高等专科学校教材

软件系统开发技术

潘 锦 平

西安电子科技大学出版社

高等学校教材

软件系统开发技术

潘 锦 平

西安电子科技大学出版社

1993

(陕)新登字 010 号

内 容 简 介

本书介绍大型软件系统的开发技术，主要是目前软件界最为流行，也较实用的结构化方法，包括支持这一方法的工具和环境；还简述了其他一些方法和技术，如 Jackson 方法、Parnas 方法、测试和维护技术、以及数据库设计等，使材料更系统化，并有所比较.

本书用作高等院校计算机或非计算机专业本科高年级的教材，也可作软件实际工作者的参考书.

高等学校教材

软件系统开发技术

潘锦平

责任编辑 谭玉瓦

西安电子科技大学出版社出版

陕西省军区长城印刷厂印刷

陕西省新华书店发行 各地新华书店经售

开本 787×1092 1/16 印张 14 4/16 字数 330 千字

1989 年 6 月第 1 版 1993 年 4 月第 5 次印刷 印数 18 001—23 000

ISBN 7-5606-0068-9/TP·0024(课) 定价: 3.75 元

出 版 说 明

根据国务院关于高等学校教材工作分工的规定，我部承担了全国高等学校、中等专业学校工科电子类专业教材的编审、出版的组织工作。由于各有关院校及参与编审工作的广大教师共同努力，有关出版社的紧密配合，从1978年至1985年，已编审、出版了两轮教材，正在陆续供给高等学校和中等专业学校教学使用。

为了使工科电子类专业教材能更好地适应"三个面向"的需要，贯彻"努力提高教材质量，逐步实现教材多样化，增加不同品种、不同层次，不同学术观点、不同风格、不同改革试验的教材"的精神，我部所属的七个高等学校教材编审委员会和两个中等专业学校教材编审委员会，在总结前两轮教材工作的基础上，结合教育形势的发展和教学改革的需要，制订了1986～1990年的"七五"(第三轮)教材编审出版规划。列入规划的教材、实验教材、教学参考书等近400种选题。这批教材的评选推荐和编写工作由各编委会直接组织进行。

这批教材的书稿，是从通过教学实践、师生反映较好的讲义中经院校推荐，由编审委员会(小组)评选择优产生出来的。广大编审者、各编审委员会和有关出版社为保证教材的出版和提高教材的质量，作出了不懈的努力。

限于水平和经验，这批教材的编审、出版工作还会有缺点和不足之处，希望使用教材的单位，广大教师和同学积极提出批评建议，共同为不断提高工科电子类专业教材的质量而努力。

电子工业部教材办公室

前　言

本教材系按电子工业部制定的工科电子类专业教材 1986～1990 年编审出版规划，由计算机与自动控制教材编审委员会计算机教材编审小组组织征稿、评选、推荐出版的。

本教材由上海交通大学潘锦平编写，哈尔滨工业大学李莲治担任主审。

本课程的参考学时数为 60 学时，其主要内容为大型软件系统的开发技术，特别是目前在软件产业界最为流行、且较为实用的结构化方法(结构化分析、结构化设计和结构化程序设计)，包括支持这一方法的工具和环境；为使材料系统化，并有所比较，也简述了其他开发技术，如 Jaskson 方法、Parnas 方法和一些测试、维护、数据库设计方法等。

使用本教材时应注意以下几点:

1.学生事先应修完程序设计、数据结构、操作系统、数据库原理等课程，并且最好有一定的软件实践经验。

2.阅读教材中的实例和在实际课题中试用软件方法是掌握这些技术的重要途径，所以对各章的实例应该认真讨论；如果还能组织学生自始至终开发一个规模适度的系统，可望获得较好的教学效果。

3.由于方法同工具有密切的联系，所以第八章的内容可以分别穿插在前面各章讲授，如果学时较少，也可将第八章略去。

由于编者水平有限，书中难免还存在一些缺点和错误，殷切希望广大读者批评指正。

编　者

1987.8

目　录

第一章 绪 言

1.1 软件工程学的背景和目的

计算机专业的学生在修完编程(Programming)等课程之后，对编写小型程序，例如字符编辑程序或报表打印程序等，一定是很有把握了．但是，如果需要研制一个大型的软件系统，例如飞机订票系统或学校管理系统(包括教务、财务、人事、物资等各系科的全面管理)，相信会遇到许多困难，因此还必须学习"软件工程学"．

