高等院校计算机专业及专业基础课系列教材

软件工程

（第二版）

王立福　麻志毅　张世琨　编著

北京大学出版社

高等院校计算机专业及专业基础课系列教材

软件工程

(第二版)

王立福　麻志毅　张世琨　编著

Y.T.P

北京大学出版社
北　京

内容简介

本书是在北京大学计算机科学技术系使用《软件工程》教材的基础上，根据全国高等教育自学考试指导委员会制定的《软件工程考试大纲》的要求，由主讲、主考教师编写而成的，既是北京大学计算机系本科生指定教材，也是北京市高等教育自学考试指定教材。

本书结合国内外软件工程的发展，特别是国家"八五"、"九五"攻关实践，详细地讲述了软件工程的基本内容，包括基本概念、基本模型、基本方法及相应的支持工具。本书注重基础知识的系统性，同时注意选材的先进性，内容全面、层次清楚。

图书在版编目(CIP)数据

软件工程(第二版)/王立福等编著. －北京:北京大学出版社,2002.3
ISBN 7-301-03227-7

Ⅰ.软… Ⅱ.王… Ⅲ.软件工程-概论 Ⅳ.TP311.5

书　　名：软件工程(第二版)
著作责任者：王立福　麻志毅　张世琨
责 任 编 辑：沈承凤
标 准 书 号：ISBN 7-301-03227-7/TP·0318
出　版　者：北京大学出版社
地　　址：北京市海淀区中关村北京大学校内　100871
网　　址：http://cbs.pku.edu.cn
电　　话：发行部 62757298　编辑部 62752038
电 子 信 箱：zpup@pup.pku.edu.cn
排　版　者：兴盛达打字服务社　62549189
印　刷　者：中国科学院印刷厂
发　行　者：北京大学出版社
经　销　者：新华书店
787 毫米×1092 毫米　16 开本　17.25 印张　430 千字
2002 年 3 月第 2 版　2002 年 10 月第 1 次修订
2003 年 12 月第 4 次印刷
定　　价：23.00 元

序

“科教兴国”战略强调教育对国民经济的基础地位，要求高等教育“实施全面素质教育，加强思想品德教育和美育，改革教育内容、课程体系和教学方法……”。为了落实好“科教兴国”这一战略决策，北京大学计算机科学技术系与北京大学出版社合作，编审出版基础主干课和专业主干课系列教材。

目前，伴随着微电子和计算机科学技术渗透到社会的各个领域，人类正跨步迈进知识经济时代。在知识经济时代，具有创新能力的高素质人才是经济持续发展的必备条件。

计算机科学技术包括科学和技术两部分，不仅强调严谨的科学性，同时也注重工程性，是一门科学性和工程性并重的学科。信息科学技术的支柱学科是微电子、计算机、通信和软件，其中微电子是基础，计算机和通信是载体，软件是核心，它们相辅相成，共同培育了知识经济。因而，高素质的信息领域科技人才应该掌握上述学科的基础理论和专业技能。

近年来，北京大学计算机科学技术系通过跟踪、分析国际知名大学的相关课程设置、教学实施情况，借鉴国内兄弟院系的课程体系调整建议，总结北京大学计算机科学技术系集计算机软、硬件技术和微电子学于一体的人才培养经验，对课程体系进行了较大力度的梳理，形成了一系列基础主干课和专业主干课。

这一系列教材正是为配合课程体系的调整而编撰的。所选书稿主要是在我系多年的教学实践中师生反映较好的讲义和教材的基础上修编而成的。我们希望这批教材能够达到“注重基础、淡化专业（或突出交叉）、内容系统、选材先进、利于教学”的要求。

对于教材中的不足之处，欢迎广大读者不吝赐教。

杨芙清

一九九九年九月

北京大学计算机科学技术系

专业基础课和专业课教材编审指导小组

组　　长： 杨芙清

成　　员：（按姓氏笔画序）

卢晓东　李晓明　许卓群　沈承凤　屈婉玲

张天义　赵宝瑛　袁崇义　董士海　程　旭

北京大学计算机系专业基础课名称

计算引论

数字逻辑

微机原理

计算机组织与体系结构

离散数学

数据结构

编译原理

操作系统

微电子学概论

集成电路原理与设计

北京大学计算机系专业课名称

计算机网络概论

数据库概论

软件工程

计算机图形学

面向对象技术引论

再版前言

编写一本适合本科生学习的软件工程教材，实在是一件很难的事情。其原因主要有三：一是软件工程这门课程所涉及的内容十分广泛，既涉及技术层面，又涉及管理层面；既关联实际问题的理解和描述，又关联软件工具的使用；……；其中既有哲学问题，又有方法学问题；二是在社会需求的拉动下，软件工程技术发展非常迅速，新概念、新技术、新方法不断出现，实在有些顾及不暇之势；三是作为一门学科，仅仅走过了30余年的发展历程，与其他学科相比，例如数学、物理、化学以及建筑、通信等，还是相当"年轻"的，但从另一个角度来说，仍是一门不算成熟的学科。因此，在教材内容的选取与组织方面，在有关概念的表述方面，真是一种挑战。

比较幸运的是，通过参与杨芙清院士主持的国家攻关项目，通过参与张效祥院士主编的《计算机科学技术百科全书》的编写，通过参与国家有关标准规范的制定，特别是通过几年来的教学实践，对软件工程这四个字还算有了一点领悟。因此，在本书的再版中，以原有教材为基础，进行了比较大的改动。

在教材内容的选取上，遵循以下两条原则，一是选取的内容能够有助于提高读者求解软件问题的能力，特别是提高读者直接参与软件开发实践和工程管理的能力；二是选取的内容应该是基础性的，是比较"稳定"的，但这并不意味着不能引入新的方法和技术，而是要讲清楚。

在教材内容的组织上，基本上以软件工程概念和相关的软件工程框架为"纲"，逐"目"展开，使概念与相关技术、方法有机地融为一体。

在概念的表述上，依据内容组织的特定层面，尽力引用《计算机科学技术百科全书》中有关的条目，并注重语义和概念之间关系的阐述。

由于时间仓促，更主要的是由于水平问题，再版中依然还会存在很多不足和错误，真诚地希望读者提出，并通过电子邮件(wlf@cs.pku.edu.cn)和其他方式，进行更有意义的讨论。

目　录

第一章　软件工程概论 …… (1)

1.1　软件工程概念 …… (1)

1.2　软件工程框架 …… (2)

习题一 …… (3)

第二章　软件开发模型 …… (5)

2.1　瀑布模型 …… (5)

2.2　演化模型 …… (7)

2.3　螺旋模型 …… (7)

2.4　喷泉模型 …… (9)

2.5　增量模型 …… (10)

习题二 …… (10)

第三章　结构化需求分析 …… (11)

3.1　需求获取 …… (11)

3.2　需求规约 …… (19)

*3.3　需求验证 …… (28)

3.4　需求分析文档 …… (32)

3.5　实例研究 …… (35)

习题三 …… (41)

第四章　结构化设计 …… (43)

4.1　总体设计的目标及其表示 …… (43)

4.2　总体设计方法 …… (46)

4.3　设计评价准则与启发式规则 …… (56)

4.4　设计优化——初始模块结构图的精化 …… (62)

4.5　详细设计 …… (65)

4.6　软件设计规格说明书 …… (71)

习题四 …… (74)

第五章　面向对象方法 …… (77)

5.1　概念与表示法 …… (77)

*5.2　过程指导 …… (103)

*5.3　OSA 方法简介 …… (124)

习题五 …… (153)

第六章　软件测试 …… (154)

6.1　软件测试目标与软件测试过程模型 …… (154)

6.2　软件测试技术 …… (155)

6.3 软件测试步骤 …… (168)
*6.4 程序证明技术 …… (172)
习题六 …… (182)
第七章 软件过程与改善 …… (184)
7.1 软件过程 …… (184)
*7.2 ISO9000-3 简介 …… (202)
7.3 能力成熟度模型(CMM)简介 …… (210)
习题七 …… (226)
第八章 软件开发工具与环境 …… (227)
8.1 CASE 概述 …… (227)
8.2 工作台 …… (230)
8.3 软件开发环境 …… (235)
习题八 …… (257)
附录 1 面向对象分析实践指南(要点) …… (259)
附录 2 面向对象设计实践指南(要点) …… (263)
参考文献 …… (268)

注：目录中带有＊号的章节，不作为自考学生的考试内容。

第一章　软件工程概论

软件工程这一术语首次出现在1968年的NATO会议上。60年代以来，随着计算机的广泛应用，软件生产率、软件质量远远满足不了社会发展的需求，成为社会、经济发展的制约因素。当时，软件开发虽然有一些工具支持，例如编译连接器等，但基本上还是依赖开发人员的个人技能，没有可遵循的原理、原则和方法，也缺乏有效的管理。软件可靠性、可维护性较差，而且往往超出预期的开发时间要求。软件工程这一概念的提出，其目的是倡导以工程的原理、原则和方法进行软件开发，以期解决当时出现的"软件危机"。

产生软件危机的原因很多，除了与软件本身固有的特征有关以外，还与软件开发范型、软件设计方法、软件开发支持以及软件开发管理等有关。

软件工程作为一门学科已有近30年的历史，其发展大体可划分为两个时期。

60年代末到80年代初，软件系统的规模、复杂性以及在关键领域的广泛应用，促进了软件开发过程的管理及工程化开发。这一时期主要围绕软件项目，开展了有关开发模型、支持工具以及开发方法的研究。其主要成果体现为：提出了瀑布模型；开发了诸多结构化语言（例如PASCAL语言、C语言、Ada语言等）和结构化方法（例如"自顶向下"方法），试图向程序员提供好的需求分析和设计方法，并开发了一些支持工具，例如调试工具等；开始出现各种管理方法，例如费用估算、文档复审，开发了一些相应支持工具，例如计划工具、配置管理工具等。这一时期的主要特征可概括为：前期主要研究系统实现技术，后期则开始强调管理及软件质量。

自"软件工厂"这一概念提出以来，80年代初主要围绕软件工程过程，开展了有关软件生产技术，特别是软件复用技术和软件生产管理的研究和实践。其主要成果是提出了具有广泛应用前景的面向对象方法和相关的语言（例如Smalltalk，C++，Eiffel等）；大力开展了计算机辅助软件工程（CASE）的研究与实践（例如我国在"七五"、"八五"、"九五"期间，均把这一研究作为国家重点科技攻关项目），各类CASE产品相继问世。其间，最显著的事件是过程改进项目，该项目的目标是在工业实践中，建立一种量化的评估程序，判定软件组织成熟的程度。

近几年来，软件工程的研究已从过程（管理）转向产品（开发），更加注重新的程序开发范型和软件生产。其中，围绕网络，特别是Internet网的广泛应用，以软件复用技术研究为基础，在软件构件技术及软件"平台"技术研究方面，在需求工程技术及领域分析技术研究方面，以及在软件体系结构、应用框架、面向应用的语言研究方面，均取得了非常有影响的成果，有力地促进了软件工程学科和软件产业的发展。

1.1　软件工程概念

计算机系统中的程序及其文档称为软件。其中，程序是计算机任务的处理对象和处理规则的描述；文档是为了理解程序所需的阐述性资料。细言之，软件一词具有三层含义。一为个体含义，即指计算机系统中的程序及其文档；二为整体含义，即指在特定计算机系统中所有上述个体含义下的软件的总称，亦即计算机系统中硬件除外的所有成分；三为学科含义，即指在

研究、开发、维护以及使用前述含义下的软件所涉及的理论、方法、技术所构成的学科。一般而言,工程是将科学论理和知识应用于实践的科学。在了解了“软件”和“工程”两个概念的基础上,软件工程可定义如下:

软件工程是一类求解软件的工程。它应用计算机科学、数学及管理科学等原理,借鉴传统工程的原则、方法,创建软件以达到提高质量、降低成本的目的。其中,计算机科学、数学用于构造模型与算法,工程科学用于制定规范、设计范型、评估成本及确定权衡,管理科学用于计划、资源、质量、成本等管理。软件工程是一门指导计算机软件开发和维护的工程学科。

1.2 软件工程框架

软件工程与其他工程(例如土木工程)一样,有其自己的目标、活动和原则。软件工程框架如图 1.1 所示。

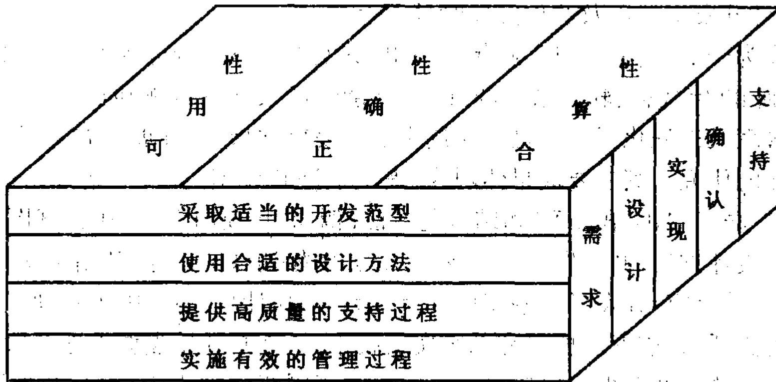


图 1.1 软件工程框架

软件工程的目标可概括为“生产具有正确性、可用性以及开销合宜的产品”。正确性意指软件产品达到预期功能的程度。可用性意指软件基本结构、实现及文档为用户可用的程度。开销合宜是指软件开发、运行的整个开销满足用户要求的程度。这些目标的实现不论在理论上还是在实践中均存在很多问题有待解决,它们形成了对过程、过程模型及工程方法选取的约束。

软件工程活动是“生产一个最终满足需求且达到工程目标的软件产品所需要的步骤”。主要包括需求、设计、实现、确认以及支持等活动。需求活动包括问题分析和需求分析。问题分析获取需求定义,又称软件需求规约。需求分析生成功能规约。设计活动一般包括概要设计和详细设计。概要设计建立整个软件体系结构,包括子系统、模块以及相关层次的说明、每一模块的接口定义。详细设计产生程序员可用的模块说明,包括每一模块中数据结构说明及加工描述。实现活动把设计结果转换为可执行的程序代码。确认活动贯穿于整个开发过程,实现完成后的确认,保证最终产品满足用户的要求。支持活动包括修改和完善。伴随以上活动,还有管理过程、支持过程、培训过程等。

围绕工程设计、工程支持以及工程管理,提出了以下四条基本原则:

第一条原则是选取适宜的开发模型。该原则与系统设计有关。在系统设计中,软件需求、

硬件需求以及其他因素之间是相互制约、相互影响的，经常需要权衡。因此，必须认识需求定义的易变性，采用适宜的开发模型予以控制，以保证软件产品满足用户的要求。

第二条原则是采用合适的设计方法。在软件设计中，通常要考虑软件的模块化、抽象与信息隐蔽、局部化、一致性以及适应性等特征。合适的设计方法有助于这些特征的实现，以达到软件工程的目标。

第三条原则是提供高质量的工程支持。“工欲善其事，必先利其器”。在软件工程中，软件工具与环境对软件过程的支持颇为重要。软件工程项目的质量与开销直接取决于对软件工程所提供的支撑质量和效用。

第四条原则是重视开发过程的管理。软件工程的管理，直接影响可用资源的有效利用，生产满足目标的软件产品，提高软件组织的生产能力等问题。因此，仅当软件过程予以有效管理时，才能实现有效的软件工程。

在这一框架中，软件工程活动主要包括需求、设计、实现、确认和支持。其中，

(1) 需求的任务就是定义问题，即通过需求获取，得到一个需求陈述，继之，以需求陈述为基础，采用一种形式化或半形式化途径，给出被建系统的模型，进而按照一定的标准编制该系统的需求规格说明书，或曰需求规约；最后，还要验证需求陈述和需求规约之间的一致性、完整性、可跟踪性等。

(2) 软件设计的任务是在需求的基础上，给出被建系统的软件设计方案。一般来说，软件设计包括总体设计和详细设计。总体设计给出被建系统的软件体系结构，例如层次模块体系结构，消息总线体系结构，以数据库为中心的体系结构等。详细设计的任务是定义体系结构中的每一模块或构件，即给出它们的实现算法。由于在软件设计中采用的方法不同，因此一般来说其设计结果的表示风格也是不同的。

(3) 实现的任务是在软件设计的基础上，编码被建系统软件体系结构中的每一模块或构件，其方式可以是，或直接采购所需的模块和构件，或使用一种特定的语言或程序设计环境(例如 VB, VC)直接编码，或修改、包装已有的模块或构件。

(4) 确认工作贯穿软件开发的整个过程，主要包括需求复审、设计复审以及程序测试。其中应当注意的是，软件错误大部分出现在需求阶段，其次是出现在设计阶段，而在编码阶段出现的错误与前两个阶段相比是出现错误最少的阶段。

(5) 支持的任务是为系统的运行提供纠错性维护和完善性维护，即对系统运行中发现的错误进行改正，对系统运行中发现的欠缺进行修改。

综上所述，这一软件工程框架告诉我们，软件工程目标是可用性、正确性和合算性；实施一个软件工程要选取适宜的开发模型，要采用合适的设计方法，要提供高质量的工程支撑，要实行开发过程的有效管理；软件工程活动主要包括需求、设计、实现、确认和支持等活动，每一活动可根据特定的软件工程，采用合适的开发模型、设计方法、支持过程以及过程管理。根据软件工程这一框架，软件工程学科的研究内容主要包括：软件开发模型、软件开发方法、软件过程、软件工具、软件开发环境、计算机辅助软件工程(CASE)以及软件经济学等。

习 题 一

1. 解释以下术语：

软件　　工程　　软件工程

2．简述以下问题：

(1) 软件工程目标； 生产具有正确性，可用性以及开销合宜的产品

(2) 软件工程原则； 选取适宜开发模型，采用合适的设计方法，提供高质量的工程支持，重视开发过程的管理

(3) 软件与程序之间的关系； 软件是指计算机系统中的程序及其文档

程序是计算机任务的处理对象和处理规则的描述

文档是为了理解程序所需的阐述性资料

(4) 软件工程目标、原则和活动三者之间的关系。

3．概要叙述软件工程各活动的主要任务和目标。

4．简要叙述软件工程学科研究的内容。

软件开发模型

软件开发方法

软件过程

软件工具

软件开发环境

计算机辅助软件

软件经济学

软件工程活动：需求 — 给出模型 需求规格说明 验证需求描述与需求规约之间的一致性

设计 — 软件体系结构（总体）

定义体系结构中的每一模块和构件（实现细节）

实现 — 编程 直接编写或修改已有的模块与构件

确认 → 贯穿始终（需求，设计 —

支持 — 系统运行中发现的错误修改

软件

第二章　软件开发模型

软件开发模型是软件开发全部过程、活动和任务的结构框架。软件开发模型能清晰、直观地表达软件开发全部过程，明确规定要完成的主要活动和任务，它用来作为软件项目工作的基础。模型都应该是稳定和普遍适用的。

软件开发包括需求、设计、编码和测试等阶段，有时也包括维护阶段。软件开发模型对于不同的应用系统，允许采用不同的开发手段和方法，使用各种不同的程序设计语言以及各种不同技能的人员参与工作，还应允许采用不同的软件工具或各种不同的软件工程环境。

最早出现的软件开发模型是 1970 年 W. Royce 提出的瀑布模型，而后随着软件工程学科的发展和软件开发的实践，相继提出了演化模型、螺旋模型、增量模型、喷泉模型等。

2.1　瀑布模型

瀑布模型将软件生存周期的各项活动规定为依固定顺序连接的若干阶段工作，形如瀑布流水，最终得到软件产品。

瀑布模型可追溯到 50 年代末期，当时人们已感到必须先确认“做什么”，才能编制程序将其实现，即使是比较简单的小型问题也不例外。最简单的两级瀑布模型如图 2.1 所示。

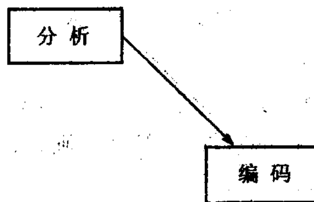


图 2.1　两级瀑布模型

对于较大软件项目，问题更加复杂，两级模型已不能满足软件开发的实际需要，一个更精确的软件开发步骤可按需要解决问题的顺序依次为：做什么—如何做—制作—检测—使用，于是一个反映软件过程的基本框架如图 2.2 所示。

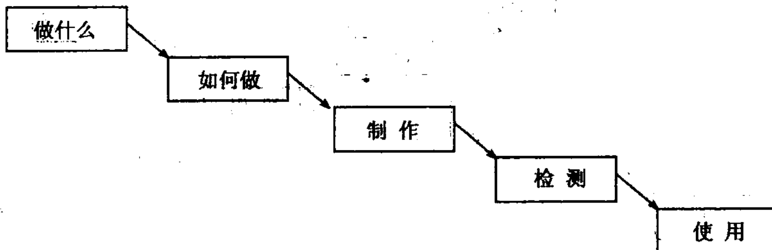


图 2.2　瀑布模型雏型

图 2.2 表明，首先应给出软件的目标，确定要做什么；然后要决定如何达到这一目标，给出策略、方法和步骤；继而加以实现，制作出所需要的软件；经过适当的检测，判定符合初始目标以后，方可投入运行和使用。可以说这是瀑布模型的雏型。

1970 年 W. Royce 首先将这一模型精确化，提出了具有多个开发阶段的瀑布模型，如图 2.3所示。

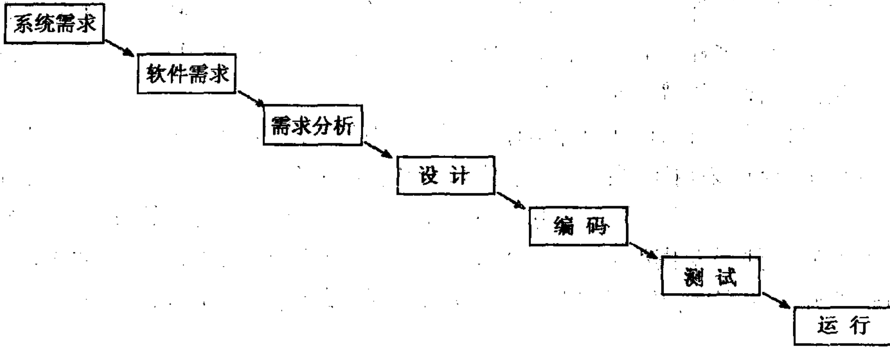


图 2.3 初始瀑布模型

这一模型规定了各开发阶段的活动为：提出系统需求、提出软件需求、需求分析、设计、编码、测试和运行，并且还规定了自上而下相互衔接的固定顺序，于是构成了人们熟知的瀑布模型。然而实践表明，各开发阶段间的关系并非完全是自上而下的线性图式，软件开发的实际情况是，每个开发阶段均具有以下特征：

(1) 从上一阶段接受本阶段工作的对象，作为输入；

(2) 对上述输入实施本阶段的活动；

(3) 给出本阶段的工作成果，作为输出传入下一阶段；

(4) 对本阶段工作进行评审，若本阶段工作得到确认，则继续下阶段工作；否则返回前一阶段，甚至更前阶段。

为表达向前阶段的反馈，在模型图中增加了虚线表示的箭头，构成了具有反馈回路的瀑布模型，如图 2.4 所示。

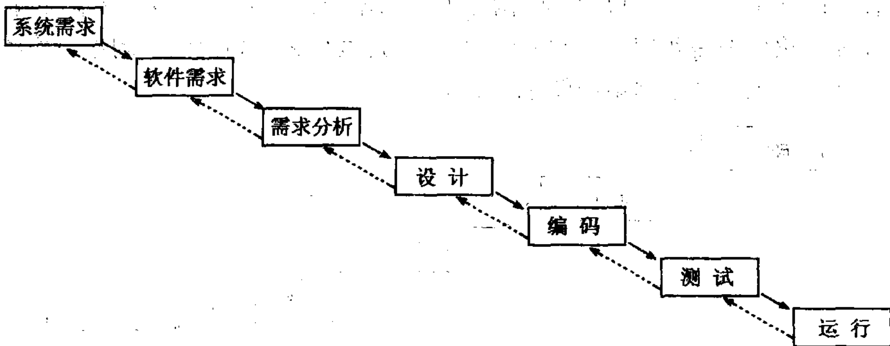


图 2.4 初始瀑布模型

瀑布模型有着不同形式的变种，比如，另一常见的具有反馈回路的瀑布模型包括的七个阶段是：可行性研究、需求分析和规约、设计和规约、编码和单元测试、集成测试和系统测试、交付、维护。不同形式瀑布模型的变种之间并无本质差别，选择哪一种形式可由软件项目特性及开发组织决定。

许多采用瀑布模型的开发组织为有效地组织实施，制定了软件开发规范或开发标准，其中明确规定了各个开发阶段应交付的产品。这就为严格控制软件开发项目的进度，最终按时交付产品以及保证软件产品质量创造了有利条件。

瀑布模型20多年来之所以广泛流行，是因为它在支持结构化软件开发、控制软件开发的复杂性、促进软件开发工程化等方面起着显著作用。与此同时，瀑布模型在大量软件开发实践中也逐渐暴露出它的缺点。其中最为突出的缺点是该模型缺乏灵活性，无法通过开发活动澄清本来不够确切的软件需求，这些问题可能导致开发出的软件并不是用户真正需要的软件，无疑要进行返工或不得不在维护中纠正需求的偏差，为此必须付出高额的代价，为软件开发带来了不必要的损失。并且，随着软件开发项目规模的日益庞大，该模型的不足所引发的问题显得更加严重。

2.2 演化模型

演化模型主要针对事先不能完整定义需求的软件开发。用户可以给出待开发系统的核心需求，并且当看到核心需求实现后，能够有效地提出反馈，以支持系统的最终设计和实现。软件开发人员根据用户的需求，首先开发核心系统。当该核心系统投入运行后，用户试用之，完成他们的工作，并提出精化系统、增强系统能力的需求。软件开发人员根据用户的反馈，实施开发的迭代过程。每一迭代过程均由需求、设计、编码、测试、集成等阶段组成，为整个系统增加一个可定义的、可管理的子集。如图2.5所示。

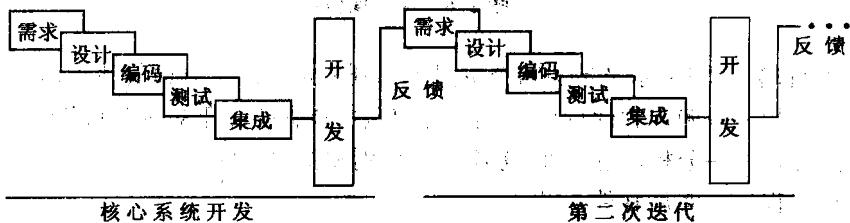


图2.5 演化模型

如果在一次迭代中，有的需求不能满足用户的要求，可在下一次迭代中予以修正。

演化模型在一定程度上减少了软件开发活动的盲目性。

2.3 螺旋模型

螺旋模型是在瀑布模型和演化模型的基础上，加入两者所忽略的风险分析所建立的一种软件开发模型。该模型于1988年由TRW公司B·鲍姆(Barry W. Boehm)提出。

软件风险是任何软件开发项目中普遍存在的问题,不同项目其风险有大有小。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就越大,承担项目所冒的风险也越大。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和软件产品的质量。软件风险驾驭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。

螺旋模型如图 2.6 所示。

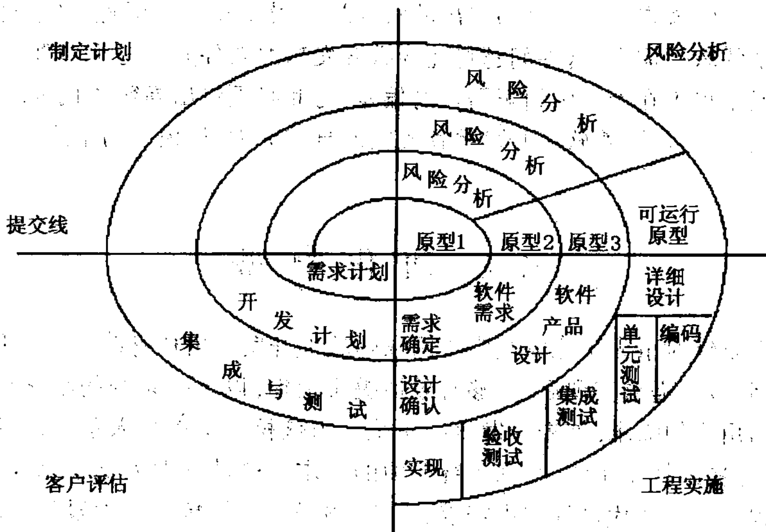


图 2.6 螺旋模型

沿着螺线旋转,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。

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,需求具有不确定性,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,考虑其他开发模型,并把需求作进一步修正。

客户对工程成果作出评价后,给出修正建议。在此基础上需再次计划,并进行风险分析。在每一圈螺线的风险分析的终点作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。图 2.7 给出了螺旋模型的另一图示。

如果对所开发项目的需求已有了较好的理解或较大的把握,无需开发原型,便可采用普通

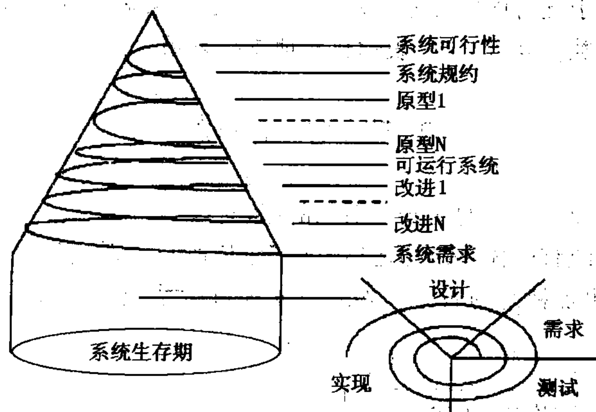


图 2.7 螺旋模型的另一种表示

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，需要开发原型，甚至需要不止一个原型的帮助，那就要经历多圈螺线。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，它是颇为实际的方法，它吸收了 T. Gilb 提出的软件工程"演化"概念。使得开发人员和客户对每个演化层出现的风险均有所了解，并继而作出反应。和其他模型相比，螺旋模型的优越性较为明显，但要求许多客户接受和相信演化方法并不容易。本模型的使用需要具有相当丰富的风险评估经验和专门知识。如果项目风险较大，又未能及时发现，势必造成重大损失。此外，螺旋模型是出现较晚的新模型，远不如瀑布模型普及，要让广大软件人员和用户接受，还有待于更多的实践。

2.4 喷泉模型

喷泉模型体现了软件创建所固有的迭代和无间隙的特征。喷泉模型如图 2.8 所示。

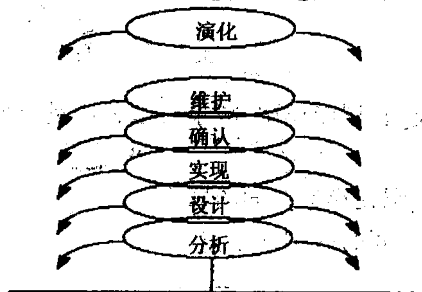


图 2.8 喷泉模型

这一模型表明了软件刻画活动需要多次重复。例如，在编码之前，再次进行分析和设计，其间，添加有关功能，使系统得以演化。同时，该模型还表明活动之间没有明显的间隙，例如在分析和设计之间没有明显的界限。

喷泉模型主要用于支持面向对象开发过程。由于对象概念的引入，使分析、设计、实现之间的表达没有明显间隙。并且，这一表达自然地支持复用。

2.5 增量模型

增量模型表达了如下所述的对开发活动的组织：

在设计了软件系统整体体系结构之后，首先完整地开发系统的一个初始子集；继之，根据这一子集，建造一个更加精细的版本。如此不断地进行系统的增量开发。这一模型如图 2.9 所示。

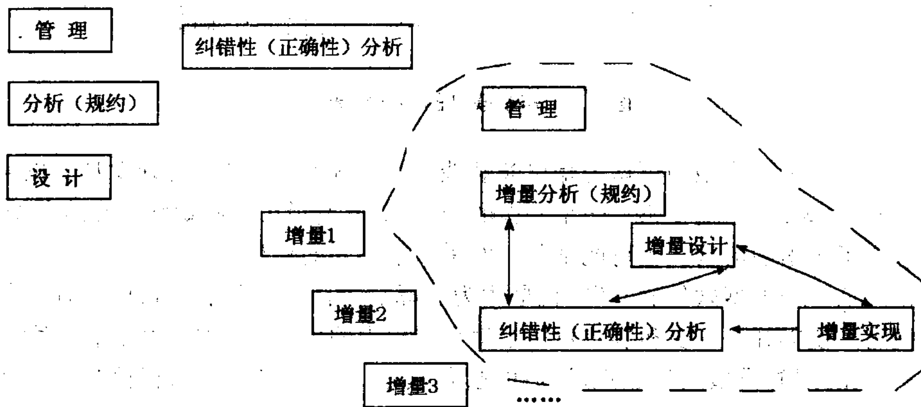


图 2.9 增量模型

该模型提供了一种在精化软件产品过程中体现用户经验的途径，也提供了一种周期性地进行最新软件维护、服务于各自用户的途径。

由于增量地进行开发，因此一个增量功能就比较容易理解和测试。这一模型广泛地使用于计算机工业中。

软件开发模型除以上介绍的几种模型外，还有原型/模拟模型、组装可复用构件模型等。

习 题

1. 简述瀑布模型、演化模型、螺旋模型、喷泉模型和增量模型的要点。

2. 分析：

(1) 瀑布模型、演化模型、螺旋模型之间的联系；

(2) 演化模型、增量模型之间的区别。

第三章 结构化需求分析

"软件需求分析"一词有两重含义。一是指从用户给出的需求陈述出发,经过用户与软件开发人员的合作,由非形式的、不精确的、不完整的需求陈述逐步转化为完整的、很可能采用一些半形式以及形式的表示方法表述的软件需求定义(有人亦称为软件需求规约)的过程。第二重含义则不仅指从用户需求出发得到需求定义,而且还包括从软件需求定义转换到相应的形式功能规约的过程。软件需求分析是软件开发的初始阶段。本书中,考虑到软件开发的实际情况,所提及的软件需求分析指的是第一重含义。

需求分析阶段位于软件开发的前期,它的基本任务是准确地定义未来系统的目标,确定为了满足用户的需要系统必须做什么。

通常,人们在开始做任何一件事情以前,都必须对所要达到的目标或所要解决的问题有一个清楚而明确的认识。同样,软件开发人员在设计一个系统之前,必须事先了解用户希望未来的系统能做什么,这项工作通常是由称为系统分析员(或简称分析员)的人来承担的。需求分析主要分为两个阶段——需求获取和需求规约。首先,系统分析员必须通过学习以及同用户的交互,熟悉用户领域的知识,并获得用户对未来系统的需求,这称为需求获取;其次,系统分析员在获得了用户的初步需求之后,必须进行一致性分析和检查,通过和用户协商解决其中存在的二义性和不一致性,并以一种规范的形式准确地表达用户的需求,形成所谓的需求规格说明书,这称为软件需求规约。

3.1 需 求 获 取

开发系统总是有目的的。当用户要求我们开发一个软件系统时,他通常会提出一些对系统做什么的要求——用新系统代替现存系统或改进现存系统的工作模式。需求就是对系统特征以及为了完成用户任务,系统必须能够做什么的一个描述,它关心的是系统目标而不是系统实现。例如,假设我们正在为用户开发一个工资管理系统,其中的一个需求可能是每个月发一次工资,另一个需求可能是把达到或高于某个工资水平的雇员的工资直接存入银行,用户可能还会提出远程访问公司的工资管理系统的要求。所有这些需求都是对未来系统的功能或特征的一些特定描述。

需求获取的目的是清楚地理解所要解决的问题,完整地获取用户需求,主要包括以下几方面的活动:通过学习、请教领域专家、向用户提问等手段,了解所要解决的问题,理解用户的需要,确认谁是真正的用户,以及系统实现所受到的各种限制。

需求获取通常面临着三大挑战:

(1) 问题空间理解

随着计算机在社会各方面不断广泛和深入的应用,大多数情况下,我们根本不可能在用户业务和应用方面以行家自居,但开发系统的任务又要求我们必须把握和深入理解问题空间,甚至比那些整天从事用户业务却没有全面考虑的人更要体察入微,而且必须尽可能快地做到这

一点。

(2) 人与人之间的通信

分析的挑战还包括通信,分析员在整个分析过程中都需要通信。为了从用户处获得对问题空间的理解和需求,分析员必须与用户通信,他要考虑通信问题而且要自行推敲,还要和同事互相切磋,最后还必须把他对问题空间的理解及由此得出的需求反馈给用户,检验他对需求的理解。他可能还要帮助用户放弃一些受财力和进度限制而难以达到的需求。

(3) 需求的不断变化

需求一直处于不断的变化之中,管理人员或用户可以在某一特定的时刻人为冻结需求。但是,真正的需求和用户所需要的系统却不断在演变,影响需求变化的因素很多:用户、竞争者、协调人员、审批人员和技术人员。正如 Gerhard Fisher 于 1989 年指出的那样:我们不得不接受变化着的需求这个现实生活中的事实,而不应指责它是混乱思想的产物。作为分析员,应千方百计构造一些符号和策略以使他们的工作能适应变化。

下面我们分别讨论需求获取包含的内容、应遵循的原则以及采用的技术。

3.1.1 需求获取的内容

需求定义描述了未来系统的行为,我们可以把任何一个系统想像为一个有限状态自动机,即在任一时刻都处于一组可接受的状态集中。系统的活动,比如和输入/输出设备的交互、某一时刻时钟事件的到来等,都促使系统从一个状态变迁到另一个状态。需求定义就描述了系统可能处于的状态集合以及从一个状态到另一个状态的变迁规则。

依此观点我们可以把用户需求分为两大类;功能性需求和非功能性需求。前者定义了系统做什么,包括系统的所有输入、输出以及如何从输入映射到输出;后者定义了系统工作时的特性,例如系统对效率、可靠性、安全性、可维护性、可移植性、吞吐量以及符合某种标准等的要求,这些要求对我们选择解决问题的方案起限制作用。

我们可以进一步把以上两类用户需求分为以下几个方面:

1. 物理环境

(1) 操作设备的地点在何处?

(2) 位置是集中的还是分散的?

(3) 有无对环境的限制,诸如温度、湿度或磁场干扰?

2. 界面

(1) 有来自其他系统的输入吗?

(2) 有到其他系统的输出吗?

(3) 对数据格式有规定吗?

(4) 对数据存储介质有规定吗?

3. 用户或人的因素

(1) 谁将使用该系统?

(2) 存在多种类型的用户吗?

(3) 每种类型的用户,熟练程度如何?

(4) 每种类型的用户,需要接受什么样的训练?

(5) 对用户来说,理解和使用系统的难度如何?

(6) 用户错误操作系统的可能性如何?

4. 功能

(1) 系统将做什么?

(2) 系统何时做什么?

(3) 系统何时及如何修改或升级?

(4) 对于执行速度、响应时间或吞吐量有无限制?

5. 文档

(1) 需要哪些文档?

(2) 文档针对什么样的读者?

6. 数据

(1) 输入和输出数据的格式如何?

(2) 接收和发送数据的频率如何?

(3) 数据的准确性如何?

(4) 计算必须达到的精度如何?

(5) 系统处理的数据流量多少?

(6) 数据必须保持一段时间吗?

7. 资源

(1) 建造、使用和维护系统需要什么设备、人员或其他资源?

(2) 开发者必须具有什么样的技能?

(3) 系统将占用多大的物理空间?

(4) 对电力、暖气或空调的要求如何?

(5) 开发有规定的时间表吗?

(6) 用于开发的软硬件投资有无限制?

8. 安全性

(1) 必须对访问系统或系统信息加以控制吗?

(2) 一个用户的数据如何同其他用户的数据隔离开来?

(3) 用户程序如何同其他程序和操作系统隔离开来?

(4) 多长时间需要对系统做一次备份?

(5) 备份的数据必须放到另外的地方吗?

(6) 需要防火或防盗吗?

9. 质量保证

(1) 对系统的可靠性要求如何?

(2) 系统必须监测和隔离错误吗?

(3) 规定的系统平均出错时间是多少?

(4) 发生错误后,重启系统允许的最大时间是多少?

(5) 系统变化将如何反映到设计中?

(6) 维护只是包括修改错误,还是也包括对系统的改进?

(7) 在资源使用和时间响应上采取了哪些行之有效的方法?

(8) 系统的可移植性如何?

3.1.2 需求获取应遵循的原则

现实世界中的事物是复杂多变的，人们常常无法一下子认识清楚，这就需要采取一些符合人类思维的方法和策略，其中，抽象和分解就是人们在认识世界和改造世界的长期实践中总结出来的行之有效的法则。反映到需求获取过程中，划分、抽象和投影是人们常用的组织信息的三条基本原则。

划分：是降低问题复杂性的基本途径之一。其中主要捕获问题空间的“整体/部分”关系。例如，让我们考察一个导弹防御系统，整个系统从功能上可以划分为目标监测、通信、情报收集与决策、防御武器发射和防御武器制导等子系统，于是处理一个复杂问题就变成了处理各个子问题，从而降低了问题的复杂性。进一步，各子问题如果仍然比较复杂，可以进一步把它们划分为子/子问题。

抽象：是认识、构造问题的一般途径。其中主要捕获问题空间的“一般/特殊”或“特例”关系。例如，还是上面的导弹防御系统，我们可以把敌方的导弹分为不同的类型：常规导弹和核导弹；弹道导弹和可制导导弹；远程、中程和近程导弹，以上三个抽象的例子表明各种类型的导弹都是某种特殊的导弹或导弹的一个特例。

投影：是描述问题的基本手段之一。其中主要捕获问题空间的多维“视图”。让我们继续考察上面的例子，从不同的角度看待系统中的导弹，这些角度包括导弹操作员、指挥员、情报分析专家以及敌方等，每个角度对系统的观察所反映的都是整个系统的一个“视图”，就像建筑设计师可以从正面、侧面、背面和剖面考察一座建筑物一样。

总之，划分、抽象和投影都可以定义问题空间中的结构或层次关系。在不同的上下文中，A 从属于 B 意味着不同的含义。对于划分，意味着 A 是 B 的一部分；对于抽象，意味着 A 是 B 的一个特例，并继承了 B 的所有属性；对于投影，意味着 A 是 B 的一个视图。

3.1.3 需求获取技术

许多系统分析员在获取用户需求时，经常是根据自己对问题域的了解，向用户提出一些有关问题，并以用户的回答，不断了解一些新的问题，其中常常由于用户或分析员对应用问题的掌握程度，导致分析员与用户之间的思路不一致、不连贯等，因此一种好的需求获取技术，对规范需求获取活动，高效地获取高质量的需求定义，是十分重要的。

作为一种好的需求获取技术，应具有以下基本特征：

(1) 提供方便通信的设施，例如，易于理解的语言；

(2) 提供定义系统边界的方法；

(3) 提供支持抽象的基本机制，例如“划分”、“映射”等；

(4) 鼓励分析员使用问题空间的术语思考问题，编写文档；

(5) 为分析员提供多种可供选择的设计方案；

(6) 适应需求的变化。

至目前为止，Ivar Jacobson(1994)提出的用况(use case)方法基本符合以上特征，并得到了比较广泛的应用。

简单地说，一个用况表示了一个系统、一个子系统或其他语义实体所提供的“一块”高内聚的功能，这样的功能是通过该语义实体与一个或多个外部交互者(称为参与者)之间所交换的

消息序列，以及该语义实体所执行的一些动作予以表现的。例如：

对于一个特定的电话交换系统，处理一次用户拨打电话的用况，用户(即参与者)和系统之间所交换的消息序列如图 3.1 所示。

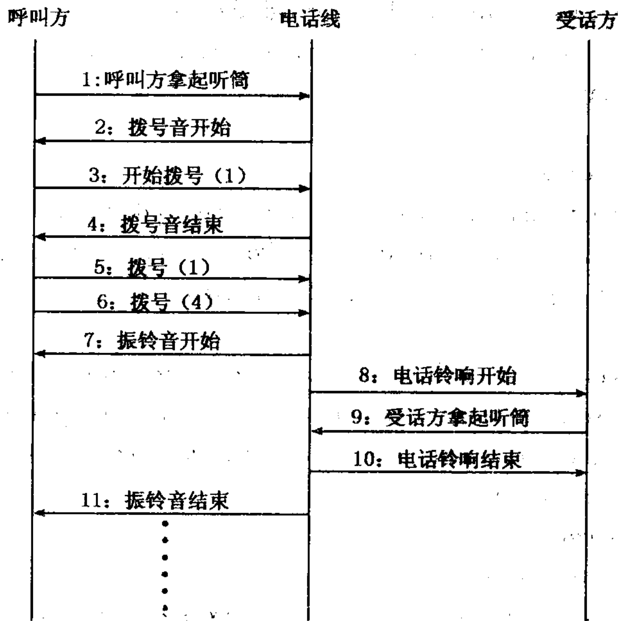


图 3.1　电话交换系统一个典型的用况

可见，一个用况捕获了参与交互的各方关于其行为的一个“约定”。通过这一约定，描述了该语义实体在不同条件下的行为对参与者一个要求的响应，以实现某一目的。不同的行为序列，依赖于所给出的特定要求以及与这些要求相关的条件。

通常，把用况表示为一个包含用况名的椭圆。如图 3.2 所示。

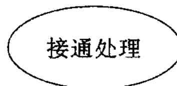


图 3.2　示例：用况表示

既然用况是各方(包含参与者)关于行为的一个“约定”，因此，为了便于讨论，还应给出它的正文形式，以表明该用况所包含的具体内容。对此，现行标准(例如，UML)还没有给出规范的形式。一般来说，在用况的正文描述中，应该给出以下一些方面的内容：

初始参与者：为了实现某一特定目标，初始与系统交互的参与者。

目标：该用况所能实现的功能目标。

层次：该用况在整个系统的用况图中的层次关系。

前置条件：交互序列执行的条件。

成功交互的主要场景：实现目标的主要交互序列。

扩展情况：实现某种特定交互的情况。

备注：给出使用这一用况的必要说明，例如：技术和数据的变化列表，响应时间，使用频率，相关的其他参与者等。

例如：

Buy Something

Primary Actor：Requestor.

Goal in Context：Requestor buys something through the system，get it. Does not include for it.

Scope：Business-the overall purchasing mechanism，electronic and non-electronic，as seen by the people in the company.

Level：Summary.

Stakeholder and interests：

Requestor：Want what he/she ordered，easy way to do that.

Company：Want to control spending but allow needed purchases.

Vendor：Want to get paid for any goods delivered.

Precondition：none.

Minimal Guarantees：Every order sent out has been approved by a valid authorizer. Order was tracked so that company can be billed only for valid goods received.

Success Guarantees：Requestor has goods，correct budget ready to be debited.

Trigger：Requestor decides to buy something.

Main Success Scenario：

1. Requestor：Initiate a request.
2. Approver：check money in budget，check price of goods，complete request for submission.
3. Buyer：check content of storage，find best vendor for goods.
4. Authorizer：validate Approver's signature.
5. Buyer：complete request for ordering，initiate PO with Vendor.
6. Vendor：deliver goods to Receiving，get receipt for delivery (out of scope of system under design).
7. Receiver：register delivery；send goods to Requestor.
8. Requestor：mark request delivered.

Extensions：

1a. Requestor does not know vendor or price：Leave those parts blank and continue.

1b. At any time prior to receiving goods，Requestor can change or cancel request：

Canceling it removes it from active processing (Delete from system?).

Reducing price leaves it intact in processing.

Raising price sends it back to Approver.

2a. Approver does not know vendor or price：Leave blank and let Buyer fill in or callback.

2b. Approver is not Requestor's manager：Still OK as long as Approver signs.

2c. Approver declines：Send back to Requestor for change or deletion.

3a. Buyer finds goods in storage：Send those up，reduce request by that amount，and carry on.

3b. Buyer fills in Vendor and price，which were missing：Request get resent to Approver.

4a. Authorizer declines Approver：Send back to Requestor and remove from active processing. (What does this mean?)

5a. Request involves multiple Vendor：Buyer generates multiple Pos.

5b. Buyer merges multiple requests: Same process, but mark PO with the requests being merged.

6a. Vendor does not deliver on time: System does alert of non-delivery.

7a. Partial delivery: Receiver mark partial delivery on PO and continues.

7b. Partial delivery of multiple-request PO : Receiver assigns quantities to requests and continues.

8a. Goods are incorrect or improper quality: Requestor refuses delivered goods. (What does this mean?)

8b. Requestor has quit the company: Buyer checks with Requestor's manager: either reassign Requestor or return goods and cancel request.

Technology and Data Variation List: None.

Priority: Various.

Releases: Several.

Response time: Various.

Frequency of Use: 3/day.

Channel to Priority Actor: Internet browser, mail, or equivalent.

Secondary Actor: Vendor.

注:本例子取自 Cockburn Highsmith, Writing Effective Use Cases, Addison-Wesley, 2001.

以上提及的参与者,实质上是定义了一组高内聚的角色。当一个用户与某一实体交互时,该用户可以扮演这一角色。可以认为,参与者针对与之通信的每一用况,分别扮演了一种不同的角色。

用况图呈现了一些参与者、一些用况以及他们之间的关系。其中的用况表示了一个语义实体(例如,一个系统,一个子系统)的功能,而且只有当外部的参与者与该实体进行交互时,该用况才显露其功能。可见,用况图是一幅由一组参与者、一组用况、可能的接口以及这些元素之间的关系所组成的图。在实际使用中,为了表示该实体的边界,可以用一个矩形把一些用况括起来。如图 3.3 所示。

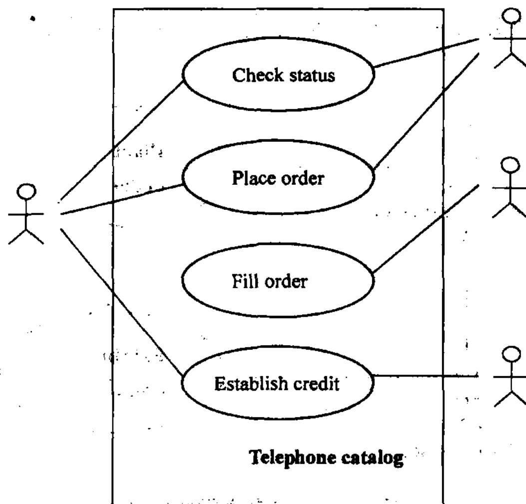


图 3.3 用况图

为了有效地组织用况图，控制其复杂性，在用况之间、参与者之间以及用况和参与者之间引入了以下一些关系：

(1) 用况之间的关系(见图 3.4)

① 包含：用况 A 到用况 B 的包含关系，表明用况 A 的一个实例在该关系定义的位置，包含了用况 B 所规约的行为。

② 扩展：用况 A 到用况 B 的扩展关系，表明用况 B 的实例是可以被用况 A 所规约的行为予以扩展(其中要根据该扩展中所指定的条件)，所扩展的行为被插到 B 中所定义的扩展点。

③ 泛化：用况 A 到用况 B 的泛化关系，表明 A 是 B 的特殊规约。

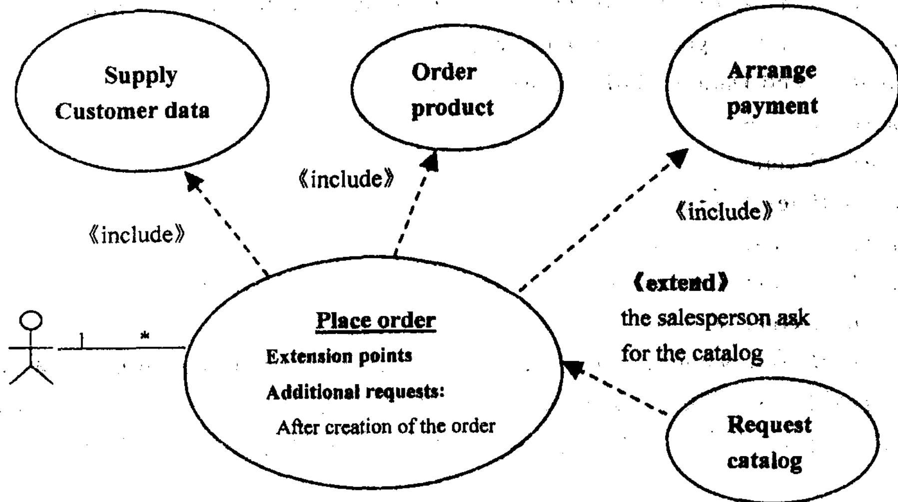


图 3.4 用况之间的关系

(2) 用况与参与者之间的关系

如上图所示，参与者与用况之间只有一种关联，即参与者实例与用况实例相互通信。

(3) 参与者之间的关系

参与者之间只有一种关系——泛化，即参与者 B 是参与者 A 的特殊规约。如图 3.5 所示。

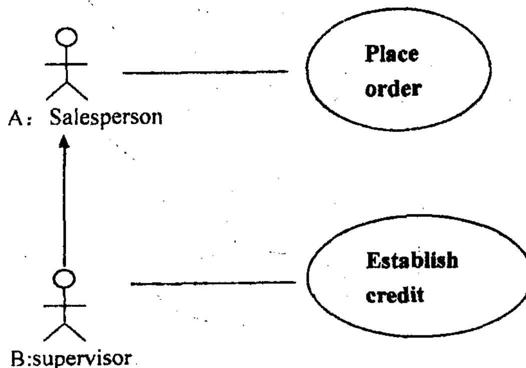


图 3.5 参与者之间的关系

由上可见：

① 用况用于定义一个实体或其他语义实体的行为，而没有揭示该实体的内部结构。

② 用况说明了一个实体为它的用户所提供的服务，即使用该实体的特定方法。由用户启动的服务是一个完整的序列。

③ 一个用况还包括其他类型的序列，例如可选择的序列、例外行为、错误处理等。完整的一组用况，说明了使用该实体的所有不同的方法。

④ 一个用况描述了用户和实体之间的交互，也描述了该实体所执行的响应，这些响应在该实体之外是可以观察到的。

⑤ 以实用的观点，用况可用于规约一个实体的需求和由该实体提供的功能。另外，用况还间接地指出了实体在用户面前的外征，即他们如何交互。

⑥ 一个用况可以使用操作和方法，以清楚的正文描述之。一个用况和参与者之间的交互，还可以用一个其他方式表示(例如协作图)，以规约这一用况的实体与其环境之间的交互。

⑦ 用况和参与者之间的关联，意指用况和参与者之间相互通信。一个用况在提供其服务时，可以与一个或多个参与者进行通信。

⑧ 用况之间的共性，可用两种不同的方法表示，即泛化和包含。

用况之间的一般关系意味着，子用况包含了在其父用况中所定义的所有属性、一系列行为、扩展点，以及父用况参与的所有关系。子用况还可以定义新的行为，也可以添加行为和指明继承父用况现存的行为。一个用况可以有多个父用况，一个用况可以是其他一些用况的父用况。

用况之间的包含关系，意味着在目标用况中定义的行为，被包含在其基用况的一个实例所执行的行为之中。当一个用况实例到达另一个用况的行为将予执行的地方，则执行被包含的用况所描述的所有行为，并且该实例依据其源用况继续。这意味着，尽管可以有多条路径通过所包含的用况(例如，由于条件语句)，但它们都必须以以下特定的方式结束：依据源用况，该用况实例可以继续。一个用况可以被包含在多个其他用况中，一个用况还可包含多个其他用况。所包含的用况可以不依赖其基用况。在这种意义上来说，被包含的用况表示了被封装的行为，它可容易地被不同的用况复用。另外，基用况不仅依赖于执行被包含行为的结果，而且还依赖于所包含的那个用况的结构，如属性和关联。

⑨ 扩展关系定义了一个用况可以使用在其他用况中定义的一些行为予以扩展。一个用况可以扩展多个用况，一个用况可以被多个用况予以扩展。

⑩ 每一参与者定义了一组高内聚的、与实体交互时用户可以扮演的角色。二个或多个参与者可以具有一些共性，即以同一方式与同一组用况通信。该共性可用泛化表达之，以建立一种公共角色的模型。

作为开发系统的第一步，整个系统的动态需求可以用用况图予以表达，即建立该系统的用况模型。

3.2 需求规约

经过需求获取阶段的工作，系统分析员已经比较全面地获取了用户的需求，并形成需求陈述，继之的主要工作是需求规约。需求规约的主要目的是对需求陈述进行分析，解决其中存在

的二义性和不一致性,并以一种系统化的形式准确地表达用户的需求,形成所谓的需求规格说明书。本章我们主要介绍结构化分析方法,在本书的第五章还将介绍面向对象方法。

结构化方法是由 Edward Yourdon, Tom DeMarco 等人于 70 年代中后期提出的一种系统化开发软件的方法,该方法基于模块化的思想,采用"自顶向下,逐步求精"的技术对系统进行划分。分解和抽象是该方法的两个基本手段。结构化方法是结构化分析、结构化设计和结构化编程的总称。

结构化分析方法最初把整个系统表示成一张环境总图,标出系统边界及所有的输入、输出;逐步对系统进行细化,每细化一次,就把一些复杂的功能分解成较简单的功能,并增加细节描述;继续这种细化,直到所有的功能都足够简单,不需要再继续细化为止。结构化方法由于其简单易懂、容易使用,且出现较早,所以获得了广泛的应用。

3.2.1 模型表示

结构化分析方法把任何软件系统都视作一个数据变换装置,它接受各种形式的输入,通过变换产生各种形式的输出。数据流图(dataflow diagram,有时简称 DFD)就是一种描述数据变换的图形工具,是结构化分析方法最普遍采用的表示手段,但数据流图并不是结构化分析模型的全部,数据字典和小说明为数据流图提供了补充,并用以验证图形表示的正确性、一致性和完整性,三者共同构成了被建系统的模型。

1. 数据流图

数据流图是一种描述数据变换的图形工具,系统接受输入的数据,经过一系列的变换(或称加工),最后输出结果数据。对于比较大型的软件系统,把整个系统画在一张图中,不仅显得凌乱,而且层次不清、难以理解,有必要把数据流图分成多层。下面我们通过一个简单的例子对数据流图的成分加以说明。

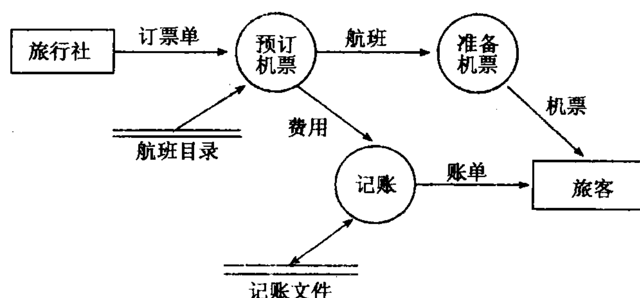


图 3.6 一个飞机票预订系统的数据流图

从图 3.6 我们可以看出,数据流图由以下四个基本成分组成。

(1) 加工(用圆圈表示)

加工是对数据进行处理的单元,它接受一定的输入数据,对其进行处理,并产生输出,如图 3.6 中的预订机票、准备机票、记账等都是加工的例子。

(2) 数据流(用箭头表示)

数据流表示数据和数据流向,如图 3.6 中的订票单、航班、费用、账单、机票等都是数据流的例子。

(3) 数据存储(用两条平行线表示)

数据存储用于表示信息的静态存储,如图 3.6 中的航班目录、记账文件等就是数据存储的例子。

(4) 数据源和数据潭(用矩形表示)

数据源和数据潭表示系统和环境的接口,是系统之外的实体,可以是人、物或其他软件系统。其中,数据源是数据流的起点,如图 3.6 中的“旅行社”就是数据源;数据潭是数据流的最终目的地,如图 3.6 中的“旅客”就是数据潭。

下面,简单地说明一下各成分的作用及命名时的一些注意问题。

(1) 加工

在分层的数据流图中,要对加工进行编号,以便于管理。加工也要选取适当的名字,以提高数据流图的可读性。下面给出一些加工的命名原则:

① 顶层的加工名就是软件项目的名字;

② 加工的名字最好使用动宾词组,如“计算费用”、“准备机票”、“检查合法性”、“产生工资单”等;也可以用主谓词组,如把上述的“计算费用”写成“费用计算”、“准备机票”写成“机票准备”等;

③ 不要使用意义空洞的动词作为加工名,如“计算”、“处理”、“加工”等,这样的名字没有给读者提供任何有意义的信息。

(2) 数据流

数据流表示数据和数据流向,通常由一组数据项组成,例如,数据流“订票单”由姓名、住址、电话、航班号、日期、始点、终点等数据项组成,而数据流“航班”则由航班号、日期、机型等数据项组成。

数据流可以从加工流向加工,在图 3.6 中,数据流“航班”是从加工“预订机票”流向加工“准备机票”的,数据流“费用”是从加工“预订机票”流向加工“记账”的;数据流也可以从数据源流向加工,或从加工流向数据潭,在图 3.6 中,数据流“订票单”是从数据源“旅行社”流向加工“预订机票”的,数据流“账单”是从加工“记账”流向数据潭“旅客”的;数据流还可以从加工流向数据存储,或从数据存储流向加工(一般流入或流出数据存储的数据流不需要标出名字,有数据存储的名字就可以了),在图 3.6 中,数据存储“航班目录”和加工“预订机票”之间的数据流,数据存储“记账文件”和加工“记账”之间的数据流就是这样的例子。

两个加工之间可以有多个数据流,这些数据流之间可以没有任何联系,数据流图也不表明它们的先后次序。

每个数据流要有一个合适的名字,一方面为了区别不同的数据流,另一方面能使人容易理解数据流的意义。下面给出一些数据流命名的方法和注意问题:

① 数据流的名字应使用名词,或名词词组;

② 数据流模型是现实系统的抽象,需求获取时已经对现实系统有了比较清楚的认识,这就为数据流命名提供了直接的参考依据,命名时应尽量使用现实系统中已有的名字;

③ 把现实环境中传递的一组数据(这组数据组成一个数据流)中最重要的那个数据的名字作为数据流的名字;

④ 不要把控制流作为数据流;

⑤ 不要使用意义空洞的名词作为数据流名,如“数据”、“信息”等,这样的名字没有给读者

提供任何有意义的信息。

(3) 数据存储

① 在分层数据流图中，数据存储一般局限在某一层或某几层，是和不同的抽象层次相关的；

② 数据存储的命名方法同数据流的命名方法相似。

(4) 数据源和数据潭

数据源和数据潭是表示系统和环境的接口，是系统之外的实体，命名时应符合环境的真实情况。

2. 数据字典

数据字典以一种准确的和无二义的方式定义所有被加工引用的数据流和数据存储，通常包括三类内容：数据流条目、数据存储条目和数据项条目。数据字典一般是按照一定的次序排列起来的，便于人们查阅。在数据字典中，经常使用一些常用的逻辑操作符(如表 3.1 所示)，以准确地表达被说明数据的结构。

表 3.1

操作符	含义描述
=	等价于(定义为)
+	与(顺序结构)
{}	重复(循环结构)
[|]	或(选择结构)
()	任选
m..n	界域

(1) 数据流条目

一个完整的数据流条目应该包括以下内容：

名称
描述
频率和数据量
数据结构

例如，本章后面“图书管理系统”例子中的“入库单”是一个数据流，对它的说明如下：

入库单 = 分类目录号 + 数量 + 书名 + 作者 + 内容摘要 + 价格 + 购书日期

(2) 数据存储条目

名称
描述
数据存储方式
关键码
频率和数据量
安全性要求
数据结构

例如，同样为本章后面例子中的“目录文件”是一个数据存储，对它的说明如下：

文件名：　目录文件

组成：　{分类目录号 + 书名 + 作者 + 内容摘要 + 价格 + 入库日期 + 总数 + 库存数

+{图书流水号}}

组织：　　按分类目录号的字母顺序排列

(3) 数据项条目

名称
描述
数据类型
取值范围及缺省值
精度
计量单位

数据项条目的格式详见本章后面“需求规格说明书”中第3.2.2节“数据项说明”。

3. 小说明

小说明是用来描述加工的，在一个分层的数据流图中，上层的加工通过细化分解为下层的更具体的加工。原则上，只要说明了最底层的基本加工，就可以理解上层的加工，对上层加工的说明是冗余的，所以小说明中可以只描述基本加工。当然，如果为了更便于理解，也可以在小说明中包括对上层加工的描述，总结和概括下层加工的功能。

小说明集中描述一个加工“做什么”，即加工逻辑，也包括其他一些和加工有关的信息，如执行条件、优先级、执行频率、出错处理等。加工逻辑是指用户对这个加工的逻辑要求，即这个加工的输入数据和输出数据的逻辑关系。小说明并不描述具体的加工过程。目前小说明一般是用自然语言、结构化自然语言、判定表和判定树等来描述。

小说明一般是按照加工的编号次序排列，当然也可以按照其他任何使用者觉得方便的次序排列，如按照加工名的字典顺序排列，特别是在计算机辅助支持下，小说明可依使用者的要求有多种排序方式。

(1) 结构化自然语言

结构化自然语言是介于形式语言和自然语言之间的一种语言。它虽然没有形式语言那样严格，但具有自然语言简单易懂的特点，同时又避免了自然语言结构松散的缺点。

结构化自然语言的语法通常分为内外两层，外层语法描述操作的控制结构，如顺序、选择、循环等，这些控制结构将加工中的各个操作连接起来。内层语法没有什么限制，一般使用自然语言描述。

(2) 判定表

判定表是用以描述加工的工具，通常用来描述一些不易用语言表达清楚或需要很大篇幅才能表达清楚的加工。例如，在飞机票预订系统中，在旅游旺季的7～9，12月份，如果订票超过20张，优惠票价的15%；20张以下，优惠5%；在旅游淡季的1～6，10，11月份，订票超过20张，优惠30%；20张以下，优惠20%。见表3.2的判定表。

表　3.2

旅游时间	7～9，12月		1～6，10，11月	
订票量	≤20	>20	≤20	>22
折扣量	5%	15%	20%	30%

如表3.3所示，判定表分为四个区。Ⅰ区内列出所有的条件类别，Ⅱ区内列出所有的条件

组合，Ⅲ区内列出所有的操作，Ⅳ区内列出在相应的组合条件下，某个操作是否执行或执行情况。在表3.2中，Ⅰ区的条件类别有两个：旅游时间和订票量，Ⅱ区内列出所有四种条件组合，Ⅲ区内只有一个操作，Ⅳ区标明在某种条件组合下操作的执行情况。

表 3.3

Ⅰ	条件类别	Ⅱ	条件组合
Ⅲ	操作	Ⅳ	操作执行

当描述的加工由一组操作组成，而且是否执行某些操作或操作的执行情况又取决于一组条件时，用判定表书写加工逻辑比较方便。表3.4是使用判定表的另一个例子。

表 3.4

考试总分	≥620	≥620	<620	<620
单科成绩	有满分	有不及格	有满分	有不及格
发升级通知书	Y	Y	N	N
发免修单科通知书	N	N	Y	N
发留级通知书	N	N	Y	Y
发重修单科通知书	N	Y	N	N

(3) 判定树

判定树用图形形式描述加工逻辑，其特点是结构清晰，易读易懂。判定表3.2可用图3.7的判定树等价表示。

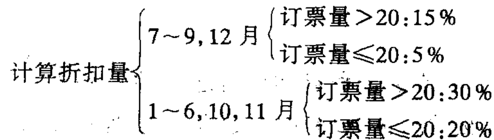


图3.7　同判定表3.2等价的判定树表示

判定表3.4可用图3.8的判定树等价表示。

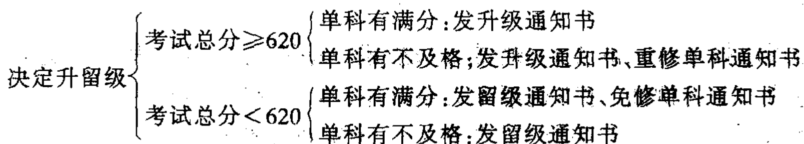


图3.8　同判定表3.4等价的判定树表示

加工逻辑可以用语言、表格、图形等多种形式来描述，也可以将它们结合使用。在保证描述的小说明清晰易懂的前提下，可以自由选择描述方法。

3.2.2　实施步骤

结构化分析从本质上说是一种运用抽象和分解技术，"自顶向下、逐步求精"的过程。前面

介绍了结构化分析的表示工具，下面讨论如何使用工具进行结构化分析。

1．确定系统边界，画出系统环境图

经过需求获取阶段的工作，分析员可以比较容易地确定系统边界，即系统与外界或环境的输入和输出，从而画出系统环境图，或称顶层数据流图。例如，上面的飞机票预订系统的顶层数据流图如图 3.9 所示。

图 3.9 一个飞机票预订系统的顶层数据流图

2．自顶向下，画出各层数据流图

有了顶层数据流图后，下面的工作就是自顶向下画出各层的数据流图。具体地说，就是对加工进行"逐层分解"，直到底层的加工足够简单，功能清晰易懂，不必再继续分解为止。

图 3.10 就是一个系统的分层数据流图。层次的编号是按顶层、0 层、1 层、2 层……的次序编排的。顶层图即系统环境图，标出了系统的边界；0 层图是顶层图中包含的惟一加工的细化，0 层图中的加工 2 和加工 3 又为 1 层图所细化。有时为方便起见，称这些图互为"父子"关系，即顶层图是 0 层图的"父图"，0 层图是 1 层包含的所有数据流图的"父图"；反过来，0 层图是顶层图的"子图"，1 层包含的所有数据流图是 0 层图的"子图"。除顶层图外，其他各层的数据流图都是某一父图的子图，这些数据流图统称为数据流子图或简称为子图。

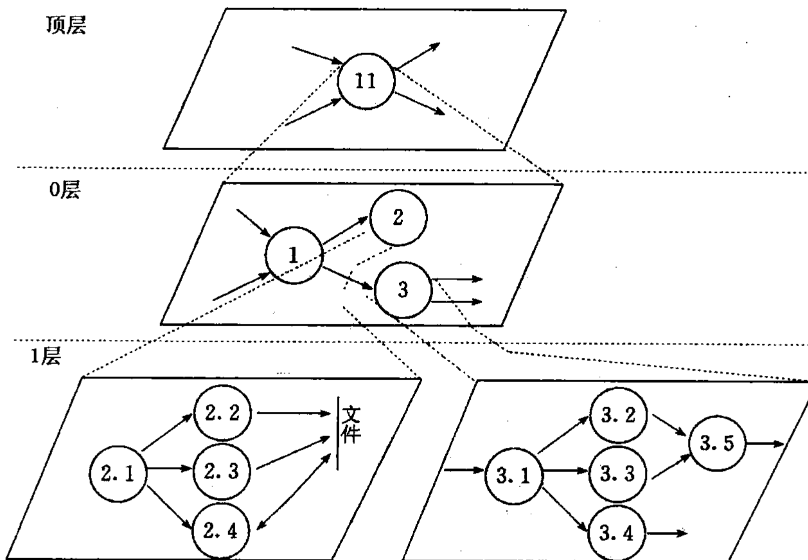


图 3.10 某个系统的分层数据流图

图 3.11 给出了一般系统的数据流图的总体结构。

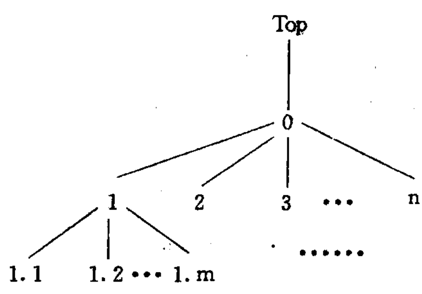


图 3.11 一般数据流图的总体结构

为了便于管理，需要给所有的数据流图和加工编号，并在整个系统中应是惟一的。本书是按下述规则为分层数据流图和图中的加工进行编号的：

(1) 顶层图不参与数据流图编号，顶层图中的惟一加工也不编号。

(2) 从 0 层图开始的所有子图和加工均需编号，子图的编号为分解的父图中相应加工的编号(如图 3.11 所示)。

(3) 加工的编号由相应的子图号、小数点、加工在子图中的顺序号组成。

0 层只有一张子图，按照规则，图中的加工编号依次为 0.1,0.2,0.3…，为了方便，通常去掉前面的 0 和小数点，则它们的编号依次为 1,2,3…；1 层子图的编号为 0 层细化的加工号，它们的编号就是 1,2,3…，相应子图中的加工编号由子图号、小数点、加工在子图中的顺序号组成，依次类推。

由“父图”生成“子图”的一般步骤是：

① 将“父图”的每一加工按其功能分解为若干个子加工——子功能。例如，图 3.9 所示的飞机票预订系统，可以将顶层的加工分解为三个子加工，即“预订机票”、“准备机票”和“记账”，如图 3.12 所示。

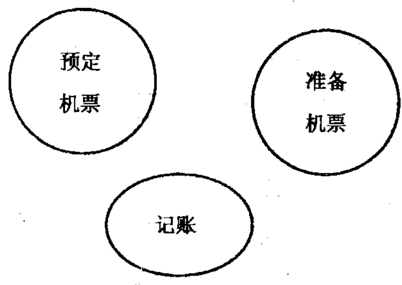


图 3.12 第一次功能分解

② 将“父图”的输入流和输出流“分派”到相关的子加工。例如，“父图”有输入流“订票单”，有输出流“机票”和“账单”。显然，应把“订票单”分派给加工“预定机票”，把“机票”分派给加工“准备机票”，把“账单”分派给加工“记账”。如图 3.13 所示。

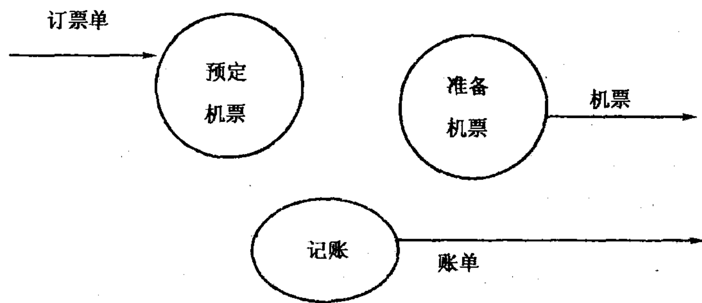


图 3.13 数据流的分派

③ 在各加工之间建立合理的关联，必要时引入数据存储，使之形成一个“有机的”整体。如图 3.14 所示。

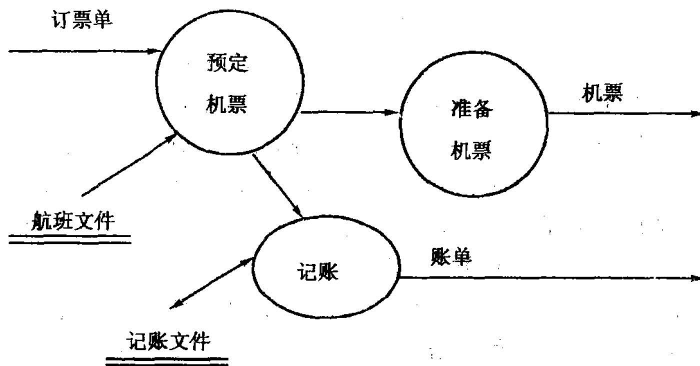


图 3.14 文件引入与精化

3. 定义数据字典

依据系统的数据流图，定义其中包含的所有数据流和数据存储。

4. 定义小说明

依据系统的数据流图，给出最底层数据流图中所包含的那些加工的说明。

5. 汇总前面各步骤的结果

以上讨论了建立系统模型的步骤，下面是一些在建立系统模型时应注意的事项。

(1) 模型平衡规则

① 数据流图中所有的图形元素必须根据它们的用法规则正确使用。例如，一个加工必须既有输入又有输出；数据流只能和加工相关，即从加工流向加工、数据源流向加工或加工流向数据潭。

② 每个数据流和数据存储都要在数据字典中有定义，数据字典将包括各层数据流图中数据元素的定义。

③ 数据字典中的定义使用合法的逻辑构造符号。

④ 数据流图中最底层的加工必须在小说明中有定义。

⑤ 父图和子图必须平衡，即父图中某加工的输入输出（数据流）和分解这个加工的子图的输入输出（数据流）必须完全一致，这种一致性不一定要求数据流的名称和个数一一对应，但它们在数据字典中的定义必须一致，数据流或数据项既不能多也不能少。

⑥ 小说明和数据流图的图形表示必须一致。例如，在小说明中，输入数据流必须说明其如何使用，输出数据流说明如何产生或选取，数据存储说明如何选取、使用或修改。

（2）控制复杂性的一些规则

① 上层数据流可以打包（打包的数据流作一特殊标记），上、下层数据流的对应关系用数据字典描述（编号对应），同层的数据流也可编号对应，避免形成复杂的连线；只有一点限制：包内流的性质（输入、输出、输入输出）必须一致。

② 为了便于人的理解，把一幅图中的图元个数控制在 7±2 以内（Miller，1956）。

③ 检查同每个加工相关的数据流，并寻找是否有其他可降低界面复杂性的划分方法（有时一个加工有太多的输入输出数据流与同一层的其他加工或抽象层次有关）。

④ 分析数据内容，确定是否所有的输入信息都用于产生输出信息；相应地，由一个加工产生的所有信息是否都能由进入该加工的信息导出。

根据以上关于结构化分析方法的介绍可知，该方法看待客观存在的信息处理系统的基本观点是：一个信息处理系统的功能表现为信息在不断的流动中并经过一系列的变换。因此，为了描述这样的系统，结构化方法给出了一组帮助系统分析员产生功能规约的原理和技术。其中的原理主要包括功能分解与抽象、自顶向下逐步求精以及信息隐蔽等；基于这些原理给出了一组既能表达系统功能，又能控制信息组织复杂性的概念、表示以及步骤。

具体地说，该方法采用数据流图、数据字典、结构化语言、判定表和判定树等表示系统功能模型。① 数据流图以图形的方式表达目标系统中信息的变换和传递，其中有五个基本要素：数据流、加工、数据存储、数据源和数据潭。数据流是具有名字的结构化数据及其流向；加工是对数据和数据流的变换；数据存储是可访问的静态结构化数据；数据源和数据潭分别表明数据处理过程的数据来源和数据去向。② 数据字典对数据流图中出现的数据元素给出其逻辑定义，用以表示该数据的结构。③ 在分层的数据流图中，最低层的数据加工可采用结构化语言、判定表和判定树等描述该加工内部过程的控制结构。

结构化分析的基本步骤是：① 通过对现实系统的了解和分析，或基于需求陈述，建立该系统的数据流图；② 基于得到的数据流图，建立该系统的数据字典；③ 基于得到的数据流图，对最低层的加工给出其控制结构描述；④ 依据需求陈述，建立人机接口和其他性能描述；⑤ 通过分析和验证，建立系统完整的需求规约。

3.3 需求验证

需求分析阶段的工作结果是开发软件系统的重要基础。大量统计数字表明，软件系统中15%的错误起源于错误的需求。为了确保软件开发成功，提高质量和降低费用，一旦对目标系统提出一组需求之后，就必须严格验证这些需求的正确性。一般说来，应该从以下几个方面的特征对软件需求规格说明书（Software Requirements Specification，以下简称 SRS）加以验证。

1．正确性

正确性指的是 SRS 中陈述的每个需求都表达了将要构造的系统的某种要求。目前尚不

存在有效的技术来保证这个质量，因为它完全依赖于当前的应用系统，例如，如果软件必须在5秒钟内对所有的按键事件做出响应，而SRS中陈述“软件应在10秒钟内对所有的按键事件做出响应”，则该需求描述是不正确的。图3.15有助于以可视化的方式解释正确性的含义。左边的圆圈表示用户的实际需求，右边的圆圈表示SRS中陈述的需求。如果SRS是正确的，则区域C为空，即SRS中的每个需求都表达了未来系统的某种要求。

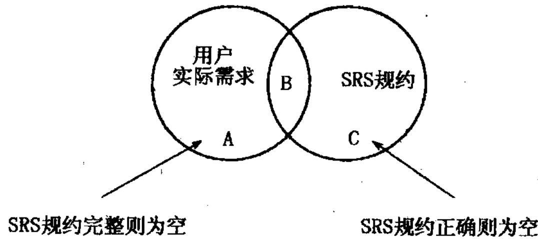


图3.15 用户实际需求——SRS规约

2. 无二义性

无二义性指的是SRS中陈述的每个需求都有惟一的解释。设想从SRS中抽出一句话，把它交给10个人理解，如果存在一种以上的解释，那么这句话就可能是有二义性(或多义性)的。为克服SRS中具有的二义性，可以把一个SRS中所有具有二义性的术语统一收集在一起，并注明在不同情况下各个术语的准确含义。然而，同二义性有关的问题远非一个简单的术语汇编所能解决的。特别地，自然语言具有不可避免的二义性，因此使用自然语言书写SRS是容易导致二义性的。

3. 完整性

如果一个SRS具有以下三个特性，那么它是完整的：

(1) 期待未来系统所做的任何事情都包括在SRS的陈述中，完整性在图3.15中就意味着区域A为空。注意，如果一个SRS既是完整的又是正确的，那么区域A和C同时为空，图3.15中的两个圆是重合的。完整性是所有属性中最难以保证的，原因是不完整意味着有一些东西不在SRS中，而这很难通过检查现有的材料发现其中不存在的东西，惟一可能发现这种疏忽或遗漏的是那些需要软件解决他们的问题的用户。诊断不完整性的一个有效技术就是做原型。此外，原型还有助于对需求的其他特征进行验证。

(2) SRS中应包括未来软件系统在所有可能情况下对所有可能的输入的响应，所有可能的输入包括有效输入和无效输入。这就意味着对于SRS中提到的每个系统输入，都应说明合适的系统输出将会是什么。值得注意的是，系统输出可能不仅仅是输入的函数，还可能是当前系统状态的函数。例如，在一个电话转接系统中，当检测到用户拨号9时，系统响应是当前状态的函数，而系统的状态依次又是用户前面动作的函数，因此，如果用户电话机的话柄没有拿起，就没有系统输出产生(即，输入9被忽略了)。如果用户刚开始拨号，则系统输出可能是另外一种拨号音；如果用户已经开始拨一个电话号码，9被收集起来作为电话号码的一部分。换句话说，SRS必须建立从输入域(I)和状态域(S)的笛卡尔乘积到输出域(O)和状态域的笛卡尔乘积的完整映射，即

$$SRS: I \times S \rightarrow O \times S$$

(3) SRS中没有任何内容被标为“待定”。只要可能就应尽量避免在SRS中插入“待定”这

个词;当包含待定时,应同时附加上谁应负责对该内容的最后确定,以及在哪个日期以前完成。这种方法就保证了待定不被随意当做无限拖延完成 SRS 的借口,就像待定意味着"明天做",而明天当然永远也不会来。通过在 SRS 中包含负有责任的人员的名字和完成日期,我们保证了待定会在将来某一时刻消失。

4. 可验证性

可验证性指的是 SRS 中陈述的每个需求都是可验证的。一个需求是可验证的,当且仅当存在一个有限代价的过程,人工或机器可以检查实际的软件产品是否符合用户的需求。首先,任何二义性必然导致不可验证性,例如,关于"产品应有易于使用的用户界面"这一陈述是存在二义性的,因为"易于使用"的观点对于不同的人可以有很不相同的理解,所以最终也是不可验证的;第二,使用不可度量的量,如"通常"或"时常",意味着不存在一个有限的测试过程,也就意味着是不可验证的,例如,关于"当按下按钮时,系统通常应亮起红灯"这一陈述就是不可验证的,因为当你验证最终系统是否符合用户需求时,连续按下按钮 1000 次,如果红灯亮了 600 次,你可能就试图声明测试成功,然而,当你再按下按钮时,有可能红灯再也没有亮,换句话说,测试"通常"的惟一方法是按下按钮无数次;第三,任何等同于停机问题的需求是不能被验证的(Turing,1936),例如,可以证明"程序将不会进入一个无限循环"等同于停机问题,因而是不可验证的。

5. 一致性

一致性指的是:① SRS 中陈述的需求同以前的文档,如系统需求规格说明书,没有发生冲突;② SRS 中陈述的各个需求之间不发生冲突。

6. 可理解性

为了使一个 SRS 减少二义性、增强可验证性、完整性和一致性,我们应努力追求高度形式化的表示法。不幸的是,这样的表示法常常缺乏另一个重要的特性:非计算机专家对 SRS 的理解。在许多情况下,SRS 的主要读者是顾客或用户,他们通常是应用领域的专家而不是计算机专家。也许达到该目标的一个途径是使用形式化的表示法,并借助工具把形式化的 SRS 自动转换为等价的易于理解的内容,这就是 Balzer 及其合作者在 GIST 项目中采用的方法(1982)。如果在形式化表示和非形式化表示之间存在一个完整的无二义的映射,那么非形式化的表示将满足所有要求的属性,包括被非计算机专家所理解。

以上主要讨论了 SRS 内容方面的属性,下面集中讨论关于 SRS 格式和风格方面的属性。

7. 可修改性

可修改性指的是 SRS 的结构和风格使任何对需求的必要的修改都易于完整和一致地进行。可修改性意味着存在一个有关 SRS 内容的列表、索引和交叉引用,如果以后必须对一个需求进行修改的话,我们可以方便地检查并定位 SRS 中所有必须加以修改的部分。例如,假设我们想把电话转接系统中对用户拨号的最小响应时间从 5 秒改为 3 秒,我们将查看"用户拨号"下面的索引,定位文档中所有对"用户拨号"的引用,以做必要的修改。改进 SRS 可读性的一个技术是在文档中不同的地方重复特定的需求,SRS 的这个属性称为冗余。例如,在描述自动用户交换机的外部接口时,我们必须定义用户和电话机之间的交互,因此,当描述市内通话的外部表现时,SRS 可能会作如下陈述:

用户首先找到一个空闲的电话机,拿起听筒,系统将发出拨号音,然后用户开始拨打受话方的 8 位电话号码……。

当描述长途通话的外部表现时,SRS 可能会作如下陈述:

用户首先找到一个空闲的电话机，拿起听筒，系统将发出拨号音，然后用户先拨 0，接着拨打受话方的 10 位电话号码……。

注意上面的描述，文档对拨打电话的前三个步骤的重复增加了可读性。然而，可读性的增加潜在地降低了可修改性，因为当以后只对其中某一处的改变将导致 SRS 的不一致。为使冗余成为可接受的，索引表或交叉引用表对多次出现的需求的定位是基本的。

8. 可被跟踪性

可被跟踪性指的是 SRS 中的每个需求的出处都是清楚的，这意味着 SRS 中包含对前期支持文档的引用表，如图 3.16 所示，让我们假设 SRS 中包含如下的需求陈述：

系统应对任何一次 X 请求在 20 秒内响应。

假设软件已被实际构造出来，当接受最终测试时，测得的响应时间是 60 秒，那么对这个问题有两种解决方案：①对软件进行重新设计或编码，以使其效率更高；或者，②把需求从 20 秒改为 60 秒。如果 SRS 在该需求陈述处没有提供任何引用，以表示 20 秒是从其他什么地方来的，而只是一个随意选定的时间约束，我们就可以选择第二种方法(并且该方法可能是一个相当满意的解决)。然而，如果该应用是一个实时病房监控系统，20 秒钟的响应时间是从前期的文档引用来的，在该实时病房监控系统安装的特定医院环境中，根据当前护士和病人的比例，每个护士每分钟至少需要对该系统查询 3 次，以保证任何病人的每次紧急情况都不会被遗漏。在这种情况下，可被跟踪性就要求在 SRS 中对时间约束的需求处插入对前期文档的引用，那么在考虑第二种解决方案时，当检查了前期的文档之后立即就可被否定。

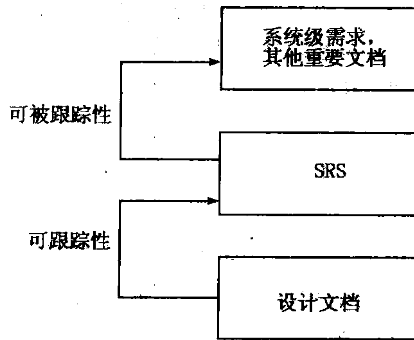


图 3.16 可跟踪性及可被跟踪性

9. 可跟踪性

为了设计或测试软件的任何成分，有必要清楚地知道该软件成分满足或部分满足哪个需求，同样地，当测试该软件系统时，有必要理解该测试针对哪些需求的有效性。可跟踪性指的是 SRS 的书写方式有助于对其中陈述的每个需求进行引用，如图 3.16 所示。存在许多达到该目标的技术和方法：

(1) 给每个段落按层次编号，以后当指明某个需求时，通过指出段落编号和句子在段落中的序号即可，如需求 2.3.2.4S3 指的是在段落 2.3.2.4 中的第 3 个句子，这是一种非常有效的方法。

(2) 给每个段落按层次编号，在任何段落中都不包括多于一个需求，以后当指明某个需求时，只需给出段落编号即可。这种方法是可行的，但易于导致难以阅读的文档。

(3) 在 SRS 中出现的每个需求都编以惟一的序号。

(4) 使用一种指示需求的惯例表示法,即总是在包含一个需求的句子中使用"shall",然后使用简单的 shall 提取工具,对所有带 shall 的句子进行提取并惟一编码。

10. 设计无关性

设计无关性指的是 SRS 不暗示特定的软件结构或算法。每个需求都限制了设计的选择,例如,如果你想要一个在某建筑物中上下运送旅客的系统,就排除了电话转接系统软件作为一种可能的设计。SRS 中的任何需求都不应把设计限制在惟一的选择上,当然,SRS 的设计无关性也存在一些例外,特别是对于一些特殊的应用,构造一个新系统的主要动机是为了使用特定的体系结构或算法,例如,一个电梯控制系统的 SRS 必然包含调度算法;一个集成电路布局程序的 SRS 无疑会包括布局和布线算法。

11. 注释

注释向开发机构提供了每个需求是否重要的指导意见。有时一个软件开发机构会花费过多的时间以满足某个特定的需求,后来发现客户宁愿得到一个需求没有完全满足的按时交货的产品,而不愿 6 个月后得到一个需求完全满足的产品。如果每个需求的相对重要性一开始就确定了,那么上面的情况显然是可以避免的。这样做的一种方法是给 SRS 中的每个个别的需求标上 E(Essential, 基本的), D(Desirable, 希望的)或 O(Optional, 任选的)。

一个 SRS 达到以上的所有目标几乎是不可能的,例如,当我们试图排除不一致性和二义性时(通常通过减少 SRS 中的自然语言), SRS 对于非计算机专家的用户将变得难以理解;当我们试图达到绝对的完整性时, SRS 和其他文档将变得十分庞大和难以阅读;如果我们通过排除冗余以增加可修改性, SRS 将变得难以琢磨和具有二义性。

我们可以得出的惟一结论是:不存在十全十美的 SRS!

