟悉的其他屏幕相似。Macintosh 软件的设计者们已经考虑到这个原则，这也正是 Macintosh 软件普遍的用户友好性的众多原因之一。

曾有人说设计一个用户友好的 HCI 只需要简单的常识就行了。不管这种说法是否属实，每个软件产品都必须要建造其 HCI 的快速原型。该产品预期的用户可以试用 HCI 的快速原型，告诉设计者目标产品是否确实是用户友好的，即设计者是否已经考虑了必要的人为因素。

10.15 节将基于快速原型讨论复用问题。

10.15　复用快速原型

在快速原型被建造用来测试用户界面之后，它在软件过程的前期就被舍弃了。一种可选择的（但一般是不明智的）处理方法是，开发并不断精化快速原型直到它成为最终的产品。理论上，这种方法成就了快速软件开发，毕竟，并没有把组成快速原型的代码和构建它的相关知识一并扔掉，而是将快速原型转变成为最终产品。然而，实践上，整个过程与图 2-8 的编码 - 修复的方法极为相似。于是，根据编码 - 修复模型，这种快速原型的形式的第一个问题来自于这样的事实：在快速原型不断精化的过程中，要对工作着的产品进行修改。正如图 1-5 所示，这是一个代价很高的做法。第二个问题是，建造快速原型的主要目标是建造的速度。快速原型是（正确地）匆忙地凑在一起而成，而非经过仔细地规定、设计并实现的。没有规格说明文档和设计文档，维护生成的代码很困难而且代价很高。建造一个快速原型，然后扔了它再从头开始设计用户界面看起来可能很浪费，但是不论从短期还是长期来看，比起把快速原型转变成有产品级质量的软件，其代价要小很多［Brooks，1975］。

不过，有一个例子，允许精化一个快速原型，或者该快速原型的某些部分。当用户界面的快速原型的某些部分是由计算机生成时，这些部分可以用在最终产品里。CASE 工具（如屏幕生成器和报表生成器（5.5 节））常被用来生成用户界面，快速原型的这些部分可以实实在在地成为有产品级质量的软件的一部分。

10.16　需求工作流的 CASE 工具

本章的许多 UML 图反映出了协助需求工作流的图形工具的重要性。也就是说，需要一个画图的工具，使用户能很容易地画出相关的 UML 图。这样的工具有两个好处。首先，对存储在此工具中的图进行修改要比手工重新画图容易得多。其次，使用这种 CASE 工具时，产品的详细信息存储在 CASE 工具本身之中，所以文档总是可用的并且是最新的。

这样的 CASE 工具的缺点是它们并不总是具有用户友好性。一个强大的图形平台或环境有这样多的功能，以至于它往往都有一个陡峭的学习曲线，有时即使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 图如同手工画出的图那样令人满意几乎是不可能的。一种可选择的方法是花相当多的时间来调整由工具生成的图，然而，有时这种方法与手工画图一样慢。更糟糕的是，许多图形 CASE 工具的缺点是，无论在一个图上花费多少时间和努力，它也不可能像手工画的图那样完美。第三个问题是许多 CASE 工具很昂贵。不可能要求每个用户花费 5 000 美元或更多钱来购买复杂的 CASE 工具。另一方面，一些开源的此类 CASE 工具可以免费下载。总的来说，本节第一段里提到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（如 System Architect 和 Software through Pictures）已经扩展到可以支持 UML 图和面向对象范型。另外，还有像 IBM Rational Rose 和 Together 这样的面向对象的 CASE 工作平台和环境，也有这类开源的 CASE 工具（包括 ArgoUML）。

10.17　需求工作流的度量

需求阶段的一个关键特点是需求小组怎样能快速确定客户的真正需求。所以这个阶段的一个有用的度量是测量需求变更率。记录下需求工作流中需求变化的频度为管理者提供一种方式，来确定需求小组集中精力于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，如访谈或者形式分析。

另一个测量需求小组工作效率的度量标准是软件开发过程的其余工作流中需求更改的数量。对于需求中的每一个这样的修改，都应该记录该修改是由客户提出的还是由开发者提出的。如果在分析、设计和随后的工作流中由开发者提出修改大量的需求，那么显然需要对需求小组在需求阶段的工作方式进行全面的复查。相反，如果是客户在随后的工作流对需求提出反复的修改，那么这个度量可用于警示客户，变化目标的问题会对项目造成不利的影响，今后的修改应该被保持在最小值。

10.18　需求工作流的挑战

像软件开发过程中的其他工作流一样，需求工作流也存在潜在的问题和缺陷。首先，从开发过程开始，就要让目标产品的潜在用户能同心同德地进行合作，这是很重要的。一些人经常会对计算机化感到害怕，唯恐计算机会取代他们的工作。这样的担心有一定道理。在过去的 30 多年里，计算机化带来的影响是减少了对非技术工人的需求，但是为技术工人创造了更多的工作岗位。总体上，作为计算机化的一个直接的结果，其所创造出来的高收入的就业机会的数量远远超过了多余的相对非技术性的工作的数量，这点可以通过失业率的下降和平均工资的上涨得以佐证。但是，作为所谓的计算机时代的一个直接或间接的后果，全世界范围内众多国家不

平衡的经济增长，无法弥补对于那些由于计算机化而失业的人们所造成的负面影响。

重要的是，需求小组的每一个成员需要时刻意识到，他们有可能接触到的客户机构里的员工其实非常关心目标软件产品对自己工作潜在的影响。最糟糕的情况是，员工可能会故意误导或提供错误的信息，来确保产品无法满足客户需求，并因此来保护自己的工作。但是，即便不存在此类故意破坏，客户机构里的一些成员也可能不会提供多大的帮助，因为他们隐隐约约地感到了计算机化带来的威胁。

另一个对需求工作流的挑战是协商（negotiate）的能力。例如，通常重要的是要减少客户之所想。不足为奇，几乎每一位客户都想要一个能够做到所能想到的事情的软件产品。这样的一个产品需要花费令人无法接受的建造时间，并消耗远远超过客户认为是合理的资金。因此，通常有必要说服客户接受少于（有时是远少于）他们所想要的功能。为讨论中的每项需求计算成本和收益（参见 5.2 节）对这方面有所帮助。

所需的协商技巧的另一个例子是，在与目标产品功能相关的管理者之间达成妥协的能力。例如，一个狡猾的管理者可能试图通过纳入一个需求来扩展自己的权力，而这个需求可以通过将当前由另一个管理者负责的某个业务功能结合进自己负责的领域来实现。显然，其他管理者发现后就会强烈反对。需求小组必须让管理者双方坐下来并解决纠纷。

需求工作流的第三个挑战是，在许多机构里，拥有需求小组需要明确的信息的一些人，没有时间见面进行深入讨论。发生这种情况时，需求小组必须通知客户，由客户来决定两者哪个更重要——是这些人的当前工作职责，还是要建造的软件产品。如果客户无法坚持软件产品优先，那么开发者别无选择只好撤出该项目，因为它注定会失败。

最后，灵活性和客观性对需求获取也是必不可少的。需求小组的成员不带任何先见地着手每次访谈至关重要。特别地，访谈者必须绝对不从先前的访谈结果对需求做任何假设，也绝不在这些假设的框架下进行随后的访谈。相反，访谈者必须有意识地压制在先前访谈中收集的信息，并以一种公正的方式引导之后的访谈。做出有关需求的不成熟的假设是很危险的，在需求工作流做出任何关于要建造软件的假设都可能会造成惨重的损失。

如何执行需求工作流

- 迭代
  - 获得对应用域的理解
  - 拟订业务模型
  - 拟订需求
- 直到需求令人满意

本章回顾

本章最后的“如何执行需求工作流”总结了需求工作流的步骤。

本章以描述确定客户需求的重要性开始（10.1 节），随后是一个需求工作流的概述（10.2 节）。在 10.3 节中，描述了理解应用域的需要。如何拟订业务模型在 10.4 节描述。访谈和需求获取的其他技术在 10.4.1 节和 10.4.2 节进行讨论。10.4.3 节介绍了使用用例对业务模型进行建模。拟订初始需求在 10.5 节描述。在接下来的 6 节中，展示了 MSG 基金会实例研究的需求工作流。获得对应用域的初始理解在 10.6 节描述；相应的初始业务模型和初始需求在 10.7 节和 10.8 节中展示。然后，需求在 10.9 节和 10.10 节进行精化。最后，描述了 MSG 基金会实例研究的测试工作流（10.11 节）。在 10.12 节中，讨论了面向对象的统一过程的需求工作流。之后，在 10.13 节和 10.14 节详细展示了快速原型；10.14 节强调了为用户界面建造快速原型的重要性。10.15 节对快速原型的复用提出了一个警告。然后，讨论了需求工作流的 CASE 工具

(10.16 节) 和需求工作流的度量 (10.17 节)。本章最后描述了需求阶段面临的挑战 (10.18 节)。

延伸阅读材料

[Jackson, 1995] 对需求分析做了精彩的介绍。[Thayer and Dorfman, 1999] 是一本关于需求分析的论文集。Berry [2004] 提出，对需求不可避免的修改带来的连锁影响是造成没有软件工程灵丹妙药的原因 (参见备忘录 3.5)。使用成本 - 收益分析为需求设定优先级在 [Karlsson and Ryan, 1997] 中有描述。非功能需求在 [Cysneiros and do Prado Leite, 2004] 中进行了讨论。《IEEE Software》杂志 2005 年 1/2 月刊包含了关于需求的多篇文章。

统一过程的需求工作流在 [Jacobson, Booch and Rumbaugh, 1999] 的第 6 章和第 7 章有详细描述。误用实例 (对软件应避免的交互进行建模的用例) 在 [I. Alexander, 2003] 里有所描述。

对于快速原型的介绍，推荐的书目包括 [Connell and Shafer, 1989] 和 [Gane, 1989]。快速原型是快速应用开发 (Rapid Application Development, RAD) 的一个版本，在《IEEE Software》杂志 1995 年 9 月刊上有关于 RAD 的各种文章。原型的重要性在 [Schrage, 2004] 里有所描述。

用户界面设计方面的经典著作是 [Shneiderman, 2003]。制作优秀用户界面的方法在 [Holzinger, 2005] 里有所描述。在《IEEE Computer》杂志 2002 年 3 月刊和《Communications of the ACM》杂志 2003 年 3 月刊上可以找到关于用户界面方面的文章。计算机系统人为因素年会的学报 (由 ACM SIGCHI 主办) 是有关人为因素的各个方面的有价值的信息源。

习题

10.1　给出一个非功能性需求，它无需目标软件产品的相关详细信息就可以进行处理。

10.2　给出一个非功能性需求，它只有在需求工作流完成之后才能处理。

10.3　请说明用例和用例图的区别。

10.4　请说明用户和参与者的区别。

10.5　画出需求工作流的流程图。

10.6　在图 10-12 所示的用例图中，为什么同一对夫妇作为两种不同的参与者 (*申请者*和*贷款者*) 出现?

10.7　注意到只有 MSG 员工才能使用软件产品，那么为什么在图 10-12 所示的用例图中会出现*申请者*和*贷款者*两个参与者呢?

10.8　画出一张电子数据表格，展示 30 年期限结束时，每月 629.30 美元的分期还款可以还清 90 000 美元的贷款及其利息 (年利率以 7.5% 计，每月按复利计算)。

10.9　假设 MSG 基金会决定在其软件产品中加入抵押贷款申请过程。请给出 `Apply for an MSG Mortgage` 的用例描述，越详细越好。

10.10　10.9 节和 10.10 节描述了 MSG 基金会用例的重构。如果像习题 10.9 那样，将 `Apply for an MSG Mortgage` 用例加入到了需求模型中，那么重构应该如何变化呢?

10.11　你刚刚作为一个软件经理加入 Angel & Iguassu 软件公司。Angel & Iguassu 软件公司多年来一直为小型商店开发财务软件，它使用瀑布模型，且经常成功。根据你的经验，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该转到统一过程上来。记住，副总裁不喜欢长度超过半页纸的报告。

10.12　你是 Angel & Iguassu 公司负责软件开发的副总裁。请回答习题 10.11 的报告。

10.13　如果快速原型没有快速地建造，那么结果会怎样?

10.14　(*分析与设计项目*) 为习题 8.7 的图书自动循环系统执行需求工作流。

10.15 （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执行需求工作流。

10.16 （分析与设计项目）为习题 8.9 的自动提款机（ATM）执行需求工作流。

10.17 （学期项目）为附录 A 中的 Osric 办公用品和装饰项目执行需求工作流。

10.18 （实例研究）MSG 基金会的理事们决定扩展他们的活动，他们会向拥有足够高学分的当前贷款者的孩子提供一笔奖学金来帮助他们接受更高的教育。请画出用例 Apply for an MSG Scholarship，并给出其用例描述，越详细越好。

10.19 （实例研究）需要生成在过去一年里颁发的全部奖学金的报表。请适当修改图 10-35 和图 10-36 将这个额外的报表并入其中。

10.20 （实例研究）为 MSG 基金会实例研究的用户界面建造一个快速原型。使用指导老师规定的软件及硬件。

10.21 （软件工程读物）教师分发［Cysneiros and do Prado Leite，2004］的复印件。讨论这篇文章是如何改变了你对于非功能性需求重要性的看法？

参考文献

[I. Alexander, 2003] I. ALEXANDER, "Misuse Cases: Use Cases with Hostile Intent," *IEEE Software* **20** (January/February 2003), pp. 58–66.

[Berry, 2004] D. M. BERRY, "The Inevitable Pain of Software Development: Why There Is No Silver Bullet," in: *Radical Innovations of Software and Systems Engineering in the Future*, Lecture Notes in Computer Science, Vol. 2941, Springer-Verlag, Berlin, 2004, pp. 50–74.

[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.

[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.

[Cysneiros and do Prado Leite, 2004] L. M. CYSNEIROS AND J. C. S. DO PRADO LEITE, "Nonfunctional Requirements: From Elicitation to Conceptual Models," *IEEE Transactions on Software Engineering* **30** (May 2004), pp. 328–50.

[Gane, 1989] C. GANE, *Rapid System Development: Using Structured Techniques and Relational Technology*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[Holzinger, 2005] A. HOLZINGER, "Usability Engineering Methods for Software Developers," *Communications of the ACM* **48** (January 2005), pp. 71–74.

[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.

[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.

[Schrage, 2004] M. SCHRAGE, "Never Go to a Client Meeting without a Prototype," *IEEE Software* **21** (2004), pp. 42–45.

[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　分析工作流

学习目标

通过本章学习，读者应能：

- 执行分析工作流。
- 提取边界类、控制类和实体类。
- 执行功能建模。
- 执行类建模。
- 执行动态建模。
- 执行用例实现。

规格说明文档必须满足两个互相矛盾的要求。一方面，对可能不是计算机专家的客户而言，文档必须是清晰、容易理解的。毕竟，客户为产品付了钱，除非他相信自己真正理解新产品将是什么样的，否则他很有可能不批准开发该产品或者让其他的软件公司来开发。

另一方面，规格说明文档必须是完善、详尽的。事实上，它是拟定设计方案的唯一信息来源。尽管客户认同需求阶段的所有需求定义准确，但如果规格说明文档包含诸如遗漏、矛盾或模棱两可此类的错误，不可避免的结果就是，设计中的错误将被带到实现阶段中去。因此，需要一种技术，使目标产品能够以某种形式来描述，该技术既不具有太强的技术性，容易被客户理解，又足够精确，使得在开发周期最后阶段交付给用户的产品不包含错误。这些分析（规格说明）技术就是本章的主题。

11.1　规格说明文档

规格说明文档（specification document）是客户和开发人员之间的合约。它精确地规定了产品必须做什么，以及对产品有哪些约束。实际上，每份规格说明文档都包含了产品必须满足的约束条件。通常交付产品的最后期限也在文档中加以说明。还有一个普遍的约定是，“产品应该以某种方式安装以使其可与现有产品并行运行”，直到客户相信新产品确实满足了规格说明文档所有方面的要求。其他约束条件还包括可移植性：产品能够在同一种操作系统的不同硬件平台或者在不同的操作系统上运行。另一个约束条件是可靠性。如果产品是用于在特护病房中监视病人，那么最重要的是它能够全天 24 小时运行。快速响应时间可能也是一个要求，一个典型的约束条件是，“95% 的类型 4 的查询要在 0.25 秒内答复”。很多响应时间的约束条件不得不以概率术语表达，因为响应时间取决于计算机的当前负荷。相反地，严格的实时约束条件必须以绝对术语表示。例如，开发一个在导弹来袭的 0.25 秒内以 95% 的概率通知战斗机飞行员的软件是毫无用处的，这里的条件约束必须是 100%。

规格说明文档的一个至关重要的部分是验收标准集。从客户和开发人员的角度来看，清楚地写出一系列能够向客户证明产品确实符合规格说明文档要求和开发人员已完成任务的检验标准是重要的。验收标准的其中一部分可能是对约束条件的重申，而其余部分则针对不同的问题。例如，客户可能提供给开发人员对产品将要处理的数据的描述。那么相应的验收标准是产品正确地处理该类型的数据并且淘汰不相容的（即错误的）数据。一旦开发人员完全理解了问题，就可以提出可行的解决策略。*解决策略*（solution strategy）是建构产品的一种通用的方法。例如，对同一个产品，一种可行的解决策略是使用联机数据库；另一种则是使用常规的简单文件，并采用长时间批量运行的方法以提取所需的信息。在确定解决策略时，不考虑规格说明文档里

面的约束条件将是一个好主意。然后，再根据约束条件评估不同的解决策略，并进行必要的调整。确定一个特定的解决方案是否满足客户的约束条件有许多方法，一个常用的方法是使用概念验证原型（proof-of-concept prototype）技术。这是一项能很好解决用户界面问题和时间约束问题的技术，该技术在 2.9.7 节已讨论过。其他确定产品是否满足约束条件的技术包括仿真[Banks, Carson, Nelson and Nichol, 2001] 和分析网络建模（analytic network modeling）[Kleinrock, and Gail, 1996]。

在确定解决策略的过程中，许多解决策略会被提出，然后又会被舍弃，因此保存一份记录所有被舍弃的策略和它们被舍弃的原因的书面记录是重要的。该记录将有助于开发小组验证所选择的解决策略的正确性。更重要的是，产品交付后的维护阶段经常存在一种危险，即在增强维护过程中可能会提出一些新的但不明智的解决策略。对于开发中某些解决策略被否决的原因的记录，在交付后的维护阶段是极其有用的。

到软件生命周期的这个阶段，开发小组已经确定了一种或多种满足约束条件的解决策略。这时需要分两步作出决定。首先，是否建议客户计算机化？如果是，应该采用哪个可行的解决策略？对于第一个问题，完全可以在成本 - 效益分析（见 5.2 节）的基础上作出回答。其次，如果客户决定继续进行该项目，客户必须告知开发人员要采用的最优化准则，例如，总预算的最小化或投资回报的最大化。然后开发人员向客户建议最符合优化准则的解决策略。

11.2　非形式化规格说明

在很多开发项目中，规格说明文档是由一页页英文，或者其他自然语言（如法语或科萨语）文字组成。下面是一个典型的非形式化规格说明的一部分内容：

BV.4.2.5. 如果当前月的销售额低于目标销售额，那么打印一份报告，除非目标销售额和实际销售额的差额小于前一个月目标销售额和实际销售额的差额的一半或者当前月的目标销售额和实际销售额的差额低于5%。

产生该部分内容的背景如下：某零售连锁机构的管理高层每个月为每个商店制定一个销售额目标，如果某个商店不能达到这个目标，则打印出一份报告。考虑如下场景：假设某个商店 1 月份的销售额目标是 10 万美元，但实际销售额只有 6.4 万美元，即低于销售目标 36%。在这种情况下，需要打印一份报告。现在假设 2 月份的目标数字是 12 万美元，实际销售额只有 10 万美元，低于目标 16.7%。尽管销售额低于目标数字，但是 2 月份的百分比差值 16.7% 低于前一个月百分比差值 36% 的一半，管理高层认为有进步，不需要打印报告。再假设在 3 月份，销售额目标又是 10 万美元，但是商店完成的销售数字是 9.8 万美元，只低于目标 2%。因为这个百分比差值是很小的，且低于 5%，不需要打印报告。

仔细重读前面的规格说明可以发现，它与零售连锁商店管理高层实际的要求存在分歧。段落 BV.4.2.5 提到“目标销售额和实际销售额的差额”，并没有提到百分比差值。1 月份的差额是 3.6 万美元，2 月份的差额是 2 万美元。正如高层管理人员所期望的，百分比差值从 1 月份的 36% 降到 2 月份的 16.7%，低于 1 月份百分比差值的一半。然而，实际差额却从 3.6 万美元降到 2 万美元，大于 3.6 万美元的一半。因此，假如开发人员如实地实现了规格说明文档，那么应该打印该报告，而这并不是管理高层所需要的。接着最后一句提到“……差额低于5%”，当然这是指 5% 的百分比差值，然而“百分比”并不出现在上段规格说明的任何地方。

因此，这个规格说明文档中包含了不少错误。首先，忽视了客户的要求。其次，存在歧义，最后一句应该解读成“百分比差值……5%”或“差额……5 000 美元”，还是其他完全不同的意思？另外，行文的风格很差。该段说的是“如果某种情况发生，打印一份报告。然而，如果

另外一种情况发生，则不用打印报告。如果第三种情况发生，同样不用打印报告。”如果该规格说明文档只是简单地说明什么时候打印出一份报告，内容就会更清楚。总之，段落 BV. 4. 2. 5 不是一个编写规格说明文档的极佳例子。

段落 BV. 4. 2. 5 是虚构的，但是不幸的是，它代表了许多规格说明文档。你可能认为这个案例有些偏倚，假如让专业的规格说明文档撰写人员来编写规格说明文档，这类问题便不会发生。为了反驳这个观点，下面简单回顾一下第 6 章的小型案例研究。

11.3　小型案例研究的正确性证明回顾

回顾 6. 5. 2 节，1969 年 Naur 写了一篇关于正确性证明的论文［Naur，1969］。他用文本处理问题来阐述他的技术，利用这种技术，Naur 构造了一个 ALGOL 60 程序，并且非形式化地证明了程序的正确性。Naur 论文的一位评审［Leavenworth，1970］指出了程序中的一个错误。随后，London［1971］发现 Naur 程序中另外 3 个错误，给出了该程序的修正版本，并形式化地证明了它的正确性。Goodenough 和 Gerhart［1975］进一步发现了 London 还未发现的 3 个错误。在 London、Goodenough 和 Gerhart 发现的总共 7 个错误中，其中两个是分析错误。例如，Naur 的规格说明中没有指明，如果输入包含两个连续的分隔符（空格或换行字符）将会发生什么。因此，Goodenough 和 Gerhart 提出了一套新的规格说明。他们的规格说明比 6. 5. 2 节给出的 Naur 的规格说明要长 4 倍。

1985 年，Meyer 写了一篇关于形式化规格说明技术的文章［Meyer，1985］。文章的要点是，用自然语言如英语编写的规格说明不可避免产生矛盾、模棱两可和遗漏错误。他推荐使用数学术语来形式化地表述规格说明。Meyer 在 Goodenough 和 Gerhart 的规格说明中发现了 12 个错误，并开发出一套数学的规格说明来纠正这些错误。Meyer 接着解释了他的数学规格说明，并把它重新构造成英语的规格说明。据我之见，Meyer 的英语规格说明含有一个错误。Meyer 在其论文中指出，假如每行的最大字符数是 10，并且输入例如是“WHO WHAT WHEN”，那么按照 Naur 及 Goodenough 和 Gerhart 的规格说明，则有两个同样有效的输出：“WHO WHAT”在第一行和“WHEN”在第二行，或“WHO”在第一行和“WHAT WHEN”在第二行。事实上，Meyer 经过释义的英语规格说明同样包含这种模棱两可的错误。

关键是，Goodenough 和 Gerhart 的规格说明是他们为了修正 Naur 的规格说明而小心翼翼编写出来的。另外，Goodenough 和 Gerhart 的论文经历了两个版本，第一个发表在一个权威会议的论文集里，第二个发表在一本权威期刊上［Goodenough and Gerhart，1975］。并且，Goodenough 和 Gerhart 都是软件工程方面的专家，尤其是在规格说明方面。因此，如果连两名专家用尽可能多的时间小心翼翼编写的规格说明，都被 Meyer 发现有 12 个错误，那么一个普通计算机工作人员在时间的压力下编写一份毫无错误的规格说明的可能性有多大？更何况，文本处理问题的代码只有 25 ~ 30 行，而现实的产品可能包括几十万甚至几百万行源代码。

面向对象分析（Object-Oriented Analysis，OOA）是一种图形化的面向对象范型技术。关于 OOA 方面的技术，目前已有 60 多种，但所有的技术都是大同小异的。在本章的进一步阅读部分中，列举了各种已发表的关于不同技术的文章以及对这些技术进行比较的参考文献。然而，正如 3. 1 节说明的，如今，统一过程（Unified Process）［Jacobson，Booch，and Rumbaugh，1999］几乎是面向对象软件开发的首选方法。因此，在本章的第一部分和最后部分将主要介绍统一过程的分析工作流。

分析工作流是面向对象范型的重要组成部分，在这个工作流中，类被提取出来。用例和类是开发软件产品的基础。（想要了解更多面向对象范型，请参见备忘录 11. 1）。

备忘录 11.1

面向对象范型的几次重大发展大多发生在 1900 年到 1995 年之间。人们要接受一项新技术，通常需要 15 年的时间，按常理所以到了 2005 年之后面向对象范型才会开始广泛流行。然而，千禧虫（millennium bug）或者说 **Y2K** 问题加快了面向对象范型的流行。

在 20 世纪 60 年代，当计算机开始大规模应用于商业的时候，硬件要比现在昂贵很多。结果，那个时期的软件产品大多数表示日期时只用了后两位数字表示年份，并默认了前面的“19”。这种做法引发的问题是，年份 00 解释为 1900 年，而不是 2000 年。

当 70 年代和 80 年代硬件逐渐变得便宜的时候，很少经理会花一大笔钱来改写现有软件，使之用四位数字来表示日期。毕竟，在 2000 年到来之前，它也将成为其他人的问题。结果是，遗留系统（legacy systems）仍然与 2000 年相抵触。当最后期限，即 2000 年 1 月 1 日到来之际，软件公司被迫争分夺秒地修改他们的软件产品，因为没有任何办法可以推迟 2000 年的到来。

对很多遗留软件产品来说，负责维护的程序员面临的问题包括文档的缺失以及编写这些软件的程序语言已经过时。当修改一个现存的软件变得不可能时，唯一的办法是从新开发新产品。一些公司决定采用 COTS 技术（见 1.10 节），其他公司则认为需要开发新的客户软件。显然，经理们希望使用低成本高效率的现代技术来开发这些软件产品，也就意味着使用面向对象范型。因此，Y2K 问题是促进面向对象被广泛接受的一个重要因素。

11.4　分析工作流

统一过程［Jacobson，Booch and Rumbaugh，1999］中的分析工作流（analysis workflow）有两个目的。从需求工作流（上一个工作流）的角度来看，分析工作流的目的是更深刻地理解需求。另一方面，从设计流和实现流（分析工作流之后的工作流）的角度看，分析工作流的目的是，采用一种可使设计和实现更易于继续进行的方式来描述需求。

统一过程是由用例驱动的。在分析工作流中，用例是根据软件产品的类来描述的。统一过程包含三种类型的类：实体类、边界类和控制类。实体类（entity class）是对持久的信息进行建模。就一个银行软件产品来说，**Account Class** 是实体类，因为账户信息必须保留在软件产品中。对于 MSG 基金会软件产品，**Investment Class** 是实体类，而且投资信息必须持久存在。

边界类（boundary class）是对软件产品和它的参与者之间的交互进行建模。通常边界类与输入和输出有关联。例如，在 MSG 基金会软件产品中，需要打印该基金会所有投资的报告和现持有的所有抵押的报告。这意味着边界类 **Investments Report Class** 和 **Mortgages Report Class** 是需要的。

控制类（control class）是对复杂的计算和算法进行建模。在 MSG 基金会软件产品的例子中，计算一周可用基金的算法就是一个控制类，即 **Estimate Funds for Week Class**。

这三种类的 UML 符号如图 11-1 所示。这些符号是构造型（stereotypes），即 UML 的扩展。UML 的一个优点是它允许定义那些不属于 UML，但在对特定系统精确建模时需要用到的结构。



图 11-1　表示一个实体类、边界类和控制类的 UML 构造型（UML 扩展机制）

正如本节开始提到的，在分析工作流中，用例是根据软件产品的类来描述的。统一过程本身并不描述如何提取类，因为统一过程的使用者要求具有面向对象分析和设计的背景知识。

在这里，我们将暂时中断统一过程的讨论以解释如何提取类。我们将在 11.18 节回到统一过程这个主题来。

这里先考虑实体类，即对持久信息进行建模的类。

11.5　实体类的提取

实体类的提取包括三个迭代和增量式执行的步骤：

1）功能建模（functional modeling）。给出所有用例的场景（场景（scenario）是用例的一个实例）。

2）实体类建模（entity class modeling）。确定实体类和它们的属性。然后，确定实体类之间的联系和交互行为，用类图表示这些信息。

3）动态建模（dynamic modeling）。确定每个实体类或其子类执行或对之执行的操作，用状态图表示这些信息。

然而，就像所有迭代增量过程一样，这三个步骤不需要总按这个顺序执行。一个模型的变化常常会引起其他两个模型相应的改变。

为了说明步骤是如何进行的，下面将对电梯问题的一个修改版本进行实体类提取。要了解电梯问题的背景，请参见备忘录11.2。

备忘录11.2

电梯问题是软件工程中的一个经典问题。它首次出现在1986年出版的Don Knuth的里程碑式作品《计算机程序设计艺术》[⊖]（The Art of Computer Programming）的第1卷中[Knuth，1968]。它模拟的是加州理工学院的数学楼里的一个电梯，目的是说明如何使用虚构程序语言MIX来编写协同程序。

到了20世纪80年代中期，电梯问题已经扩展到n个电梯。另外，问题解的某些特定性质必须证明，例如，电梯最终是否能在一段有限的时间内到达。它现在已成为了形式化（数学基础的）规格说明语言领域中研究人员所要研究的问题，并且任何新提出的形式化规格说明语言都必须对电梯问题进行阐述。

1986年，电梯问题出现在《ACM SIGSOFT Software Engineering Notes》上的第4届International Workshop on Software Specification and Design的会议征文[IWSSD，1986]中，这使得它被广泛关注。该会议于1987年在加尼福里亚Monterey举行，在研究人员提交的文章中，电梯问题是5个典型问题之一。在会议征文中，它被叫做“lift problem”，由STC-IDEC公司（位于英国StevenageStandard Telecommunications and Cable的分公司）的N.[Neil] Davis命名。

从那时起，电梯问题开始得到更广泛的关注，并且用于阐述各种不同的软件工程方面的技术，而不仅仅用于形式化规格说明语言。本书用它来阐述各项技术，读者将很快发现，电梯问题绝非看起来那么简单。

11.6　电梯问题

该问题的逻辑原理是满足以下约束条件在m个楼层之间移动n个电梯：

1）每一个电梯有m个按钮（button），每个按钮对应一层。当有人按下按钮时，按钮变亮并指示电梯到相应的楼层。当该电梯到达相应的楼层时，按钮变暗。

2）除了一楼和顶楼外，每层有两个按钮，一个请求电梯向上，一个请求电梯向下。按钮按下时变亮。当一个电梯到达该层并往请求方向移动的时候，按钮将变暗。

⊖ 本书第1卷的影印版及双语版已由机械工业出版社出版，书号分别为ISBN 978-7-111-22709-0和ISBN 7-111-18031-3。

3）当一个电梯没有请求时，它停留在当前楼层，电梯门关闭。

问题中有两组按钮。在 n 个电梯中，每个电梯有 m 个按钮，每个按钮对应一层。因为这 $n \times m$ 个按钮在电梯里面，我们称它们为*电梯按钮*。然后，每一个楼层有两个按钮，一个请求电梯向上，一个请求电梯向下。它们称为*楼层按钮*。每一个按钮可处于两个状态之一：开（变亮，illuminated）或关。最后，假设，在电梯门打开后，它们在超时（timeout）后会再度关闭。

OOA 的第一步就是对用例进行建模。

11.7　功能建模：电梯问题案例研究

用例（use case）描述待构建的软件产品与*参与者*（actor，软件产品的外部用户）之间的交互。用户和电梯之间唯一可能的交互就是，用户按下一个电梯按钮去命令一部电梯或者用户按下一个楼层按钮请求电梯停在某个特定楼层，所以这里有两个用例：Press an Elevator Button（按下一个电梯按钮）和 Press a Floor Button（按下一个楼层按钮）。这两个用例用图 11-2的用例图（10.7 节）表示。

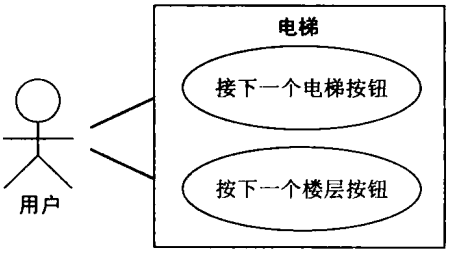


图 11-2　电梯问题案例研究用例图

用例提供整体功能的一般描述；场景是用例的一个特定实例，就像对象是类的一个实例。一般来说，场景有很多个，每个场景代表一组特定的交互。在本节中，我们考虑图 11-3 的场景，它包含了这两个用例的实例化。

图 11-3 描述了一个*正常场景*（normal scenario），即根据我们对电梯的理解，所发生的一组在用户和电梯之间的交互动作。图 11-3 的场景是在仔细地观察不同用户与电梯（或者更精确地说，是跟电梯按钮和楼层按钮）之间的交互后，构造起来的。这 15 个被标号的事件详细描述了用户 A 和电梯系统的两次交互（事件 1 和事件 6），还有电梯系统各组件执行的操作（事件 2 ~ 5 和事件 7 ~ 15）。两个事件即“用户 A 进入电梯”和“用户 A 离开电梯”没有被标号。像这种情况，实质上是注释，用户 A 进入或离开电梯时并没有与电梯的组件发生交互。

1. 用户 A 在 3 楼按下向上的楼层按钮，请求一部电梯的服务。用户 A 想上 7 楼
2. 向上的楼层按钮被打开
3. 一部电梯到达 3 楼。电梯里面有用户 B，他是在 1 楼进入电梯，并按下了去 9 楼的电梯按钮
4. 电梯门打开
5. 定时器开始计时
   用户 A 进入电梯
6. 用户 A 按下去 7 楼的电梯按钮
7. 去 7 楼的电梯按钮被打开
8. 电梯门在超时后关闭
9. 向上的楼层按钮被关上
10. 电梯移动到 7 楼
11. 去 7 楼的电梯按钮被关上
12. 电梯门打开，用户 A 离开电梯
13. 定时器开始计时
    用户 A 离开电梯
14. 电梯门在超时后关闭
15. 电梯与用户 B 向 9 楼移动

图 11-3　一个正常场景的第一次迭代

相反，图 11-4 是一个异常场景（exception scenario），它描述了当用户在 3 楼按下向上的按钮但实际上他是想去 1 楼时，所要发生的情况。这个场景同样是观察了很多在电梯里的用户的行为构造出来的，因为使用过电梯的人会认识到用户有时候会按错按钮。

1. 用户 A 在 3 楼按下向上的楼层按钮，请求一部电梯的服务。用户 A 想要去 1 楼
2. 向上的楼层按钮被打开
3. 一部电梯到达 3 楼。电梯里面有用户 B，他是在 1 楼进入电梯，并按下了去 9 楼的电梯按钮
4. 电梯门打开
5. 定时器开始计时
   用户 A 进入电梯
6. 用户 A 按下去 1 楼的电梯按钮
7. 去 1 楼的电梯按钮被打开
8. 电梯门在超时后关闭
9. 向上的楼层按钮被关上
10. 电梯移动到 9 楼
11. 去 9 楼的电梯按钮被关上
12. 电梯门打开以让用户 B 离开电梯
13. 定时器开始计时
    用户 B 离开电梯
14. 电梯门在超时后关闭
15. 电梯与用户 A 向 1 楼移动

图 11-4　一个异常场景

图 11-3 和图 11-4 的场景加上无数个其他场景，都是图 11-2 用例的特定实例。OOA 小组应该要研究足够多的场景，使得对将要建模的系统行为有一个全面的理解。这些信息将用于下一步，即实体类建模，以确定实体类。

11.8　实体类建模：电梯问题案例研究

这一步提取实体类和它们的属性，并用 UML 类图来表示。这时只确定实体类的属性，不包括方法，后者将在设计流中被指定给类。

整个面向对象范型的一个特征，就是各个不同步骤不易执行。幸运的是，使用对象的优点会让这些付出的精力有所值。因此，在分析工作流的开始部分提取实体类和它们的属性通常很难一次完成，这不足为怪。

一种确定实体类的方法是从用例推导出实体类。也就是说，开发人员仔细研究所有场景，包括正常的和异常的，并找出在用例中起作用的组件。只从图 11-3 和图 11-4 的场景可以看到，候选实体类是电梯按钮、楼层按钮、电梯、门和定时器。我们将会看到，候选实体类跟在实体类建模期间提取出的实际的类是很接近的。然而，一般来说，场景有很多个，结果可能的类也很多。缺乏经验的开发人员可能倾向于从场景中推导出太多的类。这不利于实体类建模，因为移除一个多余的实体类要比添加一个新的实体类困难得多。

确定实体类的另一个方法是使用 CRC 卡片（见 11.8.2 节），当开发人员具备特定领域专业知识时，这种方法是很高效的。然而，如果开发人员在应用领域里没有或只有很少经验，那么建议用 11.8.1 节描述的名词提取方法。

11.8.1　名词提取

对于没有领域专业知识的开发人员，一个好的方法是使用下面的两阶段*名词提取方法*（noun-extraction method），先提取候选实体类，然后对结果进行细化。

阶段 1：用一段话描述软件产品

对电梯问题案例研究，一种可能的描述方式如下：

电梯里和楼层的按钮控制一幢 m 层大楼里的 n 个电梯的移动。当按下请求电梯停在某一特定楼层的按钮时，按钮变亮；当满足该请求时，发亮中断。当一个电梯没有请求时，它停留在当前层，电梯门关闭。

阶段 2：识别名词

先在非形式化策略中识别出名词（不包括那些在问题边界外的），然后把这些名词用作候选实体类。现在制定该非形式化策略，这次识别出的名词用黑体印刷。

电梯里和楼层的按钮控制一幢 m 层大楼里的 n 个电梯的移动。当按下请求电梯停在某一特定楼层的按钮时，按钮变亮；当满足该请求时，发亮中断。当一个电梯没有请求时，它停留在当前层，电梯门关闭。

有 8 个不同名词：按钮、电梯、楼层、移动、大楼、发亮、请求和电梯门。其中三个名词（楼层、大楼和电梯门）在问题边界外，所以被排除。剩下名词中的三个（移动、发亮和请求）是抽象名词，即它们标志着没有物理存在的事物。一个实用的经验法则是，抽象名词很少是类，而往往是类的属性。例如，发亮是按钮的属性（attribute）。因此剩下的两个名词则为候选实体类：**Elevator Class**（电梯类）和 **Button Class**（按钮类）。（UML 习惯用粗字体表示类名，并大写类名中每个单词的首字母。）

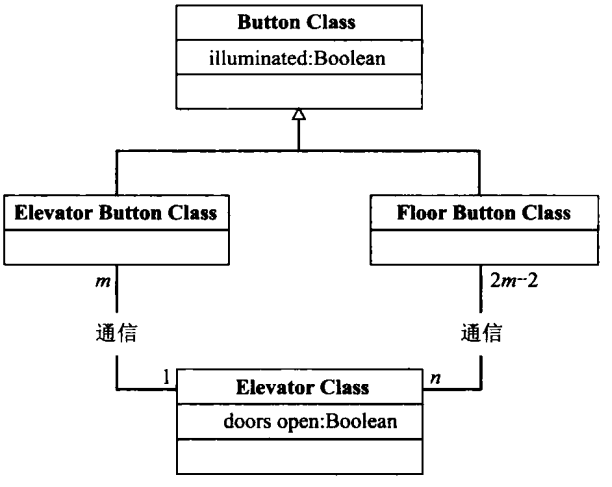


图 11-5　电梯问题案例研究类图的第一次迭代

结果得到的类图（class diagram）如图 11-5 所示。**Button Class** 有布尔类型属性 illuminated（变亮）对图 11-3 和图 11-4 场景中的事件 2、7、9 和 11 进行建模。问题规定了两种类型的按钮，所以 **Button Class** 有两个子类：**Elevator Button Class**（电梯按钮类）和 **Floor Button Class**（楼层按钮类）（在 UML 中空心三角形表示继承）。**Elevator Button Class** 和 **Floor Button Class** 的每个实例与 **Elevator Class** 的实例发生关联。后者有布尔属性 doors open（电梯门打开）对两个场景的事件 4、8、12 和 14 进行建模。

但是在现实的电梯中，按钮不是直接与电梯之间发生作用的。如果只决定指派某电梯去响应某个特定的请求，那么需要电梯控制器（elevator controller）。然而，问题描述并没有提到电梯控制器，所以它在名词提取过程没有被选择作为实体类。换句话说，本节所介绍的技术为找出候选实体类提供了一个思路，但肯定不能依赖它做更多的事。

把 **Elevator Controller Class**（电梯控制器类）添加到图 11-5 中，便产生了图 11-6。这样做当然更有意义。另外，图 11-6 中现在只有一对多关系，对它们建模要比对图 11-5 中的多对多关系建模要容易。因此看起来，此时进入到第 3 阶段是合理的，请记住任何时候都可能再返回实体类建模，甚至到后面的实现流阶段。然而，进行动态建模之前，我们考虑另一种实体类建模技术。

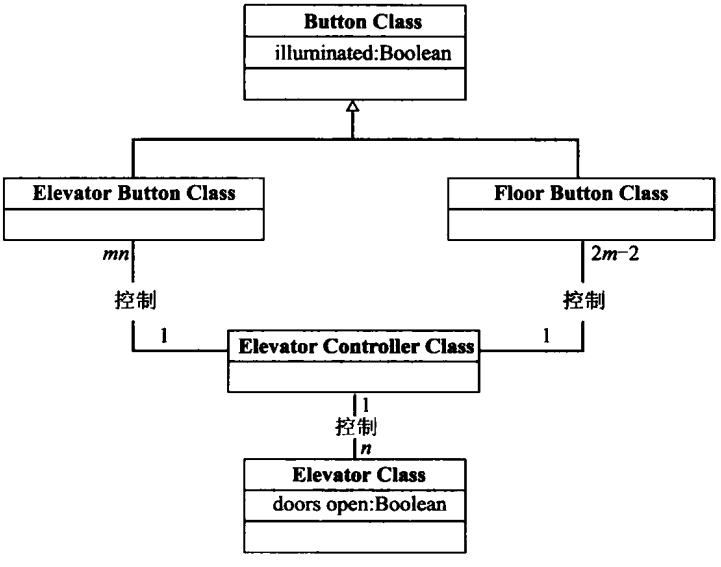


图 11-6　电梯问题案例研究类图的第二次迭代

11.8.2　CRC 卡片

类 - 职责 - 协作（Class-Responsibility-Collaboration，CRC）卡片应用于分析工作流中已经很多年了［Wirfs-Brock，Wilkerson，and Wiener，1990］。对每一个类，软件开发小组填写一张卡片，包括该类的名称、功能（职责）和被它调用以完成其功能的一组类（协作）。

这种方法已被扩展为多种形式。首先，CRC 卡片通常直接包含了类的属性和方法，而不是用某种自然语言所表达的“职责”。其次，这个技术已经发生变化。一些组织不再使用卡片，而是把类名写在报事帖上，报事帖可在白板上移动，用它们之间的连线表示协作关系。如今整个过程能够自动化进行，诸如 System Architect 这样的 CASE 工具包括了在屏幕上生成和更新 CRC“卡片”的组件。

CRC 卡片的优点在于：当开发小组使用它时，小组成员之间的交互可以发现类里面遗漏的或错误的字段，不管是属性还是方法。此外，使用 CRC 卡片可以描述类之间的关系。一个行之有效的方法是在小组成员间分发卡片，然后小组成员扮演出类的职责。某个成员可能会说“我是 **Date Class**（日期类），我的职责是创建新的日期对象。”另一个小组成员可能会接着说，他需要 **Date Class** 的额外功能，如把日期转换为一个整数，就是距离 1900 年 1 月 1 日的天数，所以要得出两个日期之间的天数就可以简单地把两个对应的整数相减（参见备忘录 11.3）。因此，扮演 CRC 卡片的职责，是验证类图是否完善和正确的一个有效手段。

如前面所提到，CRC 卡片的一个不足之处在于：除非小组成员在相关应用领域有相当丰富的经验，否则它通常不是一个识别实体类的好方法。另一方面，一旦开发人员已经确定大多数的类，并知道了它们的职责和协作关系，那么 CRC 卡片是完成整个过程并确保一切正确的一个极好的方法。这将在 11.10 节中描述。

备忘录 11.3

如何计算从 1999 年 2 月 21 日到 2007 年 8 月 16 日之间的天数？这类减法在很多财务系统中被用到，如计算支付利息或确定将来现金流的现值。通常的方法是把每个日期转换为一个整数，表示距离某个特定起始日期的天数。问题是，在使用哪个起始日期上我们无法取得一致意见。

天文学家使用 Julian 日数，即距离公元前 4713 年 1 月 1 日格林威治平午的天数。这个系统由 Joseph Scaliger 在 1582 年发明，并根据他的父亲 Julius Caesar Scaliger 命名。（如果你确实想要了解为什么选择公元前 4713 年 1 月 1 日，请参阅［USNO，2000］。）

Lilian 日期是距离 1582 年 10 月 15 日以来的天数，该日是格里高利历（Gregorian calendar）的第一日，由罗马教皇 Pope Gregory XIII 推行。Lilian 日期根据 Luigi Lilio 命名，他是格里高利历改革的主要倡导者。Lilio 负责推导出格里高利历大多算法，包括闰年规则。

至于软件，COBOL 内部函数使用 1600 年 1 月 1 日作为起始日期来计算整数日期。然而，在 Lotus 1-2-3 的带领下，几乎所有的电子制表软件都使用 1999 年 1 月 1 日作为起始日期。

11.9　动态建模：电梯问题案例研究

动态建模的目标是生成每个类的状态图（statechart），描述目标产品的动态行为。首先，考虑 **Elevator Controller Class**。为了简便起见，只考虑一个电梯。**Elevator Controller Class** 相关的状态图如图 11-7 所示。

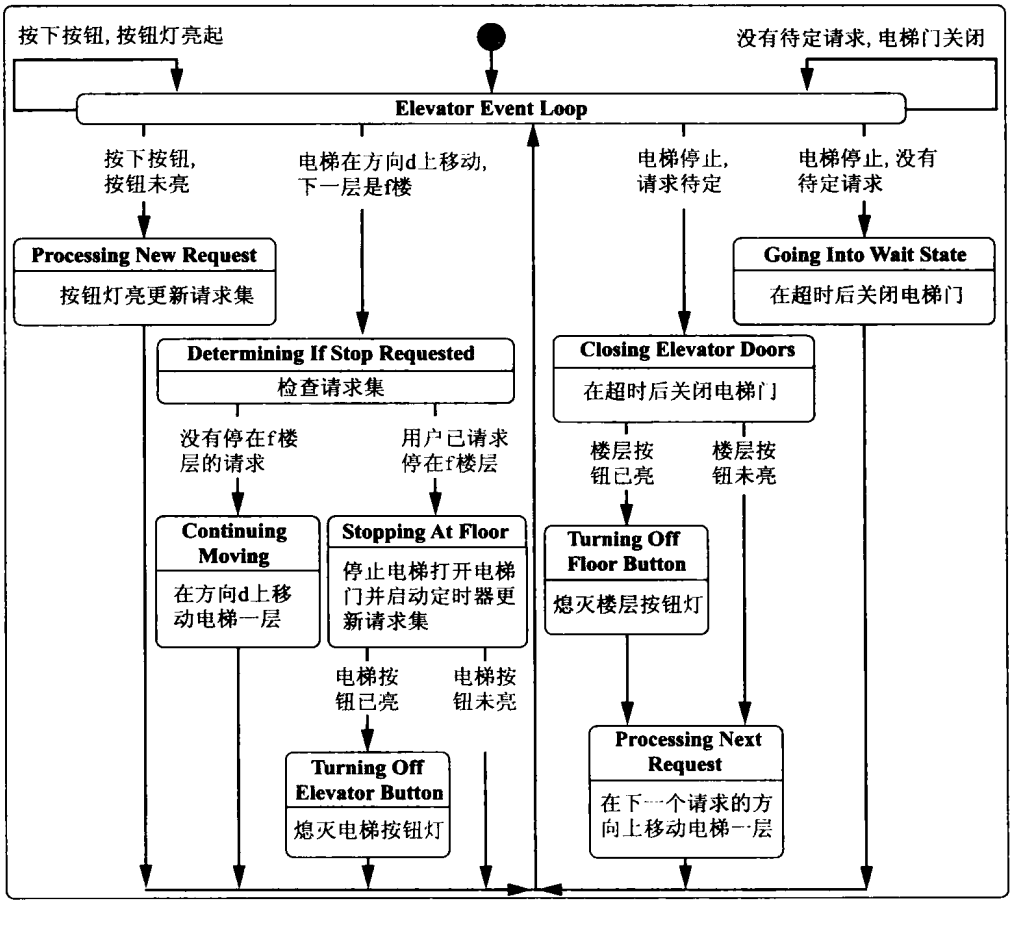


图 11-7　**Elevator Controller Class** 状态图的第一次迭代

状态图包含了状态、事件和行为。这里关键概念是状态（state）的含义。类的属性有时称为状态变量（state variable）。这个术语的由来是，在大多数面向对象实现中，产品的状态是由不同组件对象属性的值决定的。一个事件（event）的发生导致产品进入其他状态。最后，谓词的值为真或假。

状态、事件和谓词分布在状态图中。例如，如果当前状态是 **Elevator Event Loop** 并且事件“电梯停止，没有请求待定”为真，则进入图 11-7 中的状态 **Going Into Wait State**。当进入状态 **Going Into Wait State**，操作“在超时后关闭电梯门”被执行。

考虑图 11-3 场景的第一部分。事件 1 是“用户 A 在 3 楼按下向上的楼层按钮”。现在考虑图 11-7 的状态图。实心圆表示初始状态，该状态将系统带入状态 **Elevator Event Loop**。沿着最左边的垂线，如果按钮按下时是不亮的，则系统进入图 11-7 的状态 **Processing New Request**，并且打开了按钮。紧接着的状态是 **Elevator Event Loop**。

在图 11-3 场景中的事件 3 是电梯到达 3 楼。回到图 11-7 的状态图，考虑电梯接近 3 楼时将会发生什么。因为电梯是处在运动中的，下一个进入的状态是 **Determining If Stop Requested**。检查请求集，因为用户 A 已经请求电梯停在该楼，所以下一个状态是 **Stopping At Floor**。电梯停在 3 楼，电梯门打开，并且定时器开始计时。去 3 楼的电梯按钮没有按下，因此下一个状态是 **Elevator Event Loop**。

