



普通高等教育“十一五”国家级规划教材

软件工程——原理、 方法与应用 (第3版)

Software Engineering: Principles, Methods
and Applications (3rd Edition)

史济民 顾春华 郑红 编著



高等教育出版社
Higher Education Press

普通高等教育“十一五”国家级规划教材

软件工程

——原理、方法与应用（第3版）

史济民 顾春华 郑红 编著

高等教育出版社

内容提要

作为一本注重能力培养的实践型教材,第3版继承并保持了“注重实践”的风格,将全书分编为绪论和上、中、下3篇,共14章。内容分别为:上篇为传统软件工程,包括软件生存周期与软件过程、结构化分析与设计;中篇为面向对象软件工程,包括面向对象与UML、需求工程与需求分析、面向对象分析、面向对象设计、编码与测试;下篇为软件工程的近期进展、管理与环境,包括软件维护、软件复用、软件工程管理、软件质量管理、软件工程环境、软件工程高级课题。与第2版相比,本版将“并行介绍传统的和面向对象的软件工程”改变为“重点介绍面向对象的软件工程”,并在“面向对象分析”和“面向对象设计”两章分别给出案例;通过“软件工程高级课题”,对Web工程及基于体系结构的软件开发等热门课题集中进行了简要的讲解。

本书结构合理、文字通俗、例题丰富、可读性强,主要面向计算机及相关专业本科生,亦可供软件开发人员作为参考。

图书在版编目(CIP)数据

软件工程:原理、方法与应用 / 史济民,顾春华,郑红
编著. —3版. —北京:高等教育出版社,2009.3

ISBN 978-7-04-026146-2

I. 软… II. ①史…②顾…③郑… III. 软件工程-高等
学校-教材 IV. TP311.5

中国版本图书馆CIP数据核字(2009)第010893号

策划编辑 倪文慧 责任编辑 张海波 封面设计 张志奇 责任绘图 尹莉
版式设计 范晓红 责任校对 杨凤玲 责任印制 宋克学

出版发行 高等教育出版社
社址 北京市西城区德外大街4号
邮政编码 100120
总机 010-58581000

经销 蓝色畅想图书发行有限公司
印刷 高等教育出版社印刷厂

开本 787×1092 1/16
印张 24.25
字数 540 000

购书热线 010-58581118
免费咨询 800-810-0598
网址 <http://www.hep.edu.cn>
<http://www.hep.com.cn>
网上订购 <http://www.landaco.com>
<http://www.landaco.com.cn>
畅想教育 <http://www.widedu.com>

版次 1995年3月第1版
2009年3月第3版
印次 2009年3月第1次印刷
定价 28.00元

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

版权所有 侵权必究

物料号 26146-00

第3版前言

本书第2版自2002年12月首印,迄今已5年有半了。许多高等院校的计算机专业和软件学院相关专业采用该版书作为本科生“软件工程”课程的教材。为了更好地满足读者需要,我们决定编写第3版,并确定了如下的编写方针:

一、继续保持“注重实践”的风格

软件工程具有很强的实践性,但早期的软件工程教材往往偏重于理论,介绍原理有余,联系应用不足。由于缺乏可供借鉴的示例,使读者在具体开发时常有不知从何处入手的感觉。本书取名《软件工程——原理、方法与应用》,就是希望从“应用”出发,兼顾“原理”与“方法”两个方面:讲解方法时精选例题,方便模仿;上升到原理时提纲挈领,画龙点睛。目的是让读者一方面掌握软件工程的常用方法及其具体操作,另一方面又提升到以原理为指导,不致被具体方法中繁琐的细节所淹没。从本书第1版起,编者就参照国际知名教材《软件工程:实践者的方法》(Pressman 著)的做法,广举例题,注重实践,因而取得第1版连续发行12年近12万册,第2版在5年半中重印12次、累计发行14.1万册的好成绩。本版保持了上述风格,力求使之名副其实地成为理论结合实际、注重能力培养的实践型教材。

二、重点讲解面向对象软件工程

从20世纪80年代中期以来,面向对象软件工程发展迅速,主要技术已基本成熟,成为软件开发的主流范型。因此在本版编写中,将第2版“并行介绍传统的和面向对象的软件工程”,改变为“重点介绍面向对象的软件工程”。一方面,删去了Jackson与LCP等现已基本不用的方法,着重讲述以“结构化分析与设计”为代表的“结构化软件工程”技术;另一方面,把面向对象软件工程的内容由第2版的第6、7两章扩充为“面向对象与UML”、“面向对象分析”(OOA)和“面向对象设计”(OOD)等3章,并在“面向对象分析”和“面向对象设计”两章分别给出案例。为了更好反映软件工程的近期发展,除“软件复用”仍在第10章设置专章讨论外,还增编了第14章“软件工程高级课题”,对Web工程及基于体系结构的软件开发等当前的热门课题集中进行简要的讲解。

三、着重介绍软件工程技术,兼顾管理与环境

作为软件工程的综合性入门教材,本版同前两版一样主要介绍软件工程技术,兼讲一点管理与环境方面的知识,包括项目管理、项目度量、风险预测、I-CASE环境等。

四、定位与主要读者对象

经过 40 年的实践,软件工程的基本原则现已被产业界广泛接受。高等院校计算机专业普遍开设了“软件工程”课程,有些院校在非计算机专业中也设置了相关的课程。本书主要定位于计算机及相关学科各专业的本科学生,亦可供软件学院研究生及软件开发人员作为参考。全书结构合理、文字通俗、例题丰富、可读性强。但由于软件工程日新月异,新技术层出不穷,编者水平有限,疏漏难免,诚恳欢迎读者和专家批评指正。

本版由史济民主编并统稿,顾春华博士和郑红博士参与策划和编写。全书 14 章,除新编的 3 章由史济民(第 3 章)、顾春华(第 5 章)、郑红(第 14 章)分别编写外,其余 11 章也由 3 人分工修改,其中史济民改写了第 1 章,顾春华改写了第 2、4、6、7、8 等 5 章,郑红改写了第 9~13 等 5 章。杨晶、郑姜和罗珍等 3 位同志认真阅读和校对了书稿,参加了部分习题的讨论,并且从学生的角度提出不少有益的建议。

浙江大学软件学院副院长、国家级精品课程主讲人陈越教授在百忙中审查了本版书稿,并提出了宝贵的意见和建议。借此机会,表示由衷的感谢!

编 者

2008 年 6 月于华东理工大学

第2版前言

本书第一版自1990年问世以来,已经过了12年。在此期间,软件工程从第一代传统的软件工程发展为第二代面向对象的软件工程,现正向基于软件复用的第三代软件工程发展。在高等学校中,软件工程不仅已成为计算机专业学生的必修课,而且在非计算机专业的“软件技术基础”等公共课中也上升为必学的内容。

作为把软件工程推广到高校非计算机专业的一种尝试,本书第一版曾被收入高等教育出版社的“高校计算机基础教育系列教材”,但实际上多数高校仍把它用作计算机专业的教材。为了适应国内软件工程当前教学的需要,这次改版将本书仍定位为计算机专业的本科教材,并且确定了如下的编写方针。

一、继续保持“注重实践”的风格

软件工程具有很强的实践性,但由于例题难选,多数教材往往偏重于理论。本书第一版参考国际知名教材 Pressman 著的《软件工程:实践者的方法》的做法,广举实例,注重实践,因而受到读者的欢迎。第一版连续发行12年,累计印数近12万册,并于1995年荣获上海市高校优秀教材二等奖。本版保持了上述风格,使之名副其实地成为原理、方法与应用紧密结合的教材。

二、平行讲解第一、二两代软件工程

经过20世纪90年代的发展,面向对象软件工程已渐趋成熟,在许多应用领域取代了传统的软件工程。但是,传统的、基于结构化程序设计的软件工程并未退出历史舞台,其基本原理与方法有不少仍在新一代软件工程中继续应用。为此,本书将两类软件工程平行讲解,既方便读者对照,又便于展示两者“你中有我、我中有你”的关系。

三、充分反映软件工程近十几年的发展

本次改版增加了许多新方法、新模型的介绍,例如面向对象软件工程的UML语言、软件复用、质量认证标准、净室工程和软件容错技术等。对于代表新的发展方向的软件构件工程、软件过程工程等新技术,均设置专章或专节进行讲解,以突出其重要性。但鉴于本书读者对象主要是高校的本、专科学生,对于还处于形成阶段的有些新技术,本教材着重阐明它们的基本概念、产生背景和主要作用,不展开细节的讨论。

四、重点讲解软件工程技术,兼顾管理与环境

作为软件工程的综合性入门教材,本版同第一版一样主要讲解软件工程技术,兼讲一点管理与环境方面的知识。后者的重点放在质量管理与I-CASE环境上,以便读者对软件工程

的整体情况与近期发展有一比较全面的理解。

本书由史济民、顾春华共同策划。第一章由史济民改写；第二、三、四、十一、十二、十五章由顾春华改写；第五、八、九、十三章由李昌武改写。新增加的4章，分别由顾春华（第六、七章及第14.5、14.6节）、苑荣（第十章）和史济民（第十四章其余各节）编写。全书由史济民统一修改定稿。华东理工大学宋国新教授十分支持本次改版，除承担书稿审阅外，还在经费和编写人员时间安排上给予帮助。对此编者表示由衷的感谢。

由于软件工程覆盖面宽，发展又很迅速，编者水平有限，疏漏难免，诚恳希望读者不吝指正。

2002年5月于上海

郑重声明

高等教育出版社依法对本书享有专有出版权。任何未经许可的复制、销售行为均违反《中华人民共和国著作权法》，其行为人将承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。为了维护市场秩序，保护读者的合法权益，避免读者误用盗版书造成不良后果，我社将配合行政执法部门和司法机关对违法犯罪的单位和个人给予严厉打击。社会各界人士如发现上述侵权行为，希望及时举报，本社将奖励举报有功人员。

反盗版举报电话：(010) 58581897/58581896/58581879

反盗版举报传真：(010) 82086060

E-mail：dd@hep.com.cn

通信地址：北京市西城区德外大街4号

高等教育出版社打击盗版办公室

邮 编：100120

购书请拨打电话：(010) 58581118

目 录

第 1 章 绪论	1	1.3 软件工程的发展	6
1.1 软件和软件危机	1	1.3.1 3 种编程范型	7
1.1.1 软件的定义	1	1.3.2 3 代软件工程	9
1.1.2 软件的特征	2	1.4 软件工程的应用	10
1.1.3 软件危机	2	1.4.1 在各种规模软件 开发中的应用	10
1.2 软件工程学的范畴	4	1.4.2 软件工程的成就与 发展展望	12
1.2.1 软件开发方法学	5	1.5 软件工程的 _{教学} : 本书导读	13
1.2.2 软件工具	5	小结	15
1.2.3 软件工程环境	6	习题	15
1.2.4 软件工程管理	6		

上篇 传统软件工程

第 2 章 软件生存周期与软件过程	19	2.5.1 统一过程	31
2.1 软件生存周期	19	2.5.2 敏捷过程	32
2.1.1 软件生存周期的主要活动	19	2.5.3 极限编程	33
2.1.2 生存周期与软件过程的关系	20	2.6 软件可行性研究	35
2.2 传统的软件过程	21	2.6.1 可行性研究的内容与步骤	35
2.2.1 瀑布模型	21	2.6.2 软件风险分析	36
2.2.2 快速原型模型	23	2.6.3 项目实施计划	39
2.3 软件演化模型	24	小结	40
2.3.1 增量模型	24	习题	40
2.3.2 螺旋模型	25	第 3 章 结构化分析与设计	41
2.3.3 构件集成模型	27	3.1 概述	41
2.4 形式化方法模型	28	3.1.1 结构化分析与设计的由来	41
2.4.1 转换模型	28	3.1.2 SA 模型的组成与描述	43
2.4.2 净室模型	29	3.1.3 SD 模型的组成与描述	50
2.5 统一过程和敏捷过程	31	3.2 结构化系统分析	51

3.2.1	画分层数据流图	51	3.3.5	优化初始 SC 图的指导规则	62
3.2.2	确定数据定义与加工策略	54	3.3.6	教材购销系统的总体结构	63
3.2.3	需求分析的复审	55	3.4	模块设计	65
3.3	结构化系统设计	55	3.4.1	目的与任务	65
3.3.1	SD 概述	55	3.4.2	模块设计的原则与方法	66
3.3.2	SD 的步骤: 从 DFD 图 到 SC 图	57	3.4.3	常用的表达工具	68
3.3.3	变换映射	58	小结		70
3.3.4	事务映射	60	习题		71

中篇 面向对象软件工程

第 4 章	面向对象与 UML	75	小结		104
4.1	面向对象概述	75	习题		104
4.1.1	对象和类	75	第 5 章	需求工程与需求分析	105
4.1.2	面向对象的基本特征	76	5.1	软件需求工程	105
4.1.3	面向对象开发的优点	78	5.1.1	软件需求的定义	105
4.2	UML 简介	78	5.1.2	软件需求的特性	106
4.2.1	UML 的组成	79	5.1.3	需求工程的由来	107
4.2.2	UML 的特点	82	5.2	需求分析与建模	108
4.2.3	UML 的应用	83	5.2.1	需求分析的步骤	108
4.3	静态建模	83	5.2.2	需求分析是迭代过程	109
4.3.1	用例图与用例模型	84	5.3	需求获取的常用方法	110
4.3.2	类图和对象图	85	5.3.1	常规的需求获取方法	110
4.3.3	包	92	5.3.2	用快速原型法获取需求	111
4.4	动态建模	92	5.4	需求模型	112
4.4.1	消息	92	5.4.1	需求模型概述	112
4.4.2	状态图	93	5.4.2	面向对象的需求建模	114
4.4.3	时序图和协作图	95	5.5	软件需求描述	123
4.4.4	活动图	98	5.6	需求管理	123
4.5	物理架构建模	99	5.6.1	需求管理的内容	124
4.5.1	物理架构	99	5.6.2	需求变更控制	125
4.5.2	构件图和部署图	100	5.6.3	需求管理工具	128
4.6	UML 工具	101	5.7	需求建模示例	128
4.6.1	Rational Rose	101	5.7.1	问题陈述	129
4.6.2	StarUML	103	5.7.2	用例模型	130

5.7.3 补充规约	134	7.3.3 任务管理策略	179
5.7.4 术语表	135	7.3.4 分布式实现机制	182
小结	136	7.3.5 数据存储设计	185
习题	136	7.3.6 人机交互设计	188
第 6 章 面向对象分析	138	7.4 系统元素设计	189
6.1 软件分析概述	138	7.4.1 子系统设计	189
6.1.1 面向对象软件分析	138	7.4.2 分包设计	192
6.1.2 面向对象分析模型	140	7.4.3 类/对象设计	195
6.2 面向对象分析建模	141	7.5 面向对象设计示例	204
6.2.1 识别与确定分析类	141	7.5.1 关联关系的具体化	205
6.2.2 建立对象-行为模型	144	7.5.2 网上购物系统的 架构设计	206
6.2.3 建立对象-关系模型	147	7.5.3 网上购物系统的类/ 对象设计	207
6.3 面向对象分析示例	149	小结	210
6.3.1 注册	149	习题	211
6.3.2 维护个人信息	151	第 8 章 编码与测试	212
6.3.3 维护购物车	153	8.1 编码概述	212
6.3.4 生成订单	157	8.1.1 编码的目的	212
6.3.5 管理订单	159	8.1.2 编码的风格	213
小结	164	8.2 编码语言与编码工具	216
习题	164	8.2.1 编码语言的发展	216
第 7 章 面向对象设计	165	8.2.2 常用的编程语言	218
7.1 软件设计概述	165	8.2.3 编码语言的选择	220
7.1.1 软件设计的概念	165	8.2.4 编码工具	222
7.1.2 软件设计的任务	167	8.3 编码示例	222
7.1.3 模块化设计	167	8.3.1 注册功能编码实现	222
7.2 面向对象设计建模	173	8.3.2 维护购物车功能编码实现	228
7.2.1 面向对象设计模型	173	8.4 测试的基本概念	233
7.2.2 面向对象设计的任务	174	8.4.1 目的与任务	233
7.2.3 模式的应用	175	8.4.2 测试的特性	234
7.3 系统架构设计	176	8.4.3 测试的种类	235
7.3.1 系统高层结构设计	176	8.4.4 测试的文档	235
7.3.2 确定设计元素	177	8.4.5 软件测试过程	236

8.5 黑盒测试和白盒测试	236	8.7.3 集成测试	256
8.5.1 黑盒测试	237	8.7.4 确认测试	259
8.5.2 白盒测试	240	8.7.5 系统测试	260
8.6 测试用例设计	248	8.7.6 终止测试的标准	260
8.6.1 黑盒测试用例设计	248	8.8 面向对象系统的测试	261
8.6.2 白盒测试用例设计	250	8.8.1 OO 软件的测试策略	261
8.7 多模块程序的测试策略	254	8.8.2 OO 软件测试用例设计	262
8.7.1 测试的层次性	254	小结	265
8.7.2 单元测试	255	习题	265
下篇 软件工程的近期进展、管理与环境			
第 9 章 软件维护	271	10.4 面向对象与软件复用	292
9.1 软件维护的种类	271	10.4.1 OO 方法对软件复用的支持	292
9.2 软件可维护性	272	10.4.2 复用技术对 OO 方法的支持	293
9.3 软件维护的实施	274	10.4.3 基于构件软件开发的现状与问题	295
9.4 软件维护的管理	275	小结	295
9.5 软件配置管理	277	习题	296
9.6 软件再工程	279	第 11 章 软件工程管理	297
小结	281	11.1 管理的目的与内容	297
习题	282	11.2 软件估算模型	298
第 10 章 软件复用	283	11.2.1 资源估算模型	298
10.1 软件复用的基本概念	283	11.2.2 COCOMO 模型	300
10.1.1 软件复用的定义	283	11.3 软件成本估计	302
10.1.2 软件复用的措施	284	11.4 人员的分配与组织	308
10.1.3 软件复用的粒度	285	11.5 项目进度安排	311
10.2 领域工程	286	小结	314
10.2.1 横向复用和纵向复用	286	习题	315
10.2.2 实施领域分析	287	第 12 章 软件质量管理	316
10.2.3 开发可复用构件	287	12.1 从质量保证到质量认证	316
10.2.4 建立可复用构件库	289	12.2 质量保证	317
10.3 基于构件的软件开发	290		
10.3.1 构件集成模型	291		
10.3.2 应用系统工程	291		

12.2.1 软件的质量属性	317	13.2.1 CASE 的组成构件	342
12.2.2 质量保证的活动内容	318	13.2.2 CASE 的一般结构	343
12.3 软件可靠性	319	13.3 CASE 环境实例	345
12.3.1 可靠性的定义和分级	319	13.3.1 Rational SUITE Enterprise Studio	346
12.3.2 可靠性模型	321	13.3.2 青鸟系统	351
12.3.3 软件容错技术	323	小结	351
12.4 程序正确性证明	326	习题	352
12.5 CMM 软件能力成熟度模型	328	第 14 章 软件工程高级课题	353
12.5.1 CMM 的基本概念	328	14.1 Web 工程	353
12.5.2 软件能力成熟度等级	329	14.1.1 Web 工程与软件工程	354
12.5.3 CMM 的应用	330	14.1.2 Web 开发	354
12.5.4 CMM 评估的实施	331	14.2 基于体系结构的软件开发	358
12.5.5 软件过程评估的 SPICE 国际标准	331	14.2.1 应用软件的体系结构	359
12.6 ISO 9000 国际标准	332	14.2.2 编程范型对体系 结构的影响	360
12.6.1 ISO 9001 和 ISO 9000-3	332	14.2.3 编程范型对复用 粒度的影响	360
12.6.2 ISO 9000 标准对软件 企业的重要性	333	14.2.4 软件体系结构技术 仍在发展	360
12.6.3 在软件企业中实施 ISO 9000 标准	334	14.3 面向方面的软件开发	361
12.7 软件度量	334	14.3.1 面向方面编程	361
12.7.1 项目度量	334	14.3.2 AOP 语言规范	362
12.7.2 过程度量	336	14.3.3 AOP 与 OOP 比较	363
小结	337	14.3.4 面向方面软件开发	364
习题	338	14.4 形式化的软件开发	364
第 13 章 软件工程环境	339	14.4.1 形式化方法的定义	365
13.1 什么是软件工程环境	339	14.4.2 形式化的软件开发	366
13.1.1 软件开发环境的特点	339	小结	368
13.1.2 理想环境的模型	341	习题	369
13.1.3 CASE 环境	341		
13.2 CASE 环境的组成与结构	342		
附录 缩略语中英文对照表	370		
主要参考文献	372		

第1章 绪论

1969年，美国IBM公司首次宣布除操作系统继续随计算机配送外，其余软件一律计价出售，由此开创了软件成为独立商品的先河。短短30多年间，计算机软件的重要性与日俱增。从PC到笔记本式计算机，从Internet到移动电话，从先进的武器到现代家电，计算机软件几乎无处不在，无时不在。世界上最大的软件公司Microsoft及其创始人，已分别成为全球知名度最高的企业和人物之一。在很多发达国家，软件产业已成为社会的支柱产业，软件工程师也成为最受青睐的一种职业。

随着软件产业的发展，计算机应用逐步渗透到社会生活的各个角落，使各行各业都发生了很大的变化。这同时也促使人们对软件的品种、数量、功能和质量等提出了越来越高的要求。然而，软件的规模越大、越复杂，软件开发越显得力不从心。于是，业界开始重视软件开发过程、方法、工具和环境的研究，软件工程应运而生。

本章介绍软件和软件工程的基本概念，包括软件、软件危机、软件工程学、三代软件工程及其应用等。章末讨论与软件工程教学有关的几种观点，也可视为本书的导读。

1.1 软件和软件危机

和计算机硬件一样，20世纪60年代以来，软件也在规模、功能等方面得到了很大的发展，同时人们对软件质量的要求也越来越高。那么，究竟什么是软件，它有哪些主要特征呢？

1.1.1 软件的定义

众所周知，计算机硬件的发展基本上遵循了摩尔定律，即每18个月芯片的性能与速度均提高一倍。软件的发展也十分惊人，例如就体系结构而言，它经历了从主机结构到文件服务器结构，从客户/服务器系统到基于Internet的服务器/浏览器结构的体系结构等变化；从编码语言来讲，它经历了从机器代码到汇编代码，从高级程序设计语言到人工智能语言等变化；从开发工具来看，它经历了从分离的开发工具（如代码编辑器、中间代码生成器和连接器）到集成的可视化开发系统，从简单的命令行调试器到方便的多功能调试器等变化。

但是在过去40余年中，软件的基本定义却并未改变。有些初学者认为软件就是程序，这个理解是不完全的。这里引用著名的美国软件工程教材作者R. S. Pressman的定义：“软件

是能够完成预定功能和性能的可执行的计算机程序，包括使程序正常执行所需要的数据，以及有关描述程序操作和使用的文档。”简而言之，可以表述为“软件 = 程序 + 文档”。

程序是为了解决某个特定问题而用程序设计语言描述的适合计算机处理的语句序列。它是由软件开发人员设计和编码的，通常要经过编译程序，才能编译成可在计算机上执行的机器语言指令序列。程序执行时一般要输入一定的数据，同时也会输出运行的结果。而文档则是软件开发活动的记录，主要供人们阅读，既可用于专业人员和用户之间的通信和交流，也可以用于软件开发过程的管理和运行阶段的维护。为了提高软件开发的效率和方便软件产品的维护，现在软件开发人员越来越重视文档的作用及其标准化工作。我国国家标准局已参照国际标准，陆续颁布了《计算机软件开发规范》、《计算机软件需求说明编制指南》、《计算机软件测试文件编制规范》、《计算机软件配置管理计划规范》等文档规范。

1.1.2 软件的特征

要对软件有一个全面的理解，首先要了解软件的特征。当制造硬件时，生产的结果能转换成物理的形式。如果建造一台新的计算机，从设计图纸、生产部件（VLSI 芯片、线路板、面板等）到装配原型，每一步都将演化成物理的产品。而软件却是逻辑的而不是物理的，在开发、生产、维护和使用等方面，都同硬件具有完全不同的特征。

1. 软件开发不同于硬件设计
与硬件设计相比，软件更依赖于开发人员的业务素质、智力，以及人员的组织、合作和管理，而硬件设计与人的关系相对小一些。对硬件而言，设计成本往往只占整个产品成本的一小部分，而软件开发的成本很难估算，通常占整个产品成本的大部分，这意味着对软件开发项目不能像硬件设计项目那样来管理。

2. 软件生产不同于硬件制造
硬件设计完成后就投入批量制造，制造也是一个复杂的过程，其间仍可能引入质量问题；而软件成为产品之后，其制造只是简单的复制而已。

3. 软件维护不同于硬件维修
硬件在运行初期有较高的故障率（主要来源于设计或制造的缺陷），在缺陷修正后的一段时间中，故障率将下降到一个较低和稳定的水平上。随着时间的推移，故障率会再次升高，这是因为硬件将受到磨损等损害，达到一定程度后就应该报废。软件是逻辑的而不是物理的，虽然不会磨损和老化，但在使用过程中的维护却比硬件复杂得多。如果软件内部的逻辑关系比较复杂或规模比较大，在维护过程中很可能产生新的错误。

1.1.3 软件危机

随着计算机应用的逐步扩大，软件需求量迅速增加，规模也日益增长。长达数万行、数十万乃至百万行以上的软件，已不鲜见。美国阿波罗登月计划的软件长 1 000 万代码行，航天飞机软件长达 4 000 万行，就是两个突出的例子。

软件规模的增长，带来了它的复杂度的增加。如果说编写一个数十到数百行的程序连初学者也不难完成，那么开发一个数万以至数百万行的软件，其复杂度将大大上升，即使是富有经验的开发人员，也难免顾此失彼。其结果是，大型软件的开发费用经常超出预算，完成时间也常常脱期。尤其糟糕的是，软件可靠性往往随规模的增长而下降，质量保证也越来越困难。

众所周知，任何计算机系统均由硬件、软件两部分组成。在计算机应用早期，软件仅包含少量规模不大的程序，应用部门花费在软件上的投资（成本）仅占很小的份额。随着应用的不断扩大，软件的花费越来越大，所占的百分比也越来越高。B. Boehm 在 1973 年发表的一篇文章中预期，到 1985 年，美国空军的软件费用将上升到计算机总费用的 90%（参阅图 1.1）。即在每 100 元用于计算机投资总额中，软件将花费 90 元。这一预期早已为实践所证实。

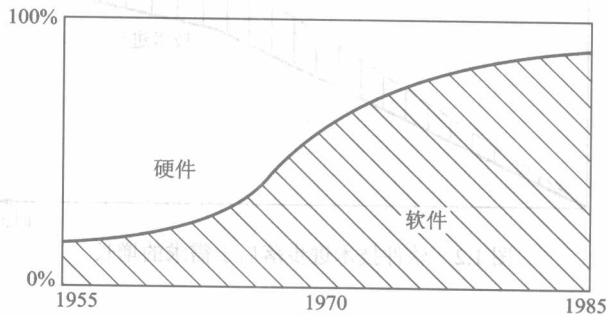


图 1.1 硬件/软件成本变化趋势

庞大的软件费用，加上软件质量的下降，对计算机应用的继续扩大构成了巨大的威胁。面对这种严峻的形势，软件界的有识之士发出了软件危机的警告。

以下再从维护和生产两个方面，进一步说明出现软件危机的原因。

① 软件维护费用急剧上升，直接威胁计算机应用的扩大。

根据一些大公司的统计，软件维护费用大约占软件总花费的 2/3，比开发费用高出一倍。一个大型软件，即使在开发时经过严格的测试与纠错，也不能保证运行中不再出现错误。维护的第一件事，就是纠正软件中遗留的错误，称为“纠错性维护”。在此后的运行过程中，还常常要为完善功能、适应环境变更等原因对软件进行修改，即所谓“完善性维护”和“适应性维护”（详见第 9 章）。不言而喻，软件的规模愈大，以上各种维护的成本必然愈高。

维护既耗费财力，也耗费人力，为了维护，要占用计算机厂家或软件公司许多软件人员，使他们不能参加新软件的开发。难怪有些文献把维护比作冰海中横在前进航道上的冰山，或直称之为维护墙（maintenance wall），将其视为软件生产和维护中难以逾越的障碍。

② 软件生产技术进步缓慢，是加剧软件危机的重要原因。

有人统计，硬件的性能价格比在过去 40 余年中增长了 10^6 。一种新器件的出现，其性能较旧器件提高，价格反而有所下降，这就是微电子技术创造的奇迹。软件则相形见绌。一方面，软件规模与复杂度增长了几个数量级，但生产方式长期未突破手工业的方式，创建新软件的能力提高得十分缓慢（参阅图 1.2）；另一方面，很多在早期用“自由化”方法开发的、带有很强“个人化”特征的程序，因缺乏文档而根本不能维护，更加剧了供需之间的矛盾。结构化程序设计的出现，使许多产业界人士认识到必须把软件生产从个人化方式改变为工程化方式，从而促使了软件工程的诞生。

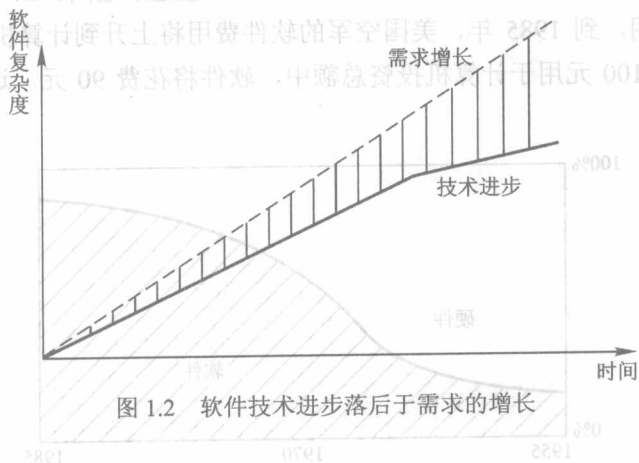


图 1.2 软件技术进步落后于需求的增长

1.2 软件工程的范畴

“软件工程”一词，首先是 1968 年北大西洋公约组织（NATO）在联邦德国召开的一次会议上提出的。它反映了软件人员认识到软件危机的出现，以及为谋求解决这一危机而做的一种努力。

人们曾从不同的角度，给软件工程下过各种定义。但是不论有多少种说法，它的中心思想都是把软件当作一种工业产品，要求“采用工程化的原理与方法对软件进行计划、开发和维护”。这样做的目的，不仅是为了实现按预期的进度和经费完成软件生产计划，也是为了提高软件的生产率与可靠性。

40 年来，人们围绕着实现软件优质高产这个目标，从技术到管理做了大量的努力，形成了“软件工程”这一计算机新学科。图 1.3 列举了它所包含的主要内容。

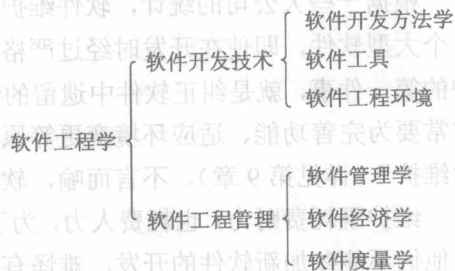


图 1.3 软件工程的范畴

1.2.1 软件开发方法学

软件的发展,大体上经历了程序、软件和软件产品 3 个阶段。早期的程序规模较小,随着系统程序的增加,人们把程序区分为系统程序和应用程序,并且通常将前者称为软件。但是,无论是软件或程序,在开发过程中都很少考虑到对它们的维护。只是当软件工程兴起之后,人们才把软件视为产品,强调软件的可维护性,同时确定了各个开发阶段必须完成的文档(documents)。

与上述 3 个发展阶段相对应,软件开发方法也发生了巨大变化。早期的程序设计基本上属于个人活动性质,开发人员各行其是,并无统一的方法可循。20 世纪 60 年代后期兴起的结构程序设计,使人们认识到采用结构化的方法来编写程序,不仅可以改善程序的清晰度,而且也能提高软件的可靠性与生产率。随后,人们又认识到编写程序仅是软件开发过程中的一个环节,有效的开发还应包括需求分析、软件设计、编码等多个阶段。把结构化的思想扩展到分析阶段和设计阶段,于是形成了结构化分析与结构化设计等传统的软件开发技术。与此同时,也出现了一些从不同基点出发的软件开发方法,如 Jackson 方法、LCP 方法等。尽管这些方法的具体内容各有不同,但它们都遵循结构化程序设计的原则,都对软件开发步骤和文档格式提出了规范化的要求。软件生产已经摆脱了过去随心所欲的个人化的状态,进入了有章可循的、向结构化和标准化迈进的工程化阶段。

20 世纪 80 年代出现的 Smalltalk、C++ 等语言,促进了面向对象程序设计的广泛流行。但是,人们继而发现,仅仅使用面向对象程序设计不会产生最好的效果。只有在软件开发的早期乃至全过程都采用面向对象技术,才能更好地发挥该技术的固有优势。于是,包括“面向对象需求分析—面向对象设计—面向对象编码”在内的软件开发方法学开始形成,并且逐步地取代了传统的软件开发方法,成为许多软件工程师的首选方法。面向对象技术还促进了软件复用技术的发展。复用加快了开发速度,开发又产生了更多的可复用软件构件,使软件复用最终成为软件开发方法学的一个重要组成部分。

1.2.2 软件工具

“工欲善其事,必先利其器”,人类早就认识到工具在生产过程中的重要作用。伴随着软件开发的发展,也研制出了众多“帮助开发软件的软件”,人们称之为软件工具(software tools)。它们对提高软件生产率,促进软件生产的自动化都有重要的作用。

设想在 PC 上用 Pascal 语言开发一个应用软件的过程。首先,要在编辑程序支持下把源程序输入计算机。然后调用 Pascal 编译程序,把源程序翻译成目标程序。如果发现错误,就重新调入编辑程序对源程序进行修改。编译通过后,再调用连接程序把所有通过编译的目标程序同与之有关的库程序连接起来,构成一个能在计算机上运行的可执行软件。在这里,编译程序、编辑程序、连接程序以及支持它们的计算机操作系统,都属于软件工具。离开了这些工具,软件开发就失去了支撑,将十分困难和低效,甚至不能工作。

以上提到的,仅是在编码阶段常用的一些软件工具。在开发的其余阶段,例如分析阶段、设计阶段和测试阶段,也研制了许多有效的工具。众多的工具组合起来,可以组成工具箱(tool box)或集成工具(integrated tool),供软件开发人员在不同的阶段按需选用。

1.2.3 软件工程环境

工具和方法,是软件开发技术的两大支柱,它们密切相关。当一种方法提出并证明有效后,往往随之研制出相应的工具,来帮助实现和推行这种方法。新方法在推行初期,总有人不愿接受和采用。若将新方法融入工具之中,使人们通过使用工具来了解新方法,就能更有效地促进新方法的推广。

方法与工具相结合,再加上配套的软、硬件支持就形成环境。创建适用的软件工程环境(software engineering environment, SE²),一直是软件工程研究中的热门课题。

为了说明软件开发对环境的依赖,不妨回顾一下分时系统所产生的影响。在批处理时代,用户开发的程序是分批送入计算中心的计算机的,有了错误,就得下机修改。软件开发人员对自己编写的程序只能断续跟踪,思路经常被迫中断,效率难于提高。分时系统的使用,使开发人员能在自己的终端上跟踪程序的开发,仅此一点,就明显提高了开发的效率。近30年来出现的UNIX环境、Windows环境,以及形式繁多的网络环境等,不仅反映了人们创造良好软件环境的努力,也把对软件工程环境的研究提升到一个新的高度。

1.2.4 软件工程管理

在工业生产中,即使有先进的设备与技术,管理不善的企业也不能获得良好的经济效益。不少软件在生产过程中不能按质、按时完成计划,管理混乱往往是其中的重要原因。可惜的是,至今软件管理尚未获得普遍的重视。

软件工程管理的目的,是为了按照进度及预算完成软件计划,实现预期的经济和社会效益。它包括成本估算、进度安排、人员组织和质量保证等多方面的内容,还涉及管理学、度量学和经济学等多个学科方面的知识。

显然,软件管理也可以借助计算机来实现。一些帮助管理人员估算成本、制定进度和生成报告等工具现在已研制出来。一个理想的软件工程环境,应该同时具备支持开发和支持管理两个方面的工具。

1.3 软件工程的发展

自1968年首次提出软件工程概念以来,已经40年了。在这一时期中,编程范型(programming paradigm)已经经历了3次演变,软件工程也从第一代发展到了第三代。本节将对此作概略的说明。

1.3.1 3种编程范型

计算机应用离不开编写程序。按不同的思路和方法来编写程序，就形成不同的编程范型。1956年，世界上第一个高级语言 FORTRAN 问世。50多年来，高级语言的编程范型大体经历了3次演变，即过程式编程范型、面向对象编程范型与基于构件技术的编程范型。

1. 过程式编程范型

过程式编程范型遵循“程序 = 数据结构 + 算法”的思路，把程序理解为由一组被动的数据和一组能动的过程所构成。编程时，先设计数据结构，再围绕数据结构编写其算法过程。程序运行后，获得预期的计算结果或正确的操作，借以满足程序的功能需求。对于比较大的程序，必须先进行功能分解，把较小的功能编写为子程序，子程序再调用子子程序，直至最终将程序做成由一組组大小适中、层层调用的模块所构成的应用程序系统。

从20世纪50年代至80年代后期，过程式编程范型广泛流行了30余年。FORTRAN、Pascal 和 C，都是当时常用的面向过程编码语言（procedure-oriented programming language, POPL），它们集中展示了这种编程范型的特点。

在客观事物中，实体的内部“状态”（一般用数据表示）和“运动”（施加于数据的操作）总是结合在一起的。可是在用 POPL 编码时，程序模型（称为解空间，solution domain）却被人多地构造为偏离客观实体本身的模型（称为问题空间，problem domain）。随着程序规模的扩大，这类编程范型的缺陷越来越明显，在错综复杂的调用下，即使功能可以满足，性能也不容易满足，使程序难于维护和移植。因此，这类范型通常只用于编写代码在50 000行以下、不会轻易更改的应用程序。

2. 面向对象编程范型

在面向对象的程序设计中，数据及其操作被封装在一个个称为对象（object）的统一体中，对象之间则通过消息（message）相互联系，“对象+消息”的机制取代了“数据结构+算法”的思路，因而较好地实现了解空间与问题空间的一致性，从而为解决软件危机带来了新的希望。从面向过程程序设计到面向对象程序设计（object-oriented programming, OOP），是程序设计方法的又一次飞跃，目前正日益显露出其优越性。

为便于对照，请读者先看一个简单的例子——银行储蓄处理事务。这一事务包含一个数据（账户余额）和3个对数据的操作——存款、取款与利息结算（每年度一次）。图1.4显示了使用两类不同编程范型实现这一事务的模型。

在采用过程式编程范型中，如图1.4（a）所示，数据“账户余额”与施加在其上的操作是分离的，存款、取款与利息结算等3个过程模块分别将相应的操作施加于“账户余额”，使数据获得更新。图1.4（b）则采用面向对象编程范型，存款、取款与利息结算作为对象内部的3个事件过程，与“账户余额”一起封装在“银行账户”对象中，通过来自对象外部的3种不同的消息（如图1.4（b）中箭头所示），即可启动相应的操作。

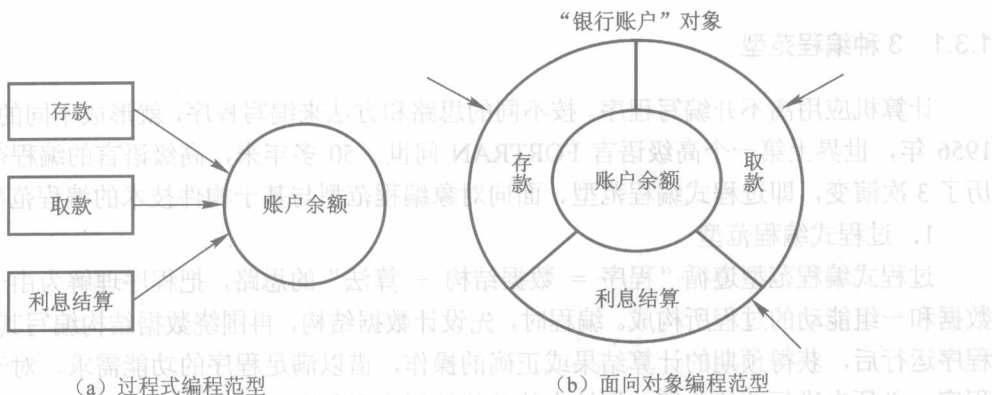


图 1.4 用两类不同编程范型实现银行储蓄处理事务

两者对照，面向对象编程范型具有明显的优势。首先，在图 1.4 (a) 中，存款、取款与利息结算都是分离的单个模块，而在图 1.4 (b) 中，“银行账户”对象一般是由多个模块组成的一个独立的单元。而且由于解空间与问题空间的一致性，软件开发人员对客观世界建立的分析模型，实际上已为软件系统的设计准备好了基本的框架。因而在大型程序的开发中，采用面向对象编程范型可以有效地降低软件的复杂性，简化软件的开发。

面向对象编程范型的优势也体现在软件的维护上。假定图 1.4 (a) 中的“账户余额”在开发时使用了整型数，维护时要改变为浮点数，则该软件中凡与“账户余额”有关的程序均需修改。但如果采用图 1.4 (b) 中的面向对象编程范型，则上述改变将限于“银行账户”对象本身，与该对象以外的其他部分以及触发该对象操作的外部消息完全无关。这将使因修改程序而引发软件故障的机会大大减少，使得大型软件的维护更加容易和快捷。

3. 基于构件技术的编程范型

正当过程式编程范型渐渐过时、逐步被方兴未艾的面向对象编程范型取而代之的时候，一种更先进的编程范型已悄悄地进入软件开发人员的视线，这就是基于构件技术的编程范型。

简言之，构件（component，有些文献翻译为组件）可以理解为标准化（或者规格化）的对象类（参见 2.3.3 节）。它本质上是一种通用的、可支持不同应用程序的组件，正如硬件中的标准件一样，插入不同的平台或环境后即可直接运行。

值得指出的是，基于构件的开发技术（component-based development, CBD）与面向对象技术其实是一脉相承的。它给软件开发人员带来了根本的变革，就是把面向特定应用的 OO 编程，扩展为面向整个“领域”（domain）的 CBD 编程，使查找与集成适合于所需领域的构件成为这一新编程范型的主要工作。由此可见，CBD 实际上是 OO 开发的延伸与归宿。现代的网络应用程序，几乎普遍采用基于构件技术的编程范型（参见第 10 章）。

4. 3 种编程范型的比较

常用编程粒度的大小来比较 3 种编程范型的差异。

过程式编程范型：着眼于程序的过程和基本控制结构，粒度最小。

面向对象编程范型：着眼于程序中的对象，粒度比较大。

基于构件技术的编程范型：着眼于适合整个领域的类对象，粒度更大。

由上可见，编程范型的演变是伴随着编程粒度的扩大而推进的，这也标志着软件开发技术的不断成熟。随着 CBD 的规范化与市场化，软件开发现已进入一个新阶段，开辟了软件应用的新纪元。

1.3.2 3代软件工程

1. 从编程范型到软件开发过程

前已指出，程序编码与软件开发过程是内涵不同的两个概念，但二者又密切联系，相互对应。编写程序仅是软件开发过程的一部分内容，在整个软件开发过程中，通常包括需求分析、软件设计、程序编码、软件测试等多个阶段。但程序编码是建立在编程范型的基础之上的，有什么样的编程范型，就将对应什么样的软件开发过程。以过程式编程范型为例，它采用 POPL，遵循以“清晰第一、效率第二”为目标的结构化程序设计的思想与方法，因而在软件开发过程中，也相应地采用以结构化分析与结构化设计为代表的结构化开发模型。与此相似，面向对象编程范型将对应于以 OO 分析及 OO 设计为代表的面向对象软件开发模型；基于构件技术的编程范型将对应于以领域分析与领域设计为代表的构件集成模型（参见第 2.3.3 节），从而形成了软件工程的分代。

2. 软件工程的分代

经过了 40 年的发展，软件工程已经历了以下 3 代：

(1) 传统软件工程或经典软件工程

它以结构化程序设计为基础，又可区分为瀑布模型、原型模型等，其开发过程一般包括以下阶段：

结构化分析 → 结构化设计 → 面向过程的编码 → 软件测试

(2) 面向对象软件工程

它以面向对象程序设计为基础，其开发过程可包括以下阶段：

OO 分析与对象抽取 → 对象详细设计 → 面向对象的编码与测试

(3) 基于构件的软件工程

它以软件复用为目标、领域工程为基础，其开发过程一般包括以下阶段：

领域分析和测试计划定制 → 领域设计 → 建立可复用构件库 → 按“构件集成模型”查找与集成构件

图 1.5 以简明的图示对照 3 代软件工程的比较。

由图 1.5 可见，在面向对象软件工程中，为软件确定（或抽取）类/对象的工作通常提前到分析阶段进行，在设计阶段主要完成对象内部的详细设计。随着测试工具的大量应用，在详细设计一个对象时，常常先对其关键算法和重要接口进行临时的编码和测试。当验证确认

可行后再纳入设计，临时代码稍微修改即可转换为正式代码，因而编码和测试并无明显的先后。可见在面向对象的软件工程中，从一个阶段过渡到另一个阶段，比传统的软件工程更加平滑，从而也降低了开发过程中的故障率。

- 传统软件工程
结构化分析 → 结构化设计 → 面向过程的编码 → 软件测试
- 面向对象软件工程
OO 分析与对象抽取 → 对象详细设计 → 面向对象的编码与测试
- 基于构件的软件工程
领域分析和测试计划定制 → 领域设计 → 建立可复用构件库 → 按“构件集成模型”查找与集成构件

图 1.5 3 代软件工程的简单比较

基于构件的软件工程是以大量的可复用构件（在应用软件中可达 80% 以上）和测试工具为后盾的。在领域分析阶段，不仅要定义领域的体系结构，还要定义初步的用户界面。这些都需要借助测试来判断能否满足用户的需求。因此在基于构件的软件工程中，测试将进一步提前到需求阶段。换言之，软件刚刚开发，需求初步定义之后，测试工作也就开始了。

1.4 软件工程的应用

软件工程已提出和宣传多年，但真正获得广泛应用的时间并不太长。本节将首先讨论它在不同规模软件开发中的应用，然后简述软件工程迄今取得的成就和今后发展展望。

1.4.1 在各种规模软件开发中的应用

软件开发在技术和管理两个方面的复杂程度，均与软件的规模密切相关。越是规模大的软件，越要在开发和维护中严格遵守软件工程的原则和方法。表 1.1 列出了软件规模的分类。以下将按照这一分类，分别说明软件工程方法在各种规模软件生产中的指导作用。

表 1.1 软件规模的分类表

分类	程序规模	子程序数	开发时间	开发人数
极小	500 行以下	10~20	1~4 周	1 人
小	1K~2K 行	25~50	1~6 月	1 人
中	5K~50K 行	250~1 000	1~2 年	2~5 人
大	50K~100K 行		2~3 年	5~20 人
甚大	1M 行		4~5 年	100~1 000 人
极大	1M~10M 行		5~10 年	2 000~5 000 人

1. 中、小型程序

包括表 1.1 中的前半部分：极小、小和中 3 类。

极小程序大都为个人软件，由个人开发和使用，且常常只用几个月就废弃了。这类程序一般不需要正式的分析 and 详细的设计文档，也不必制定完整的测试计划。但即使是这类极小的程序，如能在开发中做一点分析和系统设计，遵守结构化编码和合乎规范的测试方法，对提高程序质量和减少返工，仍会有不小的帮助。

小程序包括工程师们用于求解数值问题的科学计算程序，数据处理人员生成报表或完成数据操作所用的小型商业应用程序，以及大学生们在编译原理或操作系统等课程设计中编写的程序。这类程序的长度一般不超过 2 千行，与其他外部程序也没有什么联系。开发者通常仅有一人，无须或很少需要和用户或其他开发人员打交道。在开发这类程序时，应贯彻软件工程中的技术标准和表示方法(notations)，按标准编写文档，并系统地进行复审。当然，上述工作的正规程度不必像开发大程序时那样严格。

中规模程序包括汇编程序、编译程序、小型 MIS 系统、仓库系统以及用于过程控制的一些应用程序。这类程序可能与其他程序有少量联系，也可能没有。但是在开发过程中，开发人员与用户间或开发人员之间均存在一定的联系。所以在制定软件计划、编制文档、进行阶段复审等方面，正规化的要求都比较高。在开发中如能系统地应用软件工程的原理，对改进软件质量、提高开发人员生产率和满足用户需求，都将有很大帮助。

有些人以为，软件工程适用于开发大型软件，对开发规模小的软件就缺少用武之地了，这其实是误解。许多系统软件和大多数应用软件都属于中、小型软件。如上所说，它们的开发都需要软件工程作指导。

2. 大型程序

包括表 1.1 中的后半部分：大、甚大和极大 3 类。

大型编译程序、小型分时系统、数据库软件包以及某些图形软件和实时控制系统等，都是大型软件的实例。它们的编码长度可达 5 万至 10 万行，且通常与其他程序或软件系统有种种联系。开发人员一般由几个开发小组（如 3 个小组、每组 5 人）组成，在组与组间、组内不同成员间、开发人员同管理人员及用户之间，都存在着大量的通信。在长达两三年的开发过程中，中途离开或增加部分开发人员，也是常有的事。

长达百万行的软件称为甚大型软件，常见于实时处理、远程通信和多任务处理等应用领域。例如，大型的操作系统和数据库系统，军事部门的指挥与控制系统，等等。IBM/360 系列的操作系统，就是一个拥有 100 万行源程序的甚大型软件，其开发历时 5 年，参加开发人员达 5 000 人之多。今天，典型的 PC 上 Windows 操作系统的规模已增长到上百万行，不少 Windows 应用软件都包含若干个子系统，每一子系统均构成一个大型软件。

再以 2007 年我国发射的“嫦娥一号”月球卫星为例，为了在长达一年的绕月飞行中，

对离地 38 万千米的月球卫星实现长期管理,北京飞行控制中心开发了一个软件,统一调度各相关测控站对卫星的跟踪与测量,并通过对遥测数据的分析和计算,判断卫星在太空中的姿态、位置和星载设备的工作状况。即使在卫星携带的“星际计算机”与地面失去联系时,也能通过“自主管理”维持卫星有效地工作。这一甚大型软件包含了实时处理、长期管理、轨道控制、数据存储、指挥显示、数字仿真、国际联网等 7 个子系统,与星际计算机一起构成“嫦娥一号”的“大脑”和“中枢神经”。整个飞行控制软件共有 104 万代码行,而为了确保这些软件正确运行,还另编了 160 万行的测试程序。

极大型软件通常由数个甚大型的子系统构成,常含有实时处理、远程通信、多任务处理以及分布处理等软件。空中交通管制系统、洲际导弹防御系统以及某些军事指挥和控制系统,都是极大型软件的实例。它们的源代码往往长达数百万至数千万行,开发周期可能长达 10 年,并要求有极高的软件可靠性。

毫无疑问,所有大型以上软件的生产必须自始至终采用软件工程的方法,严格遵守标准文档格式和正规的复审制度。而且,由于大型软件本身的复杂性,对它们的计划和分析很难做到一劳永逸。因此,要十分重视管理工作,坚决贯彻软件工程关于软件管理的要求,才能避免或减少混乱。

由上可见,从极小程序到极大程序,软件工程都有它的用武之地。软件生产中是否采用工程化的方法,其结果将明显不同。

1.4.2 软件工程的成就与发展展望

40 年的发展,软件工程取得了巨大成就。但一般以为,软件工程并非解决软件危机的灵丹妙药。国内外的实践一致说明,软件生产率的提高,总是赶不上软件需求的增长。本节将首先回顾软件工程迄今取得的成就,然后对其今后发展作一展望。

1. 令人兴奋的成果

1987 年,美国计算机科学家 Frederick P. Brooks 博士(即上文提到过的 IBM/360 操作系统的项目经理,被人尊称为“IBM/360 计算机系列之父”),曾在一篇题为“没有银弹”(No Silver Bullet)的著名论文中,把软件生产的困难区分为本质问题和非本质问题两大类,并且指出,软件工程只能解决软件开发中出现的非本质问题,对解决本质问题仍无能为力。他总结说,从提出软件工程以后的 20 年(1968—1987)间,全球的软件生产率一直以 6% 左右的速度稳步增长,即每隔 12 年软件生产率大约提高一倍,其成就确实令人兴奋。但直到这篇论文发表时,并未像神话传说那样找到制服“狼人”(比喻导致软件危机的本质性问题)的“银弹”。

Brooks 在这篇文章中还预期,随着面向对象设计、程序正确性证明等技术的发展,某些非本质的难题将继续得到解决,但为了解决软件生产中固有的、本质上的困难,他建议应改变软件的生产策略,例如尽可能利用现成的软件(参见第 10 章),采用快速原型和增量开发技术(参见第 2 章)等软件开发模型等。他还把取得重大突破的希望寄托在出现能够创建诸如 UNIX、Pascal、Smalltalk 那样产品的伟大的设计者身上,主张把鼓励和培训伟大的设计者

作為提高軟件生產率的一項最重要的目標。

2. 今後發展的展望

第二、三代軟件工程的出現，給克服軟件危機再次帶來了希望。隨著面向對象編程粒度的增大，軟件工程師在前進的征途上陸續譜寫出一些新的華章。特別是構件開發的規範化與市場化，已經把軟件開發推進到一個新階段，出現了“開發伴隨軟件復用，開發為了軟件復用”(development with reuse, development for reuse)以及“軟件就是服務”(software is service)等新思想。這些突出的成就，是否表明基於構件的軟件開發，可能最終促進 Brooks 要尋找的“新”的軟件生產策略的出現呢？毋庸諱言，構件開發目前還面臨着許多問題需要解決，但如果繼續沿着這一方向前進，會不會由此找到解決“軟件生產本質困難”的“銀彈”呢？前方似乎已初露曙光，讀者請拭目以待。

1.5 軟件工程的教學：本書導讀

經過 40 年的實踐，軟件工程的基本原則現已被產業界廣泛接受。全國各高等學校在計算機專業普遍開設了“軟件工程”課程，有些還在非計算機專業中也設置了與之相關的課程。本書主要定位於計算機專業的本科生。根據編者多年來的教學經驗和體會，在這一層次的“軟件工程”課程的教學中，特別要正確認識和處理好以下的幾個關係。

1. 3 代軟件工程的相互關係

從 20 世紀 80 年代中期以來，面向對象軟件工程發展迅速，其主要技術已基本成熟，成為軟件開發的主流范型。因此本版的編寫宗旨，將從第 2 版“并行介紹傳統的和面向對象的軟件工程”，轉變為“重點介紹面向對象軟件工程”，對第 2 版的內容進行較大幅度的修改。具體涉及如下內容：

- ① 第 1、2 章仍為綜述，其基本輪廓不變，但內容向面向對象軟件工程傾斜。
- ② 傳統軟件工程從原來的多章壓縮為一章，重點講述以結構化分析與結構化設計為代表的結構化程序設計技術。
- ③ 加強面向對象軟件工程的內容，把原來第 2 版的第 6、7 兩章擴充為“面向對象與 UML”（第 4 章）、“面向對象分析”（第 6 章）和“面向對象設計”（第 7 章）等 3 章，並在第 6、7 這兩章分別給出實例。
- ④ 適當反映軟件工程的近期進展。除第 2 版的“軟件復用”仍設專章討論外，另增加第 14 章“軟件工程高級課題”，在其中簡介“Web 工程”、“基於體系結構的軟件開發”、“形式化的軟件開發”等新發展。

值得指出，從面向過程到面向對象再到基於構件，3 代軟件工程並非相互排斥，而是“你中有我，我中有你”。前已提到，構件開發其實是面向對象開發的延伸與歸宿。在第一、二兩代軟件工程之間，也有許多原理與方法是通用的。例如“分析先於設計”(analysis before design)、“設計先於編碼”(design before coding)、“使程序(的結構)適合於問題(的結構)”(make the program fit the problem)等精辟的警句，就是在第一代軟件工程時期已經提出，並

被第二、三两代软件工程继续沿用的著名原则。在开发方法方面，也可举出很多三代一脉相承的例子。例如，分析与设计都应该提倡建立模型；编码需遵循编码范型；要加强软件工具的开发，不断改进以工具集成为特点的软件工程环境等，在各代软件工程中都十分重视。

2. 软件工程技术与管理的关系

技术与管理是软件生产中缺一不可的两个方面。没有科学的管理，再先进的开发技术也不能充分发挥作用。作为主要供大学本科学生使用的教材，把重点放在软件开发技术上无疑是适当的，但也有必要讲一点软件工程管理，特别是工程管理（第 11 章）与质量管理（第 12 章）。

管理离不开度量。“靠度量来管理”（management by measurement），已成为现代管理工作的一条重要原则。软件度量学（software metrics）和软件经济学（software economics）的诞生，就是这一原则在软件工程管理中的具体体现。在本书中，对项目度量和过程度量均将作简单介绍（第 12 章）。

3. 形式化方法与非形式化方法的关系

软件工程是广大软件人员长期实践活动的科学归纳与总结，包含了他们行之有效的好方法与好经验；也包含了很多学者为实现更高的软件可靠性、更高的软件生产自动化程度所进行的一系列理论课题的研究成果。这里所说的形式化方法，就是软件工程的高级研究课题之一。它是一种基于数学的开发技术，主要采用数学的方法来描述系统的性质，例如程序变换和程序验证等。而非形式化方法（亦称欠形式化方法）则主要运用文本、图、表与符号来描述系统的模型，如结构化设计、面向对象设计和 UML 语言等。

虽然形式化方法的支持者宣称这种技术将引发软件开发技术的革命，但它的实现难度很大，进展十分缓慢。软件工程师一般而言都不具备它所要求的数学基础，这也限制了它的推广。有人认为，形式化的方法加上自动化的开发环境，可能是解决这一难题的出路。

作为大学本科学生的教材，本书主要讲述非形式化方法的软件开发技术。但为了扩展读者对软件工程的了解，在有关章节仍对形式化方法作简单的介绍，例如净室模型（第 2.4.2 节）、程序正确性证明（第 12.4 节）等。

4. 小程序设计与大程序设计的关系

前面已提到，有些初学者认为软件工程仅适用于大型软件的开发，对开发小规模软件并无多少用处。在高校师生中澄清这一误解，对做好软件工程的教学习十分重要。诚然，软件工程是在软件规模急剧增长所引起的软件危机中诞生的，但它的基本思想和许多方法，在不同规模的软件开发中都有自己的用武之地。D. Gries 说过，“只有学会有效地编写小程序的人，才能学习有效地编写大程序。”小程序设计是大程序设计的基础，两者都需要软件工程的指导。在数年前的一次软件工程教育国际会议上，有一批教师积极倡导用软件工程来统率计算机科学与工程教学，主张在高校中普遍建立软件工程系。编者在一次与青年学生的座谈中，有一位学生在谈及学习软件工程的收获时甚至说，软件工程所倡导的“Why（软件计划）—What（软件分析）—How（软件实现）”开发过程，加上在分析与设计活动中反复强调的建模（modeling）思想，不仅是有效的工程方法，而且对指导人们的日常生活和其他工作也有着重

要的意义。

还需指出,“软件工程”课程具有很强的实践性。除了在课堂教学中要强调实用,在教材中多举一些实例外,应该尽可能在课程中安排适量的课程设计,让学生实际演练,真正做到学以致用、学用结合的目的。

小 结

软件工程自 1968 年提出以来,在过去 40 年中,已发展成为用于指导软件生产工程化,覆盖软件开发方法学、软件工具与环境、软件工程管理等内容的一门新学科。随着程序设计从结构化程序设计发展到面向对象程序设计,软件工程也由传统的软件工程演变为面向对象的软件工程,现正向更新一代的基于构件的软件工程迈进。

通过长期的实践,软件工程研究人员积累了许多行之有效的原理与方法,已经为产业界广泛接受与应用。大多数国内外高校都把软件工程列为本科生的教学内容。许多高校开办了软件学院,将软件工程设置为主要课程。为了做好本科这一层次的软件工程教学,本章未提出了 4 个需要正确认识的关系:3 代软件工程的相互关系;软件工程技术与管理的关系;形式化方法与非形式化方法的关系;小程序设计与大程序设计的关系,可供读者特别是初学者参考。

习 题

1. 什么是软件危机?为什么会产生软件危机?
2. 什么是软件生产工程化?工程化生产方法与早期的程序设计方法主要差别在哪里?
3. 分别说明软件开发方法与开发工具、软件技术与软件管理的关系。
4. 试根据你的亲身实践,谈谈软件工具在软件开发中的作用。
5. 什么是软件工程环境?谈谈你对环境重要性的认识。
6. 何谓面向对象软件工程?简述它与传统软件工程的差别和联系。
7. 软件按规模可分成哪几类?简述软件工程在各种规模的软件开发中的作用。
8. 什么是形式化软件开发方法?实现这类开发的困难和出路在哪里?

上篇

传统软件工程

第2章 软件生存周期与软件过程

第3章 结构化分析与设计

第2章 软件生存周期与软件过程

本章介绍软件工程中的两个重要概念：软件生存周期和软件过程。在早期的软件工程中，这两个概念常不加区分地使用，以后演变为既相互联系又有重要差异的不同概念。本章将对它们及其他的相关问题作简要说明。

2.1 软件生存周期

一切工业产品都有自己的生存周期，软件（产品）也不例外。一个软件从开始立项起，到废弃不用止，统称为软件的生存周期（life cycle）。大约在 20 世纪 70 年代，软件工程概念问世不久，人们就提出了软件生存周期的概念。从软件立项到废弃，软件生存周期一般被划分为计划、开发与运行 3 个时期，图 2.1 列出了一个典型的软件生存周期中的主要活动。由于整个生存周期被划分为较小的阶段，每个阶段只赋予有限的活动，使得因为软件规模增长而大大增加的软件复杂性变得较易控制和管理。本节将结合图 2.1 简单说明生存周期各阶段的主要活动。

2.1.1 软件生存周期的主要活动

1. 需求分析

主要弄清用户需要用计算机来解决什么问题，由系统分析员根据自己对问题的理解，提出关于系统目标与范围的说明，从用户的视角对需求进行定义和分析，用需求模型的形式准确地表达出来，其中包括软件的功能需求、性能需求、环境约束和外部接口描述等，并需通过用户审查和认可。系统分析员与用户之间的良好配合，是做好本阶段工作的关键。

2. 软件分析

软件分析的任务是在系统需求模型的基础上，从开发人员的视角对软件的需求模型进行

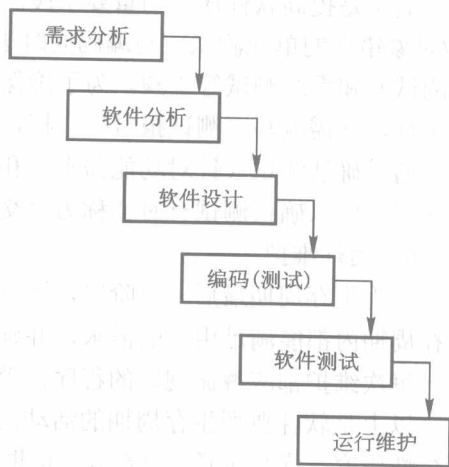


图 2.1 典型的软件生存周期

分析, 建立与需求模型一致的、与实现无关的软件分析模型, 它既是对软件系统逻辑模型的描述, 也是下一步进行软件设计的依据。

3. 软件设计

软件设计的任务, 是将软件分析模型转变为考虑具体实现技术和平台的软件设计模型。它一般又可细分为总体设计(亦称概要设计)和详细设计, 其中总体设计包括确定软件的总体结构和其他全局性的设计原则等, 详细设计则是确定软件的每一个部件的数据结构和操作。软件设计阶段的工作是最需要创造力的, 其设计结果对软件开发的成败起着关键性作用。为了完成本项活动, 应选择资深开发人员来担任软件架构师和设计师进行设计工作, 并需编写相应的设计文档。

4. 编码

编码就是按照选定的程序设计语言和可复用软件构件包, 把设计文档翻译为源程序。与前两个阶段相比, 编码的难度相对比较小, 一般由编码员或初级程序员担任。

需要指出, 前面各个阶段产生的都属于软件文档, 而这个阶段将产生可在计算机上执行的程序。早在 20 世纪 60 年代, 人们就发现编一点测试一点, 比编完了再测试更省力。所以单元测试通常与编码同时进行, 故这个阶段亦称为“编码(测试)”。

5. 软件测试

测试是提高软件质量的重要手段。按照不同的目的, 测试可细分为多个层次。除针对模块/对象错误的单元测试应与编码同时进行外, 还要执行集成测试、确认测试(包括 α 测试与 β 测试)和系统测试等步骤。为了确保这项活动不受干扰, 软件测试通常由独立的测试工程师执行, 并需编写“测试报告”文档, 包括测试计划、测试用例、测试结果等内容。

通过确认测试(针对功能需求)和系统测试(针对性能等非功能需求)后, 软件即可交付使用, 所以确认测试有时亦称为“交付测试”。

6. 运行维护

作为生存周期最后一个阶段, 运行维护阶段的任务主要是做好软件维护, 使软件在整个生存周期内都能满足用户的需求, 并延长其使用寿命。对于大型软件来说, 维护是必不可少。每次维护都应遵循规定的程序, 填写或更改相关的文档。

以上是软件典型生存周期的活动内容。比较大型的软件在开始开发前, 还需要进行一次可行性研究。详见本章第 2.6 节, 这里不再赘述。

2.1.2 生存周期与软件过程的关系

对软件生存周期的研究, 导致了软件工程的另一个重要概念——软件过程的诞生。本节将说明这两个概念的关系, 并简述软件过程的演变。

1. 从软件生存周期到过程模型

1976 年, 《IEEE 计算机学报》就刊登 B. W. Boehm 的文章, 建议将软件生存周期划分为

系统需求、软件需求、概要设计、详细设计、编码纠错、测试和预运行、运行维护等 7 个阶段。著名的瀑布模型 (waterfall model) 就是在 Boehm 建议的基础上进一步完善而形成的。这是生存周期和软件过程的研究相互结合的一个早期实例。

什么是过程? 广义地说, 人们随时间的流逝而进行的各种活动均可称为过程 (process, 或译为流程)。特定地说, 软件过程可理解为围绕软件开发所进行的一系列活动。在早些时候, 人们常常把软件过程称为“软件开发模型” (software development model)。例如, 第一代软件工程普遍采用的瀑布开发模型就简称为瀑布模型而不是瀑布过程。

按照早期软件工程的观点, 软件开发模型包含的阶段与活动同软件生存周期划分的阶段与活动基本上是一致的。因此, 在早期的软件开发中, 软件开发“过程”与软件生存周期“过程”常常不加区分, 例如图 2.1 显示的典型软件生存周期, 也可以用来展示典型瀑布模型的阶段与活动。二者都属于线性模型, 而且具有等同性。它们的共同特点是将整个“过程”严格地划分阶段, 各阶段的活动分步完成; 前一阶段的活动没有结束, 下一阶段的活动就不能进行, 恰如奔流不息、拾级而下的瀑布。

2. 软件过程的演变

在第一代软件工程期间, 瀑布开发模型在软件的开发与维护中被普遍采用。但到了 20 世纪 80 年代中期, 人们终于发现, 这种线性开发模型不适合于大型复杂系统的开发。原因很简单, 一个大型复杂系统的开发周期一般需要一年半到两年, 而在软、硬件技术快速发展的背景下, 要求一年半以前约定的需求到开发时仍保持不变, 几乎是不可能的。软件需求频繁地更改恰是开发大型复杂系统的特点, 而一律采用线性模型显然是不合适的。

正是上述的发现, 促使软件开发模型开始演变。除传统的线性开发模型外, 又陆续涌现了一批新的、允许在开发过程中任意回溯和迭代的过程模型。下面将依次介绍。

2.2 传统的软件过程

2.2.1 瀑布模型

如前所述, 瀑布开发模型是一种基于软件生存周期的线性开发模型, 是 W. Royce 在 1970 年首先提出的。本节将进一步考察这一模型的特点。不言而喻, 它与软件生存周期的特点是一致的。现简述如下。

1. 阶段间的顺序性和依赖性

在这里, 包含两重含义:

第一, 顺序性: 只有等前一阶段的工作完成之后, 后一阶段的工作才能开始。

第二, 依赖性: 前一阶段的输出文档是后一阶段的输入文档。这表明, 只有当前一阶段有正确的输出时, 后一阶段才能获得正确的结果。如果在某个阶段出现了问题, 往往要追溯

到在它之前的一些阶段，必要时可能要修改此前已完成的文档。

2. 推迟实现的观点

瀑布开发模型的实践表明，对于大、中型的软件，编码开始得愈早，完成开发需要的时间反而愈长。这是因为，过早地编码容易导致返工，甚至造成灾难性的严重后果。

所谓“推迟实现”，就是把待开发软件的逻辑设计与物理实现清楚地区别开来，即在需求分析和软件设计阶段只考虑系统的逻辑模型，等到编码阶段再来完成程序清单。这也是瀑布模型开发的一条重要的指导思想。

3. 保证质量的观点

为了保证软件的质量，瀑布模型对软件文档采取了以下两条严格的措施。

第一，每一阶段必须完成规定的文档。没有完成文档，就不能视为完成该阶段的任务。软件开发是许多人共同参与的工作，完整与合格的文档，不仅是开发阶段软件人员相互通信的媒介，也是运行阶段对软件进行维护的重要依据。

第二，每一阶段都要对完成的文档进行复审，以便尽早发现问题，消除隐患。从图 2.2 可知，愈是早期潜伏下来的故障，暴露出来的时间就愈晚，排除故障需付出的代价也愈高。及时的复审是保证软件质量、降低开发成本的重要措施。

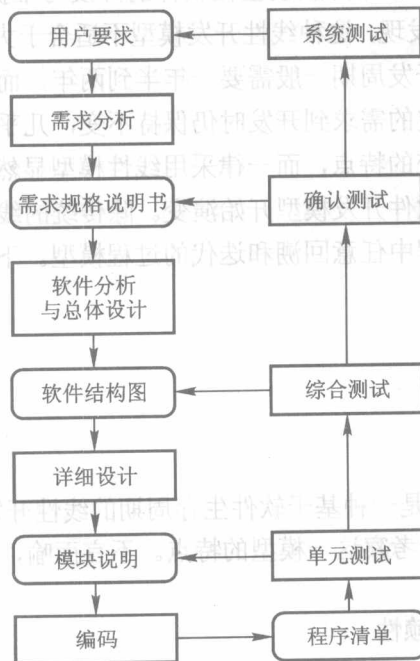


图 2.2 瀑布模型的阶段与文档

4. 存在的问题

按照瀑布模型来开发软件，只有当分析员能够做出准确的需求分析时，才能够得到预期

的结果。不幸的是，由于多数用户不熟悉计算机，系统分析员对用户的专业也往往了解不深，因而很难在开发的初始阶段彻底弄清软件需求。F. Brooks 甚至断言，“在软件产品的某个版本试用之前，要用户（即使有软件工程师的配合）完全、正确地对一个现代软件产品提出确切的需求，实际上是不可能的”。为了解决这一问题，人们提出了“快速原型模型”。

2.2.2 快速原型模型

1. 原型开发的优越性

当机械工程师接到一个设计任务后，通常会根据要求和自己的理解，在较短的时间内按一定的比例设计并制造一个样机，交给用户确认后再成批投产，这台样机可以称为原型。

快速原型模型（rapid prototype model）的中心思想是：先建立一个能够反映用户主要需求的原型，让用户实际看一下未来系统的概貌，以便判断哪些功能是符合需要的，哪些方面还需要改进。然后将原型反复改进，直至建立完全符合用户要求的新系统。

采用瀑布模型时，软件的需求分析也可以在用户和系统分析员之间反复讨论，使之逐步趋于完善。但这种讨论终究是“纸上谈兵”；而原型系统则是“真枪实弹”，能够使用户立刻与想象中的目标系统作出比较。软件开发人员向用户提供一个“样品”，用户向开发人员迅速作出反馈。逼真，快速，这就是原型软件开发的优越性。

2. 原型开发的方法

必须指出的是，在生产硬件等有形的工业产品时，先制造一个样机，成功后再成批投产，如果硬件的生产批量大，因制造样机增加的成本仅占很小的比例。但软件属于单件生产，如果每开发一个软件都先提供原型，成本就会成倍地增加。为此，在建立原型系统时常采取下述的做法：

首先，原型系统只包括未来系统的主要功能及系统的重要接口。它不包括系统的细节，例如异常处理、对非有效输入的反应等，对系统的性能需求如硬件运行速度等也可推迟考虑。

其次，为了尽快向用户提供原型，开发原型系统时应尽量使用能缩短开发周期的语言和工具。例如，UNIX 支持的 Shell 语言是一种功能很强的甚高级语言，有人用这种语言来编制一个办公室自动化系统的原型系统，仅用一天就完成了编码与测试，比使用其他高级语言快了许多倍。虽然 Shell 在运行时需要很大的支撑系统，运行速度也比较慢，不宜用来实现最终的实际系统，但用它来开发原型系统，却可以大大加快实现的速度。

最后，怎样产生最终的实际系统呢？一种自然的想法，就是把原型系统作为基础，通过补充和修改获得最终的实际系统。但在实际工作中，由于原型系统使用的语言往往存在效率不高等原因，除了少数很简单的系统外，大多数原型都废弃不用，仅把建立原型的过程当作帮助定义软件需求的一种手段。

3. 原型模型的启示

图 2.3 显示了快速原型软件开发的过模型（为了画面简洁，有一些阶段在图中省略未画）。用户的介入和反馈，使它在原型的分析与设计阶段可能出现多次回溯和迭代，从而形成非线性的开发模型。

原型开发模型改变了“把生存周期等同于过程模型”的习惯性思维，使人们认识到：生存周期只指出了整个周期中应包含哪些活动，并未规定这些活动应该发生多少次。所以软件开发人员的任务，就是通过软件过程的重新安排，使之更适合于待开发的软件系统。

4. 应该防止的偏向

采用原型法应防止两种偏向：一种可能来自用户，他们舍不得将“活生生”的原型废弃不用，要求开发者仅作“少量修改”就交付使用；另一种常常来自开发者，当他们熟悉原型后，明知它有许多不足，却不愿全部推倒重来，宁可在最终系统中保留一部分不理想的程序。这些偏向如不纠正，都可能影响软件开发质量。

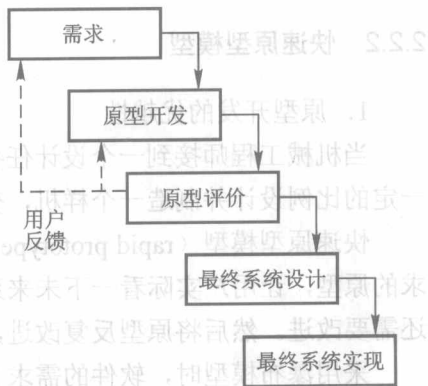


图 2.3 快速原型法的过程模型

2.3 软件演化模型

原型开发模型的出现，使人们逐渐熟悉非线性的开发模型。随着软件规模的不断增长，复杂软件开始采用渐增式或迭代式的开发方法。于是，一种称为演化模型（evolutionary model）的渐进式的开发模型应运而生。它遵循迭代的思想方法，使所开发的软件在迭代中逐步达到完善，有时也把它称为迭代化开发模型。常见的演化模型有增量模型与螺旋模型两种，一般适用于大型软件的开发。

2.3.1 增量模型

增量模型（incremental model）是瀑布模型的顺序特征与快速原型法的迭代特征相结合的产物。这种模型把软件看作一系列相互联系的增量（increments），在开发过程的各次迭代中，每次完成其中的一个增量，如图 2.4 所示。例如，如果用增量模型开发一个大型的文字处理软件，第一个发布的增量可能实现基本的文件管理、文档编辑与生成功能；第二个增量具有更加完善的文档编辑与生成能力；第三个增量完成拼写检查与文法检查；第四个增量实现页面布局等高级功能。其中任一个增量的开发流程均可按瀑布模型或快速原型法完成。

在一般情况下，第一个增量通常是软件的核心部分。首先完成这一部分，可以增强用户

和开发者双方的信心。增量模型也有利于控制技术风险。例如，难度较大或需要使用新硬件的部分可放在较后的增量中开发，避免用户长时间等待；不同的增量也可配备不同数量的开发人员，使计划增加灵活性；等等。

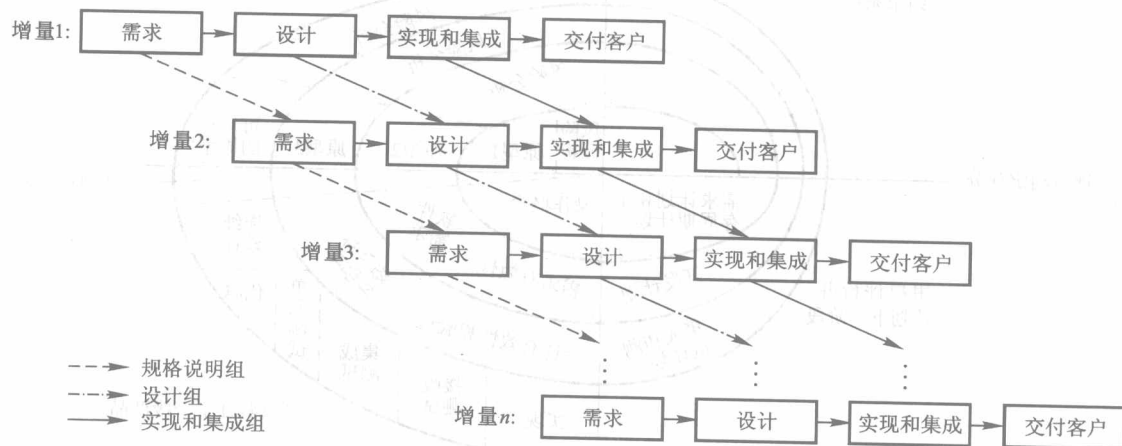


图 2.4 增量模型

2.3.2 螺旋模型

螺旋模型（spiral model）为目前软件开发中最常用的一种软件开发模型，是在结合瀑布模型与快速原型模型基础上演变而成的，尤其适用于大型软件的开发。

1. 典型的迭代模型

从总体上看，螺旋模型是一种典型的迭代模型。每迭代一次，螺旋线就前进一周。

如图 2.5 所示，当项目按照顺时针方向沿螺旋线移动时，每轮螺旋均包含计划、风险分析、建立原型、用户评审 4 种活动，按以下顺序周而复始，直到实现最终产品。

① 计划。用于选定完成本轮螺旋所定目标的策略，包括确定待开发系统的目标、选择方案、设定约束条件等。

② 风险分析。评估本轮螺旋可能存在的风险。必要时，可通过建立一个原型来确定风险的大小，然后据此决定是按原定目标执行，还是修改目标或终止项目。

③ 建立原型。在排除风险后，建立一个原型来实现本轮螺旋的目标。例如，第一圈可能产生产品的需求规格说明书，第二圈可能实现产品设计，等等。

④ 用户评审。由用户评价前一步的结果，同时计划下一轮的工作。

需要指出，如果开发小组对项目的需求已有较好的理解，则第一圈就可以直接采用瀑布模型，这时的螺旋模型可只含单圈螺旋。反之，如果对项目的需求没有把握，就需要经过多圈螺旋，并通过开发一个或多个原型来完善软件的需求。

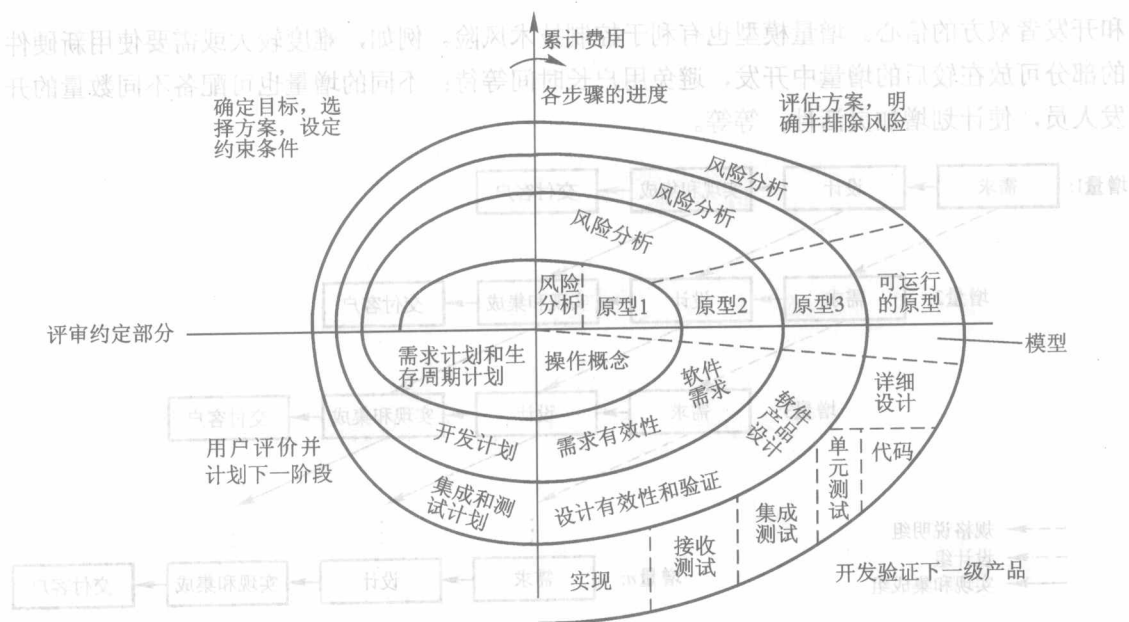


图 2.5 螺旋模型

2. 风险分析

同其他工程一样，软件开发也存在着风险。在制定项目计划时，对项目的预算、进度与人力，对需求、设计中采用的技术及存在的问题，都需要仔细分析与估算。项目越复杂，估算中的不确定因素越多，风险也越大，严重时可能会导致软件开发的失败。风险分析的目的，就是要了解、分析并设法降低和排除这种风险。

对于高风险的大型软件，螺旋模型是一个理想的开发过程。

当软件随着过程的进展而演化时，开发者和用户都需要更好地了解每一级演化存在的风险。螺旋模型利用快速原型作为降低风险的机制，在任何一次迭代中均可应用原型方法；同时，在总体开发框架上，又保留了瀑布模型所固有的顺序性和“边开发，边评审”等特点。这种将二者融合在一起的迭代框架，无疑可更真实地反映客观世界。

3. 螺旋模型的特点

螺旋模型的特点，是在项目的所有阶段都考虑各类风险，从而能在风险变成问题之前降低它的危害。但是它也有不足的地方。例如，它难以使用户相信演化方法是可控的；过多的迭代周期，也会增加开发成本和时间，等等。可以说，螺旋模型开发的成败，在很大程度上依赖于风险评估的准确性。由于风险评估是一门专门技术，其结果又受到主观因素的影响，如果一个大的风险未被发现和控制，其后果显然是严重的。

2.3.3 构件集成模型

抛开在开发中具体使用的方法与工具，仅就开发活动的框架而言，上述各种软件开发模型对面向过程软件与面向对象软件的开发是同时适用的。换句话说，面向对象开发模型均可借用面向过程开发模型的活动框架，不需要另起炉灶。但本节将要介绍的构件集成模型，则主要适用于面向对象的软件开发。在介绍这一开发模型之前，这里先讲一下面向对象方法的若干基本概念。

1. 面向对象的基本概念

面向对象的思想最初起源于 20 世纪 60 年代中期的仿真程序设计语言 Simula 67。20 世纪 80 年代初出现的 Smalltalk 语言和 90 年代推出的 C++、Java 语言及其程序设计环境，先后成为面向对象技术发展的重要里程碑。从 20 世纪 80 年代末开始，面向对象的软件设计和面向对象的需求分析都得到快速发展，特别是 20 世纪 90 年代中期由 Booch、Rumbaugh 和 Jacobson 共同提出了统一建模语言（unified modeling language, UML，详见第 4 章），把众多面向对象分析和设计工具综合成一种标准，使面向对象的方法成为主流的软件开发方法。

前面已指出，面向对象思想的最重要特征，是在解空间中引入了“对象”的概念，使之逼真地模拟问题空间中的客观实体。面向对象方法学包含了对象（object）、类（class）、继承（inheritance）、消息（message）等核心概念。Coad 和 Yourdon 认为，采用这 4 种概念进行开发的软件系统可以认为是面向对象的。为此，他们把面向对象方法归结为一个简单的公式，即面向对象=对象+分类（classification）+继承+消息通信（communication with messages）。

2. 什么是构件

对象技术将事物封装成包含数据和加工该数据的方法的对象，并抽象成为类。经过适当设计和实现的类，也可称为构件（component）。它们在某个领域中具有一定的通用性，可以在不同的计算机软件系统中复用。将这些构件存储起来构成一个构件库，就为基于构件的软件开发模型提供了技术基础，于是构件集成模型（component integration model）应运而生。

3. 构件集成模型的特征

构件集成模型利用预先封装好的构件来构造应用软件系统。它融合了螺旋模型的不少特征，也支持软件开发的迭代方法。开发活动从描述候选类开始，通过检查软件系统处理的数据以及操作这些数据的方法，把相关的数据和方法封装成一个类，然后到构件库中查找这个类。如果存在，就从库中提取出来以供复用；如果候选类不存在，则采用面向对象的方法来实现它，并把它添加到构件库中。这样，通过集成从构件库中提取的已有类和为了满足应用程序的特定需要而构建的新类，即可得到待开发软件的第一个迭代。然后进入下一轮螺旋周期，继续进行构件集成的迭代。图 2.6 显示了构件集成模型的流程。

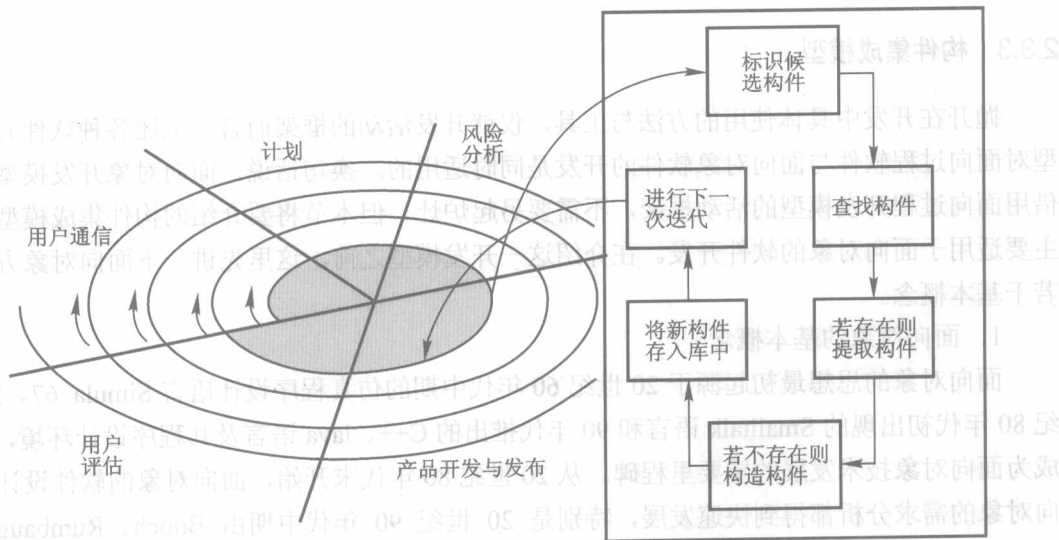


图 2.6 构件集成模型

2.4 形式化方法模型

众所周知，软件开发方法可区分为形式化方法与非形式化方法两大分支。前者以形式化的程序变换技术为主要研究内容，多流行于学术界；后者旨在用工程方法生产出高质量、易维护的软件产品，多流行于工业界。作为大学“软件工程”课程的入门教材，本书将重点放在非形式化的软件开发模型上。本节仅选讲两种形式化开发模型——转换模型与净室模型。

2.4.1 转换模型

转换模型 (transformational model) 是将形式化软件开发和程序自动生成技术相结合的一种软件开发模型。它采用严格的数学方法来表示软件需求规格说明书，然后进行一系列自动或半自动的程序变换，最终将需求规格说明书转换为计算机系统能够接受的目标程序系统。

1. 转换模型的软件开发过程

如图 2.7 所示，转换模型的开发过程大体上可分为 3 步。

(1) 确定形式化的需求规格说明书

当软件需求分析完成后，首先应选用一种形式化的语言，生成一个形式化需求规格说明书。为了确认它与软件需求的一致性，可以在形式化需求规格说明书的基础上开发一个软件原型，让用户对该原型系统从界面、功能和性能等多个方面进行比较，必要时对形式化需求规格说明书和原型进行修正，直到形式化需求规格说明书能正确反映用户需求为止。

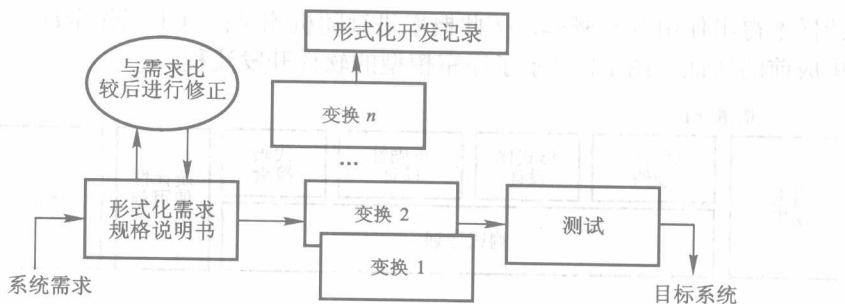


图 2.7 转换模型

(2) 进行自动的程序变换

从用户需求到目标系统实现，可能会有很多条路径。为此，通常要采用不同的变换系统和变换顺序，然后将所得到的信息保存到形式化开发记录中。这些程序变换系统可能由一系列不同的程序变换语言及其编译系统、分析验证工具、控制变换过程的工具和变换规则库等组成。

(3) 对形式化开发记录进行测试

通过一系列的人机交互或其他测试手段，最终生成计算机系统可以接受的目标代码。

2. 转换模型的常用技术

实施转换模型的常用技术，目前主要有基于模型的需求规格说明书及其变换技术、基于代数结构的需求规格说明书及其变换技术、基于时序逻辑的需求规格说明书和验证技术以及基于可视形式化的技术等。

从理论上说，一个正确的、满足客户需要的形式化需求规格说明书，经过一系列正确的程序变换后，可以确保得到一个正确的软件系统。但要得到这样一个形式化需求规格说明书，目前还有比较大的难度。因此，现在使用转换模型开发软件还很费时和昂贵。但这种模型的一个变种——即所谓“净室软件工程”（cleanroom software engineering）的方法，目前已被一些软件公司采用。今后，随着形式化需求规格说明书的逐渐流行，转换模型也会被越来越多的软件开发者所接受。

2.4.2 净室模型

净室模型（cleanroom model）是一种形式化的增量开发模型。其基本思想是力求在分析和设计阶段就消除错误，确保正确，然后在无缺陷或“洁净”的状态下实现软件的制作。

和增量模型一样，净室开发把软件看成一系列的增量，每个增量是一个用形式化方式表示的“盒”（box）。当需求收集结束后，就用盒结构来表示分析和设计模型，这种盒是在某个特定的抽象层次上对系统（或系统的某些方面）的一次封装。完成了盒结构设计之后，先对设计的软件构件进行正确性验证；通过正确性验证后，再将形式化的盒结构设计转换为适当的程序设计语言表示的源代码，并进行源代码的正确性验证；接着开始统计性使用测试，即

根据统计样本得出使用分布概率，按此概率进行随机测试；以上工作全部完成后，还要对增量进行集成前的认证。图 2.8 显示了净室模型的开发流程。

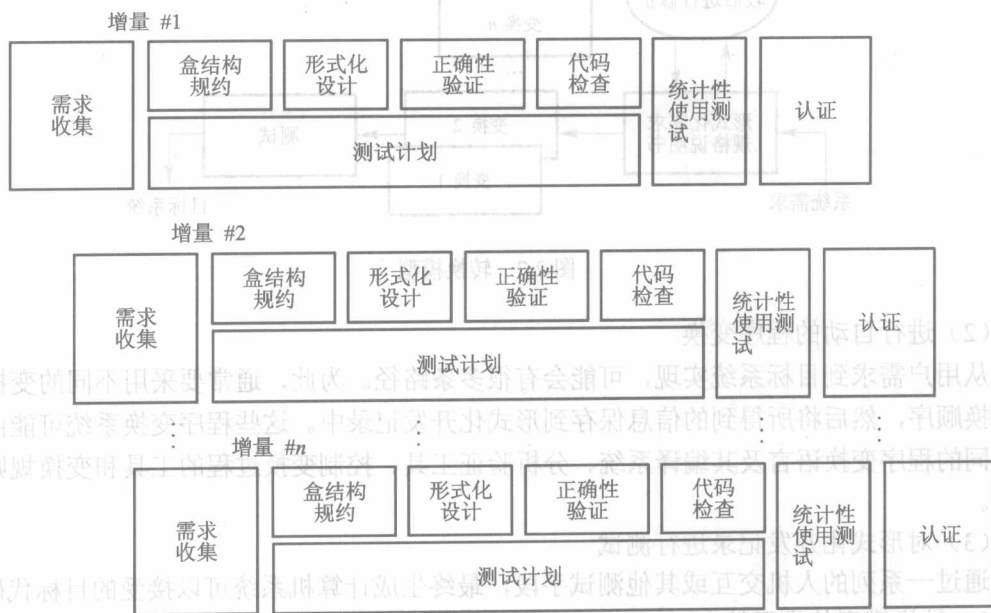


图 2.8 净室模型

以上共介绍了 7 种软件开发模型，表 2.1 显示了它们的主要特点。需要指出的是，每种模型各有优缺点，一般适用于解决软件开发中的某类问题。作为软件开发组织，应该优先选择适合本组织人员及其管理情况的软件开发模型，并随着待开发产品的特性不同而变化。在实际开发中，有时也可把几种模型组合起来使用，以便取长补短。

表 2.1 7 种软件开发模型的主要特点

开发模型	特点	适用场合
瀑布模型	线性模型，每一阶段必须完成规定的文档	需求明确的中、小型软件开发
快速原型模型	用户介入早，通过迭代完善用户需求，应用快速开发工具	需求模糊的小型软件开发
增量模型	每次迭代完成一个增量，可用于 OO 开发	容易分块的大型软件开发
螺旋模型	典型迭代模型，重视风险分析，可用于 OO 开发	具有不确定性的大型软件开发
构件集成模型	软件开发与构件开发平行进行，主要用于 OO 开发	领域工程、行业的中型软件开发
转换模型	形式化的需求规格说明书，自动的程序变换系统	理想化模型，尚无成熟工具支持
净室模型	形式化的增量开发模型，在洁净状态下实现软件制作	开发团队熟悉形式化方法，中小型软件开发

2.5 统一过程和敏捷过程

除了上面介绍的 7 种软件开发模型外，自 20 世纪末开始，又出现了两种与之前迥异的软件过程模型——统一过程和敏捷过程。本节仅对这两种过程作简单介绍。

2.5.1 统一过程

统一过程是由原 Rational 公司开发的，所以也称 RUP (Rational Unified Process)，现为 IBM 公司的一个产品。简单地说，统一过程描述了软件开发中各个环节应该做什么、怎么做、什么时候做以及为什么要做，描述了一组以某种顺序完成的活动。其结果是一组有关系的文档，例如模型和其他一些描述，以及对最初问题的解决方案等。过程描述的一个重要部分，是定义如何使用人力、机器、工具和信息等资源的一些规则来完成某个确定的目标，为用户的问题提供解决方案。

统一过程在一个二维空间中描述软件开发活动，水平轴代表时间，显示了过程动态的一面，它将一个软件生存周期分为 4 个阶段 (phase)，每个阶段又可以分为一个或多个迭代 (iteration)。迭代是一个完整的开发循环，它的结果是产品的一个可执行版本，是正在开发的最终产品的一个子集，从一个迭代到下一个迭代，不断递增成长，直到最后成为最终系统。每个阶段或迭代一般都设定一个里程碑 (milestone)，里程碑是一个时间点，在这个时间点上必须做出重要的决策，达到一些关键的目标。

垂直轴代表过程静态的一面，其中用活动 (activity) 表示怎么做，用产品 (artifact) 表示做什么，用人员 (role) 表示谁来做，用 workflow (workflow) 表示什么时候来描述。

统一过程包括 4 个阶段 (如图 2.9 所示)，分别是初始阶段、细化阶段、构造阶段和迁移阶段，每个阶段又分为若干次迭代，每次迭代有一个核心 workflow，都会经历需求、分析、设计、实现、测试等活动。下面介绍每个阶段的主要任务，从中可了解统一过程模型的思想。

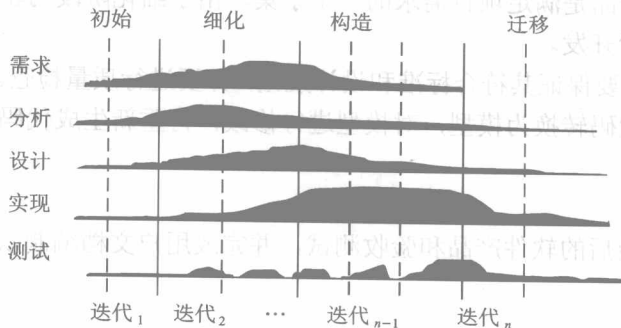


图 2.9 统一过程图示

1. 初始阶段

本阶段确定所设立的项目是否可行，具体要做如下工作：

- ① 对需求有一个大概的了解，确定系统中的大多数角色和用例（关于角色和用例的概念，可参见第4.3.1节），但此时的用例是简要的。
- ② 划分主要子系统，给出系统的体系结构概貌。
- ③ 分析项目执行的风险。
- ④ 考虑时间、经费、技术、项目规模和效益等因素。
- ⑤ 制定开发计划。

2. 细化阶段

本阶段识别出剩余的大多数用例。对当前迭代的每个用例进行细化，分析用例的处理流程、状态细节以及可能发生的状态变化。细化流程时，可以使用程序框图和协作图，还可以使用活动图、类图分析用例对风险的处理。这一阶段主要应完成以下工作：

- ① 进行需求风险分析。考虑项目的目标是否偏离了用户的需求。为了解决需求风险要充分了解用户需求以及各需求的优先级，还应尽量列出所有的用例，并要建立领域的概念模型。
- ② 进行技术风险分析。通过建立原型等方法，考察所选的技术方案是否可行。
- ③ 进行技能风险分析。考虑实施项目的人员素质能否胜任项目的要求。
- ④ 进行政策风险分析。考虑政策性因素对项目的影响。
- ⑤ 进行高层分析和设计，并作出结构性决策。
- ⑥ 产生简要体系结构，包括用例列表、领域概念模型和技术平台等。在以后的阶段中对细化阶段所建立的体系结构不能进行过大的变动。
- ⑦ 为构造阶段制定计划。

3. 构造阶段

本阶段识别出剩余的用例。每一次迭代开发都是针对用例进行分析、设计、编码、测试和集成的过程，所得到的产品是满足项目需求的一个子集。由于细化阶段的软件设计已经完成，这样各项目组可以并行开发。

在代码完成后，要保证其符合标准和设计规则，并要进行质量检查。对于新出现的变化，要通过逆向工具把代码转换为模型，对模型进行修改，再重新生成代码，以保证软件与模型同步。

4. 迁移阶段

这一阶段完成最后的软件产品和验收测试，并完成用户文档编制以及用户培训等工作。

2.5.2 敏捷过程

敏捷开发（agile development）是一种以人为核心、以迭代方式循序渐进开发的方法，其软件开发的过程称为“敏捷过程”。在这一过程中，软件项目的构建被切分成多个子项目，

各个子项目的成果都经过测试，具备集成和可运行的特征。简言之，就是把一个大项目分为多个相互联系但也可独立运行的小项目，并分别完成，在此过程中软件一直处于可用状态。

在 2001 年年初，由一些业界专家成立了敏捷联盟(agile software development alliance)，起草了敏捷软件开发宣言。该宣言针对一些企业的现状，提出了让软件开发团队具有快速工作、快速应变能力的若干价值观和原则，其中包括 4 个简单的价值观以及敏捷开发方法应遵循的 12 条原则。本书第 14 章介绍的 Web 工程就是一个采用敏捷过程的实例。

1. 敏捷开发的价值观

- ① 个人和交互胜过过程和工具。
- ② 可以运行的软件胜过面面俱到的文档。
- ③ 客户合作胜过合同谈判。

④ 响应变化胜过遵循计划。

2. 敏捷开发应遵循的 12 条原则

- ① 通过尽早地、不断地提交有价值的软件来使客户满意。
- ② 即使到了开发的后期，也欢迎改变需求。敏捷过程利用变化来为客户创造竞争优势。
- ③ 以从几个星期到几个月为周期，尽快、不断地提交可运行的软件。
- ④ 在整个项目开发期间，业务人员和开发人员必须天天都在一起工作。
- ⑤ 以积极向上的员工为中心，建立项目组，给他们提供所需的环境和支持，并对他们的工作予以充分的信任。
- ⑥ 在团队内部，最有效、效率最高的传递信息的方法，就是面对面的交流。
- ⑦ 测量项目进展的首要依据是可运行软件。
- ⑧ 敏捷过程提倡可持续的开发，责任人、开发者和用户应该为能够保持一个长期的、恒定的开发速度而努力。
- ⑨ 时刻关注技术上的精益求精和好的设计，以增强敏捷能力。
- ⑩ 简单是最根本的。
- ⑪ 最好的构架、需求和设计出于自组织的团队。
- ⑫ 每隔一定时间，团队要反省如何才能更有效地工作，然后相应地调整自己的行为。

还需指出，敏捷开发是一个持续地应用原则、模式以及实践来改进软件的结构和可读性的过程，而不是一个事件。它致力于保证系统设计在任何时间都尽可能简单、整洁及富有表现力。下面介绍的极限编程就是敏捷过程的一种方法。

2.5.3 极限编程

极限编程(extreme programming, XP)是由 Kent Beck 在 1996 年提出的轻量级的、敏捷的软件开发方法；同时它也是一个非常严谨和周密的方法。它有 4 个价值观：交流、简单、反馈和勇气；即，任何一个软件项目都可以从 4 个方面入手进行改善：加强交流；从简单做起；寻求反馈；勇于实事求是。

XP 建议采用循环迭代的开发方法，它将复杂的开发过程分解为一个个相对比较简单的小周期；通过积极的交流、反馈以及其他一系列的方法，使开发人员和客户都可以非常清楚开发进度、变化、待解决的问题和潜在的困难等，并根据实际情况及时地调整开发过程。XP 方法包括以下 12 个核心实践：

1. 完整团队 (whole team) XP 项目的所有参与者 (包括开发人员、客户、测试人员等) 应该在同一工作场所工作。

2. 计划对策 (planning game)

XP 方法中需要制定两个计划：发布计划和迭代计划。计划是根据业务的优先级和技术评估结果来制定的，早期制定的计划常常是不准确的，因此要在迭代过程中不断地修正。

3. 测试 (testing)

XP 方法提倡测试优先，先写测试后写代码，测试优先是为了在编码前使开发人员对代码进行周密思考，使开发人员尽快地检验他们的想法是否可行。

4. 简单设计 (simple design)

简单设计是指设计刚好满足当前定义的功能即可，能运行所有的测试，没有重复的逻辑，描述了程序员的重要意图，使用尽可能少的代码。

5. 结对编程 (pair programming)

XP 方法强烈推荐结对编程，即由两个程序员坐在同一台计算机前一起编程，在一个程序员开发的同时，另一个程序员负责检查程序的正确性和可读性。

6. 小软件版本 (small release)

经常、不断地发布可运行的、具有商业价值的小软件版本，供现场用户评估或最终使用。

7. 设计改进 (design improvement)

在开发过程中，对程序结构进行持续不断的梳理，在不影响程序的外部可见行为的情况下，对程序内部结构进行改进，保持代码简洁、无冗余。

8. 持续集成 (continuous integration)

持续集成是指每完成一个模块的开发后，立即将其组装到系统中，并进行集成测试。持续集成能保证项目组中所有开发好的模块始终是组装完毕、完成集成测试且是可执行的。

9. 代码集体共有 (collective code ownership)

代码集体共有是指团队中的任何人可以在任何时候修改系统任何位置上的任何代码。没有程序员对任何一个特定的模块或技术单独负责，每个人都可以参与任何其他方面的开发。

10. 编码标准 (coding standard)

XP 方法强调制定一个统一的编码标准，包括命名、注释、格式等，使系统中所有的代码看起来就好像是一个人编写的。

11. 系统比喻 (metaphor)

它是对需开发的软件的一种形象化的比喻，这种比喻描述了开发人员打算如何构建系统，起到概念框架的作用。这种比喻必须是每个团队成员都熟悉的。

12. 可持续的速度(sustainable pace)

由于人的精力是有限的,长时间超负荷的工作会影响工作效率,因此,XP方法要求每个团队队员都能始终保持精力充沛,只有持久才有获胜的希望。

极限编程是一组简单、具体的实践,这些实践结合在一起,即形成了一个敏捷开发过程。极限编程是一种优良的、通用的软件开发方法,项目团队可以拿来直接采用,也可以增加一些实践,或者对其中的一些实践进行修改后再采用。

2.6 软件可行性研究

可行性研究(feasibility study)的目的,是弄清待开发的项目是不是可能实现和值得进行,通常由系统分析员完成,并需写出可行性论证报告。如结论认为可行,即可制定项目实施计划,同时开始软件开发;如结论认为不可行,则应提出终止该项目的建议。

可行性论证其实是在高层次上进行的一次大大简化了的需求分析与设计。但它的目的不是去解决用户提出的问题,仅是确定这项开发是否值得进行,分析它存在哪些风险。换句话说,在投入大量资金前研究成功的可能性,减小可能出现的风险。即使研究的结论是不值得进行,所花的精力(约为开发费用的5%~10%)也并不白费,因为它避免了一次更大的浪费。

2.6.1 可行性研究的内容与步骤

1. 研究的内容

对研究中可能提出的任何一种解决方案,都要从经济、技术、运行和法律诸方面来研究其可行性,给出明确的结论供用户参考。

- ① 经济可行性。实现这个系统有没有经济效益?多长时间可以收回成本?
- ② 技术可行性。现有的技术能否实现这一新系统?有哪些技术难点?建议采用的技术先进程度怎样?
- ③ 运行可行性。为新系统规定的运行方式是否可行?例如,若新系统是建立在原来已担负其他任务的计算机系统上的,就不能要求它在实时在线的状态下运行,以免与原有的任务相矛盾。
- ④ 法律可行性。新系统的开发会不会在社会上或政治上引起侵权、破坏或其他责任问题?

