目录

第 1 章　软件工程学概述

迄今为止，计算机系统已经经历了 4 个不同的发展阶段，但是，我们仍然没有彻底摆脱“软件危机”的困扰，软件已经成为限制计算机系统发展的瓶颈。

为了更有效地开发与维护软件，软件工作者在 20 世纪 60 年代后期开始认真研究消除软件危机的途径，从而逐渐形成了一门新兴的工程学科——计算机软件工程学(通常简称为软件工程)。

1.1　软件危机

“软件作坊”　加剧了软件危机的出现

开发软件就是去编写程序；

接到一个项目马上开始写程序；

为了追求速度而不写或者少写文档；

代码不需要注释；

越早开始写程序，完成整个软件开发所用的时间肯定就越

短；
程序在用户处运行时出现错误，开发人员过去修改程序，改好后就运行。
“软件作坊”仍然沿用早期形成的个体化软件开发方法。
在程序运行时发现的错误必须设法改正；用户有了新的需求时必须相应地修改程序；硬件或操作系统更新时，通常需要修改程序以适应新的环境。
“软件危机”就这样开始出现了!
1968 年北大西洋公约组织的计算机科学家召开国际会议，讨论软件危机问题，在这次会议上正式提出并使用了“软件工程”这个名词，一门新兴的工程学科就此诞生了。
1.1.1　软件危机的介绍
软件危机是指在计算机软件的开发和维护过程中所遇到的一系列严重问题。
实际上，几乎所有软件都不同程度地存在这些问题。
软件危机包含下述两方面的问题：
如何开发软件，以满足对软件日益增长的需求；
如何维护数量不断膨胀的已有软件。
(1) 对软件开发成本和进度的估计常常很不准确。为了赶进度和节约成本所采取的一些权宜之计又往往损害了软件产品的质量，从而不可避免地会引起用户的不满。
(2) 用户对“已完成的”软件系统不满意的现象经常发生。软件开发人员和用户之间的信息交流往往很不充分，有行业盲区，常常导致最终产品不符合用户的实际需要。
(3) 软件产品的质量往往靠不住。
软件的特性决定了软件的可靠性和质量保证是一个模糊的概念。
(4) 软件常常是不可维护的。
很多程序中的错误是非常难改正的，不太可能使这些程序适应新的硬件环境。
(5) 软件通常没有适当的文档资料。
软件不仅仅是程序，还应该有一整套文档资料。对于维护人员而言，这些文档资料更是必不可少的。缺乏必要的文档资料或者文档资料不合格，必然给软件开发和维护带来许多严重的困难和问题。
(6) 软件成本在计算机系统总成本中所占的比例逐年上升。
硬件成本逐年下降，而软件成本却持续上升。
(7) 软件开发生产率提高的速度，远远跟不上计算机应用迅速普及深入的趋势。
软件产品“供不应求”的现象使人类不能充分利用现代计算机硬件提供的巨大潜力。
1.1.2　产生软件危机的原因
在软件开发和维护的过程中存在这么多严重问题，一方面与软件本身的特点有关，另一方面也和软件开发与维护的方法不正确有关。
软件的逻辑特性决定了软件的开发、管理和控制过程都相当困难。软件的维护过程通常意味着改正或修改原来的设计，这就在客观上使得软件较难维护。
软件经常需要由多人分工合作，然而，如何保证每个人完成的工作合在一起确实能构成一个高质量的大型软件系统，更是一个极端复杂困难的问题，不仅涉及许多技术问题，诸如分析方法、设计方法、形式说明方法、版本控制等，更重要的是必须有严格而科学的管理。
目前很多软件人员对软件开发和维护有不少糊涂观念，在实践过程中或多或少地采用了错误的方法和技术，认为软件开发就是写程序并设法使之运行，轻视软件维护等，这可能是使软件问题发展成软件危机的主要原因。
有的时候，为了赶进度，也可能采用不规范的开发方法，这种错误的作法也导致了软件危机的加剧。
没有认识到软件开发有自己本身的规律，没有循序渐进，按照软件开发的规律办事。
例如：没有成分理解需求就匆忙编写程序是许多软件开发工程失败的主要原因之一。软件开发人员需要做大量深入细致的调查研究工作，反复多次地和用户交流信息，才能真正全面、准确、具体地了解用户的要求。
事实上，越早开始写程序，完成它所需要用的时间往往越长。
一个软件从始到终，要经历一定时期，通常把软件经历的这个时期称为软件生命周期。
软件开发最初的工作是问题定义，也就是确定要解决的问题是什么；
然后要进行可行性研究，决定该问题是否存在一个可行的解决办法；
接下来进行需求分析，也就是深入具体地了解用户的要求，在所要开发的系统必须做什么这个问题上和用户取得完全一致的看法。
接下来才能进入开发时期，在开发时期首先需要对软件进行设计，然后才能进入编写程序的阶段。
然后是程序的测试期。程序编写完之后还必须经过若干阶段的测试工作，才能最终交付使用。所以，编写程序只是软件开发过程中的一个阶段，而且在典型的软件开发工程中，编写程序所需的工作量只占软件开发全部工作量的 10%～20%。