用户 A 进入电梯并按下去 7 楼的电梯按钮。因此，下一个状态又是 **Processing New Request**，接着，同样是状态 **Elevator Event Loop**。此时电梯已经停止，两个请求待定，所以下一个状态是 **Closing Elevator Doors**，电梯门在超时后关闭。3 楼的楼层按钮已由用户 A 按下，所以接下来的状态是 **Turning Off Floor Button**，则关上楼层按钮。接着是状态 **Processing Next Request**，电梯开始向 4 楼移动。

由前面的讨论中，我们可以发现图 11-7 是从场景中构造出来的，这并不足为奇。更确切地说，场景中那些特定的事件被一般化了。例如，考虑图 11-3 场景中的第一个事件“用户 A 在 3 楼按下向上的楼层按钮”。这个特定事件被一般化为按下任意一个按钮（楼层按钮或电梯按钮）。接着，有两种可能，或者按钮已经是亮的（在这种情况下没有什么事件发生），或者按钮是不亮的（在这种情况下电梯必须采取行动处理用户的请求）。

为了对这个事件建模，在图 11-7 给出了 **Elevator Event Loop** 状态。通过图 11-7 左上角的事件“按下按钮，按钮已亮”导致空操作循环，对按钮已亮的情况进行建模。另一种情况，通过标注有事件“按下按钮，按钮未亮”并指向状态 **Processing New Request** 的箭头，对按钮未亮的情况进行建模。从场景中的事件 2 可以清楚地看出，电梯处于这个状态时需要执行操作“打开按钮”。另外，用户按下任意一个按钮的目的是请求一部电梯（楼层按钮）或请求电梯移动到特定楼层（电梯按钮），所以在 **Processing New Request** 状态需要执行操作“更新请求集”。

现在考虑场景中的事件 3“一个电梯到达 3 楼”。这可推广到电梯在任意楼层间移动的情景。电梯的运动可建模成事件“电梯在方向 d 上移动，下一层是 f 楼”和状态 **Determining If Stop Requested**。但是仍然有两种可能，或者有要求停在 f 楼的请求，或者没有这样的请求。在前一种情况，对应事件“没有停在 f 楼的请求”，显然电梯必须处于方向 d 的上一层，而状态为 **Continuing Moving**。在后一种情况中（对应事件“用户已请求停在 f 楼”），从图 11-3 场景可以清楚知道有必要执行“停止电梯”（事件 3）和“打开电梯门并启动定时器”（事件 4 和 5），要执行这些操作，需要状态 **Stopping At Floor**。另外，与状态 **Processing New Request** 类似，在 **Stopping At Floor** 状态需要“更新请求集”。对场景中的事件 9 一般化，需要注意到如果楼层按钮亮着的话，它应该被关上。这可建模成状态 **Turning Off Floor Button**，表示该状态的方框上面有两个事件。同样地，对场景中的事件 11 一般化，意味着如果电梯按钮亮着的话，它应该被关上。这可建模成状态 **Turn Off Elevator Button**，表示该状态的方框上面有两个事件。

对图 11-3 场景中的事件 8 一般化，产生状态 **Closing Elevator Doors**，对事件 10 一般化，产

生状态“处理下一个请求”。然而，状态 **Going Into Wait State** 和事件“没有请求待定，电梯门关闭”是从对另一个不同场景的事件一般化推导出的，在该场景的这个事件中用户离开电梯，并且没有按钮是亮着的。

其他类的状态图相对比较简单，留作练习（习题 11.6）。

11.10　测试工作流：电梯问题案例研究

到这一步，功能、实体类和动态模型看起来都已经建好，接着进行测试工作流。下一步是检查之前完成的分析工作流。检查工作的一个部分，正如在 11.8.2 节提到过的，需要用 CRC 卡片。

相应地，对每个实体类填写 CRC 卡片：**Button Class**、**Elevator Button Class**、**Floor Button Class**、**Elevator Class** 和 **Elevator Controller Class**。图 11-8 所描述的 **Elevator Controller Class** 的 CRC 卡片是从图 11-5 的类图和图 11-7 的状态图推导出来的。更详细地说，**Elevator Controller Class** 的职责（RESPONSIBILITY）是通过列举出 **Elevator Controller Class** 状态图（图 11-7）中的所有操作而得到的。通过分析图 11-6 的类图，可以确定 **Elevator Controller Class** 的协作者（COLLABORATION），也可看到 **Elevator Button Class**、**Floor Button Class** 和 **Elevator Class** 与 **Elevator Controller Class** 之间有交互关系。

这个 CRC 卡片反映了面向对象分析第一次迭代中的两个主要问题。

1）考虑职责 1“打开电梯按钮”。在面向对象范型里，这个命令是不合适的。从职责驱动设计（responsibility-driven design）（1.9 节）的观点来看，**Elevator Button Class** 的对象（实例）负责将自己打开或关上。另外，从信息隐藏（7.6 节）的观点来看，**Elevator Controller Class** 应该不知道 **Elevator Button Class** 打开一个按钮的内部工作机制。正确的职责应该是：发送一条消息给 **Elevator Button Class**，使它将自己打开。图 11-8 的职责 2～6 需要类似的调整。这 6 个调整反映在图11-9——**Elevator Controller Class** 的 CRC 卡片的第二次迭代中。

类 **Elevator Controller Class**
职责 1. 打开电梯按钮 2. 关上电梯按钮 3. 打开楼层按钮 4. 关上楼层按钮 5. 向上移动电梯一层 6. 向下移动电梯一层 7. 打开电梯门并启动定时器 8. 在超时后关闭电梯门 9. 检查请求集 10. 更新请求集
协作者 1. **Elevator Button Class** 2. **Floor Button Class** 3. **Elevator Class**

图 11-8　对 **Elevator Controller Class** 的 CRC 卡片的第一次迭代

类 **Elevator Controller Class**
职责 1. 发送消息给 **Elevator Button Class**，使打开按钮 2. 发送消息给 **Elevator Button Class**，使关上按钮 3. 发送消息给 **Floor Button Class**，使打开按钮 4. 发送消息给 **Floor Button Class**，以关上按钮 5. 发送消息给 **Elevator Class**，使向上移动一层 6. 发送消息给 **Elevator Class**，使向下移动一层 7. 发送消息给 **Elevator Doors Class**，使打开 8. 启动定时器 9. 在超时后发送消息给 **Elevator Doors Class**，使关闭 10. 检查请求集 11. 更新请求集
协作者 1. **Elevator Button Class**（子类） 2. **Floor Button Class**（子类） 3. **Elevator Doors Class** 4. **Elevator Class**

图 11-9　对 **Elevator Controller Class** 的 CRC 卡片的第二次迭代

2）某个类被忽略。回到图 11-8，考虑职责 7“打开电梯门并启动定时器”。状态的概念有助于确定某个组件是否需要被建模成类。如果被考虑的组件包含某个在实现过程中发生变化的状态，那么它很有可能被建模成一个类。显然地，电梯门包含一个状态（开或关），所以 **Elevator Doors Class**（电梯门类）应该是一个类。

为什么 **Elevator Doors Class** 应该是一个类，还有另一个原因。面向对象范型允许状态隐藏在对象里以防止被非法改变。如果存在某个 **Elevator Doors Class** 的对象，打开或关闭电梯门的唯一的方法是发送一条消息给 **Elevator Doors Class** 对象。在错误的时间打开或关闭电梯门，可能会导致严重的事故，请参见备忘录 11.4。因此，对某些类型的产品，安全方面考虑应该包含到第 7 章和第 8 章所介绍面向对象的其他优势之中。

备忘录 11.4

几年前，我在一栋大楼的第 10 层不耐烦地等着电梯的到来。看到电梯门打开，我正要往前跨，却发现那里没有电梯。当我正要进入电梯时看到了一片漆黑，直觉告诉我电梯出现了问题因此救了我一命。

也许，如果该电梯控制系统是用面向对象范型开发的，那么第 10 层电梯门的不合时宜打开的事件便可以避免。

增加 **Elevator Doors Class** 意味着图 11-8 的职责 7 和职责 8 需要调整，类似地，从职责 1 到职责 6 也同样需要调整。也就是说，需要发送消息给 **Elevator Doors Class** 的实例，使它们自己打开或关闭。但是另外有一个问题，职责 7 是“打开电梯门并启动定时器”。

这个职责必须分解成两个单独的职责。当然，必须发送一条消息给 **Elevator Doors Class** 使打开门。但是，定时器是 **Elevator Controller Class** 的一部分，因此启动定时器是 **Elevator Controller Class** 自己的职责。**Elevator Controller Class** CRC 卡片的第二次迭代（图 11-9）表明该职责的分离已圆满完成。

除了图 11-8 的 CRC 卡片反映的两个主要问题外，**Elevator Controller Class** 的职责“检查请求集”和“更新请求集”需要增加属性 `requests`（请求）到 **Elevator Controller Class** 中。在这个阶段，`requests` 只是简单地定义为类型 `requestType`，`requests` 的数据结构将在设计流中选择。

修改过的类图如图 11-10 所示。由于对类图进行了调整，用例图和状态图也应重新检查，看它们是否也需要进一步改进。显然这里用例图不用修改。但是，必须重新调整图 11-7 的状态图中的操作以反映图 11-9 中的职责（CRC 卡片第二次迭代），而非图 11-8 中的职责（第一次迭代）。

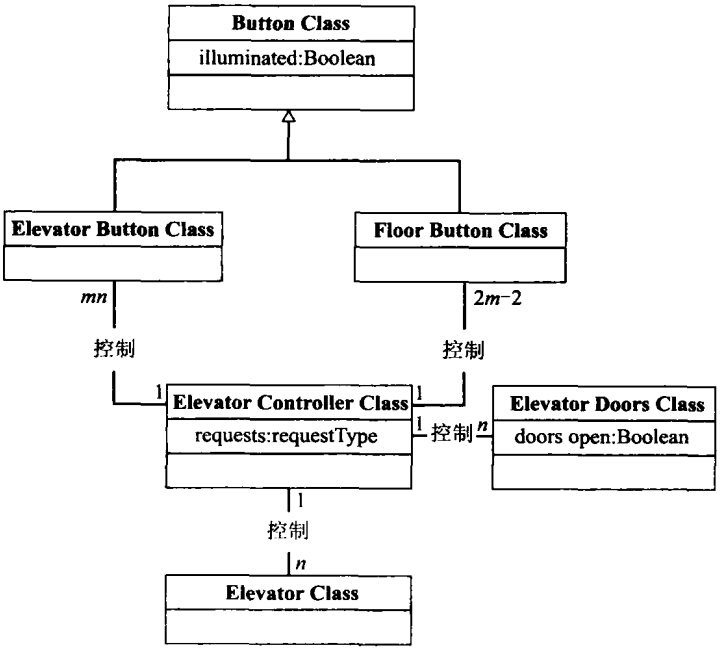


图 11-10　电梯问题案例研究类图的第三次迭代

另外，状态图集合必须扩展以包含新增的类。场景也需要更新以反映这些变化，图 11-11 显示了图 11-3 场景的第二次迭代。即使所有的修改都已完成并检查过（包括刚整过的 CRC 卡片），有必要的话，在设计流中仍然可能会返回到分析工作流，并修改一个或多个分析制品（artifacts）。在这个阶段，看起来电梯问题案例研究中的实体类已正确提取完毕。

1. 用户 A 在 3 楼按下向上的楼层按钮，请求一部电梯。用户 A 想要到 7 楼
2. 楼层按钮通知电梯控制器该楼层按钮已被按下
3. 电梯控制器发送一条消息给向上的楼层按钮，使它打开自己
4. 电梯控制器发送一条消息给电梯，使它向上移动到 3 楼。电梯里面有用户 B，用户 B 之前在 1 楼进入电梯并按下去 9 楼的电梯按钮
5. 电梯控制器发送一条消息给电梯门，使它们打开
6. 电梯控制器启动定时器
   用户 A 进入电梯
7. 用户 A 按下去 7 楼的电梯按钮
8. 电梯按钮通知电梯控制器该电梯按钮已被按下
9. 电梯控制器发送一条消息给去 7 楼的楼层按钮，使它打开自己
10. 电梯控制器发送一条消息给电梯门，使它们在超时后关闭
11. 电梯控制器发送一条消息给向上的楼层按钮，使它关上自己
12. 电梯控制器发送一条消息给电梯，使它向上移动到 7 楼
13. 电梯控制器发送一条消息给去 7 楼的电梯按钮，使它关上自己
14. 电梯控制器发送一条消息给电梯门，使它们打开以让用户 A 离开电梯
15. 电梯控制器启动定时器
    用户 A 离开电梯
16. 电梯控制器发送一条消息给电梯门，使它们在超时后关闭
17. 电梯控制器发送一条消息给电梯，使它自己和用户 B 向上移动到 9 楼

图 11-11　电梯问题案例研究的一个正常场景的第二次迭代

11.11　提取边界类和控制类

与实体类不同，通常边界类比较容易提取。一般来说，每个输入屏幕、输出屏幕和打印报告都可建模成它自己的边界类。类包括属性（数据）和操作。例如，对打印报告建模的边界类包括所有可能包含在报告单的不同数据项和打印报告所需执行的不同操作。

通常，控制类与边界类一样都容易提取。一般来说，每一个重要的计算被建模成一个控制类。

现在通过提取 MSG 基金会案例研究中的类来说明实体、边界和控制类的提取。图 10-42 的用例图，在这里被复制到图 11-12。

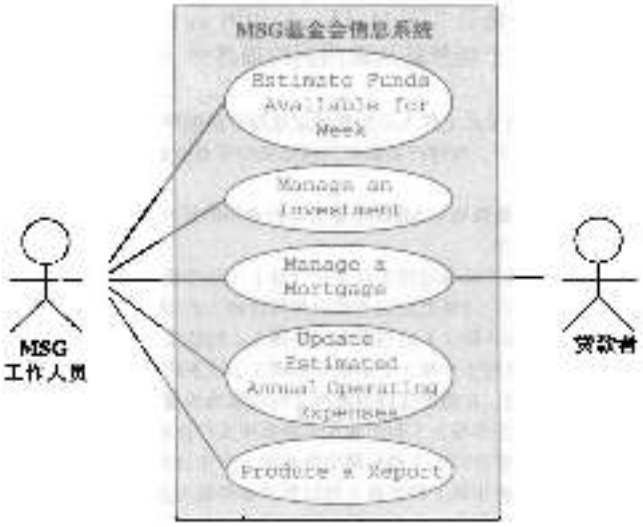


图 11-12　MSG 基金会案例研究用例图的第 7 次迭代

11.12　初始功能建模：MSG 基金会案例研究

如 11.2 节所述，功能建模目的在于找出用例的场景。场景是用例的一个实例。考虑用例 Manage a Mortgage（图 10-32 和图 10-33）。一个可能的场景如图 11-13 所示。一套 MSG 基金会提供抵押的房子每年应付的房产税发生了变化。因为贷款者每周需要支付一定的金钱来交这个税，房产税的任何变化都应该输入到相关的抵押记录里，以相应调整贷款者每周应付金额（或补助金）。扩展场景中的正常部分对 MSG 工作人员访问相关的抵押记录并修改年房产税建模。然而，有时候工作人员因为输入了错误的抵押编号可能不能正确定位那些存储在软件产品里的抵押记录。场景的异常部分对这个可能情况建模。

一个 MSG 基金会工作人员想要更新一套基金会已提供抵押的房子的年房产税
1. 工作人员输入年房产税的新值
2. 软件产品将日期更新为最近一次修改年房产税的日期
可能的选项：
A. 工作人员输入了错误的抵押编号

图 11-13　管理一项抵押的一个扩展场景

对应 Manage a Mortgage 用例的第 2 个场景（图 10-32 和图 10-33）如图 11-14 所示。这里贷款者的周收入发生变化。他们希望这个信息能反映在 MSG 基金会的记录中，以使他们每周应付金额能够被正确计算。扩展场景的正常部分展示了按期望进行的操作。这个场景的异常部分说明了两种可能性。第一，正如前一个场景，工作人员可能输入了错误的抵押编号。第二，贷款者可能没有带来足够的文件证明他们的新收入，在这两种情况下，请求的改变不予执行。

一对向 MSG 基金会借款的夫妇的周收入发生了变化。他们希望工作人员更新他们在基金会记录里的周收入，以使他们的抵押支付能够正确计算
1. 工作人员输入周收入的新值
2. 软件产品将日期更新为最近一次修改周收入的日期
可能的选项：
A. 工作人员输入了错误的抵押编号
B. 贷款者没有带来证明新收入的有关文件

图 11-14　管理一项抵押的另一个扩展场景

第 3 个场景（图 11-15）是用例 Estimate Funds Available for Week（图 10-42）的一个实例。这个场景是从该用例的描述中（图 10-43）直接得到的。

一个 MSG 基金会工作人员想要确定这周可供抵押的基金
1. 对每项投资，软件产品提取出该投资的年收益的估计值。它将各个收入相加并将结果除以 52，产生每周的估计投资收入
2. 软件产品提取每年 MSG 基金会运行费用的估计值，并将结果除以 52
3. 对每项抵押：
   3.1　软件产品通过将本息支付金加上年房产税与屋主每年应付保险费总和的 1/52，计算出这周应付金额
   3.2　然后它计算夫妇二人当前每周净收入的 28%
   3.3　如果步骤 3.1 的结果大于步骤 3.2 的结果，那么将这周的抵押支付金额确定为步骤 3.2 的结果，并且补助金额就等于步骤 3.1 结果与步骤 3.2 结果的差值
   3.4　否则，它确定这周的抵押支付金额为步骤 3.1 的结果，并且这周没有补助金
4. 软件产品将步骤 3.3 和步骤 3.4 的抵押支付金额相加，产生这周总的抵押支付金的估计值
5. 软件产品将步骤 3.3 的补助金额相加，产生这周总的补助金的估计值
6. 软件产品将步骤 1 和步骤 4 的结果相加并减去步骤 2 和步骤 5 的结果，这是当前周总的可供抵押的基金
7. 最后，软件产品打印出当前周可供新抵押的总金额

图 11-15　Estimate Funds Available for Week 用例的一个场景

图 11-16 和图 11-17 的场景是用例 Produce a Report 的实例。同样，这些场景是从相应的用例描述（图 10-39）直接得到的。剩下的场景都比较简单，留作练习（习题 11.12 和习题 11.13）。

一个 MSG 工作人员想要打印所有抵押的列表 1. 工作人员请求打印列有所有抵押的报告

图 11-16　Produce a Report 用例的一个场景

一个 MSG 工作人员想要打印所有投资的列表 1. 工作人员请求打印列有所有投资的报告

图 11-17　Produce a Report 用例的另一个场景

11.13　初始类图：MSG 基金会案例研究

第 2 步是类的建模。这一步的目标是提取实体类，确定它们之间的关联并找出它们的属性。开始这一步最好的方法通常是使用两阶段名词提取法（11.8.1 节）。

阶段 1　使用单个段落来描述软件产品。在 MSG 基金会案例研究中，可能为：

打印出每周报告，显示有多少钱可提供抵押。除此之外，投资和抵押的列表必须按要求打印出来。

阶段 2　从上述描述中识别名词。为简单起见，名词用黑体印刷。

打印出每周报告，显示有多少钱可提供抵押。除此之外，投资和抵押的列表必须按要求打印出来。

名词有报告、钱、抵押、列表和投资。名词报告和列表不是持久存在的，所以它们不可能是实体类（当然报告将证明是边界类），钱是抽象名词。这样剩下两个候选实体类，即 **Mortgage Class**（抵押类）和 **Investment Class**（投资类），图 11-18 所示的是初始类图的第 1 次迭代。

现在考虑这两个实体类之间的关系。从用例 Manage an Investment 和 Manage a mortgage 的描述（分别对应图 10-31 和图 10-33）来看，这两个实体类执行的操作可能是极其相似的，即插入、删除和修改。另外，一个报告产生过程用例描述的第二次迭代（图 10-39）表明了这两个实体类的所有成员必须按要求打印出来。换句话说，**Mortgage Class** 和 **Investment Class** 可能是某个超类的子类。我们把那个超类叫做 **Asset Class**（资产类），因为抵押和投资都是 MSG 基金会的资产。结果初始类图的第 2 次迭代如图 11-19 所示。

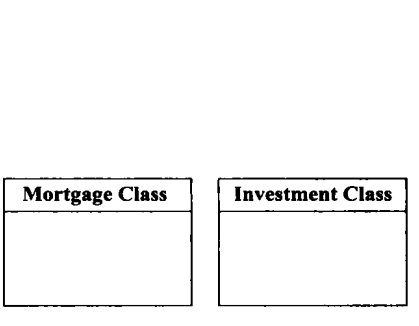


图 11-18　MSG 基金会案例研究初始类图的第 1 次迭代

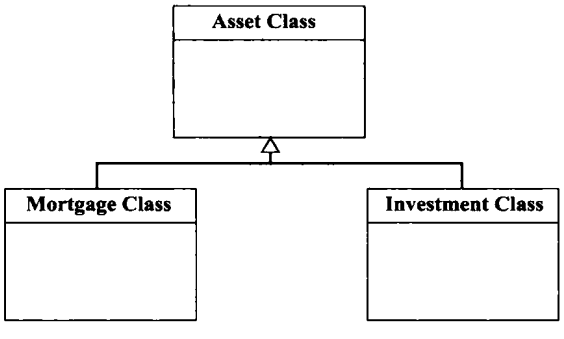


图 11-19　MSG 基金会案例研究初始类图的第 2 次迭代

构造这个超类带来的一个便利是可以再次减少用例的个数。如图 11-12 所示，目前有五个用例，包括 Manage a Mortgage 和 Manage an Investment。然而，如果认为一项抵押或投资是一项资产的特殊情况，则可以把这两个用例合并成一个用例 Manage an Asset。用例图的第 8 次迭代如图 11-20 所示，新的用例用阴影表示。图 11-21 是加入属性后的用例图。

短语“迭代和增量”亦包含了需要对到此刻为止已经开发的产品作减量（decrementation）的可能性。做这种减量有两个原因。首先，如果一个错误已经造成，纠正它的最好方法是恢复

(backtrack) 到该软件产品的较早版本，并找出一种更好的方式去实现曾被错误执行的步骤。在恢复过程中，错误步骤中被添加的所有东西现在都应该移除。其次，重新组织到目前为止开发的模型，结果可能发现一个或更多工件是多余的。为降低开发一个软件产品的难度，尽快移除多余的用例或其他部分是很重要的。

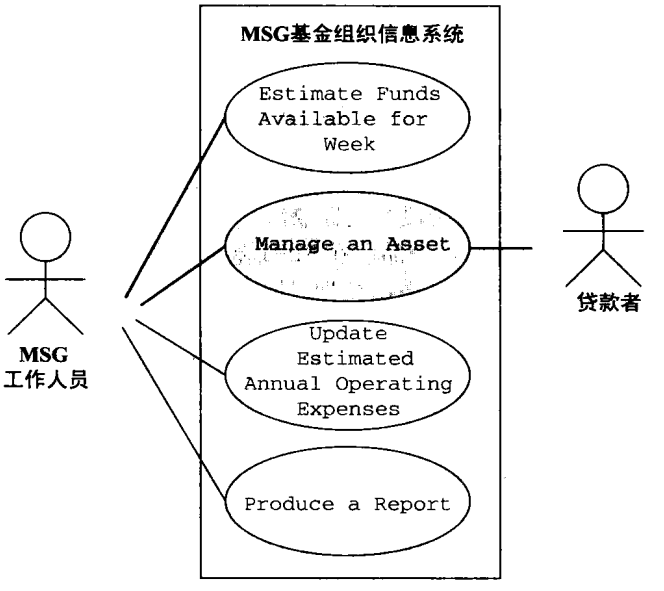


图 11-20　MSG 基金会案例研究用例图的第 8 次迭代。
新用例 Manage an Asset 用阴影表示

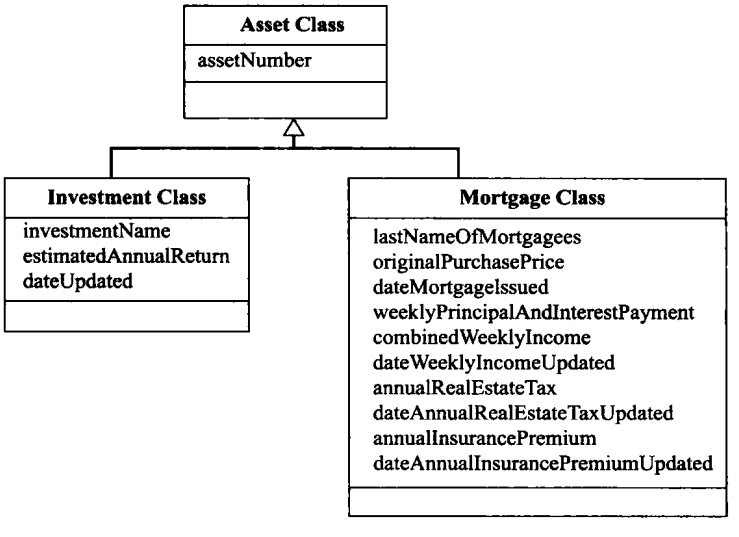


图 11-21　MSG 基金会案例研究初始类图
的第 2 次迭代新增的属性

11.14　初始动态建模：MSG 基金会案例研究

面向对象分析的第 3 步是动态建模。在这一步，状态图被创建以反映该系统或对该系统执

行的操作，并说明导致状态发生转换的事件，相关操作的信息主要来源于场景。

图 11-22 的状态图反映了整个 MSG 基金会案例研究的操作。左上角的实心圆表示初始状态，即状态图的出发点。从初始状态出发的箭头指向标志为 **MSG Foundation Event Loop** 的状态，所有状态除了初始和终止状态用圆角矩形表示。在状态 **MSG Foundation Event Loop** 中，5 个事件中有一个可能发生，详细地说，一个 MSG 工作人员可执行 5 个命令中的一个：估算一周基金、管理一项资产、更新年估计运行费用、产生一个报告或者退出。以下选择包含在这 5 个事件中："选择估算一周基金"、"选择管理一项资产"、"选择更新年估计运行费用"、"选择产生一个报告"和"选择退出"。（事件导致状态间的转换（transition））

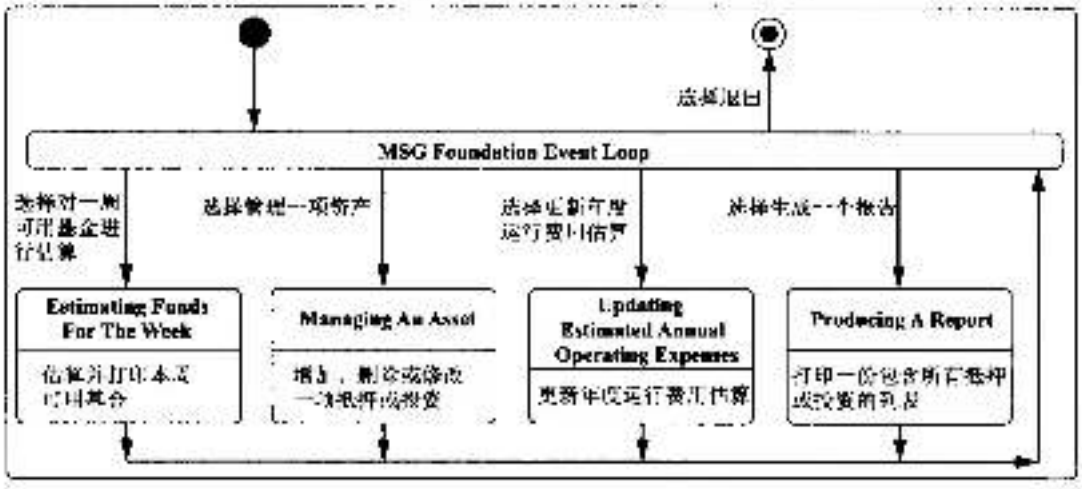


图 11-22 MSG 基金会案例研究初始状态图

当系统处于状态 **MSG Foundation Event Loop** 时，MSG 工作人员在菜单里选择的选项决定了 5 个事件中的任一个会发生。如图 11-23 所示，该菜单将包含于目标软件产品之中（在附录 G 和附录 H 给出的 MSG 基金会案例研究的 C++ 和 Java 实现程序使用的是文本界面而不是图形用户界面（GUI）。也就是说，不同如图 11-23 所示的在一个方框上点击菜单，而是用户输入一个选项，如图 11-24 所示。例如，用户输入 1 对应"估算一周可用基金"，输入 2 对应"管理一项资产"，等等。附录 G 和附录 H 中的实现程序使用图 11-24 所示的文本界面的原因是，文本界面能够在所有计算机上运行，而 GUI 一般需要特殊的软件才能运行）。



图 11-23 目标 MSG 基金会案例研究的菜单



图 11-24 图 11-23 菜单的文本版本

假设 MSG 工作人员单击图 11-23 所示菜单里的选项"管理一项资产"，事件"选择管理一项资产"（图 11-22 所示的 **MSG Foundation Event Loop** 方框下面左数第二个）即将发生。系统

从当前状态 **MSG Foundation Event Loop** 进入状态 **Managing An Asset**。在这个状态下工作人员可以执行的操作（即增加、删除或修改一项抵押或投资），显示在圆角方框的正下方。

一旦操作完成，如箭头所示，系统将返回到状态 **MSG Foundation Event Loop**。状态图其余部分的行为也是一样的简单。

总之，软件产品在状态间转换。在每个状态下，MSG 工作人员可以执行该状态支持的操作，这些操作被列在代表该状态的圆角方框的正下方。状态转换一直持续下去，直到当软件处于状态 **MSG Foundation Event Loop** 时工作人员点击菜单项“退出”。这个时候软件产品进入终止状态（用包含黑色圆心的白色圆圈表示）。一旦软件产品进入这个状态，状态图的执行终止。正如前面所述：状态图是目标软件产品的一个执行模型。

11.15　修订实体类：MSG 基金会案例研究

初始功能模型、初始类图和初始动态模型现在已经构建完毕。然而，对这 3 个模型进行检查将会发现某些东西被忽略了。

看一下图 11-22 所示的初始状态图，考虑状态 **Updating Estimated Annual Operating Expenses** 和操作“更新年估计运行费用”。这项操作必须在数据上执行，即年估计运行费用的当前值。但是年估计运行费用的值应该从哪里获取？看图 11-21，将这个值作为 **Asset Class** 或其子类的属性可能会是个严重的错误。另一方面，目前只有一个类（**Asset Class**）和它的两个子类。这意味着要长久存储某一个值的唯一方法是将它作为该类或其子类实例的属性。

解决方法是：需要另一个实体类来存储年估计运行费用的值。事实上，还有其他的值也需要存储，结果如图 11-25 所示。这里引进一个新类 **MSG Application Class** 来存储图顶端的方框里显示的不同静态属性。除此之外，**MSG Application Class** 将负有启动软件产品其余部分执行的任务。

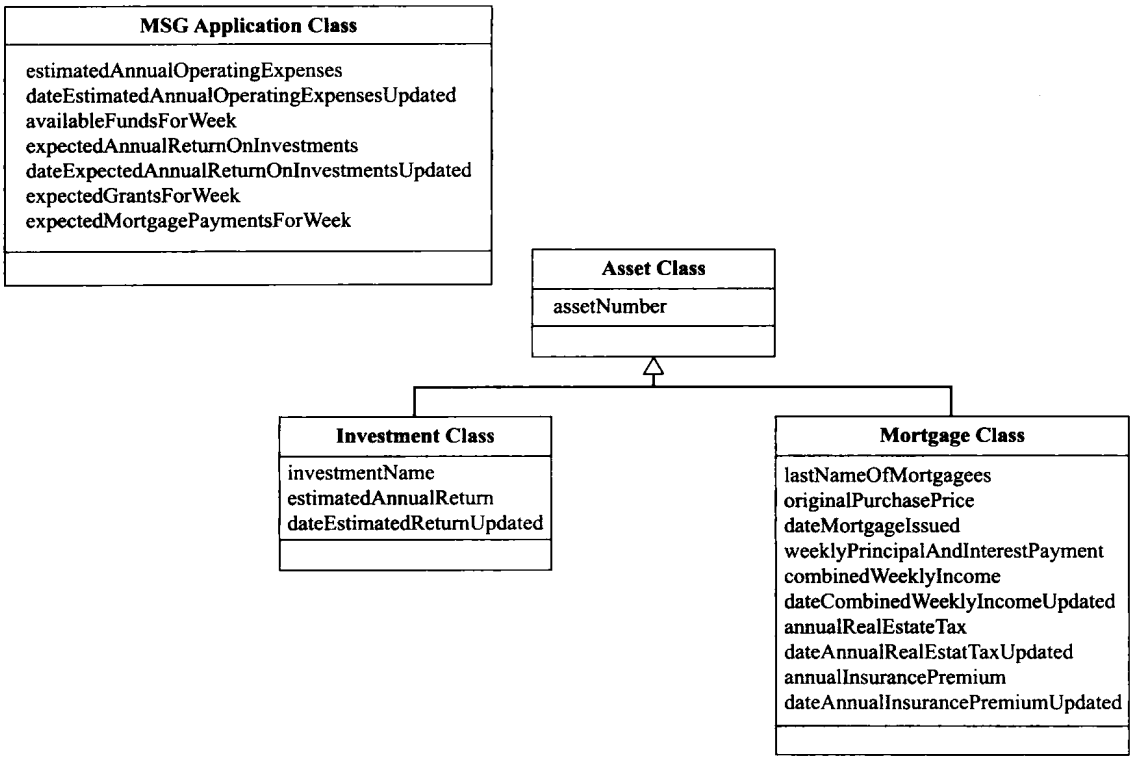


图 11-25　MSG 基金会案例研究初始类图的第 3 次迭代

现在图 11-25 的类图被重新绘制以反映构造型，如图 11-26 所示，图中 4 个类都是实体类。至少到目前为止，这些实体类看起来是正确的。下一步是确定边界类和控制类。

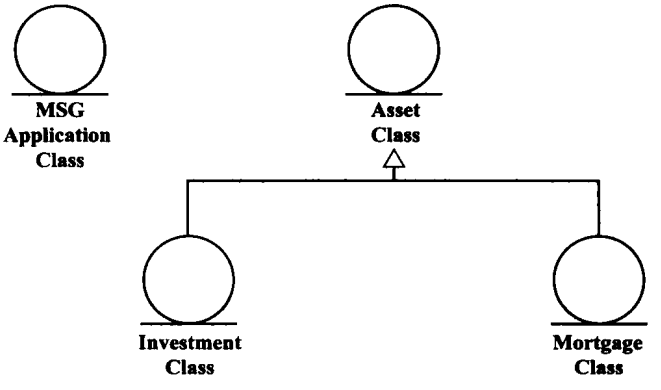


图 11-26　图 11-25 被重新绘制以反映其构造型

11.16　提取边界类：MSG 基金会案例研究

提取实体类通常要比提取边界类困难得多。毕竟，通常实体类之间有相互联系，而正如 11.11 节所指出的，每个输入屏幕、输出屏幕和打印报告常被建模成一个（独立的）边界类。

由于目标 MSG 基金会软件产品看起来相对比较简单（至少在这个统一过程的早期阶段），试图采用只构建一个屏幕使 MSG 工作人员可以用之于四个用例（Estimate Funds Available for Week、Manage an Asset、Update Estimated Annual Operating Expenses 和 Produce a Report）的做法是合理的。随着对 MSG 基金会了解越多，这一个屏幕当然有可能被细化为两个或更多个屏幕。但是初始类提取中只有一个屏幕类 **User Interface Class**。

有 3 个报告需要打印：一周基金估计报告和两个资产报告，即所有抵押或所有投资的完整列表。其中每一个报告都应该建模成一个单独的边界类，因为每个报告的内容是不同的。4 个对应的初始边界类为 **User Interface Class**、**Estimated Funds Report Class**、**Mortgages Report Class** 和 **Investments Report Class**。这 4 个类显示在图 11-27 中。

User Interface Class
Estimated Funds Report Class
Mortgages Report Class
Investments Report Class

图 11-27　MSG 基金会案例研究的初始边界类

11.17　提取控制类：MSG 基金会案例研究

正如在 11.11 节所提到的，通常控制类与边界类一样容易被提取，因为每一个重要计算几乎总被建模成一个控制类。对于 MSG 基金会案例研究，只有一个计算，即估算一周可用基金。这样便产生了初始控制类 **Estimate Funds for Week Class**，如图11-28所示。

接下来一步是检查 3 组类：实体类、边界类和控制类。仔细检查这些类，没有发现明显的矛盾。

从 11.5 节开始的有关类提取内容的介绍到此结束。现在回到统一过程。

Estimate Funds for Week Class

图 11-28　MSG 基金会案例研究初始控制类

11.18　用例实现：MSG 基金会案例研究

用例是参与者与软件产品之间的一次交互的描述。首次使用用例是在软件生命周期的开始阶段，即需求工作流。在分析工作流和设计工作流中，更多的细节被添加到每个用例中，包括

对那些涉及实现用例的相关类的描述。这个扩展和细化用例的过程，称为用例实现（use-case realization）。最后，在实现流中编码实现用例。

这个术语有些令人困惑，因为动词 realize 至少有三个不同的含义：

- 认识（understand）。“Harvey 开始慢慢认识（realize）到他在错误的教室里”。
- 获得（receive）。“Ingrid 将在股票交易中获得（realize）45 000 美元的利润”。
- 实现（accomplish）。“Janet 希望实现（realize）成立一家软件开发公司的梦想”。

在短语“realize a use case”中，单词“realize”为最后一个含义，即它表示实现该用例。

交互图（interaction diagram）、顺序图（sequence diagram）或通信图（communication diagram）描述用例的某个特定场景的实现。

下面我们先分析用例：估算一周可用基金。

11.18.1 Estimate Funds Available for Week 用例

图 11-20 的用例图给出了所有用例。这些用例包括估算一周可用基金，它单独显示在图 11-29 中。该用例的描述在图 10-43 中给出，为了使用方便，这里将图 10-43 复制到图 11-30。从该描述中可以推断出，如图 11-31 中类图所反映的，进入这个用例的类是 **User Interface Class**，且该类对用户界面建模；**Estimate Funds for Week Class**，这个控制类对计算该周可用于资助抵押基金的估计值建模；**Mortgage Class** 对该周估计补助金和支付金建模；**Investment Class** 对每周估计投资收益建模；**MSG Application Class** 对每周估计运行费用建模；**Estimated Funds Report Class** 对打印报告建模。

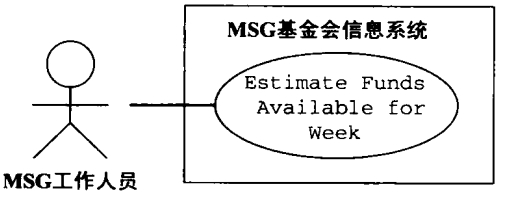


图 11-29　Estimate Funds Available for Week 用例

简要描述

Estimate Funds Available for Week 用例使一个 MSG 基金会工作人员能够估算该周有多少钱可用于资助抵押

按步骤描述

1. 对每项投资，提取出该投资的年收益的估计值。它将各个收入相加并将结果除以 52，产生每周的估计投资收入
2. 通过提取每年 MSG 基金会运行费用的估计值并将结果除以 52，确定每周估计 MSG 基金会运行费用
3. 对每项抵押：
   - 3.1　这周应付金额为将本息支付金加上年房产税与屋主每年应付保险费总和的 1/52 所得的值
   - 3.2　计算夫妇二人当前每周净收入的 28%
   - 3.3　如果步骤 3.1 的结果大于步骤 3.2 的结果，那么将这周的抵押支付金额确定为步骤 3.2 的结果，并且补助金额就等于步骤 3.1 结果与步骤 3.2 结果的差值
   - 3.4　否则，它确定这周的抵押支付金额为步骤 3.1 的结果，并且这周没有补助金
4. 将步骤 3.3 和步骤 3.4 的抵押支付金额相加，产生这周总的抵押支付金的估计值
5. 将步骤 3.3 的补助金额相加，产生这周总的补助金的估计值
6. 将步骤 1 和步骤 4 的结果相加并减去步骤 2 和步骤 5 的结果，这是当前周总的可供抵押的基金
7. 打印出当前周可供新抵押的总金额

图 11-30　Estimate Funds Available for Week 用例描述

图 11-31 是一个类图。更确切地说，它显示参与用例实现的类和它们之间的关系。另一方面，软件产品运行使用的是对象而不是类。例如，某项特定的抵押不能用 **Mortgage Class** 表示，而是用一个对象表示，即 **Mortgage Class** 的一个实例，用:**Mortgage Class** 标示。另外，图11-31 类图显示参与用例的类和它们之间的关系，它并没有显示事件发生的顺序。需要更多东西来对

一个特定场景（如图 11-15 的场景）进行建模，这里复制图 11-15 到图 11-32。

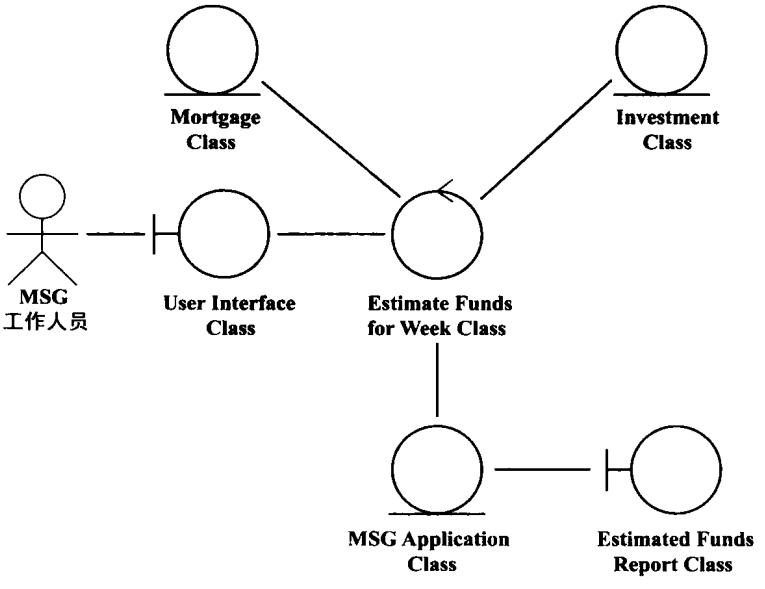


图 11-31　显示了实现 MSG 基金会案例研究 Estimate Funds Available for Week 用例的类的类图

一个 MSG 基金会工作人员想要确定这周可供抵押的基金

1. 对每项投资，软件产品提取出该投资的年收益的估计值。它将各个收入相加并将结果除以 52，产生每周估计投资收入
2. 软件产品提取每年 MSG 基金会运行费用的估计值并将结果除以 52
3. 对每项抵押：
   - 3.1　软件产品通过将本息支付金加上年房产税与屋主每年应付保险费总和的 1/52，计算出这周应付金额
   - 3.2　然后它计算夫妇二人当前每周净收入的 28%
   - 3.3　如果步骤 3.1 的结果大于步骤 3.2 的结果，那么将这周的抵押支付金额确定为步骤 3.2 的结果，并且补助金额就等于步骤 3.1 结果与步骤 3.2 结果的差值
   - 3.4　否则，它确定这周的抵押支付金额为步骤 3.1 的结果，并且这周没有补助金
4. 软件产品将步骤 3.3 和步骤 3.4 的抵押支付金额相加，产生这周总的抵押支付金的估计值
5. 软件产品将步骤 3.3 的补助金额相加，产生这周总的补助金的估计值
6. 软件产品将步骤 1 和步骤 4 的结果相加并减去步骤 2 和步骤 5 的结果，这是当前周总的可供抵押的基金
7. 最后，软件产品打印出当前周可供新抵押的总金额

图 11-32　Estimate Funds Available for Week 用例的一个场景

现在考虑图 11-33，该图是通信图（在 UML 旧版本中为“协作图”）。它显示互相作用的对象和按照顺序发送的消息。通信图描述用例的某个特定场景的实现。在这个例子中，图 11-33 描述了图 11-32 场景的实现。更详细地说，在该场景中工作人员想要计算一周可用基金，这用消息 1“请求估算一周可用基金”表示，该消息从 **MSG Staff Member** 到**:User Interface Class**——**User Interface Class** 的一个实例。

接着请求被传递给**:Estimate Funds for Week Class**——实际执行计算的控制类的一个实例，这用消息 2“转移请求”表示。

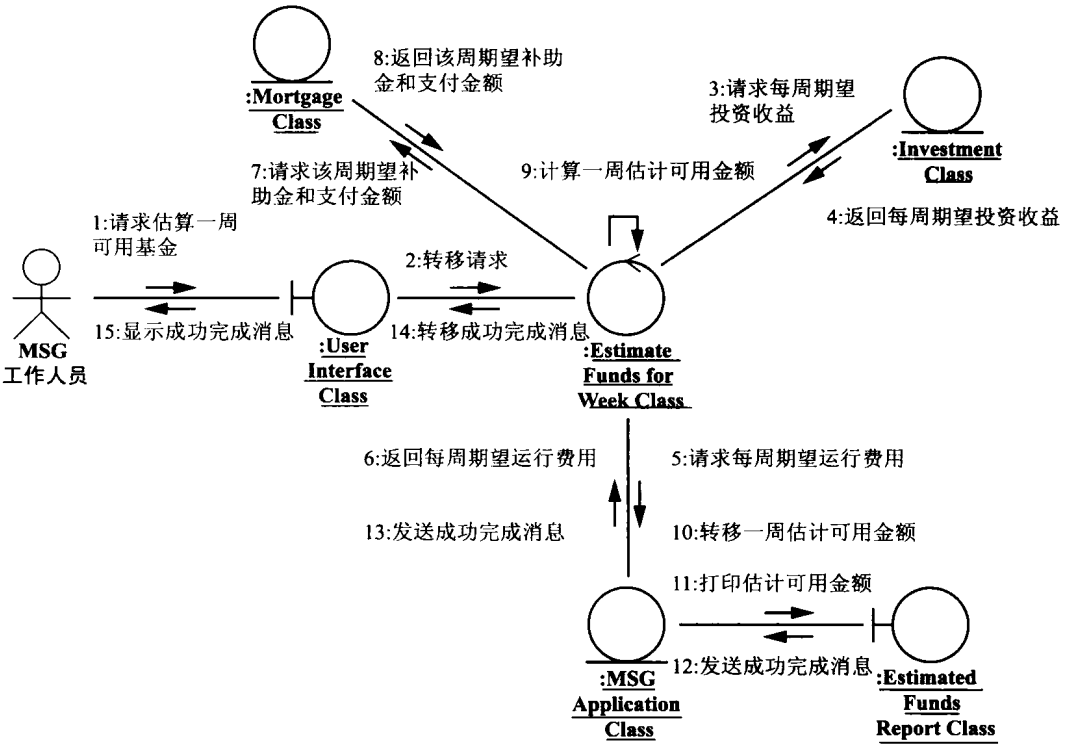


图 11-33　MSG 基金会案例研究 Estimate Funds Available for Week 用例的图 11-32 场景实现的一个通信图

现在，**:Estimate Funds for Week Class** 分别确定四个财务估算值。在场景（图 11-32）的步骤 1 中，年估计投资收益由各项投资收益相加，并将结果除以 52。对每周估计投资收益的提取在图 11-33 中，被建模成从**:Estimate Funds for Week Class** 到**:Investment Class** 的消息 3 “请求每周期望投资收益”，紧接着是相反方向的消息 4 “返回每周期望投资收益”，即返回到控制计算的对象。

在场景（图 11-32）的步骤 2 中，通过将每年估计运行费用除以 52，得到每周运行费用。对每周运行费用的提取在图 11-33 中被建模成从**:Estimate Funds for Week Class** 到**:MSG Application Class** 的消息 5 “请求每周期望运行费用”，紧接着是相反方向的消息 6 “返回每周期望运行费用”。

在场景（图 11-32）的步骤 3、步骤 4 和步骤 5 中，确定两个估算值，即该周估计补助金和该周估计支付金额。这被建模成从**:Estimate Funds for Week Class** 到**:Mortgage Class** 的消息 7 “请求该周期望补助金和支付金额” 和相反方向的消息 8 “返回该周期望补助金和支付金额”。

现在执行场景中步骤 6 的算术计算。这在图 11-33 中被建模成消息 9 “计算一周估计可用金额”，这是一个自调用，即**:Estimate Funds for Week Class** 让自己执行该计算。通过消息 10 “转移一周估计可用金额”，计算结果存储在**:MSG Application Class** 中。

接着，场景（图 11-32）的步骤 7 打印出结果。这在图 11-33 中被建模成从**:MSG Application Class** 到**:Estimated Funds Report Class** 的消息 11 “打印估计可用金额”。

最后，一个应答消息被发送给 MSG 工作人员，通知任务已经成功完成。这在图 11-33 中被建模成消息 12 “发送成功完成消息”、消息 13 “发送成功完成消息”、消息 14 “转移成功完成消息” 和消息 15 “显示成功完成消息”。

没有客户会认可规格说明文档，除非他已精确地理解了目标软件产品将会做什么。出于这个原因，通信图的文字描述是必需的。这在图 11-34 中给出，即事件流（flow of events）。最后，该场景实现的等价的顺序图，如图 11-35 所示。在构建一个软件产品时，通信图或顺序图都可能有利于更好地理解用例的实现。在某些情形下，可能同时需要两个图以充分理解给定用例的某个特定实现。这就是本章每个通信图后面都跟着等价的顺序图的缘故。图 11-35 的顺序图完全等价于图 11-33 的通信图，因此它的事件流也在图 11-34 中给出。

> 一个 MSG 工作人员请求估算一周可用抵押基金（1，2）。软件产品估算每周投资收益（3，4）、每周运行费用（5，6）和该周补助金和支付金额（7，8）。然后它估算（9），存储（10），并打印出（11-15）一周可用基金。

图 11-34　MSG 基金会案例研究 Estimate Funds Available for Week 用例的图 11-32 场景实现的图 11-33 通信图的事件流