2. 研究的步骤

可行性研究从理解系统所需解决问题的范围和目标开始,到提出关于新系统的推荐方案为止,通常要经过下列的步骤:

- ① 对当前系统进行调查和研究。这通常是了解一个陌生应用领域最快速的方法。新系统是从当前系统脱胎出来的,但又不是全盘照搬。分析员在调查中必须抓住关键,防止在不重要的细节上花费过多的时间。

② 导出新系统的解决方案。这一步的目的是根据新系统的逻辑模型，设想几种可能的解决方案，以使用户选择。在可供选择的方案中，既可以包括不同的开发方案，也可以包括购买现成软件，以及对现成软件进行改造等方案。每种方案都要先考虑在技术上是否可行，然后分析其经济与社会的可行性。

以上两步与需求阶段的工作颇为相似，但可行性研究要考虑多种解决方案，而需求阶段只需考虑已经选定的一种解决方案。

③ 提出推荐的方案。在对上一步提出的各种方案进行分析和比较的基础上，提出向用户推荐的方案。这时分析员应清楚地表明本项目的开发价值以及推荐这个方案的理由。

④ 编写可行性论证报告。可行性分析完成后，应写出可行性论证报告，交给用户和软件开发单位。报告主要由下面3个部分组成：

- 系统概述。包括对当前系统及其存在问题的简单描述，新的目标系统和它的各个子系统的功能与性能，新系统与当前系统的比较等。新系统可以用图形工具来描述，并附上重要的数据。

- 可行性分析。这是报告的主体。包括新系统在经济上、技术上和法律上的可行性，以及对建立新系统的主客观条件的分析。如有不止一种解决方案，应对几种方案进行比较，并指明推荐的方案。

- 结论意见。综合上述的分析，说明新系统是否可行。结论可区分为可立即进行、推迟进行以及不能或不值得进行这3类。

如果结论认为可行，用户也决定立即开发，下面就应开始制定软件项目计划了。

2.6.2 软件风险分析

软件开发存在着风险。软件风险具有不确定性，可能发生也可能不发生，但一旦风险变成现实，就会造成损失或产生严重后果。不同的软件其风险也不相同，项目规模越大、结构化程度越低、资源和成本等的不确定因素越大，承担这一项目所冒的风险也就越大。风险分析的任务，就是尽可能地量化不确定性的程度及每个风险导致的损失的程度，为软件开发的实施计划提供参考。如果在可行性研究阶段就进行风险分析，重视风险并且有所防范，就可以最大限度减少风险的发生与损失。

一般的说，软件风险分析可包括风险识别、风险预测和风险驾驭（或风险管理）等3项活动。

1. 风险识别

从宏观上说，风险可区分为项目风险、技术风险和商业风险。项目风险是指在预算、进度、人力、资源、客户及需求等方面潜在的问题，它们可能造成软件项目成本提高、时间延长等损失。技术风险是指设计、实现、接口和维护等方面的问题，以及由此造成的降低软件开发质量、延长交付时间等后果。商业风险包括市场、商业策略、推销策略等方面的风险，这些风险会直接影响软件的生存能力。

为了正确识别风险，可以将可能发生的风险区分为若干子类，每类建立一个风险项目检查表来识别它们。以下是常见的风险子类与需要检查的内容：

- ① 产品规模风险。检查与软件总体规模相关的风险。
- ② 商业影响风险。检查与管理与市场约束相关的风险。
- ③ 与客户相关的风险。检查与客户素质及通信能力相关的风险。
- ④ 过程风险。检查与软件过程定义和开发相关的风险。
- ⑤ 技术风险。检查与软件的复杂性及系统所包含技术成熟度相关的风险。
- ⑥ 开发环境风险。检查开发工具的可用性及与质量相关的风险。
- ⑦ 人员结构和经验风险。检查与参与工作的人员的总体技术水平及项目经验相关的风险。

以商业影响风险为例，其项目检查表中可能会包括下列问题：

- ① 建立的软件是否符合市场的需求（市场风险）？
- ② 建立的软件是否符合公司的整体商业策略（策略风险）？
- ③ 销售部门是否知道如何推销这种软件（销售风险）？
- ④ 有没有因为课题内容或人员的变动，使该项目失去管理层的支持（管理风险）？
- ⑤ 项目预算或参加人员有没有保证（预算风险）？

如果上述问题中有任何一个的答案是否定的，就可能出现风险，需要识别并预测可能产生的影响。

2. 风险预测

风险预测（risk forecast）又称为风险估计（risk estimation），一般包括两个方面的内容：风险发生的可能性；风险发生后所产生的后果。通常由参与评估风险的项目计划人员、管理人员和技术人员一起，执行以下两项风险预测活动。

（1）建立风险可能性尺度

风险可能性的尺度可以用定性或者定量的方式来定义。但一般不能用“是”或“否”来表示，较多的是使用概率的尺度，如极罕见、罕见、普通、可能或极可能等；这些概率可以从过去开发的项目、开发人员的经验或其他方面收集来的数据经统计分析估算出来。

（2）估计对产品和项目的影响

风险产生的后果通常使用定性的描述，如灾难性的、严重的、轻微的、可忽略的等。如果风险真的发生了，对产品和项目所产生的影响一般与3个因素有关，即风险的性质、范围及时间。风险的性质是指风险发生时可能产生的问题。例如，若与其他系统的接口定义得不好，就会影响软件的设计和测试，也可能导致系统集成时出现问题。风险的范围包括风险的严重性和分布情况。风险的时间是指风险的影响何时开始及风险会持续多长时间等。

对项目风险进行管理时，应综合考虑风险出现的概率和一旦发生风险可能产生的影响。对一个具有高影响但发生概率很低的风险不必花费很多的管理时间，而对于低影响但高概率的风险以及高影响且发生概率为中到高的风险，则应该优先列入需要管理的风险之中。针对不同风险概率及其可能产生后果的管理策略可参考图2.10。

Robert Charette 在他的《软件工程风险分析与管理》一书中，提出了一种主要依据风险描述、风险概率和风险影响 3 个因素对风险进行预测的方法，称之为风险评价 (risk assessment)。他用一个三元组

$$(r_i, l_i, x_i)$$

来表示这 3 个因素，其中 r_i 表示风险， l_i 表示风险发生的概率， x_i 则表示风险产生的影响。Charette 认为，各种风险通常会在成本、进度及性能等方面对软件项目产生影响。对于成本超支、进度延期和性能降低 (或它们的组合)，都应该有一个参考水平值，超过了某个具体值，即可视为项目的终止点，可能导致被迫终止计划开发的项目。因此在评价风险时，首先要确定风险影响的参考水平值，再建立三元组 (r_i, l_i, x_i) 与参考水平值之间的关系，然后确定项目终止点。图 2.11 显示了由成本超支和进度延期所构成的“项目终止”曲线。曲线右方为项目终止区域，曲线上的各点为临界区，项目终止或继续进行都是可以的。

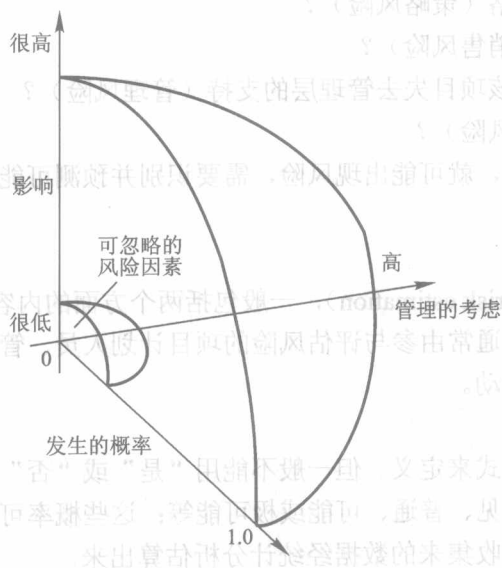


图 2.10 风险与管理的考虑

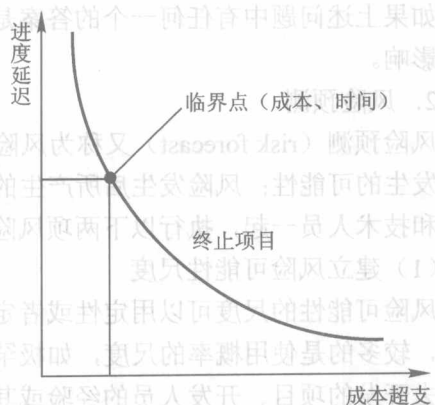


图 2.11 风险参考曲线

3. 风险的驾驭和监控

风险的驾驭与监控主要要靠管理者的经验来实施。例如，若某开发人员在开发期间中途离职的概率是 0.7，且离职后会对项目产生影响，则该风险驾驭和监控的策略如下：

- ① 与在职人员协商，了解其可能流动的原因。
- ② 在项目开始前，把缓解这些流动的相关工作列入风险驾驭计划。
- ③ 项目开始时，做好人员流动的准备，并采取措施确保一旦人员离开时，项目仍能继续。
- ④ 制定文档标准并建立一种机制，保证文档都能及时产生。
- ⑤ 对所有工作进行仔细审查，使更多的人能够按计划进度完成自己的工作。

⑥ 对每个关键性技术岗位注意培养后备人员。

风险驾驭与监控首先应该建立风险缓解、监控和管理计划 (risk mitigation, monitoring and management plan, RMMMP), 记录风险分析的全部工作, 并且作为整个项目计划的一部分为项目管理人员所使用。这些驾驭风险的措施会增加项目成本, 称之为风险成本。在考虑风险成本之后, 再决定是否采用上述策略。

2.6.3 项目实施计划

制定软件计划是计划时期的最后一项工作, 在软件已完成可行性分析、用户确定进行开发后进行。软件计划涉及实施项目的各个环节, 计划的合理性和准确性往往关系着项目的成败。针对不同的工作目标, 软件计划可分为以下 7 种主要类型:

① 项目实施计划。这是最重要的软件计划, 通常包括软件目标、功能、进度、资源和费用等多个方面。

② 质量保证计划。包括软件开发各个阶段的质量要求和质量保证活动。

③ 软件测试计划。规定各种测试活动的任务、方法、进度、资源和人员等。

④ 文档编制计划。规定项目开发各个阶段应编制的文档种类、标准和内容等。

⑤ 用户培训计划。包括对用户培训的目标、要求和进度等。

⑥ 综合支持计划。描述软件开发中所需的各个方面的支持, 以及如何获得和利用这些支持。

⑦ 软件分发计划。包括软件产品如何提交到最终用户手上。

在实际项目实践中, 根据软件规模可以选择制定一部分计划, 也可以将上述几个计划的内容合并在一个计划中。项目实施计划是一个综合性的计划, 一般不能省略。下面对它作进一步介绍。

项目实施计划是一种管理文档, 供软件开发单位使用。在开发过程中, 开发单位的管理人员根据这一计划安排和检查开发工作, 并可根据项目的进展情况定期进行必要的调整。

图 2.12 列出了项目实施计划的主要内容。实施计划一般不应写得太长、太复杂。只需把项目目标、开发周期以及所需资源和资金等写清楚即可。

<p>项目实施计划</p> <ol style="list-style-type: none"> 1. 系统概述 包括项目目标、主要功能、系统特点以及关于开发工作的安排。 2. 系统资源 包括开发和运行该软件系统所需要的各种资源, 如硬件、软件、人员和组织机构等。 3. 费用预算: 分阶段的人员费用、机时费用及其他费用。 4. 进度安排: 各阶段起止时间、完成文档及验收方式。 5. 要交付的产品清单
--

图 2.12 项目实施计划的主要内容

小结

软件开发模型是软件工程的重要内容之一，也是软件开发前需要确定的首要问题。随着软件工程的发展，陆续出现了瀑布模型、快速原型模型、增量模型、螺旋模型、转换模型、净室模型和构件集成模型等多种模型。各种软件开发模型各有优缺点。在选择软件开发模型时，开发者不仅应该了解开发过程的特点，还应该结合待开发系统的特点一起考虑。如有必要，也可以同时组合多种模型或创建新的模型。

20世纪末出现的统一过程和敏捷过程，是两种与前迥异的软件过程模型。本章仅对这两种过程作简单介绍。

本章末简叙了可行性研究的意义与内容。其目的是弄清待开发的项目有没有解，是不是值得去解。其重要意义在于，利用较短的时间和较小的代价（大约为开发费用的5%~10%）弄清投资的效益，以免冒太大的风险。

习题

1. 什么是软件生存周期？把生存周期划分为阶段的目的是什么？
2. 传统的瀑布模型把生存周期划分为哪些阶段？瀑布模型软件开发有哪些特点？
3. 简述文档和复审对于软件质量控制的作用。
4. 什么是快速原型法？其快速表现在哪里？
5. 实现快速原型法的最终系统可以有几种方法？请说明并加以比较。
6. 比较增量模型和螺旋模型的特点及异同。
7. 为什么利用转换模型开发软件有一定难度？什么是净室软件工程？
8. 哪些开发模型适用于面向对象的软件开发？
9. 比较螺旋模型和构件集成模型的异同。
10. 试比较 RUP 和 XP 的差异。
11. 可行性研究包含哪些内容？
12. 为什么要进行风险分析？

第3章 结构化分析与设计

经过 40 年的发展，软件工程已从第一代（传统软件工程）经历第二代（OO 软件工程）发展到第三代（基于构件的软件工程）。作为软件的一个重要组成部分——应用软件开发过程，也先后出现了瀑布模型、快速原型模型、增量模型、螺旋模型、转换模型、净室模型和构件集成模型等开发模型。从第 3 章起，将按照第一、二、三代软件的发展顺序，在前两章的基础上，依次展示 3 代软件工程的常用开发技术。

本章重点介绍基于瀑布模型的结构化分析与设计。为了方便读者学习，本章将相关的技术（例如模块设计）集中在一章中，并精简了一些过时的技术。

3.1 概述

3.1.1 结构化分析与设计的由来

结构化分析与设计最初是由结构化程序设计扩展而来的。

早在 20 世纪 70 年代中期，Stevens、Myers 与 Constantine 等人就在结构化程序设计的基础上，率先倡导了一种称为结构化设计（structured design, SD）的软件设计技术。20 世纪 70 年代后期，Yourdon 等人又倡导了与 SD 配套的结构化分析（structured analysis, SA）技术，合称为结构化分析与设计方法。它是第一代软件工程时期最有代表性的应用系统开发方法，不仅适用面广、流行时间长，而且与模块设计共同构成为第一代软件工程时期最为常用的技术，至今仍在某些特定类型的软件开发中有所应用。

1. 瀑布模型的首次实践

瀑布模型系由传统的生存周期过程演变而来。作为一种系统开发方法，结构化分析与设计是瀑布模型的首次实践。由第 2.2.1 节图 2.2 可见，该模型一般可划分为以下阶段：

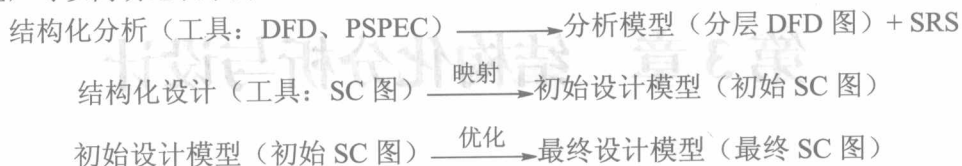
需求定义与分析→总体设计→详细设计→编码→测试→使用维护

其中，测试阶段又可细分为单元测试、综合（或集成）测试、确认测试及系统测试等子阶段。鉴于下文还将详细介绍（见 8.7 节），这里就不说了。

2. SA 与 SD 的流程

根据瀑布模型的上述流程，需求分析与软件设计是进入编码阶段前必须完成的任务。具体地说，系统开发从需求分析开始，首先建立系统的需求模型；接着通过 SD 方法提供的映

射规则，把分析模型转化为初始设计模型；然后再优化为系统的最终设计模型。系统的整个开发流程，可以简明地表示为：



简言之，SA 与 SD 的流程其实也是为待开发系统建立分析模型和设计模型的过程。

3. 基本任务与指导思想

(1) 结构化分析

SA 有两项基本任务，即建立系统分析模型 (analysis model) 和编写软件需求规格说明书 (software requirements specification, SRS)，二者都是分析阶段必须完成的文档。

① 建立分析模型。SA 模型包含描述软件需求的“一组”模型，通常有功能模型、数据模型和行为模型 3 种模型，分别表示待开发系统的功能需求、数据需求与行为需求。由于它们一般都用图形符号来表示，直观易懂，因而是形成 SRS 文档、完成软件设计的基础。

② 编写需求规格说明书。SRS 是分析阶段编写的、以文字为主的文档。当一个项目决定开发后，开发人员就应与用户共同确定软件的目标和范围 (见第 2.6.3 节“项目实施计划”)。在分析阶段，上述问题定义 (或称为问题陈述) 被进一步细化为 SRS。在国际标准 IEEE 830—1998 标准和中国国家标准 GB856D—88 中，都建议了 SRS 文档的主要内容，其中包括引言、信息描述、功能描述、行为描述、质量保证、接口描述以及其他需求等。这些标准还同时强调：

- SRS 应该具有准确性。任何微小的错漏都可能铸成大错，在纠正时需付出巨大的代价。
- SRS 应该防止二义性。不要采用用户不容易理解的专门术语，以免导致误解。
- SRS 应该直观易改。尽可能采用图形和符号，例如将分析模型作为附录放在 SRS 之后，使不熟悉计算机的用户也能一目了然。

③ 主要指导思想。抽象与分解，是结构化分析的主要指导思想。现实世界中的系统不论表现形式上怎样杂乱无章，总可以通过“分析与归纳”从中找出一些规律，再通过“抽象”构建系统的模型。由于软件工程是一种层次化的技术，所以抽象通常也可分层次进行。当需要获得系统的细节时，就应该移向低层次的抽象。抽象的层次愈低，呈现的细节也会愈多。

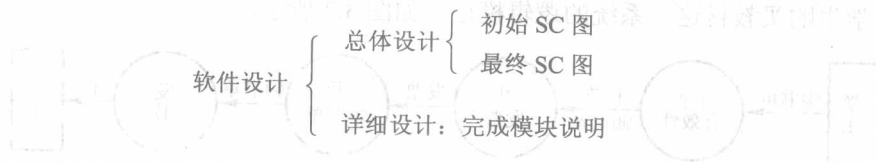
(2) 结构化设计

① 软件设计 = 总体设计 + 详细设计。由上述第一点可知，瀑布模型的软件设计包含了总体设计和详细设计两个阶段。SD 阶段把分析模型中的 DFD 图转换为最终 SC 图 (即图 2.2 中的“软件结构图”)，这仅仅完成了软件设计的第一步。在随后的详细设计中，还需用适当的工具对各个模块采用的算法和数据结构进行足够细致的描述 (即图 2.2 中的“模块说明”)，这也是模块设计阶段的基本任务。

结构化设计与模块设计相结合，即共同形成传统软件开发的常用技术。

② SC图需分两步完成。从上述SD的流程可知,结构化设计产生的SC图一般需分为两步完成:即首先通过“映射”获得初始SC图;然后通过“优化”获得最终SC图。

综合上述①、②两步,可以表示为:



③ 软件设计的指导思想。在软件开发的所有阶段中,软件设计是最富有活力、最需要发挥创造力的阶段。

分解和细化,历来是重要的软件设计策略。细化是与抽象相反而又互补的一对概念。1971年, N. Wirth 就发表了“用逐步细化 (stepwise refinement) 的方法开发程序”一文,指出程序设计是一个“渐进”的过程:“对于一个给定的程序,每一步都把其中的一条或数条指令分解为较多的更详细的指令。”Yourdon 称赞该文“开创了自顶向下设计的先河”,“虽然把一个大系统分为小片,然后又分为更小的小片在今天已众所周知,但在当时(20世纪70年代初)确实是一种革命的思想”。有人甚至誉之为“结构化程序设计的核心”。

细化的实质就是分解。在传统的软件开发中,“逐步细化”不仅相继应用于结构化程序设计、结构化设计和模块设计中,而且也扩展应用于结构化分析中(例如分层DFD图就是逐步细化的应用)。由此可见,它早已超出了设计策略的范畴,已成为问题求解的一种通用技术了。

3.1.2 SA模型的组成与描述

以下将结合一个引例,进一步对SA和SD两种模型及其描述工具进行说明。

[例 3.1] 引例:教材销售系统。

从用户调查中得知,在计划经济时期,某高校向学生销售教材的手续是:先由系办公室的张秘书开一购书证明,学生凭证明找教材科的王会计开购书发票,向李出纳交付书款,然后到书库找赵保管员领书。现欲将上述手工操作改用计算机处理,开发一个“教材销售系统”。试按照上述步骤进行需求分析。

[解] 如果把用户目前使用的系统称为“当前系统”,用计算机实现的系统称为“目标系统”,则本例的需求分析大体上可以按下述4步进行:

第一步:通过对现实环境的调查研究,获取当前系统的具体模型,如图3.1所示。

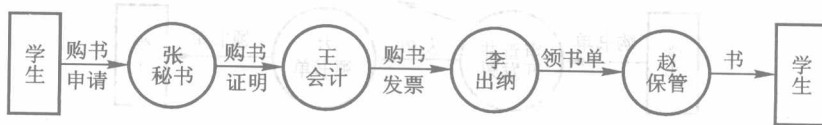


图 3.1 学生购买教材的当前系统模型

第二步：分析需求，建立系统分析模型，包括当前系统模型和目标系统模型。

① 去掉上述模型中的非本质因素，提炼出当前系统的逻辑模型。

在图 3.1 中，张、王、李、赵等具体的人是可能变动的，但需要他们处理的工作，例如审查购书有效性、开发票、开领书单等则是不变的，后者才是本质的内容。经过这样的分析，就可抽象出学生购买教材这一系统的逻辑模型，如图 3.2 所示。



图 3.2 学生购买教材的逻辑模型

② 分析当前系统与目标系统的差别，建立目标系统的逻辑模型。

目标系统是一个基于计算机的系统。一般说来，它的功能应该比当前的现行系统更强，不必也不应该完全模拟现行的系统。例如在销售教材的计算机系统中，“有效性审查”及“开发票”就可合并进行，省去“开有效购书单”这一手续，如图 3.3 所示。



图 3.3 目标系统的逻辑模型

第三步：整理综合需求，编写系统需求规格说明书。（此处从略）

第四步：验证需求，完善和补充对目标系统的描述。

① 通过目标系统的人机界面，和用户一起确认目标系统功能，主要是区分哪些功能交给计算机去做，哪些功能由人工完成。例如在图 3.3 所示的系统逻辑模型中，按照书费收款和发书这两项工作仍需由人工完成。

② 复审需求规格说明书，补充迄今尚未考虑过的细节，例如确定系统的响应时间，增加出错处理等。在本例中，假如购书单中出现了学生不该购买或已经卖完的教材，就可通过“无效书单”将相关的信息通知学生。

经过以上的修正和补充，即可得到改进后的目标系统逻辑模型，如图 3.4 所示。至此，销售系统的分析模型即告完成。它主要用图形符号来表达，在 SA 中称为 DFD 图。

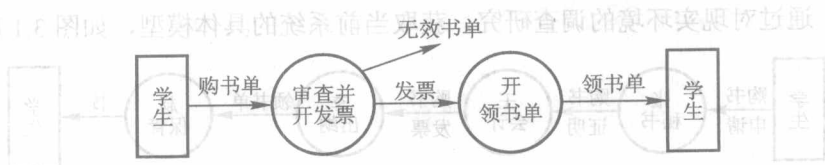


图 3.4 改进后的目标系统模型

以下将结合引例，对 SA 模型的组成及其常用描述工具举例说明。

1. SA 模型的组成

图 3.5 显示了 SA 模型的组成。由图可见，数据字典（data dictionary, DD）处于模型的核心，它是系统涉及的各种数据对象的总和。从 DD 出发可构建 3 种图：

① 实体联系图（entity-relation diagram, E-R 图）用于描述数据对象间的关系，它代表软件的数据模型，在实体联系图中出现的每个数据对象的属性，均可用数据对象说明来描述。

② 数据流图（data flow diagram, DFD）主要作用是指明系统中的数据是如何流动和变换的，以及描述使数据流进行变换的功能。在 DFD 图中出现的每个功能，则可在加工规格说明（process specification, PSPEC）中进行描述，它们一起构成软件的功能模型。

③ 状态变换图（status transform diagram, STD）用于指明系统在外来事件的作用下将如何动作，表明系统的各种状态以及状态间的变换（transfer，或称变迁），从而构成行为模型的基础。关于软件控制方面的附加信息，还可用控制规格说明（control specification, CSPEC）来描述。

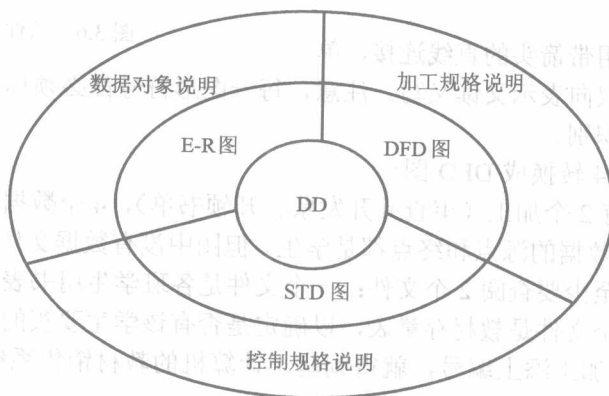


图 3.5 SA 模型的组成

需要指出，早期的 SA 模型仅包括 DD、DFD 和 PSPEC 等 3 个组成部分，主要用于描述软件的数据模型（用 DD 表示）与功能模型（用 DFD 和 PSPEC 表示）。随着社会信息的迅速发展，许多应用系统包含了较复杂的数据信息。于是在数据建模（data modeling）时，有人将原用于关系数据库设计的 E-R 图移用于 SA，以便描述具有复杂数据对象的信息模型。另一方面，随着计算机实时系统（real-time system）应用的不断扩大，人们在分析建模中发现，有些数据加工（data processing）不是由数据来触发，而是由实时发生的事件来触发/控制的，无法用传统的 DFD 图来表示。因而在 20 世纪 80 年代中期，以 Ward 和 Hatley 等为代表的学者又在功能模型之外扩充了行为模型，推荐用控制流图（control flow diagram, CFD）、CSPEC 和 STD 等工具进行描述。今天，SA 模型已可同时覆盖信息模型、功能模型和行为模型等 3 种模型，其适用的软件范围也更加扩大了。

2. SA 模型的描述工具

综上所述，SA 模型的组成及其常用描述工具可以归结为：

- ① DFD、DD 和 PSPEC。它们是早期 SA 模型的基本组成部分。
- ② CFD、CSPEC 和 STD。它们是早期 SA 模型的扩展成分，可适应实时软件的建模需要。
- ③ E-R 图。适用于描述具有复杂数据结构的软件数据模型。

现分别例示如下。

(1) 数据流图 (DFD)

任何软件系统（或计算机系统）从根本上来说，都是对数据进行加工（processing）或变换（transform）的工具。图 3.6 是一个高度抽象了的软件系统功能模型。

① 组成符号。数据流图只使用 4 种基本图形符号：圆框代表加工；箭头代表数据的流向，数据名称总是标在箭头的边上；方框表示数据的源点和终点；双杠（或单杠）表示数据文件或数据库。



图 3.6 软件功能模型

文件与加工之间用带箭头的直线连接，单向表示只读或只写，双向表示又读又写。注意，每一图形符号都必须标上名字，加工框还应该加上编号，以帮助识别。

【例 3.2】把图 3.4 转换成 DFD 图。

【解】图 3.4 现有 2 个加工（审查并开发票、开领书单），4 个数据流（购书单、发票、领书单、无效书单），数据的源点和终点都是学生。但图中没有数据文件。实际上在审查购书单和开出发票之前，至少要查阅 2 个文件：一个文件是各班学生用书表，用以核对学生是否需用这些教材；另一个文件是教材存量表，以确定是否有该学生要买的教材。把这 2 个文件加到图 3.4 中，并给加工添上编号，就得到基于计算机的教材销售系统的 DFD，如图 3.7 所示。

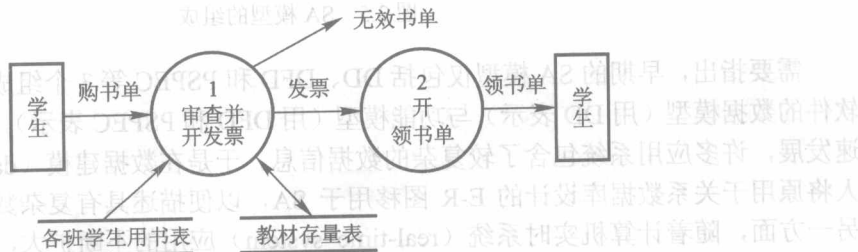


图 3.7 教材销售系统的数据流图

加工 1（即“审查并开发票”）要从教材存量表中读出数据，以判断有没有可卖的教材；售出教材后又要原存量中减去售出的数量，把新存量写回教材存量表，所以在加工 1 与教材存量表之间使用了带双箭头的连线。各班学生用书表只读不写，应用单向箭头连线连接，

图中箭头的方向表示从文件读出。

② DFD 的性质。与程序流图 (flow diagram) 不同, DFD 不能表示程序的控制结构, 例如选择结构或循环结构等。前一种图形用于表示程序的模块设计 (procedural design), 后一种图形则用作软件分析阶段的工具。由于分析阶段只需考虑软件“干什么”, 不必过问“怎么干”, 当然不应包括控制流、控制结构或激发条件之类的信息。

DFD 所表现的范围, 可大到整个系统, 小到一个模块。在需求分析中, 常常用一组 DFD 图由粗到精地表示同一软件在不同抽象级别上的功能模型, 并称之为分层数据流图 (leveled DFD, 参阅下文第 3.2.1 节)。

(2) 数据字典 (DD)

一个软件系统含有许多数据。数据字典的作用, 就是对软件中的每个数据规定一个定义条目, 以下结合图 3.7 的教材销售系统举例说明。

① 数据流。

[例 3.3] 以图 3.7 中的“发票”为例, 编写一个字典条目。

[解] “发票”是一个数据流, 其条目内容与书写格式如表 3.1 所示。

表 3.1 数据流“发票”的字典条目

数据流名: 发票
别 名: 购书发票
组 成: 学号+姓名+ {书号+单价+数量+总价} +书费合计
备 注:

在组成栏中, “学号”、“姓名”、“书号”等都是数据项的名称。就本例而言, “学号”、“姓名”与“书费合计”在数据流中仅出现一次, 其余各数据项则每购买一种书就要出现一次。条目中用 {} 表示重复。在条目内容中列入“别名”, 是因为对同一数据可能存在不同的称呼, 并不是说允许同一数据在系统中使用不同的名字。

② 数据文件。

[例 3.4] 为教材销售系统中的各班学生用书表编写一个字典条目。

[解] 如表 3.2 所示, 在该条目中, 每个记录记载一个班在一学年中需用的教材。“组织”是数据文件条目所特有的内容, 用于说明文件中的记录将按照什么规则组合成文件。

表 3.2 数据文件“各班学生用书表”的字典条目

文件名: 各班学生用书表
组 成: {系编号+专业和班编号+年级+ {书号}}
组 织: 按系、专业和班编号从小到大排列
备 注:

③ 数据项。

[例 3.5] 数据项字典条目示例。

[解] 表 3.3 及表 3.4 分别列出了两个数据项字典条目：“数量”及“书费合计”。

表 3.3 数据项“数量”的字典条目

数据项名：数量
别名：购书量
取值：正整数
备注：

表 3.4 数据项“书费合计”的字典条目

数据项名：书费合计
别名：大面
取值：00.00~99.99
备注：

(3) 加工规格说明

加工规格说明通常用结构化语言 (structured language)、判定表 (decision table) 或判定树 (decision tree) 作为描述工具。每个加工规格说明可以像字典中的条目一样记在卡片上。以下将分别对加工规格说明卡片常用的 3 种描述手段作简单介绍。

① 结构化语言。

[例 3.6] 在图 3.7 的教材销售系统中，使用结构化语言来描述加工 1 (其功能是“审查并开发票”) 的加工逻辑。

[解] 图 3.8 显示了用结构化语言描述的加工规格说明。

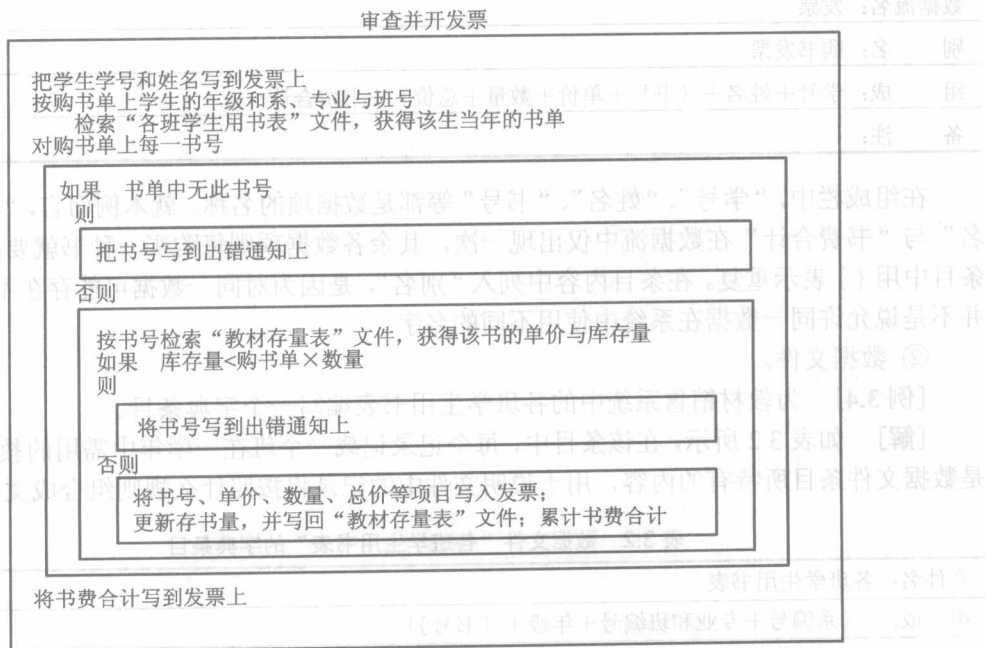


图 3.8 “审查并开发票”加工逻辑

② 判定表或判定树。

【例3.7】某公司为推销人员制定了奖励办法，把奖金与推销金额及预收货款的数额挂钩。凡每周推销金额不超过 10 000 元的，按预收货款是否超过 50%，分别奖励推销额的 6% 或 4%。若推销金额超过 10 000 元，则按预收货款是否超过 50%，分别奖励推销额的 8% 或 5%。对于月薪低于 1 000 元的推销员，分别另发鼓励奖 300、200 和 500、300 元。

试分别采用判定表和判定树为 DFD 图中用来“计算奖金”的加工写出 PSPEC。

【解】图 3.9 与图 3.10 分别显示了用判定表和判定树描述的 PSPEC，二者的含义相同。

推 销 奖 金 策 略				
规 则				
条 件	1	2	3	4
推 销 金 额	>10 000	≤10 000	>10 000	≤10 000
预 收 货 款	>50%	>50%	≤50%	≤50%
动 作				
置 奖 金 率 为	8%	6%	5%	4%
置 奖 金 额 = 奖 金 率 × 推 销 金 额				
如 果 推 销 员 月 薪 低 于 1 000 元				
另 加 奖 金 额	500	300	300	200

图 3.9 兼有结构化语言的判定表

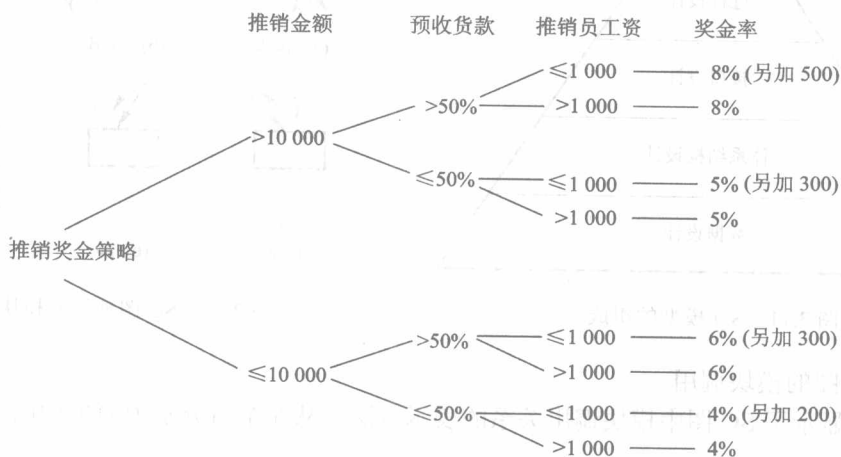


图 3.10 判定树示例

3.1.3 SD模型的组成与描述

1. SD模型的组成

SD模型是由SA模型映射而来的。例如在结构化设计中，SA模型的数据字典可转换为待开发系统的数据设计；数据流图可转换为体系结构设计（SC图）与接口设计；加工规格说明可转换为模块内部的详细过程设计；等等。图3.11显示了设计模型的组成。

2. SD模型的描述工具

在图3.11中，由下向上包含了数据设计、体系结构设计、接口设计与过程设计。顾名思义，体系结构设计是用来确定软件结构的，其描述工具为结构图（structure chart），简称SC图。过程设计主要指模块内部的详细设计，其描述工具将在“3.4 模块设计”中一并介绍。

(1) SC图的组成符号

在SC图中，用矩形框来表示模块，带箭头的连线表示模块间的调用，并在调用线的两旁标出传入和传出模块的数据流。图3.12显示了SC图允许使用的6种模块。其中，传入、传出和变换模块用来组成变换结构中的各个相应部分；源模块是不调用其他模块的传入模块，只用于传入部分的始端；漏模块是不调用其他模块的传出模块，仅用于传出部分的末端；控制模块是只调用其他模块，不受其他模块调用的模块，例如变换型结构的顶层模块，事务型结构的事务中心等，均属于这一类。

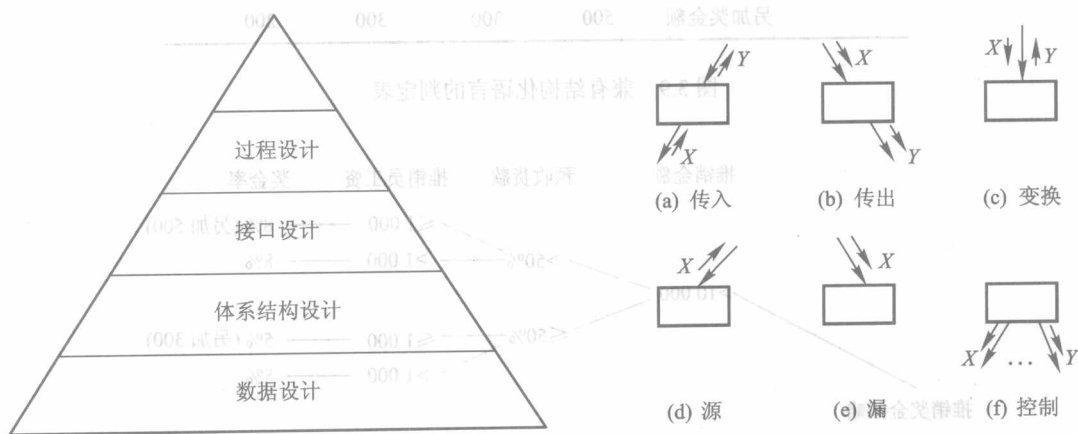


图 3.11 SD模型的组成

图 3.12 SC图使用的模块符号

(2) SC图的模块调用

图3.13显示了SC图中模块调用关系的表示方法，从左至右分别为简单调用、选择调用和循环调用。

为了画面简洁，在图3.13中调用线的两旁均未标出数据流。在实际的SC图中不允许这种省略。

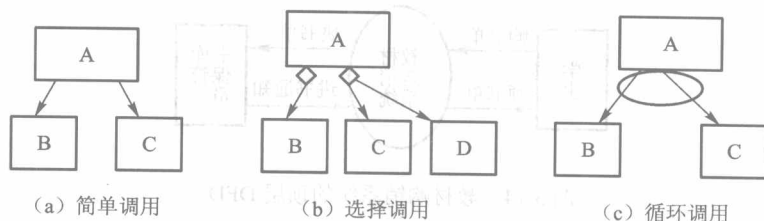


图 3.13 SC 图中模块调用关系的表示

3.2 结构化系统分析

按照 T. DeMarco 的定义,“结构化分析就是使用 DFD、DD、结构化英语、判定表和判定树等工具,来建立一种新的、称为结构化说明书的目标文档”。这里的结构化说明书就是 SRS。随着计算机应用的发展,有些学者还提出了用于结构化分析的补充工具,使之适用于实时系统的结构化分析模型,例如 CFD、CSPEC、STD 等。

结构化分析的基本步骤是:自顶向下对系统进行功能分解,画出分层 DFD 图;由后向前定义系统的数据和加工,编制 DD 和 PSPEC;最终写出 SRS。以下仍结合实例进行说明。

3.2.1 画分层数据流图

大型复杂的软件系统,其 DFD 可能含有数百乃至数千个加工,不可能一次将它们画完整。正确的做法是:从系统的基本功能模型(把整个系统看成一个加工)开始,逐层地对系统进行分解。每分解一次,系统的加工数量就增加一些,加工的功能也更具体一些。继续重复这种分解,直到所有的加工都足够简单为止。通常把这种不需再分解的加工称为“基本加工”,把上述逐步分解称为“自顶向下、逐步细化”(top-down stepwise refinement),最终为待开发的系统画出一组分层的数据流图,以代替一张含有系统全部加工的包罗万象的总数据流图。

【例 3.8】 将图 3.7 的教材销售系统扩展为教材购销系统。

例 3.2 的教材销售系统只支持销售,不支持采购。为弥补这一缺陷,拟扩展该系统的功能,使之能根据教材存量表,以“缺书单”的形式通知书库保管员,并在采购后用“进书通知”更新教材存量。试用 SA 方法,为扩展后的教材购销系统画出分层 DFD 图。

【解】 画系统分层数据流图的第一步,是画出顶层图。通常把整个系统当作一个大的加工,标明系统的输入、输出以及数据的源点与终点(统称为“外部项”)。图 3.14 显示了教材购销系统的顶层图。它表明,系统从学生那里接受购书单,经处理后把领书单返回给学生,使学生可凭单到书库领书。对脱销的教材,系统用缺书单的形式通知书库保管员;新书进库后,也由保管员将进书通知返回给系统。

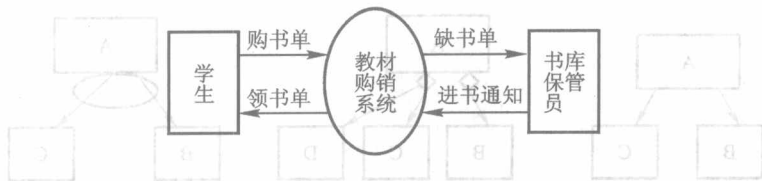


图 3.14 教材购销系统的顶层 DFD

接下来画第二层 DFD 图，把系统分解为销售和采购两大加工，如图 3.15 所示。显然，外部项学生应与销售子系统联系，外部项书库保管员应与采购子系统联系。两个子系统之间也存在两项数据联系：其一是缺货登记表，由销售子系统把脱销的教材传送给采购子系统；其二是进书通知，直接由采购子系统将教材入库信息通知销售子系统。

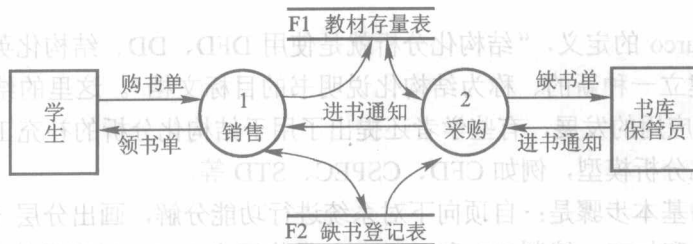


图 3.15 教材购销系统的第二层 DFD 图

继续分解，可获得第三层数据流图。其中图 3.16 由销售子系统扩展而成，图 3.17 由采购子系统扩展而成。

在图 3.16 中，销售子系统被分解为 6 个子加工，编号从 1.1 至 1.6。审查有效性时，首先要校核购书单的内容是否与学生用书表 (F3) 相符，还要通过售书登记表 (F4) 检查学生是否已买过教材。若发现购书单与学生用书不符或已买过该教材，应发出无效书单，只将通过审查的教材留在有效购书单中。“开发票”加工按照购书单查对教材存量表 (F1)，把可供的教材写入发票，数量不足的教材写入缺货单。前者在 F4 中登记后开出领书单发给学生，后者则登记到缺货登记表 (F2)，待接到进书通知后再补售给学生。补售的手续及数据流程和第一次购书相同。这里要注意的是，在上一层 DFD (图 3.15) 中，采购是系统内部的一个加工框，但在图 3.16 中，“采购”却是处于销售之外的一个外部项。

采购子系统在图 3.17 中被分解为 3 个子加工。由销售子系统建立起来的缺货登记表 (F2)，首先按书号汇总后登入待购教材表 (F5)，然后再按出版社分别统计制成缺货单，送给书库保管员作为采购教材的依据。在汇总缺货时要再次核查教材存量表 (F1)，按出版社统计时还要参阅教材一览表 (F6)，从而确定所缺教材是哪个出版社出版的。新书入库后，要及时修改教材存量表和待购教材表中的有关教材数量，同时把进书信息通知销售子系统，使销售人员能通知缺货的学生补买。

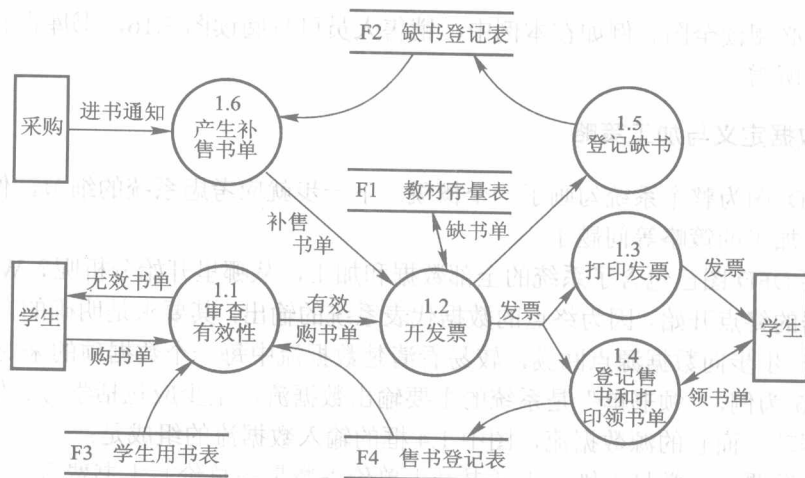


图 3.16 第三层 DFD 图——销售子系统

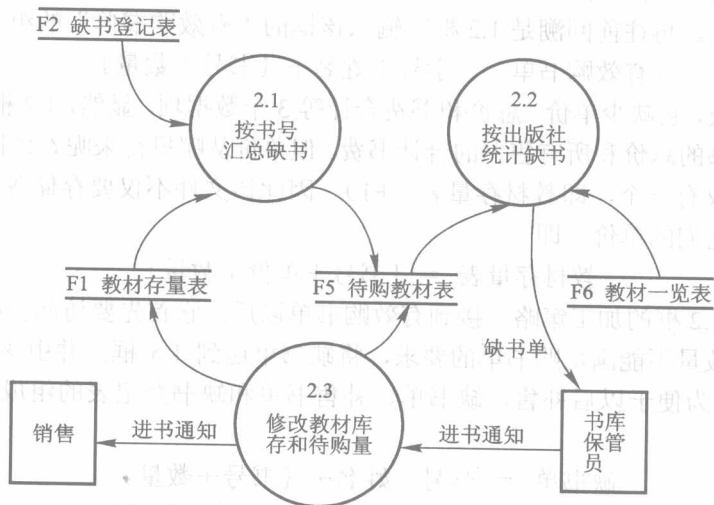


图 3.17 第三层 DFD 图——采购子系统

以上共画了 3 层、4 个 DFD 图 (图 3.14~图 3.17), 组成了教材购销系统的分层 DFD 图。愈到下层加工愈细。第三层的加工框已是足够简单的“基本加工”, 不必再分解了。

由本例可见, 分层 DFD 具有下列优点:

① 便于实现。采用逐步细化的扩展方法, 可避免一次引入过多细节, 有利于控制问题的复杂度。

② 便于使用。用一组图代替一张总图, 使用户中的不同业务人员可各自选择与本身有

关的图形，不必阅读全图。例如在本例中，销售人员可只阅读图 3.16，书库保管员可只阅读图 3.17，各取所需。

3.2.2 确定数据定义与加工策略

分层 DFD 图为整个系统勾画了一个概貌。下一步就应考虑系统的细节，例如定义系统的数据、确定加工的策略等问题了。

最低一层 DFD 图已包含了系统的全部数据和加工，从哪里开始分析呢？W. Davis 认为，一般应从数据的终点开始。因为终点的数据代表系统的输出，其要求是明确的。由这里开始，沿着 DFD 图一步步向数据源点回溯，较易看清楚数据流中每一个数据项的来龙去脉。

以图 3.16 为例，“领书单”是系统的主要输出数据流，至少应包括学号、姓名、书号和数量 4 个数据项。而它的源数据流，图中 1.4 框的输入数据流的组成是：

发票 = 学号 + 姓名 + {书号 + 单价 + 数量 + 总价} + 书费合计

可见领书单中的内容都能在发票中找到。1.4 框的策略之一，就是从发票中选择有用的数据项写入领书单。其次，该框还要登记售书，防止学生重复购买，所以售书登记表（F4）组成应与领书单组成相同。再往前回溯是 1.2 框，输入该框的“有效购书单”的组成内容是：

有效购书单 = 学号 + 姓名 + {书号 + 数量}

与发票的组成比较，它缺少单价、总价和书费合计等 3 个数据项。显然，1.2 框在开出发票前，必须先计算每种书的总价和所有售书的合计书费。但单价从哪里得来呢？由图 3.16 可知，1.2 框可访问的文件仅有一个，即教材存量表（F1）。因此该文件不仅要存储各种教材的现有数量，还应该包括它们的单价。即

教材存量表 = {书号 + 单价 + 数量}

现在再考察 1.2 框的加工策略。接到有效购书单以后，它首先要访问教材存量表（F1），查清有哪些书的数量不能满足购书单的要求，将缺书单送到 1.5 框，并由该框把信息存入缺书登记表（F2）。为便于以后补售，缺书单、补售书单和缺书登记表的组成，都可以与有效购书单相同，即：

缺书单 = 学号 + 姓名 + {书号 + 数量}

补售书单 = 学号 + 姓名 + {书号 + 数量}

缺书登记表 = {学号 + 姓名 + {书号 + 数量}}

对当前可以供应的教材，一方面要计算每种书的总价和书费累计，另一方面要更改存书数量，把剩余书数量写回教材存量表。有效购单书的上述数据流程及处理方法，也适用于补售书单，不再重复。

继续回溯，就可循有效购书单找到 1.1 框，或者循补售书单找到 1.6 框，分别得出这两个框的加工策略和各个有关数据的定义。分析的方法与以上相似，就不详细说明了。

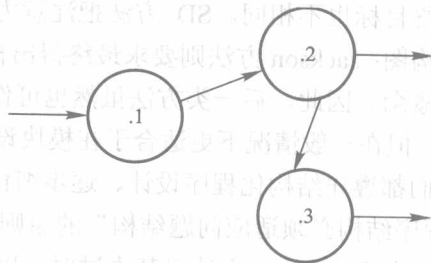
上述分析结束后，分析员应为 DFD 的每个数据逐一写出定义，每个基本加工逐个进行加工规格说明，并汇编成数据字典。

综上所述, 分层 DFD 图产生了系统的全部数据和加工, 通过对这些数据和加工的定义, 常常对分析员提出一些新问题 (例如前述的教材单价应从哪里取得), 促使他们进行新的调查和思考, 并可能导致对 DFD 的修改。画 DFD, 定义加工和数据, 再画, 再定义, 如此循环, 直至产生一个为用户和分析员一致同意的文档——SRS。

3.2.3 需求分析的复审

需求分析的文档完成后, 应由用户和系统分析员共同进行复审, 并吸收设计人员参加。

复审的重点, 是 DFD、DD 和加工规格说明等文档的完整性、易改性和易读性, 尽量多地发现文档中存在的矛盾、冗余与遗漏。例如, 要注意 DFD 图的加工编号: 通常顶层加工不编号, 第二层加工编为 1, 2, …, n 号, 第三层编为 1.1, 1.2, 1.3, …, n.1, n.2, n.3, 依此类推。与此相应, 各层 DFD 的图号是: 顶层 DFD 图无图号, 第二层编为图 0; 第三层编为图 1、图 2, 依此类推直至图 n。当层次较多时, 编号允许用简化方法表示, 如图 3.18 所示。



DFD图号: 3.6.5
加工编号: .1相当于 3.6.5.1
 .2相当于 3.6.5.2
 .3相当于 3.6.5.3

图 3.18 加工编号的简化

3.3 结构化系统设计

软件的需求分析完成后, 就可以开始软件设计了。同需求分析一样, 软件设计目前也有两种主流方法, 即基于结构化程序设计的结构化软件设计和基于面向对象技术的面向对象软件设计。

3.3.1 SD 概述

前已指出, SD 方法是率先由 Stevens、Myers 与 Constantine 等人在 20 世纪 70 年代中期倡导的。

1. 面向数据流设计和面向数据设计

按照出发点的不同, 传统的软件设计又可细分为面向数据流的设计和面向数据 (或数据结构) 的设计两大类。前者以 SD 方法为主要代表, 后者以 Jackson 方法为主要代表。

在面向数据流的方法中, 数据流是考虑一切问题的出发点。以 SD 方法为例, 在与之配套的 SA 方法中, 通常用数据流图来表示软件的逻辑模型; 在设计阶段, 又按照数据流图的不同类型 (变换型或事务型) 将它们转换为相应的软件结构。Jackson 方法则不同, 它以数据结构作为分析与设计的基础。众所周知, 算法和数据结构是传统程序设计中不可分割的两个侧面。根据 Hoare 的研究 (详见 “Notes on Data Structures”, 1972), 算法的结构在很大程度上

上依赖于它要处理的数据结构。例如当问题的数据结构具有选择性质时，就需用选择结构来处理；如果数据结构具有重复性质，就需用循环结构来处理；分层次的数据结构总是导致分层次的程序结构；等等。因此，如果事先知道了问题的数据结构，即可由此导出它的程序结构。这是面向数据结构设计方法的根据与基本思想。

基于数据流还是基于数据结构，标志了两类设计方法的不同出发点；不仅如此，它们的最终目标也不相同。SD 方法把注意力集中在模块的合理划分上，其目标是得出软件的体系结构图；Jackson 方法则要求最终得出程序的过程性描述，并不明确提出软件应该先分成模块等概念。因此，后一类方法虽然也可作为独立系统设计方法应用于小规模数据处理系统的开发，但在一般情况下更适合于在模块设计阶段使用。这两类方法也存在着许多共同点。例如，它们都遵守结构化程序设计、逐步细化等设计策略；都要从分析模型导出设计模型；并服从“程序结构必须适应问题结构”的原则。

鉴于 Jackson 方法已基本过时，以下不再展开介绍，本章仅讨论 SD 方法。

2. 从分析模型导出设计模型

设计是把用户的需求准确地转换为软件产品或系统的唯一方法。无论是传统的设计或面向对象的设计，都要从分析阶段得到的分析模型导出软件的设计模型。Pressman 用简明的图形说明了这种导出关系，如图 3.19 所示。

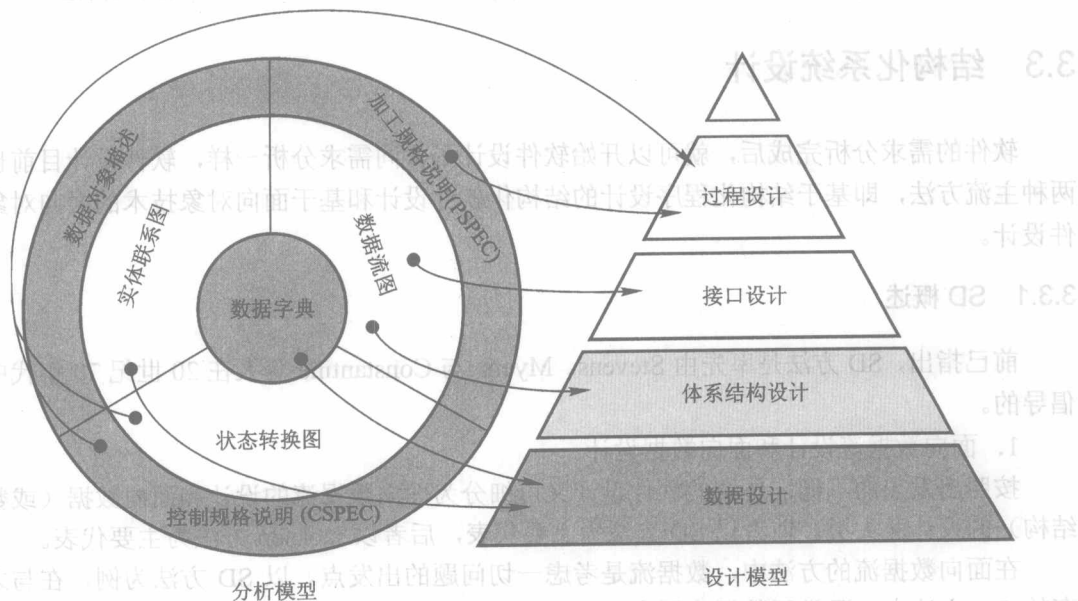


图 3.19 将分析模型转换为软件的设计模型

由图 3.19 可见，传统软件设计所产生的数据设计、体系结构设计、接口设计和过程设计，

均可从分析模型所包含的各种图形和说明中找出所需的信息。例如，体系结构与接口设计可以由数据流图导出，过程设计可根据加工规格说明、控制规格说明和状态转换图来定义，等等。其中不少系统设计方法都提供了将分析描述直接转换为设计描述的映射（mapping）规则，使软件设计变得更加容易。

3.3.2 SD 的步骤：从 DFD 图到 SC 图

结构化软件的设计，通常从 DFD 图到 SC 图的映射开始。

1. 数据流图的类型

从 SA 获得的 DFD 中，所有系统可归结为变换型结构和事务型结构两种类型。

(1) 变换型结构

由传入路径、变换中心和传出路径 3 部分组成。图 3.20 显示了该型结构的基本模型和数据流。

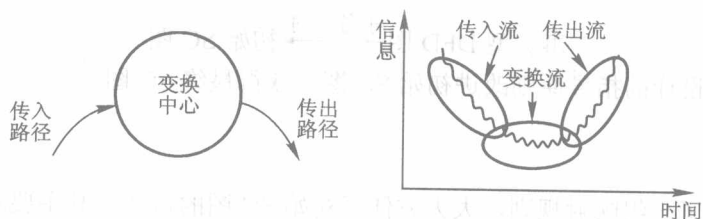


图 3.20 变换型结构的系统

(2) 事务型结构

事务一词常见于商业数据处理系统，一笔账目，一次交易，都可以看作一次事务。在更广泛的意义上，一次动作、事件或状态变化也可成为一次事务。事务型结构由至少一条接受路径、一个事务中心与若干条动作路径组成，其基本模型如图 3.21 所示。当外部信息沿着接受路径进入系统后，经过事务中心获得某一个特定值，就能据此启动某一条动作路径的操作。

在一个大型系统的 DFD 中，变换型和事务型结构往往同时存在。例如在图 3.22 所示的 DFD 中，系统的总体结构是事务型的，但是在它的某（几）条动作路径中，很可能出现变换型结构。在另一些情况下，在整体为变换型结构的系统中，其中的某些部分也可能具有事务型结构的特征。

2. SD 方法的步骤

为了有效地实现从 DFD 图到 SC 图的映射，结构化设计规定了下列 4 个步骤：

① 复审 DFD 图，必要时可再次进行修改或细化。

② 鉴别 DFD 图所表示的软件系统的结构特征，确定它所代表的软件结构是属于变换型还是事务型。

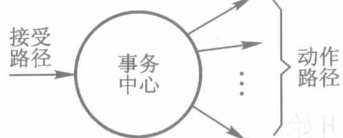


图 3.21 事务型结构的基本模型

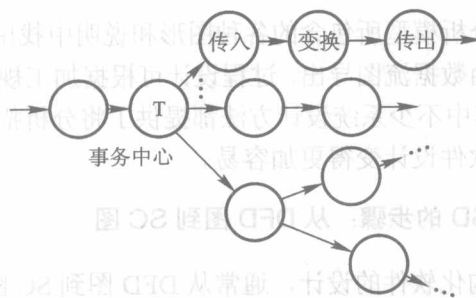


图 3.22 同时存在两类结构的系统

③ 按照 SD 方法规定的一组规则，把 DFD 图映射为初始 SC 图。

变换型 DFD 图 $\xrightarrow{\text{变换映射}}$ 初始 SC 图

事务型 DFD 图 $\xrightarrow{\text{事务映射}}$ 初始 SC 图

④ 按照优化设计的指导原则改进初始 SC 图，获得最终 SC 图。

3.3.3 变换映射

SD 方法提供了一组映射规则，大大方便了初始 SC 图的设计。其主要步骤包括：

- ① 划分 DFD 图的边界。
- ② 建立初始 SC 图的框架。
- ③ 分解 SC 图的各个分支。

【例 3.9】用变换映射规则从图 3.23 导出初始 SC 图，假设该图已经过细化与修改。

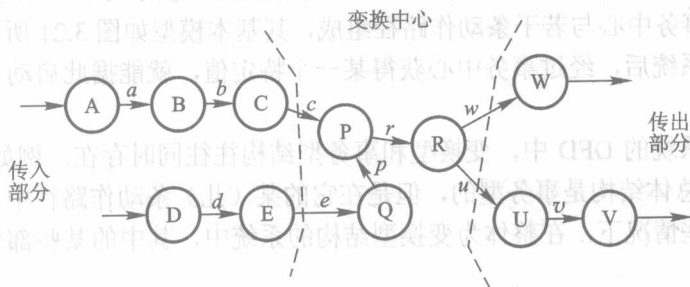


图 3.23 在 DFD 图上划分传入、传出和变换中心部分

【解】第一步：区分传入、传出和变换中心 3 个部分，在 DFD 图上标明它们的分界线。

第二步：完成第一级分解，建立初始 SC 图的框架，如图 3.24 所示。

第三步：完成第二级分解，细化 SC 图的各个分支。

在图 3.25 (a) 中，传入模块 M_A 直接调用模块 C 与 E，以取得它所需的数据流 c 与 e 。

继续下推，模块 C、E 将分别调用下属模块 B、D，以取得 b 与 d ；模块 B 又通过下属模块 A，取得数据 a 。

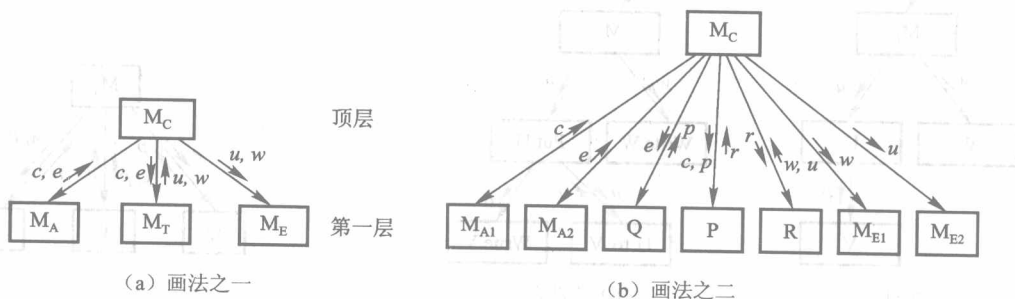


图 3.24 第一级分解后的 SC 图

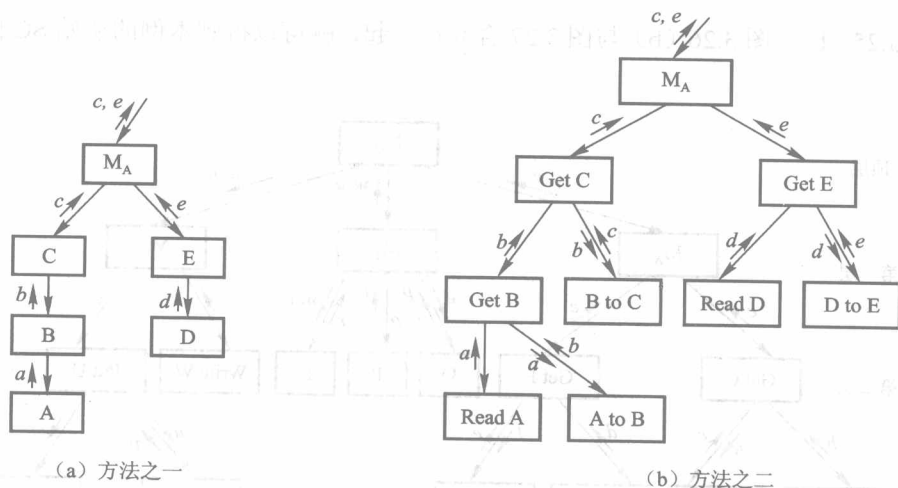


图 3.25 传入分支的分解

实际上数据流在传入的过程中，也可能经历数据的变换。以图 3.23 中的两个传入流为例，其中一路从 a 变换为 b ，再变换为 c ；另一路则从 d 变换为 e 。为了显式地表示出这种变换，可以在图 3.25 (a) 中增添 3 个变换模块，分别是“ A to B ”、“ B to C ”、“ D to E ”，并在模块名称前加上 Read、Get 等字样，如图 3.25 (b) 所示。这一改变的实质是，除了处于物理输入端的源模块以外，让每一传入模块都调用两个下属模块，包括一个传入模块和一个变换模块。图 3.25 (b) 所示的结构，显然较图 3.25 (a) 更加清楚、明了。

仿照与传入分支相似的分解方法，可得到传出分支的两种模块分解图，如图 3.26 所示。

与传入、传出分支相比，变换中心分支的情况繁简迥异，其分解也较复杂。但建立初始的 SC 图时，仍可以采取“一对一映射”的简单转换方法。图 3.27 显示了本例变换中心分支

第二级分解的结果。

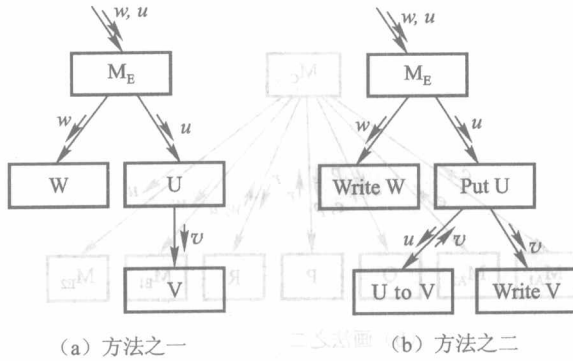


图 3.26 传出分支的分解

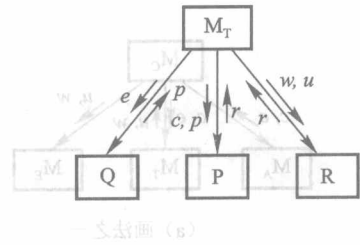


图 3.27 变换中心分支的分解

将图 3.25 (b)、图 3.26 (b) 与图 3.27 合并在一起，就可以得到本例的初始 SC 图，如图 3.28 所示。

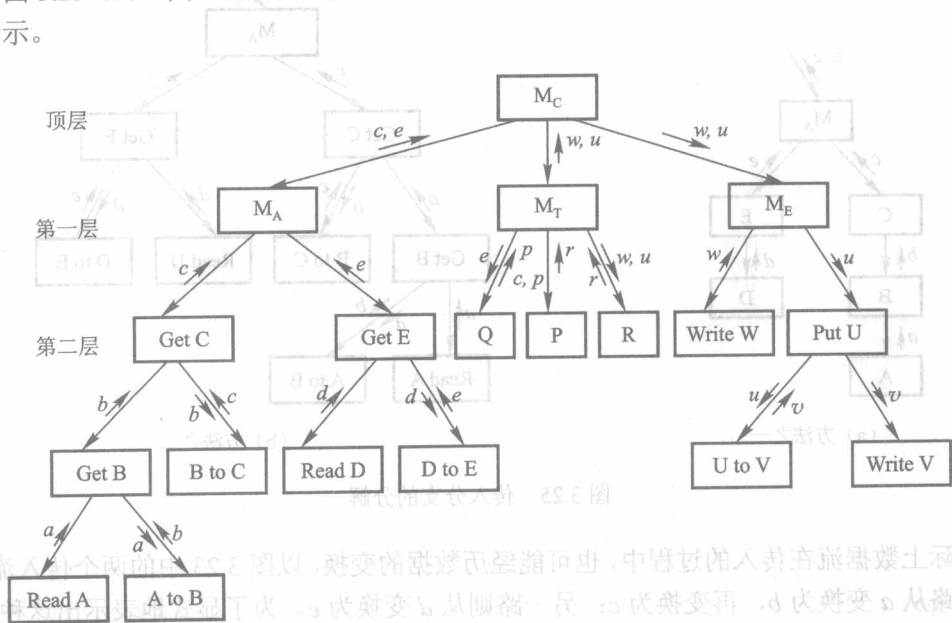


图 3.28 从图 3.23 导出的初始 SC 图

3.3.4 事务映射

与变换映射相似，事务映射也可以分为 3 个步骤：

① 在 DFD 图上确定事务中心、接受部分（包括接受路径）和发送部分（包括全部动作路径）。

② 画出 SC 图框架，把 DFD 图的 3 个部分分别映射为事务控制模块、接受模块和动作发送模块。

③ 分解和细化接受分支和发送分支，完成初始的 SC 图。

【例 3.10】 用事务映射方法，从图 3.29 所示的 DFD 图导出初始的 SC 图。

【解】 事务中心通常位于 DFD 图中多条动作路径的起点，向事务中心提供启动信息的路径，则是系统的接受路径。动作路径通常不止一条，可具有变换型或另一个事务型的结构。

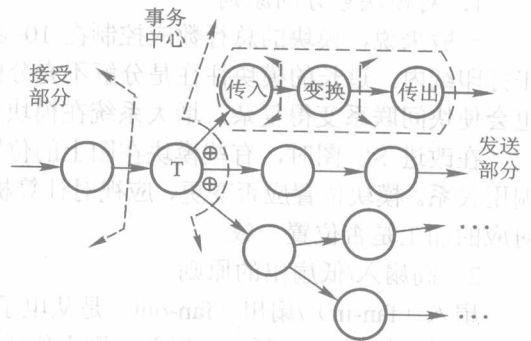


图 3.29 事务型 DFD 图的划分

第一步：划分 DFD 图的边界，并做出标记，如图 3.29 所示。

第二步：画出相应 SC 图的初始框架。图 3.30 (a) 显示了由二层组成的典型结构。

如果第一层的模块比较简单，也可以并入顶层，如图 3.30 (b) 所示。

第三步：重点是对动作（即发送）分支进行分解。其接受分支因一般具有变换特性，可以按变换映射对它进行分解。

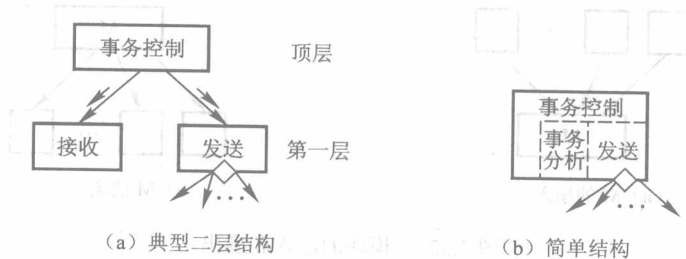


图 3.30 事务型 SC 图的上层结构

图 3.31 显示了动作分支的典型结构，含有 P、T、A、D 共 4 层。P 为处理层，相当于图 3.30 中的发送模块。T 为事务层，每一动作路径可映射为一个事务模块。在事务层以下可以再分解出操作层（actions 层）及细节层（details 层）。由于同一系统中的事务往往含有部分相同的操作，各操作又可能具有部分相同的细节，这两层的模块常能为它们的上层模块所共享，被多个上级模块调用。

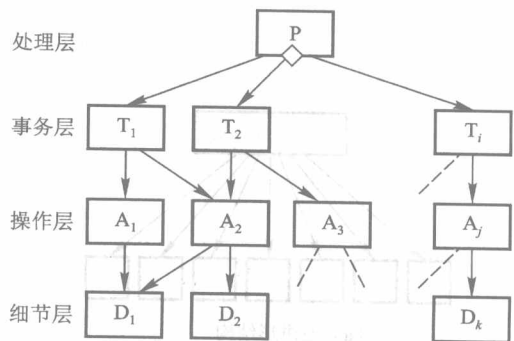


图 3.31 动作分支的典型结构

3.3.5 优化初始 SC 图的指导规则

把初始 SC 图变成设计文档中的最终 SC 图，需要进一步细化和改进。本节将介绍 SD 方法中常用于优化软件初始 SC 图的两条指导规则。

1. 对模块划分的原则

一般来说，模块的总行数应控制在 10~100 行的范围内，最好为 30~60 行，能容纳在一张打印纸内。过长的模块往往是分解不充分的表现，会增加阅读理解的难度；但小模块太多也会使块间联系变得复杂，增大系统在模块调用时传递信息所花费的开销。

在改进 SC 图时，有些模块在图上的位置可能要上升、下降或左右移动，从而变更模块调用关系。模块位置应否变更，应视对计算机处理是否方便而定，不必拘泥于它与 DFD 图上对应的加工是否位置一致。

2. 高扇入/低扇出的原则

扇入 (fan-in) / 扇出 (fan-out) 是从电子学借用过来的词，在 SC 图中可用于显示模块的调用关系，如图 3.32 所示。扇入高则上级模块多，能够增加模块的利用率；扇出低则表示下级模块少，可以减少模块调用和控制的复杂度。通常扇出数以 3~4 为宜，最好不超过 5~7。如扇出过大，软件结构将呈煎饼形 (pancaking)，如图 3.33 (a) 所示，此时可用增加中间层的方法使扇出减小，如图 3.33 (b) 所示。



图 3.32 模块的扇入和扇出

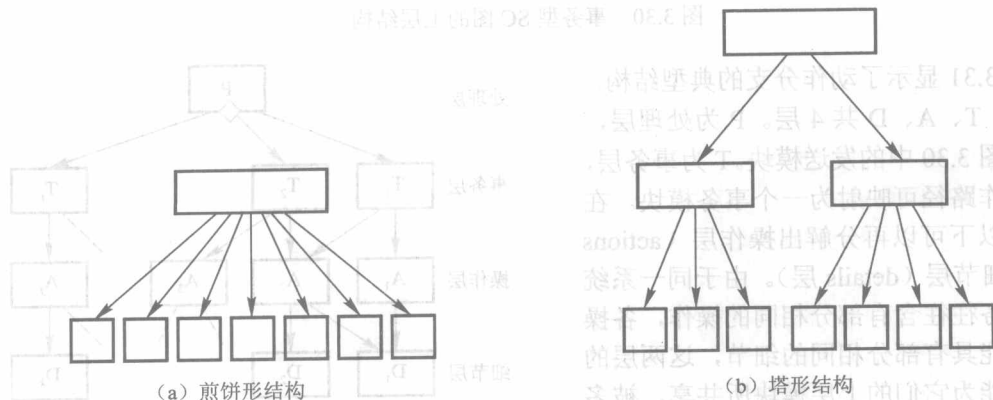


图 3.33 增加中间层可减少扇出

设计良好的软件通常具有瓮形 (oval-shaped) 结构, 两头小, 中间大, 如图 3.34 所示。这类软件在下部收拢, 表明它在低层模块中使用了较多高扇入的共享模块。煎饼形一般是不可取的, 因为它常常是高扇出的结果。

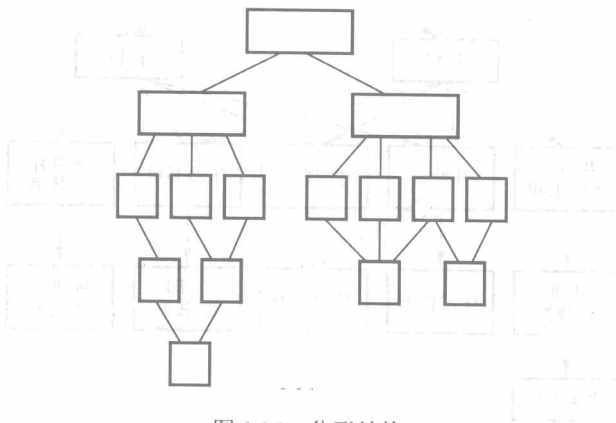


图 3.34 瓮形结构

3.3.6 教材购销系统的总体结构

[例 3.11] 在例 3.8 中, 已获得教材购销系统第三层的两张 DFD 图, 即图 3.16 的销售子系统 DFD 图和图 3.17 的采购子系统 DFD 图。试用 SD 方法从上述两张 DFD 图导出教材购销系统的总体结构, 包括初始的 SC 图和改进后的最终 SC 图。

[解] 为节省篇幅, 本例仅列出初始的 SC 图和最终 SC 图的结果, 步骤从略。

1. 初始 SC 图

包括上层框架、销售子系统和采购子系统, 详见图 3.35~3.37。

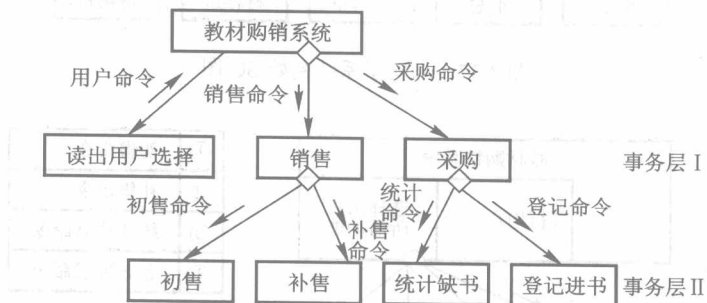


图 3.35 教材购销系统的上层框架

2. 最终 SC 图

改进后的上层框架如图 3.38 所示, 包括初售、补售、统计缺货和登记进书 4 个子系统。

作为示例，这里仅显示初售子系统的最终 SC 图（如图 3.39 所示），以示一斑。

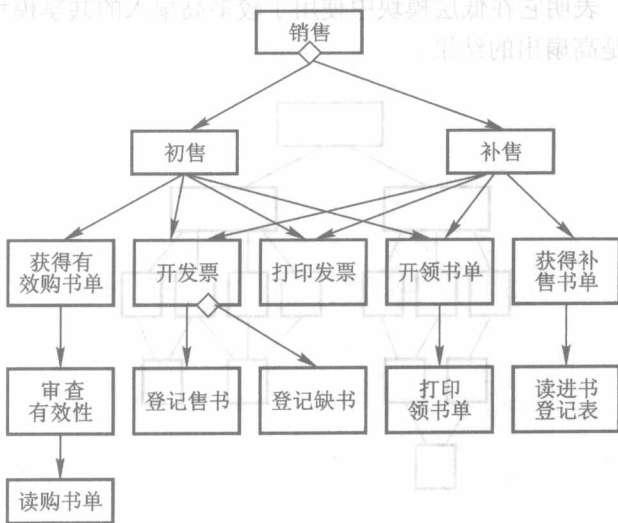


图 3.36 销售子系统初始 SC 图

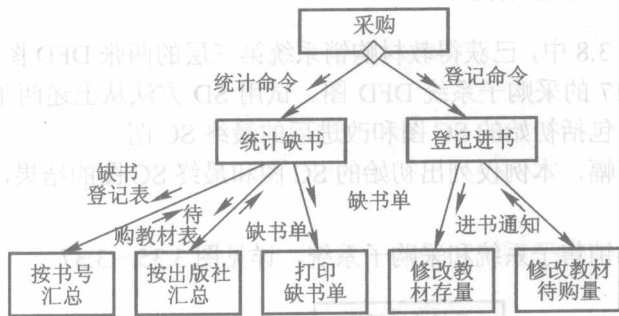


图 3.37 采购子系统初始 SC 图

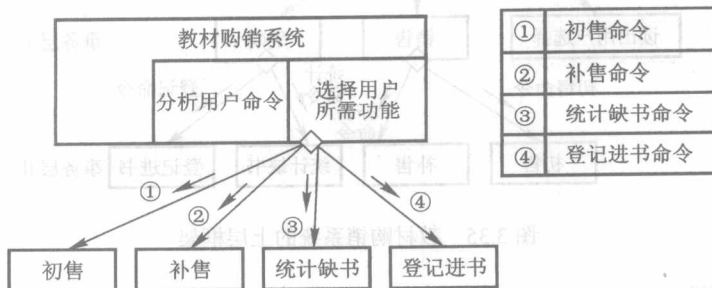


图 3.38 最终 SC 图的上层框架

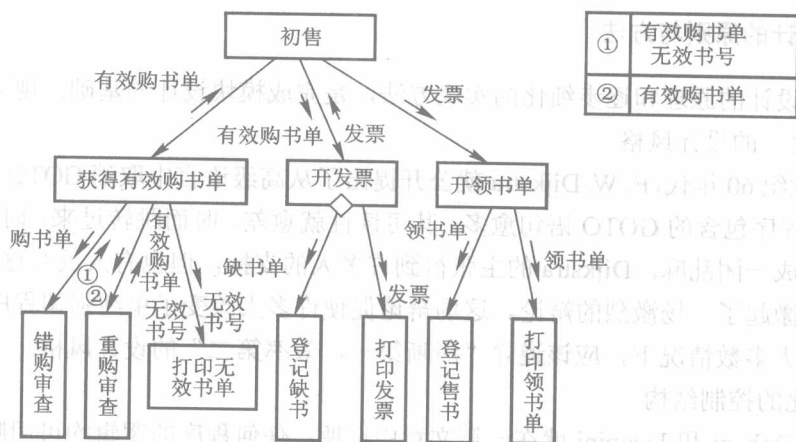


图 3.39 初售动作分支的最终 SC 图

3.4 模块设计

把 DFD 图转换为最终 SC 图，仅仅完成了软件设计的第一步。传统的软件工程将软件设计分成两步走：总体（或结构）设计——用最终 SC 图表示；模块设计——用逐步细化的方法来实现。模块设计用于对系统中的每个模块给出足够详细的逻辑描述，故亦称详细设计。这些描述可用规范化的表达工具来表示，但它们还不是程序，一般不能够在计算机上运行。

本节除说明模块设计的目的、任务与表达工具外，还要介绍如何用结构化程序设计的基本原理来指导模块内部的逻辑设计。

3.4.1 目的与任务

详细设计的目的，是为 SC 图中的每个模块确定采用的算法和块内数据结构，用选定的表达工具给出清晰的描述。表达工具可由开发单位或设计人员自由选择，但它必须具有描述过程细节的能力，而且能在编码阶段直接将它翻译为用程序设计语言书写的源程序。

这一阶段的主要任务，是编写软件的模块设计说明书。为此，设计人员应：

① 为每个模块确定采用的算法。选择某种适当的工具表达算法的过程，写出模块的详细过程性描述。

② 确定每一模块使用的数据结构。

③ 确定模块接口的细节，包括对系统外部的接口和用户界面，对系统内部其他模块的接口，以及关于模块输入数据、输出数据及局部数据的全部细节。

3.4.2 模块设计的原则与方法

结构程序设计的原理和逐步细化的实现方法，是完成模块设计的基础。现分述如下。

1. 清晰第一的设计风格

早在 20 世纪 60 年代, E. W. Dijkstra 就公开提出了从高级语言中取消 GOTO 语句的主张。他认为, 一个程序包含的 GOTO 语句愈多, 其可读性就愈差。时而跳转过来, 时而跳转过去, 可能把程序搞成一团乱麻。Dijkstra 的主张得到许多人的支持, 但也有人反对这种“一刀切”的做法, 因而激起了一场激烈的辩论。这场辩论促使许多人改变了单纯强调程序效率的旧观念, 认识到在大多数情况下, 应该遵守“清晰第一, 效率第二”的设计风格。

2. 结构化的控制结构

1966 年, Bohem 和 Jacopini 就在一篇文章中证明: 任何程序的逻辑均可用顺序、选择和循环 (DO-WHILE 型) 3 种控制结构或它们的组合来实现, 从而在理论上为结构程序设计奠定了基础。1968 年, Dijkstra 建议仅用这 3 种控制结构来构成程序。1972 年, Mills 进一步提出每个控制结构只应有一个入口和一个出口的原则。综上所述, 如果所有的模块在详细设计中都只使用单入口、单出口的 3 种基本控制结构, 如图 3.40 所示, 则不论一个程序包含多少个模块, 整个程序将仍能保持一条清晰的线索。这就是常说的控制结构的结构化, 它是详细设计阶段确保模块逻辑清晰的关键技术。

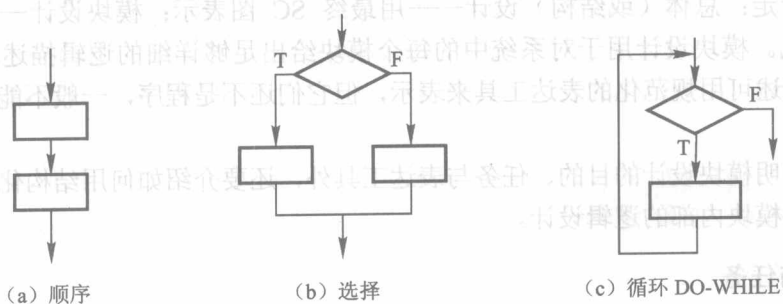


图 3.40 3 种基本控制结构的流程图

这里还有几点要补充说明:

① 为了方便使用或者提高程序效率, 大多数软件开发项目还允许在详细设计中补充使用 DO-UNTIL 和 DO-CASE 两种控制结构。

② 在许多情况下, 当程序执行到满足某种条件时, 需要立即从循环中转移出来。如果死抠单出口的原则, 就会不必要地使循环重复下去, 延长程序的执行时间。为了解决这类问题, 在 PDL 语言 (详见第 3.4.3 节) 中允许用 UNDO 语句提前退出循环, 如图 3.41 所示。

3. 逐步细化的实现方法

把给定的模块功能转换成它的详细逻辑描述,通常都采用逐步细化的策略。实践表明,逐步细化尤其适合于详细设计。由此产生的程序逻辑一般错误较少,可靠性也比较高。

[例 3.12] 在一组数中找出其中的最大数。

[解] 首先,把问题的解答描述为:

第一步:

- 1 输入一组数。
- 2 找出其中的最大数。
- 3 输出最大数。

以上 3 条中,第 1、3 两条都比较简单,所以下一步主要细化第 2 条。

第二步:

- 2.1 任取一数,假设它就是最大数。
- 2.2 将该数与其余各数逐一比较。
- 2.3 若发现有任何数大于该一假设的最大数,即取而代之。

以上 3 条是第一步中第 2 条分解的结果。

把第一步中的第 1 条具体化,同时对 2.1~2.3 继续进行相应的细化,可得到:

- 1' 输入一个数组。
- 2.1' 令最大数=数组中的第一元素。
- 2.2' 从第二元素至最末一个元素依次做。
- 2.3' 如果新元素>最大数,
则 最大数=新元素。
- 3' 输出最大数。

通过这一实例,可以把逐步细化的设计步骤归结为:

- ① 由粗到细地对程序进行逐步的细化。每一步可选择其中的一条至数条,将它(们)分解成更多或更详细的程序步骤。
- ② 在细化程序的过程时,同时对数据的描述进行细化。换句话说,过程和数据结构的细化要并行地进行,在适当的时候交叉穿插。
- ③ 每一步细化均使用相同的结构化语言,最后一步一般直接用伪代码来描述,以便编码时直接翻译为源程序。

逐步细化设计受到许多人的赞同,并不是偶然的。它的主要优点是:

- ① 每一步只优先处理当前最需要细化的部分(如上例第一步中的找出最大数),其余部分则推迟到适当的时机再考虑。先后有序,主次分明,可避免全面开花,顾此失彼。
- ② 易于验证程序正确性,比形式化的程序正确性证明更易被非专业人员接受,因而也

```
DO WHILE C1
```

```
...
IF C2 UNDO...
```

```
...
ENDDO
```

图 3.41 有两个出口 DO-WHILE 的结构

更加实用。传统的程序正确性验证采用公理化的证明规则，方法十分繁琐。一个不长的程序，常常需要一长串的验证。用逐步细化方法设计的程序，由于相邻步之间变化甚小，不难验证它们的内容是否等效。所以这一方法的实质，是要求在每一步细化中确保实现前一步的要求，不要等程序写完后再来验证。

用“结构化”保证程序的清晰、易读，用“逐步细化”实现程序的正确、可靠，由此很自然地得出如下结论：模块的逻辑设计必须用结构程序设计的原理来指导。

3.4.3 常用的表达工具

本节介绍详细设计中经常使用的几种设计表达工具。

1. 流程图和 N-S 图

流程图 (flow diagram) 是最古老的设计表达工具之一，至今仍常常使用。但随着结构化程序设计的普及，流程图在描述程序逻辑时的随意性与灵活性，恰恰变成了它的缺点。1973 年，Nassi 和 Shneiderman 发表了题为“结构化程序的流程图技术”的文章，提出用方框图来代替传统流程图的思路和方法。根据这两位创始人的姓氏，人们把它简称为 N-S 图。图 3.42 显示了采用 N-S 图和流程图来表达同一段程序的情况。

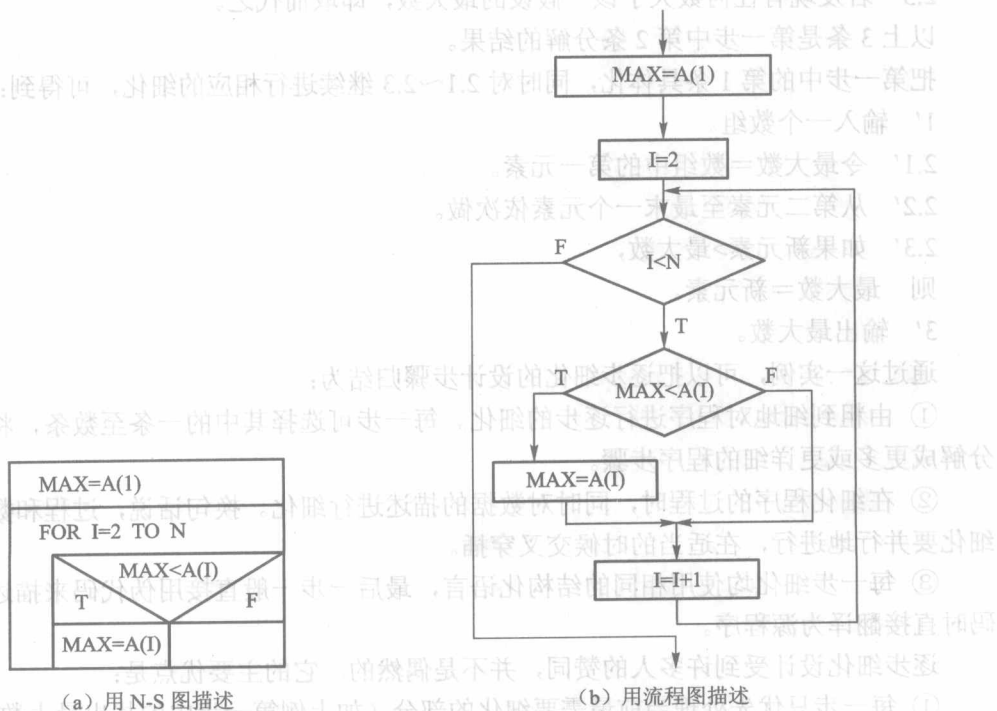


图 3.42 相同的程序用 N-S 图和流程图来表达的比较

2. 伪代码和 PDL 语言

伪代码 (pseudo code) 属于文字形式的表达工具。它形式上与代码相似, 但并非真正的代码, 也不能在计算机上执行。1975 年, Caine 与 Gordon 在“PDL: 一种软件设计工具”一文中, 报导了他们所设计的一种伪代码, 命名为 PDL (program design language)。

图 3.43、图 3.44 分别显示了用 PDL 描述选择结构和循环结构的方法。其中的循环结构又可区分为 DO-WHILE、DO-UNTIL 和 DO-FOR 等 3 类情况, 如图 3.45 所示。

```

IF <条件>
    一或数条语句
ELSEIF <条件>
    一或数条语句
    :
ELSEIF <条件>
    一或数条语句
ELSE
    一或数条语句
ENDIF
  
```

图 3.43 PDL 描述的 IF 结构

```

DO 判断重复的条件
    一或数条语句
ENDDO
  
```

图 3.44 PDL 描述的 DO 结构

```

DO WHILE THERE ARE INPUT RECORDS
DO UNTIL "END" STATEMENT HAS BEEN PROCESSED
DO FOR EACH ITEM IN THE LIST EXCEPT THE LAST ONE
  
```

图 3.45 几种不同的 DO 结构

PDL 语言还设有 DO-CASE 语句, 用以描述多分支的选择结构。

为了方便对比, 我们把图 3.42 (b) 所示的流程图用 PDL 来描述, 如图 3.46 所示, 或进一步简化为图 3.47 的样式。

还需指出, 上述 4 种工具除 N-S 图以外, 其余 3 种工具也可在总体设计阶段用于表达软件的结构。实际上, 除了 SC 图主要用作总体设计的表达工具外, 其他设计工具的描述大都可粗可细, 其范围也可大可小, 并不限于仅在某个设计阶段使用。

```

Enter a vector
Set Maximum to the value of
the first element in the vector

DO for each element
from the second one to the last
IF value of the element is greater than
the Maximum value
Set Maximum to value of the element
ENDIF
ENDDO
Print the Maximum value

```

图 3.46 用 PDL 描述的图 3.41 的程序逻辑

```

Input array A
MAX = A(1)
DO for I = 2 to N
IF MAX < A(I)
Set MAX = A(I)
ENDIF
ENDDO
Print MAX

```

图 3.47 图 3.45 的简化程序逻辑

小 结

本章讨论传统软件工程的系统开发技术，重点放在基于瀑布模型的结构化分析与设计和模块设计上，但不涉及同为传统软件工程的快速原型开发等内容。全章以实例（从“教材销售”到“教材购销”）为主线，依次展示了结构化分析、结构化设计和模块设计的常用技术。

1. 用 SA 方法建立分析模型

抽象与分解，是结构化分析的指导思想。通过由顶向下逐步细化得出的一组分层 DFD 图，是在不同的抽象级别上对系统所作的描述。逐层展开可以使问题的复杂性变得容易控制，使分析员不致突然面临一大堆细节。

2. 将分析模型映射为设计模型

建模与映射都是软件工程最常使用的技术。需求分析需建立分析模型，软件设计需建立软件模型，二者之间可通过映射实现转换。

本章例 3.1 展示了为“教材销售系统”建立分析模型的步骤。从该例可见，在面向数据流的软件系统开发中，DFD 图和加工规格说明是建立分析模型最常用的两种描述工具。通过画分层 DFD 图，可以获得待开发系统的一组分析模型；通过对加工规格说明的细化，可以获得软件的需求规格说明书（SRS）。在设计阶段，设计模型通常用 SC 图作为描述工具。SD 方法的主要任务，就是提供一组映射规则，把主要由 DFD 图和 SRS 组成的分析模型转换成由初始 SC 图描述的系统设计模型；然后通过对结构设计的优化，形成由最终 SC 图表示的系统设计模型。以下是从分层 DFD 图到最终 SC 图的一般过程：

结构化分析（工具：DFD、PSPEC）——→ 分析模型（分层 DFD 图）+ SRS

结构化设计（工具：SC图） $\xrightarrow{\text{映射}}$ 初始设计模型（初始SC图）

初始设计模型（初始SC图） $\xrightarrow{\text{优化}}$ 最终设计模型（最终SC图）

3. 模块设计方法

模块设计是传统软件工程的重要组成部分，其性质属于详细设计，大致上相当于第二代软件工程的面向对象设计（OOD）。瀑布模型将软件设计分成总体设计和详细设计两大步：把DFD图转换为最终SC图，以及为SC图中的每个模块确定采用的算法和数据结构。从例3.12可以看出：

① 模块设计要用结构化程序设计的基本原理来指导，具体地说，就是要用“结构化”的思想保证程序的清晰易读，用“逐步细化”实现程序的正确、可靠。

② 算法和数据结构应该在逐步细化过程中并行地细化。

③ 模块设计的任务，是从SC图导出模块的过程性描述。这些描述可以采用图形的或者文字的表达工具。但是不论采用哪一种工具，都必须具有描述过程细节的能力，且能在编码阶段直接将它翻译为用程序设计语言书写的源程序。

正如本章开头所指出，传统软件工程技术目前仅在某些特定类型的软件开发中使用，但它们所蕴涵的思想与方法，仍经常在第二、三代软件工程的软件开发中反映出来。本章将这些技术集中介绍，目的是方便读者学习，并进而了解3代软件工程之间既不断发展、又一脉相承的关系。

习 题

1. 需求分析的任务是什么？怎样理解分析阶段的任务是决定“做什么”，而不是“怎样做”？
2. 需求分析要经过哪些步骤？
3. 什么是结构化分析？它的“结构化”体现在哪里？
4. 需求规格说明书由哪些部分组成？各部分的主要内容是什么？
5. DFD和CFD有什么区别？
6. 为什么DFD要分层？画分层DFD要遵循哪些原则？
7. 选择一个系统（例如工资管理系统、飞机订票系统、图书馆管理系统等），用SA方法对它进行分析，并给出分析模型。
8. 简释SC图的作用。
9. 简释事务型结构。
10. 简释变换型结构。
11. 为什么事务型软件的结构常常具有中间大、两头小的形状？
12. 某事务系统具有下列功能：
 - (1) 读入用户命令，并检查其有效性。

- (2) 按照命令的编号(1~4号)进行分类处理。
- (3) 1号命令计算产品工时, 能根据用户给出的各种产品数量, 计算出各工种的需要工时和缺额工时。
- (4) 2号命令计算材料消耗, 根据产品的材料定额和用户给出的生产数量, 计算各种材料的需求量。
- (5) 3号命令编制材料订货计划。
- (6) 4号命令计算产品成本。

试用结构化分析和设计方法画出该系统的 DFD 图并据此导出系统的 SC 图。

- 13. 简述模块详细说明书的主要内容。
- 14. 简单比较本章讲解的几种模块设计表达工具的优缺点。
- 15. 选一种排序(从小到大)算法, 分别用流程图、N-S 图和 PDL 语言描述其详细过程。
- 16. 试从指导原则、出发点、最终目标与适用范围等方面, 比较面向数据流和面向数据结构两类设计方法的异同。

思考题

- 1. 需求分析的任务是什么? 怎样理解分析问题和解决问题的方法? 什么是分析的层次和范围?
- 2. 需求分析为什么要分层次和范围?
- 3. 什么是结构化分析? 它的“结构化”体现在哪里?
- 4. 需求规格说明由哪些部分组成? 各部分的主要内容是什么?
- 5. DFD 和 CED 有什么区别?
- 6. 为什么 DFD 要分层? 画分层 DFD 要遵循哪些原则?
- 7. 选择一个系统(例如加工管理信息系统), 画出顶层 DFD, 并说明其设计思路。
- 8. 简述 SC 图的作用。
- 9. 简述数据流图的作用。
- 10. 简述数据流图的作用。
- 11. 什么是数据流图? 数据流图在软件开发中的作用是什么? 数据流图由哪些部分组成?
- 12. 数据流图的基本符号有哪些? 它们的作用是什么?
- 13. 数据流图的基本符号有哪些? 它们的作用是什么?

中篇

面向对象软件工程

- 第 4 章 面向对象与 UML
- 第 5 章 需求工程与需求分析
- 第 6 章 面向对象分析
- 第 7 章 面向对象设计
- 第 8 章 编码与测试

第4章 面向对象与UML

面向对象是以问题空间中出现的物体为中心进行模型化的一种技术。建立模型是软件工程中最常使用的技术之一。无论软件分析或软件设计，都需要建立模型。UML就是OO软件工程使用的统一建模语言。它是一种图形化的语言，主要用图形方式来表示。鉴于它在OO开发中的重要地位，本章将在扼要介绍面向对象思想的基础上，介绍UML及其用法。

4.1 面向对象概述

面向对象技术通过抽象化现实世界中的物体，来描述一个系统。本书2.3.3节已经简单说明过面向对象的基本概念，本节将对面向对象的术语和相关原则进一步展开介绍。

4.1.1 对象和类

对象的概念是面向对象技术的核心。对象可代表客观世界中实际或抽象的事物，例如一个人、物品、事件、概念或者报表，每个对象都包含一定的特征和服务功能。

可以从以下两个方面去理解对象的概念：一方面，客观世界是由各种对象组成的，对象可以分解，复杂对象可以由比较简单的对象组合构成；另一方面，在计算机世界中，对象可定义为数据以及在其上的操作的封装体。它是客观世界在计算机中的逻辑表示，也就是说，对象是客观世界的实体或概念在计算机中的表示。一个对象是具有唯一对象名和固定对外接口的一组属性和操作的集合，用来模拟组成或影响现实世界的一个或一组因素。其中对象的属性表示该对象的静态特征（状态信息），方法或操作则描述对象的动态行为。对外接口是对象在约定好的运行框架和消息传递机制中与外界通信的通道。一个对象的私有成分都被“封装”在对象内部，外部不能访问。对象之间可通过消息的传递相互作用。

类是一组相似的对象共性抽象，是创建对象的有效模板。在现实世界里，一个对象通常有一些与之相似的其他对象。例如，学生之间有着相同的特征（他们做相同的事情，他们以同样一种方式被描述），课程之间也有着相同的特征。因此，我们需要将同一类相似对象的共性抽取出来统一表示，这就需要类的概念。

同样，也可以从两个角度去理解类的含义：第一，在现实世界中，类是一组客观对象的抽象。第二，在计算机世界中，类是一种提供具有特定功能的模块和一种代码共享的手段或

工具，即类是实现抽象数据类型的工具。

类与对象的关系，可看成是抽象与具体的关系。组成类的每个对象都是该类的实例；实例是类的具体事物，类是各个实例的综合抽象。通过类还可以生成许多同类型的对象。例如，桥梁是一个类，它是抽象的概念；南浦大桥、长江大桥、芦沟桥等都是桥梁的某一具体化的实例，是一个个具体的对象，这些实例继承了桥梁的公共特征。

4.1.2 面向对象的基本特征

面向对象的一个重要思想，就是模拟人的思维方式来进行软件开发，将问题空间的概念直接映射到解空间。抽象、封装、继承和多态，构成了OO的基本特征。

1. 抽象

抽象（abstraction）常用于在某个重要的或想关注的侧面来表示某个物体或概念。它忽略主题中与当前目标无关的因素，以便更充分地注意与当前目标相关的因素。

世界是复杂的，抽象是处理复杂性的好方法。例如对于人的抽象，从受教育的视角来看，需要知道人的姓名、住址、电话号码、社会保险号以及教育背景；从警察的视角来看，需要知道人的姓名、住址、电话号码、体重、身高、国籍、民族、头发颜色、眼睛颜色等。所关注的是同一个人，只是抽象的角度不同而已。

抽象是一个分析过程，应根据需要考虑应用程序感兴趣的功能、属性、方法，将其他因素忽略。

2. 封装

封装（encapsulation）可用于把操作和数据包围起来，对数据的访问只通过已定义的接口来完成。面向对象计算始于这个基本概念，即现实世界可以描绘成一系列完全自治、封装的对象，这些对象通过一个受保护的接口来访问其他对象。

封装特性使所构建的系统在改变它的实现时，只要接口不发生变化，就不会影响系统中的其他部分。例如，当银行使用计算机记录顾客的账户信息时，使用什么数据库、什么操作系统，顾客都不用关心；因为这些信息被封装起来了，顾客只需与出纳完成交易即可。通过隐藏实现账户交易的细节，银行能在任何时间自由改变其功能的具体实现方法，而不会影响提供给顾客的账户服务。

为了让应用程序更易于维护，需要限制对数据属性和方法的访问。当一个对象需要获取另一个对象的信息时，必须向此对象发送相应的消息，而不是直接访问对象的数据。事实上，现实世界中也可按这种方式工作。例如想了解一个人的年龄，不必直接询问这个人，看他的身份证就可以了。

3. 继承

继承（inheritance）常用于类的层次模型，它提供了一种表述共性的方法。定义一个新类，

可以从现有的类中派生出来，这称为类继承。新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。子类可以从它的父类继承方法和属性，并且用修改或增加新的属性和方法使之更适合特殊的需要。继承可在类之间建立“is a”或“is like”关系，从而使已经存在的数据和代码变得很容易复用。例如，学生和教授都有姓名、住址和电话号码等信息，此外，学生属于某一班级，教授属于某一教研室，据此就可以设计有关学生和教授的类。但更好的方法是使用继承，即首先设计一个“人”类，它包含姓名、住址和电话号码等公共属性，然后在此基础上设计“学生”类和“教授”类，添加各自所需要的特殊的属性。

4. 多态

不同类的对象可以对同一消息作出响应，执行不同的处理，这称为多态（polymorphism）。

例如，对象“Zhang”可能是一个学生、一名注册员或一位教授。如果不必为 Zhang 所属的不同类型而提供不同的处理，就会大大减少开发付出的努力。又例如，大学有雇用新成员的标准过程：一旦被雇用，雇员将被加进大学的退休金计划里，并为其创建一个雇员卡，如图 4.1 所示。大学里雇用教授也遵照同样的流程，除此之外还要给他加一个停车位。

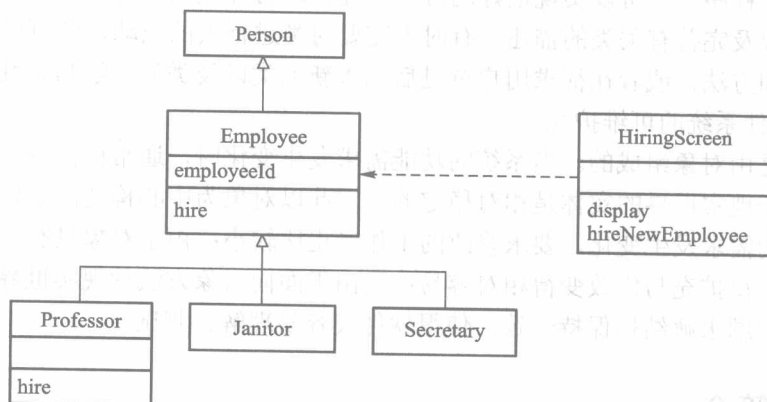


图 4.1 大学管理系统中的类图

如果 hire（雇用）方法在 Employee 类中已经实现，Employee 类会实现将这个成员加入大学退休金计划以及打印雇员卡的方法。在 Professor 类中，hire 方法被覆盖了。它应调用 Employee 类中的 hire 方法，因为那个行为同样也适用于教授，但在 Professor 类中要加上预留停车位的功能。

如果把消息 hire 发送给任何类型的雇员，这样 HiringScreen 对象中 hireNewEmployee 方法就不再需要嵌套复杂的 IF 或 CASE 语句。这种方法不需要把消息 hireProfessor 发送给 Professor 对象，也不需要把消息 hireJanitor 发送给 Janitor 对象等。仅发送消息 hire 给所有类型的雇员，各对象就能完成正确的事情。这样一来，可以加入新类型的雇员（如注册员），而根本不需要改变 HiringScreen 对象。换言之，这个类被松散耦合进 Employee 类层次，这使得可以很容易

地扩展系统。

4.1.3 面向对象开发的优点

面向对象符合人类习惯的思维方式。它使用对象模拟现实世界中的概念，而不是强调算法，开发过程始终围绕建立对象模型进行。这种开发方法主要具有如下的优点。

1. 提高软件系统的可复用性

可复用性是面向对象软件开发的核心思路。面向对象的四大特征——抽象、封装、继承和多态，都或多或少地围绕着这个核心。对象的封装性较好地实现了模块独立和信息隐藏，使得在构造新的软件时可以方便地复用已有的对象，极大地提高了软件开发的效率。

软件复用的途径主要包括：创建类的实例对象；从已有类派生新的子类。派生类既可以继承其父类的属性、方法，也可以添加新的属性和方法。

2. 提高软件系统的可扩展性

可扩展性是对现代应用软件提出的又一个重要要求，要求它们能方便、容易地被扩充和修改。面向对象程序设计可以实现很好的可扩展性，因为开发人员可以根据对用户需求的理解，不断地修改及完善有关类的描述。有时不需要对类进行大的改动，即可以利用继承对新的类添加属性和方法；或者在征求用户意见后加入新的类以及类的方法与属性等。

3. 提高软件系统的可维护性

一个系统是由对象组成的。当系统的功能需求发生变化时，通常仅需修改与之相关的对象或者类。由于现实世界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比较稳定，即使软件的需求发生变化，要求修改的工作量也比较小；由于对象具有良好的模块独立性和继承性，使得扩充与修改变得相对容易；又由于面向对象方法使现实世界的问题结构与计算机世界的问题求解结构保持一致，使得软件较容易理解、测试。

4.2 UML 简介

UML 代表了 OO 软件开发技术的发展方向，一经推出就获得了工业界和科技界的广泛支持。反过来，它在软件建模上的应用又推动了 OO 软件开发技术的发展。

1994 年 10 月，Grady Booch 和 Jim Rumbaugh 首先将 Booch 93 和 OMT-2 统一起来，并于 1995 年 10 月发布了称之为统一方法（unified method）的 UM 0.8。1995 年秋，OOSE 的创始人 Ivar Jacobson 加盟到这一工作之中。经过三人的共同努力，于 1996 年 6 月和 10 月先后发布了 UML 0.9 和 UML 0.91 两个新的版本，并将 UM 重新命名为 UML（unified modeling language，统一建模语言）。UML 的开发得到了公众的积极响应，一批公司参与了 UML 的定义工作，其中有 IBM、Microsoft、HP 和 Oracle 等。1997 年 1 月正式公布了 UML 1.0，1997 年 11 月 17 日，OMG（Object Management Group）接纳 UML 1.1 作为基于面向对象技术的标准建模语言。2001 年，UML 1.4 修订完毕，2004 年 1 月，UML 1.4.2 被国际标准化组织吸

收为国际标准，编号 ISO/IEC 19501。从 2005 年 7 月起，UML 2.x 的规范开始问世，可以表示更为精确的语义，本书主要介绍目前已比较成熟的 UML 1.x 版本。

4.2.1 UML 的组成

UML 是一种基于面向对象的可视化建模语言。它提供了丰富的以图形符号表示的模型元素，这些标准的图形符号隐含了 UML 的语法，而由这些图形符号组成的各种模型则给出了 UML 的语义。UML 简单、一致、通用的定义，使开发者能在语义上取得一致，消除了因人而异的表达方法所造成的影响。

1. UML 的模型元素

UML 定义了两类模型元素。一类模型元素用于表示模型中的某个概念，如类、对象、构件、用例、结点 (node)、接口 (interface)、包 (package) 和注释 (note) 等；另一类用于表示模型元素之间相互连接的关系，其中主要有关联、泛化、依赖、实现、聚集和组合等。这两类模型元素均可用图形符号来表示，图 4.2 给出了部分模型元素的图形符号。

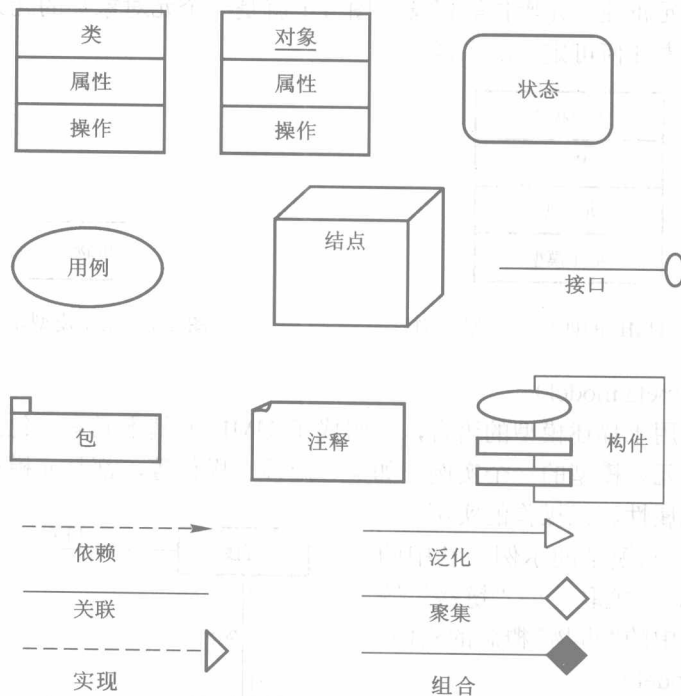


图 4.2 模型元素的图形表示

以下是几种主要连接关系的含义：

- ① 关联 (association)：模型元素实例之间的固定对应关系，为永久性的结构关系。

② 泛化 (generalization)：表示一般与特殊关系，“一般”元素是“特殊”元素的泛化，“特殊”元素是“一般”元素的特化 (specialization)。

③ 依赖 (dependency)：表示一个元素以某种方式依赖于另一个元素，为短暂性关系。

④ 实现 (realization)：表示接口和实现它的模型元素之间的关系。

⑤ 聚集 (aggregation)：表示“整体”与“部分”关系，“部分”元素是“整体”元素的一部分。

⑥ 组合 (composition)：表示强烈的“整体”与“部分”关系，“部分”不能独立于“整体”而存在。

2. UML 的元模型结构

按照 UML 的语义，UML 模型可定义为 4 个抽象层次。如图 4.3 所示，从低到高分别是元元模型、元模型、模型和用户模型。下一层是上一层的基础，上一层是下一层的实例。

(1) 元元模型 (meta-meta model)

元元模型定义了用于描述元模型的语言，它是任何模型的基础。在 UML 的元元模型中，定义了元对象类、元属性、元操作等概念。图 4.4 就是一个元对象类的元元模型，其中的概念“事物”可以代表任何可定义的东西。

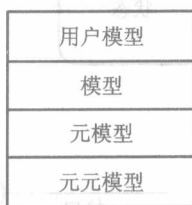


图 4.3 UML 的四层元模型结构

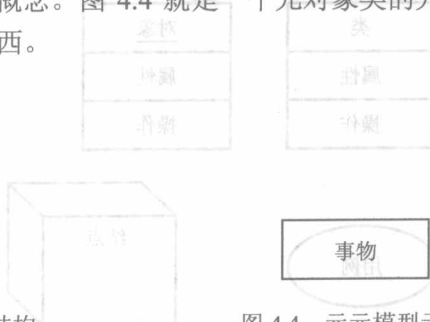


图 4.4 元元模型示例

(2) 元模型 (meta model)

元模型定义了用于描述模型的语言，它组成了 UML 的基本元素，包括面向对象和构件的概念。元模型是元元模型的一个实例，如类、属性、操作等，都是元模型层的元对象，它们分别是元类、元属性、元操作的实例。

图 4.5 是一个元模型的示例，其中的“类”、“对象”、“关联”、“链接”等概念都是元元模型中的“事物”概念的实例。

(3) 模型 (model)

模型定义了用于描述信息领域的语言，它组成了 UML 的模型。模型是对现实世界的抽象，无论是问题领域还是解决方案，都可以抽象成模型。图 4.6 给出了一个人事管理系统的部分模型。在对象类“员工”与“部门”



图 4.5 元模型示例

之间存在着“成员”关联，符号“1”和“*”表示一个具体的员工可以是一个部门的成员，而一个部门可以有多个员工作为其成员；“部门”与“公司”之间也存在着“成员”关联，一个部门只能是一个公司的成员，而一个公司可以有多个部门作为其成员。

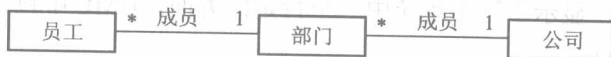


图 4.6 模型示例

(4) 用户模型 (user model)

用户模型是模型的实例，它用于表达一个模型的特定情况。例如图 4.7 表示的是图 4.6 的模型的一个实例。其中，“王平”和“李明”是“销售部”的成员，“王平”对象和“李明”对象分别与“销售部”对象有“成员”链接；“刘欣”和“张丰”是“采购部”的成员，分别与“采购部”对象有“成员”链接。一个人只能属于一个部门，一个部门可以有多个成员（本例的每个部门有两个成员）。

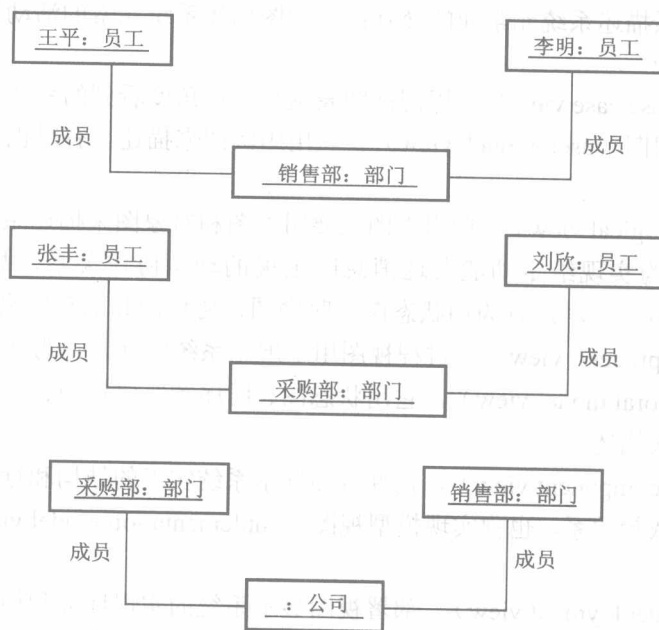


图 4.7 用户模型示例

用户模型层中的模型元素是模型层中的模型元素的实例，如图 4.7 中的“王平”、“李明”、“刘欣”、“张丰”是“员工”对象类的实例对象，“销售部”和“采购部”是“部门”对象类的实例对象，UML 中用名称下面带有下划线表示实例；“成员”关联是元关联“关联”的实例。

3. 图和视图

UML 是用来描述模型的，它用模型来描述系统的结构或静态特征以及行为或动态特征。它从不同的视角为系统建模，形成不同的视图，每个视图由一组图构成，图中包含了强调系统中某一方面的信息，显示了这个系统中一个特定的方面。UML 中包括两类图和 5 种视图。

(1) 图 (diagram)

图是系统构架在某个侧面的表示，UML 1.4 提供了两大类图——静态图和动态图，共计 9 种不同的图。

① 静态图 (static diagram)。静态图共有 5 种，包括用例图、类图、对象图、构件图和部署图。其中用例图描述系统功能；类图描述系统的静态结构；对象图描述系统在某个时刻的静态结构；构件图描述实现系统的元素的组织；部署图描述系统环境元素的配置，亦可称为配置图。

② 动态图 (dynamic diagram)。动态图共有 4 种，包括状态图、时序图、协作图和活动图。其中状态图描述系统元素的状态条件和响应；时序图按时间顺序描述系统元素间的交互；协作图按照连接关系描述系统元素间的交互；活动图描述系统元素的活动流程。

(2) 视图 (view)

① 用例视图 (use case view)。用例视图表达从用户角度看到的系统应有的外部功能，有时也称用户模型视图 (user model view)。它用用例图来描述，有时也用活动图来进一步描述其中的用例。

② 逻辑视图 (logical view)。逻辑视图主要用类图和对象图来描述系统的静态结构，它同时也描述对象间为实现给定功能发送消息时出现的动态协作关系，故称结构模型视图 (structural model view)。动态行为用状态图、时序图、协作图和活动图来描述。

③ 进程视图 (process view)。进程视图用于展示系统的动态行为及其并发性，也称行为模型视图 (behavioral model view)。它用状态图、时序图、协作图、活动图、构件图和部署图 (即配置图) 来描述。

④ 构件视图 (component view)。构件视图展示系统实现的结构和行为特征，包括实现模块和它们之间的依赖关系，也称实现模型视图 (implementation model view)。构件视图用构件图来描述。

⑤ 部署视图 (deployment view)。部署视图显示系统的实现环境和构件被部署到物理结构中的映射，如计算机、设备以及它们相互间的连接，哪个程序或对象在哪台计算机上执行等。部署视图用部署图来描述。

4.2.2 UML 的特点

UML 的出现是 OO 软件工程在近十多年来所取得的最重要的成果之一。其主要特点可以归结为以下 3 点：

1. 统一标准

UML 不仅统一了 Booch、OMT 和 OOSE 等方法中的基本概念，还吸取了面向对象技术领域其他流派的长处，其中也包括非 OO 方法的影响。UML 使用的表示符号考虑了各种方法的图形表示方法，删掉了大量易引起混乱的、多余的和极少使用的符号，也添加了一些新符号，提供了标准的面向对象的模型元素的定义和表示法，并已经成为 OMG 和 OSI 的标准。

2. 面向对象

UML 支持面向对象技术的主要概念，它提供了一批基本的表示模型元素的图形和方法，能简洁、明了地表达面向对象的各种概念和模型元素。

3. 表达能力强大、可视化

UML 是一种图形化语言，用 UML 的模型图形能清晰地表示系统的逻辑模型或实现模型。它不只是一堆图形符号，在每一个图形符号后面，都有良好定义的含义；UML 还提供了语言的扩展机制，用户可以根据需要自定义构造型、标记值和约束等，它的强大表达能力使它可以应用于各种复杂软件系统的建模。

4.2.3 UML 的应用

UML 适用于以面向对象技术来描述的任何系统，而且适用于系统开发的不同阶段，从需求规格描述直至系统完成后的测试和维护。其主要作用可以归结为以下 3 点：

- ① 通过对问题进行说明和可视化描述，帮助理解问题，并建立文档。
- ② 获取和交流有关应用问题求解的知识。
- ③ 对解决方案进行说明和可视化描述，辅助构建系统，并建立文档。

当采用面向对象技术开发系统时，第一步是描述需求；第二步是根据需求建立系统的静态模型，以构造系统的结构；第三步是描述系统的行为。其中在第一步与第二步中所建立的模型都是静态的，包括用例图、类图（包含包）、对象图、构件图和部署图等 5 种图形，是 UML 的静态建模机制。第三步中所建立的模型或者可以执行，或者表示执行时的时序状态或交互关系。它包括状态图、活动图、时序图和协作图等 4 种图形，是 UML 的动态建模机制。因此，UML 的主要内容也可以归纳为静态建模机制和动态建模机制两大类。4.3 节和 4.4 节将依次介绍这两类建模。

UML 模型还可作为测试阶段的依据。OO 系统通常也需要经过单元测试、集成测试、系统测试和验收测试。不同的测试小组使用不同的 UML 图作为测试依据：单元测试使用类图和类规格说明；集成测试使用构件图和协作图；系统测试按照用例图来验证系统的行为；验收测试由用户进行，以验证系统测试的结果是否满足在分析阶段确定的需求。

4.3 静态建模

UML 的静态建模机制包括用例图、类图和对象图。

4.3.1 用例图与用例模型

用例模型用于把应满足用户需求的基本功能聚合起来表示。对于待开发的新系统，用例描述系统应该做什么；对于已构造完毕的系统，用例应反映系统能完成什么样的功能。

用例模型由一组用例图组成，其基本组成部件是用例、参与者和系统。用例图可描述软件系统和外部参与者（actor）之间的交互。其中，用例代表从外部可见的系统的功能，可能包括完成某项任务的一系列逻辑相关的任务；参与者表示与系统交互的外部环境，可以是人、一个软件、一个硬件或其他与系统交互的实体。

1. 组成符号

如图 4.8 所示，用例图通常包含系统边界、用例、参与者和关联等组成符号。系统边界以一个矩形表示，上方标注系统名称，内部包含一个或多个用例；每个用例由一个椭圆形表示，其中标上用例的名称；参与者用一个人形的符号表示；参与者和用例之间或用例和用例之间的关联均用直线表示。

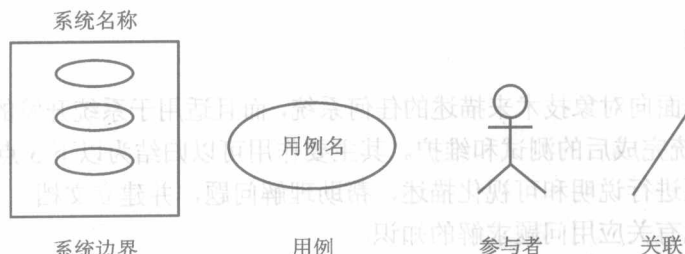


图 4.8 用例图的组成符号

2. 建立用例图

图 4.9 是某保险商务系统的用例图。在这一简化的保险系统中有 3 个用例：签订保险单、销售统计和客户统计。客户只与用例“签订保险单”有交互，保险销售员则与 3 个用例都有交互。

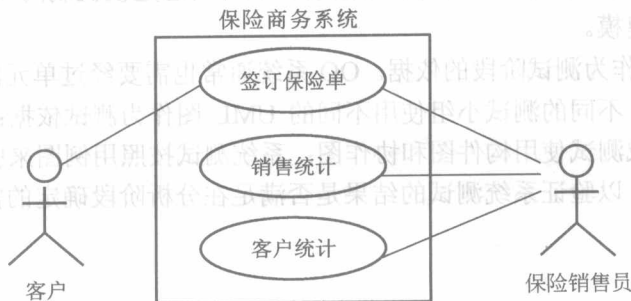


图 4.9 保险商务系统的用例图

3. 用例之间的关系

用例之间主要存在两种关系：扩展关系和包含关系。

(1) 扩展关系 (extend)

根据指定的条件，一个用例中有可能加入另一个用例的动作，这两个用例之间的关系就是扩展关系。例如，在执行“签汽车购买合同”用例时，根据汽车销售商的要求有可能需要执行“签保险单”用例的行为，当然，也有可能是另外一种情况，即在执行“签汽车购买合同”用例时不需要执行“签保险单”用例的行为。这样，从用例“签保险单”到用例“签汽车购买合同”存在扩展关系，如图 4.10 所示。

(2) 包含关系 (include)

当一个用例的行为包含另一个用例的行为时，这两个用例之间就构成了包含关系。如果若干个用例有某些行为是相同的，即可把这些相同的行为抽取出来单独成为一个用例，称为抽象用例。这样，当某个用例使用该抽象用例时，就包含了这个用例的所有行为。如图 4.11 所示，用例“签汽车保险单”和“签房屋保险单”的行为肯定都要包含用例“签保险单”的行为。这样，用例“签汽车保险单”和“签房屋保险单”到用例“签保险单”之间就存在包含关系。

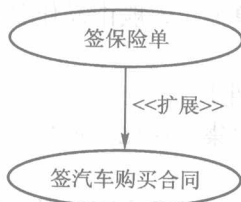


图 4.10 用例的扩展关系

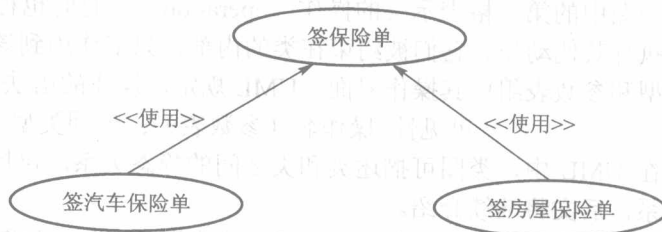


图 4.11 用例的包含关系

4.3.2 类图和对象图

在面向对象建模技术中，类和对象模型揭示了系统的静态结构。在 UML 中，类和对象模型分别由类图和对象图表示。

1. 类图和对象图

类描述同类对象的属性和行为。类图可表示类（包括类名、类的属性和操作）和类之间的关系，在 UML 中，类一般表示为一个划分成 3 格的矩形框（下面两格可省略），如图 4.12 所示。

在表示类的矩形框中，第一格指定类的名字。类的命名应尽量使用应用领域中的术语，明确且无歧义，以利于开发人员与用户之间的交流。第二格包含类的属性，用以描述该类对象的共同特点。UML 规定的类属性的语法为：

可见性 属性名: 类型 = 默认值 {约束特性}

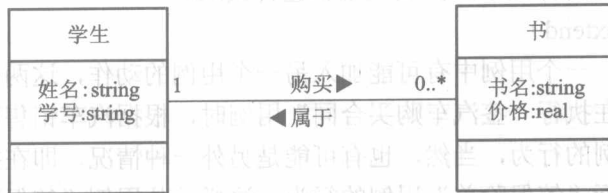


图 4.12 类图

不同属性具有不同的可见性。常用的可见性有 **Public**、**Private** 和 **Protected** 这 3 种，在 UML 中分别表示为 +、- 和 #。其中 **Public** 可见性表示所有的对象都可以访问；**Private** 可见性表示只有类本身的对象可以访问；**Protected** 可见性表示类本身及其子类的对象可以访问。类型表示该属性的数据类型，例如整数、实数、布尔型数等，也可以是用户自定义的类型，一般由所选择的程序设计语言确定。约束特性则是用户对该属性性质一个约束的说明，如“{只读}”表明该属性是只读属性。例如，属性年龄可以书写为：

-年龄: Integer = 20

类图中的第三格表示类的操作（operation），有时也称为方法，用于修改、查询类的属性或执行其他动作。它们被约束在类的内部，只能作用到该类的对象上。一般由操作名、返回类型和参数表组成其操作界面。UML 规定，操作的语法为：

可见性 操作名（参数表）: 返回类型 {约束特性}

在 UML 中，类图可描述类和类之间的静态关系，包括关联、聚集、泛化、依赖及组合等关系，后文将陆续介绍。

对象是类的实例。对象图可以看作是类图的实例，如图 4.13 所示；对象之间的链（link）是类之间的关联的实例。与类的图形表示相似，对象也可表示为划分成 3 格的矩形框（最下面的一格可以省略）。第一格是对象名：类名，并添加下划线；第二格记录属性值。链的图形表示与关联相似。对象图常用于表示复杂的类图的一个实例。

2. 关联关系

关联表示两个类之间存在的某种语义上的联系，即与该关联连接的类在对象之间的语义连接（称为链接）。例如，一个人为一家公司工作，一家公司有许多办公室，就可认为人和公司、公司和办公室之间存在某种语义上的联系。在分析/设计阶段的类图模型中，即可在“人”类和“公司”类、“公司”类和“办公室”类之间建立关联关系。根据链接的类/对象之间的具体情况，又可分为普通关联、递归关联、多重关联、有序关联、限制关联、或关联以及关联类。现逐一介绍如下。

(1) 普通关联

这是最常见的关联，可在两个类之间用一条直线连接，直线上写上关联名。关联可以有方向，表示该关联的使用方向。可以用线旁的小实心三角表示方向，也可以在关联上加上箭

头表示方向，在 UML 中也称为导航（navigability）。通常将只在一个方向上存在导航表示的关联称作单向关联，在两个方向上都存在导航表示的关联称作双向关联。图 4.12 表示了“学生”类和“书”类之间存在的双向关联，学生购买书，书属于学生。