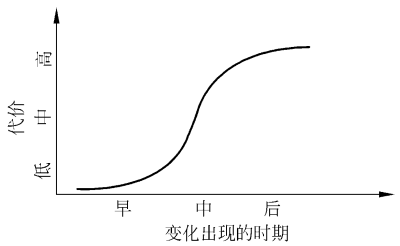


图 1.1 引入同一变动付出的代价随时间变化的趋势
所以说，不能轻视软件维护工作。在软件维护期内，不仅必须改正使用过程中发现的每一个潜伏的错误，而且当环

境变化时还必须相应地修改软件以适应新的环境。因此维护是极端艰巨复杂的工作，需要花费很大代价。
软件工程的一个重要目标就是提高软件的可维护性，减少软件维护的代价。

1.1.3　消除软件危机的途径

IEEE 对软件下的定义：计算机程序、方法、规则、相关的文档资料以及在计算机上运行程序时所必需的数据。其中，方法和规则通常是在文档中说明并在程序中实现的。
一个软件必须由一个完整的配置组成，事实上，软件是程序、数据及相关文档的完整集合。其中，程序是能够完成预定功能和性能的可执行的指令序列；数据是使程序能够适当地处理信息的数据结构；文档是开发、使用和维护程序所需要的图文资料。
更重要的是，必须充分认识到软件开发是一种组织良好、管理严密、各类人员协同配合、共同完成的工程项目。必须充分吸取和借鉴各种工程项目所积累的原理、概念、技术和方法，特别要吸取几十年来人类从事计算机硬件研究和开发的经验教训。
总之，为了解决软件危机，既要有技术措施(方法和工具)，又要有必要的组织管理措施。软件工程正是从管理和技术两方面研究如何更好地开发和维护计算机软件的一门新兴学科。

1.2　软件工程

1.2.1　软件工程的介绍

NATO 在 1968 年给出了软件工程的一个定义：“软件工程就是为了经济地获得可靠的且能在实际机器上有效地运行的软件，而建立和使用完善的工程原理。”
IEEE 在 1993 年给出了一个更全面更具体的定义：“软件工程是：①把系统的、规范的、可度量的途径应用于软件开发、运行和维护过程，也就是把工程应用于软件；②研究①中提到的途径。”
软件工程的本质特性：

1. 软件工程关注于大型程序的构造

“大”与“小”的概念是相对的。
事实上，在此处使用术语“程序”并不十分恰当，现在的软件开发项目通常构造出包含若干个相关程序的“系统”。

2. 软件工程的中心课题是控制复杂性

通常，软件所要解决的问题十分复杂，需要把问题分解，使得分解出的每个部分是简单的、可理解的，而且各部分之间保持简单的通信关系。
用这种方法并不能降低问题的整体复杂性，但是却可使它变成可以管理的。

3. 软件经常变化

绝大多数软件都模拟了现实世界的某一部分。软件为了不被很快淘汰，必须随着现实世界一起变化。

4. 开发软件的效率非常重要

目前，软件供不应求的现象日益严重。所以，软件工程的一个重要课题就是，寻求开发与维护软件的更好更有效的方法和工具。

5. 和谐地合作是开发软件的关键

软件需要多人协同工作才能解决顺利开发。为了有效地合作，必须明确规定每个人的责任和相互通信的方法。为了使大家遵守规定，应该运用标准和规程。通常，可以用工具来支持这些标准和规程。总之，纪律是成功地完成软件开发项目的一个关键。