3.4 需求分析文档

需求分析规格说明书是需求分析阶段产生的一份最重要的文档,它以一种一致的、无二义的方式准确地表达用户的需求。需求规格说明书主要起以下三方面的作用:

(1) 作为软件开发机构和用户之间一份事实上的技术合同书;

(2) 作为软件开发机构下一步进行设计和编码的基础;

(3) 作为测试和验收目标系统的依据。

在本节中我们将给出一份典型的需求规格说明书的样例,供大家在实际工作中参考。

此外,在需求分析阶段一般还会产生另外两个文档——初步测试计划和用户系统描述。

在系统开发早期设计一个软件测试计划是十分必要的。大量的统计数字表明,在系统开发早期发现并修改一个错误的代价往往很低,越到系统开发的后期,改正同样错误所花费的代价越高。举个例子,假设在需求分析阶段检测并改正一个错误的代价为 1 个单位,那么到了软件测试阶段检测并改正同样的错误所花费的代价,据典型的保守数字为 10 个单位,而到软件发布后的代价就可能高达 100 个单位。所以,尽可能地在系统开发的早期进行软件测试,就可以较小的代价检测出需求规格说明书中不可避免的错误。这个初步的测试计划应包括对未来系统中的哪些功能和性能指标进行测试,以及达到何种要求。在后阶段的软件开发中,对这个测试计划要不断地修正和完善,并成为相应阶段的文档的部分。

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

下面给出需求规格说明书的一种结构。

××××××系统需求规格说明书

1. 引言

1.1 编写目的

说明编写本需求分析规格说明书的目的。

1.2 背景说明

(1) 给出待开发的软件产品的名称；

(2) 说明本项目的提出者、开发者及用户；

(3) 说明该软件产品将做什么，如有必要，说明不做什么。

1.3 术语定义

列出本文档中所用的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出本文档中所引用的全部资料，包括标题、文档编号、版本号、出版日期及出版单位等，必要时注明资料来源。

2. 概述

2.1 功能概述

叙述待开发软件产品将完成的主要功能，并用方框图来表示各功能及其相互关系。

2.2 约束

叙述对系统设计产生影响的限制条件，并对下一节中所述的某些特殊需求提供理由，如管理模式、硬件限制、与其他应用的接口、安全保密的考虑等。

3. 数据流图与数据字典

3.1 数据流图

3.1.1 数据流图1

(1) 画出该数据流图

(2) 加工说明

(a) 编号

(b) 加工名

(c) 输入流

(d) 输出流

(e) 加工逻辑

(3) 数据流说明

3.1.2　数据流图 2

……

3.2　数据字典

3.2.1　文件说明

说明文件的成分及组织方式。

3.2.2　数据项说明

以表格的形式说明每一数据项，格式如下表所示：

名　称	类　型	含　义	度量单位	有效范围	精　度

4．接口

4.1　用户接口

说明人机界面的需求，包括：

(1) 屏幕格式；

(2) 报表或菜单的页面打印格式及内容；

(3) 可用的功能键及鼠标。

4.2　硬件接口

说明该软件产品与硬件之间各接口的逻辑特点及运行该软件的硬件设备特征。

4.3　软件接口

说明该软件产品与其他软件之间接口，对于每个需要的软件产品，应提供：

(1) 名称；

(2) 规格说明；

(3) 版本号。

5．性能需求

5.1　精度

逐项说明对各项输入数据和输出数据达到的精度，包括传输中的精度要求。

5.2　时间特征

定量地说明本软件的时间特征，如响应时间、更新处理时间、数据传输、转换时间、计算时间等。

5.3　灵活性

说明本软件所具有的灵活性，即当用户需求(如对操作方式、运行环境、结果精度、时间特性等的要求)有某些变化时，本软件的适应能力。

6．属性

6.1　可使用性

规定某些需求，如检查点、恢复方法和重启动性，以确保软件可使用。

6.2　保密性

规定保护软件的要素。

6.3　可维护性

规定确保软件是可维护的需求，如模块耦合矩阵。

6.4 可移植性

规定用户程序、用户接口的兼容方面的约束。

7. 其他需求

7.1 数据库

说明作为产品的一部分来开发的数据库的需求。如：

(1) 使用的频率；

(2) 访问的能力；

(3) 数据元素和文件描述；

(4) 数据元素、记录和文件的关系；

(5) 静态和动态组织；

(6) 数据保留要求。

7.2 操作

列出用户要求的正常及特殊的操作，如：

(1) 在用户组织中各种方式的操作；

(2) 后援和恢复操作。

7.3 故障及处理

列出可能发生的软件和硬件故障，并指出这些故障对各项性能指标所产生的影响及对故障的处理要求。

3.5 实例研究

1. 项目说明

图书管理系统旨在用计算机对图书进行管理，包括图书的购入、借阅、归还以及注销。管理人员可以查询某位读者、某种图书的借阅情况，还可以对当前图书借阅情况进行一些统计，给出统计表格，以便全面掌握图书的流通情况。鉴于篇幅所限，我们这里给出一个非常简化的图书管理系统的例子，旨在说明进行数据流分析的方法。

本系统针对图书主要进行四方面的管理：购入新书、读者借书、读者还书以及图书注销。①购入新书时需要为该书编制图书卡片，包括分类目录号、流水号(要保证每本书都有惟一的流水号，即使同类图书也是如此)、书名、作者、内容摘要、价格和购书日期等信息，写入图书目录文件中。②读者借书时填写借书单，包括读者号、欲借图书分类目录号，系统首先检查该读者号是否有效，若无效，则拒绝借书；否则进一步检查该读者所借图书是否超过最大限制数(此处我们假设每位读者同时只能借阅不超过五本书)，若已达到最大限制数(此处为五本)，则拒绝借书；否则读者可以借出该书，登记图书分类目录号、读者号和借阅日期等，写回到借书文件中去。③读者还书时，根据图书流水号，从借书文件中读出和该图书相关的借阅记录，表明还书日期，再写回借书文件中；如果图书逾期未还，则处以相应罚款。④在某些情况下，需要对图书馆的图书进行清理工作，对一些过时或无继续保留价值的图书要注销，这时可以从图书文件里删除相关记录。⑤咨询要求分为查询某位读者、某种图书和全局图书情况三种。

2. 数据流图

经过分析，得出的数据流图如图 3.17，3.18，3.19 和 3.20 所示。

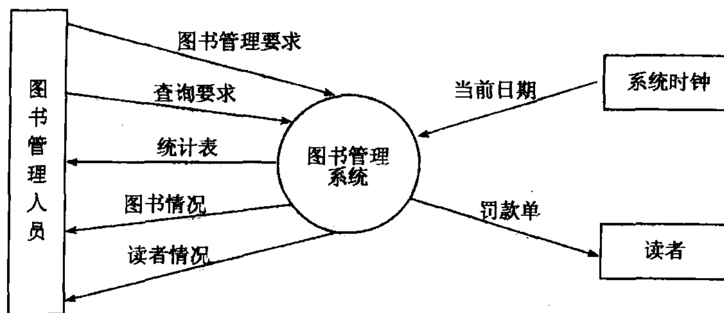


图 3.17 顶层数据流图

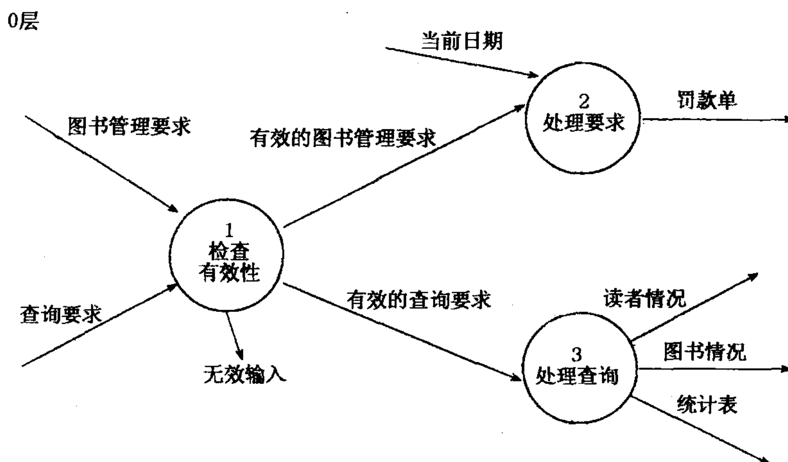


图 3.18 0 层数据流图

3. 数据字典

(1) 数据流条目

图书管理要求 =〔入库单|借书单|还书单|注销单〕

入库单 = 分类目录号 + 数量 + 书名 + 作者 + 内容摘要 + 价格 + 购书日期

借书单 = 读者号 + 分类目录号 + 借阅日期

借书记录 = 读者号 + 分类目录号 + 图书流水号 + 借阅日期

还书单 = 图书流水号 + 还书日期

罚款单 = 逾期天数 + 罚款金额

注销单 = 图书流水号

查询要求 =〔读者情况|图书情况|统计要求〕

读者情况 = 读者号 + 姓名 + 所在单位 + {借书情况}

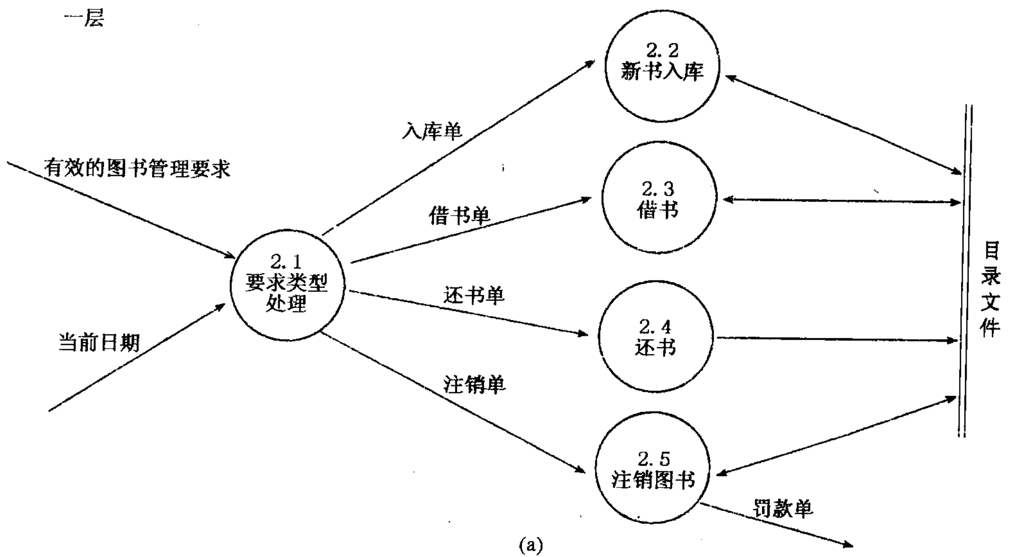
借书情况 = 书名 + 分类目录号 + 图书流水号 + 借阅日期

图书情况 = 书名 + 作者 + 分类目录号 + 总数 + 库存数

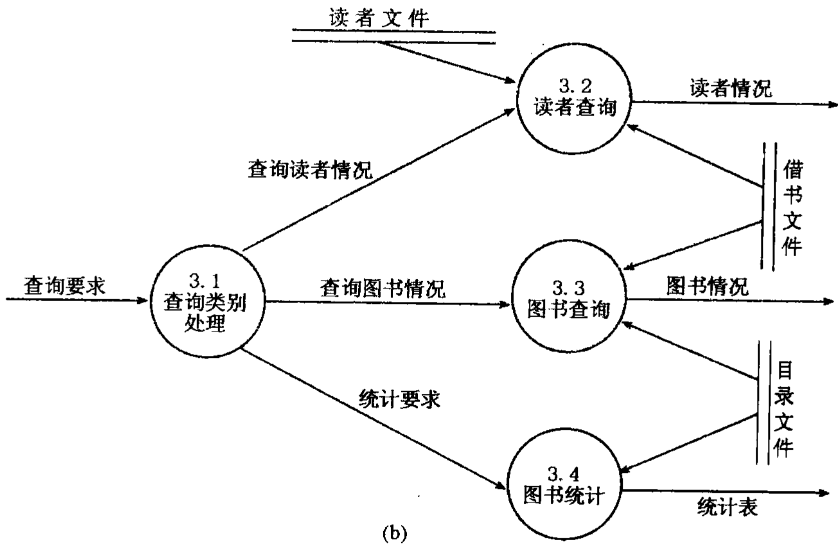
统计表 = {图书情况}

……

一层



(a)



(b)

图 3.19 一层数据流图

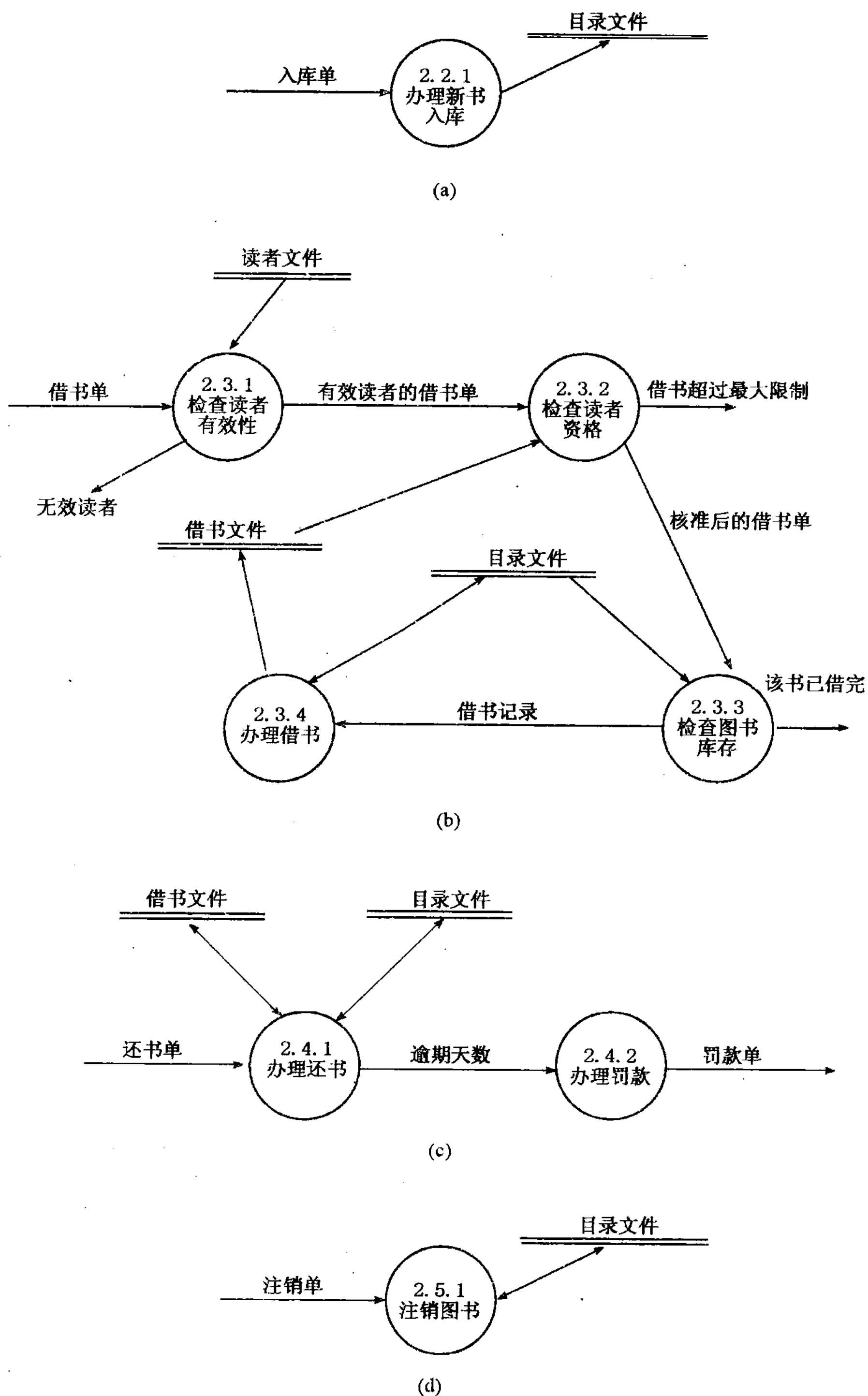


图 3.20　二层数据流图

(2) 文件条目

文件名:读者文件
组成:{读者号+姓名+所在单位}
组织:按读者号递增顺序排列

文件名:目录文件
组成:{分类目录号+书名+作者+内容摘要+价格+入库日期+总数+库存数+{图书流水号}}
组织:按分类目录号的字母顺序排列

文件名:借书文件
组成:(借书记录+还书日期)
组织:按借阅日期顺序排列

4.小说明

小说明只描述最底层的基本加工。

加工编号:1
加工名:检查有效性
输入流:图书管理要求,查询要求
输出流:有效的图书管理要求,有效的查询要求
加工逻辑:检查输入要求的有效性

加工编号:2.1
加工名:要求类型处理
输入流:图书管理要求,当前日期
输出流:入库单,借书单,还书单,注销单
加工逻辑:根据图书管理要求的类型选择
　　case 1:新书入库,输出入库单
　　case 2:借书,输出借书单
　　case 3:还书,输出还书单
　　case 4:注销图书,输出注销单

加工编号:3.1
加工名:查询类别处理
输入流:查询要求
输出流:查询读者情况,查询图书情况,统计要求
加工逻辑:根据查询类别选择
　　case 1:查询读者情况
　　case 2:查询图书情况
　　case 3:统计要求

加工编号:3.2

加工名:读者查询
输入流:查询读者情况,读者文件,借书文件
输出流:读者情况
加工逻辑:根据查询读者的情况从读者文件中读出读者记录,并从借书文件中读出该读者的借书记录,综合输出该读者的借阅情况

加工编号:3.3
加工名:图书查询
输入流:查询图书情况,借书文件,目录文件
输出流:图书情况
加工逻辑:根据查询图书的情况从目录文件中读出该书信息,并从借书文件中读出该书的借阅记录,综合输出该书的借阅情况

加工编号:3.4
加工名:图书统计
输入流:统计要求,目录文件
输出流:统计表
加工逻辑:根据统计要求从目录文件中读出所有图书的记录,输出统计表

加工编号:2.2.1
加工名:办理新书入库
输入流:入库单
输出流:目录文件
加工逻辑:输入填写好的入库单,并写入目录文件

加工编号:2.3.1
加工名:检查读者有效性
输入流:借书单,读者文件
输出流:有效读者的借书单
加工逻辑:根据借书单上的读者号和读者文件的内容,检查该读者是否为合法读者

加工编号:2.3.2
加工名:检查读者资格
输入流:有效读者的借书单,借书文件
输出流:核准后的借书单
加工逻辑:从借书文件中读出该读者的当前借书情况,检查他所借图书是否已超过最大限制

加工编号:2.3.3
加工名:检查图书库存
输入流:核准后的借书单,目录文件
输出流:借书记录

加工逻辑：根据借书单上的分类目录号和目录文件的内容，检查该书是否还有库存：若有，则填写借书记录

加工编号：2.3.4
加工名：办理借书
输入流：借书记录，目录文件
输出流：目录文件，借书文件
加工逻辑：根据借书记录的内容，对目录文件中该书的库存数量减 1，同时写入借书文件

加工编号：2.4.1
加工名：办理还书
输入流：还书单，借书文件，目录文件
输出流：借书文件，目录文件，逾期天数
加工逻辑：根据还书单，对目录文件中该书的库存数量加 1，同时把借书文件中相应记录置为无效；根据借阅日期和当前日期计算该图书是否已过期，并输出逾期天数

加工编号：2.4.2
加工名：办理罚款
输入流：逾期天数
输出流：罚款单
加工逻辑：根据图书过期天数，开具罚款单

加工编号：2.5.1
加工名：注销图书
输入流：注销单，目录文件
输出流：目录文件
加工逻辑：根据图书注销单，从目录文件中删除相应记录

习 题 三

1. 解释以下术语：
(1) 需求分析；
(2) 用况；
(3) 数据流图。
2. 举例说明用况之间的三种关系。
3. 简单回答以下问题：
(1) 用况如何显露其功能？
(2) 以结构化分析方法建立的系统模型由哪些部分组成？每一部分的基本作用是什么？
(3) 结构化分析方法为了表达系统模型，给出了几个基本概念？它们是如何表示的？
(4) 为什么说只引入操作符“+”，“|”，“{}”，在表达数据结构上是完备的？
(5) 在画出每一个加工时，应注意那些问题？
4. 举例说明结构化方法给出的控制复杂性机制。

5. 简述系统需求规格说明书的基本结构。

6. 试分析结构化方法在建造系统模型中存在的问题。

7. 针对自己给出的问题陈述，建立该问题的用况模型。

8. 针对以下给出的问题陈述：

在要建立的某商场简化的管理信息系统中，

(1) 库房管理员负责

① 输入、修改、删除入/出库商品信息(品名，编号，生产厂家，数量，单价，入/出库日期)，

② 打印库房商品库存清单(品名，编号，库存量，库存金额)；

(2) 销售员负责

① 登入商品销售信息(品名，编号，销售量，单价)，

② 输入、修改、删除入前台的商品信息(品名，编号，生产厂家，数量，单价，入库日期)，

③ 打印前台商品库存清单(品名，编号，库存量，库存金额)；

(3) 部门经理随机查询或打印统计报表

① 日、月销售金额，

② 日、月库存情况(品名，编号，库存量，库存金额)，

③ 日、月前台库存情况(品名，编号，库存量，库存金额)，

④ 打印年销售金额统计表、年库存误差统计表。

请用结构化方法，建立该系统的模型。

第四章　结构化设计

需求分析的主要任务是定义问题，即确定系统必须“做什么”。该阶段主要包括需求获取、需求规约和需求验证，最终形成该系统的软件需求规格说明书，其中主要成分是系统模型。因此，需求分析属于问题域的范畴。

设计阶段的主要任务是在需求分析的基础上，针对给定的问题，给出该问题的软件解决方案，即确定“怎么做”的问题。

软件设计可以采用多种方法，如结构化设计方法、面向数据结构的设计方法、面向对象的设计方法等，本章我们主要讨论结构化设计方法。

结构化设计又进一步分为总体设计和详细设计。总体设计的结果是被建系统的模块结构，即系统实现所需要的软件模块-系统中可标识的软件成分，以及这些模块之间的调用关系。但在这时，每一模块均是一个“黑盒子”，其详细描述是详细设计的任务。

4.1　总体设计的目标及其表示

总体设计阶段的主要任务是把系统的功能需求分配给软件结构，形成软件模块结构图，如图 4.1 所示。

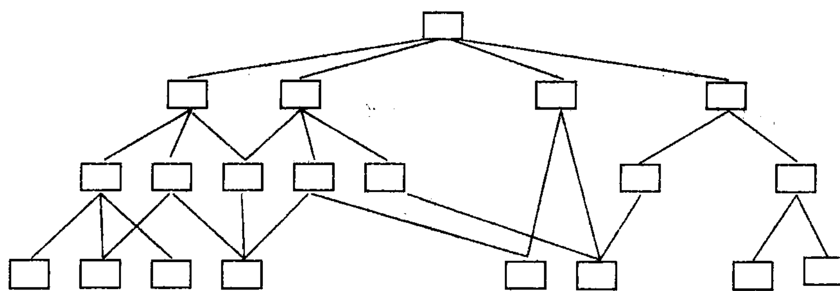


图 4.1　软件的模块结构图

在结构图中，矩形表示功能单元，称为“模块”；连接上下层模块的线段表示它们之间的调用关系。处于较高层次的是控制（或管理）模块，它们的功能相对复杂而且抽象；处于较低层次的是从属模块，它们的功能相对简单而且具体。依据控制模块的内部逻辑，一个控制模块可以调用一个或多个下属模块；同时，一个下属模块也可以被多个控制模块所调用，即尽可能地复用已经设计出的低层模块。

在总体设计阶段，每一模块通过外部特征予以标识，即给出每一模块的名字、输入和输出。这样，设计人员可以站在较高的层次上进行抽象思维，避免过早地陷入特定的条件逻辑、算法和过程步等实现细节，能够更好地确定模块和模块间的结构。

在总体设计阶段，一般来说，用于表示软件结构的工具主要包括：

1. 层次图

层次图是在软件总体设计阶段所使用的表示工具之一，用来描绘软件的层次结构。图中的每个方框代表一个模块，方框间的连线表示模块的调用关系。图 4.2 是层次图的一个例子。

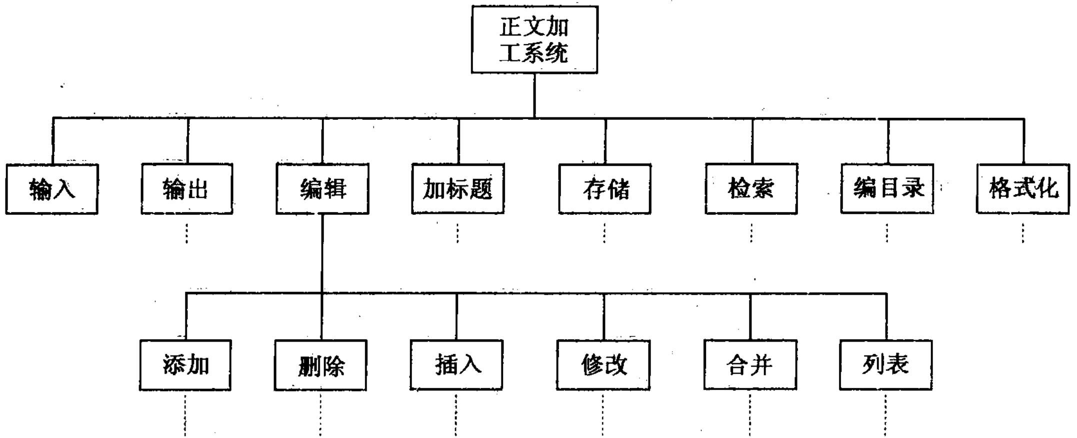


图 4.2 正文加工系统的层次图

其中，最顶层的方框代表正文加工系统的主控模块，它调用下层模块完成正文加工的全部功能；第二层的每个模块控制完成正文加工的一个主要功能，例如"编辑"模块通过调用它的下属模块可以完成六种编辑功能中的任何一种。

层次图很适合在自顶向下设计软件的过程中使用。

2. HIPO 图

HIPO 图是由美国 IBM 公司发明的，其中 HIPO 是"层次图 + 输入/处理/输出"的英文缩写。从名字可以看出，HIPO 图实际上由 H 图和 IPO 图两部分组成，而 H 图就是上一节讲述的层次图，如图 4.3 所示。

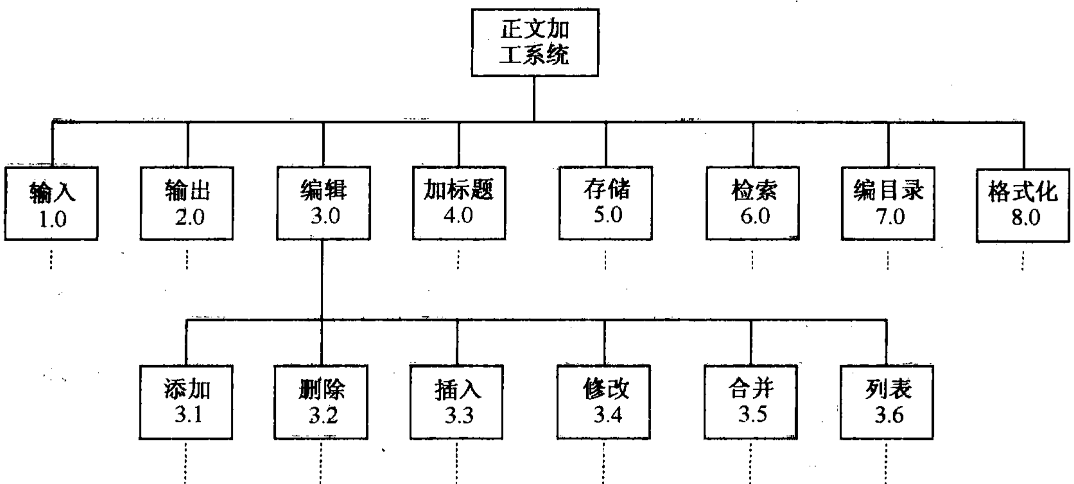


图 4.3 带编号的层次图(H 图)

其中,为了能使 HIPO 图具有可跟踪性,除 H 图(层次图)最顶层的方框之外,在每个方框都加了编号。编号规则如下:第一层中各模块的编号依次为 1.0,2.0,3.0…;如果模块 2.0 还有下层模块,那么下层模块的编号依次为 2.1,2.2,2.3…;如果模块 2.2 又有下层模块,那么下层模块的编号依次为 2.2.1,2.2.2,2.2.3…,依此类推。

对于 H 图中的每个方框,应有一张 IPO 图描述这个方框所代表的模块的处理逻辑。如图 4.4 所示。

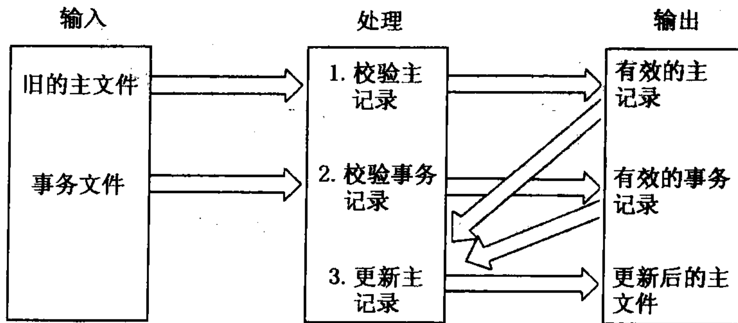


图 4.4　IPO 图的一个例子

图 4.4 是一个主文件更新的例子。由该图可知,IPO 图使用的符号既少又简单,能够方便地描述输入数据、数据处理和输出数据之间的关系。IPO 图的基本形式是在左边的框(输入框)中列出有关的输入数据,在中间的框(处理框)中列出主要的处理以及处理次序,在右边的框(输出框)中列出产生的输出数据。另外,还用类似向量符号(箭头线)清楚地指出数据通信的情况。

值得强调的是,HIPO 图中的每张 IPO 图内都应该明显地标出它所描绘的模块在 H 图中的编号,以便跟踪了解这个模块在软件结构中的位置。

3. 结构图

Yourdon 提出的结构图是进行软件结构设计的另一个有力的工具。结构图和层次图类似,也是描述软件结构的图形工具,但描述能力比层次图更强。图中每个方框代表一个模块,框内注明模块的名字或主要功能;方框之间的直线表示模块的调用关系。因为位于结构图上方的方框所代表的模块意指调用下方的模块,因此,即使使用直线也不会在模块之间调用关系这一问题上产生二义性。

在结构图中通常使用带注释的箭头表示模块调用过程中传递的信息,如果希望进一步标明传递的信息是数据还是控制信息,则可以利用注释箭头尾部的形状来区分:尾部是空心圆表示传递的是数据,实心圆表示传递的是控制信息。图 4.5 是结构图的一个例子。

以上介绍的是结构图的基本符号,也是最经常使用的符号。此外还有一些附加的符号,可以表示模块的选择调用或循环调用。图 4.6 表示当模块 M 中某个判定为真时调用模块 A,为假时调用模块 B,图 4.7 表示模块 M 循环调用模块 A,B 和 C。

注意,层次图和结构图:

(1) 不表示模块的调用次序。虽然多数人习惯于按调用次序从左到右绘画模块,但并没有这种规定。

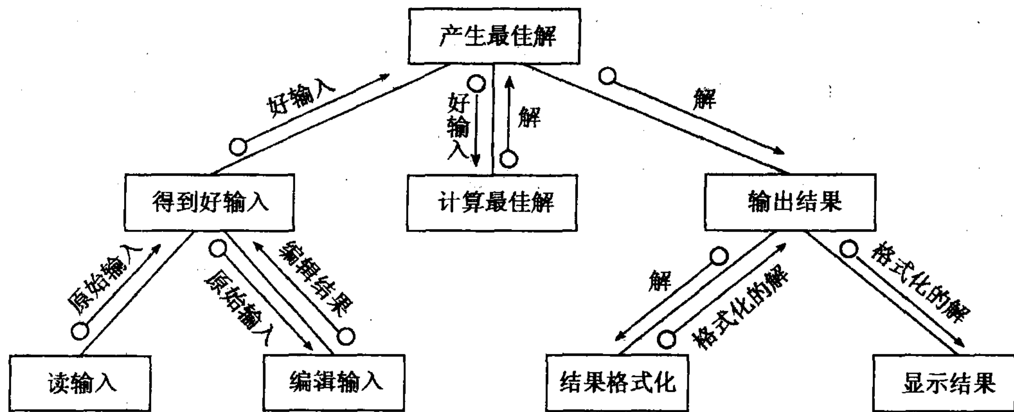


图 4.5 结构图的例子——产生最佳解的一般结构

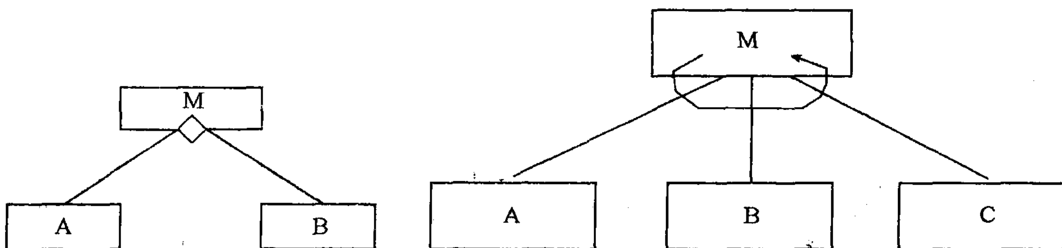


图 4.6 判定为真时调用 A,为假时调用 B

图 4.7 模块 M 循环调用模块 A,B,C

(2) 不指明什么时候调用下层模块。

(3) 只表明一个模块调用哪些模块,至于模块内还有没有其他成份则完全没有表示出来。

4.2 总体设计方法

结构化设计方法 SD(structured design)是基于模块化、自顶向下细化的设计原则上发展起来的。该方法定义了一些不同的“映射”,利用这些映射就可以把数据流图变换成软件结构图,所以这种方法有时也称为面向数据流的设计方法。该方法的基本思路为:

首先对需求规约所产生 DFD 进行分类,然后将不同类型的 DFD,采用不同方法进行映射,将 DFD 转换为初始模块结构图,再根据基本的模块化设计原则——“高内聚低耦合”,精化初始模块结构图,使之成为最终可供详细设计使用的模块结构图(MSD)。

4.2.1 数据流图的类型

通过大量软件开发的实践,人们发现,无论被建系统的数据流图如何复杂,一般总可以把他们分成两种基本的类型,即变换型数据流图和事务型数据流图。下面分别讨论这两种数据流的特征。

1. 变换型数据流图

具有较明显的输入、变换(或称主加工)、输出界面的数据流图,称为变换型数据流图。如

图 4.8 所示。

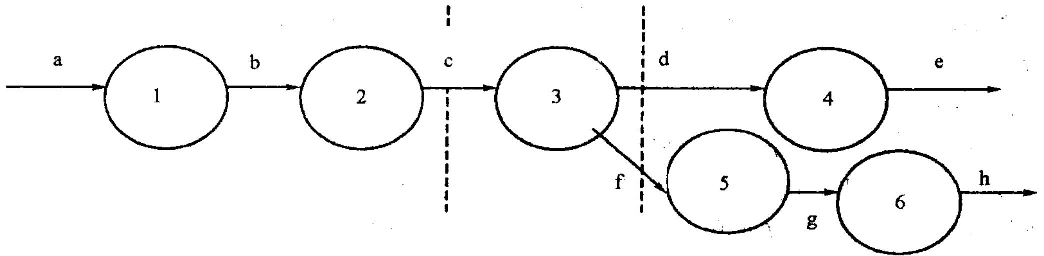


图 4.8 变换型数据流图

其中,左边那条虚线是输入与变换之间的界面,右边那条虚线是变换与输出之间的界面。为了叙述方便,将穿越左边那条虚线的输入(如图 4.8 中标识为 c 的输入),称为逻辑输入;而将穿越右边那条虚线的输出(如图 4.8 中标识为 d,f 的输出),称为逻辑输出。相对应地,将标识为 a 的输入,称为物理输入;而将标识为 e,h 的输出称为物理输出。

可见,该类 DFD 所对应的系统,在高层次上来讲,由三部分组成,即处理输入数据的部分、数据变换部分以及处理数据输出部分。数据沿输入"通路"进入系统,经"处理输入数据部分"后,由外部形式变换为系统内部形式;然后进入系统的"数据变换部分",将之转换为待输出的数据形式;最后,沿着输出"通路",既由处理数据输出部分,将待输出的数据变换为用户需要的数据形式。

由上可知,变换型数据处理问题的工作过程大致分为三步,即取得数据、变换数据和给出数据。如图 4.9 所示。这三步反映了变换型数据流图关于数据处理的基本思想,或者说这类数据流图概括而抽象地表示了这一数据处理的模式。其中,数据变换是这一数据处理模式的核心。

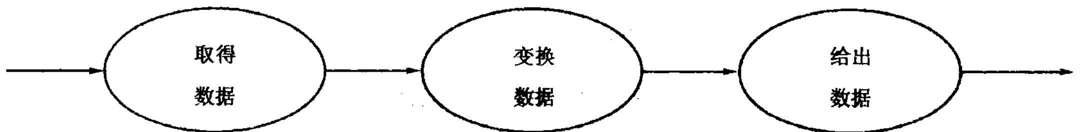


图 4.9 变换型数据流图所表示的数据处理模式

根据变换型数据流图所表示的数据处理模式,可以很容易得出:其对应的软件结构应由"主控"模块以及与该模式三个部分相对应的模块组成。就图 4.8 所示的数据流图而言,该系统的高层软件结构如图 4.10 所示。

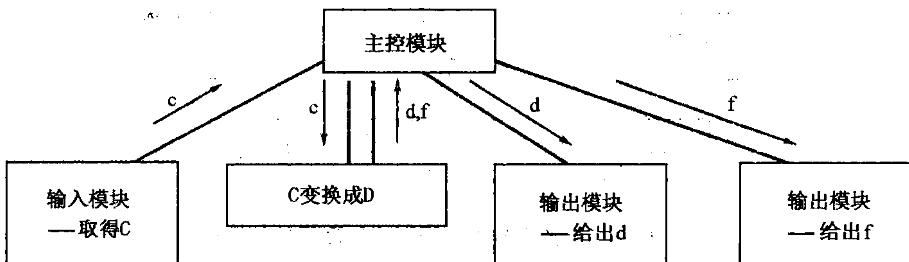


图 4.10 系统的高层软件结构

注意，在该例子中，因为数据流图有一个逻辑输入 c，因此只有一个输入模块；又因为有两个逻辑输出，因此处理输出部分就有两个输出模块，一个是输出模块——给出 d，一个是输出模块——给出 f。

2. 事务型数据流图

任何软件系统从本质上来说都是信息的变换装置，因此，原则上所有的数据流图都可以归为变换型。但是，当数据流图具有和图 4.11 类似的形状时，即数据到达一个处理 T，该处理 T 根据输入数据的类型或数据值，在其后的若干动作序列（称为一个事务）中选出一个来执行，这类数据流图称为事务型数据流图。

图 4.11 中，处理 T 称为事务中心，它完成下述任务：

(1) 接收输入数据；

(2) 分析并确定对应的事务；

(3) 选取与该事务对应的一条活动路经。

事务型数据流图所描述的系统，其数据处理模式为“集中-发散”式。

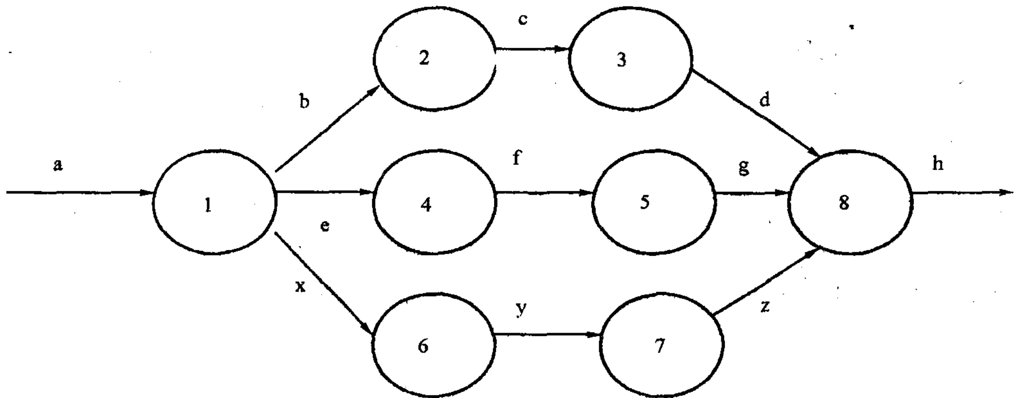


图 4.11 事务型数据流图

针对图 4.11 所示的事务型数据流图，其高层的软件结构如图 4.12 所示。

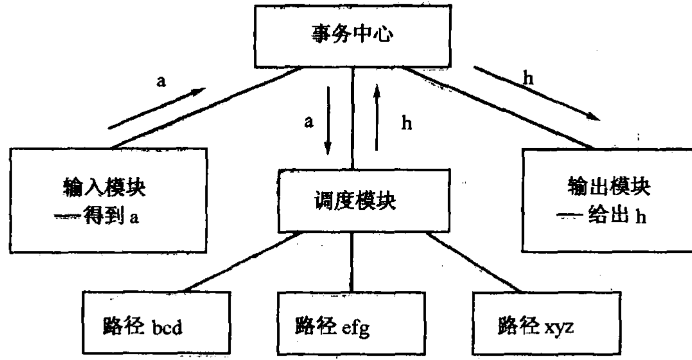


图 4.12 事务型数据流图对应的高层软件结构

其中，每一路径完成一项事务处理并且一般可能还要调用若干个操作模块。而这些模块又可以共享一些细节模块。因此，事务型数据流图可以具有多种形式的软件结构。

在实际工作中，由于数据流图的基本特征，往往将被建系统的数据流图首先看作是变换型数据流图，而把其中的某些部分作为事务型数据流图予以处理。

4.2.2 变换设计与事务设计

依据结构化设计方法解决问题的基本思路，针对两种不同类型的数据流图，分别提出了变换设计和事务设计。在使用这两种设计将数据流图“映射”为模块结构图中，首先映射为初始的模块结构图，而后，根据在实践中提炼出来的实现“高内聚低耦合”的启发式设计规则，再将初始的模块结构图转换为最终可供详细设计使用的模块结构图。如图 4.13 所示。

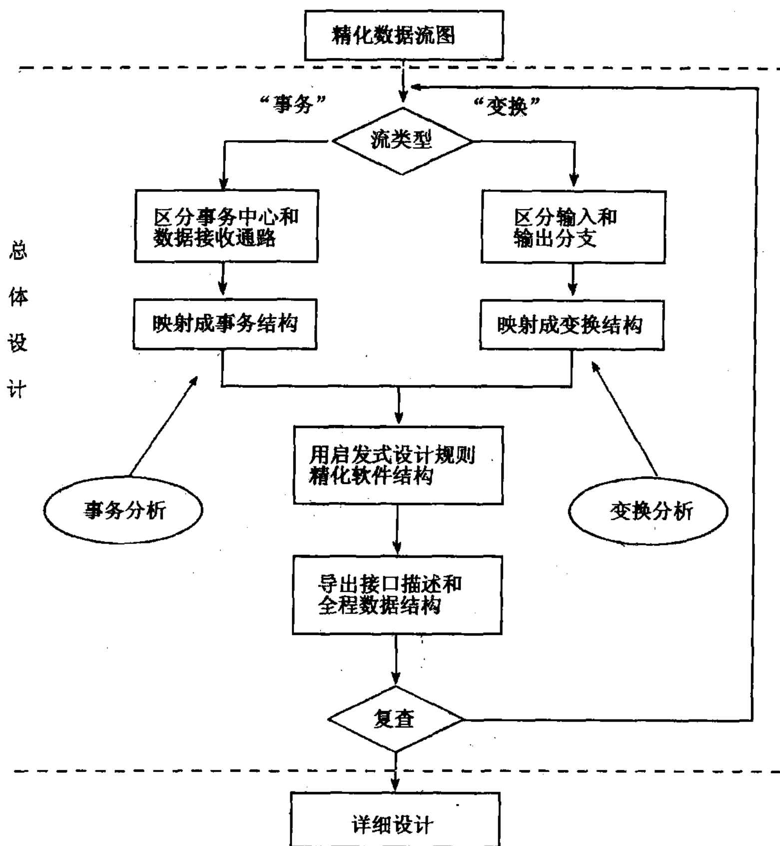


图 4.13 面向数据流方法的设计过程

1. 变换设计

变换设计是在需求规约的基础上，经过一系列设计步骤，将变换型数据流图转换为系统的模块结构图。下面通过一个简单例子说明变换设计的步骤。

(1) 例子

假设汽车数字仪表板将完成下述功能：

① 通过模-数转换，实现传感器和微处理器的接口；

② 在发光二极管面板上显示数据；

③ 指示速度(公里/小时)、行驶的里程、油耗(公里/升)等；

④ 指示加速或减速；

⑤ 超速报警：如果车速超过 55 公里/小时，则发出超速报警铃声。

在软件需求分析阶段，应对上述每一条功能要求以及系统的其他特性进行全面的分析与评价，并建立必要的文档资料，特别是数据流图。在此假定，通过需求分析之后，该系统的数据流图如图 4.14 所示。

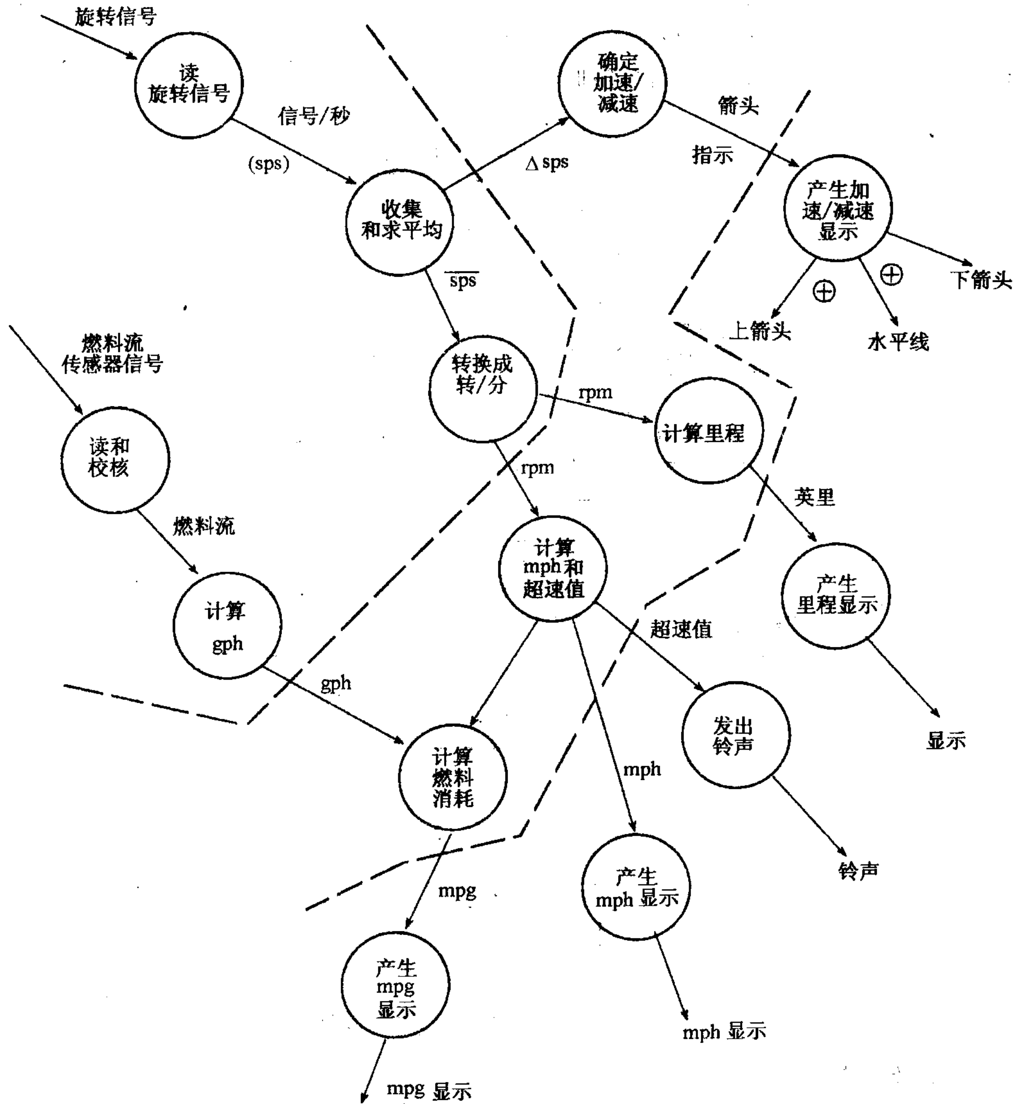


图 4.14 数字仪表板系统数据流图

其中：sps 为转速的每秒信号量；$\overline{\text{sps}}$为 sps 的平均值；Δsps 为 sps 的瞬时变化值；rpm 为每分钟转速；mph 为每小时公里数；gph 为每小时燃烧的燃料升数；rpm 为行驶里程。

(2) 设计步骤

变换设计由六步组成，其中前五步将给定的变换型数据流图转换为初始的模块结构图。最后一步，即第六步，则要根据“高内聚低耦合”的原则，将初始的模块结构图转换为最终的模块结构图。

第 1 步：复查基本系统模型。

复查已建的系统模型，确保系统的输入数据和输出数据符合实际情况。

第 2 步：复查并精化数据流图。

对在需求分析阶段得到的数据流图进行复查。复查的主要方面是：

① 该数据流图是否给出了系统正确的逻辑模型；

② 该数据流图中的每个加工是否代表了一个规模适中、相对独立的功能，即确定是否需要进一步精化。

第 3 步：确定输入、变换、输出这三部分之间的边界。

根据加工的语义以及相关的数据流，划分系统的逻辑输入、逻辑输出和中心变换部分，确定输入、输出和变换三部分之间的界面。其中，值得注意的是，输入流和变换之间的界面，以及变换和输出流之间的界面，对不同的设计人员来说，可能会在选取上有所不同，这表明他们对边界的解释有所不同。但是，这些不同通常不会对软件结构产生太大的影响。不过，该步的工作应仔细认真，以形成比较理想的结果。

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。因此可以用以下方法来确定系统的逻辑输入和逻辑输出。首先，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，一直到数据流不再被看作是系统的输入为止，则其前一个数据流就是系统的逻辑输入。也就是说，逻辑输入就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，从物理输入端开始，一步一步向系统的中间移动，就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；处在输入部分和输出部分中间的，就是中心变换部分。

第 4 步：“第一级分解”——系统模块结构图顶层和第一层的设计。

软件结构代表了对控制的自顶向下的分配，因此，所谓分解就是分配控制的过程。其关键是确定系统树形结构图的根或顶层模块，以及由这一根模块所控制的第一层模块，既“第一级分解”。

根据变换型数据流图的基本特征，显然它所对应的软件系统应由输入模块、变换模块和输出模块组成。并且，为了协调这些模块的“有序”工作，还应设计一个所谓的主模块，作为系统的顶层模块。因此，变换型数据流图所对应的软件结构有一个主模块以及由它控制的三部分组成，即：

① 主模块或称主控模块：位于最顶层，一般以所建系统的名字为其命名，它的任务是协调并控制第一层模块，完成系统所要做的各项工作；

② 输入模块部分：协调对所有输入数据的接受，为主模块提供加工数据；对于该部分的设计，一般来说有几个不同的逻辑输入，就设计几个输入模块；

③ 变换模块部分：接受输入模块部分的数据，并对这些内部形式的数据进行加工，产生系统所有的输出数据（内部形式）；

④ 输出模块部分：协调所有输出数据的产生过程，最终将变换模块产生的输出数据，以用户可视的形式输出。对该部分的设计，一般来说有几个不同的逻辑输出，就设计几个输出模块。

由此可见，顶层和第一层的设计，基本上是一个“机械”的过程。

针对以上所示的数据流图，经过“第一级分解”之后，可以得到如图 4.15 所示的顶层和第一层的模块结构图。

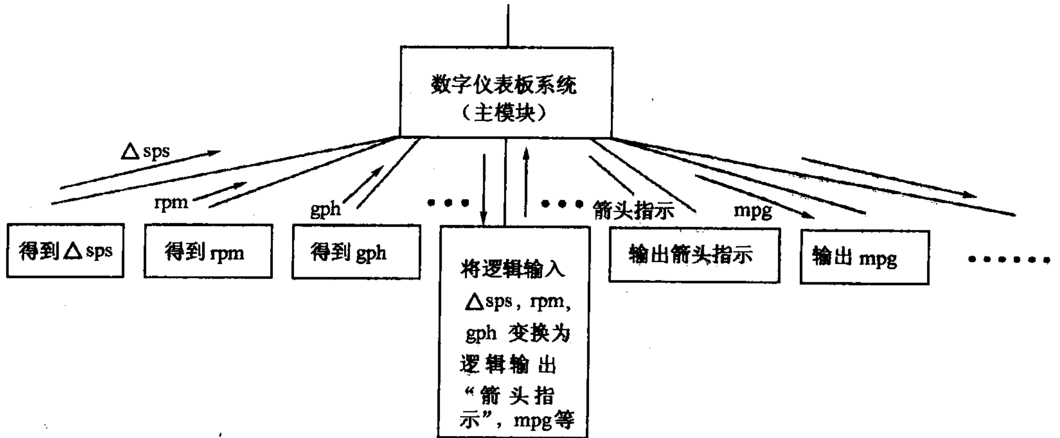


图 4.15　数字仪表板系统的第一级分解

其中，在第一层的输入模块部分中有三个模块，它们是：“得到 Δsps”、“得到 rpm”、“得到 gph”；输出部分有五个输出模块，它们是：“输出指示箭头”、“输出英里”、“输出超时值”、“输出 mph”、“输出 mpg”。

第 5 步：“第二级分解”——自顶向下，逐步求精。

第二级分解通过一个自顶向下逐步细化的过程，为每一个输入模块、输出模块和变换模块设计它们的从属模块。

① 一般来说，首先对每一输入模块设计其下层模块。输入模块的功能是向调用它的上级模块提供数据，因此它必须有一个数据来源。如果该来源不是物理输入，那么该输入模块就必须将其转换为上级模块所需的数据。因此一个输入模块通常可以分解为两个下属模块：一个是接收数据模块（也可以称为输入模块）；另一个是把接受的数据变换成它的上级模块所需的数据（通常把这一模块也称为变换模块）。如图 4.16(a)所示。

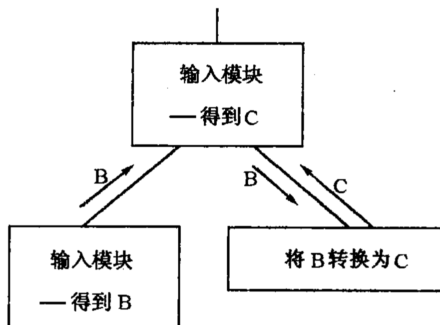
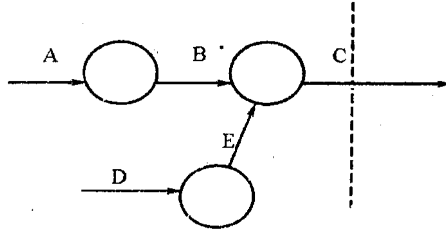


图 4.16(a)　输入模块的分解

继之，对下属的输入模块以同样方式进行分解，直到一个输入模块物理输入，则细化工作停止。

在输入模块的细化中，一般可以把它分解为“一个输入模块和一个变换模块”。但是，对于一些具体情况，要进行特定的处理，例如：



该输入部分有一个逻辑输入，根据上述第一级分解，对应第一层的一个输入模块-得到 C。但这一模块有两个数据来源，一个是 B，一个是 E，因此对这一模块的分解就有三个下属模块，其中两个是输入模块，如图 4.16(b)所示。

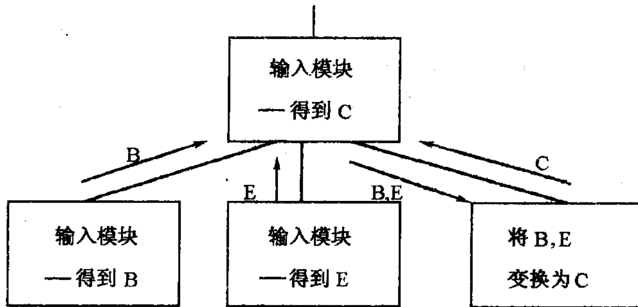
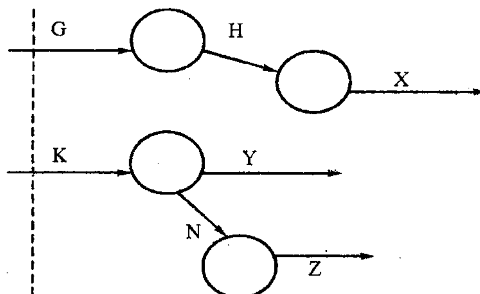


图 4.16(b) 输入模块的分解

由图 4.16(b)可以看出，对于每一输入模块的第二级分解，基本上也是可以“机械”进行的。

② 接着，对每一输出模块进行分解。如果该输出模块不是一个物理输出，那么，通常可以分解为两个下属模块：一个将得到的数据向输出形式进行转换，另一个是将转换后的数据进行输出。例如：



该输出部分有两个逻辑输出，因此通过第一级分解后有两个输出模块，通过第二级分解后，得到的模块结构图如图 4.17 所示。

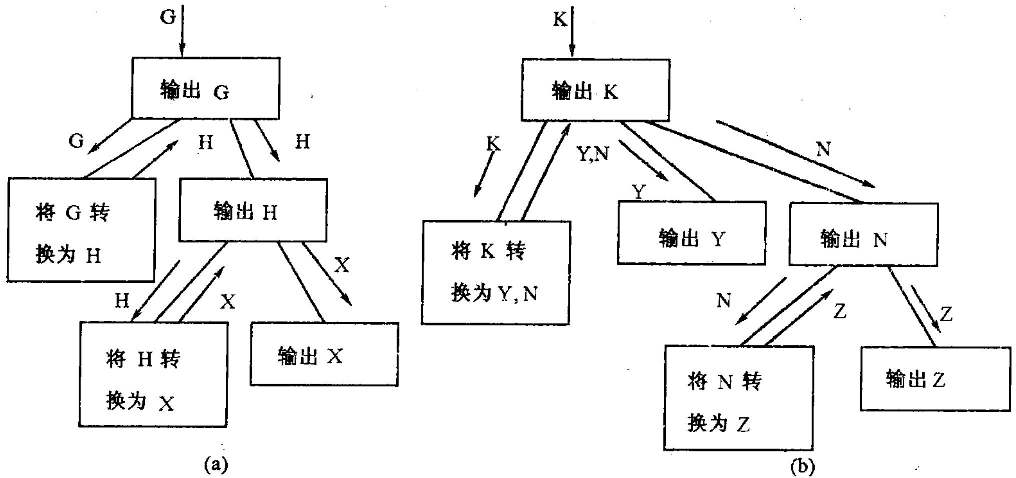


图 4.17　输出模块的分解

由该例可以看出，对于每一输出模块的第二级分解，基本上也是可以“机械”进行的。

③ 在第二级分解中，关于中心变换模块的分解一般没有一种通用的方法，通常应依据数据流图的具体情况，并以功能分解的原则，考虑如何对中心变换模块进行分解。

通过以上五步，就可以将变换型数据流图比较“机械”地转换为初始的模块结构图。这意味着将一个变换型数据流图转换为初始的模块结构图，几乎不需要设计人员的创造性劳动。但是，在将初始的模块结构图转换为最终可供详细设计使用的模块结构图中，即在第六步中，则需要设计人员的一些创造能力。

第 6 步：使用设计度量和启发式规则，对初始的模块结构图进行精化（将在第 4.4 节中进行讨论）。

通过上述六个设计步骤，便完成了总体设计，即将一个变换型数据流图转换为模块结构图，给出了被建系统的软件整体表示。一般来说，在此之后，还应对得到的软件结构进行复审，必要时还可能需要做一些精化软件结构的工作，这对提高软件的一些性质，特别是对提高软件质量将产生一定的影响。

2. **事务设计**

虽然在任何情况下都可以使用变换设计将一个给定的 DFD 转换为模块结构图，但是，当数据流图具有明显的事务型特征时，也就是有一个明显的事务处理中心时，则比较适宜采用事务设计。

事务设计的步骤和变换设计的步骤大体相同，即：

第 1 步：复查基本系统模型。

复查已建的系统模型，确保系统的输入数据和输出数据符合实际情况。

第 2 步：复查并精化数据流图。

对在需求分析阶段得到的数据流图进行复查。

第 3 步：确定事务处理中心。

第 4 步：“第一级分解”——系统模块结构图顶层和第一层的设计。

变换设计同样是以数据流图为基础，按“自顶向下，逐步细化”的原则进行的。

① 首先，为事务中心设计一个主模块；

② 然后,为每一条活动路径设计一个事务处理模块;

③ 一般来说,事务型数据流图都有输入部分,对其输入部分设计一个输入模块;

④ 如果一个事务型数据流图的各活动路径又集中于一个加工,如图 4.18(a)所示,则为此设计一个输出模块;如果各活动路径是发散的,如图 4.18(b)所示,则在第一层设计中就不必为其设计输出模块。

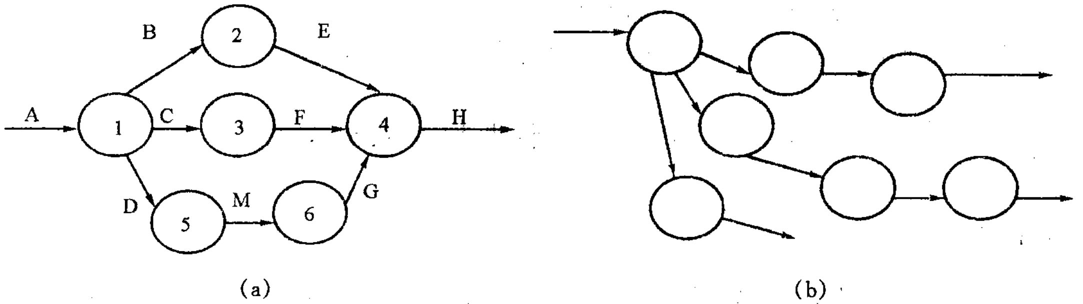


图 4.18 (a)一个事务型数据流图;(b)路径发散的事务型数据流图

针对图 4.18(a)所示的数据流图,经第一层设计后,可以得到如图 4.19 所示的高层初始的模块结构图。

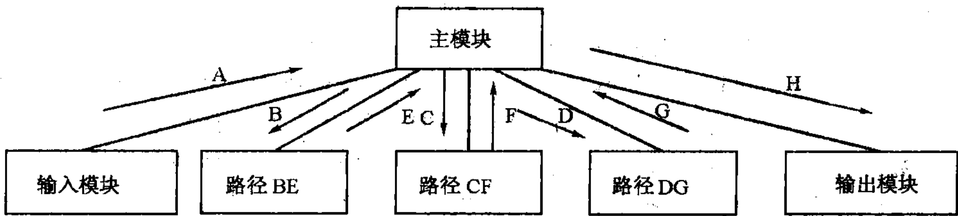


图 4.19 事务型数据流图的高层模块设计

第 5 步:"第二级分解"——自顶向下,逐步求精。

关于输入模块、输出模块的细化,如同变换设计对输入模块、输出模块的细化。关于各条活动路径模块的细化,则要根据具体情况进行,没有特定的规律可循。就图 4.18(a)而言,对应的初始模块结构图如图 4.20 所示。

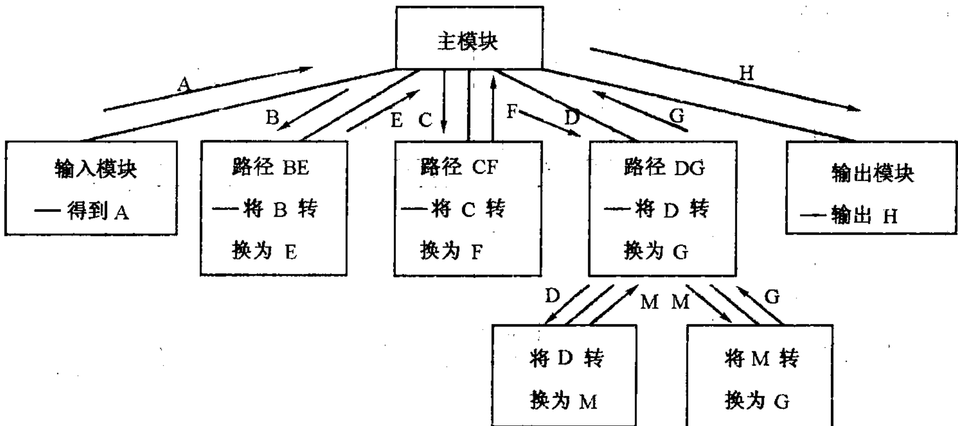


图 4.20 事务型数据流图的第二级分解

第 6 步：使用设计度量和启发式规则，对初始的模块结构图进行精化（将在第 4.4 节中进行讨论）。

实践中，一个大型的软件系统一般是变换型流图和事务型流图的混合结构。在软件总体设计中，通常以变换设计为主，事务设计为辅进行结构设计。即首先利用变换设计，把软件系统分为输入、中心变换和输出三个部分，设计上层模块，然后根据各部分数据流图的的结构特点，适当地利用变换设计和事务设计进行细化，得到初始的模块结构图，再按照“高内聚低耦合的原则”，对初始的模块结构图进行精化，得到最终的模块结构图。

4.3 设计评价准则与启发式规则

4.3.1 设计评价准则

结构化软件设计是一种典型的模块化方法，即如何把一个待开发的软件分解成若干简单的模块，这一过程称为模块化。针对其中涉及的两个主要问题：一是如何将系统分解成软件模块，二是如何设计模块，运用了人们处理复杂事物的基本原则——“分而治之”和“抽象”。即通过“信息隐蔽”，进行系统分解和模块设计。具体地说，在自顶向下逐步求精时，其较低层的设计细节都被“隐蔽”起来，只给出模块的接口，这样不仅使功能的执行机制被隐蔽起来，而且控制流程的细节和一些数据也被隐蔽起来，随着设计逐步往低层推进，其细节也逐步显露出来。

由此可见，模块是执行一个特殊任务或实现一个特殊的抽象数据类型的一组例程和数据结构。模块通常由两部分组成。一部分是接口：列出可由其他模块或例程访问的对象，例如常量、变量、数据类型、函数等；接口不但刻画了各个模块之间的关联，体现了模块功能，而且还支持分层构造软件模块的可见性控制，对其他模块的设计者和使用者是可见的。另一部分是实现模块功能的执行机制，包括私有量（只能由本模块自己使用的）及实现模块功能的过程描述或源程序代码。

评价软件设计的基本准则是“高内聚”、“低耦合”。

1. 耦合

耦合是对不同模块之间相互依赖程度的度量。紧密耦合是指两个模块之间存在着很强的依赖关系；松散耦合是指两个模块之间存在一些依赖关系，但它们之间的连接比较弱；无耦合是指模块之间根本没有任何连接（见图 4.21）。

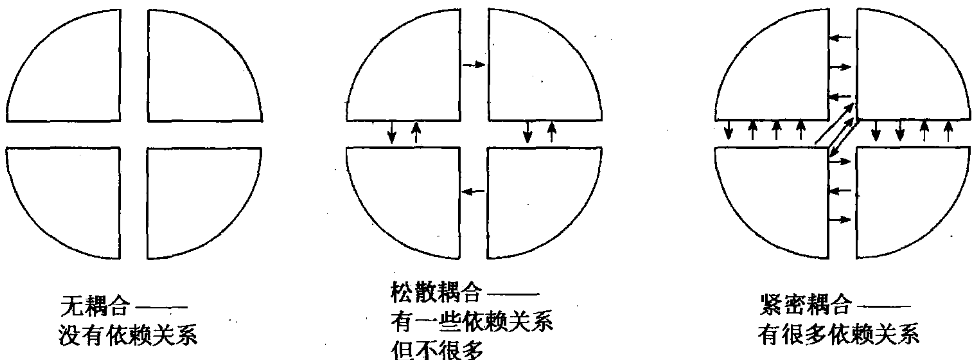


图 4.21 模块间的耦合程度

耦合的强度依赖于以下几个因素：

- 一个模块对另一个模块的引用，例如，模块 A 调用模块 B，那么模块 A 的功能依赖于模块 B 的功能；
- 一个模块向另一个模块传递的数据量，例如，模块 A 为了完成其功能需要模块 B 向其传递一组数据，那么模块 A 依赖于模块 B；
- 一个模块施加到另一个模块的控制的数量，例如，模块 A 传递给模块 B 一个控制信号，模块 B 执行的操作依赖于控制信号的值；
- 模块之间接口的复杂程度，例如，如果模块 A 给模块 B 传递一个简单的数值，但模块 C 和模块 D 之间传递的是数组，甚至是控制信号，则模块 A 和 B 之间接口的复杂度小于模块 C 和 D 之间的接口复杂度。

下面我们按从强到弱的顺序给出几种常见的模块间耦合的类型：

(1) 内容耦合

当一个模块直接修改或操作另一个模块的数据时，就发生了内容耦合。被修改的模块完全依赖于修改它的模块，主要表现为以下两种情形：

① 一个模块访问或修改另一个模块的内部数据；

② 一个模块不通过正常入口而跳转到另一个模块的内部。

内容耦合是最高程度的耦合，应该尽量避免使用。

(2) 公共耦合

我们可以通过使用全局或公共数据来多少降低一些耦合强度(如 FORTRAN 语言的 COMMAN 块或其他高级程序设计语言中声明的全局变量)，这种两个以上的模块共同引用一个全局数据项就称为公共耦合。模块间的依赖关系依旧存在，因为对全局数据项的修改作用于所有访问该数据项的模块。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。公共耦合的情形见图 4.22。

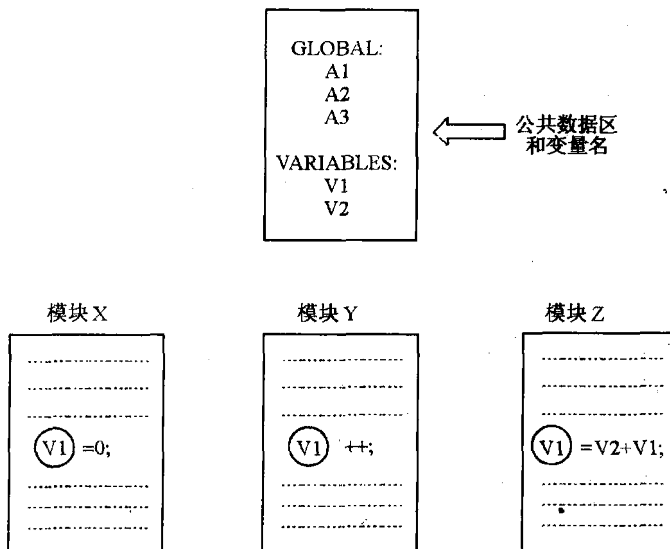


图 4.22　公共耦合示意图

(3) 控制耦合

一个模块在界面上传递一个信号控制另一个模块，接收信号的模块的动作根据信号值进

行调整，称为控制耦合。通过保证每个模块只完成一个特定的功能，可以大大地减少模块间传递的控制信息。

(4) 标记耦合

若两个模块至少有一个通过界面传递的公共参数包含内部结构，如字符串或记录，则称这两个模块之间存在标记耦合。标记耦合的例子见图 4.23。

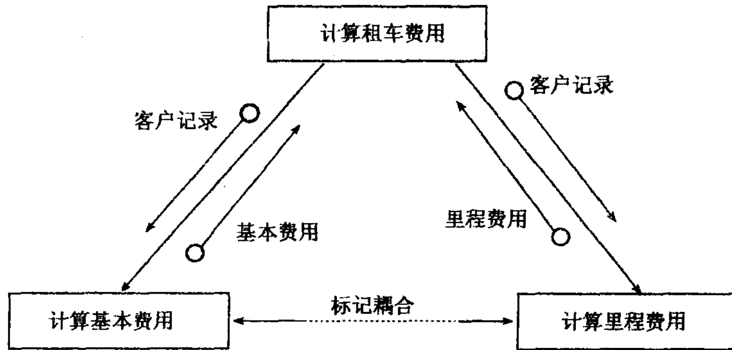


图 4.23 标记耦合的例子

(5) 数据耦合

模块间通过参数传递基本类型的数据，称为数据耦合。数据耦合是最简单的耦合形式，系统中至少必须存在这种类型的耦合，因为只有当某些模块的输出数据作为另一些模块的输入数据时，模块之间才能联结成为一个整体，从而完成有意义的功能。耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采取以下原则：如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，坚决避免使用内容耦合。

2. 内聚

不同于度量模块之间的相互依赖程度，内聚度量的是一个模块内部各成分之间相互关联的强度。一个模块内聚程度越高，该模块内部各成分之间以及同模块所完成的功能之间的关联也就越强。换句话说，如果一个模块的所有成分都直接参与并且对于完成同一功能来说都是最基本的，则该模块是高内聚的。正如耦合分成不同的级别，内聚也是如此。设计时追求的目标应尽量使每个模块做到高内聚，这样模块的各个成分都和模块的单一功能直接相关。以下从低到高给出一些常见的内聚类型，如图 4.24 所示。

(1) 偶然内聚

如果一个模块的各成分之间毫无关系，则称为偶然内聚。例如，有时在编写一段程序时，发现有一组语句在两处或多处出现，于是把这组语句作为一个模块以减少书写工作量，如果这组语句彼此间没有任何关系，这时就出现了偶然内聚。

在偶然内聚的模块中，各个成分之间没有实质性联系，很可能在一个应用场合需要修改这个模块，在另一个应用场合又不允许这种修改，从而陷入困境。事实上，偶然内聚的模块出现修改错误的概率比其他类型的模块高得多。

(2) 逻辑内聚

几个逻辑上相关的功能被放在同一模块中，则称为逻辑内聚。例如，一个模块读取各种不同类型外设的输入（包括卡片、磁带、磁盘、键盘等），而不管这些输入从哪儿来、做什么用，因为

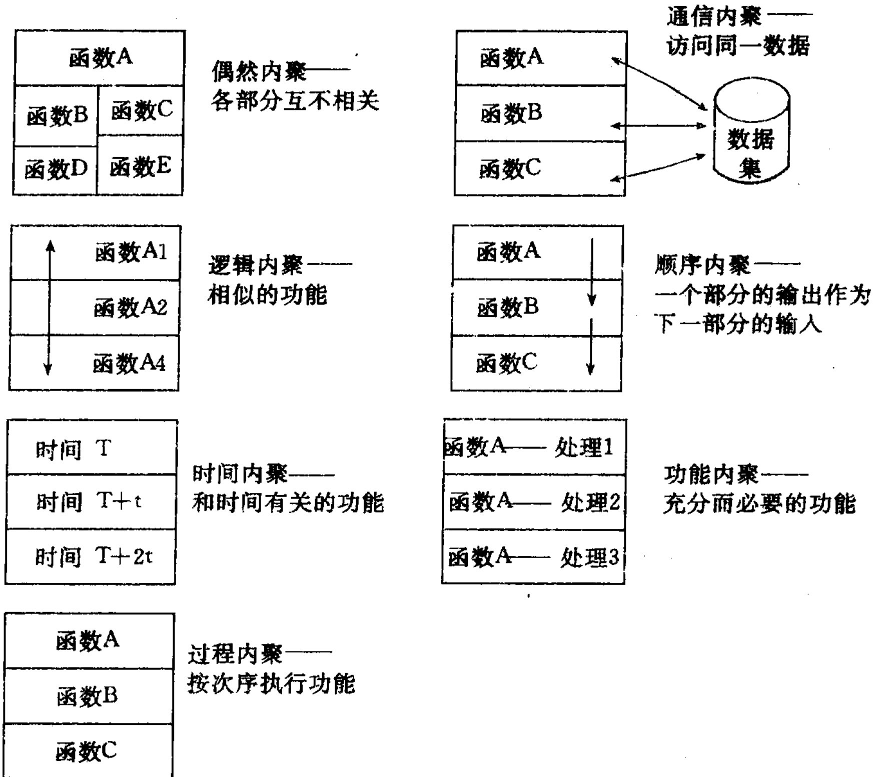


图 4.24　一些常见的内聚类型

这个模块的各成分都执行输入，所以该模块是逻辑内聚的。

尽管逻辑内聚比偶然内聚合理一些，但逻辑内聚的模块各成分在功能上并无关系，即使局部功能的修改有时也会影响全局，因此这类模块的修改也比较困难。例如在上面的例子中，因为输入可以用于不同模块的不同目的，所以实质上是把不同的功能放在一个地方。

(3) 时间内聚

如果一个模块完成的功能必须在同一时间内执行(例如，初始化系统或一组变量)，但这些功能只是因为时间因素关联在一起，则称为时间内聚。

时间内聚在一定程度上反映了系统的某些实质，因此比逻辑内聚好一些。

(4) 过程内聚

如果一个模块内部的处理成分是相关的，而且这些处理必须以特定的次序执行，则称为过程内聚。使用程序流程图作为工具设计软件时，常常通过研究流程图确定模块的划分，这样得到的往往是过程内聚的模块。

(5) 通信内聚

如果一个模块的所有成分都操作同一数据集或生成同一数据集，则称为通信内聚。例如，所有的处理都在一个磁盘或磁带上实施，有时这样的安排显得很方便。然而，通信内聚经常破坏设计的模块化和功能独立性。

(6) 顺序内聚

如果一个模块的各个成分和同一个功能密切相关，而且一个成分的输出作为另一个成分的输入，则称为顺序内聚。因为模块不是基于功能关系组织在一起的，很可能一个模块没有包含对于完成一个功能所需的全部处理。

(7) 功能内聚

最理想的内聚是功能内聚，模块的所有成分对于完成单一的功能都是基本的。功能内聚的模块对完成其功能而言是充分必要的。

内聚和耦合是密切相关的，同其他模块存在强耦合的模块常意味着弱内聚，而强内聚的模块常意味着该模块同其他模块之间松散的耦合。在进行软件设计时，应力争做到强内聚、弱耦合。

4.3.2 启发式规则

根据以上提出的设计准则，人们通过长期的软件开发实践，总结出一些启发式规则，这些规则在许多场合可以有效地改善软件结构。这些规则主要包含：

1. 改进软件结构提高模块独立性

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

2. 模块规模应该适中

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

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

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

3. 深度、宽度、扇出和扇入应适中

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

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

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

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

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

观察大量软件系统后发现，设计得很好的软件结构通常顶层扇出比较高，中层扇出较少，底

层扇入到公共的实用模块中去(底层模块有高扇入)。即系统的模块结构呈现为“葫芦”形状。

4. *模块的作用域应该在控制域之内*

模块的作用域定义为受该模块内一个判定影响的所有模块的集合。模块的控制域是这个模块本身以及所有直接或间接从属于它的模块的集合。例如,在图 4.25 中模块 A 的控制域是 A, B, C, D, E, F 等模块的集合。

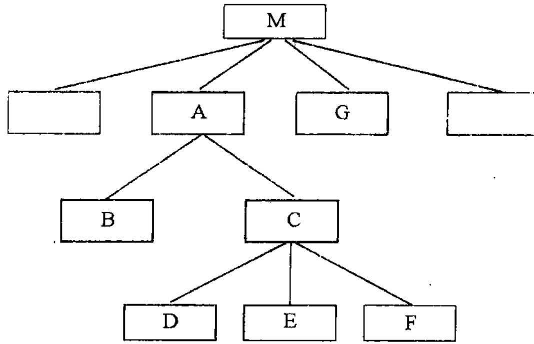


图 4.25 模块的控制域与作用域

在一个设计得很好的系统中,所有受判定影响的模块应该都从属于做出判定的那个模块,最好局限于做出判定的那个模块本身及它的直属下级模块。例如,如果图 4.25 中模块 A 做出的判定只影响模块 B,那么是符合这条规则的。但是,如果模块 A 做出的判定同时还影响模块 G 中的处理过程,又会有什么坏处呢?首先,这样的结构使得软件难于理解;其次,为了使得 A 中的判定能影响 G 中的处理过程,通常需要在 A 中给一个标记设置状态以指示判定的结果,并且应该把这个标记传递给 A 和 G 的公共上级模块 M,再由 M 把它传给 G。这个标记是控制信息而不是数据,因此将使模块间出现控制耦合。

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

到底采用哪种方法改进软件结构,需要根据具体问题统筹考虑。一方面应该考虑哪种方法更现实,另一方面应该使软件结构能更好地体现问题本来的结构。

5. *力争降低模块接口的复杂性*

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

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

QUAD-ROOT(A, B, C, ROOT1, ROOT2)

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

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

6. *模块功能应该可以预测*

模块的功能应该能够预测,但也要防止模块功能过分局限。

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

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

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

4.4 设计优化——初始模块结构图的精化

设计优化的主要任务就是将经变换设计和事务设计的前五步所得到的初始模块结构图转换为最终可供详细设计使用的模块结构图。可见，优化设计就是变换设计和事务设计的第6步。其中，对初始的模块结构图，根据模块独立性原则进行精化，使模块具有尽可能高的内聚和尽可能松散的耦合，最终得到一个易于实现、易于测试和易于维护的软件结构。

现以上面给出的数字仪表板系统为例，说明如何进行求精。

1. 输入部分的精化

针对数字仪表板系统的输入部分，其初始的模块结构图如图4.26所示。

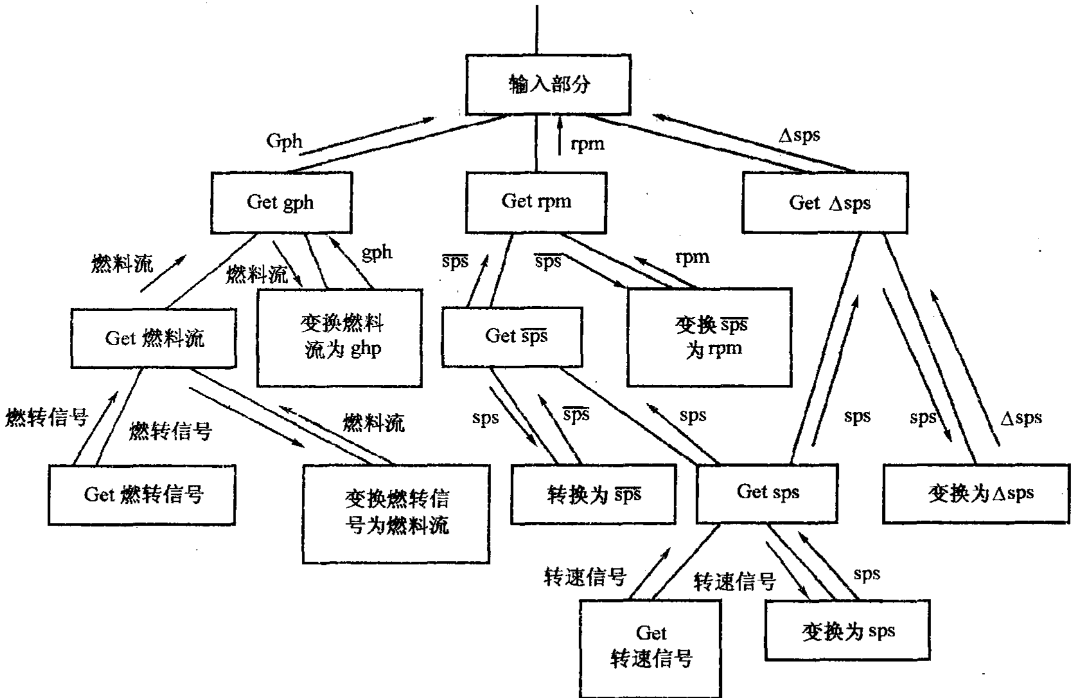


图4.26 数字仪表板系统输入部分的初始模块结构图

针对这一实例，使用启发式规则1，并考虑其他规则，可以将上述的模块结构图精化为如图4.27所示的模块结构图。

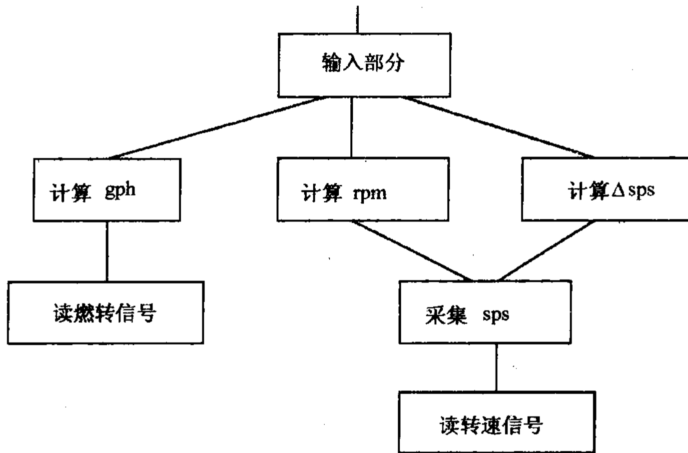


图4.27 输入部分——精化的模块结构图

由上得知，在精化输入部分中，通常是：

(1) 为每一物理输入设计一个模块，如“读转速信号”、“读燃转信号”；

(2) 对那些不进行实际数据输入的输入模块，且输入的数据是予加工或辅助加工得到的结果，例如，Get gph 模块和 Get 燃料流模块就是这样的输入模块，不需为这样的输入模块设计专门的软件模块，应将它们与其他模块合并在一起；

(3) 对于那些既简单、规模又小的模块，可以合并在一起，这样，不但提高了模块内的联系，而且还减少了模块间的耦合。

就以上的例子而言，运用(2)，(3)可以：

① 把“Get gph”模块和“Get 燃料流”模块，与“变换燃转信号为燃料流”模块、“变换燃料流为 ghp”模块合并为模块“计算 gph”；

② 把“get rpm”模块、“get $\overline{\text{sps}}$”模块与“变换为 $\overline{\text{sps}}$”模块以及“变换$\overline{\text{sps}}$为 rpm”模块，合并为“计算 rpm”模块；

③ 把“get sps”模块、“变换为 sps”模块，合并为“采集$\overline{\text{sps}}$”模块；

④ 把“get Δsps”模块、“变换为 Δsps”模块，合并为“计算 Δsps”模块。

2. 输出部分的精化

还是以数字仪表板系统为例，说明输出部分的求精。该系统的输出部分的初始模块结构图如图4.28所示。

对于这一初始的模块结构图，一般情况下应：

(1) 把相同或类似的物理输出合并为一个模块，以减少模块之间的关联。就本例而言：

左边前三个“显示”，基本上属于相似的物理输出，因此可以把它们合并为一个显示模块。而将“PUT mpg”模块和相关的“生成显示”的模块合并为一个模块；同样地，应把“PUT mph”模块、“PUT 里程”各自与相关的生成显示的模块合并为一个模块，参见图4.29。

(2) 其他求精的规则，与输入部分类同。例如，可以将“PUT 加/减速”模块与其下属的两个模块合并为一个模块，将“PUT 超速量”模块与其下属的两个模块合并为一个模块。

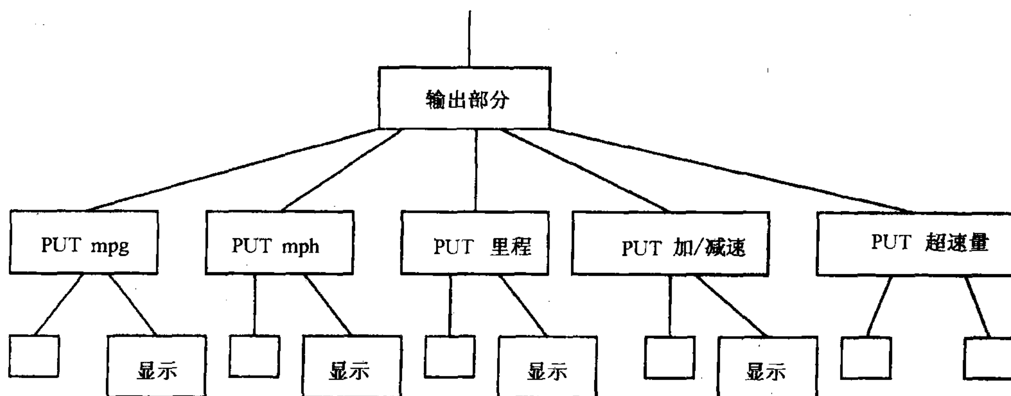


图 4.28 数字仪表板系统输出部分的初始模块结构图

通过以上求精之后,数字仪表板系统的输出部分的软件结构如图 4.29 所示。

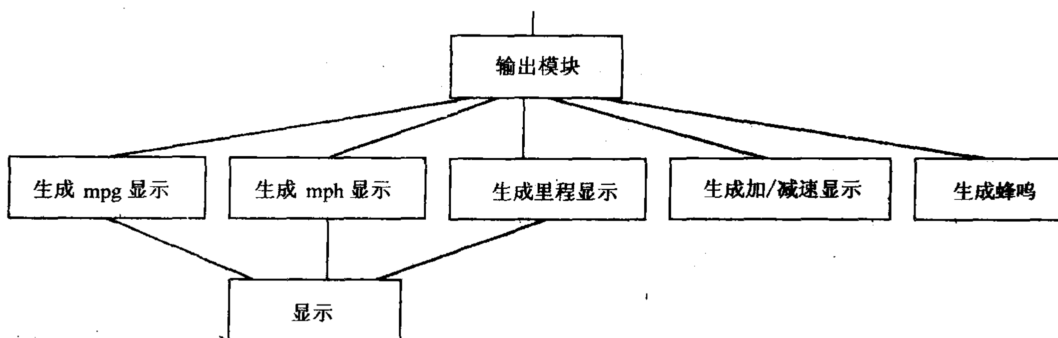


图 4.29 输出部分——精化的模块结构图

3. 变换部分的精化

对于变换部分的求精,如前所述,这是一项具有挑战性的工作。但是,其中主要是根据设计准则,并要通过实践,不断地总结经验,才能设计出合理的模块结构。就给定的数字仪表板系统而言,如果把"确定加/减速"的模块放在"计算速度 mph"模块下面,则可以减少模块之间的关联,提高模块的独立性。通过这一求精,对于变换部分,就可以得到如图 4.30 所示的模块结构图。

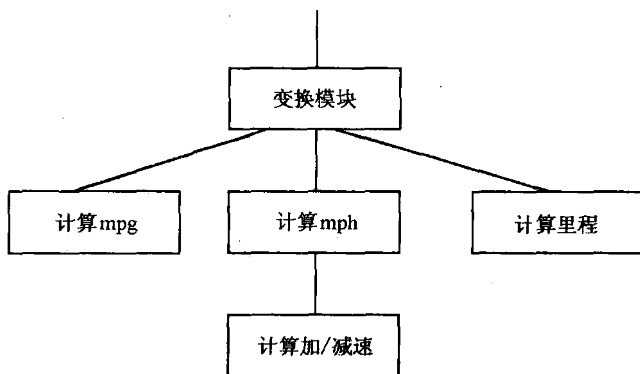


图 4.30 变换部分——精化的模块结构图

通过以上讨论，可以看出，在总体设计中，如果说将一个给定的DFD转换为初始的模块结构图基本上是一个“机械”的过程，无法体现设计人员的创造力，那么，优化设计将一个初始的模块结构图转换为最终的模块结构图，对设计人员将是一种挑战，其结果将直接影响软件系统开发的质量。

4.5 详细设计

经过总体设计阶段的工作，已经确定了软件的模块结构和接口描述，但这时每个模块仍处于黑盒子级。详细设计阶段的根本目标是确定怎样具体地实现所要求的系统，也就是说，经过这个阶段的设计工作，应该得出对目标系统的精确描述，从而在编码阶段可以将这个描述直接翻译成用某种程序设计语言书写的程序。因此，详细设计的结果基本上决定了最终的程序代码的质量。

详细设计以总体设计阶段的工作为基础，但又不同于总体设计阶段，主要表现为以下两个方面：

(1) 在总体设计阶段，数据项和数据结构以比较抽象的方式描述，例如，总体设计可以声明一组值从概念上表示一个矩阵，详细设计就要确定用什么数据结构来实现这样的矩阵，比如特殊的稀疏矩阵技术也许是最合适的。