"软件工程"(Software Engineering)是从"编程"演变过来的，后者一般考虑小型程序的编写，前者则考虑大型软件系统的研制．"软件工程学"出现至今只有二十年，是一门新的学科．本节先讨论软件工程学产生的背景及这门学科的目的．

60年代以来，在一些技术领先的国家中，计算机的应用领域越来越广，它几乎涉及到社会生活的各个方面，如工厂管理、银行事务、病人监护、学校档案、图书馆流通、旅馆预订……，这些系统的软件规模都相当大，逻辑很复杂，而且功能上需要不断更改和扩充．至于军事方面的导弹防御系统、宇航方面的飞行控制系统，其软件就更为庞大和复杂了．

由于软件是非实物性、不可见的，开发软件本质上又是一个"思考"的过程，很难进行管理，开发人员可以按各自的爱好和习惯进行工作，没有统一的标准可以遵循，只能以手工艺的方式进行．管理人员事前很难估计项目所需的经费和时间，技术人员在项目完成之前也难以预料系统是否能成功．

人们在开发上述大型软件系统时遇到了许多困难，有的系统最终彻底失败了；有的系统虽然完成了，但比原定计划迟了好几年，而且经费上大大超过了预算；有的系统未能完满地符合用户当初的期望；有的系统则无法进行修改维护．更糟的是，失败的系统往往无可挽回，除非再从头做起，但由于时间和经费的限制，这又是不可能的．

IBM公司的OS／360系统和美国空军某后勤系统在开发过程中遭受的挫折是众所周知的，以OS／360为例：它由4000个模块组成，共约100万条指令，人工为5000人年(一个人工作一年其工作量相当于一个人年)，经费达数亿美元，但结果却是令人沮丧的，人们在程序中发现的错误达2000个以上．

OS／360系统的负责人Brooks曾生动地描述了开发过程中的困难和混乱：

> "……像巨兽在泥潭中作垂死挣扎，挣扎得越猛，泥浆就沾得越多，最后没有一个野兽能逃脱淹没在泥潭中的命运……程序设计就像是这样一个泥潭……一批批程序员在泥潭中挣扎……没人料到问题竟会这样棘手……"[1]．

人们发现，研制软件系统需要投入大量的人力和物力，但系统的质量却难以保证，也就是说，开发软件所需的高成本同产品的低质量之间有着尖锐的矛盾，这种现象就是所谓的"软件危机"(Software Crisis)．

与此同时，由于新的电子元器件的出现，计算机硬件的功能和质量在不断地提高，而

价格却在大幅度地下降，因此同硬件投资相比，软件投资比例上升极快，图 1.1 是 Boehm 在 1976 年对美国计算机总投资的统计和预测，从图中可以看出，在 50 年代，软件投资约占 20%，至 70 年代已超过 60%，当时预测到 1985 年软件投资将高达 85%。

硬件价格的大幅度下降意味着计算机可以广泛使用，因此对软件的需求量将会迅速上升，但是软件危机的事实告诉人们：软件技术没有跟上硬件技术的高速发展，人们意识到计算机要推广使用，其瓶颈在于软件开发技术的革新。