6. 软件必须有效地支持它的用户

有效地支持用户意味着必须仔细地研究用户，以确定适当的功能需求、可用性要求及其他质量要求(例如，可靠性、响应时间等)。
有效地支持用户还意味着，软件开发不仅应该提交软件产品，而且应该写出用户手册和培训材料。

7. 在软件工程领域中是由具有一种文化背景的人替具有另一种文化背景的人

1.2.2　软件工程的基本原理

软件工作者提出了软件工程的 7 条基本原理。这 7 条原理是确保软件产品质量和开发效率的原理的最小集合。

1. 用分阶段的生命周期计划严格管理

统计显示，在不成功的软件项目中有一半左右是由于计划不周造成的，可见把建立完善的计划作为第一条基本原理是吸取了前人的教训而提出来的。
应该把软件生命周期划分成若干个阶段，并相应地制定出切实可行的计划，然后严格按照计划对软件的开发与维护工作进行管理。
绝不能受客户或上级人员的影响而擅自背离预定计划。

2. 坚持进行阶段评审

软件的质量保证工作贯穿软件开发的始终。因此，在每个阶段都进行严格的评审，以便尽早发现在软件开发过程中所犯的错误，是一条必须遵循的重要原则。

3. 实行严格的产品控制

当需求改变时，为了保持软件各个配置成分的一致性，必须实行严格的产品控制，其中主要是实行基线配置管理。
基线配置管理也称为变动控制：一切有关修改软件的建议，特别是涉及到对基准配置的修改建议，都必须按照严格的规程进行评审，获得批准以后才能实施修改。

4. 采用现代程序设计技术

采用先进的技术不仅可以提高软件开发和维护的效率，而且可以提高软件产品的质量。

5. 结果应能清楚地审查

软件产品是逻辑产品。软件开发人员的工作进展情况可见性差，难以准确度量，从而使得软件产品的开发过程比一般产品的开发过程更难于评价和管理。为了提高软件开发过程的可见性，更好地进行管理，应该根据软件开发项目的总目标及完成期限，规定开发组织的责任和产品标准，

从而使得所得到的结果能够清楚地审查。

6. 开发小组的人员应该少而精

开发小组人员的素质和数量是影响软件产品质量和开发效率的重要因素。

软件开发小组的组成人员的素质应该好，而人数则不宜过多。

7. 承认不断改进软件工程实践的必要性

遵循上述 6 条基本原理，就能够按照当代软件工程基本原理实现软件的工程化生产，但是，仅有上述 6 条原理并不能保证软件开发与维护的过程能跟上技术的不断进步。因此，Boehm 提出软件工程的第 7 条基本原理。按照这条原理，不仅要积极主动地采纳新的软件技术，而且要注意不断总结经验。

1.2.3 软件工程方法学

软件工程包括技术和管理两方面的内容，是技术与管理紧密结合所形成的工程学科。

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

通常把在软件生命周期全过程中使用的一整套技术方法的集合称为方法学(methodology)，也称为范型(paradigm)。

软件工程方法学包含 3 个要素：方法、工具和过程。其中，方法是完成软件开发的各项任务的技术方法，回答“怎样做”的问题；工具是为运用方法而提供的自动的或半自动的软件工程支撑环境；过程是为了获得高质量的软件所需要完成的一系列任务的框架，它规定了完成各项任务的工作步骤。

目前使用得最广泛的软件工程方法学，分别是传统方法学和面向对象方法学。

1. 传统方法学

传统方法学也称为生命周期方法学或结构化范型。它采用结构化技术来完成软件开发的各项任务，并使用适当的软件工具或软件工程环境来支持结构化技术的运用。这种方法学把软件生命周期的全过程划分为若干个阶段，然后顺序地完成每个阶段的任务。采用这种方法学开发软件的时候，从对问题的抽象逻辑分析开始，一个阶段一个阶段地进行开发。

