

计算机专业大专系列教材

软件工程概论

郑人杰 殷人昆 编著

清华大学出版社

(京)新登字 158 号

内 容 简 介

软件工程是 20 世纪 60 年代开始发展起来的新学科。随着计算机的普及, 作为其核心部分的软件已深入到社会生产活动和生活的各个领域。软件的开发和维护都需要软件工程知识, 因此有人称它为软件产业的支柱。本书是作者根据在清华大学多年教学的讲义改编的。内容包括: 软件工程概述; 软件需求分析; 软件设计; 详细设计描述的工具; 程序编码; 面向对象技术; 软件测试; 软件维护; 软件工程标准化与软件文档。书中适当介绍了软件管理和软件工程标准化问题。掌握这些知识将有助于读者在软件工程项目中体现工程化和标准化。内容通俗易懂, 图文并茂, 原理、方法与实例结合。

本书适于作大专院校中计算机或软件专业的教材, 也可供计算机软件人员和计算机用户阅读。

版权所有, 翻印必究。

本书封面贴有清华大学出版社防伪标签, 无标签者不得销售。

图书在版编目(CIP)数据

软件工程概论/ 郑人杰, 殷人昆编著. - 北京: 清华大学出版社, 1998

计算机专业大专系列教材

ISBN 7-302-02909-1

. 软... . 郑... 殷... . 软件工程 . TP311. 5

中国版本图书馆 CIP 数据核字(98)第 07165 号

出版: 清华大学出版社(北京清华大学学研大厦, 邮编 100084)

<http://www.tup.tsinghua.edu.cn>

印刷: 印刷厂

发行: 新华书店总店北京科技发行所

开本: 787x 1092 1/16 印张: 18.75 字数: 441 千字

版次: 1998 年 4 月第 1 版 2000 年 12 月第 2 次印刷

书号: ISBN 7- 302- 02909- 1/TP · 1539

印数: 80001 ~ 16000

定价: 0.00 元

序 言

为什么要组织编写这套计算机专业大专使用的教材？根据什么来组织这套教材？这套教材的特点是什么？它能起到哪些作用？这就是我们在这篇序言中，要回答，也必须回答的问题。

计算机专业大专教育发展非常迅速，它满足了社会对这个层次人才的需求。在数量上已经超过了对本科人才的需求。大专这个层次有自己的特殊性，时间只有三年，要学习的内容很多，怎样精选教学内容，就成为十分重要的问题，他们又不同于中专层次，要求既有相当坚实的理论基础，又要运用理论解决实际问题，因此，如何处理好理论和实践的关系就十分重要。

大专这个层次人才的重要性是不言而喻的。但是，在培养大专这个层次人才的过程中，突出的矛盾之一，就是缺乏合适的大专教材。目前，不是用本科教材代用，就是很难及时获得所需教材。这就是组织这套专为大专使用的教材的起因。

那么，我们组织编写这套教材以什么为依据呢？中国计算机学会教育委员会与全国高等学校计算机研究会联合推荐的《计算机学科教学计划 1993》（以下简称“93 计划”）是我们这次组织编写这套教材的主要依据。

“93 计划”所提供的指导思想和学科内容不仅适合大学本科，也适合于大专的需要。

“93 计划”明确规定了计划实施的目标：1. 要为“计算机学科”的毕业生提供一个广泛坚实的基础；2. 在培养人才的过程中，必须反映培养目标的差异；3. 要为学生毕业后，进一步学习新的知识和迎接新的工作挑战，做好理论和实践上的准备；4. 要学生能够把在校学到的知识，用到解决实际问题的过程中去。

在学科内容方面“93 计划”概括了九个科目领域。九个科目领域组成“计算机学科”的主科目。每个科目领域都有重要的理论基础、重要的抽象（实验科学）、重要的设计和实现的成就。

这九个科目领域作为教学计划的公共要求，它们是：算法与数据结构、计算机体系结构、人工智能与机器人学、数据库与信息检索、人-机通信、数值与符号计算、操作系统、程序设计语言、软件方法学和工程。

我们根据上述指导思想和学科要求精选了十三门课，作为大专用的主干教材。它们是：《数据结构》、《数字电路逻辑设计》、《计算机组成原理》、《微机原理与应用》、《微机接口技术》、《计算机网络》、《数据库原理及应用》、《操作系统基础》、《汇编语言程序设计》、《C 程序设计》、《软件工程概论》、《微机系统应用基础》和《离散数学》。

这十三种教材大体上反映了除人工智能与机器人学和数值与符号计算之外的全部要求，足以满足大专主干课程教学的需求。

这套教材我们都是聘请大专院校有丰富教学实践经验的、工作在第一线的专家、教授编写。在编写过程中,充分考虑了大专的特点,在选材上贯彻少而精的原则,在处理上贯彻理论密切联系实际的原则,力求深入浅出,便于教学。并且在主要章节后均附有适量的习题。

这套教材适合于计算机专业大专生使用,也可供非计算机专业的本科生使用。

主编 李大友

1996. 6

前 言

软件工程是一门实用性很强的年青学科。尽管其中也包含了某些理论的内容,但它具有一个显著特点是实践性。软件工程学科的实践性不仅体现在,它的形成和发展得益于软件工程项目的推动,或者说,是人们在软件开发的实践中碰壁之后为寻求“软件危机”的出路而总结出的原则和方法;而且它的实践性还体现在对于软件开发项目的实际指导作用。许多人感到,理解和掌握这一学科的知识并不难,然而,常常发生的问题是不能坚持按照它所提供的原则和方法去做。例如,有些规模不小的软件项目因为一开始就忽视了按软件工程的要求开发,致使开发后期,或是在维护阶段步入了十分被动的境地。由于结构性、清晰性和可扩充性差,导致整体的可维护性差,运行中发现了问题,多次修改形成“补丁上加补丁”之后,难于再行修补。加上文档编制得不够理想,以致没有人愿意承担维护工作。但若将其放弃重新开发,从时间、资源多方面考虑,又是不可能的。这类情况一再发生表明,其教训并没有被人们认真吸取。一个中型以上的软件开发项目成功与否很大程度上取决于管理工作,这一点已逐渐成为人们的共识。为防止类似事件的重演,建议从两个方面着手,即:

1. 加强软件过程的管理,不断改进已为人们习惯了的传统开发过程。这就要克服轻视项目管理和文档工作、程序编写任意性等传统习惯。这一点正是本书编入最后两章内容的初衷。

2. 初学者在一接触本学科时,即强调应用,强调实践。希望初学者在一开始就打下良好的软件工程观念的基础,并在今后的软件开发工作中得到贯彻。

本书主要针对大专以上的读者,如感到其中材料不足,可参阅作者的另一本教材《实用软件工程》(第二版),清华大学出版社出版。

特别感谢殷人昆副教授,他在繁忙的教学科研工作中抽空帮我最后脱稿付梓,才使本书与读者见面。

郑人杰

1997年9月22日于清华园

目 录

第 1 章	软件工程概述	1
1.1	软件的概念、特点和分类	1
1.1.1	软件的概念与特点	1
1.1.2	软件的分类	3
1.2	软件的发展和软件危机	6
1.3	软件工程过程和软件生存期	8
1.3.1	软件工程过程(software engineering process)	8
1.3.2	软件生存期(life cycle)	9
1.4	软件生存期模型	10
1.4.1	瀑布模型(waterfall model)	10
1.4.2	演化模型(evolutional model)	11
1.4.3	螺旋模型(spiral model)	11
1.4.4	喷泉模型(water fountain model)	13
1.4.5	智能模型(intelligence model)	13
1.5	软件工程的基本目标	13
1.5.1	软件工程的定义	13
1.5.2	软件工程项目的基本目标	14
第 2 章	软件需求分析	15
2.1	软件需求分析概述	15
2.1.1	软件需求分析的任务	15
2.1.2	需求分析的过程	16
2.1.3	软件需求分析的原则	19
2.2	结构化分析方法	21
2.2.1	数据流图(DFD, data flow diagram)	21
2.2.2	数据词典(DD, data dictionary)	24
2.2.3	加工逻辑说明	27
2.3	结构化数据系统开发方法(DSSD)——面向数据结构的分析方法之一	30
2.3.1	Warnier 图	30
2.3.2	DSSD 的分析方法	31
2.4	Jackson 系统开发方法(JSD)——面向数据结构的分析方法之二	34
2.4.1	进程模型	35
2.4.2	JSD 方法的步骤	35
2.4.3	实体动作分析	36

2.4.4	实体结构分析	37
2.4.5	定义初始模型	38
2.5	原型化方法 (prototyping)	40
2.5.1	软件原型的分类	41
2.5.2	快速原型开发模型	41
2.6	系统动态分析.....	44
2.6.1	状态迁移图	44
2.6.2	Petri 网	45
2.7	结构化分析与设计方法 (SADT)	48
第 3 章	软件设计	51
3.1	软件设计的目标和任务.....	51
3.1.1	软件设计在开发阶段中的重要性	51
3.1.2	软件设计任务	52
3.2	程序结构与程序结构图.....	54
3.2.1	程序的树状结构和网状结构	55
3.2.2	结构图(structure chart, 简称 SC)	55
3.3	模块的独立性.....	57
3.3.1	模块(module)	57
3.3.2	模块独立性(module independence)	58
3.3.3	耦合性(coupling)	58
3.3.4	内聚性(cohesion)	60
3.3.5	信息隐蔽	63
3.4	结构化设计方法——面向数据流的设计方法.....	63
3.4.1	典型的系统结构形式	64
3.4.2	变换分析	66
3.4.3	事务分析	69
3.4.4	软件模块结构的改进	71
3.5	结构化数据系统开发方法(DSSD) ——面向数据结构的设计方法之一.....	75
3.5.1	一种简化的设计方法	75
3.5.2	导出逻辑输出结构	76
3.5.3	导出逻辑处理结构(LPS)	76
3.6	Jackson 系统开发方法 (JSD) ——面向数据结构的分析与设计方法之二.....	78
3.6.1	JSD 功能描述	78
3.6.2	决定系统时间特性	82
3.6.3	实现	82
第 4 章	详细设计描述的工具	87

4.1	程序流程图(program flow chart)	87
4.2	N-S 图	90
4.3	PAD	91
4.4	PDL	93
第5章	程序编码	100
5.1	对源程序的质量要求	100
5.2	结构化程序设计	101
5.2.1	关于 GOTO 语句的争论	101
5.2.2	结构化程序设计的原则	102
5.2.3	程序设计自顶向下, 逐步求精	104
5.3	程序设计风格	106
5.3.1	源程序文档化	107
5.3.2	数据说明	110
5.3.3	语句结构	110
5.3.4	输入和输出(I/O)	114
5.4	程序复杂性度量	115
5.4.1	代码行度量法	115
5.4.2	McCabe 度量法	116
5.4.3	Halstead 的软件科学	117
第6章	面向对象技术	120
6.1	面向对象的概念	120
6.2	基于复用的开发过程	123
6.2.1	应用生存期	123
6.2.2	类生存期	124
6.3	面向对象分析与模型化	126
6.3.1	面向对象分析(OOA, object-oriented analysis)	126
6.3.2	论域分析(domain analysis)	126
6.3.3	应用分析(application analysis)	129
6.3.4	对象模型技术(OMT, object model tech.)	129
6.4	高层设计	134
6.5	类的设计	135
6.5.1	通过复用设计类	135
6.5.2	类设计的方针	136
6.5.3	类设计的过程	138
6.6	Coad 与 Yourdon 面向对象分析与设计技术	143
6.6.1	面向对象的分析	143
6.6.2	面向对象的设计	145
6.7	Booch 的方法	146

6.7.1	Booch 方法的设计过程	147
6.7.2	Booch 方法的基本的模型	147
6.8	面向对象设计的实现	151
6.8.1	类的实现	151
6.8.2	系统的实现	153
第 7 章	软件测试	155
7.1	软件测试的基础	155
7.1.1	什么是软件测试	155
7.1.2	软件测试的目的和原则	156
7.1.3	软件测试的对象	157
7.1.4	测试信息流	158
7.1.5	测试与软件开发各阶段的关系	159
7.2	测试用例设计	160
7.3	白盒测试的测试用例设计	161
7.3.1	逻辑覆盖	161
7.3.2	语句覆盖	162
7.3.3	判定覆盖	163
7.3.4	条件覆盖	163
7.3.5	判定-条件覆盖	164
7.3.6	条件组合覆盖	164
7.3.7	路径测试	165
7.4	黑盒测试的测试用例设计	165
7.4.1	等价类划分	165
7.4.2	边界值分析	168
7.4.3	错误推测法	171
7.4.4	因果图	171
7.5	软件测试的策略	174
7.5.1	单元测试(unit testing)	175
7.5.2	组装测试(integrated testing)	177
7.5.3	确认测试(validation testing)	181
7.5.4	系统测试(system testing)	183
7.5.5	测试的步骤及相应的测试种类	183
7.6	人工测试	186
7.6.1	静态分析	186
7.6.2	人工测试	187
7.7	调试(Debug, 排错)	189
7.7.1	调试的步骤	189
7.7.2	几种主要的调试方法	190

7.7.3	调试原则.....	193
第 8 章	软件维护.....	194
8.1	软件维护的概念	194
8.1.1	软件维护的定义.....	194
8.1.2	影响维护工作量的因素.....	195
8.1.3	软件维护的策略.....	195
8.2	软件维护活动	196
8.2.1	软件维护申请报告.....	196
8.2.2	软件维护工作流程.....	197
8.2.3	维护档案记录.....	198
8.2.4	维护评价.....	198
8.3	程序修改的步骤及修改的副作用	198
8.3.1	分析和理解程序.....	199
8.3.2	修改程序.....	199
8.3.3	重新验证程序.....	202
8.4	软件可维护性	202
8.4.1	软件可维护性的定义.....	203
8.4.2	可维护性的度量.....	203
8.5	提高可维护性的方法	206
8.5.1	建立明确的软件质量目标和优先级.....	206
8.5.2	使用提高软件质量的技术和工具.....	206
8.5.3	进行明确的质量保证审查.....	207
8.5.4	选择可维护的程序设计语言.....	208
8.5.5	改进程序的文档.....	209
8.6	逆向工程和再工程	210
第 9 章	软件工程标准化与软件文档.....	211
9.1	软件工程标准化	211
9.1.1	什么是软件工程标准.....	211
9.1.2	软件工程标准化的意义.....	213
9.1.3	软件工程标准的层次.....	213
9.1.4	中国的软件工程标准化工作.....	214
9.2	软件质量认证	215
9.2.1	ISO 9000 系列标准及软件质量认证	215
9.2.2	ISO 9000 系列标准的内容	216
9.2.3	制定与实施 ISO 9000 系列标准	217
9.2.4	ISO 9000-3 的要点	218
9.3	在开发机构中推行软件工程标准化	220
9.4	软件文档的作用与分类	221

9.4.1	软件文档的作用和分类.....	221
9.4.2	对文档编制的质量要求.....	223
9.4.3	文档的管理和维护.....	225
9.5	软件过程成熟度模型	227
9.5.1	软件机构的成熟性.....	227
9.5.2	软件过程成熟度模型.....	228
9.5.3	关键过程领域.....	229
9.5.4	成熟度提问单.....	230
第 10 章	软件管理	232
10.1	软件生产率和质量的度量.....	232
10.1.1	软件度量	232
10.1.2	面向规模的度量	232
10.1.3	面向功能的度量	233
10.1.4	软件质量的度量	234
10.1.5	影响软件生产率的因素	235
10.2	软件项目的估算.....	236
10.2.1	对估算的看法	236
10.2.2	软件项目计划的目标	237
10.2.3	软件的范围	237
10.2.4	软件开发中的资源	237
10.2.5	软件项目估算	240
10.2.6	分解技术	241
10.3	软件开发成本估算.....	243
10.3.1	软件开发成本估算方法	244
10.3.2	专家判定技术	245
10.3.3	软件开发成本估算的经验模型	245
10.4	软件项目进度安排.....	249
10.4.1	软件开发小组人数与软件生产率	250
10.4.2	任务的确定与并行性	251
10.4.3	制定开发进度计划	251
10.4.4	进度安排的方法	252
10.4.5	项目的追踪和控制	254
10.5	软件项目的组织与计划.....	255
10.5.1	软件项目管理的特点	255
10.5.2	制定计划	257
10.5.3	软件项目组织的建立	258
10.5.4	人员配备	262
10.5.5	指导与检验	263

10. 6	软件配置管理.....	265
10. 6. 1	软件配置管理	265
10. 6. 2	配置标识	267
10. 6. 3	版本控制	269
10. 6. 4	变更控制	269
10. 6. 5	配置状态报告(configuration status reporting, CSR)	271
10. 6. 6	配置审计(configuration audit)	271
附录	软件产品开发文档编写指南.....	273
参考文献	287

第 1 章 软件工程概述

在近代技术发展的历史上,工程学科的进步一直是产业发展的巨大动力。传统的工程学科走过的道路已为人们所熟知。水利工程、建筑工程、机械工程、电力工程等对工农业、商业、交通业的影响是极为明显的。近年来人们开始对气象工程、生物工程、计算机工程等有了新的认识。然而对工程学科家族的另一新成员——软件工程却不很熟悉。事实上,软件工程的地位非常重要,它对软件产业的形成和发展起着决定性的推动作用,在计算机的发展和应用中至关重要,在人类进入信息化社会时成为新兴信息产业的支柱。

本章将对软件的地位和作用、软件的特点、软件的发展和软件危机、软件工程学科的形成、软件生存期及软件工程过程等方面的问题和基本概念给出简要的介绍,以便使读者在进入软件工程专题以前,对总体框架获得一般性的理解。

1.1 软件的概念、特点和分类

1.1.1 软件的概念与特点

“软件”这一名词于 20 世纪 60 年代初从国外传来。英文 software 一词确是 soft 和 ware 两字组合而成。有人译为“软制品”,也有人译为“软体”。现在人们统称它为软件。对于它的一种公认的解释为,软件是计算机系统中与硬件相互依存的另一部分,它是包括程序、数据及其相关文档的完整集合。其中,程序是按事先设计的功能和性能要求执行的指令序列;数据是使程序能正常操纵信息的数据结构;文档是与程序开发、维护和使用有关的图文材料。

为了能全面、正确地理解计算机和软件,必须了解软件的特点。

1) 软件是一种逻辑实体,不是具体的物理实体。它具有抽象性。这个特点使它和计算机硬件,或是其它工程对象有着明显的差别。人们可以把它记录在纸面上,保存在计算机的存储器内部,也可以保留在磁盘、磁带和光盘上,但却无法看到软件本身的形态,必须通过观察、分析、思考、判断,去了解它的功能、性能及其它特性。

2) 软件的生产与硬件不同,在它的开发过程中没有明显的制造过程。也不像硬件那样,一旦研制成功,可以重复制造,在制造过程中进行质量控制。软件是通过人们的智力活动,把知识与技术转化成信息的一种产品。一旦某一软件项目研制成功,以后就可以大量地复制同一内容的副本。所以对软件的质量控制,必须着重在软件开发方面下功夫。由于软件的复制非常容易,因此出现了软件产品的保护问题。

3) 在软件的运行和使用期间,没有硬件那样的机械磨损、老化问题。任何机械、电子设备在使用过程中,其失效率大都遵循如图 1.1(a)所示的 U 型曲线(即浴盆曲线)。因为在刚投入使用时,各部件尚未做到配合良好、运转灵活,容易出现問題。经过一段时间运行,就可以稳定下来。而当设备经历了相当长时间的运转,就会出现磨损、老化,使失效率

越来越大。当失效率达到一定程度,就到达了寿命的终点。而软件的情况与此不同,它没有 U 型曲线的右半翼,因为它不存在磨损和老化问题。然而它存在退化问题。在软件的生存期中,为了它能够克服以前没有发现的故障,使它能够适应硬件、软件环境的变化以及用户新的要求,必须要多次修改(维护)软件,而每次修改不可避免地引入新的错误,导致软件失效率升高,如图 1.1(b) 所示,从而使得软件退化。

图 1.1 失效率曲线

4) 软件的开发和运行常常受到计算机系统的限制,对计算机系统有着不同程度的依赖性。软件不能完全摆脱硬件单独活动。有的软件这种依赖性大些,常常为某个型号的计算机所专用。有的软件依赖于某个操作系统。为了解除这种依赖性,在软件开发中提出了软件移植的问题。

5) 软件的开发至今尚未完全摆脱手工艺的开发方式。软件产品大多是“定做”的,很少能做到利用现成的部件组装成所需的软件。近年来软件技术虽然取得了不少进展,提出了许多新的开发方法,例如利用现成软件的复用技术、自动生成技术,也研制了一些有效的软件开发工具或软件开发环境,但在软件项目中采用的比率仍然很低。由于传统的手工艺开发方式仍然占据统治地位,开发的效率自然受到很大限制。

6) 软件本身是复杂的。软件的复杂性可能来自它所反映的实际问题的复杂性,例如,它所反映的自然规律,或是人类社会的事务,都具有一定的复杂性;另一方面,也可能来自程序逻辑结构的复杂性。软件开发,特别是应用软件的开发常常涉及到其它领域的专门知识,这对软件人员提出了很高的要求。软件的复杂性与软件技术的发展不相适应的状况越来越明显。图 1.2 显示出软件技术的发展落后于复杂的软件需求。

7) 软件成本相当昂贵。软件的研制工作需要投入大量的、复杂的、高强度的脑力劳动,它的成本比较高。问题不仅于此,值得注意的是硬件软件的成本近 30 年来发生了戏剧性的变化。无论研制也好,向厂家购买也好,在 20 世纪 50 年代末,软件的开销大约占总开销的百分之十几,大部分成本要花在硬件上;但 80 年代这个比例完全颠倒过来,软件的开销大大超过硬件的开销,如图 1.3 所示。今天的情况更是这样。美国每年投入软件开发的经费要有几百亿美元。然而,也并非在所有软件开发上的花费都能获得成果。

8) 相当多的软件工作涉及到社会因素。许多软件的开发和运行涉及机构、体制及管理方式等问题,甚至涉及到人的观念和人们的心理。对于这些人的因素重视得不够,常常是软件工作遇到的问题之一。例如,由于主管部门对正在开发的软件不够理解,因而软件

开发得不到应有的重视和必要的支持,造成人力和资金上的困难,它直接影响到项目的成败。

图 1.2 软件技术的发展
落后于需求

图 1.3 计算机系统硬、软件
成本比例的变化

1.1.2 软件分类

以上讨论的是区别于计算机硬件或其它工程对象的各种软件的共同特点。下面讨论软件类型。事实上,要给软件做出科学的分类很难,但鉴于不同类型的工程对象,对其进行开发和维护有着不同的要求和处理方法,因此仍需要对软件类型进行必要的划分。

1) 按软件的功能进行划分

- 系统软件: 能与计算机硬件紧密配合在一起,使计算机系统各个部件、相关的软件和数据协调、高效地工作的软件。例如,操作系统、设备驱动程序以及通信处理程序等。系统软件的工作通常伴随着: 频繁地与硬件交往、资源的共享与复杂的进程管理,以及复杂数据结构的处理。系统软件是计算机系统必不可少的一部分。
- 支撑软件: 是协助用户开发软件的工具软件,其中包括帮助程序人员开发软件产品的工具,也包括帮助管理人员控制开发进程的工具。表 1.1 给出了一些支撑软件的实例。
- 应用软件: 是在特定领域内开发,为特定目的服务的一类软件。现在几乎所有领域都使用了计算机,为这些应用领域服务的应用软件种类繁多。例如商业数据处理软件、工程与科学计算软件、计算机辅助设计/制造(CAD/CAM)软件、系统仿真软件、智能产品嵌入软件(如汽车油耗控制、仪表盘数字显示、刹车系统),以及人工智能软件(如专家系统、模式识别)等。而在事务管理、办公自动化方面的软件也在企事业单位迅速推广,中文信息处理、计算机辅助教学(CAI)等软件使得计算机向家庭普及,甚至连娃娃也能在计算机上学习和游戏。

2) 按软件规模进行划分

按开发软件所需的人力、时间以及完成的源程序行数,可确定 6 种不同规模的软件,如表 1.2 所示。

- 微型: 只是一个人,甚至是半日工作,在几天内完成的软件。写出的程序不到 5 百行语句,仅供个人专用。通常这种小题目无需做严格的分析,也不必要有一套完整的设计、测试资料。但事实说明,即使这样小的题目,如果经过一定的分析、系统设计、结构化编码以及有步骤地测试,肯定也是非常有益的。

表 1.1 支撑软件举例	
一般类型	支持需求分析
文本编辑程序	PSL/PSA 问题描述语言、问题描述分析程序
文件格式化程序	关系数据库系统
磁盘向磁带向数据传输的程序	一致性检验程序
程序库系统	CARA 计算机辅助需求分析程序
支持设计	支持实现
图形软件包	编辑程序
结构化流程图绘图程序	交叉编辑程序
设计分析程序	预编译程序
程序结构图编辑程序	连接编辑程序
支持测试	支持管理
静态分析程序	PERT 进度计划评审方法绘图程序
符号执行程序	标准检验程序
模拟程序	库管理程序
测试覆盖检验程序	

表 1.2 软件规模的分类			
类别	参加人员数	研制期限	产品规模(源程序行数)
微型	1	1 ~ 4 周	0.5k
小型	1	1 ~ 6 月	1k ~ 2k
中型	2 ~ 5	1 ~ 2 年	5k ~ 50k
大型	5 ~ 20	2 ~ 3 年	50k ~ 100k
甚大型	100 ~ 1000	4 ~ 5 年	1M(= 1000k)
极大型	2000 ~ 5000	5 ~ 10 年	1M ~ 10M

- 小型: 一个人半年内完成的 2 千行以内的程序。例如, 数值计算问题或是数据处理问题就是这种规模的课题。这种程序通常没有与其它程序的接口。但需要按一定的标准化技术、正规的资料书写以及定期的系统审查。只是没有大题目那样严格。
- 中型: 5 个人以内在一年多时间里完成的 5 千到 5 万行的程序。这种课题出现了软件人员之间、软件人员与用户之间的联系、协调的配合关系问题。因而计划、资料书写以及技术审查需要比较严格地进行。这类软件课题比较普遍, 许多应用程序和系统程序就是这样的规模。在开发中使用系统的软件工程方法是完全必要的。
- 大型: 5 至 10 个人在两年多的时间里完成的 5 万到 10 万行的程序。例如编译程序、小型分时系统、应用软件包、实时控制系统等。对于这样规模的软件, 采用统一的标准, 实行严格的审查是绝对必要的。由于软件的规模庞大以及问题的复杂性, 往往会在开发的过程中出现一些事先难于做出估计的不测事件。

- 甚大型: 100 至 1000 人参加, 用 4 到 5 年时间完成的具有 100 万行程序的软件项目。这种甚大型项目可能会划分成若干个子项目。子项目之间具有复杂的接口。例如, 远程通信系统、多任务系统、大型操作系统、大型数据库管理系统、军事指挥系统通常具有这样的规模。很显然, 这类问题没有软件工程方法的支持, 它的开发工作是不可想象的。
- 极大型: 2000 人到 5000 人参加, 10 年内完成的 1000 万行以内的程序。

从以上介绍可知, 规模大、时间长、很多人参加的软件项目, 其开发工作必须要有软件工程的知识做指导。而规模小、时间短、参加人员少的软件项目也得有软件工程概念, 遵循一定的开发规范。其原则是一样的, 只是对软件工程技术依赖的程度不同。

3) 按软件工作方式划分

- 实时处理软件: 指在事件或数据产生时, 立即予以处理, 并及时反馈信号, 控制需要监测和控制的过程的软件。主要包括数据采集、分析、输出三部分, 其处理时间是被严格限定的, 如果在任何时间超出了这一限制, 都将造成事故。
- 分时软件: 允许多个联机用户同时使用计算机。系统把处理机时间轮流分配给各联机用户, 使各用户都感到只是自己在使用计算机的软件。
- 交互式软件: 能实现人机通信的软件。这类软件接收用户给出的信息, 但在时间上没有严格的限定。这种工作方式给用户很大的灵活性。近年来, 终端设备更加普及, 交互式软件到处可见。日益显得突出的一个重要问题就是用户界面设计。良好的用户界面设计将给用户带来极大的方便。
- 批处理软件: 把一组输入作业或一批数据以成批处理的方式一次运行, 按顺序逐个处理完的软件。这是最传统的工作方式。

4) 按软件服务对象的范围划分

软件工程项目完成后可以有两种情况提供给用户:

- 项目软件: 也称定制软件, 是受某个特定客户(或少数客户)的委托, 由一个或多个软件开发机构在合同的约束下开发出来的软件。例如军用防空指挥系统、卫星控制系统的软件就属于这一类。为取得客户的委托项目, 软件开发机构的质量管理、技术实力、开发经验以及履行合同的信誉成为受到重视的问题。
- 产品软件: 是由软件开发机构开发出来直接提供给市场, 或是为千百个用户服务的软件。这是一些服务于多个目的及多个用户的软件。例如, 文字处理软件、文本处理软件、财务处理软件、人事管理软件等。由于要参与市场竞争, 其功能、使用性能以及培训和售后服务显得尤为重要。

5) 按使用的频度进行划分

有的软件开发出来仅供一次使用。例如用于人口普查、工业普查的软件。由于若干年才进行一次普查, 前些年开发的软件在若干年以后很难适用。有的统计资料或试验数据需按年度做统计分析, 相应的软件每年运行一次。另外有些问题, 需要每天及时进行数据处理, 如天气预报。这类软件具有较高的使用频度。显然, 开发不同使用频度的软件, 有不同的要求, 不可一律看待。

6) 按软件失效的影响进行划分

工作在不同领域的软件, 适应其不同的需求, 在运行中对可靠性也有不同的要求。有的软件如果在工作中出现了故障, 造成软件失效, 可能给软件整个系统带来的影响不大。例如可能带来一些不便, 却能勉强工作。但有的软件一旦失效, 可能酿成灾难性后果, 其严重损失难以挽回。例如控制载人飞行物的软件, 如果不能正常工作, 可能以人的生命为代价。

1.2 软件的发展和软件危机

20 世纪 40 年代中出现了世界上第一台计算机以后, 就有了程序的概念。可以认为它是软件的前身。经历了几十年的发展, 人们对软件有了更深刻的认识。在这几十年中, 计算机软件经历了三个发展阶段:

- 程序设计阶段, 约为 20 世纪 50 至 60 年代。
- 程序系统阶段, 约为 20 世纪 60 至 70 年代。
- 软件工程阶段, 约为 20 世纪 70 年代以后。

从表 1.3 中可看到三个发展时期主要特征的对比。

表 1.3 计算机软件发展的三个时期及其特点

特 点	时 期		
	程序设计	程序系统	软件工程
软件所指	程 序	程序及规格说明书	程序、文档、数据
主要程序设计语言	汇编及机器语言	高 级 语 言	软 件 语 言
软件工作范围	编写程序	包括设计和测试	软 件 生 存 期
需 求 者	程序设计者本人	少 数 用 户	市 场 用 户
开发软件的组织	个 人	开 发 小 组	开发小组及大中型软件开发机构
软 件 规 模	小 型	中、小型	大、中、小型
决定质量的因素	个人程序设计技术	开发小组技术水平	管 理 水 平
开发技术和手段	子程序 程序库	结构化程序设计	数据库、开发工具、开发环境、 工程化开发方法、标准和规范、 网络和分布式开发、对象技术
维护责任者	程序设计者	开 发 小 组	专职维护人员
硬 件 特 征	价格高、存储容量 小、工作可靠性差	降价, 速度、容量及 工作可靠性明显提高	向超高速、大容量、微型化及 网络化方向发展
软 件 特 征	完全不受重视	软件技术的发展不能 满足需要, 出现软件 危机。	开发技术有进步, 但未获突破 性进展, 价格高, 未完全摆脱 软件危机。

这里软件语言包括需求定义语言、软件功能语言、软件设计语言、程序设计语言等。

几十年来最根本的变化体现在:

1) 人们改变了对软件的看法。20 世纪 50 年代到 60 年代时, 程序设计曾经被看做是一种任人发挥创造才能的技术领域。当时人们认为, 写出的程序只要能在计算机上得出正

确的结果,程序的写法可以不受任何约束。而且只有那些通篇充满了程序技巧的程序才是高水平的好程序,尽管这些程序很难被别人看懂。然而随着计算机的广泛使用,人们逐渐抛弃了这种观点。对于稍大的、需要较长时间为许多人使用的程序,人们要求这些程序容易看懂、容易使用,并且容易修改和扩充。于是,程序便从个人按自己意图创造的“艺术品”转变为能被广大用户接受的工程化产品。

2) 软件的需求是软件发展的动力。早期的程序开发者只是为了满足自己的需要,这种自给自足的生产方式仍然是其低级阶段的表现。进入软件工程阶段以后,软件开发的成果具有社会属性,它要在市场中流通以满足广大用户的需要。软件开发者和用户的分工和责任也是十分清楚的。

3) 软件工作的范围从只考虑程序的编写扩展到涉及整个软件生存期。

在软件技术发展的第二阶段,随着计算机硬件技术的进步,计算机的容量、速度和可靠性有明显提高,生产硬件的成本下降。计算机价格的下跌为它的广泛应用创造了条件。在这一形势下,要求软件能与之相适应。一些开发复杂的、大型的软件项目提了出来。然而软件技术的进步一直未能满足形势发展提出的要求。在软件开发中遇到的问题找不到解决的办法,致使问题积累起来,形成了日益尖锐的矛盾。这些问题归结起来有:

(1) 由于缺乏软件开发的经验和软件开发数据的积累,使得开发工作的计划很难制定。主观盲目地制定计划,执行起来和实际情况有很大差距,致使经费预算常常突破。对于工作量估计不准确,进度计划无法遵循,开发工作完成的期限一拖再拖。为加快已拖延项目的进度而增加人力,结果适得其反,不仅未能加快,反而更加延误。

(2) 作为软件设计依据的需求,在开发的初期阶段提得不够明确,或是未能得到确切的表达。开发工作开始后,软件人员和用户又未能及时交换意见,使得一些问题不能及时解决而隐藏下来,造成开发后期矛盾的集中暴露。这时问题既难于分析,也难于挽回。

(3) 开发过程没有统一的、公认的方法论和规范指导,参加的人员各行其事。加之不重视文字资料工作,使设计和实现过程的资料很不完整;或是忽视人与人的接口部分,发现了问题修修补补,这样的软件很难维护。

(4) 未能在测试阶段充分做好检测工作,提交用户的软件质量差,在运行中暴露出大量的问题。在应用领域工作的不可靠软件,轻者影响系统的正常工作,重者发生事故,甚至造成生命财产的重大损失。

这些矛盾表现在具体的软件开发项目上,最突出的实例就是美国 IBM 公司在 1963 年至 1966 年开发的 IBM 360 机的操作系统。该项目花了 5000 人年的工作量,最多时有 1000 人投入开发工作,写出了近 100 万行源程序。尽管投入了这样的人力和物力,得到的结果却非常糟。据统计,这个操作系统每次发行的新版本都是从前一版本中找出 1000 个程序错误而修正的结果。可以设想这样的软件质量糟到什么地步。难怪这个项目的负责人 F. D. Brooks 事后总结了他在组织开发过程中的沉痛教训时说:“.....正像一只逃亡的野兽落到泥潭中做垂死的挣扎,越是挣扎,陷得越深。最后无法逃脱灭顶的灾难.....程序设计工作正像这样一个泥潭.....一批批程序员被迫在泥潭中拼命挣扎.....谁也没有料到问题竟会陷入这样的困境.....”IBM 360 操作系统的历史教训成为软件开发项目的典型事例为人们所记取。

以上这些矛盾多少描绘了软件危机的某些侧面, 如果这些障碍不能突破, 进而摆脱困境, 软件的发展是没有出路的。

1.3 软件工程过程和软件生存期

从上述软件危机的现象和发生危机的原因可以看出, 想摆脱危机不能只从一两个方面着手解决。如何针对软件的特点, 把它与其它产业部门工作对象的相同和相异之处加以分析, 排除人们的一些传统观念和某些错误认识, 成为非常重要的任务。只有端正了对软件的认识, 真正抓住了它的特点和发展趋势, 才能逃出危机, 走上软件发展的正确道路。

开发一个软件, 除去那些规模很小的项目以外, 通常要由多个软件人员分工合作、共同完成; 开发阶段之间的工作也应有很好的衔接; 开发工作完成以后, 软件成果要面向用户, 在应用中接受用户的检验。所有这些活动都要求人们改变过去那种把软件当做个人才智产物的观点, 抛弃那些只按自己工作习惯、不顾与周围其它人员配合关系的做法。在这一点上, 软件开发与计算机硬件研制, 甚至与高楼建设没有本质的差别。任何参加这些工程项目的人员, 它们的才能只有在工程项目的总体要求和技術规范的约束下充分发挥和施展。许多计算机和软件科学家尝试, 把其它工程领域中行之有效的工程学知识运用到软件开发工作中来。经过不断实践和总结, 最后得出一个结论: 按工程化的原则和方法组织软件开发工作是有意义的, 也是摆脱软件危机的一个主要出路。

1.3.1 软件工程过程 (software engineering process)

软件工程过程是为获得软件产品, 在软件工具支持下由软件人员完成的一系列软件工程活动。每个软件开发机构都可以规定自己的软件工程过程。针对不同类型的软件产品, 同一软件开发机构也可能使用多个不同的软件工程过程。但无论是哪种情况, 软件工程过程通常包含四种基本的过程活动。

- 1) 软件规格说明: 规定软件的功能及其运行的限制;
- 2) 软件开发: 产生满足规格说明的软件;
- 3) 软件确认: 确认软件能够完成客户提出的要求;
- 4) 软件演进: 为满足客户的变更要求, 软件必须在使用的过程中演进。

事实上, 软件工程过程是一个软件开发机构针对某一类软件产品为自己规定的工作步骤, 它应当是科学的、合理的, 否则必将影响到软件产品的质量。因此, 有理由要求软件工程过程具有如下的特性:

- 1) 可理解性。
- 2) 可见性: 每个过程活动均能以取得明确的结果告终, 使过程的进展对外可见。
- 3) 可支持性: 易于得到计算机辅助软件工程(CASE)工具的支持。
- 4) 可接受性: 易于为软件工程师接受和使用。
- 5) 可靠性: 不会出现过程错误, 或使其在出现产品故障之前即被发现。
- 6) 健壮性: 不受意外发生问题的干扰。
- 7) 可维护性: 过程可随软件机构需求的变更或随认定的过程改进而演进。

8) 速度: 从给出规格说明起, 就能较快地完成开发而交付。

1.3.2 软件生存期 (life cycle)

如同任何事物一样, 软件也有一个孕育、诞生、成长、成熟、衰亡的生存过程, 我们称其为计算机软件的生存期。根据这一思想, 把上述基本的过程活动进一步展开, 可以得到软件生存期的六个步骤, 即制定计划、需求分析、设计、程序编码、测试及运行维护。以下对这六个步骤的任务作一概括的描述。

1) 制定计划 (planning)

确定要开发软件系统的总目标, 给出它的功能、性能、可靠性以及接口等方面的要求; 由分析员和用户合作, 研究完成该项软件任务的可行性, 探讨解决问题的可能方案, 并对可利用的资源(计算机硬件、软件、人力等)、成本、可取得的效益、开发的进度做出估计, 制定出完成开发任务的实施计划, 连同可行性研究报告, 提交管理部门审查。

2) 需求分析和定义 (requirement analysis and definition)

对待开发软件提出的需求进行分析并给出详细的定义。软件人员和用户共同讨论决定: 哪些需求是可以满足的, 并对其加以确切地描述。然后编写出软件需求说明书或系统功能说明书以及初步的系统用户手册, 提交管理机构评审。

3) 软件设计 (software design)

设计是软件工程的技术核心。在设计阶段, 设计人员把已确定了的各项需求转换成一个相应的体系结构。结构中的每一成份都是意义明确的模块, 每个模块都和某些需求相对应, 即所谓概要设计。进而对每个模块要完成的工作进行具体的描述, 为源程序编写打下基础, 即所谓详细设计。所有设计中的考虑都应设计说明书的形式加以描述, 以供后续工作使用, 并提交评审。

4) 程序编写 (coding, programming)

把软件设计转换成计算机可以接受的程序代码, 即写成以某一种特定程序设计语言表示的“源程序清单”。这一步工作称为编码。自然, 写出的程序应当是结构良好、清晰易读, 且与设计相一致。

5) 软件测试 (testing)

测试是保证软件质量的重要手段, 其主要方式是在设计测试用例的基础上检验软件的各个组成部分。首先是进行单元测试, 查找各模块在功能和结构上存在的问题并加以纠正; 其次是进行组装测试, 将已测试过的模块按一定顺序组装起来; 最后按规定的各项需求, 逐项进行确认测试, 决定已开发的软件是否合格, 能否交付用户使用。

6) 运行/维护 (running/maintenance)

已交付的软件投入正式使用, 便进入运行阶段。这一阶段可能持续若干年甚至几十年。软件在运行中可能由于多方面的原因, 需要对它进行修改。其原因可能有: 运行中发现了软件中的错误需要修正; 为了适应变化了的软件工作环境, 需做适当变更; 为了增强软件的功能需做变更等。

1.4 软件生存期模型

类似于其它工程项目中安排各道工序那样,为反映软件生存期内各种活动应如何组织,上节所给出的六个步骤应如何衔接,需要用软件生存期模型做出直观的图示表达。

软件生存期模型是从软件项目需求定义直至软件经使用后废弃为止,跨越整个生存期的系统开发、运作和维护所实施的全部过程、活动和任务的结构框架。

到现在为止,已经提出了多种软件生存期模型。例如,瀑布模型、演化模型、螺旋模型、喷泉模型和智能模型等。

1.4.1 瀑布模型 (waterfall model)

瀑布模型规定了各项软件工程活动,包括:制定开发计划,进行需求分析和说明,软件设计,程序编码。测试及运行维护,参看图 1.4。并且规定了它们自上而下,相互衔接的固定次序,如同瀑布流水,逐级下落。

图 1.4 软件生存期的瀑布模型

然而软件开发的实践表明,上述各项活动之间并非完全是自上而下、呈线性图式。实际情况是,每项开发活动均应具有以下特征:

- 1) 从上一项活动接受本项活动的工作对象,作为输入;
- 2) 利用这一输入实施本项活动应完成的内容;
- 3) 给出本项活动的工作成果,作为输出传给下一项活动;

4) 对本项活动实施的工作进行评审。若其工作得到确认,则继续进行下一项活动,在图 1.4 中用向下指的箭头表示;否则返回到前项,甚至更前项的活动进行返工,在图 1.4 中由向上指的箭头表示。

需要说明的是,软件维护在软件生存期中有它的特点。一方面,维护的具体要求是在软件投入运行以后提出来的,经过“评价”,确定变更的必要性,才进入维护工作。另一方面,所谓维护中对软件的变更仍然要经历上述软件生存期在开发中已经历过的各项活动。如果把这些活动一并表达,就构成了生存期循环,如图 1.5 所示。

图 1.5 软件生存期循环

事实上,有人把维护称为软件的二次开发,正是出于这种考虑。由于软件在投入使用以后可能经历多次变更,为把开发活动和维护活动区别开来,便有了 b 形的软件生存期表示,如图 1.6 所示。

图 1.6 具有维护循环的软件生存期

1.4.2 演化模型 (evolutional model)

由于在项目开发的初始阶段人们对软件的需求认识常常不够清晰,因而使得开发项目难以做到一次开发成功,出现返工再开发在所难免。有人说,往往要“干两次”后开发出的软件才能较好地令用户满意。第一次只是试验开发,其目标只是在于探索可行性,弄清软件需求;第二次则在此基础上获得较为满意的软件产品。通常把第一次得到的试验性产品称为“原型”。显然,演化模型在克服瀑布模型缺点、减少由于软件需求不明确而给开发工作带来风险方面,确有显著的效果。

1.4.3 螺旋模型 (spiral model)

对于复杂的大型软件,开发一个原型往往达不到要求。螺旋模型将瀑布模型与演化模型结合起来,并且加入两种模型均忽略了的风险分析,弥补了两者的不足。

在制定软件开发计划时,系统分析员必须回答:项目的需求是什么,需要投入多少资

源以及如何安排开发进度等一系列问题。然而,若要他们当即给出准确无误的回答是不容易的,甚至几乎是不可能的。但系统分析员又不可能完全回避这一问题。凭借经验的估计出发给出初步的设想便难免带来一定风险。实践表明,项目规模越大,问题越复杂,资源、成本、进度等因素的不确定性越大,承担项目所冒的风险也越大。总之,风险是软件开发不可忽视的潜在不利因素,它可能在不同程度上损害到软件开发过程或软件产品的质量。驾驭软件风险的目标是在造成危害之前,及时对风险进行识别、分析,采取对策,进而消除或减少风险的损害。

螺旋模型沿着螺线旋转,如图 1.7 所示,在笛卡尔坐标的四个象限上分别表达了四个方面的活动,即:

- 1) 制定计划——确定软件目标,选定实施方案,弄清项目开发的限制条件;
- 2) 风险分析——分析所选方案,考虑如何识别和消除风险;
- 3) 实施工程——实施软件开发;
- 4) 客户评估——评价开发工作,提出修正建议。

图 1.7 螺旋模型

沿螺线自内向外每旋转一圈便开发出更为完善的一个新的软件版本。例如,在第一圈,确定了初步的目标、方案和限制条件以后,转入右上象限,对风险进行识别和分析。如果风险分析表明,需求有不确定性,那么在右下的工程象限内,所建的原型会帮助开发人员和客户,考虑其它开发模型,并对需求做进一步修正。

客户对工程成果做出评价之后,给出修正建议。在此基础上需再次计划,并进行风险分析。在每一圈螺线上,风险分析的终点做出是否继续下去的判断。假如风险过大,开发者和用户无法承受,项目有可能终止。多数情况下沿螺线的活动会继续下去,自内向外,逐步延伸,最终得到所期望的系统。

如果软件开发人员对所开发项目的需求已有了较大把握,则无需开发原型,可采用普通的瀑布模型。这在螺旋模型中可认为是单圈螺线。与此相反,如果对所开发项目的需求理解较差,则需要开发原型,甚至需要不止一个原型的帮助,那就需要经历多圈螺线。在这种情况下,外圈的开发包含了更多的活动。也可能某些开发采用了不同的模型。

螺旋模型适合于大型软件的开发,可谓最为实际的方法,它吸收了软件工程“演化”的概念,使得开发人员和客户对每个演化层出现的风险有所了解,继而做出应有的反映。

1.4.4 喷泉模型 (water fountain model)

瀑布模型的另一个不足之处在于,它对软件复用和生存期中多项开发活动的集成并未提供支持,因而难于支持面向对象的开发方法。“喷泉”一词本身体现了迭代和无间隙特性。系统某个部分常常重复工作多次,相关功能在每次迭代中随之加入演进的系统。所谓无间隙是指在开发活动,即分析、设计和编码之间不存在明显的边界。

1.4.5 智能模型 (intelligence model)

智能模型也称为基于知识的软件开发模型,它综合了上述若干模型,并把专家系统结合在一起。该模型基于规则,采用归约和推理机制,帮助软件人员完成开发工作,并使维护在系统规格说明一级进行。为此,建立了知识库,将模型本身、软件工程知识与特定领域的知识分别存入数据库。以软件工程知识为基础的生成规则构成的专家系统与含有应用领域知识规则的其它专家系统相结合,构成了这一应用领域软件的开发系统。

1.5 软件工程的基本目标

1.5.1 软件工程的定义

Boehm 曾经为软件工程下了定义:“运用现代科学技术知识设计并构造计算机程序及为开发、运行和维护这些程序所必需的相关文件资料”。这里对“设计”一词应有广义的理解,它应包括软件的需求分析和对软件进行修改时所进行的再设计活动。1983 年 IEEE 给出的定义为:“软件工程是开发、运行、维护和修复软件的系统方法”,其中,“软件”的定义为:计算机程序、方法、规则、相关的文档资料以及在计算机上运行时所必需的数据。后来尽管又有一些人提出了许多更为完善的定义,但主要思想都是强调在软件开发过程中需要应用工程化原则的重要性。

软件工程包括三个要素:方法、工具和过程。

软件工程方法为软件开发提供了“如何做”的技术。它包括了多方面的任务,如项目计划与估算、软件系统需求分析、数据结构、系统总体结构的设计、算法过程的设计、编码、测试以及维护等。软件工程方法常采用某一种特殊的语言或图形的表达方法及一套质量保证标准。

软件工具为软件工程方法提供了自动的或半自动的软件支撑环境。目前,已经推出了许多软件工具,已经能够支持上述的软件工程方法。特别地,已经有人把诸多的软件工具集成起来,使得一种工具产生的信息可以为其它的工使用,这样建立起一种被称之为

计算机辅助软件工程(CASE)的软件开发支撑系统。CASE 将各种软件工具、开发机器和一个存放开发过程信息的工程数据库组合起来形成一个软件工程环境。

软件工程的过程则是将软件工程的方法和工具综合起来以达到合理、及时地进行计算机软件开发的目的。过程定义了方法使用的顺序、要求交付的文档资料、为保证质量和协调变化所需要的管理及软件开发各个阶段完成的里程碑。

软件工程就是包含上述方法、工具及过程在内的一些步骤。

1.5.2 软件工程项目的基本目标

组织实施软件工程项目,从技术上和管理上采取了多项措施以后,最终希望得到项目的成功。所谓成功指的是达到以下几个主要的目标:

- 付出较低的开发成本;
- 达到要求的软件功能;
- 取得较好的软件性能;
- 开发的软件易于移植;
- 需要较低的维护费用;
- 能按时完成开发工作,及时交付使用。

在具体项目的实际开发中,企图让以上几个目标都达到理想的程度往往是非常困难的。而且上述目标很可能是互相冲突的。例如,若只顾降低开发成本,很可能同时也降低了软件的可靠性。另一方面,如果过于追求提高软件的性能,可能造成开发出的软件对硬件有较大的依赖,从而直接影响到软件的可移植性。

图 1.8 表明了软件工程目标之间存在的相互关系。其中有些目标之间是互补关系,例如,易于维护和高可靠性之间、低开发成本与按时交付之间。还有一些目标是彼此互斥的,例如,上面指出的互相冲突的情况。

图 1.8 软件工程目标之间的关系

第 2 章 软件需求分析

软件需求分析工作是软件生存期中重要的一步,也是决定性的一步。只有通过需求分析才能把软件功能和性能的总体概念描述为具体的软件需求规格说明,从而奠定软件开发的基础。软件需求分析工作也是一个不断认识和逐步细化的过程。该过程将软件计划阶段所确定的软件范围逐步细化到可详细定义的程度,并分析出各种不同的软件元素,然后为这些元素找到可行的解决方法。软件开发的实践表明,需求分析并不是一件轻而易举的事情。软件危机发生的原因之一就是忽视了需求分析这一重要的步骤。往往是软件开发人员和用户未能全面地、准确地理解需求,或是未能恰当地表达这些需求,以致把需求分析阶段的问题遗留到开发工作的后续阶段,最终酿成不良后果。

制定软件的需求规格说明不只是软件开发人员的事,用户也起着至关重要的作用。用户必须对软件功能和性能提出初步要求,并澄清一些模糊概念。而软件分析人员则要认真学习用户的要求,细致地进行调查分析,把用户“做什么”的要求最终转换成一个完全的、精细的软件逻辑模型并写出软件的需求规格说明,准确地表达用户的要求。

2.1 软件需求分析概述

2.1.1 软件需求分析的任务

需求分析阶段研究的对象是软件项目的用户要求。需要注意的是,必须全面理解用户的各项要求,但又不能全盘接受所有的要求。因为并非所有用户提出的全部要求都是合理的。对其中模糊的要求还需要澄清,然后才能决定是否可以采纳。对于那些无法实现的要求应向用户做充分的解释,以求得谅解。

准确地表达被接受的用户要求,是需求分析的另一个重要方面。只有经过确切描述的软件需求才能成为软件设计的基础。

通常软件开发项目是要实现目标系统的物理模型,就是要确定被开发软件系统的系统元素,并将功能和信息结构分配到这些系统元素中。它是软件实现的基础。但是目标系统的具体物理模型是由它的逻辑模型经实例化,即具体到某个业务领域而得到的。与物理模型不同,逻辑模型忽视实现机制与细节,只描述系统要完成的功能和要处理的信息。作为目标系统的参考,需求分析的任务就是借助于当前系统的逻辑模型导出目标系统的逻辑模型,解决目标系统的“做什么”的问题。其实现步骤如图 2.1 所示。

1) 获得当前系统的物理模型。所谓当前系统可能是需要改进的某个已在计算机上运行的数据处理系统,也可能是一个人工的数据处理过程。在这一步首先分析现实世界,到现场调查研究,理解当前系统是如何运行的,了解当前系统的组织机构、输入输出、资源利用情况和日常数据处理过程,并用一个具体模型反映自己对当前系统的理解。这一模型应客观地反映现实世界的实际情况。

图 2.1 参考当前系统建立目标系统模型

2) 抽象出当前系统的逻辑模型。在理解当前系统“怎样做”的基础上,抽取其“做什么”的本质,从而从当前系统的物理模型抽象出当前系统的逻辑模型。

在物理模型中有许多物理的因素,随着分析工作的深入,有些非本质的物理因素就成为不必要的负担,因而需要对物理模型进行分析,区分出本质的和非本质的因素,去掉那些非本质的因素即可获得反映系统本质的逻辑模型。

3) 建立目标系统的逻辑模型。分析目标系统与当前系统逻辑上的差别,明确目标系统到底要“做什么”,从而从当前系统的逻辑模型导出目标系统的逻辑模型。具体做法:

- 决定变化的范围,即决定目标系统与当前系统在逻辑上的差别;
- 将变化的部分看做是新的处理步骤,对数据流图进行调整;
- 由外向里对变化的部分进行分析,凭经验推断其结构,获得目标系统的逻辑模型。

4) 补充目标系统的逻辑模型。为了对目标系统做完整的描述,还需要对前面得到的结果做一些补充:

- 说明目标系统的用户界面。根据目标系统所处的应用环境及它与外界环境的相互关系,研究所有可能与它发生联系和作用的部分,从而决定人机界面。
- 说明至今尚未详细考虑的细节。这些细节包括系统的启动和结束、出错处理、系统的输入输出和系统性能方面的需求。
- 其它。例如系统的其它必须满足的性能和限制等等。

2.1.2 需求分析的过程

需求分析阶段的工作,可以分成以下四个方面:对问题的识别、分析与综合、制定规格说明和评审。下面分别介绍。

1) 问题识别

首先系统分析员要研究可行性分析报告(如果有的话)和软件项目实施计划。主要是从系统的角度来理解软件并评审用于产生计划估算的软件范围是否恰当,确定对目标系统的综合要求,即软件的需求,并提出这些需求实现条件,以及需求应达到的标准。也就是解决要求被开发软件做什么,做到什么程度。这些需求包括:

- 功能需求:列举出目标软件在职能上应做什么。这是最主要的需求。
- 性能需求:给出目标软件的技术性能指标,包括存储容量限制、运行时间限制等。

例如,求解一个大型结构(如地下机库,水库大坝)应力应变特性问题时对大型线性方程组的求解精度与存储容量、计算迭代的时间等;使用情报检索网络查找某

个专业资料信息时, 搜索命中率、检索响应时间、检索频度等都是系统的性能。

- 环境需求: 这是对目标系统运行时所处环境的要求。例如在硬件方面, 采用的机型、外部设备、数据通信接口等。在软件方面, 支持系统运行的系统软件 (操作系统、网络软件、数据库管理系统等)。在使用方面, 使用部门在制度上、操作人员的技术水平上应具备的条件等。
- 可靠性需求: 各种软件在运行时, 失效的影响各不相同。在需求分析时应针对目标软件在投入运行后不发生故障的概率, 按实际的运行环境提出要求。对于重要软件, 或是运行失效会造成严重后果的软件, 应提出较高的可靠性要求, 以期在开发的过程中采取必要的措施, 使软件产品能够高度可靠地稳定运行。
- 安全保密要求: 工作在不同环境的软件对其安全、保密的要求显然是不同的。应当把这方面的需求恰当地做出规定, 以便对被开发的软件给予特殊的设计, 使其在运行中其安全保密方面的性能得到必要的保证。
- 用户界面需求: 软件与用户界面的友好性为用户能够方便有效愉快地使用该软件的关键之一。例如, 现在流行的 Windows 系统以图形方式与用户交互, 有效地管理计算机内的各种资源。从市场角度来看, 具有友好用户界面的软件有很强的竞争力。因此, 必须在需求分析时, 为用户界面细致地规定达到的要求。
- 资源使用需求: 这是指目标软件运行时所需的数据、软件、内存空间等各项资源。另外, 软件开发时所需的人力、支撑软件、开发设备等则属于软件开发的资源, 需要在需求分析时加以确定。
- 软件成本消耗与开发进度需求: 在软件项目立项后, 要根据合同规定, 对软件开发的进度和各步骤的费用提出要求, 作为开发管理的依据。
- 预先估计以后系统可能达到的目标。这样, 在开发过程中, 可对系统将来可能的扩充与修改做准备。一旦需要时, 就比较容易进行补充和修改。

如果在做需求分析之前没有做过可行性分析, 那么补充完成这部分工作是必要的。在定义阶段尽早发现将来可能在开发过程中遇到的问题, 及早做出决定, 可以避免大量的人工、金钱、时间上的浪费。

可行性与风险分析在许多方面是相关的。如果项目的风险很大, 就会降低产生高质量软件的可行性。可行性研究主要集中在以下四个主要方面:

(1) 经济可行性。进行开发成本的估算及可能取得效益的评估, 确定目标系统是否值得投资开发。

(2) 技术可行性。对目标系统进行功能、性能和限制条件的分析, 确定在现有的资源的条件下, 技术风险有多大, 系统是否能实现。这里, 资源包括已有的或可以搞到的硬件、软件资源、现有技术人员的技术水平与已有的工作基础。

(3) 法律可行性。对目标系统可能会涉及的侵权、妨碍、责任等问题做出决定。

(4) 抉择。对系统开发的不同方案进行比较评估。成本和时间的限制会给方案的选择带来局限性, 对于一些合理的方案都应加以考虑。

在进行可行性研究时, 需要从问题定义和调查研究入手, 与用户密切联系, 详细了解问题提出的背景, 弄清要解决什么问题。然后从软件系统特性和用户目标出发, 做市场调

查和现场考察。仔细收集信息之后进行数据分析和功能分析,建立系统的高层逻辑模型,再进一步做成本/效益分析。最后提交一份可行性分析报告,从技术、经济、社会效应等方面论证可行性,以确认软件开发的目標是否可行。

问题识别的另一项工作是建立分析所需要的通信途径,以保证能顺利地对问题进行分析。分析所需的通信途径如图 2.2 所示。分析员必须与用户、软件开发机构的管理部門、软件开发组的人员建立联系。项目负责人在此过程中起协调人的作用。分析人员通过这种通信途径与各方商讨,以便能按照用户的要求去识别问题的基本内容。

图 2.2 软件需求分析的通信途径

2) 分析与综合

问题分析和方案的综合是需求分析的第二方面的工作。分析员必须从信息流和信息结构出发,逐步细化所有的软件功能,找出系统各元素之间的联系、接口特性和设计上的约束,分析它们是否满足功能要求、是否合理。依据功能需求、性能需求、运行环境需求等,剔除其不合理的部分,增加其需要部分。最终综合成系统的解决方案,给出目标系统的详细逻辑模型。

在这个步骤中,分析和综合工作反复地进行。在对现行问题和期望的信息(输入和输出)进行分析的基础上,分析员开始综合出一个或几个解决方案,然后检查它的工作是否符合软件计划中规定的范围等等,再进行修改。总之,对问题进行分析和综合的过程将一直持续到分析人员与用户双方都感到有把握正确地制定该软件的规格说明为止。

常用的分析方法有面向数据流的结构化分析方法、面向数据结构的 Jackson 方法、结构化数据系统开发方法、面向对象的分析方法等,对于要求时序的调度类软件,还可使用状态迁移图、Petri 网等。

3) 编制需求分析阶段的文档

已经确定下来的需求应当得到清晰准确的描述。通常把描述需求的文档叫做软件需求说明书或软件需求规格说明。同时为了确切表达用户对软件的输入输出要求,还需要制定数据要求说明书及编写初步的用户手册,反映目标软件的用户界面和用户使用的具体要求。此外,依据在需求分析阶段对系统的进一步分析,从目标系统的精细模型出发,可以

更准确地估计被开发项目的成本与进度,从而修改、完善与确定软件开发实施计划。

4) 需求分析评审

作为需求分析阶段工作的复查手段,在需求分析的最后一步,应该对功能的正确性、完整性和清晰性,以及其它需求给予评价。评审的主要内容是:

- 系统定义的目标是否与用户的要求一致;
- 系统需求分析阶段提供的文档资料是否齐全;
- 文档中的所有描述是否完整、清晰、准确反映用户要求;
- 与所有其它系统成份的重要接口是否都已经描述;
- 被开发项目的数据流与数据结构是否足够、确定;
- 所有图表是否清楚,在不补充说明时能否理解;
- 主要功能是否已包括在规定的软件范围之内,是否都已充分说明;
- 设计的约束条件或限制条件是否符合实际;
- 开发的技术风险是什么;
- 是否考虑过软件需求的其它方案;
- 是否考虑过将来可能会提出的软件需求;
- 是否详细制定了检验标准,它们能否对系统定义,是否成功进行确认;
- 有没有遗漏、重复或不一致的地方。

为保证软件需求定义的质量,评审应以专门指定的人员负责,并按规程严格进行。评审结束应有评审负责人的结论意见及签字。除分析员之外,用户/需求者、开发部门的管理者、软件设计、实现、测试的人员都应当参加评审工作。一般地,评审的结果都包括了一些修改意见,待修改完成后并经评审通过,才可进入设计阶段。

图 2.3 给出了需求分析阶段工作的流程。图中所谓的有效性准则,主要是指性能界限、测试种类、期望的软件响应及其它特殊考虑等。

2.1.3 软件需求分析的原则

近年来已提出了许多软件分析与说明的方法。虽然各种分析方法都有其独特的描述方法,但总的看来,所有分析方法还是有它们共同适用的基本原则。

1) 能够表达和理解问题的信息域和功能域

所有软件定义与开发工作最终是为了解决好数据处理问题,就是将一种形式的数据转换成另一种形式的数据。其转换过程必定经历输入、加工数据和产生结果数据等步骤。

对于计算机程序处理的数据,其信息域应包括信息流、信息内容和信息结构。信息流即数据通过一个系统时的变化方式。参看图 2.4。输入数据首先转换成中间数据,然后转换成输出数据。在此期间可以从已有的数据存储(如磁盘文件或内存缓冲区)中引入附加数据。对数据进行转换是程序中应有的功能或子功能。两个转换功能之间的数据传递就确定了功能间的接口。

信息内容是可构成更大信息项的单个数据项。例如,学生名册包含了班级、人数、每个学生的学号、姓名、各科成绩等。学生名册的内容由它所包含的项定义。为了理解对学生名册的处理,我们必须理解它的信息内容。

图 2.3 软件需求分析工作的流程图

信息结构是各种数据项的逻辑组织。数据是组织成 n 维表格, 还是组织成有层次的树型结构? 在结构中数据项与其它哪些数据项相关? 所有信息是在一个信息结构中, 还是在几个信息结构中? 一个结构中的数据与其它结构中的数据如何联系? 这些问题都由信息结构的分析来解决。

2) 能够对问题进行分解和不断细化, 建立问题的层次结构

通常软件要处理的问题, 作为一个整体来看, 显得太大, 太复杂, 很难理解。把问题以某种方式分解为几个较易理解的部分, 并确定各部分间的接口, 从而实现整体功能。

图 2.4 信息流

在需求分析阶段, 软件的功能域和信息域都能做进一步的分解。这种分解可以是同一层次上的, 称为横向分解; 也可以是多层次的纵向分解。可参看图 2.5。例如, 把一个功能分解成几个子功能, 并确定这些子功能与父功能的接口, 就属于横向分解。但如果继续分解, 把某些子功能又分解为小的子功能, 某个小的子功能又分解为更小的子功能, 这就属于纵向分解了。

3) 需要给出系统的逻辑视图和物理视图

给出系统的逻辑视图和物理视图, 这对系统满足处理需求所提出的逻辑限制条件和系统中其它成份提出的物理限制条件是必不可少的。

图 2.5 问题的分解

软件需求的逻辑视图给出的是软件要达到的功能和要处理的信息之间的关系,而不是实现的细节。例如,一个商店的销售处理系统要从顾客那里获取订单,系统读取订单的功能并不关心订单数据本身的物理形式和用什么设备读入,也就是说无需关心输入的机制,只是读取顾客的订单而已。类似地,系统中检查库存的功能只关心库存文件的数据结构,而不关心在计算机中的具体存储方式。软件需求的逻辑描述是软件设计的基础。

软件需求的物理视图给出的是处理功能和信息结构的实际表现形式,这往往是由设备本身决定的。比如一些软件靠终端键盘输入数据,另一些数据处理软件靠模数转换设备提供数据。软件分析人员必须弄清已确定的系统元素对软件的限制,并考虑功能和信息结构的物理表示。这并不是说要求分析人员在需求分析中考虑“如何实现”的具体问题,而是仅限于“做什么”的范围。

2.2 结构化分析方法

结构化分析方法(structured analysis, SA)是面向数据流进行需求分析的方法,是 20 世纪 70 年代末经 Yourdon E., Constantine L., DeMarco T. 等人提出和发展,适合于数据处理类型软件的需求分析方法。由于利用图形来表达需求显得清晰、简明,易于学习和掌握,近年来得到了广泛使用。

具体来说,结构化分析方法就是按照功能分解的原则,根据软件内部数据传递、变换的关系,自顶向下逐层分解,直到找到满足功能要求的所有可实现的软件为止。

根据 DeMarco 的论述,结构化分析方法使用了以下几个工具:数据流图、数据词典、结构化英语、判定表与判定树。其中,数据流图用以表达系统内数据的运动情况。数据词典定义系统中的数据。结构化英语、判定表和判定树都用以描述数据加工。

2.2.1 数据流图(DFD, data flow diagram)

数据流图是描述数据处理过程的有力工具。数据流图从数据传递和加工的角度,以图形的方式刻画数据流从输入到输出的移动变换过程。

1) 数据流图中的主要图形元素

这里,以人们熟悉的事务处理——去银行取款为例,说明数据流图如何描述数据处理过程。图 2.6 表示储户携带存折前去银行办理取款手续。他应把存折和填好的取款单一并交给银行工作人员检验。工作人员需核对账目,发现存折有效性问题、取款单填写问题

或是存折、账卡与取款单不符等问题时, 均应报告储户。在检验通过的情形下, 银行则应将取款信息登记在存折和账卡上, 并通知付款。根据付款通知给储户付款, 从而完成这一简单的数据处理活动。

图 2.6 办理取款手续的数据流图

从数据流图中可知, 数据流图的基本图形元素有 4 种, 如图 2.7 所示。

图 2.7 DFD 的基本图形符号

数据流是沿箭头方向传送的数据, 在加工之间传输的数据流一般是有名的, 连接数据存储文件和加工的数据流有些没有命名, 这些数据流虽然没有命名, 但因所联接的是有名加工和有名文件, 所以其含意也是清楚的。

同一数据流图上不能有两个数据流名字相同。

加工是以数据结构或数据内容作为加工对象的。加工的名字应是一个动词性短语, 简明扼要地表明完成的是什么加工。

文件在数据流图中起保存数据的作用, 因而称为数据存储。它可以是数据库文件或任何形式的数据组织。指向文件的数据流可理解为写入文件或查询文件, 从文件中引出的数据流可理解为从文件读取数据或得到查询结果。

数据流图中第四种元素是数据源点或汇点, 它表示图中要处理数据的输入来源或处理结果要送往何处。它不属于数据流图的核心部分, 只是数据流图的外围环境中的实体部分, 故称外部实体。在实际问题中它可能是人员、计算机外围设备或是传感装置。

2) 数据流与加工之间的关系

在数据流图中, 如果有两个以上数据流指向一个加工, 或是从一个加工中引出两个以上的数据流, 这些数据流之间往往存在一定的关系。为表达这些关系, 在这些数据流的加

工附近可以标上不同的标记符号。这里以向某一加工流入两个数据流或流出两个数据流为例,说明其间符号的作用。所用符号及其含意在图 2.8 中给出。

图 2.8 表明多个数据流与加工之间关系的符号

其中星号“ * ”表示相邻的一对数据流同时存在,有“ 与 ”的含义;“ ”则表示相邻的两数据流只取其一,有“ 或 ”之意。

3) 为了表达数据处理过程的数据加工情况,用一个数据流图是不够的。稍微复杂的实际问题,在数据流图上常常出现十几个甚至几十个加工。这样的数据流图看起来很不清楚。分层的数据流图能很好地解决这一问题。按照系统的层次结构进行逐步分解,并以分层的数据流图反映这种结构关系,能清楚地表达和容易理解整个系统。

可以把整个数据处理过程暂且看成一个加工,它的输入数据和输出数据实际上反映了本系统与外界环境的接口。这就是分层数据流图的顶层。但仅此一图并未表明数据的加工要求,需要进一步细化。如果这个数据处理 S 包括三个子系统,就可以画出表示三个子系统 1, 2, 3 的加工及其相关的数据流,参看图 2.9。这是顶层下面的第一层数据流图,

图 2.9 分层数据流图

记为 DFD/L1。继续分解三个子系统,可得到第二层数据流图 DFD/L2.1, DFD/L2.2 及 DFD/L2.3,它们分别是子系统 1, 2 和 3 的细化。以 DFD/L2.2 为例,其中的 4 个加工的编号均可联系到其上层图中的子系统 2。这样得到的多层数据流图可十分清晰地表达整个数据加工系统的真实情况。对任何一层数据流图来说,它的上层图为父图,在它下一层

的图则为子图。必须注意, 各层数据流图之间应保持“平衡”关系。例如, 图 2.9 中 DFD/L1 中子系统 3 有两个输入数据流和一个输出数据流, 那么它的子图 DFD/L2.3 也要有同样多的输入数据流和输出数据流, 才能符合子图细化的实际情况。

在多层数据流图中, 可以把顶层流图、底层流图和中间层流图区分开来。顶层流图仅包含一个加工, 它代表目标系统。它的输入流是该系统的输入数据, 输出流是该系统的输出数据。顶层流图的作用在于表明目标系统的范围, 以及它和周围环境的数据交换关系。为逐层分解打下基础。而底层流图是指其加工不需再做分解的数据流图, 它处在最底层, 也称其加工为“基本加工”。中间层流图则表示对其上层父图的细化。它的每一加工可能继续细化, 形成子图。

4) 数据流图画法要求

上面的例子给出数据流图的一个概况。下面对数据流图的画法规定一些要求。

- (1) 数据流图上所有图形符号只限于前述四种基本图形元素。
- (2) 数据流图的主图必须包括前述四种基本元素, 缺一不可。
- (3) 每个加工至少有一个输入数据流和一个输出数据流。
- (4) 可以在数据流图中加入物质流, 帮助用户理解数据流图。
- (5) 数据流图中不可夹带控制流。因为数据流图是实际业务流程的客观映象, 说明系统“做什么”而不是要表明系统“如何做”, 不是系统的执行顺序, 不是程序流程图。
- (6) 初画时可以忽略琐碎的细节, 以集中精力于主要数据流。一些枝节问题可以缓一步再画。例如从某些加工框出来的错误信息可以不画在数据流图中, 关于异常状态的处理留待设计阶段处理。

需要说明的是, 为了使数据流图便于在计算机上输入和输出, 免去画曲线、斜线和圆的困难, 常常使用另一套符号, 这一套符号与图 2.8 给出的符号完全等价, 如图 2.10 所示。

图 2.10 符号对照

采用这组符号画出的图 2.6 所示的银行取款过程的数据流图在图 2.11 中给出。

为便于存放数据流图, 可将它表示成等价矩阵的形式。图 2.11 所示取款数据流图的等价矩阵如图 2.12 所示。其中的 E1 和 E2 是数据文件; A, B, C, ..., H, I, J 是数据流。

2.2.2 数据词典(DD, data dictionary)

数据词典是结构化分析方法的另一个工具, 它与数据流图配合, 能清楚地表达数据处理的要求。数据流图给出系统的组成及其内部各元素相互间的关系, 但未说明数据元素的

图 2.11 银行取款过程的数据流图

图 2.12 等价矩阵

具体含意。仅靠数据流图人们无法理解它所描述的对象。数据词典的任务是对于数据流图中出现的所有命名元素,包括数据流、加工、数据文件,以及数据的源、汇点等,在数据词典中作为一个词条加以定义,使得每一个图形元素的名字都有一个确切的解释。

数据词典中所有的定义应是严密的、精确的,不可有半点含混,不可有二义性。

1) 词条描述

对于在数据流图中每一个被命名的图形元素,均加以定义,其内容有:

- 图形元素的名字:某一词条的名字,要求无二义性,为人们所公认;
- 别名或编号;
- 分类:加工,数据流,数据文件,数据元素,数据源、汇点等;
- 描述:功能,特点;
- 定义:该词条的组成,数据结构等;
- 位置:数据流的来源、去处,加工框的编号、输入、输出,数据元素在哪个数据结构中等;
- 其它:数据流的数据量、流通量;数据文件的存储方式、存取要求;数据加工的加工顺序,以及外部实体的数量等。

2) 数据结构的描述

在数据流图中,数据流和数据文件都具有一定的数据结构,必须以一种清晰、准确、无二义性的方式描述。下面给出的定义方式是一种严格的描述方式。表2.1给出在数据词典的定义式中常出现的一些符号。

表 2.1 数据词典定义式中的符号

符号	含 义	解 释
=	被定义为	
+	与	例如, $x = a + b$, 表示 x 由 a 和 b 组成。
[...,...]	或	例如, $x = [a, b]$, $x = [a@b]$, 表示 x 由 a 或由 b 组成。
[...@l...]	或	
{...}	重 复	例如, $x = \{a\}$, 表示 x 由 0 个或多个 a 组成。
m{...}n	重 复	例如, $x = 3\{a\}8$, 表示 x 中至少出现 3 次 a , 至多出现 8 次 a 。
(...)	可 选	例如, $x = (a)$, 表示 a 可在 x 中出现, 也可不出现。
"..."	基本数据元素	例如, $x = "a"$, 表示 x 为取值为 a 的数据元素。
..	连结符	例如, $x = 1..9$, 表示 x 可取 1 到 9 之中的任一值。

在图 2.11 表示的取款数据流图中,数据文件“存折”的格式如图 2.13 所示,它在数据词典中的定义格式为:

图 2.13 存折格式

存折= 户名+ 所号+ 账号+ 开户日+ 性质+ (印密)+ 1{存取行}50
户名= 2{字母}24
所号= “001”..“999” 注: 储蓄所编码, 规定 3 位数字
账号= “00000001”..“99999999” 注: 账号规定由 8 位数字组成
开户日= 年+ 月+ 日
性质= “1”..“6” 注: “1”表示普通户, “5”表示工资户等
印密= “0” 注: 印密在存折上不显示
存取行= 日期+ (摘要)+ 支出+ 存入+ 余额+ 操作+ 复核
日期= 年+ 月+ 日
.....
这表明存折是由 6 部分组成, 第 6 部分的“存取行”要重复出现多次。如果重复的次数

是常数,例如为 50,则可表示为{存取行}50 或 {存取行}⁵⁰;如果重复的次数是变量,那么要估计其变动范围。例如,存取行从 1 到 50,则可记为 1{存取行}50。

这种定义方法是从顶向下,逐级给出定义式,直到最后给出基本数据元素为止。

3) 数据词典的使用

在结构化分析的过程中,可以通过名字方便地查阅数据的定义;同时可按各种要求,随时列出各种表,以满足分析员的需要。还可以反过来,按描述内容(或定义)查询数据的名字。通过检查各个加工的逻辑功能,可以实现和检查在数据与程序之间的一致性和完整性。

2.2.3 加工逻辑说明

在数据流图中,每一个加工框中只简单地写上了一个加工名,这显然不能表达加工的全部内容。随着自顶向下逐层细化,功能越来越具体,加工逻辑也越来越精细。到底一层,加工逻辑详细到可以实现的程度,因此称为“基本加工”。如果我们写出每一个基本加工的全部详细逻辑功能,再自底向上综合,就能完成全部逻辑加工。

在写基本加工逻辑的说明时,主要目的是要表达“做什么”,而不是“怎样做”。因此它应满足如下的要求:

- 对数据流图的每一个基本加工有一个加工逻辑说明;
- 加工逻辑说明描述基本加工如何把输入数据流变换为输出数据流的加工规则;
- 加工逻辑说明描述实现加工的策略而不是实现加工的细节;
- 加工逻辑说明中包含的信息应是充足的、完备的、有用的、没有重复的多余信息。

目前用于写加工逻辑说明的工具具有结构化英语、判定表和判定树。

1) 结构化英语(structured english)

结构化英语也叫做程序设计语言(program design language),简称 PDL,是一种介于自然语言和形式化语言之间的半形式化语言。它是在自然语言基础上加了一些限制而得到的语言。它使用有限的词汇和有限的语句来描述加工逻辑。

结构化英语的词汇表由英语命令动词、数据词典中定义的名字、有限的自定义词和逻辑关系词 IF_THEN_ELSE, WHILE_DO, REPEAT_UNTIL, CASE_OF 等组成。其动词的含义要具体,不要用抽象的动词。尽可能少用或不用形容词和副词。

语言的正文用基本控制结构进行分割,加工中的操作用自然语言短语表示。其基本控制结构有三种:

- 简单陈述句结构:避免复合语句;
- 判定结构:IF_THEN_ELSE 或 CASE_OF 结构;
- 重复结构:WHILE_DO 或 REPEAT_UNTIL 结构。

此外在书写时,必须按层次横向向右移行,续行也同样向右移行,对齐。

下面是商店业务处理系统中“检查发货单”的例子。

IF the invoice exceeds \$ 500 THEN

IF the account has any invoice more than 60 days overdue THEN

发货单金额超过 \$ 500

欠款超过 60 天

the confirmation pending resolution of the debt	在偿还欠款前不予批准
ELSE (account is in good standing)	欠款未超期
issue confirmation and invoice	发批准书及发货单
ENDIF	
ELSE (invoice \$ 500 or less)	发货单金额未超过 \$ 500
IF the account has any invoice more than 60 days overdue THEN	欠款超过 60 天
issue confirmation, invoice and write messagy on credit action report	
	发批准书, 发货单及赊欠报告
ELSE (account is in good standing)	欠款未超期
issue confirmation and invoice	发批准书及发货单
ENDIF	
ENDIF	

关于结构化英语的进一步讨论, 将在第 4 章讲述 PDL 时介绍。

2) 判定表(decision table)

在某些数据处理问题中, 某数据流图的加工需要依赖于多个逻辑条件的取值, 就是说完成这一加工的一组动作是由于某一组条件取值的组合而引发的。这时使用判定表来描述比较合适。因为这时需要描述的加工是由一组操作组成的, 其中有些操作是否执行又取决于一组条件。若使用判定表, 比较容易保证所有条件和操作都被说明, 不容易发生错误和遗漏。

下面以“ 检查发货单 ”为例, 说明判定表的构成。参看图 2. 14。

图 2. 14 “ 检查发货单 ”的判定表

判定表由四个部分组成, 如图 2. 15 所示。双线分割开的 4 部分中, 左上部分是条件荏, 列出了各种可能的条件。左下部分是动作荏, 列出了可能采取的动作。右上部分是条件项, 是针对各种条件给出的多组条件取值的组合。右下部分是动作项, 是和条件项紧密相关的, 它指出了在条件项的各组取值的组合情况下应采取的动作。

通常将任一条件取值组合及其相应要执行的动作称为规则, 它在判定表中是纵向贯穿条件项和动作项的一列。显然, 判定表中列出了多少个条件取值的组合, 也就有多少条规则。

在实际使用判定表时, 常常先把它化简。如果表中有两条或更多的规则具有相同的动

作, 并且其条件项之间存在着某种关系, 就可设法将它们合并。例如图 2. 16(a) 表示了两个规则的动作项一致, 条件项中的第三条件的取值不同, 这表明在第一、第二条件分别取真值和假值时, 第三条件不论取何值, 都执行同一动作。这样, 我们将这两条规则合并, 合并后的第三条件取值用“-”表示, 以示与取值无关。类似地, 无关条件项“-”在逻辑上又可包含其它的条件项取值, 具有相同动作的规则还可进一步合并, 如图 2. 16(b) 所示。

图 2. 15 判定表的结构

图 2. 16 动作相同的规则合并

判定表能够把在什么条件下, 系统应完成哪些操作表达得十分清楚、准确、一目了然。这是用语言说明难以准确、清楚表达的。但是用判定表描述循环比较困难。有时, 判定表可以和结构化英语结合起来使用。

3) 判定树(decision tree)

判定树也是用来表达加工逻辑的一种工具。有时候它比判定表更直观。用它来描述加工, 很容易为用户接受。

图 2. 17 判定树

下面把前面的“检查发货单”的例子用判定树表示。参看图 2. 17。在表达一个基本加工逻辑时, 结构化英语、判定表和判定树常常被交叉使用, 互相补充。因为这三种手段各有优缺点。对于不太复杂的判定条件, 或者使用判定表有困难时, 使用判定树较好。而在一个加工逻辑中, 如同时存在顺序、判断和循环时, 使用结构化英语较好。最后, 对于复杂的判定, 组合条件较多, 则使用判定表较好。

总之, 加工逻辑说明是结构化分析方法的一个组成部分, 使用的手段应当以结构化英语为主, 对存在判断问题的加工逻辑, 可辅之以判定表和判定树。

2.3 结构化数据系统开发方法(DSSD) ——面向数据结构的分析方法之一

结构化数据系统开发方法(DSSD, data structured systems development), 也称为 Warnier-Orr 方法, 是由 J. D. Warnier 提出的。他提出利用三种基本构造, 即顺序、选择、重复构造表示信息的分层结构, 并进而由数据结构直接导出软件结构。Orr 将其扩充, 形成了结构化数据系统开发方法。该方法考虑了信息流和功能特性以及数据的分层关系。

2.3.1 Warnier 图

Warnier 图是表示层次信息结构的一种紧凑而直观的方式, 很容易被人们理解。以报纸的自动编辑系统为例。通常报纸的版面采用以下格式。

头版部分	社论部分	副刊部分
头条新闻	社论	体育新闻
国内新闻	专栏	商业新闻
当地新闻	读者来信	广告
	讽刺漫画	

上面给出的报纸概观就是一个信息的层次结构, Warnier 图可在细节的任一层次上表示层次结构。图 2.18(a) 给出了用 Warnier 图表示的报纸的信息层次结构。

图 2.18 报纸编辑的信息层次结构

在这个 Warnier 图中, 用大括号“{”表示层次关系, 在同一括号下, 自上到下是顺序排列的信息项。在有些信息项名字的后面附加了圆括号, 给出该信息项重复的次数。例如, 社论 (1, 1) 表示社论占一栏; 专栏 (1, 3) 表示专栏占 1 到 3 栏; 讽刺漫画 (0, 1) 表示讽刺漫画可有可无, 若有就占一栏。

另外, Warnier 图可以通过细化复合信息项进一步分解信息域。图 2.18(b) 是副刊部分的细化结果。异或符号()表示对位于它上下两边的信息项条件的选择, 不是经营信息, 就是雇员信息, 二者择一。

2.3.2 DSSD 的分析方法

用 DSSD 方法进行分析时,不是从考察信息的层次结构开始,而是首先研究“应用环境”。即分别站在信息的产生者和接受者的角度,观察信息如何在产生者和接受者之间流动。然后,用类似于 Warnier 图的表示方法描述信息项和对信息项的处理,从而确定问题的功能。最后,利用 Warnier 图建立问题结果的模型。使用这一方法做需求分析,会涉及到信息项的所有属性:数据流、数据内容和数据结构。

下面结合一个简单的电话订购系统的实例,说明 DSSD 方法的步骤。为了表达电话订购业务的整体信息流,使用了数据流图。其实,数据流图并不是 DSSD 方法的要求,只是因为人们比较熟悉它。

从图 2.19 可以看到,销售者在接到订购货物的电话后,要记录订购信息,并建立一个订单文件。其内容包括:顾客姓名、地址、订购日期、货号、批号、品名、规格、数量、单价和总计。在为一个特定的订单指定了一个订单号后,就把它传送给发货部门,在那里根据订单文件准备装运。其它的业务活动如会计、管理等也可对其进行存取。

图 2.19 电话订购业务系统信息流图

1) 应用环境

为了确定问题的应用环境,需要围绕以下三个方面描述问题:

- 要处理的信息项有哪些?
- 谁是信息项的产生者和接受者?
- 每个信息项的产生者和接受者如何看待顾客环境中的其它信息?