(2) 详细设计要提供关于算法的更多的细节，例如，总体设计可以声明一个模块的作用是对一个表进行排序，详细设计则要确定使用哪种排序算法。在详细设计阶段为每个模块增加了足够的细节，使得程序员能够以相当直接的方式编码每个模块。

因此，详细设计的模块包含实现对应的总体设计的模块所需要的处理逻辑，主要有：

① 详细的算法；

② 数据表示和数据结构；

③ 实施的功能和使用的数据之间的关系。

每个模块被编码成过程、子程序、函数或其他类型的命名实体。

4.5.1 结构化程序设计

结构化程序设计是一种特定的程序设计方法学。严格地说，程序设计方法学是以程序设计方法为研究对象的学科(第一种含义)。它主要涉及用于指导程序设计工作的原理和原则，以及基于这些原理和原则的设计方法和技术，着重研究各种方法的共性和个性，各自的优缺点。一方面要涉及到方法的理论基础和背景，另一方面也要涉及到方法的基本体系结构和实用价值。程序设计方法学的第二种含义是，针对某一领域或某一领域的特定一类问题，所用的一整套特定程序设计方法所构成的体系。例如，基于Ada程序设计语言的程序设计方法学。关于程序设计方法学的两种含义之间的基本关系是，第二种含义是第一种含义的基础，第一种含义是在第二种含义的基础上的总结、提高，上升到原理、原则和理论的高度。

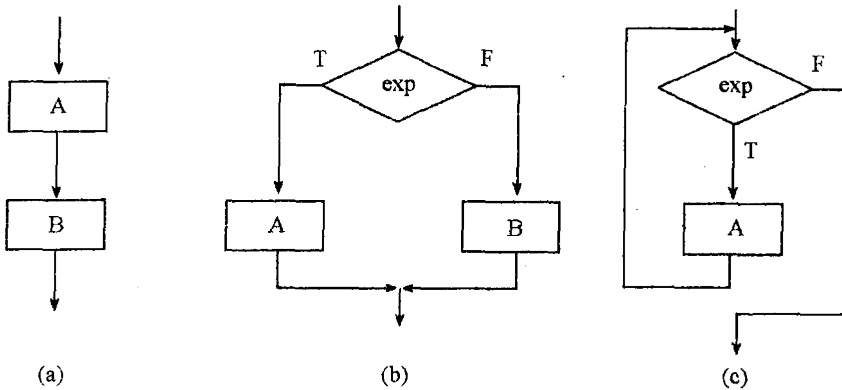
作为一整套特定程序设计方法所构成的体系(第二种含义)，目前已出现多种程序设计方法学，例如，结构化程序设计方法学、各种逻辑式程序设计方法学、函数式程序设计方法学、面向对象程序设计方法学等。

结构化程序设计是一种结构性的编程方法。结构性主要反映如下：第一，编程工作为一演

化过程，即按抽象级别依次降低、逐步精化、最终得出所需程序的方法编程；第二，按模块组装的方法编程，亦即将所需程序编制成只含顺序构造、判定构造以及重复构造，其中每一构造只允许一个入口和一个出口。

采用结构化程序设计方法编程，旨在提高编程(过程)质量和所编程序的质量。自顶向下、逐步求精方法，有利于在每一抽象级上尽可能保证编程工作与所编程序的正确性；按模块组装方法编程以及所编程序只含顺序、判定、重复三种构造，可使程序结构良好、易读、易理解、易维护，并易于保证及验证程序的正确性。

结构化程序设计的概念最早由 E. W. Dijkstra 在 60 年代中期提出，并在 1968 年著名的 NATO 软件工程会议上首次引起人们的广泛关注。1966 年，C. Bohm 和 G. Jacopini 在数学上证明了，只用三种基本的控制结构就能实现任何单入口单出口的程序，这三种基本的控制结构就是“顺序”、“选择”和“循环”，它们的流程图表示见图 4.31。实际上，用顺序结构和循环结构(又称 DO-WHILE 结构)完全可以实现选择结构(又称 IF-THEN-ELSE 结构)，因此，理论上最基本的控制结构只有两种。



(a) 顺序结构，先执行 A 再执行 B；(b) IF-THEN-ELSE 型选择(分支)结构；
(c) DO-WHILE 型循环结构

图 4.31 三种基本的控制结构

Bohm 和 Jacopini 的证明给结构化程序设计技术奠定了理论基础。

与此同时，结构化程序设计技术作为一种新的程序设计思想、方法和风格，也开始引起工业界的重视。1971 年，IBM 公司在纽约时报信息库管理系统的设计中使用了结构化程序设计技术，获得了巨大的成功，于是开始在整个公司内部全面采用结构化程序设计技术，并介绍给了它的许多用户。IBM 在计算机界的影响为结构化程序设计技术的推广起到了推波助澜的作用。

结构化程序设计的目标之一是使程序的控制流程线性化，即程序的动态执行顺序符合静态书写结构，这就增强了程序的可读性，不仅容易理解、调试、测试和排错，而且给程序的形式化证明带来了方便。正如 Bohtn 和 Jacopini 所证明的，顺序结构、选择结构和循环结构构成了结构化程序设计的核心，它们组合使用可以实现任意复杂的处理逻辑，除此之外无需其他控制结构，这样一来，无条件转移指令 GOTO 语句变得多余了。

1968 年，ACM 通信发表了 Dijkstra 的短文“GOTO Statement considered harmful”，引起了人们的极大关注。Dijkstra 认为：GOTO 语句是构成程序结构混乱不堪的主要原因，一切高级

程序设计语言应该删除 GOTO 语句。自此开始了关于 GOTO 语句的学术讨论。

这一场讨论直到 1974 年人们才达成了共识。N. Wirth 在 PASCAL User Manual and Report 一书中给出了关于 GOTO 语句的建议:

当出现算法的自然结构被破坏的异常情况时,应保留 GOTO 语句。一个好的原则是避免使用跳转表达正常的循环或条件语句,因为这样的跳转破坏了程序的静态文本结构在动态计算结构中的反映。换句话说,如果程序的静态结构和动态结构没有较好对应起来,就会影响程序的清晰度,并使验证工作变得更加困难。

关于 GOTO 语句讨论的实质在于:程序设计首先是讲究结构,还是讲究效率。好结构的程序不一定是效率最高的程序。结构化程序设计的观点是要求设计好结构的程序。在计算机硬件技术迅速发展的今天,人们已普遍认为,除了系统的核心程序部分以及其他一些有特殊要求的程序以外,在一般情况下,宁可牺牲一些效率,也要保证程序有一个好的结构。

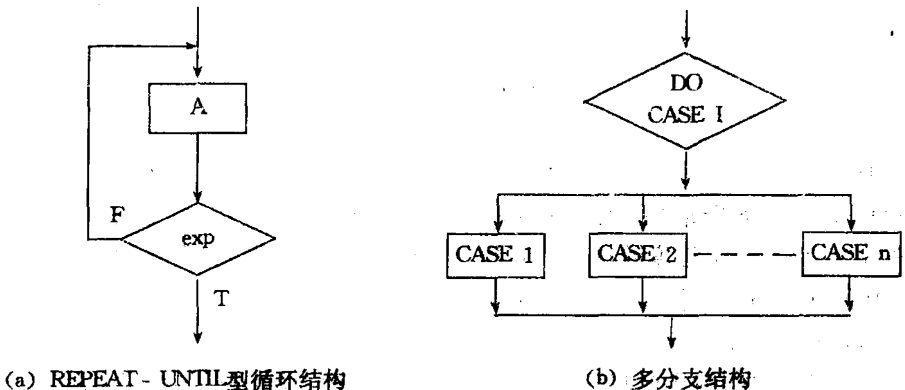
4.5.2 详细设计的工具

详细设计的任务是给出软件模块结构中各个模块的内部过程描述,也就是模块内部的算法设计。我们这里并不打算讨论具体模块的算法设计(感兴趣的读者可以参考 N. Wirth 所著的 Algorithms + Data Structures = Programs 一书),而是讨论这些算法的表示形式。详细设计的工具可以分为图形、表格和语言三种,无论是哪类工具,对它们的基本要求都是能提供对设计的无歧义的描述,即能指明控制流程、处理功能、数据组织以及其他方面的实现细节,从而在编码阶段能把设计描述直接翻译成程序代码。下面我们介绍一些典型的详细设计工具。

1. 程序流程图

程序流程图又称为程序框图,它是历史最悠久、使用最广泛的描述软件设计的方法,然而它也是用得最混乱的一种方法。从 40 年代末到 70 年代中期,程序流程图一直是软件设计的主要工具。它的主要优点是对控制流程的描绘很直观,便于初学者掌握。由于流程图历史悠久,广泛为人所熟悉,所以尽管它有种种缺点,甚至许多人建议停止使用它,但至今仍在广泛使用着。不过总的趋势是越来越多的人不再使用程序流程图了。

程序流程图中使用的主要符号包括顺序结构、选择结构和循环结构,如上节图 4.31 所示。值得注意的是,程序流程图中的箭头代表的是控制流而不是数据流,这一点是同数据流图中的箭头不同的。除了以上的三种基本控制结构外,为了方便起见,程序流程图中还经常使用其他一些等价的符号,如图 4.32 所示。



(a) REPEAT - UNTIL型循环结构　　(b) 多分支结构

图 4.32 其他常用的控制结构

程序流程图的主要缺点如下：

(1) 程序流程图本质上不是逐步求精的好工具，它诱使程序员过早地考虑程序的控制流程，而不去考虑程序的全局结构；

(2) 程序流程图中用箭头代表控制流，因此程序员不受任何约束，可以完全不顾结构程序设计的精神，随意转移控制；

(3) 程序流程图不易表示数据结构。

应该指出，详细的微观程序流程图——每个符号对应于源程序的一行代码，对于提高大型系统的可理解性作用甚微。

2. 盒图(N-S 图)

出于要有一种不允许违背结构程序设计精神的考虑，在70年代早期，Nassi 和 Shneiderman 提出了盒图，又称为 N-S 图。同程序流程图相比，它以一种结构化的方式严格地限制从一个处理到另一个处理的控制转移。

每一个 N-S 图开始于一个大的矩形，表示它所描述的模块。该矩形的内部被分成不同的部分，分别表示不同的子处理过程，这些子处理过程又可以进一步分解成更小的部分。由于每次分解都只能使用图 4.33 给出的基本符号，因此最终得到的详细设计必然是结构化的。

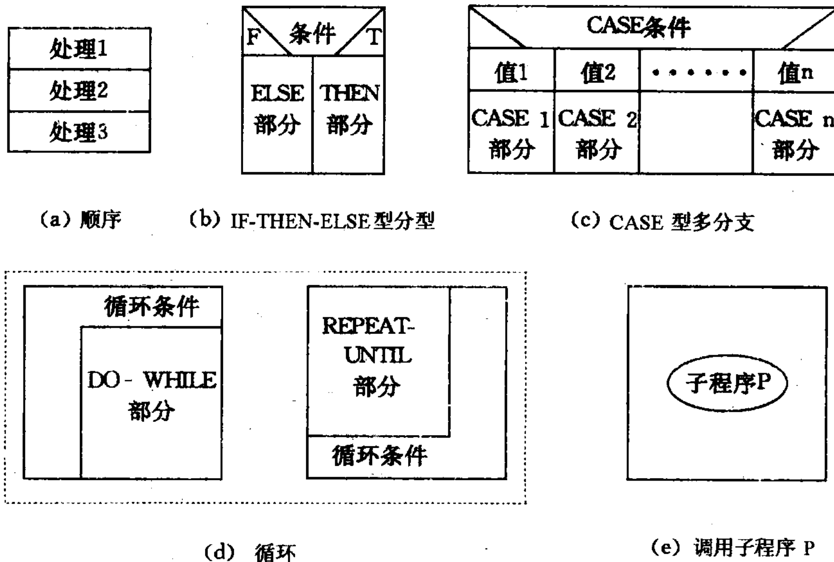


图 4.33 盒图的基本符号

3. PAD 图

PAD 是问题分析图(Problem Analysis Diagram)的英文缩写，自 1973 年由日本日立公司发明以来，已经得到一定程度的推广。它用二维树形结构的图表示程序的控制流，将这种图转换为程序代码比较容易。图 4.34 给出 PAD 图的基本符号。

PAD 图的主要优点如下：

(1) 使用表示结构化控制结构的 PAD 符号所设计出来的程序必然是结构化程序；

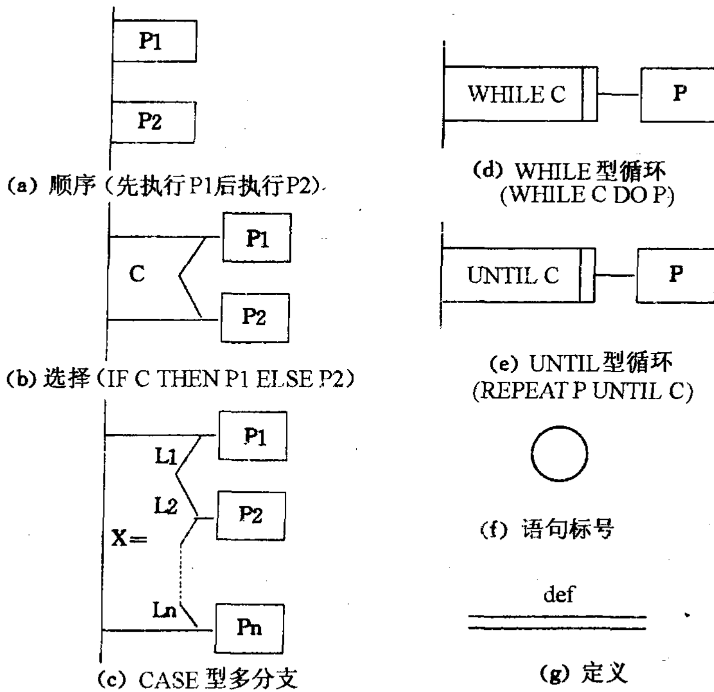


图 4.34　PAD 图的基本符号

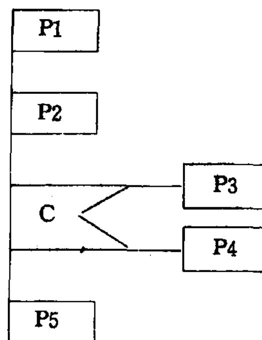
(2) PAD 图所描述的程序结构十分清晰。图中最左边的竖线是程序的主线，即第一层控制结构。随着程序层次的增加，PAD 图逐渐向右延伸，每增加一个层次，图形向右扩展一条竖线。PAD 图中竖线的总条数就是程序的层次数；

(3) 用 PAD 图表现程序逻辑，易读、易懂、易记。PAD 图是二维树型结构的图形，程序从图中最左边上端的结点开始执行，自上而下，从左向右顺序执行；

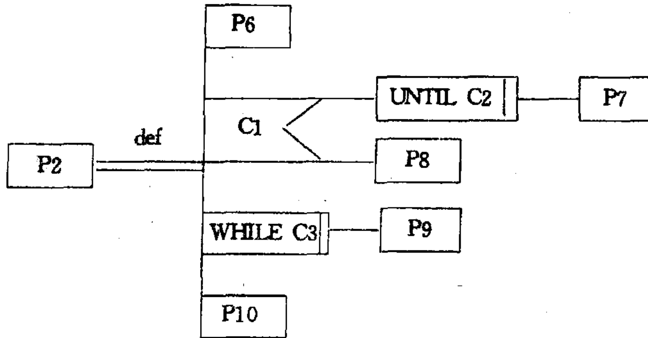
(4) 很容易将 PAD 图转换成高级语言源程序，这种转换可用软件工具自动完成，从而可省去人工编码的工作，有利于提高软件可靠性和软件生产率。

(5) 既可用于表示程序逻辑，也可用于描述数据结构；

(6) PAD 图的符号支持自顶向下、逐步求精方法的使用。开始时设计者可以定义一个抽象程序，随着设计工作的深入而使用“def”符号逐步增加细节，直至完成详细设计，如图 4.35 所示。



(a) 初始的PAD图



(b) 使用def符号细化处理P2

图 4.35 使用 PAD 图提供的定义功能来逐步求精的例子

PAD 图是面向高级程序设计语言的，为 FORTRAN，COBOL，PASCAL 和 C 等常用的高级程序设计语言都提供了一整套相应的图形符号。由于每种控制语句都有一个图形符号与之对应，显然将 PAD 图转换成与之对应的高级语言程序比较容易。

4．类程序设计语言(PDL)

类程序设计语言(Program Design Language，简称 PDL)也称为伪码，这是一个笼统的名称，现在有许多种不同的 PDL 在使用。它是用正文形式表示数据结构和处理过程的设计工具。

一般说来，PDL 是一种"混合"语言，一方面，PDL 具有严格的关键字外部语法，通常是借用某种结构化的程序设计语言(如 PASCAL 或 C)的语法控制框架，用于定义控制结构和数据结构；另一方面，PDL 使用一种语言(通常是某种自然语言，如汉语或英语)的词汇，灵活自由地表示实际操作和判定条件，以便可以适应各种工程项目的需要。

(1) PDL 具有下述特点：

① 关键字的固定语法，提供了结构化控制结构、数据说明和模块化的手段。为了使结构清晰和可读性好，通常在所有可能嵌套使用的控制结构的头和尾都有关键字，例如，if…fi(或endif)等等；

② 自然语言的自由语法，用于描述处理过程和判定条件；

③ 数据说明的手段，既包括简单的数据结构(例如纯量和数组)，又包括复杂的数据结构(例如，链表或层次的数据结构)；

④ 模块定义和调用的技术，提供各种接口描述模式。

(2) PDL 作为一种设计工具有如下一些优点：

① 可以作为注释直接插在源程序中间。这样做能促使维护人员在修改程序代码的同时也相应地修改 PDL 注释，因此有助于保持文档和程序的一致性，提高了文档的质量；

② 可以使用普通的正文编辑程序或文字处理系统，很方便地完成 PDL 的书写和编辑工作；

③ 已经有自动处理程序存在，而且可以自动由 PDL 生成程序代码。

PDL 的缺点是不如图形工具形象直观，描述复杂的条件组合与动作间的对应关系时，不如判定表或判定树清晰简单。

除以上介绍的外，详细设计工具还包括我们前面介绍过的 IPO 图、判定树和判定表等。

第三章和第四章比较详细地介绍了结构化方法，包含结构化需求分析方法和结构化软件设计方法。概括地说，作为一种特定的软件方法学，结构化方法为了支持问题定义和软件求解，紧紧围绕“过程抽象”和“数据抽象”，给出了完备的符号体系、可操作的过程和相应的表示工具。例如，为了支持问题定义，结构化方法提出了以下五个概念，它们是：数据源、数据潭、数据流、加工和数据存储，并给出了相应的表示。应该说，这些概念对于规约软件系统的功能是完备的，即它们可以“覆盖”客观世界的一切事物，并且这些概念的语义还相当简单，容易理解和掌握。另外，结构化方法还给出了控制信息组织复杂性机制，例如“数据打包”等。但是，从软件方法学研究的角度，结构化方法仍然存在一些问题，其中最主要的问题是结构化方法没有“摆脱”冯·诺依曼体系结构的影响，捕获的“功能(过程)”和“数据”恰恰是客观事物的易变性质，并由此建造的系统模型也没有与客观实际系统在结构上保持一致，从而为系统的验证和维护带来相当大的困难，甚至是“灾难性”的。在某种意义上来讲，就是这些问题，促使了面向对象方法学的产生和发展。

通过结构化方法的学习，可以比较好地理解软件方法学这一概念：*软件方法学是以软件方法为研究对象的学科。主要涉及指导软件设计的原理和原则，以及基于这些原理、原则的方法和技术。狭义的软件方法学也指某种特定的软件设计指导原则和方法体系。*

4.6 软件设计规格说明书

在完成软件设计之后，应产生设计规约。设计规约是对软件的组织或其组成部分的内部结构的描述，满足系统需求规约所指定的全部功能及性能要求。通常有概要设计规约和详细设计规约，分别为相应设计过程的输出文档。

概要设计规约指明软件的组织结构，其主要内容包括：

(1) 系统环境　硬件、软件接口与人机界面；外部定义的数据库；与设计有关的限定条件。

(2) 设计描述　数据流和主要数据结构；软件模块的结构；模块之间的接口。

(3) 对每个模块的描述　处理过程外部行为；界面定义；数据结构；必要的注释。

(4) 文件结构和全局数据　文件的逻辑结构、记录描述以及访问方式；交叉引用信息。

此外，还应包括有关软件测试等方面的要求与说明。

概要设计规约是面向软件开发者的文档，主要作为项目管理人员、系统分析人员与设计人员之间交流的媒体。

详细设计规约是对软件各组成部分内部属性的描述，它是概要设计的细化。即在概要设计规约的基础上，增加以下内容：

① 各处理过程的算法；

② 算法所涉及的全部数据结构的描述，特别地，对主要数据结构往往包括与算法实现有关的描述。

详细设计规约主要作为软件设计人员与程序员之间交流的媒体。

随着软件开发环境的不断发展，概要设计与详细设计的内容可以有所变化。

下面给出可供参考的设计规约格式。

1. 引言

　1.1　编写目的

说明编写本软件设计说明书的目的。

1.2 背景说明

(1) 给出待开发的软件产品的名称；

(2) 说明本项目的提出者、开发者及用户；

(3) 说明该软件产品将做什么，如有必要，说明不做什么。

1.3 术语定义

列出本文档中所用的专门术语的定义和外文首字母组词的原词组。

1.4 参考资料

列出本文档中所引用的全部资料，包括标题、文档编号、版本号、出版日期及出版单位等，必要时注明资料来源。

2. 总体设计

2.1 需求规定

说明对本软件的主要输入、输出、处理的功能及性能要求。

2.2 运行环境

简要说明对本软件运行的软件、硬件环境和支持环境的要求。

2.3 处理流程

说明本软件的处理流程，尽量使用图、文、表的形式。

2.4 软件结构

在 DFD 图的基础上，用模块结构图来说明各层模块的划分及其相互关系，划分原则上应细到程序级(即程序单元)，每个单元必须执行单独一个功能(即单元不能再分了)。

3. 运行设计

3.1 运行模块的组合

说明对系统施加不同的外界运行控制时所引起的各种不同的运行模块的组合，说明每种运行所经历的内部模块和支持软件。

3.2 运行控制

说明各运行控制方式、方法和具体的操作步骤。

4. 系统出错处理

4.1 出错信息

简要说明每种可能的出错或故障情况出现时，系统输出信息的格式和含义。

4.2 出错处理方法及补救措施

说明故障出现后可采取的措施，包括：

(1) 后备技术。当原始系统数据万一丢失时启用的副本的建立和启动的技术，如周期性的信息转储；

(2) 性能降级。使用另一个效率稍低的系统或方法(如手工操作、数据的人工记录等)，以求得到所需结果的某些部分；

(3) 恢复和再启动。用建立恢复点等技术，使软件再开始运行。

5. 模块设计说明

以填写模块说明表的形式，对每个模块给出下述内容：

(1) 模块的一般说明，包括名称、编号、设计者、所在文件、所在库、调用本模块的模块

名和本模块调用的其他模块名；

(2) 功能概述；

(3) 处理描述，使用伪码描述本模块的算法、计算公式及步骤；

(4) 引用格式；

(5) 返回值；

(6) 内部接口，说明本软件内部各模块间的接口关系，包括：

(a) 名称，

(b) 意义，

(c) 数据类型，

(d) 有效范围，

(e) I/O 标志；

(7) 外部接口，说明本软件同其他软件及硬件间的接口关系，包括：

(a) 名称，

(b) 意义，

(c) 数据类型，

(d) 有效范围，

(e) I/O 标志，

(f) 格式，指输入或输出数据的语法规则和有关约定，

(g) 媒体；

(8) 用户接口，说明将向用户提供的命令和命令的语法结构，以及软件的回答信息，包括：

(a) 名称，

(b) 意义，

(c) 数据类型，

(d) 有效范围，

(e) I/O 标志，

(f) 格式，指输入或输出数据的语法规则和有关约定，

(g) 媒体。

附：模块说明表

模块说明表 制表日期： 年 月 日

<table>
<tr><td>模块名：</td><td>模块编号：</td><td>设计者：</td></tr>
<tr><td>模块所在文件：</td><td colspan="2">模块所在库：</td></tr>
<tr><td colspan="3">调用本块的模块名：</td></tr>
<tr><td colspan="3">本模块调用的其他模块名：</td></tr>
<tr><td colspan="3">功能概述：</td></tr>
<tr><td colspan="3">处理描述：</td></tr>
<tr><td colspan="3">引用格式：</td></tr>
<tr><td colspan="3">返回值：</td></tr>
</table>

续表

	名 称	意 义	数据类型	数值范围	I/O 标志
内部接口					

	名 称	意 义	数据类型	I/O 标志	格 式	媒 体	
外部接口							
用户接口							

习 题 四

1. 解释以下术语：

   变换型数据流图

   事务型数据流图

   模块

   模块耦合

   模块内聚

   模块的控制域

   模块的作用域

2. 简答以下问题：

(1) 结构化方法总体设计的任务及目标；

(2) 结构化方法详细设计的任务及目标；

(3) 变换设计与事务设计之间的区别；

(4) 提出启发式规则的基本原理；

(5) 为什么说结构化分析与结构化设计之间存在一条“鸿沟”；

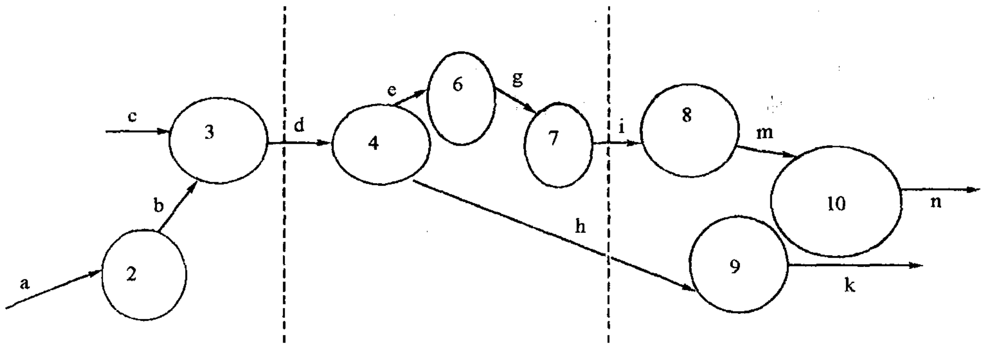
(6) 依据一个系统的 DFD，将其转换为 MSD 的基本思路。

3. 举例说明变换设计的步骤。

4. 举例说明事务设计的步骤。

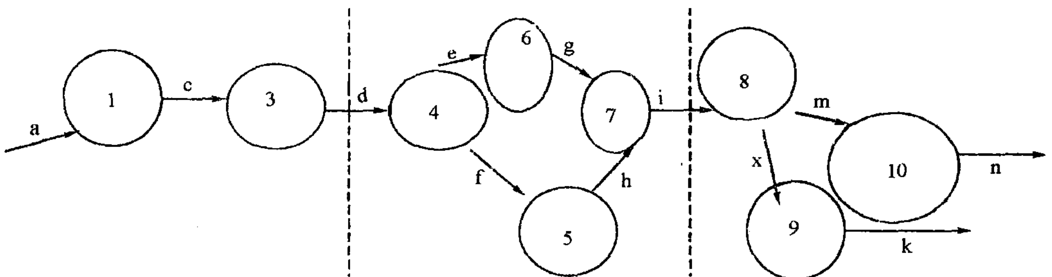
5. 把下面的 DFD 图转换为初始的 MSD 图。

(1)



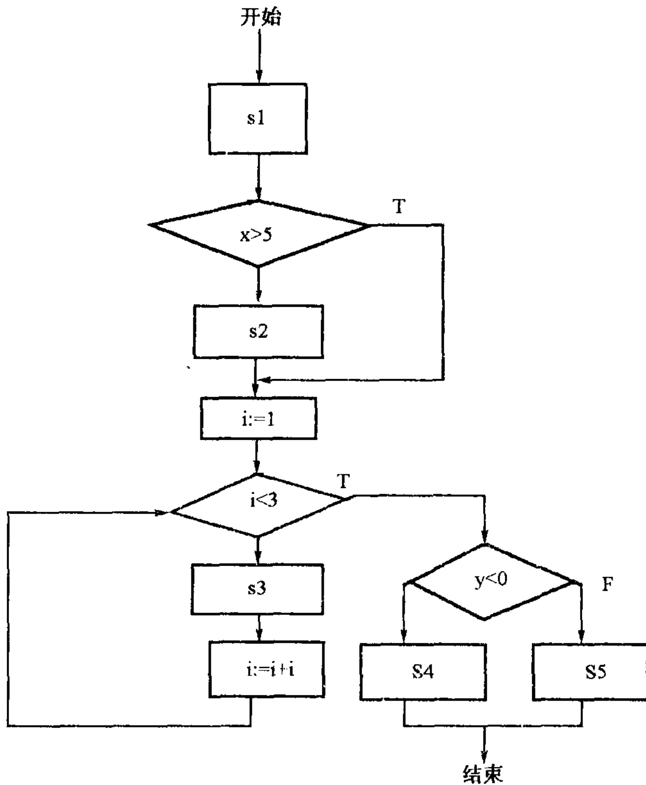
其中，竖虚线表示输入、变换、输出之间的界面。

(2)



其中，竖虚线表示输入、变换、输出之间的界面；变换部分为一个事务型数据流图。

6. 把下面的程序流程图转换为 PAD, N-S 图和伪码。



7. 综合实践题：

假定教务管理包括以课程为中心进行资源(教师、教室、学生)配置，并根据各科考试成绩进行教学分析。在这一假定下，结合实际情况，给出教务管理系统的需求陈述，建立该系统的结构化模型，并在此基础上给出该系统的模块结构图。最后对这一实践进行必要的总结。

第五章　面向对象方法

自20世纪80年代以来,面向对象方法发展迅速,目前已深入到计算机科学技术的很多领域。当前,面向对象方法已经成为软件开发的主流方法,适合于在各种问题域中建造各种规模和复杂度的系统。

长期以来,关于如何建造一个软件系统中的模块,先后出现了四种基本观点。① 以"过程"或"函数"为基点来构造一个模块,使每一模块实现一项功能;② 以数据结构为基点来构造一个模块,使每个模块容纳一个数据结构;③ 以事件驱动为基点来构造一个模块,使每一模块能够识别一个事件并对该事件作出反应;④ 以问题域中的成分为基点来构造一个模块,使每一模块惟一地对应现实世界中的某一事物。目前,这种观点即面向对象的观点,认为计算机软件的结构应该与所要解决的问题结构一致,而不应与某种分析方法保持一致。经验表明,对任何软件系统而言,其中最稳定的成分也许就是那些问题域中的成分。由此,面向对象方法把客观世界中的对象作为软件系统中的基本成分,并认为客观世界是由对象组成的,对象有其自己的属性和活动规律;对象之间的相互依赖和相互作用,构成了现存的各式各样的系统;并在构造软件系统中充分运用人类认识客观世界、解决实际问题的思维方式和方法。

面向对象方法起源于面向对象编程语言。20世纪60年代后期,Simula-67语言中出现了类和对象的概念,类作为语言机制用来封装数据和相关操作。70年代前期,A. Kay在Xerox公司设计出了Smalltalk语言,并于1980年推出了商品化的Smalltalk-80,标志着面向对象的程序设计已进入实用阶段。随后,出现了一系列面向对象编程语言,如C++,Object-C,Clos,Eiffel等。自80年代中期到90年代,面向对象的研究重点已经从语言转移到需求分析与设计方法,提出了一些面向对象开发方法和设计技术。其中具有代表的工作有:B. Henderson-Sellers和J. M. Edwards提出的面向对象软件生存周期的"喷泉"模型及面向对象系统开发的框架;G. Booch提出的面向对象开发方法;P. Coad和E. Yourdon提出的面向对象分析(OOA)和面向对象设计(OOD);J. Rumbaugh等提出的OMT方法;OMG推出的UML等。这些方法和语言的提出,标志着面向对象方法逐步发展成为一类完整的方法学和系统化的技术体系。

在本章中,首先讲述构造面向对象系统模型所涉及的主要概念及其表示法,随后给出两种面向对象开发的过程指导:一种是基于特定活动组织的,一种是基于统一软件开发过程的。最后,简单介绍一种特定的面向对象方法OSA。并且,为了准确地表述每一概念,本章中的内容尽量采用有关标准中的语言,由此也会出现一些"向后引用"问题。

5.1　概念与表示法

5.1.1　类图

在用面向对象方法建模时,类图是最常用的图。类图用以表示模型的静态结构,即表示静态元素及其之间各种静态关系。

为了控制信息组织的复杂性,可以把类图组织成包(具体可参见本节12),但这并不表示

对构成模型的基本元素的划分。

本节主要讨论类图中的类、类的内部结构以及类之间的关系。

1. 类

(1) 语义

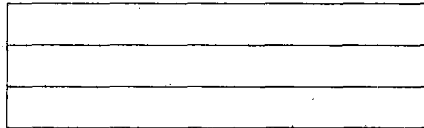
类表示正被建模系统的一个概念。类是具有相同结构、行为和关系的一组对象的描述符。类本身具有数据结构、行为以及与其他元素的关系。

类是在类图中声明的,但在其他模型图中可以按文字的方式引用之。

类名的作用域是声明它的包。在类所属的包中,所有类名都是惟一的。

(2) 表示法

用一个被水平线划分成三个分栏的实线矩形表示类。如:



其中,在最上面的那个栏给出类名,称之为名称栏;在中间的那个栏给出属性列表,称之为属性栏;在最下面的那个栏给出操作列表,称之为操作栏。除了这三个栏外,用户还可以增加其他栏,来表示预定义的或者用户自定义的值,例如责任、规则或历史更改纪录等。

在属性栏和操作栏中,可以使用性质串来表示属性和操作的性质。性质串是用花括号括起来的一个字符串,并作为一个列表元素。性质串的作用域为:该性质串的所有后继的列表元素,直到另一个作为列表元素的性质串出现为止,或到达它所在的分栏的底部。这等价于对每个列表元素分别附上一个性质串。

性质串可用于各种模型元素。

一般情况下,不需要对名字栏进行命名,也不需要对属性栏和操作栏进行命名;而对用户自定义的栏,可以对其命名,以表明它的类型。如果给出栏的名字,则可用特定的字体把栏名显示在一个栏的顶部中间。例如,图 5.1 是一个具有四个栏的类,其中三个栏有名称。

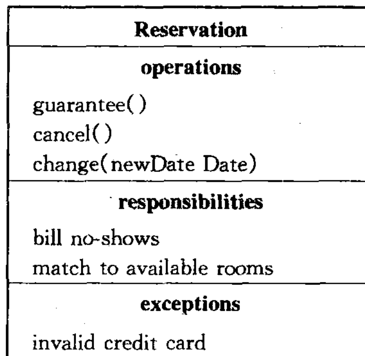


图 5.1 类 Reservation 的表示法

对于属性栏、操作栏或用户自定义的栏,可以按某种次序给出列表元素。例如,可以按字母排序,按某种性质排序或按可见性排序(公共的、受保护的、私有的)。

在默认的情况下,一个包中所给出的类就被认为是在那个包中定义的。若使用在其他包

中定义的类，则需在该类名前填加包名，其语法为：

包名::类名

其中，通过使用双冒号"::"将包名与类名分隔。

例如，图5.2中的Banking::CheckingAccount表示类CheckingAccount出现在包Banking中。类Deposit有两个属性，它们是time和amount。time的类型为DateTime包中的类Time，amount的类型为Currency包中的类Cash。

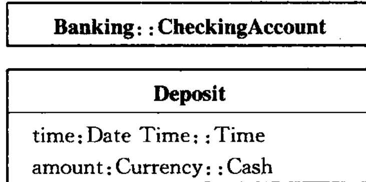


图5.2 在其他包中的类的路径名

(3) 表示选项

在有些情况下，可以显示部分属性和操作，或不显示属性或操作。对于用户自定义的栏，可以用来表示用户定义的模型性质(例如，表示业务规则、责任、变体、事件处理、异常的产生等)。用户定义的栏大多数只是一些简单的串列表，用户可自己规定其格式，但最好根据栏的内容定义之，必要时可以使用栏名。

(4) 风格指导

① 类名使用黑体字，位于分栏中央；

② 关键字用普通格式的字体，位于书名号《》内，并放在类名的上方，位于栏的中央；

③ 类名以大写字母开始；

④ 属性和操作用普通格式，左对齐；

⑤ 属性和操作以小写字母开始；

⑥ 抽象类(没有实例的类)或者抽象操作的特征标记以斜体字表示。

(5) 示例(见图5.3)

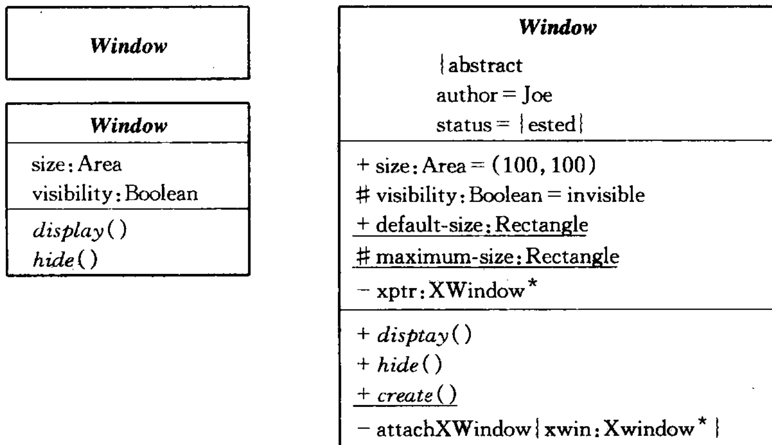


图5.3 类的三种表示法

左上方的类图符没有显示细节，左下方的类图符显示分析级细节，右边的类图符显示实现

级细节。(注:关于细节的含义可参见以后有关小节)

2. 属性

(1) 语义

属性是类的一个命了名的特性,它描述具有该特性的实例可以取值的范围。属性的类型是类型表达式,例如 array[String] of Point。具体的规约或所使用的编程语言支持的表达式语法规定了属性类型。

(2) 表示法

属性表示为具有一定语法格式的文本串。其默认的语法是:

可见性　名称 [多重性]:类型表达式 = 初始值 {性质串}

其中:

① 可见性:其值可以为

+ 公有的

受保护的

- 私有的

当把可见性作为属性栏中的性质串时,使用关键字 public, protected 和 private,分别表示公有的、受保护的和私有的。

② 名称:是表示属性名字的标识串。

③ 多重性:表示属性的多重性(见本节 9)。多重性是可以省略的,在这种情况下,多重性是 1..1。

④ 类型表达式:是属性实现类型的规约,与具体实现语言有关。

⑤ 初始值:是与语言相关的表达式,用于为新建立的对象赋予初始值。初始值是可选的(此时等号也被省略)。对象的构造函数可以参数化或者修改默认的初始值。

⑥ 性质串:表示应用到该元素的性质值。性质串是可选择的(若不指定特性串,则省略花括号)。性质串位于属性字符串之后。

如果在属性名和类型表达式之下给出下划线,则表示它们是类范围的属性,即该类的所有对象共享的属性,否则属性是实例范围的。

性质"{frozen}"表示属性是不可以改变的;如果没有给出性质"{frozen}",则表示该属性是可以改变的。

可以在属性名之后用[]中的多重性指示符来表示多重性。例如:

colors[3]:Color

points[2..*]:Point

注意,如果多重性是 0..1,就有可能出现空值。例如,下面的声明允许字符串 name 为空值或空串:

name[0..1]:String

(3) 风格指导

属性名通常以小写字母开头。用普通字体书写属性名。

(4) 例子

如下是对几个属性的说明:

+ size:Area = (100,100)

visibility: Boolean = invisible

+ default-size: Rectangle

maximum-size: Rectangle

- xptr: XWindowPtr

3．操作

(1) 语义

操作是类的实例被要求执行的服务。它有名字和参数列表。

(2) 表示法

操作表示为具有一定语法格式的文本串。其默认的语法是：

可见性　名字(参数列表)：返回类型表达式 {性质字符串}

其中：

① 可见性：其值可以为

+ 公有的

受保护的

- 私有的

当把可见性作为操作分栏中的性质串时，使用关键字 public，protected 和 private，分别表示公有的、受保护的和私有的。

② 名称：是标识符串。

③ 返回类型表达式：是操作的实现类型或者操作的返回的值类型的规约，它与具体的实现语言有关。如果操作没有返回值(例如 C++ 中的 void)，就省略冒号和返回类型。当需要表示多个返回值时，可以使用表达式列表。

④ 参数列表：是以逗号分隔的形参列表，各形参的语法为：

　类型　名称：类型表达式 = 默认值

(i) 类型是 in，out 或 inout，分别用以表示输入、输出和既可以输入又可以输出，默认的是 in；

(ii) 名称是形参的名字；

(iii) 类型表达式是实现类型的(与语言有关)规约；

(iv) 默认值是可选的值表达式，用最终的目标语言表示。

⑤ 性质字符串：指明应用于元素的性质值。性质字符串是可选的(若没有指定特性串，则省略括号)。性质串列表在整个操作串的后面。

可以省略全部的参数列表和返回类型，但不准只省略其中的一部分。另外，操作的特征标记串的语法可以遵循特定的编程语言，比如 C++ 或 Smalltalk。

通过对名字和类型表达式串加下划线方式，表示对类范围的操作。实例范围的操作是默认的，对其不用标记。

不改变系统状态(没有副作用)的操作用性质"{query}"指定；否则，操作可能更改系统的状态。

操作的并发语义用形式为"{concurrency = name}"的性质串指定，其中的 name 是下列之一：sequential，guarded，concurrent，分别表示操作是顺序的、受监护的和并发的。可以采用缩写方式，直接在性质串中用这三个值表示相应的并发值。在没有对并发语义进行详细说明的

情况下，则假设操作是顺序执行的。

在类的继承结构(见泛化)上层的类中，对没有实现的操作(即没有提供方法)，要用“{abstract}”标记，或者把操作的特征标记写成斜体，以表示它是抽象的。若该操作的特征标记在它的子类中出现，且没有带{abstract}，就表明它的子类给出了这个操作的方法。

可以把方法的具体文字描述或算法放在依附操作条目的注释中，并把注释依附到操作上，有关对注释的解释，参见“本节 14. 注释”。图 5.4 是一个示例。

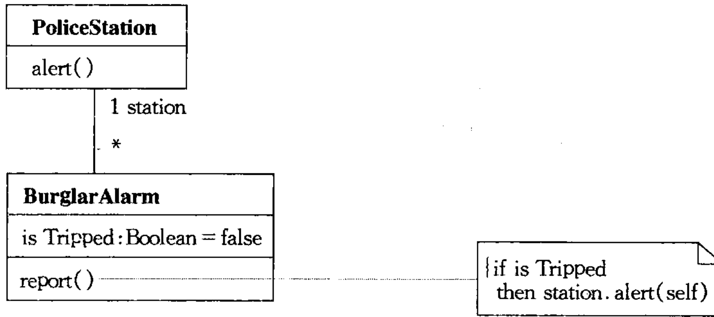


图 5.4 在注释内表示方法体

如果类的对象接收或者响应一个给定的信号，用带有关键字《signal》的操作表示这个类接收给定的信号，其语法和操作是一样的。注意，信号是实例之间异步传送的消息的规格说明。对象对接收的信号的反应用一个状态机表示。这种表示法也能表示一个类的对象对错误条件和异常的反应，这时应该把这些错误条件和异常建模为信号。

(3) 风格指南

操作名通常以小写字母开头，用普通字体书写。可以把抽象操作表示成斜体。

(4) 例子

如下是对几个操作的说明：

\+ *display*()：*Location*

\+ *hide*()

\+ *create*()

\- attachXWindow(xwin：Xwindow*)

4. **对象**

(1) 语义

对象是类的一个特定实例。它有标识和属性值。

(2) 表示法

对象的表示法是从类的表示法衍生出来的，即用具有两个栏的矩形表示。但是要在对象名之下给出下划线。参见图 5.5。

在顶部的栏中给出对象的名字和它属于的类，并加有下划线，其语法为：

对象名：类名

如果需要，在类名之前可以加上该类所属于的包，即给出该类的全路径名。例如：displywindow：WindowingSystem：：GraphicWindows：：Window，包名前边的类名和最前面的包名用冒号分隔，包名之间用双冒号分隔。

为了表示对象是多个类的实例，把对象所属于的类的名字用逗号分隔，形成一个列表。但这样的对象必须满足以下规则：在一个时刻只属于一个实现类，但允许有多个类型。

在第二个栏中给出对象的属性以及属性值的列表。每行使用的语法为：

属性名：类型 = 值

其中：① 在属性的声明中，可以省略类型；② 值为文字值；③ 用户可以自己定义文字值表达式的语法，或使用某种编程语言规定的语法。

(3) 表示法选项

可以省略对象的名字。在这种情况下，需要保留冒号和类名。这表示给定类的一个匿名对象。

可以不显示对象的类（和冒号一起），也可以不显示整个属性栏，还可以不显示对其值不感兴趣的属性。

(4) 风格指南

可以在类图中表示对象，用以表示类的数据结构示例。

(5) 例子（见图 5.5）

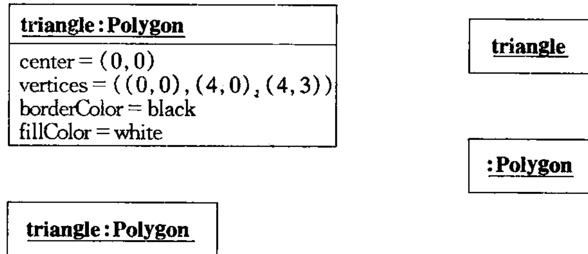


图 5.5 对象的几种表示法

左上方的对象符表明，类 Polygon 的对象 triangle 有 4 个属性，且每一属性均具有值，但没有操作。左下方的对象符表明，类 Polygon 的对象 triangle 没有给出属性信息。右上方的对象符仅表明有一个对象，其名字为 triangle。右下方的对象符表明该对象是匿名的，是类 Polygon 的对象。

5. **接口**

(1) 语义

接口描述类、构件或者子系统的外部可见操作，并不描述内部结构。以下仅以类为例讲述接口。通常，一个接口仅描述一个特定类的有限行为。接口没有实现，接口也没有属性、状态或者关联，接口只有操作。接口只可以被其他类使用，而其本身不能访问其他类。可以对接口使用泛化关系。接口在形式上等价于一个没有属性、没有方法而只有抽象操作的抽象类。

(2) 表示法

可以用带有栏和关键字《interface》的矩形符号来表示接口。在操作栏中给出接口支持的操作列表。因为接口的属性栏总是空的，所以可以把它省略。一般情况下，接口中操作的实现是由特定的类提供的，通常把这种实现关系显示为带有实三角箭头的虚线；并用带有《use》标记的虚线箭头，表示在箭头尾部的类使用或者需要接口提供的操作（参见图 5.6）。

也可以把接口表示成小圆圈。接口名放在小圆圈的下面，并用实线把圆圈连接到支持它的类上。这意味着这个类要提供在接口中的所有操作，其实类提供的操作可能要更多。如果

需要显示接口的操作列表，就不能使用圆圈表示法，而应该使用矩形表示法。可以用指向圆圈的虚线箭头，表示使用或者需要接口提供的操作。

上述的虚线箭头还意味着在箭头尾部的类使用接口，但它并不一定需要使用接口给出的全部操作。

(3) 例子

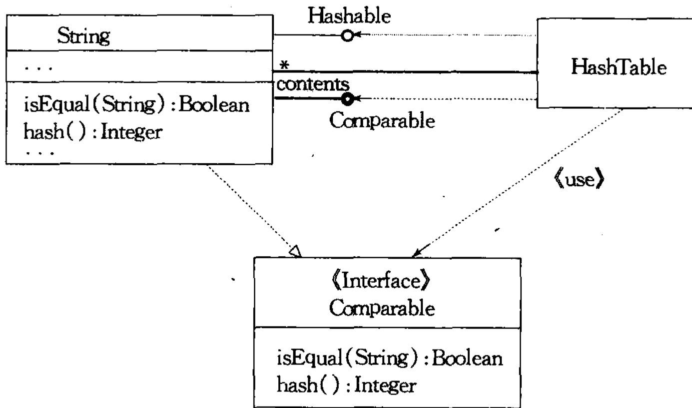


图 5.6 图中的接口表示法

该图表明，类 String 支持接口 Hashable 和 Comparable，而类 HashTable 使用接口 Hashable 和 Comparable。

6. 关联

对象之间可能存在有引用关系，把这种引用的元组称为链。把对一组具有相同结构特性、行为特性和语义的链的描述称为关联，它代表类的实例(对象)间的一组链。简言之，关联是两个或多个类之间的一种语义关系，分为二元关联和多元关联。关联不仅适用于类，也适用于接口、信号、用况和子系统。以下仅以类为例，讲述关联。

(1) 二元关联

① 语义

二元关联是两个类之间的关联。这两个类可以是同一个类，把这种情况称为自身关联。

② 表示法

把二元关联表示成连接两个类符号的实线路径，两个路径端点可以连接到相同的类，但是端点是不同的。路径可以由一条或者多条相连接的线段组成。

把连接到类的关联的端点称为关联端点，关联所具有的性质都依附在关联端点上。在(2)中详细地讨论了关联端点。

可以把关联名、关联类符号和 Xor-关联符号依附在路径上。

(i) 关联名

关联名是可选的。如果使用关联名，就把它表示成放在路径上的名字串，但不能太靠近端点，这样会与角色名混淆。名字串前可以有一个小的黑实心三角形，它是可选的。三角形的顶点指示从哪个方向读名字，它没有实际语义意义，纯粹是描述性的。按这样的箭头所指出的方向对关联中的类进行排序。

性质串可以放置在关联名的后面或者下面。

(ii) 关联类符号

有一些关联具有像类那样的性质，比如属性、操作和其他关联，把这样的关联称为关联类。有关关联类的细节，在(4)中详细描述。

把关联类表示成一个用虚线连接到关联路径的类符号，可以把关联类符号从路径上拖开，但是虚线必须连接路径和类符号。关联路径和关联类符号表示同一基础模型元素，它们的名字相同。名字可以放置在路径上或类符号上(但它们的名字必须相同)。

(iii) Xor-关联符号

具有一个公共类的两个或多个二元关联之间可能存在异或约束，把这种结构称为 Xor-关联。Xor-关联表明：一次仅可在多个潜在的关联中实例化一个关联。把 Xor-关联表示成连接两个或者多个关联的虚线，虚线上标有约束串"{xor}"。公共类的任何实例一次只可以参加一个关联，且每个角色名必须是不同的。

③ 例子(见图 5.7)

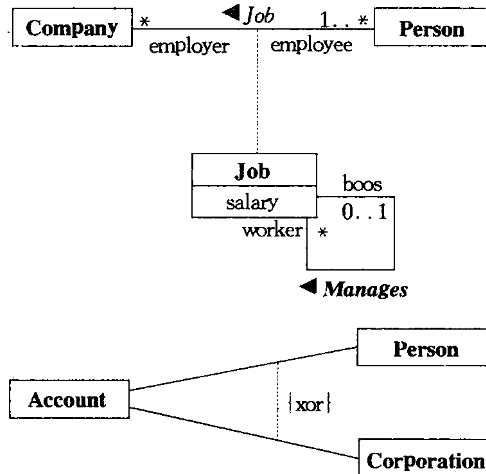


图 5.7 关联表示法

在上图中，有一个名为 Job 的关联类，并且该关联类还有一个自身关联 boss。下图为一个 Xor-关联。

(2) 关联端点

① 语义

关联端点就是连接类的关联的端点，它是关联的组成部分，并不是一个独立的元素。每个关联有两个或多个关联端点。

② 表示法

关联路径通过端点与类符号相连。可能在关联路径的每个端点上都附属有描述信息，这些信息表明与类相连接的关联的性质。附属的信息属于关联符号，而不属于类符号。一般地，这些信息放在端点附近，包括：多重性(参见本节中的 9)、排序、限定符(参见(3))、导航性、聚合符、角色名、接口说明符、可变性和可见性等。其中：

(i) 多重性

多重性由特定语法给出。对特殊的关联(如 N 元关联)或在不完整的模型中，可以不指定

模型中的多重性。

(ii) 排序

如果多重性比 1 大，相应的元素集合可以是有序的，也可以是无序的。如果没有指明多重性，相应的元素形成集合就是无序的。在关联端点上，可以通过关键字“{ordered}”来指定对元素的排序。但这并没有指定排序是怎样建立和保持的，仅仅表明元素是有序的，在实现时可以按各种方式排序。

(iii) 导航性

箭头附属到关联路径的端点表明，朝着连接到箭头的类的方向，支持导航。箭头可以附属 0 个、1 个或 2 个路径端点。

(iv) 聚合符

聚合是一种特殊的关联，表示整体类和部分类之间的“整体 - 部分”关系，其中的整体类称之为聚集。聚合符是一个空心菱形，它附属在表示聚合的关联路径的那一端，即附属在为聚集的那个类上，不能附属在路径的两端。聚合是可选的。如果菱形是实心的，那么它就代表一种称为组合的强形式聚合，有关组合的内容在“组合”那一节中进行了详述。

(v) 角色名

角色名是靠近关联路径端点的名称串。它表示在其附近的与关联路径端点相连接的类所扮演的角色。角色名是可选的。

(vi) 接口说明符

接口说明符的语法为：

[角色]:[接口名],...

例如，worker: employee。其中，worker 为角色名，该角色需要接口 employee 提供的操作。可见，接口说明符说明了在一个关联的实现中，一个实例(角色)所要求的由接口表达的行为。一般情况下，接口提供的操作是提供该接口操作的那个类所表达的操作的一个子集。

(vii) 可变性

性质串{frozen}表明：在对象被产生和初始化后，不能加入、产生或移走从该对象出发的链。性质串{addOnly}表明：可以加入另外的链，然而不能调整或删除链。如果链是可变的(能加入、删除和移走)，就不需要标识信息。

(viii) 可见性

在角色名的前面加可见性说明符(+, #, - 或{public}, {protected}, {private})。可见性指明了朝着给定角色名方向的关联的可见性。

③ 表示选项

如果同一聚集有两个或多个聚合，通过将聚合端点合并成一段，可以把它们画为一棵树。这要求在聚合端点上的所有的附属信息是一致的。这纯粹是一种可选的表示法，没有什么附加的语义。

④ 风格指南

如果在一个单关联端点上有多个信息，应以下面的顺序表示它们，从与类相连的路径的端点到路径的纵深依次是：限定符、聚合符和导航箭头。

角色名和多重性应放置在路径端点的附近，这样不会与其他的信息相混淆。角色名和多重性可以放在同一关联端点的两侧，或是放在一起(例如：“ * employee”)。

⑤ 例子(见图 5.8)

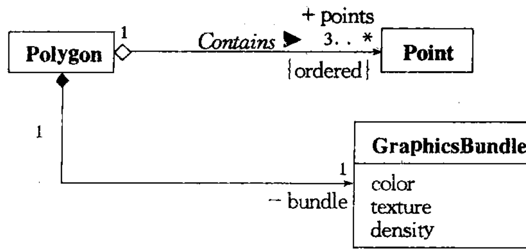


图 5.8 关联角色上的多种信息

本图中有两个关联,一个是聚合,另一个是组合。聚合(组成)端的类 Polygon 能访问类 Point 和类 GraphicsBundle。

(3) 限定符

① 语义

一个限定符是一个属性或是属性表,这些属性的值将与一个对象相关联的对象集做了一个划分。限定符是关联的属性。

② 表示法

把限定符表示成一个小的矩形,该矩形依附到一个关联路径的端点,它位于最终路径和它连接到的类的符号之间。限定符矩形是关联路径的一部分,而不是类的一部分。限定符附加在关联的源端点。源类的实例,同限定符的值一起,惟一地对在关联另一端的目标类实例的集合分类(例如,每一个目标都精确地落入一个分类)。

与目标端点相连的多重性表示,由源类的实例和限定符的值共同选择的目标实例集的可能基数。通常的值包括:

(i) "0..1"

对每一个可能的限定符的值,所选择出的目标实例都是惟一的,但也可能没有对应的目标实例。

(ii) "1"

对每一个可能的限定符的值,所选择出的目标实例都是惟一的;因此限定符的值域必须是有限的。

(iii) "*"

限定符的值是一个索引,它将目标实例划分为子集。

在限定符框内放限定符属性。如果有一个或多个属性,就在每行中放一个。除了限定符属性无初始值外,限定符属性和类属性表示法相同。

允许一个关联的每一端都具有限定符,但这种情况很少见。

③ 例子

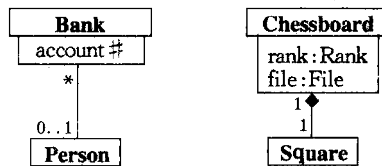


图 5.9 限定关联

左图的限定符有一个属性 account #，它与类 Bank 一起表明：在一个银行中，一个账户对应一个人员，或没有对应人员；而一个人员可能在一个或多个银行开设一个或多个账户，也可能没有开设账户。右图的限定符有两个属性，它们与 Chessboard 一起确定了 Square，且 Square 是其组成部分。

(4) 关联类

① 语义

一个关联类是一个关联，该关联也有类的一些性质；也可以说一个关联类是一个类，该类具有关联的性质。尽管把一个关联类画成一个关联和一个类，但它仍然是一个单一的模型元素。

② 表示法

把关联类表示成用一条虚线连接到关联路径的类符号(矩形)，见图 5.10。类符号里的名字和该类符号所依附的关联路径上的名字串应该是一致的。在关联路径的两端可能都具有通常的附属信息，类符号也可以具有通常的内容，但在虚线上没有附属信息。

注意，关联路径和关联类是一个单独的模型元素，并且名字相同。名字可以出现在路径上或类的符号中。如果一个关联类只有属性而没有操作或其他的关联，名字可以显示在关联路径上，并且从关联类符号中省去，以强调其“关联性质”。如果它有操作和其他的关联，那么可以省略路径中的名字，并将他们放在类的矩形中，以强调其“类性质”。两种情况的实际语义是相同的。

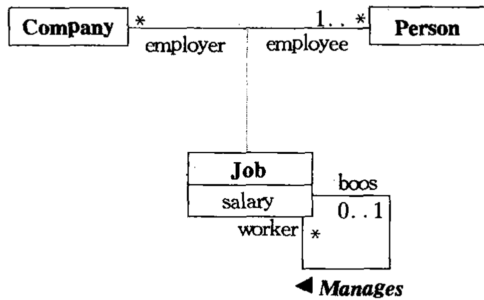


图 5.10　关联类

(5) N 元关联

① 语义

一个 N 元关联是三个或多个类之间的一个关联，其中的一个类可能在一个 N 元关联中多次出现。该关联的每一个实例是一个 N 元组，即它由 N 个分别来自相关的类的对象组成。二元关联是 N 元关联的特殊情况，二者都有自己的表示法。

可以规约 N 元关联的多重性，但与二元关联的多重性相比，并不那样明显。在一个角色上的多重性是指，当该 N 元关联中的其他 N－1 个类的对象被确定时，该关联潜在的实例元组的数目。

对于一个 N 元关联，在任意角色上都不能有聚合标志。

② 表示法

通过一个大的菱形表示一个 N 元关联，这个菱形有很多与各参与的类相连接路径。如果

关联有名字,就显示在菱形附近。同二元关联一样,角色可以显示在每一条路径上。可以指出多重性;然而不允许有限定符和聚合符。

可以用虚线把关联类符号与菱形连接起来,表示具有属性、操作或关联的 N 元关联。

③ 例子

本例描述了一个具有一个特定守门员的球队在每一个赛季的记录(见图 5.11)。假设在本赛季可出售守门员,且该守门员可出现在多个球队。

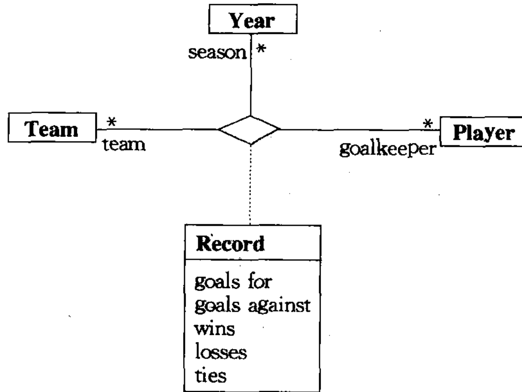


图 5.11 三重关联(也是关联类)

7. 组合

(1) 语义

组合是一种关联,是聚合的一种形式。其部分和整体之间具有很强的"属于"关系,并且它们的生存期是一致的。这种聚集(也称为组成)端的多重性不能超过 1。

在一个组合中,其部分可以包含一些类和关联,如果需要的话,也可以把它们规约为关联类。在一个组合中,一个关联的意义为:由一个链所连接的对象而构成的任何元组,必须都属于同一个容器对象。

(2) 表示法

可以通过用实心菱形作为关联端点的附属,表示组合。为了在很多情况下更方便地表示组合,也可以采用图形嵌套表示法。

可以通过依附在关联上的实心菱形表现组合,与之相连的元素是整体。可以按正常的方式显示多重性。

如果采用图形嵌套表示法,就要把表示部分的元素放在表示整体元素的符号内部。像类那样的嵌套元素在其组成元素内可以有多重性。把多重性表示在作为部分的元素符号的右上角。如果省略多重性的标志,那么缺省的多重性就是"多"。这样就表示了在组成类中作为部分的多重性。嵌套元素在组成中可以有角色名。名字显示在它的类型前面,其语法为:

角色名:类名

这代表在该组合关联内的角色名。完全画在组成边界内的关联被认为是组合的一部分。任何在其单个链上的实例必须来自同一个组成。一个其路径穿越了组成边界的关联就不被看作是组合的一部分,其中在单个链上的任何实例可以来自相同或不同的组成。

(3) 例子(见图 5.12)

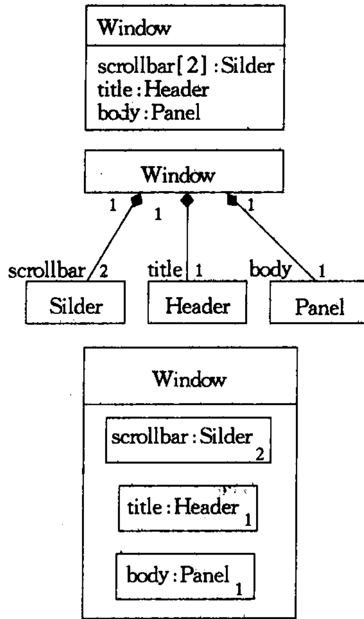


图 5.12　表示组合的不同方法

本图给出了三种表示组合的方法。这些图都表明了类 Window 由类 Silder(角色为 Scrollbar),Header(角色为 title)和 Panel(角色为 body)组成,且它们同生共死。

8. 链

(1) 语义

链是对象引用的元组(列表)。在最常见的情况下,它是一对对象引用。它是关联的一个实例。

(2) 表示法

把二元链表示为两个实例之间的路径。可以把一个实例与它自身之间的链表示为一个具有单一实例的环。

在链的各端可以表示角色名。链没有实例名,从与它们相关的实例中得到它们的标识。多重性不能显示在链上,因为链是实例。其他的关联附属物(组合、聚合和导航)可以表示在链端点上。

限定符可以表示在链上,限定符的值也表示在它的框中。

用带有连向各个参与实例的路径的菱形表示 N 元链。N 元链的其他内容与二元链相同。

(3) 例子(见图 5.13)

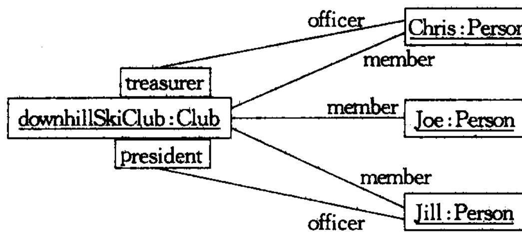
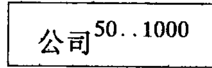


图 5.13　链

9. 多重性

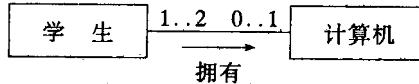
(1) 语义

多重性规约了一个集合可以假定的、可允许的基之范围。例如：可以用多重性规约一个类所包含的对象的数目范围。



其中，50..1000 为多重性，它指出公司(类)的职员数目范围。

再例：可以用多重性规约在一个关联中参与相应关系的对象数目范围。



其中，1..2 和 0..1 为该关联的多重性，“1..2”指出每一学生拥有 1 台计算机，最多拥有 2 台计算机；“0..1”指出每一计算机或属于一个学生，或不属于任何学生。

可见，多重性规约是非负整数开集的一个子集。

(2) 表示法

把多重性表示成由用逗号分开的整数间隔序列组成的字符串，间隔代表整数的范围(可能无限)，其格式为：

下限..上限

其中的下限和上限都是文字整型值，说明从下限到上限的整数闭区间。此外星号(*)可以用于上限，表明不限制上限。

如果指定的是单个的整型值，那么整数范围就为单个的整数值。

如果多重性规约由单个的(*)构成，那么它就表明了无穷的非负正整数的范围，也即它等价于 0..*。

0..0 多重性没有实际的意义，它表明没有实例能产生。

(3) 风格指南

间隔最好单调递增。例如：“1..3,7..10”比“7..10,1..3”更合适。

两个相连的间隔应该合并为一个间隔。例如：“0..1”比“0,1”更合适。

(4) 例子

0..1　　1　　0..*　　*　　1..*　　1..6　　1..3,7..10,15,19..*

10. 泛化

(1) 语义

泛化是一般元素(父亲)和特殊元素(儿子)之间的一种分类关系，其中特殊元素完全与一般元素一致，并附加了一些信息。泛化可用于类、包、用况、关联和其他元素，此处仅使用类描述泛化。

(2) 表示法

把泛化表示成从子类(特殊类)到父类(一般类)的实线路径，在一般元素的路径端有一个空心三角。

泛化路径上可以有一个称为区分器的文本标记，它是对父类的子类的划分的命名。在给定的划分下声明子类。没有区分器标记，表明区分器为空。

可以使用如下关键字来表示对子类的语义约束(参见图 5.15)。一个放在括号内的用逗号间隔的关键字表，或者放在共享的三角形的附近(如果几条路径共享一个三角形的话)，或者放在横穿所有涉及到的泛化线的虚线的附近。这些关键词是：

① overlapping(重叠)

作为“祖先”的元素可以在集合中有两个或多个子类。一个实例可以是两个或更多个子类的直接或不直接的实例。

② disjoint (不相交)

作为“祖先”的元素不可以有两个或更多的子类。没有实例可以是两个或更多子类的直接或不直接的实例。

③ complete (完全)

已经说明了所有的子类,不再期望有另外的子类。

④ incomplete(不完全)

已经说明了一些子类,但知道这是不完全的,还有其他的子类没有被说明。

在给定父类的属性和关联角色当中,区分器必须是惟一的。相同的区分器名字可以多次出现,表示子类属于同一划分。

(3) 表示选项

可以把对给定的父类的一组泛化路径表示成一棵树,父类位于树根(三角形指向它)。

如果一个文本标志出现在由通向子类的泛化路径所共享的树的主干上,则相当于标志出现在所有的泛化路径上。换句话说,所有的子类共享给定的性质。

(4) 例子

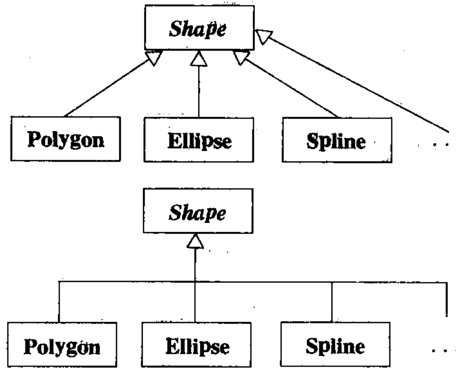


图 5.14 泛化的表示法

图 5.14 中,上图为分离表示法,下图为共享表示法。

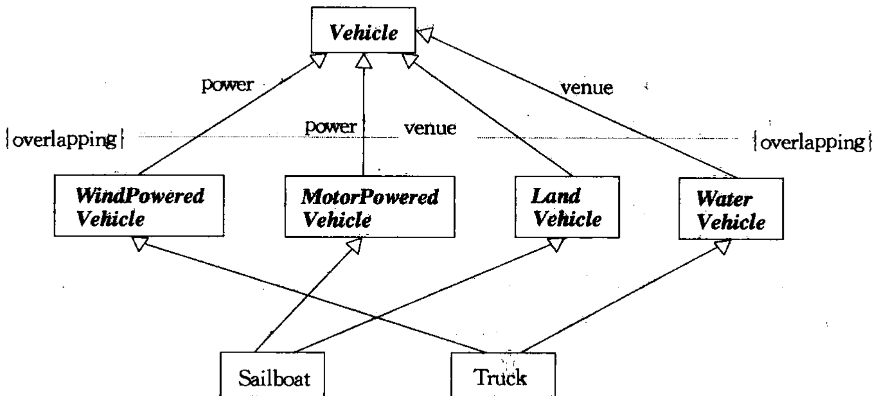


图 5.15 具有区分器和约束的泛化(分离表示法)

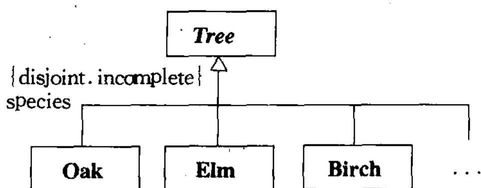


图 5.16　使用共享表示法的泛化

11. 依赖

(1) 语义

一个依赖规约了两个模型元素(或两个模型元素集合)之间的一种语义关系。就依赖的意义而言,与模型元素本身有关,并不需要一组实例。依赖指出了这样一种情况,即对目标元素的改变可能需要改变该依赖中的源元素。

(2) 表示法

把依赖表示为两个模型元素之间的虚线箭头。在箭头尾部的模型元素(客户)依赖箭头头部的模型元素(提供者)。

可能会有一组元素作为客户或提供者。在这种情况下,把一个或多个尾部在客户端的箭头,连接到头部在提供者端的一个或多个箭头的尾部。如果需要,可以在连接处放置小的圆点,作为连接点。依赖的注释应该依附在连接点上。

在一些面向对象的建模语言中,预定义一些种类的依赖。其表示法为,在虚线箭头上附加放在书名号内的关键字。

注意,不带箭头的虚线用以连接元素与注释,这不是依赖。

(3) 例子(见图 5.17,图 5.18)

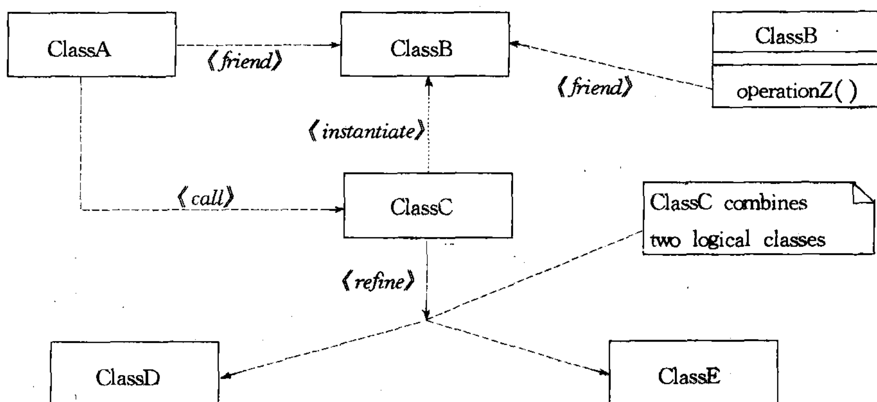


图 5.17　类间的各种依赖

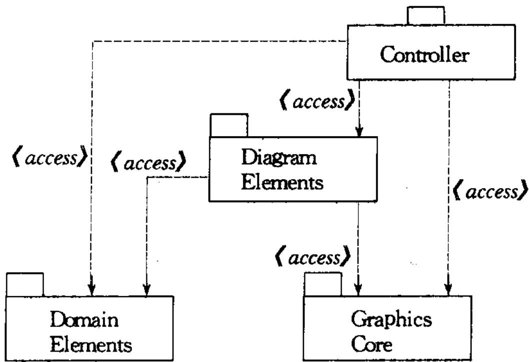


图 5.18 包间的依赖

12. 包

(1) 语义

包是模型元素的一个分组。一个包本身可以嵌套在其他包中,并且可以含有子包以及其他种类的模型元素。这样,每个元素可以直接地属于某一个包,并且包之间可以构成一个层次,且是一棵严格的树。此外,通过使用访问依赖和引入依赖,一个包可以访问和引入其他包,在“访问和引入包”中详述了访问依赖和引入依赖。包之间的其他种类依赖,通常隐含了元素间存在着的一个或多个依赖。

(2) 表示法

把包表示成一个大矩形,并且在这一矩形的左上角还有一个小矩形(作为一个“标签”),即包的符号是通常的文件夹图标(参见图 5.19,图 5.20)。

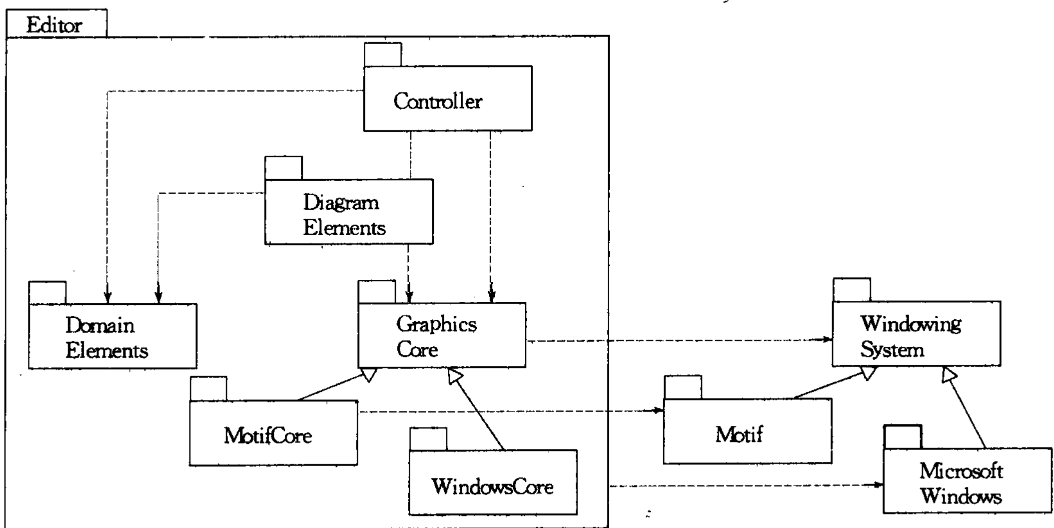


图 5.19 包及其间访问、引用和泛化关系

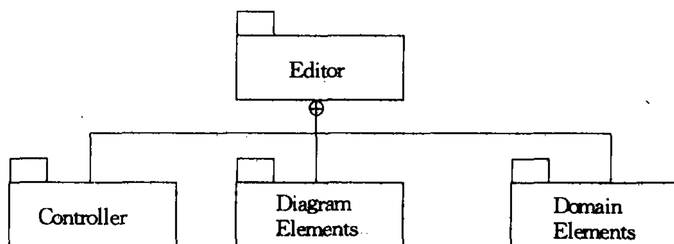


图 5.20 以树结构显示 Editor 包中的一些内容

通常在大矩形中描述包的内容,这时可以把该包的名字放在左上角的小矩形中。也可以把所包含的元素画在包的外面,但要用多条线段把这些元素与该包连接,并在连接到该包处画一个内含加号(+)的圆,这时可以把该包的名字放在大矩形中。用花括号括起来的性质串,放在包名的后边或下边。

一个包中的元素在包外的可见性,可以通过在该元素名字前加上一个可见性符号(+:公共的,-:私有的,#:受保护的)来指示。

可以在包符号间绘制关系,以显示该包中一些元素间的关系。两个包之间的访问依赖和引入依赖,被绘制成带有箭头的虚线,其上分别标有串《import》和《access》。

13. 访问或者引入包

(1) 语义

一个元素可以引用其他包中的元素。在包的层次上,访问依赖表明:目标包的内容可以被客户包引用,或被递归嵌套在客户包中的其他包引用。就引用而言,目标包必须具有适当的可见性,即对一个不相关的要访问它的包,目标包应有可见性“public”;对目标包的子孙,目标包应有可见性“public”或“protected”,或对嵌套在目标包内部的包,目标包应有任意的可见性(对于这种情况,不需要访问依赖)。如果在提出访问的那个包中还存在包,那么嵌套在其中的包能得到与外层包同样的访问。

注意,一个访问依赖,并不修改客户的名字空间,或以任何其他方式自动创建一些引用;访问依赖仅仅准予建立一些引用。

引入依赖将被访问包(箭头指向的包)中那些具有合适可见性的名字引入到访问包(即对它们的引用可以不需要一个路径名)。

(2) 表示法

把访问依赖表示成从客户包到目标包的依赖箭头。箭头上带有串《access》(参见图5.21)。这个依赖表明客户包中的元素可以合法地引用目标包中的元素。引用必须满足由提供者指定的可见性约束。注意,依赖并不自动创建任何引用,它只是允许创建引用。

引入依赖除了箭头上的串为《import》外,具有和访问依赖一样的表示法。

(3) 例子

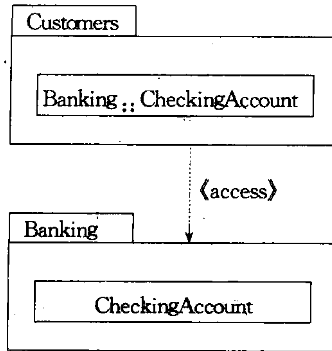


图 5.21 包间的访问依赖

14. 注释

一个注释是一个图形符号，该符号包含了一些文本信息(可能也包含一些嵌入的图像)，直接附属于一个模型元素(见图 5.22)。

一个注释可以把任意的文本信息附属到任一假定重要的模型元素上，但这样的文本信息没有语义作用。

(1) 语义

注释是一个符号项，用以表示某一语义元素的一些文本信息。

(2) 表示法

把注释表示为带有折角(右上角)的矩形。它可含有任意的文本，出现在特定的图中，并可通过虚线依附到多个模型化元素上，或单独存在。

(3) 例子

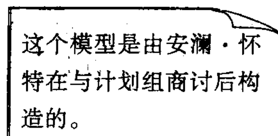


图 5.22 注释的表示法示例

5.1.2 顺序图

顺序图表示实例之间的按时间顺序排列的交互，它适合于对实时系统和复杂的场景进行详细的建模。特别是，它用实例的生命线表示参与交互的实例，并按时间顺序排列实例之间交换的消息，它不表示对象之间的关联。

一个顺序图对实例及其间的关系的识别不具有全局性，只是对一组实例及其之间的关系进行描述。

在下文中用对象代替实例，但在使用对象的地方也可以使用参与者的实例。

1. 顺序图

(1) 语义

顺序图是一种表达对象间交互的图，由一组对象及其间可发送的消息组成，强调消息之间

的顺序。在"消息"那节中对消息进行了解释。

(2) 表示法

顺序图是二维的,其中:垂直方向表示时间,水平方向表示不同的对象。

正常地,时间维由上到下(根据需要,也可以由下到上)。通常只有时间顺序是重要的,但在实时应用中时间轴是能度量的。对象的水平顺序并不重要。

(3) 表示选项

生命线之间的顺序是任意的。

可以交换轴,即时间轴水平向右,而不同的对象用水平的线表示。

可以把各种标签(例如,计时约束,对活动中的行为描述等)放在图的边缘或放在它们标记的消息的旁边。

可以在消息名上用时间表达式,表示计时约束。可以把函数 sendTime(对象发送消息的时间)和函数 receiveTime(对象接收消息的时间)应用到消息名上,以计算时间。时间函数集是可扩充的,因此用户可根据需要发明新的函数,以用于特定的情境或实现特性。

可以在图中使用一些构造标记,用于指示时间间隔,且约束可以附在其上(如图 5.23 的右下部)。在很多情况下,转换时间是可以忽略的。类似地,可以用消息名表示在计时表达式中的发送或接收消息的时间(如图 5.23 中的"b.receiveTime - a.sendTime<1 sec.")。

(4) 例子

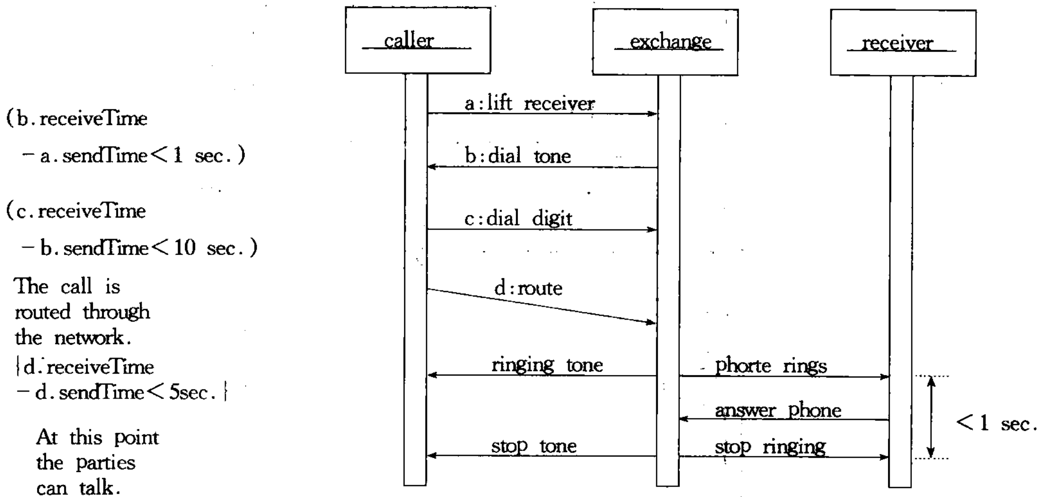


图 5.23 具有并发对象(对象表示符带有黑框)的简单顺序图

2. 对象生命线

(1) 语义

在顺序图中,对象生命线表示扮演特定角色的对象。在生命线之间的箭头表示扮演这些角色的对象之间的通信。在顺序图中表示处于一个角色状态的对象的存在和持续,但没有表示对象之间的关系。类的角色规定了对象的角色,它描述了扮演该角色的对象的性质。

(2) 表示法

把对象表示成称之为"生命线"的垂直虚线,生命线代表一个对象在特定时间内的存在。

如果对象在图中所示的时间段内被创建或者销毁，那么它的生命线就在适当的点开始或结束。否则，生命线应当从图的顶部一直延续到底部。在生命线的顶部画对象符号。如果一个对象在图中被创建，那么就把创建对象的箭头的头部画在对象符号上。如果对象在图中被销毁，那么用一个大的"╳"标记它的析构，该标记或者放在引起析构的箭头处，或者放在从被销毁的对象最终返回的箭头处（在自析构的情况下）。在图的顶部（第一个箭头之上）放置在交互开始时就存在的对象，而在整个交互完成时仍然存在的对象的生命线，要延伸超出最后一个箭头。

生命线可以分裂成两条或更多条并发的生命线，以表示条件性。这样的每一个生命线对应于通信中的一个条件分支。生命线可以在某个后续点处合并。

(3) 例子(见图 5.24)

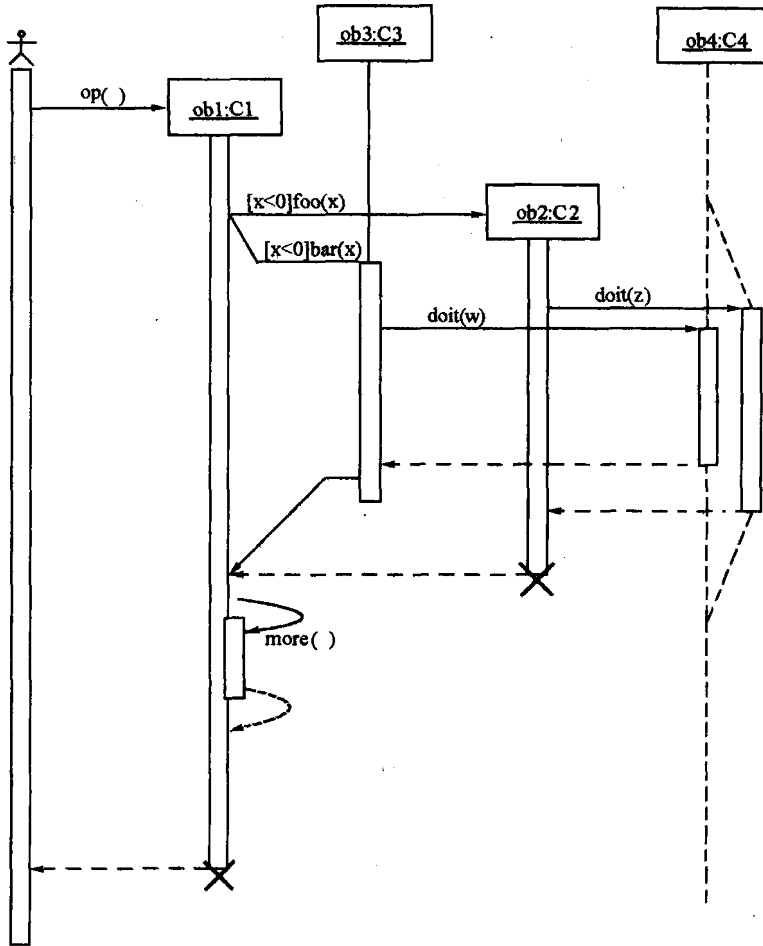


图 5.24 具有激活、条件、递归、创建和析构的顺序图

3. *激活*

(1) 语义

激活（控制焦点）表示一个对象直接或者通过从属例程执行一个行为的时期。它既表示了行为执行的持续时间，也表示了活动和它的调用者之间的控制关系。

(2) 表示法

用一个窄长的矩形表示激活，矩形顶端和它的开始时刻对齐，末端和它的结束时刻对齐。可以用文本标注被执行的动作，依赖于整体风格，或者把标注放在激活符号的旁边，或者放在图左边的空白处。在程序的控制流中，激活符号的顶端画在进入的箭头的尖端（开始该动作的那个箭头），底端画在返回的箭头的尾部。

在每个拥有自己的控制线程的对象并发的情况下，一个激活说明了一个对象执行一个操作的持续时间，与其他对象的操作并不相关。如果直接计算和间接计算（由嵌套过程执行）的区别并不重要，那么可以用整条生命线表示一个激活。

在过程性代码的情况下，一个激活表示在一个对象中一个过程是活动的或者它的从属过程（可能在其他的对象中）是活动的持续时间。换句话说，可以在一个特定的时间看到所有活动着的嵌套过程激活。在递归调用一个已激活的对象的情况下，第二个激活符号画在第一个符号稍微靠右的位置，在视觉上它们看起来像是叠起来一样（可以按任意的深度嵌套递归调用）。

4. 消息

(1) 语义

消息是两个对象间的通信，这样的通信用于传输将产生的动作所需要的信息。一个消息会引起一个被调用的操作，产生一个信号，或者引起一个对象被创建或者被消除。

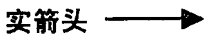
(2) 表示法

在顺序图中，把消息表示为从一个对象生命线到另一个对象生命线的一个水平实线箭头。对于对象到自身的消息，箭头就从同一个对象符号开始和结束。用消息（操作或信号）的名字及其参数值或者参数表达式标识箭头。

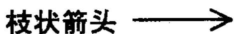
箭头也可以用一个序列数标识，以表示消息在整个交互中的顺序。序列数对于标识并发控制线程很有用处。另外，还可以用监护条件标识消息。

(3) 表示选项

可以用如下种类的箭头表示不同种类的通信：



过程调用或其他的嵌套控制流。在外层控制恢复之前，要完成整个嵌套序列。可以把它用于普通的过程调用。在某个主动对象发送信号并等待完成嵌套的行为序列时，也可以把它用于并发的主动对象。



发送者发送一个消息后，立即进行其执行的下一步，没有控制的嵌套问题，所有的消息都是异步的。

从过程调用的返回。

在控制的过程流中，可以省略返回箭头（暗示激活结束），假设每个调用在任何消息后都有一个配对的返回，并可以把返回值标示在初始的箭头上。对于非过程控制流（包括并发处理和异步消息），都应当显式地标出返回。

在并发系统中，一个实箭头表示产生一个控制线程（有等待的语义），一个枝状箭头表示发送一个消息而不产生控制（有非等待的语义）。

通常消息箭头都画成水平的。这表示发送消息所需要的持续时间是“原子的”(也即,它与交互的粒度相比是短暂的,并且在传送消息的中间不能发生任何事情)。这在很多计算机中都被假设是正确的。如果消息需要一段时间到达,且这中间可能发生某些事情(例如反向的消息),那么箭头可以向下倾斜,使箭头头部在尾部下方。

把分支画成从一个点出发的多个箭头,每个箭头由监护条件标识。依据监护条件是否互斥,这个结构可以表达条件或者并发。

可以把一组相连的箭头闭合,表示为迭代。在一般的顺序图中,迭代表示一组消息的分发可以多次发生。在一个过程中,可以在迭代的底部标出迭代持续的条件。如果存在并发,那么图中的一些箭头可以是迭代的一部分,而另一些是独自执行的。希望对图进行布局,以使迭代箭头能容易地闭合在一起。

5. 转换时间

(1) 语义

消息可以指定几个不同的时间(例如,发出时间和接受时间)。这些时间可以是用在约束表达式中的正式名称。不同种类的时间集都是可以扩展的,这样用户就可以按需要为特定的目的而发明新的时间,如 elapsedTime(占用时间)和 startExecutionTime(开始执行时间)。可以在约束中使用这些表达式,以给消息指派有效的具体时间约束。

(2) 表示法

可以赋予消息一个名字。把计时约束写成为一个基于消息名字的表达式。例如,如果消息的名字是 stim ,用 stim. sendTime()表示发送时间,用 stim. receiveTime()表达接收时间。可以把计时约束表示在与箭头对齐的图的左边上,也可以通过把布尔表达式(可能包括时间表达式)放在括号中表示约束。

(3) 表示选项

若在上下文很清楚,消息的名字或消息的名字本身可以用于指示转换开始的时间。在转换的执行不是瞬间的情况下,可以用带有附加撇号的名字指定转换结束的时间。

5.1.3 状态图

状态图用于描述像对象这样的模型元素的行为。特别是,用它描述元素的状态的可能序列和动作的可能序列,由于对具体事件(如信号和操作调用)的响应,元素在其生命期中要经历这样的状态,并执行相应的动作。

