

教育部高职高专规划教材

高职高专现代信息技术系列教材

软 件 工 程

张海藩 编著

人 民 邮 电 出 版 社

图书在版编目 (CIP) 数据

软件工程/张海藩编著. —北京：人民邮电出版社，2003.7

ISBN 7-115-11258-4

. 软... . 张... . 软件工程-高等学校：技术学校-教材 . TP311.5

中国版本图书馆 CIP 数据核字 (2003) 第 034405 号

内 容 提 要

本书总结了编者多年来从事软件工程教学与研究的经验，并吸取了国内外众多同类教科书的精华。

本书共 7 章。第 1 章概述软件工程与软件过程；第 2 章讲述结构化分析的任务、过程、方法和工具；第 3 章讲述结构化设计的任务、准则、方法和工具；第 4 章着重介绍几种常用的测试技术；第 5 章讲述面向对象的概念、模型、分析、设计与实现；第 6 章讲述软件维护；第 7 章讲述软件项目的计划、组织和质量保证，并简要地介绍了能力成熟度模型。

本书的特点是：讲解深入浅出，通俗易懂，便于自学；把丰富的实例与原理性论述紧密配合，着重讲透基本的概念、原理、技术和方法；特别注重实用性，用几个综合性实例概括了本书的主要内容。认真阅读这些实例，不仅对读者深入理解软件工程很有帮助，而且有助于读者学会把软件工程的理论与技术运用到实际工作中去，这些实例还可作为上机实习的材料。

本书可作为大学专科或高等职业技术学院软件工程课程教材，也可作为大学本科相应课程的教学参考书。

教育部高职高专规划教材 高职高专现代信息技术系列教材 软 件 工 程

编 著 张海藩
责任编辑 潘春燕

人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号

邮编 100061 电子函件 315@pptph.com.cn

网址 <http://www.pptph.com.cn>

读者热线 010-67180876

北京汉魂图文设计有限公司制作

北京印刷厂印刷

新华书店总店北京发行所经销

开本：787 × 1092 1/16

印张：

字数：千字 2003 年 月第 1 版

印数：1- 000 册 2003 年 月北京第 1 次印刷

ISBN 7-115-11258-4/TP · 3443

定价：.00 元

本书如有印装质量问题，请与本社联系 电话：(010) 67129223

高职高专现代信息技术系列教材

编 委 会 名 单

主 编 高 林

执行主编 张强华

委 员 (以姓氏笔画为序)

吕新平 林全新 郭力平

丛书前言

江泽民总书记在十五大报告中提出了培养数以亿计高素质的劳动者和数以千万计专门人才的要求，指明了高等教育的发展方向。只有培养出大量高素质的劳动者，才能把我国的人数优势转化为人力优势，提高全民族的竞争力。因此，我国近年来十分重视高等职业教育，把高等职业教育作为高等教育的重要组成部分，并以法律形式加以约束与保证。高等职业教育由此进入了蓬勃发展时期，驶入了高速发展的快车道。

高等职业教育有其自身的特点。正如教育部“面向 21 世纪教育振兴行动计划”所指出的那样，“高等职业教育必须面向地区经济建设和社会发展，适应就业市场的实际需要，培养生产、管理、服务第一线需要的实用人才，真正办出特色。”因此，不能以本科压缩和变形的形式组织高等职业教育，必须按照高等职业教育的自身规律组织教学体系。为此，我们根据高等职业教育的特点及社会对教材的普遍需求，组织高等职业学校有丰富教学经验的老师，编写了这套《高职高专现代信息技术系列教材》。本套书已纳入教育部高职高专规划教材。

本套教材充分考虑了高等职业教育的培养目标、教学现状和发展方向，在编写中突出了实用性。本套教材重点讲述目前在信息技术行业实践中不可缺少的、广泛使用的、从业人员必须掌握的实用技术。即便是必要的理论基础，也从实用的角度、结合具体实践加以讲述。大量具体操作步骤、许多实践应用技巧、接近实际的实训材料保证了本套教材的实用性。

在本套教材编写大纲的制定过程中，广泛收集了高等职业学院的教学计划，调研了多个省市高等职业教育的实际，反复讨论和修改，使得编写大纲能最大限度地符合我国高等职业教育的要求，切合高等职业教育实际。

在选择作者时，我们特意挑选了在高等职业教育一线的优秀骨干教师。他们熟悉高等职业教育的教学实际，并有多年教学经验；其中许多是“双师型”教师，既是教授、副教授，同时又是高级工程师、认证高级设计师；他们既有坚实的理论知识，很强的实践能力，又有较多的写作经验及较好的文字水平。

目前我国许多行业开始实行劳动准入制度和职业资格制度，为此，本套教材也兼顾了一些证书考试(如计算机等级考试)，并提供了一些具有较强针对性的训练题目。

对于本套教材我们将提供教学支持(如提供电子教案等)，同时注意收集本套教材的使用情况，不断修改和完善。

本套教材是高等职业学院、高等技术学院、高等专科学院教材。适用于信息技术的相关专业，如计算机应用、计算机网络、信息管理、电子商务、计算机科学技术、会计电算化等。也可供优秀职高学校选作教材。对于那些要提高自己应用技能或参加一些证书考试的读者，本套教材也不失为一套较好的参考书。

最后，恳请广大读者将本套教材的使用情况及各种意见、建议及时反馈给我们，以便我们在今后的工作中，不断改进和完善。

编者的话

软件工程是指导计算机软件开发与维护的一门工程学科。经过 30 多年的研究与发展，软件工程正逐步走向成熟，应用日益广泛。目前，业内人士已经普遍认识到，如果一个软件项目不用软件工程来指导，则必然会受到实践的惩罚。

计算机专业的学生毕业之后，无论从事软件开发、维护还是销售，都离不开软件工程的知识。可以说，软件工程概论课是他们参加工作后马上就要直接应用的一门专业课。但是，在校学生由于缺乏从事软件项目的实践经验，往往认识不到软件工程的重要性，更难于掌握软件工程的精髓。

为帮助读者更好地学习、掌握软件工程，本书首先从具体的例子谈起，强调指出开发软件不等于编写程序，学习和运用软件工程非常必要。然后用丰富的实例与原理性论述紧密配合，讲透软件工程的基本概念、原理、技术和方法，在讲完每个重点专题之后，都用一个综合性的实例概括全章内容。认真阅读这些综合性实例，不仅对读者深入理解软件工程很有帮助，而且有助于读者学会把软件工程的理论与技术运用到实际工作中去。

本书共 7 章。第 1 章概述，讲述软件生命周期各阶段的基本任务，全面概括地介绍软件工程这门学科以及典型的软件过程模型。第 2 章结构化分析，讲述结构化分析的任务、过程、方法和工具，最后讲述了一个实际软件(工资支付系统)的结构化分析过程。第 3 章结构化设计，讲述软件设计的任务、准则和启发规则，介绍典型的设计方法和工具，最后讲述了一个实际软件(汉字行编辑程序)的结构化设计过程。第 4 章结构化实现，讲述编码和测试，重点介绍了几种常用的测试技术。第 5 章面向对象方法学导论，首先通过一个面向对象程序设计实例引入面向对象的基本概念和方法，然后系统地讲述面向对象的概念、方法和模型，并用一个综合性的实例(C++类库管理系统的面向对象分析与设计)概括了本章的主要内容，最后简要地介绍面向对象实现及面向对象方法学的主要优点。第 6 章软件维护，讲述软件维护的定义、特点和过程，讨论影响软件可维护性的主要因素，并简要地介绍预防性维护。第 7 章软件项目管理，讲述软件项目的计划、组织和质量保证，并简要地介绍国际上流行的能力成熟度模型。

在本书的编写过程中，牟永敏博士曾对本书编写大纲提出过一些有益的建议，张劲松和张展新用 VC++实现了书中讲述的 C++类库管理系统，张雯和张杰为本书的出版做了许多具体工作，谨在此向他们表示感谢。

编者
2003 年 3 月

目 录

第 1 章 概述.....	1
1.1 开发软件不等于编写程序	1
1.1.1 开发软件应该完成的工作远远多于编写程序应该完成的工作	1
1.1.2 错误做法导致软件危机	2
1.1.3 消除软件危机的途径	5
1.2 软件工程	5
1.2.1 软件工程的定义	5
1.2.2 软件工程的基本原理	6
1.2.3 软件工程方法学	8
1.3 软件生命周期	10
1.4 软件过程	12
1.4.1 瀑布模型	13
1.4.2 快速原型模型	15
1.4.3 增量模型	16
1.4.4 螺旋模型	17
1.5 小结	18
习题一	20
第 2 章 结构化分析	22
2.1 可行性研究的任务.....	22
2.2 可行性研究过程.....	23
2.3 需求分析的任务.....	25
2.4 需求分析的过程.....	27
2.5 与用户沟通的方法.....	29
2.5.1 访谈	30
2.5.2 简易的应用规格说明技术	30
2.6 分析建模与规格说明.....	32
2.6.1 分析建模	32
2.6.2 软件需求规格说明书	32
2.7 验证软件需求.....	35
2.7.1 至少从四个方面验证软件需求	35
2.7.2 验证软件需求的方法	35
2.7.3 用于需求分析的软件工具	36
2.8 系统流程图.....	37

2.8.1 系统流程图的符号.....	37
2.8.2 举例.....	37
2.8.3 分层画系统流程图.....	39
2.8.4 系统流程图的用途.....	39
2.9 实体-联系图.....	40
2.10 数据流图	41
2.10.1 数据流图的符号	42
2.10.2 举例	43
2.10.3 命名	45
2.10.4 数据流图的用途	46
2.11 数据字典.....	47
2.11.1 数据字典的内容	48
2.11.2 定义数据的方法	48
2.11.3 数据字典的用途	49
2.11.4 实现数据字典的途径	49
2.12 其他图形工具.....	50
2.12.1 层次方框图	51
2.12.2 Warnier 图	51
2.12.3 IPO 图	52
2.13 成本/效益分析	53
2.13.1 成本估计	53
2.13.2 成本/效益分析方法	55
2.14 结构化分析实例	56
2.14.1 工资支付问题定义	56
2.14.2 可行性研究	58
2.14.3 需求分析	66
2.15 小结	74
习题二	76
第 3 章 结构化设计	78
3.1 软件设计的任务	78
3.1.1 概要设计的任务	78
3.1.2 详细设计的任务	79
3.2 从分析过渡到设计	79
3.3 软件设计准则	80
3.3.1 模块化与模块独立	80
3.3.2 抽象	82
3.3.3 逐步求精	83
3.3.4 信息隐藏	83

目 录

3.4 度量模块独立性的标准.....	84
3.4.1 耦合.....	84
3.4.2 内聚.....	85
3.5 启发规则.....	86
3.5.1 改进软件结构提高模块独立性.....	86
3.5.2 模块规模应该适中.....	86
3.5.3 深度、宽度、扇出和扇入都应适当.....	87
3.5.4 模块的作用域应该在控制域之内.....	87
3.5.5 力争降低模块接口的复杂程度.....	88
3.5.6 设计单入口单出口的模块.....	88
3.5.7 模块功能应该可以预测.....	88
3.6 描绘软件结构的图形工具.....	88
3.6.1 层次图和 HIPO 图.....	88
3.6.2 结构图.....	89
3.7 面向数据流的设计方法.....	91
3.7.1 概念.....	91
3.7.2 变换分析.....	92
3.7.3 事务分析.....	98
3.7.4 设计优化.....	99
3.8 人机界面设计.....	99
3.8.1 应该考虑的设计问题.....	100
3.8.2 人机界面设计过程.....	101
3.8.3 界面设计指南.....	102
3.9 过程设计.....	104
3.10 过程设计的工具.....	106
3.10.1 程序流程图.....	106
3.10.2 盒图.....	107
3.10.3 PAD 图.....	107
3.10.4 判定表.....	110
3.10.5 判定树.....	111
3.10.6 过程设计语言(PDL)	111
3.11 面向数据结构的设计方法.....	112
3.11.1 Jackson 图.....	113
3.11.2 改进的 Jackson 图.....	113
3.11.3 Jackson 方法.....	114
3.12 结构化设计实例.....	118
3.12.1 汉字行编辑程序的规格说明.....	119
3.12.2 概要设计.....	121
3.12.3 概要设计结果.....	124

3.12.4 详细设计.....	127
3.12.5 详细设计结果.....	134
3.13 小结.....	165
习题三.....	166
第4章 结构化实现.....	168
4.1 编码.....	168
4.1.1 选择适当的程序设计语言.....	168
4.1.2 正确的编码风格.....	169
4.2 软件测试概述.....	172
4.2.1 软件必须测试.....	172
4.2.2 软件测试的目标.....	172
4.2.3 两类测试方法.....	173
4.2.4 软件测试准则.....	174
4.3 白盒测试技术.....	175
4.3.1 逻辑覆盖.....	175
4.3.2 控制结构测试.....	178
4.4 黑盒测试技术.....	186
4.4.1 等价划分.....	186
4.4.2 边界值分析.....	189
4.4.3 错误推测.....	190
4.5 测试策略.....	191
4.5.1 测试步骤.....	191
4.5.2 单元测试.....	191
4.5.3 集成测试.....	195
4.5.4 确认测试.....	199
4.6 调试.....	200
4.6.1 调试过程.....	200
4.6.2 调试途径.....	201
4.7 软件可靠性.....	202
4.7.1 基本概念.....	203
4.7.2 估算平均无故障时间的方法.....	203
4.8 小结.....	205
习题四.....	206
第5章 面向对象方法学导论.....	210
5.1 一个面向对象的程序实例.....	210
5.1.1 用对象分解取代功能分解.....	210
5.1.2 设计类等级.....	212

目 录

5.1.3 定义属性和服务.....	214
5.1.4 用 C++语言实现.....	215
5.2 面向对象的概念.....	223
5.2.1 对象.....	223
5.2.2 其他面向对象的概念.....	225
5.3 面向对象方法学概述.....	229
5.3.1 面向对象方法学的要点.....	229
5.3.2 面向对象建模.....	231
5.3.3 面向对象的软件过程.....	232
5.4 对象模型.....	233
5.4.1 表示类的图形符号.....	234
5.4.2 表示关系的图形符号.....	235
5.5 动态模型.....	241
5.5.1 概念.....	241
5.5.2 图示符号.....	242
5.6 面向对象分析.....	244
5.6.1 确定问题域内的对象.....	245
5.6.2 确定关联.....	246
5.6.3 确定属性.....	247
5.6.4 建立继承关系.....	248
5.6.5 建立动态模型.....	248
5.6.6 建立功能模型.....	249
5.6.7 定义服务.....	249
5.7 面向对象设计.....	249
5.7.1 面向对象设计准则.....	250
5.7.2 启发规则.....	251
5.8 面向对象分析与设计实例.....	253
5.8.1 面向对象分析.....	253
5.8.2 面向对象设计.....	254
5.9 面向对象实现.....	260
5.9.1 面向对象的程序设计语言.....	260
5.9.2 面向对象程序设计风格.....	261
5.9.3 面向对象测试.....	263
5.10 面向对象方法学的主要优点.....	264
5.11 小结.....	267
习题五.....	269
第 6 章 软件维护.....	270
6.1 软件维护的定义与策略.....	270

6.1.1 定义.....	270
6.1.2 策略.....	271
6.2 软件维护的特点.....	272
6.2.1 结构化维护与非结构化维护差别悬殊.....	272
6.2.2 维护的代价高昂.....	273
6.2.3 维护的问题很多.....	273
6.3 软件维护过程.....	274
6.3.1 维护组织.....	274
6.3.2 维护报告.....	274
6.3.3 维护的事件流.....	275
6.3.4 保存维护记录.....	276
6.3.5 评价维护活动.....	276
6.4 软件的可维护性.....	277
6.4.1 决定软件可维护性的因素.....	277
6.4.2 文档.....	278
6.4.3 可维护性复审.....	279
6.5 预防性维护.....	279
6.5.1 必要性.....	279
6.5.2 可行性.....	280
6.6 软件再工程过程.....	281
6.7 小结.....	284
习题六.....	284
 第 7 章 软件项目管理.....	286
7.1 度量软件规模.....	286
7.1.1 代码行技术.....	286
7.1.2 功能点技术.....	287
7.2 估算软件开发工作量.....	289
7.2.1 静态单变量模型.....	289
7.2.2 动态多变量模型.....	289
7.2.3 COCOMO2 模型.....	290
7.3 进度计划.....	293
7.3.1 估算开发时间.....	293
7.3.2 甘特(Gantt)图.....	295
7.3.3 工程网络.....	296
7.3.4 估算进度.....	298
7.3.5 关键路径.....	299
7.3.6 机动时间.....	299
7.4 人员组织.....	301

目 录

7.4.1 民主制程序员组.....	301
7.4.2 主程序员组.....	302
7.4.3 现代程序员组.....	303
7.5 质量保证.....	305
7.5.1 软件质量的定义.....	305
7.5.2 软件质量保证措施.....	306
7.6 软件配置管理.....	309
7.7 能力成熟度模型.....	310
7.8 小结.....	312
习题七.....	313
参考文献.....	314

第1章 概述

开发软件是不是就是编写程序？怎样才能以较低的成本开发出高质量的软件？为了开发出一个符合用户需要的软件，应该完成哪些工作？本章将详细地讨论并概括地回答上述这些问题。

1.1 开发软件不等于编写程序

1.1.1 开发软件应该完成的工作远远多于编写程序应该完成的工作

开发软件是否就是编写程序呢？为了回答这个大家关心的问题，让我们来考察两个具体例子。

假设讲授 C 语言程序设计课的老师给学生布置了一道作业：“编写一个程序，读入圆半径、三角形的底边长和高、矩形的两条边长，计算圆形、三角形和矩形的面积。”怎样编写这个程序呢？因为在中学数学课上已经学过计算上述几种图形面积的公式，这道作业看起来比较容易完成。一些同学可能先在主函数 main 中说明几个浮点型的变量，分别用于保存圆半径、三角形底边长和高以及矩形两条边长的值，然后用输入语句读入原始数据，最后用已知的公式计算并用输出语句输出圆形、三角形和矩形的面积值。而一些对结构程序设计技术掌握得比较好的同学，可能先定义三个函数，它们的功能分别是计算圆面积、计算三角形面积和计算矩形面积，然后在主函数 main 中调用这三个函数。一些细心的同学还可能想到，在计算面积之前应该先对输入的原始数据进行校核，如果是非法数据（例如，负数），则提醒用户重新输入。

从上面这个例子可以看出，由于对程序的需求很明确，编写程序所应完成的工作主要是设计算法（即完成指定功能的步骤），然后用程序设计语言（例如，C 语言）表达该算法。

下面让我们再通过一个具体例子考察开发软件所应完成的工作。

小王是计算机应用专业的一名毕业生，毕业后在一所职业高中工作，分配给他的工作是为该校开发一个工资支付软件，以便每月运行该软件生成当月的工资明细表和相应的财务报表。怎样完成这项软件开发任务呢？

在校期间，小王的 C 语言程序设计课是全班学得最好的，为了尽快完成任务，他想立即动手用 C 语言编写工资支付程序。但是，一旦开始编写程序，他才发现问题并不像他想象的那样简单。怎样计算每个人的工资呢？教师的工资怎样计算？干部的工资怎样计算？工人的工资怎样计算？工资明细表中还应该包含哪些内容？会计科需要哪些财务报表？每张报表中包含哪些条目？……上述种种问题小王都不能确切地回答。然而不能准确地回答这些问题就根本无法编写程序。为了搞清楚这些问题，必须向工资支付系统的用户（即会计）虚心请教，

也就是说，必须不厌其烦地做深入的调查研究工作，以便准确具体地了解用户对软件的需求。通常把通过调查研究向用户了解需求的工作过程称为需求分析。

是否知道了用户对软件的需求之后就可以立即动手编写程序了呢？回答仍然是否定的。为了开发一个软件，所需编写的程序规模通常都很大，必须先经过精心的设计过程，依据设计好的方案才能编写出合格的程序。这就好像为了盖一栋大楼必须先经过设计过程，画出蓝图，然后施工人员严格按照蓝图施工才能盖出高质量的大楼。盖大楼与搭一个简易的小棚子完全不同，搭小棚子可能无须设计，一边搭一边想也可以，盖大楼则必须先进行精心的设计，否则必垮无疑。同样，编写一个小程序可以边干边想，而编写一个大型程序则必须先进行设计，边干边想必然陷入混乱而不可收拾。

事实上，为了盖一栋大楼，工程师必须首先知道大楼的用途，并且事先进行地质勘探，以探明该地的地质条件是否适合盖这么高的大楼，与此同时有关部门则从环保、消防等方方面面审查是否可以在该地区盖这样的大楼，这些都属于可行性研究的范畴。如果在该地区可以盖这样的大楼，工程师则进一步深入地了解用户对该大楼的具体需求，以便以这些需求为依据进行设计。类似地，为了开发一个软件，软件工程师必须首先弄清楚要用这个软件解决用户的什么问题（这个过程称为问题定义），以便明确该软件的“主攻”方向。接下来往往需要进行可行性研究，以便弄清楚用户的问题是否有行得通的解决办法，如果没有可行的解决方案就鲁莽地着手开发软件，必然造成大量资源的浪费。通过了可行性研究之后，还必须进行需求分析和设计，然后才能编写程序。

众所周知，大楼盖好之后需要经过严格的检验过程，仅当大楼确实符合用户需求而且质量合格时，才能交付用户使用。软件开发与此类似，程序编写出来之后，也必须经过严格的检验过程（称为测试），软件符合用户需求而且质量合格，才能交付给用户使用。

从上述论述可知，开发软件并非就是编写程序，事实上编写程序仅仅是开发软件所应完成的工作的一部分，而且只占一小部分。为了开发出一个符合用户需要、质量合格的软件，软件工程师必须首先弄清楚用户面临的问题是什么，也就是要明确软件的“主攻”方向；接下来应该进行可行性研究，分析用户面临的问题是否有行得通的解决方案，为避免浪费资源，仅在该软件的开发是可行的前提下，才进行实质性的开发工作；然后应该进行需求分析工作，通过与用户的反复交流，搞清楚用户对该软件的具体需求，这些需求是进行软件设计的依据；在编写程序之前需要先进行设计，通常，大型软件的设计工作又分成两个阶段进行，先进行总体设计（又称为概要设计），再进行详细设计；编写程序实质上是把设计结果翻译成用某种程序设计语言书写的程序；程序编写出来之后，还需要经过严格的测试过程，软件确实符合用户需求而且质量合格，才能交付给用户使用。

此外，规模较大的软件都是由多人分工协作开发出来的，为了使开发人员相互之间能准确地交换信息，同时也为了在软件交付给用户使用期间能比较容易地修改或扩充，必须把软件开发全过程中各个阶段的工作成果用文字、图表等形式准确地记录下来（通常把这些记录称为文档）。事实上，软件是由程序、数据和相关的文档组成的。从软件的构成可以看出，程序仅仅是组成软件的一个成分，从这个角度说，开发软件也绝不等于编写程序。

1.1.2 错误做法导致软件危机

刚才我们通过两个具体例子说明了开发软件不等于编写程序，但是，迄今为止，仍然有

不少人错误地认为开发软件就是编写程序，或者认为开发软件主要就是编写程序。人们之所以有错误的认识并在开发软件时采用了错误的做法，主要可归因于在计算机系统发展的早期阶段“开发软件”的个体化特点。

在计算机系统发展的早期（20世纪60年代中期以前），通用硬件已经相当普遍，软件却是为每个具体应用而专门编写的，大多数人认为开发软件是不需要预先计划的事情。这时的“软件”实际上就是规模较小的程序，程序的编写者和使用者往往是同一个（或同一组）人。由于规模小，程序编写起来相当容易，当时既没有什么系统化的软件开发方法，也没有对软件开发工作进行任何管理。这种个体化的软件环境，使得软件设计往往只是在人们头脑中隐含进行的一个模糊过程，除了程序清单之外，根本没有其他文档资料保存下来。

从60年代中期到70年代中期是计算机系统发展的第二代时期，这个时期的一个重要特征是出现了“软件作坊”，广泛使用产品软件。但是，“软件作坊”基本上仍然沿用早期形成的个体化软件开发方法。随着计算机应用的日益普及，软件数量急剧膨胀。在程序运行时发现的错误必须设法改正；用户有了新的需求时必须相应地修改程序；硬件或操作系统更新时，通常需要修改程序以适应新的环境。在软件已经交付给用户使用之后，为了改正错误或满足新的需求而修改软件的过程，称为软件维护。上述种种软件维护工作，以令人吃惊的比例耗费资源。更严重的是，许多软件的个体化特性使得它们最终成为不可维护的。“软件危机”就这样开始出现了！

所谓软件危机是指在计算机软件的开发和维护过程中所遇到的一系列严重问题。这些问题绝不仅仅是不能正常运行的软件才具有的，实际上，几乎所有软件都不同程度地存在这些问题。

概括地说，软件危机包含下述两方面的问题：怎样开发软件，以满足人们对软件日益增长的需求；如何维护数量不断增加的已有软件。

具体说来，软件危机主要有以下一些典型表现：

? 对软件开发成本和进度的估计常常很不准确。实际成本比估计成本高出几倍甚至十几倍，实际进度比预期进度拖延几个月甚至几年的现象并不罕见。

? 用户对“已完成的”软件系统不满意甚至拒绝接受的现象经常发生。

? 软件产品的质量往往靠不住。

? 软件常常是不可维护的。很多程序中的错误都非常难以改正，实际上不可能使这些程序适应新的运行环境，也不能根据用户的需要在原有程序中增加一些新的功能。

? 软件通常没有适当的文档资料。缺乏必要的文档资料或者文档资料不合格，必然给软件开发和维护带来许多严重的困难和问题。

? 软件成本在计算机系统总成本中所占的比例逐年上升，例如，美国在1985年软件成本大约已占计算机系统总成本的90%

? 软件开发生产率提高的速度，既跟不上硬件的发展速度，也远远跟不上计算机应用迅速普及的趋势。软件产品“供不应求”的现象使得人类无法充分利用现代计算机硬件提供的巨大潜力。

以上列举的仅仅是软件危机的一些明显的表现，与软件开发和维护有关的问题远远不止这些。

出现软件危机的主要原因是人们在开发软件时使用了错误的方法，而错误做法是在错误认识的指导下采用的。为了消除软件危机首先应该树立起对软件和软件开发的正确认识。因此下面着重批驳一些典型的似是而非的论调。

“有一个对目标的概括描述就足以着手编写程序了，许多细节可以在以后再补充。”

事实上，对用户要求没有完整准确的认识就匆忙着手编写程序是许多软件开发工程失败的主要原因之一。只有用户才真正了解他们自己的需要，但是许多用户在开始时并不能准确具体地叙述他们的需要，软件开发人员需要做大量深入细致的调查研究工作，反复多次地和用户交流信息，才能真正全面、准确、具体地了解用户的要求。对问题和目标的正确认识是解决任何问题的前提和出发点，软件开发同样也不例外。急于求成仓促上阵，对用户要求没有正确认识就匆忙着手编写程序，这就如同不打好地基就盖高楼一样，最终必然垮台。

“所谓软件开发就是编写程序并设法使它运行。”

一个软件从定义、开发、使用和维护，直到最终被废弃，经历了一个漫长的时期，这就如同一个人要经过胎儿、儿童、青年、中年、老年，直到最终死亡的漫长时期一样。通常把软件经历的这个漫长的时期称为软件生命周期（也称为生存周期）。正如我们在 1.1.1 小节中已经讲过的那样，软件开发最初的工作应是问题定义，也就是确定要求解决的问题是什么；然后要进行可行性研究，决定该问题是否存在一个可行的解决办法；接下来应该进行需求分析，也就是深入具体地了解用户的要求，在所要开发的系统（不妨称之为“目标系统”）必须做什么这个问题上和用户取得完全一致的看法。经过上述软件定义时期的准备工作才能进入开发时期，而在开发时期首先需要对软件进行设计（通常又分为总体设计和详细设计两个阶段），然后才能进入编写程序的阶段，程序编写完之后还必须经过大量的测试工作（需要的工作量通常占软件开发全部工作量的 40% ~ 50%）才能最终交付使用。所以，编写程序只是软件开发过程中的一一个阶段，而且在典型的软件开发工程中，编写程序所需的工作量只占软件开发全部工作量的 10% ~ 20%。

另一方面还必须认识到程序只是完整的软件产品的一个组成部分，在上述软件生命周期的每个阶段都要得出最终产品的一个或几个组成部分（这些组成部分通常以文档资料的形式存在）。也就是说，一个软件产品必须由一个完整的配置组成，软件配置成分主要有程序、文档和数据。必须清除只重视程序而忽视软件配置其余成分的糊涂观念。

“用户对软件的要求不断变化，然而软件是柔软而灵活的，可以轻易地改动。”

确实，用户对软件的要求经常改变，特别是一个大型软件开发项目持续的时间往往相当长，在这段时间内由于外界环境变化以及人的认识不断深化，都会或多或少地改变对软件的要求。但是，必须看到也有相当多的改动不是由于用户要求的变化所造成的，而是由于软件开发人员在开发初期没有完全理解用户的要求，直到设计阶段甚至验收阶段才发现“已完成的”软件不完全符合用户的需要，从而必须进行修改。

严重的问题是，在软件开发的不同阶段进行修改需要付出的代价是很不相同的，在早期引入变动涉及的面较少，因而代价也比较低；而在开发的中期软件配置的许多成分已经完成，引入一个变动要对所有已完成的配置成分都做相应的修改，不仅工作量大而且逻辑上也更复杂，因此付出的代价较大；在软件“已经完成”时再引入变动，当然需要付出更高的代价。根据美国一些软件公司的统计资料，在后期引入一个变动比在早期引入相同变动所需付出的代价高 2~3 个数量级。图 1.1 定性地描绘了在不同时期引入同一个变动需要付出的

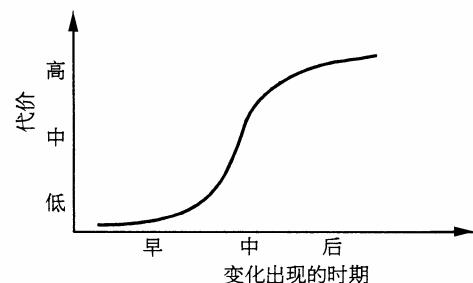


图 1.1 不同时期引入同一变动付出的代价

代价的变化趋势。

“软件投入生产性运行以后需要的维护工作并不多，而且维护是一种很容易做的简单工作。”

由于有这种看法，给维护分配的预算往往比较少，配备的人员也比较弱；然而，轻视维护是一个最大的错误。许多软件产品的使用寿命长达10年甚至20年，在这样漫长的时期中不仅必须改正使用过程中发现的每一个潜伏的错误，而且当环境变化（例如硬件或系统软件更新换代）时，还必须相应地修改软件以适应新的环境，特别是必须经常改进或扩充原来的软件以满足用户不断变化的需要。所有这些改动都属于维护工作，而且是在软件已经完成之后进行的，因此维护是极端艰巨复杂的工作，需要花费很大代价。统计数据表明，实际上用于软件维护的费用占软件总费用的55%~70%。软件工程学的主要目标就是提高软件的可维护性，减少软件维护的代价。

1.1.3 消除软件危机的途径

为了消除软件危机，首先应该对计算机软件有一个正确的认识。正如前面已经讲过的，应该彻底清除在计算机系统早期发展阶段形成的“软件就是程序”的错误观念。一个软件必须由一个完整的配置组成，事实上，软件是程序、数据及相关文档的完整集合。其中，程序是能够完成预定功能且具有预期性能的可执行的指令序列；数据是使程序能够适当地处理信息的数据结构；文档是开发、使用和维护程序所需要的图文资料。

更重要的是，必须充分认识到软件开发不是某种个体劳动的神秘技巧，而应该是一种组织良好、管理严密、各类人员协同配合共同完成的工程项目。应该充分吸取和借鉴人类长期以来从事各种工程项目所积累的行之有效的原理、概念、技术和方法，特别要吸取几十年来人类从事计算机硬件研究和开发的经验教训。

应该推广使用在实践中总结出来的开发软件的成功的技术和方法，并且研究探索更好更有效的技术和方法，尽快消除在计算机系统早期发展阶段形成的一些错误概念和做法。

应该开发和使用更好的软件工具。正如机械工具可以“放大”人类的体力一样，软件工具可以“放大”人类的智力。在软件开发的每个阶段都有许多繁琐重复的工作需要做，在适当的软件工具辅助下，开发人员可以把这类工作做得既快又好。如果把各个阶段使用的软件工具有机地结合成一个整体，支持软件开发的全过程，则称为软件工程支撑环境。

总之，为了消除软件危机，既要有技术措施（方法和工具），又要有必要的组织管理措施。软件工程正是从管理和技术两方面研究如何更好地开发和维护计算机软件的一门新兴学科。

1.2 软件工程

1.2.1 软件工程的定义

为了消除软件危机，软件工作者们认真总结人类长期以来从事软件开发与维护的经验教训，逐渐形成了一门新兴的工程学科——软件工程。那么，什么是软件工程呢？

概括地说，软件工程是指导计算机软件开发与维护的工程学科。它采用工程的概念、原

理、技术和方法来开发与维护软件，把经过时间考验而证明是正确的管理技术和目前能够得到的最有效的技术与方法结合起来，以经济地开发出高质量的软件并有效地维护它，这就是软件工程。

人们从不同的角度出发，曾经给软件工程下过多个定义，认真学习、领会这些定义，有助于建立起对软件工程这门工程学科的全面的整体性认识。下面介绍对软件工程的几个典型的定义。

1983 年美国电气和电子工程师协会（ IEEE ）给软件工程下的定义是：“软件工程是开发、运行、维护和修复软件的系统方法。”这个定义相当概括，它主要强调了软件工程是系统化的方法而不是某种神秘的个人技巧。

Fairly 认为：“软件工程学是为了在成本限额以内按时完成开发和修改软件产品所需要的系统生产和维护技术及管理学科。”这个定义明确地指出了软件工程的目标是，在成本限额以内按时完成开发和修改软件产品的工作，同时也指出了软件工程包含技术和管理两方面的内容。

Fritz Bauer 给出了下述定义：“软件工程是为了经济地获得可靠的而且能在实际机器上有效地运行的软件，而建立和使用的完善的工程化原则。”这个定义不仅指出了软件工程的目标是经济地开发出高质量的软件，而且强调了软件工程是一门工程学科，它应该建立并使用完善的工程化原则。

1993 年 IEEE 进一步给出了软件工程的一个更全面的定义：

软件工程是： 把系统化的、规范的、可度量的途径应用于软件开发、运行和维护的过程，也就是把工程化应用于软件工作中； 研究 中提到的途径。

1.2.2 软件工程的基本原理

自从 1968 年在原联邦德国召开的国际会议上正式提出并使用了“软件工程”这个术语以来，研究软件工程的专家学者们陆续提出了 100 多条关于软件工程的准则。著名的软件工程专家 B.W.Boehm 综合了这些学者们的意見并总结了 TRW 公司多年开发软件的经验，于 1983 年在一篇论文中提出了软件工程的 7 条基本原理。他认为这 7 条原理是确保软件产品质量和开发效率的最小集合。可以证明，在此之前已经提出的 100 多条软件工程原理或准则，都可以由这 7 条原理的任意组合蕴含或派生出来。学习、领会这 7 条原理对于学习下面将要讲述的软件工程的具体概念、原理、技术和方法是有一定帮助的，因此我们简要地介绍软件工程的这 7 条基本原理。

1. 用分阶段的生命周期计划严格管理

统计数字表明，在不成功的软件项目中有一半左右是由于计划不周造成的，可见 Boehm 把建立完善的计划作为第 1 条基本原理是吸取了前人的教训而提出来的。

在软件开发与维护的漫长生命周期中，需要完成许多性质各异的工作。这条基本原理意味着，应该把软件生命周期划分成若干个阶段，并相应地制定出切实可行的计划，然后严格地按照计划对软件的开发与维护工作进行管理。

2. 坚持进行阶段评审

当时已经认识到，软件的质量保证工作不能等到编码阶段结束之后再进行。这样说至少有下述的两个理由：第一，大部分错误是在编码之前造成的，例如，根据 Boehm 等人的统计，

设计错误占软件错误的 63%，而编码错误仅占 37%；第二，错误发现与改正得越晚，改正错误所需付出的代价也越高（参见图 1.1）。因此，在每个阶段都进行严格的评审，以便尽早发现在软件开发过程中所犯的错误，并及时加以改正，是一条必须遵循的重要原则。

3. 实行严格的产品控制

在开发软件的过程中不应该随意改变需求，因为改变一项需求往往需要付出较高的代价。但是，在软件开发过程中改变需求又是难免的。由于外部环境变化或主观认识的提高，相应地改变用户对软件的需求是一种客观需要，显然不能硬性禁止客户提出改变软件需求的要求，而只能依靠科学的产品控制技术来顺应这种要求。也就是说，当改变需求时，为了保持软件各个配置成分的一致性，必须实行严格的产品控制，其中主要是实行基准配置管理。所谓基准配置又称为基线配置，它们是经过阶段评审后的软件配置成分（各个阶段产生的文档或程序代码）。基准配置管理也称为变动控制：一切有关修改软件的建议，特别是涉及到对基准配置的修改建议，都必须按照严格的规程进行评审，获得批准以后才能实施修改，修改后还必须按照严格的规程进行软件质量保证活动。绝对不允许随意修改软件（包括尚在开发过程中的软件）。

4. 采用现代程序设计技术

在这条原理中使用的术语“程序设计技术”并不仅仅局限于编程技术，而是泛指软件开发技术。从提出软件工程的概念以来，人们一直把主要精力用于研究各种新的程序设计技术。20世纪60年代末提出的结构程序设计技术已经成为绝大多数人公认的先进的程序设计技术。以后又进一步发展出各种结构化分析（SA）与结构化设计（SD）技术。近年来，面向对象技术已经在许多领域中迅速地取代了传统的结构化技术。实践表明，采用先进的技术不仅可以提高软件开发和维护的效率，而且可以提高软件产品的质量。

5. 结果应该能够清楚地审查

软件产品不同于一般的物理产品，它是看不见摸不着的逻辑产品。软件开发人员的工作进展情况可见性差，难以准确度量，从而使得软件产品的开发过程比一般产品的开发过程更难于评价和管理。为了提高软件开发过程的可见性，更好地进行管理，应该依据软件开发项目的总目标和完成期限，规定开发小组的责任、产品标准及完成日期，从而使得所得到的结果能够清楚地审查。

6. 开发小组的人员应该少而精

这条基本原理的含义是，软件开发小组的组成人员的素质应该好，而小组人数则不宜过多。开发小组人员的素质和数量是影响软件产品质量和开发效率的重要因素。素质高的人员的开发效率比素质低的人员的开发效率可能高几倍至几十倍，而且素质高的人员所开发的软件中的错误明显少于素质低的人员所开发的软件中的错误。此外，随着开发小组人员数目的增加，为了交流信息、讨论问题而造成的通信开销也急剧增加。当开发小组人数为 N 时，可能的通信路径有 $N(N - 1)/2$ 条，可见随着人数 N 的增大，通信开销将急剧增加。因此，组成少而精的开发小组是软件工程的一条基本原理。

7. 承认不断改进软件工程实践的必要性

遵循上述 6 条基本原理，就能够按照当代软件工程的基本原理实现软件的工程化生产。但是，仅有上述 6 条基本原理还不能保证软件开发与维护的过程能赶上时代前进的步伐，能跟上技术的不断进步。因此，Boehm 提出把承认不断改进软件工程实践的必要性作为软件工程的第七条基本原理。按照这条原理，不仅要积极主动地采用新的软件技术，而且要注意不

断总结经验。例如，及时收集进度和资源消耗数据，收集出错类型和问题报告数据等。这些数据不仅可以用来评价新的软件技术的效果，而且可以用来指明必须着重开发的软件工具和应该优先研究的技术。

1.2.3 软件工程方法学

正如前面已经讲过的，软件工程包含管理和技术两方面的内容，是管理方法与软件技术紧密结合的工程学科。

所谓管理就是通过计划、组织和控制等一系列活动，合理地配置和使用各种资源，以达到既定目标的过程。

通常把在软件生命周期全过程中使用的一整套技术的集合称为方法学，也称为范型。在软件工程范畴中，这两个词的含义基本相同。

软件工程方法学包括三个要素，这就是方法、工具和过程。其中，方法是完成软件开发各项任务的技术方法，回答“如何做”的问题；工具为方法的运用提供自动的或半自动的软件支撑环境；过程是为了获得高质量的软件所需要完成的一系列任务的框架，它规定了完成各项任务的工作步骤，回答“何时做”的问题。

目前使用得最广泛的软件工程方法学，主要有传统方法学和面向对象方法学。

1. 传统方法学

传统方法学也称为生命周期方法学或结构化范型。它采用结构化技术（结构化分析、结构化设计、结构程序设计和结构化测试）来完成软件开发的各项任务，并使用适当的软件工具或软件工程环境来支持结构化技术的运用。这种方法学把软件生命周期的全过程依次划分为若干个阶段，然后顺序地逐步完成每个阶段的任务。采用这种方法学开发软件的时候，从对任务的抽象逻辑分析开始，一个阶段一个阶段地进行开发。前一个阶段任务的完成是开始进行后一个阶段工作的前提和基础，而前一阶段任务的完成通常是使前一阶段提出的解法更进一步具体化，加入了更多的实现细节。每一个阶段的开始和结束都有严格标准，对于任何两个相邻的阶段而言，前一阶段的结束标准就是后一阶段的开始标准。在每一个阶段结束之前都必须进行正式严格的技术审查和管理复审，从技术和管理两方面对这个阶段的开发成果进行检查，通过之后这个阶段才算结束；如果检查通不过，则必须进行必要的返工，并且返工后还要再经过审查。审查的一条主要标准就是每个阶段都应该交出“最新的”（即和所开发的软件完全一致的）高质量的文档资料，从而保证在软件开发工程结束时有一个完整准确的软件配置交付使用。文档是通信的工具，它们清楚准确地说明了到这个时候为止，关于该项工程已经知道了什么，同时确立了下一步工作的基础。此外，文档也起备忘录的作用，如果文档不完整，那么一定是某些工作忘记做了，在进入生命周期的下一阶段之前，必须补足这些遗漏的细节。在完成生命周期每个阶段的任务时，应该采用适合该阶段任务特点的系统化的技术方法——结构化分析、结构化设计、结构程序设计和结构化测试技术。

把软件生命周期划分成若干个阶段，每个阶段的任务相对独立，而且比较简单，便于不同人员分工协作，从而降低了整个软件开发工程的困难程度；在软件生命周期的每个阶段都采用科学的管理技术和良好的技术方法，而且在每个阶段结束之前都从技术和管理两个角度进行严格的审查，合格之后才开始下一阶段的工作，这就使软件开发工程的全过程以一种有条不紊的方式进行，保证了软件的质量，特别是提高了软件的可维护性。总之，采用生命周

期方法学可以大大提高软件开发的成功率，软件开发的生产率也能明显提高。

目前，生命周期方法学仍然是人们在开发软件过程中使用得非常广泛的软件工程方法学。由于这种方法学历史悠久，为广大软件工程师所熟悉，而且在开发某些类型的软件时也比较有效，因此，在相当长一段时间内这种方法学还会有生命力。此外，如果没有完全理解传统方法以及这种传统方法与面向对象方法的差别，也就不可能理解面向对象方法学为何优于传统方法学。本书不仅讲述面向对象方法学，也讲述传统方法学。

2. 面向对象方法学

当软件规模较大，或者对软件的需求是模糊的或随时间变化的时候，使用结构化范型开发软件往往不成功；此外，使用传统方法学开发出的软件，维护起来通常都很困难。

结构化范型只能获得有限成功的一个重要原因是，这种技术要么面向行为（即对数据的操作），要么面向数据，却没有既面向数据又面向行为的结构化技术。众所周知，软件系统本质上是信息处理系统。离开了操作便无法更改数据，而脱离了数据的操作是毫无意义的。数据和对数据的处理原本是密切相关的，把数据和处理人为地分离成两个独立的部分，自然会增加软件开发与维护的难度。与传统方法学相反，面向对象方法学把数据和行为看成同等重要，它是一种以数据为主线，把数据和对数据的操作紧密地结合在一起的方法。

概括地说，面向对象方法学具有下述四个要点。

？把对象（Object）作为融合了数据及在数据上的操作行为的统一的软件构件。面向对象程序是由对象组成的，程序中任何元素都是对象，复杂对象由比较简单的对象组合而成。

？把所有对象都划分成类（Class）。每个类都定义了一组数据和一组操作，类是对具有相同数据和相同操作的一组相似对象的定义。数据用于表示对象的静态属性，是对象的状态信息，而施加于数据之上的操作用于实现对象的动态行为。

？按照父类（或称为基类）与子类（或称为派生类）的关系，把若干个相关类组成一个层次结构的系统（也称为类等级）。在类等级中，下层派生类自动拥有上层基类中定义的数据和操作，这种现象称为继承。

？对象彼此之间仅能通过发送消息互相联系。对象与传统数据有本质区别，它不是被动地等待外界对它施加操作，相反，它是进行处理的主体，必须向它发消息请求它执行它的某个操作以处理它的数据，而不能从外界直接对它的数据进行处理。也就是说，对象的所有私有信息都被封装在该对象内，不能从外界直接访问，这就是通常所说的封装性。

面向对象方法学的出发点和基本原则，是尽可能模拟人类习惯的思维方式，使开发软件的方法与过程尽可能接近人类认识世界解决问题的方法与过程，从而使得描述问题的问题空间与实现解法的解空间在结构上尽可能一致。

传统方法学强调自上向下一个阶段接着一个阶段地顺序完成软件开发工作。事实上，人类认识客观世界解决现实问题的过程，是一个反复迭代的渐进过程，人的认识需要在继承以前的相关知识的基础上经过多次反复才能逐步深化。在人的认识深化的过程中，既包括了从一般到特殊的演绎思维过程，也包括了从特殊到一般的归纳思维过程。

用面向对象方法学开发软件的过程，是一个主动地多次反复迭代的演化过程。面向对象方法在概念和表示方法上的一致性，保证了各项开发活动之间的平滑（无缝）过渡。面向对象方法广泛使用的对象分类过程，支持从特殊到一般的归纳思维过程；而通过建立类等级获得的继承性，支持从一般到特殊的演绎思维过程。

正确运用面向对象方法学开发软件，则最终得到的软件产品是由许多较小的、基本上相互独立的对象组成，而且大多数对象都与现实世界中的实体相对应，因此，降低了软件产品的复杂性，提高了软件产品的可理解性，简化了软件的开发和维护工作。由于对象是相对独立的实体，容易在以后的软件产品中再次使用。因此，面向对象方法学的另一个重要优点是促进了软件重用。而面向对象方法学特有的继承性，则进一步提高了面向对象软件的可重用性。

1.3 软件生命周期

概括地说，软件生命周期由软件定义、开发和运行维护三个时期组成，每个时期又可以进一步划分成若干个阶段。

软件定义时期的任务是定义所要开发的软件，具体地说，就是确定软件开发工程必须完成的总目标；确定工程的可行性；导出实现工程目标应该采用的策略及软件必须具有的功能；估算完成该项开发工程需要的资源和成本，并且制定工程进度表。这个时期的工作通常又称为系统分析，由系统分析员采用结构化分析方法或面向对象分析方法完成。通常把软件定义时期进一步划分成问题定义、可行性研究和需求分析三个阶段。

开发时期具体设计和实现在前一个时期所定义的软件，它通常由下述四个阶段组成：概要设计，详细设计，编码和单元测试，综合测试。其中前两个阶段又称为系统设计，常用的设计方法有结构化设计方法和面向对象设计方法；后两个阶段又称为系统实现，常用的实现方法同样有结构化实现方法和面向对象实现方法。

运行维护时期的主要任务是使得软件持久地满足用户需要并长期为用户服务。具体地说，当软件在使用过程中发现错误时应该加以改正；当环境改变时应该修改软件以适应新的环境；当用户有新的要求时应该及时修改或扩充软件以满足用户的新需求。通常对维护时期不再进一步划分阶段，但是每一次维护活动实质上都是一次压缩和简化了的定义和开发过程。

下面简要地介绍上述各个阶段应该完成的基本任务。

1. 问题定义

问题定义阶段必须回答的关键问题是：“要解决的问题是什么？”如果不知道问题是什么就试图解决这个问题，显然是盲目的，只会白白浪费时间和金钱，最终得出的结果很可能是毫无意义的。尽管确切地定义问题的必要性是十分明显的，但是在实践中它却可能是最容易被忽视的一个步骤。

通过调研，系统分析员应该提出关于问题性质、工程目标和工程规模的书面报告，并且需要得到用户对这份报告的确认。

2. 可行性研究

这个阶段要回答的关键问题是：“上一个阶段所确定的问题是否有行得通的解决办法”。并非所有问题都有切实可行的解决办法，事实上，许多问题不可能在预定的系统规模或时间期限之内解决。如果问题没有可行的解决办法，那么花费在这项工程上的任何时间、资源和经费都是无谓的浪费。

可行性研究的目的就是用最小的代价在尽可能短的时间内确定问题是否能够解决。必须记住，可行性研究的目的不是解决问题，而是确定问题是否值得去解。怎样达到这个目的？

当然不能靠主观猜想而只能靠客观分析。系统分析员必须进一步概括地了解用户的需求，并在此基础上提出若干种可能的系统实现方案，对每种方案都从技术、经济、社会因素（例如，法律）等方面分析可行性，从而最终确定这项工程的可行性。

3. 需求分析

这个阶段的任务仍然不是具体地解决客户的问题，而是准确地回答“目标系统必须做什么”这个问题。

虽然在可行性研究阶段已经粗略了解了用户的需求，甚至还提出了一些可行的方案，但是，可行性研究的基本目的是用较小的成本在较短的时间内确定是否存在可行的解法，因此许多细节被忽略了。然而在最终的系统中却不能遗漏任何一个微小的细节，所以可行性研究并不能代替需求分析，它实际上并没有准确地回答“系统必须做什么”这个问题。

需求分析的任务还不是确定系统怎样完成它的工作，而仅仅是确定系统必须完成哪些工作，也就是对目标系统提出完整、准确、清晰、具体的要求。

用户了解他们所面对的问题，知道必须做什么，但是通常不能完整准确地表达出他们的要求，更不知道怎样利用计算机解决他们的问题；软件开发人员知道怎样用软件实现人们的要求，但是对特定用户的具体要求并不完全清楚。因此，系统分析员在需求分析阶段必须和用户密切配合，充分交流信息，以得出经过用户确认的系统需求。

这个阶段的另外一项重要任务，是用正式文档准确地记录对目标系统的需求，这份文档通常称为规格说明（specification）。

4. 概要设计

这个阶段的基本任务是，概括地回答“怎样实现目标系统？”这个问题。概要设计又称为初步设计、逻辑设计、高层设计或总体设计。

首先，应该设计出实现目标系统的几种可能的方案。软件工程师应该用适当的表达工具描述每种可能的方案，分析每种方案的优缺点，并在充分权衡各种方案的利弊的基础上，推荐一个最佳方案。此外，还应该制定出实现所推荐的方案的详细计划。如果客户接受所推荐的系统方案，则应该进一步完成本阶段的另一项主要任务。

上述设计工作确定了解决问题的策略及目标系统中应包含的程序，但是，对于怎样设计这些程序，软件设计的一条基本原理指出，程序应该模块化，也就是说，一个程序应该由若干个规模适中的模块按合理的层次结构组织而成。因此，概要设计的另一项主要任务就是设计程序的体系结构，也就是确定程序由哪些模块组成以及模块间的关系。

5. 详细设计

概要设计阶段以比较抽象概括的方式提出了解决问题的办法。详细设计阶段的任务就是把解法具体化，也就是回答“应该怎样具体地实现这个系统”这个关键问题。

这个阶段的任务还不是编写程序，而是设计出程序的详细规格说明。这种规格说明的作用很类似于其他工程领域中工程师经常使用的工程蓝图，它们应该包含必要的细节，程序员可以根据它们写出实际的程序代码。

详细设计也称为模块设计、物理设计或低层设计。在这个阶段将详细地设计每个模块，确定实现模块功能所需要的算法和数据结构。

6. 编码和单元测试

这个阶段的关键任务是写出正确的容易理解、容易维护的程序模块。

程序员应该根据目标系统的性质和实际环境，选取一种适当的高级程序设计语言（必要时用汇编语言），把详细设计的结果翻译成用选定的语言书写的程序，并且仔细测试编写出的每一个模块。

7. 综合测试

这个阶段的关键任务是通过各种类型的测试（及相应的调试）使软件达到预定的要求。

最基本的测试是集成测试和验收测试。所谓集成测试是根据设计的软件结构，把经过单元测试检验的模块按某种选定的策略装配起来，在装配过程中对程序进行必要的测试。所谓验收测试则是按照规格说明书的规定（通常在需求分析阶段确定），由用户（或在用户积极参加下）对目标系统进行验收。

必要时还可以再通过现场测试或平行运行等方法对目标系统进一步测试检验。

为了使用户能够积极参加验收测试，并且在系统投入生产性运行以后能够正确有效地使用这个系统，通常需要以正式的或非正式的方式对用户进行培训。

通过对软件测试结果的分析可以预测软件的可靠性；反之，根据对软件可靠性的要求，也可以决定测试和调试过程什么时候可以结束。

应该用正式的文档资料把测试计划、详细测试方案以及实际测试结果保存下来，做为软件配置的一个组成部分。

8. 软件维护

维护阶段的关键任务是，通过各种必要的维护活动使系统持久地满足用户的需要。

通常有四类维护活动：改正性维护，也就是诊断和改正在使用过程中发现的软件错误；适应性维护，即修改软件以适应环境的变化；完善性维护，即根据用户的要求改进或扩充软件使它更完善；预防性维护，即修改软件为将来的维护活动预先做准备。

虽然没有把维护阶段进一步划分成更小的阶段，但是实际上每一项维护活动都应该经过提出维护要求（或报告问题），分析维护要求，提出维护方案，审批维护方案，确定维护计划，修改软件设计，修改程序，测试程序，复查验收等一系列步骤，因此实质上是经历了一次压缩和简化了的软件定义和开发的全过程。

每一项维护活动都应该准确地记录下来，做为正式的文档资料加以保存。

我国国家标准《计算机软件开发规范》(GB8566-88)也把软件生命周期划分成8个阶段，这些阶段是：可行性研究与计划，需求分析，概要设计，详细设计，实现，组状测试，确认测试，使用和维护。其中，实现阶段即是编码与单元测试阶段，组装测试即是集成测试，确认测试即是验收测试。可见，国家标准中划分阶段的方法与前面讲的阶段划分方法基本相同，差别仅仅是：因为问题定义的工作量很小（一般只需要一天甚至更少的时间），没有把它作为一个独立的阶段列出来；因为综合测试的工作量过大而把它分解成两个阶段。

在实际从事软件开发工作时，软件规模、种类、开发环境及开发时使用的技术方法等因素，都影响阶段的划分。例如，开发规模较小的软件时，只需把软件生命周期划分成较少的几个阶段，相反，开发规模很大的软件时，则阶段应该划分得更细一些。

1.4 软件过程

在1.2.3小节中已经提到，软件工程过程是为了获得高质量的软件所需要完成的一系列任

务的框架，它规定了完成各项任务的工作步骤。事实上，承担的软件项目不同，应该完成的任务也有差异，没有一个适用于所有软件项目的任务集合。例如，适用于大型复杂项目的任务集合，对于小型简单的项目而言往往就过于复杂了。因此，一个科学、有效的软件工程过程应该定义一组适合于所承担的项目特点的任务。一个任务集合通常包括一组软件工程工作任务、里程碑和应该交付的产品（软件配置成分）。

总之，软件过程定义了运用软件开发方法的顺序、应该交付的文档资料、为保证软件质量和协调变化所需要采取的管理措施以及标志软件开发各个阶段任务完成的里程碑。为获得高质量的软件产品，软件过程必须科学而且合理。软件过程是构成软件工程方法学的一个重要的成分。

软件生命周期模型规定了把软件生命周期划分成哪些阶段及各个阶段的执行顺序，也称为软件过程模型，它是描述软件过程的一种常见的方式。

在上一节最后曾经讲过，实际承担软件开发工作时应该根据软件项目的特点来划分阶段，但是，下面讲述典型的软件过程模型时并不是针对某个特定项目讲的，因此，只能使用“通用的”阶段划分方法。由于下面将要讲述的瀑布模型与快速原型模型的主要区别是获取用户需求的方法不同，因此，在讲述软件过程模型时我们把“规格说明”作为一个阶段独立出来。此外，问题定义和可行性研究的一项主要任务都是概括地了解用户的需求，为了简洁地描述软件过程，把它们都归并到需求分析中去了。同样，为了简单起见，把概要设计和详细设计合并在一起称为“设计”。

下面介绍几种典型的软件过程模型。

1.4.1 瀑布模型

在 20 世纪 80 年代之前，瀑布模型一直是惟一被广泛采用的软件过程模型，现在它仍然是软件工程中应用得非常广泛的过程模型。图 1.2 所示为传统的瀑布模型。

按照传统的瀑布模型开发软件，有下述三个特点：

(1) 阶段间具有顺序性和依赖性

这个特点有两重含义：必须等前一阶段的工作任务完成之后，才能开始后一个阶段的工作；前一个阶段的输出文档就是后一个阶段的输入文档，因此，只有前一阶段的输出文档正确，后一阶段的工作才能获得正确的结果。但是，万一在软件生命周期的某一个阶段发现了错误，其根源很可能需要追溯到在它之前的一些阶段，通常还要修改前面阶段已经完成的文档。然而在软件生命周期后期改正早期阶段造成的错误，需要付出很高的代价。这就好像水已经从瀑布顶端流泻到底部，要想使它再返回高处需要付出很大能量一样。

(2) 推迟实现的观点

缺乏软件工程实践经验的软件开发人员，接到软件开发任务之后常常急于求成，总想尽早开始编写程序。但是，实践经验告诉我们，对于规模较大

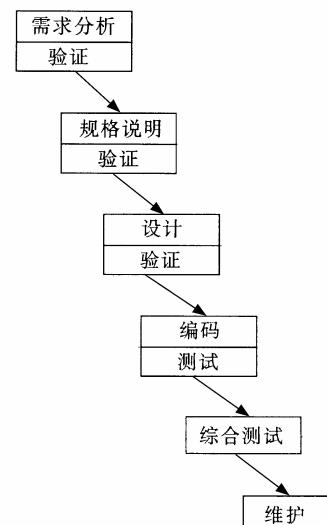


图 1.2 传统的瀑布模型

的软件项目来说，往往编码工作开始得越早最终完成开发工作所需要的时间反而越长。这是因为，前面阶段的工作没做或做得不扎实，就过早地编写程序，往往导致大量返工，有时甚至发生无法弥补的问题，带来灾难性的后果。

瀑布模型在编码之前明确地设置了系统分析与系统设计的各个阶段，分析与设计阶段的基本任务规定，在这两个阶段主要考虑目标系统的逻辑模型，不涉及软件的物理实现，从而可以防止软件开发人员过早地动手编写程序。

清楚地区分逻辑设计与物理设计，尽可能推迟软件的物理实现，是按照瀑布模型开发软件的一条重要的指导思想。

(3) 质量保证的观点

软件工程的基本目标是优质、高产。为了保证所开发出的软件的质量，在瀑布模型的每个阶段都坚持下述的两个重要做法：

? 每个阶段都必须完成规定的文档，没有交出合格的文档就是没有完成该阶段的任务。完整、准确的合格文档不仅是软件开发过程中各类人员之间相互沟通的媒介，也是运行时期对软件进行维护的重要依据。

? 每个阶段结束前都要对该阶段所完成的文档（或程序）进行评审（或测试），以便尽早发现问题，及时改正错误。事实上，如果没有在每个阶段结束前都进行评审，则越是早期阶段犯下的错误，暴露出来的时间就越晚，排除故障改正错误所需付出的代价也越高。因此，及时审查是保证软件质量、降低软件成本的重要措施。

但传统的瀑布模型过于理想化，事实上，人在工作过程中不可能不犯错误，评审也并不能保证发现所有错误，潜伏的错误将在后续阶段被陆续发现。在设计阶段可能发现规格说明文档中的错误，而设计上的缺陷或错误可能在实现过程中显现出来，在综合测试阶段将发现需求分析、设计或编码阶段的许多错误。当然，发现错误就必须及时改正，因此，实际的瀑布模型是带“反馈环”的，如图 1.3 所示（图中实线箭头表示开发过程，虚线箭头表示维护过程）。

当在后面阶段发现前面阶段所犯的错误时，需要沿图中左侧的“反馈线”返回前面的阶段，修正前面阶段的产品（文档或程序）之后再回来继续完成后面阶段的任务。

瀑布模型有下述的许多优点：

? 它可强迫开发人员采用规范的开发方法（例如，结构化技术）。

? 它严格地规定了每个阶段必须提交的文档。

? 它要求每个阶段所交出的一切产品都必须经过质量保证小组的仔细验证。

各个阶段产生的文档是维护软件产品时必不可少的，没有文档的软件几乎是不可能维护的。遵守瀑布模型的文档约束，每个阶段都提交合格的文档，将使软件维护变得比较容易一些。由于绝大部分软件预算都花费在软件维护上，因此，使软件变得比较容易维护就能显著降低软件预算。可以说，

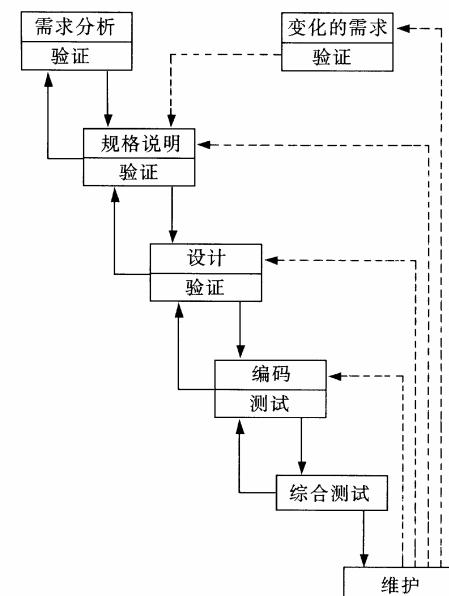


图 1.3 实际的瀑布模型

瀑布模型的成功在很大程度上是因为它基本上是一种文档驱动的模型。

但是，“瀑布模型是由文档驱动的”这个事实也是它的一个主要缺点。在可运行的软件产品交付给用户使用之前，用户只能通过文档来了解产品是什么样的。但是，仅仅通过写在纸上的静态的规格说明，很难全面正确地认识动态的软件产品，因此，用户很难评价规格说明书是否真正描述了他们的需求。而且事实证明，一旦一个用户开始使用一个软件，在他的头脑中关于该软件应该做什么的想法就会或多或少地发生变化，这就使得最初提出的需求变得不完全适用了。事实上，要求用户不经过实践就提出完整准确的需求，在许多情况下都是不切实际的。总之，由于瀑布模型几乎完全依赖于书面的规格说明，有可能导致最终开发出的软件产品不能真正满足用户的需要。

下一小节将介绍快速原型模型，它的优点是有助于保证用户的真实需要得到满足。

1.4.2 快速原型模型

所谓快速原型是快速建立起来的可以在计算机上运行的程序，它能完成的功能往往是最终的软件产品所能完成的功能的一个子集。如图 1.4 所示（图中实线箭头表示开发过程，虚线箭头表示维护过程），按照快速原型模型开发软件时，第一步是快速建立一个能反映用户主要需求的原型系统，让用户在计算机上试用它，通过实践来了解目标系统的概貌。通常，用户试用原型系统之后会提出许多修改意见，开发人员按照用户意见快速地修改原型系统，然后再次请用户试用，……。经过多次反复之后，一旦用户认为现在这个原型系统确实能做他们所需要的工作，开发人员便可以依照这个原型系统书写规格说明文档，根据这份文档开发出的软件应该能够满足用户的真实需要。

从图 1.4 可以看出，快速原型模型是不带反馈环的，这正是这种模型的主要优点：软件产品的开发基本上是线性顺序进行的。能做到基本上线性顺序开发的主要原因如下：

? 原型系统已经通过与用户的交互而得到验证，据此书写的规格说明文档正确地描述了用户需求，因此，在开发过程的后续阶段不会因为发现了规格说明文档的错误而进行较大的返工。

? 开发人员通过建立原型系统的过程已经学到了许多东西（至少知道了“系统不应该做什么，以及怎样防止做不该做的事情”），因此，在设计和编码阶段发生错误的可能性也比较小，这自然减少了在后续阶段需要改正在前面阶段所犯错误的可能性。

软件产品一旦交付给用户使用之后，维护活动便开始了。根据所需要完成的维护工作的类型，可能需要返回到需求分析、规格说明、设计或编码等不同阶段，如图 1.4 中虚线箭头所示。

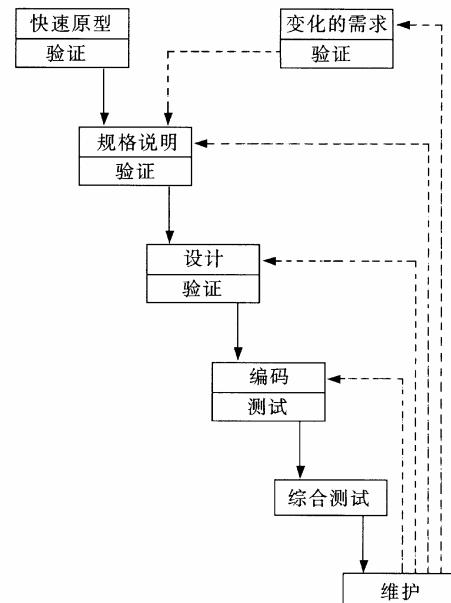


图 1.4 快速原型模型

快速原型模型的关键是“快速”。开发人员应该尽可能快地建造出（或修改好）原型系统，以加速软件开发过程，节约软件开发成本。原型的主要用途是获取用户的真实需求，一旦需求确定下来了，原型将被抛弃。因此，原型系统的内部结构并不重要，重要的是，必须迅速地构建出原型然后根据用户的意見迅速地修改原型，为此应该使用快速原型语言或工具来构建原型。

当快速原型系统的某个部分是利用软件工具由计算机自动生成的时候，可以把这部分用到最终的软件产品中。例如，用户界面通常是快速原型的一个关键部分，当使用屏幕生成程序和报表生成程序自动生成用户界面时，实际上可以把这样得到的用户界面用在最终的软件产品中。

1.4.3 增量模型

增量模型也称为渐增模型，如图 1.5 所示。使用增量模型开发软件时，把软件产品作为一系列的增量构件来设计、编码、集成和测试。每个构件由多个相互作用的模块构成，并且能够完成特定的功能。使用增量模型时，第一个增量构件往往实现软件的基本需求，提供最核心的功能。例如，使用增量模型开发字处理软件时，第一个增量构件可能提供基本的文件管理、编辑和文档生成功能；第二个增量构件提供更完善的编辑和文档生成功能；第三个增量构件实现拼写和语法检查功能；第四个增量构件完成高级的页面排版功能。把软件产品分解成增量构件时，应该使构件的规模适中，规模过大或过小都不好。最佳分解方法因软件产品特点和开发人员的习惯而异。分解时惟一必须遵守的约束条件是，当把新构件集成到现有软件中时，所形成的产品必须是可测试的。

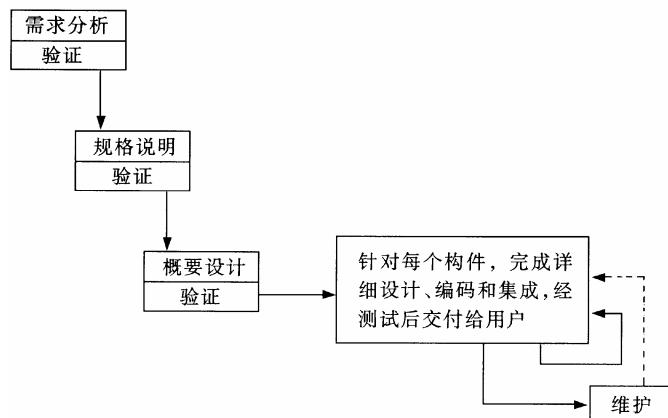


图 1.5 增量模型

采用瀑布模型或快速原型模型开发软件时，目标都是一次就把一个满足所有需求的产品提交给用户。增量模型则与之相反，它分批地逐步向用户提交产品，每次提交一个满足用户需求子集的可运行的产品。整个软件产品被分解成许多个增量构件，开发人员一个构件接一个构件地向用户提交产品。每次用户都得到一个满足部分需求的可运行的产品，直到最后一次得到满足全部需求的完整产品。从第一个构件交付之日起，用户就

能做一些有用的工作。显然，能在较短时间内向用户提交可完成一些有用的工作的产品，是增量模型的一个优点。

增量模型的另一个优点是，逐步增加产品功能可以使用户有较充裕的时间学习和适应新产品，从而减少一个全新的软件可能给客户组织带来的冲击。

使用增量模型的困难是，在把每个新的增量构件集成到现有软件体系结构中时，必须不破坏原来已经开发出的产品。此外，必须把软件的体系结构设计得便于按这种方式进行扩充，向现有产品中加入新构件的过程必须简单、方便，也就是说，软件体系结构必须是开放的。但是，从长远观点看，具有开放结构的软件拥有真正的优势，这样的软件的可维护性明显好于封闭结构的软件。因此，尽管采用增量模型比采用瀑布模型和快速原型模型需要更精心的设计，但在设计阶段多付出的劳动将在维护阶段获得回报。如果一个设计非常灵活而且足够开放，足以支持增量模型，那么，这样的设计将允许在不破坏产品的情况下进行维护。事实上，使用增量模型时开发软件和扩充软件功能（完善性维护）并没有本质区别，都是向现有产品中加入新构件的过程。

从某种意义上说，增量模型本身是自相矛盾的。它一方面要求开发人员把软件看作一个整体，另一方面又要求开发人员把软件看作构件序列，每个构件本质上都独立于其他构件。除非开发人员有足够的技术能力协调好这个明显的矛盾，否则用增量模型开发出的软件产品可能并不令人满意。

图 1.5 所示的增量模型要求，在开始实现各个构件之前就完成需求分析、规格说明和概要设计的全部工作。由于在开始构建第一个构件之前已经完成了总体设计工作，在总体框架内实现各个构件，因此风险比较小。如果一旦确定了用户需求之后，就并行地完成各个构件的规格说明、设计、编码和集成等工作，虽然有可能加快工程进度，但是，这样的增量模型将冒构件无法集成到一起的风险，笔者不提倡这种风险较大的增量模型。

1.4.4 螺旋模型

软件开发几乎总要冒一定的风险，例如，产品交付给用户之后用户可能对产品不满意，到了预定的交付日期软件可能还未开发出来，实际的开发成本可能超过了预算，产品完成之前一些关键的开发人员可能“跳槽”了，产品投入市场之前竞争对手发布了一个功能相近、价格更低的软件等等。软件风险是任何软件开发项目中都普遍存在的实际问题，项目越大，软件产品越复杂，承担该项目所冒的风险也越大。软件风险

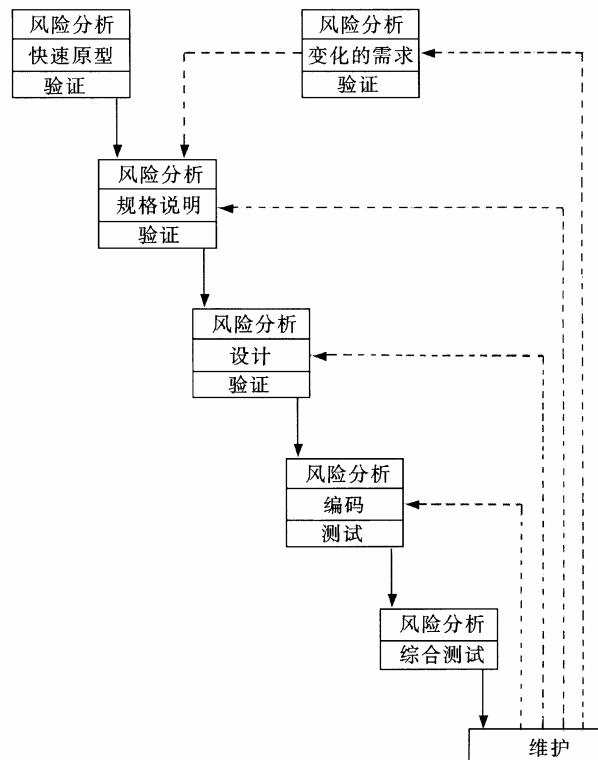


图 1.6 简化的螺旋模型

可能在不同程度上损害软件开发过程和软件产品质量。因此，在软件开发过程中必须及时识别和分析风险，并且采取适当措施以消除或减少风险的危害。

构建原型是一种能使某些类型的风险降至最低的方法。正如 1.4.2 小节所述，为了降低交付给用户的软件产品不能满足用户需要的风险，一种行之有效的方法是在需求分析阶段快速地构建出一个原型。在后续阶段中也可以通过构造适当的原型来降低某些技术风险。当然，原型并不是万能的，它并不能“包治百病”，对于某些类型的风险（例如，聘请不到所需要的的专业人员或关键的技术人员，在项目完成之前“跳槽”），原型方法是无能为力的。

螺旋模型的基本思想是，使用原型及其他方法以尽可能地降低风险。理解这种模型的一个简易方法，是把它看作在每个阶段之前都增加了风险分析过程的快速原型模型，如图 1.6 所示。

完整的螺旋模型如图 1.7 所示。图中带箭头的点划线的长度代表当前累计的开发费用，螺

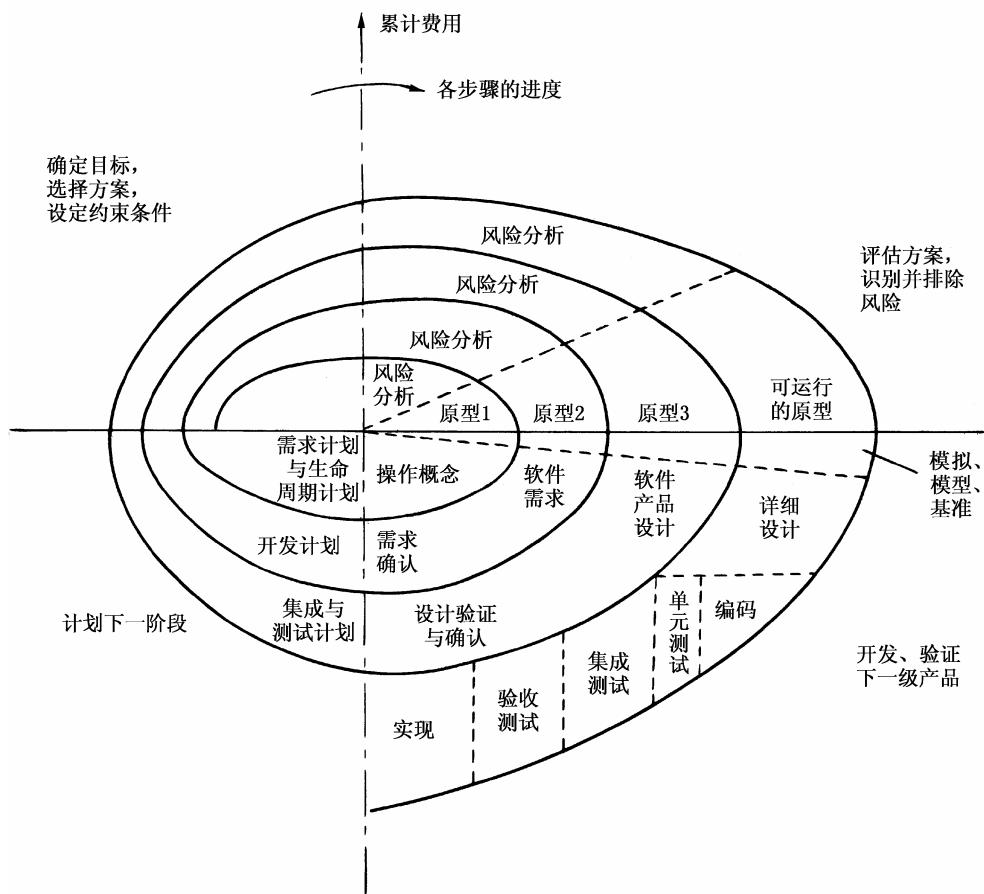


图 1.7

线旋过的角度值代表开发进度。螺旋线每个周期对应于一个开发阶段。每个阶段开始时（左上象限）的任务是，确定该阶段的目标、为完成这些目标选择方案及设定这些方案的约束条件。接下来的任务是，从风险角度分析上一步的工作结果。努力排除各种潜在的风险，通常用建造原型的方法来排除风险。如果风险不能排除，则停止开发工作或大幅度地削减项目规模。如果成功地排除了所有风险，则启动下一个开发步骤（右下象限），在这个步骤的工作过

程相当于纯粹的瀑布模型。最后是评价该阶段的工作成果并计划下一个阶段的工作。

螺旋模型有许多优点：对可选方案和约束条件的强调有利于已有软件的重用，也有助于把软件质量作为软件开发的一个重要目标；减少了过多测试（浪费资金）或测试不足（产品故障多）所带来的风险；更重要的是，在螺旋模型中维护只是模型的另一个周期，在维护和开发之间并没有本质区别。

螺旋模型主要适用于内部开发的大规模软件项目。如果进行风险分析的费用接近整个项目的经费预算，则风险分析是不可行的。事实上，项目越大，风险也越大，因此，进行风险分析的必要性也越大。此外，只有内部开发的项目，才能在风险过大时方便地中止项目。

螺旋模型的主要优势在于，它是风险驱动的，但是，这也可能是它的一个弱点。除非软件开发人员具有丰富的风险评估经验和这方面的专门知识，否则将出现真正的风险：当项目实际上正在走向灾难时，开发人员可能还认为一切正常。

1.5 小结

目前仍然有不少人错误地认为开发软件就是编写程序，或者认为开发软件的主要工作就是编写程序，这些错误认识危害很大，在这种思想指导下的错误做法导致出现了软件危机。

事实上，编写程序仅仅是开发软件所应完成的工作中的一部分，而且是一小部分。一个软件是由程序、数据和相关的文档组成的，程序仅仅是组成软件的一个配置成分，从这个角度看，软件开发也绝不等于编写程序。

为了消除软件危机，应该彻底清除错误认识，批判一些似是而非的错误论调，树立起关于软件和软件开发的正确认识；应该认真总结人类长期以来从事软件开发与维护的经验教训，借鉴其他工程领域的管理技术，逐步使软件工程这门新兴的工程学科发展和完善起来，并在软件工程实践中切实运用软件工程。

目前，传统的生命周期方法学和面向对象方法学是应用得最广泛的软件工程方法学。生命周期方法学把软件生命周期划分成若干个相对独立的阶段，每个阶段完成一些确定的任务，交出最终的软件配置的一个或几个成分；基本上按顺序完成各阶段的任务，在完成每个阶段的任务时采用行之有效的结构化技术和适当的软件工具；在每个阶段结束前都进行严格的技术审查和管理复审。

当软件规模较大或对软件的需求模糊易变时，采用生命周期方法学开发往往不成功，近年来在许多应用领域面向对象方法学已经迅速取代了传统方法学。面向对象方法学有四个要点，可以用下列方程式概括面向对象方法：

面向对象方法 = 对象 + 类 + 继承 + 用消息通信

也就是说，面向对象方法就是既使用对象又使用类和继承等机制，而且对象之间仅能通过传递消息实现彼此通信的方法。

面向对象方法简化了软件的开发和维护工作，提高了软件的可重用性。

软件工程方法学包括方法、工具和过程等三个要素。软件过程是为了获得高质量的软件产品所需要完成的一系列任务的框架，它规定了完成各项任务的工作步骤。软件过程必须科

学、合理，才能开发出高质量的软件产品。

按照在软件生命周期全过程中应完成的任务的性质，在概念上可以把软件生命周期划分成问题定义、可行性研究、需求分析、概要设计、详细设计、编码和单元测试、综合测试以及维护等 8 个阶段。实际从事软件开发工作时，软件规模、种类、开发环境及使用的技术方法等因素，都影响阶段的划分。因此，一个科学、有效的软件过程应该定义一组适合于所承担的项目特点的任务集合。

生命周期模型（即软件过程模型）规定了把生命周期划分成的阶段及各个阶段的执行顺序。本章介绍了四类典型的软件生命周期模型。

瀑布模型历史悠久、广为人知，它的优势在于它是规范的、文档驱动的方法；这种模型的问题是，最终交付的产品可能不是用户真正需要的。

快速原型模型正是为了克服瀑布模型的缺点而提出来的。它通过快速构建起一个可运行的原型系统，让用户试用原型并收集用户反馈意见的办法，获取用户的真实需求。

增量模型具有能在软件开发的早期阶段使投资获得明显回报和易于维护的优点，但是，要求软件具有开放结构是使用这种模型时固有的困难。

风险驱动的螺旋模型适用于大规模的内部开发项目，但是，只有在开发人员具有风险分析和排除风险的经验及专门知识时，使用这种模型才会获得成功。

每个软件开发组织都应该选择适合于本组织及所要开发的软件特点的软件生命周期模型。这样的模型应该把各种生命周期模型的合适特性有机地结合起来，以便尽量减少它们的缺点，充分利用它们的优点。

习 题 一

1. 什么是软件危机？为什么会出现软件危机？

2. 假设你是一家软件公司的总工程师，把图 1.1 给手下的软件工程师们看，告诉他们及早发现并改正错误的重要性。其中有人不同意你的观点，认为要求在错误进入软件之前就清除它们是不合理的，并指出“如果一个故障是编码错误造成的，那么一个人怎样能在设计阶段清除它呢？”你怎么反驳他？

3. 什么是软件工程？怎样运用软件工程消除（至少是缓解）软件危机？

4. 简述结构化范型和面向对象范型的要点，并比较这两种范型的优缺点。

5. 根据历史数据可以做出如下的假设。

（1）对计算机存储容量的需求大致按下面公式描述的趋势逐年增加：

$$M=4080e^{0.28(Y-1960)}$$

（2）存储器的价格按下面公式描述的趋势逐年下降：

$$P_1=0.3 \times 0.72^{Y-1974} \text{ (美分 / 位)}$$

如果计算机字长为 16 位，则存储器价格下降的趋势为：

$$P_2=0.048 \times 0.72^{Y-1974} \text{ (美元 / 字)}$$

在上列公式中 Y 代表年份， M 是存储容量（字数）， P_1 和 P_2 代表价格。

基于上述假设可以比较计算机硬件和软件成本的变化趋势。

(1) 在 1985 年对计算机存储容量的需求估计是多少？如果字长为 16 位，这个存储器的价格是多少？

(2) 假设在 1985 年一名程序员每天可开发出 10 条指令，程序员的平均工资是每月 4000 美元。如果一条指令为一个字长，计算使存储器装满程序所需用的成本。

(3) 假设在 1995 年存储器字长为 32 位，一名程序员每天可开发出 30 条指令，程序员的月平均工资为 6000 美元，重复(1)(2)题所问。

6. 美国某科幻电影中有一个描写计算机软件错误的故事，很富于戏剧性。故事情节如下：

由计算机 HAL 控制的宇宙飞船在飞往木星的旅途中，飞行指挥员鲍曼和 HAL 之间有一段对话。鲍曼命令道：“HAL，请对备用舱进行故障预报测试。”10s 钟后 HAL 报告：“一切正常”。

但是，地面上的飞行指挥中心在重复做了故障预报测试后，却得出了相反的结论：“鲍曼，我是飞行指挥中心，你的计算机在预报故障时可能犯了错误，我们的两台 HAL 计算机都得出了和你的计算机相反的结论”。

鲍曼用手指敲着控制台说：“HAL，是不是有什么东西干扰了你，以致出了这个差错”？

“听着，鲍曼，我知道你很想帮助我，但是我的信息处理是正常的。不信就查看我的记录吧，你会看到它是完全正确的”。

“我看过了你的服务记录，但是……谁都有可能犯错误啊”。

“我并不固执己见，但是，我是不可能犯错误的……”。

“喂，我是飞行指挥中心，我们已经彻底分析了你所遇到的麻烦，我们的两台计算机得出了完全一致的结论。问题出在故障预报系统中，我们确信是程序设计有错误。你必须断开你的计算机并改为地面控制模式，我们才能改正这个错误”。

当鲍曼断开计算机时，HAL 立即又把自己接了上去。最后，鲍曼只好拆下计算机的存储器，才得以控制他的宇宙飞船。

请问：

(1) 为什么鲍曼拆下存储器就能摆脱计算机的干扰而独自控制宇宙飞船？我们现在遇到的软件问题有这么严重吗？

(2) 如果不依靠飞行指挥中心，鲍曼怎样才能知道 HAL 的故障预报有问题？

(3) 应该怎样设计计算机系统，才能避免出现故事中描述的这些问题？

7. 什么是软件过程？它与软件工程方法学有何关系？

8. 假设你要开发一个软件，它的功能是把 73624.9385 这个数开平方，所得到的结果应该精确到小数点后 4 位。一旦实现并测试完之后，该产品将被抛弃。你打算选用哪种软件生命周期模型？请说明你做出这样选择的理由。

9. 假设你要为一家生产和销售长统靴的公司开发一个软件，该产品将监控该公司的存货：跟踪从购买橡胶开始，到靴子生产，发货给各个连锁店，直至卖给顾客的全过程。你在为这个项目选择生命周期模型时使用什么准则？

10. 列出在开发上一题所述软件产品的过程中可能遇到的风险。你打算怎样排除这些风险？

第2章 结构化分析

传统的软件工程方法学采用结构化分析技术完成系统分析（问题定义、可行性研究、需求分析）的任务。

所谓“结构化”，就是用一组标准的准则和工具来完成某项工作。具体说到结构化分析，它主要有以下几个要点：

- ? 采用自顶向下功能分解的方法；
- ? 强调逻辑功能而不是实现功能的具体方法；
- ? 使用图形（最主要的是数据流图）表达系统分析的结果。

2.1 可行性研究的任务

并不是任何问题都有简单明了的解决办法，事实上，许多问题不可能在预定的系统规模和期限之内解决。如果问题没有可行的解决办法，那么花费在这项开发工程上的时间、资源、人力和经费都是无谓的浪费。

可行性研究的目的就是，用最小的代价在尽可能短的时间内研究并确定所面临的问题是否能够解决。必须记住，可行性研究的目的不是解决问题，而是确定问题是否值得去解决。怎样达到这个目的呢？当然不能靠主观猜想而只能靠客观分析。必须分析几种主要的可能解法的利弊，从而判断原定的系统目标和规模是否现实，系统完成后所能带来的效益是否大到值得投资开发这个系统的程度。因此，可行性研究实质上是要进行一次大大压缩和简化了的系统分析和设计的过程，也就是在较高层次上以较抽象的方式进行的系统分析和设计的过程。

首先需要分析和澄清问题定义。例如，仅仅知道“用户需要一个计算机辅助设计系统，因为他们的手工设计系统很糟糕”是远远不够的，除非开发人员准确地知道目前使用的手工系统什么地方很糟糕，否则他们开发出的计算机辅助设计系统很可能也同样糟糕。类似地，如果一个微型计算机制造商打算开发一个新的操作系统，他首先应该做的工作就是听取广大用户对目前使用的操作系统的意见，准确地分析它不能令人满意的原因，否则新开发出的操作系统很可能仍然不能令人满意。只有开发人员对用户所面临的问题有了清楚、准确的认识之后，才能正确地回答出“什么是新产品必须做到的”这个关键问题。在问题定义阶段初步确定的系统规模和目标，如果是正确的就进一步加以肯定，如果有错误就应该及时改正，如果发现对目标系统有任何约束和限制，也必须把它们清楚地列举出来。

在澄清了问题定义之后，分析员应该导出系统的逻辑模型。然后从系统逻辑模型出发，

探索若干种可供选择的主要解法(即系统实现方案)。对每种解法都应该仔细研究它的可行性，一般说来，至少应该从下述三方面研究每种解法的可行性。

- (1) 技术可行性：使用现有的技术能实现这个系统吗？
- (2) 经济可行性：这个系统的经济效益能超过它的开发成本吗？
- (3) 操作可行性：系统的操作方式在这个用户组织内行得通吗？

必要时还应该进一步从法律、社会效益等更广泛的角度研究每种解法的可行性。

分析员应该为每个可行的解法制定一个粗略的实现进度。

当然，可行性研究最根本的任务是对以后的行动方针提出建议。如果问题没有可行的解，分析员应该建议停止这项开发工程，以避免时间、资源、人力和金钱的浪费；如果问题值得解，分析员应该推荐一个较好的解决方案，并且为工程制定一个初步的计划。

可行性研究需要的时间长短取决于工程的规模，一般说来，可行性研究的成本只是预期的工程总成本的 5% ~ 10%。

2.2 可行性研究过程

怎样进行可行性研究呢？典型的可行性研究过程有下述一些步骤。

1. 复查系统规模和目标

分析员访问关键的人员，仔细阅读和分析有关的资料，以便对问题定义阶段书写的关于系统规模和目标的报告书作进一步复查确认，改正含糊的或不确切的叙述，清楚地描述对目标系统的所有限制和约束。

这个步骤的工作，实质上是为了确保分析员正在解决的问题确实是要求解决的问题。

2. 研究目前正在使用的系统

现有的系统是信息的重要来源。显然，如果目前有一个系统正被人使用，那么这个系统必定能完成某些有用的工作，因此，新的目标系统必须也能完成它的基本功能；另一方面，如果现有的系统是完美无缺的，用户自然不会提出开发新系统的要求，因此，现有的系统必然有某些缺点，新系统必须能解决旧系统中存在的问题。此外，运行使用旧系统所需要的费用是一个重要的经济指标，如果新系统不能增加收入或减少使用费用，那么从经济角度看新系统就不如旧系统。

应该仔细阅读分析现有系统的文档资料和使用手册，也要实地考察现有的系统。应该注意了解这个系统可以做什么，为什么这样做，还要了解使用这个系统的代价。在了解上述这些信息的时候显然必须访问有关的人员。在调查访问时分析员和用户之间的关系有点类似于医生和病人的关系，用户叙述的往往是“症状”而不是实际问题，分析员必须分析解释所得的信息。

常见的错误做法是花费过多的时间去分析研究现有的系统。这个步骤的目的是了解现有的系统能做什么，而不是了解它怎样做这些工作。分析员应该画出描绘现有系统的高层系统流程图（参见 2.8 节），并请有关人员检验他对现有系统的认识是否正确。千万不要花费太多时间去了解和描述现有系统的实现细节。

没有一个系统是在“真空”中运行的，绝大多数系统都和其他系统有联系。分析员应该

注意了解并记录现有系统和其他系统之间的接口情况，这是设计新系统时的重要约束条件。

3. 导出新系统的高层逻辑模型

研究现有的系统的主要成果是，画出描绘目前正在使用的系统的高层系统流程图。系统流程图是描绘具体系统的好方法，但是，有时可能因为图中符号表达的物理含义过分的具体反而不适合需要。系统流程图是把系统逻辑功能和实现功能的具体方案融合在一起了。例如，图中不仅表示了哪些数据需要存储起来，而且具体表明了哪些数据是存储在磁盘上的，哪些数据是存储在磁带上的。但是，我们的目标并不是一成不变地复制现有的物理系统，而是建立一个能完成类似功能的新系统。因此，分析员希望以另一种方式总结从现有系统中获得的信息，不是准确地描绘具体的实现方法，而是着重描绘系统的逻辑功能。数据流图（参见 2.10 节）是完成这个目标的极好工具。

优秀的设计过程通常总是从现有的物理系统出发，导出现有系统的逻辑模型，再参考现有系统的逻辑模型，设想目标系统的逻辑模型，最后根据目标系统的逻辑模型建造新的物理系统。

通过前一步的工作，分析员对目标系统应该具有的基本功能和所受的约束已有一定了解，能够使用数据流图，描绘数据在系统中流动和处理的情况，从而概括地表达出他对新系统的设想。通常为了把新系统描绘得更清晰准确，还应该有一个初步的数据字典（参见 2.11 节），定义系统中使用的数据。数据流图和数据字典共同定义了新系统的逻辑模型，以后可以从这个逻辑模型出发设计新系统。

4. 进一步定义问题

新系统的逻辑模型实质上表达了分析员对新系统必须做什么的看法。用户是否也有同样的看法呢？分析员应该和用户一起再次复查问题定义、工程规模和目标，这次复查应该把数据流图和数据字典作为讨论的基础。如果分析员对问题有误解或者用户曾经遗漏了某些要求，那么现在是发现和改正这些错误的时候了。

可行性研究的前四个步骤实质上构成一个循环。分析员定义问题，分析这个问题，导出一个试探性的解；在此基础上再次定义问题，再一次分析这个问题，修改这个解；继续这个循环过程，直至提出的逻辑模型完全符合系统目标。

5. 导出和评价供选择的解法

分析员应该从他建议的系统逻辑模型出发，导出若干个较高层次的（较抽象的）物理解法供比较和选择。导出供选择解法的最简单的途径，是从技术角度出发考虑解决问题的不同方案。例如，在 2.10 节中将举例说明在数据流图上划分不同的自动化边界，从而导出不同的物理实现方案的方法。分析员可以划分出几组不同的自动化边界，然后针对每组边界考虑如何实现所要求的系统。

当从技术角度提出了一些可能的物理系统之后，应该根据技术可行性的考虑初步排除一些不现实的系统。例如，如果要求系统的响应时间不超过几秒钟，显然应该排除任何批处理方案。把技术上行不通的解法去掉之后，就剩下了一组技术上可行的方案。

其次可以考虑操作方面的可行性。分析员应该根据使用部门处理事务的原则和习惯检查技术上可行的那些方案，去掉其中从操作方式或操作过程的角度看用户不能接受的方案。

接下来应该考虑经济方面的可行性。分析员应该估计余下的每个可能的系统的开发成本和运行费用，并且估计相对于现有的系统而言这个系统可以节省的开支或可以增加的收入。

在这些估计数字的基础上，对每个可能的系统进行成本 / 效益分析（参见 2.13 节）。一般说来，只有投资预计能带来效益的系统才值得进一步考虑。

最后为每个在技术、操作和经济等方面都可行的系统制定实现进度表，目前这个进度表不需要（也不可能）制定得很详细，通常只需要估计出生命周期每个阶段的工作量就够了。

6. 推荐行动方针

根据可行性研究结果应该做出的一个关键性决定是，是否继续进行这项开发工程。分析员必须清楚地表明他对这个关键性决定的建议。如果分析员认为值得继续进行这项开发工程，那么他应该选择一种最好的解法，并且说明选择这个解决方案的理由。通常使用部门的负责人主要根据经济上是否划算决定是否投资于一项开发工程，因此分析员对于所推荐的系统必须进行比较仔细的成本 / 效益分析。

7. 草拟开发计划

分析员应该进一步为推荐的系统草拟一份开发计划，除了工程进度表之外还应该估计对各种开发人员（系统分析员，程序员，资料员等等）和各种资源（计算机硬件，软件工具等等）的需要情况，应该指明什么时候使用以及使用多长时间。此外还应该估计系统生命周期每个阶段的成本。最后应该给出下一个阶段（需求分析）的详细进度表和成本估计。

8. 书写文档提交审查

应该把可行性研究的上述各个步骤的工作结果写成清晰的文档，请用户和使用部门的负责人仔细审查，以决定是否继续这项工程以及是否接受分析员推荐的方案。

2.3 需求分析的任务

为了开发出真正满足用户需求的软件产品，首先必须确切地知道用户的需求。对软件需求的深入理解，是软件开发工作获得成功的前提和关键，不论我们把设计和编码工作做得多么出色，不能真正满足用户需求的软件只会给用户带来失望，给开发者带来烦恼。

需求分析是软件定义时期的最后一个阶段，它的基本任务是准确地回答“系统必须做什么？”这个问题。

虽然在可行性研究阶段已经粗略地了解了用户的需求，甚至还提出了一些可行的方案，但是，可行性研究的基本目的是用较小的成本在较短的时间内确定是否存在可行的解法，因此许多细节被忽略了。然而在最终的系统中却不能遗漏任何一个微小的细节，所以可行性研究并不能代替需求分析，它实际上并没有准确地回答“系统必须做什么？”这个问题。

需求分析的任务还不是确定系统怎样完成它的工作，而仅仅是确定系统必须完成哪些工作，也就是对目标系统提出完整、准确、清晰、具体的要求。

可行性研究阶段产生的文档，特别是数据流图，是需求分析的出发点。数据流图中已经划分出系统必须完成的许多基本功能，在需求分析阶段系统分析员将仔细研究这些功能并进一步将它们具体化。在这个阶段结束时交出的文档中应该包括详细的数据流图，数据字典和一组简明的算法描述。

需求分析的结果是系统开发的基础，关系到工程的成败和软件产品的质量。因此，必须用行之有效的方法对软件需求进行严格的审查验证。

下面简要地叙述需求分析阶段的具体任务。

1. 确定对软件系统的综合要求

对系统的综合要求主要有下述四个方面。

(1) 系统功能要求

应该划分出系统必须完成的所有功能。

(2) 系统性能要求

例如，联机系统的响应时间（即对于从终端输入的一个“事务”，系统在多长时间之内可以做出响应），系统需要的存储容量以及后备存储，重新启动和安全性等方面的考虑都属于性能要求。

(3) 运行要求

这类要求集中表现为对系统运行时所处环境的要求。例如，支持系统运行的系统软件是什么，采用哪种数据库管理系统，需要什么样的外存储器和数据通信接口等。

(4) 将来可能提出的要求

应该明确地列出那些虽然不属于当前系统开发范畴，但是据分析将来很可能会提出来的要求。这样做的目的是在设计过程中对系统将来可能的扩充和修改预做准备，以便一旦需要时能比较容易地进行这种扩充和修改。

2. 确定对系统的数据要求

任何一个软件系统本质上都是信息处理系统，系统必须处理的信息和系统应该产生的信息在很大程度上决定了系统的面貌，对软件设计有深远影响，因此，必须分析系统的数据要求，这是软件需求分析的一个重要任务。分析系统的数据要求通常采用建立概念模型的方法（参见 2.9 节）。

复杂的数据由许多基本的数据元素组成，数据结构表示数据元素之间的逻辑关系。利用数据字典可以全面准确地定义数据，但是数据字典的缺点是不够形象直观。为了提高可理解性，常常利用图形工具辅助描绘数据结构。常用的图形工具有层次方框图和 Warnier 图，在本章 2.12 节中将简要地介绍这两种图形工具。

3. 导出系统的逻辑模型

综合上述两项需求分析的结果，可以导出系统详细的逻辑模型。通常用数据流图、数据字典和主要的处理算法描述这个逻辑模型。

4. 修正系统开发计划

根据在需求分析过程中获得的对软件系统更深入、更具体的认识，可以较准确地估计系统的成本和进度，从而可以修正在前一个阶段所制定的开发计划。

5. 开发原型系统

在需求分析过程中使用原型系统的主要目的是，使得用户通过实践获得关于未来的系统将怎样为他们工作的更直接更具体的概念，从而可以更准确地提出和确定他们对所开发的软件的要求。

具体地说，把建立原型系统作为一种应该采取的策略的主要理由如下：

- ? 由于人类认识能力的局限性，用户往往不能预先提出全部要求；
- ? 在用户和系统分析员之间存在固有的通信鸿沟，分析员常常误解了用户的需求；
- ? 用户需要一个可运行的系统模型，以便获得有关未来系统的实践经验；
- ? 在软件开发过程中重复和反复是必要的和不可避免的；

? 目前已经有快速建立原型系统的工具可供选用。

用户试用了原型系统之后能够指出系统的哪些特性是他们喜欢的，哪些是他们感到不能接受的，以及系统中还应该增加哪些他们迫切需要的功能。根据经过实践检验的用户需求而开发出来的软件产品，通常能够真正满足用户的需要。因此，不仅在使用由快速原型模型描述的软件过程开发软件时必须建立原型系统。使用其他软件过程开发软件时通常也把建立原型系统作为获取用户需求的重要手段，特别当所开发的软件是全新的，用户缺乏使用类似系统经验时，更应该认真考虑开发原型系统的必要性和可能性。

2.4 需求分析的过程

在上一节中讲述了需求分析的主要任务，分析员怎样完成这些任务呢？

软件系统本质上是信息处理系统，而任何信息处理系统的基本功能都是把输入数据转变成需要的输出信息。数据决定了需要的处理和算法，看来数据显然是分析的出发点。在可行性研究阶段许多实际的数据元素被忽略了，当时分析员还不需要考虑这些细节，现在是定义这些数据元素的时候了。

结构化分析方法(简称 SA 方法)就是面向数据流自顶向下逐步求精进行需求分析的方法。通过可行性研究已经得出了目标系统的高层数据流图，需求分析的目的之一就是把数据流和数据存储定义到元素级。为了达到这个目的，通常从数据流图的输出端着手分析，这是因为系统的目标是产生这些输出，输出数据确定了系统必须具有的最基本的组成元素。

典型的需求分析过程主要由下述步骤组成。

1. 沿数据流图回溯

输出数据是由哪些元素组成的呢？通过调查访问不难搞清这个问题。那么，每个输出数据元素又是从哪里来的呢？既然它们是系统的输出，显然它们或者是从外面输入到系统中来的，或者是通过计算由系统中产生出来的。沿数据流图从输出端往输入端回溯，应该能够确定每个数据元素的来源，与此同时也就初步定义了有关的算法。但是，可行性研究阶段产生的高层数据流图，许多具体的细节没有包括在里面，因此沿数据流图回溯时常常遇到下述问题：为了得到某个数据元素需要用到数据流图中目前还没有的数据元素，或者得出这个数据元素需要用的算法尚不完全清楚。为了解决这些问题，往往需要向用户和其他有关人员请教，他们的回答使分析员对目标系统的认识更深入更具体了，系统中更多的数据元素被划分出来了，更多的算法被搞清楚了。通常把分析过程中得到的有关数据元素的信息记录在数据字典中(所谓 IPO 图即是输入 \ 处理 \ 输出图 参见本章 2.12.3 小节)。通过分析而补充的数据流、数据存储和处理，应该添加到数据流图的适当位置上。

2. 请用户复查

系统分析员已经把沿着数据流图回溯过程中所划分出来的数据元素记录在数据字典中，并且用 IPO 图或其他适当的工具扼要地记录了许多主要的算法，此外还补充和修正了在可行性研究阶段得到的数据流图。下一步应该做什么呢？可以肯定地说，还有许多问题存在。数据字典准确完整吗？算法正确吗？有没有遗漏必要的处理或数据元素？某些数据元素是从哪里来的？……分析员必须得出这些问题的确切答案。

关于一个系统的详细信息只能来源于直接在这个系统上工作的人——系统的用户。因此，必须请用户对前一个分析步骤中得出的结果仔细地进行复查，数据流图是帮助复查的极好工具。从输入端开始，分析员借助数据流图以及数据字典和简明的算法描述向用户解释输入数据是怎样一步一步地转变成输出数据的。这些解释集中反映了通过前面的分析工作分析员所获得的对目标系统的认识。这些认识正确吗？有没有遗漏？用户应该注意倾听分析员的报告，并及时纠正和补充分析员的认识。复查过程验证了已知的元素，补充了未知的元素，填补了文档中的空白。

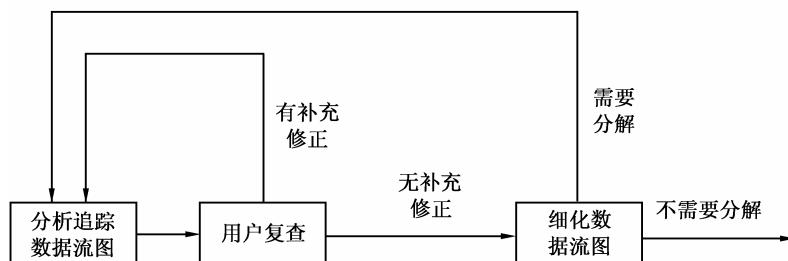
通过复查获得的新知识很可能又引出新的问题。例如，一个新补充进来的数据元素是怎样得出来的，它的源是什么？需要通过进一步的调查询问寻求这些问题的解答，当然，在这个过程中应该及时修改和补充数据流图、数据字典和 IPO 图（或描述算法的其他工具），把对于系统的新认识及时记录下来。因此，追踪数据流图和复查系统的逻辑模型这两个步骤实质上构成一个循环。对数据流图的分析产生问题，这些问题的答案使分析员对系统有更深入更具体的认识，同时也可能又引出新的问题，寻求这些新问题的答案导致对系统有了更进一步的认识，与此同时可能又发现了更深入的问题，……每经过一次循环都会了解到未来的逻辑系统的更多细节。在分析员对目标系统的认识螺旋式上升的过程中，用户及其他和系统有关的人员的参与是必不可少的：分析过程中产生的问题依靠他们来回答，分析员对系统的更深入的认识必须经过他们的检验和确认。在分析过程中必须充分重视和使用数据流图、数据字典和算法描述工具。为了完成这些正式文档必须收集必要的信息，遗漏的信息往往是明显的，这在开发复杂的大系统时尤其重要。人在处理复杂问题时很容易遗漏某些细节，正式的文档可以起备忘录作用，有助于消除这种危险。这些文档也是极好的通信工具，非常有助于审查和复查过程的成功，并且将成为软件工程下一个阶段工作的基础。

3. 细化数据流图

反复进行上述分析过程，分析员越来越深入地定义了系统中的数据和系统应该完成的功能。为了追踪更详细的数据流，分析员应该把数据流图扩展到更低的层次。

通过功能分解可以完成数据流图的细化。在数据流图中选出一个功能比较复杂的处理，并把它的功能分解成若干个子功能，这些较低层的子功能成为一张新数据流图上的处理，在这张新数据流图上还应该包括自己的数据存储和数据流。

对数据流图细化之后得到一组新的数据流图，不同的系统元素之间的关系变得更清楚了。对这组新数据流图的分析追踪可能产生新的问题，这些问题的答案可能又在数据字典中增加一些新条目，并且可能导致新的或精化的算法描述。随着分析过程的进展，经过发现问题和解答问题的反复循环，分析员越来越深入具体地定义了目标系统，最终得到对系统数据和功能需求的满意了解。图 2.1 粗略地概括了上述分析过程。



4. 修正开发计划

在可行性研究阶段，分析员根据当时对系统的认识，曾经草拟了一份开发计划。经过需求分析阶段的工作，分析员对目标系统有了更深入更具体的认识，因此可以对系统的成本和进度作出更准确的估计，在此基础上应该对开发计划进行修正。

5. 书写文档

经过分析确定了系统必须具有的功能和性能，定义了系统中的数据并且简略地描述了处理数据的主要算法。下一步应该把分析的结果用正式的文档记录下来，作为最终软件配置的一个组成成分。根据要求分析阶段的基本任务，在这个阶段可能应该完成下述四份文档资料：

系统规格说明。主要描述目标系统的概貌、功能要求、性能要求、运行要求和将来可能提出的要求。在分析过程中得出的数据流图是这份文档的一个重要组成部分，用 IPO 图或其他工具简要描述的系统主要算法是这份文档的另一个重要组成部分。此外，这份文档中还应该包括用户需求和系统功能之间的参照关系，以及设计约束等等。

数据要求。主要包括通过需求分析建立起来的数据字典，以及描绘数据结构的层次方框图或 Warnier 图。此外，还应该包括对存储信息（数据库或普通文件）分析的结果。

用户系统描述。这份文档从用户使用系统的角度描述系统，相当于一份初步的用户手册。内容包括对系统功能和性能的扼要描述，使用系统的主要步骤和方法，以及系统用户的责任等等。在软件开发过程的早期，准备一份初步的用户手册是非常必要的，它使得未来的系统用户能够从使用的角度检查审核目标系统，因此比较容易判断这个系统是否符合他们的需要。为了书写出这份文档，也迫使系统分析员从用户的观点考虑软件系统。有了这份文档审查和复审时就更容易发现不一致和误解的地方，这对保证软件质量和项目的成功是很重要的。

修正的开发计划。包括修正后的成本估计、资源使用计划、进度计划等等。

6. 审查和复审

分析过程的最后一步是按照需求分析阶段的结束标准，对本阶段的工作成果进行正式的技术审查和管理复审。

技术审查从技术角度审查分析员的工作成果，审查小组的任务是发现问题而不是解决问题，改正错误是分析员的责任。在分析员改正了审查过程中发现的问题之后，如果审查小组认为分析员确实理解了所要解决的用户问题，并且确实知道为了解决这些问题软件系统必须做什么，那么审查小组的成员应该在正式的审查表上签名，以指出迄今为止分析员所做的工作在技术上是正确的。