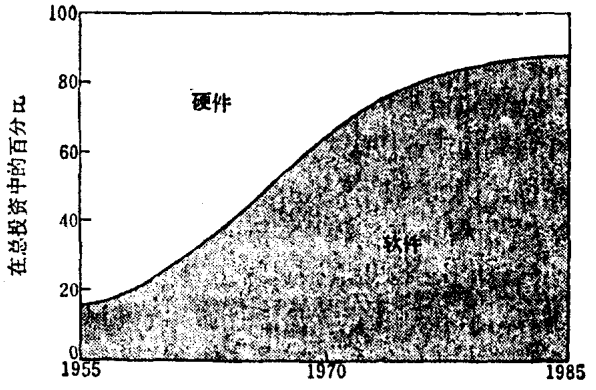


图 1.1

Parnas 认真分析了开发大型软件和编制小型程序之间的差别，他发现，从所需人力来看，小型程序从确定要求、编制、使用、直至修改往往是由同一个人完成的，因此只要他本人心里明白程序的构思就够了，而大型系统则必须由许多人(包括用户、项目负责人、分析员、高初级程序员、资料员、操作员等)组成一支开发队伍来协同完成，所以人与人之间必须准确地进行协商讨论；另外，从产品使用情况来看，小型程序往往是"一次性"的，意即如果需要作较大的修改，人们通常宁可丢弃旧的程序而重新编写，但大型系统的开发耗费了大量的人力与物力，所以人们一般不会轻易将其抛弃，而总是在旧程序的基础上一改再改，希望延长它的使用期，因而是"多个版本"的。

所以，Parnas 将大型软件开发的特点总结为：由"多个人"来开发具有"多个版本"的程序。

大型软件系统的开发提出了许多新的问题，诸如：如何将一个系统分解成若干个部分，以便各人分工开发；如何精确地说明每个部分的规格要求；怎样才能使软件产品易于修改维护；……。

显然，传统的"编程"没有考虑这些问题。

量变带来了质变，系统规模的增大使问题的性质发生了变化，人们认识到：正像不能用造独木船的手工艺方式来研制航空母舰一样，沿用 50 年代个人编写小型程序的那种手工艺方式来开发大型软件系统也是不行的，必须寻找新的技术来指导软件的大规模生产。

考虑到机械、建筑等领域都经历过从手工艺方式演变成严密完整的工程科学的过程，一些有识之士认为大型软件系统的开发也应该向"工程化"方向发展，逐步发展成一门严格的工程科学。

1968 年在北大西洋公约组织的学术会议上有人第一次提出了"软件工程"这个词，还提出了一些软件工程化的技术并进行了讨论。

60 年代末至 70 年代初，"软件工程学"还处于学术研究阶段，但已对软件开发的实践产生了巨大的影响。1971 年 IBM 公司运用一些软件工程技术成功地研制了纽约时报情报库系统和空间实验室的飞行模拟系统，这也是两个著名的例子，尽管两个系统都很庞大，用户要求又有很多变化，并又减少了人力和削减了经费，但由于适当地采用了工程化的技

术，还是按时、高质量地完成了，软件生产率比以前提高了一倍。

这些事实说明用"工程化"的思想作指导，可以大大减少软件开发成本并提高软件质量。"工程化"为人们开辟了新的道路。"软件工程学"这门富有潜力的学科就此蓬勃地发展起来了。

在我国，计算机领域近年来迅猛发展，前面讨论的种种问题和矛盾也就在国内暴露出来了。国内的软件工作者也意识到要进一步发展我国的计算机事业，软件工程学是个关键。目前国内、国外都有许多人在从事这一领域的研究，软件工程学已成为计算机学科中的一个重要分支。

作为一门学科，软件工程学研究的是：如何应用一些科学理论和工程上的技术来指导大型①软件系统的开发，使其发展成一门严格的工程科学；软件工程学的最终目的是以较低的成本研制具有较高质量的软件。所以，研究软件工程学无疑将促进计算机推广应用的步伐，直接或间接地产生巨大的社会效益和经济效益。

表 1.1

规模分类	开发人员数	周 期	源程序行数	实 例
超小型	1	1—4 周	500	简单程序
小 型	1	1—6 月	1—2K	数值程序
中 型	2—5	1—2 年	5—50K	汇编程序
大 型	5—20	2—3 年	50—100K	数据库
超大型	100—1000	4—5 年	1M	操作系统
极大型	2000—5000	5—10 年	1—10M	空中交通控制

软件工程学包括的面很广，有基础理论研究，也有应用研究以及实际开发；除了技术问题之外，它还涉及与开发软件有关的所有活动，例如管理学、心理学、经济学法律、道德等方面。本书只讨论软件工程中的技术问题，重点是软件产业界近年来较流行的实用技术。

1.2 软件和软件生命期模型

计算机领域从 50 年代到 80 年代有了突飞猛进的发展，人们对许多问题的看法也发生了根本的变化。所以，在学习具体的软件开发技术之前，我们有必要对软件的一些基本概念，如什么是软件、软件开发过程包括哪些活动、如何评价软件产品的质量等，重新进行讨论。读者将会看到目前的看法同传统的观点已有了相当大的差别。