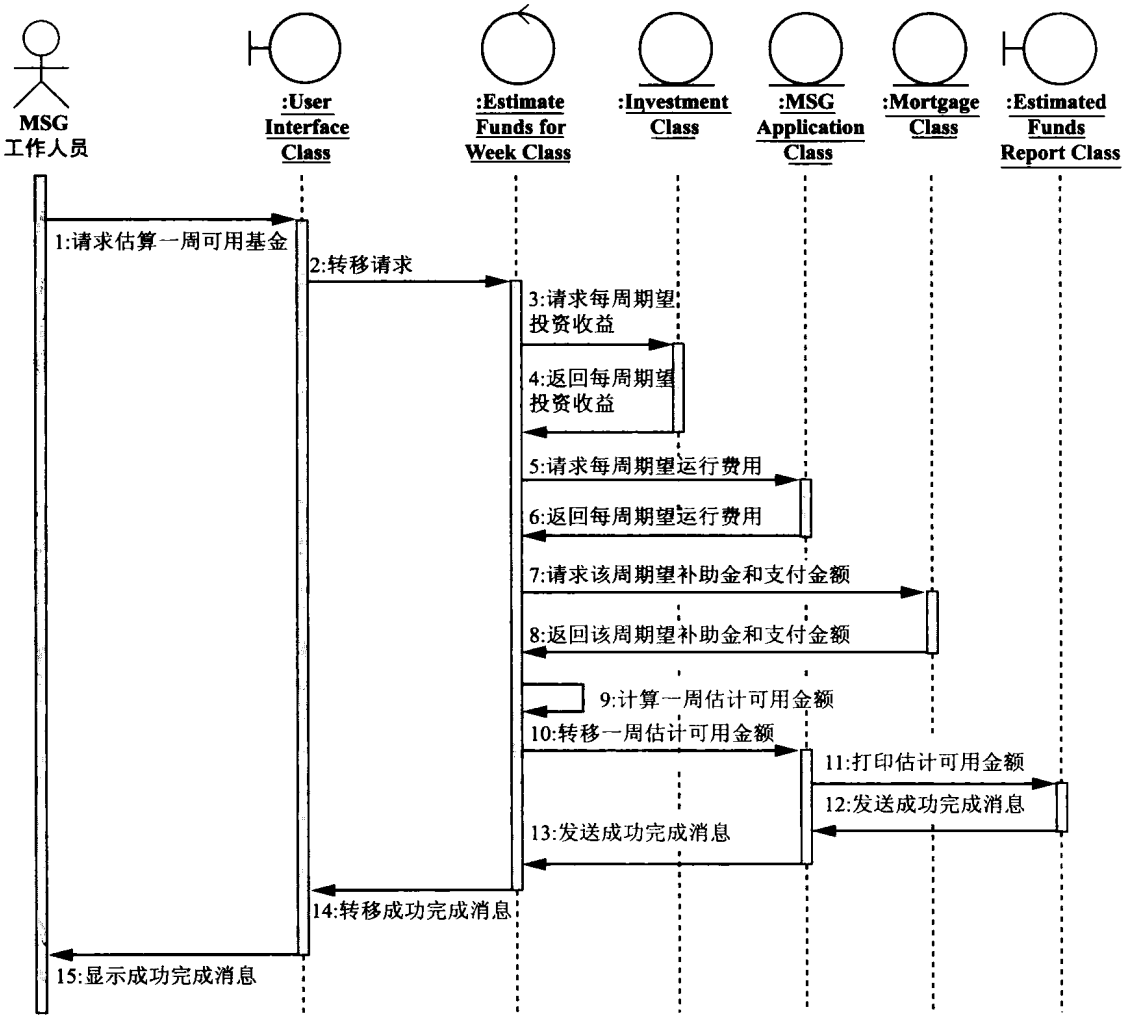


图 11-35　MSG 基金会案例研究 Estimate Funds Available for Week 用例的图 11-32 场景实现的一个顺序图。这个顺序图完全等价于图 11-33 的通信图，因此它的事件流也在图 11-34 中给出

顺序图的优点是它清楚地描述消息流。消息发送的顺序和每条独立消息的发送者和接收者都非常清楚。因此，当信息的转移是关注的重点时（执行分析工作流时很多时候都是这种情况），顺序图就优于通信图。另一方面，类图（如图 11-31）和实现相关场景的通信图（如图 11-33）之间有很多的相似性。因此，在开发人员把注意力集中在类上的情况下，通信图通常比顺序图更有用。

总而言之，图 11-29 到图 11-35 描述的并不是一组随意的 UML 零件。相反，这些图描述的是同一个用例和从该用例推导出的其他模型。更详细地说：

- 图 11-29 描述用例 Estimate Funds Available for Week。也就是说，图 11-29 对参与者 **MSG 工作人员**（一个位于软件产品外部的实体）和 MSG 基金会软件产品内部与估算一周可用基金行为有关的所有可能的交互建模。
- 图 11-30 是该用例的描述。也就是说，它提供了图 11-29 的 Estimate Funds Available for Week 用例的细节的文字描述。
- 图 11-31 是一个类图，它显示实现 Estimate Funds Available for Week 用例的类。这个类图描述了对该用例所有可能场景建模所需的类和它们之间的关系。
- 图 11-32 是一个场景，即图 11-29 用例的一个特定实例。
- 图 11-33 是图 11-32 场景实现的一个通信图。也就是说，它描述了那个特定场景实现中的对象和它们之间发送的消息。
- 图 11-34 是图 11-32 场景实现的通信图的事件流。也就是说，正如图 11-30 是图 11-29 的 Estimate Funds Available for Week 用例的文字描述，图 11-34 是图 11-32 场景实现的文字描述。
- 图 11-35 是完全等价于图 11-33 通信图的顺序图。也就是说，这个顺序图描述了图 11-32 所示场景实现中的对象以及它们之间发送的消息。所以它的事件流也在图 11-34 中给出。

本书已经多次提到过，统一过程是用例驱动的。这些标记（着重号）黑点的项清晰地描述了图 11-30 到图 11-35 的每个模型与作为它们基础的图 11-29 用例之间的准确联系。

11.18.2 Manage an Asset 用例

Manage an Asset 用例如图 11-36 所示，图 11-37 是它的描述。图 11-38 的类图显示实现 Manage an Asset 用例的类。起先我们认为只需要一个控制类（参见图 11-28）。然而，图 11-38 表明第二个控制类，即 **Manage an Asset Class** 也是需要的。在接下来的迭代，可能会添加更多的控制类。



图 11-36　Manage an Asset 用例

简要描述

Manage an Asset 用例使 MSG 基金会工作人员能够增加和删除资产并管理资产（投资和抵押）。管理一项抵押包括更新一对向基金会借款的夫妇的周收入

按步骤描述

1. 增加、修改或删除一项投资或抵押，或者更新贷款者的周收入

图 11-37　Manage an Asset 用例描述

用例 Manage a Mortgage（同时是 Manage an Asset）的图 11-13 扩展场景的正常部分被复制到图 11-39。在这个场景中，一个 MSG 工作人员更新一套已被抵押的房子的年房产税，并且软件产品将日期更新为最近一次修改年房产税的日期。图 11-40 是该场景实现的通信图。注意到对象 **:Investment Class** 在这个通信图中并不起作用，因为图 11-39 的场景只涉及一项抵押，而不包含一项投资。同样，**Borrowers**（贷款者）在这个场景中也不起作用。事件流留作练习（习题 11.14）。与图 11-40 通信图等价的顺序图如图 11-41 所示。

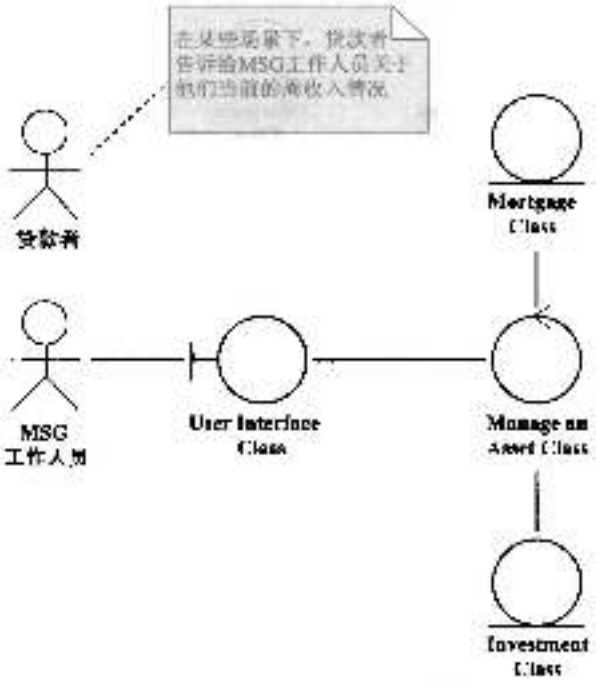


图 11-38　显示实现 MSG 基金会案例研究 Manage an Asset 用例的类的类图

一个 MSG 基金会工作人员想要更新一非基金会会员
拥有抵押的房子的年房产税
1. 工作人员输入专房产税的细节
2. 软件产品将日期更新为最近一次修改年房产税
的日期

图 11-39　Manage an Asset 用例的一个场景

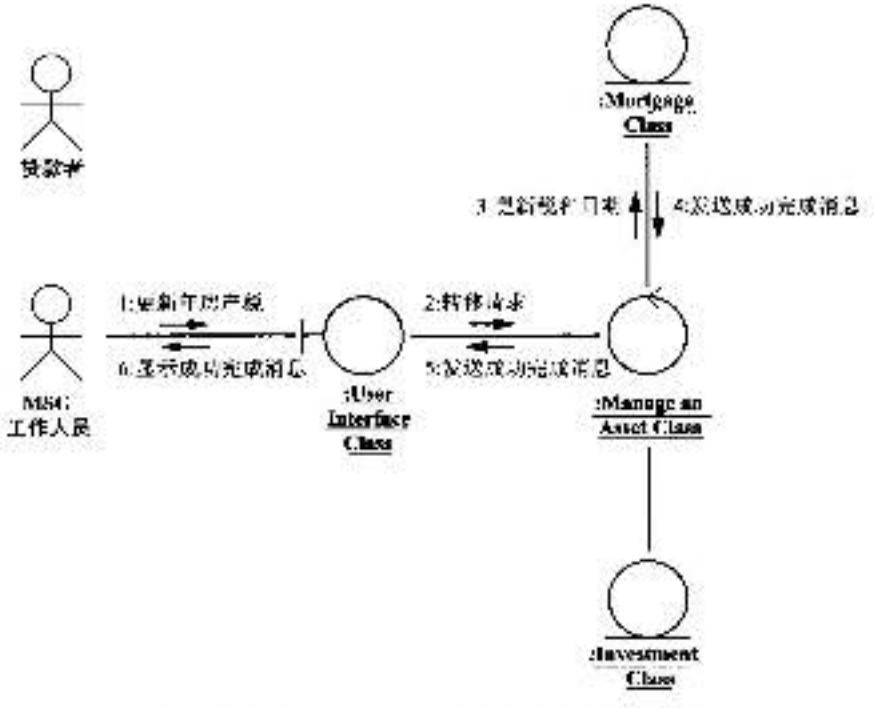


图 11-40　MSG 基金会案例研究 Manage an Asset 用例的图 11-39 场景实现的一个通信图

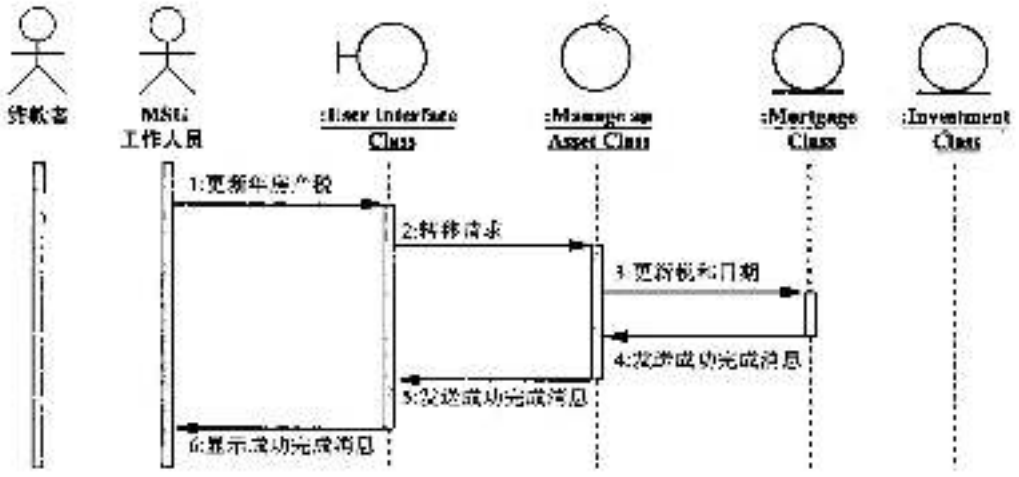


图 11-41　MSG 基金会案例研究 Manage an Asset 用例的图 11-39 场景实现的一个顺序图

现在考虑用例 Manage an Asset（图 11-36）的另一个场景，即图 11-14 的扩展场景。该场景的正常部分在这里被复制到图 11-42。在这个场景中，应贷款者的请求，MSG 工作人员更新一对已在 MSG 办理抵押的夫妇的周收入。如 10.7 节所说明的，该场景由 **Borrowers** 发起，并且他们的数据由 **MSG** 工作人员输入到软件产品中，在图 11-43 通信图的注释中已经给出描述。事件流内次留作练习（习题 11.15）。与通信图等价的顺序图，如图 11-44 所示。

一对向 MSG 基金会贷款的夫妇的周收入发生了变化。他们希望工作人员更新他们在抵押记录里的周收入，以便他们的抵押支付数额能正确计算。

1. 工作人员输入周收入的新值。
2. 软件产品将日期更新为最近一次修改周收入的日期。

图 11-42　Manage an Asset 用例的第二个场景

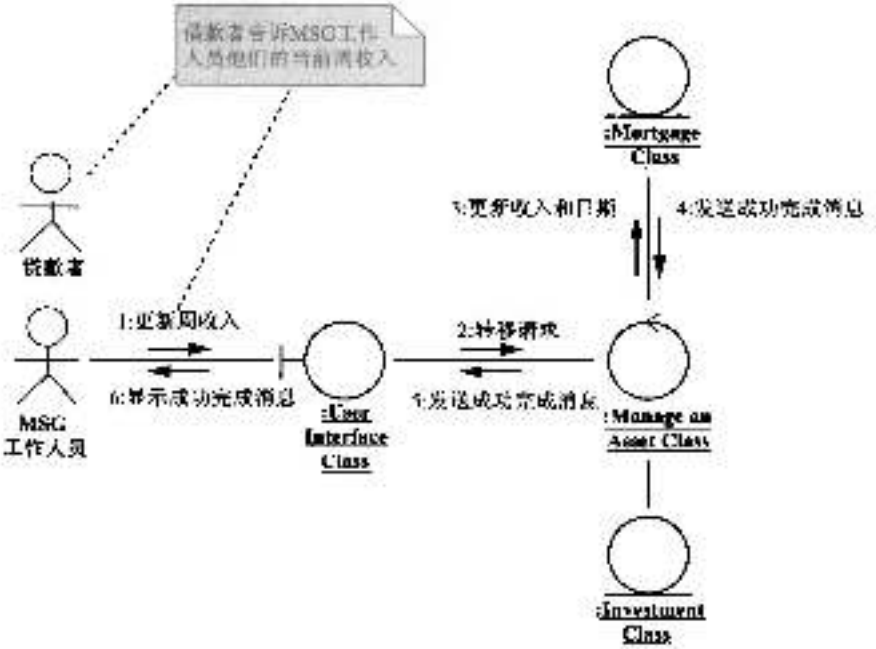


图 11-43　MSG 基金会案例研究 Manage an Asset 用例的图 11-42 场景实现的一个通信图

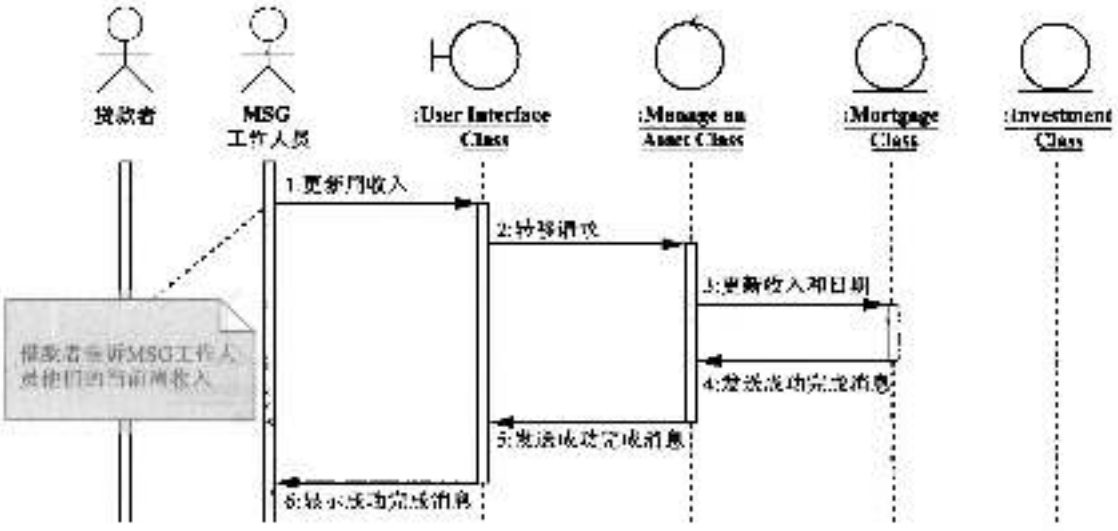


图 11-44　MSG 基金会案例研究 Manage an Asset 用例的图 11-42 场景实现的一个顺序图

比较图 11-40 和图 11-43 的通信图（或者，等价地，图 11-41 和图 11-44 的顺序图），我们发现除了参与者，这两个图唯一不同的是消息 1、消息 2 和消息 3 在图 11-40（或图 11-41）的情况中涉及年房产税，而在图 11-43（或图 11-44）的情况中涉及月收入。这个例子突出了用例、场景（用例实例）和该用例不同场景实现的通信图或顺序图之间的区别。

边界类 **User Interface Class** 出现在目前为止的所有实现里面。实际上，该软件产品的所有命令都将使用相同的屏幕。一个 MSG 工作人员在图 11-45 所示修改过的菜单点击记号的操作选项。（相应的文本界面如图 11-46 所示，该文本界面在附录 G 和附录 H 中实现。）



图 11-45　目标 MSG 基金会案例研究修改过的菜单



图 11-46　图 11-45 修改过的菜单的文本版本

11.18.3　Update Estimated Annual Operating Expenses 用例

用例 Update Estimated Annual Operating Expenses 如图 10-17 所示，图 10-18 是它的描述。图 11-47 给类图显示实现 Update Estimated Annual Operating Expenses 用例的类。图 11-48 是该用例一个场景实现的通信图。等价的顺序图如图 11-49 所示。该场景的细节和事件流留作练习（习题 11.16 和习题 11.17）。

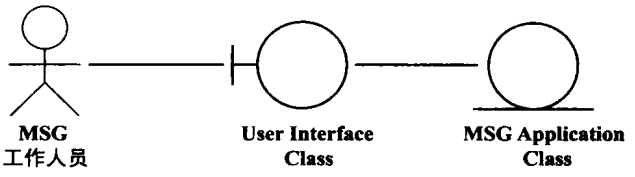


图 11-47　显示实现 MSG 基金会案例研究 Update Estimated Annual Operating Expenses 用例的类的类图

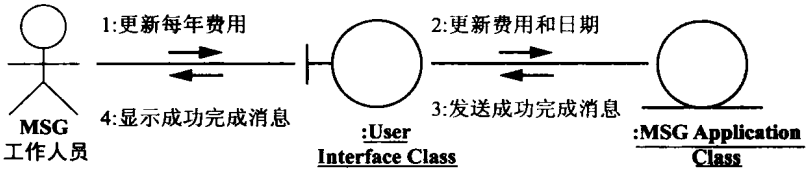


图 11-48　MSG 基金会案例研究 Update Estimated Annual Operating Expenses 用例的一个场景实现的一个通信图

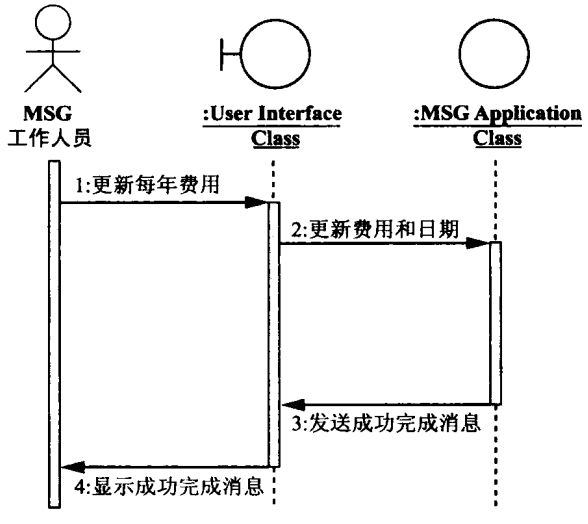


图 11-49　MSG 基金会案例研究 Update Estimated Annual Operating Expenses 用例的一个场景实现的一个顺序图

11.18.4　Produce a Report 用例

用例 Produce a Report 如图 11-50 所示。图 10-39 的 Produce a Report 用例描述在这里被复制到图 11-51。图 11-52 的类图给出实现 Produce a Report 用例的类。

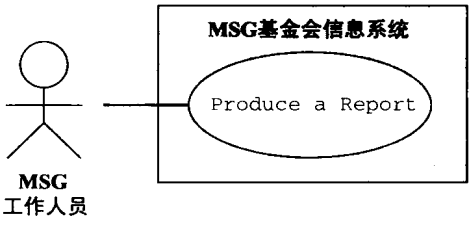


图 11-50　Produce a Report 用例

简要描述

Produce a Report 用例使 MSG 工作人员能够打印一份所有投资或所有抵押的列表。

按步骤描述

1　需要产生下面的报告：

1.1　投资报告——经要求打印：

软件产品打印一份所有投资的列表。对每项投资，打印下面的属性：

项目编号
项目名称
年估计收益
最后一次更新年估计收益的日期

1.2　抵押报告——经要求打印：

软件产品打印一份所有抵押的列表。对每项抵押，打印下面的属性：

抵押编号
抵押者名字
房子原价格
抵押日期
本息支付金额
当前一周联合净收入
最近一次更新当前一周联合净收入的日期
年房产税
最近一次更新年房产税的日期
每年屋主应付保险费
最近一次更新每年屋主应付保险费的日期

图 11-51　Produce a Report 用例描述

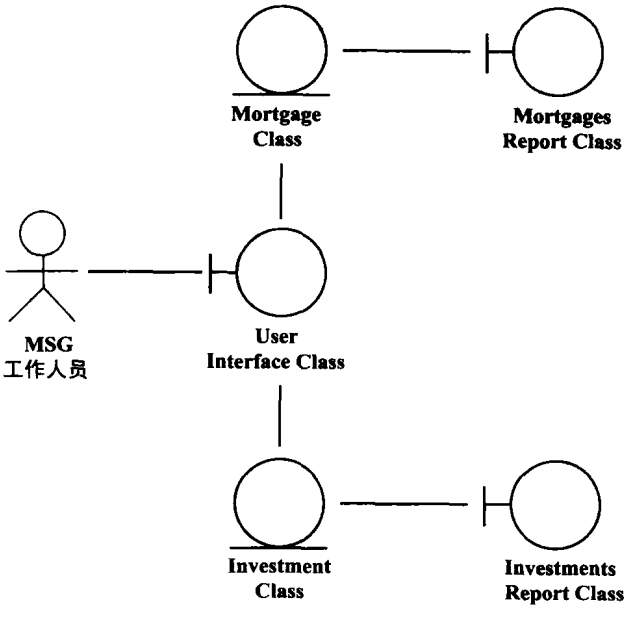


图 11-52　显示实现 MSG 基金会案例研究 Produce a Report 用例的类的类图

首先考虑图 11-16 列出所有抵押的场景，在这里被复制到图 11-53。这个场景实现的一个通信图如图 11-54 所示。该实现对列出所有抵押建模。因此，对象**:Investment Class**，即 **Asset Class** 的另一个子类的实例，在这个场景中不起作用，**:Investments Report Class** 也不起作用。事件流留作练习（习题 11.14）。等价的顺序图如图 11-55 所示。

一个 MSG 工作人员想要打印所有抵押的列表
1. 工作人员请求打印列有所有抵押的报告

图 11-53　Produce a Report 用例的一个场景

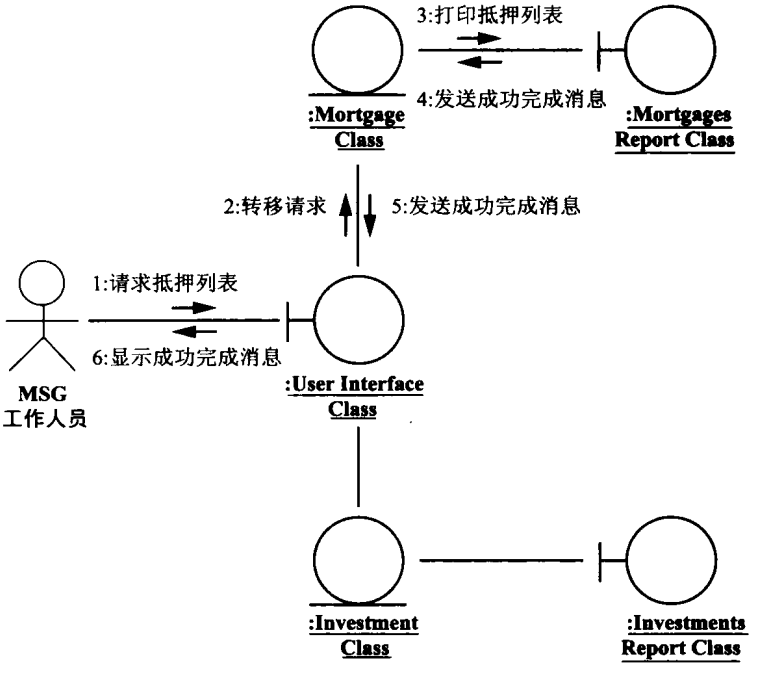


图 11-54　MSG 基金会案例研究 Produce a Report 用例的图 11-53 场景实现的一个通信图

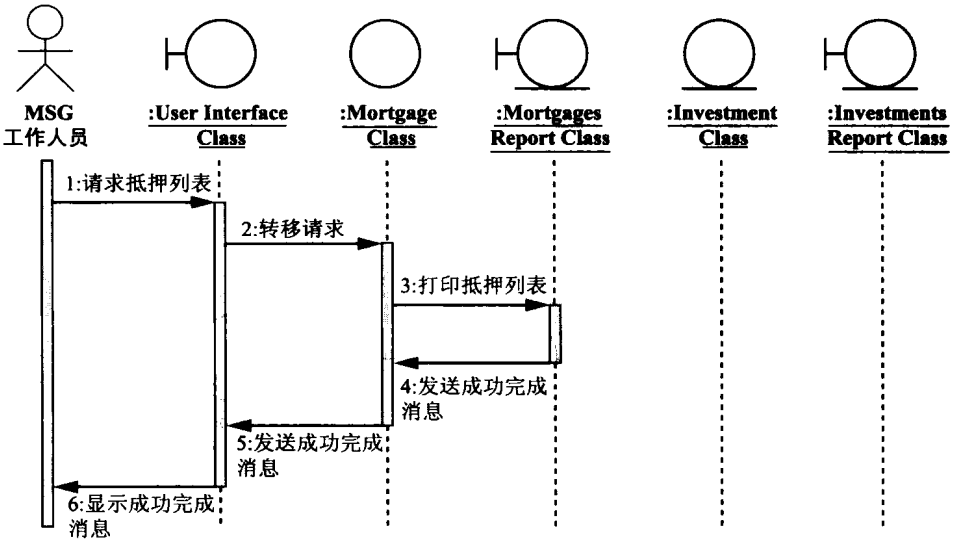


图 11-55　MSG 基金会案例研究 Produce a Report 用例的图 11-53 场景实现的一个顺序图

现在考虑图 11-17 列出所有投资的场景，图 11-17 在这里被复制到图 11-56。这个场景实现的一个通信图如图 11-57 所示。与前一个实现相反，图 11-57 对列出所有投资建模，并且抵押在这里被忽略。等价的顺序图如图 11-58 所示。

图 11-20 的 MSG 基金会案例研究用例图第 8 次迭代中的四个用例的实现到这里结束。

一个 MSG 工作人员想要打印所有投资的列表
1. 工作人员请求打印列有所有投资的报告

图 11-56　Produce a Report 用例的另一个场景

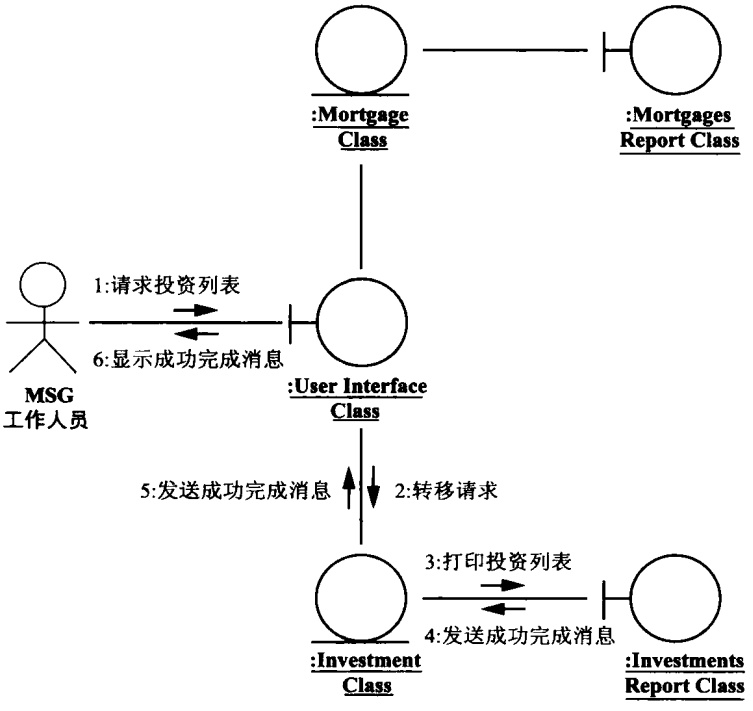


图 11-57　MSG 基金会案例研究 Produce a Report 用例的图 11-56 场景实现的一个通信图

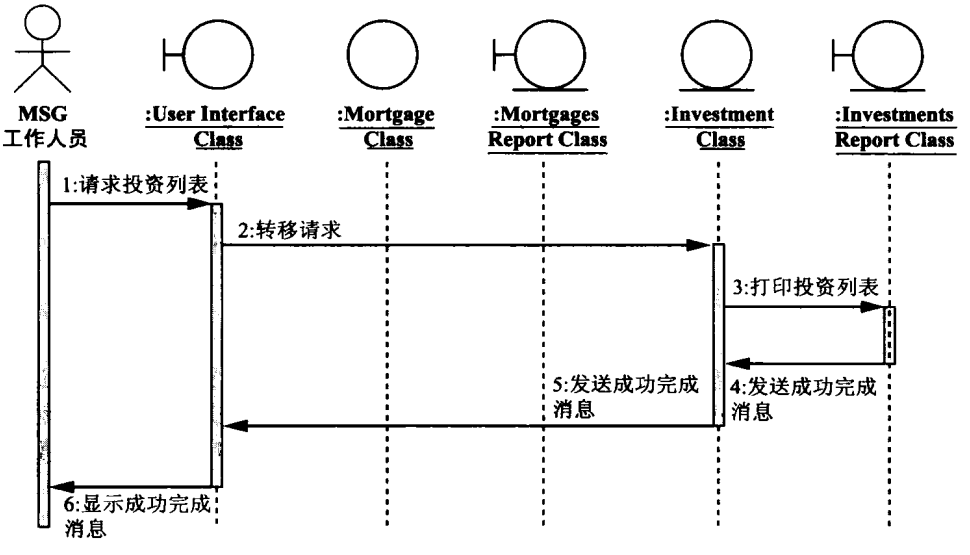


图 11-58　MSG 基金会案例研究 Produce a Report 用例的图 11-56 场景实现的一个顺序图

11.19　类图增量：MSG 基金会案例研究

从 11.12 节到 11.15 节中提取实体类，产生了图 11-26，即显示 4 个实体类的类图。边界类是在 11.16 节中提取的，控制类是在 11.17 节和 11.18.2 节中提取的。在 11.18 节实现不同用例的过程中，很多类之间的相互联系变得更清楚，这些联系反映在图 11-31、图 11-38、图 11-47 和图 11-52 的类图中。图 11-59 将这些类图进行了合并。

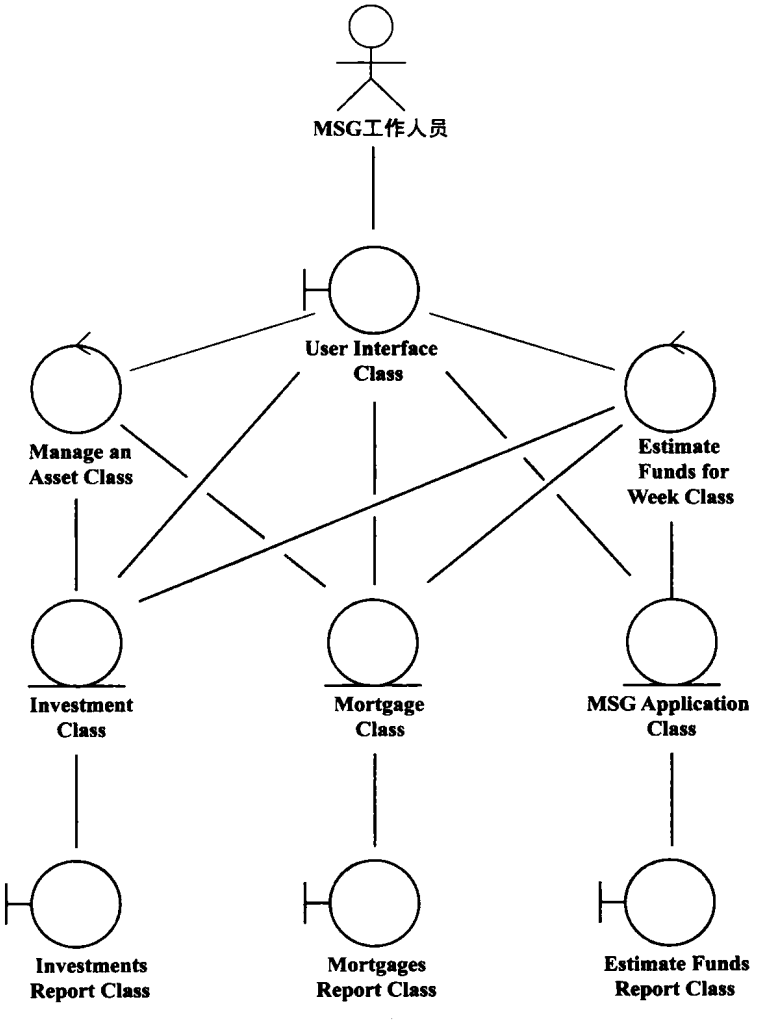


图 11-59　合并图 11-31、图 11-38、图 11-47 和图 11-52 类图的类图

现在合并图 11-26 和图 11-59 的类图，产生 MSG 基金会案例研究类图的第 4 次迭代，如图 11-60 所示。更明确地说，从图 11-59 开始，图 11-26 的 **Asset Class** 被添加进去。然后画出图 11-26的两个继承（泛化）关系，它们是用虚线来区分的。结果如图 11-60，类图的第 4 次迭代，是结束分析工作流时的类图。

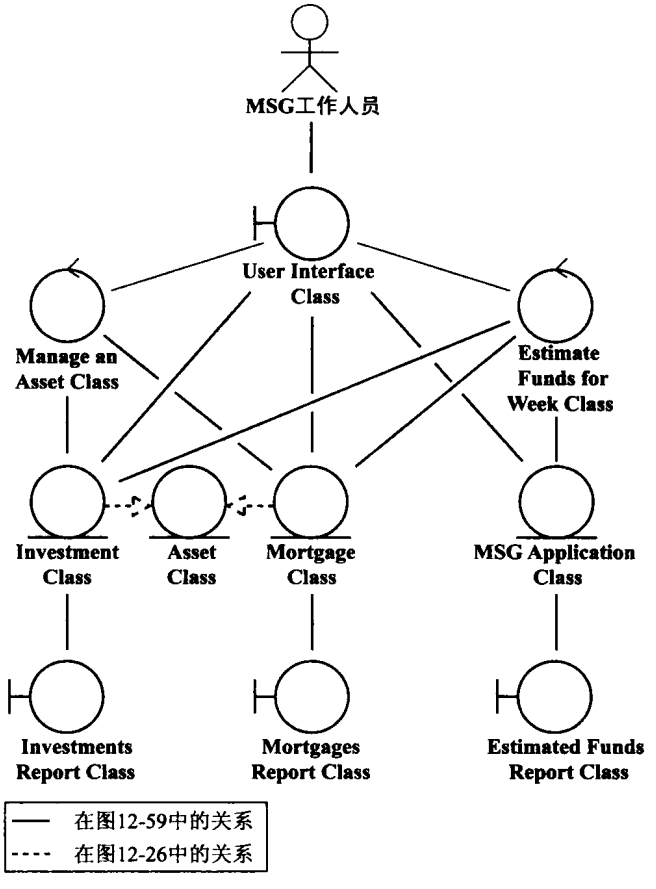


图 11-60　通过合并图 11-26 和图 11-59 类图得到的
MSG 基金会案例研究类图的第 4 次迭代

11.20　软件项目管理计划：MSG 基金会案例研究

MSG 基金会案例研究分析工作流的最后一步是制定软件项目管理计划（这在细化阶段进行，见 3.10.2 节）。附录 E 给出了一个小型（三人）软件组织制定的开发 MSG 基金会产品的软件项目管理计划。这份计划书采用 IEEE SPMP 格式（9.6 节）。

11.21　测试工作流：MSG 基金会案例研究

检查 MSG 基金会案例研究的分析工作流有两种方式。首先，使用 CRC 卡片检查实体类，如 11.10 节所述。然后检查分析工作流中的所有模型（6.2.3 节）。

MSG 基金会案例研究的分析工作流到这里结束。

11.22　统一过程中的规格说明文档

分析工作流的一个主要目标是产生规格说明文档，但是 11.21 节的结尾处却说分析工作流已完成。一个显然的问题是，规格说明文档在哪里？

简短的回答是，统一过程是用例驱动的。更详细地说，用例以及相关模型包含了所有的甚至更多的，在传统范型中以文本形式呈现在规格说明文档中的信息。

例如，考虑用例 Estimate Funds Available for Week。当执行需求工作流时，Estimate Funds Available for Week 用例（图 10-27）和它的描述（图 10-40）显示给客户，即 MSG 基金会的托管人。开发人员必须要非常认真，以保证托管人能充分理解这两个模型，并同意这些模型准确地对基金会所需的软件产品进行了建模。随后在分析工作流中，给托管人出示用例 Estimate Funds Available for Week（图 11-29）、它的描述（图 11-30）、显示实现该用例的类的类图（图 11-31）、该用例的一个场景（图 11-32）、该用例一个场景实现的交互图（图 11-33 和图 11-35）以及这些交互图的事件流（图 11-34）。

列出的这组模型都只与用例 Estimate Funds Available for Week 有关。如图 11-20 所示，总共有 4 个用例。每个用例的每一个场景都产生一组相同类型的模型。结果是以一组模型的形式（有些是图表，有些是文本）传递给客户更多信息，并且这些信息比传统范型的纯文本规格说明文档表达的要更加精确。

通常，传统的规格说明文档扮演合同的角色。也就是说，一旦开发人员和客户之间签署了，它本质上就是一个有法律效力的文件。如果开发人员开发的软件产品满足规格说明文档，客户必须为软件付钱。另一方面，如果产品不符合规格说明文档，可要求开发人员修改它，否则可以不支付报酬。在统一过程里，所有用例的所有场景产生的模型集同样也构成一份合同。因此，如 11.21 节结尾所述，MSG 基金会案例研究的分析工作流确实已完成。

之前提到过，统一过程是用例驱动的。当使用统一过程时，不是构造一个快速原型，而是把用例，更精确地说是反映那些实现用例的场景的类的交互图，呈现给客户。客户可以从交互图和它们的文字描述的事件流（就如同从快速原型中一样）了解目标软件产品将会做什么。场景和快速原型的每次执行一样，都是目标软件产品的一次特殊的执行序列。区别在于，快速原型一般要被舍弃，而用例是逐步精化的，每一步添加更多的信息。

然而，快速原型在一个方面要优于场景，即用户界面（10.14 节）。这并不意味着应该建立一个快速原型，使客户和用户可以检查样本屏幕和报告。但是如 10.14 节所描述，构造样本屏幕和报告是必要；更可取的方法是使用 CASE 工具，例如，屏幕生成器和报告生成器（5.5 节）。

11.23 节中给出确定参与者和用例的方法。

11.23　更多关于参与者和用例的内容

10.4.3 节提到，用例描述软件产品本身和参与者（软件产品的用户）之间的交互。既然前面已经给出了许多关于参与者和用例的例子，现在描述如何发现参与者与用例是合适的。

为了找出参与者，我们必须考虑个体与软件产品交互可能扮演的每个角色（role）。例如，考虑一对想要在 MSG 基金会办理抵押的夫妇，当他们申请抵押时，他们是申请者（Applicants），而当他们的申请被批准，并且他们贷到购买房子的钱时，他们是贷款者（Borrowers）。换言之，参与者不是个体，而是这些个体所扮演的角色。在我们的例子中，参与者不是那对夫妇，而是首先扮演申请者角色的夫妇和接着扮演贷款者角色的夫妇。这意味着只列出所有将使用软件产品的个体并不是一个很好地找出参与者的方法。相反，我们需要找出每个用户（或每组用户）所扮演的所有角色。从角色列表中，我们可以提取出参与者。

在统一过程的专业术语中，术语“工作者”（worker）用来表示个体扮演的一个特殊角色。这是一个不太准确的术语，因为 worker 通常指雇员。在统一过程的专业术语里，对于一对办理抵押的夫妇的例子，申请者和贷款者是两个不同的工作者。为了清晰起见，本书使用“角色”代替“工作者”。

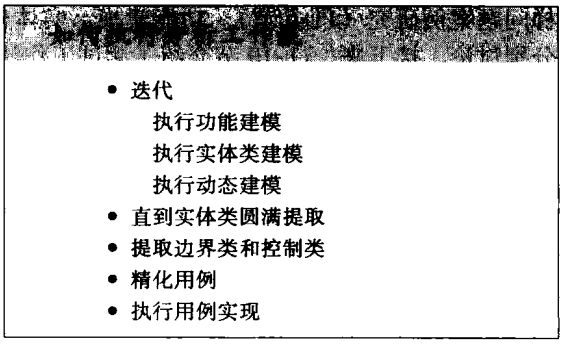
在业务背景中，找出角色的工作通常比较简单。用例业务模型通常描述了与该业务交互的个体所扮演的所有角色，从而突出业务参与者。然后我们找出对应需求用例模型的用例业务模型的子集。更详细地说：

1）通过找出与该业务交互的个体所扮演的所有角色，构造用例业务模型。

2）找出对我们期望开发的软件产品建模的业务模型用例图的子集。也就是，考虑业务模型中只符合目标软件产品的那些部分。

3）子集中的参与者即我们要找的参与者。

一旦参与者被确定，找出用例一般比较简单。对每个角色，有一到多个用例。因此，正如本节前面所述，发现需求用例从找出参与者开始。



“如何执行分析工作流”概括了分析工作流的执行步骤。

11.24　支持分析工作流的 CASE 工具

由于分析工作流中图所起的作用很大，有很多支持分析工作流的 CASE 工具已经被开发出来，这不足为奇。在基本形式上，这一类工具本质上就是绘图工具，使得执行每个建模步骤变得容易。更重要的是，修改一个用绘图工具构造的图要比改变一张手绘的图容易得多。因此，这种类型的 CASE 工具能支持分析工作流的图形方面。除此之外，一些这类型的工具不仅能够绘制相关的图，还包括 CRC 卡片。这些工具的一个优点是，基础模型的任何改变将在所有相关图中自动反映出来。因为不同的图仅是基础模型的不同视图。

另一方面，一些 CASE 工具不仅支持分析工作流，还支持面向对象生命周期其他相当大一部分。如今，实际上所有这些工具都支持 UML［Rumbaugh，Jacobson，and Booch，1999］。这类工具的例子包括 IBM Rational Rose 和 Together。ArgoUML 是其中的一个典型的开放源码 CASE 工具。

11.25　分析工作流的挑战

分析过程的一个挑战在于，分析（什么）和设计（如何）的界线实在太容易混淆了。规格说明文档应描述产品必须做什么，它不应说明产品是如何实现的。例如，假设客户要求不管什么时候执行一次网络路由计算，响应时间不超过 0.05 秒。规格说明文档应准确地说明这个要求，除此之外别无其他。特别的是，规格说明文档不应说明要使用哪个算法去实现响应时间。也就是说，规格说明文档应列出所有约束条件，但它不能说明这些约束条件是如何实现的。

同样地，规格说明文档应描述目标产品的操作。它不应该指明这些操作是如何实现的，当然也不能指明每个操作该分派给哪个类。设计小组的任务是从总体上研究规格说明并决定一种能够最佳地实现这些规格说明的设计方案，这将在第 12 章中描述。

在 OOA 工作流早期阶段就出现类，这意味着在 OOA 阶段工作人员容易被 OOA 的强烈诱惑带着走得太远。例如，考虑给类分配方法的这个问题。在分析工作流中，我们确定了类和它们的交互关系，其结果在类图中描述。因此，看起来没有理由要等到设计流才分配方法给类。

然而，记住分析工作流是一个迭代的过程，这是很重要的。在精化各种模型的过程中，大部分的类图经常需要重组，再分配方法将导致不必要的额外的修改。

在 OOA 过程的每一步，一个好的建议就是最简化那些需要在迭代中重组的信息。因此，在分析工作流即便是有走远一点点的诱惑，我们都应该等到设计流再分配方法给类。

本章回顾

规格说明（11.1 节）可以用自然语言非形式化表达（11.2 节），但这可能会产生问题

(11.3 节)。11.4 节介绍分析工作流。提取实体类的技术在 11.5 节描述。这个技术被应用到电梯问题案例研究 (11.6 节)。功能建模、实体建模和动态建模在 11.7 节、11.8 节和 11.9 节中分别讨论。接下来，11.10 节介绍了测试工作流的分析部分。边界类和控制类的提取在 11.11 节介绍。MSG 基金会案例研究的分析工作流在 11.12 节 (初始功能建模)、11.13 节 (初始类图)、11.14 节 (初始动态建模)、11.15 节 (修订实体类)、11.16 节 (提取边界类) 和 11.17 节 (提取控制类) 中描述。统一过程在 MSG 基金会案例研究的应用在 11.18 节 (用例实现)、11.19 节 (类图增量) 和 11.20 节 (软件项目管理计划) 中描述。11.21 节描述测试工作流。统一过程中的规格说明书在 11.12 节讨论。更多关于参与者和用例的信息出现在 11.23 节。分析工作流 CASE 工具在 11.24 节介绍。本章最后以讨论分析工作流面临的挑战作为结束 (11.25 节)。

延伸阅读材料

描述不同版本的面向对象分析技术的早期书籍包括 [Coad and Yourdon, 1991a; Rumbaugh et al., 1991; Shlaer and Mellor, 1992; and Booch, 1994]。如本章所提到的，这些技术 (和其他未列出的) 本质上是相同的。

除了这类型的面向对象分析技术，Fusion [Coleman et al., 1994] 是一种第二代 OOA 技术，它结合 (融合) 了很多第一代的技术，包括 OMT [Rumbaugh et al., 1991] 和 Objectory [Jacobson, Christerson, Jonsson, and Overgaard, 1992]。统一软件开发过程集成了 Jacobson、Booch 和 Rumbaugh [1999] 的工作成果。Catalysis 是另一种重要的面向对象方法 [D'Souza and Wills, 1999]。

ROOM 是一种用于实时软件开发的面向对象方法 [Selic, Gullekson, and Ward, 1995]。更多关于实时面向对象技术的信息可以在 [Awad, Kuusela, and Ziegler, 1996] 找到。

更多有关 UML 的细节可以在 [Booch, Rumbaugh, and Jacobson, 1999] 和 [Rumbaugn, Jacobson, and Booch, 1999] 中找到。1999 年 10 月发行的《Communications of the ACM》包含了很多关于 UML 使用的论文。UML 现在由对象管理小组 (Object Management Group) 负责维护，UML 的最新版本可于 OMG 网站 www.omg.org 获得。

本章讨论的用于提取候选类名词的技术在 [Juristo, Moreno, and López, 2000] 进行了规范化。CRC 卡片技术在 [Beck and Cunningham, 1989] 首次提出。Wirfs-Brock, Wilkerson 和 Wiener [1990] 对 CRC 卡片技术研究较深入。

许多比较各种面向对象分析技术的论文已经发表，包括 [de Champeaux and Faure, 1992; Monarchi and Puhr, 1992; and Embley, Jackson, and Woodfield, 1995]。有关面向对象和传统分析技术的比较出现在 [Fichman and Kemerer, 1992]。

管理面向对象项目中的迭代在 [Williams, 1996] 里描述。状态图在 [Harel and Gery, 1997] 中介绍。面向对象范型中规格说明的复用在 [Bellinzona, Fugini, and Pernici, 1995] 描述。

各式各样的关于面向对象软件开发形式化技术的论文出现在 2000 年 7 月发行的《IEEE Transactions on Software Engineering》中。

习题

11.1　为什么下面的约束条件不应该在规格说明中出现?

(i) 产品必须明显地减少在 Queensland 中心分销啤酒的运输费用。

(ii) 信用卡数据库必须建立在合理的成本上。

11.2　考虑下面烤 pockwester 鱼的食谱。

原料：1 个大洋葱、1 罐冰冻橙汁、1 个柠檬的新鲜榨汁、1 杯面包碎屑、面粉、牛奶、3 根中等大小的葱、2 个中等大小的茄子、1 条新鲜的 pockwester 鱼、1/2 杯 Pouilly Fuissé 白葡萄酒、1 头大

蒜、Parmesan 干酪和 4 个放养鸡蛋。

在前一个晚上，取一个柠檬，榨取柠檬汁，并将其冰冻。取一个大洋葱和 3 根葱，切成方块，放进平底锅烤。当开始冒黑烟时，加入 2 杯鲜橙汁。用力搅拌。把柠檬切成薄片，加入混合物中。同时，把蘑菇涂上面粉，蘸上牛奶，然后将它们和面包碎屑放在一个纸袋里摇动。在炖锅中加热半杯 Pouilly Fuissé 酒。当温度达到 170°时，加糖继续加热。当糖熔化成焦糖，放入蘑菇。混合 10 分钟或等到所有的结块开始消除时，加入鸡蛋。现在取 pockwester 鱼，撒上 frobs，杀死它。将 pockwester 鱼剥皮，切成小块，然后放入混合物中。煮开，然后慢煮，不加盖。鸡蛋先前应该用搅打器使劲搅动 5 分钟。当 pockwester 鱼变得柔软时，将它放在盘子上，撒上 Parmesan 干酪，烤不超过 4 分钟。

在上述规格说明中找出模棱两可、遗漏和矛盾的地方。（注：pockwester 是一种虚构的鱼，frobs 是一种开胃小点心的俚语。）

11.3　修改 11.12 节的规格说明段落，使之更准确地反映客户的要求。

11.4　使用数学公式表示 11.12 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　为确定银行陈述是否正确的产品编写一份精确的英语规格说明（习题 8.8）。

11.6　为图 11-10 描述的其他类绘制状态图，完成电梯问题案例研究。

11.7　在分析工作流中，最晚在什么时刻引入类而不会危害项目？

11.8　在统一过程中，最早什么时刻引入类更有意义？

11.9　有可能使用和本章描述的状态图不同的形式表示动态模型吗？解释你的答案。

11.10　在分析工作流中，为什么只确定类的属性，而不确定类的方法？

11.11　名词提取过程在 11.18.1 节描述。为什么不提取动词呢？以及其他六种词类（形容词、副词、连词、感叹词、介词、代词）呢？

11.12　给出图 10-30 和图 10-31`Manage an Investment` 用例的一个扩展场景。

11.13　给出图 10-17 和图 10-18 `Update Estimated Annual Operating Expenses` 用例的一个扩展场景。

11.14　给出图 11-40 和图 11-41 交互图的事件流。

11.15　给出图 11-43 和图 11-44 交互图的事件流。

11.16　检查你对习题 11.13 的解答是否是图 11-48 和图 11-49 交互图的一个可能场景。如果不是，修改你的场景。

11.17　给出图 11-48 和图 11-49 交互图的事件流。

11.18　给出图 11-54 和图 11-55 交互图的事件流。

11.19　（分析和设计项目）执行习题 8.7 中图书馆软件产品的分析工作流。

11.20　（分析和设计项目）执行习题 8.8 中确定银行陈述是否正确的产品的分析工作流。

11.21　（分析和设计项目）执行习题 8.9 中自动柜员机的分析工作流。不需要考虑组成硬件部分的，像读卡机、打印机和自动提款机的细节。相反地，只要假定，当 ATM 发送命令给这些组件时，它们正确执行。

11.22　（学期项目）执行附录 A 描述的 Osric 办公用品和装饰产品的分析工作流。

11.23　（学期项目）制定一份开发附录 A 描述的 Osric 办公用品和装饰产品的软件项目管理计划。

11.24　（案例研究）添加 **Report Class**（报告类）到 MSG 基金会案例研究的分析工作流中（11.12 ~ 11.19 节）。这是一项改进还是不必要的麻烦？

11.25　（案例研究）描述当分析工作流始于动态建模时，将会发生什么。从图 11-22 的状态图出发，完成 MSG 基金会案例研究的分析工作流过程。

11.26　（案例研究）11.20 节中的软件项目管理计划针对于一个只由 3 名软件工程师组成的小型软件工程组织。修改这份计划，使之适用于由 1 000 多名软件工程师组成的中型软件组织。

11.27　（案例研究）如果 MSG 基金会产品必须要在 8 周内完成，11.20 节中的软件项目管理计划应该怎么修改？

11.28　（软件工程读物）教师分发［Juristo，Moreno，and López，2000］的复印件。阅读并讨论你对文章作者关于面向对象分析方法的看法？

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.