DSSD 以实体图为机制回答了这些问题。实体图很像数据流图,但图中所用符号的含义却与数据流图完全不同。在实体图中,圆形框代表了信息的产生者和接受者(如人、机器、另一个系统等)。图 2.20(a)给出了系统中的 5 个信息产生者和接受者。图(b)是销售/电话订购部门业务的实体图,它从销售/电话订购的观点把销售与顾客之间的接口全部显示出来。图(c)是可收账目实体图,图(d)是顾客实体图,图(e)是顾客服务实体图。

在对各个实体图的正确性进行评审之后,可以建立一个包括全部产生者和接受者信息的复合实体图,参看图 2.21。

图 2.20 实体机制图

图 2.21 复合实体图

图中的虚线框内是订购系统业务,虚线框称为问题的边界,也叫做应用边界。在为实体图规定了应用边界之后,哪些实体落在要讨论的范围之内就一目了然了。

应用边界内的细节可以暂时隐蔽,如图 2.22 所示。穿过边界的信息都要得到电话订购业务系统的处理。

图 2.22 应用层的实体图

2) 问题功能

研究跨越边界的信息流可以弄清楚电话订购业务系统应当实现的功能。信息项穿越边界的顺序用编号标在图 2.22 中。使用类似 Warnier 的表示法, DSSD 可以把信息和施加于信息之上的加工(或称变换、功能)联系起来,形成作业线图(assembly line diagram, ALD),见图 2.23。从概念上讲,这个图起到了与数据流图相同的作用。

图 2.23 作业线图

作业线图的制作是以图 2.22 中最大编号信息流开始,直到最小编号的信息流,依相反顺序逐渐递推画出来的。每个信息流项是由前一个编号的信息流项与产生本信息流项的过程结合起来得到的。按照自左到右的方式读为:月报表(根据每月数据而得)通过接受银行收据信息和应用报表生成过程加工而得,加号(+)表示过程与信息之间的耦合。收据信息是通过银行存款信息及其相关的过程而得到的。依次向右推进,直至最小编号的订购信息为止。在这个作业线图中,每个加工过程用一个处理说明细化,该说明包括输出、动作、动作的频度及输入。然后用 Warnier-Orr 图表示每一个加工的过程细节。

3) 问题的结果

DSSD 要求对系统的输出建立书面原型, 以表明主要的系统输出和构成输出的信息项组织。有了这个原型就可利用 Warnier-Orr 图描述信息的层次结构了。其实, Warnier-Orr 图与 Warnier 图的差别是很小的, 只是在符号和格式上有一些小的变动而已。图 2. 24 (a) 和(b) 是电话订购处理系统要求输出的月报表原型和它对应的 Warnier -Orr 图。

图 2. 24 问题结果的 Warnier 图

4) 物理需求

DSSD 表示法包括实体图、作业线图以及从逻辑角度建立软件需求模型的 Warnier-Orr 图。此外, 作为需求分析的一部分工作, 还需要确定物理需求。主要有以下几项。

- (1) 性能: 包括算法的执行时间限制, 人机交互系统的响应时间范围等。
- (2) 可靠性: 过程的设计和实现方式要保证达到规定的可靠性要求。
- (3) 安全性: 信息的保密、保护要求, 这对系统的实现方式有深刻的影响。
- (4) 硬件/ 软件: 当软件要利用宿主计算机的一些特有的性能或功能特性时, 属于一种非标准方式的耦合。类似地, 对操作系统特性也要予以考虑。
- (5) 接口: 与外部数据库、设备、网络 and 通信线路的接口协议限制。

2. 4 Jackson 系统开发方法 (JSD)
——面向数据结构的分析方法之二

Jackson 方法是一种典型的面向数据结构的分析与设计方法。早期的 Jackson 方法用于小系统的设计, 称之为 Jackson 结构程序设计方法, 简称 JSP 方法。它是按输入、输出和

内部信息的数据结构进行软件设计的,即把数据结构的描述映射成程序结构描述,设计出反映数据结构的数据结构。但是,当把 JSD 方法用于大系统设计时,就会出现大量复杂的难以对付的结构冲突。因此,促使 M. J. Jackson 提出了 JSD 方法,即 Jackson 系统开发方法。JSD 方法以活动(即事件)为中心,一连串活动的顺序组合构成进程。系统模型抽象为一组以通信方式互相联系的进程。

2.4.1 进程模型

在许多情况下,从现实世界的活动抽象而形成系统模型时,时序往往是一个必须要考虑的重要因素。例如,某书店的售书方式为书店仅存有一些样书。顾客先在店堂里挑选所需要的书籍或期刊,填写一张订单交给书店。书店向出版社订货,待货到后,书店在店堂内一张书刊信息表中通知顾客,顾客看到后,按原价的九折付款并提书。这每一个活动的发生必须遵守一定的时间顺序。因此,在相应的系统模型抽象中,自然也就必须要关注这些活动发生的先后次序。

传统的数据模型,只能表示系统的静态特性,表示数据对象及它们之间的静态关系,但缺乏表示系统动态特性的机制。在数据流模型中,加工是一种数据变换,加工之间通过数据流发生联系。但数据流的“流动”并不表示时间上的先后次序。由于现实世界中的活动往往都是时序相关的,系统模型也应当能反映这种时序关系,因此, JSD 方法采用进程模型作为这类系统的抽象。

进程是依一定次序安排的一串活动。例如,在刚才列举的书店购书的例子中,顾客选书、提交订单、通知顾客、顾客付款提书等,就是一串活动,它们组成顾客进程。当然,还可以从书店活动中抽象出其它一些进程。例如服务进程:店员从上班开始,多次登记订单、发出订购单、填写信息表、开出付款单、包装已售书刊等,直到下班。另外如会计进程:重复收款活动,最后结账,下班。在书店业务活动中,顾客进程、服务进程和会计进程三者之间互相关联,有些进程活动是并行的,也有些进程活动是同步进行的:一个活动必须暂停,等待另一进程活动的发生。

JSD 的系统模型是互相通信的一组进程的集合。进程间的通信可采用三种方式:

- 1) 进程活动同步发生。
- 2) 通过数据流通道发送/接收活动发生。
- 3) 访问公用存储信息。

这样建立的 JSD 进程模型不能直接在计算机上运行,因为可能出现太多的进程和太长的进程生存期,会使系统模拟无法实现。因此,需要确认模型,预先提出一些初始条件,把系统模型控制在一定限度的范围,使之能够有效地在计算机上运行。

2.4.2 JSD 方法的步骤

Jackson 系统开发(JSD)方法是一种典型的面向数据结构的分析与设计方法。与 DSSD 方法类似, Jackson 方法把分析的重点放在构造与系统相关联的现实世界,并建立现实世界的信息域的模型上。它实际上是支持软件分析与设计的一组连续的技术步骤。JSD 方法的最终目标是生成软件的过程性描述。

使用 JSD 方法的步骤如下:

1) 实体动作分析: 从问题的简单描述中, 选出软件系统要产生和运用的实体(人、物或组织), 以及现实世界作用于实体上的动作(事件)。实体从名词中选出, 动作从动词中选出。当然, 只有与问题求解直接有关的实体和动作才能被选出作进一步的分析。

2) 实体结构分析: 把作用于实体的动作或由实体执行的动作, 按时间发生的先后次序排序, 并用一个层状的 Jackson 结构图表示。

3) 定义初始模型: 把实体和动作表示成一个过程性的模型, 定义模型与现实世界的联系。模型系统的规格说明可用系统规格说明图(SSD)表示。

4) 功能描述: 详细说明与已定义的动作相对应的功能。

5) 决定系统时间特性: 对进程调度特性进行评价和说明。

6) 实现: 设计组成系统的硬件和软件。

JSD 方法的前 3 步属于需求分析阶段, 后 3 步属于软件设计阶段。

2.4.3 实体动作分析

分析实体的动作, 需要从一段用自然语言给出的问题描述(通常是一个段落)入手。作为一个实例, 分析一个基于软件控制的大学交通车服务系统(university shuttle service, 简称 USS)的需求, 它的问题描述如下。

“某大学分布在相隔两公里多的两个校园区内。为帮助学生在两个校园区之间来回穿行, 以保证他们按时上课, 学校计划建一专用交通车服务设施。该设施利用一辆在轨道上运行的高速短程列车(以下简称列车), 在两个校园区的两个车站之间往返行驶。每个车站设有一个呼叫按钮。学生可按下按钮, 要求搭乘列车。当列车到站时, 他们 also 需按下调度按钮。若列车本来就停于此站, 等学生上车后, 列车就驶向对方站。若学生按动呼叫按钮时, 列车正在行驶中, 呼叫者需等待, 等列车停靠到对方站, 搭乘上学生(如果有)返回。若学生呼叫时, 列车正停在对方站上, 得到信号后, 列车就会赶来供搭乘。若没有学生呼叫, 则列车总是停在某一车站上等待呼叫。”

通过对以上问题描述的分析, 可从所有出现的名词中选出实体。可作为实体的名词有: “大学”、“校园”、“学生”、“上课”、“列车”、“车站”、“按钮”。其中, “校园”、“上课”、“学生”、“车站”与当前的问题并无直接的关系, 它们落在要解决问题模型的边界之外, 因此它们不被选定为实体。此外, “大学”是两个校园的总称, 把它也从实体中排除。最后, 只考虑“列车”和“按钮”。

每个动作在特定的时刻施加于实体上。可通过检查问题说明中出现的动词来选择。供考虑的动作有: “来回穿行”、“到达”、“按下”、“搭乘”、“驶离”和“等待”。因为“来回穿行”一词涉及到“学生”, 而“学生”并未被选为实体; 又因为“等待”代表了一种状态, 而不是一种动作, 所以去掉这两个词。最后只选定“到达”、“按下”和“驶离”。

应当注意的是, 当选定了实体和动作的时候, 实际上已经把要开发系统的范围划定了。例如, 把“学生”从实体中划掉了, 就把生成“今天有多少学生要乘车”之类的信息等问题排除了。当然, 随着分析工作的逐渐展开, 实体和动作的范围还可能会变动。

2.4.4 实体结构分析



在 JSD 方法中, 实体的结构通过在一段时间内的动作来描述实体的历史情况。为了表示实体结构, Jackson 引入了如图 2.25 所示的结构图。在这个图中, 给出了对实体的三种典型的动作, 即顺序的、选择的和重复的。其中, 顺序型是指对于实体 A 的动作 B, C 是按时间顺序先左后右执行; 选择型使用了符号“”, 表示实体 A 的两个动作 B 与 C 在某一时刻只做一个; 重复型表示带有“”的动作重复执行多次。

图 2.25 结构图表示法

图 2.26(a)和(b)给出了实体“ 列车 ”和“ 按钮 ”的实体结构图。

图 2.26 列车和按钮的结构图

从图 2.26(a)中可以看出, 列车的活动在 1 号站开始和结束。作用于实体的动作只有到达和驶离(均在叶结点上)。在开始时, 列车停在 1 号站, 其后的一段时间内往返于 1 号站与 2 号站之间, 最后返回 1 号站。从图 2.26 中还可以看出, 通常到达某站后, 紧跟着的动作必定是驶离该站。表示这一现象的是 arrive(i) 和 leave(i), 其中的 i 是站号。

对于那些无法用 JSD 记号表达的限制, 常需要在结构图上加注解。例如, “ i 的值只能取‘ 1 ’和‘ 2 ’, 两次连续出现车站时, i 的值必定改变。”

图 2.26(b)表示作用于实体“ 按钮 ”上的重复动作“ 按下 ”。

从这个实例可以看到, 结构图给出了实体执行的动作或施加于实体上的动作的时序关系。因此, 它对现实世界的描述比简单的实体表和动作表要准确得多。在为每一个实体构造结构图时, 可以附加一段文字注解。

2. 4. 5 定义初始模型

以上两步只是现实世界的一种抽象描述,完成的工作有:选定实体和动作,并用结构图建立它们之间的关系。这一步则要对系统构造规格说明,使其成为现实世界的模型。规格说明可用系统规格说明图(SSD)表示。这种图所使用的符号见图 2. 27。

图 2. 27 系统规格说明图(SSD)表示法

数据流连接用于图示进程之间信息流的传出/传入的关系。其中,矩形框表示发送信息流或接收信息流的进程,箭头表示信息流传递的方向,圆圈表示数据流。圆圈中的 B 是 button 的缩写,D 是 data stream 的缩写。数据流假定是存放在一个容量不受限制的先进先出 (FIFO) 缓冲区中。

状态向量连接用于指明一个进程需要直接检查另一个进程的状态向量。其中,矩形框与箭头表示的意义同数据流连接,菱形框表示相关的状态向量。框中的 SV 即 state vector 的缩写。状态向量连接在过程控制应用中用的较多。在图 2. 27 中,用后缀 0 代表现实世界的进程,用后缀 1 代表系统模型的进程。

现在再来考虑大学交通车服务系统(USS)的实例。图 2. 28 给出了 USS 的系统规格说明图。

图 2. 28 USS 的 SSD

只要有可能,总是把模型中的进程与现实世界的实体通过数据流联系起来,以确保模型行为和现实世界行为的一致性。在上面的实例中,呼叫按钮- 0 按下时,按钮- 0 进程就发送出一个脉冲。这个脉冲传送给按钮- 1 进程。这就是一个数据流连接。这里,我们应当假定检测列车到达或驶离的传感器不发送脉冲,它近似于一个电器开关的功能。开关的状态(开/ 闭)是可访问的。因而需要一个状态向量连接来检测开关的状态。

模型进程的内部细节可用结构正文(structure text)来描述。结构正文与结构图给出了相同的信息(图 2. 29), 同样包括顺序、选择、重复三类,只不过它是以正文格式表达。

按钮-1 的结构正文是:

图 2. 29 结构图表示与结构正文表示

```

BOTTON- 1 seq
    Read BD;
    PUSH-BDY iter while SD
        PUSH;
        Read BD;
    PUSH-BDY end
BOTTON- 1 end

```

BOTTON- 1 的结构与 BOTTON- 0 的结构几乎完全对应。不过, BOTTON- 1 的结构正文中增加了将现实世界连接到系统的读 (Read) 操作。

如前所述, 进程“列车- 1”(Shuttle- 1) 不能通过数据流连接与其现实世界的对应物建立联系。它只能不断地查询开关的状态, 从开关的开/ 闭来推知列车的到达和驶离。

系统进程要经常检查现实世界的实体, 确保不会遗漏所有的动作。这一点可以用操作 getsv (意为获取状态向量) 来实现。它可取得现实世界实体的状态向量。很可能系统进程在状态向量改变以前多次读取它。因此, 必须对这个模型进程进行细化, 使它能够反映状态向量“在变化中”的值。下面, 列车- 1 的结构正文可描述如下。

```

SHUTTLE- 1 seq
    getsv SV;
    WAIT- BDY iter while WAIT 1
        getsv SV;
    WAIT- BDY end
    LEAVE( 1);
    TRANSIT- BDY1 iter while TRANSIT 1
        getsv SV;
    TRANSIT- BDY1 end
    SHUTTLE- BDY1 iter
        STATION seq
            ARRIVE(i);
            WAIT- BDY iter while WAITi
                getsv SV;
            WAIT- BDY end
            LEAVE(i);
            TRANSIT- BDY iter while TRANSIT i
                getsv SV;

```

```

TRANSIT- BDY  end
STATION end
SHUTTLE- BDY1  end
ARRIVE(1);
SHUTTLE- 1  end

```

在此结构正文描述中, WAIT 和 TRANSIT 的状态值表示了到达 (ARRIVE) 和驶离 (LEAVE) 开关的值。现实世界进程“列车- 0 (Shuttle- 0)”引起了开关状态的变化, 系统进程“列车- 1 (Shuttle- 1)”执行操作 getsv 感知了这个变化。图 2.30 给出了对应于 Shuttle- 1 结构正文的结构图。

图 2.30 对应于结构正文的结构图

JSD 的下面步骤, 可自然过渡到软件设计阶段, 将放在软件设计部分介绍。

2.5 原型化方法 (prototyping)

在需求分析阶段, 要想得到一个完整准确的规格说明不是一件容易的事。特别是对于一些大型的软件项目, 在开发的早期用户往往对系统只有一个模糊的想法, 很难完全准确地表达对系统的全面要求, 软件开发者对于所要解决的应用问题认识更是模糊不清。经过详细的讨论和分析, 也许能得到一份较好的规格说明, 但却很难期望该规格说明能将系统的各个方面都描述得完整、准确、一致, 并与实际环境相符。很难通过它在逻辑上推断出

(不是在实际运行中判断评价)系统运行的效果,以此达到各方者对系统的共同理解。随着开发工作向前推进,用户可能会产生新的要求,或因环境变化,要求系统也能随之变化;开发者又可能在设计与实现的过程中遇到一些没有预料到的实际困难,需要以改变需求来解脱困境。因此规格说明难以完善、需求的变更、以及通信中的模糊和误解,都会成为软件开发顺利推进的障碍。为了解决这些问题,逐渐形成了软件系统的快速原型的概念。

2.5.1 软件原型的分类

通常,原型是指模拟某种产品的原始模型。在软件开发中,原型是软件的一个早期可运行的版本,它反映最终系统的部分重要特性。如果在获得一组基本需求说明后,通过快速分析构造出一个小型的软件系统,满足用户的基本要求。使得用户可在试用原型系统的过程中得到亲身感受和受到启发,做出反应和评价。然后开发者根据用户的意见对原型加以改进。随着不断试验、纠错、使用、评价和修改,获得新的原型版本,如此周而复始,逐步减少分析和通信中的误解,弥补不足之处,进一步确定各种需求细节,适应需求的变更,从而提高了最终产品的质量。

虽然软件原型化方法是在研究分析阶段的方法和技术中产生的,但它更针对传统方法所面临的困难。因此,也面向软件开发的其它阶段。由于软件项目的特点和运行原型的目不同,原型有两种不同的类型:

1) 废弃(throw away)型:先构造一个功能简单而且质量要求不高的模型系统,针对这个模型系统反复进行分析修改,形成比较好的设计思想,据此设计出更加完整、准确、一致、可靠的最终系统。系统构造完成后,原来的模型系统就被废弃不用。

2) 追加(add on)型:先构造一个功能简单而且质量要求不高的模型系统,作为最终系统的核心,然后通过不断地扩充修改,逐步追加新要求,最后发展成为最终系统。

采用什么形式主要取决于软件项目的特点和开发者的素质,以及支持原型开发的工具和技术。要根据实际情况的特点加以决策。

原型系统不同于最终系统,它需要快速实现,投入运行。因此,必须注意功能和性能上的取舍。可以忽略一切暂时不必关心的部分,力求原型的快速实现。但要能充分地体现原型的作用,满足评价原型的需求。要根据构造原型的目的,明确规定对原型进行考核和评价的内容,如界面形式、系统结构、功能或模拟性能等等。构造出来的原型可能是一个忽略了某些细节或功能的整体系统结构,也可以仅仅是一个局部,如用户界面、部分功能算法程序或数据库模式等。总之,在使用原型化方法进行软件开发之前,必须明确使用原型的目,从而决定分析与构造内容的取舍。

2.5.2 快速原型开发模型

原型的开发和使用过程叫做原型生存期。图 2.31(a)是原型生存期的模型,图 2.31(b)是模型的细化。

(1) 快速分析

在分析者和用户的紧密配合下,快速确定软件系统的基本要求。根据原型所要体现的特性(或界面形式、或处理功能、或总体结构、或模拟性能等),描述基本规格说明,以满足

图 2.31 原型生存期

开发原型的需要。快速分析的关键是要注意选取分析和描述的内容,围绕使用原型的目标,集中力量,确定局部的需求说明,从而尽快开始构造原型。

当系统规模很大、要求复杂、系统服务不清晰时,在需求分析阶段先开发一个系统原型是很值得的。特别是当性能要求比较高时,在系统原型上先做一些试验也是很必要的。

(2) 构造原型

在快速分析的基础上,根据基本规格说明,尽快实现一个可运行的系统。为此需要强有力的软件工具的支持,例如采用非常高级的语言实现原型,或者引入以数据库为核心的开发工具等。并忽略最终系统在某些细节上的要求,如安全性、健壮性、异常处理等。主要考虑原型系统应充分反映的待评价的特性。

提交一个初始原型所需要的时间,根据问题的规模、复杂性、完整程度的不同而不同。3~6周提交一个系统的初始原型应是可能的,最大限度不能超过两个月。两个月后提交的应是一个系统而不是一个原型。

(3) 运行和评价原型

这是频繁通信、发现问题、消除误解的重要阶段。其目的是验证原型的正确程度,进而开发新的并修改原有的需求。它必须通过所有相关人员的检查、评价和测试。由于原型忽略了许多内容,它集中反映了要评价的特性,外观看起来可能会有些残缺不全。用户要在开发者的指导下试用原型,在试用的过程中考核评价原型的特性,分析其运行结果是否满足规格说明的要求,以及规格说明的描述是否满足用户的愿望。纠正过去交互中的误解和

分析中的错误,增补新的要求,并为满足因环境变化或用户的新设想而引起系统需求的变动并提出全面的修改意见。

(4) 修正和改进

根据修改意见进行修改。若原型运行的结果未能满足规格说明中的需求,反映出对规格说明存在着不一致的理解或实现方案不够合理。若因为严重的理解错误而使正常操作的原型与用户要求相违背时,有可能会产生废品。如果发现是废品应当立即放弃,而不能再凑合。大多数原型不合适的部分可以修正,使之成为新模型的基础。如果是由于规格说明不准确(有多义性或者未反映用户要求)、不完整(有遗漏)、不一致,或者需求有所变动或增加,则首先要修改并确定规格说明,然后再重新构造或修改原型。

如果用修改原型的过程代替快速分析,就形成了原型开发的迭代过程。开发者和用户在一次次的迭代过程中不断将原型完善,以接近系统的最终要求。

在修改原型的过程中会产生各种各样的积极的或消极的影响,为了控制这些影响,应当有一个词典,用以定义应用的信息流,以及各个系统成份之间的关系。另外,在用户积极参与的情况下,保留改进前后的两个原型,一旦用户需要时可以退回,而且贯穿地演示两个可供选择的对象,有助于决策。

(5) 判定原型完成

经过修改或改进的原型,达到参与者一致认可,则原型开发的迭代过程可以结束。为此,应判断有关应用的实质是否已经掌握,迭代周期是否可以结束等。

判定的结果有两个不同的转向,一是继续迭代验证,一是进行详细说明。

(6) 判断原型细部是否说明

判断组成原型的细部是否需要严格地加以说明。原型化方法允许对系统必要成份进行严格的详细的说明。例如将需求转化为报表,给出统计数字等等。这些不能通过模型进行说明的成份,必要的话,需提供说明,并利用屏幕进行讨论和确定。

(7) 原型细部的说明

对于那些不能通过原型说明的所有项目,仍需通过文件加以说明。例如,系统的输入、输出、加工、系统的逻辑功能、数据库组织、系统的可靠性、用户地位等等。原型化对完成严格的规格说明是有帮助的。如输入、输出记录都可以通过屏幕进行统计和讨论。

严格说明的成份要作为原型化方法的模型编入词典,以得到一个统一的连贯的规格说明提供给开发过程。

(8) 判定原型效果

考察用户新加入的需求信息和细部说明信息,看其对模型效果有什么影响?是否会影响模块的有效性?如果模型效果受到影响,甚至导致模型失效,则要进行修正和改进。

(9) 整理原型和提供文档

整理原型的目的是为进一步开发提供依据。原型的初期需求模型是一个自动的文档。

总之,利用原型化技术,可为软件的开发提供一种完整的、灵活的、近似动态的规格说明方法。

2.6 系统动态分析

系统的需求规格说明通常用自然语言叙述,但是用自然语言描述所表现的方式和内容自由度太大,往往会出现歧义性。机械地验证规格说明的合理性是不合适的。为了直观地分析系统的动作,从特定的视点出发描述系统的行为,需要采用动态分析的方法。其中最为常用的动态分析方法有状态迁移图、时序图、Petri 网等。

2.6.1 状态迁移图

状态迁移图是描述系统的状态如何相应外部的信号进行推移的一种图形表示。在状态迁移图中,用圆圈“ ”表示可得到的系统状态,用箭头“ ”表示从一种状态向另一种状态的迁移。在箭头上要写上导致迁移的信号或事件的名字。如图 2.32(a)所示,系统中可取得的状态= S1, S2, S3, 事件= t1, t2, t3, t4。事件 t1 将引起系统状态 S1 向状态 S3 迁移,事件 t2 将引起系统状态 S3 向状态 S2 迁移等等。

状态迁移图表示的关系还可以用表格形式表达,我们称这样的表格为状态迁移表。图 2.32(b)是与图 2.32(a)等价的状态迁移表。表中第 i 行第 j 列的元素是一个状态,它是从现在的状态 j 因事件 i 而要移到的下一个状态。由于系统中可得到的状态是有限的,因此在根据现在的状态和输入信号(到来的事件)确定下一个状态时,状态迁移图是一个很有效的图形方法。

如何设置系统的状态,需要根据分析的目标和表达的目的而定。下面举一个在操作系统中根据调度的要求设置进程状态的例子。

图 2.32 状态迁移图与其等价的状态迁移表例

图 2.33 给出了当有多个申请占用 CPU 运行的进程时,有关 CPU 分配的进程的状态迁移。进程是分配 CPU 的最小处理单位。可得到的状态= 就绪 (ready), 运行(running), 等待(wait)。其中,“就绪”为等待分配 CPU 状态;“运行”为正在 CPU 上做处理状态;“等待”为放弃 CPU 状态。生成的事件= t1, t2, t3, t4。其中,“t1”为因要求 I/O 等而要求中断的事件;“t2”为处理中断的事件;“t3”为分配 CPU 的事件;“t4”为已用完分配的 CPU 时间的事件。

如果系统比较复杂,可以把状态迁移图分层表示。例如,在图 2.33 进程的状态迁移确定了如图 2.34 所示那样的大状态 S1, S2, S3 之后,接下来可

把状态 S1, S2, S3 细化。在该图中对状态 S1 进行了细化。

此外,在状态迁移图中,由一个状态和一个事件决定的下一状态可能会有多个。实际会迁移到哪一个是由更详细的内部状态和更详细的事件信息决定的。此时,可采用状态迁移图的一种变形,如图 2. 35 那样,使用加进判断框和处理框的记法。

图 2. 34 状态迁移图的网

图 2. 35 状态迁移图的变形




状态迁移图的优点是,第一,状态之间的关系能够直观地捕捉到,这样用眼睛能看到是否所有可能的状态迁移都已纳入图中,是否存在不必要的状态等。第二,由于状态迁移图的单纯性,能够机械地分析许多情况,可很容易地建立分析工具。例如,可以简单地做出回答诸如“指定事件序列 P、状态 A 和 B,可以因为 P 从 A 到 B 迁移吗?”,“找出经过所有状态的事件序列”等问题的工具。

2. 6. 2 Petri 网



Petri 网的思想是 1962 年由德国人 C. A. Petri 提出来的。本来是表达异步系统的控制规则的图形表示法,现在已广泛地应用于硬件与软件系统的开发中,它适用于描述与分析相互独立、协同操作的处理系统。在软件需求分析与设计阶段都可以使用。

1) 基本概念

Petri 网简称 PNG(petri net graph),是一种有向图,它有两种结点:

- 位置(place):符号为“”,它用来表示系统的状态,也可以解释为使事件工作的条件,或使之工作的要求。
- 迁移(transition):符号为“”或“”,它用来表示系统中的事件。

图中的有向边表示对迁移的输入,或由迁移的输出。

- 符号“”:表示事件发生的前提,即对迁移(事件)的输入。
- 符号“”:表示事件的结果,即由迁移(事件)的输出。

通常称迁移的启动为激发或开火(fire),它是迁移的输出。只有当作为输入的所有位置的条件都满足时才能引起激发。例如,图 2. 36(a)给出了一个简单的 PNG 的例子,它表示了一个处于静止状态的系统。图中只给出系统中各个状态通过迁移而表示出来的相互关系。通过 Petri 网的执行才能完成系统及其行为的完整模型。为了描述系统的一种动态的行为,引入了标记(token,也称为令牌)的概念。

在图 2. 36(b)中,表示位置 P3 与 P5 的圆圈中间点了一个黑点,称之为标记。标记在位置上的出现表明了处理要求的到来。例如在图 2. 36(b)中, p3 与 p5 上都出现了标记,标明这两个状态都有了处理要求,亦即迁移(事件)t3 激发的两个前提都已具备,迁移 t3 激

图 2.36 一个简单的 Petri 网

发。做为执行的结果,位置 P3 与 P5 上的标记移去,标记迁移到了位置 P4 上。

反过来,当激发产生的结果有几个时,将随机地选择一个结果输出,并把作为结果的位置的状态加上标记。标记在 PNG 中的游动,就出现了“状态的迁移”。

图 2.37 就是根据迁移的激发而得到的状态迁移。图中(b)表明有时当出现了两个以上有可能激发的迁移时,一个迁移激发了,而另一个迁移则没有激发。

图 2.37 状态迁移例

2) 简单的 Petri 网模型

下面以一个具体例子说明 Petri 网模型的建立。有一个环形铁路,在 A 站与 B 站之间是单轨,在某一时刻只能走一列火车。但 A 站与 B 站都是双向运行的。上行/下行列车交替行驶。参看图 2.38。

图 2.38 环形铁路例

用如图 2.39 所示的 PNG 描述两列火车在铁路上的运行实况。由于在 A 站和 B 站之间有一段单轨线路,在某一时刻只允许有一列火车通过,因此只有当单线上没有列车通过时,火车才能进入单线运行,否则火车只能在 A 站或者 B 站等候,等待单线空出来使用。

图 2.40 所示的例子在数据处理和实时控制领域更具普遍意义。设在一个多任务系统中有两个进程 PR1 和 PR2,它们使用了一个公共资源 R。该资源在系统运行的某一时刻只能为一个进程所占用。为了解决两个进程在运行中可能会同时申请资源的矛盾,要用原语 LOCK(对资源加锁)和 UNLOCK(对资源解锁)控制 R 的使用,保证进程间的同步。

把这两个进程的动作用 PNG 表达,如图 2.41 所示。

图 2.39 环形铁路运行的 PNG

图 2.40 进程同步的机制

图 2.41 进程同步机制的 PNG

3) 可达树

设系统有 n 个位置, 每个位置因标记的有无, 可有“1”与“0”两种状态。这样整个系统的状态可以用各位置上的标记符号“1”, “0”来表示。现在把系统在某一时刻的状态用 n 元组 (M_1, M_2, \dots, M_n) 表示, 其中, M_i 是第 i 个位置中标记符号, 有“1”与“0”两种取值。

如图 2.41 所示例子, 当前系统的状态是 $(0, 1, 0, 1, 0, 0, 0)$, 从这个状态开始, 随着状态的激发, 状态向前推移。如图 2.42 所示。

其中, 用虚线连接的两个状态是相同的状态。这样, 状态推移出现了往复循环。我们称这种有限的图为可达树。

利用这种可达树, 可以检查可达性。若给出某一状态 v , 那么从初始状态 v_0 到状态 v ,

图 2.42 进程同步机制 PNG 的可达树

有没有迁移激发(状态推移)序列,就可以用可达树检查。若树中出现了叶结点,则系统中出现了死锁,状态推移到此,就无法再推移下去。此外,利用可达树还可以检查系统的有界性、安全性、覆盖性、等价性等等。

4) 为了建立 Petri 网,并使用 Petri 网。可采用如下办法。

确定系统的独立成份(位置);

确定这些成份动作的前提和结果(迁移);

根据条件,确定成份之间的相互关系,建立 Petri 网;

通过建立可达树,对 Petri 网进行分析,确定系统特性;

根据需要重复上述 ~ 步,修改模型,直到满意为止。

2.7 结构化分析与设计技术(SADT)

SADT(Softtech 公司的商标)是 D.T. Ross 等人 1977 年提出来的一种结构化分析与设计的技术,已广泛地应用于系统定义、软件需求分析、系统设计与软件设计。最初,SADT 是作为一种手工方法而被开发的。它由以下几方面组成:

分解软件(或系统)功能的过程;

能沟通软件内部的功能和信息联系的图解表示,即 SADT 活动图和数据图;

使用 SADT 进行项目管理的指导书。

使用 SADT,可以建立一个由许多分层定义的活动图(actgraph)和数据图(data-graph)组成的模型,叫做 SA(structured analysis)图。这种表示的格式如图 2.43 所示。在该图中(a)和(b)的左图给出的是活动图和数据图的一般表示,而其右图分别给出它们的实例。

在活动图中,结点表示活动,弧表示活动之间的数据流。因此,活动图是数据流图的另一种形式。但要注意不可与数据流图混淆。活动图有四种不同的数据流与每一个结点有关。如图 2.43(a)所示,一个结点的输出可以作为另一个结点的输入或控制,而某些结点的输出则是整个系统对外界环境的输出。这样,每个结点的输出必须联到其它结点或外部

环境。它的输入与控制也必须来自其它结点的输出或者来自外部环境。

图 2. 43 活动图和数据图的组成

在数据图中, 结点表示数据对象, 弧表示对于数据对象的活动。其中, 输入是生成数据对象的活动, 输出是使用数据对象的活动, 而控制是对结点数据对象使用的一种控制。如图 2. 43(b) 所示。因而数据图与活动图是对偶的。在实践中, 活动图应用得更广泛。不过, 数据图由于以下两个特点, 也是很有用的。

它能指明一个给定数据对象所影响的所有活动;

可利用由活动图构造数据图的办法检查某个 SADT 模型的完全性和一致性。

图 2. 44 是 SA 图的结构特性。在其(a)图中, I1 是外部输入, C1 是外部控制, O1 和 O2 是最后的输出, m 是活动 A1 的处理机构。在 SA 图中每个方框还可以进一步细化, 其方式与数据流图经历的细化方式完全相同。图(b)就是图(a)的细化, 图中的 A11, A12, A13, A14 是 4 个活动。各活动之间的关系如图所示。m1, m2, m3, m4 是实现各活动的处理机构。这个活动图对应于其上一层(图(a))的一项活动 A1。

图 2. 44 SA 图的结构特性

描述软件工程定义阶段最初几个步骤的 SADT 活动图例示于图 2. 45 中。

总之, SADT 以一种清晰、精确的方式提供了理解和表达复杂需求的方法。它的主要特点是:

- 自顶向下把高层的结点分解成为若干个下属图;
- 对每个结点分清它们的输入、输出;
- 控制和机构, 活动图和数据图的对偶性;

图 2.45 描述需求定义的 SADT 活动图

- 为开发、审查和调整 SADT 模型所需要的管理技术。

SADT 也可应用到其它类型的系统中,并不只限于软件开发。它适合于大型、复杂项目的分析。

第 3 章 软件 设计

对于任何工程项目来说,在它施工之前,总要先完成设计。因此,设计往往是开发活动的第一步。通常,人们把设计定义为“应用各种技术和原理,对设备、过程或系统作出足够详细的定义,使之能够在物理上得以实现”。

软件的设计与其它领域的工程设计一样,也需要有好的方法、好的分析策略等等。把软件设计看作仅仅是程序设计或者编制程序,这是很片面的。实际上,程序设计只是软件设计的实现,不能把它们混同起来。

3.1 软件设计的目标和任务

一旦软件需求确定之后,就进入开发阶段。开发阶段由三个互相关联的步骤组成:设计、编码和测试。每个步骤都按某种方式进行信息变换,最后得到有效的计算机软件。

3.1.1 软件设计在开发阶段中的重要性

在软件需求分析阶段已经完全弄清楚了软件的各种需求,较好地解决了要让所开发的软件“做什么”的问题,并已在软件需求说明书和数据要求说明书中详尽和充分地阐明了这些需求以后,下一步要着手实现软件的需求,即要着手解决“怎么做”的问题。

开发阶段的信息流如图 3.1 所示。在设计步骤中,根据软件的功能和性能需求等,采用某种设计方法进行数据设计、系统结构设计和过程设计。

图 3.1 开发阶段的信息流

数据设计侧重于数据结构的定义。系统结构设计定义软件系统各主要成份之间的关

系。过程设计则是把结构成份转换成软件的过程性描述。在编码步骤中, 根据这种过程性描述, 生成源程序代码, 然后通过测试最终得到完整有效的软件。

3. 1. 2 软件设计任务

软件设计是一个把软件需求变换成软件表示的过程。最初这种表示只是描绘出软件的总的框架, 然后进一步细化, 在此框架中填入细节, 把它加工成在程序细节上非常接近于源程序的软件表示。

从工程管理的角度来看, 软件设计分两步完成。首先作概要设计, 将软件需求转化为数据结构和软件的系统结构。然后是详细设计, 即过程设计。通过对结构表示进行细化, 得到软件的详细的数据结构和算法。

在概要设计过程中需要完成的工作具体地讲, 有以下几个方面:

1) 制定规范

在进入软件开发阶段之初, 首先应为软件开发组制定在设计时应该共同遵守的标准, 以便协调组内各成员的工作。它包括:

- 阅读和理解软件需求说明书, 在给定预算范围内和技术现状下, 确认用户的要求能否实现。从而确定设计的目标, 以及它们的优先顺序。
- 根据目标确定最合适的设计方法。
- 规定设计文档的编制标准, 包括文档体系、用纸及样式、记述的详细程度、图形的画法等。
- 规定编码的信息形式(代码体系)、与硬件/ 操作系统的接口规约、命名规则等。

2) 软件系统结构的总体设计

在需求分析阶段, 已经从系统开发的角度出发, 把系统按功能逐次分割成层次结构, 使每一部分完成简单的功能且各个部分之间又保持一定的联系, 这就是所谓的功能设计。在设计阶段, 基于这个功能的层次结构把各个部分组合起来成为系统。它包括:

- 采用某种设计方法, 将一个复杂的系统按功能划分成模块的层次结构。
- 确定每个模块的功能, 建立与已确定的软件需求的对应关系。
- 确定模块间的调用关系。
- 确定模块间的接口, 即模块间传递的信息。设计接口的信息结构。
- 评估模块划分的质量及导出模块结构的规则。

3) 处理方式设计

- (1) 确定为实现软件系统的功能需求所必需的算法, 评估算法的性能。
- (2) 确定为满足软件系统的性能需求所必需的算法和模块间的控制方式(性能设计)。性能主要是指周转时间、响应时间、吞吐量、精度。
- (3) 确定外部信号的接收发送形式。

4) 数据结构设计

确定软件涉及的文件系统的结构以及数据库的模式、子模式, 进行数据完整性和安全性的设计。它包括:

- (1) 确定输入, 输出文件的详细的数据结构。

(2) 结合算法设计, 确定算法所必需的逻辑数据结构及其操作。

(3) 确定对逻辑数据结构所必需的那些操作的程序模块(软件包)。限制和确定各个数据设计决策的影响范围。

(4) 若需要与操作系统或调度程序接口所必需的控制表等数据时, 确定其详细的数据结构和使用规则。

(5) 数据的保护性设计

- 防卫性设计: 在软件设计中插入自动检错、报错和纠错的功能。
- 一致性设计: 有两个方面, 其一是保证软件运行过程中使用的数据的类型和取值范围不变。其二是在并发处理过程中使用封锁和解除封锁机制保持数据不被破坏。
- 冗余性设计: 针对同一问题, 由两个开发者采用不同的程序设计风格、不同的算法设计软件, 当两者运行结果之差不在允许范围内时, 利用检错系统予以纠正, 或使用表决技术决定一个正确的结果, 以保证软件容错。

5) 可靠性设计

可靠性设计也叫质量设计。在软件开发的一开始就要确定软件可靠性和其它质量指标, 考虑相应措施, 以使得软件易于修改和易于维护。

6) 编写概要设计阶段的文档

概要设计阶段完成时应编写以下文档:

- 概要设计说明书。给出系统目标、总体设计、数据设计、处理方式设计、运行设计、出错设计等。
- 数据库设计说明书。给出所使用的数据库简介、数据模式设计、物理设计等。
- 用户手册。对需求分析阶段编写的初步的用户手册进行审定。
- 制定初步的测试计划。对测试的策略、方法和步骤提出明确的要求。

7) 概要设计评审

在以上几项工作完成之后, 应当组织对概要设计工作的评审。评审的内容包括:

- 可追溯性: 即分析该软件的系统结构、子系统结构, 确认该软件设计是否覆盖了所有已确定的软件需求, 软件每一成份是否可追溯到某一项需求。
- 接口: 即分析软件各部分间的联系, 确认该软件的内部与外部接口是否已经明确定义。模块是否满足高内聚和低耦合的要求。模块作用范围是否在其控制范围之内。
- 风险: 即确认该软件设计在现有技术条件下和预算范围内是否能按时实现。
- 实用性: 即确认该软件设计对于需求的解决方案是否实用。
- 技术清晰度: 即确认该软件设计是否以一种易于翻译成代码的形式表达。
- 可维护性: 从软件维护的角度出发, 确认该软件设计是否考虑了方便未来的维护。
- 质量: 即确认该软件设计是否表现出良好的质量特征。
- 各种选择方案: 看是否考虑过其它方案, 比较各种选择方案的标准是什么。
- 限制: 评估对该软件的限制是否现实, 是否与需求一致。
- 其它具体问题: 对于文档、可测试性、设计过程等进行评估。

需要特别指出的是, 软件系统的一些外部特性的设计, 例如软件的功能、一部分性能、以及用户的使用特性等, 在软件需求分析阶段就已经开始。这些问题的解决, 多少带有一些“ 怎么做 ”的性质, 因此有人称之为软件的外部设计。

在详细设计过程中, 需要完成的工作是:

- 1) 确定软件各个组成部分内的算法以及各部分的内部数据组织。
- 2) 选定某种过程的表达形式来描述各种算法。
- 3) 进行详细设计的评审

软件设计的最终目标是要取得最佳方案。所谓“ 最佳 ”, 是指在所有候选方案中, 节省开发费用、降低资源消耗、缩短开发时间的条件、选择能够赢得较高的生产效率、较高的可靠性和可维护性的方案。在整个设计的过程中, 各个时期的设计结果需要经过一系列的设计质量的评审, 以便及时发现和及时解决在软件设计中出现的问题, 防止把问题遗留到开发的后期阶段, 造成后患。在评审以后, 必须针对评审中发现的问题, 对设计的结果进行必要的修改。为了评价设计质量的高低, 必须有一个标准。在后面各节中, 将围绕设计质量的标准进行详细讨论, 图 3. 2 给出了软件设计的流程。

图 3. 2 软件设计的流程

3. 2 程序结构与程序结构图

程序结构表明了程序各个部件(模块) 的组织情况, 它通常是树状结构或网状结构, 并蕴含了在程序控制上的层次关系。但要注意的是, 程序结构是软件的过程表示, 但并未表明软件的某些过程性特征。比如, 软件的动态特性, 在程序结构中就未明确体现。

3.2.1 程序的树状结构和网状结构

由模块连接而得到的程序结构最普通的形式就是树状结构和网状结构。在树状结构中,位于最上层的根部是顶层模块,它是程序的主模块。与其联系的有若干下属模块,各下属模块还可以进一步引出更下一层的下属模块。如图 3.3(a)所示的树状结构可以看出模块的层次关系。模块 A 是顶层模块,如果算作第 0 层,则其下属模块 B 和 C 为第 1 层,模块 D、E 和 F 是第 2 层等等。

图 3.3 程序的树状结构和网状结构

从图 3.3(a)中可知,树状结构的特点是:整个结构只有一个顶层模块,而对于任何一个下属模块来说,它只有一个上级模块,而且同一层模块之间不发生联系。

网状结构的情况则完全不同。在网状结构中,任意两个模块间都可以有双向的关系。由于不存在上级模块和下属模块的关系,也就分不出层次来。任何两个模块都是平等的,没有从属关系。图 3.3(b)和图 3.3(c)给出了网状结构的两个例子。

分析两种结构的特点之后可以看出,对于不加限制的网状结构,由于模块间相互关系的任意性,使得整个结构十分复杂,处理起来势必引起许多麻烦,这与原来划分模块为便于处理的意图相矛盾。所以在软件开发的实践中,人们通常采用树状结构,而不采用网状结构。

3.2.2 结构图 (structure chart, 简称 SC)

结构图是精确表达程序结构的图形表示方法。它作为软件文档的一部分,清楚地反映出程序中模块之间的层次调用关系和联系:它不仅严格地定义了各个模块的名字、功能和接口,而且还集中地反映了设计思想。换句话说,它以特定的符号表示模块、模块间的调用关系和模块间信息的传递。结构图的主要内容有:

1) 模块:在结构图中,模块用矩形框表示,并用模块的名字标记它。模块的名字应当能够表明该模块的功能。对于现成的模块,则以双纵边矩形框表示。见图 3.4。

2) 模块的调用关系和接口:在结构图中,两个模块之间用单向箭头联结。箭头从调用模块指向被调用模块,表示调用模块调用了被调用模块。但其中隐含了一层意思,就是被调用模块执行完成之后,控制又返回到调用模块。图 3.5(a)表示模块 A 调用了模块 B。

3) 模块间的信息传递:当一个模块调用另一个模块时,调用模块把数据或控制信息

图 3.4 模块的表示

图 3.5 模块间的调用关系和接口表示

传送给被调用模块, 以使被调用模块能够运行。而被调用模块在执行过程中又把它产生的数据或控制信息回送给调用模块。为了表示在模块之间传递的数据或控制信息, 在联结模块的箭头旁边给出短箭头, 并且用尾端带有空心圆的短箭头表示数据信息, 用尾端带有实心圆的短箭头表示控制信息。通常在短箭头附近应注有信息的名字如图 3.5(b) 所示。

有的结构图对这两种信息不加以区别, 一律用注有信息名的短箭头“ ”来表示。

4) 两个辅助符号: 当模块 A 有条件地调用另一个模块 B 时, 在模块 A 的箭头尾部标以一个菱形符号, 当一个模块 A 反复地调用模块 C 和模块 D 时, 在调用箭头尾部则标以一个弧形符号, 参看图 3.6。在结构图中这种条件调用所依赖的条件和循环调用所依赖的循环控制条件通常都无需注明。

5) 结构图的形态特征

图 3.7 是一个结构图的示例。它是一个软件系统的分层模块结构图。在图中, 上级模块调用下级模块。它们之间存在主从关系, 即自上而下“ 主宰 ”, 自下而上“ 从属 ”。而同一层的模块之间并没有这种主从关系。

- 一个模块如果调用了多个下属模块, 这些下属模块在结构图中所处的左右位置是无关紧要的。例如, 在图 3.8(a), 图 3.8(b) 和图 3.8(c) 都是等价的, 它们代表着同一个程序结构。但如果对下属模块的调用次序不是任意的, 例如, 必须按 A, B, C 的次序调用下属模块, 那么, 最好是采用图 3.8(a) 的形式, 因为人们习惯于从左向右读图。
- 结构图的深度: 在多层次的结构图中, 其模块结构的层次数称为结构图的深度。如在图 3.7 中, 结构图的深度为 5。结构图的深度反映了程序结构的规模和复杂程度。对于中等规模的程序, 其结构图的深度约为 10 左右。对于一个大型程序, 其深度可以有几十层。
- 结构图的宽度: 结构图中同一层模块的最大模块数称为结构图的宽度。图 3.7 中

图 3.6 条件调用和循环调用的表示

图 3.7 结构图示例

图 3.8 同一结构图的几种画法

结构图的宽度为 7。

- 模块的扇入和扇出: 扇出表示一个模块直接调用(或控制)的其它模块数目。扇入则定义为调用(或控制)一个给定模块的模块个数。多扇出意味着需要控制和协调许多下属模块。而多扇入的模块通常是公用模块。图 3.7 中模块 M 的扇出为 3, 模块 T 的扇入为 4。

3.3 模块的独立性

目前, 模块化方法已为所有工程领域所接受。模块化设计带来了许多好处。一方面, 模块化设计降低了系统的复杂性, 使得系统容易修改; 另一方面, 推动了系统各个部分的并行开发, 从而提高了软件的生产效率。

3.3.1 模块(module)

工程上许多大的系统都是由一些较小的单元组成。例如建筑工程中的砖瓦和构件, 机器中的各种零部件等。这样做的优点是便于加工制造, 便于维修。而且有些零部件或构件可以标准化, 为多个系统所共用, 这样也较节省。同样, 一个大而复杂的软件系统, 也可以根据其功能, 划分成许多较小的单元, 或较小的程序, 这些较小的单元就是模块。

“模块”, 又称“构件”, 一般指用一个名字可调用的一段程序。类似于高级语言中的 procedure(过程)、function(函数)、subroutine(子程序)、section(节)、block(块)等。它一

般具有如下三个基本属性:

- 1) 功能: 即指该模块实现什么功能, 做什么事情。必须注意的是, 这里所说的模块功能, 应是该模块本身的功能加上它所调用的所有子模块的功能。
- 2) 逻辑: 即描述模块内部怎么做。
- 3) 状态: 即该模块使用时的环境和条件。

在描述一个模块时, 必须按模块的外部特性与内部特性分别进行描述。模块的外部特性, 是指模块的模块名、参数表、其中的输入参数和输出参数, 以及给程序以至整个系统造成的影响。而模块名是外部环境调用时用以标识该模块的名字, 此名字应当反映该模块的功能。而模块的内部特性则是指完成其功能的程序代码和仅供该模块内部使用的数据。

对于调用这个模块的上级模块来说, 只需要了解这个模块的外部特性就足够了, 不必了解它的内部特性。软件设计阶段先确定模块的外部特性, 然后再确定它的内部特性。前者是软件概要设计的任务, 后者是详细设计的任务。

3.3.2 模块独立性 (module independence)

所谓模块的独立性, 是指软件系统中每个模块只涉及软件要求的具体的子功能, 而和软件系统中其它的模块的接口是简单的。例如, 若一个模块只具有单一的功能且与其它模块没有太多的联系, 那么, 我们称此模块具有模块独立性。

一般采用两个准则度量模块独立性。即模块间的耦合性和模块的内聚性。耦合性是模块之间互相连接的紧密程度的度量。模块之间的连接越紧密, 联系越多, 耦合性就越高, 而其模块独立性就越弱。内聚性是一个模块内部各个元素彼此结合的紧密程度的度量。一个模块内部各个元素之间的联系越紧密, 则它的内聚性就越高, 相对地, 它与其它模块之间的耦合性就会减低, 而模块独立性就越强。因此, 模块独立性比较强的模块应是高内聚低耦合的模块。

3.3.3 耦合性 (coupling)

耦合性是程序结构中各个模块之间相互关联的度量。它取决于各个模块之间接口的复杂程度、调用模块的方式以及哪些信息通过接口。一般模块之间可能的连接方式有七种, 构成耦合性的七种类型。它们之间的关系为

1) 非直接耦合 (nondirect coupling)

如果两个模块之间没有直接关系, 它们之间的联系完全是通过主模块的控制和调用实现的, 这就是非直接耦合。这种耦合的模块独立性最强。

2) 数据耦合 (data coupling)

如果一个模块访问另一个模块时, 彼此之间是通过数据参数(不是控制参数、公共数

据结构或外部变量)交换输入、输出信息的,则称这种耦合为数据耦合。由于限制了只通过参数表传递数据,所以按数据耦合开发的程序界面简单、安全可靠。因此,数据耦合是松散的耦合,模块之间的独立性比较强。在软件程序结构中至少必须有这类耦合。

3) 标记耦合 (stamp coupling)

如果一组模块通过参数表传递记录信息,就是标记耦合。事实上,这组模块共享了这个记录,它是某一数据结构的子结构,而不是简单变量。这要求这些模块都必须清楚该记录的结构,并按结构要求对此记录进行操作。在设计中应尽量避免这种耦合,如果我们把在数据结构上的操作全部集中在一个模块中,就可以消除这种耦合。

4) 控制耦合 (control coupling)

如果一个模块通过传送开关、标志、名字等控制信息,明显地控制选择另一模块的功能,这就是控制耦合。如图 3.9 所示。这种耦合的实质是在单一接口上选择多功能模块中的某项功能。因此,对被控制模块的任何修改,都会影响控制模块。另外,控制耦合也意味着控制模块必须知道被控制模块内部的一些逻辑关系,这些都会降低模块的独立性。

5) 外部耦合 (external coupling)

一组模块都访问同一全局简单变量而不是同一全局数据结构,而且不是通过参数表传递该全局变量的信息,则称之为外部耦合。

图 3.9 控制耦合

6) 公共耦合 (common coupling)

若一组模块都访问同一个公共数据环境,则它们之间的耦合称为公共耦合。公共的数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。

这种耦合会引起下列问题:

- 所有公共耦合模块都与某一个公共数据环境内部各项的物理安排有关,若某个数据的大小被修改,将会影响到所有的模块。
- 无法控制各个模块对公共数据的存取,严重影响软件模块的可靠性和适应性。
- 公共数据名的使用,明显降低了程序的可读性。

公共耦合的复杂程度随耦合模块的个数增加而显著增加。因此,只有在模块之间共享的数据很多,且通过参数表传递不方便时,才使用公共耦合;否则,还是使用模块独立性比较高的数据耦合好些。

7) 内容耦合 (content coupling)

如果发生下列情形,两个模块之间就发生了内容耦合;参看图 3.10。

- 一个模块直接访问另一个模块的内部数据;
- 一个模块不通过正常入口转到另一模块内部;
- 两个模块有一部分程序代码重迭 (只可能出现在汇编语言中);
- 一个模块有多个入口。

在内容耦合的情形,被访问模块的任何变更,或者用不同的编译器对它再编译,都会造成程序出错。好在大多数高级程序设计语言(除基本 BASIC 和 FORTRAN66 以下的版

图 3.10 内容耦合

本外)已经设计成不允许出现内容耦合。它一般出现在汇编语言程序中。这种耦合是模块独立性最弱的耦合。

以上由 Myers 给出的七种耦合类型, 只是从耦合的机制上所作的分类, 它给设计人员在设计程序结构时提供了一个决策准则。实际上, 开始时两个模块之间的耦合不只是一种类型, 而是多种类型的混合。这就要求设计人员按照 Myers 提出的方法进行分析, 比较和分析, 逐步加以改进, 以提高模块的独立性。

原则上讲, 模块化设计的最终目标, 是希望建立模块间耦合尽可能松散的系统。在这样一个系统中, 设计、编码、测试和维护其中任何一个模块, 就不需要对系统中其它模块有很多的了解。此外, 由于模块间联系简单, 发生在某一处的错误传播到整个系统的可能性很小。因此, 模块间的耦合情况在很大程度上影响到系统的可维护性。

3.3.4 内聚性 (cohesion)

一个内聚程度高的模块应当只完成软件过程中的一个单一的任务, 而不与程序的其它部分的过程发生联系。也就是说, 一个内聚性高的模块(在理想情况下)应当只做一件事。一般模块的内聚性分为七种类型, 它们的关系如图 3.11 所示。

图 3.11

在上面的关系中可以看到, 位于高端的几种内聚类型最好, 位于中段的几种内聚类型是可以接受的, 但位于低端的内聚类型很不好, 一般不能使用。模块的内聚性在系统的模块化设计中是一个关键的因素。

内聚性和耦合性是相互关联的。在程序结构中各模块的内聚程度越高, 模块间的耦合程度就越低。但这也不是绝对的。软件概要设计的目标是力求增加模块的内聚性, 尽量减少模块间的耦合性, 但增加内聚性比减少耦合性更重要, 应当把更多的注意力集中到提高模块的内聚程度上来。

下面分别对这几种内聚类型加以说明。

1) 功能内聚 (functional cohesion)

一个模块中各个部分都是完成某一具体功能必不可少的组成部分,或者说该模块中所有部分都是为了完成一项具体功能而协同工作、紧密联系、不可分割的。称该模块为功能内聚模块。功能内聚模块的优点是它们的功能明确,模块间的耦合简单。但是,如果把一个功能分成两个模块来解决,就会导致模块之间的很强的耦合。

2) 信息内聚 (informational cohesion)

这种模块完成多个功能,各个功能都在同一数据结构上操作,每一项功能有一个唯一的入口点。例如,图 3.12 所示的模块具有 4 个功能。这个模块将根据不同的要求,确定该执行哪一个功能。由于这个模块的所有功能都是基于同一个数据结构(符号表),因此,它是一个信息内聚的模块。

信息内聚模块可以看成是多个功能内聚模块的组合,并且达到信息的隐蔽。即把某个数据结构、资源或设备隐蔽在一个模块内,不为别的模块所知晓。这类模块的优点是,当把程序某些方面细节隐藏在一个模块中时,各个模块的独立性就增加了。

图 3.12 信息内聚模块

图 3.13 通信内聚模块

3) 通信内聚 (communicational cohesion)

如果一个模块内各功能部分都使用了相同的输入数据,或产生了相同的输出数据,则称之为通信内聚模块。通常,通信内聚模块是通过数据流图来定义的,图 3.13 所示。在图中,以虚线方框表示两个通信内聚模块。它们或者有相同的输入记录,或者有相同的输出结果。

通信内聚模块的内聚程度比过程内聚模块的内聚程度要高,因为在通信内聚模块中包括了许多独立的功能。但是,由于模块中各功能部分使用了相同的输入/输出缓冲区,因而降低了整个系统的效率。

4) 过程内聚 (procedural cohesion)

如果一个模块内的处理是相关的,而且必须以特定次序执行,则称这个模块为过程内聚模块。使用流程图作为工具设计程序的时候,常常通过流程图确定模块划分。把流程图中的某一部分划出组成模块,得到过程内聚模块。例如,把流程图中的循环部分、判定部分、计算部分分成三个模块,这三个模块都是过程内聚模块。这类模块的内聚程度比时间内聚模块的内聚程度更强一些。另外,因为过程内聚模块仅包括完整功能的一部分,所以

它的内聚程度仍然比较低, 模块间的耦合程度还比较高。

5) 时间内聚 (classical cohesion)

时间内聚又称为经典内聚。这种模块大多为多功能模块, 但模块的各个功能的执行与时间有关, 通常要求所有功能必须在同一时间段内执行。例如初始化模块和终止模块。初始化模块要为所有变量赋初值, 对所有介质上的文件置初态, 初始化寄存器和栈等, 因此要求在程序开始执行的最初一段时间内, 模块中所有功能全部执行一遍。

因为时间内聚模块中所有各部分都要在同一时间段内执行, 而且在一般情形下, 各部分可以以任意的顺序执行, 所以它的内部逻辑更简单, 存在的开关(或判定)转移更少。但是, 时间内聚模块把很多功能、任务组合在一起, 这给维护与修改造成了困难。

6) 逻辑内聚 (logical cohesion)

这种模块把几种相关的功能组合在一起, 每次被调用时, 由传送给模块的判定参数来确定该模块应执行哪一种功能。例如, 根据输入的控制信息, 或从文件中读入一个记录, 或向文件写出一个记录, 如图 3. 14 所示。这种模块是单入口多功能模块。类似的有错误处理模块。它接收出错信号, 对不同类型的错误打印出不同的出错信息。

逻辑内聚模块表明了各部分之间在功能上的相关关系。但是它所执行的不是一种功能, 而是执行若干功能中的一种, 因此它不易修改。另外, 当调用时需要进行控制参数的传递, 这就增加了模块间的耦合程度。

7) 巧合内聚 (coincidental cohesion)

巧合内聚又称为偶然内聚。当模块内各部分之间没有联系, 或者即使有联系, 这种联系也很松散, 则称这种模块为巧合内聚模块, 它是内聚程度最低的模块。例如, 一些没有任何联系的语句可能在许多模块中重复多次, 程序员为了节省存储, 把它们抽出来组成一个新的模块, 这个模块就是巧合内聚模块。如图 3. 15 所示, 模块 A, B 和 C 中都包括 3 个同样的语句, 在功能上并未给予独立的含义。为节省空间, 把它们抽出来组成一个模块 M, 在该模块中语句间没有任何联系, 模块 M 属于巧合内聚模块。

图 3. 14 逻辑内聚

图 3. 15 巧合内聚

这种模块的缺点首先是不易修改和维护。例如, 如果模块 B 由于应用上的需要, 必须将其 READ 语句改成 READ TRANSACTION FILE(读下一事务文件记录)。但模块 A 与模块 C 又不允许改, 这样可能会陷入困境。其次是这种模块的内容不易理解, 很难描述

它所完成的功能,增加了程序的模糊性。另外,可能会把一个完整的程序段分割到许多模块内,在程序运行过程中将会频繁地互相调用和访问数据。因此,在通常情况下应避免构造这种模块,除非系统受到存储空间的限制。

3.3.5 信息隐蔽

有的开发人员会问:“如何分解一个软件才能得到最佳的模块组合呢?”为了明确怎样去做,需要了解什么是“信息隐蔽”。

由 parnas 方法提倡的信息隐蔽是指,每个模块的实现细节对于其它模块来说是隐蔽的。也就是说,模块中所包含的信息(包括数据和过程)不允许其它不需要这些信息的模块使用。

通常有效的模块化可以通过定义一组相互独立的模块来实现,这些模块相互间的通信仅仅使用对于实现软件功能来说是必要的信息。通过抽象,帮助我们确定组成软件的过程(或信息)实体,而通过信息隐蔽,则可定义和实施对模块的过程细节和局部数据结构的存取限制。

由于一个软件系统在整个软件生存期内要经过多次修改,所以在划分模块时要采取措施,使得大多数过程和数据对软件的其它部分是隐蔽的。这样,在将来修改软件时偶然引入错误所造成的影响就可以局限在一个或几个模块内部,不致波及到软件的其它部分。

一个对象的抽象数据类型,就是信息隐蔽的示例。例如,对于栈 stack,可以定义它的操作 makenull(置空栈)、push(进栈)、pop(退栈)、gettop(取栈顶)和 empty(判栈空)。这些操作所依赖的数据结构是什么样的?它们是如何实现的?都被封装在其实现模块中。软件的其它部分可以直接使用这些操作,不必关心它的实现细节。一旦实现栈 stack 的模块里内部过程或局部数据结构发生改变,只要它相关操作的调用形式不变,则软件中其它所有使用这个栈 stack 的部分都可以不修改。这样的模块结构具有很强的可移植性,在移植的过程中,修改的工作量很小,发生错误的可能性也小。

3.4 结构化设计方法 ——面向数据流的设计方法

设计是一个将信息需求转换成数据结构、程序结构和过程性表示的多步骤过程。从系统设计的角度出发,软件设计方法可以分为三大类。一类是根据系统的数据流进行设计,一类是根据系统的数据结构进行设计。前者称为面向数据流的设计(data flow-oriented design)或者过程驱动的设计(process-driven design),后者称为面向数据结构的设计(data structure-oriented design)或者数据驱动的设计(data-driven design)。第三类设计方法即面向对象的设计(object-oriented design)。

结构化设计方法是由 L. Constantine, E. yourdon 等提出来的,它是基于模块化、自顶向下细化、结构化程序设计等程序设计技术基础上发展起来的。该方法实施的要点是:

- 首先研究、分析数据流图。从软件的需求规格说明中弄清数据流加工的过程。
- 然后根据数据流图决定问题的类型。数据处理问题中典型的类型有两种:变换型

和事务型。针对两种不同的类型分别进行分析处理。

由数据流图推导出系统的初始结构图。

利用启发性原则来改进系统的初始结构图,直到得到符合要求的结构图为止。

修改和补充数据词典。

制定测试计划。

由于该方法的工作与软件需求分析阶段的结构化分析方法相衔接,所以该方法是面向数据流的设计方法。

3.4.1 典型的系统结构形式

结构化设计可以很方便地将用数据流图表示的信息转换成程序结构的设计描述。在讨论如何转换之前,先讨论典型的系统结构类型。

1) 在系统结构图中的模块

在系统结构图中,称不能再分解的底层模块为原子模块。如果一个软件系统,它的全部实际加工(即数据计算或处理)都由底层的原子模块来完成,而其它所有非原子模块仅仅执行控制或协调功能,这样的系统就是完全因子分解的系统。如果系统结构图是完全因子分解的,就是最好的系统。但实际上,这只是我们力图达到的目标,大多数系统做不到完全因子分解。

一般地,在系统结构图中有 4 种类型的模块:

(1) 传入模块——从下属模块取得数据,经过某些处理,再将其结果传送给上级模块。见图 3.16(a)。它传送的数据流叫逻辑输入数据流。

(2) 传出模块——从上级模块获得数据,进行某些处理,再将其结果传送给下属模块。见图 3.16(b)。它传送的数据流叫作逻辑输出数据流。

(3) 变换模块 k_k 也叫加工模块。它从上级模块取得数据,进行特定的处理,转换成其它形式,再传送回上级模块。见图 3.16(c)。它加工的数据流叫作变换数据流。大多数计算模块(原子模块)属于这一类。

图 3.16 系统结构图的四种模块类型

(4) 协调模块——对所有下属模块进行协调和管理的模块,见图 3.16(d)。在系统的输入/输出部分或数据加工部分可以找到这样的模块。在一个好的系统结构图中,协调模块应在较高层出现。

在实际系统中有些模块属于上述某一类型,有些模块是上述各种类型的组合。

2) 变换型系统结构图——典型的系统结构形式之一

变换型数据处理问题的过程大致分为三步,即取得数据,变换数据和给出数据。参看图 3.17。这三步反映了变换型问题数据流图的基本思想,或者说是这类问题数据流

图概括而抽象的模式。其中,变换数据是数据处理过程的核心工作,而取得数据只不过是为其做准备,给出数据则是对变换后的数据进行后处理工作。

图 3.17 变换型数据流图

变换型系统结构图如图 3.18 所示,相应于取得数据、变换数据、给出数据,系统的结构图由输入、中心变换和输出等三部分组成。

在图 3.18 中,顶层模块(图中的)首先得到控制,沿着结构图的左支依次调用其下属模块,直至底层读入数据 A。然后,对 A 进行预加工(图中的),转换成 B 向上回送。再继续对 B 进行加工(图中的),转换成逻辑输入 C 回送给主模块。主模块得到数据 C 之后,控制中心变换模块(图中的),将 C 加工成 D。在调用传出模块输出 D 时,由传出模块调用后处理模块(图中的),将 D 加工成适于输出的形式 E,最后输出结果 E。

图 3.18 变换型的系统结构图

3) 事务型系统结构图——典型的系统结构形式之二

另一类典型的数据处理问题是事务型的。通常它是接受一项事务,根据事务处理的特点和性质,选择分派一个适当的处理单元,然后给出结果。我们把完成选择分派任务的部分叫作事务处理中心。或分派部件。这种事务型数据处理问题的数据流图参看图 3.19。其中,输入数据流在事务中心处作出选择,通过调度激活某一种事务处理加工。

事务型数据流图所对应的系统结构图是事务型系统结构图。如图 3.19 所示。在事务型系统结构图中,事务中心模块按所接受的事务的类型,选择某一个事务处理模块执行。各个事务处理模块是并列的,依赖于一定的选择条件,分别完成不同的事务处理工作。每个事务处理模块可能要调用若干个操作模块,而操作模块又可能调用若干个细节模块。由于不同的事务处理模块可能有共同的操作,所以某些事务处理模块可能共享一些操作模块。同样,不同的操作模块可以有相同的细节,所以,某些操作模块又可以共享一些细节模块。

事务型系统的结构图可以有多种不同的形式。例如,有多层操作层或没有操作层。
图 3.19 的简化形式是把分析作业和调度都归入事务中心模块,这样的系统结构图可

图 3.19 事务型系统结构图

以用如图 3.20 所示的结构图来表示。

图 3.20 简化的事务型系统结构图

3.4.2 变换分析

变换分析是系统结构设计的一种策略。运用变换分析方法建立初始的变换型系统结构图,然后对它作进一步的改进,最后得到系统的最终结构图。

变换分析方法的步骤如下。

1) 重画数据流图

在需求分析阶段得到的数据流图侧重于描述系统如何加工数据,而重画数据流图的出发点是描述系统中的数据是如何流动。因此,重画数据流图应注意以下几点:

以需求分析阶段得到的数据流图为基础重画数据流图时,可以从物理输入到物理输出,或者相反。还可以从顶层加工框开始,逐层向下。

在图上不要出现控制逻辑。用箭头表示的是数据流,而不是控制流。

不要去管系统的开始和终止(假定系统在不停地运行)。

省略每一个加工框的简单例外处理。

当数据流进入和离开一个加工框时,要仔细地标记它们,不要重名。

如有必要,可以使用逻辑运算符* (表示逻辑与)和 (表示异或)。

仔细检查每层数据流的正确性。

2) 在数据流图上区分系统的逻辑输入、逻辑输出和中心变换部分

在这一步,可以暂时不考虑数据流图的一些支流,例如错误处理等等。

如果设计人员的经验比较丰富,对要设计系统的软件规格说明又很熟悉,那么决定哪些加工框是系统的中心变换则是比较容易的。例如,几股数据流汇集的地方往往是系统的中心变换部分。在图 3.21 中,框 就是中心变换框。它有一个输入和两个输出,是数据流图内所有加工框中数据流比较集中的一个框。

图 3.21 数据流图中的输入、中心变换与输出部分

另外,可用以下的试探方法来确定系统的逻辑输入和逻辑输出在哪里。

从数据流图上的物理输入端开始,一步一步向系统的中间移动,一直到遇到的数据流不再被看作是系统到输入为止,则其前一个数据流就是系统的逻辑输入。也就是说,逻辑输入是离物理输入端最远的,但仍被看作是系统输入的数据流。类似地,从物理输出端开始,一步一步地向系统的中间移动,可以找到离物理输出端最远的,但仍被看作是系统输出的数据流,它就是系统的逻辑输出。从物理输入端到逻辑输入,构成系统的输入部分;从物理输出端到逻辑输出,构成输出部分;夹在输入部分和输出部分的就是中心变换部分。

中心变换部分是系统的中心加工部分。从输入设备获得的物理输入一般要经过编辑、数制转换、格式变换、合法性检查等一系列预处理,最后才变成逻辑输入传送给中心变换部分。同样,从中心变换部分产生的是逻辑输出,它要经过格式转换、组成物理块等一系列后处理,才成为物理输出。

有的系统仅有输入部分和输出部分,没有中心变换部分。

3) 进行一级分解,设计系统模块结构的顶层和第一层

自顶向下设计的关键是找出系统树形结构图的根或顶层模块。事实上,数据流图的中心变换部分应当与程序结构图的主要模块有着对应关系。我们首先设计一个主模块,并用程序的名字为它命名,然后将它画在与中心变换相对应的位置上。作为系统的顶层,它的功能是调用下一层模块,完成系统所要作的各项工作。

主模块设计好之后,下面的程序结构可按输入、中心变换和输出等分支处理。程序结构的第一层可以这样设计:为每一个逻辑输入设计一个输入模块,它的功能是为模块提供数据;为每一个逻辑输出设计一个输出模块,它的功能是将主模块提供的数据输出;为中心变换设计一个变换模块,它的功能是将逻辑输入转换成逻辑输出。

第一层模块与主模块之间传送的数据应与数据流图相对应,参看图 3.22。在图中,主模块控制、协调第一层的输入模块、变换模块和输出模块的工作。一般说来,它要根据一些逻辑(条件或循环)来控制对这些模块的调用。

4) 进行二级分解,设计中、下层模块

图 3.22 变换型问题的数据流图导出结构图

这一步工作是自顶向下,逐层细化,为每一个输入模块、输出模块、变换模块设计它们的从属模块。

设计下层模块的顺序是任意的。但一般是先设计输入模块的下层模块。

输入模块的功能是向调用它的上级模块提供数据,所以它必须要有一个数据来源。因而它必须有两个下属模块:一个是接收数据;另一个是把这些数据变换成它的上级模块所需的数据。但是,如果输入模块已经是原子模块,即物理输入端,则细化工作停止。同样,输出模块是从调用它的上级模块接收数据,用以输出,因而也应当有两个下属模块:一个是将上级模块提供的数据变换成输出的形式;另一个是将它们输出。设计中心变换模块的下层模块没有通用的方法,一般应参照数据流图的中心变换部分和功能分解的原则来考虑如何对中心变换模块进行分解。

图 3.22 是进行设计的例子。其中的“计算”是系统的核心数据处理部分,即中心变换。中心变换左边的“编辑”和“检验”是为“计算”作准备的预变换。预变换以后,送入主模块的数据流是系统的逻辑输入。中心变换送出的数据流是系统的逻辑输出。中心变换右边的“格式化 1”和“格式化 2”都是对计算值作格式化处理的变换。

运用变换分析方法建立系统的结构图时应当注意以下几点:

在选择模块设计的次序时,不一定要沿一条分支路径向下,直到该分支的最底层

模块设计完成后,才开始对另一条分支路径的下层模块进行设计。但是,必须对一个模块的全部直接下属模块都设计完成之后,才能转向另一个模块的下层模块的设计。参看图 3. 23。如果我们已设计了主模块和第一层 A, B, C 模块,下一步要分解模块 A。那么,应当先设计模块 A 的直接下属 E, F, G 模块,然后才可以去设计模块 B 和 C 的直接下属模块。

在设计下层模块时,应考虑模块的耦合和内聚问题,以提高初始结构图的质量。

注意“黑盒”技术的使用。在设计当前模块时,先把这个模块的所有下层模块定义成“黑盒”,并在系统设计中利用它们,暂不考虑它们的内部结构和实现方法。在这一步定义好的“黑盒”,由于已确定了它的功能和输入、输出,在下一步可以对它们进行设计和加工。这样,又会导致更多的“黑盒”。最后,全部“黑盒”的内容和结构应完全被确定。这就是我们所说的自顶向下、逐步求精的过程。使用黑盒技术的主要好处是使设计人员可以只关心当前的有关问题,暂时不必考虑进一步的琐碎的次要的细节,待进一步分解时才去关心它们的内部细节与结构。