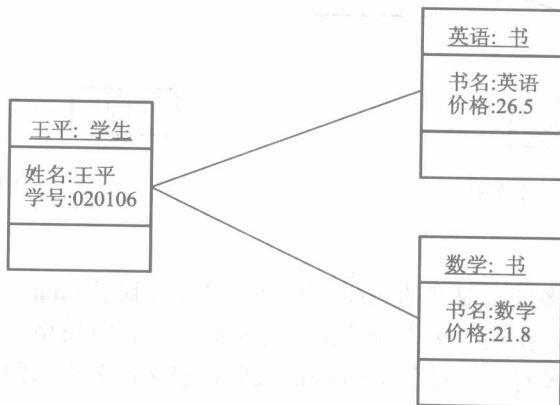


图 4.13 对象图

在关联的两端可写上一个被称为重数（multiplicity）的数值范围，表示该类有多少个对象可与对方的一个对象连接。重数的符号表示有：

- 1 表示 1 个对象，重数的默认值为 1。
- 0..1 表示 0 或 1。
- 1..* 表示 1 或多。
- 0..* 表示 0 或多，可以简化表示为*。
- 2..4 表示 2~4。

图 4.12 表示一个学生可以购买 0 或多本书，而一本书只能属于一个学生。

(2) 递归关联

UML 中允许一个类与自身关联，这称为递归关联。例如在一个公司中，一个老板管理多个工人，而工人和老板都是公司的员工，属于“员工”类。这样就形成了“员工”类到“员工”类的递归关联，图 4.14 表示了这种关联。关联的两端还标上了角色名，表示类在这个关联中所扮演的角色，图中的“老板”、“工人”就是员工充当的不同角色。



图 4.14 递归关联

(3) 多重关联

多重关联是指两个以上的类之间互相关联。例如，工人操作机器生产产品，可用图 4.15 表示为三重关联。图中省略了重数和角色名。

(4) 有序关联

在重数为“多”的关联端的对象可以是一个无序的集合，如果这些对象必须是有序的，

则可以在关联的“多”端写上“{ordered}”来指明。如图 4.16 所示，一个目录下可以有多个有序的文件，而一个文件只属于一个目录。

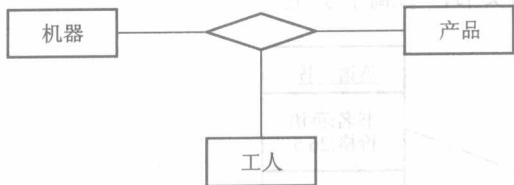


图 4.15 三重关联

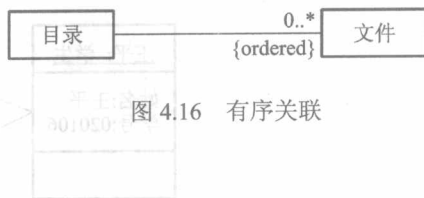


图 4.16 有序关联

(5) 限制关联

限制关联用于一对多或多对多的关联。用一个称为限制子(qualifier)的小方块来区分关联“多”端的对象集合，指明“多”端的某个特殊对象，它将模型中一对多的关系简化为一对一，将多对多简化为多对一。对图 4.16 中的关联，取文件名作为限制子，此时的关联就变为一对一了，如图 4.17 所示。

(6) 或关联

个人或者公司均可购买多个房产，但个人和公司不能购买同一个房产，即一个房产只能归属于个人或公司中的一方，这称为或关联，如图 4.18 所示。此时要在两个关联间加上虚线，并标以“{or}”来描述。



图 4.17 限制关联

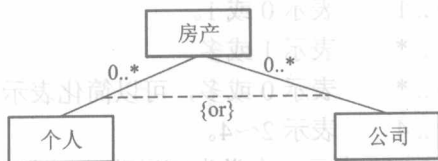


图 4.18 或关联

(7) 关联类

当两个类之间的关联重数是多对多时，可以把该关联定义成关联类。关联类也可有属性、操作和其他的关联。在图 4.19 中，用户和工作站的授权关联就是关联类的例子。

3. 聚集关系

聚集是一种特殊形式的关联。一般表示类之间具有整体与部分的关系，还有一种特殊聚集称为组合关系。在 UML 中，聚集表示为空心菱形，组合表示为实心菱形。

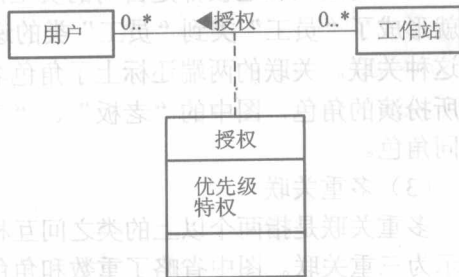


图 4.19 关联类

(1) 聚集

一般聚集也称共享聚集 (shared aggregation)，其特征是“部分”对象可以是多个任意“整体”对象的一部分。例如，课题组包含许多人，但是每个人又可以是另一个课题组的成员，即“部分”可以参与到多个“整体”中。图 4.20 表示“课题组”类和“个人”类间的共享聚集。

(2) 组合

在组合中，“整体”强烈拥有“部分”，“部分”与“整体”共存。如果“整体”不存在了，“部分”也会随之消失。例如，一个窗口由标题、外框和显示区组成，一旦窗口被关闭，则各部分同时消失。“整体”的重数必须是 0 或 1，而“部分”的重数可以是任意的。图 4.21 给出了组合的例子。



图 4.20 聚集关系

图 4.21 组合关系

4. 泛化

泛化也称为继承，例如，飞鸟和走兽可泛化为动物，男人和女人可泛化为人。UML 对泛化有 3 个要求：

- ① 一般元素所具有的关联、属性和操作，特殊元素也都隐含性地具有。
- ② 特殊元素应包含额外信息。
- ③ 允许使用特殊元素实例的地方，也应能使用一般元素。

泛化又分为普通泛化和限制泛化，下面分别介绍。

(1) 普通泛化

在 UML 中，泛化常表示为一端带空心三角形的连线。空心三角形紧挨着父类，子类则继承父类的属性、操作和所有的关联关系。以图 4.22 为例，父类是“交通工具”，“车”和“船”是它的子类；与此同时，“车”又是“轿车”、“卡车”和“客车”的父类。

没有具体对象的类称为抽象类，可用于描述其子类的公共属性和行为（操作）。图 4.22 中的“交通工具”就是一个抽象类，一般用一个附加标记值“{abstract}”来表示。

(2) 限制泛化

限制泛化是指在泛化关系上附加一个约束条件，以便进一步说明泛化关系的使用方法或扩充方法。在 UML 中，预定义的约束有 4 种：多重、不相交、完全和不完全（也称非完全）。

- ① 多重继承指的是子类的子类可以同时继承多个上一级子类。图 4.23 中，“水陆两用”

类就是通过多重继承得到的，子类“车”和“船”能被“水陆两用”类同时继承。允许多重继承的父类（“交通工具”）被“水陆两用”类继承了两次。

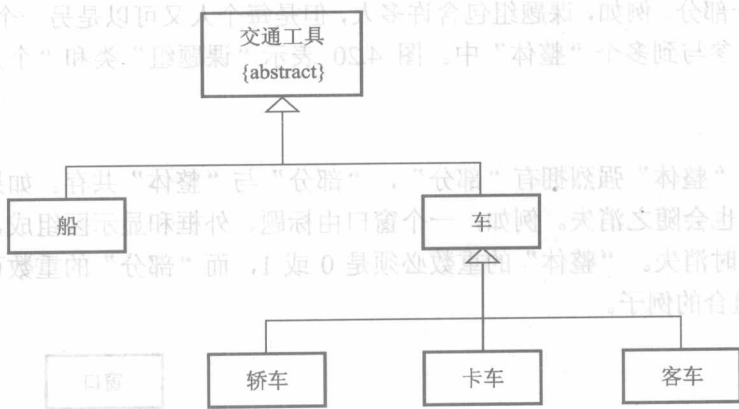


图 4.22 普通泛化

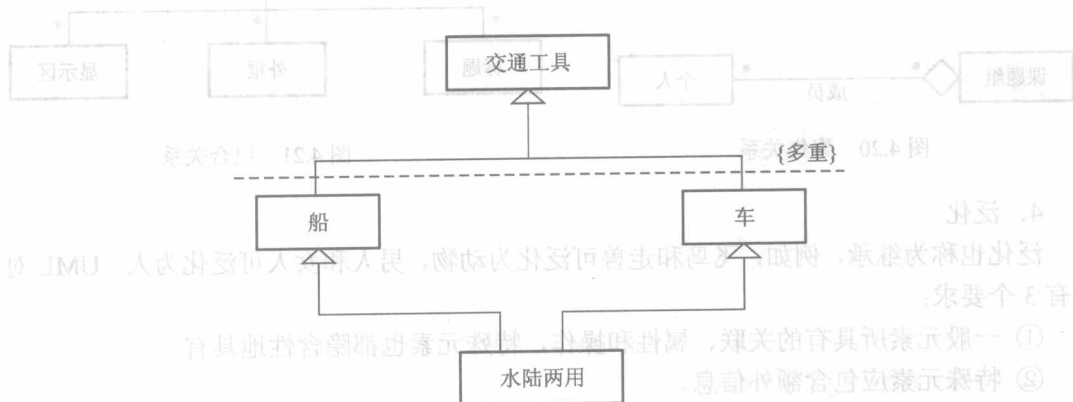


图 4.23 多重继承

② 与多重继承对立的是不相交继承，即一个子类不能同时继承多个上一级子类。图 4.23 中如果没有指定“多重”约束，则不允许“水陆两用”类如此继承。如不作特别声明，一般的继承都是不相交继承。

③ 完全继承是指父类的所有子类都被穷举完毕，不可能再有其他未列出的子类存在。图 4.24 所示的就是一个完全继承。

④ 不完全继承恰好与完全继承相反，父类的子类可以不断地补充和完善。不完全继承是默认的继承标准。

5. 依赖

假设有两个元素 X、Y，如果修改 X 的定义可能会引起对 Y 的定义的修改，则称元素 Y 依赖于元素 X。例如，某个类中使用另一个类的对象作为操作中的参数，一个类存取作为全

局对象的另一个类的对象，或一个类的对象是另一个类的类操作中的局部变量等，都表示这两个类之间有依赖关系。依赖关系的图形表示为带箭头的虚线，箭头指向独立的类，箭头旁边还可以带一个标签，具体说明依赖的种类。图 4.25 所示的是一个友元依赖（friend dependency）关系。它使其他类中的操作可以存取该类中的私有或保护属性。

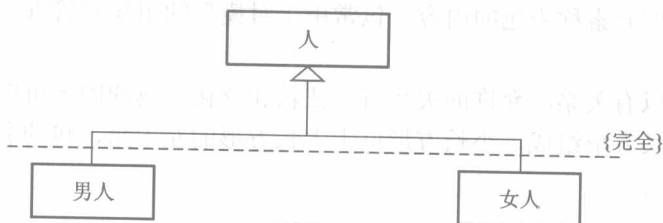


图 4.24 完全继承

6. 约束与派生

前已介绍，约束包含或关联、有序关联等，继承也分多重、不相交、完全和不完全等约束；派生则用于描述某种事物的产生规则，例如，人的年龄可以由出生日期和当前日期派生出来（两者之差）。

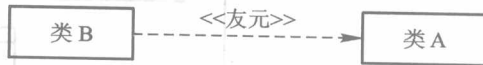


图 4.25 友元依赖关系

约束和派生机制能应用于任何模型元素。

在模型图中，约束和派生一般用花括号括起来放在模型元素旁边，或者用圆括号括起来以注释的方式与模型元素相连。

典型的属性约束是该属性的取值范围，例如，属性“年龄”的约束可以是 $\{0 \leq \text{年龄} < 150\}$ ；派生属性可由其他属性通过某种方式计算得到，通常在派生属性前面加一个“/”表示，但它不出现在类的对象中。派生属性的计算公式用花括号括起来放在类的下方，图 4.26 中的属性“利润”就是一个派生属性。

关联关系可以被约束，也可以被派生。如果一个关联是另一个关联的子集，则它们之间就有约束关联。例如，客户和服务公司之间存在着合同关联，但其中一些重要客户与服务公司间可派生出“金牌客户”关联。图 4.27 中的“管理”关联中的学生是“组成”关联中学生的子集。

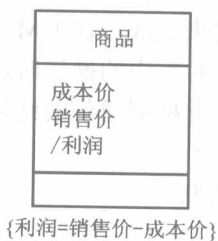


图 4.26 派生属性

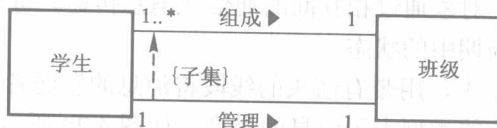


图 4.27 约束关联

4.3.3 包

在 OO 设计中，可将许多类集成一个更高层次的单位，形成一个高内聚、低耦合的类的集合。UML 把这种将一些模型元素组织成语义上相关的组的分组机制称为包。

包中的所有模型元素称为包的内容。包常用于对模型的组织和管理。包的实例没有任何语义。

包与包之间可以有关系，允许的关系有：依赖和泛化。包的图形可以表示为类似书签卡片的形状，由两个长方形组成，小长方形位于大长方形的左上角，包的名字可以写在大长方形内，如图 4.28 所示。

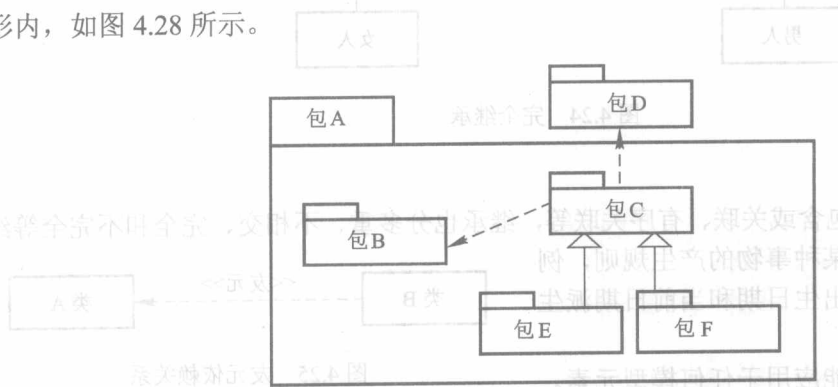


图 4.28 包图

4.4 动态建模

UML 不仅可描述系统的静态结构，同时也提供了描述系统动态行为的图形工具，如状态图、时序图、协作图和活动图等，用于描述系统中的对象在执行期间的不同时间点是如何进行动态交互的。

4.4.1 消息

在面向对象技术中，对象间的交互通过对象间的消息传递来完成。在 UML 的 4 种动态模型中，均用到了消息这个概念。通常当一个对象调用另一个对象中的操作时，即完成了一次消息传递；对象通过相互间的通信（消息传递）进行合作，并根据通信的结果不断改变其自身在生存周期中的状态。

在 UML 中，用带有箭头的线段将消息的发送者和接收者联系起来，箭头的类型表示消息的类型，如图 4.29 所示。



图 4.29 消息符号

UML 定义的消息类型有以下 3 种：

① 简单消息 (simple message)：表示简单的控制流。用于描述控制是如何在对象间进行传递的，而不考虑通信的细节。

② 同步消息 (synchronous message)：表示嵌套的控制流。操作的调用是一种典型的同步消息。调用者发出消息后必须等待消息返回，只有当处理消息的操作执行完毕后，调用者才可继续执行自己的操作。

③ 异步消息 (asynchronous message)：表示异步控制流，主要用于描述实时系统中的并发行为。当调用者发出消息后，不用等待消息的返回即可继续执行自己的操作。

4.4.2 状态图

状态图 (state diagram) 用来描述一个特定对象的所有可能状态以及引起其状态转移的事件。大多数面向对象技术都用状态图表示单个对象在其生存周期中的行为。一个状态图包括一系列的状态以及状态之间的转移。

1. 状态

状态是对象执行了一系列活动的结果，它通常由其属性值和与其他对象的链接来确定。例如，飞机 (对象) 起飞 (状态)，电话 (对象) 响铃 (状态)，电梯 (对象) 停在底楼 (状态)。当某个事件发生后，对象的状态将发生变化，从一个状态转移到另一个状态。

状态图有初态、终态与中间状态 3 种状态。一个状态图只能有一个初态，而终态则可以有多。图 4.30 是电梯的状态图。电梯从底楼启动，当它在某层处于空闲状态时，如发生上楼或下楼事件，就会向上或向下移动，如发生超时事件，就会返回底楼。

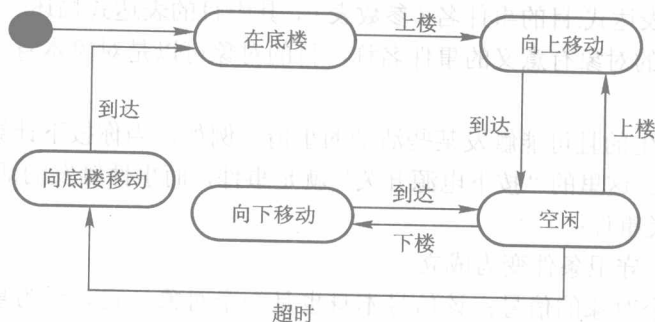


图 4.30 电梯状态图

一个状态由状态名、状态变量和活动 3 部分组成，其中状态变量和活动是可选的。前者表示对象在该状态时的属性值；后者列出在该状态时要执行的事件和动作。可用 3 个标准事件 entry、exit 和 do 分别说明在进入状态、退出状态和处于进入/退出状态中间时需执行的动作。

活动的表示格式一般为：

事件名 / 参数表 / 动作表达式

图 4.31 给出了一个 login 状态的例子。

2. 状态转移

对象从一种状态改变成另一种状态称为状态转移，在状态图中用带箭头的连线表示。状态的变迁通常是由事件触发的，应在转移上标出触发转移的事件表达式；如果未标明事件，则表示在源状态的内部活动执行完毕后自动触发转移。状态转移的表示格式一般为：

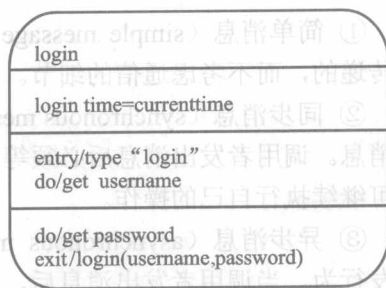


图 4.31 login 状态

事件说明 [守卫条件] / 动作表达式 ^ 发送子句

事件说明 (event_signature) 由事件名后接用括号括起来的参数表组成，它指出触发转移的事件以及与该事件相连接的附加数据，参数由参数名、冒号和类型组成。

守卫条件 (guard_condition) 是一个布尔表达式，如果一个转移中同时有守卫条件和事件说明，则当且仅当条件为真且事件发生时相应的状态转移才会发生。如果只有守卫条件，则只要条件为真，状态转移就会发生。

动作表达式 (action_expression) 是一个触发状态转移时可执行的过程表达式，一个状态转移上可以有多个动作表达式，它们之间用“/”分隔，执行时按从左到右的次序。不允许有嵌套的动作表达式或递归的动作表达式。

发送子句 (send_clause) 是动作的一个特例，它被用来在两个状态转移之间发送消息，它的格式为：目的表达式.目的事件名(参数表)，其中目的表达式描述一个对象或一组对象，目的事件名是对目的对象有意义的事件名称，目的对象可以是对象本身。

3. 事件

事件是指已发生的且可能触发某些活动的事情。例如，当你按下计算机的电源开关时，计算机就开始启动。这里的“按下电源开关”就是事件，而事件触发的活动是“开始启动”。

UML 中有 4 类事件：

- ① 条件变真：守卫条件变为成立。
- ② 收到另一个对象的信号：该信号本身也是一个对象，它表示为事件说明，这类事件也称消息。
- ③ 收到其他对象（或对象本身）的操作调用：它表示为事件说明，这类事件也称消息。
- ④ 经过指定的时间间隔：该时间通常是在另一个指定的事件后被计算或经过一个给定的时间量，它表示状态转移上的时间表达式。

4. 在状态图之间发送消息

状态图可以给其他状态图发送消息。状态图之间发送消息可以通过动作（即在发送子句中指定接收者）或在状态图间用虚线箭头来表示。如果使用虚线箭头来表示，则必须使用矩形框将状态图中的所有对象组合在一起。

图 4.32 表示的是遥控器对象的状态图和 CD 机对象的状态图之间的消息发送。

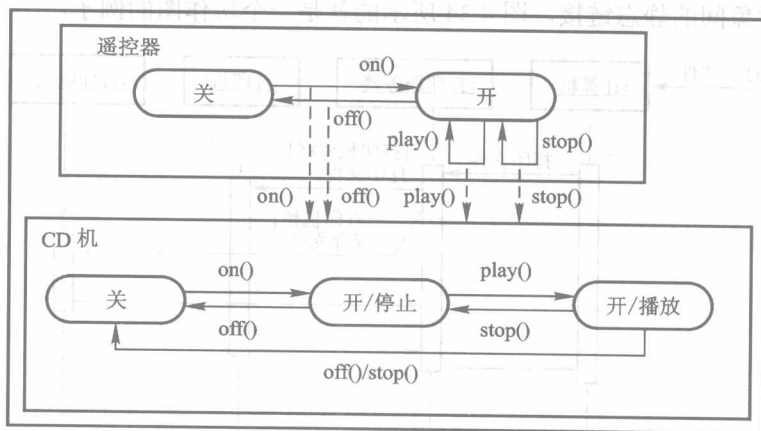


图 4.32 遥控器发消息给 CD 机

4.4.3 时序图和协作图

1. 时序图

时序图 (sequence diagram) 用来描述对象之间的动态交互，着重体现对象间消息传递的时间顺序。它以垂直轴表示时间，水平轴表示不同的对象。对象用一个带有垂直虚线的矩形框表示，并标有对象名和类名。垂直虚线是对象的生命线，用于表示在某段时间内对象是存在的。对象间的通信在对象的生命线间通过消息符号来表示，消息的箭头指明消息的类型，如图 4.29 所示。

时序图中的消息可以是信号 (signal) 或操作调用，或类似于 C++ 中的 RPC (remote procedure calls) 和 Java 中的 RMI (remote method invocation)。当收到消息时，接收对象即开始活动，表明对象被激活。通过在对象生命线上显示一个细长矩形框来表示激活。

消息可以用消息名及参数来标识。消息还可带有条件表达式，表示分支或决定是否发送消息。如果用于表示分支，则每个分支是相互排斥的，即在某一时刻仅可发送分支中的一个消息。

在时序图的左边可以有说明信息，用于说明消息发送的时刻、描述动作的执行情况以及约束信息等。一个典型的例子就是用于说明一个消息是重复发送的。另外，可以定义两个消息间的时间限制。图 4.33 所示的就是一个文档打印系统的时序图。

一个对象可以通过发送消息来创建另一个对象，当一个对象被删除或自我删除时，该对象用“×”标识。

2. 协作图

协作图 (collaboration diagram) 用于描述相互协作的对象间的交互和链接。虽然时序图

也可用来描述对象间的交互，但侧重点不同：时序图着重体现交互的时间顺序，而协作图则着重体现交互对象间的静态链接。图 4.34 所示的就是一个协作图的例子。

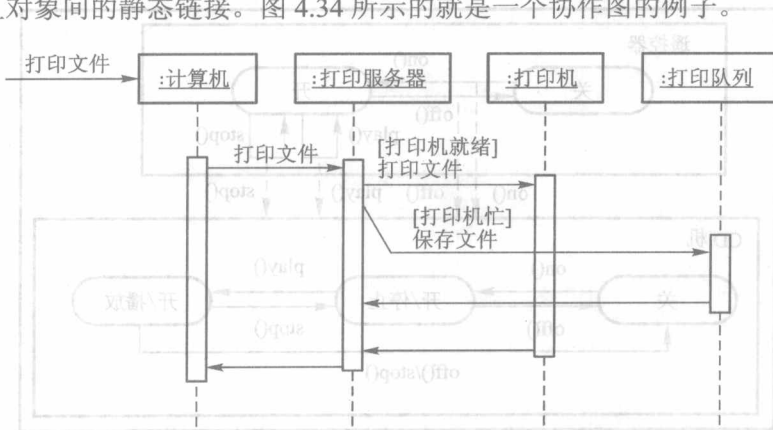


图 4.33 文档打印系统的时序图

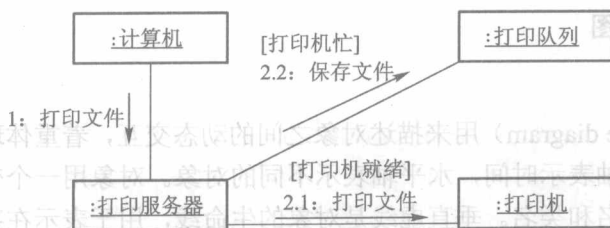


图 4.34 文档打印系统的部分事件流程图

协作图中对象的外观与时序图中的对象类似。如果一个对象在消息的交互中被创建，可在对象名称之后标以“{new}”；类似地，如果一个对象在交互期间被删除，可在对象名称之后标以“{destroy}”；“{transient}”则表示对象在同一个协作期间被创建并消亡。对象间的链接关系类似于类图中的关联（但无重数标志）。通过在对象间的链接上标志带有消息串的消息（简单、异步或同步消息）来表达对象间的消息传递。

(1) 链接

链接是两个对象之间的一种连接，用于表示对象间的各种关系。各种链接关系与类图中的定义相同，在链接的端点位置可以显示对象的参与者名和约束。在链接上附加的约束可以是 global、local、parameter、self、vote 或 broadcast。

① global 是加在链接参与者上的约束，说明与对象对应的实例是可见的，因为它是在全局范围内的（可以通过系统范围内的全局名来访问对象）。

② local 也是加在链接参与者上的约束，说明与对象对应的实例是可见的，因为它是操作中的一个局部变量。

③ **parameter** 也是加在链接参与者上的约束，说明与对象对应的实例是可见的，因为它是操作中的一个参数。

④ **self** 是加在链接参与者上的约束，说明一个对象可以给自己发送消息。

⑤ **vote** 是加在消息上的约束，限制一组返回值。**vote** 限制说明返回值必须在返回的值中通过多数投票才能选出。

⑥ **broadcast** 是加在一组消息上的约束，说明这组消息不按一定顺序激活。

(2) 消息流

在协作图的链接线上，可以用带有消息串的消息来描述对象间的交互。消息的箭头指明消息的流动方向。消息串说明要发送的消息、消息的参数、消息的返回值以及消息的序列号等信息。协作图中的消息串用下面的语法规则来书写：

前缀 守卫条件 序列表达式 返回值: =说明

其中，前缀 (**predecessor**) 用下面的语法来描述：

序列号 , ... /

前缀是一个用来同步的表达式，意思是在发送当前消息之前指定序列号的消息被处理（也就是说，必须连续执行）。序列号之间用逗号分隔。

守卫条件用下面的语法来描述：

[条件子句]

条件子句通常用伪代码或真正的编程语言来表示。UML 并不规定其语法。

序列表达式的语法规则可定义如下：

[integer | name][recurrence]:

其中，**integer** 为指定消息顺序的序列号。消息 1 总是消息序列的开始消息，消息 1.1 是消息 1 的处理过程中的第一条嵌套的消息，消息 1.2 是消息 1 的处理过程中的第二条嵌套的消息。一个消息序列的例子如消息 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3 等。这样的序列号不仅能够表示消息的顺序，而且还能表示消息的嵌套关系（当消息是异步消息时，消息为嵌套的操作调用及返回）。**name** 表示并发控制线程。例如，1.2a 和 1.2b 为同时发送的并发消息。序列表达式用冒号连接消息内容。

recurrence 表示一个条件或迭代的执行。有两种选择：

* [循环子句]

[循环子句]

循环子句 (**iteration-clause**) 用来指定一个循环（重复执行），它是循环的条件，如 **[i: =1..n]**。例如，对于一个包括循环的消息标签应该表示成：

1. 1 ***[x: 1..10]** : doSomething()

而条件子句一般用来表示分支，而不是用作守卫条件。**[x<0]**和**[x>=0]**是两个可以用来分支的条件子句，这两个条件只能有一个为真，因而只有一个分支被执行（发送与分支有关的消息）。条件子句和循环子句都可以用伪代码或真正的编程语言来表示。

应该给返回值指定一个消息说明。消息说明由消息名和参数表组成。返回值表示一个操作调用（消息）的结果。

4.4.4 活动图

活动图（activity diagram）显示动作流程及其结果，它既可用于描述操作（类的方法）的行为，也可以描述用例和对象内部的工作过程。活动图是由状态图变化而来的，它们各自用于不同的目的。活动图依据对象状态的变化来捕获动作（将要执行的工作或活动）与动作的结果。与状态图不一样，活动图中动作状态之间的迁移不是靠事件触发的，当动作状态中的活动完成时就触发迁移，活动图中一个活动结束后将立即进入下一个活动。在活动图中，还使用了泳道（swimlane）概念。

1. 活动和转移

一项操作可以描述为一系列相关的活动。活动仅有一个起点，但可以有多个终点，起点用黑圆点表示，终点用黑圆点外加一个小圆圈表示。

活动间的转移允许带有守卫条件、发送子句和动作表达式，其语法与状态图中的定义相同。一个活动可以顺序地跟在另一个活动之后，这是简单的顺序关系。如果在活动图中使用一个菱形的判断标志，则可以表达条件关系，如图 4.35 所示，判断标志可以有多个输入和输出转移，但在活动的运作中仅触发其中的一个输出转移。

2. 泳道

活动图可表示发生了什么，但不能表达该项活动由谁来完成。在程序设计中，这意味着活动图没有描述出各个活动由哪个类来完成。泳道解决了这一问题，它将活动图的逻辑描述与时序图、协作图的描述结合起来。泳道用矩形框来表示，属于某个泳道的活动放在该矩形框内，将对象名放在矩形框的顶部，表示泳道中的活动由该对象负责。

3. 对象

在活动图中可以出现对象。对象可以作为活动的输入或输出，对象与活动间的输入输出关系由虚线箭头来表示。如果仅表示对象受到某一活动的影响，则可用不带箭头的虚线来连接对象与活动。

4. 信号

在活动图中可以表示信号的发送与接收，分别用发送和接收标志来表示。发送和接收标志也可与对象相连，用于表示消息的发送者和接收者。

以上对 UML 中用于描述系统动态行为的状态图等 4 种图简单地作了介绍，它们均可用于系统的动态建模，但各自的侧重点不同，分别用于不同的场合。下面对如何正确使用这几类图进行总结，在实际的建模过程中，要根据具体情况灵活运用。

首先，不需要对系统中的每个类都画出状态图。状态图描述跨越多个用例的单个对象的行为，而不适合描述多个对象间的行为合作。为此，常将状态图与其他图（如时序图、协作图和活动图）组合使用。

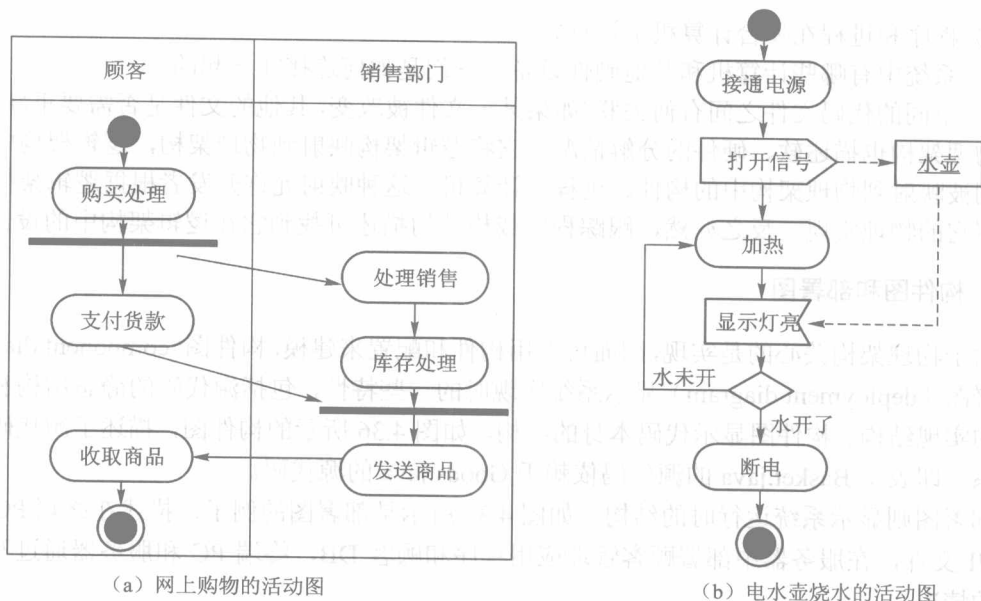


图 4.35 活动图示例

时序图和协作图适合描述单个用例中几个对象的行为。其中时序图突出对象间交互的顺序，而协作图的布局方法能更清楚地表示出对象之间静态的连接关系。当行为较为简单时，时序图和协作图是较好的选择。但当行为比较复杂时，这两个图将失去其清晰度。因此，如果想表现跨越多用例或多线程的复杂行为，应考虑使用活动图。另外，时序图和协作图仅适合描述对象之间的合作关系，而不适合对行为进行精确定义，如果仅描述跨越多个用例的单个对象的行为，仍应使用状态图。

4.5 物理架构建模

系统架构是对系统各部分构成（结构、接口、通信机制）的框架性描述。架构给开发人员提供了目标系统的视图，通过它可使开发人员知道系统是如何构造的，以及某一构件或子系统在何处。

4.5.1 物理架构

物理架构详细描述系统的软件和硬件。它所描述的硬件结构，包括不同的结点以及结点间的连接关系。物理架构还说明实现逻辑架构中定义的概念所对应的代码模块的物理结构和相关性，以及软件运行时，进程、程序和其他构件的分布情况。物理架构试图有效地利用软、硬件资源，并需解决以下问题：

- ① 类和对象物理上分布在哪个程序或进程中？

- ② 程序和进程在哪台计算机上运行?
- ③ 系统中有哪些计算机和其他硬件设备,它们是如何连接在一起的?
- ④ 不同的代码文件之间有何关联?如果某一文件被改变,其他的文件是否需要重新编译?

物理架构也描述软、硬件的分解情况,它将逻辑架构映射到物理架构,逻辑架构中的类和机制被映射到物理架构中的构件、进程和计算机。这种映射允许开发者根据逻辑架构中的类找到它的物理实现。反之亦然,跟踪程序或构件的描述可找到它在逻辑架构中的设计。

4.5.2 构件图和部署图

由于物理架构关心的是实现,因而可以用构件和配置来建模。构件图(component diagram)和部署图(deployment diagram)显示系统实现时的一些特性,包括源代码的静态结构和运行时刻的实现结构。构件图显示代码本身的结构,如图4.36所示的构件图,描述了源代码的依赖关系,即表示Basket.java的源代码依赖于Goods.java的源代码。

部署图则显示系统运行时的结构,如图4.37所示是部署图的例子,描述在终端PC中部署GUI文件,在服务器中部署顾客管理应用程序和顾客DB,终端PC和服务器通过互联网连接的情况。

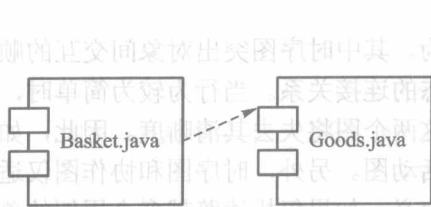


图 4.36 构件图

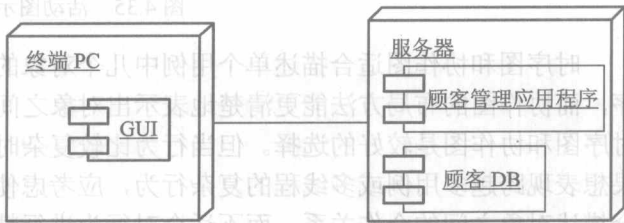


图 4.37 部署图

① 构件图。构件图显示软件构件之间的依赖关系。一般来说,软件构件就是一个实际文件,可以是源代码文件、二进制代码文件和可执行文件等。构件图可以用来表现编译、链接或执行时构件之间的依赖关系。

② 部署图。部署图描述系统硬件的物理拓扑结构以及在此结构上执行的软件。部署图可以显示计算结点的拓扑结构和通信路径、结点上运行的软件构件、软件构件包含的逻辑单元(对象、类)等。部署图常常用于帮助理解分布式系统。

③ 结点和连接。结点(node)代表一个物理设备以及其上运行的软件系统,如一台UNIX主机、一个PC终端、一台打印机、一个传感器等。结点之间的连线表示系统之间进行交互的通信路径,在UML中被称为连接(connection)。通信类型则放在连接旁边的“<< >>”之间,表示所用的通信协议或网络类型。

④ 构件和接口。在部署图中,构件代表可执行的物理代码模块,如一个可执行程序。逻辑上它可以与类图中的包或类对应。因此,部署图中显示运行时各个包或类在结点中的分布情况。在面向对象方法中,类和构件等元素并不是所有的属性和操作都对外可见。但它们

对外提供了可见操作和属性，称之为类和构件的接口。接口可以表示为一端是小圆圈的直线。

⑤ 对象 (object)。一个面向对象软件系统中可以运行很多对象。由于构件可以看作与包或类对应的物理代码模块，因此，构件中应包含一些运行的对象。部署图中的对象与对象图中的对象表示法一样。

4.6 UML 工具

随着 UML 在软件开发中的日益普及，人们研制了很多软件工具用于 UML 建模，从而代替手工建模，这不仅提高了效率，保证一致性，而且使模型的自动转换和生成成为可能。下面介绍两种常用的 UML 工具：Rational Rose 和 StarUML。

4.6.1 Rational Rose

Rational Rose 是对软件系统进行面向对象分析和设计的功能强大的可视化工具。软件开发团队可以使用 Rational Rose 来对系统进行建模，然后再编写代码，从而保证系统结构合理；同时，利用可视化模型可以方便地捕获设计缺陷，从而降低修正这些缺陷的成本。在过去的软件开发中，程序员利用手工建模，既耗费了大量的时间和精力，又无法对整个复杂系统进行全面、准确的描述，以致直接影响应用系统的开发质量和速度。

Rose 模型是用图形符号对系统的需求和设计进行形式化描述。描述语言是 UML，它支持各种 UML 图，开发人员可以用模型作为所建系统的蓝图。由于 Rose 模型包含许多不同的图，使项目小组（客户、设计人员、项目经理、测试人员等）可以从不同角度来分析这个系统。

目前 Rational Rose 已经从原先单一的建模工具发展成为一套完整的软件开发工具族，包括系统建模、模型集成、源代码生成、软件系统测试、软件文档的生成、模型与源代码间的双向工程、软件开发项目管理、团队开发管理以及 Internet Web 发布等工具，构成了一个强大的软件开发集成环境。

1. Rose 界面

Rose 提供了一套十分友好的界面来让用户对系统进行建模。Rose 界面包括浏览区、工具栏、图形窗格、文档窗格和日志窗格，如图 4.38 所示。

其中，浏览区用于在模型中迅速漫游，它可以显示模型中的参与者、用例、类和组件等。浏览区可按 4 种视图方式显示，分别是用例视图、逻辑视图、构件视图和部署视图。

文档窗格可以查看或更新模型元素的文档。

工具栏用于访问常用命令。Rose 中有两个工具栏：标准工具栏和图形工具栏。标准工具栏总是显示包含任何图形中都可以使用的选项；图形工具栏随每种 UML 图形而改变。

图形窗格用于显示和编辑一个或几个 UML 图形。改变图形窗格中的元素时，Rose 会自动更新浏览区。同样，在浏览区中改变某一元素时，Rose 自动更新相应图形，从而保证模型的一致性。



图 4.38 Rose 界面

日志窗格用于查看错误信息和报告各个命令的结果。

2. Rose 模型的 4 个视图

Rose 模型的 4 个视图分别是用例视图、逻辑视图、构件视图和部署视图。每个视图针对不同对象，具有不同用途。

用例视图包括系统中的所有参与者、用例和用例图，还可能包括一些时序图或协作图。用例视图是系统中与实现无关的视图，关注系统功能的高层形式，而不关注系统的具体实现方法。项目开始时，用例视图的主要使用者是客户、分析人员和项目管理员，这些人员利用用例、用例图和相关文档来确定系统的高层视图。随着项目的推进，小组的所有成员可通过用例视图了解正在建立的系统，通过用例描述事件流程。利用用例视图，质量保证人员可以开始编写测试脚本，技术文档制作者可以开始编写用户文档，分析人员和客户可以从中确认捕获了所有需求。一旦用户同意了用例模型，就确定了系统的范围，然后就可以在逻辑视图中继续开发，关注系统如何实现用例中提出的功能。

逻辑视图提供系统的详细描述，主要包括类、类图、交互图（时序图和协作图）、状态图等。开发小组中每个人都会用到逻辑视图中的信息，但主要是设计人员和架构师。软件设计人员关心生成哪些类，每个类所包含的信息和功能；架构师则更关心系统的总体结构，他们要负责保证系统结构稳定，因此必须考虑复用，使系统能灵活地适应需求变化。一旦标识类并创建逻辑视图后，就可以转入构件视图，了解物理结构。

构件视图包含模型代码库、执行文件、运行库和其他组件的信息。构件视图的主要用户是负责控制代码和编译部署应用程序的人。

部署视图关注系统的实际部署。显示网络上的进程和设备及其相互间的实际连接。

3. 使用 Rose

Rose 中的所有工作都是基于所创建模型完成的。

使用 Rose 的第一步是创建模型。模型可以从头创建，也可以使用现有框架来构造模型。

Rose 模型（包括所有图形、模型元素）都保存在一个扩展名为.mdl 的文件中。

创建新模型的步骤为从菜单中选择 File→New，或单击标准工具栏中的 New 按钮，此时会出现一些可用框架，只要选择要用的应用框架就可以了。每个应用框架针对不同的编程语言，提供语言本身的预制模型和应用开发框架。选择 Cancel 则不使用系统提供的框架。

要保存模型，从菜单中选择 File→Save，或单击标准工具栏中的 Save 按钮。整个模型都保存在一个文件中。

面向对象机制的一大好处是可重复使用。重复使用不仅适用于代码，也适用于模型。Rose 支持对模型和模型元素的导入与导出。选择菜单 File 中的 Export Model 可进行导出操作；选择 Import Model 可进行导入操作。这样，可以对现有模型进行复用。

4.6.2 StarUML

开源软件 StarUML 是一个运行在 Windows 上快速、灵活、可扩展、功能强大的 UML/MDA 平台。StarUML 提供了一个可以用来代替知名商业 UML 工具的建模工具和平台。

StarUML 是一种软件建模平台，它基于 UML 1.4，并兼容 UML 2.0 符号，支持 11 个不同类型的图。StarUML 具有很强的可扩展性，擅长定制到用户的环境下，和其他工具集成使用。熟练使用 StarUML 软件建模工具，可保证软件项目开发的生产效率和质量。

StarUML 提供了良好的可扩展性和灵活性。图 4.39 是 StarUML 的主界面，关于 StarUML 的详细介绍参见 <http://www.staruml.com>，这里不再详细介绍了。

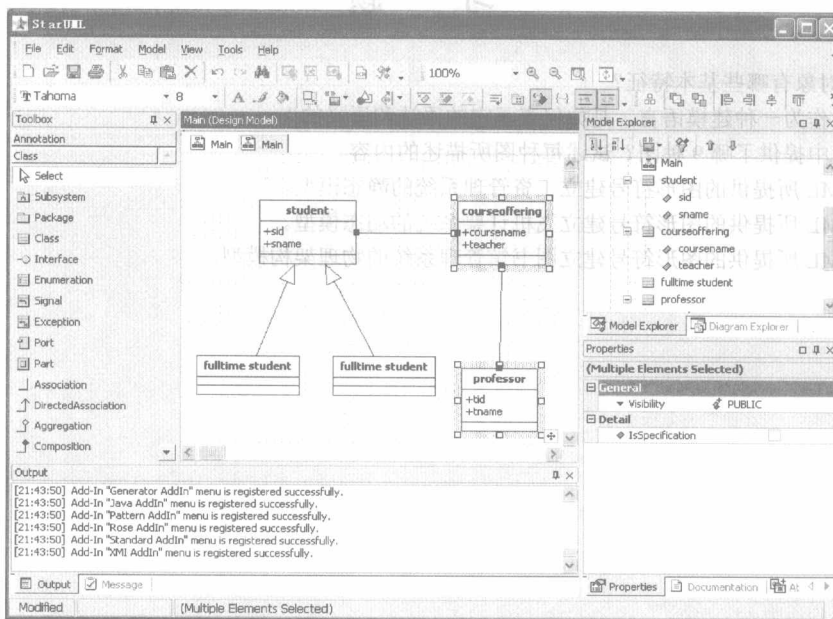


图 4.39 StarUML 的主界面

小结

面向对象开发是按人的思维方式来理解和解决问题的，将问题空间的概念直接映射到解空间。面向对象的基本特征是抽象、封装、继承和多态。利用这种方法开发的软件具有很好的软件可复用性、可扩展性和可维护性。

统一建模语言（UML）实现了面向对象建模工具的统一，现已成为面向对象方法中的一个标准。它是在分析了许多学者提出的各种软件建模方法的基础上，通过软件工程师 Grady Booch、Ivar Jacobson、Jim Rumbaugh 等人的共同努力，综合了他们各自的面向对象分析和设计方法，并吸收其他方法的优点的基础上提出来的。

作为一种著名的建模语言，UML 用图表示它的语法，用元模型表达它的语义，采用模型来描述系统的结构（或静态特征）以及行为（或动态特征）。它能从不同的视角为系统建模，形成不同的视图；每个视图代表系统完整描述中的一个抽象，显示系统中的一个特定的方面；每个视图由一组图构成，其中包含了强调系统中某一方面的信息。本章介绍了 UML 中包含的 5 种视图和 9 种图，以及由用例图、类图、对象图、构件图、部署（配置）图表示的系统静态模型，由状态图、时序图、协作图、活动图描述的系统动态模型。

习 题

1. 面向对象有哪些基本特征？
2. UML 作为一种建模语言，是如何表示它的语法和语义的？
3. UML 中提供了哪 9 种图？试述每种图所描述的内容。
4. 用 UML 所提供的图形符号建立工资管理系统的静态模型。
5. 用 UML 所提供的图形符号建立飞机订票系统的动态模型。
6. 用 UML 所提供的图形符号建立图书馆管理系统的物理架构模型。

第5章 需求工程与需求分析

“需求工程”与“需求分析”，是软件需求中一对既相互联系、又存在重要区别的基本概念。作为软件开发的预备知识，本章将首先阐明这两个概念的含义。然后在第3章讨论的结构化分析的基础上，介绍一种面向对象的需求建模方法——基于用例模型的软件需求建模，并给出若干示例，供读者学习参考。

5.1 软件需求工程

“万事开头难”，就软件开发而言，首要任务是确定软件需求。据统计，软件项目中40%~60%的问题源自软件需求阶段，因为需求模糊或错漏都会造成软件开发者与用户对软件的理解产生差异。那么，究竟什么是软件需求？它应该包括哪些内容呢？

5.1.1 软件需求的定义

在装修一套房子之前，首先要明确对装修效果的要求。用户和装修公司会讨论有关房子装修的各种细节，使双方正确理解房子将装修成什么样子，可称之为装修需求。

软件需求主要指一个软件系统必须遵循的条件或具备的能力。这里的条件或能力可以从两个方面来理解：一是用户解决问题或达到目标所需的条件或能力，即系统的外部行为；二是系统为了满足合同、规范或其他规定文档所需具有的条件或能力，即系统的内部特性。

软件需求一般包括3个不同的层次：业务需求、用户需求和功能需求。

第一个层次是业务需求，这是客户或市场对软件的高层次目标要求。通过对企业目前的业务进行评估，包括对业务流程建模、对业务组织建模、改进业务流程、领域建模等方面，并结合未来一段时间内可能的业务发展需要，即可归结出业务需求。具体说，就是从业务的角度分析项目成功的预期效果。在确定业务需求前，还应该在各类业务相关人员范围内达成一致。它好比需求过程中的基石，其他需求（如用户需求和功能需求）都必须与之相符。

第二个层次是用户需求，即从用户使用角度来描述软件产品必须完成的任务。通常在用例模型文档中描述这个层次的需求，同时，从用户需求还可以引申出软件的质量属性，例如软件可持续正常工作的时间等。用户需求的重心，是如何收集用户的需求，即确定软件系统为用户提供的功能以及软件与环境的交互。获取用户需求不是一件容易的事，因为很多需求是隐性、非直接和含糊不清的，很难正确获取并保证需求完整，而且需求又是易变的。

第三个层次是功能需求，它定义软件开发人员必须实现的软件功能，以及为了有效实现这些功能而必须达到的非功能要求、约束条件等，从而使用户能完成他们的任务，满足业务需求。功能需求依赖于用户需求，是用户需求在系统上的具体反映。从用户需求到功能和非功能需求，思考的角度从用户转移到了开发者。在这个层次上，通常可利用快速原型法为用户开发一个软件原型，先让用户对软件有一个直观的印象和概念，以降低用户在软件开发完成后才看到软件所带来的风险。有时也可以用一些简便的方法来生成原型。例如，要开发一个 Web 系统，可以先做几个页面给用户看看；如果开发一个 C/S 系统，即使做一个界面给用户看看也很有用，甚至在黑板上勾画将来软件的概貌也可以。

图 5.1 显示了这 3 个需求层次之间的关系。

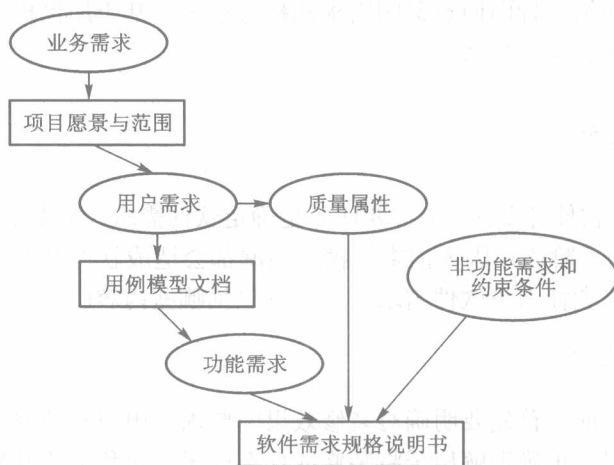


图 5.1 软件需求的层次关系

5.1.2 软件需求的特性

从上述 3 个层次的软件需求可以看出，要描述一个软件，必须从功能、非功能和质量等多个方面来进行。由此可见，软件需求包括以下 6 个特性：功能性、可用性、可靠性、性能、可支持性和设计约束，以下将逐一介绍。

1. 功能性

功能性需求是软件最重要的需求，也是最直观、用户最关心的软件需求，它又可分为普通功能和全局性功能。普通功能泛指软件完成的一个功能或提供的一个服务，例如，一个订单管理软件通常有输入订单和订单查询等功能；全局性功能是适用于软件所有应用场景的功能，如出错处理等。

2. 可用性

可用性泛指能使最终用户方便使用软件的相关需求，例如，系统使用者所需的培训时间，

是否符合一些常见的可用性标准，如 Windows 界面风格等。提高可用性、关注用户体验是软件获得成功的重要因素。

3. 可靠性

包括与系统可靠性相关的各种指标，主要有：

① 正常运行率：例如规定可用时间百分比，或系统处于使用、维护、降级模式（其含义见下文）等操作的小时数等。

② 平均无故障时间（mean time to failure, MTTF）：通常表示为小时数，但也可表示为天数、月数或年数。

③ 平均修复时间（mean time to repair, MTTR）：指系统在发生故障后可以暂停运行的时间。

④ 精确度：指系统的输出要求具备的精密度（分辨率）和精确度（按照某一已知的标准）。

⑤ 最高错误或缺陷率：通常表示为 bugs/KLOC（每千行代码的错误数目）或 bugs/function-point（每个功能点的错误数目）。

4. 性能

记录与系统性能相关的各种指标，其中包括：

① 对事务的响应时间：包括平均响应时间和最长响应时间。

② 吞吐量：如每秒处理的事务数。

③ 容量：如系统可以容纳的客户或事务数。

④ 降级模式：当系统以某种形式降级使用时可接受的运行模式。

⑤ 资源利用情况：内存、磁盘、通信等。

5. 可支持性

定义所有与系统的可支持性或可维护性相关的需求，其中包括编码标准、命名约定、类库以及如何对系统进行维护操作和相应的维护实用工具等。

6. 设计约束

设计约束代表已经批准并必须遵循的设计决定，其中包括软件开发流程、开发工具、系统构架、编程语言、第三方构件库、运行平台和数据库系统等。

5.1.3 需求工程的由来

需求工程是随着计算机软件的发展而发展起来的。早期的软件规模不大，开发人员主要关注的是代码编写，需求很少受到重视。后来引入了生存周期的概念，定义需求成为软件开发的第一项活动。随着软件规模不断扩大，需求定义与分析在整个软件开发与维护过程中越来越重要。人们逐渐认识到，需求活动不仅限于软件开发的最初阶段，而是贯穿于系统开发的整个生存周期。于是从 20 世纪 80 年代中期起，逐渐形成了软件工程的一个子领域——需

求工程 (requirement engineering, RE)。

进入 20 世纪 90 年代以来, 需求工程成为许多学者研究的热点。从 1993 年起, 每两年会举办一次需求工程国际会议; 1996 年起, 在 Springer-Verlag 出版发行了《Requirements Engineering》。有关需求工程的一些工作小组也相继成立并开展工作, 例如欧洲的 RENOIR (Requirements Engineering Network of International Cooperating Research Groups)。至此, 一门新的学科——“软件需求工程”应运而生。

所谓软件需求工程, 是一门应用有效的技术和方法、合适的工具和符号, 来确定、管理和描述目标系统及其外部行为特征的学科。它通过合适的工具和记号, 系统地描述待开发系统及其行为特征和相关约束, 形成需求文档, 并能对不断变化的需求演进给予支持。具体地说, 软件需求工程是一门分析、记录并维护软件需求的学科, 它把系统需求分解成若干子系统和任务, 再把这些子系统或任务分配给软件, 并通过一系列重复的分析、设计、比较研究、原型开发等过程, 把这些系统需求转换成软件的需求描述和性能参数。

要为 RE 建立模型, 一般采用图形和自然语言相结合的方法, 并需结合使用多种图形描述工具。

5.2 需求分析与建模

需求工程是软件工程的一个子领域, 贯穿于软件整个生存周期的始终。而需求分析与之不同, 它通常指软件开发的第一项活动, 而该项活动的目的主要是为待开发的软件系统进行需求定义与分析, 并建立一个需求模型 (requirement model)。

5.2.1 需求分析的步骤

软件需求分析一般包括如下的 4 个步骤: 需求获取、需求建模、需求描述 (即编写 SRS) 和需求验证, 如图 5.2 所示。

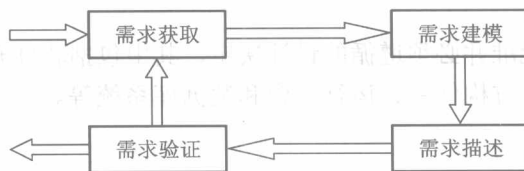


图 5.2 需求分析的步骤

1. 需求获取

需求获取通常从分析当前系统包含的数据开始。例如, 当前系统使用的账册、卡片和报表, 手工处理当前信息的方法及其不足, 用户希望改进的主要问题及其迫切性等。为了收集

全面、完整的信息，可以按使用频率、使用特性、优先级等方面对客户进行分类，然后每类选择若干用户代表，从他们那里收集所期望的软件系统功能、用户与系统间的交互和对话方式等。在获取功能需求之后，再考虑对质量的要求，包括性能、可维护性、可靠性、可用性等。如果客户的要求和已有的产品存在某些相似之处，还需考虑可否复用一些已有的软件组件。

2. 需求建模

需求分析的主要任务是建立需求模型。图形化的模型是可视化地说明软件需求的最好手段，常用的模型包括用例图、数据流图、实体联系图、控制流图和状态转换图等。

除需求模型外，有些软件还需要绘制系统关联图、创建用户接口原型、确定需求优先级等。系统关联图可用于划分系统与系统外部实体间的界限，定义接口的简单模型和通过接口的信息流和物质流。当开发人员或用户一时难以确定需求时，还可以先开发一个用户接口原型，通过原型评价，使用户和其他参与者能更好地理解所要解决的问题。对一个特定的系统，如果各项需求、软件特性的优先级别不同，还必须确定本版产品将包括哪些特性或哪类需求。

3. 需求描述

需求描述即编写软件需求规格说明书（SRS），必须用统一格式的文档进行描述。为了统一风格，可以采用已有的且可满足项目需要的模板，例如国际标准 IEEE 标准 830—1998(IEEE—1998) 或中国国家推荐性标准 GB 9385 中描述的 SRS 模板，也可以根据项目特点和软件开发小组的特点对标准进行适当的改动，形成自己的 SRS 模板。

为使所有相关人员明白 SRS 为何要提出这些功能需求，在编写 SRS 时应该：

① 指明需求的来源（例如来自客户要求或是某项更高层的系统需求），或业务规范、政策法规、标准或其他外部来源等。

② 为每项需求注上标号，以便进行跟踪；记录需求的变更，并为需求状态及其变更活动建立度量。

4. 需求验证

在很多情况下，由分析员提供的 SRS 初看起来是正确的，实现时却出现需求不清、不一致等问题。以需求规格说明书为依据编写测试计划时，也可能发现说明中的二义性。这些问题都必须通过验证来改善，确保需求规格说明书可作为软件设计和最终系统验收的依据。

5.2.2 需求分析是迭代过程

由图 5.2 可见，这 4 步周而复始，实际上组成了一个迭代的过程，直到所编写的 SRS 真正符合用户的需求为止。

从下节开始，将对前 3 个步骤逐一进行说明。

5.3 需求获取的常用方法

初学者可能认为，需求获取的手段无非是调查研究，只需多问多看即可。殊不知开发者和用户之间的交流与理解，是一条曲折不平的道路。虽然双方都希望开发获得成功，但双方都希望控制事情的进程，任何一方的意图和言语都可能被对方误解。

以下主要介绍常规的需求获取方法，以及快速原型技术在需求获取中的应用。

5.3.1 常规的需求获取方法

为了获取正确的需求信息，系统分析员常采用以下常用的需求获取方法和技术，例如建立联合分析小组、用户访谈和问题分析与确认等。

1. 建立联合分析小组

系统开发之初，分析员往往对用户的业务和术语不熟悉，用户也不熟悉计算机的处理过程。所以用户提供的需求信息，在系统分析员看来往往是零散和片面的，需要由领域专家来沟通。因此，建立一个由用户、系统分析员和领域专家组成的联合分析小组，可极大地方便系统开发人员和用户之间的沟通，对需求获取非常有利。有些学者将这种方法称为“便利的应用规约技术”（facilitated application specification techniques, FAST）。

在参与 FAST 小组的人员中，用户方的业务人员应是系统开发的主体，是理所当然的“演员”和“主角”；系统分析员好比“导演”；其他开发人员是“配角”。因此在需求获取阶段，切忌忽视用户业务人员的作用，由系统开发人员越俎代庖。

2. 用户访谈

用户访谈的对象既包括高层用户，也包括直接用户，访谈是一个深入现场、直接与更多用户交流的过程。根据用户使用该软件的功能、频率、优先级或熟练程度等方面的差异，可将他们分成若干类，对每类用户采用现场参观、个别座谈或小组会议等不同形式，了解他们对现行系统的评价和对新系统功能的期望。

由于是面对面的交流，如果系统分析员事先未做好充分的准备，很容易引起用户的反感。为了在用户访谈中尽快找到与用户的“共同语言”，愉快地进行交谈，分析员在与用户接触之前必须进行充分的准备。首先，必须对问题的背景和系统环境有全面的了解；其次，尽可能了解将要进行交谈的用户的个性特点及任务状况；最后，事先准备好提问的问题，并在交流时遵守循序渐进、逐步逼近的原则，切不可急于求成。下面举例说明。

[例 5.1] 现需开发一个学生“选课系统”。通过用户访谈，获得了用户对该系统的期望和需求。请将相关的内容整理为问题陈述，作为今后需求建模的依据。

[解] 图 5.3 显示了“选课系统”的问题陈述。

“选课系统”的问题陈述

开发一个学生“选课系统”。通过这个系统，学生可以选课和查看成绩报告单，教授可以选择所教的课和记录学生的成绩。

学校保留原有的“课程目录”数据库系统来维护课程信息，但该系统的性能是很差的。所以新系统必须确保能及时访问旧系统中的数据。但新系统只能读取旧系统的课程信息，不能对旧系统进行更新。

每学期开始时，学生请求查看本学期开设的课程目录。有关课程的信息，包括教授名和所开设的系部等，将帮助学生做出选课决定。系统允许学生每学期选择4门课，如果学生没有选到主选的课程，还有两门备选课程可选。每门课的学生人数限3~10人。不满3人的课程将被取消。另外，每个学期有一段时间允许学生更改所选课程。学生可在该时段内访问系统并添加/删除课程。某个学生的选课一旦结束，选课系统即将此学生本学期的账单信息送到财务系统。如果在选课时某门课人数已满，学生在提交信息前会被告知。学期结束，学生可进入系统查看自己的成绩。成绩属于隐秘信息，系统必须提供额外的安全措施阻止未授权的访问。

教授应能访问系统以指定其主讲的课程，他们也需要知道是哪些学生选择了自己的课程。另外，教授也能登记学生的成绩。

图 5.3 “选课系统”的问题陈述

3. 问题分析与确认

既不要期望用户在一两次交谈中就会对目标软件的需求阐述清楚，也不能限制用户在回答问题时自由发挥。每次访谈后要及时整理，去掉错误的、无关的信息，留下和整理有用的内容，以便在下次与用户见面时进行确认。同时，准备下一次访谈时需要进一步了解的更细节的问题。如此循环，一般需要2~5个来回。

5.3.2 用快速原型法获取需求

前已提到，快速原型法可以用作一种有效的需求分析方法（见第2.2.2节）。在分析阶段，开发人员利用快速开发工具先建立一个系统原型，然后让用户参加评估并提出修改意见，进而逐步、准确地确定软件系统的外部行为和特征。

作为开发人员和用户的交流手段，快速原型可以获取两个层面上的需求。第一层为联机屏幕，用于确定屏幕及报表的版式和内容，屏幕活动的顺序，以及屏幕排版的方法。第二层用于模拟系统的外部特征，包括引用数据库的交互作用及数据操作，执行系统关键操作等。

快速原型法一般可按照以下的步骤进行：

- ① 利用各种分析技术和方法，生成一个简化的需求规格说明书。

② 对需求规格说明书进行必要的检查和修改后，确定原型的软件结构、用户界面和数据结构等。

③ 在现有的工具和环境的帮助下快速生成可运行的软件原型并进行测试和改进。

④ 将原型提交给用户评估并征求用户的修改意见。

⑤ 重复上述过程，直到原型得到用户的认可。

有学者提出了以下6个问题，可用来帮助判断是否要选择原型法：

① 需求已经建立，并且可以预见是相当稳定的吗？

② 软件开发人员和用户已经理解目标软件的应用领域了吗？

③ 问题是否可被模型化？

④ 用户能否清楚地确定基本的系统需求？

⑤ 有任何需求是含糊的吗？

⑥ 已知的需求中存在矛盾吗？

如果第一个问题得到肯定回答，就不需要采用快速原型法。否则，如其他问题得到肯定回答，就适于采用快速原型法。

为了快速开发出系统原型，必须充分利用快速开发技术或软件复用技术。但如果演示原型系统需要手工编写数千行甚至数万行代码，那显然代价太大，就没有现实意义了。

第四代开发技术（4GT）是快速原型法的常用技术，它利用第四代语言或开发工具，如数据库查询和报表语言、程序和应用软件生成器以及其他高级的非过程语言等，可使软件工程师快速地生成可执行代码。IBM公司的RAD、Borland公司的Delphi、Sybase公司的PowerBuilder、微软公司的Visual Studio以及一些CASE工具等，都是常见的例子。

近年来，随着软件构件化和软件复用技术的发展，用户利用现有的数据结构或数据库构件、软件过程构件或其他可视化构件，即可快速地装配出一个原型系统，无须了解构件内部的工作细节。

5.4 需求模型

建立分析模型是需求分析的首要任务。在本书第3.2节中，阐明了如何用SA方法来构建传统软件工程的需求分析模型，并以“教材购销系统”为例，展示了随需求分析建立起来的结构化需求模型。本节将重点介绍面向对象的需求分析模型。

5.4.1 需求模型概述

在长期的实践中，软件工程专家们提出了多种需求建模的方法，其中占主导地位的有结构化分析建模和面向对象分析建模两种。图5.4和图5.5显示了这两种分析模型的组成

结构。

1. 结构化需求模型

如图 5.4 所示, 该模型主要由 3 部分组成, 即: 包括数据流图和加工规格说明的功能模型; 主要由数据字典和 E-R 图等组成的数据模型; 由状态转换图、控制流图和控制规格说明等组成的行为模型。

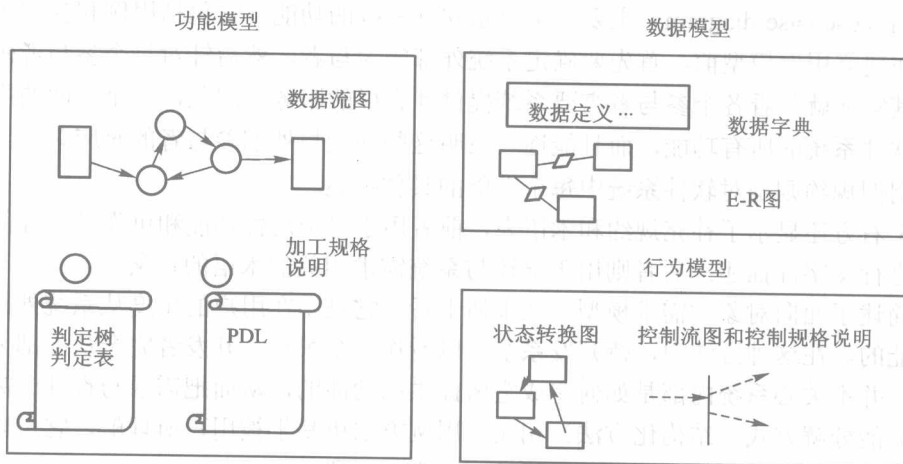


图 5.4 结构化需求模型

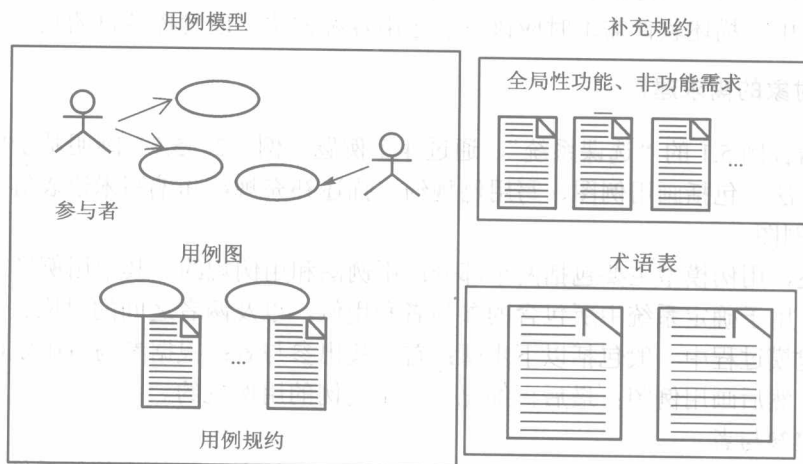


图 5.5 面向对象需求模型

由于第 3 章已对它们做过详细介绍, 这里不再重复。

2. 面向对象需求模型

采用面向对象的思想进行软件需求建模,就可得到面向对象的需求模型。图 5.5 显示的是面向对象需求模型的常用组成结构。

如图 5.5 所示,面向对象需求模型由 3 个部分组成:用例模型、补充规约和术语表,其中用例模型又包括用例图和用例规约。

用例图(use case diagram)主要用于显示软件系统的功能,它包括用例和参与者两方面的内容。在建立用例模型前,首先要确定系统外部的参与者,然后针对每个参与者确定系统的用例,其实质就是看各个参与者需要系统提供什么样的服务。因此,一个全面用例图不仅可显示软件系统的所有功能,而且能逐一表明这些功能与外部参与者的对应关系。而用例图下方的用例规约则是对软件系统中每个功能的具体描述。

图 5.5 右方还显示了补充规约和术语表,前者用于对全局性功能性和可靠性、性能等非功能性需求进行文字性描述;后者则用于描述与系统需求相关的术语的定义。

以上简述了面向对象的需求模型。从本质上看,它是站在用户的角度从系统外部来描述系统的功能的。在这种方法中,待开发系统可以看作一个黑箱,开发者完全从外部来定义系统的功能,并不关心系统内部是如何完成它所提供的功能的,从而把需求与设计分离开来。与传统的功能分解方式(结构化方法)相比,用例方法更易于被用户所理解,它可以作为开发人员和用户之间针对系统需求进行沟通的一个有效手段。

但是,并非所有的系统需求都适合用用例模型来描述。例如,编译器就很难用这种方法来表述它所处理的语言的方法和规则。此时,采用传统的 BNF 范式来表述可能更加合适。可见在实际应用中,描述软件需求时应该灵活运用各种方法,以发挥各自的长处。

5.4.2 面向对象的需求建模

本节将结合例 5.1 的“选课系统”,通过 4 个例题(例 5.2~5.5)阐明基于用例的面向对象需求建模方法,包括画用例图、写用例规约、描述补充规约和编写术语表等 4 步。

1. 画用例图

如前所述,用例模型主要包括两个部分:用例图和用例规约。其中用例图主要描述系统的外在功能,用于确定系统中所包含的参与者和用例,以及两者之间的对应关系。

在用例建模过程中一般包括以下步骤:首先找出参与者;根据参与者确定同每个参与者相关的用例;然后画用例图;最后再细化每一个用例的用例规约。

(1) 确定参与者

参与者泛指所有存在于系统外部并与系统进行交互的人、硬件或其他系统。通俗地讲,参与者主要是待开发系统的使用者。寻找参与者可以从以下问题入手:

- ① 系统开发完成之后,有哪些人会使用这个系统?
- ② 系统需要从哪些人或其他系统中获得数据?

③ 系统会为哪些人或其他系统提供数据?

④ 系统会与哪些其他系统相关联?

⑤ 系统是由谁来维护和管理的?

(2) 确定用例

找到参与者之后,就可以根据参与者来确定系统的用例了。主要是考察各参与者需要系统提供什么样的服务,或者说参与者是如何使用系统的。寻找用例时,可以针对每一个参与者从以下问题入手:

① 参与者为什么要使用该系统?

② 参与者是否会在系统中创建、修改、删除、访问和存储数据?如果是,参与者又是如何来完成这些操作的?

③ 参与者是否会将外部的某些事件通知该系统?

④ 系统是否会将内部的某些事件通知该参与者?

第一步,确定参与者。系统的参与者可以是人、其他软件系统或硬件设备。当参与者是人时必须是角色(role),不同的人也可能是同一种角色。例如,新生和老生是不同的参与者吗?假设你回答“是”,则下一步要确定参与者怎样和系统交互。如果新生和系统的交互方式与老生不一样,即作为不同的角色,则新生和老生不是同一个参与者;但如果他们和系统的交互操作是一样的,则应该视作同一个参与者。另一个例子,就是同一个人扮演不同的角色,例如课程注册系统中的助教。助教可以选课和任课(即扮演学生和教授两种角色),故同一个人可以表示为学生和教授两个参与者。

第二步,确定用例。确定用例时要解决的问题之一是用例描述的详细程度,即用例要多大(或多小)才合适。对此并无标准的答案。通常的规则是:用例应该典型地描绘系统功能中某个从开始到结束的过程,并且给参与者提供某些信息。例如在“选课系统”中,学生选择课程,然后增加到课程表中去。这算两个用例还是一个呢?由于该功能描绘了从开始到结束的一个完整的过程,因此应视为一个用例。

(3) 绘制和检查用例图

用例和参与者确定后,就可以据此画出用例图了。用例图绘制完后,为了避免差错,还需对画好的用例图进行以下的检查:

① 每个用例至少应该涉及一个参与者。如果存在不与参与者进行交互的用例,就应该考虑将其并入其他用例,或者检查是否有参与者被遗漏。反之,如果发现某参与者未与任何用例相关联,就应考虑确定一个新的用例,或者把该参与者作为多余的元素删除。

② 参与者和用例的名称是否符合统一的命名约定和风格。例如,参与者的名称一般采用名词,用例名称一般采用动宾词组等。

③ 用例建模通常属于团队开发。为使用例模型易于理解,应尽可能以图形的方式进

行可视化建模，借以增强团队的沟通能力。不同的人员对于同一用例模型的理解应完全一致。

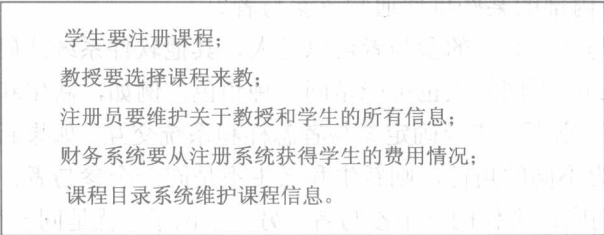
还应指出，对于同一个系统，不同的人可能有不同的抽象结果，因而得到不同的用例模型。如果可能，应在多个用例模型方案中选择一种最佳（或较佳）的结果，一个好的用例模型，应该容易被与软件相关的各类人员所理解。

下面请看一个例子。

[例 5.2] 续例 5.1，先为“选课系统”的用例图确定用例和参与者，然后画出用例图。

[解] 按照基于用例的面向对象需求建模步骤，首先要找出“选课系统”的参与者；再确定同每个参与者相关的用例。具体步骤如下。

① 确定参与者。根据对选课系统“问题陈述”文档的分析，可以创建以下 5 种参与者，即学生、教授、注册员以及财务系统和课程目录系统。图 5.6 列出了他们需要的服务。

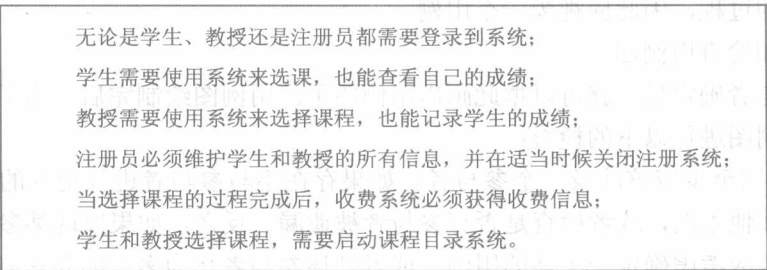


学生要注册课程；
教授要选择课程来教；
注册员要维护关于教授和学生的所有信息；
财务系统要从注册系统获得学生的费用情况；
课程目录系统维护课程信息。

图 5.6 “选课系统”的参与者及所需要的服务

② 确定“选课系统”的用例。为满足这些参与者的需要，可考虑生成以下用例：系统登录，注册课程，查看报告，选择所教课程，提交成绩，维护教授信息，维护学生信息，关闭注册。

图 5.7 显示了这些用例与参与者的相对应关系。



无论是学生、教授还是注册员都需要登录到系统；
学生需要使用系统来选课，也能查看自己的成绩；
教授需要使用系统来选择课程，也能记录学生的成绩；
注册员必须维护学生和教授的所有信息，并在适当时候关闭注册系统；
当选择课程的过程完成后，收费系统必须获得收费信息；
学生和教授选择课程，需要启动课程目录系统。

图 5.7 “选课系统”生成的用例

③ 绘制用例图。综上所述，“选课系统”用例图可绘制如图 5.8 所示。

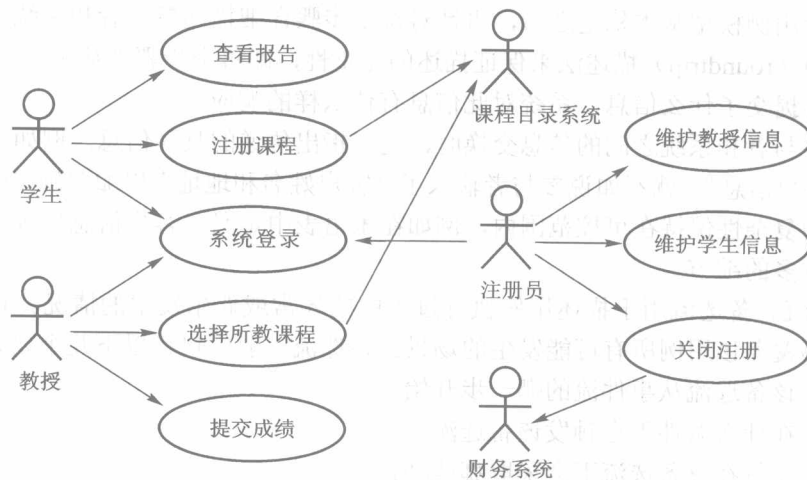


图 5.8 “选课系统”用例图

2. 写用例规约

前已指出，用例模型是由用例图和用例规约组成的。用例图从总体上描述了系统所能提供的各种服务，让我们对系统的功能有一个大致的认识。用例规约则用来描述每一个用例的功能，一个用例对应一个用例规约，用来描述用例的细节。

(1) 用例规约的内容

用例规约文档一般应包含以下内容：

- 简要说明 (brief description)：简要介绍该用例的作用和目的。
- 事件流 (flow of event)：包括基本流和备选流，表示出所有可能的活动及流程。
- 特殊需求 (special requirement)：描述与该用例相关的非功能性需求（包括性能、可靠性、可用性和可扩展性等）和设计约束（所使用的操作系统和开发工具等）。
- 前置条件 (pre-condition) 和后置条件 (post-condition)。

① 简要说明主要用文本方式表述。为了清晰地描述事件流，也可以用 UML 图（状态图、活动图或时序图）来辅助说明，或者在用例中粘贴用户界面和流程的图形化显示方式，使表达更加简洁、明了。例如，活动图有助于描述复杂的决策流程，状态转换图有助于描述与状态相关的系统行为，时序图适合于描述基于时间顺序的消息传递，等等。

② 基本流。所谓基本流，是指该用例最正常的一种场景。在基本流中，系统执行一系列活动来响应参与者提出的服务请求。一般用以下格式来描述基本流：

- 每个步骤都需要用数字编号，以清楚地标明步骤的先后顺序。
- 每个步骤的主要内容用一句简短的标题来概括，使阅读者可通过浏览标题快速地了解用例的主要步骤。在用例建模的早期，只需要描述到事件流步骤标题这一层，以免过早地陷入用例描述的细节。

• 当整个用例模型基本稳定之后，再针对每个步骤详细描述参与者和系统之间的交互。通常采用双向（roundtrip）描述法来保证描述的完整性，即每个步骤都从正反两方面来描述：参与者向系统提交了什么信息，系统对此信息有什么样的响应。

在描述参与者和系统之间的信息交换时，建议指出传递的具体信息。例如，如果说参与者输入了“客户信息”，就不如说参与者输入了“客户姓名和地址”更加明确。还可利用术语表，使用例的复杂性保持在可控范围内，例如在术语表中定义“客户信息”等内容，使用例不至于陷入过多的细节。

③ 备选流。备选流用于描述用例执行过程中的异常或偶尔发生的情况。它和基本流组合起来，能够覆盖该用例所有可能发生的场景。备选流一般应包括以下几个要素：

- 起点：该备选流从事件流的哪一步开始。
- 条件：在什么条件下会触发该备选流。
- 动作：系统在该备选流下会采取哪些动作。
- 恢复：该备选流结束之后，该用例应如何继续执行。

备选流的描述格式可与基本流一致，即需要编号并以标题概述其内容，编号前可加一字母前缀“A”（alternative），以示与基本流相区别。

④ 特殊需求。特殊需求通常是非功能性需求，它为一个用例所专有，但不适合在用例的事件流文本中进行说明。特殊需求的例子包括法律或法规方面的需求、应用程序标准和所构建系统的质量属性（包括可用性、可靠性、性能或支持性需求等）。此外，其他一些设计约束，如操作系统及环境和兼容性需求等，也可以在此部分记录。

需要注意的是，这里记录的是专属于该用例的特殊需求；对于一些全局的非功能性需求和设计约束，它们并不是该用例所专有的，应把它们记录在补充规约中。

⑤ 前置和后置条件。前置条件是执行用例之前必须存在的系统状态，后置条件是用例执行完毕后系统可能处于的一组状态。

（2）用例规约示例

[例 5.3] 续例 5.2，写出“选课系统”的用例规约。

[解] 根据上文的描述，不难写出“选课系统”的用例规约，如图 5.9 所示。

选课用例的用例规约

1. 简要说明

本用例允许学生选本学期提供的课程。在学期开始时允许在“选课系统”中添加/删除某门课程阶段，学生可以修改或删除选择的课程。课程目录系统提供了当前学期开设的所有课程的列表。

2. 事件流

2.1 基本事件流

用例开始于学生选择课程，或修改已存在的课程表。

- ① 系统要求学生指出要执行的操作（创建、修改或删除课程表）。
- ② 一旦学生提供了所需要的信息，以下的一个子事件流将被执行：
 - 如果选择的是“创建课程表”，创建课程表子事件流将被执行；
 - 如果选择的是“修改课程表”，修改课程表子事件流将被执行；
 - 如果选择的是“删除课程表”，删除课程表子事件流将被执行。

2.1.1 创建课程表

- ① 系统从课程目录系统中得到可选择的课程列表，并将列表显示给学生。
- ② 学生从课程列表中选择 4 门主要的和两门备选的课程。
- ③ 一旦学生确定了选课情况，系统为他创建一个包含所选课程的课程表。
- ④ 执行提交课程表子事件流。

2.1.2 更新课程表

- ① 系统得到并显示学生当前的课程表（例如用于本学期的课程表）。
- ② 系统从课程目录系统中得到可选择的课程列表，并将列表显示给学生。
- ③ 学生可以通过删除或添加课程来修改课程选择。学生从有效的课程列表中选择课程并添加。也可以从现存的课程表中选择任何课程将其删除。
- ④ 一旦学生决定了选课情况，系统用学生所选的课程更新课程表。
- ⑤ 执行提交课程表子事件流。

2.1.3 删除课程表

- ① 系统得到并显示学生当前的课程表（例如用于本学期的课程表）。
- ② 系统提示学生确认删除课程表。
- ③ 学生确认删除。
- ④ 系统删除课程表。如果课程表包含“已登记”的课程，学生必须从课程表中删除。

2.1.4 提交课程表

对于课程表中未标记“已登记”的已选课程，系统检验学生是否满足必要的预备条件，并且课程未被选满、没有时间冲突，系统将学生添加到选择的课程中，这个课程在课程表中标明“已登记”。系统保存课程表。

2.2 备选事件流

2.2.1 保存课程表

在任何时候，学生都可以选择保存课程表而不提交它。如果这样，提交课程表的步骤将被以下步骤所取代：课程在课程表中的标识不是“已登记”而是“已选”，系统保存课程表。

2.2.2 不满足预备条件，课程人数已满或课程时间冲突

如果在提交课程表子事件流中，系统确定学生不满足必要的预备条件，或者选择的课程已满，或课程有时间冲突，将显示一条错误信息。学生可选择其他课程，此用例继续；或取消操作，这时本用例重新开始。

2.2.3 未找到课程表

如果在修改或删除课程表子事件流中，系统无法返回学生的课程表，将显示错误消息。学生确认错误，这时本用例重新开始。

2.2.4 无法访问课程目录系统

如果系统无法与课程目录系统取得联系，系统将显示错误消息。学生确认错误，本用例终止。

2.2.5 课程注册系统被关闭

当用例开始时，如果确定用于本学期的注册系统已被关闭，显示一条消息给学生，本用例终止。学生不能在注册系统关闭后选课。

2.2.6 删除被取消

如果在删除课程表子事件流中，学生决定不删除课程表，本用例重新开始。

3. 特殊需求

无

4. 前置条件

本用例开始前学生必须已经登录该系统。

5. 后置条件

如果用例成功，学生的课程表被创建、修改或删除，否则系统状态不变。

图 5.9 “选课系统”的用例规约

(3) 用例模型的检查

为了发现用例模型的错漏，可以从以下几个方面来进行检查：

① 功能需求的完备性。现有的用例模型是否完整地描述了系统功能，这也是判断用例建模工作是否结束的标志。如果发现还有系统功能没有被记录在现有的用例模型中，那么就需要抽象一些新的用例来记录这些需求，或是将它们归纳在一些现有的用例之中。

② 模型是否易于理解。用例模型最大的优点就在于它应该易于被与软件相关的各类人员所理解，因而用例建模最主要的指导原则就是它的可理解性。用例的粒度、个数以及模型元素之间的关系复杂程度都应该由该指导原则决定。

③ 是否存在不一致性。系统的用例模型是由多个系统分析员协同完成的，模型本身也是由多个文档所组成的，所以要特别注意不同文档之间是否存在前后矛盾或冲突的地方，避免在模型内部产生不一致性。不一致性会直接影响需求定义与分析的准确性。

④ 避免二义性语义。好的需求描述应该是无二义性的，即不同的人对于同一需求的理解应该是一致的。在用例规约的描述中，应该避免定义含义模糊的需求，即无二义性。

3. 描述补充规约

补充规约用于记录在用例模型中不易表述的系统需求。请看下面的例子。

[例 5.4] 续例 5.3，写出“选课系统”的补充规约。

[解] 图 5.10 显示了“选课系统”的补充规约。

4. 编写术语表

术语表主要用于定义软件开发项目特定的术语，它有助于开发人员对项目中所用的术语有统一的理解并能正确地使用，它也是后续阶段中进行对象抽象的基础。

选课系统的补充规约

1. 目标

本文档的目的是定义选课系统的需求。本补充规约列出了不便于在用例模型的用例中获取的系统需求。它和用例模型一起记录关于系统的一整套需求。

2. 范围

本补充规约适用于选课系统，除定义了在许多用例中所共有的功能性需求以外，还定义了系统的非功能性需求，例如，可靠性、可用性、性能和可支持性等（功能性需求在用例规约中定义）。

3. 参考

无。

4. 功能

多个用户必须能同时执行操作。

如果某个学生所建的课程表中包含人数已满的课程，必须通知这位学生。

5. 可行性

桌面用户界面应与 Windows 98/2000/XP 兼容。

6. 可靠性

选课系统在每周 7 天、每天 24 小时内都应是可用的。宕机的时间应少于 10%。

7. 性能

① 在任意既定时刻、系统最多可支持 2 000 名用户同时使用中央数据库，并在任意时刻最多可支持 500 名用户同时使用本地服务器。

② 系统将能在 10 秒种内提供对遗留课程目录数据库的访问。

注意：基于风险分析的原型发现遗留课程目录数据库在没有利用中间层处理能力的前提下，无法满足性能上的需求。

③ 系统必须能够在 2 分钟内完成所有事务的 80%。

8. 可支持性

无。

9. 安全性

① 系统必须能防止某一学生修改他人的课程表以及某一教授修改其他教授所开设的课程。

② 只有教授能输入学生的成绩。

③ 只有注册员能更改学生与教授的信息。

10. 设计约束

该系统应与现有遗留系统：课程目录系统（是 RDBMS 数据库）集成在一起。

系统必须提供基于 Windows 桌面的接口。

图 5.10 “选课系统”的补充规约

[例 5.5] 续例 5.4，编写“选课系统”的术语表。

[解] 图 5.11 给出“选课系统”的术语表。

选课系统的术语表

1. 简介

这份文档是用来对一些术语进行定义的,同时将用例说明或其他文档中读者不太熟悉的术语进行解释性的描述。通常来说,这份文档对一些数据信息进行定义,从而使得用例规约和其他的文档显得简洁、易懂。

2. 定义

这份术语表包含了选课系统中核心概念的定义。

2.1 课程: 大学提供的某一门课。

2.2 开设课程: 某一课程的具体安排情况,包括一周上课的天数、时间和教授。

2.3 课程目录: 大学所开设的所有课程的完整目录。

2.4 教员: 所有在此大学内任教的教授。

2.5 财务系统: 用来处理收费信息的系统。

2.6 成绩: 学生某门课程的成绩。

2.7 教授: 在该大学任教的个人。

2.8 报告: 一个学生在某一学期内所有已修课程的成绩。

2.9 花名册: 在某门课程中登记的所有学生。

2.10 学生: 在该大学某一班级注册的个人。

2.11 课程表: 学生在当前学期所选的课程。

2.12 总成绩单: 某个学生所有课程成绩的历史记录。

2.13 注册员: 对课程注册和课程开设负责的大学管理员。

图 5.11 “选课系统”的术语表

5. 调整用例模型

在一般的用例图中,只表述参与者和用例之间的关系,即它们之间的关联。除此之外,还可以描述参与者与参与者之间的泛化(**generalization**)、用例和用例之间的包含(**include**)、扩展(**extend**)和泛化关系。利用这些关系来调整已有的用例模型,把一些公共的信息抽取出来进行重用,可使用例模型更易于维护。但是在应用中要小心选用这些关系,一般来说这些关系都会增加用例和关系的个数,从而增加用例模型的复杂度,而且一般都是在用例模型完成之后才对用例模型进行调整,所以在用例建模的初期不必急于抽象用例之间的关系。

用例模型建成之后,可以对用例模型进行检查,看是否可以进一步简化用例模型、提高重用程度、增加模型的可维护性。主要可以从以下方面进行检查:

① 用例之间是否相互独立?如果两个用例总是以同样的顺序被激活,可能需要将它们合并为一个用例。

② 多个用例之间是否有非常相似的行为或事件流?如果有,可以考虑将它们合并为一个用例。

③ 用例事件流的一部分是否已被构建为另一个用例？如果是，可以让该用例包含另一个用例。

④ 是否应该将一个用例的事件流插入另一个用例的事件流中？如果是，利用与另一个用例的扩展关系来建立此模型。

5.5 软件需求描述

软件需求规格说明书简称 SRS，是软件开发人员在分析阶段需要完成的用于描述需求的文档。

早在 3.1.1 节就已指出，在分析阶段要编写 SRS，包括引言、信息描述、功能描述、行为描述、质量保证、接口描述和其他描述等内容。

引言主要叙述在问题定义阶段确定的关于软件的目标与范围，简要介绍系统背景、概貌、软件项目约束和参考资料等。

需求规格说明书的主体描述软件系统的分析模型，包括信息描述、功能描述和行为描述。这部分内容除了可用文字描述外，也可以附上各种图形模型，如用例图、E-R 图、DFD 和 CFD 等。

信息描述给出对软件所含信息的详细描述，包括信息的内容、关系、数据流向、控制流向和结构等。根据系统所选用的不同分析方法（结构化分析或面向对象分析），可以用前面介绍的工具描述软件涉及的数据的定义和系统的信息逻辑模型。

功能描述是对软件功能需求的说明，包括系统功能划分、每个功能的处理说明、限制和控制描述等。对软件性能的需求，包括软件的处理速度、响应时间和安全限制等内容，通常也在此叙述。

行为描述包括对系统状态变化以及事件和动作的叙述，据此可以检查外部事件和软件内部的控制特征。

质量保证阐明在软件交付使用前需要进行的功能测试和性能测试，并且规定源程序和文档应该遵守的各种标准。此部分说明文字用于检查所交付的软件是否达到了 SRS 的规定。这可能是 SRS 中最重要的内容，但在实际工作中却容易被忽略，值得引起注意。

接口描述包括系统的用户界面、硬件接口、软件接口和通信接口等的说明。

其他描述阐述系统设计和实现上的限制，系统的假设和依赖等其他需要说明的内容。

可为上文的“选课系统”编写一个 SRS。为节约篇幅，这里就省略了。

5.6 需求管理

需求管理（requirements management, REQM）是随着软件需求的发展而成长起来的管理技术。无论是贯穿于整个软件生存周期的需求工程，还是随需求分析建模而产生的用例模型，在定义需求的时候总伴有许多可变因素。因此，需求应该具有弹性的结构，使之能适应可能的变更。一旦其中有某些需求发生变化，就能确定它可能带来的影响，进而制定出相应的

策略。

5.6.1 需求管理的内容

1. 需求管理的特定实践

需求管理包含 5 个特定实践，如图 5.12 所示。现简介如下。

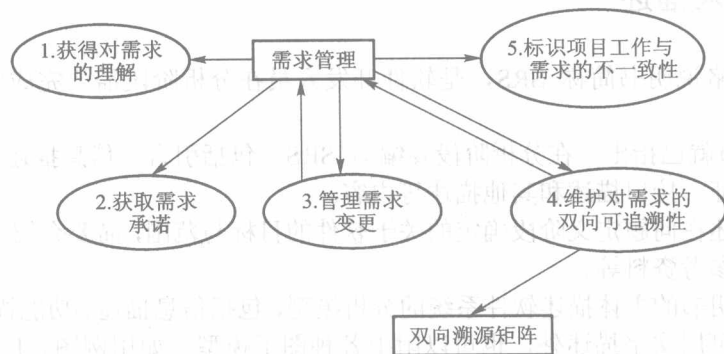


图 5.12 需求实践示意图

① 获得对需求的理解。在初步整理需求的基础上，项目小组和用户代表通过初步的分析讨论，对当前项目的需求达成共识，并在需求列表中作相应记录。

② 获取需求承诺。通过项目参与者的书面承诺，建立各方或各项工作的基准。

③ 管理需求变更。维护变更历史，为调整与控制提供数据。

④ 在需求变更后维护对需求的双向可追溯性。从软件可维护性的角度提出管理要求。

⑤ 标识项目工作（包括计划和产品）与需求的不一致性。若发现不一致性，即启动纠正措施。

2. 需求管理的管理流程

上述 5 个特定实践，可归结为以下 3 项活动，即需求确认、需求跟踪和需求变更。

(1) 需求确认

包括图 5.12 中第 1、2 两个特定实践。由开发方和客户共同对主要需求文档“软件需求规格说明书”进行评审，双方达成共识后作出书面承诺，使需求文档具有商业合同效力。

由此可见，需求确认实际上包含了两个重要工作：需求评审和需求承诺。其中需求承诺是双方对通过正式评审后的“软件需求规格说明书”作出的共同承诺。承诺书的格式如下：

本“软件需求规格说明书”是建立在双方对需求的共同理解基础之上的，我们同意后续的开发工作根据该“软件需求规格说明书”进行。如果需求发生变化，我们将按照“需求变更流程”执行，即需求的变更将导致双方重新协商成本、资源和进度等。

项目经理签字

客户或客户代表签字

该承诺书将附在“软件需求规格说明书”后，一同存档保存。

(2) 需求跟踪

包括图 5.12 的第 4、5 两个特定实践，即维护对需求的双向可追溯性和标识项目工作与需求的不一致性。

为了有效地检验最终软件产品能否满足所有需求，对项目的需求要进行跟踪管理。跟踪的目的，是建立与维护“需求—设计—编程—测试”之间的一致性，确保所有工作成果都符合用户需求。为此可采用需求大纲中的需求跟踪矩阵，对每个需求追踪到实现该需求的设计、编码以及测试案例，从而验证该软件产品是否实现了所有需求，是否对所有需求进行过测试。

5.6.2 需求变更控制

需求变更要进行控制，严格防止因失控而导致项目混乱，出现重大的风险。

1. 需求变更的利弊

随着项目的进展，用户和开发方对需求的了解越来越深入，原先的需求文档很可能存在错误或不足。另一方面，市场会发生变化，原先的文档也可能跟不上当前的市场需求。可见需求变更总是不可避免的，有些是为了修正缺陷，有些属于增强功能。

对项目开发小组而言，变更需求通常意味着要调整资源、重新分配任务，并修改前期的工作成果，有时要付出较大的代价。如果动不动就变更需求，某些项目也许永远不能按时完成。为此，需求变更必须遵守利大于弊的原则，并做到：

① 为避免出现失控等风险，对纳入基线以前的需求文档，可通过正常的 check-in 和 check-out 进行更改。而纳入基线以后的需求文档，更需按照预定的变更控制规程，确保快速、顺利和有序地进行变更。

② 遵照如图 5.13 所示的需求变更流程来处理。下文将具体介绍这一流程。

2. 需求变更的流程

需求变更通常按变更申请→审批→更改→重新确认的流程进行。

(1) 变更申请

此时的状态为“请求变更”。首先由申请人提交需求变更申请书，其内容应该包括：

① 变更源类型。指引起变更的原因类型，可分为需求变更、设计变更、代码优化、用户文档优化和计划变更等。

② 变更优先级。依据变更的重要性、紧迫性和对关键业务的影响程度，以及对系统安全性和稳定性的影响程度，可分为 critical、high、middle 和 low 等 4 级。

③ 变更标志。分为新增、修改和删除。

④ 变更影响分析。包括变更影响的工作产品和负责人，对工作量 and 进度的影响，发生风险的可能性与影响程度，以及需要回测的范围。

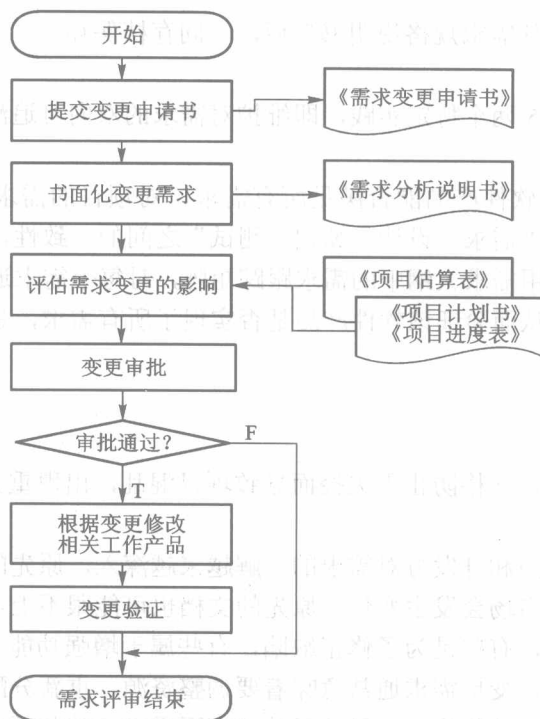


图 5.13 需求变更流程图

⑤ 可能影响的工作产品。包括项目计划、需求文档、概要设计文档、详细设计文档、源代码和程序、测试计划和测试案例以及用户文档。

上述申请书应由项目经理进行评估，其内容包括：

① 该需求变更在技术上是否可行。

② 对工期、成本、质量的影响。首先评估单个模块工期的影响，即实现该需求变更需要的成本和工作量；然后评估实现该需求变更对整体工期工作量和成本的影响。

(2) 变更审批

按照影响的大小由不同的负责人审批。

① 对影响小的变更，由项目经理直接审批。

② 对影响大的变更，提交软件变更控制委员会（Software Change Control Board, SCCB）审批。

③ 项目 SCCB 仍无法决定的变更，再提交高层 SCCB 决定。

所谓影响大的变更一般包括下列情况：

① 变更影响的模块数超过 10 个或超过 50%，或者可能影响软件系统的框架。

② 变更会影响对客户的承诺。

③ 变更会带来“高”或者“高中”程度的风险。

如果审批请求未通过，则该变更请求结束。

(3) 变更修改

如果需求变更已审批通过，应指定相关的责任人对产品进行修改，并指定人员对更改后的产品进行审核。还应在产品列表中记录具体修改的产品名称、修改描述和是否完成修改的状态。包括：

- ① 应及时更新相应的需求大纲和需求分析说明。
- ② 如果影响项目计划的内容，修改项目计划，以反映需求的变更。
- ③ 如果影响到概要设计文档、详细设计文档、源代码和程序、测试计划和测试案例或者用户文档，它们也需要被及时更新。
- ④ 如果影响到测试，还需要进行回归测试。
- ⑤ 如果对文档进行修改，需在修改历史表格中注明修改人、修改时间以及修改原因。
- ⑥ 如果对原文件修改过大，必要时项目经理可以重新组织工作产品的评审。
- ⑦ 如果对代码进行修改，需要导出编译申请表，通知编译和测试。

(4) 变更关闭

如果修改后不需要进行测试，则当所有产品全部修改完成时，由最后完成修改的人关闭该变更。如果变更修改后提交测试，则由测试人员负责该变更是否最后关闭：

- ① 如果测试未通过，则返回修改者继续修改。
- ② 如果所有工作产品全部修改完成，并且测试通过，关闭该变更。

图 5.14 为需求变更的状态转换图，从中可看到各种需求变更状态的转换。

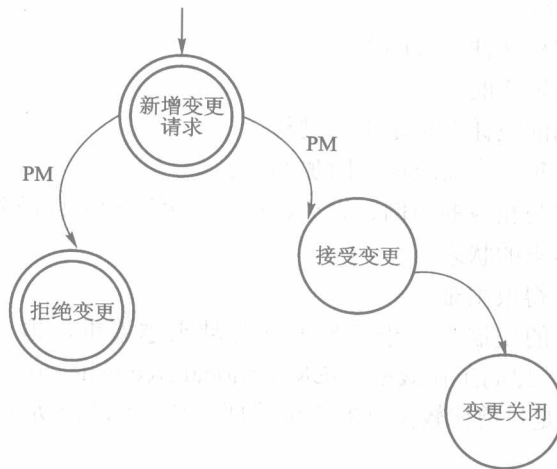


图 5.14 需求变更状态转换图

3. 需求变更的数据项

为了确切记录需求变化，还需登记如表 5.1 所示的变更数据列表。

表 5.1 需求变更数据列表

数据项名称	定 义
项目名称和 ID	变更所在项目的名称和 ID
变更阶段	需求阶段、设计阶段、编码、测试和验收阶段。不同阶段的需求变更请求对整个项目开发的影响也不同
变更优先级	每个变更的相对重要性
变更标志	变更的状态
变更原因描述	简单描述提出变更的原因
变更内容描述	对变更的内容进行简单描述
相关的变更请求	是否有相关的变更请求，如果有，指定相关的变更请求
变更的状态信息	包括变更请求人、变更批准人、当前负责人、变更关闭人、请求日期、审批日期、期望解决日期以及关闭日期
变更影响分析	基于受影响工作产品对变更的影响进行分析
变更处理信息	所影响的工作产品列表以及各工作产品对变更的处理状态

5.6.3 需求管理工具

在软件规模很小的时候，人们采用文档文件的方式来存储软件需求规格说明书和其他文档。在一些小规模的软件系统开发中，人们也还这样做。但是，随着各种计算机应用系统越来越复杂，软件规模也越来越庞大。这时传统的基于文档文件存储需求的方式越来越显露出它的局限性，主要体现在：

- ① 手工维护大量文档文件十分困难。
- ② 很难保持文档与现实的一致。
- ③ 通知受变更影响的设计人员是手工过程。
- ④ 不太容易做到为每一个需求保存附加的信息。
- ⑤ 很难在功能需求与相应的用例、设计、代码、测试和项目任务之间建立联系链。
- ⑥ 很难跟踪每个需求的状态。
- ⑦ 异地协作开发变得很困难。

随着软件工程技术的发展，需求管理的任务越来越繁重，迫切需要研制需求管理工具来自动化地管理需求，提高工作效率。IBM Rational RequisitePro、Telelogic DOORSreg 和 Borland CaliberRM 等都是目前比较流行的需求管理工具，可以帮助开发团队有效地管理软件需求。

5.7 需求建模示例

本节以一个案例的形式，为“网上购物系统”建立面向对象的用例模型。第 5.7.1 节介

绍网上购物系统的问题陈述。5.7.2 节~5.7.4 节依次阐明如何建立该系统的用例模型、补充规范和术语表,可供读者学习与参考。

5.7.1 问题陈述

当今,网上购物已成为一种时尚。本示例作为 Web 应用的一例,主要为普通购物用户和管理员服务。

普通购物用户在使用本系统的购物功能前,必须先注册账号。在注册页面中填写个人信息,如使用本系统的账号、密码和联系地址等。在提交表单和完成注册后,系统将保存信息,以方便管理员管理用户信息和联系用户。

如果用户已经在系统中注册过,可以在登录页面输入账号和密码。如果信息正确,用户就可以购物,否则只能做一般的页面浏览。

进入系统后,用户也可选择维护自己的信息,比如修改账号、密码和联系地址等。如果直接进行购物,系统可让用户首先浏览商品信息,使之对商品的数量和种类有一个大概的了解。如果用户对某件商品感兴趣,就可以选择特定商品查看其详细信息,接着选择将商品加入购物车,或继续查看其他商品。当购物结束时,用户首先要浏览一下已经保存在购物车中的商品项目,包括数量、单价及总价。这时用户可以更改任何已保存在购物车中的商品数量。如果确定要购买购物车内的商品,系统即生成一份订购商品的订单(包括所有商品的名称、单价、小计和总价),然后由用户填写包括用户姓名、家庭地址、信用卡号码和电子邮件地址等信息,并提交订单。以后,系统自动将用户信息、信用卡信息和购物总价发送到银联系统,由银联系统验证信用卡信息并执行扣款,并将银联系统操作成功与否的信息返回到系统。系统根据银联系统的操作结果,向用户发送 E-mail,提示用户操作成功与否的消息。如果扣款成功,就与物流系统连接,安排给用户派送购买的商品。

管理员进入系统时,首先要输入口令。如果检查通过,就可以对系统中的信息进行维护和管理,其主要工作包括:

① 管理用户信息,包括启用或冻结用户账号。当有些用户有不正常操作时,如填写订单时使用不存在的信用卡号码,应将此用户账号冻结,但管理员无权修改客户信息。

② 管理系统中的商品信息,例如有新的商品时,管理员可向系统中添加此商品。当商品的价格或规格发生变化时,管理员也可以对它们作修改,使用户及时了解商品的最新情况。若某件商品没有存货或不再出售时,管理员可删除系统中的此项商品记录。

③ 管理客户订单。及时获得客户的资料(资料中有电子邮件地址),以便与客户联系。

要求系统对数据库的存取速度要尽量快,并保证系统在配置完成以后 24 小时都可用。还要求系统有较高的安全性,当生成订单时,用户的信用卡号码要在网上传输,所以必须提供额外的安全措施。

5.7.2 用例模型

用例模型包括用例图和用例规约两方面的内容。本节将用两个例子（例 5.6 和例 5.7）分别介绍它们。

1. 用例图

[例 5.6] 画出“网上购物系统”的用例图。

[解] 根据“网上购物系统”的问题陈述，可画出该软件的用例图，如图 5.15 所示。

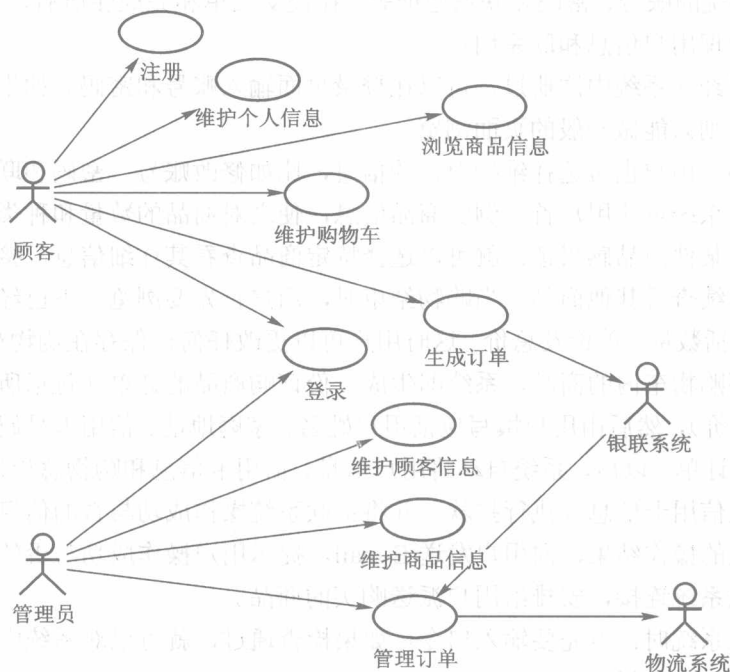


图 5.15 “网上购物系统”的用例图

由图 5.15 可见，该用例图包含 9 个用例、4 个参与者。

用例的编号和名称是：1.注册，2.登录，3.维护个人信息，4.浏览商品信息，5.维护购物车，6.生成订单，7.维护顾客信息，8.维护商品信息，9.管理订单。

参与者的名称是：顾客，管理员，银联系统，物流系统。

2. 用例规约

[例 5.7] 试编写“网上购物系统”中注册、维护个人信息、维护购物车、生成订单和管理订单 5 个用例的用例规约。

[解] 如图 5.15 所示，“网上购物系统”有 9 个用例。本例将写出其中 5 个用例的用例

规约, 如图 5.16 所示。

1. 注册

1.1 简要说明

本用例用于向顾客提供注册功能。每位顾客必须注册后才能购物。注册信息包括使用本系统的账号、密码、联系地址和电子邮件等。注册完成后, 系统保存这些信息, 以方便管理员管理及联系用户。

1.2 事件流

1.2.1 基本流

当用户进行注册时, 开始执行以下基本流:

- ① 系统要求用户填写个人信息, 包括使用本系统的账号、密码、联系地址和电子邮件等。
- ② 用户填写个人信息。
- ③ 系统验证用户所填写的信息的格式和内容。
- ④ 保存该用户信息。

1.2.2 备选流

1.2.2.1 用户信息验证错误

如果系统检测到用户输入的信息格式或内容有错, 例如账号中含有非法字符、输入密码和确认输入密码不一致等, 会给予错误提示, 并清空填写错误的文本框, 要求用户重新输入。

1.2.2.2 用户信息保存失败

如果系统发现数据库中已经保存了同样账号的用户记录, 会向用户报告保存失败的错误信息, 并使页面跳回注册页面, 要求用户修改注册信息。

1.3 特殊需求

无。

1.4 前置条件

用户必须首先访问网上购物的主页, 然后单击注册。

1.5 后置条件

如果该用例成功, 系统数据库中 will 增加一条该用户的信息。否则, 系统维持原状。

1.6 扩展点

无。

2. 维护个人信息

2.1 简要说明

本用例用于给顾客维护个人信息。包括修改本人的账号、密码和联系地址等信息。

2.2 事件流

2.2.1 基本流

当顾客查看并修改个人信息时, 开始执行以下基本流:

- ① 系统返回给当前顾客在系统数据库中目前存储的个人信息。
- ② 顾客可以对本人信息的一项或几项进行修改。
- ③ 顾客向系统提交修改后的个人信息。

2.2.2 备选流

2.2.2.1 顾客输入的新信息验证错误

如果系统检测到顾客输入的信息格式或内容有错（如输入新密码和确认输入新密码不一致等），会向顾客给予错误提示，并清空填写有误的文本框，要求用户重新输入或取消修改的操作。

2.3 特殊需求

无。

2.4 前置条件

顾客必须首先登录系统，然后才能进入本用例。

2.5 后置条件

如果本用例成功，顾客在系统数据库中的个人信息会被修改。否则，系统维持原状。

2.6 扩展点

无。

3. 维护购物车

3.1 简要说明

本用例用于维护正在购物顾客的购物车。凡是登录到系统的顾客，系统会为他产生一个购物车。在生成顾客订单之前，购物车里的商品种类和数量都由顾客本人维护。

3.2 事件流

3.2.1 基本流

当顾客想要维护自己的购物车时，开始执行以下基本流：

① 顾客请求查看购物车，系统显示当前购物车中的信息。

② 一旦顾客确定了对购物车的操作，则转向以下子流程：

如果顾客选择了“删除购物车中的商品”，则“删除商品”子流程开始执行；

如果顾客选择了“修改购物车中的商品”，则“修改商品”子流程开始执行。

3.2.1.1 删除商品

当顾客想要删除购物车中的某商品时，选中该商品进行删除。

3.2.1.2 修改商品

当顾客想要修改购物车中的某商品时，选中该商品进行修改，如商品的数量。

3.2.2 备选流

3.2.2.1 修改商品数量失败

当顾客查看购物车时，可能希望对已选商品的数量进行修改。如果修改的数值大于此时商品库存数量，或者超过该商品的限量出售值，系统将向顾客返回修改失败的信息，并告知顾客失败的原因。

3.3 特殊需求

无。

3.4 前置条件

使用本用例的顾客必须先登录到该系统中。

3.5 后置条件

无。

3.6 扩展点

无。

4. 生成订单

4.1 简要说明

本用例用于生成顾客订单。它要求顾客填写个人信息和银行账号信息，获得当前购物车中顾客选定的商品信息（包括所有商品 ID、名称、单价、数量和总价），然后将生成的订单发送给银联系统。

4.2 事件流

4.2.1 基本流

当顾客确定要购买购物车中的商品时，开始执行以下基本流：

- ① 顾客填写用户名、家庭地址和银行账号等必要信息，系统生成具有顾客信息的订单。
- ② 系统从当前购物车中获取购物车中的商品信息，计算出所有商品总价并填入订单。
- ③ 系统生成了具有顾客信息和商品信息的订单。
- ④ 系统将订单发送给银联系统处理。

4.2.2 备选流

4.2.2.1 提交订单失败

如果顾客提交了信息不完整的订单，则系统将向顾客返回错误信息，并要求顾客重新检查并填写订单信息或者取消该订单。

4.2.2.2 顾客取消订单提交

如果顾客取消提交订单，则系统销毁该订单。

4.3 特殊需求

无。

4.4 前置条件

顾客确定要购买购物车中的商品。

4.5 后置条件

如果该用例成功，则把生成好的完整订单发送给银联系统。否则，该系统维持原状。

4.6 扩展点

无。

5. 管理订单

5.1 简要说明

本用例是管理员用来管理顾客订单信息之用。该用例接收从银联系统反馈来的关于某顾客的订单是否扣款成功的信息，然后把该信息以电子邮件的方式通知该客户。对于扣款成功的订单，通知物流系统给该订单的顾客配送所购商品。

5.2 事件流

5.2.1 基本流

当接收到银联系统发回的订单反馈信息时，本用例开始。

- ① 根据银行的反馈信息，进行不同的处理：

银行账号存在且余额充足，扣款成功，并将订单递交给物流系统。形成内容为“扣款成功并发货”的电子邮件；

银行账号不存在，管理员冻结该用户账号。形成内容为“冻结用户账号”的电子邮件；

银行账号存在但余额不足或欠费，扣款不成功。形成内容为“余额不足扣款不成功”的电子邮件。

② 根据订单号获取该订单顾客的个人信息，主要是获得该顾客的电邮地址。

③ 向顾客发送电子邮件。

5.2.2 备选流

5.2.2.1 发送电子邮件失败

如果发送电子邮件失败，则系统会向管理员发送错误信息。

5.3 特殊需求

无。

5.4 前置条件

管理员必须首先登录到该系统中。

5.5 后置条件

如果该用例成功，会生成通知顾客订单是否成功扣款的电子邮件，并把扣款成功的订单转发给物流系统。否则，系统维持原状。

5.6 扩展点

无。

图 5.16 “网上购物系统”部分用例规约

以上通过两个例题（例 5.6 和例 5.7），介绍了“网上购物系统”建立面向对象的用例模型的过程。下面两节将再用两个例子（例 5.8 和例 5.9），介绍“网上购物系统”的补充规约和术语表。

5.7.3 补充规约

[例 5.8] 为网上购物系统编写一个补充规约。

[解] 图 5.17 显示了“网上购物系统”补充规约的内容。

1. 目的

本补充规约列出了网上购物系统的非功能性需求和部分全局性需求。它和用例模型一起，组成了完整的系统需求规格说明书。

2. 范围

本说明书除定义了在许多用例中共有的功能性需求以外，还定义了系统的非功能性需求，如可靠性、可用性、系统性能和可支持性等。

3. 参考

无。

4. 功能性

- 4.1 满足多个用户的并发执行。
- 4.2 当用户购买某个商品时，系统必须判断该商品是否还有剩余，若该商品已售完，需提醒用户，并在管理员登录时，提醒管理员。
- 4.3 系统能提供管理客户订单的功能给管理员，管理员通过客户的资料来与客户取得联系。
5. 可用性
 - 用户界面视窗与 Windows 系统兼容。
6. 可靠性
 - 保证系统在配置完成以后 24 小时都可用。平均无故障时间应超过 300 小时。
7. 性能
 - 7.1 该系统应能支持多达 5 000 名用户在任何特定时间使用中央数据库，并支持多达 500 名用户在任何时间访问本地服务器。
 - 7.2 系统要求对数据库的访问，存取速度要快，特别是对商品目录数据的访问的反应时间要在 10 秒以内。
 - 7.3 系统要求在 2 分钟内完成 80% 的交易。
8. 可支持性
 - 无。
9. 安全性
 - 系统要求有较高的安全性，由于在生成订单时，用户的信用卡账号需要在网络上传输，所以必须提供额外的安全措施。
10. 设计约束
 - 无。

图 5.17 “网上购物系统”补充规约

5.7.4 术语表

[例 5.9] 为“网上购物系统”编写一个术语表。

[解] 图 5.18 显示了“网上购物系统”术语表的内容。

1. 简介

本文档用来对一些术语进行定义，同时对用例说明或其他文档中读者不太熟悉的术语进行解释性的描述。一般地说，它可用作一种信息数据字典，使得用例规约和其他文档显得简洁、易懂。
2. 名词定义

这份术语表包含了网上购物系统的重要概念。

 - 2.1 顾客：指每个使用该系统进行网上购物的人。
 - 2.2 管理员：负责管理用户和商品信息以及系统维护。
 - 2.3 银联系统：验证用户及信用卡信息并执行扣款操作。
 - 2.4 物流系统：负责给用户派送所购买的商品。
 - 2.5 商品：本系统所出售的物品。

- | |
|--|
| <p>2.6 个人信息：用户注册时填写的个人信息，如使用本系统的账号、密码、联系方式和电子邮件等。</p> <p>2.7 商品信息：商品的名称、价格、产地、所属种类和现有数量等信息。</p> <p>2.8 订单信息：用户订购商品的信息，包括所有商品 ID、名称、单价、数量以及总价。用户在未向系统提交之前可以任意修改所要购买商品的信息。</p> <p>2.9 购物车：用来存放用户所选商品，便于用户查看已选商品数量和价格等信息。</p> <p>2.10 用户账号：用户使用本系统购物之前要先注册，用户账号标识特定的用户，作为进入本系统的凭证。只有注册过的用户才可以购买商品，否则只能浏览商品信息。</p> |
|--|

图 5.18 “网上购物系统”术语表

小 结

正如章名所示，本章讨论了软件需求工程和需求分析等最基本的概念，为学习本书后续各章的内容，特别是面向对象的软件开发技术奠定初步的基础。

从用户的角度看，随着市场的发展与变化，软件需求也应该与时俱进。以文字处理软件为例，其近期版本的功能需求就远远超过了早期版本的功能需求。这表明，需求工程所研究的问题反映了用户对软件随时间而变化的需求，永远没有止境。需求分析则与此不同。所谓需求分析建模，通常是指针对一个特定的版本，从开发者的角度建立起来的分析模型。无论是结构化分析模型还是用例模型，都只考虑某个特定版本（或特定时段）的需求。

作为软件开发的第一个阶段，需求分析由需求获取、需求建模、需求描述和需求验证 4 个步骤组成。需求获取的目的是让开发人员通过各种方式充分和用户交流，全面、准确地了解系统需求；建立需求模型是需求分析的核心，它通过各种图形及符号，可视化地从各个侧面描述系统需求；需求描述即编写需求规格说明书，它以各方共同认可的文档形式表述，是软件设计和系统验收的可靠依据；需求验证用来检验以上各步的工作成果。以上 4 个步骤周而复始，组成了一组迭代的活动，直到获得准确的软件需求。