但是，在转入概要设计阶段之前还必须进行管理复审。虽然需求分析的结果在技术上是正确可行的，但是使用部门的负责人还关心其他问题，例如，修正后的成本和进度是否可以接受？经过管理复审获得批准之后，开发工作才能继续进行下去。

2.5 与用户沟通的方法

软件需求分析过程总是从两方或多方之间的沟通与交流开始。用户面临的问题需要使用基于计算机的方案来解决；软件开发者的任务是开发出真正满足用户需求的软件产品。通常，用户对他们所面临的问题有亲身体会，知道必须做什么，但是往往不能完整准确地表达出他

们的要求，更不知道怎样利用计算机软件解决他们的问题；软件开发人员知道怎样用软件实现人们的要求，但是对特定用户的具体要求并不完全清楚。这样就需要相互沟通和交流信息。但是，从开始通信到真正相互理解的道路通常是充满坎坷的，良好的通信技术有助于加快相互理解的过程。

正如我们在 2.3 节已经指出的那样，原型系统是促进沟通的重要手段，快速建立软件原型是最准确、最有效、最强大的需求分析技术。除了通过原型系统相互沟通之外，还有一些与用户通信的技术，本节将介绍两种典型技术。

2.5.1 访谈

访谈（或称为会谈）是最早开始运用的获取用户需求的技术，也是迄今为止仍然广泛使用的主要的需求分析技术。

访谈有两种基本形式，分别是正式的和非正式的访谈。在正式的访谈中，系统分析员将提出一些事先准备好的具体问题，例如，询问客户公司销售的商品种类、雇用的销售人员数目以及信息反馈时间应该多快等。在非正式的访谈中，将提出一些可以自由回答的开放性问题，以鼓励被访问的人员表达自己的想法，例如，询问用户为什么对目前正在使用的系统感到不满意。

当需要调查大量人员的意见时，向被调查的人员分发调查表是一个十分有效的做法。经过仔细考虑的书面回答可能比被访者对问题的口头回答更准确。系统分析员仔细阅读收回的调查表，然后再有针对性地访问一些用户，以便向他们询问在分析调查表时发现的新问题。

在对用户进行访谈的过程中使用情景分析技术往往非常有效。所谓情景分析就是对用户运用目标系统解决某个具体问题的方法和结果进行分析。例如，假定目标系统是一个制定减肥计划的软件，当给出某个肥胖症患者的年龄、性别、身高、体重、腰围及其他数据时，就出现了一个可能的情景描述。系统分析员根据自己对目标系统应具备的功能的理解，给出适用于该患者的菜单。客户公司的饮食学家可能指出，那些菜单对于有特殊饮食需求的病人（例如，糖尿病人、素食者）是不合适的。这就使系统分析员认识到，在目标系统制定菜单之前还应该先询问患者的特殊饮食需求。利用情景分析技术，使得系统分析员能够获知用户的具体需求。

情景分析的用处主要体现在下述两个方面：

? 它能在某种程度上演示产品的行为，从而便于用户理解，而且很可能进一步揭示出一些系统分析员目前还不知道的需求。

? 由于情景分析较易为用户所理解，因此，使用这种技术能保证用户在需求分析过程中始终扮演一个积极主动的角色。需求分析的目标是了解用户的真正需求，而这一信息的惟一来源是用户，因此，让用户起积极主动的作用对需求分析工作获得成功是至关重要的。

2.5.2 简易的应用规格说明技术

使用传统的访谈技术定义需求时，用户和开发者往往有意无意地区分“我们和他们”。由于不能做到像同一个团队的人那样同心协力地识别和精化需求，这种方法的效果有时并不理想（经常发生误解，还可能遗漏重要的信息）。

为了解决上述问题，人们研究出了一种面向团队的需求收集法，称为简易的应用规格

说明技术。这种方法提倡用户与开发者密切合作，共同标识问题，提出解决方案的要素，商讨不同的方法并指定基本的需求。今天，简易的应用规格说明技术已经成为信息系统界使用的主流技术。

尽管存在许多不同的简易应用规格说明方法，但是它们遵循的基本准则是相同的，如下所述：

- ？ 在中立地点举行由开发者和用户双方出席的会议。
- ？ 制定准备会议和参加会议的规则。
- ？ 提出一个议事日程，这个日程应该足够正式以便能够涵盖所有要点，同时这个日程又应该足够非正式，以便鼓励自由思维。
- ？ 由一个“协调人”来主持会议，他既可以是用户也可以是开发者还可以是从外面请来的人。
- ？ 使用一种“定义机制”（例如，工作表、图表等）。
- ？ 目标是标识问题、提出解决方案要素、商讨不同的方法以及在有利于实现目标的氛围中指定初步的需求。

通常，首先进行初步的访谈（参见 2.5.1 节），通过用户对基本问题的回答，对待解决的问题的范围和解决方案有了总体认识，然后开发者和用户都分别写出“产品需求”。选定会议地点、日期和时间，并选举一个协调人。邀请开发者和用户双方的代表出席会议，在会议日期之前把写好的产品需求分发给每位与会者。

要求每位与会者在开会的前几天认真复审产品需求，并且列出作为系统环境组成部分的对象、系统将产生的对象以及系统为了完成自己的功能将使用的对象。此外，还要求每位与会者列出操作这些对象或与这些对象交互的服务（即处理或功能）。最后，还应该列出约束条件（例如成本、规模、完成日期）和性能标准（例如速度、精度）。并不期望每位与会者列出的内容都是毫无遗漏的，但是，希望能准确表达出每个人对目标系统的认识。

会议开始之后，讨论的第一个议题是否需要这个新产品，一旦大家都同意确实需要这个新产品，每位与会者就应该展示他们在会前准备好的列表供大家讨论。可以把这些列表抄写在大纸上钉在墙上，或者写在白板上挂在墙上。理想的情况是，表中每一项都能单独移动，这样就能够方便地删除或增添表项，或者把不同的列表组合在一起。在这个阶段，严格禁止批评与争论。

在展示了每位与会者针对某个议题的列表之后，全组人员共同创建一张针对该议题的组合列表。在组合列表中消去了冗余的项，加入了在展示过程中产生的新想法，但是并不删除任何实质性的内容。在针对每个议题的组合列表都建立起来之后，由协调人主持讨论。经过讨论之后，组合列表将被缩短、加长或重新措辞，以便更恰当地描述所要开发的软件产品。讨论的目标是，针对每个议题（对象、服务、约束和性能）都创建出一张意见一致的列表。

一旦得出了意见一致的列表，就把与会者分成更小的小组，每个小组的工作目标是，为每张列表中的若干个项目制定出小型规格说明。小型规格说明是对列表中包含的词汇或短语的准确说明。

然后，每个小组都向全体与会者展示他们制定出的小型规格说明供大家讨论。通过讨论可能会增加或删除一些内容，也可能做一些进一步的精化工作。

在完成了小型规格说明之后，每位与会者都制定出软件产品的一整套确认标准，并把所

制定出的确认标准提交会议讨论，以创建出意见一致的确认标准。最后，由一名或多名与会者依据会议成果起草完整的规格说明。

简易的应用规格说明技术并不是解决需求分析阶段遇到的所有问题的“万能灵药”，但是，这种面向团队的需求收集法确实有许多突出的优点：

- ? 开发者与用户不分彼此，集思广益密切合作；
- ? 即时进行讨论和求精；
- ? 能够导出规格说明的具体步骤。

2.6 分析建模与规格说明

2.6.1 分析建模

系统分析，特别是需求分析的主要任务，是理解用户的需求定义所要开发的目标系统。为了更好地理解问题，人们常常采用建立模型的方法。在本书前面的章节中已经非正式地使用了术语“模型”，所谓模型就是为了理解事物而对事物做出的一种抽象，是对事物的一种无歧义的书面描述。通常，模型由一组图形符号和组织这些符号的规则组成。

从技术的角度看，软件工程实质上是从一系列建模活动开始的，这些建模活动导致对要求开发的软件要有完整的需求规格说明和全面的设计表示。

结构化分析就是一种建立结构化的分析模型的活动。从不同角度描述或理解软件系统，需要不同的模型。结构化分析主要建立功能模型和数据模型。数据流图是建立功能模型的基础，在本章 2.10 节将详细讲述数据流图。实体—联系图是用于建立数据模型的图形，将在 2.9 节中讲述。

2.6.2 软件需求规格说明书

在需求分析阶段除了创建分析模型之外，还应该写出完整的软件需求规格说明书，这些文档是软件配置的重要成分，在很大程度上决定着软件开发工程的成败。

由国家标准局发布的国家标准 GB8567-88 “计算机软件产品开发文件编制指南”，规定了在需求分析阶段应写出两份文档，分别是“软件需求说明书”和“数据要求说明书”。综合上述两份文档及其他资料的内容，本书以下给出了软件需求规格说明书的编写大纲，供读者在实际工作中使用。

. 引言

1. 编写目的

说明编写这份文档的目的，指出预期的读者。

2. 背景

指明项目的委托单位、开发单位、主管部门和用户，说明所开发的软件系统与其他系统的关系。

3. 定义

列出本文档中使用的专门术语的定义和缩写词的含义。

4. 参考资料

列出与本项目，特别是与本文档有关的参考资料，应该列出这些参考资料的标题、文件编号、发表日期和出版单位，说明可得到这些资料的来源。

. 任务概述

1. 目标

叙述该项软件开发的意图、应用目标、作用范围以及与其他相关软件之间的关系。如果所开发的软件是一个更大的系统的一个组成部分，则还应该说明它与该系统中其他各组成部分之间的关系。

2. 用户的特点

列出本软件的最终用户的特点，详细说明操作人员与维护人员的学历和技术专长，以及本软件的预期使用频度。这些是软件设计工作的重要约束。

3. 假定和约束

列出从事本项软件开发工作的假定和约束（例如经费限制、开发期限）。

. 数据的逻辑描述

1. 静态数据

所谓静态数据，指在运行过程中主要作为参考的数据，它们在很长一段时间内不变化，通常不随运行过程而改变。应该列出所有作为控制或参考的静态数据元素。

2. 动态数据

所谓动态数据，包括在运行过程中要发生变化的所有数据，以及在运行中要输入或输出的数据。应该列出所有动态输入或输出的数据元素，以及软件内部生成的数据。

3. 数据库

说明本软件所使用的数据库的名称和类型。

4. 数据字典

用数据字典定义本软件的数据。

5. 数据采集

说明数据采集的要求和范围，指明采集数据的方法，说明数据采集工作的承担者，列出对数据采集和预处理过程的专门规定。

. 功能需求

1. 功能分解

列出本软件应该具有的每一个具体功能，用适当形式（例如，数据流图）描述软件的整体结构及软件功能与其他系统元素的关系。

2. 功能描述

用列表（例如 IPO 表）的方式，逐项描述对本软件所提出的功能要求，说明输入什么量、经过怎样的处理过程、得到什么输出。

. 性能需求

1. 精度

说明对本软件的输入、输出数据精度的要求，必要时还应说明对传输过程中精度的要求。

2. 时间特性

说明对本软件的时间特性（例如，响应时间、更新处理时间、解题时间、数据转换与传

送时间)的要求。

3. 灵活性

说明对本软件的灵活性的要求，即当环境或需求发生变化时，本软件对这些变化应有的适应能力。

. 运行环境需求

1. 设备

描述运行本软件所需的硬件设备

2. 支撑软件

列出支持本软件的支撑软件，包括要用到的操作系统、编译程序、测试支持软件等。

3. 用户界面

说明需要的屏幕格式、报表格式、菜单格式、输入输出时间等。

4. 硬件接口

说明本软件与硬件之间的接口、数据通信协议等。

5. 软件接口

说明本软件与其他软件之间的接口、数据通信协议等。

. 其他专门需求

列出使用部门对安全保密、使用方便性、可维护性、可靠性、可移植性等方面特殊要求。

. 将来可能提出的需求

1. 功能扩充

列出预期将来可能要求增添的功能。

2. 性能提高

列出预期将来可能提出的提高性能的要求。

3. 其他需求

列出预期将来可能提出的其他需求。

. 确认标准

1. 性能范围

列出用户可以接受的性能变化范围。

2. 测试种类

列出为了确认本软件的功能、性能和约束符合用户需要，应该进行的测试的类型。

3. 预期的软件响应

列出预期本软件对每类测试的响应。

以上是本书推荐的软件需求规格说明书编写提纲。其中最重要然而又最常被忽略的内容，可能就是“确认标准”。我们怎样判断软件开发是否成功？为了确认软件产品的功能、性能和约束确实符合用户的需要，应该进行哪些测试呢？之所以会忽略这些内容，是因为要写出他们需要对软件需求有透彻的理解，然而有时分析员在这个阶段还未能做到彻底地理解软件需求。实际上，写出确认标准的过程是对其他所有需求的隐式复审，因此，把时间和精力用到这部分内容上是很值得的。

在许多情况下，软件需求规格说明书可能都附有可执行的原型（在某些情况下可替代规

格说明书)及初步的用户手册。用户手册把软件产品看作一个黑盒子，也就是说，手册重点描述用户的输入和软件的输出结果。通过该手册往往能发现人机界面问题。

2.7 验证软件需求

2.7.1 至少从四个方面验证软件需求

需求分析阶段的工作结果是开发软件系统的重要基础，大量统计数字表明，软件系统中15%的错误起源于错误的需求。为了提高软件质量，确保软件开发成功，降低软件开发成本，一旦对目标系统提出一组要求之后，必须严格验证这些需求的正确性。一般说来，应该从下述四个方面进行验证：

? 一致性

所有需求必须是一致的，任何一条需求都不能和其他需求互相矛盾。

? 完整性

需求必须是完整的，软件需求规格书应该包括用户需要的每一个功能或性能。

? 现实性

指定的需求应该是用现有的硬件技术和软件技术基本上可以实现的。一般说来，对硬件技术的进步可以做些预测，对软件技术的进步则很难做出预测，只能从现有技术水平出发判断需求的现实性。

? 有效性

必须证明需求是正确有效的，确实能解决用户所面临的问题。

2.7.2 验证软件需求的方法

上一小节已经指出，至少必须从一致性、完整性、现实性和有效性等四个不同角度验证软件需求的正确性。那么，怎样验证软件需求的正确性呢？验证的角度不同，验证的方法也不同。

1. 验证需求的一致性

当需求分析的结果是用自然语言书写的时候，除了靠人工技术审查验证软件系统规格说明书的正确性之外，目前还没有其他更好的“测试”方法。但是，这种非形式化的规格说明书是难于验证的，特别在目标系统规模庞大、规格说明书篇幅很长的时候，人工审查的效果是没有保证的，冗余、遗漏和不一致等问题可能没被发现而继续保留下来，以致软件开发工作不能在正确的基础上顺利进行。

为了克服上述困难，人们提出了形式化的描述软件需求的方法。当软件需求规格说明书是用形式化的需求陈述语言书写的时候，可以用软件工具验证需求的一致性(参见2.7.3节)，从而能有效地保证软件需求的一致性。

2. 验证需求的现实性

为了验证需求的现实性，分析员应该参照以往开发类似系统的经验，分析用现有的软、硬件技术实现目标系统的可能性。必要的时候应该采用仿真或性能模拟技术，辅助分析软件

需求规格说明书的现实性。

3. 验证需求的完整性和有效性

只有目标系统的用户才真正知道软件需求规格说明书是否完整、准确地描述了他们的需求。因此，检验需求的完整性，特别是证明系统确实满足用户的实际需要（即，需求的有效性），只有在用户的密切合作下才能完成。然而许多用户并不能清楚地认识到他们的需要（特别在要开发的系统是全新的，以前没有使用类似系统的经验时，情况更是如此），不能有效地比较陈述需求的语句和实际需要的功能。只有当他们有某种工作着的软件系统可以实际使用和评价时，才能完整确切地提出他们的需要。

理想的做法是先根据需求分析的结果开发出一个软件系统，请用户试用一段时间以便能认识到他们的实际需要是什么，在此基础上再写出正式的“正确的”规格说明书。但是，这种做法将使软件成本增加一倍，因此实际上几乎不可能采用这种方法。使用原型系统是一个比较现实的替代方法，开发原型系统所需的成本和时间可以大大少于开发实际系统所需要的。用户通过试用原型系统，也能获得许多宝贵的经验，从而可以提出更符合实际的要求。

使用原型系统的目的，通常是显示目标系统的主要功能而不是性能。为了达到这个目的可以使用超高级语言或第四代语言实现原型系统，并且可以适当降低对接口、可靠性和程序质量的要求，此外还可以省掉许多文档资料方面的工作，从而可以大大降低原型系统的开发成本。

2.7.3 用于需求分析的软件工具

为了更有效地保证软件需求的正确性，特别是为了保证需求的一致性，需要有适当的软件工具支持需求分析工作。这类软件工具应该满足下列要求。

(1) 必须有形式化的语法（或表），因此可以用计算机自动处理使用这种语法说明的内容；

(2) 使用这个软件工具能够导出详细的文档；

(3) 必须提供分析（测试）规格说明书的不一致性和冗余性的手段，并且应该能够产生一组报告指明对完整性分析的结果；

(4) 使用这个软件工具之后，应该能够改进通信状况。

作为需求工程方法学的一部分，在1977年设计完成了RSL（需求陈述语言）。RSL中的语句是计算机可以处理的，处理以后把从这些语句中得到的信息集中存放在一个称为抽象系统语义模型（ASSM）的数据库中。有一组软件工具处理ASSM数据库中的信息以产生出用PASCAL语言书写的模拟程序，从而可以检验需求的一致性、完整性和现实性。

1977年美国密执安大学开发了PSL/PSA（问题陈述语言/问题陈述分析程序）系统。这个系统是“计算机辅助设计和规格说明分析工具（CADSAT）”的一部分，它的基本结构类似于RSL。其中PSL是用来描述系统的形式语言，PSA是处理PSL描述的分析程序。用PSL描述的系统属性放在一个数据库中。一旦建立起数据库之后即可增加信息、删除信息或修改信息，并且保持信息的一致性。PSA对数据库进行处理以产生各种报告，测试不一致性或遗漏，并且生成文档资料。

2.8 系统流程图

在进行可行性研究时需要了解和分析现有的系统，并以概括的形式表达对现有系统的认识；进入设计阶段以后应该把设想的新系统的逻辑模型转变成物理模型，因此需要描绘未来的物理系统的概貌。

怎样概括地描绘一个物理系统呢？系统流程图是描绘物理系统的传统工具。它的基本思想是用图形符号以黑盒子形式描绘系统里面的每个部件（程序、文件、数据库、表格、人工过程等等）。系统流程图表达的是信息在系统各部件之间流动的情况，而不是对信息进行加工处理的控制过程，因此尽管系统流程图使用的某些符号和程序流程图中用的符号相同，但是它却是物理数据流图而不是程序流程图。

2.8.1 系统流程图的符号

当以概括的方式抽象地描绘一个系统时，仅使用图 2.2 中列出的基本符号就够了，其中每个符号表示系统中的一个部件。

符 号	名 称	说 明
	处理	能改变数据值或数据位置的加工或部件，例如，程序、处理机、人工加工等都是处理
	输入/输出	表示输入或输出(或既输入又输出)，是一个广义的不指明具体设备的符号
	连接	指出转到图的另一部分或从图的另一部分转来，通常在同一页上
	换页连接	指出转到另一页图上或由另一页图转来
	数据流	用来连接其他符号，指明数据流动方向

图 2.2 基本符号

当需要更具体地描绘一个物理系统时，还需要使用图 2.3 中列出的系统符号。利用这些系统符号可以把一个广义的输入 / 输出操作具体化为读 / 写存储在某种特殊设备（例如磁盘）上的文件（或数据库），把抽象的处理具体化为特定的程序或人工操作等等。

2.8.2 举例

介绍系统流程图的最好方法可能是举一个具体例子说明它的用法，下面是一个简单的例子。

某装配厂有一座存放零件的仓库，仓库中现有的各种零件的数量以及每种零件的库存量临界值等数据，记录在库存清单主文件中。当仓库中零件数量有变化时，应该及时修改库存清单主文件，以反映零件数量的变化，如果哪种零件的库存量少于它的库存量临界值，则应该报告给采购部门以便定货，规定每天向采购部门送一次定货报告。

符号	名称	说明
	穿孔卡片	表示用穿孔卡片输入或输出，也可表示一个穿孔卡片文件
	文档	通常表示打印输出，也可表示用打印终端输入数据
	磁带	磁带输入/输出，或表示一个磁带文件
	联机存储	表示任何种类的联机存储，包括磁盘、磁鼓、软盘和海量存储器件等
	磁盘	磁盘输入/输出，也可表示存储在磁盘上的文件或数据库
	磁鼓	磁鼓输入/输出，也可表示存储在磁鼓上的文件或数据库
	显示	CRT终端或类似的显示部件，可用于输入或输出，也可既输入又输出
	人工输入	人工输入数据的脱机处理，例如，填写表格
	人工操作	人工完成的处理，例如，会计在工资支票上签名
	辅助操作	使用设备进行的脱机操作
	通信链路	通过远程通信线路或链路传送数据

图 2.3 系统符号

该装配厂使用一台小型计算机处理更新库存清单主文件和产生定货报告的任务。零件库存量的每一次变化称为一个事务，由放在仓库中的终端输入到计算机中；系统中的库存清单程序对事务进行处理，更新存储在磁盘上的库存清单主文件，并且把必要的定货信息写在磁带上。最后，每天由报告生成程序读一次磁带，并且打印出定货报告。图 2.4 的系统流程图描绘了上述系统的概貌。

注意图 2.4 如何描绘这个物理系统。图中每个符号用黑盒子形式定义了组成系统的一个部件，然而并没有指明每个部件的具体工作过程；图中的箭头确定了信息通过系统的逻辑路径（信息流动路径）。

系统流程图的习惯画法是使信息在图中从顶向下或

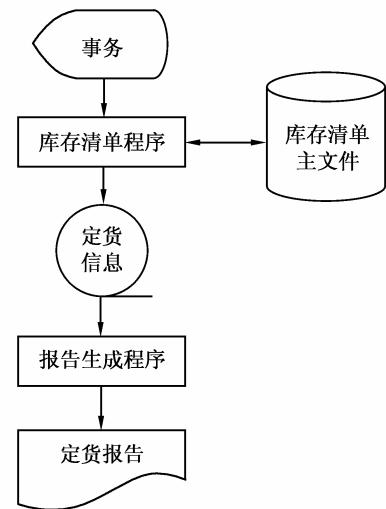


图 2.4 库存清单系统的系统流程图

从左向右流动。

图2.4中每个符号都有名称，因此可以起文档的作用。许多分析员喜欢在系统流程图上加更详细的注释，有些人甚至另加一页纸来解释系统流程图。

2.8.3 分层画系统流程图

描绘复杂系统的系统流程图包含很多符号，不能画在一页纸上。用换页连接符号虽然可以把一张大的系统流程图分别画在几页纸上，但是这种由多页纸组成的图通常很难阅读和理解。

面对复杂的系统时，一个比较好的方法是分层次地描绘这个系统。首先用一张高层次的系统流程图描绘系统总体概貌，表明系统的关键功能。然后分别把每个关键功能扩展到适当的详细程度，画在单独的一页纸上。这种分层次的描绘方法便于阅读者按从抽象到具体的过程逐步深入地了解一个复杂的系统。

例如，上述库存清单系统可以分为两个层次来描绘（只是为举例说明分层次描绘的方法，实际上这样简单的系统用不着分层描绘）。第一张系统流程图上只有三个符号：加工库存清单和产生报告这样两个处理，及连接上述两个处理的一个磁带文件（见图2.5）。在第二页纸上进一步描绘加工库存清单的过程，画出了终端、库存清单程序、磁盘主文件和磁带（见图2.6）。图中的换页连接符表明这张图逻辑上与第三张图连接。第三张图详细描绘产生报告的过程（见图2.7）。

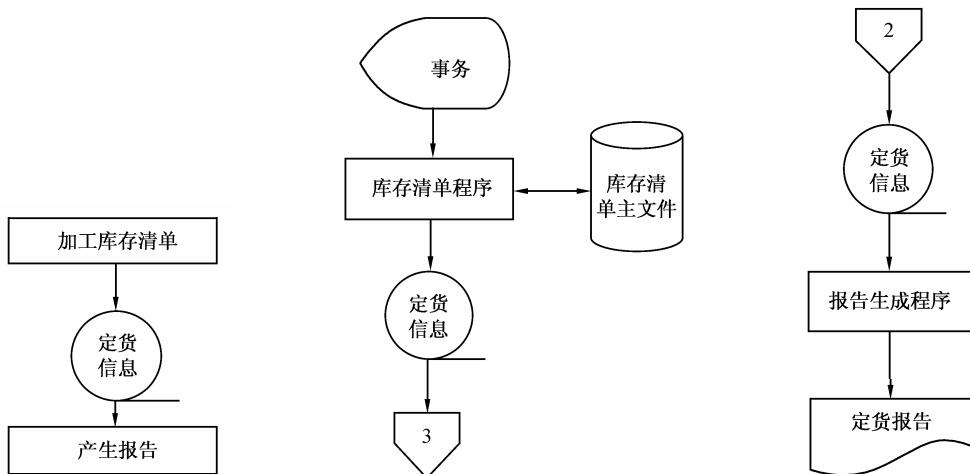


图2.5 库存清单系统的高层流程图

图2.6 加工库存清单的过程

图2.7 产生报告的过程

2.8.4 系统流程图的用途

分析员通常必须研究现有的物理系统，目的不是提供完整的文档资料，只是为了了解这个系统。画系统流程图是概括大量技术信息以及发现遗漏疏忽的极好方法。

在设计阶段需要把设计结果从抽象的逻辑模型转变成具体的物理系统，用特定的程序或过程代替广义的处理，用具体的文件或数据库代替一般的数据存储。画出系统流程图就可以具体设想系统将如何实现。总之，不论在分析阶段还是设计阶段，系统流程图作为一种清晰

简明的通信手段都是很有价值的，非常有助于开发人员和用户交流信息。

当一个复杂的大系统由许多组软件工程师共同开发时，系统流程图可以作为公共参考文件，指明每个开发小组的工作在系统中的地位。

2.9 实体-联系图

软件系统本质上是信息处理系统，因此，在软件系统的整个开发过程中都必须考虑两方面的问题——“数据”及对数据的“处理”。在需求分析阶段则既要分析用户的数据要求（即需要有哪些数据、数据之间有什么联系、数据本身有什么性质、数据的结构等等），又要分析用户的处理要求（即对数据进行哪些处理、每个处理的逻辑功能等等）。

为了把用户的数据要求清晰明确地表达出来，系统分析员通常建立一个概念性的数据模型（也称为信息模型）。概念性数据模型是一种面向问题的数据模型，是按照用户的观点来对数据和信息建模。它描述了从用户角度看到的数据，它反映了用户的现实环境，且与在软件系统中的实现方法无关。

最常用的表示概念性数据模型的方法，是实体-联系方法（Entity-Relationship Approach）。这种方法用实体-联系图（也称为实体-关系图或ER图）描述现实世界中的实体，而不涉及这些实体在系统中的实现方法。用这种方法表示的概念性数据模型又称为ER模型。

实体-联系模型中包含“实体”、“联系”和“属性”等三个基本成分，下面分别介绍这三个基本成分。

1. 实体

实体是客观世界中存在的且可相互区分的事物。实体可以是人也可以是物；可以是具体事物也可以是抽象概念。例如，职工、学生、课程、教师等都是实体。

在ER图中用矩形框代表实体（见图2.8）。

2. 联系

客观世界中的事物彼此间往往是有联系的。例如，教师与课程间存在“教”这种联系，而学生与课程间则存在“学”这种联系。联系可分为以下三类：

(1) 一对一联系(1 1)

例如，一个丈夫有一个妻子，而每个妻子只有一个丈夫，丈夫与妻子的联系是一对一的。

(2) 一对多联系(1 N)

例如，某校教师与课程之间存在一对多的联系“教”，即每位教师可以教多门课程，但是每门课程只能由一位教师来教（见图2.8）。

(3) 多对多联系(M N)

例如，图2.8表示学生与课程间的联系（“学”）是多对多的，即一个学生可以学多门课程，而每门课程可以有多个学生来学。

在ER图中，用连接相关实体的菱形框表示联系（见图2.8）。

3. 属性

属性是实体或联系所具有的性质。通常一个实体由若干个属性来刻画，例如，“学生”实体有学号、姓名、性别、系、年级等属性；“教师”实体有教工号、姓名、性别、职称、职务

等属性；“课程”实体有课程号、课名、学时、学分等属性（见图2.8）。

联系也可能有属性。例如，学生“学”某门课程所取得的成绩，既不是学生的属性也不是课程的属性。由于“成绩”既依赖于某名特定的学生又依赖于某门特定的课程，所以这是学生与课程之间的联系“学”的属性（见图2.8）。

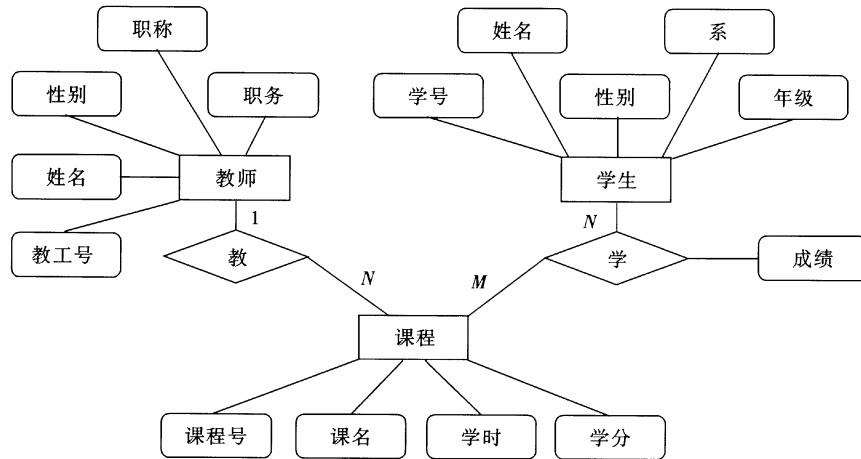


图2.8 某校教学管理ER图

应该根据所要解决的问题，来确定特定的数据（即实体）的一组合适的属性。例如，为了开发机动车管理系统，描述汽车的属性应该是制造厂、品牌、型号、发动机号码、车体类型、颜色、车主姓名、住址、驾驶证号码、生产日期以及购买日期等。但是，为了开发设计汽车的CAD系统，用上述这些属性描述汽车就不合适了。显然，车主姓名、住址、驾驶证号码、生产日期和购买日期等属性应该删去，而描述汽车技术指标的大量属性应该增添进来。

在ER图中，用椭圆形或圆角矩形表示实体（或联系）的属性，并用直线把实体（或联系）与其属性连接起来（见图2.8）。

人们通常就是用实体、联系和属性这三个概念来理解现实问题的，因此，ER模型比较接近人的习惯思维方式。此外，ER模型使用简单的图形符号表达分析员对所要解决的问题的理解，不熟悉计算机技术的用户也能理解它，因此，ER模型可以作为分析员与用户之间有效的交流工具。

2.10 数据流图

当数据在软件中“移动”时，它将被一系列“变换”所修改。数据流图（DFD）是一种形象直观的图形，它描绘数据在软件中从输入移动到输出的过程中所经受的变换（即加工处理）。在数据流图中没有任何具体的物理元素，它仅仅描绘数据在软件中流动和被处理的情况。因为数据流图是软件系统逻辑功能的图形表示，即使不是专业的计算机技术人员也很容易理解，所以它是分析员与用户之间极好的通信工具。此外，设计数据流图时只需考虑系统必须完成的基本逻辑功能，完全不需要考虑实现这些功能的具体方法。因此，它也是今后进行软

件设计的很好的出发点。

可以在任何抽象层次上用数据流图描绘软件系统。事实上，可以分层次地画数据流图，高层数据流图描绘系统的整体概貌，层次越低表现出的信息流细节和功能细节也越多。

2.10.1 数据流图的符号

如图 2.9 (a) 所示，数据流图有四种基本符号：正方形（或立方体）表示数据的源点或终点；圆角矩形（或圆形）代表变换数据的处理；开口矩形（或两条平行横线）代表数据存储；箭头表示数据流，即特定数据的流动方向。注意，数据流与程序流程图（参见第 3 章）中用箭头表示的控制流有本质不同，千万不要混淆。熟悉程序流程图的初学者在画数据流图时，往往试图在数据流图中表现分支条件或循环，殊不知这样做将造成混乱，画不出正确的数据流图。在数据流图中应该描绘所有可能的数据流向，而不应该描绘出现某个数据流的条件。

处理并不一定是一个程序。一个处理框可以代表一系列程序、单个程序或者程序的一个模块；它甚至可以代表用穿孔机穿孔或目视检查数据正确性等人工处理过程。一个数据存储也并不等同于一个文件，它可以表示一个文件、文件的一部分、数据库的元素或记录的一部分等等；数据可以存储在磁盘、磁带、磁鼓、主存、微缩胶片、穿孔卡片及其他任何介质上（包括人脑）。

数据存储和数据流都是数据，仅仅所处的状态不同。数据存储是处于静止状态的数据，数据流是处于运动中的数据。

通常在数据流图中忽略出错处理，也不包括诸如打开或关闭文件之类的内务处理。数据流图的基本要点是描绘“做什么”而不考虑“怎样做”。

有时数据的源点和终点相同，如果只用一个符号代表数据的源点和终点，则将有两个箭头和这个符号相连（一个进一个出），可能其中一条箭头线相当长，这将降低数据流图的清晰度。另一种表示方法是再重复画一个同样的符号（正方形或立方体）表示数据的终点。有时

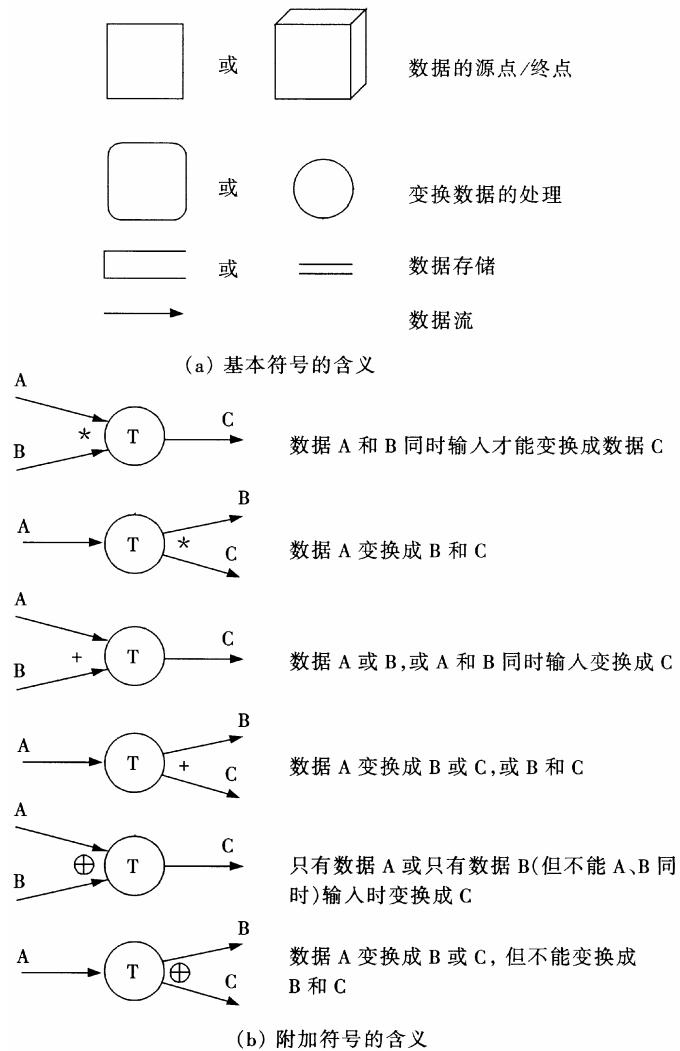


图 2.9 数据流图的符号

数据存储也需要重复，以增加数据流图的清晰程度。为了避免可能引起的误解，如果代表同一个事物的同样符号在图中出现在 n 个地方，则在这个符号的一个角上画 $n - 1$ 条短斜线做标记。

除了上述四种基本符号之外，有时也使用几种附加符号。星号(*)表示数据流之间是“与”关系（同时存在）；加号(+)表示“或”关系；⊕号表示只能从中选一个（互斥的关系）。图2.9(b)给出了这些附加符号的含义。

2.10.2 举例

下面通过一个简单例子具体说明怎样画数据流图。

假设一家工厂的采购部每天需要一张定货报表，报表按零件编号排序，表中列出所有需要再次定货的零件。对于每个需要再次定货的零件应该列出下述数据：零件编号，零件名称，定货数量，目前价格，主要供应商，次要供应商。零件入库或出库称为事务，通过放在仓库中的终端把事务报告给定货系统。当某种零件的库存数量少于库存量临界值时就应该再次定货。

怎样画出上述定货系统的数据流图呢？数据流图有四种成分：源点或终点，处理，数据存储和数据流。因此，第一步可以从问题描述中提取数据流图的四种成分。为此，首先考虑数据的源点和终点，从上面对系统的描述可以知道“采购部每天需要一张定货报表”，“通过放在仓库中的终端把事务报告给定货系统”，所以采购员是数据终点，而仓库管理员是数据源点。接下来考虑处理，再一次阅读问题描述，“采购部需要报表”，显然他们还没有这种报表，因此必须有一个用于产生报表的处理。事务的后果是改变零件库存量，然而任何改变数据的操作都是处理，因此对事务进行的加工是另一个处理。注意，在问题描述中并没有明显地提到需要对事务进行处理，但是通过分析可以看出这种需要。最后，考虑数据流和数据存储：系统把定货报表送给采购部，因此定货报表是一个数据流；事务需要从仓库送到系统中，显然事务是另一个数据流。产生报表和处理事务这两个处理在时间上明显不匹配——每当有一个事务发生时立即处理它，然而每天只产生一次定货报表。因此，用来产生定货报表的数据必须存放一段时间，也就是应该有一个数据存储。

注意，并不是所有数据存储和数据流都能直接从问题描述中提取出来。例如，“当某种零件的库存数量少于库存量临界值时就应该再次定货”，这个事实意味着必须在某个地方有零件库存量和库存量临界值这样的数据。因为这些数据元素的存在时间看来应该比单个事务的存在时间长，所以认为有一个数据存储保存库存清单数据是合理的。

表2.1总结了上面分析的结果，其中加星号标记的是在问题描述中隐含的成分。

表2.1 从描述问题的信息中提取数据流图的元素

源点 / 终点	处 理	数据流 (续)	数据存储 (续)
采购员	产生报表	目前价格 主要供应商 次要供应商	库存量 库存量临界值
仓库管理员	处理事务		
数据流	数据存储		
定货报表	定货信息	事务	
零件编号	(见定货报表)	零件编号*	
零件名称	库存清单*	事务类型	
定货数量	零件编号*	数量*	

一旦把数据流图的四种成分都分离出来以后，就可以着手画数据流图了。但是，怎样开始画呢？注意，数据流图是系统的逻辑模型，然而任何计算机系统实质上都是信息处理系统，也就是说计算机系统本质上都是把输入数据变换成输出数据。因此，任何系统的基本模型都由若干个数据源点／终点以及一个处理来组成，这个处理就代表了系统对数据加工变换的基本功能。对于上述的定货系统可以画出图 2.10 这样的基本系统模型。

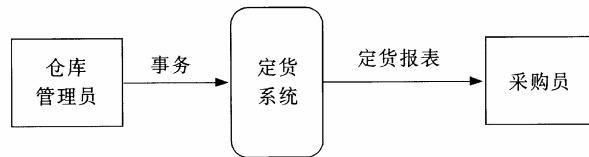


图 2.10 定货系统的基本系统模型

从基本系统模型这样非常高的抽象层次开始画数据流图是一个好方法。基本系统模型突出表明了数据的源点和终点，在这张高层次的数据流图上是否列出了所有给定的数据源点／终点是一目了然的，因此它是很有价值的通信工具，通过它用户容易判断分析员对目标系统的理解是否正确。

然而，图 2.10 毕竟太抽象了，从这张图上对定货系统所能了解到的信息非常有限。下一步应该把基本系统模型细化，描绘系统的主要功能。从表 2.1 可知，“产生报表”和“处理事务”是系统必须完成的两个主要功能，它们将代替图 2.10 中的“定货系统”（见图 2.11）。此外，细化后的数据流图中还增加了两个数据存储：处理事务需要“库存清单”数据；产生报表和处理事务在不同时间，因此需要存储“定货信息”。除了表 2.1 中列出的两个数据流之外还有另外两个数据流，它们与数据存储相同。这是因为从一个数据存储中取出来的或放进去的数据通常和原来存储的数据相同，也就是说，数据存储和数据流只不过是同样数据的两种不同形式。

在图 2.11 中给处理和数据存储都加了编号，这样做的目的是便于引用和追踪。

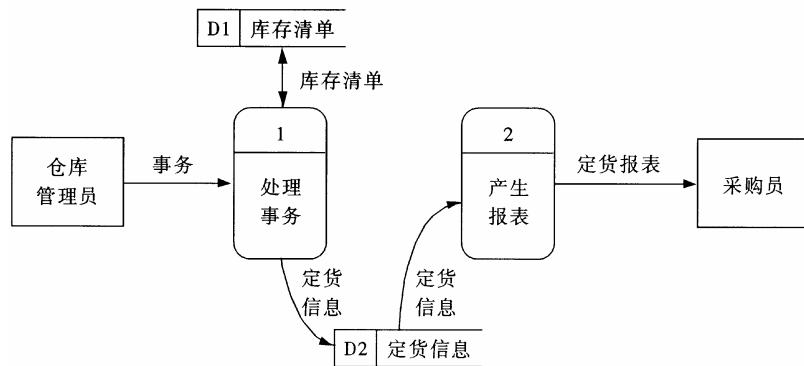


图 2.11 定货系统的功能级数据流图

接下来应该对功能级数据流图中描绘的系统主要功能进一步细化。考虑通过系统的逻辑数据流：当发生一个事务时必须首先接收它；随后按照事务的内容修改库存清单；最后，如果更新后的库存量少于库存量临界值，则应该再次定货，也就是需要处理定货信息。因此，把“处理事务”这个功能分解为“接收事务”、“更新库存清单”和“处理定货”这样三个子

功能，在逻辑上是合理的。图 2.12 描绘了上述的功能分解结果。

为什么不进一步分解“产生报表”这个功能呢？定货报表中需要的数据在存储的定货信

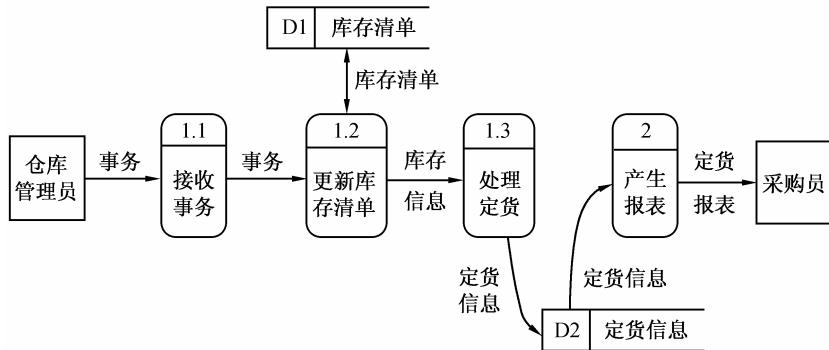


图 2.12 把处理事务的功能进一步分解后的数据流图

息中全都有，产生报表只不过是按一定顺序排列这些信息，再按一定格式打印出来。然而这些考虑纯属具体实现的细节，不应该在数据流图中表现。同样道理，对“接收事务”或“更新库存清单”等功能也没有必要进一步细化。总之，当进一步分解将涉及如何具体地实现一个功能时就不应该再分解了。

当对数据流图分层细化时必须保持信息连续性，也就是说，当把一个处理分解为一系列处理时，分解前和分解后的输入 / 输出数据流必须相同。例如，图 2.10 和图 2.11 的输入 / 输出数据流都是“事务”和“定货报表”；图 2.11 中“处理事务”这个处理框的输入 / 输出数据流是“事务”、“库存清单”和“定货信息”，分解成“接收事务”、“更新库存清单”和“处理定货”三个处理之后（见图 2.12），它们的输入 / 输出数据流仍然是“事务”、“库存清单”和“定货信息”。

此外还应该注意在图 2.12 中对处理进行编号的方法。处理 1.1、1.2 和 1.3 是更高层次的数据流图中处理 1 的组成元素。如果处理 2 被进一步分解，它的组成元素的编号将是 2.1，2.2，…；如果把处理 1.1 进一步分解，则将得到编号为 1.1.1，1.1.2，…的处理。这种编号方法有利于理解和追踪数据流图。

2.10.3 命名

数据流图中每个成分的命名是否恰当，直接影响数据流图的可理解性，因此，给这些成分起名字时应该仔细推敲。下面讲述在命名时应注意的问题。

1. 为数据流（或数据存储）命名
 - ? 名字应代表整个数据流（或数据存储）的内容，而不是仅仅反映它的某些成分。
 - ? 不要使用空洞的、缺乏具体含义的名字（如“数据”、“信息”、“输入”之类）。
 - ? 如果在为某个数据流（或数据存储）起名字时遇到了困难，则很可能是因为对数据流图分解不恰当造成的，应该试试重新分解，看是否能克服这个困难。
2. 为处理命名
 - ? 通常先为数据流命名，然后再为与之相关联的处理命名。这样命名比较容易，而且体现了人类习惯的“由表及里”的思考过程。
 - ? 名字应该反映整个处理的功能，而不是它的一部分功能。

? 名字最好由一个具体的及物动词，加上一个具体的宾语组成。应该尽量避免使用“加工”、“处理”等空洞笼统的动词作名字。

? 通常名字中仅包括一个动词，如果必须用两个动词才能描述整个处理的功能，则把这个处理再分解成两个处理可能更恰当些。

? 如果在为某个处理命名时遇到困难，则很可能是发现了分解不当的迹象，应考虑重新分解。

数据源点 / 终点并不需要在开发目标系统的过程中设计和实现，它并不属于数据流图的核心内容，只不过是目标系统的外围环境部分（可能是人员、计算机外部设备或传感器装置）。通常，为数据源点 / 终点命名时采用它们在问题域中习惯使用的名字（如“采购员”、“仓库管理员”等）。

2.10.4 数据流图的用途

画数据流图的基本目的是利用它作为交流信息的工具。分析员把他对现有系统的认识或对目标系统的设想用数据流图描绘出来，供有关人员审查确认。由于在数据流图中通常仅仅使用四种基本符号，而且不包含任何有关物理实现的细节，因此，绝大多数用户都可以理解和评价它。

从数据流图的基本目标出发，可以考虑在一张数据流图中包含多少个元素为合适的问题。一些调查研究表明，如果一张数据流图中包含的处理多于 5~9 个，人们就难于领会它的含义了。因此数据流图应该分层，并且在把功能级数据流图细化后得到的处理超过 9 个时，应该采用画分图的办法，也就是把每个主要功能都细化为一张数据流分图，而原有的功能级数据流图用来描绘系统的整体逻辑概貌。

数据流图的另一个主要用途是作为分析和设计的工具。分析员在研究现有的系统时常用系统流程图表达他对这个系统的认识，这种描绘方法形象具体，比较容易验证它的正确性；但是，开发工程的目标往往不是完全复制现有的系统，而是创造一个能够完成相同的或类似的功能的新系统。用系统流程图描绘一个系统时，系统的功能和实现每个功能的具体方案是混在一起的。因此，分析员希望以另一种方式进一步总结现有的系统，这种方式应该着重描绘系统所完成的功能而不是系统的物理实现方案。数据流图是实现这个目标的极好手段。

当用数据流图辅助物理系统的设计时，以图中不同处理的定时要求为指南，能够在数据流图上画出许多组自动化边界，每组自动化边界可能意味着一个不同的物理系统，因此可以根据系统的逻辑模型考虑系统的物理实现方案。例如，考虑图 2.12，事务随时可能发生，因此处理 1.1（“接收事务”）必须是联机的；采购员每天需要一次定货报表，因此处理 2（“产生报表”）应该以批量方式进行。问题描述并没有对其他处理施加限制，例如，可以联机地接收事务并放入队列中，然后更新库存清单、处理定货和产生报表以批量方式进行（见图 2.13）。当然，这种方案需要增加一个数据存储以存放事务数据。

改变自动化边界，把处理 1.1、1.2 和 1.3 放在同一个边界内（见图 2.14），这个系统将联机地接收事务、更新库存清单和处理定货及输出定货信息；然而处理 2 将以批量方式产生定货报表。还能设想出建立自动化边界的其他方案吗？如果把处理 1.1 和处理 1.2 放在一个自动化边界内，把处理 1.3 和处理 2 放在另一个边界内，意味着什么样的物理系统呢？

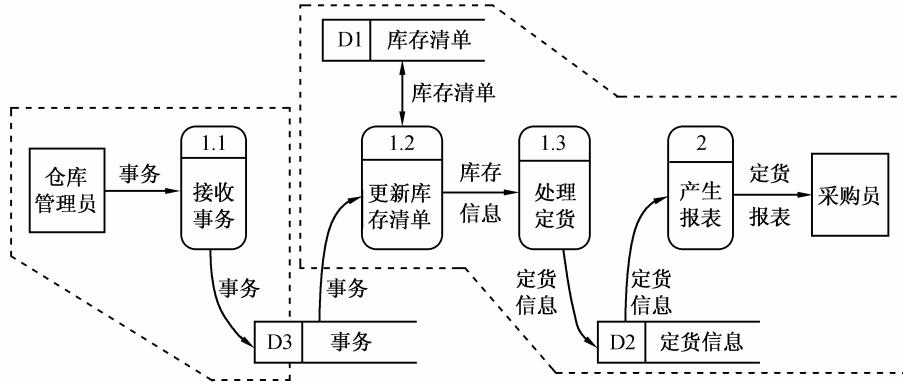


图 2.13 这种划分自动化边界的方法暗示以批量方式更新库存清单

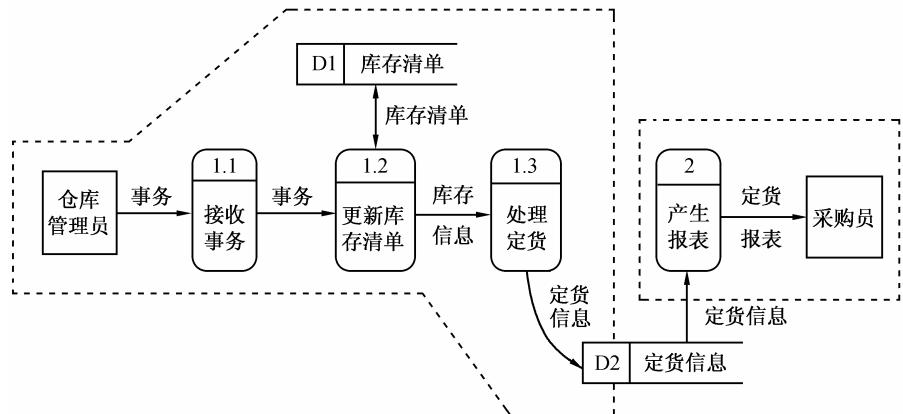


图 2.14 另一种划分自动化边界的方法建议以联机方式更新库存清单

数据流图对更详细的设计步骤也有帮助，本书第 3 章将讲述从数据流图出发映射出软件结构的方法——面向数据流的设计方法。

2.11 数 据 字 典

数据字典是与所开发的系统相关的所有数据的有组织的列表，并且包含了对这些数据的精确、严格的定义，从而能够使得用户和系统分析员双方对目标系统输入、输出、存储的数据以及中间计算结果有共同的理解。简而言之，数据字典是描述数据的信息的集合，是对目标系统中使用的所有数据的定义的集合。

任何字典最主要的用途都是供人查阅对不了解的事物的解释。数据字典的作用也正是在软件分析和设计过程中给人提供关于数据的描述信息。

数据流图和数据字典共同构成系统的逻辑模型。没有数据字典准确地描述数据流图中使

用的数据，数据流图就不严格。反之，没有数据流图，数据字典也难于发挥作用。只有把数据流图和对数据流图中每个数据的精确定义放在一起，才能共同构成系统的规格说明。

2.11.1 数据字典的内容

一般说来，数据字典中应该包含对下列四类元素的定义：

- (1) 数据流；
- (2) 数据流分量（即数据元素）；
- (3) 数据存储；
- (4) 处理。

但是，对数据处理的定义用其他工具（如 IPO 图或 PDL）描述更方便，因此本书中数据字典将主要由对数据的定义组成，这样做可以使数据字典的内容更单纯，形式更统一。

除了数据定义之外，数据字典中还应该包含关于数据的一些其他信息。典型的情况是，在数据字典中记录数据元素的下列信息：一般信息（名字、别名、描述等等），定义（数据类型、长度、结构等等），使用特点（值的范围、使用频率、使用方式——输入／输出／本地、条件值等等），控制信息（来源、用户、使用它的程序、改变权、使用权等等）和分组信息（父结构、从属结构、物理位置——记录、文件和数据库等等）。

其中，有关数据的使用地点与使用方式的信息，可以从数据流图中自动提取得到。表面看来，数据字典中包含这些信息好像并不重要，实际上这是数据字典最主要的优点之一。在系统分析过程中经常要对已完成的工作成果进行修改，对于大型项目来说，确定修改的影响范围往往十分困难，许多软件工程师都遇到过下述问题：“这个数据在什么地方使用？如果修改了它相应地还应该再做哪些修改？”利用数据字典中关于数据的使用地点与使用方式的信息，完全可以回答上述问题。

数据的别名就是该数据的其他等价的名字，出现别名主要有下述三个原因：

- ? 对于同样的数据，不同的用户使用了不同的名字；
- ? 一个分析员在不同时期对同一个数据使用了不同的名字；
- ? 两个分析员分别分析同一个数据流时，使用了不同的名字。

虽然应该尽量减少出现别名，但是不可能完全消除别名。

2.11.2 定义数据的方法

定义绝大多数复杂事物的方法，都是用被定义事物成分的某种组合表示这个事物，这些组成成分又由更低层的成分的组合来定义。从这个意义上说，定义就是自顶向下的分解，所以数据字典中的定义就是对数据自顶向下的分解。那么，应该把数据分解到什么程度呢？一般说来，当分解到不需要进一步定义，每个和工程有关的人也都清楚其含义的元素时，这种分解过程就完成了。

由数据元素组成数据的方式只有下述三种基本类型：

- ? 顺序 即以确定次序连接两个或多个分量。
- ? 选择 即从两个或多个可能的元素中选取一个。
- ? 重复 即把指定的分量重复零次或多次。

因此，可以使用上述三种关系算符定义数据字典中的任何条目。为了说明重复次数，重复算

符通常和重复次数的上下限同时使用（当上下限相同时表示重复次数固定）。当重复的上下限分别为 1 和 0 时，可以用重复算符表示某个分量是可选的（可有可无的）。但是，“可选”是由数据元素组成数据时一种常见的方法，把它单独列为一种算符可以使数据字典更清晰一些。因此，增加了下述的第四种关系算符：

? 可选 即一个分量是可有可无的（重复零次或一次）。

虽然可以使用自然语言描述由数据元素组成数据的关系，但是为了更加清晰简洁起见，建议采用下列符号：

=意思是等价于（或定义为）；

+意思是和（即，连接两个分量）；

[] 意思是或（即，从方括弧内列出的若干个分量中选择一个），通常用“ | ”号分开供选择的分量；

{ } 意思是重复（即，重复花括弧内的分量）；

() 意思是可选（即，圆括弧里的分量可有可无）。

常常使用上限和下限进一步注释表示重复的花括弧。一种注释方法是在开括弧的左边用上角标和下角标分别表明重复的上限和下限；另一种注释方法是在开括弧左侧标明重复的下限，在闭括弧的右侧标明重复的上限。例如

$\overset{5}{\underset{1}{\{A\}}}$ 和 $1\{A\}5$ 含义相同。

下面举例说明上述描述数据内容的符号的使用方法：某种程序设计语言规定，用户说明的标识符是长度不超过 8 个字符的字符串，第一个字符必须是字母字符，随后的字符既可以是字母字符也可以是数字字符。利用上面讲述的符号，可以像下面那样定义标识符：

标识符 = 字母字符 + 字母数字串

字母数字串 = 0{字母或数字}7

字母或数字 = [字母字符 | 数字字符]

由于和项目有关的人都知道字母字符和数字字符的含义，因此，关于标识符的定义分解到这种程度就可以结束了。

2.11.3 数据字典的用途

数据字典最重要的用途是作为分析阶段的工具。在数据字典中建立的一组严密一致的定义很有助于改进分析员和用户之间的通信，因此将消除许多可能的误解。对数据的这一系列严密一致的定义也有助于改进在不同的开发人员或不同的开发小组之间的通信。如果要求所有开发人员都根据公共的数据字典描述数据和设计模块，则能避免许多麻烦的接口问题。

数据字典中包含的每个数据元素的控制信息是很有价值的。因为列出了使用一个给定的数据元素的所有程序（或模块），所以很容易估计改变一个数据将产生的影响，并且能对所有受影响的程序或模块作出相应的改变。

最后，数据字典是开发数据库的第一步，而且是很有价值的一步。

2.11.4 实现数据字典的途径

目前实现数据字典有三种常见的途径：全人工过程，全自动化过程（利用数据字典处理

程序)和混合过程(用正文编辑程序,报告生成程序等已有的实用程序帮助人工过程)。不论使用哪种途径实现的数据字典都应该具有下述特点。

- (1) 通过名字能方便地查阅数据的定义;
- (2) 没有冗余;
- (3) 尽量不重复在规格说明的其他组成部分中已经出现的信息;
- (4) 容易更新和修改;
- (5) 能单独处理描述每个数据元素的信息;
- (6) 定义的书写方法简单方便而且严格。

此外,如果再带有产生交叉参照表、错误检测、一致性校验等功能则更好。

如果暂时还没有自动的数据字典处理程序,建议采用卡片形式书写数据字典,每张卡片上保存描述一个数据元素的信息。这种做法较好地实现了上述要求,特别是更新和修改起来很方便,能够单独处理每个数据元素的信息。每张卡片上主要应该包含下述这样一些信息:

名字、别名、描述、定义、位置。

当开发过程进展到能够知道数据元素的控制信息和使用特点时,把这些信息记录在卡片的背面。

下面给出 2.10 节的例子中几个数据元素的数据字典卡片,以具体说明数据字典卡片中上述几项内容的含义:

名字:定货报表
别名:定货信息
描述:每天一次送给采购员的需要定货的零件表
定义:定货报表=零件编号+零件名称+定货数量+
 目前价格+主要供应者+次要供
 应者
位置:输出到打印机

名字:零件编号
别名:
描述:惟一地标识库存清单中一个特定零件的关键域
定义:零件编号=8{字符}8
位置:定货报表
 定货信息
 库存清单

名字:定货数量
别名:
描述:某个零件一次定货的数量
定义:定货数量=1{数字}5
位置:定货报表
 定货信息

2.12 其他图形工具

在结构化分析过程的不同阶段,或者在同一分析阶段完成性质不同的任务时,分析员往

往使用不同的图形工具辅助完成分析工作。本章前面几节已经分别介绍了系统流程图、实体—联系图、数据流图等图形工具。确实，在描述复杂事物时，图形描绘比文字叙述更优越，它形象直观一目了然。本节再补充介绍在需求分析阶段可能用到的三种图形工具。

2.12.1 层次方框图

层次方框图用树形结构的一系列多层次的矩形框描绘数据的层次结构。树形结构的顶层是一个单独的矩形框，它代表完整的数据结构，下面的各层矩形框代表这个数据的子集，最底层的各个框代表组成这个数据的实际数据元素（不能再分割的元素）。

例如，描绘一家计算机公司全部产品的数据结构可以用图 2.15 中的层次方框图表示。这家公司的产品由硬件、软件和服务三类产品组成，软件产品又分为系统软件和应用软件，系统软件又进一步分为操作系统、编译程序和软件工具，……

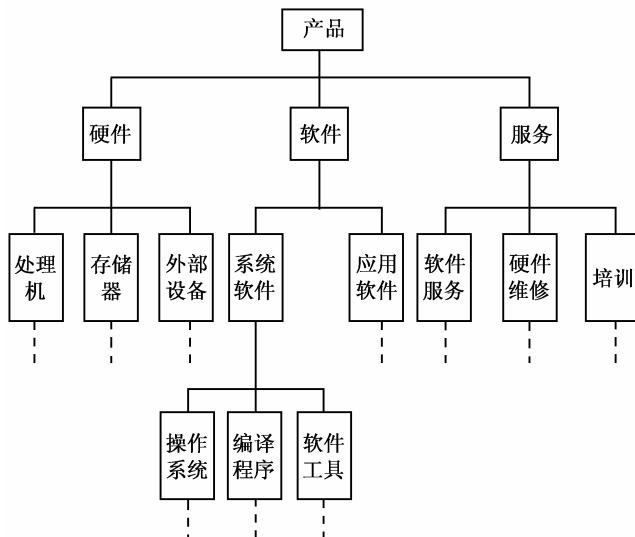


图 2.15 层次方框图的一个例子

随着结构的精细化，层次方框图对数据结构也描绘得越来越详细，这种模式非常适合于需求分析阶段的需要。系统分析员从对顶层信息的分类开始，沿图中每条路径反复细化，直到确定了数据结构的全部细节时为止。

2.12.2 Warnier 图

法国计算机科学家 Warnier 提出了表示信息层次结构的另外一种图形工具。和层次方框图类似，Warnier 图也用树形结构描绘信息，但是这种图形工具比层次方框图提供了更丰富的描绘手段。

用 Warnier 图可以表明信息的逻辑组织，也就是说，它可以指出一类信息或一个信息量是重复出现的，也可以表示特定信息在某一类信息中是有条件地出现的。因为重复和条件约束是说明软件处理过程的基础，所以很容易把 Warnier 图转变成软件设计的工具。

图 2.16 是用 Warnier 图描绘一类软件产品的例子，它说明了这种图形工具的用法。图中花

括号用来区分数据结构的层次，在一个花括号内的所有名字都属于同一类信息；异或符号(\oplus)表明一类信息或一个数据元素在一定条件下才出现，而且在这个符号上、下方的两个名字所代表的数据只能出现一个；在一个名字下面(或右边)的圆括号中的数字指明了这个名字代表的信息类(或元素)在这个数据结构中重复出现的次数。

根据上述符号约定，图 2.16 中的 Warnier 图表示一种软件产品要么是系统软件要么是应用软件。系统软件中有 P1 种操作系统，P2 种编译程序，此外还有软件工具。软件工具是系统软件的一种，它又可以进一步细分为编辑程序、测试驱动程序和设计辅助工具，图中标出了每种软件工具的数量。

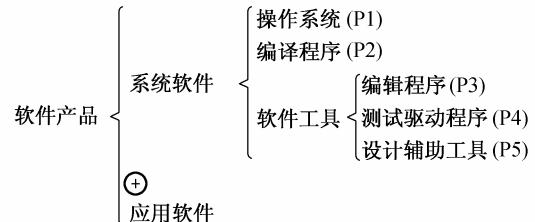


图 2.16 Warnier 图的一个例子

2.12.3 IPO 图

IPO 图是输入 / 处理 / 输出图的简称，它是美国 IBM 公司发展完善起来的一种图形工具，能够方便地描绘输入数据、对数据的处理和输出数据之间的关系。

IPO 图使用的基本符号既少又简单，因此很容易学会使用这种图形工具。它的基本形式是在左边的框中列出有关的输入数据，在中间的框内列出主要的处理，在右边的框内列出产生的输出数据。处理框中列出处理的次序暗示了执行的顺序，但是用这些基本符号还不足以精确描述执行处理的详细情况。在 IPO 图中还用类似向量符号的粗大箭头清楚地指出数据通信的情况。图 2.17 是一个主文件更新的例子，通过这个例子不难了解 IPO 图的用法。

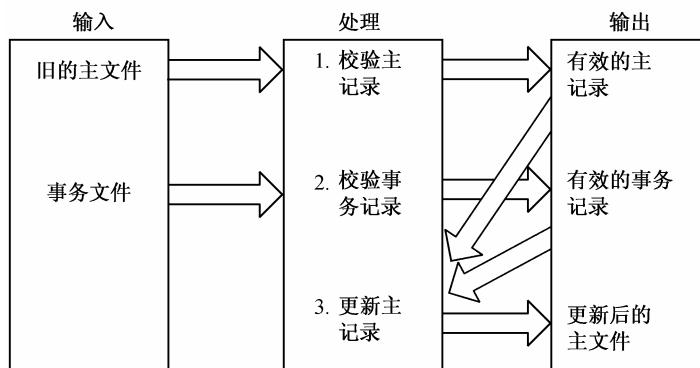


图 2.17 用 IPO 图描绘主文件更新功能

本书建议使用一种改进的 IPO 图(也称为 IPO 表)，这种图中包含某些附加信息，在软件设计过程中将比原始的 IPO 图更有用。如图 2.18 所示，改进的 IPO 图中包含的附加信息，主要有系统名称，图的作者，完成本图的日期，本图描述的模块的名字，模块在层次图中的编号，调用本模块的模块清单，本模块调用的模块的清单，注释，以及本模块使用的局部数据元素等。在需求分析阶段可以使用 IPO 图简略地描述数据流图中各个处理的基本算法(着重说明处理功能而不是具体实现功能的算法)。当然，在需求分析阶段，IPO 表中的许多附加信

息暂时还不具备。但是，在软件设计阶段可以进一步补充、修正这些表，继续作为设计阶段的文档。这正是在需求分析阶段用 IPO 表作为描述基本算法的工具的重要优点。

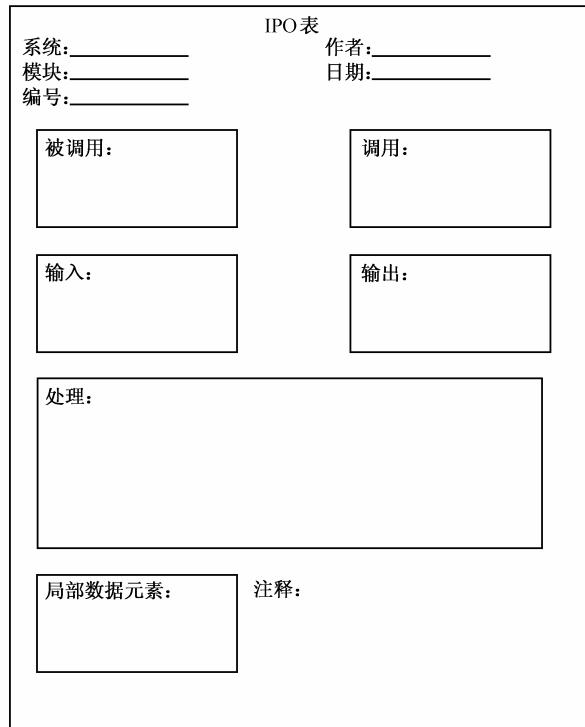


图 2.18 改进的 IPO 图的形式

2.13 成本 / 效益分析

一般说来，人们投资于一项事业的目的是为了在将来得到更大好处。开发一个系统也是一种投资，期望将来获得更大的经济效益。经济效益通常表现为减少运行费用或（和）增加收入。但是，投资开发新系统往往要冒一定风险，系统的开发成本可能比预计的高，效益可能比预期的低。把钱存到银行或贷给其他企业也有明显的经济效益（利息），而且风险小得多。那么，在什么情况下投资开发新系统更划算呢？成本 / 效益分析的目的正是要从经济角度分析开发一个特定的新系统是否划算，从而帮助使用部门负责人正确地作出是否投资于这项开发工程的决定。

为了对比成本和效益，首先需要估计它们的数量。

2.13.1 成本估计

软件开发成本主要表现为人力消耗（乘以平均工资则得到开发费用）。成本估计不是精确的科学，因此应该使用几种不同的估计技术以便相互校验。下面简单介绍两种估算技术。

1. 代码行技术

代码行技术是比较简单的定量估算方法，它把开发每个软件功能的成本和实现这个功能需要用的源代码行数联系起来。通常根据经验和历史数据估计实现一个功能需要的源程序行数。当有以往开发类似软件的历史数据可供参考时，这个方法是非常有效的。

一旦估计出源代码行数以后，用每行代码的平均成本乘以行数就可以确定软件的成本。每行代码的平均成本主要取决于软件的复杂程度和工资水平。

表 2.2 是用代码行技术分析一个过程控制系统的例子。该系统有五个主要功能，根据经验估计了实现每个功能需要的代码行数（表中第 3 列）；历史经验还提供了生产率数据（表中第 2 列），生产率用每人每月生产的代码行数表示，注意，生产一行代码所需完成的工作不仅仅是编码，还需要完成系统分析、设计和测试等工作；根据工资率可以知道每行代码的成本（表中第 4 列）；使用上述数据算出了软件成本（表中第 5 列）和开发这个系统需要使用的人力（表中第 6 列）。

表 2.2 应用代码行技术的一个例子

	生产率(行 / 人月)	估计行数	每行成本(元 / 行)	成本(元)	人力(人月)
获取实时数据	92	840	108	90720	9.1
更新数据库	102	1210	54	65340	11.8
脱机分析	134	600	72	43200	4.4
产生报告	145	450	33	14850	3.1
实时控制	80	1100	135	148500	13.7
				362610	42.1

2. 任务分解技术

这种方法首先把软件开发工程分解为若干个相对独立的任务。再分别估计每个单独的开发任务的成本，最后累加起来得出软件开发工程的总成本。估计每个任务的成本时，通常先估计完成该项任务需要用的人力（以人月为单位），再乘以每人每月的平均工资而得出每个任务的成本。

最常用的办法是按开发阶段划分任务。如果软件系统很复杂，由若干个子系统组成，则可以把每个子系统再按开发阶段进一步划分成更小的任务。

典型环境下各个开发阶段需要使用的人力的百分比大致如表 2.3 所示。当然，应该针对每个开发工程的具体特点，并且参照以往的经验尽可能准确地估计每个阶段实际需要使用的人力（包括书写文档需要的人力）。

表 2.3 典型环境下各个开发阶段需要使用的人力的百分比

任 务	人 力 (%)	任 务	人 力 (%)
可行性研究	5	编码和单元测试	20
需求分析	10	综合测试	40
设计	25	总计	100

对于刚才提到的过程控制系统的例子，使用任务分解技术估计该系统的开发成本，得到的结果列在表 2.4 中。与表 2.2 相对比可以看出，用不同估计技术得到的结果有些差异。

表 2.4 用任务分解技术估计软件开发成本的例子

任务	估计人力(人月)	元 / 人月	成本(元)
需求分析	5.0	10200	51000
设计	15.0	9600	144000
编码和单元测试	8.0	7950	63600
综合测试	16.5	8700	143550
总计	44.5		402150

2.13.2 成本 / 效益分析方法

成本 / 效益分析的第一步是估计开发成本、运行费用和新系统将带来的经济效益，上一节已经简单介绍了估计开发成本的基本方法，以后还要介绍更多成本估计技术。运行费用取决于系统的操作费用（操作人员数，工资水平，工作时间，消耗的物资等）和维护费用。而系统的经济效益等于因使用新系统而增加的收入加上使用新系统可以节省的运行费用。因为运行费用和经济效益两者在软件的整个生命周期内都存在，总的效益和生命周期的长度有关，所以应该合理地估计软件的寿命。虽然许多系统在开发时预期生命周期长达十年以上，但是时间越长系统被废弃的可能性也越大，为了保险起见，以后在进行成本 / 效益分析时一律假设生命周期为 5 年。

应该比较新系统的开发成本和经济效益，以便从经济角度判断这个系统是否值得投资，但是，投资是现在进行的，效益是将来获得的，不能简单地比较成本和效益，应该考虑货币的时间价值。

1. 货币的时间价值

通常用利率的形式表示货币的时间价值。假设年利率为 i ，如果现在存入 P 元，则 n 年后可以得到的钱数为：

$$F=P(1+i)^n$$

这也就是 P 元钱在 n 年后的价值。反之，如果 n 年后能收入 F 元钱，那么这些钱的现在价值是

$$P=F/(1+i)^n$$

例如，修改一个已有的库存清单系统，使它能在每天送给采购员一份定货报表。修改已有的库存清单程序并且编写产生报表的程序，估计共需 5000 元；系统修改后能及时定货将消除零件短缺问题，估计因此每年可以节省 2500 元，五年共可节省 12500 元。但是，不能简单地把 5000 元和 12500 元相比较，因为前者是现在投资的钱，后者是若干年以后节省的钱。

假定年利率为 12%，利用上面计算货币现在价值的公式可以算出修改库存清单系统后每年预计节省的钱的现在价值，如表 2.5 所示。

软件工程

表 2.5

将来的收入折算成现在值

年	将来值(元)	$(1 + i)^n$	现在值(元)	累计的现在值(元)
1	2500	1.12	2232.14	2232.14
2	2500	1.25	1992.98	4225.12
3	2500	1.40	1779.45	6004.57
4	2500	1.57	1588.80	7593.37
5	2500	1.76	1418.57	9011.94

2. 投资回收期

通常用投资回收期衡量一项开发工程的价值。所谓投资回收期就是使累计的经济效益等于最初投资所需要的时间。显然，投资回收期越短就能越快获得利润，因此这项工程也就越值得投资。

例如，修改库存清单系统两年以后可以节省 4225.12 元，比最初的投资（5000 元）还少 774.88 元，第三年以后将再节省 1779.45 元。 $774.88 / 1779.45 = 0.44$ ，因此，投资回收期是 2.44 年。

投资回收期仅仅是一项经济指标，为了衡量一项开发工程的价值，还应该考虑其他经济指标。

3. 纯收入

衡量工程价值的另一项经济指标是工程的纯收入，也就是在整个生命周期之内系统的累计经济效益（折合成现在值）与投资之差。这相当于比较投资开发一个软件系统和把钱存在银行中（或贷给其他企业）这两种方案的优劣。如果纯收入为零，则工程的预期效益和在银行存款一样，但是开发一个系统要冒风险，因此从经济观点看这项工程可能是不值得投资的。如果纯收入小于零，那么这项工程显然不值得投资。

例如，上述修改库存清单系统，工程的纯收入预计是

$$9011.94 - 5000 = 4011.94 \text{ (元)}$$

2.14 结构化分析实例

本章前面各节分别讲述了结构化分析的任务、过程和技术，本节通过一个实例进一步讲述在开发一个软件的过程中怎样应用结构化分析技术完成系统分析工作。

2.14.1 工资支付问题定义

王晓明是某高等职业技术学院计算机应用专业的学生，毕业后在一所职业高中工作，分配给他的工作是建立并管理学校的微机机房。经过一段时间的辛勤工作，微机机房建立起来了，并且开始接待学生上机实习。

一天，学校的财务科长把小王找去，请他研究用学校自己的微型计算机生成工资明细表和各种财务报表的可能性。

1. 定义问题的过程

小王应该从何处着手解决财务科长提出的问题呢？立即开始考虑实现工资支付系统的详细方案并动手编写程序，无疑是很有诱惑力的。技术问题对技术人员最有吸引力。为了实现工资支付系统需要购买新的硬件吗？用什么语言写程序更方便呢？在这个软件中应该使用文件系统还是数据库来存储数据呢？……

显然，需要考虑的具体技术问题很多，但是在这样的早期阶段就考虑这么具体的技术问题，却很可能使我们迷失前进的方向。会计部门（用户）并没有要求小王在学校自己的计算机上实现工资支付系统，仅仅要求他研究这种可能性。后者是一个非常重要的然而和前者又很不相同的问题，它实际上是问，在自己的计算机上实现工资支付系统，预期将获得的经济效益能超过开发这个系统的成本吗？换句话说，这项工作值得做吗？

优秀的系统分析员还应该进一步考虑，我们所面临的问题究竟是什么。财务科长为什么要求他研究在自己的计算机上实现工资支付系统的可能性呢？通过询问财务科长小王了解到，该校原来一直由会计人员人工计算工资并编制财务报表，随着学校规模的扩大工作量也越来越大，目前每个月都需要两名会计紧张工作半个月才能完成，这样做不仅效率低而且成本高，以后学校规模将进一步扩大，人工计算工资的成本还会进一步提高。

因此，我们的目标是寻找一种比较便宜的生成工资明细表和相应的财务报表的办法，并不一定必须在我们自己的计算机上实现工资支付系统。当财务科长要求小王“研究用学校自己的微型计算机生成工资明细表和各种财务报表的可能性”时，实际上并不是描述应该解决的问题，而是在建议一种解决问题的方案。这种解决方案可能是一个好办法，分析员当然应该认真考虑它，但是也还应该考虑其他可能的解决办法，以便从中选出最好的办法。好的问题定义应该明确地描述实际问题，而不是隐含地描述解决问题的办法。

小王应该考虑的另一个关键问题是预期的项目规模。改进的工资支付系统有价值吗？虽然没人明确提出来，但是肯定会有某个隐含的限度。目前用人工计算工资并编制财务报表，如果新系统比人工系统成本更高，显然不值得开发。应该考虑下述的三个基本数字：目前计算工资所需花费的成本，新系统的运行费用和新系统的开发成本。新系统的运行费用必须低于目前的成本，而且节省的费用应该能使学校在一个合理的期限内收回开发新系统时的投资。

虽然知道了目前用人工计算工资所需的成本，但是在这样的早期阶段，小王对新系统的运行费用和开发成本却只能猜测。但是，规定未来系统的规模却仍然是可能的。目前，每个月需要由两名会计花费半个月的时间来计算工资和编制报表，一名会计每个月的工资和岗位津贴共约 2000 元，因此，每年为此项工作花费的人工费约 2.4 万元。显然，任何新系统的运行费用也不可能减少到小于零，因此，新系统每年最多可能获得的经济效益是 2.4 万元。

为了每年节省 2.4 万元，投资多少钱是可以接受的呢？绝大多数单位都希望在三年内收回投资，因此，对于这个项目来说，7.2 万元开发成本可能是一个合理的上限值。虽然这是一个很粗略的数字，但是它确实能使得用户对项目规模有一些了解。如果这项工作不能在 7.2 万元之内完成，那么它可能是不值得做的。

2. 关于系统规模和目标的报告书

现在小王对需要解决的问题和新系统的规模都有了一些认识，是否会计们心中也是这样

想的呢？小王会不会误解了问题的某些方面呢？肯定会有误解的地方！对错误问题的解答即使再圆满又有什么价值呢？！如果小王对问题的认识与会计或校长的认识不一致，那么他无论怎样努力也开发不出能解决实际问题的系统。一个系统，甚至一个“好”系统，如果不能解决实际问题，那么它就是一文不值的，只不过是白白浪费开发资源和经费。因此，在系统生命周期的这个早期阶段，小王清晰地表达出他对问题的认识并请用户和领导审查、纠正他的认识，是极其重要的。典型地，用一个简单的书面备忘录表达分析员对问题的认识，这份文档称为“关于系统规模和目标的报告书”（见表 2.6）。

表 2.6

关于工资支付系统规模和目标的报告书

关于系统规模和目标的报告书		2002.12.26
项目名称：工资支付。		
问题：目前计算工资和编制报表的费用太高。		
项目目标：研究开发费用较低的新工资支付系统的可能性。		
项目规模：开发成本应该不超过 7.2 万元（±50%）。		
初步设想：用学校自己的计算机系统生成工资明细表和财务报表。		
可行性研究：为了更全面地研究工资支付项目的可能性，建议进行大约历时两周的可行性研究。这个研究的成本不超过 4000 元。		

关于系统规模和目标的报告书并没有标准格式，它的具体格式往往随项目而异。书写这份文档的基本原则是，分析员应该尽可能简明清晰地叙述他对问题的理解。应该为项目取一个恰当的名字，应该明确地定义问题，并且应该清楚地说明整个工程项目和可行性研究的规模。通常一份简短的备忘录就足够了，关键是交换想法而不是遵守某种固定格式。

对项目规模的估计目前还是很粗略的，因此在报告书中注明有±50%的误差。然而对系统生命周期下一个阶段（可行性研究）的成本估计却应该相当准确。实际上，小王要求校长先向“工资支付”项目投资 4000 元，以便更准确地估计成本和效益。在可行性研究结束时，将能够断定是否效益大到值得投资的程度。

假设校长和财务科同意小王提出的关于系统规模和目标的报告书，并且支付了供两个星期可行性研究的经费。现在小王可以对“工资支付”项目进行更仔细的研究了。

2.14.2 可行性研究

系统分析员面临的问题通常并没有简单明显的解决办法。事实上，许多问题不能在已确定的系统规模内解决。可行性研究的目的正是要花尽可能少的成本来确定问题是否存在可行的解法。如果没有可行的解法就盲目地着手开发该系统，显然花在开发项目上的时间、人力和金钱都是浪费。

可行性研究是抽象和简化了的系统分析和设计的全过程，它的目标是用最小的代价尽快确定问题是否能够解决。可以把可行性研究看作是一种保险策略，任何投资都要冒一定的风险，因此，很有必要在大量投入经费和人力之前，先研究投资获得成功的可能性。

应该记住，可行性研究的目的不是解决问题，而是确定是否值得去解决这个问题。在进行可行性研究时很容易陷到具体的技术问题里面，分析员必须时刻记住可行性研究的目的，记住能够用于这个阶段的时间和经费都是很有限的（在关于系统规模和目标的报告书中明确规定了这个限度）。

1. 澄清系统规模和目标

在问题定义阶段确定的系统规模和目标准确到什么程度呢？校长和财务科实际需要什么样的系统呢？分析员在进行可行性研究时首先应该进一步澄清问题定义，为此需要进行一系列调查访问。

工资支付系统的用户是会计部门（财务科），因此小王首先访问财务科长。第一个问题是个很灵活的问题：“当你建议我调查研究工资支付问题时，你心里是怎么想的呢？”财务科长的回答使小王具体感受到实际存在的问题：人工计算工资和编制财务报表的费用随着学校规模扩大而不断增加；人工计算速度慢，随着教职工人数增加发工资的日期也一再延后，引起教职工不满；人工计算容易出差错；……其他问题涉及到分析解决方案的可行性时也是必须考虑的限制，例如，教职工代表大会的决议，劳动法等有关法律的规定。

没有哪个系统是在“真空”中运行的，因此，小王问财务科长是否有其他系统使用工资支付数据，科长告诉他工资总数应该记入分类日记账，显然，新工资支付系统不能忽略分类账系统。最后小王问，谁具体处理工资支付事务。财务科长告诉他两位会计的名字，这两个人是小王以后访问的对象。

接下来小王拜访学校的校长。如果开发新工资系统则需要学校投资，那么，校长喜欢什么样的系统呢？粗略估计项目规模是7.2万元，校长对这个数字是怎样想的呢？认为这是一个粗略的估计数字呢，还是认为这是一个绝对不可以超过的限度？新系统应该比现有的人工系统强多少他才愿意投资呢？在做进一步的开发工作之前，分析员清楚地了解系统的实际规模和目标是很重要的。

最后，小王应该做一些必要的准备工作。需要他研究解决的是工资支付问题，工资支付有它自己的专门术语和专门知识。分析员在分析一个问题之前，必须学习了解一些这个应用领域的基本知识。学习的目的不是要变成一个工资支付问题专家，只是要获得对这个应用领域的基本了解。

2. 研究现有的系统

了解任何应用领域最快速有效的方法，可能都是研究现有的系统。但是，分析员应该记住，研究现有系统的目的是了解它，用它作为开发新系统的借鉴，千万不要被实现现有系统的技术细节迷住而花费过多时间和精力。

首先应该访问关键人员。怎样知道谁是关键人员呢？以前访问财务科长时曾了解到两名具体处理工资事务的会计的姓名，因此可以从访问这两个人入手。经过询问，小王知道了处理工资事务的大致过程。在现阶段小王把学校的工资支付系统还看作一个黑盒子，他用图2.19描绘了处理工资支付事务的大致过程。

图2.19描绘处理工资事务的大致过程是每月月末教师把他们当月授课时数登记在课时表上，由各系汇总后交给财务科，职工把他们当月完成承包任务的情况登记在任务表上，由各科室汇总后交给财务科。两名负责工资事务的会计，根据这些原始数据计算每名教职工的工资，编制工资表、工资明细表和财务报表，然后把记有全校教职工每人工资总额的工资表报送银行，由银行把钱打到每名教职工的工资存折上，同时把每名教职工的工资明细表分发给教职工。小王的目标是了解图2.19中这个黑盒子（工资支付系统）的内容。怎样达到这个目标呢？通常，从黑盒子的边缘开始了解，由表及里逐步深入。谁接收课时表和任务表？谁分发工资明细表？对这些问题的回答能使小王知道一些处在黑盒子内部的人员，显然这些人比

小王对工资支付系统了解得更多，通过他们可以了解到更多情况。上述做法虽然简单但却比较有效，概括地说，就是从你已经知道的事物开始，访问处在你所知道的事物边缘的人，通过他们了解边缘功能，并请他们建议下一步应了解的事物。这样逐步做下去，将使得你对现有系统了解得越来越多。

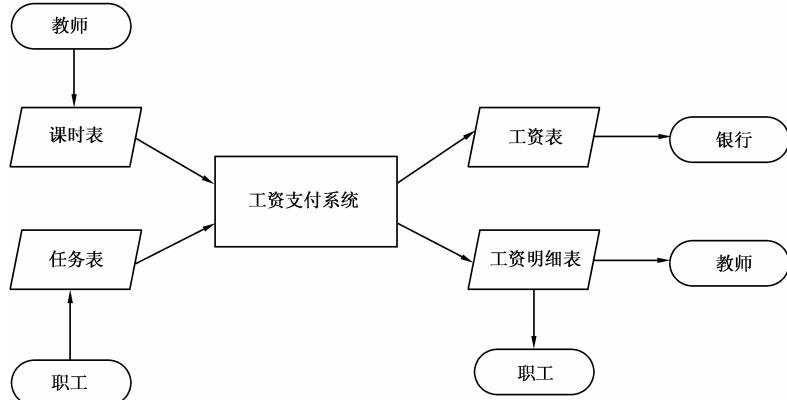


图 2.19 处理工资事务的大致过程

通过向财务人员多次询问，小王了解到现有的人工系统计算工资和编制报表的流程如下：接到课时表和任务表之后，首先审核这些数据，然后把审核后的数据按教职工编号排序并抄到专用的表格上，该表格预先印有教职工编号、姓名、职务、职称、基本工资、生活补贴、书报费、交通费、洗理费等数据。接下来根据当月课时数或完成承包任务的情况，计算课时费或岗位津贴。得出每个人的工资总额之后，再计算应该扣除的个人所得税，应交纳的住房公积金和保险费，最后得出每个人当月的实发工资数。把计算得出的上述各项数据登记到前述的专用表格上，就得到了工资明细表，然后对数据进行汇总，编制出各种财务报表，其中工资表不过是简化的工资明细表，它只包括工资明细表中教职工编号、姓名和实发工资三项内容。图 2.20 所示的系统流程图描绘了现有的人工工资支付系统的工作流程。

现在，小王已经用一张系统流程图（图 2.20）描绘了现有的工资支付系统。但是，这张图对现有系统的描绘准确吗？小王已经把工资支付系统的所有关键功能都划分出来了吗？小王应该请有关人员仔细审查这张系统流程图，有错误就应该改正，有遗漏就应该补充。

和现有的物理系统相联系的一个问题是，常常很难区分“做什么”和“怎样做”这两类不同范畴的知识。因此，下一步应该导出工资支付系统的高层逻辑模型。

3. 导出高层逻辑模型

系统流程图是描绘物理系统的好方法，然而有时可能因为图中符号表达的含义过分具体反而不符合需要。例如，图 2.20 中梯形框代表人工完成的数据处理功能。但是，我们的目标并不是一成不变地复制现有的人工系统，而是开发一个用计算机完成同样功能的新系统。因此，小王希望用另一种方式总结从现有系统中获得的知识，不是准确地描绘具体的实现方法，而是着重描绘系统的逻辑功能。数据流图是实现这个目标的极好工具。

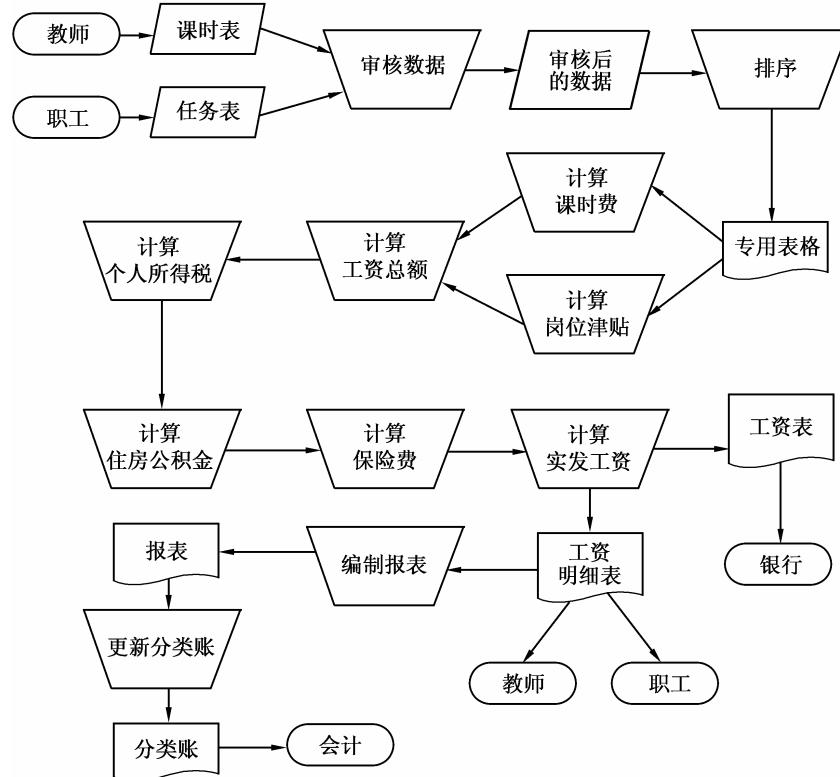


图 2.20 现有的工资支付系统

怎样得出系统的数据流图呢？首先应该找出构成数据流图的四种基本元素（数据流的源点或终点，处理，数据流，数据存储），第二步是把找出的四种基本元素组合成数据流图。

在描绘现有系统的系统流程图（见图 2.20）中，什么是数据流的源点或终点呢？教师提供课时表并接受工资明细表，职工提供任务表并接受工资明细表，因此，教师和职工既是数据流的源点又是终点。此外，会计接受报表，银行接受工资表，因此会计和银行也是数据流的终点。

其次考虑处理。工资支付系统需要完成哪些工作呢？先收集课时表和任务表等数据（把这些数据笼统称为事务数据），然后加工事务数据（包括计算课时费、岗位津贴、工资总额、个人所得税、住房公积金、保险费和实发工资），产生工资表和工资明细表，编制报表，分发工资明细表。最后，作为和这个系统有关的系统的一部分，还应该更新会计分类账。

数据存储有哪些呢？从教职工那里收集并保存起来的事务数据是一个数据存储，此外，工资表、工资明细表和各种报表也是数据存储。为什么我们不再区分排序前的事务数据和排序后的事务数据呢？图 2.20 描绘的把事务数据排序之后再加工处理这些数据的方法，只不过是计算工资的一种方法，但并不是唯一的方法。我们希望数据流图描绘出工资支付系统必须完成的逻辑功能，而不是现有系统中采用的完成这些功能的方法。在现阶段还不需要考虑具体采用什么技术方法，将来完全可以采用其他解决办法。

什么是数据流，它和数据存储有何不同呢？数据流是移动的数据，而数据存储是静止的数据，它们不过是同一事物（数据）的两种不同形式。数据流给数据存储注入数据，而数据存储又往往是数据流的源泉。在更具体的层次上，我们可能需要研究个别的数据元素，考虑几个数据流如何组合成一个数据存储，或者一个数据存储怎样为几个数据流分别提供不同的数据元素。但是，在可行性研究阶段，我们应该始终处在很高的抽象层次上，因此可以把数据流和数据存储看作是同样数据的两种不同形式。

通过上述分析得到的工资支付系统数据流图的四种基本元素列在表 2.7 中。

表 2.7 工资支付系统数据流图元素

源点 / 终点	数据存储	处理	数据流
教师	事务数据	收集数据	(与数据存储相同)
职工	工资表	审核数据	
会计	工资明细表	加工事务数据	
银行	报表	分发工资明细表	
	分类账	更新分类账	

根据表 2.7 中列出的基本元素，可以画出工资支付系统的数据流图（见图 2.21）。从这张图中很容易看出从收集数据到加工事务数据产生工资表、工资明细表和财务报表的过程。

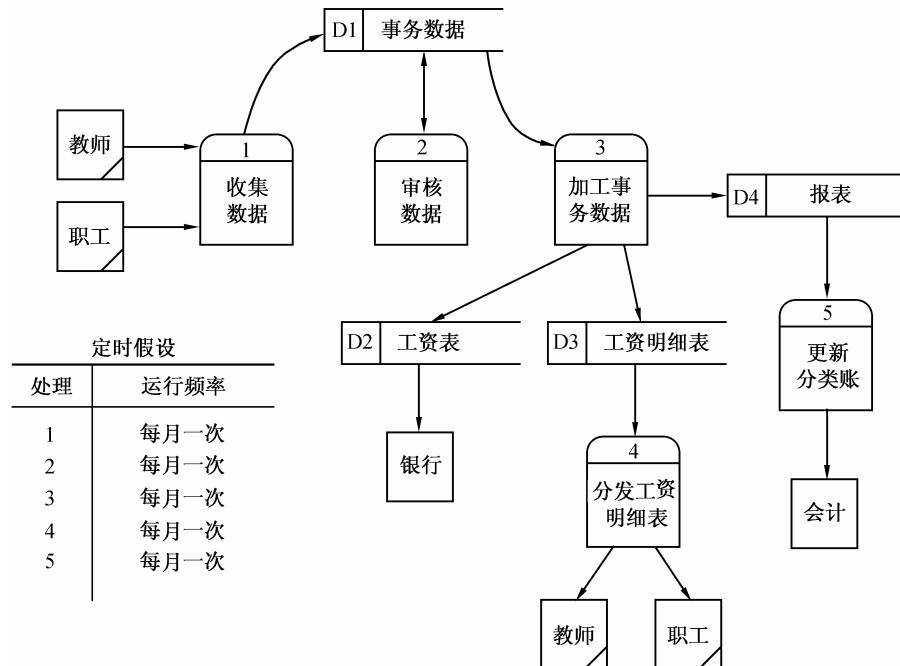


图 2.21 工资支付系统的数据流图

注意，从“加工事务数据”处理框流出的“报表”进入另一个处理框“更新分类账”。对分类账的处理是另一个系统的任务，但是，工资支付系统至少必须和这个系统通信，因此搞清楚它们之间的接口要点是很重要的。

最后，分析员应该在数据流图上直接注明关键的定时假设。在以后的系统设计过程中这些假设将起重要作用。清楚地注明这些假设也可以增加及时发现并纠正误解的机会。

数据流图代表系统的逻辑模型。分析员可以借助数据流图向用户和管理人员讲解他对系统的认识，并请他们补充和纠正他的认识。在设计新系统的过程中，这个逻辑模型也很有用。

4. 进一步确定系统规模和目标

现在小王应该再次访问会计和财务科长，访问时讨论的焦点是数据流图，它代表了小王对系统的认识。这种认识对不对？有没有什么遗漏？所做的假设正确吗？数据流图这样的逻辑模型能够以容易被人理解的形式概括大量信息，通过仔细分析和讨论数据流图，能够及时发现并纠正分析员对系统的误解，补充被他忽视了的内容。

在讨论图 2.21 所示的数据流图时，两名具体负责工资事务的会计提出了疑问：为什么在数据流图中没有计算课时费、岗位津贴、工资总额、个人所得税、住房公积金、保险费、实发工资等处理？小王解释说，这张数据流图描绘的是系统高层逻辑模型，在现阶段还不需要考虑完成“加工事务数据”功能的具体算法，因此没有把它分解成一系列更具体的数据处理功能。通过讨论，大家认为图 2.21 正确地描绘了目标系统的高层逻辑模型。

小王现在对工资支付系统的认识已经比问题定义阶段深入多了，根据现在的认识，最初对系统规模和目标的报告仍然正确吗？如果小王现在认识到不能在 7.2 万元的成本内开发出工资支付系统，则应该及时报告给校长。校长的决定可能是提高成本限额，也可能是放弃这个开发项目，总之需要校长做出明确的决定。当然，如果系统规模和（或）目标改变了，小王必须重复前面做过的工作。

可行性研究的上述 4 个步骤可以看作是一个循环。分析员定义问题，分析这个问题，导出试探性的逻辑模型，在此基础上再次定义问题，……重复这个循环直至得出准确的逻辑模型为止，然后分析员可以开始考虑实现这个系统的方案。

5. 导出供选择的解法

现在小王对用户的问题已经有了比较深入的理解，但是，问题能够解决吗？有行得通的解决办法吗？回答这些问题的唯一方法是，导出一些供选择的解决办法，并且分析这些解法的可行性。

怎样导出供选择的解法呢？一个常用的简单方法是从数据流图（见图 2.21）出发，假设几种划分自动化边界的模式，并且为每种模式设想一个系统。例如，可以把“收集数据”和“审核数据”两个处理放在同一个边界内，从而意味着一个数据收集程序；对事务数据的处理放在另外的边界内，因此代表另一个程序。或者考虑把“审核数据”和“加工事务数据”放在同一个自动化边界内，这可能意味着一个批处理程序，它首先校核数据然后加上数据。总之，每当分析员选取一组不同的自动化边界时，就可能意味着一种不同的解法。

在上述设想供选择的解法的过程中，分析员首先考虑的是技术上的可行性。不能在现有硬件上实现的或与这个应用有定时冲突的方案都不需要考虑。显然，技术上不可能实现的方案是没有意义的。但是，技术可行性只是必须考虑的一个方面，还必须能同时通过其他检验，一种解决方案才可以说是可行的。

另一个必须考虑的关键问题是操作可行性。即使一种解法在技术上是完全可行的，如果它与使用单位的运作方式有冲突，那么它也是行不通的。在对学生开放的计算机机房内运行工资支付程序，打印工资表、工资明细表和各种财务报表，显然是不合适的。这样做不仅不安全而且会暴露教职工个人的隐私（工资收入等个人信息）。因此，必须为工资支付系统单独购置一台微型计算机及必要的外部设备（例如，打印机），并安放在一间专用的房间里。总之，技术可行性问：“我能这样做吗？”操作可行性问：“在这里我能这样做吗？”

最后，必须考虑经济可行性问题，即“效益大于成本吗？”现在已经有一个工资支付系统，如果新系统比现有的系统成本更高，为什么还要花力气去开发它呢？因此，分析员必须对已经通过技术可行性和操作可行性检验的各种解决方案再进行成本／效益分析。

在任何情况下，给用户和使用单位负责人提供在一定范围内进行选择的余地，都是一个好主意。分析员至少应该提出三种类型的解法供他们选择：低成本的系统；能较好地完成任务的中等成本的系统；包含所有可能需要的功能的高成本系统。此外分析员还应该考虑现有的系统，它是一个正在工作着的系统，没有风险也不需要重新投资。当然，它的运行费用看来太高了，但是，如果没有其他问题，这个费用正是评价其他代替它的方案的标准。

有没有不需要投资或只用极少投资就可以减少运行费用的办法呢？回答是肯定的。该职业高中可以把每月发一次工资改为每两个月发一次工资，这样做每年为发工资而花费的费用大约可减少一半，即每年可节省 1.2 万元。除了已经进行的可行性研究之外，不再需要任何新的投资。这是一个极好的低成本方案。

当然也必须考虑到上述低成本方案的缺点：两个月发一次工资违反常规；教职工肯定会反对这种“改革”；这个解决办法实际上并不能解决根本问题，随着学校规模扩大，人工处理工资事务的费用也将成比例地增加。虽然有这些问题，但是这个办法不需要投资而能显著减少处理工资事务的费用，因此，仍然是一个值得考虑的办法。

作为中等成本的解决办法，小王建议基本上复制现有的系统：课时表和任务表不再送到财务科交人工处理，而是交到处理工资事务的专用机房，工资支付系统的操作员把这些数据通过终端送入计算机，数据收集程序接收并校核这些事务数据，把它们存储在磁盘上。在所有事务数据都输入磁盘以后，运行工资支付程序，这个程序从磁盘中读取事务数据，计算工资，印出工资表、工资明细表和财务报表。图 2.22 所示的系统流程图描绘了上述的系统。

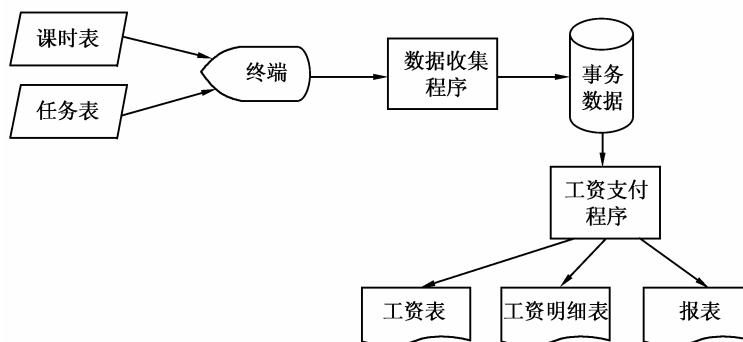


图 2.22 中等成本方案的系统流程图

这个中等成本的解决方案看来是现实的，因此小王完成了完整的成本 / 效益分析，分析结果列在表 2.8 中。当然，对于每个认真考虑过的解决办法，都应该完成类似的分析。据估计，开发中等成本的工资支付系统大约需要一个人用 4 个月时间才能完成，按每人每月的成本 8000 元计算，4 个月共需 3.2 万元，此外，购买硬件约需 1 万元，因此总成本是 4.2 万元。

表 2.8 中等成本方案的成本 / 效益分析

中等成本方案的成本 / 效益分析			
开发成本			
人力 (4 人月, 8000 元 / 人月)			3.2 万元
购买硬件			1.0 万元
总计			4.2 万元
新系统的运行费用			
人力和物资 (250 元 / 月)			0.3 万元 / 年
维护			0.1 万元 / 年
总计			0.4 万元 / 年
现有系统的运行费用			
2.4 万元 / 年			
每年节省的费用			
2.0 万元			
年	节 省	现在值 (以 5% 计算)	累计现在值
1	20000 元	19047.62 元	19047.62 元
2	20000 元	18181.82 元	37229.44 元
3	20000 元	17241.38 元	54470.82 元
投资回收期			2.28 年
纯收入			12470.82 元

为了回收上述投资，新系统必须能够节省开支。小王估计新系统每月的使用费用是 250 元，即每年的使用费用大约是 3000 元，此外，估计每年还需要维护费用 1000 元，因此总计每年的运行费用是 4000 元。现有系统每年的运行费用是 2.4 万元，因此每年可以节省 2.0 万元。表 2.8 总结了从这些基本数字出发进行的一些经济分析的结果。分析结果告诉我们，3 年内（实际是 2.28 年）可以收回全部投资，3 年中纯收入约为 12471 元。看来中等成本的解决方案是很合理的，经济上是可行的。

最后，小王考虑一种成本更高的方案。学校刚刚开始向数据处理自动化这个目标迈进，为什么不在开始时就把事情做得更完美一些呢？应该考虑建立一个中央数据库，为开发一个完整的管理信息系统做好准备，并且把工资支付系统作为该系统的第一个子系统。这样做开发成本可能会增加到 12 万元，然而从工资支付这项应用中能够获得的经济效益并不变（即每年可比现在节省 2 万元）。因此，如果仅考虑这一项应用，投资是不划算的，但是，将来其他应用（例如，教学管理，物资管理，人力资源管理）能以低得多的成本实现，在长期运行中效益将是显著的。如果校长对这个方案感兴趣，小王可以完成更详尽的可行性研究，这样的

可行性研究大约需要用 1 万元。

6. 推荐最佳方案

低成本方案虽然很吸引人，但是实现它需要对学校的现行制度作重大变动，此外还有前述的一些其他缺点；高成本的系统从长远看是合理的，但是它牵涉到学校的未来规划。从已经确定的工资支付系统的规模和目标来看，显然中等成本的方案是最好的，小王将把它作为推荐的最佳方案。

7. 草拟开发计划

小王下一步应该做的工作，是为所推荐的方案草拟一份开发计划。在这样非常早期的开发阶段，很难准确估计分析、设计和实行的工资支付系统需要用的时间、人力和经费，但是根据系统生命周期还是能够作出相对合理的估计的。下面分别考虑生命周期的每个阶段。

在需求分析阶段，分析员的任务是定义系统的功能需求，为了完成这个任务必须学习大量有关工资问题的知识，看来这个阶段需要用一个月的时间。

概要设计阶段的任务，主要是推荐最佳系统实现方案和设计工资支付软件的体系结构，由于这个软件比较简单，估计这个阶段仅需用半个月的时间。

在详细设计阶段，软件工程师将详细地设计组成系统的每个模块，估计需要用一个月的时间。

实现阶段的任务除了编码之外主要是测试（包括平行运行），估计这个阶段需要 2 个月的时间。

已经用时 2 周完成的可行性研究，也是开发过程的一个阶段。表 2.9 总结了上述的工资支付系统实现计划。

这个粗略的实现计划回答了两个重要问题：项目需要多长时间（5 个月）；需要多少人参加（可能只需要 1 个人）。根据成本 / 效益分析的结果和这个粗略的进度计划，校长可以作出是否进行这个项目的合理决定。

8. 写出文档提交审查

现在可行性研究工作实质上已经完成了。小王应该归纳整理有关的信息并写成正式文档（其中成本 / 效益分析的内容，依照表 2.9 所示的实现计划稍加修正），提交由校长和财务科全体人员参加的会议审查。假设他们对小王的工作表示满意，接受了他的建议，于是接下来将进行需求分析阶段的工作。

表 2.9 实现中等成本的工资支付系统的粗略计划

阶 段	需要的时间(月)
可行性研究	0.5
需求分析	1.0
概要设计	0.5
详细设计	1.0
实现	2.0
总计	5.0

2.14.3 需求分析

需求分析的目的是确切地回答下述问题：“系统必须做什么？”虽然经过可行性研究阶段的工作，分析员对于目标系统应该完成的基本功能已经有了一些了解，但是实际上仍然不能确切地回答上述问题。可行性研究的目的，是用较少成本在较短时间内确定是否有可行的解法存在。因此，在这个过程中许多细节被忽略了，然而在最终的系统中是不能遗漏任何一个细节的。在需求分析阶段，分析员将努力获得对系统功能的完整、准确的了解。这个阶段的工

作还不是确定系统将怎样工作，而是确定系统必须做什么。

需求分析在可行性研究的基础上进行，前一阶段产生的文档，特别是数据流图（见图 2.21），是这个阶段工作的出发点。在需求分析阶段系统分析员将设计出更精确的数据流图，此外还将写出数据字典，及一系列简明的算法描述。这些文档是软件需求规格说明书的重要组成部分，而完整、正确的规格说明书是需求分析阶段结束的标准，它总结了分析员对目标系统的理解，经过用户和使用单位领导的审查批准之后，将用来帮助设计实际的物理系统。

需求分析阶段的主要任务是更详尽地定义系统应该完成的每一个逻辑功能。分析员怎样完成这个任务呢？

任何数据处理系统的基本功能，都是把输入数据转变成需要的输出信息。数据决定了处理和算法，看来数据应该是分析工作的出发点。在可行性研究阶段，由于不需要了解太多细节，因此忽略了大部分实际的数据元素，现在是定义这些数据元素的时候了。

必须经过计算才能得到的数据元素引出了必要的算法，算法反过来又引出了更多的数据元素。对数据的描述记录在数据字典中，对算法的描述记录在一组初步的 IPO 表中。需要说明的是，在需求分析阶段描述算法的目的是为了准确地定义处理数据的功能，因此，这个阶段描述的是原理性的算法，而不是实现数据处理功能的具体算法，具体算法将在详细设计阶段设计出来。