图 3. 23 分解次序

3. 4. 3 事务分析

在很多软件应用中,存在某种作业数据流,它可以引发一个或多个处理,这些处理能够完成该作业要求的功能。这种数据流叫做事务。有关各个处理的数据流图和结构图在前面已经有过介绍,下面讨论如何从数据流图建立系统结构图。

与变换分析一样,事务分析也是从分析数据流图开始,自顶向下,逐步分解,建立系统到结构图。这里取图 3. 24(a)所示的数据流图为例。

1) 识别事务源——利用数据流图和数据词典,从问题定义和需求分析的结果中,找出各种需要处理的事务。通常,事务来自物理输入装置。有时,设计人员还必须区别系统的输入、中心加工和输出中产生的事务。在变换型系统的上层模块设计出来之后常常会遇到这种情形。

2) 规定适当的事务型结构——在确定了该数据流图具有事务型特征之后,根据模块划分理论,建立适当的事务型结构。例如,图 3. 19 和图 3. 20 所示的事务型系统结构图。

3) 识别各种事务和它们定义的操作——对于系统内部产生的事务,必须仔细地定义它们的操作。

4) 注意利用公用模块——在事务分析的过程中,如果不同事务的一些中间模块可由具有类似的语法和语义的若干个低层模块组成,则可以把这些低层模块构造成公用模块。

5) 对每一事务,或对联系密切的一组事务,建立一个事务处理模块——如果发现在系统中有类似的事务,可以把它们组成一个事务处理模块。但如果组合后的模块是低内聚的,则应该再打散重新考虑。

6) 对事务处理模块规定它们全部的下层操作模块——下层操作模块的分解方法类似于变换分析。但要注意,事务处理模块共享公用(操作)模块的情形相当常见。

7) 对操作模块规定它们的全部细节模块——对于大型系统的复杂事务处理,可能有若干层细节模块。另外,尽可能使类似的操作模块共享公用的细节模块。

例如,在图 3.24(a)所示的数据流图中,首先确定了它是具有事务型特征的数据流图。也就是说,数据流 A 是一个带有“请求”性质的信息,即为事务源。而加工 I 则具有“事务中心”的功能,它后继的 3 个加工 L、M、N 是并列的,在加工 A 的选择控制下完成不同功能的处理。最后,经过加工 O 将某一加工处理的结果整理输出。

为此,首先建立一个主模块用以代表整个加工,它位于 P—层(即主层)。然后考虑被称为 T—层(事务层)第一层模块。这时,第一层模块只能是 3 类:取得事务、处理事务和给出结果。

在图 3.24(b)中,依据并列的 3 个加工,在主模块之下建立了 3 个事务模块,分别完成 L、M 和 N 的工作。并在主模块的下沿以菱形引出对这 3 个事务模块的选择,而在这些事务模块的左右两边则是对应于加工 I 和 O 的“取得 A”模块和“给出 H”模块。

各个事务模块下层的操作模块,即 A—层(活动层)和细节模块,即 D—层(细节层),在图中未画出,可以继续分解扩展,直至完成整个结构图。

图 3.24 事务型问题导出的系统结构图

变换分析是软件系统结构设计的主要方法。因为大部分软件系统都可以应用变换分析进行设计。但是,在很多情况下,仅使用变换分析是不够的,还需要用其它方法作为补充。事务分析就是最重要的一种方法。虽然不能说全部数据处理系统都是事务型的,但是很多数据处理系统属于事务型系统。一个典型的商业数据处理系统的主要组成部分(包括输入和输出部分)也可以使用事务处理方法。所以事务分析方法很重要。

一般,一个大型的软件系统是变换型结构和事务型结构的混合结构。通常利用以变换分析为主,事务分析为辅的方式进行软件结构设计。

在系统结构设计时,首先利用变换分析方法把软件系统分为输入、中心变换和输出 3 个部分,设计上层模块,即主模块和第一层模块。然后根据数据流图各部分的结构特点,适当地利用变换分析或事务分析,可以得到初始系统结构图的某个方案。

图 3.25 所示的例子是一个典型的变换k 事务混合型问题的结构图。系统的输入、中心变换、输出 3 个部分是利用变换分析方法确定的,由此得到主模块“x x 系统”及其下属

的第一层模块“得到 D ”、“变换 ”和“给出 K ”。对图中的输入部分和变换部分又可以利用事务分析方法进行设计。例如,模块“调度 BC ”及其下属模块、模块“变换 ”及其下属模块都属于事务型。

图 3. 25 一个典型的变换k 事务混合型问题的结构图

3. 4. 4 软件模块结构的改进

为了改进系统的初始模块结构图,人们经过长期软件开发的实践,得到了一些启发式规则,利用它们,可以帮助设计人员改进软件设计,提高设计的质量。

1) 模块功能的完善化

一个完整的功能模块,不仅能够完成指定的功能,而且还应当能够告诉使用者完成任务的状态,以及不能完成的原因。也就是说,一个完整的模块应当有:

执行规定的功能的部分;

出错处理的部分。当模块不能完成规定的功能时,必须回送出错标志,向它的调用者报告出现这种例外情况的原因。

所有上述部分,都应当看作是一个模块的有机组成部分,不应分离到其它模块中去,否则将会增大模块间的耦合程度。

2) 消除重复功能, 改善软件结构

在系统的初始结构图得出之后, 应当审查分析这个结构图。如果发现几个模块的功能有相似之处, 可以加以改进。

完全相似: 在结构上完全相似, 可能只是在数据类型上不一致。此时可以采取完全合并的方法, 只需在数据类型的描述上和变量定义上加以改进就可以了。

图 3.26 相似模块的各种合并方案

局部相似: 如图 3.26(a) 所示, 虚线框部分是相似的。此时, 不可以把两者合并为一, 如图中(b)所示, 因为这样在合并后的模块内部必须设置许多开关, 如图中(f)所示, 势必把模块降低到逻辑内聚一级。一般处理办法是分析 R1 和 R2, 找出其相同部分, 从 R1 和 R2 中分离出去, 重新定义成一个独立的下一层模块。R1 和 R2 剩余的部分根据情况还可以与它的上级模块合并, 以减少控制的传递、全局数据的引用和接口的复杂性, 这样就形成了图中如(c)、(d)、(e)的各种方案。

3) 模块的作用范围应在控制范围之内

模块的控制范围包括它本身及其所有的从属模块。如图 3.27(a), 模块 A 的控制范围为模块 A, B, C, D, E, F, G。模块 C 的控制范围为模块 C, F, G。模块的作用范围是指模块内一个判定的作用范围, 凡是受这个判定影响的所有模块都属于这个判定的作用范围。如果一个判定的作用范围包含在这个判定所在模块的控制范围之内, 则这种结构是简单的, 否则, 它的结构是不简单的。

下面给出几种不同的作用范围/控制范围的实例, 并讨论模块间的关系。

图 3.27 中(b)表明作用范围不在控制范围之内。模块 G 作出一个判定之后, 若需要模块 C 工作, 则必须把信号回送给模块 D, 再由 D 把信号回送给模块 B。这样就增加了数据的传送量和模块间的耦合, 使模块之间出现了控制耦合, 这显然不是一个好的设计。图中反白框表示判定的作用范围。图中(c)虽然表明模块的作用范围是在控制范围之内, 可是判定所在模块 TOP 所处层次太高, 这样也需要经过不必要的信号传送, 增加了数据的传送量。虽然可以用, 但不是较好的结构。图中(d)表明作用范围在控制范围之内, 只有一个判定分支有一个不必要的穿越, 是一个较好的结构。图中(e)是一个比较理想的结构。

从以上的比较中可知, 在一个设计得很好的系统模块结构图中, 所有受一个判定影响

图 3.27 模块作用范围与影响范围的关系

的模块应该都从属于该判定所在的模块,最好局限于作出判定的那个模块本身及它的直接下属模块,如图中(e)那样。

4) 尽可能减少高扇出结构,随着深度增大扇入

模块的扇出数,是指模块调用子模块的个数。如果一个模块的扇出数过大,就意味着该模块过分复杂,需要协调和控制过多的下属模块。一般说来,出现这种情况是由于缺乏中间层次。应当适当增加中间层次的控制模块。如图 3.28(a)所示,模块 P 的扇出数为 10,属于高扇出结构。通过增加两个中间层次的模块 P1 和 P2,可将模块 P 改造成如图中(b)所示的模块结构。比较适当的扇出数为 2~5,最多不要超过 9。

图 3.28 高扇入和高扇出的分解

模块的扇出数过小,例如总是 1,也不好。这样将使得结构图的深度大大增加,不但增大了模块接口的复杂性,而且增加了调用和返回上的时间开销,降低了工作效率。

一个模块的扇入数越大,则共享该模块的上级模块数目越多。但如果一个模块的扇入数太大,例如超过 8,而它又不是公用模块,说明该模块可能具有多个功能。在这种情况下应当对它进一步分析并将其功能分解。如图 3.28(c)所示模块 Q 的扇入数为 9,它又不是公用模块,通过分析得知它是 3 功能的模块。对它进行分解,增加 3 个中间控制模块 Q1, Q2 和 Q3,而把真正公用部分提取出来留在 Q 中,使它成为这 3 个中间模块的公用模块,使各模块的功能单一化,从而改善了模块结构,如图中(d)所示。经验证明,一个设计得很好的软件模块结构,通常上层扇出比较高,中层扇出较少,底层扇入到有高扇入的公用模

块中。

5) 模块的大小要适中

模块的大小, 可以用模块中所含语句的数量的多少来衡量。有人认为限制模块的大小也是减少复杂性的手段之一, 因而要求把模块的大小限制在一定的范围之内。通常规定其语句行数在 50~100 左右, 保持在一页纸之内, 最多不超过 500 行。这对于提高程序的可理解性是有好处的, 但只能作一个参考数字。根本问题还是要保证模块的独立性。

6) 设计功能可预测的模块, 但要避免过分受限制的模块

一个功能可预测的模块可以被看成是一个“黑盒”, 不论内部处理细节如何, 但对相同的输入数据, 总能产生同样的结果。但是, 如果模块内部蕴藏有一些特殊的鲜为人知的功能时, 这个模块就可能是不可预测的。对于这种模块, 如果调用者不小心使用, 其结果将不可预测。图 3. 29(a) 是一个功能不可预测的例子。在模块内部保留了一个内部标记 M, 模块在运行过程中由这个内部标记确定作什么处理。由于这个内部标记对于调用者来说是隐藏起来的, 因而调用者将无法控制这个模块的执行, 或者不能预知将会引起什么后果, 最终会造成混乱。

图 3. 29 可预测模块和受限模块

如果一个模块的局部数据结构的大小、控制流的选择或者与外界(人、硬软件)的接口模式被限制死了, 则很难适应用户新的要求或环境的变更, 给将来的软件维护造成了很大的困难, 使得人们不得不花费更大的代价来消除这些限制。

为了能够适应将来的变更, 软件模块中局部数据结构的大小应当是可控制的, 调用者可以通过模块接口上的参数表或一些预定义外部参数来规定或改变局部数据结构的大小。另外, 控制流的选择对于调用者来说, 应当是可预测的。而与外界的接口应当是灵活的, 也可以用改变某些参数的值来调整接口的信息, 以适应未来的变更。

3.5 结构化数据系统开发方法(DSSD) ——面向数据结构的设计方法之一

在第 2 章介绍了结构化数据系统开发方法(DSSD)中面向分析的几个步骤,下面讨论设计的方法。图 3.30 所示的 Warnier 图描述了 DSSD 设计的过程。在 DSSD 的设计过程中,输入是指应用环境、功能描述、问题结果等需求分析的信息。在这些信息表示中所包含的各种图和数据将是 DSSD 进行逻辑设计和物理设计的基础。逻辑设计集中在输出、接口和软件的过程设计,物理设计则引伸了逻辑设计,集中于软件的“存储”,以最有效地实现期望的性能、可维护性和其它系统环境所要求的设计限制。

图 3.30 DSSD 设计过程

DSSD 方法是一种文档化的易于理解的软件设计方法。为了便于学习,这里介绍一种简化的 DSSD 方法。

3.5.1 一种简化的设计方法

逻辑设计处理可以分成两种活动:导出逻辑输出结构(LOS, logical output structure),及定义逻辑处理结构(LPS, logical process structure)。对于导出 LOS,已经提出了一种简化的方法。在这种方法中,数据项作为问题信息域的一部分,采用与 Jackson 方法和 Warnier 方法相同的方式,分层进行组织。为了导出 LOS,需要以下 4 步处理。

- 1) 针对问题的陈述或相关的需求信息进行分析,把所有不能再作进一步划分的数据项(叫作原子项)列表。
- 2) 确定每个原子项出现的频度。
- 3) 对所有可以再分解的数据项进行分析。
- 4) 开发 LOS 的图形表示。

图 3.31 报告原型

为了具体说明以上的 4 步处理,下面介绍一个简单的实例。这就是在一个大型自动化制造信息系统中产生一个“机器工具使用日报表”的部分(图 3.31)。我们将用简化的 DSSD 设计方法导出 LOS 和 LPS。

3.5.2 导出逻辑输出结构

逻辑输出结构(LOS)是构成基于计算机的系统输出的各数据项的一个层次表示。导出 LOS 的第一步是提取所有的原子项(不能再分割的数据项)。这可以通过审查问题的

陈述, 或者通过检查原型报表的格式本身来实现。第二步是把每个原子项出现的频度记录下来, 得到如图 3. 32 所示的统计表。

当所有的原子项及它们的频度都定义了之后, 设计人员要开始检查组合项。组合项是由原子项与其它组合项复合而成的数据项或数据类。在这个例子中, 组合项有: 报表(出现 1 次)、工具类别(每个报表出现 t 次)、工具标识(每个工具类出现 i 次)。

利用图 3. 32 给出的信息, 并收集对组合项分析得到的结果, 就能够导出图 3. 33 所示的“ 机器工具使用日报表 ”的 Warnier -Orr 图。

图 3. 32 原子项的频度

图 3. 33 LOS 的 Warnier -Orr 图

3. 5. 3 导出逻辑处理结构(LPS)

逻辑处理结构(LPS)是为了对相应的 LOS 进行处理所需的软件过程性表示。DSSD 导出 LPS 的步骤如下。每个组合数据项被转换成一个重复结构, 其中加入适当的处理指令。

- 1) 从 LOP 的 Warnier -Orr 图中消去所有的原子项。
 - 2) 对所有的组合项(重复)加上一对 BEGIN...END 定界语句。
- 在执行了以上两步后, 得到了如图 3. 34 所示的 Warnier -Orr 图。
- 为了导出 LPS, 继续作下面的步骤。
- 3) 定义所有的初始化指令、结束指令或处理。
 - 4) 确定所有的计算或非数值处理。

在执行了第 3 步和第 4 步后, 得到了如图 3. 35 所示的扩展 Warnier -Orr 图。

图 3.34 由 LOS 导出 LPS(1)

图 3.35 由 LOS 继续导出 LPS(2)

- 5) 定义所有的输出指令和处理。
- 6) 定义所有的输入指令和处理。

作了上面两步后, 完成了 LPS 的规格说明, 如图 3.36 所示。

图 3.36 由 LOS 最后得到 LPS(3)

3.6 Jackson 系统开发方法 (JSD) ——面向数据结构的分析与设计方法之二

3.6.1 JSD 功能描述

Jackson 系统开发方法中第 4 步是功能描述。功能描述的目的在于利用数据流连接及状态向量连接,把已定义的功能进程连接到系统模型进程,从而扩充系统规格说明图。

在 Jackson 系统开发方法中定义了三种功能:

- 1) 嵌入功能: 通过把操作分配(或写入)到模型进程的结构正文内而作成的功能,是嵌入功能。
- 2) 强制功能: 此功能检查模型进程的状态向量,并给出输出结果。
- 3) 交互功能: 此功能检查模型进程的状态向量,写入一个作用于模型进程活动的数据流,或引入一个写出结果的操作。

功能进程的输出是系统的输出,可以是报告、对硬件设备的命令、或者任何其它的输出信息。

用前面给出的大学交通车服务系统 USS 的实例来具体说明功能的描述。考察列车(Shuttle)模型。在列车上安装了一个指示灯板,灯亮表示列车到达。指示灯 i 用亮灯与熄灯命令 LON(i) 和 LOFF(i) 控制其接通或切断。这时,可以在列车进程模型中嵌入一个功能,当列车到达站 i 时,它给 lamp(i) 写出一个接通命令,当列车驶离站 i 时,它给 lamp(i) 写出一个断开命令。因此,当列车在两个车站之间行驶的时候,它将写出一个与指示灯的控制命令完全一致的数据流 (lamp CMDS)。

为了实现指示灯控制, Shuttle-1 的结构正文作了如下的修改:

SHUTTLE- 1 seq	列车- 1
LON(1);	点亮指示灯 1;
getsv SV;	获取状态向量(SV);
WAIT- BDY iter while WAIT1	只要仍是 WAIT1, 则一直检测
getsv SV;	获取状态向量(SV);
WAIT- BDY end	等待循环基体完
LOFF(1);	熄灭指示灯 1;
LEAVE(1);	列车驶离站 1;
TRANSIT- BDY1 iter while TRANSIT1	只要仍是 TRANSIT1, 则一直检测
getsv SV;	获取状态向量(SV)
TRANSIT- BDY1 end	移动循环基体 1 完
SHUTTLE- BDY1 iter	列车循环基体 1
STATION seq	车站
ARRIVE(i);	列车到达站 i;
LON(i);	点亮指示灯 i;
WAIT- BDY iter while WAITi	只要仍是 WAITi, 一直检测
getsv SV;	获取状态向量(SV);
WAIT- BDY end	等待循环基体完
LOFF(i);	熄灭指示灯 i;
LEAVE(i);	列车驶离站 i;

TRANSIT - BDY iter while TRANSIT _i	只要仍是 TRANSIT ₁ , 一直检测
getsv SV;	获取状态向量(SV);
TRANSIT - BDY end	移动循环基体完
STATION end	车站基体完
SHUTTLE - BDY ₁ end	列车循环基体 1 完

从上面的结构正文可知, 最开始, 先发出一个命令(LON(1)), 接通指示灯面板上表明列车停在站 1 的显示信息。这是列车生存期的初始状态, 在列车没有驶离站 1 之前, 它将一直保持这个状态。一旦传感器探测到列车出站, 便熄灭相应的指示灯(LOFF(i)); 当探测到列车到站时, 便接通相应的指示灯(LON(i))。

还有一种功能是产生动力的命令: 启动(START)和制动(STOP), 用以控制列车的运行。当传感器探测到列车进站时, 就发出制动(STOP)命令; 当按钮按下(第一次), 请求列车服务, 而且列车正停在某一个车站等待时, 就发出启动(START)命令。

制动(STOP)命令的发出是由列车到达唯一决定。但是, 启动(START)命令发出的时间却受按钮和列车二者的制约。因此我们引入一个称之为 mcontrol(动力控制)的功能进程, 它根据从列车- 1(Shuttle- 1)进程和按钮(button)进程接收到的数据进行处理, 发出制动(STOP)或启动(START)命令。

在列车- 1(Shuttle- 1)进程和动力控制(mcontrol)进程之间通过数据流 S1D 进行连接。这意味着列车- 1 进程不能错过列车到达的信息, 而若是对状态向量进行周期性检查的话, 可能会错过。

下面再一次给出 Shuttle- 1 进程的结构正文, 这一次加进了指示灯和动力控制。

SHUTTLE- 1 seq	列车- 1
LON(1);	点亮指示灯 1;
getsv SV;	获取状态向量(SV);
WAIT - BDY iter while WAIT ₁	只要仍是 WAIT ₁ , 则一直检测
getsv SV;	获取状态向量(SV);
WAIT - BDY end	等待循环基体完
LOFF(1);	熄灭指示灯 1;
LEAVE(1);	列车驶离站 1;
TRANSIT - BDY ₁ iter while TRANSIT ₁	只要仍是 TRANSIT ₁ , 一直检测
getsv SV;	获取状态向量(SV)
TRANSIT - BDY ₁ end	移动循环基体 1 完
SHUTTLE - BDY ₁ iter	列车循环基体 1
STATION seq	车站
ARRIVE(i);	列车到达站 i;
write arrive to S1D;	将列车到达信息写入 S1D;
LON(i);	点亮指示灯 i;
WAIT - BDY iter while WAIT _i	只要仍是 WAIT _i , 一直检测
getsv SV;	获取状态向量(SV);
WAIT - BDY end	等待循环基体完
LOFF(i);	熄灭指示灯 i;

LEAVE(i);	列车驶离站 i;
TRANSIT- BDY iter while TRANSITi	只要仍是 TRANSIT1, 一直检测
getsv SV;	获取状态向量(SV);
TRANSIT- BDY end	移动循环基体完
STATION end	车站基体完
SHUTTLE- BDY1 end	列车循环基体 1 完
ARRIVE(1);	列车到达站 1;
write arrive to S1D;	将列车到达信息写入 S1D;
SHUTTLE- 1 end	列车- 1

必须确保 Shuttle-1 进程和 mcontrol 进程能够以足够高的频率执行 getsv SV 操作和从数据流 S1D 读取到达记录, 以便及时制动列车。时序限制、调度和实现将在 JSD 方法的下一步考虑。

为了完成这个 USS 实例,再回到“按钮”实体的模型。原来的“按钮”模型 Button -1 是一个关于按钮活动的精确描述。但现在我们还需要区分是列车实际开动前请求乘车的第一次按下, 还是后来的按下。为了满足这些需求, 设计出一个新的二级进程 Button-2 。

图 3.37 给出了该进程的 Jackson 结构图。

图 3.37 大学交通车服务——功能描述

功能进程检查 Button 的状态向量, 以确定是否还有未完成的乘车请求。当一个请求服务已经完成, 即列车已经到达请求的车站时, mcontrol 功能会通知 Button- 2 进程。这将通过传送从 Button- 1 进程接收来的到达记录实现。因此, 定义了一个关于 Button-2 进程的交互功能。

Button-2 进程的结构正文描述如下:

```

BUTTON-2  seq
    request:= no;
    read MBD and B1D;
    BUTTON-BDY  iter
        PUSH-GROUP  seq
            EXTRA-AR-BDY  iter while (ARRIVAL)
                read MBD and B1D;
```

```

        EXTRA-AR-BDY  end
        RQ-PUSH  seq
            request:= yes;
            read MBD and B1D;
        RQ-PUSH  end
        EXTRA-RQ-PUSH  iter while
            read MBD and B1D;
        EXTRA-RQ-PUSH  end
        ARRIVAL  seq
            request:= no;
            read MBD and B1D;
        ARRIVAL  end
    PUSH-GROUP  end
BUTTON-BDY  end
BUTTON-2  end

```

Button- 2 的输入由两个数据流组成,以一种称之为“粗合并”的方式加以合并。这种“粗合并”发生于读进程简单地接收数据流中的下一个记录之时。因此,记录处理的次序潜在地依赖于两个异步的写进程。在这个 USS 实例中,粗合并就足够了。当然, JSD 方法还提供了其它类型的合并方式,它们一般不会导致不确定性的结果,这里不再介绍。

图 3.38 给出的系统规格说明图反映了在功能描述步作出的所有改变。

图 3.38 对于功能 1 和功能 2 扩充了的 SSD

在 Shuttle-1 进程内的一个嵌入功能产生了指示灯控制命令,而一个新的功能进程 mcontrol 则把一个交互功能引入到 Shuttle- 2 进程中,并为列车产生动力控制命令。在 MDB 输出上的双线表明了“一对多”的连接。mcontrol 进程的结构可依据输入与输出数据结构导出。

3.6.2 决定系统时间特性

在这个 JSD 步骤中,设计人员将定义系统的时间限制。前几步设计步骤已建立了一个由顺序的进程组成的系统,而在此系统中的顺序的进程则通过数据流及直接检查状态向量进行通信。进程的相对调度将是不确定的。

一种能够用于同步进程的机制叫做时间间隔标志(time grain marker, 简称 TGM)。它是一个数据记录, 表明一个特别时间间隔的存在, 可用来使时间通道成为可能, 以影响一个进程的各个活动。

对于大学交通车服务(USS)实例的时间限制包括:

- 1) 在 STOP 命令内基于列车向前的速度和刹车的动力而必须留给它的时间。
- 2) 接通/切断指示灯面板的响应时间。

对于 USS 实例, 不需要引入任何特殊的同步机制。数据交换已经加入了某种程度的同步。

3. 6. 3 实现

JSD 方法的实现步基于早期的工作, 而提出从问题的数据结构导出未完成的程序或进程的结构的方法。下面将介绍 Jackson 程序设计方法(JSP)的概貌。这种方法所用到的映象是我们将这种设计法归于面向数据结构设计方法的原因。

用 Jackson 自己的话来说, 实现步的本质是“问题应当被分解为可以用三种结构形式表示的部件的层次结构。”Jackson 所说的“结构形式”是指顺序、选择和重复, 实际上, 它们是过程性构造, 并将成为结构化程序设计方法基础。

1) 数据结构表示法

Jackson 提出的数据结构表示是 Jackson 实体结构图的变种。3 种基本的构造类型, 如图 3. 39(a), (b) 和(c)所示。在图(a)中表示的是顺序结构, 即数据结构 A 由 B, C, D3 个成份组成, 且按 B, C, D 顺序排列。在图(b)中表示的是选择结构, 即数据结构 A 或者是由 B 组成, 或者是由 C 组成, 二者必居其一。可选择的数据(子结构或数据项)加“°”表示。在图(c)中表示的是重复结构, 即数据结构 A 由多个 B 子结构组成, 子结构用“*”加以标记。

图 3. 39 Jackson 数据结构图

图 3. 40 数据结构组合表示

三种基本结构可以组合, 形成更复杂的结构体系。如图 3. 40 所示, 结构 A 由子结构 B 的多次重复组成, 而子结构 B 又是由子结构 C 的多次重复和另一个子结构 D 组成; D 又是由子结构 E 或 F 组成。

这种数据结构图可以同样方便地应用于输入、输出和数据库结构。

下面用一个信用卡记账的例子具体说明。信用卡记账系统的输入数据结构是两个实际的账册, 它们对应的两个输入文件如图 3. 41 所示。

两个实际输入账册都是按顾客号码进行登录的, 所以两个输入文件也是以顾客号码

图 3. 41 信用卡记账系统的输入

组织记录的。在支付账册中, 每个顾客号码行上要登记支付金额、支付日期。那么在支付文件中的每个记录中也对应应有支付金额(AMT)和支付日期(DATE) 项, 另外还有标识这个记录的顾客号码(CNO) 项。在支付账册中是以顾客号码进行排序的, 顾客号码相同的支付记录在支付账册中排列在一起, 构成关于该顾客的顾客号码组, 在支付文件中也有与之对应的顾客号码组。在顾客主账册中, 每一个顾客号码行上登记了顾客号码(CNO) 和结余(BAL), 给出某一位顾客的支付能力情况。与之对应的顾客主文件每个记录也有这两项。两个输入文件的内容是一致的。对应这两个输入文件的输入数据结构画在文件图示的左侧。

图 3. 42(a) 给出了信用卡记账系统的输出记账报告, 在报告中有“ 总计 ”这一行, 它由两项内容组成, 即交易额总计和结余总计, 因此, 在报告中隐含了一个“ 店方总计 ”层次。报告中其它部分是顾客信息, 因此, 隐含了一个与“ 店方总计 ”同一层次的“ 顾客数据 ”子结构。图 3. 42(b) 给出了根据输入和输出数据结构的对应关系建立的输出文件。

依据上述情况可知, 构造数据结构时, 必须仔细分析各个数据元素之间的内在联系, 才能构造出反映真实情况的数据结构。如在上例中, 记账报告中的数据分两类, 即顾客数据和店方总计, 因此, 在数据结构中应当增加一个层次。

从层次性的输入和(或) 输出数据结构可以直接推导出程序或进程的过程性表示。例如从图 3. 42(b) 所示的输出文件的 Jackson 数据结构图导出的程序结构如图 3. 43 所示。

在确定程序或进程的结构时, 有三条规则可供参考:

- 1) 对于每对有对应关系的数据单元, 按照它们在数据结构中所在的层次, 在程序或进程结构的适当位置画一个程序框。

图 3. 42 信用卡记账系统的输出

图 3. 43 从输出数据结构导出的程序结构

2) 对于每个在输入数据结构中的数据单元, 若它在输出数据结构中没有对应的数据单元, 则为它在程序或进程结构的适当位置画一个程序框。

3) 对于每个在输出数据结构中的数据单元, 若它在输入数据结构中没有对应的数据

单元, 则为它在程序或进程结构的适当位置画一个程序框。

简单地说, 对于输入数据结构与输出数据结构中的数据单元, 每对有对应关系的数据单元合画一个程序框, 没有对应关系的所有数据单元, 各画一个程序框。

最后, 利用 Jackson 给出的三种图解逻辑(schematic logic), 表示程序或进程的执行逻辑。这种图解逻辑类似于程序设计语言, 实际上它是一种伪码表示(关于伪码的介绍在后面讨论)。三种基本控制结构的图解逻辑如图 3. 44 所示。

图 3. 44 三种基本控制结构的图解逻辑

在给出程序或进程的过程性描述时, 还需要添加一些必要的可执行操作。例如, 打开文件、读文件结束符、读表头数据项、读行数据项、打印符号行、打印数字行、关闭文件等等。把它们分配到程序或进程结构的适当位置, 以得到一个完整的过程性描述。

对于上例中“处理顾客数据”部分, 给出过程性描述如下:

```
PROCESS_CUST_DATA seq
    open PAY_FILE; open CUST_M_FILE;           分别打开支付文件和顾客主
    文件
    处理顾客号组 CNO_GROUP iter until eof: PAY_FILE;
    读支付文件下一个记录 PAY_FILE;
    读顾客主文件一个记录找老结余
    处理顾客号组中每个支付记录 CORD iter until end: CNO_GROUP;
    写出报告行    write report line;
    计算总支付额    compute total payments;
    读支付文件下一个记录 PAY_FILE;
    一位顾客数据处理完
    计算顾客总数 COMPUTE_CUST_TOTAL;
    计算结余 COMPUTE_BALANCE seq
    处理老结余    PROCESS_OLD_BALANCE;
    计算新结余    COMPUTE_NEW_BALANCE;
    写出报告行    write report line;
    计算结余完毕 COMPUTE_BALANCE end;
    支付文件处理完 CNO_GROUP end;
END PROCESS_CUST_DATA
```

用 JSP 方法得到的程序或进程结构图, 一般都需要求精和优化。因为这种方法是从输入输出数据结构导出程序结构图, 因此有些中间处理过程在结构图中反映不出来。在求精过程中, 可以对结构图进行改进和细化, 使之完整和易于实现。

第 4 章 详细设计描述的工具

概要设计完成了软件系统的总体设计,规定了各个模块的功能及模块之间的联系,进一步就要考虑实现各个模块规定的功能。从软件开发的工程化观点来看,在使用程序设计语言编制程序以前,需要对所采用算法的逻辑关系进行分析,设计出全部必要的过程细节,并给予清晰的表达,使之成为编码的依据。这就是详细设计的任务。

详细设计也叫过程设计或程序设计(program design),它不同于编码(coding)或编程(programming)。在详细设计阶段,要决定各个模块的实现算法,并精确地表达这些算法。前者涉及所开发项目的具体要求和每个模块规定的功能。以及算法的设计和评价,这不属于本书讨论的范围。后者需要给出适当的算法描述,为此应提供过程设计的表达工具。在理想情况下,算法过程描述应当采用自然语言表达,这样不熟悉软件的人要理解这些规格说明就比较容易,不需要重新学习。但是,自然语言在语法上和语义上往往具有多义性,常常要依赖上下文才能把问题交代清楚。因此,必须使用约束性更强的方式表达过程细节。

4.1 程序流程图(program flow chart)

程序流程图也称程序框图,是软件开发者最熟悉的一种算法表达工具。它独立于任何一种程序设计语言,比较直观、清晰,易于学习掌握。因此,至今仍是软件开发者最普遍采用的一种工具。

但是,流程图也存在一些严重的缺点。例如流程图使用的符号不够规范,常常使用一些习惯性用法。特别是表示程序控制流程的箭头,使用的灵活性极大,程序员可以不受任何约束,随意转移控制。这些问题常常会使程序质量受到很大的影响。为了消除这些缺点,应对流程图所使用的符号作出严格的定义,不允许人们随心所欲地画出各种不规范的流程图。

首先,为使用流程图描述结构化程序,必须限制流程图只能使用图 4.1 所给出的五种基本控制结构。

这五种基本的控制结构是:

- 1) 顺序型:几个连续的加工步骤依次排列构成;
- 2) 选择型:由某个逻辑判断式的取值决定选择两个加工中的一个;
- 3) 先判定(while)型循环:在循环控制条件成立时,重复执行特定的加工;
- 4) 后判定(until)型循环:重复执行某些特定的加工,直至控制条件成立;
- 5) 多情况(case)型选择:列举多种加工情况,根据控制变量的取值,选择执行其一。

任何复杂的程序流程图都应由这五种基本控制结构组合或嵌套而成。作为上述五种控制结构相互组合和嵌套的实例,图 4.2 给出一个程序的流程图。图中增加了一些虚线

框, 目的是便于理解控制结构的嵌套关系。显然, 这个流程图所描述的程序是结构化的。

图 4. 1 流程图的基本控制结构

图 4. 2 嵌套构成的流程图实例

其次, 需要对流程图所使用的符号作出确切的规定。除去按规定使用定义了的符号之外, 流程图中不允许出现任何其它符号。图 4. 3 给出国际标准化组织提出, 并已为我国国家技术监督局批准的一些程序流程图标准符号, 其中多数所规定的使用方法与普通的使用习惯用法一致。

需要说明的几点是:

- 1) 循环的界限设有一对特殊的符号。循环开始符是削去上面两个直角的矩形, 循环结束符是削去下面两个直角的矩形, 其中应当注明循环名和进入循环的条件(对于 while

型循环)或循环终止的条件(对于 until 型循环)。通常这两个符号应在同一条纵线上,上下对应,循环体夹在其间。参看图 4.4 表示的两种类型循环的符号用法。

图 4.3 标准程序流程图的规定符号

- 2) 流线表示控制流的流向。在自上而下,或自左而右的自然流向情形,流线可不加箭头。否则必须在流线上加上箭头。
- 3) 注解符可用来标识注解内容,其虚线连在相关的符号上,或连接一个虚线框(框住一组符号)。参看图 4.5 例子。

图 4.4 循环的标准符号

图 4.5 注解符的使用

4) 判断有一个入口,但有多个可选出口。在判断条件取值后有一个且仅有一个出口被激活。取值结果可在流线附近注明。显然,两出口的判断就是前面提到的选择型结构,多出口的判断即为 CASE 型结构。图 4.6 给出多出口判断的表示。图(a)、(b)和(c)图分别表示具有 3,5 和 4 个出口的判断。

图 4.6 多出口判断

- 5) 虚线表示两个或多个符号间的选择关系(例如,虚线连接了两个符号,则表示这两个符号中只选用其中的一个。)另外,虚线也可配合注解使用,参看图 4.5。
- 6) 外接符及内接符表示流线在另外一个地方接续,或者表示转向外部环境或从外部环境转入。

4.2 N-S 图

Nassi 和 Shneiderman 提出了一种符合结构化程序设计原则的图形描述工具,叫作盒图(box-diagram),也叫 N-S 图。在 N-S 图中,为了表示五种基本控制结构,规定了五种图形构件,参看图 4.7。

图 4.7 N-S 图的五种基本控制结构

表示按顺序先执行处理 A,再执行处理 B。表示若条件 P 取真值,则执行“T”下面框 A 的内容;取假值时,执行“F”下面框 B 的内容。若 B 是空操作,则拉下一个箭头“ ”。和 表示两种类型的循环,P 是循环条件,S 是循环体。其中, 是先判断 P 的取值,再执行 S; 是先执行 S,再判断 P 的取值。 给出了多出口判断的图形表示,P 为控制条件,根据 P 的取值,相应地执行其值下面各框的内容。

为了说明 N-S 图的使用,仍沿用图 4.2 给出的实例,将它用如图 4.8 所示的 N-S 图表示。

图 4.8 N-S 图的实例

N-S 图有以下几个特点:

- 1) 图中每个矩形框(除 CASE 构造中表示条件取值的矩形框外)都是明确定义了的

功能域(即一个特定控制结构的作用域),以图形表示,清晰可见。

- 2) 它的控制转移不能任意规定,必须遵守结构化程序设计的要求。
- 3) 很容易确定局部数据和(或)全局数据的作用域。
- 4) 很容易表现嵌套关系,也可以表示模块的层次结构。

如前所述,任何一个 N-S 图,都是前面介绍的五种基本控制结构相互组合与嵌套的结果。当问题很复杂时,N-S 图可能很大,在一张纸上画不下,这时,可给这个图中一些部分取个名字,在图中相应位置用名字(用椭圆形框住它)而不是用细节去表现这些部分。然后在另外的纸上再把这些命名的部分进一步展开。

例如,图 4. 9(a)中判断 X1 取值为“ T ”部分和取值为“ F ”部分,用矩形框界定的功能域中画有椭圆形标记 k 和 l,表明了它们的功能进一步展开在另外的 N-S 图,即图 4. 9(b)与 4. 9(c)中。

图 4. 9 N-S 图的扩展表示

4. 3 PAD

PAD 是 problem analysis diagram 的缩写,它是日本日立公司提出,由程序流程图演化来的,用结构化程序设计思想表现程序逻辑结构的图形工具。现在已为 ISO 认可。

PAD 也设置了五种基本控制结构的图式,并允许递归使用。这些控制结构的图式如图 4. 10 所示。其中 表示按顺序先执行 A,再执行 B。 给出了判断条件为 P 的选择型结构。当 P 为真值时执行上面的 A 框,P 取假值时执行下面的 B 框中的内容。如果这种选择型结构只有 A 框,没有 B 框,表示该选择结构中只有 THEN 后面有可执行语句 A,没有 ELSE 部分。 与 中 P 是循环判断条件,S 是循环体。循环判断条件框的右端为双纵线,表示该矩形域是循环条件,以区别于一般的矩形功能域。 是 CASE 型结构。当判定条件 P= 1 时,执行 A1 框的内容,P= 2 时,执行 A2 框的内容,P= n 时,执行 An 框的内容等等。

作为 PAD 应用的实例,图 4. 11 给出了图 4. 2 程序的 PAD 表示。

为了反映增量型循环结构,在 PAD 中增加了对应于

```
for i = n1 to n2 step n3 do
```

的循环控制结构,如图 4. 12(a)所示。其中,n1 是循环初值,n2 是循环终值,n3 是循环增量。

另外,PAD 所描述程序的层次关系表现在纵线上。每条纵线表示了一个层次。把

图 4. 10 PAD 的基本控制结构

图 4. 11 PAD 实例

图 4. 12 PAD 的扩充控制结构

PAD 图从左到右展开。随着程序层次的增加, PAD 逐渐向右展开, 有可能会超过一页纸, 这时, 对 PAD 增加了一种如图 4. 12(b) 所示的扩充形式。图中用实例说明, 当一个模块 A 在一页纸上画不下时, 可在图中该模块相应位置矩形框中简记一个“NAME A”, 再在另一页纸上详细画出 A 的内容, 用 def 及双下划线来定义作 A 的 PAD。这种方式可使在一张纸上画不下的图, 分在几张纸上画出, 还可以用它定义子程序。

PAD 所表达的程序, 结构清晰且结构化程度高。作为一种详细设计的图形工具, PAD 比流程图更容易读。图中最左纵线是程序的主干线, 即程序的第一层结构。其后每增加一个层次, 图形向右扩展一条纵线。因此, 程序中含有的层次数即为 PAD 中的纵线数。

PAD 的执行顺序从最左主干线的上端的结点开始,自上而下依次执行。每遇到判断或循环,就自左而右进入下一层,从表示下一层的纵线上端开始执行,直到该纵线下端,再返回上一层的纵线的转入处。如此继续,直到执行到主干线的下端为止。

由于 PAD 的树形特点,使它比流程图更容易在计算机上处理。

4.4 PDL

PDL 是一种用于描述功能模块的算法设计和加工细节的语言。称为设计程序用语言。它是一种伪码(pesude code)。一般地,伪码的语法规则分为“外语法(outer syntax)”和“内语法(inter syntax)”。外语法应当符合一般程序设计语言常用语句的语法规则;而内语法可以用英语中一些简单的句子、短语和通用的数学符号,来描述程序应执行的功能。PDL 是这样一种伪码。

下面举一个例子,看 PDL 的使用。

PROCEDURE spellcheck IS	查找错拼的单词
BEGIN	
split document into single words	把整个文档分离成单词
load up words in dictionary	在字典中查这些单词
display words which are not in dictionary	显示字典中查不到的单词
create a new dictionary	造一新字典
END spellcheck	

从上面的例子可以看到,PDL 语言具有正文格式,很像一个高级语言。人们可以很方便地使用计算机完成 PDL 的书写和编辑工作。从其来源看,PDL 可能是某种高级语言(例如 PASCAL)稍加变化后得到的产物,例如在算法描述时常用的类 PASCAL 语言。PDL 还可能是为设计程序而专门设计的语言。

PDL 作为一种用于描述程序逻辑设计的语言,具有以下特点:

- 1) 有固定的关键字外语法,提供全部结构化控制结构、数据说明和模块特征。属于外语法的关键字是有限的词汇集,它们能对 PDL 正文进行结构分割,使之变得易于理解。为了区别关键字,规定关键字一律大写,其它单词一律小写。或者规定关键字加下划线,或者规定它们为黑体字。
- 2) 内语法使用自然语言来描述处理特性,为开发者提供方便,提高可读性。内语法比较灵活,只要写清楚就可以,不必考虑语法错,以利于人们可把主要精力放在描述算法的逻辑上。
- 3) 有数据说明机制,包括简单的(如标量和数组)与复杂的(如链表和层次结构)的数据结构。
- 4) 有子程序定义与调用机制,用以表达各种方式的接口说明。

使用 PDL 语言,可以做到逐步求精:从比较概括和抽象的 PDL 程序起,逐步写出更详细的更精确的描述。例如,前面给出的例子看起来比较粗糙,为进一步表明查找拼错的单词的 4 个步骤如何实现,可以对它每一步进行细化:


```
PROCEDURE spellcheck
```

```
  BEGIN
```

```
    --* split document into single words
```

```
    LOOP get next word
```

```
      add word to word list in sortorder
```

```
      EXIT WHEN all words processed
```

```
    END LOOP
```

```
    --* look up words in dictionary
```

```
    LOOP get word from word list
```

```
      IF word not in dictionary THEN
```

```
        --* display words not in dictionary
```

```
        display word prompt on user terminal
```

```
        IF user response says word OK THEN
```

```
          add word to good word list
```

```
        ELSE
```

```
          add word to bad word list
```

```
        ENDIF
```

```
      ENDIF
```

```
      EXIT WHEN all words processed
```

```
    END LOOP
```

```
    --* create a new words dictionary
```

```
    dictionary:= merge dictionary and good word list
```

```
  END spellcheck
```

在用 PDL 书写的正文中, 可以用“ --* ”打头的注释行对语句进行注解, 起到提高可读性的目的。作为一个例子, 考虑一种基于任一种比较通用的高级语言而得到的 PDL。

1) 数据说明

它的功能是定义数据的类型和作用域, 其一般形式是

```
TYPE < 变量名> AS < 限定词 1> < 限定词 2>
```

其中, < 变量名> 是一个模块内部使用的变量或模块间共用的全局变量名。< 限定词 1> 指明数据类型, 计有 SCALE(纯量), LIST(表), ARRAY(数组), CHAR(字符), STRUCT(结构)等。而< 限定词 2> 指明该变量的作用域, 即在模块或程序环境中该变量如何使用。

PDL 还允许定义用于问题定义的抽象数据类型。例如,

```
TYPE table0 IS INSTANCE OF symboltable
```

symboltable 是抽象数据类型, 将根据设计中其它部分的数据类型来定义。

以一个 CAD 系统为例。我们用 PDL 定义 drawing, 它必须是对许多模块都有效的全局性数据。drawing 是由不同的数据类型构成的, 并具有混合结构的特点。用 PDL 对 drawing 的定义如下。

TYPE drawing IS STRUCTURE DEFINED	--* 绘图
number IS STRING LENGTH (12);	--* 图形数目
geometry DEFINED	--* 几何形状
lines: (x, y) start; (x, y) end; line. type;	--* 直线
circle: (x, y) center, radius, arc. angle;	--* 圆
point: (x, y);	--* 点
curve: (x[i], y[i]) for i> 2;	--* 曲线
notes IS STRING LENGTH (256);	--* 注解
BOM DEFINED	--* 材料清单
part. sequence IS LIST;	--* 零件序列
part. no: STRING format aa-nnnnnn;	--* 零件号
pointer IS PTR;	--* 指针
END drawing TYPE;	

注意, 上述关于 drawing 的描述并不是一个程序设计语言(programming language) 的描述。设计者应遵守 PDL 的所有外语法规则, 但在定义 drawing 的组成部分时使用任何方式都是允许的。当然, 在将 PDL 翻译成某一程序设计语言的源代码时必须精确地遵循该语言的语法。

2) 程序块

PDL 的过程成份是块结构的。也就是说, 伪码可以定义为许多块, 而块将作为一个单独的实体来执行。块可以按如下的方式划分:

```
BEGIN < 块名>
    < 一组伪码语句> ;
END
```

< 一组伪码语句> 则是由所有其它的 PDL 结构组成。例如:

BEGIN < draw-line-on-graphics-terminal>	--* 在图形终端上画直线
get end-points from display list;	--* 从显示表取端点
scale physical end-points to screen coordinates;	--* 标定端点屏幕坐标
DRAW a line using screen coordinates;	--* 用屏幕坐标画直线
END	

在上面的块定义中使用了一个专门的关键字 DRAW, 这说明 PDL 还可以扩充用户的功能以满足某一个特定的应用。

3) 子程序结构

把 PDL 中的过程称为子程序, 它可以用如下的 PDL 结构定义:

```
PROCEDURE < 子程序名> < 一组属性>
INTERFACE < 参数表>
    程序块和/ 或一组伪码语句
END
```

其中, 子程序的< 一组属性> 描述了该子程序的引用特性(例如表明是一个 INTERNAL

模块, 还是一个 EXTERNAL 模块) 以及其它一些有关实现的特性(例如使用什么程序设计语言)。INTERFACE 用于定义模块的参数表, 包括所有输入和输出信息的标识符。

对子程序的调用使用关键字 PERFORM。如果子程序带有参数, 则调用语句为

```
PERFORM < 子程序名> USING < 参数表>
否则, 调用语句形式为
PERFORM < 子程序名>
```

4) 基本控制结构

- (1) 顺序型结构: 在这类结构中, 语句按排列的先后次序执行。
- (2) 选择型结构: 这类结构采取了传统的 if-then-else 形式:

```
IF < 条件描述>
    THEN < 程序块或伪码语句组> ;
    ELSE < 程序块或伪码语句组> ;
ENDIF
```

这里, 首先计算< 条件描述> 的值, 结果为真值(true)时, 执行 THEN 后面的部分, 结果为假值(false)时, 执行 ELSE 后面的部分, 或者什么也不作。例如, 下面给出的 PDL 程序段描述了一个工资单系统的判定序列:

```
IF year.to.date.FICA< maximum
    THEN BEGIN
        calculate FICA.deduction (see formula no. 30-1);
        IF (year.to.date.FICA+ FICA.deduction)> maximum
            THEN set FICA.deduction= maximum- year.to.date.FICA;
            ELSE skip;
        ENDIF
    END
ELSE set FICA.deduction= 0;
ENDIF
```

上面给出的 PDL 程序段有两个嵌套的 IF 语句。外层 IF 语句的 then 部分包含了一个程序块, 它由内层 IF 语句和一些伪码语句组成。ELSE skip 表示 else 部分被跳过。ENDIF 则用来明确地指出 IF 结构的结束, 在表达嵌套的 IF 语句时特别有用。END 跟在第一个 ENDIF 后面, 表示处理 FICA 信息的程序块的结束。

(3) 重复型结构

这类结构包括先测试型循环和后测试型循环, 以及下标型循环三种。作为一个 PDL 循环结构的例子, 考虑下面的起泡排序。

IF size of table > 1 THEN	--* 表中有不止一个元素则排序
REPEAT UNTIL no items were interchange	--* 后测试型循环, 直到表中不再有交换
DO FOR each pair of items in table	--* 下标型循环, 比较表中各对元素
IF first item of pair > seconditem of pair	--* 发生逆序

```

        THEN PERFORM interchange-the-two-items --* 则交换这两项
        ELSE skip --* 否则不工作
    ENDIF
ENDFOR
ENDREP
ENDIF

```

从例中可以看到,先测试型循环结构的一般形式为:

```

DO WHILE < 条件描述>
    < 程序块或伪码语句组> ;
ENDDO

```

后测试型循环结构的一般形式为

```

REPEAT UNTIL < 条件描述>
    < 程序块或伪码语句组> ;
ENDREP

```

或

```

DO LOOP
    < 程序块或伪码语句组> ;
    EXIT WHEN < 条件描述>
END LOOP

```

下标型循环结构,也称为步长型或增量型循环,其一般形式为

```

DO FOR < 下标= 下标表,表达式或序列>
    < 程序块或伪码语句组> ;
ENDFOR

```

作为对标准循环结构的补充, PDL 支持两个关键字 NEXT 和 EXIT。在实际应用中,有时需要从一个嵌套的循环中中断循环退出,使用 NEXT 和 EXIT,就能够有限制地违反一个纯结构化结构的规则,实现这一要求,参看图 4. 13。在图中可以看出,EXIT 使得控制转向 EXIT 所在循环后面的第一个语句,而 NEXT 使控制转移到 NEXT 所在循环的最后,并将开始下一个循环周期。通过给外层循环加标号,可以使用 EXIT 和 NEXT,从嵌套的循环中退出。例如,在图 4. 13 中,通过 ELSE EXIT loop-y,使控制从最内层循环中退出到加了标号 loop-y 的外层循环。

(4) 多路选择型结构

这类结构实际上是一组嵌套的 IF 语句的压缩。其表示为

```

CASE OF < case 变量名> :
    WHEN < case 条件 1> SELECT < 程序块或伪码语句组> ;
    WHEN < case 条件 2> SELECT < 程序块或伪码语句组> ;
    .....
    WHEN < 最后的 case 条件> SELECT < 程序块或伪码语句组> ;

```

DEFAULT: < 缺省或错误 case: 程序块或伪码语句组> ;
ENDCASE

图 4.13 “违例”退出

通常这种结构首先针对一组条件测试一个专门的参数(case 变量)。如果满足一种条件, 则执行与此条件对应的一个程序块或一个单独的伪码语句。作为 PDL 的 CASE 结构的例子, 考虑以下关于系统 I/O 处理的程序段:

```
CASE OF communication-status-bits(csb):  
  WHEN csb= clear-to-send SELECT  
    BEGIN  
      select channel path;  
      initiate message transmission;  
    END;  
  WHEN csb= clear-to-receive SELECT initiate buffer management;  
  WHEN csb= busy SELECT set queuing bit;  
  DEFAULT: process csb content error;  
ENDCASE
```

(5) 输入/ 输出结构

在 PDL 中, 输入/ 输出语句的规定是非常灵活的。典型的方式是:

READ/ WRITE TO < 设备> < I/O 表>

或者

ASK < 询问> ANSWER < 响应选项>

这里, < 设备> 指的是物理的 I/O 设备(例如 CRT、磁盘、打印机、磁带等), < I/O 表> 包含要传送的变量名。ASK-ANSWER 用于人机对话的设计, 这时宜于使用问答形式。例如,

ASK “ select processing option ” ANSWER “ COST ”, “ SCHEDULE ”;

I/O 的规定常常要扩展,以包括一些专门的特征,诸如声音输出或图形显示等等。需要注意的是,PDL 能够扩充。通过增加关键字,可以表达多任务处理、并发处理、中断处理、内部进程的同步以及许多其它的功能。在使用 PDL 的应用中,应当对 PDL 的最终形式加以裁定。

第 5 章 程序编码

前面介绍的软件工程步骤都是为了迈向一个最终的目标:即将“软件表示”变换成计算机能够“理解”的形式。本章介绍的内容是把“设计”变换成用程序设计语言编写的程序。

作为软件工程过程的一个阶段,程序编码是设计的继续。然而,在编码中遇到的问题,例如,程序设计语言的特性和程序设计风格会深刻地影响软件的质量和可维护性。本章不是具体介绍如何编写程序,而是从软件工程这个更广泛的范围讨论与程序设计语言及程序编码有关的问题。

5.1 对源程序的质量要求

为了保证程序编码的质量,程序员必须深刻地理解、熟练地掌握并正确地运用程序设计语言的特性,例如一些语法规则和语义的细节。只有语法上没有错误的程序才能通过编译系统的语法检查。然而,软件工程项目对代码编写的要求,绝不仅仅是源程序语法上的正确性,也不只是源程序中没有各种错误,此外,还要求源程序具有良好的结构性和良好的程序设计风格(programming style)。

为什么程序的正确性不是对程序质量的唯一要求呢?

也许有一天计算机有能力去理解人们用自然语言描述的程序要求,那将极大地简化软件的开发工作。或者能做到:这一台计算机能为那一台计算机编写源程序,甚至一台计算机能为自己编写源程序。如果能达到这一步,软件开发和维护工作就方便得多了。不仅可以免去软件设计和源程序编写的繁重而复杂的脑力劳动,而且可以免去令人厌烦的阅读程序之苦。然而,这毕竟是若干年之后才能做到的事。目前和今后的若干年内,人们编写源程序还只能用某种程序设计语言,并且写出的源程序除送入计算机运行外,还必须让人能够容易看懂。这一点作为软件工程项目和软件产品是一个必不可少的质量要求。实践表明,一个软件产品完成开发工作,投入运行以后,如果发生了问题,很难依靠原的开发人员解决。因为人员的工作流动是不可避免的。即使找到了当时的程序编写者本人,他们大多已无法记起几年前编写程序的许多细节了。因此,在程序编写时应考虑到,所写出的程序将被别人阅读,一定要尽量使程序写得容易被人读懂。

假如人们写出的源程序便于阅读,又便于测试和排除所发现的程序故障,就能够有效地在开发期间消除绝大多数在程序中隐藏的故障,使得程序可以做到正常稳定地运行,极大地减小了运行期间软件失效的可能性,大大提高了软件的可靠性。

如果写出的源程序在运行过程中发现了问题或错误时很容易修改,而且当软件在使用过程中,能根据用户的需要很容易扩充其功能及改善其性能,则这样的程序就具有较好的可维护性,维护人员可以很方便地对它进行修改、扩充和移植。

本章以后的几节将着重从程序的结构化和程序设计风格方面,讨论如何保证达到上

述程序的质量要求。

5.2 结构化程序设计

程序编码这一阶段的工作是把软件的详细设计变换成用某一种程序设计语言编写的可实现的源程序,结构化程序设计是在编写程序时首先必须考虑的问题。

5.2.1 关于 GOTO 语句的争论

早在 1963 年,针对当时流行的 ALGOL 语言,Peter Naur 指出,在程序中大量地,没有节制地使用 GOTO 语句,会使程序结构变得非常混乱。但是很多人还不太注意这一问题。以致许多人写出来的程序仍然是纷乱如麻的。参看图 5.1。

图 5.1 纷乱如麻的程序流程

1965 年,E. W. Dijkstra 在一次会议上提出,应当把 GOTO 语句从高级语言中取消。并指出,程序的质量与程序中包含的 GOTO 语句的数量成反比。在这种思想的影响下,当时新开发的几种高级程序设计语言,例如 LISP, ISWIM, BLISS 等,都把 GOTO 语句取消了。

1966 年,Bohm 与 Jacopini 证明了任何单入口单出口的没有“死循环”的程序都能由三种最基本的控制结构构造出来。这三种基本控制结构是“顺序结构”、“选择 IF- THEN- ELSE 结构”、“重复 DO- WHILE 或 DO- UNTIL 结构”。

1968 年,Dijkstra 在写给< ACM> (美国计算机协会通讯)杂志编辑部的信中再一次建议从一切高级语言中取消 GOTO 语句,只使用三种基本控制结构编写程序。他的建议引起了激烈的争论。争论集中在如何看待 GOTO 语句的问题上。赞成取消 GOTO 语句的一方认为,GOTO 语句对程序清晰性有很大破坏作用,凡是使用 GOTO 语句多的程序,其控制流时而 GOTO 向前,时而 GOTO 向后,常常使程序变得很难理解,从而增加查错和维护的困难,降低程序的可维护性。但以 D. E. Knuth 为代表的另一方认为,GOTO 语句虽然存在着破坏程序清晰性的问题,但不应完全禁止。因为 GOTO 语句概念简单、使用方便,在某些情况下,保留 GOTO 语句反能使写出的程序更加简洁,并且 GOTO 语句可直接得到硬件指令的支持。经过争论,人们认识到,不是简单地去掉 GOTO 语句的问题,而是要创立一种新的程序设计思想、方法和风格,以显著提高软件生产率和软件质量,降低软件维护的成本。20 世纪 70 年代初 N. Wirth 在设计 Pascal 语言时对 GOTO 语句的处理可被当作对 GOTO 语句争论的结论。在 Pascal 语言中设置了支持上述三种基本控

制结构的语句; 另一方面, GOTO 语句仍然保留在该语言中。不过, N. Wirth 解释说, 通常使用所提供的几种基本控制结构已经足够, 习惯于这样做的人不会感到 GOTO 语句的必要。也就是说, 在一般情况下, 可以完全不使用 GOTO 语句。如果在特殊情况下, 由于特定的要求, 偶然使用 GOTO 语句能解决问题, 那也未尝不可, 只是不应大量使用罢了。

5.2.2 结构化程序设计的原则

综合在围绕 GOTO 语句的争论中众多学者的意见, 对结构化程序设计的概念逐渐清晰起来。E. W. Dijkstra 提出了程序要实现结构化的主张, 并将这一类程序设计称为结构化程序设计 (structured programming)。其主要的原则有:

- 1) 使用语言中的顺序、选择、重复等有限的基本控制结构表示程序逻辑。
- 2) 选用的控制结构只准许有一个入口和一个出口。
- 3) 程序语句组成容易识别的块(block), 每块只有一个入口和一个出口。
- 4) 复杂结构应该用基本控制结构进行组合嵌套来实现。
- 5) 语言中没有的控制结构, 可用一段等价的程序段模拟, 但要求该程序段在整个系统中应前后一致。
- 6) 严格控制 GOTO 语句, 仅在下列情形才可使用:
 用一个非结构化的程序设计语言实现一个结构化的构造。
 在某种可以改善而不是损害程序可读性的情况下。

大量采用 GOTO 语句实现控制路径, 会使程序路径变得复杂而且混乱, 从而使程序变得不易阅读, 给程序的测试和维护造成困难, 还会增加出错的机会, 降低程序的可靠性。因此要控制 GOTO 语句的使用。但有时完全不用 GOTO 语句进行程序编码, 比用 GOTO 语句编出的程序可读性差。例如, 在查找结束时、文件访问结束时, 出现错误情况要从循环中转出时、使用布尔变量和条件结构实现就不如用 GOTO 语句来得简洁易懂。

例 1 图 5.2 是使用 FORTRAN 语言编写的一个打印 A, B, C 三数中最小者的程序的流程图。其中出现了 6 个 GOTO 语句, 程序可读性很差。

```

        if ( A .LT. B ) goto 120
        if ( B .LT. C ) goto 110
100    write ( 6, * ) C
        goto 140
110    write ( 6, * ) B
        goto 140
120    if ( A .LT. C ) goto 130
        goto 100
130    write ( 6, * ) A
140    continue
```

图 5.2 打印 A, B, C 三数中最小者

如果使用在 FORTRAN 中没有, 而 FORTRAN77 中才提供的 if- then- else 结构化构造, 则上述程序段可改成如下形式。

```

if ( A .LT. B .AND. A .LT. C ) then
```

```

        write ( 6, * ) A
    else if ( A . GE. B . AND. B . LT. C ) then
        write ( 6, * ) B
    else
        write ( 6, * ) C
    endif
endif
endif
```

这种程序结构清晰,可读性好。

例 2 设在闭区间 $[a, b]$ 上函数 $F(X)$ 有唯一的一个零点,如图 5.3 所示。下面给出一个用 C 语言写出的程序段,用二分法求方程 $F(X) = 0$ 在区间 $[a, b]$ 中的根。程序段中 X_0, X_1 是当前求根区间 $[X_0, X_1]$ 的下上界, X_m 是该区间的中点, eps 是一个给定的很小正数,用于迭代收敛的判断。在程序中采取了用 goto 语句和标号 finish 控制在循环中途转出循环。

```

程序段 1
F0 = F(a); F1 = F(b);
if ( F0* F1 <= 0 ) {
    X0 = a; X1 = b;
    for ( i = 1; i <= n; i++ ) {
        Xm = (X0+ X1)/2; Fm = F(Xm);
        if ( abs(Fm)< eps @abs(X1- X0)< eps )
            goto finish;
        if ( F0* Fm> 0 )
            { X0 = Xm; F0 = Fm; }
        else
            X1 = Xm;
    }
finish:    printf(" \n The root of this equation is % d\n ",Xm);
}
```

图 5.3 函数 $F(X)$ 曲线

这类循环结构出现了两个循环出口。一个是 for 循环的正常出口:当循环控制变量 i 超出了循环终值 n 时退出循环;另一个是 for 循环的非正常出口:当某种条件满足时,从循环中间某处转出循环,执行循环后面的语句。它不满足结构化的要求。作为对照,再看下面的两个程序段:

```

程序段 2
F0 = F(a); F1 = F(b);
if ( F0* F1 <= 0 ) {
    X0 = a; X1 = b;
    for ( i = 1; i <= n; i++ ) {
        Xm = (X0+ X1)/2; Fm = F(Xm);
        /* 区间两端点的函数值 */
        /* 区间中没有根,不作 */
        /* 设置当前求根区间的两个端点 */
        /* 最多允许迭代 n 次 */
        /* 求中点及中点的函数值 */
```

```

        if ( abs(Fm)< eps @abs(X1- X0)< eps ) break; /* 求到转出循环* /
        if ( F0* Fm> 0 )                               /* 没有求到, 缩小求根区间* /
            { X0 = Xm; F0 = Fm; }                       /* 向右缩小区间* /
        else
            X1 = Xm;                                     /* 向左缩小区间* /
    }
}

```

这段程序仍然不是结构化的程序, 利用了 C 语言中的一个语句 break, 它的功能是将控制转移到它所在循环的后面第一个后续语句处。它与 程序段 1 完成的工作相同, 由于将转移语句与转出条件的判断直接联系在一起, 可读性较好。

再看 程序段 3 , 它利用了一个布尔变量 finished, 该变量的初值为 false, 当循环中求到了要求的结果时, 将此变量的值改变为 true, 表示循环应结束, while 循环测试到 finished 为 true, 就自动退出循环, 执行后续的语句。

程序段 3

```

F0 = F(a); F1 = F(b);
if ( F0* F1 <= 0 ) {
    X0 = a; X1 = b; i = 1; finished = 0;
    while ( i <= n && finished == 0 ) {
        Xm = (X0+ X1)/2; Fm = F(Xm);
        if ( abs(Fm)< eps @abs(X1- X0)< eps ) finished = 1;
        if ( finished == 0 )
            if ( F0* Fm> 0 )
                { X0 = Xm; F0 = Fm; }
            else
                X1 = Xm;
    }
}

```

此程序段中各种结构均为单入口与单出口, 且没有 GOTO 语句, 可以说它是一个结构化的程序。它的结构很简单, 只有一重循环, 但由于引入一个布尔变量来控制循环结束, 可读性比 程序段 1 程序段 2 要差。在只有一重循环的情形, 差的程度还不很明显, 在多重循环的情形, 引入多个布尔变量, 可读性就很差了。理解程序的时间差几倍到几十倍不止。因此, 对于这种单入口多出口的循环, R. S. Pressman 明确说明, 用 GOTO 语句可得到较好的清晰性。

5.2.3 程序设计自顶向下, 逐步求精

在概要设计阶段, 已经采用自顶向下、逐步细化的方法, 把一个复杂问题的解法分解和细化成了一个由许多功能模块组成的层次结构的软件系统。在详细设计和编码阶段, 还

应当采取自顶向下、逐步求精的方法, 把一个模块的功能逐步分解, 细化为一系列具体的步骤, 进而翻译成一系列用某种程序设计语言写成的程序。

例如, 要求用筛选法求 100 以内的素数。所谓的筛选法, 就是从 2 到 100 中去掉 2, 3, ..., 9, 10 的倍数, 剩下的就是 100 以内的素数。

为了解决这个问题, 可先按程序功能写出一个框架。

```
main () {
    建立 2 到 100 的数组 A[ ], 其中 A[i] = i; ..... 1
    建立 2 到 10 的素数表 B[ ], 其中存放 2 到 10 以内的素数; ..... 2
    若 A[i] = i 是 B[ ] 中任一数的倍数, 则剔除 A[i]; ..... 3
    输出 A[ ] 中所有没有被剔除的数; ..... 4
}
```

上述框架中每一个加工语句都可进一步细化成一个循环语句。

```
main () {
    /* 建立 2 到 100 的数组 A[ ], 其中 A[i] = i * / ..... 1
    for ( i = 2; i <= 100; i++ ) A[i] = i;
    /* 建立 2 到 10 的素数表 B[ ], 其中存放 2 到 10 以内的素数 * / ..... 2
    B[1] = 2; B[2] = 3; B[3] = 5; B[4] = 7;
    /* 若 A[i] = i 是 B[ ] 中任一数的倍数, 则剔除 A[i] * / ..... 3
    for ( j = 1; j <= 4; j++ )
        检查 A[ ] 所有的数能否被 B[j] 整除并将能被整除的数从 A[ ] 中剔除; ..... 3.1
    /* 输出 A[ ] 中所有没有被剔除的数 * / ..... 4
    for ( i = 2; i <= 100; i++ )
        若 A[i] 没有被剔除, 则输出之 ..... 4.1
}
```

继续对 3.1 和 4.1 细化下去, 直到最后每一个语句都能直接用程序设计语言表示为止。

```
main () {
    /* 建立 2 到 100 的数组 A[ ], 其中 A[i] = i * /
    for ( i = 2; i <= 100; i++ ) A[i] = i;
    /* 建立 2 到 10 的素数表 B[ ], 其中存放 2 到 10 以内的素数 * /
    B[1] = 2; B[2] = 3; B[3] = 5; B[4] = 7;
    /* 若 A[i] = i 是 B[ ] 中任一数的倍数, 则剔除 A[i] * /
    for ( j = 1; j <= 4; j++ )
        /* 检查 A[ ] 所有的数能否被 B[j] 整除并将能被整除的数从 A[ ] 中剔除 * /
        for ( i = 2; i <= 100; i++ )
            if ( A[i] / B[j] * B[j] == A[i] )
                A[i] = 0;
    /* 输出 A[ ] 中所有没有被剔除的数 * /
    for ( i = 2; i <= 100; i++ )
```

```

/* 若 A[i] 没有被剔除, 则输出之 */
if ( A[i] != 0 )
    printf ( " A[%d]= %d\n ", i, A[i] );
}

```

自顶向下, 逐步求精方法的优点:

自顶向下、逐步求精方法符合人们解决复杂问题的普遍规律。可提高软件开发的成功率和生产率;

用先全局后局部、先整体后细节、先抽象后具体的逐步求精的过程开发出来的程序具有清晰的层次结构, 因此程序容易阅读和理解;

程序自顶向下、逐步细化, 分解成一个树形结构 (如图 5.4 所示)。在同一层的节点上作细化工作, 相互之间没有关系, 因此它们之间的细化工作相互独立。在任何一步发生错误, 一般只影响它下层的节点, 同一层其它节点不受影响。在以后的测试中, 也可以先独立地一个节点一个节点地作, 最后再集成。

图 5.4 程序的树形结构

程序清晰和模块化, 使得在修改和重新设计一个软件时, 可复用的代码量最大;

程序的逻辑结构清晰, 有利于程序正确性证明。

每一步工作仅在上层节点的基础上作不多的设计扩展, 便于检查;

有利于设计的分工和组织工作。

5.3 程序设计风格

有相当长的一段时间, 许多人认为程序只是给机器执行的, 而不是供人阅读的, 所以只要程序逻辑正确, 能被机器理解并依次执行就足够了。至于“文体(即风格)”如何是无关紧要的。但是, 随着软件规模越来越大, 复杂性增加, 人们逐渐看到, 在软件生存期中, 人们经常要阅读程序。特别是在软件测试阶段和维护阶段, 编写程序的人与参与测试、维护的人都要阅读程序。人们认识到, 阅读程序是软件开发和维护过程中的一个重要组成部分, 而且读程序的时间比写程序的时间还要多。因此, 程序实际上也是一种供人阅读的文章, 既然如此, 就有一个文章的风格问题。20 世纪 70 年代初, 有人提出在编写时, 应该使程序具有良好的风格。这个想法很快就为人们所接受。人们认识到, 程序员在编写程序时, 应当意识到今后会有人反复地阅读这个程序, 并沿着你的思路去理解程序的功能。所以应当在编写程序时多花些工夫, 讲求程序的风格, 这将大量地减少人们读程序的时间, 从整体

上看,效率是高的。

在这一节,将对程序设计风格的 4 个方面,即源程序文档化,数据说明的方法,语句结构和输入/输出方法中值得注意的问题进行概要的讨论,力图从编码原则的角度探讨提高程序的可读性,改善程序质量的方法和途径。

5.3.1 源程序文档化

源程序文档化包括选择好标识符(变量和标号)的名字、安排注释以及程序的视觉组织等等。

1) 符号名的命名

符号名即标识符,包括模块名、变量名、常量名、标号名、子程序名以及数据区名、缓冲区名等。这些名字应能反映它所代表的实际东西,应有一定的实际意义,使其能够见名知意,有助于对程序功能的理解。例如,表示次数的量用 times,表示总量用 total,表示平均值用 average,表示和的量用 sum 等。为达此目的,不应限制名字的长度。下面是三种不同的程序设计语言对同一变量的命名。

NEW.BALANCE.ACCOUNTS.PAYABLE	(PASCAL)
NBALAP	(FORTRAN)
N	(BASIC)

第一个是 PASCAL 语言中的命名,它给变量赋予一个明确的意义,在读程序时对它的使用可一目了然。第二个是 FORTRAN 语言中的命名,由于许多 FORTRAN 语言的版本规定其编译器只能识别名字的前 6~8 个字符,所以把变量名进行了缩写。它虽然提供了较多的信息,但由于一个字符代替了一个词的意思,一旦程序员误操作,意思可能就完全变了。第三个是 BASIC 语言中的命名,由于变量过于简单,使得该名字的含义不清。

名字不是越长越好,过长的名字会增加工作量,给程序员或操作员造成不稳定的情绪,会使程序的逻辑流程变得模糊,给修改带来困难。所以应当选择精炼的意义明确的名字,才能简化程序语句,改善对程序功能的理解。使用缩写名字时要注意缩写规则要一致,并且要给每一个名字加注释。同时,在一个程序中,一个变量只应用于一种用途。就是说,在同一个程序中一个变量不能身兼几种工作。例如在一个程序中定义了一个变量 temp,它在程序的前半段代表“温度(temperature)”,在程序的后半段则代表“临时变量(temporary)”,这样就会给读者阅读程序造成混乱。

2) 程序的注释

夹在程序中的注释是程序员与日后的程序读者之间通信的重要手段。正确的注释能够帮助读者理解程序,可为后续阶段进行测试和维护,提供明确的指导。因此注释决不是可有可无的,大多数程序设计语言允许使用自然语言写注释,这给阅读程序带来很大的方便。一些正规的程序文本中,注释行的数量占到整个源程序的 1/3 到 1/2。

注释分为序言性注释和功能性注释。

序言性注释通常置于每个程序模块的开头部分,它应当给出程序的整体说明,对于理解程序本身具有引导作用。有些软件开发部门对序言性注释作了明确而严格的规定,要求

程序编制者逐项列出。有关项目包括：

- 程序标题；
- 有关本模块功能和目的说明；
- 主要算法；
- 接口说明：包括调用形式，参数描述，子程序清单；
- 有关数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
- 模块位置：在哪一个源文件中，或隶属于哪一个软件包；
- 开发简历：模块设计者，复审者，复审日期，修改日期及有关说明等。

图 5.5 给出了一个序言性注释的例子。

```
C  TITLE : SUBROUTINE NGON
C  PURPOSE : THE PURPOSE OF TO CONTROL THE DRAWING OF NGONS
C  SAMPLE CALL : CALL NGON(KROW, IX, IY, KN)
C  INPUTS : KROW = IS THE LINE ON THE TABLET WHERE THE NEXT LINE
              OF
OUTPUT WILL BE PRINTED
IX = X. COORDINATE OF THE LEFT END OF THE BOTTOM SEGMENT
IY = Y. COORDINATE OF THE LEFT END OF THE BOTTOM SEGMENT
KN = IS THE NUMBER OF THE LAST NGON
C  OUTPUT : KROW = IS THE INCREMENTED ROW COUNTER
KN = IS THE INCREMENTED NGON COUNTER
C  SUBROUTINES REFERENCED : 1.) DBNGON
C                           2.) ALPHA
C                           3.) ROWCOL
C  PERTINENT DATA :
C  KROW IS CHECKED TO SEE IF TABLET IS FULL. IF IT IS THEN REPNT
C  IS CALLED TO REFRESH THE SCREEN AND PUT UP A NEW TABLET.
C  THE NGON COUNTER (KN) IS INCREMENTED AND THE POINTER ARRAY
C  PO IS WRITTEN TO THE DISPLAY FILE.
C
C  A PROMPT IS THEN ISSUED FOR THE NUMBER OF SIDES AND THE ORIEN-
C  TATION OF THE NGON WITH RESPECT TO THE X. AXIS. THE ARRAY NG
IS
C  LOADED AND WRITTEN TO THE OBJECT FILE.
C
C  THE ROUTINE DBNGON DOES THE ACTUAL DRAWING. IT REQUIRES THE
C  NUMBER OF SIDES, THE LENGTH OF A SIDE, THE ORIENTATION, AND THE
C  COORDINATES OF THE STARTING POINT AND IPEN.
C
C  AUTHOR : M. WRIGHT
C  AUDITOR : D. CURRIE
```

```

C   DATE : 10/30/86
C   MODIFICATIONS:
C       11/29/86 D. C.
C       CHANGES MADE TO ALLOW TABLES TO BE BUILT FOR REPNT
C       1/7/87 R. P. S.
C       ADD ERROR CHECKING COMMON SPACAL AND ERROR
HANDLING.

SUBROUTINE NGON (KROWW, IX, IY, KN)
DIMENSION PO (10), NG (10)
INTEGER * 2 ARG1, ARG2, ARG3, RPTFLG, RPTNUM

```

图 5.5 源程序的序言性注释

功能性注释嵌在源程序体中,用以描述其后的语句或程序段是在作什么工作,也就是解释下面要“作什么”,或是执行了下面的语句会怎么样。而不要解释下面怎么作,因为解释怎么作常常是与程序本身重复的。例如,

```

/* ADD AMOUNT TO TOTAL */
TOTAL = AMOUNT+ TOTAL

```

这样的注释行仅仅重复了后面的语句,对于理解它的工作并没有什么作用。如果注明把月销售额计入年度总额,便使读者理解了下面语句的意图:

```

/* ADD MONTHLY- SALES TO ANNUAL- TOTAL */
TOTAL = AMOUNT+ TOTAL

```

书写功能性注释,要注意以下几点:

- 用于描述一段程序,而不是每一个语句;
- 用缩进和空行,使程序与注释容易区别;
- 注释要正确。

有合适的,有助于记忆的标识符和恰当的注释,就能得到比较好的源程序内部的文档。有关设计的说明,也可作为注释,嵌入源程序体内。

3) 视觉组织k —空格、空行和移行(indentation)

一个程序如果写得密密麻麻,分不出层次来常常是很难看懂的。优秀的程序员在利用空格、空行和移行的技巧上显示了他们的经验。恰当地利用空格,可以突出运算的优先性,避免发生运算的错误。例如,将表达式

```
(A< - 17) ANDNOT ( B< = 49 )ORC
```

写成

```
( A< - 17 ) AND NOT ( B< = 49 ) OR C
```

就更清楚。自然的程序段之间可用空行隔开;移行也叫作向右缩格。它是指程序中的各行不必都在左端对齐,都从第一格起排列。因为这样作使程序完全分不清层次关系。因此,对于选择语句和循环语句,把其中的程序段语句向右作阶梯式移行。这样可使程序的逻辑

结构更加清晰,层次更加分明。例如,两重选择结构嵌套,写成下面的移行形式,层次就清楚得多。

```
IF (...) THEN
  IF (...) THEN
    .....
  ELSE
    .....
ENDIF
.....
ELSE
  .....
ENDIF
```

5.3.2 数据说明

虽然在设计阶段,已经确定了数据结构的组织及其复杂性。在编写程序时,则需要注意数据说明的风格。为了使程序中数据说明更易于理解和维护,必须注意以下几点。

1) 数据说明的次序应当规范化,使数据属性容易查找,也有利于测试、排错和维护。原则上,数据说明的次序与语法无关,其次序是任意的。但出于阅读、理解和维护的需要,最好使其规范化,使说明的先后次序固定。例如,常量说明,简单变量类型说明,数组说明,公用数据块说明,所有的文件说明。

在类型说明中还可进一步要求。例如,可按如下顺序排列:整型量说明,实型量说明,字符型量说明,逻辑量说明。

2) 当多个变量名用一个语句说明时,应当对这些变量按字母的顺序排列。类似地,带标号的全局数据(例如 FORTRAN 的公用块)也应当按字母的顺序排列。例如,把

```
INTEGER size, length, width, cost, price
```

写成

```
INTEGER cost, length, price, size, width
```

3) 如果设计了一个复杂的数据结构,应当使用注释说明在程序实现时这个数据结构的固有特点。例如对 PL/1 的链表结构和 PASCAL 中用户自定义的数据类型,都应当在注释中做必要的补充说明。

5.3.3 语句结构

在设计阶段确定了软件的逻辑流结构,但构造单个语句则是编码阶段的任务。语句构造力求简单、直接,不能为了片面追求效率而使语句复杂化。

1) 在一行内只写一条语句,并且采取适当的移行格式,使程序的逻辑和功能变得更加明确。例如,许多程序设计语言允许在一行内写多个语句。但这种方式会使程序可读性变差。因而不可取。例如,有一段排序程序。

```
FOR I = 1 TO N- 1 DO BEGIN T = I; FOR J = I+ 1 TO N DO IF
```

```

A[J] < A[T] THEN T = J; IF T = I THEN BEGIN WORK = A[T];
A[T] = A[I]; A[I] = WORK; END END;