图 5.25-1 是一个示例性的状态图,它描述了一个简单的电话对象的状态转换。

1. 状态

(1) 语义

一个状态是对象(类)生命期的一个阶段,在该阶段中该对象要满足一些特定的条件,并可从事特定的活动。

在概念上,对象要在一个状态内维持一段时间。然而,这样的语义也允许对瞬时状态建模,以及对非瞬时的转换建模。

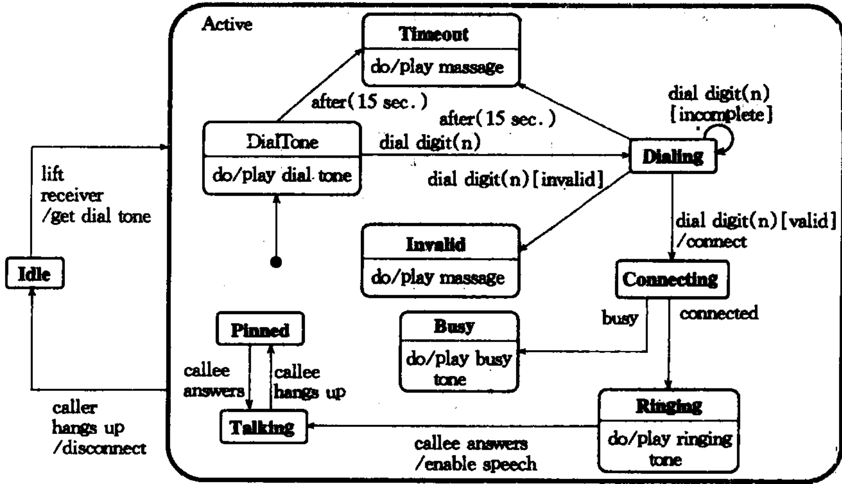


图 5.25-1　状态图示例

(2) 表示法

把一个状态表示成一个四角均为圆角的矩形(见图 2.25-1)。可以有选择地把表示状态的矩形划分成由水平线相互分隔的多个分栏：

① 名称分栏

可以在该分栏中放置状态名。在同一张状态图里不应该出现具有相同名称的状态，因为这样可能会引起冲突。没有名称的状态是匿名的，但同一张图中的匿名状态是各不相同的。

② 内部转换分栏

用该分栏给出对象在这个状态中所执行的内部动作或活动的列表。其中，动作是一个可执行的不可中断的原子计算；活动是一个非原子的执行过程。

该分栏中各项的表示法的一般格式为：

动作标号'/' 动作表达式

用动作标号标识，在该环境下要调用的由动作表达式指定的动作。动作表达式可以使用对象范围内的任何属性和链。若动作表达式为空，则可省略斜线分隔符。

如下是几个专用的动作标号，它们不能用作事件名(下节讲述事件)：

(i) entry

这个标号标识由相应的动作表达式规定的动作，在进入状态时执行该动作(进入动作)。

(ii) exit

这个标号标识由相应的动作表达式规定的动作，在退出状态时执行该动作(退出动作)。

(iii) do

这个标号标识一类活动("do 活动")，只要被建模的元素是在状态中，或没有完成由动作表达式指定的计算，就执行这个活动(后者可能导致一个完成事件的产生)。

在所有的其他情况中，动作标号标识触发相应动作表达式的事件。把这些事件称为内部转换，在语义上它们等价于自转换，只是不退出状态或再进入状态。这意味着不执行相应的退出和进入动作。书写内部转换的一般形式为：

事件名　'('用逗号分隔的参数表')'　'['监护条件']'　'/'　动作表达式

如果监护条件不同，在每个状态中同一个事件名可以出现多次。事件参数和监护条件是

可选的。如果事件有参数，就可以通过当前的事件变量把参数用在动作表达式中。

(3) 例子

下面给出表示状态的一个例子。

Typing Password
entry/set echo invisible exit/set echo normal character/handle character help/display help

2. 事件

(1) 语义

按照状态图的具体用意，事件是指可以引发状态转换的所发生的事情。事件可以分为几种(不必互斥)：

① 信号事件

一个对象对另一个对象的显式信号的接收，导致一个信号事件。对于这类事件，把事件的特征标记放由它所触发的转换上。

② 调用事件

对操作的调用的接收，导致一个调用事件。

③ 时间事件

在指定事件(经常是当前状态的入口)后，经过了一定的时间或到了指定日期/时间，导致一个时间事件。

④ 条件事件

用布尔表达式描述的指派条件变为真，就导致了一个变为真的事件。无论何时，只要表达式的值由假变成真，这种事件就发生。注意，条件不同于监护条件。无论什么时候，激发具有监护条件的事件，都对监护条件进行求值。如果求值的结果为假，转换就不发生，并且事件丢失。

(2) 表示法

可以按如下的格式定义信号事件或调用事件：

事件名 ‘(’用逗号分隔的参数列表‘)’

参数的格式如下：

参数名 ‘:’ 类型表达式

在类图中，在类符号上用关键字〈〈signal〉〉声明信号。把该关键字放在信号名的上面，把参数说明为信号的属性。用这样的方式定义可以用于触发转换的信号名。注意，信号是实例之间异步传送的消息的规格说明。可把信号指定为另一个信号的子类，这表示子事件的出现触发依赖它的任何转换或依赖它的任何祖先的转换。

可以用关键词“after”和计算时间量的表达式表示时间事件，比如“after (5 秒)”或者“after (从状态 A 退出后经历了 10 秒)”。如果没指明时间起始点，那么从进入当前状态开始计时。可把其他的时间事件指定为条件，比如“when (date= 2000 年 1 月 1 日)”。

用关键词“when”和布尔表达式表示变为真的事件。可以把其看作是连续测试条件，直到它为真，但实际上只有在值改变时才检测它。

3. 转换

(1) 语义

转换是两个状态之间的关系,表示当一个特定事件出现时,如果满足监护条件,对象就从第一个状态进入第二个状态,并执行一定的动作。把这样的状态的改变,称为“触发”转换。事件可能有参数,这样的参数可由转换指定的动作使用,也可由与源和目标相联系的退出和进入动作分别使用。在状态图中,每次处理一个事件。如果事件没有触发任何转换,就丢弃它。如果在同一个简单状态图中触发了多个转换,就只对优先级最高的那个转换点火。如果这些相冲突的转换具有相同的优先级,就随机地选择一个转换,进行触发。

(2) 表示法

把转换表示成从源状态出发并在目标状态上终止的带箭头的实线。它可以由转换串标记,如下是转换串的格式:

事件特征标记 ‘[’监护条件‘]’ ‘/’ 动作表达式

事件特征标记描述带有参数的事件:

事件名 ‘(’由逗号分隔的参数表‘)’

监护条件是布尔表达式,根据触发事件的参数和拥有这个状态机的对象的属性和链来书写这样的布尔表达式。也可用监护条件显式地指定某个可达对象的状态(例如,“in State1”或“not in State2”)。

如果触发了转换,就执行动作表达式。可以根据对象的属性、操作和链以及触发事件的参数,或在其范围内的其他特征书写动作表达式。在考虑任何其他的动作前,必须完整地执行相应的动作。执行的模型与“运行到完成”的语义有关。动作表达式可以是由一些有区别的动作组成的动作序列,其中包括显式地产生事件的动作,如发送信号或调用操作。表达式的细节与为模型选择的动作语言有关。

(3) 例子

如下定义了两个转换:

right-mouse-down(location)[location in window]/object: = pick-object(location);

object. highlight;

4. 状态图

(1) 语义

状态图用于描述具有动态行为能力的实体的行为,其中具体描述了对事件实例的接收的响应。通常用它描述类的行为,也可以用它描述其他模型实体(如用况、参与者、子系统、操作或方法)的行为。

(2) 表示法

状态图是表示状态机的图。用适当的状态和伪状态符号表示状态机图中的状态和伪状态,而一般用连接状态的有向弧表示转换。状态图中的初始状态和终止状态都是伪状态,分别表示为:

(3) 实例

图 5.25-2 描述了一个负责监视某些传感器的控制器的状态机。

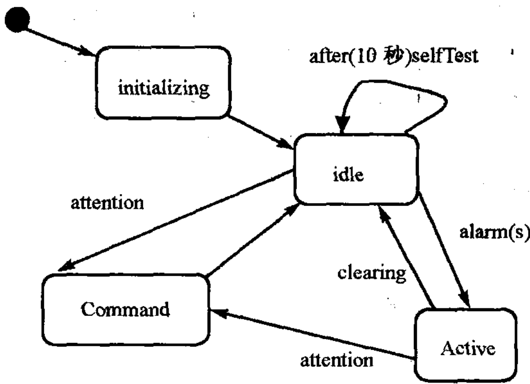


图 5.25-2 一个控制器的状态图

在这个状态图中有四个状态：Initializing(控制器开始运行)、Idle(控制器准备好，并等待警报或来自用户的命令)、Command(控制器正在处理来自用户的命令)和 Active(控制器正在处理一个警报条件)。当第一次创建这个控制器对象时，首先进入 Initializing 状态，然后无条件地进入 Idle 状态。Idle 状态上有一个由时间事件触发的自转换。当接收到一个 alarm 事件(包括参数 s，表示被触发的传感器)时，控制从 Idle 状态传送到 Active 状态。仅当发生 clearing 事件时，或是用户向控制器发 attention 信号(用以发布一个命令时)，才退出状态 Active。注意本图没有终止状态。这也是在嵌入式系统中常见的，希望系统不间断地运行。

5. 组合状态

上面讲述的状态图中的状态都是简单状态，而图 5.25-1 中的状态 Active 是一个组合状态，其中 DialTone 和 Timeout 等状态均为 Active 的子状态。

(1) 语义

组合状态是由两个或多个子状态构成的状态。一种构成方式是将一些子状态分派到不同的并发区域，把按这种方式构成的组合状态称为并发组合状态。另一种构成方式是把子状态组织成不相交的，把按这种方式构成的组合状态称为顺序组合状态。组合状态仅分为这样的两种。任何组合状态的子状态也可以是这两种类型的组合状态。

新创建的对象，从最外层的初始伪状态开始，执行其最外层的缺省转换。若对象转换到了最外层的终结状态，则对象的生命期终止。

一个状态内的各区域可以有初始伪状态和终止状态。到封闭状态的转换表示到其初始伪状态的转换。到最终状态的转换表示封闭区域中的活动的完成。在所有并发区域中的活动的完成，表示经由封闭状态的活动的完成，并触发封闭状态上的完成事件。

(2) 表示法

把组合状态展开是为了表示它的内部状态机结构。除了(可选的)名称和内部转换分栏外，状态可以包含容纳嵌套图的附加分栏。通过在图形区域里显示嵌套状态图，把状态展开，表示其不相交的子状态。

用虚线划分图形区域，以表示把状态展开，描述其并发的子状态。每个区域都是一个并发的子状态。每个区域有一个可选的名称，但必须包含一张具有不相交状态的嵌套状态图。用实线把整个状态的文本分栏与并发的子状态相分离。

(3) 实例

图 5.26-1 和图 5.26-2 分别描述了组合状态 Dialing 和组合状态 Taking Class 及其内的子状态。

组合状态 Dialing 内的子状态顺序执行。当一个转换进入 Dialing 时，首先进入其内的子状态 Start。当到达 Dialing 的终止状态时，就退出 Dialing。

组合状态 Taking Class 含有三个子状态 Incomplete、Passed 和 Faild，其中 Incomplete 中的

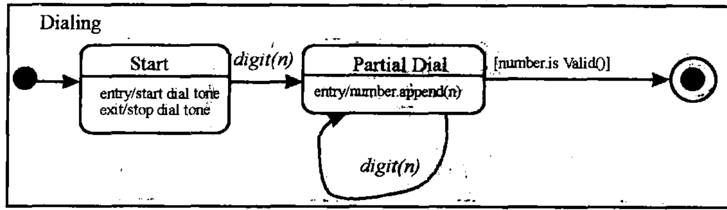


图 5.26-1 顺序子状态示意图

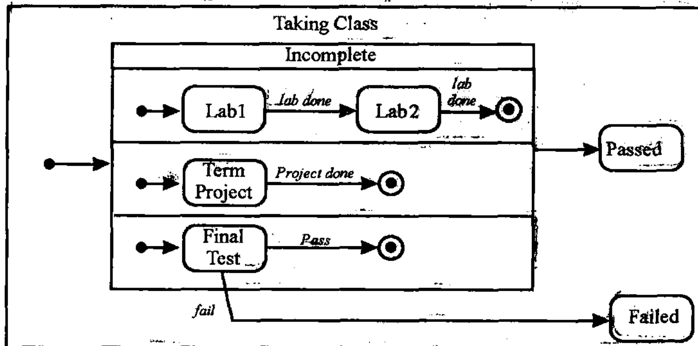


图 5.26-2 并发子状态示意图

子状态分为三组，每组之间是并发的。当进入 Taking Class 时，就到达了 Taking Class 内的初始状态，这意味要同时进入到 Incomplete 内的三个初始状态。当 Incomplete 内转换都到达了其内的三个终止状态后，就触发从 Incomplete 到 passed 的转换。

5.2 过 程 指 导

5.2.1 基于特定活动组织的过程

为了明确软件开发人员在系统分析与设计阶段的工作内容，本节为面向对象的分析过程与设计过程提供了指导性意见。

在分析阶段，需要充分地研究用户需求，认识清楚系统责任。根据问题域和系统责任的复杂性，抽象出多层次的子问题域，用包及相应的抽象机制来描述各个子问题域的信息。通过对问题域的分析研究，建立用况图、类图、状态图和顺序图。

在设计阶段中，考虑与实现有关的问题，建立针对具体实现环境的面向对象的设计模型。

在建模过程中，应借鉴以往同类系统的面向对象分析与设计的结果，尽可能地加以复用。

1. 面向对象分析的过程指导

图 5.27 给了面向对象分析阶段的建模过程框架。除了在具有阴影的椭圆中的活动外，其余的均可省略。

分析阶段的主要目标是建立系统模型。针对面向对象方法而言，就是要根据项目需要，有选择地建立一组用况图、一组类图、一组顺序图和一组状态图。

在基于特定活动组织的过程中，除具有阴影的活动外，以及这些活动所包含的子活动，没有给出特定的次序要求。

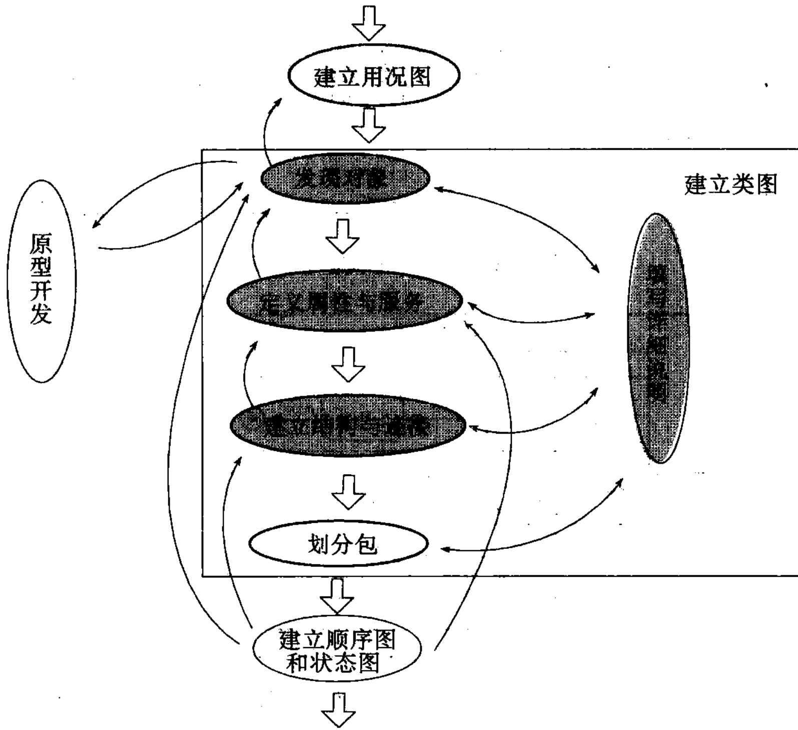


图 5.27　面向对象分析阶段的建模过程框架

2. 活动指导

(1) 初步划分子系统

若所要建模的软件系统规模很大或构成较为复杂，要按照用户需求及相关的业务模型或领域模型，对其进行分解，形成一个子系统的集合。对于小型系统可以不划分子系统。

系统与子系统形成一个层次结构，用文字描述子系统之间及子系统与系统外部的接口。要求子系统本身尽可能独立，在逻辑、功能和物理位置上是高内聚和低耦合的。

按这些子系统进行分工，各个分析小组对自己分工的子系统进行正规的面向对象分析与设计，最后将结果合并成一个大的系统模型。

(2) 建立用况图

① 确定系统边界

在面向对象分析和面向对象设计中定义并且在编程时加以实现的系统元素位于系统边界以内，而其余的元素位于系统边界以外。

② 定义参与者

结合问题域，从用户需求、业务模型(或领域模型)中发现参与者，凡是与系统直接进行交互(交换信息)的系统以外的事物均为系统的参与者。常见的参与者有：

(i) 人员：从接受系统服务的人员，以及为系统服务的各类人员中选择。

(ii) 设备：从与系统相连，向系统提供外界信息或者在系统的控制下运行的设备中找参与者。不与系统直接相连的设备以及计算机系统附带的设备，不能被看作参与者。

(iii) 外系统：具有如下两个条件的外系统可作为参与者：

● 与本系统相连，并进行信息交互。

● 对它的开发不是开发本系统的人员的当前责任。

③ 定义用况

(i) 发现用况

针对每一个参与者，考虑它使用系统的每一项功能的交互过程，以发现用况。着重考虑如下几点：

● 从各参与者如何与系统进行对话的角度，观察或设想参与者怎样与系统交换信息，以及系统向参与者提供什么功能。

● 一个参与者至少对应一个用况。

● 系统对外提供的每一项功能应该属于一个用况。

对已发现的用况给出一个能反映出它如何与系统进行交互的名字。

搜集所有的这样用况，形成一个候选的用况集合。

(ii) 描述用况

● 对每个候选的用况，用文字进行简要描述，要叙述清楚用况与参与者交互的动作序列，包括对正常、异常和出错情况的处理。它集中地体现系统责任，强调彼此为对方直接做了什么事，不描述怎样做。

● 标明引起各动作序列执行的参与者。

(iii) 确认

● 要以穷举的方式检查用户对系统的功能需求是否能在各个用况中体现出来。

● 各用况的行为序列应该是正确的、完整的和一致的，并对客户和开发者来说都是可理解的。

● 检查对各用况的描述与对相应的参与者的描述的一致性。

● 若用况对外界提供的功能过少，考虑把它与其他的用况合并。

● 若一个用况与过多的参与者交互，可考虑将其分解。

● 考虑将具有类似功能用况合并。

● 应使各用况尽可能地相互独立，通过对用况的拆分与组合，尽可能地减少用况间的相关之处。

● 消除各用况间相矛盾之处。

(iv) 描述非功能需求

对相应的用况描述非功能需求，如接口需求、设计约束和实现约束等。

④ 建立用况图

一个用况图描述系统的静态使用情况的一个方面，系统的所有用况图描述了系统的全部静态使用情况。

(i) 建立参与者与用况间的关系

上述的过程已经以穷举的方式考虑了每一个参与者与系统的交互情况，据此画出参与者与用况间的关联。

(ii) 建立用况之间的包含和扩展关系

● 在已经识别的用况中，若有一些共同的功能，可把它们提取出来，形成共用的用况。在需共享用况的内部，设立引入点。对于例外和异常等情况的处理，放在相应的用况中，在需要它们的用况中，设立扩展点及扩展条件。

● 画出用况间的包含和扩展关系。

● 若发现一个要被包含或扩展的用况已经不涉及系统内外的交互，而只是描述系统内部的功能细节，则要废除这样的关系。

⑤ 重新组织子系统

(i) 把支持一个特定的业务过程或参与者的用况分为一组，形成一个包。

(ii) 把具有包含或扩展关系的用况放入同一个包中。

(iii) 当一个用况跨越多个包时，要标识它在哪个包中被描述，并被哪些包引用。

(iv) 结合分组情况，重新组织子系统，并用文字明确地描述子系统之间的接口，以及子系统与系统外部的接口。

(3) 建立类图

① 发现类及对象

在面向对象分析的其他活动中，随时可对类及对象进行增删。

(i) 寻找后选类及对象

● 从问题域出发，考虑其中的人员、组织、物品、设备、事件、表格和结构等。

● 考虑与系统边界有关的人员、设备和外系统，确定它们是用参与者还是后选的类及对象。

● 从系统责任考虑，检查每一项系统功能，从中发现类及对象。

把所发现的各对象归于相应的类中。

系统责任要求的某些功能可能与实现环境(如图形用户界面系统，数据库管理系统等)有关，这些应该推迟到设计阶段考虑。

(ii) 审查和筛选

● 一个类在系统中要负有一定的责任、为其他类提供特定的服务或与其他类合作来提供某种服务，据此从属性或服务方面进行判断设置该类的必要性。舍弃重复的或无用的类。

● 有些名词所指称的事物可能更适合作为对象的属性或服务。

● 问题域中的某些事物实际上是另一种事物的附属品和在一定意义上的抽象，应避免这种信息冗余。

(iii) 异常情况的检查和调整

● 如果一个类的某些属性或某些服务只能适合该类的一部分对象，而对另一些对象不适合，则需考虑建立泛化关系，将这些属性或服务下放到它们能够适合的特殊类中。

● 如果两个(或两个以上)类的属性和服务有许多是相同的，则考虑建立泛化或聚合关系，将这些属性或服务提升到一般类中。

● 若一个类过大，考虑把它分解；若一个类过小，考虑把它与其他类合并。

(iv) 识别主动对象

从问题域和系统责任考虑：哪些对象具有某种不需要其他对象请求就主动表现的行为。

从系统执行情况考虑：如果系统的一切对象服务都是顺序执行的，那么就先找出最先执行的服务在哪个对象，这个对象应该是系统中惟一的主动对象。如果系统是并发执行的，那么就先找出每条并发执行的控制线程起点在哪个对象，这样的对象应该是主动对象。

从系统边界考虑：找出系统边界以外的活动者与系统中哪些对象直接进行交互，看处理这些交互的对象服务是否需要与其他系统活动并发地执行，这些对象很可能是主动对象。

从处理机上的对象分布考虑：在每个处理机上分布的一组对象中，至少应有一个对象是主

动对象；所有的被动对象都是在位于本处理机上的主动对象直接或间接驱动下运行的。如果这一条不满足，则应考虑增加主动对象，或把其中的某些被动对象改为主动对象。

认识主动对象与认识主动服务是一致的，请参看如何识别主动服务部分。

对于暂时不能确定的主动对象，仍按一般对象处理，到设计阶段再对其作考虑。

(v) 描述所识别出来的类

对于每个类，应设立一个详细描述其细节的模板(称为类描述模板)，在其中以文字的形式对它进行描述。在面向对象分析的其他活动中也可以随时填写类描述模板。

② 定义属性

(i) 发现后选的属性

- 按一般常识及当前的问题域，根据系统责任的要求，确定类应该有哪些属性。
- 考虑所建立的类要保存和管理的信息。
- 通过服务实现的功能，考虑需要增设的属性。
- 确定对象有哪些需要区别的状态，并决定是否需增加属性来区别这些状态。
- 确定用什么属性表示聚合和关联。对于聚合，整体对象中应有表明其部分对象的属性；对于关联，应该在一个对象中有表明与它连接的其他对象的属性。

(ii) 审查与筛选

- 丢弃没有为系统提供有用信息的属性。
- 若一个属性没有描述类本身的特征，则在本类中不应该出现这个属性。
- 如果发现属性的设置破坏了理解的完整性，则应该加以修改。
- 凡是在一般类中定义了的属性都不要在特殊类中重复出现。
- 若属性能从其他属性直接导出，则应该去掉它。
- 如果属性太复杂，可把一些属性分离出来，形成为一个或几个类。

(iii) 属性的定位

把属性放置到由它直接描述的那个类中。此外，在泛化关系中通用的属性应放在一般类中，专用的属性应放在特殊类中。

(iv) 属性的详细说明

在类描述模板中详细描述属性。

③ 定义服务

(i) 发现服务

- 考虑系统责任

逐项审查用户需求中提出的每一项功能要求，看它应该由哪些对象来提供，从而在该对象中设立相应的服务。

根据设置对象的目的(例如，保持某些信息，进行某种计算或加工，对活动者的响应或检测等)，分析该对象所应有的服务。

由于封装而需定义的对属性读写之类的操作，在此处不作考虑。

- 考虑问题域

考虑对象在问题域中具有哪些行为，确定其中的哪些行为与系统责任有关，据此设立服务来模拟这些行为。

- 分析对象的状态

画出状态转换图，考虑：在每一种状态下对象可能发生的行为，这应该由什么服务来描述；引起对象从一种状态转换到另一种状态的操作是什么，对此是否已经设立了相应的服务。

● 跟踪服务的执行路线

以穷举式的搜索，模拟全部的服务，以发现遗漏的服务。

(ii) 审查与调整

● 检查每个服务是否真正有用，无用的服务应该丢弃。

● 对于低内聚的服务考虑把它分解为多个服务。

● 若存在一个独立的功能被分解到多个对象的服务中，考虑把这些服务合并。

(iii) 识别主动服务

● 从问题域考虑，这个服务所描述的对象行为是由该对象主动呈现的，还是由外来的因素引发的。

● 考虑与系统边界以外的活动者直接进行交互的对象中是否有主动服务。

● 进行服务执行路线的逆向追踪，若发现某个服务不被其他成分所请求，则它应该是一个主动对象的主动服务。

面向对象分析阶段标注的主动对象和主动服务不一定是最终的结果，因为在面向对象设计阶段可能要增删一些新的主动对象和主动服务。

(iv) 服务的定位

把一个服务放置在哪个对象中，应和问题域中拥有这种行为的实际事物相一致。在泛化关系中，与属性的定位原则一样，通用的服务放在一般类，专用的服务放在特殊类。

(v) 服务的详细说明

在类描述模板中，对服务进行详细描述。

④ 建立泛化(泛化)关系

(i) 寻找泛化关系

● 学习当前领域的分类学知识

● 按常识考虑事物的分类

● 把每个类看作一个对象集合。如果一个类是另一个类的子集，则应把它们组织到同一个泛化关系中；若一个类具有另一个类的全部特征，则应把它们组织到同一个泛化关系中。

● 如果某些属性或服务只能适合该类的一部分对象，说明应该从这个类中划分出一些特殊类，建立泛化关系。

● 如果有两个(或更多的)类含有一些共同的属性和服务，则考虑把这些共同的属性和服务提取出来，构成一个在概念上包含原先那些类的一般类，组成一个泛化关系。

(ii) 审查与调整

对候选的泛化关系逐个审查，取消那些不合适的结构或对它们进行调整与修改。

● 按问题域和系统责任检查这样分类的合理性。

● 检查是否符合分类学的常识。

检查这种错误的方法是用“是一个”或“是一种”关系来衡量每一对一般类与特殊类。

● 检查特殊类是否真正地从一般类继承了有用的属性或服务。

● 不要在关系间产生循环。

(iii) 对泛化关系的简化

要重点地检查以下几种情况：

● 若特殊类没有自己特殊的属性与服务,则可取消这样的特殊类。

● 若某些特殊类之间的差别可以由一般类的某个属性值来体现,而且除此之外没有更多的不同,则可通过增加属性来简化泛化关系。

● 若一个一般类之下只有惟一的特殊类,并且这个一般类没有可创建的对象实例,则可把这两个类合并。

(iv) 多继承及多态性问题

可以从两个思路来发现多继承结构。一是看一个特殊类是否同时需要两个(或更多)一般类的属性与服务;二是看特殊类是不是被包含在两个(或更多)一般类的交集中。

在泛化关系中检查每一条继承路径,若有属性名或服务名重复,则要加以修改,或表示出属性和服务的多态性。对后一种情况,如果重复的属性名或服务名在语义上不同,则在特殊类中的属性或服务的名字前加"*"符号;如果是拒绝继承,则在在特殊类中的属性或服务名字前加"×"符号;此外,要在详细说明中分别给出不同的定义。

(v) 对一般特殊结构的详细说明

在类描述模板中对一般特殊结构进行详细描述。

⑤ 建立聚合关系

(i) 寻找聚合关系

注重考虑以下几个方面:

● 物理上的整体事物和它的组织部分;

● 组织机构和它的下级组织及部分;

● 团体(组织)与成员;

● 一种事物在空间上包容的其他事物;

● 抽象事物的整体与部分;

● 具体事物和它的某个抽象方面。

(ii) 审查与筛选

● 按问题域和系统责任检查这样分类的合理性。

● 如果部分对象只有一个属性,则应考虑把它取消,收缩(合并)到整体对象中去,也即变为整体对象的一个属性(除非还有其他理由)。

● 如果两个对象之间不能明显地分出谁是部分、谁是整体(但它们确实存在一种需要通过属性表示的关系),则不应该用聚合结构表示,而应该用关联。

(iii) 对聚合结构的详细说明

在类描述模板中对聚合进行详细描述。

⑥ 建立关联关系

(i) 寻找关联关系

● 从问题域和系统责任考虑,认识对象之间的静态联系。

● 认识关联的属性。

● 分析并标注关联的多重性

(ii) 异常情况处理

对于复杂关联,可增设关联类。

(iii) 关联的定位

若当连接线的某一端是一个泛化关系时，则要考虑：如果这个关联适应结构中的每一个类的对象，则把连接线画到一般类上；如果只适应其中某些特殊类，则把连接线画到相应的特殊类上。

(iv) 对关联的详细说明

在类描述模板中对关联进行详细描述。

⑦ 建立包

(i) 低层包的划分

- 把类图中的一组联系较为紧密的类划分到一个包中。
- 考虑把类图中的每个泛化和每个聚合划分到一个包中。
- 考虑把通过关联而发生互相联系的类划分到一个包中。
- 可把既不属于任何结构，也没有关联的类，暂时都划到一个包中。

建议：划分包的数量应在 7±2 之内，如数量较多，则应进行包合并。

(ii) 合并包

将一些较小的包合并成一个较大的包。下层的包被合并之后，可以取消它们，也可以保留它们(即产生一个包含它们的高层包)。最终应使每个包内部成分(类或底层的包)的数量不超过 7±2。

合并包的策略是：

- 根据问题域的情况，如果某几个包所包含的对象类在概念上比较接近，或者有较强的相关性，则可考虑把它们合并为一个包。
- 考虑系统责任，如果某几个包所涉及的系统责任有较大的相关性，或者说，它们的功能同属某项大的功能，则可考虑把它们合并为一个包。
- 把强耦合的包合并为一个包。这包括以下几种情况：

包之间有交叉，即有一个(或几个)类同时属于多个包，可考虑把它们合并；

包之间有较多的关系，说明它们之间有较强的联系，可考虑把它们合并。

- 在分布式应用系统中，可参考系统功能的分布情况进行包的划分与合并。分布在同一种结点上的几个较小的包可合并到同一个大包中。

合并后，建议：如果最上层包的数量仍然较多，则进一步按以上策略对高层包进行合并，直到其数量满足 7±2。

(iii) 包层次的控制

建议最好不超过三层。

(iv) 异常问题的处理

- 包的交叉问题

把属于多个包的类(或子包)在其中一个包中用常规的方法画出(作为正本)。在其他包中也用常规的方法画出，并用引用表示法“包名::类”表明该类(或子包)属于别的包。

- 可以突破 7±2

如果一个包的内部成分在数量上超过 7±2，但其中的类之间联系很紧密，或者很有规律，此时可以突破 7±2，把它们保持在一个包中。

⑧ 详细说明

类图的详细说明的活动可以作为一个独立的活动。更自然的做法是分散地进行，结合在其他活动之中，最后做一次集中的审查与补充。如下是两点说明：

(i) 描述类描述模板

为每个类建立一个类描述模板。

(ii) 关系的说明策略

- 对泛化只在特殊类中指出一般类;
- 对聚合只在整体对象类中指出部分对象类。
- 对关联,除非能预见到在设计与实现时,两边类的定义确有必要互相引用,否则只在连接一端的对象类中指出另一端的对象类。若连接线的某一端标注的多重性是固定的,且数量较少,则在另一端的对象中进行说明。

(4) 建立顺序图

顺序图基于用况详细而直观地描述了对象之间以及对象与参与者之间的关系。一般应在建立类图后建立顺序图。

可对一个用况或其中的一个行为序列建立一个顺序图,也可以对多个行为序列(可能分属于不同的用况)中的协作对象建立一个顺序图。

① 建立顺序图的步骤

(i) 考虑用况中对参与者的行为描述,从中找出参与者与系统的交互动作,考虑这些交互应由系统的哪个(或哪些)对象响应和处理。

(ii) 考虑相互协作的对象间的关系。

(iii) 把这些对象在图的顶部表示出来。

(iv) 在各对象的生命线上,把它的相关服务展开。

(v) 按服务间的关系,画出在相关服务间发送的消息。

② 对一些情况的处理

(i) 若当前的行为序列过大,可把它分解成几个片段,分别建立顺序图。

(ii) 若一个对象的几个服务可能要在同一时间段发送/接受消息,则按顺序图的分叉法表示。

(iii) 若需要指出消息的前置或后置条件,或若需要标明从消息的发送到消息的接受所需要的时间,直接用文字标注。

(iv) 若一个消息跨越的对象生命线过多,可水平移动相关的对象,以达到最佳效果。

③ 建立控制线程内部的消息连接

从类图中每个主动对象的主动服务(在顺序系统中它是惟一的)开始:

(i) 人为地模拟当前对象服务的执行。考虑为了完成当前的工作,需要请求其他对象(或本对象)提供什么服务。发现的每一种新的请求,就是一种新的消息。

(ii) 分析该消息的发送者与接收者在执行时是否属于同一个控制线程。这可从几个不同的角度去判断:

- 按问题域的情况和系统责任的要求,二者应该顺序地执行还是并发地执行。
- 从发送者的执行到接收者的执行是否引起控制线程的切换。
- 接收者是否只有通过当前这种消息的触发才能执行。

(iii) 沿着控制线程内部的每一种消息追踪接收该消息的对象服务,重复进行以上的工作。这种追踪可按宽度优先或深度优先的原则,进行穷举式的搜索。

当这项工作完成后,每个类的每个服务都应被模拟并执行过。否则这样的服务要么是多余的,要么是遗漏了消息。

④ 建立控制线程之间的消息连接

这仅在对并发系统的分析中需要。

以源于主动对象的控制线程作为并发执行单位，对每个控制线程主要考虑以下问题：

(i) 它在执行时，是否需要请求其他控制线程中的对象为它提供某种服务。考察这种请求是由哪个对象发出的，并由另一个控制线程的哪个对象中的服务进行处理。

(ii) 它在执行时是否要向其他控制线程中的对象提供或索取某些数据。

(iii) 它在执行时是否将产生某些对其他控制线程的执行有影响的事件。

(iv) 对于各个控制线程的并发执行，是否需要传递一些同步控制信号。

(v) 一个控制线程将在何种条件下中止执行。在它中止之后将在何种条件下由其他控制线程唤醒，用什么办法唤醒。

(vi) 分布在不同处理机上的对象之间的消息通信只能在不同控制线程之间进行。

在做完上述工作后，从当前服务所在的对象向所有接收消息的对象画出消息连接线。

(5) 建立状态图

一个状态图应只注重于系统动态交互的一个方面，仅包含对理解这个方面是重要的那些元素，且所提供的细节与当前的抽象层次要相一致。

① 针对类、用况或整个系统设置状态机的语境。

如果语境是一个类或一个用况，则收集邻近的类，包括这个类的所有父亲和通过依赖或关联能到达的所有类。这些相邻类是状态图中的动作的候选目标或在监护条件中要包含的候选项。

如果语境是整个系统，则要将注意力集中到这个系统的一个行为上。除非系统很小，否则对一个系统建立一个完整的状态图是很难的。

② 建立一个对象的初态和终态。为了指导对其余部分的建模，可能的话，分别声明初态和终态的前置条件和后置条件。

③ 找出这个对象可能响应的事件。如果已经找出，事件就应该出现在该对象的接口处；如果还没找出，就要考虑在你的语境中哪些对象与该对象交互，然后找出它们可能发送那些事件。

④ 初态开始到终态，列出这个对象可能处于的状态。用被适当的事件触发的转换将这些状态连接起来，接着向这些转换中添加动作。

⑤ 识别任何进入或退出的动作。

⑥ 检查在状态机中提到的所有事件是否和该对象接口所期望的事件相匹配。类似地，检查该对象的接口所期望的所有事件是否都被状态机所处理。最后，找出明显地想忽略的事件的地方。

⑦ 检查在状态机中提到的所有动作是否被闭合对象的关系、方法和操作所支持。

⑧ 不管是手动还是通过工具，跟踪检查状态机中的事件的顺序和对它们的响应。尤其要努力地寻找那些未达到的状态和导致机器停止的状态。

⑨ 在重新安排状态机后，按所期望的顺序再进行检查，以确保你没有改变该对象的语义。

3. 面向对象的设计阶段的过程指导

面向对象设计分为四个部分：问题域部分、界面交互部分、任务管理部分和数据管理部分。对于后三个部分中所涉及到的类图，所做的工作与在分析阶段建立类图时相同。

(1) 面向对象设计的问题域部分

把面向对象分析中产生的类图直接引入设计的问题域部分，根据具体的实现环境，面向对

象设计过程要对其进行调整、增补和改进。

① 复用

在类库中查找已有的可复用的类。

对查找到的类去掉无用的属性和服务。若该类能与问题域中的类构成泛化关系，则根据继承的类对本类进行调整，去掉多余的部分。修正问题域中的类与本类间的关系。

② 引入一个根类

为了把问题域中的各类组合在一起，可引入一个根类。此项为可选。

③ 引入一个附加类

引入一个类，它含有服务的公共集合，服务的细节在特殊类中定义。

④ 调整多继承

若泛化关系包括多继承，在使用一种只有单继承和无继承的编程语言时，需要调整。

对单继承的语言进行调整有两种方法：①用聚合关系或关联分解多继承。②把多继承调整为单继承。

对无继承的语言进行调整：把多继承展平成一组类及对象。

⑤ 决定关系的实现方式

聚合：决定在整体类中指出部分类时，是用部分的类直接作为整体对象的数据类型，还是用指针或对象标识定义构成整体对象的部分对象。

关联：在指出另一端对象的对象说明中设立指针。

⑥ 改进性能

(i) 为提高或降低系统的并发度，可能要人为地增加或减少主动对象。

(ii) 为类增加保存临时结果的属性。

(iii) 为类增加与实现有关的底层服务。

⑦ 提供数据管理部分

有两种策略：

(i) 每个对象自己保存自己；

(ii) 每个对象把自己传送给公共的数据管理部分，让数据管理部分进行管理。

⑧ 对例外的处理

考虑对输入错误、来自中间件或其他软硬件的错误的消息以及其他例外情况的处理。

⑨ 编程语言限制了可用的属性类型

根据具体的编程实现语言，考虑其支持与不支持的属性类型，对不支持的类型进行调整。

⑩ 其他

为了达到实用，要考虑一些用于为其他的类服务的类，诸如进行输入数据验证这样的类。

(2) 面向对象设计的界面交互部分

按用户要求和选定的 GUI，设计系统的人与计算机的界面。

① 建立初始命令层

此部分工作包括建立菜单条、菜单屏幕和快捷键等。

建议命令层的宽度尽量满足 7+2 原则，深度不多于 3 层。

② 报表及报告

对要生成的报表和报告格式等进行设计。

每一种报表或报告应对应于一个类。

③ 窗口

设计诸如安全、登录、设置和业务功能之类的窗口。

每一种窗口对应于一个类。窗口类型的类有确定的属性，如大小、颜色和滚动条等。服务也是实现问题域需求的服务，且实现这些服务需要人机交互。

④ 引入 UNDO 功能

至少要部分实现此功能。

⑤ 继续做原型

对所做的设计根据需要，可继续制作原型，任何存在的用户接口原型有一个基本输入。

⑥ 设计界面交互类

根据选择的编程语言，对上述的设计建立界面交互类。

(3) 面向对象设计的任务管理部分

为本系统中的任务管理进行建模。

① 识别由事件驱动的任务

识别一个由事件触发的任务。

② 识别由时钟驱动的任务

识别按特定的时间间隔或到达了特定的时间被触发而进行某些处理的任务。

③ 识别优先任务及关键任务

按响应时间的紧迫程度，识别出各种任务的优先级。按影响系统成败关键的程度，识别出关键任务。

④ 审查每个任务

一个任务必属于上述任务的范畴。

⑤ 定义各任务及有关协调者

(i) 描述任务

对任务命名，并进行简单说明。

(ii) 定义任务协调者及任务通信情况

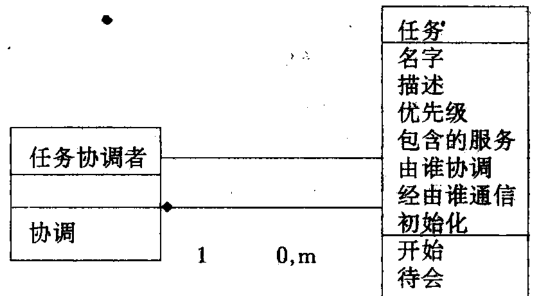


图 5.28 任务与协调者之间的关系

定义任务之间的协调者(类)，并描述任务之间的通信(参见图 5.28)。

(4) 面向对象设计的数据管理部分

为了存储问题域的持久对象，封装这些对象的查找和存储机制，以及为了隔离数据管理方案的影响，要提供在数据管理系统中存储和检索对象的基本结构。这可通过定义与数据存储系统交互的专门对象来负责对象存储及检索，也可通过对象自己进行存储管理，如下针对前一种方法进行了描述。

① 数据存放设计

(i) 对永久对象类的存放设计

可以把一个对象类映射到一个或多个文件(或表)，但通常把一个对象类映射到一个文件(或表)。

可采用下述方法之一，进行数据存放。

● 普通文件

列出各类及其属性，形成若干个表；各表要符合第一范式；为每个符合第一范式的表定义文件。

● 关系数据库

列出各类和它的属性，形成若干个表；各表要符合第三范式；为每个符合第三范式的表定义数据库表。

为了改善系统的性能也可以对类进行水平或垂直划分。

● 面向对象的数据库

扩充的关系方法——同(ii)。

扩充的 OOPL 方法——标出要长期保存的对象。

(ii) 对关联的存储设计

在关系数据库中按下述方法对关联进行数据存放。

● 每个多对多的关联映射到一张独立的表。

该表的主关键字是两个进行关联的表的主关键字的拼接。

● 每个一对多的关联映射到一张独立的表或在“多”的类中用外键隐含。

● 每个一对一的关联映射到一张独立的表或在各类中用外键隐含；也可把两个对象和关联放在同一表中，以提高性能。

(iii) 对聚合的存储设计

与对关联的处理类似。

(iv) 对泛化的存储设计

可采用下述方法之一，在关系数据库中进行数据存放。

● 把一般类的各个子类的属性都集中到一般类中，创建一个表。此方法较少使用。

● 为一般类创建一个表，并为它的各个特殊类各创建一个表。一般类的表与各子类的表要用同样的属性作为主关键字。

● 把一般类的属性放于各个子类中，为它的子类各建立一个表。

上述是对单继承的处理方法，对于多继承的处理与此类似。

② 相应的服务设计

为每个要存储对象的类增加属性和服务，也可以通过继承来获得这样的属性与服务。

按所选择的数据存储方式，进行相应的服务设计。

● 普通文件

定义一个类，有 2 个服务：告诉各对象如何存储它自己；检索被存储的对象。

● 关系数据库

定义一个类，有 2 个服务：告诉各对象如何存储自己；检索被存储的对象。

● 面向对象的数据库

扩充的关系方法——同上述的采用关系数据库的方法。

扩充的 OOPL 方法——不需要增加新的属性与服务。

图 5.29 为数据接口类的示意结构。

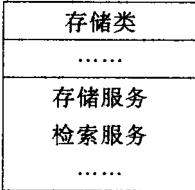


图 5.29 数据接口类的示意结构

5.2.2 统一软件开发过程

统一软件开发过程 (Unified Software Development Process，简称 USDP)是由统一建模语言(Unified Modeling Language，简称 UML)的开发者们提出来的，并为对象管理组织(Object

Management Group，简称 OMG）所推荐。统一软件开发过程是在权衡了三十余年的软件开发实践的基础上形成的。例如，它吸取了数百个用户多年的现场经验以及 Ratioanal 公司多年的工作成果。

USDP 对于如何运用 UML 的概念进行软件开发提供了详细指导，它指导开发队伍安排其开发活动的次序，为各开发者和整个开发组指定任务，明确地规定需要开发的制品，提供对项目中的制品和活动进行监控与度量的准则。

USDP 涉及的重要因素有：开发人员、项目、过程和工具。其中，开发人员包括设计人员、编程人员、测试人员、管理人员，以及用户、客户和其他人员（投资者、市场人员和法律代表等）等。项目在其生命期内涉及到物理设施、开发人员、其他人员以及所有的软件制品。过程是将用户需求转换成产品所需要的活动集的完整定义，还提供活动指南以及对产生需求报告和文档的要求。工具是支持过程定义中的活动实施的软件。

1．统一软件开发过程概述

USDP 是以用况为驱动的、以体系结构（architecture）为中心的、迭代的（iterative）、增量的（incremental）过程。具体地讲：

（1）以用况为驱动

在系统的生命周期中，以用况为驱动意味着：为建立所要求的系统，与系统有关的人员对这种要求需要进行交流，而将用况作为主要制品。以用况为驱动还意味着：用况是对系统进行分析、设计、实现和测试的基本输入，包括对系统体系结构的创建、验证和确认。即以用况为单位制定计划、分配任务、监控执行和进行测试等，将实现软件开发的核心工作有机地组合为一体，在产品开发的各个阶段中都可以回溯到用户的实际需求。

通过用况，可以得到体系结构和其他制品。首先选择在体系结构上具有重要意义的用况，接着实现那些关键的用况，构造出系统的初步体系结构，逐步得到稳定的体系结构。通过枚举用况的不同执行路径，可导出测试案例和测试规程。通过用况，还可估算系统性能、硬件需求和可用性，并能进行用户界面设计，也可以把用况作为编写用户手册的基础。

对用况的细化要在一定的程度上涉及系统的内部功能，对各用况进行完整的功能描述。细化时要区分用况中的三类事物：反映系统与参与者（actor）之间的接口，在用况中那些包含实现接口（功能）的活动和属性的实体，以及对前两者进行协调和控制的机制。

以用况为驱动也有助于模型之间的追踪和系统演化。

（2）以体系结构为中心

在系统的生命周期中，以体系结构为中心意味着将系统体系结构作为构思、构造、管理和改善系统的主要制品。

简单地讲，系统体系结构是所有与项目有关的人员都能理解的对系统的要求。建立系统体系结构便于对系统的概貌进行语义表述，以及控制系统的开发、复用和演化，还便于用户和其他关注者的理解系统，达到共识。

系统体系结构包括如下方面的决策：软件系统的组织；构成系统的结构元素和各元素的接口，以及由元素间的各种协作所规定的各元素的行为；结构元素和行为元素构成的不断增大的子系统；支持这种组织的体系结构风格；系统的性能、功能以及其他约束。要注意，系统体系结构描述系统中诸如子系统、构件、接口、协作、关系和节点等重要的模型元素，而忽略其他细节。

获得系统体系结构的主要步骤如下：

① 在普遍了解用况之后，得到系统体系结构的粗略纲要(独立于特定用况和平台)；

② 关注关键用况。关键用况是有助于降低最大风险的用况，对系统用户来说是最重要的用况，以及有助于实现所有重要的功能而不遗留任何重大问题的用况；

③ 随着对用况的规约，并考虑软件需求、中间件、遗产系统和非功能性需求等，不断产生更加成熟的用况和更多的系统体系结构成分；

④ 经过多次迭代，得到可执行的体系结构基线；

⑤ 经过逐步演化，形成稳定的系统体系结构描述。

在细化阶段末期，得到的系统体系结构即为系统体系结构基线(Architecture Baseline)，它是系统的"骨架"，是开发人员目前和将来进行开发时都要遵循的标准。它包括早期版本的用况模型、分析模型、设计模型、部署模型、实现模型和测试模型(其中用况模型和分析模型较为成熟)，这些模型中含有系统部件、中间件、要复用的遗产系统以及系统分布情况等。该基线与最终系统(对客户发布的产品)有同样的骨架。从最初体系结构基线到最终系统实现之间要经过几个内部发布，最后形成的体系结构基线为一个模型集合和一个体系结构描述，其中包括重要的用况及其实现。

建立可执行的体系结构基线是细化阶段的一个目标，进入构造阶段的体系结构基线应是坚实的。细化阶段结束时的体系结构基线在构造阶段变化不大，即体系结构基线到最终系统的体系结构之间的改动不大。

体系结构描述是模型的体系结构视图的集合，包括用况模型、分析模型、设计模型、部署模型和实现模型的体系结构视图，用于在系统的整个生命周期内指导开发。为了便于理解，可以对这些视图进行改写。体系结构描述强调最重要的设计问题，用于对系统的讨论和对问题的解决，它应该包括开发人员为了完成他们的工作所需要的全部内容。然而体系结构描述中不包括体系结构基线中的非体系结构级的元素，这是因为体系结构基线要对系统需求建模，以制订详细的开发计划，因此其用况模型中包含的内容比系统体系结构描述要多。具体地讲，体系结构描述中包括：

① 用以展示模型视图的体系结构意义上的用况、子系统(不涉及子系统的隐含成分和私有成分，及其变种)、接口、类(量很少，主要为主动类)、构件、节点和协作；

② 某些没有用用况描述的对系统体系结构有意义的需求，例如性能、安全、分布和并发；

③ 对平台、遗产、所用的商业软件、框架和模板机制等的简述；

④ 各种体系结构模式。

其实体系结构基线与体系结构描述同时开发，指导在一个生命周期中的软件开发。体系结构描述的第一个版本就是对第一个生命周期中细化阶段末的系统模型的抽取，还可能对其进行改写。体系结构描述在系统生命周期中不断更新，以反映体系结构基线的变化，包括新发现的抽象类和接口、为已有子系统增加的新功能、可复用构件的更新版本、进程结构的重新组织、体系结构基线的新的表示形式等。

(3) 迭代与增量

迭代是按照专门的计划和评估标准产生一个内部的或外部的发布版本，所进行的一组明确的活动。简单地讲，迭代的开发过程是将整个项目分为一系列的小项目，通过对一个小项目进行迭代，即执行需求规约、分析、设计、实现和测试这五个活动，对迭代结果进行评价，然后计划并进行下一个小项目，直至完成整个项目。迭代被组织到4个阶段中，参见图5.30。

如下说明了如何制订迭代计划：

① 早期迭代关注对问题和技术的理解，计划相对粗略，也更具有挑战性；

② 细化阶段为构造阶段制定计划，其中构造阶段的第一次迭代最清楚，后面逐渐模糊，在后期要进行修订；

③ 前期制订的移交计划需要在构造结束时修订；

④ 计划时要考虑前期迭代的结果、新加入迭代的用况、与新迭代有关的风险和模型的最新版本集。

早期迭代要积累需求、迁移主要风险、明确问题、寻找解决问题的知识、进行项目定位、建立体系结构基线和为后续开发制订详细计划，用较少的人力和物力进行此项工作；后期迭代要降低风险、实现构件和产生增量。

贯穿整个生命周期的迭代有主里程碑和次里程碑。在四个阶段的结束时都有一个主里程碑。主里程碑是管理者与开发者的同步点，以决定是否继续进行项目，以及确定项目的进度、预算和需求等。当一次迭代结束时，存在一个次里程碑，用以决定如何进行后续的迭代。在一次迭代结束时，把模型集合所处的具体状态定义为基线。

增量是系统中一个较小的、可管理的部分(一个或几个构造块)，通常指两次相邻迭代得到的内部发布之差，也即一次迭代的结果是一个增量。通过不断的迭代，不但获得增量，尽快获得反馈，而且也在不断地在降低风险。

2. USDP 中的阶段和核心工作流

在 USDP 中，系统的生命周期由一系列的周期组成，每个周期的产生的结果是用户产品的发布。第一次经历四个阶段被称作初始生命周期，后面的每一次对系统的演化而经历的四个阶段都被称作演化周期。每个周期有四个阶段，每个阶段又细分为若干次迭代，每次迭代都要经过需求获取、分析、设计、实现和测试，请看图 5.30。

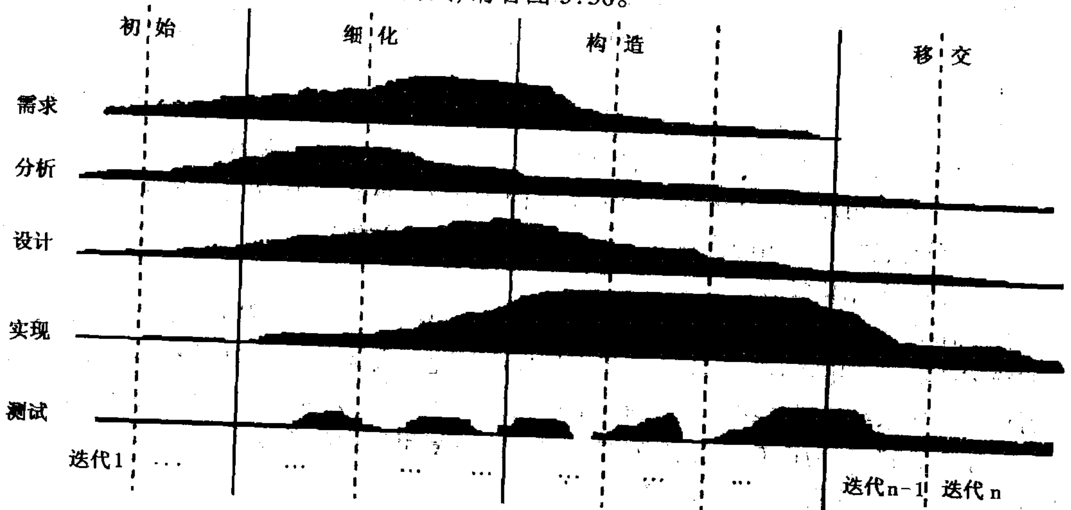


图 5.30 USDP 中的核心工作流和 4 个阶段

(1) 核心工作流

① 捕获需求

从客户、用户、计划者、开发者的想法(需求或用况)中搜取特征，形成特征表。对特征表中

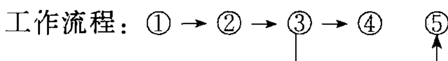
的每个特征给出简洁的定义，并描述其状态(例如，提议、批准、合并和验证等)、实施的代价及风险、重要的程度以及对其他特征的影响等。

先用领域模型和业务模型捕获需求，其中的活动如表 5.1 所示。用领域模型捕获系统环境中最重要的对象，即业务对象(如账目和合同等)、现实世界中的对象和概念以及事件。领域模型用 UML 的类图描述，此时类图中包含领域类(UML 类的构造型)及它们间的关系。业务模型由业务用况模型和业务对象模型支持。业务用况模型描述了具体的业务过程，其中含有业务用况和业务参与者。业务对象模型是一个业务的内部模型，它描述一个业务用况如何由工作者、业务实体(如支票)和工作单元(一些业务实体形成的有机体)实现。用交互图和活动图描述业务用况的实现，即业务对象模型。

从特征表和领域模型(或业务模型)中产生的制品有参与者、用况、用况模型和体系结构描述。其中详述的用况模型由整体描述、一组图和详述的用况组成。

表 5.1 捕获需求阶段的活动

序号	输　　入	活动	执行者	输　　出
1	业务模型或领域模型，补充需求，特征表	发现参与者和用况	系统分析员、客户、用户、其他分析员	用况模型$_{\text{概述}}$，术语表
2	用况模型$_{\text{概述}}$，补充需求，术语表	赋予用况优先级	体系结构设计者	体系结构描述$_{\text{用况模型角度}}$
3	用况模型$_{\text{概述}}$，补充需求，术语表	细化用况	用况描述者	用况$_{\text{详述}}$
4	用况$_{\text{详述}}$，用况模型$_{\text{概述}}$，补充需求，术语表	人机接口原型化	人机接口设计者	人机接口原型
5	用况$_{\text{详述}}$，用况模型$_{\text{概述}}$，补充需求，术语表	构造用况模型	系统分析员	用况模型$_{\text{详述}}$



② 分析

分析是对需求的精练，并进行构造，分析阶段的活动如表 5.2 所示。

表 5.2 分析阶段的活动

序号	输　　入	活动	执行者	输　　出
1	用况模型、补充需求、业务模型或领域模型、体系结构描述$_{\text{用况模型角度}}$	体系结构分析	体系结构设计者	分析包$_{\text{概述}}$、分析类$_{\text{概述}}$、体系结构描述$_{\text{分析模型角度}}$
2	用况模型、补充需求、业务模型或领域模型、体系结构描述$_{\text{分析模型角度}}$	分析用况	用况工程师	用况$_{\text{实现-分析}}$、分析类$_{\text{概述}}$
3	用况$_{\text{实现-分析}}$、分析类$_{\text{概述}}$	对类分析	构件工程师	分析类$_{\text{完成}}$
4	系统体系结构描述$_{\text{分析模型角度}}$、分析包$_{\text{概述}}$	对包进行分析	构件工程师	分析包$_{\text{完成}}$

此阶段产生的制品有：

(i) 分析类　多为概念性的，粒度比类大，很少有操作和特征标记，要用责任定义其行为，

在高层有概念性属性和关系。分析类可分为用于对系统和参与者之间的接口建模的边界类、对一些概念和现象的信息以及相关行为建模的实体类,以及协调、顺序化、处理和控制其他类的控制类。

(ii) 用况$_{\text{实现-分析}}$　注重于功能需求,包括类图(使用分析类)、交互图(用协作图描述分析类之间的交互)、事件流$_{\text{分析}}$(用文本解释图)和补充需求(用文本描述的关于持久、分布、并发、安全特征、缺陷的忍耐等方面的非功能需求)。

(iii) 分析包　由分析类、用况$_{\text{实现-分析}}$以及其他分析包组成。一般地把支持一个特定的业务过程或参与者的一些用况组织在一个包中,或把具有泛化或扩展关系的用况组织在一个包中。

(iv) 体系结构描述$_{\text{分析模型角度}}$　主要包括由分析模型分解成的分析包及其之间的依赖、关键分析类和实现重要及关键功能的用况$_{\text{实现-分析}}$。

工作流程:①→②→③→④

③ 设计

设计是给出系统的软件解方案,设计阶段的活动如表 5.3 所示。

表 5.3　设计阶段的活动

序号	输　　入	活动	执行者	输　　出
1	用况模型、补充需求、分析模型、体系结构描述$_{\text{分析模型角度}}$	体系结构设计	体系结构设计者	子系统$_{\text{概述}}$、接口$_{\text{概述}}$、设计类$_{\text{概述}}$、部署模型$_{\text{概述}}$、体系结构描述$_{\text{设计、部署模型角度}}$
2	用况模型、补充需求、分析模型、设计模型、部署模型	设计用况	用况工程师	用况$_{\text{实现-设计}}$、设计类$_{\text{概述}}$、子系统$_{\text{概述}}$、接口$_{\text{概述}}$
3	用况$_{\text{实现-设计}}$、设计类$_{\text{概述}}$、接口$_{\text{概述}}$、分析类$_{\text{完成}}$	对类设计	构件工程师	设计类$_{\text{完成}}$
4	体系结构描述(从设计模型角度)、子系统$_{\text{概述}}$、接口$_{\text{概述}}$	设计子系统	构件工程师	子系统$_{\text{完成}}$、接口$_{\text{完成}}$

此阶段产生的制品有:

(i) 设计模型　描述用况的物理实现的对象模型,注重于功能需求和实现环境对系统的影响。

(ii) 设计类　考虑与实现有关的因素,具体描述操作的参数、属性和类型等。

(iii) 用况$_{\text{实现-设计}}$　是设计模型中的一个协作,按设计类及其对象的术语,描述一个具体的用况如何实现和执行。由类图、交互图、事件流$_{\text{设计}}$(按对象或子系统的术语进行的文本描述)和与实现相关的需求组成。

(iv) 设计子系统　是组织设计模型制品的一种手段,用以描述大粒度的构件,由设计类、用况$_{\text{实现}}$、接口和其他子系统组成。

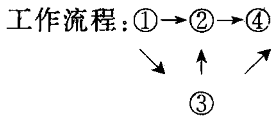
(v) 接口　表示由设计类和子系统提供的操作。

(vi) 体系结构描述$_{\text{设计模型角度}}$　主要包括由设计模型分解的子系统、接口、依赖、关键设计类和用况$_{\text{实现-设计}}$。

(vii) 部署图　是一个对象模型,按在计算节点上的功能分布描述系统的物理分布。

(viii) 体系结构描述$_{\text{部署图的角度}}$　包括部署模型的体系结构方面的视图。

其中:活动 1 把分析模型的分析包变为设计模型的子系统。活动 2 中的用况里的各个流应该各对应一个顺序图。



④ 实现

此阶段要实现设计类和子系统,其主要活动如表 5.4 所示。

表 5.4 实现阶段的活动

序号	输　　入	活动	执行者	输　　出
1	设计模型、部署模型、体系结构描述设计模型、部署模型角度	实现体系结构	体系结构设计者	构件概述、体系结构描述实现模型、部署模型角度
2	补充需求、用况模型、设计模型、实现模型当前建造	集成系统	系统集成者	集成建造计划、实现模型连续的建造
3	集成建造计划、体系结构描述实现模型角度、设计子系统已设计、接口已设计	实现子系统	构件工程师	实现子系统建造完成,接口建造完成
4	设计类已设计、接口由设计类提供	实现类	构件工程师	构件完成
5	构件完成、接口	完成单元测试	构件工程师	构件已完成单元测试

此阶段产生的制品有:

(i) 实现模型　描述设计模型中的元素如何用构件实现,并描述如何按实现环境组织构件,也描述了构件间的依赖关系。

(ii) 构件　是对模型元素(如设计模型中的设计类)的物理封装,要把它映射到节点上。

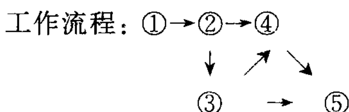
(iii) 实现子系统　由构件、接口和其他子系统组成。

(iv) 接口　用于表示由构件和实现子系统所实现的操作,设计时的接口能用在此阶段。

(v) 体系结构描述实现模型的角度　包括由实现模型分解成的子系统、子系统间的接口、子系统间的依赖以及关键构件。

(vi) 集成建造计划　在增量的管理步中,每一步的结果即为一个建造(bulid),即系统的一个可执行的版本。在一个迭代中,可能创建一个建造序列,该序列即集成建造计划。

其中:在活动 3 中,相应设计子系统中的每个类和每个接口要由实现子系统中的构件实现。在活动 4 中包含如下任务:列出包含源代码的文件构件,从设计类中生成代码,为设计类提供操作的方法,使构件提供的接口要与设计类提供的相一致。



⑤ 测试

测试包括内部测试、中间测试和最终测试,其主要活动如表 5.5 所示。特别是,当细化阶段中体系结构基线变为可执行时,构造阶段中系统变为可执行时,以及移交阶段中检测到缺陷时,都要进行测试。

表 5.5 测试阶段的活动

序号	输入	活动	执行者	输出
1	补充需求、用况模型、分析模型、设计模型、实现模型、体系结构描述模型的体系结构角度	计划测试	测试工程师	测试计划
2	补充需求、用况模型、分析模型、设计模型、实现模型、体系结构描述模型的体系结构角度、测试计划策略、时间表	设计测试	测试工程师	测试用况 测试过程
3	测试用况、测试过程、实现模型被测试的建造	实现测试	构件工程师	测试构件
4	测试用况、测试过程、测试构件、实现模型被测试的建造	执行集成测试	集成测试者	缺陷
5	测试用况、测试过程、测试构件、实现模型被测试的建造	执行系统测试	系统测试者	缺陷
6	测试计划、测试模型、缺陷	评价测试	测试工程师	测试评价

在测试阶段，要产生如下制品：

(i) 测试模型　主要描述在实现时可执行的构件怎样被集成测试者和系统测试者测试，以及描述系统的特殊方面（如用户接口、可用性、一致性以及用户手册是否达到目的等）怎样被测试。测试模型是测试用况、测试过程和测试构件的集合体。

(ii) 测试用况　描述测试系统的方式。一般描述如何测试用况（或部分），包括输入、输出和条件。

(iii) 测试过程　描述怎样执行一个或几个测试用况，也可以描述其中的片段。

(iv) 测试构件　测试构件用于测试实现模型中的构件。测试时，要提供测试输入，并控制和监视被测构件的执行。用脚本语言描述或编程语言开发测试构件，也可以用一个测试自动工具进行记录，以对一个或多个测试过程或它们片段进行自动化。

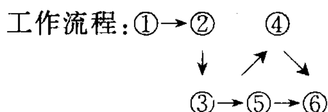
(v) 测试计划　测试计划描述测试策略、资源和时间表。测试策略包括对各迭代进行测试的种类、目的、级别、代码覆盖率以及成功的准则。

(vi) 缺陷　系统的异常现象。

(vii) 评价测试　对一次迭代，对测试用况覆盖率、代码覆盖率和缺陷情况（可绘制缺陷趋势图）进行评价。把评价结果与目标比较，准备度量。

其中：在活动 2 中，对各建造要设计集成测试用况、系统测试用况和回归测试用况。在活动 4 中，对一个迭代中的各个建造执行集成测试。当集成测试满足当前迭代计划中的目标时，要进行活动 5。

工作流程：



(2) USDP 的四个阶段

下面对 USDP 的四个阶段进行简述。

① 初始阶段

本阶段确定所设立的项目是否可行，并确定系统的目标。

(i) 对需求有一个大概的了解，确定系统中的大多数角色和用况，但此时的用况是简要的，这些用况构成了一个简要的用况模型。对给出的系统体系结构的概貌，细化到主要子系统即可。

(ii) 考虑时间、经费、技术、项目规模和效益等因素。

(iii) 识别并降低影响项目可行性的最不利的风险。

(iv) 关注于业务情况,建立初始业务用况。

② 细化阶段

简要地说,通过对问题域的分析,在该阶段要得到系统的框架。

(i) 识别出剩余的大多数用况。对当前迭代的每个用况进行细化,分析用况的处理流程、状态细节以及可能发生的状态改变。细化流程时,可以使用程序框图和协作图,还可以使用活动图和类图分析用况。

(ii) 降低重要的风险。

(iii) 进行高层的分析和设计并作出结构性决策。所产生的体系结构基线包括用况列表、领域概念模型和技术平台。以后的阶段对细化阶段建立的体系结构不能进行大的变动。体系结构基线的稳定是细化阶段结束的准则。

(iv) 为构造阶段制订计划。细化阶段完成,意味着能给出项目的成本和进度的估算。

③ 构造阶段

主要目标是开发整个系统,并确保产品可以向用户移交。

(i) 识别出剩余的用况;

(ii) 此阶段的每一次迭代开发都针对用况进行分析、设计、编码、测试和集成,所得到产品满足项目需求的一个子集;

(iii) 在代码完成后,要保证其符合某些标准和设计规则,并要进行质量检查。

④ 移交阶段

(i) 进行最后的验收测试,完成最后的软件产品;

(ii) 完成用户文档以及对用户培训等工作。

5.3 OSA 方法简介

如前所述,至今已提出了许多面向对象分析技术,但大体上可分为两大学派。一派称之为"方法驱动的方法"(例如 COAD-YOURDON 方法),一派称之为"模型驱动的方法"(例如本章介绍的 OSA 方法和 RUMBAUGH 等提出的 OMT——对象建模技术)。为了对面向对象方法有一个较为全面的了解,本章将简单介绍一下 OSA 方法。

OSA(Object-oriented Systems Analysis)是由 D. W. Embley 等于 1992 年提出的一种面向对象的系统分析方法。

OSA 不仅是面向对象的,而且还是模型驱动的。就是说,它提供了一组基本的模型化概念以及三种 OSA 模型(对象关系模型,对象行为模型,对象相互作用模型),分析员在进行系统分析中,以给定的模型化概念为指导,以模型构造(model construction)为驱动,产生对系统的询问,捕获系统的知识,建立系统的 OSA 模型。因此,对于 OSA 的了解,最重要的是了解构造系统的概念框架。

OSA 方法强调对系统的理解以及文档的组织,为此,OSA 提供了三种系统概念模型,它们从不同角度描述了所考虑的系统,形成了互补且统一的系统视图;对于文档的组织,OSA 还提出了一致的控制复杂性机制,最终产生的文档为以后的设计和实现提供了一致的基础。

由于 OSA 是模型驱动的,因此产生的 OSA 模型通常缺乏一定的表达能力。但是,按照

OSA的观点，分析阶段的任务是要研究理解系统，建立其相应的文档，而不应涉及与以后设计和实现有关的许多思想。

建造OSA系统分析模型的过程与方法驱动的分析不同，它不是一步一步的过程，而是随着相互作用的模型化活动，并发进行的。例如，一旦标识了一个对象类，分析员可以或在当时，或在以后为该对象类建立相应的行为模型(即状态网)，开发对象关系模型中与该对象类有关的部分。

尽管OSA并不规定OSA模型成分的建造次序，但是，OSA的目标确是十分明确的，即建造能够精确反映系统的分析模型，为以后的设计和实现提供必要的理解。因此，在系统开发过程中，可以自底向上，也可以自顶向下。即是说，OSA允许自由、灵活的系统分析风格。

5.3.1 OSA的对象关系模型(ORM)

OSA的对象关系模型(ORM)是用对象关系图表示的。ORM主要用于记录一个系统的说明性信息，即记录有关对象及对象之间关系的信息。为此，OSA给出了如下几个模型化概念：

·对象 ·关系 ·对象类 ·关系集合 ·约束

这些概念独立于相应的图示约定，即它们可以用其他图示符号表示。

1. 基本的模型化概念

(1) 对象

一个对象是一个单一的实体或概念。例如，一个人是一个对象，一个地方是一个对象，一件事情是一个对象。对象可以是物理的，也可以是概念的。一个对象可以与其他对象发生联系，也可以由其他对象组成，但是，每一对象必须是可标识的。

图5.31 对象

在自然语言中，名词或名词短语通常表示着对象。

在OSA中，用实心点表示一个对象，并且为了标识该对象，在实心点附近给出一个或多个名词或名词短语。如图5.31所示。

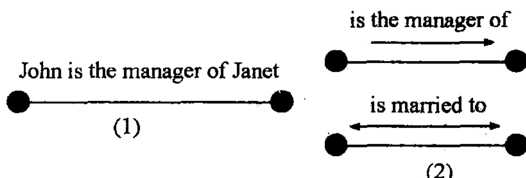
(2) 关系

关系是对象间的一种逻辑连接(a logical connection)。例如，"John is the manager of Janet"，就是对象John和Janet之间的一个关系。

在自然语言中，动词或动词短语常常表示着关系。

在OSA中，用一条线段表示对象间的关系，并且为了标识这一关系，给出该关系的描述，即给出关系名——通常是一个语句，以便使这一关系容易理解。如图5.32(1)所示。

为了简化关系的描述，对于常用的二元关系，可以省略关系名中所包含的对象名，只给出动词或动词短语，并在动词或动词短语下面给出一条单箭头线或双箭头线。如图5.32(2)所示。



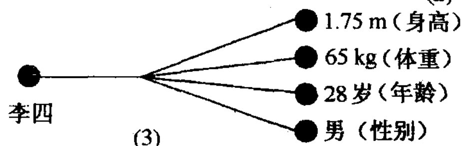


图5.32 关系

在一个关系中，连接的数目称为该关系的"元"。在OSA中，允许存在多元关系。在图5.32(3)中，我们给出了一个五元关系。

在 ORM 图中，就单独的一个对象而言，看起来往往不易理解它代表什么，只有了解了它与其他对象间的关系，才能知道它所具有的语义。

(3) 对象类

通常，我们按照某一逻辑理由，把一些对象分为若干个集合，以便易于处理，这一活动称为分类。对于当今所面临的、复杂的软件系统，也必须引入一种组织技术，以便控制复杂性。随之，产生了“对象类”这一概念。

按照某一逻辑理由，把一些对象分为若干个对象集合，我们称其中任一对象集合为一个对象类。

在 OSA 中，用矩形表示一个对象类，其中包含该对象类的名字。如图 5.33 所示。命名一个对象类，通常采用一般性的、并且可以描述该对象类中任一成员的名字，参见图 5.33。

图 5.33　对象类

OSA 鼓励分析员把一些对象组织为若干个对象类，即标识对象类。对于分类规则，OSA 不作任何限制。分析员可以按照自己认为有意义的逻辑理由，对对象进行分类。其中，应当考虑对象类中的对象，除了他们同属一个对象类外，一般还具有一些共同的特性；另外，还应注意使用的分类规则，应尽量避免产生通信上的更多困难。

(4) 关系集合

一个关系集合是一组关系，其中每一关系均具有相同的结构和相同的语义。即在这组关系中：

① 每一关系所涉及的对象数目相同；

② 每一关系所涉及的对象分别属于同一对象类；

③ 每一关系中对象名在关系中的位置相同；

④ 把每一关系名中的对象名去掉，所余部分不仅位置相同，而且是等同的。

我们可以把这样一组关系共同对应的结构看作是一块模板(template)。如图 5.34 所示。

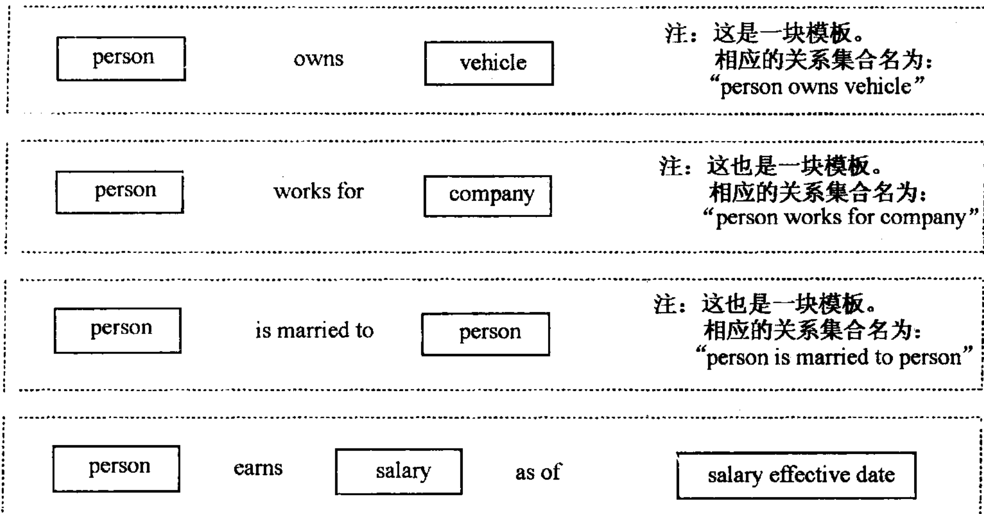


图 5.34　关系集合模板

在每一模板中，对象类指定了参与关系的那些对象的“源”，短语(例如：“owns”)表达了对象间的逻辑联系。

一个关系集合的命名规则是使用相应模板中出现的文字。例如，“person owns vehicle”。实际工作中，我们在说明一个关系集合时，往往首先说明了一块模板，因此，能够容易遵守关系集合的命名规则。一个关系集合可以有多个名字。

在 OSA 中，用菱形以及连接到相关对象类的线段来表示一个关系集合，把关系集合的名字写在菱形附近。见图 5.35(1)。

对于二元关系集合的名字，由于大多数具有统一的模式，即两个对象类的名字分别位于那个短语的前后，因此，可以把二元关系集合的名字简写，如图 5.35(2)，即省略了菱形，省略了对象类名，增加了一条有向箭头线。

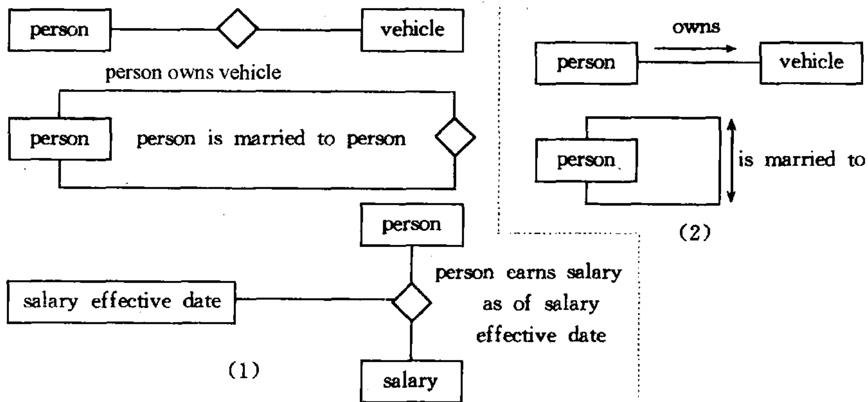


图 5.35　关系集合图例

(5) 约束

通过以上给出的模型化概念，我们可以建立一个系统的对象关系模型。但是，为了使描述的系统更令人满意，常常需要对其中的对象类和关系集合增加一些约束，指出它们所具有的其他性质。为此，OSA 引入了不同类型的约束，包括基本约束、参与约束、并发约束、特殊约束和一般约束等。

为了以后叙述方便，在本节中，首先简单介绍一下参与约束、并发约束和基本约束。

① 参与约束(participation constraints)

给定一个关系集合，参与约束定义了对象类中的一个对象可以参与该关系集合中的关系数目。例如，图 5.36 是一个关系集合。

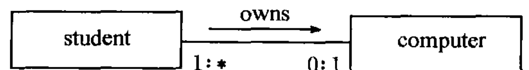


图 5.36　参与约束示例

如果把这一关系集合“展开”为如图 5.37 形式，就可以很容易理解该参与约束的含义。

其中，“1: * ”和“0:1”均是参与约束。“1: * ”表明一名学生可以有一台或多台计算机，而“0:1”表明一台计算机最多能够属于一名学生。

在 OSA 中，参与约束的基本形式是 min:max。其中，min 是非负整数，它表示该类的一个对象可参与的最小的关系数目；max 是非负整数或为“ * ”，“ * ”表示一个大于 min 的非负整数，max 指出该类的一个对象可参入的最大的关系数目。如果 min 和 max 相等，那么该参与

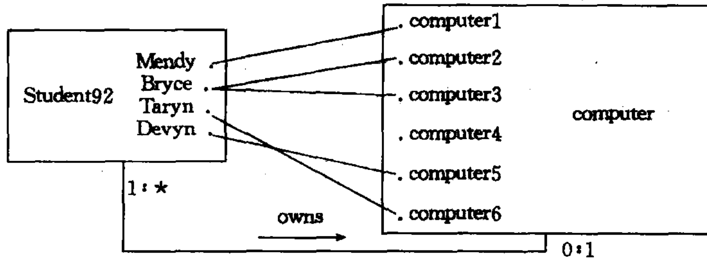


图 5.37 参与约束的解释

约束可写为 min 或 max。参与约束应写在对象类的连接附近。

② 并发约束(co-occurrence constraints)

对于多元关系而言,参与约束有时还不能很好地表达我们的一些要求或限制,为此,OSA 引入了并发约束。

在一个多元关系集合中,并发约束定义了一个对象类中有多少个不同对象可以与其他对象类中的特定的一个对象或特定的一组对象一起出现。如图 5.38 所示。

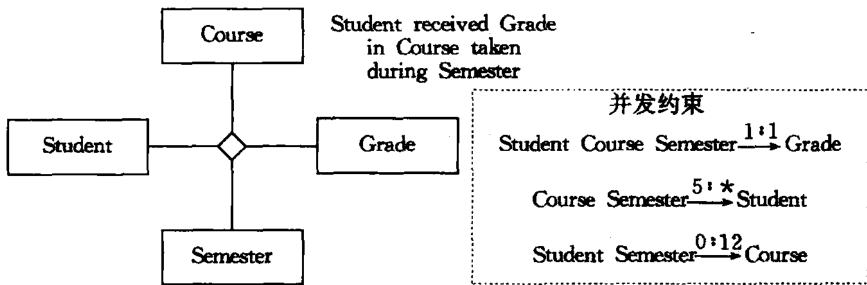


图 5.38 并发约束示例

其中,"Student Semester $\xrightarrow{1:12}$ Course"表明,对于 Course 对象类,可以与特定的那组对象"Student Semester"同时出现的不同对象数目最小为 1,最大为 12。就是说,一名学生在一学期中,最少可以学习一门课程,最多可以学习 12 门课程。

在使用并发约束时,应注意关系集合的有效性。

③ 基本约束(cardinality constraints)

基本约束定义了一个对象类中对象的数目。如图 5.39 所示。

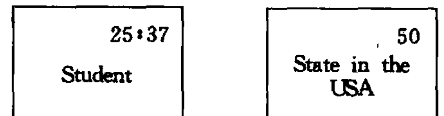


图 5.39 基本约束示例

由图 5.39 可知,基本约束的形式与参与约束的形式一样,只不过基本约束要写在矩形的右上角。当没有给出基本约束时,则意味着该约束为 0: * 。

④ 一般约束(general connstraints)

尽管以上给出的约束可以表达有关 ORM 的一些限制,但是,在分析中,它们还不可能满足分析员的需要,为此,OSA 引入了一般约束。

一般约束是一陈述语句,用于进一步限制 ORM 图中的对象类和关系集合。分析员可以按着自己希望的形式给出一般约束,例如,谓词演算符号或自然语言等。

2. 特殊的关系集合

在建造系统模型中,人们经常使用"抽象"等构造方法。为此,OSA 引入了三种特殊的关系集合:"is a"关系集合、"part of"关系集合以及"is member of"关系集合。在本章中,为了叙述方便,一律把"关系集合"简称为"关系"。

(1) 一般——"is a"关系

"is a"关系指出,一个对象类中的每一对象是另一对象类的一个对象。

如果对象类 B 中的每一对象是对象类 A 中的对象,那么,我们把 A 称为超类或称为一般类;相似地,把 B 称为子集或称为特殊类。通常,"is a"关系表达了继承。

在构造系统中,由于"is a"关系是我们经常使用的构造方法,因此 OSA 引入特殊的符号来表述这一关系,即用空心三角形以及和一般类、特殊类相连的线段来表示。如图 5.40(1)所示。

当一般类与多个特殊类具有"is a"关系时,可以把这一情况以图 5.40(2)的形式给出。

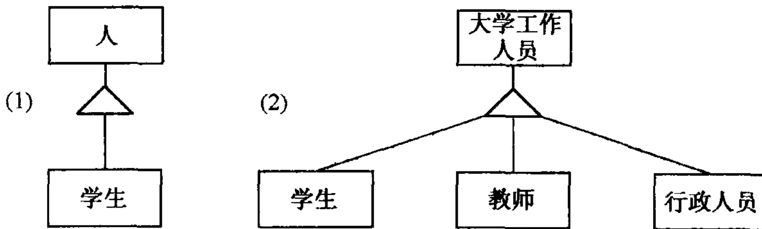


图 5.40 "一般/特殊"关系示例

(2) 聚合——"is part of"关系

"is part of"关系表明:一个对象,称之为聚合,是由一些称之为成分的对象构成的。在构造对象关系模型图中,我们常常涉及的"组装"关系、"容纳"关系、"包含"关系等,都是"is part of"关系,因此,这种关系也是我们经常使用的一种构造方法。

由于可以使用不同的方法把一个聚合分为若干个成分,因此对于一个聚合对象类而言,可以存在多个"is part of"关系。于是,"is part of"关系的一般表示如图 5.41 所示。

在"is part of"关系中,可以使用参与约束。考虑到各成分对象参与各自关系的数目可能有所差异,因此,如果采用图 5.40(2)的形式表达聚合,那么对聚合的参与约束就要写在实心三角形的下面,如图 5.41 所示。

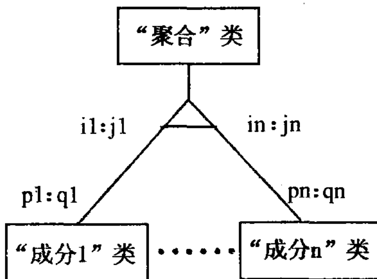


图 5.41 "一般/特殊"关系

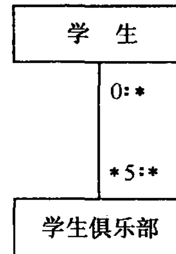


图 5.42 "联合"示例

(3) 联合——"is member of"关系

"is member of"关系用于生成一个由对象构成的集合,并把该集合看作是一个对象。

如果一个对象类，它的对象是集合，我们把这样的对象类称为集合类或联合(association)。如果联合的每一对象是由某一对象类中的对象构成的，那么我们把这一对象类称为成员类或全体(universe)。“联合”意指一些成员联合在一起构成了一个对象；“全体”意指生成子集的那个对象的集合。如图 5.42 所示。其中，“学生俱乐部”是集合类(联合)，“学生”是成员类(全体)。每一俱乐部是一个联合，是成员类的一个子集。

在 OSA 中，用一条带“*”的线段表示“is member of”关系，其中，要把“*”写在联合那一端。如图 5.42 所示。这一关系的读法，是从“成员类”读向“联合”。就图 5.42 给出的例子而言，应读为“学生是学生俱乐部的成员”。

参与约束可以用于“is member of”关系。在图 5.42 中，参与约束“0: *”表明，一名学生不一定是任一俱乐部的成员，然而，每一学生可能是多个俱乐部的成员。“5: *”表明，每一学生俱乐部必须至少有 5 名学生。

“is member of”关系总是二元关系。

3. 特殊对象类、资格条件、注释

(1) 特殊对象类

在 OSA 中，有两种特殊对象类。一种是“单一对象类”，即在该对象类中只有一个对象；一种是“关系对象类”，即在该对象类中，每一对象均是一个关系。

引入单一对象类，主要是为了解决概念上的问题。由于 OSA 只定义了关系和关系集合，而没有定义对象与对象类的连接，因此引入单一对象类，便很容易地解决了这一问题。

(2) 资格条件

在构造 ORM 中，为了确定一个对象是否属于某一对象类，为了确定一个关系是否属于某一关系集合，OSA 引入了“资格条件”这一概念。

对于某一对象类或某一关系集合，所谓资格条件就是对该对象类或该关系集合施加的那些约束的交。如果一个对象满足某一对象类的资格条件，那么它就是该对象类的一个成员。如果一个关系满足某一关系集合的资格条件，那么它就是该关系集合的一个成员。

(3) 注释

OSA 允许分析员为 ORM 添加注释，包括图示、解释等，以便使 ORM 具有更多的信息，使之更易理解。但是，添加的注释不能形成对对象类、关系集合的限制。

4. 对象关系模型小结

为了捕获一个系统的说明性信息，目前的分析技术以及 CASE 工具正摆脱传统的数据结构模型，逐渐采用源于 ER 模型的语义数据模型，引入了“一般”、“特殊”、“聚合”和“联合”等抽象机制。OSA 的 ORM 也是一种语义数据模型，并用 ORM 图表示之(如图 5.43 所示)。

为了构造 ORM 图，OSA 给出了五个基本概念，即对象、关系、对象类、关系集合和约束。其中，对象类是 OSA 的基础，刻画关系集合的图是 OSA 对象关系模型的基本成分。

在 OSA 中，不把属性作为 ORM 的基本构造，并认为在分析阶段说明属性是超前活动，有着潜在的危害。其理由是：

(1) 在分析过程中，很难定义属性，因为这一概念涉猎过广；并且很难区分对象和属性。

(2) 在分析过程中，若标识属性，会导致不必要的复杂构造，例如，重值属性、复合属性以及弱实体等；还可能表达了不该表达的属性间关系，引起以后数据规范化和模型集成中的一系列问题。

OSA 认为，在设计阶段，可以使用类似综合数据库系统的过程，自动地定义属性。

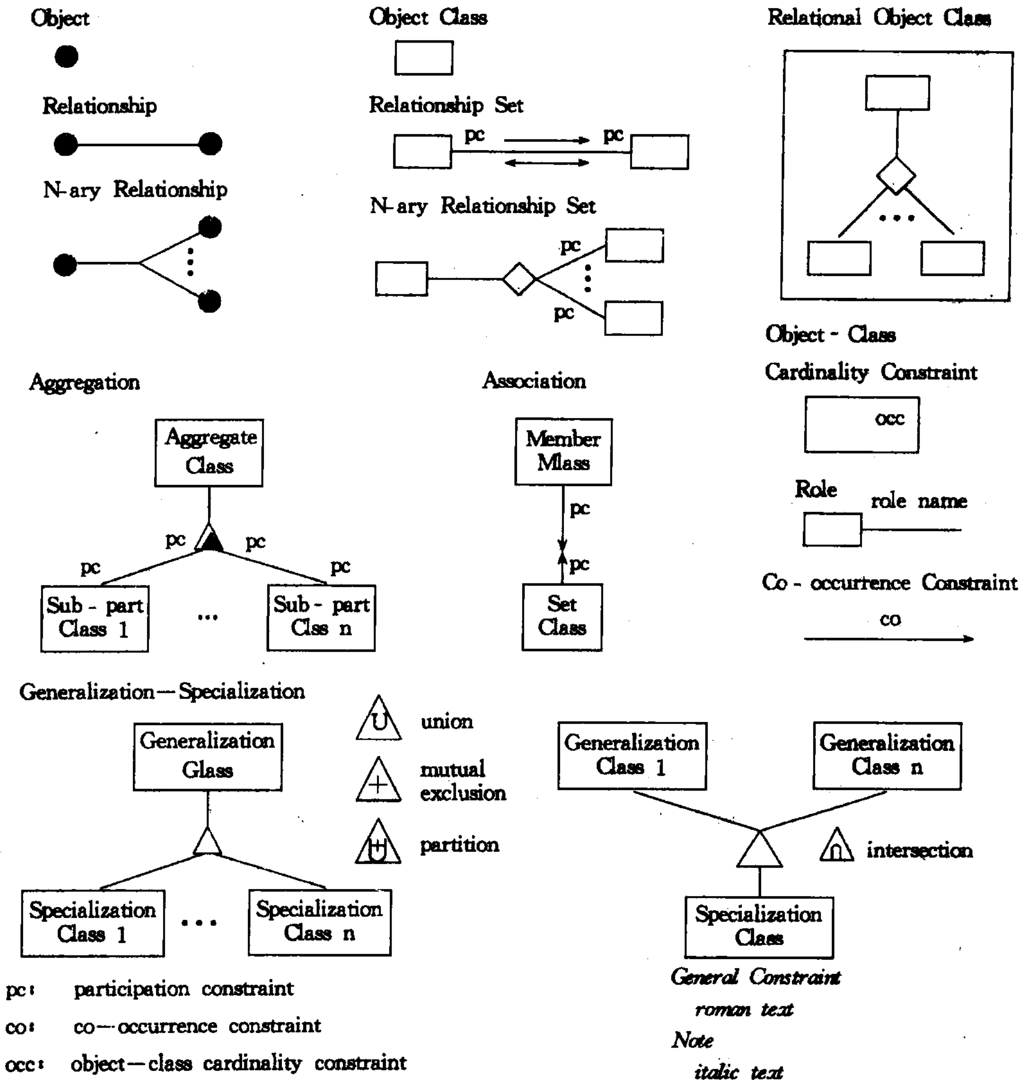


图 5.43 ORM 图

5.3.2 对象行为模型

对象行为模型用于描述系统中各对象的动态结构，即记录可察觉的对象状态、从一种状态转换为另一种状态的条件和事件以及对象执行的动作和对它所施行的动作。OSA 的对象行为模型是用状态网(state nets)表示的。

为了构造对象行为模型，OSA 集中于三个基本概念：状态(state)、触发(trigger)和动作(action)。这三个概念的模型化是状态网的基本构造。

1. 基本概念及概念模型化

(1) 基本概念

① 状态

在 OSA 中,一个状态表达了一个对象的外征(status)、阶段(phase)或活动(activity)。如图 5.44 所示。

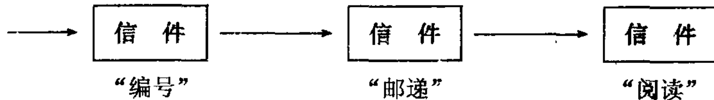


图 5.44 “信件”的状态

其中,“编号”、“邮递”、“阅读”均是信件的不同状态。这组状态是可观察到的,它们刻画了信件的变化。并且,还表达了一类“信件”变化所具有的共性,因而 OSA 为每类对象建立相应的对象行为模型。其中,对象在不同阶段所呈现的各种状态的集合,构成了行为模型化的基本部分。