根据对系统的更新、更深入的认识，可以进一步细化数据流图。在细化数据流图的过程中，又会进一步加深对系统的认识。这样一步一步地分析，将更详尽地定义出所需要的逻辑系统。

需求分析过程结束之前，必须对分析得出的结果进行严格的技术审查和管理复审。

下面具体讲述工资支付系统的需求分析过程。

1. 沿数据流图回溯

应该仔细研究在可行性研究阶段得到的数据流图（见图 2.21），分析员现在的目标是把数据流和数据存储定义到元素级。从什么地方着手分析呢？随着应用性质不同，可以从图中不同地方开始分析，但是，一般说来从输出端开始是有意义的。这是因为，系统最基本的功能是产生需要的输出，在输出端出现的数据元素决定了系统的基本构成。

从图 2.21 的数据终点“教师”和“职工”开始分析，流入他们的数据流是“工资明细表”。工资明细表由哪些数据元素组成呢？从该职业高中目前使用的工资明细表上可以看出它包含许多数据元素，表 2.10 列出了这些数据元素。这些数据元素是从什么地方来的呢？既然它们是工资支付系统的输出，它们或者是从外面输入进系统的，或者是由系统经过计算产生出来的。沿数据流图从输出端往输入端回溯，分析员应该可以确定每个数据元素的来源。如果分析员不能确定某个数据元素的来源，那么，工资问题的专家应该知道，因此需要再次调查访问。这样有条不紊地分析下去，分析员将逐渐定义出系统的详细功能。

表 2.10 工资明细表上包含的数据元素

教职工编号	职称	洗理费	个人所得税
教职工姓名	生活补贴	课时费	住房公积金
基本工资	书报费	岗位津贴	保险费
职务	交通费	工资总额	实发工资

例如，考虑表 2.10 中的数据元素“工资总额”，它是怎样得出来的呢？从图 2.22 可以看出，包含工资总额这个数据元素的工资明细表是从处理 4（“分发工资明细表”）来的，然而这个处理只是分发已经印好的工资明细表，并不能生成新的数据元素。沿着数据流图回溯（即逆着数据流箭头方向前进），接下来遇到数据存储 D3（“工资明细表”）。数据存储只不过是保存数据的介质，它里面的数据元素必须和它的输入 / 输出数据流相同，它不会改变任何东西，因此也不会生成工资总额这项数据元素。再回溯则遇到处理 3（“加工事务数据”），工资总额一定是由这个处理框计算出来的，因此需要确定相应的算法，以便更准确地定义这个处理框的功能。

根据常识，工资总额等于各项收入之和，也就是基本工资、生活补贴、书报费、交通费、洗理费、课时费（对教师而言）或岗位津贴（对职工而言）之和。虽然不同教职工的基本工资、生活补贴、书报费、交通费和洗理费的数额可能并不相同，但是对同一个人来说，在一段时间内这些数值是稳定不变的，并不需要在每次计算工资总额时都从外面输入这些数据，事实上，在事务数据中并不包含这些数据元素，因此，它们必定保存在某个数据存储中。目前，小王还不知道这些数据保存在何处，他在自己的笔记本中记下“必须搞清楚基本工资、生活补贴、书报费、交通费和洗理费等数据元素存储在何处”。此外，计算工资总额时还必须知道课时费或岗位津贴的数值，课时费和岗位津贴怎样计算呢？小王目前还不知道计算课时费和岗位津贴的方法，他在笔记本中记下“必须深入了解课时费和岗位津贴的计算方法”。然后转去分析另一个重要的数据元素“实发工资”。

显然，从工资总额中扣除个人所得税、住房公积金和保险费之后，余下的就是实发工资。沿数据流图回溯可知，个人所得税、住房公积金和保险费的数值都由处理 3（“加工事务数据”）计算得出。但是，小王目前还不知道怎样计算这些数值，他在自己的笔记本中记下“必须搞清楚个人所得税、住房公积金和保险费的计算方法”。

2. 写出文档初稿

分析员在分析过程中不断加深对目标系统的认识，因此应该把收集到的信息用一种容易修改与更新的形式记录下来。

然而分析员并不是唯一需要使用系统文档的人。在分析阶段结束之前，用户和使用部门负责人必须对分析员的工作成果进行复审，将来还会有其他人需要参考这些文档。通常，一个系统会涉及许多人，他们彼此理解相互通信是至关重要的，文档是主要的通信工具，因此，文档必须是一致的和容易理解的。我们使用的结构化分析方法，主张在需求分析阶段完成的正式文档（需求规格说明书）中必须包含三个重要成分：数据流图、数据字典，以及一组黑盒形式的算法描述。

数据字典是描述数据的信息的集合。对于每个数据元素来说，诸如元素名字，描述，格式，来源和用途等信息，都应该记录在数据字典中。在分析阶段数据字典能帮助分析员组织有关数据的信息，并且是和用户通信的有力工具。此外，它也能起帮助记忆的作用，数据字典要求为每个数据元素记录确定的关键信息，遗漏信息是明显的，因而能够提醒分析员去获得需要的信息。

在分析阶段完成之后，数据字典仍然很有用。因为它是描述数据的信息的集合，在概要设计和详细设计阶段，可以用它产生记录、文件和数据库的格式；在实现阶段它是所有参与实现这个系统的程序员的公共基础，他们可以根据它确定自己的数据描述。在系统投入运行

以后，数据字典可以清楚地告诉维护人员，一个具体的数据元素在系统中是怎样使用的，当必须修改程序时，这样的信息是极其宝贵的。

虽然现在已经有许多商品化的数据字典软件包，但是在我国还没有普及，因此本书使用模拟的数据字典：为每个数据元素填写一张小卡片。使用卡片的好处是可以独立处理每张卡片，分别增添、删除或修改关于每个数据元素的信息，不需要重新抄写就可以把卡片重新排列或重新分组。用大纸或表格作为数据字典显然不如卡片灵活。

小王为工资支付系统中几个数据元素填写的数据字典卡片显示在图 2.23 中。

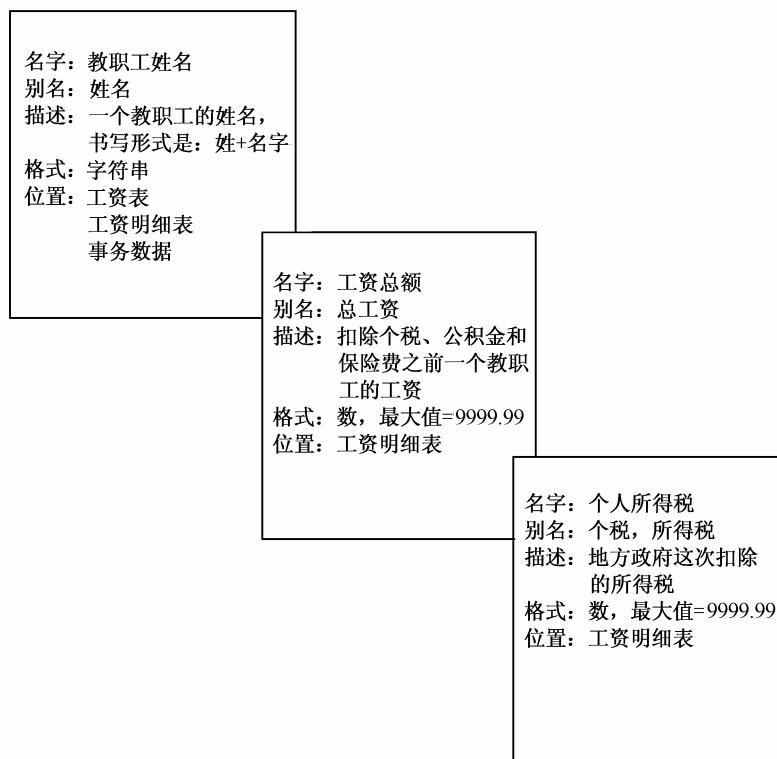


图 2.23 工资支付系统的数据字典卡片

在图 2.23 所示的数据字典卡片中包含的只是最基本的信息，许多实用的数据字典包含的信息比这都更多一些。

分析员还应该以黑盒形式记录算法。所谓黑盒子就是不考虑一个功能或处理的具体实现方法，只把它看作给予输入之后就能够产生一定输出的黑盒子。这正是在这样早期阶段，分析员对算法应持有的正确观点，细节可以等到以后的阶段再确定。当然，如果一个算法是显而易见的，或者在调查访问时了解到一个具体算法，分析员也应该把它们记录下来。但是，描述黑盒子内部的内容并不是本阶段工作的重点。

建议使用 IPO 表记录对算法的初步描述。以后可以进一步精化它，而且在详细设计阶段可以把它作为 HIPO 图的一部分。图 2.24 是小王做出的描述工资总额初步算法的 IPO 表。

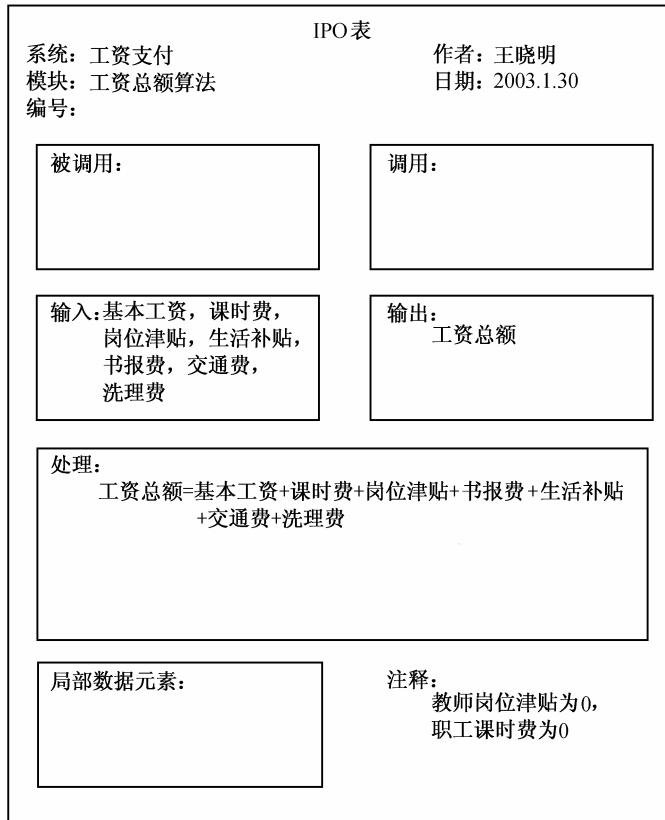


图 2.24 描述工资总额初步算法的 IPO 表

目前，小王写出的文档还仅仅是初稿，写出文档初稿的目的，一方面是记录已经知道的信息，另一方面是让用户审查。随着需求分析工作的深入，这些文档还将进一步修改完善。

3. 定义逻辑系统

现在小王已经划分出许多必须在工资支付系统中流动的数据元素，并且把它们记在了初步的数据字典中，此外，还把某些算法以黑盒子形式记录在 IPO 表中。下一步做什么呢？可以肯定还会存在不少问题。数据字典准确吗？完整吗？算法描述准确完整吗？某些数据元素（例如，基本工资、生活补贴、书报费、交通费、洗理费）是从哪里来的呢？小王必须设法得到这些问题的答案。

以前已经多次讲过，关于工资支付系统的详细信息只能来源于直接工作在这个系统上的人——用户。因此，小王再次访问财务科长和具体处理工资事务的两位会计。数据流图（见图 2.21）是使讨论时焦点集中的极好工具，访问时从数据流图的源点开始，沿着数据流循序讨论。事务数据从教职工流入收集数据这个处理中，小王已经在数据字典中描述了组成事务数据的元素（图 2.23 中未列出这张卡片），这个描述正确吗？有没有遗漏？收集数据这个处理框的功能是什么？审核数据看来需要某种算法，小王描述的算法

正确吗?……对于分析员来说,数据流图、数据字典和算法描述可以作为校核时的清单或帮助记忆的工具。必须校核已经知道的信息,还必须补充目前尚不知道的内容,填补文档中的空白。

例如,考虑工资总额的算法。假设小王和具体处理工资事务的会计正在讨论数据流图中“加工事务数据”这个处理。小王已经知道工资总额的算法并且用图2.24所示的IPO表描述了这个算法,在此之前小王还已经知道基本工资、生活补贴、书报费、交通费和洗理费等数据应该存储起来,那么,它们到底存储在哪个数据存储中呢?会计回答说,这些数据是从人事档案中来的。但是,在图2.21所示的数据流图中并没有一个数据存储保存人事数据,显然应该修改数据流图,补充进这个数据存储。小王就是这样一步一步地分析数据流找出未知的数据元素,未知的数据元素引出访问时的问题,而问题的答案又引入一个以前不知道的系统成分——人事数据存储。

这个新发现又引出下一个问题:人事数据存储是从哪里来的呢?显然,如果系统中有这些数据存在,它们必须从某个地方进入系统。经询问得知,这些数据的来源是人事科,而且需要一个新的处理——“更新人事数据”。

接下来讨论计算课时费和岗位津贴的方法。会计告诉小王,课时费等于教师当月完成的授课时数乘上每课时的课时费,再乘上职称系数和授课班数系数;岗位津贴由职工的职务和当月完成任务的情况决定。通过讨论,小王还进一步了解到,需要在每年年末计算超额课时费,也就是说,如果一名教师一年完成的授课时数超过学校规定的定额,则超出的部分,每课时的课时费按正常值的1.5倍计算。显然,为了计算超额课时费需要保存每名教师当年完成的授课时数,也就是说,需要一个数据存储来存放“年度数据”。

接下来做什么呢?应该讨论“加工事务数据”这个处理需要的其他算法,可进一步了解到更多的算法。例如,在讨论住房公积金的算法时小王了解到,根据国务院2002年3月24日修订的《住房公积金管理条例》的规定,“职工住房公积金的月缴存额为职工本人上一年度月平均工资乘以职工住房公积金缴存比例”,“职工和单位住房公积金的缴存比例均不得低于职工上一年度月平均工资的5%”。因此,需要存储每名教职工上一年度的月平均工资,显然,这个数据元素也应该存储在“年度数据”中,表2.11是年度数据包含的数据元素。相应地,应该有一个新的处理(“更新年度数据”)在每年年末更新年度数据。

最后,小王把新发现的数据源点、处理和数据存储补充到数据流图中,得到新数据流图(见图2.25)。读者应该注意,在上述分析过程中分析员是怎样认真仔细而又有条不紊地工作的,这种结构化分析方法是开发一个好系统的重要途径。

4. 细化数据流图

现在小王对工资支付系统已经比以前了解得更深入、更具体了,原来的数据流图已经不能充分表达他对系统的认识。为了描绘完成复杂功能和供应数据的顺序,应该进一步细化数据流图。

表2.11 年度数据包含的数据元素

教职工编号
教职工姓名
本年度累计工资总额
本年度累计实发工资
本年度累计授课时数
上年度月平均工资

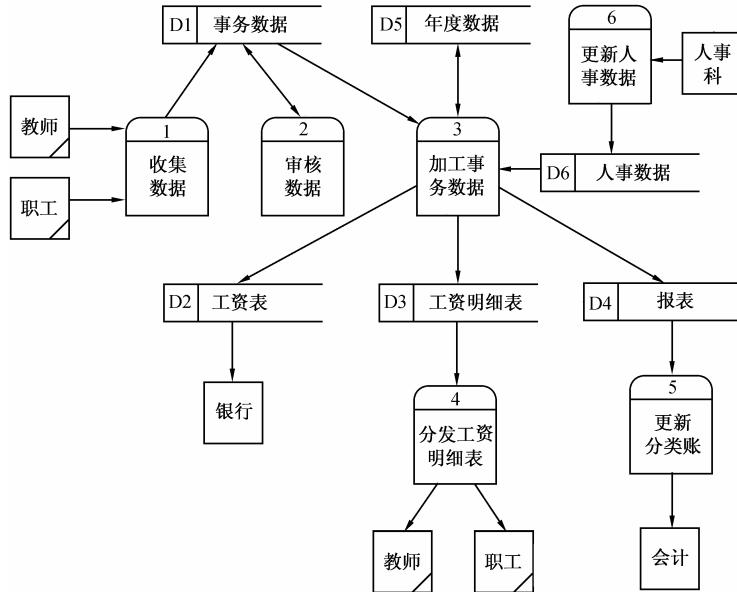


图 2.25 补充后的工资支付系统数据流图

可以通过下述的功能分解方法达到细化数据流图的目的：选取数据流图上功能过分复杂的处理，把它分解成若干个子功能，这些较低层次的子功能成为新数据流图上的处理，它们相应地有自己的数据存储和数据流。

例如，小王感到图 2.25 中“加工事务数据”这个处理的功能太复杂了，用一个处理框不能清晰地描绘它的功能，因此需要进一步分解细化。根据小王现在对加工事务数据功能的了解，他把这个处理分解成下述 5 个关键功能：

(1) 取数据

取出事务数据、年度数据和人事数据供处理。

(2) 计算正常工资

计算不包含超额课时费的工资。

(3) 计算超额课时费

每年年末计算超额课时费，并把得到的数值加到 12 月份的工资总额中。

(4) 更新年度数据

把每月工资总额、实发工资及授课时数累加到年度数据中，且在年末计算本年度月平均工资。

(5) 印表格

印工资表、工资明细表和各种财务报表。

上述每个功能包含一个或多个算法，它们之间的关系可以用一张数据流分图描绘（见图 2.26）。把这个处理框分解的结果加到原来的数据流图中，得到一张更详细的新数据流图（见图 2.27）。注意，为了便于理解和追踪，从处理 3 分解出来的处理的编号是 3.1，3.2，……，3.5。

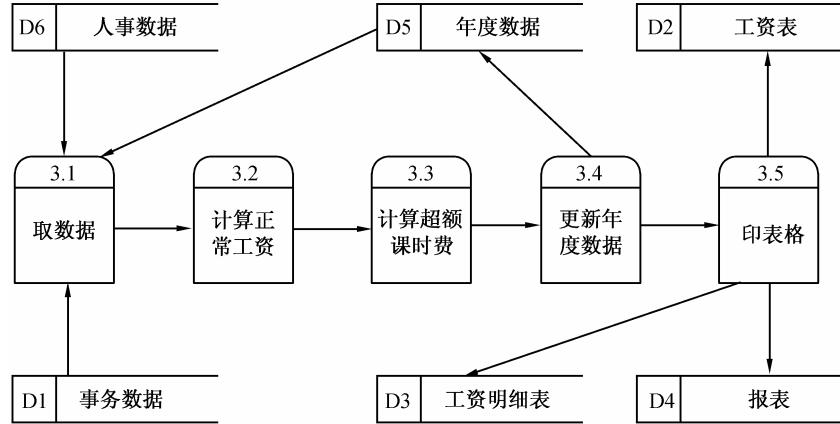


图 2.26 对“加工事务数据”的细化

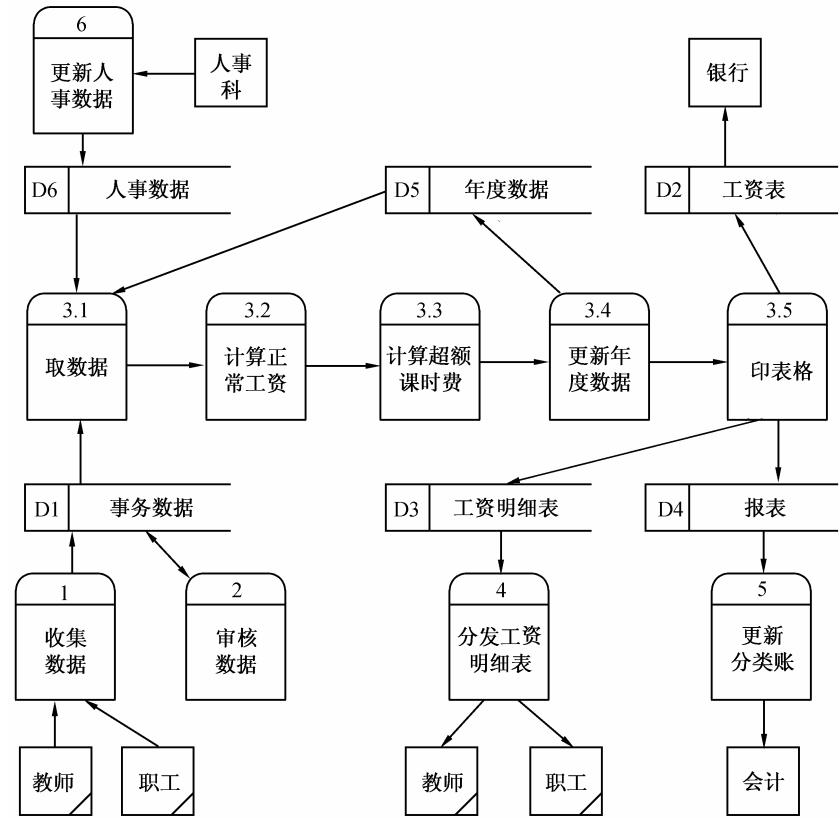


图 2.27 工资支付系统完整的数据流图

新数据流图对工资支付系统的逻辑功能描绘得比以前更深入了，特别是它说明了主要功能之间的逻辑关系。一般说来，其他功能也可以类似地分解，然而对于这个具体系统而言，已经没有必要再分解其他功能了。由处理 3 分解出来的处理还需要进一步分解吗？例如，是否

应该把“取数据”进一步分解成“取事务数据”、“取年度数据”和“取人事数据”呢？这样做可能走得太远了。当你看到一个只从某个具体的数据存储中取数据的处理框时，你会想到什么呢？你可能想到“读”指令，还可能想到出错处理问题。但是，必须记住需求分析阶段的任务是确定系统必须做什么，而不是确定应该怎样做。因此，如果进一步分解将使你开始思考为了完成一个功能需要写出的代码，那么就不应该再分解了。在需求分析阶段分析员应该只在逻辑功能层上工作，代码已经属于物理实现层了。

5. 书写正式文档

数据流图细化之后，组成系统的各个元素之间的关系变得更清楚了。但是每个改动都可能引出新的问题，这些问题的答案可能反过来又导出数据字典的新条目，以及新的或精化了的算法描述。随着分析过程的进展，通过询问与回答的循环将把系统定义得越来越准确、详尽。最终，分析员对系统功能需求有了令人满意的认识，应该把他的认识用正式文档“软件需求规格说明书”准确地记录下来。交出令人满意的软件需求规格说明书，是结束需求分析阶段的标准。数据流图（细化到适当层次）、数据字典和黑盒形式的算法描述，是组成软件需求规格说明书的重要成分。

6. 技术审查和管理复审

需求分析阶段的最后一步是对分析结果进行正式的技术审查。审查小组的组长是从外单位聘请来的一位有经验的系统分析员，小组成员有具体处理工资事务的两名会计，以及这个项目的系统分析员王晓明。图 2.27 所示的数据流图是审查的重点；用数据字典和 IPO 表辅助对数据流图的理解，由审查小组中的一位会计朗读软件需求规格说明书，大家一步一步地仔细分析研究这份文档。小王的责任是回答大家提出来的技术问题。审查的目的是发现错误或遗漏，而不是对前一段的工作进行批评或争论。在审查过程中发现并记录下几个小错误，小王随后应改正这些错误。审查小组成员一致认为小王确实理解了工资支付问题，并且知道了为了解决这个问题必须做什么。最后，大家在审查表上签名，指出迄今为止的工作在技术上是令人满意的。

在转入概要设计阶段之前，还必须进行管理复审。虽然技术审查已经表明能够为该职业高中开发出令人满意的工资支付系统，但是，校长还关心其他方面的问题。进度如何？项目能按时完成吗？成本怎样？在可行性研究阶段粗略估计开发成本是 4.2 万元人民币，小王现在对系统有了更具体的了解，他是否仍然认为以前对成本的估计基本上是合理的呢？上述这些问题都是校长非常关心的问题。假设小王在需求分析过程中没有发现意外情况，最初的成本估计和进度表现在看来仍然是合理的。因此，校长批准这项开发工作继续进行下去。

2.15 小结

传统的软件工程方法学采用结构化分析技术完成系统分析（问题定义、可行性研究、需求分析）工作。

可行性研究进一步探讨问题定义阶段所确定的问题是否有可行的解。在对问题正确定义的基础上，通过分析问题（往往需要研究现在正在使用的系统），导出试探性的解，然后

复查并修正问题定义，再次分析问题，改进提出的解法，……经过定义问题，分析问题，提出解法的反复过程，最终提出一个符合系统目标的高层次的逻辑模型。然后根据系统的这个逻辑模型设想各种可能的物理系统，并且从技术、经济和操作等各方面分析这些物理系统的可行性。最后，系统分析员提出一个推荐的行动方针，提交用户和使用部门负责人审查批准。

在表达分析员对现有系统的认识和描绘他对未来的物理系统的设想时，系统流程图是一个很好的工具。系统流程图实质上是物理数据流图，它描绘组成系统的主要物理元素以及信息在这些元素间流动和处理的情况。

数据流图的基本符号只有四种，它是描绘系统逻辑模型的极好工具。通常数据字典和数据流图共同构成系统的逻辑模型。没有数据字典精确定义数据流图中每个元素，数据流图就不够严密；然而没有数据流图，数据字典也很难发挥作用。

成本 / 效益分析是可行性研究的一项重要内容，是使用部门负责人从经济角度判断是否继续投资于这项工程的主要依据。

需求分析是软件生命周期的一个重要阶段，它最根本的任务是确定为了满足用户的需要系统必须做什么。具体地说，应该确定系统必须具有的功能和性能，系统要求的运行环境，并且预测系统发展的前景；必须仔细分析系统中的数据，既要分析系统中的数据流又要分析长期使用的数据存储。通过分析应该得出用数据流图、ER图、数据字典和简洁的算法描述所定义的详细的系统逻辑模型。图形工具比文字叙述能更好地表达重要的细节，数据流图能够极好地概括描述一个系统的信息。ER图能直观、准确地描绘系统的数据需求。数据字典也是重要的，数据是把一个系统的各个组成元素连接在一起的“粘合剂”，为了成功地把所有系统元素连接起来，这些元素必须共享公共的数据定义，数据字典正是这些数据定义的集合。算法同样是重要的，分析的目的是确定系统必须做什么，广义地说，任何一个计算机系统的基本功能都是把输入数据转变成输出信息，算法定义了转变的规则。因此，没有对算法的了解就不能确切知道系统必须做什么。此外，在需求分析阶段还应该根据对目标系统的更深入更具体的认识，修正开发目标系统的计划。

需求分析是在可行性研究的基础上进行的，可行性研究实质上是一次完整的分析和设计过程，只不过是在抽象的层次上进行的大大压缩和简化了的分析和设计过程。因此在可行性研究阶段已经进行了某些初步的分析，特别是已经得出了高层次的数据流图。但是，需求分析的主要任务是得出详细的系统逻辑模型。通常需求分析工作从可行性研究得出的数据流图出发，首先确定构成输出数据的各个数据元素，沿数据流图回溯寻求每个数据元素的源，在此过程中确定必要的处理算法并补充必要的数据元素，同时也会产生一些新的问题。在寻求这些问题的答案时，将澄清一些算法并划分出更多的数据元素，可能也会再遇到一些问题。对这些问题的解答又将导致对系统的更深入更具体的认识。在对系统的主要处理算法有充分了解之后，应该把数据流图进一步分层细化。

为了更准确、更具体地确定用户的要求，往往需要快速地构造出目标系统的原型，供用户试用并听取用户的反馈意见。此外，访谈和简易的应用规格说明技术也是与用户沟通，获取用户需求的常用方法。

通过需求分析应该得出用数据流图、ER图、数据字典和 IPO 图（或 PDL 等其他描述算法的工具）描绘的精确的系统逻辑模型。为了提高文档的可理解性，还可以用层次方框图或

Warnier 图等图形工具辅助描绘系统中的数据结构。通常，数据流图、数据字典和 IPO 图(表)，是需求分析阶段正式文档（“软件需求规格说明书”）的主要成分。本章给出的软件需求规格说明书编写大纲，可供读者在实际工作中使用。

需求分析的结果是软件开发的基础，必须仔细验证它的正确性，开发人员必须和用户取得完全一致的意见，需求分析的文档应该被用户所确认。然而这并不意味着分析员应该不加分析地全盘接受用户提出的所有要求，对用户提出的笼统要求应该分解细化；对用户提出的含混要求需要进一步澄清；对用户提出的不切实际的要求必须做深入细致的解释说服工作，以便动员用户放弃不合理的要求。

最后，我们用一个工资支付系统的结构化分析实例，作为对本章内容的总结。这个例子深入细致地讲述了工资支付系统的问题定义、可行性研究和需求分析过程，在对这个实际系统进行结构化分析的过程中，综合运用了本章前面各节中讲述的主要技术方法。把这个例子和本章前面各节讲述的内容结合起来认真学习，对读者深入理解结构化分析方法并逐步学会运用这种方法解决实际问题，是很有帮助的。

习题二

1. 在软件开发的早期阶段为什么要进行可行性研究？应该从哪些方面研究目标系统的可行性？

2. 为什么要验证软件需求？怎样验证软件需求？

3. 银行计算机储蓄系统的工作过程大致如下：储户填写的存款单或取款单由业务员键入系统，如果是存款则系统记录存款人姓名、住址（或电话号码）、身份证号码、存款类型、存款日期、到期日期、利率及密码（可选）等信息，并印出存款单给储户；如果是取款而且存款时留有密码，则系统首先核对储户密码，若密码正确或存款时未留密码，则系统计算利息并印出利息清单给储户。

请用数据流图描绘本系统的功能，并用实体—关系图描绘系统中的数据对象。

4. 为方便旅客，某航空公司拟开发一个机票预订系统。旅行社把预订机票的旅客信息（姓名、性别、工作单位、身份证号码、旅行时间、旅行目的地等）输入进该系统，系统为旅客安排航班，印出取票通知和账单，旅客在飞机起飞的前一天凭取票通知和账单交款取票，系统校对无误即印出机票给旅客。

请用实体—关系图描绘本系统中的数据对象并用数据流图描绘本系统的功能。

5. 目前住院病人主要由护士护理，这样做不仅需要大量护士，而且由于不能随时观察危重病人的病情变化，还会延误抢救时机。某医院打算开发一个以计算机为中心的患者监护系统，请画出本系统的实体—关系图和数据流图。

医院对患者监护系统的基本要求是随时接收每个病人的生理信号（脉搏、体温、血压、心电图等），定时记录病人情况以形成患者日志，当某个病人的生理信号超出医生规定的安全范围时向值班护士发出警告信息，此外，护士在需要时还可以要求系统印出某个指定病人的病情报告。

6. 某高校使用的电话号码有以下几类：校内电话号码由 4 位数字组成，第 1 位数字不是

0；校外电话又分为本市电话和外地电话两类，打校外电话需先拨 0，如果是本市电话再接着拨 8 位电话号码（第 1 位不是 0），如果是外地电话则需拨 3 位或 4 位区码，然后再拨 7 位或 8 位电话号码（第 1 位不是 0）。

请用 2.11 节讲述的方法，定义该校使用的电话号码。

7. 全面完成 2.14 节讲述的工资支付系统的需求分析工作，写出该系统的需求规格说明书。

第3章 结构化设计

经过需求分析阶段的工作，系统必须“做什么”已经清楚了，现在是决定“怎样做”的时候了。为此，必须首先进行设计，软件设计的目标是设计出符合用户需求的软件的模型。

传统的软件工程方法学采用结构化设计技术完成软件设计工作，结构化设计技术主要有以下几个要点：

- ? 软件系统由层次化结构的模块构成；
 - ? 模块是单入口单出口的；
 - ? 构造和联结模块的基本准则是模块独立；
 - ? 用图（主要是层次图或结构图）表达软件系统的结构，并使之与问题结构尽量一致。
- 通常把软件设计划分为概要设计和详细设计这样两个阶段。

3.1 软件设计的任务

3.1.1 概要设计的任务

概要设计也称为总体设计或初步设计，这个设计阶段主要有两项任务。

1. 设计实现软件的最佳方案

概要设计过程首先设想实现目标系统的各种可能的方案，需求分析阶段得到的数据流图是设想各种可能方案的基础。一种常用的方法是，设想把数据流图中的处理分组（即画自动化边界）的各种可能方法。

然后，分析员从设想出的这些供选择的方案中选取若干个合理的方案。在判断哪些方案合理时，应该考虑在系统分析过程中确定下来的项目规模和目标，有时还需要进一步征求用户的意见。应该为每个合理的方案都画一份系统流程图，列出组成系统的物理元素（程序、文件、数据库、人工过程和文档等）清单，进行成本 / 效益分析，并且制定实现这个方案的进度计划。

最后，分析员应该综合分析对比所选取的各种合理方案的利弊，从中选出一个最佳方案，并且为推荐的这个最佳方案制定详细的实现计划。一旦用户和使用部门的负责人接受了分析员推荐的方案，就应该开始概要设计的第二项工作。

2. 设计软件体系结构

概要设计的第二项重要任务是设计软件的体系结构，也就是确定软件系统中每个程序是由哪些模块组成的，以及这些模块相互间的关系。

通常，程序中的一个模块完成一个适当的子功能。应该把模块组织成良好的层次系统，顶层模块通过调用它的下层模块来实现程序的完整功能，顶层模块下面的每个模块再调用更下层的模块从而完成程序的一个子功能，最下层的模块完成最具体的功能。

设计出初步的软件结构之后，分析员还应该从多方面改进软件结构，以便得到更合理的软件结构。

从上面的叙述中不难看出，在详细设计之前先进行概要设计的必要性：分析员可以站在全局高度上，花较少的成本，在比较抽象的层次上分析对比多种可能的系统实现方案和多种可能的软件体系结构，从中选出最佳方案和最合理的软件结构，从而用较低的成本开发出较高质量的软件系统。

3.1.2 详细设计的任务

详细设计阶段的根本目标，是确定怎样具体地实现所要求的软件系统，也就是说，经过这个阶段的设计工作，应该得出对目标系统的精确描述，从而在编码阶段可以把这个描述直接翻译成某种程序设计语言书写的程序。

具体说来，详细设计主要有以下三项任务：

- ? 过程设计，即设计软件体系结构中所包含的每个模块的实现算法；
- ? 数据设计，即设计软件中需要的数据结构；
- ? 接口设计，即设计软件内部各模块之间、软件与协作系统之间以及软件与使用它的人之间的通信方式。

详细设计阶段的任务还不是具体地编写程序，而是要设计出程序的“蓝图”，以后程序员将根据这个蓝图写出实际的程序代码。因此，详细设计的结果基本上决定了最终的程序代码的质量。考虑程序代码的质量时必须注意，程序的“读者”有两个，那就是计算机和人。在软件的生命周期中，设计测试方案，诊断程序错误，修改和改进程序等等都必须首先读懂程序。实际上对于长期使用的软件系统而言，人读程序的时间可能比写程序的时间还要长得多。因此，衡量程序的质量不仅要看它的逻辑是否正确，性能是否满足要求，更主要的是要看它是否容易阅读和理解。详细设计的目标不仅仅是逻辑上正确地实现每个模块的功能，更重要的是设计出的处理过程应该尽可能简明易懂。结构程序设计技术是实现上述目标的关键技术，因此是详细设计的逻辑基础。

3.2 从分析过渡到设计

系统分析的基本任务是定义用户需要的软件系统，也就是回答系统必须“做什么”这个关键问题；系统设计的基本任务是设计实现目标系统的具体方案，也就是回答“怎样做”这个关键问题。虽然系统分析与系统设计的任务性质不同，但是两者却有十分密切的关系。

软件设计必须依据用户对软件的需求来进行，因此，结构化分析的结果是结构化设计的最基本、最重要的输入信息。

体系结构设计的任务是，确定程序由哪些模块组成以及这些模块相互间的关系。在需求分析阶段画出的数据流图，是进行体系结构设计的主要依据，为体系结构设计提供了最基本

的输入信息。本章 3.7 节讲述的面向数据流的设计方法，甚至直接从数据流图映射出软件结构。

数据设计把需求分析阶段创建的信息模型转变成实现软件所需要的数据结构。在实体—联系图中定义的数据和数据之间的关系，以及数据字典中给出的详细的数据定义，共同为数据设计活动奠定了坚实的基础。

接口设计的结果描述了软件内部、软件与协作系统之间以及软件与使用它的人之间的通信方式。接口意味着信息的流动（数据流或控制流），因此，数据流图提供了进行接口设计所需要的基本信息。

过程设计决定程序中包含的每个模块的实现算法，需求分析阶段画出的 IPO 图（表）为过程设计奠定了基础。

虽然需求分析为结构化设计提供了最基本、最重要的输入信息，但是，并不是说我们可以简单地把结构化分析的结果映射成结构化设计的结果。实际上，结构化设计过程综合了下述诸多因素：从以往开发类似软件的经验中获得的直觉和判断力，指导软件模型演化的一组准则（或称为原理）和启发规则，评价软件质量的一组标准，以及导出最终设计结果的迭代过程。

我们在软件设计过程中所做出的决策，将最终决定软件开发能否成功，更重要的是，这些设计决策将决定软件维护的难易程度。

软件设计之所以如此重要，是因为设计是软件开发过程中决定软件产品质量的关键阶段。设计为我们提供了可以进行质量评估的软件表示（即软件模型），设计是我们把用户需求准确地转变为最终的软件产品的唯一方法。软件设计是后续的一切软件开发和维护步骤的基础，如果不进行设计，我们就会冒构造出不稳定的软件系统的风险：稍做改动这样的系统就可能崩溃；这样的系统很难维护；这样的系统很难测试；直到软件工程过程的后期（例如，编码结束），才能评价这样的系统的质量，但是，这时才发现软件质量问题已经为时过晚了。

3.3 软件设计准则

刚才讲过，软件设计是软件开发过程中决定软件产品质量的关键阶段，也就是说，只有软件设计是高质量的才有可能最终开发出高质量的软件产品，反之，如果设计质量很低，则最终开发出的软件产品的质量也必定很低。为了保证设计的质量，在软件设计过程中应该遵循一些基本的准则（或称为原理）。

下面讲述在软件设计过程中应该遵循的基本准则以及相关的概念。

3.3.1 模块化与模块独立

模块化和模块独立，是关系非常密切的两条设计准则，下面介绍这两条准则。

1. 模块化

模块是由边界元素限定的相邻的程序元素（例如，数据说明，可执行的语句）的序列，而且有一个总体标识符来代表它。像 Pascal 或 Ada 这样的块结构语言中的 Begin...end 对，或者 C, C++ 和 Java 语言中的 { ... } 对，都是边界元素的例子。因此，过程、函数、子程序和宏等，都可作为模块。面向对象范型中的对象（见第 6 章）是模块，对象内的方法也是模块。

模块是构成程序的基本构件。

模块化就是把程序划分成独立命名且可独立访问的模块，每个模块完成一个子功能，把这些模块集成起来构成一个整体，可以完成指定的功能满足用户的需求。

有人说，模块化是为了使一个复杂的大型程序能被人的智力所管理，软件应该具备的惟一属性。如果一个大型程序仅由一个模块组成，它将很难被人所理解。下面根据人类解决问题的一般规律，论证上面的结论。

设函数 $C(x)$ 定义问题 x 的复杂程度，函数 $E(x)$ 确定解决问题 x 需要的工作量（时间），对于两个问题 P_1 和 P_2 ，如果

$$C(P_1) > C(P_2)$$

显然

$$E(P_1) > E(P_2)$$

根据人类解决一般问题的经验，另一个有趣的规律是

$$C(P_1 + P_2) > C(P_1) + C(P_2)$$

也就是说，如果一个问题由 P_1 和 P_2 两个问题组合而成，那么它的复杂程度大于分别考虑每个问题时的复杂程度之和。

综上所述，得到下面的不等式

$$E(P_1 + P_2) > E(P_1) + E(P_2)$$

这个不等式导致“各个击破”的结论——把复杂的问题分解成许多容易解决的小问题，原来的问题也就容易解决了。这就是模块化的根据。

由上面的不等式似乎还能得出下述结论：如果无限地分割软件，最后为了开发软件而需要的工作量也就小得可以忽略了。事实上，还有另一个因素在起作用，从而使得上述结论不能成立。参见图 3.1，当模块数目增加时每个模块的规模将减小，开发单个模块需要的成本（工作量）确实减少了；但是，随着模块数目增加，设计模块间接口所需要的工作量也将增加。根据这两个因素，得出了图中的总成本曲线。每个程序都相应地有一个最适当的模块数目 M ，使得系统的开发成本最小。

虽然目前我们还不能精确地算出 M 的数值，但是在考虑程序模块化的时候，总成本曲线确实是有用的指南。下面即将讲述的模块独立准则以及 3.5 节将讲述的启发规则，可以在一定程度上帮助我们决定合适的模块数目。

采用模块化原理可以使软件结构清晰，不仅容易设计也容易阅读和理解。因为程序错误通常局限在有关的模块及它们之间的接口中，所以模块化使软件容易测试和调试，因而有助于提高软件的可靠性。因为变动往往只涉及少数几个模块，所以模块化能够提高软件的可修改性。模块化也有助于软件开发工程的组织管理，一个复杂的大型程序可以由许多程序员分工编写不同的模块，并且可以进一步分配技术熟练的程序员编写困难的模块。

2. 模块独立

是不是只要把程序划分成若干个模块，就可以获得上述的模块化所带来的好处呢？事实

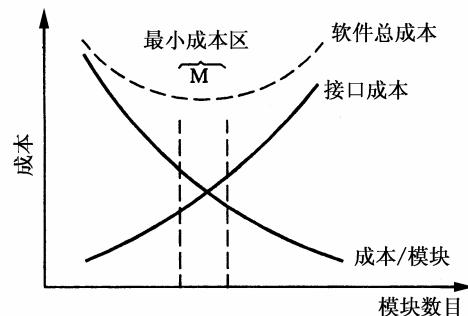


图 3.1 模块化和软件成本的关系

并非如此。模块划分和组织得合理，能大大地提高软件的质量，反之，软件质量不仅不会提高反而可能会下降。

怎样划分和组织模块才合理呢？一条指导模块划分和组织的重要准则，就是“模块独立”。

开发具有独立功能而且和其他模块之间没有过多的相互作用的模块，就可以做到模块独立。换句话说，希望这样设计软件结构，使得每个模块完成一个相对独立的特定子功能，并且和其他模块之间的关系很简单。

为什么模块的独立性很重要呢？主要有两条理由：第一，有效的模块化（即具有独立的模块）的软件比较容易开发出来。这是由于能够分割功能而且接口可以简化，当许多人分工合作开发同一个软件时，这个优点尤其重要。第二，独立的模块比较容易测试和维护。这是因为相对来说，修改设计和程序需要的工作量比较小，错误传播范围小，需要扩充功能时能够“插入”模块。总之，模块独立是好设计的关键，而设计又是决定软件质量的关键环节。

模块的独立程度可以由两个定性标准度量，这两个标准分别称为内聚和耦合。耦合衡量不同模块彼此间互相依赖（连接）的紧密程度；内聚衡量一个模块内部各个元素彼此结合的紧密程度。3.4 节将仔细讲述这两个标准。

3.3.2 抽象

人类在认识复杂现象的过程中使用的最强有力的思维工具是抽象。人们在实践中认识到，在现实世界中一定事物、状态或过程之间总存在着某些相似的方面（共性）。把这些相似的方面集中和概括起来，暂时忽略它们之间的差异，这就是抽象。或者说抽象就是抽出事物的本质特性而暂时不考虑它们的细节。

由于人类思维能力的限制，如果每次面临的因素太多，是不可能做出精确思维的。处理复杂系统的惟一有效的方法是用层次的方式构造和分析它。一个复杂的动态系统首先可以用一些高级的抽象概念构造和理解，这些高级概念又可以用一些较低级的概念构造和理解，如此进行下去，直至最低层次的具体元素。

这种层次的思维和解题方式必须反映在定义动态系统的程序结构之中，每级的一个概念将以某种方式对应于程序的一组成分。

当我们考虑对任何问题的模块化解法时，可以提出许多抽象的层次。在抽象的最高层次使用问题环境的语言，以概括的方式叙述问题的解法；在较低抽象层次采用更过程化的方法，把面向问题的术语和面向实现的术语结合起来叙述问题的解法；最后，在最低的抽象层次用可以直接实现的方式叙述问题的解法。

软件工程过程的每一步都是对软件解法的抽象层次的一次精化。在可行性研究阶段，软件作为系统的一个完整部件；在需求分析期间，软件解法是使用在问题环境内熟悉的方式描述的；当我们由总体设计向详细设计过渡时，抽象的程度也就随之减少了；最后，当源程序写出来以后，也就达到了抽象的最低层。

逐步求精（见 3.3.3 小节）和模块化的概念，与抽象是紧密相关的。随着软件开发工程的进展，在软件结构每一层中的模块，表示了对软件抽象层次的一次精化。事实上，软件结构顶层的模块，控制了系统的主要功能并且影响全局；在软件结构底层的模块，完成对数据的一个具体处理，用自顶向下由抽象到具体的方式分配控制，简化了软件的设计和实现，提高了软件的可理解性和可测试性，并且使软件更容易维护。

3.3.3 逐步求精

逐步求精是人类解决复杂问题时采用的基本技术，也是许多软件工程技术（例如，规格说明技术，设计和实现技术、测试和集成技术）的基础。可以把逐步求精定义为：“为了能集中精力解决主要问题而尽量推迟对问题细节的考虑。”

逐步求精之所以如此重要，是因为人类的认知过程遵守 Miller 法则：一个人在任何时候都只能把注意力集中在 7 ± 2 个知识块上。

但是，在开发软件的过程中，软件工程师的大脑需要在一段时间内考虑的知识块数远多于 7，例如，一个程序通常不止使用 7 个数据，一个用户也往往有不止 7 个方面的需求。逐步求精技术的强大作用就在于，它能帮助软件工程师把精力集中在与当前开发阶段最相关的那些方面上，而忽略那些对整体解决方案来说是必要的，然而目前还不需要考虑的细节，这些细节将留到以后再考虑。Miller 法则是人类智力的基本局限，我们不可能战胜我们的自然本性，只能承认这个法则，接受自身的局限性，并在这个前提下尽我们的最大努力工作。

事实上，可以把逐步求精看作是一项把一个时期内必须解决的种种问题按优先级排序的技术。逐步求精确保每个问题都将被解决，而且每个问题都在适当的时候解决，但是，在任何时候一个人都不需要同时处理 7 个以上知识块。

逐步求精最初是由 Niklaus Wirth 提出的一种自顶向下的设计策略。按照这种设计策略，程序的体系结构是通过逐步精化过程细节的层次而开发出来的。通过逐步分解对功能的宏观陈述而开发出层次结构，直至最终得出用程序设计语言表达的程序。

Wirth 本人对逐步求精策略曾做过如下的概括说明：

“我们对付复杂问题的最重要的办法是抽象，因此，对一个复杂的问题不应该立刻用计算机指令、数字和逻辑符号来表示，而应该用较自然的抽象语句来表示，从而得出抽象程序。抽象程序对抽象的数据进行某些特定的运算并用某些合适的记号（可能是自然语言）来表示。对抽象程序做进一步的分解，并进入下一个抽象层次，这样的精细化过程一直进行下去，直到程序能被计算机接受为止。这时的程序可能是用某种高级语言或机器指令书写的。”

求精实际上是细化过程。我们从在高抽象级别定义的功能陈述（或信息描述）开始。也就是说，该陈述仅仅概念性地描述了功能或信息，但是并没有提供功能的内部工作情况或信息的内部结构。求精要求设计者细化原始陈述，随着每个后续求精（细化）步骤的完成而提供越来越多的细节。

抽象与求精是一对互补的概念。抽象使得设计者能够说明过程和数据，同时却忽略低层细节。事实上，可以把抽象看作是一种通过忽略多余的细节同时强调有关的细节，而实现逐步求精的方法。求精则帮助设计者在设计过程中揭示出低层细节。这两个概念都有助于设计者在设计演化过程中创造出完整的设计模型。

3.3.4 信息隐藏

应用模块化原理时，自然会产生一个问题：“为了得到最好的一组模块，应该怎样分解软件”。信息隐藏原理指出：应该这样设计和确定模块，使得一个模块内包含的信息（过程和数据）对于不需要这些信息的模块来说，是不能访问的。

实际上，应该隐藏的不是有关模块的一切信息，而是模块的实现细节。因此，有人主张

把这条原理称为“细节隐藏”。但是，由于历史原因，人们已经习惯于把这条原理称为：“信息隐藏”。

“隐藏”意味着有效的模块化可以通过定义一组独立的模块来实现。这些模块彼此之间只交换那些为了完成系统功能而必须交换的信息。

如果在测试期间和以后的软件维护期间需要修改软件，那么使用信息隐藏原理作为模块化系统设计的标准就会带来极大好处。因为绝大多数数据和过程对于软件的其他部分而言是隐蔽的（也就是“看”不见的），在修改期间由于疏忽而引入的错误就很少可能传播到软件的其他部分。

3.4 度量模块独立性的标准

“模块独立”概念是模块化、抽象、逐步求精和信息隐藏等概念的直接结果，也是完成有效的模块设计的基本准则。

模块的独立程度通常由两个定性标准来度量，这两个标准分别是内聚和耦合。

3.4.1 耦合

耦合是对一个软件结构内不同模块之间互连程度的度量。耦合强弱取决于模块间接口的复杂程度，进入或访问一个模块的点，以及通过接口的数据。

在软件设计中应该追求尽可能松散耦合的系统。在这样的系统中可以研究、测试或维护任何一个模块，而不需要对系统的其他模块有很多了解。此外，由于模块间联系简单，发生在一处的错误传播到整个系统的可能性就很小。因此，模块间的耦合程度强烈影响系统的可理解性、可测试性、可靠性和可维护性。

下面介绍怎样具体区分模块间耦合程度的强弱。

如果两个模块中的每一个都能独立地工作而不需要另一个模块的存在，那么它们彼此完全独立，这意味着模块间无任何连接，耦合程度最低。但是，在一个软件系统中不可能所有模块之间都没有任何连接。

如果两个模块彼此间通过参数交换信息，而且交换的信息仅仅是数据，那么这种耦合称为数据耦合。如果传递的信息中有控制信息（尽管有时这种控制信息以数据的形式出现），则这种耦合称为控制耦合。

数据耦合是低耦合。系统中至少必须存在这种耦合，因为只有当某些模块的输出数据作为另一些模块的输入数据时，系统才能完成有价值的功能。一般说来，一个系统内可以只包含数据耦合。控制耦合是中等程度的耦合，它增加了系统的复杂程度。控制耦合往往是多余的，在把模块适当分解之后通常可以用数据耦合代替它。

如果被调用的模块需要使用作为参数传递进来的数据结构中的所有元素，那么把整个数据结构作为参数传递是完全正确的。但是，当把整个数据结构作为参数传递而被调用的模块只需要使用一部分数据元素时，就出现了特征耦合。在这种情况下，被调用的模块可以处理的数据多于它确实需要的数据，这将导致对数据的访问失去控制，从而给计算机犯罪提供了机会。

当两个或多个模块通过一个公共数据环境相互作用时，它们之间的耦合称为公共环境耦合。公共环境可以是全局变量、共享的通信区、内存的公共覆盖区、任何存储介质上的文件和物理设备等。

公共环境耦合的复杂程度随耦合的模块个数而变化，当耦合的模块个数增加时复杂程度显著增加。如果只有两个模块有公共环境，那么这种耦合有下面两种可能。

? 一个模块往公共环境送数据，另一个模块从公共环境取数据。这是数据耦合的一种形式，是比较松散的耦合。

? 两个模块都既往公共环境送数据又从里面取数据，这种耦合比较紧密，介于数据耦合和控制耦合之间。

如果两个模块共享的数据很多，都通过参数传递可能很不方便，这时可以利用公共环境耦合。

最高程度的耦合是内容耦合。如果出现下列情况之一，两个模块间就发生了内容耦合：

- ? 一个模块访问另一个模块的内部数据；
- ? 一个模块不通过正常入口而转到另一个模块的内部；
- ? 两个模块有一部分程序代码重叠（只可能出现在汇编程序中）；
- ? 一个模块有多个入口（这意味着一个模块有几种功能）。

应该坚决避免使用内容耦合。事实上许多高级程序设计语言已经设计成不允许在程序中出现任何形式的内容耦合。

总之，耦合是影响软件复杂程度的一个重要因素。应该采取下述设计原则：

尽量使用数据耦合，少用控制耦合，限制公共环境耦合的范围，完全不用内容耦合。

3.4.2 内聚

内聚度量一个模块内的各个元素彼此结合的紧密程度，它是信息隐藏概念的自然扩展。简单地说，理想内聚的模块只做一件事情。

设计时应该力求做到高内聚，通常中等程度的内聚也是可以采用的，而且效果和高内聚相差不多；但是，低内聚很坏，不要使用。

内聚和耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合。内聚和耦合都是进行模块化设计的有力工具，但是实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上。

低内聚有如下几类：如果一个模块完成一组任务，这些任务彼此间即使有关系，关系也是很松散的，就叫做偶然内聚。有时在写完一个程序之后，发现一组语句在两处或多处出现，于是把这些语句作为一个模块以节省内存，这样就出现了偶然内聚的模块。如果一个模块完成的任务在逻辑上属于相同或相似的一类（例如，一个模块产生各种类型的全部输出），则称为逻辑内聚。如果一个模块包含的任务必须在同一段时间内执行（例如，模块完成各种初始化工作），就叫时间内聚。

在偶然内聚的模块中，各种元素之间没有实质性联系，很可能在一种应用场合需要修改这个模块，在另一种应用场合又不允许这种修改，从而陷入困境。事实上，偶然内聚的模块出现修改错误的概率比其他类型的模块高得多。

在逻辑内聚的模块中，不同功能混在一起，公用部分程序代码，即使局部功能的修改有

时也会影响全局。因此，这类模块的修改也比较困难。

时间关系在一定程度上反映了程序的某些实质，所以时间内聚比逻辑内聚好一些。

中内聚主要有两类：如果一个模块内的处理元素是相关的，而且必须以特定次序执行，则称为过程内聚。使用程序流程图作为工具设计软件时，常常通过研究流程图确定模块的划分，这样得到的往往是过程内聚的模块。如果模块中所有元素都使用同一个输入数据和（或）产生同一个输出数据，则称为通信内聚。

高内聚也有两类：如果一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行（通常一个处理元素的输出数据作为下一个处理元素的输入数据），则称为顺序内聚。根据数据流图划分模块时，通常得到顺序内聚的模块，这种模块彼此间的连接往往比较简单。如果模块内所有处理元素属于一个整体，完成一个单一的功能，则称为功能内聚。功能内聚是最高程度的内聚。

耦合和内聚的概念是 Constantine , Yourdon , Myers 和 Stevens 等人提出来的。按照他们的观点，如果给上述七种内聚的优劣评分，将得到如下结果：

功能内聚	10 分	时间内聚	3 分
顺序内聚	9 分	逻辑内聚	1 分
通信内聚	7 分	偶然内聚	0 分
过程内聚	5 分		

事实上，没有必要精确确定内聚的级别。重要的是设计时力争做到高内聚，并且能够辨认出低内聚的模块，有能力通过修改设计提高模块的内聚程度降低模块间的耦合程度，从而获得较高的模块独立性。

3.5 启发规则

软件工程师们在开发计算机软件的长期实践中积累了丰富的经验，总结这些经验得出了一些启发规则。这些启发规则虽然不像前两节讲述的基本原理那样普遍适用，但是在许多场合仍然能给软件工程师有益的启示，往往能帮助他们找到改进软件设计提高软件质量的途径，因此有助于实现有效的模块化。下面介绍几条常用的启发规则。

3.5.1 改进软件结构提高模块独立性

设计出软件的初步结构以后，应该审查分析这个结构，通过模块分解或合并，力求降低耦合提高内聚。例如，多个模块公有的一个子功能可以独立成一个模块，由这些模块调用；有时可以通过分解或合并模块以减少控制信息的传递及对全程数据的引用，并且降低接口的复杂程度。

3.5.2 模块规模应该适中

经验表明，一个模块的规模不应过大，最好能写在一页纸内（通常不超过 60 行语句）。有人从心理学角度研究得知，当一个模块包含的语句数超过 30 以后，模块的可理解程度迅速下降。

过大的模块往往是由于分解不充分，但是进一步分解必须符合问题结构，一般说来，分解后不应该降低模块独立性。

过小的模块开销大于有效操作，而且模块数目过多将使系统接口复杂。因此过小的模块有时不值得单独存在，特别是只有一个模块调用它时，通常可以把它合并到上级模块中去而不必单独存在。

3.5.3 深度、宽度、扇出和扇入都应适当

深度表示软件结构中控制的层数，它往往能粗略地标志一个系统的大小和复杂程度。深度和程序长度之间应该有粗略的对应关系，当然这个对应关系是在一定范围内变化的。如果层数过多则应该考虑是否许多管理模块过分简单了，能否适当合并。

宽度是软件结构内同一个层次上的模块总数的最大值。一般说来，宽度越大系统越复杂。对宽度影响最大的因素是模块的扇出。

扇出是一个模块直接控制（调用）的模块数目，扇出过大意味着模块过分复杂，需要控制和协调过多的下级模块；扇出过小（例如总是1）也不好。经验表明，一个设计得好的典型系统的平均扇出通常是3或4（扇出的上限通常是5~9）。

扇出太大一般是因为缺乏中间层次，应该适当增加中间层次的控制模块。扇出太小时可以把下级模块进一步分解成若干个子功能模块，或者合并到它的上级模块中去。当然分解模块或合并模块必须符合问题结构，不能违背模块独立原理。

一个模块的扇入表明有多少个上级模块直接调用它，扇入越大则共享该模块的上级模块数目越多，这是有好处的，但是，不能违背模块独立原理单纯追求高扇入。

观察大量软件系统后发现，设计得很好的软件结构通常顶层扇出比较高，中层扇出较少，底层扇入到公共的实用模块中去（底层模块有高扇入）。

3.5.4 模块的作用域应该在控制域之内

模块的作用域定义为受该模块内一个判定影响的所有模块的集合。模块的控制域是这个模块本身以及所有直接或间接从属于它的模块的集合。

例如，在图3.2中模块A的控制域是A、B、C、D、E、F等模块的集合。

在一个设计得很好的系统中，所有受判定影响的模块应该都从属于做出判定的那个模块，最好局限于做出判定的那个模块本身及它的直属下级模块。例如，如果图3.2中模块A做出的判定只影响模块B，那么是符合这条规则的。但是，如果模块A做出的判定同时还影响模块G中的处理过程，又会有什么坏处呢？首先，这样的结构使得软件难于理解。其次，为了使得A中的判定能影响G中的处理过程，通常需要在A中给一个标记设置状态以指示判定的结果，并且应该把这个标记传递给A和G的公共上级模块M，再由M把它传给G。这个标记是控制信息而不是数据，因此将使模块间出现控制耦合。

怎样修改软件结构才能使作用域是控制域的子集呢？一个方法是把做判定的点往上移，例如，把判定从模块A中移到模块M中。另一个方法是把那些在作用域内但不在控制域内的模块移到控制域内，例如，把模块G移到模块A的下面，成为它的直属下级模块。

到底采用哪种方法改进软件结构，需要根据具体问题统筹考虑。一方面应该考虑哪种方

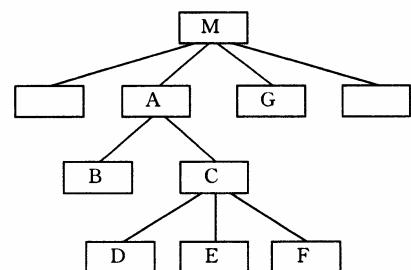


图3.2 模块的作用域和控制域

法更现实，另一方面应该使软件结构能最好地体现问题原来的结构。

3.5.5 力争降低模块接口的复杂程度

模块接口复杂是软件发生错误的一个主要原因。应该仔细设计模块接口，使得信息传递简单并且和模块的功能一致。

例如，求一元二次方程的根的模块 QUAD-ROOT (TBL , X)，其中用数组 TBL 传送方程的系数，用数组 X 回送求得的根。这种传递信息的方法不利于对这个模块的理解，不仅在维护期间容易引起混淆，在开发期间也可能发生错误。下面这种接口可能是比较简单的：

QUAD_ROOT (A , B , C , ROOT1 , ROOT2) 其中 A 、 B 、 C 是方程的系数， ROOT1 和 ROOT2 是算出的两个根。

接口复杂或不一致（即看起来传递的数据之间没有联系），是紧耦合或低内聚的征兆，应该重新分析这个模块的独立性。

3.5.6 设计单入口单出口的模块

这条启发规则告诫软件工程师不要使模块间出现内容耦合，设计出的每一个模块都应该只有一个入口一个出口。当控制流从顶部进入模块并且从底部退出来时，软件是比较容易理解的，因此也是比较容易维护的。

3.5.7 模块功能应该可以预测

模块的功能应该能够预测，但也要防止模块功能过分局限。

如果一个模块可以当做一个黑盒子，也就是说，只要输入的数据相同就产生同样的输出，这个模块的功能就是可以预测的。带有内部“存储器”的模块的功能可能是不可预测的，因为它的输出可能取决于内部存储器（例如某个标记）的状态。由于内部存储器对于上级模块而言是不可见的，所以这样的模块既不易理解又难于测试和维护。

如果一个模块只完成一个单独的子功能，则呈现高内聚；但是，如果一个模块任意限制局部数据结构的大小，过分限制在控制流中可以做出的选择或者外部接口的模式，那么这种模块的功能就过分局限，使用范围也就过分狭窄了。在使用过程中将不可避免地需要修改功能过分局限的模块，以提高模块的灵活性，扩大它的使用范围；但是，在使用现场修改软件的代价是很高的。

以上列出的启发式规则多数是经验规律，对改进设计，提高软件质量，往往有重要的参考价值；但是，它们既不是设计的目标也不是设计时应该普遍遵循的原理。

3.6 描绘软件结构的图形工具

3.6.1 层次图和 HIPO 图

通常使用层次图描绘软件的层次结构。在图 3.2 中已经非正式地使用了层次图。在层次图中一个矩形框代表一个模块，框间的连线表示调用关系（位于上方的矩形框所代表的模块调

用位于下方的矩形框所代表的模块)。图3.3是层次图的一个例子,最顶层的矩形框代表正文加工系统的主控模块,它调用下层模块以完成正文加工的全部功能;第二层的每个模块控制完成正文加工的一个主要功能,例如,“编辑”模块通过调用它的下属模块,可以完成六种编辑功能中的任何一种。在自顶向下逐步求精设计软件的过程中,使用层次图很方便。

HIPPO图是美国IBM公司发明的“层次图加输入/处理/输出图”的英文缩写。为了使

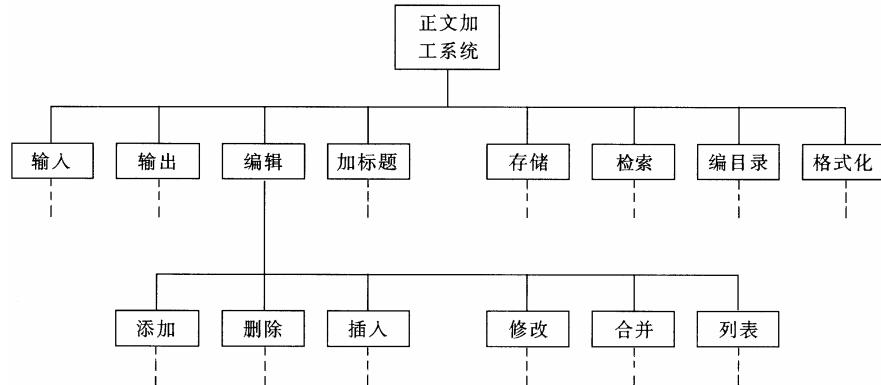


图3.3 正文加工系统的层次图

HIPPO图具有可追踪性,在H图(即层次图)里除了顶层的方框之外,每个方框都加了编号。编号方法与本书第3章3.5.2节中介绍的数据流图的编号方法相同,例如,把图3.3加了编号之后得到图3.4。

和H图中的每个方框相对应,应该有一张IPO图(或表)描绘这个方框代表的模块的处理过程。

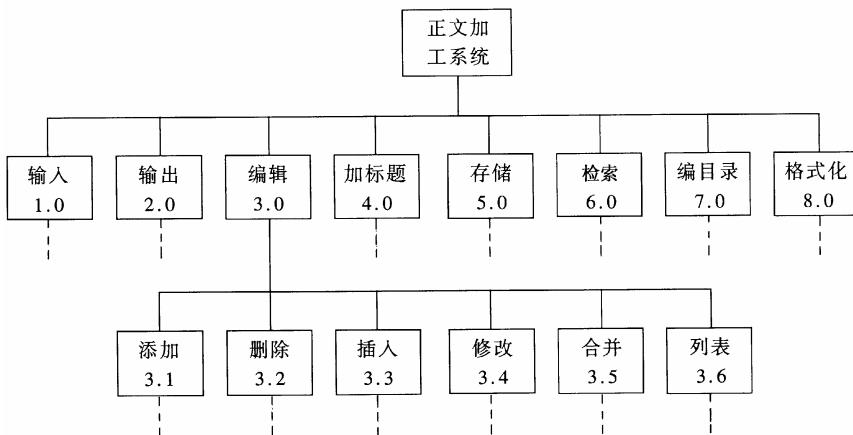


图3.4 正文加工系统的H图

3.6.2 结构图

Yourdon提出的结构图是进行软件结构设计的另一个有力工具。结构图和层次图类似,也是描绘软件结构的图形工具,图中一个方框代表一个模块,框内注明模块的名字或主要功能;方框之间的箭头(或直线)表示模块的调用关系。因为按照惯例总是图中位于上方的方框代

表的模块调用下方的模块，即使不用箭头也不会产生二义性，为了简单起见，可以只用直线而不用箭头表示模块间的调用关系。

在结构图中通常还用带注释的箭头表示模块调用过程中来回传递的信息。如果希望进一步标明传递的信息是数据还是控制信息，则可以利用注释箭头尾部的形状来区分：尾部是空心圆表示传递的是数据，实心圆表示传递的是控制信息。图 3.5 是结构图的一个例子，描绘了产生最佳解的一般结构。

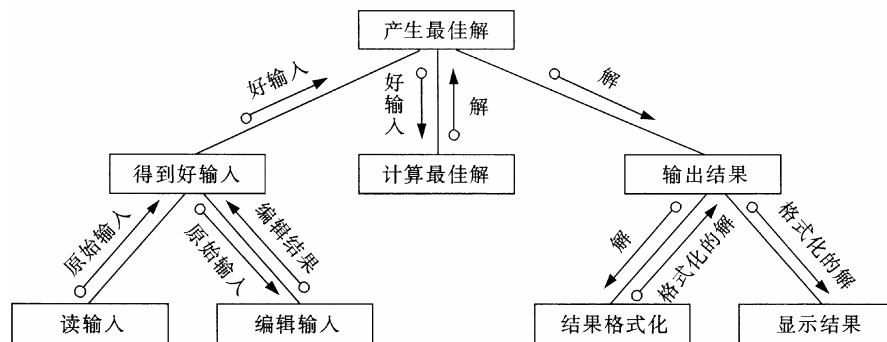


图 3.5 结构图的一个例子

以上介绍的是结构图的基本符号，也就是最经常使用的符号。此外还有一些附加的符号，可以表示模块的选择调用或循环调用。图 3.6 表示当模块 M 中某个判定为真时调用模块 A，为假时调用模块 B。图 3.7 表示模块 M 循环调用模块 A、B 和 C。

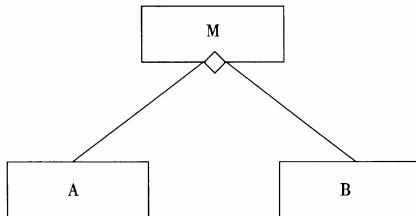


图 3.6 判定为真时调用 A，为假时调用 B

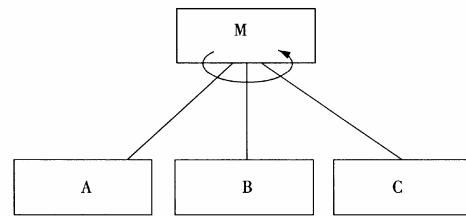


图 3.7 模块 M 循环调用模块 A、B、C

注意，层次图和结构图并不严格表示模块的调用次序。虽然多数人习惯于按调用次序从左到右画模块，但并没有这种规定，出于其他方面的考虑（例如为了减少交叉线），也完全可以不按这种次序画。此外，层次图和结构图并不指明什么时候调用下层模块。通常上层模块中除了调用下层模块的语句之外还有其他语句，究竟是先执行调用下层模块的语句还是先执行其他语句，在图中丝毫没有指明。事实上，层次图和结构图只表明一个模块调用哪些模块，至于模块内还有没有其他成分则完全没有表示。

通常用层次图作为描绘软件结构的文档。结构图作为文档并不很合适，因为图上包含的信息太多有时反而降低了清晰程度。但是，利用 IPO 图或数据字典中的信息得到模块调用时传递的信息，从而由层次图导出结构图的过程，却可以作为检查设计正确性和评价模块独立性的好方法。传送的每个数据元素是否都是完成模块功能所必须的？反之，完成模块功能必

须的每个数据元素是否都传送来了？所有数据元素是否都只和单一的功能有关？如果发现结构图上模块间的联系不容易解释，则应该考虑是否设计上有问题。

3.7 面向数据流的设计方法

面向数据流的设计方法的目标是给出设计软件结构的一个系统化的途径。

在软件工程的需求分析阶段，信息流是一个关键考虑，通常用数据流图描绘信息在系统中加工和流动的情况。面向数据流的设计方法定义了一些不同的“映射”，利用这些映射可以把数据流图转换成软件结构。因为任何软件系统都可以用数据流图表示，所以面向数据流的设计方法理论上可以设计任何软件的结构。通常所说的结构化设计方法（简称SD方法），也就是基于数据流的设计方法。

3.7.1 概念

面向数据流的设计方法把信息流映射成软件结构，信息流的类型决定了映射的方法。信息流有下述两种类型。

1. 变换流

根据基本系统模型，信息通常以“外部世界”的形式进入软件系统，经过处理以后再以“外部世界”的形式离开系统。

如图3.8所示，信息沿输入通路进入系统，同时由外部形式变换成内部形式，进入系统的信息通过变换中心，经加工处理以后再沿输出通路变换成外部形式离开软件系统。当数据流图具有这些特征时，这种信息流就叫作变换流。

2. 事务流

基本系统模型意味着变换流，因此，原则上所有信息流都可以归结为这一类。但是，当数据流图具有和图3.9类似的形状时，这种数据流是“以事务为中心的”，也就是说，数据沿输入通路到达一个处理T，这个处理根据输入数据的类型在若干个动作序列中选出一个来执行。这类数据流应该划为一类特殊的数据流，称为事务流。图3.9中的处理T称为事务中心，它完成下述任务：

（1）接收输入数据（输入数据又称为事务）；

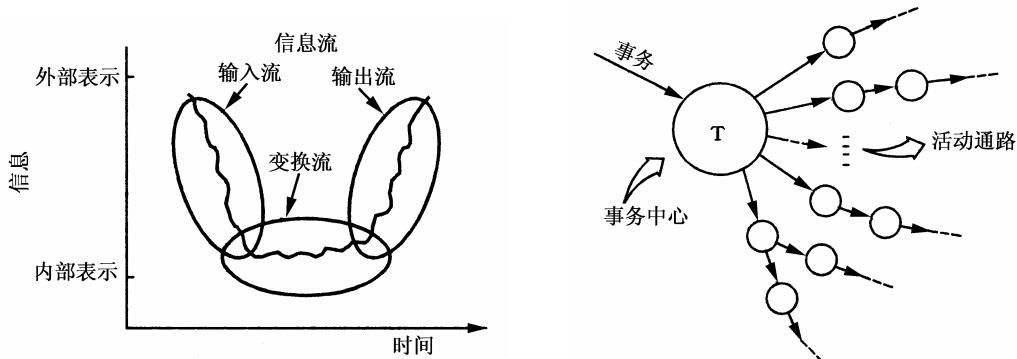


图 3.9 事务流

- (2) 分析每个事务以确定它的类型；
- (3) 根据事务类型选取一条活动通路。

3. 设计过程

图 3.10 说明了使用面向数据流方法逐步设计的过程。

应该注意，任何设计过程都不是机械地一成不变的，设计首先需要人的判断力和创造精神，这往往会凌驾于方法的规则之上。

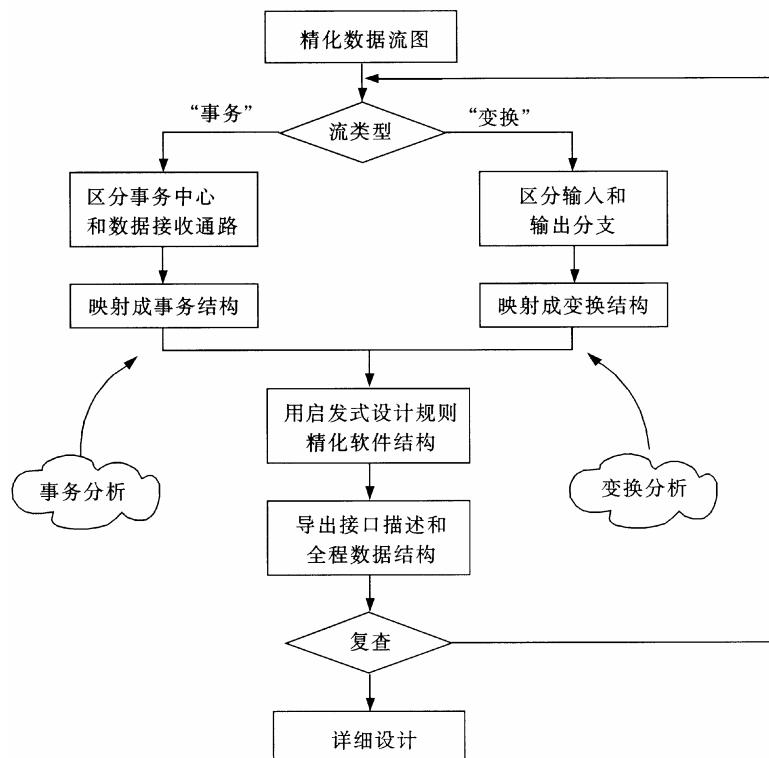


图 3.10 面向数据流方法的设计过程

3.7.2 变换分析

变换分析是一系列设计步骤的总称，经过这些步骤把具有变换流特点的数据流图按预先确定的模式映射成软件结构。下面通过一个例子说明变换分析的方法。

1. 例子

我们已经进入“智能”产品时代。在这类产品中把软件做在只读存储器中，成为设备的一部分，从而使设备具有某些“智能”。因此，这类产品的设计都包含软件开发的任务。作为面向数据流的设计方法中变换分析的例子，考虑汽车数字仪表板的设计。

假设的仪表板将完成下述功能：

- ? 通过模-数转换实现传感器和微处理机接口；
- ? 在发光二极管面板上显示数据；
- ? 指示每小时英里数 (mile / h)，行驶的里程，每加仑油行驶的英里数 (mile / Gal) 等等；

? 指示加速或减速；

? 超速警告：如果车速超过 55mile / h，则发出超速警告铃声。

在软件需求分析阶段应该对上述每条要求以及系统的其他特点进行全面的分析评价，建立起必要的文档资料，特别是数据流图。

2. 设计步骤

第1步 复查基本系统模型。

复查的目的是确保系统的输入数据和输出数据符合实际。

第2步 复查并精化数据流图。

应该对需求分析阶段得出的数据流图认真复查，并且在必要时进行精化。不仅要确保数据流图给出了目标系统的正确的逻辑模型，而且应该使数据流图中每个处理都代表一个规模适中相对独立的子功能。

假设在需求分析阶段产生的数字仪表板系统的数据流图如图 3.11 所示。

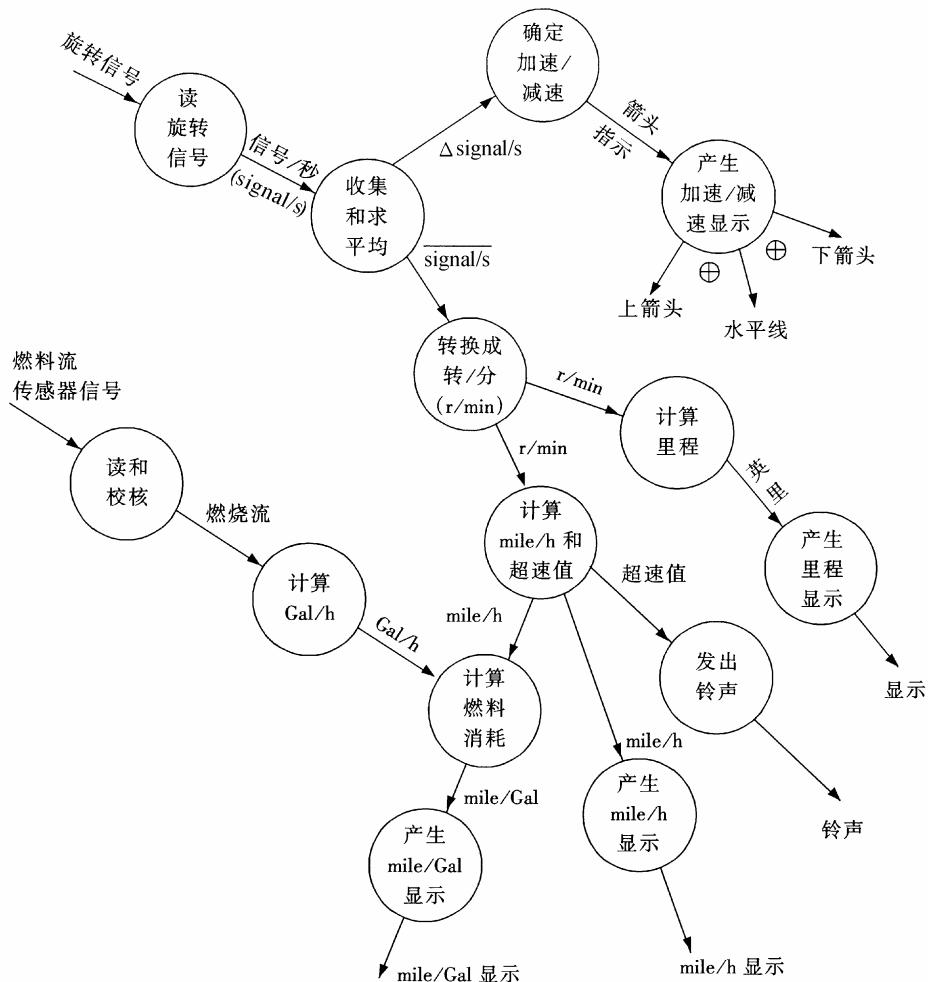


图 3.11 数字仪表板系统的数据流图

这个数据流图对于软件结构设计的“第一次分割”而言已经足够详细了，因此不需要精细化就可以进行下一个设计步骤。

第3步 分析确定数据流的类型。

一般地说，一个系统中的所有信息流都可以认为是变换流，但是，当遇到有明显事务特性的信息流时，建议采用事务分析方法进行设计。在这一步设计人员应该根据数据流图中占优势的属性，确定数据流的全局特性。此外还应该把具有和全局特性不同的特点的局部区域孤立出来，以后可以按照这些子数据流的特点精化根据全局特性得出的软件结构。

从图 3.11 可以看出，数据沿着两条输入通路进入系统，然后沿着五条通路离开，没有明显的事务中心。因此可以认为这个信息流具有变换流的总特征。

第4步 确定输入流和输出流的边界，从而孤立出变换中心。

输入流和输出流的边界和对它们的解释有关，也就是说，不同设计人员可能会在流内选取稍微不同的点作为边界的位置。当然在确定边界时应该仔细认真，但是把边界沿着数据流通路移动一个处理框的距离，通常对最后的软件结构只有很小的影响。

对于汽车数字仪表板的例子，设计人员确定的流的边界如图 3.12 所示。

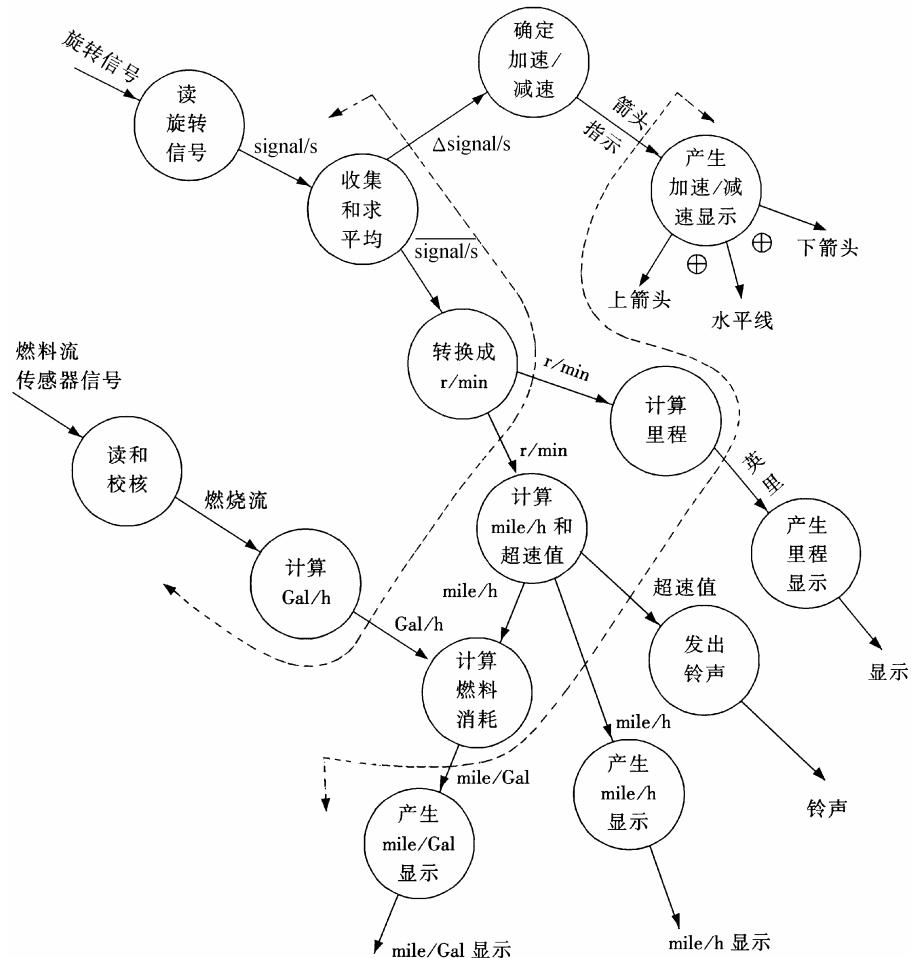


图 3.12 具有边界的的数据流图

第5步 完成“第一级分解”。

软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。

对于变换流的情况，数据流图被映射成一个特殊的软件结构，这个结构控制输入、变换和输出等信息处理过程。图3.13说明了第一级分解的方法。位于软件结构最顶层的控制模块Cm协调下述从属的控制功能：

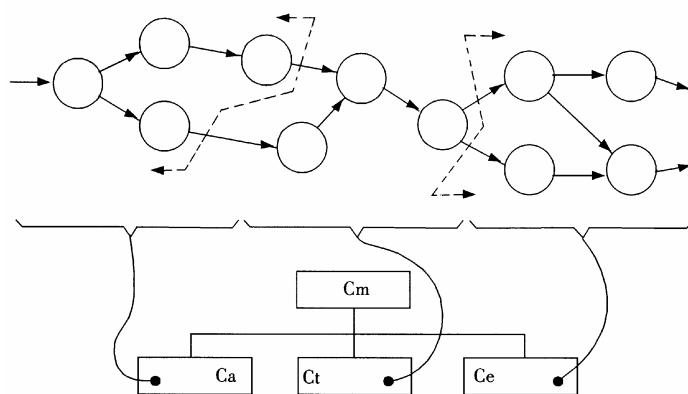


图3.13 第一级分解的方法

- ? 输入信息处理控制模块Ca，协调对所有输入数据的接收；
- ? 变换中心控制模块Ct，管理对内部形式的数据的所有操作；
- ? 输出信息处理控制模块Ce，协调输出信息的产生过程。

虽然图3.13意味着一个三叉的控制结构，但是，对一个大型系统中的复杂数据流可以用两个或多个模块完成上述一个模块的控制功能。应该在能够完成控制功能并且保持好的耦合和内聚特性的前提下，尽量使第一级控制中的模块数目取最小值。

对于数字仪表板的例子，第一级分解得出的结构如图3.14所示。每个控制模块的名字表明了为它所控制的那些模块的功能。

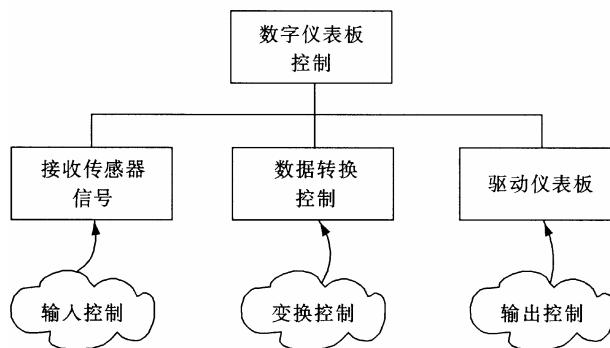


图3.14 数字仪表板系统的第一级分解

第6步 完成“第二级分解”。

所谓第二级分解就是把数据流图中的每个处理映射成软件结构中一个适当的模块。完成

第二级分解的方法是，从变换中心的边界开始沿着输入通路向外移动，把输入通路中每个处理映射成软件结构中 C_a 控制下的一个低层模块；然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受模块 C_e 控制的一个低层模块；最后把变换中心内的每个处理映射成受 C_t 控制的一个模块。图 3.15 表示进行第二级分解的普遍途径。

虽然图 3.15 描绘了在数据流图中的处理和软件结构中的模块之间的一对一的映射关系，但是，不同的映射经常出现。应该根据实际情况以及“好”设计的标准，进行实际的第二级分解。

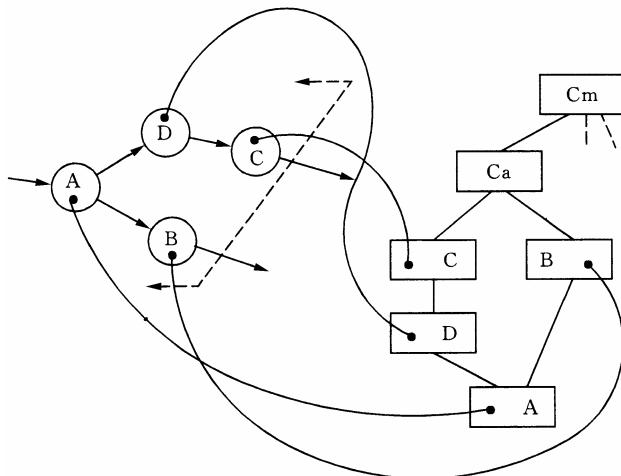


图 3.15 第二级分解的方法

对于数字仪表板系统的例子，第二级分解的结果分别用图 3.16, 3.17 和 3.18 描绘。这三张图表示对软件结构的初步设计结果。虽然图中每个模块的名字表明了它的基本功能，但是仍然应该为每个模块写一个简要说明，描述：

- ? 进出该模块的信息（接口描述）；
- ? 模块内部的信息；
- ? 过程陈述，包括主要判定点及任务等；
- ? 对约束和特殊特点的简短讨论。

当然，也可以用 2.12.3 节中讲述的 IPO 表来描述每个模块。

这些描述是第一代的设计规格说明，在这个设计时期进一步的精化和补充是经常发生的。

第 7 步 使用设计度量和启发式规则对第一次分割得到的软件结构进一步精化。

对第一次分割得到的软件结构，总可以根据模块独立原理进行精化。为了产生合理的分解，得到尽可能高的内聚、尽可能松散的耦合，最重要的是，为了得到一个易于实现、易于测试和易于维护的软件结构，应该对初步分割得到的模块进行再分解或合并。

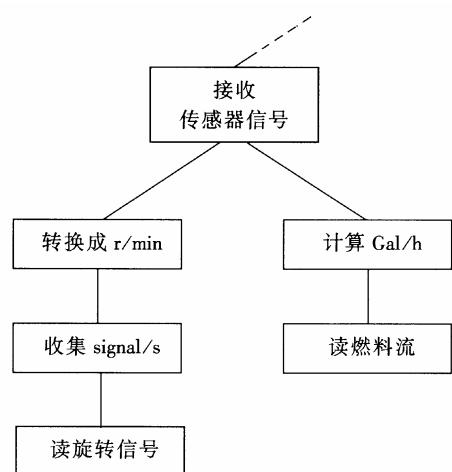


图 3.16 未经精化的输入结构

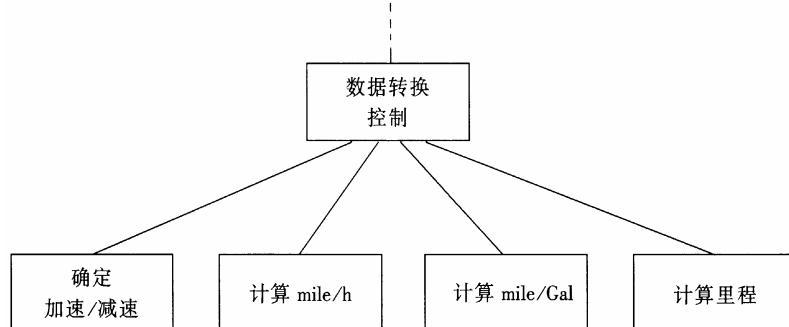


图 3.17 未经精化的变换结构

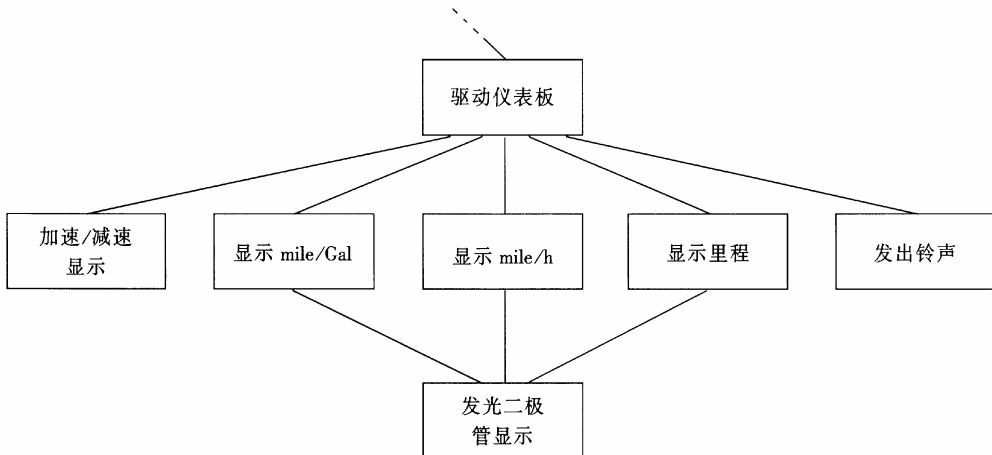


图 3.18 未经精化的输出结构

具体到数字仪表板的例子，对于从前面的设计步骤得到的软件结构，还可以做许多修改。下面是某些可能的修改：

- ? 输入结构中的模块“转换成 r / min ”和“收集 Signal / s ”可以合并；
- ? 模块“确定加速 / 减速”可以放在模块“计算 mile / h ”下面，以减少耦合；
- ? 模块“加速 / 减速显示”可以相应地放在模块“显示 mile / h ”的下面。

经过上述修改后的软件结构如图 3.19 所示。

上述七个设计步骤的目的是，开发出软件的整体表示。也就是说，一旦确定了软件结构就可以把它作为一个整体来复查，从而能够评价和精化软件结构。在这个时期进行修改只需要很少的附加工作，但是却能够对软件的质量特别是软件的可维护性产生深远的影响。

至此读者应该暂停片刻，思考上述设计途径和“写程序”的差别。如果程序代码是对软件的惟一描述，那么软件开发人员将很难站在全局的高度来评价和精化软件，而且事实上也不能做到“既见树木又见森林”。

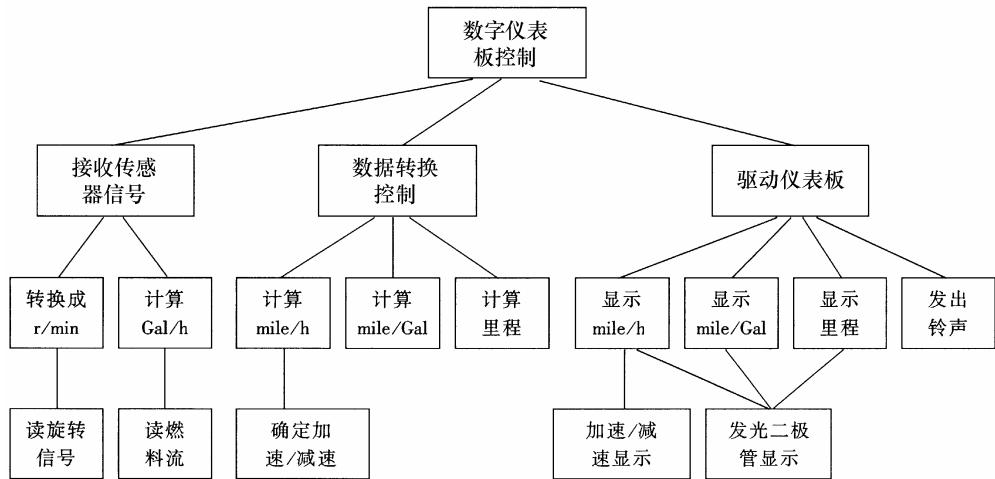


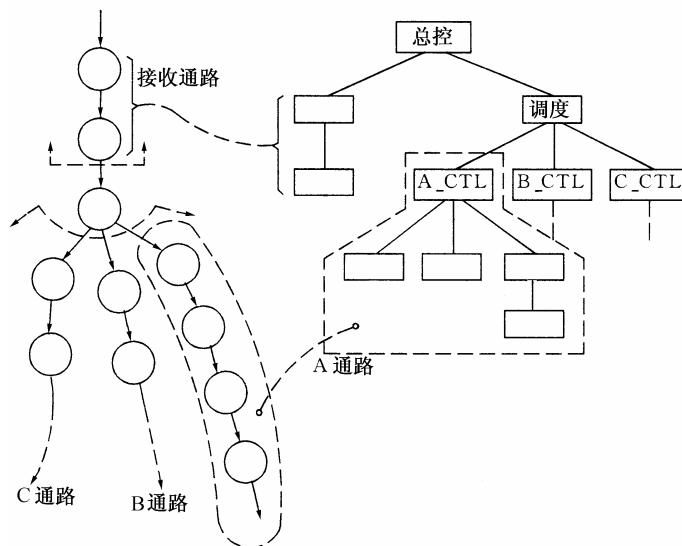
图 3.19 精化后的数字仪表板系统的软件结构

3.7.3 事务分析

虽然在任何情况下都可以使用变换分析方法设计软件结构，但是在数据流具有明显的事务特点时，也就是有一个明显的“发射中心”（事务中心）时，还是以采用事务分析方法为宜。

事务分析的设计步骤和变换分析的设计步骤大部分相同或类似，主要差别仅在于由数据流图到软件结构的映射方法不同。

由事务流映射成的软件结构包括一个接收分支和一个发送分支。映射出接收分支结构的方法和变换分析映射出输入结构的方法很相像，即从事务中心的边界开始，把沿着接收流通路的处理映射成模块。发送分支的结构包含一个调度模块，它控制下层的所有活动模块；然后把数据流图中的每个活动流通路映射成与它的流特征相对应的结构。图 3.20 说明了上



述映射过程。

对于一个大系统，常常把变换分析和事务分析应用到同一个数据流图的不同部分，由此得到的子结构形成“构件”，可以利用它们构造完整的软件结构。

一般说来，如果数据流不具有显著的事务特点，最好使用变换分析；反之，如果具有明显的事务中心，则应该采用事务分析技术。但是，机械地遵循变换分析或事务分析的映射规则，很可能得到一些不必要的控制模块，如果它们确实用处不大，那么可以而且应该把它们合并。反之，如果一个控制模块功能过分复杂，则应该分解为两个或多个控制模块，或者增加中间层次的控制模块。

3.7.4 设计优化

考虑设计优化问题时应该记住，“一个不能工作的‘最佳设计’的价值是值得怀疑的”。软件设计人员应该致力于开发能够满足所有功能和性能要求，而且按照设计原理和启发式设计规则衡量是值得接收的软件。

应该在设计的早期阶段尽量对软件结构进行精化。可以导出不同的软件结构，然后对它们进行评价和比较，力求得到“最好”的结果。这种优化的可能，是把软件结构设计和过程设计分开的真正优点之一。

注意，结构简单通常既表示设计风格优雅，又表明效率高。设计优化应该力求做到在有效的模块化的前提下使用最少量的模块，以及在能够满足信息要求的前提下使用最简单的数据结构。

对于时间是决定性因素的应用场合，可能有必要在详细设计阶段，也可能在编写程序的过程中进行优化。软件开发人员应该认识到，程序中相对说比较小的部分（典型地，10%~20%），通常占用全部处理时间的大部分（50%~80%）。用下述方法对时间起决定性作用的软件进行优化是合理的：

- (1) 在不考虑时间因素的前提下开发并精化软件结构；
- (2) 在详细设计阶段选出最耗费时间的那些模块，仔细地设计它们的处理过程（算法），以求提高效率；
- (3) 使用高级程序设计语言编写程序；
- (4) 在软件中孤立出那些大量占用处理器资源的模块；
- (5) 必要时重新设计或用依赖于机器的语言重写上述大量占用资源的模块的代码，以求提高效率。

上述优化方法遵守了一句格言：“先使它能工作，然后再使它快起来。”

3.8 人机界面设计

人机界面设计是接口设计的一个重要的组成部分。对于交互式系统来说，人机界面设计和数据设计、体系结构设计、过程设计一样重要。近年来，人机界面在系统中所占的比例越来越大，在个别系统中设计人机界面所用的工作量甚至占设计总工作量的一半以上。

人机界面的设计结果，将对用户工作时的心情和工作效率产生重要影响。人机界面设计得好，会使软件系统对用户产生吸引力，用户在使用系统的过程中会感到兴奋，这样的系统

能够激发用户的创造力，提高用户的工作效率；相反，人机界面设计得不好，用户在使用系统的过程中就会感到不方便、不习惯，甚至会产生厌烦和恼怒的情绪。

人机界面的设计质量，直接影响用户对软件产品的评价，从而影响软件产品的竞争力和使用寿命，因此，必须对人机界面设计给予足够重视。

由于对人机界面的评价，在很大程度上由人的主观因素决定，因此，使用基于原型的系统化的设计策略，是成功地设计人机界面的关键。

3.8.1 应该考虑的设计问题

在设计用户界面的过程中，设计者几乎总会遇到下述四个问题：系统响应时间、用户帮助设施、出错信息处理和命令交互。不幸的是，许多设计者直到设计过程的后期才开始考虑这些问题，这样做往往导致不必要的设计反复、项目延期完成和使用户产生挫折感。最好在设计人机界面的初期就把这些问题作为重要的设计问题来考虑，这时修改比较容易，代价也低。下面讨论这四个问题。

1. 系统响应时间

系统响应时间是许多交互式系统用户经常抱怨的问题。一般说来，系统响应时间指的是从用户完成某个控制动作（例如，按回车键或点击鼠标），到软件给出预期的响应（输出或做预期的动作）之间的这段时间。

系统响应时间有两个重要属性，分别是长度和易变性。如果系统响应时间过长，用户就会感到紧张和沮丧。但是，当用户的工作速度是由人机界面决定的时候，系统响应时间过短也不好，这会迫使用户加快操作节奏，从而可能犯错误。

易变性指系统响应时间相对于平均响应时间的偏差，在许多情况下，这是系统响应时间更重要的属性。即使系统响应时间较长，响应时间易变性低也有助于用户建立起稳定的工作节奏。例如，稳定在 1s 的响应时间比从 0.1s 到 2.5s 之间变化的响应时间要好。用户通常比较敏感，他们往往担心响应时间变化暗示系统工作出现了异常。

2. 用户帮助设施

交互式系统的用户在使用系统的过程中几乎都需要帮助，当遇到复杂问题时甚至需要查看用户手册以寻找答案。大多数现代软件都提供联机帮助设施，这使得用户可以不离开用户界面就解决自己的问题。

常见的帮助设施有集成的和附加的两类。集成的帮助设施从一开始就设计在软件里面，通常它对用户的工作内容是敏感的，因此用户可以从与刚刚完成的操作有关的主题中选择一个请求帮助。显然，这可以缩短用户获得帮助所需的时间，并能增加界面的友好性。附加的帮助设施是在系统建成后再添加到软件中的，在多数情况下，它实际上是一种查询能力有限的联机用户手册。人们普遍认为，集成的帮助设施优于附加的帮助设施。

具体设计帮助设施时，必须解决下述的一系列问题。

? 在用户与系统交互期间，是否能在任何时候都获得关于系统任何功能的帮助信息？有两种选择：提供部分功能的帮助信息和提供全部功能的帮助信息。

? 用户怎样请求帮助？有三种选择：帮助菜单，特殊功能键和 HELP 命令。

? 怎样显示帮助信息？有三种选择：在独立的窗口中，指出参考某个文档（不理想）和在屏幕固定位置显示简短提示。

- ? 用户怎样返回到正常的交互方式？有两种选择：屏幕上的返回按钮和功能键。
- ? 怎样组织帮助信息？有三种选择：平面结构（所有信息都通过关键字访问），信息的层次结构（用户可在该结构中查到更详细的信息）和超文本结构。

3. 出错信息处理

出错信息和警告信息，是出现问题时交互式系统给出的“坏消息”。出错信息设计得不好，将向用户提供无用的或误导的信息，反而增加了用户的挫折感。

一般说来，交互式系统给出的出错信息或警告信息，应该具有下述属性。

- ? 信息应该以用户可以理解的术语描述问题。
- ? 信息应该提供有助于从错误中恢复的建设性意见。
- ? 信息应该指出错误可能导致哪些负面后果（例如，破坏数据文件），以便用户检查是否出现了这些问题，并在确实出现问题时予以改正。
- ? 信息应该伴随着听觉上或视觉上的提示，也就是说，在显示信息时应该同时发出警告声，或者信息用闪烁方式显示，或者信息用明显表示出错的颜色显示。
- ? 信息不能带有指责色彩，也就是说，不能责怪用户。

当确实出现了问题的时候，有效的出错信息能够提高交互式系统的质量，减少用户的挫折感。

4. 命令交互

命令行曾经是用户和系统软件交互的最常用方式，而且也曾经广泛地用于各种应用软件中。现在，面向窗口的、点击和拾取方式的界面已经减少了用户对命令行的依赖，但是，许多高级用户仍然偏爱面向命令的交互方式。在多数情况下，用户既可以从菜单中选择软件功能也可以通过键盘命令序列调用软件功能。

在提供命令交互方式时，必须考虑下列设计问题。

- ? 是否每个菜单选项都有对应的命令？
- ? 采用何种命令形式？有三种选择：控制序列（例如，Ctrl + P），功能键和键入命令。
- ? 学习和记忆命令的难度有多大，忘记了命令怎么办？
- ? 用户是否可以定制或缩写命令？

在越来越多的应用软件中，界面设计者都提供了“命令宏机制”，使用这种机制用户可以用自己定义的名字代表一个常用的命令序列。需要使用这个命令序列时，用户无需依次键入每个命令，只需输入命令宏的名字就可以顺序执行它所代表的全部命令。

在理想的情况下，所有应用软件都有一致的命令使用方法。如果在一个应用软件中，命令 Ctrl + D 表示复制一个图形对象，而在另一个应用软件中 Ctrl + D 命令的含义是删除一个图形对象，显然会使用户感到困惑，并且往往会导致错误。

3.8.2 人机界面设计过程

用户界面设计是一个迭代的过程，也就是说，通常先创建设计模型，再用原型实现这个设计模型，并由用户试用和评估，然后根据用户的意见进行修改。为了支持这种迭代过程，各种用于界面设计和原型开发的工具应运而生。这些工具被称为用户界面工具箱或用户界面开发系统（UIDS），它们为简化窗口、菜单、设备交互、出错信息、命令及交互环境的许多其他元素的创建，提供了各种例程或对象。这些工具所提供的功能既可以用基于语言的方式也可以用基于图形的方式来实现。

一旦建立起用户界面原型，就必须对它进行评估，以确定其是否满足用户的需求。评估可以是非正式的，例如，用户即兴发表一些反馈意见；评估也可以十分正式，例如，运用统计学方法评价全体终端用户填写的调查表。

用户界面评估周期如下所述：完成初步设计后就创建第一级原型；用户试用并评估该原型，直接向设计者提出对界面功效的评价；设计者根据用户意见修改设计并实现下一级原型。上述评估过程不断进行下去，直到用户感到满意，不需要再修改界面设计时为止。

当然，也可以在创建原型之前就对用户界面设计的质量进行初步评估。如果能及早发现并改正潜在的问题，就可以减少评估周期执行的次数，从而缩短软件的开发时间。在创建了界面的设计模型之后，可以运用下述评估标准对设计进行早期复审。

? 系统及其界面的规格说明的长度和复杂程度，预示了用户学习使用该系统所需要的工作量。

? 命令或动作的数量、命令的平均参数个数或动作中单个操作的个数，预示了系统的交互时间和总体效率。

? 设计模型中给出的动作、命令和系统状态的数量，预示了用户学习使用系统时需要记忆的内容的多少。

? 界面风格、帮助设施和出错处理协议，预示了界面的复杂程度和用户接受该界面的程度。

3.8.3 界面设计指南

用户界面设计主要依靠设计者的经验。总结众多设计者的经验而得出的设计指南，有助于设计者设计出友好、高效的人机界面。本节介绍三类人机界面设计指南。

1. 一般交互指南

一般交互指南涉及信息显示、数据输入和整体系统控制，因此，这些指南是全局性的，忽略它们将承担较大风险。下面叙述一般交互指南。

? 保持一致性。为人工界面中的菜单选择、命令输入、数据显示以及众多的其他功能，使用一致的格式。

? 提供有意义的反馈。向用户提供视觉的和听觉的反馈，以保证在用户和界面之间建立双向通信。

? 在执行有较大破坏性的动作之前要求用户确认。如果用户要删除一个文件，或覆盖一些重要信息，或请求终止一个程序运行，应该给出“您是否确实要……”的信息，以请求用户确认他的命令。

? 允许取消绝大多数操作。UNDO 或 REVERSE 功能使众多终端用户避免了大量时间浪费。每个交互式应用系统都应该能方便地取消已完成的操作。

? 减少在两次操作之间必须记忆的信息量。不应该期望用户能记住一大串数字或名字，以便在下一步操作中使用它们。应该尽量减少记忆量。

? 提高对话、移动和思考的效率。应该尽量减少击键次数，设计屏幕布局时应该考虑尽量减少鼠标移动的距离，应该尽量避免出现用户问：“这是什么意思”的情况。

? 允许犯错误。系统应该保护自己不受致命错误的破坏。

? 按功能对动作分类，并据此设计屏幕布局。下拉菜单的一个主要优点就是能按动作类型组织命令。实际上，设计者应该尽力提高命令和动作组织的“内聚性”。

? 提供对工作内容敏感的帮助设施（参见 3.8.1 节）。

? 用简单动词或动词短语作为命令名。过长的命令名难于识别和记忆，也会占据过多的菜单空间。

2. 信息显示指南

如果人机界面显示的信息是不完整的，含糊的或难于理解的，则应用软件显然不能满足用户的需求。可以用多种不同方式“显示”信息：用文字、图片和声音；按位置、移动和大小；使用颜色、分辨率和省略。下面是关于信息显示的设计指南。

? 只显示与当前工作内容有关的信息。用户在获得有关系统的特定功能的信息时，不必看到与之无关的数据、菜单和图形。

? 不要用数据淹没用户，应该用便于用户迅速地吸取信息的方式来表示数据。例如，可以用图形或图表来取代巨大的表格。

? 使用一致的标记、标准的缩写和可预知的颜色。显示的含义应该非常明确，用户不必参照其他信息源就能理解。

? 允许用户保持可视化的语境。如果对图形显示进行缩放，原始的图像应该一直显示着（以缩小的形式放在显示屏的一角），以使用户知道当前观察的图像部分在原图中所处的相对位置。