在每一个阶段结束之前都必须进行正式严格的技术审查和管理复审，从技术和管理两方面对这个阶段的开发成果进行检查。

审查的主要标准就是每个阶段都应该交出最新的文档资料。

采用生命周期方法学可以大大提高软件开发的成功率，软件开发的生产率也能明显提高。

2. 面向对象方法学

与传统方法不同，面向对象方法把数据和行为看成同等重要，它是一种以数据为主线，把数据和对数据的操作紧密地结合起来的方法。

概括地说，面向对象方法学具有下述 4 个要点。

(1) 把对象(object)作为融合了数据及其操作行为的统一软件构件。用对象分解取代了传统方法的功能分解。

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

(3) 继承。

(4) 封装性。

用面向对象方法学开发软件的过程，是一个主动地多次反复迭代的演化过程。面向对象方法在概念和表示方法上的一致性，保证了在各项开发活动之间的平滑过渡。

最终的产品由许多独立的对象组成，降低了软件产品的复杂性，提高了软件的可理解性，简化了软件的开发和维护工作。对象可重复使用，对象具有继承性和多态性，进一步提高了面向对象软件的可重用性。

1.3 软件生命周期

软件生命周期由软件定义、软件开发和软件维护 3 个时期组成，每个时期又划分成若干个阶段。

软件定义时期的任务是： 确定软件开发必须完成的总目标；确定工程的可行性；导出实现工程目标应该采用的策略及系统必须完成的功能；估计完成该项工程需要的资源和成本，并且制定工程进度表。这个时期通常又称为系统分析，由系统分析员负责完成。

软件定义时期通常划分成 3 个阶段，即：问题定义、可行性研究和需求分析。

开发时期具体设计和实现在前一个时期定义的软件，通常由 4 个阶段组成：总体设计，详细设计，编码和单元测试，综合测试。其中前两个阶段称为系统设计，后两个阶段称为系统实现。

维护时期的主要任务是使软件持久地满足用户的需要。具体地说，当软件在使用过程中发现错误时应该加以改正；当环境改变时应该修改软件以适应新的环境；当用户有新要求时应该及时改进软件以满足用户的新需要。

1. 问题定义

问题定义阶段必须回答的关键问题是： “要解决的问题是什么?” 在实践中它可能是最容易被忽略的一个步骤。

通过对客户的访问调查，系统分析员扼要地写出关于问题性质、工程目标和工程规模的书面报告，经过讨论和必要的修改之后这份报告应该得到客户的确认。

2. 可行性研究

这个阶段要回答的关键问题是： “对于上一个阶段所确定的问题有行得通的解决办法吗?”

为了解决这个问题，系统分析员需要进行一次头脑中的系

统分析和设计过程。
可行性研究过程比较简短，这个阶段的任务不是具体解决问题，而是研究问题是否值得去解，是否有可行的解决办法。
可行性研究的结果是使用部门负责人作出是否继续进行这项工程的决定的重要依据。
3. 需求分析
这个阶段的任务是准确地确定“为了解决这个问题，目标系统必须做什么”，主要是确定目标系统必须具备哪些功能。
用户的长处和困难
开发人员的长处和困难
因此，系统分析员在需求分析阶段必须和用户密切配合，充分交流信息，以得出经过用户确认的系统逻辑模型。
通常用数据流图、数据字典和简要的算法表示系统的逻辑模型。
系统逻辑模型是设计和实现目标系统的基础，因此必须准确完整地体现用户的要求。需要用文档准确地记录对目标系统的需求，这份文档通常称为规格说明书(specification)。
4. 总体设计
这个阶段必须回答的关键问题是：“概括地说，应该怎样实现目标系统?”总体设计又称为概要设计。
首先，应该设计出实现目标系统的几种可能的方案。软件工程师应该用适当的表达工具描述每种方案，分析每种方案的优缺点，并在充分权衡各种方案的利弊的基础上，推荐一个最佳方案。此外，还应该制定出实现最佳方案的详细计划。如果客户接受所推荐的方案，则应该进一步完成下述的另一项主要任务。
设计工作确定了解决问题的策略及目标系统中应包含的程序。总体设计的另一项主要任务就是设计程序的体系结构，也就是确定程序由哪些模块组成以及模块间的关系。
5. 详细设计
详细设计阶段的任务就是把解法具体化，也就是回答下面这个关键问题：“应该怎样具体地实现这个系统呢?”
这个阶段的任务还不是编写程序，而是设计出程序的详细规格说明。程序员可以根据它们写出实际的程序代码。
详细设计也称为模块设计，在这个阶段将详细地设计每个模块，确定实现模块功能所需要的算法和数据结构。
6. 编码和单元测试
这个阶段的关键任务是写出正确的容易理解、容易维护的程序模块。
程序员根据目标系统的性质和实际环境，选取一种适当的程序设计语言，把详细设计的结果翻译成用选定的语言书写的程序，并且测试每一个模块。
7. 综合测试
这个阶段的关键任务是通过各种类型的测试使软件达到预定的要求。
最基本的测试是集成测试和验收测试。
集成测试是根据设计的软件结构，把经过单元测试检验的模块按某种选定的策略装配起来，在装配过程中对程序进行必要的测试。
验收测试则是按照规格说明书的规定，由用户对目标系统进行验收。
必要时还可以再通过现场测试或平行运行等方法对目标系统进一步测试检验。
7. 综合测试
为了使用户能够积极参加验收测试，并且在系统投入运行以后能够正确有效地使用这个系统，通常需要对用户进行培训。
通过对软件测试结果的分析可以预测软件的可靠性；反之，根据对软件可靠性的要求，也可以决定测试和调试过程什么时候可以结束。
同时用文档资料把测试计划、详细测试方案以及实际测试结果保存下来，作为软件配置的一个组成部分。
8. 软件维护
维护阶段的关键任务是，通过各种必要的维护活动使系统持久地满足用户的需要。
通常有四类维护活动：
改正性维护：诊断和改正在使用过程中发现的软件错误；
适应性维护：修改软件以适应环境的变化；
完善性维护：根据用户的要求改进或扩充软件使它更完善；
预防性维护：修改软件为将来的维护活动预先做准备。
每一项维护活动都应该准确地记录下来，作为正式的文档资料加以保存。
在实际从事软件开发工作时，软件规模、种类、开发环境及开发时使用的技术方法等因素，都影响阶段的划分。