我们可以用许多方法获得对象状态信息。一般,通过对系统中对象的观察或想像,可以得到该对象应具有的不同状态。

② 触发与转换

尽管标识对象状态是行为模型化的主要部分,但是,如果不了解一个状态与另一状态的联系,它也是没有多大用处的。因此,必须知道促使状态变化的事件和系统条件。

对象状态变化的过程称为转换。促使状态转换的事件和条件称为触发。如图 5.45 所示。其中,“⟶”表示转换,“⟶”之上的“封装与投递”等描述了触发。对象“信件”响应这一触发,从“编号”状态转换为“邮递”状态。

③ 动作

为了建立对象行为模型,不但要把状态转换模型化,还要把动作模型化。

在 OSA 中,把动作分为两类:不可中断动作和可中断动作。所谓不可中断动作是指分析员期望的那些一直执行到完成的动作,除非系统运行失败。所谓可中断动作是指那些在其执行中可以被挂起,以后还可以重新执行的动作。OSA 把不可中断的动作视为与状态转换有关的动作,而把可中断的动作视为与状态有关的动作,并把可中断动作模型化为一个离散的、运行时间长的活动(activity),该活动可以被中断,以响应作用于对象的事件或条件。例如,图 5.45中的“封装与投递”就是一种不可中断的动作,而“阅读”状态中,阅读信件就是一种可中断的动作。

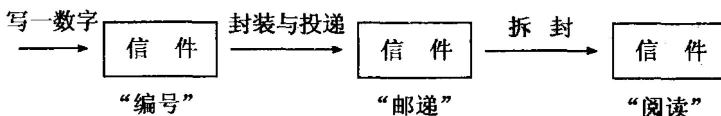


图 5.45 状态转换的触发

(2) 概念的模型化

① 状态的模型化

在 OSA 的状态网中,用圆角矩形表示对象状态。每一这样的矩形对应对象的一种状态,

其中给出表示该状态的状态名。例如，假定我们有一台机器人叉车，它的职能是把粮库中的粮袋装入运粮卡车。为了简化，我们假设该叉车有以下四种状态——“呆闲”、“驶向粮库”、“搬运粮袋”和“驶向卡车”，那么该叉车的这组状态如图 5.46 所示。

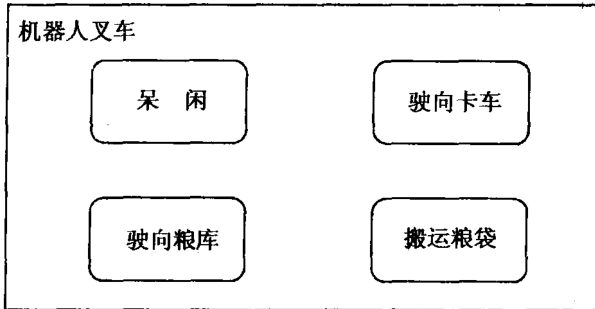


图 5.46 状态模型化示例

为了标识特定对象类的一组状态，用表示类的矩形把该组状态框起，并在矩形的左上角，给出该对象类的名字。

相对于特定的对象类，任一时刻，一种状态只有两种值：on 或 off。当对象正处于该状态时，其值为 on，否则为 off。

② 转换的模型化

在 OSA 的对象行为模型中，用矩形表示触发和动作。其中，把矩形分为两部分。上半部分给出触发的描述，即给出条件（当该条件满足时，引起一个状态转换）。下半部分描述在状态转换期间所发生的动作。如图 5.47 所示。

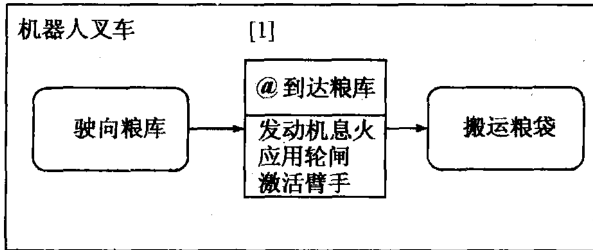


图 5.47 转换模型化示例

由该例可以看出，其中的动作可由一个或多个操作组成，这些操作在状态转换的某一时刻可能执行了，也可能没有执行，即是说，动作在概念上有多条执行路线。但是，只有所有路线均执行完成时，才实现了这一转换。

图 5.47 中触发“到达粮库”前的符号“@”指出该触发是基于事件的。在 OSA 中，把这一符号读为“基于”、“当”等。就图 5.47 给出的例子而言，可以读为“当到达粮库”。

在转换框左上角给出的“[1]”为转换标识符，以便以后引用这一转换。

在转换之前的状态称为“前状态”，而称转换后的状态为“后状态”。当一个对象处于某一转换的前状态时，我们称这一前状态为该转换的就绪状态。当一个转换处于就绪状态时，相应的触发才能使该转换发生。在完成了转换之后，对象处于后状态。

(3) 触发条件和事件

正如以上所述，当指定的系统事件发生或当确定的系统条件满足时，一个触发导引了一个对象的转换发生。相对于一个对象而言，这些事件和条件可能是外部的，也可能是内部的；甚至可能是一些事件和条件的组合构成了一个触发。一般而言，一个触发是一个产生真假值的布尔表达式。

① 基于条件的触发

条件是有关对象当前状态、系统环境当前状态、对象存在不存在、对象之间关系存在不存在的逻辑陈述。在 OSA 中，可以使用形式的或非形式的逻辑陈述，作为触发的条件。当条件为真时，相应的触发引起就绪的转换发生。

② 基于事件的触发

一个事件是系统的任一变化。例如，创建一个对象、删除一个对象、一个对象的状态转换、开始一个活动、接收一个命令、发送或接收一条消息等。任一可被察觉的变化均可被模型化为一个事件。因此，对象响应一个事件相当于该对象响应系统的一个可被察觉的变化。

在 OSA 的行为模型中，把一个触发中发现事件的那一部分称为事件监视器。事件监视器是概念上的设备，它能够发现一定类型的系统事件。事件监视器的名字是由相对应的触发和该触发所监视的事件类型合在一起而构成的。例如，图 5.47 中的"@到达粮库"就是事件监视器的名字。

事件与条件之间的区别是：事件仅在发生时触发了一个就绪的转换，而条件在该条件满足的整个期间触发就绪的转换。

在某些特定的情况下，为了记录事件及与该事件相关的信息，OSA 允许把事件作为对象，把一些类似的事件作为一个对象类，也允许对事件对象类添加关系的参与约束。

③ 复合触发

在触发描述中，把条件和事件监视器一起使用，便形成了复合触发。如图 5.48 所示。

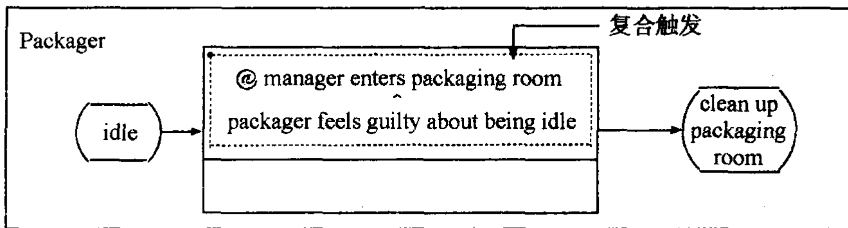


图 5.48 复合触发

其中，OSA 使用"∧"和"∨"，分别表示布尔"与"和布尔"或"。对此，只要适于相应的问题域和读者。OSA 允许使用其他等价的符号，由此可见，触发可以是事件和条件的"与"和"或"。

精确地说，任一时刻，系统中不可能有两个事件同时发生，因此，在构造复合触发时，若出现两个事件的"与"，就应当认真地考虑，看它是否符合实际系统的行为，千万不能为了缩短行为模型化过程而简单地使用复合触发。

2. 状态网

前面已经提及，OSA 用状态网来表示对象行为。构造状态网的主要成分是模型化的状态

和转换，因此，状态网是一种符号结构，表示对象类中所有对象的状态和状态转换。状态网可以被看作是行为模板，指出一个对象类中的每一实例具有该模板所表示的行为。

当分析员发现系统中存在类似的行为时，可以使用状态网来描述之。如果把一个状态网与一个对象类联系起来，则说明了该对象类的每一实例具有该状态网所描述的行为。OSA 给出了如下几种结构的状态网。

(1) 后状态(subsequent states)

后状态是转换之后的状态。在状态网中，用箭头线的箭头指出转换的后状态，而箭头线的尾和表示转换的矩形相连。

转换可以有不同形式的后状态。在图 5.49 中，给出了三种基本形式。

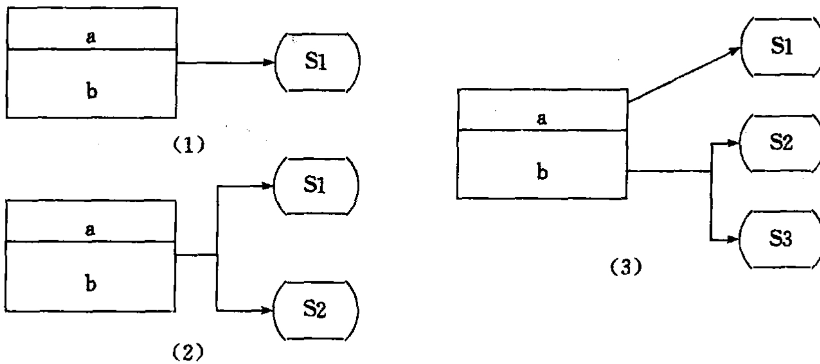


图 5.49 转换的后状态

图 5.49(1)表明，当转换的动作 b 完成后，状态 s1 变为 on，即对象处于这一状态。

图 5.49(2)表明，当转换的动作 b 完成后，状态 s1，s2 变为 on，即对象同时执行这两种状态所对应的活动。

在图 5.49(3)中，由于该转换有多条箭头线，分别指出多个后状态，因此存在一个选择问题，并且只能选择其中一条予以使用。即当转换完成时，对象或处于 s1 状态，或处于 s2，s3 状态。对于后状态的不确定性，可以使用约束，限制后状态的选择。如图 5.50 所示。

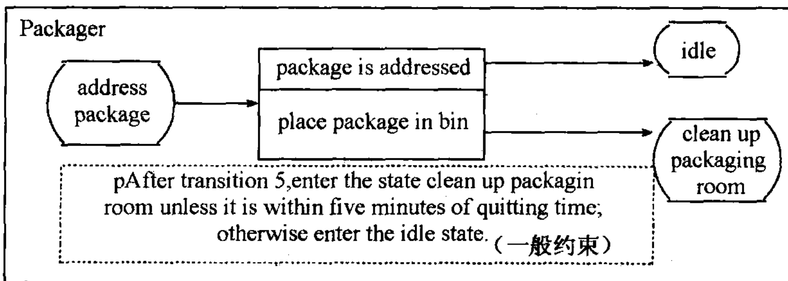


图 5.50 带约束的后状态

(2) 前状态(prior states)

前状态是一个转换之前的状态。在状态网中，用箭头线的箭头指出转换，箭头线的尾与前状态相连。和后状态类似，也存在不同形式的前状态。图 5.51 给出了三种基本形式：

图 5.51(1)表明：当状态 s1 有值为“on”时，该转换是就绪的，即在这种情况下，若触发 a 发生，则可以使该对象从 s1 状态转向相应的后状态。

图 5.51(2)表明：当状态 s1 和状态 s2 均有值为“on”时，才能使该转换成为就绪的。

图 5.51(3)表明：状态 s1 为“on”，或状态 s2, s3 为“on”，或状态 s1, s2, s3 为“on”，该状态是就绪的。当三个状态均为“on”时，依然存在一个选择问题。如果没有给出约束，则该选择是随机的。当触发发生时，使选取的状态变为 off。

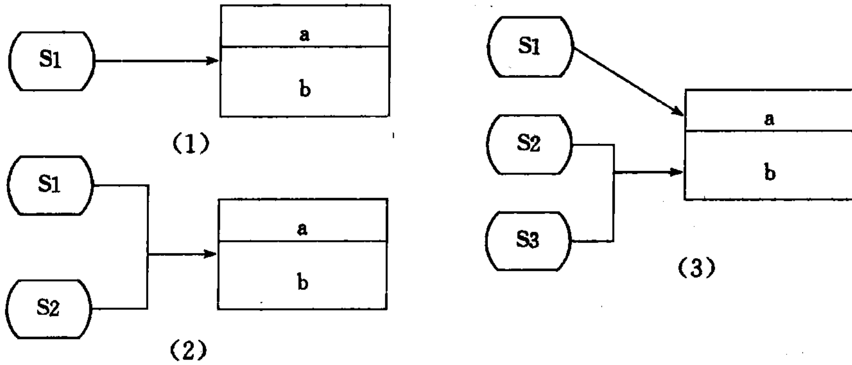


图 5.51 转换的前状态

(3) 初始转换(initial transition)

当对象最初进入系统时所呈现的状态，称为该对象的初始状态。初始转换激活对象的初始状态。显然，初始转换没有相应的前状态，它总是就绪的，即不管它的触发是否被满足。完整的状态网必须给出一个初始转换。

“@create”一般用作察觉新对象产生的事件监视器。许多初始转换使用这一事件触发器，当然，也可以使用其他与之等价的事件触发器，例如，“@birth”，“@hire”等。图 5.52 给出了两个等价的状态图，都初始化了一个对象“雇员”。

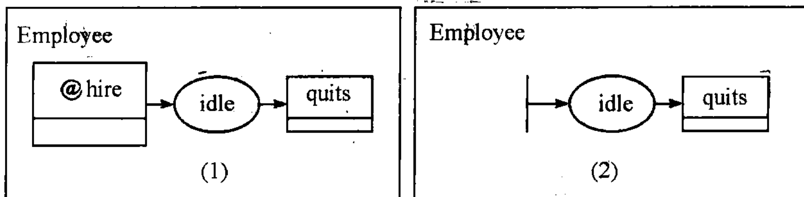


图 5.52 初始与最终的状态转换

其中，给出的初始转换描述了雇员的创建，“idle”是雇员的初始状态。

如果在初始转换中没有转换动作，则可以把这一初始转换简写为一个竖杠，如图 5.52(2)所示。即竖杠表示着没有初始化动作的初始转换，其触发为“@create”或其他等价的事件监视器。

如果分析员需要说明初始化动作，那么就不应该使用这样的简写形式，而应采用图 5.52(1)所示的形式，并在转换的动作部分给出初始化动作的描述。

(4) 最终转换(final transition)

最终转换是没有后状态的转换。当最终转换发生时，其前状态变为“off”。如果一个对象的所有状态均为“off”，则意味着该对象结束其生存，在系统中消失了。

最终转换是可选的。若一个对象有离开系统的要求，才有必要选用最终转换。与初始转换类似，最终转换的事件监视器为“@destroy”或其他等价的事件监视器，它触发了转换中的动作，导致对象的终止。在图 5.53 中，给出了最终转换的两种表示形式。

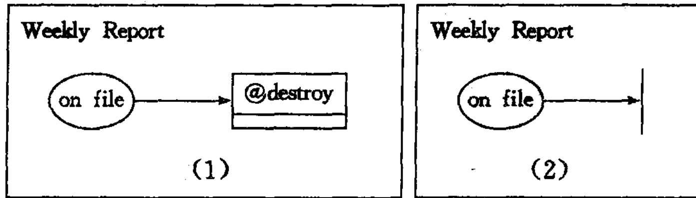
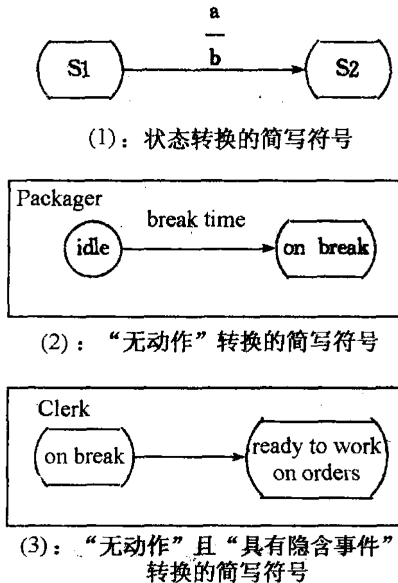


图 5.53　最终转换的简写符号

图 5.53 中，(2)是(1)的简写形式。竖杠“|”意指该转换是“@destroy”或其他等价的没有转换动作的事件监视器。与初始转换一样，如果需要描述最终转换的动作，那么就应当采用(1)的形式。

(5) 转换的缩写

大多数转换具有单一入口和单一出口，对于这样的转换，OSA 给出了相应的简写形式。如图 5.54 所示。



(1)：状态转换的简写符号

(2)：“无动作”转换的简写符号

(3)：“无动作”且“具有隐含事件”转换的简写符号

图 5.54　转换的简写符号

在图 5.54(1) 中，给出了状态转换的一般简写形式，即在横杠“—”上给出触发，在横杠下给出转换动作。

图 5.54(2)表明：如果一个转换没有转换动作，那么可以只给出触发，并把该触发写在箭头线之上。如果状态转换不需要什么事件或条件，也不需要转换动作，那么可以用一条箭头线直接把该转换的前状态和后状态连接起来。图 5.54(3)就是这样的一种情况。它表明只要

"on break"状态为"on","clerk"便进入"ready to work on orders"状态。

除了以上的简写形式外,还有一种转换的简写形式,即在状态网中使用了双箭头线。双箭头线意味着:处于某一状态的对象,当响应了一个触发,在执行完该转换动作后,还需要返回那个状态。图5.55表示了这种情况的简写形式。有时,为了使语义更加确切,往往需要添加一般约束。

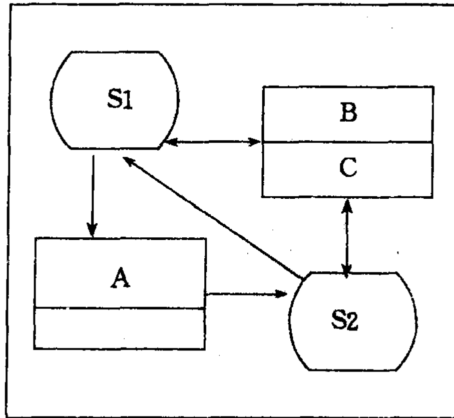


图5.55 状态网中的双箭头线

(6) 状态的保留

OSA允许一个对象进入新状态后,并不离开以前的状态,使该对象同时处于两个状态之中,保留了原来的状态,使之依然为"on"。在图5.56中,给出了状态保留的示例。

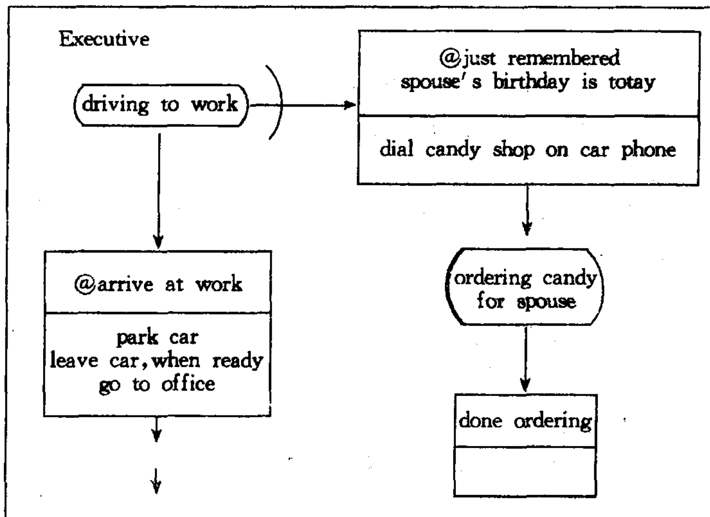
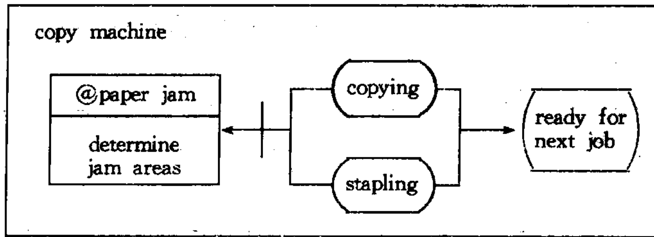


图5.56 不改变前状态值的转换

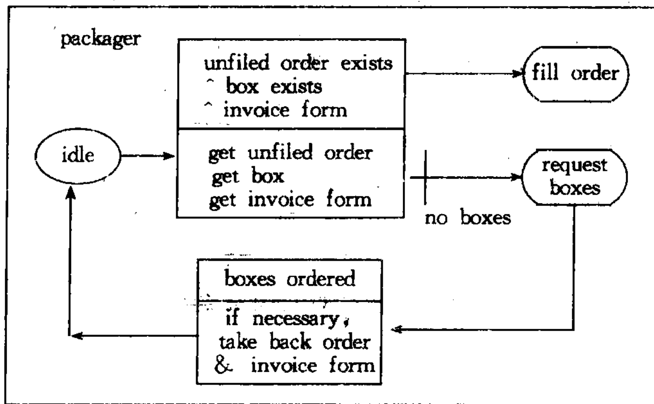
图中在状态"driving to work"右边的")"表示了对这一状态的保留。

(7) 例外(exception)

例外是一种系统事件或条件,它不是系统正常行为的一部分。如图5.57所示。图中箭头线上的竖线"|"表示例外。



（1）例外的模型化



（2）转换期间的例外

图 5.57 例外

图 5.57(1)表示，触发“@paper jam”是一例外。

在状态的转换期间也可能发生例外，即转换动作的例外。在这种情况下，要把竖线画在转换框附近，并给出例外的描述。如图 5.57(2)所示。

在状态网中，给出例外或不给出例外，它所描述的行为是一样的。给出例外，其好处是，可以使分析员指定一条特殊的路径，作为正常行为的例外，以便施行有效的控制。

(8) 实时约束(real-time constraints)

当时间是行为描述的一个重要成分时，分析员可以在状态网中给出有关时间的约束，以说明时间上的需求。OSA 允许把时间约束应用于“触发”、“活动”、“状态”以及“状态转换路径”。图 5.58中的“{…}”就是一些时间约束。

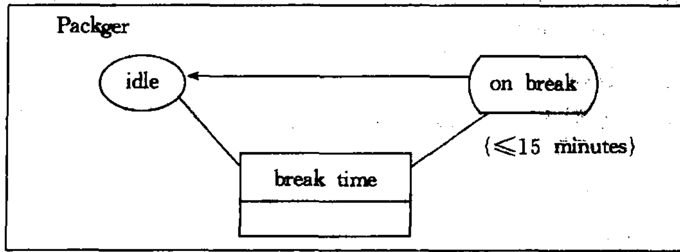
在图 5.58(1)中，状态“on break”附近的约束“{≤15 minutes}”表明，“packager”处于“on break”状态最多只能 15 分钟。即限制了对象“packager”在状态“on break”中的持续时间。

图 5.58(2)中的约束“{≤30 seconds}”限制了转换活动的最大持续时间。

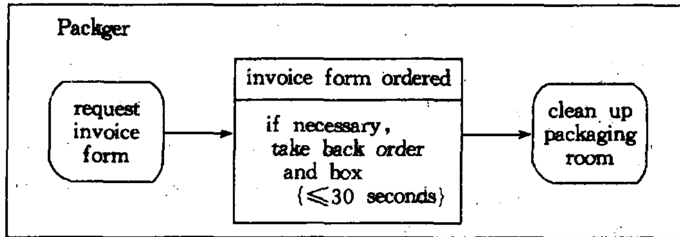
图 5.58(3)中的约束指出：当经理进入包装车间时，在“idle”状态中的“packager”必须在两秒钟内进入下一状态。

图 5.58(4)中的约束给出了整个转换的时间限制。

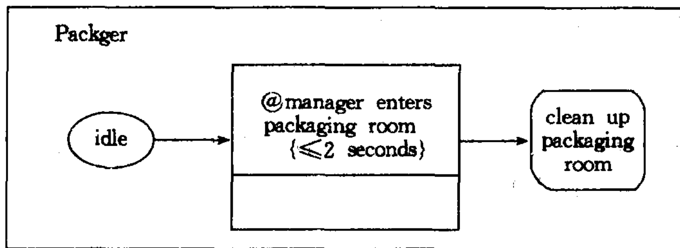
图 5.58(5)中，使用了路径标识“{a}”，“{b}”，时间约束“{a to b≤10 minutes}”表示，在“{a}”和“{b}”标识的路径内，其转换和活动最多使用 10 分钟。



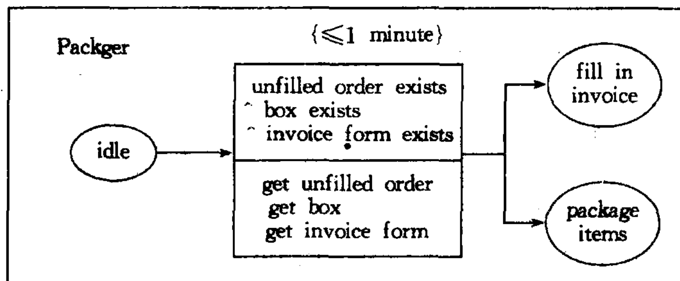
(1) 状态持续时间的约束



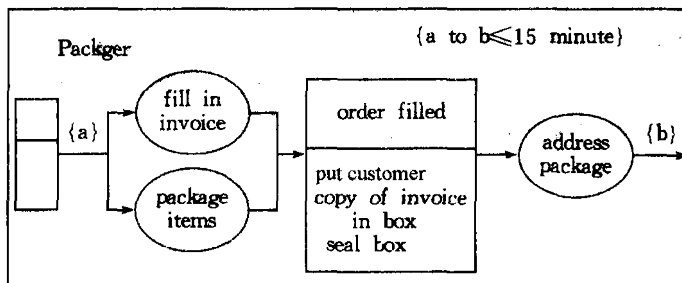
(2) 转换中动作的时间约束



(3) 触发的时间约束



(4) 整个触发的时间约束



(5) 状态网路径的时间约束

图 5.58 实时约束

OSA没有限定时间约束的表示格式,在具体的系统分析中,可以采取灵活自由的格式。一般地,当初始的对象行为模型建立以后,就可以给出时间约束。

(9) 状态网的一般/特殊

在实际系统中,不但对象类存在着"一般/特殊"关系,而且在行为描述方面也存在着"一般/特殊"问题。

特殊类的对象不仅表现了该类的共同行为,还表现了相应一般类的行为,即特殊对象的行为包含着一般对象的行为。

OSA允许从两个不同方向开发状态网。当我们把一个状态网特殊化时,可以使其中的触发和动作具有更多的特性,增加一些特殊的行为。从而,可以认为,特殊类的状态网是由一般类的状态网生成的。反之,我们也可以对给定的一组状态网,通过标识和提取共同行为,生成一般的状态网。从而,也可以认为,一般类的状态网是由特殊类的状态网生成的。

当特殊类和一般类之间的行为,相对于状态和转换而言没有多大区别,或这一区别可以标识时,则不必重新把整个特殊类的状态网画出,可以只画出那些与一般类不同的行为,形成特殊类简化的状态网。这一简化有以下两点益处:

① 由于简化的状态网,仅给出特殊类与一般类状态网的不同,因而可以减少潜在的错误;

② 使用简化的状态网,容易了解特殊行为与一般行为的不同。

如果对象类的状态网是以简化的方式给出,那么,在概念上就相当于对"一般/特殊"层次结构中的所有一般类增加了细节。

复杂的状态网(例如,多层次、多继承),使用简化形式,会在概念上产生控制上的困难。因此,在这种情况下,OSA建议使用标准的状态网。

3. 对象行为模型小结

为了捕获行为信息,目前的分析技术一般不大采用整个的系统模型,像操作模型、过程相互作用模型以及有限状态机等,而使用面向对象的行为模型。

OSA的对象行为模型是一种面向对象的行为模型,并且用状态网表示之,它描述了对象类中所有对象的共同行为。

在构造对象的行为模型中,OSA围绕着三个基本概念:状态、触发、动作,来组织对象的行为信息。这三个概念的模型化是状态网的主要成分,其中,对象的状态集合是状态网的基本框架。

支持并发是OSA对象行为模型的突出特征。由于它是面向对象的,因此可以表达不同对象类的对象并发;由于一个状态网可以被看作为一类对象的行为模板,因此可以表达同一对象类中不同对象的并发;又由于OSA提供了多重后状态、多重前状态以及状态保留等机制,于是可以表达一个特定对象不同动作的并发。

另外,OSA还支持具有时间延迟的转换,支持例外,支持整个对象的"一般/特殊"抽象等,这一切弥补了模型驱动分析技术在表示能力上的一些不足,形成了自己独特的风格。

图5.59是OSA对象行为模型中所用的表示符号。

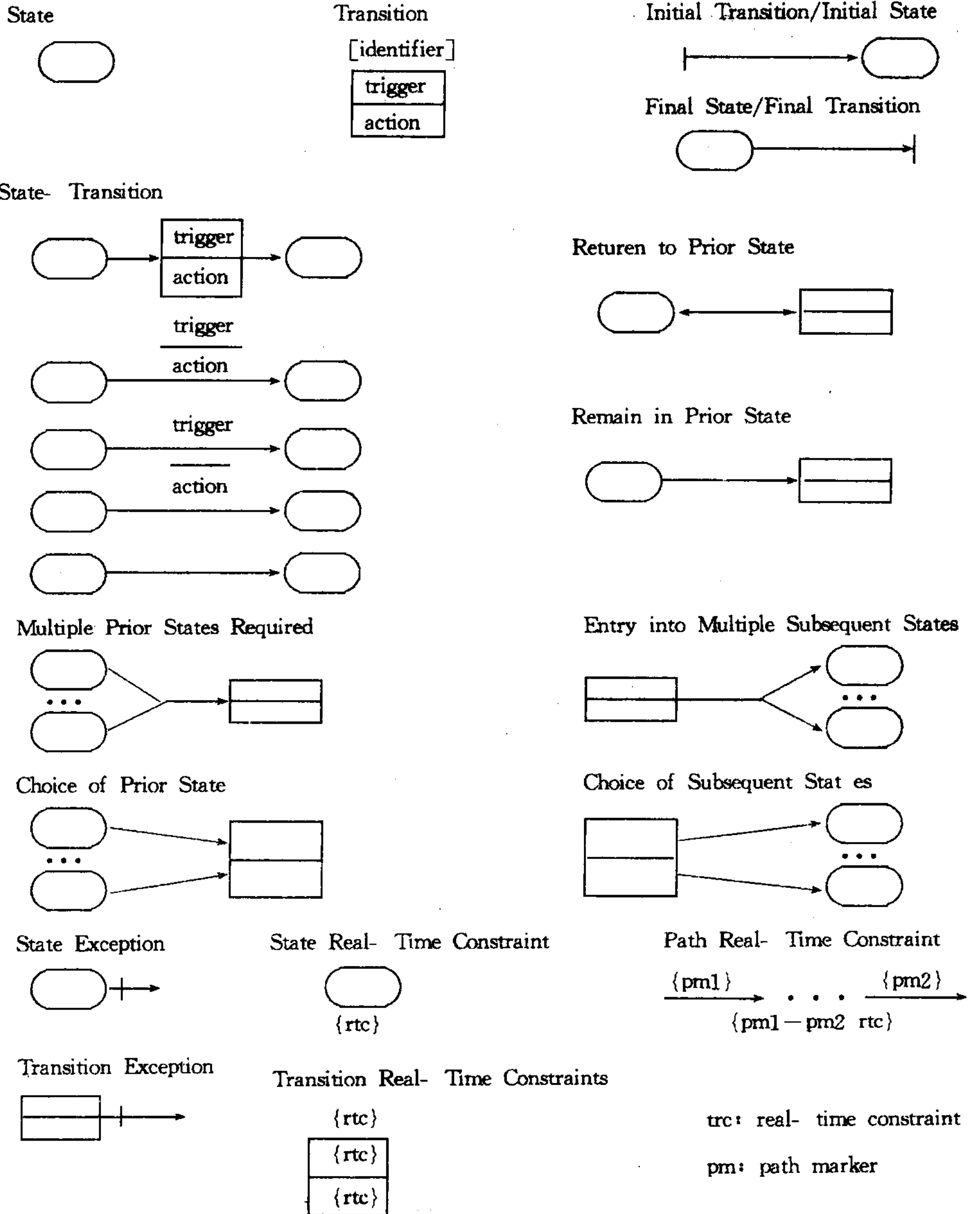


图 5.59 OSA 对象行为模型中所用的表示符号

5.3.3 对象交互模型

分析员可以使用对象关系模型来描述对象之间的关系，可以使用对象行为模型来描述各类对象的行为，但是，为了宏观了解系统行为及功能，就必须刻画对象之间的相互作用。

刻画对象之间的相互作用,就是描述以下三方面的内容:

(1) 在对象交互中,涉及了哪些对象;

(2) 在对象交互中,每一对象是如何活动的;

(3) 对象交互的本质是什么。

由 5.3,5.4 节可知,对象关系模型描述了第一方面的内容,对象行为模型则描述了第二方面的内容。OSA 引入一种新的机制来描述交互的本质,即描述交互的行为以及交互中被交换的信息。根据这一机制,以及相关的 ORM 和状态网,可以构造 OSA 对象交互模型。

1. 基本的对象交互

在 OSA 中,把交互的基本元素分别称为:起始对象、终点对象以及交互链。如图 5.60 所示。

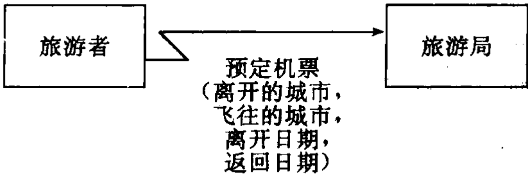


图 5.60 交互描述示例

其中,"旅游者"为起始对象,"旅游局"为终点对象,"Z 形箭头线"为交互链。并把"预订飞机票(飞往的城市……)"称为交互标识。图 5.60 构成了一个简单的交互描述。它描述了该交互的行为和交互中被交换的信息。

OSA 提供了以下四种基本描述交互的格式,如图 5.61 所示。

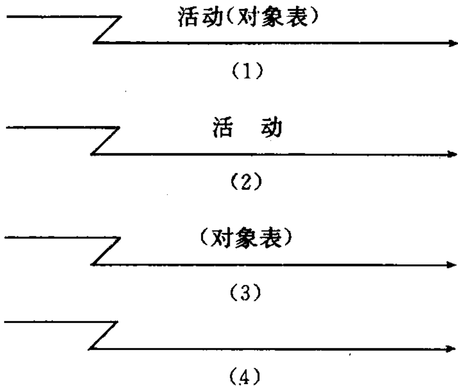


图 5.61 描述交互的基本格式

图 5.61 中,图 5.61(1)表示:在这一交互中,既有交互的行为,又有被交换的信息;图 5.61(2)表示:在这一交互中,只有交互的行为;图 5.61(3)表示:在这一交互中,只有被交换的信息。这样的交互通常被称为通信,即一个对象向另一对象发送一条消息,其中动作的默认值为通信或其他可能的同义词。由于消息可能是可感知的,或是不能被感知的,因而,把消息放在括号内。

在图 5.61(4)所示的交互中,没有给出交互标识。它表明该交互或不言自明,或分析员不能

用简单的交互标识清晰地描述之。

2. 特殊类型交互的描述

(1) 对象间的同步交互

在实际生活中，存在对象间的同步交互。例如，在我们正常通信中，当发送者发送一条消息时，接收者必须做好接收这一消息的准备。如果接收者没有做好接收这一消息的准备，则该消息自然丢失。因此，为了成功的通信，发送者和接收者必须了解通信发生的条件。据此，OSA 借助于状态网，描述对象间的同步交互。如图 5.62 所示。

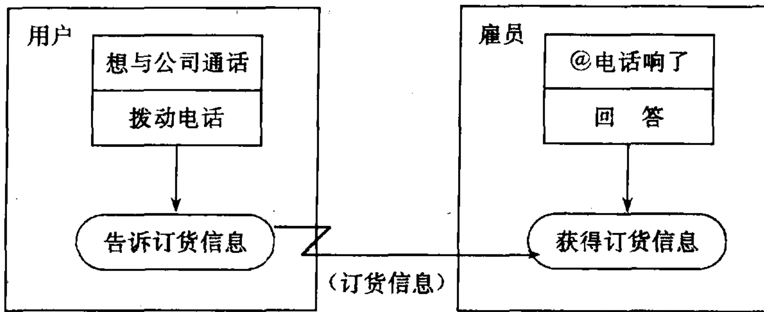


图 5.62 同步交互示例

(2) 对象间的异步交互

对象间的异步交互在实际中是经常发生的。我们给自己的亲人或朋友寄信，就是对象间的异步交互的例子。

在 OSA 中，通过提供“中介”对象(例如，邮局)，建立对象间的异步交互。如图 5.63 所示。

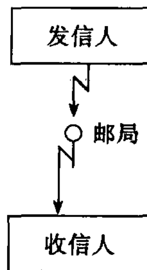


图 5.63 异步交互示例

(3) 说明特定的交互对象

在以上的例子中，均没有指出特定的交互对象。但在实际中，常常需要指出参与交互的特定对象。为此，OSA 引入相应的符号，描述这一类型的交互。如图 5.64 所示。

其中，“TO:…”为标识信息，用于指出一个对象类中的特定对象。即 OSA 使用了“TO——短语”来标识交互中特定的终点对象；并使用了“FROM——短语”来标识交互中特定的起始对象。

(4) 与多个对象的交互

在实际生活中，还存在涉及多个起始对象和多个终点对象的交互。例如，新闻广播就是这样

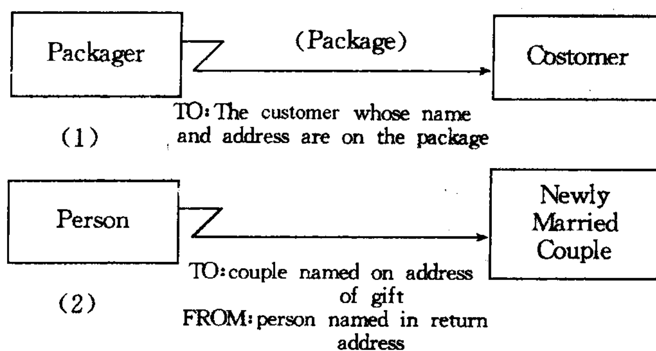


图 5.64 特定对象间的交互

一类的交互。

在 OSA 中,使用如下符号,描述这一类型的交互。如图 5.65 所示。

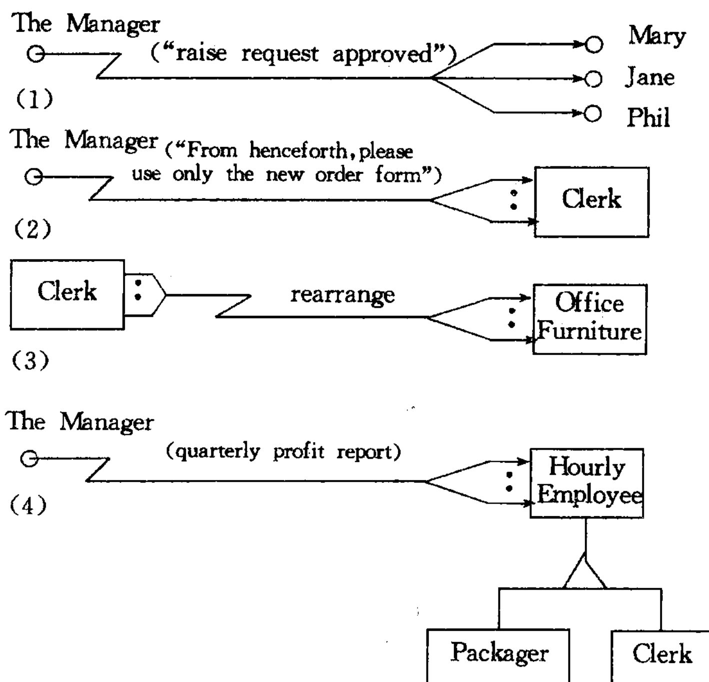


图 5.65 与多个对象交互

图 5.65 中,图 5.65(1)表明:一个对象与多个对象发生交互;图 5.65(2)表明:一个对象与一个对象类中的多个对象发生交互;图 5.65(3)表明:一个对象类中的多个对象与另一个对象类中的多个对象发生交互。在这一抽象级上,我们可以认为类 clerk 中的所有雇员一起工作,布置屋内所有设备;图 5.65(4)表明:公司经理向职员和工人发布消息。

(5) 双向交互

有时,一对交互是紧密相关的。在这种情况下,OSA 允许把这样的一对交互看作是同一交互,因此,可以把它简写,如图 5.66(1)所示。

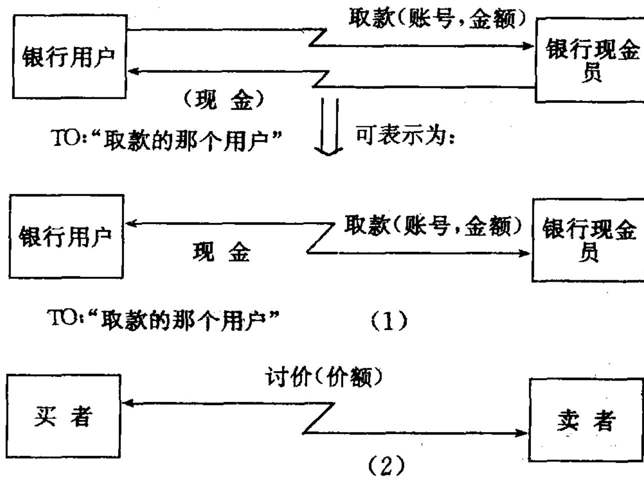


图 5.66 双向交互示例

如果双向交互两边的描述是一样的,则只需在 Z 形附近给出一次交互描述。如图 5.66(2)所示。

(6) 特殊的交互活动

分析员在系统分析中,常常需要“ACCESS”,“MODIFY”,“REMOVE”,“DESTROY”,“ADD”,“CREATE”这样一些活动。由于这些活动与正在建造的系统模型发生交互作用,因此把它们称为特殊的交互活动。

“ACCESS”活动用于获得对象的有关信息;“MODIFY”活动用于改变现有的对象;“REMOVE”活动用于把某一对象类中的指定对象从该类中删除,但该对象还在系统中;“DESTROY”活动用于删除一个对象;“ADD”活动用于把一个对象添加到某一对象类中,“CREATE”活动用于创建一个对象,并把该对象添加到某一对象类中。

(7) “公告板”通信

通过一个公告板协议,进行对象之间的通信,这是一种常用的方法。一个对象“贴出”一条消息,其他对象“阅读”之。我们可以使用“ACCESS”,“DESTROY”,“CREATE”等,建立“公告板”通信的模型。图 5.67 给出了这类通信的例子。

其中,“Secretary”类的对象可以创建“Message”类的对象,也可以删除“Message”类的对象。“Person”类的一个对象可以访问“Message”类的所有对象,也可以阅读“secretary”对象当前创建的“Message”对象。

系统中,无论是概念实例,还是具体实例,有关它们之间的“公告板”通信,均可采用这一模型化技术。

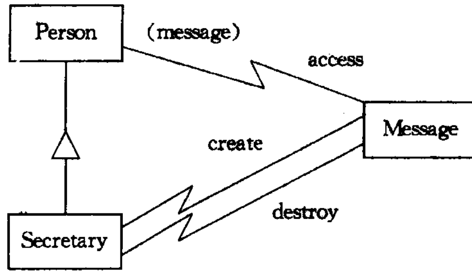


图 5.67 "公告板"通信

(8) 穿越模型边界的交互

如果一个交互，其起始对象或终点对象不在分析模型中，即一个对象或消息从一个未被说明的起始处进入分析模型，或一个对象或消息从分析模型进入尚未说明的终点，OSA 把这样的交互称为穿越模型边界的交互。

OSA 用缺"头"或缺"尾"的交互链来表示这一类型的交互。如图 5.68 所示。

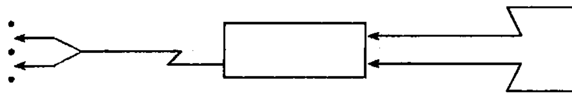


图 5.68 超越边界的交互

(9) 不间断的交互

交互可以是连续不断的。例如，巡航控制系统中的速度传感器，它连续不断地检测航行速度，并不间断地显示当前速度。实际中，往往是尽管起始对象不断地发送信息，但终点对象并非不间接地接收之。例如，我们带的手表，它不断地"报告"时间，而我们只在需要时才"接收"手表"发来的消息"。

对于这样一类的交互，OSA 使用双箭头线表示。如图 5.69 所示。

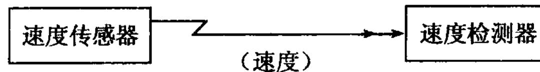


图 5.69 不间断交互示例

(10) 同类对象之间的交互

有时，我们需要建立同类对象之间的通信模型。同类对象之间的通信可以分为两种情况。一种是一个对象与其自身进行通信；另一种是同类对象相互之间的通信。

对于第一种情况，OSA 借助于该对象所对应的状态网，用状态或状态转换来描述一个对象与自身的通信。

对于第二种情况，OSA 使用该类状态网的不同拷贝来描述同类对象中不同对象之间的交互。

除此之外，还可以使用 ORM 层上的对象类，来表示同类对象中不同对象之间的通信。如图 5.70 所示。

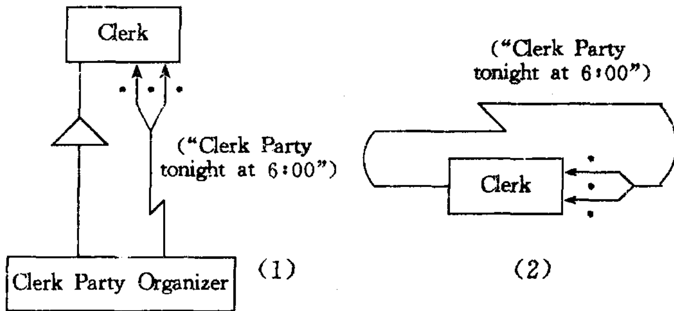


图 5.70 同一对象类中的对象交互

3. 交互的约束、继承

(1) 交互的继承

"一般/特殊"是系统模型化的抽象方法和组织方法。它提供了继承机制。特殊类的一个对象可以继承一般类对象所具有的性质。对于交互而言，也可以应用这一机制，使所有特殊对象继承为一般对象所定义的交互，并且还具有一般对象所没有的交互。图 5.71 给出了这样的一个例子。

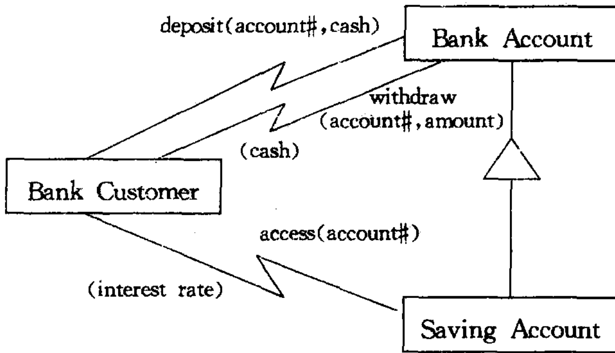


图 5.71 交互的"一般/特殊"

其中，"银行账"是"存款账"的一般类。由于"存款账"是"银行账"的特殊类，因此它继承了"银行账"的所有交互，即图中的"存款"、"取款"等。从而用户可以与"存款账"发生"存款"、"取款"交互。这些交互是隐式的，不必在交互模型中标出。另外，特殊类中的对象还可以直接与其他对象发生交互。例如图 5.71 中的用户可以访问"存款账"，以便了解当前的利率。由于不是所有的"银行账"均具有利率信息，因而要了解这一信息就不能与"银行账"发生交互。

(2) 交互的约束

OSA 提供了两种关于交互的约束。一种是交互的实时约束，一种是交互的一般约束。

交互的实时约束限制了完成交互所需要的时间。完成交互意指参与交互的所有起始对象和终点对象均实现了它们自己的作用。在 OSA 中，交互实时约束的形式与对象行为的实时约束一样。如图 5.72 所示。

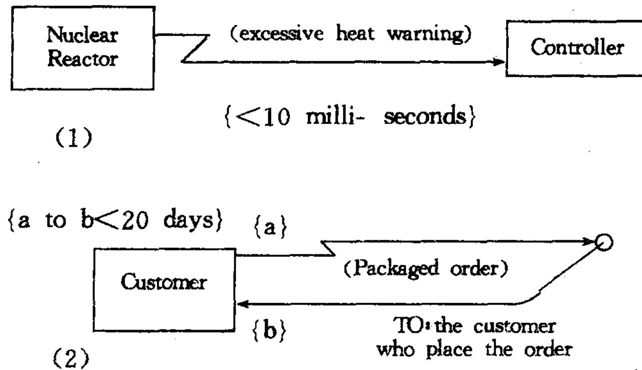


图 5.72 限制时间的交互

在图 5.72(2)中，{a}，{b}是交互序列的起始标志和终止标志。约束{a to b<20 days}意指在 20 天内必须完成这一交互序列。

交互的一般约束用于强调各种不同的交互问题。例如，在通信模型中，可以使用一般约束来限制通信频率、通信质量以及消息发送和消息接收的时间等；在异步交互模型中，可以使用一般约束来规定哪个对象具有较高的交互优先级。图 5.73 中给出了这一用途的一般约束。

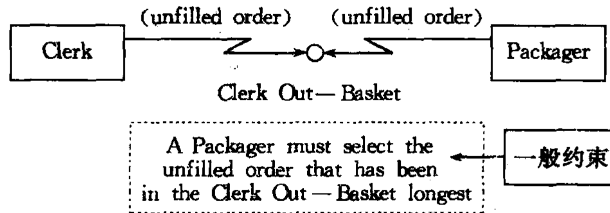


图 5.73 一般约束示例

4. 对象交互模型小结

在 OSA 的交互模型中，主要涉及了以下三个问题：

(1) 交互的基本元素以及描述对象之间交互的基本格式；

(2) 如何建立各种不同交互类型的模型，如同步交互、异步交互等；

(3) 交互约束和继承。

OSA 提出的交互基本格式，表达了交互的本质。各种不同交互类型的表示，显示了这一基本格式的表达能力。从而，形成了 OSA 对象交互模型的独特风格。

交互模型，无论是面向过程的交互模型，还是面向对象的交互模型，都是分析模型的一个重要组成部分。

结构化分析使用了面向过程的交互模型，并且使用 DFD 表示之。

OOA[Coad, 1991]的交互模型是基于消息连接的，表示了一个对象对其他对象服务的相关性。OMT[Rumbaugh, 1991]使用事件跟踪，建立对象之间事件相互作用模型，并使用面向过程的 DFD 来表示对象间的信息交互。

OSA 与其他交互模型的最大区别是允许灵活地描述大量类型的交互。

图 5.74 及图 5.75 是 OSA 对象交互模型中所用的表示符号。

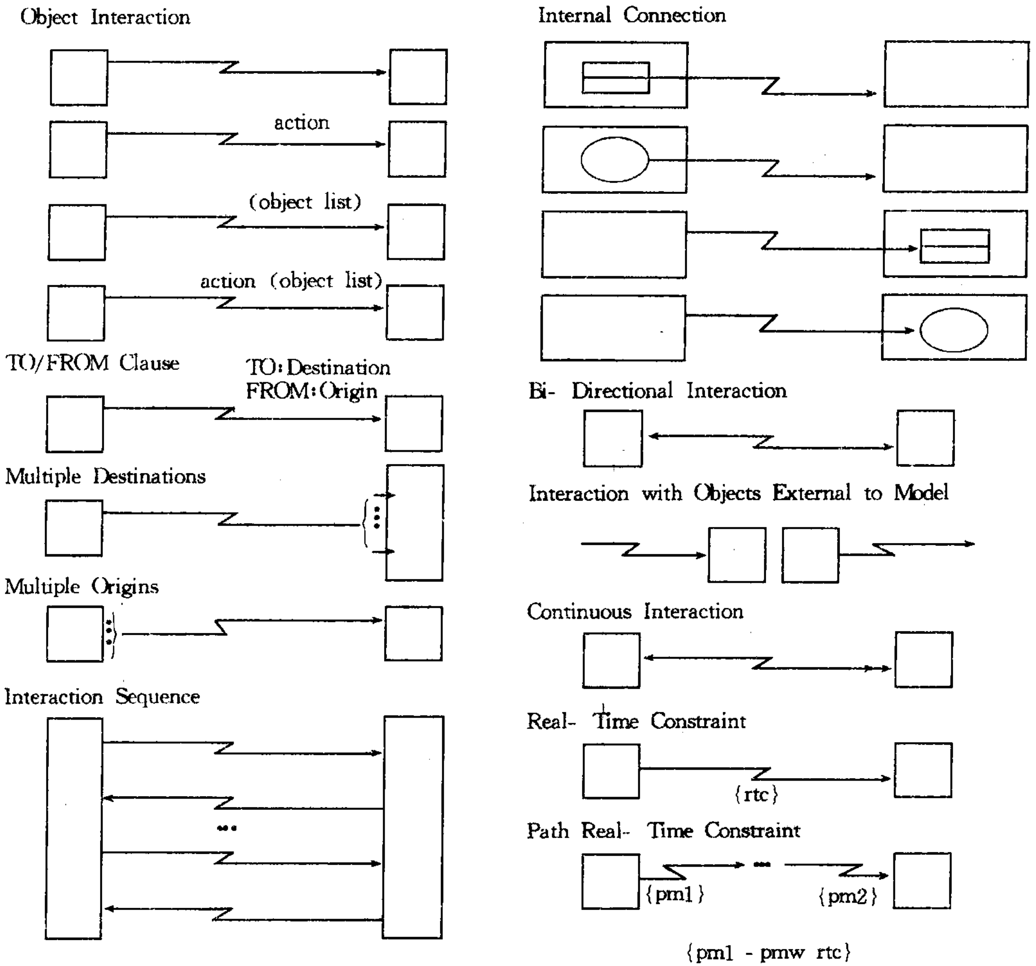


图 5.74 OSA 对象交互模型中所用的表示符号(1)

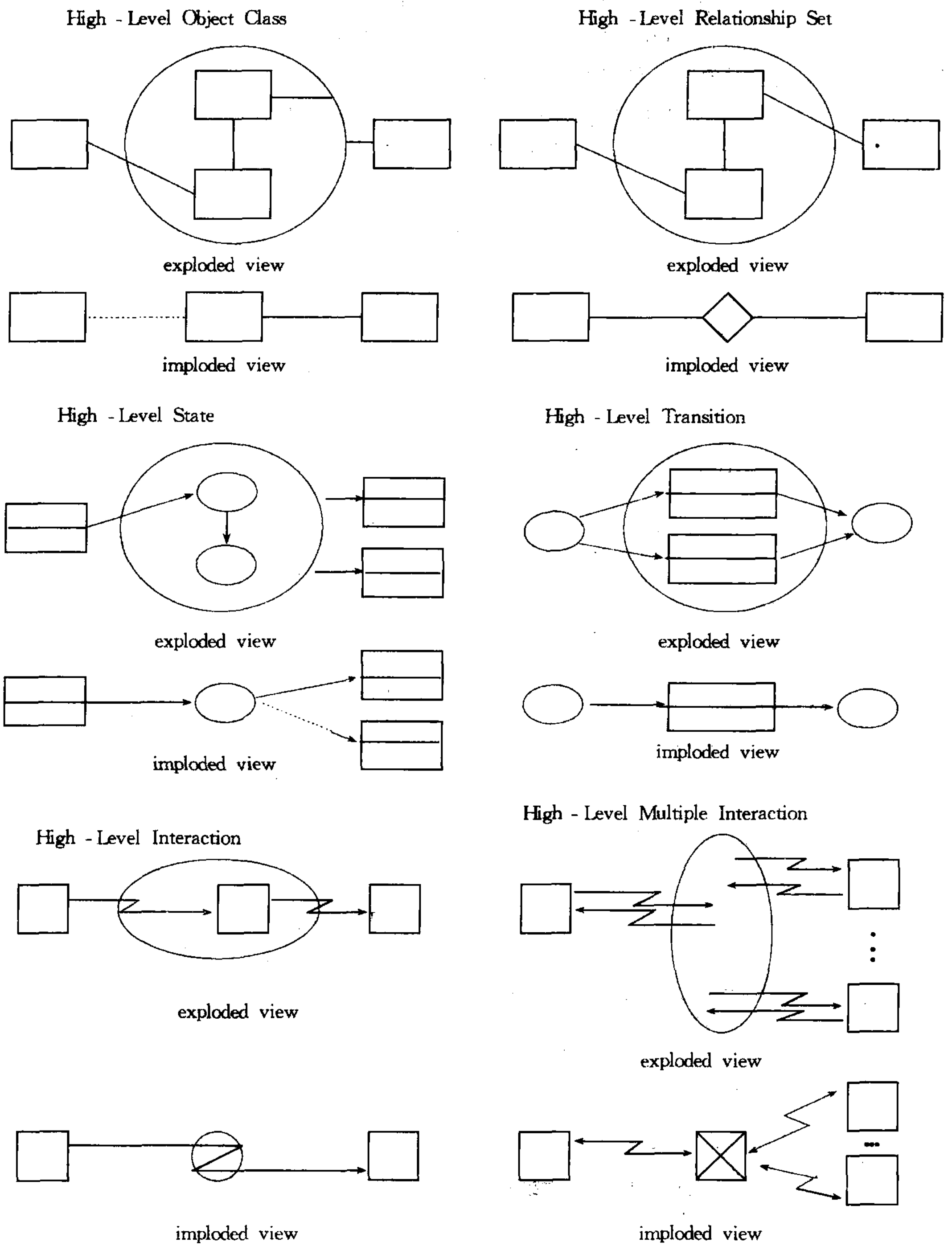


图 5.75 OSA 对象交互模型中所用的表示符号(2)

习 题 五

1. 解释以下术语：

对象 属性 操作 关联 状态 事件 类 链 泛化 聚合 接口消息 依赖

并举例说明之。

2. 简要回答：

对象的构成与表示；

对象的基本特性；

类图的构成；

状态图的构成；

描述关系所使用的概念；

在统一软件开发过程中，各阶段所要完成的主要工作；

统一软件开发过程中的核心工作流；

统一软件开发过程的核心思想；

面向对象方法为什么要从多个侧面建立系统模型。

3. 分析：

(1) 对象操作与对象状态之间的关系；

(2) 引入“操作”以及其同义词“方法”的目的及必要性；

(3) 在描述客观事物方面，面向对象方法与结构化方法提取信息的不同角度，以及对建造的系统模型所产生的影响；

(4) 面向对象方法与结构化方法在控制信息组织复杂性方面引入的机制。

4. 实践题：

(1) 假定教务管理包括以课程为中心进行资源(教师、教室、学生)配置，并根据各科考试成绩进行教学分析。在这一假定下，结合实际情况，给出教务管理系统的需求陈述，建立该系统的类图，并选取其中一个典型的对象类，给出它的状态图。

(2) 一个目录文件包含该目录中所有文件信息。一个文件可以是一个普通文件，也可以是一个目录文件。绘制描述目录文件和普通文件的类图。

(3) 考虑使用网络打印机进行打印时出现的各种情况，绘制顺序图。

(4) 一个光盘商店从事订购、出租、销售光盘业务。光盘按类别分为游戏、CD、程序三种。每种光盘的库存量有上下限，当低于下限时要及时定货。在销售时，采取会员制，即给予一定的优惠。请按需求建模，并对图中的各种元素进行简要说明。

(5) 用状态图描述一部电梯的运行。

(6) 阅读参考书中有关 Jackson 方法的讲解，并在“概念与表示”、“过程”、“工具”三个方面进行比较。

第六章　软件测试

软件产品与其他产品不同，其最大的成本是检测软件错误、修正错误的成本，以及为了发现这些错误所进行的设计测试程序和运行测试程序的成本。据有关统计，软件测试作为软件质量保证的一个重要组成部分，在整个软件开发中占据了一半或一半以上的工作量，因此，对软件测试技术的研究一直是人们关注的课题，

本章主要针对程序测试，介绍两种常用的测试技术——基于“白盒”的路径测试技术和基于“黑盒”的测试技术。

6.1　软件测试目标与软件测试过程模型

6.1.1　软件测试目标

关于软件测试目标，人们在长期的实践中逐渐有了一个统一的认识。一般地说，其第一目标是预防错误。如果能够实现这一目标，那么就不需要修正错误和重新测试。可惜的是，由于软件开发至今离不开人的创造性劳动，这一目标几乎是不可实现的。因此，测试的目标即第二目标只能是发现错误。

软件错误的表现形态是多种多样的，并且，不同的错误可以有同样的表现形态，因此，即便知道一个程序有错误，也可能不知道该错误是什么。这样，要实现第二目标，也需要研究软件测试理论、技术和方法。

人们关于软件测试目的的认识，大体经历了五个阶段。第一阶段认为软件测试和软件调试没有什么区别；第二阶段认为测试是为了表明软件能正常工作；第三阶段认为测试是为了表明软件不能正常工作；第四阶段认为测试仅是为了将已察觉的错误风险减少到一个可接受的程度；第五阶段认为测试不仅仅是一种行为，而是产生低风险软件的一种认识上的训练。

根据以上讨论，软件测试可定义为：按照特定规程，发现软件错误的过程。在 IEEE 提出的软件工程标准术语中，对软件测试下的定义是：“使用人工或自动手段，运行或测定某个系统的过程，其目的是检验它是否满足规定的需求，或是清楚了解预期结果与实际结果之间的差异。”

着重指出的是，软件测试和软件调试在目的、技术和方法等方面存在着很大区别，主要表现在以下几个方面：

(1) 测试从一个侧面证明程序员的“失败”；而调试是为了证明程序员的正确。

(2) 测试以已知条件开始，使用预先定义的程序，且有预知的结果，不可预见的仅是程序是否通过测试。调试一般是以不可知的内部条件开始，除统计性调试外，结果是不可预见的。

(3) 测试是有计划的，并要进行测试设计；而调试是不受时间约束的。

(4) 测试是一个发现错误、改正错误、重新测试的过程；而调试是一个推理过程。

(5) 测试的执行是有规程的，而调试的执行往往要求程序员进行必要推理以至知觉的“飞跃”。

(6) 测试经常是由独立的测试组在不了解软件设计的条件下完成的;而调试必须由了解详细设计的程序员完成。

(7) 大多数测试的执行和设计可由工具支持,而调试时,程序员能利用的工具主要是调试器。

6.1.2 测试过程模型

软件测试是一有规则的过程,包括测试设计、测试执行以及测试结果比较等。这一过程见图 6.1。

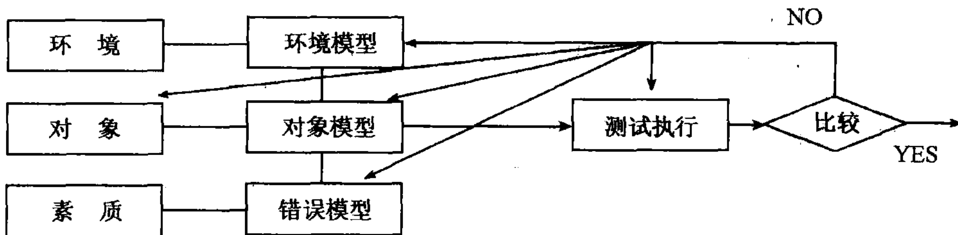


图 6.1 软件测试过程模型

其中,程序环境包括支持其运行的硬件、固件和软件,例如计算机、终端设备、网卡、操作系统、编译系统、实用程序等。一般来说,程序环境经过了生产厂家的严格测试,出现错误的概率比较小,软件可靠性较好。因此,对环境的抽象——环境模型,只考虑计算机指令系统、操作系统宏指令、操作系统命令以及高级语言语句等。

另外,为了测试,我们必须简化程序概念,形成被测对象的简化版本,即程序模型。不同测试技术,对同一被测对象——程序,可产生不同的程序模型。这一简化或着重于程序的控制结构,或着重于处理过程,于是形成了所谓的"白盒"测试和"黑盒"测试。如果程序的简单模型不能解释未料到的行为,则必须修改程序模型,使其包含更多的事实和细节。如果还有问题,则要考虑是否修改程序。

由于参与软件开发的人员众多,且各有各的侧重面,因此,他们对"什么是错误"往往在认识上是不一致的。有的问题,对开发者来说,它称不上是一个"错误",而对测试人员来说,它就是一个"错误"。为了统一认识,必须定义"什么是错误",即给出"错误模型"。

在建立了环境模型、程序模型、以及错误模型的基础上,才能执行测试及测试结果的比较。如果预料结果与实际结果不符,就要考虑是否是环境模型、程序、程序模型和错误模型的问题。

6.2 软件测试技术

软件测试技术大体上可分为两大类:一类是白盒测试技术,典型的是路径测试技术;一类是黑盒测试技术,又称为功能测试技术,包括事务处理流程技术、状态测试技术、定义域测试技术等。白盒测试技术依据的是程序的逻辑结构,而黑盒测试技术依据的是软件行为的描述。在具体介绍这两种技术之前,首先让我们讨论一下软件错误的分类。

关于软件错误分类,至今没有统一的标准。针对本章介绍的测试技术和软件体系结构,可

以把软件错误分为结构错误、数据错误。编程错误和接口错误。其中结构错误包括逻辑错误、数据流错误和初始化错误等;接口错误包括内部接口错误、外部接口错误、资源管理错误、操作系统错误、集成化错误以及系统错误等。

人们在长期测试实践中,首先提出了路径测试技术,以发现程序中的错误。

6.2.1 路径测试技术

如上所述,路径测试技术依据的是程序的逻辑结构。表达这一结构的有力工具是控制流程图。通过合理地选择一组穿过程序的测试路径,以实现达到某种测试度量。例如,选择足够的路径,确保每一个语句至少执行一次。

路径测试对错误的假定是软件通过了与预想不同的路径。

1. 基本概念

(1) 控制流程图

控制流程图是程序控制结构的图形表示,其基本元素是过程块(简称过程)、结点、判定。如图 6.2 所示。

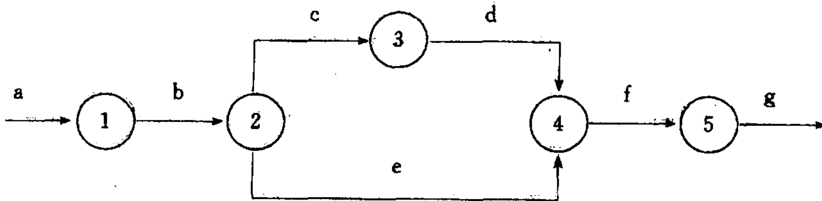


图 6.2 控制流程图

其中,①,③,⑤是过程块,②是一个判定,④是一个结点。过程块是既不能由判定,也不能由结点分开的一组程序语句。其基本属性是:如果过程块中的某个语句被执行,那么块中的所有语句都被执行。按照测试的观点,对于一个过程块,如果其操作细节不影响控制流,那么这些操作就被视为是不重要的;如果其操作细节影响控制流,那么这一影响只能在其后的判定中表现出来。

判定是一个程序点,此处控制流可以分叉。在一个判定中,可以包含处理成分。判定可以是二分支的,也可以是三分支的。按照测试的观点,判定和语言上的判定语句没有本质上的差异。

结点是程序中的一个点, 此处控制流可以结合。例如, 汇编语言中跳转指令的目标, PASCAL语言中的语句标号。

控制流程图与程序流程图之间的差异是在控制流程图中,不显示过程块的细节,而在程序流程图中,着重于过程属性的描述。

(2) 路径

路径是一串指令或语句。它在一个入口、结点、判定处开始,在另一入口(或同一入口)、结点、判定或出口处结束。显然一条路径可一次或多次地穿过几个结点、过程块或判定。

路径是由链支组成的。链支由其连接的“结点对”命名。例如图 6.2 中的(①,②),(④,⑤)等;也可以直接命名,例如 a, b, c, …。路径的长度由其链支数目决定。对于软件测试,路

径的严格含义是它在程序的入口处开始，在出口处结束。

2. 路径测试策略

为了回答什么是“完整的测试”，路径测试有着各种策略。下面以图 6.3 所示的程序控制流程图为例(其中，为了说明方便起见，分支和过程块使用了习惯上的符号)，分别对这几种典型的测试策略进行介绍。该例子中有两个判断，每个判断都包含复合条件的逻辑表达式，并且以符号“∧”表示“与”运算，以符号“∨”表示“或”运算，用上画线“—”表示“非”运算。

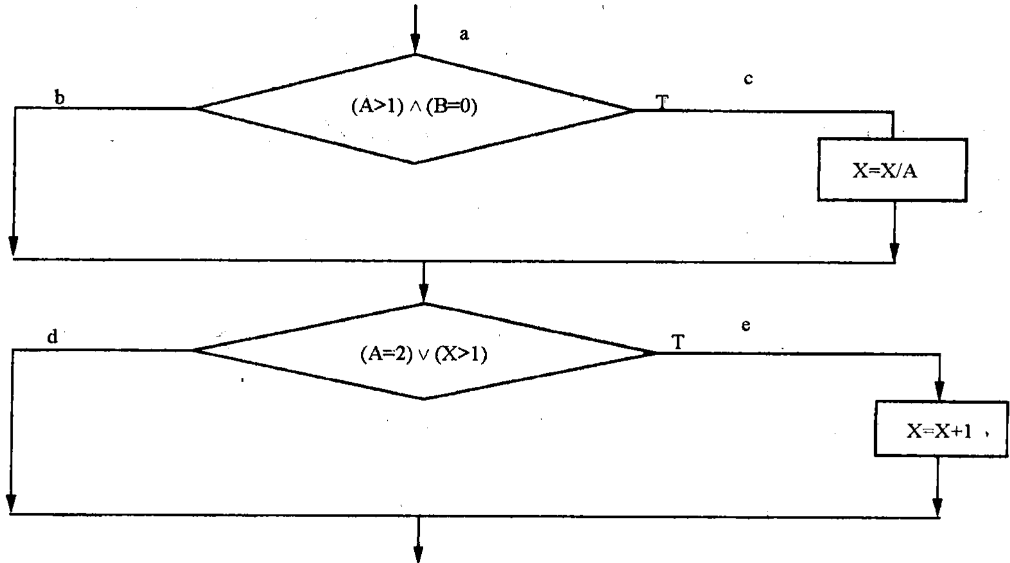


图 6.3 某被测程序的控制流程图

分析图 6.3 可知，该程序流程有 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 和 d, 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。

(1) 路径测试(PX)

执行所有可能的穿过程序的控制流程路径。一般来说，这一测试严格地限制为所有可能的入口/出口路径。如果遵循这一规定，则我们说达到了 100% 路径覆盖率。在路径测试中，该策略是最强的，但一般是不可实现的。

在图 6.3 所示的例子中，要想实现路径覆盖，可选择以下一组测试用例(测试用例指的是为了发现程序中的故障而专门设计的一组或多组数据。在本章中，规定测试用例的格式为：【输入的(A, B, X)，输出的(A, B, X)】)。

测试用例	覆盖路径
【(2,0,4),(2,0,3)】	L1
【(1,1,1),(1,1,1)】	L2
【(1,1,2),(1,1,3)】	L3
【(3,0,3),(3,0,1)】	L4

(2) 语句测试(P1)

至少执行程序中所有语句一次。如果遵循这一规定,则我们说达到了100%语句覆盖率(用C1表达)。

在图6.3所示的例子中,只要设计一种能通过路径ace的测试用例,就覆盖了所有的语句。所以可选择测试用例如下:

【(2,0,4),(2,0,3)】 覆盖L1

语句覆盖的逻辑覆盖程序较低。如果在图6.3所给出的程序控制流程中,两个判断的逻辑运算有问题,例如,第一个判断中的逻辑运算符"∧"错写成了"∨",或者第二个判断中的逻辑运算符"∨"错写成了"∧",利用上面的测试用例,仍可覆盖所有4个可执行的路径,而发现不了判断中逻辑运算符出现的错误。与后面所介绍的其他覆盖相比,语句覆盖是最弱的逻辑覆盖准则。

(3) 分支测试(P2)

至少执行程序中每一分支一次。如果遵循这一规定,则我们说达到了100%分支覆盖率(用C2表示)。

分支覆盖比语句覆盖标准强。但若程序中分支的判定是由几个条件联合构成时,它未必能发现每个条件的错误。

例如,对于图6.3所示的程序控制流程,如果选择路径L1和L2,就可得到实现分支覆盖的测试用例:

【(2,0,4),(2,0,3)】 覆盖L1

【(1,1,1),(1,1,1)】 覆盖L2

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L3

【(3,0,3),(3,1,1)】 覆盖L4

分支覆盖是一种比语句覆盖稍强的逻辑覆盖,因为如果通过了各个分支,则各语句也都覆盖了。但分支覆盖还不能保证一定能查出在判断的条件中存在的错误。例如,在图6.3所示的程序控制图中,若第二个分支X>1错写成X<1,利用上述两组测试用例进行测试,无法查出错误。因此,需要更强的逻辑覆盖准则去检验判定的内部条件。

(4) 条件组合测试

分支测试虽然是一种比语句测试稍强的逻辑覆盖测试,但它还不能保证一定能查出组成判定的各条件中存在的错误。于是又出现了逻辑覆盖性更强的条件组合测试。

条件组合测试,就是设计足够的测试用例,使每个判定中的所有可能的条件取值组合至少执行一次。如果遵循这一规定,则我们说就实现了条件组合覆盖。只要满足了条件组合覆盖,就一定满足分支覆盖。

在条件组合覆盖技术发展过程中,最初,在设计测试用例时,人们只考虑使分支中各个条件的所有可能结果至少出现一次。但发现该测试技术未必能覆盖全部分支。例如,在图6.3中有四个条件:A>1,B=0,A=2,X>1。

条件A>1取真值标记为T1,取假值标记为$\overline{T1}$;

条件B=0取真值标记为T2,取假值标记为$\overline{T2}$;

条件A=2取真值标记为T3,取假值标记为$\overline{T3}$;

条件 X>1 取真值标记为 T4，取假值标记为$\overline{T4}$。

在设计测试用例时，要考虑如何选择测试用例，实现 T1，$\overline{T1}$，T2，$\overline{T2}$，T3，$\overline{T3}$，T4，$\overline{T4}$的全部覆盖。

例如，可设计如下测试用例，实现条件覆盖：

测试用例	通过路径	条件取值	覆盖分支
【(1,0,3)，(1,0,4)】	L3	$\overline{T1}$ T2 $\overline{T3}$ T4	b, e
【(2,1,1)，(2,1,2)】	L3	T1 $\overline{T2}$ T3 $\overline{T4}$	b, e

从上面的测试用例，我们可以看到该组测试用例虽然实现了判定中各条件的覆盖，但没有实现分支覆盖，因为该组测试用例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。在发现条件覆盖技术的不足后，人们又进一步提出了条件组合覆盖技术。

例如，在图 6.3 所示的例子中，要满足条件组合覆盖，设计的测试用例必须满足以下八种条件组合：

① (A>1)，(B=0)，可标记为 T1，T2；

② (A>1)，(B≠0)，可标记为 T1，$\overline{T2}$；

③ (A≤1)，(B=0)，可标记为$\overline{T1}$，T2；

④ (A≤1)，(B≠0)，可标记为$\overline{T1}$，$\overline{T2}$；

⑤ (A=2)，(X>1)，可标记为 T3，T4；

⑥ (A=2)，(X≤1)，可标记为 T3，$\overline{T4}$；

⑦ (A≠2)，(X>1)，可标记为$\overline{T3}$，T4；

⑧ (A≠2)，(X≤1)，可标记为$\overline{T3}$，$\overline{T4}$。

我们可以采用以下四组测试数据，实现条件组合覆盖。

测试用例	覆盖条件	覆盖组合号	通过路径
【(2,0,4)，(2,0,3)】	T1 T2 T3 T4	①，⑤	L1
【(2,1,1)，(2,1,2)】	T1 $\overline{T2}$ T3 $\overline{T4}$	②，⑥	L3
【(1,0,3)，(1,0,4)】	$\overline{T1}$ T2 $\overline{T3}$ T4	③，⑦	L3
【(1,1,1)，(1,1,1)】	$\overline{T1}$ $\overline{T2}$ $\overline{T3}$ $\overline{T4}$	④，⑧	L2

这组测试用例实现了分支覆盖，也实现了条件的所有可能取值的组合覆盖，但没有实现所有路径的全部覆盖，因为路径 L4 没有被覆盖。这说明条件组合覆盖还不能实现完全的测试。

可见，测试覆盖率定量地描述了一个或一组测试的效率（或称测试完成程度）。

在以上几种策略中，没有涉及一个判定所依据的变量值。如果所有判定所依据的变量值以及该判定相关的过程，与其他变量没有任何关系，则可以得到该判定的各种组合，所以路径是可达的；如果所有判定所依据的变量值以及该判定相关的过程，与其他变量有一定关系，此时有的路径就可能是不可达到的。

在路径测试中，判定处求值的逻辑函数，称为判断。例如，

"A>0"

显然，程序在一个判定之后的行为取决于该判定相关变量的值。实际上，每一选取的路径都有一组布尔表达式，称为路径判断表达式。例如，

$$\begin{cases} X1+3X2+17\geqslant 0, \\ X3=17, \\ X4-X1\geqslant 14X2 \end{cases}$$

对于这一路径判断表达式，或有解，或有无穷多解，或无解。如果无解，那么该路径是不可达的。

对于路径判断表达式，如果能够找到一种有效的办法，使其路径是可达的，这一过程称为路径敏化。路径敏化一般来说，不存在一种通用的算法，在实际测试中，往往从几个不同的角度出发，力求找出不可解的问题。

显然，判断覆盖率是更强的一种覆盖。

3. 路径选取与用例设计

在 IEEE 单元测试标准中，最小的强制性测试需求是语句覆盖率。在 IBM 单元测试标准中，最小的强制性测试需求是语句和分支覆盖率。

为了得到 C1 + C2 覆盖，要选取足够的路径。对于图 6.4 所示的例子：

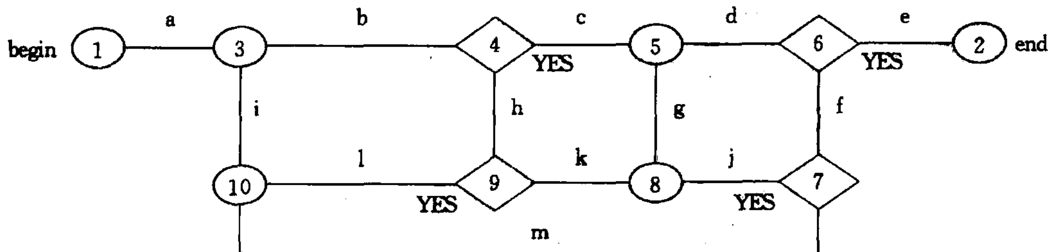


图 6.4 路径选取示例

要选取如下路径：abcde，abhkgde，

abhlibcde，abcdfjgde，

abcdfmibcde。

路径选取的一般规则是：

- 选择最简单的、具有一定功能含义的入口/出口路径；
- 对已选的路径进行演化，选取无循环的路径；选取短路径、简单路径；
- 选取没有明显功能含义的路径，此时要研究这样的路径为什么存在，为什么没有通过功能上合理的路径得到覆盖。

对于循环，为了选取路径，需要把循环进行分类："单循环"、"嵌套循环"、"级联循环"和"混杂循环"，并针对不同类的循环，给出相应的路径选取规则：

(1) 单循环

单循环又可分为以下三种情况：

① 最小循环次数为零，最大次数为 N，且无"跳跃"值

选取"旁通循环(零次循环)的路径；

对循环控制变量指定为负数；

一次通过循环；

典型的重复次数；

重复次数为 N－1；

重复次数为 N；

重复次数为 N＋1。

② 非零最小循环次数，且无"跳跃"值

重复次数为最小次数减 1；

重复次数为最小次数；

重复次数为最小次数加 1；

一次通过循环，除非已经覆盖；

二次通过循环，除非已经覆盖；

典型的重复次数；

重复次数为最大次数减 1；

重复次数为最大次数；

重复次数为最大次数加 1。

③ 具有跳跃值的单循环

除把每一"跳跃"边界，按"最小循环次数"和"最大循环次数"处理外，其他均同前两种单循环的路径选取规则。

(2) 嵌套循环

嵌套循环的路径选取策略是：

① 在最深层的循环开始，设定所有外层循环取它的最小值；

② 测试最小值、最小值＋1、典型值、最大值－1、最大值，与此同时，测试最小值减 1、最大值＋1 以及"跳跃值"边界；

③ 设定内循环在典型值处，按②测试外层循环，直到覆盖所有循环。

(3) 级联循环

如果在退出某个循环以后，到达另一个循环，且还在同一入口/出口路径上，则称这两个循环是级联的。其中，如果在退出某个循环以后，到达另一个循环，且循环的重复值与另一个循环的重复值相关，该循环还在同一入口/出口路径上，则可认为是嵌套循环。如果两个循环不在同一入口/出口路径上，则可认为它们是单循环。

综上所述，路径测试是一种强有力的单元测试技术。路径测试的目标是：执行足够的测试，确保实现某一确定的覆盖率。在路径测试中，一般要从最简单、最熟悉的从入口到出口的路径以及与正常路径有偏差的路径开始选取，继之按需要增加路径。并且对于循环还要适当地增加一些路径，以覆盖循环和循环组合：

无循环

一次循环

二次循环

比最大值小 1

最大值

最大值加 1

最小值减 1

在选取测试路径的基础上，为实现某一特定的测试覆盖率进行用例设计。测试用例指的

是为了发现程序中的故障而专门设计的一组或多组数据。

6.2.2 事务处理流程测试技术

事务处理流程是系统行为的一种表示方法，为功能测试建立了程序的动作模式。其中，使用了控制流程的概念成分，例如链支、结点等。应该说，不论是结构测试，还是功能测试，基于结点、链支的图形表示技术都是一种强有力的概念工具。这种测试技术的基本步骤可概括为：定义有用的图形模式，设计必要的测试用例以覆盖之。

1. 事务与事务流程

事务是从系统用户的角度出发所见到的一个工作单元。一个事务由一系列操作组成，其中，某些操作由系统执行，某些操作由用户或系统之外的设备执行。例如，在联机信息检索系统中，一个事务可以是：

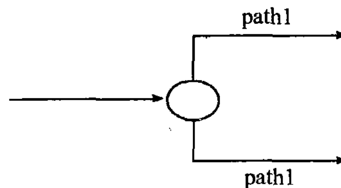
接受输入
有效输入
向用户发送礼节性信息
进行输入处理
检索文件
向用户请求指令
接受输入
有效输入
处理请求
更改文件
传送输出
记录事务注册和清除(结束)

由上可见，事务有其“产生”和“消亡”。

一个系统的行为表现为多个事务的执行，这一行为可抽象为事务处理流程图。

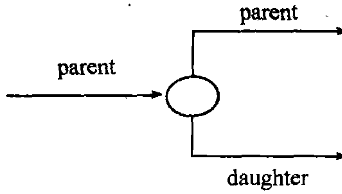
事务处理流程图与控制流程图的类同点是使用了相同的概念成分，例如处理(对应于过程块)、分支、结点。它们不同之处是，事务流程图是一种数据流程图，从操作应用的历史，观察数据对象。因此，链支和过程块的定义有所差异。另外事务流程图的判定结点可能是一个复杂的过程，从而事务流程图中的判定只能是“抽象”。第三点不同之处是事务流程图中存在“中断”的作用，中断可以把一个过程等价地变换为具有繁多出口的链支，对此也要予以抽象。具体地说：

事务流程图中的判定可以是：

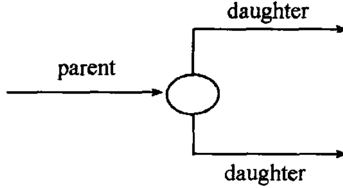


即事务处理将选取一个分支执行，其语义与控制流程图的分支相同。

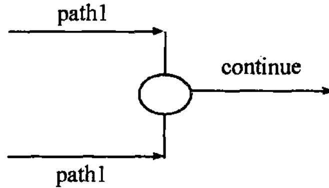
事务流程图中的判定也可以是：



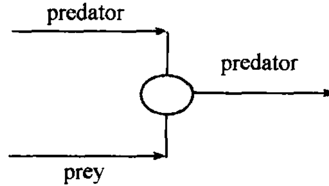
即事务处理产生一个新事务，由此这两个事务继续执行，称为“并生”。
事务流程图中的判定还可以是：



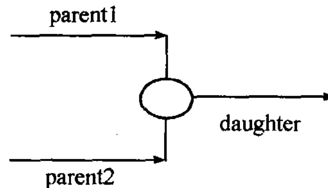
即事务处理产生两个新事务，称为“丝分裂”。
事务流程图的结点可以是：



即事务的不同活动可以汇集一处，其语义与控制流程图相同。
事务流程图的结点也可以是：



即一个事务可以被另一事务“吸食”，称为“吸收”。
事务流程图的结点还可以是：



即两个事务结合后生成一个新的事务，称为“结合”。
由此可见，事务流程图要比控制流程图在语义上更为复杂。
事务处理流程图往往具有很差的结构，其主要原因是：

① 它是一种处理流程，人可包含在循环、判定中；

② 某些部分可能与我们不能控制的行为有关；

③ 性能的增加，可使事务数目和单个事务处理流程具有相当的复杂性；

④ 事务流程表达的系统模型更接近现实，例如中断、多任务、同步、并行处理……，所有这一切已不再适合结构化概念。

2. 事务处理流程测试步骤

事务处理流程测试的步骤大体分为以下三步：

第一步：获得事务处理流程图

由于在实际的软件工程中，不论是系统分析规约，还是系统设计规约，均采用了非形式化的表达技术，因此要获得测试可用的事务处理流程通常是最困难的。

第二步：浏览、复审

这一步主要是对事务分类，为测试用例的设计奠定基础。

第三步：用例设计

设计足够的测试用例，实现 C1 + C2 覆盖。并且

- 循环、最大值、域界，选取附加的事务处理路径；
- 具有很大危险或潜在危险的事务选取附加的路径，包括“并生”、“丝分裂”、“吸收”和“结合”；
- 选择测试路径的一个子集，作为系统功能测试的基础。

对于事务处理流程测试，为了实施以上活动，要解决以下几个问题：

(1) 路径选取

选取一个基于功能的、切合实际的事务路径覆盖集，其中，找出从事务处理流程入口到出口的最复杂、最长和异常的路径。特别是对那些异常路径，以期发现：

遗漏互锁

重复互锁

接口

交叉影响

重复处理等问题

并建立路径目录。

(2) 激活

一般来说，80%—95%的路径(C1 + C2)容易激活，但余下的路径是不易激活的。不易激活的路径一般来说或是事务流程的错误，或是设计上的错误。

(3) 测试设备

为了支持事务处理流程测试，应配备中心事务调度设备和事务跟踪装置，以便有效地实施测试。

(4) 测试数据库

由于事务处理流程测试的复杂性，应建立相应的测试数据库，以保留测试数据，支持有效的测试配置。

综上所述，事务处理流程测试技术是将路径测试技术用于功能测试的结果，有关单元和程序的路径测试方法可用于解释基于事务处理流程的功能测试；对于事务处理流程测试而言，最

大的问题和最大的代价是获得事务处理流程图;一般地,事务处理流程测试要求达到 C1 + C2 路径覆盖;但是,大多数错误将在奇异的、不受注意的或非法的路径上发现;更为重要的是,在事务处理流程测试中,如果设计的测试能与设计者讨论,将可以发现比运行测试更多的错误。

6.2.3 其他功能测试技术简述

黑盒测试完全不考虑程序的内部结构,而对软件功能规约或用户手册中所列的每项功能逐一进行测试。包括对正常和异常的输入(或操作)、出错处理、边界情况和极端情况等进行测试。下面就分别介绍几种典型的功能测试技术:

1. 等价类划分

等价类划分方法是把所有可能的输入数据即程序的输入域划分成若干部分(即若干等价类),然后从每一部分中选取数据作为测试用例。这里的等价类是指某个输入域的子集合。在该等价类中,各个输入数据对于发现程序中错误的几率是等效的。这样,只要选取一个等价类中的一个输入数据作为测试数据进行测试而发现错误,那么使用这一等价类中的其他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数据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他输入数据也同样查不出错误。这样,先把全部输入数据合理地划分为若干等价类,在每一个等价类中取一数据作为测试数据,就可用少量测试用例,而取得较好的测试效果。

(1) 划分等价类

对于等价类划分,人们从实践角度,经常从有效和无效的角度对输入数据进行等价类划分。

- 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不符合规格说明的要求。

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,这样获得的软件才具有较高的可靠性。

采用等价类划分的技术设计测试用例,应分两步进行,即首先划分等价类,然后确定测试用例。

划分等价类的方法是根据每个输入,找出两个或更多的等价类,并将其列表。下面给出几条确定等价类的参考原则:

① 如果某个输入条件规定了输入数据的取值范围,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,对输入条件限定为其数值为 1 到 100,则有效等价类是“1≤输入数据≤100”,两个无效等价类是“输入数据<1”或“输入数据>100” 。

② 如果某个输入条件规定了输入数据的个数,则可划分一个有效等价类和两个无效等价类。例如,在程序的功能规约中,规定“一名教师在一学期只能教授 1 到 2 门课程”。则有效等价类是“1≤教授课程≤2”,两个无效等价类是(不教授课程和教授课程超过 2 门)。

③ 如果输入条件规定了输入数据的一组可能取的值,而且程序可以对每个输入值分别进行处理,则可为每一个输入值确立一个有效等价类,而针对这组值确定一个无效等价类。例

如，在高校本科生管理系统中，要对大一、大二、大三、大四的学生分别进行管理，则可确定4个有效等价类为大一、大二、大三、大四的学生，一个无效等价类是所有不符合以上身份的人员的输入值集合。

④ 如果输入条件是一个布尔量，则可以确立一个有效等价类和一个无效等价类。

⑤ 如果某个输入条件规定了必须符合的条件，则可划分一个有效等价类和一个无效等价类。例如某系统中各数据项的关键字的首字符必须是K，则可划分一个有效等价类(首字符为K的输入值)，一个无效等价类(首字符不为K的输入值)。

⑥ 若在已划分的某一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为更小的等价类。

(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
……	……	……
……	……	……

再根据等价类来设计测试用例，过程如下：

① 为每一个等价类规定一个惟一的编号；

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。

之所以这样做，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他输入错误的检查。因此设计无效等价类的测试用例时应该仅包括一个未被覆盖的无效等价类。

2. 边界值分析

测试工作经验表明，大量的错误经常发生在输入或输出范围的边界上。因此，使用等于、小于或大于边界值的数据对程序进行测试，发现错误的概率较大。在设计测试用例时，应选择一些边界值，这就是边界值分析测试技术的主要思想。

边界值分析是一种最常用的黑盒测试技术。使用边界值分析设计测试用例可以遵循以下原则：

(1) 如果某个输入条件规定了输入值的范围，则应选择正好等于边界值的数据，以及刚刚超过边界值的数据作为测试数据。例如，若输入值的范围是"-1.0～1.0"，则可选取"-1.0"，"1.0"，"-1.001"，"1.001"作为程序的测试数据。

(2) 如果某个输入条件规定了值的个数，则可用最大个数、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1～255各记录，则可以选择1个、255个记录以及0个和256个记录作为测试的输入数据。

(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旧费，最低折旧费是0元，最高折旧费是100元。则可设计一些测试用例，使它们恰好产生0元和100元的折旧费结果。此外，还要设计测试用例，使输出结果为负值或大于100元。

这里要注意的是，由于输入值的边界不一定与输出值的边界相对应，所以直接应用输入边

界值不一定能直接得到输出边界值,而要产生超出输出值之外的结果也不一定办得到。但分析这些情况将有利于程序的测试工作。

(4) 根据规格说明的每个输出条件,使用前面的原则(2)。例如,一个网上检索系统,根据输入条件,要求显示最多 10 条的相关查询结果。可设计一些测试用例,使得程序分别显示 0、1 和 10 个查询结果,并设计一个有可能使程序显示 11 个查询结果的测试用例。