? 产生有意义的出错信息（参见3.8.1节）。

? 使用大小写、缩进和文本分组以帮助理解。人机界面显示的信息大部分是文字，文字的布局和形式对用户从中吸取信息的难易程度有很大影响。

? 使用窗口分隔不同类型的信息。利用窗口用户能够方便地“保存”多种不同类型的信息。

? 使用“模拟”显示方式表示信息，以使信息更容易被用户吸取。例如，显示炼油厂储油罐的压力时，如果使用简单的数字表示压力，则不易引起用户注意。但是，如果用类似温度计的形式来表示压力，用垂直移动和颜色变化来指示危险的压力状况，就能引起用户的警觉，因为这样做为用户提供了绝对值和相对值两方面的信息。

? 高效率地使用显示屏。当使用多窗口时，应该有足够的空间使得每个窗口至少都能显示出一部分。此外，屏幕大小应该选得和应用系统的类型相配套（这实际上是一个系统工程问题）。

3. 数据输入指南

用户的大部分时间用在选择命令、键入数据和向系统提供输入。在许多应用系统中，键盘仍然是主要的输入介质，但是，鼠标、数字化仪和语音识别系统正迅速地成为重要的输入手段。下面是关于数据输入的设计指南。

? 尽量减少用户的输入动作。最重要的是减少击键次数，这可以用下列方法实现：用鼠标从预定义的一组输入中选一个；用“滑动标尺”在给定的值域中指定输入值；利用宏把一次击键转变成更复杂的输入数据集合。

? 保持信息显示和数据输入之间的一致性。显示的视觉特征（例如，文字大小、颜色和位置）应该与输入域一致。

? 允许用户自定义输入。专家级的用户可能希望定义自己专用的命令或略去某些类型的警告信息和动作确认，人机界面应该允许用户这样做。

? 交互应该是灵活的，并且可调整成用户最喜欢的输入方式。用户类型与喜欢的输入方

式有关，秘书可能非常喜欢键盘输入，而经理可能更喜欢使用鼠标之类的点击设备。

? 使在当前动作语境中不适用的命令不起作用。这可使用户不去做那些肯定会导致错误的动作。

? 让用户控制交互流。用户应该能够跳过不必要的动作，改变所需做的动作的顺序（在应用环境允许的前提下），以及在不退出程序的情况下从错误状态中恢复正常。

? 对所有输入动作都提供帮助（参见 3.8.1 节）。

? 消除冗余的输入。除非可能发生误解，否则不要要求用户指定工程输入的单位；不要要求用户在整钱数后面键入.00；尽可能提供缺省值；绝对不要求用户提供程序可以自动获得或计算出来的信息。

3.9 过程设计

过程设计应该在数据设计、体系结构设计和接口设计完成之后进行，它是详细设计阶段应该完成的主要任务。

过程设计的任务还不是具体地编写程序，而是要设计出程序的“蓝图”，以后程序员将根据这个蓝图写出实际的程序代码。因此，过程设计的结果基本上决定了最终的程序代码的质量。考虑程序代码的质量时必须注意，程序的“读者”有两个，那就是计算机和人。在软件的生命周期中，设计测试方案，诊断程序错误，修改和改进程序等都必须首先读懂程序。实际上对于长期使用的软件系统而言，人读程序的时间可能比写程序的时间还要长得多。因此，衡量程序的质量不仅要看它的逻辑是否正确，性能是否满足要求，更主要的是要看它是否容易阅读和理解。过程设计的目标不仅仅是逻辑上正确地实现每个模块的功能，更重要的是设计出的处理过程应该尽可能简明易懂。结构程序设计技术是实现上述目标的关键技术，因此是过程设计的逻辑基础。

结构程序设计的概念最早由 E.W.Dijkstra 提出。1965 年他在一次会议上指出：“可以从高级语言中取消 GO TO 语句”，“程序的质量与程序中所包含的 GO TO 语句的数量成反比”。1966 年 Böhm 和 Jacopini 证明了，只用三种基本的控制结构就能实现任何单入口单出口的程序。这三种基本的控制结构是“顺序”、“选择”和“循环”，它们的流程图分别为图 3.21 (a)，3.21 (b) 和 3.21 (c)。

实际上用顺序结构和循环结构（又称 DO WHILE 结构）完全可以实现选择结构（又称 IF THEN ELSE 结构），因此，理论上最基本的控制结构只有两种。

Böhm 和 Jacopini 的证明给结构程序设计

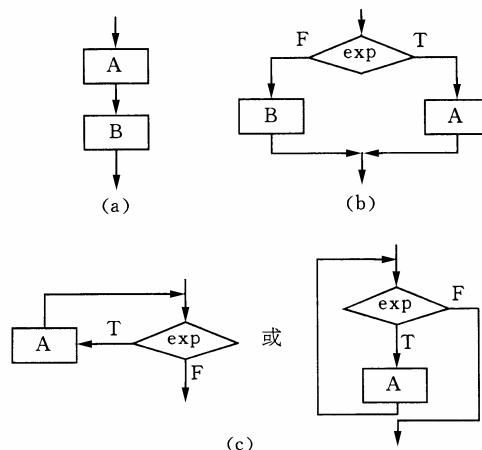


图 3.21 三种基本的控制结构

(a) 顺序结构，先执行 A 再执行 B；

(b) IF_THEN_ELSE 型选择（分支）结构；

(c) DO WHILE 型循环结构

技术奠定了理论基础。

1968年Dijkstra再次建议从一切高级语言中取消GO TO语句，只使用三种基本控制结构写程序。他的建议引起了激烈争论，经过讨论人们认识到，不是简单地去掉GO TO语句的问题，而是要创立一种新的程序设计思想、方法和风格，以显著地提高软件生产率和降低软件维护代价。

1972年IBM公司的Mills进一步提出，程序应该只有一个入口和一个出口，从而补充了结构程序设计的规则。

1971年IBM公司在纽约时报信息库管理系统的成功地使用了结构程序设计技术，随后在美国宇航局空间实验室飞行模拟系统的设计中，结构程序设计技术再次获得圆满成功。这两个系统都相当庞大，前者包含8万3千行高级语言源程序，后者包含40万行源程序，而且在设计过程中用户需求又曾有过很多改变，然而两个系统的开发工作都按时并且高质量地完成了。这表明，软件生产率比以前提高了一倍，结构程序设计技术成功地经受了实践的检验。

结构程序设计的经典定义如下所述：

如果一个程序的代码块仅仅通过顺序、选择和循环这三种控制结构进行连接，并且每个代码块只有一个入口和一个出口，则称这个程序是结构化的。

这个经典定义过于狭隘了，结构程序设计本质上并不是无GO TO语句的编程方法，而是一种使程序代码容易阅读、容易理解的编程方法。在大多数情况下，无GO TO的代码确实是容易阅读、容易理解的代码，但是，在某些情况下，为了达到容易阅读和容易理解的目的，反而需要使用GO TO语句。例如，当出现了错误条件时，重要的是在数据库崩溃或栈溢出之前，尽可能快地从当前程序退到一个出错处理程序，实现这个目标的最好方法就是使用前向GO TO语句（或与之等效的专用语句），机械地使用三种基本控制结构实现这个目标反而会使程序晦涩难懂。因此，下述的结构程序设计的定义可能更全面一些：

结构程序设计是尽可能少用GO TO语句的程序设计方法。最好仅在检测出错误时才使用GO TO语句，而且应该总是使用前向GO TO语句。

虽然从理论上只用上述三种基本控制结构就可以实现任何单入口单出口的程序，但是为了实际使用方便起见，常常还允许使用DO_UNTIL和DO_CASE两种控制结构，它们的流程图分别如图3.22(a)和图3.22(b)所示。

有时需要立即从循环（甚至嵌套的循环）中转移出来，如果允许使用LEAVE（或BREAK）结构，则不仅方便而且会使效率提高很多。LEAVE或BREAK结构实质上是受限制的GO TO语句，用于转移到循环结构后面的语句。

如果只允许使用顺序、IF_THEN_ELSE型分支和DO_WHILE型循环这三种基本控制结构，则称为经典的结构程序设计；如果除了上述三种基本控制结构之外，还允许使用DO_CASE

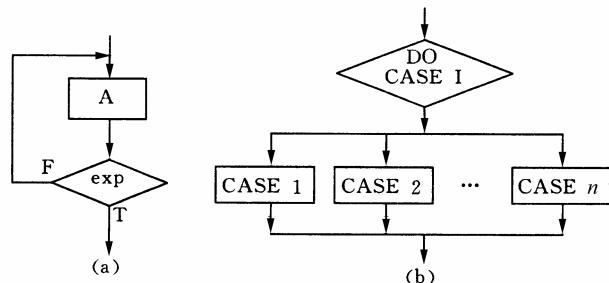


图3.22 其他常用的控制结构
(a) DO_UNTIL型循环结构；(b) 多分支结构

型多分支结构和 DO_UNTIL 型循环结构，则称为扩展的结构程序设计；如果再加上允许使用 LEAVE（或 BREAK）结构，则称为修正的结构程序设计。

3.10 过程设计的工具

描述程序处理过程的工具称为过程设计的工具，它们可以分为图形、表格和语言三类。不论是哪类工具，对它们的基本要求都是能提供对设计的无歧义的描述，也就是应该能指明控制流程、处理功能、数据组织以及其他方面的实现细节，从而在编码阶段能把对设计的描述直接翻译成程序代码。此外，这类工具应该尽可能的形象直观，应该易学、易懂。

3.10.1 程序流程图

程序流程图又称为程序框图，它是历史最悠久使用最广泛的描述过程设计的方法，然而它也是用得最混乱的一种方法。

在 3.9 节中已经用程序流程图描绘了一些常用的控制结构，相信读者对程序流程图中使用的基本符号已经有了一些了解。图 3.23 中列出了程序流程图中使用的各种符号。

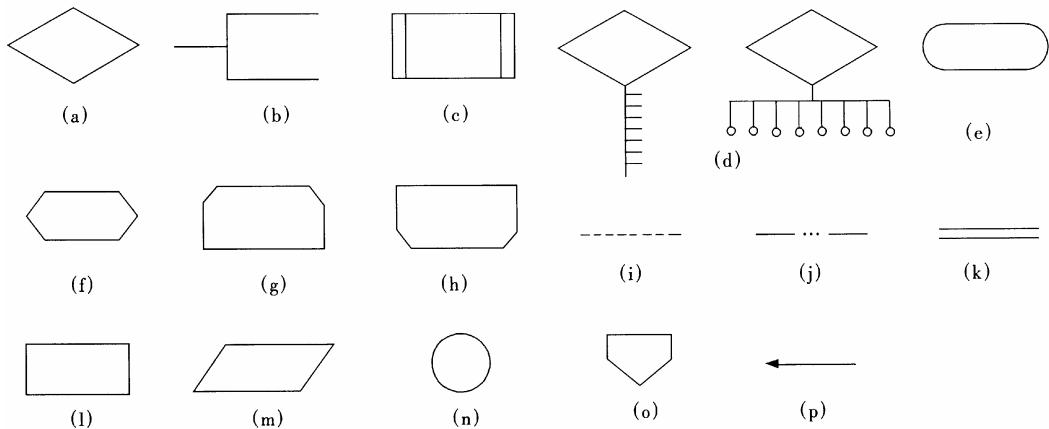


图 3.23 程序流程图中使用的符号

- (a) 选择（分支）；(b) 注释；(c) 预先定义的处理；(d) 多分支；(e) 开始或停止；
- (f) 准备；(g) 循环上界限；(h) 循环下界限；(i) 虚线；(j) 省略符；(k) 并行方式；
- (l) 处理；(m) 输入 / 输出；(n) 连接；(o) 换页连接；(p) 控制流

从 20 世纪 40 年代末到 70 年代中期，程序流程图一直是过程设计的主要工具。它的主要优点是对控制流程的描绘很直观，便于初学者掌握。由于程序流程图历史悠久，为最广泛的人所熟悉，尽管它有种种缺点，许多人建议停止使用它，但至今仍在广泛使用着。不过总的趋势是越来越多的人不再使用程序流程图了。

程序流程图的主要缺点如下。

? 程序流程图本质上不是逐步求精的好工具，它诱使程序员过早地考虑程序的控制流程，

而不去考虑程序的全局结构。

? 程序流程图中用箭头代表控制流，因此程序员不受任何约束，可以完全不顾结构程序设计的精神，随意转移控制。

? 程序流程图不易表示数据结构。

应该指出，详细的微观程序流程图——每个符号对应于源程序的一行代码，对于提高大型系统的可理解性作用甚微。

3.10.2 盒图

出于要有一种不允许违背结构程序设计精神的图形工具的考虑，Nassi 和 Shneiderman 发明了盒图，又称为 N-S 图。它有下述特点：

- (1) 功能域（即，一个特定控制结构的作用域）明确，可以从盒图上一眼就看出来。
- (2) 不可能任意转移控制。
- (3) 很容易确定局部和全程数据的作用域。
- (4) 很容易表现嵌套关系，也可以表示模块的层次结构。

图 3.24 给出了结构化控制结构的盒图表示，也给出了调用子程序的盒图表示方法。

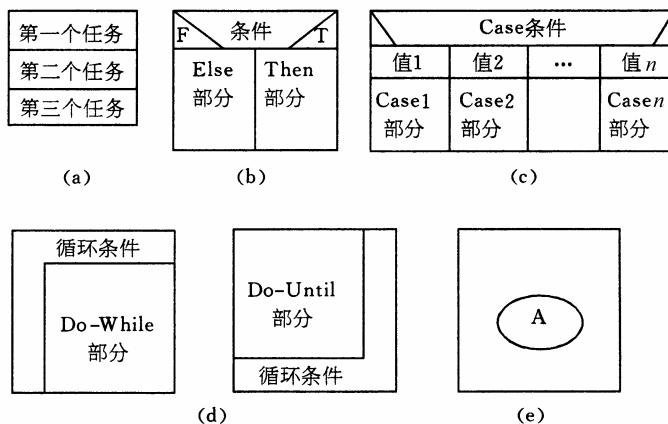


图 3.24 盒图的基本符号

(a) 顺序；(b) IF_THEN_ELSE 型分支；(c) CASE 型多分支；(d) 循环；(e) 调用子程序 A

虽然用惯了程序流程图的人初次接触盒图时可能感到不太习惯，但是它一点也不比程序流程图复杂。图 3.25 是盒图的一个例子，图 3.25 (a) 是一张程序流程图，图 3.25 (b) 是等效的盒图。

盒图没有箭头，因此不能够随意转移控制。坚持使用盒图作为过程设计的工具，可以使程序员逐步养成用结构化的方式思考问题、解决问题的习惯。

3.10.3 PAD 图

PAD 是问题分析图（Problem Analysis Diagram）的英文缩写，自 1973 年由日本日立公司发明以后，已得到一定程度的推广。它用二维树形结构的图来表示程序的控制流，将这种图翻

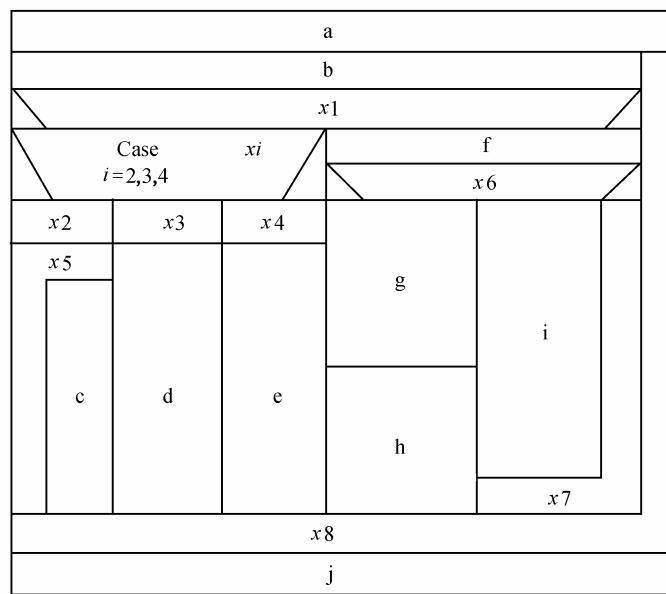
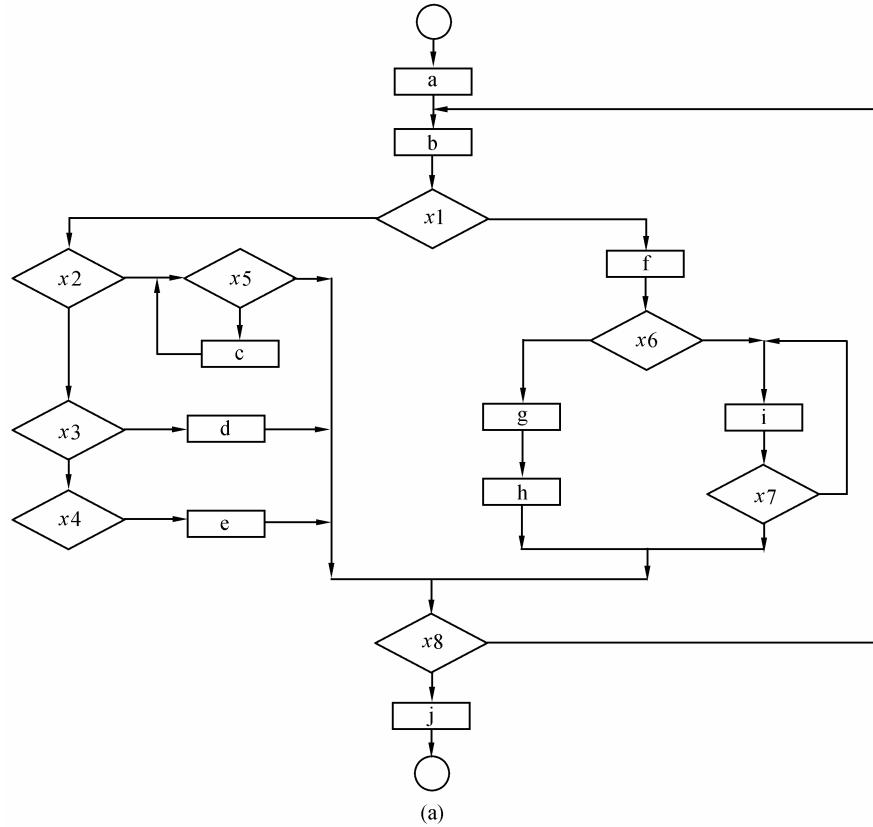


图 3.25 程序流程图和等效的盒图

(a) 流程图 ;(b) 等效的盒图

译成程序代码比较容易。图3.26给出PAD图的基本符号。

PAD图的主要优点如下：

(1) 使用表示结构化控制结构的PAD符号所设计出来的程序必然是结构化程序。

(2) PAD图所描绘的程序结构十分清晰。图中最左面的竖线是程序的主线，即第一层结构。随着程序层次的增加，PAD图逐渐向右延伸，每增加一个层次，图形向右扩展一条竖线。PAD图中竖线的总条数就是程序的层数。

(3) 用PAD图表现程序逻辑，易读、易懂、易记。PAD图是二维树型结构的图形，程序从图中最左竖线上端的节点开始执行，自上而下，从左向右顺序执行，遍历所有节点。

(4) 容易将PAD图转换成高级语言源程序，这种转换可用软件工具自动完成，从而可省去人工编码的工作，有利于提高软件可靠性和软件生产率。

(5) 既可用于表示程序逻辑，也可用于描绘数据结构。

(6) PAD图的符号支持自顶向下、逐步求精方法的使用。开始时设计者可以定义一个抽象的程序，随着设计工作的深入而使用def符号逐步增加细节，直至完成详细设计，如图3.27所示。

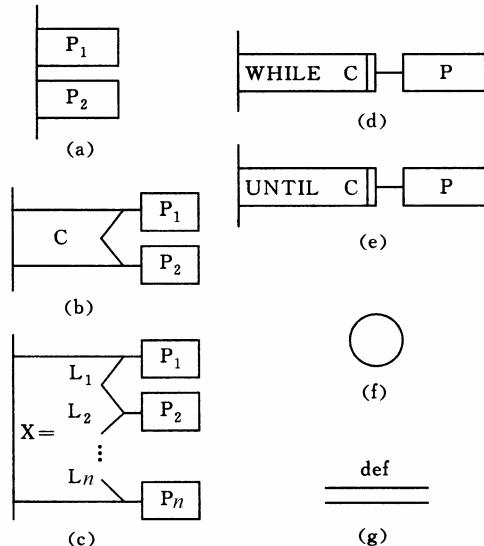


图3.26 PAD图的基本符号

(a) 顺序 (先执行 P₁ 后执行 P₂)；(b) 选择 (IF C THEN P₁ ELSE P₂)；

(c) CASE型多分支；(d) WHILE型循环 (WHILE C DO P)；

(e) UNTIL型循环 (REPEAT P UNTIL C)；(f) 语句标号；(g) 定义

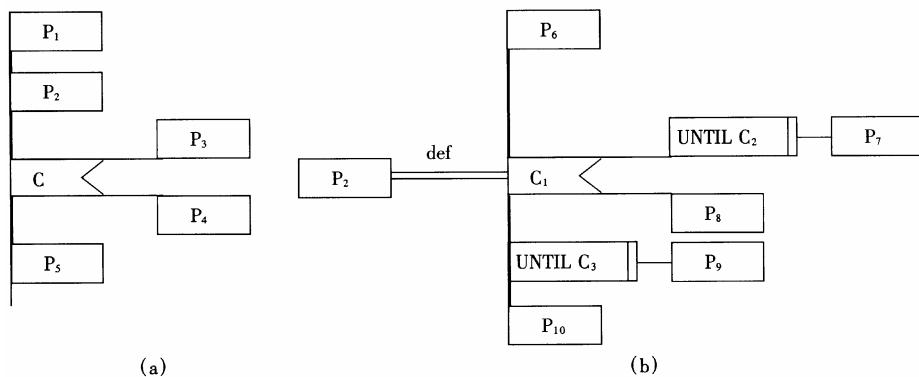


图3.27 使用PAD图提供的定义功能来逐步求精的例子

(a) 初始的PAD图；(b) 使用def符号细化处理框P₂

PAD图是面向高级程序设计语言的，为FORTRAN, COBOL和PASCAL等每种常用的高级程序设计语言都提供了一整套相应的图形符号。由于每种控制语句都有一个图形符号与之

对应，显然将 PAD 图转换成与之对应的高级语言程序比较容易。

3.10.4 判定表

当算法中包含多重嵌套的条件选择时，用程序流程图、盒图、PAD 图或后面即将介绍的过程设计语言（PDL）都不易清楚地描述。然而判定表却能够清晰地表示复杂的条件组合与应做的动作之间的对应关系。

一张判定表由四部分组成，左上部列出所有条件，左下部是所有可能做的动作，右上部是表示各种条件组合的一个矩阵，右下部是和每种条件组合相对应的动作。判定表右半部的每一列实质上是一条规则，规定了与特定的条件组合相对应的动作。

下面以行李托运费的算法为例说明判定表的组织方法。假设某航空公司规定，乘客可以免费托运重量不超过 30kg 的行李。当行李重量超过 30kg 时，对头等舱的国内乘客超重部分每 kg 收费 4 元，对其他舱的国内乘客超重部分每 kg 收费 6 元，对外国乘客超重部分每 kg 收费比国内乘客多一倍，对残疾乘客超重部分每 kg 收费比正常乘客少一半。用判定表可以清楚地表示与上述每种条件组合相对应的动作（算法），如表 3.1 所示。

表 3.1 用判定表表示计算行李费的算法

	1	2	3	4	5	6	7	8	9
国内乘客		T	T	T	T	F	F	F	F
头等舱		T	F	T	F	T	F	T	F
残疾乘客		F	F	T	T	F	F	T	T
行李重量 $W - 30$	T	F	F	F	F	F	F	F	F
免费	x								
$(W - 30) \times 2$				x					
$(W - 30) \times 3$					x				
$(W - 30) \times 4$		x						x	
$(W - 30) \times 6$			x						x
$(W - 30) \times 8$						x			
$(W - 30) \times 12$							x		

在表的右上部分中“T”表示它左边那个条件成立，“F”表示条件不成立，空白表示这个条件成立与否并不影响对动作的选择。判定表右下部分中画“×”表示做它左边的那项动作，空白表示不做这项动作。从表3.1可以看出，只要行李重量不超过30kg，不论这位乘客持有

何种机票，是中国人还是外国人，是残疾人还是正常人，一律免收行李费，这就是表右部第一列（规则1）表示的内容。当行李重量超过30kg时，根据乘客机票的等级、国籍、是否残疾人而使用不同算法计算行李费，这就是规则2到规则9表示的内容。

从上面这个例子可以看出，判定表能够简洁而又无歧义地描述处理规则。当把判定表和布尔代数或卡诺图结合起来使用时，可以对判定表进行校验或化简。但是，判定表并不适于作为一种通用的设计工具，没有一种简单的方法使它能同时清晰地表示顺序和重复等处理特性。

3.10.5 判定树

判定表虽然能清晰地表示复杂的条件组合与应做的动作之间的对应关系，但其含义却不是一眼就能看出来的，初次接触这种工具的人要理解它需要有一个简短的学习过程。此外，当数据元素的值多于两个时（例如，3.10.4例子中假设对机票需细分为头等舱、二等舱和经济舱等多种级别时），判定表的简洁程度也将下降。

判定树是判定表的变种，也能清晰地表示复杂的条件组合与应做的动作之间的对应关系。判定树的优点在于，它的形式简单到不需任何说明，一眼就可以看出其含义，因此易于掌握和使用。多年来判定树一直受到人们的重视，是一种比较常用的系统分析和设计的工具。图3.28是和表3.1等价的判定树。从图3.28可以看出，虽然判定树比判定表更直观，但简洁性却不如判定表，数据元素的同一个值往往要重复写多遍，而且越接近树的叶端重复次数越多。此外还可以看出，画判定树时分枝的次序可能对最终画出的判定树的简洁程度有较大影响，在这个例子中如果不是把行李重量做为第一个分枝，而是将它作为最后一个分枝，则画出的判定树将有16片树叶而不是只有9片树叶。显然判定表并不存在这样的问题。

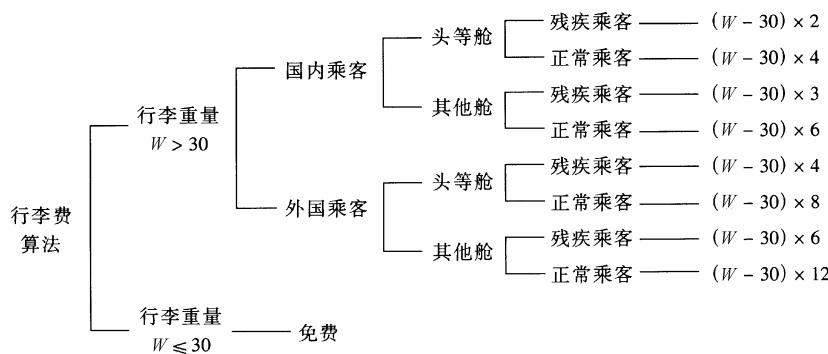


图3.28 用判定树表示计算行李费的算法

3.10.6 过程设计语言（PDL）

PDL也称为伪码，这是一个笼统的名称，现在有许多种不同的过程设计语言在使用。它是用正文形式表示数据和处理过程的设计工具。

PDL具有严格的关键字外部语法，用于定义控制结构和数据结构；另一方面，PDL表示实际操作和条件的内部语法通常又是灵活自由的，以便可以适应各种工程项目的需要。因此，一般说来PDL是一种“混杂”语言，它使用一种语言（通常是某种自然语言）的词汇，同时

却使用另一种语言（某种结构化的程序设计语言）的语法。

PDL 应该具有下述特点：

(1) 关键字的固定语法，它提供了结构化控制结构、数据说明和模块化的特点。为了使结构清晰和可读性好，通常在所有可能嵌套使用的控制结构的头和尾都有关键字，例如，if...fi (或 endif) 等等。

(2) 自然语言的自由语法，它描述处理特点。

(3) 数据说明的手段。应该既包括简单数据结构（例如纯量和数组），又包括复杂的数据结构（例如，链表或层次的数据结构）。

(4) 模块定义和调用的技术，应该提供各种接口描述模式。

本章 3.12 节将讲述用结构化方法设计一个汉字行编辑程序的过程，这个行编辑程序的实现算法就是用 PDL 描述的，为节省篇幅本节不再另外给出过程设计语言的具体例子了。

PDL 作为一种设计工具有如下一些优点：

(1) 可以作为注释直接插在源程序中间。这样做能促使维护人员在修改程序代码的同时也相应地修改 PDL 注释，因此有助于保持文档和程序的一致性，提高了文档的质量。

(2) 可以使用普通的正文编辑程序或文字处理系统，很方便地完成 PDL 的书写和编辑工作。

(3) 已经有自动处理程序存在，而且可以自动由 PDL 生成程序代码。

PDL 的缺点是不如图形工具形象直观，描述复杂的条件组合与动作间的对应关系时，不如判定表清晰简单。

3.11 面向数据结构的设计方法

计算机软件本质上是信息处理系统，因此，可以根据软件所处理的信息的特征来设计软件。前面曾经介绍了面向数据流的设计方法，也就是根据数据流确定软件结构的方法，本节将介绍面向数据结构的设计方法，也就是根据数据结构设计程序处理过程的方法。

在许多应用领域中信息都有清楚的层次结构，输入数据、内部存储的信息（数据库或文件）以及输出数据都可能有独特的结构。数据结构既影响程序的结构又影响程序的处理过程，重复出现的数据通常由具有循环控制结构的程序来处理，选择数据（即，可能出现也可能不出现的信息）要用带有分支控制结构的程序来处理。层次的数据组织通常和使用这些数据的程序的层次结构十分相似。

面向数据结构的设计方法的最终目标是得出对程序处理过程的描述。这种设计方法并不明显地使用软件结构的概念，模块是设计过程的副产品，对于模块独立原理也没有给予应有的重视。因此，这种方法最适合于在详细设计阶段使用，也就是说，在完成了软件结构设计之后，可以使用面向数据结构的方法来设计每个模块的处理过程。

Jackson 方法和 Warnier 方法是最著名的两个面向数据结构的设计方法，本节结合一个简单例子扼要地介绍 Jackson 方法，使读者对面向数据结构的设计方法有初步了解。希望了解 Warnier 方法的读者，请参阅《软件工程导论（第三版）》^[1]，需要深入了解 Warnier 方法的读者，请参阅 Warnier 本人的专著^[9]。

使用面向数据结构的设计方法，当然首先需要分析确定数据结构，并且用适当的工具清晰地描绘数据结构。本节先介绍 Jackson 方法的工具——Jackson 图，然后介绍 jackson 程序设计方法的基本步骤。

3.11.1 Jackson 图

虽然程序中实际使用的数据结构种类繁多，但是它们的数据元素彼此间的逻辑关系却只有顺序、选择和重复三类，因此，逻辑数据结构也只有这三类。

1. 顺序结构

顺序结构的数据由一个或多个数据元素组成，每个元素按确定次序出现一次。图 3.29 是表示顺序结构的 Jackson 图的一个例子。

2. 选择结构

选择结构的数据包含两个或多个数据元素，每次使用这个数据时按一定条件从这些数据元素中选择一个。图 3.30 是表示三个中选一个结构的 Jackson 图。

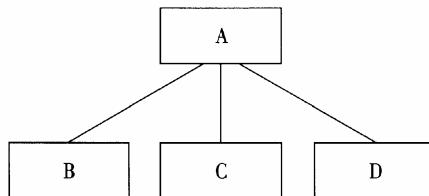


图 3.29 A 由 B、C、D 三个元素顺序组成
(每个元素只出现一次，出现的次序依次是 B、C 和 D)

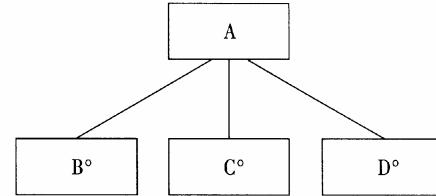


图 3.30 根据条件 A 是 B 或 C 或 D 中的某一个
(注意：在 B、C 和 D 的右上角有小圆圈做标记)

3. 重复结构

重复结构的数据，根据使用时的条件由一个数据元素出现零次或多次构成。图 3.31 是表示重复结构的 Jackson 图。

Jackson 图有下述优点：

- ? 便于表示层次结构，而且是对结构进行自顶向下分解的有力工具；
- ? 形象直观可读性好；
- ? 既能表示数据结构也能表示程序结构
(因为结构程序设计也只使用上述三种基本结构)。

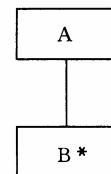


图 3.31 A 由 B 出现 N 次 (N > 0) 组成
(注意，在 B 的右上角有星号标记)

3.11.2 改进的 Jackson 图

上一小节介绍的 Jackson 图的缺点是，用这种图形工具表示选择或重复结构时，选择条件或循环结束条件不能直接在图上表示出来，影响了图的表达能力，也不易直接把图翻译成程序，此外，框间连线为斜线，不易在行式打印机上输出。为了解决上述问题，本书建议使用

图 3.32 中给出的改进的 Jackson 图。

请读者注意，虽然 Jackson 图和描绘软件结构的层次图形形式相当类似，但是含义却很不相同：层次图中的一个方框通常代表一个模块；Jackson 图即使在描绘程序结构时，一个方框也并不代表一个模块，通常一个方框只代表几个语句。层次图表现的是调用关系，通常一个模块除了调用下级模块外，还完成其他操作；Jackson 图表现的是组成关系，也就是说，一个方框中包括的操作仅仅由它下层框中的那些操作组成。

3.11.3 Jackson 方法

Jackson 结构程序设计方法基本上由下述五个步骤组成。

第 1 步 分析并确定输入数据和输出数据的逻辑结构，并用 Jackson 图描绘这些数据结构。

第 2 步 找出输入数据结构和输出数据结构中有对应关系的数据单元。所谓有对应关系是指有直接的因果关系，在程序中可以同时处理的数据单元（对于重复出现的数据单元必须重复的次序和次数都相同才可能有对应关系）。

第 3 步 用下述三条规则从描绘数据结构的 Jackson 图导出描绘程序结构的 Jackson 图：

第一，为每对有对应关系的数据单元，按照它们在数据结构图中的层次在程序结构图的相应层次画一个处理框（注意，如果这对数据单元在输入数据结构和输出数据结构中所处的层次不同，则和它们对应的处理框在程序结构图中所处的层次与它们之中在数据结构图中层次低的那个对应）。

第二，根据输入数据结构中剩余的每个数据单元所处的层次，在程序结构图的相应层次分别为它们画上对应的处理框。

第三，根据输出数据结构中剩余的每个数据单元所处的层次，在程序结构图的相应层次分别为它们画上对应的处理框。

总之，描绘程序结构的 Jackson 图应该综合输入数据结构和输出数据结构的层次关系而导出来。在导出程序结构图的过程中，由于改进的 Jackson 图规定在构成顺序结构的元素中不能有重复出现或选择出现的元素，因此可能需要增加中间层次的处理框。

第 4 步 列出所有操作和条件（包括分支条件和循环结束条件），并且把它们分配到程序结构图的适当位置。

第 5 步 用伪码表示程序。

Jackson 方法中使用的伪码和 Jackson 图是完全对应的，下面是和三种基本结构对应的伪码。

和图 3.32 (a) 所示的顺序结构对应的伪码，其中 ‘ seq ’ 和 ‘ end ’ 是关键字：

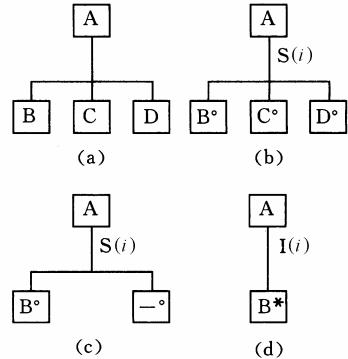


图 3.32 改进的 Jackson 图

- (a) 顺序结构，B、C、D 中任一个都不能是选择出现或重复出现的数据元素（即，不能是右上角有小圆或星号标记的元素）；(b) 选择结构，S 右面括号中的数字 i 是分支条件的编号；(c) 可选结构，A 或者是元素 B 或者不出现（可选结构是选择结构的一种常见的特殊形式）；(d) 重复结构，循环结束条件的编号为 i

A seq

B

C

D

A end

和图 3.32 (b) 所示的选择结构对应的伪码，其中 ‘select’、‘or’ 和 ‘end’ 是关键字，cond1、cond2 和 cond3 分别是执行 B、C 或 D 的条件：

A select cond1

B

A or cond2

C

A or cond3

D

A end

和冬

关键字（重复结构有 until 和 while 两种形式），cond 是条件：
A iter until (或 while) cond

B

丁酉年

【例 1】一个正文文件由若干个记录组成，每个记录是一个字符串。

[例] 一个正文文件由若干个记录组成，每个记录是一个字符串。要求统计每个记录中空格字符的个数，以及文件中空格字符的总个数。要求的输出数据格式是，每复制一行输入字符串之后，另起一行印出这个字符串中的空格数，最后印出文件中空格的总个数。

对于这个简单例子而言，输入和输出数据的结构很容易确定。图 3.33 是用 Jackson 图描绘的输入 / 输出数据结构。

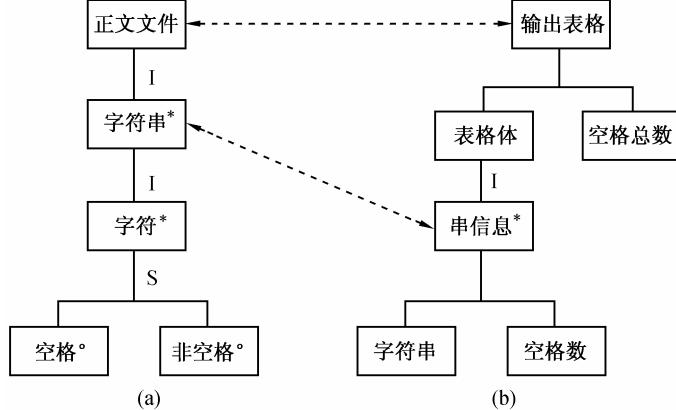


图 3.33 表示输入 / 输出数据结构的 Jackson 图

(a) 输入数据结构 ; (b) 输出数据结构

确定了输入 / 输出数据结构之后，下一步是分析确定在输入数据结构和输出数据结构中有对应关系的数据单元。在这个例子中哪些数据单元有对应关系呢？输出数据总是通过对输入数据的处理而得到的，因此在输入 / 输出数据结构最高层次的两个单元（在这个例子中是“正文文件”和“输出表格”）总是有对应关系的。这一对单元将和程序结构图中最顶层的方框（代表程序）相对应，也就是说经过程序的处理由正文文件得到输出表格。下面还有哪些有对应关系的单元呢？因为每处理输入数据中一个“字符串”之后，就可以得到输出数据中一个“串信息”，它们都是重复出现的数据单元，而且出现次序和重复次数都完全相同，因此，“字符串”和“串信息”也是一对有对应关系的单元。

还有其他有对应关系的单元吗？为了回答这个问题我们依次考察输入数据结构中余下的每个数据单元。“字符”不可能和多个字符组成的“字符串”对应，和输出数据结构中其他数据单元也不能对应。“空格”能和“空格数”对应吗？显然，单个空格并不能决定一个记录中包含的空格个数，因此没有对应关系。通过类似的考察发现，输入数据结构中余下的任何一个单元在输出数据结构中都找不到对应的单元，也就是说，在这个例子中输入 / 输出数据结构中只有上述两对有对应关系的单元。在图 3.33 中用一对虚线箭头把有对应关系的数据单元连接起来，以突出表明这种对应关系。

Jackson 程序设计方法的第三步是从数据结构图导出程序结构图。按照前面已经讲述过的规则，这个步骤的大致过程是：

首先，在描绘程序结构的 Jackson 图的最顶层画一个处理框“统计空格”，它与“正文文件”和“输出表格”这对最顶层的数据单元相对应。但是接下来还不能立即画与另一对数据单元（“字符串”和“串信息”）相对应的处理框，因为在输出数据结构中“串信息”的上层还有“表格体”和“空格总数”两个数据单元，在程序结构图的第二层应该有与这两个单元对应的处理框——“程序体”和“印总数”。因此，在程序结构图的第三层才是与“字符串”和“串信息”相对应的处理框——“处理字符串”。在程序结构图的第四层似乎应该是和“字符串”、“字符”及“空格数”等数据单元对应的处理框“印字符串”、“分析字符串”及“印空格数”，这三个处理是顺序执行的。但是，“字符”是重复出现的数据单元，因此“分析字符串”也应该是重复执行的处理。改进的 Jackson 图规定顺序执行的处理中不允许混有重复执行或选择执行的处理，所以在“分析字符串”这个处理框上面又增加了一个处理框“分析字符串”。最后得到的程序结构图为图 3.34。

Jackson 程序设计方法的第四步是列出所有操作和条件，并且把它们分配到程序结构图的适当位置。首先，列出统计空格个数需要的全部操作和条件如下：

- (1) 停止 (2) 打开文件
- (3) 关闭文件 (4) 印出字符串
- (5) 印出空格数目 (6) 印出空格总数
- (7) sum := sum + 1 (8) totalsum := totalsum + sum

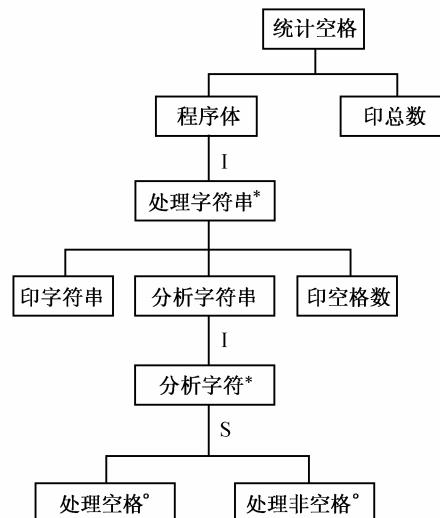


图 3.34 描绘统计空格程序结构的 Jackson 图

(9) 读入字符串 (10) sum := 0

(11) totalsum := 0 (12) pointer := 1

(13) pointer := pointer + 1 I(1) 文件结束

I(2) 字符串结束

S(3) 字符是空格

在上面的操作表中，sum 是保存空格个数的变量，totalsum 是保存空格总数的变量，而 pointer 是用来指示当前分析的字符在字符串中的位置的变量。

经过简单分析后不难把这些操作和条件分配到程序结构图的适当位置，结果如图 3.35 所示。Jackson 方法的最后一步是用伪码表示程序处理过程。因为 Jackson 方法使用的伪码和

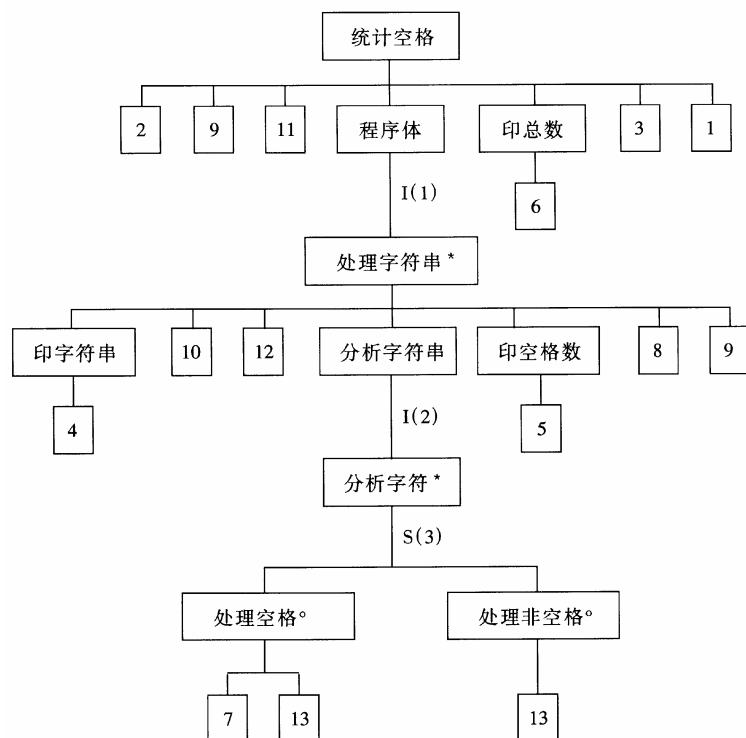


图 3.35 把操作和条件分配到程序结构图的适当位置

Jackson 图之间存在简单的对应关系，所以从图 3.35 很容易得出下面的伪码：

统计空格 seq

打开文件

读入字符串

totalsum := 0

程序体 iter until 文件结束

处理字符串 seq

印字符串 seq

印出字符串

```

印字符串 end
sum : = 0
pointer : = 1
分析字符串 iter until 字符串结束
    分析字符 select 字符是空格
        处理空格 seq
            sum : = sum + 1
            pointer : = pointer + 1
        处理空格 end
    分析字符 or 字符不是空格
        处理非空格 seq
            pointer : = pointer + 1
        处理非空格 end
    分析字符 end
分析字符串 end
印空格数 seq
    印出空格数目
印空格数 end
totalsum : = totalsum + sum
读入字符串
处理字符串 end
程序体 end
印总数 seq
    印出空格总数
印总数 end
关闭文件
停止
统计空格 end

```

以上简单介绍了由英国人 M.Jackson 提出的结构程序设计方法。这个方法在设计比较简单 的数据处理系统时特别方便，当设计比较复杂的程序时常常遇到输入数据可能有错、条件不 能预先测试、数据结构冲突等问题。为了克服上述困难，把 Jackson 方法应用到更广阔的领域， 需要采用一系列比较复杂的辅助技术，详细介绍这些技术已经超出本书的范围。需要更深入 了解 Jackson 方法的读者，请参阅 Jackson 本人的专著^[8]。

3.12 结构化设计实例

本章前面各节讲述了软件设计的任务，指导软件设计的准则和启发规则，软件设计的技 术方法以及工具，本节将讲述一个汉字行编辑程序的结构化设计过程。在设计这个软件的过

程中，综合应用了前面各节讲述的知识。通过对这个实际例子的学习，读者不仅能加深对本章前面各节所述内容的理解，也可初步学会运用结构化设计方法开发实际软件。

本节所设计的汉字行编辑程序的功能虽然比较简单，但是也包含了绝大多数常用的基本编辑命令，而且在汉字操作系统的支持下，可以输入和编辑汉文，因此有一定的实用价值。

本节一方面通过一个实用程序的设计过程进一步讲述结构化设计方法，另一方面也为广大读者提供了一份适当的实习材料。读者可以依据描述设计结果的伪码写程序，从而实习编码和测试的过程，还可以进一步增加编辑命令或改进已有的设计结果，从而初步体验软件维护的过程。

3.12.1 汉字行编辑程序的规格说明

正文编辑语言是对一个正文编辑程序的基本需求，该编辑程序必须能够正确地处理指定的编辑语言。这个例子中正文编辑语言包括外部编辑命令、编辑命令和输出信息。

我们所设计的这个正文编辑程序有两种工作模式，分别是输入模式和编辑模式。

3.12.1.1 外部编辑命令

术语“外部行”系指用户在输入模式或编辑模式打入的字符串。一个外部行定义为一个字符串（可以是空字符串），这个字符串由可以打字的字符组成。在换行之前输入的字符串的长度不允许超过预先规定的固定长度。

正文编辑程序一个字符一个字符地接受一个外部行，并由它构造出一个内部行。如果输入的字符不满一行，那么这行的右侧由空格填充。

当用户打进一个外部行时可能会犯错误，因此需要编辑该行正文。为此给用户提供了三个外部编辑命令（参看表 3.2）。正文编辑程序识别出这些命令，并且在构造内部行时对外部行进行编辑。字符删除命令（@）的含义是“删除前一个字符或汉字”，行删除命令（#）的含义是“删除这一行中已输入的所有字符”，换行命令使用符号‘!’。在输入模式时，如果在某一行的开头就输入一个换行符，则表明这行是一个空行，它命令编辑程序由输入工作模式转到编辑工作模式。如果在一行的开头输入一个或多个空格后再输入换行符，则表明这行是空格行，它不是外部编辑命令。

表 3.2

外部编辑命令

命 令	含 义	命令（续）	含义（续）
@	字符删除命令	!	换行
#	行删除命令	空行 ^{注1}	转到编辑模式

【注】 在输入模式中，某一行开始就是换行符时，即为空行。

3.12.1.2 编辑命令

这个正文编辑程序工作在编辑模式时，共有 12 个编辑命令。表 3.3 列出了这些命令的名字、格式和编辑操作。正文编辑程序对一个不正确的编辑命令的响应是输出‘?!’。

注意，命令‘T’、‘Q’、‘E’和‘I’没有操作数；命令‘U’、‘N’、‘L’、‘D’、‘C’和

‘S’都只有一个操作数(最多为四位的十进制数)；命令‘F’有一个字符串作为操作数；命令‘R’有两个字符串和一个数(可任选)作为操作数。

表 3.3 编辑命令

名 字	命 令 ^{【注1】}	编 辑 操 作
TOP	T	使当前行指针 BPTR 指向虚拟行 ^{【注2】}
UP	Ub N	把指针 BPTR 往上移 N 行 ^{【注3】}
NEXT	Nb N	把指针 BPTR 往下移 N 行
ENTER	E	进入输入模式
LIST	Lb N	从当前行开始输出 N 行正文
DELETE	Db N	从当前行开始删除 N 行正文。删掉的正文放在自由链前面
REPLACE	Rb / STR1 / STR2 / bN	在从当前行开始的 N 行正文中，用 STR2 替换每一个 STR1
FIND	Fb / STRING1/	从下一行开始扫描各行正文，使指针 BPTR 指向第一个包含字符串 STRING1 的正文行
COPY	Cb N	从正文链中复制 N 行正文放到工作链的尾部，但并不把这些正文行从正文链中删掉
STORE	Sb N	从正文链中把 N 行正文移到工作链的尾部
INSERT	I	把工作链中的所有正文行插到正文链中当前行的后面
QUIT	Q	停止编辑程序的运行

【注 1】N 代表空格或十进制正整数(最多 4 位)。STR1(或 STRING1)代表需要在正文链中匹配的字符串，STR2 代表替换字符串。N 是空格时补缺值为 1。

【注 2】BPTR 代表正文链中的当前行指针。

【注 3】b 代表空格。

3.12.1.3 输出信息

除了列出正文链中的一行行正文之外，正文编辑程序还能输出下列 7 个信息：‘EDIT!’、‘INPUT!’、‘?!’、‘TOF!’、‘EOF!’、‘NOTEXT!’ 和 ‘NOFREE!’。表 3.4 中列出了这些信息的含义，下面进一步解释这些信息。

1. ‘EDIT!’ 响应

这是当正文编辑程序处在输入模式中时，对在终端上输入一个空行(‘!’)所做的响应。它指出已经进入了编辑模式，以后从终端输入的正文行将被解释为编辑命令。

2. ‘INPUT!’ 响应

这是在编辑模式中给出 ENTER 命令(‘E’)之后，程序的响应。它指出正文编辑程序已经进入输入模式，以后从终端输入的字符串将作为正文行加入到正文链中。

表 3.4

输出信息

输出信息	含 义
EDIT !	指出已经进入编辑模式 ^{【注1】}
INPUT !	指出已经进入输入模式 ^{【注2】}
? !	无效的编辑命令，或输入字符超过了一行的长度，或在转换中发现一个错误
TOF !	正文链的顶 ^{【注3】}
EOF !	正文链结束
NOTEXT !	正文链是空的 ^{【注4】}
NOFREE !	自由链中已经没有存储块了

【注1】为了进入编辑模式，用户在输入模式打进一个空行（仅一个换行符）。

【注2】为了进入输入模式，用户打入 ENTER 命令（‘E’）。

【注3】当前行指针 BPTR 指向虚拟行（正文链中第一行的前面）。

【注4】当执行 ENTER 命令和 INSERT 命令时例外。

3. ‘?!’ 响应

这是对无效的编辑命令的响应。正文编辑程序忽略无效的命令，并且仍然处于编辑模式中，等待接收下一个编辑命令。

4. ‘TOF!’ 响应

这是对把当前行指针移到正文链中第一行前面去的 UP 命令的响应。

5. ‘EOF!’ 响应

每当一个编辑命令试图把当前行指针移到正文链中最后一行的后面去的时候，正文编辑程序都显示这个信息。在这种情况下，当前行指针总是指向正文链中的最后一行（如果正文链是空的，则指向虚拟行）。

6. ‘NOTEXT!’ 响应

当正文链是空的时，正文编辑程序对于除 ENTER 和 INSERT 之外的所有编辑命令，都回答‘NOTEXT! ’。

7. ‘NOFREE!’ 响应

当在输入模式时这个响应指出，已经没有空闲的主存储块可以用来存放输入的正文行了。当出现这种情况时，正文编辑程序忽略最后输入的正文行，并且自动进入编辑模式。仍然可以命令它转回输入模式，但是如果试图往正文链中增加一行正文，它将再次给出‘NOFREE!’响应。如果没有足够的存储块供 COPY 命令使用，在终端上也将显示‘NOFREE! ’，这时将忽略这个 COPY 命令并且不做任何改动。

3.12.2 概要设计

概要设计给出这个汉字行编辑程序的概貌。它描述该编辑系统的正文文件、工作模式、当前行指针以及正文文件的虚拟行等。

3.12.2.1 正文文件

需要编辑的正文储存在正文文件中。如图 3.36 所示，正文文件由字块组成，每个字块有三个域：N、U 和 S。域 N 是一个指针，它指向下一个字块。域 U 也是一个指针，它指向前一个字块。域 S 是一个固定长度的字符串（也就是一行正文）。在正文编辑期间，这些字块由这些指针链接起来，最多可以形成三个双链结构，它们组成正文文件的三个部分：正文链、工作链和自由链。正文链中当前编辑的那行正文由当前行指针指定。此外，可以想象在正文链最前面还有一行事实上并不存在的正文，称为“虚拟行”。

1. 正文链

正文链是双链结构。对于编辑程序的用户而言，正文链由正文行组成，每行可以容纳固定数目的字符。如图 3.36 所示，指针 T PTR 指向正文链的第一行，而指针 B PTR 指向当前行。如果指针 T PTR 为 ϕ ，则表明正文链中没有正文。符号 ϕ 称为零指针，它表明链结构结束。

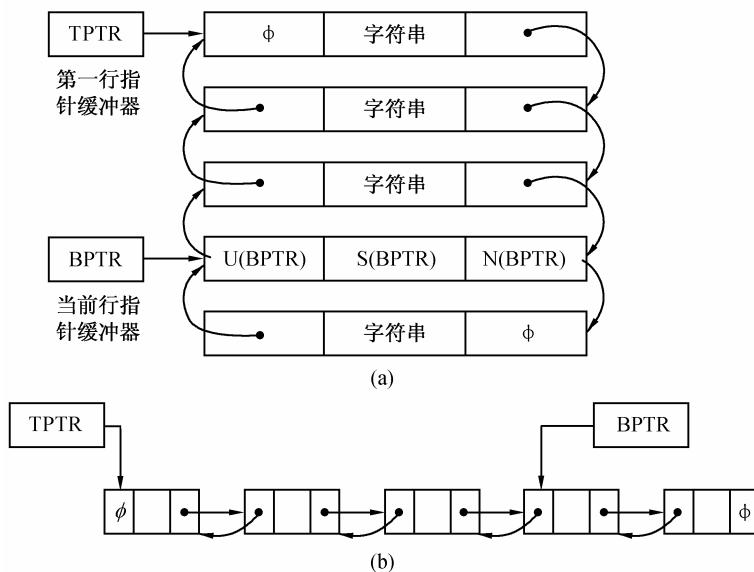


图 3.36 正文文件的组织

(a) 对正文文件的一种看法；(b) 对正文文件的另一种看法

$N(B PTR)$: 字块中的一个指针，指向下一个字块

$U(B PTR)$: 字块中的一个指针，指向前一个字块

$S(B PTR)$: 指针 B PTR 指定的字块中的字符串

ϕ : 零指针

2. 工作链

工作链是双链结构的先进先出队列。编辑命令 COPY、STORE 和 INSERT 使用工作链。

3. 自由链

自由链也是一个双链结构，它链接所有自由的字块（也就是既不属于正文链也不属于工作链的字块）。在正文编辑期间，编辑程序动态地分配和释放字块。

4. 当前行指针

当前行指针 BPTR 和正文链有关，它指向正文链中当时正被处理的那一行（或者正在产生这个当前行或者正在编辑这个当前行）。编辑命令的一个功能就是移动这个当前行指针。可以把这个指针从它的当前位置上移或下移给定的行数，也可以把它移到指向包含给定字符串的那一行。

搜索字符串的能力减轻了用户的负担：用户可以不关心行号，也可以不记录插入和删除的行号，只需关心各行正文的内容即可。

应该注意，工作模式改变之后，当前行指针的位置如下：

(1) 当从输入模式改变到编辑模式时，当前行指针指向由终端输入的最后一行。如果没有输入，那么它仍然保持指向进入输入模式时指向的那一行。

(2) 当从编辑模式改变到输入模式时，当前行指针的位置不变。随后在输入模式输入的正文行，将紧接在当前行后面插入到正文中。

5. 虚拟行

在正文链第一行前有一行虚拟行。在当前行指针 BPTR 是 的时候，当前行即是这个虚拟行。在 TOP 命令之后，或某些情况在 UP 或 DELETE 命令之后，BPTR 将指向虚拟行。

3.12.2.2 两个工作模式

本编辑系统有两个工作模式——输入模式和编辑模式。当编辑程序的主过程——EDITOR ——运行时，它或者处于输入模式中，或者处于编辑模式中。

1. 输入模式

为了输入一行行的正文而使用输入模式。当正文编辑程序处于输入模式中时，它将把在终端打进的任何正文行加入到正文链中。为了从编辑模式进入输入模式，用户打入 ENTER 命令 ‘E!’（‘!’代表终端上的换行符）。EDITOR 回答 ‘INPUT! ’，并且容许用户开始输入一行正文。注意，空行和空格行是不同的。为了在正文链中插入一行空格，用户必须至少打进一个空格，然后再打一个换行符；为了打进一个空行，仅需简单地打一个换行符。

2. 编辑模式

为了进入编辑模式（也就是离开输入模式），用户打进一个空行，然后编辑程序在终端上显示 ‘EDIT! ’，告诉用户已经进入编辑模式。仅当处在编辑模式时，才能输入编辑命令。如果在输入模式时打入编辑命令，那么这些编辑命令将成为正文链中的正文行。如果输入一个无效的编辑命令，EDITOR 将忽略这个命令，显示 ‘?! ’，并且等待下一个命令。

3.12.2.3 数据元素

在概要设计中选取一个表格、五个缓冲器和一个开关作为编辑程序的数据元素。

表格 TEXT 有三个域，用来存放正文文件：域 S 存放一行正文，域 U 和域 N 存放指针。

五个缓冲器中有两个为指针缓冲器 (TPTR 和 BPTR)，另三个为输入 / 输出缓冲器 (BUF，IN 和 CHAR)。缓冲器 BUF 存放一个字符，这个字符是准备传送给终端的，或者是从终端输入进来的；缓冲器 IN 存放一行输入的（或输出的）正文；缓冲器 CHAR 存放一个从缓冲器 IN 取来的字符。

开关 IND 处于状态 0 时表明是在编辑模式，处于状态 1 或 2 时表明是在输入模式（以后将解释用两个状态标志输入模式的原因）。

图 3.37 描绘了这些数据元素以及它们之间的关系。

3.12.2.4 过程

所谓过程 (Procedure) 就是本设计中的模块。

正文编辑程序的主过程叫 EDITOR , 它根据开关 IND 的状态 , 决定是接受输入的正文还是执行编辑命令 , 并相应地完成下述步骤 :

步骤 1 : 从终端读入一行 ;

步骤 2 : 根据开关 IND 确定模式 ;

(1) 输入模式 : 转到步骤 3。

(2) 编辑模式 : 转到步骤 4。

步骤 3 : 调用过程 INPUT , 以处理输入的一行正文 ; 转到步骤 5 ;

步骤 4 : 确定编辑命令 ;

(1) 如果是命令 ‘T’ , 执行过程 TOP ;

(2) 如果是命令 ‘U’ , 执行过程 UP ;

(3) 如果是命令 ‘N’ , 执行过程 NEXT ;

(4) 如果是命令 ‘E’ , 执行过程 ENTER ;

(5) 如果是命令 “L” , 执行过程 LIST ;

(6) 如果是命令 ‘D’ , 执行过程 DELETE ;

(7) 如果是命令 ‘R’ , 执行过程 REPLACE ;

(8) 如果是命令 ‘F’ , 执行过程 FIND ;

(9) 如果是命令 ‘C’ , 执行过程 COPY ;

(10) 如果是命令 ‘S’ , 执行过程 STORE ;

(11) 如果是命令 ‘I’ , 执行过程 INSERT ;

(12) 如果是命令 ‘Q’ , 停止编辑程序的运行 ;

(13) 如果是其他字母 , 在终端上显示 ‘?! ’。

步骤 5 : 转到步骤 1。

过程 EDITOR 总共调用 14 个过程 , 以完成这个行编辑程序的输入和编辑功能。下一节描述这个行编辑程序的概要设计结果。

3.12.3 概要设计结果

在 3.12.2.3 小节中已经描述了通过概要设计为这个编辑程序选取的数据元素 , 本节着重描述程序结构和组成程序的主要过程。

通过概要设计得出的行编辑程序结构如图 3.38 所示。

通过概要设计 , 我们把这个行编辑程序按照功能分解成 15 个模块 (即过程) 。主过程 EDITOR 调用 14 个下属过程 , 其中 11 个下属过程与 11 条编辑命令一一对应 , 每个过程完成一条编辑命令 , 其余三个下属过程分别完成处理一行正文输入 (INPUT) 、从终端读入一行正

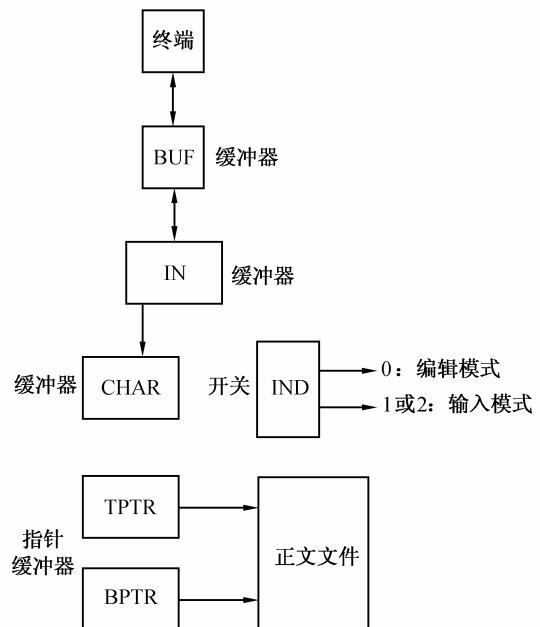


图 3.37 编辑程序的数据元素

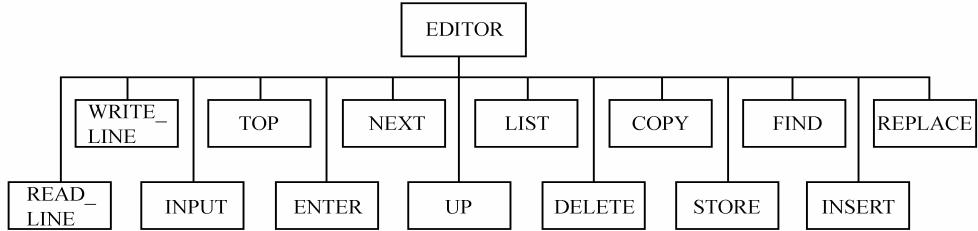


图 3.38 编辑程序的结构 (概要设计)

文 (READ_LINE) 和把一行正文显示在终端上 (WRITE_LINE) 的功能。

下面用伪码描述每个过程。本书 3.10 节已经介绍了伪码的特点，此处不再赘述。在概要设计中，重点应该设计程序结构，并确定所需要的数据结构和每个模块的功能，完成每个模块功能的精确算法应该在详细设计阶段设计。因此，下面对每个过程的描述，实际上是对每个过程功能的描述。

为了便于阅读便于理解，对每个过程的描述主要使用自然语言。为了突出控制结构，使控制流程一目了然，在描述概要设计结果的伪码中也使用了一些关键字(例如，用关键字 Loop 和 Endloop 把循环体括起来)。

下面列出对每个过程的伪码描述：

```

Procedure EDITOR ;
  把开关 IND 置成编辑模式 ;
  Loop
    调用过程 READ_LINE 从终端把一行正文读入到缓冲器 IN 中 ;
    If IND 是输入模式
      Then 调用过程 INPUT 处理正文输入 ;
      Else /*是编辑模式*/
        把缓冲器 IN 中的下一个字符移到缓冲器 CHAR 中 ;
        Case Of 缓冲器 CHAR 的内容
          'T' : 调用过程 TOP 以执行 TOP 命令 ;
          'U' : 调用过程 UP 以执行 UP 命令 ;
          'N' : 调用过程 NEXT 以执行 NEXT 命令 ;
          'E' : 调用过程 ENTER 以执行 ENTER 命令 ;
          'L' : 调用过程 LIST 以执行 LIST 命令 ;
          'D' : 调用过程 DELETE 以执行 DELETE 命令 ;
          'R' : 调用过程 REPLACE 以执行 REPLACE 命令 ;
          'F' : 调用过程 FIND 以执行 FIND 命令 ;
          'C' : 调用过程 COPY 以执行 COPY 命令 ;
          'S' : 调用过程 STORE 以执行 STORE 命令 ;
          'I' : 调用过程 INSERT 以执行 INSERT 命令 ;
          'Q' : Exitloop ;

```

软件工程

```
Else : 在缓冲器 IN 中放 “ ?! ” ;
      调用过程 WRITE_LINE 把缓冲器 IN 的内容写在终端上 ;
Endcase ;
Endif ;
Endloop ;
End EDITOR ;

Procedure          TOP ;
      把当前行指针 BPTR 移到虚拟行 ;
      /* 把 BPTR 置成 0* /
End TOP ;

Procedure UP ;
      把当前行指针 BPTR 往上移给定行数 ( N ), 然后在终端上显示新的当前行 ;
      如果 N 是空格 , 那么它的值是 1 ;
End UP ;

Procedure NEXT ;
      把当前行指针 BPTR 往下移给定行数 ( N ), 然后在终端上显示新的当前行 ;
      如果 N 是空格 , 那么它的值等于 1 ;
End NEXT ;

Procedure ENTER ;
      从编辑模式转变到输入模式 ;
      /* 把开关 IND 置成 1* /
End ENTER ;

Procedure LIST ;
      在终端上显示正文文件中从当前行开始的 N 行正文 ;
      如果 N 是空格 , 它的值相当于 1 ;
End LIST ;

Procedure DELETE ;
      删除正文文件中从当前行开始的 N 行正文 ;
      把删掉的那些行加在自由链的前面 ;
      如果 N 是空格 , 它的值相当于 1 ;
End DELETE ;

Procedure REPLACE ;
```

```
在从当前行开始的 N 行正文中，每出现一个 ‘STR1’ 就用 ‘STR2’ 替换它；  
如果 N 是空格，它的值相当于 1；  
End REPLACE；  
  
Procedure FIND；  
    从当前行的下面一行开始扫描各行正文，把当前行指针移到指向第一次出现  
    ‘STRING1’ 的那一行；  
End FIND；  
  
Procedure INSERT  
    把工作链中的所有正文都移到正文链中，插在当前行的后面；  
End INSERT；  
  
Procedure COPY；  
    把正文链中从当前行开始的 N 行正文复制到自由链中，再从自由链移到工作链的尾部；  
    如果 N 是空格，它的值相当于 1；  
End COPY；  
  
Procedure STORE；  
    删掉正文链中从当前行开始的 N 行正文，把它们接在工作链的尾部；  
    如果 N 是空格，它的值相当于 1；  
End STORE；  
  
Procedure READ_LINE；  
    从终端一个一个地读入字符，在缓冲器 IN 中把它们装配成一行正文；  
End READ_LINE；  
  
Procedure WRITE_LINE；  
    把缓冲器 IN 中的字符一个接一个地显示在终端上；  
End WRITE_LINE；  
  
Procedure INPUT；  
    在输入模式中，这个过程从缓冲器 IN 一行一行地接收正文，并把它们一行一行地存  
    进自由链中；  
    在离开输入模式之前，把存在自由链中的这些行正文链接到正文链中；  
End INPUT；
```

3.12.4 详细设计

3.12.4.1 数据元素

我们从选取数据元素开始详细设计，选取的数据元素示于图 3.39 中。其中有 21 个缓冲器

和 1 个表格。因为处理指针是这个编辑程序最经常的操作，所以在 21 个缓冲器中有 12 个是为指针而设立的。表格 TEXT 用于存放正文文件。下面详细叙述这些数据元素。

1. 正文文件

正文文件是一个表格，每个表格项是一个字块；每个字块有三个域，分别是指向下一个字块的指针 (TEXT.N)，指向前面一个字块的指针 (TEXT.U) 和用来存放一行正文的固定长度的字符串 (TEXT.S)。用这些指针把字块链接成三个链——工作链、自由链和正文链。

2. 输入字符串缓冲器 IN

在缓冲器 IN 中构造内部行。这个缓冲器中的字符位置由指针 I 指定。输出信息时也用缓冲器 IN 作为输出字符串的缓冲器。

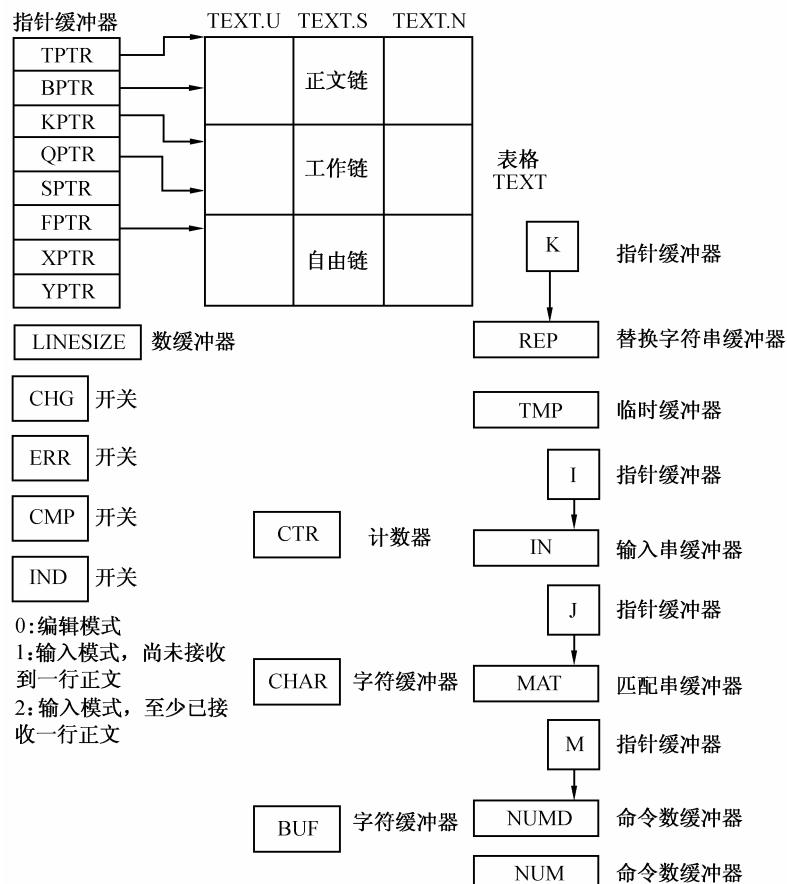


图 3.39 详细设计中的编辑程序数据元素

3. 匹配字符串缓冲器 MAT

缓冲器 MAT 中存放 REPLACE 和 FIND 两个命令使用的匹配字符串。指针 J 指向缓冲器 MAT 中的字符。

4. 替换字符串缓冲器 REP

缓冲器 REP 中存放供 REPLACE 命令使用的替换字符串。指针 K 指向这个缓冲器中的字符。

5. 临时存储缓冲器 TMP

缓冲器 TMP 临时存放缓冲器 IN 或 REP 的内容。

6. 命令数缓冲器 NUMD 和 NUM

缓冲器 NUMD 存放使用数字的编辑命令的数域。过程 CONVERT 把缓冲器 NUMD 中以字符形式编码的十进制数转变成数字编码的十进制数，并且放到缓冲器 NUM 中。指针 M 指向缓冲器 NUMD 中的字符。

7. 字符缓冲器 BUF 和 CHAR

数据在终端和编辑程序之间以字符或汉字为单位传送。在终端和缓冲器 IN 之间的传送用 BUF 做缓冲器。缓冲器 CHAR 中存放一个从缓冲器 IN 取来的供处理的字符。

8. 计数器 CTR

在执行有数字作参数的编辑命令时，先把数从缓冲器 NUM 中传送到 CTR 中，然后用 CTR 作为命令行数记数器。

9. 行长缓冲器 LINESIZE

缓冲器 LINESIZE 是一个只读缓冲器，它放一行正文的长度，这个长度在实现时决定。

10. 指针缓冲器

如图 3.39 所示，共有 12 个缓冲器用来存放指针。缓冲器 I、J、K 和 M 中的指针分别指向缓冲器 IN、MAT、REP 和 NUMD 中的字符。缓冲器 TPTR 和 BPTR 中的指针，分别指向正文链的第一行和当前行。缓冲器 KPTP 和 QPTR 中的指针，分别指向工作链的第一行和最后一行。缓冲器 FPTP 中的指针指向自由链的第一行。缓冲器 SPTR、XPTR 和 YPTR 临时存放指针。

3.12.4.2 控制数据元素

选取四个开关——CMP、CHG、ERR 和 IND——作为控制数据元素。开关 IND 指示工作模式——编辑或输入；开关 CMP 标志在一次比较操作中已经匹配；开关 ERR 指出已发现了一个错误；开关 CHG 标志命令 REPLACE 规定的对一行正文的子串替换已经完成。

3.12.4.3 编辑过程

通过详细设计，不仅细化了概要设计中已经有的过程，而且还增加了一些新过程。经过详细设计之后，编辑程序共有 26 个过程，其中 EDITOR 是主过程。这一小节描述：(1) 过程 EDITOR，它直接或间接地调用其余所有过程；(2) 过程 INIT，它为编辑程序做初始化的工作；(3) 过程 READ_LINE，它完成输入操作；(4) 过程 WRITE_LINE，它完成输出操作。

1. 过程 EDITOR

在初始化之后它从终端读进一行正文，因为初始化过程 INIT 把它置成编辑模式，所以读进的这一行正文应该包含一个编辑命令。

如果开关 IND 不是零，那么过程 EDITOR 是在输入模式中。EDITOR 调用过程 INPUT，从终端把一行正文输入到正文文件中。它继续不断地从终端接受一行行的正文，直到接收到一个空行为止。空行——前面没有空格或其他字符的一个换行符——是从输入模式进入编辑模式的命令。

如果开关 IND 是零，那么过程 EDITOR 是在编辑模式。它把从终端读进的一行字符解释为一个编辑命令，并且调用适当的过程来执行这个命令。如果用户打进一个非法命令，它输

出信息‘?!’，然后从终端读进另一行字符。

2. 过程 INIT

它为过程 EDITOR 做初始化工作，把所有开关置成零。计数器 CTR，缓冲器 CHAR、IN、BUF、MAT、REP、UNMD、NUM 和 TMP，以及指针缓冲器 XPTR、YPTR 和 SPTR 的初始内容是无关紧要的。正文链和工作链在开始时是空的，缓冲器 BPTR、TPTR、KPTR 和 QPTR 的初始内容为（零指针）。正文文件在开始时链接成自由链，指针 FPTR 指向自由链的第一行。

3. 过程 READ_LINE

它利用读操作在缓冲器 IN 中一个字符一个字符地装配一行内部行。读操作把一个字符或一个汉字从终端传送到缓冲器 BUF，如果这个字符不是‘@’、‘#’或‘!’（它们是外部编辑命令），那么过程 READ_LINE 把它装配到缓冲器 IN 中去。当遇到换行符或输入的字符数已经超过一行的固定长度时，过程 READ_LINE 结束。

4. 过程 WRITE_LINE

这个过程利用写操作输出缓冲器 IN 中的字符串，它一次传送一个字符到缓冲器 BUF，再从这里传送到终端。当达到行的固定长度时输出一个换行符到终端以结束这一行。

3.12.4.4 输入模式的过程

在输入模式期间，把正文装配到正文文件中，过程 INPUT、CONNECT、GO_EDIT 和 INSERT_CHAIN 一起完成这项任务。

1. 过程 INPUT

这个过程从缓冲器 IN 接受输入的一行正文，并且把这行正文放到自由链的第一个可用的字块中去。在离开输入模式之前把新构造的正文链与自由链分离开，并且把它紧接在原有正文链当前行的后面，从而与原有的正文链合并。

当开关 IND 是 1 或 2 的时候，正文编辑程序处在输入模式中。在接收到一行输入字符之前，开关 IND 是 1；一旦接收到一行输入字符之后，如果满足下述条件：

- (1) 缓冲器 IN 中的第一个字符不是‘!’；
- (2) 开关 IND 是 1；
- (3) 缓冲器 FPTR 中不是零指针。则把开关 IND 置成 2。

当这个过程在缓冲器 IN 中发现一个空行（第一个字符是‘!’的字符行）时，它调用过程 GO_EDIT 把当时的输入模式（IND = 1 或 2）转变成编辑模式（IND = 0）。然而，如果这个空行不是输入的第一行，那么它在转变工作模式之前，先调用过程 CONNECT 和 INSERT_CHAIN，以便把已经接收到的那些行正文从自由链移到正文链中去。

2. 其他过程

在输入模式期间，缓冲器 XPTR、YPTR 和 SPTR 存储辅助的指针。缓冲器 XPTR 中的指针指向自由链内当前输入的那一行；缓冲器 YPTR 指向输入的第一行正文（也就是自由链中由缓冲器 FPTR 指定的那一行）；而缓冲器 SPTR 将临时指向正文链中的一行正文，当把输入的正文接到正文链中去的时候，这行正文紧接在输入的正文下面。

过程 GO_EDIT 把开关 IND 置成 0，从而把正文编辑程序从输入模式转变到编辑模式，然后它在终端上印出信息‘EDIT!’，提醒用户现在已经进入了编辑模式。

过程 CONNECT 把缓冲器 FPTR 中的指针（指向目前仍然在自由链中的输入的第一行正

文),临时存放在缓冲器 YPTR 中。然后它把输入的最后一行正文(指针 XPTR 指向这一行)的“下链”指针(即指向下一个字块的指针)存到缓冲器 FPTR 中。测试这个指针是否是零指针,如果不是零指针则把指针 FPTR 指定的那行的“上链”指针(即指向一个字块的指针)置成。

当过程 INPUT 调用过程 INSERT_CHAIN 的时候,该过程把在自由链中新构成的正文链同自由链分离开,然后把新正文链合并到原有的正文链中去,紧接着在原有正文链的当前行后面。如果是过程 INSERT 调用 INSERT_CHAIN 过程,那么它把缓冲器 YPTR 和 XPTR 所指定的工作链中的那些行正文分离下来,然后把它们插到正文链中去,紧接着在由指针 BPTR 指定的当前行的后面。

在执行过程 INSERT_CHAIN 期间对指针的操作由图 3.40 描绘。这个图描绘的情况是,指针 BPTR 不指向虚拟行,而且正文链的后半部分不是空的。其中 S PTR 是一个临时的指针缓冲器。这个过程共完成六个指针操作,其中两个是对输入字块的第一行正文的上链指针和最后一行正文的下链指针的操作;另外两个分别是对输入字块将要插入其后面的那行正文的下链指针的操作。

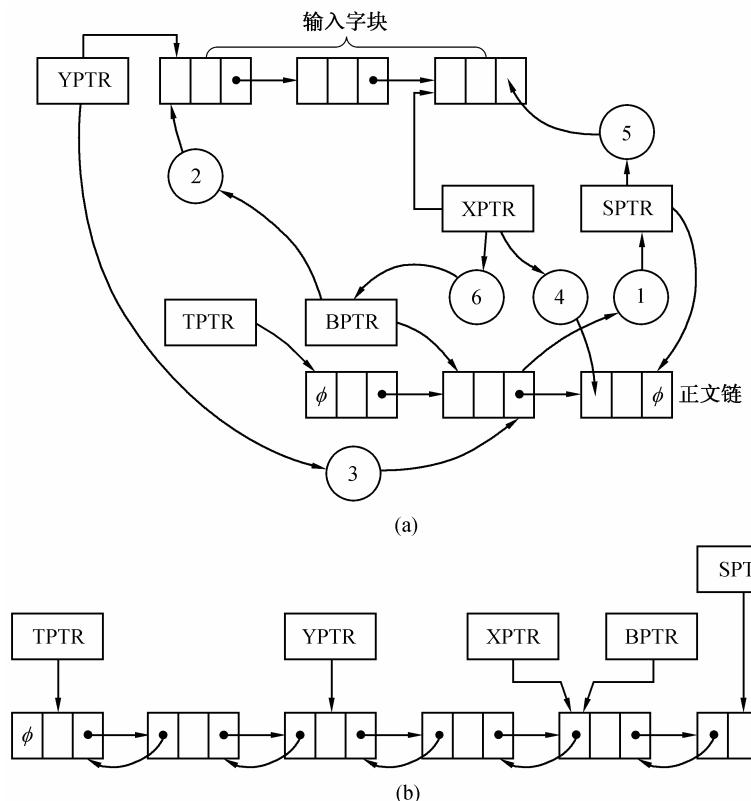


图 3.40 过程 INSER_CHAIN 中的指针操作

(a) 插入期间的正文链;(b) 插入后的正文链

六个操作是:(1) S PTR := TEXT.N(B PTR);(2) TEXT.U(Y PTR) := B PTR;(3) TEXT.N(B PTR) := Y PTR;

(4) TEXT.U(S PTR) := X PTR;(5) TEXT.N(X PTR) := S PTR;(6) B PTR := X PTR

注意:此图描绘的情况是,B PTR 不在虚拟行而且正文链的后半部分不是空的

指针，及输入字块将要插到其前面的那行正文的上链指针的操作；最后两个是和使用临时指针 S PTR 有关的操作。

3.12.4.5 编辑模式的过程

在编辑模式中，使用下述的简单编辑过程、正文加工过程和字符串编辑过程对正文进行编辑。

1. 简单编辑过程

为了执行六个简单编辑命令，需要使用七个过程，描述详细设计结果的伪码，详细定义了这七个过程（见 3.12.5 节）。如表 3.3 所示，QUIT、TOP 和 ENTER 命令每个都仅由一个字母组成，其他三个命令——UP、NEXT 和 LIST——每个由一个字母后跟一个空格和一个正整数 N 所组成。下面扼要介绍这几个简单编辑过程。

（1）过程 TOP

这个过程执行 TOP 命令，它把当前行指针 B PTR 移到指向虚拟行（也就是变成零指针）。

（2）过程 ENTER

这个过程执行 ENTER 命令，把正文编辑程序从编辑模式转变到输入模式：把开关 IND 置成 1，然后印出信息‘ INPUT! ’提醒用户已经进入了输入模式。

（3）过程 UP

这个过程执行 UP 命令，把当前行指针往上移指定行数（N），然后印出新的当前行。应注意下述几点：

- a. 和所有用一个数作变元的编辑命令一样，如果命令中省掉了这个数，那么它的补缺值是 1。
- b. 如果把当前行指针移到了指向虚拟行的位置，或者要求该指针指向第一行前面的某个位置，那么把指针留在指向虚拟行的位置，并且印出信息‘ TOF! ’。
- c. 如果正文链是空的，那么印出信息‘ NOTEXT! ’。

（4）过程 GET_NUMBER 和 CONVERT

过程 GET_NUMBER 确定用数作变元的那些编辑命令中的数。它把编辑命令中用字符串形式表示的那个十进制数从缓冲器 IN 传送到缓冲器 NUMD，然后调用过程 CONVERT，把它从字符编码形式转变成数字编码的十进制数，存放在缓冲器 NUM 中。如果发现错误，则把开关 ERR 置成 1，并且输出信息‘ ?! ’提醒用户。

（5）过程 NEXT

这个过程执行 NEXT 命令，把当前行指针从当前行往下移指定行数，然后输出新的当前行。请注意下述几点：

- a. 如果命令要求当前行指针指向正文链中最后一行后面的某个位置，那么使该指针指向最后一行，并且印出信息‘ EOF! ’。
- b. 如果当前行指针是在指向虚拟行的位置，那么命令‘ N1 ’将导致该指针指向正文链中的第一行。
- c. 如果正文链是空的，那么印出信息‘ NOTEXT! ’。

（6）过程 LIST

这个过程执行 LIST 命令，输出正文链中给定行数的正文（其中包括当前行），然后使当前行指针指向输出的最后一行正文。请注意下述几点：

- a. 如果命令要求列出的行数多于可以得到的行数，那么将仅列出能够列出的那些行正文。

然后使当前行指针指向正文链中的最后一行正文，并且印出信息‘EOF!’。

- b. 如果正文链是空的，那么印出信息‘NOTEXT!’。
- c. 如果当前行指针原来指向虚拟行，那么命令‘L1’将导致印出正文链中的第一行正文。

2. 正文处理过程

为了执行四个正文处理命令，需要使用六个过程，在描述详细设计结果的伪码中给出了这些过程的定义。如表3.3所示，命令INSERT仅有一个单独的字母，其余三个命令——DELETE、COPY和STORE——每一个都由一个字母后跟一个空格，空格后再跟一个十进制正整数所组成。

(1) 过程 DELETE 和 DELETE_LINK

这两个过程删除正文链中从当前行开始的N行正文，并把删掉的字块加到自由链的前面去。把当前行指针置成指向被删掉的那些行正文后面的第一行正文；如果后面已经没有正文了，则把当前行指针置成指向正文链中剩余的最后一行正文（可能是虚拟行）。请注意下述几点：

a. 如果命令要求删掉的行数多于能够得到的行数，那么将仅删除能够删除的那些行。在这种情况下，当前行指针将指向正文链中最后一行正文（可能是虚拟行），并且将印出信息‘EOF!’。

- b. 如果正文链是空的，则将印出信息‘NOTEXT!’。
- c. 如果当前行指针指向虚拟行，那么命令‘D1’将导致删掉正文链中的第一行正文。

(2) 过程 COPY 和 COPY_LINK

这两个过程把正文链中从当前行开始的N行正文复制到自由链中，然后把它们同自由链分离开，并且链接到工作链的尾部。请注意下述几点：

a. 执行完一条COPY命令之后，如果被复制的正文后面还有正文，则把当前行指针置成指向被复制的正文后面的第一行正文，否则把它置成指向被复制的最后一行正文。

- b. 如果自由链中没有足够的字块，则忽略这个命令。

c. 如果要求复制的行数多于可得到的行数，则将仅复制可以复制的那些行正文。在这种情况下，当前行指针将指向被复制的最后一行正文，并且将印出信息‘EOF!’。

- d. 如果正文链是空的，则将印出信息‘NOTEXT!’。

- e. 如果当前行指针指向虚拟行，那么命令‘C1’将仅复制正文链中的第一行正文。

(3) 过程 STORE

这个过程完成的操作实质上是COPY命令和DELETE命令完成的操作的组合。除了删掉的行是链接到工作链的尾部而不是放回自由链之外，STORE命令和DELETE命令所做的动作完全相同。当前行指针的位置以及输出的信息都和DELETE命令相同。

(4) 过程 INSERT

这个过程把工作链中所有正文都链接到正文链中，紧接在当前行的后面。然后使当前行指针指向插入的最后一行正文。如果工作链是空的，则忽略这个命令。

3. 字符串编辑过程

为了执行两个字符串编辑命令需要使用五个过程，在描述详细设计结果的伪码中，给出了这五个过程的定义。表3.3中给出了字符串编辑命令FIND和REPLACE的格式。

(1) 过程 FIND

这个过程扫描正文链并且使当前行指针指向包含最先出现字符串‘STRING1’的那一行正文。扫描从当前行下面的那行正文开始，应该扫描所有字符。请注意下述几点：

a. 如果扫描完正文链中的最后一行正文仍然找不到给定的字符串，则使当前行指针指向正文链中最后一行正文，并且印出信息‘EOF!’。

b. 如果查找成功，则在终端上印出找到的那一行正文。

c. 如果正文链是空的，则在终端上印出信息‘NOTEXT!’。

(2) 过程 GET_STRING1

这个过程从缓冲器 IN 中取来字符串放到缓冲器 MAT 中。如果发现错误则印出信息‘?!’提醒用户。

(3) 过程 COMP

FIND 和 REPLACE 命令都使用过程 COMP，这个过程一个字符一个字符地比较放在缓冲器 IN 和 MAT 中的两个字符串。它用开关 CMP 的状态指示比较的结果，状态 1 和 0 分别表示“匹配”和“不匹配”(匹配系指所有字符都相同)。

(4) 过程 REPLACE

这个过程用字符串‘STR2’代替从当前行开始的 N 行正文中的第一个字符串‘STR1’。如果替换导致超过固定行长则进行截尾。请注意下述几点：

a. 执行完这个过程之后，当前行指针指向所有被检查过的正文行中的最后一行；这个过程在终端上印出所有被修改过的正文行。

b. 如果命令要求改变的行数多于能得到的行数，则仅改变能够改变的那些行正文。

在这种情况下，当前行指针将停留在指向正文链中最后一行的位置，并且印出信息‘EOF!’。

c. 如果正文链是空的，则在终端上印出信息‘NOTEXT!’。

d. 如果当前行指针指向虚拟行，则命令‘R / STR1 / STR2 /’将只改变正文链中的第一行正文。

(5) 过程 GET_STRING2

这个过程把 REPLACE 命令中的第二个字符串从缓冲器 IN 移到缓冲器 REP 中。如果遇到错误则印出信息‘?!’。

3.12.5 详细设计结果

在 3.12.4.1 小节中已经讲述了在详细设计过程中，为这个编辑程序选取的数据元素，并在图 3.39 中描绘了这些数据元素。本节描述编辑程序的详细结构，以及组成程序的每个过程的实现算法。对实现算法使用类 PASCAL 语言的伪码描述。

3.12.5.1 编辑程序的详细结构

在概要设计阶段，根据这个编辑程序应该完成的功能把它分解成 15 个过程，在详细设计中不仅细化了这 15 个过程，而且根据实现的需要，还增加了一些完成具体功能的过程，从而使这个编辑程序总共包含了 26 个过程。

由于总共有 26 个过程，其中某些过程调用的过程又相当多，如果仍然使用层次图描述程序结构，则画图的工作量相当大。因此我们使用列表的方法表示程序结构，即针对每个过程列出下列内容：

<编号><层号><过程名> =<被调过程表>

其中，

编号是该过程的编号；

层号表示该过程在程序结构中所处的层次，层号越大则所处层次越低；

符号 = 的意思在此处是“调用”；

被调过程表即直接被该过程调用的过程表，每个过程名之间用逗号分开。如果该过程不调用其他过程，则被调过程表为空。

```
#1 10 EDITOR = INIT, READ_LINE, WRITE_LINE, INPUT, TOP, ENTER, UP,
              NEXT, LIST, DELETE, COPY, STORE, INSERT, FIND, REPLACE
#2 20 INIT
#3 20 READ_LINE = WRITE_LINE
#4 20 WRITE_LINE
#5 20 INPUT = WRITE_LINE, GO_EDIT, CONNECT, INSERT_CHAIN
#6      30 GO_EDIT = WRITE_LINE
#7      30 CONNECT
#8      30 INSERT_CHAIN
#9 20 TOP
#10 20 ENTER = WRITE_LINE
#11 20 UP = WRITE_LINE, GET_NUMBER
#12      30 GET_NUMBER = WRITE_LINE, CONVERT
#13      40 CONVERT
#14 20 NEXT = WRITE_LINE, GET_NUMBER
#15 20 LIST = WRITE_LINE, GET_NUMBER
#16 20 DELETE = WRITE_LINE, GET_NUMBER, DELETE_LINK
#17      30 DELETE_LINK = WRITE_LINE
#18 20 COPY = WRITE_LINE, GET_UNMBER, COPY_LINK
#19      30 COPY_LINK = WRITE_LINE
#20 20 STORE = WRITE_LINE, GET_NUMBER, DELETE_LINK
#21 20 INSERT = INSERT_CHAIN
#22 20 FIND = WRITE_LINE, GET_STRING1, COMP
#23      30 GET_STRING1 = WRITE_LINE
#24      30 COMP
#25 20 REPLACE = WRITE_LINE, GET_NUMBER, GET_STRING1, COMP,
               GET_STRING2
#26      30 GET_STRING2 = WRITE_LINE
```

3.12.5.2 类 PASCAL 伪码

由于技术背景、习惯和爱好的不同，人们在描述算法时使用的伪码也并不相同。我们在详细设计这个编辑程序的过程中，使用了一种类似 PASCAL 语言的伪码表示算法。为便于读

者阅读，下面简要介绍我们所使用的伪码。

1. 运算符

除了人们在 PASCAL 语言中已经熟悉的基本运算符之外，针对字符串类型的数据，还提供了下述运算符。

X Cat Y : 连接两个字符串 X 和 Y ;
Length (X) : 得到字符串 X 的长度 ;
Substr (X , I , J) : 从字符串 X 的第 I 个字符开始，取出一个长度为 J 的子字符串 ;
Letter (X) : 测试字符串 X 中每个字符是否都是字母 ;
Digit (X) : 测试字符串 X 中的每个字符是否都是数字 ;
Match (X , Y) : 测试字符串 X 与字符串 Y 是否能匹配，若能匹配则返回 1 ;
String_conv (N) : 把数值 N 转变成字符串 ;
Number_conv (X) : 把字符串 X 转变成数值。

2. 控制结构

在本伪码中常用的控制结构（或称为控制语句）主要有：Block 语句，Set 语句，If 语句，Case 语句，Loop 语句，Exitloop 语句，Call 语句，Return 语句和 Unwind 语句。下面简要地介绍这些控制语句。

Block 语句把数据流语句（赋值语句，读语句，写语句等）聚集成一个程序块，以显式地表明它是单入口单出口的顺序结构。Block 语句用控制保留字“Block”开头，用控制保留字“Endblock”结尾。

Set 语句用来设置开关（即控制数据）的状态，它的一般形式是：

Set 开关名 To 状态

If 语句是常用的分支结构，它有两种形式，一种有“Else”部分，另一种没有“Else”部分。即，
If...Then...Else...Endif

或

If...Then...Endif

Case 语句是多分支结构，它也有两种常用的形式，一种形式是：

Case exp Of

 value_1 S₁ ;

 value_2 S₂ ;

 value_n S_n ;

 Else : S_{n+1}

Endcase ;

其中 exp 是一个表达式，value_1 ~ value_n 是这个表达式可能取的值，S₁ ~ S_{n+1} 是不同的语句表。当 exp 取某个值 value_i 时，则做与之对应的语句表 S_i；如果 exp 的取值和 value_1 ~ value_n 中任一个都不同，则做与 Else 对应的语句表 S_{n+1}。

Case 语句的第二种形式是：

Case value Of

 exp_1 S₁ ;

```
exp_2 S2;
```

```
exp_n Sn;
```

```
Endcase;
```

其中 exp_1 ~ exp_n 是不同的表达式，value 是一个值或表达式，当 value 和某个 exp_i 相等时，则执行相应的语句表 S_i。

Loop 语句是循环结构，它用保留字“Loop”和“Endloop”标志循环结构的开始和结束。常用 Exitloop 语句终止循环，把控制转移到紧接在这个循环结构后面的语句（即，接在保留字“Endloop”后面的语句）。此外，循环结束条件还可以用“while 逻辑表达式”的形式表示（相当于 DO WHILE 型循环结构），也可以用“Until 逻辑表达式”的形式表达（相当于 DO_UNTIL 型循环结构）。

Call 语句是模块调用语句。Call 语句把控制转移到被调用的 Procedure 的入口；Return 语句用于从被调用的 Procedure 返回，它把控制从被调用模块归还给调用模块中紧接在 Call 语句后面的语句。Return 语句有两种形式，一种形式没有回送值，仅仅归还控制；另一种形式有回送值，回送值写在保留字“Return”后面的括号中。

通常由被调用模块返回到直接调用它的模块，必要时可以使用 Unwind 语句（即，直接返回语句），从被调用模块返回到某个间接调用它的模块，这个间接调用它的模块需要在 Unwind 语句中指定。因此，Unwind 语句的一般形式是：

```
Unwind To 模块名
```

执行这个语句则由被调用模块直接返回到由“模块名”指定的那个模块。直接返回通常用于错误出口。

3.12.5.3 实现编辑程序的算法

下面给出用类 PASCAL 伪码描述的编辑程序中每个过程的实现算法。

```
#1 Procedure EDITOR; /*主过程*/
    Call INIT;
    Loop
        Call READ_LINE;
        Block
            I:=1;
        Endblock;
        If IND=0
        Then /*在输入模式中*/
            Call INPUT;
        Else /*在编辑模式中*/
            Block
                CHAR:=Substr(IN,I,1);
            Endblock;
            Case CHAR Of
```

```
'T':  
    Call TOP;  
'U':  
    Block  
        I:=I+1;  
    Endblock;  
    Call UP;  
'N':  
    Block  
        I:=I+1;  
    Endblock;  
    Call NEXT;  
'E':  
    Call ENTER;  
'L':  
    Block  
        I = I+1;  
    Endblock;  
    Call LIST;  
'D':  
    Block  
        I:=I+1;  
    Endblock;  
    Call DELETE;  
'R':  
    Block  
        I:=I+1;  
    Endblock;  
    Call REPLACE;  
'F':  
    Block  
        I:=I+1;  
    Endblock;  
    Call FIND;  
'I':  
    Call INSERT;  
'C':  
    Block  
        I:=I+1;
```

```

        Endblock ;
        Call  COPY ;
'S':
        Block
          I:= I + 1 ;
        Endblock ;
        Call  STORE ;
'Q':
        Exitloop ;
Else :
        Block
          IN:= '? ! ' ;
        Endblock ;
        Call  WRITE_LINE ;
      Endcase ;
    Endif ;
  Endloop
End EDITOR ;

#2 Procedure INIT ; /*初始化正文编辑程序*/
  Set CMP , CHG , ERR  To 0 ;
  Set IND To  0 ; /*编辑模式*/
  Block
    IN  = 'EDIT ' ;
  Endblock ;
  Call WRITE_LINE ;
  Block
    BPTR  = TPTR  = 0 ; /*无正文链*/
    KPTR  = QPTR  = 0 ; /*无工作链*/
  Endblock ;
  把正文文件链接成一条自由链，用指针 FPTC 指向链中第一行 ;
  把自由链第一行的上链指针和最后一行的下链指针都置成 0 ;
End INIT ;

#3 Procedure READ_LINE; /*从终端接收一行正文并且把它存到缓冲器 IN 中*/
  Block
    IN:= '' ; /*把缓冲器 IN 初始化为全是空格*/
    I:= 1;
  Endblock ;
  Loop /*每循环一次读进一个字符或一个汉字*/
  Block

```

软件工程

```
Read ( Substr ( BUF , 1 , 1 ));  
Endblock ;  
If Ord ( Substr ( BUF,1,1 ) ) > 127 /*汉字由两字节编码，每字节最高位均为1*/  
Then  
    Block /*是汉字时同时读入两个字节*/  
        Read ( Substr ( BUF , 2 , 1 ));  
    Endblock ;  
Endif ;  
Case Substr ( BUF , 1 , 1 ) Of  
    '@': /*删除一个字符或一个汉字的命令*/  
        If I = 1  
            Then  
                If Ord ( Substr ( IN , I-1 , 1 )) > 127  
                    Then /*是汉字时同时删掉两个字节*/  
                        Block /*把这两个字节恢复为空格*/  
                            I:= I - 2 ;  
                            Substr ( IN , I , 2 ) := '' ;  
                        Endblock ;  
                    Else /*不是汉字时只删掉一个字节*/  
                        Block  
                            I = I - 1;  
                            Substr ( IN , I , 1 ) := '' ;  
                        Endblock ;  
                Endif ;  
            Endif ;  
        '#': /*行删除命令*/  
            Block  
                IN:= '' ; /*把缓冲器 IN 复原成全是空格*/  
                I:= 1 ;  
            Endblock ;  
        '!': /*换行字符*/  
            If I = 1  
                Then /*仅当换行符为这行的第一个字符时才存入*/  
                    Block  
                        Substr ( IN , I , 1 ) := Substr ( BUF , 1 , 1 );  
                    Endblock ;  
                Endif ;  
            Return ;  
        Else:
```

```

If ( Ord ( Substr ( BUF,1,1 ) ) > 127 ) And ( I < LINESIZE )
Then /*是汉字时把两个字节同时存入 IN 中*/
    Block
        Substr ( IN,I,2 ) = Substr ( BUF,1,2 );
        I:=I+2;
    Endblock ;
Else
    If ( Ord ( Substr ( BUF,1,1 ) ) > 127 ) And ( I < LINESIZE )
    Then /*不是汉字时每次存一个字节*/
        Block
            Substr ( IN , I , 1 ) = Substr ( BUF , 1 , 1 );
            I:=I+1 ;
        Endblock ;
    Else /*输入字符串长度超过了规定长度*/
        Block
            TMP:=IN ;
            IN:= ' ? !';
        Endblock ;
        Call WRITE_LINE ;
        Block
            IN = TMP ; /*恢复 IN 的内容*/
        Endblock ;
        Return ;
    Endif ;
Endif ;
Endcase ;
Endloop ;
End READ_LINE ;
#4 Procedure WRITE_LINE ;
/*把缓冲器 IN 的内容输出到终端*/
Block
    I:=1 ;
Endblock ;
Loop /*每循环一次输出一个字符*/
    Block
        Substr ( BUF,1,1 ) := Substr ( IN,I,1 );
        Write ( Substr ( BUF,1,1 ) );
        I:=I+1 ;
    Endblock ;

```

```

If I > Length ( IN )
Then
    Exitloop ;
Endif ;
Endloop ;
Block
    BUF:= '!' ; /* '!' 代表换行符 */
    Write ( BUF );
Endblock ;
End WRITE_LINE ;
#5 Procedure INPUT ;
/*从缓冲器 IN 取来一行正文放到正文文件中*/
Block
    CHAR = Substr ( IN , I , 1 );
Endblock ;
If IND = 1 /*尚未接收到正文*/
Then
    If CHAR = '!'
        Then /*空行*/
            Call GO_EDIT ; /*转到编辑模式*/
        Else /*不是空行*/
            If FPTR = 0
                Then /*自由链是空的*/
                    Block
                        IN:= 'NOFREE' ;
                    Endblock ;
                    Call WRITE_LINE ;
                Else /*存到自由链第一个自由项中*/
                    Block
                        XPTR:= FPTR ;
                        TEXT.S ( XPTR ) :=IN ;
                    Endblock ;
                    Set IND To 2 ;
                Endif ;
            Endif ;
    Else /*至少已经接收过一行正文*/
        If CHAR = '!'
            Then /*空行，从自由链把正文移往正文链并转到编辑模式*/
                Call CONNECT ;

```

```

Call INSERT_CHAIN ;
Call GO_EDIT ;
Else /*一行正文*/
  If TEXT.N ( XPTR ) = 0
    Then /*再也没有自由空间了，这行相当于是空行*/
      Call CONNECT ;
      Call INSERT_CHAIN ;
      Block
        IN:= 'NOFREE';
      Endblock ;
      Call WRITE_LINE ;
      Call GO_EDIT ;
    Else /*把这行正文放入自由链的下一项*/
      Block
        XPTR:= TEXT.N ( XPTR ); /*转到下一个自由行*/
        TEXT.S ( XPTR ) := IN ; /*填入一行正文*/
      Endblock ;
      Endif ;
    Endif ;
  Endif ;
End INPUT ;

#6 Procedure GO_EDIT ; /*转到编辑模式*/
  Set IND To 0
  Block
    IN:= 'EDIT';
  Endblock ;
  Call WRITE_LINE ;
End GO_EDIT ;

#7 Procedure CONNECT ;
/*从自由链中分离出输入的正文，分别用 YPTR 和 XPTR 指向它的第一行和最后一行
 */
  Block
    YPTR:= FPTR ;
    /*把新正文的起始位置存放在缓冲器 YPTR 中*/
    FPTR:= TEXT.N ( XPTR );
    /*把 FPTR 置成指向剩余自由链的第一项*/
  Endblock ;
  If FPTR = 0
    Then /*更新自由链的上链指针*/

```

```

Block
TEXT.U ( FPTR ) := 0 ;
Endblock ;
Endif ;
End CONNECT ;
#8 Procedure INSERT_CHAIN ;
/*把新输入的正文加到正文链中，紧接着在当前行的后面。开始时 YPTR 指向输入
正文的第一行，XPTR 指向最后一行。指针 BPTR 把原有的正文分成两部分，
分别称为前半和后半*/
If BPTR = 0 /*指向虚拟行*/
Then /*新输入的正文作为正文文件的起始部分*/
If TPTR = 0 /*原没有正文链，前半和后半全为空。把插入指针 SPTR 置成顶
指针*/
Then
Block
SPTR = TPTR ; /*TPTR 是 0*/
Endblock ;
Else /*仅前半为空*/
Block /*新正文插在第一行前面*/
TEXT.U ( TPTR ) := XPTR ; /*链接新正文的最后一行和原有正文
的第一行*/
SPTR := TPTR ; /*SPTR 存放新正文插入前指向第一行正文的指针*/
Endblock ;
Endif ;
Block
TPTR := YPTR ; /*新正文的第一行成为正文链的第一行*/
Endblock ;
Else /*当前行不是虚拟行*/
Block /*链接输入正文的前面部分*/
SPTR := TEXT.N ( BPTR ) ; /*保存指向后半第一行的指针*/
TEXT.U ( YPTR ) := BPTR ;
/*新正文第一行的上链指针指向原有正文前半的最后一行*/
TEXT.N ( BPTR ) := YPTR ;
/*原有正文前半的最后一行的下链指针指向新正文的第一行*/
Endblock ;
If SPTR 0 /*后半不是空的*/
Then /*链接原有正文后半的第一行和新正文的最后一行*/
Block
TEXT.U ( SPTR ) := XPTR ;

```

```

        Endblock ;
    Endif ;
    Endif ;
    Block
    TEXT.N ( XPTR ) := SPTR ;
    /*把新正文最后一行和原有下文的后半链接起来。如后半是空的，这下链指针将为
    零*/
    BPTR:=XPTR ; /*新正文的最后一行是新的当前行*/
    Endblkok ;
End INSERT_CHAIN ;
#9 Procedure TOP ; /*使 BPTR 指向虚拟行*/
    Block
        BPTR:= 0 ;
    Endblock ;
End TOP ;
#10 Procedure ENTER ; /*进入输入模式*/
    Set IND To 1 ;
    Block
        IN:= ' INPUT ' ;
    Endblock ;
    Call WRITE_LINE ;
End ENTER ;
#11 Procedure UP /*把指针 BPTR 从当前行往上移 N 行*/
    If Call GET_NUMBER = 1 /*不成功返回*/
    Then
        Return ;
    Endif ;
    CTR:= NUM ; /*调用 GET_NUMBER 得到的数在 NUM 中*/
    If T PTR = 0 /*正文链为空*/
    Then
        Block
            IN:= ' NOTEXT ' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Endif ;
    If BPTR = 0 /*在虚拟行*/
    Then
        Block

```

```

        IN:= ' TOF ' ;
    Endblock ;
    Call WRITE_LINE ;
    Return ;
Endif ;
Loop
If CTR < 0 /*继续上移 BPTR*/
Then /*往上移一行*/
    Block
        BPTR:= TEXT.U ( BPTR ) ;
        CTR:= CTR - 1 ;
    Endblock ;
    If BPTR = 0
    Then
        Block
            IN:= 'TOF' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Endif ;
Else /*已经移了 N 行*/
    Exitloop ;
Endif ;
Endloop ;
Block
    IN:= TEXT.S ( BPTR ); /*取当前行*/
Endblock ;
Call WRITE_LINE ; /*印出当前行*/
End UP ;
# 12 Procedure GET_NUMBER; /*从编辑命令中取来数 N , 把它存在缓冲器 NUMD 中。
                           如果 N 是空格则在过程 CONVERT 中把补缺值置为 1*/
Set ERR To 0 ;
If Substr ( IN,I,1 ) = '' /*命令字母后面的字符应是空格,否则命令是错的*/
Then
    Block
        I:= I + 1 ;
    Endblock ;
Else
    Block