1.4 软件过程

软件过程是为了获得高质量软件所需要完成的一系列任务的框架，它规定了完成各项任务的工作步骤。
在完成开发任务时必须进行一些开发活动，并且使用适当的资源，在过程结束时将把输入转化为输出。因此，ISO 9000 把过程定义为：使用资源将输入转化为输出的活动所构成的系统。
“系统”的含义是广义的：“系统是相互关联或相互作用的一组要素。”
过程定义了运用方法的顺序、应该交付的文档资料、为保证软件质量和协调变化所需要采取的管理措施，以及标志软件开发各个阶段任务完成的里程碑。
没有一个适用于所有软件项目的任务集合。通常，一个任务集合包括一组软件工程任务、里程碑和应该交付的产品。
使用生命周期模型描述软件过程。生命周期模型规定了把

生命周期划分成哪些阶段及各个阶段的执行顺序，也称为过程模型。

1.4.1 瀑布模型

瀑布模型是应用得最早，也是最广泛的生命周期模型。传统软件工程方法学的软件过程，基本上可以用瀑布模型来描述。

图 1.2 所示为传统的瀑布模型。按照传统的瀑布模型开发软件，有下述的几个特点。

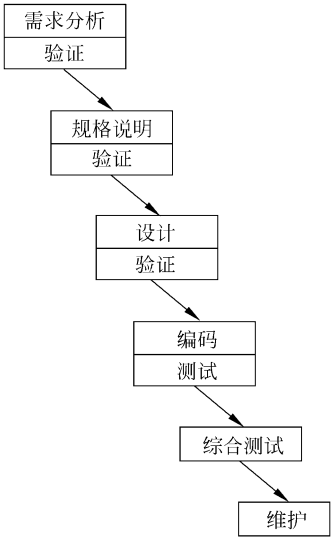


图 1.2 传统的瀑布模型

1. 阶段间具有顺序性和依赖性

这个特点有两重含义： ①必须等前一阶段的工作完成之后，才能开始后一阶段的工作； ②前一阶段的输出文档就是后一阶段的输入文档，因此，只有前一阶段的输出文档正确，后一阶段的工作才能获得正确的结果。

2. 推迟实现的观点

对于规模较大的软件项目来说，往往编码开始得越早最终完成开发工作所需要的时间反而越长。这是因为，前面阶段的工作没做或做得不扎实，过早地考虑程序实现，往往导致大量返工，有时甚至发生无法弥补的问题。