[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.

[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. D'SOUZA AND H. 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.

[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.

[Harel and Gery, 1997] D. HAREL AND E. GERY, "Executable Object Modeling with Statecharts," *IEEE Computer* **30** (July 1997), pp. 31–42.

[IWSSD, 1986] Call for Papers, Fourth International Workshop on Software Specification and Design, *ACM SIGSOFT Software Engineering Notes* **11** (April 1986), pp. 94–96.

[Jacobson, Booch, and Rumbaugh, 1999] G. BOOCH, I. JACOBSON, AND J. RUMBAUGH, *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.

[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. I, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[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.

[Meyer, 1985] B. MEYER, "On Formalism in Specifications," *IEEE Software* **2** (January 1985), pp. 6–26.

[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.

[Naur, 1969] P. NAUR, "Programming by Action Clusters," *BIT* **9** (No. 3, 1969), pp. 250–58.

[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 12 章　设计工作流

学习目标

通过本章学习，读者应能：

- 完成设计工作流。
- 进行面向对象设计。

在过去近 40 年间，人们提出了数以百计的设计技术。其中一些是对已有技术的改进，另一些则与原有的完全不同。只有少部分的设计技术被成千上万的软件工程师所使用，而大部分仅仅是那些作者自己使用。一些设计策略，特别是那些由理论专家所提出的，有着坚实理论基础的，或者更实用的，之所以被提出来是因为那些作者发现它们在实际工作中效果很好。大部分设计技术都是人工的，但是自动化正渐渐成为设计的一个重要方面，特别是用于帮助文档管理。

在如此多的设计技术中，一个确定的基本模式渐渐形成。如前面所讨论的，一个软件产品的两个必需元素是它的操作和用于操作的数据。所以，设计一个产品的两种基本的方式是面向操作的设计和面向数据的设计。*在面向操作的设计*（Operation-Oriented Design）中，强调的是操作。例如数据流程分析［Yourdon and Constantine，1979］，其目标是设计高内聚的模块。*在面向数据的设计*（Data-Oriented Design）中，数据是被优先考虑的。例如，在 Michael Jackson 的技术中［Jackson，1975］，数据结构首先被确定，然后那些操作被设计成符合那些数据结构的。

面向操作的设计的缺点在于它集中于操作，而忽略了数据的重要性。面向数据的设计同样过分强调数据，而忽略了操作的重要性。解决方案是运用面向对象设计技术，它同等地对待操作和数据。在本章中，将在范例中详细讨论面向对象设计。

12.1　面向对象设计

统一过程的叙述需要*面向对象设计*（Object-Oriented Design，OOD）的知识。因此，先介绍面向对象设计，然后在 12.4 节中讨论统一过程的设计工作流。

面向对象设计的目标是将产品设计成对象，即在分析流中提取出来的类及其子类的实例。一些经典的程序设计语言，例如，C、老版的（2000 年前）COBOL 和 Fortran 并不支持对象的概念。这似乎意味着只有那些使用面向对象设计语言，如 Smalltalk［Goldberg and Robson，1989］、C++［Stroustrup，2003］、Ada 95［ISO/IEC 8652，1995］和 Java［Flanagan，2005］，的人才能够使用面向对象设计。

然而实际上并非如此。虽然一些经典程序设计语言并不支持面向对象设计，但是依然可以使用 OOD 的大部分方法。如 7.7 节所述，类是有继承属性的抽象数据类型，对象则是类的实例。当使用的编程语言不支持继承的时候，解决方法是利用编程语言所能够支持的面向对象设计的内容，也就是说，使用*抽象数据类型设计*（abstract data type design）。抽象数据类型可以使用任何支持类型（type）声明的编程语言实现。即使某种语言由于没有类型声明而无法支持抽象数据类型，它依然可能实现数据封装。图 7-28 描述了一种从模块到对象的层次设计概念。在无法完全运用面向对象设计的时候，开发人员应该尽量确保他们的设计能够使用多种语言所支持的图 7-28 所提到的层次概念中的最高层级。

面向对象设计的两个关键步骤是完成类图设计和详细设计。首先看第一步，完成类图

(class diagram)，在这一步中，需要确定属性的格式以及将方法分配给相关的类。属性的格式基本上可以从分析结果中直接得出，例如，在美国，日期，如 1947 年 12 月 3 日，被写成 12/03/1947（mm/dd/yyyy 的格式），而在欧洲则写成 03/12/1947（dd/mm/yyyy 的格式）。而无论使用哪种惯例，都总共需要 10 个字符。

决定属性格式的信息可在分析流中获得，可以在那时将其加入到类图中。然而，面向对象范型是迭代的。每一次迭代都可能改变原有的设计。出于实际原因，信息应该尽可能晚地加入到 UML 模型。可以参考一下图 11-18、图 11-19、图 11-25、图 11-60 这 4 个 MSG 基金会案例的前 4 次迭代的类图。没有哪一次的迭代结果包含类的属性。如果属性被过早地决定，它们可能会被修改，或是从一个类转移到另一个类，直到分析团队对完成的类图满意为止。也就是说，那些类图本身必须不断的被修改。一般而言，在确定属性必须被加入某个类图（或其他 UML 图）前这样做没有任何意义，因为这样做将会给下一次迭代增加不必要的工作。当然，在确定之前决定属性的格式也是没有意义的。

面向对象设计第一步的另一个重要部分是分配方法（操作的实现）给类。通过检查每一个场景的交互图来确定产品的所有操作。这个很容易，难的是如何确定哪些方法应该属于某个类。

当需要发送消息给某个类或者客户（对象的客户是发送消息给该对象的程序单元）时，可以将方法分配给这个类或客户。一个可以帮助决定如何分配操作的原则是信息隐藏（7.6 节）。也就是说，当一个类的状态变量应该被声明为 **private**（仅在该类的对象内可访问）或者 **protected**（仅在该类或其子类的对象内可访问）时，相应的操作这些变量的方法应该分配给这个类。

第二个原则是，如果一个操作被一个对象的不同客户所调用，那么将这个操作实现为该对象的一个方法比在每一个客户中实现它要有意义。

第三个用于分配操作的原则是使用职责驱动设计。如 1.9 节所述，职责驱动设计（responsibility-driven design）是面向对象设计的一个重要方面。如果一个客户发送一个消息给一个对象，该对象有责任实现完成此客户请求的每个细节。客户不知道此请求是如何完成的，也不允许知道。一旦请求完成，控制就回到客户手中。就这点来说，每个客户都知道请求已经完成了，但并不知道它是如何完成的。

面向对象设计的第二步是细节设计，在这个过程中每个类将被详细设计（detailed design）。例如，选择特定的数据结构和算法。一种表示细节设计的方法如图 12-1 所示，这是 MSG 基金会案例中 **Mortgage** 类的 find 方法的细节设计。

类　名	**Mortgage**
方法名	find
返回类型	**boolean**
输入参数	String findMortgageID
输出参数	None
错误信息	If file not found, prints message `***** Error: Mortgage.find () *****`
文件访问	`mortgage.dat`
文件修改	None
方法调用	None
描述说明	Method find locates a given mortgage record if it exists. It returns true if the mortgage is located, otherwise false.

图 12-1　**Mortgage** 类中 find 方法的细节设计

另一种表示细节设计的方法如图 12-2 和图 12-3 所示。图 12-2 显示了 MSG 基金会案例中 **EstimateFundsForWeek** 类的 **computeEstimatedFunds** 方法，该方法调用了 **Mortgage** 类的 **totalWeeklyNetPayments** 方法，如图 12-3 所示。

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
   float expectedWeeklyInvestmentReturn;          (expected weekly investment return)
   float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                  (expected total mortgage payments
                                                  less total weekly grants)
   float estimatedFunds = (float) 0.0;            (total estimated funds for week)
Create an instance of an investment record.
   Investment inv = new Investment ( );
Create an instance of a mortgage record.
   Mortgage mort = new Mortgage ( );
Invoke method totalWeeklyReturnOnInvestment.
   expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );
Invoke method expectedTotalWeeklyNetPayments          (参见图12-3)
   expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );
Now compute the estimated funds for the week.
   estimatedFunds = (expectedWeeklyInvestmentReturn
         - (MSGApplication.getAnnualOperatingExpenses ( ) / (float) 52.0)
         + expectedTotalWeeklyNetPayments);

Store this value in the appropriate location.
   MSGApplication.setEstimatedFundsForWeek (estimatedFunds);
} // computeEstimatedFunds
```

图 12-2　MSG 基金会案例中 **Estimate Funds For week** 类的 **compute Estimated Funds** 方法的细节设计

图 12-2 和图 12-3 是用 Java 风格的程序描述语言（Program Description Language，PDL，早期也称为伪代码）写的。PDL 实际上由编程语言的控制语句和注释组成。PDL 的优点在于清晰、简明，实现阶段通常只需要将注释转化为相应的程序代码。而缺点是它趋向于使设计者涉及太多细节，甚至产生完整的类的代码实现而不仅仅是一个 PDL 的细节设计。

```
This method computes the net total weekly payments made by the mortgagees, that is, the expected total weekly
mortgage amount less the expected total weekly grants.
{
  File mortgageFile = new File ("mortgage.dat");          (file of mortgage records)
  float expectedTotalWeeklyMortgages = (float) 0.0;       (expected total weekly mortgage payments)
  float expectedTotalWeeklyGrants = (float) 0.0;          (expected total weekly grants)
  float capitalRepayment;                                 (capital repayment)
  float interestPayment;                                  (interest payment)
  float escrowPayment;                                    (escrow payment)
  float tempMortgage;                                     (temporary value)
  float maximumPermittedMortgagePayment;                  (maximum amount the couple may pay)
Open the file of mortgages, name it inFile, and read each element in turn.
  {
    read (inFile);
Compute the capital repayment, interest payment, and escrow payment for this mortgage.
    capitalRepayment = price / NUMBER_OF_MORTGAGE_PAYMENTS;
    interestPayment = mortgageBalance * INTEREST_RATE / WEEKS_IN_YEAR ;
    escrowPayment = (annualPropertyTax + annualInsurancePremium) / WEEKS_IN_YEAR;
First assume that the couple can pay the mortgage in full, without a grant.
    tempMortgage = capitalRepayment + interestPayment + escrowPayment;
Add this amount to the running total of mortgage payments.
    expectedTotalWeeklyMortgages += tempMortgage;
Now determine how much the couple can actually pay.
    maximumPermittedMortgagePayment = currentWeeklyIncome *
          MAXIMUM_PERC_OF_INCOME;
If a grant is needed, add the grant amount to the running total of grants.
    if (tempMortgage > maximumPermittedMortgagePayment)
      expectedTotalWeeklyGrants += tempMortgage - maximumPermittedMortgagePayment;
  }
Close the file of mortgages.  Return the total expected net payments for the week.
  return (expectedTotalWeeklyMortgages - expectedTotalWeeklyGrants);
} // totalWeeklyNetPayments
```

图 12-3　MSG 基金会案例中 **Mortgage** 类的 **totalWeeklyNetPayments** 方法

为了说明这些原则是如何工作的，接下来用两个案例来阐述。和前面一样，第一个案例是电梯问题，只考虑一个电梯的简单情况，然后再回到 MSG 基金会案例。

12.2　面向对象设计：电梯问题案例研究

步骤 1　完成类图。

通过向图 11-10 中的类图增加操作（方法）可以得到设计工作流中的类图（图 12-4）。（用 Java 实现时，需要两个额外的类：**Elevator Application** 类对应C++ 的主函数，**Elevator Utilities** 类包含在C++ 中类以外的函数的 Java 实现）。

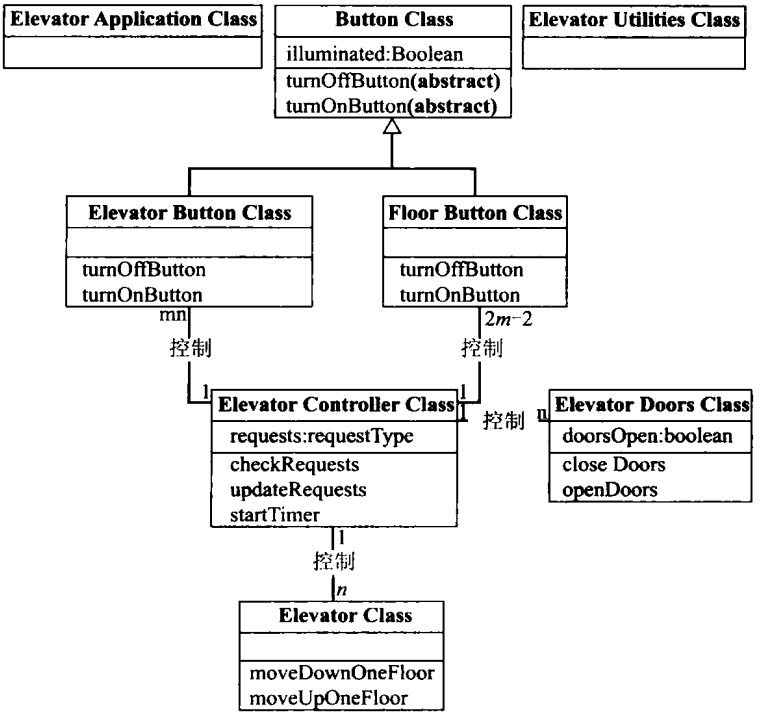


图 12-4　电梯问题案例的详细类图

考虑电梯控制器中 CRC 卡片的第 2 次迭代（图 11-9）。所有的职责分成两组。其中三项职责“8. 启动计时器”、“10. 检查请求”、“11. 更新请求”，依据职责驱动设计的原则分配给电梯控制器；这些任务由电梯控制器本身负责执行。

其余的 8 项职责（事件 1 ~ 7 和事件 9）都是“发送消息给另一个类来通知它执行某些任务”的形式。这意味着此处使用的分配职责给类的原则也是职责驱动设计。另外，出于安全性的考虑，信息隐藏的原则在这里同样适用。

基于以上两个考虑，方法 `closeDoors` 和 `openDoors` 分配给 **Elevator Doors** 类，也就是说，**Elevator Doors** 类的客户（在这里就是 **Elevator Controller** 类的实例）发送消息给 **Elevator Doors** 类的实例来打开或关闭电梯的门，这个请求由相应的方法来执行。这两个方法的具体细节被封装在 **Elevator Doors** 类中。另外，信息隐藏使得 **Elevator Doors** 类完全独立，可以独立地进行细节设计和实现，并可以在其他的产品中复用。

相同的设计原则可以应用于方法 `moveDownOneFloor` 和 `moveUpOneFloor`，它们被分配给 `Elevator` 类。这里并不需要一条额外指令来停止电梯运行。如果这两个方法都没有被调用，电梯便无法移动；除了调用这两个方法的其中之一以外，没有其他方式可以改变电梯的状态。

最后，方法 `turnOffButton` 和 `turnOnButton` 同时分配给 **Elevator Button** 类和 **Floor Button** 类。理由与分配方法给 **Elevator Doors** 类和 **Elevator** 类一样。第一，职责驱动设计的原则要求按钮可

以完全控制它们自己的开关状态。第二，信息隐藏的原则要求按钮的内部状态被隐藏。因此打开或关闭按钮的方法必须在 **Elevator Button** 类的内部。同样的道理也适用于 **Floor Button** 类。利用多态和动态绑定，方法 `turnOffButton` 和 `turnOnButton` 可以在基类 **Button** 类中声明为 **abstract**（**virtual**），理由可以参考 7.8 节。在运行时，`turnOffButton` 或 `turnOnButton` 的正确版本将被调用。

步骤 2　执行细节设计。

现在可以开始对所有类进行详细设计。任何合适的详细设计技术都可以使用，例如，第 5 章描述的逐步精化。方法 elevatorEventLoop 的详细设计如图 12-5 所示。此设计用基于C++ 的 PDL 实现。

图 12-5 基于状态图 11-7。例如，在图 12-5 的第一部分，事件 `button pushed`、`button unlit` 通过两层嵌套的 **if** 语句实现。接下来是 **Process Requests** 状态的两个操作。**else-if** 语句对应于 **Elevator Event Loop** 状态的下一个事件，电梯向 d 层方向移动，下一层是 f。剩下的部分按照同样的过程就可以了。

```
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
      {
        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;
      determine direction of next request;
      if (appropriate floorButton is on)
        floorButton::turnOffButton;
      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 12-5　方法 `elevatorEventLoop` 的详细设计

现在考虑 MSG 基金会案例的面向对象设计。

12.3　面向对象设计：MSG 基金会案例研究

如 12.1 节所描述的，面向对象设计由两个步骤组成。

步骤 1　完成类图。

MSG 基金会案例的最终的类图如图 12-6 所示。

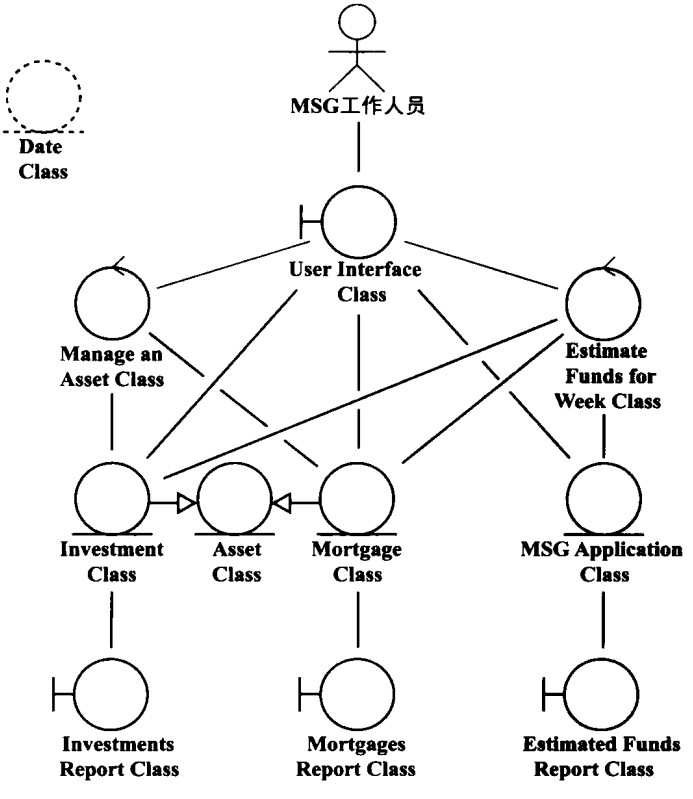


图 12-6　MSG 基金会案例的全部类图

用户定义的 **Date Class** 用虚线表示说明它仅仅需要一个C++ 的实现；Java 有内建的类来处理日期，包括 **java. text. Dateformat** 和 **java. util. Calendar**。

类的属性的格式可以从分析流中和客户以及用户的讨论中得到，在这点上，对结果的检查（见 10.4.2 节）对此也非常有用。结果的一部分如图 12-7 所示。

产品的方法可以从众多的交互图中得到。设计者的任务是将方法分配给特定的类。例如，在面向对象软件产品中，属性的改变通过给类的每一个属性关联两个方法来实现，一个是修改方法 **setAttribute**，用于给属性赋值，另一个是访问方法 **getAttribute**，用于获得属性的当前值。

例如，考虑方法 **setAssetNumber**，用于将一个数值赋给某个资产（**investment** 或者是 **mortgage**）。在传统的设计中，需要两个单独的函数 **set_investment_number** 和 **set_mortgage_number** 来实现。由于面向对象设计支持继承，因此方法 **setAssetNumber** 可以分配给 **Asset Class**。于是，如图 12-8 所反映的，此方法不仅仅可以应用在 **Asset Class** 的实例上，由于继承的结果，同样可以应用在 **Asset Class** 的任何子类的实例上，也包括 **Investment Class** 和

Mortgage Class 的实例。同样的，方法 **getAssetNumber** 应该分配给超类 **Asset Class**。

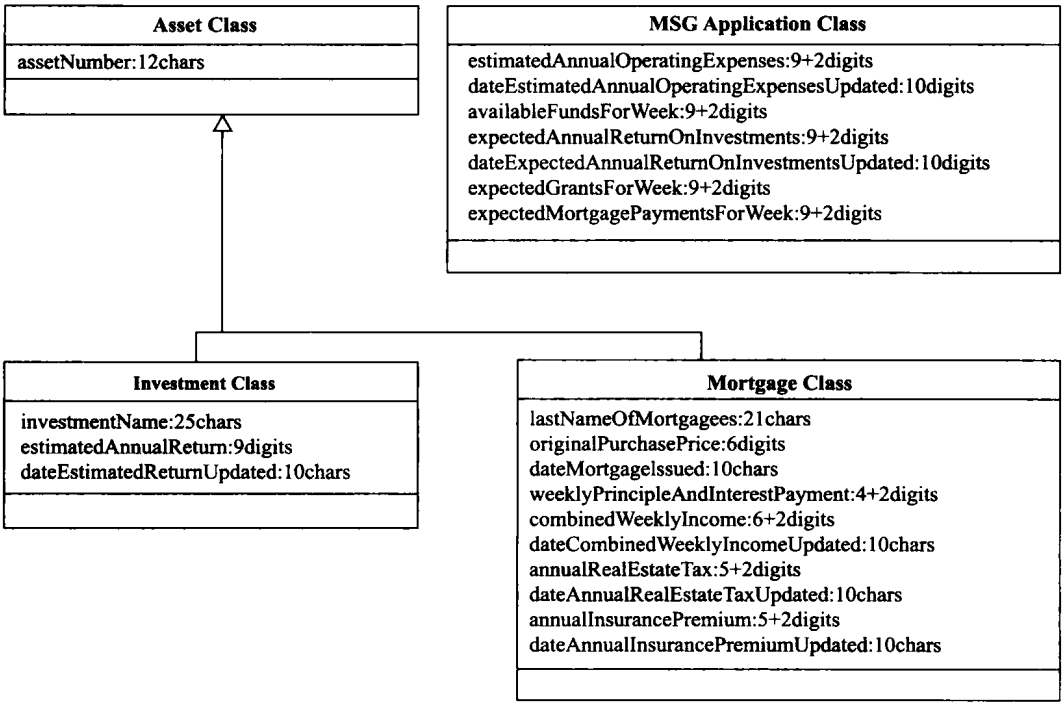


图 12-7　带属性格式的 MSG 基金会案例的部分类图

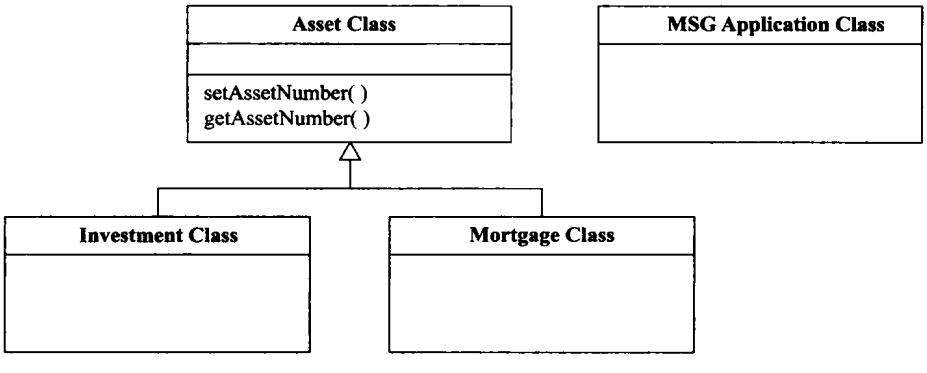


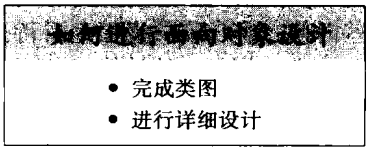
图 12-8　把方法 **setAssetNumber** 和 **getAssetNumber** 分配给 **Asset Class** 的 MSG 基金会案例的部分类图

同样地分配其他的方法给适当的类。最后的设计结果见附录 F。

步骤 2　执行详细设计。

接下来，详细设计的内容是决定每一个方法完成什么任务。其中 3 个方法的详细设计已经在 12.1 节讨论过了。**Mortgage** 类的 **findMortgageID** 方法的表格式详细设计如图 12-1 所示。MSG 基金会案例中的 **EstimateFundsForWeek** 类的 **computeEstimatedFunds** 方法的细节设计（基于 Java 的 PDL）如图 12-2 所示。该方法调用的 **Mortgage** 类的 **totalWeeklyNetPayment** 方法如图 12-3 所示。

面向对象设计的步骤小结见“如何进行面向对象设计”。



12.4　设计工作流

设计工作流的输入是分析工作流的结果（第11章）。这些结果在设计工作流中通过迭代不断被完善，直到成为程序员可以用的形式。

迭代的一方面是确定方法以及将其分配给适当的类。另一方面是进行详细设计。这两个步骤组成了面向对象设计的设计工作流。

除面向对象设计之外，许多决定需要作为设计工作流的一部分。其中一个决定就是选择用于实现软件产品的编程语言。这部分将在第13章中讨论。另一个决定是现有软件产品在新的软件中的复用性。复用性在第8章描述。可移植性是另一个设计中要考虑的问题。这个问题同样在第8章中讨论。此外，大型的软件产品通常都在网络环境中运行的，于是，另一个问题是如何分配软件组件在不同的硬件上运行。

统一过程开发的主要目的是提供一种适用于大型软件开发的方法，其代码量在500 000行以上。附录G和H中关于MSG基金会案例的C++和Java的实现各自都不超过5 000行代码。也就是说，统一过程主要用于开发至少是本书中的MSG基金会案例100倍大小的软件产品。因此，统一过程的许多方面并不适用于这个案例。例如，分析流的一个重要部分是将软件产品划分成不同的包。每一个包（package）由一些相关的类组成，这些类通常是与参与者的一个子集相关，并且可以作为一个单独的单元实现。例如，应付账目、可接受账目和总账都是典型的包。包的根本意义在于它是比大型软件更容易实现的小型系统。所以，如果一个大型软件能够分解成相互独立的包，将更容易实现。

同样，将大的工作流划分成相互独立的小的设计工作流的思想也被引入设计工作流中。于是，现在的目标是将接下来的实现流程划分为可管理的子系统（subsystem）的小块。再次说明，将MSG基金会案例划分成子系统没有任何意义，因为它已经足够小了。

将大型的工作流划分成子系统有两个理由：

1）如前面所说，实现一些子系统比实现整个大的系统相对要容易些。

2）如果要实现的子系统足够的相对独立，那么它们可以由不同的团队同时开发，这样一来整个软件开发的时间将可以缩短。

回顾一下8.5.4节，软件的体系结构包括众多的组件和它们的整合方式。将组件划分成子系统是构架任务的一个重要方面。确定一个软件的体系结构决不是一件容易的事情，除了一些极小的软件之外，这一般是由被称为软件构架师（architect）的专家来完成。

除了必须是技术上的专家，软件构架师还必须知道如何选择折衷的方案。一个软件必须满足功能上的需求，也就是用例。同时它也需要满足一些非功能上的需求，包括可移植性（第8章）、稳定性（6.4.2节）、健壮性（6.4.3节）、可维护性以及安全性。开发者需要在预算和时间的约束下完成所有的这些事情。当然，开发一个软件产品几乎不可能在预算和时间的约束下同时满足所有这些功能性和非功能性的需求。所以折衷总是需要的。客户要么放宽一下要求，要么增加预算，或者延长开发期限，甚至需要同时放宽这两个或更多的条件。软件构架师必须根据客户的决定来制定出折衷的方案。

有些时候折衷是显而易见的。例如，软件构架师可能指出满足一些符合新的安全标准的安全性需求需要多3个月的时间和350 000美元的额外预算。如果产品是一个国际银行业务网络系统，那么折衷问题将是毫无异议的，客户无论如何不可能在安全性方面做出任何退让。然而有些时候，客户需要谨慎地看待折衷，并需要在软件构架师的专业意见的指导下来做出正确的商业决定。例如，软件构架师可能指出在开发过程中推迟某个特殊的需求到后期，可能会节省150 000美元的成本，但在以后要整合这个功能却可能需要花费300 000美元（图1-5）。是否推迟某些需求只能由客户来决定，但他需要软件构架师的专业意见来帮助他做出正确的决定。

体系结构是决定一个软件产品最终成败的至关重要的因素。一些关于体系结构的重要决定需要在设计工作流中做完。如果需求分析阶段完成的不是很好，通过在分析流中付出更多的时间和金钱，依然可以成功地完成一个项目。同样的，就算分析工作流没做好，还可以在设计工作流中努力来弥补。但是，如果体系结构没设计好，将是无法补救的；唯一的方法就是立即重新设计。这就是为什么一个开发团队需要一个具备专业知识和交流技能的软件构架师。

12.5　测试工作流：设计

对设计进行测试的目的是检验产品规格说明是否被正确、完整地整合进设计中，以及确定设计本身的正确性。例如，设计必须没有逻辑错误，并且所有的接口都被正确的定义。在开始编码前发现设计中的所有错误是非常重要的。否则，修复错误的代价将格外的高，就像图 1-5 所反映的。设计错误可以通过设计审查和设计走查等手段来发现。本节剩下的内容将讨论设计审查，这些内容同样适用于设计走查。

事务（transaction）对产品的用户而言就是一个操作，就像“处理一个请求”和“打印今天的订单列表”。当一个产品是面向事务的，设计审查就必须能反映出事务［Beizer，1990］。审查必须考虑所有可能的事务类型，将设计中的每一个事务与规格说明联系起来，并在规格说明文档中说明事务是如何发生的。例如，对于一个自动取款机的应用来说，事务对应用户所能够执行的每一个操作，例如，用一个信用卡账户存款和取款。在某些情况下，规格说明和事务之间的对应关系可能并不一定是一对一的。在一个交通灯控制系统中，如果一辆机动车通过一个传感器导致系统决定将一个灯由红变为绿 15 秒，而这之后该传感器的脉冲将被忽略。相反地，为了提高交通流量，一个单一的脉冲可能导致一系列的交通灯由红变为绿。

严格的事务驱动审查（transaction-driven inspections）无法检查出那些规格说明中要求而被设计者忽略的事务实例。举个极端的例子，一份交通灯控制系统的规格说明可能规定在 11: 00 P. M 和 6: 00 A. M 之间一个方向的所有灯都闪烁黄光，而另一个方向则是红光。如果设计者忽略了这个规定，那么在 11: 00 P. M 和 6: 00 A. M 时间段的时钟事务将不被包括在设计里；一旦这个事务被忽略，它们也不会在基于事务的设计审查中被测试。因此，仅仅基于事务驱动的设计审查是不够的。规格驱动审查同样是必需的，以便确保规格说明文档里的声明没有被忽略或者误解。

12.6　测试工作流：MSG 基金会案例

到现在为止，所有的设计都完成了，MSG 基金会案例的设计的所有方面都必须通过设计审查的方式来检测（6.2.3 节）。特别需要注意的，每一项设计结果都需要检查。在实现阶段，即使没有发现任何错误，设计也可能再一次被更改，甚至是从根本上改变。

12.7　详细设计的形式化技术

前面章节已经介绍过一种详细设计技术。第 5.1 节给出了逐步精化方法，其后介绍了如何把流程图应用于详细设计。除逐步精化技术之外，形式化技术的使用也有利于详细设计。第 6 章曾提到实现一个完整的系统然后再证明它的正确性是达不到预期目标的。然而，如果证明过程与详细设计并行进行并且仔细地测试代码，则效果明显不一样。形式化技术能在以下 3 个方面给详细设计带来帮助：

1）关于正确性证明的情况是，虽然它无法用于证明整个系统的正确性，却适用于模块大小的子系统的正确性证明。

2）提出证明并进行详细设计相比不进行正确性证明可以得到一个错误更少的设计。

3）如果由同一个程序员来完成详细设计和实现，那么他能够确信他的设计是正确的，并且这种自信能使得他的代码的错误更少。

12.8　实时设计技术

如6.4.4节所说，实时软件（real-time software）的特点是严格的时间约束，这种时间约束的性质是，一旦约束不被满足，信息将会丢失。也就是说，每一个输入都必须在下一个输入到来之前被处理。实时系统的一个例子是由计算机控制的核反应堆。反应堆核心的温度以及腔中水量的多少等信息持续地发送至计算机，计算机在下一次输入到来之前根据当前的输入值进行一些必要的处理。另一个例子是由计算机控制的看护病房。这里涉及病人的两种数据：每一个病人的心率、体温和血压等常规信息，以及紧急信息，当系统确定某个病人的情况变得危急的时候，在这种紧急情况下，软件必须能够同时处理那些常规信息以及当前的紧急信息。

大部分实时系统的特点是它们都通过分布式系统来实现。例如，控制一架战斗机的软件可能通过5台计算机实现：第1台用于导航、第2台用于控制武器系统、第3台用来制定电子对策、第4台用来控制机翼和引擎等设备，而第5台用于提供战斗策略。由于硬件并不是百分百可靠的，可能还有一些备用的计算机用于自动代替出现故障的单元。设计这样的系统的关键不仅仅是通信问题，也包括时间问题，除此之外，还有一系列由系统的分布性带来的问题。例如，在战斗环境中，战术系统可能建议驾驶员上升，而武器系统则建议降低高度以便于某个武器在最佳条件下发射。然而驾驶员可能最后决定右转，于是发送一个信号给硬件控制系统以做出必要的调整来使飞机按指示的方向飞行。所有的这些信息都必须被小心地处理以便让飞机执行的实际动作优先于建议的动作。此外，实际的动作必须反馈给战术系统和武器系统，以便它们能根据实际情况再给出新的建议。

在实时系统中一个更难的问题是同步。如果一个实时系统需要在分布式的硬件上实现，那么当两个操作都独占一个数据并都需要对方的数据的时候，就会发生死锁。当然，死锁并不只是发生在分布式的实时系统中。但这确实给那些无法控制这种情况的实时系统带来了一定的麻烦。并且在分布式的实时系统中情况将变得更为复杂。除了死锁之外，其他的同步问题也可能发生，例如竞争条件。需要详细了解的读者可以参考［Silberschatz，Galvin，and Gagne，2002］或者其他关于操作系统的书籍。

从以上示例中可以清楚地看到，实时系统设计最主要的困难在于使设计满足那些时间约束。也就是说，设计技术能够提供一种机制来检测和确定，当产品完成后，它能够以要求的速率读取和处理数据。更进一步的，能够确定设计中的同步问题被很好地处理。

从计算机时代开始，硬件技术的发展几乎总是超过了软件技术。因此，尽管能够处理上述实时系统中的细节问题的硬件已经存在，软件设计技术却远远地落后了。实时软件工程的某些领域，已经取得了很大的进步。例如，第11章的分析技术已经拓展到了特定的实时系统。不幸的是，软件设计依然没有达到理论上的水平。虽然有了很大的发展，但现在软件设计的状况仍然无法达到分析的相同水准。因为实时系统的任何设计技术几乎都比根本没有技术强，所以有些设计技术还是在实际中得到了应用。但是要做到能够设计一个实时系统，在系统被实现前就确定所有的实时约束都被满足和所有的同步问题都不会发生，还有很长的一段路要走。

一些旧的实时系统设计技术，比如实时系统的结构化开发［Ward and Mellor，1985］是非实时技术向实时领域的扩展。一些新的技术可以参考［Liu，2000］和［Gomaa，2000］。

如前面所描述的，令人遗憾的是实时设计技术的现状并不像期望的一样好，但无论如何，这种状况在慢慢改变。

12.9　用于设计的CASE工具

如12.5节所述，设计的一个重要方面是测试设计结果是否正确地符合分析的每个方面。因此需要一个能够同时用于分析结果和设计结果的CASE工具，也就是所谓的前端工具或者高端

CASE 工具（相对于有助于实现制品的后端工具或是低端 CASE 工具）。

市场上已经有一些高端 CASE 工具。它们通常都是基于数据字典实现的。典型的数据字典形式如图 12-9 所示。CASE 工具整合了一个能够用于确定是否所有数据字典中的项目都在设计结果中提到过以及是否所有设计结果都被反映在分析结果中的一致性检测工具。更进一步的，许多高端 CASE 工具还整合了屏幕和报告生成器。也就是说，客户可以制定哪些项目出现在报告中或输出到屏幕上以及它们放置的位置和方式。因为每一个项目的所有细节都在数据字典中，CASE 工具能够根据用户的需求很容易地生成打印报告或是显示输入屏幕的代码。一些高端 CASE 工具同时整合用于估算和计划的管理工具。

关于设计工作流，像 Together 和 IBM Rational Rose 等 CASE 工具就能够在面向对象的生命周期中给设计工作流提供支持。类似的开源的 CASE 工具有 ArgoUML。

名　字	访问修饰	描　述	说　明
Asset	**package private** （默认）	抽象类 属性： assetNumber 访问方法/修改方法： getAssetNumber setAssetNumber 虚方法： read print write find obtainNewData performDeletion 方法： add delete	**Investment** 和 **Mortgage** 类的抽象子类。包含用户用于增加或删除一项资产的属性和方法
assetNumber	**protected**	12 位整数	方法 getAssetNumber 返回的特有数值。前 10 位包含资产号本身后 2 位是校验位
delete	**public**	方法返回类型： **void** 输入参数： 无 输出参数： 无	该方法调用方法 obtainNewData、save 和 UserInterface. pressEnter 以增加一个新的资产（投资或抵押）

图 12-9　MSG 基金会案例中 **Asset Class** 的数据字典形式

12.10　设计的度量

有许多的度量方法用来刻画设计的各个方面。例如，方法和类的数量是度量目标产品大小的粗略量度。内聚和耦合是和错误数量一样用于衡量设计质量的方法。在任何设计审查中，记

录下发现的设计错误的数量和类型是至关重要的。这些信息可用于产品代码的审查以及后继产品的设计审查。

详细设计的*秩复杂度*（cyclomatic complexity）*M* 可以是设计中二元判定（谓词）的数目加1，也可以是代码中的分支数。秩复杂度被认为是设计质量的一种度量，*M* 值越小，设计就越好。这种度量的一个优点在于它易于计算。然而它有一个内在的问题。秩复杂度是完全关于控制复杂度的度量，而数据复杂度被忽略了。也就是说秩复杂度无法度量用于数据驱动的代码的复杂度，例如表格中的值。举个例子，假设一个设计师不知道C++ 的库函数 toascii，并打算设计一段代码用于读取用户输入的一个字符并返回它的 ASCII 代码（一个 0 ~ 127 之间的整数）。一种实现这个设计的方法是用一个有 128 个分支的 **switch** 语句。另一种方法是按 ASCII 码顺序将 128 个 ASCII 代码存入一个数组，并利用一个循环来比较用户输入的字符和数组中的每一个值比较，当相等时退出循环，循环变量的当前值就是正确的 ASCII 值。这两种设计的功能是完全一样的，却各自有不同的秩复杂度 128 和 1。

另一个问题是一个类的秩复杂度通常都很低，因为大部分类都包括很多小而简单的方法。更进一步，如前面所指出的，秩复杂度忽略了数据复杂度。由于数据和操作是面向对象范型中同样重要的两个部分，秩复杂度忽略了决定一个对象复杂度的重要组成部分。因此，与秩复杂度相关的衡量类的复杂度的方法一般没有什么意义。

一种设计工作流中度量类的方法是基于由设计转换成的一张有向图，图中的节点代表类，节点间的连线（边）代表类之间的流（将消息发送给方法）。一个类的 **fan-in** 定义为指向该类的边的数量与该类访问的全局数据结构的数量之和。同样的，一个类的 **fan-out** 定义为从该类发出的边的数量与该类更新的全局数据结构的数量之和。一个类的复杂度就可以用 length × (fan-in × fan-out)^2 [Henry and Kafura, 1981] 来定义，其中 **length** 是该类大小的度量（见 9.2.1 节）。因为 fan-in 和 fan-out 的定义涉及全局数据，这种度量有数据相关的部分。然而，实验表明这种复杂度的度量方法并不比秩复杂度 [Kitchenham, Pickard, and Linkman, 1990; Shepperd, 1990] 之类的简单度量方法好。

前面提出过许多面向对象设计的度量方法，例如 [Chidamber and Kemerer, 1994]。而所有这些方法在理论和实验上被质疑过 [Binkley and Schach, 1996; 1997; 1998]。

12.11　设计工作流面临的挑战

如 11.25 节所指出的，在分析工作流中不要过于深入，这一点很重要，因为分析团队不应该过早的开始设计工作流。在设计工作流中，设计团队可能会犯两种错误：做的过多或是做的太少。

考虑图 12-2、图 12-3 和图 12-5 的 PDL（伪代码）详细设计。对一个喜爱编程的设计者来说，用C++ 或 Java 来完成详细设计而不仅仅是 PDL 对他们诱惑是很大的。也就是说，设计者可能编写所有实现的代码，而不仅仅是用伪代码勾勒细节。而实现整个类要比概述这个类花更多的时间，并将导致花更多的时间用于修复设计中发现的错误（见图 1-5）。如同分析团队一样，设计团队应该尽量只完成要求他们做的工作。

此外，还有一个更重要的挑战。在《No Silver Bullet》[Brooks, 1986]（见备忘录 3.5）一文中，Brooks 描述了他所定义的“优秀的设计师”（就是比设计团队中其他成员更突出的设计师）的缺乏。Brooks 认为，一个软件项目成功与否很大程度上取决于设计团队是否有一个优秀的领导者。好的设计是可以学习的，优秀的设计只能出自于优秀的设计师，但优秀的设计师实在太少了。

于是，挑战在于培养优秀的设计师。他们应该被越早识别越好（最好的设计师并不一定是

经验丰富的设计师），给他们指定一个指导者，提供正规的培训和训练以及成为优秀设计师的学徒期，并允许他们与其他设计者相互学习。这样，他们将拥有一个明确的职业生涯规划，而且他们得到的薪水应当与优秀设计师对项目所做的贡献相称。

本章回顾

12.1 节介绍了面向对象设计。12.2 节中的电梯问题案例和 12.3 节中的 MSG 基金会案例是面向对象设计的应用。12.4 节介绍了设计工作流。测试工作流的设计方面在 12.5 中描述，并应用于 12.6 节中的 MSG 基金会案例。详细设计的形式化技术在 12.7 节中讨论。12.8 节描述了实时系统的设计。CASE 工具和设计的度量分别在 12.9 节和 12.10 节中讲述。本章最后是一个关于设计工作流挑战的讨论（12.11 节）。

延伸阅读材料

关于面向对象设计的信息可以从［Wirfs-Brock，Wilkerson，and Wiener，1990；Coad and Yourdon，1991b；Shlaer and Mellor，1992；and Jacobson，Booch，and Rumbaugh，1999］获得。面向对象设计技术的比较可以参考［Monarchi and Puhr，1992］和［Walker，1992］。Briand，Bunse 和 Daly［2001］讨论了面向对象设计的可维护性。［Fichman and Kemerer，1992］中对面向对象设计技术与经典的设计技术进行了对比。［Jackson and Chapin，2000］中描述了空中交通控制系统的重新设计。高性能的可靠系统的设计技术可以参考［Stolper，1999］。

［Hoare，1987］中讲述了形式化设计技术。

关于设计工作流审查，早期有关设计审查的论文有［Fagan，1976］，详细的信息可以从这篇论文中获得。之后审查技术的发展可以参考［Bias，1991］。体系结构的审查在［Maranzano et al.，2005］有讨论。

关于实时系统设计，一些特别的技术可以从［Liu，2000］和［Gomaa，2000］中找到。［Kelly and Sherif，1992］中对比了 4 种实时系统设计技术。［Luqi，Zhang，Berzins，and Qiao，2004］中描述了复杂实时系统设计的文档驱动方法。［Magee and Kramer，1999］中描述了并发系统的设计。IEEE Software 2005 年的 3 月刊和 4 月刊包含一些设计方面的论文。

［Henry and Kafura，1981］和［Zage and Zage，1993］中描述了设计的度量。［Chidamber and Kemerer，1994］和［Binkley and Schach，1996］中讨论了面向对象设计的度量。面向对象的质量模型在［Bansiya and Davis，2002］中讨论到。

International Workshops on Software Specification and Design 有大量设计技术的相关信息。

习题

12.1　用 PDL 表示图 12-1 中的表格式的详细设计。

12.2　用表格的方式表示图 12-2 中 PDL 描述的详细设计。

12.3　用表格的方式表示图 12-3 中 PDL 描述的详细设计。

12.4　为什么在设计工作流中分配方法给类而不是在分析流中？

12.5　为什么在分析流中分配属性给类而不是在设计工作流中？

12.6　为什么秩复杂度度量方法的有效性会被质疑？

12.7　（分析与设计项目）以习题 11.19 中图书馆自动借阅系统的分析工作流结果为基础执行设计工作流。

12.8　（分析与设计项目）以习题 11.20 中银行状态确认系统的分析工作流结果为基础执行设计工作流。

12.9　（分析与设计项目）以习题 11.21 中 ATM 软件的分析工作流结果为基础执行设计工作流。

12.10　（学期项目）以习题 11.22 的分析工作流结果为基础，执行 Osric 办公用品和装饰产品（附录 A）

的设计工作流。

12.11 （案例研究）用老师指定的方法重新设计 MSG 基金会产品。

12.12 （软件工程读物）教师分发［Luqi，Zhang，Berzins，and Qiao，2004］的复印件。讨论你是否会使用其中的方法来设计实时系统？给出你的理由？

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 CliffsNJ, 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, 2005] D. FLANAGAN, *Java in a Nutshell: A Desktop Quick Reference*, 5th ed., O'Reilly and Associates, Sebastopol, CA, 2005.

[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.

[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] I. JACOBSON, G. BOOCH, AND J. RUMBAUGH, *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," *Software Engineering Journal* **5** (January 1990), pp. 50–58.

[Liu, 2000] J. W. S. LIU, *Real Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[Luqi, Zhang, Berzins, and Qiao, 2004] LUQI, L. ZHANG, V. BERZINS, AND Y. QIAO, "Documentation Driven Development for Complex Real-Time Systems," *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 936–52.

[Magee and Kramer, 1999] J. MAGEE AND J. KRAMER, *Concurrency: State Models & Java Programs*, John Wiley and Sons, New York, 1999.

[Maranzano et al., 2005] J. F. MARANZANO, S. A. ROZSYPAL, G. H. ZIMMERMAN, G. W. WARNKEN, P. E. WIRTH, AND D. M. WEISS, "Architecture Reviews: Practice and Experience," *IEEE Software* **22** (March/April 2005), pp. 34–43.

[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.

[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, 2003] B. STROUSTRUP, *The C++ Standard: Incorporating Technical Corrigendum No. 1*, 2nd ed., John Wiley and Sons, New York, 2003.

[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.