软件和软件生命期模型是软件工程学中两个重要的概念。过去，人们一般认为所谓"软件"就是指"程序"，所谓"开发软件"也仅仅就是"编程序"而已。但是，对于大型软件系统而言，这样的理解是不合适的。

---

① IBM公司按源程序行数(Lines of Code)为软件规模分类，通常小于2K行作为超小型，2K至16K 行为小型，16K 至 64K 行为中型，64K 至 512K 行为大型，512K 行以上为超大型。
Fairley 主张参考开发人员数、开发周期及源程序行数等诸因素为软件规模分类[6]，见表 1.1。

在机械工程中，一台机器的生命期(即从开始研制至机器最终被废弃不再使用为止)要经过分析要求、设计、制造、测试、运行(此时需要不断地维护)等几个阶段．伴随着机器的设计制造，设计人员需要按步骤及时、认真地建立一整套图纸资料，例如结构图、零件图、使用手册、维护手册等等，这些图纸资料是研制、使用和维护机器所必需的．研制机器的全过程大致如图 1.2(*a*)所示．

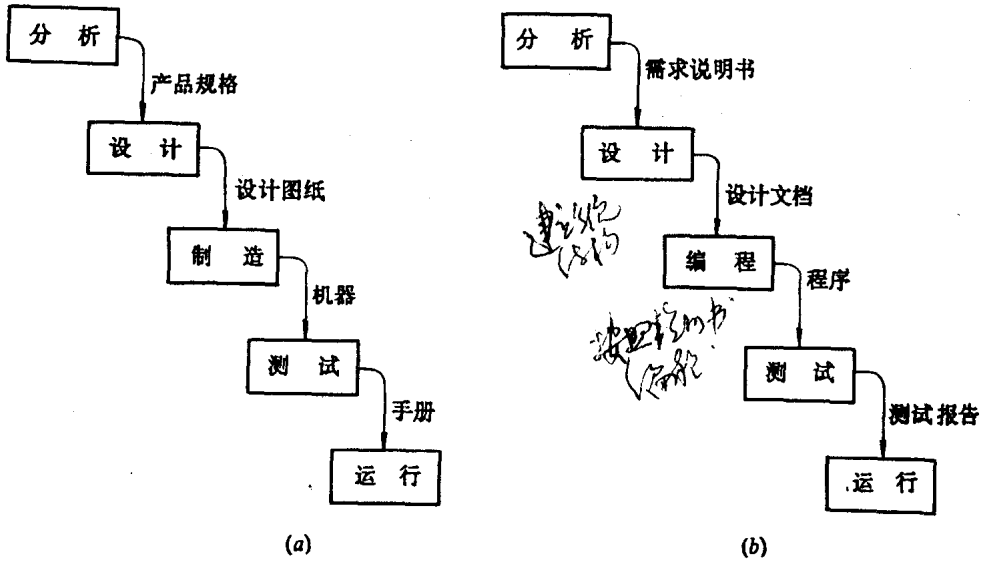


图 1.2

为了有效地进行管理，软件生命期亦可按类似的模式组织，即分为需求分析、设计、编程、测试和运行等五个阶段，每个阶段都有明确的任务，并需产生一定规格的文档资料交付给下一阶段，下阶段在上阶段交付文档的基础上继续开展工作．图 1.2(*b*)是这种软件生命期模型的(Software life cycle Model)的示意图，由于其形状似多级瀑布，常称为"瀑布模型"．

生命期模型中，前四个阶段有时又总称为开发期，最后一个阶段称为运行期．

表 1.2 概括地列出了每个阶段的基本任务、工作结果(即提交的文档)以及参加人员．

表 1.2

阶段		基本任务	工作结果	参加者
开发期	分析	理解和表达用户的要求	需求说明书	用户、高级程序员
	设计	建立系统的结构	模块说明书、数据说明	高级程序员
	编程	编写程序	程序	高级程序员、初级程序员
	测试	发现错误和排除错误	测试报告	另一独立的部门
运行期	运行	维护	改进的系统	

下面简述各阶段的工作概况。

1. 需求分析和规格说明阶段(简称分析阶段)