```

由于一行中包括了多个语句,掩盖了程序的循环结构和条件结构,使其可读性变得很差。如将此程序段改写成如下形式:

```

FOR I = 1 TO N- 1 DO
  BEGIN
    T = I;
    FOR J = I+ 1 TO N DO
      IF A[J] < A[T] THEN T = J;
    IF T = I THEN
      BEGIN
        WORK = A[T];
        A[T] = A[I];
        A[I] = WORK;
      END
    END
  END;

```

这样的形式可使程序的逻辑和结构的层次变得十分清晰易读。

2) 程序编写首先应当考虑清晰性,不要刻意追求技巧性,使程序编写得过于紧凑。在20世纪50年代到70年代,为了能在小容量的低速计算机上完成工作量很大的计算,必须考虑尽量节省存储,提高运算速度。因此,对程序必须精心制作。但是近年来由于硬件技术的发展,已为软件人员提供了十分优越的开发环境。大容量和高速度的条件下,程序人员完全不必在程序中精心设置技巧。与此相反,软件工程技术要求软件生产工程化,规范化,为了提高软件开发的生产率,特别是提高程序的可读性,减少出错的可能性,要求把程序的清晰性放在首位。因此,写出的程序必须让人很容易读懂。例如,有一个用PASCAL语句写出的程序段:

```

A[I] = A[I] + A[T];
A[T] = A[I] - A[T];
A[I] = A[I] - A[T];

```

阅读此段程序,读者可能不易看懂。这段程序实际上就是交换A[I]和A[T]中的内容。目的是为了节省一个工作单元。如果改一下:

```

WORK = A[T];
A[T] = A[I];
A[I] = WORK;

```

就能让读者一目了然了。

3) 程序编写得要简单,写清楚,直截了当地说明程序员的用意。例如,下面是由三句FORTRAN语句组成的程序段:

```

DO 5 I= 1, N
DO 5 J= 1, N

```

$$5 \quad V(I, J) = (I/J) * (J/I)$$

事实上, 这是一个有双重循环的程序段, 得到的结果是一个 $N \times N$ 的二维数组, 只是说明语句被略去了。我们知道, 在 FORTRAN 语言中, 除法运算(/) 在除数和被除数都是整型量时, 其结果只取整数部分, 而得到整型量。因此, 程序运算得到的结果 V 是一个单位矩阵, 如图 5. 6(a) 所示。按 FORTRAN 的规则, 使用 V 作变量名, 实际上得到的结果还经过了整数向实数的转换。因此, 最后结果单位矩阵 V 就如图 5. 6(b) 所示。这个程序构思巧妙, 但不易理解, 读者可能要花很大的力气才能弄清程序编制者的真正意图, 这无疑给软件的维护带来很大困难。

图 5. 6 单位矩阵

如果我们写成以下的形式, 就能让读者直接了解程序编写者的意图了。

```
DO 5 I= 1, N
  DO 5 J= 1, N
    IF (I .EQ. J) THEN
      V(I, J) = 1.0
    ELSE
      V(I, J) = 0.0
    ENDIF
  5 CONTINUE
```

4) 除非对效率有特殊的要求, 程序编写要做到清晰第一, 效率第二。不要为了追求效率而丧失了清晰性。事实上, 程序效率的提高主要通过选择高效的算法来实现。同样的布局算法, 执行同样规模的集成电路芯片布局工作, 有的算法要 33 个小时, 有的算法只要 8 分钟。通过对程序代码的某些语句进行优化, 有时可提高一些效率, 但与选择好的算法提高效率相比, 在保持程序的清晰性的前提下, 宁可选择后者。

5) 首先要保证程序正确, 然后才要求提高速度。

6) 让编译程序作简单的优化。

7) 尽可能使用库函数。

8) 避免使用临时变量而使可读性下降。例如, 有的程序员为了追求效率, 往往喜欢把表达式 $X = A[I] + 1/A[I]$ 写成 $AI = A[I]; X = AI + 1/AI$ 。因为他意识到简单变量的运算比下标变量的运算要快。这样作, 虽然效率要高一些, 但引进了临时变量, 把一个计算公式拆成了几行, 增加了理解的难度。而且将来一些难以预料的修改有可能会更动这几行的顺序, 或在其间插入语句, 并顺带着(误)改变了这个临时变量的值, 就容易造成逻辑上的错误。不如在一个算式中表达较为安全可靠。

9) 尽量用公共过程或子程序去代替重复的功能代码段。要注意, 这段代码应具有一

个独立的功能,不要只因代码形式一样便将其抽出组成一个公共过程或子程序。

10) 使用括号清晰地表达算术表达式和逻辑表达式的运算顺序。例如,算术表达式 $X = A * B / C * D$,可能被人理解为 $X = (A * B / C) * D$,也可能被人理解为 $X = (A * B) / (C * D)$ 。所以最好添加括号,免得误解。

11) 避免不必要的转移。同时如果能保持程序的可读性,则不必用 GOTO 语句。下面举一例说明。看图 5.7 给出的流程图。

```
IF ( X .LT. Y ) GOTO 30
IF ( Y .LT. Z ) GOTO 50
SMALL= Z
GOTO 70
30  IF ( X .LT. Z ) GOTO 60
    SMALL= Z
    GOTO 70
50  SMALL= Y
    GOTO 70
60  SMALL= X
70  CONTINUE
```

图 5.7 求 X, Y, Z 中最小者

这个程序包括了 6 个 GOTO 语句,看起来很不好理解。仔细分析可知道它是想让 SMALL 取 X, Y, Z 中的最小值。这样作完全是不必要的。为求最小值,程序只需编写成:

```
SMALL= X
IF ( Y .LT. SMALL ) SMALL= Y
IF ( Z .LT. SMALL ) SMALL= Z
```

所以程序应当简单,不必过于深奥,避免使用 GOTO 语句绕来绕去。

12) 用逻辑表达式代替分支嵌套。例如,用

```
IF ( CHAR .GE. 0 .AND. CHAR .LE. 9 ) ...
```

来代替

```
IF ( CHAR. GE. 0 ) THEN
    IF ( CHAR .LE. 9 ) THEN ...
```

13) 避免使用空的 ELSE 语句和 IF...THEN IF...的语句。在早期使用 ALGOL 语言时发现这种结构容易使读者产生误解。例如,写出了这样的 BASIC 语句:

```
IF ( CHAR> = A ) THEN
IF ( CHAR< = Z ) THEN
PRINT " This is a letter。"
ELSE
PRINT " This is not a letter。"
```

这里的 ELSE 到底是否定的哪一个 IF? 语言处理程序约定,是否定离它最近的那个未带 ELSE 的 IF,但是不同的读者可能会产生不同的理解,出现了二义性问题。

14) 避免使用 ELSE GOTO 和 ELSE RETURN 结构。

15) 使与判定相联系的动作尽可能地紧跟着判定。

16) 避免采用过于复杂的条件测试。

17) 尽量减少使用“否定”条件的条件语句。例如, 如果在程序中出现

```
IF NOT((CHAR< 0) OR (CHAR> 9)) THEN .....
```

改成

```
IF (CHAR>= 0) AND (CHAR<= 9) THEN .....
```

不要让读者绕弯子想。

18) 避免过多的循环嵌套和条件嵌套。

19) 不要使 GOTO 语句相互交叉。看下面的 FORTRAN 程序:

```
MAXVAL= A(I)
DO 40 I= 2, 10
IF ( A(I) .GT. MAXVAL ) GOTO 30
GOTO 40
30 MAXVAL= A(I)
40 CONTINUE
```

GOTO 语句相互交叉在程序中会引起混乱, 几处交叉积累起来, 会使程序十分难懂。事实上, 把上面程序稍加改变就能简化程序。

```
MAXVAL= A(I)
DO 40 I= 2, 10
IF ( A(I) .GT. MAXVAL ) MAXVAL= A(I)
40 CONTINUE
```

20) 对递归定义的数据结构尽量使用递归过程。

21) 经常反躬自省:“ 如果我不是编码的人, 我能看懂它吗? ”考虑它的可理解性达到什么程度。

5.3.4 输入和输出(I/O)

输入和输出信息是与用户的使用直接相关的。输入和输出的方式和格式应当尽可能方便用户的使用。一定要避免因设计不当给用户带来的麻烦。因此, 在软件需求分析阶段和设计阶段, 应基本确定输入和输出的风格。系统能否被用户接受, 有时取决于输入和输出的风格。

输入/输出的风格随着人工干预程度的不同而有所不同。例如, 对于批处理的输入和输出, 总是希望它能按逻辑顺序要求组织输入数据, 具有有效的输入/输出出错检查和出错恢复功能, 并有合理的输出报告格式。而对于交互式的输入/输出来说, 更需要的是简单而带提示的输入方式, 完备的出错检查和出错恢复功能, 以及通过人机对话指定输出格式和输入/输出格式的一致性。

此外, 不论是批处理的输入/输出方式, 还是交互式的输入/输出方式, 在设计和程序编码时都应考虑下列原则:

1) 对所有的输入数据都进行检验, 从而识别错误的输入, 以保证每个数据的有效性;

- 2) 检查输入项的各种重要组合的合理性,必要时报告输入状态信息;
- 3) 使得输入的步骤和操作尽可能简单,并保持简单的输入格式;
- 4) 输入数据时,应允许使用自由格式输入;
- 5) 应允许缺省值;
- 6) 输入一批数据时,最好使用输入结束标志,而不要由用户指定输入数据数目;
- 7) 在以交互式输入/输出方式进行输入时,要在屏幕上使用提示符明确提示交互输入的请求,指明可使用选择项的种类和取值范围。同时,在数据输入的过程中和输入结束时,也要在屏幕上给出状态信息;
- 8) 当程序设计语言对输入/输出格式有严格要求时,应保持输入格式与输入语句的要求的一致性;
- 9) 给所有的输出加注解,并设计输出报表格式。

输入/输出风格还受到许多其它因素的影响。如输入/输出设备(例如终端的类型,图形设备,数字化转换设备等)、用户的熟练程度、以及通信环境等。

总之,要从程序编码的实践中,积累编制程序的经验,培养和学习良好的程序设计风格,使编写出来的程序清晰易懂,易于测试和维护。在程序编码阶段改善和提高软件的质量。

5.4 程序复杂性度量

程序复杂性主要指模块内程序的复杂性。它直接关系到软件开发费用的多少,开发周期的长短和软件内部潜伏错误的多少。同时它也是软件可理解性的另一种度量。

减少程序复杂性,可提高软件的简单性和可理解性,并使软件开发费用减少,开发周期缩短,软件内部潜藏错误减少。

为了度量程序复杂性,要求复杂性度量满足以下假设:

- 它可以用来计算任何一个程序的复杂性;
- 对于不合理的程序,例如对于长度动态增长的程序,或者对于原则上无法排错的程序,不应当使用它进行复杂性计算;
- 如果程序中指令条数、附加存储量、计算时间增多,不会减少程序的复杂性。

5.4.1 代码行度量法

度量程序的复杂性,最简单的方法是统计程序的源代码行数。此方法的基本考虑是统计一个程序模块的源代码行数,并以源代码行数作为程序复杂性的度量。若设每行代码的出错率为每 100 行源程序中可能有的错误数目,例如每行代码的出错率为 1%,则是指每 100 行源程序中可能有一个错误。

此时,源代码行数与每行代码的出错率之间的关系不太好估计。Thayer 曾指出,程序出错率的估算范围是从 0.04% ~ 7% 之间,即每 100 行源程序中可能存在 0.04 ~ 7 个错误。他还指出,每行代码的出错率与源程序行数之间不存在简单的线性关系。Lipow 进一步指出,对于小程序,每行代码的出错率为 1.3% ~ 1.8%; 对于大程序,每行代码的出错

率增加到 2.7% ~ 3.2% 之间,但这只是考虑了程序的可执行部分,没有包括程序中的说明部分。Lipow 及其他研究者得出一个结论:对于少于 100 个语句的小程序,源代码行数与出错率是线性相关的。随着程序的增大,出错率以非线性方式增长。所以,代码行度量法只是一个简单的,估计得很粗糙的方法。

5.4.2 McCabe 度量法

M McCabe 度量法是由 Thomas McCabe 提出的一种基于程序控制流的复杂性度量方法。McCabe 定义的程序复杂性度量值又称环路复杂度,它基于一个程序模块的程序图中环路的个数,因此计算它先要画出程序图。

程序图是退化的程序流程图。也就是说,把程序流程图中每个处理符号都退化成一个结点,原来联结不同处理符号的流线变成联接不同结点的有向弧,这样得到的有向图叫作程序图。

程序图仅描述程序内部的控制流程,完全不表现对数据的具体操作,以及分支和循环的具体条件。因此,它往往把一个简单的 IF 语句与循环语句的复杂性看成是一样的,把嵌套的 IF 语句与 CASE 语句的复杂性看成是一样的。

下面给出计算环路复杂性的方法。

根据图论,在一个强连通的有向图 G 中,环的个数由以下公式给出:

$$V(G) = m - n + p$$

其中, V(G)是有向图 G 中环路数, m 是图 G 中弧数, n 是图 G 中结点数, p 是图 G 中的强连通分量个数。

Myers 建议,对于复合判定,例如 (A= 0) and (C= D) or (X= A)计作 3 个判定。

在一个程序中,从程序图的入口点总能到达图中任何一个结点,因此,程序总是连通的,但不是强连通的。为了使图成为强连通图,从图的入口点到出口点加一条用虚线表示的有向边,使图成为强连通图。这样可以使用上式计算环路复杂性。

以图 5.8 所给的例子示范,其中,结点数 n= 11,弧数 m= 13, p= 1,则有

$$V(G) = m - n + p = 13 - 11 + 1 = 3.$$

即 McCabe 环路复杂度度量值为 3。它也可以看作由程序图中的有向弧所封闭的区域个数。假定模块是单入口单出口的程序,因此整个模块中只有一个程序,这时的 p= 1。因此可以简化公式,不加入从出口到入口的那条虚线而当作无向图:

$$V(G) = m - n + 2$$

当分支或循环的数目增加时,程序中的环路也随之增加,因此 McCabe 环路复杂度度量值实际上是为软件测试的难易程度提供了一个定量度量的方法,同时也间接地表示了软件的可靠性。实验表明,源程序中存在的错误数以及为了诊断和纠正这些错误所需的时间与 McCabe 环路复杂度度量值有明显的关系。

利用 McCabe 环路复杂度度量时,有几点说明。

- 1) 环路复杂度取决于程序控制结构的复杂度。当程序的分支数目或循环数目增加时其复杂度也增加。环路复杂度与程序中覆盖的路径条数有关。
- 2) 环路复杂度是可加的。例如,模块 A 的复杂度为 3,模块 B 的复杂度为 4,则模块 A

图 5.8 程序流程图及相应的程序图

与模块 B 的复杂度是 7。

3) McCabe 建议, 对于复杂度超过 10 的程序, 应分成几个小程序, 以减少程序中的错误。Walsh 用实例证实了这个建议的正确性。他发现, 在 276 个子程序中, 有 23% 的子程序的复杂度大于 10, 而这些子程序中发现的错误占总错误的 53%。而且复杂度大于 10 的子程序中, 平均出错率比小于 10 的子程序高出 21%。这说明在 McCabe 复杂度为 10 的附近, 存在出错率的间断跃变。

4) 这种度量的缺点: 对于不同种类的控制流的复杂性不能区分; 简单 IF 语句与循环语句的复杂性应同等看待; 嵌套 IF 语句与简单 CASE 语句的复杂性是一样的; 模块间接口当成一个简单分支处理; 一个具有 1000 行的顺序程序与一行语句的复杂性相同。

尽管 McCabe 复杂度度量法有许多缺点, 但它容易使用, 而且在选择方案和估计排错费用等方面都是很有效的。

5.4.3 Halstead 的软件科学

Halstead 软件科学确定计算机软件开发中的一些定量规律, 它采用以下一组基本的度量值, 这些度量值通常在程序产生之后得出, 或者在设计完成之后估算出。

1) 程序长度, 即预测的 Halstead 长度

令 n_1 表示程序中不同运算符(包括保留字)的个数, 令 n_2 表示程序中不同运算对象的个数, 令 H 表示“程序长度”, 则有

$$H = n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2$$

这里, H 是程序长度的预测值, 它不等于程序中语句个数。

在定义中, 运算符包括算术运算符、关系运算符、逻辑运算符、赋值符(= 或 =)、数组操作符、分界符(, 或; 或:)、子程序调用符、括号运算符、循环操作符等。特别地, 成对的运算符, 例如“BEGIN...END”、“FOR...TO”、“REPEAT...UNTIL”、“WHILE...DO”、

“ IF...THEN...ELSE ”、“ (...) ”等都当作单一运算符。

运算对象包括变量名和常数。

2) 实际的 Halstead 长度

设 N1 为程序中实际出现的运算符总个数, N2 为程序中实际出现的运算对象总个数, N 为实际的 Halstead 长度, 则有

$$N = N1 + N2$$

3) 程序的词汇表

Halstead 定义程序的词汇表为不同的运算符种类数和不同的运算对象种类数的总和。若令 n 为程序的词汇表, 则有

$$n = n1 + n2$$

图 5.9 是用 FORTRAN 语言写出的交换排序的例子。

```
SUBROUTINE SORT(X,N)
DIMENSION X(N)
IF (N .LT. 2) RETURN
DO 20 I= 2,N
  DO 10 J= 1,I
    IF ( X(I) .GE. X(J) ) GO TO 10
    SAVE = X(I)
    X(I)= X(J)
    X(J)= SAVE
10  CONTINUE
20  CONTINUE
RETURN
END
```

交换排序程序的运算符交换排序程序的运算对象

运 算 符	计 数	运 算 对 象	计 数
可执行语句结束	7	X	6
数组下标	6	I	5
=	5	J	4
IF()	2	N	2
DO	2	2	2
,	2	SAVE	2
程序结束	1	1	1
.LT.	1	n2= 7 N 2= 22	
.GE.	1		
GO TO 10	1		
n= 10	N 1= 28		

图 5.9 一个交换排序程序的例子

4) 程序量可用下式算得

$$V = N \times \log_2(n_1 + n_2)$$

它表明了程序在“词汇上的复杂性”。

对于图 5.9 的例子, 利用 n, N_1, n_2, N_2 , 可以计算得

$$V = (28 + 22) \times \log_2(10 + 7) = 204.$$

等效的汇编语言程序的 $V = 328$ 。这说明汇编语言比 FORTRAN 语言需要更多的信息量 (以 bit 表示)。

$$H = 10 \times \log_2 10 + 7 \times \log_2 7 = 52.87$$

$$N = 28 + 22 = 50$$

5) 程序员工作量

$$E = V/L \quad \text{或} \quad E = H \times \log_2(n_1 + n_2) \times [(n \times N_2) / (2 \times n_2)]$$

6) 程序的潜在错误

Halstead 度量可以用来预测程序中的错误。认为程序中可能存在的差错应与程序的容量成正比。因而预测公式为

$$B = (N_1 + N_2) \times \log_2(n_1 + n_2) / 3000.$$

其中, B 表示该程序的错误数。

例如, 一个程序对 75 个数据库项共访问 1300 次, 对 150 个运算符共使用了 1200 次, 那么预测该程序的错误数:

$$B = (1300 + 1200) \times \log_2(75 + 150) / 3000 = 6.5.$$

即预测该程序中可能包含 6 ~ 7 个错误。

7) Halstead 的重要结论之一是: 程序的实际 Halstead 长度 N 可以由词汇表 n 算出。即使程序还未编制完成, 也能预先算出程序的实际 Halstead 长度 N , 虽然它没有明确指出程序中到底有多少个语句。这个结论非常有用。经过多次验证, 预测的 Halstead 长度与实际的 Halstead 长度是非常接近的。

Halstead 度量是目前最好的度量方法。但它也有缺点:

没有区别自己编的程序与别人编的程序。这是与实际经验相违背的。这时应将外部调用乘上一个大于 1 的常数 K_f (应在 1 ~ 5 之间, 它与文档资料的清晰度有关)。

没有考虑非执行语句。补救办法: 在统计 n_1, n_2, N_1, N_2 时, 可以把非执行语句中出现的运算对象, 运算符统计在内。

在允许混合运算的语言中, 每种运算符必须与它的运算对象相关。如果一种语言有整型、实型、双精度型三种不同类型的运算对象, 则任何一种基本算术运算符(+、-、 \times 、/) 实际上代表了 $A_3^2 = 6$ 种运算符。如果语言中有 4 种不同类型的算术运算对象, 那么每一种基本算术运算符实际上代表了 $A_4^2 = 12$ 种运算符。在计算时应考虑这种因数据类型而引起差异的情况。

第 6 章 面向对象技术

面向对象技术是一个有全新概念的开发模式,其特点是:(1) 它是对软件开发过程所有阶段进行综合考虑而得到的;(2) 从生存期的一个阶段到下一个阶段所使用的方法与技术具有高度的连续性;(3) 它将面向对象分析(OOA)、面向对象设计(OOD)、面向对象程序设计(OOP)集成在一起。

6.1 面向对象的概念

1) 什么是面向对象

什么是面向对象? Coad 和 Yourdon 为此下了一个定义:面向对象= 对象+ 类+ 继承+ 通信。如果一个软件系统是使用这样 4 个概念设计和实现的,则认为这个软件系统是面向对象的。一个面向对象的程序的每一组成部分都是对象,计算是通过建立新的对象和对象之间的通信来执行的。

2) 对象(object)

通常将对象定义为它本身的一组属性和它可执行的一组操作。属性一般只能通过执行对象的操作来改变。操作又称为方法、服务或成员函数,它描述了对象执行的功能,若通过消息传递,还可以为其它对象使用。这里,消息是一个对象与另一个对象的通信单元,是要求某个对象执行类中定义的某个操作的规格说明。发送给一个对象的消息定义了一个操作名和一个参数表(可能是空的),并指定某一个对象。而由一个对象接收的消息则调用消息中指定的操作,并将形式参数与参数表中相应的值结合起来。

例如,在计算机屏幕上画多边形。每个多边形是一个由有序顶点集定义的对象。顶点集定义了一个多边形对象的状态,包括它的形状和它在屏幕上的位置,如图 6.1 所示。在多边形上的操作包括 draw, move, contains?

对象不仅仅是物理对象,还可以是某一类概念实体的实例。例如,操作系统中的进程、室内照明的等级、在一个特别审判中律师的作用等等都是对象。

“对象”有两个视图,分别表现在设计和实现方面。从设计方面来看,对象是一些模型化的实体,直接对应于现实世界的实体。这个视图把对象看作实体,产生实体的声明、描述实体、包括实体的属性和可以执行的操作。从实现方面来看,一个对象是实际使用的数据结构与操作。两个视图的作用,是把说明与实现分离,对数据结构和相关操作的实现进行封装。

3) 类(class)

类是一组具有相同数据结构和相同操作的对象的集合。类的定义包括一组数据属性和在数据上的一组合法操作,如图 6.2 所示。类定义可以视为一个具有类似特性与共同行为的对象的模板,可用来产生对象。在一个类中,每个对象都是类的实例(instance),它们

图 6.1 多边形对象

都可使用类中提供的函数。为了从类定义中产生对象, 必须有实例建立操作。C++ 定义了一个 new 操作, 可建立一个类的新实例。C++ 还引入了构造函数, 用它在声明一个对象时建立实例。此外, C++ 给出了一个 delete 操作, 可以释放一个对象所用的空间。C++ 还允许每个类定义自己的析构函数, 在撤销一个对象时调用它。

可以把类看作是一个抽象数据类型(ADT)的实现。但如果把类看作是某种概念的一个模型更合适。可以使用类的界面(public)中定义的操作对类的实例进行操作。例如, 发送一个消息 draw 给类 quadrilateral 的一个实例, 将会引发该实例执行它的 draw 操作, draw 操作会画出一个四边形, 并通过该实例内的 point 数据确定图形的位置。

图 6.2 四边形的类定义

类的实现常使用其它类的实例, 以得到该类所需要的服务。在图 6.2 所示的 quadrilateral 的例子中, 定义类 quadrilateral 的某一个对象所引用的 4 个顶点是 4 个 point 对象。这些 point 对象将成为 quadrilateral 对象的私有数据成员, 不能被其它对象存取。如果其它对象必须存取这些点, 需要在类的界面上增加存取这些点的操作。

类的实现还可能包括某些私有(private)操作和私有数据, 只有该类的对象可以使用, 而其它任何类的对象都不能使用。例如, 可以有一个私有操作, 从一系列的点中找出最靠近(0, 0)的点, 这个操作对其它类的对象将是隐蔽的。

类,就其是一个数据值的集合的意义上看,它与 PASCAL 中的记录或 C 中的结构类似,但又有差别。类可提供几种级别的访问。就是说,记录的某些成分可能是不可访问的。类不同于记录,因为它们包括了操作的定义。

4) 继承(inheritance)

继承是使用已存在的定义作为基础建立新定义的技术。新类的定义可以是既存类所声明的数据,再加上新类增加的数据的组合。新类也可以不加修改地复用既存类的定义。因为既存类已经实现和测试,故开发费用较少。在继承结构中,既存类是父类(基类),新类是子类(派生类)。例如,定义如图 6.2 所示的 quadrilateral 类时,如果 polygon 类已经存在,则 quadrilateral 类可以作为它的子类来定义。图 6.3 (b) 中斜体部分表示在 polygon 类中已经定义,并通过继承加到 quadrilateral 类的定义中。若这些部分作为 polygon 类的一部分已经测试,那么在 quadrilateral 类中就无需像新写的代码那样作严格测试。

图 6.3 类与子类的定义

5) 多态性(polymorphism)

在具有多态性的语言中,在对照形式参数检查实际参数时,实际参数可以是属于若干类型组成的特定集合中的一个类型。图 6.4 给出 4 个类的继承层次。类 quadrilateral 的实

图 6.4 类的继承层次

际参数可代替类 polygon 的形式参数。类 quadrilateral 的界面可响应类 polygon 界面的所有消息。例如,想要在屏幕上画一系列多边形。多态性允许一个对象同时属于几个类型。一个数据结构(如一个表)所包含的元素可以属于一组指定的类型而不仅是一个类型,可以认为这是一个类族,通过对表元素的形式类型规格说明的继承,把这些类关联在一起。可以通过遍历这个表,发送给各个表元素以 draw 消息,“画出”多边形表中所有的项。

6.2 基于复用的开发过程

软件人员根据生存期组织和管理开发过程。为生存期中每个阶段规定一定的任务。

面向对象方法改进了在生存期各个阶段之间的界面,因为在生存期各个阶段开发出来的“部件”都是类。在面向对象生存期的各个阶段对各个类的信息进行细化,类成为分析、设计和实现的基本单元。

6.2.1 应用生存期

传统的生存期是瀑布模型,已广泛用于过程性项目。在这个模型中,从问题的分析开始,直到维护老化,各个阶段细化成许多实际的子处理。由于这个生存期对整个的开发过程进行了模型化,所以称它为应用的生存期。

面向对象开发过程的应用生存期模型如图 6.5 所示。图中各个阶段的顺序是线性的,但开发过程实际上不是线性的。还没有办法用图逼真地反映在面向对象开发过程中各个阶段之间的复杂的相互关系。有一部分分析工作要在设计之前进行,但有些分析工作要与其它部分的设计与实现并行地进行。实际的相互作用更复杂。

使用传统的应用生存期(如瀑布模型)时存在的问题是它没有考虑超出一个单独的项目的情形,也没有考虑任何比整个系统更小的“产品”。软件开发经济学强调软件部件的复用。因此,必须把开发可复用的软件部件作为系统开发过程的一部分。

面向对象把类当作单元,而且可分别考虑类的生存期与应用生存期,它可独立于应用生存期操作。这个生存期可包含在图 6.5 所示的类开发阶段中,可与应用生存期集成。

图 6.5 面向对象开发过程的应用生存期模型

6.2.2 类生存期

系统开发的各个阶段都可能会标识新的类。但在各个阶段标识的类所起的作用是不

同的。在分析阶段标识的类是针对特定问题的,而在实现阶段标识的类要给出基本的数据结构,这些数据结构可在许多系统中使用。

图 6.6 给出类的生存期模型。随着各个新类的标识,类生存期引导开发工作逐个阶段循序渐进。如在应用分析时标识了对一个“图形显示设备”的要求。如果这样一个“图形显示设备”类不存在,就应着手开发。由于要求把显示器上的所有可能操作全都用上的系统很少见。因此,若把操作的开发当作一个特定应用开发的一部分,只有为当前系统所要求的那些操作才会被标识和实现。但若考虑让部件独立于系统,就必须能够综合出超出当前系统需求的开发要求。这要求生成能表示成一个完全的概念模型并可复用的类。

图 6.6 类的生存期模型

在纯面向对象的系统开发中,一个系统是一个类,为解决问题所需要的所有元素都放在一个类中,并通过该类的一个实例来解决问题。

1) 类的规格说明

类的规格说明是要标识类并给出它的规格说明,包括类的实例可执行的操作和它们的数据表示。类的界面包括了该类的对象所有可为其它类所使用的行为。若改变一个类的界面,将会影响所有其它使用这个类的类。

(1) 既存类的复用

有这样一些类,它们被选用来提供所需要的行为。有时应用要解决的问题与以前遇到的一些问题密切相关,因此那些问题中定义和实现的类可以复用。然而,可以原封不动地复用的既存类多为低层上最基本的类,像基本数据结构。对于较通用的结构,可以利用参数规定它们的行为。

(2) 从既存类进行演化

多数复用情况是一个类已经存在,它提供的行为类似于要为新类定义的行为。开发人员可以使用既存类作为定义新类的起点。新类将根据既存类渐进式地演变而成。

演化可以是横向的,也可以是纵向的。横向的演化导致既存类的一个新的版本;纵向的演化从既存类导出新类。

(3) 从废弃型进行开发

这个生存期的分支仅在不得已的情况下使用。任何一个类, 只要它的开发不涉及既存类, 就可看作是一个新的继承结构的开始。因此, 将建立两种类: 一是抽象类, 它概括了将要表达的概念; 另一是具体类, 它要实现这个概念。

2) 设计

把应用生存期的分析阶段产生的结果当作输入, 确定类的其它属性, 给出类的所有细节。在设计阶段的输出是有关类的属性的足够的细节, 可支持它们的实现。单个类的设计包括构造类的数据存储结构, 可以由基本对象(如整数和实数)和复合对象(由其它对象组成)组成。内部表示还可以包括一些私有函数, 它们实现了共有操作的某些细节。单个类的低层设计涉及一些重要联系, 如继承和组装。

(1) is part of (组装关系)

这个关系指明, 一个实体表示的一部分可以由其它实体给出。新类的实现可以有效地使用既存类的行为。例如, 一个 `displayable shape` 类中。每个形状有一个基准点, 因此, `displayable shape` 类表示的一个部分可以通过一个 `point` 类的实例给出。我们称 `point` 类是 `shape` 类表示中的一个部分。定义实例建立的联系:

```
class point is part of class displayable shape
```

(2) is a (继承关系)

这个关系使用一个类的既存定义作为一个新类定义的一部分, 在两个类间的“is a”联系表征了一个类是另一个类的特殊情形。这种联系将一个既存类的定义和一个新类的特殊部分定义组织成为一个完整的类定义。

考虑 `triangle` 类。类 `triangle` 的某些行为是类 `displayable shape` 的一部分。只要在类 `displayable shape` 的共有定义上增加新类的定义, 就可以建立 `triangle` 的定义。这样就建立了“`triangle is a displayable shape`”的联系。

3) 实现

通过实例的声明、操作界面的实现及支持界面操作的函数的实现, 可实现一个类的行为和状态。

在实例中存储的数据通常是另一些类的实例, 它们提供了为该类的开发所需的服务。例如, 散列表(`hash table`)类的表示可以包括一个数组类的实例(作为散列表项的存放空间)、一个整数(记录当前在表中已有的项数)及一个散列函数。此外, 还包括如 `storeItem`(存储一项)和 `retrieveItem`(检索一项)等各个界面操作的实现。

4) 测试

单个类为测试提供了自然的单元。如果类的定义提供的界面比较狭窄, 那么穷举测试就有可能实现。类的测试从最抽象的层次开始, 沿继承关系自顶向下进行, 新类可以完全测试, 而已测试的部分无需从新测试。

5) 求精和维护

概念的封装和实现的隐蔽, 使得类具有更大的独立性。在任一时刻都可以在类的界面上增加新的操作, 并能够修改实现, 以改进性能, 或引入原来设计中没有的新服务。为便于类的调整, 应尽量作到定义与实现分离。

6.3 面向对象分析与模型化

面向对象分析过程分为论域分析和应用分析。论域分析建立大致的系统实现环境,应用分析则根据特定系统的需求进行论域分析。

6.3.1 面向对象分析 (OOA, object-oriented analysis)

面向对象分析是软件开发过程中的问题定义阶段。这一阶段最后得到的是对问题论域的清晰、精确的定义。传统的系统分析产生一组过程性的文档,定义目标系统功能。面向对象分析则产生一种描述系统功能和问题论域的基本特征的综合文档。

面向对象分析文档的视点不同于过程性分析的文档。传统的文档面向功能,其视点是把系统看作一组功能。面向对象的分析文档把问题当作一组互相作用的实体,并确定这些实体之间的关系。在这种描述开发出来之后,分析员就能够很好地建立特定系统的系统规格说明。

6.3.2 论域分析 (domain analysis)

论域分析的主要思想是想把考虑的领域放宽一些,把相关的概念都标识到,以帮助更好地掌握应用的核心知识。当用户改变他们对系统的需求时,范围广泛的分析可以帮助预测这些变化。分析由小组进行,其成员包括有关知识领域的专家和分析员等。分析的目的是标识基本概念,识别论域的特征,把这些概念集成到论域模型中。

1) 语义数据模型 (semantic data models)

论域分析的输出和对应用分析和高层设计的输入构成问题论域模型。已有许多建立这种模型的技术,一种特别适用的技术是语义数据模型。

语义数据模型来源于 Codd 的关系数据模型和实体-联系模型,并对这类模型进行了扩充。语义数据模型可以表达问题论域的内涵,还可以表示复杂的对象和对象之间的联系。

作为数据库结构标准的 ANSI/SPARC 建议提出了三层模型:外部模型、概念模型和内部模型。这三层可以被映射到面向对象设计的三个层次上去,如表 6.1 所示。外部模型与概念模型层相当于高层设计阶段。

表 6.1 语义数据模型与面向对象视图

语义数据模型	主 要 特 征	面向对象设计
外部模型	数据的用户视图	类的定义(规格说明)
概念模型	实体及其之间联系的内涵	类与类之间的应用级联系
内部模型	数据的物理模型	类的实现

外部模型:外部模型层是来自系统外部的现实世界的视图,它反映了用户对问题的理解而不是实现者对问题的理解。在这一层开发的类应具有对应于用户活动的操作规格

说明。

概念模型: 概念模型层考虑外部模型层所标识的实体之间的联系。这些联系是可直接观察到的交互关系。联系的重要属性是实例连接。一个联系的实例连接是指在该联系中一个实体的实例对应于该联系中其它实体的实例的数目。

内部模型: 这一层考虑实体的物理模型, 这是生存期中的类设计阶段。物理模型包括两类属性: 数据和方法。方法属性对实体的行为模型化, 而数据属性对实体的状态模型化。在模型中方法分为两种: 一种可作为共有界面来使用, 而另一种是私有的。

在分析阶段所标识的属性是描述性的, 它们提供了各个实体的行为和状态的描述。另外, 在详细的类设计阶段, 还要加入一些追加属性来帮助描述属性, 从而提供类的实现。这些追加属性是类定义实现的一部分, 它们不能用于公共存取。

2) 在语义数据模型中的关系

语义数据模型中的一般联系在面向对象设计中起着核心作用。通过它们给出了继承和实例化等概念的表达。表 6.2 给出了在语义数据模型中联系的类型。

表 6.2 在语义数据模型中联系的类型

联 系	描 述
泛 化	一个实体表现了在其它几个实体背后所具有的概念的共性
聚 合	一个实体由其它几个实体的实例组成
分 类	一个实体是另一个实体的实例
关 联	一个实体是另一些实体的包容(container)

泛化联系: 泛化和它的逆(即特化) 可用来按层次渐增式地定义抽象。低层抽象是高层抽象的特化。这种抽象层次构成了论域模型的基础。一旦标识了高层抽象, 模型就可逐步展开。例如, “ 小汽车 ”, “ 卡车 ”, “ 公共汽车 ” 可以蕴含在更一般的概念“ 汽车 ” 中。这个较泛化的抽象还可以帮助定义其它比较特殊的抽象, 如“ 赛车 ”, “ 面包车 ” 和“ 牵引车 ”。

聚合联系: 这个联系支持从几个较小的和较简单的元素来开发一个抽象表示。它大略相应于一个记录中成分的声明。例如, 一个航班可以被认为有 6 个属性: 飞机编号、机组编号、离开和到达地点、起飞和降落时间。因此, 航班类有一个聚合联系, 它利用了表示飞机、人员、空间的类, 并增加了时间窗口。

分类联系: 在系统的设计中出现了一个类的各个实例间有共享信息时, 常常要用到分类联系。在开发时, 面临的一个问题是要否开发子类, 以表达在某些概念上的差别。例如, 已给出“ 学生 ” 这样一个类, 那么是否应当建立“ 男学生 ” 和“ 女学生 ” 子类呢?

关联联系: 关联也称为组装, 它指定一个抽象作为其它抽象实例的包容(container)。关联和聚合之间的差别在于组合实体的意图。聚合指定一组实体中的某些部分作为一个类的组成, 而关联是指群集的相互有关联的实体群。这个群集中每个分量都是独立地用在系统的其余地方。例如, 一个部门包含有人, 这样一个部门关联了所有被分配给这个部门的人。

3) 标识对象和类

在分析阶段所标识的对象是应用级的对象,即它们对应于应用论域的用户视图。由它们产生的类是对用户视图的模型而不是对物理数据结构的模型化。例如,计算机显示屏幕是一个对象,它给出用户界面部分。它被抽象成一个类“显示器屏幕”,以反映用户的想法。以后将标识的实现级的类可以是一个特殊的位图存储结构,即“位图”类。

4) 标识对象

在面向对象设计中一个重要的任务就是标识正确的对象,并从中抽象出适当的类。对象可以分为5种:物理对象,角色,偶发事件,交互,规格说明。每个系统可以具有某几种或所有各种对象,但也不必特意为每个对象进行分类。

物理对象:物理对象是最容易和最明显的可识别的实体,它们通常可以在问题论域的陈述中直接找到,而且它们的属性可以标识和测量。用户要求、蓝图和设备清单都能提供对象描述信息。例如,大学的选课系统中包含表示选课学生的对象;网络管理系统中包括的各种对应于网络物理资源的对象,如开关、CPU和打印机。

偶发事件对象:一个偶发事件是某个活动的一次出现。一个偶发事件对象通常是一个数据实体,它管理围绕“出现”的重要信息。偶发事件对象的操作主要用于提供对数据的存取。例如,“航班”是在一个空中交通控制系统中的一个偶发事件,这个事件对象保持诸如航班号、目的地、乘客数目和有关飞机的信息。

交互对象:交互表示了在两个对象之间的联系,这种类型的对象类似于在数据库设计时所涉及的连接实体。当实例连接是多对多的时候,就需要这样的实体。交互对象把第一个类的一个特定实例关联到第二个类的一个特定实例。交互对象的数据包括用来定位将要关联的两个实例的信息,还包括表征联系的数据。例如,在学生和课程之间的实例连接是多对多的联系,一个选课系统包括许多选课对象的实例,这些对象把一个单独的学生对应到一个单独的课程,并且自身包含表明一学期学习情况的数据。

规格说明对象:规格说明对象表明对某些实体的需求。规格说明对象中的操作提供一部分选定的结构的视图,支持把简单的对象合并成较复杂的对象。规格说明通常参照其它的对象。规格说明的数据部分将用这些其它对象来标识。例如,一个自动的硬件测试系统将包括表示各个部分电路的简图的对象。在开发测试计划和实际测试时,将存取这些对象作为参考。一个烹饪对象定义调料和它们的量以及它们组合的次序和方式。一个颜色对象将定义在颜色监控器上建立颜色所要求的红、绿、蓝的级别。

5) 标识联系

在分析阶段,要被标识的联系是特定应用的,这些联系涉及问题论域中实体之间的相互作用。在设计中,应用级联系用以下两种方式中的一种来表示。

消息连接——两个对象之间的通信联系可以表示为两个对象之间的消息。当学生希望选一门课时,这个学生对象将发送一个消息给选课对象,请求一个课程号。

交互对象实例连接——两个对象之间的联系可以通过一个不依赖于两个对象中的任一方的交互对象来表示。在大学选课系统中,在学生类的一个实例和课程类的一个实例之间的联系可用一个交互实体来处理,这个交互实体称为选课。

6.3.3 应用分析(application analysis)

应用分析的依据是在论域分析时建立起来的问题论域模型,并把它用于当前正在建立的系统当中。客户对系统的需求可以当作限制来使用,用它们缩减论域的信息量。论域分析产生的模型并不需要用任何基于计算机系统的程序设计语言来表示,而应用分析阶段产生影响的条件则伴随着某种基于计算机系统的程序设计语言来表示。响应时间需求、用户界面需求和某些特殊的需求,如数据安全等,都在这一层分解抽出。通常我们着重考虑两个方面:应用视图和类视图。必须对每个类的规格说明和操作详细化,还必须对形成系统结构的类之间的相互作用加以表示。

6.3.4 对象模型技术(OMT, object model tech.)

Rumbaugh 等人提出对象模型技术,它把分析时收集的信息构造在三类模型中,即对象模型、功能模型和动态模型。这三个模型的建立次序如图 6.7 所示。

图 6.7 对象模型、功能模型和动态模型

应当注意的是,从功能模型回到对象模型的箭头表明,这个模型化的过程是一个迭代的过程。每一次迭代都将对这三个模型作进一步的检验、细化和充实。

1) 对象模型

对象模型是三个模型中最关键的一个模型,它的作用是描述系统的静态结构,包括构成系统的类和对象,它们的属性和操作,以及它们之间的联系。事实上,对象模型与 E-R 模型十分相似(当然,也可以直接利用扩展了的 E-R 图)。它可以不加限制地表示实体或联系的类型,各种以问题为中心的联系可以很自然地处理成继承联系。

OMT 为每一个模型都提供了图形表示。图 6.8 给出了类与对象的表示方法。在 OMT 中,类与类之间的联系叫作关联。关联代表一组存在于两个或多个对象之间的、具有相同结构和含义的具体连接。关联可以是物理的,也可以是逻辑的。图 6.9 给出了聚合联系,它代表整体与部分的关系。上面图中的右例以聚合图形符号说明“段落是由多个句子组成”。表示不同数量的符号,如实心圆表示“多个”,空心圆表示“0 个”,不加圆表示“1 个”。中间的图是限定,用以对关联的含义作某种约束。从中间的图右例子中可看到,附加的限定词在说明“一个目录包含多个文件”的基础上,更明确地指明“每一个文件都可由目录中的文件名属性唯一地标识”。下面的图是角色,用来说明关联的一端。由于多数关联具有两个端点,因而涉及到两个角色。就下面的图右的例子而言,在公司与个人的关联中,公司的角色是雇主,个人的角色是雇员。

说明对象之间连接的属性称为连接属性。再看图 6.9 中的例子,由于一个人在成为公司的雇员时才有工资和职务。因此,工资和职务不是个人的属性,而是公司与个人之间的连接属性。OMT 指出,这种情况常常意味着有必要定义一个新类来取而代之。

图 6.8 类与对象的表示

图 6.9 关联的表示方法和实例

OMT 定义了泛化关系(也称为继承性),如图 6.10 所示。泛化关系通常包含一个(如果允许多重继承,也可以是几个)基类和几个子类。基类表示了一个较为一般、普遍的概念,而每个子类则是它的某个特殊形态。因而子类除了自然地继承基类所具有的属性和操作外,还具有反映自身特点的属性和操作。泛化关系有助于代码共享和复用。

图 6.10 泛化关系(继承性)的表示方法

2) 动态模型

时序联系是很难掌握的。要想对一个系统了解得比较清楚,首先应当考察它的静态结

构,即在某一时刻它的对象和这些对象之间相互联系的结构;然后应当考察在任何时刻对对象及其联系的变化。系统的这些涉及时序和变化的状况,用动态模型来描述。动态模型着重于系统的控制逻辑。它包括两个图,一是状态图,一是事件追踪图。

(1) 状态图

状态图是一个状态和事件的网络,侧重于描述每一类对象的动态行为。在如图 6. 11 所示的状态图中,状态是对某一时刻中属性特征的概括。而状态迁移则表示这一类对象在何时对系统内外发生的哪些事件作出何种响应。图中的椭圆表示状态,状态之间的箭头表示从一个状态到另一个状态的迁移,附加在箭头上的短语说明触发此状态迁移的事件。图中,“事件 A ”是一个单纯事件,而“事件 B[条件] ”是一个条件事件,在给定条件满足时才起作用。

OMT 区分两种不同的行为,即操作和活动。操作是一个伴随状态迁移的瞬时发生的行为,与触发事件一起表示在有关的状态迁移之上。活动则是发生在某个状态中的行为,往往需要一定的时间来完成,因此与状态名一起出现在有关的状态之中。状态图中所有这些成份都可根据具体要求而予以取舍。

在一个系统中,对一个事件的响应依赖于接收它的对象的状态,它可以包括状态的变化、发送另一个事件给原来的发送者或第三个对象。对于一给定类的事件、状态及状态的迁移,可以抽象并表示成一个状态图。

图 6. 11 状态图

动态模型由多个状态图组成,对于每一个具有重要动态行为的类都有一个状态图,从而表明整个系统活动的模式。各个状态图并发地执行,并可以独立地改变状态。对于各种类的状态图可以通过共享事件组合到一个动态模型中。

下面特别对事件加以说明。一个事件发生在某一时刻,例如“按下左按钮”或“21 次列车离开北京站”,事件没有持续时间。一个事件在逻辑上可能领先于另一个事件,或者两者没有关系。21 次列车到达上海之前必须离开北京,这两个事件有因果关系;而 12 次列车离开沈阳和 38 次列车离开武昌,这两个事件可能没有因果关系。两个没有因果关系的事件叫作并发事件,它们之间互相不影响。

在模型化一个系统时,一般不考虑并发事件的次序,因为它们可能以任一次序发生。一个分布系统的真实模型必须包括并发事件和活动。每个事件都是单独发生的,我们把它们纳入事件类中,并给每个事件一个名字,以指明共同结构和行为。这种结构是分层的。例如,“列车出发”有属性“线路”、“班次”和“城市”。对于所有的事件来说,事件发生的时间是隐蔽的。

事件从一个对象向另一个对象传送信息。有些事件类可能传送的是简单的信号“要发生某件事”,而其它事件类则可能传送的是数据值。由事件传送的数据值叫作属性,属性可

以在事件类名之后用括号列出。如图 6.12 所示。

列车出发(线路、班次、城市)
按下鼠标按钮(按钮、位置)
拿起电话受话器
数字拨号(数字)

图 6.12 事件及其属性的表示

(2) 事件追踪图

事件追踪图侧重于说明发生于系统执行过程中的一个特定“ 场景（scenarios）”。场景也叫脚本, 是完成系统某个功能的一个事件序列。

场景通常起始于一个系统外部的输入事件, 结束于一个系统外部的输出事件, 它可以包括发生在这个期间的系统所有的内部事件, 也可以只包括那些撞到的或由系统中某些对象生成的事件。场景可以是系统执行的历史记录, 还可以是执行一个拟议中系统的设想实验。图 6.13 给出了使用电话的一个场景图。这个场景图只包含影响电话的事件。从场景图中可以看出, 每个事件从一个对象向另一个对象传送信息。例如, “ 电话忙音开始 ” 从电话传递了一个信号给打电话者。

1 打电话者拿起电话受话器	10 铃声在打电话者的电话上传出
2 电话忙音开始	11 接电话者回答
3 打电话者拨数字(8)	12 接电话者的电话停止振铃
4 电话忙音结束	13 铃声在打电话者的电话中消失
5 打电话者拨数字(2)	14 通电话
6 打电话者拨数字(3)	15 接电话者挂断电话
7 打电话者拨数字(7)	16 电话切断
8 打电话者拨数字(3)	17 打电话者挂断电话
9 接电话者的电话开始振铃	

图 6.13 使用电话的一个场景图

在写出这个场景图之后, 下一步是要标识每个事件的发送者对象和接收者对象。各种有关事件的序列关系以及由此而表现出来的对象之间的交互作用可以通过如图 6.14 所示的事件追踪图表达。在这个图上, 竖线表示对象, 带箭头的横线表示事件, 箭头从发送者对象指向接收者对象。时间自上向下延续, 与间隔的空间无关。这只是事件序列的表示, 没有精确的时序。此外, 并发事件也可以发送。例如, 电话线并发地发送事件给打电话者与接电话者。

概括地讲, 状态图叙述一个对象的个体行为, 而事件追踪图则给出多个对象所表现出来的集体行为。它们从不同侧面说明同一系统的行为, 因此必然存在着密切的内在联系。例如, 一个事件追踪图指出某一对象在接受一个事件之后发出另一事件, 同一行为在此对象的状态图中也应当有所表示。这种内在的联系实际上是一致性检验的基本依据。

图 6.14 打电话的事件追踪图

另外, 由于状态图的复杂程度会随着应用问题的增大而呈指数型迅速增加, 这使得分析和验证十分困难。针对这个问题, OMT 采用了一种结构化的方法, 可以将一个复杂状态分解为一些比较简单的子状态, 以前者对系统或系统的某一部分作高度的抽象和概括, 同时以后者提供在前者中省略掉的有关细节。这样, 不仅保证了对系统行为的充分说明, 而且有效地限制了状态图本身的复杂性。

3) 功能模型

功能模型着重于系统内部数据的传送和处理。它是模型化三角架的第三只脚。功能模型定义“做什么”, 动态模型定义“何时做”, 对象模型定义“对谁做”。

功能模型表明, 通过计算, 从输入数据能得到什么样的输出数据, 不考虑参加计算的数据按什么时序执行。功能模型由多个数据流图组成, 它们指明从外部输入, 通过操作和内部存储, 直到外部输出的整个的数据流情况。功能模型还包括了对象模型内部数据间的限制。数据流图不指出控制或对象的结构信息, 它们包含在动态模型和对象模型中。

图 6.15 中的实线表示数据流, 虚线表示控制流, 圆框代表处理数据的过程, 矩形框表

图 6.15 数据流图的表示方法

示产生与接收数据的对象, 平行线框表示数据存储区。它基本上是传统的数据流图加上控

制流。但 OMT 指出, 虽然控制流有时是有用的, 但它重复了动态模型中的有关部分, 因而应尽量少用。

功能模型中所有的数据流图往往形成一个层次结构。在这个层次结构中, 一个数据流图中的过程可以由下一层的数据流图作进一步的说明。一般来讲, 高层的过程相应于作用在聚合对象上的操作, 而低层的过程则代表作用于一个简单对象上的操作。

6.4 高层设计

通常, 面向对象设计分为两个阶段, 即高层设计和低层设计。高层设计包括开发像用户界面那样的问题解决部分。低层设计集中于类的详细设计阶段。高层设计阶段开发系统的结构, 即用来构造系统的总的模型, 并把任务分配给系统的各个子系统。

面向对象系统的一个典型的系统结构是如图 6.16 所示的模型/视图/控制程序 (MVC) 框架结构。其中, 模型是指应用中的对象, 它的操作独立于用户界面; 视图则管理用户界面的输出; 控制器处理应用的输入。

图 6.16 模型/视图/控制程序(MVC) 框架结构
(Model/View/Controller)

输入事件给出要发送给模型的消息。一旦模型改变了它的状态, 就立即通过关联机制通知视图, 让视图刷新显示。这个关联机制定义了模型与各个视图之间的关系, 它允许模型的运行独立于与它相关联的视图。控制器在一个输入事件发生时将对视图及模型进行操作。此外, 还使用了许多其它配置。

系统结构实际上是一些抽象类的集合, 提供了系统的最上层设计。定义这些抽象类的新的子类和把它们实例化可以实现所要求的特定行为, 籍此构造待开发的系统。

对于 MVC 来说, 可以通过开发模型的一个子类, 履行与应用相关联的处理, 建立这个系统软件。用户界面通过定义视图和控制程序子类来建立, 这些子类许多是可复用的, 像按钮和对话框等在多数系统中已经存在。这样就导致了新子系统的建立。

类与具有概念封装的子系统十分类似。事实上, 每个子系统都可以当做一个类来实现, 高层设计标识在计算机环境中进行问题解决工作所需要的概念, 并增加了一批需要的类。这些类包括那些可使软件与系统的外部世界交互的类, 包括与其它系统(如数据库管理系统、鼠标和键盘)的界面, 与用来进行数据收集或者负责控制的硬件设备的界面等。此阶段的输出是适合应用要求的类、类间的联系以及应用的子系统视图规格说明。

高层设计可以表征为标识和定义模块的过程。模块可以是一个单个的类, 也可以是由一些类组合成的子系统。定义过程是职责驱动的。

图 6.17 绘图系统的子系统结构

一个典型的绘图系统结构如图 6.17 所示。用户界面处理与用户的所有交互活动,它可以是一个事件驱动的系统,响应鼠标按钮或键盘击键,也可以是一个简单的轮询系统。不论是何种方式,对其它两个子系统的设计都没有影响。图形引擎负责产生各种图形对象、利用来自用户的信息考虑对象的大小和位置。在一个较为复杂的系统中,图形引擎还用来提供有关对象的几何信息。永久存储系统把图形对象存储在永久存储中。

6.5 类 的 设 计

在分析与高层设计阶段标识一个“概念”时,常常需要使用多个类来表示,通常把一个类看作一个模块,并且需要对一个概念进行分解,用一组类来表示这个概念。一个独立的类可只表示一个概念。

在类的文档中应当对类的用途作出清楚的标识和精确的陈述,在类的共有界面上定义操作时,需要定义每个操作的特征、先决条件和后置条件。这种完善的规格说明可为以后的基于规格说明的测试提供足够详细的细节。

6.5.1 通过复用设计类

利用既存类来设计类,有 4 种方式:选择、分解、配置和演变。这是面向对象技术的一个重要优点。许多类的设计都是基于既存类的复用。

1) 选择

最简单的设计一个类的方法,是从既存的部件中简单地选择合乎需要的软件部件。这是开发软件库的目的。一个面向对象开发环境常常提供一个常用部件库。特别在许多语言环境中都带有一个原始部件库,如整数、实数和字符,它是提供其它所有功能的基础层。任一基本部件库(如“基本数据结构”部件)都应建立在这些原始层上。这些都是些易于作成一般的和可复用的类,例如,列表、集合、栈和队列等。这个层还包括一组提供其它应用领域服务的一般类,如窗口系统和图形图元。

2) 分解

最初标识的“类”常常是几个概念的组合。在着手设计时,可能会发现所标识的操作散落在分散的几个概念中,或者数据属性被分开放到模型中而拆散概念形成的几个组内。这样必须把一个类分成几个类,希望新标识的类容易实现,或者它们已经存在。

3) 配置

在设计类时,可能需要由既存类的实例提供类的某些特性。通过把相应类的实例声明为新类的属性来配置新类。例如,仿真服务器可能要求使用一个计时器来跟踪服务时间。设计者不必开发在这个行为中所需的数据和操作,而是应当找到计时器类,并在服务器类的定义中声明它。这个服务器还要求有一个存储管理器,在客户到达和等待服务时存放它们。为此,需要一个队列类的实例来作客户排队工作。对每一个客户的服务时间由一个已知的概率分布来确定,因此,可使用一个具有泊松分布的随机变量的类的实例,具有均匀分布的随机变量的类的实例。

4) 演化

要求开发的新类可能与一个既存类非常类似,但不是完全相同。不适宜采用“选择”操作。可以从一个既存类演化成一个新类。利用继承机制表示泛化/特化联系。特化处理有三种可能的方式,如图 6.18 所示。

图 6.18 泛化/特化联系的方式

特化运算可以从该既存类的定义产生新类的初始构造,这是典型的类继承的使用。既存类 A 的数据结构和操作可以成为新类 B 的一部分,如图 6.18 (a) 所示。如果新概念是一个既存类所表示概念的一个特殊情况,既存类 A 的公共操作成为新类 B 的共有界面部分。

如果新类比软件库中那些既存类更一般,则新类 B 不具有既存类 A 的全部特性,泛化运算把两个类中共同的特性移到新的更高层的类中,高一层的类是将要设计的 B。原来的类 A 成为新类 B 的子类,如图 6.18 (b) 所示。一个既存类 A 与设计的新类 B 共享概念的某一个部分,则两个概念的共同部分形成新类的基础,且既存类与新类两者成为子类,如图 6.18 (c) 所示。后两种涉及既存类的修改。既存类中定义的操作或数据被移到新类中。遵循信息隐蔽的原理,这种移动应不影响已有的使用这些类的应用。类的界面保持一致。

6.5.2 类设计的方针

这里讨论的类设计的方针是类的模块设计的方针。

1) 信息隐蔽

软件设计通过信息隐蔽可增强抽象,并可保护抽象数据类型的存储表示,不被抽象数据类型实例的用户直接存取。对其表示的唯一存取途径只能是界面,对类也是如此。

例如,在设计 point 类时,应保护作为点坐标的 x, y 不被直接存取。当不作这样的保护时,对存储表示的改变可能导致所有对数据值的引用出错,如图 6.19 (a) 所示。在另一个 point 类的设计中。对数据的存取全部通过共有界面上的操作,对该类的用户屏蔽把存

储表示的改变,如图 6. 19 (b)所示。

2) 消息限制

类的设计者应当为类的命令设计一个显式的界面,该类实例的用户应当只使用界面提供的操作。

3) 狭窄界面

当且仅当一个操作对类的实例的用户有用时,它才是类公共界面的一个成员。对于上面所讨论的 hash table 类,界面应包括插入和检索表的操作,而不应包括用一项的关键字值计算散列值的操作。散列函数不应由类的实例的用户来访问。它应是一个独立的操作,以便容易调整或改变散列函数,且它应是隐蔽实现的部分。

图 6. 19 Point 类的两种设计方案

4) 强内聚

模块内部的各个结构应有较强的联系,即它们不可能分别标识。在面向对象系统中,类的某些操作应是简单的存取函数,它们简单地返回某些属性的值。对于维持类的信息隐蔽特性,它们是十分重要的,因为它们隐蔽了属性的实际物理表示。

5) 弱耦合

一个模块应尽量不依赖于其它模块。如果类 A 的实例建立类 B 的局部实例;或者如果类 A 的操作需要类 B 的实例作为参数;或者如果类 A 是类 B 的一个子类,则称类 A

“依赖于”类 B。模块之间的耦合程度部分依赖于所使用的分解方法。类 A 之所以依赖于类 B,是因为类 A 要求类 B 提供服务。这个依赖性可通过复制类 A 中的类 B 的功能来消除。但代码的复制减少了系统的灵活性并增加了维护的困难。

继承结构损害了弱耦合的概念。因为在建立泛化-特化联系的时候,继承引入了依赖。

6) 显式信息传递

除了依赖于最少的类外,还应该明确在这些类之间的信息流。在类之间全局变量的共享隐含了信息的传递,并且是一种依赖形式。两个类之间的交互应当仅涉及显式信息传递,即通过参数表传递信息。

显式信息传递要最小化,要求在实例间传递大量数据,常常表明两个紧密相关的类的分解不正确。

7) 子类当作子类型

至少有两种方式可以用类帮助开发子类:由既存类提供子类实现的服务;或提供类的共有界面使之成为子类界面部分。

如果父类是共有的,则其共有界面将成为新的子类的共有界面部分,这表明父类的行为成为子类的行为部分。这类似于类型与子类型之间的关系。如果父类是私有的,它的行为将不是继承类的公共行为部分而是实现部分。它的提出是为了提供实现新类的服务。使用有关实现的继承将使得实现类的内部改变和复用类变得很困难。因此,每个子类应该当作父类的特化来开发,父类所具有的公共界面可成为子类的共有界面的一个子集。

8) 抽象类

某些语言提供了一个类,用它作为继承结构的开始点,所有用户定义的类都直接或间接以这个类为父类。这个抽象模型生成一个类,不用它产生实例。它定义了一个最小的共有界面,许多子类可以加到这个界面上以给出概念的一个特定视图。

6.5.3 类设计的过程

类设计与其它设计一样,是三类设计,即数据设计、结构设计和过程设计的组合。通过标识对象和类,建立数据抽象;通过把操作结合到数据中,定义模块,建立软件结构;通过开发有关使用对象的机制,描述界面。

类设计一般可根据类的类型,导出类的描述。类的描述包括三个部分:(1) 属性定义:说明类的数据类型、操作方法等;(2) 该类的共有界面:一般包括继承界面和外部界面;(3) 类实例的可能状态之间的有效变换集。

要根据功能模型及动态模型,以及实际情况设计对象的消息模式。

确定各类之间的继承关系时,将各对象的公共性质放在较上层的类中描述,通过继承而共享对公共性质的描述。类实质上定义的是对象的类型,它描述了它们所有的性质。

类又是一种分层结构,类的上层有父类,下层有子类。子类直接继承父类的全部描述,这也叫作传递性。类可以有多个父类和多个子类,这叫作多重继承;如果只限于一个父类,叫作简单继承,在这种情况下,类的层次结构是树型结构。在实现时,利用继承性可把通用的类和专用的类存储于类库中,根据需要可以复用它们。

下面利用一个例子说明如何进行设计。例子中使用 Smalltalk 语言, 主要目标是定义和表征抽象, 最终定义所有重要的对象、类、操作和消息。

- 1) 对每个子系统标识系统中的对象。标识过程根据需求文档自顶向下地进行。
- 2) 对每个对象标识属性。属性是每个类中的实例变量(操作数据)。还可能需 要其它实例变量以响应来自系统其它对象的请求。而包含属性的数据结构的规格说明则推迟到详细设计阶段。
- 3) 标识每个对象中的操作(或过程)。某些操作存取和更新实例变量, 而其它操作则执行单一的类操作。在此, 不定义操作的实现细节, 只是描述功能。如果新的对象继承了另一个对象, 则需要检查该对象的操作, 看是否有要被新对象取代的操作。操作的内部设计推迟到详细设计阶段, 可以使用传统的设计技术。
- 4) 标识对象之间的通信。这一步定义对象发送给其它每一个对象的消息。此时, 定义操作之间的对应和执行操作的消息。虽然面向对象的实现还没有计划, 但消息可帮助设计通信及可用于下一步写出脚本。要考虑消息命名的一致性, 作出协议。
- 5) 使用场景测试设计。场景由给对象的消息组成。用来测试设计结果是否与系统需求规格说明匹配。

6) 在适当的地方应用继承。如果对象- 类的标识过程是自顶向下执行的, 将引入继承。目标是尽可能复用已经设计出来的数据和操作。在这一步要找出公共数据和操作, 把那些公共实例变量和操作组合到新的类中。这个类作为一个对象, 可能有意义, 可能没有意义。它的单一用途是收集公共实例变量和操作, 故称为抽象类。

作为以上描述的类设计的例子, 考虑一个简单的画图程序, 它是要查看和操纵各种二维图元。该系统的需求简单描述如下。

“该系统允许用户在彩色图形终端上建立和操作二维多边形、样条和二次曲线。使用的图形输入设备有鼠标器等。用户可以对各个图元移动、旋转、缩放和填色。”这是一个含糊的需求定义, 但它包含了足够的信息, 可帮助我们着手设计处理。下面将集中于二次曲线的设计。二次曲线是一个二阶隐式曲线, 形如

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

二次曲线包括圆、椭圆、双曲线和抛物线。

第一趟

- 1) 对每个子系统标识对象类。

系统有三个不同的图元, 但最初要寻找那些对于所有图元都是公共的东西。初始类层次如图 6.20 (a) 所示。图中最高层的类为 geometric object, 它所执行的函数适用于所有的类。primitive(图元)类是一个抽象类, 它的作用是包容所有的为低层的类所使用的数据和操作。

- 2) 对每个对象标识属性。从需求知, 用户将移动、缩放和旋转图元, 这样, 位置、缩放因子、朝向就是合适的实例变量。

- 3) 标识每个对象中的操作。需求文档常常显式陈述操作。系统需求定义了 creat, move, scale 和 rotate 操作。还需要一些操作来存取实例变量。有关图元的操作列表如下。

(a) 类的初始层次 (b) 抽象的第二层

图 6.20 类的层次

create_primitive	建立一个图元
set_position	设置 x, y 位置
get_pisition	取得 x, y 位置
add_position	增加 x, y 位置
set_orientation	设置旋转角度
get_orientation	取得旋转角度
add_orientation	增加旋转角度
set_scale	设置 x, y 缩放因子
get_scale	取得 x, y 缩放因子
add_scale	增加 x, y 缩放因子
set_color	设置颜色
get_color	取得颜色

4) 标识对象之间的通信。通过把消息联结到上面定义的操作上, 定义对象间的协议。

new!	建立一个图元
position=	设置 x, y 位置
pisition?	取得 x, y 位置
position+	增加 x, y 位置
orientation=	设置旋转角度
orientation?	取得旋转角度
orientation+	增加旋转角度
scale=	设置 x, y 缩放因子
scale?	取得 x, y 缩放因子
scale+	增加 x, y 缩放因子
color=	设置颜色
color?	取得颜色

5) 使用场景测试设计。使用一些简单的场景来匹配需求, 展示如何追踪每一个需求。

需 求	脚 本
Create:	primitive new! name= aPrimitive 建立图元

Move:	aPrimitive position = (1, 2)	移动
Rotate:	aPrimitive orientation = 30	旋转
Scale:	aPrimitive scale = (10, 1)	缩放
Set color:	aPrimitive color = (1, 0, 0)	设置颜色

6) 在适当的地方应用继承。到目前为止只有一个继承层次,跳过这一步。

第二趟

现在重复执行这 6 步,继续增加对象。

1) 对每个子系统标识对象。图元的类型有多边形、二次曲线和样条三种。这一层对象的标识结果如图 6. 20 (b) 所示。

2) 对每个对象标识属性。下面步骤只是对二次曲线进行讨论。用 6 个系数定义一条二次曲线,并成为该类的实例变量。

3) 标识每个抽象中的操作。增加新的操作来建立和检索这 6 个系数。

set_ coefficients	设置系数
get_ coefficients	取得系数

此外,由于二次曲线具有与图元不同的数据结构,需要一个操作来建立一条二次曲线。

create_ conic	建立一条二次曲线
---------------	----------

4) 标识对象之间的通信。

new!	create_ conic	建立一条二次曲线
coefficients=	set_ coefficients	设置系数
coefficients?	get_ coefficients	取得系数

5) 使用脚本测试设计。

需 求	脚 本	
Create:	Conic new! name = aConic	建立曲线
Modify:	aConic coefficients = (1, 0, 3, 4, 5, 3)	修改曲线
Move:	aConic position = (1, 2)	移动
Rotate:	aConic orientation = 30	旋转
Scale:	aConic scale = (10, 1)	缩放
Set color:	aConic color = (1, 0, 0)	设置颜色

6) 在适当的地方应用继承。由于是按自顶向下方式继续作的,无继承。

第三趟

在这一趟将完成处理。

1) 对每个子系统标识数据抽象。6 个系数确定了一个二次曲线,但这对于用户定义形状来说,不是一个方便的操作。我们注意到,用户能够很容易地确定圆和椭圆的参数,无论是数值的还是图形的。只要能确定它们的参数,则可以类似地把双曲线和抛物线也加进

来。这样,就得到了新的一层抽象,如图 6.21 所示。

图 6.21 抽象的第三层

2) 对每个对象标识属性。

下面只对 circle(圆)标识属性。在 circle 中要追加一个属性,即 circle 的半径。

3) 标识每个抽象中的操作。

set_radius	设置圆的半径
get_radius	取得圆的半径

圆是一个特殊的二次曲线,用一个操作建立一个圆。

set_circle	为圆设置二次曲线系数
------------	------------

由于 circle 继承了 Conic(二次曲线)的操作,必须检查 Conic 的操作,看是否需要取代。例如,在 Conic 方程中, circle 有某些设置为 0 的系数,如果让 circle 继承 Conic 的 set_coefficients 操作,则可能建立的是我们认为是 circle,但实际是 Conic 的图形形状。必须替代掉 Conic 的 set_coefficients 操作。此时 Conic 的 get_coefficients 操作仍然有效。

set_coefficients

4) 标识对象之间的通信。

new!	create_circle	建立一个圆
radius=	set_radius	设置半径
radius?	get_radius	取得半径
set_coefficients	set_coefficients	设置系数

5) 使用脚本测试设计。

Create:	Circle new! name= aCircle	建立圆
Modify:	aCircle radius= 10	修改圆
Move:	aCircle position= (1, 2)	移动
Rotate:	aCircle orientation= 30	旋转
Scale:	aCircle scale= (10, 1)	缩放
Set color:	aCircle color= (1, 0, 0)	设置颜色

6) 在适当的地方应用继承。在对椭圆重复这些步骤之后,我们认识到圆是椭圆的一

个特化。

6.6 Coad 与 Yourdon 面向对象分析与设计技术

Coad 与 Yourdon 的方法是在信息模型化技术、面向对象程序设计语言及知识库系统的基础上发展起来的。这个方法分为面向对象分析(OOA)和面向对象设计(OOD)两个部分。

6.6.1 面向对象的分析

Coad 与 Yourdon 认为 OOA 的主要考虑在于与一个特定应用有关的对象以及对象与对象之间在结构与相互作用上的关系。OOA 有两个任务。

- 1) 形式地说明我们所面对的应用问题, 最终成为软件系统基本构成的对象, 还有系统必须遵从的, 由应用环境所决定的规则和约束。
- 2) 明确地规定构成系统的对象如何协同合作, 完成指定的功能。

通过 OOA 建立的系统模型是以对象概念为中心的, 因此称为概念模型。这样的模型由一组相关的类组成。OOA 可以采用自顶向下的方法, 逐层分解建立系统模型, 也可以自底向上地从已有定义的基本类出发, 逐步构造新的类。软件规格说明将基于这样的概念模型形成, 以模型描述为基本部分, 在加上接口要求、性能限制等其它方面的要求说明。

构造和评审 OOA 概念模型的顺序由五个层次组成。这五个层次不是构成软件系统的层次, 而是分析过程中的层次, 也可以说是问题的不同侧面。每个层次的工作都为系统的规格说明增加了一个组成部分。当五个层次的工作全部完成时, OOA 的任务也就完成了。这五个层次是: 类与对象、属性、服务、结构和主题。

图 6.22 给出了这五个层次, 以及每个层次中涉及到的主要概念和相应的图形表示。

第一个层次主要是识别类和对象。类和对象是对与应用有关的概念的抽象。不仅是说明应用问题的重要手段, 同时也是构成软件系统的基本元素。这一层的工作是整个分析模型的基础。Coad 与 Yourdon 还给出帮助识别类和对象的具体建议, 例如, 应注意信息结构、外部系统和设备、需要记住的事件、事物所扮演的角色、操作程序、组织机构和地理位置, 以便从中发现有用的类和对象。

下面两个层次称为属性层和服务层, 对前面已识别的类和对象作进一步的说明, 参看图 6.23 与图 6.24。在这里, 对象所保存的信息称为它的属性, 对象收到消息后所能执行的操作称为它可提供的服务。对每个对象和结构的增加、修改、删除、选择等服务有时是隐含的, 在图中不标出, 但在存储类和对象有关信息的对象库中有定义。其它服务则必须显式地在图中画出。

图 6. 22 分析过程中的五个层次

图 6. 23 属性层的例子

图 6. 24 服务层的例子

两个对象往往由于受制于相同的应用规则而发生联系,这称为实例连接。代表报刊订阅的对象与代表订户的对象之间存在着实例连接。为了表示一份报刊订阅仅仅与一个订户有关,可以附加适当的数字在表示实例连接的线段两端。这种情况可能是某个报社的特别规定,意味着在一份报刊订阅取消后,有关订户的信息也应随之删除。

两个对象之间还可能存在由于通信的需要而形成的联系,称为消息连接。消息连接表示从一个对象发送消息到另一个对象,由那个对象完成某些处理。它们在图中用箭头表示,方向从发消息的对象指向收消息的对象。

在图 6. 24 中可看到,两个对象都可以提供一些服务,它们之间的有向线段表明“报刊订阅”对象发送消息给“订户”对象,请求后者提供某个服务。

属性层侧重于定义类与对象的属性以及实例连接;服务层则侧重于说明类与对象能提供的服务,还有消息连接。

第四层是结构层。OOA 允许两种类型的基本结构。一种是整体与部分的结构,也叫

作组装结构。组装结构表示聚合,即由属于不同类的成员聚合而形成新的类,它用符号表示有向性,在 的上面是一个整体对象,下面是部分对象。一个整体可有多个部分,也可有不同种类的部分。通常,在分析中常遇到的集合-成员关系,就是组装结构。图6.25中的例子说明报社是由采访组、编辑室和印刷厂等几个部门组成,同时也指出,一个报社只有一个编辑室、一个印刷厂,但可以有一至多个采访组。

另一种是泛化与特化的结构,也叫作分类结构。如图 6.26 所示。其中,特化类是泛化类的子类,泛化类是特化类的父类。分类结构具有继承性,泛化类和对象的属性和服务一旦被识别,即可在特化类和对象中使用。

采用继承来显式地表达属性和服务的公共部分,可以实现在分类结构中恰如其分地分配属性和服务。将共同的属性放在上层,而将特有的属性放在下层;将共同的服务放在上层,而将特有的服务放在下层。

分类结构还可以形成层次或网络,以描述复杂的特化类,有效地表示公共部分。图中的例子说明“发表的文章”和“接受的文章”是“文章”的特殊形态,而“文章”则是它们的一般化。同时,“文章”对象所具有的属性和服务可自动地为“发表的文章”和“接受的文章”所继承。

图 6.25 整体与部分结构的例子

图 6.26 泛化与特化结构的例子

最后的一个层次是主题层。面向对象的概念模型相当大,是一个包含大量类和对象的平面图。通过对主题的识别,将这些类和对象作进一步组合。因而,主题可以看成是高层的模块或子系统。图 6.27 给出的“编辑管理”主题包括所有与编辑功能直接相关的类和对象。

最后,应当指出的是,服务层中的消息连接实际上引入了对系统动态行为的描述。消息连接的图形表示非常简单,但伴随图形表示的文字说明却十分详尽。Coad 与 Yourdon 对文字说明部分的要求也有详细规定。例如,在说明中可以看到一个对象在什么状态下对哪个消息作出怎样的反应。也就是说,每个对象被看成一个自动机。系统中所有这样的自动机构成了描述系统动态行为的基础。

在 Coad 最近出版的书中加上了交互作用图(类似于 OMT 的事件追踪图),以描述几个对象如何协同合作,以完成某个特定的系统功能。

6.6.2 面向对象的设计

Coad 与 Yourdon 在设计阶段中继续采用分析阶段中提到的五个层次,他们认为这有助于从分析到设计的过渡。不同的是,在设计阶段中,这五个层次是用于建立系统的四

图 6.27 主题层的例子(编辑管理主题)

个组成成分上。这四个组成成分和问题论域、用户界面、任务管理和数据管理。在 OOA 中实际只涉及到问题论域部分,其它三个部分是在 OOD 中加进来的,参看图 6.28。

图 6.28 OOD 设计导出的系统结构

问题论域部分包括与应用问题直接有关的所有类和对象。由于识别和定义这些类和对象的工作在 OOA 中已开始,这里只是对它们作进一步的细化。例如,加进有关如何利用现有的程序库的细节,以利于系统的实现。在其它的三个部分中,识别和定义新的类和对象。这些类和对象形成问题论域部分与用户、与外部系统和专用设备,以及与磁盘文件和数据库管理系统的界面。Coad 与 Yourdon 强调这三部分的作用主要是保证系统基本功能的相对独立,以加强软件的可复用性。假如外部的通信系统更新了,相应的通信协议也应有所变化。在这种情况下,我们只需修改任务管理部分中的某些类和对象,而不必对其它几个部分作任何修改。

6.7 Booch 的方法

就面向对象的方法而言,Booch 是最早的倡导者之一,他的工作可回溯到 20 世纪 80 年代初期。

6.7.1 Booch 方法的设计过程

Booch 认为软件开发是一个螺旋上升的过程。在这个螺旋上升的每个周期中, 有以下几个步骤:

- 1) 发现类和对象;
- 2) 确定它们的含义;
- 3) 找出它们之间的相互关系;
- 4) 说明每一个的界面和实现。

Booch 对每一步的目的、具体做法、最后的产品、以及检验方法都做了详细的讨论。

在第一步中, 从用来说明应用问题的词法和概念中识别对象, 通过对具体对象的抽象化发现类。然后考虑已识别的类和对象在完成系统功能上应承担的责任和所起到的作用, 在这个基础上确定每一类的属性和行为。第三步中, 找出类与类、对象与对象之间的相互关系。密切相关的一些对象协同作业, 以完成部分的系统功能, 同时也构成系统的一个必要组成部分, Booch 称之为“机构”。识别这样的机构也是这一步的一个重要目的。最后一步说明每一类的界面和实现, 同时将类和对象分配到不同的模块中, 将可同时执行的进程分配到不同的处理机上。这一步是对已有定义的细化和完善过程, 往往有助于发现新的类和对象, 因而导致下一周期的开发工作。

图 6.29 给出了 Booch 的面向对象的开发模型。这个模型分为逻辑设计和物理设计两个部分。逻辑设计部分包括两个文件: 类图和对象图, 着重于类和对象的定义。物理设计部分也包括两个文件: 模块图和进程图, 是针对着软件系统的结构的。

Booch 还区分静态模型和动态模型。静态模型侧重于系统的构成和结构, 而动态模型则侧重于系统在执行过程中的行为。除了上面提到的几个基本文件以外, Booch 的方法还包括状态迁移图和时序图, 这两个文件主要用于描述系统的动态行为。

图 6.29 面向对象的
软件开发模型

6.7.2 Booch 方法的基本的模型

在 Booch 的方法中, 用于说明系统要求的表示方法和手段非常丰富, 相当灵活。这里只介绍主要的部分。

1) 类图

类图用于表示类的存在以及类与类之间的相互关系, 是从系统构成的角度来描述正在开发的系统。虚线为边界的云状图符(云图), 表示一个类。它的名字、属性和操作则可列于其中, 参看图 6.30。

在 Booch 的方法中, 所有的图形表示都伴随有详细的文字说明, 因而不必将全部细节都画在图中。

根据面向对象的观点, 一个类的存在不是孤立的。类与类之间以不同方式互相合作,

共同完成某些系统功能。几种基本的关系及其图形表示如图 6.31 所示。

图 6.30 类的图形表示

图 6.31 类与类之间的相互关系图示

第一种是关联关系,表示两个类之间存在着某种语义上的联系,其真正含义要由附加在横线之上的一个短语予以说明。中间两个关系分别是继承和包含关系。在表示继承关系的图符中,箭头由子类指向基类。在表示包含关系的图符中,带有实心圆的一端表示整体,相反的一端表示部分。最后一个是使用关系。带有空心圆的一端连接在请求服务(或操作)的类,相反的一端连接在提供服务(也就是执行操作)的类。图 6.32 给出一个有关温室管理系统的类图。在“管理计划”类中,有一个“作物”属性,以及“实施”和“可否收获”两个操作。在“执行机构”的图符中出现的包含字母 A 的倒三角指出“执行机构”类是一个抽象的基类,它提供这一类设备共同具有的基本操作。例如“启动”和“关闭”;需要有关温度的数据。“环境控制器”包含着一个“暖气”、一个“冷气”和多个“灯光”。