[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 13 章　实现工作流

学习目标

通过本章学习，读者应能：

- 执行实现工作流。
- 执行黑盒测试、玻璃盒测试和基于非执行的单元测试。
- 执行集成测试、产品测试和验收测试。
- 认识到良好的编程实践和编程标准的必要性。

实现是将详细设计变成代码的过程。如果让一个人来做，此过程还比较容易理解。但是，当今现实生活中的多数产品过于庞大，以至于不可能只由一个程序员在给定的时限内完成。实现产品的通常是一个团队，团队成员同时实现不同组件，这种方式称为*多方编程*（programming-in-the-many）。有关多方编程的问题将在本章讨论。

13.1　选择编程语言

多数情况下，不会由开发人员选择用何种语言来实现软件。假如客户需要使用 Smalltalk 语言编写某个产品，虽然在开发团队看来 Smalltalk 完全不合适这个产品，但这样的看法跟客户毫不相关，开发团队的经理只有两个选择：用 Smalltalk 进行开发或者放弃。

相似地，如果产品必须在特定的计算机上实现，而该计算机上仅有的可用语言是汇编语言，在这种情况下也没有选择。如果没有其他语言可用，原因可能是使用的计算机尚未存在高级语言编译器，或者是经理并不准备花钱买个新的C++ 编译器，那么讨论选择编程语言的问题就变得毫无意义了。

更有趣的情形是：合同要求产品将用“最合适的”编程语言开发，要选什么语言呢？为了回答这个问题，考虑以下情景。QQQ 公司用 COBOL 编写产品已经 30 多年了。QQQ 公司的全部 200 名软件员工，从最年轻的程序员到软件副总裁，都是 COBOL 专家。地球上还有哪一种比 COBOL 更合适的编程语言呢？引进一种新的语言，比如 Java，意味着不得不雇用新的程序员，或者至少现有的职员不得不重新接受高强度的培训。如果投入了这么多财力和精力在 Java 的培训上，经理也很可能会决定今后的产品也必须用 Java 来写。然而，所有现存的 COBOL 产品仍然需要维护。于是必须有两类程序员，COBOL 维护人员和 Java 编程人员。糟糕的是，人们几乎总是认为维护人员的地位比开发人员低，因此那些 COBOL 程序员会产生明显的不满情绪。如果 Java 程序员短缺，这种不满情绪还会由于 Java 程序员通常比 COBOL 程序员有更高的薪水而加重。QQQ 公司虽然有顶尖的 COBOL 开发工具，还是必须购买 Java 编译器，同样合适的 Java CASE 工具也少不了，还要购买或租用额外的硬件，使新的软件可以运行。最重要的是，QQQ 公司已经积累了数百人年的 COBOL 经验，这种经验只能通过实践才能获得。例如，屏幕上出现某个含义模糊的错误消息时应该做什么，或者如何处理编译器的某些古怪特性。简而言之，对 QQQ 公司来说，看上去只有 COBOL 是“最合适的”编程语言，其他任何选择将是经济上的自杀，无论从投入的成本还是从加重员工的精神负担而导致代码质量下降的后果来看，都会是一样的效果。

然而，从 QQQ 公司最近负责的项目来说，最合适的编程语言可能确实是别的语言，而不是 COBOL，虽然 COBOL 是世界上使用最广泛的语言（请参见备忘录 13.1），但 COBOL 只适合某一类软件产品，即数据处理应用，如果 QQQ 公司接到这类之外的软件需求，COBOL 将很快没有了吸引力。例如，如果 QQQ 公司想使用人工智能（Artificial Intelligence，AI）技术构建一个

基于知识的产品，则可采用像 Lisp 这样的 AI 语言。因为 COBOL 完全不合适 AI 方面的应用。如果要建立大规模通信软件，QQQ 公司可能需要通过卫星连接世界各地的上百个分支机构，那么事实将证明 Java 这样的语言比 COBOL 更为合适。如果 QQQ 公司将进入系统软件业务，开发诸如操作系统、编译器和连接器这样的系统软件，那么 COBOL 毫无疑问是不合适的。再假设 QQQ 公司准备签个国防项目合同，那么公司的管理层会很快发现，COBOL 根本无法用于实时嵌入式软件。

通常可以通过成本－效益分析法（见 5.2 节）来决定使用哪种编程语言的问题，也就是说，管理者必须计算当前用 COBOL 实现的成本，以及使用 COBOL 后现在和将来的收益。这种计算要针对每个可选的语言重复进行，具有最大期望回报的语言，即估计效益和估计成本之差最大的那种语言，就是合适的实现语言。另一种确定选择哪种语言的办法是采用风险分析，对于考虑的所有语言，列出潜在的风险及解决它们的办法的清单，风险总和最小的语言将是要选择的语言。

备忘录 13.1

COBOL 语言编写的代码行数远远超过所有其他语言编写的代码行数的总和，COBOL 被最广泛使用的主要原因是 COBOL 是美国国防部（Department of Defense，DoD）的产品，COBOL 于 1960 年被 DoD 认可。从那以后，DoD 只购买配有 COBOL 编译器的硬件来运行数据处理应用［Sammet，1978］。DoD 直到现在都是世界上最大的计算机硬件购买者，在 20 世纪 60 年代，DoD 开发的很大一部分软件都是用于数据处理的。结果，几乎为每台计算机配备 COBOL 编译器成了强制要求。COBOL 的广泛可用性以及当时可选的语言一般只有汇编语言，这使得 COBOL 变成世界上最流行的编程语言。

像C++ 和 Java 这样的语言毫无疑问在新的应用中逐渐流行。然而，交付后的维护仍然是项主要的软件活动，而维护则是对现存的 COBOL 软件进行。简而言之，DoD 凭借它的第一个主流编程语言 COBOL 在世界软件行业烙下了印记。

促使 COBOL 流行的另一个原因是它是实现数据处理产品的最合适语言，尤其在涉及钱款的时候，COBOL 是首选语言。财务账本要保持收支平衡，这样就不允许舍入误差混进里面，因此，必须用整数算法完成全部的计算。COBOL 支持大数字（即几十亿美元）的整数运算。另外，COBOL 也可以处理非常小的数字，如比美分更小的单位。银行规定利息最少计算从美分开始的小数点后四位，而 COBOL 可以轻易的完成这个运算。最后，COBOL 是第 3 代语言（或高级语言）中具有最好的格式化、排序以及报表生成设施的语言。所有这些原因使 COBOL 成为开发数据处理产品的优秀语言。

如 8.10.4 节所提到的那样，COBOL 语言的当前标准是面向对象的。这个 OO-COBOL 标准毫无疑问会进一步促使 COBOL 流行。

现在，软件公司在使用面向对象的语言，不管是哪一种面向对象语言，开发新的软件都会承受压力。随之产生的问题是：哪种面向对象语言比较合适？20 年前，仅仅只有一种选择，即 Smalltalk。然而今天，最广泛应用的面向对象编程语言是C++［Borland，2002］，其次是Java。有多方面的原因促使C++ 流行，C++ 编译器的广泛可用性是其中之一。实际上，某些C++ 编译器只是简单的将C++ 源代码翻译成为 C，然后调用 C 编译器，因此，任何带有 C 编译器的计算机都可以处理C++ 代码。

C++ 流行的真正原因是它与 C 具有明显的相似性。令人遗憾的是许多管理者把C++ 看作 C 的一个扩展集，从而得出任何了解 C 的程序员能够迅速的掌握额外部分的结论。若只是从句法的观点看，C++ 的确是 C 的一个扩展集，毕竟几乎任何 C 程序可以采用C++ 编译器来编译。然而，从观念上看，C++ 与 C 是完全不同，C 是个经典范型的产品，而C++ 是面向对象范型的产品。只有在采用了面向对象技术，并且产品是由对象和类而不是函数来组织的时候，C++ 的使

用才有意义。

因此，在组织采用C++之前，相关软件专业人员经过面向对象范型方面的培训是非常重要，尤其是本书第7章的知识要被灌输。除非所有涉及人员，尤其是管理者清楚了解了面向对象范型是一种不同的软件开发方法以及其与经典范型的区别所在，否则经典范型将被继续使用，只不过是用C++编写而不是C。当组织对从C转换到C++带来的结果感到失望时，一个主要的因素是相关人员缺乏面向对象范型的培训。

假定组织决定采用Java，在这种情况下，从经典范型渐变到面向对象范型是不可能的。Java是种纯面向对象程序语言，不支持传统范型的函数和过程。不像C++这样的混合面向对象语言，Java程序员从开始就必须使用面向对象范型（而且只能使用面向对象范型）。由于突然需要从一个范型转到另一个范型，与转变到像C++或OO-COBOL这种混合型的面向对象语言相比，采用Java（或其他纯粹的面向对象语言，如Smalltalk）所需的教育和培训显得更为重要。

在决定实现语言之后，接下来的问题将是如何采用软件工程原理获得更高质量的代码。

13.2　良好的编程实践

关于编程风格，许多好的建议都和特定的语言相关。例如，对于使用COBOL88的入口或Lisp圆括号的建议，正在用Java实现产品的程序员可能对此并不感兴趣。因此，这里为Java和C++这样的面向对象语言给出一些与语言无关的良好编程实践（good programming practice）的建议。

13.2.1　使用一致和有意义的变量名

如第1章中所述，平均至少2/3的软件预算用于交付后的维护，这意味着开发某个代码制品的程序员只是众多工作在该代码制品上的人之一，且是第一人。程序员给出只对他自己有意义的变量名是不够的。在软件工程的领域，术语有意义的变量名（meaningful variable name）意思是“从将来维护程序员的角度来看是有意义的”。这个观点在备忘录13.2中有更详细的说明。

> **备忘录13.2**
>
> 在20世纪70年代后期，南非约翰内斯堡有个小型软件公司，由两个编程团队组成。团队A由来自莫桑比克的人组成，他们拥有葡萄牙血统，母语是葡萄牙语。他们的代码写得很好，变量名是有意义的，但不幸的是仅对说葡萄牙语的人有意义。团队B由以色列移民组成，母语是希伯来语。他们的代码写得一样好，他们选择的变量名同样是有意义的，但是仅对说希伯来语的人有意义。
>
> 一天，团队A和他们的负责人一同辞职了。团队B完全无法维护团队A曾经编写的任何优秀代码，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，对于语言能力仅限于希伯来语和英语的以色列人是完全不可理解的。公司老板无法雇用足够的会说葡萄牙语的程序员代替团队A。公司很快在大量不满客户的诉讼压力下破产了，因为这些客户的代码基本上是不可维护的了。
>
> 这种情况很容易避免，公司的领导应当在一开始就坚持用英语命名全部的变量名，而英语是每个南非计算机专业人员都理解的语言，于是变量名对每个维护人员就都是有意义的了。

除了使用有意义的变量名之外，选用一致的变量名（consistent variable name）也同等重要。例如，以下4个变量声明在同一个代码制品中：`averageFreq`、`frequencyMaximum`、`minFr`和`frqncyTotl`。试图理解这段代码的维护程序员必须知道是否`freq`、`frequency`、`fr`和`frqncy`指同一种东西。如果是，那么应当使用唯一的单词，建议使用`frequency`，当然`freq`或`frqncy`也勉强可以接受，`fr`则不行。但是如果某个或多个变量名表示的是不同的量，则应

使用一个完全不同的名字，如用 **rate** 表示。相反，同一个概念不要使用两个不同的名字，例如，不应当在同一个程序中同时使用 average 和 mean。

一致性的第二个方面是构成变量名的组件顺序。举例来说，如果某个变量命名为 **frequencyMaximum**，那么变量名 **minimumFrequency** 将会使人迷惑，应当采用 **frequencyMinimum**。为了使将来的维护程序员对代码有清晰而没有歧义的理解，前面所列的 4 个变量应当分别命名为 **frequencyAverage**、**frequencyMaximum**、**frequencyMinimum** 和 **frequencyTotal**。同样，**frequency** 也可以放在 4 个变量名的末尾，产生变量名 **averageFrequency**、**maximumFrequency**、**minimumFrequency** 和 **totalFrequency**。显然两组中的哪一个被选择没有关系，重要的是变量名都来自一组或另一组。

许多不同的命名约定已经被提出来以使代码更容易理解，它的思想是变量名应该包括类型信息。例如，**ptrChTmp** 可能表示一个临时变量（**Tmp**）用来指向字符（**Ch**）的指针（**ptr**）。采用这种方案最著名的是匈牙利命名法［Klunder，1988］（如果你想知道为什么叫“匈牙利”，请参见备忘录 13.3）。这类方案的缺点之一是，当参与者不能拼读出变量名时，代码审查（参见 13.13 节）的效果将降低，逐个字母地读出变量名尤其痛苦。

备忘录 13.3

术语匈牙利命名法（Hungarian Naming Conventions）*有两种解释。第一，这些约定是由出生于匈牙利的* Charles Simonyi *提出的；第二，人们普遍承认，具有符合该约定的变量名的程序阅读起来将像阅读匈牙利文一样容易。而且，使用它们的组织*（*如* Microsoft）*声称，对于那些有匈牙利命名法经验的人来说，它们增强了代码的可读性。*

13.2.2　自文档化代码的问题

当问到为何他们的代码没有包含注释时，程序员常常会自豪地说：“我写的是自文档化代码（self-documenting code）”。意思是他们的变量名经过认真选择，代码编写得十分精巧，以至于没有添加注释的必要。自文档化代码的确存在，但是非常稀少。通常的情形是程序员在编写代码制品时的确认真考虑代码中每个名词的细微差别。可以想象，程序员在所有代码制品使用相同的风格，即使经过 5 年时间后，最初编写代码的程序员也仍然对他当年所写代码的方方面面了然于胸。很遗憾，这没用。最关键的是其他必须阅读此代码的程序员是否能容易地理解并且不产生歧义，从软件质量保证小组成员到交付后维护程序员。当把交付后维护任务交给没有经验的程序员并且没有人认真指导他们时，这个问题变得更加尖锐。对于一个有经验的程序员来说未加注释的代码制品也仅仅部分可懂，当维护程序员缺乏经验时情形就更糟糕了。

要理解为何会产生这类问题，考虑变量 **xCoordinateOfPositionOfRobotArm**，无论从哪个角度来看，这个变量名毫无疑问是自文档化的，但很少程序员愿意使用 31 个字符的变量名，尤其该变量名经常被使用的话。取而代之的是使用一个短名字，如 **xCoord**。这样做的理由是，如果整个代码制品负责一个机器人手臂的移动，**xCoord** 只能表示机器人手臂位置的 x 坐标。尽管这样的说法在开发过程这个特定的环境中说得通，在交付后维护中就不一定正确了。维护人员对该产品可能没有足够的整体认识，或者可能没有必要的注释来帮助理解该代码制品，从而不知道在此代码制品内 **xCoord** 指的是机器人手臂的坐标。避免这类问题的方法是坚持在代码制品的开始，即在*序言注释*（prologue comments）中对每个变量名做解释。如果遵循这条规则，维护程序员将很快理解变量 **xCoord** 用来表示机器人手臂位置的 x 坐标。

在每个代码段制品中序言注释是必须要有的，每个代码制品的顶部必须提供图 13-1 中列出最基本信息。

即使代制品写的很清晰，也没有理由指望谁去阅读每一行代码，理解该代码制品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 小组的成员或要修改某一特定代码

制品的维护程序员才要阅读该代码制品的每一行。

除了序言注释外，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。专家建议，行内注释的使用场合应该仅仅是当代码编写的方式并不明显，或者使用了该语言中某些难理解的方面的时候。相反，含糊不清的代码应当以清晰的方式重新编写。行内注释是帮助维护程序员的一种手段，不应当助长拙劣的编程实践或为其寻找借口。

```
代码制品名字
关于代码制品做什么的简单描述
程序员名字
代码制品编写的日期
代码制品被认可的日期
认可代码制品人的名字
代码制品的参数
代码制品中每个变量的名字列表，最好按字母顺序排列，并有用法的简短描述
被代码制品访问的文件名
被代码制品修改的文件名
输入－输出（如果有的话）
错误处理能力
包含测试数据的文件名（以后用于回归测试）
对代码制品所作修改的列表，修改日期及认可人
任何已知错误
```

图 13-1　一个代码制品的最小的序言注释

13.2.3　使用参数

很少有真正的常量，也就是说，它的值“永远不会”改变。例如，为了得到反映珍珠港精确位置的更准确的地理数据，要在导航系统中加进夏威夷珍珠港的纬度和经度，而使用卫星照片会迫使潜艇导航系统根据变化做出调整。另一个例子，销售税不是真正的常量，立法机关可能不时地调整销售税率，假定销售税率当前是6.0%，如果数值6.0被硬编码到了某产品的许多代码制品中，那么改变该产品将是一项浩大的工程，可能导致忽视“常量”6.0的一个或两个实例，并可能错误地改变一个不相关的6.0。一个更好的解决方案是如下面的C++声明：

```
const float SALES_TAX_RATE = 6.0;
```

或者在 Java 中：

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

那么，无论哪里需要销售税率的值，应当使用常量 **SALES_TAX_RATE** 而不是数值 **6.0**。如果销售税率改变了，只需采用编辑器替换包含 **SALES_TAX_RATE** 的值的那行代码。比这还要好的办法是，运行开始时从参数文件中读入销售税率的值。所有这样明显的常量都应该当作参数处理，如果某个值由于任何原因需要改变，那这个改变就能够快速和有效地实现。

13.2.4　为增加可读性的代码编排

要让一个代码制品易于阅读相当简单。例如，一行中不应当出现多个语句，即使许多编程语言允许这样。缩进也许是增加代码可读性最重要技术，设想一下如果没有使用缩进来帮助代码理解，第7章中的代码例子是多么难于阅读。在C++或Java中，缩进可以用来匹配相应的{…}对，还能显示哪些语句属于给定的代码块。事实上，正确的缩进太重要了，以至于不能由人工完成。如5.6节所描述的那样，而应该使用CASE工具确保缩进的正确性。

另一个有用的帮手是空行。方法之间应当用空行隔开，此外，用空行隔开大的代码块常常

是有帮助的。额外的“空白区域”让代码更容易阅读，因此也更容易理解。

13.2.5　嵌套的 if 语句

考虑下面的例子，一张地图包括两个方块，如图13-2所示。要求编写代码确定地球表面的一个点是否位于 **mapSquare1** 或 **mapSquare2** 中，或者根本不在图中。图 13-3 的解决方案格式太差，很难理解。进行适当格式编排后的版本如图 13-4，尽管如此，**if-if** 和 **if-else-if** 结构的组合也复杂到难于检查代码片段是否正确。图 13-5 对此进行了修正。

遇到包含 **if-if** 结构的复杂代码时，一种简化的方法是采用下面的结构：if-if 组合

if <条件1>

if <条件2>

与单个条件等价

if <条件1> **and** <条件2>

假定即使 < 条件 1 > 不满足，也定义 < 条件 2 >。例如，< 条件 1 > 检测某个指针不是空的，如果不是空的，那么 < 条件 2 > 可以使用那个指针。（Java 或C++ 中不存在这个问题，&& 操作定义为，如果 < 条件 1 > 为假，则不计算 < 条件 2 >。）

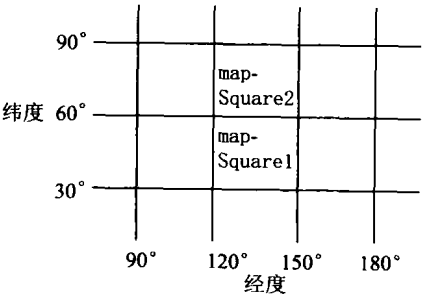


图 13-2　地图坐标

if-if 结构的另一个问题是嵌套过深导致代码难以阅读。从经验来看，嵌套 **if** 语句超过 3 层是糟糕的编程实践，应当避免。

```
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 13-3　难读的嵌套 **if** 语句格式

```
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 13-4　格式良好但晦涩的 **if** 嵌套结构语句

```
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 13-5　可接受的嵌套 **if** 语句

13.3　编码标准

编码标准（coding standard）是福也是祸。7.2.1 节指出过，带有偶然性内聚的模块通常是应用某些规则的结果，如“每个模块将由 35～50 条可执行语句组成”。与这种教条的风格不同，更好的形式是：“程序员在构造一个少于 35 或多于 50 条可执行语句的模块之前，应当征求管理者的意见”。问题的关键是没有哪种编码标准适用于所有可能的情形。

像上面这样强加的编码标准常常被忽视，如前文所述，一条有用的经验是，if 语句嵌套深度不应超过 3 层。如果向程序员展示因为 if 嵌套过深而导致代码不可读的例子，他们很可能会遵守这样的规则。但是，程序员不可能恪守一长串强加给他们、不经讨论和解释的规则。进一步说，这样的标准可能导致程序员和管理者之间产生摩擦。

此外，一套编码标准除非可以由机器检验，否则会浪费 SQA 小组大量的时间，或者简单的被程序员和 SQA 小组忽视。另一方面，考虑下面的规则：

- if 语句嵌套不应当超过 3 层的深度，除非得到团队领导的特许。
- 模块应当由 30～50 条语句组成，除非得到团队领导的特许。
- 应当避免使用 goto 语句，然而，在得到团队领导特许的情况下，向前的 goto 语句可以用于错误处理。

假如建立某种机制，用于捕捉有关的允许偏离标准的数据，这样的规则就可以由机器检验。

编码标准的目标是使维护更加容易，然而，如果使用一套标准的结果是使软件开发者的开发工作更加困难，那么，该标准应当修改，即使处在项目的中期。对于严格的编码标准将达不到预期目标。如果程序员必须在这样一个框架下开发软件，软件产品的质量将不可避免受损失。另一方面，如前面列出的关于 if 语句嵌套、模块大小以及 goto 语句的那些标准，与偏离标准的检验机制结合起来，将可以提高软件的质量，毕竟这是软件工程的一个主要目标。

13.4　代码复用

第 8 章中对复用进行过详细的介绍，事实上，有关复用的材料几乎出现在本书的每个章节，因为软件过程的所有流程的制品都可以复用，包括说明、合同、计划、设计和代码制品等部分。这就是为什么复用的材料放在本书的第一部分，而不是某个特定的流程中。特别的，不把复用的材料放在这一章的一个很重要原因是为了强调如下的事实：即使代码的复用是迄今为止最普遍的复用形式，但也不光代码可以复用。

13.5　集成

考虑图 13-6 中描绘的产品，产品集成（integration）的一种方法是单独编写和测试每个代码制品，然后连接 13 个代码制品，最后对产品的整体进行测试。使用这种方法存在两个困难。第一，考虑制品 a，它不能依靠自身进行测试，因为它调用了制品 b、c 和 d。因此，要测试制品 a，制品 b、c、d 必须作为存根（stub）编码。存根的最简形式就是一个空的制品。相对有效的一种存根是打印一条消息，例如“`displayRadarPattern called`”。较好的存根应该能够返回与预先计划好的测试用例相吻合的值。

现在，考虑制品 h。测试制品 h 需要一个能调用它一次或多次的代码制品，即驱动器（driver），如



图 13-6　一种典型的层次图

果可能，要检查被测试代码制品的返回值。相似的，测试制品 d 需要一个驱动与两个存根。因此，伴随单独实现和集成，第一个问题出现了：需要花大量的精力在构建存根与驱动上，而这些制品在单元测试完成后都会被抛弃。

在实现完成之后，集成工作开始之前，出现的第二个更为重要的困难是缺乏错误隔离的手段。如果产品作为整体测试，而在某个特定的测试用例下产品失败了，这个错误可能存在于 13 个代码制品或 13 个接口的任何地方。在具有 103 个代码制品和 108 个接口的大型软件中，那么可能隐藏错误的地方不会少于 211 个。

解决这两个困难的方案是将单元测试与集成测试结合起来。

13.5.1　自顶向下的集成

在自顶向下集成（top-down integration）中，若代码制品 **mAbove** 发送一个消息给代码制品 **mBelow**，那么代码制品 **mAbove** 比代码制品 **mBelow** 要先被实现和集成。假设图 13-6 中的产品是自顶向下实现与集成的，自顶向下的一种可能顺序是 a、b、c、d、e、f、g、h、i、j、k、l 和 m。首先，编写代码制品 a，并用存根 b、c、d 测试 a。接下来存根 b 扩展为代码制品 b，并与代码制品 a 连接，同时用存根制品 e 对 b 进行测试。实现与集成按照这种方式进行下去，直到所有代码制品都已经集成到产品中。自顶向下的另一种可能顺序是 a、b、e、h、c、d、f、i、g、j、k、l 和 m。在这种顺序下，集成的部分工作可以并行进行，方式如下：在 a 编码和测试结束后，一个程序员可以利用代码制品 a 来实现与集成 b、e、h，而另一个程序员可以利用代码制品 a 并行地工作于 c、d、f 和 i。一旦 d 与 f 完成，第三个程序员可以开始对 g、j、k、l 和 m 进行工作。

假定代码制品 a 在某个特定的测试用例上执行是正确的，而当 b 编码完成并集成到产品中后（此时产品由代码制品 a 和 b 连接而成），提交同样的测试数据时，测试结果失败了。错误可能在两个地方之一：代码制品 b 或者代码制品 a 与 b 之间的接口。一般情况下，当代码制品 **mNew** 加入已经成功通过测试的产品后，以前成功的测试用例失败了，那么错误几乎可以肯定存在于 **mNew** 或者 **mNew** 与产品的其他部分之间的接口中。这样，自顶向下集成支持错误隔离。

自顶向下集成的另一个优势是设计错误的早期显现。软件的代码制品可以分为两组：逻辑制品和操作制品。*逻辑制品*（logic artifacts）本质上表现为软件产品控制方面的决策流。逻辑制品通常是那些在连通图中离根较近的代码制品。例如，在图 13-6 中，认为代码制品 a、b、c、d 或许还有 g、j 是逻辑代制品是合理的。另一方面是软件产品中进行实际操作的*操作制品*（operational artifacts）。例如，操作制品可能命名为 **getLineFromTerminal** 或者 **measureTemperatureOfReactorCore**。操作制品一般位于连通图中离叶子较近的较低层。在图 13-6 中，制品 e、f、h、i、k、l 和 m 是操作制品。

在对操作制品进行编码和测试前，对逻辑制品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错误较早的显现。如果整个产品完成后才发现一个严重错误，那么产品的大部分代码都要重写，特别是包含控制流程的逻辑制品。许多操作制品在产品的重建过程中可以被复用，例如，代码制品 **getLineFromTerminal** 或代码制品 **measureTemperatureOfReactorCore**，不管产品如何重建都是需要的，尽管操作制品与其他代码制品之间的连接方式由于不必要的工作可能需要发生变化。因此，设计错误越早被发现，修正产品错误并使软件开发回到计划中的花费就越小，而且时间也就越短。采用自顶向下的策略进行制品实现与集成，确保了逻辑制品在操作制品之前实现与集成，因为在连通图中，逻辑制品几乎总是操作制品的祖先，这是自顶向下集成的一个主要优势。

然而，自顶向下集成的也有一个缺点：可复用代码制品可能测试不充分，如下面将要讨论的那样。复用那些误认为已经过充分测试的代码制品，很可能比重写代码更没效率，因为当产品失败时，那些复用代码制品正确的假象会造成错误的结论。测试者可能不会怀疑复用代码制品没有充分测试，而是考虑错误隐藏在别的地方，由此造成精力的浪费。

逻辑制品很可能只适用于某些特定的问题，因而在另一种环境中不可用。然而，操作制品，尤其是具有信息性内聚（见7.2.7节）时，在将来的产品中也许可以复用，因而需要彻底的测试。遗憾的是，操作制品通常是处于交互流程图的较低层代码制品。因此不能像上层制品那样测试得频繁。举例来说，如果存在184个代码制品，根制品可能会被测试184次，而被集成到产品中的最后一个制品可能只被测试1次。操作制品的测试不充分，自顶向下的集成方法将导致复用的风险。

如果产品设计良好，那么情况可能会更糟。事实上，产品设计的越好，制品可能测试得越不彻底。为了理解这一点，考虑制品`computeSquareRoot`。这个制品需要2个参数：浮点型数x，它的平方根将被计算；`errorFlag`，如果x是负数，将置为true。进一步假定，`computeSquareRoot`将被制品a3调用，并且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`将永远不会被调用。因此，x为负值的情况永远不会测试到，也就无从观察这种情况下该制品是否能正确执行。这类在调用制品之前进行安全检查的设计方式一般称为保守编程（defensive programming）。保守编程的后果是，如果自顶向下的集成，次要的操作制品不可能彻底测试。与保守编程相对的另一种方法是职责驱动设计（见1.9节），在这种方法中，必要的安全性检查设置在被调用的制品中，而不是调用者。还有一种办法是在被调用的制品中使用断言（见6.5.3节）。

13.5.2　自底向上的集成

在自底向上集成（bottom-up integration）中，如果制品`mAbove`给制品`mBelow`发送了一个消息，则制品`mBelow`比制品`mAbove`先进行实现和集成。在图13-6中，一种可能的自底向上的顺序是l、m、h、i、j、k、e、f、g、b、c、d、a。要让一个团队编写产品的话，下面的自底向上的顺序更好：h、e、b交给一个程序员，i、f、c交给另一个。第3个程序员从l、m、j、k、g开始，然后实现d，并将他的工作和第2个程序员集成。最后，b、c、d成功的集成以后，就可以实现和集成a了。

于是，当采用自底向上策略时，操作制品可以得到充分的测试。另外，测试通过驱动的协助完成，而不是通过错误保护、保守编程的制品来完成。尽管自底向上的集成解决了自顶向下集成的主要难题，并且与自顶向下的集成一样具有错误隔离的优势，但遗憾的是，它还是有自己的难题。特别是重大设计错误要到实现工作流后期才发现，要错制品最后集成，因此，如果有重大设计错误，将需要花费巨大的精力来重新设计和编写大部分的产品代码。

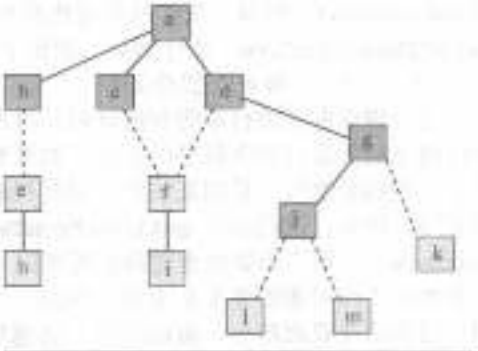


图13-7　采用三明治集成图13-6的产品

所以，自顶向下与自底向上的集成各有其优劣，产品开发的解决方案就是结合这两种策略，扬长避短，于是就有了三明治集成理念。

13.5.3　三明治集成

考虑图13-7所示的结构图，逻辑制品是a、b、c、d、g、j这6个代码制品，因此应该采用自顶向下集成。操作制品是e、f、h、i、k、l、m这7个代码制品，应该采用自底向上集成。因为无论是自顶向下还是自底向上集成都不能满足所有制品，所以将它们分开处理。6个逻辑制品采用自顶向

下集成，则任何重大设计错误能够及早发现。7个操作制品采用自底向上集成，将不到保守编程制品的保护，从而得到充分的测试，因此可以在其他产品中放心的复用。所有制品都正确的集成后，再一个个的测试两组制品之间的接口，这整个过程中都有错误隔离，被称为三明治集成（sandwich integration）（参见备忘录13.4）。

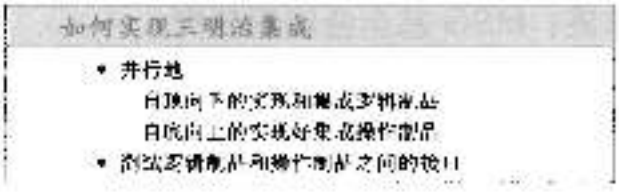
> **备忘录13.4**
>
> 术语三明治集成［Myers，1979］基于把逻辑制品和操作制品看作三明治的顶层与底层，而连接它们的接口就像三明治的馅，如图13-7所示。

图13-8总结了三明治集成的优缺点，以及本章前面讨论过的其他集成技术。

方 法	优 势	缺 点
实现后再集成（13.5节）		没有错误隔离，主要设计错误出现较迟，潜在复用代码制品测试不充分
自底向下集成（13.5.1节）	错误隔离 主要设计错误早期暴露	潜在复用代码制品测试不充分
自底向上集成（13.5.2节）	错误隔离 潜在复用代码制品测试充分	主要设计错误出现较迟
三明治集成（13.5.3节）	错误隔离 主要设计错误早期暴露 潜在复用代码制品测试充分	

图13-8 本章及各节描述集成方法小节

“如何实现三明治集成”总结了三明治集成的方法。



13.5.4 集成技术

对象既可以自底向上的集成，也可以自顶向下的集成。如果选用自顶向下的集成方法，每个方法都可以使用存根。

如果用自底向上的集成，那些不发送消息给其他对象的对象首先实现和集成，然后实现和集成那些发送消息的对象。如此下去，直到实现和集成完产品中的所有对象。（此过程如果包含递归则需要修改。）

因为同时支持自顶向下和自底向上，所以三明治集成也可以使用。如果产品是用C++这样的混合型面向对象语言实现的，通常类都是操作制品，因此自底向上集成。许多不是类的制品都是逻辑制品，因此自顶向下的实现与集成。剩下的制品都是操作性的，因此自底向上的实现与集成。最后，所有的非对象制品都集成到对象中。

就算产品采用Java这样的纯面向对象语言来实现，类方法（有时也称为静态方法（static methods）），如main以及实用程序方法通常与逻辑模块在结构上相似。因此，类方法也是自顶向下的实现，然后集成到其他其他对象中去。换句话说，实现和集成面向对象产品，也会用到三明治集成的变种。

13.5.5　集成管理

集成阶段发现的管理问题是，代码制品不能简单的连接在一起。例如，程序员1编写了对象o1，程序员2编写了对象o2。在程序员1使用的设计文档中，对象o1发送消息给对象o2，传递4个变量，但是程序员2使用的设计文档明确指出只有3个变量传给o2。如果没有通知开发小组的全体成员，仅仅对设计文档的一份拷贝进行修改，就会出现这样的问题。两个程序员都认为自己是正确的，谁也不愿意妥协，因为做出让步的程序员必须重写产品的大部分代码。

要解决这些类似的不兼容的问题，整个集成过程必须由SQA小组实行，并且在其他阶段的测试中，如果集成测试执行不成功，那么SQA小组要负主要责任。因此，SQA小组会确保测试彻底的实行。SQA小组的负责人要对集成测试的各方面负责，他必须决定哪些制品采用自顶向下的实现和集成，哪些制品采用自底向上的实现和集成，把集成测试任务分配给正确的人选。SQA小组在软件项目管理计划中制定了集成测试计划，同样要负责实行该计划。

集成过程的最后阶段是，所有的代码制品都已经测试过并合并成为单一的产品。

13.6　实现工作流

实现工作流（implementation workflow）的目标是用所选的语言实行目标软件产品。更准确地说，如12.4节所解释，大型的软件产品被分解为一些小的子系统，然后由编码团队并行的实现，同样，这些子系统又由*代码制品*（code artifact）组成。

一旦完成代码制品的编码，程序员就对其进行测试，称为*单元测试*（unit testing）。一旦程序员认为代码制品是正确的，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测试是测试流的一部分，本章后面13.19～13.21节将讨论。

13.7　实现工作流：MSG基金会案例研究

MSG基金会产品的C++和Java版本的完整实现可以从www.mhhe.com/schach下载。程序员在其中加进了各种注释来帮助交付后维护人员。

接下来介绍实现工作流中的测试。

13.8　测试工作流：实现

实现工作流要执行许多不同种类的测试，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，代码要经历两种类型的测试：程序员在开发代码段时进行非正规的单元测试，当程序员认为代码段的功能正常以后，由SQA小组进行系统单元测试。13.9～13.13节描述这些系统测试方法，依次介绍两类基本的系统测试：*基于非执行的测试*，代码段由一个小组评审；*基于执行的测试*，对照测试用例运行代码段。现在介绍选择测试用例的技术。

13.9　测试用例的选择

使用随意的测试数据测试代码制品是最差的方法。测试者坐在键盘前，只要制品要求输入，测试者以任意数据响应。将会看到，这样只能测试所有可能的测试用例的极小部分，时间不允许测试更多的数据，因为它们轻易就可以达到比10^{100}还多。能够运行的少数测试用例（可能在1 000这个量级上）非常宝贵，不能将它浪费在随意的数据上。更糟糕的是，如果机器对同样的输入数据响应多次，就会浪费更多的测试用例，显然*测试用例选择*（test case selection）必须系统化进行。

13.9.1　规格说明测试与代码测试

单元测试的测试数据可以采用两种基本的方法进行系统的构建。第一种是规格说明测试，这项技术也叫做黑盒、行为、数据驱动、功能及输入/输出驱动测试。这种方法忽视代码本身，拟定测试用例时使用的仅有的信息是规格说明文档。代码测试是另一个极端，在选择测试用例时忽视规格说明文档。这项技术的其他名称有玻璃盒（glass-box）、白盒（white-box）、结构（structural）、逻辑驱动（logic-driven）以及面向路径的测试（path-orientedtesting）。关于为什么有这么多不同的术语的解释，见备忘录 13.5。

现在考虑这两项技术的可行性，从规格说明测试开始。

备忘录 13.5

可以理解，对相同的测试概念有这么多不同的名称会产生疑问。软件工程中常发生这样的事情，不同的研究者独立的发现了相同的概念，每个人都发明了自己的术语。当软件工程界认识到它们是同一概念的不同称呼时，已经太晚了，不同的名称已经蔓延到软件工程的词汇表中来了。

在这本书中，采用术语“黑盒测试”和“玻璃盒测试”。这些术语很有描述性，当测试规格说明时，把代码当作一个完全不透明的黑盒。相反，当测试代码时，需要看到盒子内部，因此使用术语“玻璃盒测试”。本书避免使用术语“白盒测试”，因为它多少有些误解，毕竟，一个涂成白色的盒子和一个涂成黑色的盒子一样是不透明的。

13.9.2　规格说明测试的可行性

考虑以下例子。假定某个数据处理产品的规格说明要求，必须包括 5 种佣金和 7 种折扣。仅测试佣金和折扣的每种可能组合需要 35 个测试用例。如果说佣金和折扣在两个完全独立的制品中作计算，因此可以独立的测试这对黑盒测试是没有用的，因为在黑盒测试中，产品将被当作黑盒看待，因此内部结构也是完全不相关的。

这个例子只包含两个因素，佣金和折扣，它们分别取 5 个和 7 个值。任何实际的产品即使没有上千也有上百个不同的因素，即使只有 20 个因素，每个因素只取 4 个不同的值，也必须测试一共 4^{20} 或 1.1×10^{12} 个不同的测试用例。

要了解超过 1 万亿个测试用例意味着什么，可以考虑一下对它们进行全部测试要花多少时间。如果可以找到一个程序员团队，平均以每 30 秒一个的速率生成、运行和检查测试用例，那么将花费超过 100 万年来完成该产品的测试。

因此，由于组合爆炸，彻底的规格说明测试在实践中是不可行的，有太多的测试用例需要考虑，现在分析代码测试。

13.9.3　代码测试的可行性

最常见形式的代码测试要求代码制品中的每条路径至少执行一遍。

- 要了解这样为什么不可行，考虑图 13-9 的代码片段，对应的流程图如图 13-10 所示。尽管这个流程图看起来微不足道，它还是包含了 1 012 条不同路径。有 5 条可能的路径穿过中间的 6 个打阴影的框，因此所有经过流程图的路径的数目是

$$5^1+5^2+5^3+\cdots+5^{18}=\frac{5\times(5^{18}-1)}{(5-1)}=4.77\times10^{12}$$

如果包含一个循环的流程图也有这么多条路径，可以推断出与此代码段具有相当路径数目大小和复杂度的路径总数，更不要说包含多个循环了。简而言之，可能包含的路径的巨大数目导致彻底的代码测试同彻底的规格说明测试一样不可行。

```
read (kmax)                          // kmax 是1~18之间的一个整数
for (k = 0; k < kmax; k++) do
{
   read (myChar)                     // myChar 是字母 A, B 或 C
   switch (myChar)
   {
     case 'A':
       blockA;
       if (cond1) blockC;
       break;
     case 'B':
       blockB;
       if (cond2) blockC;
       break;
     case 'C':
       blockC;
       break;
   }
   blockD;
}
```

图 13-9　代码片段

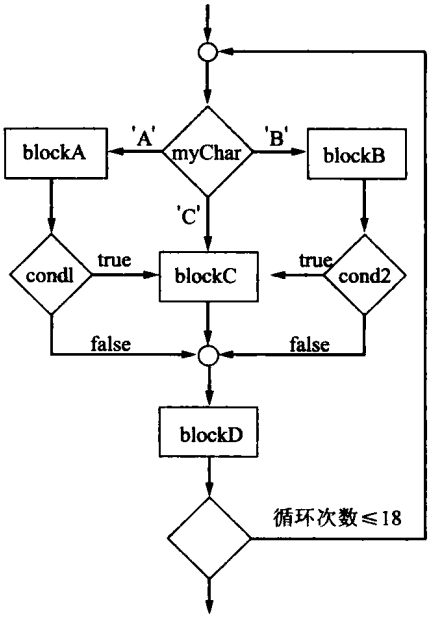


图 13-10　具有超过 10^{12} 种路径的流程图

- 进一步说，代码测试要求测试者试验每条路径，有可能试验了每条路径却不能发现产品的每个错误，也就是说，代码测试是不可靠的。要了解这一点，考虑图 13-11 中的代码片段［Myers，1976］。这个片段用于测试 3 个整数 **x**、**y**、**z** 是否相等，使用了完全不合理的假定，即如果 3 个数的平均值和第 1 个数相等，那么 3 个数相等。图 13-11 给出了 2 个测试用例。第一个测试用例中，3 个数的平均数是 6/3，即 2，不等于 1。于是产品正确地告诉测试者 x、y、z 不相等。第二个测试用例，x、y、z 都等于 2，于是产品计算出它们的平均值是 2，等于 x，从而得出正确的结论，即 3 个数相等。于是，通过产品的 2 条路径都试验到了，但错误没有发现。当然，如果用到了像 x = 2、y = 1、z = 3 这样的测试数据，错误就会显现。

```
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 13-11　一个判断 3 个数是否相等的不正确的代码片断和 2 个测试用例

```
if (d == 0)
  zeroDivisionRoutine ();
else
  x = n/d;
```

a）

```
x = n/d;
```

b）

图 13-12　两个计算商的代码片断

- 路径测试的第 3 个难题是，仅当路径存在时才能测试。考虑图 13-12a 中的代码片段。显然，对应用例 d = 0 和 d≠0 的 2 条路径将被测试。接下来，考虑图 13-12b 中的单一语句。现在只有 1 条路径，并且测试这条路径检测不到错误。事实上，程序员在他的代码中忽略了检查是否 d = 0，很可能是不知道潜在的危险，从而在他的测试数据中没有包括 d = 0 的情况。这个问题引出一个附加的观点：应该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。

这些例子最后说明，“试验产品中的所有路径”的准则是不可靠的，因为产品中存在给定的路径，试验某些数据能发现错误，其他数据对同一路径试验却不能。尽管如此，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排除在外。

由于组合爆炸，纯粹的规格说明测试或纯粹的代码测试都不可行。我们需要退一步，使用能尽可能多地找出错误的技术，同时承认没有哪种方法能保证发现所有错误。一种合理的方法是先用黑盒测试用例（测试规格说明），再开发额外的玻璃盒测试用例（测试代码）。

13.10　黑盒单元测试技术

彻底的黑盒测试一般要求数 10 亿的测试用例，测试的艺术在于设计一个较小的容易管理的测试用例集，使发现错误的几率最大化，同时使多个测试用例发现同一个错误而浪费测试用例的几率最小化，所选的每个测试用例必须能发现前面没有检出的错误。一种这样的黑盒测试技术是等价测试结合边界值分析。

13.10.1　等价测试和边界值分析

假定某数据库产品的规格说明指出，产品必须能处理 1 ~ 16 383（$2^{14}-1$）的任意条记录。如果产品能处理 34 条和 14 870 条记录，那么有很大几率 8 252 条记录也可以正常工作。事实上，选择 1 ~ 16 383 之间的任何多条记录作为测试用例，发现错误的几率（如果存在）很可能是一样的。反过来，如果产品对 1 ~ 16 383 之间的某一个测试用例正确工作，那么很可能对这个范围内的其他多条记录同样成立。1 ~ 16 383 构成了一个等价类（equivalence class），即一个测试用例集，其中的任何一个成员和其他测试用例同样好。更精确地说，该产品必须能处理的记录数目的范围定义了 3 个等价类：

等价类 1　小于 1 条记录。

等价类 2　从 1 到 16 383 条记录。

等价类 3　超过 16 383 条记录。

于是，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。来自等价类 2 的测试用例应该正确处理，而对来自等价类 1 和等价类 3 的测试用例应该打印错误消息。

成功的测试用例能检测到先前没有发现的错误。要使找到这样的错误的机会最大化，边界值分析（boundary value analysis）是一种高回报的技术。

经验表明，选择处于或接近等价类边界的测试用例时，发现错误的可能性增大。因此，测试数据库产品时，应该选择 7 个测试用例：

测试用例 1　0 条记录：等价类 1 的成员，临近边界值。

测试用例 2　1 条记录：边界值。

测试用例 3　2 条记录：临近边界值。

测试用例 4　723 条记录：等价类 2 的成员。

测试用例 5　16 382 条记录：临近边界值。

测试用例 6　16 383 条记录：边界值。

测试用例 7　16 384 条记录：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，另一个同样强大的技术是检查输出规格说明。例如，2006 年，美国税法规定允许对从个人薪水中被扣除的社会保险额进行减免，更准确的说是被减免的老年人、幸存者和残疾人的保险款（美国联邦社保基金，OASDI）最低是 0 美元，而最高是 5 840.40美元，后者对应 94 200 美元的总收入。因此，测试一个工资单产品时，测试工资中的社会保险扣减的用例应该包括正好能产生 0 和 5 840.40 美元的扣减的输入数据。此外，建立的测试数据应当能产生比 0 美元少和比 5 840.40 美元多的扣减。

一般来说，对输入或输出规格说明中列出的每个范围（R_1，R_2），应该选择 5 个测试用例，对应于小于 R_1、等于 R_1、大于 R_1但小于 R_2、等于 R_2和大于 R_2的值。如果指定了某一项必须是

一个特定集合的成员（例如，输入必须是一个字母），必须有两个等价类，一个是指定集合的成员，另一个不是。如果规格说明指定了一个精确的值（例如，响应必须以#号结尾），那么又有两个等价类，指定的值和其他值。

对于产生较小的测试数据集来说，结合等价类和边界值分析来测试输入规格说明和输出规格说明是一项有价值的技术，因为对于测试数据集中的数据，如果没有使用强有力的技术进行选取的话，它就不能揭示潜在的仍然没有发现的错误。

"如何实现等价测试"总结了等价测试的过程。

如何实现等价测试

- 对输入和输出规格说明
  - 对每个范围 (L, U)
    - 选择3个测试用例：小于L、等于L、大于L但小于U、等于U和大于U
  - 对每个集合S
    - 选择2个测试用例：S的成员和非S的成员
  - 对每个精确值V
    - 选择2个测试用例：V和其他任何值

13.10.2　功能测试

黑盒测试的一种简化形式是基于代码段的功能性而选择测试数据。功能测试（functional testing）[Howden，1987] 是对代码制品实现的方法进行测试，其对每个方法单独设计测试数据。如今，功能测试又前进了一步，如果代码制品由低层功能项层次化构成，并由结构化编程中的控制结构连接起来，那么功能测试将递归的进行。例如，如果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>、<低层功能2>已经通过了功能测试，<高层功能>就可以使用分支覆盖进行测试，即13.12.1节中将会描述的玻璃盒技术。这种形式的结构测试是一种混合技术，低层功能用黑盒技术测试，而高层功能却用玻璃盒技术测试。

然而实践中，低层功能构成高层功能的方式并不是结构化的，相反，低层功能往往相互纠缠在一起。这种情况下要确定错误，需要进行功能分析（functional analysis），这是个相对有些复杂的过程，细节参见 [Howden，1987]。一个更复杂的因素是功能通常与代码制品边界不一致，因此，单元测试和集成测试的区分变得模糊；单一的代码段只能和它所用到功能的其他代码段同时测试。当一个对象的方法发送消息给（或调用）另一个对象的方法时，也会出现这个问题。

从功能测试的角度看，代码制品间随机的相互关系可能产生管理者不可接受的后果。例如，里程碑和最终期限可能定义的不好，从而难以确定对应于软件项目管理计划的产品状态。

13.11　黑盒测试用例：MSG基金会案例研究

图13-13和图13-14包含MSG基金会案例研究的黑盒测试用例。首先考虑由等价类和边界值分析衍生出来的测试例。图13-13中的第一个测试用例测试如果一项投资的itemName不是以字母开头，产品是否能发现错误。接下来的5个测试用例检查itemName是否由1～25个字符组成。类似的测试用例检查规格说明中的其他语句，如图13-13所示。

现在来看功能测试，规格说明文档中列出了10项功能，如图13-14所示。另外11个测试用例对应于这些功能的误用。

投资数据	
itemName 的等价类	
1. 第一个字符不是字母	错误
2. 少于一个字符	错误
3. 一个字符	可接受
4. 字符个数在 1 ~ 25 之间	可接受
5. 25 个字符	可接受
6. 大于 25 个字符	错误（名字太长）
itemNumber 的等价类	
1. 非数字字符	错误（不是数字）
2. 少于 12 个数字	可接受
3. 12 个数字	可接受
4. 大于 12 个数字	错误（太多数字）
estimatedAnnualReturn 和 expectedAnnualOperatingExpenses 的等价类	
1. <0. 0 美元	错误
2. 0. 0 美元	可接受
3. 0. 01 美元	可接受
4. 0. 01 ~ 999 999 999. 97 美元	可接受
5. 999 999 999. 98 美元	可接受
6. 999 999 999. 99 美元	可接受
7. 1 000 000 000. 00 美元	错误
8. >1 000 000 000. 00 美元	错误
9. 字符而非数字	错误（不是一个数）
抵押信息：	
accountNumber 的等价类与 itemNumber 相同	
抵押人姓的等价类	
1. 第一个字符不是字母	错误
2. 小于一个字符	错误
3. 一个字符	可接受
4. 字符个数在 1 ~ 21 之间	可接受
5. 21 个字符	可接受
6. 大于 21 个字符	可接受（截断到 21 个字符）
最初住房价格，目前家庭收入和抵押余额的等价类	
1. <0. 0 美元	错误
2. 0. 0 美元	可接受
3. 0. 01 美元	可接受
4. 0. 01 ~ 999 999. 98 美元	可接受
5. 999 999. 98 美元	可接受
6. 999 999. 99 美元	可接受
7. 1 000 000. 00 美元	错误
8. >1 000 000. 00 美元	错误
9. 字符而非数字	错误（不是一个数）
每年房产税和房主的等价类	
1. <0. 0 美元	错误
2. 0. 0 美元	可接受
3. 0. 01 美元	可接受
4. 0. 01 ~ 99 999. 98 美元	可接受
5. 99 999. 98 美元	可接受
6. 99 999. 99 美元	可接受
7. 100 000. 00 美元	错误
8. >100 000. 00 美元	错误
9. 字符而非数字	错误（不是一个数）

图 13-13　由等价类和边界值分析得到的 MSG 基金会案例的黑盒测试例

规格说明文档中列出的功能用来创建测试用例
1. 增加一项抵押
2. 增加一项投资
3. 修改一项抵押
4. 修改一项投资
5. 删除一项抵押
6. 删除一项投资
7. 更新运营费用
8. 计算购买房屋的资金
9. 打印抵押列表
10. 打印投资列表
除了这些直接的测试，需要进行以下额外的测试
11. 试图增加一项已经在文件上的抵押
12. 试图增加一项已经在文件上的投资
13. 试图删除一项不在文件上的抵押
14. 试图删除一项不在文件上的投资
15. 试图修改一项不在文件上的抵押
16. 试图修改一项不在文件上的投资
17. 试图两次删除一项已经在文件上的抵押
18. 试图两次删除一项已经在文件上的投资
19. 试图两次更新一项抵押的每个域并检查到存储了第二个版本
20. 试图两次更新一项投资的每个域并检查到存储了第二个版本
21. 试图两次更新运营费并检查到存储了第二个版本

图 13-14　MSG 基金会案例的功能分析测试例

一旦完成了分析流程，就可以开发这些测试用例，知道这一点很重要，它们出现在这里的唯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。每个测试计划的一个主要组件是：应该约定，只要分析结果获得通过，就马上提出黑盒测试用例，供 SQA 小组在实现工作流期间使用。

13.12　玻璃盒单元测试技术

在玻璃盒测试技术中，测试用例的选择是基于对代码而不是规格说明的检查。有许多不同形式的玻璃盒测试，包括语句、分支和路径覆盖。

13.12.1　结构测试：语句、分支和路径覆盖

玻璃盒单元测试的最简单形式是*语句覆盖*（statement coverage），即运行一系列测试用例，期间每条语句至少执行一次。为了跟踪哪些语句还需要执行，CASE 工具记录了在一系列测试中每条语句执行的次数，**PureCoverage** 是一个这种的工具。

这种方法的一个缺点是不能保证所有的分支结果得到了恰当的测试。为了理解这点，考虑图 13-15 的代码片段，程序员犯了一个错误，复合条件 `s>1 && t==0` 应该是 `s>1 || t==0`。图中的测试数据能够让语句 `x=9` 执行到，却不能发现错误。

```
if (s > 1 && t == 0)
  x = 9;

测试用例: s = 2, t = 0.
```

图 13-15　带有测试数据的代码片段

语句覆盖的一种改进是*分支覆盖*（branch coverage），即运行一系列测试，确保所有的分支至少被测试一次。同样，测试者通常需要工具来帮助他来跟踪哪些分支已经或还没有测试到。像语句覆盖和分支覆盖这样的技术称为*结构测试*（structural test）。

路径覆盖（path coverage）是结构测试中最强大的形式，即测试所有的路径。前文提到，在包含循环的产品中，路径数目确实会非常大。

因此，研究人员一直在研究在降低需要检查的路径数目的同时揭示比使用分支覆盖更多错误的方法。选择路径的一条准则是将测试用例局限于*线性代码序列*（linear code sequences）[Woodword, Hedley, and Hennell, 1980]。为此，首先标记出控制流中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 **if** 或 **goto** 语句。线性代码序列是那些以 L 中的元素开头，并且终止于 L 中的元素的路径。这项技术获得了成功，它发现了许多错误，而不必测试每条路径。