在设计软件系统之前，首先必须确定用户究竟要求软件系统做什么，所以分析阶段的基本任务是理解用户的需求，并将用户的需求用书面形式表达出来。这一阶段产生的文档是需求规格说明书(简称需求说明书)，它明确地描述了用户的要求。需求说明书是以后各阶段工作的基础。用户和软件人员双方都应有代表参加这一阶段的工作，需求说明书就是双方充分讨论交流后达成的协议。

2. 设计阶段

在设计阶段，要在需求说明书的基础上建立软件系统的"结构"，包括数据结构和模块结构。设计阶段一般又可分为两步：概要设计(或称为总体设计)和详细设计，前者主要考虑模块的分解，后者考虑每个模块内部的细节。本阶段产生的文档包括模块说明书、数据库或文件结构说明等。由于软件产品的质量在很大程度上取决于设计方案的质量，所以设计工作要由经验较丰富的高级软件人员来完成。

3. 编程阶段

在编程阶段，按模块说明书的要求为每个模块编写程序。相对地说，这个阶段是最简单的，初级程序员可以参加编程阶段的工作。

4. 测试阶段

由于前面三个阶段都可能产生各种各样的错误，所以测试阶段的任务就是发现并排除这些错误。静态检验工作实际上在上述每个阶段的最后就必须进行，而测试阶段则进行最后的动态检验。测试通常又可分为模块测试、集成测试和系统测试等几步。测试应该由另一个独立的部门(如不参加系统的设计或编程的人员)来完成。经过测试就得到了软件系统的第一个版本。

5. 运行阶段

由于软件系统经测试后仍然可能隐含着错误，用户的需求和系统的操作环境又可能发生变化，所以在运行阶段仍需对软件作继续排错和修改扩充，这类工作称为维护，其工作量是相当大的。软件经过维护就产生了多种不同的版本。

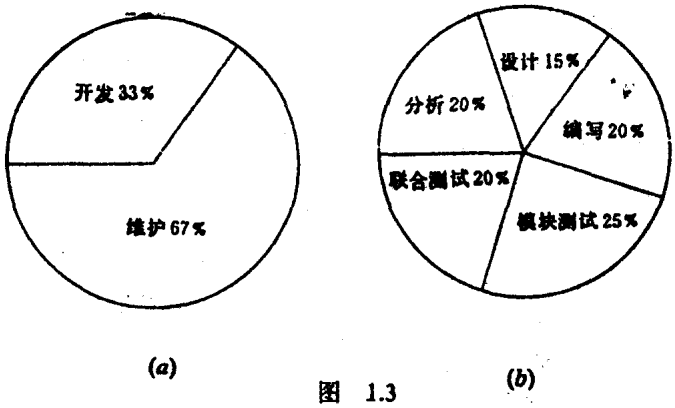
大型软件系统的运行期往往达 10 年或更长，软件经反复多次修改后，直至维护人员认为：再继续对其进行修改以便符合当初未曾预料到的一些需求，已是不可靠的了，此时软件系统才会被舍弃，软件生命期也就到此结束了。

Zelkowitz 对一些软件项目中各阶段的工作量进行了统计[3]，图 1.3 是他给出的结果。图 1.3(*a*)说明在整个生命期中，开发期和运行期所占的工作量，可以看出维护工作量要占一半以上。图 1.3(*b*)说明在开发期中各阶段所占的工作量，可以看出，测试工作量约占其中的一半。

图 1.3 告诉我们，编程工作在整个生命期中只占很小的比例，所以"开发软件仅仅是编程"的想法显然是错误的。

图 1.2(*b*)的瀑布模型将软件生命期组织成五个阶段，这是比较有代表性的模式。也有人提出其他一些模式，表 1.3 列出了 Freeman、Metzger、Boehm 等人的模式，可以看出，各种模式基本上是类似的。

由于软件规模大、逻辑复杂、生命期又长，而人的记忆力是有限的，所以有关软件系统的所有资料，必须以书面文档的形式记录下来，这样开发人员就能以文档为基础协同工作，各阶段之间也可通过文档实现过渡。显然，文档健全与否直接影响着最终产品的质量。



(a)　　(b)