瀑布模型在编码之前设置了系统分析与系统设计的各个阶段，分析与设计阶段的基本任务规定，在这两个阶段主要考虑目标系统的逻辑模型，不涉及软件的物理实现。

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

3. 质量保证的观点

软件工程的基本目标是优质、高产。为了保证所开发的软件的质量，在瀑布模型的每个阶段都应坚持两个重要做法：

1. 每个阶段都必须完成规定的文档，没有交出合格的文档就是没有完成该阶段的任务。完整、准确的合格文档不仅是软件开发时期各类人员之间相互通信的媒介，也是运行时期对软件进行维护的重要依据。
2. 每个阶段结束前都要对所完成的文档进行评审，以便尽早发现问题，改正错误。及时审查，是保证软件质量，降低软件成本的重要措施。

实际的瀑布模型是带“反馈环”的，如图 1.3 所示。当在后面阶段发现前面阶段的错误时，需要沿图中左侧的反馈线返回前面的阶段，修正前面阶段的产品之后再回来继续完成后面阶段的任务。

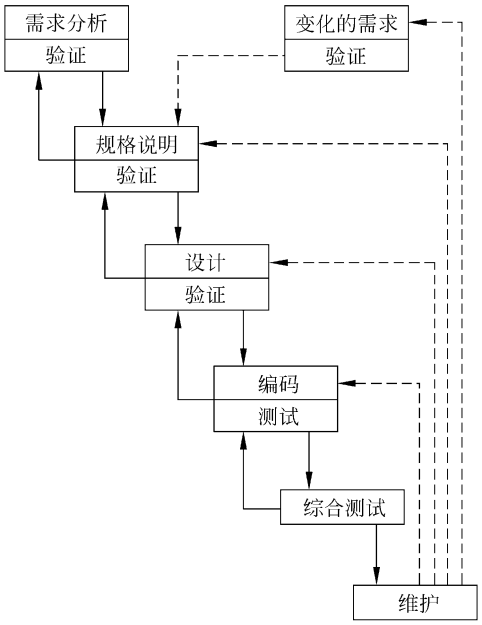


图 1.3 实际的瀑布模型

瀑布模型有许多优点：可迫使开发人员采用规范的方法；严格地规定了每个阶段必须提交的文档；要求每个阶段交出的所有产品都必须经过质量保证小组的验证。

各个阶段产生的文档是维护软件产品时必不可少的，没有文档的软件几乎是不可能维护的。遵守瀑布模型的文档约束，将使软件维护变得比较容易一些，能显著降低软件预算。

可以说，瀑布模型的成功在很大程度上是由于它基本上是一种文档驱动的模型。在软件产品交付之前，用户只能通过文档来了解产品是什么样的。事实上，要求用户不经过实践就提出完整准确的需求，在许多情况下是不切实际的。总之，由于瀑布模型几乎完全依赖于书面的规格说明，很可能导致最终开发出的软件产品不能真正满足用户的需要。

采用瀑布模型进行软件开发，往往需要较长的开发周期。

1.4.2 快速原型模型

快速原型是快速建立起来的可以在计算机上运行的程序，它所能完成的功能往往是最终产品能完成的功能的一个子集。如图 1.4 所示，快速原型模型的第一步是快速建立一个能反映用户主要需求的原型系统，让用户在计算机上试用它，通过实践来了解目标系统的概貌。

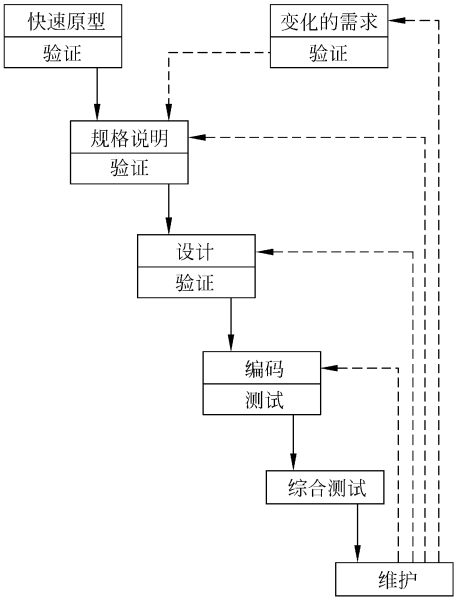


图 1.4 快速原型模型

通常，用户试用原型系统之后会提出许多修改意见，开发人员按照用户的意见快速地修改原型系统，然后再次请用户试用……一旦用户认为这个原型系统确实能做他们所需要的工作，开发人员便可据此书写规格说明文档，根据这份文档开发出的软件可以满足用户的真实需求。