另一种减少测试路径数的方法是*全定义使用路径覆盖*（all-definition-use-path coverage）[Rapps and Weyuker, 1985]。这项技术的立足点是，源代码中变量（如变量 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()`。标出变量的定义和定义的使用之间的全部路径（现在可以通过自动工具完成）。最后，为每条这样的路径建立一个测试用例。全定义使用路径覆盖是一项优秀的测试技术，通过相当少的测试用例往往可以发现大量错误。然而，全定义使用路径覆盖也有缺点，路径数的上界是 2^d，其中 d 是产品中判定语句（分支）的数目，可以构建例子展示这个上界。然而，与人为制造的例子不同的是，实际产品中这个上界是达不到的，实际的路径数是与 d 成比例的 [Weyuker, 1988a]。换句话说，全定义使用路径覆盖需要的测试用例数通常比理论的上界小很多。所以，全定义使用路径覆盖是一项实用的测试用例选择技术。

使用结构测试时，测试者可能没有提出检查某一语句、分支或路径的测试用例，代码制品中也许存在不可行的路径（“死代码”），即对任何输入数据都不可能执行到的路径。图 13-16 展示了两个不可行路径的例子。图 13-16a 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。类似的，在图 13-16b 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。程序员本来打算检查 `j < 10`，但书写错误。使用语句覆盖的测试者很快就会意识到两条语句都执行不到，从而错误将被发现。

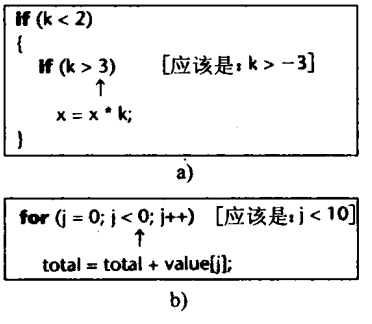


图 13-16　两个不可达路径的例子

13.12.2　复杂性度量

质量保证提供了另一种玻璃盒单元测试方法。假定经理被告知代码制品 m1 比代码制品 m2 更复杂，且不论复杂这个术语是如何准确定义的，经理直觉上就相信 m1 会比 m2 错误更多。顺着这条思路，计算机科学家已经开发出许多软件*复杂性*（complexity）的测度方法，以帮助确定哪些代码制品更可能有错误。如果发现一个代码制品的复杂度高得离谱，经理可能直接要求重新设计和实现这个代码制品，因为与努力调试一个有很多错误的代码制品相比，可能重新来过的代价更小，速度更快。

预测错误数量的一个简单的测度方法是计算代码行数。基本的假设是一行代码包含一个错误的概率恒定为 p。如果测试者相信每行代码包含一个错误的平均概率是 2%，而接受测试的代码制品是 100 行，则意味着此代码制品预计包含 2 个错误；而另一个 2 倍长的代码制品可能有 4 个错误。[Basili and Hutchens, 1983] 和 [Takahashi and Kamayachi, 1985] 指出，错误数量的确与软件产品的整体规模有关。

在寻找更精密的基于产品复杂性测度的错误预报器这个方向上大家已经做了很大的努力。一个有代表性的成果是 McCabe [1976] 的*秩复杂性*（cyclomatic complexity）度量，即二元判定（预测）的数目加 1。如 12.10 节所述，秩复杂度本质上是代码制品中的分支数，因此，秩复杂度可以作为代码制品分支覆盖所需的测试用例数目的一个测度标准，这是所谓的*基于结构的测试*（structured testing）的基础 [Watson and McCabe, 1996]。

McCabe 的测度标准几乎可以和代码行数一样容易计算，在某些方面已经显现出它是一个很好的预测错误的测度标准，M 值越高，代码制品包含错误的可能性就越大。例如，Walsh［1979］分析了 Aegis 系统（舰船海战系统）中的 276 个模块。他测量了秩复杂度 M，发现占 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 的模块比 M 值较小的模块每行代码包含的错误多 21%。然而，［Shepperd，1988］和［Shepperd and Ince，1994］从理论根据和许多不同实验的基础上都对 McCabe 的测度的有效性提出了严肃的质疑。

Musa、Iannino 和 Okumoto［1987］分析了有关错误密度的相关数据。他们得出结论，多数复杂度测度标准（包括 McCabe 的测度标准）都显示出与代码行数很高的相关性，或者更精确地说，与可交付、可执行的源代码指令数有很高的相关性。换句话说，当研究者测量他们所认为的代码制品或产品的复杂度时，他们得到的结果很可能是代码行数的反映，而这又与错误数量有很强的相关性。此外，复杂度衡量与通过代码行数预报错误相比几乎没有什么改进，［Shepperd and Ince，1994］中讨论了其他关于复杂度的问题。

13.13　代码走查和审查

6.2 节介绍了通常使用走查和审查的极端情况，代码走查和代码审查的观点是一致的。简单的说，这两项静态技术的错误发现能力将错误检测引向快速、彻底和提前。由于在集成阶段出现的错误较少而提高了生产率，用于代码走查和审查的额外时间得到了巨大的回报，并且，代码审查可降低高达 95% 正确性维护成本［Crossman，1982］。

进行代码审查的另一个原因是，基于执行的测试（测试用例）在以下两个方面代价非常大。第一，耗费时间。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更早期得到检测和纠正。如图 1-5 所示，越早发现和纠正一个错误，花费的成本就越少。一个极端的高成本运行测试用例的例子是 NASA 的阿波罗计划，软件预算的 80% 消耗在测试上［Dunn，1984］。

13.14 节将进一步给出支持走查和审查的观点。

13.14　单元测试技术的比较

许多研究者对单元测试策略进行了比较。Myers［1978a］比较了黑盒测试、黑盒测试和玻璃盒测试的结合以及第三方代码走查。实验由 59 个资深程序员测试相同的产品。在发现错误方面 3 项技术同样有效，但是代码走查被证明比其他两项技术成本低。Hwang 比较了黑盒测试、玻璃盒测试和单人代码阅读［Hwang，1981］，这 3 项技术效果一样，不过每项技术都有其优势和弱点。Basili 和 Selby 进行了一项较大规模的实验［1987］，与 Hwang 的实验一样，被比较的技术有黑盒测试、玻璃盒测试和单人代码阅读。实验由 32 名专业程序员和 42 名高年级学生构成。每人测试 3 个产品，每次使用一项技术。使用分数因子设计［Basili and Weiss，1984］补偿由于不同参与者使用不同方法测试产品而造成的差异，没有参与者用一种以上方法测试同一个产品。从两组参与者中得到了不同的结果：专业程序员通过代码阅读方式比其他两种技术发现更多的错误，而且错误发现速率也更快。其中两组由高年级学生参加，其中一组，3 项技术之间没有发现明显的差别；另一组中，代码阅读和黑盒测试一样，都比玻璃盒测试性能好。然而，学生们发现错误的速率对于这 3 项技术都是一样的。总的来说，代码阅读会比另外两项技术发现更多的接口错误，而黑盒测试更容易发现控制方面的错误。从这个实验总结的主要结论是，代码审查在发现错误方面跟玻璃盒测试和黑盒测试技术是一样成功的。

有项开发技术很好地利用了这个结论，这就是净室软件开发技术。

13.15　净室

净室（cleanroom）技术［Linger，1994］是许多不同软件开发技术的组合，包括增量式生

命周期模型、分析和设计的形式化技术，以及代码阅读［Mills，Dyer，and Linger，1987］、代码走查和审查（13.13 节）这样的静态单元测试技术。净室的一个关键特性是代码段在没有通过审查前不能编译，即代码段仅在成功完成静态测试后才进行编译。

许多使用这项技术的项目获得了巨大成功，比如用净室为美国海军水下系统中心开发的一个自动化文档系统原型［Trammel，Binder，and Snyder，1992］。在设计经历“功能验证”这个引入正确性证明技术（6.5 节）的检查过程时一共发现了 18 个错误。尽可能使用 6.5.1 节给出的非形式化的证明，仅当参与者不确定所检查的部分设计的正确性时才给出完整的数学证明。对 1820 行的 FoxBASE 代码进行走查又发现 19 个错误；编译代码时没有任何编译错误。而且在执行期间没有任何故障。这是展现静态测试技术能力的又一个标志性项目。

这样的结果当然使人印象深刻，但是前面提到，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。尽管如此，净室应用于大型产品的结果也给人留下了深刻印象。相关的测度标准是*测试错误率*（testing fault rate），即每 KLOC（千行代码）发现的错误总数，在软件业是一个相当通用的测度标准。然而，当净室技术与传统开发技术对比使用时，这种测度方法有明显的差别。

6.6 节指出，当采用传统开发技术时，代码制品在开发时由程序员进行非正式测试，然后由 SQA 小组进行系统的测试。在开发代码期间由程序员检测到的错误是不进行记录的，而从代码制品离开程序员的个人工作区，提交给 SQA 小组接受动态的和静态测试开始，检测到的错误总数是要记录的。相反，当采用净室技术时，“测试错误”是从编译时开始计数，然后错误计数持续到动态测试。换句话说，采用传统开发技术时，由程序员非正式检测到的错误不计入测试错误率；使用净室技术时，编译之前的审查和其他静态测试期间的发现的错误是记录的，但它们不计入测试错误率。

17 种净室产品的报告出现在［Linger，1994］中。例如，350 000 行的 Ericsson Telecom OS32 操作系统采用净室技术开发，该产品由 70 人团队在 18 个月内开发完成，测试错误率只有每千行代码 1.0 个错误。另一个产品是前面介绍的自动化文档系统原型，1 820 行程序的测试结果是每千行代码 0.0 个错误。17 种产品总代码量大约 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 认为这是一项卓越的质量成就，这种赞誉并不为过。

13.16　测试中的问题

推动面向对象范型使用的众多原因之一是它降低了对测试的需求。由继承而产生的复用是该范型的一个主要的优势。一旦类测试完了，变量传递了，就没有必要重新测试了。进一步地，在测试过的类的子类中新添加的新方法是需要测试，但从父类继承过来的方法不需要进一步测试。

事实上，上面的两种说法都只是部分正确。另外，对象的测试引起了面向对象所特有的问题，这里将讨论这些问题。

首先，有必要澄清一个关于类的测试和对象的测试的问题。7.7 节解释过，类是种抽象数据类型，它支持继承，而对象是类的实例，即类没有具体的实现，而对象是在一个特定环境内执行的物理代码块。因此，不可能对一个类进行动态测试，而只能进行静态测试，例如可以做审查。

信息隐藏及许多方法只有少数几行代码的事实，对测试具有重要的影响。首先，考虑采用传统范型开发的产品。现在，这样的产品一般由具有大约 50 条可执行指令的模块组成。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：调用模块时提供的输入参数和模块返回的输出参数。测试一个模块包括提供数值给输入参数、调用模块以及将输出参数的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，它们中有很多是相当小的，往往只有2～3 条可执行语句［Wilde，Matthews，and Huitt，1993］。这些方法不向调用者返回值，而是改变对象的状态，即这些方法修改对象的属性（状态变量）。这里的难题是，要测试状态改变是否正确

执行，必须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，测试某个deposit方法是否已经正确执行的唯一途径是，在调用方法deposit之前和之后，都调用方法determineBalance观察储蓄余额的变化情况。

如果对象不包括可以调用来确定所有状态变量值的方法，情况就更糟了。一种选择是为此添加额外的方法，然后使用条件编译确保它们除了测试用途之外不可用（C++使用#ifdef实现）。测试计划（9.7节）应规定在测试期间每个状态变量的值可以访问。为了达到这个需求，设计流中需要在有关的类中把返回状态变量值的附加方法添加进去。于是，通过查询适当的状态变量的值，测试调用对象的某个特定方法的效果成为可能。

非常出乎意料的是，继承的方法可能仍然需要测试。也就是说，即使一个方法已经通过了足够的测试，当子类不加改变的继承它时，可能仍然需要全面的测试。要理解后一个观点，考虑图13-17给出的类层次。基类**RootedTreeClass**中定义了两个方法displayNodeContents和printRoutine，其中方法displayNodeContents使用了方法printRoutine。

接下来考虑子类**BinaryTreeClass**，它从基类**RootedTreeClass**继承了方法printRoutine。此外，它定义了新的方法displayNodeContents，覆盖**RootedTreeClass**中定义的同名方法，这个新的方法仍调用printRoutine。如果用Java表示，BinaryTreeClass.displayNodeContents使用RootedTreeClass.printRoutine。

```
class RootedTreeClass
{
  ...
  void displayNodeContents (Node a);
  void printRoutine (Node b);
//
// 方法 displayNodeContents 调用方法 printRoutine
//
  ...
}

class BinaryTreeClass extends RootedTreeClass
{
  ...
  void displayNodeContents (Node a);
//
// 这个类中定义的方法使用displayNodeContents
// 从RootedTreeClass 继承的方法printRoutine
//
  ...
}

class BalancedBinaryTreeClass extends BinaryTreeClass
{
  ...
  void printRoutine (Node b);
//
// (从BinaryTreeClass 继承的)方法displayNodeContents 使用
// BalancedBinaryTreeClass 中printRoutine的局部版本
//
  ...
}
```

图13-17　树状层次结构的Java实现

现在考虑子类**BalancedBinaryTreeClass**。这个子类从它的基类**BinaryTreeClass**继承了方法displayNodeContents。此外，它定义了一个新方法printRoutine，覆盖**RootedTreeClass**中那个方法的定义。当displayNodeContents在**BalancedBinaryTreeClass**环境中调用printRoutine时，C++和Java的作用域规则规定使用printRoutine的局部版本。用Java表

示的话，在 **BalancedBinaryTreeClass** 的上下文范围内调用方法 BinaryTreeClass. displayNodeContents 时，它使用方法 BalancedBinaryTreeClass. printRoutine。

因此，在 **BinaryTreeClass** 的实例中调用 displayNodeContents 时，实际执行的代码（方法 printRoutine）跟在 **BalancedBinaryTreeClass** 的实例中调用 displayNodeContents 时执行的代码不同。这种观点始终成立，尽管方法 displayNodeContents 本身由 **BalancedBinaryTreeClass** 从 **BinaryTreeClass** 中不加改变的继承。因此，即使方法 displayNodeContents 已经在 **BinaryTreeClass** 对象中全面测试过了，在 **BalancedBinaryTreeClass** 环境中重用时，必须从头开始重新测试。在更复杂情况下，理论上需要用不同的测试用例重新测试［Perry and Kaiser, 1990］。

必须指出，不能以这些复杂性为理由抛弃面向对象范型。第一，它们只出现在方法有交互时（在例子中是 displayNodeContents 和 printRoutine）。第二，可以确定什么时候需要重新测试［Harrold, McGregor, and Fitzpatrick, 1992］。

假定一个类的实例已经全面测试过了，那么它的子类中任何新的或重新定义的方法需要测试，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交互。简而言之，使用面向对象范型很大程度上降低了测试的需求，这种说法是成立的。

现在考虑单元测试中涉及管理方面的内容。

13.17　单元测试的管理方面内容

每个代码制品的开发期间必须做出的的一个重要的决定是，在该代码制品的测试上要花费多少时间以及资金。和软件工程中有许多其他的经济因素一样，成本－效益分析（见 5.2 节）可以发挥作用。例如，基于成本－效益分析可以决定，正确性证明所用的成本是否超出为保证该产品满足其规格说明所带来的效益。成本－效益分析也可以用于比较运行附加的测试用例的成本和由不足的测试引起的可交付产品故障的成本。

另一个方法用于决定是否继续测试某一代码制品，还是认为几乎全部的错误已经排除了。即可靠性分析技术，这种技术可以用来提供遗留错误数的统计估计。目前已经提出各种不同的技术用来确定遗留错误数的统计估计。这些技术的基本思想是这样的：假定一个代码制品测试持续一周，在星期一找到 23 个错误，星期二又找到 7 个，星期三又找到 5 个错误，星期四 2 个，星期五没有找到错误。因为错误发现率从 23 个错误开始每天稳步递减一直到没有，看起来大多数错误已经找到，对该代码制品的测试可以停止了。确定代码中不再有错误出现的概率需要一定层次的数理统计知识，这超出了本书的知识范围，因此细节在这里没有给出，对可靠性分析感兴趣的读者可以参考［Grady, 1992］。

13.18　何时重写而不是调试代码制品

前面提到，当 SQA 小组的成员发现故障（错误的输出）时，该代码制品必须返回给原来的程序员进行调试（debugging），即检测错误，改正代码。在有些情况下，扔掉该代码段从头开始重新设计和重新编写可能更可取，可以由最初的程序员完成，也可以由另一个更资深的开发小组成员完成。

要明白为什么需要这样，考虑图 13-18。该图展示了一个与直觉相反的概念，代码制品中存在更多错误的概率与该代码制品中目前发现的错误数成正比［Myers, 1979］。要理解为什么会这样，考虑两个代码制品 a1 和 a2。假定这两个代码制品长度相近，经过相同时长的测试。进一步假设在 a1 中只发现 2 个错误，而在 a2 中发现 48 个错误，那么在 a2 中比 a1 中很可能仍然存在更多的错误，并且对 a2 进行额外的测试和调试的过程可能更长，甚至会怀疑 a2 仍不完善。无论从短期还是长期看，最好的办法是放弃 a2，重新设计和编写。

错误在模块中的分布是不均匀的。Myers［1979］引用了用户在 OS/370 中发现的错误的例子，他发现 47% 的错误只与 4% 的模块有关。更早一些，Endres［1975］在德国 Böblingen 的 IBM 实验室所做的关于 DOS/VS（28 版）的内部测试显示了类似的不均匀性。202 个模块总共发现 512 个错误，有 112 个模块只发现 1 个错误；另一方面，某些模块分别发现了 14、15、19 和 28 个错误。Endres 指出，后 3 个模块是产品中最大的 3 个模块，每个都由超过 3 000 行的 DOS 宏汇编语言组成，而发现 14 个错误的模块是个已知非常不稳定的小模块，这类模块可以考虑丢弃和重写。

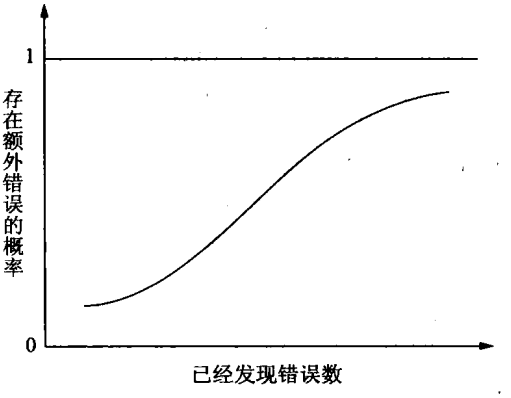


图 13-18　将发现错误的可能性与已检测出的错误数成比例图示

管理者处理这种情况的办法是，预先确定一个给定代码制品在开发期间所允许的最大错误数，一旦达到该最大值，必须丢弃该代码制品，然后由有经验的软件设计人员重新设计和编写。最大值会随着应用领域的不同而不同，还会随着代码制品的不同而不同。当然，从数据库读取记录并检查该部分数据有效性的代码制品中允许发现的最大错误数，应该比一个需要整理来自各种传感器的数据，并将炮口瞄准目标的坦克武器控制系统的复杂代码制品少得多。确定某个代码制品允许最大错误数的有效办法是参考某个类似的已得到纠错性维护的代码制品的错误情况。但是，不管采用什么估计技术，一旦超出预定的错误数，管理者必须保证放弃该代码制品（参见备忘录 13.6）。

备忘录 13.6

讨论中涉及的开发期间一个代码制品允许发现的最大错误数，准确说是指“开发期间”所允许的错误最大值。产品交付给用户后允许发现的最大错误数，对于全部产品的所有代码制品都应该是“零”。也就是说，向客户交付无错误的代码应该是所有软件工程师的目标。

13.19　集成测试

每个新的代码制品在加入到已集成的模块中时都必须进行集成测试（integration test）。这里的关键点是首先采用 13.9 ~ 13.13 节所述（单元测试）测试新的代码制品，然后检查这部分产品的其余部分的行为是否和集成这个新代码制品之前一样。

如果产品有图形用户接口，集成测试会出现新的问题。把测试用例的输入数据存放在一个文件上可以简化产品的测试。然后执行该产品，提交相关的数据。借助 CASE 工具，整个过程可自动进行，设定一组测试用例，同时包括每个用例的期望输出，CASE 工具运行每个测试用例，把实际结果和期望结果相比较，并向用户报告每个测试用例的情况。然后保存测试用例，以便修改产品时做回归测试。SilkTest 就是这类测试工具的一个例子。

然而，当产品具有图形用户接口时，这种方法就没用了。特别是下拉菜单或点击鼠标的测试数据不能像通常的测试数据一样存放在文件中。同时，手工测试 GUI 是耗时而枯燥的。解决这个难题的办法是利用能保存鼠标点击、按键等记录的特殊 CASE 工具。通过手工测试 GUI 一次来使 CASE 工具建立测试文件。然后将这个文件用于随后的测试。有许多支持 GUI 测试的 CASE 工具，包括 QAPun 和 Xrunner。

集成过程完成后，软件产品作为整体进行测试，术语叫产品测试（product testing）。当开发者对软件产品各方面的正确性都能保证时，就把它交给客户进行验收测试（acceptance testing）。

接下来详细地讨论这两种形式的测试。

13.20　产品测试

成功集成最后一个代码制品到产品中并不意味着开发人员的任务结束，SQA 小组还必须执行许多测试任务以确保产品成功。软件主要有两种类型：商用现货（COTS）软件（1.10 节）和定制软件。COTS 产品测试的目标是确保产品整体上没有错误。产品测试完成后，经受 α 测试和 β 测试，如 3.7.4 节所述，即将产品的初步版本有选择的送给预期的买家，从他们那里得到反馈意见，特别需要注意 SQA 小组没有注意到的残留错误。

另一方面，定制软件需要经受一些不同的产品测试。SQA 小组执行很多测试任务以确保产品在验收测试时通过，这是定制软件开发团队必须跨越的最后一道障碍。定制软件如果未能通过验收测试，基本上属于开发组织管理能力低下的反映。客户会得出结论，这些开发者能力不行，从而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，故意交付不合格的产品，以便尽早结束合同拿到酬金。如果客户真这样认为，并告诉其他潜在客户，开发者便会面临巨大的公共关系难题。所以 SQA 小组必须确保产品成功通过验收测试。

为了确保验收测试成功，SQA 小组必须用自认为最接近即将到来的验收测试的方式测试产品：

- 必须针对产品整体运行黑盒测试用例。到目前为止，已经建立了基于各个代码制品和类的测试用例，以确保每个代码制品和类满足其规格说明；
- 必须对整个产品的健壮性进行测试。集成期间已经测试了单个代码制品或类的健壮性，现在必须建立和运行测试用例来测试整个产品的健壮性。此外，产品必须经受压力测试（stress testing），也就是说，确保产品在峰值负荷下操作仍能正确运行，例如，所有终端同一时刻都在尝试登录或者客户同时在操作所有的自动柜员机。产品还要经受容量测试（volume testing），例如，确保它可以处理大的输入文件。
- SQA 小组必须对产品是否满足所有约束进行检验。例如，如果规格说明规定产品在满负荷下工作时对 95% 的查询响应时间必须小于 3 秒，那么 SQA 的职责就是验证产品确实满足这种要求。毫无疑问，客户在验收测试期间会检验约束，如果产品有一条重要约束没有达到，开发组织将失去相当多的信任。同样，存储性约束与安全性约束也必须检验。
- SQA 小组必须检查所有随代码一起交付给客户的文档。SQA 小组必须遵照 SPMP 中的标准校验文档。此外，文档必须对照产品校验，例如，SQA 小组必须确认用户手册能真正反映产品的正确使用方法，同时产品的功能必须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刁难，产品（代码及文档）就可以移交给客户组织进行验收测试。

13.21　验收测试

客户进行验收测试的目的是确定产品是否确实具有开发者在规格说明中所声称的功能。验收测试可以由客户组织实施，也可以在有客户代表在场的情况下由 SQA 小组实施，或者由客户雇用的独立 SQA 小组实施。验收测试自然包括正确性测试，但除此之外，还需要进行性能和健壮性测试。验收测试的 4 个主要组成部分，即正确性测试、健壮性测试、性能测试、文档测试，实际上就是开发者在产品测试期间所做的事，这并不奇怪，因为产品测试就是验收测试的全面预演。

验收测试的关键在于必须在真实数据上而不是在测试数据上实施测试。无论测试用例设定如何完善，本质上还是人工设定的。更重要的是，测试数据应该是对应真实数据的反映，但在实践中情况并不总是这样。例如，负责编写真实数据规格说明的团队成员可能没有正确的完成任务；或者，即使正确规定了数据，使用这些数据规格说明的 SQA 小组成员也可能产生误解，

导致测试用例并不能正确反映真实数据，产生未经充分测试的产品。因此，验收测试必须在真实数据上实施。进一步地，因为开发团队要努力保证产品测试全方位的模拟验收测试，用真实数据实施的产品测试也是越多越好。

当新产品要替换现有产品时，规格说明文档几乎总是包括这样一条，即新产品安装后必须与现有产品并行运行。原因在于新产品在某些方面很有可能存在错误，而现有产品运行正常但在某些方面存在不足。如果现有产品被运行不正常的新产品替换，用户将会有麻烦。因此，新旧产品必须并行的运行，直到客户觉得新产品完全可以替代现有产品的功能。并行运行的成功促使验收测试的通过，这时现有产品就可以退役了。

产品通过验收测试后，开发者的任务就完成了，现在起对产品所做的任何更改都属于交付后维护。

13.22　测试流：MSG 基金会案例研究

MSG 基金会产品的C++ 和 Java 实现（可以从 www. mhhe. com/schach 下载）经过图 13-13 和图 13-14 的黑盒测试用例测试，也经过习题 13. 30 ~ 13. 34 的玻璃盒测试用例测试。

13.23　用于实现的 CASE 工具

第 5 章详细阐述了支持代码制品实现的 CASE 工具。集成需要版本控制工具、创建工具和配置管理工具（第 5 章）。因为，处于测试中的代码制品随着一系列错误被发现并更正将不断变化，而这些 CASE 工具对于保证编译及连接每一代码制品的适当版本是必要的。现有的商业配置控制平台有 PVCS 和 SourceSafe。CVS 是个流行的开源配置控制工具。

到目前为止，每章中已经针对每个工作流的 CASE 工具和平台进行了阐述。前面已经阐述了开发过程的所有工作流，现在可以对开发过程的 CASE 工具进行整体研究了。

13.23.1　软件开发全过程的 CASE 工具

CASE 工具本身是个自然发展的过程。如 5. 5 节中所述，最简单的 CASE 设备是个单独的工具（tool），比如在线接口检查器或创建工具。接下来，可以对工具进行组合，由此产生支持一个或两个软件开发过程活动的工作平台（workbench），比如配置控制或编码。然而，这样的工作平台甚至不能为软件开发过程中它所运用的有限部分提供有用的管理信息，更不用说为整个项目了。最终将形成为开发过程提供大部分（即使不是全部）的计算机辅助支持的环境（environment）。

理想状态下，每个软件开发组织应当使用一种环境。但使用环境的成本可能会非常大，不仅仅是软件包本身，还包括该软件运行的硬件设备。对于小公司来说，一个工作平台或者也许一套工具就足够了。但是，如果可能的话，应该采用集成环境（integrated environment）来进行开发和维护。

13.23.2　集成开发环境

集成在 CASE 环境中的最普遍的含义是用户接口集成（user interface integration）。即所有工具在环境中共享通用的用户接口。这句话的内在含义是，如果所有工具都有一样的可视界面，那么使用环境中某个工具的用户可以没有任何困难的学习和使用其中的另一种工具。这种思想在 Macintosh 中已经得到了成功的实现，Macintosh 中的大部分应用软件都有相似的“外观和感受”。当然这只是通常的含义，也有其他类型的集成。

术语工具集成（tool integration）是指所有的工具采用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 管道格式假定所有数据采用 ASCII 流的形式。因此，将一个

工具的输出流指向另一工具的输入流就可以容易地实现两个工具的组合。Eclipse 是用于工具集成的开源环境。

过程集成（process integration）指支持一个特定软件过程的环境。基于技术的环境（technique-based environment）（参见备忘录 13.7）是这类环境的子集。这类环境只支持开发软件的某一特定技术，而不是全过程。这些环境采用图形化界面为分析和设计提供支持，并集成了数据字典，此外，还提供了一些一致性检验。环境中往往还集成了对开发过程管理的支持。目前有许多该类型的商业环境，包括支持状态图［Harel et al.，1990］的 Rhapsody 和支持统一过程［Jacobson，Booch，and Rumbaugh，1999］的 IBM Rational Rose。此外，一些较老的环境已经得到扩展，可以支持面向对象范型，Software through Pictures 就是这类例子。

备忘录 13.7

文献中基于技术的环境的另一种叫法为基于方法的环境（method-based environment）。面向对象范型的出现，给了方法第二种含义（在软件工程环境中）。它的原本含义为技术或途径，也就是在基于方法环境中的含义，在面向对象中的含义是对象或类中的操作。可惜的是，有时从其上下文并不能弄清楚它的具体含义。

因此，本书仅在面向对象范型上下文中使用方法这个词，其他场合采用技术或途径来表达。这就是为什么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，在本章中采用基于技术的环境这一术语。

大多数基于技术的环境重点强调对由这种技术规定的软件开发人工操作的支持和形式化。也就是说，这些环境迫使用户一步一步的按照环境开发者预定的方式使用这种技术，同时通过提供图形工具、数据字典和一致性检验来帮助用户。这种计算机化的框架工作在迫使用户使用并正确使用特定的技术方面加强了基于技术的环境。但这也同时可能是个缺点。除非组织的软件过程集成了该项特定的技术，否则使用基于技术的环境可能达不到预期目的。

13.23.3　商业应用环境

另一类是用于开发面向商业产品的重要环境。它强调易用性，采用多种方法实现。特别是里面包括一些标准界面，通过友好的 GUI 用户可以对其进行各种修改。代码生成器是这种环境的一个普遍特征，产品的最底层抽象为详细的设计。详细设计作为代码生成器的输入，代码生成器则会自动生成某种编程语言的代码，例如，C++、Java。只需编译这些自动生成的代码，无需执行任何形式的“编程”工作。

详细设计的语言将可能是未来的编程语言。编程语言的抽象层次已经从物理机层次的第一代（机器码）和第二代编程语言上升到了抽象机层次的第三代（高级）和第四代编程语言。到今天，详细设计层次已经是这类环境的抽象层次，是一种可移植层次。使用代码生成器可以更好地实现更快捷开发并使交付后维护更容易，与编译器和解释器不同，程序员只需为代码生成器提供很少的细节。因此，支持代码生成器的面向商业环境将提高软件生产率。

目前有多种这类环境可用，包括 Oracle Developer Suite。切记面向商用 CASE 环境的市场规模。在未来几年可能会有更多这类环境出现。

13.23.4　公共工具基础结构

欧洲信息技术研究战略计划（ESPRIT）开发了一种支持 CASE 工具的基础结构。尽管名为可移植公共工具环境（Portable Common Tool Environment，PCTE）［Long and Morris，1993］，但它不是环境。相反，它只是为 CASE 工具提供所需服务的基础结构，这与 UNIX 为其用户产

品提供所需的操作系统服务相类似。（PCTE 中“common”的含义是“public”或“not copyrighted”。）

PCTE 已经得到了广泛认可。例如，PCTE 及其 C 和 Ada 的接口在 1995 年已经被采用为 ISO/IEC13719 的标准。PCTE 的实现包含了 Emeraude 与 IBM 的实现。

将来希望有更多的 CASE 工具遵守 PCTE 标准。而 PCTE 自身也能在更广泛的计算机上实现。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，将会在更大的范围内出现 CASE 工具，反过来，这又将会带来更好的软件过程和更高质量的软件。

13.23.5　环境的潜在问题

对所有产品和所有组织都是最理想的环境是不存在的，也没有任何一种编程语言被认为是“最好的”。每种环境都有其优势和缺点，选用不合适的环境可能比不使用环境的情况还要糟糕。例如，13.23.2 节中解释过，基于技术的环境本质上是使人工开发过程自动化。如果一个开发组织选用了一种强制其采用某种技术的环境，而这种技术从整体上对该组织并不合适，或者对于正在开发的软件产品是不合适的，则使用该 CASE 环境将达不到预期目的。

更糟糕的情况是，如果该组织忽视 5.10 节的建议，即除非该组织达到 CMM3 级标准，否则应该避免使用 CASE 环境。当然，每个组织都应该使用 CASE 工具，使用工作平台一般不会带来损害。然而，在使用环境是，它将自动化的软件过程施加于使用它的组织。如果该组织正在使用一个良好的过程，也就是说，该组织为 3 级或更高，通过过程的自动化，环境可以在软件生产的各个方面起到帮助。如果组织处于危机驱动的 1 级或 2 级，则不存在这样的过程。对不存在的过程进行自动化，即采用 CASE 环境（与 CASE 工具或 CASE 工作平台相对），只可能会带来混乱。

13.24　测试工作流的 CASE 工具

在实现工作流期间有许多 CASE 工具用来支持不同类型测试的执行。先看单元测试。XUnit 测试框架包括用于 Java 的 JUnit 和用于C++ 的 CppUnit，是一组开源的用于单元测试的自动化工具，即，它们用来依次测试每个类。准备一组测试用例，该工具通过给类发送消息同时检查返回的期望应答结果。有许多供应商提供类似的商用工具，包括 Parasoft。

现在看集成测试。商用的支持自动集成测试（还有单元测试）工具包括 SilkTest 和 IBM Rational Functional Tester。这类工具通常汇集单元测试用例并采用得到的测试用例集来进行集成测试和回归测试。

在测试流期间，最重要的是管理人员要知道所有缺陷的状态，特别是要知道哪些缺陷已经检测出来但还没有纠正。最常用的是一个开源缺陷跟踪工具 Bugzilla。

再回到图 1.5，尽可能早的检测出代码错误是很重要的。这一点可以通过使用 CASE 工具来分析代码，发现通常的句法和语义错误，或者将会导致问题的构造。这样的工具包括 IBM Rational Purify、Sun 公司的 Jackpot Source Code Metrics 以及 3 个 Microsoft 工具：PREfix、PREfast 和 SLAM。

Hyades 项目（又称为 Eclipse 测试和性能工具项目）是一个集成测试、跟踪、监视环境的开源项目，可以用于 Java 和C++ 。它具有用于不同测试工具的设施。由于越来越多的工具供应商把它们的工具纳入 Eclipse 中工作，用户对测试工具将拥有广泛的选择，并且它们能够相互配合工作。

13.25　实现工作流的度量

一些实现工作流中不同的复杂性测度在 13.12.2 节中已经讨论了，包括代码行数和 McCabe

的秩复杂度。

从测试的角度看，相关的测度包括测试用例的总数及导致失败的测试用例的数目。通常的错误在代码审查中必须统计。错误总数是非常重要的，因为在代码制品中所检测到的错误数超过预定的最大数量后，如 13.19 节中所述，该代码制品将必须重新设计和重新编写。另外，还需进行关于错误的种类的详细统计。典型错误类型包括对设计的理解错误、初始化操作未进行及变量使用前后不一致。错误数据将被纳入到检查列表中，该列表将在未来产品的代码审查中使用。

针对面向对象范型的一些测度已经提出，例如，继承树的高度［Chidamber and Kemerer, 1994］。许多这样的测度在理论和实践中都被质疑［Binkley and Schach, 1996; 1997］。进一步地，Alshayeb 和 Li［2003］曾指出，尽管面向对象测度能够相当精确的预计在敏捷过程中增加、修改和删除的代码行数，在基于框架的过程（8.5.2 节）中，它们在预期同样这些测度的时候基本没有什么用处。相对可同样应用于面向对象软件的传统测度，是否需要特别的面向对象测度仍需进一步观察。

13.26　实现工作流面临的挑战

自相矛盾的是，在实现工作流之前就已经遇到实现工作流面临的主要问题。如第 8 章中所描述，代码复用是降低软件开发成本和缩短交付时间的一个有效途径。然而，如果延迟到实现工作流才做这方面的工作，就很难实现代码复用。

例如，假定决定采用 L 语言实现产品。在一半代码制品已经实现和测试后，管理者决定采用软件包 P 来设计产品的图形用户界面。不管 P 的功能有多么强大，如果它们的编写语言与 L 兼容困难，那么就不能复用在软件产品中。

即使语言的互操作没有问题，如果复用的代码段不能很好的适合设计，代码段的复用没有什么意义。修改现成的代码段可能比从新编写代码段所需的工作量更大。

因此代码复用应该从一开始就成为软件产品的一部分。复用不仅是客户需求，也是规格说明文档的约束要求。软件项目管理计划（见 9.6 节）必须包含复用。而且，设计文档必须写明将要实现哪些代码段，以及将要复用哪些代码段。

因此，如本节开始所指出，虽然代码复用是实现工作流面临的一个重要挑战，但代码复用还必须结合在需求、分析和设计流当中。

从纯技术的角度看，实现工作流相对直接。如果需求、分析及设计流达到满意结果的话，有能力的程序员应该能很好的完成实现任务。然而集成的管理尤其重要，实现工作流面临的挑战主要在这个方面。

典型的不成功便成仁的议题包括：使用恰当的 CASE 工具（13.23 节），客户签订规格说明后的测试计划（9.7 节），保证设计的改变能够及时知会相关人员（13.5.5 节），决定何时结束测试并将产品交付给客户（6.1.2 节）。

本章回顾

本章给出了由团队实现产品的各种相关问题。它们包括编程实现语言的选择（13.1 节），13.2 节介绍了良好的编程习惯，实用编码标准的必要性在 13.3 节给出。然后，对代码复用做出评论（13.4 节）。实现和集成活动必须并行的开展（13.5 节），13.5.1 ~ 13.5.3 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。集成技术在 13.5.4 节中讨论，集成管理在 13.5.5 节中讨论。实现工作流在 13.6 节给出，并在 13.7 节应用于 MSG 基金会案例研究。接下

来 13.8 节讨论的是测试流的实现方面问题。测试用例必须系统地选择（13.9 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分别在 13.10 节、13.12 节和 13.13 节中作了介绍，然后在 13.14 节作了比较。13.11 节给出了 MSG 基金会案例研究的黑盒测试。13.15 节描述了净室技术。13.16 节中讨论了对象的测试，随后讨论了单元测试管理上的实现（13.17 节）。13.18 节讨论了另一个问题是何时重写而不是调试代码段，集成测试在 13.19 节进行了讨论，产品测试在 13.20 节描述，验收测试在 13.21 节描述。MSG 基金会案例研究的测试流在 13.22 中作了概括性的介绍。13.23 节描述了实现工作流的 CASE 工具。具体地说，完整过程的 CASE 工具在 13.23.1 节中讨论，集成开发环境的 CASE 在 13.23.2 节中讨论，商业应用环境的 CASE 在 13.23.3 节中给出，13.23.4 节讨论了公共工具基础结构，接下来讨论了环境的潜在问题（13.23.5 节）。然后讨论了测试流的 CASE 工具（13.24 节）。实现工作流的测度在 13.25 节中讨论。最后以对实现工作流面临的挑战分析结束本章（13.26 节）。

延伸阅读材料

良好编程实践方面的优秀著作有［Kernighan and Plauger，1974］和［McConnell，1993］。

最重要的关于基于执行的测试的早期著作可能是［Myers，1979］。［Beizer，1990］是有关一般测试的全面信息来源。［Howden，1987］描述了功能测试，［Clarke，Podgurski，Richardson，and Zeil，1989］对结构化技术进行了比较。在［Beizer，1995］中有黑盒测试的深入描述，［Yamaura，1998］给出了黑盒测试用例的设计。［Horgan，London，and Lyu，1994］讨论了各种结构化测试的覆盖测度和软件质量之间的关系。［Stocks and Carrington，1996］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。

在［Linger，1994］中介绍了净室概念，［Sherer，Kouchakdjian，and Arnold，1996］给出了交付后维护期间净室的使用，在［Beizer，1997］中给出了净室的准则。

［Musa and Everett，1990］对有关软件可靠性方面做了很好的介绍。此外，每年软件可靠性工程国际会议论文文集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会议论文集中包含了相当广泛的测试问题。2000 年国际会议录用的文章收录在《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综述。关于这个主题的两篇重要文章是［Perry and Kaiser，1990］和［Harrold，McGregor，and Fitzpatrick，1992］。关于面向对象范型，［Jorgensen and Erickson，1994］介绍了面向对象软件的集成测试。

关于实现的度量，McCabe 在［McCabe，1976］中首次提出秩复杂度，设计度量的扩展出现在［McCabe and Butler，1989］中。对秩复杂度的有效性提出质疑的文章包括［Shepperd，1988；Weyuker，1988b；and Shepperd and Ince，1994］。［Barnard and Price，1994］中描述了管理代码审查的测度。面向对象测度的有效性在［Alshayeb and Li，2003］中讨论。

在［Harrold and Soffa，1991］中有描述了集成测试中测试数据的选择，GUI 的测试用例的生成在［Memon，Pollack，and Soffa，2001］中有描述。

每两到三年，ACM SIGSOFT 和 SIGPLAN 会发起一个有关软件开发环境实践的讨论会，会议论文集提供了广泛的工具包和环境的信息。每年的计算机辅助软件工程国际专题的论文集也提供非常有帮助的信息。

关于 PCTE，［Long and Morris，1993］包含许多相关的信息资料。

习题

13.1　如果让你实现 Osric 的办公用品和装饰产品（附录 A），你打算采用哪种语言实现该产品，为什么？在可使用的语言中，给出它们的效益和成本，不要试图在你的答案上附上美元值。

13.2　对电梯问题案例研究（11.6 节）重做习题 13.1。

13.3　对图书馆自动循环系统（习题 8.7）重做习题 13.1。

13.4　对确定银行储户报告书是否正确的软件产品（习题 8.8）重做习题 13.1。

13.5　对自动柜员机（习题 8.9）重做习题 13.1。

13.6　在最近编写的代码段中增加序言注释。

13.7　在编码标准方面单人软件公司与拥有 300 名软件专业人员的公司有什么不同？

13.8　在编码标准方面，对于开发和维护特别护理单元软件的公司，与开发和维护财务产品的组织有什么不同？

13.9　建立 Naur 文本处理问题（6.5.2 节）的黑盒测试用例，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。

13.10　采用习题 6.15 的解答（或者老师发的代码），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。

13.11　对分支覆盖，重做习题 13.10。

13.12　对全定义 - 使用路径覆盖，重做习题 13.10。

13.13　对路径覆盖，重做习题 13.10。

13.14　对线性代码序列，重做习题 13.10。

13.15　绘出习题 6.15 解答（或者老师发的代码）的流程图，确定它的秩复杂度。如果不能确定分支数，把流程图作为一个有向图考虑，确定边数 e，节点数 n，以及连通分支 c 的数量（每个方法构成一个连通组件），秩复杂度 V 由下式给出［McCabe，1976］：

$$V = e - n + 2c$$

13.16　假定你是某个单人软件公司的唯一雇主和雇员，同时买了 5.6 节描述的编程平台，按照它对你的重要程度级别，列出它的 5 种能力，给出理由。

13.17　作为 Very Big Software 公司的软件技术副总裁，公司有 17 500 名员工。如何按级排列 5.6 节描述的编程工作平台的能力？解释对这个问题的答案和对上一题的答案的不同点。

13.18　作为软件开发公司的 SQA 管理者，负责确定测试期间给定代码段中可以发现的最大错误数。如果超出这个最大值，那么该代码段必须重新设计和编写。将采用什么准则确定给定代码段允许的最大错误数？

13.19　解释逻辑代码段和操作代码段的区别。

13.20　保守编程是种良好的软件工程习惯。但同时，复用时它可能会妨碍操作代码段的充分测试，如何解决这个明显的冲突？

13.21　产品测试与验收测试的相似点是什么？主要区别是什么？

13.22　在实现阶段 SQA 小组的主要扮演的角色是什么？

13.23　假定你是某个单人软件公司的唯一雇主和雇员。为了使公司更具竞争力，决定购买 CASE 工具。为此需要从银行贷款 15 000 美元。银行管理人员要求出示一份长度不多于一页（越短越好）的说明，解释为什么需要这些 CASE 工具。写出这份说明。

13.24　Ye Olde Fashioned 软件公司新任命的软件开发副总裁雇用你帮助改变公司软件开发的方式。公司共有 650 名员工，在没有任何 CASE 工具的帮助下编写 COBOL 源代码。请为副总裁写一份备忘录，建议应该购买什么样的 CASE 工具。同时论证你的选择。

13.25　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，留下了 100 万美元前提是你将钱花在面向商业的环境和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？

13.26　假定你是个小型的人文艺术学院的计算机科学教授。计算机科学课程的编程作业需要在 35 台个人

计算机构成的网络上完成。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使用许可，否则每种 CASE 工具都需要购买 35 份。对此有何建议？

13.27 假定你刚刚被选为某个大城市的市长。发现为城市开发的软件中没有使用任何 CASE 工具，你该怎么办？

13.28 （学期项目）为习题 11.22 中规定的产品设计黑盒测试用例。对于每个测试用例，说明要测试什么，对该测试用例期望的输出是什么。

13.29 （学期项目）实现并集成 Osric 办公用品和装饰产品软件（附录 A）。使用教师指定的实现语言。教师将说明构造的软件是否是基于 Web 用户接口、图形用户接口或是基于文本的用户接口。请使用习题 13.28 中所得到的黑盒测试用例进行代码测试工作。

13.30 （案例研究）下载一份 13.7 节中描述的 MSG 基金会产品的实现副本，为该产品设计语句覆盖测试用例。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。

13.31 （案例研究）对分支覆盖，重做习题 13.30。

13.32 （案例研究）对全定义使用路径覆盖，重做习题 13.30。

13.33 （案例研究）对路径覆盖，重做习题 13.30。

13.34 （案例研究）对线性代码序列，重做习题 13.30。

13.35 （案例研究）从附录 F 的详细设计开始，使用除C++ 和 Java 之外的面向对象语言编写 MSG 基金会案例研究。

13.36 （案例研究）用纯 C 重新编写 MSG 基金会案例研究（13.7 节），不要使用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的实现。如何实现多态和动态绑定呢？

13.37 （案例研究）对于 13.7 节中实现代码的文档，到什么样的程度是不合适的？给出必要的补充。

13.38 （软件工程读物）教师分发［Alshayeb and Li，2003］的复印件，讨论这篇文章能让你确信对面向对象测度的有效性的信心吗？

参考文献

[Alshayeb and Li, 2003] M. Alshayeb, and W. Li, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes," *IEEE Transactions on Software Engineering* **29** (November 2003), pp. 1043–49.

[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, 2003] C. J. DATE, *An Introduction to Database Systems*, 8th ed., Addison-Wesley, Reading, MA, 2003.

[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.

[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 and Soffa, 1991] M. J. HARROLD AND M. L. SOFFA, "Selecting and Using Data for Integration Testing," *IEEE Software* **8** (1991), pp. 58–65

[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.

[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.

[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.