图 6.32 温室管理系统的类图

2) 对象图

对象图用于表示对象的存在以及它们之间的相互关系。在一个系统的生存期中,类的存在基本上是稳定的,而对象则不断地从产生到消灭,经历着一系列的变化。一个对象图则是描述在这个过程中某一时刻的脚本,也可以说是用来说明决定系统行为的基本结构。从图 6.33 所示的对象图中可以看到,Booch 的方法用实线的云状图符表示对象,右下侧有黑影的云状图符表示系统程序库中的公用程序,云状图符之间的连线及其之上的箭头

图 6.33 温室管理系统的对象图

表示消息的传送和方向, 为表示消息传送的次序, 序列号码附加在每个消息之间。

图中的对象图描述的是在执行温室管理系统的一个常用功能时的脚本。这个功能是预估某一作物的成本和产量。完成这个功能需要几个不同对象的合作。首先,“计划分析”类的一个无名对象发给“计划度量”类公用程序一个消息, 要求以 C、“谷物”类的一个对象作为参数调用它的“收获时间”操作。然后,“计划度量”类公用程序发送消息给“管理计划”类的一个未命名的对象, 要求调用它的“状态”操作。其后,“管理计划”的这个对象向“谷物”类的对象 C 要求调用“成熟时间”操作。此后, 控制返回到“计划分析”类的那个无名对象, 继续其它操作。图中右下角的便笺状的图符用于注释, 它指出所有这一切都发生在某个计划正在执行中的情况下。

3) 状态迁移图

状态迁移图用来说明每一类的状态空间、触发状态迁移的事件, 以及状态迁移所执行的操作。它提供了描述一个类的动态行为的手段。

图 6.34 给出了“环境控制器”类的状态迁移图。从图中可以看到 Booch 的表示方法非常类似于 OMT 的表示方法。同样地, Booch 也主张采用类似结构化的方法来减少状态迁移图的复杂性。

图 6.34 “环境控制器”的状态迁移图

4) 交互作用图

Booch 方法中的交互作用图不仅在概念上,而且在表示方法上都与 OMT 的事件序列图十分相似,如图 6.35 所示。不同的是 Booch 的交互作用图主要表示操作而不是事件。交互作用图用于追踪系统执行过程中的一个可能的脚本,也就是几个对象在共同完成某一系统功能中所表现出来的交互关系。

图 6.35 温室管理系统的交互作用图

Booch 指出交互作用图实际上是另一种形式的对象图,完全有可能利用软件工具在一个图的基础上自动生成另一个。交互作用图可以清楚地展示消息传送的序列,无需使用数字。而对象图则能够表现较为复杂的操作调用,同时出示其它的有关信息。因而,Booch 的方法包括这两种图。

5) 模块图

模块图在系统的物理设计中说明如何将类和对象分配到不同的软件模块中。具体方法与最后代码编写时所采用的程序设计语言有关。在多数语言中,文件是基本的模块。而有些语言,例如 C++ ,文件分为声明文件(以.h 为后缀的文件)和定义文件(以.cpp 为后缀的文件)。

图 6.36 给出了一个模块图的例子。图中的方形图符用以表示声明文件,其右下侧的

图 6.36 黑影表示温室管理系统的模块图

相应的定义文件。从一个文件图符到另一个文件图符的箭头表示两者之间的编译依赖关系,也就是说,在箭头所指向的文件编译完成之后,相反一端的文件不可进行编译。

除了文件层的模块图以外,Booch 的方法还建议用子系统层的模块图来描述系统的主要部分之间以及与外部系统(如数据库)之间的关系。

6) 进程图

进程图在系统的物理设计中说明如何将可同时执行的进程分配到不同的处理机上。即使对于运行于单处理机之上的系统,进程图也是有用的,因为它可以表示同时处于活动状态的对象,以便决定进程调度方法。图 6. 37 给出的是温室管理系统的进程图。立方形的图符代表进程,它们之间的连线表示进程之间的通信关系。

图 6. 37 温室管理系统的进程图

总之,Booch 的方法内涵丰富,涉及到面向对象的软件系统所有各个方面,是广为使用的几个面向对象的方法之一。Booch 不仅建立了开发方法,还提出了对设计人员的技术要求,以及在开发过程的不同阶段(分析、设计、代码编写、测试与集成)中资源与人力的分配。此外,他还讨论了设计阶段的工作何时结束,实现阶段的工作何时结束这样一些较为困难的问题。但是,Booch 的方法偏重于设计,虽然也讨论了面向对象的分析,但未能提供足够的指导。与其它方法相比,Booch 的方法用到的概念和符号要复杂一些,软件工具的支持是必要的。应当指出,这样的软件工具已经在市场上出现。

6. 8 面向对象设计的实现

在开发过程中,类的实现是核心问题。在用面向对象风格所写的系统中,所有的数据都被封装在类的实例中。而整个程序则被封装在一个更高级的类中。在使用既存部件的面向对象系统中,可以只花费少量时间和工作量来实现软件。在实现一个特定的软件之前,可以把许多必要的功能事先设计在类中。只要增加类的实例,开发少量的新类和实现各个对象之间互相通信的操作,就能建立需要的软件。

6. 8. 1 类的实现

类的实现有多种方案。一种方案是先开发一个比较小的比较简单的类,作为开发比较大的、比较复杂的类的基础。即从简单到复杂的开发方案。在这种方案中,类的开发是分

层的。一个类建立在一些既存的类的基础上,而这些既存的类又是建立在其它既存的类的基础上。

1) “原封不动”复用

寻找“原封不动”使用的既存类,提供所需要的特性。此时,所需要的类已经存在,建立它的一个实例,用以提供所需要的特性。这个实例可直接为软件利用,或者它可以用来作另一个类的实现部分。通过复用一个既存类,可得到不加修改就能工作的已测试的代码。由于大多数面向对象语言的两个特性,即类的规格说明与实现的分离(信息隐蔽)和封装,这种复用一般是成功的。

2) 进化性复用

此时,一个能够完全符合要求特性的类可能并不存在。但是,如果具有类似功能的类存在,则可以通过继承,由既存类渐进式地设计新类。如图 6.38 所示,如果新类将要成为一个既存类的子类,它应当继承这个既存类的所有特性。然后新类可以对需要追加的数据及必需的功能做局部定义。还可以将几个既存类的特性混合起来开发出新的类。每个既存类是某些概念的模型。混合起来则产生了一个为特定应用所用的具有多重概念的类。一个既存类可能会提供某些在我们的新类中需要的特性以及某些新类中不需要的特性。因此,可以先建立一个新的更抽象的类,使之成为我们要设计的类的父类,然后,修改既存类以继承新的父类。既存类 A 的某些特性成为新类 B 的一个部分,同时被类 A 和类 C 继承。类 A 的某些特性保留在类 A 中,它不被类 C 继承。

图 6.38 进化性复用

3) “废弃性”开发

不用任何复用,开发一个新类。虽然不需要使用既存类来演变成新类,但还是有复用的可能。在新类实现时,通过说明一些既存类的实例,可以加快一个类的实现。像表格、硬件接口等都可以用来作为一个新类的局部。

4) 断言(asserttions)

实现类的一个方法是把类的设计信息直接组织到代码中。特别要求把参数约束、循环执行等编入到代码中。这可以通过某些表示断言的语言机制来实现。一个断言是一个语句,它表达对一个过程、一个值、甚至一段代码的需求。在栈的 ADT 的描述中,可以使用断言控制进栈(push)和退栈(pop)功能的操作:

```
procedure push(var S: Stack_Type; New_Item: Item_Type);
    assert: The stack S is not full
    .....
    assert: The top of stack S contains New_Item
end;
```

```

procedure pop(var S: Stack_Type) return Item_Type;
    assert: The stack S is not empty
    .....
    assert: The stack S has one fewer items than it did on entry
end;

```

有一类断言叫作先决条件,因为它们陈述了操作执行事先必须满足的条件;还有一类断言叫作后置条件,因为它们陈述了在操作执行之后必须满足的条件。例如,使用 pop 命令时,在操作执行之后,栈中元素的个数应当比操作执行之前栈中元素个数少 1。在 C 与 C++ 中有一种 header 文件,叫作“assert.h”,它给出了支持断言的格式。在这种文件中有一些宏,允许实现者把一些条件插入到操作中。宏展开时把条件提交给检验工具。实际上,工具包括检验工具和报告工具。实现者可能针对 pop 操作,陈述断言如下:

```
assert(TOP > 0)
```

这样,宏会检查在从栈中退出一个项之前栈是否空。如果条件测试失败,则会打印出一条消息,报告源文件名及在文件中发生失效的行号。这是一个简单的方法,但它允许实现者在代码的临界点快速地进行检验。代码保持了简洁,且没有会干扰实现者的源代码的(插入)条件。eiffel 的设计者 Meyer 相信,使用这种方法能够得到简洁的设计,通过在命令执行之前检验先决条件,实现者有可能在程序终止前准确地报告错误。这是一个安全的方法。

5) 错误处理(error handling)

一个类应是自主的,有责任定位和报告错误,但可能因为错误的严重程度和对它所处理的不同而不能实现。C 程序员在错误处理中使用状态码方法。函数可返回一个整数值,它就是状态码,可供调用此函数的例程检查。各种不同的状态码的值能够指明任务的执行是成功还是失败,若是失败又是哪种程度的失败。例如, C 中函数 fopen(打开文件)所返回的状态码。如果打开失败,则返回零值;如果打开成功,则返回文件的标志。

使用这种方法的难点在于各层程序代码必须知道该层所调用函数的状态码,并且检验这些状态码及采取行动。问题在比它发生的那一层更高的一层进行处理,这将产生比预想更高程度的耦合。如有可能,问题应当在它发生的那一层进行处理。例如,在 fopen 打开文件失败时,如果当前的文件名不存在,待开发软件可以要求用户键入另一个文件名。

6) 多重实现(multiple implementation)

软件库的概念和关系的概念形成了同一个类的多重实现的基础。软件库必须对库中的每一部分都能保留充足的信息,使得定义能同时关联到不止一个实现。

图 6.39 说明了把一个定义联结到几个实现所使用的联系。设计者必须指出要求的实例所在的类,并确定所需的特定实现。画出在静态存储与动态存储管理之间进行的选择,或在基于树形结构和基于散列表结构的实现之间进行的选择,无需把每一种不同的实现都画出来。

6.8.2 系统的实现

系统的实现是在所有的类都被实现之后的事情。事实上,实际实现一个系统是一个比

用过程性方法更简单、更简短的过程。因为在 C++ 程序中它的主过程很小。实际上,当把类开发出来时就已实现了系统。每个类提供了完成系统所需要的某种功能。完成类的过程要求建立这些类的实例。有些实例将在其它类的初始化过程中使用。而其余的则必须用某种主过程显式地加以说明,或者当作系统最高层的类的表示的一部分。

在 C++ 和 C 中有一个 main() 函数。可以使用这个过程说明构成系统主要对象的那些类的实例。以图形为例,建立一个用户界面实例。一旦它建立起来,就发送一个消息,启动绘图程序的命令循环。然后,这个对象担负起在系统寿命的其余时期协调通信联系并建立对象的责任。对于纯面向对象的语言,在系统中的每个“事物”都是对象。在这些语言中没有“主过程”,且常常是交互的。用户在他们的环境中建立起一个类的实例,然后,接受控制和执行操作,产生实例建立的结果或接收由用户发送来的消息。由那些原始消息而产生的消息序列就是所要开发的软件。

图 6.39 把一个定义联结到几个实现所使用的联系

第 7 章 软件测试

软件系统的开发体现了人们智力劳动的成果。在软件开发过程中, 尽管人们利用了许多旨在改进、保证软件质量的方法去分析、设计和实现软件, 但难免会在工作中犯这样那样的错误。这样, 在软件产品中就会隐藏许多的错误和缺陷。对于规模大、复杂性高的软件更是如此。在这些错误中, 有些甚至是致命的错误, 如果不排除, 就会导致财产以至生命的重大损失。例如, 1963 年在美国发生了这样一件事: 一个 FORTRAN 程序的循环语句

```
DO 5 I= 1, 3
```

被误写为

```
DO 5 I= 1. 3
```

由于空格对 FORTRAN 编译程序没有实际意义, 误写的语句被当作了赋值语句

```
DO5I= 1. 3
```

这里“ , ”被误写为“ . ”, 一点之差致使飞往火星的火箭爆炸, 造成 1000 万美元的损失。这种情况迫使人们必须认真计划、彻底地进行软件测试。

7.1 软件测试的基础

7.1.1 什么是软件测试

为了保证软件的质量和可靠性, 人们力求在分析、设计等各个开发阶段结束之前, 对软件进行严格的技术评审。但即使如此, 由于人们本身能力的局限性, 审查还不能发现所有的错误。而且在编码阶段还会引进大量的错误。这些错误和缺陷如果在软件交付投入生产性运行之前不能加以排除的话, 在运行中迟早会暴露出来。但到那时, 不仅改正这些错误的代价更高, 而且往往造成很恶劣的后果。

软件测试是在软件投入生产性运行之前, 对软件需求分析、设计规格说明和编码的最终复审, 是软件质量保证的关键步骤。如果给软件测试下定义的话, 可以这样讲: 软件测试是为了发现错误而执行程序的过程。或者说, 软件测试是根据软件开发各阶段的规格说明和程序的内部结构而精心设计一批测试用例(即输入数据及其预期的输出结果), 并利用这些测试用例去运行程序, 以发现程序错误的过程。

软件测试在软件生存期中横跨两个阶段: 通常在编写出每一个模块后就对它作单元测试。模块的编写者与测试者是同一个人。在每个模块都完成单元测试后, 对软件系统还要进行各种综合测试, 通常由专门的测试人员承担这项工作。

现在, 软件开发机构将研制力量的 40% 以上投入到软件测试之中的事例越来越多。特殊情况下, 对于性命攸关的软件, 例如飞行控制、核反应堆监控软件等, 其测试费用甚至高达所有其它软件工程阶段费用总和的 3 ~ 5 倍。

7.1.2 软件测试的目的和原则

基于不同的立场,存在着两种完全不同的测试目的。从用户的角度出发,普遍希望通过软件测试暴露软件中隐藏的 errors 和缺陷,以考虑是否可以接受该产品。而从软件开发者的角度出发,则希望测试成为表明软件产品中不存在错误的过程,验证该软件已正确地实现了用户的要求,确立人们对软件质量的信心。因此,他们会选择那些导致程序失效概率小的测试用例,回避那些易于暴露程序错误的测试用例。显然,这样的测试对提高软件质量毫无价值。如果我们站在用户的角度,替他们设想,就应当把测试活动的目标对准揭露程序中存在的 errors。在选取测试用例时,考虑那些易于发现程序错误的 data。鉴于此,Grenford J. Myers 就软件测试目的提出以下观点。

- 1) 测试是程序的执行过程,目的在于发现 errors;
- 2) 一个好的测试用例在于能发现至今未发现的 errors;
- 3) 一个成功的测试是发现了至今未发现的 errors 的测试。

这几句话的意思就是说,设计测试的目标是想以最少的时间和人力系统地找出软件中潜在的各种 errors 和缺陷。如果我们成功地实施了测试,就能够发现软件中的 errors。测试的附带收获是,它能够证明软件的功能和性能与需求说明相符合。此外,实施测试收集到的测试结果 data 为可靠性分析提供了依据。这里,特别需要说明的是,测试不能表明软件中不存在 errors,它只能说明软件中存在 errors。

根据这样的测试目的,软件测试的原则应该是:

- 1) 应当把“ 尽早地和不断地进行软件测试 ”作为软件开发者的座右铭。

由于原始问题的复杂性,软件本身的复杂性和抽象性,软件开发各个阶段工作的多样性,以及参加开发各种层次人员之间工作的配合关系等因素,使得开发的每个环节都可能产生 errors。所以我们不应把软件测试仅仅看作是软件开发的一个独立阶段,而应当把它贯穿到软件开发的各个阶段中。坚持在软件开发的各个阶段的技术评审,这样才能在开发过程中尽早发现和预防 errors,把出现的 errors 克服在早期,以提高软件质量。

- 2) 测试用例应由测试输入 data 和与之对应的预期输出结果这两部分组成。

测试以前应当根据测试的要求选择测试用例(test case),以在测试过程中使用。测试用例主要用来检验程序员编制的程序,因此不但需要测试的输入 data,而且需要针对这些输入 data 的预期输出结果,作为检验实测结果的基准。

- 3) 程序员应避免检查自己的程序。

程序员应尽可能避免测试自己编写的程序,程序开发小组也应尽可能避免测试本小组开发的程序。如果条件允许,最好建立独立的软件测试小组或测试机构。这是因为人们常由于各种原因具有一种不愿否定自己工作的心理,认为揭露自己程序中的问题总不是一件愉快的事。这一心理状态就成为测试自己程序的障碍。另外,程序员对软件规格说明理解 errors 而引入的 errors 则更难发现。但这并不是说程序员不能测试自己的程序。而是说由别人来测试可能会更客观、更有效,并更容易取得成功。

- 4) 在设计测试用例时,应当包括合理的输入条件和不合理的输入条件。

所谓合理的输入条件是指能验证程序正确的输入条件,而不合理的输入条件是指异

常的、临界的、可能引起问题异变的输入条件。在测试程序时,人们常常倾向于过多地考虑合法的和期望的输入条件,以检查程序是否作了它应该做的事情,而忽视了不合法的和预想不到的输入条件。事实上,软件在投入运行以后,用户的使用往往不遵循事先的约定,使用了一些意外的输入,如果我们开发的软件遇到这种情况时不能作出适当的反应,就容易产生故障,轻则给出错误的结果,重则导致软件失效。因此,用不合理的输入条件测试程序时,往往比用合理的输入条件进行测试能发现更多的错误。

5) 充分注意测试中的群集现象。

测试时不要被一开始发现的若干错误所迷惑,找到了几个错误就以为问题已经解决,不需要继续测试了。经验表明,测试后程序中残存的错误数目与该程序的错误检出率成正比。如图 7.1 所示。根据这个规律,应当对错误群集的程序段进行重点测试。

在被测程序段中,若发现错误数目多,则残存错误数目也比较多。这种错误群集性现象,已为许多程序的测试实践所证实。例如美国 IBM 公司的 OS/370 操作系统中,47% 的错误仅与该系统的 4% 的程序模块有关。这种现象对测试很有用。

6) 严格执行测试计划,排除测试的随意性。

对于测试计划,要明确规定,不要随意解释。

7) 应当对每一个测试结果作全面检查。

有些错误的征兆在输出实测结果时已经明显地出现了,但是如果

图 7.1 错误群集现象

如果不仔细地全面地检查测试结果,就会使这些错误被遗漏掉。所以必须对预期的输出结果明确定义,对实测的结果仔细分析检查,抓住征候,暴露错误。

8) 妥善保存测试计划、测试用例、出错统计和最终分析报告,为维护提供方便。

7.1.3 软件测试的对象

软件测试并不等于程序测试。软件测试应贯穿于软件定义与开发的整个期间。因此,需求分析、概要设计、详细设计以及程序编码等各阶段所得到的文档资料,包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序,都应成为软件测试的对象。软件测试不应仅限在程序测试的狭小范围内,而置其它阶段的工作于不顾。

另一方面,由于定义与开发各阶段是互相衔接的,前一阶段工作中发生的问题如未及时解决,很自然要影响到下一阶段。从源程序的测试中找到的程序错误不一定是程序编写过程中造成的。不能简单地把程序中的错误全都归罪于程序员。据美国一家公司的统计表明,在查找出的软件错误中,属于需求分析和软件设计的错误约占 64%,属于程序编写的错误仅占 36%。

事实上,到程序的测试为止,软件开发工作已经经历了许多环节,每个环节都可能发生问题。为了把握各个环节的正确性,人们需要进行各种确认和验证工作。

所谓确认(validation),是一系列的活动和过程,其目的是想证实在一个给定的外部环境中软件的逻辑正确性。它包括需求规格说明的确认和程序的确认,而程序的确认又分为静态确认与动态确认。静态确认一般不在计算机上实际执行程序,而是通过人工分析或

者程序正确性证明来确认程序的正确性;动态确认主要通过动态分析和程序测试来检查程序的执行状态,以确认程序是否有问题。

所谓验证(verification),则试图证明在软件生存期各个阶段,以及阶段间的逻辑协调性、完备性和正确性。图 7.2 中所示的是软件生存期各个重要阶段之间所要保持的正确性。它们是验证工作的主要对象。

图 7.2 软件生存期各个阶段之间需要保持的正确性

确认与验证工作都属于软件测试。在对需求理解与表达的正确性、设计与表达的正确性、实现的正确性以及运行的正确性的验证中,任何一个环节上发生了问题都可能在软件测试中表现出来。

7.1.4 测试信息流

测试信息流如图 7.3 所示。测试过程需要三类输入:

- 1) 软件配置:包括软件需求规格说明、软件设计规格说明、源代码等;
- 2) 测试配置:包括测试计划、测试用例、测试驱动程序等;从整个软件工程过程看,测试配置是软件配置的一个子集。
- 3) 测试工具:为提高软件测试效率,测试工作需要测试工具的支持,它们的工作是为测试的实施提供某种服务,以减轻人们完成测试任务中的手工劳动。例如,测试数据自动生成程序、静态分析程序、动态分析程序、测试结果分析程序、以及驱动测试的测试数据库等等。

测试之后,要对所有测试结果进行分析,即将实测的结果与预期的结果进行比较。如果发现出错的数据,就意味着软件有错误,然后就需要开始排错(调试)。即对已经发现的错误进行错误定位和确定出错性质,并改正这些错误,同时修改相关的文档。修正后的程序和文档一般都要经过再次测试,直到通过测试为止。

图 7.3 测试信息流

通过收集和分析测试结果数据,开始对软件建立可靠性模型。如果经常出现需要修改设计的严重错误,那么软件质量和可靠性就值得怀疑,同时也表明需要进一步测试。如果与此相反,软件功能能够正确完成,出现的错误易于修改,那么就可以断定软件的质量和可靠性达到可以接受的程度;或者所做的测试不足以发现严重的错误;

最后,如果测试发现不了错误,那么几乎可以肯定,测试配置考虑得不够细致充分,错误仍然潜伏在软件中。这些错误最终不得不由用户在使用中发现,并在维护时由开发者去改正。但那时改正错误的费用将比在开发阶段改正错误的费用要高出 40 倍到 60 倍。

7.1.5 测试与软件开发各阶段的关系

软件开发过程是一个自顶向下、逐步细化的过程,而测试过程则是依相反的顺序安排的自底向上、逐步集成的过程。低一级测试为上一级测试准备条件。当然不排除两者平行地进行测试。

参看图 7.4,首先对每一个程序模块进行单元测试,消除程序模块内部在逻辑上和功能上的错误和缺陷。再对照软件设计进行集成测试,检测和排除子系统(或系统)结构上的错误。随后再对照需求,进行确认测试。最后从系统全体出发,运行系统,看是否满足要求。

图 7.4 软件测试与软件开发过程的关系

7.2 测试用例设计

既然测试的目的在于寻找错误,并且找出的错误越多越好。很自然会提出这样的问题,能不能把所有隐藏的错误全都找出来呢?或者说能不能把所有可能做的测试无遗漏地一一做完,找出所有的错误呢?下面按两种常用的测试方法作出具体分析。

1) 黑盒测试

软件的测试设计与软件产品的设计一样,是一项需要花费许多人力和时间的工作。我们希望以最少量的时间和人力,最大可能地发现最多的错误。

任何工程产品都可以使用以下两种方法之一进行测试:

- (1) 已知产品的功能设计规格,可以进行测试证明每个实现了的功能是否符合要求。
- (2) 已知产品的内部工作过程,可以通过测试证明每种内部操作是否符合设计规格要求,所有内部成分是否已经过检查。

前者是黑盒测试,后者是白盒测试。

就软件测试来讲,软件的黑盒测试意味着测试要根据软件的外部特性进行。也就是说,这种方法是把测试对象看作一个黑盒子,测试人员完全不考虑程序内部的逻辑结构和内部特性,只依据程序的需求规格说明书,检查程序的功能是否符合它的功能说明。

黑盒测试方法主要是为了发现:是否有不正确或遗漏了的功能?在接口上,输入能否正确地接受?能否输出正确的结果?是否有数据结构错误或外部信息(例如数据文件)访问错误?性能上是否能够满足要求?是否有初始化或终止性错误?所以,用黑盒测试发现程序中的错误,必须在所有可能的输入条件和输出条件中确定测试数据,检查程序是否都能产生正确的输出。

现在假设一个程序 P 有输入量 X 和 Y 及输出量 Z,参看图 7.5。在字长为 32 位的计算机上运行。如果 X, Y 只取整数,考虑把所有的 X, Y 值都作为测试数据,按黑盒方法进行穷举测试,力图全面、无遗漏地“挖掘”出程序中的所有错误。

这样作可能采用的测试数据组 (X_i, Y_i) , 不同测试数据组合的最大可能数目为

$$2^{32} \times 2^{32} = 2^{64}$$

如果程序 P 测试一组 X, Y 数据需要 1 毫秒,而且假定一天工作 24 小时,一年工作 365 天,要完成 2^{64} 组测试,需要 5 亿年。

图 7.5 黑盒子

2) 白盒测试

软件的白盒测试是对软件的过程性细节作细致的检查。这一方法是把测试对象看作一个打开的盒子,它允许测试人员利用程序内部的逻辑结构及有关信息,设计或选择测试用例,对程序所有逻辑路径进行测试。通过在不同点检查程序的状态,确定实际的状态是否与预期的状态一致。因此白盒测试又称为结构测试或逻辑驱动测试。

软件人员使用白盒测试方法,主要想对程序模块进行检查:对程序模块的所有独立的执行路径至少测试一次;对所有的逻辑判定,取“真”与取“假”的两种情况都能至少测试一次;在循环的边界和运行界限内执行循环体;测试内部数据结构的有效性等等。

但是对一个具有多重选择和循环嵌套的程序,不同的路径数目可能是天文数字。而且即使精确地实现了白盒测试,也不能断言测试过的程序完全正确。举例来说,现在给出一个如图 7.6 所示的小程序的流程图,它对应了一个有 100 行源代码的 PASCAL 语言程序,其中包括了一个执行达 20 次的循环。它所包含的不同执行路径数高达 $5^{20}(= 10^{13})$ 条,若要对它进行穷举测试,即要设计测试用例,覆盖所有的路径。假使有这么一个测试程序,对每一条路径进行测试需要 1 毫秒,同样假定一天工作 24 小时,一年工作 365 天,那么要想把如图 7.6 所示的小程序的所有路径测试完,则需要 3170 年。

图 7.6 白盒测试中的穷举测试

以上情况表明,实行穷举测试,由于工作量过大,需用的时间过长,实施起来是不现实的。任何软件项目都要受到期限、费用、人力和机时等条件的限制,如果打算针对所有可能的数据进行测试,以充分揭露程序中的所有隐藏错误,这种作法是不现实的。

在测试阶段既然穷举测试不可行,就必须精心设计测试用例,从数量极大的可用测试用例中精心地挑选少量的测试数据,使得采用这些测试数据能够达到最佳的测试效果,或者说它们能够高效率地把隐藏的错误揭露出来。

以上事实说明,软件测试有一个致命的缺陷,即测试的不完全、不彻底性。由于任何程序只能进行少量(相对于穷举的巨大数量而言)的有限的测试,在发现错误时能说明程序有问题;但在未发现错误时,不能说明程序中没有错误,不能说明程序中没有问题。

下面将介绍几种实用的测试用例设计方法。其中,逻辑覆盖属于白盒测试,等价类划分、边界值分析、因果图等属于黑盒测试。

7.3 白盒测试的测试用例设计

7.3.1 逻辑覆盖

逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。它属白盒测试。这一方法要求测试人员对程序的逻辑结构有清楚的了解,甚至要能掌握源程序的所有细节。由于覆盖测试的目标不同,逻辑覆盖又可分为:语句覆盖、判定覆盖、判定-条件覆盖、条件组合覆盖及路径覆盖。以下将分别作出扼要的介绍。在所介绍的几种逻辑覆盖中,均以图 7.7 所示的程序段为例。其中有两个判断,每个判断都包含复合条件的逻辑表达式,并且,符号“ \wedge ”表示“and”运算,“ \vee ”表示“or”运算。

观察图 7.7 所给的例子, 可知该程序段有 4 条不同的路径。为了清楚起见, 分别对第一个判断的取假分支、取真分支及第二个判断的取假分支、取真分支命名为 b, c, d 和 e。这样所有 4 条路径可表示为: L1(a c e), L2(a b d), L3(a b e) 和 L4(a c d), 或简写为 ace、abd、abe 及 acd。若把各条路径应满足的逻辑表达式综合起来, 可以进行如下的推导, 其中的上划线“ ”表示“非”运算:

$$\begin{aligned}
 &L1(a \quad c \quad e) \\
 &= \{(A > 1) \text{ and } (B = 0)\} \text{ and } \{(A = 2) \text{ or } (X/A > 1)\} \\
 &= (A > 1) \text{ and } (B = 0) \text{ and } (A = 2) \text{ or } (A > 1) \text{ and } (B = 0) \text{ and } (X/A > 1) \\
 &= (A = 2) \text{ and } (B = 0) \text{ or } (A > 1) \text{ and } (B = 0) \text{ and } (X/A > 1) \\
 &L2(a \quad b \quad d) \\
 &= \overline{\{(A > 1) \text{ and } (B = 0)\}} \text{ and } \overline{\{(A = 2) \text{ or } (X > 1)\}} \\
 &= \overline{\{(A > 1) \text{ or } (B = 0)\}} \text{ and } \overline{\{(A = 2) \text{ and } (X > 1)\}} \\
 &= (A > 1) \text{ and } (A = 2) \text{ and } (X > 1) \text{ or } (B = 0) \text{ and } (A = 2) \text{ and } (X > 1) \\
 &= (A = 1) \text{ and } (X = 1) \text{ or } (B = 0) \text{ and } (A = 2) \text{ and } (X = 1) \\
 &L3(a \quad b \quad e) \\
 &= \overline{\{(A > 1) \text{ and } (B = 0)\}} \text{ and } \{(A = 2) \text{ or } (X > 1)\} \\
 &= \overline{\{(A > 1) \text{ or } (B = 0)\}} \text{ and } \{(A = 2) \text{ or } (X > 1)\} \\
 &= (A > 1) \text{ and } (X > 1) \text{ or } (B = 0) \text{ and } (A = 2) \text{ or } (B = 0) \text{ and } (X > 1) \\
 &= (A = 1) \text{ and } (X > 1) \text{ or } (B = 0) \text{ and } (A = 2) \text{ or } (B = 0) \text{ and } (X > 1) \\
 &L4(a \quad c \quad d) \\
 &= \{(A > 1) \text{ and } (B = 0)\} \text{ and } \overline{\{(A = 2) \text{ or } (X/A > 1)\}} \\
 &= (A > 1) \text{ and } (B = 0) \text{ and } (A = 2) \text{ and } (X/A = 1)
 \end{aligned}$$

图 7.7 测试用例设计的参考例子

在上面为各条路径所导出的逻辑式中, 由符号“and(与)”联结起来的断言是为了遍历这条路径, 各个输入变量应取值的范围, 而用符号“or(或)”划分了几组可选的取值。依据以上推导出来的结果可以设计满足要求的测试用例。

7.3.2 语句覆盖

所谓语句覆盖就是设计若干个测试用例, 运行被测程序, 使得每一个可执行语句至少执行一次。例如在图 7.7 所给出的例子中, 正好所有的可执行语句都在路径 L1 上, 所以选择路径 L1 设计测试用例, 就可以覆盖所有的可执行语句。

测试用例的设计格式如下:

【输入的(A, B, x), 输出的(A, B, x)】

为图 7.7 所示例子设计满足语句覆盖的测试用例是

【2, 0, 4), (2, 0, 3)】覆盖 ace 【1】

从程序中每个可执行语句都得到执行这一点来看, 语句覆盖的方法似乎能够比较全面地检验每一个可执行语句。但与后面介绍的其它覆盖相比, 语句覆盖是最弱的逻辑覆盖

准则。

7.3.3 判定覆盖

所谓判定覆盖就是设计若干个测试用例, 运行被测程序, 使得程序中每个判断的取真分支和取假分支至少经历一次。判定覆盖又称为分支覆盖。例如对于图 7.7 给出的例子, 如果选择路径 L1 和 L2, 可得满足要求的测试用例:

【2, 0, 4), (2, 0, 3)】覆盖 ace 【1】

【1, 1, 1), (1, 1, 1)】覆盖 abd 【2】

如果选择路径 L3 和 L4, 还可得另一组可用的测试用例:

【2, 1, 1), (2, 1, 2)】覆盖 abe 【3】

【3, 0, 3), (3, 1, 1)】覆盖 acd 【4】

所以, 测试用例的取法不唯一。注意有例外情形, 例如, 若把图 7.7 例中第二个判断中的条件 $x > 1$ 错写成 $x < 1$, 那么利用上面两组测试用例, 仍能得到同样结果。这表明, 只是判定覆盖, 还不能保证一定能查出在判断的条件中存在的错误。因此, 还需要更强的逻辑覆盖准则检验判断内部条件。

以上仅讨论了两出口的判断, 我们还应把判定覆盖准则扩充到多出口判断(如 CASE 语句)的情况。

7.3.4 条件覆盖

所谓条件覆盖就是设计若干个测试用例, 运行被测程序, 使得程序中每个判断的每个条件的可能取值至少执行一次。例如在图 7.7 所给出的例子中, 我们事先可对所有条件的取值加以标记。例如,

对于第一个判断: 条件 $A > 1$ 取真值为 T1, 取假值为 $\overline{T1}$

条件 $B = 0$ 取真值为 T2, 取假值为 $\overline{T2}$

对于第二个判断: 条件 $A = 2$ 取真值为 T3, 取假值为 $\overline{T3}$

条件 $x > 1$ 取真值为 T4, 取假值为 $\overline{T4}$

则可选取测试用例如下:

测试用例	通过路径	条件取值	覆盖分支
【2, 0, 4), (2, 0, 3)】	ace (L1)	T1 T2 T3 T4	c, e
【1, 0, 1), (1, 0, 1)】	abd (L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$	b, d
【2, 1, 1), (2, 1, 2)】	abe (L3)	T1 $\overline{T2}$ T3 $\overline{T4}$	b, e

或

测试用例	通过路径	条件取值	覆盖分支
【1, 0, 3), (1, 0, 4)】	abe (L3)	$\overline{T1}$ T2 $\overline{T3}$ T4	b, e
【2, 1, 1), (2, 1, 2)】	abe (L3)	T1 $\overline{T2}$ T3 $\overline{T4}$	b, e

注意, 前一组测试用例不但覆盖了所有判断的取真分支和取假分支, 而且覆盖了判断中所有条件的可能取值。但是后一组测试用例虽满足了条件覆盖, 但只覆盖了第一个判断的取假分支和第二个判断的取真分支, 不满足判定覆盖的要求。为解决这一矛盾, 需要对

条件和分支兼顾,有必要考虑以下的判定- 条件覆盖。

7.3.5 判定- 条件覆盖

所谓判定- 条件覆盖就是设计足够的测试用例,使得判断中每个条件的所有可能取值至少执行一次,同时每个判断本身的所有可能判断结果至少执行一次。例如,对于图 7.7 中的各判断,若 T1, T2, T3, T4 及 $\overline{T1}, \overline{T2}, \overline{T3}, \overline{T4}$ 的含意如前所述,则只需设计以下两个测试用例便可覆盖图 7.7 的 8 个条件取值以及 4 个判断分支。

测 试 用 例	通过路径	条件取值	覆盖分支
【2, 0, 4), (2, 0, 3)】	ace (L1)	T1 T2 T3 T4	c, e
【1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T1} \ \overline{T2} \ \overline{T3} \ \overline{T4}$	b, d

判定- 条件覆盖也有缺陷。从表面上来看,它测试了所有条件的取值,但是事实并非如此。因为往往某些条件掩盖了另一些条件。对于条件表达式 (A> 1) and (B= 0) 来说,若 (A> 1) 的测试结果为真,则还要测试 (B= 0),才能决定表达式的值;而若 (A> 1) 的测试结果为假,可以立刻确定表达式的结果为假。这时,往往就不再测试 (B= 0) 的取值了。因此,条件 (B= 0) 就没有检查。

同样,对于条件表达式 (A= 2) or (X> 1) 来说,若 (A= 2) 的测试结果为真,就可以立即确定表达式的结果为真。这时,条件 (X> 1) 就没有检查。因此,采用判定- 条件覆盖,逻辑表达式中的错误不一定能够查得出来。

为彻底地检查所有条件的取值,可以将图 7.7 给出的多重条件判定分解,形成图 7.8 所示的由多个基本判断组成的流程图。这样可以有效地检查所有的条件是否正确。

图 7.8 分解为基本判定的例子

7.3.6 条件组合覆盖

所谓条件组合覆盖就是设计足够的测试用例,运行被测程序,使得每个判断的所有可能的条件取值组合至少执行一次。现在考察图 7.7 给出的例子,先对各个判断的条件取值组合加以标记。例如,

记 A> 1, B= 0 作 T1 $\overline{T2}$, 属第一个判断的取真分支;
 A> 1, B 0 作 T1 T2, 属第一个判断的取假分支;

A = 1, B = 0 作 $\overline{T1}$ $\overline{T2}$, 属第一个判断的取假分支;
A = 1, B = 0 作 $\overline{T1}$ $\overline{T2}$, 属第一个判断的取假分支;
A = 2, x > 1 作 T3 $\overline{T4}$, 属第二个判断的取真分支;
A = 2, x = 1 作 T3 $\overline{T4}$, 属第二个判断的取真分支;
A = 2, x > 1 作 $\overline{T3}$ $\overline{T4}$, 属第二个判断的取真分支;
A = 2, x = 1 作 $\overline{T3}$ $\overline{T4}$, 属第二个判断的取假分支。

对于每个判断, 要求所有可能的条件取值的组合都必须取到。在图 7.7 中的每个判断各有两个条件, 所以, 各有 4 个条件取值的组合。取 4 个测试用例, 可用以覆盖上面 8 种条件取值的组合。必须明确, 这里并未要求第一个判断的 4 个组合与第二个判断的 4 个组合再进行组合。要是那样的话, 就需要 $4^2=16$ 个测试用例了。

测试用例	通过路径	覆盖条件	覆盖组合号
【2, 0, 4), (2, 0, 3)】	ace(L1)	T1 $\overline{T2}$ T3 $\overline{T4}$,
【2, 1, 1), (2, 1, 2)】	abe(L3)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$,
【1, 0, 3), (1, 0, 4)】	abe(L3)	$\overline{T1}$ T2 $\overline{T3}$ T4	,
【1, 1, 1), (1, 1, 1)】	abd(L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$,

这组测试用例覆盖了所有条件的可能取值的组合, 覆盖了所有判断的可取分支, 但路径漏掉了 L4。测试还不完全。

7.3.7 路径测试

路径测试是设计足够的测试用例, 覆盖程序中所有可能的路径。若仍以图 7.7 为例, 则可以选择如下的一组测试用例, 覆盖该程序段的全部路径。

测试用例	通过路径	覆盖条件
【2, 0, 4), (2, 0, 3)】	ace(L1)	T1 T2 T3 T4
【1, 1, 1), (1, 1, 1)】	abd(L2)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$
【1, 1, 2), (1, 1, 3)】	abe(L3)	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ T4
【3, 0, 3), (3, 0, 1)】	acd(L4)	T1 T2 $\overline{T3}$ $\overline{T4}$

7.4 黑盒测试的测试用例设计

7.4.1 等价类划分

等价类划分是一种典型的黑盒测试方法, 也是一种非常实用的重要测试方法。
前面已经说过, 不可能用所有可以输入的数据来测试程序, 而只能从全部可供输入的数据中选择一个子集进行测试。如何选择适当的子集, 使其尽可能多地发现错误。解决的办法之一是等价类划分。使用这一方法设计测试用例要经历划分等价类(列出等价类表)和选取测试用例两步。以下分别加以说明, 然后给出实例。

1) 划分等价类

首先把数目极多的输入数据(有效的和无效的)划分为若干等价类。所谓等价类是指

某个输入域的子集合。在该子集合中,各个输入数据对于揭露程序中的错误都是等效的。并合理地假定:测试某等价类的代表值等价于对这一类其它值的测试。或者说,如果某个等价类中的一个输入条件作为测试数据进行测试查出了错误,那么使用这一等价类中的其它输入条件进行测试也会查出同样的错误;反之,若使用某个等价类中的一个输入条件作为测试数据进行测试没有查出错误,则使用这个等价类中的其它输入条件也同样查不出错误。因此,可以把全部输入数据合理划分为若干等价类,在每一个等价类中取一个数据作为测试的输入条件,就可以用少量代表性测试数据,取得较好的测试效果。

等价类的划分有两种不同的情况:

有效等价类:是指对于程序的规格说明来说,是合理的、有意义的输入数据构成的集合。利用它,可以检验程序是否实现了规格说明预先规定的功能和性能。

无效等价类:是指对于程序的规格说明来说,是不合理的、无意义的输入数据构成的集合。程序员主要利用这一类测试用例检查程序中功能和性能的实现是否有不符合规格说明要求的地方。

在设计测试用例时,要同时考虑有效等价类和无效等价类的设计。软件不能都只接收合理的数据,还要经受意外的考验,接受无效的或不合理的数据,这样获得的软件才能具有较高的可靠性。

以下结合具体实例给出几条划分等价类的原则。

(1) 如果输入条件规定了取值范围或值的个数,则可以确立一个有效等价类和两个无效等价类。例如,在程序的规格说明中,对输入条件有一句话:

“ 项数可以从 1 到 999 ”

则有效等价类是“ 1 项数 999 ”,两个无效等价类是“ 项数 < 1 ”或“ 项数 > 999 ”。在数轴上表示成

(2) 如果输入条件规定了输入值的集合,或者是规定了“ 必须如何 ”的条件,这时可确立一个有效等价类和一个无效等价类。例如,在 PASCAL 语言中对变量标识符规定为“ 以字母打头的 串 ”。那么所有以字母打头的构成有效等价类,而不在集合内(不以字母打头)的归于无效等价类。

(3) 如果输入条件是一个布尔量,则可以确定一个有效等价类和一个无效等价类。

(4) 如果规定了输入数据的一组值,而且程序要对每个输入值分别进行处理。这时可为 每一个输入值确立一个有效等价类,此外针对这组值确立一个无效等价类,它是所有不允许的输入值的集合。例如,在教师分房方案中规定对教授、副教授、讲师和助教分别计算分数,作相应的处理。因此可以确定 4 个有效等价类为教授、副教授、讲师和助教,以及一个无效等价类,它是所有不符合以上身分的人员的输入值的集合。

(5) 如果规定了输入数据必须遵守的规则,则可以确立一个有效等价类(符合规则)和若干个无效等价类(从不同角度违反规则)。例如,PASCAL 语言处理时规定“ 一个语句

必须以分号‘；’结束”。这时,可以确定一个有效等价类“以‘；’结束”,若干个无效等价类“以‘：’结束”、“以‘,’结束”、“以‘ ’结束”、“以LF结束”等等。

(6) 如果我们确知,已划分的等价类中各元素在程序中的处理方式不同,则应将此等价类进一步划分成更小的等价类。

2) 确立测试用例

在确立了等价类之后,建立等价类表,列出所有划分出的等价类:

输入条件	有效等价类	无效等价类
.....
.....

再从划分出的等价类中按以下原则选择测试用例。

- (1) 为每一个等价类规定一个唯一的编号;
- (2) 设计一个新的测试用例,使其尽可能多地覆盖尚未被覆盖的有效等价类,重复这一步,直到所有的有效等价类都被覆盖为止;
- (3) 设计一个新的测试用例,使其仅覆盖一个尚未被覆盖的无效等价类,重复这一步,直到所有的无效等价类都被覆盖为止。

之所以要这样作,是因为某些程序中对某一输入错误的检查往往会屏蔽对其它输入错误的检查。例如学校领导分为校长、副校长、书记、副书记,年龄(AGE)在 25 AGE 75。若给出一个无效等价类的测试用例为(妇联主任,5岁),它覆盖了两个错误的输入条件(职务,年龄),但当程序检查到职务时发现了错误,就可能不再去检查年龄错误,因此必须针对每一个无效等价类,分别设计测试用例。

3) 用等价类划分法设计测试用例的实例

在某一 PASCAL 语言版本中规定:“标识符是由字母开头、后跟字母或数字的任意组合构成。有效字符数为 8 个,最大字符数为 80 个。”并且规定:“标识符必须先说明,再使用。”在同一说明语句中,标识符至少必须有一个。”

为用等价类划分的方法得到上述规格说明所规定的要求,本着前述的划分原则,建立输入等价类表,如表 7.1 所示:

表 7.1 等价类表

输入条件	有效等价类	无效等价类
标识符个数	1 个 (1), 多个 (2)	0 个 (3)
标识符字符数	1 ~ 8 个 (4)	0 个 (5), > 8 个 (6), > 80 个 (7)
标识符组成	字母 (8), 数字 (9)	非字母数字字符 (10), 保留字 (11)
第一个字符	字母 (12)	非字母 (13)
标识符使用	先说明后使用 (14)	未说明已使用 (15)

下面选取了 9 个测试用例,它们覆盖了所有的等价类。

```
VAR x, T1234567: REAL;           } (1), (2), (4), (8), (9), (12), (14)
  BEGIN x:= 3. 414; T1234567:= 2. 732;.....
VAR :REAL;                       } (3)
```

```

VAR x,:REAL;                } ( 5)
VAR T12345678:REAL;         } ( 6)
VAR T12345.....:REAL;      } ( 7)
多于 80 个字符
VAR T$:CHAR;                } (10)
VAR GOTO:INTEGER;           } (11)
VAR 2T:REAL;                } (13)
VAR PAR:REAL;               } (15)
BEGIN .....
    PAP:= SIN(3.14* 0.8)/ 6;

```

7.4.2 边界值分析

1) 边界值分析方法的考虑

边界值分析也是一种黑盒测试方法,是对等价类划分方法的补充。

人们从长期的测试工作经验中得知,大量的错误是发生在输入或输出范围的边界上,而不是在输入范围的内部。因此针对各种边界情况设计测试用例,可以查出更多的错误。比如,在作三角形计算时,要输入三角形的三个边长: A, B 和 C。应注意到这三个数值应当满足 $A > 0, B > 0, C > 0, A + B > C, A + C > B, B + C > A$,才能构成三角形。但如果把六个不等式中的任何一个大于号“ $>$ ”错写成大于等于号“ \geq ”,那就不能构成三角形。问题恰出现在容易被疏忽的边界附近。这里所说的边界是指,相当于输入等价类和输出等价类而言,稍高于其边界值及稍低于其边界值的一些特定情况。

使用边界值分析方法设计测试用例,首先应确定边界情况。通常输入等价类与输出等价类的边界,就是应着重测试的边界情况。应当选取正好等于、刚刚大于、或刚刚小于边界的值作为测试数据,而不是选取等价类中的典型值或任意值作为测试数据。

2) 选择测试用例的原则

边界值分析方法选择测试用例的原则在很多方面与等价类划分方法类似。

(1) 如果输入条件规定了值的范围,则应取刚达到这个范围的边界的值,以及刚刚超越这个范围边界的值作为测试输入数据。例如,若输入值的范围是“ $-1.0 \sim 1.0$ ”,则可选取“ -1.0 ”,“ 1.0 ”,“ -1.001 ”,“ 1.001 ”作为测试输入数据。

(2) 如果输入条件规定了值的个数,则用最大个数、最小个数、比最大个数多 1、比最小个数少 1 的数作为测试数据。例如,一个输入文件可有 $1 \sim 255$ 个记录,则可以分别设计有 1 个记录、255 个记录以及 0 个记录和 256 个记录的输入文件。

(3) 根据规格说明的每个输出条件,使用前面的原则 (1)。例如,某程序的功能是计算折扣量,最低折扣量是 0 元,最高折扣量是 1050 元。则设计一些测试用例,使它们恰好产生 0 元和 1050 元的结果。此外,还可考虑设计结果为负值或大于 1050 元的测试用例。由于输入值的边界不与输出值的边界相对应,所以要检查输出值的边界不一定可能,要产生超出输出值值域之外的结果也不一定办得到。尽管如此,必要时还需一试。

(4) 根据规格说明的每个输出条件,使用前面的原则 (2)。例如,一个信息检索系统根据用户打入的命令,显示有关文献的摘要,但最多只显示 4 篇摘要。这时可设计一些测

试用例,使得程序分别显示 1 篇、4 篇、0 篇摘要,并设计一个有可能使程序错误地显示 5 篇摘要的测试用例。

(5) 如果程序的规格说明给出的输入域或输出域是有序集合(如有序表,顺序文件等),则应选取集合的第一个元素和最后一个元素作为测试用例。

(6) 如果程序中使用了一个内部数据结构,则应当选择这个内部数据结构的边界上的值作为测试用例。例如,如果程序中定义了一个数组,其元素下标的下界是 0,上界是 100,那么应选择达到这个数组下标边界的值,如 0 与 100,作为测试用例。

(7) 分析规格说明,找出其它可能的边界条件。

3) 应用边界值分析方法设计测试用例的实例

举例说明,如果已经编制了一个为学生标准化考试批阅试卷、产生成绩报告的程序。其规格说明如下。

程序的输入文件由一些有 80 个字符的记录(卡片)组成。输入数据记录格式如图 7.9 所示。

图 7.9 学生考卷评分和成绩统计程序输入数据形式

所有这些记录分为三组。

标题。这一组只有一个记录。其内容是成绩报告的名字。

各题的标准答案。每个记录均在第 80 个字符处标以数字“ 2 ”。该组的第 1 个记录

的第 1 ~ 3 个字符为试题数(取值为 1 ~ 999)。第 10 ~ 59 个字符给出第 1 ~ 50 题的标准答案 (每个合法字符表示一个答案)。该组的第 2、第 3, ... 个记录相应为第 51 ~ 100 题、第 101 ~ 150 题, ... 题的标准答案。

学生的答卷。每个记录均在第 80 个字符处标以数字“ 3 ”。每个学生的答卷在若干个记录中给出。比如, 某甲的首记录第 1 ~ 9 个字符给出学生的学号, 第 10 ~ 59 个字符列出的是某甲所作的第 1 ~ 50 题的解答。若试题数超过 50, 则其第二、第三, ... 个记录分别给出他的第 51 ~ 100 题、第 101 ~ 150,题的解答。然后是某乙的答卷记录。学生人数不超过 200 人, 试题个数不超过 999。程序的输出有 4 个报告。

按学号排列的成绩单, 列出每个学生的成绩 (百分制)、名次;

按学生成绩排序的成绩单;

平均分数及标准偏差的报告;

试题分析报告。按试题号排列, 列出各题学生答对的百分比。

下面分别考虑输入条件和输出条件、以及边界条件, 选择测试用例。

输入条件	测 试 用 例
输入文件	[空输入文件]
标题	[没有标题记录] [标题只有一个字符] [标题有 80 个字符]
试题数	[试题数为 1] [试题数为 50] [试题数为 51] [试题数为 100] [试题数为 999] [试题数为 0] [试题数含有非数字字符]
标准答案记录	[没有标准答案记录, 有标题] [标准答案记录多一个] [标准答案记录少一个]
学生人数	[0 个学生] [1 个学生] [200 个学生] [201 个学生]
学生答题	[某学生只有一个回答记录, 但有两个标准答案记录] [该学生是文件中的第一个学生] [该学生是文件中最后一个学生(记录数出错的学生)]
学生答题	[某学生有两个回答记录, 但只有一个标准答案记录] [该学生是文件中第一个学生(指记录数出错的学生)] [该学生是文件中最后一个学生]
输出条件	测 试 用 例
学生成绩	[所有学生的成绩都相等] [每个学生的成绩都互不相同] [部分(不是全体)学生的成绩相同(检查是否能按成绩正确排名次)] [有个学生得 0 分] [有个学生得 100 分]
输出报告	[有个学生的学号最小(检查按学号排序是否正确)] [有个学生的学号最大(检查按学号排序是否正确)] [适当的学生人数, 使产生的报告刚好印满一页(检查打印页数)] [学生人数比刚才多出 1 人(检查打印换页)]
输出报告	[平均成绩为 100 分(所有学生都得满分)] [平均成绩为 0 分(所有学生都得 0 分)] [标准偏差为最大值(有一半学生得 0 分, 其他 100 分)] [标准偏差为 0 (所有学生的成绩都相等)]
输出报告	[所有学生都答对了第一题] [所有学生都答错了第一题] [所有学生都答对了最后一题] [所有学生都答错了最后一题] [选择适当的试题数, 使第四个报告刚好打满一页] [试题数比刚才多 1 题, 使报告打满一页后, 刚好剩下一题未打]

上述 43 个测试用例可以发现在程序中大部分常见的错误。如果用随机方法设计测试用例不一定会发现这些错误。如果使用得当,边界值分析方法是很有效的。

这个方法看起来似乎很简单,但是由于许多程序中的边界情况很复杂,要找出适当的测试用例还需针对问题的输入域、输出域边界,耐心细致地逐个考虑。

7.4.3 错误推测法

人们也可以靠经验和直觉推测程序中可能存在的各种错误,从而有针对性地编写检查这些错误的例子。这就是错误推测法。

错误推测法的基本想法是:列举出程序中所有可能有的错误和容易发生错误的特殊情况,根据它们选择测试用例。例如,在介绍单元测试时曾列出许多在模块中常见的错误,这些是单元测试经验的总结。此外,对于在程序中容易出错的情况,也有一些经验总结出来。例如,输入数据为 0,或输出数据为 0 是容易发生错误的情形,因此可选择输入数据为 0,或使输出数据为 0 的例子作为测试用例。又例如,输入表格为空或输入表格只有一行,也是容易发生错误的情况。可选择表示这种情况的例子作为测试用例。再例如,若两个模块间有共享变量,则要设计测试用例检查当让一个模块去修改这个共享变量的内容后,另一个模块的出错情况等等。

在介绍边界值分析方法时,举了一个“批阅学生考卷,给出成绩报告”的例子。现在用错误推测法还可以补充设计一些测试用例:程序是否把空格作为回答(即学生没有作某道题);在回答记录中混有标准答案记录;除了标题记录外,还有一些记录的最后一个字符既不是“2”,也不是“3”;有两个学生的学号相同;试题数是负值。

7.4.4 因果图

1) 因果图的适用范围

前面介绍的等价类划分方法和边界值分析方法,都是着重考虑输入条件,但未考虑输入条件之间的联系。如果在测试时考虑输入条件的各种组合,可能又会产生一些新情况。例如,在边界值分析方法举例时提到的“学生标准化考试评阅试卷、产生成绩报告”的程序,还应检查试题数与学生人数的乘积是否会超出存储容量,而用边界值分析方法则无法检查程序在这类问题上是否有错误。

所有输入条件之间的组合情况往往相当多。必须考虑使用一种适合于描述对于多种条件的组合,相应产生多个动作的形式来考虑设计测试用例,这就需要利用因果图。因果图方法最终生成的是判定表。它适合于检查程序输入条件的各种组合情况。

2) 用因果图生成测试用例的基本步骤

(1) 分析软件规格说明描述中,哪些是原因(即输入条件或输入条件的等价类),哪些是结果(即输出条件),并给每个原因和结果赋予一个标识。

(2) 分析软件规格说明描述中的语义,找出原因与结果之间、原因与原因之间对应的是什么关系?根据这些关系,画出因果图。

(3) 由于语法或环境限制,有些原因与原因之间、原因与结果之间的组合情况不可能出现。为表明这些特殊情况,在因果图上用一些记号标明约束或限制条件。

- (4) 把因果图转换成判定表。
 - (5) 把判定表的每一列拿出来作为依据, 设计测试用例。
- 3) 在因果图中出现的基本符号

图 7. 10 因果图的图形符号

通常在因果图中用 C_i 表示原因, 用 E_i 表示结果, 其基本符号如图 7. 10 所示。主要的原因和结果之间的关系有:

- (a) 恒等: 表示原因与结果之间一对一的对应关系。若原因出现, 则结果出现。若原因不出现, 则结果也不出现。
- (b) 非: 表示原因与结果之间的一种否定关系。若原因出现, 则结果不出现。若原因不出现, 反而结果出现。
- (c) 或 (): 表示若几个原因中有一个出现, 则结果出现, 只有当这几个原因都不出现时, 结果才不出现。
- (d) 与 (): 表示若几个原因都出现, 结果才出现。若几个原因中有一个不出现, 结果就不出现。

4) 表示约束条件的符号

为了表示原因与原因之间、结果与结果之间可能存在的约束条件, 在因果图中可以附加一些表示约束条件的符号。若从输入(原因)考虑, 有以下四种约束, 参看图 7. 11。

- 输入 (1) E (互斥): 表示 a, b 两个原因不会同时成立, 两个中最多有一个可能成立。
- (2) I (包含): 表示 a, b, c 三个原因中至少有一个必须成立。
- (3) O (唯一): 表示 a 和 b 当中必须有一个, 且仅有一个成立。
- (4) R (要求): 表示当 a 出现时, b 必须也出现。不可能 a 出现, b 不出现。
- 输出 (5) M (屏蔽): 表示当 a 是 1 时, b 必须是 0。而当 a 为 0 时, b 的值不定。

图 7. 11 因果图的约束符号

5) 利用因果图设计测试用例的实例

[例] 有一个处理单价为 5 角钱的饮料的自动售货机软件测试用例的设计。规格说

明为:“ 若投入 5 角钱或 1 元钱的硬币, 押下 橙汁 或 啤酒 的按钮, 则相应的饮料就送出来。若售货机没有零钱找, 则一个显示 零钱找完 的红灯亮, 这时在投入 1 元硬币并押下按钮后, 饮料不送出来而且 1 元硬币也退出来; 若有零钱找, 则显示 零钱找完 的红灯灭, 在送出饮料的同时退还 5 角硬币。”

(1) 分析这一段说明, 列出原因和结果

原 因	1. 售货机有零钱找	结 果	21. 售货机 零钱找完 灯亮
	2. 投入 1 元硬币		22. 退还 1 元硬币
	3. 投入 5 角硬币		23. 退还 5 角硬币
	4. 押下橙汁按钮		24. 送出橙汁饮料
	5. 押下啤酒按钮		25. 送出啤酒饮料

(2) 画出因果图, 如图 7.12 所示。所有原因结点列在左边, 所有结果结点列在右边。建立两个中间结点, 表示处理的中间状态。

- 中间结点:
11. 投入 1 元硬币且押下饮料按钮

12. 押下 橙汁 或 啤酒 的按钮

13. 应当找 5 角零钱并且售货机有零钱找

14. 钱已付清

图 7.12 因果图

- (3) 由于 2 与 3, 4 与 5 不能同时发生, 分别加上约束条件 E。
- (4) 因果图(图 7.12)
- (5) 转换成判定表

在判定表中, 阴影部分表示因违反约束条件的不可能出现的情况, 删去。第 16 列与第 32 列因什么动作也没做, 也删去。最后可根据剩下的 16 列作为确定测试用例的依据。

图 7.13 由因果图得到的判定表

7.5 软件测试的策略

测试过程按 4 个步骤进行,即单元测试、组装测试、确认测试和系统测试。图 7.14 示出软件测试经历的 4 个步骤。

图 7.14 软件测试的过程

开始是单元测试,集中对用源代码实现的每一个程序单元进行测试,检查各个程序模块是否正确地实现了规定的功能。然后,把已测试过的模块组装起来,进行组装测试,主要对与设计相关的软件体系结构的构造进行测试。为此,在将一个一个实施了单元测试并确保无误的程序模块组装成软件系统的过程中,对正确性和程序结构等方面进行检查。确认测试则是要检查已实现的软件是否满足了需求规格说明中确定了的各种需求,以及软件配置是否完全、正确。最后是系统测试,把已经经过确认的软件纳入实际运行环境中,与其它系统成分组合在一起进行测试。严格地说,系统测试已超出了软件工程的范围。

7.5.1 单元测试 (unit testing)

单元测试又称模块测试,是针对软件设计的最小单位k —程序模块,进行正确性检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试。

1) 单元测试的内容

在单元测试时,测试者需要依据详细设计说明书和源程序清单,了解该模块的 I/O 条件和模块的逻辑结构,主要采用白盒测试的测试用例,辅之以黑盒测试的测试用例,使之对任何合理的输入和不合理的输入,都能鉴别和响应。这要求对所有的局部的和全局的数据结构、外部接口和程序代码的关键部分,都要进行桌前检查和严格的代码审查。在单元测试中进行的测试工作如图 7.15 所示,需要在五个方面对被测模块进行检查。

图 7.15 单元测试的工作

(1) 模块接口测试

在单元测试的开始,应对通过被测模块的数据流进行测试。如果数据不能正确地输入和输出,就谈不上进行其它测试。为此,对模块接口可能需要如下的测试项目:调用本模块时的输入参数与模块的形式参数的匹配情况;本模块调用子模块时,它输入给子模块的参数与子模块中的形式参数的匹配情况;是否修改了只作输入用的形式参数;全局量的定义在各模块中是否一致;限制是否通过形式参数来传送。

当模块通过外部设备进行输入/输出操作时,必须附加如下的测试项目:文件属性是否正确;OPEN 语句与 CLOSE 语句是否正确;规定的 I/O 格式说明与 I/O 语句是否匹配;缓冲区容量与记录长度是否匹配;在进行读写操作之前是否打开了文件;在结束文件处理时是否关闭了文件;正文书写/输入错误以及 I/O 错误是否检查并作了处理。

(2) 局部数据结构测试

模块的局部数据结构是最常见的错误来源,应设计测试用例以检查以下各种错误:不正确或不一致的数据类型说明;使用尚未赋值或尚未初始化的变量;错误的初始值或错误的缺省值;变量名拼写错或书写错;不一致的数据类型。可能的话,除局部数据之外的全局数据对模块的影响也需要查清。

(3) 路径测试

选择适当的测试用例,对模块中重要的执行路径进行测试。应当设计测试用例查找由

于错误的计算、不正确的比较或不正常的控制流而导致的错误。对基本执行路径和循环进行测试可以发现大量的路径错误。

常见的不正确计算有: 运算的优先次序不正确或误解了运算的优先次序; 运算的方式错, 即运算的对象彼此在类型上不兼容; 算法错; 初始化不正确; 运算精度不够; 表达式的符号表示不正确等。

常见的比较和控制流错误有: 不同数据类型量的相互比较; 不正确的逻辑运算符或优先级; 因浮点数运算精度问题而造成的两值比较不等; 关系表达式中不正确的变量和比较符; “差 1”错, 即不正确地多循环一次或少循环一次; 错误的或不可能的循环终止条件; 当遇到发散的迭代时不能终止的循环; 不适当地修改了循环变量等。

(4) 错误处理测试

比较完善的模块设计要求能预见出错的条件, 并设置适当的出错处理, 以便在一旦程序出错时, 能对出错程序重作安排, 保证其逻辑上的正确性。若出现下列情况之一, 则表明模块的错误处理功能包含有错误或缺陷: 出错的描述难以理解; 出错的描述不足以对错误定位, 不足以确定出错的原因; 显示的错误与实际的错误不符; 对错误条件的处理不正确; 在对错误进行处理之前, 错误条件已经引起系统的干预等。

(5) 边界测试

在边界上出现错误是常见的。例如, 在一段程序内有一个 n 次循环, 当到达第 n 次重复时可能会出错。还有在取最大值或最小值时也容易出错。因此, 要特别注意数据流、控制流中刚好等于、大于或小于确定的比较值时出错的可能性。对这些地方要仔细地选择测试用例, 认真加以测试。

此外, 如果对模块运行时间有要求的话, 还要专门进行关键路径测试, 以确定最坏情况下和平均意义下影响模块运行时间的因素。这类信息对进行性能评价是十分有用的。

总之, 由于模块测试针对的程序规模较小, 便于查错, 而且发现错误后容易确定错误所在的位置, 便于纠错。同时多个模块可以并行测试。所以作好模块测试将可为后续的测试打下良好的基础。

2) 单元测试的步骤

通常单元测试是在编码阶段进行的。在源程序代码编制完成, 经过评审和验证, 确认没有语法错误之后, 就开始进行单元测试的测试用例设计。利用设计文档, 设计可以验证程序功能、找出程序错误的多个测试用例。对于每一组输入, 应有预期的正确结果。

模块并不是一个独立的程序, 在考虑测试模块时, 同时要考虑它和外界的联系, 用一些辅助模块去模拟与被测模块相联系的其它模块。这些辅助模块分为两种。

(1) 驱动模块(driver)——相当于被测模块的主程序。它接收测试数据, 把这些数据传送给被测模块, 最后再输出实测结果。

(2) 桩模块(stub)——也叫作存根模块。用以代替被测模块调用的子模块。桩模块可以作少量的数据操作, 不需要把子模块所有功能都带进来, 但不允许什么事情也不作。被测模块、与它相关的驱动模块及桩模块共同构成了一个“测试环境”, 见图 7. 16。驱动模块和桩模块的编写会给测试带来额外的开销。因为它们在软件交付时不作为产品的一部分一同交付, 而且它们的编写需要一定的工作量。特别是桩模块, 不能只简单地给出“曾经进

入'的信息。为了能够正确地测试软件,桩模块可能需要模拟实际子模块的功能,这样,桩模块的建立就不是很轻松了。

图 7.16 单元测试的测试环境

模块的内聚程度高,可以简化单元测试过程。如果每一个模块只完成一种功能,则需要的测试用例数目将明显减少,模块中的错误也容易预测和发现。

当然,一个模块有多种功能,且以程序包(package)的形式出现的也不少见,例如Ada中的包,C++中的类。这时可将模块看成由几个小程序组成。必须对其中的每个小程序先进行单元测试要作的工作,对关键模块还要作性能测试。对支持某些标准规则的程序,更要着手进行互联测试。有人把这种情况特别称为模块测试,以区别单元测试。

7.5.2 组装测试(integrated testing)

组装测试也叫作集成测试或联合测试。通常,在单元测试的基础上,需要将所有模块按照设计要求组装成为系统。这时需要考虑的问题是:

- 1) 在把各个模块连接起来的时候,穿越模块接口的数据是否会丢失;
- 2) 一个模块的功能是否会对另一个模块的功能产生不利的影响;
- 3) 各个子功能组合起来,能否达到预期要求的父功能;
- 4) 全局数据结构是否有问题;
- 5) 单个模块的误差累积起来,是否会放大,从而达到不能接受的程度。

因此在单元测试的同时可进行组装测试,发现并排除在模块连接中可能出现的问题,最终构成要求的软件系统。

选择什么方式把模块组装起来形成一个可运行的系统,直接影响到模块测试用例的形式、所用测试工具的类型、模块编号的次序和测试的次序、以及生成测试用例的费用和调试的费用。通常把模块组装为系统的方式有两种:一次性组装方式和增殖式组装方式。

1) 一次性组装方式(big bang)

它是一种非增殖式组装方式。也叫作整体拼装。使用这种方式,首先对每个模块分别进行模块测试,然后再把所有模块组装在一起进行测试,最终得到要求的软件系统。例如,有一个模块系统结构,如图 7.17(a)所示。其单元测试和组装顺序如图 7.17(b)所示。

在图中,模块 d₁, d₂, d₃, d₄, d₅ 是对各个模块作单元测试时建立的驱动模块, s₁, s₂, s₃, s₄, s₅ 是为单元测试而建立的桩模块。这种一次性组装方式试图在辅助模块的协助下,在分别完成模块单元测试的基础上,将被测模块连接起来进行测试。但是由于程序中不可避免地存在涉及模块间接口、全局数据结构等方面的问题,所以一次试运行成功的可能性不很大。

图 7.17 一次性组装方式

2) 增殖式组装方式

这种组装方式又称渐增式组装,首先是对一个个模块进行模块测试,然后将这些模块逐步组装成较大的系统,在组装的过程中边连接边测试,以发现连接过程中产生的问题。最后通过增殖逐步组装成为要求的软件系统。

(1) 自顶向下的增殖方式

这种组装方式是将模块按系统程序结构,沿控制层次自顶向下进行组装。其步骤如下:

以主模块为被测模块兼驱动模块,所有直属于主模块的下属模块全部用桩模块代替,对主模块进行测试。

采用深度优先(参看图 7.18)或宽度优先的策略,用实际模块替换相应桩模块,再用桩模块代替它们的直接下属模块,与已测试的模块或子系统组装成新的子系统。

进行回归测试(即重新执行以前作过的全部测试或部分测试),排除组装过程中引入新的错误的可能。

判断是否所有的模块都已组装到系统中?是则结束测试,否则转到 去执行。

图 7.18 自顶向下增殖方式的例子

自顶向下的增殖方式在测试过程中较早地验证了主要的控制和判断点。在一个功能划分合理的程序模块结构中,判断常常出现在较高的层次里,因而较早就能遇到。如果主要控制有问题,尽早发现它能够减少以后的返工。如果选用按深度方向组装的方式,可以首先实现和验证一个完整的软件功能。

自顶向下的组装和测试存在一个逻辑次序问题。在为了充分测试较高层的处理而需要较低层处理的信息时,就会出现这类问题。在自顶向下组装阶段,还需要用桩模块代替较低层的模块,所以关于桩模块的编写,根据不同情况可能有如图 7.19 所示的几种选择。

为了能够准确地实施测试,应当让桩模块正确而有效地模拟子模块的功能和合理的接口,不能是只包含返回语句或只显示该模块已调用信息,不执行任何功能的哑模块。

图 7.19 桩模块的几种选择

(2) 自底向上的增殖方式

这种组装的方式是从程序模块结构的最底层的模块开始组装和测试。因为模块是自底向上进行组装,对于一个给定层次的模块,它的子模块(包括子模块的所有下属模块)已经组装并测试完成,所以不再需要桩模块。在模块的测试过程中需要从子模块得到的信息可以由直接运行子模块得到。

图 7.20 自底向上增殖方式的例子

自底向上增殖的步骤如下:

- 由驱动模块控制最底层模块的并行测试;也可以把最底层模块组合成实现某一特定软件功能的簇,由驱动模块控制它进行测试。
- 用实际模块代替驱动模块,与它已测试的直属子模块组装成为子系统。
- 为子系统配备驱动模块,进行新的测试。
- 判断是否已组装到达主模块。是则结束测试,否则执行 。

以图 7.17(a)所示的系统结构为例,用图 7.20 说明自底向上组装和测试的顺序。

自底向上进行组装和测试时,需要为被测模块或子系统编制相应的驱动模块。常见的几种类型的驱动模块如图 7.21 所示。

随着组装层次的向上移动,驱动模块将大为减少。如果对程序模块结构的最上面两层模块采用自顶向下进行组装和测试,可以明显地减少驱动模块的数目,而且可以大大减少把几个子系统组装起来所需要作的工作。

(3) 混合增殖式测试

图 7.21 驱动模块的几种选择

自顶向下增殖的方式和自底向上增殖的方式各有优缺点。一般来讲,一种方式的优点是另一种方式的缺点。

自顶向下增殖方式的缺点是需要建立桩模块。要使桩模块能够模拟实际子模块的功能将是十分困难的,因为桩模块在接收了被测模块发送的信息后需要按照它所代替的实际子模块功能返回应该回送的信息,这必将增加建立桩模块的复杂度。同时涉及复杂算法和真正输入/输出的模块一般在底层,它们是最容易出问题的模块,到组装和测试的后期才遇到这些模块,一旦发现问题,导致过多的回归测试。而自顶向下增殖方式的优点是能够较早地发现在主要控制方面的问题。

自底向上增殖方式的缺点是“程序一直未能作为一个实体存在,直到最后一个模块加上后才形成一个实体”。也就是说,在自底向上组装和测试的过程中,对主要的控制直到最后才接触到。但这种方式的优点是不需要桩模块,而建立驱动模块一般比建立桩模块容易,同时由于涉及到复杂算法和真正输入/输出的模块最先得得到组装和测试,可以把最容易出问题的部分在早期解决。此外自底向上增殖的方式可以实施多个模块的并行测试,以提高测试效率。

鉴于此,通常是把以上两种方式结合起来进行组装和测试。下面简单介绍三种常见的综合的增殖方式。

衍变的自顶向下的增殖测试:它的基本思想是强化对输入/输出模块和引入新算法模块的测试,并自底向上组装成为功能相当完整且相对独立的子系统,然后由主模块开始自顶向下进行增殖测试。

自底向上一自顶向下的增殖测试:它首先对含读操作的子系统自底向上直至根结点模块进行组装和测试,然后对含写操作的子系统作自顶向下的组装与测试。

回归测试:这种方式采取自顶向下的方式测试被修改的模块及其子模块,然后将这一部分视为子系统,再自底向上测试,以检查该子系统与其上级模块的接口是否适配。

3) 组装测试的组织和实施

组装测试是一种正规测试过程,必须精心计划,并与单元测试的完成时间协调起来。在制定测试计划时,应考虑如下因素:

- (1) 采用何种系统组装方法进行组装测试。
- (2) 组装测试过程中连接各个模块的顺序。
- (3) 模块代码编制和测试进度是否与组装测试的顺序一致。
- (4) 测试过程中是否需要专门的硬件设备。

解决了上述问题之后,就可以列出各个模块的编制、测试计划表,标明每个模块单元测试完成的日期、首次组装测试的日期、组装测试全部完成的日期、以及需要的测试用例和所期望的测试结果。

在完成预定的组装测试工作之后,测试小组应负责对测试结果进行整理、分析,形成测试报告。测试报告中要记录实际的测试结果、在测试中发现的问题、解决这些问题的方法以及解决之后再次测试的结果。此外还应提出目前不能解决、还需要管理人员和开发人员注意的一些问题,提供测试评审和最终决策,以提出处理意见。

7.5.3 确认测试 (validation testing)

确认测试又称有效性测试。它的任务是验证软件的有效性,即验证软件的功能和性能及其它特性是否与用户的要求一致。对软件的功能和性能要求在软件需求规格说明书中已经明确规定。

在确认测试阶段需要作的工作如图 7.22 所示。首先要进行有效性测试以及软件配置复审,然后进行验收测试和安装测试,在通过了专家鉴定之后,才能成为可交付的软件。

图 7.22 确认测试的步骤

1) 进行有效性测试(黑盒测试)

有效性测试是在模拟的环境(可能就是开发的环境)下,运用黑盒测试的方法,验证被测软件是否满足需求规格说明书列出的需求。为此,需要首先制定测试计划,规定要作测试的种类。还需要制定一组测试步骤,描述具体的测试用例。通过实施预定的测试计划和测试步骤,确定软件的特性是否与需求相符,确保所有的软件功能需求都能得到满足,所有的软件性能需求都能达到,所有的文档都正确且便于使用。同时,对其它软件需求,例如可移植性、兼容性、出错自动恢复、可维护性等,也都要进行测试,确认是否满足。

2) 软件配置复查

软件配置复查的目的是保证软件配置的所有成分都齐全,各方面的质量都符合要求,具有维护阶段所必须的细节,而且已经编排好分类的目录。

除了按合同规定的内容和要求,由人工审查软件配置之外,在确认测试的过程中,应当严格遵守用户手册和操作手册中规定的使用步骤,以便检查这些文档资料的完整性和正确性。必须仔细记录发现的遗漏和错误,并且适当地补充和改正。软件配置请参看有关软件配置管理的章节。

3) 测试和 测试

在软件交付使用之后,用户将如何实际使用程序,对于开发者来说是无法预测的。因为用户在使用过程中常常会发生对使用方法的误解、异常的数据组合、以及产生对某些用户来说似乎是清晰的但对另一些用户来说却难以理解的输出等等。

当软件是为特定用户开发的时候,需要进行一系列的验收测试,让用户验证所有的需求是否已经满足。这些测试是以用户为主,而不是以系统开发者为主进行的。验收测试可以是一次简单的非正式的“测试运行”,也可以是一组复杂的有组织有计划的测试活动。事实上,验收测试可能持续几个星期到几个月。

如果软件是为多个用户开发的产品,让每个用户逐个执行正式的验收测试是不切实际的。很多软件产品生产者采用一种称之为 测试和 测试的测试方法,以发现可能只有最终用户才能发现的错误。

测试是由一个用户在开发环境下进行的测试,也可以是公司内部的用户在模拟实际操作环境下进行的测试。软件在一个自然设置状态下使用。开发者坐在用户旁边,随时记下错误情况和使用中的问题。这是在受控制的环境下进行的测试。测试的目的是评价软件产品的 FLURPS (即功能、局域化、可使用性、可靠性、性能和支持)。尤其注重产品的界面和特色。测试人员是除产品开发人员之外首先见到产品的人,他们提出的功能和修改意见是特别有价值的。测试可以从软件产品编码结束之时开始,或在模块(子系统)测试完成之后开始,也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。有关的手册(草稿)等应事先准备好。

测试是由软件的多个用户在一个或多个用户的实际使用环境下进行的测试。这些用户是与公司签定了支持产品预发行合同的外部客户,他们要求使用该产品,并愿意返回有关错位错误信息给开发者。与 测试不同的是,开发者通常不在测试现场。因而,测试是在开发者无法控制的环境下进行的软件现场应用。在 测试中,由用户记下遇到的所有问题,包括真实的以及主观认定的,定期向开发者报告,开发者在综合用户的报告之后,作出修改,最后将软件产品交付给全体用户使用。测试主要衡量产品的 FLURPS。着重于产品的支持性,包括文档、客户培训和支持产品生产能力。只有当 测试达到一定的可靠程度时,才能开始 测试。由于它处在整个测试的最后阶段,不能指望这时发现主要问题。同时,产品的所有手册文本也应该在此阶段完全定稿。

由于 测试的主要目标是测试可支持性,所以 测试应尽可能由主持产品发行的人员管理。

4) 验收测试 (acceptance testing)

在通过了系统的有效性测试及软件配置审查之后,应开始系统的验收测试。验收测试是以用户为主的测试。软件开发人员和 QA(质量保证)人员也应参加。由用户参加设计测试用例,使用用户界面输入测试数据,并分析测试的输出结果。一般使用生产中的实际数

据进行测试。在测试过程中,除了考虑软件的功能和性能外,还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。

验收测试实际上是对整个测试计划进行的一种“走查(walkthrough)”。

5) 确认测试的结果

在全部确认测试的测试用例运行完后,所有的测试结果可以分为两类:

(1) 测试结果与预期的结果相符,这说明软件的这部分功能或性能特征与需求规格说明书相符合,从而这部分程序可以接受。

(2) 测试结果与预期的结果不符,这说明软件的这部分功能或性能特征与需求规格说明不一致,因此需要开列一张软件各项缺陷表或软件问题报告,通过与用户的协商,解决所发现的缺陷和错误。