```

```

    IN: = ' ? ! ' ;
Endblock ;
Call WRITE_LINE ;
Return ( 1 ); /*不成功返回 */
Endif ;
Block
    NUMD: = '' ; /*清除缓冲器*/
Endblock ;
Loop /*在 NUMD 中装配数 N */
If Length ( NUMD ) = 4 Or I > LINESIZE
Then
    Exitloop ;
Else
    If Not Digit ( Substr ( IN , I , 1 ) )
    Then /*该字符不是数字*/
        Exitloop ;
    Else
        Block
            NUMD: = NUMD Cat Substr ( IN , I , 1 ) /*装配这个数*/
            I: = I + 1
        Endblock ;
    Endif ;
    Endif ;
Endloop ;
If I < LINESIZE And Substr ( IN,I,1 ) '' /*跟在这个数后面的字符应是空格*/
Then
    Block
        IN: = ' ? ! ' ;
    Endblock ;
    Call WRITE_LINE ;
    Return ( 1 ); /*不成功返回 */
Endif ;
Call CONVERT ; /*把字符形式的数转变成内部形式的十进制数 */
If ERR = 1 /*转变错误*/
Then
    Block
        IN: = ' ? ! ' ;
    Endblock ;
    Call WRITE_LINE ;

```

```

        Return ( 1 ); /*不成功返回*/
    Else
        Return ( 0 ); /*成功返回*/
    Endif ;
End GET_NUMBER ;
# 13 Procedure CONVERT ;
    /*这个过程把缓冲器 NUMD 中用字符形式表示的十进制数转变成十进制数的内部表示
形式，并存放到缓冲器 NUM 中。如果 NUMD 中全部是空格，则在 NUM 中放补缺值 1。
如果 NUM 的值为 0，则把开关 ERR 置成 1 以指示错误；否则把它置为 0 表明转变成功
*/
Buffer
    DIGITVALUE Of Number , /*说明局部数据*/
Set ERR To 0 ;
If NUMD = ''
Then
    Block
        NUM: = 1 ;
    Endblock ;
Else
    Block
        NUM: = 0 ;
        M: = 1 ;
    Endblock ;
Loop
If M = 4 And Digit ( Substr ( NUMD , M , 1 ) )
Then
    Block
        DIGITVALUE: = ORD ( NUMD [ M ] - ORD ( '0' );
        NUM: = NUM *10 + DIGITVALUE ;
        M: = M + 1 ;
    Endblock ;
Else
    Exitloop ;
Endif ;
Endloop ;
If NUM = 0 /*非法的数*/
Then
    Set ERR To 1 ;
Endif ;

```

```
Endif ;
End CONVERT ;

#14 Procedure NEXT ; /*把指针 BPTR 从当前行往下移 N 行*/
If Call GET_NUMBER = 1 /*不成功返回*/
Then
    Return ;
Endif ;
CTR:= NUM ; /*调用 GET_NUMBER 得到 NUM 中的数*/
If BPTR = 0 /*在虚拟行*/
Then
    If T PTR = 0 /*没有正文*/
    Then
        Block
        IN:= 'NOTEXT' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Else
        Block /*把 BPTR 移到第一行*/
        BPTR:= T PTR ;
        CTR:= CTR - 1 ;
        Endblock ;
    Endif ;
Endif ;
Loop While CTR > 0
If TEXT.N ( BPTR ) = 0 /*最后一行*/
Then
    Block
    IN:= 'EOF' ;
    Endblock ;
    Call WRITE_LINE ;
    Return ;
Endif ;
Block
    BPTR:= TEXT.N ( BPTR ); /*把 BPTR 往下移一行*/
    CTR:= CTR - 1 ;
Endblock ;
Endloop ;
Block ;
```

软件工程

```
IN:= TEXT.S ( BPTR ); /*取当前行*/
Endblock ;
Call WRITE_LINE ;
End NEXT ;

# 15 Procedure LIST ; /*输出正文链中从当前行开始的 N 行正文*/
If Call GET_NUMBER = 1 /*不成功返回*/
Then
    Return ;
Enif ;
CTR = NUM ; /*调用 GET_NUMBER 得到 NUM 中的数*/
If BPTR = 0 /*在虚拟行*/
Then
    If T PTR = 0 /*没有正文*/
    Then
        Block
            IN: = 'NOTEXT' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Else
        Block
            BPTR: = T PTR ; /*移到第一行*/
        Endblock ;
    Endif ;
Endif ;
Block
    IN: = TEXT.S ( BPTR ); /*取当前行*/
    CTR: = CTR - 1 ;
Endblock ;
Call WRITE_LINE ; /*印出当前行*/
Loop While CTR > 0 /*继续印出正文*/
If TEXT.N ( BPTR ) = 0
Then
    Block
        IN: = 'EOF' ;
    Endblock ;
    Call WRITE_LINE ;
    Return ;
Endif ;
```

```

Block
    BPTR:= TEXT.N ( BPTR ); /*下移一行*/
    IN:= TEXT.S ( BPTR );
    CTR:= CTR - 1 ;
Endblock ;
Call WRITE_LINE ; /*印出该行正文*/
Endloop ;
End LIST ;

# 16 Procedure DELETE ; /*删掉正文链中从当前行开始的 N 行正文，把它们放在自由链的前面。如果 N 是空格，则补缺值为 1*/
If Call GET_NUMBER = 1 /*数 N 错*/
Then
    Return ;
Endif ;
CTR:= NUM ; /*调用 GET_NUMBER 得到的数在 NUM 中*/
If BPTR = 0 /*在虚拟行*/
Then
    If T PTR = 0 /*没有正文*/
    Then
        Block
            IN:= 'NOTEXT' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Else
        Block
            BPTR:= T PTR ; /*移到第一行*/
        Endblock ;
    Endif ;
Endif ;
Call DELETE_LINK ; /*从正文链上把删除的字块分离下来，使指针 X PTR 指向它的最后一行，指针 Y PTR 指向它的第一行*/
Block /*把这些字块放进自由链*/
    TEXT.N ( X PTR ) := F PTR ; /*被删除的最后一行的下链指针指向自由链第一行*/
Endblock ;
If F PTR 0 /*自由链原来不是空的*/
Then
    Block
        TEXT.U ( F PTR ) := X PTR ; /*自由链原上链指针指向放进来的最后一行*/

```

```

Endblock ;
Endif ;
Block
    TEXT.U ( YPTR ) := 0 ;
        /*为了表明是最上面的一行，在新放进来的正文的第一行的上链指针中放零*/
        FPTR:=YPTR ; /*指向新放入的第一行，这行已经成为新自由链的第一行*/
    Endblock ;
End DELETE ;

# 17 Procedure DELETE_LINK ;
    /*从正文链中把要删除的那些行分离下来。使 YPTR 和 XPTR 分别指向被删除的第一行和最
后一行。过程 DELETE 和 STORE 调用这个过程*/
Block /*使头指针和尾指针在开始时都指向当前行*/
    YPTR:=BPTR ;
    XPTR:=BPTR ;
    CTR:=CTR - 1 /*因为已分离下当前行*/
Endblock ;
Loop While CTR > 0 And TEXT.N ( XPTR ) = 0
    /*CTR 中存放要删除的行数*/
Block /*移动 XPTR 使它指向被删除的最后一行*/
    XPTR:=TEXT.N ( XPTR );
    CTR:=CTR - 1 ;
Endblock ;
Endloop ;
If TEXT.N ( XPTR ) = 0 /*情况 ( a ) 或 ( b ): 被删除的字块在正文链的尾部*/
Then
    If CTR > 0 /*要求删除的比能够删除的多*/
    Then
        Block
            IN:= ' EOF ';
        Endblock ;
        Call WRITE_LINE ;
    Endif ;
    If TEXT.U ( YPTR ) = 0 /*情况 ( a ): 已经把正文链全都删除了*/
    Then
        Block /*把正文链指针置成零*/
            T PTR:= 0 ;
            B PTR:= 0 ;
        Endblock ;
    Else /*情况 ( b ): 还有正文链*/

```

```

Block
    BPTR:=TEXT.U ( YPTR ) ; /*当前行指针指向剩余正文的最后一行*/
    TEXT.N ( BPTR ) := 0 ;
    /*把最后一行的下链指针置成零，以表明它是最后一行*/
    Endblock ;
    Endif ;
Else /*情况(c)或(d)：被删除的字块不在正文文件的尾部*/
    Block /*使被删除的字块下面的那一行成为新的当前行*/
        BPTR:=TEXT.N ( XPTR );
        Endblock ;
        If TEXT.U ( YPTR ) = 0 /*情况(c)：被删除的字块从正文文件的第一行
开始*/
        Then
            Block
                T PTR:=BPTR ; /*重置顶指针*/
                TEXT.U ( BPTR ) := 0 ; /*重置正文文件第一行的上链指针*/
                Endblock ;
            Else /*情况(d)：被删除的字块不是从正文文件的第一行开始*/
                Block
                    SPTR:=TEXT.U ( YPTR ); /*SPTR 指向被删除的字块的上面那行正
文*/
                    TEXT.N ( SPTR ) := BPTR ; /*重置被删除字块上面那行的下链指针*/
                    TEXT.U ( BPTR ) := SPTR ; /*重置被删除字块下面那行的上链指针*/
                Endblock ;
            Endif ;
        Endif ;
    End DELETE_LINK ;
# 18 Procedure COPY ;
    /*把正文链中从当前行开始的 N 行正文复制到自由链中，然后把它们移到工作链的尾部。这
N 行正文并不从正文链中删除。如果 N 是空格，则补缺值为 1*/
    If Call GET_NUMBER = 1 /*不成功返回*/
    Then
        Return ;
    Endif ;
    CTR:=NUM ; /*调用 GET_NUMBER 得到的数在 NUM 中*/
    If BPTR = 0 /*在虚拟行*/
    Then
        If T PTR = 0 /*无正文*/
        Then

```

```
Block
    IN = 'NOTEXT';
Endblock;
Call WRITE_LINE;
Return;

Else
    Block
        BPTR:=TPTR; /*用第一行作为当前行*/
    Endblock;
    Endif;
Endif;

Block
    YPTR:=BPTR; /*使头指针等于当前行指针*/
    XPTR:=BPTR; /*开始时尾指针也等于当前行指针*/
    SPTR:=FPTR; /*存自由指针*/
Endblock;
If SPTR=0 /*自由链是空的*/
Then
    Block
        IN:='NOFREE';
    Endblock;
    Call WRITE_LINE;
    Return;
Endif;
Call COPY_LINK; /*把 N 行正文复制到自由链中*/
If SPTR=0 /*自由字块不够多，不能满足复制命令的要求，忽略这个命令*/
Then
    Return;
Endif; /*现在 FPTR 和 SPTR 分别指向自由链中正文副本的起始行和最后一行。KPTR 和 QPTR 分别指向工作链的起始行和最后一行*/
If QPTR=0
Then /*工作链是空的*/
    Block /*使工作链的超始行等于正文副本的第一行*/
        KPTR:=FPTR;
    Endblock;
Else /*工作链不空*/
    Block
        TEXT.U (FPTR) := QPTR; /*把正文副本链接到工作链的尾部*/
        TEXT.N (QPTR) := FPTR; /*把工作链的尾部和正文副本第一行链接

```

```

        起来* /
    Endblock ;
Endif ;
Block
    QPTR:=SPTR ; /*使 QPTR 指向新工作链的最后一行*/
    FPTN:= TEXT.N ( SPTR ) ; /*重置自由链指针*/
    TEXT.N ( QPTR ) = 0 ; /*使工作链最后一行的下链指针为零*/
Endblock ;
If FPTN 0
Then /*自由链不是空的*/
    Block
        TEXT.U ( FPTN ) := 0 ; /*清除自由链的上链指针*/
    Endblock ;
Endif ;
End COPY ;

# 19 Procedure COPY_LINK ;
/*把 N 行正文复制到自由链中。开始时 SPTR 指向第一个自由行，YPTN 和 XPTN 指向当前行。最后，SPTR 指向自由链副本的最后一行，YPTN 和 XPTN 分别指向被复制的正文的第一行和最后一行*/
Loop /*复制 N 行正文到自由链中*/
    Block /*把一行正文从正文链复制到自由链*/
        TEXT.S ( SPTR ) := TEXT.S ( XPTN );
        CTR:= CTR - 1 ;
    Endblock ;
    If CTR 0 And TEXT.N ( XPTN ) 0
    Then /*还有正文需复制*/
        Block /*使 XPTN 指向下一行应该复制的正文*/
            XPTN:= TEXT.N ( XPTN );
        Endblock ;
    Else
        Exitloop ; /*现在 XPTN 指向被复制正文的最后一行*/
    Endif ;
    If TEXT.N ( SPTR ) = 0 /*现在自由链是空的*/
    Then
        Block
            IN:= 'NOFREE' ;
            SPTR:= 0 ;
        Endblock ;
        Call WRITE_LINE ;

```

```

Return ;
Endif ;
Block
    SPTR:= TEXT.N ( SPTR );
Endblock ;
Endloop ; /*现在SPTR指向自由链副本的结尾，X PTR指向被复制的正文的结尾*/
If TEXT.N ( X PTR ) = 0
Then /*已经复制到正文链尾*/
    If CTR > 0
        Then /*命令要求复制的行数多于可以得到的行数，给出一个信息提醒用户*/
            Block
                IN:= 'EOF' ;
            Endblock ;
            Call WRITE_LINE ;
        Endif ;
        Block
            BPTR:= X PTR ; /*使被复制的正文的最后一行成为当前行*/
        Endblock ;
    Else /*尚未复制到正文链尾*/
        Block /*使BPTR指向被复制正文后面那一行*/
            BPTR = TEXT.N ( X PTR );
        Endblock ;
    Endif ;
End COPY_LINK ;
#20 Procedure STORE ;
/*把正文链中从当前行开始的N行正文同其他正文分离开，然后把它们移到工作链的尾部。
如果N是空格，则它的补缺值为1*/
If Call GET_NUMBER = 1
Then /*不成功返回*/
    Return ;
Endif ;
CTR:= NUM ; /*数N在NUM中*/
If BPTR = 0 /*在虚拟行*/
Then
    If T PTR = 0 /*无正文*/
        Then
            Block
                IN = 'NOTEXT' ;
            Endblock ;

```

```

        Call WRITE_LINE ;
        Return ;
    Else
        Block /*用第一行作为当前行*/
        BPTR:=TPTR ;
        Endblock ;
        Endif ;
    Endif ;
    Call DELETE_LINK /*把正文链中从当前行开始的 N 行正文分离下来*/
    Block /*把需要存储的那块正文的头和工作链的尾链接起来*/
        TEXT.U ( YPTR ) := QPTR ; /*QPTR 指向工作链最后一行*/
        Endblock
        If QPTR = 0
            Then /*开始时工作链是空的*/
                Block /*存进的正文的第一行就是新工作链的第一行*/
                KPTR:=YPTR ;
                Endblock ;
            Else /*工作链原来不是空的*/
                Block /*存储的正文链接在工作链尾*/
                TEXT.N ( QPTR ) := YPTR ;
                Endblock ;
            Endif ;
            Block
                TEXT.N ( XPTR ) = 0 ; /*把存储的正文的最后一行的下链指针置成零,
                                         以表明是最后一行*/
                QPTR:=XPTR ; /*新工作链的尾*/
                Endblock ;
            End STORE ;
# 21 Procedure INSERT ;
/*把工作链中的全部正文都插到正文链中，紧接在当前行后面*/
If KPTR = 0
Then /*工作链是空的*/
    Return ; /*没有正文可插入*/
Endif ;
Block
    YPTR:=KPTR ; /*工作链的头*/
    XPTR:=QPTR ; /*工作链的尾*/
    KPTR:=QPTR = 0 /*工作链已经成为空的了*/
Endblock ;

```

```

Call INSERT_CHAIN ; /*插到正文链中去*/
End INSERT ;

#22 Procedure FIND ;
/*从当前行下面一行开始扫描正文链，使当前行指针指向第一次出现字符串 STRING1 的那一行
 */
If Call GET_STRING1 = 1
Then /*不成功返回*/
    Retrun ;
Endif ;
If T PTR = 0 /*没有正文*/
Then
    Block
        IN: = 'NOTE TEXT' ;
    Endblock ;
    Call WRITE_LINE ;
    Return ;
Endif ;
If B PTR = 0
Then /*在虚拟行*/
    Block /*从第一行开始*/
        B PTR: = T PTR ;
    Endblock ;
Else
    If TEXT.N ( B PTR ) = 0
    Then /*当前行不是正文链的最后一行，从下一行开始扫描*/
        Block
            B PTR: = TEXT.N ( B PTR );
        Endblock ;
    Else
        Block
            IN: = 'EOF' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Endif ;
Endif ;
Loop /*每循环一次扫描一行正文*/
Block /*把当前这行正文移到缓冲器 IN 中*/
    IN: = TEXT.S ( B PTR );

```

```

I:=1 ;
Endblock ;
Loop /*每循环一次比较这行正文中的一个子字符串*/
Call COMP ; /*比较缓冲器 MAT 中的字符串和缓冲器 IN 中的子字符串,
如果两者相同则把开关 CMP 置成 1*/
If CMP = 1
Then /*已匹配*/
Set CMP To 0 ; /*复原成初始状态*/
Call WRITE_LINE ; /*印出找到的这行正文*/
Return ;
Else
Block
I = I + 1
Endblock ;
Endif ;
If I > LINESIZE
Then /*没找到*/
Exitloop
Endif ;
Endloop ;
If TEXT.N ( BPTR ) = 0
Then /*不在正文链尾，把当前行指针下移一行*/
Block
BPTN = TEXT.N ( BPTR );
Endblock ;
Else /*在正文链尾*/
Exitloop ;
Endif ;
Endloop ;
Block
IN: = 'EOF' ;
Endblock ;
Call WRITE_LINE ;
End FIND ;

# 23 Procedure GET_STRING1 ;
/*从命令中取得第一个字符串，并且把它存在缓冲器 MAT 中*/
If Substr ( IN , I , 1 ) =
Then /*缓冲器 IN 中的第二个字符不是空格，命令形式错，出错返回*/
Block

```

```

IN = '? !';
Endblock
Call WRITE_LINE ;
Return ( 1 ); /*不成功返回*/
Endif ;
If Substr ( IN , I = I + 1,1 ) = '/'
Then /*缓冲器 IN 中的第三个字符不是斜线，命令形式错*/
    Block
        IN:= '? !';
    Endblock ;
    Call WRITE_LINE ;
    Return ( 1 ); /*不成功返回*/
Endif ;
If Substr ( IN , I:= I + 1 , 1 ) = '/'
Then /*STRING1 是空字符串，命令错*/
    Block
        IN:= '? !';
    Endblock
    Call WRITE_LINE ;
    Return ( 1 ); /*不成功返回*/
Endif ;
Block
    J = 1 ;
Endblock ;
Loop /*每循环一次执行一次斜线匹配*/
If Match ( '/' , Substr ( IN , I , 1 )) = 1
Then /*找到第二条斜线，字符串 STRING1 结束*/
    Exitloop ;
Else /*尚未找到第二条斜线*/
    If I = LINESIZE
    Then /*遗漏第二条斜线，命令错*/
        Block
            IN:= '? !';
        Endblock ;
        Call WRITE_LINE ;
        Return ( 1 );/*不成功返回*/
    Else /*移到下一个字符*/
        Block
            I:= I + 1 ;

```

```

J:= J + 1 ; /*最后 , J - 1 代表 STRING1 的长度*/
Endblock ;
Endif ;
Endif ;
Endloop ;
Block
    MAT:= Substr ( IN , I - J + 1 , J - 1 ); /*把 STRING1 存到 MAT 中*/
    I:= I + 1 ; /*使 I 指向第二条斜线后面的第一个字符*/
Endblock ;
Return ( 0 ); /*成功返回*/
End GET_STRING1 ;

# 24 Procedure COMP ;
/*这个过程比较缓冲器 IN 中从第 I 个字符开始的子字符串和缓冲器 MAT 中的字符串。开关
CMP 指示比较的结果 , 状态 1 和 0 分别表明匹配和不匹配*/
If ( LINESIZE - ( I - 1 ) < Length ( MAT ) )
Then /*缓冲器 IN 中剩余的子字符串比缓冲器 MAT 中的字符串短 , 不可能匹配*/
    Set CMP To 0 ;
    Return ;
Endif ;
If Match ( MAT , Substr ( IN , I , Length ( MAT )) ) = 1
Then /*匹配*/
    Set CMP To 1 ; /*指出匹配*/
Else
    Set CMP To 0 ; /*表明不匹配*/
Endif ;
End COMP ;

# 25 Procedure REPLACE ;
/*在从当前行开始的 N 行正文中 , 每出现一个字符串 STRING1 , 就用字符串 STRING2 替换它
*/
If Call GET_STRING1 = 1
Then /*不成功返回*/
    Return ;
Endif ;
/*STRING1 已经在缓冲器 MAT 中*/
If Call GET_STRING2 = 1
Then /*不成功返回*/
    Return ;
Endif ;
/*STRING2 在缓冲器 REP 中*/

```

```

If Call GET_NUMBER = 1
Then /*不成功返回*/
    Return ;
Endif ;
/*数 N 在缓冲器 NUM 中*/
If BPTR = 0
Then /*在虚拟行*/
    If T PTR = 0
    Then /*无正文*/
        Block
            IN:= 'NOTEXT' ;
        Endblock ;
        Call WRITE_LINE ;
        Return ;
    Else
        Block
            B PTR:= T PTR ; /*第一行作为当前行*/
        Endblock ;
    Endif ;
Endif ;
CTR:= NUM ; /*把行数 N 存到计数器 CTR 中*/
Loop /*每循环一次编辑一行正文*/
    While CTR > 0
        Set CHG To 0 ; /*初始化*/
        Block /*取一行正文*/
            IN:= TEXT.S ( B PTR );
            I:= 1 ;
        Endblock ;
        Loop /*每循环一次在这行正文中找一个替换*/
        While I LINESIZE
            Call COMP ; /*比较 IN 中的子字符串和 MAT 中的字符串，如果匹配就把
开关 CMP 置成 1*/
            If CMP = 1
                Then /*找到了字符串*/
                    Set CMP To 0 ; /*复原*/
                    Set CHG To 1 ; /*指出改变了一行正文*/
                    Block /*用字符串 REP 替换在缓冲器 IN 中的字符串 MAT*/
                        IN:= Substr ( IN , 1 , I - 1 ) Cat REP Cat Substr ( IN , I + Length
(MAT),Length(IN) - (I + Length(MAT)) + 1)Cat All ' ' ;

```

```

    /*如果在前两次连接后 IN 的长度仍小于 LINESIZE，则用空格
     填在 IN 的尾部，使得替换后 IN 的长度等于 LINESIZE*/
    I = I + Length ( REP );
Endblock ;
Else /*尚未匹配*/
    Block
        I:=I + 1
    Endblock ;
    Endif ;
Endloop ;
If CHG = 1 /*需要替换*/
Then /*替换正文链中的这行正文*/
    Block
        TEXT.S ( BPTR ) := IN ;
    Endblock ;
    Call WRITE_LINE ; /*印出这行正文*/
Endif ;
Block
    CTR:=CTR - 1
Endblock ;
If CTR = 0
Then
    Return ;
Endif ;
If TEXT.N ( BPTR ) = 0
Then /*到正文链最后一行了*/
    Block
        IN:= 'EOF' ;
    Endblock ;
    Call WRITE_LINE ;
    Return ;
Else
    Block
        BPTR:=TEXT.N ( BPTR ); /*指向下一行正文*/
    Endblock ;
    Endif ;
Endloop ;
End REPLACE ;
# 26 Procedure GET_STRING2 ;

```

软件工程

```
/*从编辑命令中获取第二个字符串 STRING2，开始时指针 I 指向第二条斜线后的第一个字符。  
取得的字符串放在缓冲器 REP 中*/  
If Substr ( IN , I , 1 ) = '/'  
Then /*第二个字符串是空串*/  
    Block  
        IN: = '? !';  
    Endblock;  
    Call WRITE_LINE;  
    Return ( 1 ); /*不成功返回*/  
Endif;  
Block  
    K: = 1; /*K 指向 REP*/  
Endblock;  
Loop /*每循环一次执行一次斜线匹配*/  
    If Match ( '/' , Substr ( IN , I , 1 )) = 1  
        Then /*找到了第三条斜线*/  
            Exitloop;  
        Else /*尚未找到斜线*/  
            If I = LINESIZE  
                Then /*遗漏第三条斜线*/  
                    Block  
                        IN: = '? !';  
                    Endblock;  
                    Call WRITE_LINE;  
                    Return ( 1 ); /*不成功返回*/  
                Else /*移到下一个字符*/  
                    Block  
                        I: = I + 1  
                        K: = K + 1; /*最后，K - 1 代表第二个字符串的长度*/  
                    Endblock;  
                Endif;  
            Endif;  
    Endloop;  
Block  
    REP: = Substr ( IN , I - K + 1 , K - 1 ); /*把 STRING2 存在 REP 中*/  
    I: = I + 1 /*指向第三条斜线后面的第一个字符*/  
Endblock;  
Return ( 0 ); /*成功返回*/  
End GET_STRING2;
```

3.13 小结

软件设计的目标是设计出所要开发的软件的模型，传统的软件工程方法学采用结构化设计技术完成软件设计工作。通常把软件设计过程划分为概要设计和详细设计这样两个阶段，两个阶段的任务性质明显不同。

软件设计在软件工程过程中处于技术核心地位，是软件开发过程中决定软件产品质量的关键阶段。

软件设计必须依据对软件产品的需求来进行，因此，结构化设计把结构化分析的结果作为基本输入信息。

为了获得高质量的软件设计结果，应该遵循模块化、模块独立、抽象、逐步求精和信息隐藏等设计准则（也称为设计原理），特别是其中的模块独立原理，对软件体系结构设计和接口设计都具有非常重要和十分具体的指导作用。总结众多软件工程师在开发软件的长期实践中所积累的丰富经验，形成了一些启发规则。这些启发规则虽然不像设计准则那样普遍适用，但是在许多场合都能给软件工程师有益的启示，有助于他们设计出有效的模块化的软件。

通常，使用层次图或结构图表示软件结构，这些图形工具具有形象直观、容易理解的优点，读者应该学会用这类图形描绘软件结构。

面向数据流的设计方法是设计软件体系结构的一种系统化的方法，它定义了一些映射规则，可以把数据流图转换成软件的初步结构。得出软件的初步结构之后，还必须根据设计好的标准、基本设计原理和启发规则为指南，对所得到的软件结构进行仔细优化，才能设计出令人满意的软件体系结构。

人机界面设计是接口设计的一个组成部分。对于交互式系统来说，人机界面设计和数据设计、体系结构设计、过程设计一样重要。人机界面的质量直接影响用户对软件产品的接受程度，因此，必须对人机界面设计给予足够重视。在设计人机界面的过程中，必须充分重视并认真处理好系统响应时间、用户帮助设施、出错信息处理和命令交互等四个设计问题。用户界面设计是一个迭代过程，通常，先创建设计模型，接下来用原型实现这个设计模型并由用户试用和评估原型，然后根据用户意见修改原型，直到用户满意为止。总结人们在设计人机界面过程中积累的经验，得出了一些关于用户界面设计的指南，认真对待这些设计指南有助于设计出友好、高效的人机界面。

过程设计应该在数据设计、体系结构设计和接口设计完成之后进行，它是详细设计阶段的主要任务。过程设计的目标不仅是保证算法正确，更重要的是设计出的处理过程应该尽可能简明易懂。结构程序设计技术是实现上述目标的关键技术，因此是过程设计的逻辑基础。

描述程序处理过程的工具，可分为图形、表格和语言三类，这三类工具各有所长，读者应该能够根据需要选用适当的工具。

在许多应用领域中信息都有清楚的层次结构，在开发这类应用系统时可以采用面向数据结构的设计方法完成过程设计。

本章最后讲述了一个汉字行编辑程序的结构化设计过程，并且给出了完整的设计结果。

在对这个实际软件进行概要设计和详细设计的过程中，我们综合运用了本章前面各节中讲述过的主要技术方法。把这个实际例子和本章前面各节讲述的内容结合起来认真学习，对读者深入理解结构化设计方法并逐步学会在开发实际软件的过程中运用这种方法，是很有帮助的。

习题三

1. 需求分析得出的结果中哪些信息为数据设计奠定了基础？哪些信息为软件体系结构设计奠定了基础？哪些信息为接口设计奠定了基础？哪些信息为过程设计奠定了基础？
2. 为每种类型的模块耦合举一个具体例子。
3. 为每种类型的模块内聚举一个具体例子。
4. 举例说明信息隐藏和模块独立的关系。
5. 举例说明耦合和可移植性的关系。
6. 从你用过的软件中分别举出不好的用户界面和优秀的用户界面的例子，并运用本章讲述的关于人机界面设计的知识去评论它们。
7. 用面向数据流的方法设计下列软件系统的结构：

- (1) 储蓄系统（见习题二第3题）。
- (2) 机票预定系统（见习题二第4题）。
- (3) 患者监护系统（见习题二第5题）。

8. 研究下面的伪码程序：

```
LOOP : Set I to ( START + FINISH ) / 2
      If TABLE ( I ) = ITEM goto FOUND
      If TABLE ( I ) < ITEM Set START to ( I + 1 )
      If TABLE ( I ) > ITEM Set FINISH to ( I - 1 )
      If ( FINISH - START ) > 1 goto LOOP
      If TABLE ( START ) = ITEM goto FOUND
      If TABLE ( FINISH ) = ITEM goto FOUND
      Set FLAG to 0
      Goto DONE
FOUND : Set FLAG to 1
DONE : Exit
```

要求：

- (1) 画出程序流程图；
 - (2) 程序是结构化的吗？说明理由；
 - (3) 若程序是非结构化的，请设计一个等价的经典的结构化程序并且画出程序流程图和盒图；
 - (4) 此程序的功能是什么？它完成预定功能有什么隐含的前提条件吗？
9. 某交易所规定给经纪人的手续费计算方法如下：总手续费等于基本手续费加上与交易中的每股价格和股数有关的附加手续费。如果交易总金额少于1000元，则基本手续费为交易

金额的 8.4%；如果交易总金额在 1 000 元到 10 000 元之间，则基本手续费为交易金额的 5%，再加 34 元；如果交易总金额超过 10 000 元，则基本手续费为交易金额的 4% 加上 134 元。当每股售价低于 14 元时，附加手续费为基本手续费的 5%，除非买进、卖出的股数不是 100 的倍数，在这种情况下附加手续费为基本手续费的 9%。当每股售价在 14 元到 25 元之间时，附加手续费为基本手续费的 2%，除非交易的股数不是 100 的倍数，在这种情况下附加手续费为基本手续费的 6%。当每股售价超过 25 元时，如果交易的股数零散（即，不是 100 的倍数），则附加手续费为基本手续费的 4%，否则附加手续费为基本手续费的 1.5%。

要求：

- (1) 用判定表表示手续费的计算方法；
 - (2) 用判定树表示手续费的计算方法。
10. 假设只有顺序和 DO WHILE 两种控制结构，怎样利用它们完成 IF THEN ELSE 操作？
 11. 把 3.11 节中统计空格程序的 Jackson 图改画为盒图和 PAD 图。
 12. 人机对话由操作员信息和系统信息交替组成，假设一段对话总是由操作员信息开始以系统信息作为结束。请用 Jackson 图描绘上述的人机对话过程。

第4章 结构化实现

所谓软件实现通常指的是编码和测试这两个阶段。

编码就是把软件设计的结果翻译成用某种程序设计语言书写的程序。作为软件工程过程的一个阶段，编码是软件设计的自然结果，因此，程序的质量主要取决于软件设计的质量。但是，所选用的程序设计语言的特点和编程时的风格，也会对程序的可靠性、可读性、可测试性和可维护性产生深远的影响。

正如任何产品在交付使用之前都必须经过严格的检验过程一样，由于软件开发的复杂性和困难性，软件产品在交付使用之前尤其应该经过严格的质量检验过程。通常把对软件的质量检验过程称为测试。目前，软件测试仍然是保证软件质量的主要途径，它是对软件需求规格说明、软件设计和编码的最后复审。

仅就测试而言，它的目标是发现软件中的错误，但是，发现错误并不是我们的最终目的。软件工程的根本目标，是开发出高质量的完全符合用户需要的软件产品，因此，通过测试发现软件错误之后还必须诊断并改正错误，这就是调试（也称为纠错）的任务。调试是测试阶段最困难的工作。

在对测试结果进行收集和评价的时候，软件产品所达到的可靠性也逐渐明朗了。软件可靠性模型使用故障率数据，预测软件的可靠性。

4.1 编 码

4.1.1 选择适当的程序设计语言

程序设计语言是人和计算机通信的基本工具，它的特点不可避免地会影响人思维和解决问题的方式，会影响人和计算机通信的方式和质量，也会影响其他人阅读和理解程序的难易程度。因此，编码之前的一项重要工作就是选择一种适当的程序设计语言。

适当的程序设计语言能使程序员在根据设计编码时遇到的困难最少，可以减少需要的程序测试量，并且可以写出更容易阅读和更容易维护的程序。由于软件系统的绝大部分成本用在生命周期的测试和维护阶段，因此容易测试和容易维护是极端重要的。

使用汇编语言编码需要把软件设计翻译成机器操作的序列，由于这两种表示方法很不相同，因此汇编程序设计既困难又容易出差错。一般说来，高级语言的源程序语句和汇编代码指令之间有一句对多句的对应关系。统计资料表明，程序员在相同时间内可以写出的高级语言语句数和汇编语言指令数大体相同，因此用高级语言写程序比用汇编语言写程序生产率可

以提高好几倍。高级语言一般都允许用户给程序变量和子程序赋予含义鲜明的名字，通过名字很容易把程序对象和它们所代表的实体联系起来；此外，高级语言使用的符号和概念更符合人的习惯。因此，用高级语言写的程序容易阅读，容易测试，容易调试，容易维护。

总的说来。高级语言明显优于汇编语言，因此，除了在很特殊的应用领域（例如，对程度执行时间和使用的空间都有很严格限制的情况；需要产生任意的甚至非法的指令序列；体系结构特殊的微处理机，以致在这类机器上通常不能实现高级语言编译程序），或者大型系统中执行时间非常关键的（或直接依赖于硬件的）一小部分代码需要用汇编语言书写之外，其他程序应该一律用高级语言书写。

为了使程序容易测试和维护以减少生命周期的总成本，选用的高级语言应该有理想的模块化机制，以及可读性好的控制结构和数据结构；为了便于调试和提高软件可靠性，语言特点应该使编译程序能够尽可能多地发现程序中的错误；为了降低软件开发和维护的成本，选用的语言应该有良好的独立编译机制。上述这些要求是选择语言的理想标准，但是在实际选用语言时不能仅仅考虑理论上的标准，还必须同时考虑实用方面的各种限制。重要的实用标准有下述几条：

（1）系统用户的要求

如果所开发的系统由用户负责维护，用户通常要求用他们熟悉的语言书写程序。

（2）可以使用的编译程序

运行目标系统的环境中可以提供的编译程序往往限制了可以选用的语言的范围。

（3）可以得到的软件工具

如果某种语言有支持程序开发的软件工具可以利用，则目标系统的实现和验证都变得比较容易。

（4）工程规模

如果工程规模很庞大，现有的语言又不完全适用，那么设计并实现一种供这个工程项目专用的程序设计语言，可能是一个正确的选择。

（5）程序员的知识

虽然对于有经验的程序员来说，学习一种新语言并不困难，但是要完全掌握一种新语言却需要实践。如果和其他标准不矛盾，那么应该选择一种已经为程序员所熟悉的语言。

（6）软件可移植性要求

如果目标系统将在几台不同的计算机上运行，或者预期的使用寿命很长，那么选择一种标准化程度高、程序可移植性好的语言就是很重要的。

（7）软件的应用领域

所谓的通用程序设计语言实际上并不是对所有应用领域都同样适用，例如，FORTRAN 语言特别适合于工程和科学计算，COBOL 语言适合于商业领域应用，C 语言和 Ada 语言适用于系统和实时应用领域，LISP 语言适用于组合问题领域，PROLOG 语言适于表达知识和推理。因此，选择语言时应该充分考虑目标系统的应用范围。

4.1.2 正确的编码风格

虽然选取了好的程序设计语言有助于写出既可靠又容易阅读、容易维护的程序，但是，工具再好使用不当也不会达到预期的效果。按照软件工程方法学开发软件，程序是表达软件

设计结果的一种更具体的形式，程序的质量基本上由设计的质量决定，但是，编码风格也在很大程度上决定着程序的质量。

所谓编码风格就是程序员在编写程序时遵循的具体准则和习惯做法。源程序代码的逻辑简明清晰、易读易懂是好程序的一个重要标准，为了写出好程序应该遵循下述规则。

1. 程序内部必须有正确的文档

在本书前面的章节中我们曾多次强调指出文档的重要性，事实上，除了存在于程序外部的文档之外，程序内部的文档对于提高程序的可读性和可理解性也是非常有帮助的。

所谓程序内部的文档包括恰当的标识符、适当的注解和程序的视觉组织等等。

选取含义鲜明的名字，使它能正确地提示程序对象所代表的实体，这对于帮助阅读者理解程序是很重要的。如果使用缩写，那么缩写规则应该一致，并且应该给每个名字加注解。

注解是程序员和程序读者通信的重要手段，正确的注解非常有助于对程序的理解。通常在每个模块开始处有一段序言性的注解，简要描述模块的功能、主要算法、接口特点、重要数据以及开发简史。插在程序中间与一段程序代码有关的注解，主要解释包含这段代码的必要性。对于用高级语言书写的源程序，不需要用注解的形式把每个语句翻译成自然语言，应该利用注解提供一些额外的信息。应该用空格或空行清楚地区分注解和程序。注解的内容一定要正确，错误的注解不仅对理解程序毫无帮助，反而会妨碍对程序的理解。

程序清单的布局对于程序的可读性也有很大影响，应该利用适当的阶梯形式使程序的层次结构清晰明显。

2. 数据说明应便于查阅易于理解

虽然在设计期间已经确定了数据结构的组织和复杂程度，然而数据说明的风格却是在写程序时确定的。为了使数据更容易理解和维护，有一些比较简单的原则应该遵循。

数据说明的次序应该标准化（例如，按照数据结构或数据类型确定说明的次序）。有次序就容易查阅，因此能够加速测试、调试和维护的过程。

当多个变量名在一个语句中说明时，应该按字母顺序排列这些变量。

如果设计时使用了一个复杂的数据结构，则应该用注解说明用程序设计语言实现这个数据结构的方法和特点。

3. 语句应该尽量简单清晰

设计期间确定了软件的逻辑结构，然而个别语句的构造却是编写程序的一个主要任务。构造语句时应该遵循的原则是，每个语句都应该简单而直接，不能为了提高效率而使程序变得过分复杂。下述规则有助于使语句简单明了：

- ? 不要为了节省空间而把多个语句写在同一行；
- ? 尽量避免复杂的条件测试；
- ? 尽量减少对“非”条件的测试；
- ? 避免大量使用循环嵌套和条件嵌套；
- ? 利用括号使逻辑表达式或算术表达式的运算次序清晰直观。

4. 正确的输入/输出风格

在设计和编写程序时应该考虑下述有关输入/输出风格的规则：

- ? 对所有输入数据都进行检验；
- ? 检查输入项重要组合的合法性；

- ? 保持输入格式简单；
- ? 使用数据结束标记，不要求用户指定数据的数目；
- ? 明确提示交互式输入的请求，详细说明可用的选择或边界数值；
- ? 当程序设计语言对格式有严格要求时，应保持输入格式一致；
- ? 设计良好的输出报表；
- ? 给所有输出数据加标志。

5. 不要盲目追求高效率

效率主要指处理机时间和存储器容量两个方面。虽然值得提出提高效率的要求，但是在进一步讨论这个问题之前应该记住三条原则：首先，效率是性能要求，因此应该在需求分析阶段确定效率方面的要求。软件应该像对它要求的那样有效，而不应该如同人类可能做到的那样有效。其次，效率是靠好设计来提高的。第三，程序的效率和程序的简单程度是一致的。不要牺牲程序的清晰性和可读性来不必要地提高效率。下面从三个方面进一步讨论效率问题：

(1) 程序运行时间

源程序的效率直接由详细设计阶段确定的算法的效率决定，但是，写程序的风格也能对程序的执行速度和存储器要求产生影响。在把详细设计结果翻译成程序时，总可以应用下述规则：

- ? 写程序之前先简化算术的和逻辑的表达式；
- ? 仔细研究嵌套的循环，以确定是否有语句可以从内层往外移；
- ? 尽量避免使用多维数组；
- ? 尽量避免使用指针和复杂的表；
- ? 使用执行时间短的算术运算；
- ? 不要混合使用不同的数据类型；
- ? 尽量使用整数运算和布尔表达式。

在效率是决定性因素的应用领域，尽量使用有良好优化特性的编译程序，以自动生成高效目标代码。

(2) 存储器效率

在大型计算机中必须考虑操作系统页式调度的特点，一般说来，使用能保持功能域的结构化控制结构，是提高效率的好方法。

在微处理机中如果要求使用最少的存储单元，则应选用有紧缩存储器特性的编译程序，在非常必要时可以使用汇编语言。

提高执行效率的技术通常也能提高存储器效率。提高存储器效率的关键同样是“简单”。

(3) 输入/输出的效率

如果用户为了给计算机提供输入信息或为了理解计算机输出的信息，所需花费的脑力劳动是经济的，那么人和计算机之间通信的效率就高。因此，简单清晰同样是提高人-机通信效率的关键。

硬件之间的通信效率是很复杂的问题，但是，从写程序的角度看，却有些简单的原则可以提高输入/输出的效率。例如：

- ? 所有输入/输出都应该有缓冲，以减少用于通信的额外开销；
- ? 对二级存储器（如磁盘）应选用最简单的访问方法；
- ? 二级存储器的输入/输出应该以信息组为单位进行；

? 如果“超高效的”输入/输出很难被人理解，则不应采用这种方法。
这些简单原则对于软件工程的设计和编码两个阶段都适用。

4.2 软件测试概述

4.2.1 软件必须测试

任何产品在交付使用之前都必须经过严格的质量检验过程，软件产品也不例外，而且对于软件产品来说，测试的必要性尤其突出。

无论怎样强调软件测试的重要性和它对软件可靠性的影响都不过分。在开发大型软件系统的漫长过程中，面对着极其错综复杂的问题，人的主观认识不可能完全符合客观现实，与工程密切相关的各类人员之间的通信和配合也不可能完美无缺，因此，在软件生命周期的每个阶段都不可避免地会产生差错。我们力求在每个阶段结束之前通过严格的技术审查，尽可能早地发现并纠正差错；但是，经验表明审查并不能发现所有差错，此外在编码过程中还不可避免地会引入新的错误。如果在软件投入生产性运行之前，没有发现并纠正软件中的大部分差错，则这些差错迟早会在生产过程中暴露出来，那时不仅改正这些错误的代价更高，而且往往会造成很恶劣的后果。测试的目的就是在软件投入生产性运行之前，尽可能多地发现软件中的错误。目前软件测试仍然是保证软件质量的关键步骤，它是对软件规格说明、设计和编码的最后复审。

软件测试在软件生命周期中横跨两个阶段。通常在编写出每个模块之后就对它做必要的测试（称为单元测试），模块的编写者和测试者是同一个人，编码和单元测试属于软件生命周期的同一个阶段。在这个阶段结束之后，对软件系统还应该进行各种综合测试，这是软件生命周期中的另一个独立的阶段，通常由专门的测试人员承担这项工作。

大量统计资料表明，软件测试的工作量往往占软件开发总工作量的 40% 以上，在极端情况，测试那种关系人的生命安全的软件所花费的成本，可能相当于软件工程其他步骤总成本的 3~5 倍。因此，必须高度重视软件测试工作，绝不要以为写出程序之后软件开发工作就接近完成了，实际上，大约还有同样多的开发工作量需要完成。

4.2.2 软件测试的目标

什么是测试？它的目标是什么？G.Myers 给出了关于测试的一些规则，这些规则也可以看作是测试的目标或定义：

- (1) 测试是为了发现程序中的错误而执行程序的过程；
- (2) 好的测试方案是极可能发现迄今为止尚未发现的错误的测试方案；
- (3) 成功的测试是发现了至今为止尚未发现的错误的测试。

从上述规则可以看出，测试的正确定义是“为了发现程序中的错误而执行程序的过程”。这和某些人通常想象的“测试是为了表明程序是正确的”，“成功的测试是没有发现错误的测试”等等是完全相反的。正确认识测试的目标是十分重要的，测试目标决定了测试方案的设计。如果为了表明程序是正确的而进行测试，就会设计一些不易暴露错误

的测试方案；相反，如果测试是为了发现程序中的错误，就会力求设计出最能暴露错误的测试方案。

表面看来，软件测试的目的与软件工程所有其他阶段的目的都相反。软件工程的其他阶段都是“建设性”的：软件工程师力图从抽象的概念出发，逐步设计出具体的软件系统，直到用一种适当的程序设计语言写出可以执行的程序代码。但是，在测试阶段测试人员努力设计出一系列测试方案，目的却是为了“破坏”已经建造好的软件系统——竭力证明程序中有错误不能按照预定要求正确工作。

当然，这种反常仅仅是表面的，或者说是心理上的。暴露问题并不是软件测试的最终目的，发现问题是为了解决问题，测试阶段的根本目标是尽可能多地发现并排除软件中潜藏的错误，最终把一个高质量的软件系统交给用户使用。但是，仅就测试本身而言，它的目标可能和许多人原来设想的很不相同。

由于测试的目标是暴露程序中的错误，从心理学角度看，由程序的编写者自己进行测试是不恰当的。因此，在综合测试阶段通常由其他人员组成测试小组来完成测试工作。

此外，应该认识到测试决不能证明程序是正确的。即使经过了最严格的测试之后，仍然可能还有没被发现的错误潜藏在程序中。测试只能查找出程序中的错误，不能证明程序中没有错误。关于这个结论下面还要讨论。

4.2.3 两类测试方法

怎样对程序进行测试呢？测试任何产品都有两种方法：如果已经知道了产品应该具有的功能，可以通过测试来检验是否每个功能都能正常使用；如果知道产品内部预定的工作过程，可以通过测试来检验产品内部动作是否按照规格说明书的规定正常进行。前一个方法称为黑盒测试，后一个方法称为白盒测试。

对于软件测试而言，黑盒测试法把程序看成一个黑盒子，完全不考虑程序的内部结构和处理过程。也就是说，黑盒测试是在程序接口进行的测试，它只检查程序功能是否能按照规格说明书的规定正常使用，程序是否能适当地接收输入数据产生正确的输出信息，并且保持外部信息（如，数据库或文件）的完整性。黑盒测试又称为功能测试。与黑盒测试法相反，白盒测试法的前提是可以把程序看成装在一个透明的白盒子里，也就是完全了解程序的结构和处理过程。这种方法按照程序内部的逻辑测试程序，检验程序中的每条通路是否都能按预定要求正确工作。白盒测试又称为结构测试。

粗看起来，不论采用上述哪种测试方法，只要对每一种可能的情况都进行测试，就可以得到完全正确的程序。包含所有可能情况的测试称为穷尽测试，对于实际程序而言，穷尽测试通常是不可能做到的。使用黑盒测试法，为了做到穷尽测试，至少必须对所有输入数据的各种可能值的排列组合都进行测试，但是，由此得到的应测试的情况数往往大到实际上根本无法测试的程度。例如，一个程序需要三个整数型的输入数据，如果计算机字长16位，则每个整数可能取的值有 2^{16} 个，三个输入数据的各种可能值的排列组合共有

$$2^{16} \times 2^{16} \times 2^{16} = 2^{48} \quad 3 \times 10^{14} \text{ 种}$$

也就是说，大约需要把这个程序执行 3×10^{14} 次才能做到“穷尽”测试。假定每执行一次程序需要一毫秒，执行这么多次大约需要一万年！不仅测试时间长得叫人不可思议，测试得出的输出数据更是多得叫人完全无法分析。然而严格地说这还不能算穷尽测试，为了保证测试能发

现程序中的所有错误，不仅应该使用有效的输入数据（对这个例子来说是合法的整数），还必须使用一切可能的输入数据（例如，不合法的整数、实数、字符串等等）。实践表明，用无效的输入数据比用有效的输入数据进行测试，往往能发现更多的错误。

使用白盒测试法，为了做到穷尽测试，程序中每条可能的通路至少都应该执行一次（严格地说每条通路都应在每种可能的输入数据下执行一次）。即使测试很小的程序，通常也不能做到上述这一点。例如，一段程序对嵌套的 IF 语句循环执行 20 次（图 4.1 是它的程序流程图），在这段程序中共有 5^{20} (10^{14}) 条可能的执行通路，显然，即使每条通路只执行一次也是不可能的。

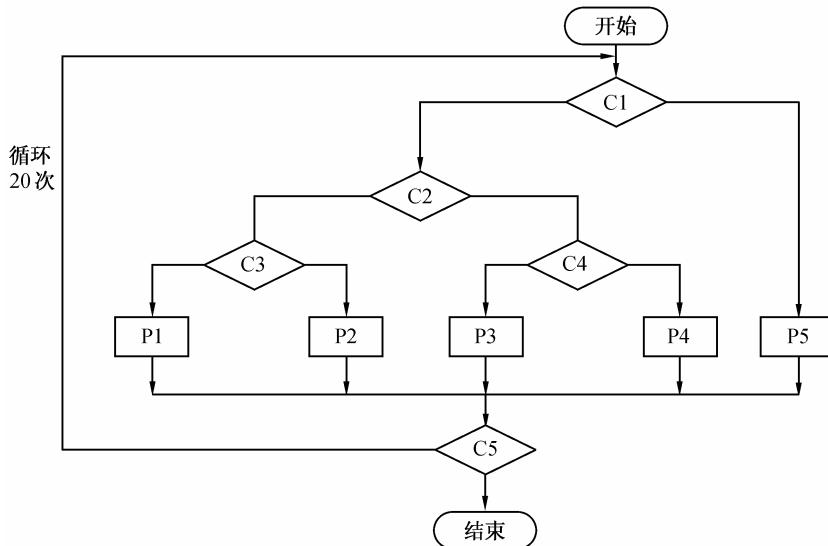


图 4.1 一个小程序的程序流程图

因为不可能进行穷尽测试，所以软件测试不可能发现程序中的所有错误，也就是说，通过测试并不能证明程序是正确的。但是，我们的目的是要通过测试保证软件的可靠性，因此，必须仔细设计测试方案，力争用尽可能少的测试发现尽可能多的错误。

4.2.4 软件测试准则

为了能够设计出有效的测试方案，软件工程师必须充分理解并正确运用指导软件测试工作的基本准则。下面叙述主要的测试准则：

? 所有测试都应该能够追溯到用户需求。正如前面讲过的，软件测试的目标是发现程序中的错误。从用户角度看，最严重的错误是程序不能满足用户需求的那些错误，因此应该围绕用户的需求来测试程序。

? 应该在开始测试之前预先制定出测试计划。一旦完成了需求分析就可以着手制定测试计划，在确定了设计模型之后就可以立即开始设计详细的测试方案。因此，在编码之前就可以对所有测试工作进行计划和设计。

? 在软件测试过程中应该应用 Pareto 原理。Pareto 原理告诉我们，测试所发现的错误中的 80% 很可能是由程序中 20% 的模块造成的。因此，应该尽量找出这些可疑的模块并彻底地测试

它们。

? 应该从“小规模”测试开始，逐步过渡到“大规模”测试。通常，首先测试单个程序模块，进一步的测试重点随后转向在集成的模块簇中寻找错误，最后对整个软件系统进行测试。

? 穷举测试是不可能的。在上一小节已经举例说明了这个事实，此处不再赘述。这个事实表明，测试只能证明程序中有错误，不能证明程序中没有错误。但是，精心设计测试方案，有可能充分覆盖程序逻辑并发现大部分程序错误。

? 为了达到最佳的测试效果，应该由独立的第三方来从事测试工作。所谓“最佳测试效果”是指用尽可能少的测试方案发现了尽可能多的程序错误。由于在4.2.2小节中已经讲过的原因，创建软件系统的软件工程师并不是完成全部测试工作的最佳人选（他们通常主要承担模块测试工作）。

4.3 白盒测试技术

白盒测试方法是按照程序内部预期应有的逻辑测试程序，检验程序中的每条执行通路是否都能按预定要求正确工作。

设计测试方案是测试阶段的关键技术问题。所谓测试方案包括下述三方面内容：具体的测试目的（例如，要测试的具体功能），应该输入的测试数据和预期的输出结果。通常又把测试数据和预期的输出结果称为测试用例。

不同的测试数据发现程序错误的能力差别很大，为了提高测试效率降低测试成本，应该选用高效的测试数据。因为不可能进行穷尽的测试，选用少量“最有效的”测试数据，做到尽可能完备的测试就更重要了。

4.3.1 逻辑覆盖

逻辑覆盖是设计白盒测试方案的一种常用技术。通常不可能进行穷尽的测试，因此，有选择地执行程序中某些最有代表性的通路，是用白盒方法测试程序时对穷尽测试惟一可行的替代办法。所谓逻辑覆盖，是对一系列测试过程的总称，这组测试过程逐渐进行越来越完整的通路测试。测试数据执行（或称为覆盖）程序逻辑的程度可以划分成哪些不同的等级呢？从覆盖源程序语句的详尽程度分析，大致有以下一些不同的覆盖标准。

1. 语句覆盖

为了暴露程序中的错误，至少每个语句应该执行一次。语句覆盖的含义是，选择足够多的测试数据，使被测程序中每个语句至少执行一次。

例如，图4.2是一个被测模块的流程图，它的源程序（用PASCAL语言书写）应该如下：

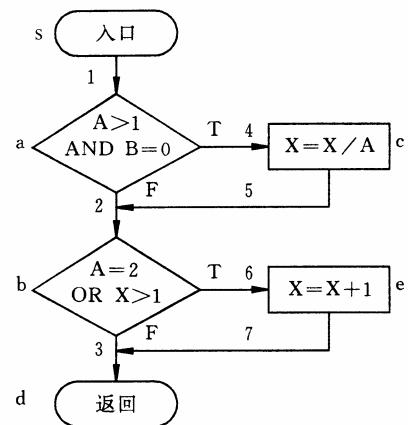


图4.2 被测试模块的流程图

```

PROCEDURE EXAMPLE ( A , B : REAL ; VAR X : REAL );
BEGIN
  IF ( A > 1 ) AND ( B = 0 )
    THEN X : = X / A ;
  IF ( A = 2 ) OR ( X > 1 )
    THEN X : = X + 1
END ;

```

为了使每个语句都执行一次，程序的执行路径应该是 `sacbed`，为此只需要输入下面的测试数据（实际上 X 可以是任意实数）：

$$A = 2, B = 0, X = 4$$

语句覆盖对程序的逻辑覆盖很少，在上面例子中两个判定条件都只测试了条件为真的情况，如果条件为假时处理有错误，显然不能发现。此外，语句覆盖只关心判定表达式的值，而没有分别测试判定表达式中每个条件取不同值时的情况。在上面的例子中，为了执行 `sacbed` 路径，以测试每个语句，只需两个判定表达式 $(A > 1) \text{ AND } (B = 0)$ 和 $(A = 2) \text{ OR } (X > 1)$ 都取真值，因此使用上述一组测试数据就够了。但是，如果程序中把第一个判定表达式中的逻辑运算符“AND”错写成“OR”，或把第二个判定表达式中的条件“ $X > 1$ ”误写成“ $X < 1$ ”，使用上面的测试数据并不能查出这些错误。

综上所述，可以看出语句覆盖是很弱的逻辑覆盖标准，为了更充分地测试程序，可以采用下述的逻辑覆盖标准。

2. 判定覆盖

判定覆盖又叫分支覆盖，它的含义是，不仅每个语句必须至少执行一次，而且每个判定的每种可能的结果都应该至少执行一次，也就是每个判定的每个分支都至少执行一次。

对于上述例子来说，能够分别覆盖路径 `sacbd` 和 `sabd` 的两组测试数据，或者可以分别覆盖路径 `sacbd` 和 `sabed` 的两组测试数据，都满足判定覆盖标准。例如，用下面两组测试数据就可做到判定覆盖：

- . $A = 3, B = 0, X = 3$ (覆盖 `sacbd`)
- . $A = 2, B = 1, X = 1$ (覆盖 `sabed`)

判定覆盖比语句覆盖强，但是对程序逻辑的覆盖程度仍然不高，例如，上面的测试数据只覆盖了程序全部路径的一半。

3. 条件覆盖

条件覆盖的含义是，不仅每个语句至少执行一次，而且使判定表达式中的每个条件都取到各种可能的结果。

图 4.2 的例子中共有两个判定表达式，每个表达式中有两个条件，为了做到条件覆盖，应该选取测试数据使得在 a 点有下述各种结果出现：

$$A > 1, A = 1, B = 0, B = 0$$

在 b 点有下述各种结果出现：

$$A = 2, A = 2, X > 1, X = 1$$

只需要使用下面两组测试数据就可以达到上述覆盖标准：

- . $A = 2, B = 0, X = 4$

(满足 $A > 1, B = 0, A = 2$ 和 $X > 1$ 的条件，执行路径 sacbed)

. $A = 1, B = 1, X = 1$

(满足 $A = 1, B = 0, A = 2$ 和 $X = 1$ 的条件，执行路径 sabd)

条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却只关心整个判定表达式的值。例如，上面两组测试数据也同时满足判定覆盖标准。但是，也可能有相反的情况：虽然每个条件都取到了两个不同的结果，判定表达式却始终只取一个值。例如，如果使用下面两组测试数据，则只满足条件覆盖标准并不满足判定覆盖标准（第二个判定表达式的值总为真）：

. $A = 2, B = 0, X = 1$

(满足 $A > 1, B = 0, A = 2$ 和 $X = 1$ 的条件，执行路径 sacbed)

. $A = 1, B = 1, X = 2$

(满足 $A = 1, B = 0, A = 2$ 和 $X > 1$ 的条件，执行路径 sabed)

4. 判定/条件覆盖

既然判定覆盖不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖，自然会提出一种能同时满足这两种覆盖标准的逻辑覆盖，这就是判定/条件覆盖。它的含义是，选取足够多的测试数据，使得判定表达式中的每个条件都取到各种可能的值，而且每个判定表达式也都取到各种可能的结果。

对于图 4.2 的例子而言，下述两组测试数据满足判定/条件覆盖标准：

. $A = 2, B = 0, X = 4$

. $A = 1, B = 1, X = 1$

但是，这两组测试数据也就是为了满足条件覆盖标准最初选取的两组数据，因此，有时判定/条件覆盖也并不比条件覆盖更强。

5. 条件组合覆盖

条件组合覆盖是更强的逻辑覆盖标准，它要求选取足够的测试数据，使得每个判定表达式中条件的各种可能组合都至少出现一次。

对于图 4.2 的例子，共有八种可能的条件组合，它们是：

(1) $A > 1, B = 0$

(2) $A > 1, B = 0$

(3) $A = 1, B = 0$

(4) $A = 1, B = 0$

(5) $A = 2, X > 1$

(6) $A = 2, X = 1$

(7) $A = 2, X > 1$

(8) $A = 2, X = 1$

和其他逻辑覆盖标准中的测试数据一样，条件组合 (5) ~ (8) 中的 X 值是指在程序流程图第二个判定框 (b 点) 的 X 值。

下面的四组测试数据可以使上面列出的八种条件组合每种至少出现一次：

. $A = 2, B = 0, X = 4$

(针对 1, 5 两种组合，执行路径 sacbed)

```

.A = 2 , B = 1 , X = 1
( 针对 2 , 6 两种组合 , 执行路径 sabed )
.A = 1 , B = 0 , X = 2
( 针对 3 , 7 两种组合 , 执行路径 sabed )
.A = 1 , B = 1 , X = 1
( 针对 4 , 8 两种组合 , 执行路径 Sabd )

```

显然，满足条件组合覆盖标准的测试数据，也一定满足判定覆盖、条件覆盖和判定/条件覆盖标准。因此，条件组合覆盖是前述几种覆盖标准中最强的。但是，满足条件组合覆盖标准的测试数据并不一定能使程序中的每条路径都执行到，例如，上述四组测试数据都没有测试到路径 sacbd。

以上按照测试数据对源程序语句检测的详尽程度，划分出几种逻辑覆盖标准。在上面的分析过程中，我们常常谈到测试数据所执行的程序路径，显然，测试数据可以检测的程序路径的多少，也表明了对程序逻辑测试的详尽程度。从对程序路径的覆盖程度分析，可以划分出点覆盖、边覆盖和路径覆盖等三种逻辑覆盖标准，感兴趣的读者请参阅《软件工程导论（第三版）》^[1]。

4.3.2 控制结构测试

另外一类重要的白盒测试技术，是根据程序的控制结构设计测试用例的技术，本节讲述其中一些常用的技术。

1. 流图

按照程序的控制结构设计测试用例时，往往需要仔细分析程序的控制流。为了突出表现程序的控制流，可以使用流图（也称为程序图）。流图仅仅描绘程序的控制流程，它完全不表现对数据的具体操作以及分支或循环的具体条件。

在流图中用圆表示节点，一个圆代表一条或多条语句。程序流程图中一个顺序执行的处理框序列和一个菱形判定框，可以映射成流图中的一个节点。流图中的箭头线称为边，它和程序流程图中的箭头线类似，代表控制流。在流图中一条边必须终止于一个节点，即使这个节点并不代表任何语句（实际上相当于一个空语句）。由边和节点围成的面积称为区域，当计算区域数时应该包括图外部未被围起来的那个区域。

图 4.3 举例说明把程序流程图映射成流图的方法。

事实上，用任何方式表示的过程设计结果，都可以翻译成流图。下面讲述基本路径测试方法时，将把用 PDL 描述的一个处理过程转换成流图。

当过程设计结果中包含复合条件时，翻译成流图的方法稍微复杂一些。所谓复合条件，就是在判定条件中包含了一个或多个布尔运算符（逻辑 OR，AND，NAND，NOR）。在这种情况下，应该把复合条件分解为若干个简单条件，每个简单条件对应流图中的一个节点。通常把包含条件的节点称为判定节点，从每个判定节点引出两条或多条边。图 4.4 是由包含复合条件的 PDL 片断翻译成的流图，注意，与复合条件“a OR b”对应的节点有两个，这两个节点分别标记为“a”和“b”。

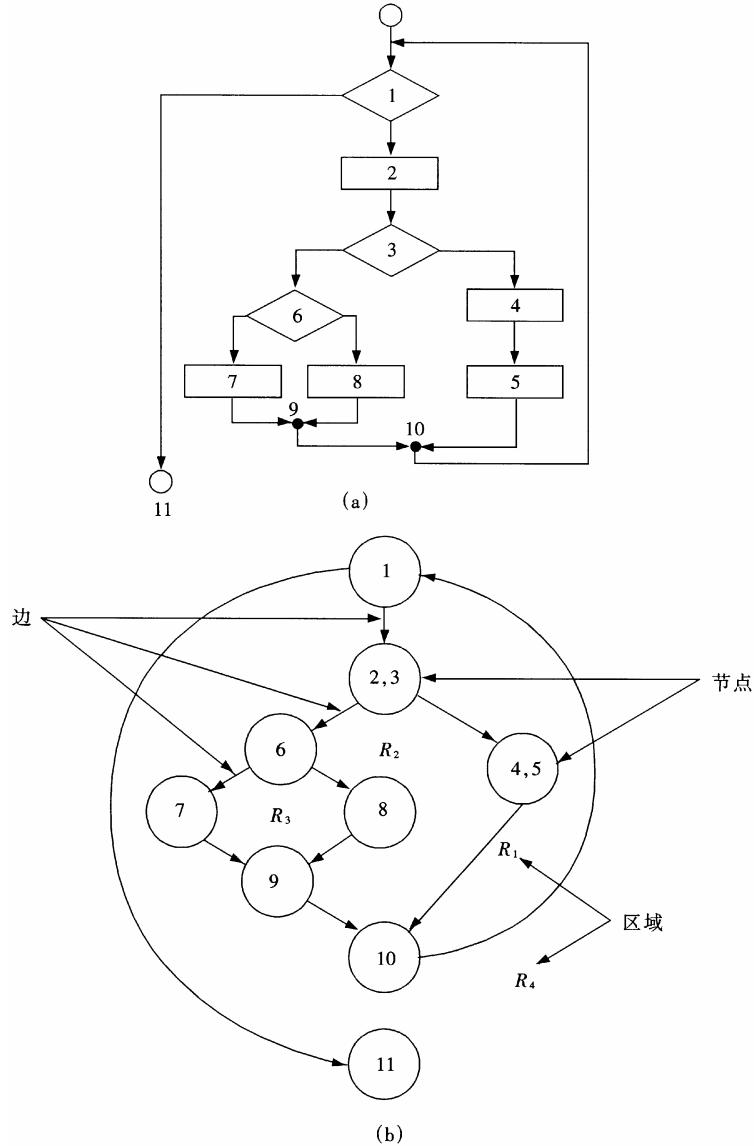


图 4.3 把程序流程图映射成流图

(a) 程序流程图 ;(b) 流图

2. 基本路径测试

基本路径测试是一种常用的白盒测试技术。使用这种技术设计测试用例时，首先计算被测过程的逻辑复杂度，并依据算出的复杂度定义执行路径的基本集合，从该基本集合导出的测试用例可以保证程序中的每条语句至少被执行一次，而且每个判定条件在执行时都分别取 true（真）和 false（假）两个值。

使用基本路径测试技术设计测试用例的步骤如下：

第1步 依据过程设计的结果画出相应的流图。

例如，为了测试下列的用 PDL 描述的求平均值的过程，首先画出图 4.5 所示的流图。注意，为了正确地画出流图，我们把被映射为流图节点的 PDL 语句编了号。

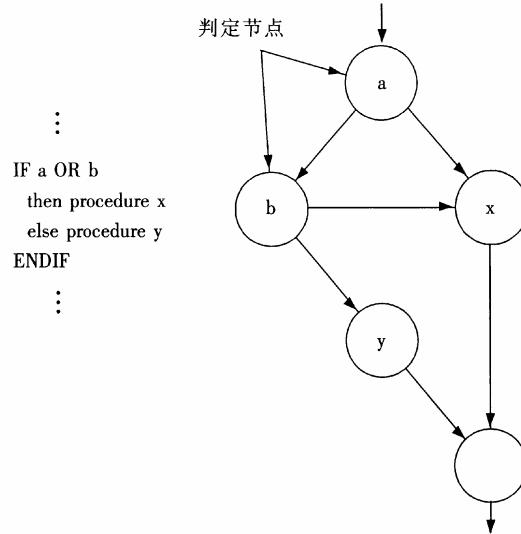


图 4.4 由包含复合条件的 PDL 映射成的流图

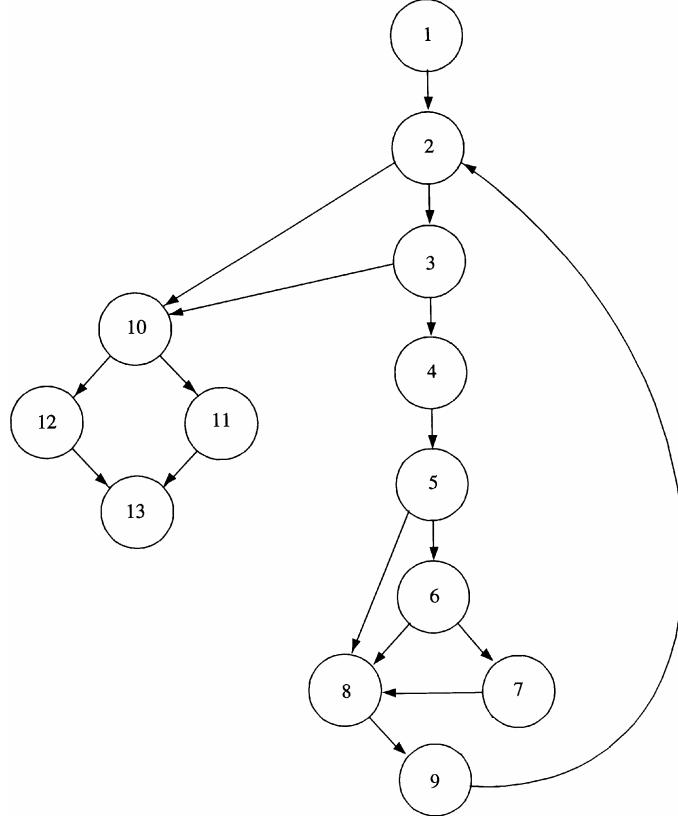


图 4.5 求平均值过程的流图

```

PROCEDURE average ;
  / *这个过程计算不超过 100 个在规定值域内的有效数字的平均值；同时计算有效
    数字的总和及个数。*/
  INTERFACE RETURNS average , total. Input , total.valid ;
  INTERFACE ACCEPTS value , minimum , maximum ;
  TYPE value [ 1..100 ] IS SCALAR ARRAY ;
  TYPE average , total.input , total.valid ;
    minimum , maximum , sum IS SCALAR ;
  TYPE i IS INTEGER ;
 1 :   i = 1 ;
  total.input = total.valid = 0 ;
  sum = 0 ;
 2 :   DO WHILE value [ i ] < > - 999
 3 :     AND total.input < 100
 4 :     increment total.input by 1 ;
 5 :     IF value [ i ] > = minimum
 6 :       AND value [ i ] < = maximum
 7 :     THEN increment total.valid by 1 ;
      sum = sum+value [ i ];
 8 :   ENDIF
  increment i by 1 ;
 9 : ENDDO
10 : IF total.valid > 0
11 : THEN average = sum / total.valid ;
12 : ELSE average = - 999 ;
13 : ENDIF
END average

```

第2步 计算流图的环形复杂度。

环形复杂度定量度量程序逻辑的复杂程度。画出描绘程序控制流的流图之后，可以使用下述三种方法中的任一种来计算环形复杂度。

(1) 流图中的区域数等于环形复杂度。

(2) 流图 G 的环形复杂度 $V(G)$ 由下式计算：

$$V(G) = E - N + 2$$

其中， E 是流图中边的条数， N 是流图中节点数。

(3) 流图 G 的环形复杂度 $V(G)$ 也可由下式计算：

$$V(G) = P + 1$$

其中， P 是流图中判定节点的数目。

例如，使用上述任何一种方法，都可以计算出图 4.5 所示流图的环形复杂度为 6。

第3步 确定线性独立路径的基本集合。

所谓线性独立路径是指这样的路径，该路径至少引入了程序的一个新处理语句集合或一个新条件，用流图术语描述，独立路径中至少包含一条在定义该路径之前不曾用过的边。

使用基本路径测试法设计测试用例时，程序的环形复杂度决定了程序中独立路径的数量，而且这个数是确保程序中所有语句至少被执行一次所需要的测试数量的上限。

对于图 4.5 所描述的求平均值过程来说，由于环形复杂度为 6，因此共有 6 条独立路径。例如，下面列出了 6 条独立路径：

路径 1：1 - 2 - 10 - 11 - 13

路径 2：1 - 2 - 10 - 12 - 13

路径 3：1 - 2 - 3 - 10 - 11 - 13

路径 4：1 - 2 - 3 - 4 - 5 - 8 - 9 - 2 - ...

路径 5：1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 2 - ...

路径 6：1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 2 - ...

路径 4、5、6 后面的省略号 (...) 表示，可以后接通过控制结构其余部分的任意路径（例如，10 - 11 - 13）。

通常，在导出测试用例的过程中，应该首先识别出程序中的判定节点。例如，图 4.5 所示流图中节点 2、3、5、6 和 10 是判定节点。

第 4 步 设计出可强制执行基本集合中每条路径的测试用例。

应该选取测试数据，使得在测试每条路径时都适当地设置好了各个判定节点的条件。可以测试上一步针对求平均值过程确定的基本集合的测试用例如下：

路径 1 的测试用例：

value [k] = 有效输入值，其中 $k < i$ (i 的定义在下面)

value [i] = -999，其中 $2 \leq i \leq 100$

预期结果：基于 k 的正确平均值和总数

注意，路径 1 无法独立测试，必须作为路径 4、5 或 6 的一部分来测试。

路径 2 的测试用例：

value [1] = -999

预期结果：average = -999，其他都保持初始值

路径 3 的测试用例：

试图处理 101 个或更多个值

前 100 个数值应该是有效输入值

预期结果：与测试用例 1 相同

注意，路径 3 也无法独立测试，必须作为路径 4、5 或 6 的一部分来测试。

路径 4 的测试用例：

value [i] = 有效输入值，其中 $i < 100$

value [k] < minimum，其中 $k < i$

预期结果：基于 k 的正确平均值和总数

路径 5 的测试用例：

value [i] = 有效输入值，其中 $i < 100$

value [k] > maximum，其中 $k < i$

预期结果：其于 k 的正确平均值和总数

路径 6 的测试用例：

value [i] = 有效输入值，其中 $i < 100$

预期结果：正确的平均值和总数

在测试过程中，执行每个测试用例并把程序实际输出的结果与预期结果相比较。一旦执行完全部测试用例，就可以确保程序中所有语句都至少被执行了一次，而且每个判定条件都分别取过 true 值和 false 值。

应该注意，某些独立路径（例如，本例中的路径 1 和路径 3）不能以独立的方式测试，也就是说，程序的正常流程不能形成独立执行该路径所需要的数据组合（例如，为了执行本例中的路径 1，需要满足条件 $total.valid > 0$ ，然而独立执行路径 1 无法满足这个条件）。在这种情况下，这些路径必须作为另一个路径的一部分来测试。

3. 条件测试

尽管基本路径测试技术简单而且高效，但是仅有这种技术仍然不够，还需要同时使用其他控制结构测试技术，才能进一步提高白盒测试的质量。

用条件测试技术设计出的测试用例，能够检查程序模块中包含的逻辑条件。一个简单条件是一个布尔变量或一个关系表达式，在布尔变量或关系表达式之前还可能有一个 NOT（“`~`”）算符。关系表达式的形式如下：

$E_1 < \text{关系算符} > E_2$

其中， E_1 和 E_2 是算术表达式，而 $< \text{关系算符} >$ 是下列算符之一：“`<`”，“`>`”，“`=`”，“`≠`”，“`<=`”或“`>=`”。复合条件由两个或多个简单条件、布尔算符和括弧组成。布尔算符有 OR（“`|`”），AND（“`&`”）和 NOT（“`~`”）。不包含关系表达式的条件称为布尔表达式。

因此，一个条件中可能包含布尔变量、布尔算符、布尔括弧（括住简单条件或复合条件）、关系算符及算术表达式等成分。

如果条件不正确，则至少组成该条件的一个成分不正确。因此，条件错误的类型如下：

- ? 布尔变量错。
- ? 布尔算符错（布尔算符不正确，遗漏布尔算符或有多余的布尔算符）。
- ? 布尔括弧错。
- ? 关系算符错。
- ? 算术表达式错。

条件测试方法着重测试程序中的每个条件，这种测试方法有下述两个优点：

- (1) 容易度量条件的测试覆盖率；
- (2) 程序中条件的测试覆盖率可以指导附加测试的设计。

条件测试的目的，不仅仅是检测程序条件中的错误，而且也检测程序中其他类型的错误。如果程序 P 的测试集能有效地检测 P 中条件的错误，则它很可能也可以有效地检测 P 中的其他错误。此外，如果一个测试策略对检测条件错误是有效的，则该策略很可能对检测程序的其他错误也是有效的。

软件工程师们已经提出了许多条件测试策略。分支测试可能是最简单的条件测试策略。对于复合条件 C 来说，C 的真分支和假分支以及 C 中的每个简单条件，都应该至少执行一次。

域测试要求对一个关系表达式执行三个或四个测试。对于形式为

$$E_1 < \text{关系算符} > E_2$$

的关系表达式来说，需要三个测试分别使 E_1 的值大于、等于或小于 E_2 的值。如果 $<$ 关系算符 $>$ 错误而 E_1 和 E_2 正确，则这三个测试能够发现关系算符的错误。为了发现 E_1 和 E_2 中的错误，让 E_1 值大于（或小于） E_2 值的测试数据应该使这两个值之间的差别尽可能小。

在上述种种条件测试技术的基础上，K.C.Tai 提出了一种被称为 BRO (Branch and Relational Operator) 测试的条件测试策略。如果在条件中所有布尔变量和关系算符都只出现一次而且没有公共变量，则 BRO 测试保证能发现该条件中的分支错和关系算符错。

BRO 测试利用条件 C 的条件约束来设计测试用例。包含 n 个简单条件的条件 C 的条件约束定义为 (D_1, D_2, \dots, D_n) 其中 $D_i (0 < i < n)$ 表示条件 C 中第 i 个简单条件的输出约束。如果在条件 C 的一次执行过程中， C 中每个简单条件的输出都满足 D 中对应的约束，则称 C 的这次执行覆盖了 C 的条件约束 D 。

对于布尔变量 B 来说， B 的输出约束指出， B 必须是真 (t) 或假 (f)。类似地，对于关系表达式来说，用符号 $>$ ， $=$ 和 $<$ 指定表达式的输出约束。

作为一个例子，考虑下列条件

$$C_1: B_1 \& B_2$$

其中， B_1 和 B_2 是布尔变量。 C_1 的条件约束形式为 (D_1, D_2) ，其中 D_1 和 D_2 中的每一个都是“ t ”或“ f ”。值 (t, f) 是 C_1 的一个条件约束，并由使 B_1 值为真 B_2 值为假的测试所覆盖。BRO 测试策略要求，约束集 $\{(t, t)(f, t)(t, f)\}$ 被 C_1 的执行所覆盖。如果 C_1 因布尔算符错误而不正确，则至少上述约束集中的一个约束将迫使 C_1 失败。

作为第二个例子，考虑条件

$$C_2: B_1 \& (E_3 = E_4)$$

其中， B_1 是布尔变量， E_3 和 E_4 是算术表达式。 C_2 的条件约束形式为 (D_1, D_2) ，其中 D_1 是“ t ”或“ f ”， D_2 是 $>$ ， $=$ 或 $<$ 。除了 C_2 的第二个简单条件是关系表达式之外， C_2 和 C_1 相同，因此，可以通过修改 C_1 的约束集 $\{(t, t)(f, t)(t, f)\}$ 得出 C_2 的约束集。注意，对于 $(E_3 = E_4)$ 来说，“ t ”意味“ $=$ ”，而“ f ”意味着“ $<$ ”或“ $>$ ”，因此，分别用 $(t, =)$ 和 $(f, =)$ 替换 (t, t) 和 (f, t) ，并用 $(t, <)$ 和 $(t, >)$ 替换 (t, f) ，就得到 C_2 的约束集 $\{(t, =), (f, =), (t, <), (t, >)\}$ 。覆盖上述条件约束集的测试，保证可以发现 C_2 中布尔算符和关系算符的错误。

作为第三个例子，考虑条件

$$C_3: (E_1 > E_2) \& (E_3 = E_4)$$

其中， E_1 、 E_2 、 E_3 和 E_4 是算术表达式。 C_3 的条件约束形式为 (D_1, D_2) ，而 D_1 和 D_2 的每一个都是 $>$ ， $=$ 或 $<$ 。除了 C_3 的第一个简单条件是关系表达式之外， C_3 和 C_2 相同，因此，可以通过修改 C_2 的约束集得到 C_3 的约束集，结果为：

$$\{((>, =), (=, =), (<, =)), (>, <), (>, >)\}$$

覆盖上述条件约束集的测试，保证可以发现 C_3 中关系算符的错误。

4. 循环测试

循环是绝大多数软件算法的基础，但是，在测试软件时却往往未对循环结构进行足够的测试。

循环测试是一种白盒测试技术，它专注于测试循环结构的有效性。在结构化的程序中通常只有三种循环，分别是简单循环、串接循环和嵌套循环，如图 4.6 所示。下面分别讨论不同类型循环的测试方法。

(1) 简单循环

应该使用下列测试集来测试简单循环，其中 n 是允许通过循环的最大次数。

- ? 跳过循环。
- ? 只通过循环一次。
- ? 通过循环两次。
- ? 通过循环 m 次，其中 $m < n - 1$ 。
- ? 通过循环 $n - 1, n, n + 1$ 次。

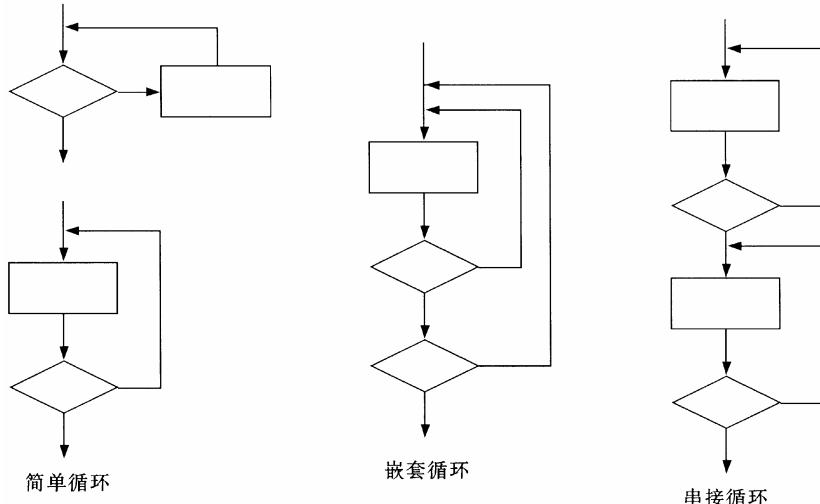


图 4.6 三种循环

(2) 嵌套循环

如果把简单循环的测试方法直接应用到嵌套循环，可能的测试数就会随嵌套层数的增加按几何级数增长，这会导致不切实际的测试数目。B.Beizer 提出了一种能减少测试数的方法。

- ? 从最内层循环开始测试，把所有其他循环都设置为最小值。
- ? 对最内层循环使用简单循环测试方法，而使外层循环的迭代参数（例如，循环计数器）取最小值，并为越界值或非法值增加一些额外的测试。
- ? 由内向外，对下一个循环进行测试，但保持所有其他外层循环为最小值，其他嵌套循环为“典型”值。
- ? 继续进行下去，直到测试完所有循环。

(3) 串接循环

如果串接循环的各个循环都彼此独立，则可以使用前述的测试简单循环的方法来测试串接循环。但是，如果两个循环串接，而且第一个循环的循环计数器值是第二个循环的初始值，则这两个循环并不是独立的。当循环不独立时，建议使用测试嵌套循环的方法来测试串接循环。

4.4 黑盒测试技术

黑盒测试着重检验程序的功能是否满足对它的需求，也就是说，黑盒测试让软件工程师设计出能充分检查程序所有功能需求的输入条件集。黑盒测试技术并不能取代白盒测试技术，它是与白盒测试技术互补的方法。黑盒测试很可能发现白盒测试不易发现的其他不同类型的错误。

确切地说，黑盒测试力图发现下述类型的错误：

- ? 程序功能不正确或遗漏了用户需要的功能；
- ? 界面错误；
- ? 数据结构错误或外部数据库访问错误；
- ? 性能达不到要求；
- ? 初始化和终止错误。

通常，白盒测试在测试过程的早期阶段进行，而黑盒测试则主要用在测试过程的后期。黑盒测试故意不考虑程序的控制结构，而把注意力集中于信息域。

设计黑盒测试用例时应该仔细考虑下述问题：

- ? 怎样测试程序功能的有效性？
- ? 哪些类型的输入可构成高效的测试用例？
- ? 程序是否对特定的输入值特别敏感？
- ? 怎样划定数据类的边界？
- ? 系统能够承受什么样的数据率和数据量？
- ? 数据的特定组合将对系统运行产生什么影响？

应用黑盒测试技术可以设计出满足下述标准的测试用例集：

1. 所设计出的测试用例能够减少为达到合理测试而需要设计的附加测试用例的数目；
2. 所设计出的测试用例能够告诉我们，是否存在某些类型的错误，而不是仅仅指出与特定测试相关的错误是否存在。

4.4.1 等价划分

等价划分是一种黑盒测试技术，这种方法首先把程序的输入域划分成若干个数据类，然后根据划分出的输入数据种类设计测试用例。一个理想的测试用例能够独自发现一类错误（例如，对所有字符数据的处理都不正确）。

以前曾经讲过，穷尽的黑盒测试（即使用所有有效的和无效的输入数据测试程序）通常是不现实的。因此，只能选取少量最有代表性的输入数据作为测试数据，以便用较小的代价测试出较多的程序错误。

什么样的输入数据是最有代表性的呢？如果把所有可能的输入数据（既包括有效的输入数据也包括无效的输入数据）划分成若干个等价类，则可以合理地做出下述假定：每类数据中的一个典型值在测试中的作用与这一类中所有其他值的作用相同。因此，可以从每个等价类

中只取一组数值作为测试数据。这样选取的测试数据最有代表性，最可能发现程序中的错误。事实上，等价划分法力图设计出一个能发现若干类错误的测试用例，从而减少必须设计的测试用例的数目。

使用等价划分法设计测试方案首先需要划分输入数据的等价类，为此需要研究程序的功能说明，从而确定输入数据的有效等价类和无效等价类。在确定输入数据的等价类时常常还需要分析输出数据的等价类，以便根据输出数据的等价类导出对应的输入数据等价类。

划分等价类需要经验，下述几条启发式规则可能有助于等价类的划分：

? 如果规定了输入值的范围，则可划分出一个有效的等价类（输入值在此范围内），两个无效的等价类（输入值小于最小值或大于最大值）；

? 如果规定了输入数据的个数，则类似地也可以划分出一个有效的等价类和两个无效的等价类；

? 如果规定了输入数据的一组值，而且程序对不同输入值做不同处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效的等价类（任一个不允许的输入值）；

? 如果规定了输入数据必须遵循的规则，则可以划分出一个有效的等价类（符合规则）和若干个无效的等价类（从各种不同角度违反规则）；

? 如果规定了输入数据为整型，则可以划分出正整数、零和负整数等三个有效类；

? 如果程序的处理对象是表格，则应该使用空表，以及含一项或多项的表。

以上列出的启发式规则只是测试时可能遇到的情况中的很小一部分，实际情况千变万化，根本无法一一列出。为了正确划分等价类，一是要注意积累经验，二是要正确分析被测程序的功能。此外，在划分无效的等价类时还必须考虑编译程序的检错功能，一般说来，不需要设计测试数据用来暴露编译程序肯定能发现的错误。最后说明一点，上面列出的启发式规则虽然都是针对输入数据说的，但是其中绝大部分也同样适用于输出数据。

划分出等价类以后，根据等价类设计测试方案时主要使用下面两个步骤：

(1) 设计一个新的测试方案以尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步骤直到所有有效等价类都被覆盖为止；

(2) 设计一个新的测试方案，使它覆盖一个而且只覆盖一个尚未被覆盖的无效等价类，重复这一步骤直到所有无效等价类都被覆盖为止。

注意，通常程序发现一类错误后就不再检查是否还有其他错误，因此，应该使每个测试方案只覆盖一个无效的等价类。

下面用等价划分法设计一个简单程序的测试方案。

假设有一个把数字串转变成整数的函数。运行程序的计算机字长 16 位，用二进制补码表示整数。这个函数是用 PASCAL 语言编写的，它的说明如下：

```
function stroint (dstr: shortstr): integer;
```

函数的参数类型是 shortstr，它的说明是：

```
type shortstr = array [1..6] of char;
```

被处理的数字串是右对齐的，也就是说，如果数字串比六个字符短，则在它的左边补空格。如果数字串是负的，则负号和最高位数字紧相邻（负号在最高位数字左边一位）。

考虑到 PASCAL 编译程序固有的检错功能，测试时不需要使用长度不等于 6 的数组做实在参数，更不需要使用任何非字符数组类型的实在参数。

分析这个程序的规格说明，可以划分出如下等价类：

有效输入的等价类有

- (1) 1~6个数字字符组成的数字串（最高位数字不是零）；
- (2) 最高位数字是零的数字串；
- (3) 最高位数字左邻是负号的数字串；

无效输入的等价类有

- (4) 空字符串（全是空格）；
- (5) 左部填充的字符既不是零也不是空格；
- (6) 最高位数字右面由数字和空格混合组成；
- (7) 最高位数字右面由数字和其他字符混合组成；
- (8) 负号与最高位数字之间有空格；

合法输出的等价类有

- (9) 在计算机能表示的最小负整数和零之间的负整数；
- (10) 零；
- (11) 在零和计算机能表示的最大正整数之间的正整数；

非法输出的等价类有

- (12) 比计算机能表示的最小负整数还小的负整数；
- (13) 比计算机能表示的最大正整数还大的正整数。

因为所用的计算机字长 16 位，用二进制补码表示整数，所以能表示的最小负整数是 -32768，能表示的最大正整数是 32767。

根据上面划分出的等价类，可以设计出下述测试方案（注意，每个测试方案由三部分内容组成）：

- (1) 1~6个数字组成的数字串，输出是合法的正整数。

输入：'1'

预期的输出：1

- (2) 最高位数字是零的数字串，输出是合法的正整数。

输入：'000001'

预期的输出：1

- (3) 负号与最高位数字紧相邻，输出合法的负整数。

输入：'- 00001'

预期的输出：- 1

- (4) 最高位数字是零，输出也是零。

输入：'000000'

预期的输出：0

- (5) 太小的负整数。

输入：'- 47561'

预期的输出：错误——无效输入

- (6) 太大的正整数。

输入：'132767'

预期的输出：错误——无效输入

(7) 空字符串。

输入：''

预期的输出：错误——没有数字

(8) 字符串左部字符既不是零也不是空格。

输入：'x x x x x'

预期的输出：错误——填充错

(9) 最高位数字后面有空格。

输入：'1 2'

预期的输出：错误——无效输入

(10) 最高位数字后面有其他字符。

输入：'1 x x 2'

预期的输出：错误——无效输入

(11) 负号和最高位数字之间有空格。

输入：'- 12'

预期的输出：错误——负号位置错

4.4.2 边界值分析

经验表明，处理边界情况时程序最容易发生错误。例如，许多程序错误出现在下标、纯量、数据结构和循环等等的边界附近。因此，设计使程序运行在边界情况附近的测试方案，暴露出程序错误的可能性更大一些。

使用边界值分析方法设计测试方案首先应该确定边界情况，这需要经验和创造性，通常输入等价类和输出等价类的边界，就是应该着重测试的程序边界情况。选取的测试数据应该刚好等于、刚刚小于和刚刚大于边界值。也就是说，按照边界值分析法，应该选取刚好等于、稍小于和稍大于等价类边界值的数据作为测试数据，而不是选取每个等价类内的典型值或任意值作为测试数据。

通常设计测试方案时总是联合使用等价划分和边界值分析两种技术。例如，为了测试前述的把数字串转变成整数的程序，除了上一小节已经用等价划分法设计出的测试方案外，还应该用边界值分析法再补充下述测试方案：

(1) 使输出刚好等于最小的负整数。

输入：'- 32768'

预期的输出为： - 32768

(2) 使输出刚好等于最大的正整数。

输入：'32767'

预期的输出：32767

原来用等价划分法设计出来的测试方案(5)最好改为：

(3) 使输出刚刚小于最小的负整数。

输入：'- 32769'

预期的输出：“错误——无效输入”

原来的测试方案(6)最好改为：

(4)使输出刚刚大于最大的正整数。

输入：'32768'

预期的输出：错误——无效输入

此外，根据边界值分析方法的要求，应该分别使用长度为0, 1和6的数字串作为测试数据。上一小节中设计的测试方案1, 2, 3, 4和7已经包含了这些边界情况。

4.4.3 错误推测

使用边界值分析和等价划分技术，可以帮助我们设计出具有代表性的，因而也就容易暴露程序错误的测试方案。但是，不同类型不同特点的程序通常又有一些特殊的容易出错的情况。此外，有时分别使用每组测试数据时程序都能正常工作，这些输入数据的组合却可能检测出程序的错误。一般说来，即使是一个比较小的程序，可能的输入组合数也往往十分巨大，因此必须依靠测试人员的经验和直觉，从各种可能的测试方案中选出一些最可能引起程序出错的方案。对于程序中可能存在哪类错误的推测，是挑选测试方案时的一个重要因素。

错误推测法在很大程度上依靠测试人员的直觉和经验进行。它的基本做法是，列举出程序中可能有的错误和容易发生错误的特殊情况，并且根据它们设计测试方案。对于程序中容易出错的情况已有一些经验总结出来，例如，输入数据值为零或输出数据值为零往往容易发生错误；如果输入或输出的数目允许变化（例如，被检索的或程序生成的表的项数），则输入或输出的数目为0和1的情况（例如，表为空或只有一项）是容易出错的情况。还应该仔细分析程序规格说明书，注意找出其中的遗漏或省略的部分，以便设计相应的测试方案，检测程序员对这些部分的处理是否正确。

此外，经验还告诉我们，在一段程序中已经发现的错误数目往往和尚未发现的错误数成正比。例如，在IBM OS / 370操作系统中，用户发现的全部错误的47%只与该系统4%的模块有关。这个事实再次证实了本章4.2.4小节提到的Pareto原理。因此，在进一步测试时应该着重测试那些已经发现了有较多错误的程序段。

等价划分法和边界值分析法都只孤立地考虑各个输入数据的测试功效，而没有考虑多个输入数据的组合效应，可能会遗漏了输入数据易于出错的组合情况。选择输入组合的一个有效途径是利用判定表或判定树为工具，列出输入数据各种组合与程序应作的动作（及相应的输出结果）之间的对应关系，然后为判定表的每一列至少设计一个测试用例。

选择输入组合的另一个有效途径是把计算机测试和人工检查代码结合起来。例如，通过代码检查发现程序中两个模块使用并修改某些共享的变量，如果一个模块对这些变量的修改不正确，则会引起另一个模块出错，因此这是程序发生错误的一个可能的原因。应该设计测试方案，在程序的一次运行中同时检测这两个模块，特别要着重检测一个模块修改了共享变量后，另一个模块能否像预期的那样正常使用这些变量。反之，如果两个模块相互独立，则没有必要测试它们的输入组合情况。通过代码检查也能发现模块相互依赖的关系，例如，某个算术函数的输入是数字字符串，调用4.4.1节例子中的`sttoint`函数，把输入的数字串转变成内部形式的整数。在这种情况下，不仅必须测试这个转换函数，还应该测试调用它的算术函

数在转换函数接收到无效输入时的响应。

4.5 测 试 策 略

软件测试策略把设计测试用例的方法集成到一系列经过周密计划的测试步骤中去，从而大大提高软件测试的效果，使得软件开发获得成功。任何测试策略都必须与测试计划、测试用例设计、测试执行以及测试结果数据的收集与分析紧密地结合在一起。

4.5.1 测试步骤

除非是测试一个小程序，否则一开始就把整个系统作为一个单独的实体来测试是不现实的。与开发过程类似，测试过程也必须分步骤进行，后一个步骤在逻辑上是前一个步骤的继续。

从过程的观点考虑测试，在软件工程环境中的测试过程，实际上是顺序进行的四个步骤的序列。最开始，着重测试每个单独的模块，以确保它作为一个单元来说功能是正确的。因此，这种测试称为单元测试。单元测试大量使用白盒测试技术，检查模块控制结构中的特定路径，以确保做到完全覆盖并发现最大数量的错误。接下来，必须把模块装配（即集成）在一起形成完整的软件包。在装配的同时进行测试，因此称为集成测试。集成测试同时解决程序验证和程序构造这两个问题。在集成过程中最常用的是黑盒测试用例设计技术，当然，为了保证覆盖主要的控制路径，也可能使用一定数量的白盒测试。在软件集成完成之后，还需要进行一系列高级测试。必须测试在需求分析阶段确定下来的确认标准，确认测试是对软件满足所有功能的、行为的和性能的需求的最终保证。在确认测试过程中仅使用黑盒测试技术。

高级测试的最后一个步骤已经超出了软件工程的范畴，而成为计算机系统工程的一部分。软件一旦经过确认之后，就必须和其他系统元素（例如，硬件、人员、数据库）结合在一起。系统测试的任务是，验证所有系统元素都能正常地配合，从而可以完成整个系统的功能并达到预期的性能。

4.5.2 单元测试

通常，单元测试和编码属于软件工程过程的同一个阶段。在编写出源程序代码并通过了编译程序的语法检查之后，可以应用人工测试和计算机测试这样两种类型的测试，完成单元测试工作。这两种类型的测试各有所长，互相补充。

下面讲述和单元测试有关的问题。

1. 单元测试的重点

在单元测试期间应该着重从下述五个方面对模块进行测试：模块接口，局部数据结构，重要的执行通路，出错处理通路，影响上述各方面特性的边界条件。

（1）模块接口

首先应该对通过模块接口的数据流进行测试。如果数据不能按预定要求进出模块，所有其他测试都是不切实际的。根据经验，在对接口进行测试时主要应该检查下述各点：

? 模块参数数目和由调用它的模块送来的变元的数目是否相等?

- ? 参数的属性和变元的属性是否匹配?
- ? 参数和变元的单位系统是否相同?
- ? 传送给被调用模块的变元的数目是否等于那个模块的参数的数目?
- ? 传送给被调用模块的变元属性与参数属性是否一致?
- ? 传送给被调用模块的变元所用的单位系统与该模块参数的单位系统是否相同?
- ? 传送给库函数的变元属性、数目、次序及所用的单位系统是否正确?
- ? 模块是否修改了只作为输入数据使用的变元?
- ? 全局变量的定义和用法在各个模块中是否一致?

如果一个模块完成外部文件的输入或输出后，还应该再着重检查下述各点：

- ? 文件属性是否正确?
- ? 打开文件的语句是否正确?
- ? 缓冲区大小与记录长度是否匹配?
- ? 使用文件之前先打开文件了吗?
- ? 使用文件之后关闭文件了吗?
- ? 输入/输出错误检查并处理了吗?
- ? 输出信息中的文字书写有错误吗?