需求建模时可以采用结构化方法和面向对象方法。本章着重介绍了面向对象的用例模型，它也是全书的重点，由用例模型、补充规约和术语表一起组成。其中用例模型包括用例图和用例规约，主要用于描述软件的功能需求；补充规约用于描述系统的全局性功能和非功能性需求；术语表统一地定义了软件相关术语的含义。结合用例模型，还比较详细地介绍了“选课系统”和“网上购物系统”等示例，可供读者模仿和参考。

随着人们对需求重要性的认识逐渐深入，软件需求管理应运而生。软件规模的扩大，使需求变更不可避免地会发生，而需求变更的失控会导致软件开发延期或失败等严重后果。使用需求管理工具，可以明显改善软件管理和控制的效果。

习 题

1. 什么是软件需求？可以从哪些方面描述软件需求？

2. 软件需求的任务是什么？要经过哪些步骤？
3. 有哪两种主要的需求模型？它们各由哪些部分组成？
4. 基于用例的面向对象的需求建模包括哪些步骤？
5. 建立用例模型时，如何确定参与者和用例？
6. 需求描述（或需求规格说明书）由哪些部分组成？各部分的主要内容是什么？
7. 用例规约应该包含哪些内容？什么是基本流和备选流？它们有什么区别？
8. 术语表有何作用？
9. 什么是软件需求管理？它包含哪 3 个主要活动？
10. 选择一个系统（例如工资管理系统、飞机订票系统、图书馆管理系统等），用基于用例的面向对象需求建模方法对它进行需求分析，并给出需求模型。

第6章 面向对象分析

结构化分析是第一代软件工程常用的分析技术，而面向对象分析属于第二代软件工程常用的分析技术。第5章已明确指出，与在整个软件生存周期内不断与时俱进的需求工程不同，对需求的分析通常是指软件系统在某一特定时段（或某个特定版本）的需求。本章主要讨论面向对象的分析，重点放在基于用例的分析模型上。并结后第5章“网上购物系统”的部分用例，在本章末对其进行用例分析，并建立静态模型和动态模型。

6.1 软件分析概述

用户和开发者都会关心软件的需求，都希望通过需求分析弄清楚“需要软件做什么”，但他们理解问题的角度却各不相同。简单地说，用户一般只注重软件的外在表现，即所谓的软件需求；而开发者更加关注软件的内部逻辑结构，通常称之为软件分析。前者是从软件使用者的角度出发；而后者则是从软件开发者的角度出发的。

以下主要介绍面向对象的软件分析。它所产生的分析模型，也是随后软件设计的基础。

6.1.1 面向对象软件分析

在长期的软件开发实践中，人们提出了多种软件分析和建模的方法，其中占主导地位的主要有结构化分析和面向对象分析两种。在建立的分析模型中，普遍采用图形和自然语言相结合的表达法，并使用多种图形描述工具。第4章介绍的UML，就是面向对象分析(object-oriented analysis, OOA)的重要表达工具。

1. OOA 的主要任务

首先要理解用户的需求，包括全面理解和分析用户需求，明确所开发的软件系统的职责，形成文件并规范地加以表述。

然后进行分析，提取类和对象，并结合分析进行建模。其基本步骤是：标识类，定义属性和方法；刻画类的层次；表示对象以及对象与对象之间的关系；为对象的行为建模。这些步骤可反复进行，直至完成建模。

2. OOA 的模型

图6.1显示了面向对象的分析模型的组成结构。

如图 6.1 所示, 处于 OOA 模型核心的是以用例模型为主体的需求模型。当软件开发小组获得软件需求后, 分析员即可据此创建一组场景 (scenario)。从这些场景出发, 进一步抽取和定义 OOA 模型的 3 种子模型: 即类/对象模型, 描述系统所涉及的全部类和对象, 每一个类/对象都可通过属性、操作和协作者来进一步描述; 对象-关系模型, 描述对象之间的静态关系, 同时定义了系统中对象间所有重要的消息路径, 它也可以具体到对象的属性、操作和协作者; 对象-行为模型, 描述了系统的动态行为, 即在特定的状态下对象间如何协作来响应外界的事件。

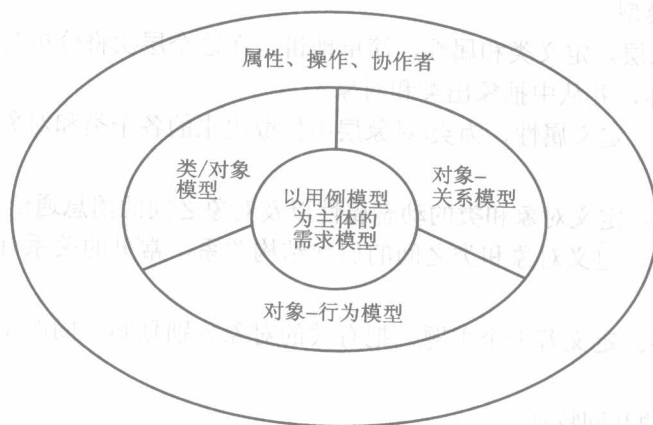


图 6.1 面向对象分析模型的组成结构

3. OOA 的优点

与传统的软件分析方法相比较, 面向对象分析具有如下优点:

- ① 同时加强了对问题空间和软件系统的理解。
- ② 改进包括用户在内的与软件分析有关的各类人员之间的交流。
- ③ 对需求的变化具有较强的适应性。
- ④ 很好地支持软件复用。
- ⑤ 确保从需求模型到设计模型的一致性。

4. 分析模型的一般特点

如图 6.1 所示, 分析模型系由一组子模型组成, 每一子模型都使用图形或符号来表示。从本质上说, 分析模型是一种概念模型 (conceptual model)。不管使用哪种分析方法进行软件分析, 这类模型都具有下述的一般内容和特点:

第一, 全面覆盖软件的功能需求。分析模型是在需求模型的基础上建立起来的, 同时也是软件设计的基础, 任何一个功能需求的遗漏, 都会影响软件设计的质量。

第二, 分析模型与软件的实现无关。这也是称它为概念模型的由来, 它通常忽略了与实现相关的细节, 仅在理想化的状态下考虑问题的解决方案。

第三,分析模型的表述方法与所采用的分析技术有关。不同的分析方法,通常使用不同的分析工具来表示所建立的分析模型。

6.1.2 面向对象分析模型

随着对象技术的成熟与流行,软件工程领域涌现了众多的 OOA 方法,现简介如下。

1. 典型的五层次模型

最典型的是 Coad 和 Yourdon 的 OOA 方法。它采用一个五层次的 OOA 模型,通过下列的步骤来建立各层模型。

① 建立类/对象层。定义类和属性,简单地讲,在这个层次将分析与待开发软件对应的各个现实世界的实体,并从中抽象出类和对象。

② 建立属性层。定义属性,为类/对象层中抽取出来的各个类和对象设计静态属性和它们之间的关系。

③ 建立服务层。定义对象和类的动态属性以及对象之间的消息通信。

④ 建立结构层。定义对象和类之间的层次结构关系,常见的关系有包含关系、继承关系和关联关系。

⑤ 建立主题层。定义若干个主题,把有关的对象分别划归不同的主题,每个主题构成一个子系统。

2. OOA 方法的共同特征

除上述的 OOA 方法外,还陆续出现过多种其他的 OOA 方法。尽管这些方法的内容和步骤均有所不同,但大多数方法都具有下列的共同特征,即:类和类层次的表示;建立对象-关系模型;建立对象-行为模型。在基于面向对象技术的总前提下,这些方法通常具有相似的建模步骤,其内容大致如下:

- 需求理解。
- 定义类和对象。
- 标识对象的属性和操作。
- 标识类的结构和层次。
- 建立对象-关系模型。
- 建立对象-行为模型。
- 评审 OOA 模型。

3. OOA 模型在软件开发中的地位

在结束本节的讨论前,这里再简单谈一下 OOA 模型在软件开发中的地位和作用。

如图 6.2 所示,从建立软件需求模型到开发出软件成品,先后要生成分析、设计和实现 3 种模型。面向对象分析(OOA)是一种从问题空间中通过提取类和对象来进行

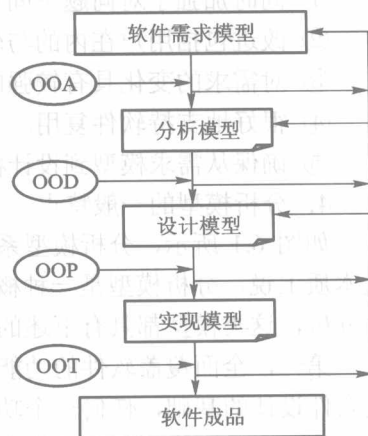


图 6.2 面向对象的软件开发过程

分析的方法，用于建立一个与具体实现无关的面向对象分析模型；面向对象设计（OOD）则从问题空间转移到解空间，在分析模型的基础上考虑实现细节，形成面向对象的设计模型；而面向对象编程（OOP）则用于将设计模型转换成实现模型，可获得源代码和相应的可执行代码。位于最后环节的面向对象测试（OOT），则通过运行可执行代码来检测程序存在的问题。综上所述，面向对象开发的全过程其实是 OOA、OOD、OOP 和 OOT 的迭代过程。

6.2 面向对象分析建模

用例模型是面向对象分析最常采用的一种模型。本节将结合学生“选课系统”，阐明基于用例的面向对象分析方法的建模步骤。

如图 6.1 所示，以用例模型为主体的需求模型，是以模型中的每个用例为研究对象的，不需要考虑实现的细节。通常把这样从用例开始的分析过程称为用例分析，在这一阶段定义的类称为分析类。其目的是为后续的设计活动提供必要的铺垫，无须确定详细的属性和操作。

用例分析的步骤一般是：首先回顾需求阶段产生的用例规约，补充必要的详细信息；然后研究用例的事件流，将用例的职责分配给若干分析类；基于这些职责分配以及分析类之间的协作，即可开始为分析类间的关系建模了。分析了用例以后，需要察看所确定的类，确保它们被详尽地描述，并确保分析模型各个部分之间的一致。

为用例规约补充必要的详细信息，是因为用例规约好比一份黑盒说明，仅仅从用户角度对每个用例进行说明。由于客户一般对系统内部情况不感兴趣，因此在用例规约中很可能省略了有关系统响应动作的内部详细信息，即使说明了也非常粗略。因此，如果要查找执行用例的对象，还需要有一份从内部角度观察系统响应的说明。

6.2.1 识别与确定分析类

从以文字说明的软件需求过渡到以图形来描述的分析模型，是一个渐进的过程。而查找一组备选的分析类，通常是这个过程的第一步。

1. 分析类的类型

通常，分析类被划分为 3 种类型：边界类、控制类和实体类，可分别用标记 <<boundary>>、<<control>>和 <<entity>>来表示。它们依次代表系统与外部环境之间交互的边界、系统在运行中的控制逻辑以及系统要存储和维护的信息。把分析类区分为不同的类型有助于建立一个稳固的对象模型，这是因为分类之后，对模型进行的变更往往只影响到某一特定的部分。例如，用户界面的变更仅仅影响边界类，控制流的变更仅仅影响控制类，系统存储信息的变更仅仅影响实体类。此外，通过区分类型也有助于在分析和初期设计阶段中辨识类。

图 6.3 显示了分析类的图形表述。

(1) 边界类

边界类提供了对参与者或外部系统交互协议的接口，边界类将系统和外界的变化隔离

开，使外界环境的变化不会直接影响系统内部元素。

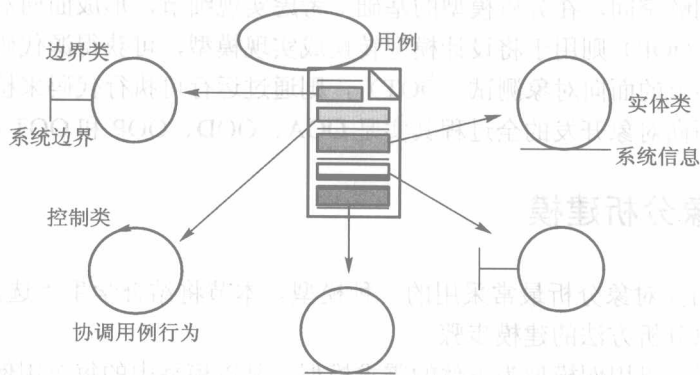


图 6.3 3 种类型的分析类

一个系统可能有多种边界类：

- ① 用户界面类：用于和系统用户进行通信。
- ② 系统接口类：用于和其他软件系统进行通信。
- ③ 设备接口类：为硬件设备（如传感器）提供接口。

边界类对系统中依赖于环境的那些部分进行建模。实体类和控制类对与系统外部环境无关的那部分进行建模。因此，如果更改 GUI 或通信协议，将只更改边界类，对实体类和控制类毫无影响。

（2）控制类

控制类用于封装一个或几个用例所特有的流程控制行为，通过它可建立系统的动态行为模型。它有效地分离了边界类对象和实体类对象，使系统更能承受边界的变更，它还将用例所特有的行为与实体类对象分开，使实体类对象在用例和系统中具有更高的可复用性。

但是，边界类和实体类之间并非始终需要一个控制类，只有当用例的事件流比较复杂并具有可以独立于系统的接口（边界类）或者存储信息（实体类）的动态行为时，才需要控制类。例如，事务管理器、资源协调器和错误处理器等都可以作为控制类。

控制类所提供的行为具有以下特点：

- ① 独立于环境，不随环境的变更而变更。
- ② 确定用例中的控制逻辑（事件顺序）和事务。
- ③ 在实体类的内部结构或行为发生变更时，也不会变更。
- ④ 使用或规定若干实体类的内容，协调这些实体类的行为。
- ⑤ 可能按不同的流程或方式执行（事件流具有多种状态）。

（3）实体类

实体类用于对必须存储的信息和相关的行为建模，其主要职责是存储和管理系统中的信

息。它通常具有持久性，即它们的属性和关系需要长期保存，有时甚至在系统整个生存周期都存在。

实体类对象（实体类的实例）用于保存和更新一些对象的有关信息，例如，事件、人员或者一些现实生活中的对象。一个实体类对象通常不是某个用例所特有的，有时一个实体类对象甚至不专用于一个系统，其属性和关系的值通常来自于参与者。实体类对象是独立于外部环境的。

2. 查找分析类

前已指出，查找分析类通常以每一个用例作为一个研究对象。现分述如下。

① 为每对参与者/用例确定一个边界类，请看下面的示例。

[例 6.1] 为学生选课用例确定其边界类。

[解] 边界类对象担负着协调用例与参与者之间进行交互的职责。对于基于窗口的 GUI 应用程序来说，每个窗口或对话框通常都对应一个边界类。图 6.4 显示了“选课”（即“注册课程”）用例的两个边界类 RegisterForCoursesForm 和 CourseCatalogSystem。

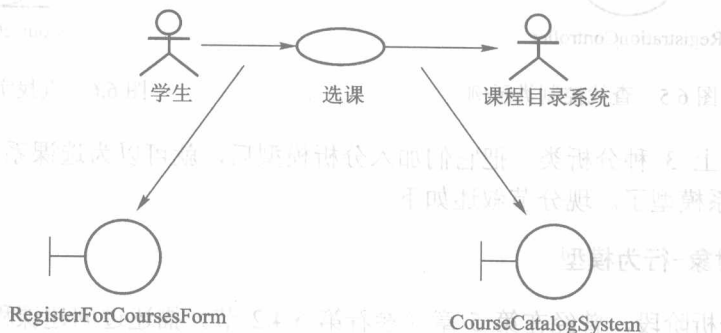


图 6.4 查找边界类示例

边界类对象 RegisterForCoursesForm 包含学生当前的课表，显示当前学期开设的课程供学生选择。边界类对象 CourseCatalogSystem 则提供大学的完整课程目录，并可与系统交互。

② 为每个用例设置一个控制类，请看下面的示例。

[例 6.2] 为学生选课用例确定控制类。

[解] 对于控制类，最简单的查找方法是为每个用例设置一个控制类。图 6.5 显示了与选课用例对应的控制类 RegistrationController。

随着分析的逐步深入，该控制类有可能分解为多个控制类或与其他控制类合并。这时，每个控制类将负责控制对相关用例所描述的功能实现的处理流程。

③ 确定相关的各个实体（包括属性与方法）。在当前的用例模型中，实体信息可以定义成类，或定义成类的属性。如果一个拟建的实体类 A 仅被另一个实体类 B 引用，可考虑将 A 作为实体类 B 的属性；反之，如果某一实体信息可能被多个类引用，或者该实体信息具有明

显的行为特征，则通常将其定义为一个独立的实体类。请看下面的示例。

[例 6.3] 为学生选课用例确定实体类。

[解] 图 6.6 显示了选课用例的实体类。在本例中，学生（Student）、开设课程（CourseOffering）和学生选课的课表（Schedule）均确定为实体。

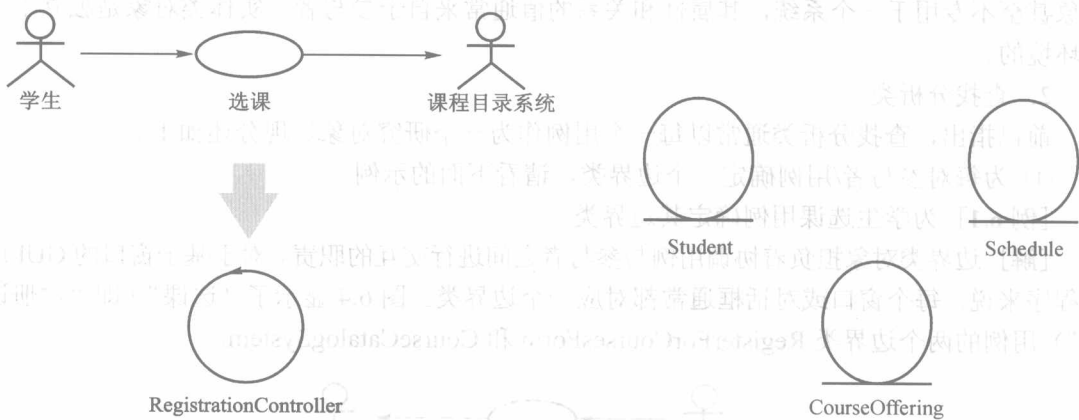


图 6.5 查找控制类示例

图 6.6 查找实体类示例

确定了以上 3 种分析类，把它们加入分析模型后，就可以为选课系统建立对象-行为模型和对象-关系模型了。现分节叙述如下。

6.2.2 建立对象-行为模型

在需求分析阶段，曾经在第 5 章（参看第 5.4.2 节）描述过创建课程表的事件流。其内容依次如下：

- ① 系统从课程目录系统中得到可选择的课程列表，并将列表显示给学生。
- ② 学生从课程列表中选择 4 门主要的和两门备选的课。
- ③ 一旦学生决定了选课情况，系统为他创建一个包含所选课程的课程表。
- ④ 执行提交课程表子事件流。

根据以上的事件流，就可以绘制选课用例的动态图了，其中又包括时序图和协作图。分述如下。

1. 时序图

时序图按时间顺序描述系统元素之间的交互。首先，在时序图中加入触发这个用例的参与者，然后加入边界类对象，接着加入控制类对象，最后是实体类对象。每个用例都是由参与者触发的，故最先应放置参与者；在参与者和用例间总有一个边界类对象，在边界类对象和实体类对象间总会有一个控制类对象；最后才是一些实体类对象。

[例 6.4] 绘制选课用例创建课表事件流的时序图。

[解] 选课用例对应的时序图从学生申请新建课程表开始，消息被 RegisterForCoursesForm 边界类对象接收，然后它请求控制类对象获取开设课程信息。由于开设课程信息存于课程目录系统中，消息需要传给它。课程目录系统也是参与者，所以与它交互前需要先用 CourseCatalogSystem 边界类对象交互，由边界类对象再向课程目录系统传递消息。

接着，RegisterForCoursesForm 边界类对象调用自己的方法向学生显示课表信息，并提供空白课程表供学生选择课程。

再后，学生根据要求从课程列表中选择 4 门主要的和两门备选的课，并将选课信息提交给 RegisterForCoursesForm 边界类对象，边界类对象将创建课程表的消息传递给控制类对象，控制类对象创建课表并将信息加入相应实体类。

图 6.7 显示了这一动态交互过程所对应的时序图。

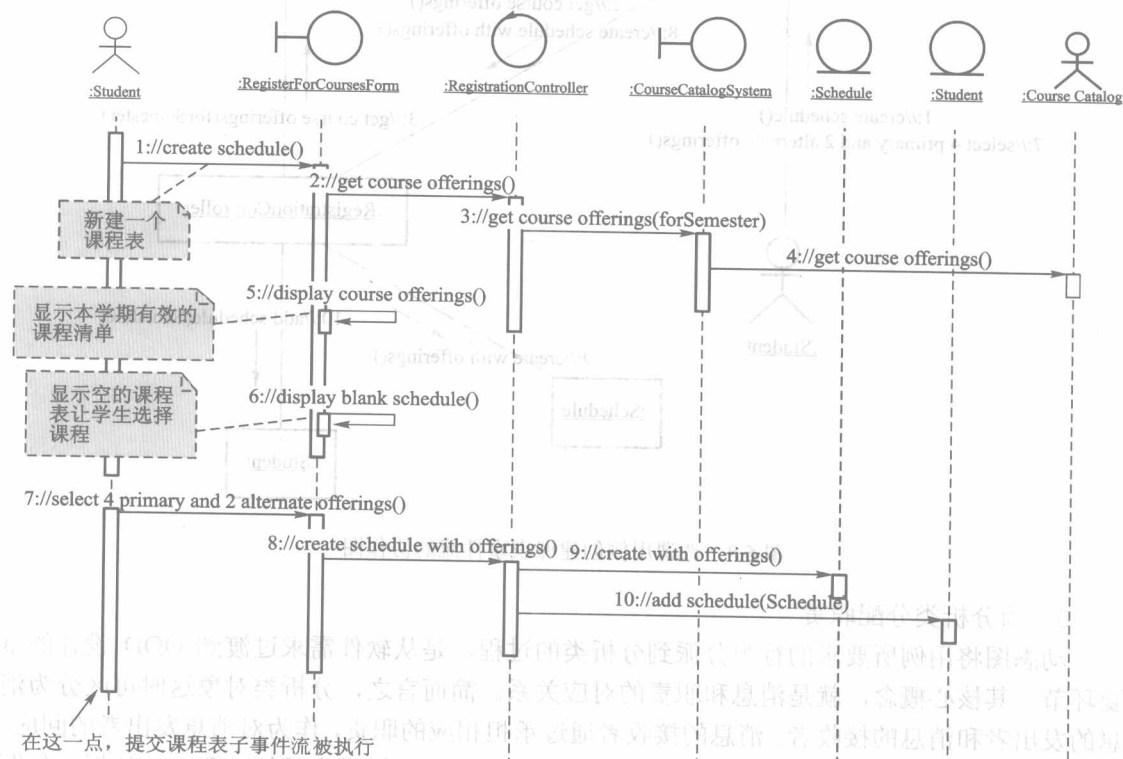


图 6.7 选课用例创建课表事件流的时序图

2. 协作图

协作图按照时间和空间的顺序描述系统元素之间的交互及相互关系。请看下面的例子。

[例 6.5] 绘制选课用例创建课表事件流的协作图。

[解] 图 6.8 显示了与图 6.7 的时序图语义相同的协作图。由此图可见, 在使用 UML 图形符号描述的协作图中, 其事件流同时包含了时间和空间两个维。

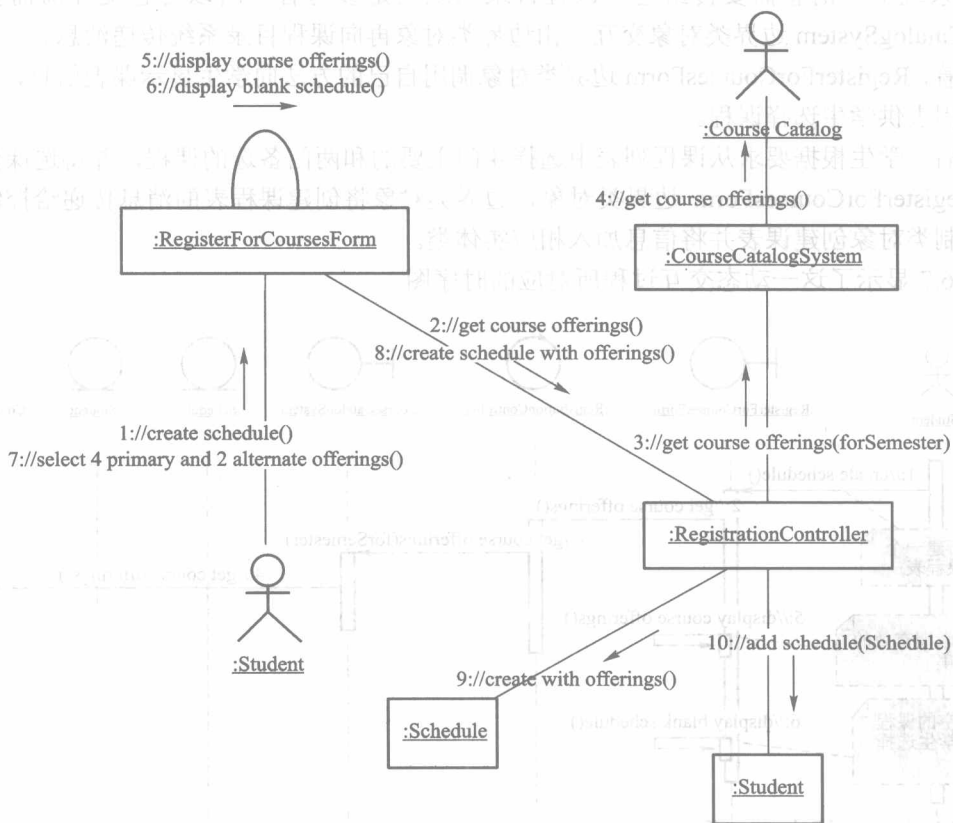


图 6.8 选课用例创建课表事件流的协作图

3. 为分析类分配职责

动态图将用例所要求的行为分派到分析类的过程, 是从软件需求过渡到 OOD 设计的重要环节。其核心概念, 就是消息和职责的对应关系。简而言之, 分析类对象这时可区分为消息的发出者和消息的接收者。消息的接收者通过承担相应的职责, 作为对消息发出者的回应。一个分析类的实例在事件序列中接收的消息集合, 就是该分析类应承担的职责的依据, 如图 6.9 所示。

在动态图中, 分析类的职责可以从交互提供的消息中得到。对每一条消息, 首先检查接收它的对象所属的类。如果职责尚不存在, 则创建一个新的职责以便提供需要的行为。职责大多沿用消息的名称, 它还不是类的操作, 故习惯上用“//”作为前缀, 如图 6.9 所示。

在后续的开发活动中, 分析类将逐步演变为具体的设计元素。相应地, 分析类的职责也

将逐步演变为设计元素的行为，具体讲就是设计类的操作和子系统接口的行为。

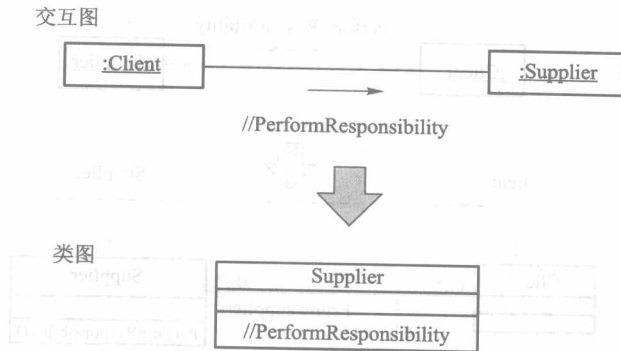


图 6.9 消息与职责的对应关系

4. 状态图

协作图和时序图能很好地表示类对象间为了响应某一事件所发生的消息交互。但有些类所涉及的用例行为比较复杂，并且分散到不同的事件序列中，这时就需要为这个类创建一个状态图，专门针对一个类的状态变化，研究该类的动态行为。

6.2.3 建立对象-关系模型

上节讨论了用例模型的对象-行为模型，又称为动态模型。本节将接着介绍用例模型的对象-关系模型，即静态模型，主要涉及分析类的属性、分析类的关联、分析类图和分析类的合并等内容。现分述如下。

1. 分析类的属性

所谓分析类的属性，即分析类本身具有的信息。类可以用属性来存储信息。

属性名称应当是一个名词，属性类型是简单的数据类型，包括字符串、整型和数值型等。在用例分析阶段，分析类是粗略的，因而其属性也是相对粗略的，目的仅在于在逻辑上支撑分析类所承担的职责。

2. 分析类的关联

分析类具有指向其他分析类的关联，通过这种关联能够找到其他分析类。

在协作图中，对象之间的链接是相应分析类之间存在关联关系的动态表现形式。它表明两个类的对象需要互相交流，以便执行用例的行为。图 6.10 展示了链与关联关系间的对应关系。

3. 分析类图

分析类图用于表现分析类及其关系，其中描述某个用例的分析类图称为参与类图（view of participating classes, VOPC）。从逻辑上说，每个用例可对应一张完整的参与类图。在具体实践中，一个用例可以绘制多张参与类图，用每张图展示不同的侧重点。图 6.11 显示了选课

用例的参与类图。

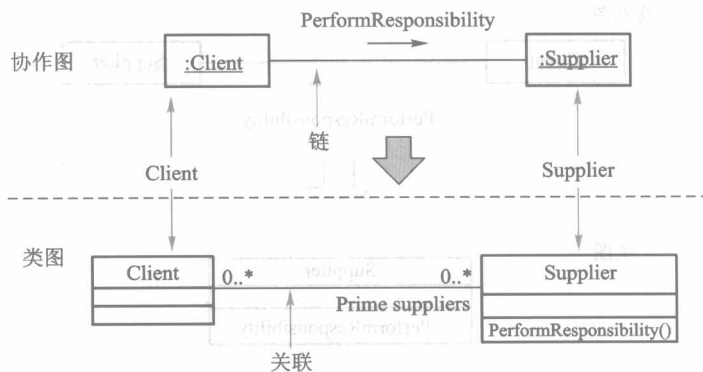


图 6.10 链与关联关系间的对应关系

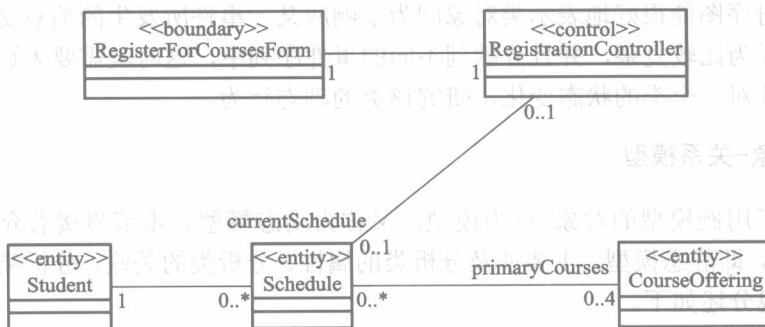


图 6.11 选课用例的参与类图

由图 6.11 可见，参与类图可显示类的实例之间的数量关系：每个 RegisterForCoursesForm 都有自己的 RegistrationController；一个 RegistrationController 每次处理一张课表 Schedule（学生注册课程的当前课表 currentSchedule）；每个学生 Student 可以选择 4 门主要课程 primaryCourses；同一课程 CourseOffering 可以出现在许多课表中作为主要或备选课程；一个学生可以有多张或没有课表；一张课表只能唯一属于一个学生，不存在不与学生联系的课表。

4. 分析类的合并

每个分析类都代表一个明确定义的概念，具有不重叠的职责。一个类可以参与任何数量的用例，因此就整个系统而言，需要合并分析类，把具有相似行为的类合并为一个。每当更新了一个类，就要更新或补充用例规约，必要时还要更新原始的需求。

图 6.12 显示了选课与关闭选课两个用例的分析类的合并。图中 Student、CourseOffering、Schedule 等类同时出现在 RegisterForCourse 和 CloseRegistration 两个用例中，可以将它们分

别合并。

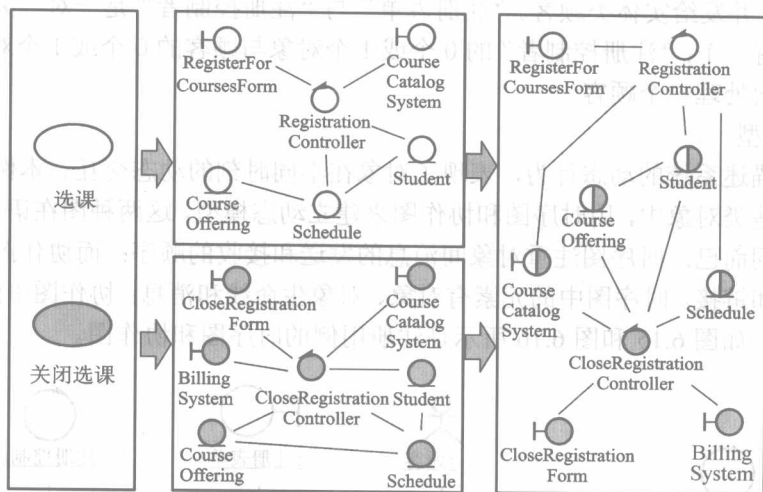


图 6.12 分析类的合并

6.3 面向对象分析示例

作为示例，本节将对第 5.7 节所述的“网上购物系统”用面向对象方法来进行分析建模。从第 6.3.1 节到 6.3.5 节，分别对注册、维护个人信息、维护购物车、生成订单和管理订单等 5 个用例进行用例分析，并建立静态模型和动态模型。

6.3.1 注册

[例 6.6] 为注册用例进行用例分析，并建立静态模型和动态模型。

[解]

1. 确定分析类

如图 6.13 所示为注册用例的分析类。

边界类：本例使用“注册表单”来抽象顾客与系统交互的图形界面。

控制类：本例控制类为“注册控制者”，负责接收边界类“注册表单”的消息，将其发给实体类。

实体类：本用例只涉及顾客的注册，所以实体类也只有“顾客”。

2. 静态模型

用图 6.14 所示参与类图描述各类之间的关系。

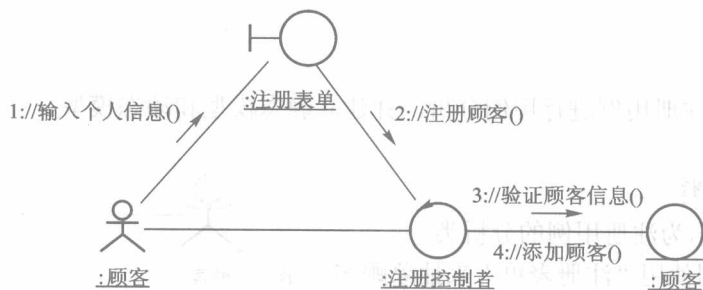
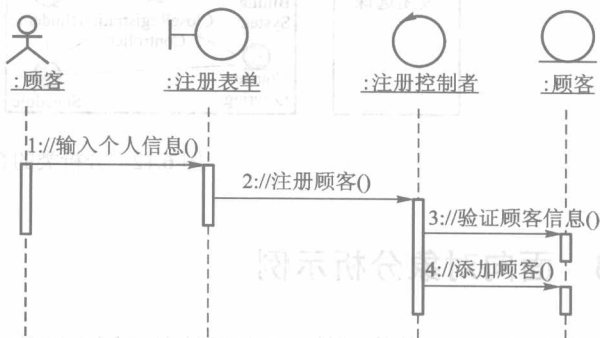
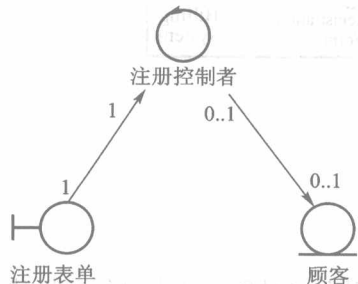


图 6.13 注册用例分析类

在图 6.14 中,各个类之间都是单向关联。控制类“注册控制者”接收来自边界类“注册表单”的消息,并发给实体类顾客。“注册表单”与“注册控制者”是一对一关联,因为顾客注册的界面只有一个;“注册控制者”的 0 个或 1 个对象与顾客的 0 个或 1 个对象关联,一个注册控制者每次处理一个顾客。

3. 动态模型

动态模型描述系统的动态行为,表现了对象在不同时刻的动态交互。本例将注册用例的行为分派到这些类对象中,用时序图和协作图来建立动态模型。这两种图在语义上是相同的,只是侧重点不同而已。时序图注重对象间消息的发送和接收的顺序;而协作图着重于协作对象之间的交互和链接。时序图中的元素有对象、对象生命线和消息;协作图中的元素有对象、链接和消息流。如图 6.15 和图 6.16 所示是注册用例的时序图和协作图。



说明:首先由顾客输入注册时需要填写的个人信息,“注册控制者”接收由“注册表单”发送过来的信息,由顾客实体类在数据库中验证是否已有相同的账号。若顾客所填信息符合要求,则“注册控制者”向顾客实体类发送添加顾客的消息,将该顾客添加到数据库中。

由于顾客注册时需要填写个人信息，包括使用本系统的账号、密码、联系地址以及电子邮件等。所以可得出顾客实体类的属性有：`customerID`，`customerName`，`customerPassword`，`customerAddr`，`customerEmail`。

交互图中的每一个消息对应消息接收对象的一个服务，即方法，本例将使用以下方法：
注册表单的方法：输入个人信息 `enterIndividualInfo()`。

注册控制者的方法：注册顾客 `regCustomer()`。

顾客的方法：验证顾客信息 `hasCustomer()`，添加顾客 `addCustomer()`。

6.3.2 维护个人信息

[例 6.7] 为维护个人信息用例进行用例分析，并建立静态模型和动态模型。

[解]

1. 确定分析类

如图 6.17 所示为维护个人信息用例的分析类。

边界类：本例使用“维护个人信息表单”来封装面向顾客这个参与者的接口。

控制类：本例控制类为“维护个人信息控制者”，负责接收边界类“维护个人信息表单”的信息，将其发给实体类。

实体类：该用例是顾客用来维护注册时填写过的个人信息，故涉及的实体类依然只有“顾客”。

2. 静态模型

用图 6.18 所示参与类图描述各类之间的关系。



图 6.17 维护个人信息用例分析类

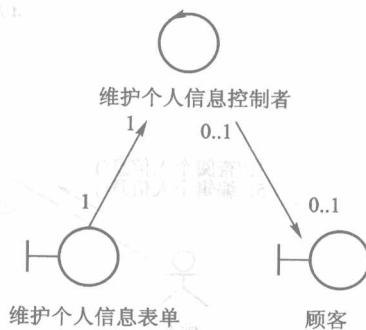


图 6.18 维护个人信息的参与类图

如图 6.18 所示，各个类之间都是单向关联关系。控制类“维护个人信息控制者”接收来自边界类“维护个人信息表单”的消息，并发给实体类顾客。“维护个人信息表单”与“维护个人信息控制者”是一对一关联；“维护个人信息控制者”的 0 个或 1 个对象与顾客的 0 个或

1个对象关联,一个“维护个人信息控制者”每次处理一个顾客。

3. 动态模型

顾名思义,维护个人信息可能出现两种情况。其一是顾客仅仅查看其在注册时填写的个人信息,不加修改;二是查看后接着修改个人信息。动态模型应同时满足这两种情况。如图6.19和图6.20所示是维护个人信息的时序图和协作图。

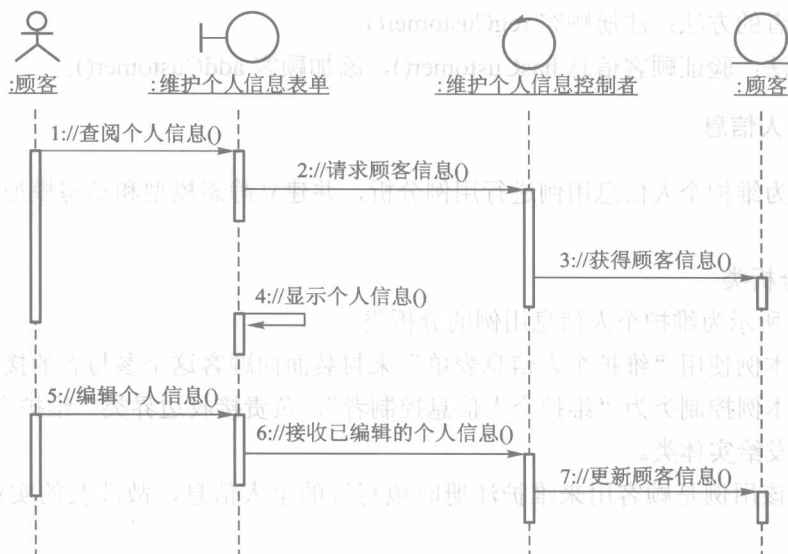


图 6.19 维护个人信息的时序图

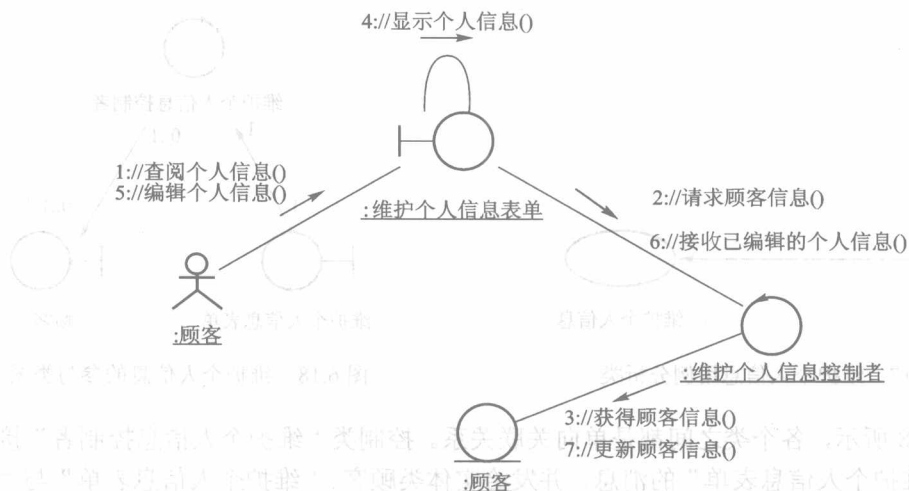


图 6.20 维护个人信息的协作图

说明：顾客申请查看其在注册时填写的个人信息，通过“维护个人信息表单”向“维护个人信息控制者”发出获得其信息的请求，顾客实体类从数据库中取得该顾客的个人信息，交付“维护个人信息表单”加以显示。然后顾客对他的个人信息进行编辑修改，之后发送给“维护个人信息控制者”，最后由其向顾客实体类发送更新顾客信息的信息。这样，数据库中该顾客记录将被修改。

从以上用例的交互图中，可以得出“维护个人信息”用例中分析类的方法。

维护个人信息表单的方法：查阅个人信息 `checkIndividualInfo()`，编辑个人信息 `editIndividualInfo()`，显示个人信息 `displayIndividualInfo()`。

维护个人信息控制者的方法：请求顾客信息 `requestCustomerInfo()`，接收已编辑的个人信息 `receiveEditedIndividualInfo()`。

顾客的方法：获得顾客信息 `getCustomerInfo()`，更新顾客信息 `updateCustomer()`。

6.3.3 维护购物车

[例 6.8] 为维护购物车用例进行用例分析，并建立静态模型和动态模型。

[解]

1. 确定分析类

如图 6.21 所示为维护购物车用例的分析类。

边界类：设计一个边界类来封装面向顾客这个参与者的接口，即“维护购物车表单”。

控制类：本用例的控制类为“维护购物车控制者”，负责接收边界类“维护购物车表单”的信息，然后将其分发给实体类。

实体类：本用例涉及的实体类有“购物车”和“购物项目”。

2. 静态模型

用图 6.22 的参与类图描述各类之间的关系。

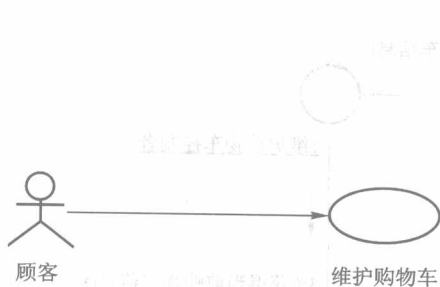


图 6.21 维护购物车用例分析类

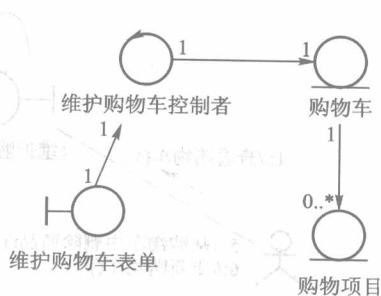


图 6.22 维护购物车的参与类图

由于控制类“维护购物车控制者”接收来自边界类的信息并分发给实体类，所以它与边界类和实体类都存在着关联关系。由图 6.22 可见，“维护购物车表单”与“维护购物车控制

者”是一对一关联，因为顾客使用的界面只有一个，每个表单都有自己的控制者；“维护购物车控制者”与购物车是一对一关联，而购物车与购物项目是一对多的关系，表明一个购物车中可以加入多个购物项目。

3. 动态模型

完成维护购物车，通常要经过“查看—确认（允许删除或更新）”这两步。

(1) 维护购物车

如图 6.23 和图 6.24 所示为维护购物车的时序图和协作图。

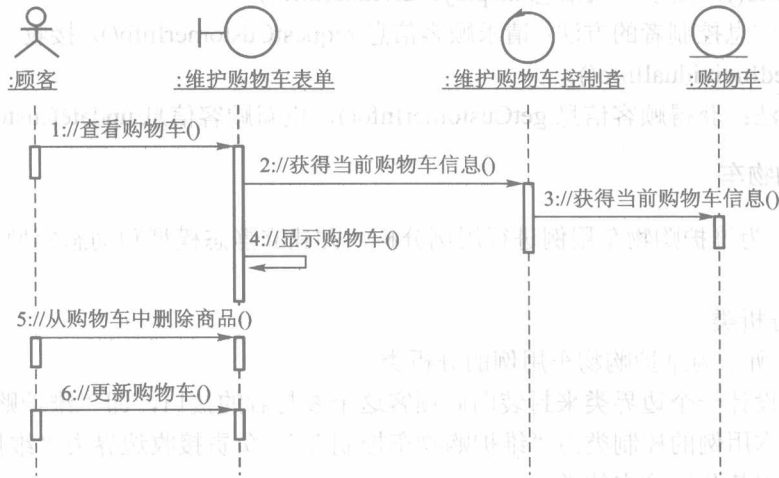


图 6.23 维护购物车的时序图

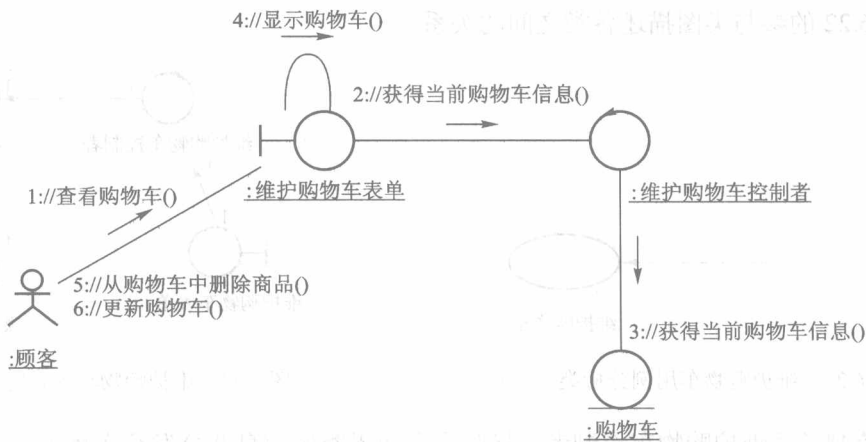


图 6.24 维护购物车的协作图

说明：在顾客进行维护购物车之前，首先向“维护购物车表单”发送“查看购物车”的消息，从“维护购物车控制器”获取当前购物车的信息。当购物车实体类从数据库中取得当前购物车的信息后，即用“维护购物车表单”加以显示。接下来顾客便可以进行确认操作，如删除购物车中的商品和修改商品数量等。

(2) 子事件

① 从购物车中删除商品：如图 6.25 和图 6.26 所示为从购物车中删除商品的时序图和协作图。

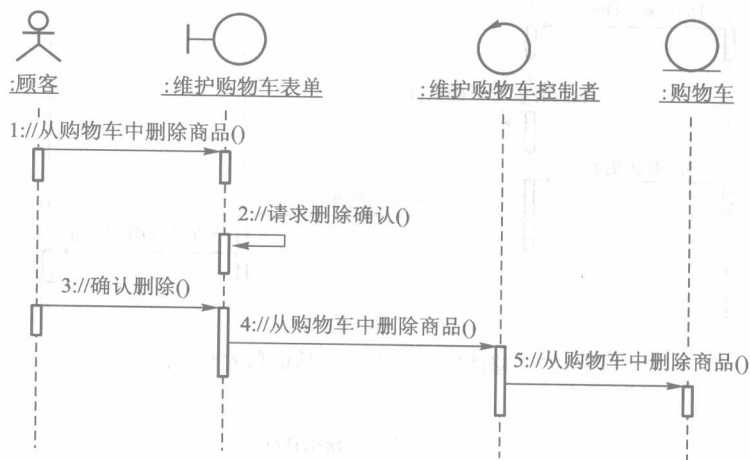


图 6.25 从购物车中删除商品的时序图

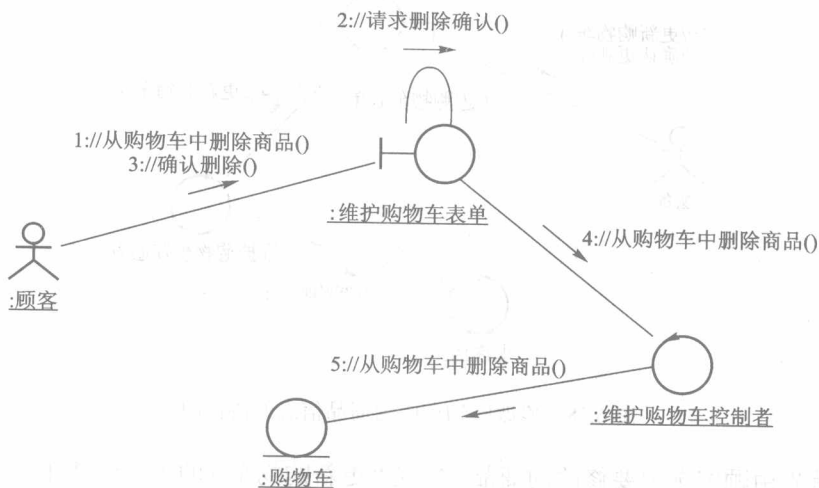


图 6.26 从购物车中删除商品的协作图

说明：首先由顾客选中要删除的商品，发送“从购物车中删除商品”的消息给“维护购物车表单”，“维护购物车表单”请求其进行删除确认，如果顾客确认删除，则发送删除的消息给“维护购物车控制者”，由其向购物车实体类发送删除商品的消息。

② 修改购物车中的商品信息：如图 6.27 和图 6.28 所示为其时序图和协作图。

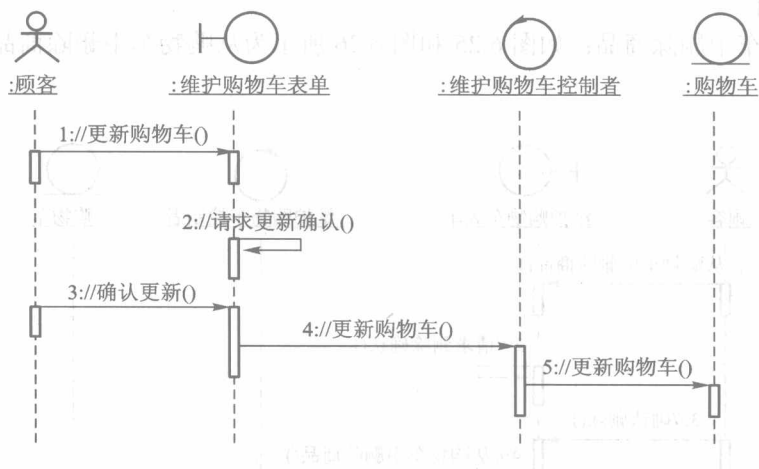


图 6.27 修改购物车中的商品信息的时序图

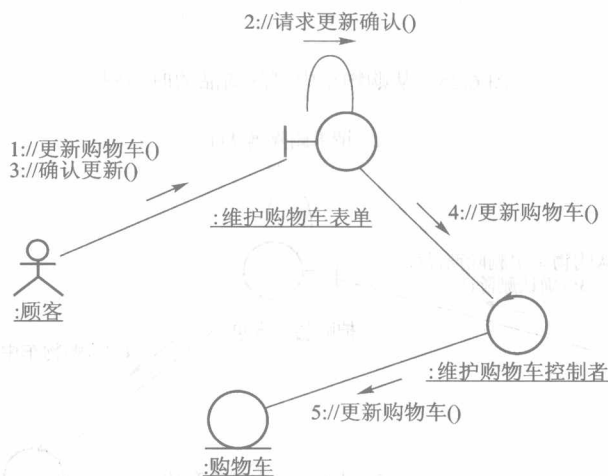


图 6.28 修改购物车中的商品信息的协作图

说明：首先由顾客选中要修改的商品，发送“更新购物车”的消息给“维护购物车表单”，“维护购物车表单”请求其进行更新确认，如果顾客确认更新，则发送更新的消息给“维护购物车控制者”，由其向购物车实体类发送更新商品信息（如商品的数量等）的消息。

本例涉及购物车中的商品信息有商品编号、商品名、价格、数量和总价。由此可得出购物项目实体类的属性有：`commodityID`，`commodityName`，`price`，`amount`，`totalprice`。而购物车中的内容是多个购物项目，所以购物车的属性就是一个购物项目的列表，令其属性名为 `clist`。

“维护购物车”用例中分析类的方法如下：

维护购物车表单：查看购物车 `viewCart()`，显示购物车 `displayCart()`，从购物车中删除商品 `deleteCommodity()`，更新购物车 `updateCart()`，请求删除确认 `requestDeletionConfirmation()`，确认删除 `confirmDeletion()`，请求更新确认 `requestUpdateConfirmation()`，确认更新 `confirmUpdate()`。

维护购物车控制者：获得当前购物车信息 `getCurrentCart()`，从购物车中删除商品 `deleteCommodity()`，更新购物车：`updateCart()`。

购物车：获得当前购物车信息 `getCurrentCart()`，从购物车中删除商品 `deleteCommodity()`，更新购物车：`updateCartCommodity()`。

6.3.4 生成订单

[例 6.9] 为“生成订单”用例进行用例分析，并建立静态模型和动态模型。

[解]

1. 确定分析类：

如图 6.29 所示为其分析类。



图 6.29 生成订单用例分析类

边界类：有两个边界类，它们分别是“生成订单表单”和“银联系统”。

控制类：本例的控制类为“生成订单控制者”，负责接收“生成订单表单”的信息，并将其分发给实体类。

实体类：因为本例是用来生成订单的，所以涉及的实体类为“订单”。

2. 静态模型

用图 6.30 的参与类图描述各类之间的关系。

在生成订单的类图中，各个类之间都是单向关联。控制类“生成订单控制者”接收来自边界类“生成订单表单”的消息，并发给实体类订单。“生成订单表单”与“生成订单控制者”是一对一关联，因为顾客用以填写订单的界面只有一个；“生成订单控制者”的一个对象与订

单的0个或1个对象关联；与银联系统也是一对一关联；而订单与购物车则为依赖关系。

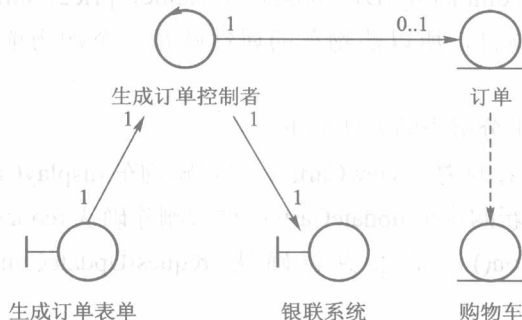


图 6.30 生成订单的参与类图

3. 动态模型

在识别上述类之后，通过建立交互图，将生成订单用例的行为分派到这些类对象中。我们依旧用时序图和协作图（如图 6.31 和图 6.32 所示）来建立动态模型。

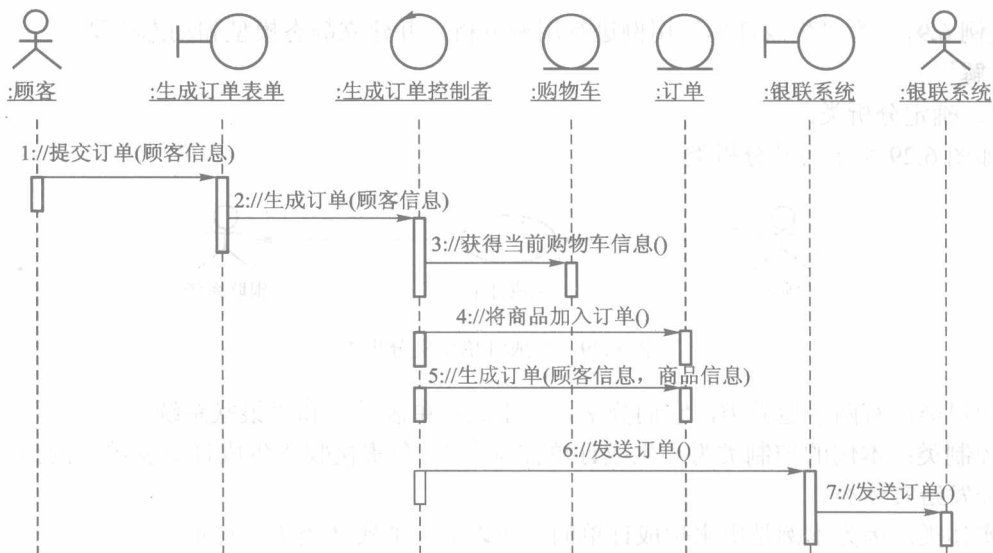


图 6.31 生成订单的时序图

说明：顾客确定要购买购物车中的商品后，填写姓名、地址和银行账号等信息，然后由“生成订单表单”向控制者发送“生成订单”的消息，接着从购物车中获取当前购物车中的商品，计算所有商品总价之和并加入订单，这样，订单实体类就负责向数据库中添加新订单信

息。最后，将生成的订单发送给银联系统进行处理。

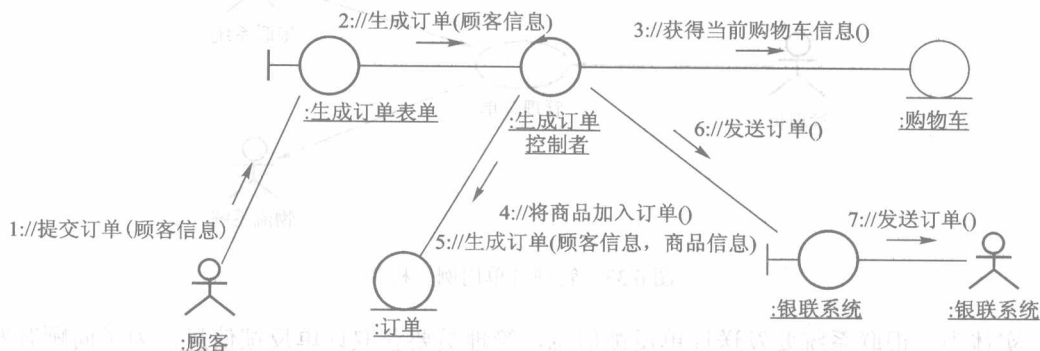


图 6.32 生成订单的协作图

这里涉及订单上的所有信息，包括订单号、订单日期、所购买的商品信息（即购物车中的内容）、所有商品总价之和、顾客姓名、地址和银行账号等。所以可以得出订单实体类的属性有：`orderId`，`orderDate`，`cart`，`sum`，`custName`，`custAdd`，`creditCard`。

因为在交互图中的每一个消息对应消息接收对象的一个服务，即方法，所以在以上用例的交互图中，可以得出“生成订单”用例中分析类的方法。

生成订单表单的方法：提交订单 `submitOrder()`。

生成订单控制者的方法：生成订单 `generateOrder()`。

订单的方法：将商品加入订单 `setCart()`，生成订单 `generateOrder()`。

银联系统的方法：发送订单 `sendOrder()`。

6.3.5 管理订单

[例 6.10] 要求完成以下功能：

- ① 对管理订单用例进行用例分析，并建立静态模型和动态模型。
- ② 向顾客发送电子邮件。

[解] 首先完成第①项要求的功能。

1. 确定分析类

如图 6.33 所示为其分析类。

边界类：本例有 3 个边界类，即管理订单表单、银联系统和物流系统，分别与 3 个参与者进行交互。

控制类：按照每个用例设计一个控制类的原则，本例只设置“管理订单控制者”一个控制类。

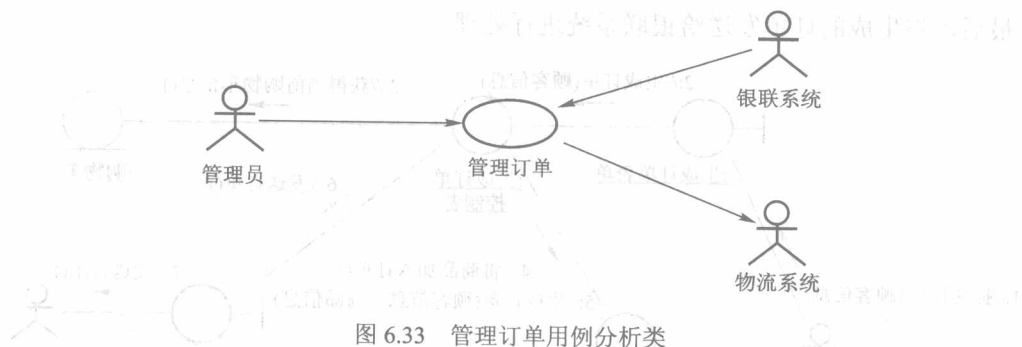


图 6.33 管理订单用例分析类

实体类：银联系统要发送订单反馈信息，管理员要获取订单反馈信息，为了向顾客发送电子邮件，还需获取顾客的电子邮件地址，所以涉及的实体类有“订单”和“顾客”。

2. 静态模型

用图 6.34 的参与类图描述各类之间的关系。

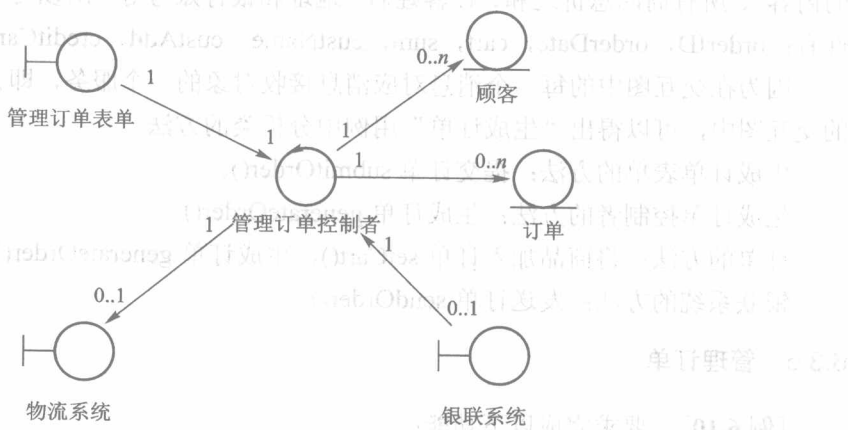


图 6.34 管理订单的参与类图

在这一类图中，各个类之间都属于单向关联。例如，控制类“管理订单控制者”接收来自边界类“管理订单表单”的消息；“管理订单表单”与“管理订单控制者”是一对一关联，即每个“管理订单表单”有自己的控制者；“管理订单控制者”的一个对象与物流系统的 0 个或 1 个对象关联；与银联系统的 0 个或 1 个对象关联。

此外，“管理订单控制者”还与顾客的 0 个或多个对象关联，表明可同时获得多个顾客的电子邮件地址；与订单的 0 个或多个对象关联，表明可获得多个订单反馈信息。

3. 动态模型

为了将“管理订单”用例的行为分派到相关的分析类对象中，下面将建立动态模型。

(1) 获取订单反馈信息，且将扣款成功的订单发送至物流系统

为提高动态模型的可读性，看起来更加简洁，这里将它分绘为两部分，即由银联系统发送订单反馈信息；管理员获取订单反馈信息，并发送成功扣款的订单。

① 银联系统发送订单反馈信息。其时序图和协作图如图 6.35 和图 6.36 所示。

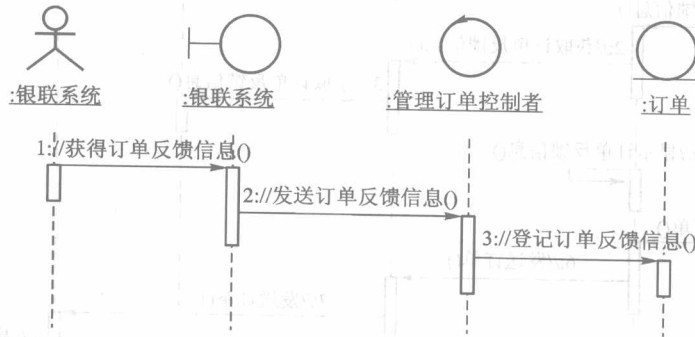


图 6.35 银联发送订单反馈信息的时序图

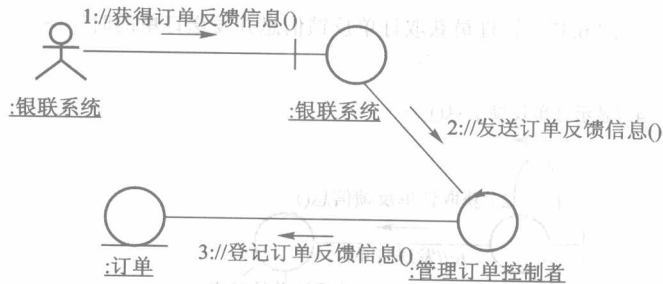


图 6.36 银联发送订单反馈信息的协作图

说明：首先由边界类银联系统获得订单反馈信息，然后发送给“管理订单控制者”，由其向实体类订单发送消息；再由实体类订单负责登记反馈信息到数据库中。

由于模型中涉及订单的反馈信息，相应的实体类订单应有属性 `bankFeedback`。同样，边界类银联系统也有属性 `bankFeedback`。

根据交互图中的“每一个消息需对应消息接收对象的一个服务”的原则，在以上交互图中，还将涉及相关分析类的下列方法。

银联系统：获得订单反馈信息 `getBankFeedback()`。

管理订单控制者：发送订单反馈信息 `sendBankFeedback()`。

订单：登记订单反馈信息 `setBankFeedback()`。

② 管理员获取订单反馈信息，并发送成功扣款的订单。其时序图和协作图如图 6.37 和

图 6.38 所示。

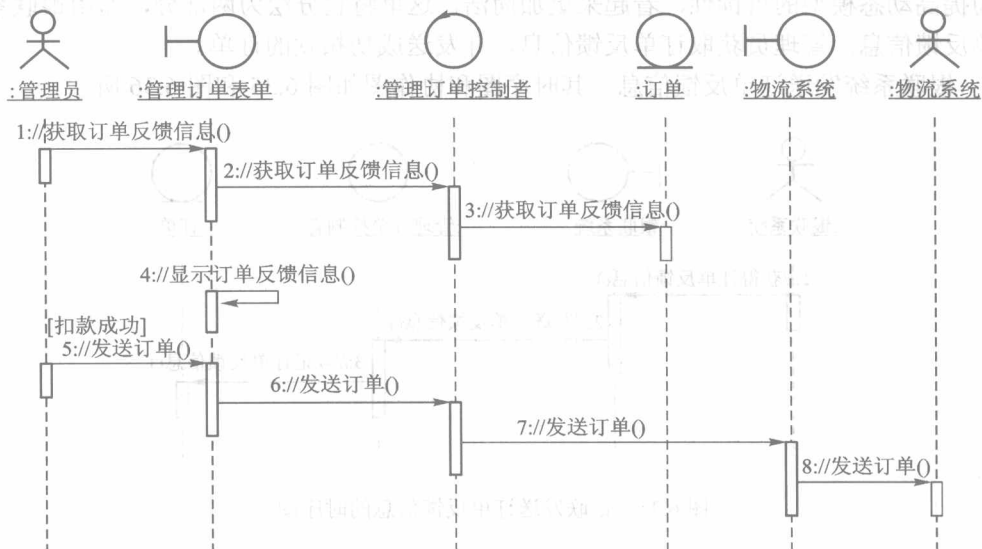


图 6.37 管理员获取订单反馈信息并发送订单的时序图

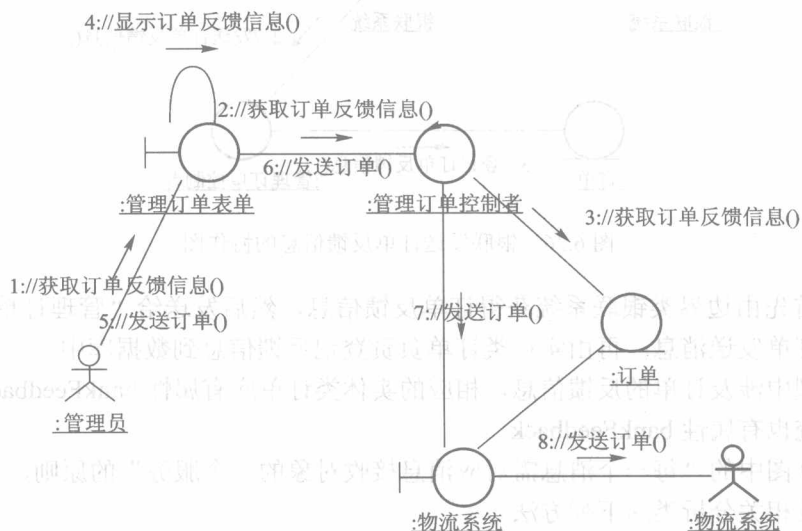


图 6.38 管理员获取订单反馈信息并发送订单协作图

说明：首先由管理员通过“管理订单表单”，向“管理订单控制者”发出获取订单反馈信息的请求；接下来实体类订单从数据库中取得相应订单的反馈信息，由“管理订单表单”

加以显示。若反馈信息显示扣款成功，则管理员申请发送订单；消息被“管理订单表单”接收，再发送给“管理订单控制者”，由其向边界类物流系统发送消息；最后由边界类物流系统发送至系统外的参与者物流系统。

与①相似，可以得出相关分析类的方法：

管理订单表单：获取订单反馈信息 `getBankFeedback()`，显示订单反馈信息：`displayBankFeedback()`，发送订单 `sendOrder()`。

管理订单控制者：获取订单反馈信息 `getBankFeedback()`，发送订单 `sendOrder()`。

订单：获取订单反馈信息 `getBankFeedback()`。

物流系统：发送订单 `sendOrder()`。

(2) 给顾客发送邮件

下面完成本例第②项要求的功能。给顾客发送邮件的时序图和协作图如图 6.39 和图 6.40 所示。

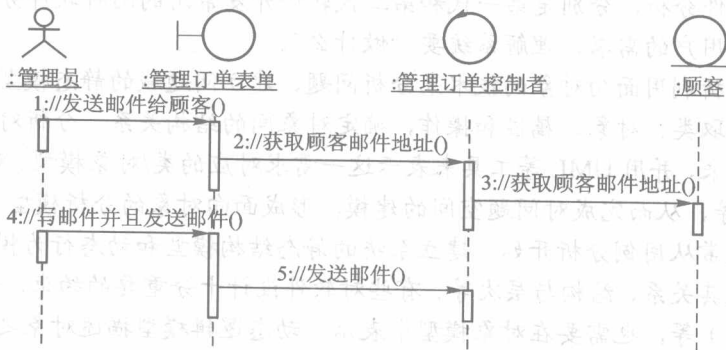


图 6.39 给顾客发送邮件的时序图

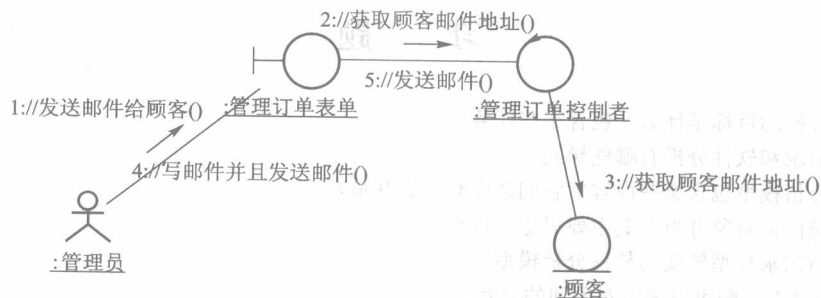


图 6.40 给顾客发送邮件的协作图

说明：当管理员获得银联系统发送的订单反馈信息之后，就可以申请发送电子邮件给顾客了。首先，管理员申请发送电子邮件，通过“管理订单表单”向“管理订单控制者”发送

获取邮件地址的消息；接着顾客实体类从数据库中取得电子邮件地址；然后，管理员便可以根据银行的反馈信息进行不同的处理，形成不同内容的电子邮件并发送给顾客。

由此可以得出相关分析类的方法：

管理订单表单：发送邮件给顾客 `sendEmailToCustomer()`。写并且发送邮件：`writeAndSendEmail()`。

管理订单控制者：获取顾客邮件地址 `getCustomerEmail()`，发送邮件 `sendEmail()`。

顾客：获取顾客邮件地址 `getCustomerEmail()`。

小 结

软件分析将软件需求阶段产生的需求模型转变为软件分析模型。分析模型其实就是从软件开发者的角度，在静态结构和动态行为两个方面来描述待开发的软件系统。结构化软件分析和面向对象软件分析，分别是第一代和第二代软件开发常用的两种软件分析方法。但它们的目的是分析用户的需求，理解系统要“做什么”。

面向对象分析利用面向对象的技术来分析问题、建立问题域的静态模型和动态模型，通过理解问题，抽取类、对象、属性和操作，确定对象间的结构关系，分析对象的状态变迁及对象间的消息往来，并用 UML 等工具来表示这一需求对应的类/对象模型、对象-关系模型和对象-行为模型等，从而完成对问题空间的建模，形成面向对象的分析模型。

软件分析通常从用例分析开始，建立系统的静态结构模型和动态行为模型。静态逻辑模型描述类对象及其关系、结构与层次等。有些对软件设计十分重要的约束，如重数（一对一、一对多、多对多）等，也需要在对象模型中表示。动态逻辑模型描述对象之间的互相作用。互相作用通过一组协同的对象、对象之间消息的有序序列及参与对象的可见性定义来定义系统运行时的行为。

习 题

1. 软件分析的目标是什么？包含哪些任务？
2. 软件需求和软件分析有哪些异同？
3. 软件分析模型包含哪些内容？它们之间有什么联系？
4. 什么是面向对象分析？其主要思想是什么？
5. 如何将需求模型转变为软件分析模型？
6. 用哪种 UML 图可以表示对象间的交互？
7. 分析一个用例的行为时，通常需要画几个交互图？为什么？
8. 分析模型中静态结构模型和动态行为模型间有何联系？
9. 选择一个系统（例如工资管理系统、飞机订票系统或图书馆管理系统等），分别用 OOA 方法对它进行分析，并给出分析模型。

第7章 面向对象设计

面向对象设计的任务，是将分析阶段建立的分析模型转变为软件设计模型。与传统方法不同，面向对象分析和面向对象设计之间的界限表面上不很明显，再者面向对象分析和面向对象设计一般都是迭代过程，设计之后可能再回到分析。但是同传统方法一样，OOD方法也是在基于抽象、信息隐蔽、功能独立和模块化等重要的软件设计概念的基础上进行的，不过它的模块化不仅仅局限在过程处理部分，而是通过将数据和对数据的操作封装在一起，共同完成信息和处理的双重模块化。

7.1 软件设计概述

在软件开发时期，设计阶段是最富有活力和最需要发挥创造力的阶段。从早期的模块化设计和自顶向下设计，到软件工程时代的各种系统设计方法，许多学者做出了重要的贡献。和软件分析一样，软件设计也有两种主流设计方法：以结构化程序设计为基础的结构化软件设计和由面向对象方法导出的面向对象软件设计。

7.1.1 软件设计的概念

设计的目标，是细化解决方案的可视化设计模型，确保设计模型最终能平滑地过渡到程序代码。关键是构造解决问题的方案，并在决定实施细节的基础上获得该方案的设计模型。

从表面上看，设计模型同分析模型有许多相似之处，但两者的目的有本质的区别。分析模型强调的是软件“应该做什么”，并不给出解决问题的方案，也不涉及具体的技术和平台。例如，它不必关心在 J2EE 还是 .NET 平台上实现，是否应用 EJB 或一般的 JavaBean，系统是安装在 WebSphere 还是 WebLogic 环境下。而设计模型要回答“该怎么做”的问题，而且要提供解决问题的全部方案，包括软件如何实现、如何适应特定的实施环境等。当设计模型完成后，编程人员便可以进行编程了。

通过多年的实践，软件设计已形成了一系列基本概念，成为各种设计方法的基础。本节将作简单介绍。

1. 模块与构件

模块 (module) 的概念由来已久。汇编语言中的子程序，FORTRAN 语言中的辅程序，

Pascal 语言中的过程, Java 语言中的类都是模块的实例。在软件工程时代, 模块化仍是大型软件设计的基本策略 (strategy), 而且有所发展, 较过去更完善了。

模块是一个拥有明确定义的输入、输出和特性的程序实体。如果模块的所有输入都是实现功能必不可少的, 所有输出都有动作产生, 即成为定义明确的模块。也就是说, 如果少了一个输入, 模块就不能实现全部功能; 它没有不必要的输入, 每个输入都用于产生输出; 每个输出都是模块执行某一功能的结果, 没有未经模块的转换就变成输出的输入。

广义地说, 对象也是一种模块。在模块设计中要求的高内聚、低耦合等性质 (参见 7.1.3 节), 在对象设计中仍然适用。但由于对象具有自我封闭的特点, 因而更容易在软件设计中被重复使用。这种可重复使用的软件组件就称为软件构件 (software component), 现有的可复用构件, 大多数是在对象的基础上创建的。

2. 抽象与细化

随着软件规模的不断增大, 设计的复杂性也不断增大, 抽象 (abstraction) 便成了控制复杂性的基本策略之一。软件工程是一种层次化的技术, 抽象也是分层次的。在传统软件工程中, 分层数据流图 (分析文档, 见 3.2.1 节) 就体现了抽象的层次 (或等级) 思想。在软件的抽象层次中, 最高层 (级) 的抽象程度最高, 若需要系统某部分的细节, 可移向较低层次 (级) 的抽象。越是到较低层次, 越可看到更多的细节。

软件设计其实就是在不同抽象级别考虑和处理问题的过程。首先在最高抽象级别上, 用面向问题空间的语言描述问题, 概括问题解的形式; 然后不断具体化, 降低抽象级别; 最后在最低的抽象级别上给出实现问题的解, 即源代码。在由高级抽象到低级抽象转换的过程中, 要进行一连串的过程抽象和数据抽象, 这就是细化 (refinement)。过程抽象是把完成一个特定功能的动作序列抽象为一个过程名和参数表, 然后通过指定过程名和实际参数调用此过程; 数据抽象把一个数据对象的定义抽象为一个数据类型名, 用此类型名可定义多个具有相同性质的数据对象。在抽象数据类型的定义中还可以加入一组操作的定义, 用以确定在此类数据对象上可以进行的操作。

细化的实质就是分解。在逐步细化中, 特别强调这种分解的“逐步”性质, 即每一步分解仅较其前一步增加少量的细节。这样, 在相邻两步之间就只有微小的变化, 不难验证它们的内容是否等效。事实上, 在传统软件开发中, 逐步细化不仅应用于软件设计, 而且应用于软件分析。它不仅是一种有用的设计策略, 而且已成为问题求解的通用技术。

3. 信息隐藏

早在 1972 年, D. L. Parnas 就提出了把系统分解为模块时应遵守的指导思想, 称之信息隐藏 (information hiding)。他认为, 模块内部的数据与过程, 应该对不需要了解这些数据与过程的模块隐藏起来。只有为了完成软件的总体功能而必须在模块间交换的信息, 才允许在模块间进行传递。

这一指导思想的目的, 是为了提高模块的独立性, 当修改或维护模块时, 减少将一个模块的错误扩散到其他模块中去的机会。应用这一思想, 在软件开发中先后出现了数据封装

(data encapsulation, 指在一个模块中包含一个数据结构和在此数据结构上执行的操作)、抽象数据类型 (abstract datatype, 指一种数据类型, 其中可包含在该类型的实例上执行的操作) 等设计方法。其后, 又在 OOD 中进一步发展为具有继承特性的类 (可以看成是一个支持继承的抽象数据类型) 和对象。

试比较 3 种不同的设计思想: 面向过程的思想、面向功能的思想和面向对象的思想。第一种思想的设计结果是, 各模块的功能可能相互交叉或重叠, 模块间常常存在数据的共享或数据结构的共享, 很难把这些模块移用到其他应用中去。第二种思想设计出来的模块, 各模块的功能单一, 如能将它们与其他模块的数据共享降到最低限度, 就可以在有些应用中重用。利用最后一种思想设计出来的模块是一个个独立的单位, 不仅重用性较好, 而且易于测试、联调和维护。

4. 软件复用

很久以来, 人们就盼望着软件可以复用。使开发人员能充分利用已有的现成构件, 不必一切都从头做起。OO 技术的流行加快了这一理想的实现。今天复用已经成为软件开发, 尤其是软件设计中的一项重要活动, 使软件复用与软件设计结下了不解之缘, 上升为软件设计中的又一基本策略。有些以介绍 OO 开发为主的教材, 甚至取消了传统的“软件设计”一章的章名, 由“软件复用”取而代之, 以突出它的重要性。

7.1.2 软件设计的任务

分析阶段对目标系统的数据、功能和行为进行了建模, 这是软件设计的基础。软件设计的任务, 就是把分析阶段产生的分析模型转换为用适当手段表示的软件设计模型 (见 7.2.1 节)。

不管采用何种软件设计方法, 软件设计一般包括数据设计、体系结构设计、接口设计和过程设计等内容。数据设计将分析阶段创建的信息模型转变成实现软件所需的数据结构; 体系结构设计定义软件主要组成部件之间的关系; 接口设计描述软件内部、软件和接口系统之间以及软件与人之间是如何通信的 (包括数据流和控制流); 过程设计将软件体系结构的组成部件转变成对软件组件的过程性描述。

传统的设计任务通常分两个阶段完成。第一个阶段是概要设计, 包括结构设计和接口设计, 并编写概要设计文档; 第二阶段是详细设计, 其任务是确定各个软件部件的数据结构和操作, 产生描述各软件部件的详细设计文档。每个阶段完成的文档, 都必须经过复审。

7.1.3 模块化设计

模块化设计 (modular design) 由来已久, 目的是按照规定的原则把大型软件划分为一个较小的、相对独立但相互关联的模块。

分解和模块独立性, 是实现模块设计的重要指导思想。

1. 分解

分解 (decomposition) 是处理复杂问题常用的方法。在传统的软件工程中, 在分析阶段

靠分解来画分层 DFD 图；在设计阶段用分解来实现模块化设计。在 OO 软件工程中，靠分解来划分类和对象。不管采用哪种设计方法，都要将系统分解成它的组成部分，成为模块、对象或构件。

1965 年，G. A. Miller 在他的著名文章“奇妙的数字 7 ± 2 ——人类信息处理能力的限度”中指出，普通人分辨或记忆同类信息的不同品种或等级的数量，一般不超过 $5 \sim 9$ 项（即 7 ± 2 ）。这表明，要使人的智力足以管理好程序，坚持模块性（modularity）不仅是一个良好的主意，也是必不可少的措施。正如不分段的长篇文章可能使读者感到头痛一样，大型的单模块软件（monolithic software）不仅可读性差，可靠性也常常难以保证。

对问题求解的大量实验进一步表明，将一个复杂的问题分解为几个较小的问题，能够减小解题所需要的总工作量。用数学公式来表示，可以写成：

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

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

其中， P_1 、 P_2 系由问题 P_1+P_2 分解而得， C 为问题的复杂度， E 为解题需要的工作量。工作量通常用人-年（man-year）或人-月（man-month）来表示。

继续进行分解，问题的总复杂度和总工作量将继续减小，但如果无限地分下去，是否会使总工作量越来越小，最终变成可以忽略呢？不会。因为在一个软件系统内部，各组成模块之间是相互关联的。模块划分的数量越多，各模块之间的联系也就越多。模块本身的复杂度和工作量虽然随模块的变小而减小，模块的接口工作量却随着模块数的增加而增大。如图 7.1 所示，每个软件都存在一个最小成本区，把模块数控制在这一范围，可以使总的开发工作量保持最小。

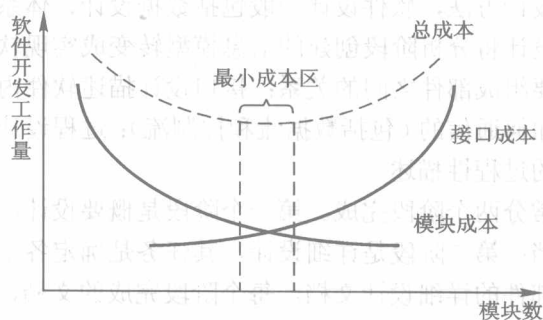


图 7.1 模块数与开发工作量的关系

2. 模块独立性

模块独立性（module independence）概括了把软件划分为模块时要遵守的准则，也是判断模块构造是否合理的标准。坚持模块的独立性，一般认为是获得良好设计的关键。

独立性可以从两个方面来度量，即模块本身的内聚（cohesion）和模块之间的耦合

(coupling)。前者指模块内部各个成分之间的联系，所以也称为块内联系或模块强度；后者指一个模块与其他模块间的联系，所以又称为块间联系。模块的独立性愈高，则块内联系越强，块间联系越弱。C. Myers 把内聚和耦合各划分为 7 类，现分别介绍如下。

(1) 内聚

内聚是从功能的角度对模块内部聚合能力的量度。按照由弱到强的顺序，Myers 把它们分为 7 类，如图 7.2 所示，从左到右内聚强度逐步增强。

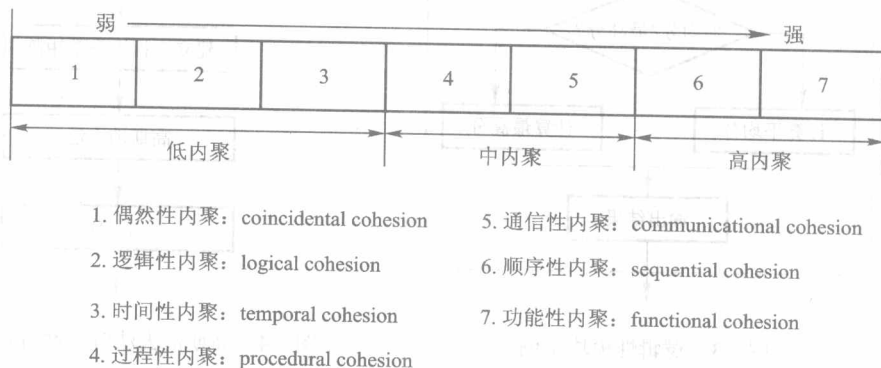


图 7.2 内聚强度的划分

低内聚包括左侧 3 类模块，即：

① 偶然性模块。块内各组成成分在功能上是互不相关的。例如，如果有几个模块都需要执行“读 A”、“写 B”等相同的一组操作，为了避免重复书写，可以把这些操作汇成一个模块，供有关的模块调用。这类模块内部成分的组合纯属偶然，称为偶然性内聚。

② 逻辑性模块。通常由若干个逻辑功能相似的成分组成。例如一个用于计算全班学生平均分和最高分的模块。如图 7.3 所示，无论计算哪种分数，都要经过读入全班学生分数、进行计算、输出计算结果等步骤。实际上除中间的一步需按不同的方法计算外，前、后这两步都是相同的。把这两种在逻辑上相似的功能放在一个模块中，就可省去程序中的重复部分。其主要缺点是，执行中要从模块外引入用作判断的开关量，这会增大块间耦合。

③ 时间性模块。这类模块所包含的成分，是由相同的执行时间而联结在一起的。例如一个初始化模块可能包含“为变量赋初值”、“打开某个文件”等为正式处理作准备的功能。由于要求它们在同一时间内执行，故称为时间性内聚。

中内聚包括图 7.2 中的第 4、5 两类模块：

① 过程性模块。当一个模块中包含的一组任务必须按照某一特定的次序执行时，就称为过程性模块。图 7.4 显示了用高斯消去法解线性方程组的流程。如果把全部任务均纳入一个模块，便得到一个过程性模块。

② 通信性模块。图 7.5 显示了通信性模块的几个例子。这类模块的标志是，模块内部的

各个成分都使用同一种输入数据，或者产生同一个输出数据。它们靠公用数据而联系在一起，故称为通信性内聚。

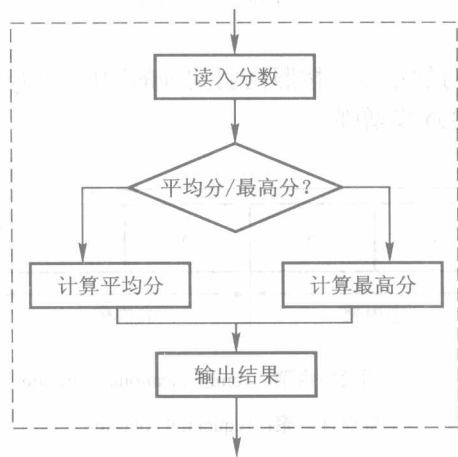


图 7.3 逻辑性模块示例

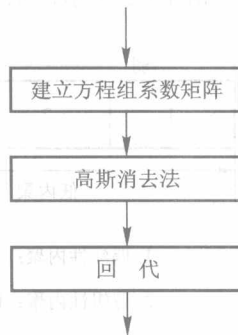


图 7.4 高斯消去法解线性方程组流程

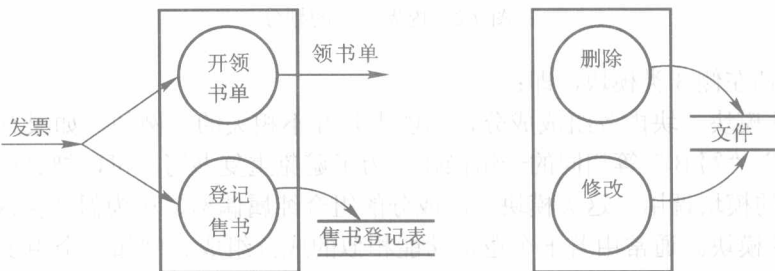


图 7.5 通信性模块示例

图 7.2 中最右端的两类属于高内聚模块。其中包括：

① 顺序性模块。顾名思义，这类模块中的各组成部分是顺序执行的。在通常情况下，上一个处理框的输出就是下一个处理框的输入。例如，把图 7.4 中的前两个任务组合在一起，就可得到一个顺序性模块。

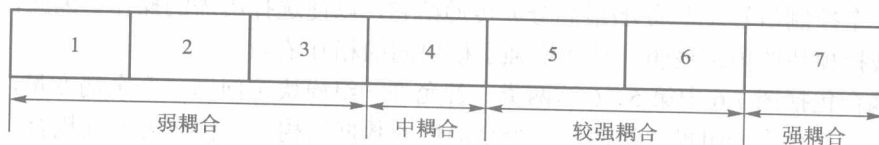
② 功能性模块。这是块内联系最强的一类模块。在这类模块中，所有的成分结合在一起，用于完成一个单一的功能。例如对一个数开平方，求一组数中的最大值，从键盘上读入一行等。在图 7.4 中，如果将每一处理框编制成一个模块，则产生的 3 个模块都是功能性模块。

显然，功能性模块具有内聚高、与其他模块的联系少等优点。“一个模块，一个功能”，

已成为模块化设计的一条准则，也是设计人员争取的目标。当然，其他的高内聚和中内聚模块也是允许使用的，低内聚模块因块内各成分的联系松散，可维护性和可重用性都比较差，在设计中应尽可能避免使用。

(2) 耦合

耦合是对软件内部块间联系的度量。按照 Myers 的划分，也归纳为 7 类，如图 7.6 所示。



- | | |
|------------------------------|----------------------------|
| 1. 非直接耦合: no direct coupling | 5. 外部耦合: external coupling |
| 2. 数据耦合: data coupling | 6. 公共耦合: common coupling |
| 3. 特征耦合: stamp coupling | 7. 内容耦合: content coupling |
| 4. 控制耦合: control coupling | |

图 7.6 耦合强度的等级

弱耦合包括图 7.6 左侧的 3 类情况。在图 7.7 所给出的示例中，模块 1 与模块 2 为同级模块，相互之间没有信息传递，属于非直接耦合。模块 3、4 都是模块 1 的下属模块。模块 1 调用它们时，可通过参数表与它们交换数据。如果交换的都是简单变量，便构成数据耦合（如模块 1、3 之间）；如果交换的是数据结构，便构成特征耦合（如模块 1、4 之间）。

例如在图 7.8 中，“计算应扣款”模块把“用水量”和“用电量”分别传递给“计算水费”与“计算电费”两个模块，然后从这两个模块分别获取“水费”和“电费”，上、下层模块间存在的耦合便是数据耦合。如果事先定义下列的数据结构：

房租水电=房租+用水量+用电量

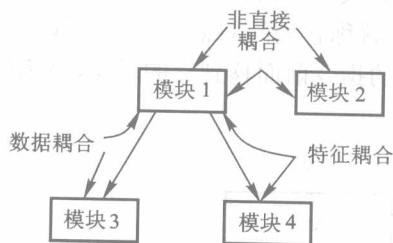


图 7.7 弱耦合示例

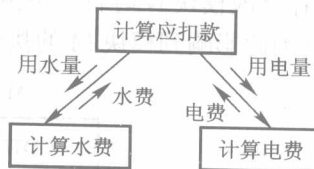


图 7.8 数据耦合示例

并把这一数据结构直接传递给“计算水费”与“计算电费”两个模块，在上、下层之间就构成特征耦合了，如图 7.9 所示。在这种情况下，不仅在模块间传递的数据量要增加，而且当

“房租水电”的数据结构或其格式发生变化时，图 7.9 中的 3 个模块都要作相应的更改。由此可见，特征耦合会使本来无关的模块（例如“计算水费”和“计算电费”）变为有关，耦合强度显然比数据耦合要高。

控制耦合是中等强度的耦合。此时在模块间传递的信息不是一般的数据，而是用作控制信号的开关值或标志量（flag）。以图 7.3 的逻辑性模块为例，当调用这一模块时，调用模块必须先把一个控制信号（平均分/最高分）传递给它，以便选择所需的操作。因此，控制模块必须知道被控模块的内部逻辑，从而增强了模块间的相互依赖。

较强耦合包括图 7.6 中第 5、6 这两类，若允许一组模块访问同一个全局变量，可称它们为外部耦合；若允许一组模块访问同一个全局性的数据结构，则称之为公共耦合。在图 7.10 中，假定模块 C、D、N 均可访问公用区中的某一数据结构。模块 C 首先从公共区中读出数据，然后启动模块 D，对该数据进行计算和更新。如果模块 D 有错误，计算与更新的数据也随之出错，则当模块 N 在以后某个时候读取上述数据并作处理时，就会得出错误的结果。虽然问题是在调用 N 时暴露的，但根源在于 D，这就增加了调试和排错的困难。

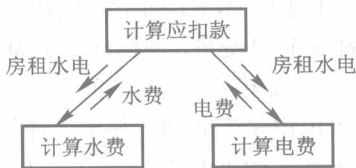


图 7.9 特征耦合示例

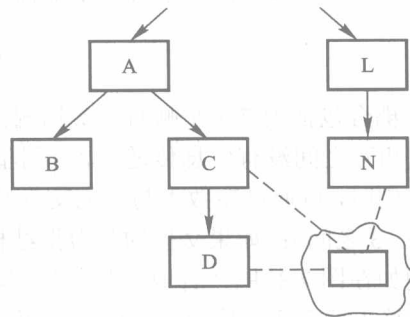


图 7.10 公共耦合示例

最后一类，也是最强的一类耦合是内容耦合。如果一个模块可以直接调用另一模块中的数据，或者允许一个模块直接转移到另一模块中去，就称它们间的耦合为内容耦合。如图 7.11 所示，假定在修改模块 N 时将一条指令从标号为 A 的指令前面移到它的后面，就需特别细心，检查会不会因此影响到模块 M 的执行结果。

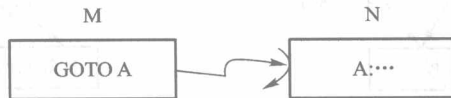


图 7.11 内容耦合示例

耦合越弱，则表明模块的独立性越强。但实际工作中，中等甚至较强的耦合不可能也不必完全禁用。例如 FORTRAN 语言中的 COMMON 区可以令访问它的模块间发生公共耦合，

当然不应滥用。但当在一组模块之间存在较多的公用数据时，使用 COMMON 区往往又使编程比较方便。所以问题不在于禁止使用它们，而是要了解各种耦合的特点与不足，以便在需要使用它们时能预见到可能产生的问题。至于最强类的内容耦合，由于会给维护工作带来很大的困难，有人称之为“病态联系”，故应该尽量不用。

7.2 面向对象设计建模

在基于面向对象分析阶段确定了问题领域的类/对象以及它们的关系和行为的基础上，就可以开始面向对象设计了。它主要考虑“如何实现”的问题，其注意的焦点从问题空间转移到解空间，着重完成各个不同层次的模块设计。因此，它不仅要说明为实现需求必须引入的类、对象以及它们之间是如何关联的，描述对象间如何传递消息和对象的行为如何实现，还需要从提高软件设计质量和效率等方面考虑如何改进类结构和可复用类库中的类。

7.2.1 面向对象设计模型

Pressman 把 OO 设计模型定义成一个金字塔层次结构。图 7.12 显示了怎样从 OO 分析模型导出 OO 设计模型，以及两种模型的相互关系。

如图 7.12 所示，右边的金字塔是一个 OO 设计模型，由系统架构层、类和对象层、消息层、责任层等 4 个层次组成。系统架构层描述了整个系统的总体结构，使所设计的软件能够满足客户定义的需求，并实现支持客户需求的技术基础设施；类和对象层包含类层次关系，使得系统能够以通用的方式创建并不断逼近特殊需求，该层同时包含了每个对象的设计表示；消息层描述对象间的消息模型，它建立了系统的外部 and 内部接口，包含使得每个对象能够和其协作者通信的细节；责任层包含针对每个对象的所有属性和操作的数据结构和算法的设计。

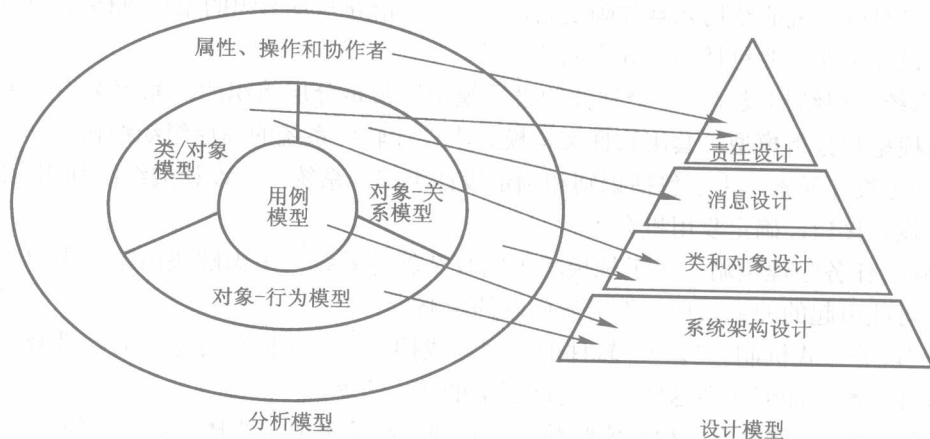


图 7.12 OO 分析模型转换到 OO 设计模型

图 7.12 同时也显示了 OO 分析模型和 OO 设计模型的对应关系。系统架构设计可通过考虑整体客户需求（由用例模型表示）和外部可观察到的事件和状态（对象-行为模型）导出；类和对象设计可由类/对象模型以及属性、操作和协作者的描述映射得来；消息设计可由对象-关系模型导出；责任设计则可利用类/对象模型以及属性、操作和协作者导出。

需要指出，上述金字塔中包含的 4 个层次，都是针对特定的应用或产品而言的。在实际的 OOD 活动中，任何应用（或产品）总是从属于某个领域的，所以在 Coad 和 Yourdon 倡导的 OOD 方法中，强调在应用设计的同时还需考虑该应用所在领域的基础设施（infrastructure）。因此 Pressman 认为，可以想象在上述 4 层次的下方还存在一个称为领域对象（domain object）的层次。作为整个金字塔的基础，该层次可以在人机交互界面、任务管理和数据管理等方面对本领域的应用（或产品）提供支持。

7.2.2 面向对象设计的任务

在 7.1 节已经讲过，设计阶段的主要任务是数据设计、体系结构设计、接口设计和过程设计。在面向对象的设计中，数据和过程被封装为类/对象的属性和操作；接口被封装成为对象间的消息；而体系结构的设计则表现为系统的技术基础设施和具有控制流程的对象间的协作。

正如传统设计可分为概要设计和详细设计两个阶段，OOD 的软件设计也可划分为两个层次：系统架构设计和系统元素设计，分别由系统架构师（system architect）和软件设计师（software designer）完成。下面简要介绍这两个层次的设计任务。

1. 系统架构设计

软件系统架构是指系统主要组成元素的组织或结构，以及其他全局性决策，组成元素之间通过接口进行交互。系统架构包含关于软件系统组织的许多重要决定，例如，从不同抽象层次上选择组成系统的结构元素并确定它们的接口，指导开发组织的架构风格等。具体地说，该层次的设计工作主要包括以下 6 方面的活动：

① 系统高层结构设计。一个软件的设计模型通常是分层组织的，系统架构师需要根据软件需求模型和分析模型，套用软件架构模式来设计软件系统的高层组织结构。

② 确定设计元素。主要包括识别和确定设计类和子系统，将设计类组织到相应的包中，为子系统设计接口，确定复用机会等。

③ 确定任务管理策略。对于规模较大的复杂软件系统，考虑解决由于多用户、并发执行任务等可能引起的冲突或运行性能等问题的策略。

④ 实现分布式机制。当一个软件的不同组成构件位于不同服务器上时，选择支持远程通信的构件，给出如何实现这些构件之间通信的统一方案。

⑤ 设计数据存储方案。根据实际数据库管理系统的类型，选择数据库访问的支持构件，设计类/对象数据的存储、读取、删除或修改等操作的方法。

⑥ 人机界面设计。考虑人机界面的统一要求和规范，确定实现的技术基础和工具等。

系统架构设计是针对整个系统的，其设计结果将影响整个系统，软件设计人员在设计系统中的每个设计元素以及它们之间的交互时都必须遵循系统架构设计文档。

2. 系统元素设计

系统元素包括组成系统的类、子系统与接口、包等。系统元素设计是对每一个设计元素进行详细的设计，主要包括以下设计内容：

① 类/对象设计。类是组成系统的最基本单位，类/对象设计在分析类的基础上对每个设计类的属性及其类型、操作及其算法、接收及发送的消息等进行详细设计。

② 子系统设计。设计和确定子系统以及每个子系统内部组织（包含设计元素以及它们之间的关系）、子系统对应的接口、子系统之间的关联等。

③ 包设计。设计包，将逻辑上相关的设计元素组织在一起。

面向对象设计的过程是循环渐进的，从需求和实现两个角度对设计模型进行逐步完善。通过对设计结果的复审，并伴随着附加的软件分析活动，使设计模型既与软件需求分析模型相一致，也为软件实现提供基础。

7.2.3 模式的应用

为了利用已取得成功的设计结果和经验，提倡在面向对象设计中充分应用设计模式（pattern）。这样做既可以减少工作量，也可以提高设计结果的质量。

1. 模式的定义

模式是解决某一类问题的方法论，也是对通用问题的通用解决方案。其目的是把解决某类问题的方法总结、归纳到理论高度，供其他人员在解决类似问题时参考或直接套用。

Alexander 给出了模式的经典定义：每个模式都描述了一个在某个特定环境中不断出现的问题，然后描述该问题解决方案的核心。通过这种方式，就可以无数次地使用那些已有的解决方案，无须再重复相同的工作。

模式通常区分为不同的领域，建筑领域有建筑模式，软件设计领域也有设计模式。当一个领域逐渐成熟的时候，自然会出现许多模式。

2. 软件模式的分类

就其抽象的级别而言，软件模式可以分为架构模式、设计模式和习惯用法 3 种。

① 架构模式。表示软件系统的基本结构组织方案。它提供了一组预定义的子系统，指定它们的职责，并且包括用于组织其间关系的指导规则。比较常见的架构模式有层次架构模式、MVC 架构模式等。不同的架构模式可能同时适用于一个拟建系统的设计，但各有特色和针对性，在应用场合上并不互相排斥。

② 设计模式。提供对面向对象的具体设计问题的解决方案，使设计的结果具有更良好的可扩展性和重用性。GoF 在《Design Pattern》一书中介绍了 23 个常用的模式。根据其设计功用，可分为构建型、结构型和行为型 3 类，包括桥接模式、工厂模式、组合模式等。

③ 习惯用法 (idiom)。是指针对具体程序设计语言的使用模式, 主要涉及如何用特定方法来解决程序代码编写过程中所遇到的问题, 如何编写更优的程序代码等。Java、C++等程序设计语言都有相应的习惯用法。

7.3 系统架构设计

随着计算机软件的规模越来越大, 系统架构也越来越重要。就像造大楼时打地基一样, 系统架构设计是整个软件的基础, 其设计质量是一个软件开发成功与否的关键。如果软件的系统架构设计得不好, 那么即使软件开发成功, 质量也不会高, 寿命也不会长; 相反, 如果一个软件的系统架构设计得很成功, 既可以提高软件开发质量, 也能大大提高软件开发效率。下面介绍系统架构设计的主要内容。

7.3.1 系统高层结构设计

在进行具体元素设计之前, 首先要确定系统的高层结构。系统高层结构为后续设计提供一个公共的基础框架, 用以承载逐步演进和累加的设计内容。这个基础框架的重要性远远超出框架中的具体内容。如果在没有明确框架的约束下, 先搜集大量杂乱无章的具体内容, 事后重新构筑框架, 其难度和负担将成倍增长。

在设计系统高层结构时, 可以选用架构模式作为模板来定义系统的高层框架。常用的架构模式有层次架构 (layers)、模型-视图-控制架构 (model-view-control, MVC)、管道与过滤器架构 (pipes and filters) 和黑板架构 (blackboard) 等。层次架构是在系统高层结构设计过程中经常选用的架构模式, 下面就以层次架构为例具体介绍, 其他架构模式可仿此进行。

层次架构的基本原则是将系统划分成不同层次。越靠下面的层次, 包含的内容越具有一般性, 或者说与软件需求中特定应用逻辑的关系越松散。这样做的好处是, 提高日后重复利用设计结果的可能性和可操作性。实践中可以结合实际情况, 决定适宜的层数以及层次界定的内涵。

以下给出一种针对中型或大型软件的典型的分层方法, 包括 4 个层次:

- ① 应用子系统层。包括应用程序特有的服务。
- ② 业务专用层。包括在一些应用程序中使用的业务专用构件。
- ③ 中间件层。包括各种复用构件, 例如 GUI 构建器、与数据库管理系统的接口、独立于平台的操作系统服务以及诸如电子表格程序、图表编辑器等 OLE 构件。
- ④ 系统软件层。它包括操作系统、数据库、与特定硬件的接口等构件。

这 4 个层次的关系如表 7.1 所示。

由于前期的工作重点是对问题本身的分析, 因而只需相对明确地界定层次架构的较高层, 即应用子系统层和业务专用层。这两个层次将承载那些与应用逻辑密切相关的要素。前阶段做出的层次界定, 可以在后续设计中得到验证和调整。因为在层次架构中, 较低层次的

界定往往有赖于较高层次对较低层次的具体服务要求。

在初步确立较高层次之后，即可以建立一张反映层次之间的依赖关系的类图。每个层用带有“<<Layer>>”标记的模型元素包来表示，层与层之间用虚线箭头连接，表示调用关系。

表 7.1 层次架构

	层 次	功 能
特殊 ↑	应用子系统层	组成所开发应用的独特应用子系统
	业务专用层	该应用所属业务类型专用的一些可复用子系统
	中间件层	提供实用程序的子系统，为异构环境中分布式对象计算提供独立于平台的服务等
一般 ↓	系统软件层	构成实际基础设施的软件，如操作系统、与特定硬件的接口、设备驱动程序等

7.3.2 确定设计元素

确定设计元素从分析模型出发，对现有分析类的交互进行分析，以确定设计模型元素。确定设计元素的主要工作是确定设计类、子系统以及子系统接口，并找出可能复用的元素。

1. 映射分析类到设计元素

一个分析类可以映射为一个设计类或者多个设计类的简单组合。如果一个分析类很简单，并已代表单一独立的逻辑抽象，就可将其直接映射到设计类。通常在分析类中，与主动参与者连接的边界类、控制类和一般的实体类可以被映射为设计类。

如果分析类的职责比较复杂，其行为很难由单个设计类或者设计类的简单组合承担，也可以将其映射为子系统接口。在后续设计活动中，特定的子系统将在相应子系统接口的封装下实现相应的行为。从子系统的外部看，它和一个设计类在概念上是一样的。通常被动参与者对应的边界类被映射为子系统接口。

2. 确定子系统

子系统实际上是一种特殊的包，这种包具有统一的接口，接口提供了一个封装层，从而使其他模型元素看不到子系统的内部设计。子系统这一概念用于将它和普通包区分开来：普通包是无语义的模型元素容器，而子系统则表示具有与类相似的（行为）特征的包的特定用法。

是否将一组协作的分析类创建为子系统，这取决于该协作是否紧密，是否代表相对独立的功能，以及是否可由单独的设计团队来独立开发。子系统中的元素和协作被一个或多个接口隔离开来，因此子系统的外部客户只能依赖于接口来访问系统中的元素。这样，子系统内部元素的设计就完全脱离了外部依赖关系；虽然设计人员（或设计团队）需要指定接口的实现方式，但他们可以自由地更改子系统的内部设计，而不会影响外部依赖关系。

将系统分为若干个子系统，不仅可以独立开发、配置或交付它们，也可以在一组分布式

计算结点上独立部署它们，还可以在不破坏系统其他部分的情况下独立地进行更改；此外，子系统还可以将系统分为若干单元，以提供对关键资源的安全保护，并且可以在设计时代表现有产品或外部系统。

每个应用软件划分成几个子系统是不确定的，这需要软件设计人员根据实际情况来确定。以下是确定子系统的一些指导性参考原则：

① 对象协作原则。如果某个协作中的各个类仅在相互之间进行交互，并且可生成一组定义明确的结果，就应将该协作的类封装在一个子系统中。

② 可选性原则。如果特定的对象协作代表可选行为，则应将其封装在一个子系统中。

③ 用户界面原则。如果用户界面独立于系统中的实体类（即二者都可以且将独立地变更），则应创建横向集成的子系统：将相关的用户界面边界类归入一个子系统，而将相关的实体类归入另一个子系统。如果用户界面和它所显示的实体类紧密耦合（即一方的变更会触发另一方的变更），则应创建纵向集成的子系统：将相关的边界类和实体类装入同一个子系统中。

④ 参与者原则。将两个不同参与者使用的功能分离到不同的子系统，因为每个参与者可能会独立变更自己对系统的需求。

⑤ 耦合和内聚原则。将耦合度较高的类组织成一个子系统，沿着弱耦合的界线将类分开到不同的系统。在某些情况下，可以将类分成更小的类，使其具有内聚度更高的职责，从而完全消除弱耦合。

⑥ 分布原则。如果必须在不同的结点上执行同一个子系统行为，则需要将该子系统分成更小的子系统。确定必须存在于每个结点上的功能，并创建一个新的子系统，使其拥有该功能，然后相应地在该子系统内分布职责和相关元素。

3. 定义子系统接口

子系统是对一组承担职责的设计元素的统称，子系统接口明确定义这些职责，但不约束职责的实现者及实现方式。通过明确的定义，子系统接口将其使用方法和实现方式彻底分开。借助子系统接口的定义，可以将系统中某一部分的复杂行为和其他的设计内容进行隔离。这样，在降低耦合程度的同时提升了设计模型的整体延展性，增加了组织设计任务的灵活性。子系统接口的定义通常要充分考虑和复用已经存在的设计内容，在完整的基础上力求简单。

子系统接口只有逻辑上的意义，没有物理意义，它说明子系统的使用者和子系统服务的提供者之间共同认可的约定。通常按照以下步骤来确定子系统接口：

① 为子系统确定一个备选接口集。将子系统职责按相关性和耦合度分组，这些组定义了子系统的初始接口集，同时为每个职责定义一个操作（包括参数和返回值）。

② 寻找接口之间的相似点。从备选接口集中，寻找相似的名称、相似的职责和相似的操作。如果几个接口中存在相同的操作，则重新分解接口的要素，并抽取共同的操作来组成一个新接口。同时，注意检查现有的接口，尽可能复用这些接口。

③ 定义接口依赖关系。每个接口操作的参数与返回值都有其各自的特定类型，如果参数是实现某一特定接口的对象，则应定义该接口与它所依赖的接口之间的依赖关系。通过定义接口间的依赖关系，可以为架构设计师提供有用的耦合信息，因为接口依赖关系定义了设计模型中各元素间的主要依赖关系。

④ 将接口映射到子系统。接口一旦确定，就应创建子系统与它所实现的接口之间的实现关联关系。从子系统到接口的实现表明，子系统内部存在一个或多个实现接口操作的元素。随后，当设计子系统时，将会改进这些子系统到接口的实现，并由子系统设计人员来指定子系统中实现接口操作的具体元素。

⑤ 定义接口所指定的行为。如果必须按某种特定顺序来调用接口操作（例如，必须先打开数据库连接，然后才能使用数据库），则可以用活动图或状态图来表示。

定义子系统接口时，首先命名和描述接口，通常在相应类的名称前加上前缀“**I-**”，或将类加上“`<<interface>>`”标记来表述子系统接口，并用简明的文字描述它在系统中的作用；然后，定义子系统的行为，即定义操作集合，取代笼统的职责。职责通常用“//”作为前缀，而表示操作时不需要这样的前缀。操作的命名不能像职责的命名那么随意（接近自然语言），不仅要反映出操作内容，而且必须考虑程序设计的统一约定。子系统接口中对操作的描述应该具体化，至少应说明以下内容：操作的名称、操作的返回值含义及类型、操作使用的参数名称及类型和操作应该做什么（包括关键的算法）。

7.3.3 任务管理策略

为了实现应用软件对多用户、多任务的支持，即满足并行处理的需求，任务管理是在软件设计阶段必须考虑的内容。在 OOA 模型中，对象-行为模型可能包含并发事件的信息。对于不是并发活动的对象（或子系统），不需要并发处理，就可以在同一个处理器硬件上实现。而某些事件可能同步作用于多个对象（或子系统），则必须考虑同步措施。

面对并行需求，主要有以下 3 种解决方案：

- ① 多处理器方案。将并发子系统分配到不同的处理器。
- ② 操作系统方案。将并发子系统分配到相同的处理器并由操作系统提供同步控制。
- ③ 应用程序方案。应用软件负责在适当的时间从一个代码分支切换到另一个代码分支。

在进行系统架构设计时，架构师必须考虑有哪些并行需求，采用哪种技术来解决的问题。下面介绍两种实现并行需求的技术：引进任务管理部件以及基于进程和线程的控制。

1. 引进任务管理部件

在 OOD 中引进任务管理部件，有两点原因。一是在多用户、多任务或多线程操作系统上开发应用程序的需要，二是在通过任务描述目标软件系统中各子系统间的通信和协同时，引入任务概念能简化某些应用的设计和编码。Coad 和 Yourdon 建议采用如下设计管理并发任务对象的策略：

- ① 确定任务的特征。

② 定义一个协调者任务和与之关联的对象。

③ 集成其他任务和协调者。

确定任务的特征通常是由理解任务如何被激活开始的。事件驱动任务和时钟驱动任务是最常见的两类任务，均由中断激活，但是前者来自外部的中断源（如某个传感器），后者来自系统时钟。

具体而言，任务管理部件的设计一般遵循如下的步骤与策略：

① 识别由事件驱动和时间驱动的任务。有些任务是由事件或时间驱动的。事件驱动的任务通常完成通信工作，例如，与设备、屏幕上的窗口、其他任务或处理器通信。这类任务通常的工作流程为：任务处于睡眠状态，等待事件；一旦接到事件触发的中断就唤醒该任务，接收数据并执行相应的操作；该任务重新回到睡眠状态。时间驱动的任务是指按一定时间周期激活的任务。例如，PLC 计数器每隔一段固定的时间将数据传输到工控机，大屏幕每隔 20 秒读取实时数据并显示出来等都是时间驱动的任务。

② 识别关键性任务、任务优先级以及任务管理类。关键性任务是指对整个系统成败起重要作用的任务，这些处理要求有较高的可靠性。任务优先级能根据需要调节实时处理的优先级次序，保证紧急事件能在限定的时间内得到处理。任务管理类是为了实现而引入的专门用于管理和协调其他任务的任务。当任务达到 3 个或 3 个以上时，应增加一个任务管理类。

③ 定义任务。说明任务的名称、描述任务的功能、优先级任务与其他任务的协同方式以及任务的通信方式（如采用终端输入、邮箱、缓冲区或信号量等）。

④ 必要时要在 OOD 中扩充有关任务的类及对象，调整原有的语法成分，以适应任务定义的要求。例如，当某个外部服务涉及多个任务时，分离此服务并重新命名，使每个服务对应一个任务，对包含任务协同、任务通信的外部服务，应在外部服务的规格说明中扩充协同和通信的协议描述。

[例 7.1] 举例说明显像管生产监测系统中实时数据采集子系统的一个任务对象。

[解] 以实时翻页任务为例，可给出如下的定义：

任务名称：实时翻页。

描述：此任务在指定的时间间隔内读出最新的半小时数据。

协同方式：时钟驱动，间隔 20 秒。

通信方式：从数据库中读取最新的半小时数据。

2. 基于进程和线程的控制

当操作系统提供多任务时，常见的并行单元为进程。进程是由操作系统提供、支持和管理的实体，而操作系统提供执行程序的环境。进程为其应用程序提供可独占使用的内存空间，执行它的线程，提供与其他进程进行消息收发的方式等。