7.5.4 系统测试(system testing)

所谓系统测试,是将通过确认测试的软件,作为整个基于计算机系统的一个元素,与计算机硬件、外设、某些支持软件、数据和人员等其它系统元素结合在一起,在实际运行(使用)环境下,对计算机系统进行一系列的组装测试和确认测试。

系统测试的目的在于通过与系统的需求定义作比较,发现软件与系统定义不符合或与之矛盾的地方。系统测试的测试用例应根据需求分析说明书设计,并在实际使用环境下运行。

7.5.5 测试的步骤及相应的测试种类

软件测试实际上是由一系列不同的测试组成。尽管每种测试各有不同的目的,但是所有的工作都是为了证实所有的系统元素组装正确,并且执行着为各自分配的功能。下面着重介绍几种软件的测试及它们与各个测试步骤中的关系,参看图 7.23。

下面给出图 7.23 中各类测试的定义。

1) 功能测试(function testing): 功能测试是在规定的一段时间内运行软件系统的所有功能,以验证这个软件系统有无严重错误。

2) 回归测试(regression testing): 这种测试用于验证对软件修改后有没有引出新的错误,或者说,验证修改后的软件是否仍然满足系统的需求规格说明。

3) 可靠性测试(reliability testing): 如果系统需求说明书中有对可靠性的要求,则需进行可靠性测试。通常使用平均失效间隔时间 MTBF 与因故障而停机的时间 MTTR 来度量系统的可靠性。

4) 强度测试(stress testing): 强度测试是要检查在系统运行环境不正常到发生故障的情况下,系统可以运行到何种程度的测试。因此进行强度测试,需要提供非正常数量、频率或总量资源来运行系统。

强度测试的一个变种是敏感性测试。在数学算法中经常可以看到,在程序有效数据界限内一个非常小的范围内的一组数据可能引起极端的或不平稳的错误处理出现,或者导致极度的性能下降的情况发生。因此利用敏感性测试以发现在有效输入类中可能引起某种不稳定性或不正常处理的某些数据的组合。

说明: M= 必要的(mandatory) H= 积极推荐(highly recommended)
S= 建议使用(suggested)

图 7.23 各测试步骤中的测试种类

5) 性能测试(performance testing): 性能测试是要检查系统是否满足在需求说明书中规定的性能。特别是对于实时系统或嵌入式系统, 软件只满足要求的功能而达不到要求的性能是不行的。所以还需要进行性能测试。性能测试可以出现在测试过程的各个阶段, 甚至在单元层次上, 也可以进行性能测试。这时, 不但需要对单个程序的逻辑进行白盒测试(结构测试), 还可以对程序的性能进行评估。然而, 只有当所有系统的元素全部组装完毕, 系统性能才能完全确定。

6) 恢复测试(recovery testing): 恢复测试是要证实在克服硬件故障(包括掉电、硬件或网络出错等) 后, 系统能否正常地继续进行工作, 并不对系统造成任何损害。为此, 可采用各种人工干预的手段, 模拟硬件故障, 故意造成软件出错。并由此检查:

错误探测功能——系统能否发现硬件失效与故障;

能否切换或启动备用的硬件;

在故障发生时能否保护正在运行的作业和系统状态;

在系统恢复后能否从最后记录下来的无错误状态开始继续执行作业等。

如果系统的恢复是自动的(由系统自身执行),则应对重新初始化、数据恢复、重新启动等逐个进行正确性评价。如果恢复需要人工干预,就需要对修复的平均时间进行评估以判定它是否在允许的范围之内。

7) 启动/停止测试(startup/shutdown testing):这类测试的目的是验证在机器启动及关机阶段,软件系统正确处理的能力。这类测试包括反复启动软件系统(例如,操作系统自举、网络的启动、应用程序的调用等),及在尽可能多的情况下关机。

8) 配置测试(configuration testing):这类测试是要检查计算机系统内各个设备或各种资源之间的相互联结和功能分配中的错误。它主要包括以下几种:

配置命令测试:验证全部配置命令的可操作性(有效性);特别对最大配置和最小配置要进行测试。软件配置和硬件配置都要测试。

循环配置测试:证明对每个设备物理与逻辑的、逻辑与功能的每次循环置换配置都能正常工作。

修复测试:检查每种配置状态及哪个设备是坏的。并用自动的或手工的方式进行配置状态间的转换。

9) 安全性测试(security testing):系统的安全性测试是要检验在系统中已经存在的系统安全性、保密性措施是否发挥作用、有无漏洞。为此要了解破坏安全性的方法和工具,并设计一些模拟测试用例对系统进行测试,力图破坏系统的保护机构以进入系统。

假如有充分的时间和资源,好的安全性测试最终应当能突破保护,进入系统。因此,系统设计者的任务应该是尽可能增大进入的代价,使进入的代价比进入系统后能得到的好处还要大。

10) 可使用性测试(usability testing):可使用性测试主要从使用的合理性和方便性等角度对软件系统进行检查,发现人为因素或使用上的问题。要保证在足够详细的程度下,用户界面便于使用;对输入量可容错、响应时间和响应方式合理可行、输出信息有意义、正确并前后一致;出错信息能够引导用户去解决问题;软件文档全面、正规、确切;如果产品销往国外,要有足够的译本。由于衡量可使用性有一定的主观因素,因此必须以原型化方法等获得的用户反馈作为依据。

11) 可支持性测试(supportability testing):这类测试是要验证系统的支持策略对于公司与用户方面是否切实可行。它所采用的方法是试运行支持过程(如对有错部分打补丁的过程,热线界面等),对其结果进行质量分析,评审诊断工具、维护过程、内部维护文档;衡量修复一个明显错误所需的平均最少时间。还有一种常用的方法是,在发行前把产品交给用户,向用户提供支持服务的计划,从用户处得到对支持服务的反馈。

12) 安装测试(installation testing):安装测试的目的不是找软件错误,而是找安装错误。在安装软件系统时,会有多种选择。要分配和装入文件与程序库,布置适用的硬件配置,进行程序的联结。而安装测试是要找出在这些安装过程中出现的错误。

在一些大型的系统中,部分工作由软件自动完成,其它工作则需由各种人员,包括操

作员、数据库管理员、终端用户等,按一定规程同计算机配合,靠人工来完成。指定由人工完成的过程也需经过仔细的检查,这就是所谓的过程测试(Procedure Testing)。

13) 互连测试(interoperability testing):互连测试是要验证两个或多个不同的系统之间的互连性。这类测试对支持标准规格说明,或承诺支持与其它系统互连的软件系统有效。例如,HP公司的文件传送存取方法FTAM,Honeywell公司NS/9000机器上的FTAM与NFT可以互连。

14) 兼容性测试(compatibility testing):这类测试主要想验证软件产品在不同版本之间的兼容性。有两类基本的兼容性测试:向下兼容和交错兼容。向下兼容测试是测试软件新版本保留它早期版本的功能的情况;交错兼容测试是要验证共同存在的两个相关但不同的产品之间的兼容性。

15) 容量测试(volume testing):容量测试是要检验系统的能力最高能达到什么程度。例如,对于编译程序,让它处理特别长的源程序;对于操作系统,让它作业队列“满员”;对于有多个终端的分时系统,让它所有的终端都开动;对于信息检索系统,让它使用频率达到最大。在使系统的全部资源达到“满负荷”的情形下,测试系统的承受能力。

16) 文档测试(documentation testing):这种测试是检查用户文档(如用户手册)的清晰性和精确性。用户文档中所使用的例子必须在测试中一一试过,确保叙述正确无误。

7.6 人工测试

人工测试不要求在计算机上实际执行被测程序,而是以一些人工的模拟技术和一些类似动态分析所使用的方法对程序进行分析和测试。

7.6.1 静态分析

静态分析主要是对源程序进行静态分析。通常采用以下方法进行。

1) 生成各种引用表

在源程序编制完成后生成各种引用表,是为了支持对源程序进行静态分析。引用表按功能分类,有以下三种。

- 直接从表中查出说明/使用错误。如循环层次表、变量交叉引用表、标号交叉引用表等。
- 为用户提供辅助信息。如,子程序(宏、函数)引用表、等价(变量、标号)表、常数表等。
- 用来作错误预测和程序复杂度计算。如,操作符和操作数的统计表等。

标号交叉引用表——它列出在各模块中出现的全部标号。在表中标出标号的属性:已说明、未说明、已使用、未使用。表中还有在模块以外的全局标号、计算标号等。

变量交叉引用表——即变量定义与引用表。在表中标明各变量的属性:已说明、未说明、隐式说明、以及类型及使用情况。进一步还可区分是否出现在赋值语句的右边,是否属于COMMON变量、全局变量或特权变量等。

子程序、宏和函数表——在表中,各个子程序、宏和函数的属性:已定义、未定义、

定义类型; 参数表: 输入参数的个数、顺序、类型; 输出参数的个数、顺序、类型; 已引用、未引用、引用次数等等。

等价表——表中列出在等价语句或等值语句中出现的全部变量和标号。

常数表——在表中列出全部数字常数和字符常数, 并指出它们在哪些语句中首先被定义, 即首先出现在哪些赋值语句的左部或哪些数据语句或参数语句中。

2) 静态错误分析

静态错误分析主要用于确定在源程序中是否有某类错误或“危险”结构。它有以下几种:

(1) 类型和单位分析: 为了发现源程序中数据类型、单位上的不一致性, 建立一些程序语言的预处理程序, 分析程序中在“下标”类型及循环控制变量方面的类型错误, 以及通过使用一般的组合/消去规则, 确定表达式的单位错误。

(2) 引用分析: 沿着程序的控制路径, 检查程序变量的引用异常问题。

- (3) 表达式分析: 对表达式进行分析, 以发现和纠正在表达式中出现的错误。包括:
- 在表达式中不正确地使用了括号造成错误;
 - 数组下标越界造成错误;
 - 除式为零造成错误;
 - 对负数开平方, 或对 求正切值造成错误;
 - 浮点数计算的误差。

(4) 接口分析: 接口的一致性错误。

- 模块之间接口的一致性和模块与外部数据库之间接口的一致性。
- 过程、函数过程之间接口的一致性。全局变量和公共数据区在使用上的一致性。

7.6.2 人工测试

静态分析中进行人工测试的主要方法有桌前检查、代码审查和走查。经验表明, 使用这种方法能够有效地发现 30% 到 70% 的逻辑设计和编码错误。

1) 桌前检查(desk checking)

这是一种传统的检查方法。由程序员自己检查自己编写的程序。程序员在程序通过编译之后, 进行单元测试设计之前, 对源程序代码进行分析、检验、并补充相关的文档, 目的是发现程序中的错误。检查项目有:

(1) 检查变量的交叉引用——检查未说明的变量和违反了类型规定的变量; 还要对照源程序逐个检查变量的引用、变量的使用序列; 临时变量在某条路径上的重写情况; 局部变量、全局变量与特权变量的使用。

(2) 检查标号的交叉引用——验证所有标号的正确性: 检查所有标号的命名是否正确; 转向指定位置的标号是否正确。

(3) 检查子程序、宏、函数——验证每次调用与被调用位置是否正确; 确认每次被调用的子程序、宏、函数是否存在; 检验调用序列中调用方式与参数的一致性。

(4) 常量检查k k 确认每个常量的取值和数制、数据类型; 检查常量每次引用同它的取值、数制和类型的一致性。

(5) 标准检查——用标准检查程序或手工检查程序中违反标准的问题。

(6) 风格检查——检查在程序设计风格方面发现的问题。

(7) 比较控制流——比较由程序员设计的控制流图和由实际程序生成的控制流图,寻找和解释每个差异,修改文档和校正错误。

(8) 选择、激活路径——在程序员设计的控制流图上选择路径,再到实际的控制流图上激活这条路径。如果选择的路径在实际控制流图上不能激活,则源程序可能有错。用这种方法激活的路径集合应保证源程序模块的每行代码都被检查,即桌前检查应完成至少是语句覆盖。

(9) 对照程序的规格说明,详细阅读源代码——程序员对照程序的规格说明书、规定的算法和程序设计语言的语法规则,仔细地阅读源代码,逐字逐句进行分析和思考,比较实际的代码和期望的代码,从它们的差异中发现程序的问题和错误。

(10) 补充文档——桌前检查的文档是一种过渡性的文档,不是公开的正式文档。通过编写文档,也是对程序的一种下意识的检查和测试,可以帮助程序员发现和抓住更多的错误。这种桌前检查,由于程序员熟悉自己的程序和自身的程序设计风格,可以节省很多的检查时间,但应避免主观片面性。

2) 代码会审(code reading review)

代码会审是由若干程序员和测试员组成一个会审小组,通过阅读、讨论和争议,对程序进行静态分析的过程。

代码会审分两步:第一步,小组负责人提前把设计规格说明书、控制流程图、程序文本及有关要求、规范等分发给小组成员,作为评审的依据。小组成员在充分阅读这些材料之后,进入审查的第二步:召开程序审查会。在会上,首先由程序员逐句讲解程序的逻辑。在此过程中,程序员或其它小组成员可以提出问题,展开讨论,审查错误是否存在。实践表明,程序员在讲解过程中能发现许多原来自己没有发现的错误,而讨论和争议则促进了问题的暴露。例如对某个局部性小问题修改方法的讨论,可能发现与之牵连的其它问题,甚至涉及到模块的功能说明、模块间接口和系统总体结构的大问题,从而导致对需求的重定义、重设计和重验证,进而大大改善了软件质量。

在会前,应当给会审小组每个成员准备一份常见错误的清单,把以往所有可能发生的常见错误罗列出来,供与会者对照检查,以提高会审的实效。

这个常见错误清单也叫作检查表,它把程序中可能发生的各种错误进行分类,对每一类列举出尽可能多的典型错误,然后把它们制成表格,供会审时使用。这种检查表类似于本章单元测试中给出的检查表。在代码会审之后,需要作以下几件事:

把发现的错误登记造表,并交给程序员;

若发现错误较多,或发现重大错误,则在改正之后,再次组织代码会审;

对错误登记表进行分析、归类、精炼,以提高审议效果。

3) 走查(walkthroughs)

走查与代码会审基本相同,其过程分为两步:

第一步把材料先发给走查小组每个成员,让他们认真研究程序,然后再开会。开会的程序与代码会审不同,不是简单地读程序和对照错误检查表进行检查,而是让与会者“充

当“计算机”。即首先由测试组成员为被测程序准备一批有代表性的测试用例,提交给走查小组。走查小组开会,集体扮演计算机角色,让测试用例沿程序的逻辑运行一遍,随时记录程序的踪迹,供分析和讨论用。

人们借助于测试用例的媒介作用,对程序的逻辑和功能提出各种疑问,结合问题开展热烈的讨论和争议,能够发现更多的问题。

7.7 调试(Debug, 排错)

软件调试则是在进行了成功的测试之后才开始的工作。它与软件测试不同,软件测试的目的是尽可能多地发现软件中的错误,但进一步诊断和改正程序中潜在的错误,则是调试的任务。调试活动由两部分组成:

确定程序中可疑错误的确切性质和位置。

对程序(设计,编码)进行修改,排除这个错误。

通常,调试工作是一个具有很强技巧性的工作。一个软件工程人员在分析测试结果的时候会发现,软件运行失效或出现问题,往往只是潜在错误的外部表现,而外部表现与内在原因之间常常没有明显的联系。如果要找出真正的原因,排除潜在的错误,不是一件易事。因此可以说,调试是通过现象找出原因的一个思维分析的过程。

7.7.1 调试的步骤

调试不是测试,但是,它是作为测试的后继工作而出现的。

调试的执行步骤如下(参看图 7.24):

图 7.24 调试的活动

- 1) 从错误的外部表现形式入手,确定程序中出错位置;
- 2) 研究有关部分的程序,找出错误的内在原因;
- 3) 修改设计和代码,以排除这个错误;
- 4) 重复进行暴露了这个错误的原始测试或某些有关测试,以确认:
该错误是否被排除;

是否引进了新的错误。

5) 如果所作的修正无效,则撤销这次改动,重复上述过程,直到找到一个有效的解决办法为止。

调试之所以困难,是由于人的心理因素以及技术方面的原因所致。从心理因素方面看,调试的能力因人而异,虽然也有经验造成的差距,但是,对于有同样教育背景与经验的程序员,他们的调试能力差别也很大。从技术角度看,查找错误的难度在于:

- 1) 现象与原因所处的位置可能相距甚远。就是说,现象可能出现在程序的一个部位,而原因可能在离此很远的另一个位置。高耦合的程序结构中这种情况更为明显。
- 2) 当其它错误得到纠正时,这一错误所表现出的现象可能会暂时消失,但并未实际排除。
- 3) 现象实际上是由一些非错误原因(例如,舍入不精确)引起的。
- 4) 现象可能是由于一些不容易发现的人为错误引起的。
- 5) 错误是由于时序问题引起的,与处理过程无关。
- 6) 现象是由于难于精确再现的输入状态(例如,实时应用中输入顺序不确定)引起。
- 7) 现象可能是周期出现的。在软、硬件结合的嵌入式系统中常常遇到。

7.7.2 几种主要的调试方法

调试的关键在于推断程序内部的错误位置及原因。为此,可以采用以下方法:

1) 强行排错

这是目前使用较多,效率较低的调试方法。它不需要过多的思考,比较省脑筋。例如:

(1) 通过内存全部打印来排错(memory dump)

将计算机存储器和寄存器的全部内容打印出来,然后在这大量的数据中寻找出错的位置。虽然有时使用它可以获得成功,但效率极低。其缺点是:

- 建立内存地址与源程序变量之间的对应关系很困难,仅汇编和手编程序才有可能。
- 人们将面对大量(八进制或十六进制)的数据,其中大多数与所查错误无关。
- 一个内存全部内容打印清单只显示了源程序在某一瞬间的状态,即所谓静态映象;但为了发现错误,需要的是程序的随时间变化的动态过程。
- 一个内存全部内容打印清单不能反映在出错位置处程序的状态。程序在出错时刻与打印信息时刻之间的时间间隔内所作的事情可能会掩盖所需要的线索。
- 缺乏从分析全部内存打印信息来找到错误原因的算法。

(2) 在程序特定部位设置打印语句

把打印语句插在出错的源程序的各个关键变量改变部位、重要分支部位、子程序调用部位,跟踪程序的执行,监视重要变量的变化。这种方法能显示出程序的动态过程,允许人们检查与源程序有关的信息。因此,比全部打印内存信息优越,但是它也有缺点:

- 可能输出大量需要分析的信息,大型程序或系统更是如此,造成费用过大。
- 必须修改源程序以插入打印语句,这种修改可能会掩盖错误,改变关键的时间关系或把新的错误引入程序。

(3) 自动调试工具

利用某些程序语言的调试功能或专门的交互式调试工具,分析程序的动态过程,而不必修改程序。

可供利用的典型语言功能有:打印出语句执行的追踪信息,追踪子程序调用,以及指定变量的变化情况。

自动调试工具的功能是:设置断点,当程序执行到某个特定的语句或某个特定的变量值改变时,程序暂停执行。程序员可在终端上观察程序此时的状态。

应用以上任一种方法之前,都应当对错误的征兆进行全面彻底的分析,得出对出错位置及错误性质的推测,再使用一种适当的排错方法来检验推测的正确性。

2) 回溯法排错

这是在小程序中常用的一种有效的排错方法。一旦发现了错误,人们先分析错误征兆,确定最先发现“症状”的位置。然后,人工沿程序的控制流程,向回追踪源程序代码,直到找到错误根源或确定错误产生的范围。例如,程序中发现错误的地方是某个打印语句。通过输出值可推断出程序在这一点上变量的值。再从这一点出发,回溯程序的执行过程,反复考虑:“如果程序在这一点上的状态(变量的值)是这样,那么程序在上一点的状态一定是这样……”,直到找到错误的位置,即在其状态是预期的点与第一个状态不是预期的点之间的程序位置。

回溯法对于小程序很有效,往往能把错误范围缩小到程序中的一小段代码;仔细分析这段代码不难确定出错的准确位置。但对于大程序,由于回溯的路径数目较多,回溯会变得很困难。

3) 归纳法排错

归纳法是一种从特殊推断一般的系统化思考方法。归纳法排错的基本思想是:从一些线索(错误征兆)着手,通过分析它们之间的关系找出错误。归纳法排错步骤大致分为以下四步:

(1) 收集有关的数据——列出所有已知的测试用例和程序执行结果。看哪些输入数据的运行结果是正确的,哪些输入数据的运行结果有错误。

(2) 组织数据——由于归纳法是从特殊到一般的推断过程,所以需要组织整理数据,以便发现规律。常用的构造线索的技术是“分类法”。用图 7.25 中所示的 3W1H 形式来组织可用的数据:

“What” 列出一般现象;

“Where” 说明发现现象的地点;

“When” 列出现象发生时所有已知情况;

“How” 说明现象的范围和量级;

而在“ Yes ”和“ No ”这两列中,“ Yes ”描述了出现错误的现象的 3W1H,“ No ”作为比较,描述了没有错误的现象的 3W1H。通过分析,找出矛盾来。

(3) 提出假设——分析线索之间的关系,利用在线索结构中观察到的矛盾现象,设计一个或多个关于出错原因的假设。如果一个假设也提不出来,归纳过程就需要收集更多的数据。此时,应当再设计与执行一些测试用例,以获得更多的数据。如果提出了许多假设,

图 7. 25 归纳法排错的步骤

则首先选用最有可能成为出错原因的假设。

(4) 证明假设——把假设与原始线索或数据进行比较, 若它能完全解释一切现象, 则假设得到证明; 否则, 认为假设不合理, 或不完全, 或是存在多个错误, 以致只能消除部分错误。有人想越过这一步, 立刻就去改正错误。这样, 假设是否合理, 是否完全, 是否同时存在多个错误都不甚清楚, 因此就不能有效地消除多个错误。

4) 演绎法排错

演绎法是一种从一般原理或前提出发, 经过排除和精化的过程推导出结论的思考方法。演绎法排错是测试人员首先根据已有的测试用例, 设想及枚举出所有可能出错的原因作为假设; 然后再用原始测试数据或新的测试, 从中逐个排除不可能正确的假设; 最后, 再用测试数据验证余下的假设确是出错的原因。演绎法主要有以下四个步骤(参看图7. 26)。

图 7. 26 演绎法排错的步骤

(1) 列举所有可能出错原因的假设——把所有可能的错误原因列成表。它们不需要完全的解释, 而仅仅是一些可能因素的假设。通过它们, 可以组织、分析现有数据。

(2) 利用已有的测试数据, 排除不正确的假设——仔细分析已有的数据, 寻找矛盾, 力求排除前一步列出所有原因。如果所有原因都被排除了, 则需要补充一些数据(测试用例), 以建立新的假设; 如果保留下来的假设多于一个, 则选择可能性最大的原因作基本的假设。

(3) 改进余下的假设——利用已知的线索, 进一步改进余下的假设, 使之更具体化, 以便可以精确地确定出错位置。

(4) 证明余下的假设——这一步极端重要, 具体作法与归纳法的第(4)步相同。

7.7.3 调试原则

在调试方面,许多原则本质上是心理学方面的问题。因为调试由两部分组成,所以调试原则也分成两组。

1) 确定错误的性质和位置的原则

用头脑去分析思考与错误征兆有关的信息。最有效的调试方法是用头脑分析与错误征兆有关的信息。一个能干的程序调试员应能作到不使用计算机就能够确定大部分错误。

避开死胡同。如果程序调试员走进了死胡同,或者陷入了绝境,最好暂时把问题抛开,留到第二天再去考虑,或者向其它人讲解这个问题。事实上常有这种情形:向一个好的听众简单地描述这个问题时,不需要任何听讲者的提示,你自己会突然发现问题的所在。

只把调试工具当作辅助手段来使用。利用调试工具,可以帮助思考,但不能代替思考。因为调试工具给你的是一种无规律的调试方法。实验证明,即使是对一个不熟悉的程序进行调试时,不用工具的人往往比使用工具的人更容易成功。

避免用试探法,最多只能把它当作最后手段。初学调试的人最常犯的一个错误是想试试修改程序来解决问题。这还是一种碰运气的盲目的动作,它的成功机会很小,而且还常把新的错误带到问题中来。

2) 修改错误的原则

在出现错误的地方,很可能还有别的错误。经验证明,错误有群集现象,当在某一程序段发现有错误时,在该程序段中还存在别的错误的概率也很高。因此,在修改一个错误时,还要查一下它的近邻,看是否还有别的错误。

修改错误的一个常见失误是只修改了这个错误的征兆或这个错误的表现,而没有修改错误的本身。如果提出的修改不能解释与这个错误有关的全部线索,那就表明了只修改了错误的一部分。

当心修正一个错误的同时有可能会引入新的错误。人们不仅需要注意不正确的修改,而且还要注意看起来是正确的修改可能会带来的副作用,即引进新的错误。因此在修改了错误之后,必须进行回归测试,以确认是否引进了新的错误。

修改错误的过程将迫使人们暂时回到程序设计阶段。修改错误也是程序设计的一种形式。一般说来,在程序设计阶段所使用的任何方法都可以应用到错误修正的过程中来。

第 8 章 软件维护

在软件开发完成交付用户使用后,就进入软件运行/维护阶段。此后的工作就是要保证软件在一个相当长的时期能够正常运行,这样对软件的维护就成为必不可少的了。

8.1 软件维护的概念

8.1.1 软件维护的定义

我们称在软件运行/维护阶段对软件产品所进行的修改就是所谓的维护。根据要求维护的原因,维护的活动可以分为三种类型。

1) 改正性维护 (corrective maintenance)

在软件交付使用后,由于开发时测试的不彻底、不完全,必然会有一部分隐藏的错误被带到运行阶段来。这些隐藏下来的错误在某些特定的使用环境下会暴露出来。为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的误使用,应进行的诊断和改正错误的过程,是改正性维护。例如,改正性维护可以是改正原来程序中开关使用的错误;解决开发时未能测试各种可能情况带来的问题;解决原来程序中遗漏处理文件中最后一个记录的问题等。

2) 适应性维护 (adaptive maintenance)

随着计算机的飞速发展,外部环境(新的硬、软件配置)或数据环境(数据库、数据格式、数据输入/输出方式、数据存储介质)可能发生变化,为了使软件适应这种变化,而修改软件的过程叫作适应性维护。例如,适应性维护可以是将某个应用程序从 DOS 环境移植到 Windows 环境;将原来在 VAX750 机上用 Oracle 的 SQL 实现的数据库移到 Compaq 机上;修改程序,使其适用于另外一种终端。

3) 完善性维护 (perfective maintenance)

在软件的使用过程中,用户往往会对软件提出新的功能与性能要求。为了满足这些要求,需要修改或再开发软件,以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性。这种情况下进行的维护活动叫作完善性维护。例如,完善性维护可能是修改一个计算工资的程序,使其增加新的扣除项目;缩短系统的应答时间,使其达到特定的要求;把现有程序的终端对话方式加以改造,使其具有方便用户使用的界面;改进图形输出;增加联机求助(HELP)功能;为软件的运行增加监控设施。

在维护阶段的最初一二年,改正性维护的工作量较大。随着错误发现率急剧降低,并趋于稳定,而进入了正常使用期。然而,由于改造的要求,适应性维护和完善性维护的工作量逐步增加,在这种维护过程中又会引入新的错误,从而加重了维护的工作量。

4) 预防性维护 (preventive maintenance)

除了以上三类维护之外,还有一类维护活动,叫作预防性维护。这是为了提高软件的

可维护性、可靠性等, 为以后进一步改进软件打下良好基础。通常, 预防性维护定义为:“ 把今天的方法学用于昨天的系统以满足明天的需要 ”。也就是说, 采用先进的软件工程方法对需要维护的软件或软件中的某一部分(重新) 进行设计、编制和测试。

在整个软件维护阶段花费的全部工作量中, 预防性维护只占很小的比例, 而完善性维护占了几近一半的工作量。参看图 8. 1。从图 8. 2 中可以看到, 软件维护活动花费的工作占整个生存期工作量的 70% 以上, 这是由于在漫长的软件运行过程中需要不断对软件进行修改, 以改正新发现的错误、适应新的环境和用户新的要求, 这些修改需要花费很多精力和时间, 而且有时修改不正确, 还会引入新的错误。同时, 软件维护技术不像开发技术那样成熟、规范化, 自然消耗工作量就比较多。

图 8. 1 三类维护占总维护比例

图 8. 2 维护在软件生存期所占比例

8. 1. 2 影响维护工作量的因素

在软件的维护过程中, 需要花费大量的工作量, 从而直接影响了软件维护的成本。因此, 应当考虑有哪些因素影响软件维护的工作量, 相应应该采取什么维护策略, 才能有效地维护软件并控制维护的成本。在软件维护中, 影响维护工作量的程序特性有以下 6 种。

- 1) 系统大小。
- 2) 程序设计语言。
- 3) 系统年龄。
- 4) 数据库技术的应用。
- 5) 先进的软件开发技术。
- 6) 其它: 如应用的类型、数学模型、任务的难度、开关与标记、IF 嵌套深度、索引或下标数等, 对维护工作量都有影响。

此外, 许多软件在开发时并未考虑将来的修改, 这为软件的维护带来许多问题。

8. 1. 3 软件维护的策略

根据影响软件维护工作量的各种因素, 针对三种典型的维护, James Martin 等提出了一些策略, 以控制维护成本。

1) 改正性维护

通常要生成 100% 可靠的软件并不一定合算, 成本太高。但使用新技术可大大提高可

靠性,并减少进行改正性维护的需要。这些技术包括:数据库管理系统、软件开发环境、程序自动生成系统、高级(第四代)语言。应用以上4种方法可产生更可靠的代码。此外,

- (1) 利用应用软件包,可开发出比由用户完全自己开发的系统可靠性更高的软件。
- (2) 使用结构化技术,开发的软件易于理解和测试。
- (3) 防错性程序设计。把自检能力引入程序,通过非正常状态的检查,提供审查跟踪。
- (4) 通过周期性维护审查,在形成维护问题之前就可确定质量缺陷。

2) 适应性维护

这一类的维护不可避免,但可以控制。

(1) 在配置管理时,把硬件、操作系统和其它相关环境因素的可能变化考虑在内,可以减少某些适应性维护的工作量。

(2) 把与硬件、操作系统,以及其它外围设备有关的程序归到特定的程序模块中。可把因环境变化而必须修改的程序局部于某些程序模块之中。

(3) 使用内部程序列表、外部文件,以及处理的例行程序包,可为维护时修改程序提供方便。

(4) 使用面向对象技术,增强软件系统的稳定性,易于修改和移植。

3) 完善性维护

利用前两类维护中列举的方法,也可以减少这一类维护。特别是数据库管理系统、程序生成器、应用软件包,可减少系统或程序员的维护工作量。

此外,建立软件系统的原型,把它在实际系统开发之前提供给用户。用户通过研究原型,进一步完善他们的功能要求,可以减少以后完善性维护的需要。

8.2 软件维护活动

为了有效地进行软件维护,应事先就开始作组织工作,建立维护的机构,申明提出维护申请报告的过程及评价的过程;为每一个维护申请规定标准的处理步骤;还必须建立维护活动的登记制度以及规定评价和评审的标准。

8.2.1 软件维护申请报告

所有软件维护申请应按规定的方式提出。软件维护组织通常提供维护申请报告 MRP (maintenance request form),或称软件问题报告,由申请维护的用户填写。如果遇到一个错误,用户必须完整地说明产生错误的情况,包括输入数据、错误清单以及其它有关材料。如果申请的是适应性维护或完善性维护,用户必须提出一份修改说明书,列出所有希望的修改。维护申请报告将由维护管理员和系统监督员来研究处理。

维护申请报告是由软件组织外部提交的文档,它是计划维护工作的基础。软件组织内部应相应地作出软件修改报告 SCR (software change report),并指明:

- 1) 所需修改变动的性质;
- 2) 申请修改的优先级;
- 3) 为满足某个维护申请报告,所需的工作量;

4) 预计修改后的状况。

软件修改报告应提交修改负责人,经批准后才能开始进一步安排维护工作。

8.2.2 软件维护工作流程

软件维护工作流程如图 8.3 所示。第一步是先确认维护要求。这需要维护人员与用户反复协商,弄清错误概况以及对业务的影响大小,以及用户希望作什么样的修改,并把这些情况存入故障数据库。然后由维护组织管理员确认维护类型。

图 8.3 软件维护的工作流程

对于改正性维护申请,从评价错误的严重性开始。如果存在严重的错误,则必须安排人员,在系统监督员的指导下,进行问题分析,寻找错误发生的原因,进行“救火”性的紧急维护;对于不严重的错误,可根据任务、机时情况,视轻重缓急进行排队,统一安排时间。

所谓“救火”式的紧急维护,是指如果发生的错误非常严重,不马上修理往往会导致重大事故,这样就必须紧急修改,暂不再顾及正常的维护控制,不必考虑评价可能发生的副作用。在维护完成、交付用户之后再去作补偿工作。

对于适应性维护和完善性维护申请,需要先确定每项申请的优先次序。若某项申请的优先级非常高,就可立即开始维护工作;否则,维护申请和其它的开发工作一样,进行排队,统一安排时间。并不是所有的完善性维护申请都必须承担,因为进行完善性维护等于是作二次开发,工作量很大,所以需要根据商业需要、可利用资源的情况、目前和将来软件的发展方向、以及其它的考虑,决定是否承担。

尽管维护申请的类型不同,但都要进行同样的技术工作。这些工作有:修改软件需求说明、修改软件设计、设计评审、对源程序作必要的修改、单元测试、集成测试(回归测试)、确认测试、软件配置评审等。

在每次软件维护任务完成后,最好进行一次情况评审,对以下问题作一总结:

- 1) 在目前情况下,设计、编码、测试中的哪一方面可以改进?
- 2) 哪些维护资源应该有,但没有?
- 3) 工作中主要的或次要的障碍是什么?
- 4) 从维护申请的类型来看是否应当有预防性维护?

情况评审对将来的维护工作如何进行会产生重要的影响,并可为软件机构的有效管理提供重要的反馈信息。

8.2.3 维护档案记录

为了估计软件维护的有效程度,确定软件产品的质量,同时确定维护的实际开销,需要在维护的过程中作好维护档案记录。其内容包括程序名称、源程序语句条数、机器代码指令条数、所用的程序设计语言、程序安装的日期、程序安装后的运行次数、与程序安装后运行次数有关的处理故障次数、程序改变的层次及名称、修改程序所增加的源程序语句条数、修改程序所减少的源程序语句条数、每次修改所付出的“人时”数、修改程序的日期、软件维护人员的姓名、维护申请报告的名称、维护类型、维护开始时间和维护结束时间、花费在维护上的累计“人时”数、维护工作的净收益等。对每项维护任务都应该收集上述数据。

8.2.4 维护评价

评价维护活动比较困难,因为缺乏可靠的数据。但如果维护的档案记录作得比较好,可以得出一些维护“性能”方面的度量值。可参考的度量值如:

- 每次程序运行时的平均出错次数;
- 花费在每类维护上的总“人时”数;
- 每个程序、每种语言、每种维护类型的程序平均修改次数;
- 因为维护,增加或删除每个源程序语句所花费的平均“人时”数;
- 用于每种语言的平均“人时”数;
- 维护申请报告的平均处理时间;
- 各类维护申请的百分比。

这七种度量值提供了定量的数据,据此可对开发技术、语言选择、维护工作计划、资源分配、以及其它许多方面作出判定。因此,这些数据可以用来评价维护工作。

8.3 程序修改的步骤及修改的副作用

在软件维护时,必然会对源程序进行修改。通常对源程序的修改不能无计划地仓促上阵,为了正确、有效地修改,需要经历以下三个步骤。

8.3.1 分析和理解程序

经过分析,全面、准确、迅速地理解程序是决定维护成败和质量好坏的关键。在这方面,软件的可理解性和文档的质量非常重要。必须:

- 1) 研究程序的使用环境及有关资料,尽可能得到更多的背景信息;
- 2) 理解程序的功能和目标;
- 3) 掌握程序的结构信息,即从程序中细分出若干结构成份。如程序系统结构、控制结构、数据结构和输入/输出结构等;
- 4) 了解数据流信息,即所涉及到的数据来源何处,在哪里被使用;
- 5) 了解控制流信息,即执行每条路径的结果;
- 6) 如果设计存在,则可利用它们来帮助画出结构图和高层流程图;
- 7) 理解程序的操作(使用)要求。

为了容易地理解程序,要求自顶向下地理解现有源程序的程序结构和数据结构,为此可采用如下几种方法:

1) 分析程序结构图。

(1) 搜集所有存储该程序的文件,阅读这些文件,记下它们包含的过程名,建立一个包括这些过程名和文件名的文件;

(2) 分析各个过程的源代码;

(3) 分析各个过程的接口,估计更改的复杂性。

2) 数据跟踪。

(1) 建立各层次的程序级上的接口图,展示各模块或过程的调用方式和接口参数;

(2) 利用数据流分析方法,对过程内部的一些变量进行跟踪;维护人员通过这种数据流跟踪,可获得有关数据在过程间如何传递,在过程内如何处理等信息。对于判断问题原因特别有用。在跟踪的过程中可在源程序中间插入自己的注释。

3) 控制跟踪。

控制流跟踪同样可在结构图基础上或源程序基础上进行。可采用符号执行或实际动态跟踪的方法,了解数据如何从一个输入源到达输出点的。

4) 在分析的过程中,充分阅读和使用源程序清单和文档,分析现有文档的合理性。

5) 充分使用由编译程序或汇编程序提供的交叉引用表、符号表、以及其它有用的信息。

6) 如有可能,积极参加开发工作。

8.3.2 修改程序

对程序的修改,必须事先作出计划,有预谋地、周密有效地实施修改。

1) 设计程序的修改计划

程序的修改计划要考虑人员和资源的安排。小的修改可以不需要详细的计划,而对于需要耗时数月的修改,就需要计划立案。此外,在编写有关问题和解决方案的大纲时,必须充分地描述修改作业的规格说明。修改计划的内容主要包括:

- 规格说明信息: 数据修改、处理修改、作业控制语言修改、系统之间接口的修改等;
- 维护资源: 新程序版本、测试数据、所需的软件系统、计算机时间等;
- 人员: 程序员、用户相关人员、技术支持人员、厂家联系人、数据录入员等;
- 提供: 纸面、计算机媒体等。

针对以上每一项, 要说明必要性、从何处着手、是否接受、日期等。通常, 可采用自顶向下的方法, 在理解程序的基础上作如下工作。

(1) 研究程序的各个模块、模块的接口、及数据库, 从全局的观点提出修改计划。

(2) 依次地把要修改的、以及那些受修改影响的模块和数据结构分离出来。为此, 要作如下工作。

- 识别受修改影响的数据;
- 识别使用这些数据的程序模块;
- 对于上面程序模块, 按是产生数据、修改数据, 还是删除数据进行分类;
- 识别对这些数据元素的外部控制信息;
- 识别编辑和检查这些数据元素的地方;
- 隔离要修改的部分。

(3) 详细地分析要修改的、以及那些受变更影响的模块和数据结构的内部细节, 设计修改计划, 标明新逻辑及要改动的现有逻辑。

(4) 向用户提供回避措施。用户的某些业务因软件中发生问题而中断, 为不让系统长时间停止运行, 需把问题局部化, 在可能的范围内继续开展业务。可以采取的措施有两种。

在问题的原因还未找到时, 先就问题的现象提供回避的操作方法, 可能情况有以下几种。

- 意外停机, 系统完全不能工作——作为临时的处置, 消除特定的数据, 插入临时代码(打补丁), 以人工方式运行系统。
- 安装的期限到期——系统有时要延迟变更。例如, 税率改变时, 继续执行其它处理, 同时修补有关的部分再执行它, 或者制作特殊的程序, 然后再根据执行结果作修正。
- 发现错误运行系统——人工查找错误并修正之。

必须正确地了解以现在状态运行系统将给应用系统的业务造成什么样的影响, 研究使用现行系统将如何及多大程度地促进应用的业务。

如果弄清了问题的原因, 可通过临时修改或改变运行控制以回避在系统运行时产生的问题。

2) 修改代码, 以适应变化

在修改时, 要求:

(1) 正确、有效地编写修改代码;

(2) 要谨慎地修改程序, 尽量保持程序的风格及格式, 要在程序清单上注明改动的指令;

(3) 不要匆忙删除程序语句, 除非完全肯定它是无用的;

(4) 不要试图共用程序中已有的临时变量或工作区, 为了避免冲突或混淆用途, 应自

行设置自己的变量;

(5) 插入错误检测语句;

(6) 保持详细的维护活动和维护结果记录;

(7) 如果程序结构混乱, 修改受到干扰, 可抛弃程序重新编写;

3) 修改程序的副作用

所谓副作用是指因修改软件而造成的错误或其它不希望发生的情况, 有以下三种副作用。

(1) 修改代码的副作用

在使用程序设计语言修改源代码时, 都可能引入错误。例如, 删除或修改一个子程序、删除或修改一个标号、删除或修改一个标识符、改变程序代码的时序关系、改变占用存储的大小、改变逻辑运算符、修改文件的打开或关闭、改进程序的执行效率, 以及把设计上的改变翻译成代码的改变、为边界条件的逻辑测试作出改变时, 都容易引入错误。

(2) 修改数据的副作用

在修改数据结构时, 有可能造成软件设计与数据结构不匹配, 因而导致软件出错。数据的副作用是修改软件信息结构导致的结果。例如, 在重新定义局部的或全局的常量、重新定义记录或文件的格式、增大或减小一个数组或高层数据结构的大小、修改全局或公共数据、重新初始化控制标志或指针、重新排列输入/输出或子程序的参数时, 容易导致设计与数据不相容的错误。数据副作用可以通过详细的设计文档加以控制。在此文档中描述了一种交叉引用, 把数据元素、记录、文件和其它结构联系起来。

(3) 文档的副作用

对数据流、软件结构、模块逻辑或任何其它有关特性进行修改时, 必须对相关技术文档进行相应修改。否则会导致文档与程序功能不匹配、缺省条件改变、新错误信息不正确等错误。使得软件文档不能反映软件的当前状态。对于用户来说, 软件事实上就是文档。如果对可执行软件的修改不反映在文档里, 会产生文档的副作用。例如, 对交互输入的顺序或格式进行修改, 如果没有正确地记入文档中, 可能引起重大的问题。过时的文档内容、索引和文本可能造成冲突, 引起用户的失败和不满。因此, 必须在软件交付之前对整个软件配置进行评审, 以减少文档的副作用。事实上, 有些维护请求并不要求改变软件设计和源代码, 而是指出在用户文档中不够明确的地方。在这种情况下, 维护工作主要集中在文档上。

为了控制因修改而引起的副作用, 要作到:

(1) 按模块把修改分组;

(2) 自顶向下地安排被修改模块的顺序;

(3) 每次修改一个模块;

(4) 对于每个修改了的模块, 在安排修改下一个模块之前, 要确定这个修改的副作用。可以使用交叉引用表、存储映象表、执行流程跟踪等。

8.3.3 重新验证程序

在将修改后的程序提交用户之前, 需要用以下的方法进行充分的确认和测试, 以保证

整个修改后的程序的正确性。

1) 静态确认

修改软件,伴随着引起新的错误的危险。为了能够作出正确的判断,验证修改后的程序至少需要两个人参加。要检查:

- (1) 修改是否涉及到规格说明? 修改结果是否符合规格说明? 有没有歪曲规格说明?
 - (2) 程序的修改是否足以修正软件中的问题? 源程序代码有无逻辑错误? 修改时有无修补失误?
 - (3) 修改部分对其它部分有无不良影响(副作用)?
- 对软件进行修改,常常会引发别的问题,因此有必要检查修改的影响范围。

2) 计算机确认

在充分进行了以上确认的基础上,要用计算机对修改程序进行确认测试。

- (1) 确认测试顺序: 先对修改部分进行测试,然后隔离修改部分,测试程序的未修改部分,最后再把它集成起来进行测试。这种测试称为回归测试。
 - (2) 准备标准的测试用例。
 - (3) 充分利用软件工具帮助重新验证过程。
 - (4) 在重新确认过程中,需邀请用户参加。
- 3) 维护后的验收——在交付新软件之前,维护主管部门要检验。

- (1) 全部文档是否完备,并已更新;
- (2) 所有测试用例和测试结果已经正确记载;
- (3) 记录软件配置所有副本的工作已经完成;
- (4) 维护工序和责任已经确定。

从维护角度来看所需测试种类如:

- | | |
|----------------|-----------------|
| · 对修改事务的测试; | · 对修改程序的测试; |
| · 操作过程的测试; | · 应用系统运行过程的测试; |
| · 使用过程的测试; | · 系统各部分之间接口的测试; |
| · 作业控制语言的测试; | · 与系统软件接口的测试; |
| · 软件系统之间接口的测试; | · 安全性测试; |
| · 后备/恢复过程的测试。 | |

8.4 软件可维护性

许多软件的维护十分困难,原因在于这些软件的文档和源程序难于理解,又难于修改。从原则上讲,软件开发工作应严格按照软件工程的要求,遵循特定的软件标准或规范进行。但实际上往往由于种种原因并不能真正作到。例如,文档不全、质量差、开发过程不注意采用结构化方法,忽视程序设计风格等等。因此,造成软件维护工作量加大,成本上升,修改出错率升高。此外,许多维护要求并不是因为程序中出错而提出的,而是为适应环境变化或需求变化而提出的。由于维护工作面广,维护难度大,稍有不慎,就会在修改中给软件带来新的问题或引入新的差错。所以,为了使得软件能够易于维护,必须考虑使软

件具有可维护性。

8.4.1 软件可维护性的定义

所谓软件可维护性,是指纠正软件系统出现的错误和缺陷,以及为满足新的要求进行修改、扩充或压缩的容易程度。可维护性、可使用性、可靠性是衡量软件质量的几个主要质量特性,也是用户十分关心的几个方面。可惜的是影响软件质量的这些重要因素,目前尚没有对它们定量度量的普遍适用的方法。但是就它们的概念和内涵来说则是很明确的。

软件的可维护性是软件开发阶段各个时期的关键目标。

目前广泛使用的是用如下的七个特性衡量程序的可维护性。而且对于不同类型的维护,这七种特性的侧重点也不相同。表 8.1 显示了在各类维护中应侧重哪些特性。图中的“ ”表示需要的特性。

表 8.1 在各类维护中的侧重点

	改正性维护	适应性维护	完善性维护
可理解性			
可测试性			
可修改性			
可靠性			
可移植性			
可使用性			
效率			

上面列举的这些质量特性通常体现在软件产品的许多方面,为使每一个质量特性都达到预定的要求,需要在软件开发的各个阶段采取相应的措施加以保证。即是说,这些质量要求要渗透到各开发阶段的各个步骤当中。因此,软件的可维护性是产品投入运行以前各阶段面向上述各质量特性要求进行开发的最终结果。

8.4.2 可维护性的度量

人们一直期望对软件的可维护性作出定量度量,但要作到这一点并不容易。许多研究工作集中在这个方面,形成了一个引人注目的学科——软件度量学。下面介绍度量一个可维护的程序的七种特性时常用的方法。这就是质量检查表、质量测试、质量标准。

质量检查表是用于测试程序中某些质量特性是否存在的一个问题清单。评价者针对检查表上的每一个问题,依据自己的定性判断,回答“ Yes ”或者“ No ”。质量测试与质量标准则用于定量分析和评价程序的质量。由于许多质量特性是相互抵触的,要考虑几种不同的度量标准,相应地去度量不同的质量特性。

1) 可理解性

可理解性表明人们通过阅读源代码和相关文档,了解程序功能及其如何运行的容易程度。一个可理解的程序主要应具备以下一些特性:模块化(模块结构良好、功能完整、简明),风格一致性(代码风格及设计风格的一致性),不使用令人捉摸不定或含糊不清的代

码,使用有意义的数据名和过程名,结构化,完整性(对输入数据进行完整性检查)等。

对于可理解性,可以使用一种叫作“90- 10 测试”的方法来衡量。即把一份被测试的源程序清单拿给一位有经验的程序员阅读 10 分钟,然后把这个源程序清单拿开,让这位程序员凭自己的理解和记忆,写出该程序的 90%。如果程序员真的写出来了,则认为这个程序具有可理解性,否则这个程序要重新编写。

2) 可靠性

可靠性表明一个程序按照用户的要求和设计目标,在给定的一段时间内正确执行的概率。关于可靠性,度量的标准主要有:平均失效间隔时间 MTTF(mean time to failure)、平均修复时间 MTTR(mean time to repair error)、有效性 A (= MTBD/(MTBD + MDT))。度量可靠性的方法,主要有两类。

(1) 根据程序错误统计数字,进行可靠性预测。常用的方法是利用一些可靠性模型,根据程序测试时发现并排除的错误数,预测平均失效间隔时间 MTTF。

(2) 根据程序复杂性,预测软件可靠性。用程序复杂性预测可靠性,前提条件是可靠性与复杂性有关。因此可用复杂性预测出错率。程序复杂性度量标准可用于预测哪些模块最可能发生错误,以及可能出现的错误类型。了解了错误类型及它们在哪里可能出现,就能更快地查出和纠正更多的错误,以提高可靠性。

3) 可测试性

可测试性表明论证程序正确性的容易程度。程序越简单,证明其正确性就越容易。而且设计合用的测试用例,取决于对程序的全面理解。因此,一个可测试的程序应当是可理解的、可靠的、简单的。

对于程序模块,可用程序复杂性来度量可测试性。程序的环路复杂性越大,程序的路径就越多。因此,全面测试程序的难度就越大。

4) 可修改性

可修改性表明程序容易修改的程度。一个可修改的程序应当是可理解的、通用的、灵活的、简单的。其中,通用性是指程序适用于各种功能变化而无需修改。灵活性是指能够容易地对程序进行修改。

测试可修改性的一种定量方法是修改练习。其基本思想是通过作几个简单的修改,来评价修改的难度。设 C 是程序中各个模块的平均复杂性,n 是必须修改的模块数,A 是要修改的模块的平均复杂性。则修改的难度 D 由下式计算:

$$D = A / C$$

对于简单的修改,若 $D > 1$,说明该程序修改困难。A 和 C 可用任何一种度量程序复杂性的方法计算。

5) 可移植性

可移植性表明程序转移到一个新的计算环境的可能性的。或者它表明程序可以容易地、有效地在各种各样的计算环境中运行的容易程度。

一个可移植的程序应具有结构良好、灵活、不依赖于某一具体计算机或操作系统的性能。

6) 效率

效率表明一个程序能执行预定功能而又不浪费机器资源的程度。这些机器资源包括内存容量、外存容量、通道容量和执行时间。

7) 可使用性

从用户观点出发,把可使用性定义为程序方便、实用及易于使用的程度。一个可使用的程序应是易于使用的、能允许用户出错和改变,并尽可能不使用户陷入混乱状态的程序。

用于可使用性度量的检查项目主要有:

- (1) 程序是否具有自描述性?
- (2) 程序是否能始终如一地按照用户的要求运行?
- (3) 程序是否让用户对数据处理有一个满意的和适当的控制?
- (4) 程序是否容易学会使用?

(5) 程序是否使用数据管理系统来自动地处理事务性工作和管理格式化、地址分配及存储器组织。

- (6) 程序是否具有容错性?
- (7) 程序是否灵活?

8) 其它间接定量度量可维护性的方法

Gilb 提出了与软件维护期间工作量有关的一些数据,可以使用它们间接地对软件的可维护性作出估计。

- (1) 问题识别的时间;
- (2) 因管理活动拖延的时间;
- (3) 收集维护工具的时间;
- (4) 分析、诊断问题的时间;
- (5) 修改规格说明的时间;
- (6) 具体的改错或修改的时间;
- (7) 局部测试的时间;
- (8) 集成或回归测试的时间;
- (9) 维护的评审时间;
- (10) 恢复时间。

这些数据反映了维护全过程中检错- 纠错- 验证的周期,即从检测出软件存在的问题开始至修正它们并经回归测试验证这段时间。可以粗略地认为,这个周期越短,维护越容易。

8.5 提高可维护性的方法

软件的可维护性对于延长软件的生存期具有决定的意义,因此必须考虑如何才能提高软件的可维护性。为了作到这一点,需从以下五个方面着手。

8.5.1 建立明确的软件质量目标和优先级

一个可维护的程序应是可理解的、可靠的、可测试的、可修改的、可移植的、效率高的、可使用的。但要实现所有这些目标,需要付出很大的代价,而且也不一定行得通。因为某些质量特性是相互促进的,例如可理解性和可测试性、可理解性和可修改性。但另一些质量特性却是相互抵触的,例如效率和可移植性、效率和可修改性等。因此,尽管可维护性要求每一种质量特性都要得到满足,但它们的相对重要性应随程序的用途及计算环境的不同而不同。例如,对编译程序来说,可能强调效率;但对管理信息系统来说,则可能强调可使用性和可修改性。所以,应当对程序的质量特性,在提出目标的同时还必须规定它们的优先级。这样有助于提高软件的质量,并对软件生存期的费用产生很大的影响。

8.5.2 使用提高软件质量的技术和工具

1) 模块化

模块化技术的优点是如果需要改变某个模块的功能,则只要改变这个模块,对其它模块影响很小;如果需要增加程序的某些功能,则仅需增加完成这些功能的新的模块或模块层;程序的测试与重复测试比较容易;程序错误易于定位和纠正;容易提高程序效率。

2) 结构化程序设计

结构化程序设计不仅使得模块结构标准化,而且将模块间的相互作用也标准化了。因而把模块化又向前推进了一步。采用结构化程序设计可以获得良好的程序结构。

3) 使用结构化程序设计技术,提高现有系统的可维护性

(1) 采用备用件的方法——当要修改某一个模块时,用一个新的结构良好的模块替换掉整个模块。这种方法要求了解所替换模块的外部(接口)特性,可以不了解其内部工作情况。它有利于减少新的错误。

(2) 采用自动重建结构和重新格式化的工具(结构更新技术)——这种方法采用如代码评价程序、重定格式程序、结构化工具等自动工具,把非结构化代码转换成良好结构代码。

(3) 改进现有程序的不完善的文档——改进和补充文档的目的是为了提高程序的可理解性,以提高可维护性。

(4) 使用结构化程序设计方法实现新的子系统。

(5) 采用结构化小组——在软件开发过程中,建立主程序员小组,实现严格的组织化结构,强调规范,明确领导以及职能分工,能够改善通信,提高程序生产率;在检查程序质量时,采取有组织分工的结构普查,分工合作,各司其职,能够有效地实施质量检查。同样,在软件维护过程中,维护小组也可以采取与主程序员小组和结构普查类似的方式,以保证程序的质量。

8.5.3 进行明确的质量保证审查

质量保证审查对于获得和维持软件的质量,是一个很有用的技术。除了保证软件得到适当的质量外,审查还可以用来检测在开发和维护阶段内发生的质量变化。一旦检测出问

题来,就可以采取措施纠正,以控制不断增长的软件维护成本。

为了保证软件的可维护性,有四种类型的软件审查。

1) 在检查点进行复审

保证软件质量的最佳方法是在软件开发的最初阶段就把质量要求考虑进去,并在开发过程每一阶段的终点,设置检查点进行检查。检查的目的是要证实,已开发的软件是否符合标准,是否满足规定的质量需求。

在不同的检查点,检查的重点不完全相同,如图 8.4 所示。

图 8.4 软件开发期间各个检查点的检查重点

例如在设计阶段,检查重点是可理解性、可修改性、可测试性。可理解性检查的重点是程序的复杂性。对每个模块可用 McCabe 环路计算模块的复杂性,若大于 10,则需重新设计。

可以使用各种质量特性检查表,或用度量标准来检查可维护性。各种度量标准应当在管理部门、用户、软件开发人员、软件维护人员当中达成一致意见。审查小组可以采用人工测试一类的方式进行审查。

2) 验收检查

验收检查是一个特殊的检查点的检查,是交付使用前的最后一次检查,是软件投入运行之前保证可维护性的最后机会。它实际上是验收测试的一部分,只不过它是从维护的角度提出验收的条件和标准。

下面是验收检查必须遵循的最小验收标准。

(1) 需求和规范标准

需求应当以可测试的术语进行书写,排列优先次序和定义;

区分必须的、任选的、将来的需求;

包括对系统运行时的计算机设备的需求;对维护、测试、操作、以及维护人员的需求;对测试工具等的需求。

(2) 设计标准

程序应设计成份层的模块结构。每个模块应完成唯一的功能,并达到高内聚、低耦合;

通过一些知道预期变化的实例,说明设计的可扩充性、可缩减性和可适应性。

(3) 源代码标准

尽可能使用最高级的程序设计语言,且只使用语言的标准版本;

所有的代码都必须具有良好的结构;

所有的代码都必须文档化,在注释中说明它的输入、输出、以及便于测试/再测试的一些特点与风格。

(4) 文档标准:文档中应说明程序的输入/输出、使用的方法/算法、错误恢复方法、所有参数的范围以及缺省条件等。

3) 周期性地维护审查

检查点复查和验收检查,可用来保证新软件系统的可维护性。对已有的软件系统,则应当进行周期性的维护检查。

软件在运行期间,为了纠正新发现的错误或缺陷,为了适应计算环境的变化,为了响应用户新的需求,必须进行修改。因此会导致软件质量有变坏的危险,可能产生新的错误,破坏程序概念的完整性。因此,必须像硬件的定期检查一样,每月一次,或二月一次,对软件作周期性的维护审查,以跟踪软件质量的变化。

周期性维护审查实际上是开发阶段检查点复查的继续,并且采用的检查方法、检查内容都是相同的。维护审查的结果可以同以前的维护审查的结果、以及以前的验收检查的结果和检查点检查的结果相比较,任何一种改变都表明在软件质量上或其它类型的问题上可能起了变化。

4) 对软件包进行检查

软件包是一种标准化了的、可为不同单位、不同用户使用的软件。软件包卖主考虑到他的专利权,一般不会提供给用户他的源代码和程序文档。因此,对软件包的维护采取以下方法。

使用单位的维护人员首先要仔细分析、研究卖主提供的用户手册、操作手册、培训教程、新版本说明、计算机环境要求书、未来特性表、以及卖方提供的验收测试报告等,在此基础上,深入了解本单位的希望和要求,编制软件包的检验程序。

该检验程序检查软件包程序所执行的功能是否与用户的要求和条件相一致。为了建立这个程序,维护人员可以利用卖方提供的验收测试实例,还可以自己重新设计新的测试实例。根据测试结果,检查和验证软件包的参数或控制结构,以完成软件包的维护。

8.5.4 选择可维护的程序设计语言

程序设计语言的选择,对程序的可维护性影响很大。低级语言,即机器语言和汇编语言,很难理解,很难掌握,因此很难维护。高级语言比低级语言容易理解,具有更好的可维护性。但同是高级语言,可理解的难易程度也不一样。例如,COBOL语言比FORTRAN语言容易理解,因为它更接近于英语;PL/1语言比COBOL语言容易理解,因为它有更丰富、更强的指令集。

图 8.5 程序设计语言对可维护性的影响

第四代语言,例如查询语言、图形语言、报表生成器、非常高级的语言等,有的是过程性的语言,有的是非过程性的语言。不论是哪种语言,编制出的程序都容易理解和修改,而且,其产生的指令条数可能要比用 COBOL 语言或用 PL/1 语言编制出的少一个数量级,开发速度快许多倍。有些非过程性的第四代语言,用户不需要指出实现的算法,仅需向编译程序或解释程序提出自己的要求,由编译程序或解释程序自己作出实现用户要求的智能假设,例如自动选择报表格式,选择字符类型和图形显示方式等。从维护角度来看,第四代语言比其它语言更容易维护。

8.5.5 改进程序的文档

程序文档是对程序总目标、程序各组成部分之间的关系、程序设计策略、程序实现过程的历史数据等的说明和补充。程序文档对提高程序的可理解性有着重要作用。即使是一个十分简单的程序,要想有效地、高效率地维护它,也需要编制文档来解释其目的及任务。而对于程序维护人员来说,要想对程序编制人员的意图重新改造,并对今后变化的可能性进行估计,缺了文档也是不行的。因此,为了维护程序,人们必须阅读和理解文档。那种对文档的价值估计过低的眼光,是由于过低估计了用户对改变的要求而造成的。

在软件维护阶段,利用历史文档,可以大大简化维护工作。例如,通过了解原设计思想,可以指导维护人员选择适当的方法去修改代码而不危及系统的完整性。又例如,了解系统开发人员认为系统中最困难的部分,可以向维护人员提供最直接的线索,判断出错之处。历史文档有三种:

系统开发日志:它记录了项目的开发原则、开发目标、优先次序、选择某种设计方案的理由、决策策略、使用的测试技术和工具、每天出现的问题、计划的成功和失败之处等。系统开发日志在日后对维护人员想要了解系统的开发过程和开发中遇到什么问题是非常必要的。

错误记载:它把出错的历史情况记录下来,对于预测今后可能发生的错误类型及出错频率有很大帮助。也有助于维护人员查明出现故障的程序或模块,以便去修改或替换它们。此外,对错误进行统计、跟踪,可以更合理地评价软件质量以及软件质量度量标准和软件方法的有效性。

系统维护日志:系统维护日志记录了在维护阶段有关系统修改和修改目的的信息。包括修改的宗旨。修改的策略、存在的问题、问题所在的位置、解决问题的办法、修改要求和说明、注意事项、新版本说明等信息。它有助于人们了解程序修改背后的思维过程,以进一步了解修改的内容和修改带来的影响。

8.6 逆向工程和再工程

术语“逆向工程(reverse engineering)”来自硬件。软件公司对竞争对手的硬件产品进行分解,了解竞争对手在设计和制造上的“隐秘”。如果竞争对手的设计与制造的规格说明能够得到,要掌握这些隐秘并不难。然而,这些文档是保密的,软件公司作逆向工程时是不能利用的。成功的逆向工程应当通过考察产品的实际样品,导出该产品的一个或多个设计与制造的规格说明。

软件的逆向工程是完全类似的。但是,要作逆向工程的程序常常不是竞争对手的,因为要受到法律约束。公司作逆向工程的程序,一般是自己的程序,有些是在多年以前开发出来的。这些程序没有规格说明,对它们的了解很模糊。因此,软件的逆向工程是分析程序,力图在比源代码更高抽象层次上建立程序表示的过程。逆向工程是设计恢复的过程。逆向工程工具可以从已存在程序中抽取数据结构、体系结构和程序设计信息。

再工程(re-engineering),也叫复壮(修理)或再生。它不仅能从已存在的程序中重新获得设计信息,而且还能使用这些信息改建或重构现有的系统,以改进它的综合质量。一般软件人员利用再工程重新实现已存在的程序,同时加进新的功能或改善它的性能。

每一个大的软件开发机构(或许多小的软件开发单位)的程序有着上百万行的老代码,它们都是逆向工程或再工程的可能对象。但是由于某些程序并不频繁使用而且不需要改变,而且逆向工程和再工程的工具还处于摇篮时代,仅能对有限种类的应用执行逆向工程或再工程,代价又十分昂贵,因此对其库中的每一个程序都作逆向工程或再工程是不现实的。

为了执行预防性维护,软件开发组织必须选择在最近的将来可能变更的程序,作好变更它们的准备。逆向工程和再工程可用于执行这种维护任务。

逆向工程就好像是一个魔术管道。把一个非结构化的无文档的源代码或目标代码清单喂入管道,则从管道的另一端出来的是计算机软件的全部文档。逆向工程可以从源代码或目标代码中提取设计信息,其中抽象的层次、文档的完全性、工具与人的交互程度、以及过程的方法都是重要的因素。

逆向工程的抽象层次和用来产生它的工具提交的设计信息是原来设计的赝品,它是从源代码或目标代码中提取出来的。理想情况是抽象层次尽可能地高,也就是说,逆向工程过程应当能够导出过程性设计的表示(最低层抽象)、程序和数据结构信息(低层抽象)、数据和控制流模型(中层抽象)和实体联系模型(高层抽象)。随着抽象层次的增加,可以给软件工程师提供更多的信息,使得理解程序更容易。

如果逆向工程过程的方向只有一条路,则从源代码或目标代码中提取的所有信息都将提供给软件工程师。他们可以用来进行维护活动。如果方向有两条路,则信息将反馈给再工程工具,以便重新构造或重新生成老的程序。

第 9 章 软件工程标准化与软件文档

9.1 软件工程标准化

人要和计算机打交道, 需要程序设计语言, 这种语言不仅应让计算机理解, 而且还应让别人看懂, 使其成为人际交往的工具。程序设计语言的标准化最早提到日程上来。20 世纪 60 年代程序设计语言蓬勃发展, 出现了各种名目繁多的语言, 这对于推动计算机语言的发展有着重要作用, 但同时也带来许多麻烦。即使同一种语言, 由于在不同型号的计算机上实现时, 作了不同程度的修改和变动, 形成了这一语言的种种“方言”, 为程序的交流设置了障碍。制定标准化程序设计语言, 为某一程序设计语言规定若干个标准子集, 对于语言的实现者和用户都带来了很大方便。

9.1.1 什么是软件工程标准

随着软件工程学的发展, 人们对计算机软件的认识逐渐深入。软件工作的范围从只是使用程序设计语言编写程序, 扩展到整个软件生存期。诸如软件概念的形成、需求分析、设计、实现、测试、安装和检验、运行和维护, 直到软件淘汰。同时还有许多技术管理工作(如过程管理、产品管理、资源管理)以及确认与验证工作(如评审和审计、产品分析、测试等)常常是跨越软件生存期各个阶段的专门工作。所有这些方面都应当逐步建立起标准或规范来。

另一方面, 软件工程标准的类型也是多方面的。它可能包括过程标准(如方法、技术、度量等)、产品标准(如需求、设计、部件、描述、计划、报告等)、专业标准(如职别、道德准则、认证、特许、课程等), 以及记法标准(如术语、表示法、语言等)。下面根据中国国家标准 GB/T 15538—1995《软件工程标准分类法》给出软件工程标准的分类。

软件工程的标准可用一张二维的表格来表示。表 9.1(a) 和表 9.1(b) 给出了这个二维表的大致格式。(b) 表是(a) 表的继续。表中作为举例填入了三个标准(请注意它们在表中出现的位置)。

- 1) FIPS 135 是美国国家标准局发布的《软件文档管理指南》。
- 2) NSAC—39 是美国核子安全分析中心发布的《安全参数显示系统的验证与确认》。
- 3) ISO5807 是国际标准化组织公布(现已成为中国的国家标准) 的《信息处理——数据流程图、程序流程图、系统流程图、程序网络图和系统资源图的文件编制符号及约定》。

这个图表不仅表明了软件工程标准的范围 and 如何对标准分类, 而且对标准的开发具有指导作用。已经制定的标准都可在表中找到相应的位置, 而且它可启发人们制定新的标准。

表 9.1(a) 软件工程标准分类

			软 件 生 存 期							
			概念	需求	设计	实现	测试	安装与检验	运行与维护	引退
标准类型	过程	方法								
		技术								
		度量								
	产品	需求								
		设计								
		部件								
		描述								
		计划								
		报告								
	专业	职别								
		道德准则								
		认证								
		特许								
		课程								
	记号	术语								
		表示法			ISO5807					
		语言								

表 9.1(b) 软件工程标准分类

			技 术 管 理			确 认 与 验 证		
			过程管理	产品管理	资源管理	评审与审计	产品分析	测试
标准类型	过程	方法				NSAC- 39	NSAC- 39	NSAC- 39
		技术	FIPS1, 05					
		度量						
	产品	需求						
		设计						
		部件						
		描述						
		计划						
		报告						
	专业	职别						
		道德准则						
		认证						
		特许						
		课程						
	记号	术语						
		表示法						
		语言						

9.1.2 软件工程标准化的意义

为什么要积极推行软件工程标准化,其道理是显而易见的。仅就一个软件开发项目来说,有许多层次、不同分工的人员相互配合,在开发项目的各个部分以及各开发阶段之间也都存在着许多联系和衔接问题。如何把这些错综复杂的关系协调好,需要有一系列统一的约束和规定。在软件开发项目取得阶段成果或最后完成时,需要进行阶段评审和验收测试。投入运行的软件,其维护工作中遇到的问题又与开发工作有着密切的关系。软件的管理工作则渗透到软件生存期的每一个环节。所有这些都要求提供统一的行动规范和衡量准则,使得各种工作都能有章可循。

软件工程的标准化会给软件工作带来许多好处,比如,

- 1) 可提高软件的可靠性、可维护性和可移植性(这表明软件工程标准化可提高软件产品的质量);
- 2) 提高软件的生产率;
- 3) 提高软件人员的技术水平;
- 4) 提高软件人员之间的通信效率,减少差错和误解;
- 5) 有利于软件管理;有利于降低软件产品的成本和运行维护成本;
- 6) 有利于缩短软件开发周期。

9.1.3 软件工程标准的层次

根据软件工程标准制定的机构和标准适用的范围有所不同,它可分为五个级别,即国际标准、国家标准、行业标准、企业(机构)标准及项目(课题)标准。以下分别对五级标准的标识符和标准制定(或批准)的机构作一简要说明。

1) 国际标准

由国际联合机构制定和公布,提供各国参考的标准。

(1) ISO(international standards organization)——国际标准化组织。这一国际机构有着广泛的代表性和权威性,它所公布的标准也有较大的影响。20世纪60年代初,该机构建立了“计算机与信息处理技术委员会”,简称ISO/TC97,专门负责与计算机有关的标准化工作。这一标准通常冠有ISO字样,如ISO 8631—86 information processing _ program constructs and conventions for their representation《信息处理——程序构造及其表示法的约定》。该标准现已由中国收入国家标准。

2) 国家标准

由政府或国家级的机构制定或批准,适用于全国范围的标准,如:

(1) GB——中华人民共和国国家技术监督局是中国的最高标准化机构,它所公布实施的标准简称为“国标”。现已批准了若干个软件工程标准(详见本章9.4节)。

(2) ANSI (american national standards institute)——美国国家标准协会。这是美国一些民间标准化组织的领导机构,具有一定的权威性。

(3) FIPS (NBS) { federal information processing standards (national bureau of standards) }——美国商务部国家标准局联邦信息处理标准。它所公布的标准均冠有FIPS

字样。如, 1987 年发表的 FIPS PUB 132-87 guideline for validation and verification plan of computer software(软件确认与验证计划指南)。

(4) BS (british standard)——英国国家标准。

(5) DIN (deutsches institut für normung)——德国标准协会。

(6) JIS (japanese industrial standard)——日本工业标准。

3) 行业标准

由行业机构、学术团体或国防机构制定, 并适用于某个业务领域的标准, 如:

(1) IEEE (institute of electrical and electronics engineers)——美国电气与电子工程师学会。近年该学会专门成立了软件标准分技术委员会(SESS), 积极开展了软件标准化活动, 取得了显著成果, 受到了软件界的关注。IEEE 通过的标准经常要报请 ANSI 审批, 使之具有国家标准的性质。因此, 日常看到 IEEE 公布的标准常冠有 ANSI 的字头。例如, ANSI/IEEE Str 828—1983《软件配置管理计划标准》。

(2) GJB——中华人民共和国国家军用标准。这是由中国国防科学技术工业委员会批准, 适合于国防部门和军队使用的标准。例如, 1988 年实施的 GJB 437—88《军用软件开发规范》; GJB 438—88《军用软件文档编制规范》。

(3) DOD-STD (department of defense—standards)——美国国防部标准, 适用于美国国防部门。

(4) MIL-S(mILitary—standard)——美国军用标准, 适用于美军内部。

此外, 近年来中国许多经济部门(例如, 原航空航天部、原国家机械工业委员会、对外经济贸易部、石油化学工业总公司等)都开展了软件标准化工作, 制定和公布了一些适合于本部门工作需要的规范。这些规范大都参考了国际标准或国家标准, 对各自行业所属企业的软件工程项目起了有力的推动作用。

4) 企业规范

一些大型企业或公司, 由于软件工程工作的需要, 制定适用于本部门的规范。例如, 美国 IBM 公司通用产品部 (general products division), 1984 年制定的《程序设计开发指南》, 仅供该公司内部使用。

5) 项目规范

由某一科研生产项目组织制定, 且为该项任务专用的软件工程规范。例如, 计算机集成制造系统 (CIMS) 的软件工程规范。

9.1.4 中国的软件工程标准化工作

1983 年 5 月中国原国家标准总局和原电子工业部主持成立了“计算机与信息技术标准化技术委员会”, 下设十三个分技术委员会。与软件相关的程序设计语言分委员会和软件工程技术分委员会。中国制定和推行标准化工作的总原则是向国际标准靠拢, 对于能够在中国适用的标准一律按等同采用的方法, 以促进国际交流。这里, 等同采用是要使自己的标准与国际标准的技术内容完全相同, 仅稍作编辑性修改。

从 1983 年起到现在, 中国已陆续制定和发布了 20 项国家标准。这些标准可分为 4 类: 基础标准; 开发标准; 文档标准; 管理标准。

在表 9.2 所示的表中分别列出了这些标准的名称及其标准号。

表 9.2 中国的软件工程标准

分类	标 准 名 称	标 准 号
基础 标准	软件工程术语	GB/T 11457—89
	信息处理——数据流程图、程序流程图、系统流程图、程序网络图和系统资源图的文件编制符号及约定	GB 1526—891 (ISO 5807—1985)
	软件工程标准分类法	GB/T 15538—1995
	信息处理——程序构造及其表示法的约定	GB 13502—92 (ISO 8631)
	信息处理——单命中判定表规范	GB/T 15535—1995 (ISO 5806)
	信息处理系统 计算机系统配置图符号及其约定	GB/T 14085k 93 (ISO 8790)
开发 标准	软件开发规范	GB 8566—88
	计算机软件单元测试	GB
	软件支持环境	GB
	信息处理——按记录组处理顺序文卷的程序流程	GB***** (ISO 6593—1985)
	软件维护指南	GB/T 14079—93
文档 标准	软件文档管理指南	GB
	计算机软件产品开发文件编制指南	GB 8567—88
	计算机软件需求说明编制指南	GB 9385—88
	计算机软件测试文件编制指南	GB 9386—88
管理 标准	计算机软件配置管理计划规范	GB/T 12505—90
	信息技术 软件产品评价——质量特性及其使用指南	GB
	计算机软件质量保证计划规范	GB 12504—90
	计算机软件可靠性和可维护性管理	GB/T 14394k 93
	质量管理和质量保证标准 第三部分： 在软件开发、供应和维护中的使用指南	GB/T 19000—3—94

除去国家标准以外,近年来中国还制定了一些国家军用标准。根据国务院、中央军委在 1984 年 1 月颁发的军用标准化管理办法的规定,国家军用标准是指对国防科学技术和军事技术装备发展有重大意义而必须在国防科研、生产、使用范围内统一的标准。凡已有的国家标准能满足国防系统和部队使用要求的,不再制定军用标准。出于他们的特殊需要,近年已制定了以“ GJB ”为标记的软件工程国家军用标准 12 项。

9.2 软件质量认证

9.2.1 ISO 9000 系列标准及软件质量认证

近年来,国际上影响最为深远的质量管理标准当属国际标准化组织于 1987 年公布的

ISO 9000 系列标准了。这一国际标准发源于欧洲经济共同体,但很快就波及美国、日本及世界各国。到目前为止,已有70多个国家在它们的企业中采用和实施这一系列标准。中国对此也十分重视,采取了积极态度。一方面确定对其等同采用,发布了与其相应的质量管理国家标准系列 GB/ T 19000;同时积极组织实施和开展质量认证工作。计算机软件行业自然也和它领域一样被席卷进去。

分析 ISO 9000 系列标准如此迅速地在国际上广为流行,其原因主要在于:

1) 市场经济,特别是国际贸易的驱动。无论任何产业,其产品的质量如何都是生产者、消费者、以及中间商十分关注的问题。市场的竞争很大程度上反映了在质量方面的竞争。ISO 9000 系列标准客观地对生产者(也称供方)提出了全面的质量管理要求、质量管理方法,并且还规定了消费者(也称需方)的管理职责,使其得到双方的普遍认同,从而将符合 ISO 9000 标准的要求作为国际贸易活动中建立互相信任关系的基石。于是近年来在各国企业中形成了不通过这一标准认证就不具备参与国际市场竞争实力的潮流,并且在国际贸易中,把生产者是否达到 ISO 9000 质量标准作为购买产品的前提条件,取得 ISO 9000 质量标准认证被人们当作进入国际市场的通行证。

2) ISO 9000 系列标准适用领域广阔。它的出现最初针对制造行业,但现已面向更为广阔的领域,这包括:

- (1) 硬件:指不连续的具有特定形状的产品,如机械、电子产品,不只是计算机硬件。
- (2) 软件:通过支持媒体表达的信息所构成的智力产品。计算机软件当然属于其中。
- (3) 流程性材料:将原料转化为某一特定状态的产品。如,流体、粒状、线状等,通过瓶装、袋装等或通过管道传输交付。
- (4) 服务:为满足客户需求的更为广泛的活动。

9.2.2 ISO 9000 系列标准的内容

ISO 9000 系列标准为:

- ISO 9000 质量管理和质量保证标准——选择和使用的导则;
- ISO 9001 质量体系——设计/开发、生产、安装和服务中的质量保证模式;
- ISO 9002 质量体系——生产和安装中的质量保证模式;
- ISO 9003 质量体系——最终检验和测试中的质量保证模式;
- ISO 9004 质量管理和质量体系要素——导则。

ISO 9000 系列标准的主体部分可以分为两组:一组是用于“需方对供方要求质量保证”的标准——9001~9003;一组是用于“供方建立质量保证体系”的标准——9004。

9001,9002 和 9003 之间的区别,在于其对象的工序范围不同:9001 范围最广,包括从设计直到售后服务;9002 为 9001 的子集,而 9003 又是 9002 的子集。

9000 系列标准原本是为制造硬件产品而制定的标准,不能直接用于软件制作。后来,曾试图将 9001 改写用于软件开发方面,但效果不佳。于是,以 ISO 9000 系列标准的追加形式,另行制定出 9000- 3 标准。这样,9000- 3 就成了用于“使 9001 适用于软件开发、供应及维护”的“指南”。不过,在 9000- 3 的审议过程中,日本等国曾先后提出过不少意见。

所以, 在内容上与 9001 已有相当不同。参看图 9. 1。ISO 9000- 3(即 GB/ T19000. 3- 94), 全称为《质量管理和质量保证标准第三部分: 在软件开发、供应和维护中的使用指南》。

图 9. 1 ISO 9000- 3 在 ISO 9000 系列中的位置

9. 2. 3 制定与实施 ISO 9000 系列标准

1) 强调质量并非在产品检验中得到, 而是形成于生产的全过程。ISO 9000- 3 叙述了需方和供方应如何进行有组织的质量保证活动, 才能得到较为满意的软件; 规定了从双方签订开发合同到设计、实现以至维护整个软件生存期中应当实施的质量保证活动, 但并没有规定具体的质量管理和质量检验方法和步骤。

ISO 9000- 3 的核心思想是“ 将质量制作入产品之中 ”。其实道理是很明显的, 软件在完成编码以后, 不论花多大的力气用于测试, 提高质量都是有限度的, 更不必说需求规格说明存在的问题常常是测试无法发现的。事实上, 软件产品的质量取决于软件生存期所有阶段的活动。