原型模型是不带反馈环的， 软件产品的开发基本上是线性顺序进行的。能做到基本上线性顺序开发的主要原因如下：

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

（2） 开发人员通过建立原型系统已经了解许多，因此，在设计和编码阶段发生错误的可能性也比较小。

快速原型的本质是“快速”。开发人员应该尽可能快地建造出原型系统，以加速软件开发过程，节约软件开发成本。

原型的用途是获知用户的真正需求，一旦需求确定了，原型将被抛弃。

1.4.3 增量模型

增量模型也称为渐增模型，如图 1.5 所示。使用增量模型开发软件时，把软件产品作为一系列的增量构件来设计、编码、集成和测试。每个构件由多个相互作用的模块构成，并且能够完成特定的功能。使用增量模型时，第一个增量构件往往实现软件的基本需求，提供最核心的功能。第二个增量构件提供更完善的编辑和文档生成功能；第三个增量构件实现拼写和语法检查功能；第四个增量构件完成高级的页面排版功能。

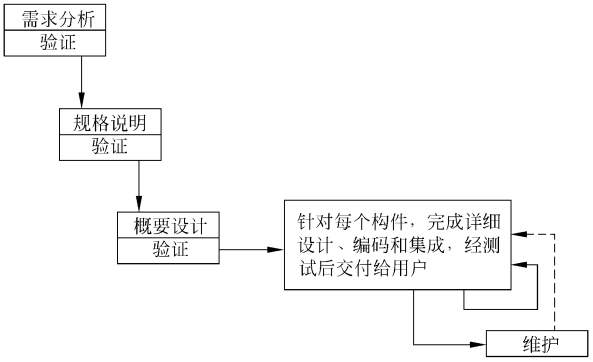


图 1.5 增量模型

把软件产品分解成增量构件时，应该使构件的规模适中，规模过大或过小都不好。

采用瀑布模型或快速原型模型开发软件时，目标都是一次就把一个满足所有需求的产品提交给用户。增量模型则与之相反，它分批地逐步向用户提交产品，整个软件产品被分解成许多个增量构件，开发人员一个构件接一个构件地向用户提交产品。从第一个构件交付之日起，用户就能做一些有用的工作。能在较短时间内向用户提交可完成部分工作的产品，是增量模型的一个优点。

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

使用增量模型的困难是，在把每个新的增量构件集成到现有软件体系结构中时，必须不破坏原来已经开发出的产品。必须把软件的体系结构设计得便于按这种方式进行扩充，向现有产品中加入新构件的过程必须简单、方便，软件体系结构必须是开放的。从长远观点看，具有开放结构的软件拥有真正的优势，这样的软件的可维护性明显好于封闭结构的软件。

1.4.4 螺旋模型

软件风险是任何软件开发项目中都普遍存在的实际问题，项目越大，承担该项目所冒的风险也越大。软件风险可能在不同程度上损害软件开发过程和软件产品质量。因此，在软件开发过程中必须及时识别和分析风险，并且采取适当措施以消除或减少风险的危害。