图 1.3

为了强调软件产品除程序之外还必须包括各种文档资料，Boehm 为软件给出了新的定义："软件是程序以及开发、使用和维护程序所需的所有文档"[2]，因此，需求说明书、模块说明书、数据库和文件说明、程序、测试计划、测试用例、使用手册、维护手册……各阶段交付文档的全体就是"软件"。

由此可见，作为一名称职的软件开发人员，光会编程是不够的，他还必须掌握分析、设计、测试等方法和工具，学会编写上述各种文档。培养学生掌握这些技术就是本书的目的。

与传统的手工艺开发方式相比，上述生命期模型有两个明显的长处，第一，由于强调要将每个阶段的工作结果用书面形式描述出来，这就使原来"不可见"的软件变成了"可见"的文档资料；第二，开发过程分阶段按步骤进行，以交付某种特定规格的文档作为标志某个阶段完成的里程碑，这就使原来"难以管理的思考过程"变为"可以管理的生产过程"了。显然，这两点长处为提高软件生产率、改进软件质量创造了极为有利的条件。

表 1.3

Freeman	Metzger	Bochm
需求分析	系统定义	系统需求分析
说明		软件需求分析
总体设计	设计	基本设计
详细设计		详细设计
实现	编程	编程和排错
	测试 验收	测试和试行
	安装和运行	
维护		运行和维护

必须指出的是，实际软件系统的开发不可能理想化地按瀑布模型进行，由于人们理解问题总有一个反复的过程，所以从后阶段回复到前阶段是不可避免的，例如在设计阶段发现需求说明书有不完整或不正确之处，就必须进行"再分析"，测试阶段发现模块界面有错误，就必须进行"再设计"，在运行阶段为了扩充系统的功能又要进行"再分析"、"再设计"、"再编程"等。另外，阶段之间也没有明确的界线，如分析阶段需要考虑系统的可行性，就一定是会涉及一些设计方面的问题；又由于采用了模块化的技术，某些模块的编程有可能与另一些模块的测试并行进行。

1.3 软件质量的评价

软件工程学的最终目标是获得高质量的软件，所以如何评价软件质量是一个重要的问题．以前，对小型程序，人们一般比较强调程序的正确性和效率，近年来随着软件规模的增大和复杂性的上升，对问题的看法已发生了变化．目前，软件质量的定义还是非常模糊的，人们对此尚未形成一致的看法，但一般说来倾向于从可维护性、可靠性、可理解性和效率等方面对软件作较全面的评价，下面分别讨论之．

1. 可维护性(Maintainability)

软件在运行阶段尚需不断"修正"，因为软件虽经测试但不可避免地总还隐含着各种错误，这些错误在运行阶段会逐步暴露出来，因而就要进行排错．例如 IBM 公司的 OS／360 系统，据说每个版本中约有成千个错误；又如某军事系统在运行初期，平均每月发现 900 个错误，纠正一个错误平均需修改 17 条指令．

软件在运行阶段尚需不断"完善"，因为系统经过一个时期的使用后，用户必然会逐步提出一些更改或扩充要求，软件就需要相应地不断作修改．

软件在运行阶段往往还需作"适应性"修改，因为近年来计算机业发展迅速，一般在 3 至 5 年内，硬件或系统软件就会出现更新换代的新产品，于是应用软件系统也就需要作相应的调整或移植．

在运行期中，对软件所作的上述修正性、完善性和适应性修改，总称为"维护"，它涉及再分析、再设计、再编程、再测试等活动．考虑到大型软件系统的运行期可达 10 年以上，所以维护的工作量是极大的．

另外，维护工作也是相当困难的，由于软件逻辑上的复杂性，修改往往会带来新的错误．据统计，软件错误中有 19%是由于修改造成的；有人还统计出，如果一个修改涉及 5 至 10 个语句，修改成功的可能性是 50%，如果一个修改涉及 40 至 50 个语句，则修改成功的可能性下降至 20%，因此软件维护是很困难，很冒风险的．

图 1.4 说明了在计算机软硬件的总投资中，软件维护所占的比例．可以看出，维护费用近年正在迅速上升，按这样的趋势发展下去，现有的人力物力将全部被束缚在维护原有系统上，就可能再也没有力量去开发新的系统了．因此软件维护引起了人们的普遍关注，人们已意识到，一个软件系统即使其他方面都相当理想，但是如果不容易维护，它将不会有什么实际使用价值，所以，"可维护性"应该作为评价软件质量的重要准则．