(2) 局部数据结构

对于一个模块而言，局部数据结构是常见的错误来源。应该仔细设计测试方案，以便发现下述类型的错误：

- ? 错误的或不相容的数据说明；
- ? 使用了尚未赋值或尚未初始化的变量；
- ? 错误的初始值或不正确的缺省值；
- ? 变量名字不正确（拼写错或截短了）；
- ? 数据类型不相容；
- ? 上溢、下溢或地址异常。

除了局部数据结构之外，如果可能，在单元测试期间还应该查明全局数据对模块的影响。

(3) 重要的执行通路

由于通常不可能进行穷尽测试，因此，在单元测试期间选择最有代表性、最可能发现错误的执行通路进行测试就是十分关键的。应该设计测试方案用来发现由于错误的计算、不正确的比较或不适当的控制流而造成的错误。在计算中比较常见的错误是：

- ? 计算次序不对或误解了运算符的优先次序；
- ? 混合运算（运算对象的类型彼此不相容）；
- ? 变量初始值不正确；
- ? 精度不够；
- ? 表达式的符号表示错误。

比较和控制流彼此间是紧密结合的，比较之后常常发生控制流的变化。测试方案应该能发现下面这样一些常见的错误：

- ? 比较数据类型不同的量；
- ? 逻辑运算符不正确或误解了逻辑运算符的优先次序；

- ? 当由于精度问题两个量不可能相等时，程序中却期待着相等条件的出现；
- ? “差1”错（即，多循环一次或少循环一次）；
- ? 错误的或不可能达到的循环终止条件；
- ? 当遇到发散的迭代时循环不能终止；
- ? 错误地修改循环变量。

(4) 出错处理通路

好的设计应该能预见出现错误的条件，并且设置适当的处理错误的通路，以便在真的出现错误时执行相应的出错处理通路或干净地结束处理。不仅应该在程序中包含出错处理通路，而且应该认真测试这种通路。当评价出错处理通路时，应该着重测试下述一些可能发生的错误：

- ? 对错误的描述是难于理解的；
- ? 记下的错误与实际遇到的错误不同；
- ? 在对错误进行处理之前，错误条件已经引起系统干预；
- ? 对错误的处理不正确；
- ? 描述错误的信息不足以帮助确定造成错误的位置。

(5) 边界条件

边界测试是单元测试中最后的也可能是最重要的任务。软件常常在它的边界上失效，例如，处理 n 元数组的第 n 个元素时，或做到 i 次循环中的第 i 次重复时，往往会发生错误。使用刚好小于、刚好等于和刚好大于最大值或最小值的数据结构、控制量和数据值的测试方案，非常可能发现软件中的错误。

2. 代码审查

人工测试源程序可以由编写者本人非正式地进行，也可以由审查小组正式进行。后者称为代码审查，它是一种非常有效的程序验证技术，对于典型的程序来说，可以查出30%~70%的逻辑设计错误和编码错误。审查小组最好由下述四人组成：

- ? 组长，他应该是一个很有能力的程序员，而且没有直接参与这项工程；
- ? 程序的设计者；
- ? 程序的编写者；
- ? 程序的测试者。

如果一个人既是程序的设计者又是编写者，或既是编写者又是测试者，则审查小组中应该再增加一个程序员。

审查之前，小组成员应该先研究设计说明书，力求理解这个设计。为了帮助理解，可以先由设计者扼要地介绍他的设计。在审查会上由程序的编写者解释他是怎样用程序代码实现这个设计的，通常是逐个语句地讲述程序的逻辑，小组其他成员仔细倾听他的讲解，并力图发现其中的错误。当发现错误时由组长记录下来，审查会继续进行（审查小组的任务是发现错误而不是改正错误）。

审查会还有另外一种常见的进行方法（称为预排）：由一个人扮演“测试者”，其他人扮演“计算机”。会前测试者准备好测试方案，会上由扮演计算机的成员模拟计算机执行被测试的程序。当然，由于人执行程序速度极慢，因此测试数据必须简单，测试方案的数目也不能过多。但是，测试方案本身并不十分关键，它只起一种促进思考引起讨论的作用。在大多数

情况下，通过向程序员提出关于他的程序的逻辑和他编写程序时所做的假设的疑问，可以发现的错误比由测试方案直接发现的错误还多。

代码审查比计算机测试优越的是：一次审查会上可以发现许多错误；用计算机测试的方法发现错误之后，通常需要先改正这个错误才能继续测试，因此错误是一个一个地发现并改正的。也就是说，采用代码审查的方法可以减少系统验证的总工作量。

实践表明，对于查找某些类型的错误来说，人工测试比计算机测试更有效；对于其他类型的错误来说则刚好相反。因此，人工测试和计算机测试是互相补充，相辅相成的，缺少其中任何一种方法都会使查找错误的效率降低。

3. 驱动程序和存根程序

模块并不是一个独立的程序，因此必须为每个单元测试开发驱动软件和（或）存根软件。通常驱动程序也就是一个“主程序”，它接收测试数据，把这些数据传送给被测试的模块，并且印出有关的结果。存根程序代替被测试的模块所调用的模块。因此存根程序也可以称为“虚拟子程序”。它使用被它代替的模块的接口，可能做最少量的数据操作，印出对入口的检验或操作结果，并且把控制归还给调用它的模块。

例如，图 4.7 是一个正文加工系统的部分层次图，假定要测试其中编号为 3.0 的关键模块——正文编辑模块。因为正文编辑模块不是一个独立的程序，所以需要有一个测试驱动程序来调用它。这个驱动程序说明必要的变量，接收测试数据——字符串，并且设置正文编辑模块的编辑功能。因为在原来的软件结构中，正文编辑模块通过调用它的下层模块来完成具体的编辑功能，所以需要有存根程序简化地模拟这些下层模块。为了简单起见，测试时可以设置的编辑功能只有修改 (CHANGE) 和添加 (APPEND) 两种，用控制变量 CFUNCT 标记要求的编辑功能，而且只用一个存根程序模拟正文编辑模块的所有下层模块。下面是用伪码书写的存根程序和驱动程序。

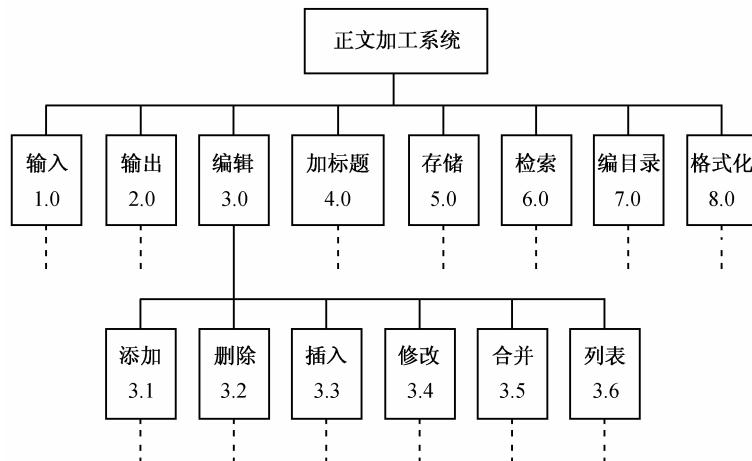


图 4.7 正文加工系统的层次图

. TEST STUB (*测试正文编辑模块用的存根程序*)

初始化；

输出信息“进入了正文编辑程序”；

```

    输出“输入的控制信息是”CFUNCT ;
    输出缓冲区中的字符串 ;
    IF CFUNCT = CHANGE
        THEN
            把缓冲区中第二个字改为***
        ELSE
            在缓冲区的尾部加???
    END IF ;
    输出缓冲区中的新字符串 ;
END TEST STUB
.TEST DRIVER (*测试正文编辑模块用的驱动程序*)
    说明长度为 2500 个字符的一个缓冲区 ;
    把 CFUNCT 置为希望测试的状态 ;
    输入字符串 ;
    调用正文编辑模块 ;
    停止或再次初启 ;
END TEST DRIVER

```

驱动程序和存根程序代表开销，也就是说，为了进行单元测试必须编写测试软件，但是通常并不把它们作为软件产品的一部分交给用户。许多模块不能用简单的测试软件充分测试，为了减少开销可以使用下节将要介绍的渐增式测试方法，在集成测试的过程中同时完成对模块的详尽测试。

模块的内聚程度高可以简化单元测试过程。如果每个模块只完成一种功能，则需要的测试方案数目将明显减少，模块中的错误也更容易预测和发现。

4.5.3 集成测试

集成测试是测试和组装软件的系统化技术，在把模块按照设计要求组装起来的同时进行测试，主要目标是发现与接口有关的问题。例如，数据穿过接口时可能丢失；一个模块对另一个模块可能有由于疏忽而造成的有害影响；把子功能组合起来可能不产生预期的主功能；个别看来是可以接受的误差可能积累到不能接受的程度；全程数据结构可能有问题等等。不幸的是，可能发生的接口问题多得不胜枚举。

由模块组装成程序时有两种方法。一种方法是先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序，这种方法称为非渐增式测试方法；另一种方法是把下一个要测试的模块同已经测试好的那些模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合起来测试。这种每次增加一个模块的方法称为渐增式测试。

非渐增式测试一下子把所有模块放在一起，并把整个程序作为一个整体来进行测试，测试者面对的场面往往混乱不堪。测试时会遇到许许多多的错误，改正错误更是极端困难，因为在庞大的程序中想要诊断定位一个错误是非常复杂非常困难的。而且一旦改正一个错误之后，马上又会遇到新的错误，这个过程会继续下去，看起来好像永远也没有尽头。

渐增式测试与“一步到位”的非渐增式测试相反，把程序划分成小段来构造和测试，在

这个过程中比较容易分离和改正错误；对接口可能进行更彻底的测试；而且可以使用系统化的测试方法。因此，在进行集成测试时普遍使用渐增式测试方法。下面讨论两种不同的渐增式集成策略。

1. 自顶向下集成

自顶向下的集成（结合）方法是一个日益为人们广泛采用的组装软件的途径。从主控制模块（主程序）开始，沿着软件的控制层次向下移动，从而逐渐把各个模块结合起来。在把附属于（以及最终附属于）主控制模块的那些模块组装到软件结构中去时，或者使用深度优先的策略，或者使用宽度优先的策略。

参看图 4.8，深度优先的结合方法先组装在软件结构的一条主控制通路上的所有模块。选择一条主控制通路取决于应用的特点，并且有很大任意性。例如，选取左通路，首先结合模块 M_1 、 M_2 和 M_5 ；其次， M_8 或 M_6 （如果为了使 M_2 具有适当功能需要 M_6 的话）将被结合进来。然后构造中央的和右侧的控制通路。而宽度优先的结合方法，是沿软件结构水平地移动，把处于同一个控制层次上的所有模块组装起来。对于图 4.8 来说，首先结合模块 M_2 、 M_3 和 M_4 （代替存根程序 S_4 ）。然后结合下一个控制层次中的模块 M_5 、 M_6 和 M_7 ；如此继续进行下去，直到所有模块都被结合进来为止。

把模块结合进软件结构的具体过程由下述四个步骤完成：

第一步，对主控制模块进行测试，测试时用存根程序代替所有直接附属于主控制模块的模块；

第二步，根据选定的结合策略（深度优先或宽度优先），每次用一个实际模块代换一个存根程序（新结合进来的模块往往又需要新的存根程序）；

第三步，在结合进一个模块的同时进行测试；

第四步，为了保证加入模块没有引进新的错误，可能需要进行回归测试（即，全部或部分地重复以前做过的测试）。

从第二步开始不断地重复进行上述过程，直到构造起完整的软件结构为止。图 4.8 描绘了这个过程。假设选取深度优先的结合策略，软件结构已经部分地构造起来了，下一步存根程序 S_7 将被模块 M_7 取代。 M_7 可能本身又需要存根程序，以后这些存根程序也将被相应的模块所取代。

自顶向下的结合策略能够在测试的早期对主要的控制或关键的抉择进行检验。在一个分解得好的软件结构中，关键的抉择位于层次系统的较上层，因此首先碰到。如果主要控制确实有问题，早期认识到这类问题是很有好处的，可以及早想办法解决。如果选择深度优先的结合方法，可以在早期实现软件的一个完整的功能并且验证这个功能。早期证实软件的一个完整功能，可以增强开发人员和用户双方的信心。

自顶向下的方法讲起来比较简单，但是实际使用时可能遇到逻辑上的问题。这类问题中最常见的是，为了充分地测试软件系统的较高层次，需要在较低层次上的处理。然而在自顶

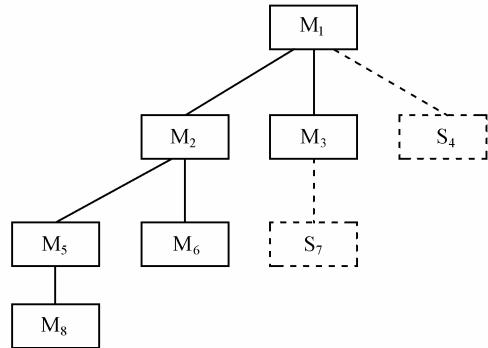


图 4.8 自顶向下结合

向下测试的初期，存根程序代替了低层次的模块，因此，在软件结构中没有重要的数据自下往上流。为了解决这个问题，测试人员有两种选择：

第一，把许多测试推迟到用真实模块代替了存根程序以后再进行；

第二，从层次系统的底部向上组装软件。

第一种方法失去了在特定的测试和组装特定的模块之间的精确对应关系，这可能导致在确定错误的位置和原因时发生困难。后一种方法称为自底向上的测试，下面讨论这种方法。

2. 自底向上集成

自底向上测试从“原子”模块（即在软件结构最低层的模块）开始组装和测试。因为是从底部向上结合模块，总能得到需要的下层模块处理功能，所以不需要存根程序。

用下述步骤可以实现自底向上的结合策略：

第1步，把低层模块组合成实现某个特定的软件子功能的族；

第2步，写一个驱动程序（用于测试的控制程序），协调测试数据的输入和输出；

第3步，对由模块组成的子功能族进行测试；

第4步，去掉驱动程序，沿软件结构自下向上移动，把子功能族组合起来形成更大的子功能族。

上述第2步到第4步实质上构成了一个循环。图4.9描绘了自底向上的结合过程。首先把模块组合成簇1、簇2和簇3，使用驱动程序（图中用虚线方框表示）对每个子功能簇进行测试。簇1和簇2中的模块附属于模块M_a，去掉驱动程序D₁和D₂，把这两个簇直接同M_a连接起来。类似地，在和模块M_b结合之前去掉簇3的驱动程序D₃。最终M_a和M_b这两个模块都与模块M_c结合起来。

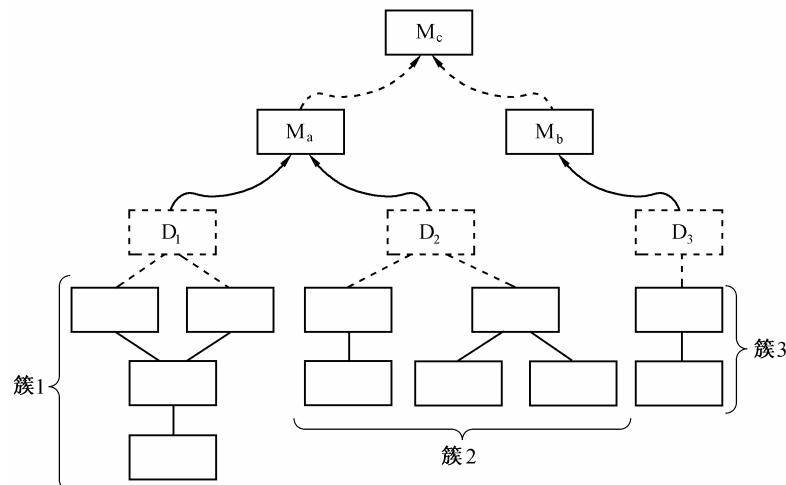


图4.9 自底向上结合

随着结合向上移动，对测试驱动程序的需要也减少了。事实上，如果软件结构的顶部两层用自顶向下的方法组装，可以明显减少驱动程序的数目，而且簇的结合也将大大简化。

3. 两种集成测试策略的比较

上面介绍了集成测试的两种策略，到底哪种方法更好一些呢？一般说来，一种方法的优点正好对应于另一种方法的缺点。自顶向下测试方法的主要优点是不需要测试驱动程序，能够

在测试阶段的早期实现并验证系统的主要功能，而且能在早期发现上层模块的接口错误。自顶向下测试方法的主要缺点是需要存根程序，可能遇到与此相联系的测试困难，低层关键模块中的错误发现较晚，而且用这种方法在早期不能充分展开人力。可以看出，自底向上测试方法的优缺点与上述自顶向下测试方法的优缺点刚好相反。

在测试实际的软件系统时，应该根据软件的特点以及工程进度安排，选用适当的测试策略。一般说来，纯粹自顶向下或纯粹自底向上的策略可能都不实用，人们在实践中创造出许多混合策略：

改进的自顶向下测试方法 基本上使用自顶向下的测试方法，但是在早期，就使用自底向上的方法测试软件中的少数关键模块。一般的自顶向下方法所具有的优点在这种方法中也都有，而且能在测试的早期发现关键模块中的错误；但是，它的缺点也比自顶向下方法多一条，即，测试关键模块时需要驱动程序。

混合法 对软件结构中较上层，使用的是自顶向下方法；对软件结构中较下层，使用的是自底向上方法，两者相结合。这种方法兼有两种方法的优点和缺点，当被测试的软件中关键模块比较多时，这种混合法可能是最好的折衷方法。

在进行集成测试的时候，测试人员应该尽早识别出被测程序中的关键模块。所谓关键模块是指具有下述一个或多个特征的模块：

- (1) 与多项软件需求有关；
- (2) 含有高层控制功能（模块位于程序结构的较高层次）；
- (3) 模块本身是复杂的或容易出错的（可以用环形复杂度指示模块的复杂程度）；
- (4) 有确定的性能需求。

在集成测试期间应该尽可能早地测试关键模块，此外，回归测试也应该着重测试关键模块的功能。

4. 回归测试

在集成测试期间，每当一个新模块作为被测程序的一部分加进来的时候，软件就发生了变化：可能建立了新的数据流路径，也可能出现了新的I/O操作或者激活了新的控制逻辑。这些变化有可能使原来正常工作的程序功能出现问题。在集成测试范畴中，所谓回归测试是指重新执行已经做过的测试的某个子集，以保证上述这些变化没有带来非预期的副作用。

更广义地说，任何成功的测试都会发现错误，而且必须改正所发现的错误。每当改正软件错误的时候，软件配置的某些成分（程序、文档或数据）也被修改了。回归测试就是用于保证由于测试或其他原因引起的软件变化，不会导致非预期的行为或额外错误的测试活动。

回归测试可以通过重新执行全部测试用例的一个子集人工进行，也可以使用自动化的捕获回放工具自动进行。利用捕获回放工具，软件工程师能够捕获测试用例和实际运行结果，然后可以回放（即重新执行测试用例），并且把回放时得到的运行结果与以前的运行结果相比，以判断软件变化是否导致了非预期的软件行为或额外错误。

回归测试集（已执行过的测试用例的子集）包括下述三种不同的测试用例：

- ? 检测软件全部功能的代表性测试用例；
- ? 专门针对可能受修改影响的软件功能的附加测试；
- ? 针对被修改过的软件成分设计的测试。

在集成测试期间，回归测试的数量可能变得非常大。因此，应该把回归测试集设计为只

包括下述那样一些测试，这些测试检测程序每个主要功能中的一类或若干类错误。如果一旦修改软件之后就重新执行检测程序每个功能的全部测试用例，则将是低效而且不切实际的。

4.5.4 确认测试

确认测试也称为验收测试，它的目标是验证软件的有效性。

上面我们使用了确认（Validation）和验证（Verification）这样两个不同的术语，为了避免混淆，首先扼要地解释一下这两个术语的含义。通常，验证指的是保证软件正确地实现了某一特定要求的一系列活动，而确认指的是保证软件的实现满足了用户需求的一系列活动。B.W.Boehm 用另一种方式说明了这两个术语的区别。

验证：“我们是否正确地构造了产品？”

确认：“我们是否构造了正确的产品？”

那么，什么样的软件才是有效的呢？软件有效性的一个简单定义是：

如果软件的功能和性能如同用户所合理地期待的那样，那么，软件就是有效的。

需求分析阶段产生的软件需求规格说明，准确地描述了用户对软件的合理期望，因此是软件有效性的标准，也是进行确认测试的基础。

1. 确认测试的范围

确认测试必须有用户积极参与，或者以用户为主进行。用户应该参加设计测试方案，使用用户接口输入测试数据并且分析评价测试的输出结果。为了使用户能够积极主动地参与确认测试，特别是为了使用户能有效地使用这个系统，通常在验收之前由开发部门对用户进行培训。

确认测试一般使用黑盒测试法。应该仔细设计测试计划和测试过程，测试计划包括要进行的测试的种类和进度安排，测试过程规定用来检验软件是否与需求一致的测试方案。通过测试要保证软件能满足所有功能要求，能达到每个性能要求，文档资料是准确而完整的，此外，还应该保证软件能满足其他预定的要求（例如，可移植性、兼容性和可维护性等）。

确认测试有两种可能的结果：

? 功能和性能与用户要求一致，软件是可以接受的；

? 功能或性能与用户的要求有差距。

在确认测试期间发现的问题，往往和需求分析阶段分析员所犯的错误有关，而且这类问题的涉及面通常比较广，因此解决起来也比较困难。为了确定解决确认测试过程中发现的软件缺陷或错误的策略，软件工程师通常需要和用户充分协商。

2. 复查软件配置

确认测试的一项重要任务是复查软件配置。复查的目的是，保证软件配置的所有成分都齐全，各方面的质量都符合要求，文档内容与程序完全一致，具有软件维护阶段所必须的细节，而且全部文档都已经编好目录。

除了按软件开发合同规定的内容和要求，由人工审查软件配置之外，在确认测试的过程中还应该严格遵循用户手册以及其他操作程序，以便检验这些手册的完整性和正确性。必须仔细记录测试期间发现的遗漏或错误，并且适当地补充和改正。

3. Alpha 测试和 Beta 测试

如果软件是一个客户开发的，则可以进行一系列验收测试以便用户确认所有需求都已满足了。验收测试是由最终用户而不是系统的开发者进行的。事实上，验收测试可以持续几

个星期或几个月，因此可以发现随着时间流逝可能会降低系统质量的累积错误。

如果一个软件是为许多客户开发的（例如，向大众出售的盒装软件产品），那么让每个客户都进行正式的验收测试是不现实的。在这种情况下，绝大多数软件开发商都使用被称为 Alpha 测试和 Beta 测试的过程，来发现那些看起来只有最终用户才能发现的错误。

Alpha 测试由用户在开发者的场所进行，并且在开发者对用户的“指导”下进行测试。开发者负责记录错误和使用中遇到的问题。总之，Alpha 测试是在受控的环境中进行的。

Beta 测试由软件的最终用户们在一个或多个客户场所进行。与 Alpha 测试不同，开发者通常不在 Beta 测试的现场，因此，Beta 测试是软件在开发者不能控制的环境中的“真实”应用。用户记录下在 Beta 测试过程中遇到的一切问题（真实的或想象的），并且定期把这些问题报告给开发者。接收到 Beta 测试期间报告的问题之后，软件开发者对产品进行修改，并准备向全体客户发布最终的软件产品。

4.6 调试

仅就测试而言，它的目标是发现软件中的错误，但是，发现错误并不是我们的最终目的。软件工程的根本目标，是开发出高质量的完全符合用户需要的软件产品，因此，通过测试发现软件错误之后还必须诊断并改正软件错误，这就是调试（也称为纠错）的任务。

调试作为成功的测试的后果而出现，也就是说，调试是在测试发现错误之后排除错误的过程。虽然调试可以而且应该是一个有序的过程，但是在很大程度上它仍然是一项技巧。软件工程师在评估测试结果时，往往仅面对着软件问题的症状，也就是说，错误的外部表现和它的内在原因之间可能并没有明显的联系。调试就是把症状和原因联系起来的尚未被人很好理解的智力过程。

4.6.1 调试过程

调试不是测试，但是它与测试关系密切，它总是发生在测试之后。如图 4.10 所示，调试

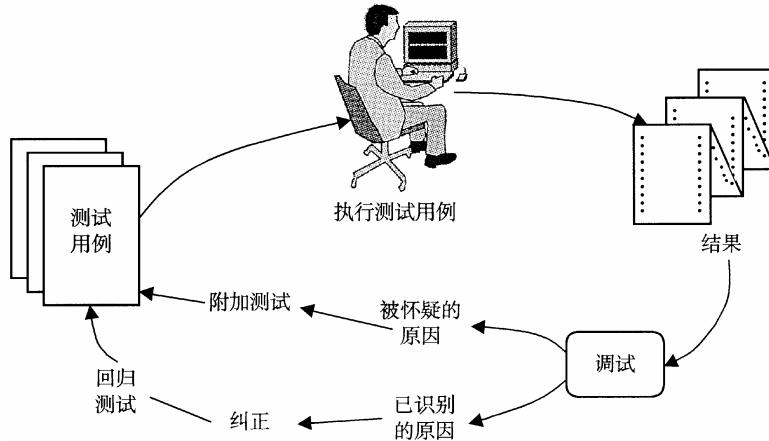


图 4.10 调试过程

过程从执行一个测试用例开始，评估测试结果，如果发现实际结果与预期结果不一致，则这种不一致就是一个症状，它表明在软件中存在着隐藏的问题。调试过程试图找出产生症状的原因，并且改正软件错误。

调试过程总会产生以下两种结果之一：找到了出现问题的原因并把问题改正和排除掉了；没有找到出现问题的原因。在后一种情况下，调试人员可以猜想一个原因，并且设计测试用例来验证这个假设，重复这个过程直到找出出现问题的原因并改正错误。

调试是软件开发过程中最艰巨的脑力劳动。调试工作如此困难，可能心理方面的原因多于技术方面的原因，但是，下述软件错误的特征也是相当重要的原因：

? 症状和产生症状的原因在程序中可能相距甚远，也就是说，症状可能在程序的一个地方出现，而产生症状的实际原因（程序的逻辑错误）可能在程序中与之相距很远的另一个地方。紧耦合的程序结构更加剧了这种情况。

- ? 当改正了另一个错误之后，症状可能暂时消失了。
- ? 症状可能并不是由程序错误引起的（例如，舍入误差引起的症状）。
- ? 症状可能是由不易跟踪的人为错误引起的。
- ? 症状可能是由定时问题而不是由处理问题引起的。
- ? 可能很难再现与出现症状时完全一样的输入条件（例如，输入顺序不确定的实时应用系统）。
- ? 症状可能时有时无，这种情况在硬件和软件紧密地耦合在一起的嵌入式系统中特别常见。
- ? 症状可能是由分布在许多任务中的原因共同引起的，而且这些任务运行在不同的处理器上。

在调试过程中会遇到恼人的小错误（例如，输出格式不正确），到灾难性的大错误（例如，系统失效导致严重的经济损失）等各种不同的错误。错误的后果越严重，尽快查找出错误原因并改正错误的压力也越大。通常，这种压力会导致软件开发人员在改正一个错误的同时引入两个甚至更多个新错误。因此，软件开发人员在调试软件时一定不能慌乱，必须通盘考虑，慎重行事，修改软件后还必须进行充分的回归测试。

4.6.2 调试途径

无论采用什么方法，调试的根本目标都是寻找出现软件错误的原因并正确地改正。这个目标是通过把系统的评估、直觉和运气结合起来实现的。常见的调试途径有蛮干法、回溯法和原因排除法等三种方法，下面简要地介绍这三种方法。

1. 蛮干法

蛮干法可能是为了找到软件错误的原因而最常使用的最低效的方法。仅当所有其他方法都失败了的情况下，才应该使用这种方法。按照“让计算机自己寻找错误”的策略，这种方法打印出内存内容，激活对运行过程的跟踪并在程序中到处都写上 WRITE（输出）语句，希望在这样生成的信息海洋中的某个地方发现错误原因的线索。虽然所生成的大量信息也可能最终导致成功，但是，在更多的情况下这样做只会浪费时间和精力。在使用任何一种调试方法之前，必须首先进行周密的思考，必须有明确的目的，尽量减少无关信息的数量。

2. 回溯法

回溯是一种相当常用的调试方法，当调试小程序时这种方法是有效的。这种方法的具体做法是，从发现症状的地方开始，人工沿程序的控制流往回追踪源程序代码，直到找出错误原因为止。但是，随着程序规模扩大，应该回溯的路径数目也变得越来越大，以至彻底回溯变成完全不可能了。

3. 原因排除法

调试的第三种方法（原因排除法），采用对分查找法或归纳法或演绎法完成调试工作。

对分查找法的基本思路是，如果已经知道每个变量在程序内若干个关键点的正确值，则可以用赋值语句（或输入语句）在程序中点附近“注入”这些变量的正确值，然后运行程序并检查程序的输出。如果输出结果是正确的，则错误原因在程序的前半部分；反之，错误原因在程序的后半部分。对错误原因所在的那部分再重复使用这个方法，直到把出错范围缩小到容易诊断的程度为止。

归纳法是从个别现象推断出一般性结论的思维方法。采用这种方法调试程序时，首先把和错误有关的数据组织起来进行分析，以便发现可能的错误原因。然后导出对错误原因的一个或多个假设，并利用已有的数据来证明或排除这些假设。当然，如果已有的数据尚不足以证明或排除这些假设，则需设计并执行一些新的测试用例，以获得更多的数据。

演绎法从一般原理或前提出发，经过排除和精化的过程推导出结论。采用这种方法调试程序时，首先设想出所有可能的出错原因，然后试图用测试来排除每一个假设的原因，如果测试表明某个假设的原因可能是真的原因，则对数据进行细化以精确定位错误。

上述每一种方法都可以使用调试工具辅助完成，但是工具并不能代替对全部设计文档和源程序的仔细评估。

如果各种调试方法和调试工具都用过了却仍然找不出错误的原因，则应该请求别人帮助。把遇到的问题向同行陈述并一起分析讨论，往往能开阔思路，很快找出错误原因。

一旦找到错误就必须改正它，但是，前面已经提醒过，改正一个错误可能引入更多的其他错误，以至“得不偿失”。因此，在动手改正软件错误之前，每个软件工程师都应该仔细考虑下述三个问题：

? 是否同样的错误也存在于程序的其他地方？在许多情况下，一个程序错误是由错误的逻辑思维模式引起的，而这种逻辑思维模式也可能用在别的地方。仔细分析这种逻辑模式，可能会发现其他错误。

? 将要进行的修改可能会引入的“下一个错误”是什么？在改正错误之前应该仔细研究源程序（最好也研究设计文档），以评估逻辑和数据结构的耦合程度。如果所要做的修改位于程序的高耦合段中，则在修改时必须特别小心谨慎。

? 为防止今后出现类似的错误，应该做什么？如果我们不仅修改了软件产品还改进了软件过程，则不仅排除了现有程序中的错误，还避免了今后在程序中可能出现的错误。

4.7 软件可靠性

测试阶段的根本目标是消除错误保证软件的可靠性。读者可能会问，什么是软件的可靠

性呢？应该进行多少测试，软件才能达到所要求的可靠程度呢？这些正是本节要着重讨论的问题。

4.7.1 基本概念

1. 软件可靠性的定义

对于软件可靠性有许多不同的定义，其中多数人承认的一个定义是：

软件可靠性是程序在给定的时间间隔内，按照规格说明书的规定成功地运行的概率。

在上述定义中包含的随机变量是时间间隔。显然，随着运行时间的增加，运行时遇到程序错误的概率也将增加，即可靠性随着给定的时间间隔的加大而减少。

根据 IEEE 的定义，术语“错误”的含义是由开发人员造成的软件差错，而术语“故障”的含义是由错误引起软件的不正确行为。在下面的论述中，我们将按照 IEEE 规定的含义使用这两个术语。

2. 软件的可用性

通常用户也很关注软件系统可以使用的程度。一般说来，对于任何其故障是可以修复的系统，都应该同时使用可靠性和可用性衡量它的优劣程度。

软件可用性的一个定义是：软件可用性是程序在给定的时间点，按照规格说明书的规定，成功地运行的概率。

可靠性和可用性之间的主要差别是，可靠性意味着在 0 到 t 这段时间间隔内系统没有失效，而可用性只意味着在时刻 t ，系统是正常运行的。因此，如果在时刻 t 系统是可用的，则有下述种种可能：在 0 到 t 这段时间内，系统一直没失效（可靠）；在这段时间内失效了一次，但是又修复了；在这段时间内失效了两次修复了两次；……

如果在一段时间内，软件系统故障停机时间为 t_{d1}, t_{d2}, \dots ，正常运行时间为 t_{u1}, t_{u2}, \dots ，则系统的稳态可用性为：

$$A_{ss} = \frac{T_{up}}{T_{up} + T_{down}} \quad (4.1)$$

其中

$$T_{up} = t_{ui}, T_{down} = t_{di}$$

如果引入系统平均无故障时间 MTTF 和平均维修时间 MTTR 的概念，则 (4.1) 式可以变成

$$A_{ss} = \frac{MTTF}{MTTF + MTTR} \quad (4.2)$$

平均维修时间 MTTR 是修复一个故障平均需要用的时间，它取决于维护人员的技术水平和对系统的熟悉程度，也和系统的可维护性有重要关系，本书下一章将讨论软件维护问题。平均无故障时间 MTTF 是系统按规格说明书规定成功地运行的平均时间，它主要取决于系统中潜在的错误的数目，因此和测试的关系十分密切。

4.7.2 估算平均无故障时间的方法

软件的平均无故障时间 MTTF 是一个重要的质量指标，往往作为对软件的一项要求，由

用户提出来。为了估算 MTTF，首先引入一些有关的量。

1. 符号

在估算 MTTF 的过程中使用下述符号表示有关的数量：

E_T ——测试之前程序中错误总数；

I_T ——程序长度（机器指令总数）；

——测试（包括调试）时间；

$E_d(\cdot)$ ——在 0 至 T 期间发现的错误数；

$E_c(\cdot)$ ——在 0 至 T 期间改正的错误数。

2. 基本假定

根据经验数据，可以作出下述假定：

(1) 在类似的程序中，单位长度里的错误数 E_T / I_T 近似为常数。美国的一些统计数字表明，通常

$$0.5 \times 10^{-2} \quad E_T / I_T \quad 2 \times 10^{-2}$$

也就是说，在测试之前每 1000 条指令中大约有 5~20 个错误。

(2) 失效率正比于软件中剩余的（潜藏的）错误数，而平均无故障时间 MTTF 与剩余的错误数成反比。

此外，为了简化讨论，假设发现的每一个错误都立即正确地改正了（即，调试过程没有引入新的错误）。因此

$$E_c(\cdot) = E_d(\cdot)$$

剩余的错误数为

$$E_r(\cdot) = E_T - E_c(\cdot) \quad (4.3)$$

单位长度程序中剩余的错误数为

$$e_r(\cdot) = E_T / I_T - E_c(\cdot) / I_T \quad (4.4)$$

3. 估算平均无故障时间

经验表明，平均无故障时间与单位长度程序中剩余的错误数成反比，即

$$\text{MTTF} \approx \frac{1}{K(E_T / I_T - e_r(\cdot) / I_T)} \quad (4.5)$$

其中 K 为常数，它的值应该根据经验选取。美国的一些统计数字表明， K 的典型值是 200。

估算平均无故障时间的公式，可以评价软件测试的进展情况。此外，由 (4.5) 式可得

$$E_c \approx E_T \approx \frac{I_T}{K \cdot \text{MTTF}} \quad (4.6)$$

因此，也可以根据对软件平均无故障时间的要求，估计需要改正多少个错误之后，测试工作才能结束。

4. 估计错误总数的方法

程序中潜藏的错误的数目是一个十分重要的量，它既直接标志软件的可靠程度，又是计算软件平均无故障时间的重要参数。显然，程序中的错误总数 E_T 与程序规模、类型、开发环境、开发方法论、开发人员的技术水平和管理水平等都有密切关系。下面介绍估计 E_T 的两个方法。

(1) 植入错误法

使用这种估计方法，在测试之前由专人在程序中随机地植入一些错误，测试之后，根据测试小组发现的错误中原有的和植入的两种错误的比例，来估计程序中原有错误的总数 E_T 。

假设人为地植入的错误数为 N_s ，经过一段时间的测试之后发现 n_s 个植入的错误，此外还发现了 n 个原有的错误。如果可以认为测试方案发现植入错误和发现原有错误的能力相同，则能够估计出程序中原有错误的总数为

$$\hat{N} \approx \frac{n}{n_s} N_s \quad (4.7)$$

其中 \hat{N} 即是错误总数 E_T 的估计值。

(2) 分别测试法

植入错误法的基本假定是所用的测试方案发现植入错误和发现原有错误的概率相同。但是，人为地植入的错误和程序中原有的错误可能性质很不相同，发现它们的难易程度自然也不相同，因此，上述基本假定可能有时和事实不完全一致。

如果有办法随机地把程序中一部分原有的错误加上标记，然后根据测试过程中发现的有标记错误和无标记错误的比例，估计程序中的错误总数，则这样得出的结果比用植入错误法得到的结果更可信一些。

为了随机地给一部分错误加标记，分别测试法使用两个测试员（或测试小组），彼此独立地测试同一个程序的两个副本，把其中一个测试员发现的错误作为有标记的错误。具体做法是，在测试过程的早期阶段，由测试员甲和测试员乙分别测试同一个程序的两个副本，由另一名分析员分析他们的测试结果。用 T 表示测试时间，假设

- $= 0$ 时错误总数为 B_0 ；
- $= 1$ 时测试员甲发现的错误数为 B_1 ；
- $= 1$ 时测试员乙发现的错误数为 B_2 ；
- $= 1$ 时两个测试员发现的相同错误数为 b_c 。

如果认为测试员甲发现的错误是有标记的，即程序中有标记的错误总数为 B_1 ，则测试员乙发现的 B_2 个错误中有 b_c 个是有标记的。假定测试员乙发现有标记错误和发现无标记错误的概率相同，则可以估计出测试前程序中的错误总数为

$$\hat{B}_0 \approx \frac{B_2}{b_c} B_1 \quad (4.8)$$

使用分别测试法，在测试阶段的早期，每隔一段时间分析员分析两名测试员的测试结果，并且用(4.8)式计算 \hat{B}_0 。如果几次估算的结果相差不多，则可用 \hat{B}_0 的平均值作为 E_T 的估计值。此后一名测试员可以改做其他工作，由余下的一名测试员继续完成测试工作，因为他可以继承另一名测试员的测试结果，所以分别测试法增加的测试成本并不太多。

4.8 小结

实现包括编码和测试两个阶段。按照传统的软件工程方法学，编码是在对软件进行了概

要设计和详细设计之后进行的，编码不过是把软件设计的结果翻译成用某种程序设计语言书写的程序，因此，程序的质量基本上由设计的质量决定。但是，编码使用的语言，特别是写程序的风格，也对程序质量有相当大的影响。

大量实践结果表明，高级程序设计语言较汇编语言有很多优点。因此，除非在非常必要的场合，一般不要使用汇编语言写程序。至于具体选用哪种高级程序设计语言，则不仅要考虑语言本身的特点，还应该考虑使用环境等一系列实际因素。

程序内部的良好文档资料，有规律的数据说明格式，简单清晰的语句构造和输入/输出格式等，都对提高程序的可读性有很大作用，也在相当大的程度上改进了程序的可维护性。

目前，软件测试仍然是保证软件可靠性的主要手段。测试阶段的根本任务是发现并改正软件中的错误。

设计测试方案是测试阶段的关键技术问题，其基本目标是选用尽可能少的高效测试数据，做到尽可能完善的测试，从而尽可能多地发现软件中的错误。

白盒测试和黑盒测试是软件测试的两类不同方法，这两类方法各有所长，相互补充，在测试过程中应该联合使用这两类方法。通常，在测试过程的早期阶段主要使用白盒测试技术，而在测试的后期主要使用黑盒测试技术。

为了设计出有效的测试方案，软件工程师必须深入理解并应用指导软件测试的基本准则。

设计白盒测试方案的技术主要有，逻辑覆盖和控制结构测试；设计黑盒测试方案的技术主要有，等价划分、边界值分析和错误推测。

大型软件的测试应该分阶段进行，通常分为单元测试、集成测试、确认测试和系统测试（如果软件是新开发的计算机系统的一部分）等四个阶段。

在测试过程中发现的软件错误必须及时改正，这就是调试的任务。为了改正错误，首先必须确定错误的准确位置，这是调试过程中最困难的任务，需要审慎周密的思考和推理。改正错误往往需要修正原来的设计，必须通盘考虑而不能“头疼医头脚疼医脚”，应该尽量避免在调试过程中引进新的错误。

测试和调试是软件测试阶段中的两个关系极端密切的过程，它们常常交替进行。

程序中潜藏的错误的数目，直接决定了软件的可靠性。通过测试可以估计出程序中剩余的错误数。根据测试和调试过程中已经发现和改正的错误数，可以估计软件的平均无故障时间；反之，根据要求达到的软件平均无故障时间，可以估计应该发现和改正的错误数，从而能够判断测试阶段何时可以结束。

习 题 四

1. 下面给出的伪码中有一个错误。请仔细阅读这段伪码，说明该伪码的语法特点，找出并改正伪码中的错误。字频统计程序的伪码如下：

```
INITIALIZE the Program  
READ the first text record  
DO WHILE there are more words in the text record  
    DO WHILE there are more words in the text record
```

```
EXTRACT the next text word
SEARCH the word_table for the extracted word
IF the extracted word is found
    INCREMENT the word's occurrence count
ELSE
    INSERT the extracted word into the table
END IF
INCREMENT the words_processed count
END DO at the end of the text record
READ the next text record
END DO when all text records have been read
PRINT the table and summary information
TERMINATE the program

2. 研究下面给出的伪码程序，要求：
(1) 画出它的程序流程图；
(2) 它是结构化的还是非结构化的？说明你的理由；
(3) 若是非结构化的，则
    (a) 把它改造成仅用三种控制结构的结构化程序；
    (b) 写出这个结构化设计的伪码；
    (c) 用盒图表示这个结构化程序。
(4) 找出并改正程序逻辑中的错误。

COMMENT : PROGRAM SEARCHES FOR FIRST N REFERENCES
          TO A TOPIC IN AN INFORMATION RETRIEVAL
          SYSTEM WITH T TOTAL ENTRIES
          INPUT N
          INPUT KEYWORD ( S ) FOR TOPIC
          I = 0
          MATCH = 0
          DO WHILE I < T
              I = I + 1
              IF WORD = KEYWORD
                  THEN MATCH = MATCH + 1
                  STORE IN BUFFER
              END
              IF MATCH = N
                  THEN GOTO OUTPUT
              END
          END
          IF N = 0
```

软件工程

```
THEN PRINT " NO MATCH "
OUTPUT : ELSE CALL SUBROUTINE TO PRINT BUFFER
          INFORMATION
          END
```

3. 在第 2 题的设计中若输入的 N 值或 KEYWORD 不合理，会发生问题。

- (1) 给出这些变量的不合理值的例子；
- (2) 将这些不合理值输入程序会有什么后果？
- (3) 怎样在程序中加入防错措施，以防止出现这些问题？

4. 回答下列问题：

- (1) 什么是模块测试、验收测试和集成测试？它们各有什么特点？

(2) 假设有一个由 1 000 行 FORTRAN 语句构成的程序（经编译后大约有 5 000 条机器指令），你估计在对它进行测试期间将发现多少个错误？为什么？

5. 设计下列伪码程序的语句覆盖和分支覆盖测试用例。假设机器字长为 16 位，若对此程序进行穷尽测试，则共需进行多少次测试？

```
START
INPUT ( A,B,C )
IF A > 5
  THEN X = 10
  ELSE X = 1
END IF
IF B > 10
  THEN Y = 20
  ELSE Y = 2
END IF
IF C > 15
  THEN Z = 30
  ELSE Z = 3
END IF
PRINT ( X,Y,Z )
STOP
```

6. 某图书馆有一个使用 CRT 终端的信息检索系统，该系统有下列四个基本检索命令：

名 称	语 法	操 作
BROWSE (浏览)	b (关键字)	系统搜索给出的关键字，找出字母排列与此关键字最相近的字。然后在屏幕上显示约 20 个加了行号的字，与给出的关键字完全相同的字约在中央
SELECT (选取)	s (屏幕上的行号)	系统创建一个文件保存含有由行号指定的关键字的全部图书的索引，这些索引都有编号（第一个索引的编号为 1，第二个为 2，…依此类推）

续表

名 称	语 法	操 作
DISPLAY (显示)	d (索引号)	系统在屏幕上显示与给定的索引号有关的信息，这些信息与通常在图书馆的目录卡片上给出的信息相同。这条命令接在 BROWSE / SELECT 或 FIND 命令后面用，以显示文件中的索引信息
FIND (查找)	f (作者姓名)	系统搜索指定的作者姓名，并在屏幕上显示该作者的著作的索引号，同时把这些索引存入文件

要求：

- (1) 设计测试数据以全面测试系统的正常操作；
- (2) 设计测试数据以测试系统的非正常操作。

7. 对一个包含 10 000 条机器指令的程序进行一个月集成测试后，总共改正了 15 个错误，此时 $NTTF = 10h$ ；经过两个月测试后，总共改正了 25 个错误（第二个月改正了 10 个错误）， $NTTF = 15h$ 。

要求：

(1) 根据上述数据确定 MTTF 与测试时间之间的函数关系，画出 MTTF 与测试时间 的关系曲线。在画这条曲线时你做了什么假设？

(2) 为做到 $MTTF = 100h$ ，必须进行多长时间的集成测试？当集成测试结束时总共改正了多少个错误，还有多少个错误潜伏在程序中？

8. 如对一个长度为 100 000 条指令的程序进行集成测试期间记录下面的数据：

- (a) 七月一日：集成测试开始，没有发现错误。
- (b) 八月二日：总共改正 100 个错误，此时 $MTTF = 0.4h$
- (c) 九月一日：总共改正 300 个错误，此时， $MTTF = 2h$

根据上列数据完成下列各题：

- (1) 估计程序中的错误总数；
- (2) 为使 MTTF 达到 $10h$ ，必须测试和调试这个程序多长时间？
- (3) 画出 MTTF 和测试时间 之间的函数关系曲线。

9. 在测试一个长度为 24 000 条指令的程序时，第一个月由甲、乙两名测试员各自独立测试这个程序。经一个月测试后，甲发现并改正 20 个错误，使 MTTF 达到 $10h$ 。与此同时，乙发现 24 个错误，其中 6 个甲也发现了。以后由甲一个人继续测试这个程序。问：

- (1) 刚开始测试时程序中总共有多少个潜藏的错误？
- (2) 为使 MTTF 达到 $60h$ ，必须再改正多少个错误？还需用多长测试时间？
- (3) 画出 MTTF 与集成测试时间 之间的函数关系曲线。

第 5 章 面向对象方法学导论

传统的软件工程方法学曾经给计算机软件产业带来了巨大的进步，使用结构化范型开发的许多中、小规模的软件项目获得了成功，从而部分地缓解了软件危机。但是，当把结构化范型应用于大型软件产品的开发时，似乎很少取得成功。此外，使用传统的软件工程方法学开发软件时，生产率提高的幅度远远不能满足社会对计算机软件日益增长的需要，软件重用的程度还很低，所开发出的软件产品仍然很难维护。正如软件工程第七条基本原理所指出的那样，必须承认不断改进软件工程实践的必要性。软件工程作为一门新兴学科，尤其需要不断的发展和完善。

面向对象的软件开发方法在 20 世纪 60 年代后期首次提出，经过将近 20 年的发展，这种技术逐渐得到广泛应用。到了 20 世纪 90 年代，面向对象的软件工程方法学已经成为人们在开发软件时首选的方法学。目前看来，面向对象技术似乎是迄今为止人们所知道的最好的软件开发技术。本章对面向对象方法学做深入浅出的入门介绍，为读者今后进一步深入学习这种方法学奠定了坚实基础。希望进一步学习面向对象方法学的读者，请参阅《软件工程》⁽²⁾。

5.1 一个面向对象的程序实例

为了介绍面向对象的软件工程方法学，我们首先从一个面向对象的程序设计实例谈起。假设对一个简单的图形程序的需求如下所述：

在显示器荧光屏上圆心坐标为 (100, 100) 的位置画一个半径为 40 的圆，在圆心坐标为 (200, 300) 的位置画一个半径为 20 的圆，在圆心坐标为 (400, 150) 的位置画一条弧，弧的起始角度为 30 度，结束角度为 120 度，半径为 50。

怎样设计上述这个程序呢？

5.1.1 用对象分解取代功能分解

如果用传统的结构化方法设计上述的图形程序，我们首先定义两个函数，其中一个函数的功能是在荧光屏上用由参数传入的圆心坐标值和圆半径值画一个圆，另一个函数的功能是在荧光屏上用由参数传入的圆心坐标、半径、起始角度和结束角度之值画一条弧；然后，在主函数中说明几个变量，分别用于保存圆心的 X, Y 坐标，圆的半径，弧的起始角度和结束角度；最后，用赋值语句或输入语句给这些变量赋上指定的值，并用它们作为变元调用相应的函数来画圆和画弧。

传统的程序设计方法，实质上是自顶向下的功能分解，也就是通过逐步求精的过程把程序分解成一系列完成单一处理功能的模块，然后传送适当的变元来调用这些模块以完成整个程序的功能。

传统的程序设计方法是面向过程的设计方法，这种方法以算法为核心，把数据和处理过程作为相互独立的部分，数据代表问题域中的实体，而程序代码则用于处理这些数据。

把数据和代码作为分离的实体，反映了计算机的观点，因为在计算机内部数据和程序代码是分开存放的。但是，这样做的时候总存在使用错误的数据调用正确的程序模块，或使用正确的数据调用错误的程序模块的危险。使数据和操作保持一致，是软件工程师的一个沉重负担，在多人分工合作开发一个大型软件的过程中，如果负责设计数据结构的人中途改变了对某个数据的设计，而又没有及时通知所有有关人员，则会发生许多不该发生的错误。

实际上，用计算机解决的问题都是现实世界中的问题，这些问题无非由一些相互间存在一定联系的事物（或称为实体）所组成。每个具体的事物都具有行为和属性两方面的特征，因此，把描述事物静态属性的数据结构和表示事物动态行为的操作放在一起构成一个整体，才能完整、自然、准确地代表客观世界中的实体。

面向对象的程序设计技术以对象（Object）为核心，用这种技术开发出的程序由一系列对象组成。对象是对现实世界实体的正确抽象，它是由描述内部状态、表示静态属性的数据，以及可以对这些数据施加的操作（实现对象的动态行为），封装在一起所构成的统一体。对象之间通过传递消息互相通信，以模拟现实世界中不同实体彼此之间的联系。

传统的程序设计方法把精力集中于设计解题算法（即处理数据的过程），因此也称为面向过程的程序设计方法。这样做实质上也是在用计算机的观点进行程序设计工作。因为计算机的工作过程是一步一步进行的，为了完成指定的功能必须告诉它详细的解题步骤，也就是必须向计算机详细描述解题算法。面向过程程序设计就是按照计算机的要求，围绕算法进行程序设计。设计者站在计算机的立场，“设身处地”地设计解题步骤，并用适当的程序设计语言把解题步骤描述出来。

但是，计算机观点与人类观点终究有很大区别，面向过程的思维方式也并不符合人类习惯的思维方式。正由于用面向过程方法开发软件的方法与过程，不同于人类认识世界解决问题时习惯采用的方法与过程，因此使得实现解法的解空间与描述问题的问题空间在结构上明显不同，这不仅增加了开发软件的难度，也使得所开发出的软件难于理解。

那么，什么是人类习惯采用的解决问题的方法呢？让我们观察一个日常生活中常见的事例：一位厨师头发长了需要理发，他走进理发馆，告诉理发师要理什么发式。也就是说，为了解决头发过长的问题，厨师只需向理发师提出要求，告诉他“做什么”（即，理什么发式），并不需要告诉理发师“怎样做”，理发师自己知道工作步骤。类似地，理发师肚子饿了，只需走进餐馆点好自己要吃的菜，厨师自己知道怎样做菜，并不需要顾客告诉他做菜的具体步骤，事实上顾客并不需要知道做菜的步骤。

从上述事例可以看出，人类习惯的解决问题的方法是使用“顾客—服务员”的工作模式。人类社会中不同职业的人具有不同技能，需要完成一项复杂任务时，只需把具有完成这项任务所需技能的各类人员集中起来，向每个人提出具体要求。至于每个人如何完成自己承担的具体任务，并不需要在布置任务时详细说明，因为他们具备完成自己承担的任务所需要的技能。

面向对象程序设计方法模仿人类习惯的解题方法，用对象分解取代功能分解，也就是把程

序分解成一系列对象，每个对象都既有自己的数据（描述该对象所代表的实体的属性），又有处理这些数据的函数（通常称为服务或方法，它们实现该对象应有的行为）。不同对象之间通过发送消息向对方提出服务要求，接受消息的对象主动完成指定功能提供所要求的服务。程序中所有对象分工协作，共同完成整个程序的功能。事实上，对象是组成面向对象程序的基本模块。面向对象程序设计方法的提出，是软件开发方法的一次革命，它代表了计算机程序设计的一种新颖的思维方法，是解决软件开发所面临的困难的最有希望的方法之一。

具体到我们这个简单的图形程序，从本节开头给出的需求陈述中很容易看出，这个程序中只涉及两类实体（用面向对象术语说，是两类对象），它们分别是圆（Circle）和弧（Arc）。在这个问题中实际上要求画两个具体的圆和一条具体的弧，换句话说，在问题域中有圆类的两个实例和弧类的一个实例。所谓“实例”也就是具体的对象。

从需求陈述中不难看出，圆的基本属性是圆心坐标和半径，弧的基本属性是圆心坐标、半径、起始角度和结束角度。但是，通常不可能在需求陈述中找到所有属性，还必须借助于领域知识和常识，才能分析得出所需要的全部属性。众所周知，一个图形既可以在荧光屏上显示出来，也可以不显示出来。也就是说，一个图形可以处于两种可能的状态之一（可见或不可见），因此，本问题中的圆和弧都应该再增加一个属性——可见性。

分析需求陈述得知，圆和弧都应该提供在荧光屏上“画自己”的服务。所谓画自己，就是用当前的前景颜色在屏幕上显示自己的形状。这个例子是一个相当简单的图形应用程序，它的功能很简单，在需求陈述中只提出了这一项最基本的要求。但是，根据常识我们知道，一个图形既可以在屏幕上显示出来，也可以隐藏起来（实际上是用背景颜色显示）。在属性中我们已经设置了“可见性”这个属性来标志图形当前是否处于可见状态，因此，相应地也应该提供“隐藏自己”这样一个服务。

此外，为了便于使用，通常对象的每个属性都是可以访问的。当然，可以访问并不是可以从对象外面随意读/写对象的属性，那样做将违反信息隐藏原理，也违背由对象主动提供服务而不是被动地接受处理的面向对象设计准则。所谓可以访问是指提供了读/写对象属性的服务。

我们可以用图来形象地描绘程序中的对象（严格地说是对象类），如图 5.1 所示。图中用一个矩形框代表一个对象类，矩形框被两条水平线段分割成三个区域，最上面那个区域中写类名，中部区域内列出该类对象的属性，下部区域内列出该类对象提供的服务。

圆	弧
圆心坐标	圆心坐标
半径	半径
可见性	起始角度
读/写圆心坐标	结束角度
读/写半径	可见性
读/写可见性	读/写圆心坐标
显示	读/写半径
隐藏	读/写起始角度
	读/写结束角度
	读/写可见性
	显示
	隐藏

图 5.1 圆类和弧类

5.1.2 设计类等级

刚才讲过，这个简单的图形程序需要使用圆类和弧类这两类对象，也就是说，我们把该程序中的对象划分成两类。实际上，在设计任何面向对象的程序时，都应该把程序中使用的所有对象都划分成对象类（简称为类，Class），每个对象类都定义了一组数据（即属性）和一组操作（即服

务)。每当建立该对象类的一个新实例时，就按照类中对数据的定义为这个新对象生成一组专用的数据，以便描述该对象独特的属性值。例如，在荧光屏不同位置显示的半径不同的几个圆，虽然都是圆类的对象，但是，各自都有自己专用的数据，以便记录各自的圆心位置和半径等值。

类中定义的服务，是允许施加于该类对象的数据上的操作，是该类所有对象共享的，并不需要为每个对象都复制操作的代码。

除了把对象分类之外，还应该进一步按照子类(或称为派生类)与父类(或称为基类)的关系，把若干个相关的对象类组成一个层次结构的系统(也称为类等级)。在这种层次结构中，下层的派生类自动具有和上层的基类相同的特性(包括数据和操作)，这种现象称为继承。

面向对象程序的许多突出优点来源于继承性。为了利用继承机制减少冗余信息，必须建立适当的类等级。只要不违背领域知识和常识，就应该抽取出相似类的公共属性和公共服务，以建立这些相似类的父类，并在类等级的适当层次中正确地定义各个属性和服务。

从图5.1可以看出，圆和弧的许多属性和服务都是公共的。如果分别定义圆类和弧类，则这些公共的属性和服务需要在每个类中重复定义，这样做势必形成许多冗余信息。反之，如果让圆作为父类，弧作为从圆派生出来的子类，则在圆类中定义了圆心坐标、半径和可见性等属性之后，弧类就可以直接继承这些属性而无需再次重复定义它们，因此，在弧类中仅需定义本类特有的属性(起始角度和结束角度)。类似地，在圆类中定义了读/写圆心坐标、读/写半径和读/写可见性等服务之后，在弧类中只需定义读/写起始角度和读/写结束角度等弧类特有的服务。需要注意的是，虽然在图5.1中圆类和弧类都有名字相同的服务“显示”和“隐藏”，但是它们的具体功能是不同的(显示或隐藏的图形形状不同)。因此，在把弧类作为圆类的子类之后，仍然需要在这两个类中分别定义“显示”和“隐藏”服务。

在我们这个简单例子中，仅涉及圆和弧两类图形，当开发更复杂的图形程序时，将涉及更多的图形种类。但是，无论何种图形都有“坐标”和“可见性”等基本属性。当然，针对不同图形“坐标”的物理含义可能不同，例如，对圆来说指圆心坐标，对矩形来说指某个顶点的坐标。坐标和可见性实质上是荧光屏上一个“点”的属性，如果我们把这两个基本属性抽象出来，放在点(Point)类中定义，并把点类作为各种图形类的公共父类，则可进一步减少冗余信息，并能增加程序的可扩充性。类似地，读/写坐标和读/写可见性等服务也应该放在点类中定义。当然，点类中还需要定义其专用的显示和隐藏服务。

进一步分析“点”的属性，我们发现可以把它们划分为两类基本信息：一类信息描述了点在哪儿(位置)，另一类信息描述了点的状态(可见性)。在上述两类信息中，位置是更基本的信息。因此，我们可以定义一个更基本的基类“位置”，它仅仅含有坐标信息，代表一个几何意义上的点。从位置(Location)类派生出屏幕上的点类，它继承了位置类中定义的每样东西(属性和服务)，并且加进了该类特有的新内容。

综上所述，得到图5.2所示的类等级。为简明起见，图中

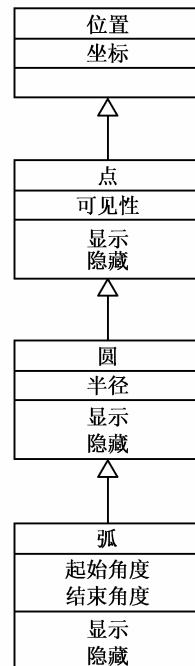


图5.2 简单图形程序的类等级

没有列出读 / 写属性值的常规服务。注意，图中用一端为空心三角形的连线表示继承关系，三角形的顶角紧挨着基类。

5.1.3 定义属性和服务

在上述设计类等级的过程中，已经把需要的属性和服务分配到类等级的适当层次上了。但是，为了能最终实现这个程序，还必须进一步定义属性和服务。

1. 定义属性

首先考虑定义属性的问题，所谓定义属性就是确定每个属性的数据类型和数据结构，同时还要确定每个属性的访问权限。

参见图 5.2，在“位置”类中应该定义属性“坐标”。由于程序处理的是平面上的点，因此坐标由 X 坐标和 Y 坐标组成。我们用屏幕上的像素作为坐标值的单位，这样每个坐标值都是整数。根据上述分析，位置类中包含的属性坐标，由两个简单的整型变量来定义，我们把它们分别命名为 X 和 Y。

“点”类中的属性“可见性”只有两个可能取的值：true（真，即可见）和 false（假，即不可见）。通常把只能取值 true 或 false 的数据类型称为布尔型。我们把可见性属性命名为 Visible。

圆的属性“半径”同样用像素为单位，因此也是整型的简单变量，我们把它命名为 Radius。

弧的属性“起始角度”和“结束角度”都用度为单位，在本例中假设角度只取整数值，因此，这两个属性都用简单的整型变量来表示，我们分别把它们命名为 StartAngle 和 EndAngle。

接下来再考虑每个属性的访问权限（即可访问性）。面向对象程序的一个基本特征就是具有信息隐藏能力，这同时也是面向对象程序的一个突出优点。通常，不允许从对象外面直接访问对象的属性，只能通过对对象向外界公开提供的接口访问对象的属性。

但是，父类的某些属性被子类继承之后，在子类中往往需要频繁地使用这些属性。如果子类使用从父类继承来的属性时也需要通过接口，则会明显降低效率。因此，这些属性的访问权限应该是“在本类及其子类中可以直接访问，超出上述范围则不能直接访问”。在本例中，X 坐标、Y 坐标、可见性（Visible）和半径（Radius）等属性的访问权限就应该是这样的，而起始角度（StartAngle）和结束角度（EndAngle）这两个属性，因为没有子类需要使用它们，访问权限应该是“仅在本类中可以直接访问”。

2. 定义服务

在 5.1.1 小节中曾经讲过，人类习惯解决问题的方法，是采用“顾客—服务员”工作模式。因此，每个对象应该知道怎样完成自己负责提供的服务功能，使用者只需向对象发送消息提出要求即可。但是，对象怎么知道完成服务功能的具体步骤呢？回答是，需要由程序的开发者设计出完成每项服务功能的算法，从而“教会”对象完成其承担任务的方法。定义服务的主要任务就是设计完成每项服务功能的算法。在设计算法时应该使用第 3 章 3.9 节中讲述过的结构程序设计技术。

如前所述，本例每个类中定义的属性都是简单变量。因此，实现常规服务（即，读 / 写属性值）的算法非常简单，仅用一条读 / 写语句即可实现。但是，显示（或隐藏）各类图形的算法就比较繁琐了，幸运的是，各种程序设计语言一般都提供了实现这类常用功能的库函数，我们可以直接调用已有的库函数完成画图功能。因此，实现“显示”服务的算法概括起来就是，把“可见性”属性设置为 true，然后调用相应的库函数用当前的前景颜色画出所要的图形。实

现“隐藏”服务的算法概括地说就是，把“可见性”属性设置为 false，然后调用相应的库函数用当前的背景颜色画出所要的图形。

5.1.4 用 C++ 语言实现

C++ 语言是目前使用得最广泛的面向对象程序设计语言，这一小节将用它实现前面所设计的简单图形程序。本节的目的是使读者对面向对象的 C++ 程序有一个初步的整体概念。本节仅简要地介绍为实现这个简单的图形程序所用到的 C++ 语言的基本语法，读者如果需要系统、全面地学习使用 C++ 语言进行面向对象程序设计的技术，请参阅《面向对象程序设计实用教程》^[3]。

1. 定义类

从 5.1.2 小节可知，为了解决这个简单的图形应用问题，需要定义 4 个类，它们构成一个类的层次系统。为了使计算机知道这个设计结果，必须使用计算机能接受的语言把设计结果表达出来。

用 C++ 语言定义一个类，首先写出关键字 class，编译程序扫描到这个关键字就知道现在要定义一个类。随后应该写出类的名字，在类名后面的一对花括号“{”和“}”之间，说明本类的数据和服务。C++ 语言习惯于把类的数据称为类的数据成员，把类的服务称为类的成员函数。最后，用分号“；”结束一个类的定义。

例如，下面是定义本例中最上层基类“位置”的框架：

```
class Location
{
    // 以下说明数据成员
    //

    // 以下说明成员函数
    //

};
```

其中，从双斜线“//”开始至该行结束所写的内容是注释。注释是给读程序的人提供的解释性信息，它有助于人们理解程序。编译程序在把源程序翻译成目标程序的过程中将略去注释。

上例中 Location（位置）是类名。在一对花括号中用省略号代表暂时尚未写出的数据成员说明和成员函数说明的具体内容。虽然在上面给出的类定义框架中把数据成员说明写在成员函数说明的前面，但是 C++ 语言对说明这两种成员的次序并没有规定。本节随后将讲述用 C++ 语言说明数据成员和成员函数的具体方法。

在定义派生类的时候，必须明确指出它的父类是谁，以便继承父类的成员，C++ 语言的做法是，在派生类类名后面写一个冒号“：“，冒号后写上访问权修饰符，然后再写上父类名。

例如，定义从“位置”类派生出的“点”类的框架如下：

```
class Point : public Location
{
    // 以下说明数据成员
    //

    // 以下说明成员函数
    //

};
```

```

    };

```

访问权修饰符影响从父类继承来的成员（既包括数据成员也包括成员函数）在子类中的可访问性。本例中访问权修饰符为 public（公有派生），它的含义是继承来的成员在子类中的可访问性与在父类中的可访问性相同，但是，仅在父类中可直接访问的私有成员在子类中则不能直接访问。

2. 说明数据成员

用 C++ 语言说明数据成员时，首先用访问权符指定数据的可访问性，接下来用数据类型名指明当前说明的数据的数据类型，然后写出所说明的数据的名字，最后用分号“；”结束对一个数据的说明。

C++ 语言提供了一些预定义的数据类型，每种预定义的数据类型都有一个保留字作为数据类型名。当所要说明的数据成员的数据类型为预定义类型时，可以直接使用相应的保留字作为数据类型名。例如，从设计结果可知，本例中使用的数据成员绝大多数为整型，这是 C++ 语言的预定义类型，相应的数据类型名为 int。

如果需要使用的数据类型不是 C++ 语言预定义的数据类型，则必须在程序中先定义这个类型。例如，本例中“点”类的数据成员“可见性”为布尔型，C++ 语言并没有预先定义这种数据类型，必须由程序员来定义它。如前所述，布尔型数据只有两个可能取的值，它是一种特殊的枚举（enumerated）类型。在定义具体的枚举类型时，需要把允许取的值一一列举出来。因此，我们应该在程序中预先像下面那样定义布尔类型：

```
enum Boolean { false , true };
```

这样定义之后，就可以把 Boolean 作为数据类型名使用了。

综上所述，我们可以把定义 Location 类的框架进一步具体化为：

```

class Location
{
    以下说明数据成员
protected:
    int X ;
    int Y ;
    以下说明成员函数
    :
};


```

定义 Point 类的框架可以进一步具体化为：

```

class Point
{
    以下说明数据成员
protected :
    Boolean Visible ;
    以下说明成员函数
    :
};


```

```
};
```

在上面列出的类定义中，protected（保护的）是访问权符。这个访问权符的含义是，在它下面说明的成员（数据成员或成员函数）仅在本类及其子类中可以直接访问。因此，描述坐标的两个数据成员 X 和 Y，在 Location 类中可直接访问（即，Location 的成员函数可直接读 / 写这两个数据成员），在 Location 类的子类 Point 中也可以直接访问继承来的数据成员 X 和 Y。类似地，数据成员 Visible 既可以在 Point 类中直接访问，也可在 Point 类的子类 Circle（圆）中直接访问。如果要使成员仅在说明它的类中可直接访问，超出该类范围均不能直接访问，则在此成员前面应该写上访问权符 private（私有的）。相反，如果要想使成员可以被任何函数直接访问，则应在该成员前面写上访问权符 public（公有的）。

由于在定义 Point 类时使用的访问权修饰符为 public（Point 类从其父类公有派生），因此，它从父类 Location 继承来的数据成员 X 和 Y，在 Point 类中的可访问性与在 Location 类中的可访问性相同，都是 protected（保护的），也就是说，在 Point 类的子类 Circle 中仍然可以直接访问这两个数据成员。

3. 说明和定义成员函数

（1）说明成员函数

与说明数据成员类似，在说明成员函数时也必须首先用访问权符说明它的可访问性，然后应该给出该成员函数的原型说明。所谓原型说明，由函数返回值类型、函数名及参数特征三部分内容组成。其中参数特征为，在一对圆括号“（”和“）”之间依次列出每个参数的数据类型，相邻参数之间用逗号“，”隔开。

C++ 语言中的函数与数学中的函数有些类似，它对通过参数传递进来的数据（相当于数学函数的自变量）进行某些处理，然后把得到的处理结果作为函数返回值送出来。

例如，Location 类提供的常规服务——读 X 坐标值，由成员函数 GetX 实现，它的原型如下：

```
int GetX();
```

其中，int 为返回值类型。因为 GetX 读出 X 坐标的当前值，如前所述，坐标值为整数，故返回值类型为整型 int。GetX 是函数名，这个函数不需要参数，因此在用来放参数的一对圆括号内为空白（也可以写上保留字 void，它的含义是“空”）。

在 Location 类中对 GetX 的说明如下：

```
public :
```

```
    int GetX();
```

其中，public 是访问权符。因为成员函数 GetX 是 Location 类对外界提供的接口，可以在该类外面直接使用，所以用访问权符 public 来说明它。

（2）定义成员函数

多数成员函数都是在类定义体内部说明，在类定义体外部定义。定义函数就是写出实现函数功能的程序代码，也就是按照 C++ 语言的语法把实现函数功能的算法写出来。

例如，我们可以在 Location 类的定义体之外，像下面那样定义成员函数 GetX：

```
int Location::GetX()
{ return X; }
```

由于在不同类中的成员函数可以使用相同的名字，所以在类定义体外定义成员函数时，必

须在函数名前面冠以类名，类名和函数名之间用作用域分辨符“`:`”来连接，从而构成函数的“全名”，这样才能准确地指明正在定义的是哪个成员函数。

函数原型后面一对花括号内的内容，是实现本函数功能的程序代码。GetX 的算法非常简单，就是送出 X 坐标的当前值。

(3) 定义构造函数

分析对这个简单图形程序的需求可以知道，仅需在程序开始运行时给坐标、半径、起始角度和结束角度赋一次值，通常把给数据设置初始值的操作称为初始化。

为了方便初始化工作，C++ 语言提供了一类特殊的成员函数，称为构造函数。当创建一个新对象时（例如，说明类的一个实例时），系统自动调用构造函数完成初始化工作。

为了使编译程序能够很容易地判断出哪些成员函数是构造函数，C++ 语言有下述规定：

- ? 构造函数的名字必须与类名相同。

- ? 构造函数的原型说明中没有返回值类型，也就是说，在函数名前面为空白。

例如，Location 类的构造函数定义如下：

```
Location Location ( int InitX , int InitY )
{
    X = InitX ;
    Y = InitY ;
}
```

其中，等号“`=`”为 C++ 语言的赋值运算符。

派生类的构造函数不仅要初始化本类中定义的数据成员，还应该初始化从父类继承来的数据成员，因此在创建子类的新对象时，应该同时调用父类的构造函数。C++ 语言的做法是，在定义派生类构造函数时增加一个“初始化表”。

例如，Point 类的构造函数定义如下：

```
Point Point ( int InitX , int InitY ) : Location ( InitX , InitY )
{
    Visible = false ;
}
```

其中，冒号后面的 Location (InitX , InitY) 就是所谓的初始化表。在执行 Point 类构造函数的函数体 `Visible = false` 之前，先调用父类的构造函数 Location (InitX , InitY)，用参数值 InitX 和 InitY 给从父类继承来的数据成员 X 和 Y 赋初值。

4. 说明对象和使用对象

定义了一个类之后就可以说明该类的对象。类实质上是程序员自定义的数据类型，定义了一个类之后，就可以用类名作为数据类型名来说明“数据”，不过这样说明的不是普通变量，而是“对象”，也就是按照类定义建立的该类实例。

如果使用不带参数的构造函数初始化所创建的对象，则说明对象的格式和说明普通变量的格式相同，在类名后面写上对象名即可。

如果使用带参数的构造函数初始化所创建的对象，则在对象名后面应该有一个初值表，系统自动把初值表中列出的值作为实在参数传给被调用的构造函数。例如，

```
Circle c1 ( 100,100,40 );
```

说明了 Circle (圆) 类的一个对象 c1，它的圆心坐标为 (100,100)，半径为 40。

对象与传统的变量有本质区别，它不是被动地等待从外界对它施加操作，相反，它

是数据处理的主体。必须发送消息请求对象主动地执行它的某个操作，处理它的私有数据，从而对外提供指定的服务，而不能从外界直接对它的私有数据进行操作。因此，所谓使用对象，就是向对象发送消息，请求对象主动地执行它的某个操作，从而提供外界所需要的服务。对象响应消息的机制，实际上就是调用对象内由消息名指定的那个公有的成员函数。

向对象发送消息的格式如下：

对象名.成员函数名(实参表);

例如，

c1.Show();

向刚才说明的圆对象 c1 发送消息 Show(), 要求它在荧光屏上显示自己。

5. 完整的 C++ 程序

下面用 C++ 语言编写前面讲述的简单图形程序。为了充分展现面向对象的 C++ 程序本身的特点，我们写出下面这个在 DOS 环境中运行的完整的面向对象的 C++ 图形程序：

```
#include <graphics.h>
#include <conio.h>
enum Boolean{false,true} ;
class Location{
protected :
    int X ;
    int Y ;
public :
    Location ( int InitX , int InitY );
    int GetX ( );
    int GetY ( );
} ;
class Point : public Location{
protected :
    Boolean Visible ;
public :
    Point ( int InitX , int InitY );
    void Show ( );
    void Hide ( );
    Boolean IsVisible ( );
} ;
class Circle : public Point{
protected :
    int Radius ;
public :
    Circle ( int InitX , int InitY , int InitRadius ) ;
```

```
void Show ( );
void Hide ( );
int GetRadius ( );
};

class Arc : public Circle{
private :
    int StartAngle ;
    int EndAngle ;
public :
    Arc ( int InitX , int InitY , int InitRadius , int InitStartAngle , int InitEndAngle );
    void Show ( );
    void Hide ( );
    int GetStartAngle ( );
    int GetEndAngle ( );
};

}
```

下面是成员函数的定义

```
Location Location ( int InitX , int InitY )
{
    X = InitX ;
    Y = InitY ;
}
int Location GetX ( )
{
    return X ;
}
int Location GetY ( )
{
    return Y ;
}
Point Point ( int InitX , int InitY ): Location ( InitX , InitY )
{
    Visible = false ;
}
void Point Show ( )
{
    Visible = true ;
    putpixel ( X , Y , getcolor ( ) );
}
void Point Hide ( )
```

```
{  
    Visible = false ;  
    putpixel ( X,Y,getbkcolor ( ) );  
}  
Boolean Point  IsVisible ( )  
{  
    return Visible ;  
}  
Circle  Circle ( int InitX , int InitY , int InitRadius ): Point ( InitX , InitY )  
{  
    Radius = InitRadius ;  
}  
void Circle  Show ( )  
{  
    Visible = true ;  
    circle ( X,Y,Radius );  
}  
void Circle  Hide ( )  
{  
    int TempColor ;  
    TempColor = getcolor ( );  
    Setcolor ( getbkcolor ( ) );  
    Visible = false ;  
    circle ( X , Y , Radius );  
    setcolor ( TempColor );  
}  
int Circle  GetRadius ( )  
{  
    return Radins ;  
}  
Arc  Arc ( int InitX , int InitY , int InitRadius , int InitStartAngle , int InitEndAngle ):  
    Circle ( InitX , InitY , InitRadius )  
{  
    StartAngle = InitStartAngle ;  
    EndAngle = InitEndAngle ;  
}  
void Arc  Show ( )  
{  
    Visible = true ;
```

```
arc ( X , Y , StartAngle , EndAngle , Radius );
}

void Arc Hide ( )
{
    Visible = false ;
    int TempColor ;
    TempColor = getcolor ( );
    setcolor ( getbkcolor ( ) );
    arc ( X , Y , StartAngle , EndAngle , Radins );
    setcolor ( TempColor );
}

int Arc GetStartAngle ( )
{
    return StartAngle ;
}

int Arc GetEndAngle ( )
{
    return EndAngle ;
}

下面是本程序的主函数
void main ( )
{
    下面这两条语句用于初始化图形系统
    int graphdriver = DETECT , graphmode ;
    initgraph ( &graphdriver , &graphmode , " \bgi " );
    Circle c1 ( 100 , 100 , 40 ), c2 ( 200 , 300 , 20 );
    Arc a1 ( 400 , 150 , 50 , 30 , 120 );
    c1.Show ( );
    c2.Show ( );
    a1.Show ( );
    getch ( ); 等待用户敲任意键
    closegraph ( ); 关闭图形系统
}
```

为帮助读者理解上列程序，下面给出几点说明：

(1) 程序中使用了 C++ 系统预定义的图形库函数和读字符库函数，因此，需要利用包含命令 (# include)，把说明这些库函数的“头文件”graphics.h 和 conio.h 放到程序中。

(2) 成员函数 Point Show 调用库函数 putpixel (X , Y , getcolor ()), 用当前的前景颜色(由库函数 getcolor 送出)，在屏幕上坐标为 (X , Y) 的位置显示一个点。类似地，成员函数 Point Hide 用当前的背景颜色(由库函数 getbkcolor 送出)显示一个点，也就是使得这个点

成为看不见的。

(3) 成员函数 Circle Show 和 Arc Show 的工作原理与 Point Show 类似，不同的是，它们调用的库函数 circle 和 arc 没有颜色参数，固定用当前的前景颜色画图。因此，成员函数 Circle Hide 和 Arc Hide 需要先用局部变量 TempColor 把当前的前景颜色保存起来，然后调用库函数 setcolor 把当前的背景颜色设置为新的前景颜色，画完图（即隐藏了该图形）之后，再取出 TempColor 中保存的颜色值，使之重新成为当前的前景颜色（即恢复原来的前景颜色）。

(4) 如果一个函数不返回数值，则在该函数的原型说明中把返回值类型写为 void，如本程序中 Point，Circle 和 Arc 等类定义中的成员函数 Show 和 Hide 的说明即是如此。

(5) 名字为保留字 main 的函数是 C++ 程序的主函数，运行程序时系统首先执行这个主函数。本程序主函数的工作过程是，初始化图形系统之后说明了 Circle 类的两个实例 c1 和 c2 及 Arc 类的一个实例 a1，系统在创建这些实例的过程中自动调用相应的构造函数，并用 c1，c2 和 a1 后面圆括号内的数作为参数，从而使得这些实例的圆心坐标、半径、起始角度和结束角度分别为需求陈述中所列的值。然后向圆对象 c1 和 c2 及弧对象 a1 发送消息，要求它们在屏幕上显示自己的形状。当你不想继续欣赏这些图形时，敲任意一个键则程序运行结束。

5.2 面向对象的概念

上一节在讲述面向对象程序设计实例的过程中，已经使用了对象、类、属性、服务、消息等面向对象的概念。正确理解面向对象的概念，对学习和掌握面向对象方法学是至关重要的。本节系统、深入地讲述面向对象的概念。

5.2.1 对象

在应用领域中有意义的、与所要解决的问题有关系的任何事物都可以作为对象（Object），它既可以是具体的物理实体的抽象，也可以是人为的概念，或者是任何有明确边界和意义的东西。例如，一名职工、一家公司、一个窗口、一座图书馆、一本图书、贷款、借款……等等，都可以作为一个对象。总之，对象是对问题域中某个实体的抽象，设立某个对象就反映了软件系统具有保存有关它的信息并且与它进行交互的能力。

由于客观世界中的实体通常都既具有静态的属性，又具有动态的行为，因此，面向对象方法学中的对象是由描述该对象属性的数据以及可以对这些数据施加的所有操作封装在一起构成的统一体。对象可以作的操作表示它的动态行为，在面向对象分析和面向对象设计中，通常把对象的操作称为服务或方法。

1. 对象的形象表示

为有助于读者理解对象的概念，图 5.3 形象地描绘了具有三个操作的对象。

看了图 5.3 之后，读者可能会联想到一台录音机。确实，可以用一台录音机比喻一个对象，通俗地说明对象的某些特点。

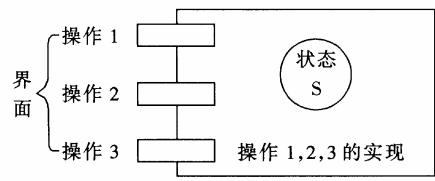


图 5.3 对象的形象表示

当使用一台录音机的时候，总是通过按键来操作：按下“Play（放音）”键，则录音带正向转动，通过喇叭放出录音带中记录的歌曲或其他声音；按下“Record（录音）”键，则录音带正向转动，在录音带中录下新的音响……。完成录音机各种功能的电子线路被装在录音机的外壳中，人们无须了解这些电子线路的工作原理就可以随心所欲地使用录音机。为了使用录音机根本没有必要打开外壳去触动壳内的各种零部件，事实上，不是专业维修人员的一般用户，完全不允许打开录音机外壳。

一个对象很像一台录音机。当在软件中使用一个对象的时候，只能通过对对象与外界的界面来操作它。对象与外界的界面也就是该对象向公众开放的操作，例如，C++语言中对象的公有的（public）成员函数。使用对象向公众开放的操作就好像使用录音机的按键，只需知道该操作的名字（好像录音机的按键名）和所需要的参数（提供附加信息或设置状态，例如听录音前先装录音带并把录音带转到指定位置），根本无须知道实现这些操作的方法。事实上，实现对象操作的代码和数据是隐藏在对象内部的，一个对象好像是一个黑盒子，表示它内部状态的数据和实现各个操作的代码及局部数据，都被封装在这个黑盒子内部，在外面是看不见的，更不能从外面去访问或修改这些数据或代码。

使用对象时只需知道它向外界提供的接口形式而无须知道它的内部实现算法，不仅使得对象的使用变得非常简单、方便，而且具有很高的安全性和可靠性。对象内部的数据只能通过对对象的公有方法（如C++的公有成员函数）来访问或处理，这就保证了对这些数据的访问或处理，在任何时候都是使用统一的方法进行的，不会像使用传统的面向过程的程序设计语言那样，由于每个使用者各自编写自己的处理某个全局数据的过程而发生错误。

此外，录音机中放置的录音带很像一个对象中表示其内部状态的数据，当录音带处于不同位置时按下play键所放出的歌曲是不相同的，同样，当对象处于不同状态时，做同一个操作所得到的效果也是不同的。

2. 对象的定义

目前，对对象所下的定义并不完全统一，人们从不同角度给出对象的不同定义。这些定义虽然形式不同，但基本含义是相同的。下面给出对象的几个定义。

(1) 定义 1

对象是具有相同状态的一组操作的集合。

这个定义主要是从面向对象程序设计的角度看“对象”。

(2) 定义 2

对象是对问题域中某个东西的抽象，这种抽象反映了系统保存有关这个东西的信息或与它交互的能力。也就是说，对象是对属性值和操作的封装。

这个定义着重从信息模拟的角度看待“对象”。

(3) 定义 3

对象 = ID, MS, DS, MI

其中，ID 是对象的标识或名字

MS 是对象中的操作集合

DS 是对象的数据结构

MI 是对象受理的消息名集合（即对外接口）

这个定义是一个形式化的定义。

总之，对象是封装了数据结构及可以施加在这些数据结构上的操作的封装体，这个封装体有可以唯一地标识它的名字，而且向外界提供一组服务（即公有的操作）。对象中的数据表示对象的状态，一个对象的状态只能由该对象的操作来改变。每当需要改变对象的状态时，只能由其他对象向该对象发送消息。对象响应消息时，按照消息模式找出与之匹配的方法，并执行该方法。

从动态角度或对象的实现机制来看，对象是一台自动机。具有内部状态 S ，操作 $f_i (i = 1, 2, \dots, n)$ ，且与操作 f_i 对应的状态转换函数为 $g_i (i = 1, 2, \dots, n)$ 的一个对象，可以用图 5.4 所示的自动机来模拟。

3. 对象的特点

对象有如下一些基本特点。

？以数据为中心。操作围绕对其数据所要做的处理来设置，不设置与这些数据无关的操作，而且操作的结果往往与当时所处的状态（数据的值）有关。

？对象是主动的。它与传统的数据有本质不同，不是被动地等待对它进行处理，相反，它是进行处理的主体。为了完成某个操作，不能从外部直接加工它的私有数据，而是必须通过它的公有接口向对象发消息，请求它执行它的某个操作，处理它的私有数据。

？实现了数据封装。对象好像是一只黑盒子，它的私有数据完全被封装在盒子内部，对外是隐藏的、不可见的，对私有数据的访问或处理只能通过公有的操作进行。

为了使用对象内部的私有数据，只需知道数据的取值范围（值域）和可以对该数据施加的操作（即，对象提供了哪些处理或访问数据的公有方法），根本无须知道数据的具体结构以及实现操作的算法。这也就是抽象数据类型的概念。因此，一个对象类型也可以看作是一种抽象数据类型。

？模块独立性好。对象是面向对象的软件的基本模块，为了充分发挥模块化简化开发工作的优点，希望模块的独立性强。具体来说，也就是要求模块的内聚性强，耦合性弱。如前所述，对象是由数据及可以对这些数据施加的操作所组成的统一体，而且对象是以数据为中心的，操作围绕对其数据所要做的处理来设置，没有无关的操作。因此，对象内部各种元素彼此结合得很紧密，内聚性相当强。由于完成对象功能所需要的元素（数据和方法）基本上都被封装在对象内部，它与外界的联系自然就比较少，因此，对象之间的耦合通常比较松。

？本质上具有并行性。对象是描述其内部状态的数据及可以对这些数据施加的全部操作的集合。不同对象各自独立地处理自身的数据，彼此通过发消息传递信息完成通信。因此，本质上具有并行工作的属性。

5.2.2 其他面向对象的概念

1. 类 (Class)

现实世界中存在的客观事物有些是彼此相似的，例如，张三、李四、王五……虽说每个人职业、性格、爱好、特长等等各有不同，但是，他们的基本特征是相似的，都是黄皮肤、黑头发、黑眼睛，于是人们把他们统称为“中国人”。人类习惯于把有相似特征的事物归为一类，

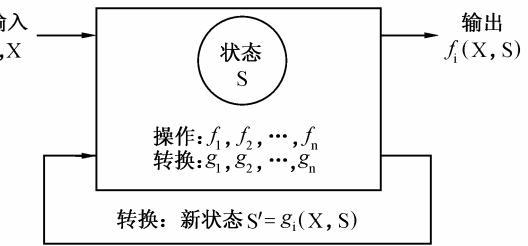


图 5.4 用自动机模拟对象

分类是人类认识客观世界的基本方法。

在面向对象的软件技术中，“类”就是对具有相同数据和相同操作的一组相似对象的定义，也就是说，类是对具有相同属性和行为的一个或多个对象的描述，通常在这种描述中也包括对怎样创建该类的新对象的说明。

例如，5.1节讲述的简单图形程序，在显示器荧光屏的不同位置用不同半径分别画两个圆。这两个圆的圆心坐标和半径都不相同，显然是不同的对象，但是，它们都有相同的数据（圆心坐标、半径、可见性）和相同的操作（显示自己、隐藏自己等等），因此，它们是同一类事物，可以用Circle类来定义。在Circle类中说明和定义的构造函数，说明了在创建该类的新对象时怎样初始化它的数据成员。

2. 实例 (Instance)

实例就是由某个特定的类所描述的一个具体的对象。类是对具有相同属性和行为的一组相似的对象的抽象，类在现实世界中并不能真正存在。在地球上并没有抽象的“中国人”，只有一个个具体的中国人，例如，张三、李四、王五……同样，谁也没见过抽象的“圆”，只有一个个具体的圆。

实际上类是建立对象时使用的“样板”，按照这个样板所建立的一个个具体的对象，就是类的实际例子，通常称为实例。

当使用“对象”这个术语时，既可以指一个具体的对象，也可以泛指一般的对象，但是，当使用“实例”这个术语时，必然是指一个具体的对象。

3. 消息 (Message)

消息，就是要求某个对象执行在定义它的那个类中所定义的某个操作的规格说明。通常，一个消息由下述三部分组成：

- ? 接收消息的对象；
- ? 消息选择符（也称为消息名）；
- ? 零个或多个变元。

例如，在5.1节讲述的面向对象的图形程序中有下列三条向对象发送消息的语句：

```
c1.Show();  
c2.Show();  
a1.Show();
```

其中，

c1, c2, a1 是接收消息的对象的名字；

Show 是消息选择符（即消息名）

这个消息不需要变元，因此圆括号内为空白。当对象接收到消息之后，将执行在Circle类中所定义的Show操作，即提供“显示自己”的服务。

4. 方法 (Method)

方法，就是对象所能执行的操作，也就是类中所定义的服务。方法描述了对象执行操作的算法，响应消息的方法。在C++语言中把方法称为成员函数。

例如，在5.1节讲述的面向对象程序实例中 Location类定义了GetX和GetY等方法，Circle类定义了Show和Hide等方法。

5. 属性 (Attribute)

属性，就是类中所定义的数据，它是对客观世界实体所具有的性质的抽象。类的每个实例都有自己特有的属性值。

在 C++ 语言中把属性称为数据成员。例如，Circle 类中定义的代表圆的半径的数据成员 Radius，就是圆的属性。

6. 继承 (Inheritance)

广义地说，继承是指能够直接获得已有的性质和特征，而不必重复定义它们。在面向对象的软件技术中，继承是子类自动地共享基类中定义的数据和方法的机制。

面向对象软件技术的许多强有力的功能和突出的优点，都来源于把类组成一个层次结构的系统（类等级）：一个类的上层可以有父类，下层可以有子类。这种层次结构系统的一个重要性质是继承性，一个类直接继承其父类的全部描述（数据和操作）。为了更深入、具体地理解继承性的含义，图 5.5 描绘了实现继承机制的原理。

图中以 A、B 两个类为例，其中 B 类是从 A 类派生出来的子类，它除了具有自己定义的特性（数据和操作）之外，还从父类 A 继承特性。当创建 A 类的实例 a1 的时候，a1 以 A 类为样板建立实例变量（在内存中分配所需要的空间），但是它并不从 A 类中拷贝所定义的方法。

当创建 B 类的实例 b1 的时候，b1 既要以 B 类为样板建立实例变量，又要以 A 类为样板建立实例变量，b1 所能执行的操作既有 B 类中定义的方法，又有 A 类中定义的方法，这就是继承。当然，如果 B 类中又定义了和 A 类中同名的数据或操作，则 b1 仅使用 B 类中定义的这个数据或操作，除非采用特别措施，否则 A 类中与之同名的数据或操作在 b1 中就不能使用。

在 5.1 节讲述的面向对象的图形程序中，Location 类是最上层的基类，Point 类是 Location 类的子类。如果说说明了 Point 类的一个对象 p1，则它不仅要以 Point 类为样板建立实例变量 p1.Visible，还要以 Location 类为样板建立实例变量 p1.X 和 p1.Y，p1 所能执行的操作既有 Point 类中定义的方法（如 Show，IsVisible 等方法），也有 Location 类中定义的方法 GetX 和 GetY。

继承具有传递性，如果类 C 继承类 B，类 B 继承类 A，则类 C 继承类 A。因此，一个类实际上继承了它所在的类等级中在它上层的全部基类的所有描述，也就是说，属于某类的对象除了具有该类所描述的性质外，还具有类等级中该类上层全部基类描述的一切性质。

例如，在 5.1 节讲述的程序实例中，Circle 类是从 Point 类派生出来的，它不仅继承了 Point 类中定义的数据和操作，还继承了 Point 类的父类 Location 中定义的数据和操作。因此，Circle 类的对象 c1 不仅拥有数据 Radius，而且还拥有数据 Visible，X 和 Y；c1 所能执行的操作（即可以向 c1 发送的消息）既有在 Circle 类中定义的方法 Show，Hide 和 GetRadius，又有在 Point 类中定义的方法 IsVisible，还有在 Location 类中定义的方法 GetX 和 GetY。需要注意的是，由于在 Circle 类中定义了与它父类 Point 类中同名的方法 Show 和 Hide，因此，如果不采取特别措施，Circle 类的对象 c1 只能使用本类定义的方法 Show 和 Hide，而不能使用 Point 类定义的 Show 和 Hide。

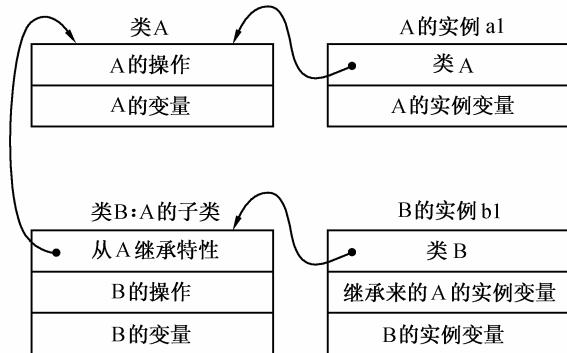


图 5.5 实现继承机制的原理

当一个类只允许有一个父类时，也就是说，当类等级为树形结构时，类的继承是单继承；当允许一个类有多个父类时，类的继承是多重继承。多重继承的类可以组合多个父类的性质构成所需要的性质，因此，功能更强，使用更方便；但是，使用多重继承时要注意避免二义性。

继承性使得相似的对象可以共享程序代码和数据结构，从而大大减少了程序中的冗余信息。在程序执行期间，对对象某一性质的查找是从该对象类在类等级中所在的层次开始，沿类等级逐层向上进行的，并把第一个被找到的性质作为所要的性质。因此，低层的性质将屏蔽高层的同名性质。

使用从原有类派生出新的子类的办法，使得对软件的修改变得比过去容易得多了。当需要扩充原有的功能时，派生类的方法可以调用其基类的方法，并在此基础上增加必要的程序代码；当需要完全改变原有操作的算法时，可以在派生类中实现一个与基类方法同名而算法不同的方法；当需要增加新的功能时，可以在派生类中实现一个新的方法。

继承性使得用户在开发新的应用系统时不必完全从零开始，可以继承原有的相似系统的功能或者从类库中选取需要的类，再派生出新的类以实现所需要的功能。

利用继承性还可以用把已有的一般性的解加以具体化的办法，来达到软件重用的目的：首先，使用抽象的类开发出一般性问题的解，然后，在派生类中增加少量代码使一般性的解具体化，从而开发出符合特定应用需要的具体解。

7. 封装（Encapsulation）

从字面上理解，所谓封装就是把某个事物包起来，使外界不知道该事物的具体内容。

在面向对象的程序中，我们把数据和实现操作的代码集中起来放在对象内部。一个对象好像是一个不透明的黑盒子，表示对象状态的数据和实现各个操作的代码与局部数据，都被封装在黑盒子里面，从外面是看不见的，更不能从外面直接访问或修改这些数据及代码。

使用一个对象的时候，只需知道它向外界提供的接口形式而无须知道它的数据结构细节和实现操作的算法。

综上所述，具有封装性的条件如下：

? 有一个清晰的边界。所有私有数据和实现操作的代码都被封装在这个边界内，从外面看不见更不能直接访问。

? 有确定的接口（即协议）。这些接口就是对象可以接受的消息，只能通过向对象发送消息来使用它。

? 受保护的内部实现。实现对象功能的细节（私有数据和代码）不能在定义该对象的类的范围外进行访问。

封装性也就是信息隐藏，通过封装把对象的实现细节对外界隐藏起来了。

对象类实质上是抽象数据类型。类把数据说明和操作说明与数据表达和操作实现分开了，使用者仅需知道它的说明（值域及可对数据施加的操作）就可以使用它。

8. 多态（Polymorphism）

从字面上理解，所谓多态就是同一个东西有许多不同的形态。在日常生活中，我们有时用同一个动词表示不同的动作。例如，我们可以说：“请挪开那块石头。”也可以说：“请挪开那辆自行车。”还可以说：“请挪开那辆汽车。”在上述三句话中，虽然使用了同一个动词“挪开”，但是它作用在不同种类的对象上就有不同的含义：“挪开石头”指的是人搬走石头，“挪开自行车”的含义是人推走自行车，而“挪开汽车”的含义则是人开走汽车。

在面向对象的软件技术中，多态性是指子类对象可以像父类对象那样使用，同样的消息既可以发送给父类对象也可以发送给子类对象。也就是说，在类等级的不同层次中可以共享（公用）一个行为（方法）的名字，然而不同层次中的每个类却各自按自己的需要来实现这个行为。当对象接收到发送给它的消息时，根据该对象所属于的类动态选用在该类中定义的实现算法。

在 C++ 语言中，多态性是通过虚函数来实现的。在类等级不同层次中可以说明名字、参数特征和返回值类型都相同的虚拟成员函数，而不同层次的类中的虚函数实现算法各不相同。虚函数机制使得程序员能在同一个类等级中使用相同函数的多个不同版本，在运行时刻才根据接收消息的对象所属于的类，决定到底执行哪个特定的版本，这称为动态联编，也叫滞后联编。

多态性机制不仅增加了面向对象软件系统的灵活性，进一步减少了信息冗余，而且显著提高了软件的可重用性和可扩充性。当扩充系统功能增加新的实体类型时，只须派生出与新实体类相应的新的子类，并在新派生出的子类中定义符合该类需要的虚函数，完全无须修改原有的程序代码，甚至不需要重新编译原有的程序（仅需编译新派生类的源程序，再与原有程序的.OBJ 文件连接）。

9. 重载（Overloading）

有两种重载：函数重载是指在同一作用域内的若干个参数特征不同的函数可以使用相同的函数名字；运算符重载是指同一个运算符可以施加于不同类型的操作数上面。当然，当参数特征不同或被操作数的类型不同时，实现函数的算法或运算符的语义是不相同的。

在 C++ 语言中函数重载是通过静态联编（也叫先前联编）实现的，也就是在编译时根据函数变元的个数和类型，决定到底使用函数的哪个实现代码；对于重载的运算符，同样是在编译时根据被操作数的类型，决定使用该算符的哪种语义。

重载进一步提高了面向对象系统的灵活性和可读性。

5.3 面向对象方法学概述

5.3.1 面向对象方法学的要点

面向对象方法学的出发点和基本原则，是尽可能模拟人类习惯的思维方式，使开发软件的方法与过程尽可能接近人类认识世界解决问题的方法与过程，也就是使描述问题的问题空间（也称为问题域）与实现解法的解空间（也称为求解域）在结构上尽可能一致。

客观世界的问题都是由客观世界中的实体及实体相互间的关系构成的。我们把客观世界中的实体抽象为问题域中的对象（Object）。因为所要解决的问题具有特殊性，因此，对象是不固定的。一个雇员可以作为一个对象，一家雇用了许多雇员的公司也可以作为一个对象，到底应该把什么抽象为对象，由所要解决的问题决定。

从本质上说，我们用计算机解决客观世界的问题，是借助于某种程序设计语言的规定，对计算机中的实体施加某种处理，并用处理结果去映射解。我们把计算机中的实体称为解空间对象。显然，解空间对象取决于所使用的程序设计语言。例如，汇编语言提供的对象是存储单元；面向过程的高级语言提供的对象，是各种预定类型的变量、数组、记录和文件等等。一旦提供了某种解空间对象，就隐含规定了允许对该类对象施加的操作。

从动态观点看，对对象施加的操作就是该对象的行为。在问题空间中，对象的行为是极其丰富多彩的，然而解空间中的对象的行为却是非常简单呆板的。因此，只有借助于十分复杂的算法，才能操纵解空间对象从而得到解。这就是人们常说的“语义断层”，也是长期以来程序设计始终是一门学问的原因。

通常，客观世界中的实体既具有静态的属性又具有动态的行为。然而传统语言提供的解空间对象实质上却仅是描述实体属性的数据，必须在程序中从外部对它施加操作，才能模拟它的行为。

众所周知，软件系统本质上是信息处理系统。数据和处理原本是密切相关的，把数据和处理人为地分离成两个独立的部分，会增加软件开发的难度。与传统方法相反，面向对象方法是一种以数据或信息为主线，把数据和处理相结合的方法。面向对象方法把对象作为由数据及可以施加在这些数据上的操作所构成的统一体。对象与传统的数据有本质区别，它不是被动地等待外界对它施加操作，相反，它是进行处理的主体。必须发消息请求对象主动地执行它的某些操作，处理它的私有数据，而不能从外界直接对它的私有数据进行操作。

面向对象方法所提供的“对象”概念，是让软件开发者自己定义或选取解空间对象，然后把软件系统作为一系列离散的解空间对象的集合。应该使这些解空间对象与问题空间对象尽可能一致。这些解空间对象彼此间通过发送消息而相互作用，从而得出问题的解。也就是说，面向对象方法是一种新的思维方法，它不是把程序看作是工作在数据上的一系列过程或函数的集合，而是把程序看作是相互协作而又彼此独立的对象的集合。每个对象就像一个微型程序，有自己的数据、操作、功能和目的。这样做就向着减少语义断层的方向迈了一大步，在许多系统中解空间对象都可以直接模拟问题空间的对象，解空间与问题空间的结构十分一致，因此，这样的程序易于理解和维护。

概括地说，面向对象方法具有下述四个要点：

(1) 认为客观世界是由各种对象组成的，任何事物都是对象，复杂的对象可以由比较简单的对象以某种方式组合而成。按照这种观点，可以认为整个世界就是一个最复杂的对象。因此，面向对象的软件系统是由对象组成的，软件中的任何元素都是对象，复杂的软件对象由比较简单的对象组合而成。

(2) 把所有对象都划分成各种对象类(简称为类，Class)，每个对象类都定义了一组数据和一组方法。数据用于表示对象的静态属性，是对象的状态信息。因此，每当建立该对象类的一个新实例时，就按照类中对数据的定义为这个新对象生成一组专用的数据，以便描述该对象独特的属性值。例如，荧光屏上不同位置显示的半径不同的几个圆，虽然都是 Circle 类的对象，但是，各自都有自己专用的数据，以便记录各自的圆心位置、半径等等。

类中定义的方法，是允许施加于该类对象上的操作，是该类所有对象共享的，并不需要为每个对象都复制操作的代码。

(3) 按照子类(或称为派生类)与父类(或称为基类)的关系，把若干个对象类组成一个层次结构的系统(也称为类等级)。在这种层次结构中，通常下层的派生类具有和上层的基类相同的特性(包括数据和方法)，这种现象称为继承(Inheritance)。但是，如果在派生类中对某些特性又做了重新描述，则在派生类中的这些特性将以新描述为准，也就是说，低层的特性将屏蔽高层的同名特性。

(4) 对象彼此之间仅能通过传递消息互相联系。

对象与传统的数据有本质区别，它不是被动地等待外界对它施加操作，相反，它是进行处理的主体，必须发消息请求它执行它的某个操作，处理它的私有数据，而不能从外界直接对它的私有数据进行操作。也就是说，一切局部于该对象的私有信息，都被封装在该对象类的定义中，就好像装在一个不透明的黑盒子中一样，在外界是看不见的，更不能直接使用，这就是“封装性”。

综上所述，面向对象的方法学可以用下列方程来概括：

面向对象=对象 + 类 + 继承 + 用消息通信

也就是说，面向对象方法就是，既使用对象又使用类和继承等机制，而且对象彼此之间只能通过传递消息互相联系的方法。

5.3.2 面向对象建模

众所周知，在解决问题之前必须首先理解所要解决的问题。对问题理解得越透彻，就越容易解决它。当我们完全、彻底地理解了一个问题的时候，通常就已经解决了这个问题。

为了更好地理解问题，人们常常采用建立问题模型的方法。所谓模型，就是为了理解事物而对事物作出的一种抽象，是对事物的一种无歧义的书面描述。通常，模型由一组图示符号和组织这些符号的规则组成，利用它们来定义和描述问题域中的术语和概念。更进一步讲，模型是一种思考工具，利用这种工具可以把知识规范地表示出来。

模型可以帮助我们思考问题、定义术语、在选择术语时作出适当的假设，并且可以帮助我们保持定义和假设的一致性。

为了开发复杂的软件系统，系统分析员应该从不同角度抽象出目标系统的特性，使用精确的表示方法构造系统的模型，验证模型是否满足用户对目标系统的需求，并在设计过程中逐渐把和实现有关的细节加进模型中，直至最终用程序实现模型。对于那些因过分复杂而不能直接理解的系统，特别需要建立模型，建模的主要目的是为了减少复杂性。人的头脑每次只能处理一定数量的信息，模型通过把系统的重要部分分解成人的头脑一次能处理的若干个子部分，从而减少系统的复杂程度。

在对目标系统进行分析的初始阶段，面对大量模糊的、涉及众多专业领域的、错综复杂的信息，系统分析员往往感到无从下手。模型提供了组织大量信息的一种有效机制。

一旦建立起模型之后，这个模型就要经受用户和各个领域专家的严格审查。由于模型的规范化和系统化，因此比较容易暴露出系统分析员对目标系统认识的片面性和不一致性。通过审查，往往会出现许多错误，发现错误是正常现象，这些错误可以在成为目标系统中的错误之前，就被预先清除掉。

通常，通过快速建立原型，让用户和领域专家经过亲身体验，对系统模型进行更有效的审查。模型常常会经过多次必要的修改，通过不断改正错误的或不全面的认识，最终，软件开发人员对问题有了透彻的理解，从而为后续的开发工作奠定了坚实基础。

用面向对象方法成功地开发软件的关键，同样是对问题域的理解。面向对象方法最基本的原则，是按照人们习惯的思维方式，用面向对象观点建立问题域的模型，开发出尽可能自然地表现求解方法的软件。

用面向对象方法开发软件，通常需要建立三种形式的模型，它们分别是描述系统数据结构的对象模型，描述系统控制结构的动态模型和描述系统功能的功能模型。这三种模型都涉及到数据、控制和操作等共同的概念，只不过每种模型描述的侧重点不同。这三种模型从三个不同

但又密切相关的角度模拟目标系统，它们各自从不同侧面反映了系统的实质性内容，综合起来则全面地反映了对目标系统的需求。一个典型的软件系统组合了上述三方面内容：它使用数据结构（对象模型），执行操作（动态模型），并且完成数据值的变化（功能模型）。

为了全面地理解问题域，对任何大系统来说，上述三种模型都是必不可少的。当然，在不同的应用问题中，这三种模型的相对重要程度会有所不同，但是，用面向对象方法开发软件，在任何情况下，对象模型始终都是最重要、最基本、最核心的。在整个开发过程中，三种模型一直都在发展、完善。在面向对象分析过程中，构造出完全独立于实现的应用域模型；在面向对象设计过程中，把求解域的结构逐渐加入到模型中；在实现阶段，把求解域的模型编成程序代码并进行严格的测试验证。

本书第2章已经讲述过功能模型，本章下面两节将讲述对象模型和动态模型。

5.3.3 面向对象的软件过程

在开发本章5.1节所述的面向对象的图形程序的过程中，我们首先陈述了对这个程序的需求，然后用面向对象观点分析需求，确定了问题域中的对象并把对象划分成类，接下来设计出了适当的类等级以及每个类的属性和服务，最后用C++语言编写出这个面向对象的图形程序。

事实上，不论采用什么方法学开发软件，都必须完成一系列性质各异的工作。这些必须完成的工作要素是：确定“做什么”，确定“怎样做”，“实现”和“完善”。使用不同的方法学开发软件的时候，完成这些工作要素的顺序、工作要素的名称和相对重要性有可能也不相同，但是却不能忽略其中任何一个工作要素。

一般说来，使用面向对象方法学开发软件时，工作重点应该放在生命周期中的分析阶段。这种方法在开发的早期阶段定义了一系列面向问题的对象，并且在整个开发过程中不断充实和扩充这些对象。由于在整个开发过程中都使用统一的软件概念“对象”，所有其他概念（例如功能、关系、事件等）都是围绕对象组成的，目的是保证分析工作中得到的信息不会丢失或改变，因此，对生命周期各阶段的区分自然就不重要、不明显了。分析阶段得到的对象模型也适用于设计阶段和实现阶段。由于各阶段都使用统一的概念和表示符号，因此，整个开发过程都是吻合一致的，或者说是“无缝”连接的，这自然就很容易实现各个开发步骤的多次反复迭代，达到认识的逐步深化。每次反复都会增加或明确一些目标系统的性质，但却不是对先前工作结果的本质性改动，这样就减少了不一致性，降低了出错的可能性。

迭代是软件开发过程中普遍存在的一种内在属性。经验表明，软件过程各个阶段之间的迭代或一个阶段内各个工作步骤之间的迭代，在面向对象范型中比在结构化范型中更常见，也更容易实现。

图5.6所示的喷泉模型是典型的面向对象生命周期

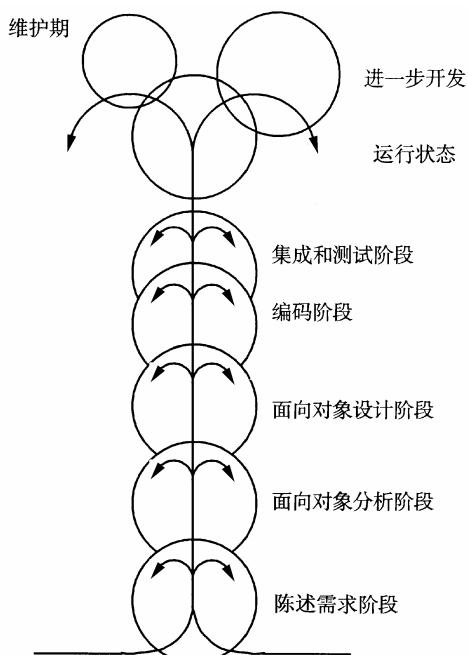


图5.6 喷泉模型

模型。

“喷泉”这个词形象地表明了面向对象软件开发过程迭代和无缝的特性。图中代表生命周期不同阶段的圆圈相互重叠，这明确地表示两个阶段的活动之间存在交迭；而面向对象方法学在概念和表示方法上的一致性，保证了在各项开发活动之间的无缝过渡。事实上，用面向对象方法学开发软件时，在分析、设计和编码等项开发活动之间并不存在明显的边界。图中在一个圆圈内的向下箭头代表该阶段内的迭代或求精。图中用较小的圆圈代表维护，这形象地表示采用了面向对象方法学之后维护时间缩短了。

为避免使用喷泉模型开发软件时开发过程过分无序，应该把一个线性过程（例如，快速原型模型或图 5.6 的中心垂线）作为总目标。但是，同时也应该记住，面向对象范型本身要求经常对软件开发活动进行迭代或求精。

5.4 对象模型

对象模型表示结构化的系统的静态的“数据”性质。它是对模拟客观世界实体的对象以及对象彼此间的关系的映射，描述了软件系统的静态结构。正如本章前面各节反复讲述的，面向对象方法学强调围绕对象而不是围绕功能来构造系统。因此，对象模型为建立动态模型和功能模型，提供了实质性的框架。

在建立对象模型时，我们的目标是从客观世界中提炼出对具体应用有价值的概念。

为了建立对象模型，需要定义一组图形符号，并且规定一组组织这些符号以表示特定语义的规则。对面向对象方法学的建立和发展作出过重大贡献的著名专家，如 Grady Booch，Peter Coad & Ed Yourdon，James Rumbaugh，Sally Shlaer 等人，都提出了自己的表示方法。这些表示方法虽然形式各异，但是，它们有一个共同的特点，那就是独立于具体的开发模式。

与结构化分析和结构化设计方法不同，用面向对象方法开发软件时，没有必要在不同发展阶段使用不同的表示方法。现有的用于建立对象模型的表示方法，都包含下列符号：

- ? 表示类的符号（应该既能表示属性又能表示服务）；
- ? 表示对象（类实例）的符号；
- ? 表示继承关系的符号；
- ? 表示类和（或）对象间其他关系的符号。

但是，面对众多的建模语言，用户很难找到一种最适合其应用特点的语言，而且不同建模语言之间存在的细微差别也极大地妨碍了用户之间的交流。为了促进面向对象方法学的发展和应用，应该组织联合设计小组，集中各种面向对象建模语言的优点，克服缺点，把面向对象建模语言统一起来。

经过两年的辛勤工作，Booch，Rumbaugh 和 Jacobson 于 1996 年共同提出了统一建模语言 UML。1997 年 11 月 OMG 组织采纳 UML，作为面向对象建模的标准语言，目前 UML 已经成为可视化建模语言事实上的工业标准。

本节介绍 UML 提供的适用于建立对象模型的类图，下节介绍适用于建立动态模型的状态图。

类图不仅定义软件系统中的类，描述类与类之间的关系，它还表示类的内部结构（类的属

性和操作）。类图描述的是一种静态关系，它是从静态角度表示系统的，因此，类图建立的是一种静态模型，它在系统的整个生命周期内都是有效的。类图是构建其他图的基础，没有类图就没有状态图等其他图，也就无法表示系统其他方面的特性。下面介绍类图的主要表示符号。

5.4.1 表示类的图形符号

1. 定义类

表示类的图形符号为一个长方形，它分成上、中、下三个区域，上面区域中写类名，中部区域为类的属性，下部区域为类的操作（即服务）。本章 5.1 节已经使用了表示类的图形符号，如图 5.1 所示。

当不需要详细描述一个类内定义了哪些属性和操作时，可以仅在一个矩形框内写上类名代表该类，也就是说，代表类的长方形的下面两个区域可以省略（也可只省略下部区域）。

2. 命名

类名是一类对象的名字。为类命名时应该尽量使用应用领域中的标准术语，它的含义应该明确、无歧义、以利于开发人员与用户之间的相互理解和交流。通常，用名词作为类的名字。

具体说来，为类命名时应该遵守以下几条准则：

(1) 使用标准术语

应该使用在应用领域中人们习惯的标准术语作为类名，不要随意创造名字。例如，“交通信号灯”比“信号单元”这个名字好，“传送带”比“零件传送设备”好。

(2) 使用具有确切含义的名词

尽量使用能表示类的含义的日常用语作名字，不要使用空洞的或含义模糊的词作名字。例如，“库房”比“房屋”或“存物场所”更确切。

(3) 必要时用名词短语作名字

为使名字的含义更准确，必要时用形容词加名词或其他形式的名词短语作名字。例如，“最小的领土单元”、“储藏室”、“公司员工”等都是比较恰当的名字。

总之，名字应该是富于描述性的、简洁的而且无二义性的。

3. 描述类的属性

(1) 选取属性

类的属性描述该类对象的共同特征，放在表示类的长方形的中部区域（可省略）。选取属性时应该遵守下述原则：

- ? 类的属性应该能够描述并区分该类的每个对象；
- ? 只有系统需要使用的那些特征才抽取出来作为类的属性；
- ? 选取属性时应该充分考虑系统建模的目的。

(2) 描述属性

UML 描述属性的语法格式如下：

可见性 属性名：类型名=初值 {性质串}

其中，属性名和类型名必须有，其他部分根据需要可有可无。

属性的可见性（即可访问性）通常分为公有的、私有的和保护的三种，分别用加号（+）、减号（-）和井号（#）表示。如果在属性名前面没有任何符号，则表示该属性的可见性尚未

定义。注意，UML 没有缺省的可见性。

属性名和类型名之间用冒号（:）分隔。类型名表示该属性的数据类型，它既可以是基本数据类型，如整数、实数等，也可以是用户自定义的类型。一般说来，可以使用的类型由所涉及的程序设计语言决定。

属性的缺省值用初值表示，类型名和初值之间用等号隔开。

最后是用花括号括起来的性质串，列出该属性所有可能的取值。枚举类型的属性经常使用性质串，串内每个枚举值之间用逗号分隔。当然，也可以用性质串说明该属性的其他信息，例如，约束说明 {只读} 表明该属性是只读属性。

例如，本章 5.1 节所述例子中的 Point 类在 UML 类图中像图 5.7 那样描述。

4. 描述类的操作

(1) 选取操作

操作用于检索、修改类的属性或执行某些动作。操作也称为功能，但是只能作用于该类的对象上。在类图中操作放在代表类的长方形的下部区域内（可省略），参见图 5.7。选取操作时应该遵守下述准则：

- ? 操作围绕对类的属性数据所需要做的处理来设置，不设置与这些数据无关的操作；
- ? 只有系统需要使用的那些操作才抽取出来作为类的操作；
- ? 选取操作时应该充分考虑用户的需求。

(2) 描述操作

UML 描述操作的语法格式为：

可见性 操作名(参数表): 返回值类型 {性质串}

其中，可见性和操作名是不可缺少的。

操作的可见性通常分为公有的（用加号表示）和私有的（用减号表示）两种，其含义与属性可见性的含义相同。

参数表由若干个彼此间用逗号（,）隔开的参数构成。书写参数的语法格式如下：

参数名：参数类型名=缺省值

5.4.2 表示关系的图形符号

如前所述，类图由类及类与类之间的关系构成。类与类之间的关系（或一类对象与另一类对象之间的关系），是对客观世界实体相互间关系的抽象。

定义了类之后，就可以定义类之间的各种关系了。类与类之间通常有关联、泛化（继承）、依赖和细化等四种关系。

1. 关联关系

关联关系表示两类对象之间存在着某种语义上的联系，也就是对象之间有相互作用、相互依靠的关系。

通常把两类对象之间的关联关系再细分为一对一(1 1)、一对多(1 M)和多对多(M N)等三种基本类型，类型的划分依据参与关联的对象的数目。例如，一个国家有一个合法政府，一个政府仅属于一个国家，国家与政府是一对一的关系；一位教师有许多本书，一本书只

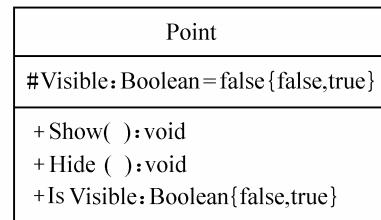


图 5.7 用 UML 类图描述 Point 类

属于一位教师，教师与书是一对多的关系；一位作家写了许多本书，一本书可能由几名作者合写，作家与书是多对多的关系。

(1) 表示符号

常见的关联都是双向的关系，其图示符号是连接两个类的一条直线。例如，作家使用计算机，作家与计算机之间的关联关系如图 5.8 所示。



图 5.8 作家与计算机之间的关联关系

图 5.8 所示的关联是双向的，可在一个方向上为关联起一个名字，在另一个方向上为关联起另一个名字（也可不起名字）。为避免混淆，在名字前面（或后面）加一个表示关联方向的黑三角。