[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 14 章　交付后维护

学习目标

通过本章学习，读者应能：

- 执行交付后维护。
- 认识到交付后维护的重要性。
- 描述交付后维护面临的挑战。
- 描述面向对象范型中的维护的含义。
- 描述维护所需的技能。

本书一个主要的主题就是说明软件交付后维护是极其重要的。因此，读者对于本章篇幅相对较短可能会惊讶。由于产品从一开始就应该把可维护性纳入其中，并且在开发过程中的任何阶段都不能削弱。相应的，前面所有的章节实际上已经关注交付后维护这个主题。本章要讨论的主要是如何在交付后维护期间确保产品的可维护性不被削弱。

14.1　开发与维护

一旦产品经过了验收测试，产品就交付给了客户，产品安装后并按照建造它的用途使用。然而，任何有用的产品都将经受交付后维护，要么修正错误（纠错性维护）要么扩展（强化）产品功能。

由于产品不仅包括源代码，在产品交付给客户后任何对文档、手册或其他任何的改动均属于交付后维护的范畴。有些计算机科学家喜欢用演化而不喜欢用维护这个词来说明产品随着时间的推移而改进。事实上，有些人把软件整个生命周期从开始到结束看作逐渐演化的过程。

这是维护在统一过程中的看法。事实上，在 Jacobson、Booch 和 Rumbaugh［1999］中几乎没有出现维护一词，只是含蓄地将维护看作软件产品的一个增量。然而，开发与维护具有基本的区别，通过下面的例子来表达这个区别。

假设一名妇女在 18 岁时有了自己的一幅画像。该画描绘了她的头部和肩膀。20 年后她结婚了，她想要修改这幅画以同时包含丈夫和自己。按照要求修改肖像将产生 4 个方面的困难。

1）画布不够大以至无法添加她丈夫的头像。

2）原来的肖像悬挂时由于白天太阳照在上面，画的颜色有点褪色了。另外，原画使用的油彩品牌已经不生产了。由于这两个原因，很难在颜色上达到一致。

3）原来的画家已经退休，所以在绘画风格上很难达到一致。

4）自从原画完成后，该妇女的脸型经历了 20 年岁月的洗礼，要确保改后的画像是她本人，有相当多的工作需要做。

基于上述原因，考虑修改原画感觉有点可笑，相反，可以请一位新的画家为这对夫妇重画肖像（参见备忘录 14.1）。

备忘录 14.1

伦敦国家美术馆内收藏有一幅画作，该画由于添加了一个头像而被毁坏。1515 年，画家 Lorenzo Lotto（公元 1480—1556）为 Giovanni Agostino della Torre（一位住在意大利 Bergamo，后移居到威尼斯的医师）画了一幅肖像画。下载这幅画［Lotto，1515］并检查就可发现，画家

在原画完成后又添加了 della Torre 的儿子 Niccolo，因而不可修复地毁坏了该画作。

在伊斯坦布尔 Hagia Sophia 博物馆收藏了另一幅被毁掉的名作，12 世纪南部走廊墙壁上的马赛克描绘了拜占庭帝国的 Zoe 女皇（公元 978—1050）。背靠 Zoe 的是她的第三个丈夫——康斯坦丁四世，从［Magnus，2005］可以看到，他看上去脖子很短，原因是马赛克原来描绘的是 Zoe 和她的丈夫——贾柳斯三世。马赛克画布在她第三次结婚后更新了，但画家把康斯坦丁的头画得太大了，所以头的下部盖住了贾柳斯的脖子的大部分。

现在考虑最初使用 200 万美元开发的软件的维护问题，要解决的困难有 4 项：

1）不幸的是，存放数据库的磁盘空间满了，当前的磁盘容量不够大以至于无法添加更多的数据。

2）生产磁盘的原来公司已不再经营了，因此更大容量的磁盘需要从另一个生产商那里购买。然而，现有的软件产品和新磁盘（8.10.1 节）之间存在着硬件上的不兼容，为使用新的磁盘需要花费大约 10 万美元修改软件。

3）原来的开发者在几年前离职了，因此对产品的修改将由以前从没接触过这个软件的维护团队完成。

4）原软件产品采用传统范型开发的，而现在面向对象范型被广泛使用（特别是统一过程）。

肖像画的情况与软件产品的情况有着明显的相似性，关于油画，答案无疑是重画一幅新的肖像。这是不是意味着不选择花费 10 万美元的维护工作，而应当花费 200 万美元去开发一个全新的软件产品呢？

答案是相似性的推广不能太过。应该重新绘画新的肖像，但同样也是很显然的是，应该只花费开发新软件产品成本的 5% 对现有的软件产品进行维护。

然而，从这个不太好的类比中可得到一条重要的经验。无论处理的是肖像还是软件产品，创建一个新版本比修改现有版本相对容易。对于肖像情况，不仅修改已有的肖像是不可能，而且修改现有肖像的开支肯定比重画一幅的开支大。在软件产品的情况下，不仅修改是可行的，而且修改的成本只是重新开发新软件产品所需成本的一小部分。换句话说，尽管对现存制品修改比重新建立制品更难，但从经济方面考虑，维护比重新开发更可行。

14.2　为什么交付后维护是必要的

对产品修改主要出于三方面的原因：

1）错误需要纠正，无论是分析错误、设计缺陷、编码错误、文档错误还是其他任何错误，这称为*纠错性维护*（corrective maintenance）。

2）在*完善性维护*（perfective maintenance）中，代码修改是为了提高产品的有效性。例如，客户可能希望增加功能，或对产品进行修改使它运行速度更快。提高产品的可维护性是另一个完善性维护的例子。

3）在*适应性维护*（adaptive maintenance）中，为适应产品运行环境的变化对产品所做的修改。例如，如果产品需要移植到新的编译器、操作系统或硬件平台上，那么修改几乎是必须进行的。例如，税收代码的每次改动，生成纳税回执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，原来只允许 5 位邮政编码的产品不得不进行调整。适应性维护并不是客户提的要求，而是外界给客户造成的。

14.3　交付后维护程序员要求具备什么

交付后维护工作在软件生命周期中所占的时间比其他任何活动都多。实际上，平均来看，

产品总成本至少有67%用于交付后维护，如图 1-3 所示。但直到现在，许多组织仍然把交付后维护任务交给刚入门或能力不强的程序员，而把产品开发中“亮点”部分留给更出色或更具经验的程序员。

事实上，交付后维护是软件产品各方面中最困难的部分。一个主要原因是，交付后维护包含了软件开发过程中所有工作流的各个方面。设想一下当一份缺陷报告送到维护程序员手上时所发生的情况（1.10 节中缺陷（defect）是差错、故障或错误的统称）。从用户的角度看，如果产品没有按用户手册上的说明运行，他就会提交缺陷报告。由多种可能的原因。首先，软件本身根本没错，只是用户没有正确理解用户手册或者没有正确使用该产品。另一种情况是产品中确实有错误，也可能只是用户手册编写有错误，而代码本身没有任何问题。然而，大部分情况是代码中有错误。但是在做任何修改之前，维护程序员必须依据用户的缺陷报告和源代码（一般不会再有其他可依据的了）来准确判定错误原因。因此，维护程序员需要有非同一般的调试能力，因为错误可能存在于产品内的任何位置。而且缺陷可能是由到目前为止还不存在的分析或设计制品所导致的。

如果维护程序员找出了错误的位置，那么必须在修复该错误的同时防止无意中在产品其他位置引入另一个错误，即*回归错误*（regression fault）。如果希望回归错误最小化，需要整个产品及各个代码制品的详细文档。然而，软件专业人员是以不喜欢一切书面形式工作（特别是建立文档）而著称的。文档不全、存在错误或完全丢失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得到的唯一有效的文档——源代码，来推测避免引入回归错误的一切信息。

在确定可能的错误并试图将其纠正后，维护程序员必须对所做修改的正确性进行测试，以及是否引入了回归错误。为了检查修改本身，维护程序员必须设计专门的测试用例。回归错误的检查需要使用为进行*回归测试*（regression testing）而存储的测试数据集来完成（3.8 节）。接下来，为检查修改而设计的测试用例必须加入存储的测试用例集中，以便供将来产品调整后做回归测试用。另外，如果纠正错误时对分析或设计进行了修改，那么这些修改也必须检查。因此，具备测试方面的专业知识是交付后维护工作的另一前提条件。最后，维护程序员为每一处修改建立文档是非常重要的。上述讨论主要与纠错性维护相关。这时，维护程序员首先必须是一个出色的诊断专家用来确定是否存在错误。如果有错误，他还必须作为熟练的纠错技师来修复错误。

另外两个主要的维护任务是适应性维护和完善性维护。为执行这些维护任务，维护程序员必须以现存的产品作为起点，完成需求流、分析流、设计流和实现流。对于某些类型的修改，需要额外设计和实现代码制品。在其他情况中，需要对已有的代码制品的设计和实现进行修改。因此，尽管经常规格说明文档是由分析专家完成，设计由设计专家完成，代码由编程专家完成，但维护程序员需要是这三个领域的专家。与纠错性维护一样，适当文档的缺乏同样是完善性维护和适应性维护面临的不利影响。而且，如纠错性维护中一样，在完善性维护和适应性维护中，需要设计合适测试用例及编写良好文档的能力。因此，没有任何形式的维护工作可以由缺乏经验的程序员完成，除非有优秀的计算机专家监督维护过程。

从前面的讨论可以清楚的看出，维护程序员几乎必须拥有软件专业人员所应该具备的全部技能。但他们得到了怎样的回报呢？

- 无论从哪个角度看，交付后维护都是一项吃力不讨好的工作。维护人员总是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。
- 用户遇到的问题通常是由产品开发人员导致的，而不是维护人员。
- 代码本身写得不好，会加重维护人员的挫折感。
- 许多软件开发人员通常不愿意做交付后维护，他们认为开发才是一项有技术含量的工作，而维护只适合初级程序员或能力不强的程序员。

交付后维护很可能作为售后服务。产品已经交付给客户。但现在客户并不满意，因为产品运行不正确，可能是产品不能满足客户目前的所有要求，或者产品当时建立的环境发生了某种变化。除非软件公司提供良好的维护服务，否则客户以后将会把开发业务交给其他软件公司。当客户和软件开发部门属于同一个组织时，双方不可避免的要从将来的工作考虑，如果客户不满意，他会使尽一切好的坏的办法，损坏软件团队的信誉。这反过来又会导致软件团队内外信心危机，退出开发工作，或者客户不再雇用。通过提供优秀的交付后维护服务令客户保持满意对每个软件组织都是一项重要的工作。因此，对于一件接一件的产品来说，交付后维护是软件生产过程中最富挑战性的方面，而且通常是吃力不讨好的。

这种状况怎样才能改观呢？管理者必须把交付后维护工作交给那些具备维护所需的全部技能的程序员。管理者必须让组织内的其他人知道，只有顶级的计算机专业人员才有资格做维护，同时要向维护程序员支付相应级别的报酬。如果管理层认为维护工作是一项具有挑战性的工作，良好的维护对本组织的成功非常重要，那么大家对交付后维护工作的态度将慢慢得到改善（参见备忘录 14.2）。

备忘录 14.2

Tom Pigoski［1996］的《实用软件维护》（Practical Software Maintenance）一书介绍了其在佛罗里达 Pensacola 如何建立美国海军软件交付后维护机构的情景。他的观点是，如果提前告诉未来的雇员，他们将是维护程序员，他们将会对交付后维护工作持积极的态度。另外，通过确保员工受到各种各样的培训，并在工作过程中有机会到全世界旅行的方式来保持员工高昂的士气。附近的美丽海滩以及使用的新办公楼在这方面也同样起到积极的作用。

然而，员工在这个机构开始工作的 6 个月内，他们都询问什么时候才能参加一些开发工作。看来改变大家对待维护工作的态度是非常困难的。

对维护程序员遇到的一些问题现在可以通过小型案例研究加以强调。

14.4　交付后维护小型案例研究

在中央经济体制的国家里，政府掌控着农产品的分配和采购。假定有这样一个国家，温带水果委员会（TFC）负责所有温带水果（如桃子、苹果和梨）的各项事宜。有一天，TFC 的主席要求政府计算机顾问将 TFC 的工作计算机化。主席告诉计算机顾问，只有 7 种温带水果，苹果、杏、樱桃、油桃、桃子、梨和李子。数据库的设计应恰好只容纳这 7 种水果，不能冗余，也不能少。总之，这就是世界的一切，计算机顾问不需要浪费时间和金钱考虑任何扩展性问题。

产品按时交付给了 TFC。大约 1 年后，TFC 主席把负责产品维护的程序员集中到一起。问道："你们了解猕猴桃吗？"程序员迷惑地回答："不清楚。"主席说："好，猕猴桃是我们国家刚开始种植的温带水果，TFC 将对此负责。请你们对软件产品进行相关修改。"

维护程序员幸运地发现，那位计算机顾问并没有一字不差地按 TFC 主席原来的指示进行软件开发。计算机顾问心中牢固的良好编程习惯使他考虑产品将来的扩展性，在相关数据库记录中预留了许多空字段。只要对数据库中的某些项稍加重新调整，维护程序员就能把第 8 种温带水果，猕猴桃，加入到软件产品之中。

又过了 1 年，产品运行正常。后来，维护程序员再次被叫到主席办公室，他心情不错并告诉程序员，政府对农产品的分配与采购进行了重组。他的委员会现在不仅仅负责温带水果，而是负责本国所有的水果。为了容纳他交给维护程序员的水果清单上的另外 26 种水果，软件必须

加以修改，程序员们抗议说，修改的时间跟从头重写这个软件差不多。主席回答说“胡说！在增加猕猴桃的时候不是不存在问题吗？同样的工作做 26 次就行了！”

从这个案例可以得出许多重要的教训：

- 产品本身没有提供可扩展性的问题是由开发人员导致的，而不是维护程序员。开发人员错误执行了 TFC 主席关于软件产品未来扩展性方面的指示，但维护程序员要承担其带来的后果。实际上，开发这个产品的计算机顾问如果不阅读本书，可能永远不会意识到她开发的产品根本算不上成功。交付后维护工作一个更令人恼火的方面是维护程序员是需要负责纠正别人的错误。造成问题的人可能另有工作或已经离职，但造成的后果却要由维护程序员来承担。
- 客户通常不理解交付后维护是困难的，而且在有些情况下甚至是不可能的。维护程序员以前可能成功地执行了完善性和适应性维护，但对突然提出新的维护任务却完成不了，虽然这些任务表面上与以前没什么困难的任务差不多。这个时候，问题就更突出了。
- 所有软件开发都应该带着交付后维护的眼光进行。如果那位计算机顾问在设计软件时就考虑可以处理任意数目和种类的水果的话，则后来增加猕猴桃和另外 26 种水果就不会有任何问题。

正如多次指出的那样，交付后维护是软件生产中一个最重要的方面，也是最耗资源的一项工作。在产品开发过程中，重要的一点是开发团队不能忽视了维护程序员，后者将在产品安装后对产品负责。

14.5　交付后维护的管理

现在考虑有关软件交付后维护的管理问题。

14.5.1　缺陷报告

维护产品的第一需要是对产品修改的机制。对于剔除遗留错误的纠错性维护，也就是在产品运行不正确时，则缺陷报告应该由用户提交。缺陷报告（defect report）必须包括足够的信息，以使维护程序员可以重现该问题，通常这些问题属于某种类型的软件故障。另外，维护程序员必须给出缺陷的严重性，典型的严重性分类包括关键的、主要的、普通的、小的和微不足道的。

理想情况下，每个用户提出的缺陷应立即被处理掉。而实际上，软件开发公司程序员通常满负荷工作，开发和维护工作都会相对滞后。如果缺陷是关键的，比如工资发放软件在发工资前一天或正在发工资时崩溃了，那么立即纠正措施必须马上执行。其他情况下，每一份缺陷报告必须得到初步调查。

维护程序员应该首先针对缺陷报告文件给出方案。这个方案包括了所有目前发现但尚未纠正的缺陷，以及在缺陷得到纠正之前用户应如何绕过它们的建议。如果该缺陷以前被报告过，缺陷报告中的任何信息都应该转给该用户。但如果用户报告的是新缺陷，那么维护程序员应研究该问题并试图找到原因和解决问题方法。否则，应该试图找到绕过该问题的办法，因为有可能需要经历 6～9 个月的时间才能分配人员对软件做出必要的修改。考虑到程序员，尤其是优秀的维程序员的短缺，对于那些不是太紧急的缺陷报告，建议用户通过某种方法继续使用带有缺陷的软件，直到缺陷解决是唯一的办法。

然后，维护程序员的结论和所有支持其结论的文档，用以得到结论的清单、设计、手册等，应该一起加入缺陷报告文件中。负责交付后维护的管理者应当定期考虑该报告，确定各种修复任务的优先级。该文件同样应包括客户在完善性维护和适应性维护方面的要求。对产品下一个修改将是优先级最高的缺陷。

如果有若干套产品副本分布到不同地方，那么该缺陷报告必须向该产品的所有用户分发，

该报告应该给出纠正这些缺陷的预期日期。然后，如果在另一个地方出现相同的问题，用户可以查阅相关的缺陷报告来确定是否可以绕过缺陷以及什么时候这些缺陷能被纠正。当然最理想的是每个错误得到马上纠正并向所有用户发送新版本的产品。考虑到世界范围内目前优秀程序员的短缺，以及交付后软件维护的现实状况，发布缺陷报告也许是目前能用的最佳办法。

缺陷不能得到立即纠正通常还有另一个原因。大量缺陷一次性修改，然后对全部修改进行测试，修改文档并安装产品的新版本，比单独纠正每个错误、进行测试、文档化并安装新产品，再对下一处错误重复整个周期来说成本更低。当每个新版本软件必须安装在大量计算机（如C/S网络上的大量客户端）上或软件运行在不同地方时，上述情况更为明显。结果是，开发单位倾向于积累非关键性的维护任务，然后批量处理。

14.5.2　授权对产品的修改

一旦做出进行纠错性维护的决定，维护程序员就承担了查找失败的原因并对该错误进行修复的任务。代码被修改后，必须像测试整个产品一样对修复进行测试（回归测试）。然后文档必须更新以反映做过的修改。特别是对修改过的代码制品，要在其序言注释中加入关于做了哪些修改、为什么要修改、由谁做的修改，以及何时做的修改等方面的信息（见图13-1）。如果必要的话，分析和设计制品也需要修改。在进行完善性和适应性维护时，也需要进行类似的步骤。唯一的区别在于完善性维护和适应性维护是应对需求变化进行的，而不是由缺陷报告引起的。

到这里为止剩下的工作将是把新版本发布给用户。但是，如果维护程序员对修改所做的测试不充分会怎样呢？在产品发布前，该产品要提交给一个独立的小组进行软件质量保证，即维护SQA小组的成员提交报告的管理者不能是维护程序员的管理者。SQA小组管理上的独立性非常重要（6.1.2节）。

前面给出了为什么交付后维护工作困难的原因。同样的原因，维护工作也是容易出错的。交付后维护的测试是困难而且耗时的，SQA小组不应该低估测试对软件维护的影响。一旦新版本得到SQA小组的认可则可以发布。

在什么时候使用软件基线技术或私人拷贝（5.8.2节）是管理层需要保证工作程序得到严格遵守的另一个领域。如果程序员打算修改**Tax Provision Class**，则**Tax Provision Class**和维护工作所需的其他所有代码制品被复制，通常包括产品中的所有其他类。程序员对**Tax Provision Class**做了修改并对其进行测试。现在，**Tax Provision Class**的原先版本被冻结，修改后的**Tax Provision Class**安装在软件基线之上。但是，当修改后的产品移交到客户手上后马上崩溃了。问题出在维护程序员使用的是私人工作空间的副本对修改后的**Tax Provision Class**进行测试，也就是刚开始对**Tax Provision Class**进行维护的这个时刻软件基线上的其他代码制品的副本。但同时，维护同一产品的其他程序员对某个其他代码制品进行了更新。这个教训非常深刻，在安装某一代码制品之前，必须采用当前软件基线所有其他代码制品进行测试，而不能用程序员的私有拷贝进行测试。这也是建立独立的SQA小组的进一步原因，SQA小组的成员没有权利访问程序员的私人工作空间。第三个原因在于对某个错误的初次纠正本身有70%是不正确的[Parnas，1999]。

14.5.3　确保可维护性

交付后维护不是一项一劳永逸的工作。一件好的产品在其生命周期中要经历一系列的版本变化。所以，在整个软件生产过程中有必要对交付后维护进行规划。例如，在设计工作流，信息隐藏技术（7.6节）应该被采用；在实现工作流，变量名的选择要使将来的维护程序员容易理解（13.2.1节）。文档要完整、正确，并能够反映出产品每一代码制品组件的当前版本。

在交付后维护期间，最重要的是不能削弱从一开始就建立起来的软件可维护性。换句话说，正如软件开发人员应该意识到软件的交付后维护是不可避免的--样，软件维护程序员也应始终

意识到软件将来同样不可避免的需要进一步交付后维护。开发期间建立的可维护性原则同样适用于交付后维护。

14.5.4　反复维护的问题

移动目标问题（moving-target problem）（2.4 节）是产品开发中一个更令人恼火的问题。客户改变需求的速度如同开发者建立产品的速度一样快，这个问题不仅困扰开发小组，而且频繁的变化将导致生产出低质量的产品。另外，这样的变化提高了产品的成本。

在交付后维护时，这个问题更为严重。产品修改的越多，它就越偏离原来的设计，以后的修改也就变得越困难。经历反复维护之后，文档可能会比平常更不可靠，回归测试文件也可能过时。如果还需做更多的维护，整个产品可能需要一次全部重写。

移动目标问题明显是个管理问题。在理论上，如果管理部门对待用户态度坚决，并在项目的开始就把问题解释清楚，那么从签订规格说明开始，需求将被冻结，直到产品交付为止。还有，在每次完善性维护需求被提出后，需求也将冻结 3 个月或 1 年。而实际上，这种方法是不可行的。例如，客户恰好是某公司的总裁，而开发组织恰好隶属这个公司的软件部，那么总裁可以每周一到周四都要求对软件进行修改，而且他们也必须实现。那句古老的谚语“出钱的人说了算”形容这种情况太贴切了。也许软件副总裁能做的最有效的事情就是试图向总裁解释反复维护对产品的影响，然后当继续修改会对产品的完整性造成损害时干脆重写产品。

采用拖延修改的时间来反对进一步修改是不可行的，这只能导致相关人员被准备以更快的速度完成任务的人替换掉。简而言之，如果要求反复修改的人有足够的权力，则没有办法解决移动目标问题。

14.6　维护问题

推动面向对象范型使用的原因之一是它提高了产品的可维护性。毕竟对象是程序中的独立单元。更明确的说，设计良好的对象表现在概念上的独立，这也称为*封装*（encapsulation）（7.4 节）。产品中由对象建模的现实世界相关的方面均定位于对象本身。另外，对象也表现出物理的独立性，信息隐藏技术用来确保对象的实现细节对外不可见（见 7.6 节）。唯一允许的通信方式是向对象发送消息来调用某个特定的方法。

对象容易维护主要有两方面的原因。首先，概念独立性意味着容易发现产品的哪一部分需要进行修改以达到某个特定的维护目标，可以是增强性或纠错性维护。其次，信息隐藏技术确保对象本身的修改不会对对象以外造成影响，因此回归错误的数量得到大幅度的下降。

然而，实际情况并不这样理想。实际上，有 3 个专属面向对象软件维护的障碍。其中之一可以通过采用适当的 CASE 工具来解决，其他的则不易处理：

1）观察图 14-1 所示的C++ 类层次结构。方法 displayNode 在 **UndirectedTreeClass** 类中定义，同时由 **DirectedTreeClass** 类继承，最后在 **RootedTreeClass** 类中重新定义。这个重定义的版本被 **BinaryTreeClass** 类和 **BalancedBinaryTreeClass** 类继承，并在 **BalancedBinaryTreeClass** 类中使用。因此，维护程序员必须研究整个继承层次结构才能理解 **BalancedBinaryTreeClass** 类。更糟糕的是，层次通常不是按图 14-1 所示的线性结构放在一起，而是分布在整个产品中。所以，为了理解 displayNode 方法在 **BalancedBinaryTreeClass** 类中的具体行为，维护程序员需要仔细阅读产品的大部分代码。这与本节开始描述的对象“独立性”观念相去甚远。解决这个问题的办法非常直接：采用适当的 CASE 工具。正如C++ 编译器可以在 **BalancedBinaryTreeClass** 类的实例的内部精确判定 displayNode 方法的版本那样，维护程序员使用的编码平台可以提供类的“展开”版本，即类的全部定义，包括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 14-1 中 **BalancedBinaryTreeClass** 类的展开形式包括继承过来的 **RootedTreeClass** 中对 displayNode 方法的定义。

```
class UndirectedTreeClass
{
  ...
  void displayNode (Node a);
  ...
}// class UndirectedTreeClass

class DirectedTreeClass : public UndirectedTreeClass
{

  ...
}// class DirectedTreeClass

class RootedTreeClass : public DirectedTreeClass
{

  ...
  void displayNode (Node a);
  ...
}// class RootedTreeClass

class BinaryTreeClass : public RootedTreeClass
{

  ...
}// class BinaryTreeClass

class BalancedBinaryTreeClass : public BinaryTreeClass
{
  Node        hhh;
  displayNode (hhh);
}// class BalancedBinaryTreeClass
```

图 14-1　C++ 实现的一个类层次

2）对采用面向对象语言开发的产品维护时遇到的另一个障碍不容易解决。这是由 7.8 节介绍的多态和动态绑定介绍的两个概念所导致的。在那一节给出一个这样的例子：基类名是 **File Class**，一起还有 3 个子类 **Disk File Class**、**Tape File Class** 和 **Diskette File Class**。图 7-33b 显示了这几个类，为了方便把图 7-33b 复制为图 14－2。在基类 **File Class** 里，声明了一个哑（*抽象方法或虚函数*）方法 open。然后在 3 个子类分别给出了该方法的特定实现，每个方法的名称都跟图 14-2 中的 open 一样。假设 myFile 声明为 **File Class** 类的对象，如果要维护的代码中包含了 myFile. open( ) 方法调用。由于多态和动态绑定的原因，myFile 在运行时可能是 **File Class** 类的 3 个派生类（硬盘文件、磁带文件或软盘文件）中任何一个类的对象。运行时系统一旦确定了 myFile 所属的类，特定版本的 open 方法将被调用。这对维护工作是相当不利的。如果维护程序员在代码中遇到了 myFile. open( ) 方法调用，为了理解产品的这部分，必须考虑 myFile 分别作为 3 个子类中任意一个的实例时所发生的情况。CASE 工具在这种情况下提供不了任何帮助，因为静态方法无法解决动态绑定问题。在特定环境下确定大量绑定中到底哪个发生了，唯一的方法是对代码进行跟踪，或者通过运行代码，或者进行手工跟踪。多态和动态绑定实际上是有利于面向对象产品的开发的非常强大的面向对象技术之一。然而，它们对维护工作却是有害的，迫使维护程序员研究运行时发生的各种可能绑定，然后在方法的大量版本中确定在代码这一点上调用特定版本。

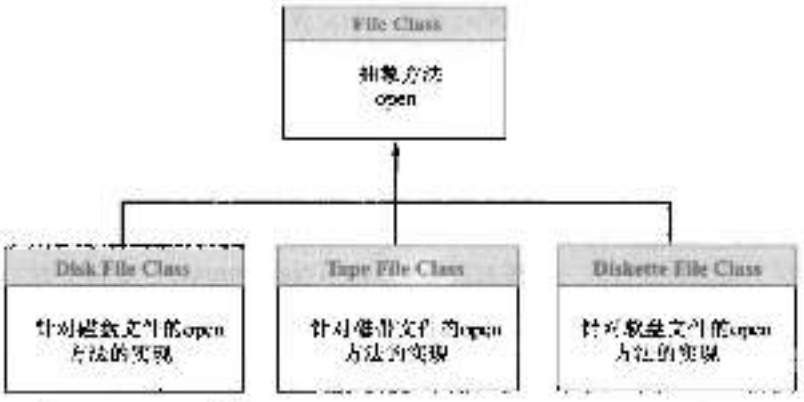


图 14-2 基类 File Class 和子类 Disk File Class、Tape File Class、Diskette File Class 的定义

3）最后一个问题来自于继承（inheritance）。假设某一基类实现了新产品设计中大多数但不是全部需求。因此需要定义一个派生类，在许多方面这个派生类和基类相同，但加入了新特性，重命名某些原有特征、重新实现、禁用或做一些其他方面的改动。另外，这些修改可能不会对基类或其他派生类产生影响。然而，如果基类本身改变了，所有派生类会跟随相同的变化。换句话说，继承的优势在于给继承树（继承图）加入新叶子并不改变继承树（如果实现的是C++这种支持多继承的语言，那么也可以是继承图）中其他的类。但是，如果继承树的内部节点发生了任何改变，那么这种变化将会影响到它的所有子孙节点（脆弱基类问题，fragile base class problem）。

由此，继承是对开发有重大的积极影响但对维护却存在负面影响的面向对象技术的另一特征。

14.7 交付后维护技能与开发技能

本章前面对交付后维护所需的技能已经讨论了许多。

- 对于纠错性维护，大型产品确定运行故障原因的能力是非常重要的。但不仅在产品交付后维护中需要这一技能，它贯穿与集成与产品测试始终。
- 在没有充足文档的情况下有效工作的能力是另外一项重要的技能。再者，在产品集成和测试时，文档几乎是不完整的。
- 另外要强调的是，对于适应性维护和完善性维护来说，分析、设计、实现和测试有关的技能也是非常重要的。在开发过程中同样要进行这些活动，而且每一项活动都需要特别的技能才能正确完成。

换句话说，交付后维护程序员所需的技能与软件生产其他方面的专业人员所需掌握的技能是没有什么不同。关键在于维护程序员不能仅泛泛地掌握不同领域的一些技能，而是要高度熟练掌握那些领域的技能。虽然一般的软件工程师只专长于软件开发的某一领域，如设计或测试，但软件维护程序员必须是软件生产所有领域的专家。毕竟，交付后维护与开发同等重要。

14.8 逆向工程

正如已经指出的那样，有时候交付后维护仅仅只有源代码本身作为的唯一文档。（在维护遗留系统（legacy systems）时这种情况经常发生，也就是说系统可能是15年或20年前开发的，目前仍在使用的软件。）在这种情况下，维护代码将极其困难。解决这类问题的一种方法是从源代码开始，试图重新创建设计文档甚至规格说明。这个过程叫逆向工程（reverse engineering）。

CASE工具能够辅助完成这一过程。最简单的例子是5.6节中的小型打印机，它能够帮助生

清楚的显示代码。其他工具可以直接从源代码建立如流程图或 UML 图这样的图，这些可视化辅助工具可以帮助设计恢复过程。

一旦维护小组重建设计后，有两种方案可选。一是试图重建产品规格说明，对建立后的规格说明加以修改以反映必要的变化，最后按通常的方法重新实现产品。（在逆向工程领域，通常从产品规格说明到设计再到编码的开发过程，称为正向工程（forward engineering）。在正向工程之后的逆向工程有时叫做再工程（reengineering）。）在实际中，重建规格说明是件非常困难的事情。通常的情况是，重新建立的设计要经过修改，然后采用修改后的设计实施正向工程。

在维护过程中经常要进行的一项相关活动是重组（restructuring）。逆向工程使产品从低层次抽象转换为高层次抽象，例如，从代码到设计。正向工程是使产品从高层次抽象转到低层次抽象。然而，重构是在同一层次上发生。它是在不改变产品功能的前提下完善产品的过程。实现精美打印是重构的一种形式，同样将代码由非结构化转化为结构化也属于重构。总的来说，重构是使源代码（或设计，甚至数据库）的维护更容易。当采用敏捷过程（2.9.5 节）时，被称为重构（refactoring）的设计修改是重组的另一个例子。

如果只剩下产品的执行程序，源代码丢失了，情况变得更糟。起初看来，得到源代码的唯一方式就是采用反汇编程序得到汇编代码，然后设计一种工具（可以称为反编译器）试图恢复产品的高级语言代码。这种方法将伴随着大量难以解决的问题：

- 经过最初的编译后，变量名都将不存在了。
- 许多汇编器对代码进行某种方式的优化，这使重建源代码变得极其困难。
- 汇编程序中的结构（如循环）可能与源代码中多种不同的结构相关联。

因此，现实中现有产品被看成一个黑盒子，需要采用逆向工程依据现有产品的行为来推导产品原来的规格说明。重建的规格说明做了必要的修改，并以这个规格说明为基础，通过正向工程开发产品的新版本。

14.9　交付后维护期间的测试

在产品开发阶段，开发团队的大部分人员对整个产品都有个全局的了解。但由于计算机从业人员的快速流动，交付后维护团队的成员不大可能参见过原来的开发。因此维护程序员倾向与把产品看成一系列松散相关的组件的集合，并没有意识到一个代码制品的改变可能将严重影响一个或更多的其他制品甚至整个产品。即使维护程序员想理解产品的各个方面，但修复和扩展产品的压力也使他们没有时间进行深入研究。另外，在很多情况下，很少或几乎没有文档来协助他们理解产品。把这种困难极小化的一种方法是采用回归测试，即采用以前的测试用例对修改过的产品进行测试，确保产品还能正常运转。

由于这个原因，把所有测试用例及预期的测试结果以机器可以识别的形式保留是十分重要的。由于产品的变化，某些保存的测试用例必须加以修改。例如，如果由于纳税法规的修订引起所得税比例的变化，那么涉及所得税的、用于测试工资发放软件的测试用例也要修改。同样，如果根据卫星观察数据要对某一个岛的经纬度进行修正，那么由该岛坐标计算飞机位置的软件的输出结果也要做相应的调整。随着维护内容的改变，有些有效的测试用例将变得无效。其实修正存储的测试用例时所需的计算跟为验证维护正确性而建立新测试用例所需的计算本质上是一样的。因此，对测试用例及其预期结果文件的维护并不需要更多的工作。

很多人认为回归测试就是浪费时间，因为回归测试要求对整个产品采用大量测试用例进行重新测试，而测试用例中的多数表面上跟在产品维护过程中修改过的代码制品没有任何关系。在前面的语句中，表面这个词非常关键。没有意识到维护工作副作用（即引入回归错误）的危险太大而使上面的论点难以成立。在所有情形下，回归测试是维护的一个重要方面。

14.10　交付后维护的 CASE 工具

指望维护程序员手工跟踪各种修订版本号，并在代码制品每次更新后为其指定下一个修订版本号是不合理的。除非操作系统包含版本控制功能，否则需要使用一个诸如 UNIX 工具中的 sccs（源代码控制系统）[Rochkind，1975] 和 rcs（修订版控制系统）[Tichy，1985]，或开源的 CVS（并发版本系统）[Loukides and Oram，1997] 这样的版本控制工具。同样，指望手工控制第 5 章所讨论的冻结技术，以及确保修订版能够相应更新的其他方法也是不合理的。这需要一个配置控制工具。典型的商业工具例子是 CCC（改变和配置控制）和 IBM Rational ClearCase。即使软件公司不愿意购买全套配置控制工具，也至少应该连同版本控制工具购买一套建造工具。在交付后维护期间另一类必要的 CASE 工具是缺陷跟踪工具——用于记录目前没有纠正但已经报告的缺陷。

14.8 节描述了一些 CASE 工具，它们有助于逆向工程和再工程。以可视化方式显示产品结构的工具例子包括 IBM Rational Rose 和 Together。Doxygen 是以 HTML 方式生成文档的开源工具。

缺陷跟踪是交付后维护的一个重要方面，确定当前每个已报告缺陷的状态非常重要。IBM Rational ClearQuest 是个商用的*缺陷跟踪工具*（defect-tracking tool），Bugzilla 是个流行的开源工具。这样的工具可以用来记录缺陷的严重性（14.5.1 节）及所处的状态（特别是缺陷是否修复）。另外，一些缺陷跟踪工具可以实现缺陷报告与配置管理工具相连接，这样在建立新版本时，维护程序员可以选择将特定的缺陷修复报告包括在新版本中。

交付后维护是件困难而且令人恼火的事情。管理部门至少必须做的是给维护团队提供必要的工具，以保证产品维护的效率和效力。

14.11　交付后维护的度量

交付后维护的主要活动包括分析、设计、实现、测试以及文档修订。因此，度量这些活动的测度方法同样适用于维护。例如，13.12.2 节讨论的复杂性测度与交付后维护极其相关，因为复杂度高的代码制品更可能引入回归错误。修改这种代码制品时需要特别注意。

另外，专属交付后维护的测度包括与软件缺陷报告（如缺陷报告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，缺陷报告当前状态相关的信息也是需要的。例如，在 2007 年报告并纠正了 13 个关键缺陷，与在那年只报告了 2 个关键缺陷但都没有得到纠正这两种情况是有本质性的区别。

14.12　交付后维护：MSG 基金会案例研究

在 MSG 基金会案例研究的源代码中已经存在一些错误，另外，必须进行完善性维护。这些维护任务作为练习（习题 14.11 ~ 14.16）。

14.13　交付后维护面临的挑战

这章描述了许多交付后维护面临的挑战。最难改变的是以下现状：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瞧不起，收入经常比开发程序员低。

本章回顾

本章首先比较了开发和维护（14.1 节）。交付后维护是项重要并且具有挑战性的软件活动（14.2 节及 14.3 节），这一点通过 14.4 节的小型案例研究进行说明。关于交付后维护管理有关

的问题在14.5节进行了讨论，包括反复维护的问题（14.5.4节）。面向对象软件的交付后维护问题在14.6节进行了讨论。维护程序员所需的技能与开发人员相同，唯一的差别在于，开发人员可以专长于软件开发过程的某个方面，而维护程序员必须精通软件生产过程的各个方面（14.7节）。14.8节描述了逆向工程。接下来描述了交付后维护的测试问题（14.9节）及交付后维护中的CASE工具（14.10节）。14.11节描述了交付后维护的测度。14.12节讨论了MSG基金会案例研究的交付后维护（留作练习）。本章最后讨论了交付后维护面临的挑战（14.13节）。

延伸阅读材料

交付后维护经典的信息来源是［Lientz and Swanson，1978］，尽管其中一些结果现在受到质疑（参见备忘录1.3）。回归测试用例的选择在［Harrold，Rosenblum，Rothermel and Weyuker，2001］中进行了讨论，［Rothermel，Untch，Chu，and Harrold，2001］讨论了设置回归测试用例的优先级。在［Onoma，Tsai，Poonawala，and Suganuma，1998］中讨论了工业环境下的回归测试。［Autoniol，Cimitile，Di Lucca，and Di Penta，2004］中描述了交付后维护期间人员需求估算的方法。

［Sneed，1995］讨论了再工程的规划，一些有关再工程的文章收录在《IEEE Software》杂志1995年1月刊中。再工程的成本和效益在［Adolph，1996］中进行了讨论。［Charette，Adams，and White，1997］描述了交付后维护框架内的风险管理。［von Mayrhauser and Vana，1997］讨论了大规模软件产品交付后维护的几种程序理解机制。

［Teng，Jeong，and Grover，1998］给了几个成功再工程项目的概要。遗留软件产品的维护在［Bisbal，Lawless，Wu，and Grimson，1999］中进行了描述。［Fioravanti and Nesi，2001］给出了估算适应性维护工作量的测度。［Rajlich，Wilde，Buckellew，and Page，2001］中讨论了遗留系统的理解问题。［Ebner and Kaindl，2002］的主题是再工程领域内可跟踪性的重要性。［Bandi，Vaishnavi，and Turk，2003］中讨论了可维护性领域内测度的应用。［Samoladas，Stamelos，Angelis，and Oikonomou，2005］中给出了开源软件维护中可能出现的问题。

对象的使用对交付后维护的影响在［Henry and Humphrey，1990］和［Mancl and Havanas，1990］中进行了描述。具体面向对象产品的交付后维护在［Lejter，Meyers，and Reiss，1992］和［Wilde，Matthews，and Huitt，1993］中进行了讨论。［Briand，Bunse，and Daly，2001］讨论了面向对象设计的可维护性。在［Prechelt，Unger-Lamprecht，Philippsen，and Tichy，2002］中描述了评估设计模式文档对交付后维护的影响的相关知识。面向对象软件的可维护性在［Lim，Jeong，and Schach，2005］和［Freeman and Schach，2005］中进行了讨论。

《Communications of the ACM》杂志1994年5月刊有软件维护方面的论文。《IEEE Software》1998年7/8月发表了几篇遗留系统方面的论文，尤其是［Rugaber and White，1998］。软件维护年会的论文集有大量的各种各样与维护有关的信息。

习题

14.1　你认为是什么原因使大家经常错误地认为软件交付后维护比不上软件开发？

14.2　假设某种产品是检查计算机是否感染了病毒。描述为什么这个产品的许多代码制品有多个版本。它对交付后维护有什么影响？如何解决这些问题？

14.3　针对习题8.7中的图书馆自动循环系统，重做习题14.2。

14.4　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，重做习题14.2。

14.5　针对习题8.9中提到的自动柜员机，重做习题14.2。

14.6　假设你是一个大型软件公司主管交付后维护的经理。在雇用新员工的时候，你希望他具备哪些素质？

14.7　交付后维护工作对由 1 个人组成的软件产品公司有哪些影响？

14.8　如果要求你建立一份计算机化的缺陷报告文件，你需要在文件中保存哪些类型的数据？你的工具能够提供哪些方面的查询？不能提供哪些方面的查询？

14.9　如果你收到一份来自 Ye Olde Fashioned 公司（习题 13.24）副总裁的备忘录，里面指出在不久的将来，Ye Olde Fashioned 公司将要维护数千万行 COBOL 代码，向你咨询关于采用哪些交付后维护 CASE 工具，你该如何建议？

14.10　（学期项目）假设附录 A 中的 Osric 办公用品和装饰产品按所描述的那样实现。现在 Osric 想要对产品做出修改使得顾客在队列中的优先级能够手动修改。当前产品应该按什么方式进行修改？放弃一切从头开始是否是更好的方案？把答案与习题 1.17 的答案进行对比。

14.11　（案例研究）通过调整不同组件的横向排列，改进 13.7 节的实现报表的外观。

14.12　（案例研究）假定 MSG 基金会的需求发生了改变，使一对夫妇每周付给 MSG 基金会的金额不会再超过周总收入的 26%（而不是当前规定的 28%）。13.7 节有多少处实现需要修改？

14.13　（案例研究）MSG 基金会决定开始基于月来进行操作，而不是基于周进行。修改 13.7 节的相应实现。

14.14　（案例研究）将 13.7 节的实现中菜单驱动的输入例程取代为用户图形接口（GUI）。

14.15　（案例研究）将 13.7 节的实现修改为基于互联网的形式。

14.16　（软件工程读物）教师分发论文［Freeman and Schach，2005］的复印件。阅读并讨论你是否认为该论文给出了面向对象可促进可维护性这个问题的答案？论证你的答案。

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.

[Antoniol, Cimitile, Di Lucca, and Di Penta, 2004] G. Antoniol, A. Cimitile, G. A. Di Lucca, and M. Di Penta, "Assessing Staffing Needs for a Software Maintenance Project through Queuing Simulation," *IEEE Transactions on Software Engineering* **30** (January 2004), pp. 43–58.

[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, Lwawless, Wu, and Grimson, 1999] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy Software Products: Issues and Directions," *IEEE Software* **16** (September/October 1999), pp. 103–111.

[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.

[Freeman and Schach, 2005] G. L. Freeman, Jr., and S. R. Schach, "The Task-Dependent Nature of the Maintenance of Object-Oriented Programs," *Journal of Systems and Software* **76** (May 2005), pp. 195–206.

[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.

[Jacobson, Booch, and Rumbaugh, 1999] I. Jacobson, G. Booch, and J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.

[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.

[Lim, Jeong, and Schach, 2005] J. S. Lim, S. R. Jeong, and S. R. Schach, "An Empirical Investigation of the Impact of the Object-Oriented Paradigm on the Maintainability of Real-World Mission-Critical Software," *Journal of Systems and Software* **77** (August 2005), pp. 131–38.

[Lotto, 1515] L. Lotto, *Giovanni Agostino della Torre and his Son, Niccolò*, oil on canvas, 1515, www.nationalgallery.org.uk/cgi-bin/WebObjects.dll/CollectionPublisher.woa/wa/largeImage?workNumber=NG699.

[Loukides and Oram, 1997] M. K. Loukides and A. Oram, *Programming with GNU Software*, O'Reilly and Associates, Sebastopol, CA, 1997.

[Magnus, 2005] A. P. Magnus "9th Kings, Popes, and Symphonies," www.math.ucl.ac.be/~magnus/onine/const9.jpg.

[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 in 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.

[Samoladas, Stamelos, Angelis, and Oikonomou, 2005] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou, "Open Source Software Development Should Strive for Even Greater Code Maintainability," *Communications of the ACM* **47** (October 2004), pp. 83–87.

[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 1999), 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 Vana, 1997] A. von Mayrhauser and A. M. Vana, "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 15 章　UML 的进一步讨论

学习目标

通过本章学习，读者应能：

- 用 UML 用例、类图、注释、用例图、交互图、状态图、活动图、包、组件图和部署图对软件进行建模。
- 理解 UML 是一种语言而不是一种方法。

本书的前面各章中，已经介绍了 UML［Booch，Rumbaugh，and Jacobson，1999］的不同元素。具体来讲，第 7 章介绍了类图、继承、聚合和关联的概念，第 10 章介绍了用例、用例图和注释，而第 11 章又讲述了状态图、通信图和顺序图。

掌握上面这些概念就足够理解本书内容了，并且能完成所有的习题包括附录 A 中的学期项目。然而，现实世界的软件产品比 MSG 基金会案例和附录 A 中的学期项目更大，而且相对来说也更复杂。因此，为了做好进入现实世界的准备，本章会对 UML 进行进一步的深入讨论。

在学习本章之前，有必要先弄明白什么是 UML，就像所有的尖端计算机语言一样，它是不断变化的。在这本书写作的时候，UML 的最新版本是 2.0。然而，当你阅读本书的时候，UML 的某些方面有可能已经改变了。正如备忘录 3.3 所描述的，UML 现在受对象管理组织控制。在进一步深入学习之前，有必要先去对象管理组织的官方网站（www. omg. org）查看一下更新情况。

15.1　UML 不是一种方法学

在更详细了解 UML 之前，有必要弄清 UML 是什么，更重要的是，UML 不是什么。UML 是统一建模语言（Unified Modeling Language）的缩写。也就是说，UML 是一种语言。作为一种语言（如英语），它可以被用来编写小说、百科全书、诗歌、祷告、新闻报道，甚至是软件工程的教科书。所以，语言是表达思想的一种工具。一种特定的语言不应约束语言所能表达的思想的类型或者表达的方式。

UML 作为一种语言，它可以用来描述用传统范型或者面向对象范型的诸多版本（包括用统一过程）来开发的软件。换句话说，它是一种符号记法，不是一种方法学。它是一种可以与任何方法学一起使用的符号记法。

实际上，UML 不仅仅是一种符号记法，而且是一种特定的符号记法。很难想象现在一本关于软件工程的书不用 UML 来描述软件。UML 已经变成一种世界标准，以至于一个不熟悉 UML 的人将难以成为一个软件专家。

本章的标题是 UML 的进一步讨论。记住 UML 扮演的主要角色，对以后呈现的 UML 是很有必要的。然而，UML 2.0 手册有 1 200 页之长，所以，在本章覆盖 UML 手册全部内容不是一个很好的主意，但是，不知道 UML 的每一个方面，有可能成为一个有竞争力的软件专家吗？

关键是 UML 是一种语言。英语有超过 100 000 的单词，但是几乎所有讲英语的人都只掌握了所有英语词汇的一部分。同样，在本章，会涉及 UML 的所有类型的图，包括其中可以做的一些选择。第 7 章、第 10 章和第 11 章讲述的关于 UML 的一小部分对介绍本书内容已经足够了。同时，本章关于 UML 的更多内容对软件开发和维护已经够用。

15.2　类图

图 15-1 是最简单的类图。它描述的是 **Bank Account Class**，更多的关于 **Bank Account Class** 的细节见图 15-2。UML 关键的地方就是图 15-1 和图 15-2 都是有效的类图。换句话说，在 UML

语言中，对于当前的迭代与增量可把合适的细节添加到 UML 中。

这种自由延伸到了对象的表示。符号**bank account** 可以非正式的用于一个类的特定对象，所以完整的 UML 符号是：

bank account : Bank Account Class

也就是说，**bank account** 是一个对象，是 **Bank Account Class** 的一个实例。更具体地说，下划线的部分代表对象，冒号代表是"……的一个实例"，而首字母大写的粗体字是类。然而，在不引起混淆时，UML 允许使用较短的记法**bank account**

现在假设要对一个任意的银行账户建模。也就是说不希望它是银行账户类（Bank Account Class）的一个特定的对象。这样的 UML 符号是：

: Bank Account Class

正如刚才指出的，冒号是指"……的一个实例"，所以**: Bank Account Class** 意思是 **Bank Account Class** 的一个实例。这正是所要的。这个符号在第 11 章经常用到。相反的，在图 11-48，对 **MSG** 基金会软件产品的用例 Update Estimated Annual Operating Expenses 所画的通信图中，参与者被标识为 **MSG Staff Member** 而不是**: MSG Staff Member**。这是因为 **MSG Staff Member** 是一个参与者，而**: MSG Staff Member** 表示的是（一个不存在的）类 **MSG Staff Member Class** 的一个实例。

7.6 节介绍了信息隐藏的概念。在 UML 中，前缀"+"代表一个属性或者方法是 **public** 的，同样的，前缀"-"表示一个属性或者方法是 **private** 的，如图 15-3 采用了这种符号。**Bank Account Class** 的属性被声明是 **private** 的（因此里面的信息是隐藏的），然而另外两个操作都是 **public** 的，所以可以在任何地方调用它。第三种标准的可见性是 **protected**，用前缀#来表示。如果属性是 **public** 的，在任何地方可见，如果是 **private** 的，则仅在定义它的类的内部可见。而如果是 **protected** 的，就在定义它的内部类可见，同时也在这个类的子类可见。

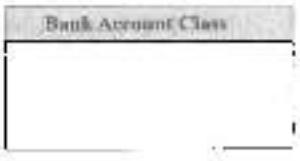


图 15-1　最简单的类图

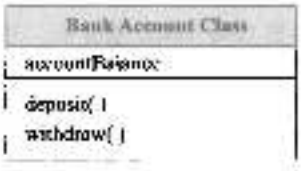


图 15-2　在图 15-1 之上添加了一个属性和 2 个方法

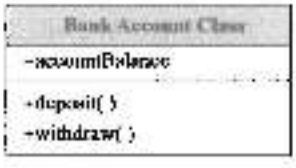


图 15-3　对图 15-2 加了可见性的记号

本章到此为止，类图仅包含一个类。15.2.1 节将介绍具有多个类的类图。

15.2.1　聚合

一辆汽车，由底盘、引擎、车轮和座位组成，图 15-4 表示对它的建模。前面章节已经介绍空心菱形代表聚合。聚合是部分整体关系（part-whole relationship）的 UML 术语。底盘、引擎，车轮，座位都是车的一部分。这个空心菱形指向整体（汽车），而不是部分（底盘、引擎、车轮、座位），线连接到整体的每一个部分。

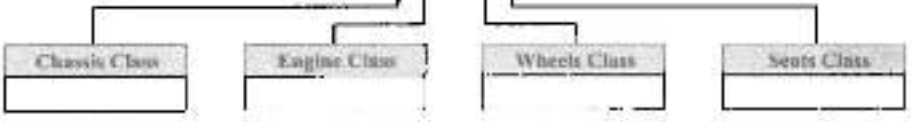


图 15-4　一个聚合的例子

15.2.2　多重性

假设现在要对这样一辆车建模。它有 1 个底盘、一个引擎、4 个或者 5 个车轮，可选的天窗、0 个或者多个悬挂在后视镜上的广角镜，还有 2 张或者更多的座位。如图 15-5 所示。线两端的数字就代表多重性（multiplicity），数字表示一个类与另外一个类关联的的次数。

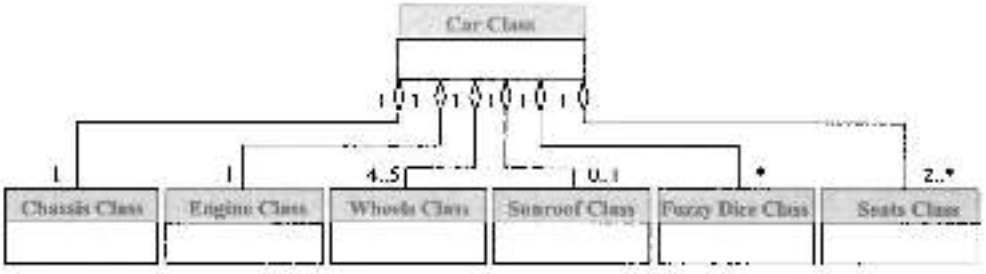


图 15-5　具有多重性聚合例子

先来看 **Chassis Class** 连到 **Car Class** 这根线，在“部分”那端的那个 1 表示在这个关联中涉及 1 个底盘，而在“整体”那端的那个 1 表示 1 辆车，也就是说每辆车都只有 1 个底盘。同样 **Engine Class** 到 **Car Class** 也是。

现在来看 **Wheels Class** 到 **Car Class** 这条线，“整体”端是 1，“部分”端是 4..5 表示每辆车有 4 ~5 个轮子（第 5 个轮子是备用胎）。因为类的实例成批地创建，意思是，这个 UML 模型是对 1 辆有 4 个轮子或者按需要对有 5 个轮子的车建模

一般的，两点.. 表示范围。由此 0..1 表示 0 个或者 1 个，这就是 UML 中表示可选的方式。这也就是为什么在 **Sun Roof Class** 到 **Car Class** 的线上有 0..1。

现在来看连接 **Fuzzy Class** 到 **Car Class** 的那条线。在“部分”端是 *。星号本身意味着 0 个或者多个。因此，图 15-5 中的 * 意味着一辆车没有或者有多个广角镜挂在后视镜上。（如果想更清楚的了解星号，参看备忘录 15.1）

> **备忘录 15.1**
>
> Stephen Kleene 是对计算机科学有着重要影响的数理逻辑的一个分支递归函数论的奠基者。Kleene star（在图 15-5 中表示 0 个或者多个的星号）就是以他的名字命名的。Kleene star 在数学界和计算机界中都很有名。而相对很少有人知道的是，kleene 的发音为“clay knee”（重音在第一个音节上）而不是“clean knee”。

再来看连接 **Seats Class** 到 **Car Class** 的这条线。这次“部分”端是 2..*，星号本身表示 0 个或者多个，有范围的星号表示有多个。与之对应，图 15-5 中的 2..* 表示一辆车有 2 个或者多个座位。

因此，在 UML 中，如果知道确切的多重数，就使用数字。图 15-5 中 8 个位置出现的 1 就是一个例子。如果范围知道的话，范围符号就会被用到，如 0..1 和 4..5。还有如果个数不确定，星号就会被用到。如果范围的上界不知道，范围符号可以和星号一起使用，如图 15-5 中的 2..*。顺便提一下，UML 中多重性的符号是基于传统数据库理论的实体关系图的。

15.2.3　组合

图 15-6 是另一个聚合的例子，它是对棋盘和棋盘上的方格建模。每个棋盘有 64 个方格，事实上，进一步，这个关系是一个组合（composition）的例子，组合是聚合的更强的形式。就像前面所说的，对部分整体关联关系建模，当存在组合的时候，每个部分只属于一个整体，如果整体被删除了，部分也要被删除。上述例子中，如果有很多不同的棋盘，每个方格仅属于一个棋盘，如果棋