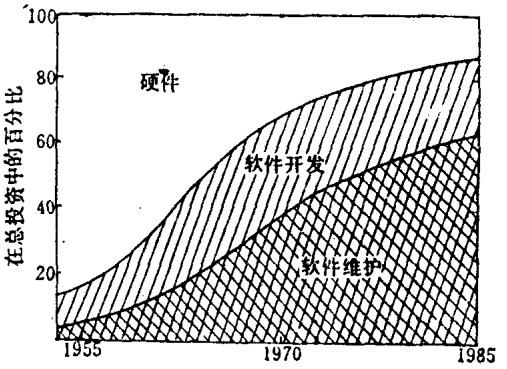


图 1.4

"可维护性"通常包括了"可读性"(Readability)、"可修改性"(Modifiability)、"可测试性"(Testability)等含义．为了使软件具有较好的可维护性，早在开发期的各个阶段就应采取一系列技术措施．这样做虽然开发期的工作量也许会大些，但考虑到维护工作在整个生命期中所占的比例，总的看来还是值得

的．反之，如果开发时目光短浅，不考虑长远利益，就可能在维护时遭受重大挫折，到时就无法挽救了．

本书第七章还要进一步讨论软件维护问题．

2. 可靠性(Reliability)

可靠性通常包括正确性(Correctness)和健壮性(Robustness)这两个相互补充的方面．

正确性是指软件系统本身没有错误，所以在预期的环境条件下能够正确地完成期望的功能．毋容置疑，正确性对系统正常发挥作用是完全必要的．

对于一个小型程序，我们可以希望它是完全正确的，但对长达几万行甚至几十万行的大型软件，我们一般不能奢望它是"完全"正确的，而且这一点也是无法证实的．

此外，一个大型系统运行时，完全可能遇到一些意外，例如：某部分硬件出现故障、软件中某个隐含的错误暴露出来、或者操作员无意地输入了一个离奇古怪的数字或符号．有的系统虽然是正确的，但是它非常"脆弱"，一旦发生上述异常情况，就可能遭到意想不到的破坏．

1972 年 6 月的"计算机世界"刊有一则报导："操作员手指的一滑使税收损失 30 万美元"，据载，1972 年 6 月美国 Woonsocket 市在用计算机计算税率时，由于操作员的右手小拇指无意滑到键 P 上，使汽车价格 950 美元误为 7 000 950 美元，从而导致了巨大的财政损失．

这就产生了一个新的概念——"健壮性"，其含义是指：当系统万一遇到意外时(具体是什么意外，事先是很难预料的)能按某种预定的方式作出适当的处理，如能立即意识到异常情况的出现，保护起重要的信息，隔离故障区防止事故蔓延，并能及时通知管理人员请求人工干预，事后从故障状态恢复到正常状态亦比较容易．所以健壮的系统应该能避免出现灾难性的后果．

正确性与健壮性是相互补充的，前面计算税收的系统可能是正确的，但它不是健壮的；而健壮的程序并不一定是绝对正确的，例如一个计算工资的程序，它可能错误地计算了某种罕见的病假折扣值，但它对输入数据以及系统内部的数据状态都进行仔细的查核，因而在绝大多数情况下能够可靠地工作．

总的说来，可靠的软件系统在正常情况下能够正确地工作，而且在意外情况下，亦能适当地作出处理，因而不会造成严重的损失．

当代计算机使用的范围很广，有些软件系统的故障可能造成生命财产的巨大损失，如核反堆泄漏，飞船失事爆炸等，所以，"可靠性"无疑是绝对重要的．人们宁可在开发时多化些代价，提高系统的可靠性，与发生事故后造成的损失相比，这些代价还是值得的．

3. 可理解性(Understandability)

在相当长一段时间中，人们一直认为程序只是提供给计算机的，而不是给人阅读的，所以只要它逻辑正确，计算机能按其逻辑正确执行就足够了，至于它是否易于被人理解则是无关紧要的．