2) 为把握产品的质量, ISO 9000 要求“ 必须使影响产品质量的全部因素在生产全过程中始终处于受控状态 ”。为使软件产品达到质量要求, ISO 9000- 3 要求软件开发机构建立质量保证体系。首先要求明确供需双方的职责, 针对所有可能影响软件质量的各个因素都要采取有力措施, 作出如何加强管理和控制的决定。对与质量有关的人员规定其职责和职权, 使之责任落实到人, 产品质量真正得到控制。

3) ISO 9000 标准要求证实:“ 企业具有持续提供符合要求产品的能力 ”。质量认证是取得这一证实的有效方法。产品质量若能达到标准提出的要求, 由不依赖于供方和需方的第三方权威机构对生产厂家审查证实后出具合格证明。显然, 如果这一认证工作是公正的、可靠的, 其公证的结果应当是可以信赖的。正确实施产品质量认证制度自然会在促进产品质量提高, 指导消费者选购产品, 提高质量合格产品企业的声誉, 以及节省社会检验大量费用等方面发挥积极作用。

生产企业为了达到质量标准,取得质量认证,必须多方面开展质量管理活动。其中,企业负责人的重视以及企业全体人员的积极参与是取得成功的关键。

4) ISO 9000 标准还强调“质量管理必须坚持进行质量改进”。贯彻 ISO 9000 标准是企业加强质量管理、提高产品质量的过程,这个过程包含许多工作,绝非轻而易举、一蹴而就所能奏效的。即使已经取得了质量认证也不能认为一劳永逸而放松质量管理。实际上认证通常以半年为有效期。取得认证之后尚需接受每年 1~2 次的定期检查,其目的在于促使企业坚持进行质量改进。

9.2.4 ISO 9000- 3 的要点

1) ISO 9000- 3 标准不适用于面向多数用户销售的程序包软件,仅适用于依照合同进行的单独的订货开发软件。也就是说,ISO 9000- 3 是用于按照双边合同进行的软件开发的过程中,需方彻底要求供方进行质量保证活动的标准。

ISO 9000- 3 也是用户企业的系统部门在建立质量保证系统时的指南。如果将使用部门看作是需方,将系统部门看作是供方,则可以将这两者之间的关系视为在企业内部以“双边合同”的形式进行软件开发的事例。

2) ISO 9000- 3 标准对供需双方领导的责任都作了明确的规定,并没有单纯地把义务全部加在供方身上。

标准要求需方设置“代表”,作为与供方联系的窗口。当委托软件厂家开发软件的需方为客户企业时,需方“代表”通常是该企业系统部门的人员。需方应当收集使用部门的意见,并归纳成为需方的要求,清楚地传达给供方。需方代表只有负责地明确所要求的技术条件,才可能对供方提出实施质量保证体系的要求。需方代表还应当承担责任,诸如敲定需方对供方的要求;回答来自供方的问题;批准供方的提案;与供方缔结协议;保证需方组织机构监督与供方签定协议的执行;确定接收标准和程序;处理需方提供的不合用的软件。

3) 在包括合同在内的全部工序中进行审查,并彻底文件化。具体来说,就是由需方与供方一起进行核查,找出含混不清的部分和问题,以便能及早消除将会产生麻烦的根源。核查结果都应以文件的方式体现出来。这样,文件就成为质量保证体系实施的“证据”。主要核查对象是:

- (1) 软件产品对需方规范的符合性;
- (2) 验证结果;
- (3) 接收测试结果。

在形成文件时。不是仅仅将问题列述出来,而是将当时所确认的内容全部记录在案。双方达成默契而不形成文件是不够的。由于这样作的结果,文件数量会很庞大。为了对这些文件进行保存、管理,在必要时调出来使用,必须用计算机实施。否则文件管理费工费时,将会成为整个开发作业的“瓶颈”。

所形成的文件,有可能成为提交给需方的质量活动报告,也将是供方通过 9000- 3 标

准认证审查时不可缺少的证据。

4) 在 ISO 9000- 3 中,最重要的是质量保证“ 体系 ”。ISO 9000- 3 是指南性的标准,叙述了需方与供方应如何合作进行有组织的质量保证活动才能制作出完美的软件,规定了从合同到设计、制作以至维护的整个生存期的全过程中应实施的质量保证活动;而没有规定具体的质量管理和测试等的方法和程序。其要点主要有:

(1) 强调软件质量保证体系是贯穿整个生存期的集成化过程体系,而不仅仅体现在最后产品验收时;

(2) 强调防患于未然而不是事后纠正;

(3) 更加强调质量体系的文件化;

(4) 强调对每一项软件开发都按计划开展质量活动,并且确保相关组织机构的了解和监督。其核心是“ 将质量制作入产品之中 ”。众所周知,在程序编制完成之后,不论再进行什么样的严格测试以消除缺陷,都已经为时已晚;而且从目前情况来看,软件由于初始的规格缺陷而发生大问题的事例并不少见。因此,必须建立质量保证体系,以避免发生上述问题。ISO 9000- 3 要求需方与供方双方首先整顿自己的组织体制;在实际进行软件开发方面,明确双方达成协议的事项和责任范围,对每道工序均进行检查并形成文件,就可以保证将质量制作入产品之中。

5) 供方应实施内部质量审核制度。具体来讲,有以下几方面内容:

(1) 要求供方为了进行质量保证活动而整顿其组织机构,设置质量保证管理负责人,建立程序与工序等明确的质量体系,并编制将程序与工序等形成文件的“ 质量手册(质量保证规定) ”。

(2) 在企业内部必须建立可以监督质量体系的体制。当认证制度正式实施以后,这一内部质量体系的监督,将会成为认证的重要内容。

(3) 对于每一个软件的开发活动,均应编制“ 质量计划 ”,以实施基于质量体系的质量保证活动,并形成相应的文件。

(4) ISO 9000- 3 中含有与 TQC(全面质量管理)相似的东西,但也有区别。TQC 的关键词是“ 协同作业 ”,而 ISO 9000- 3 是“ 权限与责任 ”。就是说,TQC 重视的是生产现场的自主性,而 ISO 9000- 3 则要求在领导层的指导下,以保持业务的一贯性为目的而开展质量保证活动。

(5) ISO 9000- 3 标准与具体的开发模式无关,它将软件全过程工序从管理角度、合同角度、工程角度分为三大类,列出适用于三大类的通用过程——文件化等的支援过程,以及过程的开发评估等。

(6) ISO 9000- 3 规定的是用以建立质量保证体系的“ 应作事项 ”的框架,其中并未规定具体的实施程序或文件格式等,就连质量的定义也未作出规定。

因此,在执行 ISO 9000- 3 时,必须引用其它的有关标准才能展开质量保证活动。在表 9.3 中列出了这些有关的标准。

表 9.3 与软件质量保证有关的主要标准

标准名称	说 明
ISO 8402	规定与质量(质量,质量体系或检验等)有关的22个“术语”。目前,ISO/TC 176正在修订,预计术语将增至67个。信息处理术语的定义引用自ISO 2382- 1。
ISO DIS 9000- 4 (IEC 300- 1)	适用于“可靠性管理”的标准。它将对软件可靠性管理产生影响。这一标准的内容与国际电工委员会的IEC 300- 1相同。
ISO/IEC 9126	对ISO 9000- 3未具体示出的软件“质量特性”作出规定的标准。具体规定了功能性、可靠性、可使用性、效率、可维护性和可移植性等6个主特性和21个二级特性。
ISO 13011- 1	对质量体系核查指南标准中的“核查步骤”作出规定。ISO 9000- 3在其第4章第3节(规定在供方内部进行质量体系核查)中引用了ISO 13011- 1。
软件开发全过程的管理	作为按双边合同进行软件开发时规定从合同到开发及维护的“作业项目的意义与位置”的标准。ISO/IEC JTC1正在审议。
软件配置管理	有关软件“配置要素”的标准,用以管理软件的调试修改,多次更改等业务。对于生产与维护效率的提高极重要。ISO/TC 176和ISO/IEC JTC 1正在审议。

9.3 在开发机构中推行软件工程标准化

组织开发和制定软件工程标准固然重要,在软件企业或其它软件开发机构中实施软件工程标准更为重要。特别是当前中国许多软件项目忽视工程化的情况下,尤其需要研究如何在软件企业中推行软件工程标准化的问题。

一种比较常见的现象是软件工程师并不真正理解软件工程标准化的意义。他们把贯彻标准看作是额外负担,是为了应付管理人员,或是认为与开发工作无关,与自己无关。有时尽管他们口头上也同意贯彻实施软件工程标准是十分必要的,但仍然会找到一些理由,力图说明标准的要求并不适用于他们的具体项目。

另一方面,的确有些标准并非真正适合于开发机构直接采用,有些标准不能反映当前的软件技术实践,也许有的标准被开发人员认为是过时的,难于实施。

为解决这些问题,把软件工程标准化工作切实地向前推进一步,抓住以下几点对于许多软件开发机构是有意义的。

- 1) 参考国际标准、国家标准或行业标准,制定适用于本单位软件开发的企业标准,编写软件工程标准化手册。自然,企业标准应当突出其实用性,所有规定都应当明确、具体,以便于实施。同时,企业标准应能体现国际标准、国家标准以及行业标准的原则,其所有规定不应与上述几类标准发生冲突。
- 2) 制定企业标准或是软件产品标准应当吸收软件工程师参加,让它们充分理解开发和实施标准的意义,以及它们自己在其中的责任。标准文件中不仅需要叙述应当遵循的要求,而且还需要扼要说明为什么要规定这些要求。
- 3) 为适应技术发展的形势,对已制定的标准,需要及时组织审查和更新。企业标准一经制定出来便载入标准手册,多年不作更动,就无法反映环境和技术变更的情况,这自然会暴露出实施的问题来,并且形成人们对它的种种偏见。

4) 贯彻标准应当得到辅助工具的支持。在制定企业标准时最好同时规划、配备工具,以提高贯彻标准的效率,解决开发人员经常抱怨的繁琐、乏味问题。如果有工具支持,便不会有太多的额外工作量加到开发中来,因而标准更容易为软件人员接受。

9.4 软件文档的作用与分类

9.4.1 软件文档的作用和分类

1) 什么是文档

文档(document)是指某种数据媒体和其中所记录的数据。它具有永久性,并可以由人或机器阅读,通常仅用于描述人工可读的东西。在软件工程中,文档常常用来表示对活动、需求、过程或结果进行描述、定义、规定、报告或认证的任何书面或图示的信息。它们描述和规定了软件设计和实现的细节,说明使用软件的操作命令。文档也是软件产品的一部分,没有文档的软件就不成其为软件。软件文档的编制(documentation)在软件开发工作中占有突出的地位和相当大的工作量。高质量、高效率地开发、分发、管理和维护文档对于转让、变更、修正、扩充和使用文档,对于充分发挥软件产品的效益有着重要的意义。然而,在实际工作中,文档在编制和使用中存在着许多问题,有待于解决。软件开发人员中较普遍地存在着对编制文档不感兴趣的现象。从用户方面来看,他们又常常抱怨:文档售价太高、文档不够完整、文档编写得不好、文档已经陈旧或是文档太多,难于使用等等。究竟应当怎样要求它,文档应当写哪些内容,说明什么问题,起什么作用?这里将给予简要的介绍。

2) 软件文档的作用

在软件的生产过程中,总是伴随着大量的信息要记录、要使用。因此,软件文档在产品的开发生产过程中起着重要的作用。

(1) 提高软件开发过程的能见度。把开发过程中发生的事件以某种可阅读的形式记录在文档中。管理人员可把这些记载下来的材料作为检查软件开发进度和开发质量的依据,实现对软件开发的工程管理。

(2) 提高开发效率。软件文档的编制,使得开发人员对各个阶段的工作都进行周密思考、全盘权衡、从而减少返工。并且可在开发早期发现错误和不一致性,便于及时加以纠正。

(3) 作为开发人员在一定阶段的工作成果和结束标志。

(4) 记录开发过程中的有关信息,便于协调以后的软件、开发、使用和维护。

(5) 提供对软件的运行、维护和培训的有关信息,便于管理人员、开发人员、操作人员、用户之间的协作、交流和了解。使软件开发活动更科学、更有成效。

(6) 便于潜在用户了解软件的功能、性能等各项指标,为他们选购符合自己需要的软件提供依据。

文档在各类人员、计算机之间的多种桥梁作用可从图 9.2 中看出。

既然软件已经从手工艺人的开发方式发展到工业化的生产方式,文档在开发过程中就起到关键作用。从某种意义上来说,文档是软件开发规范的体现和指南。按规范要求生

图 9.2 文档的桥梁作用

成一整套文档的过程是按照软件开发规范完成一个软件开发的过程。所以, 在使用工程化的原理和方法指导软件的开发和维护时, 应当充分注意软件文档的编制和管理。

3) 文档的分类

软件文档从形式上来看, 大致可分为两类: 一类是开发过程中填写的各种图表, 可称之为工作表格; 另一类是应编制的技术资料或技术管理资料, 可称之为文档或文件。

软件文档的编制可以用自然语言、特别设计的形式语言、介于两者之间的半形式语言(结构化语言)、各类图形表示、表格编制文档。文档可以书写, 也可以在计算机支持系统中产生, 但它必须是可阅读的。

按照文档产生和使用的范围, 软件文档大致可分为三类。

(1) 开发文档: 这类文档是在软件开发过程中, 作为软件开发人员前一阶段工作成果的体现和下一阶段工作依据的文档。包括软件需求说明书、数据要求说明书、概要设计说明书、详细设计说明书、可行性研究报告、项目开发计划。

(2) 管理文档: 这类文档是在软件开发过程中, 由软件开发人员制定的需提交管理人员的一些工作计划或工作报告。使管理人员能够通过这些文档了解软件开发项目安排、进度、资源使用和成果等。包括项目开发计划、测试计划、测试报告、开发进度月报及项目开发总结。

(3) 用户文档: 这类文档是软件开发人员为用户准备的有关该软件使用、操作、维护的资料。包括用户手册、操作手册、维护修改建议、软件需求说明书。

4) 软件文档的工作

国家标准局在 1988 年 1 月发布了《计算机软件开发规范》和《软件产品开发文件编制指南》, 作为软件开发人员工作的准则和规程。它们基于软件生存期方法, 把软件产品从形成概念开始, 经过开发、使用和不断增补修订, 直到最后被淘汰的整个过程应提交的文档归于以下 13 种。

(1) 可行性研究报告: 说明该软件项目的实现在技术上、经济上和社会因素上的可行性, 评述为合理地达到开发目标可供选择的各种可能的实现方案, 说明并论证所选定实施方案的理由。

(2) 项目开发计划: 为软件项目实施方案制定出的具体计划。它应包括各部分工作的负责人员、开发的进度、开发经费的概算、所需的硬件和软件资源等。项目开发计划应提供

给管理部门, 并作为开发阶段评审的基础。

(3) 软件需求说明书: 也称软件规格说明书。其中对所开发软件的功能、性能、用户界面及运行环境等作出详细的说明。它是用户与开发人员双方对软件需求取得共同理解基础上达成的协议, 也是实施开发工作的基础。

(4) 数据要求说明书: 该说明书应当给出数据逻辑描述和数据采集的各项要求, 为生成和维护系统的数据文件作好准备。

(5) 概要设计说明书: 该说明书是概要设计工作阶段的成果。它应当说明系统的功能分配、模块划分、程序的总体结构、输入输出及接口设计、运行设计、数据结构设计和出错处理设计等, 为详细设计奠定基础。

(6) 详细设计说明书: 着重描述每一个模块是如何实现的, 包括实现算法、逻辑流程等。

(7) 用户手册: 详细描述软件的功能、性能和用户界面, 使用户了解如何使用该软件。

(8) 操作手册: 为操作人员提供该软件各种运行情况的有关知识, 特别是操作方法细节。

(9) 测试计划: 针对组装测试和确认测试, 需要为组织测试制定计划。计划应包括测试的内容、进度、条件、人员、测试用例的选取原则、测试结果允许的偏差范围等。

(10) 测试分析报告: 测试工作完成以后, 应当提交测试计划执行情况的说明。对测试结果加以分析, 并提出测试的结论性意见。

(11) 开发进度月报: 该月报是软件人员按月向管理部门提交的项目进展情况的报告。报告应包括进度计划与实际执行情况的比较、阶段成果、遇到的问题和解决的办法, 以及下个月的打算等。

(12) 项目开发总结报告: 软件项目开发完成之后, 应当与项目实施计划对照, 总结实际执行的情况, 如进度、成果、资源利用、成本和投入的人力。此外, 还需对开发工作作出评价, 总结经验和教训。

(13) 维护修改建议: 软件产品投入运行之后, 可能有修正、更改等问题, 应当对存在的问题、修改的考虑以及修改的影响估计等作详细的描述, 写成维护修改建议, 提交审批。

以上这些软件文档是在软件生存期中, 随着各个阶段工作的开展适时编制的。其中, 有的仅反映某一个阶段的工作, 有的则需跨越多个阶段。图 9.3 给出了各种文档应在软件生存期中哪个阶段编写。

9.4.2 对文档编制的要求

为使软件文档能起到多种桥梁的作用, 使它有助于程序员编制程序, 有助于管理人员监督和管理软件的开发, 有助于用户了解软件的工作和应作的操作, 有助于维护人员进行有效的修改和扩充, 文档的编制必须保证一定的质量。

如果不重视文档编写工作, 或是对文档编写工作的安排不当, 就不可能得到高质量的文档。质量差的文档不仅使读者难于理解, 给使用者造成许多不便, 而且会削弱对软件的管理(难以确认和评价开发工作的进展情况), 提高软件成本(一些工作可能被迫返工), 甚至造成更加有害的后果(如误操作等)。

图 9.3 软件开发项目生存期各阶段与各种文档编制工作的关系

高质量的文档应当体现在以下几个方面

1) 针对性: 文档编制以前应分清读者对象。按不同的类型、不同层次的读者, 决定怎样适应他们的需要。例如, 管理文档主要是面向管理人员的, 用户文档主要是面向用户的, 这两类文档不应像开发文档(面向开发人员)那样过多使用软件的专用术语。

2) 精确性: 文档的行文应当十分确切, 不能出现多义性的描述。同一课题几个文档的内容应当是协调一致、没有矛盾的。

3) 清晰性: 文档编写应力求简明, 如有可能, 配以适当的图表, 以增强其清晰性。

4) 完整性: 任何一个文档都应当是完整的、独立的, 它应自成体系。例如, 前言部分应作一般性介绍, 正文给出中心内容, 必要时还有附录, 列出参考资料等。

同一课题的几个文档之间可能有些部分内容相同, 这种重复是必要的。不要在文档中出现转引其它文档内容的情况。例如, 一些段落没有具体描述, 而用“见××文档××节”的方式, 这将给读者带来许多的不便。

5) 灵活性: 各个不同软件项目, 其规模和复杂程度有着许多实际差别, 不能一律看待。

(1) 应根据具体的软件开发项目, 决定编制的文档种类。

软件开发的管理部门应该根据本单位承担的应用软件的专业领域和本单位的管理能力, 制定一个对文档编制要求的实施规定。主要是: 在不同条件下, 应该形成哪些文档? 这些文档的详细程度? 该开发单位每一个项目负责人都应当认真执行这个实施规定。

对于一个具体的应用软件项目, 项目负责人应根据上述实施规定, 确定一个文档编制计划。其中包括:

- 应该编制哪几种文档, 详细程度如何。

- 各个文档的编制负责人和进度要求。
- 审查、批准的负责人和时间进度安排。
- 在开发时期内各文档的维护、修改和管理的负责人, 以及批准手续。
- 有关的开发人员必须严格执行这个文档编制计划。

(2) 当所开发的软件系统非常大时, 一种文档可以分成几卷编写。例如,

- 项目开发计划可分写为: 质量保证计划、配置管理计划、用户培训计划、安装实施计划等。
- 系统设计说明书可分写为: 系统设计说明书、子系统设计说明书。
- 程序设计说明书可分写为: 程序设计说明书、接口设计说明书、版本说明。
- 操作手册可分写为: 操作手册、安装实施过程。
- 测试计划可分写为: 测试计划、测试设计说明、测试规程、测试用例。
- 测试分析报告可分写为: 综合测试报告、验收测试报告。
- 项目开发总结报告也可分写成: 项目开发总结报告、资源环境统计。

(3) 应根据任务的规模、复杂性、项目负责人对该软件的开发过程及运行环境所需详细程度的判断, 确定文档的详细程度。

(4) 对国标 GB8567- 88《计算机软件产品开发文件编制指南》所建议的所有条款都可以扩展, 进一步细分, 以适应需要; 反之, 如果条款中有些细节并非必需, 也可以根据实际情况压缩合并。

(5) 程序的设计表现形式, 可以使用程序流程图、判定表、程序描述语言 (PDL)、或问题分析图 (PAD) 等。

(6) 对于文档的表现形式, 没有规定或限制。可以使用自然语言、也可以使用形式化的语言。

6) 可追溯性: 由于各开发阶段编制的文档与各个阶段完成的工作有密切的关系, 前后两个阶段生成的文档, 随着开发工作的逐步延伸, 具有一定的继承关系, 在一个项目各开发阶段之间提供的文档必定存在着可追溯的关系。例如, 某一项软件需求, 必定在设计说明书、测试计划、甚至用户手册中有所体现。必要时应能作到跟踪追查。

9. 4. 3 文档的管理和维护

在整个软件生存期中, 各种文档作为半成品或是最终成品, 会不断生成、修改或补充。为了最终得到高质量的产品, 达到上节提出的质量要求, 必须加强对文档的管理。以下几个方面是应当作到的。

- 1) 软件开发小组应设一位文档保管员, 负责集中保管本项目已有文档的两套主文本。这两套主文本的内容完全一致。其中的一套可按一定手续, 办理借阅。
- 2) 软件开发小组的成员可根据工作需要在自己手中保存一些个人文档。这些一般都是主文本的复制件, 并注意与主文本保持一致, 在作必要的修改时, 也应先修改主文本。
- 3) 开发人员个人只保存着主文本中与它工作有关的部分文档。

表 9. 4 关于文档编制的十二项衡量因素

编号	因素	因素取值				
		1	2	3	4	5
1	创新程度	没有, 在不同设备上重编程序	有限, 只是具有更严格的要求	很多, 具有新的接口	大量, 应用新的现代开发技术	重大, 应用先进的开发和管理技术
2	通用程度	很强的限制单一目标	有限制, 功能的范围是量化的	有限的灵活性允许格式上有某些变化	多用途, 灵活的格式、有一个主题领域	很灵活, 能在不同的设备上处理范围广泛的主题
3	应用范围	局部单位(团以下)	本地应用(师级)	行业推广(军级)	全国推广(大军区)	国际项目(全军范围)
4	应用环境的变化	没有	很少	偶而有	经常	不断
5	设备复杂性	单机、常规处理	单机、常规处理、扩充的外设系统	多机、标准的外设系统	多机、复杂的外设系统和显示	主机控制系统多机自动 I/O
6	参加开发的人数	1~2 人	3~5 人	6~13 人	11~18 人	19 人以上
7	开发投资(人月)	6 以下	6~36	36~120	120~360	360 以上
8	重要程度	一般数据处理	常规过程控制	人身安全	单位成败	国家安危
9	完成程序修改的平均时间	2 周以上	1~2 周	3~7 天	1~3 天	24 小时以内
10	从数据输入到输出的平均时间	2 周以上	1~2 周	1~7 天	1~24 小时	1 小时以内
11	编程语言	高级语言	高级语言, 少量的汇编	高级语言, 带相当多的汇编	汇编语言	机器语言
12	并行软件开发	没有	有限	中等程度	很多	全部

4) 在新文档取代旧文档时, 管理人员应及时注销旧文档。文档的内容有更动时, 管理人员应随时修订主文本, 使其及时反映更新了的内容。

5) 项目开发结束时, 文档管理人员应收回开发人员的个人文档。发现个人文档与主文本有差别时, 应立即着手解决。这往往是在开发过程中没有及时修订主文本造成的。

6) 在软件开发的过程中, 可能发现需要修改已完成的文档。特别是规模较大的项目, 主文本的修改必须特别谨慎。修改以前要充分估计修改可能带来的影响, 并且要按照:
 ——提议——评议——审核——批准——实施——
 的步骤加以严格的控制。

表 9.5 因素值总和与文档编制种类的关系

文档的种类 因素值总和	可行性研究报告	项目开发计划	软件需求说明书	数据要求说明书	概要设计说明书	详细设计说明书	测试计划	用户手册	操作手册	测试分析报告	开发进度月报	项目开发总结	程序维护手册
12 ~ 18										*			
16 ~ 26				**									
24 ~ 38				**									
36 ~ 50				**									
48 ~ 60				**									

9.5 软件过程成熟度模型

9.5.1 软件机构的成熟性

多年来软件开发项目不能如期交付, 软件产品的质量不能令客户满意, 加之软件开发的开销超出项目开始时所作的预算, 这些是许多软件开发机构遇到的难题。近 20 年来, 不少人力图采用新的软件开发技术解决软件生产率和软件质量存在的问题, 但结果却不令人十分满意。这一现象促使人们进一步考察软件过程, 从而发现, 关键问题在于软件过程的管理不尽人意。事实表明, 在无规则和混乱的管理条件下, 先进的技术和工具并不能发挥应有的作用。人们认识到, 改进软件过程的管理是解决上述难题的突破口, 再也不能忽视软件过程的影响了。

我们观察到, 有时个别项目完成得比较好, 那是因为有个别优秀的软件人员参与工作, 并不是因为遵循了成熟的软件过程。要想使多个项目都能很好地完成, 不出现上述问题, 除非让这几个优秀的软件人员承担所有的项目。但这毕竟是不可能的。稳定、持续地保证软件高质量地完成, 只能依靠建立反映有效软件工程实践和管理实践的过程基础设施才能达到。让我们抛开个人因素不谈, 因为这在集体完成项目的过程中, 不是起决定性作用的。对于不同的软件开发机构, 在组织人员完成软件项目中所依据的管理策略有很大差别, 因而软件项目所遵循的软件过程也有很大差别。在此, 我们用软件机构的成熟度 (maturity) 加以区别。不成熟的软件机构有着自己的特征。

- (1) 软件过程一般在项目进行过程中由参与开发的人员临时确定。有时即使确定了, 实际上并不严格执行。
- (2) 软件机构是反应型的, 管理人员经常要集中精力去应付难以预料的突发事件。
- (3) 项目的进度和经费预算由于估计得不切实际, 所以常常突破。在项目进度拖延、交付时间紧迫的情况下, 往往不得不削减软件的功能, 降低软件的质量。
- (4) 产品质量难以预测。质量保证活动, 如质量评审、测试等, 常被削弱或被取消。成熟的软件机构具有的特点是:

建立了机构级的软件开发和维护过程。软件人员对其有较好的理解。一切活动均遵循过程的要求进行,作到工作步骤有次序,且有章可循。

软件过程必要时可作改进,但需在经小型试验和成本-效益分析基础上进行。

软件产品的质量和客户对软件产品的满意程度不是由开发人员,而是由负责质量保证的经理负责监控。

项目进度和预算是根据以往项目取得的实践经验确定,因而比较符合实际情况。

9.5.2 软件过程成熟度模型

在各个软件机构的过程成熟度有着相当大的差别面前,为了做出客观、公正的比较,需要建立一种衡量的标尺。使用这个标尺可以评价软件承包机构的质量保证能力,在软件项目评标活动中,选择中标机构。另一方面,这一标尺也必然成为软件机构改进软件质量,加强质量管理,以及提高软件产品质量的依据。

1987 年美国卡内基-梅隆大学软件工程研究所 SEI 受国防部资助,提出了软件机构的能力成熟度模型 CMM(capability maturity model),经过几年的使用及 1991 年和 1993 年两次修改,现已成为具有广泛影响的模型。

CMM 将软件过程的成熟度分为 5 个等级,如图 9.4 所示。以下给出具有 5 个等级的软件机构的特征。

图 9.4 软件过程成熟度模型

- (1) 初始级 (initial)
 - 工作无序,项目进行过程中常放弃当初的规划。
 - 管理无章,缺乏健全的管理制度。
 - 开发项目成效不稳定,优秀管理人员的管理办法可能取得成效,但他一离开,工作秩序面目全非,产品的性能和质量依赖于个人能力和行为。
- (2) 可重复级 (repeatable)
 - 管理制度化,建立了基本的管理制度和规程,管理工作有章可循。
 - 初步实现标准化,开发工作较好地实施标准。
 - 变更均依法进行,作到基线化。
 - 稳定可跟踪,新项目的计划和管理基于过去的实践经验,具有重复以前成功项目的环境和条件。
- (3) 已定义级 (defined)
 - 开发过程,包括技术工作和管理工作,均已实现标准化、文档化。
 - 建立了完善的培训制度和专家评审制度。
 - 全部技术活动和管理活动均稳定实施。
 - 项目的质量、进度和费用均可控制。

- 对项目进行中的过程、岗位和职责均有共同的理解。
- (4) 已管理级 (managed)
- 产品和过程已建立了定量的质量目标。
 - 过程中活动的生产率和质量是可度量的。
 - 已建立过程数据库。
 - 已实现项目产品和过程的控制。
 - 可预测过程和产品质量趋势, 如预测偏差, 实现及时纠正。
- (5) 优化级 (optimizing)
- 可集中精力改进过程, 采用新技术、新方法。
 - 拥有防止出现缺陷、识别薄弱环节以及加以改进的手段。
 - 可取得过程有效性的统计数据, 并可据此进行分析, 从而得出最佳的方法。

9.5.3 关键过程领域

除去初始级以外, 其它 4 级都有若干个引导软件机构改进软件过程的要点, 称为关键过程领域 KPA (key process area)。每一个关键过程领域是一组相关的活动, 成功地完成这些活动, 将会对提高过程能力起重要作用, 图 9.5 给出了各成熟度等级对应的关键过程领域。

图 9.5 关键过程领域

9.5.4 成熟度提问单

为把上述过程成熟度分级的方法推向实用化,需要为其提供具体的度量标尺。这个度量标尺就是成熟度提问单。CMM 在以下七个方面列出了大量的问题,每个问题都可针对特定的被评估软件机构给出肯定或否定的回答。提问单涉及的方面包括组织结构资源、人员及培训技术管理文档化标准及工作步骤过程度量数据管理和数据分析过程控制。

限于篇幅,在此不便于将全部成熟度提问单给出。为读者理解其主要内容,以下列举各个级别的部分提问单中的问题。

(1) 二级的问题

- 软件质量保证活动是否独立于软件开发的项目管理?
- 在接受委托开发合同以前,是否有严格的步骤进行软件开发的管理评审?
- 是否有严格的步骤用以估计软件的规模?
- 是否有严格的步骤用以得到软件开发的进度?
- 是否有严格的步骤用以估计软件开发的成本?
- 是否对代码错误和测试错误作了统计?
- 高层管理机构是否有一种机制对软件开发项目的状态进行正规的评审?
- 是否有一种机制能够控制软件需求的变更?
- 是否有一种机制控制代码变更?

(2) 三级的问题

- 是否有一个小组专门考虑软件工程过程的问题?
- 是否有针对软件开发人员的软件工程师培训计划?
- 是否有针对设计评审或代码评审负责人的正规培训计划?
- 每一个项目是否采用标准化、文档化的软件开发过程?
- 软件设计错误是否作了累积统计?
- 设计评审中提出的问题是否已追踪到底?
- 代码评审中提出的问题是否已追踪到底?
- 是否有一种机制确保对软件工程标准的依从性?
- 是否作过软件设计的内部评审?
- 是否有一种机制用以控制软件设计的变更?
- 是否作过代码评审?
- 是否有一种机制验证根据软件质量保证所检查的内容确能对整个工作具有代表性?
- 是否有机制保证回归测试的充分性?

(3) 四级的问题

- 是否有管理和支持引进新技术的机制?
- 是否有代码评审标准?
- 设计错误是否已预计到,是否与实际作过比较?
- 代码错误和测试错误是否已预计到,是否与实际作过比较?

- 设计评审与代码评审覆盖是否作了度量和记录?
- 对每一个功能测试覆盖是否作了度量和记录?
- 是否已为所有项目的过程度量数据建立了可控制的过程数据库?
- 在设计评审中收集到的评审数据是否得到分析?
- 为确定软件产品中残留的错误分布和特征, 是否对代码评审和测试中发现的错误数据做了分析?
- 是否对错误作了分析, 以便确定出错过程与其原因的关系?
- 项目评审的效率是否分析过?
- 是否有一种机制对软件工程过程实现定期评估, 并实施已经指出的改进?

(4) 五级的问題

- 是否有判别并替换过时技术的机制?
- 是否有分析错误原因的机制?
- 是否有一种机制可以得到回避错误的措施?
- 为确定避错所需的过程变更, 错误原因是否已经过评审?

第 10 章 软件管理

软件项目管理的对象是软件工程项目。它所涉及的范围覆盖了整个软件工程过程。为使软件项目开发获得成功,一个关键问题是必须对软件开发项目的工作范围、可能遇到的风险、需要的资源(人、硬/软件)、要实现的任务、经历的里程碑、花费工作量(成本)、以及进度的安排等等作到心中有数。而软件项目管理可以提供这些信息。通常,这种管理在技术工作开始之前就应开始,而在软件从概念到实现的过程中继续进行,并且只有当软件工程项目过程最后结束时才终止。

10.1 软件生产率和质量的度量

对于任何一个工程项目来说度量都是最基本的工作,软件工程也不例外。

10.1.1 软件度量

对于计算机软件来说,人们主要关心生产率与质量的度量,即以投入工作量为依据的软件开发活动的度量和开发成果质量的度量。从计划和估算的目的出发,人们总希望了解。在以往的项目中,软件开发生产率指的是什么?生产出的软件的质量又是什么?如何通过以往的生产率数据和质量数据来推断出现在的生产率和质量?它将如何帮助我们进行更精确的计划和估算?

软件度量分为两类。软件工程过程的直接度量包括所投入的成本和工作量。软件产品的直接度量包括产生的代码行数(LOC)、执行速度、存储量大小、在某种时间周期中所报告的差错数。而产品的间接度量则包括功能性、复杂性、效率、可靠性、可维护性和许多其它的质量特性。只要事先建立特定的度量规程,很容易作到直接度量开发软件所需要的成本和工作量、产生的代码行数等。但是,软件的功能性、效率、可维护性等质量特性却很难用直接度量判明,只有通过间接度量才能推断。

10.1.2 面向规模的度量

面向规模的度量是对软件和软件开发过程的直接度量。软件开发机构可以建立一个如图 10.1 所示的面向规模的数据表格来记录项目的某些信息。该表格列出了在过去几年完成的每一个软件开发项目和关于这些项目的相应面向规模的数据。如在图 10.1 中,项目 aaa- 01 的开发规模为 110.1 KLOC(千代码行),整个软件工程的(分析、设计、编码和测试)的工作量用了 24 个人月,成本为 168000 元。进一步地,开发出的文档页数为 365,在交付用户使用后第一年内发现了 29 个错误,有 3 个人参加了项目 aaa- 01 的软件开发工作。

对于每一个项目,可以根据表格中列出的基本数据进行简单的面向规模的生产率和

图 10.1 面向规模的度量

质量的度量。例如,可以根据图10.1 对所有的项目计算出平均值:

$$\text{生产率} = \text{KLOC} / \text{PM}(\text{人月})$$
$$\text{成本} = \text{元} / \text{LOC}$$

$$\text{质量} = \text{错误数} / \text{KLOC}$$
$$\text{文档} = \text{文档页数} / \text{KLOC}$$

10.1.3 面向功能的度量

面向功能的软件度量是对软件和软件开发过程的间接度量。面向功能度量的着眼点在程序的“功能性”和“实用性”上,而不是对 LOC 计数。该度量是由 Albrecht 首先提出来的。他提出了一种叫作功能点方法的生产率度量法,该方法利用软件信息域中的一些计数度量和软件复杂性估计的经验关系式而导出功能点 Fps (function points)。

功能点通过填写如图 10.2 所示的表格来计算。首先要确定 5 个信息域的特征,并在表格中相应位置给出计数。信息域的值以如下方式定义:

图 10.2 功能点度量的计算

- 1) 用户输入数: 是面向不同应用的输入数据, 对它们都要进行计数。输入数据应区别于查询数据, 它们应分别计数。
- 2) 用户输出数: 是为用户提供的面向应用的输出信息, 它们均应计数。在这里的“输出”是指报告、屏幕信息、错误信息等, 在报告中的各个数据项不应再分别计数。
- 3) 用户查询数: 查询是一种联机输入, 它导致软件以联机输出的方式生成某种即时的响应。每一个不同的查询都要计数。
- 4) 文件数: 每一个逻辑主文件都应计数。这里所谓的逻辑主文件是指逻辑上的一组数据, 它们可以是一个大的数据库的一部分, 也可以是一个单独的文件。

5) 外部接口数: 对所有被用来将信息传送到另一个系统中的机器可读写的接口 (即磁带或磁盘上的数据文件) 均应计数。

一旦收集到上述数据, 就可以计算出与每一个计数相关的复杂性值。使用功能点方法的机构要自行拟定一些准则以确定一个特定项是简单的、平均的还是复杂的。尽管如此, 复杂性的确定多少还是带点主观因素。

计算功能点, 使用关系式

$$FP = \text{总计数} \times [0.65 + 0.01 \times \text{SUM}(Fi)] \tag{10.1}$$

其中, 总计数是由图 10.2 所得到的所有加权计数项的和; Fi ($i=1$ 到 14) 是复杂性校正值, 它们应通过逐一回答图 10.3 所提问题来确定。SUM (Fi) 是求和函数。上述等式中的常数和应用于信息域计数的加权因数可经验地确定。

图 10.3 计算功能点的校正值

一旦计算出功能点, 就可以仿照 LOC 的方式度量软件的生产率、质量和其它属性:

生产率 = FP/PM (人月)	成本 = 元/ FP
质量 = 错误数/ FP	文档 = 文档页数/ FP

10.1.4 软件质量的度量

质量度量贯穿于软件工程的全过程中以及软件交付用户使用之后。在软件交付之前得到的度量提供了一个定量的根据, 以作出设计和测试质量好坏的判断。这一类度量包括程序复杂性、有效的模块性和总的程序规模。在软件交付之后的度量则把注意力集中于还未发现的差错数和系统的可维护性方面。

1) 影响软件质量的因素

20 多年以前, McCall 和 Cavano 定义了一组质量因素。这些因素从三个不同的方面评估软件质量:

- 产品的运行 (使用);
- 产品的修正 (变更);
- 产品的转移 (为使其在不同的环境中工作而作出修改, 即移植)。

2) 质量的度量

虽然已经有许多软件质量的度量方法, 但事后度量使用得最广泛。它包括正确性、可维护性、完整性和可使用性。

(1) 正确性: 一个程序必须正确地运行, 而且还要为它的用户提供某些输出。正确性要求软件执行所要求的功能。对于正确性, 最一般的度量是每千代码行(KLOC)的差错数, 其中将差错定义为已被证实是不符合需求的缺陷。差错在程序交付用户普遍使用后由程序的用户报告, 按标准的时间周期 (典型情况是 1 年) 进行计数。

(2) 可维护性: 软件维护比起其它的软件工程活动来, 需要作更多的工作量。可维护性是指当程序中发现错误时, 要能够很容易地修正它; 当程序的环境发生变化时, 要能够很容易地适应之; 当用户希望变更需求时, 要能够很容易地增强它。直接度量可维护性的方法还没有, 必须采取间接度量。有一种简单的面向时间的度量, 叫作平均变更等待时间 MTTC (mean time to change)。这个时间包括开始分析变更要求、设计合适的修改、实现变更并测试它、以及把这种变更发送给所有的用户。一般地, 一个可维护的程序与那些不可维护的程序相比, 应有较低的 MTTC (对于相同类型的变更)。

(3) 完整性: 在计算机犯罪和病毒侵犯的年代里, 完整性的重要性在不断增加。这个属性度量一个系统抗拒对它的安全性攻击(事故的和人为的)的能力。软件的所有三个成份程序、数据和文档都会遭到攻击。

为了度量完整性, 需要定义两个附加的属性: 危险性和安全性。危险性是特定类型的攻击将在一给定时间内发生的概率, 它可以被估计或从经验数据中导出。安全性是排除特定类型攻击的概率, 它也可以被估计或从经验数据中导出。一个系统的完整性可定义为

$$\text{完整性} = (1 - \text{危险性}) \times (1 - \text{安全性})$$

其中, 对每一个攻击的危险性和安全性都进行累加。

(4) 可使用性: 在软件产品的讨论中, “用户友好性”这个词汇使用越来越普遍。如果一个程序不具有“用户友好性”, 即使它所执行的功能很有价值, 也常常会失败。可使用性力图量化“用户友好性”, 并依据以下四个特征进行度量。

- 为学习系统所需要的体力上的和智力上的技能;
- 为达到适度有效地使用系统所需要的时间;
- 当软件被某些人适度有效地使用时所度量的在生产率方面的净增值;
- 用户角度对系统的主观评价 (可以通过问题调查表得到)。

10. 1. 5 影响软件生产率的因素

Basili 和 Zelkowitz 确定了五种影响软件生产率的重要因素:

- 人的因素 软件开发组织的规模和专长;

- 问题因素 问题的复杂性和对设计限制, 以及需求的变更次数;
- 过程因素 使用的分析与设计技术、语言和 CASE 工具的有效性, 及评审技术;
- 产品因素 计算机系统的可靠性和性能;
- 资源因素 CASE 工具、硬件和软件资源的有效性。

对于一个给定的项目, 上述的某一因素的取值高于平均值 (条件非常有利), 那么与那些同一因素的取值低于平均值 (条件不利) 的项目比, 它的软件开发生产率较高。

对于上述的五个因素, 从高度有利到不利, 条件的改变将以表 10. 1 所示的方式影响生产率。

表 10. 1 影响生产率的因素

因 素	人的因素	问题因素	过程因素	产品因素	资源因素
近似的百分比变化	90	40	50	140	40

为了了解这些数字的含义, 可以假定有两个软件项目组, 他们的成员在使用相同的资源和过程方面具有同等的能力。一个项目组面对的是一个相对简单具有平均可靠性和性能需求的问题, 另一个项目组面对的是一个复杂的具有极高可靠性和性能目标值的问题。根据上面列出的数字, 第一个项目组呈现出的软件生产率 (在 40% 与 140% 之间) 可能会优于第二项目组。

10. 2 软件项目的估算

软件项目管理过程开始于项目计划。在作项目计划时, 第一项活动是估算。在作估算时往往存在某些不确定性, 使得软件项目管理人员无法正常进行管理而导致产品迟迟不能完成。现在已使用的实用技术是时间和工作量估算。因为估算是所有其它项目计划活动的基石, 且项目计划又为软件工程过程提供了工作方向, 所以我们不能没有计划就开始着手开发, 否则将会陷入盲目性。

10. 2. 1 对估算的看法

估算资源、成本和进度时需要经验、有用的历史信息、足够的定量数据和作定量度量的勇气。估算本身带有风险。

项目的复杂性对于增加软件计划的不确定性影响很大。复杂性越高, 估算的风险就越高。但是, 复杂性是相对度量, 它与项目参加人员的经验有关。例如, 一个实时系统的开发, 对于过去仅作过批处理应用项目的软件开发组来说是非常复杂的, 但对于一个过去开发过许多高速过程控制软件的软件小组来说可能就是很容易的了。此外, 这种度量一般用在设计或编码阶段, 而在软件计划时(设计和代码存在之前)使用就很困难。因此, 可以在计划过程的早期建立其它较为主观的复杂性评估, 如在第 10. 1 节所描述的功能点复杂性校正因素。

项目的规模对于软件估算的精确性和功效影响也比较大。因为随着软件规模的扩大,

软件相同元素之间的相互依赖、相互影响程度也迅速增加,因而估算时进行问题分解也会变得更加困难。由此可知,项目的规模越大,开发工作量越大,估算的风险越高。

项目的结构化程度也影响项目估算的风险。这里的结构性是指功能分解的简便性和处理信息的层次性。结构化程度提高,进行精确估算的能力就能提高,而风险将减少。

历史信息的有效性也影响估算的风险。回顾过去,我们就能够仿效作过的事,且改进出现问题的地方。在对过去的项目进行综合的软件度量之后,就可以借用来比较准确地进行估算,安排进度以避免重走过去的弯路,而总的风险也减少了。

风险靠对不确定性程度定量地进行估算来度量。此外,如果对软件项目的作用范围还不十分清楚,或者用户的要求经常变更,都会导致对软件项目所需资源、成本、进度的估算频频变动,增加估算的风险。计划人员应当要求在软件系统的规格说明中给出完备的功能、性能、接口的定义。更重要的是,计划人员和用户都应认识到经常改变软件需求意味着在成本和进度上的不稳定性。

10.2.2 软件项目计划的目标

软件项目管理人员在开发工作一开始就会面临一个问题:需要进行定量估算,可是确切的信息又不一定有效。进行详细的软件需求分析能够得到估算所必需的信息,然而分析常常要花费几周或几个月的时间才能完成,但估算却要求“现在”就要。

软件项目计划的目标是提供一个能使项目管理人员对资源、成本和进度作出合理估算的框架。这些估算应当在软件项目开始时的一个有限的时间段内作出,并且随着项目的进展定期进行更新。

10.2.3 软件的范围

软件项目计划的第一项活动是确定软件的范围。软件范围包括功能、性能、限制、接口和可靠性。在估算开始之前,应对软件的功能进行评价,并对其进行适当的细化以便提供更详细的细节。由于成本和进度的估算都与功能有关,因此常常采用功能分解。性能的考虑包括处理和响应时间的需求。约束条件则标识外部硬件、可用存储或其它现有系统对软件的限制。

软件范围最不明确的方面是可靠性的讨论。软件可靠性的度量已经存在,但在项目计划阶段难得用上。因此,可以按照软件的一般性质规定一些具体的要求以保证它的可靠性。例如,用于空中交通指挥系统或者宇宙飞船(两者都是与人有关的系统)的软件就不能失效,否则就会危及人身安全。一个销售管理系统或字处理器软件也不应失效,但失效的影响不一定那么严重。当然还不能像在软件范围的描述中那样精确地量化软件的可靠性,但可以利用该项目的性质帮助估算工作量和成本以保证可靠性。

10.2.4 软件开发中的资源

软件项目计划的第二个任务是对完成该软件项目所需的资源进行估算。图 10.4 把软件开发所需的资源画成一个金字塔,在塔的底部必须有现成的用以支持软件开发的工具——硬件工具及软件工具,在塔的高层是最基本的资源——人。通常,对每一种资源,应说

明以下四个特性:资源的描述;资源的有效性说明;资源在何时开始需要;使用资源的持续时间。最后两个特性统称为时间窗口。对每一个特定的时间窗口,在开始使用它之前应说明它的有效性。

图 10.4 软件开发所需的资源

1) 人力资源

在考虑各种软件开发资源时,人是最重要的资源。在安排开发活动时必须考虑人员的技术水平、专业、人数、以及在开发过程各阶段中对各种人员的需要。

计划人员首先估算范围并选择为完成开发工作所需要的技能。还要在组织状况(如管理人员、高级软件工程师等)和专业(如通信、数据库、微机等)两方面作出安排。对一些规模较大的项目,在整个软件生存期中,各种人员的参与情况是不一样的。图 10.5 画出了各类不同的人员随开发工作的进展在软件工程各个阶段的参与情况的典型曲线。

图 10.5 管理人员与技术人员的参与情况

在软件计划和需求分析阶段,对软件系统进行定义,主要工作是由管理人员和高级技术人员在作,初级技术人员参与较少。待到对软件进行具体设计、编码及测试时,管理人员逐渐减少对开发工作的参与,高级技术人员主要在设计方面把关,具体编码及调试参与较少,大量的工作将由初级技术人员去作。到了软件开发的后期,需要对软件进行检验、评价和验收,管理人员和高级技术人员又将投入很多的精力。

一个软件项目所需要的人数只能在对开发的工作量作出估算(例如,多少人月或多少人年)之后才能决定。

2) 硬件资源

硬件作为软件开发项目的一种工具而投入。在软件项目计划期间考虑三种硬件资源:

- 宿主机 (host machine)——软件开发时使用的计算机及外围设备;
- 目标机 (target machine)——运行已开发成功软件的计算机及外围设备;
- 其它硬件设备——专用软件开发时需要的特殊硬件资源。

宿主机连同必要的软件工具构成一个软件开发系统。通常这样的开发系统能够支持多种用户的需要,且能保持大量的由软件开发小组成员共享的信息。因为许多软件开发机构都有很多人需要使用开发系统,因此,计划人员应当仔细地规定所需的时间窗口,并且验证资源是否有效。

但在许多情况下,除了那些很大的系统之外,不一定非要配备专门的开发系统。因此,所谓硬件资源,可以认为是对现存计算机系统的使用,而不是去购买一台新的计算机。宿主机与目标机可以是同一种机型。

3) 软件资源

软件在开发期间使用了许多软件工具来帮助软件的开发。最早的软件工具是自展程序。一个初步的汇编语言翻译器用机器语言写成,可用它来开发更高级的汇编器。在已有软件能力的基础上,软件开发人员最终将其自展成高级语言的编译器及其它工具。

现在,软件工程人员在许多方面都使用类似于硬件工程人员所使用的计算机辅助设计和计算机辅助工程 (CAD/ CAE) 工具的软件工具集。这种软件工具集叫作计算机辅助软件工程 (CASE)。主要的软件工具可作如下分类。

(1) 业务系统计划工具集——业务系统计划工具集可提供一种“元语言”,依靠模型化一个组织的战略信息的需求,导出特定的信息系统。这些工具要回答一些简单但重要的问题,例如,业务关键数据从何处来? 这些信息又向何处去? 如何使用它们? 当它们在业务系统中传递时又如何变换? 要增加什么样的新信息? 业务系统计划工具帮助软件开发人员建立信息系统,把数据通过某一路线送到要求这些信息的地方,并接收外来信息。在最后的分析中,改进数据的传送,并促进判断的作出。

(2) 项目管理工具集——项目管理人员使用这些工具可生成关于工作量、成本及软件项目的持续时间的估算、定义开发策略及达到这一目标的必要的步骤、计划可行的项目进程安排、以及持续地跟踪项目的实施。此外,管理人员还可使用工具收集建立软件开发生产率 and 产品质量的那些度量数据。

(3) 支援工具——支援工具可以分为文档生成工具、网络系统软件、数据库、电子邮件、通报板、以及在开发软件时控制和管理所生成信息的配置管理工具。

(4) 分析和设计工具——分析和设计工具可帮助软件技术人员建立目标系统的模型。这些工具还帮助人们进行模型质量的评价。它们靠对每一个模型进行执行一致性和有效性的检验,帮助软件技术人员在错误扩散到程序中之前排除之。

(5) 编程工具——系统软件实用程序、编辑器、编译器及调试程序都是 CASE 中必不可少的部分。而除这些工具之外,还有一些新的有用的编程工具。面向对象的程序设计工具、第四代程序生成语言、高级数据库查询系统,还有一大批 PC 工具 (如表格软件) 都属于这一 CASE 工具类。

(6) 组装和测试工具——测试工具为软件测试提供了各种不同类型和级别的支持。有些工具,像路径覆盖分析器为测试用例设计提供了直接支持,并在测试的早期使用。其它工具,像自动回归测试和测试数据生成工具,在组装和确认测试时使用,它们能帮助减少在测试过程中所需要的工作量。

(7) 原型化和模拟工具——原型化和模拟工具是一个很大的工具集,它包括的范围从简单的窗口画图到实时嵌入系统时序分析与规模分析的模拟产品,功能杂而全。它们最基本的情况是,原型化工具把注意力集中在建立窗口和为使用户能够了解一个信息系统或工程应用的输入/输出域而提出的报告。它们最完全的情况是,使用模拟工具建立嵌入式的实时应用,例如,为一个精炼设备建立过程控制系统的模型,或者为一架飞机建立航空控制系统的模型。在系统建立之前,可以对用模拟工具建立起来的模型进行分析,有时还要执行,以便对所建议的系统的运行时间性能进行评价。

(8) 维护工具——维护工具可以帮助分解一个现存的程序并帮助软件技术人员理解这个程序。然而,软件技术人员必须利用直觉、设计观念和人的智慧来完成逆向(还原)工程过程及重建应用。这种人的成份也是逆向工程过程及重建应用工具的一个组成部分,并且不大可能在可预见的未来被完全自动化所代替。

(9) 框架工具——这些工具能够提供一个建立集成项目支撑环境(IPSE)的框架。在多数情况下,框架工具实际提供了数据库管理和配置管理的能力与一些实用工具,能够把各种工具集成到 IPSE 中。

4) 软件复用性及软件部件库

为了促成软件的复用,以提高软件的生产率和软件产品的质量,可建立可复用的软件部件库。根据需要,对软件部件稍作加工,就可以构成一些大的软件包。这要求这些软件部件应加以编目,以利引用,并进行标准化和确认,以利于应用和集成。在使用这些软件部件时,有两种情况必须加以注意。

- 如果有现成的满足要求的软件,应当设法搞到它。因为搞到一个现成的软件所花的费用比重新开发一个同样的软件所花的费用少得多。
- 如果对一个现存的软件或软件部件,必须修改它才能使用。这时必须多加小心、谨慎对待,因为修改时可能会引出新的问题。而修改一个现存软件所花的费用有时会大于开发一个同样软件所花的费用。

10.2.5 软件项目估算

在计算技术发展的早期,软件的成本在整个计算机系统的总成本中所占百分比很小。在软件成本的估算上,出现一个数量级的误差,其影响相对还是比较小的。但现在软件在大多数基于计算机的系统中已成为最昂贵的部分。如果软件成本估算的误差很大,就会使盈利变成亏损。对于开发者来说,成本超支可能是灾难性的。

软件成本和工作量的估算很难精确作出,因为变化的东西太多,人、技术、环境、政治都会影响软件的最终成本和开发的工作量。但是,软件项目的估算还是能够通过一系列系统化的步骤,在可接受的风险范围内提供估算结果。

10.2.6 分解技术

当一个待解决的问题过于复杂时,可以把它进一步分解,直到分解后的子问题变得容易解决为止。然后,分别解决每一个子问题,并将这些子问题的解答综合起来,从而得到原问题的解答。通常,这是解决复杂问题的最自然的一种方法。

软件项目估算是一种解决问题的形式,在多数情况下,要解决的问题(对于软件项目来说,就是成本和工作量的估算)非常复杂,因此,可以对问题进行分解,将其分解成一组较小的接近于最终解决的可控的子问题,再定义它们的特性。

1) LOC 和 FP 估算

LOC 和 FP 是两个不同的估算技术。但两者有许多共同特性。项目计划人员首先给出一个有界的软件范围的叙述,再由此叙述尝试着把软件分解成一些小的可分别独立进行估算的子功能。然后对每一个子功能估算其 LOC 或 FP (即估算变量)。接着,把生产率度量(如,LOC/PM 或 FP/PM)用作特定的估算变量,导出子功能的成本或工作量。将子功能的估算进行综合后就能得到整个项目的总估算。

LOC 或 FP 估算技术对于分解所需要的详细程度是不同的。当用 LOC 作为估算变量时,功能分解是绝对必要的且需要达到很详细的程度。而估算功能点所需要的数据是宏观的量,当把 FP 当作估算变量时所需要的分解程度可以不很详细。还应注意,LOC 可直接估算,而 FP 要通过估计输入、输出、数据文件、查询和外部接口的数目,以及在 10.1 节图 10.3 中描述的 14 种复杂性校正值间接地确定。

避开所用到的估算变量,项目计划人员可对每一个分解的功能提出一个有代表性的估算值范围。利用历史数据或凭实际经验(当其它的方法失效时),项目计划人员对每个功能分别按最佳的、可能的、悲观的三种情况给出 LOC 或 FP 估计值。记作 a, m, b 。当这些值的范围被确定之后,也就隐含地指明了估计值的不确定程度。接着计算 LOC 或 FP 的期望值 E 。

$$E = (a + 4m + b) / 6 \quad (\text{加权平均})$$

假定实际的 LOC 或 FP 估算结果落在最佳值与悲观值范围之外的概率很小。使用标准的统计技术可以计算估算值的标准偏差。

一旦确定了估算变量的期望值,就可以用作 LOC 或 FP 生产率数据。这时,计划人员可以采用下列两种不同方法中的一种进行估算。

(1) 所有子功能的总估算值除以相应于该估算的平均生产率度量,得到工作量估算。例如,如果假定总的 FP 估算值是 310,基于过去项目的平均 FP 生产率是 5.5 FP/PM,则项目的总工作量是

$$\text{工作量} = 310 / 5.5 = 56 \text{ PM (人月)}$$

(2) 每一个子功能的估算值乘上根据子功能复杂性修正了的生产率校正值,得到生产率度量:对平均复杂性的功能,使用平均生产率度量。然而,对于一个特定的子功能,根据其复杂性比平均复杂性高或是低的情况,将向上或向下(多少有点主观地)调整平均生产率度量。例如,如果平均生产率是 490 LOC/PM,那么对比平均复杂性高的子功能,估算的生产率可以仅仅为 300 LOC/PM,而简单的功能,可以是 650 LOC/PM。

必须特别注意, 为了反映通货膨胀、项目复杂性的增加、新人手或其它开发特性的影响, 应当随时修正平均生产率度量。

作为 LOC 和 FP 估算技术的一个实例, 考察一个为计算机辅助设计(CAD)应用而开发的软件包。系统定义指明, 软件是在一个工作站上运行, 其接口必须使用各种计算机图形设备, 包括鼠标器、数字化仪、高分辨率彩色显示器和激光打印机。

在这个实例中, 使用 LOC 作为估算变量。当然, FP 也可使用, 但这时需要进行在第 10.1 节讨论过的信息域取值的估算。

根据系统定义, 软件范围的初步叙述如下:“ 软件将从操作员那里接收 2 维或 3 维几何数据。操作员通过用户界面与 CAD 系统交互并控制它, 这种用户界面将表现出很好的人机接口设计特性。所有的几何数据和其它支持信息保存在一个 CAD 数据库内。要开发一些设计分析模块以产生在各种图形设备上显示的输出。软件要设计得能控制并能与各种外部设备, 包括鼠标器、数字化仪、激光打印机和绘图仪交互。”

现在假定进一步的细化已作过, 并且已识别出下列主要的软件功能: 用户界面和控制功能、二维几何分析、三维几何分析、数据库管理、计算机图形显示功能、外设控制、设计分析模块。通过分解, 可得到如表 10.2 所示的估算表。表中给出了 LOC 的估算范围。

表 10.2 估 算 表

功 能	最佳值 a	可能值 m	悲观值 b	期望值 E	元 / 行	行 / PM	成本 (元)	工作量(PM)
用户接口控制	1800	2400	2650	2340	14	315	32760	7.4
二维几何造型	4100	5200	7400	5380	20	220	107600	24.4
三维几何造型	4600	6900	8600	6800	20	220	136000	30.9
数据结构管理	2950	3400	3600	3350	18	240	60300	13.9
计算机几何显示	4050	4900	6200	4950	22	200	108900	24.7
外部设备控制	2000	2100	2450	2140	28	140	59920	15.2
设 计 分 析	6600	8500	9800	8400	18	300	151200	28.0
总 计				33360			656680	144.5

计算出的各功能的期望值放入表中的第 4 列。然后对该列垂直求和, 得到该 CAD 系统的 LOC 估算值 33360。

从历史数据求出生生产率度量, 即行/PM 和元/行。计划人员需要根据复杂性程度的不同, 对各功能使用不同的生产率度量值。在表中的成本和工作量这两列的值分别用 LOC 的期望值 E 与元/行相乘, 及用 LOC 期望值 E 与行/PM 相除得到。因此可得, 该项目总成本的估算值为 657000 元, 总工作量的估算值为 145 人月 (PM)。

2) 工作量估算

工作量估算是估算任何工程开发项目成本的最普遍使用的技术。每一项目任务的解决都需要花费若干人日、人月或人年。每一个工作量单位都对应于一定的货币成本, 从而可以由此作出成本估算。

类似于 LOC 或 FP 技术, 工作量估算开始于从软件项目范围抽出软件功能。接着给

出为实现每一软件功能所必须执行的一系列软件工程任务, 包括需求分析、设计、编码和测试。在表 10.3 中列出了各个软件功能和相关的软件工程任务, 表中对每一个软件功能提供了用人月(PM)表示的每一项软件工程任务的工作量估算值。横向和纵向的总计给出所需要的工作量。应当注意,“ 前期 ”的开发任务 (需求分析和设计), 花费了 75 个人月, 说明这些工作相当重要。

表 10.3 工作量估算表

除特别指出的地方之外, 都按人月估算工作量。

计划人员针对每一软件功能, 把与每个软件工程任务相关的劳动费用率记入表中费用率(元)这一行, 这些数据反映了“ 负担 ”的劳动成本, 即包括公司开销在内的劳动成本。一般来说, 对于每个软件工程任务, 劳动费用率都可能不同。高级技术人员主要投入到需求分析和早期的设计任务中, 而初级技术人员则进行后期设计任务、编码和早期测试工作, 他们所需成本比较低。因此, 在表中需求分析的劳动成本为 5200 元/PM, 比编码和单元测试的劳动成本高出 22% 。

最后一个步骤是计算每一个功能及软件工程任务的工作量和成本。如果工作量估算是依赖 LOC 或 FP 估算而实现的, 那么就得到两组能进行比较和调和的成本与工作量估算。如果这两组估算值合理地一致, 则估算值是可靠的。如果估算的结果不一致, 就有必要作进一步的检查与分析。

10.3 软件开发成本估算

软件开发成本主要是指软件开发过程中所花费的工作量及相应的代价。它不同于其它物理产品的成本, 它不包括原材料和能源的消耗, 主要是人的劳动的消耗。人的劳动消耗所需代价是软件产品的开发成本。另一方面, 软件产品开发成本的计算方法不同于其它物理产品成本的计算。软件产品不存在重复制造过程, 它的开发成本是以一次性开发过程所花费的代价来计算的。因此软件开发成本的估算, 应是从软件计划、需求分析、设计、编码、单元测试、组装测试到确认测试, 整个软件开发全过程所花费的代价作为依据的。

10.3.1 软件开发成本估算方法

对于一个大型的软件项目, 由于项目的复杂性, 开发成本的估算不是一件简单的事, 要进行一系列的估算处理。主要靠分解和类推的手段进行。

基本估算方法分为三类。

1) 自顶向下的估算方法

这种方法的主要思想是从项目的整体出发, 进行类推。即估算人员根据以前已完成项目所消耗的总成本(或总工作量), 来推算将要开发的软件的总成本(或总工作量), 然后按比例将它分配到各开发任务单元中去, 再来检验它是否能满足要求。Boehm 给出一个参考例子, 参看表 10.4。

表 10.4 软件开发各阶段工作量的分配

软件		库存情况更新		开发者 W. Ward		日期 2/ 8/ 82	
阶 段		项目任务		工作量分布 (1/ 53)		小计 (1/ 53)	
计 划 与 需 求		软件需求定义		5		6	
		开发计划		1			
产 品 设 计		产品设计		6		10	
		初步的用户手册		3			
		测试计划		1			
详 细 设 计		详细 PDL 描述		4		12	
		数据定义		4			
		测试数据及过程设计		2			
		正式的用户手册		2			
编码与单元测试		编码		6		16	
		单元测试结果		10			
组装与联合测试		按实际情况编写文档		4		9	
		组装与联合测试		5			
总 计						53	

这种方法的优点是估算工作量小, 速度快。缺点是对项目中的特殊困难估计不足, 估算出来的成本盲目性大, 有时会遗漏被开发软件的某些部分。

2) 自底向上的估计法

这种方法的主要思想是把待开发的软件细分, 直到每一个子任务都已经明确所需要的开发工作量, 然后把它们加起来, 得到软件开发的总工作量。这是一种常见的估算方法。它的优点是估算各个部分的准确性高。缺点是缺少各项子任务之间相互联系所需要的工作量, 还缺少许多与软件开发有关的系统级工作量 (配置管理、质量管理、项目管理)。所以往往估算值偏低, 必须用其它方法进行检验和校正。

3) 差别估计法

这种方法综合了上述两种方法的优点, 其主要思想是把待开发的软件项目与过去已完成的软件项目进行类比, 从其开发的各个子任务中区分出类似的部分和不同的部分。类

似的部分按实际量进行计算, 不同的部分则采用相应的方法进行估算。这种方法的优点是
可以提高估算的准确程度, 缺点是不容易明确‘ 类似 ’ 的界限。

10. 3. 2 专家判定技术

专家判定技术是由多位专家进行成本估算。由于单独一位专家可能会有种种偏见, 譬
如有乐观的、悲观的、要求在竞争中取胜的、让大家都高兴的种种愿望及政治因素等。因
此, 最好由多位专家进行估算, 取得多个估算值。

有多种方法把这些估算值合成一个估算值。例如, 一种方法是简单地求各估算值的中
值或平均值。其优点是简便, 缺点是可能会由于受一、二个极端估算值的影响而产生严重
的偏差。另一种方法是召开小组会, 使各位专家们统一于或至少同意某一个估算值。优点
是可以摒弃无根据的估算值, 缺点是一些组员可能会受权威或政治因素的影响。

10. 3. 3 软件开发成本估算的经验模型

软件开发成本估算是依据开发成本估算模型进行估算的。开发成本估算模型通常采
用经验公式来预测软件项目计划所需要的成本、工作量和进度数据。用以支持大多数模型
的经验数据都是从有限的一些项目样本中得到的。因此, 还没有一种估算模型能够适用于
所有的软件类型和开发环境, 从这些模型中得到的结果必须慎重使用。

1) Putnam 模型

这是 1978 年 Putnam 提出的模型, 是一种动态多变量模型。它是假定在软件开发的
整个生存期中工作量有特定的分布。这种模型是依据在一些大型项目 (总工作量达到或
超过 30 个人年) 中收集到的工作量分布情况而推导出来的, 但也可以应用在一些较小的
软件项目中。

大型软件项目的开发工作量分布可以用图 10. 6 所示的曲线表示。

图 10. 6 大型项目的工作量分布情况

该曲线的典型形状由 Lord Rayleigh 最早有分析地导出, 并由 Norden 使用收集到的

软件开发中的经验数据证实了这条曲线。因此,图 10.6 所示的工作量分布曲线被称为 Rayleigh-Norden 曲线。用 Rayleigh-Norden 曲线可以导出一个“软件方程”,把已交付的源代码(源语句)行数与工作量和开发时间联系起来。

$$L = C_k \cdot K^{\frac{1}{3}} \cdot t_d^{\frac{4}{3}}$$

其中,td 是开发持续时间(以年计),K 是软件开发与维护在内的整个生存期所花费的工作量(以人年计),L 是源代码行数(以 LOC 计),Ck 是技术状态常数,它反映出“妨碍程序员进展的限制”,并因开发环境而异。其典型值的选取如表 10.5 所示。

表 10.5 技术状态常数 Ck 的取值

Ck 的典型值	开发环境	开 发 环 境 举 例
2000	差	没有系统的开发方法,缺乏文档和复审,批处理方式。
8000	好	有合适的系统开发方法,有充分的文档和复审,交互执行方式。
11000	优	有自动开发工具和技术。

改写上述公式,得工作量公式

$$K = \frac{L^3}{C_k^3 \cdot t_d^4}$$

若引入一个劳动率因子(¥/人年),就可以从 K 得到开发费用。

从上述方程,还可以估算开发时间。

$$t_d = \left(\frac{L^3}{C_k^3 \cdot K} \right)^{\frac{1}{4}}$$

2) COCOMO 模型 (constructive cost model)

这是由 TRW 公司开发,Boehm 提出的结构型成本估算模型,是一种精确、易于使用的成本估算方法。在该模型中使用的基本量有以下几个:

- DSI (源指令条数) 定义为代码或卡片形式的源程序行数。若一行有两个语句,则算作一条指令。它包括作业控制语句和格式语句,但不包括注释语句。K DSI = 1000DSI。
- MM (度量单位为人月) 表示开发工作量。定义 1MM= 19 人日= 152 人时= 1/ 12 人年。
- TDEV (度量单位为月) 表示开发进度。它由工作量决定。

(1) 软件开发项目的分类

在 COCOMO 模型中,考虑开发环境,软件开发项目的总体类型可分为三种。

组织型(organic): 相对较小、较简单的软件项目。对此种软件,一般需求不那么苛刻。开发人员对软件产品开发目标理解充分,与软件系统相关的工作经验丰富,对软件的使用环境很熟悉,受硬件的约束较少,程序的规模不是很大(< 5 万行)。例如,批数据处理、科学计算模型、商务处理模型、熟悉的操作系统、编译程序、简单的库存/ 生产控制系统,均属此种类型。

嵌入式(embadded): 此种软件要求在紧密联系的硬件、软件 and 操作的限制条件下

运行, 通常与某些硬设备紧密结合在一起。因此, 对接口、数据结构、算法要求较高。软件规模任意。例如, 大而复杂的事务处理系统、大型/ 超大型的操作系统、航天用控制系统、大型指挥系统, 均属此种类型。

半独立型(semidetached): 对此种软件的要求介于上述两种软件之间, 但软件规模和复杂性都属于中等以上, 最大可达 30 万行。例如, 大多数事务处理系统、新的操作系统、新的数据库管理系统、大型的库存/ 生产控制系统、简单的指挥系统, 均属此种类型。

(2) COCOMO 模型的分类

COCOMO 模型按其详细程度分成三级: 即基本 COCOMO 模型、中间 COCOMO 模型、详细 COCOMO 模型。基本 COCOMO 模型是一个静态单变量模型, 它用一个以已估算出来的源代码行数(LOC)为自变量的 (经验) 函数计算软件开发工作量。中间 COCOMO 模型则在用 LOC 为自变量的函数计算软件开发工作量 (此时称为名义工作量) 的基础上, 再用涉及产品、硬件、人员、项目等方面属性的影响因素调整工作量的估算。详细 COCOMO 模型包括中间 COCOMO 模型的所有特性, 但用上述各种影响因素调整工作量估算时, 还要考虑对软件工程过程中每一步骤(分析、设计等)的影响。

使用这三种模型估算工作量和进度的基本(名义)公式都是

工作量: $MM = r (KDSI)^c$ 进度: $TDEV = a (MM)^b$

其中, 经验常数 r, c, a, b 取决于项目的总体类型。

(3) 基本 COCOMO 模型

通过统计 63 个项目的历史数据, 得到如表 10. 6 所示的基本 COCOMO 模型的工作量和进度公式。

表 10. 6 基本 COCOMO 模型的工作量和进度公式

总体类型	工 作 量	进 度
组 织 型	$MM= 10. 4(KDSI)^{1. 05}$	$TDEV= 10. 5(MM)^{0. 38}$
半独立型	$MM= 3. 0(KDSI)^{1. 12}$	$TDEV= 10. 5(MM)^{0. 35}$
嵌 入 型	$MM= 3. 6(KDSI)^{1. 20}$	$TDEV= 10. 5(MM)^{0. 32}$

利用上面公式, 可求得软件项目或分阶段求得各软件任务的开发工作量和开发进度。

(4) 中间 COCOMO 模型

进一步考虑以下 15 种影响软件工作量的因素, 通过定下乘法因子, 修正 COCOMO 工作量公式和进度公式, 可以更合理地估算软件(各阶段)的工作量和进度。

中间 COCOMO 模型的名义工作量与进度公式如表 10. 7 所示。

表 10. 7 中间 COCOMO 模型的名义工作量与进度公式

总体类型	工 作 量	进 度
组 织 型	$MM= 3. 2(KDSI)^{1. 05}$	$TDEV= 10. 5(MM)^{0. 38}$
半独立型	$MM= 3. 0(KDSI)^{1. 12}$	$TDEV= 10. 5(MM)^{0. 35}$
嵌 入 型	$MM= 10. 6(KDSI)^{1. 20}$	$TDEV= 10. 5(MM)^{0. 32}$

对 15 种影响软件工作量的因素 f_i 按等级打分, 如表 10.8 所列。

表 10.8 15 种影响软件工作量的因素 f_i 的等级分

工作量因素 f_i		非常低	低	正常	高	非常高	超高
产品因素	软件可靠性	0.75	0.88	1.00	1.15	1.40	
	数据库规模		0.94	1.00	1.08	1.16	
	产品复杂性	0.70	0.85	1.00	1.15	1.30	1.65
计算机因素	执行时间限制			1.00	1.11	1.30	1.66
	存储限制			1.00	1.06	1.21	1.56
	虚拟机易变性		0.87	1.00	1.15	1.30	
	环境周转时间		0.87	1.00	1.07	1.15	
人员因素	分析员能力		1.46	1.00	0.86	0.71	
	应用领域实际经验	1.29	1.13	1.00	0.91	0.82	
	程序员能力	1.42	1.17	1.00	0.86	0.70	
	虚拟机使用经验*	1.21	1.10	1.00	0.90		
	程序语言使用经验	1.41	1.07	1.00	0.95		
项目因素	现代程序设计技术	1.24	1.10	1.00	0.91	0.82	
	软件工具的使用	1.24	1.10	1.00	0.91	0.83	
	开发进度限制	1.23	1.08	1.00	1.04	1.10	

* : 这里所谓虚拟机是指为完成某一个软件任务所使用的硬件、软件的结合。

此时, 工作量计算公式改成

$$MM = r \times \prod_{i=1}^{15} f_i \times (KDSI)^c$$

例 1 一个 32KDSI 的声音输入系统是一个输入原型, 或是一个可行性表演模型。所需可靠性非常低, 因为它不打算投入生产性使用。把此模型看作半独立型软件。则有

$$MM = 3.0 (32)^{1.12} = 146$$

又查表知 $f_1 = 0.75$, 其它 $f_i = 1.00$, 则最终有

$$MM = 146 \times 0.75 = 110.$$

例 2 一个规模为 10KDSI 的商用微机远程通信的嵌入型软件, 使用中间 COCOMO 模型进行软件成本估算。其要求见表 10.9。

表 10.9 影响工作量的因素 f_i 的取值

影响工作量因素 f_i	情 况	取 值
1 软件可靠性	只用于局部地区, 恢复问题不严重	1.00 (正常)
2 数据库规模	20000 字节	0.94 (低)
3 产品复杂性	用于远程通信处理	1.30 (很高)
4 时间限制	使用 70% 的 CPU 时间	1.10 (高)
5 存储限制	64K 中使用 45K	1.06 (高)
6 机器	使用商用微处理机	1.00 (额定值)
7 周转时间	平均 2 小时	1.00 (额定值)

续表

影响工作量因素 f_i	情 况	取 值
8 分析员能力	优秀人才	0.86 (高)
9 工作经验	远程通信工作 3 年	1.10 (低)
10 程序员能力	优秀人才	0.86 (高)
11 工作经验	微型机工作 6 个月	1.00 (正常)
12 语言使用经验	12 个月	1.00 (正常)
13 使用现代程序设计技术	1 年以上	0.91 (高)
14 使用软件工具	基本的微型机软件	1.10 (低)
15 工期	9 个月	1.00 (正常)

程序名义工作量 $MM = 10.6 \times (10)^{\frac{1.20}{15}} = 44.38 \text{ (MM)}$

程序实际工作量 $MM = 44.38 \times \prod_{i=1}^{15} f_i = 44.38 \times 1.17 = 51.5 \text{ (MM)}$

开发所用时间 $TDEV = 10.5 \times (51.5)^{0.32} = 8.9 \text{ (月)}$

如果分析员与程序员的工资都按每月 6000 美元计算, 则该项目的开发人员的工资总额为

$51.5 \times 6000 = 309000 \text{ (美元)}$

(5) 详细 COCOMO 模型

详细 COCOMO 模型的名义工作量公式和进度公式与中间 COCOMO 模型相同。但工作量因素分级表(类似于上表), 分层、分阶段给出。针对每一个影响因素, 按模块层、子系统层、系统层, 有三张工作量因素分级表, 供不同层次的估算使用。每一张表中工作量因素又按开发各个不同阶段给出。

例如, 关于软件可靠性(RELY)要求的工作量因素分级表(子系统层), 如表 10. 10 所示。

表 10. 10 软件可靠性工作量因素分级表(子系统层)

<div>阶段 RELY 级别</div>	需求和 产品设计	详细 设计	编程及 单元测试	集成及 测试	综合
非学低	0.80	0.80	0.80	0.60	0.75
低	0.90	0.90	0.90	0.80	0.88
正常	1.00	1.00	1.00	1.00	1.00
高	1.10	1.10	1.10	1.30	1.15
非常高	1.30	1.30	1.30	1.70	1.40

使用这些表格, 可以比中间 COCOMO 模型更方便、更准确地估算软件开发工作量。

10. 4 软件项目进度安排

软件开发项目的进度安排有两种考虑方式。

- 1) 系统最终交付日期已经确定, 软件开发部门必须在规定期限内完成;
- 2) 系统最终交付日期只确定了大致的年限, 最后交付日期由软件开发部门确定。

后一种安排能够对软件开发项目进行细致的分析, 最好地利用资源, 合理地分配工作, 而最后的交付日期则可以在对软件进行仔细地分析之后再确定下来; 但前一种安排在实际工作中常遇到, 如不能按时完成, 用户会不满意, 甚至还会要求赔偿经济损失, 所以必须在规定的期限内合理地分配人力和安排进度。

进度安排的准确程度可能比成本估算的准确程度更重要。软件产品可以靠重新定价或者靠大量的销售来弥补成本的增加, 但是进度安排的落空, 会导致市场机会的丧失, 使用户不满意, 而且也会导致成本的增加。因此, 在考虑进度安排时, 要把人员的工作量与花费的时间联系起来, 合理分配工作量, 利用进度安排的有效分析方法严密监控软件开发的进展情况, 以使得软件开发的进度不致拖延。

10. 4. 1 软件开发小组人数与软件生产率

对于一个小型的软件开发项目, 一个人就可以完成需求分析、设计、编码和测试工作。但是, 随着软件开发项目规模的增大, 会有更多的人共同参与同一软件项目的工作。例如 10 个人 1 年可以完成的项目, 若让 1 个人干 10 年是不行的。因此, 需要多人组成开发小组共同参加一个项目的开发。但是, 当几个人共同承担软件开发项目中的某一任务时, 人与人之间必须通过交流来解决各自承担任务之间的接口问题, 即所谓通信问题。通信需花费时间和代价, 会引起软件错误增加, 降低软件生产率。

若两个人之间需要通信, 则称在这两个人之间存在一条通信路径。如果一个软件开发小组有 n 个人, 每两人之间都需要通信, 则总的通信路径有 $n \times (n-1)/2$ (条)。