盘没了，那么64个方格也就同时没了。组合是聚合的扩展，用实心菱形表示，如图15-7所示。

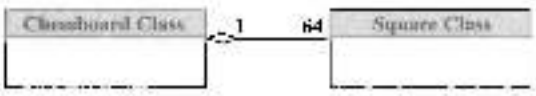


图15-6　另一个聚合的例子（也请参阅图15-7）



图15-7　组合的例子

15.2.4　泛化

继承是面向对象的一个非常重要特征，也是泛化的一个特例。UML关于泛化（generalization）的符号是个空心三角形，有时用一个区别符（discriminator）来标识这个空心三角。如图15-8所示，对两种投资类型建模：证券和股票。空心三角旁边的InvestmentType表示每个**Investment Class**的实例或者它的子类都有InvestmentType属性，而这个属性可以用来区分证券和股票的实例。

15.2.5　关联

在7.7节中，讲述了2个类关联（association）的例子，关联的方向是通过一个导航箭头来区别的，这个导航箭头也是实心三角。图15-9是图7-32的重新绘制。

在有些情况下，两个类之间的关联本身需要被建模成一个类。例如，假设图15-9中的放射线学者会在很多不同的场合咨询律师，每种场合时间长度都不同。为了能使律师正确收取放射线学者的费用，诸如图15-10的类图是必须的。现在consults也成为了一个类，即**Consults Class**，叫做关联类（association Class）（因为它既是一个关联也是一个类）。

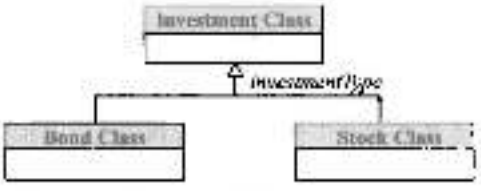


图15-8　带有区别符的泛化（继承）实例



图15-9　一个关联

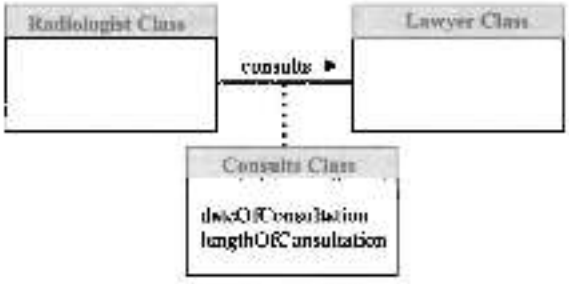


图15-10　一个关联类

15.3　注释

当需要对**UML**图添加注解的时候，只需要把它放入注释框（note）中（一个右上角带折线的矩形）。虚线指向要注释的条目。图11-38给出了个注释的示例。

15.4　用例图

正如10.4.3节所说，用例（use case）是软件产品外部角色和软件产品本身的交互模型。更确切地说，参与者（actor）是担负特定任务的用户，用例图（use-case diagram）是用例的集合。

在10.4.3节中，描述了参与者语境中的泛化，如图10-2所示。图15-11是另外一个例子，它描绘了**Manager**是**Employee**的一个特例。基于这些类，空心箭头表示指向更一般的用例。

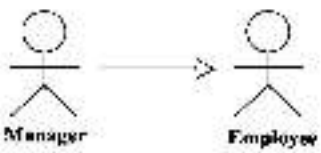


图15-11　参与者的泛化

15.5　构造型

美国个人收入税的三种主要表格是 1040、1040A 和 1040EZ 表。图 15-12 中用例 Prepare Form 1040、Prepare Form 1040A 和 Prepare Form 1040EZ 都包含了用例 Print Tax Form，从图中构造型给出的 **include** 关系可看出这一点。

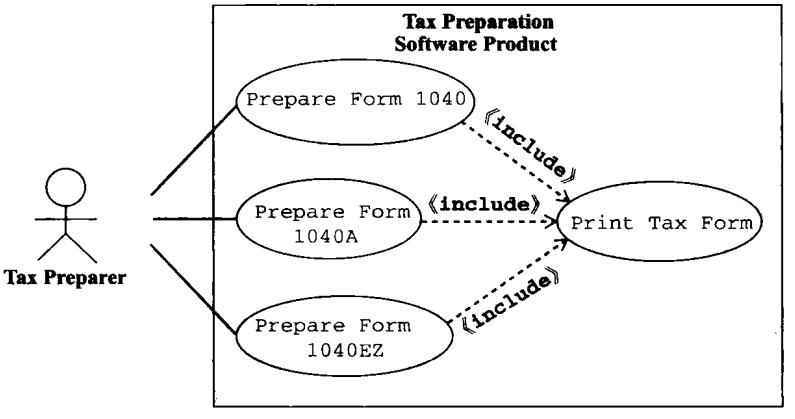


图 15-12　用例 Prepare Form 1040、Prepare Form 1040A 以及 Prepare Form 1040EZ 都包含用例 Print Tax Form

在 UML 中的构造型（stereotype）是对 UML 扩展的一种方式。也就是说，如果需要定义一种 UML 中没有的构造型，可以用这种方式。在第 11 章中讲述了 3 种构造型：边缘类、控制类和实体类。总的来说，构造型的名字出现在书名号的中间，例如，《**this is my construct**》。因此，可以用标准类的矩形符号，并在里面标识《**boundary class**》来表示边缘类，同样，也可以表示控制类和实体类，而不用任何特殊的符号。

图 15-12 中的那个 **include** 关系就是 UML 中的一种构造型。该图中标注《**include**》表示公用功能，在这个例子中就是用例 Print Tax Form。另一个关系是 **extend** 关系（relationship），表示某个用例是一个标准用例的变种。例如，可以希望有一个个别用例，一个客户预订了一个夹饼但不要油煎的那种。《**extend**》符号就可以用于这种目的，如图 15-13 所示。然而，对于这个关系，开口箭头指向另一个方向。

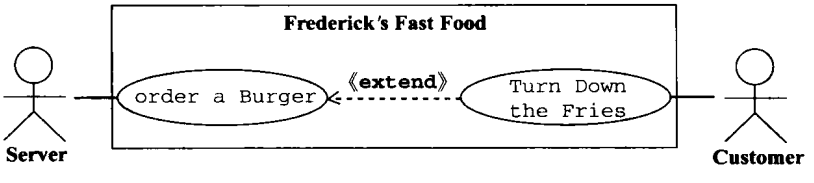


图 15-13　用例 Order a Burger 表示用户取消油煎时的变种

15.6　交互图

交互图（interaction diagram）描述的是软件产品中一个对象与另一个的交互。第 11 章介绍了两种类型的 UML 交互图：顺序图和通信图。

首先来看顺序图（sequence diagram）。假设一个人在网上订了一个东西，但当包括了销售税和邮递费的总费用算出来后，购买者觉得价格太高而决定取消订单。图 15-14 描述了这个动

态的创建过程和取消订单的过程。

1）考虑图 15-14 中的生命线。当一个对象是活动的时候，用虚线上的窄矩形框（激活框，**activation box**）表示。例如，**:Price Class** 在消息 5 Determine price of order 传递到至消息 6 Return price 发出的时候是活动的，其他对象也是同样的。

2）**:Order Class** 对象是在当**:Assemble Order Control Class** 发送消息 3 Create order 到**:Order Class** 的时候创建的。就用它来表示动态创建的生命周期。

3）图 15-14 还描述了对象**:Order Class** 在收到消息 9 Destroy order 后的销毁。用 × 来表示销毁。

4）销毁发生在一个返回值返回之后。在事件 9 下面用一个带开箭头的水平虚线表示。在顺序图的其余部分，每个消息都最终跟随一个返回消息，用来发送给初始发送消息的对象。事实上，接受到内容是可有可无的，一个消息发送出去但最终没有受到任何回复也是有效的。即使如果有了回复，也没有必要发回特定的新消息。相反，用带开箭头的虚线（返回，return）来表示是对最初信息的返回，跟新的消息相对。

5）在消息 9 的地方有一个保护条件：[price too high] Destroy order。也就是说，只有在消费者觉得价格太高而决定不买商品时消息 9 才发送。保护（guard）是个值可真可假的条件，只有真的时候消息才被发送。在 15.7 节中，状态图中描述了保护条件，但现在只用在顺序图中。

（在图 15-14 中，消息 9：[price too high] Destroy order 应该从 Buyer 发送给**:User interface Class** 对象，后者再向**:Assemble Order Control Class** 发送一个消息，接下来，**:Assemble Order Control Class** 对象给**:Order Class** 对象发送一个消息，指示它取消订单。为了突出对象的动态析构，这些细节已从图 15-14 中去掉。）

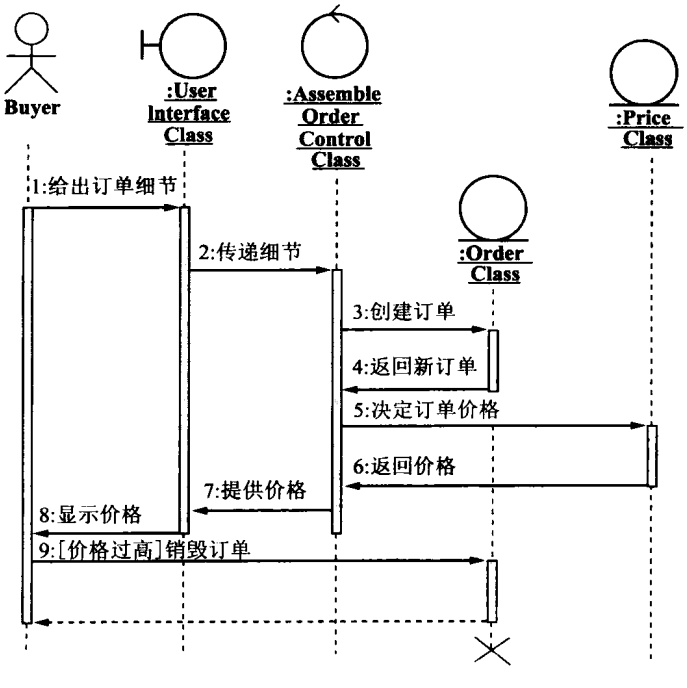


图 15-14　顺序图显示一个对象的动态创建和析构，返回以及显示的激活

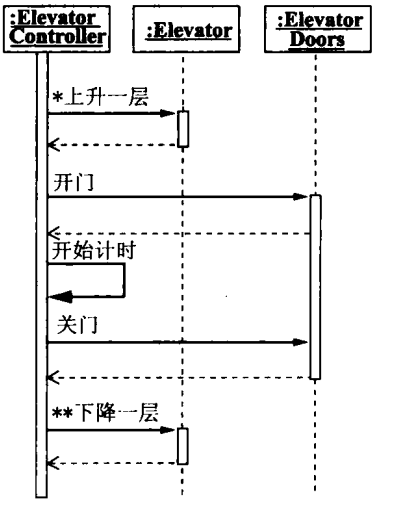


图 15-15　显示迭代和自调用的顺序图

UML 交互图还支持很多其他的选项。例如，假设对一个上升的电梯建模，提前并不知道到几层的按钮被按下，所以不知道要上升到几楼。如图 15-15，通过标注 `*move up one floor` 消

息来建模这个迭代。星号就是 Kleene star（参看备忘录 15.1）。所以这个星号的意思就是上升零层或者多层。

对象可以对自己发送信息，这被叫做自调用（self-call）。例如，假设电梯到了某一层，电梯控制器发送信息要求打开门。一旦这个返回消息收到，电梯控制器发送一个消息给自己来重启计时器，见图 15-15。在计时满后，电梯控制器发送消息要关门。当第二条返回消息收到的时候（也就是门已安全关上），电梯才被命令开始启动。

现在再来看通信图（communication diagram）。在 11.18 节提到，通信图和交互图是等价的，所以如图 11-33 所示，所有表现在交互图中的特征在通信图中同样适用。

15.7　状态图

考虑图 15-16 的状态图（statechart），它和图 11-22 的状态图很相似，但它是采用保护条件建模而不是事件。图中，一个表示初始状态的实心圆通过没有标注的转移（transition）指向了 **MSG Foundation Event Loop** 状态，然后又有 5 个转移从这个状态出发，这 5 个状态都有一个保护条件，值或为真或为假。当其中一个保护条件是真的时候，该转移将发生。

事件（event）也能引起状态间转移。一个常见的事件是接收一条消息。如图 15-17，描述了电梯状态图的一部分。这个电梯在状态 **Elevator Moving**，当保护条件 [no message received yet] 一直为真时，它由 Move up one floor 来保持电梯移动，直到收到消息 Elevator has arrived at floor。这个消息的接收使得保护条件返回假，同时状态转移到 **Stopped At Floor**。在这个状态下，活动 Open the elevator doors 被执行。

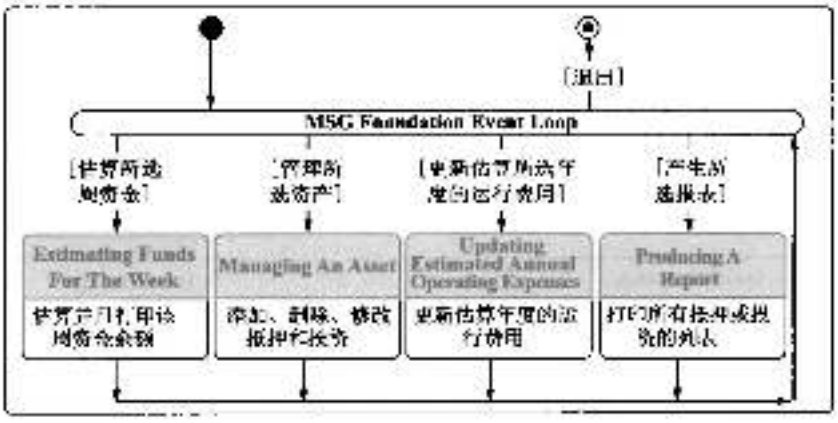


图 15-16　案例 MSG 基金会案例研究的状态图

到目前为止，转移标签是以保护条件或事件的形式。事实上，转移标签更一般形式是

事件 [保护条件] /动作

也就是说，如果事件发生了，而且保护条件是正确的，那么转移就发生了。发生的同时，动作也被执行了。这样一个转移标签的例子如图 15-18，和图 15-17 等价。转移标签是 Elevator has arrived at floor [a message has been received] /Open the elevator doors。当事件 Elevator has arrive at floor 发生的时候，同时消息被有效发送时，保护条件 [a message has been arrived] 返回真，动作（action）Open the elevator doors 被执行，通过斜杠 "/" 来表示。

对比图 15-17 和图 15-18，发现状态图中动作执行有 2 个地方。第一个，反映在图 15-17 的状态 **Stopped At Floor**，当进入了这个状态的时候动作被执行。这样的动作在 UML 中被称为活动（ac-

tivity)。第二个，如图 15-18 中，动作作为转移的一部分被执行。（从技术上讲，在动作和活动间略微有些不同，动作假定本来要在瞬间发生，但活动可能不那么快发生，也许要在几秒后发生。）

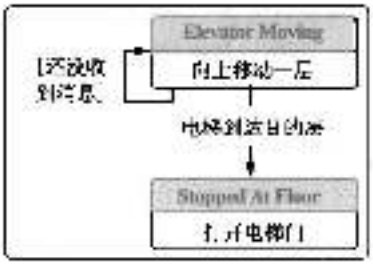


图 15-17　电梯状态图的一部分

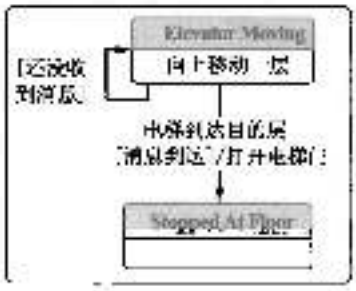
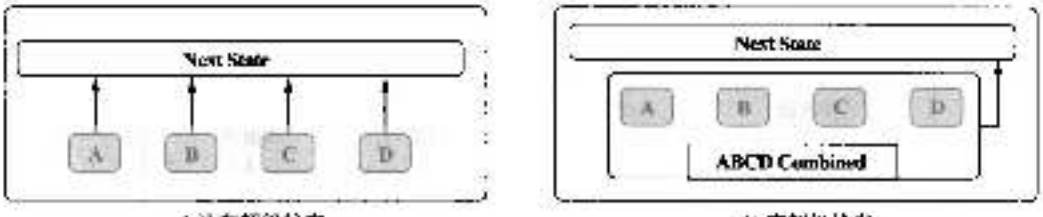


图 15-18　与图 15-17 等价的状态图

在状态图中 UML 支持很多不同种类的动作和事件。例如，一个事件可以用 When 或者 After 来表示。因此，一个事件可能在 when（成本 >1000）或 after（2.5 秒）时发生。

一个具有很多状态的状态图就有很多转移。这么多表示转移的箭头使得状态图看起来像一人就意大利面。一个解决这个问题的技术就是用超级状态（superstate）。例如，在图 15-19a 中，4 个状态 A、B、C、D 都有转移到 **Next State**。如图 15-19b 所示，4 个状态被合成了一个超级状态 **ABCD Combined**，这时只需要一个转移，从而把 4 个转移的箭头减少到了 1 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就不受影响，也没有现有的转移进入那些状态。另外一个超级状态的例子是图 15-20，其中图 15-16 中的 4 个低级别状态被合成了一个超级状态 **MSG Foundation Combined**，使得图更加清晰和干净。



a) 没有超级状态　　b) 有超级状态

图 15-19　状态图

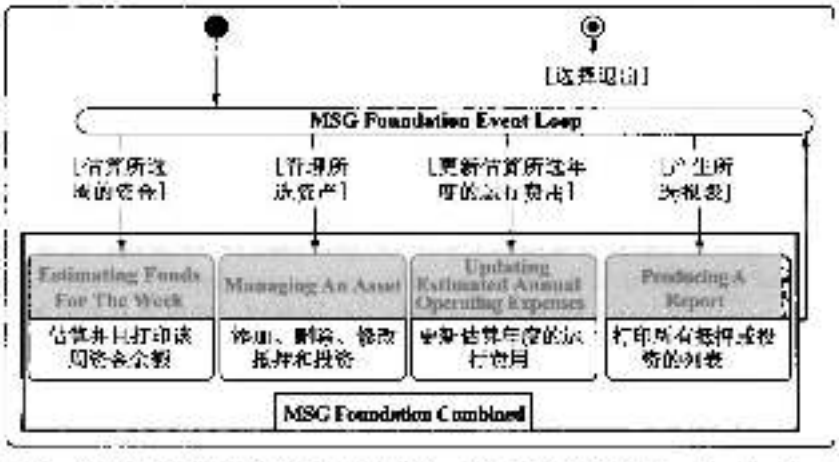


图 15-20　带有 4 个状态的图 15-16 合并成为一个超级状态 **MSG Foundation Combined**

15.8　活动图

活动图（activity diagram）表示的是不同的事件间的协调，在活动并行进行的时候经常被用到。

假如一对夫妻在一个餐馆里点菜。一个点了鸡肉，另一个点了鱼。服务员写下了他们的订单后把它交给厨师，这时厨师才知道该准备什么菜。这个时候无所谓哪个菜先做好，因为就像图 15-21 所示的那样，只有两个菜都做好的时候才上菜。上面那条粗的水平线叫交叉（fork），下面那条叫结合（join）。一般地，交叉仅有一个输入转移和多个输出转移，每一个转移开始一个与他活动并行执行的活动。相反，结合有很多输入转移，它们也是并行执行的，而那个输出转移要等那些并行活动都结束了以后才执行。

活动图在有很多活动并行执行的业务逻辑建模很有用。例如，一家为顾客组装指定配置电脑的公司，如图 15-22 的活动图，当接收到订单的时候，它被发送到了装配部门（Assembly Department），同时也被发送到了结算部门（Accounts Receivable Department）。当电脑被组装好并交付的时候，订单才完成，然后顾客的支付被处理。这里牵涉到三个部门：组装部门（Assembly Department）、订货部门（Order Department）和结算部门（Accounts Receivable Department），每个都在自己的泳道（swimlane）内完成。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些部门、哪些任务并行完成，以及哪些任务在下一个任务开始之前必须要完成。

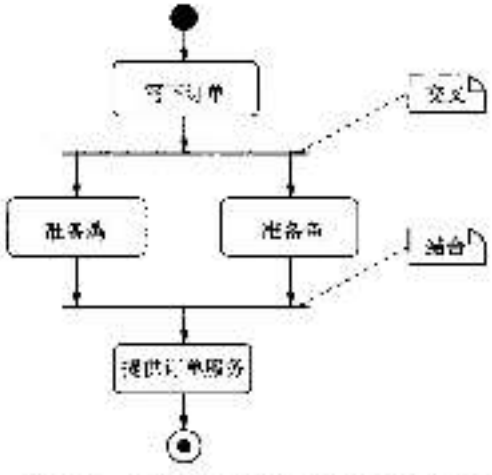


图 15-21　餐馆为两个并发人员订餐活动图

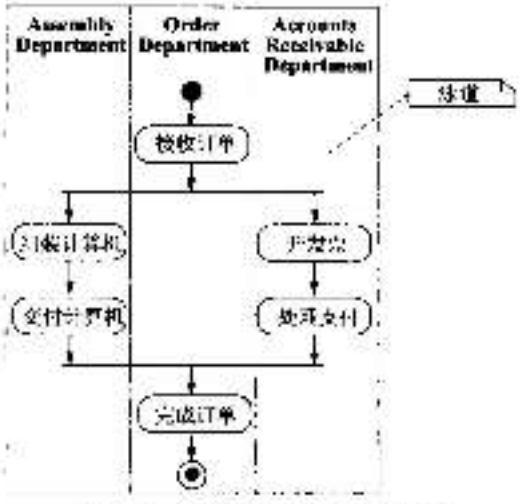


图 15-22　计算机组装公司的活动图

15.9　包

正如 12.4 节所说的，处理一个较大的软件产品的方法是把它分解成一个个相对独立的包（package）。UML 中包的符号是一个矩形加一个名字标签，正如图 15-23 所示的那样。图中 My Package 是一个包，但矩形是空的。这是一个有效的 UML 图，这个图只是简单表达 My Package 是一个包。图 15-24 更有趣一点是，它里面还有其他的内容，包括类、实体类和另外一个包。还可以继续提供更多的细节，直到这个包对当前的迭代和增加来说已经足够了。

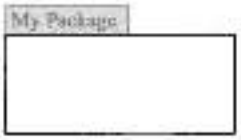


图 15-23　包的 UML 表示

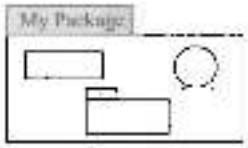


图 15-24　图 15-23 的更详细显示的包

15.10　组件图

组件图（component diagram）表示的是软件组件的依赖，包括了源代码、编译后的代码和可执行的载入映像。例如，图 15-25 的组件图就有源代码（由一个便笺代表）和由它产生的可执行的载入映像。

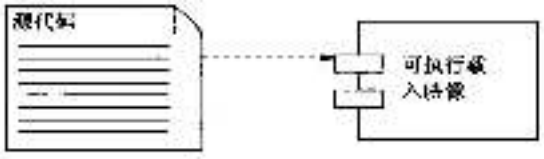


图 15-25　组件图

15.11　部署图

部署图（deployment diagram）表示了每个软件组件安装（或部署）在哪个硬件组件上。它也表示了硬件组件之间的通信链接。一个简单的部署图如图 15-26 所示。

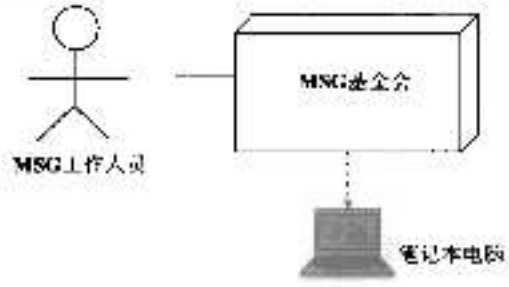


图 15-26　部署图

15.12　UML 图回顾

这章讲了很多不同类的 UML 图。为了理解更清楚一些，下面列举了一些可能感到困惑的图表类型：

- 用例对参与者（软件产品的外部用户）与软件产品本身之间的交互进行建模。
- 用例图是一个包括多个用例的图。
- 类图对类进行建模，同时表示类之间的静态关系，包括关联和泛化。
- 状态图描述的是状态（对象的特定属性值），引起状态转移的事件及对象所做的动作和活动。状态图是一个动态模型，反应了对象的行为，也就是它们对特定事件的反应。
- 交互图（顺序图和通信图）描述的是一个对象和另一个对象之间以信息传递的方式进行交互。这是另外一个动态模型，显示对象是怎么样行动的。
- 活动图描述的是同时发生的事件是怎么样协调的。这也是另外一个动态模型。

15.13　UML 和迭代

对于状态图，一个转移可以用保护条件、事件、动作或者三个一起来标注。对于顺序图，生命线可以包括也可以不包括激活框，可能有也可能没有返回值，可能有也可能没有对消息的保护条件。

每个 UML 图都有很多不同的可选项，也就是说，一个有效的 UML 图由一些必要部分和任意可选项组成。UML 图有这么多的可选项主要有两个原因，首先，不是每个 UML 的特性对每个软件产品都有用，所以对这些可选项的选择是自由的；其次，除非可以逐步添加对图有用的特性，否则不能对统一过程执行迭代和递增，而不是在开始就创建完整的图。也就是说，UML 允许以一个基本的图开始，之后添加需要的额外特性，记住，任何时刻 UML 图都是有效的。这也是 UML 图适合统一过程的原因之一。

本章回顾

15.1 节已经介绍了 UML 是一种语言而不是一种方法学。15.2 节描述了类图，讨论了类图的多个方面，包括聚集（15.2.1 节）、多重性（15.2.2 节）、组合（15.2.3 节）、泛化（15.2.4 节）和关联（15.2.5 节）。接下来，给出了多种 UML 图，包括注释（15.3 节）、用例图（15.4 节）、构造型（15.5 节）、交互图（包括顺序图和通信图，15.6 节）、状态图（15.7 节）、活动图（15.8 节）、包（15.9 节）、组件图（15.10 节）和部署图（15.11 节）。本章是对 UML 图的一个总览（15.12 节），并讨论了为什么 UML 对统一过程适用（15.13 节）。

延伸阅读材料

没有比查看 UML 手册的当前版本更好的选择了，读者可以在 OMG 的官方网站 www.omg.org 上获得 UML 手册。两本较好的介绍 UML 的书籍是［Flower and Scott，2000］和［Stevens and Pooley，2000］。

习题

15.1　使用 UML 对飞机场建模（提示：不需要很详细，但要能很好地回答这个题目）。

15.2　用 UML 对巧克力蛋糕建模。这个巧克力蛋糕由鸡蛋、面粉、糖、发酵粉、牛奶和可可粉制成。巧克力蛋糕先搅拌、烘烤、冰冻，然后吃掉。为了防止没有授权的人烘烤巧克力蛋糕，配料是私有的，除了最后一步外，每个步骤都是如此。

15.3　用 UML 图对饭厅建模。每个饭厅有一张桌子、4 把椅子和一个餐具柜，还有一个可选的烤炉。

15.4　用聚合和组合来对习题 15.3 的饭厅问题建模。

15.5　修改习题 15.3 的结果，以反映饭厅只是房间的一种。

15.6　对习题 15.2 添加注释，指出你的蛋糕是巧克力蛋糕。

15.7　采用 UML 对 1952 年 John Cage 创作的颇具争议的标题为 4′33″的钢琴曲进行建模。乐曲包括了三个无声乐章，长度分别为 30 秒、2 分 23 秒和 1 分 40 秒（标题来源于它的总长度）。他走上舞台，手持秒表和乐谱（使用常规的音乐符号，但是各小节是空白的）。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乐章的开始。在该乐章结束时（即在 30 秒内钢琴家小心地跟随着空白乐谱静默，必要的时候翻动谱页），抬起钢琴的盖子表示第一乐章的结束。在第二乐章（2 分 23 秒）和第三乐章（1 分 40 秒）重复这些动作。最后他合上乐谱，收起秒表和乐谱，起身离开舞台。

参考文献

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

[Date, 2003] C. J. Date, An Introduction to Database Systems, 8th ed., Addison-Wesley, Reading, MA, 2003.

[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：Osric 办公用品和装饰公司项目

Osric 办公用品和装饰（OOA&D）公司归 Osric Ormondsey 公司所有。Osric 在装修高级商业经理办公室方面拥有相当成功的经验，它雇用了大量的转包商和技术员，这样它就可以提供包括电信（电话、传真、高速数据链接等）服务在内的一整套承包业务。其成功可以大致归功于它响应客户需求的快速性，只需要两天时间它就可以完成一个大的执行经理的套间的装修，并因此而闻名。

相比他们多次和当地电话公司技术员接触的糟糕经历，Osric 的客户经常热情洋溢地赞扬 Osric 的电信技术员的精湛技术。由此 Osric 认识到它能够通过增加一个电信部门来扩展它的业务。它决定雇用能找到的最好的技术员，给他们提供丰厚的薪水和红利，并向已经装饰过办公室的执行官宣传他们的服务。那些执行官们认为付给 Osric 公司高额的费用，这样就能让 Osric 技术精湛的技术员在不到一两小时的时间赶到并快速解决问题，这远比花费两到三天等来一个技能并不全面的技术员把问题弄糟要高效。

Osric 那个其他人无论如何也想不到的创意已经成功。他们的技术员总是处于被高度需求的情况，以至于尽管 Osric 的技术员每天 24 小时电话在线，但等待一个 OOA&D 技术员有时要耗费两天以上的时间。这对于不论 Osrich 还是它的客户都是不可接受的，因此 Osric 试图调整他们的方案。由于不能雇用到足够的具备所需高标准的技术水平的技术员，所以他决定分发服务。具体来说，他决定给用户制定优先级，这样当有人需要服务时，Osric 就根据优先级决定把谁放在等待技术员的清单之上。

每个客户公司都有一个 5 位的客户编号及被赋予的优先级。

优先级 4：已经雇用 Osric 装修其经理办公室的公司。

优先级 3：之前有 3 次以上电话通信服务要求的公司。

优先级 2：以前有 1 ~ 2 次电话通信服务需求的公司。

优先级 1：第一次提出需求的公司。

当有人打进电话需求服务时，助理就会询问打入电话者他或她们公司的客户编号。如果客户不知道这个客户编号，助理就会问这个公司的名字。如果名字也找不到，软件就认为这是一个新客户。新客户只有被放在等待清单上时才会被分配到一个编号（请看下面）。

当有服务需求的时候，这个公司就会被添加到等待清单上。在等待清单上的公司按优先级排队，在相同优先级内根据电话进来的日期和时间。

Osric 的技术员 8 小时轮班工作。它目前有 7 名技术员白天工作，2 名技术员隔天夜班，但这也可以有所变动。Osric 观察到业务电话的持续平均时间通常是 5.5 小时，标准偏差为 9.8 个

小时。而完成工作的最短时间是 1.9 小时，最长为 23.1 小时[⊖]。早上 8 点时，每个技术员将分别分配到清单最上面的 7 个客户之一，中午 12 点的情况类似，如果存在未完成的工作，按下段所描述的方式进行处理。

如果一个工作在指定的 4 小时之内完成了，客户将按 4 小时付费，技术员返回 OOA&D。如果工作在中午 12 点未完成，就分配下一个 4 小时给这名技术员去完成任务，这就减少了可以进行新工作的技术员的数量。

Osric 提供一天 24 小时的服务。到了晚上，值班助理不再接受新服务需求。他的唯一任务就是监督晚班技术员。在下午 4 点，值班助理检查工作是否未完成。如果未完成，他就打电话问公司是否需要夜间技术员去继续工作（费用翻倍），或者是否他们需要同样的技术员下一个早上再继续工作。因为夜间技术员比白天的要少，就有可能不能够完成夜间的所有需求。因此，助理打电话之间问那些等待清单上需要特别工作的公司。这也是按照优先级的，若优先级相同，就按照打来电话的日期和时间的先后。

同样地，工作在夜间也是按 4 小时区间进行分配和管理的。如果需要，夜间工作可以在下一天继续，这时按白天费用付费。如果这样，同样的白天技术员将会被分配到这项工作。

一个公司在等待清单上等了 2 个白天之后，它的优先级在下一个工作分配给技术员之前会临时被提升 1。于是，4 级用户通常会在 4 小时之内得到服务，但新客户（优先级为 1）可能会等到第 4 天才得到服务。

当一个客户打进电话请求服务时，软件将估算出客户获得服务的时间，主要基于以下几个方面的考虑：到现在为止每个工作所花费的时间、目前等待清单的长度和完成一项工作的平均时间。根据该公司在接受服务前达到优先级为 4 的级别的时间，软件也会给出一个最糟糕情况下的估计。助理把这个信息递送给客户，并询问客户如果需要，是否准备等待。每天早上助理都打电话给所有的等待清单上的客户，通知他们需要等待服务的时间。同样，这个信息由产品通过当前时间所获得的信息计算出来。

如果一个新公司需要服务，必须首先得到一名经理的允许。通常情况下，如果在等待清单上客户的数目超过了白天技术员数目的 2 倍，这样的申请将不被批准。偶尔 Osric 会增加一个特定的新公司到它的客户列表。因此，经理决定是否增加新客户，在助理打电话时，经理及即时给予答复。

当技术员完成了客户服务，他要把工作卡交给助理，助理输入客户编号以及完成工作需要的白天和夜间的区间数目。如果可行，软件首先会使用这个信息去更新公司的优先级，然后使用该数据生成一份账单并通过邮件发给客户。一个区间的价格是白天 480 美元，晚上 960 美元。Osric 的付账地址是：Suite 16、Kronborg Castle、Helsingør、Sjælland、Denmark。（参见备忘录 A.1）

⊖ 许多编程语言提供伪随机数生成器。在 Java 语言中，函数 `nextGaussian` 生成了呈正态分布的伪随机数，其平均值是 0，标准差是 1。在C++ 语言中，函数 `rand()` 生成 0 和 1 之间呈均匀分布的数字；Box-Muller 转换可被用作生成正态分布的伪随机数（参考 G. E. P. Box 和 M. E. Muller 所著的 "A Note on the Generation of Random Normal Deviates"，*Annuals of Mathematical Statistics* 29（1958），pp. 610-611）。另外，许多免费的用于C++ 语言的高斯伪随机数生成器可以从网络下载，如 GNU 科学实验室（GSL）的 `gsl_ran_Gaussian`。
如果 x 是一个取自标准正态分布的伪随机数（平均值为 0，标准差为 1），$\mu + x * \sigma$ 就符合平均值为 μ、标准差为 σ 的正态分布。界外的数（太大或者太小的数）将被丢弃。

备忘录 A.1

纨绔侍臣 Osric 的首次亮相是莎士比亚的悲剧“丹麦王子哈姆雷特”的最后一场（第五幕，第二场），场景是 Helsingør（英文称为 Elsinore）的哈姆雷特城堡。本书作者总是为扮演 Osric 的演员感到惋惜，在全剧的前三个小时他没有任何事情可做，只有一句台词（“A hit，a very palpable hit ”），并被 Hamlet 和 Laertes 无情嘲弄。Tom Stoppard 的“哈姆雷特”版本将 Hamlet 的朋友 Rosencrantz 和 Guildenstern 作为主角，剧目名为“Rosencrantz 和 Guildenstern 之死”。出于同样的想法，学期项目取 Osric 作为主角。

你需要通过模拟仿真来确定 Osric 方案的有效性。首先，生成一个工作混合集，即一个工作集。对于工作集中的任何工作，明确必要的属性，包括打进电话的时间、公司优先级和持续时间，可能还需要其他属性。对于每个打电话进来申请服务的工作，添加其到队列中。在每个工作区间的开始，工作被移除出队列，并分配给可以工作的技术员。当工作完成，技术员结束服务，或者接着开始下一个区间的工作，或者回家休息。

为了得到方案的有效性证明，需要保存特定的统计数据，包括工作开始之前的平均等待时间、平均队列长度、白天和晚上队列为空的百分比、技术员空闲时区间的数量（这时不会给 OOA&D 带来受益）和因夜间没有技术员可用而造成的不能继续工作的数目。

现在，运用同样的工作集，不使用 Osric 的方案而计算同一统计数据，即在纯粹先来先服务（新用户也是如此）的基础上，确定统计数据。决定不同优先级下客户的平均等待时间。对不同的服务工作集和不同的技术员人数重复该工作。确定 Osric 的方案是否对 Osric 案例有效，以及是否可减少客户的等待时间。

附录 B 软件工程资源

有两种很好的获取更多的软件工程方面信息的途径：阅读更多的期刊文章和会议论文集；访问因特网和万维网。

已有的专注于软件工程的期刊（如《IEEE Transactions on Software Engineering》）以及一些主题更加宽广的期刊（如《Communication of the ACM》），不少软件工程方面的重要文章在其上发表。由于空间有限，此处仅列出经过筛选的部分刊物。这些刊物是依个人观点挑选的，也是作者目前认为最有用的刊物。

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 Engineering 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

Software Quality Journal

另外许多会议论文集包含了软件工程的重要文章。下面列出经过筛选的部分会议，会议以赞助商的首字母或名字命名，会议简称在括号中。

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)

因特网是另一个很有价值的软件工程方面的信息来源。下面列出两个一向对作者非常有用的 Usenet 新闻组：

comp. object

comp. software-eng

下面几个新闻组，有时也能发现相关的内容：

comp. lang. C++. moderated

comp. lang. java. programmer

comp. risks

comp. software. config-mgmt

附录 C 需求工作流：MSG 基金会案例研究

MSG 基金会案例的需求工作流在本书第 10 章介绍。

附录 D 分析工作流：MSG 基金会案例研究

分析工作流在第 11 章描述。

附录 E 软件工程管理计划：MSG 基金会案例研究

此计划是为一个小型软件组织开发 MSG 产品而制定的。该组织包含 3 个成员：Almaviva（公司的所有者）以及两个软件工程师（Bartolo 和 Cherubini）。

1 总览

1.1 项目概要

1.1.1 目的、适用范围和对象

这个项目是要开发一个软件产品，协助 Martha Stockton Greengage（MSG）基金会为已婚

人群提供住房抵押服务时做出决策。这个产品允许用户添加、修改、删除基金投资、操作成本及个人抵押等方面的信息。产品将执行所需的计算，产生投资、抵押、及每周操作成本的报表。

1.1.2 假设和约束

约束条件包含以下几条：
在截止日期前完成。
符合预算限制。
产品必须可靠。
架构必须是开放式的，将来可能会添加额外的功能。
产品必须是用户友好的。

1.1.3 项目交付日期

完整的产品（包含用户手册）将在项目开始后 10 周交付。

1.1.4 时间安排和预算摘要

持续时间、人员需求及每个工作流的预算如下：
需求流（1 周，2 个团队成员，3 740 美元）
分析流（2 周，2 个团队成员，7 480 美元）
设计流（2 周，2 个团队成员，7 480 美元）
实现流（3 周，3 个团队成员，16 830 美元）
测试流（2 周，3 个团队成员，11 220 美元）
总的开发时间是 10 周，总的内部成本是 46 750 美元。

1.2 项目管理计划的演化

项目管理安排中的任何修改都必须在实施之前取得 Almaviva 的同意。所有的修改都必须文档化，以保证项目管理计划的正确与同步。

2 参考资料

所有的制品必须和公司的编程、文档和测试标准相一致。

3 定义和缩写

MSG-Martha Stockton Greengage；MSG 基金会是产品的客户。

4 项目组织

4.1 外部接口

项目上的所有工作将由 Almaviva、Bartolo 和 Cherubini 完成。Almaviva 将每周会见客户、报告进展和讨论可能的修改。

4.2 内部结构

开发团队包括 Almaviva、Bartolo 和 Cherubini。

4.3 角色和责任

Bartolo 和 Cherubini 将完成设计工作流。Almaviva 将实现类的定义和报告制品，Bartolo 将构造处理投资和操作成本的制品，Cherubini 将开发处理抵押的制品。每一个成员都将为他自己做的制品的质量负责。Almaviva 将检查集成效果和软件产品的整体质量，并和客户保持联络。

5 管理性的进程计划

5.1 启动计划

5.1.1 预估计划

如前文所说，总的开发时间估计为 10 周，总的内部开销为 46 750 美元。这些数字由行内类似情况推算，即相似项目的比较获取。

5.1.2 人员计划

Almaviva 需要这 10 周都扑在项目上，前 5 周仅处理一些管理性的事务，后 5 周兼任经理和程序员。Bartolo 和 Cherubini 也需要 10 周都在项目上，前 5 周作为系统分析师和设计者，后 5 周作为开发者和测试者。

5.1.3 资源获取计划

项目所需要的所有硬件、软件和 CASE 工具已经可用了。产品将交付到 MSG 基金会，并安装到一台通常从提供商处租借的台式电脑上。

5.1.4 项目人员培训计划

这个项目不要额外的人员培训。

5.2 工作计划

5.2.1～2 工作活动及时间分配

第 1 周 （已完成）会见客户，决定需求制品。检查需求制品。

第 2、3 周 （已完成）产生分析制品，并检查分析制品。将分析制品提交给客户，征得客户的同意。产生软件项目管理计划，并检查软件项目管理计划。

第 4、5 周 产生设计制品，检查设计制品。

第 6～10 周 实现并检查每一个类，单元测试且文档化，类的集成及集成测试，产品测试，文档的检查。

5.2.3 资源分配

3 个团队成员将为他们各自负责的制品独立工作。Almaviva 所负责的是监督另外 2 个成员的每日进度，检查实现结果，且为整体质量负责，并和客户交流。每日工作之后团队成员将在一起讨论问题和进度。和客户的例会是每周末报告进度及决定是否有修改。Almaviva 将负责确保时间和预算满足需要。风险管理同样是 Almaviva 的职责。

错误最小化和用户友好性最大化在 Almaviva 考虑问题时有着最高的优先级。Almaviva 也对所有的文档化工作和保证文档及时更新负责。

5.2.4 预算分配

每一个工作流的预算如下：

需求流	3 740（美元）
分析流	7 480
设计流	7 480
实现流	16 830
测试流	11 220
总共	46 750

5.3 控制计划

任何将影响到时间节点或预算的大的修改都必须先被 Almaviva 批准且文档化。没有外部的质量确保人员参与。每个人测试另一个人的产品，这样比让开发者自己做测试更有利。

Almaviva 需要保证项目在预算之内按期完成。这通过团队成员的每日例会来实现。Bartolo 和 Cherubini 将报告每日的进度和问题。Almaviva 判定他们是否完成了预想的进度，是否遵从了规格说明文档和项目管理计划。任何团队成员所面临的大的问题都要立即报告给 Almaviva。

5.4 风险管理计划

风险因素和跟踪机制如下：

开发新产品之前没有已在使用的同类的产品。因此这个产品不可能和一个老版本同时工作。所以这个产品需要广泛的测试。

假定客户对计算机毫不了解的。因此，在分析流时和与客户交流时应该特别用心。产品必须尽可能的做到用户友好。

因为大的设计错误有可能存在，广泛的测试工作从设计流就开始执行。相应地，每个团队成员将首先测试他自己的代码，然后再测试其他人的。Almaviva 将负责集成测试和产品测试。

信息必须符合指定的存储要求和响应时间。这不应该是一个大的问题，因为产品的规模较小。在整个开发过程中，对性能的监察同样由 Almaviva 负责。

硬件出问题可能性很小，如果发生，就租借另一台机器。如果编译器有错误，就换一个编译器。这些都能在硬件和编译器厂商提供的保证中得到确认。

5.5 项目收尾计划

不可用。

6 技术性的过程计划

6.1 过程模型

使用统一过程模型。

6.2 方法、工具和技术

按照统一过程模型完成工作流。产品将使用 Java 语言实现。

6.3 基础设施计划

产品将使用一台安装 Linux 的个人电脑上的 ArgoUML 开发。

6.4 产品验收安排

产品的验收将按照统一过程的步骤，由客户执行。

7 支持性过程计划

7.1 配置管理计划

所有制品将使用 CVS 来管理。

7.2 测试计划

按照统一过程的测试流完成。

7.3 文档计划

文档将按照统一过程模型所描述来生成。

7.4～5 质量保证和审查与审计计划

Bartolo 和 Cherubini 将测试彼此的代码，Almaviva 将进行集成测试。广泛的产品测试将由三个人共同完成。

7.6 问题解决计划

如 5.3 中所述，团队成员所遇到的所有问题将被立即报告给 Almaviva。

7.7 次承包商管理计划

不可用。

7.8 过程完善计划

所有活动的管理将和公司计划相一致，2 年内从 CMM 第 2 级提升到 CMM 第 3 级。

8 其他的计划

额外的部分：

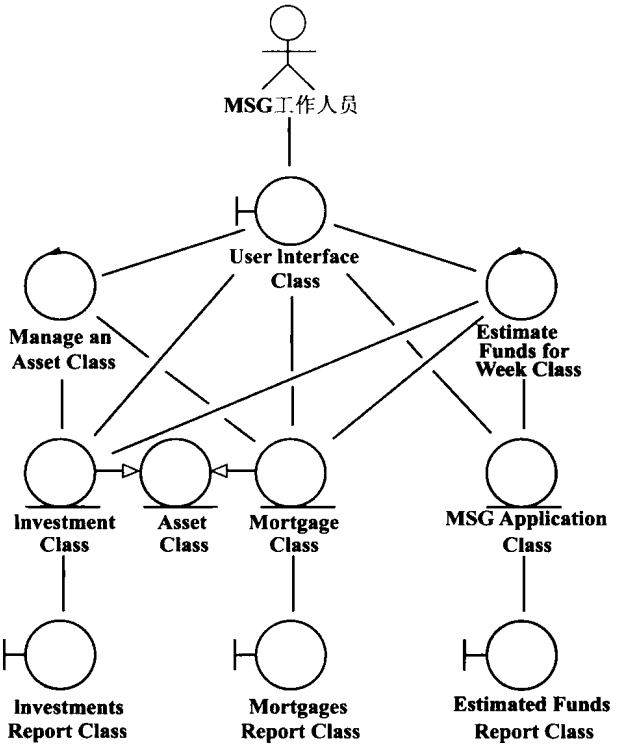
安全。使用产品需要密码。

培训。培训将在交付的时候由 Almaviva 完成。因为产品简单易用，所以一天时间用于培训足够了。Almaviva 将为第一年的产品使用提供免费的咨询。

维护。团队免费提供 12 个月的正确性维护。若需增加维护时间则需订立一个独立的合同。

附录 F 设计工作流：MSG 基金会案例研究

本附录包含了 MSG 基金会案例研究的类图的最终版本。在给出整体类图之后，依字母顺序列出 10 个组件类的 UML 图（如图 F-1 所示）。这些 UML 图包含属性和方法。正如 15.2 节所述，UML 图中可见性的前缀“-”表示 **private**、“+”表示 **public**、“#”表示 **protected**。这些属性和方法在 Java 的程序描述语言 PDL 里给出。同样，此处也没有 **Date Class**（参考 12.3 节）。



《实体类》 **Asset Class**
#assetNumber: string
+getAssetNumber():string +setAssetNumber(a:string):void +**abstract** read(fileName:RandomAccessFile):void +**abstract** obtainNewData():void +**abstract** performDeletion():void +**abstract** write(fileName:RandomAccessFile):void +**abstract** save():void +**abstract** print():void +**abstract** find(s:string):Boolean +delete():void +add():void

《控制类》 **Estimate Funds for Week Class**
+ ≪static≫ compute():void

图 F-1 MSG 基金会案例研究的最终类图

《边界类》 **Estimate Funds Report Class**
+ ≪static≫ printReport():void

《实体类》 **Investment Class**
- investmentName:string - expectedAnnualReturn:float - expectedAnnualReturnUpdated:string
+ getInvestmentName():string + setInvestmentName(n:string):void + getExpectedAnnualReturn():float + setExpectedAnnualReturn(r:float):void + getExpectedAnnualReturnUpdated():string + setExpectedAnnualReturnUpdated(d:string):void + totalWeeklyReturnOnInvestment():float + find(findInvestmentID:string):Boolean + read(fileName:RandomAccessFile):void + write(fileName:RandomAccessFile):void + save():void + print():void + printAll():void + obtainNewData():void + performDeletion():void + readInvestmentData():void + updateInvestmentName():void + updateExpectedReturn():void

《边界类》 **Investments Report Class**
+ ≪static≫ printReport():void

《控制类》 **Manage an Asset Class**
+ ≪static≫ manageInvestment():void + ≪static≫ manageMortgage():void

图 F-1 （续）

《实体类》 **Mortgage Class**
- mortgageeName:string - price:float - dateMortgageIssued:string - currentWeeklyIncome:float - weeklyIncomeUpdated:string - annualPropertyTax:float - annualInsurancePremium:float - mortgageBalance:float + ≪static final≫ INTEREST_RATE:float + ≪static final≫ MAX_PER_OF_INCOME:float + ≪static final≫ NUMBER_OF_MORTGAGE_PAYMENTS:int + ≪static final≫ WEEKS_IN_YEAR:float + getMortgageeName():string + setMortgageeName(n:string):void + getPrice():float + setPrice(p:float):void + getDateMortgageIssued():string + setDateMortgageIssued(w:string):void + getCurrentWeeklyIncome():float + setCurrentWeeklyIncome(i:float):void + getWeeklyIncomeUpdated():string + setWeeklyIncomeUpdated(w:string):void + getAnnualPropertyTax():float + setAnnualPropertyTax(t:float):void + getAnnualInsurancePremium():float + setAnnualInsurancePremium(p:float):void + getMortgageBalance():float + setMortgageBalance(m:float):void + totalWeeklyNetPayments():float + find(findMortgageID:string):Boolean + read(fileName:RandomAccessFile):void + write(fileName:RandomAccessFile):void + obtainNewData():void + performDeletion():void + print():void + ≪static≫ printAll():void + save():void + readMortgageData():void + updateBalance():void + updateDate():void

图 F-1 （续）

+ updateInsurancePremium():void + updateMortgageeName():void + updatePrice():void + updatePropertyTax():void + updateWeeklyIncome():void

《边界类》 **Mortgages Report Class**
+ ≪static≫ printReport():void

《实体类》 **MSG Application Class**
- ≪static≫ estimatedAnnualOperatingExpenses:float - ≪static≫ estimatedFundsForWeek:float
- ≪static≫ getAnnualOperatingExpenses():float - ≪static≫ setAnnualOperatingExpenses(e:float):void + ≪static≫ getEstimatedFundForWeek():float + ≪static≫ setEstimatedFundForWeek(e:float):void + ≪static≫ initializeApplication():void + ≪static≫ updateAnnualOperatingExpenses():void + ≪static≫ main()

《边界类》 **User Interface Class**
+ ≪static≫ clearScreen():void + ≪static≫ pressEnter():void + ≪static≫ displayMainMenu():void + ≪static≫ displayInvestmentMenu():void + ≪static≫ displayMortgageMenu():void + ≪static≫ displayReportMenu():void + ≪static≫ getChar():char + ≪static≫ getString():string + ≪static≫ getInt():int

图 F-1 （续）

附录 G 实现工作流：MSG 基金会案例研究（C++版）

MSG 基金会软件产品的完整C++ 源代码可以从 www. mhhe. com/schach 获取。

附录 H 实现工作流：MSG 基金会案例研究（Java 版）

MSG 基金会软件产品的完整 Java 源代码可以从 www. mhhe. com/schach 获取。

附录 I 测试工作流：MSG 基金会案例研究

MSG 基金会案例研究中的测试流在以下 4 节描述：

10. 11 节（需求）

11. 21 节（分析）

12. 6 节（设计）

13. 22 节（实现）