(2) 重数

在类图中还可以表示关联中的数量关系，即参与关联的对象的个数。在 UML 中用重数说明数量或数量范围，例如，

- $0 \dots 1$ 表示 0 至 1 个对象
- $0 \dots *$ 或 $*$ 表示 0 至多个对象
- $1 \dots *$ 表示 1 至多个对象
- $1 \dots 15$ 表示 1 至 15 个对象
- 8 表示 8 个对象

如果图中没有明确地标出关联的重数，则缺省的重数是 1。

例如，图 5.8 表示，1 位作家可以使用 1 到多台计算机，一台计算机可被 0 到多个作家使用。

(3) 角色

在任何关联中都会涉及参与此关联的对象所扮演的角色（即所起的作用）问题，在某些情况下显式标明角色有助于别人理解类图。例如，图 5.9 是一个递归关联（即一类对象与该类对象本身有关系）的例子，一个人与另一个人结婚，必然一个人扮演丈夫的角色，另一个人扮演妻子的角色。

如果在类图中没有标出角色名，则意味着用类名作为角色名。

(4) 限定

一个受限的关联由两类对象及一个限定词组成。利用限定词通常能有效地减少关联的重数。

通常，限定关联用于一对多或多对多的关联关系中，它可以把模型中的重数从一对多变成一对一。在类图中把限定词放在表示关联关系的直线末端的一个小方框内。

例如，某操作系统中一个目录下有许多文件，一个文件仅属于一个目录，在一个目录内文件名确定了唯一一个文件。图 5.10 利用限定词“文件名”表示了目录与文件之间的关系，可见，利用限定词把一对多关系简化成了一对一关系。

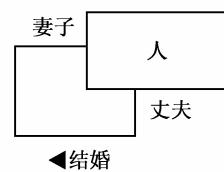


图 5.9 关联的角色

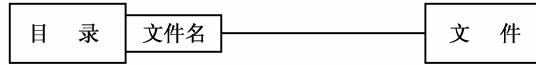


图 5.10 一个受限的关联

限定提高了语义精确性，增强了查询能力。在图 5.10 中，限定的语法表明，文件名在其目录内是唯一的。因此，查找一个文件的方法就是，首先定下目录，然后在该目录内查找指定的文件名。由于目录加文件名可唯一地确定一个文件，因此，限定词“文件名”应该放在靠近目录对象的那一端。

(5) 关联类

为了说明关联的性质，可能需要提供一些附加信息。可以引入一个关联类来记录这些附加信息。关联中的每个连接与关联类的一个对象相联系。在类图中关联类通过一条虚线与关联连接。通常把关联类的属性称为链属性。

例如，图 5.11 是一个电梯系统的类图，其中“队列”就是电梯控制器类与电梯类的关联关系上的关联类。从图中可以看出，一个电梯控制器控制着 4 台电梯，这样，控制器和电梯之间的实际连接就有 4 个，每个连接都对应着一个队列对象，每个队列对象都存储着来自控制器和该电梯内部按钮的请求服务信息。电梯控制器通过读取队列信息，选择一台合适的电梯为乘客服务。关联类与一般的类一样，也有属性、操作和关联。

(6) 聚集

聚集也称为聚合，是关联的特例。聚集表示一类对象与另一类对象之间的关系，是整体与部分的关系。在陈述需求时使用的“包含”、“组成”、“分为……部分”等字句，往往意味着存在聚集关系。除了一般的聚集关系之外，还有两种特殊的聚集关系，分别称为共享聚集和复合聚集，如下所述。

共享聚集

如果在聚集关系中处于部分方的对象可以同时参与多个处于整体方对象的构成，则该聚集称为共享聚集。

例如，一个课题组包含许多成员，每个成员又可以同时是另一个课题组的成员，则课题组和成员之间是共享聚集关系，如图 5.12 所示。一般聚集和共享聚集的图示符号，都是在表示关联关系的直线末端紧挨着整体类的地方画一个空心菱形。

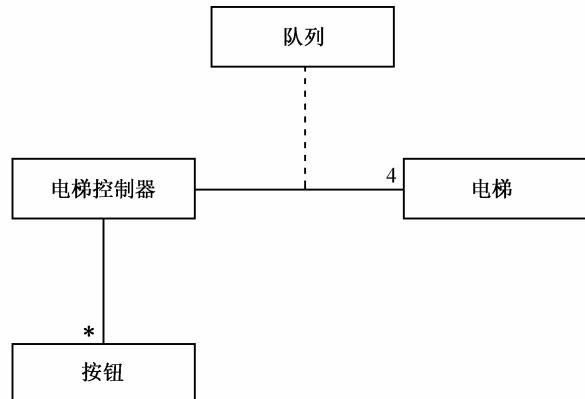


图 5.11 关联类示例



图 5.12 共享聚集示例

复合聚集

如果部分类对象完全隶属于整体类对象，部分与整体共存，整体不存在了部分也会随之消失（或失去存在价值了），则该聚集称为复合聚集（简称为组成）。

例如，我们在屏幕上打开一个窗口。它就由文本框、列表框、按钮和菜单组成，一旦关闭了窗口，各个组成部分也同时消失了，窗口

和它的组成部分之间存在着复合聚集关系。

图 5.13 描述了窗口的组成，从图上可以看出，组成关系用实心菱形表示。

2. 泛化关系

UML 中所说的泛化关系，就是通常所说的继承关系，它是通用元素和具体元素之间的一种分类关系，表明在通用元素（类）和具体元素（类）之间存在一般与特殊的关系。具体元素拥有通用元素的全部信息，并且还可以附加一些其他信息。UML 对定义泛化关系有下述三条要求：

？具体元素应与通用元素完全一致，通用元素具有的属性、操作和关联，具体元素也都隐含地具有。

？具体元素还应包含通用元素所没有的额外信息。

？允许使用通用元素实例的地方，也应能够使用具体元素的实例。

在 UML 的类图中，用一端为空心三角形的连线表示泛化关系，三角形的顶角紧挨着通用元素，如图 5.2 所示。

注意，泛化关系针对类型而不针对类的实例，一个类可以继承另一个类（即两个类之间存在泛化关系），但是，一个对象不能继承另一个对象。

泛化可以进一步划分成普通泛化和受限泛化，如下所述。

(1) 普通泛化

普遍泛化的概念与本章 5.2.2 小节中讲过的继承概念基本相同，此处不再赘述。

需要特别说明的是，没有具体对象的类称为抽象类。抽象类通常作为父类，用于描述其他类（子类）的公共属性和行为。图示抽象类时，在类名下方附加一个标记值 { abstract }，如图 5.14 所示。图下方的两个折角矩形是 UML 模型元素“笔记”的符号，其中的文字是注释，分别说明两个子类的操作 drive 的功能。

抽象类通常都具有抽象操作。抽象操作仅用来指定该类的所有子类都应具有哪些行为。抽象操作的图示方法与抽象类相似，在操作标记后面跟随一个性质串 { abstract }。

与抽象类相反的类是具体类，具体类有自己的对象（即实例），并且该类的操作都有具体的实现方法。

图 5.15 给出一个比较复杂的类图的例子，这个例子综合应用了前面讲过的许多概念和图示符号。图 5.15 表明，一幅工程蓝图由许多图形组成，图形可以是直线、圆、多边形或组合图，而多边形由直线组成，组合图由各种线型混合而成。当用户要求画一幅蓝图时，系统便通过蓝图与图形之间的关联（聚集）关系，由图形来完成画图工作，但是图形是抽象类，没有画具体

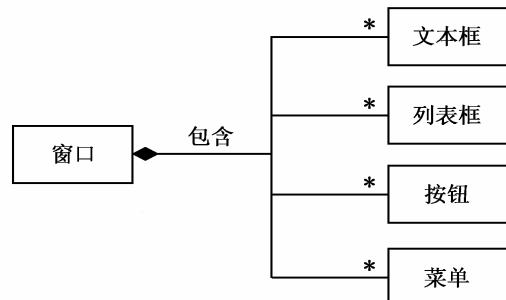


图 5.13 复合聚集示例

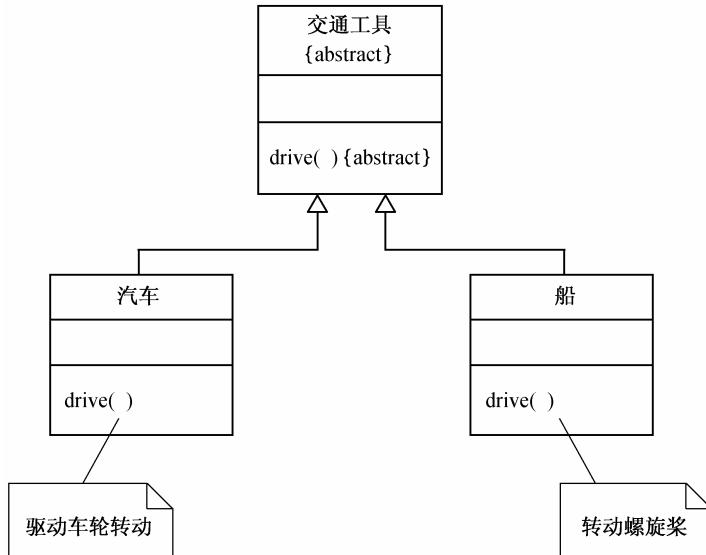


图 5.14 抽象类示例

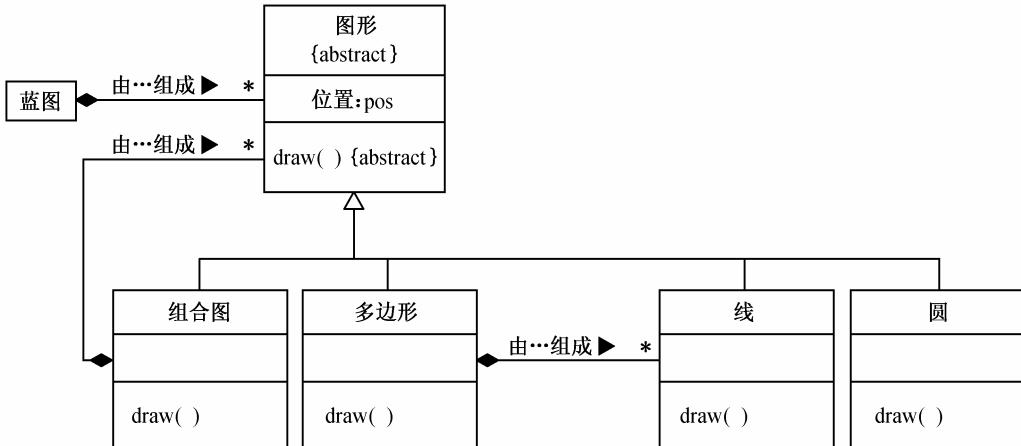


图 5.15 复杂类图示例

图形的功能，因此当涉及到某种具体图形（如，直线、圆等）时，便使用其相应子类中具体实现的 draw 功能完成绘图工作。

(2) 受限泛化

可以给泛化关系附加约束条件，以进一步说明该泛化关系的使用方法或扩充方法，这样的泛化关系称为受限泛化。UML 中预定义的约束有下述 4 种：多重、不相交、完全和不完全。这些约束都是语义约束。

多重继承指的是，一个子类可以同时多次继承同一个上层基类，例如图 5.16 中的水陆两用车类继承了两次交通工具类。

不相交继承与多重继承正好相反，它指的是一个子类不能同时多次继承同一个上层基类。如果类图中没有指定 { 多重 } 约束，则是不相交继承，一般的继承都是不相交继承。

完全继承指的是父类的所有子类都已在类图中穷举出来了，图示符号是指定 { 完全 } 约束。

不完全继承与完全继承恰好相反，父类的子类并没有都穷举出来，随着对问题理解的深入，可不断补充和修改，这为日后系统的扩充和维护带来很大方便。非完全继承是一般情况下默认的继承关系。

3. 依赖和细化

(1) 依赖关系

依赖关系描述两个模型元素（类、用例等）之间的下述语义连接关系：其中一个模型元素是独立的，另一个模型元素不是独立的，它依赖于独立的模型元素，如果独立的模型元素改变了，将影响依赖于它的模型元素。例如，一个类使用另一个类的对象作为操作的参数，一个类用另一个类的对象作为自己的数据成员，一个类向另一个类发送消息等，这样的两个类之间都存在依赖关系。

在 UML 的类图中，用带箭头的虚线连接有依赖关系的两个类，箭头指向其中独立的类。在虚线上可以带一个版类标签，具体说明依赖的种类，例如，图 5.17 表示一个友元依赖关系，该依赖关系使得 B 类的操作可以直接使用 A 类中私有的或保护的成员。

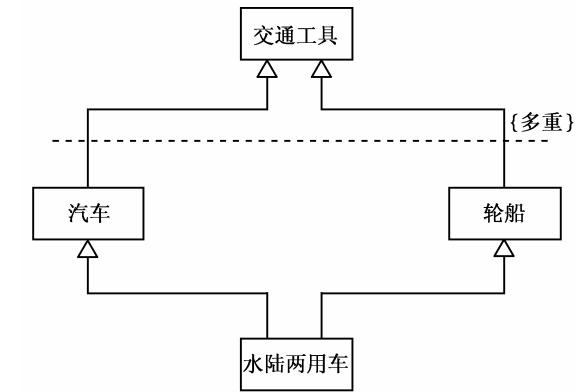


图 5.16 多重继承示例



图 5.17 友元依赖关系

(2) 细化关系

虽然在软件开发的不同阶段都使用类图，但是在不同阶段画出的类图表示了不同层次的抽象。类图可以分为三个抽象层次，当然，其他任何模型也都可以分为三个层次，只是对于类图来说，三个层次显得更为明显。下面简要地介绍这三个层次。

? 概念层

在需求分析阶段使用概念层类图描述应用领域中的概念。概念层模型应该独立于实现它的软件和程序设计语言。

? 说明层

在设计阶段使用说明层类图描述软件的接口部分（类与类之间的接口，即公有的操作），而不是描述软件的实现部分。

? 实现层

在实现阶段使用实现层类图描述软件中类的实现。只有在实现层才真正有类的概念（在类图中对类做了完全符合 UML 规定的、完整准确的描述），并描述了软件的实现部分。这可能是多数人最常用的类图，但是在许多情况下概念层和说明层的类图对于开发者之间的相互理解和交流更有帮助。

理解上述层次对于画类图和读懂类图都是至关重要的。画图时要从一个清晰的层次观念出发；而在读图时则要弄清该图是根据哪种层次观点绘制的。要正确地理解类图，首先应该正确地理解上述三个层次。虽然把类图分成三个层次的观点并不是 UML 的组成部分，但是它们对于建模或者评价模型都非常有用。尽管迄今为止人们似乎更强调实现层类图，但是应用 UML 可建立任何层次的类图，而且实际上另外两个层次的类图更有用。

当对同一事物在不同抽象层次上描述时，这些描述之间具有细化关系。细化是 UML 中的术语，表示对事物更详细一层的描述。假设两个元素 A 和 B 描述同一个事物，它们的区别是抽象层次不同，如果 B 是在 A 的基础上的更详细的描述，则称 B 细化了 A，或称 A 细化成了 B。细化的图示符号为由元素 B 指向元素 A 的、一端为空心三角的虚线（不是实线）。

例如，在需求分析阶段粗略地描述了一个类，在设计阶段对这个类做了更全面更具体地描述，这两个类图之间存在细化关系，如图 5.18 所示。

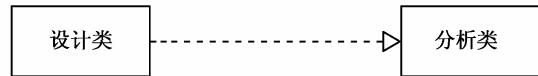


图 5.18 细化关系的图示符号

5.5 动 态 模 型

动态模型表示瞬时的、行为化的系统的“控制”性质，它规定了对象模型中的对象的合法变化序列。

一旦建立起对象模型之后，就需要考察对象的动态行为。所有对象都具有自己的生命周期（或称为运行周期）。对一个对象来说，生命周期由许多阶段组成，在每个特定阶段中，都有适合该对象的一组运行规律和行为规则，用以规范该对象的行为。生命周期中的阶段也就是对象的状态。所谓状态，是对对象属性值的一种抽象。当然，在定义状态时应该忽略那些不影响对象行为的属性。各对象之间相互触发（即作用），就形成了一系列的状态变化。我们把一个触发行为称作一个事件。对象对事件的响应，取决于接受该触发的对象当时所处的状态，响应包括改变自己的状态或者又形成一个新的触发行为。

通常，用状态图来描绘对象的状态、触发状态转换的事件、以及对象的行为（对事件的响应）。

每个类的动态行为用一张状态图来描绘，各个类的状态图通过共享事件合并起来，从而构成系统的动态模型。也就是说，动态模型是基于事件共享而互相关联的一组状态图的集合。

5.5.1 概念

下面讲述在建立动态模型时需要使用的一些基本概念，正确地理解这些概念是成功地建立

起软件系统的动态模型的关键。

1. 事件

事件是某个特定时刻所发生的事情，它是对引起对象从一种状态转换到另一种状态的现实世界中的事件的抽象。事件没有持续时间，是瞬间完成的。

事件也就是信息从一个对象到另一个对象的单向传送。受此事件触发的第二个对象，可以发送答复事件也可以不发送答复事件，即使发送答复事件，它也是受第二个对象控制的一个独立事件。

简而言之，事件就是引起对象状态转换的控制信息。

2. 状态

状态就是对象在其生命周期中的某个特定阶段所处的某种情形，它是对影响对象行为的属性值的一种抽象。

状态规定了对象对输入事件的响应方式。对象对事件的响应，既可以是作一个（或一系列）动作，也可以是仅仅改变对象本身的状态。

状态有持续性，它占用一段时间间隔。状态与事件密不可分，一个事件分开两个状态，一个状态隔开两个事件。事件表示时刻，状态代表时间间隔。

在定义状态的时候，应该忽略那些不影响对象行为的属性。

3. 行为

所谓行为，是指对象达到某种状态时所做的一系列处理操作。这些操作是需要耗费时间的。

5.5.2 图示符号

图 5.19 给出了状态图中使用的表示符号。



图 5.19 状态图中使用的表示符号

1. 状态

状态是可以被观察到的系统行为模式，一个状态代表系统的一种行为模式。任何对象都具有状态，状态是对象执行了一系列活动的结果。当某个事件发生后，对象的状态将发生变化。

在状态图中可能定义了下述三种状态：初态（初始状态）、终态（最终状态）和中间状态。这三种状态分别用不同的图示符号表示，如图 5.19 所示。

(1) 初态和终态

状态图既可以表示循环运行过程，也可以表示单程生命周期。当描述循环运行过程时，通常不关心循环是怎样启动的。当描述单程生命周期时，需要标明初始状态和最终状态（创建对象时进入初始状态，对象生命期结束时到达最终状态）。在状态图中，初始状态用实心圆表示，

最终状态用一对同心圆（内圆为实心圆）表示。

在一张状态图中只能有一个初态，而终态则可以有多个。

（2）中间状态

中间状态用圆角矩形表示，它可能包含三个部分，如图 5.19 所示。第一部分为状态的名称；第二部分为状态变量的名字和值，这部分是可选的（即可有可无）；第三部分是活动表，这部分也是可选的。

UML 描述“活动”的语法如下：

事件名（参数表）/ 动作表达式

其中，

“事件名”可以是任何事件的名称，最经常使用的是 entry（进入）、exit（退出）和 do（做）这三种标准事件。entry 事件指定进入该状态的动作，exit 事件指定退出该状态的动作，而 do 事件则指定在该状态下的动作。

需要时可以为事件指定参数表，它的语法格式与类的操作的参数表语法格式相似（参见 5.4.1 小节）。

动作表达式指定应做的动作。

2. 状态转换

状态图中两个状态之间带箭头的连线称为状态转换，也称为状态变迁或状态迁移。

状态变迁通常是由事件触发的，在这种情况下应该在表示状态转换的箭头线上标出触发转换的事件表达式；如果在箭头线上未标出事件表达式，则意味着在源状态的内部活动执行完之后自动触发状态转换。

事件表达式的语法如下：

事件说明 [守卫条件] / 动作表达式^发送子句

其中，

事件说明的语法为：事件名（参数表）。

守卫条件是一个布尔表达式。如果同时使用守卫条件和事件说明，则当且仅当事件发生且布尔表达式成立时，状态转换才发生。如果只有守卫条件没有事件说明，则只要守卫条件为真状态转换就发生。

动作表达式是一个过程表达式，当状态转换开始时执行该表达式。

发送子句是动作的特例，它被用来在状态转换期间发送消息。

3. 举例

为了具体说明状态图的画法，下面以大家熟悉的电梯系统为例，画出它的状态图，如图 5.20 所示。

在“空闲”状态，把状态变量 timer 的值置为 0，然后连续递增 timer 的值，直到“上楼”或“下楼”事件发生或守卫条件“timer = 超时值”为真，触发状态转换。注意，从“空闲”状态到“在第一层”状态之间的状态转换，有一个守卫条件和一个动作表达式，但没有事件说明。因此，只要守卫条件“timer = 超时值”为真，状态转换就发生，这时将执行动作“下楼（第一层）”，然后状态由“空闲”转变为“在第一层”。

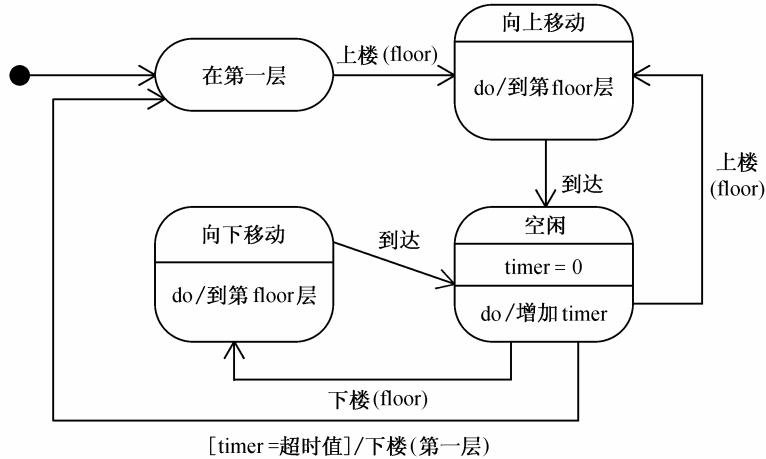


图 5.20 电梯的状态图

5.6 面向对象分析

不论采用哪种方法学开发软件，分析过程都是提取用户需求的过程。面向对象分析（通常缩写为 OOA），就是抽取和整理用户需求并用面向对象观点建立问题域模型的过程。

分析工作主要包括三项内容，这就是理解、表达和验证。

首先，系统分析员通过与用户及领域专家的多次反复交流，力求充分理解用户需求和该应用领域中的关键性背景知识。通常，快速建立起一个可在计算机上运行的原型系统，让用户试用原型系统并听取用户的反馈意见，能够更准确地抽取出用户的需求。

然后，系统分析员把对用户需求的理解，用无二义性的文档资料表达出来。分析过程得出的最重要的文档资料是软件需求规格说明书。面向对象的软件需求规格说明书，主要由对象模型、动态模型和功能模型组成。解决的问题不同，这三个模型的重要程度也不同：在任何情况下，对象模型都是最基本、最核心和最有价值的；当问题涉及交互作用和时序时（例如，人机界面和过程控制等），动态模型是重要的；解决运算量很大的问题时（例如，高级语言编译和工程计算等），功能模型是重要的。

由于要求解决的问题通常很复杂，而且人与人之间的交流带有随意性和非形式化的特点，上述理解和表达用户需求的过程不可能一次就达到理想的效果。因此，还必须进一步验证软件需求规格说明书的正确性、完整性和有效性，如果发现了问题则应及时修正。显然，需求分析过程是系统分析员与用户及领域专家反复交流和多次修正的过程。也就是说，理解、表达和验证的过程通常交替进行，反复迭代，而且往往需要利用原型系统作为辅助沟通的工具。

面向对象分析的关键，是识别出问题域内的对象，并分析确定它们相互之间的关系，最终建立起问题域的简洁、精确、可理解的正确模型。

建立对象模型的工作大体上按照下列顺序进行：寻找问题域内的对象，识别出对象间的关系、定义属性，定义服务。事实上，分析工作不可能严格地按照预定顺序进行，系统的模型往往需要反复构造多遍才能建成。通常，先构造出模型的子集，然后再逐渐扩充，直到完全、充分地理解了要求解决的问题，才能最终把问题域的模型建立起来。通常把分析阶段建立的问题域模型称为分析模型。

5.6.1 确定问题域内的对象

对象是在问题域中客观存在的，系统分析员的主要任务，就是通过分析找出这些对象。首先，找出所有候选的对象，然后，从候选对象中筛选掉不正确的或不必要的，从而得出问题域内应有的对象。

1. 找出候选的对象

对象是对问题域中有意义的事物的抽象，它们既可能是物理实体，也可能是抽象概念。具体地说，大多数客观事物可分为下述五类：

- (1) 可感知的物理实体，例如，飞机、汽车、书、房屋等等。
- (2) 人或组织的角色，例如，医生、教师、雇主、雇员、计算机系、财务处等等。
- (3) 应该记忆的事件，例如，飞行、演出、访问、交通事故等等。
- (4) 两个或多个对象的相互作用，通常具有交易或接触的性质，例如，购买、纳税、结婚等等。
- (5) 需要说明的概念，例如，政策、保险政策、版权法等等。

在分析所要解决的问题时，可以参照上列五类常见事物，找出在当前问题域内的候选对象。

另一种更简单的分析方法，是所谓的非正式分析。这种分析方法以用自然语言书写的需求陈述为依据，把陈述中的名词作为对象的候选者，用形容词作为确定属性的线索，把动词作为服务（操作）的候选者。当然，用这种简单方法确定的候选者是非常不准确的，其中往往包含大量不正确的或不必要的事物，还必须经过更进一步的严格筛选。通常，非正式分析是更详细、更精确的正式的面向对象分析的一个很好的开端。

例如，仔细阅读本章 5.1 节给出的对一个简单图形程序的需求陈述，可以看出，在需求陈述中使用了下列名词（或名词短语）：

显示器荧光屏，圆心坐标，位置，半径，圆，弧，起始角度，结束角度。

上列 8 个名词（或名词短语）可以作为该问题域内对象的候选者。

通常，在需求陈述中不会一个不漏地写出问题域内所有有关的对象，因此，分析员应该根据领域知识或常识进一步把隐含的对象找出来。

2. 筛选出正确的对象

显然，仅通过一个简单、机械的过程不可能正确地完成分析工作。非正式分析仅仅帮助我们找到一些候选的对象，接下来应该严格考察每个候选对象，从中去掉不正确的或不必要的，仅保留确实应该记录其信息或需要其提供服务的那些对象。

筛选时主要依据下列标准，删除不正确的或不必要的对象：

(1) 冗余

如果两个名词（或名词短语）代表同样的事物，则应该仅保留在此问题域中最富于描述力的名称。

(2) 无关

现实世界中存在许多对象，不能把它们都纳入到系统中去，仅需要把与本问题密切相关的对象放在目标系统中。有些对象在其他问题中可能很重要，但与当前要解决的问题无关，同样也应该把它们删掉。

(3) 笼统

在陈述需求时往往使用一些笼统的、泛指的名词，虽然在初步分析时把它们作为候选的对象列出来了，但是，要么系统无须记忆有关它们的信息，要么在需求陈述中有更明确更具体的名词对应它们所暗示的事物，因此，通常把这些笼统的或模糊的对象去掉。

(4) 属性

在需求陈述中有些名词实际上描述的是其他对象的属性，应该把这些名词从候选对象中去掉。当然，如果某个性质具有很强的独立性，则应该把它作为对象而不是作为属性。

(5) 操作

在需求陈述中有时可能使用一些既可作为名词，又可作为动词的词，应该慎重考虑它们在本问题中的含义，以便正确地决定把它们作为对象还是作为对象的操作。一般说来，本身具有属性需要独立存在的操作，应该作为对象；反之，则应该作为对象的操作。

例如，谈到电话时通常把“拨号”当作动词，当构造电话模型时确实应该把它作为一个操作，而不是一个类。但是，在开发电话的自动记账系统时，“拨号”需要有自己的属性（例如日期、时间、受话地点等），因此应该把它作为一个类。

(6) 实现

在分析阶段不应该过早地考虑怎样实现目标系统。因此，应该去掉仅和实现有关的候选对象。在设计和实现阶段，这些对象可能是重要的，但是在分析阶段过早地考虑它们反而会分散我们的注意力。

例如，刚才我们通过分析 5.1 节给出的对一个简单图形程序的需求陈述，找出了 8 个候选的对象。但是，它们仅仅是该问题域内对象的候选者，还需要对它们进行严格筛选，才能得出本问题域中真正有意义的对象。经过分析可以知道，“显示器荧光屏”是一种输出设备，它和需求陈述中没有提及的其他硬件设备一样，是运行程序的硬件平台，但却不是在本问题中有意义的事物，程序不需要保存有关显示器荧光屏的信息，因此应该把它从候选对象中删去。“圆心坐标”和“半径”实质上是圆和弧的基本属性，在本问题中并不需要独立存在，也应该从候选者中删去。“位置”实际上是指圆心的位置，也就是圆心坐标，没必要重复列出。“起始角度”和“结束角度”是弧的属性，并不需要独立存在，也应该从候选者中删去。最后得到的对象是“圆”和“弧”，更确切地说，是圆和弧两类对象。

5.6.2 确定关联

许多人习惯于在初步分析确定了问题域内的对象之后，接下来就分析确定在对象之间存在的关联关系。当然，这样的工作顺序并不是绝对必要的。由于在整个开发过程中面向对象概念和表示符号的一致性，分析员在选取自己习惯的工作方式时拥有相当大的灵活性。

如前所述，两个或多个对象之间的相互依赖、相互作用的关系就是关联。分析确定关联，能促使分析员考虑问题域的边缘情况，有助于发现那些尚未被发现的对象。

在分析确定关联的过程中，不必花过多的精力去区分关联和聚集。事实上，聚集不过是一

种特殊的关联，是关联的一个特例。

1. 初步确定关联

在需求陈述中使用的描述性动词或动词词组，通常表示关联关系。因此，在初步确定关联时，大多数关联可以通过直接提取需求陈述中的动词词组而得出。通过分析需求陈述，还能发现一些在陈述中隐含的关联。最后，分析员还应该与用户及领域专家讨论问题域实体间的相互依赖、相互作用关系，根据领域知识再进一步补充一些关联。

2. 筛选

经初步分析得出的关联只能作为候选的关联，还需经过进一步筛选，以去掉不正确的或不必要的关联。筛选时主要根据下述标准删除候选的关联：

(1) 已删去的对象之间的关联

如果在分析确定对象的过程中已经删掉了某个候选对象，则与这个对象有关的关联也应该删去，或用其他对象重新表达这个关联。

(2) 与问题无关的或应在实现阶段考虑的关联

应该把处在本问题域之外的关联或与实现密切相关的关联删去。

(3) 瞬时事件

关联应该描述问题域的静态结构，而不应该描述一个瞬时事件。

(4) 三元关联

三个或三个以上对象之间的关联，大多可以分解为二元关联。

(5) 派生关联

应该删掉那些可以用其他关联定义的冗余关联。

5.6.3 确定属性

属性是对象的性质，藉助于属性我们能对对象和结构有更深入更具体的认识。注意，在分析阶段不要用属性来表示对象间的关系，使用关联能够把两个对象间的关系表示得更清晰、更醒目。

一般说来，确定属性的过程包括分析和选择两个步骤。

1. 分析

通常，在需求陈述中用名词词组表示属性，例如，“汽车的颜色”或“光标的位置”。往往用形容词表示可枚举的具体属性，例如，“红色的”、“打开的”。但是，不可能在需求陈述中找到所有属性，分析员还必须藉助于领域知识和常识才能分析得出需要的属性。幸运的是，属性对问题域的基本结构影响很小。随着时间的推移，问题域中的类始终保持稳定，属性却可能改变了，相应地，类中方法的复杂程度也将改变。

属性的确定既与问题域有关，也和目标系统的任务有关。应该仅考虑与具体应用直接相关的属性，不要考虑那些超出所要解决的问题范围的属性。在分析过程中应该首先找出最重要的属性，以后再逐渐把其余属性增添进去。在分析阶段不要考虑那些纯粹用于实现的属性。

2. 选择

认真考察经初步分析而确定下来的那些属性，从中删掉不正确的或不必要的属性。通常有以下几种常见情况：

(1) 误把对象当作属性

如果某个实体的独立存在比它的值更重要，则应把它作为一个对象而不是对象的属性。

(2) 误把链属性当作对象的属性

如果某个性质依赖于某个关联的存在，则该性质是链属性，在分析阶段不应该把它作为对象的属性。

(3) 误把限定当成属性

限定实质上是一种特殊的链属性，不应该作为对象的属性。

(4) 误把内部状态当成了属性

如果某个性质是对象的非公开的内部状态，则应该从对象模型中删掉这个属性。

(5) 过于细化

在分析阶段应该忽略那些对大多数操作都没有影响的属性。

(6) 存在不一致的属性

类应该是简单而且一致的。如果得出一些看起来与其他属性毫不相关的属性，则应该考虑把该类分解成两个不同的类。

5.6.4 建立继承关系

确定了类中应该定义的属性之后，就可以利用继承机制共享公共性质，并对问题域中众多的类加以组织。正如以前曾经强调指出过的，继承关系的建立实质上是知识抽取过程，它应该反映出一定深度的领域知识，因此通常需要有领域专家密切配合才能完成。事实上，许多继承关系都是根据客观世界现有的分类模式建立起来的，只要可能就应该使用现有的概念。

一般说来，可以使用下述的两种方法建立继承（即泛化）关系：

1. 自底向上

抽象出现有类的共同性质泛化出父类，这个过程实质上模拟了人类的归纳思维过程。在开发 5.1 节所述的程序时，我们就是使用这种方法建立起类等级的。

2. 自顶向下

把现有类细化成更具体的子类，这模拟了人类的演绎思维过程。从需求陈述中往往能够看出应该做的细化工作，例如，带有形容词修饰的名词词组往往暗示了一些具体类。但是，在分析阶段应该避免过度细化。

在建立继承关系时，利用多重继承机制可以提高共享程度，但是同时也增加了概念上以及实现时的复杂程度。使用多重继承机制时，通常应该指定一个主要父类，从它继承大部分属性和行为，次要父类只补充少量属性和行为。

5.6.5 建立动态模型

对于仅存储静态数据的系统（例如数据库）来说，动态模型并没有什么意义。然而在开发交互式系统时，动态模型却起着很重要的作用。如果收集输入信息是目标系统的一项主要工作，则在开发这类应用系统时建立正确的动态模型是至关重要的。

建立动态模型的第一步，是编写典型交互行为的脚本。虽然脚本中不可能包括每个偶然事件，但是，至少必须保证不遗漏常见的交互行为。接下来从脚本中提取出事件，确定触发每个事件的动作对象以及接受事件的目标对象。第三步，排列事件发生的次序，确定每个对象可能有的状态及状态间的转换关系，并用状态图描绘它们。最后，比较各个对象的状态图，检查它

们之间的一致性，确保事件之间的匹配。

5.6.6 建立功能模型

功能模型表明了系统中数据之间的依赖关系，以及有关的数据处理功能，它由一组数据流图组成。其中的处理功能可以用 IPO 图（或表）伪码等多种方式进一步描述。

通常在建立了对象模型和动态模型之后再建立功能模型。

5.6.7 定义服务

正如以前曾经多次指出的那样，对象是由描述其属性的数据，以及可以对这些数据施加的操作（即服务），封装在一起构成的独立单元。因此，为了建立完整的对象模型，既要确定类中应该定义的属性，又要确定类中应该定义的服务。通常需要等到建立了动态模型和功能模型之后，才能最终确定类中应有的服务，因为这两个模型更明确地描述了每个类应该提供哪些服务。事实上，在确定类中应有的服务时，既要考虑该类实体的常规行为，又要考虑在本系统中特殊需要的服务。

1. 常规行为

在分析阶段可以认为，类中定义的每个属性都是可以访问的，也就是说，假设在每个类中都定义了读、写该类每个属性的操作。但是，通常无需在类图中显式表示这些常规操作。

2. 从事件导出的操作

状态图中发往对象的事件也就是该对象接收到的消息，因此该对象必须有由消息选择符指定的操作，这个操作修改对象状态（即属性值）并启动相应的服务。

3. 与处理框对应的操作

数据流图中的每个处理框都与一个对象（也可能是若干个对象）上的操作相对应。应该仔细对照状态图和数据流图，以便更正确地确定对象应该提供的服务。

4. 利用继承减少冗余操作

应该尽量利用继承机制以减少所需定义的服务数目。只要不违背领域知识和常识，就尽量抽取出相似类的公共属性和操作，以建立这些类的新父类，并在类等级的不同层次中正确地定义各个服务。

5.7 面向对象设计

如前所述，分析是提取和整理用户需求，并建立问题域精确模型的过程。设计则是把分析阶段得到的需求转变成符合成本和质量要求的、抽象的系统实现方案的过程。从面向对象分析到面向对象设计（通常缩写为 OOD），是一个逐渐扩充模型的过程。或者说，面向对象设计就是用面向对象观点建立求解域模型的过程。

尽管分析和设计的定义有明显区别，但是在实际的软件开发过程中二者的界限是模糊的。许多分析结果可以直接映射成设计结果，而在设计过程中又往往会加深和补充对系统需求的理解，从而进一步完善分析结果。因此，分析和设计活动是一个多次反复迭代的过程。面向对象方法学在概念和表示方法上的一致性，保证了在各项开发活动之间的平滑（无缝）过渡，领域

专家和开发人员能够比较容易地跟踪整个系统开发过程，这是面向对象方法与传统方法比较起来所具有的一大优势。

5.7.1 面向对象设计准则

所谓优秀设计，就是权衡了各种因素，从而使得系统在其整个生命周期中的总开销最小的设计。对大多数软件系统而言，60%以上的软件费用都用于软件维护，因此，优秀软件设计的一个主要特点就是容易维护。

本书第3章曾经讲述了指导软件设计的几条基本原理，这些原理在进行面向对象设计时仍然成立，但是增加了一些与面向对象方法密切相关的新的特点，从而具体化为下列的面向对象设计准则：

1. 模块化

面向对象软件开发模式，很自然地支持了把系统分解成模块的设计原理：对象就是模块，它是把数据结构和操作这些数据的方法紧密地结合在一起所构成的模块。

2. 抽象

面向对象方法不仅支持过程抽象，而且支持数据抽象。类实际上是一种抽象数据类型，它对外开放的公共接口构成了类的规格说明（即协议），这种接口规定了外界可以使用的合法操作符，利用这些操作符可以对类实例中包含的数据进行操作。使用者无须知道这些操作符的实现算法和类中数据元素的具体表示方法，就可以通过这些操作符使用类中定义的数据。通常把这类抽象称为规格说明抽象。

此外，某些面向对象的程序设计语言还支持参数化抽象。所谓参数化抽象，是指当描述类的规格说明时并不具体指定所要操作的数据类型，而是把数据类型作为参数。这使得类的抽象程度更高，应用范围更广，可重用性更高。例如，C++语言提供的“模板”机制就是一种参数化抽象机制。

3. 信息隐藏

在面向对象方法中，信息隐藏通过对对象的封装性实现：类结构分离了接口与实现，从而支持了信息隐藏。对于类的用户来说，属性的表示方法和操作的实现算法都应该是隐藏的。

4. 弱耦合

耦合指一个软件结构内不同模块之间互连的紧密程度。在面向对象方法中，对象是最基本的模块，因此，耦合主要指不同对象之间相互关联的紧密程度。弱耦合是优秀设计的一个重要标准，因为这有助于使得系统中某一部分的变化对其他部分的影响降到最低程度。在理想情况下，对某一部分的理解、测试或修改，无须涉及系统的其他部分。

如果一类对象过多地依赖其他类对象来完成自己的工作，则不仅给理解、测试或修改这个类带来很大困难，而且还将大大降低该类的可重用性和可移植性。显然，类之间的这种相互依赖关系是紧耦合的。

当然，对象不可能是完全孤立的，当两个对象必须相互联系相互依赖时，应该通过类的协议（即公共接口）实现耦合，而不应该依赖于类的具体实现细节。

一般说来，对象之间的耦合可分为两大类，下面分别讨论这两类耦合：

(1) 交互耦合

如果对象之间的耦合通过消息连接来实现，则这种耦合就是交互耦合。为使交互耦合尽可能

能松散，应该遵守下述准则：

? 尽量降低消息连接的复杂程度。应该尽量减少消息中包含的参数个数，降低参数的复杂程度。

? 减少对象发送（或接收）的消息数。

(2) 继承耦合

与交互耦合相反，应该提高继承耦合程度。继承是一般化类与特殊类之间耦合的一种形式。从本质上讲，通过继承关系结合起来的基类和派生类，构成了系统中粒度更大的模块。因此，它们彼此之间应该结合得越紧密越好。

为获得紧密的继承耦合，特殊类应该确实是对它的一般化类的一种具体化，也就是说，它们之间在逻辑上应该存在“ISA”的关系。因此，如果一个派生类摒弃了它基类的许多属性，则它们之间是松耦合的。在设计时应该使特殊类尽量多继承并使用其一般化类的属性和服务，从而更紧密地耦合到其一般化类。

5. 强内聚

内聚衡量一个模块内各个元素彼此结合的紧密程度。也可以把内聚定义为：设计中使用的一个构件内的各个元素，对完成一个定义明确的目的所做出的贡献程度。在设计时应该力求做到高内聚。在面向对象设计中存在下述三种内聚：

(1) 服务内聚

一个服务应该完成一个且仅完成一个功能。

(2) 类内聚

设计类的原则是，一个类应该只有一个用途，它的属性和服务应该是高内聚的。类的属性和服务应该全都是完成该类对象的任务所必需的，其中不包含无用的属性或服务。如果某个类有多个用途，通常应该把它分解成多个专用的类。

(3) 泛化内聚

设计出的泛化（继承）结构，应该符合多数人的概念，更准确地说，这种结构应该是对相应的领域知识的正确抽取。

例如，虽然表面看来飞机与汽车有相似的地方（都由发动机驱动，都有轮子，……），但是，如果把飞机和汽车都作为“机动车”类的子类，则明显违背了人们的常识，这样的泛化结构是低内聚的。正确的做法是，设置一个抽象类“交通工具”，把飞机和机动车作为交通工具类的子类，而汽车又作为机动车类的子类。

一般说来，紧密的继承耦合与高度的泛化内聚是一致的。

6. 可重用

软件重用是提高软件开发生产率和目标系统质量的重要途径。重用基本上从设计阶段开始。重用有两方面的含义：一是尽量使用已有的类（包括开发环境提供的类库，及以往开发类似系统时创建的类），二是如果确实需要创建新类，则在设计这些新类的协议时，应该考虑将来的可重复使用性。

5.7.2 启发规则

人们使用面向对象方法学开发软件的历史虽然不长，但也积累了一些经验。总结这些经验得出了几条启发规则，它们往往能够帮助软件开发人员提高面向对象设计的质量。下面介绍几

十条主要的启发规则：

1. 设计结果应该清晰易懂

使设计结果清晰、易读、易懂，是提高软件可维护性和可重用性的重要措施。显然，人们不会重用那些他们不理解的设计。保证设计结果清晰易懂的主要措施如下：

(1) 用词一致

应该使名字与它所代表的事物一致，而且应该尽量使用人们习惯的名字。不同类中相似服务的名字应该相同。

(2) 使用已有的协议

如果开发同一软件的其他设计人员已经建立了类的协议，或者在所使用的类库中已有相应的协议，则应该使用这些已有的协议。

(3) 减少消息模式的数目

如果已有标准的消息协议，设计人员应该遵守这些协议。如果确需自己建立消息协议，则应该尽量减少消息模式的数目，只要可能，就使消息具有一致的模式，以利于读者理解。

(4) 避免模糊的定义

一个类的用途应该是有限的，而且应该从类名可以较容易地推想出它的用途。

2. 泛化结构的深度应适当

应该使类等级中包含的层数适当。一般说来，在一个中等规模（大约包含 100 个类）的系统中，类等级层数应保持为 7 ± 2 。不应该仅仅从方便编码的角度出发随意创建派生类，应该使泛化结构与领域知识或常识保持一致。

3. 设计简单的类

应该尽量设计小而简单的类，以便于开发和管理。当类很大的时候，要记住它的所有服务是非常困难的。经验表明，如果一个类的定义不超过一页纸（或两屏），则使用这个类是比较容易的。为使类保持简单，应该注意以下几点：

(1) 避免包含过多的属性

属性过多通常表明这个类过分复杂了，它所完成的功能可能太多了。

(2) 有明确的定义

为了使类的定义明确，分配给每个类的任务应该简单，最好能用一两个简单语句描述它的任务。

(3) 尽量简化对象之间的合作关系

如果需要多个对象协同配合才能做好一件事，则破坏了类的简明性和清晰性。

(4) 不要提供太多服务

一个类提供的服务过多，同样表明这个类过分复杂。典型地，一个类提供的公共服务不超过 7 个。

4. 使用简单的协议

一般说来，消息中的参数不要超过 3 个。当然，不超过 3 个的限制也不是绝对的，但是，经验表明，通过复杂消息相互关联的对象是紧耦合的，对一个对象的修改往往导致其他对象的修改。

5. 使用简单的服务

面向对象设计出的类中的服务通常都很小，一般只有 3~5 行源程序语句，可以用仅包含

一个动词和一个宾语的简单句子描述它的功能。如果一个服务由过多的源程序语句来实现，或者语句嵌套层次太多，或者使用了复杂的 DO_CASE（多分支）语句，则应该仔细分析这个服务，设法分解或简化它。一般说来，应该尽量避免使用复杂的服务。如果需要在服务中使用 DO_CASE 语句，通常应该考虑用泛化结构代替这个类的可能性。

5.8 面向对象分析与设计实例

重用是保证软件质量提高软件生产率的主要措施之一，“类”是目前比较理想的可重用的软构件。但是，当类的数量积累得很多时，寻找适合当前项目需要的可重用的类，也是一件相当令人头疼的事情。因此，迫切需要有一个功能适当、使用方便的类库管理系统，以帮助软件开发人员从类库中选出符合需要的类在当前项目中重用。

本节介绍一个简化的 C++ 类库管理系统的面向对象分析和设计过程（着重讲述概要的系统设计过程）。通过这个实例，一方面进一步具体讲述面向对象的软件开发技术，另一方面也为读者提供了一份实习材料。读者可以自己完成这个类库管理系统的详细设计（即对象设计），并用 Visual C++ 语言编程实现它，从而亲身体会用面向对象方法开发软件的过程。如果时间充裕，还可以进一步充实完善这个类库管理系统，使之成为一个比较实用的面向对象的软件开发工具。

5.8.1 面向对象分析

1. 陈述需求

这个类库管理系统的最主要用途，是管理用户在用 C++ 语言开发软件的漫长过程中逐渐积累起来的类，以便在今后的软件开发过程中能够从库中方便地选取出来重用的类。它应该具有编辑（包括添加、修改和删除）、储存和浏览等基本功能，下面是对它的具体需求：

- (1) 管理用 C++ 语言定义的类。
- (2) 用户能够方便地向类库中添加新的类，并能建立新类与库中原有类的关系。
- (3) 用户能够通过类名从类库中查询出指定的类。
- (4) 用户能够查看或修改与指定类有关的信息（包括数据成员的定义、成员函数的定义及这个类与其他类的关系）。
- (5) 用户能够从类库中删除指定的类。
- (6) 用户能够在浏览窗口中方便、快速地浏览当前类的父类和子类。
- (7) 具有“联想”浏览功能，也就是说，可以把当前类的某个子类或父类指定为新的当前类，从而浏览这个新当前类的父类和子类。
- (8) 用户能够查看或修改某个类的指定成员函数的源代码。
- (9) 本系统是一个简化的多用户系统，每个用户都可以建立自己的类库，不同类库之间互不干扰。
- (10) 对于用户的误操作或错误的输入数据，系统能给出适当的提示信息，并且仍然继续稳定地运行。
- (11) 系统易学易用，用户界面应该是 GUI 的。

2. 建立对象模型

(1) 确定问题域内的对象

从对这个类库管理系统的需求不难看出，组成这个系统的基本对象是“类库”和“类”。类是类库中的“条目”，不妨把它称为“类条目”(ClassEntry)。

类条目中应该包含的信息(即它的属性)主要有类名、父类列表、成员函数列表和数据成员列表。一个类可能有多个父类(多重继承)，对于它的每个父类来说，应该保存的信息主要是该父类的名字、访问权及虚基类标志(是否是虚基类)。对于每个成员函数来说，主要应该保存函数名、访问权、虚函数标志(是否是虚函数)、返回值类型、参数及函数代码等信息。在每个数据成员中主要应该记录数据名、访问权和数据类型等信息。我们把“父类”、“成员函数”和“数据成员”也都作为对象。

根据对这个类库管理系统的需求可以想到，类条目应该提供的服务主要是：设置或更新类名；添加、删除和更改父类；添加、删除和更改成员函数；添加、删除和更改数据成员。

类库包含的信息主要是库名和类条目列表。类库应该提供的服务主要是：向类库中插入一个类条目；从类库中删除一个类条目；把类库储存到磁盘上；从磁盘中读出类库(放到内存中)。

(2) 分析确定对象之间的关系

在这个问题域中，各个对象之间的逻辑关系相当简单。分析系统需求，并结合关于C++语言语法的知识，可以知道问题域中各个对象之间的关系是：一个用户拥有多个类库，每个类库由0或多个类条目组成，每个类条目由0或多个父类，0或多个数据成员及0或多个成员函数组成。图5.21是本问题域的对象模型。

本系统的功能和控制流程都比较简单，无须建立动态模型和功能模型，仅用对象模型就可以很清楚地描述这个系统了。事实上，在用面向对象方法开发软件的过程中，建立系统对象模型是最关键的工作。

5.8.2 面向对象设计

1. 设计类库结构

通常，类库中包含一组类，这一组类通过泛化、组合等关系组成一个有机的整体，其中泛化(即继承)关系对于重用来说具有特别重要的意义。

至少有两种数据结构可用来把类条目组织成类库，一种数据结构是二叉树，另一种是链表。

当用二叉树来存储类条目的时候，左孩子是子类，右孩子是兄弟类(即具有相同父类的类)。这种结构的优点是：存储结构直接反映了类的继承关系；容易查找当前类的子类和兄弟类。缺点是：遍历二叉树的开销较大，不论用何种方法遍历都需占用大量内存；插入、删除的算法比较复杂；不易表示有多个父类的类。

当用链表存储类条目的时候，链表中每个节点都是一个类条目。这种结构的优点是：结构简单，插入和删除的算法都相当简单；容易遍历。缺点是这种存储结构并不反映继承关系。

由于C++语言支持多重继承，类库中相当多的类可能具有多个父类，因此，容易表示具有多个父类的类应该作为选择类库结构的一条准则。此外，简单、方便，容易实现编辑操作和容易遍历，对这个系统来说也很重要。经过权衡，我们决定采用链表结构来组织类库。因为在每个类条目中都有它的父类列表，查找一个类的子类则需遍历类库，虽然开销较大但算法却相当简单。为了提高性能，可以增加冗余关联(即建立索引)，以加快查找子类的速度。

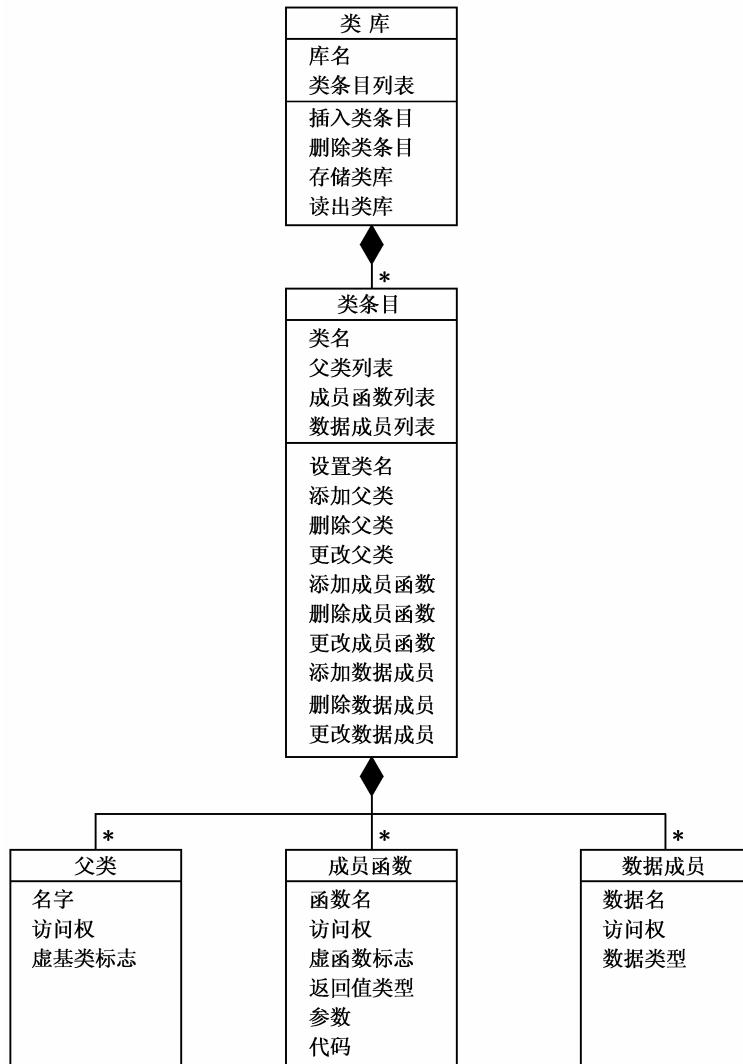


图 5.21 OOA 得出的对象模型

2. 设计问题域子系统

通过面向对象分析，我们对问题域已经有了较深入的了解，图 5.21 总结了我们对问题域的认识。在面向对象设计过程中，仅需从实现的角度出发，并根据我们所设计的类库结构，对图 5.21 所示的对象模型做一些补充和细化。

(1) 类条目 (Class Entry)

它的数据成员“父类列表”、“成员函数列表”和“数据成员列表”也都采用链表结构来存储。因此，在每个类条目的数据成员中，应该用“父类链表头指针”、“成员函数链表头指针”和“数据成员链表头指针”分别取代原来比较抽象的“父类列表”、“成员函数列表”和“数据成员列表”。

为了保存对每个类条目的说明信息，应该增加一个数据成员“注释”。

此外，类库中的各个类条目需要组成一条类链，因此，在类条目中还应该增加一个数据成员“指向下一个类条目的指针”。

类条目除了应该提供 5.8.1 节中所述的那些服务之外，为了实现需求陈述中提出的需求，还应该再增加下列服务：查找并取出指定父类的信息；查找并取出指定成员函数的信息；查找并取出指定数据成员的信息。

(2) 类库 (ClassEntryLink)

由于采用链表结构实现类库，每个类库实际上就是一条类链，因此把类库称为类条目链 (ClassEntryLink)。类库的数据成员“类条目列表”具体化为“类链头指针”。

一般说来，实用的类库管理系统应该采用数据库来存储类库。在我们这个简化的实例中，为了简化处理，决定使用标准的流式文件存储类库。

类库应该提供的服务主要有：取得库中类条目的个数；读文件并在内存中建立类链表；把内存中的类链表写到文件中；插入一个类条目；删除一个类条目；按类名查找类条目并把内容复制到指定地点。

(3) 父类 (ClassBase)，成员函数 (ClassFun) 和数据成员 (ClassData)

为了构造属于一个类条目的父类列表、成员函数列表和数据成员列表，在 ClassBase，ClassFun 和 ClassData 这三个类中除了应该定义 5.8.1 节中提到的那些数据成员之外，还应该分别增加数据成员“指向下一个父类的指针”、“指向下一个成员函数的指针”和“指向下一个数据成员的指针”。

综上所述，我们可以画出类库 (ClassEntryLink) 的示意图（见图 5.22）。

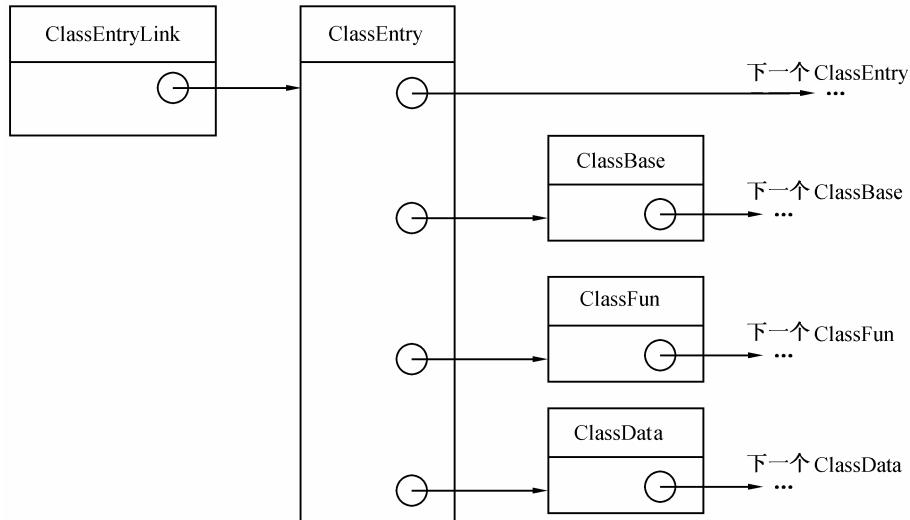


图 5.22 类库示意图

(4) 类条目缓冲区 (ClassEntryBuffer)

当编辑或查看类信息时，每个时刻用户只能面对一个类条目，我们把这个类称为当前类。为便于处理当前类，额外设置一个类条目缓冲区。它是从 ClassEntry 类派生出来的类，除了继承 ClassEntry 类中定义的数据成员和成员函数之外，主要增加了一些用于与窗口或类链交换数

据的成员函数。

每当用户要查看或编辑有关指定类的信息时，就把这个类条目从类库（即类链）中取到类条目缓冲区中。用户对这个类条目所做的一切编辑操作都只针对缓冲区中的数据，如果用户在编辑操作完成后不“确认”他的操作，则缓冲区中的数据不送回类库，因而也就不会修改类库的内容。

3. 设计人-机交互子系统

(1) 窗口

为方便用户使用，本系统采用图形用户界面。主要设计了下述一些窗口：

登录窗口

启动系统后即进入登录（即注册）窗口。它有一个编辑框供用户输入账号，一个账号与一个类库相对应。事实上，用户输入的账号就是类库的库名。如果在磁盘上已经存有与用户登录的账号相对应的类库文件，则在用户注册后，系统自动把文件中的数据读出到类链中，以便用户处理。

登录窗口中设置了“确认”和“放弃”按钮。

主窗口

用户注册之后进入主窗口，它有“创建”、“浏览”、“储存”和“退出”四个按钮。

单击“创建”按钮则进入创建窗口，在此窗口可以完成创建新的类条目或编辑原有类条目的功能。

单击“浏览”按钮则进入选择浏览方式窗口，可以选择适合自己需要的浏览方式，以浏览感兴趣的类。

单击“储存”按钮，则把内存中的类链保存到磁盘文件中。

单击“退出”按钮，则结束本系统的运行，退回到Windows操作系统。

创建窗口

本窗口有一个类名组合框，用于输入新类名或从已有类的列表中选择类名。类名指定了当前处理的类条目。

本窗口有三个分组框，分别管理对当前类的父类、成员函数和数据成员的处理。此外，本窗口还有一个编辑框，用于输入和编辑对这个类条目的说明信息。

上述三个分组框的每一个框中都有一个列表框，用户可以从中选择父类名（或成员函数名，或数据成员名）。此外，每个分组框中都有“添加”、“编辑”和“删除”等三个按钮。在不同分组框中单击“添加”或“编辑”按钮，将分别弹出父类编辑窗口、或成员函数编辑窗口、或数据成员编辑窗口。在所弹出的子窗口中可以完成添加新父类（或成员函数，或数据成员）或修改已有父类（或成员函数或数据成员）的信息的功能。这三个子窗口相对来说都比较简单，为节省篇幅，就不再单独讲述它们了。读者可以自行设计这三个子窗口。单击“删除”按钮，将删除指定的一个父类（或成员函数，或数据成员）。

在创建窗口中还设有“确认”和“放弃”按钮，单击这两个按钮中的某一个，则保留或放弃所创建（或编辑）的类条目。

选择浏览方式窗口

目前，本系统仅设计了两种浏览方式，分别是按类名浏览和按类关系浏览。因此，本窗口内设有一个分组框，框内有两个单选按钮，分别代表按类名浏览和按类关系浏览。

此外，本窗口内还有“确认”和“放弃”两个按钮。

类名浏览窗口

如果用户选定按类名浏览方式，则进入本窗口。本窗口有一个组合框，用户可以在这个框中输入类名，也可以从已有类的列表中选出一个类作为当前类。当用户通过类名指定了当前类之后，则在本窗口的一个编辑框中显示这个当前类的说明信息（即注释），并在 11 个列表框中分别列出父类名、访问权、虚基类标志；成员函数名、访问权、参数、返回类型、虚函数标志；数据成员名、访问权、类型等信息。当在上述列表框中选定一个成员函数之后，将在本窗口的另一个编辑框中显示这个函数的代码。

类关系浏览窗口

所谓按类关系浏览，就是按照类之间的继承关系浏览。本窗口中有一个组合框，用户可以输入类名，也可以从已有类的列表中选定一个类作为当前类。

此外，本窗口还有三个列表控制框，它们分别是父类框、当前类框和子类框。在用户选定了当前类之后，就在当前类框中显示这个类的图标和类名，同时在父类框和子类框中用图标和类名列出这个当前类的父类和子类。用鼠标双击某个父类或子类的图标，就把当前类改变成被双击的图标所代表的类，同时更新父类框和子类框的内容，分别列出新当前类的全部父类和子类，从而方便地做到了在相关类中漫游（即联想浏览）。

（2）重用

我们设计的是一个可重用类库管理系统，在设计和实现这个类库管理系统的过程中，自然应该尽可能重用已有的软构件。

我们采用面向对象方法分析和设计这个类库管理系统。面向对象语言是实现面向对象分析、设计结果的最佳语言。目前，C++ 语言是应用得最广泛的面向对象语言。现在在微机上流行的 C++ 开发工具主要有 Borland 公司的 BC 和 Microsoft 公司的 VC。我们经过权衡决定基于 VC 开发环境设计这个类库管理系统。

Visual C++ 所提供的 MFC 类库是编制 Windows 应用程序的得力工具。这个类库以层次结构来组织，其中封装了大部分 API 函数，它所包含的功能涉及整个 Windows 操作系统。MFC 不仅提供了 Windows 图形环境下的应用程序框架，而且提供了在创建应用程序时常用的组件。它成功地把面向对象和事件驱动这两个概念结合起来了，显示出这两种程序设计范型协同工作的强大生命力。

我们在设计过程中，尽可能重用 MFC 中提供的类，以构造我们的类库管理系统。系统中使用的许多类都是从 MFC 中的类直接派生出来的。

前述的每一个窗口都是一个适当的窗口类的实例，而这些窗口类都可以从 MFC 类库中的对话框类 CDialog 直接派生出来。对话框是一种特殊的弹出式窗口，应用程序可用它来显示某些提示信息。通常，对话框中还包含若干个控件，利用这些控件应用程序可以与用户进行数据交换，完成特定的输入 / 输出工作。由于在我们设计的窗口中全都使用控件与用户交互，因此，对话框类 CDialog 比一般的窗口类（例如 CFrameWnd）更适合本系统的需要。下面列出本系统中从 CDialog 类派生出的窗口类：

注册窗口：Login

主窗口：ClassTools

创建窗口：CreateClass

添加、编辑父类窗口 : CreateBase

添加、编辑成员函数窗口 : CreateFun

添加、编辑数据成员窗口 : CreateData

选择浏览方式窗口 : BrowseSelect

类名浏览窗口 : BrowseName

类关系浏览窗口 : BrowseInherit

4. 设计其他类

我们设计的这个类库管理系统虽然可以有多个用户，但是为了简单起见，限定各个用户只能以串行方式工作，也就是说，在同一时刻只能有一名用户使用这个系统。因此，本系统无须设置任务管理子系统。

如前所述，为了简化这个实例的分析和设计工作，我们并没有使用数据库管理系统来存储这个类库，而是使用普通文件系统存储它。读、写文件的功能由 ClassEntryLink 类中定义的两个成员函数完成，因此，本系统也不包含数据管理子系统。

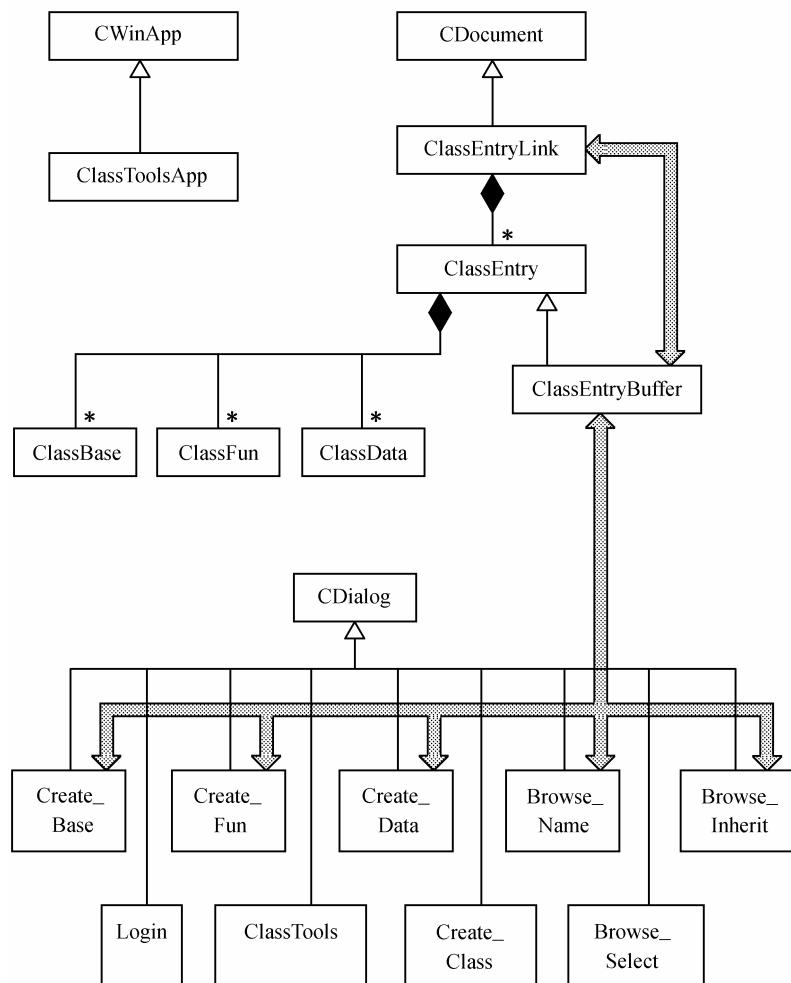


图 5.23 OOD 得出的对象模型

尽管本系统仅由问题域子系统和人-机交互子系统组成，但是，仅有前面讲述的那些类还是不够的。所有利用 MFC 类库开发的 Windows 应用程序，都必须包含一个特定的应用类及其实例。它相当于主函数，主要作用是为应用程序建立消息循环机制。通常，从 MFC 类库中的应用程序类 CWinApp，派生出应用系统需要的特定的应用类。在本系统中，从 CWinApp，派生出的应用类称为 ClassToolsApp，它主要是重载了 CWinApp 类中用于初始化应用窗口实例的成员函数 InitInstance()。

此外，类库类 ClassEntryLink 具有读、写文件的功能，因此，我们利用 MFC 类库中的文档类 CDocument 派生出这个类库类。

最后，我们用图 5.23 总结对 C++ 类库管理系统进行面向对象设计所得出的结果。图中的粗箭头线表示对象之间的消息连接（在本例中主要用于交换数据）。

5.9 面向对象实现

面向对象实现主要包括两项工作：把面向对象设计结果，翻译成用某种程序语言书写的面向对象程序；测试并调试面向对象的程序。

面向对象程序的质量基本上由面向对象设计的质量决定，但是，所采用的程序语言的特点和程序设计风格也将对程序的可靠性、可重用性及可维护性产生深远影响。

目前，软件测试仍然是保证软件可靠性的主要措施，对于面向对象的软件来说，情况仍然如此。但是，面向对象软件也给测试带来一些新特点和新问题，我们必须通过实践，努力探索适合面向对象软件的测试方法。

5.9.1 面向对象的程序设计语言

面向对象设计的结果，既可以用面向对象语言，也可以用非面向对象语言实现。使用面向对象语言时，由于语言本身充分支持面向对象概念的实现，因此，编译程序可以自动把面向对象概念映射到目标程序中。使用非面向对象语言编写面向对象程序，则必须由程序员自己把面向对象概念映射到目标程序中。因此，面向对象的程序设计语言是实现面向对象设计结果的最佳语言。

20 世纪 80 年代以来，面向对象语言像雨后春笋一样大量涌现，形成了两大类面向对象语言。一类是纯面向对象语言，如 Smalltalk 和 Eiffel 等语言。另一类是混合型面向对象语言，也就是在过程语言的基础上增加面向对象机制，如 C++ 等语言。

一般说来，纯面向对象语言着重支持面向对象方法研究和快速原型的实现，而混合型面向对象语言的目标则是提高运行速度和使传统程序员容易接受面向对象思想。成熟的面向对象语言通常都提供丰富的类库和强有力的开发环境。

自从 1983 年 AT&T 贝尔实验室推出 C++ 语言以来，就在计算机业界引起了广泛重视，成为目前应用得最广泛的面向对象语言。C++ 既支持面向过程的程序设计方法，也支持面向对象的程序设计方法，因此，是一种混合型语言。C++ 语言全面支持数据抽象、数据封装、参数化抽象、继承性和多态性，同时又充分保留了 C 语言的简洁性和高效性。

Java 是 1995 年 6 月由 Sun 公司推出的革命性编程语言，它继承了 C++ 语言的优点，克

服了 C++ 语言的缺点，具有简单、面向对象、稳定、与平台无关、解释型、多线程、动态等特点。Java 是目前使用得最广泛的网络编程语言之一。

5.9.2 面向对象程序设计风格

良好的程序设计风格对于面向对象实现来说尤其重要，不仅能明显减少维护软件的开销，而且有助于在开发新软件时重用已有的程序代码。

良好的面向对象程序设计风格，既包括传统的程序设计风格准则，也包括为适应面向对象方法所特有的概念（例如，继承性）而必须遵循的一些新准则。

1. 提高可重用性

面向对象方法的一个主要目标，就是提高软件的可重用性。重用也叫再用或复用，是指同一事物不经修改或稍加改动就在不同应用环境中多次重复使用。软件重用有多个层次，在编码阶段主要涉及代码重用问题。一般说来，代码重用有下述两种类型：一种是本项目内的代码重用，另一种是新项目重用旧项目的代码。内部重用主要是找出设计中相同或相似的部分，然后利用继承机制共享它们。为了做到外部重用（即一个项目重用另一个项目的代码），则必须有长远眼光，从有利于重用的角度反复考虑精心设计。虽然为实现外部重用而需要考虑的因素，比为实现内部重用而需要考虑的因素更多，但是，有助于实现这两类重用的程序设计准则却是相同的。下面讲述主要的程序设计准则：

(1) 提高方法的内聚

一个方法（即服务）应该只完成单个功能。如果某个方法涉及两个或多个不相关的功能，则应该把它分解成几个更小的方法。

(2) 减小方法的规模

应该减小方法的规模，如果某个方法规模过大（代码长度超过一页纸可能就太大了），则应该把它分解成几个更小的方法。

(3) 保持方法的一致性

保持方法的一致性，有助于实现实例化重用。一般说来，功能相似的方法应该有一致的名字、参数特征（包括参数个数、类型和次序）、返回值类型、使用条件及出错条件等。

(4) 把策略与实现分开

从所完成的功能看，有两种不同类型的方法。一类方法负责做出决策，提供变元，并且管理全局资源，可称为策略方法。另一类方法负责完成具体的操作，但却并不做出是否执行这个操作的决定，也不知道为什么执行这个操作，可称为实现方法。

策略方法通常紧密依赖于具体应用，这类方法比较容易编写，也比较容易理解，但是可重用性较差。

实现方法仅仅针对具体数据完成特定处理，通常用于实现复杂的算法。实现方法是自含式算法，相对独立于具体应用，因此，在其他应用系统中也可能重用它们。

为提高可重用性，在编程时不要把策略和实现放在同一个方法中，应该把算法的核心部分放在一个单独的具体实现方法中。为此需要从策略方法中提取出具体参数，作为调用实现方法的变元。

(5) 全面覆盖

如果输入条件的各种组合都可能出现，则应该针对所有组合写出方法，而不能仅仅针对当

前用到的组合情况写方法。例如，如果在当前应用中需要写一个方法，以获取表中第一个元素，则至少还应该为获取表中最后一个元素再写一个方法。

此外，一个方法不应该只能处理正常值，对空值、极限值及界外值等异常情况也应该能够作出有意义的响应。

(6) 尽量不使用全局信息

应该尽量降低方法与外界的耦合程度，不使用全局信息是降低耦合度的一项主要措施。

(7) 利用继承机制

在面向对象程序中，使用继承机制是实现共享和提高重用程度的主要途径。

调用子过程

最简单的做法是把公共的代码分离出来，构成一个被其他方法调用的公用方法。可以在基类中定义这个公用方法，供派生类中的方法调用，如图 5.24 所示。

分解因子

有时提高相似类代码可重用性的一个有效途径，是从不同类的相似方法中分解出不同的“因子”（即不同的代码），把余下的代码作为公用方法中的公共代码，把分解出的因子作为名字相同算法不同的方法，放在不同类中定义，并被这个公用方法调用，如图 5.25 所示。使用这种途径通常额外定义一个抽象基类，并在这个抽象基类中定义公用方法。把这种途径与面向对象语言提供的多态性机制结合起来，让派生类继承抽象基类中定义的公用方法，可以明显降低为增添新子类而需付出的工作量，因为只需在新子类中编写其特有的代码。

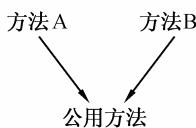


图 5.24 通过调用公用方法
实现代码重用

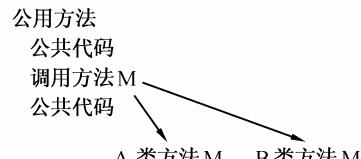


图 5.25 通过因子分解实现代码重用

2. 提高可扩充性

刚才讲述的提高可重用性的准则，也能提高程序的可扩充性。此外，下述的面向对象程序设计准则也有助于提高程序的可扩充性：

(1) 封装类的实现策略

应该把类的实现策略（包括描述属性的数据结构、修改属性的算法等）封装起来，对外只提供公有的接口，否则将降低今后修改数据结构或算法的自由度。

(2) 尽量定义简单的方法

一个方法应该只包含对象模型中的有限内容。违反这条准则将导致方法过分复杂，既不易理解，也不易修改扩充。

(3) 避免使用多分支语句

一般说来，可以利用 DO_CASE 语句测试对象的内部状态，而不要用来根据对象类型选择应有的行为，否则在增添新类时将不得不修改原有的代码。应该合理地利用多态性机制，根据对象当前类型，自动决定应有的行为。

(4) 精心确定公有方法

公有方法是向公众公布的接口。对这类方法的修改往往涉及许多其他类，因此，修改公有方法的代价通常都比较高。为提高可修改性，降低维护成本，必须精心选择和定义公有方法。私有方法是仅在类内使用的方法，通常利用私有方法来实现公有方法。删除、增加或修改私有方法所涉及的面要窄得多，因此代价也比较低。

同样，属性和关联也可以分为公有和私有两大类，公有的属性或关联又可进一步设置为具有只读权限或只写权限两类。

3. 提高健壮性

所谓健壮性，是软件的一个质量因素，指的是在硬件发生故障、输入的数据无效或操作错误等意外情况下，软件系统能做出适当响应的程度。

程序员在编写实现方法的代码时，既应该考虑效率，也应该考虑健壮性。通常需要在健壮性与效率之间做出适当的折衷。必须认识到，对于任何一个实用软件来说，健壮性都是不可忽略的质量指标。为提高健壮性应该遵守以下几条准则：

(1) 预防用户的操作错误

软件系统必须具有处理用户操作错误的能力。当用户在输入数据时发生错误，不应该引起程序运行中断，更不应该造成“死机”。任何一个接收用户输入数据的方法，对其接收到的数据必须进行检查，即使发现了非常严重的错误，也应该给出恰当的提示信息，并准备再次接收用户的输入。

(2) 检查参数的合法性

对公有方法，尤其应该着重检查其参数的合法性，因为用户在使用公有方法时可能违反参数的约束条件。

(3) 不要预先确定限制条件

在设计阶段，往往很难准确地预测出应用系统中使用的数据结构的最大容量需求。因此不应该预先设定限制条件。如果有必要和可能，则应该使用动态内存分配机制，创建未预先设定限制条件的数据结构。

(4) 先测试后优化

为在效率与健壮性之间做出合理的折衷，应该在为提高效率而进行优化之前，先测试程序的性能，人们常常惊奇地发现，事实上大部分程序代码所消耗的运行时间并不多。应该仔细研究应用程序的特点，以确定哪些部分需要着重测试（例如，最坏情况出现的次数及处理时间，可能需要着重测试）。经过测试，合理地确定为提高性能应该着重优化的关键部分。如果实现某个操作的算法有许多种，则应该综合考虑内存需求、速度及实现的简易程度等因素，经合理折衷选定适当的算法。

5.9.3 面向对象测试

测试计算机软件的经典策略是，从“小型测试”开始，逐步过渡到“大型测试”，用软件测试的专业术语来说，就是从单元测试开始，逐步进入集成测试，最后进行确认测试和系统测试。对于传统的软件系统来说，单元测试集中测试最小的可编译的程序单元（过程模块），一旦把这些单元都测试完之后，就把它们集成到程序结构中去，与此同时应该进行一系列的回归测试，以发现模块接口错误和新单元加入到程序中所带来的副作用，最后，把系统作为一个整体来测试，以发现软件需求中的错误。测试面向对象软件的策略，与上述策略基本相同，但也

有许多新特点。

1. 面向对象的单元测试

当考虑面向对象的软件时，单元的概念改变了。“封装”导致了类和对象的定义，这意味着类和类的实例（对象）包装了属性（数据）和处理这些数据的操作（也称为方法或服务）。现在，最小的可测试单元是封装起来的类和对象。一个类可以包含一组不同的操作，而一个特定的操作也可能存在于一组不同的类中。因此，对于面向对象的软件来说，单元测试的含义发生了很大变化。

不能再孤立地测试单个操作，而应该把操作作为类的一部分来测试。让我们举例说明上述论点：考虑一个类层次，操作 X 在超类中定义并被一组子类继承，每个子类都使用操作 X，但是，X 调用子类中定义的操作并处理子类的私有属性。由于在不同的子类中使用操作 X 的环境有微妙的不同，因此有必要在每个子类的语境中测试操作 X。这就意味着，当测试面向对象软件时，传统的单元测试方法是无效的，我们不能再在“真空”中（即孤立地）测试操作 X。

2. 面向对象的集成测试

因为在面向对象的软件中不存在层次的控制结构，传统的自顶向下和自底向上的集成策略就没有意义了。此外，由于构成类的成分彼此间存在直接或间接的交互，一次集成一个操作到类中（传统的渐增式集成方法），通常是不可能的。

面向对象软件的集成测试有两种不同的策略：一是基于线程的测试(thread-based testing)，这种策略把响应系统的一个输入或一个事件所需要的一组类集成起来。分别集成并测试每个线程，同时应用回归测试以保证没有产生副作用。二是基于使用的测试(use-based testing)，这种方法首先测试几乎不使用服务器类的那些类（称为独立类），把独立类都测试完之后，接下来测试使用独立类的下一个层次的类（称为依赖类）。对依赖类的测试一个层次一个层次地持续进行下去，直至把整个软件系统构造完为止。

集群测试(cluster testing)是面向对象软件集成测试的一个步骤。在这个测试步骤中，用精心设计的测试用例检查一群相互协作的类（通过研究对象模型可以确定协作类），这些测试用例力图发现协作错误。

3. 面向对象的确认测试

在确认测试或系统测试层次，不再考虑类之间相互连接的细节。和传统的确认测试一样，面向对象软件的确认测试也集中检查用户可见的动作和用户可识别的输出。为了导出确认测试用例，测试人员应该认真研究动态模型和描述系统行为的脚本，以确定最可能发现用户交互需求错误的情景。

5.10 面向对象方法学的主要优点

本章前面各节讲述了面向对象的概念和面向对象的软件开发方法，本节总结面向对象方法学的主要优点，以便读者在实际工作中主动发扬这种方法学的优点，开发出高质量的软件。

1. 与人类习惯的思维方法一致

面向对象的开发方法与传统的面向过程的方法有本质不同，这种方法的基本原理是，使用现实世界的概念抽象地思考问题从而自然地解决问题。它强调模拟现实世界中的概念而不强调

算法，它鼓励开发者在软件开发的绝大部分过程中都用应用领域的概念去思考。在面向对象的开发方法中，计算机的观点是不重要的，现实世界的模型才是最重要的。面向对象的软件开发过程从始至终都围绕着建立问题领域的对象模型来进行：对问题领域进行自然的分解，确定需要使用的对象和类，建立适当的类等级，在对象之间传递消息实现必要的联系，从而按照人们习惯的思维方式建立起问题领域的模型，模拟客观世界。

事实上，人们认识客观世界解决现实问题的过程，是一个渐进的过程，人的认识需要在继承以前的有关知识的基础上，经过多次反复才能逐步深化。在人的认识深化过程中，既包括了从一般到特殊的演绎思维过程，也包括了从特殊到一般的归纳思维过程。人在认识和解决复杂问题时使用的最强有力的思维工具是抽象，也就是在处理复杂对象时，为了达到某个分析目的集中研究对象的与此目的有关的实质，忽略该对象的那些与此目的无关的部分。

面向对象方法学的出发点和基本原则，就是分析、设计和实现一个软件系统的方法和过程，尽可能接近人们认识世界解决问题的方法和过程，也就是使描述问题的问题空间和描述解法的解空间在结构上尽可能一致。也可以说，面向对象方法学的基本原则，是按照人们习惯的思维方式建立问题域的模型，开发出尽可能直观、自然地表现求解方法的软件系统。面向对象的软件系统中广泛使用的对象，是对客观世界中实体的抽象，对象实际上是抽象数据类型的实例，提供了理想的数据抽象机制，同时又具有良好的过程抽象机制（通过发消息使用公有成员函数）。对象类是对一组相似对象的抽象，类等级中上层的类是对下层类的抽象。因此，面向对象的环境提供了强有力的抽象机制，便于人在利用计算机软件系统解决复杂问题时使用习惯的抽象思维工具。此外，面向对象方法学中普遍进行的对象分类过程，支持从特殊到一般的归纳思维过程；面向对象方法学中通过建立类等级而获得的继承特性，支持从一般到特殊的演绎思维过程。

面向对象的软件技术为开发者提供了随着对某个应用系统的认识逐步深入和具体化的过程，而逐步设计和实现该系统的可能性，因为可以先设计出由抽象类构成的系统框架，随着认识深入和具体化再逐步派生出更具体的派生类。这样的开发过程符合人们认识客观世界解决复杂问题时逐步深化的渐进过程。

2. 面向对象软件稳定性好

传统的软件开发方法以算法为核心，开发过程基于功能分析和功能分解。用传统方法所建立起来的软件系统的结构紧密依赖于系统所要完成的功能，当功能需求发生变化时将引起软件结构的整体修改。事实上，用户需求变化大部分是针对功能的，因此，这样的软件系统是不稳定的。

面向对象方法基于构造问题领域的对象模型，以对象为中心构造软件系统。它的基本作法是用对象模拟问题领域中的实体，以对象间的联系刻画实体间的联系。因为面向对象的软件系统的结构是根据问题领域的模型建立起来的，而不是基于对系统应完成的功能的分解，所以，当对系统的功能需求变化时并不会引起软件结构的整体变化，往往仅需要作一些局部性的修改。例如，从已有类派生出一些新的子类以实现功能扩充或修改，增加或删除某些对象等等。总之，由于现实世界中的实体是相对稳定的，因此，以对象为中心构造的软件系统也是比较稳定的。

3. 面向对象软件可重用性好

用已有的零部件装配新的产品，是典型的重用技术，例如，可以用已有的预制件建筑一幢

结构和外形都不同于从前的新大楼。重用是提高生产率的最主要的方法。

传统的软件重用技术是利用标准函数库，也就是试图用标准函数库中的函数作为“预制件”来建造新的软件系统。但是，标准函数缺乏必要的“柔性”，不能适应不同应用场合的不同需要，并不是理想的可重用的软件成分。实际的库函数往往仅提供最基本、最常用的功能，在开发一个新的软件系统时，通常多数函数是开发者自己编写的，甚至绝大多数函数都是新编的。

使用传统方法学开发软件时，人们认为具有功能内聚性的模块是理想的模块，也就是说，如果一个模块完成一个且只完成一个相对独立的子功能，那么这个模块就是理想的可重用模块，而且这样的模块也更容易维护。基于这种认识，通常尽量把标准函数库中的函数做成功能内聚的。但是，事实上具有功能内聚性的模块并不是自含的和独立的，相反，它必须在数据上运行。如果要重用这样的模块，则相应的数据也必须重用。如果新产品中的数据与最初产品中的数据不同，则要么修改数据要么修改这个模块。

事实上，离开了操作便无法处理数据，而脱离了数据的操作也是毫无意义的，我们应该对数据和操作同样重视。在面向对象方法所使用的对象中，数据和操作正是作为平等伙伴出现的。因此，对象具有很强的自含性，此外，对象所固有的封装性和信息隐藏机理，使得对象的内部实现与外界隔离，具有较强的独立性。由此可见，对象类提供了比较理想的模块化机制和比较理想的可重用的软件成分。

面向对象的软件技术在利用可重用的软件成分构造新的软件系统时，有很大的灵活性。有两种方法可以重复使用一个对象类：一种方法是创建该类的实例，从而直接使用它；另一种方法是从它派生出一个满足当前需要的新类。继承性机制使得子类不仅可以重用其父类的数据结构和程序代码，而且可以在父类代码的基础上方便地修改和扩充，这种修改并不影响对原有类的使用。由于可以像使用集成电路（IC）构造计算机硬件那样，比较方便地重用对象类来构造软件系统，因此，有人把对象类称为“软件IC”。

面向对象的软件技术所实现的可重用性是自然的和准确的，在软件重用技术中它是最成功的一个。

4. 较易开发大型软件产品

当开发大型软件产品时，组织开发人员的方法不恰当往往是出现问题的主要原因。用面向对象范型开发软件时，可以把一个大型产品看作是一系列本质上相互独立的小产品来处理，这就不仅降低了开发的技术难度，而且也使得对开发工作的管理变得容易多了。这就是为什么对于大型软件产品来说，面向对象范型优于结构化范型的原因之一。许多软件开发公司的经验都表明，当把面向对象技术用于大型软件开发时，软件成本明显地降低了，软件的整体质量也提高了。

5. 可维护性好

用传统方法和面向过程语言开发出来的软件很难维护，是长期困扰人们的一个严重问题，是软件危机的突出表现。

由于下述因素的存在，使得用面向对象方法所开发的软件可维护性好：

? 面向对象的软件稳定性比较好。

如前所述，当对软件的功能或性能的要求发生变化时，通常不会引起软件的整体变化，往往只需对局部作一些修改。由于对软件所需做的改动较小且限于局部，自然比较容易实现。

? 面向对象的软件比较容易修改。

如前所述，类是理想的模块机制，它的独立性好，修改一个类通常很少会牵扯到其他类。如果仅修改一个类的内部实现部分（私有数据成员或成员函数的算法），而不修改该类的对外接口，则可以完全不影响软件的其他部分。

面向对象软件技术特有的继承机制，使得对软件的修改和扩充比较容易实现，通常只须从已有类派生出一些新类，无须修改软件原有成分。

面向对象软件技术的多态性机制，使得当扩充软件功能时对原有代码所需作的修改进一步减少，需要增加的新代码也比较少。

? 面向对象的软件比较容易理解。

在维护已有软件的时候，首先需要对原有软件与此次修改有关的部分有深入理解，才能正确地完成维护工作。传统软件之所以难于维护，在很大程度上是因为修改所涉及的部分分散在软件各个地方，需要了解的面很广，内容很多，而且传统软件的解空间与问题空间的结构很不一致，更增加了理解原有软件的难度和工作量。

面向对象的软件技术符合人们习惯的思维方式，用这种方法所建立的软件系统的结构与问题空间的结构基本一致。因此，面向对象的软件系统比较容易理解。

对面向对象软件系统所做的修改和扩充，通常通过在原有类的基础上派生出一些新类来实现。由于对象类有很强的独立性，当派生新类的时候通常不需要详细了解基类中操作的实现算法。因此，了解原有系统的工作量可以大幅度下降。

? 易于测试和调试。

为了保证软件质量，对软件进行维护之后必须进行必要的测试，以确保要求修改或扩充的功能按照要求正确地实现了，而且没有影响到软件不该修改的部分。如果测试过程中发现了错误，还必须通过调试改正过来。显然，软件是否易于测试和调试，是影响软件可维护性的一个重要因素。

对面向对象的软件进行维护，主要通过从已有类派生出一些新类来实现。因此，维护后的测试和调试工作也主要围绕这些新派生出来的类进行。类是独立性很强的模块，向类的实例发消息即可运行它，观察它是否能正确地完成要求它作的工作，对类的测试通常比较容易实现，如果发现错误也往往集中在类的内部，比较容易调试。

5.11 小 结

本章从一个面向对象的程序设计实例讲起，对面向对象的软件工程方法学做深入浅出的介绍，为读者今后进一步深入学习并在实际工作中运用这种方法学奠定坚实基础。

近些年来，面向对象方法学日益受到人们的重视，特别是在用这种方法开发大型软件产品时，可以把软件看作是由本质上相互独立的一系列小产品（对象）组成，这就不仅降低了开发工作的技术难度，而且也使得对开发工作的管理变得比较容易了。因此，对于大型软件产品来说，面向对象范型明显优于结构化范型。此外，使用面向对象范型能够开发出稳定性好、可重用性好和可维护性好的软件，这些都是面向对象方法学的突出优点。

面向对象方法学比较自然地模拟了人类认识客观世界的思维方式，它所追求的目标和遵循的基本原则，就是使描述问题的问题空间和在计算机中解决问题的解空间，在结构上尽可能一致。

面向对象方法学认为，客观世界由对象组成。任何事物都是对象，每个对象都有自己的内部状态和运动规律，不同对象彼此间通过消息相互作用、相互联系，从而构成了我们所要分析和构造的系统。系统中每个对象都属于一个特定的对象类。类是对具有相同属性和行为的一组相似对象的定义。应该按照子类、父类的关系，把众多的类进一步组织成一个层次系统，这样做了之后，如果不加特殊描述，则处于下一层级上的对象可以自动继承位于上一层次的对象的属性和行为。

用面向对象观点建立系统的模型，能够促进和加深对系统的理解，有助于开发出更容易理解、更容易维护的软件。通常，人们从三个互不相同然而又密切相关的角度建立起三种不同的模型。它们分别是描述系统静态结构的对象模型、描述系统控制结构的动态模型、以及描述系统计算结构的功能模型。其中，对象模型是最基本、最核心、最重要的。

本章所讲述的面向对象方法及定义的概念和表示符号，可以适用于整个软件开发过程。软件开发人员无须像用结构化分析、设计技术那样，在开发过程的不同阶段转换概念和表示符号。实际上，用面向对象方法开发软件时，阶段的划分是十分模糊的，通常在分析、设计和实现等阶段间多次迭代。

分析就是提取系统需求并建立问题域精确模型的过程，它包括理解、表达和验证等三项主要工作内容。面向对象分析的关键工作，是分析、确定问题域中的对象及对象间的关系，并建立起问题域的对象模型。

大多数分析模型都不是一次完成的，为了理解问题域的全部含义，必须反复多次地进行分析。因此，分析工作不可能严格地按照预定顺序进行；分析工作也不是机械地把需求陈述转变为分析模型的过程。分析员必须与用户及领域专家反复交流、多次磋商，及时纠正错误认识并补充缺少的信息。

分析模型是同用户及领域专家交流时有效的通信手段。最终的模型必须得到用户和领域专家的确认。在交流和确认的过程中，原型往往能起很大的促进作用。

面向对象设计，就是用面向对象观点建立求解空间模型的过程。通过面向对象分析得出的问题域模型，为建立求解空间模型奠定了坚实基础。分析与设计本质上是一个多次反复迭代的过程，而面向对象分析与面向对象设计的界限尤其模糊。

优秀设计是使得目标系统在其整个生命周期中总开销最小的设计，为获得优秀的设计结果，应该遵循一些基本准则。本章结合面向对象方法学固有的特点讲述了面向对象设计准则，并介绍了一些有助于提高设计质量的启发式规则。

本章 5.8 节讲述了一个简化的 C++ 类库管理系统的面向对象分析与设计过程，通过这个综合性的实例对前面各节讲过的概念、方法和准则做了复习和总结，认真阅读这一节有助于读者更深入、具体地理解面向对象分析与设计的方法，并对在开发软件的实践中怎样应用面向对象方法学有初步体会，同时，这一节的内容也为读者提供了一份较好的实习材料。

面向对象方法学把分析、设计和实现很自然地联系在一起了。虽然面向对象设计原则上不依赖于特定的实现环境，但是实现结果和实现成本却在很大程度上取决于实现环境。因此，直接支持面向对象设计范型的面向对象程序语言、开发环境及类库，对于面向对象实现来说是非常重要的。

良好的程序设计风格对于面向对象实现来说格外重要。它既包括传统的程序设计风格准则，也包括与面向对象方法的特点相适应的一些新准则。

面向对象方法学使用独特的概念和技术完成软件开发工作，因此，在测试面向对象程序的时候，除了继承传统的测试技术之外，还必须研究与面向对象程序特点相适应的新的测试技术。

习 题 五

1. 如果要求在屏幕上显示 3 个圆和 2 条弧，则 5.1 节讲述的分析与设计结果将有何变化？C++ 程序需要做哪些修改？
2. 在 5.1 节给出的程序中，Circle 类并没有定义属性 X 和 Y，该类的成员函数 Show 中使用的数据 X 和 Y 是从哪里来的？X 和 Y 的初值是怎样设置的？为什么这个成员函数可以直接读 X 和 Y 的值？
3. 如果把 Arc 类作为 Circle 类的父类，则 5.1 节的分析设计结果将有何变化？程序需要做哪些修改？试比较这种做法与书中所述做法的优缺点。
4. 用面向对象方法解决下述问题，要求通过需求分析确定需要使用的类和对象，并设计合理的类等级。

在显示器荧光屏上圆心坐标为 (250, 100) 的位置，画一个半径为 25 的小圆，圆内显示字符串“you”；在圆心坐标为 (250, 150) 的位置，画一个半径为 100 的中圆，圆内显示字符串“world”；再在圆心坐标为 (250, 250) 的位置画一个半径为 225 的大圆，圆内显示字符串“Universe”。

5. 什么是面向对象方法学？这种方法学主要有哪些优点？
6. 什么是“对象”？它与传统的数据有何关系？有何不同？
7. 什么是模型？开发软件为何要建模？
8. 什么是对象模型？建立对象模型时主要使用哪些图形符号？这些符号的含义是什么？
9. 什么是动态模型？建立动态模型时主要使用哪些图形符号？这些符号的含义是什么？
10. 什么是功能模型？建立功能模型时主要使用哪些图形符号？
11. 下面是自动售货机系统的需求陈述，请建立它的对象模型、动态模型和功能模型。自动售货机系统是一种无人售货系统。售货时，顾客把硬币投入机器的投币口中，机器检查硬币的大小、重量、厚度及边缘类型。有效的硬币是一元币、五角币、一角币、五分币和一分币。其他货币都被认为是假币。机器拒绝接收假币，并将其从退币孔退出。当机器接收了有效的硬币之后，就把硬币送入硬币储藏器中。顾客支付的货币根据硬币的面值进行累加。

自动售货机装有货物分配器。每个货物分配器中包含零个或多个价格相同的货物。顾客通过选择货物分配器来选择货物。如果货物分配器中有货物，而且顾客支付的货币值不小于该货物的价格，货物将被分配到货物传送孔送给顾客，并将适当的零钱返回到退币孔。如果分配器是空的，则和顾客支付的货币值相等的硬币将被送回到退币孔。如果顾客支付的货币值少于所选择的分配器中货物的价格，机器将等待顾客投进更多的货币。如果顾客决定不买所选择的货物，他投放的货币将从退币孔中退出。

12. 面向对象设计应该遵循哪些准则？简述每条准则的内容，并说明遵循这条准则的必要性。
13. 简述有助于提高面向对象设计质量的每条主要启发规则的内容和必要性。

第6章 软件维护

在软件的开发工作已完成并把软件产品交付给用户使用之后，就进入了软件运行维护阶段。这个阶段的工作目标是保证软件在一个相当长的时期内能够正常运行，因此对软件的维护就成为必不可少的了。

软件维护需要的工作量非常大，虽然在不同应用领域维护成本差别很大，但是，平均说来，大型软件的维护成本高达开发成本的四倍左右。目前国外许多软件开发组织把 60%以上的人力用于维护已有的软件，而且随着软件数量增多和使用寿命延长，这个百分比还在持续上升。将来维护工作甚至可能会束缚住软件开发组织的手脚，使他们没有余力开发新的软件。

本书前面各章讲述的软件工程方法学的主要目的就是要提高软件的可维护性，减少软件维护所需要的工作量，降低软件系统的总成本。

6.1 软件维护的定义与策略

6.1.1 定义

所谓软件维护就是在软件已经交付使用之后，为了改正错误或满足新的需要而修改软件的过程。我们可以通过描述软件交付使用后可能进行的下述四项活动，具体地定义软件维护。

1. 改正性维护

通常，在软件开发过程中所进行的测试都是不完全、不彻底的，软件中必然会有一些潜伏的错误被带到运行阶段来。这些潜伏的错误在某些特定的使用条件下就会暴露出来，用户将把他们遇到的问题报告给软件维护人员，要求解决。我们把诊断和改正软件错误的过程称为改正性维护。例如，在软件交付用户使用之后，解决在开发时没有测试所有可能的执行通路而带来的问题；解决程序中遗漏对文件中最后一个记录的处理的错误等。

2. 适应性维护

计算机科学技术领域的各个方面都在迅速进步，大约每过 36 个月就有新一代的硬件宣告出现，经常推出新操作系统或旧系统的修改版本，时常增加或修改外部设备和其他系统部件；另一方面，应用软件的使用寿命却很容易超过十年，远远长于最初开发这个软件时的运行环境的寿命。因此，适应性维护，也就是为了和变化了的环境适当地配合而进行的修改软件的活动，是既必要又经常的维护活动。

例如，适应性维护可以是修改原在 DOS 操作系统中运行的程序，使之能在 Windows 操作系统中运行；修改两个程序，使它们能够使用相同的记录结构；修改程序，使它适用于另外

一种终端设备。

3. 完善性维护

在使用软件的过程中，用户往往提出增加新功能或改变某些已有功能的要求，还可能提出提高程序性能的要求。为了满足这类要求而修改软件的活动，称为完善性维护。

例如，在储蓄系统交付银行使用之后，增加扣除利息税的功能；缩短系统的响应时间，使之达到新的要求；改变现有程序输出数据的格式，以方便用户；在正在运行的软件中增加联机求助功能等，都是完善性维护。

4. 预防性维护

当为了提高未来的可维护性或可靠性，或为了给未来的改进工作奠定更好的基础而修改软件时，就出现了第四类维护活动，这类维护活动称为预防性维护。通常，把预防性维护定义为：“把今天的方法学应用于昨天的系统以满足明天的需要”。也就是说，预防性维护就是采用先进的软件工程方法对需要维护的软件或软件中的某一部分，主动地进行重新设计、编码和测试。

在维护阶段的最初一二年，改正性维护的工作量往往比较大。随着在软件运行过程中错误发现率迅速降低并趋于稳定，就进入了正常使用期间。但是，由于用户经常提出改造软件的要求，适应性维护和完善性维护的工作量逐渐增加，而且在这种维护过程中往往又会引入新的错误，从而进一步加大了维护的工作量。

从上述关于软件维护的定义不难看出，软件维护绝不仅限于纠正使用中发现的错误，事实上在全部维护活动中一半以上是完善性维护。国外的统计数字表明，完善性维护占全部维护活动的 50% ~ 66%，改正性维护占 17% ~ 21%，适应性维护占 18% ~ 25%，其他维护活动只占 4% 左右。

6.1.2 策略

针对上一小节所述的三种典型的维护活动，James Martin 等人提出了一些可以减少维护成本的策略。下面讲述主要的软件维护策略。

1. 降低改正性维护成本的策略

显然，软件中包含的错误越少改正性维护的成本也就越低，但是，要生成 100% 可靠的软件通常成本太高，并不一定合算。然而通过使用先进技术仍然可以大大提高软件的可靠性，从而减少改正性维护的需求。下面列出可提高软件可靠性的主要技术。

- (1) 用数据库管理系统替代文件系统来存储需长期保存的信息；
- (2) 用软件开发环境或程序自动生成系统来开发软件或自动生成程序；
- (3) 用更高级的语言（例如，第四代语言）编写程序；
- (4) 只要可能就用购买应用软件包的办法代替自己开发应用软件的办法；
- (5) 使用结构化技术或面向对象技术，可以开发出更容易理解和测试的软件。
- (6) 采用防错程序设计技术，把自检能力引入程序。

2. 降低适应性维护成本的策略

这类维护是必然要进行的，但是采取适当的策略仍然能降低这类维护的需求。

(1) 在进行配置管理时，把硬件、操作系统和其他相关的环境因素的可能变化考虑在内，可以减少某些适应性维护的工作量；

(2) 把与硬件、操作系统及其他外围设备有关的代码放到特定的程序模块中，可以把因环境变化而必须修改的程序代码局限于某些特定的程序模块内；

(3) 使用内部程序列表、外部文件及例行处理程序包，可以为维护时修改程序提供方便。

3. 降低完善性维护成本的策略

上述的减少前两类维护成本的策略，通常也能降低完善性维护的成本。特别是数据库管理系统、程序自动生成系统、软件开发环境、第四代语言和应用软件包，可明显减少维护工作量。

此外，在需求分析过程中准确地预测用户将来可能提出的需求，并且在设计时为将来可能提出的需求预先做准备，显然是降低完善性维护成本的有力措施。

在实际开发软件之前，建立软件的原型并让用户试用，以进一步完善他们对软件的功能需求，也能显著减少软件交付使用之后的完善性维护需求。

6.2 软件维护的特点

6.2.1 结构化维护与非结构化维护差别悬殊

图 6.1 描绘了面对一项维护要求时，不同的软件配置所导致的不同工作流程。

如果软件配置的惟一成分是程序代码，

那么维护活动从艰苦地评价程序代码开始，而且常常由于程序内部文档不足而使评价更困难。诸如软件结构、全程数据结构、系统接口、性能和（或）设计约束等微妙的特点是难于搞清的，而且常常误解了这一类特点。最终对程序代码所做的改动的后果是难于估量的：因为没有测试方面的文档，所以不可能进行回归测试（即为了保证所做的修改没有在以前可以正常使用的软件功能中引入错误而重复过去做过的测试）。可惜，我们正在进行非结构化维护，并且正在为此而付出代价（浪费精力和受挫折），这种维护方式是没有使用良好定义的方法学开发出来的软件的必然结果。

如果有一个完整的软件配置存在，那么维护工作从评价设计文档开始，确定软件重要的结构特点、性能特点以及接口特点；估量要求的改动将带来的影响，并且计划实施途径。然后首先修改设计并且对所做的修改进行仔细复查。接下来编写相应的源程序代码；使用在测试说明书中包含的信息进行回归测试；最后，把修改后的软件再次交付使用。

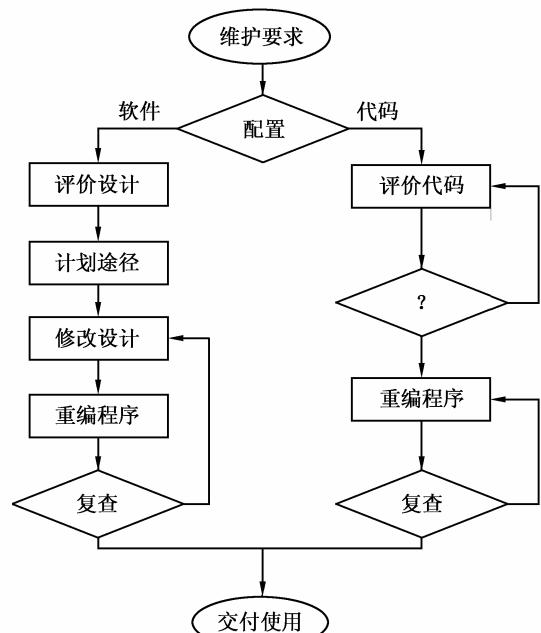


图 6.1 结构化维护与非结构化维护的对比

刚才描述的事件构成结构化维护，它是在软件开发的早期应用软件工程方法学的结果。虽然有了软件的完整配置并不能保证维护中没有问题，但是确实能减少精力的浪费并且能提高维护的总体质量。

6.2.2 维护的代价高昂

在过去的几十年中，软件维护的费用稳步上升。1970 年用于维护已有软件的费用只占软件总预算的 35% ~ 40%，1980 年上升为 40% ~ 60%，1990 年上升为 70% ~ 80%。

维护费用只不过是软件维护的最明显的代价，其他一些现在还不明显的代价将来可能更为人们所关注。

因为可用的资源必须供维护任务使用，以致耽误甚至丧失了开发新软件的良机，这是软件维护的一个无形的代价。其他无形的代价还有：

- ? 当看来合理的有关改错或修改的要求不能及时满足时将引起用户不满；
- ? 由于维护时的改动，在软件中引入了潜伏的故障，从而降低了软件的质量；
- ? 当必须把软件工程师调去从事维护工作时，将在开发过程中造成混乱。

软件维护的最后一个代价是生产率的大幅度下降，这种情况在维护旧程序时常常遇到。例如，据 Gausler 在 1976 年的报道，美国空军的飞行控制软件每条指令的开发成本是 75 美元，然而维护成本大约是每条指令 4000 美元，也就是说，生产率下降了 50 倍以上。

用于维护工作的劳动可以分成生产性活动（例如，分析评价，修改设计和编写程序代码等）和非生产性活动（例如，理解程序代码的功能，解释数据结构、接口特点和性能限度等）。下述表达式给出维护工作量的一个模型：

$$M = P + K \times \exp(c - d)$$

其中

M 是维护用的总工作量， P 是生产性工作量， K 是经验常数， c 是复杂程度（非结构化设计和缺少文档都会增加软件的复杂程度）， d 是维护人员对软件的熟悉程度。

上面的模型表明，如果软件的开发途径不好（即，没有使用软件工程方法学），而且原来的开发人员不能参加维护工作，那么维护工作量（和费用）将指数地增加。

6.2.3 维护的问题很多

与软件维护有关的绝大多数问题，都可归因于软件定义和软件开发的方法有缺点。在软件生命周期的头两个时期没有严格而又科学的管理和规划，几乎必然会导致在最后阶段出现问题。下面列出和软件维护有关的部分问题：

- ? 理解别人写的程序通常非常困难，而且困难程度随着软件配置成分的减少而迅速增加。如果仅有程序代码没有说明文档，则会出现严重的问题。
- ? 需要维护的软件往往没有合格的文档，或者文档资料显著不足。认识到软件必须有文档仅仅是第一步，容易理解的并且和程序代码完全一致的文档才真正有价值。
- ? 当要求对软件进行维护时，不能指望由开发人员给我们仔细说明软件。由于维护阶段持续的时间很长，因此，当需要解释软件时，往往原来写程序的人已经不在附近了。
- ? 绝大多数软件在设计时没有考虑将来的修改。除非使用强调模块独立原理的设计方法学，否则修改软件既困难又容易发生差错。

? 软件维护不是一项吸引人的工作。形成这种观念很大程度上是因为维护工作经常遭受挫折。

上述种种问题在现有的没采用软件工程思想开发出来的软件中，都或多或少地存在着。不应该把一种科学的方法学看做万应灵药，但是，软件工程至少部分地解决了与维护有关的每一个问题。