目前大多数操作系统都提供进程的轻量级替代物——线程。通过线程，可以在进程中对并行作出更详细的定义。每个线程都属于单个进程，一个进程中的所有线程共享该进程所占

用的内存空间以及其他资源。通常要为每一个线程分配一个需执行的动作流程。

(1) 进程和线程建模

需要为系统所需的每个独立控制流创建进程或线程，可以用主动类来建模进程或线程。主动类是指拥有自己的执行线程而且能发起控制活动的类，主动类能与其他主动类并行地执行。

分别用“<<process>>”和“<<thread>>”来标识进程和线程。进程间的通信是依赖关系。应用程序中如果只有一个进程，无须对它的进程建模。对于多进程或线程的应用，为进程或线程建模显得很重要。

对进程建模可采用类图或构件图。图 7.13 是对“选课系统”的进程建模的类图。图中的进程和线程用类来表示。独立的进程间用依赖关系联系，如果有线程，则用特殊的关联关系——组合关系（参见 7.4.3 节）来表示，因为线程在进程外无法存在。

在图 7.13 中，StudentApplication 进程包含了用户界面处理和与业务过程的交互，对于当前选课的每个学生，该进程有一个实例；CourseRegistrationProcess 进程封装了对选课的处理，对于当前选课的每个学生，该进程有一个实例；CourseCatalogSystemAccess 进程负责管理对遗留系统的访问，它是独立的进程，可被选课的用户共享，它存储最近读取的相关信息以提高性能；CourseCatalogSystemAccess 进程中的独立线程 CourseCache 和 OfferingCache 被用来异步地读取遗留系统的信息，这样可以缩短响应时间。

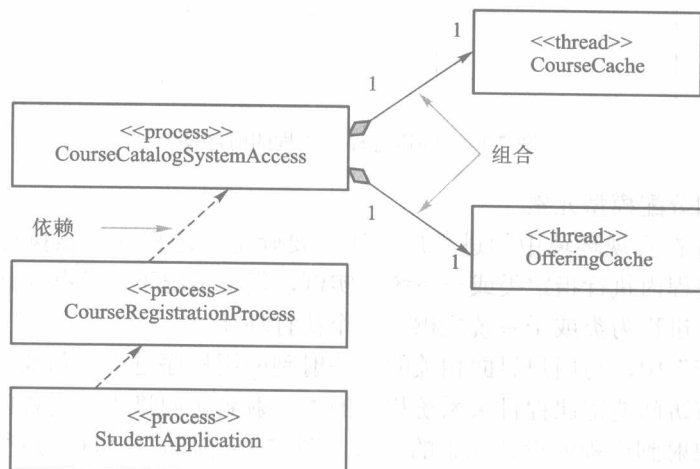


图 7.13 选课系统的进程建模

(2) 确定进程的生存周期

每个进程或线程都必须进行创建和销毁。在单进程架构中，进程在应用程序开始时创建，在应用程序结束时销毁。在多进程架构中，通常新进程（或线程）是在应用程序开始时，从

操作系统创建的初始进程中产生或派生而来，这些进程必须显式销毁。

由设计人员确定和记录进程创建与进程销毁的事件序列以及创建和删除的机制。

[例 7.2] 在“选课系统”中启动一个主要进程，其职责是协调整个系统的行为。

[解] 该进程依次产生许多下级控制线程来监视系统设备和来自课程目录系统的事件。这些进程和线程的创建可用 UML 中的主动类表示，有关的创建活动可用图 7.14 的时序图表示。

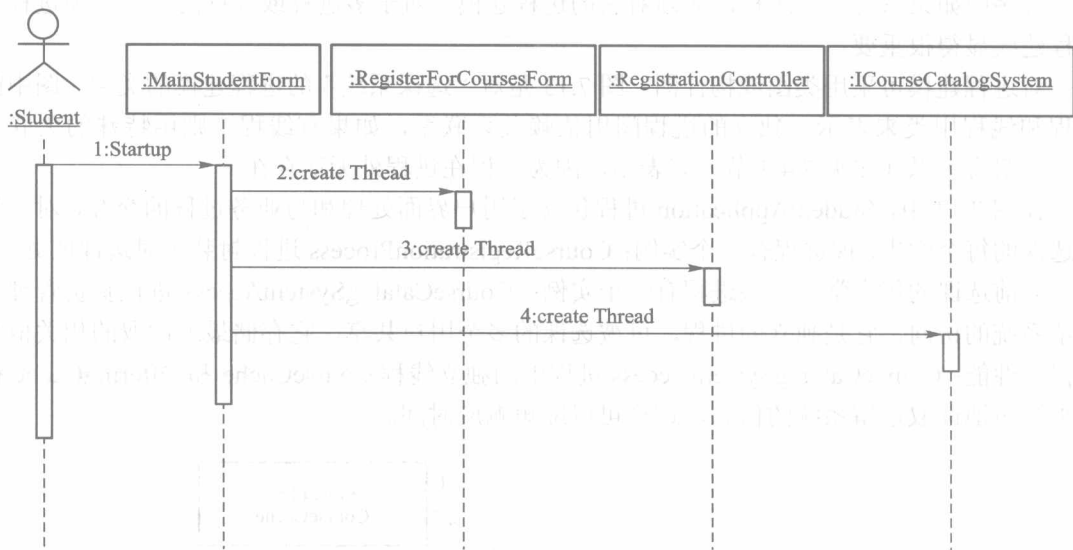


图 7.14 创建进程和线程的时序图

(3) 在进程间分配模型元素

在定义了运行在实现环境中的进程后，还需要确定在进程内应该执行的进程类和子系统。至少在一个进程内执行指定类或子系统的实例，很多情况下，这些实例有可能存在于多个不同的进程中。进程为类或子系统提供了一个执行环境。

在“选课系统”中，与用户界面相关的类映射到应用程序进程；与业务服务相关的类映射到控制进程；与访问遗留课程目录系统相关的类映射到访问进程。进程间的关系（用依赖关系建模）支持映射到进程的设计元素的关系。图 7.15 显示了“选课系统”进程视图的一部分。

7.3.4 分布式实现机制

所谓分布式应用是指应用程序的不同部件被安装在多个通过网络连接的计算机上，系统运行时不同计算机上的应用部件相互协作，提供应用服务。随着网络的普及，目前大部分应用系统都有分布式的需求。例如，C/S 系统被描述为客户-服务器体系结构，即存在专门的网

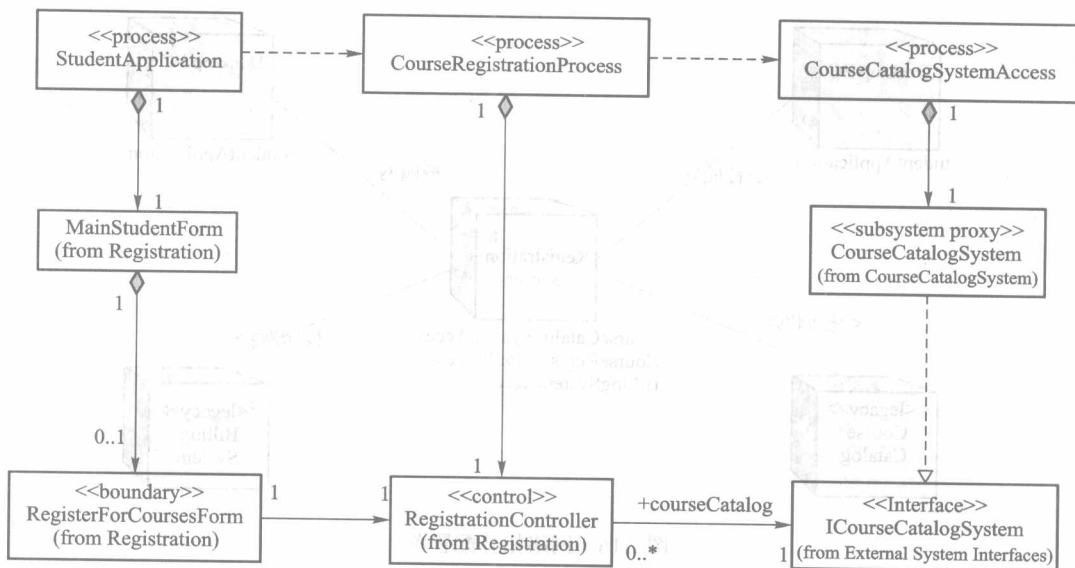


图 7.15 设计元素与进程的关系

络处理器结点，分别称为客户机和服务器。客户机是服务器提供的服务的使用者，而服务器通常同时向数台客户机提供服务，所提供的服务有数据库服务、安全性服务或打印服务等。此外，还有三层体系结构、胖客户机体系结构、胖服务器体系结构、点对点体系结构、多层体系结构等，都是常用的分布式体系结构。

为了实现应用的分布式结构，系统架构师需要完成以下工作：确定网络拓扑配置，将设计元素分配到网络结点，设计分布处理机制。

1. 确定网络拓扑配置

网络的拓扑结构、网络上处理器与设备的性能和特征将决定系统能够具备的分布性质与分布程度。定义网络配置时应获取如下信息：

- ① 网络的物理布局（包括位置）。
- ② 网络的结点及结点的配置与性能。
- ③ 网络中的各网段带宽。
- ④ 网络中的冗余路径（容错能力）。
- ⑤ 结点的分类，例如，终端用户使用的工作站结点，提供处理的服务器结点，用于开发与测试的特殊配置以及其他专用处理器等。

图 7.16 是“选课系统”的网络拓扑图。主要的业务处理硬件是选课服务器，它与两个遗留系统的主机相连。该图还显示了在结点上执行的进程和线程。

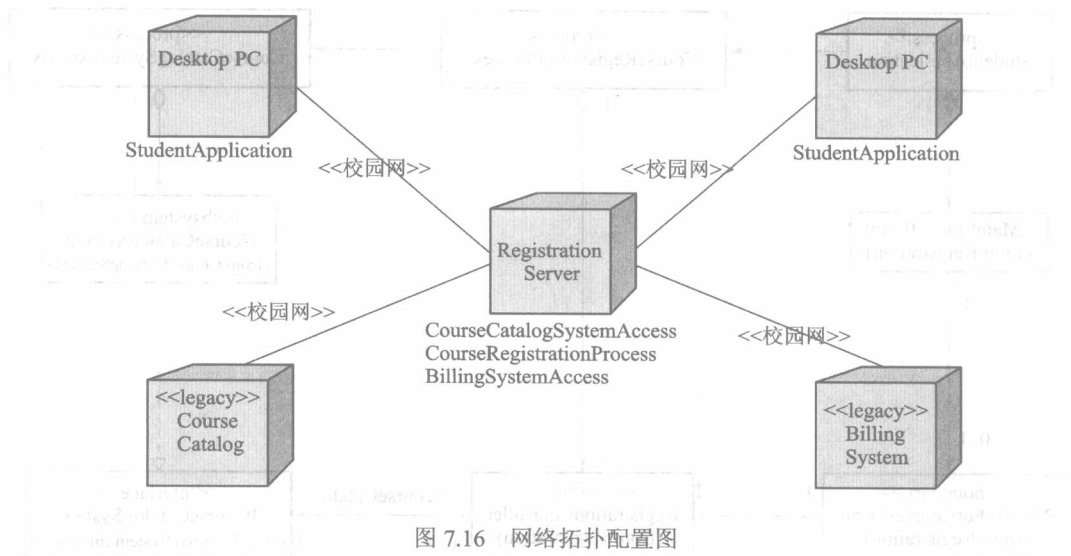


图 7.16 网络拓扑配置图

2. 将系统元素分配到结点

在确定将哪些元素映射到哪些结点时，需要考虑以下因素：

- ① 结点容量（指内存量和处理能力）。
- ② 通信介质带宽（总线、LAN、WAN）。
- ③ 硬件与通信链路的可用性、重选路由。
- ④ 对冗余与容错能力的要求。
- ⑤ 响应时间要求。
- ⑥ 吞吐量要求。

基于使跨网络通信量最小化的目的，将元素分配给结点；应将大量交互的元素分配到相同的结点中；而交互频率较低的元素则可驻留在不同的结点中。在两个或更多结点上分布进程必须同时考虑进程间通信模式。若未仔细考虑进程和结点边界，进程间通信工作负载有可能抵消由于工作负载分布带来的所有好处。

3. 设计分布处理机制

实现分布在不同网络结点上的应用逻辑之间的交互，需要底层类库的支持。这就要考虑是在 Microsoft .NET 平台上还是 Java 平台上实现。下面介绍的方法是基于 Java 类库来实现分布处理的。

基于 Java 的分布处理是通过远程方法调用（remote method invocation, RMI）来实现的。基于 RMI 的应用程序由 Java 对象构成，这些对象可以实现与对方的实际位置无关的相互调用。换言之，一个 Java 对象可调用另一个远程结点上的 Java 对象的方法，整个过程和调用同一台计算机上的某个 Java 对象的方法一样。

采用 RMI 实现分布处理机制的步骤如下：

(1) 引入可直接利用的类库

实现分布处理机制，需要使用 `java.io` 和 `java.rmi` 包中的类。将该包中相关的类加入层次架构中的中间件层。架构师必须熟悉这些用来支撑分布处理机制的类。

参与实现分布处理机制的类主要包括以下内容：

- ① **Naming**：用于寻找分布在异地的对象，每一“地点”有一个此类的实例。
- ② **Serializable**：一个 Java 接口。异地之间作为参数传送的对象必须实现这个 Java 接口。
- ③ **Remote**：一个 Java 接口。被分布到异地的对象的类必须（直接或间接）实现这个 Java 接口。对于实现 Remote 接口类，环境会建立相应的 `remote stub` 和 `remote skeleton`，存根(stub)是远程对象在客户端的代理，它将 RMI 调用传递给服务器端的骨架(skeleton)，后者负责将该调用传递给实际的远程方法。

④ **UnicastRemoteObject**：支撑创建和导出被分布到异地的对象。

(2) 建立一些带有“<<role>>”标识的类，代表实际设计元素

① <<role>> **SampleDistributedClass**：这个类代表需要被远程访问的设计元素，简称分布类。

② <<role>> **SampleDistributedClassClient**：这个类代表访问分布类的设计元素，简称分布类的客户。

③ <<role>> **SamplePassedData**：在异地消息传送中，这个类代表那些被作为参数传递的对象所对应的设计元素，可能不止一个。

④ <<interface>> **ISampleDistributedClass**：是一个 Java 的接口，代表被分布到异地的分布类在本地的“代言人”。

(3) 描述分布机制的静态结构

基于 RMI 的分布处理机制的静态结构如图 7.17 所示。

7.3.5 数据存储设计

对于实例数据需要持久保存的类，需要设计一种统一的存储、读取和修改数据的方法。本节描述一种典型的基于 Java 实现的数据存储机制。

[例 7.3] 利用 Java 机制实现数据的存储管理，实现将类实例数据存储到关系数据库系统(RDBMS)中。

[解] 可分作以下 4 步进行：

第一步，引入相关类库。

用 JDBC 实现持久性机制，需要使用 `java.sql` 包中的类。将 `java.sql` 包中的相关类加入中间件层。

参与实现持久性机制的类主要包括以下内容：

- ① **DriverManager**：用于建立数据库连接。
- ② **Connection**：用于保持与数据库之间的连接。

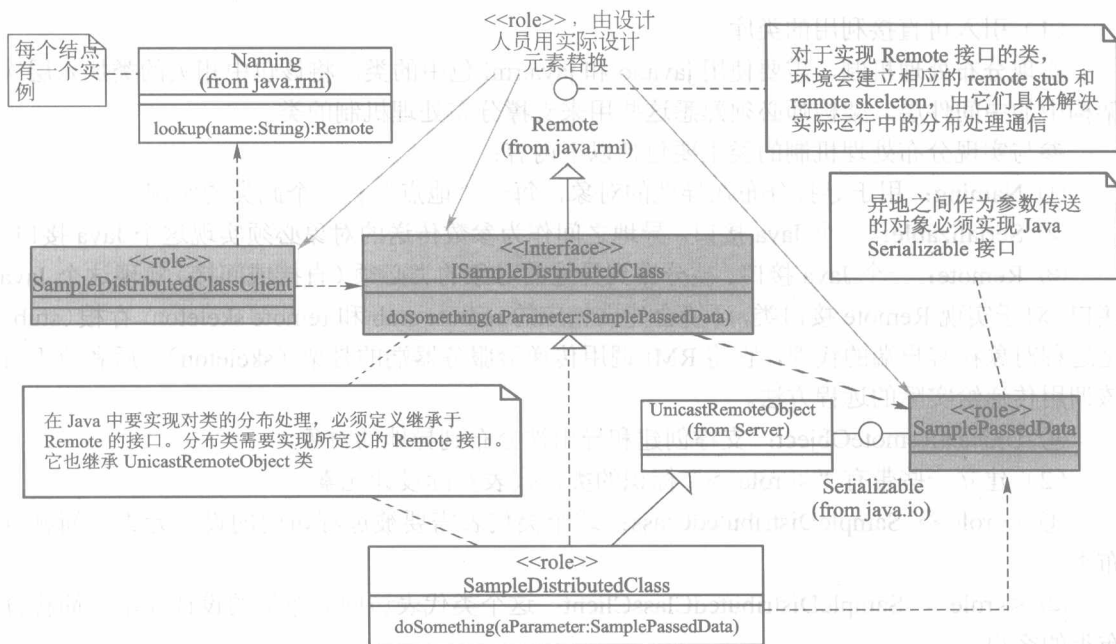


图 7.17 基于 RMI 的分布处理机制

③ **Statement**: 用于直接和数据库进行交互。

④ **ResultSet**: 用于记录 SQL 查询返回的结果。

第二步, 构造一些具有 `<<role>>` 标识的类, 代表实际设计元素。

主要包括:

① `<<role>> PersistentClass`: 这个类代表需要持久保存的设计元素。

② `<<role>> PersistentClient`: 这个类代表在持久性相关协作中的交互者所对应的设计元素。

③ `<<role>> PersistentClassList`: 这个类用于返回一组作为数据库查询结果的 `PersistentClass` 的对象。

④ `<<role>> DBClass`: 在 JDBC 中使用 `DBClass` 来读写持久性数据, `DBClass` 负责利用 `DriverManager` 类访问数据库。打开数据库连接之后, 用 `Connection` 创建 SQL 语句, 通过 `Statement` 与数据库进行交互。`DBClass` 负责创建 `PersistentClass` 的实例, `DBClass` 了解 OO 与 RDBMS 之间的映射, 能够代理系统与 RDBMS 进行交互。每个 `PersistentClass` 有一个相应的 `DBClass` 负责其持久性事宜。

第三步, 描述持久存储机制的静态结构, 如图 7.18 所示。

其中, `ResultSet`、`Statement`、`Connection` 和 `DriverManager` 等 4 个类是 `java.sql` 提供的类, 其他有 `<<role>>` 标记的类 (`DBClass` 等) 是由设计人员确定的实际设计元素。

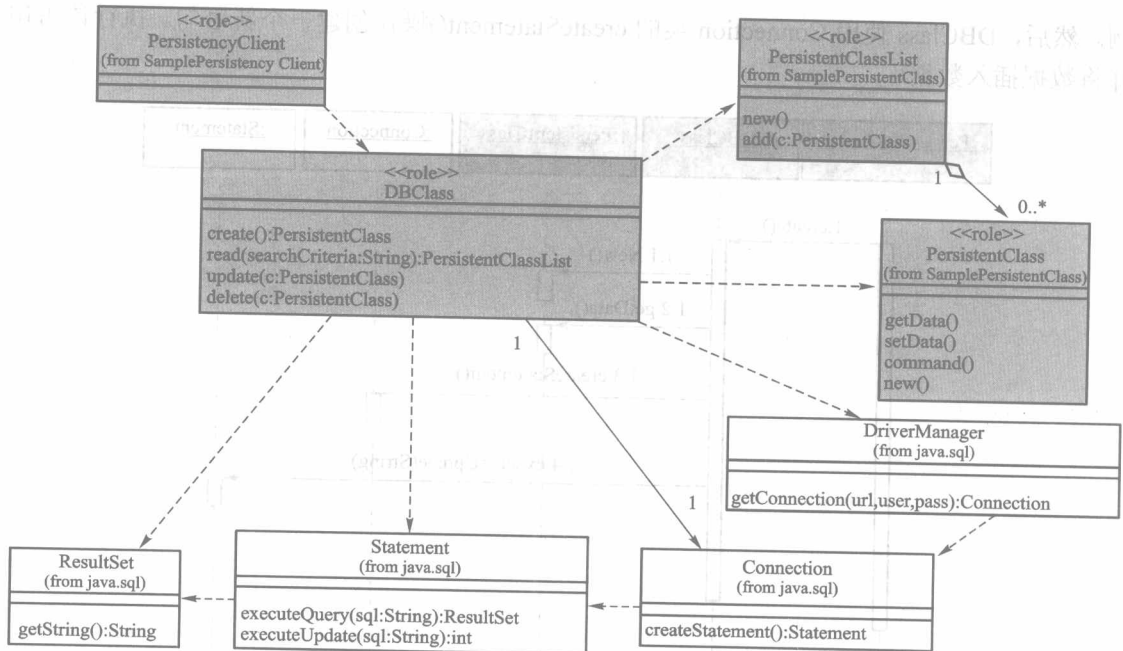


图 7.18 持久存储机制的静态结构

第四步，描述机制的典型应用场景。

典型的数据库操作就是增、删、查、改，以下分别列出相应的时序图。

初始化必须在访问任何持久类之前进行，如图 7.19 所示。

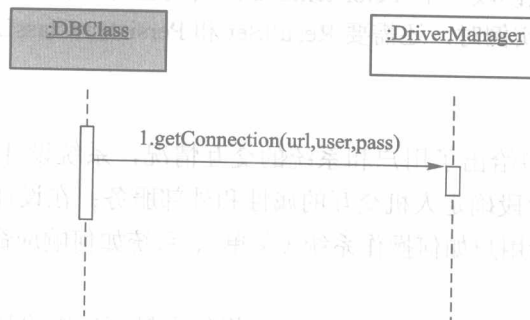


图 7.19 初始化与数据库的连接

为了初始化与数据库的连接，DBClass 必须通过提供数据库的位置（参数 url）、用户（参数 user）和口令（参数 pass）调用 DriverManager 的 getConnection() 操作，建立数据库连接。

图 7.20 描述创建一个 PersistentClass 实例对应的消息传递序列。为了创建一个新类，PersistencyClient 要求 DBClass 创建该新类。DBClass 用默认值创建 PersistentClass 的一个新实

例。然后，DBClass 使用 Connection 类的 createStatement()操作创建一个新语句。执行该语句并将数据插入数据库。

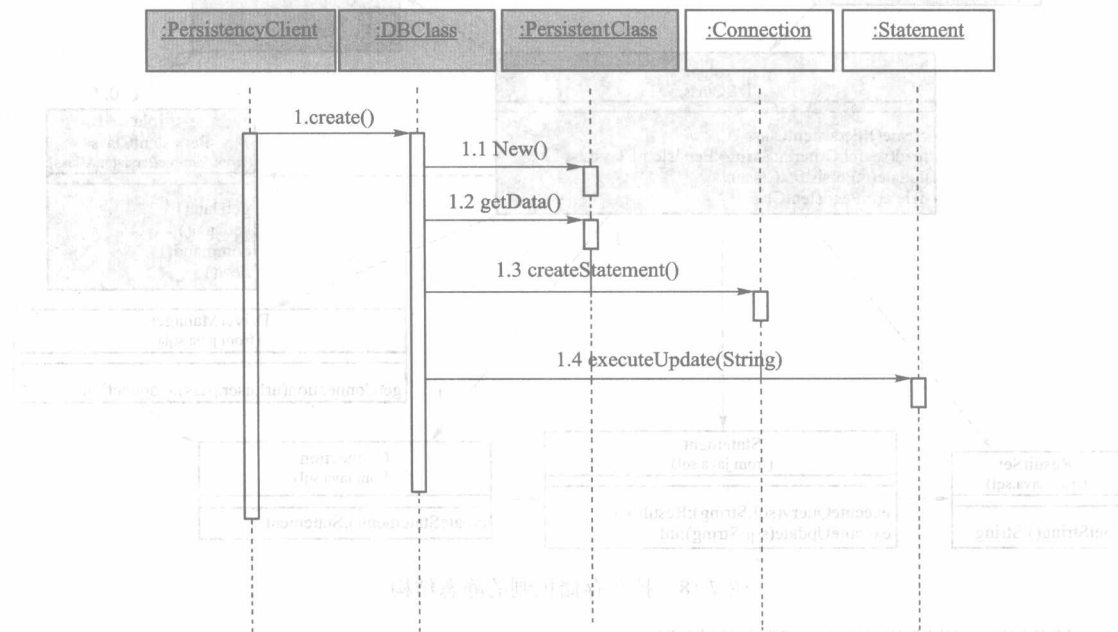


图 7.20 创建 PersistentClass 的实例

同样，删除、更新或读取一个 PersistentClass 实例也可以画出对应的消息传递序列。其中，读取 PersistentClass 实例时，还需要 ResultSet 和 PersistentClassList 类的对象。

7.3.6 人机交互设计

需求阶段的用例模型给出了用户和系统的交互情况，系统设计阶段需以此考虑人机交互。一般在需求和分析阶段确定人机交互的属性和外部服务；在设计阶段则给出有关人机交互的所有系统成分，包括用户如何操作系统（菜单）、系统如何响应命令、系统显示信息与报表的格式等。

虽然如今已有很多可视化开发工具，并且能提供大量可复用的基础图形（如窗口、菜单、对话框等）类库用于用户界面设计，但要设计一个用户满意的人机交互界面仍不是一件很容易的事。

1. 根据用户特点设计不同界面

不同类型的用户要求亦不相同，一般可按技术熟练程度、工作性质和访问权限对用户进行分类，尽量照顾到所有用户的合理要求，并优先满足特权用户的需要。各类用户的工作流

程与习惯也不相同；通常在设计一个新命令系统时，应参考优秀的商品软件，尽量遵循用户界面的一般原则和规范，先根据用户分析结果确定初步的命令系统，然后优化。

命令系统的方式可以是若干菜单，也可以是一组按钮。优化命令系统时，首先应考虑命令的顺序，常用的命令居先，命令的顺序应与用户工作习惯保持一致；其次，要根据外部服务之间的聚集关系组织相应的命令，使总体功能对应父命令，部分功能对应子命令；然后，应充分考虑人类记忆的局限性（即所谓“ 7 ± 2 ”原则或“ 3×3 ”原则），最好把命令系统组织为一棵两层的三叉树；最后还应尽可能减少完成一个操作所需的鼠标动作（如拖曳和击键等），并为熟练用户提供操作的捷径。

2. 增加用户界面专用的类与对象

用户界面专用类的设计通常与所选用的图形用户界面有关。目前流行的 Windows、Macintosh、X Window 和 Motif 等 GUI，通常都依赖于具体的平台，在字形、字体、字号、坐标系和事件处理等方面有些差异。为此，首先需利用类结构图来描述各窗口及其分量的关系；其次，需为每个窗口类定义菜单条、下拉式菜单和弹出式菜单，同时定义所必需的操作，完成菜单创建、高亮度显示所选菜单项及其对应动作等功能，以及将要在窗口内显示的所有信息。必要时还需增设诸如画图标、在窗口中快速选项、选字体和剪贴等专用类。最后，还要为每个类定义控制子类对象的方法。

3. 利用快速原型演示，来改进界面设计

为人机交互部分构造原型，是界面设计的基本技术之一。如有可能，应为用户演示界面原型，让他们直观感受界面的设计，并评判系统是否功能齐全，是否方便好用。

7.4 系统元素设计

如果把系统架构设计比作绘制房子的平面图，则系统元素是该房子内的房间和家具摆设，平面图刻画了每个房间和家具的用途，以及房间和房间、房间和外部环境间连接的机制；而系统元素设计则着重于描述建造每个房间和家具所需的细节。因而，系统元素设计的重点在于如何实现相关的类、关联、属性与操作，定义实现时所需的对象的算法与数据结构。

7.4.1 子系统设计

子系统设计主要针对子系统内部所包含的设计元素及其交互。子系统设计的具体步骤包括：将子系统行为分配给子系统元素，描述子系统元素和说明子系统依赖关系。

1. 将子系统行为分配给子系统元素

子系统的外部行为是通过它所实现的接口来定义的。当子系统实现了某个接口时，就必须实现该接口定义的每一个操作；反过来，这些操作会由子系统所包含的设计元素（即设计类或子系统）上的操作来实现。

子系统的设计元素，其协作关系可用说明子系统行为的时序图来描述。设计子系统的

内部行为时，对子系统实现的接口进行的每一个操作，都可以画一个或者多个相应的子系统交互时序图。这种内部的时序图显示了哪些类实现此接口，需要在内部进行什么操作以提供子系统的功能，以及哪个类从子系统发出消息。创建这些接口实现图时，可能需要创建新的类和子系统来执行所需的行为。对于每个接口操作，确定在当前子系统中执行该操作所需要的类/子系统，如果现有类/子系统无法提供所要求的行为，则需要创建新的类/子系统。

子系统交互图用于说明子系统的行为是如何通过子系统元素间的交互来实现的。将访问子系统的对象发出的消息传给为子系统定义的子系统代理<<subsystem proxy>>类，消息对应于接口的操作，接着子系统的代理类转发消息，执行与其他子系统设计元素的调用操作。

这里还是以“选课系统”为例，介绍子系统内部交互。前章曾介绍过“选课系统”的用例分析，当时只显示了子系统接口的交互。现在进一步介绍课程目录子系统的内部交互情况。

[例 7.4] 在“选课系统”中，课程目录是一个重要的子系统。试画出其时序图。

[解] 其时序图如图 7.21 所示。

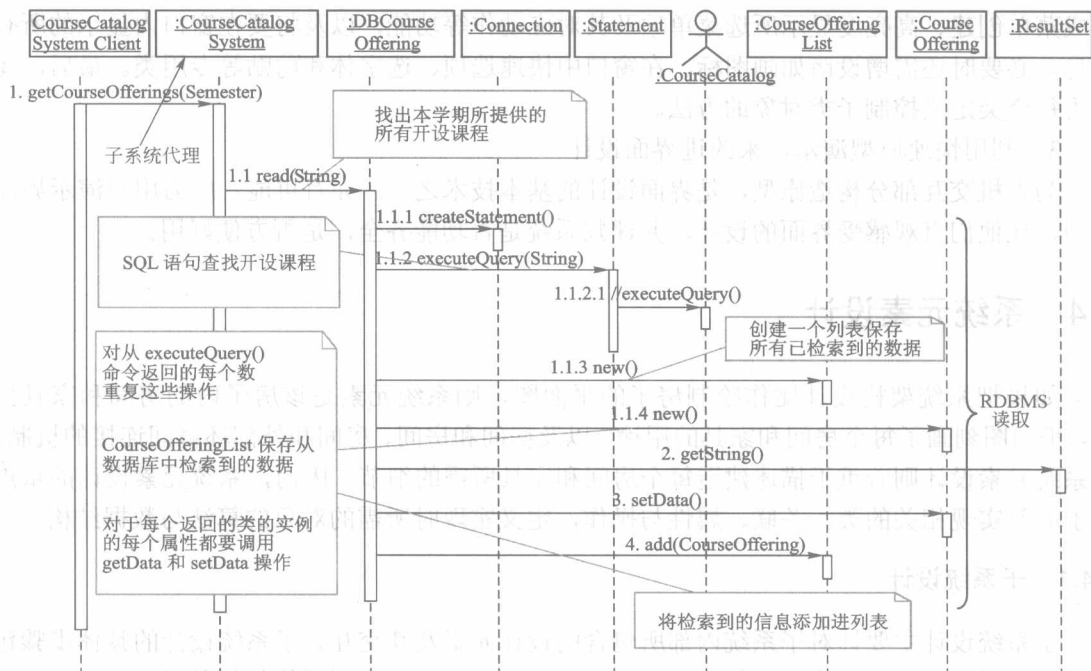


图 7.21 课程目录子系统时序图

由图 7.21 可见，在课程目录子系统的内部是如何实现接口定义的 `getCourseOfferings` 的操作的。由于存储开设课程的遗留系统是 RDBMS，因此，图 7.21 所示的时序图也说明了在 RDBMS 上的数据存储与访问是怎样在设计中实现的。

子系统的代理类 `CourseCatalogSystem` 实际上实现了接口；为了读取开设课程，课程目录子系统的代理类要求 `DBCourseOffering` 类返回指定学期的开设课程，`DBCourseOffering` 利用 `Connection` 类的 `createStatement()` 操作创建新的语句，语句被执行，数据以 `ResultSet` 对象返回，`DBCourseOffering` 然后创建开设课程实例的列表 `CourseOfferingList`，写入取回的数据并返回给客户。

2. 描述子系统元素

可以创建一个或者多个类图来表示子系统包含的元素以及它们的关系。另外，也可以用状态图来表现子系统可能出现的状态及其变迁。

图 7.22 显示了“选课系统”中课程目录子系统内部的类及类之间的关系。类 `CourseCatalogSystem` 与 `DBCourseOffering` 相互协作，来读取和写入数据库中的持久性数据，`DBCourseOffering` 负责访问 JDBC 数据库，一旦数据库的连接打开，`DBCourseOffering` 产生 SQL 语句，它利用 `Statement` 类来执行，查询的结果在 `ResultSet` 对象中返回。时序图中子系统元素间的协作，在这里演化为参与类图中设计元素的关系。

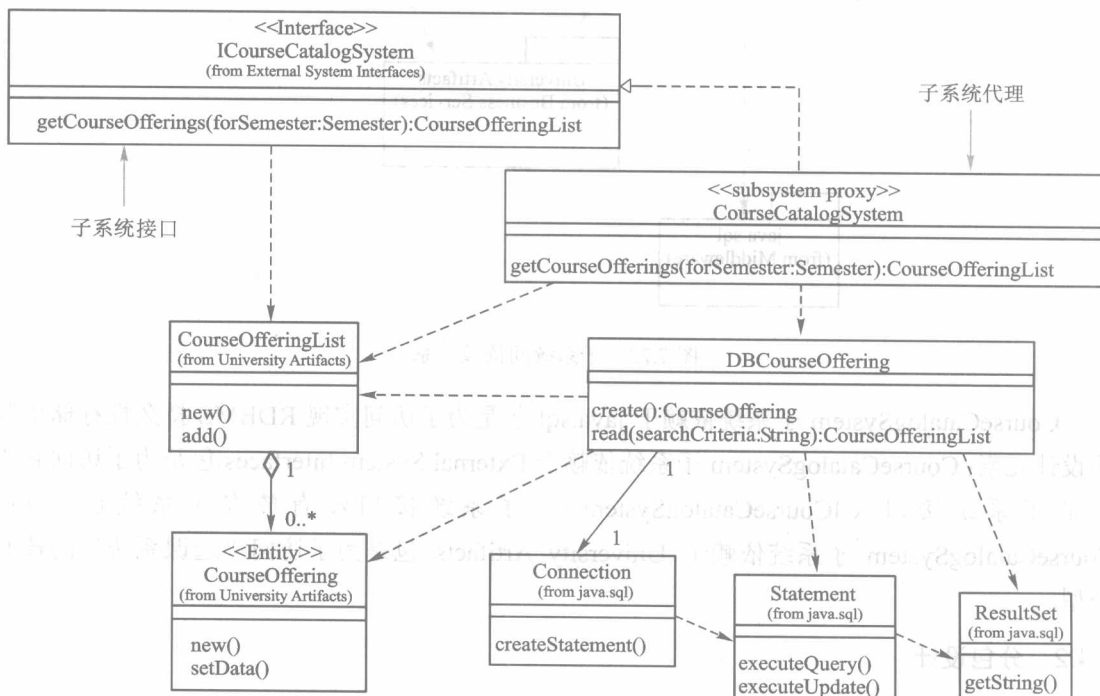


图 7.22 课程目录子系统内部类图

3. 说明子系统的依赖关系

当子系统包含的某个元素使用了另一个子系统中的某个元素行为时，就在它们所属的子

系统间创建了依赖关系。如果一个子系统的模型元素引用了另一个子系统的模型元素，则设计人员不能轻易移去该模型元素或者将该模型元素的行为重新分配给其他元素。

创建依赖关系时，还要确保子系统和接口之间没有循环的依赖关系，子系统不能既实现一个接口而又依赖于该接口。

图 7.23 显示了课程目录子系统与其他设计元素的依赖关系。这些依赖关系与在前面子系统类图中出现的类的关系是对应的。

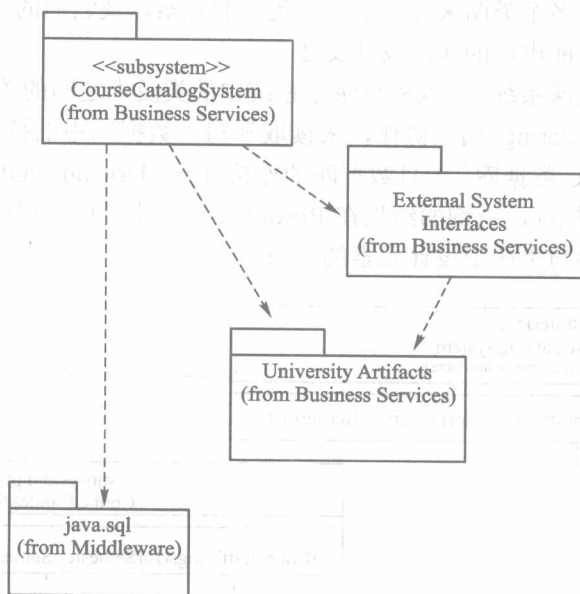


图 7.23 子系统间依赖关系

CourseCatalogSystem 子系统依赖于 java.sql 包是为了访问实现 RDBMS 持久性存储机制的设计元素；CourseCatalogSystem 子系统依赖于 External System Interfaces 包是为了访问它本身的子系统接口（ICourseCatalogSystem）（子系统接口没有放在子系统包中）；CourseCatalogSystem 子系统依赖于 University Artifacts 包是为了访问“选课系统”的核心类型。

7.4.2 分包设计

为了便于理解，在层次结构设计的基础上划分更细的组织单元，一种常用的方法是把设计元素分组放入特定的包中。分包的目的是使设计元素更有秩序，呈现出更明显的高内聚、低耦合特征。实践中，不同的包允许个人或团队相对独立地展开后续的设计和实现工作。包之间的松散耦合通常会使得大型或复杂系统的开发变得更为容易。

1. 分包的原则

可以参照下面的原则将设计元素划分到不同的包。

(1) 将边界类打包

在将边界类分发给各个包时，可以应用两种不同的策略。如果系统接口可能被替换，或者可能会发生较大的更改，就应将接口与设计模型的其他部分隔离开；如果接口不易被替换或更改，则应对系统服务进行分包，将边界类和功能相关的实体类及控制类放置在一起。这样，如果特定的实体类或控制类发生变化，就很容易看出哪些边界类受到影响。

对于在功能上与任何实体类或控制类都不相关的必选边界类，应将它们和属于同一接口的边界类一起放置在单独的包中。

(2) 将功能相关的类打包

功能上相关的类通常放在一个包内。在判断两个类是否在功能上相关时，可以应用以下四项实用标准（按重要性递减顺序排列）：

① 如果一个类的行为和（或）结构的变化使得另一个类也必须相应地变化，这两个类就在功能上相关。例如，如果向实体类 `Order`（订单信息）添加一项新属性，则很有可能使控制类 `OrderAdministrator`（订单信息管理）必须进行更新。所以，它们属于同一个包。

② 从某个类开始（例如一个实体类），检查从系统中将该类删除所带来的影响，这样就可以查明一个类与另一个类是否在功能上相关。例如，有两个控制类：`OrderAdministrator`（订单信息管理）和 `OrderRegistrar`（订单信息注册），这两个控制类都用于为订单处理服务建模，所有的订单属性和关系都由实体类 `Order` 存储，它只为订单处理而存在。如果删除该实体类，将不再需要 `OrderAdministrator` 或 `OrderRegistrar`，因为它们只有在存在 `Order` 时才有用。因此，实体类 `Order` 应该与这两个控制类包含在同一个包中。

③ 如果两个对象进行大量的消息交互，或者以其他复杂的方式互相通信，那么这两个对象就在功能上相关。

④ 如果某个边界类的功能是显示一个特定的实体类，那么它就在功能上与该实体类相关。

⑤ 如果两个类与同一个参与者进行交互，或受到对同一个参与者更改的影响，那么这两个类就在功能上相关。

⑥ 如果两个类之间存在某些关系（关联关系、聚集关系等），那么这两个类就在功能上相关。

⑦ 一个类与创建其实例的类在功能上相关。

反过来，两个类之间的相关性越弱，意味着它们之间的耦合越松散，往往不宜放在同一个包中。以下两条标准可确定何时不应将两个类放在同一个包中：其一是与不同参与者相关的两个类不应放在同一个包中，其二是可选类和必选类不应放在同一个包中。

2. 描述包之间的依赖关系

包依赖关系显示了对某些类的修改如何波及系统的其他部分。如图 7.24 所示，包 A 依赖于包 B。从包 A 指向包 B 的虚线箭头表示在包 A 中至少有一个类使用了至少一个来自包 B 的类。这就意味着对包 B 中某些类的改动会影响到包 A 中的类。另一方面，没有从包 B 指向包 A 的依赖关系表示对包 A 中的类进行修改不会对包 B 中的类产生影响。

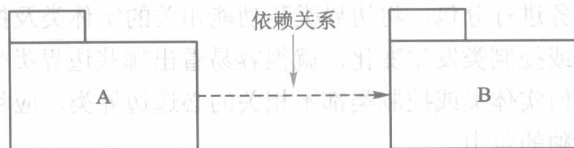


图 7.24 包的依赖关系

通常在一个高层次的视图上显示包依赖关系，它省略了单个的类，专注于所有的包和它们之间的依赖关系。这种图能够使开发人员在高层次的视图上度量系统的复杂度，并帮助他们对一次系统修改的工作量做出正确的估算。

注意：要杜绝包之间相互依赖对方的现象，否则如果对系统的一部分进行了某些修改，就很难对其他部分所受的影响做出正确的估计，因为在一个包中所做的修改将会波及整个依赖循环。

3. 包之间的耦合关系

高内聚、低耦合原则是软件设计工作努力的大方向。包之间的耦合表现为依赖关系，意味着有机会使用其他包中对象的行为。确定和调整包之间的依赖关系时，可以遵循以下的一般原则：

- ① 消除包之间的互相依赖关系，即避免包 A 依赖包 B 的同时包 B 依赖包 A。
- ② 复用价值较高的包不要依赖复用价值较低的包。在层次架构中，较低层中的包不应依赖于较高层中的包。包应只依赖于同一层及下一层中的包。
- ③ 依赖关系不应跨层。跨越层次的依赖关系只有在迫不得已的情况下才能使用。
- ④ 不要让包直接依赖包含实现子系统接口的系统元素的包，换言之，不要直接依赖这类包中的设计元素，而是依赖相应的子系统接口。

4. 分包示例

[例 7.5] 为“选课系统”进行分包设计。

- ① 在应用子系统层中建立 Registration 包，将与选课相关的类放在一起。
- ② 在业务专用层中建立 University Artifacts 包，将关系紧密的实体类放在一起。

[解] ① 建立 Registration 包。按照上述的分包原则，可以将下列与选课相关的类（如 MainStudentForm、RegisterForCoursesForm、RegistrationController、MainRegistrarForm、CloseRegistrationForm、CloseRegistrationController 等）一起打包，并用一张类图描述包内部的关系，如图 7.25 所示。

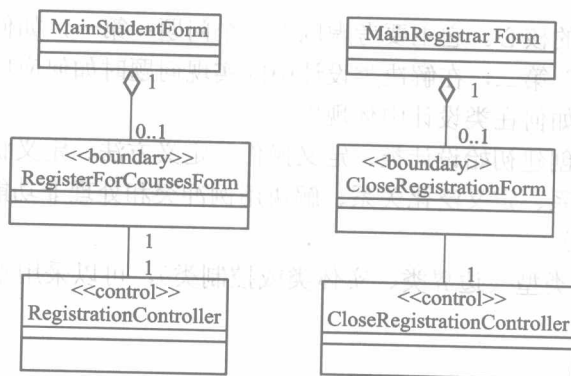


图 7.25 Registration 包内元素的类图

② 建立 University Artifacts 包，将关系紧密的实体类放在一起，同时用类图描述它们之间的关系，如图 7.26 所示。

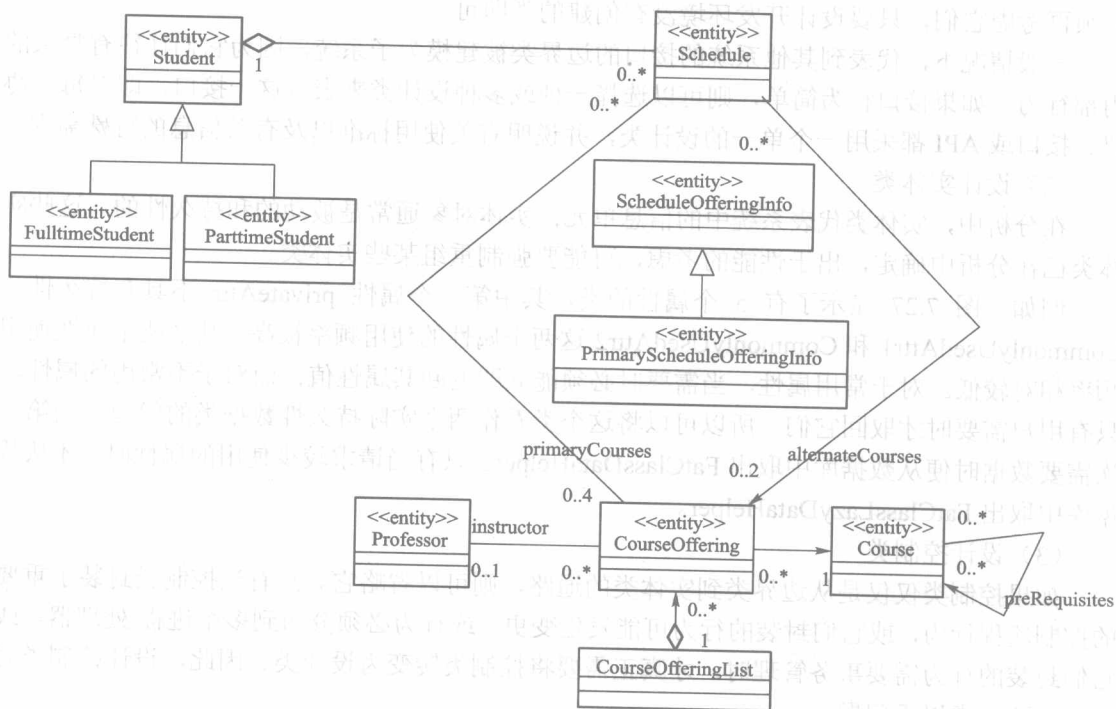


图 7.26 University Artifacts 包内元素的类图

7.4.3 类/对象设计

子系统、包以及协作关系等设计元素，仅仅说明了类的组合方式或协同操作方式。类/

对象设计才是设计工作的核心。它主要考虑以下3个问题：第一，如何实现分析模型中的边界类、实体类和控制类？第二，在解决类设计中的实现问题时如何应用设计模式？第三，系统架构中的全局性决定如何在类设计中体现？

类设计的步骤包括创建初始设计类、定义操作、定义方法、定义状态、定义属性、定义依赖关系、定义关联关系、定义泛化关系、解决用例冲突和处理非功能性需求等。

1. 创建初始设计类

根据不同的分析类类型（边界类、实体类或控制类），可以采用不同的策略来创建初始设计类。

(1) 设计边界类

在分析阶段，已经确定了边界类。在设计阶段需要完成它们的设计，根据实现平台和技术，需要创建额外的类来支持实际的 GUI 和外部的系统交互。

用户界面边界类的设计需要遵循系统架构中关于人机交互界面的决定，还要依赖项目可用的用户界面开发工具。如果 GUI 开发环境自动创建了实现用户界面必需的支持类，那么就无须再考虑它们，只要设计开发环境没有创建的类即可。

一般情况下，代表到其他系统的接口的边界类被建模为子系统，因为它们往往有复杂的内部行为。如果接口行为简单，则可以选择一种或多种设计类来表示这一接口，即对每一协议、接口或 API 都采用一个单一的设计类，并说明有关使用标准以及有关信息的特殊需求。

(2) 设计实体类

在分析中，实体类代表系统中的信息单元；实体对象通常是被动的和持久性的。这些实体类已在分析中确定，出于性能的考虑，可能要强制重组某些实体类。

例如，图 7.27 显示了有 5 个属性的类，其中第一个属性 `privateAttr` 不具有持久性，`CommonlyUsedAttr1` 和 `CommonlyUsedAttr2` 这两个属性的使用频率很高，其余两个属性使用频率相对较低。对于常用属性，当需要时必须能立即返回其属性值；而对于不常用的属性，只有用户需要时才取回它们。所以可以将这个类看作两个实际持久性数据类的代理。当第一次需要数据时便从数据库中取出 `FatClassDataHelper`。只有当请求较少使用的属性时，才从数据库中取出 `FatClassLazyDataHelper`。

(3) 设计控制类

如果控制类仅仅是从边界类到实体类的通路，则可以省略它。只有当控制类封装了重要的控制流程行为，或它们封装的行为可能发生变更，或行为必须分布到多个进程/处理器，或它们封装的行为需要事务管理时，才真正需要将控制类转变为设计类。因此，设计控制类时至少需要考虑以下问题：

① 复杂性。边界类和/或实体类可以处理简单的控制或协调行为。然而，随着应用程序的复杂程度不断增加，也将暴露这种方法的明显缺点，可采用控制类来提供与协调事件流有关的行为。

② 变更的可能性。如果事件流更改的可能性较小，那么增加专门的控制类就没有必要

了，反之则需要专门的控制类。

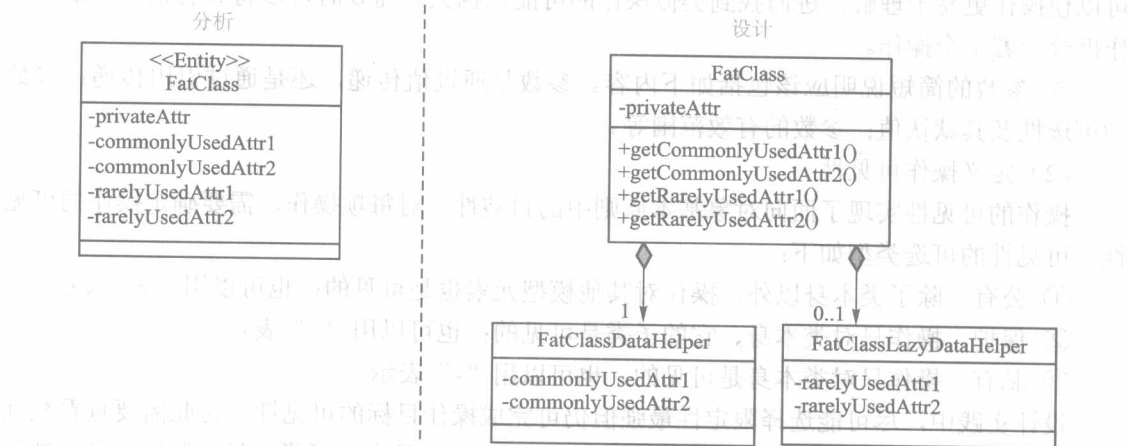


图 7.27 类的重新设计

③ 分布和性能。应用程序需要在不同结点或不同处理器上运行的需求会要求设计模型元素专用性，这种专用性一般是通过添加控制对象，并将边界类行为与实体类行为分布到控制类上来实现的。这样，边界类就提供纯粹的 UI 服务，实体类提供纯粹的数据服务，而控制类提供其余服务。

④ 事务管理。管理事务是典型的协调活动，如果没有处理事务管理的框架，则需要一个或多个事务管理器类用于交互作用，以确保事务的完整性。

2. 定义操作

对于每个确定的设计类，要确定它的所有操作。具体方法如下：

首先，研究每个分析类的职责，为每项职责创建一个操作，将职责的说明作为操作的初始说明；研究类所参与的用例的行为，查看用例中如何使用该类的操作，在此基础上改进操作及其说明、返回类型和参数；研究用例的特殊需求，确保类的操作覆盖用例的行为，没有遗漏隐含需求。

其次，用例中通常无法提供足以确定类的所有操作的有关信息，因此还要增加一些典型的操作，例如，初始化类的实例，验证类的两个实例是否等同，创建类的实例的副本等。

(1) 命名和说明操作

对于每项操作，应该定义下列内容：

① 操作名。操作名应该简短，并隐含进行操作所得到的结果。例如用 `getBalance()` 和 `calculateBalance()` 来命名获取和计算余额的操作。

② 返回类型。返回类型是操作返回对象的类。

③ 简短说明。可以为操作提供简单的说明，一般从操作用户的角度来编写操作说明。

④ 参数。使用简明、易懂的参数名称，确定参数的类，为参数提供简介。参数个数少可以使操作更易于理解，进而找到类似操作的可能性也大。必要时可以将具有很多参数的操作拆分为若干个操作。

⑤ 参数的简短说明应该包括如下内容。参数是通过值传递，还是通过引用传递；参数的可选性及其默认值；参数的有效范围等。

(2) 定义操作可见性

操作的可见性实现了面向对象基本原则中的封装性。对每项操作，需要确定操作的可见性。可见性的可选类型如下：

① 公有。除了类本身以外，操作对其他模型元素也是可见的；也可以用“+”表示。

② 保护。操作只对类本身、它的子类是可见的；也可以用“#”表示。

③ 私有。操作只对类本身是可见的，也可以用“-”表示。

设计实践中，尽可能选择限定性最强但仍可完成操作目标的可见性。为此需要查看交互图，确定每条消息是来自对象外（要求公有的可见性），还是来自子类（要求保护的可见性），或者来自类本身（要求私有的可见性）。

(3) 定义操作的作用域

操作有两种作用域：实例作用域和类作用域。实例作用域的操作是对类的实例执行的操作。如果一个操作适合于类的所有实例，它就是类作用域的操作。为了表示类作用域的操作，操作字符串加有下划线。

在“选课系统”中，定义了操作 `getNextAvailID`（获取下一个可用的学号）为类作用域的操作，在其下加有下划线。对于所有学生来说，它的值是唯一的；其余操作都是实例作用域。

图 7.28 显示了选课用例参与类图的一部分，其中为设计类定义了操作。

3. 定义方法

如果将操作的定义比作黑盒，那么方法（method）就是相应的白盒内容。方法是对操作的实现，由具体的程序设计语言完成。方法说明了实现操作的具体方式。其内容通常会涉及操作的参数、类的属性及关系在方法中的使用。如果方法的内容需要采用特定的算法，应给出相应的文字或图示进行说明。

4. 定义状态

对于某些操作，操作行为依赖于接收方对象的状态。状态图是一种有效的工具，它描述对象可以具有的状态和导致对象状态变化的事件。如果一个类有一些重要的动态行为，具有多种状态，就可以创建状态图。要确定一个类是否具有重要的动态行为，可以通过以下两种方法：

① 检查类的属性。考虑一个类的实例在属性值不同时如何表现，因为如果对象的行为表现不同，则其状态也不同。

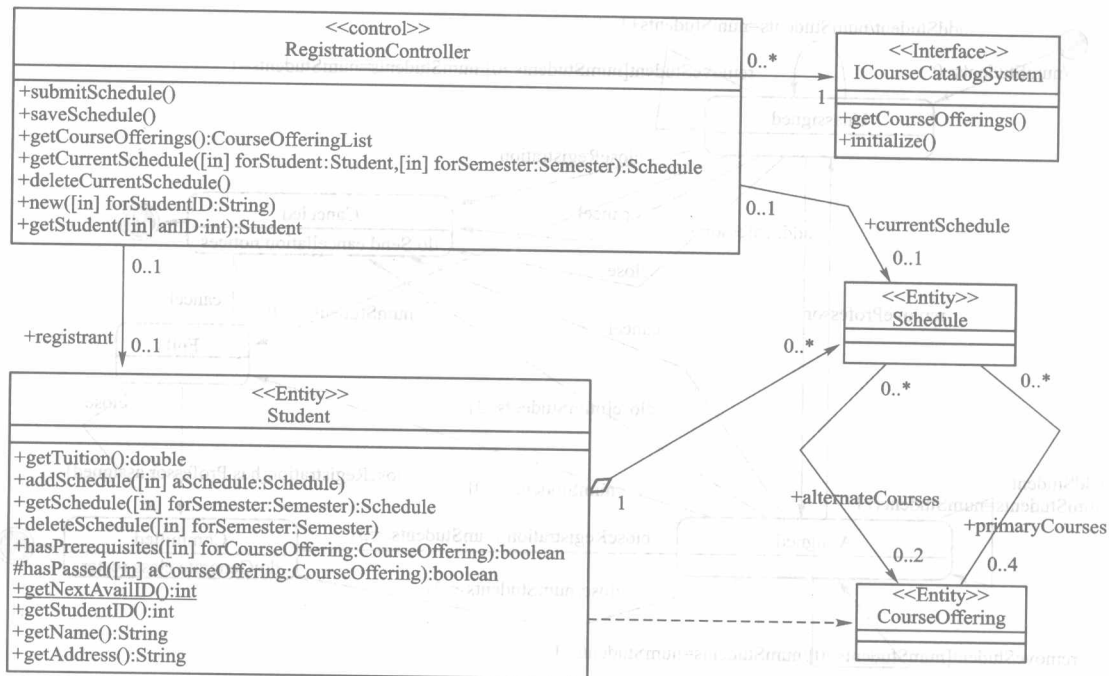


图 7.28 定义操作的类图

② 检查类的关联。看看关联重数中带 0 的关联，0 表示这个关联是可选的。关联存在和不存在时类的实例是否表现相同。如果不同，则可能有多种状态。

【例 7.6】为“选课系统”中有关开设课程的类创建状态图。

【解】图 7.29 是选课系统中与开设课程有关的类的状态图。

每个状态转换事件都与操作关联，根据对象的状态，操作可能有不同的行为，转换事件描述这种情况在何时发生。操作的方法描述应该根据特定状态的信息进行更新，表明操作对不同状态应该做什么。

5. 定义属性

在方法的定义和状态的确定中，确定了类执行其操作所需的属性。属性为类实例提供了信息存储空间，并表示类实例的状态。类本身保留的任何信息都是通过属性实现的。在分析阶段，一般只需指明属性名；在设计阶段，对于每种属性，需要定义以下内容：

- ① 属性名。通常是名词，遵循实现语言和项目的命名约定。
- ② 属性类型。它应该是实现语言支持的基本数据类型。
- ③ 属性默认值或初始值。创建类的实例时，用于初始化新建的对象。
- ④ 属性的可见性。其可用值如下：

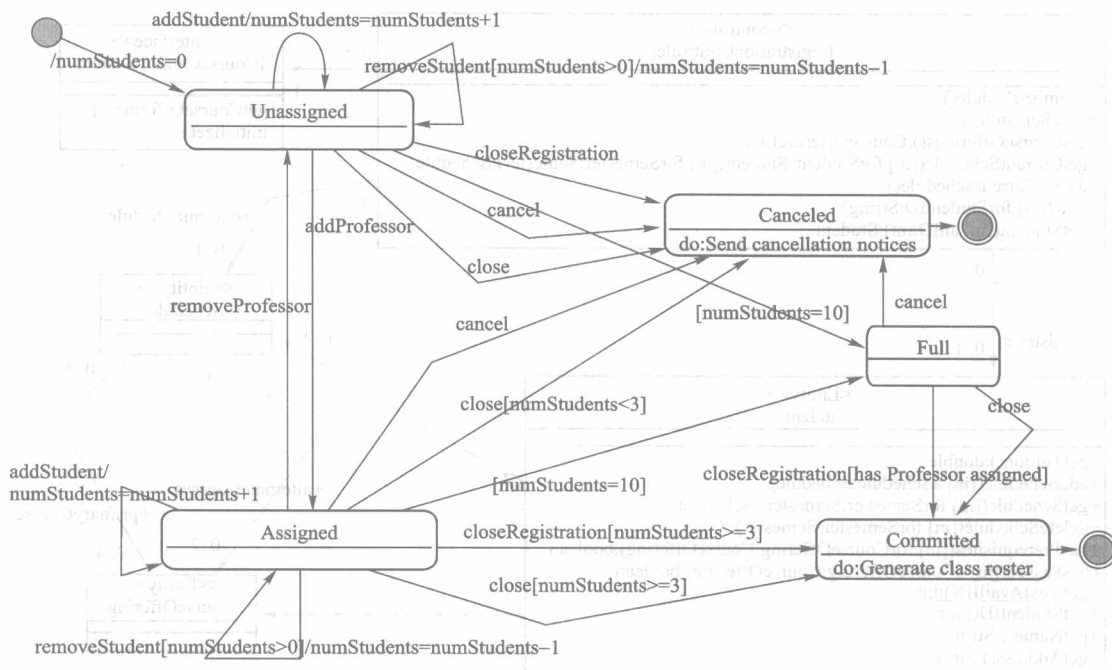


图 7.29 开设课程类的状态图

- 公有：属性对于包含类的包内和包外都是可见的。
- 保护：属性只对类本身、子类是可见的。
- 私有：属性仅对类本身可见。

检查以确保所有属性都是必需的，过多属性将会严重影响系统性能和存储需求。图 7.30 是选课用例实现的参与类图的一部分，在此标明了类的属性。当前登记开设课程的学生数目用派生属性表示，初始值为 0。

6. 定义依赖关系

在软件分析时，假定所有关系都是结构化关系，即用关联关系或聚集关系或组合关系来表述两个类的对象之间存在连接。在设计阶段，随着类的设计内容逐步明朗，有条件进一步确认对象间究竟需要何种类型的连接，从而明确类之间的关系。

类 A 的对象 a 与类 B 的对象 b 通信，a 能够向 b 发送消息的必要条件是 a 能够引用 b，其概念在协作图中表现为 a 到 b 的连接。在面向对象软件系统中，a 可以通过 4 种方式引用 b，对应于从 a 到 b 的 4 种类型的连接可见度。

- ① 全局 (global)。b 是可以在全局范围内直接引用的对象。
- ② 参数 (parameter)。b 作为 a 的某一项操作的参数或返回值。
- ③ 局部 (local)。b 在 a 的某一操作中充当临时变量。

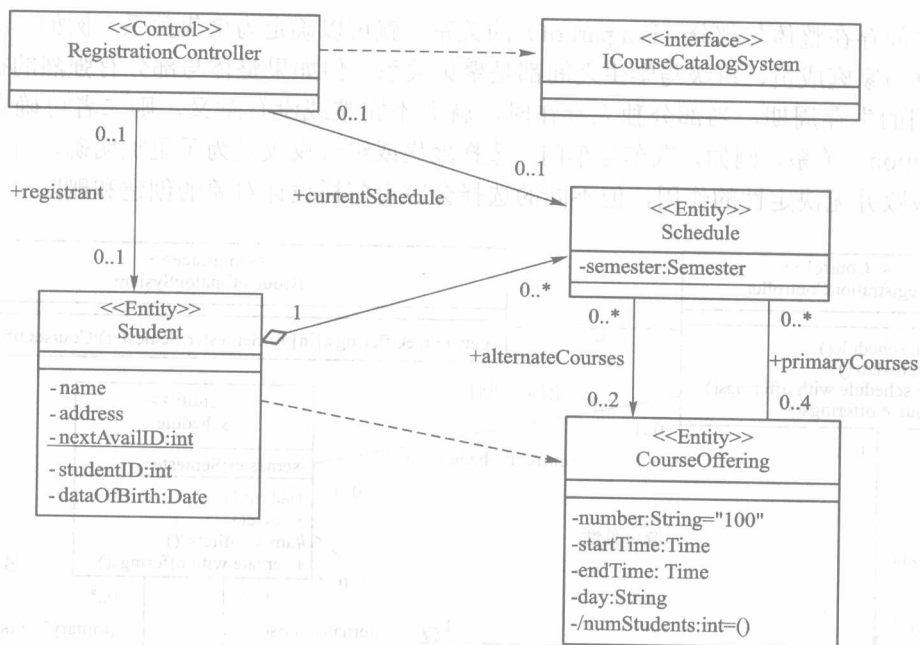


图 7.30 标明属性的类图

④ 域 (field)。b 作为 a 的数据成员。

前 3 种类型连接可见度具有暂时性，即 b 和 a 之间的连接仅在执行某个操作的过程中被建立，执行完后连接关系解除。在静态结构中，这 3 种类型的连接被建模为类 A 对类 B 的依赖关系。最后一种类型的连接可见度具有稳定性和永久性，在静态结构中这种连接被建模为类 A 到类 B 的关联关系及其强化形式。

【例 7.7】 为选课用例的参与类图定义依赖关系。

【解】 图 7.31 显示了加入依赖关系后的选课用例的参与类图。其中，子系统接口是全局范围内可利用的资源，需要同时被多个客户访问，所以对它的关系由原先的关联变为依赖；在一次选课会话中，注册控制器处理一个学生和一张课表，这些实例需要被多个注册控制器的操作访问，所以它们之间具有域可见性；一个学生管理自己的课表，所以学生与课表间具有域可见性；开设课程是定义课程表语义的一部分，因此课程表到开设课程具有域可见性；开设课程出现在学生类的操作的参数列表中，因此它们之间具有参数可见性。

7. 定义关联关系

关联关系是一种结构化的关系，在后续的实施活动中，其内容将作为类定义的组成部分。确认类之间存在关联关系之后，有条件进一步明确或改进其细节内容。

(1) 聚集还是组合

关联又可以区分为聚集或组合两种类型。聚集 (aggregation) 是一种特殊的关联，如果

两个类之间存在整体与部分 (is a part of) 的关系, 就可以确定为聚集关系。例如, 部门和雇员、家庭与家庭成员、班级与学生之间都是聚集关系。但如果整体与部分有强烈的拥有关系且有相同的生存周期, 当部分独立存在时, 就会不完整或没有意义, 则二者可确定为组合 (composition) 关系, 例如, 汽车与车门。选择聚集或组合仅仅是为了更加明确, 对分析建模工作的成败并无决定性的作用, 但不同的选择会决定怎样设计对象的创建和删除。

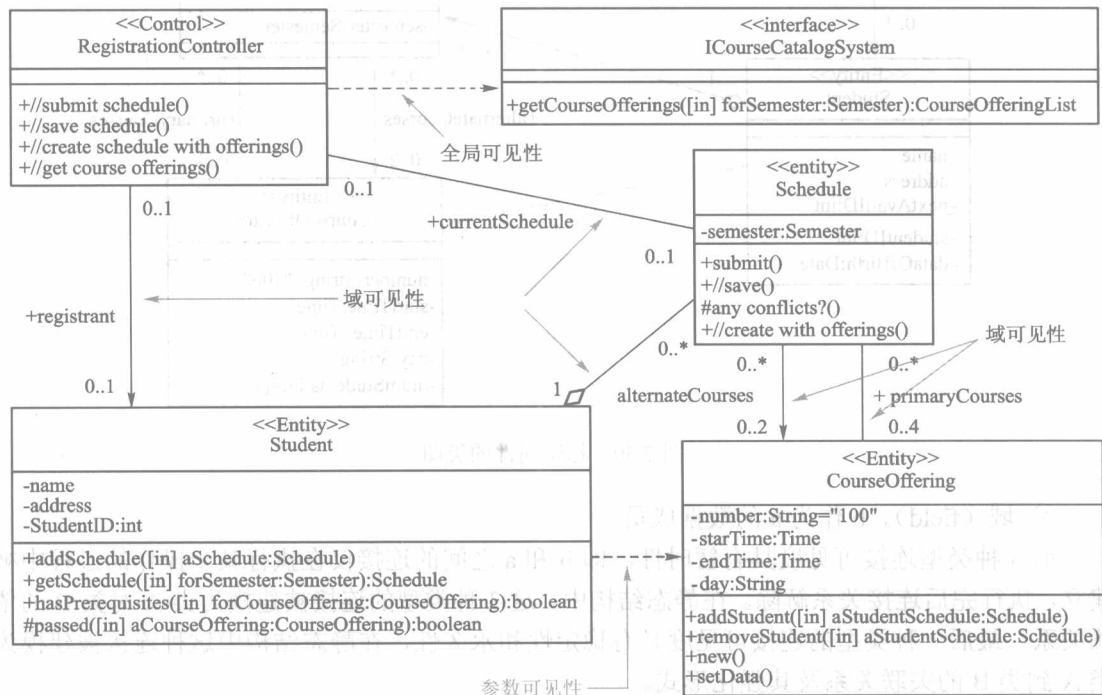


图 7.31 定义依赖关系的类图

图 7.32 显示了选课系统中组合关系的两个例子。其中学生 `Student` 和课程表 `Schedule` 间需要组合关系是因为如果删除了学生, 也就删除了那个学生所有的课表; 注册控制器的实例在课程注册会话的范围外不会存在, 既然注册课程的表单代表某个会话, 那么注册控制器也不能在某个注册表单的范围外存在, 课程注册控制器的实例随着表单的创建而创建, 随着表单的销毁而销毁。由于它们有相同的生存周期, 所以存在组合关系。

(2) 属性还是组合

类的特征就是类所了解的东西, 既可以将类的特征建模成一个类 (利用组合关系), 也可以将其建模成该类的属性集。要决定对单个类使用属性还是组合关系, 应以所表示概念之间的耦合度为依据, 如果正在建模的概念之间联系很紧密, 就应使用属性, 如果概念易于独立进行变更, 则应使用组合关系。此外还可以从以下 3 个方面考察特征来决定是否使用类和

组合关系。第一，特征是否需要独立的身份，以便被大量的对象所引用？第二，是否有大量的类需要有相同的特征？第三，特征是否具有复杂的结构和它们自己的特征、行为或关系？

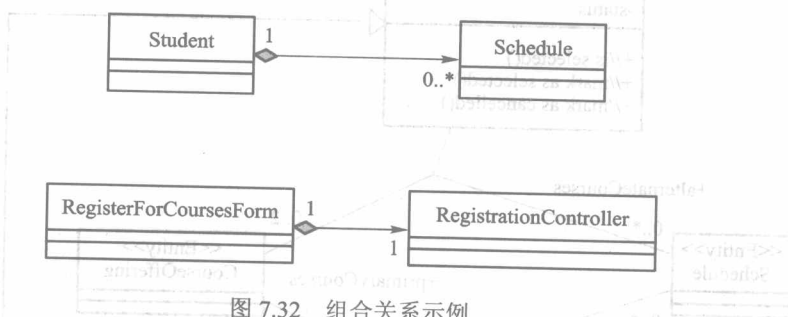


图 7.32 组合关系示例

(3) 关联的方向

关联关系是有访问方向的，需要考察双方向关联关系中某一方向存在的必要性，因为两个类之间单方向的关联关系比双方向的关系弱。

(4) 关联类

复杂的关联可以用类来表示，这就是关联类。第 4 章已经介绍过，关联类是一种关联关系，它具有类的特征（例如属性、操作和关联关系）。它用一条从关联关系连接到类符号的虚线表示，其中类符号包含此关联关系的属性、操作和关联关系，这些属性、操作和关联关系适用于原始关联关系本身。关联关系中的每个链接都有指定的特征，关联类最常见的用途是协调多对多关系。

【例 7.8】 为“选课系统”中的关联类定义关联关系。

【解】 图 7.33 显示了选课系统中关联类的设计。

(5) 确定重数

对于重数大于 1 的情形，通常需要进一步设计包容器类，比较常见的诸如集合、链表、堆栈和队列等。如果重数的最小取值为 0，意味着与关联关系相应的链接有不存在的可能，通常需要建立判断链接是否存在的操作。

8. 定义泛化关系

与依赖或关联关系不同，类之间的泛化关系并不能通过渐进的分析和设计活动而得出。设计模型的内容增多之后，属于同一概念范畴的类（子类）之间会存在相似的行为与结构。为了提高设计内容的复用能力并降低维护的难度，可以将它们的共同部分抽取出来并定义成新的类（父类），在子类和父类之间定义泛化关系。很多时候，已有的设计类可以直接作为父类，用于简化相关子类的定义。

泛化关系和聚集关系经常被混淆，其中泛化代表了“is a”或“kind-of”的关系；而聚集代表了“part-of”的关系。图 7.34 中，带滚动条的窗口是窗口的一种，所以是泛化关系；滚动条是窗口的一部分，所以是聚集关系。

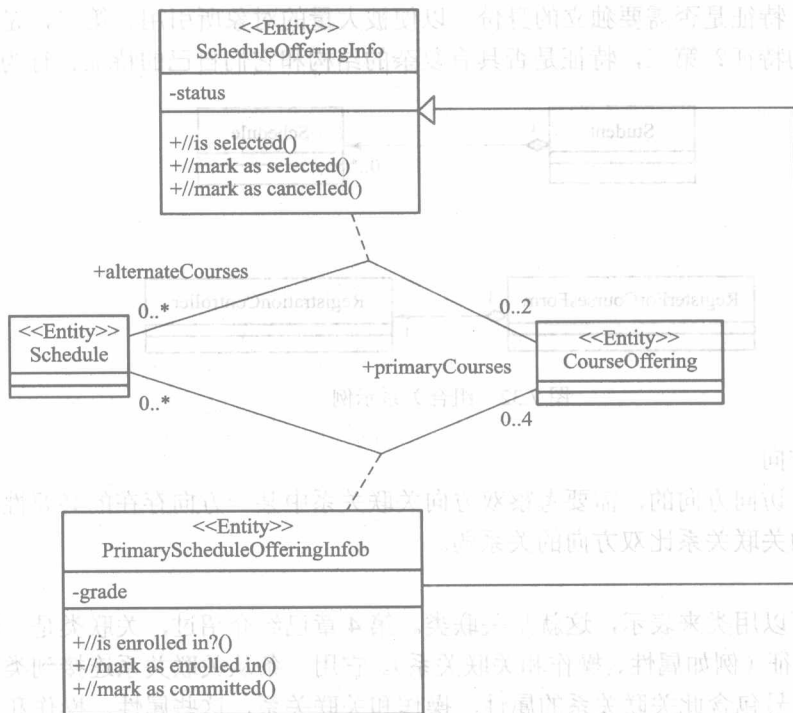


图 7.33 关联类

9. 处理非功能性需求

设计类应该加以改进，以便处理公共的非功能性需求，设计类时，需要考虑的通用设计指南和规则如下：

- 如何利用现有产品和构件？
- 如何适应编程语言？
- 如何分布对象？
- 如何达到可以接受的性能？
- 如何达到特定的安全性级别？
- 如何处理错误？



图 7.34 泛化关系与聚集关系示例

7.5 面向对象设计示例

作为示例，本节首先用 7.5.1 小节说明“网上购物系统”中聚集和组合关系，7.5.2 小节介绍系统架构设计方法，然后在 7.5.3 节，针对系统的注册、维护个人信息、维护购物车、生

成订单、管理订单等 5 个用例，展示网上购物系统的类/对象设计。

7.5.1 关联关系的具体化

面向对象分析模型中的关联关系，有时在设计阶段需要具体化为聚集、组合或依赖关系。在展开系统架构设计和系统元素设计之前，首先介绍网上购物系统中的聚集、组合和依赖关系。

1. 聚集关系

部分对象可以是任意整体对象的一部分。在“网上购物系统”中，可以找出如下的共享聚集关系，如图 7.35 所示。

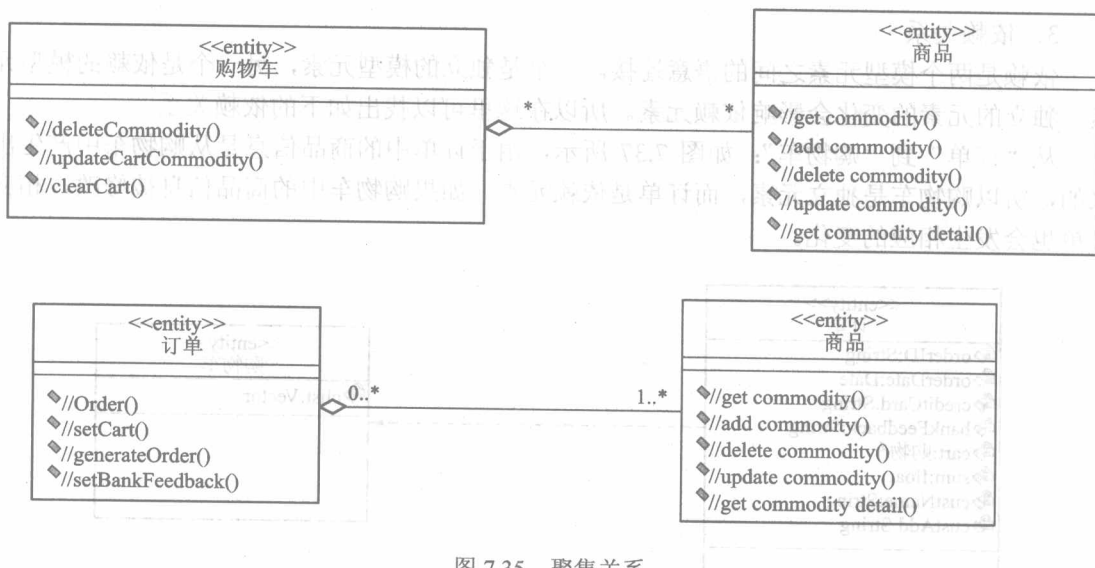


图 7.35 聚集关系

从“购物车”到“商品”：每个购物车中可以包含 0 或多种不同的商品，同一商品可以出现在多个购物车中。

从“订单”到“商品”：每个订单中可以包含 1 或多种不同的商品，同一商品可以出现在多个订单中。

2. 组合关系

在组合关系中，整体和部分有相同的生存周期，若整体不在了，部分也会随之消失。所以，可以找出如图 7.36 所示的组合关系。

从“顾客”到“购物车”：在顾客登录进入系统进行购物时，系统就为他创建了一个购物车。一个购物车只能唯一属于一位顾客，不存在没有与顾客联系的购物车。说顾客和购物车具有组合关系，是因为一旦顾客结束购物并退出系统，则属于他的购物车也就随之不存

在了。

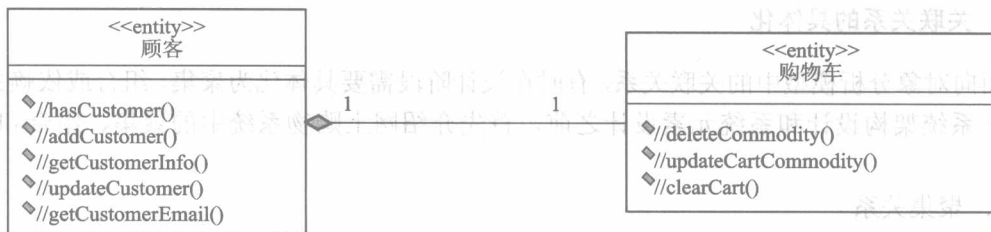


图 7.36 组合关系

3. 依赖关系

依赖是两个模型元素之间的语意连接。一个是独立的模型元素，另一个是依赖的模型元素。独立的元素的变化会影响依赖元素。所以在这里可以找出如下的依赖关系。

从“订单”到“购物车”：如图 7.37 所示，由于订单中的商品信息是从购物车中产生出来的，所以购物车是独立元素，而订单是依赖元素。如果购物车中的商品信息被修改，相应订单也会发生相应的变化。



图 7.37 依赖关系

7.5.2 网上购物系统的架构设计

考虑到该系统是一个基于 B/S 的分布式网络应用系统，所以总体上采用分层结构，使整个系统的逻辑实现更加缜密。具体地说，可采用 3 层架构来实现该系统，即表现层、业务逻辑层和数据访问层。

1. 各层分析

① 数据访问层。该层是整个系统中用于对数据库进行操作的层次，它为上层的业务逻辑层提供服务。在这一层中，对应于数据库中的每一张表，都有相对应的一个数据访问类。

② 业务逻辑层。该层主要实现本系统所需要的业务逻辑。通过数据访问层，将所需要

的数据库信息读取到数据库所对应的实体类中，并完成系统所要求实现的产品浏览、购物车和订单生成等功能。这一层的各个类就是通过数据访问层的各个方法取得数据，然后对数据进行处理，对表现层提供必要的信息。

③ 表现层。这一层就是用户所能够看到并对系统进行操作的 Web 页面，是用户与系统之间的接口，用于接受用户请求和返回系统处理结果。

2. 架构模式

本系统在表示层主要用到的设计模式是 MVC，M 为模型（model），主要实现对业务逻辑的处理。V 代表视图（view），采用 JSP 实现，主要用于显示数据和将用户请求和输入传给控制器；C 是控制器（controller），采用 Servlet 组件，用于连接 V 和 M，用于调用不同的模型处理用户请求，选择不同的视图显示系统信息。

7.5.3 网上购物系统的类/对象设计

1. 注册

[例 7.9] 为注册用例进行类/对象设计。

[解] 图 7.38 显示了注册用例的类/对象，以及各个类之间的关系。其中各个类增加了属性以及方法。

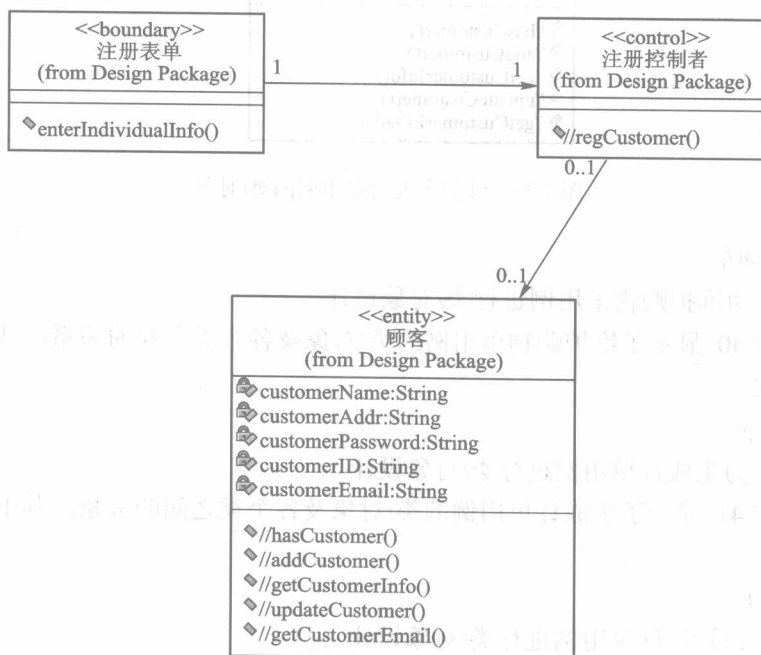


图 7.38 注册用例的类/对象

2. 维护个人信息

[例 7.10] 为维护个人信息用例进行类/对象设计。

[解] 图 7.39 显示了维护个人信息用例的类/对象，以及各个类之间的关系。其中各个类增加了属性以及方法。

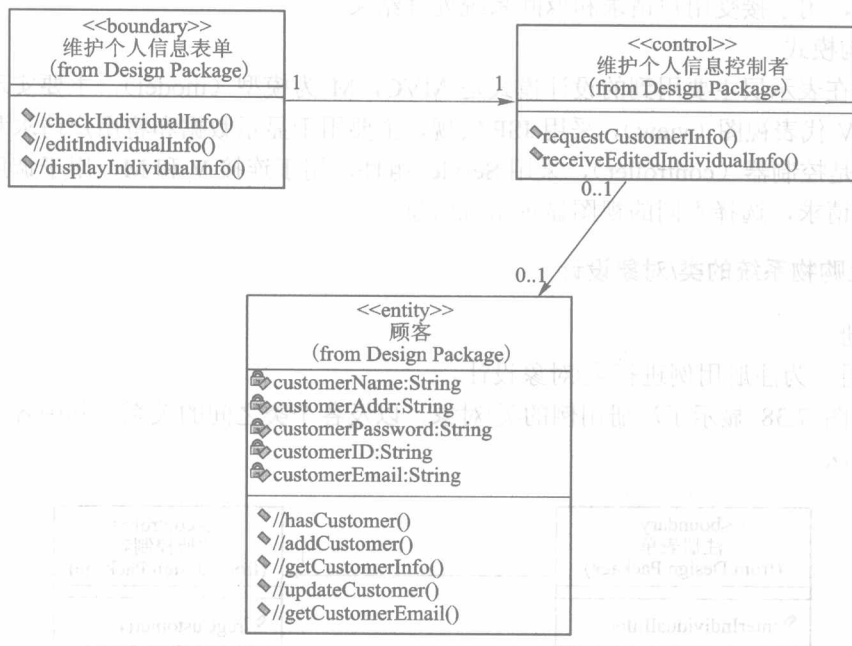


图 7.39 维护个人信息用例的类/对象

3. 维护购物车

[例 7.11] 为维护购物车用例进行类/对象设计。

[解] 图 7.40 显示了维护购物车用例的类/对象及各个类之间的关系。其中各个类增加了属性以及方法。

4. 生成订单

[例 7.12] 为生成订单用例进行类/对象设计。

[解] 图 7.41 显示了生成订单用例的类/对象及各个类之间的关系。其中各个类增加了属性以及方法。

5. 管理订单

[例 7.13] 为管理订单用例进行类/对象设计。

[解] 图 7.42 显示了管理订单用例的类/对象及各个类之间的关系。其中各个类增加了属性以及方法。

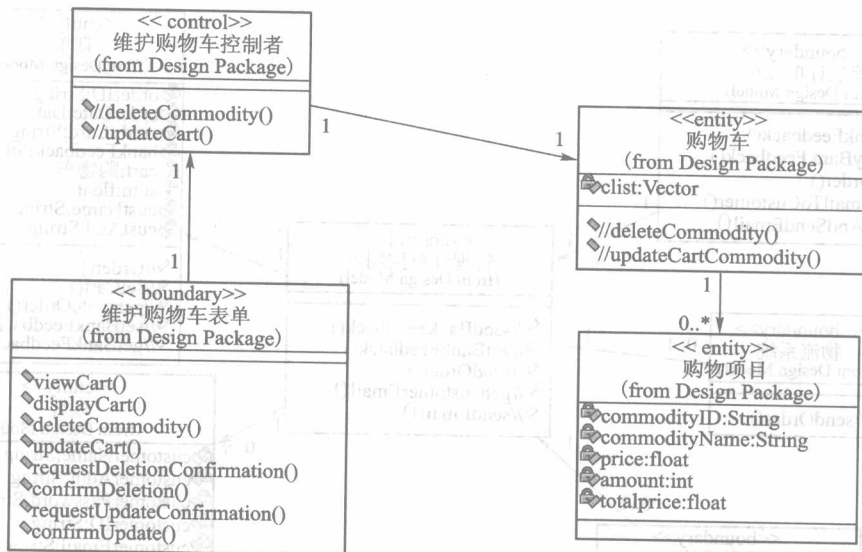


图 7.40 维护购物车用例的类/对象

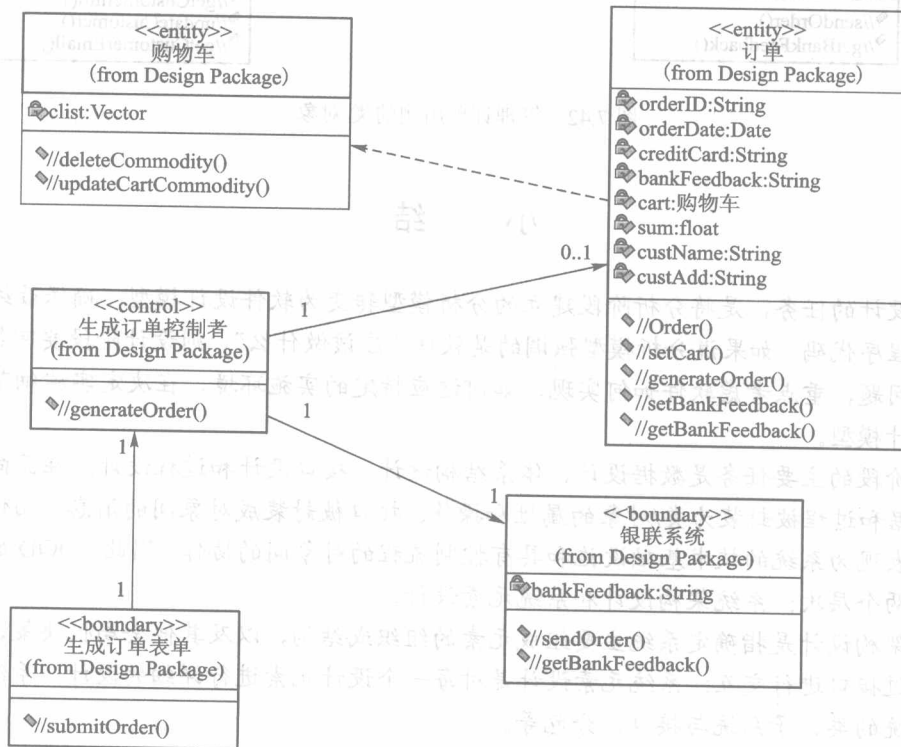


图 7.41 生成订单用例的类/对象

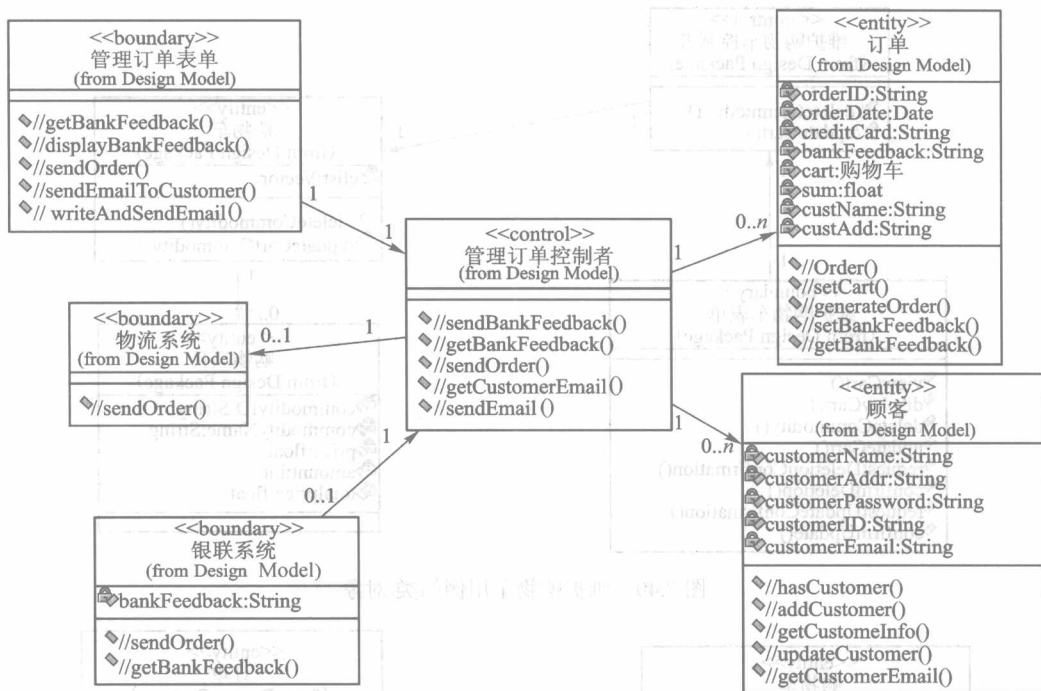


图 7.42 管理订单用例的类/对象

小 结

软件设计的任务，是将分析阶段建立的分析模型转变为软件设计模型，确保最终能平滑地过渡到程序代码。如果说分析模型强调的是软件“应该做什么”，则设计阶段要回答“该怎么做”的问题，重点考虑软件如何实现，如何适应特定的实施环境，在决定实施细节的基础上创建设计模型。

设计阶段的主要任务是数据设计、体系结构设计、接口设计和过程设计。在面向对象设计中，数据和过程被封装为类/对象的属性和操作；接口被封装成对象间的消息；而体系结构的设计则表现为系统的技术基础设施和具有控制流程的对象间的协作。因此，OOD 的软件设计也分为两个层次：系统架构设计和系统元素设计。

系统架构设计是指确定系统主要组成元素的组织或结构，以及其他全局性决策，组成元素之间通过接口进行交互；系统元素设计是对每一个设计元素进行详细的设计，系统元素包括组成系统的类、子系统与接口、分包等。

第 7.5 节给出了相应的示例。

习 题

1. 面向对象设计的任务是什么？
2. 解释下列名词：模块；模块化；模块化设计。
3. 什么是模块独立性？可用什么来度量？
4. 系统架构设计包括哪些内容？
5. 阅读相关文献，简述 Coad 和 Yourdon 的 OOD 方法和 Rumbaugh 的 OMT 技术的内容和特点。
6. 利用 JDBC，如何实现数据的持久性存储？

7. 饮料自动售货机用于顾客自助购买饮料。不同的饮料有不同的价格，顾客自助购买饮料的过程是这样的：饮料自动售货机接受投入的硬币并判断硬币是否有效及面值，金额计算器负责累计金额，待金额计算器累加到饮料售价时该饮料的选择键就会亮起来，顾客可以按选择键购买该饮料；顾客在按选择键前，若按饮料自动售货机上的退币键，则将所有金额退还顾客。试用面向对象的方法设计该系统。

设计题 1.8

图 1.8.1 饮料自动售货机系统类图

设计题 1.8

图 1.8.2 饮料自动售货机系统类图

图 1.8.3 饮料自动售货机系统类图

第 8 章 编码与测试

在软件开发中，通常把编码和单元测试安排在同一阶段完成，这无疑是合理的。但是从工作性质说，单元测试又是整个测试的一部分。为此，本书将编码和测试并入一章讨论，以便说明两者的关系。

8.1 编码概述

编码（coding）俗称编程序。软件开发的最终目标，是产生能在计算机上执行的程序。分析阶段和设计阶段产生的模型和文档，都不能在计算机上执行。只有到了编码阶段，才产生可执行的代码（executable codes），把软件的需求真正付诸实现，所以编码阶段也称为实现（implementation）阶段。

8.1.1 编码的目的

编码的目的，是使用选定的程序设计语言，把设计模型翻译为用该语言书写的源程序（或源代码）。源程序经过编译等环节，再转换成可执行代码：



为此，程序员除应熟悉所选程序设计语言的功能和程序开发环境外，尤其要仔细阅读设计文档，彻底理解设计模型，弄清要编码的模块的外部接口与内部过程。

编码产生的源程序，应该正确可靠、简明清晰，而且具有较高的效率。前两点要求是一致的，因为源代码越是清楚和简明，就越便于验证源代码和模块规格说明的一致性，越容易对它进行测试和维护。但是，清晰和效率却常有矛盾。Weinberg 曾作过一次试验，让 5 个程序员各自编写同一个程序，分别对他们提出了 5 种不同的编码要求。结果如表 8.1 所示，要求清晰性好的程序一般效率较低，而要求效率高的程序其清晰性又不好。对于大多数模块，编码时应该把简明、清晰放在第一位，如果个别模块要求特别高的效率，就应把具体要求告诉程序员，以便作特殊的处理。

设计是编码的前导。实践表明，编码中出现的问题，许多是由设计的缺陷引起的。可见，程序的质量首先取决于设计的质量。但这并不是说，编码阶段就不能有所作为。恰恰相反，

程序员应该像优秀的译员一样，在编码“翻译”中坚持简明清晰、高质量的原则，竭力避免繁杂、晦涩。为此，程序员不仅要养成良好的编码风格，而且要十分熟悉所使用的语言，以便得心应手、恰到好处地运用语言的特点，为提高程序的清晰性和效率服务。

表 8.1 Weinberg 的程序设计试验

结果 名次	评判项目	清晰性		效率		开发 时间
		程序	输出	内存数	语句数	
编码要求						
程序可读性最佳		1/2	2	3	3	4
输出可读性最佳		1/2	1	5	5	2/3
占内存最小		4	4	1	2	5
语句数最少		5	3	2	1	2/3
开发时间最短		3	5	4	4	1

顺便指出，除了编码阶段要产生源程序外，在测试阶段也需要编写一些测试程序，用于对软件进行测试。但这部分代码用过就可以废弃，不需要在程序质量上多费工夫。在快速原型化开发中产生的原型代码，也有一部分或全部都是用过就废弃的代码。

8.1.2 编码的风格

传统程序设计十分强调编码风格（coding style，又称程序设计风格）。与作家或画家相似，所谓风格，其实就是创作人员在创作中喜欢和习惯使用的表达自己作品题材的方式。

从 20 世纪 70 年代以来，程序的目标从强调效率转变到强调清晰。与此相应，编码风格也从追求聪明和技巧，变为提倡简明和直接。人们逐渐认识到，良好的编码风格能在一定程度上弥补语言存在的缺点，反之，不注意风格，即使使用了现代程序设计语言，也不一定能写出高质量的程序。当多个程序员合作编写一个大的程序时，尤其需要强调良好的和一致的风格，以利于相互通信，减少因不协调而引起的问题。

这里还要重申，清晰第一并非不要效率，而是在清晰的前提下追求效率。下面，我们借用 Kernighan 等在《程序设计风格要素》一书中的几条指导原则，来说明程序正确性、清晰度和效率之间的关系。

- Make it right before you make it faster.（先求正确后求快）
- Make it clear before you make it faster.（先求清楚后求快）
- Keep it right when you make it faster.（求快不忘保持程序正确）
- Keep it simple to make it faster.（保持程序简单以求快）
- Write clearly—don't sacrifice clarity for "efficiency".（书写清楚，不要为“效率”牺牲清楚）

1974年，Kernighan与Plauger在合作的名著《The Elements of programming Style》一书中，把编码风格归结为7个问题，共70余条指导原则。下面从控制结构、代码文档化和输入输出等3个方面，简述编码风格的要求。

1. 使用标准的控制结构

和设计阶段一样，编码阶段要继续遵循模块逻辑中采用单入口、单出口标准结构这一主要原则，确保“翻译”出来的源程序清晰可读。

大多数现代语言（如Java、C/C++、Pascal等）提供了比基本结构更多的控制结构。以Pascal为例，它的3种选择结构和3种循环结构，都是单入口、单出口的，如图8.1所示。

① IF C THEN S	④ WHILE C DO S
② IF C THEN S1 ELSE S2	⑤ REPEAT
③ CASE I OF	S
a: S1;	UNTIL C
b: S2;	⑥ FOR I := M TO N DO
...	S;
n: Sn	
ENDCASE	

(a) 3种选择结构

(b) 3种循环结构

图8.1 Pascal提供的控制结构

其实，图8.1中①、③、⑤、⑥等结构都可用顺序、IF-THEN-ELSE和WHILE-DO等3种基本结构来描述。例如，IF C THEN S ELSE null（null表示空语句）、CASE可用一系列嵌套的IF-THEN-ELSE结构代替；REPEAT和FOR两种循环结构均可用WHILE-DO来表示等。提供这些附加的控制结构，是为了增加表达上的便利或改进程序的可读性，有时还能够提高执行的效率，例如CASE结构的执行速度就可能比嵌套的IF-THEN-ELSE结构快。因此在使用这类语言编码时，允许使用3种基本结构以外的附加控制结构。

2. 实现源程序的文档化

软件=程序+文档。在传统程序设计中，编码的目的是产生程序，其余阶段则主要产生文档。但是，为了提高程序的可维护性，源代码也需要实现文档化（code documentation）。有些文献把这些称为内部文档编制（internal documentation）。

源代码的文档化主要包括以下3个方面的内容：

- ① 有意义的变量名称。在程序中采用有意义的名称，虽然长一点，但意义自明，从而提高了程序的可读性。
- ② 适当的注释。源程序的注释，通常用于每一程序单元开始处、重要的程序段或难懂的程序段之前。

在每个模块或子程序开始处，常有一段类似于序言（prologue）的注释，用于说明模块的名称、用途、变量含义、调用模块名和被什么模块调用以及程序员姓名、编译日期与修改日期等。

嵌在源代码内部的注释，经常应放在重要的程序段或每一控制结构开始的地方，难懂的程序段也应加上注释。

充分的注释对提高程序的可读性有很大的帮助。但决不能理解为多多益善。恰恰相反，通过采用有意义的变量名、结构化的结构等良好的编程风格，应尽可能减少嵌入程序内部的注释，杜绝那种重复本来就清楚的内容的鹦鹉学舌式的注释（parrot's comment）。

修改程序时，注释也应随之修改，以保持注释和代码的一致性。

③ 标准的书写格式。用统一的、标准的格式来书写源程序清单，有助于改进可读性。常用的方法有：用分层缩进的写法显示嵌套结构的层次；在注释段的周围加上边框；在注释段与程序段以及不同程序段之间插入空行；每行只写一条语句；书写表达式时，适当使用空格或圆括号等作分隔符；等等。当多个程序员共同完成一个大程序的编码时，人人都应该遵守约定的书写格式。格式化的工作可以由程序员手工完成，也可以由专用的格式化程序自动完成，后者也是编码阶段很有用的一种工具。

3. 满足用户友好的输入输出风格

绝大多数计算机系统都是人机交互系统。因此程序的运行，要充分考虑人的因素，尽量做到对用户友好（user friendly）。以下是程序设计和编码中在输入输出方面应该遵守的若干指导原则。

(1) 输入方面

- ① 让程序对输入数据进行有效检验，防止对程序的有意和无意的破坏。
- ② 输入格式力求简单、一致，并尽可能采用自由格式输入。
- ③ 使用数据结束或文件结束标志来终止输入，不要让用户自己来计算输入的项数或记录数。
- ④ 向用户显示“请输入”等提示信息，同时说明允许的选择范围和边界值。
- ⑤ 对多个相关输入项的组合进行检查，删除似是而非的输入值。例如，检查代表三角形三条边长的数据项，如果它们不能组成有效的三角形，便拒绝接收。

(2) 输出方面

- ① 标志所有的输出数据，加以必要的说明。
 - ② 使所有报表、报告具有良好的格式。
- 对于具有大量人机交互的系统，尤其要注意良好的输入输出风格。Wasserman 曾著文论述这个问题，并称之为“用户软件工程（user SE）”。他提出的主要观点有：
- ① 当用户使用程序时，能对用户做到“在线”帮助。
 - ② 对可能产生重大后果的请求，给出醒目的提示，待用户再次确认后再执行。
 - ③ 使程序具有“防狼（bulletproof）”功能，不致因用户的偶然错误使程序发生非正常的

中断。

④ 区别对待不同类的用户（例如，专业人员与非专业人员，领导干部与一般工作人员），使程序的输入要求和输出形式适应用户的习惯与水平。

⑤ 发生错误时，能迅速恢复正常。

以上概述了对编码风格的主要要求。这些要求是独立于所使用的语言的，但实现的难易又受到所用语言的影响。本章的其余部分，将进一步讨论程序设计语言对编码的影响。

8.2 编码语言与编码工具

编码需要使用某种程序设计语言，而编码工具则是用于辅助程序员进行编码的软件工具。选择合适的编码语言和编码工具，对软件编码的效率和质量都有重要意义。

8.2.1 编码语言的发展

按照软件工程的观点，语言的发展至今已经历了4代、3个阶段，如图8.2所示。

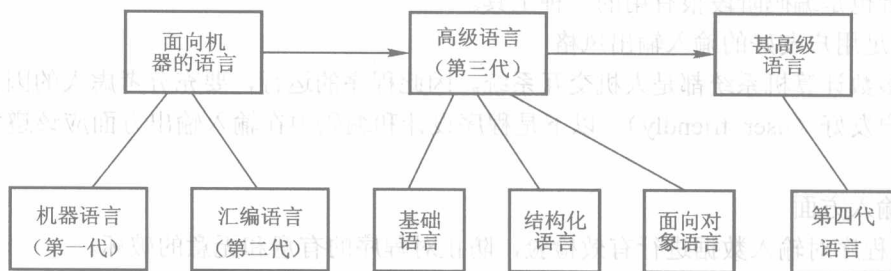


图 8.2 语言的发展和分类

1. 面向机器的语言

包括第一代机器语言和第二代汇编语言。这两代语言依赖于机器的结构，其指令系统随机器而异，难学难用。从软件工程的观点看，不仅生产率低，容易出错，而且维护困难。所以仅当计算机没有配置高级语言，或者个别特殊的场合，才会考虑使用它们。

2. 高级语言

1956年，第一个高级语言 FORTRAN 在美国诞生，成为最早的第三代程序设计语言（3GL）。有人统计，在同等条件下，一天生产的高级语言程序行数大致等于汇编语言程序的行数，但对于同一个问题，用高级语言书写的程序可比用汇编写出的程序缩短3~7倍，所以两者的生产率也要相差数倍。20世纪60年代末期，高级语言编码在科学计算领域已达到98%，但对于效率要求较高的实时系统，其比例还不到10%。在系统软件领域，当时还是汇编语言的独家天下。随着现代语言特别是C语言和Ada语言的出现，汇编语言在上述两个领域中的传统优势受到了严重的挑战。时至今日，除了使用高级语言确实不能满足软件需求的个别情

况外，已很少使用汇编语言编码。

从 FORTRAN、Pascal 到 C 语言，大多数常用的第三代语言都是面向过程的。随着面向对象程序设计的推广，在 20 世纪八九十年代，一批著名的常用语言扩展了面向对象的功能，出现了 C++、Object Pascal、Ada 95 等面向对象的第三代语言，以及全新的面向对象高级语言，如 Java、Eiffel 等。它们配合 OO 软件的开发，使编码语言有了更多的选择。

3. 第四代语言

40 余年来，高级语言的面貌发生了巨大变化，反映了人们对程序设计的认识从浅到深的过程。但是从根本上说，上述的通用语言仍都是过程化语言（procedural programming）。编码的时候，要详细描述问题求解的过程，告诉计算机每一步应该怎样做。为了把程序员从繁重的编码中解放出来，还需寻求进一步提高编码效率的新语言，这就是甚高级语言（VHLL）或第四代语言（4GL）产生的背景。

对于 4GL 语言，迄今仍没有统一的定义。一种意见认为，3GL 是过程化的语言，目的在于高效地实现各种算法；4GL 则是非过程化的语言，目的在于直接实现各类应用系统。前者面向过程，需要描述“怎样做”；后者面向应用，只需说明“做什么”。曾多年担任 IFIP 数据库专家组主席的 G. M. Nijssen 教授则主张，语言的划代应该以数据结构为标准。他认为，前 3 代语言的基础着重算法的描述，一次仅处理一个记录或数据元素，不支持对大量共享数据的处理。第四代和第五代语言应该以数据或知识为基础，以对集合的处理代替对于单个记录或元素的处理，能支持对大型数据库进行高效处理的机制。Nijssen 还认为，前 3 代语言是工业时代的产物，4GL 则标志了信息时代的开始。

综合上述的意见，4GL 应该具有以下特征：

① 它应该具有很强的数据管理能力，能对数据库进行有效的存取、查询和其他有关操作。

② 能提供一组高效的、非过程化的命令，组成语言的基本语句。编程时用户只需用这些命令说明“做什么”，不必描述实现的细节。

③ 能满足多功能、一体化的要求。为此，语言中除必须含有控制程序逻辑与实现数据库操作的语句外，还应包括生成与处理报表、表格、图形，以及实现数据运算和分析统计功能的各种语句，共同构成一个一体化的语言，以适应多种应用开发的需要。

有人把 IBM 公司的 SQL 关系数据查询语言称为最早的 4GL 语言。SQL 是在 Oracle、Sybase 和 DB2 等多种关系数据库上实现的基本语言，具有较强的数据操作功能。它采用高度非过程的命令式语句，方便易懂，既可在交互环境下独立运行，又可嵌入 COBOL、FORTRAN 等宿主语言中使用。但就语言本身来说，尚缺乏对屏幕表格、图形生成以及分析统计等功能的有力支持，还不是一体化的、能直接支持应用开发的 4GL 语言。与此相似，目前在微型机上流行的一些关系数据库语言，也仅具备部分 4GL 的特征。但不管怎样，这些语言的数据管理能力和非过程化命令语句，已经在应用开发中显示出明显的优越性。用这类语言来开发事务型应用软件，其开发时间和语句行数只相当于使用 BASIC 或 COBOL 语言的 1/5~1/10，

软件维护也较第三代语言方便。

有些文献把 UNIX 系统的 Shell 语言也归入甚高级语言。这是因为,它不仅具备高级语言的某些特点,而且能通过它所特有的命令组合、管道线、输入输出定向等功能,实现对库程序的拼接与重构。从支持快速的应用开发或原型软件开发来说,它与 4GL 有共同特点。但它并不具备上述的 4GL 主要的特征。

以上简述了程序设计语言的发展,下面再补充一点。在高级语言的应用中,人工智能也占有一席之地。最早的人工智能语言可追溯到 20 世纪 50 年代的 IPL 语言。随后出现的 LISP (50 年代后期)和 PROLOG(70 年代前期)等语言,都具有很高的知名度。有人曾经称 PROLOG 为第五代语言。但它主要是为新一代人工智能计算机设计的,故不属于本书的讨论范围。

8.2.2 常用的编程语言

高级语言种类繁多,总数已不下千种。从软件工程的角度,可以把它们分为基础语言、结构化语言和面向对象语言 3 大类。现简述如下。

1. 基础语言

FORTRAN、COBOL、BASIC 是这类语言的代表。之所以称它们为基础语言,是因为它们都有较长的使用历史,在国内外流传甚广,有大量已开发的软件,今天仍拥有广大的用户。这些语言创始于 20 世纪 50 年代或 60 年代,部分性能已趋老化,但随着版本的几次重大改进,除旧更新,至今仍被广泛使用。

(1) FORTRAN

它是使用最早的高级语言。从 1956 年到现在,始终保持着科学计算重要语言的地位。自 20 世纪 70 年代开始,我国先后引入了 FORTRAN 66、FORTRAN 77、FORTRAN 90 等新版本。从 FORTRAN 77 起,它在支持结构化与字符串处理等方面都作了较大改进,但数据类型仍不够丰富,也缺乏对复杂的数据结构支持。

(2) COBOL

作为商业数据处理中应用甚广的语言,它创始于 20 世纪 50 年代末期,后来又发表了 1972、1978 等国际标准文本。它广泛支持与事务数据处理有关的各种过程技术,使用近于自然语言的语句,虽然程序不够紧凑,但易于理解,从而受到企业、事业单位从业人员的欢迎。在美、日等国,它与 FORTRAN 并列为两大基础语言,长期拥有大量的用户。其主要不足是计算功能弱,编译速度也不够快。

(3) BASIC

它原是 20 世纪 60 年代初期为适应分时系统而研制的交互式语言,可用于一般数值计算与事务处理。由于它简单易懂,具有交互功能,成为许多初学者学习程序设计的入门语言,对计算机语言的普及起过巨大作用。其早期版本不支持结构程序设计,不区分数据类型,加上解释执行的速度较慢,不适用于较大型软件等原因,使之在 20 世纪 70 年代一度衰落。随着微型计算机的兴起,BASIC 又成为微机上配置最广的高级语言,出现了上百种不同的版本。

但这些版本各行其是，难于移植，被人称为“street BASIC”，认为不能登大雅之堂。

1985年，BASIC语言的原创始人在美国国家BASIC标准（1984）的基础上，研制了取名“True BASIC”的新版本。True BASIC保留了简单易学的特点，完全支持结构程序设计，增加或加强了绘图、窗口、矩阵运算等功能，给这一大众的语言再次注入了新的活力，在适应各种较大的应用课题上又进一步。

属于这类的基础语言还有ALGOL语言，包括ALGOL 60与ALGOL 68。其中ALGOL 60是我国引进最早的高级语言，也是20世纪70年代国内流行最广的语种之一。它对70年代出现的Pascal语言有强烈的影响，曾被认为是现代结构化语言的前驱。ALGOL 68由于过于庞大，在公布后不久就夭折了。

2. 结构化语言

20世纪70年代以来，在结构化程序设计影响下，先后出现了一批结构化语言，Pascal和C等语言就是其中著名的代表。

(1) Pascal

它是第一个系统地体现结构化程序概念的现代高级语言，是1970年由Wirth首先开发的。其最初的目标，是把它用作结构化程序设计的教学工具。由于它模块清晰，控制结构完备，有丰富的数据类型和数据结构，加上语言表达力强，移植容易，不仅被国内外许多高等学校采用作为教学语言，而且在科学计算、数据处理以及系统软件开发中都有较广的应用。1983年，美国正式公布了ANSI Pascal标准。随后便在从微型到大型的各类计算机上实现，并出现了一些有特色的微机Pascal语言（例如Turbo Pascal）。但由于Pascal不是为支持大型软件的开发设计的，现在软件开发中已很少使用。

(2) C语言

它是1973年由美国Bell实验室的Ritchie研制成功的。它起初是作为UNIX操作系统的主要语言开发的，现已成功地移植到多种微型与小型计算机上，成为独立于UNIX操作系统的通用程序设计语言。它除了具有结构化语言的公共特征，例如，表达简洁，控制结构与数据结构完备，有丰富的运算符和数据类型外，尤以移植力强、编译质量高等特点吸引了人们的注意。用C编译程序产生的目标程序，其质量可以与汇编语言产生的目标程序媲美。所以有人称之为“可移植的汇编语言”，因为它既具有汇编语言的高效率，又不像汇编语言那样只能束缚在某种处理机上运行。C语言的这些特点，使它不仅能写出效率高的应用软件，也适用于编写操作系统、编译程序等系统软件。著名的UNIX操作系统，就有90%以上的代码是用C语言书写的。

(3) Ada语言

它是集FORTRAN以来各种语言之大成的语言。其开发计划始于1975年，由美国国防部直接领导。1983年公布了Ada的美国国家标准和军用标准。1985至1986年，法、英等国和我国都相继宣布将Ada作为军用的通用语言。

Ada的目标是适用于一切嵌入式（embedded）计算机系统。它既有许多经典的语言特征，

也包含许多新的特征。为了满足实时应用，它支持并发处理与过程间的通信，支持在异常处理中实施中断，并能支持通常只能由汇编语言实现的低级操作，如位操作、字节操作等。在语言的表达与结构上，它又具有高级语言的特点，远比汇编语言容易开发和维护。Ada 还是一个充分体现软件工程思想的语言，既是编码语言，又可用作设计表达工具。Ada 提供的多种程序单元（包括子程序、程序包、任务与类属），与实现相分离的规格说明，以及分别编译等，既支持大型软件的开发，也为采用现代开发技术开发提供了便利。Ada 还明确规定了对支持环境的要求，能向软件的开发与维护提供全生存周期的支持。

3. 面向对象的语言

(1) C++语言

C++是从C语言进化而来，是C语言的超集。1980年，为了满足管理程序的复杂性的需要，贝尔实验室的Bjame Stroustrup开始对C进行改进和扩充，1983年正式取名为C++。经过3次修订后，于1994年指定了ANSI C++标准的草案。以后又经过不断完善，成为目前的C++。C++在程序结构的本质上与C语言是一致的，都是用函数驱动机制来实现。因此它既可以进行过程化程序设计，也可以进行面向对象程序设计。

(2) Java语言

Java语言是当今流行的新兴网络编程语言，它的面向对象、跨平台、分布应用等特点给编程人员带来了一种崭新的计算概念，使WWW从最初的单纯提供静态信息，发展到现在可提供各种各样的动态服务，发生了巨大的变化。Java不仅能够编写小应用程序以实现嵌入网页的声音播放和动画功能，而且还能够应用于独立的大、中型应用程序，其强大的网络功能能够把整个Internet作为一个统一的运行平台，极大地拓展了传统单机或C/S模式应用程序的外延和内涵。自从1995年正式问世以来，Java已经逐步从一种单纯的计算机高级编程语言发展为一种重要的Internet平台，并进而引发、带动了Java产业的发展和壮大，成为当今计算机业界不可忽视的计算机环境和重要的发展潮流与方向。

(3) C#语言

作为一种简洁、类型安全的面向对象的语言，C#是可以让开发人员快速地建立较大规模的基于Microsoft网络平台的应用，并且提供大量的开发工具和服务，帮助开发人员开发基于计算和通信的各种应用。C#也可以为C/C++开发人员提供快速的开发手段而不需要牺牲C/C++语言的特点/优点。从继承角度来看，C#在更高层次上重新实现了C/C++，熟悉C/C++开发的人员可以很快转变为C#开发人员。使用C#，可以创建传统的Windows客户端应用程序、XML Web Services、分布式组件、客户-服务器应用程序、数据库应用程序以及很多其他类型的程序。

8.2.3 编码语言的选择

1. 为什么要选择编码语言

D. A. Fisher说过，“程序设计语言不是引起软件问题（software problems）的原因，也不

能用它来解决软件问题。但是，由于语言在一切软件活动中所处的中心地位，它们能使现存的问题变得较易解决，或者更加严重。”这段话，言简意赅地揭示了语言在软件开发中的作用，提醒人们重视在编码以前选择适当的语言。

在编程时，每个人都习惯于使用自己熟悉的语言。但对于规模较大的程序，就要选择适当的语言进行设计。目前计算机上配置的语言越来越多，连中、低档微型机也配有多种语言。选择一种适当的编码语言，是设计与编码人员义不容辞的责任。

2. 选择编码语言的标准

语言的选择，应从何处入手呢？首先，要确定求解的问题对编码有哪些要求，把它们按轻重次序列出。然后，用这些要求衡量可提供的语言，判断哪几种语言能较好地满足它们。没有一种语言能等量地满足各种不同的要求。所以在作出选择时，必须优先考虑主要的要求，然后适当照顾其余的方面。

当衡量某一语言是否可选作编码语言时，常使用以下几项作为评价标准：

① 应用领域。各种语言都有自己的适用领域。在科学计算领域，FORTRAN 仍占优势，虽然 Pascal 与 BASIC 也常常使用。在事务处理方面，COBOL 与 BASIC 可能是合理的选择。Ada 语言或汇编语言适用于实时应用，C 语言或汇编语言适用于系统软件开发。如果开发的软件中含有大量数据操作，则采用 SQL、FoxPro 等数据库语言往往更为适宜。

② 算法与计算复杂性。FORTRAN、True BASIC 及各种现代语言，都能支持较复杂的计算与算法。但 COBOL 与大多数数据库语言，只能支持简单的运算。

③ 数据结构的复杂性。C、C#、C++ 和 Java 语言都支持数组、记录（在 C 语言中称为结构）与带指针的动态数据结构，适用于系统程序开发和需要复杂数据结构的应用程序开发。BASIC、FORTRAN 等语言只能提供简单的数据结构——数组。

④ 效率的考虑。有些实时应用要求系统具有快速的响应速度，此时可酌情选用汇编或 Ada 语言。一些典型调查还表明，一个程序的执行时间，往往有一大部分是耗费在一小部分代码上的。为了提高效率，只需将这部分代码改用汇编语言，其余代码仍可使用高级语言。有报导称，在一次实验中把高级语言程序的 5% 用汇编语言改写，执行时间就缩短了 1/3。在另 4 次试验中，他们把 12% 的高级语言编码用汇编语言替代，执行速度平均提高了 5 倍。在开发资源紧张的微处理机软件时，还可以使用像 Forth 那样的专用语言来提高软件的时空效率。

各编码语言适用的应用领域如表 8.2 所示。

表 8.2 适用各类应用领域的语言

年 代	应用 领域	主 要 语 言	其 他 语 言
20 世纪 60 年代	商业	COBOL	Assembler
	科学计算	FORTRAN	ALGOL、BASIC、APL
	系统	Assembler	Forth
	人工智能	LISP	SNOBOL