(5) 如果程序的规格说明中,输入域或输出域是有序集合(如顺序文件),在实践中,则经常选取集合的第一个元素、最后一个元素以及典型元素作为测试用例。

(6) 如果程序中使用了内部数据结构,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,作为测试数据。

(7) 分析规格说明,找出其他可能的边界条件。

通过以上原则的说明可以看出,边界值分析与等价类划分技术的区别在于:边界值分析着重于边界的测试,应选取等于、刚刚大于或刚刚小于边界的值的作为测试数据,而等价类划分是选取等价类中的典型值或任意值作为测试数据的。

3. 因果图

因果图是设计测试用例的一种工具,它着重检查各种输入条件的组合。而前面介绍的等价类划分和边界值分析由于没有考虑输入条件组合的情况,所以都不能发现这类错误。

要检查输入条件的组合首先把所有输入条件划分成等价类,它们之间的组合情况也相当多。而通过因果图,可以把用自然语言描述的功能说明转换为判定表,最后利用判定表来检查程序输入条件的各种组合情况。

因果图测试技术是通过为判定表的每一列设计一个测试用例,从而实现测试用例的设计与选择的。该方法生成测试用例的基本步骤如下:

(1) 通过软件规格说明书的分析,找出一个模块的原因(即输入条件或输入条件的等价类)和结果(即输出条件),并给每个原因和结果赋予一个标识符;

(2) 分析原因与结果之间以及原因与原因之间对应的关系,并画出因果图;

(3) 在因果图上标识出一些特定的约束或限制条件;

(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。

因果图中的基本符号表示形式参见图 6.5。

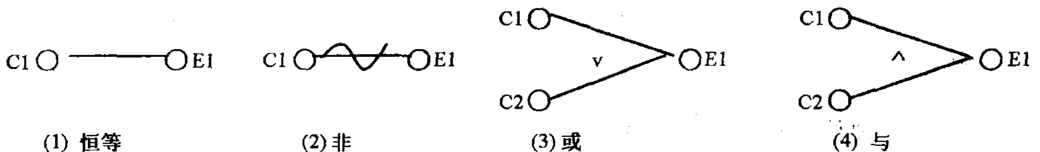


图 6.5 因果图的图形符号

通常在因果图中用 Ci 表示原因,用 Ei 表示结果。各结点表示状态,可取值“0”或“1”。“0”表示某状态不出现,“1”表示某状态出现。主要的原因和结果之间的关系有:① “逻辑恒等”关系:若 C1 出现,则 E1 出现,否则 E1 不出现。② “逻辑非”关系:若 C1 出现,则 E1 不出

现，否则E1出现。③"逻辑或"关系：若C1或C2出现，则E1出现，否则E1不出现。④"逻辑与"关系：若C1和C2同时出现，则E1出现，否则E1不出现。

为了表示原因与原因之间以及原因与结果之间可能存在的约束或限制条件，在因果图中，在基本符号表示之外，又加入了一些表示约束条件的符号。这些约束符号的表示参见图6.6。

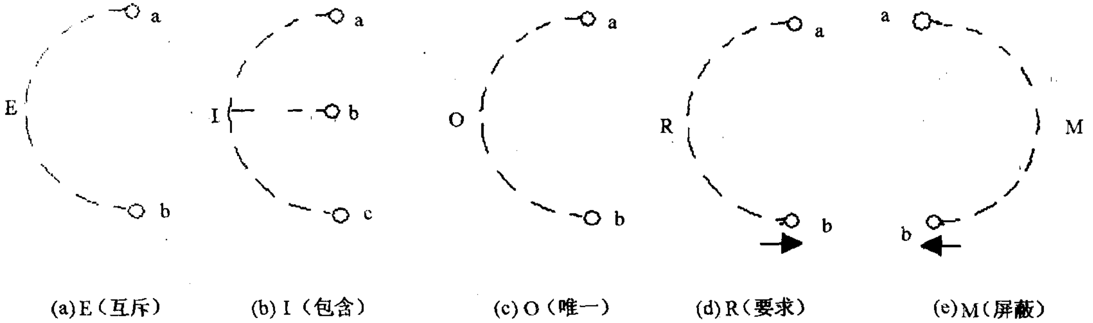


图6.6 因果图的约束符号

在图6.6中，各约束的含义如下：① E约束：表示a，b中最多有一个可能成立。② I约束：表示a，b，c中至少有一个必须成立。③ O约束：表示a和b当中必须有一个，且仅有一个成立。④ R约束：表示当a出现时，b必须也出现。⑤ M约束：表示当a是1时，b必须是0；而当a为0时，b的值不定。

6.3 软件测试步骤

由于软件错误的复杂性，在软件工程测试中我们应综合运用测试技术，并且应实施合理的测试序列：单元测试、集成测试、有效性测试和系统测试。单元测试集中于每个独立的模块；集成测试集中于模块的组装；根据软件有效性的一般定义：软件实现了用户期望的功能，有效性测试集中检验是否符合用户所见的文档，包括软件需求规格说明书，软件设计规格说明书以及用户手册等；系统测试集中检验系统所有元素（包括硬件、信息等）之间协作是否合适，整个系统的性能、功能是否达到。其中，系统测试以超出软件测试，属于计算机系统工程范畴。

下面简单介绍一下与软件测试有关的单元测试、集成测试以及有效性测试。

6.3.1 单元测试

单元测试主要检验软件设计的最小单位——模块。该测试以详细设计文档为指导，测试模块内的重要控制路径。一般来说，单元测试往往采用白盒测试技术。

在单元测试期间，通常考虑模块的以下四个特征：

(1) 模块接口；

(2) 局部数据结构；

(3) "重要的"执行路径；

(4) 错误执行路径

以及与以上四个特性相关的边界条件。

单元测试首先测试穿过模块接口的数据流，为此应当测试：输入实际参数的数目是否等于形式参数的数目、实际参数的属性与形式参数的属性是否匹配、实际参数的单位与形式参数的单位是否一致、传送给被调用模块的形式参数数目是否等于实际参数的数目、传送给被调用模块的形式参数属性是否与实际参数属性匹配，传送给被调用模块的形式参数单位是否与实际参数单位一致、对实际参数的任何访问是否与当前的入口无关、跨模块的全程变量定义是否相容等。如果该模块是实现外部 I/O 的模块，还必须测试：文件属性是否正确；I/O 语句与格式说明是否匹配；记录长度与缓冲区大小是否匹配，是否处理了文件结束条件；是否处理了 I/O 错误等。

继之，进行数据结构的测试。为此，要设计相应的测试用例，以发现下列类型的错误：不正确的或不相容的说明、置初值的错误或错误的缺省值、错误的变量名、不相容的数据类型、下溢与上溢错误等。除了局部数据结构外，还应确定全程数据对模块的影响。

第三，还要进行执行路径的选择测试。为此，也要设计相应的测试用例，以发现由于不正确的计算、错误的判定或错误的控制流而引发的错误。常见的错误有：算术运算优先级错误、置初值错误、表达式符号表示错误、计算精度错误、不同的数据类型进行比较、循环终止错误(包括循环出口错误)、不正确地修改循环变量等。

边界测试是单元测试中的最后工作，往往也是最重要的工作，因为软件常常在边界上出现错误。

在单元测试中，由于模块不是一个独立的程序，必须为每个模块单元测试开发驱动模块和(或)承接模块。驱动模块模拟“主程序”，接受测试用例的数据，将这些数据传送给要测试的模块，并打印有关的结果。承接模块代替被测模块的下属模块，打印入口检查信息，并将控制返回到它的上级模块。

驱动模块和承接模块作为单元测试的测试设备，需要花费一定的开销进行编制。

当被测模块的设计是高内聚的或是一个功能性模块时，单元测试就比较简单，只要设计少量的测试用例，便会容易地预计结果和发现错误。

6.3.2 集成测试

每个模块完成了单元测试，把它们组装在一起并不一定能够正确地工作，其原因是模块的组装存在一个接口问题。具体表现在；数据通过接口时可能予以丢失；一个模块可能对另一模块产生“副作用”；子模块的组合可能不实现所要求的主功能；模块与模块之间的误差积累可能产生不可接受的程度等等。

集成测试是软件组装的一个系统化技术，其目标是发现与接口有关的错误，将经过单元测试的模块构成一个满足设计要求的软件结构。

集成测试可“自顶向下”地进行，称为自顶向下的集成测试；也可以“自底向上”地进行，称为自底向上的集成测试。

自顶向下的集成测试是一种递增地组装软件的方法。从主控模块(主程序)开始，沿控制层次向下，或先深度或先宽度地将模块逐一组合起来，形成与设计相符的软件结构。

对于先深度的集成测试，依据应用的专业特性，选取结构的一条主线(路径)，将相关的所有模块组合起来。见图 6.7 所示例子。

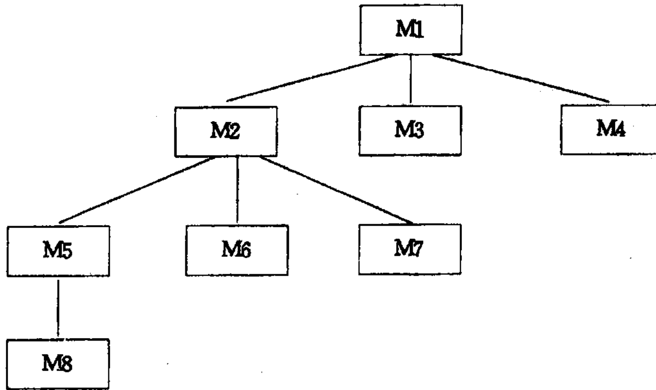


图 6.7　自底向上的集成测试

可以选择左边的路径作为主线，首先组合模块 M1，M2，M5 和 M8，然后组合 M6（如果 M2 的某个功能需要）M6，继之再构造中间和右边的控制路径。

对于先宽度的集成测试，逐层组合直接的下属模块。例如，首先组合模块 M2，M3 和 M4，继之组合 M5，M6 和 M7，……。

一般来说，集成测试是以主控模块作为测试驱动模块，设计承接模块替代其直接的下属模块，依据所选取的测试方式（先深度或先宽度），在组合模块时进行测试。每当组合一个模块时，要进行回归测试，即对以前的组合进行测试，以保证不引入新的错误。

自底向上的集成测试从软件结构最低的一层开始，逐层向上地组合模块并测试。由于在给定层次上所需要的下属模块其处理功能是可以使用的，因此无需设计承接模块。

一般来说，自底向上的集成测试首先将低层模块分类为实现某种特定功能的模块组；继之，书写一个驱动模块，用以协调测试用例的输入和输出，测试每一模块组；沿着软件结构向上，逐一去掉驱动模块，将模块组合起来。这一过程如图 6.8 所示。

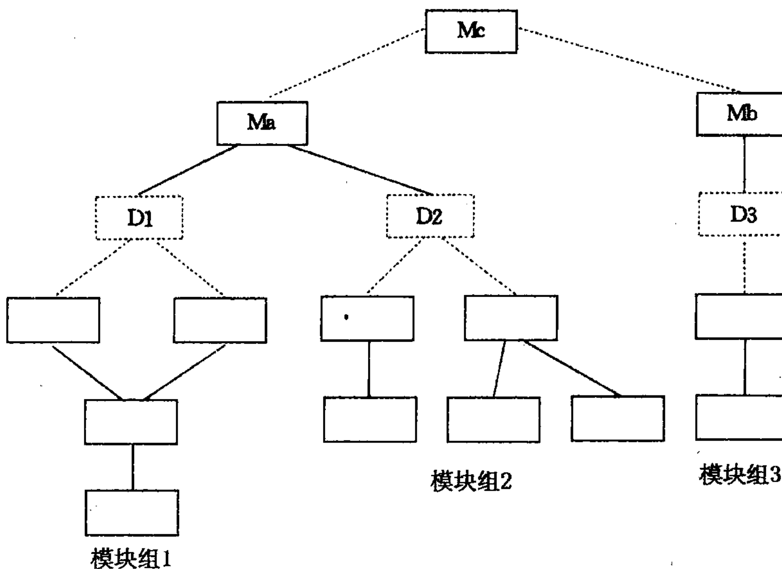


图 6.8　自底向上的集成测试

其中，有三个模块组，为每一模块组设计一个驱动模块(虚线框)，并进行测试；去掉驱动模块 D1 和 D2，将这两个模块组直接与模块 Ma 接口，类似地，去掉 D3，将另一模块组直接与模块 Mb 接口。

自顶向下和自底向上的集成测试各有其优缺点。自顶向下的主要缺点是需要设计承接模块以及随之而来的测试困难。自底向上的主要缺点是"只有在加上最后一个模块时，程序才作为一个实体而存在"。在实际的集成测试中，应根据被测软件的特性以及工程进度，选取集成测试方法。一般来说，综合地运用这两种方法，即在软件的较高层使用自顶向下的方法，而在低层使用自底向上的方法，可能是一种最好的选择方案。

6.3.3 有效性测试

有效性测试的目标是发现软件实现的功能与需求规格说明书不一致的错误。因此，有效性测试通常采用黑盒测试技术。为了实现有效性测试，应制定测试计划，该计划根据采用的测试技术，给出要进行的一组测试。继之进行测试用例和预期结果的设计。通常，在测试执行之前，应进行配置复审，其目的是保证软件配置的所有元素已被正确地开发并编排目录，具有必要的细节以支持软件生存周期中的维护阶段(如图 6.9 所示)。

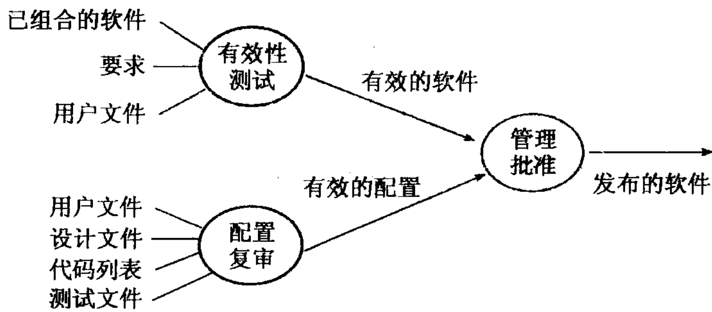


图 6.9 配置复审

在测试计划完成之前，有效性测试发现的偏差或错误一般不予纠正，通常需要和开发者一起协商，以建立解决这些缺陷的方法。

6.3.4 软件测试与程序正确性证明

通过以上内容的介绍，我们可以看到，测试可以发现错误。但是，测试并不能表明程序的正确性。如果能够开发出可靠的程序正确性证明，那么测试的工作将大幅度地减少，软件危机的一个重要方面——软件质量不可靠问题，将不复存在。

程序正确性证明涉及许多复杂的领域，利用数学归纳法或谓词演算，进行人工的正确性证明，对于小程序可能有些价值(参见附录)，但对于大一点的软件系统，这种证明几乎没有什么用处。正如 Aderson 所说："我们清楚地认识到，人工的正确性证明可能很容易包含错误，它不是避免或发现程序所有错误的灵丹妙药。"

近 20 年来，已经开发了一些自动的计算机软件正确性证明方法。利用人工智能和谓词演算为基础的自动化证明技术，针对某些特定的语言(例如 PASCAL，LISP 等)已开发了正确性

证明程序,但是把它们实际运用于大型软件,还要进行大量的工作,需要一个相当长的时间。

综上所述,软件测试在软件开发的整个技术工作中占据了很大的比重。随着信息化需求,人们越来越认识到软件测试的含义。

软件测试的目标是揭示错误。为实现这一目标,要实施一系列的测试,包括单元测试、集成测试、有效性测试和系统测试。单元测试集中于单个模块的功能和结构检验,集成测试集中于模块组合的功能和软件结构检验。有效性测试论证软件需求的可追溯性,系统测试验证将软件融于更大系统中整个系统的有效性。

每种测试将涉及一系列系统的测试技术,以支持测试用例设计和测试执行。目前,在软件开发中,通常采用人工测试技术,但随着自动化技术的研究,日益增多的自动化工具将显示出广阔的应用前景。

6.4 程序证明技术

专业人员所编制的程序,一般来说每千行均包含10个以内独立可改正的错误。因此,Hoare在第十八届国际软件工程会议(1996)上指出,"不借助程序证明,软件可靠性何以保证。"20多年来,人们一直重视程序正确性证明技术的研究,并取得了一定的成果。

分析近年来软件工程的实践,我们可以发现,为了提高软件可靠性,采用了与其他现代工程大致相同的技术,包括:

- 对设计工作进行有计划的、严格的复审;
- 不断提高机器与环境性能,支持软件设计自动化;
- 为保证软件质量,进行大规模、有目标的测试;
- 适应已广泛使用的软件产品,并进行演化;

……

应当承认,现代软件工程的进步,归功于在理论、概念和方法等诸多方面的研究成果。其中,形式化技术和证明技术,也如同其他现代工程学科一样,担当了重要的角色,起到了十分重要的作用。

今天,关于程序正确性问题,反映得就不像20年前人们想像的那么严重。特别是Mackenzie最近研究成果表明,在几千种属于计算机系统可靠性的致命危害中,仅有10个左右的问题可归结为软件中的错误,而大多数错误源于不正确的计算方法。

如同一些安全性要求很强的工程领域,软件领域的技术成果转化为生产力的速度是很慢的,至今程序证明技术还没有很好地应用于软件生产的实践,可望在不久的将来,会呈现出一种新的状况。为此,本节简单介绍一下程序证明技术——程序的公理化证明技术。

程序的一个十分重要的性质是,它是否实现了预期的功能。为了验证这一性质,可以为程序执行之后变量所能取得的值构造一个断言,以此来表示程序或部分程序是否实现了它的预期功能。通常,这一断言要指出其中每一变量值的一般性质以及值与值之间的关系。断言可以用数理逻辑的符号来表示,其中可以使用易读且熟悉的操作优先规则。

在很多情况下,程序结果的正确性依赖于该程序启动之前变量所取的值。这些前提也可以用断言来表达。

为了指出前提(P)、程序(Q)和程序执行结果(R)这三者之间的联系,我们引入一种新的符

号。

$$P\{Q\}R$$

这个符号可理解为："如果程序 Q 在启动前其断言 P 为真，那末在 Q 终止时其断言 R 要为真。"如果没有什么强加的前提，那末就可以把上式写为：

$$\text{true}\{Q\}R$$

1. Hoare 系统

(1) 赋值公理

赋值是计算机的一个最明显的特性。考虑如下赋值语句：

$$x := f$$

其中，x 是一个简单变量的标识符；f 是一个没有副作用的程序设计语言的表达式，但 f 中可能含有 x。

显然，赋值之后对 x 的值为真的任一断言 P{x}，也一定使赋值之前所采用的表达式 f(其中，f 中的 x 具有原来的值)的值为真。这样，如果在赋值之后 P{x} 为真，那末在赋值之前 P{f} 一定为真。这个事实可以更形式地表达为：

D_0 赋值公理

$$\vdash \{x := f\} P$$

其中，x 是一个变量标识符；f 是一个表达式；是通过以 f 替代 P 中所有出现的 x 而得到的。

D_0 实际上不是一个公理，而是一个公理模式。该模式描述了一个无限的公理集合，其中的公理具有同一的样式。在程序证明时，我们可以把它看作是一个公理。

(2) 推论规则

除了公理之外，作为一个演绎程序至少还需要一条推导规则，该规则容许我们从一个或多个公理或已被证明的定理中演绎新的定理。一个推导规则采用形式："**if** x **and** y **then** z"，即如果形式 x 和 y 的断言已作为定理予以证明了，那么 z 也被证明，z 即可作为一个定理。一个推导规则的最简单例子指出：如果程序 Q 的执行保证了断言 R 的真值，那么它也保证了被 R 逻辑蕴涵的每一个断言的真值；另外还有，如果对于产生结果 R 的程序 Q 而言，已知 P 是其前提，那么逻辑蕴涵 P 的另一其他断言也必是 Q 的前提。以上两个推导规则可以更形式地表示为：

D_1 推论规则

$$\textbf{if} \vdash P\{Q\}R \textbf{ and } \vdash R \supset S \textbf{ then } \vdash P\{Q\}S$$
$$\textbf{if} \vdash P\{Q\}R \textbf{ and } \vdash S \supset P \textbf{ then } \vdash S\{Q\}R$$

(3) 结构规则

一个程序一般是由一组顺序的语句组成的，这些语句一个跟着一个地执行，并以分号";"或其他等价的符号把这些语句分开：

$$(Q_1; Q_2; \cdots\cdots; Q_n)$$

为了避免冗长，可以仅讨论两个语句(;)，因为多个语句可构造为如下形式：

$$(Q_1; (Q_2; (\cdots(Q_{n-1}, Q_n)\cdots)))$$

与结构相联系的推导规则指出：如果程序的第一部分的证明结果与程序第二部分的前提是相等的，并在该前提下，程序产生了它的预期结果，那么，只要程序的第一部分的前提被满足，则整个程序将产生其预期的结果。该规则可形式地表示为：

D_2 结构规则

$$\textbf{if}\ \vdash P\{D_1\}\ \textbf{and}\ \{Q_2\}R\ \textbf{then}\ \vdash P\ \{Q_1;Q_2\}R$$

(4) 迭代规则

计算机的一个主要特性是它能够重复地执行程序 S 的某一段,直到条件 B 为假。表达这一迭代的简单方法是采用 ALGOL 60 中的 **while** 符号:

while B **do** S

根据上述语句的语义,迭代推导规则公式化的推理如下:假定 P 是一个断言,只要它在 S 初始时为真,它在 S 完成时总是真的。显然在 S 语句的任一次迭代之后(甚至 S 不重复执行) p 仍然一直为真。另外,在迭代最终结束时,条件 B 为假这是已知的。依据 B 在 S 初始时可以假定为真这一事实,迭代规则可以形式地表示为:

D_3 迭代规则

$$\textbf{if}\ \vdash p\wedge B\ \{S\}\ p\ \textbf{then}\ \vdash p\ \{\textbf{while}\ B\ \textbf{do}\ S\}\ \neg\ B\wedge p$$

以上的公理和规则作了以下的限制:

① 首先,我们假定了所引用的推导规则和公理没有求值表达式和条件的副作用。在以一种容许副作用的语言所表达的程序的性质证明中,在应用合理的证明技术之前,必须证明在每一种情况下均没有副作用。

② 其次,在以上引用的公理和规则中,都没有提供证明程序成功终止的基础。就程序的终止失败而言,其原因可能是由于死循环或由于违背了实现时所固定的限制。例如数的区域、内存的大小或操作系统的时间限制等。如此,符号"P{Q}R"应解释为:"只要程序成功地终止,那么 R 描述了程序结果的性质。"修改公理以使它们不能用于预言非终止程序的结果,这是相当容易的。但是这些公理的使用现在要依赖对许多实现相关的性质的了解,例如,计算机的大小和速度、数的区域以及溢出技术的选择。

③ 最后,还有一些不予讨论的范围。例如:实数、位及字符、复数、分数、数组、记录、重定义、文件、输入/输出、说明、子程序、参数、递归以及并行执行。甚至整数算术的特性也远不是完整的。只要程序语言是简单的,那么对这些范围的处理好像没有什么太大的困难。真正困难的问题是标号以及转移、指针、名字参数。对于使用这些性质的程序,其证明要精心编制;并且暗含着构成公理基础的复杂性。

综上所述,关于顺序程序性质证明的 Hoare 系统可归纳如下:

令 P 和 Q 是关于变量的断言,并且 S 是一个语句。非形式地,符号

$$\{P\}S\{Q\}$$

的意义为:如果 P 在 S 执行之前为真,那么 Q 在 S 执行之后为真。并没有提及 S 的终止问题。只要 S 终止,则 Q 成立。

符号

$$\frac{a}{b}$$

的意义为:如果 a 为真,则 b 也为真。为了证明顺序程序的性质,Hoare 就是使用这样的符号描述了一个推演系统。

令 P, P_i 表示断言,x 表示变量,E 表示表达式,B 表示布尔表达式;并令 S, S_i 表示语句。那么对于五类所容许的语句,其公理是:

① $\{P\}skip\{P\}$ 空

② $\{P_E^x\}x:=E\{P\}$ 赋值

其中 P_E^x 表示以 E 代替 P 中每一自由出现的 x 之后所产生的断言。

③ $\dfrac{\{P \wedge B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\text{if S then } S_1 \text{ else } S_2(Q)}$ 选择

④ $\dfrac{\{P \wedge B\}S\{P\}}{\{P\}\text{while B do S}\{P \wedge \neg B\}}$ 循环

⑤ $\dfrac{\{P_1\}S_1\{P_2\}, \{P_2\}S_2\{P_3\}, \cdots, \{P_n\}S_n\{P_{n+1}\}}{\{P_1\}\text{begin } S_1;S_2;\cdots;S_n \text{ end }\{P_{n+1}\}}$ 复合

除此之外，我们还有如下的推理规则：

⑥ $\dfrac{\{P_1\}S\{Q_1\}, \ P \vdash P_1 Q_1 \vdash Q}{\{P_1\}S\{Q\}}$ 推理

其中符号 $P \vdash Q$ 意味着可以使用 P 作为一个假设来证明 Q。但没有给出根据 P 来证明 Q 的推演系统，它可以是对于程序设计语言中所使用的数据类型以及操作均有效的系统。

现在，让我们简单地讨论一下顺序程序性质的证明。当我们写出 $\{P\}S\{Q\}$ 时，则意味着通过使用①～⑥，存在着 $\{P\}S\{Q\}$ 的一个证明。例如，假设语句 S

begin x:= a; **if** e **then** S1 **else** S2 **end**

并假设我们已经有了证明

$$\{P_1 \wedge e\}S_1\{Q_1\} \text{和} \{P_1 \wedge \neg e\}S_2\{Q_1\}$$

那么，$\{P\}S\{Q\}$ 的一个证明可以是：

① $\{P1_a^x\}x:=a\{P1\}$ 赋值

② $\dfrac{\{P1_a^x\}x:=a\{P1\}, \ P \vdash P1_a^x}{\{P\}x:=a\{P1\}}$ 推理规则

③ $\dfrac{\{P1 \wedge e\}S1\{Q1\}, \{P1 \wedge \neg e\}S2\{Q1\}}{\{P1\}\text{if e then S1 else S2}\{Q1\}}$ 选择

④ $\dfrac{\{P1\}\text{if e then S1 else S2 }\{Q1\}, Q1 \vdash Q}{\{P1\}\text{if e then S1 else S2}\{Q\}}$ 推理规则

⑤ $\dfrac{\{P\}x:=a\{P1\}, \{P1\}\text{if e then S1 else S2}\{Q\}}{\{P\}\text{ begin x := a; if e then S1 else S2 end }\{Q\}}$ 复合

下面给出这个证明的另一种证明形式。在这一证明形式中，由于程序伴随着适当位置上的断言给出，因此其证明就容易理解得多。在这样的证明形式中，两个相邻的断言 $\{P1\}\{P2\}$ 表示推理规则的一种用法，在推理中，它们被写为 $P1 \vdash P2$。

```
⊢{P}
begin {P}
      {P1ˣₐ}
      x:= a;
      {P1}
      if e then {P1 ∧ e}
                S1
                {Q1}
         else {P1 ∧ ¬ e}
```

S2

{Q1}

{Q1}

{Q}

end

{Q}

以上给出的证明形式及程序执行时性质的证明依赖于以下性质：

令 S 是程序 T 中的一个语句，并且 pre(S)是证明形式{P}T{Q}中 S 的前置条件。假设 T 的执行以 P 为真开始并达到即将执行 S 的那一点，此时，所有变量处于状态 m 之中，那么 pre(S)[m] = true。

下面，通过一个简单的程序，即计算 x 除以 y 求其商 q 与余数 r 的程序，以 Hoare 系统给出该程序的一个证明。为了简单，在此使用了最普通的连减算法，来求解这一问题。显然，该程序如下所示：

$$((r := x; q := 0); \text{While } y \leqslant r \text{ do } (r := r - y;\ q := 1 + q))$$

由于在该程序中，使用了一些变量和操作，因此，为了证明，必须限定这些变量的取值，还要假定这些操作所具有的基本性质，并作为公理予以使用。在此，我们假定所有变量的取值均为非负整数；并且，关于整数加法、减法和乘法，选取如下一些基本性质：

A1	$x + y = y + x$	加法交换律
A2	$x \times y = y \times x$	乘法交换律
A3	$(x + y) + z = x + (y + z)$	加法结合律
A4	$(x \times y) \times z = x \times (y \times z)$	乘法结合律
A5	$x \times (y + z) = x \times y + x \times z$	乘法分配律
A6	$y \leqslant x \quad (x - y) + y = x$	加减相消
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	

通过以上选取的性质，可以演绎在程序证明中所需要的一些简单定理：

① $x = x + y \times 0$ (引理 1)

证明：A7 $x = x + 0$

A8 $x = x + y \times 0$

② $y \leqslant r \quad r + y \times q = (r - y) + y \times (1 + q)$ (引理 2)

证明：A5 $(r - y) + y \times (1 + q) = (r - y) + (y \times 1) + (y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ （只要 $y \leqslant r$）

该程序的重要性质是，当其终止时：

(i) 可以把商 q 乘以除数 y，再加上余数 r 就可以得到除数 x，即 $x = r + q \times y$；

(ii) 余数 r 小于除数 y。

程序的这些性质可以形式地表示为：

$$\text{true}\{Q\}\neg\ y\leqslant r\wedge x=r+y\times q \qquad (*)$$

其中Q代表上面给出的程序。该式表达了这一程序正确性的一个必要条件(但不是充分条件)。下面给出(*)式的形式证明:

行数	证明	理由
1	$\mathbf{true}\supset x=x+y\times 0$	引理1
2	$x=x+y\times 0\{r:=x\}\ x=r+y\times 0$	赋值
3	$x=r+y\times 0\{q:=0\}\ x=r+y\times q$	赋值
4	$\text{true}\supset\{r:=x\}x=r+y\times 0$	推理(1,2)
5	$\mathbf{true}\supset\{r:=x;\ q:=0\}x=r+y\times q$	复合(4,3)
6	$x=r+y\times q\wedge y\leqslant r\supset x=(r-y)+y\times(1+q)$	引理2
7	$x=(r-y)+y\times(1+q)\{r:=r-y\}\ x=r+y\times(1+q)$	赋值
8	$x=r+y\times(1+q)\{q:=1+q\}\ x=r+y\times q$	赋值
9	$x=(r-y)+y\times(1+q)\{r:=r-y;\ q:=1+q\}\ x=r+y\times q$	复合(7,8)
10	$x=r+y\times q\wedge y\leqslant r\{r;=r-y;\ q:=1+q\}x=r+y\times q$	推理(6,9)
11	$x=r+y\times q\{\mathbf{While}\ y\leqslant r\ \mathbf{do}(r:=r-y;\ q:=1+q)\}$ $\neg\ y\leqslant r\wedge x=r+y\times q$	循环(10)
12	$\mathbf{true}((r:=x;q:=0);\ \mathbf{While}\ y\leqslant r$ $\mathbf{do}(r:=r-y;\ q:=1+q))\neg\ y\leqslant r\wedge x=r+y\times q$	复合(5,11)

应当注意,在该问题的证明中,隐含地假定了所引用的推理规则和公理没有求值表达式和条件的副作用,在引用的公理和规则中没有提供证明程序成功终止的基础,并且所涉及的对象仅为非负整数。

由以上给出的程序正确性证明的例子,可以看出:程序的正确性证明,就是验证是否达到用户的预期目的。如果这些预期目的可以通过断定予以严格描述,其中这些断定是关于在程序结束时(或在某一指定的位置)变量值的断定,那么,只要程序设计语言的实现遵循了在证明中使用的公理和规则,就可以使用这里给出的技术来证明程序的正确性。

2. 并行程序的正确性证明

为了引入并行性,我们对顺序语言予以扩充,使之具有两个新的语句。一个是为并行处理而引入的cobegin语句,另一个是为了协调并行执行的进程而引入的await语句。

令$S_1,S_2,\cdots,S_n$是一些语句,则cobegin语句:

$$\mathbf{cobegin}\ S_1\parallel S_2\parallel\cdots\parallel S_n\mathbf{coend}$$

的执行导致了语句S_i的并行执行。当所有进程S_i的执行终止时,则cobegin语句的执行才终止。对于并行执行的实现方法没有什么限制;尤其是,对于这些进程之间的相对速度更没有任何假定。

我们要求每个赋值语句的执行和每一表达式的求值要作为一个独立的、不可分割的活动。此处设程序附有如下简单的约定(本节讲述的内容遵循这一约定):

(1) 每个表达式E求值时,它至多可以引用一个可以被另一进程改变的变量y,且至多引用一次。对于赋值语句$x:=E$,具有类似的限制。

依据(1)的约定,所需要的惟一不可分活动是“内存引用”,即假定进程S_i引用变量(地址)c而另一进程S_j正改变c。我们要求S_i接受的c值,或是对c赋值之前的值,或是对c赋值之后的值,但c值不能由于赋值期间c值的“波动”而导致不真。因此,我们的并行语言就可以用于模拟在任一适当的机器上的并行执行。

我们引入的第二个语句是：

await B **then** S

其中 B 是一个布尔表达式，S 是一个不含有 cobegin 语句或含一 await 语句的语句。当一个进程企图执行 await 时，则该进程被延迟到 B 为真。因此语句 S 是作为一个不可分的活动执行的。当 S 终止时，并行处理继续。如果两个或两个以上进程正等待同一条件 B，那么当 B 变为真时，只允许它们中的一个进程优先处理，而其他进程则继续等待。在一些应用中，要求给出调度等待进程的次序；但就我们的目的而言，其调度进程的规则是任意的。注意，B 的求值是 await 语句这一不可分割活动的一组成部分；另一进程在 B 求值之后，在 S 开始执行之前均不可改变变量以使 B 成为假。

使用 await 语句可使任一语句 S 成为一个不可分割的活动：

await true **then** S

或把 await 语句纯粹用作一个同步工具：

await"某个条件"**then** skip

注意，我们宁愿把 await 语句作为表达多个标准的同步原语（例如信号量）的工具提出，而不把它作为插入到其他程序设计语言中的新的同步语句，因为要求太强，而不能有效实现。这样，为了验证使用信号量的程序，首先以 await 表达信号量操作，随之应用本节给出的技术。

现在，我们再去形式地定义(2)，(3)中的语句。await 的定义是一目了然的，但关于 cobegin 语句 D 的定义(3)需要和"无干扰"定义一起予以说明：

(2) await $\dfrac{\{P \wedge B\}S\{Q\}}{\{P\}\text{await B then S }\{Q\}}$。

(3) cobegin $\dfrac{\{P_1\}S_1\{Q_1\},\cdots,\{Pn\}Sn\{Qn\}\text{是“无干扰”}}{\{P_1 \wedge \cdots \wedge Pn\}\text{cobegin } S_1 \parallel \cdots \parallel Sn \text{ coend } \{Q_1 \wedge \cdots \wedge Qn\}}$。

定义(3)表明，只要 $S_1,\cdots,S_n$ 这些进程不相互干扰，那么并行执行它们的效果如同每个进程独自执行的效果。当然，其中关键词是"干扰"两字。获得不相互干扰的一个可能的办法是不允许它们共享变量，但这个办法太局限了。一个更有用的规则是：要求在每个进程的证明{Pi} Si {Qi}中所使用的某些断言在其他进程并行执行之后均不改变其真值。因为如果这些断言不变为假，那么证明{Pi}Si{Qi}就仍然有效，由此，终止时 Qi 的值将仍然为真！例如，断言{$x \geq y$}在执行 $x := x + 1$ 后仍为真，而断言{$x = y$}则不然。断定 P 在语句 S 执行之后的不变性由公式

$$\{P \wedge \text{pre}(S)\}S\{P\}$$

予以表达。下面，我们给出"无干扰"的定义。

(4) 定义：给定一证明{P}S{Q}以及具有前置条件 pre(T)的语句 T，如果如下两个条件成立：

① $\{Q \wedge \text{pre}(T)\}\ T\{Q\}$；

② 令 S'是 S 中的任一语句（但 S'不能在 await 中），有

$$\{\text{pre}(S') \wedge \text{pre}(T)\}T\{\text{pre}(S')\}$$

那么我们就说 T 不干扰{P}S{Q}。

(5) 定义：如果满足以下条件：

令 T 是进程 Si 的一个 await 或一个赋值语句（它不出现在一个 await 语句中），对于所有的 j($j \neq i$)，T 不干扰{Pj}Sj{Qj}，那么我们就说$\{P_1\}S_1\{Q_1\},\cdots,\{Pn\}Sn\{Qn\}$是"无干扰"的。

在出现的正确性证明中，我们需要一次又一次地实行对正确性显然没有影响的程序变换，

例如,只要赋值语句满足(1),那么就以 S 代替 begin S end, 以 x:=E 代替 await true then x:=E。在证明并行程序正确性之中,增加(或删去)对那些称为辅助变量的赋值,这样的变换是必要的。其中这些辅助变量仅在正确性证明以及证明其他辅助性质时方是需要的,而对于它们所在的程序而言并非必要。典型的辅助变量是用于记录程序执行的“历史”,或指出当前正在执行程序的哪一部分。在大多数情况下,唯有自己才知道辅助变量的意义。

(6) 定义:令 AV 是 S 中仅在赋值语句 x:=E 出现的一个变量集合,其中 x 属于 AV,那么,AV 是 S 的一个辅助变量集合。

(7) 辅助变量变换:令 AV 是 S1 的一个辅助变量集合,并且 P 和 Q 是不含有 AV 中的自由变量的断言。令 S 是把 S1 中所有对 AV 中变量赋值的语句删去后而得到的语句,则

$$\frac{\{P\}S1\{Q\}}{\{P\}S\{Q\}}$$

下面,我们将对推演系统 1.(1)~1.(6),2.(2),2.(3)以及 2.(7)的使用给出一个例子。但首先让我们讨论一下 2.(3)。规则 2.(3)告诉我们要以两步来理解并行进程。首先,在研究每一进程 Si 的证明之中,要把它们看做是独立的顺序程序,而不把它们看做是完全并行执行的程序。其次,证明每一进程 Sj 的执行不干扰 Si(i≠j)。

通常,证明不干扰的方法是看进程 Sj 的执行是否与 Si 的执行发生干扰。这样,我们可以用如下词句来表达:“假定 Sj 如此如此地执行,而 Si 执行这个又执行那个。”了解两个动态目标——Si 和 Sj 的交替执行是非常困难的,因而就更不可能了解多个并行进程的交替执行,且十分容易遗漏某处的讨论。

为了集中于 Sj 是否可能影响 Si 正确性的证明,我们来注意一个容易处理的静态目标。首先,我们编制一个表,在该表中列出 Si 的前置条件;而在另一表中列出 Sj 的赋值语句以及 await 语句,并且证明第二个表中的每一语句不干扰第一个表中的每一断言的真值,以此来证明不干扰就是十分机械的。

如果 Sj 的一个语句 T 干扰 Si 的一个前置条件 P,那么,或程序是不正确的,或 Si 的证明是不适当的。常常可以通过保持其证明仍然有效,来修改证明{Pi}Si{Qi},也就是可以弱化断言,直到 Sj 不再与它们发生干扰。在任一情况下,只要程序员始终留心,就不会轻易地遗漏一个特殊情况;这个就不是以前非形式推论那样的情况。

(8) 实例研究:我们考虑并行程序设计文献中的一个标准问题。生产者进程为消费者进程生产一串值。由于生产者和消费者着手于一个变量,而它们的速度大略相等,因此在这两个进程之间插入一个缓冲是有好处的。但内存是有限的,因而缓冲的大小设为 N。我们对这个缓冲描述如下:

① buffer[0:N-1]是共享缓冲

in = 添加到该缓冲中的元素个数

out = 自该缓冲中移走的元素个数

该缓冲含有 in - out 个值。这些值是按如下次序存放于缓冲之中:

buffer[out mod N], …, buffer[(out + in - out - 1)mod N]

在②中,我们给出了这一问题在一般环境下的解法。③是使用这一解法编制的程序,该程序把数组 A[1:M]的值复制到数组 B[1:M]中。④给出了其主程序的证明;⑤和⑥对其不同的进程给出了相应的证明。为了证明“无干扰”这一性质,首先要注意断言 I 始终是两个进程的

不变式。在消费者中，惟一可能改变生产者断言真值的赋值语句是 out:= out + 1；且 in - out<N 是消费者惟一可能被生产者改变真值的断言。但显然，语句 out:= out + 1 增加了 out 的值，使断言 in - out<N 尚为真。这样，便证明了消费者不干扰生产者；类似地可以证明生产者不干扰消费者。

```
② begin comment 缓冲描述见①;
   in :=0; out :=0;
   cobegin producer:…
     await in - out<N then skip;
     add; buffer [in mod N]:= next value;
     markin: in := in + 1;
     ‖ ….
     consumer:…
     await in - out>0 then skip;
     remove: this value := buffer [out mod N];
     markout: out := out + 1;
     …
     coend
     end
③ fgl:begin comment 缓冲的描述见①;
        in :=0; out :=0; i :=1; j:=1;
        cobegin producer: while i≤M do
        begin x := A[i];
          await in - out<N then skip;
          add: buffer [in bod N] := x;
          markin: in := in + 1;
          i := i + 1;
        end
        ‖
        consumer: while j≤M do
           begin
             await in - out>0 then skip;
             remove: y := buffer [out mod N];
             markout: out := out + 1;
             B[j] := y;
             j := j + 1;
           end
        coend
     end
```

④ fgl(主程序)的证明形式

{M≥0}

```
fgl:begin in := 0; out := 0; i := 1; j := 1;
    {I ∧ i = in + 1 = 1 ∧ j = out + 1 = 1}
    fgl' cobegin
           {I ∧ i = in + 1 = 1} producer {I ∧ i = in + 1 = M + 1}
           {I ∧ j = out + 1 = 1} consumer {I ∧ (B[k] = A[k], 1≤k≤M)}
         coend
  end
{B[k] = A[k], 1≤k≤M}
```

$$
\text{其中 } I=\begin{cases}\text{buffer}[(k-1)\ \mathbf{mod}\ N]=A[k],\ \text{out}<k\leqslant \text{in}\\ \qquad\wedge\ 0\leqslant \text{in}-\text{out}\leqslant N\\ \qquad\wedge\ 1\leqslant i\leqslant M+1\\ \qquad\wedge\ 1\leqslant j\leqslant M+1\end{cases}
$$

⑤ fgl(生产者)的证明形式。其中的不变式 I 如同④中的 I

```
{I ∧ i = in + 1}
producer: while i≤M do
    begin
        {I ∧ i = in + 1 ∧ i≤M} x := A[i]; {I ∧ i = in + 1 ∧ i≤M ∧ x = A[i]}
        await in - out < N then skip;
        {I ∧ i = in + 1 ∧ i≤M ∧ x = A[i] ∧ in - out < N}
        add: buffer[in mod N] = x;
        {I ∧ i = in + 1 ∧ i≤M ∧ buffer[in mod N] = A[i] ∧ in - out < N}
        markin: in := in + 1;
        {I ∧ i = in ∧ i≤M} i := i + 1 {I ∧ i = in + 1}
    end
    {I ∧ i = in + 1 = M + 1}
```

⑥ fgl(消费者)的证明形式。其中不变式 I 如同④中的 I。

```
{I ∧ IC ∧ j = out + 1}
consumer: while j≤M do
    begin {I ∧ IC ∧ j = out + 1 ∧ j≤M}
      await in - out > 0 then skip;
      {I ∧ IC ∧ j = out + 1 ∧ j≤M ∧ in - out > 0}
      remove: y := buffer [out mod N];
      {I ∧ IC ∧ j = out + 1 ∧ j≤M ∧ in - out > 0 ∧ y = A[j]}
      markout: out := out + 1;
      {I ∧ IC ∧ j = out ∧ j≤M ∧ y = A[j]}
      B[j] = y;
      {I ∧ IC ∧ j = out ∧ j≤M ∧ B[j] = A[j]}
      j := j + 1;
      {I ∧ IC ∧ j = out + 1 ∧ j≤M + 1}
    end
```

$\{I \wedge IC \wedge j = out + 1 = M + 1\}$

$\{I \wedge (B[k] = A[k], 1 \leqslant k \leqslant M)\}$

其中 $IC = \{B[k] = A[k], 1 \leqslant k < j\}$

关于并行程序的正确性证明，由于引入了 await 语句，因此还应讨论封锁与死锁，即要证明程序的终止等问题[9]。

习 题 六

1．解释以下术语：

软件测试　　测试用例　　测试覆盖率

2．简述测试过程模型，并分析这一模型在软件测试技术研究以及实践中的作用。

3．简要回答以下问题：

(1) 软件测试与调试之间的区别；

(2) 程序控制流程图的作用以及构成；

(3) 语句覆盖、分支覆盖、条件组合覆盖、路径覆盖之间的关系；

(4) 单元测试、集成测试、有效性测试之间的区别；

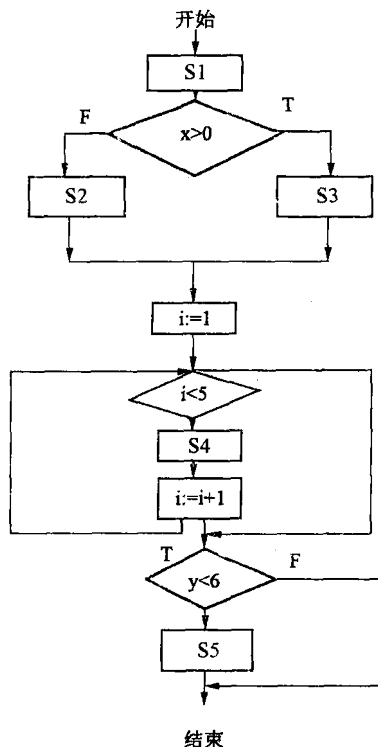
(5) 路径测试技术、事务流测试技术的主要依据；

(6) 针对程序控制流程图中出现的各种不同循环，如何选取测试路径；

(7) 程序控制流程图与事务流程图之间的主要区别，并分析出现这些区别的原因；

(8) 测试执行的基本条件。

4．根据下面给出的控制流程图(为了清晰，采用了人们"习惯"的表示法)，设计最少的测试用例，实现分支覆盖。(注：在设计测试用例时，其中的循环结构可以看作是一个过程块)



5. 针对以下的程序伪码，建立该程序的测试模型(即被测对象模型)，并设计实现分支覆盖所需要的测试用例(表达用例的方法是任意的)。

```
BEGIN
    输入一元二次方程的系数 A, B, C;
    为根变量赋初值;
    IF 平方项的系数 A=0 且一次项系数 B< >0
      THEN BEGIN root1: = -C/B;输出"A=0";
                 root2: = -C/B
           END;
    IF 平方项的系数 A< >0 且一次项系数 B=0
      THEN BEGIN
                 IF (-C/A)> =0
                   THEN BEGIN root1: = SQR(-C/A);输出"B=0";
                              root2: = -SQR(-C/A)
                        END
           END;
    IF 平方项的系数 A< >0 且一次项系数 B< >0
      THEN
           IF (B^2-4AC)> =0
             THEN BEGIN root1: = (-B+SQR(B^2-4AC))/2A;
                        root2: = (-B-SQR(B^2-4AC))/2A
                  END
             ELSE 输出"此方程无实根";
    输出 root1 和 root2 的值
END.
```

6. 针对以下给出的问题：某一 8 位计算机，其十六进制常数的定义为：以 0x 或 0X，开头的数是十六进制整数，其值的范围是 -7f 至 7f(大小写字母不加区别)，如 0x13，0X6A，-0x3c。

请用等价类划分方法，设计测试用例。

7. 简述 Hoare 系统的构成，并扼要说明在程序正确性证明中的使用。

第七章　软件过程与改善

7.1 软件过程

在“软件生存周期”概念提出以前，一提及软件开发，很容易理解为就是“编程序”。随着计算机应用范围不断扩大和深化，软件越来越复杂。软件开发实践使人们认识到，软件系统开发与其他工业产品生产一样，也有必不可少的设计、制作、检验等阶段。自 1976 年之后，提出了“软件生存周期”这一概念，开始注重编程之前的工作。

软件生存周期是软件产品或系统的一系列相关活动的全周期。从形成概念开始，经过开发、交付使用、在使用中不断修订和演化，直到最后被淘汰，让位于新的软件产品。根据这一概念，在软件开发中人们提出了一些软件生存周期模型，例如瀑布模型，用以划分若干个相互区别又彼此联系的阶段(活动)，组织、指导软件开发。

经过一段时间的实践，人们发现像瀑布模型那样划分阶段有许多缺点，随之又相继提出了演化模型、螺旋模型以及喷泉模型等。接着又发现提高软件生产率和软件质量的困难之处，在于开发和维护中的支持和管理问题。于是自 80 年代初，对软件过程进行了一系列的研究和讨论。在此基础上，IEEE 标准化委员会于 1991 年 9 月制定出“软件生存周期过程开发标准”，接着 ISO/IEC 于 1994 年制定出“软件生存周期过程”标准。

软件过程是软件生存周期中的一系列相关过程，又称为软件生存周期过程。过程是活动的集合，活动是任务的集合，任务是将输入变换为输出的操作。活动的执行可以是顺序的，可以是重复的，可以是并行的，也可以是嵌套的。

软件过程的研究主要针对软件生产和管理。为了获得满足工程目标的软件产品，不仅涉及工程开发，还要涉及工程支持和工程管理。也就是说，软件过程不仅要有工程观点，还要有系统观点、管理观点、运行观点和用户观点。

软件过程有多种分类方法。按照不同人员的工作内容来分，软件过程可分为三类：基本过程、支持过程和组织过程、并且，当把软件过程运用到相关组织，运用到具体应用领域或具体项目时，可以根据特定情况，对各种过程和活动进行剪裁，形成所需要的软件过程模型，这就是剪裁过程。

7.1.1 基本过程

基本过程是指那些与软件生产直接相关的过程，包括获取过程、供应过程、开发过程、运行过程和维护过程。

1. 获取过程

获取过程是获取者为了得到一个软件系统或软件产品所进行的一系列活动。它从确定获取该系统或软件产品的需求定义开始，经过招标准备，合同的准备和修改，对供应方的监督，直到验收完成方告结束。

2. 供应过程

供应过程是供应者为获取者提供软件产品的一系列活动。它从理解系统或软件产品的需求开始，经过投标准备，签订合同，制定计划，实施和控制，评审和评价等活动，直到交付完成。

3. 开发过程

开发过程是软件开发者所从事的一系列活动，其目的是依据合同成功地开发并交付软件。当要开发新的系统，或对已有的系统进行版本升级，以及对已有系统进行有开发活动的移植时，都要涉及开发过程。开发过程包括的活动有需求分析、设计、编码、集成、测试、安装以及验收支持等。在开发过程中，还贯穿了其他软件过程的实施；通过管理过程管理开发过程，通过剪裁过程剪裁开发过程的活动，通过改进过程参与开发过程的管理，通过基础设施过程建立基础设施，通过支持过程进行文档编制、联合评审和项目审计等。

具体地说，开发过程所涉及的活动有：

(1) 过程的实施准备

这一活动的目的是为开发过程准备基本的约定。其主要任务有：

① 依据合同及软件或系统的特点，选择以下(2)到(13)的活动，所选择的活动可重复或相关联，亦可循环交替；

② 制定本过程计划，计划中至少应包含所需的内部标准、方法、工具、行为、责任以及所使用的程序设计语言等；

③ 指定各种文档的编制方式，安排其他各支持过程的实施方法(参见支持过程)。

(2) 系统需求分析

这一活动的目的是根据合同，分析系统的具体应用意图，完成系统需求规格说明书。系统需求规格说明书的内容有：

① 对系统功能和性能的需求，包括安全、保密性；

② 人机界面、操作和维护需求；

③ 设计方面的限制和对合格的需求等。

检查该系统需求规格说明书：是否能跟踪获取过程得到的系统需求并与之保持一致(参见获取过程)；是否具备可测试性；是否具备系统结构设计的条件；操作与维护是否可行等。

(3) 系统结构设计

这一活动的目的是建立一个高层的系统体系结构。该体系结构将系统需求按硬件、软件、人工操作这三个要素分为硬件配置项、软件配置项和人工操作项。

检查该结构是否达到以下要求：是否能跟踪系统需求并保持一致；所使用的方法是否符合标准；所确定的软件配置项需求是否可行；操作和维护是否可行等。

(4) 软件需求分析

这一活动的目的是确定软件需求及质量特性需求，并完成软件需求规格说明书。软件需求规格说明书的内容包括：

① 功能和性能需求；

② 外界与软件(即软件配置项)的接口；

③ 合格需求；

④ 安全需求，包括与操作、维护、环境等对人员的伤害有关的说明；

⑤ 保密需求；

⑥ 人机工程和人机界面需求；

⑦ 数据定义和数据库需求；

⑧ 现场安装及验收需求；

⑨ 用户文档；

⑩ 用户操作和运行需求；

⑪ 用户维护需求；

等等。

检查该软件需求：是否能跟踪系统需求和系统结构；是否外部与系统需求保持一致；软件需求内部是否具备一致性；测试覆盖程度是否达到要求；是否具备可测试性；操作和维护的可行性如何；等等。

(5) 软件体系结构设计

这一活动的目的是将软件需求规格说明书转化为一个软件体系结构。其主要任务有：

① 定义并描述该结构的顶层部分；

② 为软件外部接口、数据库、软件各部件之间的接口作顶层设计；

③ 设计用户手册的初级版本；

④ 定义初步测试需求并制定软件集成计划；

等等。

检查该结构是否能跟踪软件需求，并在外部与其保持一致；结构内各部件是否具备一致性；所用设计方法与标准是否一致；是否具备进一步详细设计的条件；操作与维护是否可行；等等。

(6) 软件详细设计

这一活动的目的是详细设计软件体系结构中的每个部件。主要任务有：

① 尽可能地将每个部件细划为可进行编码、编译及测试的软件单元；

② 详细设计软件的外部接口、软件各部件之间的接口、各单元之间的接口；

③ 详细设计数据库；

④ 充实用户手册；

⑤ 定义单元测试需求并制定单元测试计划；

⑥ 充实集成测试需求并制定集成测试计划；

等等。

检查该详细设计是否能跟踪软件需求；是否在外部与体系结构具备一致性；各部件以及各单元之间是否具备一致性；所使用的设计方法是否符合标准；是否可测试；是否具备可操作性和可维护性等。

(7) 软件编码和测试

这一活动的主要任务有：

① 根据详细设计结果进行编码，提供测试数据，完成单元测试；

② 充实软件集成测试需求和集成计划；

③ 充实用户手册等；

等等。

检查本活动是否能跟踪软件需求和软件设计，并在外部与其保持一致；在软件部件内部，各单元需求之间是否具备一致性；单元的测试覆盖程度是否达到要求；编码方法是否符合标

准;是否具备软件集成及测试的条件;是否具备可操作性和可维护性;等等。

(8) 软件集成

这一活动的目的是制定计划,并将上述活动得到的各软件单元、软件部件集成为所需软件。主要任务有:

① 制定计划,计划中至少包括测试需求、步骤、数据、责任和时间进度表等内容;

② 按计划进行软件的集成;

③ 充实用户手册;

④ 针对合格需求制定合格测试集及步骤:

检查本活动以及集成好的软件是否能跟踪系统需求,并在外部是否与之保持一致性;内部是否具备一致性;测试所用的方法是否符合标准;是否符合预期结果;是否具备软件合格测试的条件;是否具备可操作性和可维护性;等等。

(9) 软件合格测试

这一活动的目的是完成软件的合格测试。依据合格要求进行合格测试,并充实用户手册。

检查本活动软件需求的测试覆盖度是否达到要求;是否符合预期结果;系统集成和测试是否可行;是否具备可操作性和可维护性等。若可能的话,应支持审计工作,审计后应准备一套完整的软件,交付给活动(10)至活动(13)。

(10) 系统集成

这一活动的目的是将交付的软件集成到系统中。主要任务有:

① 将软件同有关硬件、人工操作项以及其他必要系统集成为一个系统;

② 为系统合格测试进行必要的准备,包括开发测试集和测试步骤。

检查集成完的系统:系统需求的测试覆盖程度是否达要求;所用测试方法是否符合标准;是否符合预期结果;是否具备系统合格测试的条件;是否具备可操作性和可维护性;等等。

(11) 系统合格测试

这一活动的主要任务即依据合格要求和合格测试计划进行系统合格测试。

检查本活动:系统需求的测试覆盖是否达到要求;是否符合预期结果;是否具备可操作性和可维护性;等等。若可能的话,应支持审计工作,并在审计结束后准备一套完整的软件以便进行软件安装或软件验收支持。

(12) 软件安装

这一活动的目的是在目标环境中安装软件。主要任务:

① 制定安装计划,包括与软件安装有关的信息和资源;

② 实施软件安装,注意保证该软件和数据库能按合同的规定初始化、运行和终止。

(13) 软件验收支持

这一活动的目的是支持获取者对软件的验收评审和测试。主要任务有:

① 支持验收评审和测试;

② 按合同交付文档及代码;

③ 依据合同向获取者提供培训和支持。

4. 运行过程

运行过程是用户和操作人员在用户的业务运行环境中为了使系统或软件产品投入运行所进行的一系列有关的活动。此过程的目的是在软件开发过程完成后,将该系统从开发的环境

转移到用户的业务运行环境中运行;在运行时对用户的要求提供帮助和咨询;并对运行效果作出评价。此过程可为开发过程和维护过程提供有关的反馈信息,作为改进系统的依据。运行管理者可根据软件项目的总要求按照软件管理过程(参见“软件管理过程”)的内容对运行过程进行管理。

运行过程一般包括以下 7 个活动:实施过程的准备;运行测试;系统到业务运行环境的转移;系统运行;对用户运行的支持;系统运行评价;用户运行评价等。

(1) 在开始准备时,要了解清楚软件开发过程的结果、准备用户的业务运行环境、制定运行过程的实施计划和运行评价标准。实施计划包括任务的分配、过渡计划(例如人机并行运行考虑)、操作培训计划、运行步骤等。

(2) 为了使系统转移到用户的业务运行环境中运行,操作者必须在业务运行环境中对系统进行运行测试。如果其测试结果能满足运行的基准要求,则可将系统进行交付。

(3) 在系统向业务运行环境转移中,必须进行以下工作:编制有关转移的文档;进行数据的转移;进行程序的转移;进行试运行;在试运行中进行跟踪;进行业务转移。当这些工作都完成后,进行转移的确认,并将转移结果的评价通知开发人员。

(4) 系统投入正式运行后,必须进行管理。运行管理者和操作者应当收集运行的数据,判别、记录和解决运行中发现的问题,并改进运行环境。

(5) 操作(开发)者应为用户提供运行上的支持,包括对用户进行操作培训和其他培训;应建立接收、记录和解决用户请求的步骤;对用户的请求提供帮助和咨询服务,并对这些请求的响应进行记录和监控;必要时,操作(开发)者应将用户的请求向软件维护过程(参见“软件维护过程”)提供改进信息或修改请求,并由软件维护过程进行解决。

(6) 系统运行评价主要是指对系统的质量指标和其他方面评价。例如系统的响应分析、系统的运行效率、系统的安全性和保密性、系统故障分析、数据及媒体的管理以及人员管理、系统管理、运行时间管理等。

(7) 用户运行评价包括用户所要求的业务功能的实现程度、使用系统的容易程度、用户所设置的资源的运行和管理;系统运行效果以及用户对系统的改进要求等。

5. 维护过程

维护过程是软件维护人员所从事的一系列活动,其目的是在保持软件整体性能的同时修改它,使它达到某一需求,直到其退役才告终止。从维护方式上讲有三种维护:改正性维护、适应性维护以及完善性维护。诊断并校正一个或多个错误的活动称为改正性维护。为了适应变化的环境对软件修改的活动称为适应性维护。当软件投入运行后,根据用户新的需求,或增加新的能力,或改进现有功能所进行的活动称为完善性维护。

维护过程包含的活动有:问题分析和修改分析、修改/实施。对维护的评审/验收、移植、软件退役等。维护过程同时还贯穿软件过程中其他过程的实施;维护人员通过管理过程管理维护过程(参见管理过程);通过剪裁过程剪裁维护过程中的活动(参见剪裁过程);通过改进过程参与维护过程的管理(参见软件过程);通过支持过程中的各个过程实施维护过程中的文档编制、评审、质量保证等(参见支持过程)。当进行某个活动时,维护过程可能需进入开发过程(如在实施软件的修改时),这时维护人员即是那里的开发人员(参见开发过程)。

具体地说,维护过程所涉及的活动有:

(1) 过程的实施准备

这一活动的目的是为维护过程准备最基本的约定。其主要任务有：

① 制定(2)到(6)的活动实施计划和步骤；

② 确定对用户的问题报告和修改请求进行接收、跟踪的步骤以及向用户反馈信息的方式和步骤；

③ 指定文档编制方式、配置管理方式以及各支持过程的实施方法。

(2) 问题分析和修改分析

这一活动的目的是分析软件修改将对系统、接口系统带来的影响并确立修改方案。主要任务有：

① 分析维护类型，是改进型、改正型，还是适应新环境型的维护；分析维护的范围，包括修改规模、成本、时间；分析维护对关键问题(如性能。保密性)的影响；

② 在分析的基础上，选择实施修改的方案。

(3) 修改的实施

这一活动的目的是对问题及修改进行更为详细的分析，并实施修改。主要任务有：

① 详细分析问题报告和修改请求，决定哪些文档、软件单元和版本需要修改；

② 实施修改。此时维护人员需进入开发过程完成修改(参见开发过程)；开发过程中的需求应作出相应的修改，最低要求是保证未经修改的需求不受影响，而新的修改过程的需求得到完全、正确的实现。

(4) 对维护进行评审/验收

这一活动的任务是维护人员同授权修改的机构一起进行评审、评定经过修改之后系统的整体性能。

(5) 移植

这一活动的目的是将一个系统或软件从一个旧的操作环境移植到一个新的操作环境中，并保证该移植活动的正确性。由于涉及软件的修改，因而也是维护活动。其主要任务有：

① 制定移植计划并执行该计划。计划中至少包括对需求分析和移植的定义；移植工具的开发问题；软件和数据的转换问题；移植计划的执行问题；移植的验证问题；以后对旧环境的支持问题；等等：

② 向用户通告移植计划和移植执行情况；

③ 旧环境和新环境最好并行运行，并向用户提供必要的培训；

④ 对移植活动及移植后的结果进行评审。

(6) 软件退役

软件将根据所有者的要求予以退役。此时，维护人员应当：

① 制定软件支持的撤销计划，并执行该计划。计划中至少包括在多长时间后全部或部分地停止软件支持；系统及有关文档如何存档；若以后仍需要支持时的责任问题；若需要，转移到新的软件上的问题；

② 向用户通告退役计划及执行情况；

③ 将退役软件的所有文档、记录数据归档。

7.1.2 支持过程

支持过程是有关各方按他们的支持目标所从事的一系列相关活动。支持过程有助于提高

系统或软件产品的质量,有助于它们的满意运行。这类软件过程可被软件生存周期中其他类软件过程(如获取过程、供应过程、开发过程、运行过程、维护过程等)或本类中的其他软件过程所使用。软件过程的各活动均可使用支持过程。支持过程可由使用它们的组织来实施;或作为一种服务,由一个独立的组织来实施;也可作为项目的一项规定内容,由客户来实施。一个支持过程中的活动,可由实施该过程的组织负责。这类软件过程包括文档过程、配置管理过程、质量保证过程、验证过程、确认过程、联合评审过程、审计过程、问题解决过程等。

1. 文档过程

文档过程是一个记录由某一过程或活动所产生的信息的过程。这一过程的作用是建立计划、设计、开发、制作、编辑、发行和维护等各类文档。这些文档对系统或对软件管理者、工程师以及用户都是必需的。文档过程由以下一些活动构成:

(1) 过程的实施准备

这一活动的主要任务是制订一个文档编制计划。该计划确定需产生的所有文档。对于每一个文档,应列出其标题或名称、目的、预期的使用者、制作过程和各类参加人员的责任以及制作的计划进度等。

(2) 设计与开发

这一活动的主要任务有:

① 根据适用的文档标准,对确定的每一文档,设计其格式、内容说明、图表设置以及包装等;

② 应保证各文档输入数据的来源和适合性;

③ 应对所编制的文档的格式、技术内容以及表达方式是否符合文档标准进行审查。在分发前需经主管人员批准。

(3) 制作与发行

这一活动的主要任务有:

① 对文档进行制作和包装,并按计划提供给预期的使用者。主要材料的存放应考虑到项目的记录、安全、维护和备份;

② 应按照配置管理过程建立控制。

(4) 维护

当对现存的软件产品进行修改时,需要实施维护过程中的有关任务。对处于配置管理之下的软件产品的修改,应按照配置管理过程进行管理。

2. 配置管理过程

配置管理过程是一个应用管理和技术步骤来完成下列工作的过程。这些工作包括确定、定义一个系统中的软件配置项和基线;控制配置项的修改与交付;记录和报告配置项的完成情况和修改请求;保证配置项的完整性、相容性和正确性;以及控制配置项的存储、处理和提交。配置管理过程由以下一些活动构成;

(1) 过程的实施准备

这一活动的主要任务是制订配置管理计划。该计划应指明各项配置管理活动;实施这些活动的步骤;负责实施各项配置管理活动的机构;该机构与其他机构(如软件开发机构)的关系。

(2) 配置的确定

这一活动的主要任务有制订确定项目要控制的配置项及其各版本的方案。对每一配置项

其各版本应指明建立基线的文档;存放该文档的媒体;参照版本;等等。

(3) 配置的控制

这一活动的主要任务是修改请求的确定和记录;修改的分析与评价;批准或不批准请求;修改项的实施、验证和交付。应保存审计记录以跟踪每一修改、修改的原因和修改的授权。

(4) 配置情况报告

这一活动的主要任务是建立指出所控制的项(包括基线)的现状和历史的管理记录以及情况报告。情况报告应包括项目的修改次数、配置项的最新版本、发行标识、发行号、各次发行的比较。等等。

(5) 配置的评价

这一活动的主要任务是决定和保证各配置项对其需求而言的功能完整程度和各配置项的物理完整程度(其设计和编码是否反映最新的技术描述)。

(6) 发行管理与提交

这一活动的主要任务是控制软件和文档的发行和提交。其中主要的代码和文档应妥善保管。涉及安全或保密的关键功能的代码和文档,应按有关机构的政策处理、存储、包装和提交。

3. 质量保证过程

质量保证过程是一个为使软件过程和软件产品符合所规定的需求,并按所制订的计划完成提供适当保证的过程。为了避免产生偏见,实施质量保证的人员不能是直接负责软件产品开发的人员,并应在组织上给予独立的权限。质量保证过程由以下一些活动构成:

(1) 过程的实施准备

这一活动的主要任务有:

① 根据项目的具体情况剪裁质量保证过程,以实现质量保证的目的——保证软件产品和为提供这些产品所使用的过程符合规定的需求并按计划完成;

② 制订进行质量保证过程各项活动和任务的计划,把它编成文档并实施,在合同期内加以维护。

质量保证过程应和相关的验证过程、确认过程、联合评审过程和审计过程密切配合。

(2) 软件产品的质量保证

这一活动的主要任务有:

① 保证合同所需的各项计划均已编制成文档,内容符合合同要求,互相一致,并保证这些计划的实施;

② 保证软件和有关文档符合合同要求并按计划完成;

③ 在准备软件产品的提交时,应保证软件产品完全满足合同要求且为用户所接受。

(3) 软件过程的质量保证

这一活动的主要任务有:

① 保证项目实施的供应、开发、运行、维护和支持(包括质量保证)过程符合合同要求并按计划完成;

② 保证开发单位或组织内部的软件工程实践、开发环境、测试环境、资料等的适合且与合同要求相一致;

③ 保证用户及其他合作伙伴按合同和计划提供必要支持和合作;

④ 保证软件产品和过程的度量符合所建立的标准和步骤;

⑤ 保证项目组成员具有参加项目所必需的知识和技能,并接受必要的培训。

4. 验证过程

验证过程的目的是确定一个系统或软件的需求是否完备和正确,以及每一阶段的软件产品是否达到了前面各阶段对它提出的要求或条件。验证可以和使用它的过程(如供应过程、开发过程、运行过程或维护过程)结合在一起实施,也可由一个独立的机构来实施。

验证过程由以下一些活动构成;

(1) 过程的实施准备

① 根据项目的范围、大小和复杂性,指明需要验证的活动和软件产品,并选取相应的验证活动及其实施的方法、技术和工具;

② 根据上述指明的验证活动制订验证计划,编成文档,并组织实施;

③ 把验证所发现的问题或不一致现象,作为问题解决过程的输入。验证活动的结果应能被获取者和其他有关机构所利用。

(2) 验证

① 合同验证:主要验证供应者满足需求的能力,需求是否完备、一致和覆盖用户的要求,是否已规定处理需求变更的适当步骤,是否已规定合作伙伴间的界面与协作的步骤和范围,是否已规定验收准则和步骤等;

② 过程验证:主要验证项目计划需求是否已合适和适时,为项目选取的过程是否合适、按计划完成并满足合同要求,用于项目的标准、步骤和环境是否合适,人员是否按合同要求进行了培训等;

③ 需求验证:主要验证系统需求的完备性、相容性、正确性、可行性和可测试性,系统需求是否已适当地分配到各硬件配置项和软件配置项,软件需求的完备性、相容性、正确性、可行性、可测试性以及是否精确地反映了系统需求,与安全性、保密性和关键性有关的软件需求是否正确等;

④ 设计验证:主要验证设计是否正确并且是否符合可跟踪需求,设计是否实现了适当的事件序列、输入、输出、界面、逻辑流程、错误恢复等,所选的设计是否来自需求,设计是否正确地实现了安全性、保密性和其他关键性需求等;

⑤ 代码验证:主要验证关键的代码是否可跟踪设计和需求、可测试、正确、符合需求和编码标准,代码是否实现了适当的事件序列、输入、输出、界面、逻辑流程、错误恢复等,所选的代码是否来自设计或需求,代码是否正确地实现了安全性、保密性和其他关键性需求等;

⑥ 集成验证:主要验证每一个软件配置项的部件和单元是否已完全、正确地集成到该软件配置项中,系统的部件(硬件配置项、软件配置项、人工操作)是否已完全、正确地集成到该系统中,集成的各项任务是否均已按集成计划完成等;

⑦ 文档验证:主要验证文档是否合适、完整和相容,文档的制作是否及时,文档的配置管理是否遵照所规定的步骤等。

5. 确认过程

确认过程是确定需求和最终建成的系统或软件是否满足原计划特定应用的过程。确认可由一个独立于供应人员、开发人员、操作人员或维护人员的机构来执行。确认过程由以下一些活动构成:

(1) 过程的实施准备

这一活动的主要任务有：

① 应开发确认计划并编成文档。该计划至少应包括要确认的配置项，要实行的确认任务，确认的资源、责任和计划进度，向获取者和其他合作伙伴提交确认报告的步骤等；

② 应组织实施确认计划。对于在确认中发现的问题和不一致现象，应作为问题解决过程的输入。确认活动的结果应能被获取者和其他有关机构所利用。

(2) 确认

这一活动的主要任务有：

① 准备测试需求、测试用例和用于分析测试结果的测试规格说明书；

② 确认这些测试需求、测试用例和测试规格说明书能反映该特定应用的特殊要求；

③ 按上述要求实施测试；

④ 确认该软件所规定的衡量标准；

⑤ 确认该软件实现了安全性、保密性和其他关键性需求。

6. 联合评审过程

联合评审过程是评价项目的某个活动或阶段的执行情况和产品是否合适的过程。它可以由任意两个合作伙伴所使用，由其中的一方评审另一方。联合评审过程由以下一些活动构成：

(1) 过程的实施准备

联合评审过程按以下要求进行：

① 应按项目计划的规定定期执行；

② 需评审的资源应经双方同意；

③ 每次评审前双方应就会议日程、要评审的软件产品、范围与步骤等取得一致；

④ 评审中所发现的问题应加以记录并作为问题解决过程的输入；

⑤ 评审结果应归档并分发。评审方要把评审结果的合适性(如批准、不批准)通知被评审方。

(2) 项目管理评审

主要任务是对照项目的计划、进度表、标准和指导方针等来评价项目的进展情况。

(3) 技术评审

主要任务是评价软件产品的完备性和适合性及其和标准和规格说明书的一致程度。

7. 审计过程

审计过程的目的是确定遵照需求、计划合同的程度。审计可由任何两个合作伙伴使用，由其中一方审计另一方的软件产品或活动。审计过程由以下一些活动构成：

(1) 过程的实施准备

审计过程按以下要求进行：

① 应按项目计划的规定定期执行；

② 审计人员不能对他们审计的软件产品和活动有任何直接责任；

③ 每次审计前双方应就日程、要审计的软件产品、范围与步骤等取得一致；

④ 审计中所发现的问题应加以记录并作为问题解决过程的输入；

⑤ 审计结果应归档并提供给被审计方。

(2) 审计

审计的内容包括：

① 软件产品是否反映了设计文档的要求；

② 文档所描述的验收评审和测试需求是否适合于软件产品的验收；

③ 测试数据是否遵照规格说明书的要求；

④ 软件产品是否测试通过并满足规格说明书的要求；

⑤ 测试报告和使用手册是否完整和适合；

⑥ 各项活动是否都已按可应用的需求、计划和合同完成；

⑦ 成本和进度表是否符合于所制订的计划。

8. 问题解决过程

问题解决过程是一个用于分析和排除在开发、运行、维护或其他过程中发现的问题或不一致(不管其性质和来源)的过程。其目的是提供一种适时的、可信赖的并编成文档的手段，以保证分析和排除所有的问题并指明各种倾向。问题解决过程由以下一些活动构成：

(1) 过程的实施准备

问题解决过程应是一个保证做到以下各点的闭环：所有问题被及时报告并进入问题解决过程；启动有关活动；原因被指明、分析和消除(如有可能的话)；问题被解决；跟踪和报告实施情况；按合同规定保存问题的记录。

这一活动的主要任务有：

① 根据对问题分类及排定优先次序的方案，对每一问题进行归类；

② 评价问题是否已解决，不良倾向是否已被扭转，修改是否已在适当的过程和软件产品中正确实施，并确定是否会带来新的问题。

(2) 问题解决

当在一个软件产品或活动中发现问题或不一致时，应对所发现的每个问题编写问题/更改报告。问题/更改报告应描述需解决的问题及其原因(如可能的话)。该报告将作为问题解决过程的输入去纠正缺陷、产生缺陷的原因，以及扭转不良的倾向。

7.1.3 组织过程

组织过程是指那些与软件生产组织有关的过程，包括管理过程、基础设施过程、改进过程和培训过程。

1. 管理过程

管理过程是软件生存周期过程中管理者所从事的一系列活动。管理者负责对所从事的过程，例如获取、供应、开发和支持等过程的活动进行管理；软件管理过程适用于必须对各自的过程进行管理的任何一方。

软件管理过程的目的是在一定的时间和预算范围内，有效地利用人力、资源、技术和工具，完成预定的系统或软件产品，实现预定的功能和其他质量目标。管理技能往往是系统或软件产品成败和软件质量高低的重要条件。大型软件项目涉及较多的人力、资源和时间，管理问题尤为突出。

软件生产是一项劳动和智力密集的活动，它是以人为中心的过程，具有可见性差和定量化难的特点。可见性差是指软件研制进度不易标识，存在问题不易及时发现和纠正，其过程容易出现修改和反复。定量化难指软件的成本、生产率和质量不易度量。因此，软件管理有特殊的复杂性。

软件管理过程是随着软件工程的发展而发展的。早在60年代末，已经有人讨论大型软件

研制项目的组织管理问题。软件工程实践中发生的种种问题,往往是与管理密切相关的。例如,进度推迟、经费超支、质量差、程序人员不称职等。可管理性成为软件生产工程化的重要标志。因此,与软件工程化、产品化过程相适应,形成新的分工,出现了组织管理软件生产的管理员。这些管理员的管理活动体现了最初的软件管理过程。

软件管理活动的研究与软件管理的研究密切相关。最初主要在人员和组织方面,如软件心理学的研究。随着软件技术的发展,软件管理本身出现了专门的方法、技术和工具,来支持软件管理活动。

80 年代以来,随着软件和软件工程的进一步发展,例如软件工程模型化、标准化、软件质量度量及评价技术、软件经济学、软件开发环境和工具等方面的发展,特别是 1984 年以来软件过程的研究,明确了软件管理过程的概念和内容,进一步促进了软件管理过程的发展。

软件管理的对象是进度、系统规模和工作量估计、经费、组织和人员、风险、质量、作业、环境配置等。因此,软件管理一般可分为进度管理、成本管理、质量管理、人员管理、资源管理和标准化管理等。这些管理一般都有其各自的活动内容。软件管理过程把这些活动归纳起来,抽取其活动共性,一般可包含下述活动:

(1) 过程的实施准备

一开始先要搞清楚进行管理过程的需求,管理者通过调查研究确认达到需求的可能性。在必要时可通过有关各方协商来修订和完善需求。

(2) 管理计划的制定

制定计划的任务包括规定进度、分配资源、决定项目有关的组织和承担人员(包括人员的地位、作用、职责、规章制度等)、根据规模和工作量估计进行分配任务、风险定量化、制定质量管理指标、编制预算和成本、准备环境和基础设施等。

(3) 计划的实施和控制

管理者根据需求对过程进行控制。他们监督过程的实施、提供过程进展的内部报告和按合同规定向获取方提供外部报告,并应当调查、分析和解决在执行过程中发现的问题,对计划进行调整和修改。问题及其解决办法都应写成文档。

(4) 计划完成程度的评审和评价

管理人员应对计划完成程度进行评审,对项目进行评价,并对计划和项目进行检查,使计划和项目在完成或变更之后保持完整性和一致性。

(5) 管理过程完成时编写文档

管理者根据合同确定此过程是否完成。如果完成,应从完整性方面检查项目完成的结果和记录,并把这些结果和记录编写成文档和存档。

2. 基础设施过程

基础设施过程是建立、维护任何其他过程所需的基础设施的过程。基础设施可以包括硬件、软件、工具;技术、标准以及开发、运行、维护所需的设施。软件过程中的许多过程都应明确该过程的基础设施。本过程的主要活动是定义并建立各过程所需的基础设施,并在其他相关过程执行时维护所建立的基础设施。

3. 改进过程

改进过程是建立、评估、度量、控制和改进软件生存周期过程的过程。其主要活动有制定一套组织计划,评估相关过程并实施分析、改进过程。

软件过程中的任何一个过程都有可能涉及这一过程。

4. 培训过程

培训过程是为系统或软件产品提供人员培训的过程。软件获取、开发、运行和维护的效果主要取决于有关人员所具备的知识和熟练程度。因此,拟定人员培训计划并及早实施是非常必要的。本过程要完成的主要活动有制定所需的人员计划及培训计划,开发培训资料及实施培训活动等。

软件生存周期过程及其关系如图 7.1 所示。

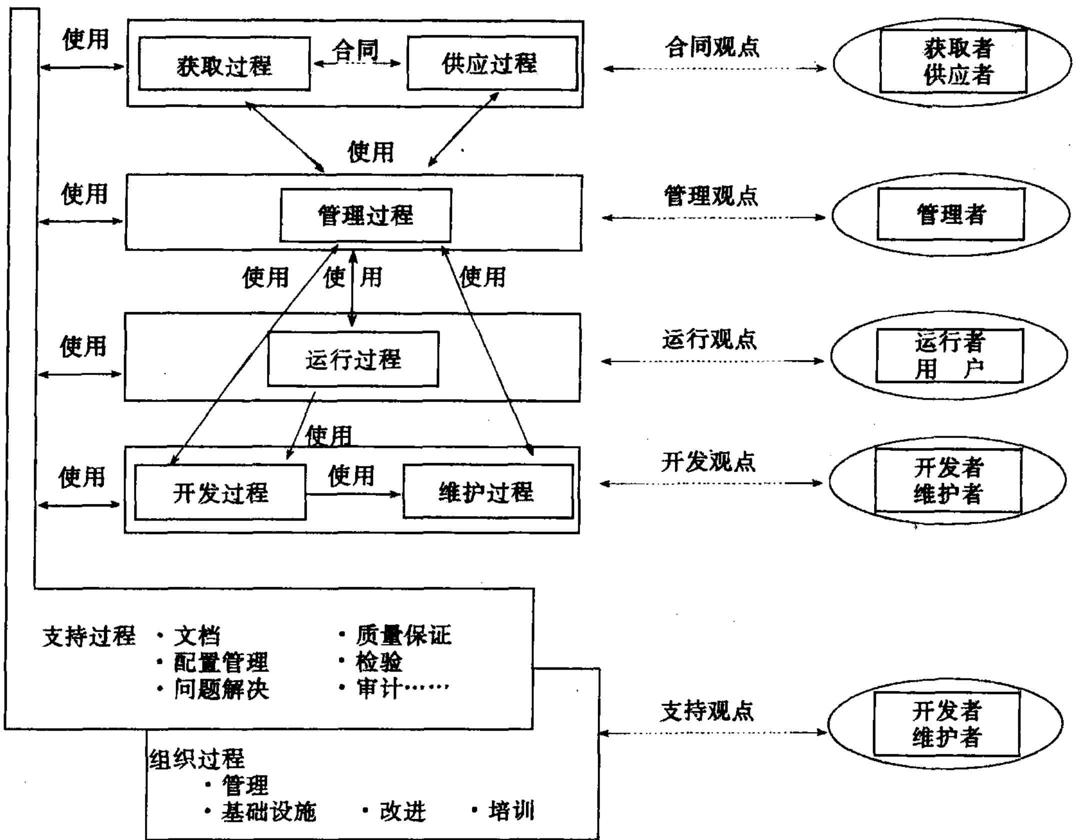


图 7.1 软件工程过程及其关系

7.1.4 剪裁过程和过程模型建造技术

1. 剪裁过程

为了有效地实施软件过程,应针对特定领域的软件工程,对选定的过程模型和标准进行剪裁,以形成这一工程的模型及标准,形成该工程的各个软件过程和活动。

剪裁过程作为一类软件过程,是对软件过程和活动实施剪裁的过程。其主要活动有:

(1) 指明工程环境

这一活动应指明影响剪裁的工程环境特征,如使用的工程模型和方法,系统和软件需求,机构的政策与策略,系统和软件的规模、重要性和类型,参与工程的人员和合作伙伴的素质、数

量等。

(2) 收集信息

一般来说，参与剪裁的人员包括用户、工程支持人员、负责合同签订的人员以及潜在的投标者，应向所有会影响剪裁的组织收集信息。

(3) 选取过程、活动和任务

这一活动将根据上述收集的数据，确定要实施的过程、活动和任务。

(4) 编制文档

这一活动为所有剪裁决定及这些决定的理由编制文档。

剪裁可分为两级，第一级是根据应用领域的不同进行剪裁，第二级是根据每一具体项目或合同进行剪裁。就第一级而言，涉及了开发过程的剪裁。对嵌入或集成在系统中的软件，应考虑软件剪裁过程的所有活动，并且必须明确开发者是否实施或支持系统的活动；对于独立的软件，关于系统的活动可以是不需要的，但应加以考虑。也涉及了与评价有关活动的剪裁。评价可分为五类，即过程内部的评价，验证和确认，联合评审和审计，质量保证以及过程改进。应根据项目或机构所涉及的范围、大小、复杂性和重要性相应地选取和剪裁这些评价。在实施剪裁过程中，还应指明或考虑一些关键的项目特征，例如机构政策、获取策略、支持的概念、生存周期模型、合作伙伴、系统级特征、软件级特征以及风险性等。

2. 过程建模技术简介

过程建模技术是当前软件工程学科的一个重要研究方向，涉及建模目的、方法及描述等内容。为了对建模目的有一个比较好的理解，首先简述一下软件过程具有的主要性质。

(1) 软件过程主要性质

① 分工协作性

一个软件项目通常是由多个人参与研究，这些人又有从事管理和从事开发工作之分；另外，就开发工作而言，又可以把参与人员分为系统开发组、测试组等不同活动的工作组。每组由多人分工负责不同的任务，从而形成一个多层次多侧面的项目组织关系。软件项目的成功与否主要依赖这种个人之间、小组之间及各类参与人员之间的密切合作。

② 易变性

软件活动由许多活动组成，这些活动在某一时刻的状态是事先难以确定的。由于用户需求经常变化，使软件过程在执行中可能增加一些活动，也可能对某项任务需要进一步迭代执行。这些都取决于任务完成情况和评审结果，这些因素是预先无法准确估计到的。

③ 动态性

软件过程具有很强的时间依赖性，即软件过程本身的状态随着开发活动的进展而变化。例如，在某一活动中的角色，可能在另一活动中担任另外的角色，而某一活动的输出产品可能是另一活动的输入。

④ 逐步细化性

软件过程的高度复杂性使人们无法在事先就对过程的所有细节都了解得很清楚，也没有必要这样做。因此，开始时只能对过程进行较粗的计划。只有到了一定阶段，才能真正详细地计划某一过程。

⑤ 不可完全形式化

软件过程中包括大量的创造性活动，一般来说，这些活动是不可形式化的，也不必形式化。

对创造性较强的智能活动，只需给出有关的约束条件，其余工作可由开发者自由的发挥。

⑥ 并行性

软件过程是由多人或多个小组分工协作完成的，所以在某个时间段内就会有多人并行地完成某些任务，这是软件过程的一个重要特征。

(2) 软件过程建模技术综述

软件过程建模是对实际的软件过程的再工程(Reengineering)，即在简化的实际过程基础上对软件过程进行抽象描述。过程建模活动是软件过程中最主要的活动，所有其他工程活动都是基于建模活动的结果来进行的。过程建模技术包括四方面内容：建模目的、建模方法、建模语言和软件过程与过程模型的度量。其中建模目的决定了建模活动的范围；建模方法和建模语言是对建模目的的支持和过程模型形式化的手段，并决定了过程模型的质量；对过程模型的度量与评估是修改和演化过程模型的基础，以便使过程模型更好地与软件过程相吻合，并充分反映一个组织的过程成熟度。可见，过程建模方法和建模语言是过程建模技术的关键。

① 过程建模目的

过程建模极为广泛，综合起来主要有：

(i) 使人们易于理解软件过程并在此基础上进行交流

由于软件过程的复杂性，使人们常常不能全面、仔细地了解过程的全貌，况且背景、经历不同的人对同一过程可能会有不同的理解。因此通过过程建模，可以使不同的人对过程达到一个共识，共享过程知识，并在此基础上进行交流。

(ii) 支持对过程的分析

通过对过程进行形式化或非形式化的建模，为进行过程分析提供了基础。人们可以对过程活动以及活动间相互关系进行分析、比较和预测，以评估和改善过程的有效性。

(iii) 支持过程中的通信

由于软件过程多是由多人在一定时间阶段内执行的，所以过程中涉及到的人员之间存在大量的信息交流活动。过程模型可以为有关人员、小组和项目提供必要的过程信息和通信支持，使人们之间协同工作更加有效。

(iv) 支持过程的改进

可以对过程模型进行分析，使人们识别过程的各个部分的功效，找出其可能的改善部分，对模型进行修改；并在模型实际修改前通过分析模型中各部分之间关系来估计出修改后产生的影响。从而支持严格管理下的过程进化。并使软件组织积累了有效的过程改善知识。

(v) 支持过程的管理

过程模型可以帮助管理人员制定项目计划，监控、管理和协调项目实施过程，估算软件创建或演化的属性。例如，估算各活动的实际进展情况。过程模型还可作为过程度量的基础，通过它来定义度量点、度量内容等。

(vi) 支持过程复用

通过过程模型可以重用良好定义的软件过程，这种复用可以在不同的项目之间进行。还可以针对具体项目的特点，对已有的过程进行裁剪、扩充，使之适合特定项目复用的要求。

(vii) 提供对过程的自动执行支持

这种对软件过程的自动执行支持需要基于一个过程驱动的软件工程环境。在这种环境中，通过过程实施机制，按照定义的过程模型自动地驱动实际过程的执行。这种支持包括对环

境用户的资源配置、提供各用户之间的通信服务以及组织和协调个人或小组之间的工作等。

② 过程建模方法

过程建模涉及到软件产品的开发与维护、软件项目管理、过程管理与过程改善等各个方面,涉及到过程的活动、角色、产品、资源和约束等各种过程实体,还涉及到建模所用的形式化方法,加之软件过程本身的复杂多样性,因而使得构造过程模型的方法也是多种多样的。把目前已有的各种建模方法及建模形式化语言综合起来考虑,大致有两种不同的过程建模分类方法,它们是主要考虑过程所涉及的实体的分类方法和主要考虑建模所采用的不同形式化方法的分类方法。下面分别评述这两种分类方法中所包含的各种建模方法。

第一种分类方法　主要考虑过程所涉及的实体的分类方法

这种分类方法是以过程所涉及的各个过程实体(如活动、角色。产品、资源和约束等)为出发点来考虑过程建模。主要有两种不同的建模方法。

(i) 以活动为中心的建模方法

这种建模方法以过程中一类主要实体——过程活动为中心构造过程模型。步骤是先确定这些活动以及它们之间的执行顺序,再收集与各个活动相关的其他数据,例如活动所涉及的角色、产品、资源和约束等。这种方法能够直观地反映实际过程的工作流程,且无二义性,使人们很容易理解、分析、计划、管理和控制过程的执行。特别对管理人员来说,他们能够清楚地监视过程的进展情况,从而进行有效的协调与管理。这种方法的缺点是,由于活动是动态实体,而且实际流程也是动态的多变的,因此过程模型缺乏稳定性。即实际过程中发生任何变化都将影响到原来的过程模型。况且在建模时就要求建模人员具有所建模之过程活动的预备知识,并要在模型实施中进行一致性维护。这种方法比较适合对实际过程的执行指导与控制等目的。支持这种方法建模的形式化语言有 APPL/A, HFSP, MELMAC, PRISM 等。

(ii) 以角色为中心的建模方法

这种方法以过程中另一类主要实体——角色为中心建模。因为角色是组织结构中的基本构成因素,是一个易于接受和理解的相对稳定的抽象实体,并且一个项目的任务通常是按照角色来分解的,因此以角色为中心的建模方法显得比较自然。建模步骤是先确定各个角色的任务以及角色之间的耦合关系,再以角色为中心收集过程的其他数据。如活动、产品、资源和约束等等。由于角色是过程的一个不变的实体,因此构造的过程模型具有较好的稳定性。这种方法还能明确地描述过程的组织方面的信息,使得参与项目的人员易于明确自己的任务,也便于对项目的计划、管理与控制。它的缺点是不能对过程的工作流程有一个明显的描述和定义,人们难以从整体上了解一个过程和他们在过程中的位置。另外,随任务的逐步分解细化,涉及的角色也增多,角色间的关系也更复杂,不利于低层次上的过程管理。这种方法的代表有 Role Interaction Nets(Cain 93)等等。

除上述两种建模方法外,在这种分类中还有以产品为中心的建模(Waters 89)和基于过程模板的建模(SPC 93)等方法,因它们都存在一些问题,在此从略。读者如有兴趣可阅读有关参考文献。

第二种分类方法　主要考虑过程建模所采用的形式化方法的分类

不论采用什么方法进行过程建模,都需要有相应的形式化方法进行支持。本节介绍主要考虑建模所采用的不同形式化方法和语言风格对过程建模方法的分类,这种分类把过程建模主要分为五种不同的方法。