6.3 软件维护过程

维护过程本质上是修改和压缩了的软件定义和开发过程，而且事实上远在提出一项维护要求之前，与软件维护有关的工作已经开始了。首先必须建立一个维护组织，随后必须确定报告和评价的过程，而且必须为每个维护要求规定一个标准化的事件序列。此外，还应该建立一个适用于维护活动的记录保管过程，并且规定复审标准。

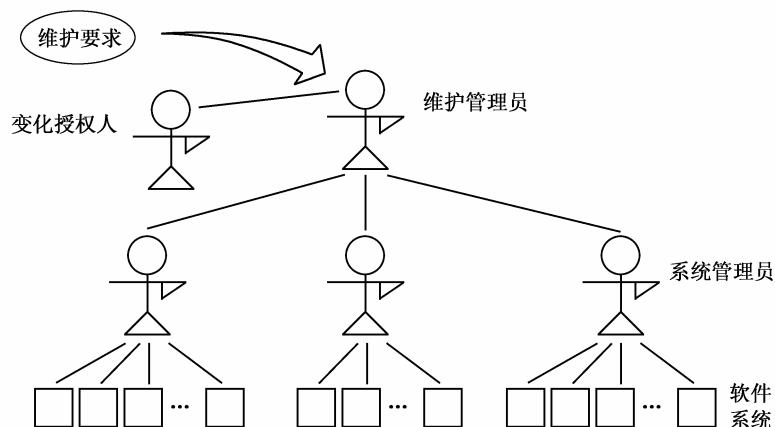


图 6.2 维护组织

6.3.1 维护组织

虽然通常并不需要建立正式的维护组织，但是，即使对于一个小的软件开发团体而言，非正式地委托责任也是绝对必要的。每个维护要求都通过维护管理员转交给相应的系统管理员去评价。系统管理员是被指定去熟悉一小部分产品程序的技术人员。系统管理员对维护任务做出评价之后，由变化授权人决定应该进行的活动。图 6.2 描绘了上述组织方式。

在维护活动开始之前就明确维护责任是十分必要的，这样做可以大大减少维护过程中可能出现的混乱。

6.3.2 维护报告

应该用标准化的格式表达所有软件维护要求。软件维护人员通常给用户提供空白的维护要求表——有时称为软件问题报告表，这个表格由要求一项维护活动的用户填写。如果遇到

了一个错误，那么必须完整描述导致出现错误的环境（包括输入数据，全部输出数据，以及其他有关信息）。对于适应性或完善性的维护要求，应该提出一个简短的需求说明书。如前所述，由维护管理员和系统管理员评价用户提交的维护要求表。

维护要求表是一个外部产生的文件，它是计划维护活动的基础。软件组织内部应该制定出一个软件修改报告，它给出下述信息：

- (1) 满足维护要求表中提出的要求所需要的工作量；
- (2) 维护要求的性质；
- (3) 这项要求的优先次序；
- (4) 与修改有关的事后数据。

在拟定进一步的维护计划之前，把软件修改报告提交给变化授权人审查批准。

6.3.3 维护的事件流

图 6.3 描绘了由一项维护要求而引出的一串事件。首先应该确定要求进行的维护的类型。用户常常把一项要求看作是为了改正软件的错误（改正性维护），而开发人员可能把同一项要求看作是适应性或完善性维护。当存在不同意见时必须协商解决。

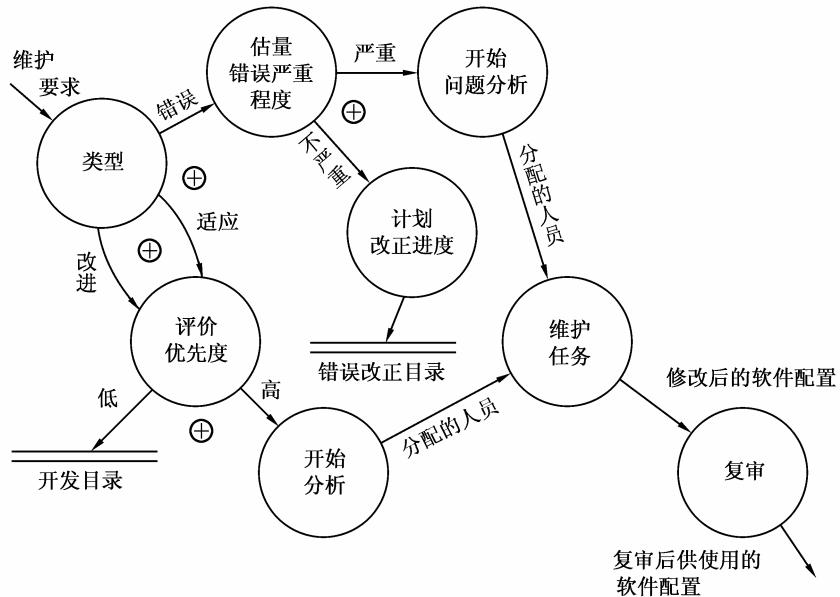


图 6.3 维护阶段的事件流

从图 6.3 描述的事件流看到，对一项改正性维护要求（图中“错误”通路）的处理，从估量错误的严重程度开始。如果是一个严重的错误（例如，一个关键性的系统不能正常运行），则在系统管理员的指导下分派人员，并且立即开始问题分析过程。如果错误并不严重，那么改正性的维护和其他要求软件开发资源的任务一起统筹安排。

适应性维护和完善性的维护的要求沿着相同的事件流通路前进。应该确定每个维护要求的优先次序，并且安排要求的工作时间，就好像它是另一个开发任务一样（从所有意图和目标

来看，它都属于开发工作）。如果一项维护要求的优先次序非常高，可能立即开始维护工作。

不管维护类型如何，都需要进行同样的技术工作。这些工作包括修改软件设计、复查、必要的代码修改、单元测试和集成测试（包括使用以前的测试方案的回归测试），验收测试和复审。不同类型的维护强调的重点不同，但是基本途径是相同的。维护事件流中最后一个事件是复审，它再次检验软件配置的所有成分的有效性，并且保证事实上满足了维护要求表中的要求。

当然，也有并不完全符合上述事件流的维护要求。当发生恶性的软件问题时，就出现所谓的“救火”维护要求，这种情况需要立即把资源用来解决问题。如果对一个组织来说，“救火”是常见的过程，那么必须怀疑它的管理能力和技术能力。

在完成软件维护任务之后，进行处境复查常常是有好处的。一般说来，这种复查试图回答下述问题：

- ? 在当前处境下设计、编码或测试的哪些方面能用不同方法进行？
- ? 哪些维护资源是应该有而事实上却没有的？
- ? 对于这项维护工作什么是主要的（以及次要的）障碍？
- ? 要求的维护类型中有预防性维护吗？

处境复查对将来维护工作的进行有重要影响，而且所提供的反馈信息对有效地管理软件组织十分重要。

6.3.4 保存维护记录

对于软件生命周期的所有阶段而言，以前记录保存都是不充分的，而软件维护则根本没有记录保存下来。由于这个原因，我们往往不能估价维护技术的有效性，不能确定一个产品程序的“优良”程度，而且很难确定维护的实际代价是什么。

保存维护记录遇到的第一个问题就是，哪些数据是值得记录的？Swanson 提出应记录下述内容：

- | | |
|--------------------|---------------------|
| (1) 程序标识； | (2) 源语句数； |
| (3) 机器指令条数； | (4) 使用的程序设计语言； |
| (5) 程序安装的日期； | (6) 自从安装以来程序运行的次数； |
| (7) 自从安装以来程序失效的次数； | (8) 程序变动的层次和标识； |
| (9) 因程序变动而增加的源语句数； | (10) 因程序变动而删除的源语句数； |
| (11) 每个改动耗费的人时数； | (12) 程序改动的日期； |
| (13) 软件工程师的名字； | (14) 维护要求表的标识； |
| (15) 维护类型； | (16) 维护开始和完成的日期； |
| (17) 累计用于维护的人时数； | (18) 与完成的维护相联系的纯效益。 |

应该为每项维护工作都收集上述数据。可以利用这些数据构成一个维护数据库的基础，并且像下一小节介绍的那样对它们进行评价。

6.3.5 评价维护活动

缺乏有效的数据就无法评价维护活动。如果已经开始保存维护记录了，则可以对维护工作做一些定量度量。至少可以从下述 7 个方面度量维护工作：

- (1) 每次程序运行平均失效的次数；
- (2) 用于每一类维护活动的总人时数；
- (3) 平均每个程序、每种语言、每种维护类型所做的程序变动数；
- (4) 维护过程中增加或删除一个源语句平均花费的人时数；
- (5) 维护每种语言平均花费的人时数；
- (6) 一张维护要求表的平均周转时间；
- (7) 不同维护类型所占的百分比。

根据对维护工作定量度量的结果，可以做出关于开发技术、语言选择、维护工作量规划、资源分配及其他许多方面的决定，而且可以利用这样的数据去分析评价维护任务。

6.4 软件的可维护性

软件可维护性可以定性地定义为：维护人员理解、改正、改动和改进这个软件的难易程度。我们一直强调，提高可维护性是支配软件工程方法学所有步骤的关键目标。

6.4.1 决定软件可维护性的因素

维护就是在软件交付使用后进行的修改，修改之前必须理解修改的对象，修改之后应该进行必要的测试，以保证所做的修改是正确的。如果是改正性维护，还必须预先进行调试以确定错误。因此，影响软件可维护性的因素主要有下述五个：

1. 可理解性

软件可理解性表现为外来读者理解软件的结构、接口、功能和内部过程的难易程度。模块化和模块独立、与源程序完全一致的完整正确详细的文档、结构化设计或面向对象设计、源程序内部的文档和良好的程序设计语言等等，都对提高软件的可理解性有重要贡献。

2. 可测试性

诊断和测试的难易程度主要取决于软件容易理解的程度。良好的文档对诊断和测试是至关重要的。此外，软件结构、可用的测试工具和调试工具，以及以前设计的测试过程也都是非常重要的。维护人员应该能够得到在开发阶段用过的测试方案，以便进行回归测试。在设计阶段应该尽力把软件设计成容易测试和容易诊断的。

3. 可修改性

软件容易修改的程度和本书第3章及第5章讲述的设计原理和启发规则直接有关。模块化、耦合、内聚、信息隐藏等等都对软件的可修改性有较大影响。

4. 可移植性

把程序从一种运行环境转移到另一种运行环境时，需要的工作量的多少即是软件的可移植性。显然，软件的可移植性直接决定了适应性维护的难易程度，对完善性维护的难易程度也有一定影响。

5. 可重用性

软件的可重用性是指同一个软件（或软件成分）不做修改或稍加改动，就可以在不同场

合多次重复使用。显然，维护一个可重用性好的软件时，需要对软件做的修改自然就比较少。用面向对象方法学开发出的软件，由于具有封装、继承、多态等机制，可重用性明显好于用传统方法学开发出的软件，因此可维护性好。

可移植性和可重用性本质上也是可修改性，是特殊条件下的可修改性。

上述五个可维护性因素是密切相关的。维护人员在正确理解一个程序之前根本不可能正确地修改它；如果不能进行完善的诊断和测试，则表面正确的修改可能引进其他错误。

6.4.2 文档

文档是影响软件可维护性的决定因素。由于长期使用的大型软件系统在使用过程中必然会经受多次修改，所以文档比程序代码更重要。

软件系统的文档可以分为用户文档和系统文档两类。用户文档主要描述系统功能和使用方法，并不关心这些功能是怎样实现的；系统文档描述系统设计、实现和测试等各方面的内容。

总的说来，软件文档应该满足下述要求：

- (1) 必须描述如何使用这个系统，没有这种描述即使是最简单的系统也无法使用；
- (2) 必须描述怎样安装和管理这个系统；
- (3) 必须描述系统需求和设计；
- (4) 必须描述系统的实现和测试，以便使系统成为可维护的。

下面分别讨论用户文档和系统文档：

1. 用户文档

用户文档是用户了解系统的第一步，它应该能使用户获得对系统的准确的初步印象。文档的结构方式应该使用户能够方便地根据需要阅读有关的内容。

用户文档至少应该包括下述五方面的内容：

- (1) 功能描述——说明系统能做什么；
- (2) 安装文档——说明怎样安装这个系统以及怎样使系统适应特定的硬件配置；
- (3) 使用手册——简要说明如何着手使用这个系统（应该通过丰富例子说明怎样使用常用的系统功能，还应该说明用户操作错误时怎样恢复和重新启动）；
- (4) 参考手册——详尽描述用户可以使用的所有系统设施以及它们的使用方法，还应该解释系统可能产生的各种出错信息的含义（对参考手册最主要的要求是完整，因此通常使用形式化的描述技术）；
- (5) 操作员指南（如果有系统操作员的话）——说明操作员应该如何处理使用中出现的各种情况。

上述内容可以分别作为独立的文档，也可以作为一个文档的不同分册，具体做法应该由系统规模决定。

2. 系统文档

所谓系统文档指从问题定义、需求说明到验收测试计划这样一系列和系统实现有关的文档。描述系统设计、实现和测试的文档对于理解程序和维护程序来说是极端重要的。和用户文档类似，系统文档的结构也应该能把读者从对系统概貌的了解，引导到对系统各个方面每

个特点的更形式化更具体的认识。

6.4.3 可维护性复审

可维护性是所有软件都应该具备的基本特点，必须在开发阶段保证软件具有6.4.1节中提到的那些可维护因素。在软件工程过程的每一个阶段都应该考虑并努力提高软件的可维护性，在每个阶段结束前的技术审查和管理复审中，应该着重对可维护性进行复审。

在需求分析阶段的复审过程中，应该对将来要改进的部分和可能会修改的部分加以注意并指明；应该讨论软件的可移植性问题，并且考虑可能影响软件维护的系统界面。

在正式的和非正式的设计复审期间，应该从容易修改、模块化和功能独立的目标出发，评价软件的结构和过程；设计中应该对将来可能修改的部分预作准备。

代码复审应该强调编码风格和内部说明文档这两个影响可维护性的因素。

每个测试步骤都可以暗示在软件正式交付使用前，程序中可能需要做预防性维护的部分。在测试结束时进行最正式的可维护性复审，这个复审称为配置复审。配置复审的目的是保证软件配置的所有成分是完整的、一致的和可理解的，而且为了便于修改和管理已经编目归档了。

在完成了每项维护工作之后，都应该对软件维护本身进行仔细认真的复审。

维护应该针对整个软件配置，不应该只修改源程序代码。当对源程序代码的修改没有反映在设计文档或用户手册中时，就会产生严重的后果。

每当对数据、软件结构、模块过程或任何其他有关的软件特点做了改动时，必须立即修改相应的技术文档。不能准确反映软件当前状态的设计文档可能比完全没有文档更坏。在以后的维护工作中很可能因文档不完全符合实际而不能正确理解软件，从而在维护中引入过多的错误。

用户通常根据描述软件特点和使用方法的用户文档来使用、评价软件。如果对软件的可执行部分的修改没有及时反映在用户文档中，则必然会使用户因为受挫折而产生不满。

如果在软件再次交付使用之前，对软件配置进行严格的复审，则可大大减少文档的问题。事实上，某些维护要求可能并不需要修改软件设计或源程序代码，只是表明用户文档不清楚或不准确，因此只需要对文档做必要的维护。

为了从根本上提高软件的可维护性，人们试图利用程序自动生成技术，通过直接维护软件规格说明书来维护软件，同时也在大力发展软件重用技术。

6.5 预防性维护

预防性维护也称为软件再工程。目前，在全部软件维护活动中，预防性维护只占很小的比例。多数软件维护人员对预防性维护还缺乏足够的了解。本节说明进行预防性维护的必要性和可行性，下一节讲述软件再工程的典型过程。

6.5.1 必要性

在现有的正在运行使用的软件中，下述类型的系统并不是极个别的：它们已经为某单位

的业务需要服务了十几年，在漫长的运行使用过程中，为了改正发现的错误，或为了适应变化了的环境，或为了扩充功能，人们曾多次修改它们。尽管负责修改这些老系统的人具有十分良好的愿望，但是，这些系统在开发时就没有遵循正规的软件工程方法学，修改时迫于其他事情的压力又把良好的软件工程习惯抛在脑后。现在，这些系统是不稳定的，它虽然还在继续工作，但是每次修改后都会产生无法预料的、严重的副作用。然而，在今后的运行过程中，用户仍然会对这些应用系统不断提出维护要求，维护人员应该怎样做？

如果用程序图描绘上述这类老系统的控制流程，那么，代表控制流的箭头线将穿来穿去地绕成一团乱麻，也有人把这样的控制流形容成一碗面条。这样的老系统虽然在开发时可能已经分解成若干个模块，但是某些特别大的“模块”甚至包含数千条语句，而且在长达几十万条的源程序语句中只有两三条有意义的注释，又没有其他文档资料。为了修改这类程序以适应用户变更了的需求，维护人员只有下列几种可能的选择：

(1) 努力地尝试各种不同的修改方法，与隐含的设计和源代码不断地“战斗”，以实现所要求的变更；

(2) 尽可能多地了解程序的内部工作细节，努力使修改工作更有效；

(3) 重新设计、重新编码并测试该程序需要修改的部分，把正确的软件工程方法学应用于所有被修改的部分；

(4) 把整个程序全部重新设计、重新编码和测试，为此可以使用软件再工程工具辅助我们理解原有的设计。

上述第1种选择是盲目的，用这种“瞎猫碰着死耗子”的方法即使能侥幸地实现所要求的修改，也往往会在程序中又引入几个新错误。

第2、3、4这三种选择属于软件再工程范畴（参见6.6节）。维护组织不应该坐等维护请求到来再被动地进行维护（这样做可能只有第1种选择是可行的），应该按照下述标准挑选一个程序，应用上述的第2、3或4种选择，主动地进行预防性维护：

(1) 该程序将在今后数年内继续使用；

(2) 当前正在成功地使用着该程序；

(3) 可能在最近的将来要对该程序做较大程度的修改或扩充。

预防性维护方法是Miller在“结构化革新”的标题下提出来的，他把这个概念定义为“把今天的方法学应用到昨天的系统以支持明天的需求”。

6.5.2 可行性

初看起来，在一个正在工作的程序版本已经存在的情况下，重新开发这个大型程序似乎是一种浪费，但是，考虑到下述事实预防性维护实际上是可行的：

(1) 维护一行源代码的成本可能是该行代码初始开发成本的20~40倍；

(2) 使用现代设计概念重新设计软件体系结构（程序结构和数据结构），对未来的维护工作将有很大帮助；

(3) 由于软件原型（即现在正在工作的程序）已经存在，软件开发生产率将远远高于平均水平；

(4) 现在用户已经有较丰富的使用该软件的经验，因此，很容易确定新的需求和变更方向；

(5) 利用软件再工程工具可以自动完成部分工作；

(6) 在完成预防性维护的过程中，可以建立起完整的软件配置（文档、程序和数据）。

当软件开发组织把软件作为产品销售时，在程序的“新版本”中往往体现了预防性维护的成果。一个大型的软件开发机构可能拥有 500~2000 个产品程序，可以根据重要性把这些程序排出优先次序，然后把它们作为预防性维护的候选者加以评估。

6.6 软件再工程过程

再工程是一种重新构建已有产品的活动。通常，在再工程过程中遵守下述原则：

? 按照预定的标准检查原有的产品，以确定它是否确实需要重建；

? 检查原有产品的结构，如果结构良好则只需改造不需重建，改造的成本远低于重建的成本；

? 在开始重建之前先了解原有产品的内部工作原理，这对重建工作是有一定帮助的；

? 开始重建工作之后，只使用最现代的、最牢固耐用的材料，这样做虽然成本高一些，但是却有助于避免以后昂贵、耗时的维护；

? 如果决定重建，一定要对重建工作严格管理，并且使用能开发出高质量产品的先进技术。

为了落实上述这些原则，建议应用如图 6.4 所示的软件再工程过程模型，它定义了 6 类活动。在某些情况下，这些活动按照图中所示次序以线性顺序进行，但是并不总是这样，例如，可能在文档重构开始之前，需要先进行逆向工程以理解程序的内部工作原理。

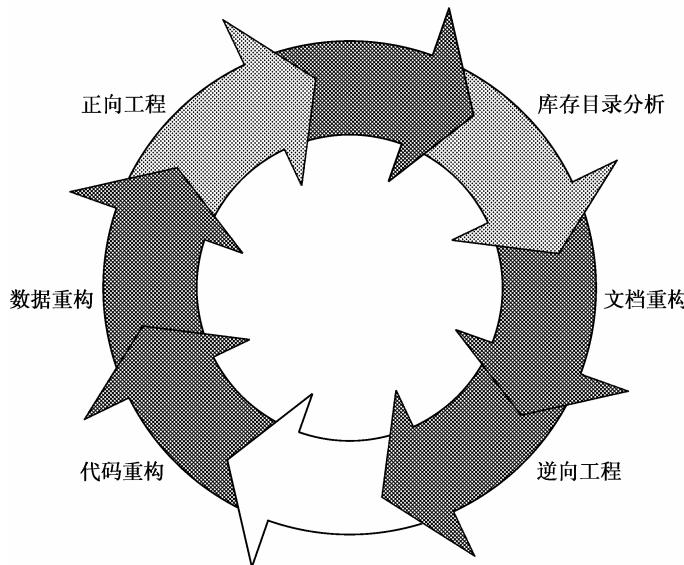


图 6.4 软件再工程过程模型

在图 6.4 中描绘的软件再工程范型是一个循环模型，这意味着作为该范型组成部分的每个活动都可能重复进行，而且对于某个特定的循环来说，过程可以在完成任意一个活动

之后终止。

下面介绍软件再工程过程模型中的每个活动。

1. 库存目录分析

每个软件组织都应该保存其负责维护的所有应用系统的库存目录。该目录可能仅仅是包含下列信息的一个电子表格模型：

- ? 应用系统的名字；
- ? 最初构建它的年份；
- ? 已对它进行过的实质性修改的次数；
- ? 完成这些修改所花费的总工作量；
- ? 最后一次实质性修改的日期；
- ? 最后一次实质性修改所花费的工作量；
- ? 它驻留的系统；
- ? 和它有接口的应用系统；
- ? 它访问的数据库；
- ? 过去 18 个月所报告的错误；
- ? 用户数量；
- ? 安装此应用系统的机器数量；
- ? 程序结构复杂程度、代码复杂程度和文档复杂程度；
- ? 文档的质量；
- ? 整体可维护性（用等级值表示）；
- ? 预期寿命（以年计）；
- ? 在未来 36 个月内的预期修改次数；
- ? 年度维护成本；
- ? 年度运行成本；
- ? 年度业务值；
- ? 业务重要程度。

应该针对每一个现有的应用系统收集上面列出的信息。通过按照业务重要程度、寿命、当前可维护性以及其他重要标准对这些信息排序的办法，可以选出软件再工程的候选者，然后可以明智地为再工程分配资源。

必须注意，应该定期地修订刚才描述的库存目录表，应用系统的状况（例如，业务重要程度）可能随着时间而改变，因此，再工程的优先级也将发生变化。

2. 文档重构

缺乏文档或文档严重不合格，是很多老系统的通病。面对这种状况，有下述三种做法可供选择：

（1）选择 1

建立文档是非常耗费时间的，既然系统能正常工作，我们就让它保持现状好了。在某些情况下，这是一个正确的做法。事实上，不可能为数百个计算机程序都重新建立文档。如果一个程序是相对静止的，正在走向其有用的终点，并且可能不会再经历什么变化，那么，让它保持现状确实是一个明智的选择。

(2) 选择 2

文档必须被更新，但是，我们只有有限的资源，因此我们将采用“在使用时建文档”的方法。也就是说，不是一次性地为某个应用系统全部重建文档，而是只为系统中当前正在修改的那些部分建立完整的文档。随着时间的流逝，我们将得到一组有用的文档。

(3) 选择 3

该系统对于完成业务工作来说是至关重要的，必须完全地重构文档。即使在这种情况下，明智的办法也是尽量把文档工作减少到必需的最小量。

上述每个选择都是可行的，软件组织必须选用最适合于实际情况的方法。

3. 逆向工程

术语“逆向工程”来源于硬件领域：一家公司通过解剖其竞争对手的硬件产品，力图了解竞争者在设计和生产方面的“秘密”。当然，如果得到了竞争者的设计和生产规格说明书，则很容易了解到这些秘密。但是，这些文档是别人的机密文件，做逆向工程的公司不可能得到它们。事实上，成功的逆向工程通过研究产品的实际样品而导出一个或多个关于产品设计和生产的规格说明书。

软件的逆向工程与硬件的逆向工程是相当类似的。但是，在绝大多数情况下，被逆向工程的程序不是来自于竞争对手，而是本公司自己的产品（通常是多年以前开发出来的）。需要了解的“秘密”是程序的内部工作原理，由于缺乏相关的文档资料，目前在公司内还没有人清楚地知道该程序的内部工作原理。因此，软件的逆向工程就是分析程序以便在比源代码更高的抽象层次上创建出该程序的某种表示的过程。逆向工程是一个恢复设计结果的过程，逆向工程工具从现存的程序中提取数据、体系结构和处理过程的设计信息。

4. 代码重构

最常见的一种再工程（实际上，在这里使用术语“再工程”可能并不十分恰当），是代码重构。某些老系统具有相对完整的程序体系结构，但是，编写某些模块代码的方法不正确，使得这些模块难于理解、测试和维护。在这种情况下，可以重构有问题的模块的代码。

为了完成代码重构工作，首先用重构工具去分析源代码，标注出和结构化程序设计概念相违背的部分，然后重构这些代码（此项工作可使用软件工具自动进行）。接下来复审和测试所得到的重构代码，以保证没有引入新的错误。最后，更新内部的代码文档。

通常，重构并不修改程序的整体体系结构，它着重关注个体模块的设计细节以及在模块内定义的局部数据结构。如果重构工作扩展到模块边界之外并涉及到程序体系结构，则重构变成了正向工程。

5. 数据重构

数据体系结构不好的程序，难于进行适应性修改和扩充。事实上，对于许多应用系统来说，数据体系结构比源代码本身对程序的长期生存力有更大的影响。

与代码重构不同，数据重构发生在比代码重构更高的抽象层次上，它是一种全范围的再工程活动。在绝大多数情况下，数据重构从逆向工程活动开始，仔细分析当前的数据体系结构。必要时定义数据模型，标识数据对象和属性，并且复审现存数据结构的质量。

当数据结构较差时（例如，在采用关系型方法可以大大简化处理的情况下，当前却使用了平坦文件），对数据进行再工程。

由于数据体系结构对程序体系结构及程序中的算法有很大影响，对数据的修改必然会导致

致程序体系结构或代码层的改变。

6. 正向工程

正向工程也称为更新或再造，它不仅从现存软件中提取设计信息，而且使用这些信息去修改或重建现存系统，以便提高系统的整体质量。

正向工程过程应用现代的软件工程概念、原理和方法来重新开发现存的某个应用系统。在大多数情况下，经过正向工程过程得到的软件，不仅重新实现了现有系统的功能，而且增加了新功能和（或）提高了整体性能。

6.7 小结

维护是软件生命周期的最后一个阶段，也是持续时间最长代价最大的一个阶段。软件工程学的主要目的就是提高软件的可维护性，降低维护的代价。

软件维护通常包括四类活动：为了纠正正在使用过程中暴露出来的错误而进行的改正性维护；为了适应外部环境的变化而进行的适应性维护；为了改进原有的软件而进行的完善性维护；以及为了改进将来的可维护性和可靠性而进行的预防性维护。

软件的可理解性、可测试性、可修改性、可移植性和可重用性是决定软件可维护性的基本因素。软件生命周期每个阶段的工作都和软件可维护性有密切关系。良好的设计，完善的文档资料，以及一系列严格的复审和测试，使得一旦发现错误时比较容易诊断和纠正；当用户有新要求或者外部环境变化时软件能较容易地适应，并且能够减少维护引入的错误。因此，在软件生命周期的每个阶段都必须充分考虑维护问题，并且为软件维护预作准备。

文档是影响软件可维护性的决定因素，因此，文档甚至比可执行的程序代码更重要。文档可分为用户文档和系统文档两大类。不管是哪一类文档都必须和程序代码同时维护，只有和程序代码完全一致的文档才是真正有价值的文档。

软件重用技术是能从根本上提高软件可维护性的重要技术，而本书第5章讲述的面向对象的软件技术是最成功的软件重用技术。

预防性维护也称为软件再工程。虽然目前在全部维护活动中预防性维护只占很小的比例，但是我们不应该忽视这类维护活动的必要性和可行性。

软件再工程包括下述的一系列活动：库存目录分析，文档重构，逆向工程，程序和数据重构以及正向工程。这些活动的目的是，创建出现存程序的质量更高和可维护性更好的版本。在大多数情况下，经过再工程所创建出的程序新版本，不仅实现了原有程序的功能，而且增加了新功能并提高了整体性能。

习题六

1. 某些软件工程师不同意“目前国外许多软件开发组织把60%以上的人力用于维护已有的软件”的说法，他们争论说：“我并没有花费我的60%的时间去改正我所开发的程序中的错误”。

请问，你对上述争论有何看法？

2. 软件的可维护性与哪些因素有关？在软件开发过程中应该采取哪些措施来提高软件产品的可维护性？
3. 为什么说用面向对象方法学开发出的软件比用传统方法学开发出的软件的可维护性好？
4. 采用什么策略可以降低改正性维护、适应性维护和完善性维护的成本？
5. 在什么情况下应该主动地进行预防性维护？
6. 假设你的任务是对一个已有的软件做重大修改，而且只允许你从下述文档中选取两份：
(a) 程序的规格说明；(b) 程序的详细设计结果（自然语言描述加上某种设计工具表示）；(c) 源程序清单（其中有适当数量的注解）。
你将选取哪两份文档？为什么这样选取？你打算怎样完成交给你的任务？
7. 分析预测在下列系统交付使用以后，用户可能提出哪些改进或扩充功能的要求。如果由你来开发这些系统，你在设计和实现时将采取哪些措施，以方便将来的修改？
 - (1) 储蓄系统（参见习题二第3题）；
 - (2) 机票预订系统（参见习题二第4题）；
 - (3) 患者监护系统（参见习题二第5题）。
8. 重构与正向工程有何相同之处？有何不同之处？

第7章 软件项目管理

所谓管理就是通过计划、组织和控制等一系列活动，合理地配置和使用各种资源，以达到既定目标的过程。

软件项目管理在任何技术活动开始之前就已经开始了，并且贯穿于软件的整个生命周期。目前，虽然好的管理不一定能确保工程百分之百地成功，但是坏的管理或不适当的管理技术是一定会导致工程失败——软件交付使用的日期将大大拖后，实际成本比预期成本高很多，而且最终得到的软件产品质量低劣、很难维护。

7.1 度量软件规模

软件项目管理过程从一组项目计划活动开始，而第一项计划活动就是“估算”。由于估算 是所有其他项目计划活动的基础，而项目计划为软件工程指出了通往成功的道路，因此，必须充分重视估算活动。

工作量估算和完成期限估算 是项目计划的基础。为了估算软件项目的工作量和完成期限，首先需要度量软件的规模。下面介绍两种常用的估算软件规模的方法。

7.1.1 代码行技术

代码行技术是比较简单的定量估算软件规模的方法。这种方法根据过去开发类似软件产品的经验和历史数据，估计实现一个功能需要的源程序行数。把实现每个功能需要的源程序行数累加起来，就可得到实现整个软件需要的源程序行数。

为了使得对程序规模的估计值尽可能接近实际值，可以由多位有经验的软件工程师分别独立地作出估计。每个人都估计程序的最小规模(a)、最大规模(b)和最可能的规模(m)，分别算出这三类规模的平均值 \bar{a} 、 \bar{b} 和 \bar{m} 之后，再用下式计算程序规模的估计值：

$$L \approx \frac{\bar{a} + 4\bar{m} + \bar{b}}{6} \quad (7.1)$$

用代码行技术度量软件规模时，当程序较小时常用的单位是代码行数(LOC)，当程序较大时常用的单位是千行代码数(KLOC)。

1. 代码行技术的优点

- 代码行是所有软件开发项目都有的“产品”，而且很容易计算；

- ? 许多现有的软件估算模型使用 LOC 或 KLOC 作为关键的输入数据；
- ? 已有大量基于代码行的文献和数据存在。
- 2. 代码行技术的缺点
 - ? 源程序仅是软件配置的一个成分，用它的规模代表整个软件的规模似乎不太合理；
 - ? 用不同语言实现同一个软件产品所需要的代码行数并不相同；
 - ? 这种方法不适用于非过程语言。

7.1.2 功能点技术

功能点技术依据对软件信息域特性和软件复杂性的评估结果，估算软件规模。这种方法用功能点 (FP) 为单位，度量软件的规模。

1. 信息域特性

功能点技术定义了信息域的 5 个特性，分别是输入项数 (Inp)、输出项数 (Out)、查询数 (Inq)、主文件数 (Maf) 和外部接口数 (Inf)。下面讲述这 5 个特性的含义。

输入项数：用户向软件输入的项数，这些输入给软件提供面向应用的数据。输入不同于查询，查询另外计数，不计入输入项数中。

输出项数：软件向用户输出的项数，它们向用户提供面向应用的信息，例如，报表、屏幕和出错信息等。报表内的数据项不单独计数。

查询数：所谓查询是一次联机输入，它导致软件以联机输出方式产生某种即时响应。

主文件数：逻辑主文件（即数据的一个逻辑组合，它可能是某个大型数据库的一部分或是一个独立的文件）的数目。

外部接口数：机器可读的全部接口（例如，磁带或磁盘上的数据文件）的数量，用这些接口把信息传送给另一个系统。

2. 估算功能点的步骤

用下述三个步骤，可以估算出一个软件的功能点数（即软件规模）。

(1) 计算未调整的功能点数 UFP

首先，把产品信息域的每个特性（即 Inp、Out、Inq、Maf 和 Inf）都分类成简单级、平均级或复杂级。根据其等级，为每个特性都分配一个功能点数，例如，一个平均级的输入项分配 4 个功能点，一个简单级的输入项是 3 个功能点，而一个复杂的输入项分配 6 个功能点。

然后，用下式计算未调整的功能点数 UFP

$UFP = a_1 \times Inp + a_2 \times Out + a_3 \times Inq + a_4 \times Maf + a_5 \times Inf$ 其中， a_i ($1 \leq i \leq 5$) 是信息域特性系数，其值由相应特性的复杂级别决定，如表 7.1 所示。

表 7.1 信息域特性系数值

特性系数 \ 复杂级别	简 单	平 均	复 杂
输入系数 a_1	3	4	6
输出系数 a_2	4	5	7
查询系数 a_3	3	4	6
文件系数 a_4	7	10	15
接口系数 a_5	5	7	10

(2) 计算技术复杂性因子 TCF

这一步将度量 14 种技术因素对软件规模的影响程度。这些因素包括高处理率、性能标准（例如，响应时间）、联机更新等，在表 7.2 中列出了全部技术因素，并用 F_i ($i = 1 \sim 14$) 代表这些因素。根据软件特点，为每个因素分配一个从 0（不存在或对软件规模无影响）到 5（有很大影响）的值。然后，用下式计算技术因素对软件规模的综合影响程度 DI：

$$DI = \sum_{i=1}^{14} F_i$$

技术复杂性因子 TCF 由下式计算：

$$TCF = 0.65 + 0.01 \times DI$$

因为 DI 的值在 0 ~ 70 之间，所以 TCF 的值在 0.65 ~ 1.35 之间。

表 7.2 技术因素

序号	F_i	技术因素
1	F_1	数据通信
2	F_2	分布式数据处理
3	F_3	性能标准
4	F_4	高负荷的硬件
5	F_5	高处理率
6	F_6	联机数据输入
7	F_7	终端用户效率
8	F_8	联机更新
9	F_9	复杂的计算
10	F_{10}	可重用性
11	F_{11}	安装方便
12	F_{12}	操作方便
13	F_{13}	可移植性
14	F_{14}	可维护性

(3) 计算功能点数 FP

功能点数 FP 由下式计算：

$$FP = UFP \times TCF$$

功能点数与所用的编程语言无关，因此，功能点技术比代码行技术更合理一些。但是，在判断信息域特性复杂级别及技术因素的影响程度时，存在相当大的主观因素。

7.2 估算软件开发工作量

软件估算模型使用由经验数据导出的公式来预测软件开发的工作量，工作量是软件规模的函数，工作量的单位通常是人月（pm）。

支持大多数估算模型的经验数据，都是从有限个项目的样本集采集来的，因此，没有一个估算模型能够适用于所有类型的软件和开发环境。下面介绍三类典型的估算模型。

7.2.1 静态单变量模型

这类模型的总体结构形式如下：

$$E = A + B \times (ev)^C$$

其中，A、B 和 C 是由经验数据导出的常数，E 是以人月为单位的软件开发工作量，ev 是估算变量（即以 KLOC 或 FP 为单位的软件规模）。下面给出几个典型的静态单变量模型：

1. 面向 KLOC 的估算模型

(1) Walston-Felix 模型

$$E = 5.2 \times (\text{KLOC})^{0.91}$$

(2) Bailey-Basili 模型

$$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$$

(3) Boehm 简单模型

$$E = 3.2 \times (\text{KLOC})^{1.05}$$

(4) Doty 模型（在 KLOC > 9 的情况下）

$$E = 5.288 \times (\text{KLOC})^{1.047}$$

2. 面向 FP 的估算模型

(1) Albrecht & Gaffney 模型

$$E = -13.39 + 0.0545FP$$

(2) Kemerer 模型

$$E = 60.62 + 7.728 \times 10^{-8}FP^3$$

(3) Matson, Barnett 和 Mellichamp 模型

$$E = 585.7 + 15.12FP$$

从上面给出的估算模型可以看出，对于相同的 KLOC 或 FP 值，用不同模型估算得出的结果并不相同。出现这种现象的主要原因是，这些模型都是仅根据若干个应用领域内有限个项目的经验数据推导出来的，适用范围有限。因此，应该根据当前项目的特点选择适用的估算模型，并且根据实际情况适当地调整估算模型。

7.2.2 动态多变量模型

动态多变量模型也称为软件方程式，它是根据从 4000 多个当代软件项目中收集的生产率数据推导出来的。这种模型把软件开发工作量看作是软件规模和开发时间这两个变量的函数。

动态多变量估算模型的形式如下：

$$E = (\text{LOC} \times B^{0.333} / P)^3 \times (1 / t)^4 \quad (7.2)$$

其中，

E 是以人月或人年为单位的工作量；

t 是以月或年为单位的项目持续时间；

B 是“特殊技术因子”，它随着对集成、测试、质量保证、文档及管理技术的需求的增长而缓慢增加，对于较小的程序 ($\text{KLOC} = 5 \sim 15$)， $B = 0.16$ ，对于超过 70KLOC 的程序， $B = 0.39$ 。

P 是“生产率参数”，它反映了下述因素对工作量的影响：

? 总体的过程成熟度及管理水平；

? 使用良好的软件工程实践的程度；

? 使用的程序设计语言的级别；

? 软件环境的状态；

? 软件项目组的技术及经验；

? 应用系统的复杂程度。

当开发实时嵌入式软件时，典型值是 $P = 2000$ ；对于电信和系统软件来说， $P = 10000$ ；对于商业系统应用， $P = 28000$ 。适用于当前项目的生产率参数，可以从历史数据导出。

应该注意，软件方程式有两个独立的变量：对软件规模的估算值（用 LOC 表示）；以月或年为单位的项目持续时间。

从 (7.2) 式可以看出，开发同一个软件（即 LOC 固定）的时候，如果把项目持续时间延长一些，则可降低完成项目所需要的工作量。

7.2.3 COCOMO 2 模型

COCOMO 是构造性成本模型 (COSt MOdel) 的英文缩写。1981 年 B.W.Boehm 在《软件工程经济学》一书中首次提出了 COCOMO 模型，1997 年 Boehm 等人提出的 COCOMO 2 模型，是原始的 COCOMO 模型的修订版。它反映了这十多年来我们在成本估计方面所积累的经验和知识。

COCOMO 2 给出了三个层次的软件开发工作量估算模型，这三个层次的模型在估算工作量时，对软件细节考虑的详尽程度逐级增加。这些模型既可以用于不同类型的项目，也可以用于同一个项目的不同开发阶段。COCOMO 2 的三个层次的估算模型分别是：

? 应用系统组成模型

这个模型主要用于估算构建原型的工作量，模型名字暗示在构建原型时大量使用现有的构件。

? 早期设计模型

这个模型适用于体系结构设计阶段。

? 后体系结构模型

这个模型适用于软件产品的实际开发阶段。

下面以后体系结构模型为例，介绍 COCOMO 2 模型。该模型把软件开发工作量表示成代码行数 (KLOC) 的非线性函数：

$$E = a \times \text{KLOC}^b \times \sum_{i=1}^{17} f_i \quad (7.3)$$

其中，

E 是开发工作量（以人月为单位），

a 是模型系数，

KLOC 是估计的源代码行数（以千行为单位），

b 是模型指数，

f_i ($i = 1 \sim 17$) 是成本因素。

每个成本因素都根据它的重要程度和对工作量影响大小赋予一定数值（称为工作量系数）。这些成本因素对任何一个项目的开发工作量都有影响，即使不使用 COCOMO 2 模型估算工作量，也应该重视这些因素。Boehm 把成本因素划分成产品因素、平台因素、人员因素和项目因素等四类。

表 7.3 给出了 COCOMO 2 使用的成本因素及与之相联系的工作量系数。与原始的 COCOMO 模型相比，COCOMO 2 模型使用的成本因素有下述变化，这些变化反映了在过去十几年中软件行业取得的巨大进步：

表 7.3

成本因素及工作量系数

成本因素	级别					
	甚低	低	正常	高	甚高	特高
产品因素						
要求的可靠性	0.75	0.88	1.00	1.15	1.39	
数据库规模		0.93	1.00	1.09	1.19	
产品复杂程度	0.75	0.88	1.00	1.15	1.30	1.66
要求的可重用性		0.91	1.00	1.14	1.29	1.49
需要的文档量	0.89	0.95	1.00	1.06	1.13	
平台因素						
执行时间约束			1.00	1.11	1.31	1.67
主存约束			1.00	1.06	1.21	1.57
平台变动		0.87	1.00	1.15	1.30	
人员因素						
分析员能力	1.50	1.22	1.00	0.83	0.67	
程序员能力	1.37	1.16	1.00	0.87	0.74	
应用领域经验	1.22	1.10	1.00	0.89	0.81	
平台经验	1.24	1.10	1.00	0.92	0.84	
语言和工具经验	1.25	1.12	1.00	0.88	0.81	
人员连续性	1.24	1.10	1.00	0.92	0.84	
项目因素						
使用软件工具	1.24	1.12	1.00	0.86	0.72	
多地点开发	1.25	1.10	1.00	0.92	0.84	0.78
要求的开发进度	1.29	1.10	1.00	1.00	1.00	

? 增加了 4 个新成本因素，它们分别是要求的可重用性、需要的文档量、人员连续性（即人员稳定程度）和多地点开发。这个变化说明，这些因素对开发成本的影响日益增加。

? 略去了原始模型中原有的 2 个成本因素，它们分别是计算机切换时间和使用现代程序设计实践。现在，开发人员普遍使用工作站来开发软件，批处理的切换时间已经不再是问题。而“现代程序设计实践”已经发展成内容更广泛的“成熟的软件工程实践”的概念，并且在 COCOMO 2 工作量方程的指数 b 中考虑了这个因素的影响。

? 某些成本因素（分析员能力、平台经验、语言和工具经验）对生产率的影响（即工作量系数最大值与最小值的比率）增加了，另一些成本因素（程序员能力）的影响减小了。

为了确定工作量方程中模型指数 b 的数值，原始的 COCOMO 模型把软件开发项目划分成组织式、半独立式和嵌入式这样三种类型，并为每种项目类型指定 b 的数值（分别是 1.05，1.12 和 1.20）。COCOMO 2 模型与原始的 COCOMO 模型不同，它具有更加精细得多的 b 分级模型。这个模型使用 5 个分级因素 W_i ($i = 1 \dots 5$)，其中每一个因素都划分成从甚低（5）到特高（0）的 6 个级别。然后用下式计算工作量方程的指数 b 的数值：

$$b = 1.01 + 0.01 \sum_{i=1}^5 W_i \quad (7.4)$$

因此， b 的取值范围为 1.01 ~ 1.26。显然，这种分级模式比原始 COCOMO 模型的分级模式更精细更灵活。

COCOMO 2 中使用的 5 个分级因素如下所述：

? 项目先例性

这个分级因素指出，对于开发组织来说该项目的新奇程度。诸如开发类似系统的经验，需要创新体系结构和算法，以及需要并行开发硬件和软件等因素的影响，都体现在这个分级因素中。

? 开发灵活性

这个分级因素反映了与预先确定的外部接口相一致的需求，以及为了及早实现所需要的额外费用。

? 风险排除度

这个分级因素反映了重大风险已被消除的比例。在多数情况下，这个比例和指定了重要模块接口（即选定了体系结构）的比例密切相关。

? 项目组的凝聚力

这个分级因素表明了开发人员相互协作时可能存在的困难。这个因素反映了开发人员在目标和文化背景等方面相一致的程度，以及开发人员组成一个小组工作的经验。

? 过程成熟度

这个分级因素反映了按照能力成熟度模型（见 7.7 节）度量出的项目组织的成熟度。

在原始的 COCOMO 模型中，仅仅粗略地考虑了前两个分级因素对指数 b 之值的影响。

工作量方程中模型系数 a 的典型值为 3.0，应该根据历史经验数据确定一个适合本组织所开发的项目类型的数值。

7.3 进度计划

不论从事何种技术性项目，实际情况都是，在实现一个大目标之前往往必须完成数以百计的小任务（也称为作业）。这些任务中有一些是处于“关键路径”（见 7.3.5 节）之外的，其完成时间如果没有严重拖后，则不会影响整个项目的完成时间；其他任务则处于关键路径之中，如果这些“关键任务”的进度拖后，则整个项目的完成日期就会拖后。

没有一个普遍适用于所有软件项目的任务集合，因此，一个有效的软件过程应该定义一组适合于所从事的项目的“任务集合”。一个任务集合包括一组软件工程工作任务、里程碑和可交付的产品。为一个项目所定义的任务集合，必须包括为最终获得高质量的软件产品而应该完成的所有工作，但是同时又不能让项目组负担不必要的工作。

项目管理者的目地是定义全部项目任务，识别出关键任务，跟踪关键任务的进展状况，以保证能及时发现拖延进度的情况。为了做到这一点，管理者必须制定一个足够详细的进度表，以便监督项目进度，并控制整个项目。

软件项目的进度安排是一项活动，它通过把工作量分配给特定的软件工程任务，并规定完成各项任务的起、止日期，从而将估算的工作量分布于计划好的项目持续期内。进度计划将随着时间的流逝而不断演化。在项目计划的早期，首先制定一个宏观的进度安排表，标识出主要的软件工程活动和这些活动影响到的产品功能。随着项目的进展，把宏观进度表中的每个条目都精化成一个详细进度表。于是完成一个活动所必须实现的特定任务被标识出来，并安排好了实现这些任务的进度。

7.3.1 估算开发时间

估算出完成给定项目所需的总工作量之后，另一个需要回答的问题是：用多长时间才能完成该项目的开发工作？对于一个估计工作量为 20 人月的项目，你可能想到下列那些进度表：

- ? 1 个人用 20 个月完成该项目；
- ? 4 个人用 5 个月完成该项目；
- ? 20 个人用 1 个月完成该项目。

但是，这些进度表并不现实，软件开发时间与从事开发工作的人数并不是简单的反比关系。

通常，成本估计模型也同时为我们提供估算开发时间 T 的方程。与工作量方程不同，各种模型估算开发时间的方程非常类似，如下所示：

(1) Walston-Felix 模型

$$T = 2.5E^{0.35}$$

(2) 原始 COCOMO 模型

$$T = 2.5E^{0.38}$$

(3) COCOMO 2 模型

$$T = 3.0E^{0.33+0.2\times(b-1.01)}$$

(4) Putnam 模型

$$T = 2.4E^{1/3}$$

其中，

E 为开发工作量（以人月为单位），

T 为开发时间（以月为单位）。

用上列方程计算出的 T 值，代表正常开发时间。客户往往希望缩短软件的开发时间，显然，为了缩短开发时间应该增加从事开发工作的人数。

但是，经验告诉我们，随着开发小组规模增大，个人的生产率将下降。出现这种现象主要有下述两个原因：

? 当小组变得更大时，每个人需要用更多时间与组内其他成员讨论问题、协调工作，因此，通信开销增加了。

? 如果在开发过程中增加小组人员，则最初一段时间内项目组总生产率不仅不会提高反而会下降。这是因为开始时新成员还不是生产力，而且在他们学习期间还需要花费小组其他成员的时间。

综合上述两个原因，存在被称为 Brooks 规律的下述现象：向一个已经延期的项目增加人力，只会使得它更加延期。

下面，让我们研究项目组规模与项目组总生产率的关系。

项目组成员之间的通信路径数，由项目组的规模和项目组的结构决定。如果项目组规模为 P （即有 P 名组员），而且每个组员必须与其他所有组员通信以协调开发活动，则通信路径数为 $P(P - 1)/2$ 。如果每个组员只需与另外一个组员通信，则通信路径数为 $P - 1$ 。通信路径数少于 $P - 1$ 是不合理的，因为那将导致出现与任何人都没联系的组员。

因此，通信路径数大约在 $P \sim P^2/2$ 的范围内变化。也就是说，在一个层次结构的项目组中，通信路径数为 P^r ，其中 $1 < r < 2$ 。

对于某一个组员来说，他与其他组员通信的路径数在 $1 \sim P - 1$ 的范围内变化。如果不与其他人通信时个人生产率为 L ，而且每条通信路径导致生产率减少 l ，则平均生产率为

$$L_r = L - l(P - 1)^r \quad (7.5)$$

其中， r 是对通信路径数的度量， $0 < r < 1$ （假设至少有一个人需要与一个以上的其他组员通信，因此 $r > 0$ ）。

对于一个规模为 P 的项目组，从 (7.5) 式导出项目组的总生产率为

$$L_{\text{tot}} = P \times L_r = P(L - l(P - 1)^r) \quad (7.6)$$

对于给定的一组 L , l 和 r 的值，总生产率 L_{tot} 是项目组规模 P 的函数，随着 P 值增加 L_{tot} 将从 0 增大到某个最大值，然后再下降。因此，存在一个最佳的项目组规模 P_{opt} ，这个规模的项目组总生产率最高。

让我们举例说明项目组规模与生产率的关系。假设个人最高生产率为 500LOC / 月（即 $L = 500$ ），每条通信路径导致生产率下降 10%（即 $l = 50$ ）。如果每个组员都必须与组内所有其他成员交互 ($r = 1$)，则项目组规模与生产率的关系列在表 7.4 中，可见，在这种情况下最佳规模是 5.5 人（即 $P_{\text{opt}} = 5.5$ ）。

表 7.4

项目组规模对生产率的影响

项目组规模	个人生产率	项目组总生产率
1	500	500
2	450	900
3	400	1200
4	350	1400
5	300	1500
5.5	275	1512
6	250	1500
7	200	1400
8	150	1200

用总生产率 L_{tot} 的最大值（与项目组最佳规模 P_{opt} 相对应）除软件规模（代码行数），就可得到最短开发时间。

事实上，做任何事情都需要时间。我们不可能用“人力换时间”的办法无限制地缩短一个软件项目的开发时间。Boehm 根据经验指出，软件项目开发时间最多可以减少至正常开发时间的 75%。如果试图把开发时间压缩得太短，则成功的概率几乎为 0。

7.3.2 甘特 (Gantt) 图

估算出完成软件项目所需要的开发时间之后，就可以着手制定进度计划了。Gantt 图是历史悠久、应用广泛的进度计划工具，下面通过一个非常简单的例子介绍这种工具。

假设有一座陈旧的矩形木板房需要重新油漆。这项工作必须分三步完成：首先刮掉旧漆，然后刷上新漆，最后清除溅在窗户上的油漆。假设一共分配了 15 名工人去完成这项工作，然而工具却很有限：只有 5 把刮旧漆用的刮板，5 把刷漆用的刷子，5 把清除溅在窗户上的油漆用的小刮刀。怎样安排才能使工作进行得更有效呢？

一种做法是首先刮掉四面墙壁上的旧漆，然后给每面墙壁都刷上新漆，最后清除溅在每个窗户上的油漆。显然这是效率最低的做法，因为总共有 15 名工人，然而每种工具却只有 5 件，这样安排工作在任何时候都有 10 名工人闲着没活干。

读者可能已经想到，应该采用“流水作业法”，也就是说，首先由 5 名工人用刮板刮掉第一面墙上的旧漆（这时其余 10 名工人休息），当第一面墙刮净后，另外 5 名工人立即用刷子给这面墙刷新漆（与此同时拿刮板的 5 名工人转去刮第二面墙上的旧漆），一旦刮旧漆的工人转到第三面墙而且刷新漆的工人转到第二面墙以后，余下的 5 名工人立即拿起刮刀去清除溅在第一面墙窗户上的油漆，……。这样安排每个工人都有活干，因此能够在较短的时间内完成任务。

表 7.5

各道工序估计需用的时间(小时)

工序 墙壁	刮 旧 漆	刷 新 漆	清 理
1 或 3	2	3	1
2 或 4	4	6	2

假设木板房的第 2、4 两面墙的长度比第 1、3 两面墙的长度长一倍，此外，不同工作需要用的时间长短也不同，刷新漆最费时间，其次是刮旧漆，清理（即清除溅在窗户上的油漆）需要的时间最少。表 7.5 列出了估计每道工序需要用的时间。我们可以使用图 7.1 中的 Gantt 图描绘上述流水作业过程：在时间为零时开始刮第一面墙上的旧漆，两小时后刮旧漆的工人转去刮第二面墙，同时另 5 名工人开始给第一面墙刷新漆，每当给一面墙刷完新漆之后，第三组的 5 名工人立即清除溅在这面墙窗户上的漆。从图 7.1 可以看出 12 小时后刮完所有旧漆，20 小时后完成所有墙壁的刷漆工作，再过 2 小时后清理工作结束。因此全部工程在 22 小时后结束，如果用前述的第一种做法，则需要 36 小时。

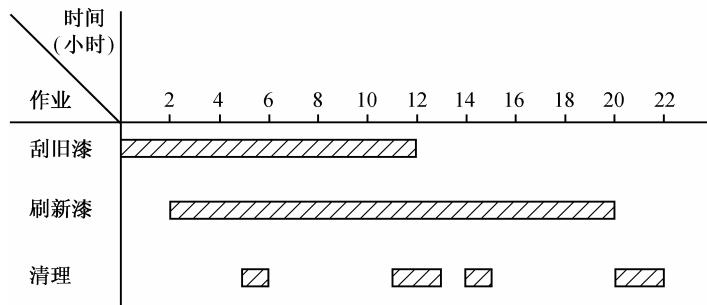


图 7.1 旧木板房刷漆工程的 Gantt 图

为了醒目地表示里程碑，可以在 Gantt 图中加上菱形标记，一个菱形代表一个里程碑，如图 7.2 所示。

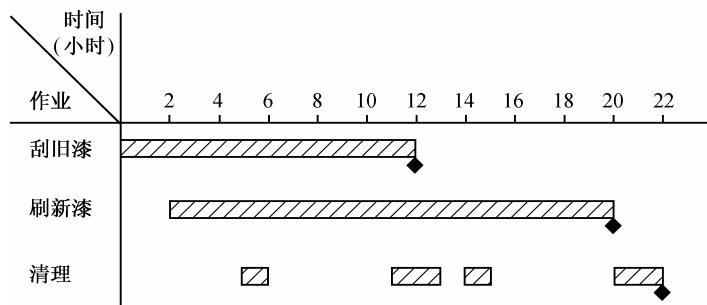


图 7.2 标有里程碑的 Gantt 图

7.3.3 工程网络

上一小节介绍的 Gantt 图能很形象地描绘任务分解情况，以及每个子任务（作业）的开始

时间和结束时间，因此是进度计划和进度管理的有力工具。它具有直观简明和容易掌握、容易绘制的优点，但是 Gantt 图也有三个主要缺点：

- (1) 不能显式地描绘各项作业彼此间的依赖关系；
- (2) 进度计划的关键部分不明确，难于判定哪些部分应当是主攻和主控的对象；
- (3) 计划中有潜力的部分及潜力的大小不明确，往往造成潜力的浪费。

当把一个工程项目分解成许多子任务，并且它们彼此间的依赖关系又比较复杂时，仅仅用 Gantt 图作为安排进度的工具是不够的，不仅难于做出既节省资源又保证进度的计划，而且还容易发生差错。

工程网络是制定进度计划时另一种常用的图形工具，它同样能描绘任务分解情况以及每项作业的开始时间和结束时间，此外，它还显式地描绘各个作业彼此间的依赖关系。因此，工程网络是系统分析和系统设计的强有力的工具。

在工程网络中用箭头表示作业（例如，刮旧漆，刷新漆，清理等），用圆圈表示事件（一项作业开始或结束）。注意，事件仅仅是可以明确定义的时间点，它并不消耗时间和资源。作业通常既消耗资源又需要持续一定时间。图 7.3 是旧木板房刷漆工程的工程网络。图中表示刮第 1 面墙上旧漆的作业开始于事件 1，结束于事件 2。用开始事件和结束事件的编号标识一个作业，因此“刮第 1 面墙上旧漆”是作业 1—2。

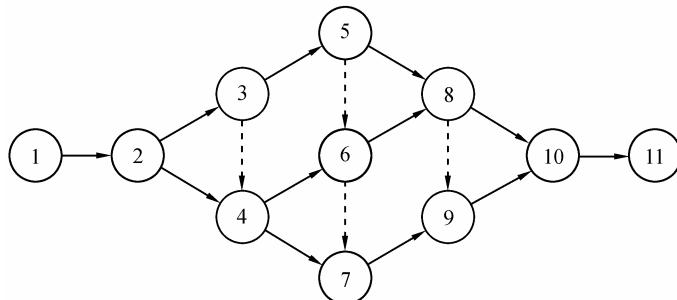


图 7.3 旧木板房刷漆工程的工程网络

图中：1—2 刮第 1 面墙上的旧漆；2—3 刮第 2 面墙上的旧漆；2—4 给第 1 面墙刷新漆；

3—5 刮第 3 面墙上的旧漆；4—6 给第 2 面墙刷新漆；4—7 清理第 1 面墙窗户；

5—8 刮第 4 面墙上的旧漆；6—8 给第 3 面墙刷新漆；7—9 清理第 2 面墙窗户；

8—10 给第 4 面墙刷新漆；9—10 清理第 3 面墙窗户；10—11 清理第 4 面墙窗户；虚拟作业：

3—4；5—6；6—7；8—9

在工程网络中的一个事件，如果既有箭头进入又有箭头离开，则它既是某些作业的结束又是另一些作业的开始。例如，图 7.3 中事件 2 既是作业 1—2（刮第 1 面墙上的旧漆）的结束，又是作业 2—3（刮第 2 面墙上的旧漆）和作业 2—4（给第 1 面墙刷新漆）的开始。也就是说，只有第 1 面墙上的旧漆刮完之后，才能开始刮第 2 面墙上的旧漆和给第 1 面墙刷新漆这两个作业。因此，工程网络显式地表示了作业之间的依赖关系。

在图 7.3 中还有一些虚线箭头，它们表示虚拟作业，也就是事实上并不存在的作业。引入虚拟作业是为了显式地表示作业之间的依赖关系。例如，事件 4 既是给第 1 面墙刷新漆结束，又是给第 2 面墙刷新漆开始（作业 4—6）。但是，在开始给第 2 面墙刷新漆之前，不仅必须已

经给第 1 面墙刷完了新漆，而且第 2 面墙上的旧漆也必须已经刮净（事件 3）。也就是说，在事件 3 和事件 4 之间有依赖关系，或者说在作业 2—3（刮第 2 面墙上旧漆）和作业 4—6（给第 2 面墙刷新漆）之间有依赖关系，虚拟作业 3—4 明确地表示了这种依赖关系。注意，虚拟作业既不消耗资源也不需要时间。请读者研究图 7.3，参考图下面对各项作业的描述，解释引入其他虚拟作业的原因。

7.3.4 估算进度

画出类似图 7.3 那样的工程网络之后，系统分析员就可以借助它的帮助估算工程进度了。为此需要在工程网络上增加一些必要的信息。

首先，把每个作业估计需要使用的时间写在表示该项作业的箭头上方。注意，箭头长度和它代表的作业持续时间没有关系，箭头仅表示依赖关系，它上方的数字才表示作业的持续时间。

其次，为每个事件计算下述两个统计数字：最早时刻 EET 和最迟时刻 LET。这两个数字将分别写在表示事件的圆圈的右上角和右下角，如图 7.4 左下角的符号所示。

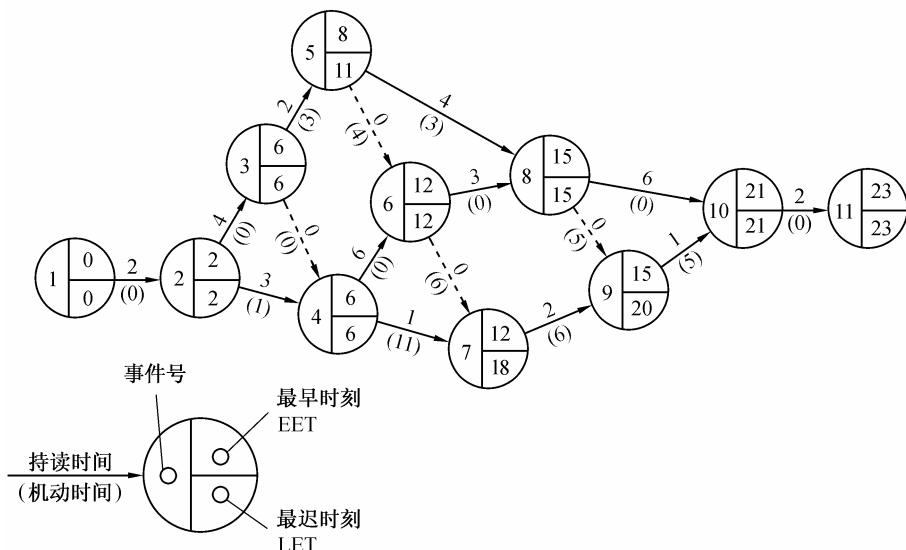


图 7.4 旧木板房刷漆工程的完整的工程网络（粗线箭头是关键路径）

事件的最早时刻是该事件可以发生的最早时间。通常工程网络中第一个事件的最早时刻定义为零，其他事件的最早时刻在工程网络上从左至右按事件发生顺序计算。计算最早时刻 EET 使用下述三条简单规则：

- (1) 考虑进入该事件的所有作业；
- (2) 对于每个作业都计算它的持续时间与起始事件的 EET 之和；
- (3) 选取上述和数中的最大值作为该事件的最早时刻 EET。

例如，从图 7.3 可以看出事件 2 只有一个作业（作业 1—2）进入，就是说，仅当作业 1—2 完成时事件 2 才能发生，因此事件 2 的最早时刻就是作业 1—2 最早可能完成的时刻。定义事件 1 的最早时刻为零，据估计，作业 1—2 的持续时间为 2 小时，也就是说，作业 1—2

最早可能完成的时刻为 2，因此，事件 2 的最早时刻为 2。同样，只有一个作业（作业 2—3）进入事件 3，这个作业的持续时间为 4 小时，所以事件 3 的最早时刻为 $2 + 4 = 6$ 。事件 4 有两个作业（2—4 和 3—4）进入，只有这两个作业都完成之后，事件 4 才能出现（事件 4 代表上述两个作业的结束）。已知事件 2 的最早时刻为 2，作业 2—4 的持续时间为 3 小时；事件 3 的最早时刻为 6，作业 3—4（这是一个虚拟作业）的持续时间为 0，按照上述三条规则，可以算出事件 4 的最早时刻为

$$EET = \max \{ 2 + 3, 6 + 0 \} = 6$$

按照这种方法，不难沿着工程网络从左至右顺序算出每个事件的最早时刻，计算结果标在图 7.4 的工程网络中（每个圆圈内右上角的数字）。

事件的最迟时刻是在不影响工程竣工时间的前提下，该事件最晚可以发生的时刻。按惯例，最后一个事件（工程结束）的最迟时刻就是它的最早时刻。其他事件的最迟时刻在工程网络上从右至左按逆作业流的方向计算。计算最迟时刻 LET 使用下述三条规则：

- (1) 考虑离开该事件的所有作业；
- (2) 从每个作业的结束事件的最迟时刻中减去该作业的持续时间；
- (3) 选取上述差数中的最小值做为该事件的最迟时刻 LET。

例如，按惯例图 7.4 中事件 11 的最迟时刻和最早时刻相同，都是 23。逆作业流方向接下来应该计算事件 10 的最迟时刻，离开这个事件的只有作业 10—11，该作业的持续时间为 2 小时，它的结束事件（事件 11）的 LET 为 23，因此，事件 10 的最迟时刻为

$$LET = 23 - 2 = 21$$

类似地，事件 9 的最迟时刻为

$$LET = 21 - 1 = 20$$

事件 8 的最迟时刻为

$$LET = \min \{ 21 - 6, 20 - 0 \} = 15$$

图 7.4 中每个圆圈内右下角的数字就是该事件的最迟时刻。

7.3.5 关键路径

图 7.4 中有几个事件的最早时刻和最迟时刻相同，这些事件定义了关键路径，在图中关键路径用粗线箭头表示。关键路径上的事件（关键事件）必须准时发生，组成关键路径的作业（关键作业）的实际持续时间不能超过估计的持续时间，否则工程就不能准时结束。

工程项目的管理人员应该密切注视关键作业的进展情况，如果关键事件出现的时间比预计的时间晚，则会使最终完成项目的时间拖后；如果希望缩短工期，只有往关键作业中增加资源才会有效果。

7.3.6 机动时间

不在关键路径上的作业有一定程度的机动余地——实际开始时间可以比预定时间晚一些，或者实际持续时间可以比预计的持续时间长一些，而并不影响工程的结束时间。一个作业可以有的全部机动时间等于它的结束事件的最迟时刻减去它的开始事件的最早时刻，再减去这个作业的持续时间：

$$\text{机动时间} = (\text{LET})_{\text{结束}} - (\text{EET})_{\text{开始}} - \text{持续时间}$$

对于前述油漆旧木板房的例子，计算得到的非关键作业的机动时间列在表 7.6 中。

表 7.6 旧木板房刷漆网络中的机动时间

作 业	LET(结束)	EET(开始)	持 续 时 间	机 动 时 间
2—4	6	2	3	1
3—5	11	6	2	3
4—7	18	6	1	11
5—6	12	8	0	4
5—8	15	8	4	3
6—7	18	12	0	6
7—9	20	12	2	6
8—9	20	15	0	5
9—10	21	15	1	5

在工程网络中每个作业的机动时间写在代表该项作业的箭头下面的括弧里（参看图 7.4）。

在制定进度计划时仔细考虑和利用工程网络中的机动时间，往往能够安排出既节省资源又不影响最终竣工时间的进度表。例如，研究图 7.4（或表 7.6）可以看出，清理前三面墙窗户的作业都有相当多机动时间，也就是说，这些作业可以晚些开始或者持续时间长一些（少用一些资源），并不影响竣工时间。此外，刮第 3、第 4 面墙上旧漆和给第 1 面墙刷新漆的作业也都有机动时间，而且这后三项作业的机动时间之和大于清理前三面墙窗户需要用的工作时间。因此，有可能仅用 10 名工人在同样时间内（23 小时）完成旧木板房刷漆工程。进一步研究图 7.4 中的工程网络可以看出，确实能够只用 10 名工人在同样时间内完成这项任务，而且可以安排出几套不同的进度计划，都可以既减少 5 名工人又不影响竣工时间。在图 7.5 中的 Gantt 图描绘了其中的一种方案。

图 7.5 的方案不仅比图 7.1 的方案明显节省人力，而且改正了图 7.1 中的一个错误：因为给第 2 面墙刷新漆的作业 4—6 不仅必须在给第 1 面墙刷完新漆之后（作业 2—4 结束），而且

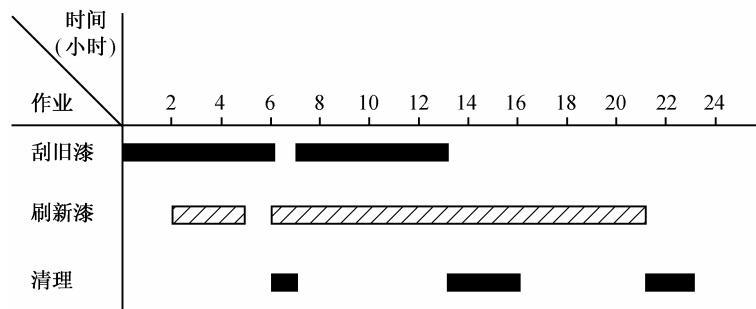


图 7.5 旧木板房刷漆工程改进的 Gantt 图之一

图中：粗实线代表由甲组工人完成的作业；斜划线代表由乙组工人完成的作业

还必须在把第 2 面墙上的旧漆刮净之后（作业 2—3 和虚拟作业 3—4 结束）才能开始，所以给第 1 面墙刷完新漆之后不能立即开始给第 2 面墙刷新漆的作业，需等到把第 2 面墙上旧漆刮净之后才能开始，也就是说，全部工程需要 23 个小时而不是 22 个小时。

这个简单例子明显说明了工程网络比 Gantt 图优越的地方：它显式地定义事件及作业之间的依赖关系，Gantt 图只能隐含地表示这种关系。但是 Gantt 图的形式比工程网络更简单更直观，为更多的人所熟悉，因此，应该同时使用这两种工具制定和管理进度计划，使它们互相补充取长补短。

以上通过旧木板房刷新漆工程的简单例子，介绍了制定进度计划的两个重要工具和方法。软件工程项目虽然比这个简单例子复杂得多，但是计划和管理的基本方法仍然是自顶向下分解，也就是把项目分解为若干个阶段，每个阶段再分解成许多更小的任务，每个任务又可进一步分解为若干个步骤等等。这些阶段、任务和步骤之间有复杂的依赖关系，因此，工程网络和 Gantt 图同样是安排进度和管理工程进展情况的强有力的工具。

本章第 7.2 节中介绍的成本估计技术可以帮助我们估算每项任务的工作量，而 7.3.1 小节讲述的方法可以帮助我们估算正常开发时间、最佳项目组规模和最短开发时间。参照这些数据可以为每项任务分配适当的人力，根据人力分配情况可以进一步确定每项任务的持续时间。从这些基本数据出发，根据作业之间的依赖关系，利用工程网络和 Gantt 图可以制定出合理的进度计划，并且能够科学地管理软件开发工程的进展情况。

7.4 人 员 组 织

软件项目成功的关键是有高素质的软件开发人员。然而大多数软件产品规模都很大，以至单个软件开发人员无法在给定期限内完成开发工作，因此，必须把多名软件开发人员组织起来，使他们分工协作共同完成开发工作。

为了成功地完成软件开发工作，项目组成员必须以一种有意义且有效的方式彼此交互和通信。如何组织项目组是一个管理问题，管理者必须合理地组织项目组，使项目组有较高生产率，能够按预定的进度计划完成所承担的工作。经验表明，项目组组织得越好，其生产率越高，而且产品质量也越高。

除了追求更好的组织方式之外，每个管理者的目标都是建立有凝聚力的项目组。一个有高度凝聚力的小组，是一批团结得非常紧密的人，他们的整体力量大于个体力量的总和。一旦项目组开始具有凝聚力，成功的可能性就大大增加了。

下面介绍几种典型的人员组织方式。

7.4.1 民主制程序员组

民主制程序员组通常采用非正式的组织方式，也就是说，虽然名义上有一个组长，但是他和组内其他成员完成同样的任务，地位是平等的。在这样的小组中，由全体组员讨论决定应该完成的工作，协商解决重大的技术问题，并且根据每个人的能力和经验分配适当的任务。

民主制程序员组的主要优点是，对发现错误抱着积极的态度，这种积极态度有助于更快地发现错误，从而导致高质量的代码。

民主制程序员组的另一个优点是，小组成员享有充分民主，小组有高度凝聚力，组内学术空气浓厚，有利于攻克技术难关。因此，当有难题需要解决时，也就是说，当所要开发的软件产品的技术难度较高时，采用民主制程序员组是适宜的。

如果组内多数成员是经验丰富技术熟练的程序员，那么上述非正式的组织方式可能会非常成功。在这样的小组内组员享有充分民主，通过协商，在自愿的基础上作出决定，因此能够增强团结、提高工作效率。但是，如果组内多数成员技术水平不高，或是缺乏经验的新手，那么这种非正式的组织方式也有严重缺点：由于没有明确的权威指导开发工程的进行，组员间将缺乏必要的协调，最终可能导致工程失败。

为了使少数组经验丰富、技术高超的程序员在软件开发过程中能够发挥更大作用，程序设计小组也可以采用下一小节中介绍的另外一种组织形式。

7.4.2 主程序员组

美国 IBM 公司在 20 世纪 70 年代初期开始采用主程序员组的组织方式。主程序员组用经验丰富、技术好、能力强、会管理的程序员作为主程序员，同时，由人和计算机在事务性工作方面给主程序员提供充分支持，而且所有通信都通过一两个人进行，以减少通信开销。这种组织方式类似于外科手术小组的组织：主刀大夫对手术全面负责，并且完成制定手术方案、开刀等关键工作，同时又有麻醉师、护士等技术熟练的专门人员协助和配合他的工作。

一个典型的主程序员组如图 7.6 所示。该组由主程序员、后备程序员、编程秘书以及 1~3 名程序员组成。在必要的时候，该组还有其他领域的专家（例如，法律专家，财务专家等）协助。

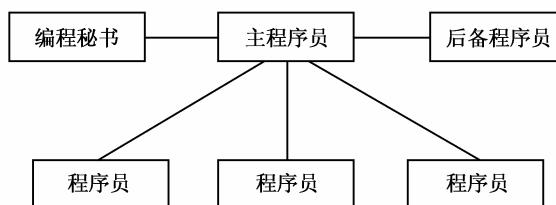


图 7.6 主程序员组的结构

主程序员组核心人员的分工如下。

? 主程序员既是成功的管理人员又是经验丰富、能力强的高级程序员，负责体系结构设计和关键部分（或复杂部分）的详细设计，并且负责指导其他程序员完成详细设计和编码工作。如图 7.6 所示，程序员之间没有通信渠道，所有接口问题都由主程序员处理。因为主程序员要对每行代码的质量负责，所以他还要对其他成员的工作成果进行复查。

? 后备程序员也应该技术熟练而且富于经验，他协助主程序员工作并且在必要时（例如，主程序员生病、出差或“跳槽”）接替主程序员的工作。因此，后备程序员必须在各个方面都和主程序员一样优秀，并且对当前项目的了解也应该和主程序员一样多。平时，后备程序员的主要工作是，设计测试方案和分析测试结果，以验证主程序员的工作结果。

? 编程秘书负责完成与项目有关的全部事务性工作，维护项目资料库和项目文档，编译、链接、执行源程序和测试用例。

注意，上述分工是在 20 世纪 70 年代初提出的，现在已经和当时大不相同了：程序员在自己的终端或工作站上完成软件开发工作，他们自己输入、编辑、编译、链接和测试源程序，无须由编程秘书统一做这些工作。典型的主程序员组的现代形式将在下一节介绍。

使用主程序员组的主要目的是提高生产率和软件产品质量。根据国外某些统计资料，使用这种组织形式时生产率大约提高了一倍，一些软件产品的质量也确实提高了。

虽然主程序员组的组织方式说起来有不少优点，但是，典型的主程序员组在许多方面是不切实际的。

首先，如前所述，主程序员应该是高级程序员和成功的管理者的结合体，承担这项工作需要同时具备这两方面的才能，但是，在现实社会中很难找到这样的人才。通常，既缺乏成功的管理者，也缺乏技术熟练的程序员。

其次，后备程序员更难找到。人们总是期望后备程序员像主程序员一样出色，但是，他们必须坐在“替补席”上，拿着较低的工资等待随时接替主程序员的工作。几乎没有一个高级程序员或高级管理人员愿意接受这样的工作。

第三，编程秘书也很难找到。软件专业人员一般都讨厌日常的事务性工作，但是，人们却期望编程秘书整天只干这类工作。

我们需要一种更合理、更现实的组织程序员组的方法，这种方法应该能充分结合民主制程序员组和主程序员组的优点，并能用于实现更大规模的软件产品。

7.4.3 现代程序员组

民主制程序员组的最大优点，是组员们都对发现程序错误持积极、主动的态度。主程序员组的组织方式与民主制程序员组很不相同，主程序员对每行代码的质量负责，必须参与全部代码审查工作，但是，主程序员同时又是负责对小组成员进行评价的管理员，他参与代码审查工作自然会把所发现的程序错误与小组成员的工作业绩联系起来，从而造成小组成员不愿意发现错误的心理。

摆脱上述矛盾的方法是，取消主程序员的大部分行政管理工作。上一小节已经指出，很难找到一个既是高度熟练的程序员又是成功的管理员的人，取消主程序员的行政管理工作，不仅可消除组员不愿意发现错误的心理障碍，也使得寻找主程序员的人选不再那么困难。于是，实际的“主程序员”应该由两个人共同担任：一个技术负责人，负责小组的技术活动；一个行政负责人，负责所有非技术的管理决策。

这样的组织结构如图 7.7 所示。

在开始工作之前明确划分技术组长和行政组长的管理权限是很重要的。但是，有时也会出现职责不清的问题，例如，考虑下述的休年假问题。行政组长有权批准某个程序员休年假的申请，因为这是一个非技术问题；但是，技术组长可能立即否决了这个申请，因为已经接近预定的产品完工日期，现在人手非常紧张。解决这类办法是求助于更高层的管理人员，对行政组长和技术

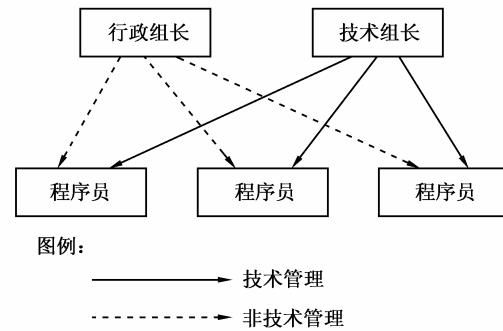


图 7.7 现代程序员组

组长都认为是属于自己职责范围的事务，制定一个处理方案。

由于程序员组的人数不宜过多，当软件项目规模较大时，应该把程序员分成若干个小组，采用图 7.8 所示的组织结构。该图描绘的是技术管理组织的结构，非技术管理组织的结构与此类似。由图可以看出，产品作为一个整体其开发工作是在项目经理的指导下进行的，程序员向他们的组长汇报工作，而组长向项目经理汇报工作。当产品规模更大时，可以再增加中间管理层次。

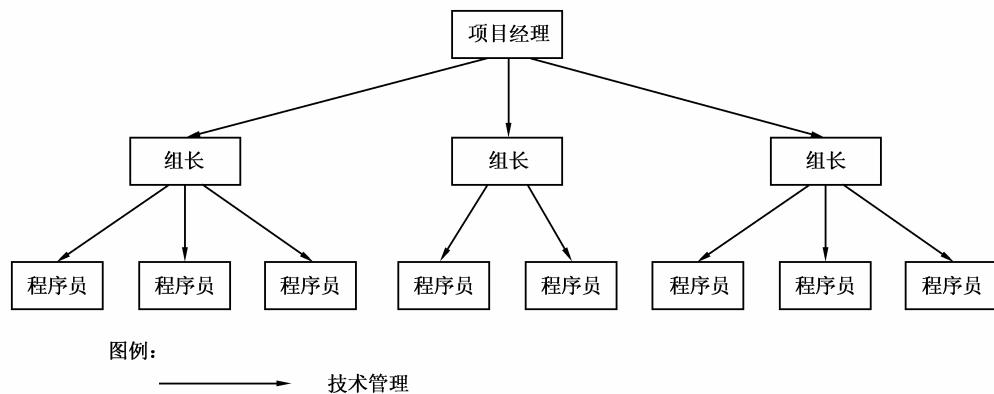


图 7.8 大型项目的技术管理组织结构

把民主制程序员组和主程序员组的优点结合起来的另一种方法，是在合适的地方采用分散做决定的方法，如图 7.9 所示。这样做有利于形成畅通的通信渠道，以便充分发挥每个程序员的积极性和主动性，集思广益攻克技术难关。这种组织方式对于适合采用民主方法的那类问题（例如，研究性项目或遇到技术难题需要用集体智慧攻关）非常有效。尽管这种组织方式适当地发扬了民主，但是上下级之间的箭头（即管理关系）仍然是向下的，也就是说，是在集中指导下发扬民主。显然，如果程序员可以指挥项目经理，则只会引起混乱。

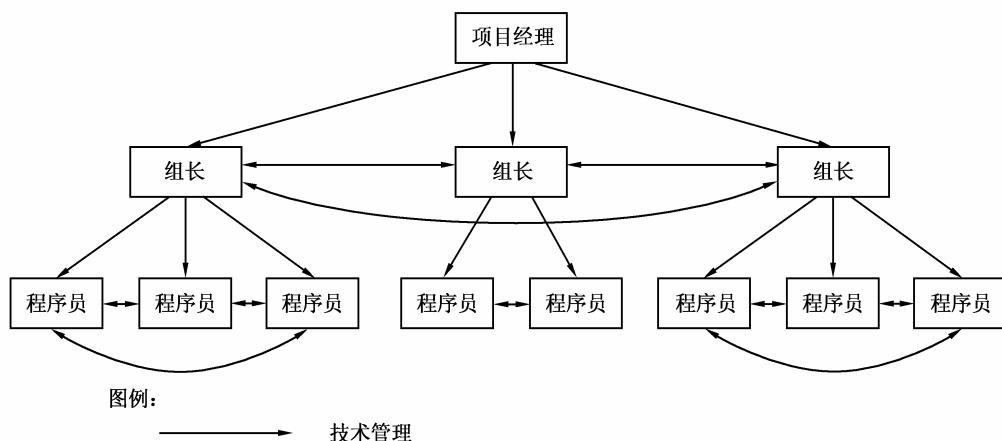


图 7.9 包含分散决策的组织方式

7.5 质量保证

质量是产品的生命，不论生产何种产品，质量都是极其重要的。软件产品开发周期长，耗费巨大的人力和物力，更必须特别注意保证质量。

7.5.1 软件质量的定义

概括地说，软件质量就是“软件与明确地和隐含地定义的需求相一致的程度”。更具体地说，软件质量是软件与明确地叙述的功能和性能需求、文档中明确描述的开发标准、以及所有专业开发的软件都应该具有的隐含特征相符合的程度。上述定义强调了下述三个要点：

- (1) 软件需求是度量软件质量的基础，软件与对它的需求不一致就是质量不高。
- (2) 指定的标准定义了一组指导软件开发的准则，如果没有严格遵守这些准则，几乎肯定会导致质量不高。
- (3) 通常，有一组没有显式描述的隐含需求（例如，软件应该易学、易用、易维护）。如果软件虽然满足明确描述的需求，但却不满足隐含的需求，那么软件的质量仍然是值得怀疑的。

目前，虽然定量度量软件质量有一定难度，但是仍然能够提出许多重要的软件质量指标（其中多数指标还处于定性度量阶段）。

下面介绍影响软件质量的主要因素，这些因素是从管理角度对软件质量的度量。可以把这些质量因素划分成三组，它们分别反映用户在使用软件产品时的三种不同倾向或观点。这三种倾向是：产品运行，产品修改和产品转移。图 7.10 描绘了软件质量因素和上述三种倾向（或称为产品活动）之间的关系，表 7.7 给出了软件质量因素的简明定义。

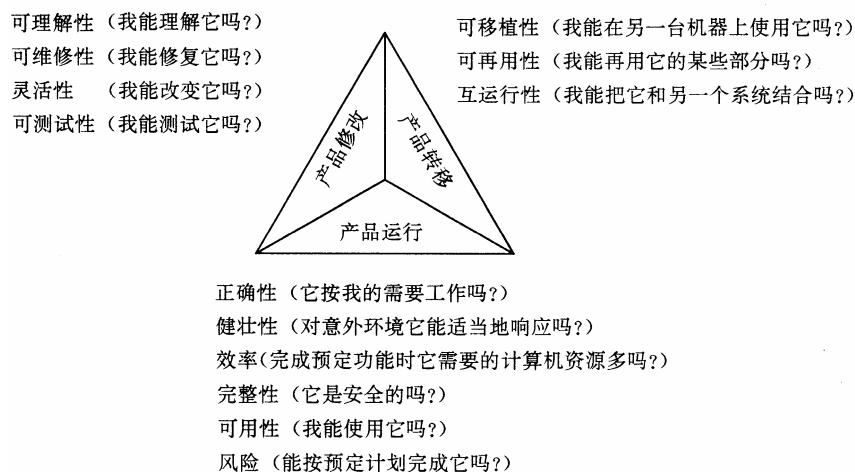


图 7.10 软件质量因素与产品活动的关系

软件工程

表 7.7

软件质量因素的定义

质量因素	定义
正确性	系统满足规格说明和用户目标的程度，即，在预定环境下能正确地完成预期功能的程度
健壮性	在硬件发生故障、输入的数据无效或操作错误等意外环境下，系统能做出适当响应的程度
效率	为了完成预定的功能，系统需要的计算资源的多少
完整性（安全性）	对未经授权的人使用软件或数据的企图，系统能够控制（禁止）的程度
可用性	系统在完成预定应该完成的功能时令人满意的程度
风险	按预定的成本和进度把系统开发出来，并且为用户所满意的概率
可理解性	理解和使用该系统的容易程度
可维修性	诊断和改正在运行现场发现的错误所需要的工作量的大小
灵活性（适应性）	修改或改进正在运行的系统需要的工作量的多少
可测试性	软件容易测试的程度
可移植性	把程序从一种硬件配置和（或）软件系统环境转移到另一种配置和环境时，需要的工作量多少。有一种定量度量的方法是：用原来程序设计和调试的成本除移植时需用的费用
可再用性	在其他应用中该程序可以被再次使用的程度（或范围）
互运行性	把该系统和另一个系统结合起来需要的工作量的多少

7.5.2 软件质量保证措施

软件质量保证（Software Quality Assurance，通常缩写为 SQA）的措施主要有，基于非执行的测试（也称为复审） 基于执行的测试（即本书第 4 章和第 5 章讲述的测试）和程序正确性证明。复审主要用来保证在编码之前各阶段产生的文档的质量；基于执行的测试需要在程序编写出来之后进行，它是保证软件质量的最后一道防线；程序正确性证明使用数学方法来严格验证程序是否与对它的说明完全一致。

参加软件质量保证工作的人员，可以分成下述两类。

？ 软件工程师通过采用可靠的技术方法和度量、进行正式的技术复审以及完成计划周密的测试来保证软件质量。

？ SQA 小组的职责是，辅助软件工程小组以获得高质量的软件产品。其从事的软件质量保证活动主要是计划、监督、记录、分析和报告。简而言之，SQA 小组的作用是，通过确保软件过程的质量来保证软件产品的质量。

1. 技术复审的必要性

正式技术复审的明显优点是，能够较早地发现错误，防止错误被传播到软件过程的后续

阶段。

统计数字表明，在大型软件产品中检测出的错误，有 60% ~ 70% 属于规格说明错误或设计错误。研究表明，正式技术复审在发现规格说明错误和设计错误方面的有效性高达 75%。由于能够检测出并排除掉绝大部分的这类错误，复审过程将极大地降低后续开发和维护阶段的成本。

正式技术复审实际上是一类复审方法，包括走查（Walkthrough）和审查（Inspection）等具体方法。走查的步骤比审查少，而且没有审查那样正规。

2. 走查

走查组由 4 ~ 6 名成员组成。以规格说明走查组为例，成员至少包括一名负责起草规格说明的人，一位负责该规格说明的管理员，一位客户代表，以及下阶段开发组（在本例中是设计组）的一名代表和 SQA 小组的一名代表。其中，SQA 小组的代表应该作为走查组的组长。

为了能发现重大的错误，走查组成员最好是经验丰富的高级技术人员。必须把被走查的材料预先分发给走查组每位成员。走查组成员应该仔细研究材料并列出两张表：一张是该成员不理解的术语，另一张是他认为不正确的术语。

走查组组长引导该组成员走查文档，力求发现尽可能多的错误。走查组的任务仅仅是标记出错误而不是改正错误，改正错误的工作应该由该文档的编写组完成。走查的时间不要超过 2 小时，这段时间应该用来发现和标记错误，而不是改正错误。

走查主要有下述两种方式。

(1) 参与者驱动法

参与者按照事先准备好的列表，提出他们不理解的术语和认为不正确的术语。文档编写组的代表必须对每个质疑做出回答，要么承认确实有错误，要么对质疑做出解释。

(2) 文档驱动法

文档编写者向走查组成员仔细解释文档。在此过程中走查组成员不时针对事先预备好的问题或解释过程中发现的问题提出质疑。这种方法可能比第一种方法更彻底，往往能检测出更多的错误。经验表明，采用文档驱动法时许多错误是由文档讲解者自己发现的。

3. 审查

审查的范围比走查更广泛，它的步骤也比较多。通常，审查有 5 个基本步骤：

(1) 综述

由负责编写文档的一名成员向审查组成员综述该文档。在综述会议结束时把文档分发给每位与会者。

(2) 准备

评审员仔细阅读文档。最好列出在审查中发现的错误的类型，并按发生频率把错误类型分级，以辅助审查工作的进行。这样的列表有助于评审员们把注意力集中到最常发生错误的区域。

(3) 审查

评审组仔细走查整个文档。和走查一样，这一步的目的也是找出文档中的错误，而不是改正它们。审查组组长应该在一天之内写出一份关于审查的报告。通常，每次审查会不超过 90 分钟。

(4) 返工

文档的作者负责解决在审查报告中列出的所有错误及问题。

(5) 跟踪

审查组组长必须确保所提出的每个问题都圆满地解决了（要么修正了文档，要么澄清了被误认为是错误的条目）。必须复查对文档所做的每个修正，以确保没有引入新的错误。如果在审查过程中返工量超过 5%，则应该由审查组再对文档全面地审查一遍。

审查组通常由 4 人组成。以设计审查为例，审查组由一位组长，以及设计人员、编程人员和测试人员各 1 名组成。组长既是审查组的管理人员又是领导人员。审查组必须包括负责当前阶段开发工作的项目组代表和负责下一阶段开发工作的项目组代表。测试人员应该是负责设计测试用例的软件工程师，当然，测试人员同时又是 SQA 小组的成员则更好。在 IEEE 标准中建议审查组由 3~6 名成员组成。

审查过程不仅步数比走查多，而且每个步骤都是正规的。这种正规性体现在，仔细划分错误类型，并把这些信息运用在后续阶段的文档审查中以及未来产品的审查中。

审查是检测错误的一种好方法，用这种方法我们可以在软件开发过程的早期阶段发现错误，也就是说，能在修正错误的代价变得很昂贵之前就发现错误。因此，审查是一种强大而且经济有效的错误检测方法。

4. 程序正确性证明

测试可以暴露程序中的错误，因此是保证软件可靠性的重要手段，但是，测试只能证明程序中有错误，并不能证明程序中没有错误。因此，对于保证软件可靠性来说，测试是一种不完善的技术，人们自然希望研究出完善的正确性证明技术。一旦研究出实用的正确性证明程序（即能自动证明其他程序的正确性的程序），软件的可靠性将更有保证，测试的工作量将大大减少。但是，即使有了正确性证明程序，软件测试也仍然是需要的，因为程序正确性证明只能证明程序功能是正确的，并不能证明程序的动态特性是否符合要求的，此外，正确性证明过程本身也可能发生错误。

正确性证明的基本思想是证明程序能完成预定的功能。因此，应该提供对程序功能的严格数学说明，然后根据程序代码证明程序确实能实现它的功能说明。

在 20 世纪 60 年代初期，人们已经开始研究程序正确性证明的技术，提出了许多不同的技术方法。虽然这些技术方法本身很复杂，但是它们的基本原理却是相当简单的。

如果在程序的若干个点上，设计者可以提出关于程序变量及它们的关系的断言，那么在每一点上的断言都应该永远是真的。假设在程序的 P_1, P_2, \dots, P_n 等点上的断言分别是 $a(1), a(2), \dots, a(n)$ ，其中 $a(1)$ 必须是关于程序输入的断言， $a(n)$ 必须是关于程序输出的断言。

为了证明在点 P_i 和 P_{i+1} 之间的程序语句是正确的，必须证明执行这些语句之后将使断言 $a(i)$ 变成 $a(i+1)$ 。如果对程序内所有相邻点都能完成上述证明过程，则证明了输入断言加上程序可以导出输出断言。如果输入断言和输出断言是正确的，而且程序确实是可终止的（不包含死循环），则上述过程就证明了程序的正确性。

人工证明程序正确性，对于评价小程序可能有些价值，但是在证明大型软件的正确性时，不仅工作量太大，更主要的是在证明的过程中很容易包含错误，因此是不实用的。为了实用的目的，必须研究能证明程序正确性的自动系统。

目前已经研究出证明 PASCAL 和 LISP 程序正确性的程序系统，正在对这些系统进行评价和改进。现在这些系统还只能对较小的程序进行评价，毫无疑问还需要做许多工作，这样的系统才能实际用于大型程序的正确性证明。

7.6 软件配置管理

在开发计算机软件的过程中，变化（或称为变动）是不可避免的。如果不能适当地控制和管理变化，势必造成混乱并产生许多严重的错误。

软件配置管理是在计算机软件整个生命期内管理变化的一组活动。具体地说，这组活动用来： 标识变化； 控制变化； 确保适当地实现了变化； 向需要知道这方面信息的人报告变化。

软件配置管理不同于软件维护。维护是在软件交付给用户使用后才发生的，而软件配置管理是在软件项目启动时就开始，并且一直持续到软件退役后才终止的一组跟踪和控制活动。

软件配置管理的目标是，使变化更容易被适应，并且在必须变化时减少所需花费的工作量。

下面着重介绍控制和实现管理的方法。

任何软件开发都是迭代过程，也就是说，在设计软件时会发现需求说明中的问题，在实现过程中又会暴露出设计中的错误，如此等等。因此，变动既是必要的，又是不可避免的。但是，变动也很容易失去控制，因此，对于管理人员来说，预先计划控制变动的机制，建立评价变动的影响和把发生的变动记入文档的过程，就是十分重要的了。

变动通常可以分为两类：

第一类是为了改正小错误需要的变动，第二类是为了增加或删掉某些功能，或者为了改变完成某个功能的方法而需要的变动。第一类变动是必须进行的，通常不需要从管理角度对这类变动进行审查和批准。但是，如果发现错误的阶段在造成错误的阶段的后面（例如，在实现阶段发现了设计错误），那么使用标准的变动控制过程，把这个变动正式记入文档，是十分必要的。这样做有可能保证：所有受这个变动影响的文档，实际上都做了相应的修改。

第二类变动总是需要经过从管理角度进行的审批过程。这类变动必须经过某种正式的变动评价过程，以估计变动需要的成本和它对软件系统其他部分的影响。如果变动的代价比较小，而且对系统其他部分没有影响或影响很小，通常应该批准这个变动。反之，如果变动的代价比较高，或者影响比较大，则必须仔细权衡利弊，以决定是否进行这个变动；如果同意进行变动，还应该进一步确定由谁支付变动需要的费用。显然，如果是用户要求的改动，则用户应该支付所需要的费用；否则，必须完成某种成本 / 效益分析，以确定变动可能带来的效益是否大到值得进行这种变动的地步。应该把所做的变动正式记入文档，并且相应地修改所有有关的文档。

高级管理人员应该注意防止在工程拖期或成本超过预算时，下级管理人员或开发人员为勉强按原定计划进行而降低质量要求的倾向。

7.7 能力成熟度模型

能力成熟度模型 (Capability Maturity Model , CMM) 并不是一个软件生命周期模型，而是改进软件过程的一种策略，它与实际使用的过程模型无关。1986 年美国卡内基—梅隆大学软件工程研究所首次提出能力成熟度模型 CMM ，不过在当时它被称为过程成熟度模型。

多年来，软件开发项目不能按期完成，软件产品的质量不能令客户满意，再加上软件开发成本超出预算，这些是许多软件开发组织都遇到过的难题。近 20 年中，不少人试图通过采用新的软件开发技术来解决在软件生产率和软件质量等方面存在的问题，但效果并不令人十分满意。上述事实促使人们进一步考察软件过程，从而发现关键问题在于对软件过程的管理不尽人意。事实表明，在无规则和混乱的管理之下，先进的技术和工具并不能发挥应有的作用。人们认识到，改进对软件过程的管理是解决上述难题的突破口，再也不能忽视软件过程中管理的关键作用了。

能力成熟度模型的基本思想是，因为问题是由我们管理软件过程的方法不当引起的，所以新软件技术的运用并不会自动提高生产率和软件质量。能力成熟度模型有助于软件开发组织建立一个有规律的、成熟的软件过程。改进后的过程将开发出质量更好的软件，使更多的软件项目免受时间拖后和费用超支之苦。

软件过程包括各种活动、技术和工具，因此，它实际上既包括了软件生产的技术方面又包括了管理方面。CMM 策略力图改进软件过程的管理，而在技术方面的改进是其必然的结果。

必须记住，对软件过程的改进不可能在一夜之间完成，CMM 是以增量方式逐步引入变化的。CMM 明确地定义了 5 个不同的成熟度等级，一个软件开发组织可用一系列小的改良性步骤向更高的成熟度等级迈进。

对软件过程的改进是在完成一个一个小小的改进步骤基础之上不断进行的渐进过程，而不是一蹴而就的彻底革命。在 CMM 中把软件过程从无序到有序的进化过程分成五个阶段，并把这些阶段排序，形成五个逐层提高的等级。这五个成熟度等级定义了一个有序的尺度，用以测量软件组织的软件过程成熟度和评价其软件过程能力，这些等级还能帮助软件组织把应做的改进工作排出优先次序。成熟度等级是妥善定义的向成熟软件组织前进途中的平台，每一个成熟度等级都为过程的继续改进提供一个台阶。CMM 通过定义能力成熟度的五个等级，引导软件开发组织不断识别出其软件过程的缺陷，并指出应该做哪些改进，但是，它并不提供做这些改进的具体措施。

能力成熟度的五个等级从低到高是：初始级、可重复级、已定义级、已管理级和优化级。下面介绍能力成熟度的这五个等级。

1. 初始级

软件过程的特征是无序的，有时甚至是混乱的。几乎没有什么过程是经过定义的，项目能否成功完全取决于个人能力。

处于这个最低成熟度等级的组织，基本上没有健全的软件工程管理制度。每件事情都以特殊的方法来做。如果一个项目碰巧由一个有能力的管理员和一个优秀的软件开发组承担，

则这个项目可能是成功的。但是，通常的情况是，由于缺乏健全的总体管理和详细计划，延期交付和费用超支的情况经常发生。结果，大多数行动只是应付危机，而不是执行事先计划好的任务。处于成熟度等级 1 的组织，由于软件过程完全取决于当前的人员配备，所以具有不可预测性，人员变了过程也随之改变。因此，不可能准确地预测产品的开发时间和成本。

目前，世界上大多数软件开发公司都处于成熟度等级 1。

2. 可重复级

建立了基本的项目管理过程，以追踪成本、进度和功能性。必要的过程规范已经建立起来了，使得可以重复以前类似项目所取得的成功。

在这一级，有些基本的软件项目管理行为、设计和管理技术，是基于相似产品中的经验确定的，因此称为“可重复”。在这一级采取了一些措施，这些措施是实现一个完备过程必不可少的第一步。典型的措施包括仔细地跟踪费用和进度。不像在第一级那样，处于危机状态下才采取行动，管理人员在问题出现时可及时发现，并立即采取补救行动，以防止问题变成危机。关键是，如果没有采取这些措施，要在问题变得无法收拾前发现它们是不可能的。在一个项目中采取的措施，也可用来为未来的项目制定实现期限和费用的计划。

3. 已定义级

用于管理和工程活动的软件过程已经文档化和标准化，并且已经集成到整个组织的软件过程中。所有项目都使用文档化的、组织批准的过程来开发和维护软件。这一级包含了第 2 级的所有特征。

在这一级已经为软件过程编制了完整的文档。对软件过程的管理方面和技术方面都明确地做了定义，并按需要不断地改进软件过程。已经采用评审的办法来保证软件的质量。在这一级可采用诸如 CASE 环境之类的软件工具或开发环境来进一步提高软件质量和生产率。而在第 1 级（初始级）中，采用“高技术”只会使这一危机驱动的软件过程更混乱。

尽管有些组织已经达到了第 2 级和第 3 级成熟度，但是，迄今为止可能还没有一个组织达到第 4 级和第 5 级（个别项目或小组达到了这两级）。因此，这两个最高级还是未来的目标。

4. 已管理级

已收集了软件过程和产品质量的详细度量数据，使用这些详细的度量数据，能够定量地理解和控制软件过程和产品。这一级包含了第 3 级的所有特征。

处于第 4 级的公司为每个项目都设定质量和生产目标，并不断地测量这两个量，当偏离目标太多时，就采取行动来修正。

5. 优化级

通过定量的反馈能够实现持续的过程改进，这些反馈是从过程及对新想法和技术的测试中获得的。这一级包含了第 4 级的所有特征。

处于第 5 级的组织的目标是持续地改进软件过程。这样的组织使用统计质量和过程控制技术。从各个方面获得的知识将运用在未来的项目中，从而使软件过程进入良性循环，使生产率和质量稳步提高。

经验表明，提高一个完整的成熟度等级通常需要花 18 个月到 3 年的时间，但是从第 1 级上升到第 2 级有时要花 3 年甚至 5 年时间。这表明要向一个迄今仍处于特殊的和被动的行动方式的公司灌输系统的方式，将多么困难。

7.8 小结

软件工程包括技术和管理两方面的内容，是管理与技术紧密结合的产物。只有在科学而严格的管理之下，先进的技术方法和优秀的软件工具才能真正发挥出它们的威力。因此，软件项目管理是大型软件工程项目成功的关键。

软件项目管理从项目计划开始，而第一项计划活动就是估算。为了估算项目工作量和完成期限，首先需要预测软件规模。

度量软件规模的常用技术主要有代码行技术和功能点技术。这两种技术各有优缺点，应该根据软件项目的特点及项目计划者对这两种技术的熟悉程度，选用适用的技术。

根据项目的规模可以估算出完成项目所需的工作量，常用的估算模型有静态单变量模型、动态多变量模型和 COCOMO 模型。为了做到较准确的项目估算，通常至少同时使用上述三种模型中的两种。通过比较和协调使用不同模型得出的估算值，有可能得到比较准确的估算结果。虽然软件项目估算并不是一门精确的科学，但是，把可靠的历史数据和系统化的技术结合起来，仍然能够提高估算的准确度。

项目管理者的目标是定义所有项目任务，识别出关键任务，跟踪关键任务的进展状况，以保证能够及时发现拖延进度的情况。为此，管理者必须制定一个足够详细的进度表，以便监督项目进度并控制整个项目。

常用的制定进度计划的工具主要有 Gantt 图和工程网络两种。Gantt 图具有历史悠久、直观简明、容易学习、容易绘制等优点，但是，它不能显式地表示各项任务彼此间的依赖关系，也不能显式地表示关键路径和关键任务，进度计划中的关键部分不明确。因此，在管理大型软件项目时，仅用 Gantt 图是不够的，不仅难于做出既节省资源又保证进度的计划，而且还容易发生差错。

工程网络不仅能描绘任务分解情况及每项作业的开始时间和结束时间，而且还能显式地表示各个作业彼此间的依赖关系。从工程网络图中容易识别出关键路径和关键任务。因此，工程网络是制定进度计划的强有力的工具。通常，联合使用 Gantt 图和工程网络这两种工具来制定和管理进度计划，使它们互相补充、取长补短。

对任何软件项目而言，最关键的因素都是承担项目的人员。必须合理地组织项目组，使项目组有较高生产率。“最佳的”小组结构取决于管理风格、组里的人员数目和他们的技术水平，以及所承担的项目的难易程度。

本章具体介绍了国外比较流行的民主制程序员组、主程序员组和现代程序员组的组织方式，讨论了不同组织方式的优缺点和适用范围。

软件质量保证是在软件过程中的每一步都进行的“保护性活动”。软件质量保证措施主要有基于非执行的测试（也称为复审）基于执行的测试（即通常所说的测试）和程序正确性证明。软件复审是最重要的软件质量保证活动之一，它的作用是，在改正错误的成本相对比较低时就及时发现并排除错误。

走查和审查是进行正式技术复审的两类具体方法。审查过程不仅步数比走查多，而且每个步骤都是正规的。由于在开发大型软件过程中所犯的错误绝大多数是规格说明错误或设计错误，而

正式的技术复审发现这两类错误的有效性高达 75%，因此是非常有效的软件质量保证方法。

软件配置管理是应用于整个软件过程中的保护性活动，它是在软件整个生命期内管理变化的一组活动。

能力成熟度模型 (CMM)，是改进软件过程的一种策略。它的基本思想是，因为问题是管理软件过程的方法不恰当引起的，所以运用新软件技术并不会自动提高软件生产率和软件质量，应当下大力气改进对软件过程的管理。对软件过程的改进不可能一蹴而就，因此，CMM 以增量方式逐步引入变化，它明确地定义了 5 个不同的成熟度等级，一个软件开发组织可用一系列小的改良性步骤迈入更高的成熟度等级。

习 题 七

1. 分析本书 2.10.2 小节所述的定货系统，要求：

- (1) 用代码行技术估算本系统的规模；
- (2) 用功能点技术估算本系统的规模；
- (3) 用静态单变量模型估算开发本系统所需的工作量；
- (4) 假设由一个人开发本系统，请制定进度计划；
- (5) 假设由两个人开发本系统，请制定进度计划。

2. 分析本书习题二第 4 题中描述的机票预订系统，要求：

- (1) 用代码行技术估算本系统的规模；
- (2) 用功能点技术估算本系统的规模；
- (3) 用静态单变量模型估算开发本系统所需的工作量；
- (4) 假设由一个人开发本系统，请制定进度计划；
- (5) 假设由两个人开发本系统，请制定进度计划。

3. 你所在的信息系统开发公司指定你为项目负责人。你的任务是开发一个应用系统，该系统类似于你的小组以前做过的那些系统，不过这一个规模更大而且更复杂一些。需求已经由客户写成了完整的文档。你将选用哪种小组结构？为什么？你准备采用哪（些）种软件过程模型？为什么？

4. 你被指派作为一个小型软件产品公司的项目负责人。你的工作是开发一个技术上具有突破性的产品，该产品把虚拟现实硬件和最先进的软件结合在一起。由于家庭娱乐市场的竞争非常激烈，因此完成这项工作的压力很大。你将选择哪种小组结构？为什么？你打算采用哪（些）种软件过程模型？为什么？

5. 比较“质量”和“可靠性”的异同。

6. 一个程序能够既正确又不可靠吗？请解释你的答案。

7. 一个程序能够正确但却质量不高吗？请解释你的答案。

8. 假设要你负责改进你所在公司开发的软件产品的质量，你应该做的第一件事是什么？然后做什么？

9. 仅当每个与会者都在事先做了准备，一次正式的技术复审才是有效的。如果你是复审小组的组长，你怎样发现事先没做准备的与会者？你打算采取什么措施来保证大家事先做准备？

参考文献

1. 张海藩. 软件工程导论(第三版). 清华大学出版社.1998
2. 张海藩. 软件工程. 人民邮电出版社. 2002
3. 张海藩 , 牟永敏. 面向对象程序设计实用教程. 清华大学出版社. 2001
4. Roger S. Pressman. Software Engineering——A Practitioner's Approach , Fourth Edition , 1997
5. Stephen R. Schach. Software Engineering with Java , 1999
6. Hans Van Vliet. Software Engineering——Principles and Practice , Second Edition , 2000
7. ERIC J. BRAUDE. Software Engineering——An Object - Oriented Perspective , 2001
8. Jackson M A. Principles of Program Design. Academic Press , 1975
9. Warnier J D. Logical Construction of Programs , Van Nostrand , 1974
10. 张海藩等. 计算机第四代语言. 电子工业出版社.1996