续表

年 代	应用 领域	主 要 语 言	其 他 语 言
现代	商业	COBOL、C#、C++、Java、电子表格	C、PL/1
	科学计算	FORTRAN、C、C++、Java	BASIC
	系统	C、C++、Java	Ada、Modula
	人工智能	LISP、PROLOG	

8.2.4 编码工具

近 10 多年来,国外不少软件公司在 4GL 的影响下,推出了一些快速开发的编码工具,其中流行较广的有 Eclipse、NetBeans、Visual Studio、Delphi、PowerBuilder 等。它们一般都限定于某些特定的应用领域(如数据库应用、网络开发),或支持某种编程特色(如可视化编程),因易学易用而受到用户欢迎。但它们多数只具有通用语言的部分特征,只能算作一种编程工具或环境。

一些高级的模型工具同时也拥有编码环境,例如 IBM 公司的 RAD (Rational Application Developer) 等,可以直接将设计模型转换为源代码。

8.3 编码示例

由于篇幅的原因,这里以“网上购物系统”的部分功能为例,介绍如何将设计模型转换为源代码。并选择 Java 作为编码语言,以例子的形式,对第 7.5.3 节中注册和维护购物车的设计模型进行编码。

8.3.1 注册功能编码实现

[例 8.1] 为如图 7.38 所示的注册部分的类图进行编码。

[解] 在图 7.38 所示的类图中,共有 3 个类:注册表单类 (RegisterForm)、注册控制者类 (RegistController) 和顾客类 (Customer)。由于“网上购物系统”是一个基于 Web 浏览器的应用,可以利用 J2EE 技术来实现,注册表单类一般用 JSP 或 HTML 编写,注册控制者类则实现为一个 Java Servlet 类,顾客类是一个普通的 Java 类。为了清楚地理解本例,读者最好先参考 J2EE 相关书籍,了解 JSP、Servlet 和 JavaBean 等知识。下面给出注册控制者类和顾客类的源代码。

注册控制者类所对应的源代码如下:

```
import javax.servlet.*;
import javax.servlet.http.*

public class RegistController extends HttpServlet {
```

```

private Customer customer;

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    /* 添加一条顾客记录 */
    regCustomer(request);
}

/**
 * 添加顾客
 * @param request
 */
public void regCustomer(HttpServletRequest request) {
    String name = request.getParameter("customerName");
    // 获得表单中顾客填写的用户名
    if (!customer.hasCustomer(name)) {
        String customerPassword = request.getParameter("customerPassword");
        String customerAddr = request.getParameter("customerAddr");
        String customerEmail = request.getParameter("customerEmail");
        customer.addCustomer(name, customerPassword, customerAddr,
            customerEmail);
    } else {
        request.setAttribute("fail", "该用户名已经存在");
    }
}
}

```

说明：为了能通过 HTTP 协议接收注册表单类发送的消息，注册控制者类继承了 `HTTPServlet` 类，`HTTPServlet` 是 `Servlet` 类的一个子类，`Servlet` 是运行在服务器端的小程序的 Java API。注册控制者类新定义了一个属性 `customer`，实现了注册控制者类和顾客类之间的关联关系；根据图 7.38 所示的设计，注册控制者类定义了一个注册顾客的操作 `regCustomer()`，在 `HTTPServlet` 的 `doGet()` 方法中调用。

为了实现顾客信息的添加和访问，必须进行数据库操作。为此，在给出顾客类的代码前，先给出一个利用 JDBC 进行数据库连接的类 `JDBCConnection`（本书第 7 章介绍了 JDBC 连接数据库的方法），然后给出顾客类的代码，如下所示：

```

import java.sql.*;

public class JDBCConnection {
    public static Connection getConnection() {
        Connection connection = null;
        String dbDriver = "com.mysql.jdbc.Driver";

```

```
String url = "jdbc:mysql://IP:Port/onlineDb";
try {
    Class.forName(dbDriver).newInstance();
    connection = DriverManager.getConnection(url, "USR", "PWD");
} catch (Exception e) {
    e.printStackTrace();
}
return connection;
}
```

顾客类的代码如下：

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Customer {
    private String customerID;
    private String customerName;
    private String customerPassword;
    private String customerAddr;
    private String customerEmail;
    private static Connection con = JDBCConnection.getConnection();

    public Customer()
    {
    }

    public Customer(String customerID, String customerName, String customerPassword, String
        customerAddr, String customerEmail)
    {
        this.customerID=customerID;
        this.customerName=customerName;
        this.customerPassword=customerPassword;
        this.customerAddr=customerAddr;
        this.customerEmail=customerEmail;
    }

    public String getCustomerAddr() {
        return customerAddr;
    }
}
```

```
public void setCustomerAddr(String customerAddr) {
    this.customerAddr = customerAddr;
}

public String getCustomerEmail() {
    return customerEmail;
}

public void setCustomerEmail(String customerEmail) {
    this.customerEmail = customerEmail;
}

public String getCustomerID() {
    return customerID;
}

public void setCustomerID(String customerID) {
    this.customerID = customerID;
}

public String getCustomerName() {
    return customerName;
}

public void setCustomerName(String customerName) {
    this.customerName = customerName;
}

public String getCustomerPassword() {
    return customerPassword;
}

public void setCustomerPassword(String customerPassword) {
    this.customerPassword = customerPassword;
}

/**
 * 通过用户名查找数据库中是否已存在用户名，即验证顾客信息
 * @param name
```

```
* @return
*/
public boolean hasCustomer(String name) {
    PreparedStatement ps = null;
    try {
        ps = con.prepareStatement("select * from customer where customerName=?");
        ps.setString(1, name);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            return true;
        }
        rs.close();
        ps.close();
        con.close();
    } catch (SQLException ex) {
    }
    return false;
}

/**
 * 添加顾客
 *
 * @param customerName
 * @param customerPassword
 * @param customerAddr
 * @param customerEmail
 */
public void addCustomer(String customerName,
    String customerPassword, String customerAddr, String customerEmail)
{
    PreparedStatement ps = null;
    try {
        ps = con.prepareStatement("insert into customer values(?,?,?,?)");
        // customer 是对应数据库中的顾客表
        ps.setString(1, customerName);
        ps.setString(2, customerPassword);
        ps.setString(3, customerAddr);
        ps.setString(4, customerEmail);
        ps.executeUpdate();
        ps.close();
        con.close();
    }
}
```

```
        } catch (SQLException ex) {
        }
    }
}

/**
 * 根据顾客的 ID 号获得客户信息
 *
 * @param customerID
 * @return
 */
public Customer getCustomerInfo(String customerID) {
    PreparedStatement ps = null;
    try {
        Customer cust = new Customer();
        ps = con.prepareStatement("select * from customer where id=?");
        ps.setString(1, customerID);
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            cust.setCustomerName(rs.getString("customerName"));
            cust.setCustomerPassword(rs.getString("customerPassword"));
            cust.setCustomerAddr(rs.getString("customerAddr"));
            cust.setCustomerEmail(rs.getString("customerEmail"));
        }
        rs.close();
        ps.close();
        con.close();
        return cust;
    } catch (SQLException ex) {
        ex.printStackTrace();
        return null;
    }
}

/**
 * 更新顾客信息
 *
 * @param customerID
 * @param customerName
 * @param customerPassword
 * @param customerAddr

```



```

    * @param customerEmail
    */
    public void updateCustomer(String customerID, String customerName,
        String customerPassword, String customerAddr, String customerEmail)
    {
        PreparedStatement ps = null;
        try {
            ps = con.prepareStatement("update customer set customerName=?,
                customerPassword=?,customerAddr=?,customerEmail=? where id=?");
            ps.setString(1, customerName);
            ps.setString(2, customerPassword);
            ps.setString(3, customerAddr);
            ps.setString(4, customerEmail);
            ps.setString(5, customerID);
            ps.executeUpdate();
            ps.close();
            con.close();
        } catch (SQLException ex) {
        }
    }
}

```

说明：顾客类实现了图 7.38 中定义的属性和操作。此外，为了执行数据库操作，增加了一个数据库连接属性 con，除了属性的 set 和 get 操作外，与用户注册相关的操作有两个：hasCustomer()和 addCustomer()，其他操作将在实现其他功能时使用。

8.3.2 维护购物车功能编码实现

[例 8.2] 为如图 7.40 所示的维护购物车部分的类图进行编码。

[解] 在图 7.40 所示的类图中共有 4 个类：维护购物车表单类 (MaintainCartForm)、维护购物车控制者类 (MaintainCartController)、购物车类 (Cart) 和购物项目类 (SellComItem)。同样地，维护购物车表单类用 JSP 或 HTML 编写，维护购物车控制者类则实现为一个 Java Servlet 类，购物车类是一个普通的 Java 类，而购物项目类实现为一个 JavaBean。其中购物车类和购物项目类之间存在一对多的数量关系，这里用容器类 Vector 实现。

下面给出维护购物车控制者类、购物车类和购物项目类的源代码。

维护购物车控制者类源代码如下：

```

import java.io.IOException;
import java.util.Vector;
import javax.servlet.http.*;

```

```
public class MaintainCartController extends HttpServlet {
    private Cart cart;
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession();
        cart = (Cart) session.getAttribute("cart");
        // 获取 session 中的 Cart 对象,即当前用户的购物车
        updateCart(request);
        String selectedRow = request.getParameter("checkdel");
        // 获得被选中的某一行商品的行号, 第一行行号为 0
        if(selectedRow!=null)
            deleteCommodity(selectedRow);
    }

    /**
     * 更新购物车, 修改购物车中商品的数量
     * @param req
     */
    public void updateCart(HttpServletRequest req) {
        for (int i = 0; i < cart.getSize(); i++) {
            String number = req.getParameter("num" + i);
            // 获得前台页面对应商品列表中第 i+1 行商品数量的值
            int num = Integer.parseInt(number);
            SellComItem scf = cart.getCommodity(i);
            scf.setAmount(num);
            cart.updateCartCommodity(scf, i);
        }
    }

    /**
     * 删除购物车中的商品
     *
     * @param selectedRow
     */
    public void deleteCommodity(String selectedRow) {
        int selectedRowNum = Integer.parseInt(selectedRow);
        cart.deleteCommodity(selectedRowNum);
    }
}
```

说明: 维护购物车控制者类有一个属性 `cart`, 实现了它与购物车类的关联关系, 主要操作有两个: 删除商品 `deleteCommodity()`和更新购物车 `updateCart()`。

购物车类源代码如下：

```
import java.util.Vector;

public class Cart {
    private Vector<SellComItem> clist;

    public Vector<SellComItem> getlist() {
        return clist;
    }

    public void setClist(Vector<SellComItem> clist) {
        this.clist = clist;
    }

    /**
     * 所有商品的总计金额
     *
     * @return
     */
    public float getSum() {
        float sum = 0;
        for (int i = 0; i < clist.size(); i++) {
            sum = sum + clist.get(i).getPrice() * clist.get(i).getAmount();
        }
        return sum;
    }

    /**
     * 向购物车中增加商品
     *
     * @param SellComItem
     */
    public void addCommodity(SellComItem SellComItem) {
        clist.add(SellComItem);
    }

    /**
     * 获取购物车中某一商品信息
     *
     * @param i
```

```
    /** @return 购物车中的商品的名称 */
    */
    public SellComItem getCommodity(int i) {
        return clist.get(i);
    }

    /**
     * 修改某一商品
     *
     * @param SellComItem
     * @param i
     */
    public void updateCartCommodity(SellComItem SellComItem, int i) {
        clist.set(i, SellComItem);
    }

    /**
     * 删除购物车中的某一商品
     *
     * @param i
     */
    public void deleteCommodity(int i) {
        clist.remove(i);
    }

    /**
     * 清空购物车
     *
     */
    public void clearCart() {
        clist.clear();
    }

    /**
     * 获取购物车的大小
     * @return
     */
    public int getSize() {
        return clist.size();
    }
}

```

说明：购物车类包括一个容器属性 `clist`，表示购物车中的商品列表，实现了购物车类和商品列表类的一对多的关联关系，它实现了各种对购物车的操作。

购物项目类源代码如下：

```
public class SellComItem {
    private String commodityID;
    private String commodityName;
    private float price;
    private int amount;
    private float totalprice;

    public SellComItem() {
    }
    public int getAmount() {
        return amount;
    }
    public void setAmount(int amount) {
        this.amount = amount;
    }
    public String getCommodityID() {
        return commodityID;
    }
    public void setCommodityID(String commodityID) {
        this.commodityID = commodityID;
    }
    public String getCommodityName() {
        return commodityName;
    }
    public void setCommodityName(String commodityName) {
        this.commodityName = commodityName;
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
        this.price = price;
    }
    public float getTotalprice() {
        return totalprice;
    }
    public void setTotalprice(float totalprice) {
        this.totalprice = totalprice;
    }
}
```

说明：此类用于表示保存购物信息的购物项目，它实现为一个 `JavaBean`。

8.4 测试的基本概念

软件测试 (software testing) 是动态查找程序代码中的各类错误和问题的过程。随着人类对计算机应用的逐步深入,人们对软件的要求也越来越高,“软件只是能用还不够,必须好用”,“不是人适应软件,而是软件适应人”等观念已经成为人们的共识。这样,软件测试显得越来越重要。

8.4.1 目的与任务

G. J. Myers 在他的名著《软件测试技巧》一书中,给出了测试的定义:“程序测试是为了发现错误而执行程序的过程”。根据这一定义,测试 (testing) 的目的与任务可以描述为:

- 目的: 发现程序的错误。
- 任务: 通过在计算机上执行程序,暴露程序中潜在的错误。

另一个与测试相关的术语是纠错 (debugging)。它的目的与任务可以规定为:

- 目的: 定位和纠正错误。
- 任务: 消除软件故障,保证程序的可靠运行。

测试与纠错的关系,可以用图 8.3 的数据流图来说明。图中表明,每一次测试都要准备好若干必要的测试数据,与被测程序一起送入计算机执行。通常把一次程序执行需要的测试数据,称为一个测试用例 (test case)。每一个测试用例产生一个相应的测试结果。如果它与期望结果不相符合,便说明程序中存在错误,需要通过纠错来改正。



图 8.3 测试和纠错信息流程

对于长度仅有数百行的小程序,测试与纠错一般由编码者一人完成;但对于大型的程序,测试与纠错必须分开进行。为了保证大程序的测试不受干扰,通常都把它交给独立的小组进行,对于程序中的错误,应退回编码者进行纠错。

有人把小程序的测试和纠错合称为调试。一旦编好了一个小程序,先在计算机上运行测试,发现错误便进行纠错,修改后再重复测试。这一交替进行的“测试—纠错—再测试—再纠错”的过程,常被通俗地称为调试程序。其实调试在英语中并无相当的词。但纠错时有一种常用的软件工具 Debugger,通常被译为调试程序,而不译为纠错程序。

8.4.2 测试的特性

与分析、设计和编码等工作相比，程序测试具有若干特殊的性质。了解这些性质，将有助于正确处理和做好测试工作。

1. 挑剔性

测试是对质量的监督与保证，所以“挑剔”和“揭短”很自然地成为测试人员奉行的信条。Myers 说得好，测试是为了证明程序有错，而不是证明程序无错。因此，对于被测程序就是要“吹毛求疵”，就是要在“鸡蛋里面挑骨头”。只有抱着为证明程序有错的目的去测试，才能把程序中潜在的大部分错误找出来。

2. 复杂性

人们常以为开发一个程序是困难的，测试一个程序则比较容易。这其实是误解。设计测试用例是一项需要细致和高度技巧的工作，稍有不慎就顾此失彼，发生不应有的疏漏。

举一个极简单的例子。假如一个程序的功能是输入 3 个数作为三角形的三条边，然后鉴别这一三角形的类别。输入 3 个 5 时程序应回答“等边三角形”，但若输入 3 个 0 程序也回答“等边三角形”，就是真假不分了。又如，三条边分别为 2、3、4 时应判断是不等边三角形，但如果对 2、3、5 或 2、3、6 也判断为不等边三角形，就会闹出笑话了。可见在设计测试用例时，需要考虑各种可能的情况，对被测程序进行多方面的考核。切忌挂一漏万，把原本复杂的问题想得过于简单。小程序尚且如此，大型程序就可想而知了。所以有人认为，做好一个大型程序的测试，其复杂性不亚于对这个程序的开发，主张挑选最有才华的程序员参与测试工作。

3. 不彻底性

E. W. Dijkstra 有一句名言：“程序测试只能证明错误的存在，不能证明错误不存在”。这句话貌似夸张，其实却揭示了测试所固有的一个重要性质——不彻底性。

可能有读者认为，测一遍不彻底可以测十遍，十遍还不彻底可测百遍。需要多少测试用例就用多少测试用例，难道还怕达不到彻底吗？为了回答这个问题，请读者看一个例子。假如有人开发了一个 C 语言的编译程序，要对它进行彻底的测试，需要设计多少个测试用例呢？从正面说，该编译程序应能把一切符合语法的 C 程序正确地翻译为目标程序；从反面说，它应对一切有语法错误的 C 程序指出程序的错误。显然，这两个“一切”都是无穷量，是无法实现的。

所谓彻底测试，就是让被测程序在一切可能的输入情况下全部执行一遍。通常称这种测试为穷举测试 (exhaustive testing)。在实际测试中，穷举测试要么像上例那样根本无法实现，要么工作量太大（例如一个小程序要连续测试许多年），在实践上行不通。这就注定一切实际测试都是不彻底的，当然也就不能够保证测试后的程序不存在遗留的错误。

4. 经济性

既然穷举测试行不通，所以在程序测试中，总是选择一些典型的、有代表性的测试用例，

进行有限的测试。通常把这种测试称为选择测试 (selective testing)。为了降低测试成本 (一般占整个开发成本的 1/3 左右), 选择测试用例时应注意遵守经济性的原则: 第一, 要根据程序的重要性和一旦发生故障将造成的损失来确定它的可靠性等级, 不要随意提高等级, 从而增加测试的成本; 第二, 要认真研究测试策略, 以使用尽可能少的测试用例, 发现尽可能多的程序错误。

8.4.3 测试的种类

按照 Myers 的定义, 测试是一个执行程序的过程, 即要求被测程序在计算机上运行。其实, 不执行程序也可以发现程序的错误。为便于区分, 一般把前者称为动态测试, 后者称为静态分析 (static analysis)。广义地说, 它们都属于程序测试, 测试分类如图 8.4 所示。

顾名思义, 静态分析就是通过对被测程序的静态审查, 发现代码中潜在的错误。它一般用人工方式脱机完成, 故亦称人工测试或代码评审 (code review); 也可借助于静态分析器在计算机上以自动方式进行检查, 但不要求程序本身在计算机上运行。按照评审的不同组织形式, 代码评审又可区分为代码会审、走查和办公桌检查 3 种。对某个具体的程序, 通常只使用一种评审方式。

动态测试也可区分为两类。一类把被测程序看成一个黑盒, 根据程序的功能来设计测试用例, 称为黑盒测试 (black box testing); 另一类则根据被测程序的内部结构设计测试用例, 测试者需事先了解被测程序的结构, 故称为白盒测试 (white box testing)。

为什么有了动态测试, 还要有代码评审呢? 简单地说, 这是因为程序中往往存在不同性质的错误, 有些适宜在计算机测试中发现, 另一些更易于在人工测试中揭露。

8.4.4 测试的文档

为了保证测试质量, 软件测试必须完成规定的文档。按照软件工程的要求, 测试文档主要包括测试计划和测试报告两个方面的内容。

测试计划的主体是测试内容说明。它包括测试项目的名称, 各项测试的目的、步骤和进度, 以及测试用例的设计等。测试报告的主体是测试结果, 它包括测试项目名称, 实测结果与期望结果的比较, 发现的问题, 以及测试达到的效果等。

前文已经指出, 测试用例就是测试时选用的例子。仍以前面说过的鉴别三角形分类程序为例, 每输入 3 个数据作为三角形的边长 (例如 5,5,5), 就可以预期从程序输出三角形的一种类别 (例如等边三角形)。由此可见, 一个程序所需的测试用例可以定义为:

测试用例 = {测试数据 + 期望结果}

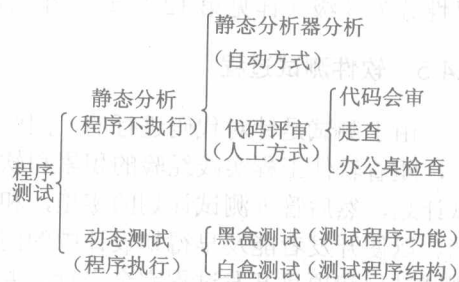


图 8.4 测试的分类

其中的{}表示重复。它表明,测试一个程序要使用多个测试用例,而每一个测试用例都应包括一组测试数据和一个相应的期望结果。如果在测试用例后面再加上“实际结果”,就成为测试结果,即

$$\text{测试结果} = \{\text{测试数据} + \text{期望结果} + \text{实际结果}\}$$

由此可见,测试用例不仅是连接测试计划与执行的桥梁,也是软件测试的中心内容。有效地设计测试用例,是做好软件测试的关键。在下节中,将向读者详细介绍测试用例的设计方法。

前已指出,测试所要求的完善程度还与程序的可靠性等级有关。可靠性要求高的程序,终止测试的标准也应定得高一些,当然开发成本将相应地增加。B. W. Boehm 建议将程序可靠性分为5级(详见第12.3节),在具体确定程序的测试终止标准时,可作参考。

8.4.5 软件测试过程

由于测试是针对代码进行的,因此很多人都认为测试只是编码之后的一个阶段。其实不然,随着软件工程实践经验的积累和软件的产品化,项目一开始就要考虑测试过程,制定测试计划,然后管理测试计划的实施,和项目开发计划的实施相配合,写出测试报告并及时归档,以便开发者能及早得知设计中的问题。测试为软件开发的各项活动提供全方位的服务。测试过程和项目开发过程完全平行,并有机地交互,将测试出的问题纳入项目的风险和进度分析中,以调整下一步的开发和测试活动。

软件测试从项目开发的一个阶段上升为一个测试过程,是随着软件开发的发展而逐步发展起来的。20世纪60年代,为了提高编写的代码的正确性,有人提出了“编一点、测一点”,这比全部编码完成后再测试要容易,因此,测试和编码无明显的先后之分。随着软件测试工具的大量使用,详细设计时对设计实现的关键算法、重要接口先进行临时编码并测试,待验证后纳入设计,重要的计算逻辑已经作了初步验证,设计就更可靠;以后随着类库、软件构件的增多,在分析设计阶段也要测试部分关键的软件构件的计算逻辑或接口,才能决定是否采用该软件构件,所以,分析和设计阶段也包含了测试。正是以大量软件构件和良好的测试工具为后盾,在需求分析和定义需求时,测试可以辅助判定需求的可行性,辅助需求演进,因为现代软件中很大比例是复用代码,定义好需求和解决方案的体系结构设计,远比模块实现重要得多,需求定义交付的不仅是本应用的体系结构,甚至初步的用户界面也要交付,说明如何满足使用需求和设计生成用户界面时也免不了测试,所以测试就提前到了需求阶段。

和软件开发过程一样,测试过程安排得好可以节省大量的人力物力,降低总的开发成本。软件工程越来越复杂,一个项目首先要做开发过程的定制或设计,然后做测试的需求和设计,再后才是测试实施。定制测试过程,是软件工程成熟的表现。

8.5 黑盒测试和白盒测试

和其他软件开发活动不同的是,在过去几十年中,软件测试的最基本技术没有大的变化。

本节介绍经典的软件测试技术：黑盒测试和白盒测试。

8.5.1 黑盒测试

黑盒测试就是根据被测试程序功能来进行测试，所以也称为功能测试。用黑盒法设计测试用例，有4种常用技术，本节仅介绍其中的3种。

1. 等价分类法 (equivalence partitioning)

所谓等价分类，就是把输入数据的可能值划分为若干等价类，使每类中的任何一个测试用例，都能代表同一等价类中的其他测试用例。换句话说，如果从某一等价类中任意选出一个测试用例未能发现程序的错误，就可以合理地认为使用该类中的其他测试用例也不会发现程序的错误。这样，就把漫无边际的随机测试变成有针对性的等价类测试，有可能用少量有代表性的例子来代替大量内容相似的测试，借以实现测试的经济性。

采用这一技术要注意以下两点：其一是划分等价类不仅要考虑代表有效输入值的有效等价类，还需考虑代表无效输入值的无效等价类；其二是每一无效等价类至少要用一个测试用例，不然就可能漏掉某一类错误，但允许若干有效等价类合用同一个测试用例，以便进一步减少测试的次数。

【例 8.3】 某工厂公开招工，规定报名者年龄应在 16 周岁至 35 周岁之间（到 2008 年 3 月止）。若出生年月不在上述范围内，将拒绝接受，并显示“年龄不合格”等出错信息。试用等价分类法设计对这一程序功能的测试用例。

【解】 第一步：划分等价类。假定已知出生年月由 6 位数字字符表示，前 4 位代表年，后 2 位代表月，则可以划分为 3 个有效等价类、7 个无效等价类，如表 8.3 所示。

表 8.3 “出生年月”的等价分类

输入数据	有效等价类	无效等价类
出生年月	① 6 位数字字符	② 有非数字字符 ③ 少于 6 个数字符 ④ 多于 6 个数字符
对应数值	⑤ 在 197302~199203 之间	⑥ <197302 ⑦ >199203
月份对应数值	⑧ 在 1~12 之间	⑨ =0 ⑩ >12

第二步：设计有效等价类需要的测试用例。表 8.3 中的①、⑤、⑧等 3 个有效等价类，可以共用一个测试用例，如表 8.4 所示。

表 8.4 有效等价类的测试用例

测试数据	期望结果	测试范围
197511	输入有效	①、⑤、⑧

第三步：为每一无效等价类至少设计一个测试用例。本例具有7个无效等价类，至少需要7个测试用例，如表8.5所示。

表8.5 无效等价类的测试用例

测试数据	期望结果	测试范围
MAY, 75	输入无效	②
19755	输入无效	③
1978011	输入无效	④
195512	年龄不合格	⑥
199606	年龄不合格	⑦
198200	输入无效	⑨
197522	输入无效	⑩

让几个有效等价类共用一个测试用例，可以减少测试次数，有利而无弊；但若几个无效等价类合用一个测试用例，就可能使错误漏检。就上例而言，假定把“195512”（对应无效等价类⑥）和“198200”（对应无效等价类⑨）合并为一个测试用例，例如“195500”，即使在测试过程中程序显示出“年龄不合格”的信息，仍不能证明程序对月份为“00”的输入数据也具有识别和拒绝接受的功能。再进一步讲，其实在第一步“划分等价类”时，就应防止有意或无意地将几个独立的无效等价类写成一个无效等价类。例如，若在上例中把⑨、⑩两个无效等价类合并写成“末两位的对应值为0或>12”，则与之相应的测试用例也将从原来的两个减为一个，对程序的测试就不够完全了。

2. 边界值分析法（boundary value analysis）

实践表明，程序员在处理边界情况时，很容易因疏忽或考虑不周发生编码错误。例如，在数组容量、循环次数以及输入数据与输出数据的边界值附近程序出错的概率往往较大。采用边界值分析法，就是要这样来选择测试用例，使得被测程序能在边界值及其附近运行，从而更有效地暴露程序中隐藏的错误。为帮助读者理解这一方法，请重温例8.3。

【例8.4】 程序设计要求同例8.3。试用边界值分析法设计其测试用例。

【解】 用等价分类法设计测试用例时，测试数据可以在等价类值域内任意选取。就拿例8.3来说，为了只接受年龄合格的报名者，程序中可能设有语句：

```
if(197302<=value(birthdate)<=199203)
  then read (birthdate)
  else write "invalid age"
```

但如果编码时把以上语句中的“<=”误写为“<”，用例8.3中的所有测试用例都不能发现这种错误。所谓边界值分析，就是要把测试的重点放在各个等价类的边界上，选取刚好等于、刚刚大于和刚刚小于边界值的数据为测试数据，并据此设计出相应的测试用例。

从例8.3可知，本例有3个有效等价类，即出生年月、对应数值、月份对应数值。采用

边界值分析法, 可为这 3 个有效等价类选取 14 个边界值测试用例 (其中有 2 个重复, 实有 13 个), 其内容如表 8.6 所示。

表 8.6 “出生年月”的测试用例(边界值分析法)

输入等价类	测试用例说明	测试数据	期望结果	选取理由
出生年月	① 1 个数字字符	5	输入无效	仅有一个合法字符
	② 5 个数字字符	19705		比有效长度恰少一个字符
	③ 7 个数字字符	1968011		比有效长度恰多一个字符
	④ 有 1 个非数字字符	19705X		非法字符最少
	⑤ 全是非数字字符	AUGUST	输入有效	非法字符最多
	⑥ 6 个数字字符	197302		类型与长度均有效的输入
对应数值	⑦ 35 周岁	197302	合格年龄	最大合格年龄
	⑧ 16 周岁	199203		最小合格年龄
	⑨ >35 周岁	197301	不合格年龄	恰大于合格年龄
	⑩ <16 周岁	199204		恰小于合格年龄
月份对应数值	⑪ 月份为 1 月	197302	输入有效	最小月份
	⑫ 月份为 12 月	199203		最大月份
	⑬ 月份<1	197300	输入无效	恰小于最小月份
	⑭ 月份>12	197413		恰大于最大月份

说明: 表中第⑥、⑦两个测试用例数据相同, 可合用一个测试用例。

将例 8.3 与例 8.4 进行比较, 可见:

① 等价分类法的测试数据是在各个等价类允许的值域内任意选取的, 而边界值分析的测试数据必须在边界值附近选取。

② 例 8.3 用了 8 个测试用例, 而例 8.4 用了 13 个。一般的说, 用边界值分析法设计的测试用例比等价分类法的代表性更广, 发现错误的能力也更强。但是对边界的分析与确定比较复杂, 要求测试人员具有更多的经验和创造性。

还需指出, 例 8.4 包含的边界情况比较简单, 只需要分析输入等价类。在有些情况下, 除了考察输入值边界外, 还需要考察输出值和其他可能存在的边界。例如, 假定被测程序是一个计算 X 的正弦值的函数 $\text{Sin}(X)$, 其输出具有 3 个边界值 -1、0 和 1, 则在选择测试用例时, 应使 X 的值能分别产生上述的 3 种输出边界值, 即选取 $-\frac{\pi}{2}$ 、0 和 $\frac{\pi}{2}$ 作为 X 的测试数据。

3. 错误猜测法 (error guessing)

所谓猜错, 就是猜测被测程序在哪些地方容易出错, 然后针对可能的薄弱环节来设计测试用例。显然, 它比前两种方法更多地依靠测试人员的直觉与经验。所以, 一般都先用前两种方法设计测试用例, 然后用猜错法补充一些例子作为辅助的手段。

仍以例 8.3 的程序设计要求为例, 在已经用等价分类法 (见例 8.3) 和边界值分析法 (见例 8.4) 设计测试用例的基础上, 还可用猜错法补充一些测试用例, 例如:

- ① 出生年月为“0”。
- ② 漏输“出生年月”。
- ③ 年月次序颠倒, 例如将“197512”误输为“121975”, 等等。

除了上述 3 种方法外, 因果图法也是较常用的一种黑盒测试技术。因果图 (cause-effect graph) 是一种简化了的逻辑图。当被测程序具有多种输入条件, 程序的输出又依赖于输入条件的各种组合时, 用因果图直观地表明输入条件和输出动作之间的因果关系, 能帮助测试人员把注意力集中到与程序功能有关的那些输入组合上, 比采用等价分类法有更高明的测试效率。这种方法操作步骤比较复杂, 详情就不再介绍了。

8.5.2 白盒测试

白盒测试以程序的结构为依据, 所以又称为结构测试。早期的白盒测试把注意力放在流程图的各个判定框, 使用不同的逻辑覆盖标准来表达对程序进行测试的详尽程度。随着测试技术的发展, 人们越来越重视对程序执行路径的考察, 并且用程序图代替流程图来设计测试用例。为了区分这两种白盒测试技术, 以下把前者称为逻辑覆盖测试 (logic coverage testing), 后者称为路径测试 (path testing)。

在本节中, 将以一个升序排序的 Pascal 程序作为引例, 分别用逻辑覆盖测试法和路径法为这一程序设计测试用例。具体代码如下:

```

LABEL
99;
CONST
n=100;
VAR
a: ARRAY[1..n] of INTEGER;
i,j,k,temp:INTEGER;
BEGIN
  READLN (k);
  FOR i:=1 TO k DO READ(a[i]);
  FOR i:=2 TO k DO
    BEGIN
      IF a[i]>=a[i-1] THEN GOTO 99;
      FOR j:=i DOWNT0 2 DO
        BEGIN
          IF a[j]>= a[j-i] THEN GOTO 99;
          temp:=a[j];

```

```

    a[j]:=a[j-1];
    a[j-1]:=temp;
  END;
99:  END
    FOR i:=1 TO k DO WRITE(a[i])
    END

```

以上排序程序采用的是冒泡排序 (bubble sorting) 算法。其基本步骤是:

- ① 从一组数中取出第一个数。
- ② 取下一个数。如数已取完，则排序结束。
- ③ 如果所取数大于等于其前邻数，则重复第②步。
- ④ 如果所取数小于其前邻数，则与其前邻数交换位置。
- ⑤ 重复第④步，直至所取已无前邻数 (即已交换到当前数列的第一位置)，或大于等于其前邻数为止。
- ⑥ 返回第②步。

图 8.5 显示了该程序排序部分的流程图。由图可知:

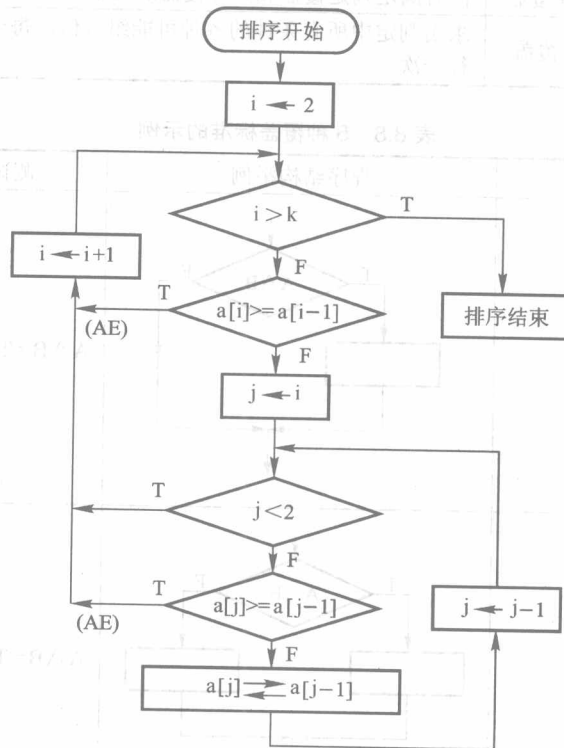


图 8.5 冒泡程序中排序部分的流程图

① 除第一个数之外，每取出一新数，便加到前面已取出的数列末尾重新排序。 k 个数重复排序 $k-1$ 次。

② 每次排序，将所取数由下向上依次与它的上一个数比较。只要它小于上一个数，就把它移到上一个数的上面。恰如水中气泡轻者上浮，从水底不断冒向水面一般。

1. 逻辑覆盖测试法

逻辑覆盖测试法通用流程图来设计测试用例，它考察的重点是图中的判定框（菱形框）。因为这些判定若不是与选择结构有关，就是与循环结构有关，因此是决定程序结构的关键成分。

按照对被测程序所作测试的有效程度，逻辑覆盖测试可由弱到强区分为5种覆盖标准，如表8.7所示。表8.8显示了实现各种覆盖标准的简单示例。

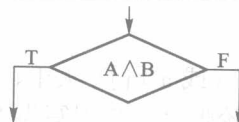
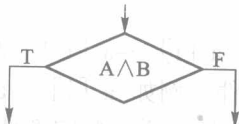
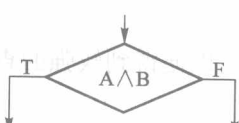
表 8.7 逻辑覆盖测试的 5 种标准

发现错误的 能力	弱	语句覆盖	每条语句至少执行一次
	↓	判定覆盖	每一判定的每个分支至少执行一次
		条件覆盖	每一判定中的每个条件，分别按“真”、“假”至少各执行一次
		判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
		强	条件组合覆盖

表 8.8 5 种覆盖标准的示例

覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		$A \wedge B = .T.$
判定覆盖		$A \wedge B = .T., A \wedge B = .F.$

续表

覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		$A=T, A=F.$ $B=T, B=F.$
判定/条件覆盖		$A \wedge B=T, A \wedge B=F.$ $A=T, A=F.$ $B=T, B=F.$
条件组合覆盖		$A=T \wedge B=T.$ $A=T \wedge B=F.$ $A=F \wedge B=T.$ $A=F \wedge B=F.$

以下结合引例，说明按照不同覆盖标准设计测试用例的方法。

(1) 对引例作逻辑覆盖测试

【例 8.5】试按表 8.7 中的不同标准，为引例中的排序程序设计测试用例。

【解】从冒泡排序程序代码和图 8.5 可知，排序程序具有双重嵌套循环结构。其内外层循环体各包含一条选择语句，用于在条件满足时提前退出循环。程序中的 4 个判断是测试时考察的重点。以下分别列出按不同覆盖标准设计的测试用例：

① 语句覆盖。稍作分析便不难看出，只要输入前大后小的两个数，程序执行时就可以遍历流程图中的所有框。因此，仅需选用一组测试数据如

$$\{a=\{8,4\},k=2\}$$

就能够实现语句覆盖。这类覆盖发现错误的能力不强，例如若将程序中的两个“>=”均误写为“=”，用上述的测试数据就不能发现。

② 判定覆盖。选用上述的测试数据，内、外层循环都是从正常的循环出口退出的。要实现判定覆盖，还需在语句覆盖的基础上，增加两个能使程序从非正常出口(在图 8.5 中用 AE 标志)退出的测试数据。例如，用以下两组数据

$$\{a=\{8,4,9\},k=3\}$$

$$\{a=\{8,4,4\},k=3\}$$

或

$$\{a=\{8,4,8,4\},k=4\}$$

则程序将在满足 $(a[i]=a[i-1])$ 或 $(a[j]=a[j-1])$ 的条件下通过非正常出口，也能实现判定覆盖。但又可能出现另一种偏向，掩盖把“>=”误写为“=”的错误，造成更加严重的测试漏洞。

③ 条件覆盖。从以上分析很容易想到，必须选取足够的测试，使多个条件中每个条件

分别按“真”、“假”出现一次，才能克服前述的缺点，进一步提高发现错误的能力。这就是条件覆盖的由来。就本例而言，如果使用测试数据

$$\{a=\{8,4,9,6\},k=4\}$$

$$\{a=\{8,4,8,4\},k=4\}$$

就能对程序实现条件覆盖。此时 $a[i]$ (或 $a[j]$) 大于、等于或小于 $a[i-1]$ (或 $a[j-1]$) 的 3 种情况将分别至少出现一次，无论把“ \geq ”误写为“ $>$ ”或“ $=$ ”，都可用这两组数据检查出来。

④ 其他覆盖。本例中的两个条件 $a[i] \geq a[i-1]$ 及 $a[j] \geq a[j-1]$ ，其组成条件都不是互相独立的。如果其中有一个条件 (例如 $a[i] > a[i-1]$) 为真，则另一个条件 (如 $a[i] = a[i-1]$) 必然为假。所以就本例来说，判定条件覆盖及条件组合覆盖都没有实际意义，可以不必讨论。

由此可见，本例宜选择条件覆盖，以便得到较强的查错能力。测试数据可选择

$$\{a=\{8,4,9,6\},k=4\}$$

$$\{a=\{8,4,8,4\},k=4\}$$

或合成一组

$$\{a=\{8,4,8,4,9,6\},k=6\}$$

(2) 关于覆盖标准的讨论

在表 8.7 的 5 种覆盖中，语句覆盖发现错误的能力最弱，一般不单独采用。判定覆盖与条件覆盖的差别在于：前者把判定看成一个整体，后者则着眼于其中的一个条件。当一个判定只含一个条件时，判定覆盖也就是条件覆盖。但如果一个判定含有一个以上的条件 (称其为复合条件)，采用判定覆盖有可能出现例如下述的漏洞，即判定中有些条件得到测试，另一些条件却被忽略，从而掩盖程序的错误。条件覆盖要求对每一条件进行单独的检查，一般说来它的查错能力比判定覆盖更强，但也并不尽然。在图 8.6 中，如果由条件 A、B 的 4 种逻辑值组成内容为 {A 真, B 假} 和 {A 假, B 真} 的两组测试数据，则无论判定包含的条件是“A and B”或“A or B”，都只覆盖一个分支，另一个分支未被覆盖。把判定覆盖和条件覆盖的要求汇集于一身的判定/条件覆盖，正是为了弥补条件覆盖的这一不足之处。

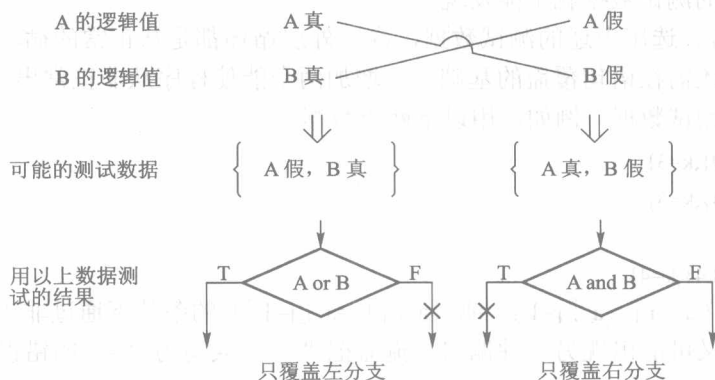


图 8.6 只能覆盖一个分支的条件覆盖一例

条件组合覆盖在 5 种覆盖中发现错误的能力最强。从表 8.8 可知, 凡满足条件组合覆盖的测试数据, 也必然满足其余 4 种覆盖标准。

2. 路径测试法

逻辑覆盖测试引导人们把注意力集中在程序的各个判定部分, 抓住了结构测试的重点。但是另一方面, 它却忽略了另一个对测试也有重要影响的方面——程序的执行路径。随着程序结构复杂性的增长和测试技术的发展, 人们逐渐认识到这种忽略所带来的缺陷。于是, 着眼于程序执行路径的测试方法便应运而生, 这就是路径测试。

路径测试离不开程序图。下面先对程序图 (program graph) 作一简单介绍。

(1) 程序图

程序图实际上是一种简化了的流程图。在路径测试中, 它是用来考察测试路径的有用工具。图 8.7 显示了分别用流程图和程序图来表示的基本控制结构, 流程图中各种不同形状的框, 在程序图中都被简化为用圆圈表示的一个个结点。由于程序图保留了控制流的全部轨迹, 舍弃了各框的细节, 因而画面简洁, 路径清楚, 用它来验证各种测试数据对程序执行路径的覆盖情况, 比流程图更加方便。

这里还有两点需要说明:

- ① 顺序执行的多个结点, 在程序图中可以合并画成一个结点, 如图 8.8 所示。

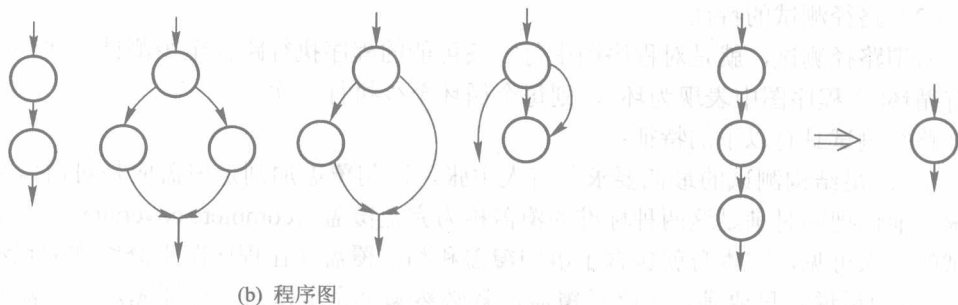
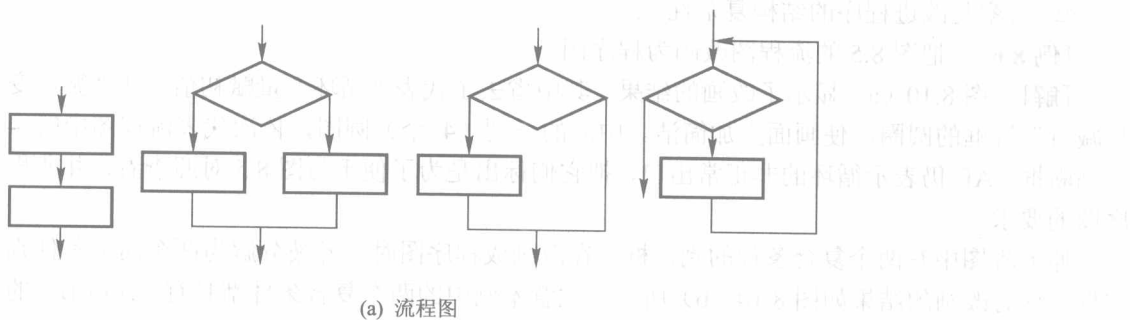


图 8.7 程序图与流程图的对照图形

图 8.8 合并结点的程序图

② 含有复合条件的判定框，应先将其分解成几个简单条件判定框，然后再画程序图，如图 8.9 所示。

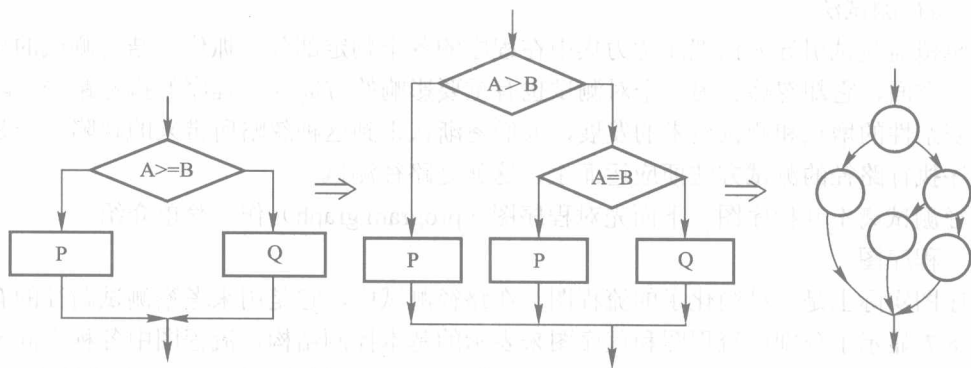


图 8.9 分解为简单条件结点后的程序图

以上的画法表明，程序图所关心的是程序中的判定框，而不是顺序执行部分的细节。这与下述用途是一致的，即：

- ① 安排与验证程序的路径测试。
- ② 考察与改进程序的结构复杂性。

【例 8.6】把图 8.5 的流程图改画为程序图。

【解】图 8.10 (a) 显示了改画的结果。其中省去了代表“循环变量赋初值”和“循环变量减 1”等框的圆圈，使画面更加简洁。中间的一列 (4 个) 圆圈，依次代表流程图中的 4 个判断框。AE 仍表示循环的非正常出口，把它们标出是为了便于与图 8.5 对照查看，并非程序图的要求。

原流程图中有两个复合条件的判定框，在改画成程序图时，各被分解为两个简单条件判定框。这时改画的结果如图 8.10 (b) 所示。注意本例中的两个复合条件都具有“ $A \text{ or } B$ ”的形式。如果复合条件具有“ $A \text{ and } B$ ”的形式，分解画法也应相应地改变。

(2) 路径测试的特征

所谓路径测试，就是对程序图中每一条可能的程序执行路径至少测试一次。如果程序中含有循环(在程序图中表现为环)，则每个循环至少执行一次。

路径测试具有以下特征：

① 满足结构测试的最低要求。有人主张，语句覆盖加判定覆盖应是对白盒测试的最低要求。他们把同时满足这两种标准的覆盖称为完全覆盖 (complete coverage)。从上述对路径测试的要求可见，它本身就包含了语句覆盖和判定覆盖 (在程序图上分别称为点覆盖与边覆盖)。换句话说，只要满足了路径覆盖，就必然满足完全覆盖，也即满足了白盒测试的最低要求。

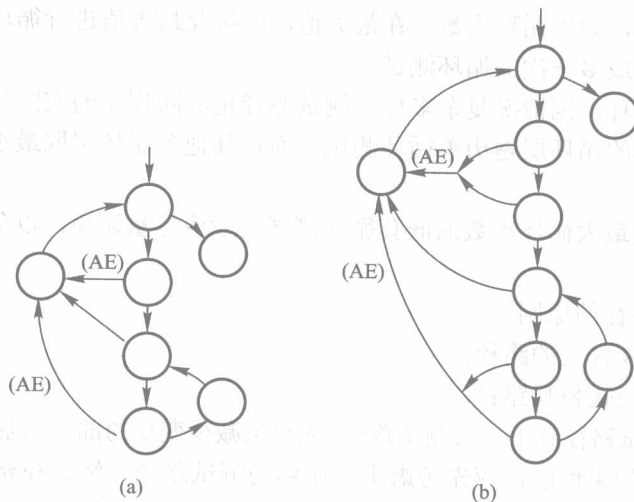
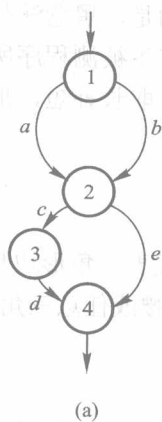


图 8.10 根据图 8.5 流程图导出的程序图

[例 8.7] 为图 8.11 (a) 所代表的被测程序设计测试路径。

[解] 这是一个不含循环的程序。从 1~4, 程序共有 4 条执行路径。图 8.11 (b) 显示了满足 3 种不同覆盖要求所需选择的测试路径。显然, 路径覆盖包含了点覆盖与边覆盖。



测试路径	覆盖结点/边	覆盖标准
<i>acd</i>	①, ②, ③, ④	点覆盖
<i>acd, be</i>	<i>a, b, c, d, e</i>	边覆盖
<i>acd, be</i> <i>ae, bcd</i>	①, ②, ③, ④/ <i>a, b, c, d, e</i>	路径覆盖

图 8.11 为一个被测程序测试路径

② 有利于安排循环测试。循环测试是路径测试的一个重要部分。从程序的执行路径而不是单纯从判定的角度来测试程序, 有利于对程序中包含的循环结构进行更有效的测试。一般的说, 对单循环结构的路径测试可包括:

- 零次循环, 即不执行循环体, 直接从循环入口跳到出口。
- 一次循环, 循环体仅执行一次, 主要检查在循环初始化中可能存在的错误。
- 典型次数的循环。

- 最大值次循环，如果循环次数存在最大值，应按此最大值进行循环，需要时还可增加比最大值次数少一次或多一次的循环测试。

对于多重嵌套循环，因情况复杂多样，测试路径也应随程序的实际需要来选择。一般的说，可以对某一指定的循环层遍历单循环测试，而在其他各循环层取最小或典型次数进行循环测试。

所有循环层均取最大循环次数的嵌套循环测试一般应尽量避免，以免过多地增加测试执行时间。

(3) 选择测试路径的原则

- ① 选择具有功能含义的路径。
- ② 尽量用短路径代替长路径。
- ③ 从上一条测试路径到下一条测试路径，应尽量减少变动的部分(包括变动的边和结点)。
- ④ 由简入繁，如果可能，应先考虑不含循环的测试路径，然后补充对循环的测试。
- ⑤ 除非不得已(如为了要覆盖某条边)，不要选取没有明显功能含义的复杂路径。

8.6 测试用例设计

上节讨论了软件测试的黑盒与白盒技术。本小节以两个简单程序为例，说明怎样从被测程序的实际出发，来确定测试策略和选择测试用例。需要指出的是，黑盒法与白盒法均包括多种技术，各有所长。但若单独采用其中的一种技术，都不能产生被测程序所需要的全部测试用例。因此在实际的程序测试中，总是将各种技术结合起来，取长补短，形成综合的测试策略。

8.6.1 黑盒测试用例设计

假设现有以下的三角形分类程序。该程序的功能是，读入代表三角形边长的3个整数，判断它们能否组成三角形。如果能够，则输出三角形是等边、等腰或任意三角形的分类信息。图8.12显示了该程序的流程图和程序图。

[例8.8] 为以上的三角形分类程序设计一组测试用例。

[解] 第一步：确定测试策略。在本例中，对被测程序的功能有明确的要求，即：判断能否组成三角形；识别等边三角形；识别等腰三角形；识别任意三角形。因此可首先用黑盒法设计测试用例，然后用白盒法验证其完整性，必要时再进行补充。

第二步：根据本例的实际情况，在黑盒法中首先可用等价分类法划分输入的等价类，然后用边值分析法和猜错法作补充。图8.13列出了用黑盒法分析程序的输入后得出的对测试用例的具体要求。

第三步：编制一组初步的测试用例，如表8.9所示。

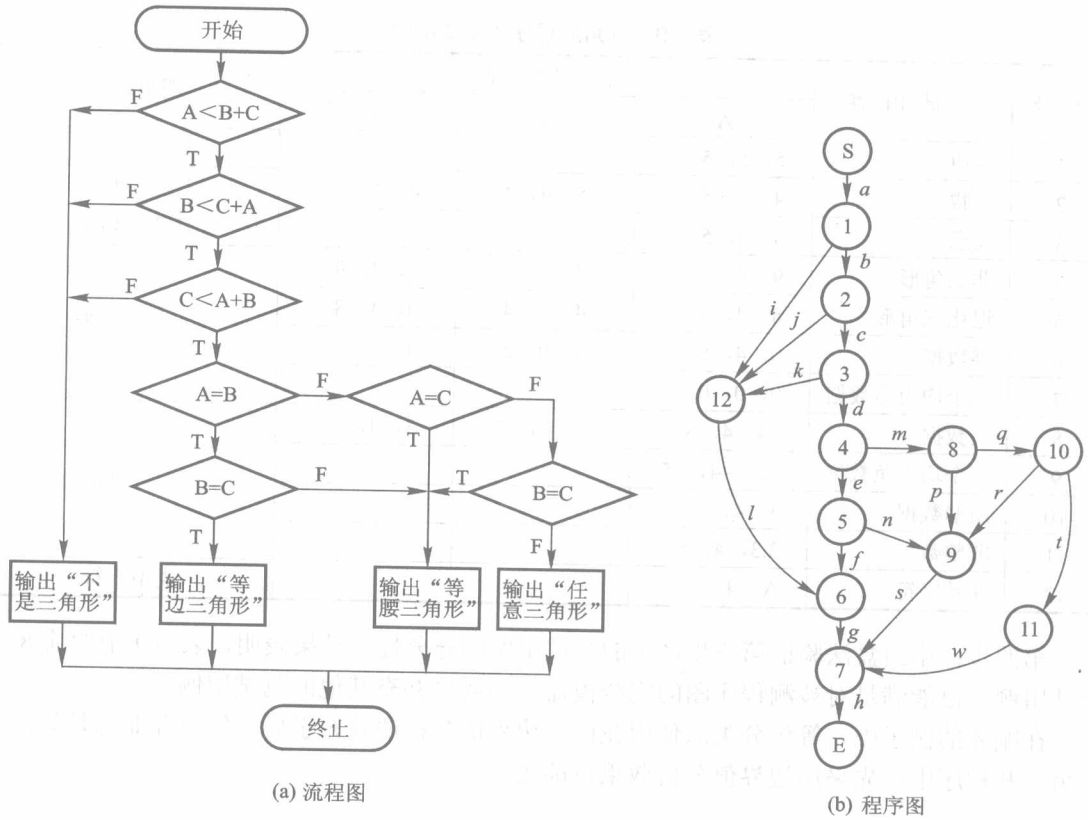


图 8.12 三角形程序的流程图与程序图

等价分类法	
有效等价类	无效等价类
输入 3 个正整数:	含有零数据⑨
3 数相等①	含有负整数⑩
3 数中有 2 数相等	少于 3 个整数⑪
	含有非整数⑫
	含有非数字符⑬
3 数均不相等⑤	
2 数之和不大于第 3 数	
	最大数为 A⑥
	最大数为 B⑦
	最大数为 C⑧
边值法: 2 数之和等于第 3 数	⑭
猜错法: 输入 3 个零	⑮
输入 3 个负数	⑯

图 8.13 用黑盒法分析程序的输入

表 8.9 三角形程序的测试用例

序号	测试内容	测试数据			期望结果
		A	B	C	
1	等边	5, 5, 5			等边三角形
2	等腰	4, 4, 5	5, 4, 4	4, 5, 4	等腰三角形
3	任意	3, 4, 5			任意三角形
4	非三角形	9, 4, 4	4, 9, 4	4, 4, 9	不是三角形
5	退化三角形	8, 4, 4	4, 8, 4	4, 4, 8	
6	零数据	0, 4, 5	4, 0, 5	4, 5, 0	
7	三个均为零数据	0, 0, 0			
8	负数据	-3, 4, 5	3, -4, 5	3, 4, -5	运行出错
9	三个均为负数据	-3, -4, -5			
10	遗漏数据	3, 4,			
11	非整数	3.3, 4, 5			
12	非数字符	A, 4, 5			类型不符

第四步：用白盒法验证第三步产生的测试用例的充分性。结果表明，表 8.9 中的前 8 个测试用例，已能满足对被测程序图的完全覆盖，不需再补充其他的测试用例。

在刚才的例子中，等价分类法使用在前，边界值分析法补充于后。但也并非总是如此。在另一些程序中，先采用边界值分析效果可能更好。

8.6.2 白盒测试用例设计

以下程序段一个用 FORTRAN 77 编写的学生成绩查询程序，最多可以存放 100 个学生的成绩，按学号排序。如果输入的学号在这批学生内，就输出该学生的成绩，否则输出学号 ×× 未找到信息。

```

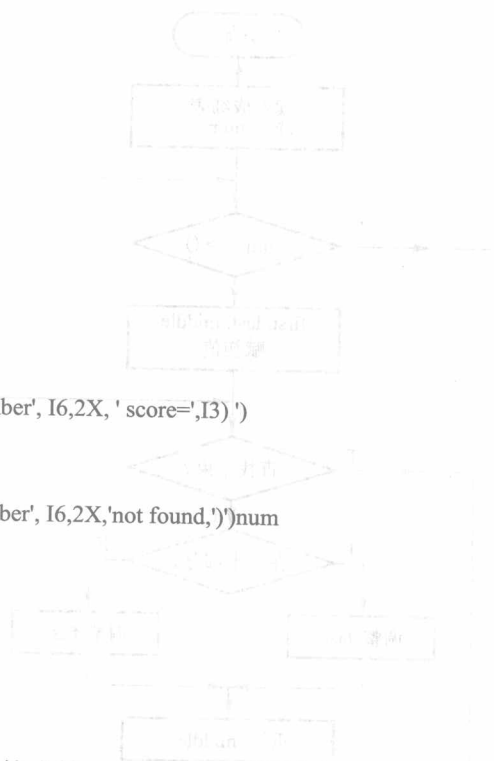
INTEGER first, last, middle,
studntno, num
DIMENSION studntno(100), score(100)
READ(*,*) k
READ(*,*) (studntno(i), score(i), i=1, k)
READ(*,*) num
DO 30, WHILE (num.NE.O)
    first=1
    last=k
    middle=(first+last)/2
    IF (num.EQ.studntno(middle).OR.
First.EQ.last) GOTO 20

```

```

10  IF (num.LT.studntno(middle))
      THEN
          last=middle
      ELSE
          first=middle+1
      ENDIF
      middle=(first+last)/2
      GOTO 10
20  CONTINUE
    IF (num.EQ.studntno(middle))
      THEN
          WRITE (*,(1X,' student number', I6,2X, ' score=',I3)')
          num,score(middle)
      ELSE
          WRITE (*,(1X,' student number', I6,2X,'not found,))num
      ENDIF
      READ(*,*)num
30  CONTINUE
END

```



阅读这段程序可知：

- ① 程序由内外两层循环嵌套而成。
- ② 外层循环保证一次可查找多个学生的成绩。一旦输入的学号为“0”，即结束查找。
- ③ 内层循环执行“折半查找”算法。它首先在已按学号排序的成绩表中找出中间项，如果要查的就是这个学号，查找就告结束。否则便判断所查的学号是在成绩表的上半部分或下半部分，保留所要的一半，舍弃无用的一半。下一轮查找，仍依照刚才的做法先找出所保留的一半的中间项，然后重复上述的判断与操作。如此继续下去，直到找出的中间项与要找的学号相符，或者只剩下的一项仍不等于要找的学号，查找宣告失败为止。

图 8.14 显示了该程序的流程图和程序图。注意在流程图中带复合条件的判定框（即 $\text{num} < 0$ ），在程序图中被分解为两个带简单条件的结点（即④与⑤）。

【例 8.9】 试为上述学生成绩查询程序设计测试用例。

【解】 第一步：确定测试策略。折半查找是实现查找的一种算法，还可以使用其他算法。就本例而言，主要应测试程序在各种典型情况下能否有效地实现查找，并选择正确的执行路径。因此，路径覆盖法显然是比较合理的测试策略。

第二步：令成绩表的长度为 1 项或小于 100 的任意项（例如 5 项）。在成绩表的不同位置上每次查 1 ± 1 个学生，考察其执行情况：

- ① 成绩表只有 1 项，查 0 个学生。
- ② 成绩表只有 1 项，查 1 个学生（是成绩表中的学生）。

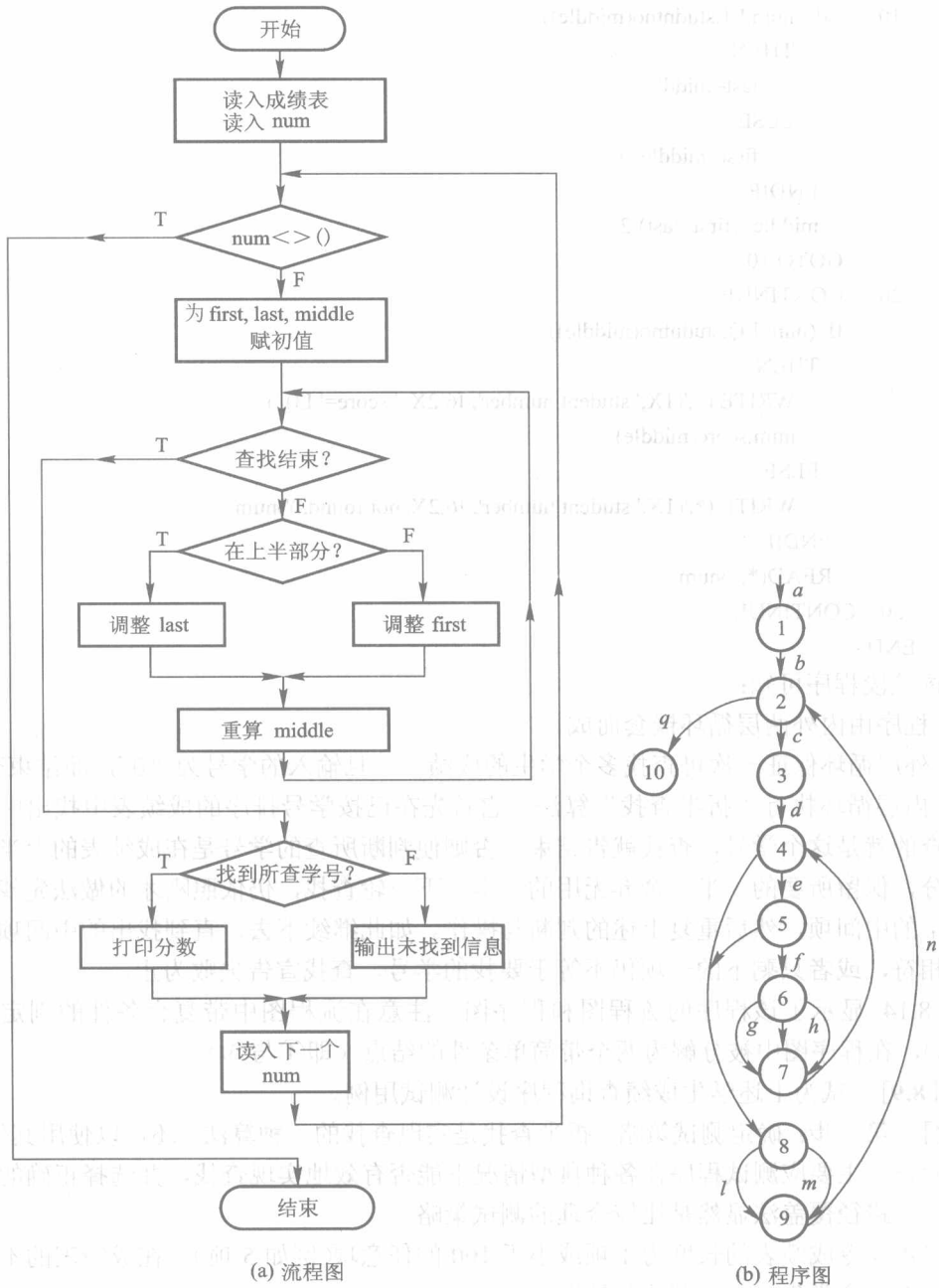


图 8.14 成绩查询程序的流程图与程序图

③ 成绩表只有 1 项，查 2 个学生（都不是成绩表中的学生）。

④ 成绩表有 5 项，查 1 个学生（正好是表的中间项）。

⑤ 成绩表有 5 项，查 1 个学生（在表的下半部分）。

⑥ 成绩表有 5 项，查 1 个学生（在表的下半部分）。

⑦ 成绩表有 5 项，查 1 个学生（不是表中的学生）。

测试数据如表 8.10 所示。由表可知，以上测试数据已达到对程序图的完全覆盖。

表 8.10 满足完全覆盖要求的测试数据

测试数据	覆盖的结点										覆盖的边															
	①	②	③	④	⑤	⑥	⑦	⑧	⑨	⑩	a	b	c	d	e	f	g	h	i	j	k	l	m	n	q	
k=1, studntno={100}																										
num={0}	√	√								√	√	√														√
num={100,0}	√	√	√	√					√	√	√	√	√	√						√		√		√	√	√
num={90,110,0}	√	√	√	√	√				√	√	√	√	√	√								√		√	√	√
k=5, studntno={2,4,6,8,10}																										
num={6,0}	√	√	√	√					√	√	√	√	√	√						√		√		√	√	
num={2,0}	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√		√	√	√		√		√	
num={8,0}	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√		√	√	√		√		√	√	
num={9,0}	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√		√	√	√		√		√	√	

说明：测试数据中略去了 score 数组的数据。本例中它们可以取任意值，不会影响测试结果。

第三步：成绩表长度为最大值（100 项）。连续查找学号为最小值、比最小值小 1、最大值、比最大值大 1、中间值以及上半、下半表中典型值的 7 个学生。具体测试数据可以是：

k=100, studntno={101,102,⋯,199,200}

score={60×20,70×20,80×20,90×20,100×20}, 其中, 60×20 表示 20 个 60 分, 其余表示含义与此类似, 共计 100 个学生。

num={100,101,200,201,150,130,170}

从以上 3 步可见：

① 虽然是路径测试，同时也测试了程序的功能。例如：

- 只查一个学生，连续查多个学生，或一个学生不查。
- 成绩表长度为 1 项，为 100 项，或为任意项（例如 5 项）。

② 虽然是按白盒法设计的测试用例，也运用了黑盒法中的等价分类与取边缘值等思想。

从前面的两个例子可知，设计测试用例的过程，实际上也是对黑盒测试技术与白盒测试技术综合运用过程。读者不可能期望只靠几条简单的规则就能解决各种实际问题，恰恰相反，有些做法“只可意会，不能言传”，真正需要的倒是举一反三，灵活地运用在黑盒和白盒测试中所蕴涵的原则与思想。

8.7 多模块程序的测试策略

实际的应用程序大都是多模块程序。像前面举例的一些程序，可能仅相当于大程序的一个模块。所以在软件工程的术语中，软件测试主要指多模块程序的测试。

前几节讨论的测试与纠错技术，对大、小程序是普遍适用的。但是，就测试的内容与实施步骤来说，多模块程序要比单模块小程序复杂得多。这种复杂性的主要表现是测试的层次性。

8.7.1 测试的层次性

按照软件工程的观点，多模块程序的测试共包括4个层次，如图8.15所示。图8.16表明了层次测试的信息流程。

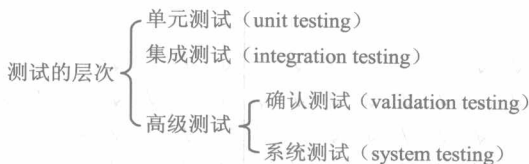


图 8.15 多模块程序的测试分层

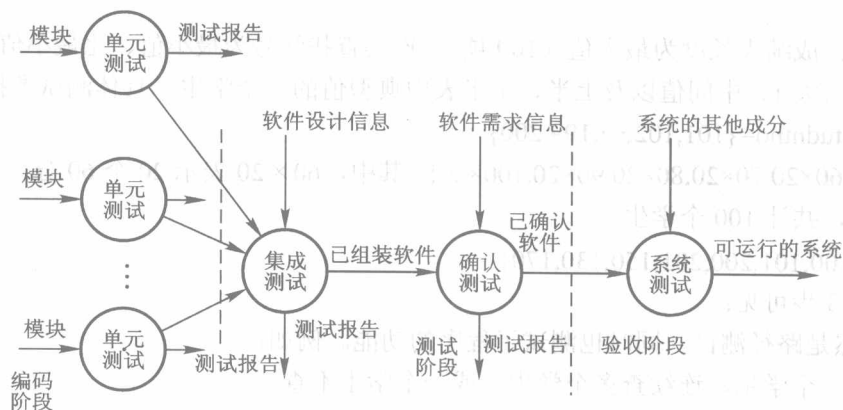


图 8.16 层次测试的信息流程

第一层为单元测试，应该在编码阶段完成。单元一般以模块或子程序为单位，故又称模块测试（module testing）。

测试阶段应完成集成测试与确认测试两个层次的测试。这一阶段的任务，是把通过了单

元测试的模块逐步组装起来，通过测试与纠错，最终得到一个满足需求的目标软件。其中，集成测试是一个逐步组装的过程。它从一个单元开始，逐一地添加新的单元，边添边测边纠错，直至最终将所有单元集成为一个系统，所以也称为集成测试。确认测试是对整个程序的测试，用于确认组装完毕的程序能满足用户的全部需求。

系统测试在验收阶段进行。目的是检查当被测程序安装到系统上以后，与系统的其他软、硬件能否协调工作，并完成说明书规定的任务。

确认测试和系统测试都是对整个程序的测试，二者都属于高级测试。

以上说的是大层次。有些层次内部还包含一些小的层次。例如单元测试就包括编译、静态分析和动态测试等 3 个层次。

8.7.2 单元测试

单元测试是层次测试的第一步，也是整个测试的基础。据估计，单元测试发现的错误，约占程序总错误数的 65%，接近 2/3。

对多模块程序的测试从单元开始，至少有以下好处：减少测试的复杂性；易于确定错误的位置（不超过一个模块）；多个单元可以并行测试，缩短测试周期。

1. 目的与任务

目的：通过对象模块的静态分析与动态测试，使其代码达到模块说明书的需求。

任务：

- ① 对模块代码进行编译，发现并纠正其语法错误。
- ② 进行静态分析，验证模块结构及其内部调用序列是否正确。
- ③ 确定模块的测试策略，并据此设计一组测试用例和必要的测试软件。
- ④ 用选定的测试用例对模块进行测试，直至满足测试终止标准为止。
- ⑤ 静态分析与动态测试的重点，均应放在模块内部的重要执行路径、出错处理路径和局部数据结构，也要重视模块的对外接口。
- ⑥ 编制单元测试报告。

2. 实施步骤

按照上述单元测试的任务，单元测试的实施步骤可以简述为：

编译 ——> 静态分析器检查 ——> 代码评审 ——> 动态测试

从测试的角度来说，编译或其他语言翻译程序也是一种静态分析工具，其检查对象是代码中的语法错误。接下来的两步仍然是静态分析，不过检查对象已从语法错误转换为以结构性错误为主的其他错误。而且，第二步是使用专用工具（静态分析器）来进行分析的，第三步则主要依靠人工。从本质上说，前 3 步都属于代码的静态分析，即不依靠执行被测程序来发现代码中的错误。

在大型程序的测试中，使用静态分析器可以节省许多人力。20 世纪 70 年代以来，已有许多静态分析器投入使用，并获得成功。但直到现在，代码评审仍然以人工方式为主，并未

被分析器所取代。这是因为多数分析工具只有有限的功能，且依赖于特定的语言（即一种工具仅适用于一种语言编写的程序），不易普遍推广。

动态测试是单元测试的最后步骤，重点是发现单元的功能性错误。根据程序的实际情况，可采用白盒测试或黑盒测试方法。读者可以参阅前面的例子，这里就不多说了。

单元测试一般安排在编码阶段完成。实施单元测试的人员可以有两种选择：一是由编码者负责实施，其优点是编码者对单元的结构与功能最熟悉，容易设计出符合需要的测试用例；二是由单元的设计者而不是编码者（如果不是同一个人）来设计测试用例，他同样对单元比较熟悉，又可以避免编码者怕暴露自己程序弊病的心理。在选择后一种做法时，应提倡编码者在单元提交正式测试前先做一次非正式的测试。

3. 代码评审

如前所述，代码评审在编译之后、动态测试之前进行。如果有静态分析工具，则应在工具分析之后进行。

评审的目的，在于发现程序在结构、功能与编码风格方面存在的问题和错误。组织良好的代码评审，可发现 30%~70% 的设计和编码错误，从而加快单元测试的进程，提高测试的质量。众所周知，动态测试仅能发现错误的症状，而代码评审一旦发现错误，就同时确定了错误的位置。所以，许多学者和公司都把代码评审看作测试中不可缺少的环节，将其视为动态测试之前的必要准备。

评审有两类组织形式。一类是办公桌检查（desk check），由编程序者自己审查自己的代码，仅适用于规模很小的程序。另一类以小组会的方式进行，又可分走查（walk through）和代码会审（code inspection）两种，适用于各种规模的程序，具体做法不再赘述，有兴趣者可参看有关书籍。

4. 测试软件

单元不是独立的程序。在多模块程序中，每一模块都可能调用其他模块或者被其他模块所调用。所以在单元测试时，需要为被测模块编制若干测试软件，给它的上级模块或下级模块做替身。代替上级模块的称为测试驱动模块（test driver），代替下级模块的称为测试桩模块（test stub）。

需要指出，替身模块应该是真实模块的简化，仅需模拟与被测模块直接相关的一部分功能。它们在全局层次测试结束后即完成了历史使命，一般不移交给用户。

8.7.3 集成测试

通过了单元测试的模块，要按照一定的策略组装为完整的程序。在组装过程中进行的测试称为集成测试或组装测试。

为什么所有的模块都经过了单元测试，组装中还会出现问题呢？这是因为：

① 单元测试中使用了测试软件。它们是真实模块的简化，与它们所代替的模块并不完全等效。因此，单元测试本身可能有不充分的地方，存在缺陷。

② 多模块程序各模块之间，可能有比较复杂的接口，稍有疏忽就易出错。例如有的数据在跨越接口时会不慎丢失，有些全局性数据在引用中可能出问题，等等。

③ 有些在单个模块中可以允许的误差，组装后的积累误差可能达到不能容忍的地步；或者模块的分功能似乎正常，组装后也可能产生了预期的综合功能。由此可见，在软件的层次测试中，集成测试不仅必要，而且占有重要的地位。

1. 目的与任务

目的：将经过单元测试的模块逐步组装成具有良好一致性的完整的程序。

任务：

① 制定集成测试实施策略。根据程序的结构，可以选择自顶向下或由底向上或二者混合的两头逼近策略（详见下文）。

② 确定集成测试的实施步骤，设计测试用例。二者的选择，应有利于揭露在接口关系、访问全局性数据（包括公用文件与数据结构）、模块调用序列和出错处理等方面存在的隐患。

③ 进行测试，即在已通过单元测试的基础上，逐一地添加模块。每并入一个模块，除进行新的测试项目外，还需重复先前已经进行过的测试，后者也称为回归测试（regression testing）。回归测试可用于防止因软件组装改变而导致新的不协调，出现不可预料的错误。它通常是原来已经进行过的测试的一个子集。当在软件维护中修改了某些模块后，也需要进行回归测试。

2. 策略与步骤

集成测试的策略，可以分为自顶向下、由底向上以及从两头逼近的混合方式等3种。现在以图 8.17 的多模块程序为例，分别说明如下。

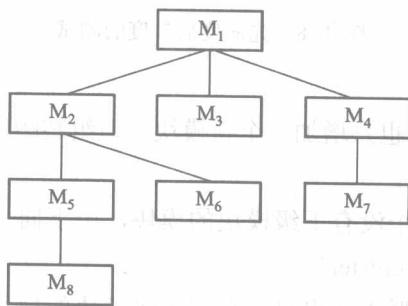


图 8.17 一个简单的多模块程序

(1) 自顶向下测试

从顶模块开始，沿被测程序的结构图逐步向下测试。按照移动路线的差异，又可区分为两种不同的实施步骤。

① 先广后深实施步骤。这时模块的组装顺序是：

$$M_1 - M_2 - M_3 - M_4 - M_5 - M_6 - M_7 - M_8$$

② 先深后广实施步骤。组装顺序为：

$M_1—M_2—M_5—M_8—M_6—M_3—M_4—M_7$

自顶向下测试要使用桩模块。图 8.18 显示了采取先深后广步骤时所需桩模块的配置情况。其中桩模块都是下标相同的真实模块的替身，例如， S_2 是 M_2 的替身， S_3 是 M_3 的替身。

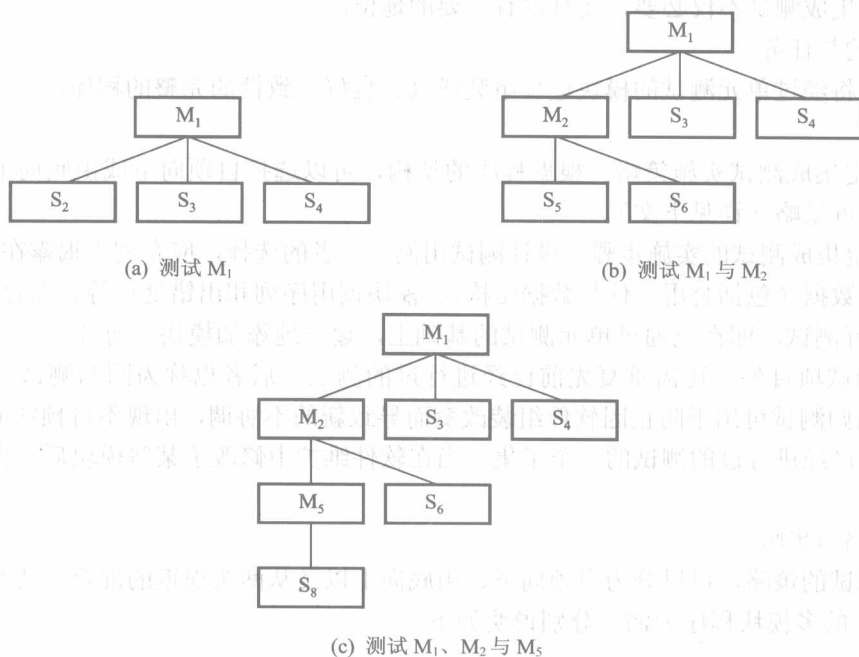


图 8.18 先深度后广度的测试

(2) 由底向上测试

它与自顶向下一样，每次也只增加一个新模块。但组装顺序恰好相反，采取由下由上的路线。其典型步骤可以描述为：

① 从下层模块中找出一个没有下级模块的模块，由下向上地逐步添加新模块，组成程序中的一个子系统或模块群（cluster）。

② 从另一子系统或模块群中选出另一个无下级模块的模块，仿照前一步组成又一个子系统；

③ 重复上一步，直至得出所有的子系统，把它们组装为完整的程序。

仍以图 8.17 的程序为例，按以上的步骤，其模块组装顺序可能是：

$M_8—M_5—M_6—M_2$

$M_7—M_4—M_3—M_1$

全部模块（即合并以上两个群）

(3) 混合方式测试 (sandwich testing)

它是上述两种测试方式的结合, 其具体步骤为:

① 对上层模块采取自顶向下测试。

② 对关键模块或子系统采取由底向上测试。

这种从两头向中间逼近的方法, 可以兼有自顶向下和由底向上策略的优点, 所以应用较广。

为了保证测试的质量, 集成测试应由独立于开发人员的测试小组负责实施。

以上 3 种测试策略都是从一个模块开始, 测一次添一个模块, 使组装程序像滚雪球一样越滚越大, 所以统称为渐增式 (incremental) 测试。与此对照, 早期程序规模较小时, 集成测试也使用过非渐增式 (non-incremental) 测试的策略, 即一次就集中程序的所有模块 (当然都已经过单元测试), 对程序作集成测试。这种策略的最大缺点是“眉毛胡子一把抓”, 不易定位产生错误的模块。随着软件规模的增长, 现在除了极小的程序外, 几乎都采用渐增式测试策略。

3. 几种策略的比较

由顶向下测试的优点是, 能较早显示整个程序的轮廓, 向用户展示程序的概貌, 取得用户的理解和支持。其主要缺点是, 当测试上层模块时因使用桩模块较多, 很难模拟出真实模块的全部功能, 使部分测试内容被迫推迟, 只能等到换上真实模块后再补充测试。在低层次上的替身模块多, 也会增加设计测试用例的困难。由底向上测试从下层模块开始, 设计测试用例比较容易, 但是在测试的早期不能显示出程序的轮廓。混合测试的优点, 正在于扬长补短, 综合了以上两种策略的长处。例如, 对关键模块采取由底向上测试, 就可能把输入输出模块提前组装进程序, 使设计测试用例变得较为容易; 或者使具有重要功能的模块早些与有关的模块相连, 以便及早暴露可能存在的问题。除关键模块及少数与之相关的模块外, 对其余模块尤其是上层模块仍采取自顶向下的测试方法, 以便尽早地显示程序的总体轮廓。

8.7.4 确认测试

1. 有效性测试 (黑盒测试) 和配置复审

确认测试继集成测试之后进行, 其目的在于确认组装完毕的程序是否满足软件需求规格说明书 (SRS) 的要求。典型的确认测试通常包括有效性测试 (黑盒测试) 和配置复审 (configuration review) 等内容。在 SRS 中一般都有标题为“有效性标准”的一节, 其中的内容就是确认测试的依据。配置复审则用于查明程序的文档是否配齐, 以及文档内容的一致性。

确认测试是由软件开发单位组织实施的最后一项开发活动。测试结束后, 软件就要交付验收了。因此, 开发单位必须十分重视这项工作。同集成测试一样, 确认测试也应由独立的测试小组负责实施。

2. 验收测试 (acceptance testing)

如果软件是给一个客户开发的, 需要进行一系列的验收测试来保证客户满足所有的需

求。验收测试主要由用户而不是开发者来进行，可以进行几个星期或者几个月，因而可发现随时间的积累而产生的错误。

3. α 与 β 测试

如果一个软件是为很多用户开发的（例如 Office 软件），让每一个用户都进行正式的验收测试显然是不切实际的。这时可使用 α 测试与 β 测试，来发现那些通常只有最终用户才能发现的错误。

α 测试是在一个受控的环境下，由用户在开发者的指导下进行的测试，由开发者负责记录错误和使用中出现的问题。

β 测试则不同于 α 测试，是由最终用户在自己的场所进行的，开发者通常不会在场，也不能控制应用的环境。所有在 β 测试中遇到的（真正的或是想象中的）问题均由用户记录，并定期把它们报告给开发者，开发者在接到 β 测试的问题报告之后，对系统进行最后的修改，然后就开始准备向所有的用户发布最终的软件产品。

8.7.5 系统测试

系统测试是在更大范围内进行的测试。除被测程序外，系统还可能包括硬件和原有的其他软件。测试的目的，是检查把确认测试合格的软件安装到系统中以后，能否与系统的其余部分协调运行，并且实现 SRS 的要求。

系统测试是验收工作的一部分，应由用户单位组织实施。软件开发单位应该为系统测试创造良好的条件，负责回答和解决测试中可能发现的一切质量问题。

8.7.6 终止测试的标准

黑盒测试和白盒测试都是选择测试，不可能彻底发现程序的所有错误。既然如此，测试该进行到什么程度才算完成任务呢？显然，测试过少，程序的遗留错误较多，将降低其可靠性；但过量的测试，也会不必要地增大软件成本。在这一小节，将向读者介绍两种实用的测试终止标准（test completion criteria）及其方法。

1. 规定测试策略和应达目标

进行白盒测试时一般可规定以完全覆盖为标准，即语句覆盖率和分支覆盖率必须分别达到 100%，满足了这些条件就可终止测试。黑盒测试时，可结合程序的实际情况选择一或多种方法（例如边值法或等价分类法）来设计测试用例。当把所有测试用例全部用完后测试便可终止。

2. 规定至少要查出的错误数量

如果已经积累了较丰富的测试经验，可以把查出预定数量的错误作为某类应用程序的测试终止标准。例如，假定被测程序是一个约有 10 000 行的管理信息系统，根据以往的经验，这么多行的程序约有 300 个设计错误及 200 个代码和结构错误。若预定的目标是消除 95% 的设计错误和 98% 的编码与结构错误，则可以规定，查出 285 个设计错误和 196 个编码与结构

错误后就可终止测试。

8.8 面向对象系统的测试

曾经有人认为,随着 OO 技术走向成熟,复用的软件会不断增加,OO 软件系统的测试工作量也会比传统软件的测试工作量逐渐减轻。可惜这并非事实。实践表明,对复用的软件仍需要重新仔细地测试。加上 OO 开发的下列特点,OO 软件系统将需要比传统软件系统更多而不是更少的测试。

首先,OO 软件中的类/对象在 OOA 阶段就开始定义了。如果在某一个类中多定义了一个无关的属性,该属性又多定义了两个操作,则在 OOD 与随后的 OOP 中均将导致多余的代码,从而增加测试的工作量。所以有人认为,OO 测试应扩大到包括对 OOA 和 OOD 模型的复审,以便及早发现错误。

其次,OO 软件是基于类/对象的,而传统软件则基于模块。这一差异,对软件的测试策略与测试用例设计均带来不小的改变,也将增加测试的复杂性。本节仅就 OO 软件的测试策略和测试用例设计作简要介绍。

8.8.1 OO 软件的测试策略

从战略上说,OO 软件测试和传统软件测试一样,也是从单元测试开始,然后经集成测试,最后进入确认与系统测试的。但是在具体做法上,OO 软件的测试策略与传统测试策略有许多不同。现分述如下。

1. OO 软件的单元测试

对 OO 软件的类测试等价于传统软件的单元测试。两者区别在于,在传统的单元测试中,单元指的是程序的函数、过程或完成某一特定功能的程序块,而对于 OO 软件而言,单元是封装的类和对象。因此,传统软件的单元测试往往关注模块的算法细节和模块接口间流动的数据,而 OO 软件的类测试是由封装在类中的操作和类的状态行为所驱动的。它并不是孤立地测试单个操作,而是把所有操作都看成是类的一部分,全面地测试类和对象所封装的属性和操纵这些属性的操作的整体。具体地说,在 OO 的单元测试中不仅要发现类的所有操作中存在的问题,还要考察一个类与其他的类协同工作时可能出现的错误。

面向对象编程的特性,使得对成员函数的测试不完全等同于传统的函数或过程测试。尤其是继承特性和多态特性,使子类继承或重载的父类成员函数出现了传统测试中未遇到的问题。

2. OO 软件的集成测试

传统的集成测试是采用自顶向下或由底向上或二者混合的两头逼近策略,通过用渐增方式集成功能模块进行的测试。但面向对象程序没有层次的控制结构,相互调用的功能也是分散在不同类中,类通过消息相互作用来申请或提供服务,所以渐增式的集成测试方法不再适

用。此外，面向对象程序具有动态特性，程序的控制流往往无法确定，因此只能进行基于黑盒方法的集成测试。

OO 的集成测试主要关注系统的结构和内部的相互作用，以便发现仅当各类相互作用时才会产生的错误。有两种 OO 软件的集成测试策略：基于线程的测试（thread-based testing）和基于使用的测试（use-based testing）。基于线程的测试用于集成系统中对一个输入或事件作出回应的一组类，有多少个线程就对应多少个类组，每个线程被集成并分别测试；基于使用的测试是从相对独立的类开始构造系统，然后集成并测试调用该独立类的类，一直持续到构造出完整的系统。

在进行集成测试时，将以类关系图或者实体关系图为参考，确定不需要被重复测试的部分，从而优化测试用例，减少测试工作量，使得测试能够达到一定的覆盖标准。测试所要达到的覆盖标准可以是：达到类所有的服务要求或对所提供的服务达到一定覆盖率；依据类间传递的消息，达到对所有执行线程的一定覆盖率；达到类的所有状态的一定覆盖率等。同时也可以考虑使用现有的一些测试工具来获得程序代码执行的覆盖率。

3. OO 软件的确认测试与系统测试

通过单元测试和集成测试，仅能保证软件的功能得以实现。但不能确认在实际运行时，它是否满足用户的需要，是否存在实际使用条件下诱发错误的隐患。为此，对完成开发的软件必须经过规范的确认测试和系统测试。

OO 软件的确认测试与系统测试忽略类连接的细节，主要采用传统的黑盒法对 OOA 阶段的用例所描述的用户交互进行测试。同时，OOA 阶段的对象-行为模型、事件流图等都可以用于导出 OO 系统测试的测试用例。

系统测试应该尽量搭建与用户实际使用环境相同的测试平台，应该保证被测系统的完整性，对临时不具备的系统设备部件，也应采取相应的手段进行模拟。系统测试时，应该参考 OOA 分析的结果，对应描述的对象、属性和各种服务，检测软件是否能够完全再现问题空间。系统测试不仅是检测软件的整体行为表现，从另一个侧面看，也是对软件开发设计的再确认。

8.8.2 OO 软件测试用例设计

与 OOA 和 OOD 不同，OO 软件的测试用例设计至今还没有统一、成熟的方法，但有关的研究一直在进行，并取得了成果。1993 年，Berard 提出了指导 OO 测试用例设计的方法要点：

- ① 每个测试用例都要有一个唯一的标识，并与被测试的一个或几个类关联。
- ② 每个测试用例都要陈述测试的目的。
- ③ 对每个测试用例要有相应的测试步骤，包括被测对象的特定状态、所使用的消息和操作、可能产生的错误、测试需要的外部环境等。

与传统的测试用例设计不同，OO 测试更多地关注根据测试类的状态设计合适的操作

序列。

1. OO 概念对测试用例设计的影响

类的封装性和继承性对 OO 软件的开发带来很多好处，但它们对 OO 软件的测试却带来了负面影响。OO 测试用例设计的目标是类，类的属性和操作是被封装的，而测试需要了解对象的详细状态。同时，测试还要检查数据成员是否满足数据封装的要求，基本原则是数据成员是否能被外界（数据成员所属的类或子类以外的调用）直接调用。更直观地说，当改变数据成员的结构时，是否影响了类的对外接口，是否会导致相应外界必须改动。例如，有时强制的类型转换会破坏数据的封装特性。假设有如下程序段：

```
class Hidden
{private:
  int a=1;
  char *p= "hidden";}
class Visible
{public:
  int b=2;
  char *s= "visible";}
...
```

```
Hidden pp;
```

```
Visible *qq=(Visible *)&pp;
```

在上面的程序段中，pp 的数据成员可以通过 qq 被随意访问。

此外，继承也给测试用例的设计带来了麻烦。事实上，并没有因继承而减少对子类的测试，相反使测试过程更加复杂化。如果子类和父类的环境不同，则父类的测试用例对子类没有什么使用价值，子类必须设计新的测试用例集。在设计面向对象的测试用例时需要注意以下两点：

(1) 继承的成员函数需要测试

对父类中已经测试过的成员函数，根据具体情况仍然需要在子类中重新测试。若存在以下两种情况则需要对成员函数重新测试：一是继承的成员函数在子类中做了改动；二是成员函数调用了改动过的成员函数的部分。例如，假设父类 Father 有两个成员函数：A()和 B()，子类 Son 只对 B()做了改动。Son::B()显然需要重新测试。对于 Son::A()，如果它有调用 B()的语句，如 x=x/B()，就也需要重新测试。即使子类完全不改地继承了父类的成员函数，但由于这时成员函数被应用于子类的私有属性和操作环境内，该环境与定义它的父类的环境有所不同，因而同样有必要对它进行有限的测试。

(2) 子类的测试用例可以参照父类

在下面例子中，Father::B()和 Son::B()是不同的成员函数，由于面向对象的继承使得两个函数有许多相似处，故可以在 Father::B()的测试要求和测试用例上添加对 Son::B()的新测试

要求和增补相应的测试用例。例如，Father::B()有如下语句：

```
If (value<0) message ("less");
else if (value==0) message ("equal");
else message ("more");
```

Son::B()中有如下语句：

```
If (value<0) message ("less");
else if (value==0) message ("It is equal");
else
{message ("more");
if (value==88)message("luck");}
```

在原有的测试上，对 Son::B()的测试只需做如下改动：修改 value==0 的测试结果期望值；增加对 value==88 这一条件的测试。

2. 类测试用例设计

传统测试中的白盒测试方法可用于类定义的操作的测试，但在面向对象程序中，类成员函数通常都很小，功能单一，所以有些人认为不必注重这种类操作的测试，应该更多地进行类级别的测试。黑盒测试对 OO 系统同样适用。

一般而言，具体设计测试用例，可参考下列步骤：

① 先选定检测的类，参考 OOD 分析结果，仔细分出类的状态和相应的行为，类或成员函数间传递的消息，输入或输出的界定等。

② 确定覆盖标准。

③ 利用结构关系图确定待测类的所有关联。

④ 根据程序中类的对象构造测试用例，确认使用什么输入激发类的状态，使用类的服务和期望产生什么行为等。

值得注意，设计测试用例时，不但要设计确认类功能满足的输入，还应该有意识地设计一些被禁止的用例，确认类是否会产生不合法的行为，如发送与类状态不相适应的消息，要求不相适应的服务等。下面介绍两种常用的类测试用例设计方法。

(1) 基于故障的测试用例设计

基于故障的测试用例设计是通过对 OOA/OOD 模型的分析，找出可能存在的故障，并以此故障设计测试用例的方法。通过这些设计用例可以测出这些可能的故障是否存在。

例如，函数之间调用频繁，容易出现一些不宜发现的错误：

```
if (-1==write (fid, buffer, amount)) error_out();
```

该语句没有全面检查 write() 的返回值，无意中断然假设了只有数据被完全写入和没有写入两种情况。当测试也忽略了数据部分写入的情况时，就给程序遗留隐患。

再如，程序中将 if (strncmp(str1, str2, strlen(str1))) 误写成了 if (strncmp(str1, str2, strlen(str2))), 如果测试用例中使用的数据 str1 和 str2 长度一样，就无法检测出这一错误。

因此，在做测试分析和设计测试用例时，应该注意面向对象程序设计的特点，仔细地进

进行测试分析并设计测试用例，尤其要注意以函数返回值作为条件判断和字符串操作等情况。当然，这个方法多依赖于测试人员的经验来感觉可能存在的故障。如果分析和测试模型可以提供故障的信息，那么基于故障的测试可以以相当低的成本来发现大量的故障。

(2) 基于用例的测试用例设计

基于故障的测试不能发现由错误的功能描述或子系统间交互引起的问题。基于用例的测试关心用户做什么而不是软件做什么，它通过用例捕获用户必须完成什么任务，并以此为依据设计所涉及的各个类的测试用例。

3. 类间测试用例设计

从 OO 软件的集成测试开始，测试用例的设计就要考虑类间的协作。通常可以从 OOA 的对象-关系模型和对象-行为模型中导出类间测试用例。

小 结

编码的目的，是把详细设计的结果“翻译”成用选定的程序设计语言书写的源程序。程序的质量主要是由设计的质量决定的。但是编码的风格和使用的语言，对编码质量也有重要的影响。

软件测试是一个与项目开发并行的过程。按照新的观点，测试活动应分布于需求、分析、设计、编码、测试和验收等各个阶段。它是软件开发时期最繁重的任务，也是保证软件可靠性最主要的手段。

测试的目的是发现程序的错误，而不是证明程序没有错误。大型软件的测试通常分散在 3 个阶段进行。编码阶段应完成单元测试，包括静态分析与动态测试。测试阶段应完成集成测试与确认测试。系统测试则放在安装与验收阶段进行。各级测试都要事先计划，事后报告，并正式存档，供以后维护时使用。

设计测试用例和纠错，是做好软件测试的关键技术。选择测试用例的目标，是用尽可能少的测试数据，达到尽可能大的程序覆盖面，发现尽可能多的软件错误和问题。单元测试应该以结构（白盒）测试为主，其余测试一般以功能（黑盒）测试为主。发现了程序有错误，应进行定位与纠正，这就是纠错的任务。定位是一个分析与推理的过程，提倡周密的思考。

面向对象的开发方法对传统的测试技术提出了新的挑战，面向对象软件的测试策略和测试用例的设计都必须考虑面向对象技术的特征。

习 题

1. 以下 3 个表达式表示的是同一个内容：

(a) $-3**D/3*X$

(b) $-(3**D/3)*X$

(c) $+(((3**D)/(-3))*X)$

- (1) 你认为哪一种可读性最好? 哪一种最差?
- (2) 如果让你列出几条关于书写表达式的指导原则, 你对表达式中运算符的数量和圆括号的层数将作何规定?

(3) 按照你的规定改写下列语句:

$A:=B*C+(((3**D)/(-3)*X)/y)$

2. 假定你想编一个解二次方程的子程序, 准备加入到你所在单位的子程序库中, 供其他程序员使用。

- (1) 试为该子程序写一个序言式的注释。
- (2) 怎样才能把不同的结果(实根、复根、降为一次方程等)有区别地通知调用者?
- (3) 用 Pascal 或其他语言写出这个子程序, 并加上注释。
3. 列出重要的编码风格指导原则, 并按照你心目中的重要性排序。
4. 选择一种结构化语言, 简要写出该语言的重要特点。并用一个小程序做例子, 说明该语言的语法。
5. 软件测试的基本任务是什么? 测试与纠错有什么区别?
6. 怎样理解下面这些话所蕴涵的意义?

(1) 程序测试只能证明错误的存在, 不能证明错误不存在 (Dijkstra)。

(2) 测试是为了证明程序有错, 而不是证明程序无错 (Myers)。

7. 程序测试包括静态分析与动态测试。二者的主要差别在哪里? 为什么有了动态测试, 还要用人工进行代码评审?

8. 用辗转相除法编写求两数最大公约数 (GCD) 的程序, 然后为它设计测试用例。

9. 一元二次方程式 $Ax^2+Bx+C=0$ 的求根程序有以下功能:

- (a) 输入 A 、 B 、 C 这 3 个系数。
- (b) 输出关于根的性质的信息, 包括两个相等或不等的实根, 两个大小相等、符号相反的实根, 仅有一个实根, 或有两个虚根等。

(c) 打印根的数值。

(1) 试用功能(黑盒)测试方法设计程序的测试用例;

(2) 用结构(白盒)测试方法进行验证, 必要时补充一些测试用例。

10. 设有下列程序:

START

INPUT (A,B,C)

IF A>5

THEN x=10

ELSE x=1

ENDIF

IF B>0

THEN Y=20

ELSE Y=2

ENDIF

IF C>5

THEN Z=30

ELSE Z=3