(i) 过程程序设计方法

这种建模方法的出发点是“软件过程也是软件”的概念(Osterweil 87)。Osterweil 认为软件过程与软件产品具有广泛的类同性,对软件过程的描述亦是一种程序设计形式。这种方法是把过程所涉及的软件对象用其所需工具与开发方法编程,它通过关系、触发器和谓词等机制对过程的功能、行为和对象进行详细、确定的算法描述。这种方法不是控制活动的执行方法,而是控制生产产品的交互过程和产品的结构。它的优点是能成功地进行交互控制工作,并自动维护对象之间的一致性。它的缺点是由于其严格过程化,不能支持大范围的并发活动,且无法描述软件过程的动态变化情况。支持这种建模方法的形式化语言有 APPL/A, Hindsight 等。

(ii) 功能分解方法

这种建模方法把一个软件过程用带有输入属性和输出属性的一个过程元素集来表示。即把一个过程定义为反映输入与输出关系的数学函数集。这些函数可以按照语法进一步进行层次分解,形成一个过程的多个子过程步。这种分解可以一直进行下去,直至产生的子过程步映射到一个外部工具或由人员操作实现。这种方法支持过程步的并行执行,以及过程步之间的串行、迭代等执行方法。它还提供了控制过程状态行为的元操作,例如创建、挂起、恢复执行等。这种支持对过程的理解、执行指导和复用等建模目的,主要面向过程的功能、行为和信息等方面。

(iii) 基于 Petri 网的建模方法

当前的过程建模方法中,基于 Petri 网或其变种(如扩充的 Petri 网、FUNSOFT 网等)进行过程建模的方法占有相当数量。因为 Petri 网具有很强的表达能力,能够有效地形式化描述软件过程的并发性和活动与产品之间关系。而且这种图形表示易于理解和管理软件过程。这种方法的优点是较好地考虑了任务的激活条件、活动的执行顺序和活动产生的信息实体之间的转换情况。它的缺点是忽略了活动对内部状态所产生的影响。另外,由于网的全局相关性,使对过程的任何改动都会影响到其他部分,不利于过程复用。这种方法建模的代表有基于 FUN-SOFT 网的 PRISM 项目、基于扩充 Petri 网的 MELMAC 和 SLANG, Process WEAVER 和 Role Interaction Nets 等。

(iv) 基于规则的建模方法

基于规则的建模方法和语言是人们普遍看好的一种过程建模技术。这种方法提供了活动的动态链接机制,从而很自然地描述了过程的不可预见性,也为人们控制过程提供了最为灵活的手段。这类语言通常还提供回溯、向前链接、向后链接等自动执行机制,以及规则推理、调度和控制过程活动的机制,并为过程的修改提供了灵活方便的环境。但是这种方法不利于人们理解软件过程,在构造、分析过程模型中也存在较大困难。它对多人并行工作和协同工作也缺乏有效的支持。这种方法的代表有 PEACE, Marvel MASP 等等。

(v) 基于知识的建模方法

基于知识建模方法提供了对过程模型的增量式形式说明能力和可复用能力。这种方法把过程知识(例如过程活动、过程实施者、产品对象和工具以及它们之间关系等)用面向对象方法抽象成各个不同的类,存于知识库中。过程建模时,根据需要查询知识库,从中获取有关过程活动及其他成分的抽象描述,从中选取或构造所需的过程模型,并对其进行分析和推理,最后生成过程实例及相应的活动计划。这种方法中,模型的构造是分层的,由活动的类和子类构成类体系结构。体系中每个活动类或子类都对应有多种资源需求,例如要加工的数据、所需工

具、开发角色等。用类与子类之间的关系描述来表示各种过程关系,如控制流关系、任务的前置和后置条件、不同角色之间的上下级关系、产品的组成关系等。这种方法的代表有 Splib(P. Mi 92),SMART(Gary 94)等。

总的来说,一种建模方法是否合适,对其评定完全依赖于这种方法所建立的模型是否达到了建模目的。为了使建模方法具有广泛通用性和适应能力,集成多种风格形式化于一身的建模方法是十分必要的,即形成混合风格的建模方法。然而这种混合风格的方法在实际应用中还存在许多问题,例如各个模型成分之间的转换、通信、协调和交互作用等。这种混合风格建模的代表有 STATEMATE 等。

③ 过程建模语言

过程建模语言是用于构造过程模型并把它形式化的基本工具。建模语言的表达能力最为重要,它的强弱直接影响到过程模型的适用范围和质量。虽然不同的建模语言因其支持的建模目的不同而在语言的风格与功能上存在差别,但是要构造一个完整且有实用价值的过程模型,不论哪种语言都应该能够描述过程的功能、行为、组织和信息等方面的内容:

功能方面　描述软件过程要执行哪些活动,它们的功能是什么,有哪些信息实体与这些活动有关。

行为方面　描述什么时候执行这些活动,怎样执行,有哪些行为约束条件。例如进入和退出活动的标准、怎样提供反馈、重复执行的条件与多活动选择执行的决策条件等。

组织方面　描述在什么地方,由谁来完成这些过程活动,参与活动和项目的成员的组织结构与成员之间的通信机制等。

信息方面　描述由过程活动操作和生成的信息实体,包括数据、文本、中介产品和最终产品、软件对象以及信息实体结构和它们之间的关系等。

目前国际上已出现 50 多种过程建模语言(Starke 93),它们在语言的基本成分、语言支持的建模目的、语言采用的形式化方法和语言对软件过程功能、行为、组织和信息的描述能力等方面都存在不同程度的差异。下面我们分别考察几种有代表性的过程建模语言。

(i) APPL/A

APPL/A(Sutton 90)是过程程序设计语言风格的典型代表。APPL/A 是 L. Osterweil 等人开发用于构造过程程序的一种形式化的建模语言。它是对 Ada 语言的扩充,增加了软件对象之间永久性关系的定义机制、关系操作的触发机制和表示关系状态的谓词机制,并继承了 Ada 语言的基本特征,例如类型系统、模块定义风格和任务通信方式等。

APPL/A 支持对过程的分析、执行控制和复用等建模目的。它通过关系、触发器和谓词等机制可以对过程的功能、行为和对象进行详细的算法描述,可以完全自动地控制过程的执行。但是 APPL/A 不能对过程的人工活动建模,也难以描述过程的动态性和不确定性。此外它也不能描述过程组织方面的信息,使得难以对参与过程的角色进行管理和调度。

(ii) Merlin

Merlin(Peushel 92)是由德国 STZ 公司和德国 University of Dortmund 合作开发的一个以过程为中心的软件开发环境。它的过程建模思想是基于一组良好定义的软件构造块及其属性和关系来构造软件。它支持基于规则的软件过程的执行。Merlin 中的软件过程是用一些规则来描述的,这些规则可以用向前链接或向后链接的方式解释执行。这些规则可以存储在一个面向对象的数据库中成为永久对象被重复使用。Merlin 中使用的建模语言是一种类 Prolog

的语言,这种语言有很大的灵活性和显式声明能力。通过在语言中增加新规则的能力使系统能适应过程的变化需求,从而形成不同的过程模型。过程实施是由环境的推理机制和驱动机制支持的。

Merlin 支持过程复用、过程改善、过程控制与实施等目的,可以描述过程的功能与行为方面的特征。

(iii) STATEMATE

STATEMATE(Kellner 89)是由美国 Carnegie Mellon 大学 M. I. Kellner 等人使用的一种过程建模语言。它是一种图形式描述语言,由活动图、状态图和模块图组成。活动图用于描述过程的功能特征,主要成分是活动、信息和数据存储方式描述。状态图用于描述过程的行为特征,主要成分是状态和状态之间的变换,其中状态表明活动当前所处的一种工作行为状况,变换则描述了一个状态变换到另一个状态的条件(触发器)。模块图用于描述过程的组织特征,主要成分是模块和信息流,其中模块表示与活动相关的小组或个人,信息流表示了小组或个人之间的信息传递通道。由于 STATEMATE 提供了关于过程的三种视图描述机制,因此大大提高了对过程模型的可理解性。它还提供了一系列对过程进行分析和模拟的设施,以及报告和查询功能等,可对模型进行一致性、完整性和正确性检查。它的缺点是没有提供类型或记录结构等形式的语言机制,从而降低了对产品的分析能力。因此,它主要是一种面向描述的过程建模语言,不能对过程提供指导与控制。

STATEMATE 以对过程的理解、通信、过程改善与过程管理为建模目的,能较全面地描述过程的功能、行为、组织和信息四方面特征,而且具有良好的可视性。

(iv) SLANG

过程建模语言 SLANG(Bandelli 94)是 S. Bandelli 等人在 SPADE-1 过程驱动环境中为过程建模和执行而开发的。SPADE-1 环境由 SLANG 解释器、面向对象数据库及存储设施、同集成化工具子环境相结合的过程解释器接口设施组成。SLANG 基于 Petri 网,提供了特殊的黑色迁移和用户接口位置来管理与外部环境的交互。黑色迁移代表任何外部程序的执行;用户接口位置反映用户或工具产生的相关事件,这些事件被翻译成内部表示与过程实施环境通信。从而使工具和人员之间的交互语义作为过程模型的一部分被描述出来。黑色迁移和用户接口位置是把任务授权给工具和人员的基本机制,也是一种控制外部发生的请求事件的基本机制。

SLANG 支持对过程的理解、通信、人员与工具的交互、过程执行与控制等目的,它可以描述过程的功能、行为等方面特征。

7.2 ISO9000-3 简介

1. ISO9000-3 标准产生的背景

1987 年,国际标准化组织(ISO)发布了 ISO9000 系列标准,包括:

(1) ISO9000 质量管理和质量保证标准——选择与使用导则;

(2) ISO9001 质量体系——设计/开发、生产、安装和服务中的质量保证模式;

(3) ISO9002 质量体系——生产和安装中的质量保证模式;

(4) ISO9003 质量体系——最终检验和测试中的质量保证模式;

(5) ISO9004 质量管理和质量体系要素——导则。

其中，作为“需方对供方要求质量保证”的标准——ISO9001，ISO9002，ISO9003，它们之间的主要区别是工序范围不同，即ISO9001范围最广，从设计一直到售后服务，而ISO9002是ISO9001的一个子集，ISO9003又是ISO9002的一个子集。ISO9004是用于“供方建立质量保证体系的标准”。

由此可见，这一系列标准旨在指导高质量产品的生产，评价、认证产品质量。它最初起源于欧洲经济共同体，并主要针对制造业。但由于市场经济，特别是国际贸易的驱动以及这一标准可适用于更广泛的领域等，因此，目前已在世界范围内极为流行。

关于质量体系，著名的质量管理专家费根堡姆认为，“在制造及传递某种合乎特定质量标准的产品时，必须配合适当的管理及技术作业程序，这些程序所组成的结构，称之为质量体系”。

在一定程度上来说，ISO9000系列标准是这一概念的具体细化，其主导思想是，产品质量形成于产品生产的全过程。因此，应将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；并且质量管理应遵循PDCA循环（即计划Plan—实施Do—检查Check—措施Action），坚持进行质量改进。

ISO9000系列标准原本是为制造业而制定的标准，通过在软件开发中的应用，发现效果并不是十分理想。其主要原因是，传统制造业的产品生产与软件开发具有很大的差异。在过程方面，制造业的产品需要经历“设计”、“生产”、“储存”、“发布”、“销售”、“运输”、“服务”等过程，而软件产品/系统基本上不需要“储存”、“运输”等过程；另外，与传统制造业产品生产相比，软件开发还具有自己的一些特点，例如：

① “设计”是核心，且“设计”到“生产”过渡的时间间隔“很小”；

② 软件质量检验技术与工具尚不完善；

③ 由于软件是知识的固化，因此不但产品的复杂度比传统制造业的产品要高，而且随着知识的快速发展，软件产品/系统更新和演化更快；

④ 开发环境需要有助于开发人员创造性的发挥；特别是，软件开发又是团队协同的工作，需要将软件开发的个体性与群体性有机结合起来；

……

于是，国际标准化组织以ISO9000系列标准为基础，以“追加”形式，制定了ISO9000-3标准，成为“使ISO9001适用于软件开发、供应及维护”的“指南”。

2. ISO9000-3要点

ISO9000-3主要是给出了软件开发中的质量体系框架。其中包括供需双方的责任，供需双方所进行的一些有组织的质量活动，以及与之相关的规范化（文档化）；而没有规定质量管理以及每一活动所采用的方法和程序。由此可以看出，ISO9000-3为软件企业实施ISO9001提供了一个指南。

为了更好地理解ISO9000-3，下面给出有关软件质量和质量体系的一种解释。

（1）软件质量与质量模型

① 软件质量

在ANSI/IEEE Std 729-1983中，把软件质量解释为“与软件产品满足规定的和隐含的需求能力有关的特征或特性的全体”。根据这一关于软件质量的解释，可以看出：需求是度量软件质量的基础，不满足需求的软件就不具备质量；满足规定的和隐含的需求能力，往往是通过一系列软件特性来体现的，这些特性包括正确性、可用性、可移植性等。

② 质量模型

关于软件质量特性，比较有影响的研究成果是 McCall 等人于 1979 年提出的 McCall 软件质量模型，如图 7.2 所示。

评测准则 \ 质量因素	正确性	可靠性	效率	完整性	可用性	可维护性	灵活性	可测试性	可移植性	复用性	互连性
可跟踪性	●										
完备性	●										
一致性	●	●				●	●				
安全性		●		●							
容错性		●									
准确性		●									
简单性		●				●	●	●			
执行效率			●								
存储效率			●								
存储控制				●							
存取检查				●							
操作性					●						
易训练性					●						
简明性		●				●		●	●		
模块独立性		●				●	●	●	●		●
自描述性						●	●	●		●	
结构性						●					
文档完备性						●					
通用性							●		●	●	●
可扩展性							●		●		
可修改性							●	●		●	
自检性				●	●			●			
机器独立性									●	●	
软件独立性									●	●	
通信性					●						
通信共享性											●
数据共享性											●
I/O 容量					●						
I/O 速度					●						

图 7.2　McCall 软件质量模型

其中：

各质量因素的含义为：

正确性：在预定的环境下，满足设计规格说明以及用户预期目标的程度。

可靠性：软件按照设计要求，在规定时间和条件下，持续运行的程度。

效率：为了完成预定功能，软件所需计算机资源的程度。

完整性：为了某一目的，保护数据免受偶然的或有意的破坏、改动或遗失的能力。

可用性：用户学习、使用软件，以及为准备输入数据和解释输出数据所需工作量的大小。

可维护性：为满足用户新的要求，或环境发生了变化，或发生了新的错误，进行相应诊断和修改所需工作量的大小。

可测试性：测试软件以确保能够实现预定功能所需工作量的大小。

灵活性：修改或改进已运行的软件所需工作量的大小。

可移植性：将一个软件系统从一个计算机系统或环境移植到另一计算机系统或环境中所需工作量的大小。

复用性：一个软件或其中的部件，能够再次用于其他应用的程度。

互连性：将一个软件连接到其他系统所需工作量的大小。其中，这里所说的“连接”，包括联网、或与其他系统通信、或把其他系统置于自己的控制之内等。该质量因素亦称为互操作性。

各评测准则的含义为：

可跟踪性：在特定的软件开发和运行的环境下，追溯设计表示的能力或实际程序部件追溯原始需求的能力。

完备性：软件需求得以实现的程度。

一致性：在软件设计和实现的整个过程中，技术和表示的一致程度。

安全性：防止软件受到有意或无意存取、使用、修改、毁坏以及泄密的程度。

容错性：当系统出现错误，例如机器故障，输入不合理的数据等，能以某种预定方式进行适当处理，使系统继续执行以及恢复系统的能力。有时，也称为健壮性。

准确性：软件系统实现计算或控制精度的程度。

简单性：在可理解的简单方式下，定义并实现软件功能的程度。

执行效率：为实现某种功能，提供使用最少处理时间的程度。

存储效率：为实现某种功能，提供使用最少存储空间的程度。

存取控制：对用户存取权限实施控制的程度。

存取检查：对用户存取进行审查的程度。

操作性：操作软件的难易程度。通常，操作性取决于软件提供的操作规程以及输入/输出方法。

易训练性：软件辅助新的用户使用系统的能力。通常，易训练性取决于软件提供帮助用户使用系统的方法和方式。

简明性：软件(程序和文档)易读的程度。有时，也称为可理解性。

模块独立性：软件模块(部件)实现“高内聚低耦合”的程度。

自描述性：软件自身对其功能描述的程度。

结构性：软件结构“良好”的程度。

文档完备性:软件文档齐全、描述清楚、满足规范或标准的程度。

通用性:软件功能覆盖可用范围的程度。

可扩展性;软件体系结构、数据设计和过程设计的可扩展程度。

可修改性:软件容易修改且不会产生副作用的程度。

自检性:监控自身操作效果和发现自身错误的能力。

机器独立性:不依赖于特定计算机和特定设备而能工作的程度。

软件独立性:不依赖于非标准的程序设计语言特性、操作系统特性,或其他环境约束,而靠自身能实现其功能的程度。

通信性:提供有效 I/O 方式的程度。

通信共享性:使用标准的通信协议、接口和带宽的标准化程度。

数据共享性:使用标准数据结构和数据类型的程度。

(2) ISO9000-3 质量体系要素

ISO9000-3 针对 20 个质量体系要素,在软件企业中实施做出了解释,并且与 ISO9001 标准的文本描述是完全对应的。

① 管理职责:负责人工作职责

(i) 组织制定机构的质量方针、质量目标和质量承诺,要求机构内各级人员理解质量方针,并贯彻执行。

(ii) 对所有与质量相关的管理人员、执行人员和验证人员规定职责、权限和相互关系,为相关活动提供充分的资源支持,委派专人负责按标准建立、实施和保持质量体系。

(iii) 负责定期组织机构内的管理评审,审查质量体系是否满足标准及企业需要,是否持续有效。

② 质量体系

(i) 建立质量体系,形成文件并加以维护。

编制质量手册,明确质量方针、目标、组织结构等各个方面,以及质量体系文件概要。

确定质量手册的管理(制定、修改、批准和控制)。

(ii) 编制相应的程序文件,并加以贯彻实施。

(iii) 质量策划与对质量计划的要求。

质量策划:确定质量以及采用质量体系要素的目标和要求的活动(构思和安排)。

质量计划:针对特定产品、项目或合同,规定专门的质量措施、资源和活动顺序的文件(具体实施)。

对新产品、新项目或新合同应制定质量计划。

③ 合同评审

(i) 在合同签订之前,应对合同、标书或订单进行全面评审,以保证其中的条款能够接受,也有能力满足。

(ii) 对上述工作程序建立文件定义,并贯彻执行:

评审参与组织及其职责、活动;

评审结论及其管理;

合同修订及其管理。

④ 设计控制

在产品设计方面进行质量控制,并保持稳定、制度化,包括:

设计和开发的策划;组织上的接口和技术上的接口;设计输入,确定对设计输入的要求;设计输出,确定对设计输出的要求;设计评审;设计验证;设计确认;设计更改。其中,

(i) 设计和开发的策划

开发策划包括:确定需求分析、设计、编码、集成、测试、安装和支持软件产品验收等各项活动,并按开发计划的方式形成文件。

开发策划宜涉及下列事项:项目定义、项目输入与输出、项目资源的组织、组织接口和技术接口、进度安排、使用工具、技术、配置管理、病毒防护等方面。制定开发计划,并标明相关计划(质量计划、风险管理计划、配置管理计划、集成计划、测试计划、安装计划、移交计划、培训计划维护计划、重用计划)。

开发计划主要包括:确定项目如何管理、要求的进度评审,并考虑合同的要求,规定提交管理者、顾客和其他有关各方的报告类型和频次。

开发计划和有关计划可以是一份独立文件,或是另一文件的部分或由若干文件组成。

(ii) 组织和技术接口

清晰规定软件产品各部分的职责范围和在各部门之间传递技术信息的方式,可以要求分承包方提交开发计划,以供评审。

确定接口时,要仔细考虑在顾客和供方之外需参与设计、安装、维护和培训活动的各方,以保证得到适当的能力和培训,达到承诺的服务水平。

明确按合同规定顾客可能有某些职责,并解决有关的事项。

进行供方和顾客同时参与的联合评审,定期安排或在发生重大项目事件时进行。联合评审要覆盖下述方面:供方软件开发的进展、顾客同意承担活动的进展、开发的产品是否符合需求规格说明、开发中涉及系统最终用户的活动的进展、验证结果、验收测试结果等。

(iii) 设计输入(需求规格说明书)

需求规格说明最好由顾客提供,也可以由供方提供。

需建立制定规格说明的形成文件的程序,包括商定需求和授权更改的方法、对原型或演示的评价方法、记录和审查双方讨论的结果、明确定义术语、解释需求背景等。要取得顾客对需求规格说明的认可。

可以采用交谈、调查、研究、提供原型、演示和分析等方法制定需求规格说明。

需求规格说明在接受时可以是不完全明确的,在项目进行期间可以继续制定;也可以修订合同,对其进行更改,但最好应加以控制。

需求包括用户要求的所有方面,包括但不限于 ISO/IEC 9126 中所给出的各个特性。

需求最好用产品验收时能确认的形式来表达。

(iv) 设计输出

要求的设计输出最好按照选定的方法予以确定,并形成文件。这种文件应是正确、完整和符合需求的。

设计输出可以包括:体系结构设计规格说明、详细设计规格说明、源代码、用户指南。

(v) 设计评审

供方应对所有软件开发项目的评审过程做出计划,并加以实施。

评审活动的正式程度和严格程度,应与产品复杂性及软件产品规定用途关联的风险程度

相适应。

应形成处理这些活动期间发现的过程缺陷和产品缺陷或不合格事项的程序文件。

设计评审中最好考虑设计活动的内在因素,如可行性、安全性、编程规划和可测试性。

评审结果以及为确保规定要求所需的进一步活动,最好予以记录,并检查。

建议只有当所有已知缺陷都得到满意的解决,或继续进行的风险已知时,才继续进行下一步设计活动。

(vi) 设计验证

建议在开发过程中,适当地进行设计验证,可以包含设计输出评审,也可以针对其他开发活动的输出进行。

按照质量计划或程序文件制定验证活动计划,实施设计验证。

对验证结果和为满足规定要求所需的进一步活动,最好予以记录,并检查。

建议对任何发现的问题都要予以充分论述并解决。

只有经验证的设计输出才能提交验收和后续使用。

(vii) 设计确认

在产品提交顾客验收之前,供方最好按规定的预期用途,确认该产品,可以进行多次确认。

对确认的结果和需要进一步采取的措施,建议予以记录,并且在措施完成时检查。

(viii) 设计更改

供方应建立和维持用于控制实施任何设计更改的程序,其目的是为了:对更改形成文件,证明更改是正确的,评价更改的后果,批准或不批准更改,实施并验收更改。

⑤ 文件和资料的控制

(i) 应建立并保持形成文件的程序,包括下述两方面文件:

对于本标准相关的所有文件和资料;

外来的原始文件等,如:标准、参考材料、顾客提供的样本等。

(ii) 文件和资料的批准与发布管理(审批适用性)程序,防止使用失效或作废的文件。

(iii) 文件和资料更改(审批更改)程序,保证文件和资料适用、系统、协调和完整。

⑥ 采购

确保采购的产品符合规定要求,包括:

对分承包方的评价;

对采购文件的要求(包括的详细信息要求及审批);

对采购产品的检验。

⑦ 顾客提供产品的控制

对顾客提供的产品,建立并保持储存和维护的控制程序,并形成文件。产品包括:顾客提供的供应品或有关活动。若出现损坏、不适用等情况,应予以记录并通告顾客。

⑧ 产品标识和可追溯性

在接受和生产、交付及安装的各阶段,对产品以适当的方式进行标识。这种标识应有惟一性和可追溯性。对成品与半成品均需管理,防止产品在加工过程中出现混乱。

⑨ 过程控制

对直接影响产品质量的生产、安装和服务过程进行有效控制,制定程序并形成文件(制度化),控制对象可以是过程本身,也可以是与过程相关的方法、设备、材料、环境以至人员等。对

影响过程质量的所有因素，包括工艺参数、人员、设备、材料、加工和测试方法、环境等加以控制。具体规定操作方法、使用设备、工具和技术等要求。

⑩ 检验和试验

为了使产品满足规定的要求，应建立并保持进行检验和试验活动的程序，并形成文件，包括：进货的检验和试验、过程的检验和试验、最终检验和试验、对检验和试验记录的要求。

⑪ 检验、测量和试验设备的控制

对用于证实产品符合要求的检验、测量和试验设备建立并保持控制、校准和维修的程序，并形成文件。确认测量任务及所要求的精度，选择合适的设备。应规定检验、测量和试验设备的采购、验收、定期校验、故障维修等控制程序。对上述校验、维修等记录需进行管理。

⑫ 检验和试验状态

对产品的不同状态，如未检、已检合格、已检不合格等，应严格区分，防止不合格的材料、半成品、部件混入或误用，并应明确标识。

⑬ 不合格品的控制

建立和保持对不合格品的控制程序，并形成文件，包括对不合格品的标识、记录、评审、隔离和处置等。

⑭ 纠正和预防措施

(i) 为消除实际已出现的不合格品，及其产生根源，应建立并保持相应控制程序，并形成文件。

(ii) 纠正措施：有效处理顾客意见和产品不合格报告；调查与产品、过程和质量体系有关的不合格产生原因，并记录调查结果；确定消除不合格根源所需的纠正措施，并保证其执行与有效性。

(iii) 预防措施：利用适当信息源，已发现、分析并消除不合格的潜在因素；确保所采取措施的信息提交管理评审。

⑮ 搬运、储存、包装、防护和交付

(i) 应建立搬运、储存、包装、防护和交付的控制程序，并形成文件。

(ii) 提供防止产品损坏或变质的搬运方法。

(iii) 使用指定的储存场地，规定接收和发放的管理方法。

(iv) 对装箱、包装和标志过程(包括材料)等进行必要的控制。采取适当的隔离和防护措施。

(v) 上述保护在合同要求下，应可以延续到交付的目的地。

⑯ 质量记录控制

应建立并保持对质量记录的标识、收集、编目、查阅、归档、储存、保管和处理的程序，并形成文件。

⑰ 内部质量审核

为验证质量活动和有关结果是否符合计划安排，并确定质量体系的有效性，应对内部质量审核工作建立和保持程序，并形成文件。

⑱ 培训

对所有与质量相关的人员进行培训，明确培训要求并建立程序。

在确定培训需求时，要考虑：软件产品开发和管理工具、技术、方法；特定领域知识和技能。

⑲ 服务

在规定由服务要求的情况下，应建立并保持有关服务的实施、验证和报告的程序，并形成文件。一般的顾客支持，在 ISO9000-2 中描述。软件产品维护，通常分为以下几类：问题解决、接口修改、功能扩展或性能改进。如果顾客要求在初始较符合安装之后，对软件产品进行维护，建议在合同中加以规定。建议供方建立并维护形成文件的程序实施维护活动，并且验证这些活动符合规定维护要求。维护活动也可以是对开发环境、工具和文档的维护。应在合同中说明需维护的软件和维护期限。所有维护活动应按照供方和顾客事先确定并协商一致的维护计划或规程实施和管理。对维护活动应加以记录并保存，供方和顾客协商建立维护报告提交规则。

⑳ 统计技术

建立并保持为分析过程能力和产品特性所采用的若干统计技术的实施程序，并形成文件。

7.3 能力成熟度模型(CMM)简介

1. 背景

自电子数字计算机问世以来，计算机软件的开发一直是广泛应用计算机的瓶颈。虽然经过几十年的努力，研究了一些新的开发方法和技术，对提高计算机软件的生产率和质量起到了很大的作用，但问题并没得到彻底解决。

在 80 年代中期，美国工业界和政府部门开始认识到，在软件开发中，关键的问题在于软件开发组织不能很好地定义和管理其软件过程，从而使一些好的开发方法和技术都起不到所期望的作用。在无纪律的、混乱的软件项目开发状态中，开发组织不可能从软件工程的研究成果，即较好的软件方法和工具中获益，致使很多软件开发组织的项目经常严重滞后、经费预算超额。尽管仍有一些软件开发组织能够开发出个别优秀软件，但其成功往往归功于软件开发组的一些杰出个人或小组的努力，而并不是通过成功的软件过程。

历史的经验表明，一个软件开发组织，只有通过建立全组织的有效的软件过程，采用严格的软件工程方法和管理，并且坚持不懈地付诸实践，才能取得全组织的软件过程能力的不断改进。

针对这一问题，1986 年 11 月，美国卡内基-梅隆大学软件工程研究所(SEI)基于 20 世纪 30 年代 Walter Shewad 发表的统计质量控制原理，开始开发过程成熟度框架。1987 年 9 月，SEI 发布了过程成熟度框架的简要描述和成熟度调查表。1991 年，SEI 将过程成熟度框架演化为 CMM 1.0 版：CMU/SEI-91-TR-25。1993 年，SEI 根据反馈，提出 CMM 1.1 版：CMU/SEI-93-TR-25。目前，在政府和工业部门的帮助下，SEI 进一步扩展和精炼了该模型，已经提出 CMM 2.0 版，其中采纳了 ISO/IEC 的软件过程评估标准 SPICE 的一些方法和内容。

2. 基本概念

为了方便叙述和理解，首先给出一些基本概念。

软件过程能力：描述(开发组织或项目组)通过遵循其软件过程能够实现预期结果的程度。一个软件开发组织或项目组的软件过程能力，提供了一种预测该组织承担下一个软件项目可能结果的方法。

软件过程性能：表示(开发组织或项目组)遵循其软件过程所得到的实际结果。可见，软件

过程性能描述已得到的实际结果，而软件过程能力则描述最可能的预期结果。由于一个特定软件项目的具体属性和执行该项目的环境所限，该项目实际的过程性能可能并不充分反映其所在组织的整个过程能力。

软件过程成熟度：一个特定软件过程被明确和有效地定义、管理、测量和控制的程度。它可以指明一个软件开发组织软件过程能力的增长潜力，并且表明一个开发组织软件过程的丰富多样性，以及在各开发项目中运用软件过程的一致性。软件过程成熟度意味着：由于开发组织通过运用软件过程，使各项目执行软件过程的纪律性一致地增强，从而导致软件生产率和质量得到不断的改进。

随着一个软件开发组织的软件过程成熟度的提高，并通过该组织的方针、标准和组织机构等将其软件过程规范化和具体化，从而使得开发组织明确定义的有关管理和工程的方法、实践和规程等，在现有人员离去后仍能继续下去。

软件能力成熟度等级：软件开发组织在走向成熟的过程中，几个具有明确定义的、可以表征其软件过程能力成熟程度的“平台”。每一个成熟度等级为达到下一个等级提供了一个基础。每一等级包含一组过程目标，当一个软件开发组织达到其中一个目标时，则表明软件过程的一个(或几个)重要成分得到了实现，从而导致该组织软件过程能力的增长。

关键过程域：互相关联的若干软件实践活动和有关基础设施的集合，称为一个过程域。每个软件过程成熟度等级包含若干个对该成熟度等级至关重要的过程域，它们的实施对达到该成熟度等级的目标起保证作用，这些过程域被称为该成熟度等级的关键过程域。

关键实践：对关键过程域的实施起关键作用的方针、规程、措施、活动以及相关的基础设施的建立。关键实践一般只描述“做什么”，而不强制规定“如何做”。关键过程域的目标是通过其包含的关键实践的实施来达到的。整个软件过程的改进是基于许多小的、进化的步骤，而不是通过一次革命性的创新而实现的[Imai 86]。这些小的进化步骤就是通过一些关键实践来实现的。

3. CMM 的级别及其行为特征

一个组织的软件过程成熟度，指明了该组织有效地定义、管理、测量和控制软件过程的程度。

在不成熟的软件开发组织中，软件过程没有一个明确的、稳定的定义；并且，实施软件过程的方式往往是反应式的，即遇到问题以后才不得不作出反应，随时解决开发中出现的问题。特别地，在这样的软件开发组织中，没有判断产品质量的客观基础，也没有解决产品或过程问题的客观基础。因此，当项目进度滞后时，常常不得不缩短评审和测试等活动，或取消这些有益提高软件质量的活动。

可是，在一个成熟的软件开发组织中，各项目组通常都能一致地遵循一个有纪律的过程，并具有管理软件开发和维护过程的能力。特别地，在判断产品质量和分析产品及过程问题方面有客观的、定量的依据。因此，在产品的成本、进度、功能和质量等方面，通常都能达到预期的结果。

从一个不成熟的软件开发组织成长为一个成熟的软件开发组织，必须为之设计一条进化途径，并通过这一途径，使之软件过程成熟度按阶段逐步提高。过程成熟度框架就是描述了一条从无序的、混乱的过程达到成熟的、有纪律的软件过程的进化途径。在这一过程成熟度框架中，把软件过程、软件过程能力、软件过程性能和软件过程成熟度等概念集为一体。从软件过

程成熟度框架导出的改进策略，对软件过程的不断改进的历程提供了一份导引图。它指导软件开发组织不断识别出其软件过程的缺陷，引导开发组织或项目组在各个平台上“做什么”改进，但它并不提供“如何做”的具体措施。

软件过程成熟度框架的基础是软件能力成熟度模型。软件能力成熟度模型是为了指导软件开发组织，通过确定当前过程的成熟度，并识别出执行软件过程的薄弱环节，通过解决对软件质量和过程改进至关重要的几个问题来形成对其过程的改进策略；通过关注并认真实施一组有限的关键实践活动，稳步地改善其全组织的软件过程，使全组织的软件过程能力持续增长。

软件能力成熟度模型把软件过程从无序到有序的进化过程分成几个阶段，并将这些阶段排序，形成一个逐层提高的平台，使在每个平台上的改进能为达到下一个平台奠定基础。

CMM 是一个五级模型，如图 7.3 所示。

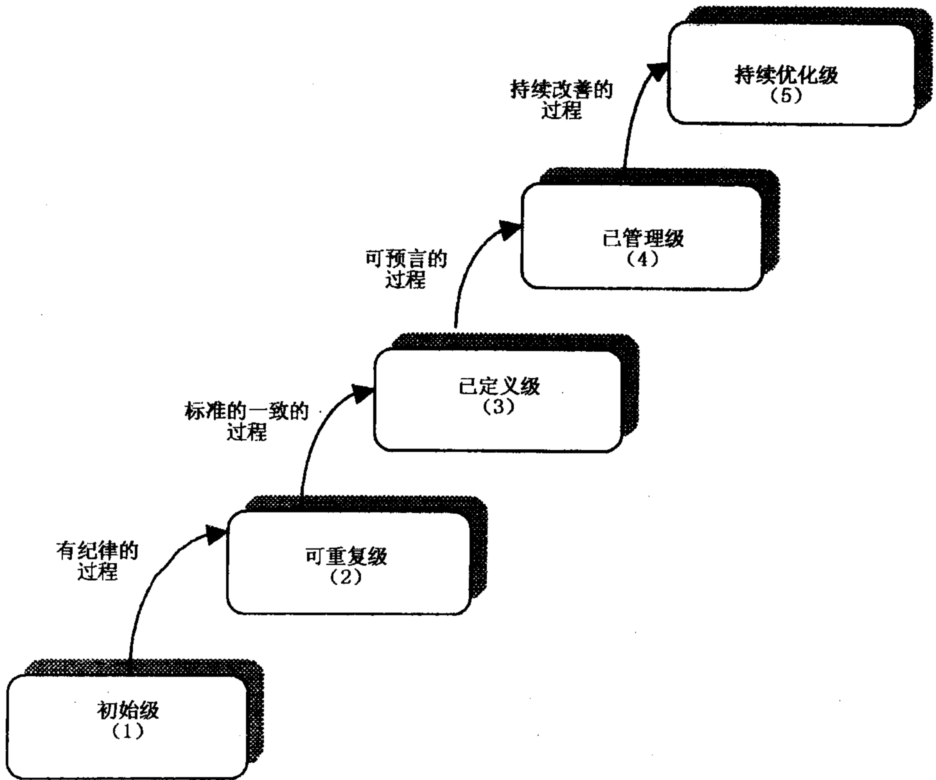


图 7.3 CMM 的五级模型

下面给出五个成熟度等级的特性。

(1) 初始级(1 级)

初始级(1 级)的过程特征与过程能力可简单地概括为：

① 软件开发无规范；

② 软件过程不确定、无计划、无秩序；

③ 过程执行不“透明”；

④ 需求和进度失控；

⑤ 过程能力不可预测。

因此，项目的成败完全取决于个人的能力和努力；软件性能随个人具有的技能、知识和动机的不同而变化，并只能通过个人的能力进行预测。

(2) 可重复级(2 级)

在可重复级(2 级)，由于实现了关键过程域：软件配置管理、软件质量保证、软件子合同管理、软件项目跟踪和监督、软件项目规划以及需求管理(参见图 7.5)，因此该级的过程特征和过程能力可简单地概括为：

① 是可重复的，即能重复以前的成功实践，尽管在具体过程中可能有所不同。这是该级的一个显著特征。

② 是基本可控的，即对软件项目的管理过程是制度化的。具体地说，对软件需求和为实现需求所开发的软件产品建立了基线：为管理、跟踪其软件项目的成本、进度和功能提供了规范；在项目的策划和服务过程中规定并设置了监测点；还提供了当不满足约定时的识别方法和纠偏措施。从而，软件项目过程基本上是可视的。

③ 过程是有效的，即对项目而言，过程可基本特征化为实用的、已文档化的、已实施的、已培训的、已测量的和能改进的。

④ 项目是稳定的，即对新项目的策划和管理是基于以往类似成功项目的经验作出的，并有明确的管理方针和确定的标准。如果有分承制方的话，也将本组织成功的经验应用于分承制方。从而可使项目的进展稳定。

⑤ 是有纪律的，即对所建立和实施的方针、规程，对软件项目过程而言，已进化为组织的行为，从而使组织对给定的软件过程能保证准确地执行。

(3) 已定义级(3 级)

在已定义级(3 级)，由于已实现了可重复级(2 级)的关键过程域：软件配置管理、软件质量保证、软件子合同管理、软件项目跟踪和监督、软件项目规划以及需求管理，并且还实现了关键过程域：组织过程焦点、组织过程定义、培训大纲、集成软件管理、软件产品工程、组间协调以及同行评审，因此该级的过程特征和过程能力可简单地概括为：

① 建立了“组织的标准软件过程”，即关注的焦点转向组织的体系和管理。全组织建立了软件开发和维护的标准过程，软件工程过程和软件管理过程，被综合为一个有机的整体，并且已经文档化。

② 建立了负责组织的软件过程活动的机构，即在软件开发组织中，存在负责软件过程活动的机构，具体实施全组织的过程制定、维护和改进活动。其中包括组织并实施全组织的人员培训，使全体成员具备必须的技能和知识，使他们能有效和高效地履行其职责。

③ 项目定义的软件过程，即项目能够依据其环境和需求等实际情况，通过剪裁组织的标准软件过程，使用组织的过程财富，建立项目自定义的软件过程。其中，自定义的过程允许有一定的自由度，但任务间的不匹配情况，应在软件过程的策划阶段就能得到识别，进行组间协调和控制管理，并与其他工程组建立积极、和谐的工作环境，使项目能更高效地满足顾客的需要。

④ 组织可视项目的进展，即项目定义的软件过程将开发活动和管理活功综合为一个协调的、妥善定义的软件过程，并明确规定了每一活动的输入、输出、标准、规程和验证判据，因此，

管理者或软件项目负责人能够洞察所有项目的技术进展、费用和进度。

⑤ 组织的软件能力均衡、一致，即整个组织范围内的软件开发和维护过程已经标准化，软件工程技术活动和软件管理活动都实现文档化的规范管理，组织和项目的软件过程都是稳定和可重复的。这种过程能力是建立在整个组织范围内对已定义过程中的活动、作用和职责的共同理解基础之上。因此，在整个组织范围内软件能力是均衡、一致的。

(4) 已管理级(4 级)

在已管理级(4 级)，由于又实现了关键过程域：定量过程管理和软件质量管理，因此该级的过程特征和过程能力可简单地概括为：

① 设置了定量的质量目标，即组织对软件产品和过程设置了定量的质量目标，软件过程具有明确定义和一致的测量方法与手段，从而使定量地评价项目的软件过程和产品质量成为可能。

② 项目产品质量和过程是受控和稳定的，即通过将项目的过程性能变化限制在一个定量的、可接受的范围之内，从而使产品质量和过程是受控和稳定的。

③ 开发新领域软件的风险是可定量估计的，即由于组织的软件过程能力是已知的，从而可以利用全组织的软件过程数据库，分析并定量地估计出开发新领域软件的风险。

④ 组织的软件过程能力是可定量预测的，即过程是经测量的并能在可预测的范围内运行，一旦发现过程和产品质量偏离所限制的范围时，能够立即采取措施予以纠正。

(5) 持续优化级(5 级)

在优化级(5 级)，由于又实现了关键过程域：缺陷预防、技术变化管理和过程变化管理，因此该级的过程特征和过程能力可简单地概括为：

① 过程不断改进，即整个组织注重不断地进行过程改进。组织有办法识别出过程的弱点，并及时地予以克服；能够利用关于软件过程有效性的数据，识别最佳软件工程实践的技术创新，并推广到整个组织。

② 缺陷能有效预防，即软件项目组能分析并确定缺陷的发生原因，认真评价软件过程，以防止同类缺陷再现，并且能将经验告知其他项目组。

③ 组织的过程能力不断提高，即组织既能在现有过程的基础上以渐进的方式，又能以技术创新等手段，不断努力地改善过程性能。

特别值得注意的是，在 CCM 定义的五个成熟度级别中，每一等级形成了一个必要的基础，从此基础出发才能达到下一个等级。因此，软件能力成熟度等级的提高是一个循序渐进的过程。

4. *成熟度等级的内部结构*

CMM 的每个等级是通过三个层次加以定义的。这三个层次分别是关键过程域、关键实践类和关键实践。每个等级由几个关键过程域组成，它们共同形成一定的过程能力。每个关键过程域又按四个关键实践类加以组织，而每个关键过程域都有一些特定的目标，通过相应的关键实践类来实现这些目标。每个关键实践类规定了相应部门或有关责任者应实施的一些关键实践，当关键过程域的这些关键实践都得到实施时，就能够实现该关键过程域的目标，其中，达到这些目标，所实施的关键实践可以有所差别。如图 7.4 所示。

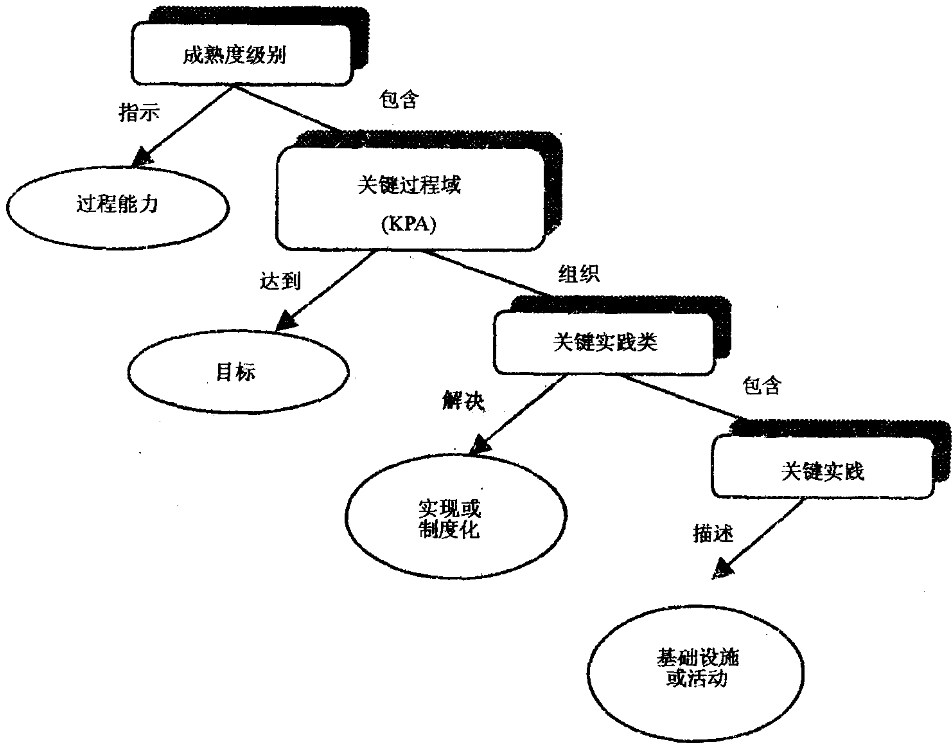


图 7.4　成熟度等级的内部结构

其中:

(1) 成熟度等级

一个成熟度等级是向成熟的软件过程目标进化中的一个平台。如图 7.4 所示,每个成熟度等级指示软件能力的一个等级。例如,在等级 2 上,通过建立健全的项目管理控制,组织的软件能力已经从基本级提高到可重复级。

(2) 关键过程域

每个成熟度等级均含几个关键过程域,指明要达到该等级必须实施这些关键过程域的关键实践。

每个关键过程域只与特定的成熟度等级直接相关,它指明一组相关的活动,当这些活动全部完成时,就能实现对增强过程能力至关重要的目标。仅当一个关键过程区域的全部目标均已达到时,该关键过程域才能实现。对于一个组织来说,仅当其所有项目均已达到一个关键过程域的目标时,才可以说,该组织已使以该关键过程域为特征的软件过程(软件能力)规范化了。

为了达到一个成熟度等级,必须实现该等级上的全部关键过程域。从一级到五级各个等级的关键过程域如图 7.5 所示。

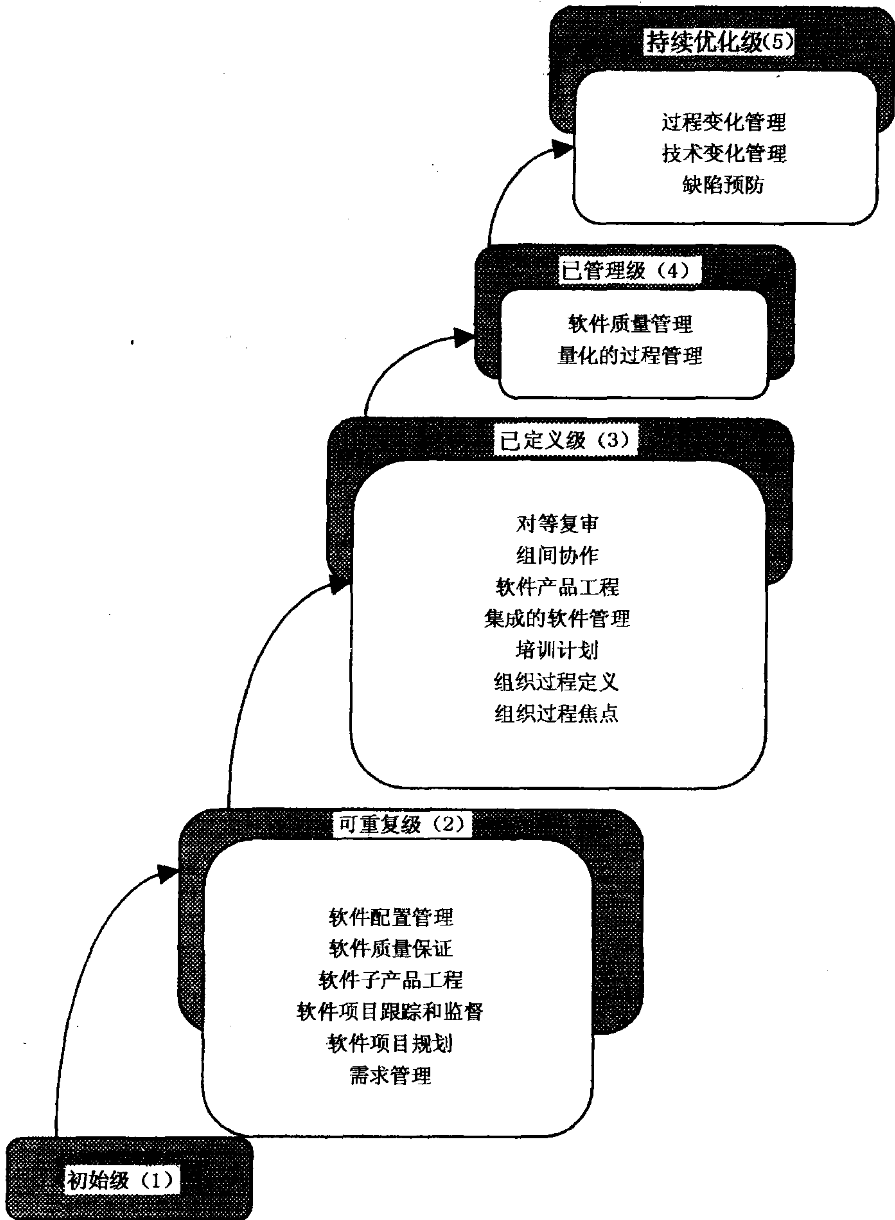


图 7.5　各级的关键过程域

目标概括一个关键过程域的关键实践，表明该关键过程域的范围、边界和意图，可用来确定某组织或某项目是否已有效地实施该关键过程域。

一个成熟度等级的关键过程域表示，这些关键过程域的完全实现是达到该成熟度等级的必要条件；而一个关键过程域的关键实践表示，实施这些关键实践是实现该关键过程域目标的必要条件。

其中：

① 软件需求管理　目的是在顾客和软件项目之间建立对顾客需求的共同理解，顾客需求将由软件项目处理。与顾客的协议是软件项目规划和管理的基础，与顾客关系的控制应遵循有效的更改控制规程（参照软件版本控制中的描述）。

② 软件项目规划　目的是制定进行软件工程和管理软件项目的合理的计划。这些计划是管理软件项目的必要基础，促进按选定的软件生存周期模型分阶段分工进行软件开发。按阶段组织检查，实施控制。没有切合实际的计划不可能实施有效的项目管理。

③ 软件项目跟踪和监督　目的是对实际进展建立适当的可视性，使管理者在软件项目实施情况（即性能）显著偏离软件计划时能及时采取有效措施。

④ 软件子合同管理　目的是选择合格的软件分承制方，并有效地管理它们。把用于基本管理控制的软件项目规划、软件项目跟踪和监督、软件质量保证和软件配置管理等关键过程域和基本级的关键过程域中的关键实践要求，全面适当地用于对分承制方实施管理。

⑤ 软件质量保证　目的是向管理者提供对于软件项目正在采用的过程和正在构造的产品的适当的可视性。软件质量保证是绝大多数软件工程过程和软件管理过程的不可缺少的部分。

⑥ 软件配置管理　目的是在项目的整个软件生存周期中建立和维护软件产品的完整性。软件配置管理是绝大多数软件工程过程和管理过程的不可缺少的部分。

⑦ 组织过程焦点　目的是规定组织在有关改进其整体软件能力的软件过程活动方面的职责。组织过程焦点活动的主要结果是一组软件过程成功实践，这是组织的财富，这些成功实践在组织过程定义中加以描述。正如集成软件管理中所描述的，这些成功实践供各有关软件项目使用。

⑧ 组织过程定义　目的是开发和保持一组便于使用的软件过程的成功实践，这些成功实践能改进各有关项目的过程性能，并为组织能获得长期累积效益奠定基础。这些成功实践提供一组稳定的基本原则，通过诸如培训等机制就能使其成为制度，培训在培训计划中加以描述。

⑨ 培训计划　目的是培育个人的技能和知识，使他们能高效率地执行其任务。尽管培训是组织的责任，但是软件项目应该确定其所需要的技能，当项目有独特的需求时，该项目应提供所需的培训。

⑩ 集成的软件管理　目的是将软件工程活动和软件管理活动集成为一个协调的、已定义的软件过程，该过程是通过剪裁组织的标准软件过程和组织过程定义中所描述的相关过程的成功实践而得到的。

⑪ 软件产品工程　目的是一致执行妥善定义的工程过程。为了能高效地生产正确、一致的软件产品，该工程过程集成全部软件工程活动。软件产品工程描述项目的技术活动，例如，需求分析、设计、编码和测试。

⑫ 组间协调　目的是为软件工程组积极参与其他工程组的工作制定一种方法，使项目能更高效地满足顾客的需求。组间协调是集成软件管理的一个涉及多学科的方面，它延伸到软件工程之外；不仅应该集成软件过程，而且软件工程组和其他组之间的相互作用也必须加以协调和控制。

⑬ 对等评审　目的是及早且高效地消除软件工作产品中的缺陷。其实施过程中一个重要的必然结果是增强对软件工作产品和可预防缺陷的了解。同行评审是一种重要且有效的工

程方法，在软件产品工程的过程中采用此方法时，可通过法根式审查(Fagan-style 审查)[Fagan 86]、结构化走查或者一些其他评审方法[Freedman 90]加以实施。

⑭ 量化的过程管理　目的是定量地控制软件项目的过程性能。软件过程性能表示遵循该软件过程所得到的实际结果。焦点是在某个可测且稳定的过程范围内鉴别出变化的特殊原因，并适时改善促使瞬时变化出现的环境。量化的过程管理给组织过程定义、集成软件管理 、组间协调和对等评审的实践附加一个必要的测量计划。

⑮ 软件质量管理　目的是建立对项目的软件产品质量的定量了解和实现特定的质量目标。软件质量管理对软件产品工程中所描述的软件工作产品实施必要的测量计划。

等级 5 上的关键过程区域包括为了实施持续不断的软件过程改进，组织和项目都必须实施的关键实践。下面列出等级 5 的每个关键过程域的描述：

⑯ 缺陷预防　目的是鉴别缺陷的原因并防止它们再次出现。正如在集成软件管理中所述，软件项目分析缺陷、鉴别其原因并更改项目定义的软件过程。并且按照过程更动管理中所述，应将具有普遍意义的过程更动通知给其他软件项目。

⑰ 技术变化管理　目的是识别出应该采用的新技术(即工具、方法和过程)，并以有序的方式将它引进到组织中去，就像过程变化管理中所述的那样。技术变化管理的关注焦点是在不断变化的环境里高效率地进行创新。

⑱ 过程变化管理　目的是改进软件质量、提高生产率和缩短产品开发周期而持续不断地改进组织中所采用的软件过程。过程变化管理既采用缺陷预防的增量式改进，又采用技术变化管理的创新式改进，并使得整个组织可以享用这些改进。

(3) 关键实践类

每个关键过程域包含如下四个关键实践类：制定方针政策、确保必备条件、实施软件过程和检查实施情况。其中每类都包含若干关键实践，指出各主要负责人(或部门)对该关键过程域的实施和规范化应起的作用和应负的责任。这些关键实践类决定着该关键过程域的实施和规范化是否有效、可重复、能持久。

① 制定方针政策　描述组织的高层管理者(决策者)应起的作用。一般说，应保证过程得以建立并持续有效，为此，应制定组织的方针或策略，规定高层管理者的支持或保障活动。由组织的最高领导及其指定代理负责。

② 确保必备条件　描述项目负责人应起的作用。一般说，应保证解决实施软件过程所必需的先决条件，为此，应建立适当的资源和组织结构，组织必要的培训。通常由项目负责人和计划、人事等部门负责。

③ 实施软件过程　描述软件开发的具体实施者应起的作用。一般说，应制定实施计划和规程，进行实践，跟踪实施，必要时采取纠正措施。通常由软件项目人员和计划、质量等部门负责。

④ 检查实施情况　描述管理者和软件质量保证部门应起的作用。一般说，应保证各项活动按照已建立的过程进行，还应确定过程实施的状态和有效性，为此，应组织适当的测量和分析、评审和审计。由质量部门有关负责人、软件课题负责人和主管领导负责。

由以上描述可知，四个关键实践类之间的关系是：实施软件过程这一关键实践类中的实践描述为了建立过程能力，过程实施者必须做些什么；其他关键实践类中的实践作为一个整体使实施软件过程中的实践规范化。

(4) 关键实践

关键实践描述了为了有效实施并规范化关键过程域,应具备的基础设施和从事的活动。

每个关键实践的描述由两部分组成:前一部分说明关键过程域的基本方针、规程和活动,称为顶层关键实践;后一部分通常是详细描述,可能包括例子,称为子实践。

关键实践描述应该做"什么",而不强制要求应该"如何"实现目标。其他替代的实践也可能实现该关键过程域的目标。要合理地解释关键实践,以便判断关键过程域的目标是否已被有效地实现。

下面,通过给出关键过程域"软件项目规划"的关键实践的描述,其中它们是按四个关键实践类(制定方针政策、确保必备条件、实施软件过程、检查实施情况)进行组织的,来进一步了解关键过程域、关键实践类、关键实践三者之间的关系。

为了保证一致地实现软件项目规划的目标,组织必须建立一个文档化的规程,规定进行软件规模估计的方法。该规程中应详细描述使用以前的规模数据,对假定条件提供文件说明,以及对估计进行评审等准则。这些准则可用来指导对是否遵循合理的规模估计规程作出判断。

软件项目规划的目的是制订进行软件工程和管理软件项目的合理计划,通常主要是指"软件开发计划"。该计划提供完成和管理软件项目活动的必要基础,并按照软件项目的资源、约束和能力,阐述对软件项目的用户所作的承诺。

制订软件项目开发计划的依据是软件需求规格说明和所选定的软件生存周期模型。

软件计划管理的步骤是:估计软件工作产品规模和所需的资源、确定待办的工作、界定软件项目的约束、陈述由需求管理的实践所建立的目标、规模和工作量估计、制订进度表、鉴别并评估软件过程风险,以及协商相应的约定。

为了使软件开发计划能切实有效地指导项目软件工程的各项活动,可能需要反复地执行这些步骤。亦即应根据软件需求的更动和软件过程的各项活动的实际进展情况经常、及时地修订计划。

目标

(1) 软件生存周期已选定,并经评审确认;

(2) 对计划中的软件规模、工作量、成本、风险等已经进行估计;

(3) 软件项目的活动和约定是有计划的;

(4) 影响计划进度的关键路径是已标识的,且受控的;

(5) 影响计划进度的关键资源需求是已标识的;

(6) 文档化的软件开发计划已经正式评审,并确认;

(7) 在软件生存周期的里程碑处,对计划的执行有检查、有记录,对问题有报告;

(8) 对于介入软件开发计划的软件负责人、软件工程师和有关人员进行了软件估计和计划方面的培训。

关键实践

制定方针政策

(1) 指定项目软件负责人,负责协商约定并制订项目的软件开发计划。

(2) 项目遵循书面的、为软件项目制订计划用的方针。

该方针通常规定:

① 根据软件需求规格说明和所选定的软件生存周期模型制订项目软件开发计划。

② 在项目/软件项目/其他软件的负责人之间协商对此项目软件的约定。

③ 与其他工程组(如,系统工程组、硬件工程组、系统测试组)协商介入事宜。

④ 受影响的组(如:软件工程组、系统工程组、系统测试组)评审软件项目的进度、工作量、规模估计、成本估计和其他约定。

⑤ 高层管理者评审所有对组织外部的个人和小组所作的软件项目约定。

⑥ 软件项目开发计划的管理和控制。

确保必备条件

(3) 软件项目有文档化的且经批准的工作说明(SOW)。

① 工作说明包含:

工作的范围;

技术目标和对象;

用户、最终用户或其代表的标识;

要求遵循的标准和规范;

所赋予的职责;

成本和进度的约束及目标;

软件项目和其他组织(如,用户、分承制方、合作伙伴)间的关系;

资源限制和目标;

对开发和维护的其他约束和目标。

② 下列人员评审工作说明:

项目负责人:

项目软件负责人;

其他软件负责人;

相关小组成员。

③ 管理和控制此工作说明。

(4) 赋予制订软件开发计划的职责。

① 项目软件负责人直接或通过委托代表协调软件项目开发计划;

② 以可追踪、可说明的方式把对软件工作产品和活动的职责赋予项目软件负责人;

软件工作产品,如:

适当时,交付给外部顾客或最终用户的产品;

供其他工程组使用的产品;

供软件工程组内部使用的主要产品。

(5) 为软件项目开发计划的制订和实现提供必要的资源和足够的经费。

① 必要时,可以雇用对该软件项目的应用领域有专业知识的、有经验的人来制订软件开发计划;

② 使用合适的计划生成支持工具。例如:

电子表格程序;

计划生成软件。

(6) 介入软件开发计划的软件负责人、软件工程师和有关人员,均已受到软件估计和计划

方面的培训。

实施软件过程

(7) 软件工程组参与下述的项目建议和评审活动:

① 建议、说明的准备、讨论和提交;

② 对软件项目的约定有影响的那些变更的协商;

③ 评审所建议的项目约定的内容。

项目约定,如:

项目的技术目标和对象;

系统和软件的技术解决办法;

软件预算、进度和资源;

软件标准和规程。

(8) 高层管理者按书面的规程评审对组织外部的个人和小组所作的软件项目约定。

(9) 选定有可管理规模的、预定阶段的软件生存周期模型。如:

① 瀑布型;

② 增量型;

③ 渐进型;

④ 螺旋型:

⑤ 逆向工程型。

(10) 标识为控制软件项目所必需的软件工作产品。

(11) 按照一定的规程估计软件工作产品和活动的规模。

该规程通常规定:

① 估计所有主要的软件工作产品和活动的规模。软件规模的度量,如:

功能数;

特征数;

代码行数;

需求数;

文档页数。

应作规模估计的产品和活动,如:

运行软件和支持软件;

可交付的和不交付的产品;

软件和非软件工作产品;

开发、验证和确认产品的活动。

② 分解软件工作产品,达到估计所需的粒度;

③ 可能时,使用历史数据;

④ 规模估计及所作的假定;

(12) 按照书面的规程估计软件项目的工作量和成本。

该规程通常规定:

① 软件项目工作量和成本的估计与对软件工作产品规模及其更动规模的估计有关;

② 将生产率数据(历史的、当前的)用于估计,并将其来源和注释文档化;

可能时，生产率和成本数据尽量取自本组织的项目；

生产率和成本数据应考虑开发软件工作产品的工作量和主要成本（如，直接劳务费、管理费、差旅费和计算机使用成本）。

③ 可能时，应利用过去在类似项目上的经验来估计工作量、人员配备和成本；

确定各种活动的时间段；

确定工作量、人员配备和成本在软件生存周期上的分布。

④ 工作量和成本的估计及所作的假定。

(13) 按照书面的规程估计关键的计算机资源需求。

关键计算机资源可以是在宿主环境中的、在集成与测试环境中的、在目标环境中的或在以上这些环境的任何组合中的。

该规程通常规定：

① 标识项目的关键计算机资源，如：

存储器容量；

处理机能力；

通信信道带宽。

② 关键计算机资源的估计与以下各项的估计有关：

软件工作产品的规模；

处理负荷；

通信量。

③ 关键计算机资源的估计。

(14) 按照书面的规程导出软件项目进度表。

该规程通常规定：

① 软件进度与以下各项有关；

软件工作产品规模及其更动规模的估计；

软件工作量估计。

② 基于过去（或类似项目）经验的软件开发进度表；

③ 开发进度表符合所规定的里程碑日期、关键的相关日期及其他限制；

④ 开发进度表中的活动和里程碑均有合适的间隔，以支持进度测量的精度；

⑤ 将对进度表所做的假定记入文档；

⑥ 把软件开发进度表文档化。

(15) 鉴别、评估与项目成本、资源、进度和技术方面有关的软件风险，并文档化。

① 根据风险对项目的潜在影响，进行风险分析及其优先级排序；

② 鉴别风险的偶发事件（如，进度受阻、人员配置计划更动、设施配置计划更动）。

(16) 制订本软件项目工程设施和支持工具的计划。

① 根据软件工作产品的规模和其他特征估计对这些设施和支持工具的需求。

软件工程设施和支持工具，如：

软件开发用的软硬件环境；

软件测试用的软硬件环境；

目标计算机的环境软件；

其他支持软件。

② 就购买或研制这些设施和支持工具问题，分配职责和商谈约定；

③ 所有受影响的组评审该计划。

(17) 按照书面的规程制订软件项目开发计划。

该规程通常规定：

① 软件开发计划根据并遵守：

用户的标准；

项目的标准；

经批准的工作说明；

分配需求；

所选定的软件生存周期模型。

② 与软件有关组及其他工程组(如，系统工程组、硬件工程组、系统测试组)协商他们介入软件工程组活动的计划，把该项支持工作编入预算，并把协议文档化。

③ 与软件有关组和其他工程组协商软件工程组介入其活动的计划，把该项支持工作编入预算，并把协议文档化。

④ 由下列人员评审软件项目开发计划：

项目负责人；

项目软件负责人；

其他软件负责人；

其他受影响的组的成员。

⑤ 管理和控制软件开发计划。

(18) 把软件项目开发计划文档化。该计划应包含：

① 软件项目的目的、范围、目标和对象；

② 软件生存周期的选择；

③ 选定开发、维护软件用的规程、方法和标准；

④ 确定待开发的软件工作产品；

⑤ 软件工作产品的规模估计；

⑥ 软件项目的工作量估计；

⑦ 软件项目的成本估计；

⑧ 软件项目的风险标识和评估；

⑨ 关键计算机资源的预计需求；

⑩ 确定软件项目的进度，包括里程碑和评审；

⑪ 软件项目工程设施和支持工具的需求计划。

(19) 记录软件策划数据。

① 所记录的信息包括估计和重估计以及评估其合理性所需的有关信息；

② 管理和控制软件策划数据。

检查实施情况

(20) 高层管理者定期(如，里程碑处)评审软件项目计划活动及其实施情况。

① 评审技术、成本、人员设置和进度等的实施情况；

② 分析在低层上未解决的矛盾和问题；
③ 分析软件项目过程的风险；
④ 制定、评审相应的措施，并跟踪到结束。
⑤ 签署评审意见，并履行审批手续。
(21) 项目负责人定期或“事件驱动”地参与评审软件项目的策划活动。
① 受影响的组应有代表参与评审；
② 对照工作说明和软件需求，评审软件项目的策划活动的状态和当前结果；
③ 分析各组间的依赖关系；
④ 分析低层上未解决的矛盾和问题；
⑤ 分析软件项目过程的风险；
⑥ 制定、评审相应的措施，并跟踪到结束；
⑦ 签署评审意见，并履行审批手续；
⑧ 撰写会议记录，并将其分发给受影响的组和个人。
(22) 评审至少应审查：
① 软件估计和策划的活动；
② 评审和制定项目约定的活动；
③ 准备软件开发计划的活动；
④ 准备软件开发计划所用的标准；
⑤ 软件开发计划的内容。
(23) 定期(至少按里程碑)检查软件开发计划的实施情况，并文档化；对于所存在的问题应提出相应的报告。
(24) 进行测量，并用测量结果确定软件策划活动的状态。测量诸如：
① 与计划比较，里程碑的完成情况；
② 与计划比较，软件策划活动中所完成的工作、所用的工作量和经费。

5. CMM 的应用

CMM 有两个基本用途：软件过程评估和软件能力评价。一是软件过程评估：确定一个组织的当前软件过程的状态，确定组织所面临的急需解决的与软件过程有关的问题，进而有步骤地实施软件过程改进，使组织的软件过程能力不断提高。二是软件能力评价：识别合格的能完成软件工作的承制方，或者监控承制方现有软件工程中软件过程的状态，进而提出承制方应改进之处。

软件过程评估集中关注一个组织的软件过程有哪些需改进之处及其轻重缓急。评估组采用 CMM 来指导他们进行调查、分析，并确定优先次序。软件开发组织可用这些调查结果，并与 CMM 中的关键实践所提供的指导相结合，来规划本组织软件过程的改进策略。

软件能力评价集中关注识别一个特定项目在进度要求和预算限制内构造出高质量软件所面临的风险。在采购过程中可以对投标者进行软件能力评价。评价的结果，可用于确定在挑选特定承制方方面的风险；也可对现有的合同进行评价以便监控承制方的过程性能，识别承制方软件过程中潜在的可改进之处。

CMM 为进行软件过程评估和软件能力评价建立一个共同的参考框架。其共同的步骤

是：

第一步：建立一个小组。该小组的成员应是具有丰富软件工程和管理方面知识的专业人员。对该小组进行 CMM 基本概念和评估或评价方法细节方面的培训。

第二步：填写提问单。让待评估或评价单位的代表完成成熟度提问单的填写。

第三步：进行响应分析：评估或评价组对提问单回答进行分析，即对提问的回答进行统计，并确定必须做进一步探查的域。待探查的域与 CMM 的关键过程域相对应。

第四步：进行现场访问。访问被评估或评价单位的现场。评估或评价组根据响应分析的结果，召开座谈会、进行文档复审，以便了解该现场所遵循的软件过程。CMM 中的关键过程域和关键实践对评审或评价组成员在提问、倾听、复审和综合各种信息方面提供指导。在确定现场的关键过程域的实施是否满足相关的关键过程域的目标方面，该组运用专业性的判断。当 CMM 的关键实践与现场的实践间存在明显差异时，该组必须用文件记下对此关键过程域做出判断的理论依据。

第五步：提出调查结果清单。在现场工作阶段结束时，评估或评价组生成一个调查结果清单，明确指出该组织软件过程的强项和弱项。在软件过程评估中，该调查结果清单作为提出过程改进建议的基础；在软件能力评价中调查结果清单作为软件采购单位所作的风险分析的一部分。

第六步：制作关键过程域(KPA)剖面图。评估或评价组制作一份关键过程域剖面图，标出该组织已满足和尚未满足关键过程域目标的域。一个关键过程域可能是已满足要求的，但仍有一些相关的调查发现问题，如果未发现或未指出这些问题，就会妨碍实现该关键过程域的某个目标的主要问题。

总之，软件过程评估和软件能力评价方法均：

(1) 采用成熟度提问单作为现场访问的出发点；

(2) 采用 CMM 作为指导现场调查研究的导引；

(3) 利用 CMM 中的关键过程域生成明确地指出软件过程强项和弱项的调查结果清单；

(4) 在对关键过程域目标满足情况进行分析的基础上，推导出一个剖面；

(5) 根据调查结果清单和关键过程域剖面，向合适的对象提出结论意见。

尽管软件过程评估和软件能力评价有上述相似之处，但软件过程评估或软件能力评价的结果可能不同。一个原因是评估或评价的范围可能变化：首先，必须确定受调查的组织情况。对一个大公司而言，关于“组织”完全可能有几种不同的定义。其范围可以根据有共同的高层管理者、共同的地理位置、统一的经济核算、共同的应用领域或者其他考虑来确定。其次，即使在同一个组织中，所选项目的样本也可能影响范围。例如，起初评估可能是对公司中的一个部门进行，那么评估组根据全部门的范围得出其调查结果清单；后来，可能对该部门中的一条产品线进行评价，此时评价组所得出的调查结果清单是根据比较窄的范围。在这些结果之间作比较是不合适的。

软件过程评估和软件能力评价，由于动机、目的、输出和结果均不同，导致在会谈目的、询问的范围、所采集的信息和结果的表示方式上有本质的不同，所采用的详细规程大不相同，培训要求也不一样。

软件过程评估都是在开放、合作的环境中进行的，评估目的在于暴露问题和帮助改进，评估的成功取决于管理者和专业人员两方面对改进组织的支持，一般能够得到较好的支持。评

估过程中提问单是个重要的工具，但更重要的通过各种会谈了解组织的软件过程。评估的结果除了识别组织所面临的软件过程问题外，最有价值的还是明确了改进组织的软件过程的途径，促进进一步行动计划的制订，使全组织关注改进过程，增强执行改进行动计划的动力和热情。

软件能力评价是在更为面向审计的环境中进行。评价的目的与被评组织的利益密切相关，因为评价组的推荐将影响承制方的选择或资金的投入。评价过程的重点放在复审已文档化的审计记录上，这些记录能揭示组织实际执行的软件过程。

6. CMM 认证现状

根据有关资料统计，至 1999 年 10 月，国际上共有 1 万多家公司取得了 CMM 证书(2—5 级)，其中大多数为 CMM 2 级，取得 CMM 3 级的约占 16%，仅 8 家取得 CMM 4 级，6 家取得 CMM 5 级，包括：洛克希德、波音、IBM、美国空军、印度 WIPRO、印度 MOTOROLA。

至 2002 年 1 月末，北京市已有 10 家公司通过 CMM 认证，它们是：联想软件(CMM 3 级，2002 年)、用友(CMM 2 级，2001 年)、东方通(CMM 2 级，2001 年)、北佳(CMM 2 级，2001 年)、方正电脑(CMM 2 级，2001 年)、神州数码(CMM 2 级，2001 年)、中软网络(CMM 2 级，2001 年)、亿阳信通(CMM 2 级 2001 年)、清华鼎鑫(CMM 2 级，2000 年)、摩托罗拉(中国)(CMM 5 级)。

以上情况表明，我国软件产业近几年来在软件质量管理方面已经取得了很大的进步。但也应看到，与软件产业比较发达的国家相比，还有很大的差距，真可谓“任重而道远”。

习 题 七

1. 解释以下术语：

软件过程　质量体系　软件过程能力　软件过程成熟度　关键过程域
软件过程成熟度
软件能力成熟度等级

2. 简单回答以下问题：

(1) 基本过程类和组织过程类包含哪些过程？

(2) 开发过程主要包含哪些活动？

(3) 基本过程、支持过程、组织过程三者之间的关系；

(4) 软件过程性能与软件过程能力之间的区别；

(5) 如何理解“软件过程成熟度框架的基础是软件能力成熟度模型”？

3. 简述：

(1) CMM 的五级和每级的关键过程域；

(2) 简述 CMM 等级的内部结构；

(3) 简述 CMM 1 级和 2 级的特征。

4. 分析 ISO9000 系列标准的组成及各标准之间的关系，以及引入 ISO9000-3 的原因。

第八章　软件开发工具与环境

20 世纪 70 年代以来，计算机辅助软件工程（CASE：Computer-Aided Software Engineering）的研究，受到各国政府和有关 IT 企业的高度重视。例如，美国政府在 1993 年科技计划中，就把 CASE 工具以及相关软件方法学的研究与开发，作为 IT 领域首要项目。应该承认，在这一领域的有关研究，对软件技术的进步和软件产业的发展都产生了很大的影响和作用。直至现在，CASE 思想和有关技术仍然是软件领域，特别是软件业的主流，例如 VB、VC 语言——一个典型的程序设计环境，中间件技术、服务器技术以及多种多样的软件设计、开发工具等。

简单地说，可以把 CASE 理解为：

CASE	=	软件工程	+	自动化工具

这意味着，蕴涵软件工程开发技术和管理技术的软件工具，可以辅助软件系统/产品的开发。

本章内容分为两部分，第一部分对 CASE 进行一般性介绍，包括 CASE 含义、分类以及 CASE 系统。第二部分集中介绍集成化软件开发环境。

8.1　CASE 概述

1．CASE 概念

自 20 世纪 40 年代电子数字计算机出现之后，软件开发一直约束了计算机的广泛应用。为缓解“软件危机”，60 年代末提出了软件工程的概念，要求人们采用“工程”的原则、方法和技术开发、维护和管理软件，从此产生了一门新的学科——软件工程。

制造业、建筑业的发展告诉我们，当采用有力的工具辅助人工劳动时，可以极大地提高劳动生产率，并可有效地改善工作质量。在需求的驱动下，并借鉴其他业界发展的影响，人们开始了计算机辅助软件工程的研究。早在 80 年代初，就涌现出许多支持软件开发的软件系统。从此，术语 CASE 被软件工程界普遍接受，并作为软件开发自动化支持的代名词。

狭义地说，CASE 是一组工具和方法的集合，可以辅助软件生存周期各个阶段的软件开发；广义地说，CASE 是辅助软件开发的任何计算机技术，其中主要包含两个含义，一是在软件开发和维护过程中提供计算机辅助支持；二是在软件开发和维护过程中引入工程化方法。

从学术研究的角度来讲，CASE 吸收了 CAD（计算机辅助设计）、操作系统、数据库、计算机网络等许多研究领域的原理和技术，把软件开发技术、方法和软件工具等集成为一个统一而一致的框架。可见，CASE 是多年来在软件开发方法、软件开发管理和软件工具等方面研究和发展的产物。

从软件产业的角度来讲，CASE 是种类繁多的软件开发和系统集成的产品与软件工具的集合。其中，软件工具不是对任何软件开发方法的取代，而是对它们的支持，旨在提高软件开发效率，增进软件产品的质量。

20 世纪 80 年代后期和 90 年代初期，在市场需求的驱动下，尽管许多 CASE 生产商提供了成百上千种 CASE 产品，但在实际中并没有获得预期的效果。究其原因，主要在于：

(1) Broods 于 1987 年指出，大型软件系统开发的根本问题是，被开发的软件系统/产品的复杂性和开发过程的复杂性。而 CASE 技术所提供的某些支持，可以控制或降低这些复杂性，但最终不能解决复杂性这一根本问题。

(2) 当时的 CASE 产品，虽然可以或多或少地支持软件开发各阶段的开发活动，但它们之间的集成通常是有限的，每种 CASE 产品几乎代表了一个孤立的“自动化岛”，从而限制了 CASE 技术的广泛应用。

(3) CASE 产品，特别是支持软件高层设计的 CASE 产品的开发，通常滞后于软件技术的发展，从而使一些 CASE 产品已不适应当时软件开发的需要。

(4) 一些软件开发组织，由于软件工程文化所限，不能适时地、有计划地对 CASE 技术进行必要的培训，从而极大地降低了 CASE 技术的效用。

2. CASE 工具与分类

(1) CASE 工具

在 CASE 术语尚未广泛使用之前，人们经常使用的是“软件工具”一词。软件工具是用于辅助计算机软件开发、运行、维护、管理、支持等过程中的活动或任务的一类软件。

引导程序、装入程序和编辑程序可以看着是最早使用的软件工具。在汇编语言和高级程序设计语言出现以后，与之相应的汇编程序、解释程序、编译程序、连接程序和排错程序就成了当时用于开发软件的主要工具。20 世纪 60 年代末出现了软件工程以后，支持软件需求分析、设计、编码、测试、维护和管理等活动的各种工具相继产生。进入 80 年代以后，随着交互式图形技术、多媒体技术的发展，出现了用户界面工具(如窗口系统)和各式各样的多媒体软件工具。近年来，由于面向对象方法学的提出和广泛应用，特别是 UML 语言的提出，又出现了一批新的支持工具。

由上可以看出，软件工具的商品化，推动了软件产业的发展；而软件产业的发展，又增强了对软件工具的需求，促进了软件工具技术的研究和软件工具的商品化进程。

综观软件工具的发展，可以总结出以下几方面主要的特点：

① 趋于工具集成：由于软件开发过程本身是由若干个活动组成的，为了更有效地支持软件开发，自然需求软件工具的集成。工具集成意味着把若干个工具或工具片段结合起来，使几个相关的工具可以协同操作。这一趋势有力地促进了集成化软件开发环境的研究。

② 重视用户界面设计：交互式图形技术及高分辨率图形终端的发展，为友好方便的用户图形界面的开发提供了物质基础。通过采用多窗口、图形表示等技术，极大地改善了用户界面的质量，使工具具有更好的可用性。

③ 采用最新理论和技术：许多工具在研制中，均采用了一些最新的技术，如交互图形技术、人工智能技术、网络技术和形式化技术等，以便提高工具的效用。

(2) CASE 工具分类

随着 CASE 术语的出现，人们不加区分地把“软件工具”和“CASE 工具”等同地予以使用。严格地说，CASE 工具是除操作系统之外的所有软件工具的总称。

关于对 CASE 工具的分类，可以依据不同的分类模式。1993 年，Fuggetta 依据 CASE 工具对软件过程的支持范围，将其分为三类，如图 8.1 所示。

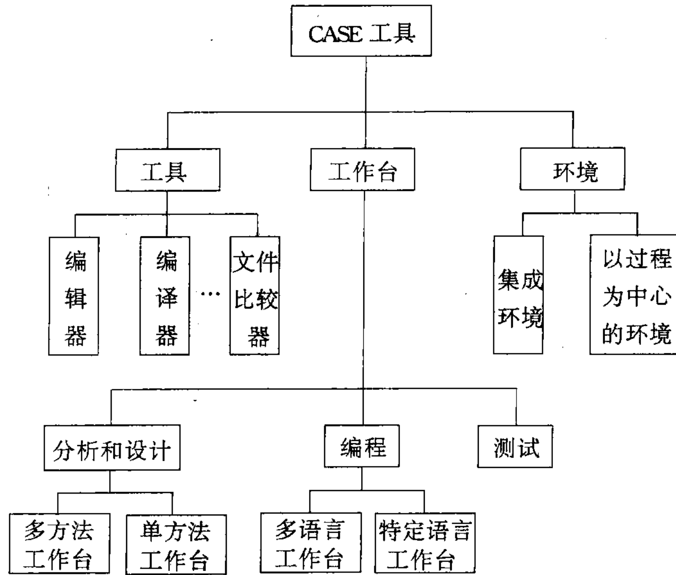


图 8.1　CASE 工具分类:工具、工作台和环境

其中:

① 工具:支持单个任务,例如:检查设计的一致性,编译一个程序,比较测试结果等。

② 工作台:支持某一软件过程或一个过程中的某些活动,例如:需求规约、软件设计、软件测试等。工作台一般以或多或少的集成度,由若干个工具组成。

③ 环境:支持某些软件过程以及相关的大部分活动。环境一般以特定的方式,集成了若干个工作台。其中,环境主要分为集成化环境和过程驱动的环境。集成化环境对数据集成、控制集成、表示集成等提供基本支持;而以过程为中心的环境通过过程模型和过程引擎,为软件开发人员的开发活动提供必要的指导。

显然,工具、工作台以及集成化环境在其构造中,采用的是一类支持软件开发(如需求规约、软件设计、软件实现和软件测试等)的 CASE 技术。市面上大多数 CASE 产品是基于这类技术的,其中关于系统建模、软件设计和编程的工具是质量、用效最好的 CASE 工具。而以过程驱动的环境在其构造中,采用的是一类支持软件开发过程管理(如过程建模、过程控制等)的 CASE 技术,通常基于这类技术的 CASE 产品包含了前一类 CASE 技术和功能,但至今仍然还不算是成熟的。

另外,还可以按支持的活动对 CASE 工具进行分类。如表 8-1 所示。

表 8-1　基于支持活动的 CASE 工具分类

支持的活动	典型工具
需求分析	数据流图工具
	实体-关系模型工具
	状态转换图工具
	数据字典工具
	面向对象建模工具
概要设计	分析、验证需求定义规约工具

续表

支持的活动	典型工具
详细设计	程序结构图(SC图)设计工具 面向对象设计工具 HIPO图工具 PDL(设计程序语言)工具 PAD(问题分析图)工具 代码转换工具
编码工具	正文编辑程序 语法制导编辑程序 连接程序 符号调试程序 应用生成程序 第四代语言 OO程序设计环境
测试	静态分析程序 动态覆盖率测试程序 测试结果分析程序 测试报告生成程序 测试用例生成程序 测试管理工具
维护与理解	程序结构分析程序 文档分析工具 程序理解工具 源程序→PAD转换工具 源程序→流程图转换工具
配置管理	版本管理工具 变化管理工具

8.2 工　作　台

如前所述，一个工作台是一组工具集，支持像设计、实现或测试等特定的软件开发阶段。将CASE工具组装成一个工作台后工具能协同工作，可提供比单一工具更好的支持。可以实现通用服务程序，这些程序能被其他工具调用。工作台工具能通过共享文件，共享仓库或共享数据结构来集成。

工作台能支持大多数的软件过程活动。其中，支持分析、设计、编程的工作台比支持其他活动的工作台更为成熟。

(1) 程序设计工作台，由支持程序设计的一组工具组成，如将编辑器、编译器和调试器等集成在一个宿主机上构成程序设计工作台供开发人员使用。

(2) 分析和设计工作台，支持软件过程的分析和设计阶段。较为成熟的是支持结构化方法的工作台，现也有支持面向对象方法进行分析和设计的工作台。

(3) 测试工作台，趋于支持特定的应用和组织机构。常具有较好的开放性。

(4) 交叉开发工作台，这些工作台支持在一种机器上开发软件，而在其他别的系统上运行所开发的软件。一个交叉开发工作台中，包括的工具有交叉编译器、目标机模拟器，从宿主机

到目标机上下载软件的通信软件包,以及远程运行的监控程序等。

(5) 配置管理(CM)工作台,这些工作台支持配置管理。如版本管理工具,改变跟踪工具,系统建造(装配)工具等。

(6) 文档工作台,这些工作台支持高质量文档的制作。如字处理器、单面印刷系统,图表图像编辑器,文档浏览器等。

(7) 项目管理工作台,这些工作台支持项目管理活动。如项目规划和质量、开支评估和预算追踪工具。

本节主要介绍程序设计工作台、分析设计工作台以及测试工作台。

1. *程序设计工作台*

程序设计工作台由支持程序开发的一组工具组成。Ivic 在 1977 年介绍的程序设计工作台是第一个 CASE 系统。那时,将编译器、编辑器和调试器这样的软件工具放在一个宿主机上,该机器是专门为程序开发而设计制作的。像 Interlisp 这样的面向语言的编程系统也在 80 年代初被开发出来。

汇编器和编译器将高级语言程序转换成机器代码,它们是程序设计工作台的核心构件。在编译阶段生成的语法和语义信息也能被其他工具使用。这包括程序分析器、程序浏览器和动态分析器。程序分析器指明在哪儿定义和使用变量,程序浏览器显示程序结构,动态分析器创建一个动态程序执行轮廓。帮助程序员查找错误的调试系统也要用到语法树和符号表中的信息。

图 8.2 即为一个程序设计工作台的结构。

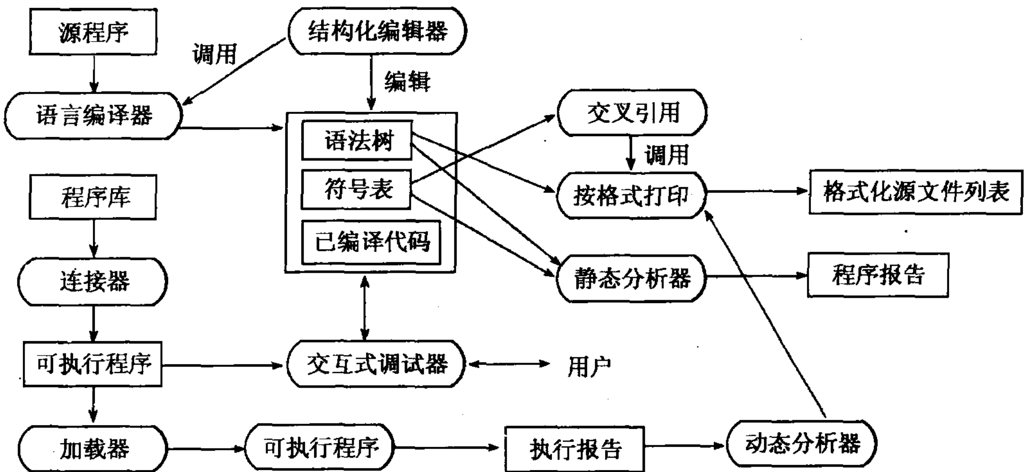


图 8.2 程序设计工作台

其中,CASE 工具以圆形框表示,工具的输入输出以矩形框显示。在这个工作台中,工具通过抽象语法树和符号表集成,抽象语法树和符号表代表源语言程序的语法语义信息。

组成程序设计工作台的工具可能为:

(1) 语言编译器:将源代码程序转换成目标码。其间,创建一个抽象语法树(AST)和一个符号表。

(2) 结构化编辑器:结合嵌入的程序设计语言知识,对 AST 中程序的语法表示进行编辑,

而不是程序的源代码文本。

(3) 连接器:将已编译的程序的目标代码模块连接起来。

(4) 加载器:在可执行程序执行之前将之加载到计算机内存。

(5) 交叉引用:产生一个交叉引用列表,显示所有的程序名是在哪里声明和使用的。

(6) 按格式打印:扫描 AST,根据嵌入的格式规则打印源文件程序。

(7) 静态分析器:分析源文件代码,找到诸如未初始化的变量,不能执行到的代码,未调用的函数和过程等异常。

(8) 动态分析器:产生带附注的一个源文件代码列表,附注上标有程序运行时每个语句执行的次数。也许还生成有关程序分支和循环的信息,还统计处理器的使用情况。

(9) 交互式调试器:允许用户来控制程序的执行次序,显示执行期间的程序状态。

图 8.2 表明,该程序设计工作台是利用语法树和符号表作为共享数据来进行工具集成的。逻辑上讲,提供各种各样工具的所有程序设计工作台中都采用这一方法。

个人计算机上已有许多程序设计工作台。由于市场原因,通常这些工作台不作为工作台而作为包括有附加工具的语言编译器来出售。以这种方式出售的语言有 Basic, C,C++,Pascal, Lisp 和 Smalltalk。

这些语言工作台通常包括一个面向语言的编辑器、编译器和调试系统。在执行过程中程序失败时,就初始化编辑器,并将编辑光标定位到导致失败的源程序语句处。而且,打开调试窗口显示失败时的程序状态。

强制式语言,诸如 C,Ada 或 C++通常如图 8.2 所示,通过 AST 和符号表实现集成。第四代语言(4GL)使用另一种方法实现,即通过数据库实现集成,如图 8.3 所示。4GL 实际上是程序设计工作台,因为它们常包括与程序设计语言无关的设施。图 8.3 给出了 4GL 工作台的一般组成。

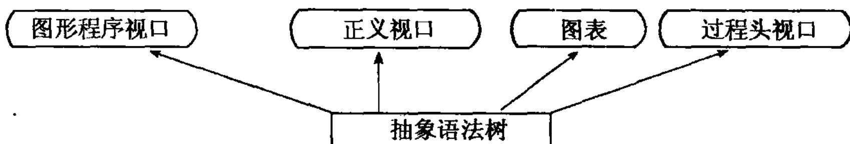


图 8.3 一个 4GL 程序设计工作台

4GL 工作台趋于产生交互式应用程序,该程序从数据库中抽取信息,将之提交给终端机或工作站用户,再随用户所做的改变来更新数据库。用户界面通常由一组标准表格或一个报表组成。

一个 4GL 工作台中可能包括如下工具:

① 诸如 SQL 的数据库查询语言,或是直接输入的,或是从由终端用户填写的表格中自动生成的。

② 一个表格设计工具,用于创建表格,数据通过表格输入和显示。

③ 一个电子报表,用于分析和操纵数字信息。

④ 一个报告生成器,用于定义和创建电子数据库信息的报告。

与强制式编程语言以程序为中心不同,一个 4GL 工作台是以数据库为中心的。数据库(而不是抽象语法树和符号表)是绑定工作台构件的集成“胶水”。一般来讲,用一个 4GL 工作

台产生一个系统，大约只花费传统程序设计语言开发系统的 10%～25%的时间。然而，这样的系统通常比用强制式语言产生的系统低效。因而对大型系统开发来讲，使用 4GL 并不很现实。

2. 分析和设计工作台

分析和设计工作台支持软件的分析和设计阶段，在这一阶段，系统模型业已建立(例如，一个数据库模型，一个实体关系模型等)。这些工作台通常支持结构化方法中所用的图形符号。支持分析和设计的工作台有时称为上游 CASE 工具。它们支持软件开发的早期过程。程序设计工作台则称为下游 CASE 工具。

这些工作台可以支持特定的设计或分析方法，诸如 JSD 或 Booch 的面向对象分析。另外它们也可能是更为通用的图表编辑系统，能处理大多数通用方法的图表类型。面向方法的工作台提供方法规则和指南，也可能进行一些自动图表检查工作。

一个分析和设计工作台可能包括的工具如图 8.4 所示。这些工具一般通过一个共享仓库集成，该仓库的结构是工作台开发商专有的，因而分析和设计工作台通常是封闭式环境。用户很难将他们自己的工具加入其中，也很难修改提供给他们的工具。