图 10. 7 软件小组中的通信路径

如果把这 n 个人看作是一个无向图的 n 个顶点, 所有的通信路径构成这个无向图的连接 n 个顶点的 $n \times (n-1)/2$ 条边。假设一个人单独开发软件, 生产率是 5000 行/人年。若 4 个人组成一个小组共同开发这个软件, 则需要 6 条通信路径 (如图 10. 7(a) 所示)。若在每条通信路径上耗费的工作量是 250 行/人年。则小组中每个人的软件生产率降低为

$$5000 - 6 \times 250 / 4 = 5000 - 375 = 4625 \text{ 行/人年}$$

如果小组有 6 名成员, 通信路径增加到 15 条 (如图 10. 7(b) 所示)。每条通信路径消耗的工作量不变, 则小组中每个成员的软件生产率降低为

$$5000 - 15 \times 250 / 6 = 5000 - 625 = 4375 \text{ 行/人年}$$

从上述简单分析可知,一个软件任务由一个人单独开发,生产率最高;而对于一个稍大型的软件项目,一个人单独开发,时间太长。因此软件开发小组是必要的。事实上,通过软件开发小组开展质量保证活动,可以获得更完善的软件分析与设计,可以减少故障数,从而降低测试工作量。

10.4.2 任务的确定与并行性

当参加同一软件工程项目的人数不止一人的时候,开发工作就会出现并行情形。图 10.8 表示了一个典型的由多人参加的软件工程项目的任务图。

在图 10.8 中可以看到,软件开发进程中设置了许多里程碑。里程碑为管理人员提供了指示项目进度的可靠依据。当一个软件工程任务成功地通过了评审并产生了文档之后,一个里程碑就完成了。

软件工程项目的并行性提出了一系列的进度要求。因为并行任务是同时发生的,所以进度计划表必须决定任务之间的从属关系,确定各个任务的先后次序和衔接,确定各个任务完成的持续时间。

此外,项目负责人应注意构成关键路径的任务,即若要保证整个项目能按进度要求完成,就必须保证这些任务要按进度要求完成。这样就可以确定在进度安排中应保证的重点。

* : 项目阶段任务的里程碑

图 10.8 软件项目的并行性

10.4.3 制定开发进度计划

在 10.2 节讨论的每一项软件估算技术都能得出完成软件开发任务所需人月(或人年)数的估算值。图 10.9 给出了在整个定义与开发阶段工作量分配的一种建议方案。这

个分配方案也称为 40- 20- 40 规则。它指出在整个软件开发过程中, 编码的工作量仅占 20% , 编码前的工作量占 40% , 编码后的工作量占 40% 。

40- 20- 40 规则只应用来作为一个指南。实际的工作量分配比例必须按照每个项目的特点来决定。一般花费在计划阶段的工作量很少超过总工作量的 2% ~ 3% , 除非是具有高风险的巨额费用的项目。需求分析可能占项目工作量的 10% ~ 25% 。花费在分析或原型化上面的工作量应当随项目规模和复杂性成比例地增加。通常用于软件设计的工作量在 20% ~ 25% 之间。而用在设计评审与反复修改的时间也必须考虑在内。

由于软件设计已经投入了工作量, 因此其后的编码工作相对来说困难要小一些, 用总工作量的 15% ~ 20% 就可以完成。测试和随后的调试工作约占软件开发工作量的 30% ~ 40% 。所需要的测试量往往取决于软件的重要程度。如果软件与人命相关, 即如果软件失效将危及人的生命, 测试可能占有更高的百分比。

图 10.9 工作量的分配

进一步地, 由 COCOMO 模型可知, 开发进度 TDEV 与工作量 MM 的关系:

$$TDEV = a (MM)^b$$

如果想要缩短开发时间或想要保证开发进度, 必须考虑影响工作量的那些因素。按可减小工作量的因素取值。但即使如此, 最多也只能压缩到名义开发时间的 75% 。因此比较精确的进度安排可利用中间 COCOMO 模型或详细 COCOMO 模型。

10.4.4 进度安排的方法

软件项目的进度安排与任何一个多重任务工作的进度安排基本差不多, 因此, 只要稍加修改, 就可以把用于一般开发项目的进度安排的技术和工具应用于软件项目。

软件项目的进度计划和工作的实际进展情况, 对于较大的项目来说, 难以用语言叙述清楚。特别是表现各项任务之间进度的相互依赖关系, 需要采用图示的方法。以下介绍几种有效的图示方法。在这几种图示方法中, 有几个信息必须明确标明:

- 各个任务的计划开始时间, 完成时间;
- 各个任务完成的标志(即 文档编写和 评审);
- 各个任务与参与工作的人数, 各个任务与工作量之间的衔接情况;
- 完成各个任务所需的物理资源和数据资源。

1) 甘特图 (gantt chart)

甘特图用水平线段表示任务的工作阶段; 线段的起点和终点分别对应着任务的开工时间和完成时间; 线段的长度表示完成任务所需的时间。图 10.10 给出了一个具有 5 个任务的甘特图 (任务名分别为 A, B, C, D, E)。

如果这 5 条线段分别代表完成任务的计划时间, 则在横坐标方向附加一条可向右移动的纵线。它可随着项目的进展, 指明已完成的任务(纵线扫过的)和有待完成的任务(纵线尚未扫过的)。我们从甘特图上可以很清楚地看出各子任务在时间上的对比关系。

在甘特图中, 每一任务完成的标准, 不是以能否继续下一阶段任务为标准, 而是必须

图 10.10 甘特图

交付应交付的文档与通过评审为标准。因此在甘特图中, 文档编制与评审是软件开发进度的里程碑。甘特图的优点是标明了各任务的计划进度和当前进度, 能动态地反映软件开发进展情况。缺点是难以反映多个任务之间存在的复杂的逻辑关系。

2) 时标网状图(timescalar network)

为克服甘特图的缺点, 用具有时标的网状图来表示各个任务的分解情况, 以及各个子任务之间在进度上的逻辑依赖关系, 参看图 10.11。从图中可以看出各任务之间在进度上的依赖关系。例如, 在甘特图中, 我们并不知道任务 A 与任务 E 之间有什么关系。但从时标网状图中可以看出, 任务 A 分为三段, 任务 E 分为两段。E2 的开始取决于 A3 的完成。

图 10.11 时标网状图

时标网状图中的箭头(直线、折线)表示各任务间的(先决)依赖关系; 箭头上的名字表示任务代号; 箭头的水平长度表示完成该任务的时间; 而圆圈表示一个任务结束、另一个任务开始的事件。

图中还有一些虚箭头, 表示虚拟任务, 即事实上不存在的任务。引入虚箭头也是为了显式地给出任务间的(先决)依赖关系。

3) PERT 技术和 CPM 方法

PERT 技术叫作计划评审技术(program evaluation & review technique), CPM 方法叫作关键路径法(critical path method), 它们都是安排开发进度、制定软件开发计划的最常用的方法。它们都采用网络图描述一个项目的任务网络, 也就是从一个项目的开始到结束, 把应当完成的任务用图或表的形式表示出来。通常用两张表来定义网络图。一张表给出与一特定软件项目有关的所有任务(也称为任务分解结构), 另一张表给出应当按照什么样的次序来完成这些任务(有时称为限制表 restrictionList)。

下面举例说明。假定某一开发项目在进入编码阶段之后, 考虑如何安排三个模块 A, B, C 的开发工作。其中, 模块 A 是公用模块, 模块 B 与 C 的测试有赖于模块 A 调试的完成。模块 C 是利用现成已有的模块, 但对它要在理解之后作部分修改。最后直到 A、B 和 C 作组装测试为止。这些工作步骤按图 10.12 来安排。在此图中, 各边表示要完成的任务, 边上均标注任务的名字, 如“ A 编码 ”表示模块 A 的编码工作。边上的数字表示完成该任务的持续时间。图中有数字编号的结点是任务的起点和终点, 在图中, 0 号结点是整个任务网络的起点, 8 号结点是终点。图中足够明确地表明了各项任务的计划时间, 以及各项任务之间的依赖关系。

图 10.12 开发模块 A, B, C 的任务网络图

在组织较为复杂的项目任务时, 或是需要对特定的任务进一步作更为详细的计划时, 可以使用分层的任务网络图。图 10.13 表明, 在父图 No. 0 上的任务 P 和 Q 均已分解出对应的两个子图: No. 1 和 No. 2。

最后, 在软件工程项目中必须处理好进度与质量之间的关系。在软件开发实践中常常会遇到这样的事情, 当任务未能按计划完成时, 只好设法加快进度赶上去。但事实告诉我们, 在进度压力下赶任务, 其成果往往是以牺牲产品的质量为代价。还应当注意到, 产品的质量与生产率有着密切的关系。例如, 日本的许多产品能作到质量与生产率的一致。日本人有个说法: 在价格和质量上折衷是不可能的, 但高质量给生产者带来了成本的下降这一事实是可以理解的, 这里的质量是指的软件工程过程的质量。

10.4.5 项目的追踪和控制

我们已经知道, 软件项目管理的一项重要工作就是在项目实施过程中进行追踪和控制。可以用不同的方式进行追踪:

- 定期举行项目状态会议。在会上, 每一位项目成员报告他的进展和遇到的问题。

图 10.13 分层的任务网络图

- 评价在软件工程过程中产生的所有评审的结果。
- 确定由项目的计划进度安排的可能选择的正式的里程碑。
- 比较在项目资源表中列出的每一个项目任务的实际开始时间和计划开始时间。
- 非正式地与开发人员交谈,以得到他们对开发进展和刚冒头的问题的客观评价。

实际上有经验的项目管理人员已经使用了所有这些追踪技术。

软件项目管理人员还利用“控制”管理项目资源、覆盖问题、及指导项目工作人员。如果事情进行得顺利(即项目按进度安排要求且在预算内实施,各种评审表明进展正常且正在逐步达到里程碑),控制将是轻微的。但当问题出现的时候,项目管理人员必须实行控制以尽可能快地排解它们。在诊断出问题之后,在问题领域可能需要一些追加资源;人员可能要重新部署,或者项目进度要重新调整。

10.5 软件项目的组织与计划

10.5.1 软件项目管理的特点

软件项目管理的解决,涉及到系统工程学、统计学、心理学、社会学、经济学、乃至法律等方面的问题。需要用到多方面的综合知识,特别是要涉及到社会的因素、精神的因素、人的因素,比技术问题复杂得多。仅靠技术、工程或科研项目的效率、质量、成本和进度等问题很难得到较好的解决。必须结合我们的工作条件、人员和社会环境等多种因素。因此,简单地照搬国外的管理技术往往无法奏效。此外,管理技术的基础是实践,为取得管理技术的成果必须反复实践。很显然,管理能够带来效率,能够赢得时间,最终将在技术前进的道路上取得领先地位。

1) 软件项目的特点

软件产品与其它任何产业的产品不同,它是无形的,完全没有物理属性。对于这样看不见、摸不着的产品,难以理解,难于驾驭。但它确实是把思想、概念、算法、流程、组织、效率、优化等融合在一起了。要开发这样的产品,在许多情况下,用户一开始给不出明确的想法,提不出确切的要求。他说不清究竟他需要的是什么。

在开发的过程中,程序与其相关的文档资料常常需要修改。在修改的过程中又可能产生新的问题,并且这些问题很可能在过了相当长的时间以后才会发现。

文档资料工作的工作量在整个项目研制过程中占有很大的比重,是十分重要的工作。但从实践中看出,人们对它不感兴趣、认为是不得不作的苦差事,不愿认真地去作。因而直接影响了软件的质量。

软件开发工作技术性很强,要求参加工作的人员具有一定的业务水平和实际工作的经验。但事实上,人员的流动对工作的影响很大。离去的人员不但带走了重要的信息,还带走了工作经验。

2) 软件项目管理的困难

智力密集,可见性差——软件工程过程充满了大量高强度的脑力劳动。软件开发的成果是不可见的逻辑实体,软件产品的质量难以用简单的尺度加以度量。对于不深入掌握软件知识或缺乏软件开发实践经验的人员,是不可能领导作好软件管理工作。软件开发任务完成得好也看不见,完成得不好有时也能制造假象,欺骗外行的领导。

单件生产——在特定机型上,利用特定的硬件配置,由特定的系统软件或支撑软件的支持,形成了特定的开发环境。再加上软件项目特定的目标,采用特定的开发方法、工具和语言,使得软件具有独一无二的特色,几乎找不到与之完全相同的软件产品。这种建立在内容、形式各异的基础上的研制或生产方式,与其它领域中大规模现代化生产有着很大的差别,也自然会给管理工作造成许多实际困难。

劳动密集,自动化程度低——软件项目经历的各个阶段都渗透了大量的手工劳动,这些劳动又十分细致、复杂和容易出错。尽管近年来开展了软件工具和CASE的研究,但总体来说,仍远未达到自动化的程度。软件产业所处的这一状态,加上软件本身的复杂性,使得软件的开发和维护难于避免多种错误,软件的正确性难于保证,软件产品质量的提高自然受到了很大的影响。

使用方法繁琐,维护困难——用户使用软件需要掌握计算机的基本知识,或者接受专门的培训,否则面对多种使用手册、说明和繁琐的操作步骤,学会使用要花费很大力气。另一方面,如果遇到软件运行出了问题,且没有配备专职维护人员,又得不到开发部门及时的售后服务,软件的使用者更是徒唤奈何。

软件工作渗透了人的因素——为高质量地完成软件项目,充分挖掘人员的智力才能和创造精神,不仅要求软件人员具有一定的技术水平和工作经验,而且还要求他们具有良好的心理素质。软件人员的情绪和他们的工作环境,对他们工作有很大的影响。与其它行业相比,它的这一特点十分突出,必须给予足够的重视。

3) 造成软件失误的原因

在总结和分析足够数量失误的软件项目之后,看出其原因大多与管理工作有关。

在软件项目开始执行时遇到的问题往往是:可供利用的资料太少;项目负责人的责任不明确;项目的定义模糊;没有计划或计划过分粗糙;资源要求未按时作出安排而落空;没有明确规定子项目完成的标准;缺乏使用工具的知识;项目已有更动,但预算未随之改变。

在软件项目执行的过程中可能会发生:项目审查只注意琐事而走过场;人员变动造成对工作的干扰;项目进行情况未能定期汇报;对阶段评审和评审中发现的问题如何处置未

作出明确规定;资源要求并不像原来预计的那样大;未能作到严格遵循需求说明书;项目管理人员不足。

项目进行到最后阶段可能会发生:未作质量评价;取得的知识和经验很少交流;未对人员工作情况作出评定;未作严格的移交;扩充性建议未写入文档资料。

总之,问题涉及到软件项目研制中的计划制定、进度估计、资源使用、人员配备、组织机构和管理方法等软件管理的许多侧面。

4) 软件管理的主要职能

软件管理的主要职能包括:

制定计划——规定待完成的任务、要求、资源、人力和进度等。

建立组织——为实施计划,保证任务的完成,需要建立分工明确的责任制机构。

配备人员——任用各种层次的技术人员和管理人员。

指导——鼓励和动员软件人员完成所分配的工作。

检验——对照计划或标准,监督和检查实施的情况。

以下将针对软件项目管理的主要问题进行讨论。

10.5.2 制定计划

软件开发项目的计划涉及到实施项目的各个环节,带有全局的性质。计划的合理性和准确性往往关系着项目的成败。据美国联邦政府调查,因软件计划不当而造成项目失败占失败总数的一半以上。

计划应力求完备。要考虑到一些未知因素和不确定因素,考虑到可能的修改。计划应力求准确。尽可能提高所依据数据的可靠程度。

1) 制定计划的目的是进行风险分析

为了使软件项目取得成功,在正式开始之前,作好计划工作的必要性是显而易见的。制定计划的目的是要回答:这个软件项目的范围是什么?需要哪些资源?花费多少工作量?要用的成本有多少?以及进度如何安排等一系列问题。在开发工作尚未开始之前,准确回答这些问题,显然是十分困难的。因为需求分析还没有进行,就连一些最必要的信息也提不出来,采用估算的办法就成为不可避免的了。既然是凭着以往的开发经验作出估算,就很难达到准确,同时从估算出发,开展的项目必然带有一定的风险。很明显,估算的准确性越差,风险也就越大。研制的软件项目越复杂,规模越大,结构化程度越低,资源、成本、进度等因素的不确定性越大,承担这一项目所冒的风险也越大。组织软件项目必须事先认清可能构成风险的因素,并研究战胜风险的对策,只有这样才能避免出现灾难性的后果,取得项目预期的成果。

2) 软件计划的类型

针对不同的工作目标,软件计划可以有以下几种类型:

(1) 项目实施计划(或称为软件开发计划)——是软件开发的综合性计划,通常应包括任务、进度、人力、环境、资源、组织等多个方面。

(2) 质量保证计划——把软件开发的质量要求具体规定为每个开发阶段可以检查的质量保证活动。

- (3) 软件测试计划——规定测试活动的任务、测试方法、进度、资源、人员职责等。
- (4) 文档编制计划——规定所开发项目应编制的文档种类、内容、进度、人员职责等。
- (5) 用户培训计划——规定对用户进行培训的目标、要求、进度、人员职责等。
- (6) 综合支持计划——规定软件开发过程中所需要的支持, 以及如何获取和利用这些支持。
- (7) 软件分发计划——软件开发项目完成后, 如何提供给用户。

3) 项目实施计划中任务的划分

软件项目的实施, 如何进行工作的划分是实施计划首先应解决的问题。常用的计划结构有:

(1) 按阶段进行项目的计划工作 (phased project planning)

按软件生存期, 把全部项目开发工作划分为若干阶段 (究竟分为几个阶段, 由管理部门具体规定), 对每个阶段的工作作出计划。再把每个阶段的工作进一步分解为若干个任务, 作出任务计划。还要把任务细分为若干步骤, 作出步骤计划。这样三层次的计划成为整个项目计划的依据。显然, 过细地作好分层计划, 可以提高计划的精确度, 减少或及早地发现问题。

(2) 任务分解结构 (work breakdown structure)

按项目本身的实际情况进行自顶向下的结构化分解, 形成树形任务结构, 如图 10. 14 所示。进一步把工作内容、所需的工作量、预计完成的期限规定下来。

图 10. 14 任务的结构化分解

这样可以把划分后的工作落实到人, 作到责任明确, 便于监督检查。

(3) 任务责任矩阵 (task responsibility mAtrix)

在任务分解的基础上, 把工作分配给相关的人员, 用一个矩阵形表格表示任务的分工和责任。例如, 把图 10. 14 已分解的任务分配给五位软件开发人员, 图 10. 15 表明了利用任务责任矩阵表达的分工情况。从图中可以看出, 工作的责任和任务的层次关系都非常明确。

10. 5. 3 软件项目组织的建立

参加软件项目的人员如何组织起来, 使他们发挥最大的工作效率, 对成功地完成软件项目极为重要。开发组织采用什么形式, 要针对软件项目的特点来决定, 同时也与参与人员的素质有关。人的因素是不容忽视的参数。对于我们来说, 国情、体制、人员的工作习惯等都应作具体分析。了解并参考国外的作法是必要的, 但无论如何不应简单地搬用。

图 10.15 任务责任矩阵

1) 组织原则

在建立项目组织时应注意到以下原则:

(1) 尽早落实责任: 在软件项目工作的开始, 要尽早指定专人负责。使他有权进行管理, 并对任务的完成负全责。

(2) 减少接口: 在开发过程中, 人与人之间的联系是必不可少的, 存在着通信路径。从 10.4.1 节可知, 一个组织的生产率因完成任务中存在的通信路径数目增多而降低。因此, 要有合理的人员分工、好的组织结构、有效的通信, 减少不必要的生产率的损失。

(3) 责权均衡: 软件经理人员所负的责任不应比委任给他的权力还大。

2) 组织结构的模式

通常有三种组织结构的模式可供选择:

(1) 按课题划分的模式 (project format)

把软件开发人员按课题组成小组, 小组成员自始至终参加所承担课题的各项任务。他们应负责完成软件产品的定义、设计、实现、测试、复查、文档编制、甚至包括维护在内的全过程。

(2) 按职能划分的模式 (functional format)

把参加开发项目的软件人员按任务的工作阶段划分成若干个专业小组。要开发的软件产品在每个专业小组完成阶段加工 (即工序) 以后, 沿工序流水线向下传递。例如, 分别建立计划组、需求分析组、设计组、实现组、系统测试组、质量保证组、维护组等。各种文档资料按工序在各组之间传递。这种模式在小组之间的联系形成的接口较多, 但便于软件人员熟悉小组的工作, 进而变成这方面的专家。

各个小组的成员定期轮换有时是必要的, 为的是减少每个软件人员因长期作单调的工作而产生乏味感。

(3) 矩阵形模式 (matrix format)

这种模式实际上是以上两种模式的复合。一方面, 按工作性质, 成立一些专门组, 如开发组、业务组、测试组等; 另一方面, 每一个项目又有它的经理人员负责管理。每个软件人员属于某一个专门组, 又参加某一项目的工作。例如, 属于测试组的一个成员, 他参加了某一项目 (如 CIMS 的 MRP) 的研制工作, 因此他要接受双重领导 (一是测试组, 一是

该软件项目的负责人)。如图 10.16 所示。

图 10.16 软件开发组织的矩阵形模式

矩阵形结构的组织具有一些优点: 参加专门组的成员可在组内交流在各项目中取得的经验, 这更有利于发挥专业人员的作用。另一方面, 各个项目有专人负责, 有利于软件项目的完成。显然, 矩阵形结构是一种比较好的形式。

3) 程序设计小组的组织形式

通常认为程序设计工作是按独立方式进行的, 程序人员独立地完成任务。但这并不意味着互相之间没有联系。人员之间联系的多少和联系的方式与生产率直接相关。程序设计小组内人数少, 如 2~3 人, 则人员之间的联系比较简单。但在增加人员数目时, 相互之间的联系复杂起来, 并且不是按线性关系增长。并且, 已经进行中的软件项目在任务紧张、延误了进度的情况下, 不鼓励增加新的人员给予协助。除非分配给新成员的工作是比较独立的任务, 并不需要对原任务有更细致的了解, 也没有技术细节的牵连。

小组内部人员的组织形式对生产率也有影响。现有的组织形式有三种。

(1) 主程序员制小组 (chief programmer team)

小组的核心由一位主程序员 (高级工程师)、二至五位技术员、一位后援工程师组成。主程序员负责小组全部技术活动的计划、协调与审查工作, 还负责设计和实现项目中的关键部分。技术员负责项目的具体分析与开发, 以及文档资料的编写工作。后援工程师支持主程序员的工作, 为主程序员提供咨询, 也作部分分析、设计和实现的工作。并在必要时能代替主程序员工作, 以便使项目能继续进行。

主程序员制小组还可以由一些专家 (例如, 通信专家或数据库设计专家)、辅助人员 (例如, 打字员和秘书)、软件资料员协助工作。软件资料员可同时为多个小组服务, 并完成下列工作: 保存和管理所有软件配置项。即文档资料、源程序清单、数据和磁介质资料等; 协助收集和整理软件生产率数据; 对可复用的模块进行分类及编写索引; 协助小组进行调查、评价和准备文档等。参看图 10.17。

图 10.17 主程序员小组的组织

主程序员制的开发小组突出了主程序员的领导。强调主程序员与其他技术人员的直接联系。总的来说简化了人际通信,参看图 10.18(a)。这种集中领导的组织形式能否取得好的效果,很大程度上取决于主程序员的技术水平和管理才能。

图 10.18 三种不同的小组结构

(上排的三种为结构形式,下排的三种为通信路径)

(2) 民主制小组 (democratic team)

在民主制小组中,遇到问题,组内成员之间可以平等地交换意见,参见图 10.18(b)。工作目标的制定及作出决定都由全体成员参加。虽然也有一位成员当组长,但工作的讨论、成果的检验都公开进行。这种组织形式强调发挥小组每个成员的积极性,要求每个成员充分发挥主动精神和协作精神。在讨论时尊重每个成员,充分听取他们的意见,并要求大家互相学习,在组内形成一个良好合作的工作气氛。但有时也会因此削弱了个人的责任心和必要的权威作用。有人认为这种组织形式适合于研制时间长、开发难度大的项目。

(3) 层次式小组 (hierarchical team)

在层次式小组中,组内人员分为三级:组长(项目负责人)一人负责全组工作,包括任务分配、技术评审和走查(walkthrough)、掌握工作量和参加技术活动。他直接领导二至三名高级程序员,每位高级程序员通过基层小组,管理若干位程序员。这种组织结构只允许必要的人际通信。比较适用于项目本身就是层次结构的课题。参看图 10.18(c)。

这种结构比较适合项目本身就是层次结构状的课题。因为这样可以把项目按功能划分成若干个子项目,把子项目分配给基层小组,由基层小组完成。例如,具有三个子项目的课题由具有三个基层小组的层次式小组完成。基层小组的领导与项目负责人直接联系。通常基层小组的人数不超过十人。因而,一个大型项目需要划分成若干层。这种组织方式比较适合于大型软件项目的开发。

以上三种组织形式可以根据实际情况,组合起来灵活运用。例如,较大的软件项目也许是把主程序员小组组织成层次式结构;也许基层小组的领导又是一个民主制小组的成员。

10.5.4 人员配备

如何合理地配备人员是成功地完成软件项目的切实保证。所谓合理地配备人员应包括按不同阶段适时任用人员,恰当掌握用人标准。

1) 项目开发各阶段所需人员

一个软件项目完成的快慢取决于参与开发人员的多少。在开发的整个过程中,多数软件项目是以恒定人力配备的。如图 10.19 所示。

图 10.19 软件项目的恒定人力配备

图 10.19 中的人力需求曲线是在 10.3 节介绍 Putnam 模型时引入的 Rayleigh - Norden 工作量分布曲线。该曲线表示需求定义结束后的软件生存期(包括运行和维护)内的工作量分布情况,它也是投入人力与开发时间的关系曲线。按此曲线,需要的人力随开发的进展逐渐增加,在编码与单元测试阶段达到高峰,以后又逐渐减少。如果恒定地配备人力,在开发的初期,将会有部分人力资源用不上而浪费掉。在开发的中期(编码与单元测试),需要的人力又不够,造成进度的延误。这样在开发的后期就需要增加人力以赶进度。因此,恒定地配备人力,对人力资源是比较大的浪费。

2) 配备人员的原则

配备软件人员时,应注意以下三个主要原则:

- 重质量——事实表明,软件项目是技术性很强的工作,任用少量有实践经验、有能力的人员去完成关键性的任务,常常要比使用较多的经验不足的人员更有效。
- 重培训——花力气培养所需的技术人员和管理人员是有效解决人员问题的好方法。
- 双阶梯提升——人员的提升应分别按技术职务和管理职务进行,不能混在一起。

3) 对项目经理人员的要求

软件经理人员是工作的组织者, 他的管理能力的强弱是项目成败的关键。除去一般的管理要求外, 他应具有以下能力:

- 把用户提出的非技术性的要求加以整理提炼, 以技术说明书的形式转告给分析员和测试员。
- 能说服用户放弃一些不切实际的要求, 以便保证合理的要求得以满足。
- 能够把表面上似乎无关的要求集中在一起, 归结为“需要什么”, “要解决什么问题”。这是一种综合问题的能力。
- 要懂得心理学, 能说服上级领导和用户, 让他们理解什么是不合理的要求。但又要使他们毫不勉强, 乐于接受, 并受到启发。

4) 评价人员的条件

软件项目中人的因素越来越受到重视。在评价和任用软件人员时, 必须掌握一定的标准。人员素质的优劣常常影响到项目的成败。能否达到以下这些条件是不应忽视的。

- 牢固掌握计算机软件的基本知识和技能。
- 善于分析和综合问题, 具有严密的逻辑思维能力。
- 工作踏实、细致, 不靠碰运气, 遵循标准和规范, 具有严格的科学作风。
- 工作中表现出有耐心、有毅力、有责任心。
- 善于听取别人的意见, 善于与周围人员团结协作, 建立良好的人际关系。
- 具有良好的书面和口头表达能力。

10.5.5 指导与检验

指导的目的是在软件项目的过程中, 动员和促进工作人员积极完成所分配的任务。实际上, 指导也是属于人员管理的范围, 是组织好软件项目不可缺少的工作。

检验是软件管理的最后一个方面。它是对照计划检查执行情况的过程, 同时也是对照软件工程标准检查实施情况的过程。在发现项目的实施与计划或标准有较大的偏离时, 应采取措施加以解决。

1) 指导工作的要点

在指导软件项目时需注意到以下几个方面的问题。

(1) 鼓励——对工作的兴趣和取得显著成绩常常能够成为推动工作的积极因素。恰当而且及时地鼓励是非常重要的。它可使人们充满信心, 勇于继续克服困难, 愿意努力进一步提供工作效率, 迎接新任务的挑战。

(2) 引导——通常, 人们愿意追随那些能够体谅个人要求或实际困难的领导。高明的领导人应能注意到这些, 并能巧妙地把个人的要求和目标与项目工作的整体目标结合起来, 至少应能作到在一定程度上的协调, 而不应眼看着矛盾的存在和发展, 以致影响工作的开展。对于合适的人员, 应让他们喜欢在你这里工作, 不愿离去。从风险分析中可知, 大幅度的人员调整是非常有害的, 它会带来许多实际问题。即使是人员的临时观念也都要使项目付出不可见的代价, 因而蒙受无形的损失。

(3) 通信——在软件项目中充满了人际通信联系。必要的通信联络肯定是不可少的。

但实践表明, 软件生产率, 即工作效率是通信量的函数。如果人际通信数量过大, 会使软件生产率迅速下降。

2) 检验管理的要点

在检验管理时应注意以下问题。

(1) 重大偏离——在软件项目实施过程中, 必须注意发现工作的开展与已制定的计划之间、或与需遵循的标准(或规范)之间的重大偏离。遇到有这种情况应及时向管理部门报告并采取相应的措施给予适当的处置。

(2) 选定标准——检验管理需要事先确定应当遵循的标准(或规范), 使得软件项目的工作进展可以用某些客观、精确且有实际意义的标准加以衡量。

(3) 特殊情况——任何事务在一般规律之外都会存在一些特殊情况。管理人员必须把注意力放在软件项目实施的一些特殊情况上, 认真分析其中的一些特殊问题, 加以解决。

3) 检验管理的工作范围

检验管理在软件项目中可能涉及到以下几个方面。

(1) 质量管理——包括明确度量软件质量的因素和准则, 决定质量管理的方法和工具, 以及实施质量管理的组织形式。

(2) 进度管理——检验进度计划执行的情况。

(3) 成本管理——度量并控制软件项目的开销。

(4) 文档管理——检验文档编写是否符合要求。

(5) 配置管理——检验软件配置。

4) 软件项目中人的因素

软件产品是人们大量智力劳动的结晶, 软件项目能否获得成功, 人的因素所起的作用比其它任何工程项目都突出。B. W. Boehm 在其著作“Software Engineering Economics”的封面上, 把影响软件生产率的因素从小到大顺序进行排列(参见图 10. 20), 图中显示, 用户是否参与及软件人员的能力等人的因素对软件生产率的影响极大。在与软件成本相关的 14 种影响因素中, 人员的能力是最大影响因素。它表明, 如果在软件项目中能够充分发挥软件人员的积极性, 使他们的才能得到尽量的施展, 软件生产率(以单位时间内开发出的源程序的平均行数)可提高 4 倍多, 有时还会有更大的差别, 高达 20 ~ 30 倍。

著名的软件工程专家 Tom DeMarco 在其著作“Productive Project and Team”中专题讨论了软件生产中人的因素问题。积 30 年软件项目管理的经验, 他认为软件项目中对于人员的管理问题不能像其它事物那样简单地划分, 机械地对待。

在该著作中, 作者考察了许多软件开发项目的实际情况, 特别注意到项目的规模、成本、缺陷、加快开发的因素以及执行进度计划中的种种问题。积累了 500 个项目开发过程的数据, 从中发现大约 15% 的项目失败了。有的是一开始就被撤销, 有的中途流产, 有的推迟了进度, 有的成果不能投入使用。而且项目规模越大, 情况越糟。究其原因, 绝大多数失败的项目竟找不出一个可以说得出口的技术障碍; 而障碍却来自人员之间的联系问题、人员的任用问题、计算机语言掌握得怎样, 对生产率的影响充其量不过 1. 2 位, 无论使用多么好的软件工具, 生产率的提高范围也只能在 50% 左右。但个人能力和小组活动对软件

图 10.20 与软件成本相关的各种属性对生产率波动范围的影响

生产影响极大。对上级或对雇主失望、工作缺乏动力或缺乏高额工程维持费用等。这些人际关系问题的解决可归结于“软件项目社会学”。

关于软件人员的办公环境,有许多因素影响软件工作的效率。DeMarco 曾于 1984 年到 1986 年在 62 个公司的 600 名软件人员进行编码和测试竞赛活动,并对竞赛结果进行统计分析。结果表明,除了对语言的熟悉程度、工作年限、工资收入等因素以外,环境因素起着很大的作用。良好的办公环境可保证软件人员高质量地完成任务。这里所说的办公环境是指每个软件人员的办公室工作面积、办公环境安静程度、专用程度、电话干扰程度、工作时间内外面来访人员次数等。DeMarco 说,你如果是一名项目管理人员,你为软件人员安排了任务,提供了工作条件,而对工作环境所带来的影响估计不足,你还是应该承担责任。

10.6 软件配置管理

软件配置管理,简称 SCM,是一种“保护伞”活动,它应用于整个软件工程过程。因为变更在任何时刻都可能发生,因此 SCM 活动的目标是为了(1)标识变更;(2)控制变更;(3)确保变更正确地实现;(4)向其他有关的人报告变更。

软件维护和软件配置管理之间的区别是:维护是一组软件工程活动,它们发生于软件已交付给用户并已投入运行之后;软件配置管理是一组追踪和控制活动,它们开始于软件开发项目开始之时,结束于软件被淘汰之时。

10.6.1 软件配置管理

软件工程过程的输出有三种信息:(1) 计算机程序(源程序及目标程序);(2) 描述计算机程序的文档(包括技术文档和用户文档);(3) 数据结构。在软件工程过程中产生的所有的信息项(文档、报告、程序、表格、数据)就构成了软件配置。软件配置是软件的具

体形态在某一时刻的瞬时影像。这样的具体形态取两种形式:

- (1) 不可直接执行的材料: 如书写的文档、程序清单、测试数据、测试结果等。
- (2) 可直接执行的材料: 如目标代码、数据库信息等。它们可由计算机处理。

随着软件工程过程的进展, 软件配置项 (software configuration item, SCI) 数目快速增加。系统规格说明可繁衍出软件项目实施计划和软件需求规格说明 (还有与硬件相关的文档)。它们又依次繁衍出建立信息层次的其它文档。如果每个 SCI 只是简单地产生其它 SCI, 造成的混乱可能微乎其微。然而在变更时会引入其它影响因素, 情况就变得复杂起来。SCM 是一组管理整个软件生存期中变更的活动, SCM 可以视为一种质量保证活动, 它应用于整个软件工程过程的所有阶段。下面将介绍主要的 SCM 任务。

1) 基线 (baseline)

基线是软件生存期中各开发阶段末尾的特定点, 又称里程碑。由正式的技术评审而得到的 SCI 协议和软件配置的正式文本才能成为基线。它的作用是把各阶段工作的划分更加明确化, 使本来连续的工作在这些点上断开, 以便于检验和肯定阶段成果。例如明确规定不允许跨越里程碑修改另一阶段的文档。如图 10. 21 所示, 是软件开发各阶段的基线。

以需求规格说明基线为例, 用户对于需求的认识是逐渐深入的。当他的初步需求得到满足后, 常常会提出进一步的需求。如果开发工作已跨过需求分析阶段, 得到各方协议通过的需求规格说明。一旦到达设计阶段或编码阶段, 为了满足需求的变更, 会引起工作的大量反复, 造成软件开发成本的大幅度提高。把需求规格说明设置为基线, 将用户需求“冻结”。如果用户提出了新的需求, 必须在完成本项目的开发后, 作为新项目的需求来考虑。

2) 软件配置项 SCI

图 10. 21 软件开发各阶段的基线

软件配置管理的对象是 SCI——软件配置项, 它们是软件工程过程中产生的信息项。在极端情况下, 可把一个 SCI 看成是一个大规格说明中的一节或一个大测试用例组中的一个测试用例。比较通用的情况是把一个文档、一整个测试用例组、一个有名字的程序部件 (如 PASCAL 过程或 Ada 包) 看成是一个 SCI。以下的 SCI 是 SCM 的对象, 并可形成基线。

- | | |
|-----------|------------------|
| · 系统规格说明 | · 软件项目实施计划 |
| · 软件需求说明 | · 可执行的或“书面”的原型 |
| · 初步的用户手册 | · 设计规格说明 |
| · 源代码清单 | · 测试计划、测试过程、测试用例 |
| · 测试结果记录 | · 操作和安装手册 |
| · 可执行程序 | · 数据库描述 |
| · 正式的用户手册 | · 软件问题报告 |
| · 维护请求 | · 工程变更次序 |
| · 软件工程标准 | · 项目开发总结 |

除了以上所列举的 SCI 以外,许多软件工程组织还把配置控制之下的软件工具列入其中,即编辑程序、编译程序、其它 CASE 工具的特定版本,都要作为软件配置的一部分加以“冻结”。因为要使用这些工具生成文档、程序和数据,因此在对软件配置作变更时它们应当是可利用的。如果编译程序的版本不同,可能产生的结果也不同。由于这个原因,工具也应当成为综合配置管理过程的一部分基线。

在实现 SCM 时,把 SCI 组织成配置对象,在项目数据库中用一个单一的名字组织它们。一个配置对象有一个名字和一组属性,并通过某些联系“连接”到其它对象,如图 10.22 所示。

图中分别对配置对象“设计规格说明”、“数据模型”、“模块 N”、“源代码”和“测试规格说明”进行了定义,每个对象与其它对象的联系用箭头表示。这些箭头指明了一种构造关系。即“数据模型”和“模块 N”是“设计规格说明”的一部分。双向箭头则表明一种相互关系。如果对“源代码”对象作了一个变更,软件工程师就可以根据这种相互关系确定其它哪些对象(和 SCI)可能受到影响。

图 10.22 配置对象

3) 软件配置管理的过程

软件配置管理(SCM)除了担负控制变更的责任之外,它还要担负标识单个的 SCI 和软件各种版本、审查软件配置以保证开发得以正常进行,以及报告所有加在配置上的变更等任务。

有关 SCM,需要考虑这样一些问题:

- 采用什么方式标识和管理许多已存在的程序(和它们的文档)的各种版本?使得变更能够有效地实现。
- 在软件交付用户之前和之后如何控制变更?
- 谁有权批准和对变更安排优先级?
- 如何保证变更得以正确地实施?
- 利用什么办法估计变更可能引起的其它问题?

这些问题归结到 SCM 的 5 个任务,即标识、版本管理、变更控制、配置审计和配置报告。

10.6.2 配置标识

软件配置实际上是一个动态的概念。一方面随着软件生存期的向前推进,SCI 的数量在不断增多。一些文档经过转换生成另一些文档,并产生一些信息。另一方面又随时会有新的变更出现,形成新的版本。因此,整个软件生存期的软件配置就像一部不断演变电影,而某一时刻的配置就是这部电影的一个片段。

为了方便对软件配置的各个片段,即 SCI 进行控制和管理,不致造成混乱,首先应给

它们命名,再利用面向对象的方法组织它们。通常需要标识两种类型的对象:基本对象和复合对象。基本对象是由软件工程师在分析、设计、编码和测试时所建立的“文本单元”。例如,基本对象可能是需求规格说明中的一节,一个模块的源程序清单、一组用来测试一个等价类的测试用例。复合对象则是基本对象或其它复合对象的一个收集。如图 10.22 所示,“设计规格说明”是一个复合对象,它是一些基本对象,如“数据模型”、“模块 N”的收集。每个对象可用一组信息唯一地标识,这组信息包括:

(名字、描述、一组“资源”、“实现”)

对象的名字是一个字符串,它明确地标识对象。对象描述是一个表项,它包括:对象表示的 SCI 类型(如文档、程序、数据)、项目标识、变更和/或版本信息。资源是“由对象提供的、处理的、引用的或其它需要的一些实体”。例如,数据类型、特定函数、甚至变量名都可以看作是对象资源。而“实现”对于一个基本对象来说,是指向“文本单元”的指针,而对于复合对象来说,则为 null(空)。

配置对象的标识还必须考虑在命名对象之间的联系。一个对象可以是一个复合对象的一个组成部分,用联系< part of> 进行标识。这个联系定义了对象的层次。例如,使用记号:

```
E-R diagram 1.4 < part of> data model;  
data model < part of> Design Specification;  
可以建立 SCI 的一个层次。
```

在对象层次中,对象之间的联系不仅存在于层次树的路径中,而且可跨越对象层次的分支相互关联。例如,数据模型与数据流图是相互关联的,而且它又与一个特定等价类的测试用例组相互关联。这些交叉的结构联系可用如下方式表达:

```
data model < interrelated> data flow model;  
data model < interrelated> test case class m;
```

整个软件工程过程中所涉及的软件对象都必须加以标识。在对象成为基线以前可能要作多次变更,在成为基线之后也可能需要频繁地变更。这样对于每一配置对象都可以建立一个演变图,这个演变图记叙了对象的变更历史。以图 10.23 为例,配置对象 1.0 经过

图 10.23 演变图

修改成为对象 1.1,又经历了小的修改和变更,产生了版本 1.1.1 和 1.1.2。紧接着对版本 1.1 作了一次更新,产生对象 1.2,又持续演变生成了 1.3 和 1.4 版本。同时对对象 1.2 作

了一次较大的修改,引出一条新的演变路径:版本 2.0。当前这两种版本都得到支持。

10.6.3 版本控制

版本控制利用工具管理在软件工程过程中建立起来的配置对象的不同版本。Clemm 描述 SCM 环境中的版本控制说:“配置管理允许用户选择适当的版本确定软件系统的配置。这可以通过把一些属性结合到各个软件版本上,再通过描述所希望的属性集合来确定(或构造)所想要的配置”。

上面所说的“属性”可能简单到一个特定的版本号,它被指派给某一特定对象;也可能复杂到一个布尔变量(开关),它指明系统功能变更的特定类型。

表达系统不同版本的一种表示是如图 10.24 所示的演变图。在图中的各个结点都是聚合对象,是一个完全的软件版本。软件的每一版本都是 SCI(源代码、文档、数据)的一个收集,且各个版本都可由不同的变种组成。为了具体说明这一概念,考虑一个简单的程序版本:它由 1, 2, 3, 4 和 5 等部件组成,如图 10.24 所示。其中部件 4 在软件使用彩色显示器时使用,部件 5 在软件使用单色显示器时使用。因此,可以定义版本的两个变种:(1) 部件 1, 2, 3, 4; (2) 部件 1, 2, 3, 5。

图 10.24 版本的变种

10.6.4 变更控制

软件生存期内全部的软件配置是软件产品的真正代表,必须使其保持精确。软件工程过程中某一阶段的变更,均要引起软件配置的变更,这种变更必须严格加以控制和管理,保持修改信息,并把精确、清晰的信息传递到软件工程过程的下一步骤。变更控制包括建立控制点和建立报告与审查制度。

对于一个大型的软件来说,不加控制的变更很快会引起混乱。因此变更控制是一项最重要的软件配置任务。图 10.25 给出了变更控制的过程。

在此过程中,首先用户提交书面的变更请求,详细申明变更的理由、变更方案、变更的影响范围等。然后由变更控制机构确定控制变更的机制、评价其技术价值、潜在的副作用、

图 10.25 变更控制过程

对其它配置对象和系统功能的综合影响以及项目的开销、并把评价的结果以变更报告的形式提交给变更控制负责人（最终决定变更状态和优先权的某个人或小组）。对每个批准了的变更产生一个工程变更顺序（ECO），描述进行的变更、必须考虑的约束、评审和审计的准则等。要作变更的对象从项目数据库中检出（check out），对其作出变更，并实施适当的质量保证活动。然后再把对象登入（check in）到数据库中并使用适当的版本控制机制建立软件的下一版本。

“检出”和“登入”处理实现了两个重要的变更控制要素，即存取控制和同步控制。存取控制管理各个工程师存取或修改一个特定软件配置对象的权限；同步控制可用来确保由不同的人员所执行的并发变更不会产生混乱。

存取和同步控制流如图 10.26 所示。根据经批准的变更请求和 ECO，软件工程师从项目数据库中检出要变更的配置对象。存取控制功能保证了软件工程师有检出该对象的权限，而同步控制功能则封锁（lock）了项目数据库中的这个对象，使得当前检出的版本在没有被置换前不能再更新它。当然，对这个对象还可以检出另外的副本，但对其也不能更新。软件工程师在对这种成为基线的对象作了变更，并经过适当的软件质量保证和测试之后，把修改版本登入项目数据库，再解除封锁（unlock）。

软件的变更通常有两类不同的情况：

1) 为改正小错误需要的变更。它是必须进行的，通常不需要从管理角度对这类变更进行审查和批准。但是，如果发现错误的阶段在造成错误的阶段的后面，例如在实现阶段发现了设计错误，则必须遵照标准的变更控制过程，把这个变更正式记入文档，把所有受

这个变更影响的文档都作相应的修改。

2) 为了增加或者删掉某些功能、或者为了改变完成某个功能的方法而需要的变更。这类变更必须经过某种正式的变更评价过程,以估计变更需要的成本和它对软件系统其它部分的影响。如果变更的代价比较小且对软件系统其它部分没有影响,或影响很小,通常应批准这个变更。反之,如果变更的代价比较高,或者影响比较大,则必须权衡利弊,以决定是否进行这种变更。如果同意这种变更,需要进一步确定由谁来支付变更所需要的费用。如果是用户要求的变更,则用户应支付这笔费用;否则,必须完成某种成本/效益分析,以确定是否值得作这种变更。

图 10. 26 存取和同步控制

应该把所作的变更正式记入文档,并相应地修改所有有关的文档。

这种变更报告和审查制度,对变更控制来说起了一个安全保证作用。在一个 SCI 成为基线之前,可以对所有合理的项目和技术申请进行非正式的变更;一旦某个 SCI 经过正式的技术评审并得到批准,它就成了基线。以后如果需要对它变更,就必须得到项目负责人的批准(限于局部的修改),或者必须得到变更控制负责人的批准(当这个变更影响到其它 SCI 时)。这种变更有时要求有变更申请、变更报告、工程变更顺序等,但必须对每一项变更进行评价并对所有的变更进行跟踪和复审。

10. 6. 5 配置状态报告 (configuration status reporting, CSR)

为了清楚、及时地记载软件配置的变化,不致于到后期造成贻误,需要对开发的过程作出系统的记录,以反映开发活动的历史情况。这就是配置状态登录的任务。

登录主要根据变更控制小组会议的记录,并产生配置状态报告。报告对于每一项变更,记录以下问题:(1) 发生了什么?(2) 为什么会发生?(3) 谁作的?(3) 什么时候发生的?(4) 会有什么影响? 图 10. 27 描述了配置状态报告的信息流。

图 10. 27 配置状态报告

每次新分配一个 SCI 或更新一个已有 SCI 的标识, 或者一项变更申请被变更控制负责人批准, 并给出了一个工程变更顺序时, 在配置状态报告中要增加一条变更记录条目。一旦进行了配置审计, 其结果也应该写入报告之中。配置状态报告可以放在一个联机数据库中, 以便软件开发人员或者软件维护人员可以对它进行查询或修改。此外在软件配置报告中新登录的变更应当及时通知给管理人员和软件工程师。

10. 6. 6 配置审计 (configuration audit)

软件的完整性, 是指开发后期的软件产品能够正确地反映用户提出的对软件的要求。软件配置审计的目的是要证实整个软件生存期中各项产品在技术上和管理上的完整性。同时, 还要确保所有文档的内容变动不超出当初确定的软件要求范围。使得软件配置具有良好的可跟踪性。这是软件变更控制人员掌握配置情况、进行审批的依据。

软件的变更控制机制通常只能跟踪到工程变更顺序产生为止, 那么如何知道变更是否正确完成了呢? 一般可以用以下两种方法审查: (1) 正式技术评审; (2) 软件配置审计。

正式的技术评审着重检查已完成修改的软件配置对象的技术正确性, 评审者评价 SCI, 决定它与其它 SCI 的一致性, 是否有遗漏或可能引起的副作用。正式技术评审应对所有的变更进行, 除了那些最无价值的变更之外。

软件配置审计作为正式技术评审的补充, 评价在评审期间通常没有被考虑的 SCI 的特性。软件配置审查提出并解答以下问题:

- (1) 在工程变更顺序中规定的变更是否已经作了? 每个附加修改是否已经纳入?
- (2) 正式技术评审是否已经评价了技术正确性?
- (3) 是否正确遵循了软件工程标准?
- (4) 在 SCI 中是否强调了变更? 是否说明了变更日期和变更者? 配置对象的属性是否反映了变更?
- (5) 是否遵循了标记变更、记录变更、报告变更的软件配置管理过程?
- (6) 所有相关的 SCI 是否都已正确地作了更新?

在某些情形下, 这些审查问题是作为正式技术评审的一部分提出的。但是当软件配置管理成为一项正式活动时, 软件配置审查就被分开, 而由质量保证小组去执行。

附录 软件产品开发文档编写指南

为使读者具体了解怎样编写文档,这里列出了 10 种文档的内容要求及其简要说明。这些文档包括:可行性研究报告、项目开发计划、需求规格说明书、概要设计说明书、详细设计说明书、用户操作手册、测试计划、测试报告、开发进度月报和项目开发总结报告。各文档内容大纲由带编号的标题构成,标题后方括号内为其说明。

这里给出一个统一的封面格式:

文档编号_____

版本号_____

文档名称 : _____

项目名称 : _____

项目负责人: _____

编写_____

_____年__月__日

校对_____

_____年__月__日

审核_____

_____年__月__日

批准_____

_____年__月__日

开发单位_____

一、可行性研究报告

1. 引言
- 1.1 编写目的

【阐明编写本可行性研究报告的目的,指出读者对象。】
- 1.2 项目背景

【应包括: a. 所建议开发软件的名称; b. 本项目的任务提出者、开发

者、用户及实现软件的单位；c. 本项目与其它软件或其它系统的关系。】

1.3 定义 【列出本文档中用到的专门术语的定义和缩写词的原文。】

1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目经核准的计划任务书、合同或上级机关的批文；b. 与本项目有关的已发表的资料；c. 本文档中所引用的资料，所采用的软件标准或规范。】

2. 可行性研究的前提

2.1 要求 【列出并说明建议开发软件的基本要求，如 a. 功能；b. 性能；c. 输出；d. 输入；e. 基本的数据流程和处理流程；f. 安全与保密要求；g. 与本软件相关的其它系统；h. 完成期限。】

2.2 目标 【可包括：a. 人力与设备费用的节省；b. 处理速度的提高；c. 控制精度或生产能力的提高；d. 管理信息服务的改进；e. 决策系统的改进；f. 人员工作效率的提高等等。】

2.3 条件、假定和限制 【可包括：a. 建议开发软件运行的最短寿命；b. 进行系统方案选择比较的期限；c. 经费来源和使用限制；d. 法律和政策方面的限制；e. 硬件、软件、运行环境和开发环境的条件和限制；f. 可利用的信息和资源；g. 建议开发软件投入使用的最迟时间。】

2.4 可行性研究方法

2.5 决定可行性的主要因素

3. 对现有系统的分析

3.1 处理流程和数据流程

3.2 工作负荷

3.3 费用支出 【如人力、设备、空间、支持性服务、材料等项开支。】

3.4 人员 【列出所需人员的专业技术类别和数量。】

3.5 设备

3.6 局限性 【说明现有系统存在的问题以及为什么需要开发新的系统。】

4. 所建议技术可行性分析

4.1 对系统的简要描述

4.2 处理流程和数据流程

4.3 与现有系统比较的优越性

4.4 采用建议系统可能带来的影响

4.4.1 对设备的影响

4.4.2 对现有软件的影响

4.4.3 对用户的影响

4.4.4 对系统运行的影响

4.4.5 对开发环境的影响

4.4.6 对运行环境的影响

4.4.7 对经费支出的影响

4.5 技术可行性评价 【包括：a. 在限制条件下，功能目标是否能达到；b. 利用现有技

术, 功能目标能否达到; c. 对开发人员数量和质量的要求, 并说明能否满足;
d. 在规定的期限内, 开发能否完成。】

5. 所建议系统经济可行性分析

5.1 支出

5.1.1 基建投资

5.1.2 其它一次性支出

5.1.3 经常性支出

5.2 效益

5.2.1 一次性收益

5.2.2 经常性收益

5.2.3 不可定量收益

5.3 收益/投资比

5.4 投资回收周期

5.5 敏感性分析 【敏感性分析是指一些关键性因素, 如: 系统生存周期长短、系统工作负荷量、处理速度要求、设备和软件配置变化对支出和效益的影响等的分析。】

6. 社会因素可行性分析

6.1 法律因素 【如, 合同责任、侵犯专利权、侵犯版权等问题的分析。】

6.2 用户使用可行性 【如, 用户单位的行政管理、工作制度、人员素质等能否满足要求。】

7. 其它可供选择的方案 【逐个阐明其它可供选择的方案, 并重点说明未被推荐的理由。】

8. 结论意见 【结论意见可能是: a. 可着手组织开发; b. 需待若干条件(如资金、人力、设备等)具备后才能开发; c. 需对开发目标进行某些修改; d. 不能进行或不必要进行(如技术不成熟, 经济上不合算等); e. 其它。】

二、项目开发计划

1. 引言

1.1 编写目的 【阐明编写本开发计划的目的, 指出读者对象。】

1.2 项目背景 【可包括: a. 本项目的委托单位、开发单位和主管部门; b. 该软件系统与其它系统的关系。】

1.3 定义 【列出本文档中用到的专门术语的定义和缩写词的原文。】

1.4 参考资料 【可包括: a. 本项目经核准的计划任务书、合同或上级机关的批文; b. 本文档所引用的资料、规范等; 列出这些资料的作者、标题、编号、发表日期、出版单位或资料来源。】

2. 项目概述

2.1 工作内容 【简要说明本项目的各项主要工作, 介绍所开发软件的功能、性能等。若不编写可行性研究报告, 则应在本节给出较详细的介绍。】

2.2 条件与限制 【阐明为完成本项目应具备的条件、开发单位已具备的条件以及尚需创造的条件。必要时还应说明用户及分合同承包者承担的工作、完成期限及其

它条件与限制。】

2.3 产品

2.3.1 程序 【列出应交付的程序名称、使用的语言及存储形式。】

2.3.2 文档 【列出应交付的文档。】

2.4 运行环境 【应包括硬件环境、软件环境。】

2.5 服务 【阐明开发单位可向用户提供的服务。如人员培训、安装、保修、维护和其它运行支持。】

2.6 验收标准

3. 实施计划

3.1 任务分解 【任务的划分及各项任务的负责人。】

3.2 进度 【按阶段完成的项目,用图表说明开始时间、完成时间。】

3.3 预算

3.4 关键问题 【说明可能影响项目的关键问题,如设备条件、技术难点或其它风险因素,并说明对策。】

4. 人员组织及分工

5. 交付期限

6. 专题计划要点 【如测试计划、质量保证计划、配置管理计划、人员培训计划、系统安装计划等。】

三、需求规格说明书

1. 引言

1.1 编写目的 【阐明编写本需求说明书的目的,指明读者对象。】

1.2 项目背景 【应包括: a. 本项目的委托单位、开发单位和主管部门; b. 该软件系统与其它系统的关系。】

1.3 定义 【列出本文档中所用到的专门术语的定义和缩写词的原文。】

1.4 参考资料 【可包括: a. 本项目经核准的计划任务书、合同或上级机关的批文; b. 项目开发计划; c. 本文档所引用的资料、标准和规范。列出这些资料的作者、标题、编号、发表日期、出版单位或资料来源。】

2. 任务概述

2.1 目标

2.2 运行环境

2.3 条件与限制

3. 数据描述

3.1 静态数据

3.2 动态数据 【包括输入数据和输出数据。】

3.3 数据库描述 【给出使用数据库的名称和类型。】

3.4 数据词典

3.5 数据采集

- 4. 功能需求
 - 4.1 功能划分
 - 4.2 功能描述
- 5. 性能需求
 - 5.1 数据精确度
 - 5.2 时间特性 【如响应时间、更新处理时间、数据转换与传输时间、运行时间等。】
 - 5.3 适应性 【在操作方式、运行环境、与其它软件的接口以及开发计划等发生变化时, 应具有适应能力。】
- 6. 运行需求
 - 6.1 用户界面 【如屏幕格式、报表格式、菜单格式、输入输出时间等。】
 - 6.2 硬件接口
 - 6.3 软件接口
 - 6.4 故障处理
- 7. 其它需求 【如可使用性、安全保密、可维护性、可移植性等。】

四、概要设计说明书

- 1. 引言
 - 1.1 编写目的 【阐明编写本概要设计说明书的目的, 指明读者对象。】
 - 1.2 项目背景 【应包括: a. 本项目的委托单位、开发单位和主管部门; b. 该软件系统与其它系统的关系。】
 - 1.3 定义 【列出本文档中所用到的专门术语的定义和缩写词的原意。】
 - 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源, 可包括: a. 本项目经核准的计划任务书、合同或上级机关的批文; b. 项目开发计划; c. 需求规格说明书; d. 测试计划(初稿); e. 用户操作手册(初稿); f. 本文档所引用的资料、采用的标准或规范。】
- 2. 任务概述
 - 2.1 目标
 - 2.2 运行环境
 - 2.3 需求概述
 - 2.4 条件与限制
- 3. 总体设计
 - 3.1 处理流程
 - 3.2 总体结构和模块外部设计
 - 3.3 功能分配 【表明各项功能与程序结构的关系。】
- 4. 接口设计
 - 4.1 外部接口 【包括用户界面、软件接口与硬件接口。】
 - 4.2 内部接口 【模块之间的接口。】
- 5. 数据结构设计

- 5.1 逻辑结构设计
- 5.2 物理结构设计
- 5.3 数据结构与程序的关系
- 6. 运行设计
 - 6.1 运行模块的组合
 - 6.2 运行控制
 - 6.3 运行时间
- 7. 出错处理设计
 - 7.1 出错输出信息
 - 7.2 出错处理对策 【如设置后备、性能降级、恢复及再启动等。】
- 8. 安全保密设计
- 9. 维护设计 【说明为方便维护工作的设施, 如维护模块等。】

五、详细设计说明书

- 1. 引言
 - 1.1 编写目的 【阐明编写本详细设计说明书的目的, 指明读者对象。】
 - 1.2 项目背景 【应包括项目的来源和主管部门等。】
 - 1.3 定义 【列出本文档中所用到的专门术语的定义和缩写词的原意。】
 - 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源, 可包括: a. 本项目的计划任务书、合同或批文; b. 项目开发计划; c. 需求规格说明书; d. 概要设计说明书; e. 测试计划(初稿); f. 用户操作手册(初稿); g. 本文档中所引用的其它资料、软件开发标准或规范。】
- 2. 总体设计
 - 2.1 需求概述
 - 2.2 软件结构 【如给出软件系统的结构图。】
- 3. 程序描述 【逐个模块给出以下的说明: 】
 - 3.1 功能
 - 3.2 性能
 - 3.3 输入项目
 - 3.4 输出项目
 - 3.5 算法 【本模块所选用的算法。】
 - 3.6 程序逻辑 【详细描述本模块实现的算法, 可采用: a. 标准流程图; b. PDL 语言; c. N-S 图; d. PAD; e. 判定表等描述算法的图表。】
 - 3.7 接口
 - 3.8 存储分配
 - 3.9 限制条件
 - 3.10 测试要点 【给出测试本模块的主要测试要求。】

六、用户操作手册

1. 引言

- 1.1 编写目的 【阐明编写本手册的目的,指明读者对象。】
- 1.2 项目背景 【说明项目来源、委托单位、开发单位及主管部门。】
- 1.3 定义 【列出本手册中使用的专门术语的定义和缩写词的原意。】
- 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源,可包括: a. 本项目的计划任务书、合同或批文; b. 项目开发计划; c. 需求规格说明书; d. 概要设计说明书; e. 详细设计说明书; f. 测试计划; g. 本手册中引用的其它资料、采用的软件工程标准或软件工程规范。】

2. 软件概述

- 2.1 目标
- 2.2 功能
- 2.3 性能
 - a. 数据精确度 【包括输入、输出及处理数据的精度。】
 - b. 时间特性 【如响应时间、处理时间、数据传输时间等。】
 - c. 灵活性 【在操作方式、运行环境需作某些变更时本软件的适应能力。】

3. 运行环境

- 3.1 硬件 【列出本软件系统运行时所需的硬件最小配置,如 a. 计算机型号、主存容量; b. 外存储器、媒体、记录格式、设备型号及数量; c. 输入、输出设备; d. 数据传输设备及数据转换设备的型号及数量。】
- 3.2 支持软件 【如: a. 操作系统名称及版本号; b. 语言编译系统或汇编系统的名称及版本号; c. 数据库管理系统的名称及版本号; d. 其它必要的支持软件。】

4. 使用说明

- 4.1 安装和初始化 【给出程序的存储形式、操作命令、反馈信息及其含意、表明安装完成的测试实例以及安装所需的软件工具等。】
- 4.2 输入 【给出输入数据或参数的要求。】
 - 4.2.1 数据背景 【说明数据来源、存储媒体、出现频度、限制和质量管理等。】
 - 4.2.2 数据格式 【如: a. 长度; b. 格式基准; c. 标号; d. 顺序; e. 分隔符; f. 词汇表; g. 省略和重复; h. 控制。】
 - 4.2.3 输入举例
- 4.3 输出 【给出每项输出数据的说明。】
 - 4.3.1 数据背景 【说明输出数据的去向、使用频度、存放媒体及质量管理等。】
 - 4.3.2 数据格式 【详细阐明每一输出数据的格式,如: 首部、主体和尾部的具体形式。】
 - 4.3.3 举例
- 4.4 出错和恢复 【给出: a. 出错信息及其含意; b. 用户应采取的措施,如修改、恢复、再启动。】

- 4.5 求助查询 【说明如何操作。】
- 5. 运行说明
 - 5.1 运行表 【列出每种可能的运行情况,说明其运行目的。】
 - 5.2 运行步骤 【按顺序说明每种运行的步骤,应包括:】
 - 5.2.1 运行控制
 - 5.2.2 操作信息
 - a. 运行目的; b. 操作要求; c. 启动方法; d. 预计运行时间; e. 操作命令格式及说明; f. 其它事项。
 - 5.2.3 输入/输出文件 【给出建立或更新文件的有关信息,如:】
 - a. 文件的名称及编号; b. 记录媒体; c. 存留的目录; d. 文件的支配 【说明确定保留文件或废弃文件的准则,分发文件的对象,占用硬件的优先级及保密控制等。】
 - 5.2.4 启动或恢复过程
- 6. 非常规过程 【提供应急或非常规操作的必要信息及操作步骤,如出错处理操作、向后备系统切换操作以及维护人员须知的操作和注意事项。】
- 7. 操作命令一览表 【按字母顺序逐个列出全部操作命令的格式、功能及参数说明。】
- 8. 程序文件(或命令文件)和数据文件一览表 【按文件名字母顺序或按功能与模块分类顺序逐个列出文件名称、标识符及说明。】
- 9. 用户操作举例

七、测试计划

- 1. 引言
 - 1.1 编写目的 【阐明编写本测试计划的目的并指明读者对象。】
 - 1.2 项目背景 【说明项目的来源、委托单位及主管部门。】
 - 1.3 定义 【列出本测试计划中所用到的专门术语的定义和缩写词的原意。】
 - 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源,可包括: a. 本项目的计划任务书、合同或批文; b. 项目开发计划; c. 需求规格说明书; d. 概要设计说明书; e. 详细设计说明书; f. 用户操作手册; g. 本测试计划中引用的其它资料、采用的软件开发标准或规范。】
- 2. 任务概述
 - 2.1 目标
 - 2.2 运行环境
 - 2.3 需求概述
 - 2.4 条件与限制
- 3. 计划
 - 3.1 测试方案 【说明确定测试方法和选取测试用例的原则。】
 - 3.2 测试项目 【列出组装测试和确认测试中每一项测试的内容、名称、目的和进度。】
 - 3.3 测试准备

- 3.4 测试机构及人员 【测试机构名称、负责人和职责。】
- 4. 测试项目说明 【按顺序逐个对测试项目作出说明：】
 - 4.1 测试项目名称及测试内容
 - 4.2 测试用例
 - 4.2.1 输入 【输入的数据和输入命令。】
 - 4.2.2 输出 【预期的输出数据。】
 - 4.2.3 步骤及操作
 - 4.2.4 允许偏差 【给出实测结果与预期结果之间允许偏差的范围。】
 - 4.3 进度
 - 4.4 条件 【给出本项测试对资源的特殊要求, 如设备、软件、人员等。】
 - 4.5 测试资料 【说明本项测试所需的资料。】
- 5. 评价
 - 5.1 范围 【说明所完成的各项测试说明问题的范围及其局限性。】
 - 5.2 准则 【说明评价测试结果的准则。】

八、测试分析报告

- 1. 引言
 - 1.1 编写目的 【阐明编写本测试分析报告的目的并指明读者对象。】
 - 1.2 项目背景 【说明项目的来源、委托单位及主管部门。】
 - 1.3 定义 【列出本测试分析报告中用到的专门术语的定义和缩写词的原意。】
 - 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源, 可包括: a. 本项目的计划任务书、合同或批文; b. 项目开发计划; c. 需求规格说明书; d. 概要设计说明书; e. 详细设计说明书; f. 用户操作手册; g. 测试计划; h. 本测试分析报告所引用的其它资料、采用的软件工程标准或软件工程规范。】
- 2. 测试计划执行情况
 - 2.1 测试项目 【列出每一测试项目的名称、内容和目的。】
 - 2.2 测试机构和人员 【给出测试机构名称、负责人和参与测试人员名单。】
 - 2.3 测试结果 【按顺序给出每一测试项目的: a. 实测结果数据; b. 与预期结果数据的偏差; c. 该项测试表明的事实; d. 该项测试发现的问题。】
 - 2.4 软件需求测试结论 【按顺序给出每一项需求测试的结论。包括: a. 证实的软件能力; b. 局限性(即本项需求未得到充分测试的情况及原因)。】
- 3. 评价
 - 3.1 软件能力 【经过测试所表明的软件能力。】
 - 3.2 缺陷和限制 【说明测试所揭露的软件缺陷和不足, 以及可能给软件运行带来的影响。】
 - 3.3 建议 【提出为弥补上述缺陷的建议。】
 - 3.4 测试结论 【说明能否通过。】

九、开发进度月报

1. 报告时间及所处的开发阶段
2. 工程进度
 - 2.1 本月内的主要活动
 - 2.2 实际进展与计划比较
3. 所用工时 【按不同层次人员分别计时。】
4. 所用机时 【按所用计算机机型分别计时。】
5. 经费支出 【分类列出本月经费支出项目, 给出支出总额, 并与计划比较。】
6. 工作遇到的问题及采取的对策
7. 本月完成的成果
8. 下月的工作计划
9. 特殊问题

十、项目开发总结报告

1. 引言
 - 1.1 编写目的 【阐明编写本总结报告的目的并指明读者对象。】
 - 1.2 项目背景 【说明项目来源、委托单位、开发单位及主管部门。】
 - 1.3 定义 【列出本报告用到的专门术语的定义和缩写词的原意。】
 - 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源, 可包括: a. 本项目经核准的计划任务书、合同或上级机关的批文; b. 项目开发计划; c. 需求规格说明书; d. 概要设计说明书; e. 详细设计说明书; f. 用户操作手册; g. 测试计划; h. 测试分析报告; i. 本报告引用的其它资料、采用的开发标准或开发规范。】
2. 开发结果
 - 2.1 产品 【可包括: a. 列出各部分的程序名称、源程序行数(包括注释行)或目标程序字节数及程序总计数量、存储形式; b. 产品文档名称等。】
 - 2.2 主要功能及性能
 - 2.3 所用工时 【按人员的不同层次分别计时。】
 - 2.4 所用机时 【按所用计算机机型分别计时。】
 - 2.5 进度 【给出计划进度与实际进度的对比。】
 - 2.6 费用
3. 评价
 - 3.1 生产率评价 【如平均每人每月生产的源程序行数、文档的字数等。】
 - 3.2 技术方案评价
 - 3.3 产品质量评价
4. 经验与教训

十一、程序维护手册

1. 引言

- 1.1 编写目的 【阐明编写本手册的目的并指明读者对象。】
- 1.2 开发单位 【说明项目的提出者、开发者、用户和使用场所。】
- 1.3 定义 【列出本报告用到的专门术语的定义和缩写词的原意。】
- 1.4 参考资料 【列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源,以及保密级别,可包括: a. 用户操作手册; b. 与本项目有关的其它文档。】

2. 系统说明

- 2.1 系统用途 【说明系统具备的功能,输入和输出。】
- 2.2 安全保密 【说明系统安全保密方面的考虑。】
- 2.3 总体说明 【说明系统的总体功能,对系统、子系统和作业作出综合性的介绍,并用图表的方式给出系统主要部分的内部关系。】
- 2.4 程序说明 【说明系统中每一程序、分程序的细节和特性。】

2.4.1 程序 1 的说明

- 2.4.1.1 功能 【说明程序的功能。】
- 2.4.1.2 方法 【说明实现方法。】
- 2.4.1.3 输入 【说明程序的输入、媒体、运行数据记录、运行开始时使用的输入数据的类型和存放单元、与程序初始化有关的入口要求。】
- 2.4.1.4 处理 【处理特点和目的,如: a. 用图表说明程序的运行的逻辑流程; b. 程序主要主要转移条件; c. 对程序的约束条件; d. 程序结束时的出口要求; e. 与下一个程序的通信与联结(运行、控制); f. 由该程序产生并提供处理程序段使用的输出数据类型和存放单元。g. 程序运行所用存储量、类型及存储位置等。】
- 2.4.1.5 输出 【程序的输出。】
- 2.4.1.6 接口 【本程序与本系统其它部分的接口。】
- 2.4.1.7 表格 【说明程序内部的各种表、项的细节和特性。对每张表的说明至少包括: a. 表的标识符; b. 使用目的; c. 使用此表的其它程序; d. 逻辑划分,如块或部,不包括表项; e. 表的基本结构; f. 设计安排,包括表的控制信息。表目结构细节、使用中的特有性质及各表项的标识、位置、用途、类型、编码表示。】
- 2.4.1.8 特有的运行性质 【说明在用户操作手册中没有提到的运行性质。】

2.4.2 程序 2 的说明 【与程序 1 的说明相同。以后其它各程序的说明相同。】

3. 操作环境

- 3.1 设备 【逐项说明系统的设备配置及其特性。】

- 3.2 支持软件 【列出系统使用的支持软件,包括它们的名称和版本号。】
- 3.3 数据库 【说明每个数据库的性质和内容,包括安全考虑。】
 - 3.3.1 总体特征 【如: a. 标识符; b. 使用这些数据库的程序; c. 静态数据; d. 动态数据; e. 数据库的存储媒体; f. 程序使用数据库的限制。】
 - 3.3.2 结构及详细说明
 - 3.3.2.1 说明该数据库的结构,包括其中的记录和项;
 - 3.3.2.2 说明记录的组成,包括首部或控制段、记录体;
 - 3.3.2.3 说明每个记录结构的字段,包括: 标记或标号、字段的字符长度和位数、该字段的允许值范围。
 - 3.3.2.4 扩充: 说明为记录追加字段的规定;
- 4. 维护过程
 - 4.1 约定 【列出该软件系统设计中使用的全部规则和约定,包括: a. 程序、分程序、记录、字段和存储区的标识或标号助记符的使用规则; b. 图表的处理标准、卡片的连接顺序、语句和记号中使用的缩写、出现在图表中的符号名; c. 使用的软件技术标准; d. 标准化的数据元素及其特征。】
 - 4.2 验证过程 【说明一个程序段修改后,对其进行验证的要求和过程(包括测试程序和数据)及程序周期性验证的过程。】
 - 4.3 出错及纠正方法 【列出出错状态及其纠正方法。】
 - 4.4 专门维护过程 【说明本文档其它地方没有提到的专门维护过程,如: a. 维护该软件的输入输出部分(如数据库)的要求、过程和验证方法; b. 运行程序库维护系统所必需的要求、过程和验证方法; c. 对闰年、世纪变更所需要的临时性修改等。】
 - 4.5 专用维护程序 【列出维护软件系统使用的后备技术和专用程序(如文件恢复程序、淘汰过时文件的程序等)的目录,并加以说明,内容包括: a. 维护作业的输入输出要求; b. 输入的详细过程及在硬设备上建立、运行并完成维护作业的操作步骤。】
 - 4.6 程序清单和流程图 【引用资料或提供附录给出程序清单和流程图。】

十二、软件问题报告

- 1. 登记号 【由软件配置管理部门为该报告规定一个唯一的、顺序的编号。】
- 2. 登记日期 【软件配置管理部门登记该报告的日期。】
- 3. 问题发现日期 【发现该问题的日期和时间。】
- 4. 活动 【在哪个阶段发现的问题,分为单元测试、组装测试、确认测试和运行维护。】
- 5. 状态 【在软件配置记录中维护的动态指示,状态表示有: a. 正在复查“软件问题报告”,以确定将采取什么行动; b. “软件问题报告”已由指定的人进行处理; c. 修改已完成,并经过测试,正准备交给主程序库; d. 主程序库已经更新,主程序库修改的重新测试尚未完成; e. 作了重新测试,问题再现; f. 作了重新测试,所作的修改无故障,“软件问题报告”被关闭; g. 留待以后关闭。】

6. 报告人 【填写“软件问题报告”人员的姓名、地址、电话。】
7. 问题属于什么方面 【区分是程序的问题, 还是模块的问题, 或是数据库的问题, 文件的问题。也可能是它们的某种组合。】
8. 模块/子系统 【出现的模块名。如果不知是哪个模块, 可标出子系统名, 尽量给出细节。】
9. 修订版本号 【出现问题的模块版本。】
10. 磁带 【包含有问题的模块的主程序库的磁带的标识符。】
11. 数据库 【当发现问题时所使用数据库的标识符。】
12. 文件号 【有错误的文件的编号。】
13. 测试用例 【发现错误时所使用测试用例的标识符。】
14. 硬件 【发现错误时所使用的计算机系统的标识。】
15. 问题描述/影响 【问题征兆的详细描述。如果可能, 则写明实际问题所在。也要给出该问题对将来测试、接口软件和文件等的影响。】
16. 附注 【记载补充信息。】

十三、软件修改报告

1. 登记号 【由软件配置管理部门为该报告规定的编号。】
2. 登记日期 【软件配置管理部门登记“软件修改报告”的日期。】
3. 时间 【准备好“软件修改报告”的日期。】
4. 报告人 【填写该报告的作者。】
5. 子系统名 【受修改影响的子系统名。】
6. 模块名 【被修改的模块名。】
7. “软件问题报告”的编号 【被“软件修改报告”处理或部分处理的“软件问题报告”的编号。如果某“软件问题报告”的问题只是部分被处理, 则在编号后附以 p, 如 1234p。】
8. 修改 【包括程序修改、文件更新、数据库修改或它们的组合。】
9. 修改描述 【修改的详细描述。如果是文件更新或数据库修改, 还要列出文件更新通知或数据库修改申请的标识符。】
10. 批准人 【批准人签字, 正式批准进行修改。】
11. 语句类型 【程序修改中涉及到的语句类型, 包括: 输入/输出语句类、计算语句类、逻辑控制语句类、数据处理语句类(如数据传送、存取语句类)。】
12. 程序名 【被修改的程序、文件或数据库的名字。】
13. 老修订版 【当前的版本/修订本标识。】
14. 新修订版 【修改后的版本/修订本标识。】
15. 数据库 【如果申请数据库修改, 则给出数据库的标识符。】
16. 数据库修改报告 【数据库修改申请号。】
17. 文件 【如果要求对文件进行修改, 则给出文件的名称。】
18. 文件更新 【文件更新通知单的编号。】
19. 修改是否已测试 【指出已对修改作了哪些测试, 如单元、子系统、组装、确认和运行测试。

试等,并注明测试成功与否。】

20. “ 软件问题报告 ”是否给出问题的准确描述 【回答‘ 是 ’或‘ 否 ’】

21. 问题注释 【准确地叙述要维护的问题。】

22. 问题源 【指明问题来自哪里,如软件需求说明书、设计说明书、数据库、源程序等。】

23. 资源 【完成修改所需资源的估计,即总的人时数和计算机时间的开销。】

参 考 文 献

- [1] Glenford J Myers. Software Reliability Principles and Practices. New York: John Wiley & Sons, Inc., 1976
- [2] Barry W Boehm. Software Engineering Economics. New York: Pretice-Hall, Inc., 1981
- [3] James Martin and Carma McClure. Software Maimtenance The Problem and Its Solutions. New York: Pretice-Hall, Inc., 1983
- [4] Glenford J Myers. The Art of Software Testing. New York: John Wiley & Sons, Inc., 1979, 中译本. 周之英, 郑人杰译. 北京: 清华大学出版社, 1985
- [5] 李友仁. 软件工程与软件质量分析. 北京: 电子工业出版社, 1987
- [6] James Vincent & Albert Waters & John Sinclair. Software Quality Assurance: Volume I, Practice and Implementation. New York: Prentice-Hall, Hall., 1988
- [7] James Vincent & Albert Waters & John Sinclair. Software Quality Assurance: Volume II, A Program Guide. New York: Prentice-Hall, Hall., 1988
- [8] 林琪超. 软件开发环境. 上海: 交通大学出版社, 1991
- [9] J. Rumbaugh et al.. Object-Oriented Modeling and Design. New York: Prentice-Hall, 1991
- [10] Peter Coad and Edward Yourdon. Object-Oriented Analysis. Second Edition, New York: Prentice-Hall, 1991
- [11] Peter Coad and Edward Yourdon. Object-Oriented Design. New York: Prentice-Hall, 1991
- [12] Roger S Pressman. Software Engineering: A Practitioner's Approach. 3rd Edition. New York: McGraw-Hill, Inc., 1992
- [13] John D McGregor & David A. Sykes. Object-Oriented Software Development: Engineering Software for Reuse. New York: Van Nostrand Reinhold, 1992
- [14] 郑人杰. 计算机软件测试技术. 北京: 清华大学出版社, 1992
- [15] 何方汉等. 美国 ISO 9000 的实施与审核. 福建省质量管理与质量保证标准化技术委员会. 福州: 1993
- [16] 郑人杰, 彭春龙. 计算机辅助软件工程-CASE 技术. 北京: 清华大学出版社, 1994
- [17] 戚奎桐, 王纬. 软件质量管理和质量保证: GB/T 19000- 3 (ISO 9000- 3) 国家标准应用指南. 北京: 中国标准出版社, 1995
- [18] Carnegie Mellon University. Software Engineering Institute. The Capability Maturity Model - Guidelines for Improving the Software Process. New York: Addison - Wesley Publishing Company, 1995
- [19] Edward Yourdon and Carl Argila. Case Studies in Object-Oriented Analysis & Design. New York: Prentice-Hall, 1996
- [20] Ronald J. Norman. Object-Oriented Systems Analysis and Design. New York: Prentice-Hall, Inc., 1996
- [21] 郑人杰, 殷人昆, 陶永雷. 实用软件工程, 第二版. 北京: 清华大学出版社, 1997