```
ENDIF  
PRINT(X,Y,Z)  
stop
```

试用以下两种标准:

- (1) 满足点覆盖与边覆盖 (即完全覆盖)。
- (2) 满足路径覆盖。

为程序设计测试用例。

11. 多模块程序的测试有哪些层次? 各层测试主要解决什么问题?
12. 集成测试有哪几种实施策略? 试比较它们的优缺点。
13. 面向对象的测试策略和传统测试策略有何区别?
14. 面向对象的测试的基本单位是什么? 如何设计测试用例?

下篇

软件工程的近期进展、 管理与环境

- 第 9 章 软件维护
- 第 10 章 软件复用
- 第 11 章 软件工程管理
- 第 12 章 软件质量管理
- 第 13 章 软件工程环境
- 第 14 章 软件工程高级课题

第9章 软件维护

软件产品交付用户后，就进入生存周期的最后一个时期——运行时期了。要想充分发挥软件的作用，产生良好的经济效益和社会效益，必须做好软件的维护。

大、中型软件产品的开发周期一般为1~3年，运行周期则可达5~10年。在这么长的时间内，除了要改正软件中的残留错误外，还可能多次更新软件的版本，以改善运行环境（包括硬件与软件的改进）和提升产品性能等需要。这些活动都属于维护工作的范畴。能不能做好这些工作，将直接影响软件的使用寿命。

维护是生存周期中花钱最多、延续时间最长的活动。在本书第1章就已指出，有人把维护比成“墙”或“冰山”，以比喻它给软件生产所造成的障碍。不少单位为了维护已有的软件，竟没有余力顾及新软件的开发。典型的情况是，软件维护费与开发费用的比例为2:1，一些大型软件的维护费用，甚至达到当时开发费用的40~50倍。这也是造成软件成本大幅度上升的一个重要原因。

9.1 软件维护的种类

软件维护的最终目的，是满足用户对已开发产品的性能与运行环境不断提高的需要，进而延长软件的寿命。软件维护是指软件系统交付使用以后，为了改正或满足新的需要而修改软件的过程。国标GB/T11457—95对软件维护的定义是：在一个软件产品交付使用后对其进行修改，以纠正故障、改进其性能和其他属性，或使产品适应改变了的环境。按照每次进行维护的具体目标，又可以分为以下3类。

1. 完善性维护（perfective maintenance）

无论是应用软件或系统软件，都要在使用期间不断改善和加强产品的功能与性能，以满足用户日益增长的需要。开发后刚投入使用的版本是第一版，以后还可能有第二版、第三版……，所以有人建议，将软件生存周期改成

开发 → 改进 → 改进 → …

才更为符合实际。在整个维护工作量中，完善性维护约占50%~60%，居于第一位。

2. 适应性维护 (adaptive maintenance)

系指使软件适应运行环境的改变而进行的一类维护。其中又包括：因硬件或支撑软件改变（如操作系统改版、增加数据库或者通信协议等）引起的变化；将软件移植到新的机型上运行；软件使用对象的较小变更，例如潘兴Ⅱ导弹改为潘兴Ⅲ只需作不大的修改。这类维护大约占整个维护的25%。

3. 纠错性维护 (corrective maintenance)

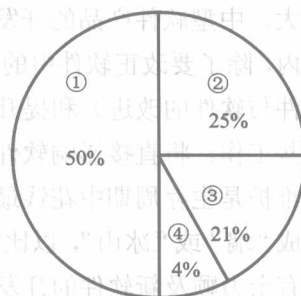
其目的在于纠正正在开发期间未能发现的遗留错误。由于软件测试的不彻底性，任何大型软件交付使用后，都会继续发现潜藏的错误。对它们进行诊断和改正，就称为纠错性维护。这类维护约占总维护量的20%。

根据 Lientz 和 Swanson 在 1980 年对 487 个软件开发机构的调查，上述各类维护在整个维护活动中所占的比重如图 9.1 所示。

除上述 3 类外，有人主张再增加一个第四类，并称之为预防性维护 (preventive maintenance)。

预防性维护是 J. Miller 首先创导的，他主张，维护人员不要单纯等待用户提出维护的请求，应该选择那些还能使用数年、目前虽能运行但不久就需作重大修改或改进的软件，进行预先的维护。其直接目的是改善软件的可维护性，减少今后对它们维护时所需要的工作量。

早期开发的软件，是预防性维护的重要对象。这类软件有一部分仍在用，但开发方法陈旧，文档也不齐全。选择其中符合上述条件的软件作预防性维护，对它们的全部或部分程序重新设计、编码和调试，在经济上常常是合算的，Miller 称之为结构化的翻新 (structured retrofit)，用他的话来说，就是“把今天的方法用于昨天的系统，支持明天的需求”。



- ① 完善性维护 50%
- ② 适应性维护 25%
- ③ 纠错性维护 21%
- ④ 其他维护 4%

图 9.1 各类维护的比重

9.2 软件可维护性

1. 可维护性的含义

所谓可维护性 (maintainability)，是衡量维护容易程度的一种软件属性。定性地说，可维护性取决于软件的可理解性、可修改性与可测试性，三者一起构成为软件的质量属性。

(1) 可理解性 (understandability)

在多数情况下，维护者并非软件的开发人员。众所周知，读懂别人写的程序是困难的，如

果仅有程序而无文档，则难度更大。例如第四代开发工具和高级语言书写的程序，比汇编语

(2) 可修改性 (modifiability)

正如纠错时可能引入新错误一样，维护时也可能把程序和文档改错。一般的说，可修改性好的程序，在维护时出错的概率也小。例如模块的独立性愈高，软件开发时采用的技术愈先进（面向对象技术和复用技术），则维护中出错的机会也就越少。

(3) 可测试性 (testability)

可测试性代表一个软件容易被测试的程度。源程序良好的可理解性、齐全的测试文档（包括开发时期用过的测试用例与结果），都能提高程序的可测试性。

上述可维护性的好与坏是定性的说法。能不能对它进行定量的度量呢？

1979年，T. Gilb 建议把维护过程中各种活动耗费的时间记录下来，并用它们来间接度量软件的可维护性。他建议记录的 10 种时间如下：

- | | |
|--------------|--------------|
| ① 问题识别时间。 | ② 管理延迟进间。 |
| ③ 收集维护工具时间。 | ④ 问题分析时间。 |
| ⑤ 修改规格说明书时间。 | ⑥ 改正（或修改）时间。 |
| ⑦ 局部测试时间。 | ⑧ 整体测试时间。 |
| ⑨ 维护复审时间。 | ⑩ 分发与恢复时间。 |

记录这些时间数据不会有什么困难。有了这些数据，对软件的可维护性就能作出定量的评价了。

2. 提高可维护性的途径

提高软件的可维护性，最根本的是使每个开发人员懂得维护的重要性，在开发阶段就以减少今后的维护工作量为努力的目标。

由上可见，有两个开发活动与提高软件的维护性关系比较大，即提供完整和一致的文档，以及采用现代化的开发方法。

文档的第一个作用，是帮助维护人员读懂程序。早期开发的非结构化软件，基本上是不可维护的。文档的第二个作用，是方便被维护软件的测试。一般的说，每一个交付使用的软件都应配置一个测试用例文件，来记录综合测试和确认测试的测试用例与测试结果。当软件在维护中被修改以后，应增加一些测试用例来检验被修改的代码，同时把原有的测试用例全部重测一遍，以检查在修改中是否产生意外的副作用。

是否使用现代化的开发方法，是影响软件可维护性的又一重要因素。除上述的尽可能采用面向对象技术和复用技术等现代化方法外，对数据量比较大的应用软件，还可以利用数据

库技术来管理软件中的数据，借以减少对报表软件的维护工作量。

还应指出，不仅在开发时期要尽量提高软件的可维护性，在维护时期更要保持程序的可维护性，使之不受损害或破坏。维护人员需知，正如你要维护别人开发的软件一样，很可能还有别人要维护你维护过的软件。

9.3 软件维护的实施

1. 软件维护的工作流程

图 9.2 概括了一项维护申请从提出到实现的基本过程。

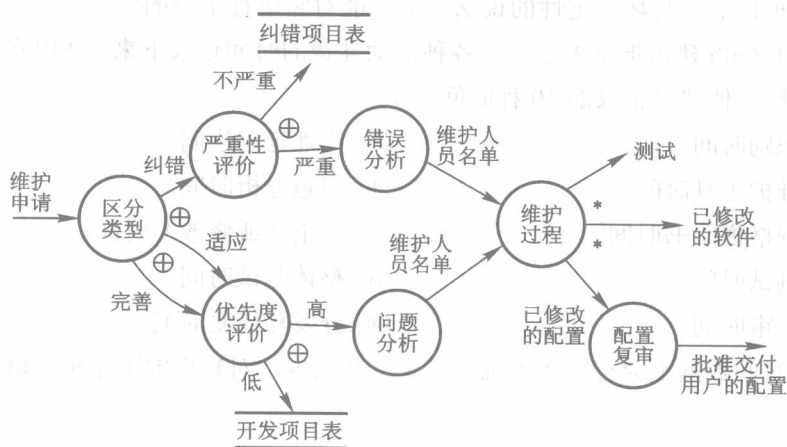


图 9.2 维护的步骤

由图 9.2 可知，管理部门在接到维护申请后，首先要区分维护的类型。如果是纠错性维护，应该先估计错误的严重性。凡属严重的错误（例如关键子系统失效），需立即进行初步的分析，指定维护人员开始工作。对于不太严重的错误，可暂记入“纠错项目表”文件，定期地（例如一周或一月）安排维护。如果是适应性或完善性维护申请，则应在与用户充分协商的基础上确定轻重缓急。除了个别特别紧迫、列入高优先级的项目需要立刻开始外，其余的申请都可以看成新的软件开发项目，记入“开发项目表”等待统筹安排。

在图 9.2 中，“维护过程”只用了一个圆框来表示。实际上，它包含了修改软件设计、设计复审、代码修改、单元测试、综合测试、直到确认测试等全部维护工作。从本质上说，维护工作可以看作开发工作的一个缩影。凡是开发时期用到的方法与文档，维护时期都使用。以下仅对维护工作不同于开发工作的地方，作一简要的补充说明。

2. 维护申请单和软件修改报告单

这是维护时期新增加的两个文档，分别用于维护开始之前和结束以后。维护申请单（maintenance request form, MRF）又称修改（change）申请单或软件问题报告单（software problem report, SPR），通常由申请维护的用户填写。纠错性维护的申请单应完整地说明导致错误发生的环境，包括输入数据、输出数据清单和其他有关材料。如果申请适应性或完善性维护，则仅需提出一个简要的需求说明。软件修改报告单（software change report, SCR）用于记录在维护时期对软件所作的每一次修改。其内容包括问题来源、错误类型、修改内容、资源（成本）耗用以及批准修改的负责人等。修改报告由直接进行修改和负责文档管理的人员共同填写。

3. 维护的副作用

另一个在维护中值得注意的问题，是防止因修改而引起副作用。对于大型软件，即使一个小小的修改，也可能引入新错误。一些报告指出，当修改范围少于 10 条语句时，一次成功的概率约为 50%，若修改范围涉及 50 条语句，一次成功的概率将降为 20% 左右。如果对修改的部分不作严格的测试，几乎不可避免会引入新的错误。

在《程序和系统的维护技术》一书中，Freedman 和 Weinberg 把修改可能产生的副作用归结为以下 3 类：

① 修改编码的副作用。较多地产生在对变量标识符、语句标号或子程序进行修改或删除的时候。修改逻辑运算符，修改打开或关闭文件的语句，也容易引入错误。当修改涉及代码的主要部分，或者为了改进程序性能而对代码进行修改时，尤其要警惕引入新的错误和问题。

② 修改数据的副作用。维护中修改数据结构或其中的某些数据项，是常有的事情。稍有不慎，修改后的数据就可能与软件设计不再适应，从而产生错误。经验表明，在做如下修改时容易引入错误：重新定义记录和文件的格式，重新定义局部或全局常数，改变子程序形式参数的顺序，改变数组或其他高级数据结构的大小，修改全局数据的相关内容。在设计文档中对数据结构作完整的描述，建立各个模块对各数据的交叉引用表，都将有助于减少这一类副作用。

③ 修改文档的副作用。文档与程序是软件的组成部分。任何对程序的修改，都应该及时反映到有关的文档上。如果只改程序而未修改有关文档（包括分发给用户的使用指南等手册），必将在今后的使用和维护中造成混乱。

9.4 软件维护的管理

本章 9.3 节提到，维护活动是开发活动的一个缩影。软件开发要同时做好技术与管理两

方面的工作，维护也是如此。本节仅就维护时期的一些管理工作进行简要的说明。

1. 维护的机构与人员

维护是软件开发单位的责任。软件开发单位根据本身规模的大小，可以指定一名高级管理人员担任维护管理员，或者建立由高级管理人员和专业人员组成的修改控制组（change control board, CCB），管理本单位开发的软件的维护工作。管理的内容，应包括对申请的审查与批准，维护活动的计划与安排，人力和资源的分配，批准并向用户分发维护的结果（如软件的新版本），以及对维护工作进行评价与分析等。

具体的维护工作，可以由原开发小组承担，也可以指定专门的维护小组进行。前者的优点是他们熟悉被维护的软件。但由于开发小组很可能已接受新的开发任务，对旧软件的维护会分散他们的精力。后者的优点是精力集中，且能使其他的开发人员无后顾之忧。不足的是，多数人不安心长期担任专职的维护人员，认为“维护程序员”的名声不如开发人员好听。有人认为，采取开发人员与维护人员定期轮换的方法，其效果可能更好。

2. 维护管理文档

除了移用开发时期的文档外，有几种文档是专供维护时期使用的。上节提到的维护申请单和软件修改报告单，就是其中的两种。这里再介绍几种常用的文档。

① 维护日志。这是维护管理员评价维护工作有效性的主要依据。按照 Swanson 的建议，日志主要包括以下 3 个方面的内容：第一是维护前程序的情况。例如程序的名称，语句或指令条数，所用的语言，安装启用日期，启用以来运行的次数和其中运行失效的次数等；第二是维护中对程序修改的情况。例如修改的级别，增加和删除的源语句数，修改日期与修改人，每一项改动耗费的人-时数等；第三是其他的重要数据，如维护申请单的编号，维护的类型，维护起止日期，耗用的总人-时数，维护完成后产生的净收益等。维护日志在每次维护完成后填写，它是软件配置的组成部分，也可以存入配置管理数据库，供需要时查询。

② 维护申请摘要报告和趋势图。前者是一种定期报告，可以每周或每月统计一次。其内容包括上次报告以来已经处理了的、正在处理的和正在处理项目新接到的维护申请项数及其处理情况，以及新申请中特别紧迫的问题。后者则是在前者基础上绘制而成，是一种不定期的报告。图 9.3 是维护趋势图的一个例子。图中显示了在统计的时期内每月收到的新维护申请以及正在处理的申请项数。

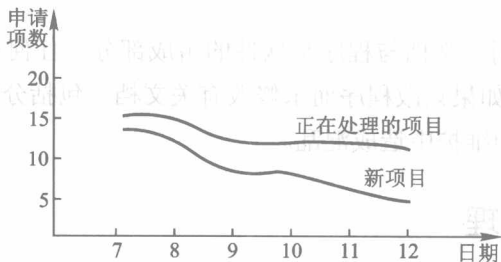


图 9.3 7~12 月 xx 软件维护趋势图

上述两种文档提供的信息有多种用途。例如，在接到新的申请时，可以用前一文档核对以往有没有接到过类似的申请。如果有，则其处理情况怎样？后一文档可以显示维护的工作量及其变化趋势。如果把维护按纠错、适应和完善等不同种类分别统计，还可以从纠错性维护次数的变化中，看出软件在维护后的质量是提高了还是下降了。

3. 维护费用的估算

如前所述，软件维护可以看作软件开发的一个缩影。因此 Boehm 建议维护的工作量可以参照开发工作量来计算。其具体方法是，首先估计在一个时期内可能要修改或增加的源代码指令数，与原有的指令总数比较，得出一个活动比（activity ratio）

$$ACT = (\text{修改的指令数} + \text{增加的指令数}) / \text{指令总数}$$

然后，就可从开发时期耗费的人-月数估算出维护所需的人-月数，即

$$MM_{\text{维护}} = ACT \times MM_{\text{开发}}$$

考虑到开发时期对软件可靠性的重视程度以及采用现代软件开发方法的多少，可以再加上一个工作量调节因子（effort adjustment factor, EAF），于是上式又变成

$$MM_{\text{维护}} = ACT \times EAF \times MM_{\text{开发}}$$

Belady 与 Lehman 则认为，在开发时期如果没有采用结构化的方法与文档，就会增加被维护软件的复杂度。另一方面，维护人员是否熟悉所维护的软件，也会直接影响维护的工作量。这二者加在一起，实际上决定了维护人员用于理解被维护软件所需要的工作量。因此，他们建议以下列公式作为估算维护工作量的模型：

$$M = P + K^{(c-d)}$$

其中， M = 维护所需总工作量；

P = 生产性工作量，包括问题分析、修改设计与编码、回归测试等；

K = 经验常数；

c = 因未采用结构化的方法与文档而增加的软件复杂度；

d = 维护人员对软件的熟悉程度。

由上述模型可见，等式右边的第二项可以看作维护人员消化被维护软件的资料所花费的工作量。如果软件开发方法陈旧，原开发人员又不能参加维护，则该项因素将指数地增加，从而使维护费用大大上升。

9.5 软件配置管理

缺乏软件开发管理，会导致种种问题的出现，这些问题使得最终开发出来的软件产品的质量难以保证，应用难以稳定。那么，怎样进行软件开发管理才能生产出高质量的软件产品呢？在 ISO 9000 质量管理和质量保证标准中，制定了《在软件开发、供应和维护中的使用指南》标准，该标准除对软件生存周期的各个阶段做了严格的规定外，还在其质量体系中规定了与阶段无关的支持活动，其中软件配置管理（software configuration management, SCM）

被放在首位。

SCM有多种定义。Wayne Babyish在1986年出版的《Software Configuration Management: Coordinating for Team Productivity》一书中,把SCM描述为“对软件开发组所建立的软件的修改进行标识、组织和控制的艺术,其目标是减少错误和提高生产力”。Steve McConnell在1993年出版的《Code Complete》一书中,又从另一个角度对SCM进行了定义:“配置管理能够系统地处理变更,从而使得软件系统可以随时保持其完整性。故配置管理也可称为‘变更控制’,用来评估提出的变更请求,跟踪变更,并保存系统在不同时间点上的状态。”

由此可见,SCM是一套规范、高效的软件开发基础结构。在发达国家软件产业的发展 and 实践中,早已被所证明为管理软件开发过程的有效方法。作为整个软件过程的软件质量保证(SQA)活动之一,它也可应用于整个软件过程的保护性活动。

1. 配置管理的功能

SCM应具备以下主要功能:系统地管理软件系统中的多重版本;全面记载系统开发的历史过程,包括为什么修改,谁作了修改,修改了什么;管理和追踪开发过程中危害软件质量以及影响开发周期的缺陷和变化。它还能对开发过程进行有效的管理和控制,可完整、明确地记载开发过程中的历史变更,形成规范化的文档,不仅使日后的维护和升级得到保证,而且更重要的是,它还会保护宝贵的代码资源,积累软件财富,提高软件重用率,加快投资回报。

2. 软件配置项

Pressman对软件配置项(software configuration item, SCI)给出了一个比较简单的定义:“软件过程的输出信息可以分为3个主要类别:其一是计算机程序(源代码和可执行程序),其二是描述计算机程序的文档(针对技术开发者和用户),其三是数据(包含在程序内部或外部)。这些项包含了所有在软件过程中产生的信息,总称为软件配置项。”由此可见,软件配置项的识别是配置管理活动的基础,也是制定配置管理计划的重要内容。

软件配置项分类软件的开发过程是一个不断变化的过程,为了在不严重阻碍合理变化的情况下来控制变化,软件配置管理引入了基线(baseline)这一概念。IEEE对基线给出了如下的定义:“已经正式通过复审批准的某规约或产品,它因此可作为进一步开发的基础,并且只能通过正式的变更控制过程变化。”根据这个定义,可以在软件的开发流程中把所有需加以控制的配置项分为基线配置项和非基线配置项两类,例如,基线配置项可能包括所有的设计文档和源程序等;非基线配置项可能包括项目的各类计划和报告等。

配置管理贯穿于整个生存周期,在运行和维护时期,其任务尤为繁重。为了方便对多种产品和多个版本进行跟踪和控制,常常借助于自动的配置管理工具。

3. 配置管理工具

第一类常用的工具是配置管理数据库。它存储关于软件结构的信息,产品的当前版本号及其状态,以及关于每次改版和维护的简单历史记录。数据库能够回答管理人员的种种问题,例如,每个产品有哪些版本,每版有什么差别,各种版本都有哪些文档,已分发给哪些用户,

以及有关产品维护历史、纠正错误的数量等。

另一类工具称为版本控制库(version control library)。它可以是上述数据库的一个组成部分,也可以单独存在。它与数据库的差别是,后者对所有软件产品进行宏观管理的工具,而版本控制库则着眼于单个产品,以文件的形式记录每一产品每种版本的源代码、目的代码、数据文件及其支持文档。每一文件均记有版本号、启用日期和程序员姓名等标识信息。管理人员根据需要,可以对任何文件进行建立、检索、编辑、编译(或汇编)等操作。

9.6 软件再工程

从理论上讲,只要应用前面所述的软件维护管理和方法,一个软件经过维护后,仍然是可维护的。但在实际工作中,软件维护工程师往往很难做到完全遵循软件工程方法。随着一次次软件的纠错或完善,会产生一些不可预测的问题,软件的稳定性和可维护性就越来越差。因而软件维护被喻为“冰山”,在表面之下是大量不可见的问题和成本。所谓软件再工程(software reengineering),就是将新技术和新工具应用于老的软件的一种较“彻底”的预防性维护。

软件再工程不同于一般的软件维护。后者是局部的,以完成纠错或适应需求变化为目的;而软件再工程则是运用逆向工程(reverse engineering)、重构(restructure)等技术,在充分理解原有软件的基础上,进行分解、综合,并重新构建软件,用以提高软件的可理解性、可维护性、可复用性或演化性(evolvability)。

1. 软件再工程过程模型

在 Pressman 建议的一个软件再工程过程模型(如图 9.4 所示)中,他为软件再工程定义了 6 类活动。一般情况下,这些活动是顺序发生的,但每个活动都可能重复,形成一个循环的过程。这个过程可以在任意一个活动之后结束。以下从信息库分析开始,依次对各类活动作简要说明。

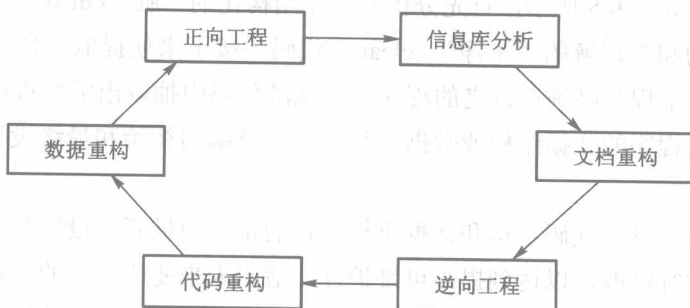


图 9.4 软件再工程过程模型

信息库中保存了由软件公司维护的所有应用软件的基本信息，包括应用软件的设计、开发及维护方面的数据，例如最初构建时间、以往维护情况、访问的数据库、接口情况、文档数量与质量、对象与关系和代码复杂性等。在确定对一个软件实施再工程之前，首先要收集上述这些数据，然后根据业务重要程度、寿命、当前可维护情况等对应用软件进行分析。

文档重构是重新构建原本缺乏文档的应用系统的文档。根据应用系统的重要性和复杂性，可以选择对文档全部重构、部分重构或维持现状。

逆向工程是一个恢复原设计过程。通过分析现存的程序，从中抽取出数据、体系结构和过程的设计信息。

代码重构是在保持系统完整的体系结构基础上，对应用系统中难于理解、测试和维护的模块重新进行编码，同时更新文档。

数据重构是重新构建系统的数据结构。数据重构是一个全范围的再工程活动，它会导致软件体系架构和代码的改变。

正向工程也称革新或改造，它根据现存软件的设计信息，改变或重构现存系统，以达到改善其整体质量的目的。

2. 逆向工程

逆向工程一词源于硬件制造业。互相竞争的公司为了了解对方设计和制造的机密，在得不到设计文档的情况下，常通过拆卸实物来获取信息。软件的逆向工程是指从源代码出发，重新恢复设计文档和需求规格说明书。目前，已经出现了一些 CASE 工具来帮助实现逆向工程，它们或使源代码能以更清晰的方式显示，或直接从源代码中产生流程图或结构图之类的图表。在理想状态下，逆向工程应该能导出多种不同层次的抽象，包括：低层的抽象——过程的设计表示；稍高一点层次的抽象——程序和数据结构信息；相对高层的抽象——数据和控制流模型；高层抽象——实体-关系模型等。

逆向工程过程如图 9.5 所示，首先分析并将非结构化的“脏”（dirty）源代码重构为结构化的程序设计结构和易理解的“干净”（clean）代码；接下来便提取、抽象，这是逆向工程的核心活动，此时工程师必须评价老的程序，并从源代码中抽取出需要程序执行的处理、程序的用户界面以及程序的数据结构或数据库结构，相继编写初始和最终设计说明文档。

3. 软件重构

软件重构又可区分为代码重构和数据重构，目的是应用最新的设计和实现技术对老系统的源代码和数据进行修改，以达到提高可维护性、适应未来变化的目的。重构不改变系统的整体体系结构，一般仅局限在单个模块的设计细节和模块内部的局部数据结构。如果超出了模块的边界并涉及软件的体系结构，这时的重构就变成正向工程了。

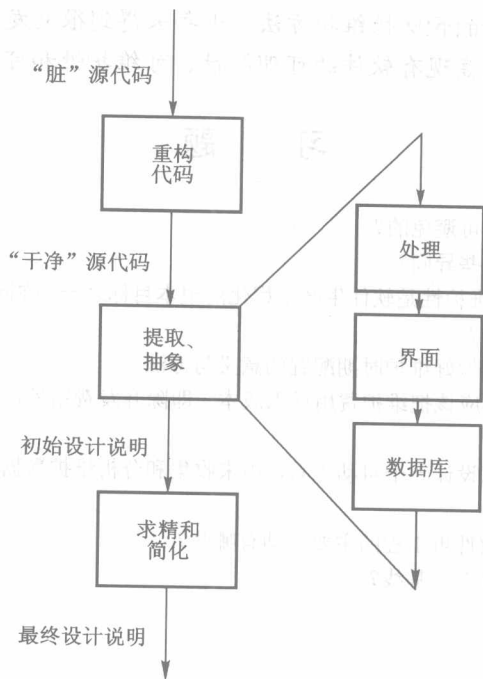


图 9.5 逆向工程过程

小 结

在软件生存周期中，维护工作是不可避免的，按照不同的目标，维护活动可以分为 4 类：以加强软件功能为目标的完善性维护；纠正软件遗留错误的纠错性维护；为了适应运行环境变化而进行的适应性维护；以及为改进软件的可维护性、减少将来的维护工作量而进行的预防性维护。花在维护上的费用，通常要占软件总费用的一半以上。对于大型和复杂的软件，维护费用可能达到开发费用的十至数十倍。

软件的可维护性，主要决定于开发时期的活动。用软件工程方法来开发软件，编制齐全的文档，尽量可能采用先进的软件开发技术，是改善软件可维护性的关键。每个开发人员都应认识到维护工作的重要性，在开发中尽力提高软件的可维护性，而不是相反。

维护工作是开发工作的缩影，但又有自己的特点。要缩小维护的副作用，尽量避免在维护中引入新错误降低软件的质量；要加强对维护的管理尤其是配置管理，有效地对软件配置进行跟踪和控制，避免造成文档的混乱。维护人员需知，不适当和不充分的维护，可能把一个原来好端端的软件变成一个不可维护的软件，造成灾难性的后果。明白了这个道理，即使对于微小的修改，也要严格遵守规定的步骤和标准，决不能掉以轻心。

软件再工程作为一种新的预防性维护方法,近年来得到很大发展。它通过逆向工程和软件重构等技术,可有效地提高现有软件的可理解性、可维护性和可复用性。

习 题

1. 为什么说软件维护是不可避免的?
2. 纠错和纠错性维护有哪些异同?
3. 有人说,提高软件的可维护性是软件生产工程化的根本目标之一。你同意这个观点吗?试说明理由。
4. 怎样避免维护的副作用?
5. 什么是软件配置?说明做好维护时期配置的意义与方法。
6. 计算软件价格时,应不应该把维护费用计入成本(即除开发费用外,还要加上预计的维护费用)?为什么?
7. 某单位的维护管理员想设计一个自动工具,用来收集和分析维护数据,你能帮助他提出对这一软件工具的需求吗?
8. 什么是软件再工程?软件再工程的主要活动有哪些?
9. 什么是软件配置项?什么是基线?

第10章 软件复用

软件复用是现代软件工程的一个重要概念。在本书第 1、2 两章中，已多次提到过软件复用。概括地说，它不仅是一种技术、一种方法，而且也是一个过程，目的是使软件开发在质量、生产效率和成本上得到改善甚至大幅度的优化。在这一章中，将对与之有关的问题集中进行一次较系统的介绍。

10.1 软件复用的基本概念

人们在考虑做一件事的时候，总是习惯地先回想一下这件事以前做过没有，是如何做的。如果没有，则从头开始，否则就会沿用已有的经验和方法，提高做事的效率和准确度。这些经验和方法不断地总结和推广，就形成了解决新问题时常见的复用思想。活字印刷术中用到的“活字”，就是生活中人们熟悉的对印刷产品进行复用的实例。

在计算机程序设计中调用标准函数库中的函数，可以视为软件复用的早期的例子。简单地说，软件复用就是将已有的软件成分用于构造新的软件系统。被复用的软件成分称为可复用构件，无论是对它原封不动地使用还是作适当的修改后再使用，只要是用它来构造新软件的，都可以看作复用。但如果在一个系统中多次使用一个相同的软件成分，则称为软件共享而不称作复用；对一个软件进行修改，使之能运行于新的软、硬件平台也不能称作复用，而称为软件移植。

10.1.1 软件复用的定义

早在 1968 年的 NATO 软件工程会议上，D. McIlroy 就提出了可复用构件的思想。1983 年，Freeman 对软件复用给出了这样的定义：“在构造新的软件系统的过程中，对已存在的软件人工制品的使用技术。”这个定义实际上包含两个方面的内容，一是制造软件构件的技术，二是使用软件构件的技术。

有人把该定义引伸为“开发伴随复用，开发为了复用”，精辟地概括了软件复用与软件开发的相互关系。所谓制造软件构件的技术，是指独立于单个软件系统开发的、可服务于整个应用领域的构件生产技术；使用软件构件的技术有时也称为基于构件的软件开发（component based software development, CBSD），是指在软件系统开发中使用已有软件构件的方法和技术。前者属于领域工程（domain engineering）的范畴，后者属于软件工程的范畴，

二者共同组成基于构件的软件工程（component based software engineering, CBSE）。图 10.1 显示了 CBSE 的一种过程模型。在本章第 10.2 节和第 10.3 节将对领域工程和基于构件的开发分别进行介绍。

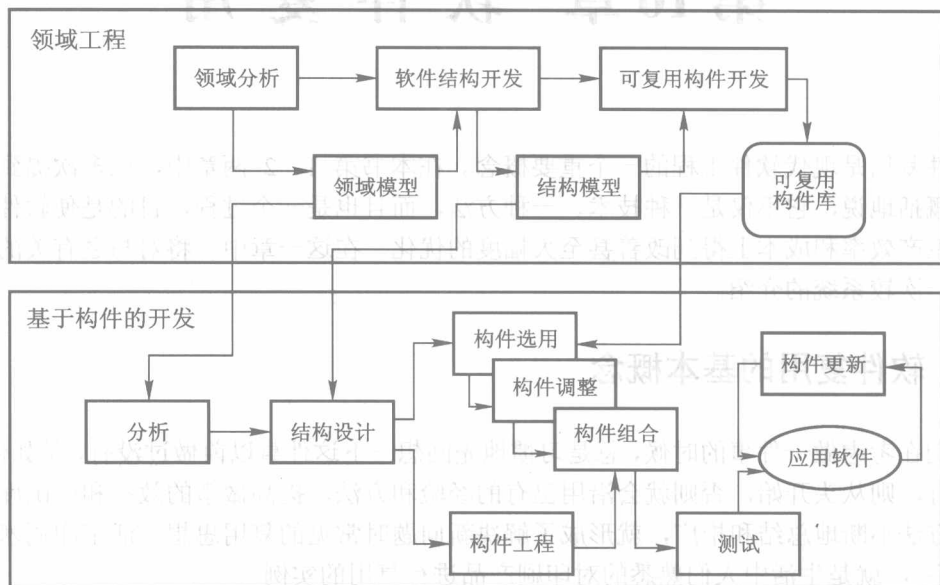


图 10.1 CBSE 的一种过程模型

10.1.2 软件复用的措施

软件复用的目的是更快、更好、成本更低地生产软件产品。一般地说，在软件开发中采用复用构件可以比从头开发这个软件更加容易。曾有人做过这样的比较，让两个技术能力相当的软件开发小组开发同一个项目，第一小组在开发过程中采用软件复用技术，有 30%~40% 的内容可从软件构件库中得到，第二小组从头开发所有需要的内容。最终的结果是：

- ① 第一小组比第二小组更快地完成项目。
- ② 第一小组完成项目所需成本大大低于第二小组。
- ③ 第一小组的产品比第二小组的产品有更少的错误。

在 20 世纪 80 年代中期，日本的软件大公司的复用率就已达到 50%。到 20 世纪末，有些日本、美国大公司的软件复用率已接近 90%。但是就我国多数软件生产企业而言，仍把软件复用看作可有可无的技术。有些企业力图在管理上做文章，通过在软件开发过程中加入有效的管理（如 CMM，详见第 12 章）来提高软件生产效率，殊不知与管理相比，采用软件复用技术可获得投入更小、回报更大的效果。

一般认为，要在企业内部充分地实现软件复用，应优先采取以下一些措施：

- ① 在充分认识软件复用的重要性的基础上，尽快建立支持软件复用的基础设施，包括可复用构件库、用于创建复用构件的工具（例如一个制造可复用构件的辅助设计系统）等。
- ② 建立相应的培训计划，帮助软件工程师和管理人员理解和应用软件复用，在企业内部形成一个使用软件复用技术的环境。
- ③ 采用更先进的、可以促进软件复用的软件开发方法，如面向对象的开发方法。
- ④ 采取相应的激励措施。目前很多企业仍沿用传统的计算工作量的方法（如计算代码行），复用后工作量反而减少了，因此要采用相应的激励措施，鼓励软件工程师使用软件复用技术。

10.1.3 软件复用的粒度

Caper Jones 定义了 10 种可供复用的软件制品，包括源代码、体系结构、需求模型和规格说明、各种设计、用户界面、数据、测试用例、用户文档和技术文档、项目计划、成本估计等。按照复用粒度的大小，这些软件制品可从小到大分为以下 4 类。

1. 源代码复用

源代码（例如子程序）复用是指对构件库中用高级语言编写的源代码构件的复用。它是最常见的复用，也是粒度最小的复用，仅能带来较小的生产率收益和可靠性收益。

实际上源代码构件本身就是为复用而开发的，存放在可供访问的构件库中。使用者通过对构件库的检索找到适用的构件，并设置参数值使之具有新的适应性，即可通过调用过程来调用构件。不难看出，这类复用的特点是：一方面由于构件是经过充分测试的，因此具有较高的可靠性，而且使用者只需设置参数而无须介入构件内部，降低了复用的难度；但另一方面，正因为构件是为复用而开发的，因此其通用性、抽象性成为在具体使用时必须面对的问题。

2. 软件体系结构复用

软件体系结构复用是指对已有的软件体系结构的复用。这类复用既可以支持高层次的复用，也可以支持低层次的复用。要求存放体系结构的库能提供有效的检索，使用者通过良好定义的接口进行集成。

这类复用的特点是：一方面，可复用较大粒度的软件制品，其修改具有局部性；另一方面，因为难以抽象出简明的描述，存放体系结构的库往往不易管理。

3. 应用程序生成器

应用程序生成器用于对整个软件系统的设计的复用，包括整个软件体系结构、相应的子系统、特定的数据结构和算法。通常，从高层的领域特定的需求规约自动生成一个完整的可执行系统，生成器根据输入的规约填充原来不具备的细节。一般仅针对一些成熟的领域。

这类复用的特点是：一方面，自动化程度高，能获取某个特定领域的标准，以黑盒形式输出结果（应用程序）；另一方面，特定的应用程序生成器不易构造。

4. 领域特定的软件体系结构的复用

这类复用是指对特定领域中存在的一个公共体系结构及其构件的复用。要求对领域有透彻的理解才能进行领域建模；库是针对特定领域的；领域模型、基准体系结构和库都随着领域的发展而不断发展。基准体系结构用体系结构描述语言来描述，根据相关领域特定的刻面（faceted，见10.2节）集合，从库中选择基准体系结构和构件，并通过良好定义的接口进行集成。

这类复用的特点是：一方面，其复用的程度高，对可复用构件的组合提供了一个通用框架；另一方面，前期投资很大。

10.2 领域工程

构件开发人员首先面对的问题，是如何定义新的构件，即如何在应用领域的模型中找出有共性、可通用的部分。为了解决这一问题，在做构件之前首先要明确它适用于哪个应用领域，然后根据这个领域的知识和应用模型，抽取最合理的构件定义。

这里的领域，通常指一组具有相似或相近软件需求的应用系统所覆盖的区域。应用系统与应用系统之间的功能交集，为软件复用提供了机会。交集的面积越大，可复用的功能就越多。由最多应用系统形成的交集，就是最优复用；其次的就是次优复用；依次类推。所谓领域工程，就是通过领域分析找出最优复用，把它们设计和构造为可复用构件，进而建立大规模的软件构件仓库的过程。

10.2.1 横向复用和纵向复用

按复用活动所应用的领域范围，复用可划分为横向复用和纵向复用两种。

横向复用是指复用不同应用领域中的软件元素，例如数据结构、分类算法、人机界面构件等。标准函数库就是一种典型的原始横向复用机制。

纵向复用是指在一类具有较多公共性的应用领域之间，对软件构件进行复用。由于在两个截然不同的应用领域之间实施软件复用非常困难，所以纵向复用才广受瞩目，并成为软件复用技术的焦点。

下面以纵向复用的领域工程为例，重点说明它所包括的以下几方面的主要活动：

① 实施领域分析。它是领域工程的核心。根据应用领域的特征及相似性，可预测软件构件的可复用性，发现并描述可复用实体，进而建立相关的模型和需求规约。

② 开发软件构件。一旦确认了软件构件的复用价值，即可进行构件的开发并对具有复用价值的构件进行抽象化、一般化和参数化，以便它们能适应新的、类似的应用领域。

③ 建立软件构件库。将软件构件及其文档分类归并，形成相关的分类检索机制，成为可供后续项目使用的可复用资源。

图10.2 简明地显示了纵向复用领域工程主要活动的内容及其与软件开发的关系。

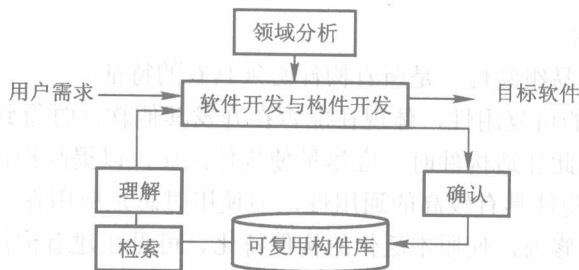


图 10.2 纵向复用领域工程的活动内容

10.2.2 实施领域分析

领域分析是在特定应用领域寻找最优复用的过程，它以公共对象、类、子集合和框架等形式进行标识，然后对它们进行分析和规约。这个阶段的主要目标是获得领域模型（domain model），而领域模型描述的需求为领域需求。根据领域需求，领域工程师寻找领域的共性，进而确定软件的可复用构件。

图 10.3 显示了怎样从与领域相关的知识，通过领域分析来获得领域的分析模型。一般的说，领域分析可成以下几个步骤：

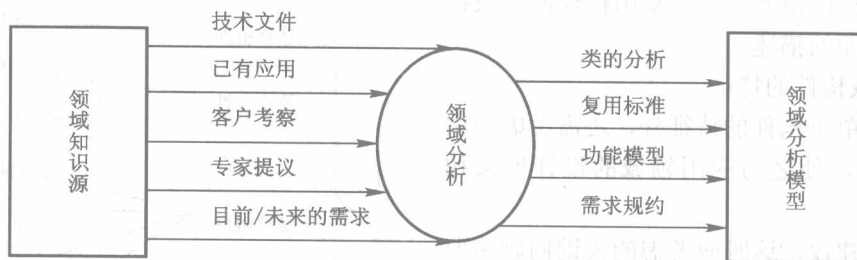


图 10.3 领域分析的输入和输出

- ① 选择特定的领域。
- ② 抽取具有最优复用的功能。
- ③ 标识该功能。
- ④ 建立领域分析模型。
- ⑤ 定义需求规约，得出由需求规约构件和若干个连接子配置构成的领域模型。

10.2.3 开发可复用构件

开发构件最终目的是为了复用，即为了以后能复用而建造构件。为使构件能具有较高的可复用性，在建造构件的时候应遵循以下的原则与方法。

1. 单个构件的特征

通用性、可变性和易组装性，是所有构件必须具有的特征。

① 通用性。构件的可复用性，体现在能否在开发其他软件时得到使用。使用率越高，说明可复用度越高。因此建造构件时，应尽量使构件泛化，以提高构件的通用性。

② 可变性。尽管构件具有较高的通用性，但使用时总是应用在一个具体的开发环境中的，其某些部分可能要修改，使原本泛化的构件特化。可见在建造构件时，应该提供构件的特化和调整机制。例如，在构件被复用时可能发生变化的相应位置上，可以标识变化点（variation point），同时为变化点附加对应的变体（variant）；每个变化点与变体可与相应的文档关联，让文档解释如何使用它以及如何选择变体。这样，当该构件被复用时，即可根据不同的应用指定不同的变体，使抽象变得具体，以适应特定应用的需要。

③ 易组装性。构件通常存放在构件库中。当开发一个软件时，首先从构件库中检索到若干合适的构件，经过特化再进行组装。无论是同构件的组装，即具有相同软、硬件运行平台的构件之间的组装，还是异构件的组装，即具有不同软、硬件运行平台的构件之间的组装，这些构件都应具有良好的封装性和良好定义的接口，构件之间应具有松散的耦合度，同时还应提供便于组装的机制。

图 10.4 显示的基于面向对象技术的复用（reuse based on objected-oriented technology, REBOOT）构件模型，对可复用构件的一般特征作了较全面的描述。

2. 领域构件的特征

除满足单个构件的特征外，还需考虑应用领域的特征，使之与应用领域的设计框架相适应。

Binder 建议，这时应考虑的关键问题主要包括：

① 标准数据。通过对应用领域的研究，标识出标准的全局数据结构（如文件结构和完整的数据库），使所有设计的构件都可以用这些标准数据结构来刻画。

② 标准接口协议。建立 3 个层次的接口协议，即构件内接口、构件外接口和人机接口。

③ 程序模板。成形的结构模型，可以用作新程序的体系结构设计模板。

一旦建立了应用领域的标准数据、标准接口协议和程序模板，设计者就有了一个可以在其中进行领域设计的框架。符合这个框架设计出来的新构件，以后在该领域的复用中将获得更高的复用概率。

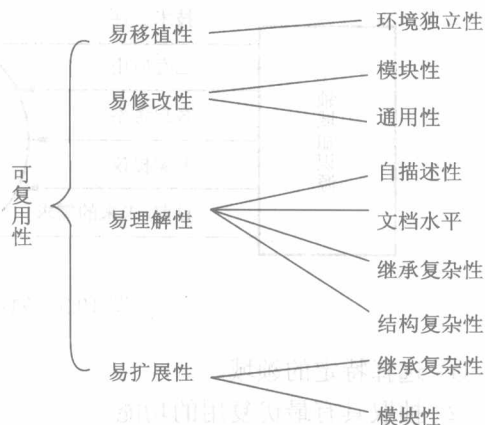


图 10.4 REBOOT 通用模型

3. 几种流行的典型构件技术

为了方便构件之间的集成和装配,必须有一个统一的标准。经过几年的发展,已经提出了多种构件的模型及规范,形成了一些较有影响的构件技术,其中有微软公司的 COM/OLE、对象管理组织(OMG)的跨平台的开放标准 CORBA 以及 OpenDoc 等。这些技术的流行为构件提供了实现标准,也为构件的集成和组装提供了很好的技术支持。

(1) COM/OLE

COM (Component Object Model) 是 Microsoft 开发的一个构件对象模型,它提供了对在单个应用中使用不同厂商生产的对象的规约。OLE (Object Linking and Embedding) 是 COM 的一部分,由于 OLE 已成为微软操作系统的一部分,因此目前应用最为广泛。最早的组件连接技术 OLE 1.0 是 Microsoft 公司于 1990 年 11 月在 COMDEX 展览会上推出的,它给出了软件构件的接口标准。任何人都可以按此标准独立地开发组件和增值组件(指在组件上添加一些功能构成的新组件),或由若干组件组建集成软件。在这种软件开发方法中,应用系统的开发人员可在组件市场上购买所需的大部分组件,因而可以把主要精力放在应用系统本身的研究上。

(2) CORBA

CORBA (Common Object Request Broker Architecture) 是由对象管理组织于 1991 年发布的一种基于分布对象技术的公共对象请求代理结构,其目的是在分布式环境下,建立一个基于对象技术的体系结构和一组规范,实现应用的集成,使基于对象的软件组件在分布异构环境中可以复用、移植和互操作。

CORBA 是一种集成技术,而不是编程技术。它提供了对各种功能模块进行构件化处理,并将它们捆绑在一起的黏合剂。一个对象请求代理提供一系列服务,它们使可复用构件能够和其他构件通信,而不管它们在系统中的位置。当用 CORBA 标准建立构件时,这些构件在某一系统内的集成就可以得到保证。加上基于 CORBA 规范的应用屏蔽了平台语言和厂商的信息,使得对象在异构环境中也能透明地通信。对于 CORBA 定义的通用对象服务和公共设施,用户还可以结合其特殊需求来构造应用对象服务,以提供企业应用级的中间件服务系统。

(3) OpenDoc

OpenDoc (开放式文档接口) 是 1995 年 3 月由 IBM、Apple 和 Novell 等公司组成的联盟推出的一个关于复合文档和构件软件的标准。该标准定义了为使不同的开发者提供的构件之间能够相互操作而必须实现的服务、控制基础设施和体系结构。由于 OpenDoc 的编程接口比 OLE 小,因此 OpenDoc 的应用程序能与 OLE 兼容。

10.2.4 建立可复用构件库

一个可复用构件库中存放着成千上万的可复用构件,要想方便地选择和使用该库中的可复用构件,要求对构件库有严密的管理、有效的分类和科学的描述。目前最常用的有 3 种分类模式。

1. 枚举分类 (enumerator classification)

通过定义一个层次结构来描述构件, 在该层次中定义软件构件的类以及不同层次的子类。枚举分类的优点在于易理解和使用, 这是因为对每个层次中都进行了详细的定义。但它的缺点也在于此, 它要求在进行领域分析时, 对每个层次都给予足够的信息, 导致了成本的提高。

2. 刻面分类 (faceted classification)

对领域进行分析后, 可标识出一组基本的描述特征, 这些特征就称为刻面 (意为呈现于用户面前的一个方面)。刻面可以描述构件的功能、被操作的数据、构件应用的语境等各种特征, 并根据其重要性区分优先次序并被联系到构件。通常一个构件的刻面描述被限定不超过 7 或 8 个刻面。

每个刻面可含有一个或多个值, 这些值一般是描述性关键词。通过关键词可以很容易地查找到所需的构件。

刻面描述具有较大的灵活性, 因为可以随时增加新的刻面值, 同枚举分类相比, 更易于扩展和进行适应性修改。

3. 属性-值分类 (attribute-value classification)

为领域中的所有构件定义一组属性, 然后赋给这组属性一组值。通过查询相应属性的值, 找出所需构件。

属性-值分类与刻面分类具有相似处。但也有不同:

- ① 属性-值分类对可使用的属性数量没有限制, 而刻面分类被限定不超过 7 或 8 个刻面。
- ② 属性-值分类中的属性没有优先级, 而刻面可以区分优先级。
- ③ 属性-值分类不具有同义词功能, 而刻面分类可以查找相关的技术同义词。

构件库不仅要选择适当的分类方法, 还需有方便的环境支撑。一个好的可使用复用构件的环境, 通常应包括以下这些内容:

- ① 存放描述构件的分类信息的数据库。
- ② 该数据库的管理系统。
- ③ 允许用户查找构件的检索系统。
- ④ 可以自动将构件加入新系统中的 CASE 工具。

10.3 基于构件的软件开发

硬件集成、独立的特点, 使计算机的拼装工作变得非常简单。使用芯片的用户不必了解芯片本身的构造, 只需知道它的功能描述 (知道它是做什么的), 然后插在对应的地方就行了。这使人们希望软件系统的开发工作也能如此轻松——把相关的构件 (已有的或经过一些修改的) 像搭积木一样搭起来, 使开发者可以把精力集中在当前问题的解决上, 而不是一切从头开始。

应运而生的 CBSD 彻底改变了软件的生产方式,从根本上提高了软件生产的效率和质量,提高了开发大型软件系统的成功率。

10.3.1 构件集成模型

在第 2 章中讨论过构件集成模型构造应用软件系统的流程(如图 2.6 所示)。这是一个针对复用的过程模型,从中可以看到如何利用软件构件迭代地进行产品的开发与发布。由于构件库中的构件都是用一定的规律进行分类的,在 CBSD 开发中,用户可首先通过相应的查询工具,在构件库中按照匹配原理查找满足自己需求的构件(当然这是以使用者对被复用对象的理解为基础的),如果找到,则将其复合组装到新的应用程序中,此过程可能要进行适应性修改。图 10.5 显示了把库中的构件集成到一个应用系统中的过程。



图 10.5 把库中的构件集成到应用系统中

10.3.2 应用系统工程

在 CBSD 中,通过复用构件系统开发单个应用系统的构件工程,称为应用系统工程(application system engineering, ASE)。ASE 过程的实质是从一个或多个构件系统中选择构件进行特化,最后把构件装配成应用系统。

以下是 ASE 过程的基本步骤。

1. ASE1: 获取需求

- ① 由客户开始,通过复用例构件构建应用系统的用例模型。
- ② 主要从客户和最终用户那里收集输入,还可以从业务模型和系统模型中收集输入。
- ③ 评价每个用例的重要性和成本。

2. ASE2: 分析

把用例模型作为输入来组装和特化设计构件。

3. ASE3: 设计应用系统

把分析模型作为输入来组装和特化设计构件。

4. ASE4: 实现应用系统

把设计模型作为输入来组装和特化实现构件。

5. ASE5: 测试应用系统

组装和特化测试构件,然后测试应用系统。

6. ASE6: 应用系统打包

应用系统打包, 以便提供给制造商、安装者或最终用户使用。

由此可见, 一个支持复用的软件开发过程包括如下活动: 检查领域模型和可复用构件; 收集和分析最终用户的需求; 设计并实现需要附加的构件; 对所提供的可复用构件进行必要的修改; 建造和测试整个应用软件等。

还需指出, CBSD 的实施不仅需要技术的支持, 还需要有机构的保证。在以往的软件开发机构中, 通常以项目为中心进行管理, 项目经理都把注意力集中在自己负责的单个项目上, 很少有人考虑建造可共享的可复用构件, 每个项目都从头做起, 因而造成资源的极大浪费。所以在具体实施软件复用时, 还需重组软件开发组织的结构, 并对软件开发过程进行调整。

10.4 面向对象与软件复用

面向对象技术开始于 20 世纪 60 年代后期, 但到 20 世纪 90 年代才获得广泛应用。其重要原因之一, 就是面向对象技术十分有利于软件复用; 反过来, 软件复用技术也对面向对象软件的开发提供了有力支持, 二者相辅相成, 互相促进。以面向对象技术中的类为例, 它具有封装、继承、多态等 3 个重要特征。封装使数据结构和对它们的操作被合并单个名称内, 有利于构件复用; 继承使子类继承其父类的所有属性和操作, 形成最直接的复用; 而多态使得一系列不同的操作具有相同的名称, 且对象之间相互独立, 更利于构件复用。

10.4.1 OO 方法对软件复用的支持

在 OOA 阶段, 面向对象开发就把软件复用作为考虑的重点。运用 OOA 定义的对象类, 就具有可复用构件的许多特征。OOA 的分析结果, 是对问题空间的良好映射, 使同类系统的开发者很容易从问题出发, 在已有的 OOA 结果中发现不同粒度的可复用构件。

以下从 8 个方面, 说明 OO 方法对软件复用的支持。

1. OOA 模型

OOA 建立的系统模型分为基本模型和补充模型, 基本模型只表示最重要的系统建模信息, 而细节的信息则在详细说明中给出。这种表示方法, 使 OOA 的基本模型体现了更高的抽象, 更容易作为可复用的系统构架。当这种构架在不同的应用系统中复用时, 往往可通过不同的详细说明体现系统之间的差异, 因此对系统构件的改动较少。

2. OOA 与 OOD 的分工

OOA 只注重与问题域及系统责任有关的信息, OOD 则需考虑与实现条件有关的因素。这种分工使 OOA 模型独立于具体的实现条件, 从而使分析结果可以在问题空间及系统责任相同而实现条件互异的多个系统中复用, 并为从同一领域的多个系统的分析模型中提炼领域模型创造了有利条件。

3. 对象的表示

所有的对象都用类作为其抽象描述。对象的一切信息，包括对象的属性、行为及其对外关系等，都是通过类/对象来表示的。类作为一种可复用构件，在应用于不同系统时，不会出现因该类对象实例不同而使系统模型有所不同的情况。

4. 一般-特殊结构

面向对象方法引入了对一般-特殊结构中多态性的表示法，从而增强了类的可复用性。通过对多态性的表示，使一个类可以在需求相似但不完全相同的系统中被复用。

5. 整体-部分结构

把部分类作为可复用构件在整个类中使用，其原理与在特殊类中使用一般类是一致的，但在某些情况下，对问题空间的映射比通过继承实现复用显得更为自然。此外，还可通过整体-部分结构来支持领域复用的策略，即从整体对象中分离出一组可在领域范围内复用的属性与服务，定义为部分对象，使之成为领域复用构件。

6. 实例连接

建议用简单的二元关系表示各种复杂关系和多元关系。这一策略使构成系统的基本成分（类/对象）以及它们之间的关系在表示形式和实现技术上都可以做到规范和一致，而这种规范性和一致性对于可复用构件的组织、管理和使用，都是很有益的。

7. 类描述模板

类描述模板作为 OOA 详细说明的主要成分，对于对象之间关系的描述注意使用者与被使用者的区别，仅在使用者一端给出类之间关系的描述信息。这说明可复用构件之间的依赖关系不是对等的。因此，在继承、聚集、实例连接及消息连接等关系的使用者一端描述这些关系，有利于这些关系信息和由它们指出的被依赖成分的同时复用。而在被使用者一端不描述这些关系，避免了因复用场合的不同所引起的修改。

8. 用例

由于用例是对用户需求的一种规范化描述，因此它比普通形式的需求文档具有更强的可复用性。每个用例是对一个角色使用系统某项功能时与系统进行交互活动所做的描述，它具有完整性和一定的独立性，因此很适于作为可复用构件。

10.4.2 复用技术对 OO 方法的支持

复用技术对 OO 方法的支持，可以从以下 5 个方面来说明。

1. 类库

在面向对象的软件开发中，类库是实现对象类复用的基本条件。人们已经开发了许多基于各种面向对象语言的编程类库，有力地支持了源程序级的软件复用，但要在更高的级别上实现软件复用，仅有编程类库是不够的。实现 OOA 结果和 OOD 结果的复用，必须有分析类库和设计类库的支持。为了更好地支持多个级别的软件复用，可以在 OOA 类库、OOD 类库和 OOP 类库之间建立各个类在不同开发阶段的对应与演化关系。即建立一种线索，表明每个

OOA 的类对应着哪个(或哪些)OOD 类,以及每个 OOD 类对应着各种 OO 编程语言类库中的哪个(或哪些)OOP 类。

2. 构件库

类库可以看作一种特殊的可复用构件库,它为在面向对象的软件开发中实现软件复用提供了一种基本的支持。但类库只能存储和管理以类为单位的可复用构件,不能保存其他形式的构件;然而它可以更多地保持类构件之间的结构与连接关系。构件库中的可复用构件,既可以是类,也可以是其他系统单位;其组织方式,可以不考虑对象类特有的各种关系,只按一般的构件描述、分类及检索方法进行组织。在面向对象的软件开发中,可以提炼比类/对象粒度更大的可复用构件,例如把某些结构或某些主题作为可复用构件;也可以提炼其他形式的构件,例如用例图或交互图。在这些构件库中,构件的形式及内容比类库更丰富,可为面向对象的软件开发提供更强大的支持。

3. 构架库

如果在某个应用领域中已经运用 OOA 技术建立一个或几个系统的 OOA 模型,则每个 OOA 模型都应该保存起来,为该领域中新系统的开发提供参考。当一个领域已有多个 OOA 模型时,便可以通过进一步抽象而产生一个可复用的软件构架。形成这种可复用软件构架的更正规的途径是开展领域分析。通过正规的领域分析获得的软件构架将更准确地反映一个领域中各个应用系统的共性,它在 OO 开发中具有很强的可复用价值。

4. 工具

有效地实行软件复用需要有一些支持复用的软件工具,包括类库或构件/构架库的管理、维护与浏览工具、构件提取与描述工具以及构件检索工具等。这些工具对 OOA/OOD 过程也具有很强的支持功能,例如,从类库或构件/构架库中寻找可复用构件;对构件进行修改,并加入当前的系统模型;把当前系统开发中新定义的类(或其他构件)提交到类库(或构件库),等等。

5. OOA 过程

在复用技术支持下的 OOA 过程,可以按两种策略进行组织。第一种策略是,基本保持某种 OOA 方法所建议的 OOA 过程原貌,在此基础上对其中的各个活动引入复用技术的支持;另一种策略是重新组织 OOA 过程。

第一种策略是在原有的 OOA 过程基础上增加复用技术的支持,应补充说明的一点是,复用技术支持下的 OOA 过程应增加一个提交新构件的活动。即在一个具体应用系统的开发中,如果定义了一些有希望被其他系统复用的构件,则应该把它提交到可复用构件库中。第二种策略的前提是,在对一个系统进行面向对象的分析之前,已经用面向对象方法对该系统所属的领域进行过领域分析,得到了一个用面向对象方法表示的领域构架和一批类构件,并且具有构件/构架库、类库及相应工具的支持。在这种条件下,重新考虑 OOA 过程中各个活动的内容及活动之间的关系,力求以组装的方式产生 OOA 模型,将使 OOA 过程更为合理。

10.4.3 基于构件软件开发的现状与问题

综上所述,面向对象技术和软件复用技术互相支持,互相促进,确实起到了相辅相成的作用。但是,早在1984年4月,在日本京都召开的基于构件的软件开发(CBSD)国际专题学术会议上就已指出,对于CBSD而言,对象技术并不是必需的,同时仅仅依靠对象技术也不能实现CBSD。很多研究软件工程的专家认为,对象技术仅仅是CBSD的开始,但就对象技术本身而言,并不能全面地表述CBSD所需的抽象概念,而且即使脱离对象技术,CBSD也完全可以实现。总而言之,对于CBSD而言,对象技术既不是必需的,仅有对象技术也是不够的。

具体地说,CBSD将导致使用对象技术的系统设计方法、项目管理方法和组织形式发生实质性的变革;但如果将构件看作是一个可替换单元时,单纯的对象技术就不够了。这是因为,构件的各种定义中都或多或少地强调了构件的一个特性——对上下文的依赖性,但是对象技术却根本不支持构件的这种特性,这就不利于进行设计层的抽象,特别是在试图使用已有的构件进行集成时,经常会遇到麻烦。

作为第三代软件工程的核心,软件复用现在面临着各方面的困难。无论是技术问题还是非技术问题,都影响到软件复用的广泛实施。具体而言,大粒度的可复用软件制品其抽象一定是复杂的,为了使用这些制品,软件开发者必须提前熟悉这些抽象,并且花时间去研究和理解它。例如,数学概念库的开发者要熟悉数学库的使用及抽象数据类型模型;语义的开发者要熟悉栈和队列的使用;某个特定应用领域的开发者要熟悉所使用领域特有的应用生成程序和领域语言等。

另外,软件产品是一种精神产品,它的产生几乎完全是人脑思维的结果。它的价值几乎完全在于其中所凝结的思想,其物质载体的制造过程与价值含量都是微不足道的。物质产品的生产只受到人类制造能力的限制,现有的一切物质产品的复杂性都没有超过这种限度。而软件却没有这种限制,只要人的大脑能想到的问题,都可能要求软件去解决。由于人脑所能思考的问题的复杂性远远超出了人类能制造的物质产品的复杂性,因而使软件的复用更为困难。

小 结

软件复用是在软件开发中避免重复劳动的解决方案。通过软件复用,可以提高软件开发的效率和质量。传统软件工程经过20世纪70年代到80年代的发展,已基本上达到成熟。20世纪90年代以后,面向对象技术和软件复用技术相继出现,它们互相促进,被人们分别称为第二代和第三代软件工程。现在,软件复用已被许多人视为解决软件危机、提高软件生产效率和质量的充满希望的途径。

实现软件复用的关键因素是软件构件技术。软件构件技术是基于面向对象发展起来的,

但它却摆脱了面向对象理论的束缚。尽管在理论上还未完备，但实际应用进展很快。软件构件技术目前还处于发展阶段，迫切需要解决以下问题：针对如何开发和应用，需要有一套开发规范和质量保证体系；如何提取领域构件，也仍然处于摸索阶段。

习 题

1. 什么是软件复用？为什么要复用软件？
2. 软件复用与软件共享及软件移植各有什么不同？试简单说明。
3. 怎样理解“开发伴随复用，开发为了复用”的含义？
4. 什么是领域和领域工程？它们和软件复用有什么关系？
5. 简述纵向复用领域工程的主要活动内容。
6. 什么是构件？
7. 当前流行的构件技术有哪几种？
8. 可复用构件库有哪几种常见的分类模式？
9. 什么是 CBSD？实施软件构件技术要解决哪些問題？
10. 软件复用当前面临的困难有哪些？

附 录

附录A 软件复用与软件共享的区别
附录B 软件复用与软件移植的区别
附录C 软件复用与软件共享的区别
附录D 软件复用与软件移植的区别
附录E 软件复用与软件共享的区别
附录F 软件复用与软件移植的区别
附录G 软件复用与软件共享的区别
附录H 软件复用与软件移植的区别
附录I 软件复用与软件共享的区别
附录J 软件复用与软件移植的区别

第11章 软件工程管理

技术和管理，是软件生产中不可缺少的两个方面。对技术而言，管理意味着决策和支持。只有对生产过程进行科学的管理，做到技术落实、组织落实和费用落实，才能达到提高生产率、改善产品质量的目的。国外的经验表明，有不少项目因管理不善，造成费用超支 2~3 倍，开发周期延长一倍或更长的严重后果。在对失败软件的原因分析中，因管理不善造成失败的项目竟占总失败项目的半数以上。

本章将从整个软件生存周期着眼，对工程化生产中的管理工作进行一次比较全面的介绍。重点是讨论软件管理的主要内容以及实现这些管理的方法和使用的技术。

11.1 管理的目的与内容

简而言之，管理的目的是为了按照预定的时间和费用，成功地完成软件的计划、开发和维护任务。软件管理主要体现在软件的项目管理中，包括对于费用、质量、人员、进度等 4 个方面的管理。前已指出（见 2.6 节），在开发软件项目前，首先需制定“项目实施计划”，以便按照计划的内容组织与实施软件的工程化生产。项目管理的最终目标，就是要以合理的费用和进度，圆满完成计划所规定的软件项目。它是费用、质量、人员和进度等管理在一个项目上的综合体现。

1. 费用管理

目的是对软件开发进行成本核算，使软件生产按照商品生产的经济规律办事。其主要任务是：以简单实用和科学的方法估算出软件的开发费用，作为签订开发合同的根据；管理开发费用的有效使用，用经济手段来保证产品如期按质完成。

2. 质量管理

目的在于保证软件产品（包括最终程序和文档）的质量。为了贯彻全面质量控制（TQC）的原则，每个项目都要制定“质量保证计划”，并由专设的质量保证小组负责贯彻，确保在各个阶段的开发和维护工作全都按软件工程的规范进行。鉴于质量管理的重要性，第 12 章将详细讨论。

配置管理是质量管理的重要组成部分，包括对于程序、文档和数据的各种版本所进行的管理，以保证资料的完整性与一致性。对于重要的和大型的软件，需要制定单独的“配置管理计划”，并由专设的人员进行管理。读者可参看第 9.5 节的有关内容。

3. 项目管理

除项目经费和软件质量外，项目的进度和人员也是项目管理的重要内容。为了按时完成进度，项目经理和下属的子项负责人应制定详细的工作计划，用网络图（详见 11.5 节）描述各部分工作进度的相互关系。对各个开发阶段所需的人力资源，也要分类做出估算，写入项目实施计划。此外，在项目计划的实施过程中，项目经理或开发组长应定期填写“项目进展（月/季）报表”，并在需要时对进度进行适当的调整。开发结束后，还应写出“项目开发总结”。报表和总结的内容请读者参阅有关的规范手册。

11.2 软件估算模型

数据是一切管理工作的基础。心中有数或心中无数，管理的效果将大不相同。E. Yourdon 说过，“对一个软件开发项目进行管理的唯一有效方法，就是对开发过程中发生的一切进行监控与度量。” T. DeMarco 也说，“你不能够管理你无法度量的事物，不进行度量的事物是控制不住的。”

需要指出，估算在软件度量中占有重要的地位。一般地说，在软件开发之后可进行度量，但在软件开发之前只能进行估算。如果说度量是科学，则估算既是科学，又是艺术。它需要管理工作者的经验和勇气。

11.2.1 资源估算模型

为了对软件开发进行度量，人们提出了软件的各种估算模型（estimation model）。其中有一类称为资源模型（resource model），用来估算软件在开发中花费的资源，如开发时间、开发人数以及用人-月或人-年计算的工作量等。

大多数资源模型是根据过去的经验，通过大量的统计和分析推导出来的。它们揭示了在一定条件下资源花费和软件规模之间的内在关系。但是，估算资源花费涉及许多可变的因素。以开发工作量为例，它与给定的开发时间、开发人员的能力、产品的规模和复杂度、要求达到的可靠性等级以及开发的环境和工具等因素都有直接的关系。很难找出一种适用于各类应用领域与开发环境的资源模型。下面将介绍几种较典型的资源模型。

1. 静态单变量资源模型

这种模型在计算软件开发的资源花费时，只需要设定被开发软件的一种参数，故称为单变量型。它的一般形式是：

$$\text{资源} = c_1 \times (\text{估计的软件特征})^{c_2}$$

其中，资源可以是开发工作量 E （单位为人-月）、开发时间 T （单位为月）或开发人数 P （单位为人）等，估计的软件特征可以是源程序长度 L （单位为千行）或软件的开发工作量 E ， c_1 和 c_2 是依赖于开发环境和软件应用领域的两个经验常数。

自 1973 年至 1977 年, Walston 与 Felix 从 60 个软件项目(源程序长度为 4~467 千行, 工作量为 12~11 758 人-月, 使用了 28 种不同的高级语言)的统计中导出了以下的一组参数方程:

$$E=5.1L^{0.91}$$

$$T=4.1L^{0.36}$$

$$T=2.47E^{0.35}$$

$$P=0.54E^{0.6}$$

$$\text{文档长度}=49L^{1.01}$$

这组方程在计算 E 、 T 、 P 等开发所需的资源时, 使用 L 作为软件的估计特征。由 L 计算开发时间、工作量或文档页数, 再根据算出的工作量 E 来计算开发时间与所需人数。

1977 年, D. L. Doty 等人发表了为美国空军进行的“软件成本估算研究”报告, 它使用的也是单变量资源模型。报告还发表了分别适用于不同语言(高级或汇编语言)和不同应用领域(指挥控制、科学计算、事务处理以及实用程序)的一组参数方程。

这类模型简单易懂, 常数 c_1 与 c_2 可以从历史数据导出。但如没有适用于本单位情况的经验常数, 就不能直接搬用。

2. Putnam 资源模型

Putnam 模型是一种多变量资源模型, 可以用下面的方程式来表示:

$$L=cK^{1/3}T^{4/3}$$

$$\text{或 } K=L^3/(c^3T^4)$$

其中 L 与 T 仍分别代表源程序长度(单位为行)和开发时间(单位为年)。 K 表示全生存周期(含维护在内)所需要的工作量(单位为人-年)。对大型软件而言, 其大小约为开发工作量 E 的 2.5 倍, 即 $E=0.4K$ 。 c 是一个与开发环境有关的常数。对优良、好与不好的 3 种环境, c 的典型值可分别取 12 500、10 000 与 6 500。

这个模型发表于 1978 年, 是 L. Putnam 对 50 个大型军用软件(生存周期工作量均在 30 人-年以上)研究得出的结果。后来又对另 150 个大型软件进行验证, 也取得满意的效果。一般认为, 规模在 10 万行以上的软件, 用这个模型是适当的。

Putnam 模型特点, 是在同一个模型中给出了 K (或 E)、 L 和 T 三者之间的关系。例如, 给定了 L 和 T , 就可以用它来估计开发所需的工作量 E 。如果估计的程序长度有一个范围(例如从 $L_1 \sim L_2$), 则在保持工作量不变的情况下, 可算出相应的开发时间 T_1 与 T_2 , 等等。Putnam 还开设了一个称为 SLIM 的软件工具, 能根据这一模型自动计算开发所需的资源, 使应用这一模型更加方便。

Putnam 模型方程揭示了 E 与 T 之间的关系。根据这一方程, 开发工作量 E 与开发时间 T 的 4 次方成反比。这表明, 开发时间的微小变化, 会引起开发工作量相当大的变化。如果把开发时间成倍延长, 则一个原来需要 100 人-月的项目, 能够把工作量降低到仅需 6.25 人-月

($=100/2^4$)。有人怀疑这个结论的可用性,认为 Putnam 模型对时间的变化可能过分敏感了。根据 Boehm 等人的研究,任何软件的开发时间都有一个最佳值。在最佳值附近增加或缩短开发时间,都只会使工作量增大。但无论如何,人们普遍赞成 Putnam 所提出的警告:压缩软件项目的开发时间,意味着显著增加项目的开发工作量。

Putnam 模型是建立在 Rayleigh-Norden 曲线(见第 11.4 节)的基础上的。后者可用来计算生存周期内各个阶段所需要的人力。所以 Putnam 模型有时也称为资源调度模型(resource deployment model)或动态多变量资源模型。

11.2.2 COCOMO 模型

1981 年,Boehm 在他的名著《软件工程经济学》一书中,详细介绍了他提出的“构造性成本模型”(constructive cost model),简称 COCOMO 模型。它以静态单变量模型为基础,但在下列两个方面作了较大的改进:

① 按照软件的应用领域和复杂程度,将它们分为组织(organic)、半独立(semidetached)和嵌入(embedded)3 种类型,每类分别使用一组不同的模型方程,如表 11.1 中由上向下,程序的复杂度逐步提高, E 和 T 的计算值也随之增大。

表 11.1 不同类型软件的 COCOMO 模型

软件类别	模型方程	适用范围
组织型	$E=3.2L^{1.05}$ $T=2.5E^{0.38}$	高级语言应用程序,如科学计算、数据处理、企业管理程序等
半独立型	$E=3.0L^{1.12}$ $T=2.5E^{0.35}$	大多数实用程序,如编辑程序、连接程序、编辑程序等
嵌入型	$E=2.8L^{1.20}$ $T=2.5E^{0.32}$	与硬件关系密切的系统程序,如操作系统、数据库管理系统、实时处理与控制程序等

② 在模型中增加一个工作量调节因子(effort adjustment factor, EAF),反映各种有关因素对软件开发的影响。Boehm 把这些因素归结为 4 类、15 种因子,如表 11.2 所示。

表 11.2 调节因子和它的值范围

属性	调节因子	调节值范围	例 11.1 中使用的值
产品属性	要求的可靠性等级	0.75~1.40	1.00
	数据库规模	0.94~1.16	0.94
	产品复杂度	0.70~1.65	1.30
计算机属性	对程序执行时间的约束	1.00~1.66	1.11
	对程序占用存储容量的约束	1.0~1.56	1.06
	开发环境的变动	0.87~1.30	1.00
	开发环境的响应时间	0.87~1.15	1.00

续表

属性	调节因子	调节值范围	例 11.1 中使用的值
人员属性	分析员水平	1.46~0.71	0.86
	程序员水平	1.42~0.70	0.86
	对应用领域的熟悉程度	1.29~0.82	1.00
	对开发环境的熟悉程度	1.21~0.90	1.10
	对所用语言的熟悉程度	1.14~0.95	1.00
项目属性	开发方法的现代化	1.24~0.82	0.91
	软件工具的数/质量	1.24~0.83	1.10
	完成时间的限制	1.23~1.10	1.00

每种因子的标称调节值为 1，可根据实际情况在一定范围内上下浮动。模型中使用的调节因子值，就是这 15 种因子的值的乘积，可以写作

$$EAF = \prod_{i=1}^{15} F_i \quad (i=1, 2, \dots, 15)$$

为了便于说明，下面引用 Boehm 书中的一个例子。

【例 11.1】 假定要在微处理器上开发一个嵌入型的电信处理程序，程序规模为 10 000 行。试计算所需的工作量与开发时间。

【解】 采用表 11.1 中第 3 类的方程组，可得

$$E = 2.8 \times 10^{1.20} = 44.4 \text{ (人-月)}$$

$$T = 2.5 \times 44.4^{0.32} = 8.4 \text{ (月)}$$

对表 11.2 中的 15 种调节因子逐项研究，假定得出了如表 11.2 最右一列显示的值。将这 15 个值相乘，可得

$$EAF = 1.00 \times 0.94 \times \dots \times 1.10 \times 1.00 \approx 1.17$$

用这个调节因子来修正上面的 E 和 T ，便得到以下修正值：

$$E' = 44.4 \times 1.17 = 51.9 \text{ (人-月)}$$

$$T' = 8.4 \times 1.17 = 9.8 \text{ (月)}$$

调节因子的引入，使我们有可能对不同的开发条件进行定量的比较。这里仍引用 Boehm 所举的例子：

① 换用水平较低的开发人员。本例中原来使用较高水平的分析员和程序员，一个人-月的花费是 6 000 美元。如果换用 5 000 美元/人-月的人员，则人员水平的两个调节因子均将从 0.86 上升为 1.00，整个 EAF 值将从原来的 1.17 变成 $1.17/0.86/0.86 \approx 1.58$ 。开发成本不仅没有节省，反比原来有所增长。请看下面的计算：

原开发成本：6 000 × 44.4 × 1.17 = 311 688

新开发成本：5 000 × 44.4 × 1.58 = 350 760

② 扩充内存存储器容量。假定原有的内存容量为 64 KB，允许软件使用的内存为 46 KB。

现决定增加 10 000 美元购买内存扩充板，使软件可用的内存从 46 KB 扩充为 96 KB。这样，内存容量的调节因子值可从原来的 1.06（见表 11.2 最右一列第 5 项）下降为 1.00，从而使 EAF 从 1.17 下降为 1.10。虽然新增加 10 000 美元的投资，总开发成本仍可望下降。以下是新成本的算式：

$$6\,000 \times 44.4 \times 1.10 + 10\,000 = 303\,040$$

还需说明，Boehm 把 COCOMO 模型分为基本、中间和详细 3 个等级。以上介绍的是它的中间模型。基本模型不用 EAF，仅用于对成本作粗略的估算。中间模型是从整个生风存期来衡量 EAF 的影响。而详细模型需要考虑各个调节因子对于不同开发阶段的影响。需要深入了解的读者，请参阅 Boehm 的原著。

Boehm 用中间和详细级的 COCOMO 模型估算了 63 个软件开发项目（包括商业、控制、科学、人机以及支持和系统软件）的资源花费。与实际结果相比，工作量（或成本）误差不大于 20%，开发时间误差不大于 70%。Boehm 认为，目前资源模型能够达到这一“合理的精度”，已经很不错了。

以上介绍了 3 种典型的资源模型。建立模型的目的，是为了找到一种方便而又合理地计算工作量和开发时间的方法。但是，正如 Boehm 所指出的，“要做的事情，比仅仅将数字代入公式并得出结果多得多”，即使有了适用的模型，也“需要使用其他方法来校验模型计算的结果”。在余下各节中，将继续向读者介绍在实际开发中常用的对成本、人员进行估算的方法。

11.3 软件成本估计

成本估计是软件费用管理的核心，也是软件工程管理中最困难、最易出错的问题之一。上节介绍的资源模型，仅是估计成本的一种手段。1974 年 Wolverton 就把成本估计方法分为 5 种，在 Boehm 的著作中，进一步把它们分为 7 种，如表 11.3 所示。为了帮助读者从众多的方法中把握要领，本书把主要的成本估计方法归并为自顶向下估计、由底向上估计和算法模型（algorithmic model）估计 3 类，下文将依次介绍，然后讨论并举例说明。

表 11.3 成本估计使用的方法

R.W.Wolverton	B.W.Boehm
自顶向下估计	自顶向下估计
由底向上估计	由底向上估计
相似与差异估计法	类比估计
比率估计法	专家判断
标准值估计法	算法模型估计
	Parkinson 法
	削价取胜法

1. 自顶向下成本估计

这类方法着眼于软件的整体。根据被开发项目的整体特性，首先估算出总的开发成本，然后在项目内部进行成本分配。因这类估计通常仅由少数上层（技术与管理）人员参加，所以属于“专家判断”的性质。这些专家依靠从前的经验，把将要开发的软件与过去开发过的软件进行类比，借以估计新的开发所需要的工作量和成本。

自顶向下估计的缺点是，对开发中某些局部的问题或特殊困难容易低估，甚至没有考虑。如果所开发的软件缺乏可以借鉴的经验，在估计时就可能出现较大的误差。

当参加估计的专家人数较多时，可采用特尔斐（Delphi）法来汇集他们的意见。特尔斐法的传统做法是：把系统定义文件或规格说明书发给各位专家，各自单独进行成本估计，填入成本估计表，如图 11.1 所示。然后由协调人综合专家意见、摘要通知大家，并开始新一轮估计，这种估计要反复多次，直到专家们的意见接近一致为止。

项目名： <u>XXX</u>		日期： <u>1/20/02</u>
以下是第1轮的估计值：		
你的估计值	平均估计值	
⊗	⊠	
你对下轮的估计值是 <u>35</u> 人-月		
你的理由是： _____		

图 11.1 特尔斐成本估计表

特尔斐法的指导思想是：首先用各自填表代替相互讨论。这样既避免了对立意见的直接交锋，又可保持各专家独立发表意见；其次对个别专家的不同估计，协调人应单独与之讨论，并请他说明理由。也有人主张，对于重大的分歧，必要时可召集专家们开会讨论，但不要公开对立双方的姓名。

2. 由底向上成本估计

与自顶向下估计相反，由底向上估计不是从整体开始，而是从一个个任务单元开始。其具体做法是，先将开发任务分解为许多子任务，子任务又分成子子任务，直到每一任务单元的内容都足够明确为止。然后把各个任务单元的成本估计出来，汇合成项目的总成本。由于任务单元的成本可交给各该任务的开发人员去估计，得出的结果常比较实际。所以，如果说自顶向下估计是走的“专家路线”，则由底向上是“群众路线”。

这种方法也有缺点。由于具体工作人员往往只注意到自己范围内的工作，对综合测试、

质量管理和项目管理等涉及全局的花费可能估计不足，甚至完全忽视。因此，就有可能使成本估计偏低。

利用系统的分类活动结构图（work breakdown structure, WBS）可有效地避免在估计中遗漏任务。WBS 图既可用来表示一个项目的产品，又能表示项目的开发过程。图 11.2 与图 11.3 显示了一个计算机辅助设计系统（CAD 系统）的两种 WBS 图。它们把一个项目开发任务的各项工作全都在图中表示出来，在估计成本时可以一览无遗。

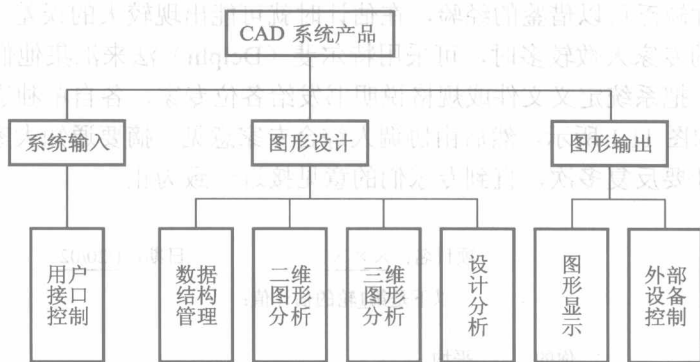


图 11.2 CAD 系统的产品 WBS 图

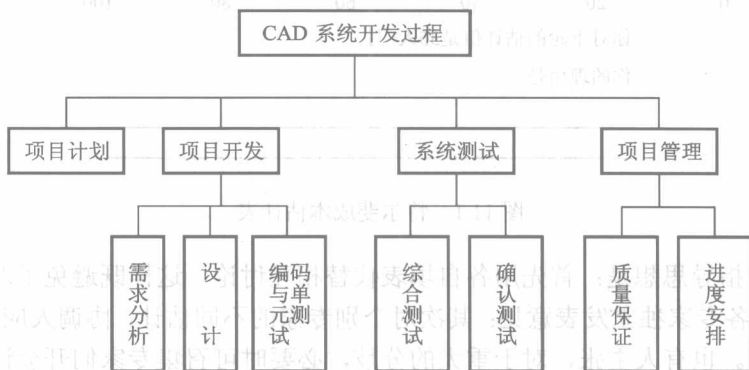


图 11.3 CAD 系统开发过程的 WBS 图

3. 算法模型估计

算法模型就是资源模型，是成本估计的又一有效工具。由于任何资源模型都是根据历史数据导出的，所以比较客观，计算结果的重复性也好（即不论什么时候使用模型，都能得出同样的结果）。像 COCOMO 这样带有调节因子的模型，还能够灵敏地反映因开发条件改变而引起的成本变化，从而能够定量分析各种因素对于成本的影响。

模型估计的关键，是要选好适用的模型。我国与外国国情不同，应该寻求适合我国实际

情况的各种系数和指数，不能简单搬用别人的模型。COCOMO 模型适应面较宽，但其调节因子取值时带有主观成分，其分寸不易掌握好。

模型估计法常与自顶向下估计或由底向上估计结合使用。以 COCOMO 模型为例，使用自顶向下方法时，可以用 COCOMO 模型来计算整项目的成本；如果使用由底向上方法，又可用这一模型来计算各个任务单元的成本。

4. 讨论与举例

在这一小节，我们着重讨论下面的两个问题：

(1) 上述 3 类成本估计方法哪种最好？

Boehm 认为，没有一种方法能够在各个方面都优于其他的方法。就自顶向下与由底向上两种估计方法而言，它们的优缺点恰好相反，即一种方法的优点正是另一种方法的缺点。模型法虽在过去十几年中取得不少进展，但不仅适用的模型难找，还有人批评它偏重于过去（指历史数据）和共性（统计平均值），忽略了将来（新技术与新开发环境的影响）和个性（开发中的特定条件）。所以 Boehm 主张几种方法同时使用，然后将所得的结果对照比较。在多数情况下，这种估计和比较要反复多次，才能取得合理的结果。

(2) 如何减小对源程序长度估计的误差？

成本估计的实质，是对源程序长度或开发工作量的估计。在开发早期，这种估计是十分粗略的。根据 Doty 的报告，早期估计的程序长度大都偏低，有时可相差 3 倍之多。还有报导说，对程序长度的估计误差在计划时期可能大于 200%，需求分析阶段会大于 100%，概要设计阶段为 75%，其余各个阶段为 50%。

Putnam 与 Fitzsimmons 曾在一篇文章中以一个大型仓库控制程序为例，说明在开发过程中，怎样通过连续的估计减小对程序的长度的估算误差。下面就摘要转引这个例子。

【例 11.2】在不同开发阶段对程序长度进行追踪估计的示例。

【解】在估计程序长度时，大多数分析员都给出一个范围，代替给出单一的长度值。假定给出的范围是 (a, m, b) ，其中 a 为最小值， b 为最大值， m 为最大可能值，则按照统计学的方法，可以方便地计算出下列指标：

- 源语句期望长度： $SS=(a+b)/2$ 或 $(a+4m+b)/6$ 。
- 标准偏差： $\sigma=(b-a)/6$ 。
- 68% 范围：即 $SS\pm\sigma$ 。程序的真实长度，有 68% 的机会落入这一范围内。
- 99% 范围：即 $SS\pm 3\sigma$ 。程序的真实长度，有 99% 的概率落入这个范围内。

对这些说明后，就可以估计程序长度了。

① 第一次估计：

- 时间：问题定义之后，距开发起点两周。
- 基本情况：对问题了解尚少，只能根据过去经验作粗略估计。
- 估计范围：50 000~140 000 程序行。
- 计算结果：

期望 $SS = (50\,000 + 140\,000) / 2 = 95\,000$ (行)

$\sigma = (140\,000 - 50\,000) / 6 = 15\,000$ (行)

68%范围 = $95\,000 \pm 15\,000$ (行)

99%范围 = $95\,000 \pm 45\,000$ (行)

② 第二次估计:

- 时间: 需求分析结束。

• 基本情况: 已知程序将包括入 A、B、C 三大功能。用特尔斐法分别对每一功能估计出程序的最大 (b)、最小 (a) 和最可能 (m) 的长度。

- 估计范围:

	a	m	b
A:	25 000	40 000	70 000
B:	5 000	15 000	26 000
C:	12 000	36 000	50 000

- 计算结果:

先用公式 $SS = (a + 4m + b) / 6$ 及 $(b - a) / 6$ 分别计算 A、B、C 三大功能的 SS_i 及 σ_i 值, 如表 11.4 所示。

表 11.4 功能 A、B、C 的程序长度

功 能	期望长度(SS_i)	标准偏差(σ_i)
A	42 500	7 500
B	15 167	3 500
C	34 333	6 333

然后计算整个系统的程序长度和标准偏差, 得出

$$SS = SS_A + SS_B + SS_C = 92\,000 \text{ (行)}$$

$$\sigma = \sqrt{\sum_{i=1}^3 (\sigma_i)^2} = \sqrt{(7\,500)^2 + (3\,500)^2 + (6\,333)^2} \approx 10\,421 \text{ (行)}$$

请读者注意上式中的 $\sigma = \sqrt{\sigma_A^2 + \sigma_B^2 + \sigma_C^2}$ 。

68%范围 = $92\,000 \pm 10\,421$ (行)

99%范围 = $92\,000 \pm 31\,263$ (行)

③ 第三次估计:

- 时间: 概要设计末尾, 距开发起点 12 周。

• 基本情况: 软件结构基本说明, 整个程序被分为 11 个功能, 每一功能均估出了 a 、 m 、 b 。

- 估计范围：为节省篇幅，这里就不列出了。下面直接给出计算的结果。
- 计算结果：仿照第二次计算所用的方法，可以算得

$$SS = 98\,475 \text{ (行)}$$

$$\sigma = 7\,081 \text{ (行)}$$

$$68\% \text{ 范围} = 98\,475 \pm 7\,081 \text{ (行)}$$

$$99\% \text{ 范围} = 98\,475 \pm 21\,243 \text{ (行)}$$

④ 小结。表 11.5 列出了 3 次估计的结果。由此可见，随着开发工作的进展，估计的标准偏差 (σ) 从原来的 15 000 行逐步下降到 7 081 行，降低一半以上。第 3 次估计时，不确定性 (uncertainty) 已降至 7.2%，与常见的工程数值的不确定性不相上下了。

表 11.5 对程序长度的 3 次估计

估计时间	SS	σ	68%范围	99%范围
问题定义之后	95 000	15 000	80 000~110 000	50 000~140 000
需求分析结束	92 000	10 421	81 579~102 421	60 737~123 263
概要设计末尾	98 475	7 081	91 394~105 556	77 232~119 718

得出了源程序长度，就可用下列公式计算成本和工作量了：

$$\text{成本 (元)} = \text{成本率 (元/行)} \times \text{程序长度 (行)}$$

$$\text{工作量 (人-月)} = \text{程序长度 (行)} / \text{生产率 (行/人-月)}$$

如果用模型法计算成本，则应先将程序长度 (L) 值代入模型方程，算出工作量 E 。然后把工作量换算为成本。这些在例 11.1 中已说明过，就不再重复了。

(3) 开发工作量的估计

通过估计工作量来计算成本，是又一种常用的成本估计法。图 11.1 的特尔斐成本估计表，其中填写的估计值就是开发工作量。现在也举一个例子。

[例 11.3] 图 11.2 与图 11.3 分别显示了一个 CAD 系统所包含的主要成分及开发每一个功能成分需完成的任务。试用由底向上方法估计开发该系统所需的工作量。

[解] 一种简便的做法，是先画一个工作量矩阵，如图 11.4 所示。把用特尔斐法或其他适当方法估计的每一任务需要的工作量填入表中，就可以根据工作量计算开发成本，即成本 (元) = 成本率 (元/人-月) × 工作量 (人-月)。

需要指出，不同的任务，成本率也各不相同。一般的说，分析与设计任务的成本率将高于编码和测试，计算成本时需分别计算；图 11.4 中没有计算花费在项目计划与管理上的工作量，在实际进行成本估算时，应把这一部分加入成本中。

成本估计也可以利用工具进行。第 11.2 节提到的 SLIM 系统，就是基于 Putnam 模型的自动成本估计工具。由于成本估计在很大程度上依赖于开发单位的条件，所以商品化的成本计算工具还不多见，多数工具仅供开发单位自己使用。

功 能	工 作 量 (人-月)	任 务				合 计
		需求分析	设 计	编 码 与 单 元 测 试	系 统 测 试	
用户接口控制		1.0	2.0	0.5	3.5	7
二维图形分析		2.0	10.0	4.5	9.5	26
三维图形分析		2.5	12.0	6.0	11.0	31.5
数据结构管理		2.0	6.0	3.0	4.0	15
图形显示		1.5	11.0	4.0	10.5	27
外部设备控制		1.5	6	3.5	5	16
设计分析		4	14	5	7	30
合 计		14.5	61	26.5	50.5	152.5
成本率 (元/人-月)		5 200	4 800	4 250	4 500	
成本 (元)		75 400	292 800	112 625	227 250	708 075

图 11.4 计算开发工作量的任务矩阵

11.4 人员的分配与组织

一个工作量为 30 人-月的软件,如要求在 6 个月内完成,则平均每月需要 5 个开发人员。但是在实际工作中,各个开发阶段需要的人力并不相同。一般的说,计划与分析阶段只需要很少的人,概要设计参加的人略多一些,详细设计的人员又多一些。到了编码和测试阶段,参加的人数达到最高峰。在运行初期,需要较多的人参加维护,但很快就可以减少下来,只需保留很少的维护人员就可以满足需要了。

怎样按照实际需要来确定各阶段所需的人力?有没有规律可循?把开发人员组织起来,有哪些可供借鉴的方式?下面将介绍一些有关的指导原则。

1. Rayleigh-Norden 曲线

最早该曲线以 Rayleigh 爵士的名字命名,主要用来解释某些科学现象。1958 年, Norden 发现这一曲线可用来说明科研及开发项目在实施期间所需要的人力。1976 年, Putnam 又把这一曲线与软件开发联系起来,发现在软件生存周期内各个阶段需要的人力分配,具有与 Rayleigh-Norden 曲线十分相似的形状。本章第 11.2 节介绍的 Putnam 资源模型,就是以这个曲线为基础推导出来的。

图 11.5 显示了 Rayleigh-Norden 曲线的典型形状。图中以横坐标表示距开发起点的时间,纵坐标代表在不同时间点需要的人力。 t_d 位于曲线的峰点。在 t_d 之前,开发所需的人力逐渐

上升,直到达到峰值 t_d ; t_d 以后,单位时间所需的人力渐趋下降。Putnam 在考察了数以百计的大、中型软件开发项目后,发现大部分项目的人力花费与这一曲线相吻合。且图中的 t_d 大致相当于软件开发完成的时间。换句话说, t_d 的左方大致相当于软件的计划与开发时期,而其右方相当于运行和维护时期。曲线下方的面积,就是整个生存周期所需的工作量。Putnam 还发现,对于大型的软件, t_d 左右两侧的面积比约为 4:6,即计划与开发所需的工作量约占生存周期总工作量的 40%,而维护工作量将占 60%左右。

图中用虚线画出的矩形,显示了平均使用人力所造成的问题。开始阶段人力过剩,造成浪费(图中①),到开发后期需要人力时,又显得人手不足(图中②),以后再补偿,已为时过晚了(图中③)。

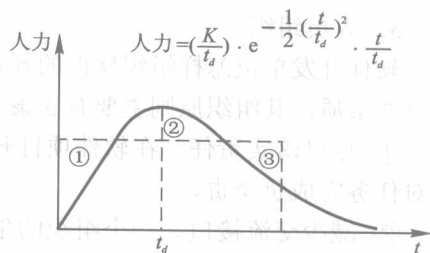


图 11.5 Rayleigh-Norden 曲线

2. 两条重要的定律

① 人员-时间权衡定律 (people-time trade-off law)。Putnam 在对 Rayleigh-Norden 曲线做了大量研究的基础上,提出了 Putnam 模型。这个模型指出,开发工作量与开发时间的 4 次方成反比(参阅第 11.2 节)。即

$$E = \text{常数} / (T \text{ 或 } t_d)^4$$

Putnam 将这一结论称为软件开发的权衡定律。

开发一个大、中型软件,一般需要 1~3 年的时间。这不仅是因为这些软件具有较大的规模(通常为 5 000~100 000 行),而且其内部的各项任务有先后,一个任务未完,另一个任务就不能开始。如果缩短开发时间,势必增加单位时间内需要的开发人员。而人员的增加,意味着有更多的时间将用于相互通信,因而减少了直接用于软件生产的时间。仍以例 11.2 的仓库控制程序为例,最终估计的源程序长度为 98 475 行。假定两年完成开发需要工作量为 25 人-年,则当开发时间压缩为 1.8 年(缩短 1/10)时,工作量将增加至 35 人-年,增长率达到 40%。表 11.6 显示了这两种开发情况的比较。

表 11.6 两种开发时间的比较

时间(年)	工作量(人-年)	平均每年需要人数
2.0	25	12.5
1.8	35	19.4

如果时间和人员可以互换,则开发时间为 1.8 年时,年平均人数应为 13.9 (= 25/1.8) 人。但按照权衡定律计算,年平均人数将需要 19.4 (= 35/1.8) 人。

② Brooks 定律。曾担任 IBM 公司 360 系列操作系统项目经理的 F. Brooks,从大量的软件开发实践中得出了另一条结论:“向一个已经延迟的项目追加开发人员,可能使它完成得更晚。”鉴于这一发现的重要性,许多文献称之为 Brooks 定律。

这里, Brooks 从另一个角度说明了“时间与人员不能互换”这一原则。对该原则的一个合理解释是,当开发人员以算术级数增长时,人员之间的通信将以几何级数增长,从而可能导致得不偿失的结果。

开发时间宁可长一点,开发人员宁可少一点,这是上述两条定律给我们的启示。

3. 人员组织

软件开发单位怎样组织软件的开发工作,取决于许多因素。根据软件项目的特点及参与人员的素质,其组织原则主要有 3 条。

① 尽早落实责任。在软件项目开始组织时,尽早指定专人负责,赋予他相应的权利,并对任务完成负全责。

② 减少交流接口。一个组织的生产率随完成任务中通信路径数目的增加而降低。有合理的人员分工、好的组织结构以及有效的通信,才有可能提高生产率。

③ 责权均衡。软件经理人员承担的责任应与赋予他的权力相当。

一般情况下,可采取以下的层次型组织结构,即软件经理→项目经理→开发小组。软件经理负责管理整个单位的开发工作。每一开发项目设一位项目经理,每一项目经理又管理若干开发小组。对于大型的、有数十人以上参加的项目,在项目经理与开发小组之间还可以添加一至几个层次,以保证管理的有效性。

在本节介绍两种不同的开发小组形式。

(1) 民主开发小组

Weinberg 认为,软件开发是一种合作的事业。最理想的形式是组成无我小组(egoless team)。这种小组提倡“无我程序设计”,人人把小组开发的程序看成“我们的”程序,而不是“我的”程序。组内人人平等,一切问题均由集体决定,甚至组长也轮流担任。这种形式的优点是,便于集思广益,取长补短;但责任不清,而且每件事都要讨论,效率不高。日本许多大公司采用了民主开发小组这种形式,强调集体讨论、人人献策,收到了较好的效果。但小组的组长由上级指定,当组内发生意见分歧时,由组长作最后决定。民主开发小组的人数,通常为 5~7 人。

(2) 主程序员小组(chief-programmer team)

这是由 Mills 和 Baker 等人提出的一种方式。它的特点是强调“一元化”领导,每一组员的工作由主程序员分配,一切重大的问题由主程序员决定。图 11.6 显示了这种小组的组成。主程序员除了领导 2~5 名程序员外,还领导一名文档员(负责文档和程序管理)、一名后援程序员(主要负责质量保证,平时作为主程序员的助手,需要时代替主程序员工作),从而形成一个集技术和管理于一身的开发小组。美国 IBM 公司有許多开发项目采用这种形式,取得了较好的成效。

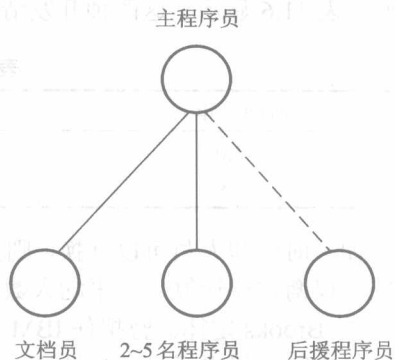


图 11.6 主程序员小组

这种形式的优点是，每个组员仅对主程序员负责，省却了组员之间的通信，提高了效率。但它的效果好坏，在很大程度上取决于主程序员的技术和管理水平，当开发小组内只有一个高级程序员，其余均为初、中级程序员时，可以采用这种方式。

11.5 项目进度安排

进度安排是项目管理的一个重要内容。对于大型和复杂的软件开发项目，也是一项困难的任务。本节将简要介绍两种安排进度计划的方法——Gantt 图法和计划评审技术。这些方法都是制定工程计划进度常用的技术，在其他工程领域中早有应用。

1. 计划评审技术

计划评审技术（Program Evaluation and Review Technique, PERT）是于 20 世纪 50 年代后期由美国海军和洛克希德公司首次提出的一项技术，并把它成功地应用于北极星导弹的研究和开发。40 余年来，它在许多工程领域获得了广泛的应用，所以有时 PERT 技术也称为工程网络技术。下面将结合一个简单的例子，说明怎样用这一技术来制定软件的进度计划。

(1) 建立 PERT 图

这是运用 PERT 技术的第一步。图 11.7 是一个简单软件项目的 PERT 图（又称网络图）。图中的每一圆圈，都代表一项开发活动。圈内的数字表示完成这一活动所需的时间，箭头代表活动发生的先后顺序。例如在图 11.7 中，完成分析需要 3 个月，完成设计需要 4 个月，设计发生在分析之后，即只有在分析结束后，才能开始设计等。

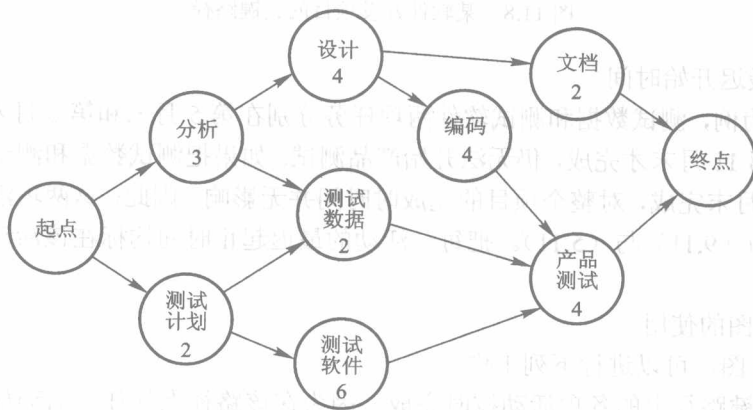


图 11.7 一个简单软件开发项目的 PERT 图

经验表明，采取从后向前建立 PERT 图的方法，常常比较容易。也就是说，首先画出终点，然后逐步前推，画出每个活动，直至项目的起点。

(2) 找出关键路径 (critical path)

从起点到终点,可能有多条路径。其中耗时最长的路径就是关键路径,因为它决定了完成整个工程所需要的时间。

与建立 PERT 图不同,寻找关键路径是从项目起点开始的。其方法是,从起点到终点,在每个活动框的上方标出该项活动的起止时间。如图 11.8 所示,起点上方的(0,0)表示其起止时间都是“0”;分析活动始于“0”,终于“3”,历时 3 个月;设计活动始于“3”,终于“7”,历时 4 个月;依此类推。显而易见,图中用双线箭头标出的路径需时最长(共 15 个月),是本例中的关键路径。

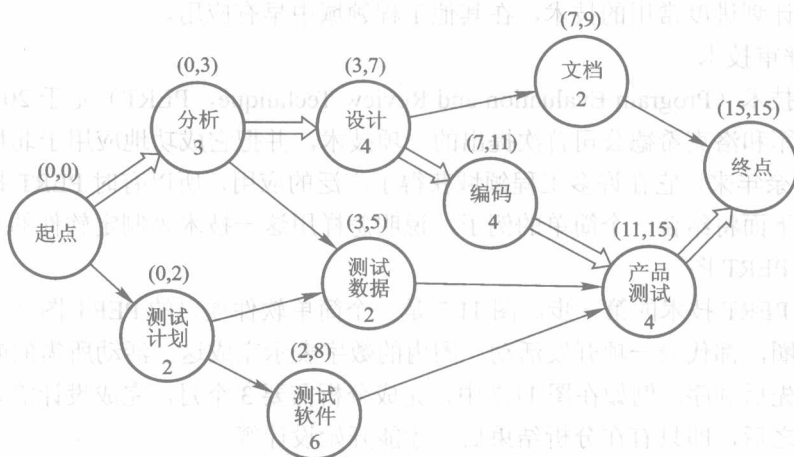


图 11.8 某软件开发项目的关键路径

(3) 标出最迟开始时间

以图 11.8 为例,测试数据和测试软件两项任务分别在第 5 月末和第 8 月末就可完成,但因为编码要在第 11 月末才完成,仍无法开始产品测试。如果把测试数据和测试软件两项活动均推迟到第 11 月末完成,对整个项目的完成时时间并无影响。因此,这两项活动的最迟起止时间可分别记为(9,11)与(5,11)。把每一活动的最迟起止时间均标在该活动的下方,就可得到图 11.9。

(4) PERT 图的使用

利用 PERT 图,可以进行下列工作:

① 确保关键路径上的各项活动按时完成。因为在该路径上的任何活动如有延期,整个项目将随之延期。

② 通过缩短关键路径上某活动的时间,达到缩短项目开发时间的目的。例如在图 11.9 中,如果把设计时间从 3 个月缩短为 2 个月,编码从 4 个月缩短为 2 个月,则项目开发时间要将从原来的 15 个月缩短为 12 个月。这时 PERT 图中将出现两条关键路径,如图 11.10 所

示。显然，此时即使把分析活动从3个月压缩为2个月，也不能再将开发时间缩短。但是，如果把处于两条关键路径上的公共活动——产品测试从4个月缩短为3个月，就可以把项目开发时间进一步缩短为11个月。

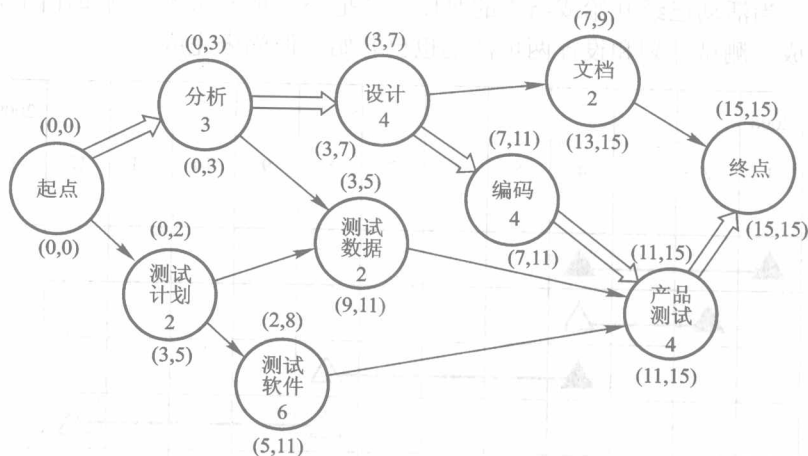


图 11.9 注有最迟开始时间的 PERT 图

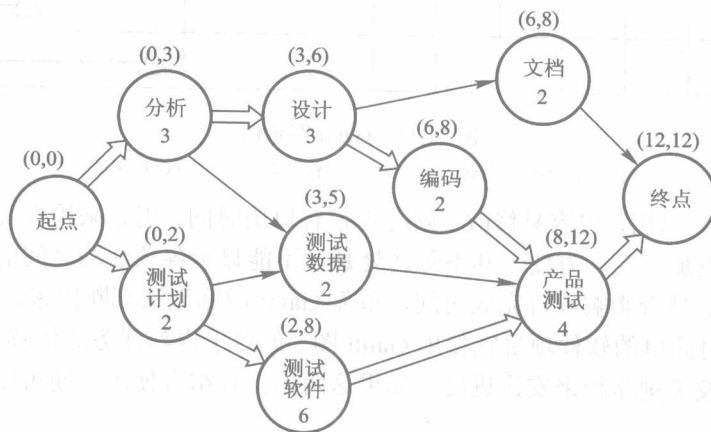


图 11.10 出现两条关键路径的 PERT 图

③ 对于不处在关键路径上的活动，可根据需要或者调整其起止时间，或者延缓活动的进度。例如在图 11.9 中，可以把测试数据活动的起止时间调整为 (8,10)，使测试软件和测试数据两项活动可以交给同一个人去完成。又如文档活动的起止时间可调整为 (7,13)，即将其完成期限从 2 个月放宽到半年，这样，参与这一任务的人员就可以适当减少。

2. Gantt 图

Gantt 图是安排软件进度计划的又一有用工具。图 11.11 显示了这种图的一般形式，在图的左方列出项目的开发活动，上方列出了日历时间。在每一活动的开始时间和结束时间各画一个小三角形，当活动已经开始或结束的时候，就把小三角形涂黑。例如在图 11.11 中，分析活动已经完成，测试计划和设计两项活动也已开始，但尚未完成。

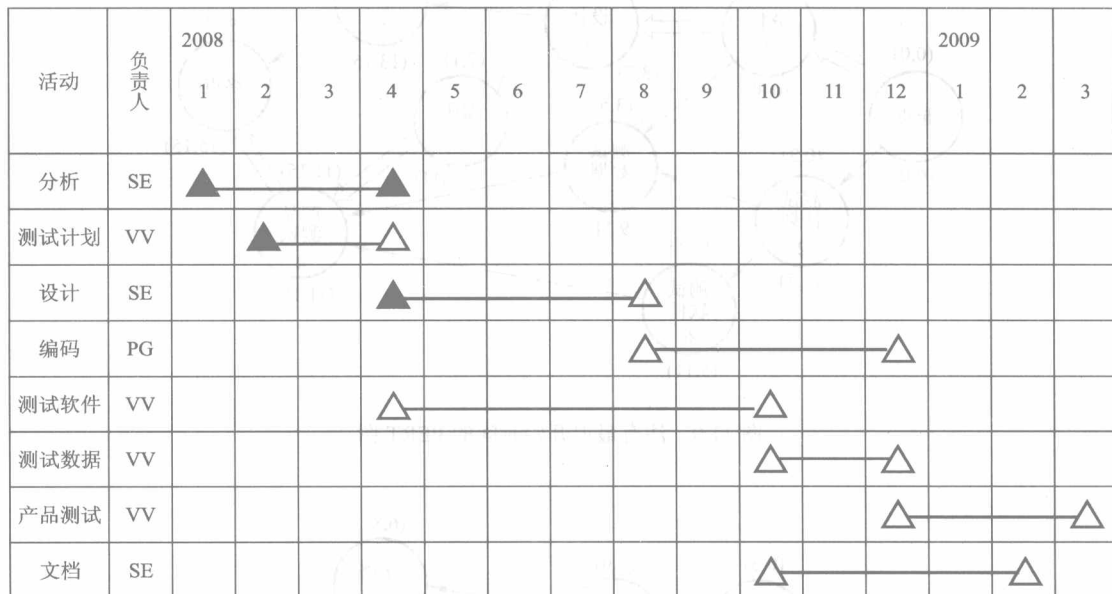


图 11.11 Gantt 图示例

注：SE 系统工程师 VV 质量保证人员 PG 程序员

Gantt 图简单、易用，也容易修改。同时因带有日历时间，用它来检查开发活动的完成情况，比 PERT 图更加直观、方便。其不足之处是，不能显示各项活动之间的依赖关系。例如某项活动延期，它是否影响项目完成时间，单靠 Gantt 图则无法判断出来。

一般说来，对简单的软件项目宜使用 Gantt 图。但对于内部任务的依赖关系复杂的项目，应使用 PERT 图及关键路径来安排进度。如果这两种工具结合使用，便可以相互取长补短，更好地安排进度。

小 结

软件工程管理是软件工程的重要组成部分。只有进行科学的管理，才能使先进的技术充分发挥作用。对于大型的软件开发，管理工作尤为重要。

项目管理是整个管理工作的基础。所谓项目管理，其主要任务是制定软件开发计划，调

度资源，跟踪、监督和协调工程进度，保证工程如期按质完成。本章从项目费用、人员、进度等方面讨论了项目管理的基本内容，介绍了在国外行之有效、比较流行的一些管理技术。其中多数技术也适用于我国，虽然有一部分（如资源模型）因国情不同不能直接搬用，但其思想与做法仍值得参考。

管理工作离不开度量。“靠度量来管理”（management by measurement）是现代管理工作的一条重要原则。无论用于估计成本的各种模型，或计算人力需求的 Rayleigh-Norden 曲线，或计算进度的 PERT 技术，都标志着现代软件管理已从定性的阶段发展到定量的阶段。软件经济学和软件度量的诞生，正是“靠度量来管理”这一原则在软件管理中的具体体现。

习 题

1. 软件工程管理有哪些主要内容？它们的作用是什么？
2. 一个 4 万行规模的应用程序，花 50 万美元可以在市场上买到。如果自己开发，则每人一月的总花费需 4 000 美元。试问是购买合算呢，还是自己开发合算呢？（注：开发成本用 COCOMO 模型计算）
3. 有人说：专家判断和算法模型估计是成本估计的两种最基本方法。你同意这种看法吗？为什么？
4. 为什么说“时间和人员不能交换”？试说明其含义。
5. 某大型软件开发中，最多时有上百人参加，构成 15~20 个主程序员小组。你认为应该采取什么样的层次组织结构，才能使大家协调工作？试画图说明，并阐述理由。
6. 图 11.12 是某软件项目的 PERT 图。

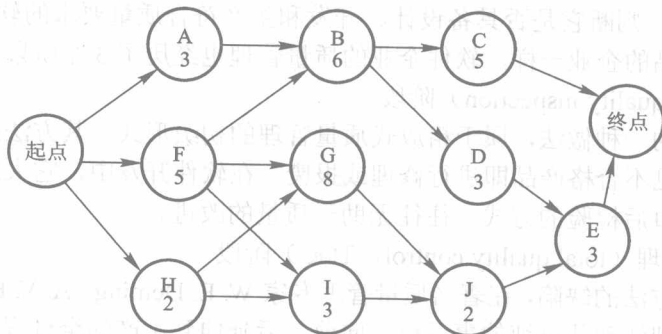


图 11.12 习题 6 图

圆圈中的数字代表活动所需的周数。要求：

- (1) 找出关键路径和完成项目的最短时间。
 - (2) 标出每项活动的最早起止时间与最迟起止时间。
7. 在上题中，若活动 G 的持续时间缩短为 (1) 7 周；(2) 6 周；(3) 5 周；试问完成项目的最短时间有何变化？
8. 将第 6 题的内容改用 Gantt 图来表示。
 9. 为什么说“靠度量来管理”？谈谈你对这个问题的认识。

第 12 章 软件质量管理

保证软件的质量，是一个贯穿于整个软件生存周期的重要问题。在以前各章中，已经多次提到过软件的质量管理。在早些时候，质量管理与质量保证可视为同义词，经常不加区分地使用。自 20 世纪 90 年代以来，质量认证逐渐在企业流行，把对于产品的质量认证扩展到对于整个企业的质量认证。于是，软件现代质量管理的内容也相应地扩展为包括质量保证与质量认证等两个方面。本章将从软件工程管理的角度，对质量保证与质量认证作一次综合性的讨论。

12.1 从质量保证到质量认证

质量保证 (quality assurance) 与质量认证 (quality certification) 是组成软件现代质量管理的两个重要方面。二者的区别在于：前者着眼于每一个软件，保证提供给用户的产品都达到规定的质量水平；后者则更加注重软件企业的整体资质，目的在于全面考察企业的质量体系 (quality system)，判断它是否具备设计、开发和生产符合质量要求的软件产品的能力。

与生产其他产品的企业一样，软件企业的质量管理也经历了 3 个阶段。

1. 质量检验 (quality inspection) 阶段

这是早期常见的一种做法，属于粗放式质量管理的初级形式。其方法是在生产线的末端逐一检验产品，遇见不合格产品即进行修理或报废。在软件开发中，它大致相当于对程序的测试与纠错。这种事后检验的方式，往往无助于质量的改进。

2. 全面质量管理 (total quality control, TQC) 阶段

为了克服上述方法的缺陷，在著名质量管理专家 W. E. Deming、A. V. Feigenbaum 等人的推动下，把质量管理活动从单纯的事后检验向前、后延伸到生产的全过程，于是全面质量管理便应运而生。这里的所谓全面，包含了全过程控制和企业全员参加两层含义。其基本思想是：

① 产品质量形成于生产全过程。正如 Deming 所说，质量是制造出来的，不是检验出来的。

② 应重视建立质量体系，Feigenbaum 认为，该体系应包括适当的管理与技术作业程序，以及由这些程序所组成的结构。

③ 上述两位专家还倡导，包括质量管理在内的任何管理工作，都应该按照 PDCA (Plan—Do—Check—Action) 循环所建议的“计划—实施—检查—措施”的顺序来实施。

把以上思想应用于软件的质量管理，便形成了软件的质量保证活动，详见第 12.2 节。

3. 质量认证阶段

从 20 世纪 80 年代后期逐渐兴起的质量认证（又称合格认证，conformity certification）把对于个别产品的质量认证扩展到对于整个企业质量体系的认证。1987 年，国际标准化组织公布了 ISO 9000 质量管理标准，从此质量认证迅速流行。软件质量认证也开始在软件工业界受到广泛地关注。

与质量保证相比，质量认证不仅范围更广（从仅对产品到包容产品与服务），而且具有第三方开展的活动的性质。它尤其强调：质量管理必须坚持进行质量改进；应该使企业具有持续提供合格产品的能力。Feigenbaum 认为，质量管理的核心是预防，而不是消极的补救；Deming 也认为，只有把不断研究与改进质量管理当作各级生产人员的责任，才能使确保顾客满意的宗旨真正实现。自 20 世纪 90 年代以来，先后出现了针对软件开发的能力成熟度模型（SEI CMM）和 SPICE 信息技术-软件过程评估等标准，它们都是软件质量认证的重要研究成果。本章后半部分将详细介绍。

12.2 质量保证

按照 IEEE 1983 年公布的《软件工程标准词汇》，质量保证被定义为：为了充分保证项目或产品符合规定技术需求而进行的一系列必要的有计划的活动。在“质量保证方法学”一文中，P. Pfau 给出了一个更为详细的定义。他说：“质量保证是指在软件开发过程中，为了保证产品满足指定标准而进行的各种活动。这些活动的作用，是减少产品在目标环境中实现其功能的怀疑和风险。”上述的定义表明，质量保证包含一系列的活动，其目的是使所开发的软件达到规定的质量标准。以下简介这两个方面的内容。

12.2.1 软件的质量属性

软件的质量标准，可以用一组有关的属性来表示。20 世纪 70 年代末期，T. McCall 与 Boehm 等人曾先后著文论述软件的质量，分别提出了一组衡量软件质量的属性，如表 12.1 所示。

表 12.1 软件的质量属性

	T. McCall 等, 1977 年	B. Boehm 等, 1978 年
运行性能	可靠性	reliability
	效率	efficiency
	运行工程	human engineering
		可靠性
		效率
		正确性
		可用性
		完整性

续表

	T. McCall 等, 1977 年	B. Boehm 等, 1978 年
维护性能	可理解性 understandability	可维护性 maintainability
	可测试性 testability	可测试性 testability
	可修改性 modifiability	可适应性 flexibility
移植性能	可移植性 portability	可移植性 portability
		可重用性 reusability
		交互操作性 interoperability

在我国国家标准《软件产品评价、质量特性及使用指南》(GB/T 16260—1996)中,把有关软件质量的属性归纳为 6 项,即功能性、可靠性、易用性、效率、可维护性和可移植性。其中功能性是指程序能够满足软件需求规格说明书(SRS)中各项功能需求的能力,大致相当于表 12.1 中的正确性;效率包括在线系统的响应时间或完成程序规定功能所需的时间,以及程序对存储空间或外部设备的占用量等资源方面的因素;其他 4 项的意义自明。把这 6 项用通俗的语言来描述,就是:功能正确,运行可靠,使用方便,效率很高,容易维护,容易移植。不难看出,这些属性概括了通常对软件质量的最基本的要求。

12.2.2 质量保证的活动内容

为了达到上述要求,在各个阶段要进行哪些活动呢?

图 12.1 是引自 Pressman 的一张图。该图直观地表明,质量保证是复审、开发方法、配置控制与程序测试的综合应用。简单地说,软件的开发方法应该符合规定的软件开发规范;计划和开发时期各个阶段的工作都要进行复审;每个阶段产生的文档都必须严格管理,以确保文档和程序的完整性与一致性;作为最后的一道防线,还要坚持对程序进行不同层次的测试。

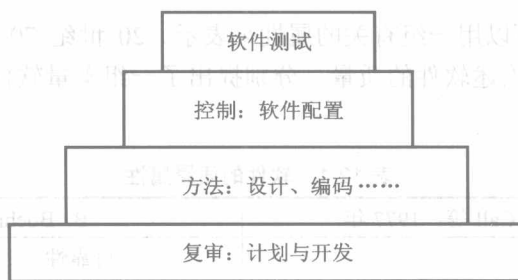


图 12.1 质量保证的活动内容

所有以上的各项活动内容,都需写入软件项目的“质量保证计划”,并由质量保证小组监督实施。由此可见,质量保证既是技术活动,也是管理活动。

鉴于上述活动大都已在前面各章中分散讲述，这里仅就几个重点问题作简略的补充。

1. 验证与确认 (verification and validation, V&V)

这是两个不同的概念。前者是为了确定开发时期中某一阶段的产品是否达到了前一阶段对它的需求，后者则是在整个开发结束时对所开发的软件能否满足软件需求的总评价。换句话说，前者仅要求两个相邻阶段间的一致性，后者则要求在整个开发时期内的一致性。Boehm曾对两者的差异作过精辟的描述和对比：

验证：我们制造产品的步骤正确吗？(Are we building the product right?)

确认：我们制造的是正确的产品吗？(Are we building the right product?)

具体地说，验证将包含开发时期各个阶段进行的复审、(人工)复查与测试活动；确认则主要指测试阶段的确认测试和验收时的系统测试等活动。两者结合起来，就构成质量保证的中心内容。在第 8.4.4 节，曾提到测试阶段使用的两种文档——测试计划和测试报告。在实际执行中，常常把上述的计划和报告扩充为关于验收与确认的计划和报告，用以代替范围较小的测试文档。

2. 开发时期的配置管理

在第 9.5 节，曾强调在维护时期坚持配置管理的重要性。事实上，对配置的控制从计划时期就开始了，一直延续到生存周期结束、软件停止使用后才终止。

前已指出，软件配置包括生存周期中各个阶段产生的文档和程序。这些文档或程序是随着软件的开发进程逐步产生的，所以也称为阶段产品。诸如软件的项目计划、需求说明、测试计划、设计文档和源程序，都属于阶段产品的范围。配置管理的中心思想，就是在软件开发的进程中，开发者有权对本阶段的阶段产品进行更改，但一旦阶段产品通过了复审，就应将它交给配置管理人员去控制，任何人(包括编制这一文档的人员)需要对它更改时，都要经过正式的批准手续。在软件工程的术语中，各个阶段产品的复审时间均称为基线(baseline)，基线之前更改自由，基线之后严格管理。正是这种对软件配置的连续控制与跟踪，保证了软件配置的完整性与一致性。

12.3 软件可靠性

在软件的质量特性中，可靠性占有重要的地位。本节将对可靠性的定义、分级和模型作一简要的介绍。

12.3.1 可靠性的定义和分级

1. 可靠性定义

软件可靠性有多种不同的定义。其中一种为多数人接受的定义是：软件可靠性是在给定的时间内，程序按照规定的条件成功地运行的概率。

假设 $R(t)$ 代表在时间 $[0 \sim t]$ 之间的软件可靠性， $P\{E\}$ 代表事件 E 的概率，则软件可靠

性可以表示为

$$R(t) = P\{\text{在时间 } [0, t] \text{ 内按规定条件运行成功}\}$$

不言而喻，可靠性与软件内部的故障密切相关。如果软件在交付使用时有遗留错误，则当程序的初始条件与输入值在运行中出现某种组合，使执行路径恰好通过包含遗留错误的路径时，就会使程序在运行中失效。当残留错误的数量为一定时，程序的运行时间越长，则发生失效的机会越多，可靠性也随之下落。为了简化所讨论的问题，我们假定软件的故障率是不随时间变化的常量，则根据经典的可靠性理论， $R(t)$ 可以表示为时间与故障率的指数函数，即

$$R(t) = e^{-\lambda t}$$

其中， t = 程序运行时间； λ = 故障率，即单位时间内程序运行失败的次数。

图 12.2 是可靠性随运行时间与故障率变化的示意图。由图可见， λ 一定时，运行时间越长， $R(t)$ 越小。

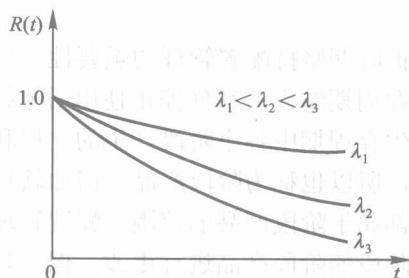


图 12.2 可靠性随 t 、 λ 的变化

在这里，读者应注意软件可靠性与计算机系统可靠性的差别。系统的可靠性，是软件、硬件和操作人员的操作这 3 种可靠性的综合反映。用式子表示，可以写作

$$R_{\text{SYS}} = R_{\text{S}} R_{\text{H}} R_{\text{OP}} = e^{-(\lambda_{\text{S}} + \lambda_{\text{H}} + \lambda_{\text{OP}}) t}$$

其中， R_{SYS} 为系统的可靠性， R_{S} 、 R_{H} 及 R_{OP} 依次表示软件、硬件和操作的可靠性。为了满足系统的可靠性， R_{S} 、 R_{H} 与 R_{OP} 都必须具有比 R_{SYS} 更高的指标。

另一种衡量可靠性的方法，是直接计算系统的平均无故障时间 (MTTF)。在故障率为常量的情况下，MTTF 可以看作故障率的倒数，即

$$\text{MTTF} = 1/\lambda$$

于是，

$$\lambda_{\text{SYS}} = \lambda_{\text{S}} + \lambda_{\text{H}} + \lambda_{\text{OP}}$$

$$\text{MTTF}_{\text{SYS}} = 1 / (1 / \text{MTTF}_{\text{S}} + 1 / \text{MTTF}_{\text{H}} + 1 / \text{MTTF}_{\text{OP}})$$

例如, 设定 $MTTF_H = MTTF_S = 500$ 小时, $MTTF_{OP} = 2\ 500$ 小时, 则可得 $MTTF_{SYS} = 227$ 小时。

2. 可靠性等级

不同的软件, 对可靠性的要求也不相同。有些软件发生故障后, 仅给工作带来轻微的不便, 或虽有损失, 也容易恢复。另一些软件一旦发生故障, 有可能造成重大经济损失, 甚至危及人的生命。Boehm 将软件可靠性分为 5 级, 如表 12.2 所示。在计划时期, 可以参考该表的分级, 确定所开发软件的可靠性等级。

表 12.2 可靠性分级表

分 级	故障的后果	开发工作量比例因子
甚低	工作略有不便	0.75
低	有损失, 但容易弥补	0.88
正常	弥补损失比较困难	1.00
高	重大的经济损失	1.15
甚高	危及人的生命	1.40

如前所述, 要想获得高的可靠性, 需要在各个开发阶段, 尤其是软件测试上下工夫。提高可靠性总是以降低生产率为代价的。从表 12.2 可知, 同一个软件, 当可靠性等级从甚低变为甚高时, 其开发工作量及成本大约要增加一倍 (0.75 : 1.40)。因此, 用户在向开发单位提出可靠性的要求时, 应该从实际需要出发, 而不是越高越好。

12.3.2 可靠性模型

为了研究软件的可靠性, 已经建立了各种关于可靠性的模型。这些模型或者在计划时期用来预测程序的可靠性, 或用在开发时期指导人们采取相应的措施, 确保被开发软件达到所需要的可靠性等级。粗略地说, 可靠性模型可区分为宏观模型和微观模型两大类。后者是建立在对程序语句和控制结构详细分析的基础之上的, 在开发时期很难建立; 前者则忽略程序在结构方面的细节, 主要从程序中残留错误的角度来建立模型, 并且用统计方法确定模型中的常数。

有人统计, 到 20 世纪 70 年代末期, 公开发表的可靠性模型已达到 20 余种。其中大多数属于宏观模型, 相互间并无本质上的差异; 虽然许多宏观模型经过实际数据的检验, 但是从实用的角度看, 它们还很不成熟。微观模型的研究, 也遇到了很多困难。以下仅选择介绍几种宏观模型, 使读者可窥一斑。

1. 正比于遗留故障数的宏观模型

前已指出, 程序的故障率与遗留错误的数量成正比。所以要根据程序中遗留错误的多少, 就可以预测程序可靠性。

假设 τ = 程序的调试 (debugging) 时间;

E_T = 调试前的错误总数;

$E_C(\tau)$ = 在时间 $0 \sim \tau$ 期间纠正的错误数;

$E_r(\tau)$ = 在时间 τ 时的遗留错误量;

I_T = 程序的长度或指令总数;

则可有

$$E_r(\tau) = E_T - E_C(\tau)$$

用 I_T 除等式两边, 可得到错误的规格化值, 即

$$\varepsilon_r(\tau) = E_T / I_T - \varepsilon_C(\tau)$$

其中 $\varepsilon(\tau) = E_T(\tau) / I_T$, $\varepsilon_C(\tau) = E_C(\tau) / I_T$

由于软件的故障率与遗留错误的数量成正比, 故有

$$\lambda = K \varepsilon_r(\tau) = K (E_T / I_T - \varepsilon_C(\tau)) \quad (12.1)$$

式中的 K 为比例常数, 可以从实验求得。

根据经典的可靠性理论, 就可得到如 (12.2) 式所示的正比于遗留故障数的模型 (bug-proportional model):

$$R(t) = e^{-\lambda t} = e^{-K(E_T / I_T - \varepsilon_C(\tau))t} \quad (12.2)$$

2. 平均无故障时间模型 (MTTF 模型)

如前所述, 软件可靠性也可用平均故障时间来衡量。这种基于 MTTF 的模型, 就称为 MTTF 模型。

已知当故障率为独立于时间的常量时,

$$\text{MTTF} = 1 / \lambda$$

代入 (12.1) 式中 λ 的值, 可得

$$\text{MTTF} = 1 / [K (E_T / I_T - \varepsilon_C(\tau))] \quad (12.3)$$

为简化讨论, 又设在时间 $0 \sim \tau$ 期间的纠错率为常数, 且等于 ρ_0 , 则

$$\varepsilon_C(\tau) = \rho_0 \tau$$

代入 (12.3) 式, 可得

$$\text{MTTF} = 1 / [K (E_T / I_T - \rho_0 \tau)] \quad (12.4)$$

或

$$\text{MTTF} = 1 / [\beta (1 - \alpha \tau)] \quad (12.5)$$

其中 $\beta = K E_T / I_T$, $\alpha = \rho_0 I_T / E_T$

式 (12.5) 表明了 MTTF 与测试时间 τ 、纠错率 ρ_0 之间的关系, 也即 MTTF 模型。图 12.3 显示了根据这一模型绘成的曲线。

图 12.3 表明, 随着调试时间 τ 的增加, MTTF 与程序的可靠性均不断提高。但是, 当 $\alpha \tau < 0.5$ 时, MTTF 的改善是缓慢的。例如, 当 $\alpha \tau$ 由 0.2 增大至 0.3 时, MTTF 仅从 $1.25 / \beta$ 增长到约 $1.4 / \beta$, 提高不到 20%。但当 $\alpha \tau > 0.75$ 时, MTTF 的值明显上升,

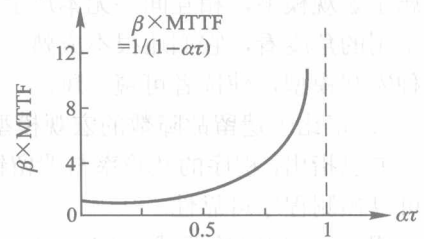


图 12.3 纠错率为常数时的 MTTF 模型

例如当 $\alpha\tau$ 由 0.8 增加到 0.9 时, MTTF 将从 $5/\beta$ 增长到 $10/\beta$, 可靠性提高了一倍。这一结论告诉我们, 如果有必要显著改善程序的可靠性, 只需把调试时间延长到 $\alpha\tau$ 接近于 1 的区域, 就可以达到目的。

还需指出, 式 (12.5) 是假定纠错率为常量时导出的。在实际调试中, 纠错率并非常量, 故图 12.3 的曲线形状也有所变化。但 MTTF 在最后一个区段急剧上升的趋势, 是不会改变的。

3. 错误植入 (error seeding) 模型

这类模型的中心思想, 是通过估计残留错误的数量, 来确定程序的可靠性。具体的做法是: 测试之前先在程序中植入一批人为的错误, 在测试过程中分别统计出由测试小组发现的原有错误和植入错误, 然后由下列算式计算原有的错误。

假设 N = 程序中原来残留的错误数;

S = 新植入程序的错误数;

n = 测试中发现的原有错误数;

s = 测试中发现的植入错误数;

如果调试中对两类错误具有同样的发现能力, 则有

$$N/n = S/s, \text{ 或 } N = nS/s \quad (12.6)$$

(12.6) 式表示的, 就是 Mill 等人提出的最初的错误植入模型。

上述模型的问题是, 怎样做到对原有错误和植入错误具有相等的发现能力? 如何确定植入错误的数量? 即需要植入多少错误才算合适? 显然, 这些问题都不易解决。

1973 年, Hyman 提出了一种改进的方法。他建议, 安排两名测试员同时对一个程序进行独立的测试。

假定 B_0 = 程序中已有的残留错误数;

B_1 = 1 号测试员在某一时间内发现的错误数;

B_2 = 2 号测试员在同一时间内发现的错误数;

B_C = 两名测试员共同发现的错误数;

则仿照公式 (12.6), 可以得到

$$B_0 = B_1 B_2 / B_C \quad (12.7)$$

如果从头至尾都用两套人马参加测试, 势必增加开发的成本。因此 Hyman 想出了一个巧妙的方法。举例说, 假如整个测试预计需要 4 个月, 起初由两人同时测试, 且每隔数周估算一次 B_0 。这样重复估算数次, 等一或两个月 (即预计测试时间的 $1/4 \sim 1/2$) 以后, 一般就能得到较为满意的估计结果。此时, 2 号测试员就可以停止测试, 由 1 号测试员单独完成其余的测试工作。由于 1 号测试员可以利用 2 号测试员的已有结果来加速自己的工作, 所以无论在时间上或经济上, 它都比早先的植入错误法优越。

12.3.3 软件容错技术

容错性是软件可靠性的子属性之一。软件开发首先要避免错误, 尽量采用无差错

(error-free) 的过程与方法, 例如净室工程方法。另一种做法, 是当软件在运行中一旦出现错误, 便将它的影响限制到可容许的范围之内, 这就是容错 (fault tolerance)。所有高可靠、高稳定的软件都非常重视采用容错技术。

1. 容错软件

所谓容错软件, 就是具有抗故障功能的软件。根据其处理错误方法的不同, 可区分为以下 3 种情况:

① 屏蔽错误。把软件的错误屏蔽掉, 使之不致产生危害。

② 修复错误。能在一定程度上使软件从错误状态恢复到正常状态。

③ 减少影响。能在一定程度上使软件完成预定的功能。

2. 冗余技术

冗余 (redundancy) 技术是实现容错软件的主要手段, 常用的冗余技术有结构冗余、时间冗余、信息冗余等多种。分述如下。