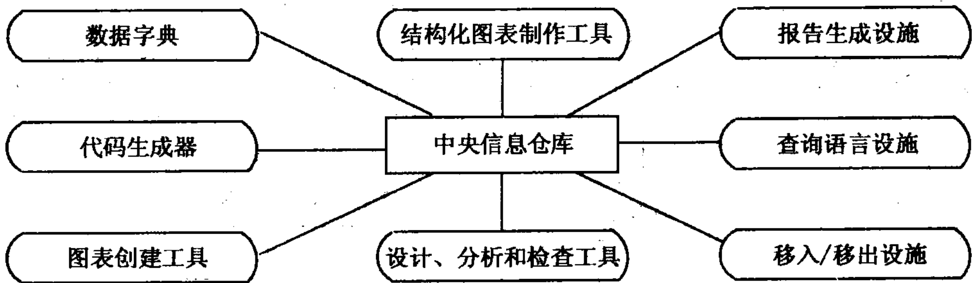


图 8.4　一个分析和设计工作台

图 8.4 中显示的分析和设计工作台的构成为：

(1) 图表编辑器，用来创建数据流图、结构图，实体关系图等。这些编辑器不仅是绘图工具，也能确认图表中出现的实体的类型。它们获取有关这些实体的信息，将这一信息存在中央仓库中(在有些工作台中称为百科字典)。

(2) 设计分析和核实工具，进行分析，并报告错误和异常情况，这些也许和编辑系统集成，以便在早期开发阶段用户能追踪错误。

(3) 仓库查询语言，允许设计者查询仓库，找到与设计相关的信息。

(4) 数据字典，维护系统设计中所用的实体信息。

(5) 报告定义和生成工具，从中央存储器中取得信息并自动生成系统文档。

(6) 移入/移出设施，允许中央仓库和其他软件开发工具互换信息。

(7) 代码生成器，从中央存储器获取设计信息，自动生成代码或代码框架。

分析和设计工作台还有许多不足之处，这大多是由于这些工作台是封闭系统而导致的。它们有自己的存储管理系统。此类缺陷为：

① 移入/移出设施受限。尽管所有的工作台能移入和移出 ASCII 码文本形式的设计结果，大多数也愿提供图表的 Postscript 输出，但不支持其他的移入/移出格式。这在与其他工作

台互换数据时会发生问题。

② 不能裁剪和修改一个设计方法。这样就不能用于特定应用或某类应用。用户通常不可能用自己的规则取代一个原有规则。

③ 工作台自己提供的配置管理系统可能与一个组织机构中使用的系统不兼容。这样，用设计工作台产生系统设计后，无法将设计结果交给组织机构中使用的配置管理和系统管理。

3. 测试工作台

测试是软件开发过程中较为昂贵和费力的阶段。在最早的第一批软件工具中，就开发了测试和调试工具。目前，这些工具提供了各种设施，使用它们可以极大地降低测试和调试阶段的费用。

图 8.5 显示一个测试工作台包含的一些工具，以及这些工具之间的互操作情况。

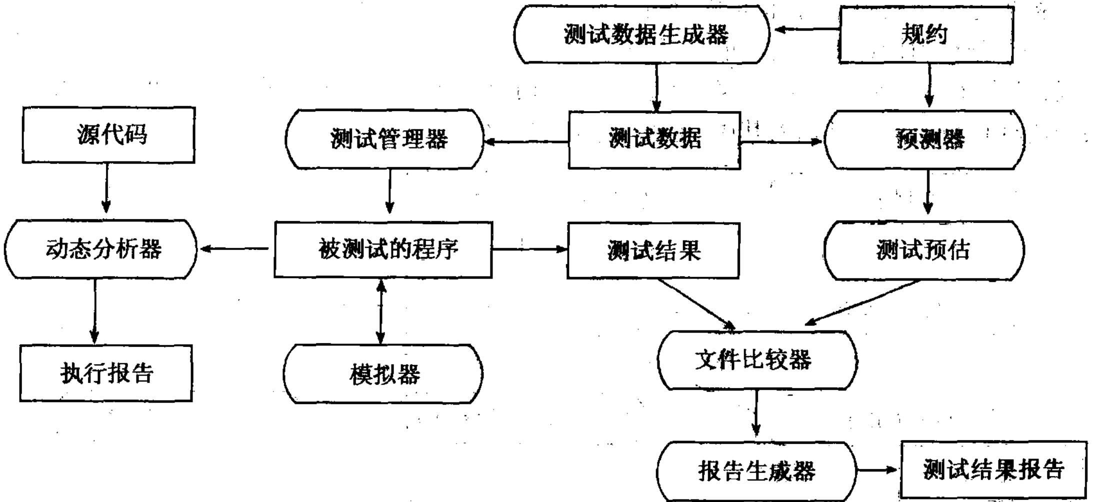


图 8.5 一个测试工作台

测试工作台包括的工具有：

(1) 测试管理器 管理程序测试的运行和测试结果报告。这涉及对测试数据的跟踪，对所期待结果的跟踪，对被测试的程序的跟踪等。

(2) 测试数据生成器 生成被测程序的测试数据。这可能是从一个数据库中选择数据，也可能是使用模式来生成正确格式的随机数据。

(3) 预测器 产生对所期待测试结果的预测。预测器所采用的或是以前的程序版本，或是原型系统。背靠背的测试涉及并行运行预测器和要测的程序，将强调显示它们输出的不同之处。

(4) 报告生成器 提供报告定义，提供测试结果的生成设施。

(5) 文件比较器 比较程序测试的结果和以前测试的结果，报告它们之间的差别。在回归测试中，比较特别有用；回归测试，比较的是新版本和旧版本的执行结果。这些结果中的差异指示了系统的新版本存在的潜在问题。

(6) 动态分析器 将代码加到一个程序中以计算每条语句被执行的次数。运行完测试之后，将生成一条执行轮廓线，显示每条语句被执行的频率。要设计测试用例，使得程序中所有语句都将至少被执行一次。

(7) 模拟器　可能提供各种不同的模拟器。目标模拟器是脚本驱动的程序，模拟多个同时进行的用户交互。I/O 模拟器的使用意味着事务次序时标是可重复再现的。这在测试实用系统时，被测系统如果带有微小的定时错误，这一功能就特别有用。

大型系统的测试需求依赖于要被开发的应用程序。因此，总要更改测试工作台以适应每个系统的测试计划。例如：

① 为测试数据生成器定义模式；

② 为动态分析器定义报告格式；

③ 基于文件测试结果的结构，编写特定目的的文件比较器。

8.3　软件开发环境

1. 软件开发环境及其发展历史

(1) 软件开发环境

软件开发环境(SDE)是支持软件系统/产品开发的软件系统。它由软件工具和环境集成机制构成，前者用于支持软件开发的相关过程、活动和任务，后者为工具的集成和软件的开发、维护及管理提供统一的支持。具体地说：

① 软件开发环境中的软件工具可包括：支持特定过程模型和开发方法的工具，如支持瀑布模型及数据流方法的分析工具、设计工具、编码工具、测试工具、维护工具，支持面向对象方法的 OOA 工具、OOD 工具和 OOP 工具等；独立于模型和方法的工具，如界面辅助生成工具和文档出版工具等；亦可包括管理类工具和针对特定领域的应用类工具。

② 环境中的集成机制按功能可划分为环境信息库、过程控制及消息服务器、环境用户界面三部分。其中：

(i) 环境信息库是软件开发环境的核心，用以储存与系统开发有关的信息并支持信息的交流与共享。库中储存两类信息，一类是开发过程中产生的有关被开发系统的信息，如分析文档、设计文档、测试报告等；另一类环境提供的支持信息，如文档模板、系统配置、过程模型、可复用构件等。

(ii) 过程控制及消息服务器是实现过程集成及控制集成的基础。过程集成是按照具体软件开发过程的要求进行工具的选择与组合，控制集成实现工具之间的通信和协同工作。

(iii) 环境用户界面包括环境总界面和由它实行统一控制的各环境部件及工具的界面。统一的、具有一致视感的用户界面是软件开发环境的重要特征，是充分发挥环境的优越性、高效地使用工具并减轻用户学习负担的保证。

较完善的软件开发环境通常具有如下功能：

① 软件开发的一致性及完整性维护；

② 配置管理及版本控制；

③ 数据的多种表示形式及其在不同形式之间自动转换；

④ 信息的自动检索与更新；

⑤ 项目控制和管理；

⑥ 对方法学的支持等。

(2) 软件开发环境的历史

软件开发环境的发展大体可以划分为三个阶段。

第一阶段:20世纪70年代中期,随着软件工程的兴起,出现了支持程序开发、维护的工具。基于以下的出发点:将文档的建立也作为软件开发过程,而不是在系统开发结束后再补写文档的,以保证文档的可信度。因此,这些工具大都将重点放在建立程序的文档上。其中典型的工具有ISDOS(Information System Design and Optimization System)系统中的PSL/PSA,主要用于系统报表和文档的生成。随后,出现了"工具箱"(Toolkits)概念,即将一些相关的工具,"集中"地放在一起,通常具有统一的用户命令界面,并采用统一的数据交换方式,从而,与单个工具相比,更方便了用户在软件开发中的使用。可以说,工具箱是软件开发环境的"萌芽"。

第二阶段:进入80年代以后,在市场需求的驱动下,有关"环境"的研究已成为学术界和产业界一个热点,主要表现在:

① 在"术语"使用方面,"环境"一词广泛使用,例如"程序设计环境"、"软件开发环境"、"软件开发环境"等。

② 在产品研发方面,支持图形设计方式的软件工具开始大量涌现,包括支持结构化分析和设计的工具,如支持数据流图、模块结构图、状态变迁图等的编辑及分析工具。而且,这一时期的软件工具,具有以下特点:

(i) 支持结构化方法的"自动化";

(ii) 支持单个系统分析员的工作;

(iii) 增进分析效率及确认等能力;

(iv) 支持能力覆盖了部分软件开发过程。

与之同时,出现了以环境信息库为核心的软件开发环境,如McDonnell Douglas于1989年开发的ProKkit * WORKBENCH, IDE(Interactive Development Environments)开发的StP(Software through Pictures)等。这些环境的特点是:

(i) 采用环境信息库,工具围绕信息库实现集成;

(ii) 支持软件开发模型和软件开发方法,如瀑布模型和结构化方法。

③ 在环境基础研究方面,提出了一系列与环境有关的模型,如Buxton提出的支持Ada语言的APSE环境模型、NIST/ECMA提出的SEE基准模型以及Wasserman提出的五级集成模型等,并在集成机制的研究方面取得了很大的发展。

应该说,这一阶段是环境技术发展最迅速、成果最丰富的一个时期。

第三阶段:进入90年代中期以来,随着面向对象方法的广泛应用,并伴随网络应用技术、构件技术以及领域分析技术的发展,开始研制新一代软件开发环境,支持面向对象方法和技术,支持特定领域应用系统开发,支持网上"虚拟软件工厂"。

2. *软件开发环境模型*

(1) APSE 模型

CASE工作台,如前所述,是以不同方式支持软件过程不同阶段的"自动化岛屿"。每个工作台管理它自己的数据,使得很难对所有软件过程输出提供一致的配置管理。在一个工作台生成供另一个不同工作台复用的信息是很难的,或是不可能的。

软件开发环境(SDE)就是要解决这种困难,支持所有或大多数软件过程活动。这样的一个环境概念首先是由Buxton于1980年介绍给公众的。Buxton在美国国防部支持下,提交了一组支持Ada程序设计环境(APSE)的需求。Buxton提出了如图8.6所示的APSE模型。

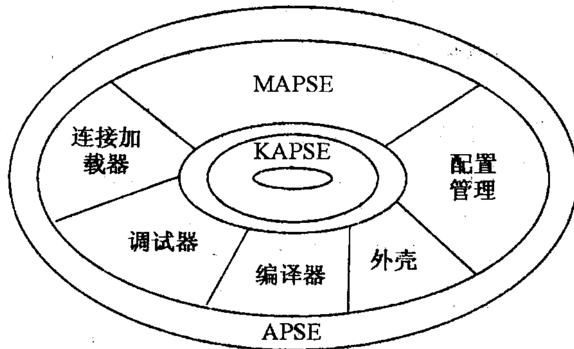


图 8.6 APSE 模型

Buxton 认识到 1980 年不可能建立一个完全的 APSE。他提议采用增量式开发方法来开发基于三个功能级别的环境：

① 一个核心 APSE，它提供环境的基础机构，对操作系统进行扩充。Buxton 建议这种核心 APSE 采用标准化的核心，并且有一个公共工具接口。从而在这一相同的核心之上建造不同的环境产品。

② 一个最小的 APSE，它基本上是一个程序设计工作台。在 1980 年所能提供的硬件平台上（仅带文本终端的分时系统），这可能是 APSE 提案所能现实提供的全部了。

③ 以增量式开发的一个完整的软件开发环境，当支持其他过程活动的工具可提供时，可将这些工具加入到最小的 APSE 以扩充其功能。

从本节有关软件开发环境发展历史的叙述中可以了解到，APSE 模型的出现以及关于 APSE 环境开发的提议，极大地影响了软件工程界对环境的研究。

(2) Wasserman 五级模型

1990 年 Wasserman 讨论软件开发环境的集成时，提出一个五级模型。即：①平台集成；②数据集成；③表示集成；④控制集成；⑤过程集成。

从用户的角度看，集成意指 CASE 工具显示了某种一致的度量。对不同的集成系统，它们的一致性差别很大。松散耦合系统也许只提供了有效的数据互换。相反，紧耦合集成系统的工具在单一、共享的软件表示上操作，使用一个一致的用户界面，从一个工具能平滑过渡到另一个工具（即无缝转移）。

① 平台集成

平台集成是指工具或工作台在相同的平台上运行，其中“平台”或是一个单一的计算机或操作系统或是一个网络系统。

② 数据集成

数据集成是指不同的软件开发环境能够相互交换数据。因而，一个工具的结果可以作为另一个工具的输入。

有许多不同级别的数据集成：

(i) 共享文件：所有工具识别一个单一文件格式。最通用的可共享文件是字符流文件。

(ii) 共享数据结构：工具使用的共享数据结构通常包括有编程和设计信息。事前，所有的工具要认可该数据结构的细节，并把该结构细节“硬件化”到工具中。

(iii) 共享仓库：工具围绕一个对象管理系统来集成，该 OMS 包括一个公有的、共享数据模型来描述能被工具操纵的数据实体和关系。这一模型可为所有工具使用，但不是工具的内在组成部分。

最简单的数据集成形式是基于一个共享文件集的集成，UNIX 系统就是这样。UNIX 有一个简单的文件模型，即非结构化字符流。任何工具都能把信息写入文件中，也能读其他工具生成的文件。UNIX 文件系统把 I/O 设备作为特殊文件来处理。它还提供管道。当进程间通过一个管道联系时，无须中间文件创建，字符流从一个进程直接流向另一个。

文件是一个用于信息交换的物理简单方法。然而，应用程序如果利用文件来访问另一个工具产生的信息时，就必须知道一个文件的逻辑结构。这个逻辑结构嵌入在写这个文件的程序中。或者所有工具依从哪个格式，该格式成为一种标准，或者使用信息的工具必须知道该信息是哪个工具创建的。

一个基于文件的集成策略导致点对点的集成方法。每对工具必须认同文件互换格式，或者由一个过滤程序来完成该共享文件格式从一个表示到另一个表示的转换。参见图 8.7。

图 8.7　基于共享文件的点到点的工具集成

原则上，共享文件集成需要为每一对要集成的工具编写一个转换过滤器，因为不同工具使用的格式可能差别相当大。

另一种方案是，认可文件格式约定，编写工具时参考这些约定。这是 UNIX 系统早期版本使用的方法。UNIX 系统是最早的有效的程序设计环境之一。然而，新的工具也许发现这些被认同的约定限制太多，因而忽视这些约定，使用自己的格式，这些工具很难和系统中已有工具兼容。

采用共享语法和语义信息的数据集成的另一个可选的方法是基于共享数据结构的。这些数据结构也许代表了诸如数据流图、实体关系图或程序设计语言等术语。有许多编程语言转换和支持工具参与这种集成方式。这些工具依赖于对程序的语法语义分析。生成的代码与符号表和语法树相连，因此程序执行信息能用高级语言术语来表示。图 8.8 图示了这种形式的数据集成。

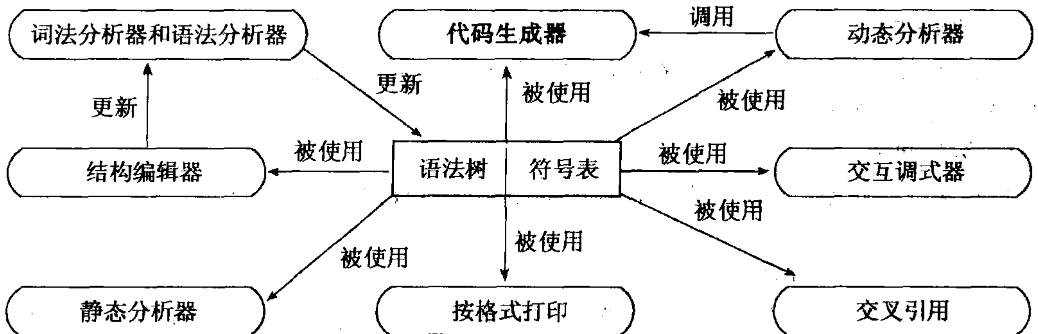


图 8.8　通过共享数据结构的集成

围绕共享数据结构的集成隐藏了工具间的差别，呈现给用户的是一个一致的程序开发系统。然而，很难将工具移到这个环境中，原因在于数据集成表示过于复杂。任何新工具必须知道该共享结构的细节，因此，基于这种形式集成的工作台通常是单独的系统，和其他 CASE 工具集成的惟一可能形式是通过程序设计语言源代码。

围绕一个仓库或对象管理系统的集成是适应性最强的一种数据集成形式。数据管理系统是一个数据库系统，包括输入实体到系统，将属性与这些实体相连，在实体间建立分类关系等设施。

这种集成方式的关键是要设计一个公共的数据库模式。在该数据库模式中定义了实体类型和实体间的关系。工具根据这一模式来读写数据。如果一个工具希望使用另一个工具产生数据，那么将用到该模式来发现那个工具所生成的数据结构(见图 8.9)。

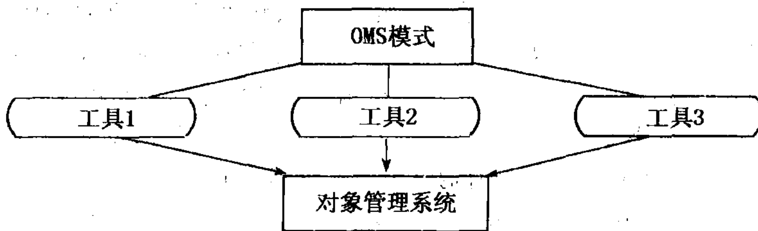


图 8.9　通过 OMS 集成

这种数据集成的方法主要有两个缺点：

- 工具要独立编码以利用 OMS。已有工具若要利用 OMS 就得付出相当的努力。
- 除工具或工作台外，CASE 用户还必须购买 OMS。

③ 表示集成

表示集成或用户界面集成意指一个系统中的工具使用共同的风格，以及采用共同的用户交互标准集。工具有一个相似的外观。

有三种不同级别的表示集成：

(i) 窗口系统集成：这一级集成的工具使用相同的基本窗口系统，窗口有共同的外观，操作窗口的命令也很相似，如每个窗口都有窗口移动、改变大小、图标化等命令。

(ii) 命令集成：这一级集成的工具对相似的功能使用同样格式的命令。如果使用一个文本界面，所有命令都使用相同的命令和参数的语法格式。如果使用一个用菜单和图标的图形界面，相似的命令就会有相同的名字。在每个应用程序中菜单项定位于相同位置。在所有的系统中，对按钮、菜单等使用相同的表示(图标)。

命令集成意指以相同方式来支持应用程序和环境的控制功能。例如，所有应用程序需要一些机制来允许用户终止其执行。而所有的应用程序可能有相同的“quit”按钮。如果工具是文本命令，那么命令应有相似或相同格式的参数名。

如果实现都按照一组指南来定义抽象用户界面操作的表示方式，就能实现命令集成。这些表示可能包括从可选集中作一个选择，拴住一个开关，显示数字或字符信息等。这样的指南的一个较早的例子是 Sommerville 等在 1989 年介绍的 ECLIPSE 环境中给出的有关定义。根据菜单、按钮、显示框、形状的“复合”等来定义用户界面。

(iii) 交互集成：这种集成是针对那些带有一个直接操纵界面的系统，通过该界面用户可

以直接与一个实体的图形或文本视图进行交互。交互集成意指在所有子系统中提供相同的直接操纵操作，如选择、删除等等操作。因而，如果通过双点按选择正文，那可能在一个图表中也以相同方式来选择实体。支持交互集成的系统例子有字处理系统和图形编辑系统。

大多数软件工具都用图形或正文来描绘它们所操纵的对象。不管这些对象是什么，交互集成要求与这些图形或文本对象交互时所使用的机制是一致和统一的。例如，如果正文一般经光标控制键横过正文来选择正文段，那么需要选择正文的所有工具都要使用同样方法。

为交互集成提供指南特别困难。这是因为可能交互的数目、文本和图形对象表示的可能的范围都难以确定。为非结构化文本和未分类的图形对象定义其可能的交互行为相当容易，但当文本或图形代表一个结构化实体，其操作是通过屏幕外观来进行时就困难多了。

1993 年，UNIX 工作站窗口系统集成的事实标准是 X 窗口 Motif 工具箱。实际上所有基于 UNIX 的 CASE 工具都认可这一标准。然而，它只解决了窗口级的集成。来自不同销售商的 UNIX 工具和工作台很少很好地在命令和交互级上集成。这些 CASE 系统也许提供命令剪裁机制来提高表示集成。但是，通常是不支持用户细节操作的交互集成的。

在 PC 平台上，事实上的标准是 Microsoft Windows。这比 X/Motif 好得多，因为它支持三个级别的表示集成。Windows 除具有通常窗口系统所应具有的功能外，还提供工具箱来支持其所制定的菜单构造指南，还支持特定形式交互。实现一个工具界面最好的方法是遵照这些指南，这样，运行了 Windows 的 CASE 工具都有一致的外观。

④ 控制集成

控制集成支持工作台或环境中一个工具对系统中其他工具的访问。除了能启动和停止其他工具外，一个工具能调用系统中另一工具所提供的服务，这些服务可通过一个程序接口访问到。例如，一个综合工具箱中，一个结构化编辑器可以调用一个语法分析器来检查所输入的程序片断的语法。

一些工作台或环境针对控制集成开发了特定机制。然而，一些销售商已采用基于消息传递的更为通用的方法。这已在 Sun 的 ToolTalk 等系统中实现。1995 年 Brown 描述了这种工具集成的方法并比较了消息传递系统的几种实现机制。

在消息传递方法中，CASE 系统通过传递消息来彼此互换信息。这些消息能提供状态信息，通知其他工具正在进行什么，或者请求特别的服务，该服务是由一个 CASE 系统提供的。一个消息服务器管理 CASE 系统间的通信。

图 8.10 说明了这个通用模型。参与集成的每个工具提供一个控制接口，通过控制接口可以访问该工具。消息服务器了解所有可访问工具的接口信息和这些工具的定位。它负责网络传输、编码和解码。

当一个工具需要和另一个工具通信时，它用已知的格式构造一条消息，写上地址，把这个消息发送到消息服务器。工具不必知道被调用的工具在哪儿，它只简单地告诉消息服务器，服务器再将该消息传给被调用的工具。因而，该模型支持分布式 CASE 系统，即系统的不同部件在不同的计算机上运行。

逻辑上，工具的广播消息被对之感兴趣的其他工具接收。实际上，这是一个低效的方法。消息服务器知道每个 CASE 系统能处理的消息，因而只传递合适的消息。为说明这一方法，假设 CASE 系统包括一个设计编辑器，一个代码生成器和一个编译器。这些都是分别实现的。作为一个编辑会话的结果，下列动作可能发生：

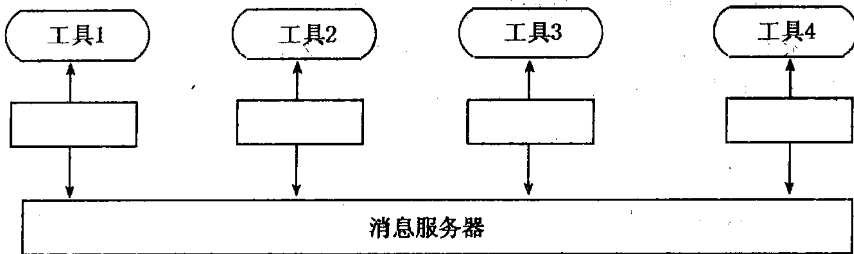


图 8.10　通过消息传递的控制集成

(i) 设计编辑器发给代码生成器一条消息，请求处理设计，并生成代码。

(ii) 生成代码后，代码生成器发送一个能被设计编辑器和编译器两者截获的消息。该设计编辑器将生成的代码文件与设计相连，编译器编译所生成的代码。

(iii) 在代码已经生成后，编译器发送一个消息，被设计编辑器截获，它告之用户编译已完成。

控制集成也需要某种级别的数据集成，以便能互换工具操作的参数。一个操作期间要互换的数据格式通常是用一个接口定义语言(IDL)来编写的。该 IDL 引入了每个工具能使用的一组标准类型。每个工具必须将要互换的数据转换成这些类型，这样，它能被接收工具处理。

但是，这种方法只适合于互换相当短的消息。大块的数据交换还必须通过文件或对象来组织。因此，系统间传递的消息一般包括对存有共享数据的文件的引用。

⑤ 过程集成

过程集成意指 CASE 系统嵌入了关于过程活动、阶段、约束和支持这些活动所需的工具的知识。CASE 系统辅助用户调用相应工具完成有关活动，并检查活动完成后的结果(见图 8.11)。

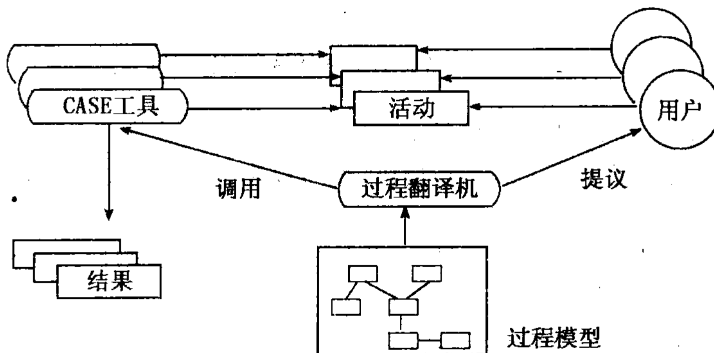


图 8.11　过程集成

过程集成需要 CASE 系统维护软件过程模型，并实例化和实施该软件过程模型。其间，要说明有哪些活动，活动之间的过渡情况如何，完成活动要哪些工具支持，整个开发采用了哪些策略等等。所有这些都在软件过程模型里表示出来，由一个过程翻译器或“引擎”实施该模型，在软件过程驱动下进行软件开发。

许多活动是并发活动，这点必须反映在过程模型中，活动是相互依赖的，因而过程模型应

是动态的。并且随着所获取的与过程活动有关的信息越来越多时,过程模型应可改变。

CASE 技术对过程集成的支持依赖于过程模型的设计。谈到过程建模,要明确以下问题:

(i) 在没有过程支持的环境中开发软件时,也常谈到过程模型,这种情况下谈及的过程模型是类属模型。它们靠人类翻译,以在任何特定环境中实例化该模型。在这些过程模型中未定义活动及其实现的细节。

(ii) 决没有一条简单的捷径来组织软件开发,系统开发时也没有一条捷径让项目管理者和开发人员随意改变过程。如果发生未预料的事情,人们可以很快地在活动间切换(例如,打印机不能打印)。但是在一个模型中嵌入这种随意性很难。

(iii) 过程模型说明软件过程的产品和开发人员之间的通信。对诸如发货处理这种结构良好的任务,可以形式化描述其通信情况。但是,要把软件开发的组织形式、如何解决具体突发问题的处理模式说清楚很困难,更不必说形式化了。

过程集成中存在的较大的阻力可能来自开发人员和经理,他们可能已经习惯用非形式的、文档化的方法进行软件开发和管理,不习惯用环境中提供的严格策略。这样,他们可能不愿意使用支持过程集成的 CASE 环境。

(3) 基于服务的环境层次模型

为了能让软件开发环境可根据不同项目需要来提供不同的支持,软件开发环境必须能容纳广泛的 CASE 工具和工作台。也必须能按需增加新的设施。这意味着一个软件开发环境可认为是一组服务的集合,那些服务能为支持终端用户的设施所用。提供这些服务的既可以是软件开发环境运行其上的平台,也可以是环境框架。图 8.12 显示了这三层模型。

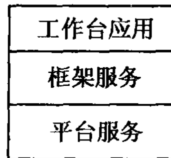


图 8.12　一个软件开发环境的层次模型

在这个模型中,一个软件开发环境是由使用环境服务的一组集成的 CASE 工作台组成的。这些服务既可由环境运行于其上的平台提供,也可以由环境框架提供。框架服务可类比于 APSE 的环境核心。

Brown 等在 1992 年指出,根据环境用户角色的不同,对这一软件开发环境模型有许多不同的看法。其中最重要的有:

● 应用软件开发人员认为环境是支持软件过程的一组工作台集。他们主要关心工作台应用层。

● 软件开发环境集成人员把环境看作一组通用服务和工具的集合,其中的工具是指那些集成在特定背景中的,用以创建高效支持环境的工具。他们主要关注图 8.12 的工作台和框架层之间的边缘部分。

● 工具或工作台开发人员将环境看作一组通用服务的集合,那些服务用于将他们的工具与其他别的工具一起集成到环境中。他们主要关注图 8.12 的中间层。由于他们不知道组成环境的别的工具的情况,因而他们不关心图 8.12 中的工作台应用层。

● 框架开发人员把环境看作一组服务的集合，必须在某些宿主机系统中实现那些服务。他们关心如何使用机器的设施来提供有效的服务实现。他们主要关注图 8.12 的中间和最底层之间的接口。

显然，一个环境提供的服务很重要，这些服务支持工具的实现和集成。下面介绍图 8.12 的服务层。

① 平台服务

一个软件开发环境运行其上的平台称为软件开发环境的宿主机系统。某些情况下，使用软件开发环境开发的软件运行在相同的平台上，但是，有许多情况，所开发的软件将分发到可能具有完全不同的构架和操作系统的一些目标机系统上。图 8.13 说明了这种宿主-目标开发方式，宿主机上所开发的软件分派到不同的目标机上。

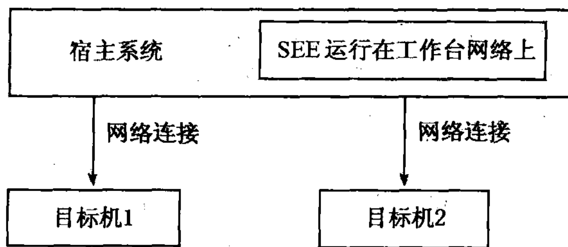


图 8.13 宿主-目标开发方式

采用宿主-目标开发方式的理由如下：

● 某些情况上，正开发的应用软件是为一个没有软件开发机制的机器所用的。其大多数是为专用计算机开发实时系统。这些计算机甚至还没有一个操作系统，只有一个简单的实时执行机制。

● 目标机也许是面向应用的(例如一个向量处理器)，不适于支持软件开发环境。

● 目标机正在使用之中，专门运行一个特定的应用(如事务处理系统)，因此不能用该目标机进行软件开发。

(i) 文件服务：一般来讲，作为软件开发环境宿主机的平台提供如下服务：

支持文件命名、创建、存储、删除，并将文件按目录结构组织起来。正常情况下，文件存储在一个或多个文件服务器上，可为网络上所有机器访问。

(ii) 进程管理服务：用于创建、开启、停止和挂起在网络计算机上运行的进程。

(iii) 网络服务：用于把进程和其相关数据从一台计算机移至另一台计算机。

(iv) 通信服务：用于与本机构其他位置上的其他的计算机通信。如果目标机与网络相连，这一服务必须能支持将程序下载到这些机器上。

(v) 窗口管理服务；允许在用户显示器上创建、移动、删除窗口，改变窗口大小等。

(vi) 打印服务：允许将环境的信息打印到纸上，或打印到其他永久性媒体诸如 CD-ROM 或缩微卡片上。

一般来讲，环境平台是由异构分布式计算机组成的。这也许包括不同类型的计算机(例如，UNIX 工作站和 PC 机)，来自相同制造商运行不同操作系统版本(如 Solaris 2.1 和 Solaris 2.4)的 UNIX 工作站，以及来自不同制造商的 UNIX 工作站。

在拥有许多不同计算机的大型组织中这种异构是不可避免的。新工作站通常只运行最新版本的操作系统。这还得让运行于老版本的操作系统之上的已有系统运行起来。因而，相同操作系统的不同版本可能在相同网络上并发运行。进一步来讲，同一组织机构的不同部分可能有不同的计算需求，因而购有不同类型的机器(如，图形工作台)。显然他们希望使用这些来运行软件开发环境，而不是专为软件开发购买别的计算机。

② 框架服务

软件开发环境中框架服务扩充了环境宿主机提供的服务集，通常框架服务是利用平台服务来实现的。这些服务专用于支持 CASE 工具或工作台的集成。

理解框架服务最好的途径是在一个软件工程构架的参考模型比较易于理解的背景下来理解它们。构架参考模型提供了一个比较不同构架的基础。软件开发环境的参考模型最早由欧洲计算机制造厂商协会(ECMA)提出的，现在已为美国国家标准和技术局(NIST)采用。“正式的”模型在 1997 年度 ECMA 的 NIST/ECMA 报告中给出。但对该模型的描述是由 Brown 等在 1992 年给出的。Brown 用这个模型比较了不同的环境框架提案。

图 8.14 显示了该模型的结构(已以“更粗”模型而闻名)。关键在于，该参考模型确定了一个环境应该提供的五类服务集。它也提供了使用这些服务的工具和工作台的“插入”设施。其结果是，根据所使用的软件过程和开发的软件类型，可以用不同的工具和工作台来配置软件。

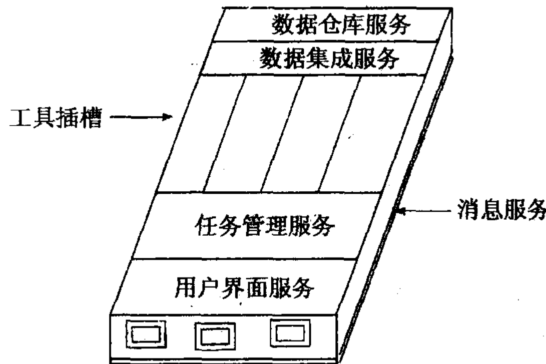


图 8.14 SEE 的参考模型

一个软件开发环境提供的服务可能为：

数据仓库服务　对数据项及其关系进行管理。

数据集成服务　提供了对数据项成组或配置进行管理的设施。它们支持配置命名，支持建立配置间的关系。这些服务和数据仓库服务是环境中数据集成的基础。

任务管理服务　提供了软件过程模型定义和实例化的设施。它们支持过程集成。

消息服务　提供了工具到工具，环境到工具，以及环境到环境通信的设施。它们支持控制集成。

用户界面服务　提供用户界面开发设施。它们支持表示集成。

(i) 数据仓库服务

数据仓库服务提供了命名实体、管理实体、建立实体间关系的基本设施。表 8-2 介绍了参考模型中所确定的数据仓库服务。

表 8-2 数据仓库服务

服务	描述
数据存储	支持实体的创建、读、更新和删除
关系	定义和管理环境实体间的关系
命名	支持实体命名。这是赋予实体的惟一的标识符
定位	在工作站网络上分派实体,因而有像移动、拷贝、重复等相关操作
数据事务	支持原子事务,允许发生失败事件时的数据库恢复
并发	支持多个事务处理同时进行
进程支持	提供诸如开启、停止、挂起等进程操作
文档	支持实体的脱机存储和恢复
备份	发生系统失败事件时,支持数据恢复

这些数据仓库服务通常是由对象管理系统(OMS)提供的。已实现的大多数对象管理系统是基于实体-关系模型的,并提供一些反映软件过程需要的内置实体和关系类型。

图 8.15 显示了在 OMS 中,代表过程或函数的实体是如何相连的。图中假定环境使用一个数据的实体关系模型。实体、属性和关系被分类,而这一类型信息为软件开发环境工具所用。

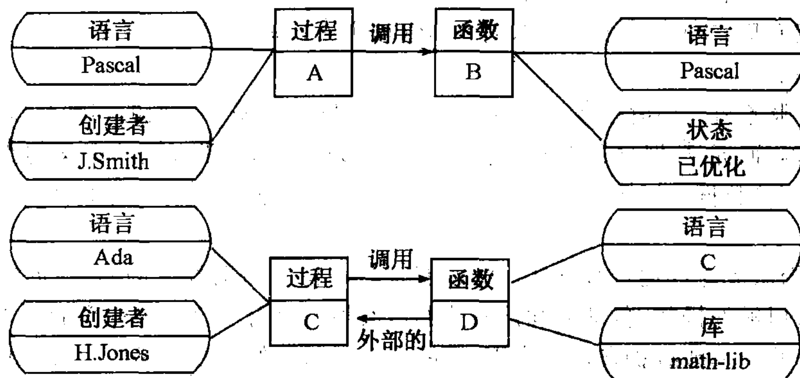


图 8.15 OMS 中分类的实体、属性和关系

在该例中,说明了用不同程序设计语言编写的 4 个例程 A, B, C, D 的情况。每个都有不同的属性,如所使用的开发语言、创建者、其状态和它存储在哪个库中。这只是一个示意图,未显示对象的所有属性。

该模型中显示过程 A 调用函数 B,两者都是用 Pascal 写的,对 B 进行某种程度的优化。过程 C 是用 Ada 写的,调用函数 D。这是一个外部函数,用 C 语言编写,存在称为 math-lib 的库中。可以提供浏览器,来浏览 OMS 中的对象及其属性,以及对象之间的关系。

一个对象管理系统或数据库的粒度取决于能有效存储和操纵的实体的最小尺寸。如果能被有效操纵的最小实体尺寸是文件,那么所管理的关系类型就允许文档被连接。然而,在这些文档中不能建立实体间的关系。相反,如果只有几字节长的细粒度实体能被管理,那就能使用更丰富的关系集。例如,OMS 就记录程序中变量声明间的连接及其使用情况。

检索数据时,细粒度系统比粗粒度系统需要更多的数据库查询。这就意味着这些系统的操作性能要比粗粒度系统差。这样一来,大多数数据仓库提案都推荐使用粗粒度而不是细粒

度数据管理。

(ii) 数据集成服务

框架中提供的数据集成服务集扩展了基本数据仓库服务。数据仓库服务是通用的，而与数据集成服务相关的操作显然是支持软件开发的。表 8-3 描述了软件开发环境参考模型中提议的数据集成服务。

表 8-3 数据集成服务

服务	描述
版本	支持实体多版本管理
配置	将实体分组，按组进行配置命名，并作为一个完整的实体来管理
查询	提供访问和更新版本的服务
元-数据	提供模式定义和管理设施
状态监控	提供触发设施，允许当达到特定数据库状态时，初始化特定操作
子环境	能定义和管理环境的数据和操作的一个子集，并将之当作一个单独的命名的环境
数据互换	支持从环境中移入和移出数据

在一个软件开发环境中，配置管理是非常关键的。广泛使用的是基于文件的配置管理工作台，其缺点是所管的实体只能通过结合在工作台中的数据松散连接。当配置支持作为一个服务，就连接仓库中所有的项。可以创建和管理版本和版本集，跟踪变化，在环境实体间建立可追踪的关系。

元-数据服务和子环境服务允许创建带自己的数据和关系类型的本地环境。元-数据是关于数据的数据，而元-数据服务能定义新的实体和关系类型，能查询所定义的类型。子环境服务允许划分环境数据。子项目可有自己的环境。因而它们同时可以与其他正进行中的项目相孤立开来。

状态监控服务允许通过数据状态的改变来启动特定的动作。例如，当创建一个实体的新版本时，就可以触发操作，自动通知该实体的所有用户已可用该实体的新的版本了。通过数据互换服务，一些工具能维护自己的仓库，还能将信息从它们局部存储区中移进和移出。

(iii) 任务管理服务

任务管理服务支持环境中的过程集成。表 8-4 介绍了参考模型已定义的服务。在这样的上下文中，术语“任务”意指某些过程中原子活动单位。

表 8-4 任务管理服务

服务	描述
任务定义	提供定义任务的机制，该机制包括前置条件和后置条件，输入和输出，需要的资源和任务中涉及的角色
任务执行	提供支持任务执行的设施，这也许包含用一个过程编程语言来描述任务的交互操作
任务事务	提供对事务的支持，这些事务在相当一段时间内与一个或多个任务执行有关，应能做到无需将系统退回任务开始执行的状态就能从失败中恢复
任务历史	提供设施来记录任务的执行，查询以前的执行
事件监控	支持事件或引起某任务执行的触发的定义
查账与记账	记录做了什么，以及环境中哪些资源被使用
角色管理	提供定义和管理环境中角色的设施

本质上讲,任务管理服务提供定义和实施(执行)过程模型的操作。这些模型包括诸如活动、角色分派的实体。必须实施这些模型提供支持软件开发过程的活动。可能将活动赋予角色,当某事件(例如一个任务开始)发生时激发活动。允许记录活动历史,记录资源使用情况以及维护资源等等。

任务管理服务可能是参考模型中定义最不完善的服务。1992 年 Brown 将几个框架提案与参考模型比较时,发现这些系统都不能很好地支持过程管理服务。

(iv) 消息服务

消息服务允许工具和框架服务通信。在软件开发环境参考模型中只定义了两种消息服务。

- 消息发派:这一服务支持工具到工具、服务到服务、框架到框架之间的消息传递。可能允许点到点(即,直接)消息互换,允许发送到所有工具和服务的广告消息,允许由一些代理来确认哪些工具和服务应接收该消息,这样可进行消息的多重传递。相关的操作有发送、接收、应答等。
- 工具注册:允许一个工具或服务作为某种类型的消息的接收者登记到消息服务器上。

在许多商用产品上,如 HP 的 SoftBench 已实现了这个简单的模型。它提供了一个分布式网络上的控制集成。一个工具或服务能在某个中央代理处登记,说明所感兴趣的消息类型。当另一个工具或服务希望通信时,它向消息发派服务发送消息。这确定了网络上该消息潜在的接收者,并将消息发送给它们。

(v) 用户界面服务

用户界面服务支持表示集成。软件开发环境参考模型的开发者没有定义一组新的用户界面服务集,而是提议用户界面服务应基于 OSF 的分层用户界面模型(见图 8.16),该模型是在 X-Window的早期工作基础上形成的。

从下往上看,这个用户界面参考模型的头两层提供了基本的物理支持,包括屏幕、输入设备处理和一组基本的图形原语(X-Lib)。接下来的两层建立在这些原语之上,提供了一组能用于表示层创建用户界面的界面对象。在这一级别上,Motif 工具箱已作为事实上的标准而出现。这些较底层的服务,可能属平台服务而非框架服务。

应用
对话
表示
工具箱
工具箱本征集
基本窗口系统接口 (X-Lib)
数据流编码

图 8.16 用户界面参考模型

这级别之上提供的支持较为模糊。表示层支持在设计应用程序界面时直接使用工具箱中提供的图形对象。表示层可能有格式设计工具、用户界面布局工具等。对话层支持界面操作的同步,因而可能包括用户界面管理系统(UIMS),还包括对话规范说明语言。最后,应用层与应用程序和用户界面间的通信有关。

与软件开发环境参考模型的其他服务不同,用户界面服务并未真正提供一个基点来比较和评估框架提议。软件开发环境参考模型假定所有的框架都支持 X/Motif 和一些相关的支撑工具。然而,这个假定并非事实。

软件开发环境参考模型的设计者假定大多数软件开发环境都将运行在 UNIX 或一些相似的操作系统上,这是不现实的。如个人计算机上运行的是不同的操作系统,现在它们的功能已强到可以在网络环境中作为一个节点来使用了。个人计算机上运行一些别的窗口管理系

统，如 MS Windows，这些就与软件开发环境参考模型的假定不相符。

③ 工具

环境框架已为工具的实现提供了一组通用服务。然而，这些服务并非为所有工具使用。一些工具提供了自己的相应的服务，也应能将之与环境集成。因此，环境在平台和框架级都应提供集成工具的设施。

软件开发环境中，有三个工具集成的级别，如图 8.17 所示。

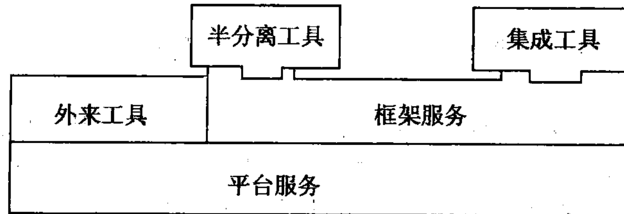


图 8.17 软件开发环境的工具集成

(i) 集成工具：这些工具用框架服务来管理它们所有的数据，其数据结构存储在对象管理系统中。

(ii) 半分离工具：这些工具与框架服务的集成不怎么紧密。它们管理自己的数据结构，但用框架服务来管理文件。文件中包含有数据结构。因而，OMS 能建立文件和文件之间的连接，不能建立文件内部结构之间的连接。

(iii) 外来工具：这些工具仅用平台服务。它们管理自己的数据，但可能使用数据互换服务来把数据传进和传出。

如果已有工具与环境支撑部分运行在相同平台上，那么将已有工具作为外来工具和半分离工具移进软件开发环境就相对容易些。但是，工具和环境服务紧密集成的问题是一个“鸡与蛋”的问题。

(4) PCTE

APSE 的出现，引起了美国和欧洲的高度重视，均在有关机构的支持下，开展了软件开发环境通用框架服务集的研究。

美国国防部基于 APSE 的提案，设立了 CAIS(Common APSE Interface Set)项目，通过研制一个 Ada 环境核心 APSE，开发了一个环境通用工具接口集 CAIS。CAIS 是面向 Ada 的。

与 CAIS 项目进行的同时，在欧洲信息技术研究战略计划(ESPRIT)中，设立了 PCTE (Portable Common Tool Environment)项目。其中，采用了 SEE 基准模型，开发了软件开发环境通用的工具接口 PCTE 第一版，成为欧洲计算机制造商协会(ECMA)的标准，并于 1984 年发布。PCTE 标准是面向 UNIX 和 C 的，旨在标准的通用性，而并非支持面向语言的环境。

应该说，当时的 PCTE 标准仍然存在一些技术缺陷，例如，缺乏对安全性和访问控制的支持，与 UNIX 平台联系过于紧密等。为了解决 PCTE 标准中的问题，国防部门又设立新的项目，资助开发 PCTE+；欧洲计算机行业协会(ECME)也设立项目，支持开发 ECMA PCTE。

由于 PCTE 和 CAIS 这两个提案有许多重复交叉之处，因此美欧双方共同对之进行了综合，并开发出一个称之为 PCIS(Portable Common Interface Standerd)标准(可移植通用接口标准)。尽管于 1994 年初发布了 PCIS 框架定义的第一版本，并于 1995 年予以原型化，但欧洲和

美国还是普遍接受ECMA PCTE。并且，一些厂家，例如IBM，Digital等，还开发出这一标准的实现。这意味着PCTE已成为当时软件开发环境框架的事实标准。

图8.18概括了围绕环境通用接口的研究及成果之间的关系。

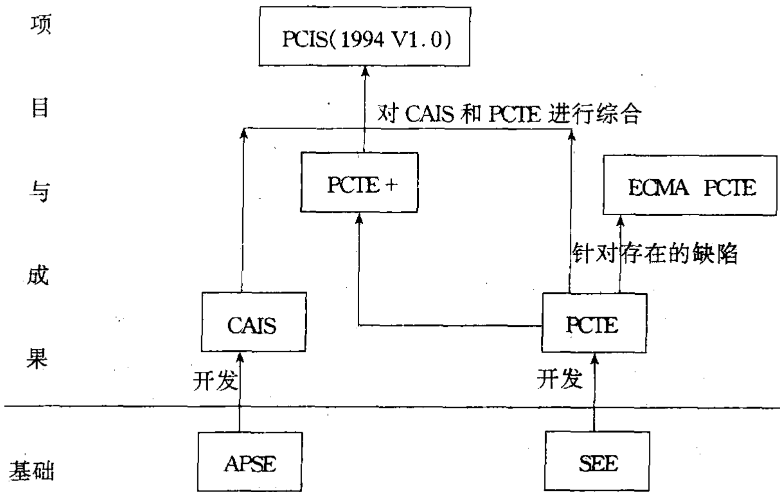


图8.18 PCTE的研究历程

ECMA PCTE的主要特征可概括为：

① 基于ERA(实体-关系-属性)模型，实现对象的管理。包括支持对象之间的连接，对象类与子对象的定义。

② 提供数据恢复、复原能力，即通过控制事务(一个事务是"原子"动作的一个集合)中动作的执行方式(或全部执行，或一个也不执行)，当事务处理中发生错误时，可以将数据库恢复到一个一致的状态。

③ 提供事务执行的管理，即支持进程之间的通信，支持进程的启动、终止和存储。

④ 支持进程和数据在网络上的分派。

⑤ 采用了一个比较复杂的安全模型，其中提供了不同的安全级别，控制对OMS中对象的访问。

1992年以后，Brown等人介绍了ECMA PCTE，并根据SEE基准模型对ECMA PCTE进行了评估，如表8-5所示。

表8-5 PCTE与SEE基准模型的比较

服务	描述
数据仓库	除备份服务外，PCTE提供了所有数据仓库服务
数据集成	除通用查询服务外，PCTE提供了所有数据仓库服务
任务管理	除查账和记账服务外，没有提供其他服务
消息	提供消息分派服务，但没有消息注册服务
用户界面	建议基于PCTE的环境，都采用X-Window实现其用户界面，没有强制采用哪些特定的库

表8.5表明，ECMA PCTE提供了一个相当完整的低层框架服务集。但与SEE基准模型

相比，还需进一步进行扩充。例如，在美国 DoD 环境框架服务的提案中，采用 PCTE 提供数据仓库和数据集成服务，采用 HP 的 SoftBench 提供控制服务，采用 X/Motif 提供用户界面服务，采用 Process Weaver 提供任务管理服务。

3. 大型软件开发环境青鸟系统简介

(1) 概述

"大型软件工程开发环境青鸟系统"是国家科技攻关课题成果，它是由北京大学牵头研制的一个面向对象的软件开发环境。

如前所述，国际上对于软件开发环境的研究与开发始于 80 年代初期。早期的软件开发环境是仅支持个别软件开发过程的单体型软件开发环境。到 80 年代中后期出现了把多种方法、技术和大量的工具集于一体、支持整个软件生存周期的集成型软件开发环境。90 年代初期，随着面向对象方法发展到软件开发过程的各个阶段，出现了支持面向对象开发方法的集成型软件开发环境，但因缺乏实用的对象管理系统的支持，此类环境尚不够完备。

CASE 是促进软件产业发展的重要因素。国外各大软件公司均有自己的软件开发环境作为提高竞争力的手段。我国政府和科技工作者多年来也十分重视软件开发环境的研究与开发。在"六五"期间，我国即安排了软件开发环境的攻关工作，北京大学在这方面的成果是"软件工程核心支撑环境 BD-85"。"七五"期间的攻关研制成功了集成化软件开发环境青鸟 I 型系统，该系统以环境信息库为核心，提供了支持软件生存周期各个过程的软件工具。在"八五"攻关期间，成功地研制了青鸟 II 型。本节主要介绍青鸟 II 型，又称 JB2。

JB2 的特点：

① 把支持面向对象的软件开发作为环境的主要目标之一，具体体现为：

(i) 研制和开发作为集成型软件开发环境核心的对象管理系统 JB2/OMS。

(ii) 以对象管理系统为核心的 JB2 环境的总体结构。

(iii) 设计并实现支持永久对象的面向对象编程语言 CASE-C++。

(iv) 在对象管理系统和 CASE-C++ 语言的支持下进行环境中大部分软件工具的开发和集成。

(v) 提供一套支持面向对象分析、设计和编程的 OO 系列工具。

(vi) 为支持图形用户界面的面向对象开发，提供一个包括大量界面常用成分的界面类库和一个面向对象的界面辅助生成器。

总之，通过对象管理系统、CASE-C++ 语言、OO 系列工具、界面类库及界面辅助生成器的研制，使环境对工具设计者和最终用户的面向对象软件开发形成强有力的支持。

② 较为成功地研制了以数据集成、控制集成和界面集成为中心的开放性环境集成机制，包括：

(i) 以对象管理系统为核心的数据集成部件。

(ii) 以消息服务器和过程控制系统作为控制集成部件。

(iii) 以基本 OSF/Motif 的界面类库和界面辅助生成器作为界面集成部件。

(iv) 在环境集成机制中提供开放性的工具插槽。

(v) 制定符合开放性要求的工具结构模型。

③ 支持多种软件开发方法，包括结构化方法，信息模型法和 OO 方法。

④ 系统既可以集成支持软件生存周期全过程的软件工具，成为通用性软件开发环境，又

可以通过剪裁而形成支持特定应用领域的专用性应用开发平台。

(2) JB2 系统介绍

下面首先介绍 JB2 的总体结构，然后介绍 JB2 环境集成机制和 JB2 环境中的软件工具。

① JB2 总体结构

JB2 是一个面向对象的集成化软件工程开发环境，系统本身的开发也较全面地采用了面向对象的技术。在设计上，JB2 参考国际上一些著名的环境模型，通过对它们的分析和评估，吸取其优点，提出了基于自己已有研究成果之上的集成环境模型。

JB2 集成机制体现了数据、控制和界面三方面的集成，其数据集成基于对象管理系统(OMS)，控制集成基于消息服务器，界面集成基于 OSF/Motif 以及在此基础上开发的 JB2 界面类库和界面辅助生成器。同时，环境中提供了一个面向永久对象的编程语言 CASE-C++ 作为工具的书写语言，保证了工具在环境中的紧密集成。

JB2 的集成机制形成了一个开放的工具插槽。工具的设计遵照统一的工具结构模型，并提倡以 CASE-C++ 语言编写，从而很容易集成到环境中并达到较高的集成度。外来工具在按规范要求封装后也容易被环境接纳。工具之间的通信由消息服务器完成。

图 8.19 显示了 JB2 的总体结构。

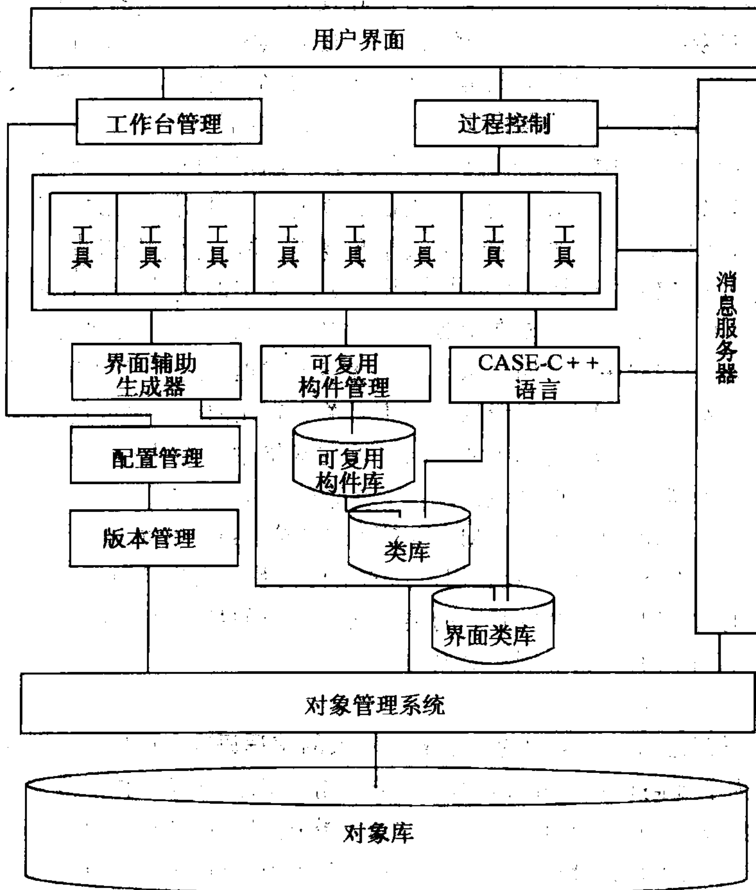


图 8.19 JB2 系统的总体结构

JB2 环境的开放的工具插槽保证了环境易于剪裁和扩充。按结构模型规范要求封装好的工具,实质上构成了一个“插件”,工具向环境的集成仅是将“插件”插入工具插槽中,工具从环境中删除只需将“插件”从插槽中“拔出”。JB2 集成机制所体现出的插槽形式为环境的扩充和剪裁提供了很大方便,也保证了集成和开放的有机统一。图 8.20 显示了工具“插件”与环境工具插槽间的基本关系。

② 环境集成机制的主要部件

JB2 环境集成机制的主要部件如下:

(i) 对象管理系统

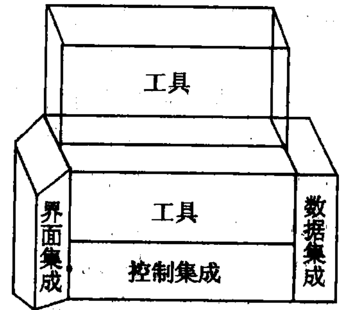


图 8.20 JB2 工具插槽

JB2 的对象管理系统(JB2/OMS)是 JB2 环境的核心,其作用是对软件开发过程中定义的对象类以及对象实例进行存储和管理。JB2/OMS 实现了永久对象的存储与管理,可并发地运行于网络中各个工作站之上,支持在多用户、多程序之间对象的共享和并发执行,由并发控制机制保证其属性信息的一致性。JB2/OMS 对于环境的工具开发者和最终用户提供了统一的面向对象技术支持,即,它管理和存储的对象,既包括构成工具本身的对象,也包括用户在使用工具时产生的结果对象,以及用户以 CASE-C++ 语言开发的任何应用程序中的对象。JB2/OMS 分为以下几个主要部分。

● 类库:JB2 的类库保存并管理环境中长期使用的全部类定义以及类之间的继承关系。库中的类包括:由环境提供的一组预定义的类,各工具开发者提供的工具成分类,一般用户在软件开发过程中不断提交的类。此外,JB2 为支持基于 OSF/Motif 的图形用户界面的开发,提供了一个丰富而高效的界面类库,它是整个 JB2 类库的一个相对独立的子库。类库的主要意义是支持类定义在不同用户、不同程序间的共享和复用,并进行权限控制。

● 对象库:JB2/OMS 的对象库保存并管理 JB2 环境中的全部永久对象的实例信息。这个对象库是 CASE-C++ 语言实现永久对象概念的基层支持。由于这种支持,使用户(包括工具开发者和一般用户)在程序中声明的永久对象能够超越程序运行时间而长期存在。此外,它使程序员可以把永久对象看作是“无缝的”,而不必关心它在内外存之间的转换。JB2/OMS 的设计采用了 Client/Server 结构,使库中的对象可以在网络中并发执行的各个程序或进程之间共享,为了保证对象的数据完整性,OMS 设立了锁管理机制。

● OMS 浏览器:OMS 浏览器是 OMS 的交互式浏览及维护界面,它允许用户浏览类库和对象库中的全部可见的类及对象以及它们之间的结构关系。用户还可以通过它交互式地定义新类、修改类的权限或修改处于开发状态的类定义。环境的特权用户除了可使用上述功能之外,还可以删除已经无用的对象实例和对象类。

(ii) CASE-C++ 语言

CASE-C++ 语言是 JB2 环境中的一个面向对象的编程语言(OOPL)。设计和实现这个语言的主要动机是为了提高在软件开发环境下面向对象的软件开发工作的效率并达到软件工具在环境中的紧密集成,而现有 OOPL 在许多方面不能很理想地达到上述要求,如对永久对象的支持。JB2 选择了目前应用比较广泛的 C++ 语言作为基础,并针对上述要求加以扩充,命名为 CASE-C++ 语言。它对 C++ 完全兼容,主要有以下几点扩充:

● 支持永久对象

CASE-C++语言在JB2的对象管理系统(JB2/OMS)支持下提供了永久对象(Persistent Object)的定义和处理能力。用户在程序中创建一个对象时,可以用保留字persistent声明它是永久的。这样的对象的生命期可超越程序运行的时间而在OMS的对象库中长期保存。本程序或其他程序再次使用这个对象时,对象呈现它上一次使用时的状态。永久对象的描述及处理机制使用户可以把对象看作是“无缝的”——程序员不必关心每个对象是在内存还是在外存。对象在内、外存之间的转储,是由CASE-C++在必要时自动地调用OMS的基本读写函数实现的。程序员不必借用其他存储管理系统来保存对象,从而解除了在程序中进行数据转换和显式地存储与恢复所带来的负担。

- 对象之间关系的强化描述及永久存储——链(Link)机制的引入

CASE-C++引入了链的概念和处理机制来表示对象之间的关系。一个链,从一个源对象出发,链接到一个目的对象。它作为对象的一个属性(实例变量)在源对象中定义,它的值则是目的对象的永久性标识。链机制可使对象之间的关系与对象一起得到长期的保存,这是链和非永久性OOPL中“对象指针”概念的重要区别。链的另一个特点是它可以带有自己的链属性,以描述较复杂的关系信息。因此,在CASE-C++中对象关系的描述能力大大增强。

- 支持类定义的复用和共享

CASE-C++语言提供了使类定义成为可复用构件并提供其他用户共享的设施。在CASE-C++程序中定义的类,可以用export语句提交到类库,以供复用和共享;另一方面,程序中可以用import语句从类库中引入自己所需要(并且有权使用)的类。

CASE-C++语言允许C++语言原有的类定义形式(以小写class为保留字),同时增加了扩充的类定义形式(以大写的CLASS为保留字),凡属以下几种情况必须使用扩充的类定义形式。

类定义的内部使用了CASE-C++的扩充语法成分(例如链);

准备移出以提供复用和共享的类;

准备创建永久对象的类。

- 可变长属性的描述——对象内容

CASE-C++以“对象内容”作为对象的一种属性,它的长度是可变的。这一扩充使对象的属性可以描述那些长度动态变化的数据信息,例如,进行交互式编辑的图形、正文信息。在软件工具中,被加工和输入/输出的信息多属此类。因此,这一扩充特别适应软件工具的开发。

(iii) 用户界面、界面类库和界面辅助生成器

JB2的用户界面是用OSF/Motif开发的。为了从根本上提高用户界面的编程效率,并保证界面风格的一致性,我们在OSF/Motif之上开发了一个JB2界面类库。界面类库是JB2总类库的一个独立的子库,库中的类是针对各种常用的界面部件定义的界面对象类。每个类具有C++和CASE-C++两种语言的定义形式,均用Motif实现。在JB2界面类库的支持下,程序员要定义一个界面成分时,只需引用相应的界面类创建一个具有指定属性值的对象。此外,可用类中提供的函数来控制界面的变化。这使得界面部分的编程工作大为简化(典型地,百余行的Motif程序可以简化到一行到几行C++或CASE-C++程序)。另一方面,它也使不熟悉Motif的程序员能够快速地胜任界面开发工作。

界面类库中预定义的界面类较全面地包括了一般用户常用的界面成分。用户还可以利用这些预定义的类通过派生或组合产生自己的新类并提交到类库。

界面辅助生成器是JB2环境中的一个交互式界面生成工具，它为用户的界面开发提供了可视化的支持。用户可以在屏幕上以“所见即所得”的方式构造自己所需要的界面成分。最终辅助生成器将把屏幕上的开发结果转换为C,C++,CASE-C++或UIL程序源代码。界面辅助生成器既可看成环境集成机制的一个组成部分，也可以作为一个独立的软件工具。

JB2的界面类库和界面辅助生成器可以有效地保证界面设计的规范化和风格的一致性，并且明显地提高了界面的开发效率和质量。

综上所述，JB2的界面开发支持层次如图8.21所示。

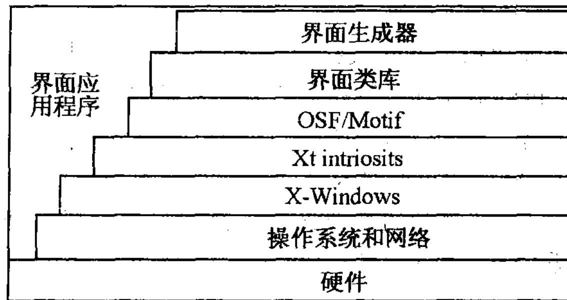


图8.21 JB2界面支持层次

(iv) 可复用构件库

JB2可复用构件库的设计目标是为环境提供一个支持多种复用级别的软件复用机制。源程序一级的复用主要在JB2/OMS中的类库和JB2界面类库的基础上实现。其他级别的复用还包括分析构件和设计构件的复用。对象化的构件是库中最规范的可复用构件，其中包括在OOA阶段产生的分析类，OOD阶段产生的设计类和用OOPL定义的编程类；同时库中也将包括按传统方法开发并按构件库的描述规范所描述的非对象化构件。

JB2构件库的组织允许表示可复用构件之间的多层关系，例如对象化构件之间的继承关系、组成同一系统的构件之间的组合关系、从分析构件到设计构件到编程构件之间的演化关系，以及应用领域分类关系等等。

JB2可复用构件库以基于关键词的检索和交互式的提问细化和理解、定位作为重要的检索策略、并在多层关系模型的支持下提高检索效率。

(v) 消息服务器

JB2的消息服务器是在UNIX系统的IPC/RPC基础上开发的一个高层次的消息服务设施，它用于工具或环境其他独立部件之间的通信服务。消息的发送和接收单位是工具(或环境部件)。消息是网络透明的，其语义由接收单位和开发单位相互约定，并按JB2的统一约定进行描述和登记。

消息服务器是提高环境的控制集成度的重要设施，它为相关工具之间的配合工具和切换提供了方便而有效的支持。例如，设计工具在工作时可以发送消息要求分析工具显示相关的分析文档并进行修改，而操作员不必在这两个工具之间频繁地退出和进入。用JB2消息服务器实现工具之间的通信比直接使用UNIX系统的IPC/RPC要简练得多，例如程序员不必关心事件队列的定义、存取和查询等细节。

消息服务器也是JB2过程控制基础，它为过程控制的实现提供了通信服务。

(vi) 过程控制

过程控制是提高软件质量及生产率的重要保证。对过程模型及其描述语言以及过程驱动软件开发环境的研究是当前 CASE 研究的一个热点。JB2 中探讨了一个面向对象的通用过程模型 OOSP，在该模型中，软件开发过程由一系列对象组成，对象间的交互构成了软件开发的活动。相应地，提出一个面向对象的过程描述语言 OOPDL，使用该语言可以灵活地描述并控制机制建立在消息服务器之上，在它的支持下开发者可以方便地运行软件过程，以提高软件生产率及软件质量。

(vii) 配置管理、版本管理和工作台管理

配置管理、版本管理和工作台管理是当前真正受用户欢迎的软件开发环境中都应具备的功能部件。

配置管理系统通过它的配置库保存用户在环境支持下产生的各种对象，支持这些对象的共享和权限管理，并实现由单元对象(或者系统对象)合成为系统对象的基本操作。

版本管理系统保存配置库中每个对象的各种版本及版本之间的演化关系，实施权限控制，并把版本按使用工作台的用户的要求提供给用户。

在对象管理系统的支持下，配置库和版本管理系统只需保持对象及其版本的逻辑映像，其物理映像由对象库存储，减轻了实现的难度，同时保证了较高的数据集成度。

JB2 中工作台的概念和我们前面介绍的工作台概念不同，在 JB2 中，工作台是环境用户完成一组工作的专用场所。环境为每个用户提供一个树形的工作台结构，用户可以在此结构上创建多个子工作台；每个工作台的作用是作为用户选择工具或使用环境设施(如配置管理、类库、可复用构件库等)的控制界面；作为工具与环境的数据集成机制之间的中转站，工具的输出对象在正式提交配置库之前以及从配置库提取出来准备加工的对象，都临时存放在工作台上；提供对工作台上所存放的对象的选取、复制、删除、提交等操作。

总之，(a) 应该充分发挥 OMS 的作用，上层部件不再直接存放对象的物理实体，而是存放和管理它们的逻辑映像。(b) 具有多个版本的对象，由版本管理部分管理其版本结构，但每个版本的实体被看作 OMS 中的一个独立的对象。这种处理方法既避免了版本管理部件的物理存储负担，又减少了 OMS 设计的逻辑复杂性。(c) 工作台管理体现了界面集成和数据集成两个方面。首先，环境用户是在统一的工作台界面上工作，通过工作台界面，用户可以进一步启动工具或进行配置管理、版本管理；其次，JB2 中的工作台提供了环境用户的工作空间和私有数据存储空间，存储用户在开发软件过程中所产生的而尚未提交的对象，以及对配置库中永久对象的加工都在这里进行。

③ 工具结构模型和环境中的工具

(i) 工具结构模型

工具与环境的接口问题是关系到环境的集成度和开放性的关键问题之一。在没有任何约定的情况下开发出来的工具很难在环境中达到紧密的集成。单纯依靠小范围内的特殊约定进行工具与集成机制的开发又很难使环境具有开放性。在统一的国际标准形成之前，解决这一问题的可行办法是寻找一个对环境适应性较强、开发者容易实施的工具结构模型。青鸟Ⅱ型采用如图 8.22 所示的工具模型。青鸟Ⅱ型系统中 20 多个工具的开发单位使用这个模型来解决工具与环境的接口问题。

在这个模型中，一个工具由四个独立部件构成，即功能部件、数据接口部件、控制接口部

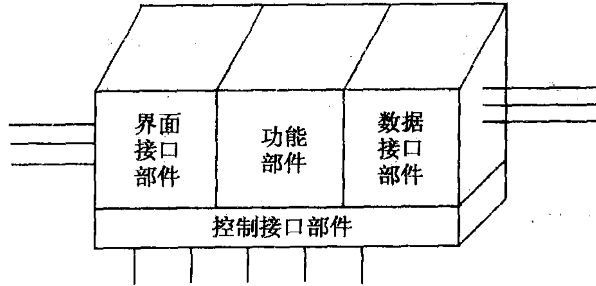


图 8.22　JB2 工具结构模型

件和界面接口部件。其中，功能部件是完成工具自身的功能所需的软件成分，其他三个部件分别实现工具与环境的数据接口、控制接口和界面接口。所有与环境接口有关的设计决策都集中体现在这三个接口部件中，从而隔离了环境对功能部件设计的影响，使工具开发者的大部分工作（有关功能部件的工作）可以独立于环境。按这一模型设计的工具，相当于环境中的一个标准插件，三个接口部件是数据、控制和界面三方面的“插头”，环境的数据集成、控制集成和界面集成机制则是与之对应的“插槽”。

这一模型对工具和环境集成机制具有双重的意义。对工具而言，由于功能部件的独立性，当需要进入一个新的环境时，只需要更换或修改它的接口部件，这使得工具对不同的环境具有较强的可移植性。对环境集成机制而言，一旦确定了“插槽”的规格，其设计则不受具体工具的影响，从而可按照统一的工具结构模式去追求较高的集成度。集成工作也大为简化，因为不必针对每个具体的工具去解决接口问题。此外，按照这个工具结构模型对“外来工具”进行改造或“封装”——使之具有符合环境要求的接口部件，如同给一个非标准的插件装上标准的插头——可以在很大程度上提高环境的开放性。

(ii) JB2 环境中的工具

JB2 环境中的工具分为三大类，即①传统类工具，覆盖整个软件生存周期，支持传统的软件开发方法；②OO 工具，包括从 OOA 到 OOP 的一系列工具；③应用类工具，支持特定应用领域的应用软件开发。

工具举例如下：

- 结构化分析工具 SAT；
- 需求文档分析工具 DAT/DFD；
- 结构化设计工具 SDT；
- 设计文档分析工具 DAT/MSD；
- 文档追踪工具 TRACE；
- 详细设计工具 DDT；
- 详细设计分析工具 DAT/PDL；
- C 编码工具 CCT；
- PAD 图到 C 转换工具 PADT；
- C 程序测试工具 CSTT；
- Fortran 程序测试工具 FSTT；

- C 程序维护工具 SMT；
- 软件项目管理工具 SPMT；
- 软件价格模型估算工具 CMET；
- 用户界面生成工具 UIGT；
- 用户界面生成工具 UIGS；
- 用户界面生成工具 GUIDS；
- 面向对象分析工具 OOAT；
- 面向对象设计编程工具 OODPT；
- 数据库设计工具集 DBTOOLS；
- 数据库应用生成工具 MISGT；
- 地理信息系统辅助工具集 GIS；
- 实时监控系统辅助生成工具 SCADA；
- 人工神经网辅助工具 NNET；
- 网络应用辅助生成工具 RODA；
- 分布式系统辅助生成工具集 DSGT；
- 文档出版工具 DPT。

④ JB2 环境的剪裁——支持不同开发方法和应用领域的软件开发平台

JB2 环境的开放性集成机制和 JB2 工具结构模型使环境具有良好的可剪裁性。根据具体应用的需要对环境进行剪裁，可产生以下几个软件开发平台：

(i) 支持结构化方法的开发平台 JB2/B(中西文版)

该平台含有结构化开发方法有关的环境部件和软件工具，支持瀑布模型中的各个软件过程。

(ii) 支持 OO 方法的开发平台 JB2/OO(中西文版)

该平台含有与 OO 开发方法有关的环境部件、软件工具语言。

(iii) 地理信息系统开发平台 JB2/GIS(中文版)

该平台含有面向对象的分析、设计与编程工具和地理信息系统专用工具集、支持对地理信息系统进行面向对象开发。

(iv) MIS 开发平台 JB2/MIS(中文版)

该平台含有支持结构化方法和 E-R 模型的前期工具和管理信息系统的专用开发工具，支持 MIS 的开发。

(v) 实时监控系统开发平台 JB2/SCADA(西文版)

该平台含有结构化开发方法的一般工具和实时监控系统的专用工具，支持实时监控系统的开发。

习 题 八

1. 解释以下术语：

软件工具　　软件开发环境

2. 简要回答：

(1) Fuggetta 对 CASE 工具的分类；

(2) 软件开发环境的组成与各成分的作用；

(3) 工作台实现软件工具集成的方式；

(4) Wasserman 关于集成化环境所提出的五级模型；

(5) PCTE 研究与开发的目的与历史；

(6) SEE 基准模型；

(7) SEE 基准模型与 PCTE 之间的关系。

3. 叙述分析与设计工作台、测试工作台的构成。

4. 分析 APSE 模型的基本思想以及在软件开发环境研究中的影响。

5. 综合思考题：将程序设计工作台与 VB，VC 进行比较。

附录1　面向对象分析实践指南(要点)

1. 提取类及对象

(1) 通过对问题域的调查研究,其中包括:

亲临现场,征询问题域专家的意见;

参阅其他类似系统;

阅读有关专业性的文献、资料等。

获得系统需求概要。

(2) 提取后选的类及对象:

问题域描述中的名词;

与该系统发生作用的其他系统;

与该系统发生作用的必要设备;

该系统必须观察、记忆的与时间有关的事件;

与系统发生交互的人以及系统必须保留其信息的人;

这些人所属的单位;

系统必须记忆、且不在问题域约束中但在系统责任中的顺序过程(为了指导人机交互),其中,属性是操作过程名、操作特权以及操作步骤等;

系统需要了解掌握的物理位置、办公地点等;

往往是候选的类及对象。

(3) 根据以下规则,对候选的类及对象进行筛选:

有无冗余的类及对象;

考虑每一类中的对象,是否存在需要记忆的属性和其他描述性信息。如果不存在这种需要,那么就要考虑是否把它作为模型中的类及对象;

考虑每一类中的对象,是否提供了某些行为。如果没有提供必要行为,那么就要考虑是否把它作为模型中的类及对象;注:这样的候选类及对象,可能是其他类及对象的属性、服务,也可能是二个类及对象之间的连接。

考虑每一类中的对象,如果只有一个属性,那么它或是反映其他问题域的一个类及对象,或是其他类及对象的属性;

考虑每一类中的对象数目,如果该类中只有一个对象,那么:

如果该类的确反映了问题域,则是一个类及对象;

如果另一个对象和它具有类似的属性、服务,且的确反映了问题域,则应考虑使用"一般-特殊"结构;

如果类及对象中的所有属性,对有些对象是不适用的,那么就要引入"一般-特殊"结构;

如果类及对象中的所有服务,对有些对象是不适用的,那么就要引入"一般-特殊"结构;

如果类及对象仅是可被导出的结果,那么就不要把它作为一个类及对象。

注:以上只关心问题域需求,而不关心系统实现需求。

2. 标识"一般-特殊"、"整体-部分"结构

(1) 提取候选的结构:

利用通常的三种关系:"组装"、"容纳"、"包含",发现可用的结构;

运用"一般-特殊"、"整体-部分"概念,把每一类及对象作为一个一般类(或整体类),提取它的特殊类(或部分类);并以同样的方式,把每一类及对象作为一个特殊类(或部分类),提取它的一般类(或整体类);

参阅同一系统和其他类似问题域中以前分析结果,提取其中可直接应用的结构。

(2) 根据以下规则,对发现或提取的结构进行筛选:

是否表达问题域中的一个有用抽象;

是否反映问题域、系统责任中的问题;

对于"一般-特殊"结构,是否存在继承;

对于"整体-部分"结构,部分类若仅捕获一个状态信息,则把它作为整体类中的一个属性;

对于新提取的类及对象,运用筛选规则进行筛选。

3. 标识主题

(1) 初选主题:

选取结构中最上层的类作为一个主题;

选取不在结构中的任一类作为一个主题;

参阅同一系统和其他类似问题域中以前分析结果,提取其中可直接使用的主题。

(2) 精练主题:按最少依赖、最少交互的原则,保留那些能够反映问题域的主题。

(3) 组织主题层次:按可见性、以及能够引导读者注意的原则,组织主题层次。

4. 标识属性

(1) 提取候选的属性:

考虑如何对每一类及对象中的对象进行一般性描述,发现该对象的一些候选的属性;

针对问题域和系统责任,考虑如何对每一类及对象中的对象进行描述,发现该对象的一些候选的属性;

考虑对象必备的状态,发现该对象的一些候选的属性;

参阅同一系统和其他类似问题域中以前分析结果,提取其中可直接使用的属性。

(2) 根据以下规则,进行筛选:

每一属性应捕获一个"原子"概念;

如果一个属性独立存在对理解系统、构造系统更有意义的话,可以把它作为一个类及对象;

如果一个属性是一个计算,那么就要考虑是否是一个尚未提取的类及对象;

如果一个属性有一个不适当的值,那么就要考虑是否存在一个尚未标识的"一般-特殊"结构。

(3) 说明属性:属性描述包括:

度量单位；

结构；

精度；

建立属性的条件；

其他属性值对它的影响；

该属性跟踪需求文档的描述；

该属性可用于的状态等。

5．标识与定义服务

（1）每一类及对象具有以下简单服务：

create 建立该类中一个新的对象，并将其初始化；

connect 建立或删除一个对象与另一对象的连接；

access 获取或设置一个对象的属性值；

release 释放（删除所有连接）一个对象。

（2）类及对象的“计算”服务：

根据一个对象的责任，确定该对象依赖于属性值的计算；

确定一个对象对每一外部系统或设备的一个变化应具有的检测和响应；

确定一个对象对状态变迁的影响；

参阅同一系统和其他类似问题域中以前分析结果，提取其中可直接使用的服务。

（3）定义类及对象的服务：

参阅同一系统和其他类似问题域中以前分析结果，发现可复用的服务定义。并发现为了定义服务可以参阅的内容；

给出每一类及对象中的服务定义（可以使用“服务流程图”、PDL 以及其他工具）。其中，对于简单服务：create，connect，access，release，其“服务流程图”只在一个类中给出。

6．标识连接

（1）提取候选的连接：

类之间的依赖性；

类之间的参考、引用；

类之间的静态特征（位置、地点等）；

对每一类及对象，如果它需要其他类及对象的服务或数据，则引出一条直线到需要服务的类及对象，并标明服务；

参阅同一系统和其他类似问题域中以前分析结果，提取其中可直接使用的连接。

（2）根据以下规则，进行筛选：

去掉冗余的连接；

去掉那些在问题域之外和与实现有关的连接；

去掉那些利用其他连接定义的连接；

去掉那些根据对象属性的条件所定义的连接。

（3）定义连接约束。

（4）特殊情况-引入关联类：

对于“多对多”连接，如果存在一些属性可以描述之，即这些属性描述了某一时刻两个对象之间的交互，那么可在该连接中插入一个新的类及对象-关联类；

对于一个类中对象之间的连接，考虑是否增加一个新的对象类，用于捕获该连接的更多细节；

对于对象之间的多重连接，考虑是否引入一个新的对象类，把多重连接“合并”为一个连接。

7. 状态分析

(1) 标识类及对象状态：

根据属性值的变化，确定对象的不同状态；

考虑对对象服务的划分或覆盖，确定对象的状态；

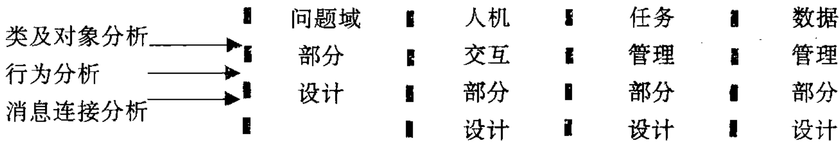
参阅同一系统和其他类似问题域中以前分析结果，提取其中可直接应用的状态。

(2) 状态图。

附录 2 面向对象设计实践指南(要点)

1. 面向对象设计模型

面向对象设计模型由四部分组成：



注：设计活动相对独立，没有固定顺序。

2. 问题域部分设计

(1) 任务

将分析结果作为输入，依据需求变更或设计需求，进行改动和增补。

(2) 实施指南

直接采用面向对象分析策略，具体地说：

复审并改进分析结果；

增补分析结果：

复用设计和编码的类，其中：

如果复用非面向对象语言编写的软件，可把它封装于一个基于服务的界面中；

如果复用面向对象语言编写的软件，可以：

标识现成类中用不到的属性和操作；

增加一个从现成类到问题域的一般/特殊关系；

标识问题域中不再需要、可以从现成类中继承的属性和服务；

修正问题域的结构和连接。

组合问题域中的专用类通过在类库中引入一个根类，对问题域的类进行组合，并通过增加一般类而建立其间的协议。

调整继承支持级别-多继承到单继承或无继承的转换

劈开：即把若干特殊类的对象模拟成由一个一般类的若干对象扮演的一些角色；

展平：即把多继承的层次结构展平为一个单继承的层次结构；

针对无继承的语言，需把每一一般/特殊结构的层次展开，形成一组类及对象。

改进性能

改进速度：主要针对对象之间具有高度频繁的消息通信

——这是一种高耦合的情况

提高观察速度：一种方法是在类及对象中扩展一些保存临时结果的构造；另一种方法是为类及对象扩充低层控制块，形成有利于提高观察速度的一般/特殊结构。

3. 人机交互部分设计

(1) 任务

设计人机交互的规格说明，实现“用户如何命令系统工作”和“系统如何向用户提交信息”

(2) 实施指南

对用户分类并描述用户的任务脚本

分类原则：按技能层次：初学者/临时人员/中级水平/高级水平

按组织层次：行政人员/办公人员/管理人员

按身份层次：客户/职员

对定义的每一类人，制作任务脚本

谁，目的，特征(年龄，教育水平，限制等)，关键的成功要素(必须/想要喜欢/不喜欢/偏见)，熟练程度，任务脚本

设计命令层次

方法：采用过程抽象，组织界面可用的服务。

设计过程：研究用户交互活动的寓意和准则，建立一个初始的命令层

精化命令层

通过考虑：

命令排序：适应用户工作习惯

服务使用的频繁程度

整体/部分组合：发现服务的整体部分结构，对服务进行分块组织

宽度/深度对比：宽度控制在7+/-2个左右，深度控制在三层以内

最少操作步骤：减少点取、拖动及键盘操作

为系统的高水平用户提供“热键”等操作设计详细交互

详细设计交互的一些准则：

采用一致的术语、步骤和活动

减少点击鼠标、键盘的次数；减少(做某些事)下拉菜单的距离

对用户完成某一任务所需要的等待，提供反馈信息

适当给出UNDO操作

减少界面的记忆负担，并合理组织界面的记忆信息

提供简短的联机参考信息

提供具有吸引力的外观等

制作原型

获取：体验

反馈：外观与感受，工作效率，……

设计人机界面类

依赖于所使用的GUI

用户界面设计

4. 任务管理部分设计

(1) 任务

标识系统任务(进程的别称),并进行适当的设计。

(2) 实施指南

标识任务

标识事件驱动的任务;

标识时钟驱动的任务;

标识优先任务和关键任务;

高优先级的服务,可能要作为附加的任务独立出来,即把这种服务与原类分离开来;同样地,低优先级的服务,也可能要作为附加的任务独立出来;关键任务是那些对于系统的成败具有重要影响的任务。有些服务,对于系统的持续操作可能是特别重要的,甚至是在系统不同的形态下所必不可少的,于是可以通过附加的任务——关键任务,将它们与原类分离开来,以隔离了高安全性、高可靠性的处理所需要的精化设计、编码和测试。

标识协调者

标识并设计必要的任务协调者,以便正确地实现多任务之间的同步/异步操作。

审查任务

确保每一任务满足任务的工程标准:时间驱动,事件驱动,优先/关键,协调者,使任务的数目保持最少。

定义任务

任务的定义

为任务命名,并给出简要的说明;

为与任务相关的每一服务增加一个约束-任务名,并为该约束分配一个值;

对于那些由多个任务使用的服务,修改其服务名及描述,以反映该服务的这一特征;

对那些与设备、其他任务和系统协调的服务,要用协议来扩展该服务的规格说明。

协调的说明

说明每个任务如何协调工作;

对于事件驱动的任务,还要描述触发该任务的事件;

对于时钟驱动的任务,还要描述在触发之前所经过的时间间隔,同时指出这一时间间隔是一次性的还是重复性的。

通信的说明

说明每个任务之间如何通信——从哪里取值,往哪里发送数据值(例如:信箱,信号量,缓冲区等)。

任务定义模板

综上,任务可以采用如下模板予以定义:

任务名
任务描述
优先级
包含的服务
任务的协调者
任务的通信

5. 数据管理部分设计

(1) 任务

提供在数据管理系统中存储、检索对象的基本结构。

(2) 实施指南

数据存储方法

采用文件的数据管理

定义一个标签语言。该语言由标签、开始标记、登记项和结束标记组成;

设计一个标签语言分析器,实现一类对象的存储,其中,该分析器包括一个记录各个登记项的登记册。

采用关系数据库的数据管理

列出每一类和该类的属性,定义第三范式表;

为每一第三范式表,定义一个数据库表;

检验性能和存储量。

(以上三步重复执行,直至满足性能和存储量需求。)

采用面向对象的数据管理

不再需要附加的服务,面向对象数据库管理系统本身为对象的存储提供了支持,只需对那些需要长期保存的对象标识出来。

服务(对象存储和检索)设计

采用文件的数据管理

定义一个类及对象,名为:ObjectServer,该类有两个服务:

1) "存储自己"的服务(包括文件定位、记录定位、检索与更新);

2) "检索对象"服务(包括搜索、取值、置初始值)。

采用关系数据库的数据管理

定义一个类及对象,名为:ObjectServer,该类有两个服务:

1) "存储自己"的服务;

2) "检索对象"服务(包括搜索、取值、置初始值)。

6. 设计的一些原则

综上,可以给出有关面向对象设计的一些原则:

耦合

对象之间的低耦合

减少对象之间消息数目

减少消息参数的个数

继承的高耦合

特殊类应真正是一般类的一个特殊类,即:具有严格定义的责任;没有继承一些无关的、不必要的属性和服务;

内聚

服务内聚:一个服务实现一个并且只实现一个功能。

类内聚:没有多余的属性和服务。

一般/特殊结构的内聚:真正表达问题域的一个有用抽象,没有多余的关系、属性和服务。

一般/特殊结构的深度

结构层次控制在 7+/-2 个;在任何情况下,不应超过 20 层。

类及对象的简单性

保持服务的简单,一个对象的公共服务控制在 7+/-2 个;

保持属性的简单(可以通过结构予以简化);

保持协议的简单,并合作对象的数目控制在 7+/-2 个。

协议和服务的简单性

消息协议中的词汇应尽可能的简单,每一消息的参数控制在 3 个以内;

服务的"长度"应控制在适当的数目内,若出现服务"过长",可适当采用一般/特殊结构处理之。

设计的清晰度

使用一致的词汇表;

遵循已有的协议和行为;

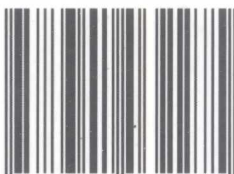
消息应采用一致的形式,更不能建立冗余的消息;

类的命名避免模糊,使其责任一目了然。

参考文献

1. 张效祥主编.计算机科学技术百科全书.北京:清华大学出版社,1998
2. 杨芙清,邵维忠,梅宏.面向对象的 CASE 环境青鸟Ⅱ型系统的设计与实现.中国科学,A 辑 1995:533—542
3. 杨芙清,何新贵.软件工程进展.北京:清华大学出版社,1996
4. 邵维忠,杨芙清.面向对象的系统分析.北京:清华大学出版社,1998 年 12 月
5. 邵维忠,麻志毅,张文娟,孟祥文译.UML 用户指南.北京:机械工业出版社,2001 年 6 月
6. Richard C. Lee, William M. Tepfenhart. UML and C++. Prentice Hall, Inc. 2001
7. 冯玉琳等.软件工程.合肥:中国科学技术大学出版社,1992
8. Donald. G. Firesmith & Edward M. Eykholt, Dictionary of Object Technology, SIGS Book, Inc 1995
9. David Gries, Programming Methodology , Springer - Verlag, New York Heideberg Berlin, 1978
10. Stephen R. Schach, Software Engineering with java, McGraw-Hill Companies, Inc, 1998
11. Jag Sodhi, Software Engineering Methods, Management, and CASE Tools, McGraw-Hill, Inc., 1991
12. Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2001
13. Karl E. Wiegers, Creating Software Engineering Culture, Dorset House Publishing, 1996
14. Ivar jacoboson, Grady Booch, James Rumbaugh. Unified Software Development Process. Addison Wesley Longman. 1999
15. 徐家福,王志坚,翟成祥.对象式程序设计语言.南京:南京大学出版社,1992
16. 邵维忠等译,杨芙清校.面向对象的分析.北京:北京大学出版社,1991
17. 邵维忠等译,杨芙清校.面向对象的设计.北京:北京大学出版社,1994
18. 朱三元,钱乐秋,宿为民.软件工程技术概论.北京:科学出版社,2002
19. 蔡希尧,陈平.面向对象技术.西安:西安电子科技大学出版社,1993
20. 郑人杰,殷人昆,陶永雷.实用软件工程.北京:清华大学出版社,1997
21. 王立福译.并行程序的一个公理化证明技术.计算机科学,1982 5 期
22. 董丽君.软件需求定义语言研究.博士论文,1996
23. 李健.软件过程建模技术和过程实施技术研究.博士论文,19951.S. Pressman,唐世渭、方裕译,软件工程-实践者的研究途径和方法.《小型微型计算机系统》,1984
24. 陈火旺,罗朝晖,马庆鸣.程序设计方法学基础.长沙:湖南科学技术出版社,1987
25. 张龙祥.UML 与系统分析设计.

●责任编辑 沈承凤 ●封面设计 张 虹


ISBN 7-301-03227-7

9 787301 032275 >

ISBN 7—301—03227—7/TP・0318

定价：23.00元