但是随着软件规模的增大，人们逐步看到，在整个软件生命期中，为进行测试、排错或修改，开发人员经常需要阅读本人或他人编写的程序和各种文档．如果软件易于理解，无疑将提高开发和维护的工作效率，而且出现错误的可能性也会大大下降．所以，可理解性应该是评价软件质量的一个重要方面．

可理解性通常是指简单性和清晰性，对于同一用户要求，解决的方案可以有多个，其中最简单、最清晰的方案往往被认为是最好的方案。

4. 效率(Efficiency)

效率是指系统能否有效地使用计算机资源，如时间和空间等。这一点以前一直是非常强调的，这是过去硬件价格昂贵造成的结果。由于以下一些原因，目前人们对效率的看法已有了变化:

首先，硬件价格近年来大幅度下降，所以效率已不像以前那样举足轻重了。

第二，人们已认识到，程序员的工作效率比程序的效率远为重要，程序员工作效率的提高不仅能减少开支，而且出错率也会降低。从汇编语言发展到高级语言以至超高级语言就是一个很好的说明。

第三，追求效率同追求可维护性、可靠性等往往是相互抵触的，例如片面地强调节省时间和空间，设计出来的系统可能结构复杂，难以理解和修改；又如高度可靠的程序中一般需要含有冗余，如为一种帐目保存几个副本等，这就要以一定的时间和空间作代价。

所以，效率虽然是衡量软件质量的一个重要方面，但在硬件价格下降、人工费用上升的情况下，人们有时也宁可牺牲效率来换取其他方面的得益。

除了这里讨论的可维护性、可靠性、可理解性和效率之外，软件系统的许多其他性质也反映了软件的质量，图 1.5 是 Boehm 提出的软件质量图，下面只解释图中最右边各因素的含义，不再作详细讨论。

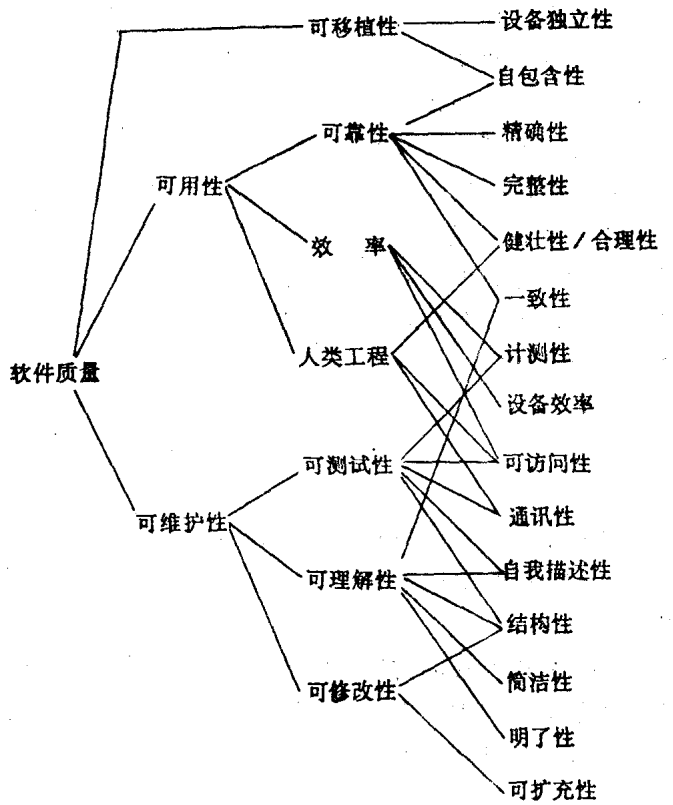


图 1.5

1) 设备独立性 (Device-independenty): 不依赖于特定的硬件设备而能工作的程度。

2) 自包含性 (Self-containedness): 不依赖于其他程序仅靠自身能实现功能的程度。

3) 精确性 (Accuracy): 能产生具有必要精确度之正确输出的程度。

4) 完整性 (Completeness): 所有部分是否齐全。每个部分是否充分。

5) 健壮性 / 合理性 (Robustness / Integrity): 即使前提条件不符合规格也能继续合理运行的程度。

6) 一致性(Consistency): 程序和文档中所用的记号、术语和表示方式的一致的程