(1) 结构冗余

又包括静态冗余、动态冗余、混合冗余等不同形式。

图 12.4 是采用三模静态冗余结构的表决系统。图中 M_1 、 M_2 、 M_3 分别代表由 3 个小组开发的具有相同功能的模块, 其输出连接到表决器 V (可用软件或硬件组成) 的输入端, 表决器的输出为

$$U = (u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$$

由上式可知, 3 个模块中无论哪一个出错, 都能被表决器屏蔽, 使系统不经切换就实现容错。

图 12.5 采用的是动态冗余结构。与静态冗余结构相似, 该系统也由多个具有相同功能的模块构成。与图 12.4 不同的是, 这些模块可以同时运行 (热备份系统), 也可以依次运行 (冷备份系统), 而且任何时刻只有一个模块的输出可连接到系统的输出。仅当当前的模块运行出错时, 其余的模块才能经过开关切换顶替出错的模块进行输出。

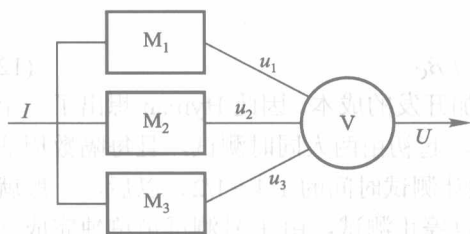


图 12.4 静态冗余结构

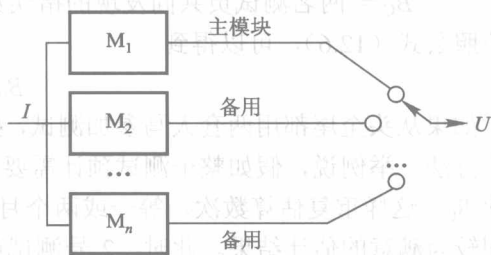


图 12.5 动态冗余结构

动态冗余结构的优点是, 除非所有的模块全部出错, 系统总能从错误状态恢复到正常状态, 但模块的切换需要一定时间。如果把多个模块中的一部分连接为表决器, 另一部分模块作为表决器的备份, 还可得到兼有静态冗余和动态冗余两种结构之长的混合冗余结构。

(2) 时间冗余

结构冗余利用多余的结构换取可靠性的提高，时间冗余则是以多花的时间为代价，来消除瞬时错误所带来的影响。其中中心思想是设置一个错误检测程序，当它检测到程序运行出错时，能发出一个错误恢复请求信号，使程序返回重新执行。

(3) 信息冗余

用于检测和纠正信息在传输或运算中发生的错误。常用的方法有奇偶码、循环码等误差校正码。其中中心思想是利用附加的冗余信息来校正可能出现的错误，其代价是增加系统的计算量和附加信息占用信道的的时间。

综上所述，无论采用哪种冗余技术，都是以额外的资源消耗换取系统的正常运行。这是实现容错软件的需要，与由于设计不当而造成的资源浪费显然不同。

3. 容错软件的设计

一般的说，设计一个容错软件需经历以下步骤：

- ① 通过常规设计获得系统的非容错结构。
- ② 分析在系统运行中可能出现的软、硬件错误，确定容错的范围。
- ③ 确定采用的冗余技术，并评估其容错效果。
- ④ 修改设计，直至获得满意的结果。

图 12.6 显示了设计容错软件的一般过程，以下再说明几点。

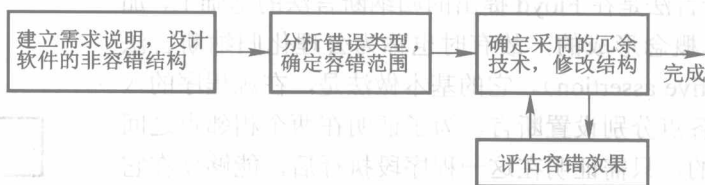


图 12.6 容错软件的一般设计过程

① 要选择好冗余单元。在通常情况下，硬件可选择功能级的部件为冗余单元，软件可以功能模块作为冗余单元。

② 冗余单元较小的系统，其可靠性一般较大。例如，图 12.7 (a) 所示结构的可靠性高于图 12.7 (b) 所示的结构。

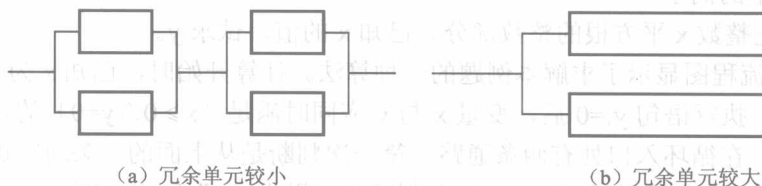


图 12.7 容错分支中的冗余单元

③ 强调容错软件中的各个冗余分支应由不同的开发小组独立进行设计。这些独立设计的分支除功能与结构要求一致外，编码与测试方法各不相同，潜在的错误也不会相同。因而在同样的输入条件下，引发相同错误的可能性也相对较小，从而可达到提高系统可靠性的目的。

12.4 程序正确性证明

迄今为止，程序测试仍然是提高软件可靠性的主要手段。但正如 Dijkstra 所指出的，测试只能说明程序有错，不能够证明程序不存在错误。所以从 20 世纪 60 年代以来，人们就寄希望于程序的正确性证明，希望研究出实用的技术与工具，来证明程序确能完成预定的功能。如果能实现这一目标，则测试工作量可以大大减轻，软件可靠性将更有保证，可靠性模型也就没有多大用处了。

程序正确性证明的基本思想，就是要通过数学的方法，证明程序具有某些需要的性质。通过许多人多年的努力，现已提出了一些有用的方法与技术，其中包括输入-输出断言法、最弱前置条件法、结构归纳法等较常用的方法。下面以输入-输出断言法为例，简单说明程序正确性证明的基本概念与步骤。

输入-输出断言法是在 Floyd 提出的归纳断言法的基础上，加上 Hoare 公理化概念形成的，故有时也称为公理化归纳断言法 (axio-matic inductive assertion)。它的基本做法是，在源程序的入口、出口和中间各点分别设置断言，为了证明在两个相邻点之间的程序段是正确的，只需证明在这一程序段执行后，能够使在它之前的断言变成其后一点的断言就可以了。如图 12.8 所示，S 代表程序中的一条或若干条语句，P 是 S 的前断言，R 是 S 的后断言。记号 $\{P\}S\{R\}$ 表示，如果在 S 执行以前断言 P 为“真”，则 S 执行以后断言 R 一定是“真”。例如在下式中

$$\{1 < i < N\} \quad i:=i+1 \quad \{2 < i < N+1\}$$

语句 $i:=i+1$ 执行后，确能由“前断言为真”导出“后断言为真”，故知这一语句是正确的。

下面请看一个简单的例子。

[例 12.1] y 是正整数 x 平方根的整数部分。已知 x 的值，试求 y 。

[解] 图 12.9 用流程图显示了求解本例题的一种算法。计算开始时，已知 x 为正整数，故断言 $\{x \geq 0\}$ 成立。执行语句 $y:=0$ 后，变量 x 与 y 应同时满足 $\{x \geq 0 \wedge y=0\}$ 的初始化条件。接着便进入循环。在循环入口处有两条通路。第一次判断是从上面的一条通路进入的。从断言 $\{x \geq 0 \wedge y=0\}$ 不难导出循环断言 $\{y^2 \leq x\}$ 。若判断的结果为“真”，则表示断言 $\{(y+1)^2 \leq x\}$ 成立，执行赋值语句 $y:=y+1$ 后，它的后断言 $\{y^2 \leq x\}$ 也必然成立。这表明，无论是第一次

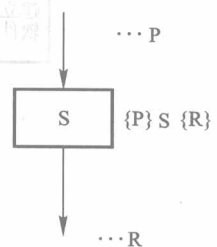


图 12.8 输入-输出断言

循环或之后的各次循环，循环入口的断言都是 $\{y^2 \leq x\}$ ，这就是所谓的循环不变式(loop invariant)。

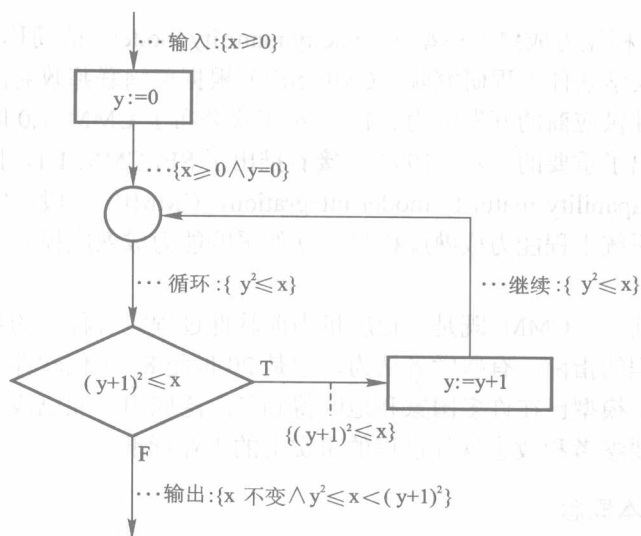


图 12.9 求平方根的整数部分的流程图

当判断 $(y+1)^2 \leq x$ 为“假”时，循环就结束了。此时的 x 值将小于 $(y+1)^2$ ，但仍然满足循环入口断言 $\{y^2 \leq x\}$ ，故 $y^2 \leq x < (y+1)^2$ 。另一方面， x 的值在程序执行过程中从未改变，所以出口断言可以写作 $\{x \text{ 不变} \wedge y^2 \leq x < (y+1)^2\}$ 。

考察题意，可知出口断言中的 y 确系题目要求的 y 值。且在程序执行中， x 的值保持不变。可见图 12.9 的程序是正确的，能够完成本例题所要求的功能。

为了方便正确性证明的推导和书写，Hoare 提出了一组公理化的证明规划，包括推论法则以及对赋值、复合、选择、循环等语句的证明规则，使这一方法更趋完善。但是，同任何数学证明一样，这种方法要求良好的数学基础和严密的推理，常常使一般程序人员望而生畏。加上步骤繁琐，为证明一个程序所写的推导，往往比这个程序本身还长。用手工方式来证明程序的正确性，除了对于很小的程序或大程序中的少量关键程序段以外，很难普遍推广。所以近十几年来，许多人致力于研究自动的程序正确性证明器。一些能验证小型 Pascal 和 LISP 语言程序的正确性证明器已经研制出来，并在使用中继续改进。

把程序正确性证明付诸实用，尤其是解决大型软件的正确性证明，还需要进行大量的工作。即使有了实用的自动证明工具，仍不能保证程序完全没有错误，因为数学证明本身就不可能保证无错。但不管怎样，大量的研究工作还在继续进行，而且它所包含的思想，包括一步步地证明程序具有完成某种功能的性质，正在启示越来越多的程序人员在建立程序的过程中就考虑程序的正确性，就像逐步细化技术所遵循的那样。

12.5 CMM 软件能力成熟度模型

CMM 是软件过程能力成熟度模型 (capacity maturity model) 的简称。20 世纪 80 年代后期, 卡内基·梅隆大学软件工程研究院 (CMU/SEI) 根据美国联邦政府的要求开始研究这一模型, 用于评估软件供应商的开发能力。1991 年正式公布了 CMM 1.0 版, 标志着软件质量管理向质量认证迈出了重要的一步。1993 年接着推出了 SEI CMM 1.1, 目前已经发展到能力成熟度模型集成 (capability maturity model integration, CMMI) 阶段, CMMI 框架包括软件能力成熟度模型、系统工程能力成熟度模型、软件采购能力成熟度模型、继承产品和过程开发等。

对于软件企业而言, CMM 既是一把度量当前软件过程完善程度的尺子, 也为软件机构提供了改进软件过程的指南。有些学者认为, 它是 20 世纪 80 年代软件工程最重要的发展之一。迄今为止, 这个模型已在许多国家和地区得到了广泛应用, 业已成为衡量软件公司软件开发管理水平的重要参考和改进软件过程的事实上的工业标准。

12.5.1 CMM 的基本概念

本节首先介绍与 CMM 相关的一些概念。

1. 软件过程

软件过程包括一个软件企业 (或软件项目开发小组, 下同) 在计划、开发和维护一个软件时所执行的一系列活动, 包括工程技术活动和工程管理活动。通常用软件过程能力来描述软件企业遵循其软件过程能够实现的预期结果; 用软件过程性能表示软件企业遵循其软件过程能够得到的实际结果。通过制度、标准和机构, 一个软件开发企业可以将其软件过程规范化和具体化, 从而不断提高软件过程的能力。

软件过程成熟度用于表达一个特定的软件过程被明确和有效地定义、管理、测量和控制的程度。软件过程成熟度高的软件企业, 对其管理和工程的方法、实践和规程等均有明确的定义, 不会因为人员的变化而随之发生变化。由于软件过程的改善是一个持续的过程, 为了刻画具体的成熟度, 可将软件过程的能力成熟度分成不同的等级, 每个等级包括一组过程目标。等级越高, 表示该软件机构的软件过程成熟度越高, 其软件过程能力也越高。

2. 关键过程域

所谓过程域, 是指互相关联的若干软件实践活动和有关基础设施的一个集合。为了确保不同等级的软件过程能力成熟度分别达到各自的目标, 特别要注意对于实现该等级的目标起关键性作用的过程域, 即关键过程域 (key process area, KPA)。对实施关键过程域起关键作用的措施、活动、规程以及相关基础设施, 可称为关键实践 (key practice)。关键过程域的目标, 就是通过它所包含的关键实践的实施来达到的。

3. CMM 模型

CMM 模型是用来确定一个软件过程的成熟程度以及指明如何提高过程成熟度的参考模型。它描述了软件过程从无序到有序、从特殊到一般、从定性管理到定量管理，直至最终达到动态优化的成熟过程，给出了不同成熟等级的基本特征和改进软件过程应遵循的原则与采取的行动。

12.5.2 软件能力成熟度等级

CMM 模型把成熟度等级分成 5 级，共包括 18 个关键过程域、52 个过程目标、316 种关键实践，从而为软件企业的过程能力提供了一个阶梯式的进化框架。每达到成熟度框架的一个等级，就建立起软件过程的一组相应成分，使软件机构的软件过程能力有一定程度的增长。表 12.3 显示了 CMM 的基本内容。

表 12.3 CMM 1.1 模型

过程能力等级	特 点	关键过程域
1 级：初始级	软件过程是无序的，对过程几乎没有定义，成功取决于个人的努力。管理是消防队救火式的	
2 级：可重复级	建立了基本的项目管理过程来跟踪费用、进度和功能特性。制定了必要的过程规则和纪律，能重复早先类似应用项目取得的成功	需求管理 软件项目策划 软件项目跟踪和监督 软件子合同管理 软件质量保证 软件配置管理
3 级：已定义级	已将软件管理和工程两个方面的过程文档化、标准化，并综合成该组织的标准软件过程。所有项目均使用经过批准、剪裁的标准软件过程来开发和维护软件	组织过程定义 组织过程焦点 培训大纲 集成软件管理 软件产品工程 组际协调 同行专家评审
4 级：已管理级	收集对软件过程和产品质量的详细度量，对软件过程和产品都有定量的理解和控制	定量的过程管理 软件质量管理
5 级：优化级	过程的量化反馈和先进的新思想、新技术促使过程不断改进	缺陷预防 技术变更管理 过程变更管理

由表 12.3 可知，软件过程成熟度 5 个等级的特点可以概括为：

① 初始级 (initial)。对软件过程缺乏明确的定义，质量管理处于消防式的、要等出了问

题再处理的状态。

② 可重复级 (repeatable)。本级主要解决质量管理体系从无到有的问题,重点是建立一套基本制度,使软件项目的基本管理可以重复实施。

③ 已定义级 (defined)。本级反映了质量管理从特殊到一般并进一步实现标准化和文档化的过程,可把质量管理再提高一个层次。

④ 已管理级 (managed)。本级反映了质量管理从定性管理提升到定量管理的过程,并通过定量控制,使质量管理的结果达到可以预测。

⑤ 优化级 (optimizing)。优化级是一个从静态管理到动态管理的过程,通过开发技术和过程的不断更新,使质量管理体系持续改进和提高。

上述的 5 个等级定义了一个有序的尺度,用来测量软件机构的成熟度和评价其过程能力。每一个成熟度等级均为过程的继续改进提供一个基础,5 个等级各有不同的行为特征及其关键过程域。

12.5.3 CMM 的应用

CMM 主要应用在以下两个方面:能力评估和过程改善。

1. 能力评估

CMM 的最初目的是评估软件开发机构的软件开发能力。有两种通用的评估方法可以评估机构软件过程的成熟度:软件过程评估和软件能力评价。

① 软件过程评估。用于确定一个机构执行软件过程的当前状态和机构在软件过程中面临的需要优先改善的问题,向机构领导层提供报告,以获得机构对改善软件过程的支持。软件过程评估集中关注机构自身的软件过程。评估的成功取决于管理者和专业人员对机构软件过程改善的支持,应在一种合作的、开放的环境中进行。

② 软件能力评价。用于识别或者监控软件承包商开发软件的过程状态。软件能力评价集中关注软件承包商在预算和进度要求范围内,高质量地完成软件产品合同的能力以及相关的风险,重点在于揭示机构实际执行软件过程的文档化的审核(或审计, audit)记录。评价应在一种审核的环境中进行。

2. 过程改善

软件过程改善是一个持续的、全员参与的过程。CMM 建立了一组有效地描述成熟软件机构特征的准则。该准则根据在软件工程技术和管理方面的优秀实践,清晰地描述了软件过程的关键域。企业可以有选择地引用这些关键实践来指导软件过程的开发和维护,不断地改善本机构软件过程,实现成本、进度、功能和产品质量等多方面的目标。

目前,欧、美等国的软件供应商与大型软件用户,共同采纳 CMM 作为供需双方的软件产品质量及工程预算标准。在亚洲,印度对 CMM 更是全力投入,每年定期进行 CMM 培训,在全球通过 4、5 两级 CMM 评估的软件企业中,印度占一半以上,其软件产品出口总值从 10 年前的 5 千万美元增长到 50 亿美元,预期 2008 年将达到 500 亿美元。现在印度约有 1 000 家软件企业,34 万从业人员,在《财富》杂志全球 500 强企业中,近半数为印度软件企业的

客户。据报道,1999 年全球共有 34 个国家的软件企业参加过 CMM 评估,其中 1 级占 43.1%,2 级占 34.2%,3 级占 17.3%,4 级占 4%,5 级占 1.4%。这些企业的规模,2 级为 25~100 人,3 级为 1 000~2 000 人,5 级多数在 5 000 人以上。

在我国,2002 年已有东大阿尔派、托普软件、华为印度研究所等多家软件企业通过了不同等级的 CMM 评估。上海市政府并颁布政策,凡通过 CMM 3~5 级认证的软件企业,可以获得 40 万~80 万的资助。从 2002~2005 年,上海市财政和企业为 CMM 领域共同投入了 1.5 亿元资金。2002 年,该市仅仅有一家企业通过 CMM 3 级认证,到 2005 年 11 月,已有 55 家企业通过 CMM 3 级或以上的评估,从而使上海软件企业的管理水平走在全国的前列。

12.5.4 CMM 评估的实施

CMM 评估要遵循 SEI 的 CAF (CMM assessment framework) 规范,由 CMU/SEI 授权的主任评估师 (lead assessor) 领导一个评审小组进行。评估过程包括员工培训 (企业的高层领导也要参加)、问卷调查和统计、文档审查、数据分析以及与企业的高层领导进行讨论等。评估结束时撰写评估报告,由主任评估师签字后生效。

目前使用的 CMM 评估方法主要有两种,一种称为 CBA-SCE (CMM-based appraisal for software capability estimation),主要用于 CMM 对机构的软件能力进行评估,由机构外部的评估小组对该机构的软件能力实施评估。另一种称为 CBA-IPI (CMM-based appraisal for internal process improvement),主要用于 CMM 对内部的过程改进实施评估,由机构内部和外部人员联合组成评估小组,针对软件机构本身进行评估。评估的结果归机构所有,以引导机构不断改进质量。

上述两种评估均由 CMU/SEI 授权的主任评估师领导,根据 CMM 模型来进行,都要审查正在使用和将来使用的文件/文档,并对不同的机构员工进行采访。两种评估的结果应该是一致的,评估结果的所有资料都应呈报给 CMU/SEI。

12.5.5 软件过程评估的 SPICE 国际标准

随着国际社会对软件过程评估需求的增长,国际化标准组织于 1995 年制定了软件过程评估的试用标准。1996 年公布了软件过程评估 (software process assessment, SPA) 标准的 1.0 版,并开始了第二批试用。1998 年发表了试用报告 ISO/IEC TR 15504 SPICE (software process improvement and capability determination),于 2001 年产生正式国际标准。

SPA 提供了一个软件过程评估的框架。它可以用于软件的设计、管理、监督、控制以及提高对软件开发、升级和支持的能力。图 12.10 显示了 SPA 所包含的 9 个组成部分。

随着网络应用的普及,软件市场的需求日新月异,而 CMM/SPICE 方法需要严格的纪律和大量的文档,使得软件人员常常觉得负担过重,难以适应快节奏的软件开发要求。为了满足市场竞争的需要,一批新的开发过程与方法又应运而生,其中比较引人注意的有极值程序设计 (extreme programming, XP)、SCRUM 软件开发过程、自适应软件开发 (adaptive software

development, ASD) 等。它们被统称为轻载 (light weight) 方法, 而前述的 CMM/SPICE 方法和下节即将介绍的 ISO 9000 标准则称为重载 (heavy weight) 方法。本书对轻载方法不再介绍, 有兴趣的读者可参阅相关文献, 如本书主要参考文献 6。

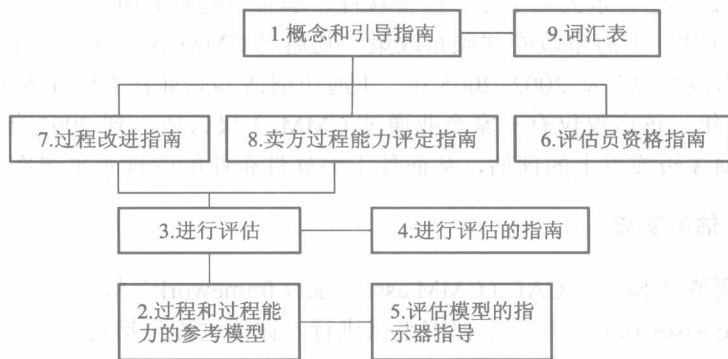


图 12.10 SPA 的组成部分

12.6 ISO 9000 国际标准

ISO 9000 国际标准是由国际化标准组织在研究欧洲和美国的质量管理标准的基础上, 于 1987 年 3 月正式公布的一组质量系列标准。到 1997 年, 全世界已经有 100 多个国家和地区据此开展了 ISO 9000 质量认证活动。

ISO 9000 由 5 个相关的标准组成。它们是:

- 质量术语标准 (ISO 8402—1994)。
- 质量保证标准 (ISO 9001 等)。
- 质量管理标准 (ISO 9004-1 等)。
- 质量管理和质量保证标准的选用和实施指南 (ISO 9000-3 等)。
- 支持性技术标准 (ISO 10005 等)。

这些标准适用于各种行业的不同工业活动, 其中为质量保证而制定的 ISO 9001 标准最适合于用作软件开发的标准。针对软件这一特殊行业, ISO 随后又公布了 ISO 9000-3, 作为将 ISO 9001 运用于软件的实施指南。

12.6.1 ISO 9001 和 ISO 9000-3

ISO 9000 标准将供需双方称为卖方和买方, 提供产品或服务的一方是卖方, 购买者或消费者是买方。标准规定了对卖方的质量要求和质量管理办法, 如果卖方认真按照标准的要求组织生产, 经过权威机构的审核并取得认证, 就会赢得买方的信任; 买方凭认证情况在市场

上选购产品，也就不必担心质量检验问题了。前面已经提到，ISO 9001 和 ISO 9000-3 都是适用于软件开发机构的质量标准。

ISO 9001 包括了设计、开发、生产、安装和服务等活动的质量保证模式，于 1994 年由国际标准化组织公布。与之相对应的国家标准是 GB/T 19001—1994。该标准中规定了质量体系 20 个方面的质量要求，覆盖了全部设计和开发活动。如果软件产品开发机构能够达到这些要求，表明它具备了质量保证能力。ISO 9000-3 的全称为“质量管理和质量保证标准：第三部分——ISO 9001—1994 在计算机软件开发、供应、安装和维护中的使用指南”。它从软件的角度，对 ISO 9001 的内容给出了具体的解释和说明，例如，结合 ISO 9001 中的 20 个质量要素，ISO 9000-3 逐一作出了针对软件产品开发的解释。

虽然 ISO 9001 标准和 CMM 都是质量标准，但 CMM 的要求与 ISO 9001 标准不尽相同。有些 ISO 9001 的要素可以在 CMM 中找到完全对应的部分，另外一些要素则仅有比较分散的对应。两者有很多相似之处，最大的相似在于两者都强调“该说的要说到，说到的要做到”；对于每一个重要的过程应形成文件，包括指导书和说明，并检查交货的质量水平。此外，CMM 强调持续改进，ISO 9001 的 1994 版标准说明的则主要是“合格质量管理体系的最低可接受水平”，不过 ISO 9001 的 2000 版标准也增加了持续改进的内容。

12.6.2 ISO 9000 标准对软件企业的重要性

ISO 9000 质量标准最初是针对一般制造企业的，但迅速崛起的软件企业及其产品却总是不能真正满足用户的需求。因而软件开发机构同样需要建立并实施 ISO 9000 质量标准。

软件是不可见的复杂的逻辑实体，其需求很难精确把握，加上其开发活动大多由手工完成，因而软件产品或多或少存在一定的质量缺陷。解决这一问题的手段有两个：技术手段和管理手段。

技术手段有两个方面，一是改进测试方法、提高测试效率，更有效地发现和排除软件开发过程中发生的各种差错，提高软件质量；二是改进开发过程，使各种差错不会或很少引入软件的开发过程。但是实践证明，从技术上解决软件质量问题的效果并不明显。虽然不断有新的测试技术和工具出现，但测试的有效性没有根本的变化；第 2 章介绍了各种软件过程模型，特别是净室模型，其基本思想就是在整个开发过程中不让差错或缺陷进入，但其前提是软件的需求要用形式化方法方便地描述出来，然后像公式推导一样进行严谨的开发，可是这些都过于理想化了，其效果并不十分理想。

管理手段就是加强软件开发过程中的质量管理。通过质量管理相对地净化环境，使得更少的差错或缺陷进入开发过程，从而减轻查错和排错的工作量，提高软件质量。这就是 ISO 9000 质量管理和质量保证标准的基本思想。它吸取了许多国家多年积累的管理经验，把与企业质量相关的问题集中到管理职责、质量体系、合同评审、设计控制、采购、检验、交付、培训及服务 20 个要素中，要求明确，具有很好的可操作性。企业遵照这些要求，就可结合自己的具体特点建立质量保证体系，贯彻实施这些质量要求，做到凡事有章可循，凡事有据

可查，凡事有人负责，凡事有人监督，从而获得较好的质量管理效果。

12.6.3 在软件企业中实施 ISO 9000 标准

软件企业按照 ISO 9000 标准建立质量保证体系，进行质量体系认证，有一系列的工作要做。一般的说，推行 ISO 9000 有如下 5 个必不可少的过程：

知识准备—立法—宣传—执行—监督、改进

将上述过程进一步细化，便可得出软件企业推行 ISO 9000 所需的详细步骤，以下列出的是一组典型的步骤。

- ① 识别原有的质量体系，找出缺陷。
- ② 任命管理者代表，组建在本企业推行 ISO 9000 的机构。
- ③ 制定目标及激励措施。
- ④ 对各级人员进行必要的管理意识和质量意识训练，特别是 ISO 9001 标准知识培训。
- ⑤ 编写质量体系文件（立法）。
- ⑥ 广泛宣传质量体系文件，正式发布、培训和试运行。
- ⑦ 训练内审人员。
- ⑧ 进行若干次内部质量体系审核。
- ⑨ 在内审的基础上对管理者进行评审。
- ⑩ 完善和改进企业的质量管理体系。
- ⑪ 申请认证。

需要指出，企业实施 ISO 9000 标准绝不是一件容易的工作，必须在原有的质量体系基础上，投入必要的人力、物力和财力，不断改进和优化质量管理，才能真正达到目标。既要避免干干停停，有始无终；也不必重起炉灶，把原来行之有效的质量管理制度完全推翻。

12.7 软件度量

软件度量是质量管理的一种重要手段，在 ISO 9000-3 标准中，已明确指出了软件度量的重要性。虽然它目前还在发展之中，但已经受到越来越多的软件企业的重视。

本节介绍的软件度量可划分为软件项目度量与软件过程度量两大类。前者为战术性的活动，目的在于改进软件产品的质量；后者为战略性的活动，目的在于改进企业的软件开发过程，提高整个过程的质量。

12.7.1 项目度量

1992 年，S. R. Schach 把软件项目度量概括为 5 种基本度量。1993 年，美国空军发表了一份称为 U.S. Air Force Acquisition Policy 93M-017 的报告，也提出了与之十分相似的项目度量内容，如表 12.4 所示。

表 12.4 项目度量的基本度量

S.R.Schach,1992	U.S.Air Force, 93M-017 报告	常用单位
size (规模)	size (规模)	LOC, KLOC
effort (工作量)	effort (工作量)	人-月
duration (时间)	schedule (进度)	月
quality (质量)	quality (质量)	错误数/KLOC
cost (成本)	rework (返工)	元

1. 项目度量的内容

在表 12.4 中, 以代码行 (lines of code, LOC) 表示的软件规模是最基本的度量。它直接关系到软件的成本、开发工作量和完成时间。在项目管理中把规模、成本、工作量和时间均列为基本度量, 理由是显而易见的。只谈产量不讲质量显然是没有意义的, 所以软件质量也是一项重要的基本度量, 通常以每千行代码中存在的错误数来衡量。例如, 若某一项目组每月可生成 2 000 LOC, 但其中半数代码因质量问题不能使用, 如不加以说明, 这一度量数据就毫无价值。

在表 12.4 中, 软件的规模、成本和工作量通常应分阶段进行度量。在例 11.2 中, 曾详细介绍过在不同的开发阶段连续对软件规模、成本和工作量进行估算的方法。把估算值与项目结束后的实际统计值比较, 就可以知道估算方法的可信度; 把实际统计值与工业界开发同类软件的平均值作比较, 将帮助管理人员衡量本单位的技术与管理水平, 找出本单位在软件开发中存在的问题, 以及解决问题的策略。

在前面介绍的项目度量中, 所有的基本度量都是以代码行为基础的。例如,

$$\text{软件成本(元)} = \text{LOC(行)} \times \text{每行代码的成本(元/行)}$$

$$\text{开发工作量(人-月)} = \text{LOC(行)} / \text{每人-月开发的代码行(行/人-月)}$$

等等 (参阅例 11.2)。由于 LOC 被用作程序长度的规格化单位由来已久, 已为大量文献和软件估算模型 (参阅第 11 章) 所采用, 所以很多软件企业乐意使用这种度量方式。它的缺点是: 依赖于程序设计语言; 特别对于使用 4GL 等非过程化语言编码的紧凑的小程序, 会产生不利的度量数据, 与其他语言编码的软件缺乏可比性; 在开发早期 (分析或设计阶段) 估算出来的 LOC, 其偏差可能较大等。于是, 一种新的、面向功能 (function-oriented) 的项目度量方式便应运而生, 并将上述的基于 LOC 的度量称为面向规模 (size-oriented) 的项目度量。

2. 面向功能的项目度量

功能度量方式是在 1979 年由 A. J. Albrecht 首先提出的。其中心思想是, 任何软件都包含若干种功能, 每种功能又包含具有不同复杂度的若干个功能点 (function points)。因此, 软件的规模也可用功能点数量的多少来表示, 以代替原来常用的 LOC 表示法。

表 12.5 是 Albrecht 建议的功能点加权计算表。表中把功能点划分为 5 种类型, 依次为用

户输入、用户输出、用户查询、主文件处理和外部界面。每类功能点按照其复杂程度可区分为简单、平均及复杂 3 种等级。例如,若某个软件分别包含 inp 个输入项, out 个输出项, inq 个查询项, mf 个主文件项和 inf 个外部界面,而且所有的功能点全都属于平均复杂程度,则该软件的加权功能点数可按照下式进行计算

$$FP = 4 \times inp + 5 \times out + 4 \times inq + 10 \times mf + 7 \times inf$$

在一般情况下,

$$FP = \sum \sum C_{ij} W_{ij} \quad (12.8)$$

其中 C 表示加权系数 (weighting coefficient), W 表示功能点个数, i 代表功能点类型, j 代表复杂度等级。

表 12.5 功能点加权系数表

功能点类型	简单	平均	复杂
用户输入 (user input)	3	4	6
用户输出 (user output)	4	5	7
用户查询 (user inquiry)	3	4	6
主文件处理 (master file)	7	10	15
外部界面 (external interface)	5	7	10

通过式 (12.8) 计算出来的 FP 值虽然使用了加权系数,仍不能适应各种不同软件千差万别的情况。为了使度量的结果更加准确,Albrecht 进一步提出用技术复杂性因子 (technical complexity factor, TCF) 对 FP 进行调节的方法,即

$$FP = FP(\text{调节前计算值}) \times TCF \quad (12.9)$$

$$TCF = 0.65 + 0.01 \times \sum F_i \quad (12.10)$$

其中 F_i 代表了包括处理能力、响应时间、有无数据通信、是否支持分布式处理等在内的 14 种技术因素,每种因素按对于开发难度的影响赋予 0 (代表没有影响) 到 5 (代表很大影响) 的一个数值。由此可见, $\sum F_i$ 的值可以在 0~70 之间变化,TCF 可以在 0.65~1.35 之间变化。当根据式 (12.9) 计算出 FP 的最终值后,就可像面向 LOC 的方法一样,计算出项目软件的其他属性,例如,每个功能点的错误数,每个功能点的成本,每人-月可以完成的功能点数量等。

根据 Albrecht 与 Jones 等人进行的试验,面向功能度量的结果往往较面向规模度量的结果更加准确。但无论加权系数还是技术复杂性因子的确定,都是与度量者经验有关的主观行为,可能导致度量的误差。

12.7.2 过程度量

如果说项目度量是对项目组中所有个人开发工作的测度 (measurement),则过程度量可以认为是对整个企业中全体项目组开发能力的衡量。Pressman 曾用两句简明的话说明了二者

的关系：把对于项目组中个人的度量组合起来，可形成对项目的度量；把所有项目组的项目度量组合起来，就形成了对整个企业的过程度量。

因此，过程度量实际上是向企业高层管理者提供的有关企业软件开发质量的状态信息。这些信息涉及企业产品、过程和资源在不同开发阶段的状态。把这些动态的、连续追踪的状态信息与软件开发前的计划数据相比，就可帮助管理者发现问题，找出改进过程的依据。

在结束本节介绍以前，需要再补充几点：

① 软件企业的高层领导应该重视收集项目度量的测量数据，及时综合企业最新的过程度量数据。

② 同一企业的所有项目组，在项目度量中应采用相同的规格化手段（如面向规模或面向功能），使不同项目组的测度数据具有可比性。

③ 过程度量的目的是为了改进企业的过程，二者应紧密结合。在 12.5 节介绍的 CMM 中，把软件成熟度的 5 个等级中的第四级称为已管理级，强调达到该级的软件企业，对软件产品和过程都应设置定量的质量目标，对所有项目的重要软件过程活动，都要测量其生产率和质量，足见 CMM 对过程度量是十分重视的。

④ 除了项目度量与过程度量外，对产品的质量也可以进行度量，称为产品度量。限于篇幅，就不详细说明了。

小 结

本章从运行、维护与软件移植等 3 方面介绍了软件的质量属性，强调指出质量保证是贯穿于整个生存周期的重要活动。无论是阶段复审、配置控制，还是在分析、设计、编码与测试中坚持遵守软件的开发规范，都表明质量保证必须从头做起，贯彻始终。V&V 是 TQC 思想在软件生产中的具体体现。它简明地提示我们既要把好最后的确认关，也要重视平时的各种验证。

可靠性在质量属性中占有主要的地位，它既可表示为一定时间内软件正常运行的概率，也可表示为软件的平均无故障时间。相同规模的软件，可靠性的要求越高，其生产率就越低，所以软件可靠性应根据实际需要确定，并非越高越好。从根本上说，可靠性又可分为避错和容错两类技术。本章除了介绍几种典型的可靠性预测模型外，还简单介绍了软件容错的概念与方法，以及保证可靠性的形式化方法——程序正确性证明的基本概念。

质量是企业生命。质量认证把质量管理从单个产品扩展到整个企业，使企业更上一层楼。20 世纪 90 年代以来，一批质量认证方法如 ISO 9001、CMM/SPICE 相继出现，为软件企业评估和改进其软件过程提供了可持续操作的标准框架，具有战略上的重要性。有人将这些发展称为软件过程工程，甚至看作继面向对象软件工程之后的新一代软件工程。近几年来出现的“轻载”方法适应了快节奏的软件开发要求，也值得引起注意。

作为质量管理的一种重要手段，软件度量正在迅速获得应用。本章简介了项目度量与过

程度量的做法与作用。

习 题

1. 如果要为某大型软件编写一份“质量保证计划”，你认为应包含哪些内容？
2. 根据本章和第 9 章、第 12 章中有关配置管理的内容，写一份关于软件配置管理的综合介绍。
3. 解释下列各对名词，并说明它们的相互关系和差别：
 - (1) 验证与确认。
 - (2) 软件保证与软件认证。
4. 什么是软件可靠性？怎样对它进行定量表示？
5. 可靠性模型有八大类？它们的主要区别是什么？
6. 简述软件容错技术的基本原理与方法。
7. 按照 Hoare 的记号 $\{P\}S\{R\}$ ，填充下表中的空格。

P	S	R
$X > 3$	$X := X + 2$	$X > 5$
$F(X) = W$	$Y := F(X)$	$Z = 5$
$x + y = 5$	$y := y + a$	$(y = (x + 2)^2) \wedge (a = 2x + 3)$

8. 什么是 CMM 软件过程能力成熟度模型？它有哪些应用？
9. CMM 有哪 5 种等级？简述它们的含义。
10. 软件认证有哪些国际标准？
11. 为什么说从软件保证到软件认证是一次飞跃？
12. 试述软件度量的重要意义。

第 13 章 软件工程环境

软件工程环境是软件工程学的组成部分，也是实现软件生产工程化的重要基础。“工欲善其事，必先利其器”，在软件开发中，无论技术活动与管理活动，都离不开环境（包括工具）的支持。近 20 多年来，各技术先进国家大力开展软件环境的研究，计算机辅助软件工程（computer-aided software engineering, CASE）、集成项目支持环境（IPSE）等课题，始终都受到人们的关注，一大批实用的环境应运而生。这些环境建立于现代软件开发的基础上，反过来又促进了现代方法的推广与流行，不仅提高了软件的生产率，而且逐渐影响和改变着软件的生产方式。本章将简要叙述软件工程环境的变迁、现状和发展趋势，使读者进一步学习和研究软件工程环境的意义。

13.1 什么是软件工程环境

“环境”一词，对不同的用户往往具有不同的含义。对于不从事软件开发的最终用户（end-user）来说，环境就是他运行程序所使用的计算机——由硬件和操作系统所组成的虚拟机。这类用户对环境的要求，主要是运行可靠、操作容易，便于掌握和使用。对于开发者来说，环境是他们进行开发活动的重要舞台。在软件工程时代，开发者要求环境支持他们按照软件工程的方法，全面完成生存周期中的各项任务。通常把这种开发环境称为软件工程环境，而把前一类环境称为运行环境或工作环境。

具体而言，软件工程环境是指支持软件产品开发、维护和管理的软件系统，它在统一的集成机制下由一系列软件工具组成。这些工具对与软件开发相关的过程、活动和任务提供全面的支持，从而大大提高软件产品的生产效率和软件产品的质量，降低软件开发、维护和管理的成本。这类环境通常都有一套包括数据集成、控制集成和界面集成的集成机制，让各个工具使用统一的规范存取环境信息仓库，采用统一的用户界面，同时为各个工具或开发活动之间的通信、切换、调度和协同工作提供支持。

本章主要讨论开发环境，下面先介绍开发环境的特点。

13.1.1 软件开发环境的特点

软件开发由来已久。一台宿主机，一个编译（或汇编）程序，加上编辑、连接、装入等少量实用程序，就构成了早期软件开发的舞台。从这个意义上讲，在软件工程方法出现以前

就有了环境。只不过早期环境的软、硬件资源都有限，软件生产效率和开发质量都比较低罢了。

随着软件工程方法的流行，人们研制了大量软件工具，用来支持软件开发中的各种方法与技术。从少量零散的工具到配套成龙的工具箱，不仅意味着功能的扩大（从支持个别阶段到支持整个生存周期，从只支持开发技术到全面支持技术与管理），也反映了集成化程度的增加。20 世纪 80 年代初期，欧洲已出现了集成化项目支持环境的提法，并以此来概括按照软件工程的要求设计的、能支持整个软件生存周期且具有统一用户界面的一体化的软件开发环境。20 世纪 80 年代后期，CASE 和集成 CASE 在美国乃至全球迅速流行。20 世纪 90 年代初期，随着面向对象方法发展并应用到软件开发过程的各个阶段，又出现了支持面向对象开发方法的集成型软件工程环境。以下将从用户界面和环境工具两个方面，进一步说明软件开发环境的特点。

1. 友好和统一的用户界面

友好的用户界面，既能为使用带来方便，又有助于提高操作者的效率。现代的开发环境常采用以下的技术来改善用户接口的友好性。

① 具有弹出（pull-out，亦称下拉，pull-down）功能的多级菜单。这种菜单将环境的各种功能组织成命令树的形式。无论是选择当前菜单的功能，还是弹出下级菜单或返回上级菜单，都只需简单的键盘操作即可实现。具有高分辨率图形显示器的环境，还可用图标（icon）和鼠标器（mouse）补充传统的菜单技术，进一步减少击键的次数。

② 屏幕提示和在线帮助（on-line help）技术。前者用于告知用户当前可选择的操作或需要注意的事项，后者则根据用户的请求，提供即时的在线帮助。

③ 采用多窗口（muti-windows）技术。除常用的显示菜单/图标的窗口外，还可以按照用户的选择，在屏幕上随时开辟多种窗口（如编辑窗口、执行窗口、对话窗口或其他窗口）以便用户在同一时间内监控或处理多个不同的任务。

④ 采用向导（wizard）技术。将完成某组动作的步骤组合起来，用户执行这组动作时，只要从一个入口进入后，按提示进行简单的“上一步”、“下一步”操作即可。

菜单、帮助（含提示）、多窗口和向导，也被称作用户界面的 4 大友好技术。它们不仅适用于软件开发环境，也适用于其他的应用软件。

除友好性以外，开发环境还十分重视用户界面的一致性（unification）。

2. 集成化的软件工具

集成化就是一体化。采用集成化工具的最终目的，就是实现开发活动之间的全自动切换，不再需要用户的干预。

在早期的编程环境中，各种工具孤立地完成各自的任務。凡是调试过程序的读者都知道，如果在编译中发现了程序有错误，首先要退出编译，重新调用编辑程序。待程序修改后，再重新调用编译。若再次发现错误，又要再重复上述的过程。仅是编译和编辑这两种工具之间的来回切换，就不胜其烦，而且花费开发者许多宝贵的时间。20 世纪 70 年代出现的工具箱，

能部分实现从一个工具到另一个工具的切换。仍以程序的调试为例，当编译中发现错误时，开发环境能自动调出编辑程序，并且在源程序出错的地方发出某种信号（例如光标闪烁），提示开发人员进行修改。只此一点，就可以有效减少调试时间，提高开发效率。

集成化工具的使用则更胜一筹。它要求在同一开发阶段或不同阶段的有关工具之间实现完全的自动切换。工具的集成化主要包括数据集成、界面集成、控制集成和过程、平台等其他方面的集成。

① 数据集成。所有的工具统一建立在公共的文件库或信息库之上。每一运行中的工具从库获得必要的信息，又随时将结果放回库中，用库内的数据作为相互通信的媒介实现各个工具之间的数据交换。

② 界面集成。各工具使用统一或一致的用户界面，采用公共的交互方式。这种集成又可分为 3 个层次：窗口系统集成、命令集成和交互集成。窗口系统集成指工具采用同一种窗口管理系统，各窗口有相同的外观和操作命令；命令集成指工具对相似的功能采用相似的命令；交互集成指工具对相似的功能采用同样的交互方式，即同样的动作可以达到同样的效果。

③ 控制集成。能支持环境中的一个工具控制另一个工具，如启动、中止某个工具，或调用某个工具的特定服务。

④ 过程集成。这类环境了解软件过程中各个活动的时序、约束等，能主动地调度这些活动，保持它们合适的顺序。

⑤ 平台集成。指工具运行在相同的硬件/操作系统下。

13.1.2 理想环境的模型

在《软件工程环境：概念与技术》一书中，R. N. Charette 将软件开发环境抽象为一个模型，并把开发环境定义为“生产一个软件系统所需要的过程（process）、方法（methods）与自动化（automation）”。他认为，开发环境模型应由上述的 3 个层次由底向上地构成，并将“理想的”开发环境描述为“全程由充分自动化了的方法所支持的完整的过程模型（process model）”，如图 13.1 所示。按照 Charette 的观点，建立一个开发环境首先要确定一种开发模型，提出成套的、有效的开发方法，然后在这一基础上，利用各种手段（例如软件工具）实现开发活动的完全自动化。

Charette 在他的环境模型中并没有包括硬件，这是为了强调软件在环境组成中的特殊地位。相同的硬件，使用不同的操作系统、环境信息仓库和工具集，可能构成相差悬殊的不同的环境。这也说明，在选择环境的软、硬件配置时，尤其要重视软件的配置。

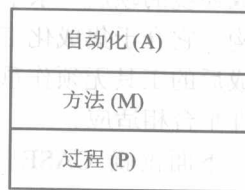


图 13.1 理想环境模型

13.1.3 CASE 环境

用 CASE 一词来描述软件环境，最早出现在 1978 年的第三届国际软件会议（ICSE）的一篇论文中。到 1986 年 9 月 24 日，美国《华尔街日报》首次将它作为一般技术名词使用，

之后 CASE 作为软件环境的称呼迅速流行起来，现已成为一切现代化软件开发环境的总称。CASE 环境、CASE 工具乃至集成 CASE (integrated CASE, I-CASE)，几乎都成了软件工程环境的代名词。除了 CASE 外，文献上对软件工程环境还有过许多不同的称呼，如软件开发环境 (software development environment, SDE)、程序设计支持环境 (programming support environment, PSE)、软件支持环境 (software support environment, S²E)、集成化项目支持 (integrated project support environment, IPSE) 等。

CASE 环境的总目的，是通过一组集成化的工具，帮助软件开发人员实现各项活动的全部自动化，帮助保证在软件产品的整个生存周期中的质量，借以提高开发和维护的质量和生产率。下面将进一步介绍 CASE 环境的组成与结构。

13.2 CASE 环境的组成与结构

如前所述，这里的 CASE 环境是一个总称，也代表了 IPSE 和 I-CASE 环境。这里先谈谈 CASE 环境的一般组成。

13.2.1 CASE 的组成构件

Pressman 把 CASE 环境的构件 (building blocks) 归纳为 6 种成分，如图 13.2 所示。这 6 种成分又可区分为 3 个层次。由硬件平台和操作系统 (包括网络和数据库管理系统) 组成的体系结构，是 CASE 环境的基础 (底层)。集成化框架 (integration framework) 由一组专用程序组成，用于建立单个工具之间的通信，建立环境信息仓库，以及向用户 (在这里是指软件开发人员) 提供外观和感觉一致的界面。它们将 CASE 工具集成在一起，构成环境的顶层。余下的一层是服务于可移植性的机构。它介于集成化工具与基础软、硬件之间，使集成后的工具无须作重大的修改即可与环境的软、硬件平台相适应。

下面仅就 CASE 工具和环境信息仓库作简要说明。

1. CASE 工具

所谓 CASE 工具，泛指用于辅助软件开发、运行、维护、管理、支持等过程中的活动的软件。自软件工程出现以来，许多学者提出了不同的软件开发模型、开发方法和软件管理的方法，只要出现一种新模型、新方法，辅助它们的软件工具也会随之出现。大多数工具仅限于支持软件生存周期过程中的某些特定的活动。可支持软件开发、维护和管理等过程中各种活动的软件工具有很多，例如，支持需求分析活动的的需求分析工具，支持分析设计建模的建

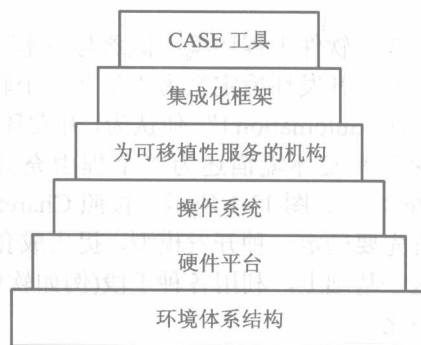


图 13.2 CASE 环境的构件

模工具，支持测试活动的测试工具，支持维护过程的维护工具等；也有支持软件开发方法的软件工具，如支持结构化方法的结构化工具、支持面向对象方法的面向对象工具、支持原型开发方法的原型工具等；还有应用类的工具，如用户界面工具、多媒体开发工具、数据库应用工具等。

CASE 环境中的工具集，应该包括支持软件开发自动化的各种工具，而且工具集是可扩充的，一种新的工具出现后，应该可以很容易地集成到工具集中。

CASE 工具种类繁多，很难有一种统一的分类方法，一个工具往往对软件生存周期中的某个（些）活动提供支持，所以有的学者按软件过程的活动将其划分为以下 3 类：支持软件开发过程的工具，包括需求分析工具、软件设计工具、编码工具、测试工具和纠错工具等；支持软件维护的工具，包括版本控制工具、文档分析工具、开发信息库工具、逆向工程工具和再工程工具等；支持软件管理过程和支持过程的工具，主要包括项目管理工具、配置管理工具和软件评价工具等。

2. 环境信息仓库

工具的集成化，需要有环境信息仓库（CASE repository）的支持。环境信息仓库向所有的工具提供统一的公共数据。对于大型的开发项目，环境信息仓库常常要存放项目的大量文档，不仅信息量大，而且在工具切换时要频繁地进行信息交换。环境信息仓库处于核心层内，可以说是核心层的核心。所以有人说，没有有效的环境信息仓库，就没有 CASE 环境。

环境信息仓库具有类似数据库管理系统的功能，Forte 在 1989 年定义了它应有的功能：

① 数据完整性。包括确认信息仓库的数据项，保证相关对象间的一致性，以及当对两个对象修改时，也要对其相关对象进行某些修改，并自动完成“关联式”修改等功能。

② 信息共享。提供在多个开发者和多个工具间共享信息的机制，管理和控制对数据及加锁/未锁对象的多用户访问，以使得修改不会被相互间不经意地覆盖。

③ 数据-工具集成。建立可以被 I-CASE 环境中所有工具访问的数据模型，控制对数据的访问，以及完成合适的配置管理功能。

④ 数据-数据集成。数据库管理系统建立数据对象间的关系，使得可以完成其他功能。

⑤ 文档标准化。在数据库中对象的定义直接导致了创建软件工程文档的标准方法。

13.2.2 CASE 的一般结构

集成 CASE 环境的优势，在于使软件项目的模型、程序、文档和数据信息平滑地从—个工具传递到另一个工具，从一个软件工程阶段平滑过渡到下一个阶段。为此，前节（第 13.2.1 节）介绍的组成 CASE 环境的各构件，在集成 CASE 环境中必须有机地结合在一起，以构成层次式的环境体系结构。

1. CASE 集成框架的典型结构

图 13.3 显示了 CASE 集成框架的层次结构模型。这是 Sharon 和 Bell 在 1995 年提出的，其层次具有一定的代表性。它把前面提到的构件分成 4 个层次，从上到下分别是：用户界面

层、工具层、对象管理层和共享中心库层。

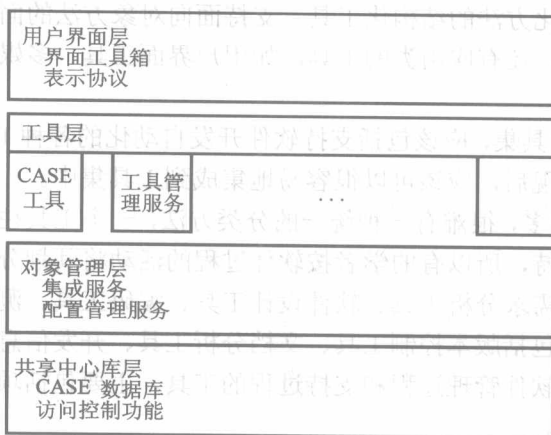


图 13.3 集成框架的层次结构模型

最上层的是用户界面层，它包括标准的界面工具箱和公共的表示协议。界面工具箱包含人机界面管理软件和显示对象库，它们为集成环境显示风格一致的用户界面提供了必要的界面元素和工具。表示协议提供一组界面约定，包括一致的屏幕布局约定、菜单名和组织、图符、对象名、键盘和鼠标的使用等。

工具层除了 CASE 工具本身外还包括一组工具管理服务。工具管理服务(TMS)负责管理工具的执行中多任务操作，包括同步和通信、协调从中心库和对象管理系统到工具的信息流以及收集关于工具使用的度量等。

对象管理层(OML)完成集成服务和配置管理功能。框架体系结构层的软件提供了 CASE 工具集成的机制。每个软件工具被插入到对象管理层，并通过一组工具和中心库耦合在一起。OML 的配置管理服务用于标识配置对象，完成版本控制，并提供对变化控制、审计以及状况说明和报告的支持。

最底层的是共享中心库层，它使得对象管理层能够与 CASE 数据库交互并完成对 CASE 数据库的访问控制。

2. CASE 结构的一个例子

为了使读者有一个形象的概念，现给出一个 CASE 结构示例，它由核心层、基本层、应用层 3 级组成，如图 13.4 所示。

最里面的一级是核心层。它包括由宿主机和操作系统构成的虚拟机，环境信息仓库（或文件库）及其支持软件（如数据库管理系统或文件管理系统），以及从工具到系统（含数据库）的接口。环境信息仓库是这个核心层的核心。它既要存放项目的各种开发文档与管理文档，又要承担各种工具之间的信息交换。对于大型的开发项目，它不仅信息量大，而且信息传送频繁。所以怎样实现环境数据库，并对它进行有效的管理，是多年来环境研究的中心课题，

至今仍为许多单位研究的重点。

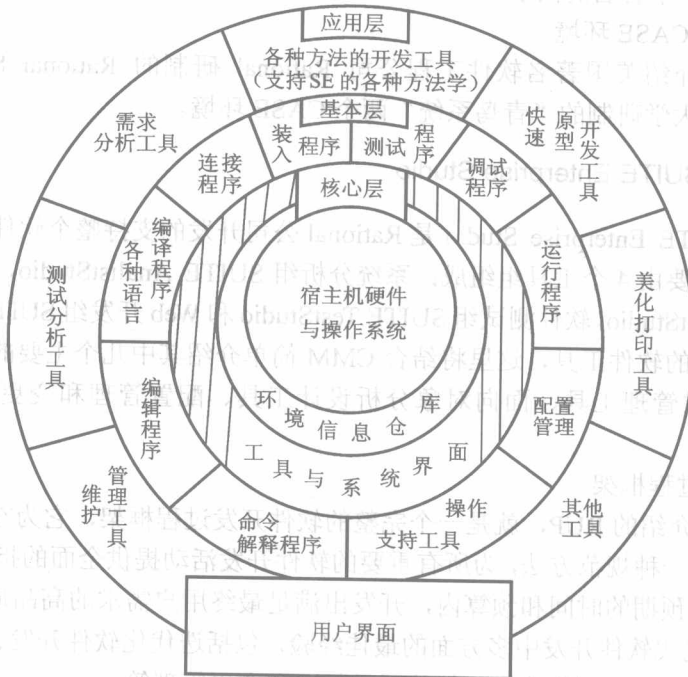


图 13.4 CASE 结构示例

第二级是基本层。它包括软件开发所需要的最小限度的工具，所以又称为最小支持环境。这些工具除包括常用的编译、编辑、调试、连接/装配、配置管理等实用程序外，通常还包括命令解释程序和支持用户操作的一些工具，例如菜单生成系统、图形系统和多窗口系统等。不同单位研制的环境，其基本层的内容略有差别，但大同小异。

第三级为应用层，包括支持生存周期内各个阶段活动的工具，以及支持不同的系统开发方法学的工具。这一层的内容比较灵活，可根据开发者的需要增删。随着 IPSE 的升级换代，这一层还将越来越多地增添一些带智能性的工具，如自动测试工具、应用生成系统等。

还需指出，现代的开发环境可能使用分布式系统作为宿主机，此时的环境信息仓库也应采用分布式数据库。此外，不少环境采用某种语言作为它的基本语言，这种基本语言既用来支持开发目标系统的软件，也常用来描述环境所配置的工具。

13.3 CASE 环境实例

自 20 世纪 80 年代以来，软件开发环境的研究工作硕果累累。20 世纪 90 年代欧洲计算

机制造商联合会（European Computer Manufacturers Association, ECMA）提出的软件开发环境参考模型，就是一个著名的例子。一些研究机构和公司也在环境的开发上下了很大的力气，实现了许多优秀的 CASE 环境。

本节将简要介绍美国著名软件工程公司 Rational 研制的 Rational SUITE Enterprise Studio 和我国北京大学研制的“青鸟系统”两个 CASE 环境。

13.3.1 Rational SUITE Enterprise Studio

Rational SUITE Enterprise Studio 是 Rational 公司开发的支持整个软件生存周期的大型软件开发环境，主要由 4 个工具组组成：系统分析组 SUITE AnalystStudio，软件设计开发组 SUITE DevelopmentStudio，软件测试组 SUITE TestStudio 和 Web 开发组 SUITE ContentStudio。各组均包括了大量的软件工具，这里将结合 CMM 简单介绍其中几个主要的工具，即软件开发过程框架、需求管理工具、面向对象分析设计工具、配置管理和变更管理工具、测试工具。

1. 软件开发过程框架

本书第 2.5 节介绍的 RUP，就是一个完整的软件开发过程框架，它为在开发组织中分配任务和职责提供了一种规范方法，为所有重要的软件开发活动提供全面的指南、模版和示例，其目标是确保在可预期的时间和预算内，开发出满足最终用户需求的高品质的软件。

RUP 凝结了现代软件开发中多方面的最佳经验，包括迭代化软件开发、需求管理、以构件为基础的软体构架、可视化建模、软件质量保证和变更管理等。

CMM 是一种软件过程控制和评估框架，它列出了每个级别需要完成的目标以及判定条件，但并没有叙述如何达成这些目标。RUP 是希望通过 CMM 认证的企业的最佳起点，它对 CMM 中的每个关键过程域都有帮助。

2. 需求管理工具

需求是软件客户的要求，它决定了软件系统的工作内容，是整个开发活动的基本出发点和最终目标。需求管理的目的是在客户和相应的软件项目之间建立共同的理解，并形成估计、策划和跟踪整个软件生存周期内软件项目活动的基础。

需求管理作为 CMM 2 级（可重复级）的 KPA 之一，它的主要工作包括：通过与项目干系人（stakeholder）交流来获取需求，并进行有效的组织和记录；使客户和项目团队在系统变更需求上达成并保持一致。

有效需求管理的关键在于：

- ① 保持需求被完整、明确地阐述。
- ② 为每种需求类型确定适用的属性。
- ③ 维护一种需求文档与其他需求和其他项目工件的可追踪性。
- ④ 引入工具以支持上述 3 项活动实现自动化。

一个优秀的需求管理工具可以在保证有效地进行需求管理工作的前提下，提高需求管理

工作流程的自动化程度，使需求管理可以真正在项目当中得到有效的推行。Rational 公司的需求工作包 AnalystStudio 就是一个出色的代表，它能够：

- ① 结合 RUP 方法，提供完整的需求分析及管理流程。
- ② 充分利用便捷的 Web 交互方式来获取用户的反馈，并加强团队之间的有效沟通。
- ③ 综合数据库和文字处理软件承载需求信息的互补性优势，用 Word 文档来保存需求的内容，而用数据库来保存需求的各种属性。
- ④ 与优秀的配置管理工具 ClearCase 集成，以提供需求文档的基线。

使用需求工作包 AnalystStudio，除了 CMM 2 级的“需求管理”之外，还可以对以下 KPA 提供帮助：

- ① “软件项目规划” 2 级。
- ② “软件项目跟踪与监督” 2 级。
- ③ “软件子合同管理” 2 级。
- ④ “软件产品工程” 3 级。
- ⑤ “组间协作” 3 级。
- ⑥ “同级复审” 3 级。
- ⑦ “定量过程管理” 4 级。

3. 面向对象分析设计工具

在 CMM 3 级的“软件产品工程”（software product engineering）这个 KPA 中，对于软件设计有着明确的要求，要求软件设计遵循一定的设计语言，采用面向对象的方法，使得设计的结果可复用等。

采用面向对象分析设计（可视化建模）的优点在于：

① 通过分析和设计两个步骤，使开发者可以先关注问题领域，再关心设计问题以及程序设计的细节，这样有利于降低整个过程的复杂性，提高分析模型和设计模型的质量，正所谓“质量从头抓起”（quality from beginning）。

② 生成的分析模型及设计模型将成为文档的主体，从而从根本上解决“先写代码、再补文档”的老问题，并能够帮助企业规避人员流动对企业造成的影响。

③ 分析模型及设计模型将成为团队内部以及团队之间的有效沟通桥梁，消除误解，从而进一步解决系统集成难的顽症；同时，也可以促进团队之间的软件复用。

Rational Rose 是 Rational 公司开发的可视化建模工具，已被 IDC 连续 5 年评为业界最佳的可视化建模工具，它主要特点如下：

① 全面支持 UML 这一面向对象建模的业界标准，采用 UML 表示方法，并在同一个模型中实现业务建模、对象建模和数据建模。所有参与软件项目的成员可以在统一的语言环境中，工作于一个模型之上，有利于改善成员之间的沟通。

② 支持多种语言（C++、Java、VB 等）的代码生成及双向工程，实现代码和模型的互相转换，并且可以将遗留代码引入模型中；和 Rational 公司的文档出版工具 SODA 集成，可

以方便地生成文档，保证代码、模型、文档之间的一致，有利于软件的维护。

③ 它带有对设计元素进行测试的模块工具(quality architect),使得设计模型是可测试的,从而尽早发现设计中的问题,真正实现“质量从头抓起”。

④ 它可以和 ClearCase 以及任何遵守 SCC API 的版本控制系统集成,将软件设计置于配置管理之中,完整地保留软件设计的全过程。

使用 Rational Rose,除了 CMM 3 级的“软件产品工程”之外,还可以对以下 KPA 提供帮助:

① “组间协作” 3 级。

② “同级复审” 3 级。

4. 配置管理和变更管理工具

软件配置管理 (SCM) 是 CMM 2 级中一个非常重要的 KPA,它的目的是在项目的软件生存周期内建立并维护软件项目产品的完整性。在 CMM 标准中,明确规定了软件配置管理 (SCM) 以及变更请求管理 (CRM) 的相关工作:

① 配置管理的主要工作包括通过创建一个软件配置管理库,定义配置项(包括需求、分析设计模型、代码、文档、测试用例和测试数据等),建立和维护软件的基线。

② 变更请求管理的主要工作包括控制和记录配置项内容的变更,建立和维护一个系统去追踪和管理变更请求和问题报告。

开发软件不是一件容易的事,首先面临的是管理多种产品和版本等问题,更为复杂的是由两组或多组人员共同开发相同的程序,再加上多样化的开发程序,使得整个开发过程很难进行有效的管理。早在 20 世纪 70 年代初期加利福尼亚大学的 Leon Presser 教授就撰写了一篇文章,提出控制变更和配置的概念,之后在 1975 年,他成立了一家名为 Soft Tool 的公司,开发了自己的配置管理工具:CCC,这也是最早的配置管理工具之一。之后,随着软件开发规模的逐渐增大,越来越多的公司和团队意识到了软件配置管理的重要性,而相应的软件配置管理工具也如雨后春笋一般,纷纷涌现。Rational ClearCase 和 ClearQuest 是业界领先的配置管理工具,在软件企业当中,实施 ClearCase 可以较快地获得如下收益:

① 利用版本对象库 (VOB) 完整地保存整个项目的开发历史,从而实现有限的软件资产管理,规避人员流动对企业造成的影响。

② 利用版本对象库 (VOB) 的安全机制,可以灵活地控制不同人员对不同配置项的检出 (checkout) 和读取的权利,从而有效地保护企业的核心机密。

③ 利用 ClearCase 可以方便地创建和重现基线,确保发布版本的准确性并能够可靠地构建和修补以前发行的产品。

④ ClearCase 强大的分支功能(如图 13.5 所示)可以帮助团队实现并行开发,从而避免影响其他开发工作,保证项目进度;同时,可以将高风险的特性分配到分支来开发,以保证项目可以按时发布。

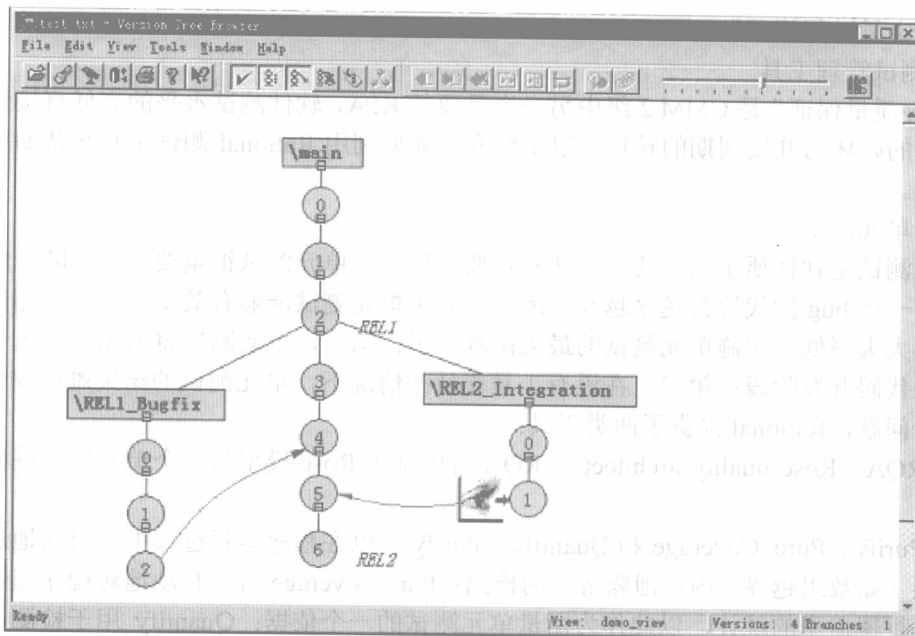


图 13.5 ClearCase 的分支功能

⑤ 利用 trigger、lock 等机制，可以将书面的配置管理规定工具化，让工具而不是人来保证各种配置管理规定被严格遵守。

Rational ClearQuest 是一套高度灵活的缺陷和变更跟踪系统，实施 ClearQuest 可以为企业带来如下好处：

① 加强开发团队与外界的沟通，用户、测试人员与市场销售人员可以直接通过 Web 来提交变更请求（change request），并及时了解进展状况；变更请求包括缺陷（defect）或功能扩充（enhancement request）。

② 用数据库统一管理所有变更请求，避免遗漏和重复，并可以在此基础上，进行定量分析。

③ 项目经理可以根据缺陷数据，定量地分配工作任务，并准确地掌握项目进度。

④ 开发人员可以明确地了解他被分配的开发任务，并根据优先级依次完成。

可以毫不夸张地说，配置管理和变更管理是软件工程的基础。使用 ClearCase 和 ClearQuest，除了 CMM 2 级的“软件配置管理”之外，还可以对以下 KPA 提供帮助：

- ① “需求管理” 2 级。
- ② “软件项目跟踪与监督” 2 级。
- ③ “软件质量保证” 2 级。
- ④ “软件产品工程” 3 级。

⑤ “定量过程管理” 4 级。

5. 测试管理工具

“软件质量保证”是 CMM 2 级中另一个重要的 KPA，软件测试水平的高低直接影响软件产品质量的好坏与开发周期的长短。以下将介绍如何利用 Rational 测试工具来帮助用户解决上述问题。

(1) 单元测试

单元测试是软件质量保证的一个非常重要的环节，单元测试很重要，因为随着时间的推移，修复一个 bug 的代价会越来越高。因此，如果单元测试能够有效实施，整个测试工作的工作量将大大降低。实施单元测试的最大困难在于：第一，单元测试将增加开发人员的工作量，延长代码开发阶段；第二，在没有工具支持的情况下，单元测试的结果难以评估。

对于问题，Rational 提供了两类工具：

① RQA (Rose quality architect)。RQA 可以基于 Rose 模型为一个单元生成桩模块和测试模块。

② Purify、Pure Coverage 和 Quantify。Purify 可以在程序运行过程中，自动地进行内存使用错误（如数组越界、内存泄露等）的检测；Pure Coverage 可以自动地对程序基于代码行或函数进行覆盖率的统计，以此作为衡量单元测试的一个依据；Quantify 用于检测函数或构件的运行性能，判断是否有性能瓶颈存在。

(2) 测试管理

有效的测试管理工作要求合理地组织测试活动，管理测试用例与测试结果，并对发现的差错进行有效的跟踪。具体来说有以下要求：

① 根据软件系统的客户需求制定相应的测试内容，并进行合理的分类、组织，形成测试时使用的测试需求，有的测试活动便是围绕这些测试需求展开的。

② 提供测试用例与测试结果集中存储与使用的平台，使得测试人员能够方便地共享测试资源，测试管理人员也可以方便地进行测试进程的分析，得到测试进度与软件质量的报告。

③ 建立一个差错跟踪的处理流程，并以此流程监视差错的处理情况。

在这些方面，Rational TeamTest 的功能模块 TestManager 与 ClearQuest 为以上要求的实现提供了很好的工具。

(3) 回归测试

软件的功能是指软件能够满足预先确定的需求。对于一个大型软件系统来说，迭代式开发已经成为一种主流的开发模式，需要在开发的每个迭代周期对软件的功能进行确认，这就是一个回归测试的过程。如果依靠人工测试的办法，这将是一个繁琐、耗时的过程。Rational TeamTest 采用面向对象的记录技术，将对系统的功能测试动作记录在测试脚本之中，当系统进入下一个迭代周期时，只需回放这个测试脚本，就可以自动地进行软件功能的确认，这种方法可以极大地提高软件测试的效率，保证软件功能测试的完整性。

(4) 性能测试

软件的性能是指软件系统在处理压力之下，能够保证足够的处理能力。以大型网站系统

为例,系统性能反映在系统在一定用户压力下客户请求的响应时间与系统在没有崩溃之前所能支持的在线用户个数。Rational TeamTest 用虚拟客户模仿真实的客户请求,向系统施加请求压力,来测试软件系统的处理能力。

13.3.2 青鸟系统

青鸟系统的全称是“大型软件工程开发环境青鸟系统”,它是国家科技攻关课题成果,由北京大学牵头研制的一个面向对象的软件开发环境。“七五”期间的攻关研制成功了集成化软件工程环境青鸟 I 型系统,该系统以环境信息库为核心,提供了支持软件生存周期各个过程的软件工具。在“八五”攻关期间,成功地研制了青鸟 II 型系统。

青鸟系统具有如下的特点:

- ① 把支持面向对象的软件开发,作为环境的主要目的之一,具体体现为:
 - 研制和开发作为集成型软件工程环境核心的对象管理系统 Js2/OMS。
 - 以对象管理系统为核心的 JB2 环境的总体结构。
 - 设计并实现支持永久对象的面向对象编程语言 CASE-C++。
 - 在对象管理系统和 CASE-C++语言的支持下对环境的大部分软件工具的开发和集成。
 - 提供一套支持面向对象分析、设计和编程的 OO 系列工具。
 - 为支持图形用户界面的面向对象开发,提供一个包括大量界面常用成分的界面类库和一个面向对象的界面辅助生成器。

总之,通过对象管理系统、CASE-C++语言、OO 系列工具、界面类库及界面辅助生成器的研制,使环境对工具设计者和最终用户的面向对象软件开发形成强有力的支持。

② 较为成功地研制了以数据集成、控制集成和界面集成为中心的开放性环境集成机制,包括:

- 以对象管理系统为核心的数据集成部件。
 - 以消息服务器和过程控制系统作为控制集成部件。
 - 以基本 OSF/Motif 的界面类库和界面辅助生成器作为界面集成部件。
 - 在环境集成机制中提供开放性的工具插槽。
 - 制定符合开放性要求的工具结构模型。
- ③ 支持多种软件开发方法,包括结构化方法、信息模型法和 OO 方法。
 - ④ 系统既可以集成支持软件生存周期全过程的软件工具,成为通用性软件工程环境,又可以通过剪裁而形成支持特定应用领域的专用性应用开发平台。

小 结

现代软件开发,一刻也离不开环境。支持各阶段软件开发活动的各种工具,是环境中

活跃的组成部分。从早期的少量零散工具到初具规模的工具箱，到完全集成化的 I-CASE 的工具集，反映了软件开发环境的巨大变化。良好的软件支持环境，已成为提高开发效率和软件质量的重要条件。

提高生产率常常是研制环境的直接目标，但并不是唯一的收获。新的工具和环境的出现，正在改变着程序设计的面貌和人们使用计算机的方式。从根本上来说，改善环境正是为了简化对计算机的使用，使计算机向着适应人的方向转化。本章末介绍的 Rational SUITE 和青鸟系统，就是现代 CASE 环境的两个实例，表明了人们为达到这一目的所作出的努力。把大量工作留给计算机和环境去完成，尽可能简化用户的工作，是今后环境发展的必然趋势。

习 题

1. 什么是软件工程环境？软件开发环境与运行环境有什么区别？
2. CASE 环境的目标是什么？
3. CASE 环境由哪些部分组成？试述各部分的作用。
4. 什么叫工具集成化？怎样实现工具的集成化？
5. 理想的环境模型应有什么样的结构？

参 考 文 献

1. 软件工程环境. 清华大学出版社, 1990.

第 14 章 软件工程高级课题

从 1968 年提出软件工程一词以来，迄今已 40 年。通过 40 年的实践，软件编程范型（programming paradigm）已从过程式编程范型、面向对象编程范型，发展到基于构件技术的编程范型。与此相对应，软件工程也从早期的结构化分析与设计，跨越第二代的面向对象分析与设计，现正向着基于软件复用的第三代软件工程阔步前进。

40 年来，广大软件工程师在精英学者的带动下，开辟了软件工程的许多研究课题。在今天仍热门的高级课题中，有些是新课题，例如随着网络的普及而兴起的 Web 工程；有些则是由老课题推陈出新，例如基于软件体系结构的软件开发、形式化的软件开发等。

作为全书的结尾，本章将对这些课题作简单的介绍与展望。

14.1 Web 工程

近年来，随着 Internet 和 Intranet/Extranet 的迅猛发展，基于 Web 的系统与应用（Web-based systems and applications, WebApp）越来越引起人们的关注。早在 1998 年，Yogesh Deshpande 和 Steve Hansen 就提出了 Web 工程（Web Engineering, WebE）的概念。根据他们的研究，Web 工程不但来自计算机科学、信息系统和软件工程，也来自其他学科，而且代表了信息技术进化的方向。为此，他们认为，应该把 Web 工程看作一门崭新的学科，而不是软件工程的一个分支。

众所周知，计算机科学概括了硬件和软件，尤其关注可靠和优化的系统性能；信息系统则来自数据处理，它主要关注企业级的信息；而软件工程则通常关注大规模的、基于团队的项目。由此可见，Web 工程是一个快速增长的领域，需要使用更加前沿的方法，而不是仅仅应用以前存在的方法和已被证明了的开发实践。据此，Yogesh Deshpande 和 Steve Hansen 进一步给 Web 工程下了一个定义，即“使用合理的、科学的工程和管理原则，用严密的和系统的方法，来开发、发布和维护基于 Web 的系统的学科”。目前，对 Web 工程的研究主要是在国外开展的，国内还刚刚起步。

这里简单回顾一下 WebApp 的发展史。在 1990—1995 年间，Web 网站仅包含一组不多的超文本静态网页，主要使用文本及少量的图形。随着网络技术的发展，HTML 通过一些开发工具（如 XML 和 Java），其功能得到了发展，从而使 Web 工程师在提供信息的同时也能提供计算功能。今天，通过一些复杂的计算工具，WebApp 不仅可以向最终用户提供单独的

计算功能，而且也集成了企业数据库和商业应用。面对上述的惊人发展，国际知名教材的作者 Pressman 在《软件工程：实践者的方法》（第 5 版）中有一个评价。他认为，“Web 是迄今计算历史中最重要的发展”，而“WebApp 则可以看作计算历史中发生的最重要的事件之一”。

14.1.1 Web 工程与软件工程

尽管 Web 工程也包含程序设计和软件开发，且采用了一些软件工程的原理，但 Web 工程不同于软件工程，基于 Web 的系统开发与传统的软件开发也存在很多差异。

Web 工程与传统的软件工程的区别，主要体现在以下方面：

① WebApp（至少目前是这样）强调信息的含量，无论静态网页或动态网页，都是面向文档的；传统软件工程则强调系统功能的完善，除了系统帮助仍是文档外，其余的页面几乎都用于实现数据交互，是面向功能的。

② WebApp 关注视觉和感觉，强调感官舒服，通过与多媒体相结合，在界面中达到色彩搭配、动画飞扬；传统的软件界面则奉行“简单为美”的原则。

③ 多数 WebApp 是数据驱动的，一个 WebApp 的主要功能是让终端用户使用超媒体来表示文本、图形、音频和视频内容，有些还用来访问那些本不属于 Web 环境下的数据库中的信息（如电子商务或财务应用）；而传统的软件开发多是功能驱动的或过程驱动的。

④ WebApp 能够适应具有不同技术和能力的用户。通过复杂的人机接口、用户界面和信息递交，实现用户形式的多样化。而传统的软件系统的用户群体则通常圈定在某个范围之内，并根据这个群体定制用户界面和人机接口。

⑤ WebApp 必须在短期内开发完成。一个完全的 Web 站点的开发时间可能只有几天或几周，所以，Web 工程很难应用传统软件工程中使用的形式化方法和测试技术。

⑥ 与传统的软件开发相比，Web 工程要求艺术、技术和科学在更大范围内相互结合。所以开发 Web 系统的人员不仅类型众多，需要的技能、知识也更加广泛。

14.1.2 Web 开发

本节从多方面阐述基于 Web 的系统开发特点，包括 Web 开发团队的结构、Web 工程过程的特点、Web 分析、WebApp 设计以及 Web 测试等。

1. Web 开发团队

开发大型基于 Web 的系统，需要一个具有不同技能、知识和能力的人组成的团队。通常把参加开发 WebApp 的人员分为 7 类，即 Web 决策人员、内容提供人员、Web 开发人员、Web 发布人员、Web 支持人员、Web 管理人员和最终用户等。其结构如图 14.1 所示。

(1) Web 决策人员

Web 决策人员一般是指能对开发 WebApp 起决策作用的领导层人物。

(2) 内容提供人员

Web 内容提供人员可以是开发组织内部人员，也可以是外部人员或最终用户。在有些

WebApp 的开发中，最终用户就是内容提供人员。在一般的网站系统中，内容提供人员还可以是网站维护和支持人员、普通的网站浏览人员和稿件作者。

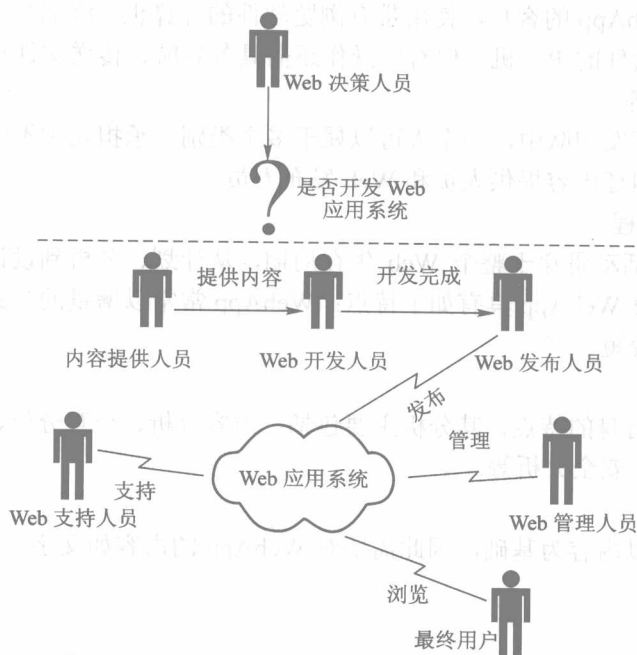


图 14.1 Web 开发团队结构

(3) Web 开发人员

Web 开发人员包括系统分析员、程序设计员、界面设计与美工人员、测试人员等技术人员。

(4) Web 发布人员

Web 发布人员的职责是将系统内容传输到一个 Web 站点上。要求他们在数据库交互、HTML 操作、B/S 功能性、服务器端的应用、CGI 及多媒体应用等方面具备相应的能力。

(5) Web 支持人员

经过适当培训后，Web 支持人员应能从事一些技术支持工作，包括更新与维护 Web 系统。此外，他们还需懂一些站点发布、数据库操作知识和存取数据的方法。为了控制站点不会超量操作，Web 支持人员还需要作 Web 访问统计。当然他们也可以与 Web 开发人员或管理员一起完成这个任务。

(6) Web 管理员

管理员又称站长，负责 Web 网络的管理，涉及 Web 软件和硬件操作技能、网络和通信技能以及与 Web 性能有关的知识。其技术工作一般包括记录文件版本、数据库操作、安全和

存取权限以及通过 CGI 程序或类似的扩展程序对服务端进行操作。

(7) 最终用户

最终用户是 WebApp 的客户，使用带有浏览软件的计算机（终端）。虽然一般的最终用户只需会操作带有软件的 PC 机，但有些操作还需具备导航、传送文件和使用复杂搜索引擎（searching engine）等。

顺便说明，在开发团队中，一个人可以属于多个类别，承担几种不同的工作。例如，最终用户也可以同时担任内容提供人员和 Web 发布人员。

2. Web 工程过程

Web 工程过程活动贯穿于整个 Web 生存周期，从计划、分析到设计、发布、不断的精化和升级系统。开发 WebApp 具有如下特点：WebApp 常常以增量的方式开发；需求经常发生变化；开发期限较短。

3. Web 分析

Web 工程根据自身的特点，其分析主要包括：内容分析、交互分析、功能分析、配置分析和导航关系分析、安全分析等。

(1) 内容分析

因为 WebApp 以内容为基础，因此需要对 WebApp 的内容如文字、图形、图像、音频、视频等进行分析。

(2) 交互分析

交互分析主要解决用户和系统之间的交互问题，包括用户与系统进行交互的方式，用户界面布局、内容、导航链接、实施交互机制及 WebApp 的整体美观度等。

(3) 功能分析

与交互分析类似，功能分析主要涉及 Web 工程操作。用户可见的功能包括任何可以直接由用户操作的功能。例如，一个购物 Web 站点可能要完成许多购物功能（如商品浏览或商品交易），从最终用户的角度来看，这些功能是可操作、可使用的。

(4) 配置分析

配置分析主要对工程所涉及的环境和基础设施进行详细的描述。Web 工程必须设计支持服务器和客户端的环境，要能安装在 Internet 或局域网中。如果整个 Web 工程还涉及数据库，还需要指明数据库的类型。客户端则是一个浏览器，由于目前存在多种不同标准的浏览器，因此需要考虑与客户端的兼容性。

(5) 导航关系分析

因为 WebApp 中的每个页面可能都要与其他页面进行链接，导致链接的复杂化。导航关系分析主要考虑各个页面之间的关系。可以通过对用户的分析和对页面单元的分析来进行。

(6) 安全分析

① Web 站点安全。Internet 不对机密信息和敏感信息提供保护，所以必须自我保护。一个 Web 站点，只要与 Internet 相连，就可以被所有人访问，除非安装某些形式的保护。作为

Web 站点的管理人员，应努力保护站点的信息资源、用户以及客户。Web 的交互性也正是它的致命弱点。Web 中各种受欢迎的功能，例如聊天室、电子商业和自动邮件回复等，都可能受到是黑客和入侵者的攻击。

② Web 站点风险。总的来说，风险分为两类：一是机密信息被窃取，二是数据和软、硬件系统被破坏。这两类风险的危害都是不可低估的。上述两类风险细分为以下 3 类：

- Web 服务器的信息被破译，最终导致闯入者进入服务器。
- Web 上的文件被未经授权的个人访问，损害了文件的隐私性、机密性和完整性。
- 当远程用户向服务器传输信息时，交易被截获。

4. Web 设计

Web 的设计包括它的外观和感观，而且也要考虑 Web 中的所有元素包括观众信息、意图和目标描述、域信息和页面的规范等。下面将从几个方面来阐述 Web 设计方法和特点。

(1) 设计原则和目标

Web 的设计坚持以用户为中心来开发，也就是说，开发过程是一个以用户的要求、兴趣、特征、能力、知识和技术为中心的过程。该规划过程应该产生一个好的观众信息集合。

在整个设计过程中要注意以下的原则和目标：

① 设计要尽可能简单。Web 设计是为用户服务的，达到用户的需求是 Web 最优先的考虑。

② 布局要尽可能美观。互联网上存在大量的信息，如果不能使布局吸引观众，那么这个 Web 设计毫无疑问是失败的设计。

③ 整体要尽可能保持一致，即生成一个一致的、令人愉快的、有效的 Web 的界面。Web 设计的目标应该是给用户一个有关它的界面的印象，这些界面反映了一个共同结构和一致的视觉线索。

(2) 界面设计

Web 工程中界面在用户心目中具有非常重要的地位。界面是 Web 应用系统给人的第一印象，因此，用户界面要易使用、易操作、直观、一致。此外，界面还要求有助于用户浏览。界面设计应考虑浏览者当前的位置，当前可进行的操作，以及当前可导航的目标。

(3) 设计方法

由于没有一个开发 WebApp 的定式，所以开发者可以在多种方法中进行选择。没有一个方法适用于所有的情况，因此在设计同一个 Web 的时候甚至可以考虑改变方法。整个 Web 应该包括哪些信息，可能一时难以确定，因此，可以考虑采用自顶向下的设计方法。

(4) 设计中的问题

尽管有些技术能够有助于保证 Web 界面的一致性，但有些问题会影响 Web 的设计，例如，缺乏导航和信息线索，信息组织和结构过于复杂，结构页面不均匀，存在链接错误页面等。这些问题是 Web 设计人员应该注意的。

(5) 设计人员的检查

设计一个 Web，主要目的就是满足用户的需要。所以一个设计者应努力遵循以用户为中心的原则和目标来开展 Web 设计工作。Web 的设计人员应该掌握用户对网站信息空间、组织和线索的体验，并使用相应技术来为信息打包和设计链接，以便满足用户需求。Web 的设计过程应该包含设计的技巧和解决问题的经验，设计者要努力去改进 Web 的设计，以便更好地满足用户的需要。

5. Web 测试

在 Web 工程过程中，Web 测试、确认和验收是一项重要而富有挑战性的工作。WebApp 的测试与传统的软件测试不同，不但需要检查和验证是否按照设计的要求运行，而且还要评价系统在不同用户浏览器上的显示是否合适。重要的是，还要从最终用户的角度进行安全性和可用性测试。Web 测试主要包括：

- ① 内容测试。检查内容的正确性、一致性、无歧义等问题。
- ② 功能测试。查找不符合用户需求的错误。
- ③ 结构测试。确保其结构是符合 WebApp 的内容和功能的，确保它是可扩展的，支持新的内容或功能。
- ④ 导航测试。确保所有的导航用法和意义都被实现，以便发现导航错误（如空链接、错误链接等）。
- ⑤ 易用性测试。确保每个不同的用户群能被 WebApp 界面支持，能学会并运用所有需要的导航用法和意义。
- ⑥ 性能测试。在不同操作条件、配置和负载下进行，确保系统能响应用户的交互操作，在可接受的性能下降的条件下处理极端的负载量。
- ⑦ 兼容性测试。在客户机和服务器上设定不同的配置条件下执行 WebApp。目的是找出那些只在特定配置下会出现的错误。
- ⑧ 协同工作测试。确保 WebApp 能很好地与其他的应用程序和数据库交互。
- ⑨ 安全性测试。评估潜在的易攻击性，任何一个成功的入侵都认为是安全方面的失败。

14.2 基于体系结构的软件开发

在英文中，“体系结构”（architecture）的含义就是建筑。早期的结构化程序以语句组成模块，通过模块的聚集和嵌套，形成层层调用的程序，这就是体系结构。但那个时代的程序规模不大，通过强调结构化程序设计，并注意模块的耦合性，就可以得到相对良好的结构，所以并未特别强调体系结构。

抽象数据类型和面向对象技术的出现，使体系结构的研究开始受到重视。封装的模式、强大的类库，大大促进了对象/构件的复用。虽然软件项目的规模更大，开发速度更快，仍能开发出相对可靠和易于修改的软件，适应了业界对软件日益增长的需求。早期的“算法 +

数据结构”的程序开发模式，正在被“构件+体系结构”模式所取代。基于构件和体系结构的软件开发方法，被认为是解决软件危机、实现深层次软件复用的有效途径，从而成为近代软件工程研究的重点。

14.2.1 应用软件的体系结构

一般的说，软件的体系结构可以从以下4个角度来分析与考察：程序的、业务的、技术的和信息的。任何软件的体系结构，都可以看成上述4种体系结构的总和。

1. 程序的体系结构

软件离不开程序，一个软件通常包含许多模块和对象。拿应用程序来说，它从头到尾，可能包括处理输入数据所需的用户界面程序，处理数据的加工/计算或以数据管理为主要业务的主程序与子程序，以及处理输出数据的打印或报表程序，等等。这些模块/对象既各有分工，又互相合作，通过相互之间的调用/通信，即形成应用程序的体系结构。

2. 业务的体系结构

采用应用程序的目的，总是为了完成某项业务。以支持超市日常营业的 POS (point of sales) 系统为例，每个超市一般设有若干收银台。当售货员将顾客所购商品输入系统后，收银台随即计算货款，打印收款单据。超市办公室的软件则统计每个收银台当天售出的商品与货款，一方面要在售货员下班时与之进行核对；另一方面可将当日售得现金送交银行，结算账目。门市经理和总经理办公室的程序，则需及时汇总各方面的信息，供领导了解和查询。这样，从经理到售货员，从超市后勤到银行，所有与本程序相关的单位与个人，共同组成一个业务的体系结构。

3. 技术的体系结构

从技术的角度看，应用程序还存在一个技术体系结构。例如，从收银台到办公室通常用局域网连接；从超市门市到总部可采用广域网连接；各点之间既可采用对等通信技术，也可采用 C/S 技术；用户界面可以采用简单的图形用户界面，也可使用最新的办公系统。

不言而喻，应用程序的模块/对象设计，与上述的技术体系结构将密切相关。

4. 信息的体系结构

最后，从信息/数据流动的角度考察，还存在一个信息的体系结构。例如，这一应用程序有哪些输入数据和输出数据？哪些是共享数据？哪些是临时数据？在数据的存储、更新、删除和使用中，需要遵循哪些规则？

信息体系结构的差异，例如是采用一般数据文件还是采用数据库，往往会直接影响应用程序的功能。

综上所述，确定软件的体系结构，在现代软件开发中具有重要的意义。通常情况下，根据用户的业务流程，应首先构造一个业务的体系结构，它大致相当于传统软件开发中按照用户的功能需求来确定软件的解决方案。与此同时，应考虑采用什么样的技术和信息体系结构，并通过修改达到圆满的效果。把上述这些加上软件交付时间和开发经费约束，就可以获得解

决方案所需的完整的体系结构。下一步，就是怎样按照既定的体系结构，组织人员来开发应用程序了。

14.2.2 编程范型对体系结构的影响

随着编程范型从第一代演变到第三代，体系结构对软件的开发日益显得重要。传统的软件工程使用过程式编程语言进行结构化编程，结构化程序本身采用强耦合的模块结构。正如第 3 章所述，这时的程序、数据、体系结构自然天成，所以当时并未太多强调“体系结构”一词。

第二代编程范型采用面向对象技术。对象被定义为数据与操作相对孤立的封装体，可以独立完成一些比较小的功能，从而为软件复用打开了方便之门。在软件开发中有强大的类库支持，就可以在它的基础上实现许多具有独立功能的构件与模式。

随着第三代编程范型的出现，基于构件的技术和互联网支持的 Web 服务被大量应用，就更加重视体系结构了。由第 10 章可知，构件把接口的功能从对象中分离出来后，即使用接口来表示构件能够“做什么”，构件体只回答“怎样做”。可见此时的体系结构，只需抓住需要软件“做什么”就足够了。

14.2.3 编程范型对复用粒度的影响

还需指出，随着编程范型的演变，软件复用的粒度（grainality）也随之不断增大，从模块/对象，到构件、子系统，一直到某个领域。打个简单的比喻，在结构化程序设计时代，人们以砖、瓦、灰、沙、石来预制梁、柱、屋面板等，盖的是平房和小楼；而在面向对象时代，人们使用的是整面墙、整间房、一层楼梯等预制件，直接盖高楼大厦。

于是，下列问题就自然地提到了软件开发者的面前：怎样搭配构件才合理？体系结构怎样能保证构造容易？如果更改了重要构件，怎样保证整个高楼不会倒塌？进而言之，每种应用领域（例如医院、工厂、机关、旅馆）需要什么样的构件？什么样的构件骨架（例如适用、美观、强度与造价合理等）建造出来的建筑（或体系结构）更能满足用户的需求？

14.2.4 软件体系结构技术仍在发展

综上所述，“体系结构”一词早已有之，并不是什么新概念。但它从不自觉地被运用，发展到今天成为人们关注的焦点，反映了业界对软件本质问题认识的深化。

从理论上说，体系结构可以看成是集合{构件，连接件，约束}。开始，人们试图用形式化的体系结构描述语言（architecture description language, ADL）来准确地描述它，也获得过一些成果。但由于各个侧面相互影响，难以量化表达，效果并不显著。

另一方面，在实践研究上却取得了比较丰硕的成果。总结历来的软件技术，人们发现有不少惯用的模式，可作为体系结构的基础。例如管道、过滤器和桥，由谁设计都差不多。如果总结出来，便可把它用作预制件，即使不做成预制件，也可以像数据结构中的队、栈、表、

图、树那样做成模板，照着编写程序，就可以开发出良好的体系结构来。

1995年，美国人 E. Gamma 等总结出以 UML 表示的 23 种最基本的模式，后来 B. Douglass 又总结了 54 种实时设计模式。模式是对象/构件组成的功能元件，也称轻量级体系结构，可以和面向对象的程序直接对应，以便程序开发人员去构造信息的、技术的和业务的体系结构。后来又有人研究粒度更大的模式，其大小相当于子系统，从而形成了由体系结构、样式、模式、构件/对象所构成的 4 级元结构。正是由于体系结构的上述实践研究与应用领域密切相关，才推动了领域工程、领域理论的研究不断向前发展。

软件工程的宗旨，就是将最科学、最本质的理念，总结为软件工程的方法学、规范、乃至行业标准，使开发人员可用较小的代价做出高质量的软件。从这个意义上说，软件体系结构技术的发展正方兴未艾，还远远没有结束。

14.3 面向方面的软件开发

面向对象技术的出现改变了人们编写软件的方法，它将软件需求中的对象抽象出来，一定程度上实现了软件结构的模块化。但随着软件设计的进一步深入，人们逐渐发现对象的抽象并不能完全解决问题。在软件系统的设计过程中，人们首先关注的往往是与系统业务相关的模块，比如银行系统中账户的存取模块、库存管理系统中的货物购置和入库模块，然后会考虑分布在多个核心模块中的公共行为，比如日志记录、安全性、缓存和权限控制等。人们把前者称为核心关注点，后者称为横切关注点。面向对象的编程技术（OOP）可以很好地完成对核心关注点的设计与开发，而对横切关注点却有些力不从心。其原因在于，横切关注点会跨越多个模块，是多维的，而 OOP 的设计方法却是一维的，把多维的需求映射到一维上，便产生了许多需要探讨的问题。

14.3.1 面向方面编程

面向方面编程（aspect-oriented programming, AOP）是对软件工程的一种革新性思考，是由施乐公司帕洛阿尔托研究中心（Xerox Palo Alto Research Center）的 Gregor Kiczales 等在 1997 年提出的，主要用来解决横切关注点问题，并开发了第一个 AOP 开发环境 AspectJ。AOP 为开发者提供了一种描述横切关注点的方法，人们可以通过它单独实现横切模块。此外，AOP 提供了一种机制，使得核心模块和横切模块能够融合在一起，从而构造出最后的实际系统。AOP 用一种边界清晰的方式把横切关注点模块化，产生一个更容易设计、实现和维护的系统架构。与常规的软件开发技术不同的是，常规技术会将这些不同的关注点实现于多个类中，而面向方面编程将使它们局部化。

这里简单介绍 AOP 中的一些基本概念。

1. 关注点（concern）

关注点也就是要考察或解决的问题。比如在一个电子商务系统中，订单的处理、用户的

验证和用户日志记录等都属于关注点。核心关注点，是指一个系统中的核心功能，也就是一个系统中跟特定业务需求联系最紧密的商业逻辑。除了核心关注点以外，还有一种关注点，它们分散在各个模块中解决同样的问题，这种跨越多个模块的关注点称为横切关注点或交叉关注点（crosscutting concerns）。在一个电子商业系统中，用户验证、日志管理、事务处理、数据缓存都属于交叉关注点。

2. 方面（aspect）

方面是一个抽象的概念，从软件的角度来说是指在应用程序不同模块中的某一个领域或方面。从程序抽象的角度来说，可以对照 OOP 中的类来理解。OOP 中的类（class）是实现世界模板的一个抽象，它包括方法、属性、实现的接口、继承等。而 AOP 中的方面是实现世界领域问题的抽象，包括属性和方法等。从抽象意义上讲，方面是对软件系统构件的性能和语法产生一定影响的一些属性；从设计上讲是横切系统的一些软件系统级关注点；从实现上讲，是一种程序设计单元，它支持将横切系统的关注点封装在单独的模块单位中，是 AOP 将横切关注点局部化和模块化的实现机制。

3. 连接点（join point）

连接点也就是运用程序执行过程中需要插入方面模块的某一点。连接点主要强调的是一个具体的“点”的概念。这个点可以是一个方法、属性、构造函数、类静态初始化模块，甚至一条语句。

4. 织入（weaving）

织入是指把解决横切问题的方面模块，与系统中的其他核心模块通过一定策略或规则组合在一起的过程。编译器织入是一种常用的织入方法，这种方法使用专门的编译器来编译包括方面模块在内的整个应用程序，在编译的过程中实现织入，这种织入是功能最强大的。编译器织入的 AOP 实现一般都是基于语言扩展的方式，即通过对标准语言进行一些简单的扩展，加入一些专用于处理 AOP 模块的关键字，定义一套语言规范，通过这套语言规范来开发方面模块，使用自己的编译器来生成目标代码或中间代码。

14.3.2 AOP 语言规范

从抽象的角度看来，一种 AOP 语言要说明下面两个方面：

① 关注点的实现。把每个需求映射为代码，然后，编译器把它翻译成可执行代码，由于关注点的实现以指定过程的形式出现，可以使用传统语言如 C、C++、Java 等。

② 织入规则规范。怎样把独立实现的关注点组合起来形成最终系统呢？为此，需要建立一种语言来指定组合不同的实现单元以形成最终系统的规则，这种指定织入规则的语言可以是实现语言的扩展，也可以是一种完全不同的语言。

AOP 设计的具体步骤如下：

① 对需求规约进行方面（Aspect）分解。

- 确定哪些功能是组件必须实现的，即提取核心关注点。

- 哪些功能可以以方面的形式动态加入到系统组件中去，即提取系统级的横切关注点。
- ② 对标识出的方面分别通过程序机制实现。
 - 构造系统的组件。利用组件语言实现系统的组件。对于 OOP 语言，这些组件可以是类；对于过程化程序设计语言，这些组件可以是各种函数和 API。
 - 构造系统的方面。利用一种或多种 AOP 语言实现方面，且 AOP 语言必须提供声明方面的机制。
- ③ 用方面编织器将所有的单元编排重组在一起，形成最终的可运行系统。
 - 为组件语言和 AOP 语言构造相应的语法树；依据方面中的连接点定义对语法树进行联结；在连接的语法树上生成中间文件或目标代码。
 - AOP 语言必须提供将方面代码和基础代码织入的机制。
 - AOP 语言必须提供生成可运行系统的实现机制。

14.3.3 AOP 与 OOP 比较

OOP 是 AOP 的技术基础，AOP 是对 OOP 的继承和发展，它们之间区别如下。

1. 可扩展性

可扩展性是指软件系统在需求更改时程序的易修改能力。由于切面模块根本不知道横切关注点，所以很容易通过建立新的切面加入新的功能。另外，如果系统中加入新的模块，已有的方面自动横切进来，使系统易于扩展，因此：

- ① OOP 主要通过提供继承和重载机制来提高软件的可扩展性。
- ② AOP 通过扩展方面或增加方面，系统相关的各个部分都随之产生变化。

2. 可重用性

可重用性指某个应用系统中的元素被应用到其他应用系统的能力。AOP 把每个切面实现为独立的模块，模块之间是松散耦合的。举例来说，可以用另外一个独立的日志写入器方面（替换当前的）把日志写入数据库，以满足不同的日志写入要求。因此：

- ① OOP 以类机制作为一种抽象的数据类型，提供了比过程化更好的重用性。
- ② OOP 的重用性对非特定于系统的功能模块有很好的支持，如堆栈的操作和窗口机制的实现。

③ 对于不能封装成类的元素，如异常处理等，很难实现重用。

④ AOP 使不能封装成类的元素的重用成为可能。

总的来说，松散耦合的实现意味着更好的代码重用性。

3. 易理解性和易维护性

① 代码集中，易于理解，从而解决了由于 OOP 跨模块造成的代码混乱和代码分散问题。

② AOP 用最小的耦合来处理每个关注点，即使是横切关注点也是模块化的，这样实现的系统，其代码的冗余小，模块化的实现使系统容易理解和维护。

14.3.4 面向方面软件开发

面向方面软件开发 (aspect-oriented software development, AOSD) 使用 AOP 方法, 在整个软件生存周期中提供系统化标识、模块化以及横切关注点, 为功能需求、非功能需求和平台特性等创造了更好的模块性, 便于开发出更易于理解的系统, 也更易于配置和扩展, 以满足和解决更广泛的需求。

面向方面软件开发可以将需要出现多次的公用代码集中到一处实现, 从而大大减少代码的冗余度和耦合度, 增强可读性。由于代码修改的后期成本大大减少, 设计师再也不必陷入设计不足或者过度设计的两难境地, 即使后期出现了额外的需求, 也可以将它封装在方面中独立实现, 图 14.2 是面向方面软件开发的具体过程。

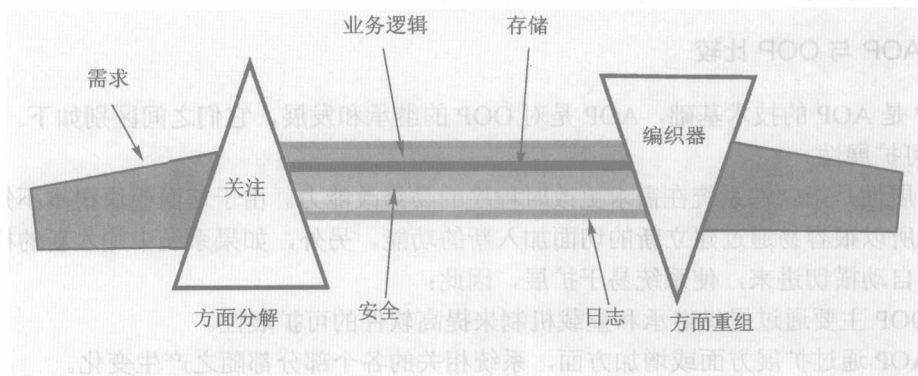


图 14.2 面向方面的软件开发过程

AOSD 包括 3 个清晰的开发步骤:

- ① 方面分解。分解需求并提取出横切关注点和一般关注点。这一步把核心模块级关注点和系统级的横切关注点分离开来。
- ② 关注点实现。各自独立地实现这些关注点。
- ③ 方面的重新组合。在这一步, 方面集成器通过创建一个模块单元——方面来指定重组的规则。重组过程 (也称织入或结合) 则使用这些信息来构建最终系统。

14.4 形式化的软件开发

软件形式化最早可追溯到 20 世纪 50 年代后期对程序设计语言编译技术的研究, 即 J. Backus 提出巴克斯范式 (Backus normal formula, BNF) 描述 ALGOL 60 语言的语法, 出现了各种语法分析程序自动生成器以及语法制导的编译方法, 使得编译系统的开发从手工艺

制作方式发展成具有牢固理论基础的系统方法。形式化方法的研究热潮始于 20 世纪 60 年代后期，针对当时的“软件危机”，人们提出种种解决方法，归纳起来有两类：一是采用工程方法来组织、管理软件的开发过程；二是深入探讨程序和程序开发过程的规律，建立严密的理论，用来指导软件开发实践。前者促使软件工程的出现和发展，后者则推动了形式化方法的深入研究。

14.4.1 形式化方法的定义

形式化方法的基本含义是借助数学的方法来研究计算机科学中的有关问题。在《Encyclopedia of Software Engineering》一书中，对形式化方法定义为：用于开发计算机系统的形式化方法是描述系统性质的基于数学的技术，这样的形式化方法提供了一个框架，可以在框架中以系统的而不是特别的方式刻画、开发和验证系统。也就是说，如果在软件开发的过程中，凡是采用严格的数学语言，具有精确的数学语义的方法，都称为形式化方法。从广义角度，形式化方法是软件开发过程中分析、设计及实现的系统工程方法。狭义地，形式化方法是软件规格（specification）和验证（verification）的方法。因此形式化方法又分为形式化规格方法和形式化验证方法。形式化规格是通过具有明确数学定义的文法和语义的方法或语言对软件的期望特性或者行为进行的精确、简洁的描述。形式化验证是基于已建立的形式化规格，对软件的相关特性进行评价的数学分析和证明。

将形式化方法用于软件开发的主要目的是保证软件的正确性。形式化方法的本质是基于数学的方法来描述目标软件系统属性的一种技术。在软件开发过程中使用数学方法具有如下优点：数学是准确的建模媒体，能够对现象、对象、动作等进行简洁、准确的描述；数学支持抽象，它使得规格说明的本质可以被展示出来，并且可以以一种有组织的方式来表示系统规格中的抽象层次；数学提供了高层确认的手段，可以使用数学证明来揭示规格中的矛盾性和不完整性，以及设计和规格之间的一致性等。不同的形式化方法的数学基础是不同的，有的以集合论和一阶谓词演算为基础，有的则以时态逻辑为基础。

1. 形式化方法的特点

- ① 提供了规格环境的基础——形式化描述的分析模型。
- ② 设计、实现和单元测试等过程被一个形式化的、逐步求精的、正确性得到保证的变换过程所替代。
- ③ 形式化方法适用于对安全性、可靠性、保密性要求极高的系统开发。

2. 形式化方法模型

形式化方法模型是基于形式化规格及程序变换的软件开发模型，又称为自动程序设计模型，或者变换模型。采用了形式化方法模型的开发以形式规格为中心，具有 3 个核心活动：规格化生成、形式化验证及计算机辅助下的程序求精变换。具体过程为：首先，根据用户需求生成软件需求规格说明书；对该规格说明书进行检查，判断其是否满足用户的需求以及其内部是否存在不一致、不完整等错误，同时对其进行形式化验证，判断其是否具有预期

望的特性；然后，以规格说明书为基础，对其进行逐步求精，每一步求精都降低了一些抽象程度或增加一些过程，最终得到可执行的程序代码。在求精的过程中需要注意保证各步求精所得结果之间的功能的一致性。理论上，只要有一个正确满足用户需求的形式化规格，经过一系列正确的程序变换后，应该能生成正确的、可接受的程序代码。形式化方法模型可以最大限度地提高软件开发的可靠性和安全性，同时，可以改善软件开发效率，有效控制开发进度。实施形式化方法模型开发的常用技术有：

- ① 基于模型的规格说明书及其变换技术。
- ② 基于代数结构的规格说明书及其变换结构。
- ③ 基于时序逻辑的规格说明书和验证技术。
- ④ 基于可视形式化的技术。

14.4.2 形式化的软件开发

形式化的软件开发采用了软件生存周期的形式化方法模型。从某种角度上来看，形式化软件开发就是把现实世界的需求映射为软件的模型化过程。在模型化的过程中，涉及 3 方面的系统模型：现实世界、模型表示和计算机系统。形式化方法软件开发的过程就是从这 3 个方面对系统进行描述和转换的过程。开发过程的任务依次为模型获取、模型验证和模型变换。模型获取是从现实世界向模型表示转换的过程，包括如何提取模型以及如何表示模型，它对应于软件生存周期中的需求分析、规格说明以及设计等活动；模型验证是对所得到的模型表示进行检验，判断其是否捕获了所有的用户需求，以及该模型是否具有所期望的特性；模型变换是从模型表示向计算机系统变换的过程，一个抽象的模型表示可以变换到各种计算机系统环境上，模型变换的一个关键任务是进行一致性测试，即判断变换后所得到的计算机系统是否与模型表示一致。模型变换对应于软件生存周期中的实现和测试等活动。这些任务分别对应于如下 3 个方面的活动：形式化规格、形式化验证以及程序求精。

软件规格是对软件系统对象及用来对系统对象进行操作的方法集合，以及对所开发的系统中的各对象在其生存周期中的集体行为的描述。规格可以采用非形式化的方式来描述，包括自然语言、图、表等，也可以用形式化的方式来描述。由于非形式化的方法本身存在二义性、模糊性和不完整性等情况，使得所得到的规格说明书不能准确地刻画系统模型，甚至会为以后的软件开发留下隐患。而对于形式化方法来说，由于其基于严格的数学描述，具有严格的语法和语义定义，从而可以准确地描述系统模型，避免了二义性和模糊性等情况；同时，在对系统进行严格描述的过程中，将会帮助用户明确其原本模糊的需求，并发现用户所陈述需求中存在的矛盾等情况，从而相对完整、正确地理解用户需求，最终得到一个完整、正确的系统模型。

形式化规格 (formal specification, 也称形式规范或形式化描述) 是对程序“做什么” (what to do) 的数学描述，是用具有精确语义的形式语言书写的程序功能描述，它是设计和编制程序的出发点，也是验证程序是否正确的依据。不同的形式化规格方法使用不同的形式化规格

语言，即用于书写形式规格的语言（也称形式化描述语言），如代数语言 OBJ、VDM、Z 语言等；进程代数语言 CSP、CCS、 π 演算等；时序逻辑语言 PLTL、CTTL、XYZ 等；这些规格语言由于基于不同的数学理论及规格方法，因而也千差万别，但它们都有一个共同的特点，即每种规格语言均由基本成分和构造成分两部分构成。前者用来描述基本（原子）规格，后者把基本部分组合成大规格。构造成分是形式化规格研究和设计的重点，也是衡量规格语言优劣的主要依据。

形式化软件开发的另一重要活动是形式化验证（formal verification）。通常，软件开发中的大部分错误是在需求分析和规格的早期阶段引入的，这些错误将随着开发的深入而逐渐放大，并且，这些错误发现得越晚，对其修改所需付出的代价也将会越大。在传统的软件开发中，除了在各个开发阶段进行评审，以便发现错误外，更多的错误则是到编码结束后的测试阶段才被检测出来的。而在形式化的软件开发中，在开发出形式规格后就进行验证，实际上是使验证工作提前进行，既可以提前发现错误，同时也可以降低改正错误的代价。形式化验证与形式化规格之间具有紧密的联系，形式化验证就是验证已有的程序（系统）是否满足其规格的要求，它也是形式化方法所要解决的核心问题。形式化验证的主要技术有模型检验和定理证明。

模型检验是一种基于有限状态模型，并验证该模型的期望特性的技术，对模型的状态空间进行搜索，以确认该系统模型是否具有某些性质。搜索的可终止性依赖于模型的有限性。模型检验主要适用于有穷状态系统，其优点是完全自动化，并且验证速度快，同时模型检验可用于系统部分规格，即对于只给出了部分规格的系统，通过搜索也可提供关于已知部分正确性的有用信息；此外，当所检验的性质未被满足时，将终止搜索过程，并给出反例，这些信息反映了系统设计中的细微错误，因而对于用户排错有极大的帮助。但模型检验方法的一个严重缺陷是“状态爆炸问题”，即随着所要检验的系统的规模增大，模型算法所需的时间/空间开销往往呈指数增长。

定理证明采用逻辑公式来规格说明系统及其性质，其中的逻辑由一个具有公理和推理规则的形式化系统给出，进行定理证明的过程就是应用这些公理或推理规则来证明系统具有某些性质。不同于模型检验，定理证明可以处理无限状态空间问题。定理证明系统可以分为自动和交互式两种类型。自动定理证明系统是通用搜索过程，适用于解决各种组合问题；交互定理证明系统需要与用户进行交互，要求用户能提供验证中创造性最强部分（建立断言等）的工作，因而其效率较低，较难用于大系统的验证。定理证明的实施同样需要定理证明器的支持。现有的定理证明器包括用户引导自动推演工具、证明检验器和复合证明器等。

程序求精又称为程序变换，它也是形式化软件开发的一项重要活动，是将自动推理和形式化方法相结合而形成的一门技术，主要研究从抽象的形式规格推演出具体的面向计算机的程序代码的全过程。

程序求精的基本思想是用一个抽象程度低、过程性强的程序代替一个抽象程度高、过程性弱的程序，并保持它们之间的一致性。这里所说程序是规格说明书、设计文档以及程序代

码的统称。因此，程序可以划分为若干层次：最高层是不能直接执行的程序，即规格，它由抽象的描述语句构成；最低层是可以直接执行的程序代码，它由可执行的命令语句构成；最高层和最低层之间的是一系列混合程序，其中既有抽象的描述语句，又有可执行的命令语句。

程序开发过程实际上是从最高层的程序开始，通过一系列的求精变换步骤，每一步降低一些抽象程度或增加一些可执行性，最终得到能够指导计算机明确执行的程序代码。在求精的过程中，要注意保持程序的正确性，保证所得到的程序是满足最初的形式规格的。程序的这种正确性可以通过求精过程中所遵循的一系列规则来保证，也可以在事后采用验证工具来证明。程序求精是形式化方法研究的一个热点，目前比较典型的是 IBM Hursley 公司和牛津大学 PRG 程序设计研究组提出的针对 Z 规格的求精方法。

经过 50 多年的研究和应用，如今人们在形式化方法这一领域取得了许多重要的成果，从早期最简单一阶谓词演算方法发展出现在可应用于不同领域、不同阶段的基于逻辑、状态机、网络、进程代数、代数等众多形式化方法。形式化方法的发展趋势逐渐融入软件开发过程的各个阶段，从需求分析、功能描述、设计、编程、测试直至维护。软件开发自动化技术是提高软件生产率的根本途径之一，形式化方法是软件自动化的根本前提。

小 结

Web 工程是一门十分年轻的学科，引起专家、学者和开发人员的广泛关注，还需要进一步化和成熟。本章简述了 Web 工程的特点，Web 开发的人员组织，并且讨论了 Web 工程过程模型，以及 Web 系统分析和设计等基本问题。

软件开发不仅要满足软件系统的功能，对非功能性问题如可修改性、可重用性和可靠性等也越来越重视。鉴于软件体系结构直接关系到软件性能的好坏，基于软件体系结构的开发目前正方兴未艾，已经成为当今软件开发中一个突出的研究课题。

所谓软件体系结构，应包括软件完成的业务、执行业务的组织、组织的位置、运行软件所需的信息和技术基础设施等多方面的内容。其好坏对软件性能和质量有重要影响，并且与具体的开发过程紧密相关。本章从系统的角度阐述了软件体系结构的概念，并由此探讨了以体系结构为中心的开发过程。

面向方面编程（AOP）是对软件工程的一种革新性思想，也是对 OOP 的继承和发展。本章介绍了 AOP 的基本概念和语言规范，简述了面向方面软件开发（AOSD），并从可扩展性、可重用性、易理解性和易维护性等方面对 AOP 与 OOP 进行了初步比较。

形式化的软件开发起源于深入探讨程序开发过程的数学理论，最早可追溯到对程序设计语言编译技术的研究。本章介绍了形式化方法的定义，狭义地说，将它归纳为形式化规格和形式化验证方法，讨论了应用形式化方法模型来进行形式化开发的过程。

通过 40 年的实践，软件工程已从早期的结构化分析与设计，跨越第二代的面向对象分析与设计，现在正朝着基于软件复用的第三代软件工程阔步前进。作为本书的结尾，本章对

当前几个热门的课题作简要的介绍与展望，希望引起读者注意。

习 题

1. 以一个实际的 Web 站点为例，评价其用户界面并给出改进建议。
2. 解释软件体系结构。
3. 什么是面向方面软件开发？简述 AOP 与 OOP 的不同。
4. 什么是形式化方法？
5. 形式化软件开发一般包括哪几个步骤？
6. 什么是程序求精？

附录 缩略语中英文对照表

4GL, the 4 th generation language,	第四代语言
ADL, architecture description language,	体系结构描述语言
AOP, aspect-oriented programming,	面向方面编程
AOSD, aspect-oriented software development,	面向方面软件开发
ASD, adaptive software development,	自适应软件开发
ASE, application system engineering,	应用系统工程
BNF, Backus normal formula,	巴克斯范式
CAD, computer-aided design,	计算机辅助设计
CAF, CMM assessment frame-work,	CMM 评估框架
CASE, computer-aided software engineering,	计算机辅助软件工程
CBA-SCE, CMM-based appraisal for software capability estimation,	软件能力评估的 CMM 评估方法
CBA-IPI, CMM-based appraisal for internal process improvement,	内部过程改进的 CMM 评估方法
CBSD, component-based software development,	基于构件的软件开发
CCB, change control board,	修改控制组
CFD, control flow diagram,	控制流图
CMM, capacity maturity model,	能力成熟度模型
CMMI, capability maturity model integration,	能力成熟度模型集成
COCOMO, constructive cost model,	构造性成本模型
COM, component object model,	构件对象模型
CORBA, common object request broker architecture,	公共对象请求代理体系结构
CRM, change request management,	变更请求管理
CSPEC, control specification,	控制说明
DD, data dictionary,	数据字典
DFD, data flow diagram,	数据流图
EAF, effort adjustment factor,	工作量调节因子
ECMA, European Computer Manufacturers Association,	欧洲计算机制造商联合会
E-R, entity-relation diagram,	实体联系图
FAST, facilitated application specification techniques,	便利的应用规约技术
I-CASE, integrated CASE,	集成 CASE
IPO, input-process-output,	输入-处理-输出
IPSE, integrated project support environment,	集成项目支持环境
JSD, Jackson system development,	Jackson 系统开发方法或 JSD 方法

KPA, key process area, 关键过程域
LCP, logical construction of programs, 程序的逻辑构造
LOC, lines of code, 代码行
MRF, maintenance request form, 维护申请单
N-S chart, Nassi and Shneiderman chart, N-S 图或结构化程序的流程图
OLE, object linking and embedding, 对象连接和嵌入技术
OMG, object management group, 对象管理组织
OMT, object modeling technique, 对象建模技术
OOA, object-oriented analysis, 面向对象分析
OOD, object-oriented design, 面向对象设计
OOP, object-oriented programming, 面向对象程序设计
PERT, program evaluation and review technique, 程序评估和复审技术
PLC, programmable logical controller, 可编程控制器
PSE, programming support environment, 程序设计支持环境
PSPEC, process specification, 加工说明
REBOOT, reuse based on objected-oriented technology, 基于面向对象技术的复用
RMMM, risk mitigation, monitoring and management plan, 风险缓解、监控和管理计划
RUP, rational unified process, rational 统一过程
SA, structured analysis, 结构化分析
SC, structure chart, 结构图
SCM, software configuration management, 软件配置管理
SCR, software change report, 软件修改报告单
SD, structured design, 结构化设计
SDE, software development environment, 软件开发环境
SE ² , software engineering environment, 软件工程环境
SP, structured programming, 结构化程序设计
SPR, software problem report, 软件问题报告单
SPA, software process assessment, 软件过程评估
SPICE, software process improvement and capability determination, 软件过程改进和能力确定
SQA, software quality assurance, 软件质量保证
SRS, software requirements specification, 软件需求规格说明书
S ² E, software supported environment, 软件支持环境
STD, status transfer diagram, 状态变迁图
TCF, technical complexity factor, 技术复杂性因子
TQC, total quality control, 全面质量管理
UML, unified modeling language, 统一建模语言
V&V, verification and validation, 验证与确认
WBS, work breakdown structure, 分类活动结构图
XP, extreme programming, 极限编程

主要参考文献

- [1] PRESSMAN R S. Software Engineering: A Practitioner's Approach[M]. 6th ed. [S. l.]: McGraw-Hill, 2004.
- [2] SCHACH S R. Software Engineering with Java[M]. [S. l.]: McGraw-Hill, 1999.
- [3] PFLEEGER S L. Software Engineering Theory and Practice[M]. 2nd ed. [S. l.]: Prentice Hall, 2001.
- [4] 郑人杰. 软件工程(高级)[M]. 北京: 清华大学出版社, 1999.
- [5] 史济民. 软件技术基础实用教材[M]. 北京: 人民邮电出版社, 1998.
- [6] 朱三元. 软件工程技术概论[M]. 北京: 科学出版社, 2002.
- [7] 齐治昌. 软件工程[M]. 2版. 北京: 高等教育出版社, 2005.
- [8] 张海藩. 软件工程师[M]. 5版. 北京: 清华大学出版社, 2008.
- [9] 杨正甫. 面向对象分析与设计[M]. 北京: 中国铁道出版社, 2001.
- [10] 麦中凡. 计算机软件技术基础[M]. 北京: 高等教育出版社, 1999.
- [11] BANIASSAD C. An Approach for Aspect-oriented Analysis and Design[C]. Proceedings of the 26 Int'l Conf. on Software Engineering, Edinburgh, Scotland. 2004: 158-167.
- [12] FILMAN R E, ELRADT T. 面向方面的软件开发[M]. 莫倩, 王恺译. 北京: 机械工业出版社, 2006.
- [13] PRESSMAN R S. 软件工程: 实践者的研究方法[M]. 6版. 郑人杰, 马素霞译. 北京: 机械工业出版社, 2007.
- [14] 钱乐秋, 赵文耘. 软件工程[M]. 北京: 清华大学出版社, 2007.
- [15] KICZALES G, LAMPING J, MENDHEKAR A, et al. Aspect-Oriented Programming[J]. LNCS, 1997, 1241: 220-242.
- [16] DESHPANDE Y, STEVE H. Web Engineering: Creating a Discipline among Disciplines. [J]. IEEE Software, 2001; (2): 82-87.
- [17] 古天龙. 软件开发的形式化方法[M]. 北京: 高等教育出版社, 2005.
- [18] MICHAEL R B, JAMES R R. Object-Oriented Modeling and Design with UML[M]. 2nd ed. [S. l.] Prentice Hall, 2004.
- [19] 麦中凡. 软件应用开发新技术浅说系列[J]. Chip 新电脑, 2005: 总第 188-192 期.

软件工程——原理、 方法与应用

(第3版)

Software Engineering: Principles, Methods
and Applications (3rd Edition)

- 从“应用”出发，兼顾“原理”与“方法”，在讲解方法时精选例题，以便读者理解；当升华到原理时提纲挈领，画龙点睛。
- 重点介绍面向对象的软件工程，同时对基于构件的软件工程进行简单的对比和介绍。
- 加强实践环节，以案例为先行和归宿。
- 加强对部分新技术的介绍，反映Web软件工程等新进展。

ISBN 978-7-04-026146-2



9 787040 261462 >

定价 28.00 元