构建原型是一种能使某些类型的风险降至最低的方法。螺旋模型的基本思想是，使用原型及其他方法来尽量降低风险。

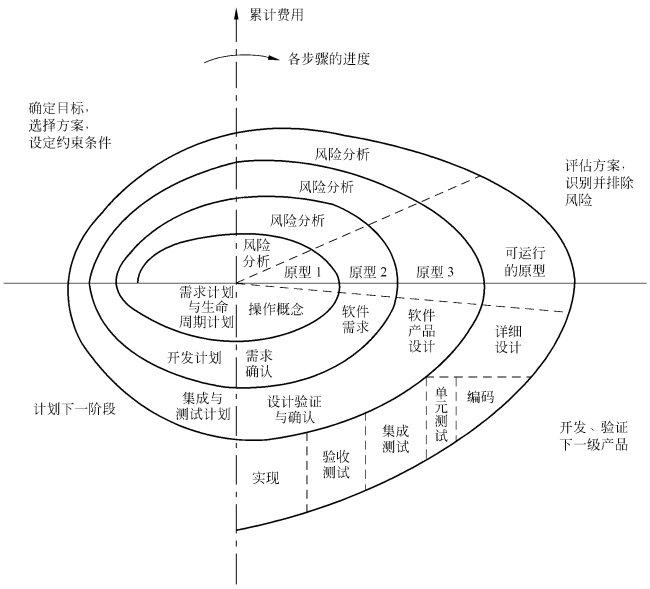


图 1.8 完整的螺旋模型

螺旋模型有许多优点：对可选方案和约束条件的强调有利于已有软件的重用，也有助于把软件质量作为软件开发的一个重要目标；减少了过多测试（浪费资金）或测试不足（产品故障多）所带来的风险。

螺旋模型主要适用于内部开发的大规模软件项目。如果进行风险分析的费用接近整个项目的经费预算，则风险分析是不可行的。事实上，项目越大，风险也越大，因此，进行风险分析的必要性也越大。

1.5 小结

本章首先介绍了软件危机产生的原因及其现象，然后介绍了软件工程的概念，并用生命周期方法学把软件生命周期划分为若干个相对独立的阶段，每个阶段完成一些确定的任务，交出最终的软件配置的一个或几个成分(文档或程序)； 基本上按顺序完成各个阶段的任务，在完成每个阶段的任务时采用结构化技术和适当的软件工具；在每个阶段结束之前都进行严格的技术审查和管理复审。

把软件生命周期划分成问题定义、可行性研究、需求分析、总体设计、详细设计、编码和单元测试、综合测试以及运行维护等 8 个阶段。

软件过程是为了获得高质量的软件产品所需要完成的一系列任务的框架，它规定了完成各项任务的工作步骤。由于没有一个适用于所有软件项目的任务集合，科学、有效的软件过程应该定义一组适合于所承担的项目特点的任务集合。

使用软件过程模型简洁地描述软件过程，本章介绍了 4 种典型的软件过程模型。

瀑布模型历史悠久、广为人知，它的优势在于它是规范的、文档驱动的方法。

快速原型模型通过快速构建起一个可在计算机上运行的原型系统，让用户试用原型并收集用户反馈意见的办法，获取用户的真实需求。

增量模型具有可在软件开发的早期阶段使投资获得明显回报和较易维护的优点，但是，要求软件具有开放的结构是使用这种模型时固有的困难。

风险驱动的螺旋模型适用于内部开发的大型软件项目，但是，只有在开发人员具有风险分析和排除风险的经验及专门知识时，使用这种模型才会获得成功。

第 2 章 可行性研究

2.1 可行性研究的任务

可行性研究的目的不是解决问题，而是确定问题是否值得去解决。

分析几种主要的可能解法的利弊，从而判断原定的系统规模和目标是否现实，系统完成后所能带来的效益是否大到值得投资开发这个系统的程度。因此，可行性研究实质上是要进行一次大大压缩简化了的系统分析和设计的过程，也就是在较高层次上以较抽象的方式进行的系统分析和设计的过程。

首先需要进一步分析和澄清问题定义。在问题定义阶段初步确定规模和目标，如果对目标系统有任何约束和限制，也必须把它们清楚地列举出来。

在澄清了问题定义之后，分析员应该导出系统的逻辑模型，研究若干种可供选择的主要解法。对每种解法都应该仔细研究它的可行性。一般说来，至少应该从下述四方面研究每种解法的可行性：

(1) 技术可行性

(2) 经济可行性

(3) 操作可行性

(4) 市场可行性

必要时还应该从法律、社会效益等更广泛的方面研究每种解法的可行性。

分析员应该为每个可行的解法制定一个粗略的实现进度。

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

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

2.2 可行性研究过程

1. 复查系统规模和目标

分析员访问关键人员，阅读和分析有关的材料，以便对问题定义阶段书写的关于规模和目标的报告书进一步复查确认，改正含糊或不确切的叙述，准确地描述对目标系统的一切限制和约束。

2. 研究目前正在使用的系统

现有的系统是信息的重要来源，新的目标系统必须能完成它的基本功能；另一方面，现有的系统必然有某些缺点，新系统必须能解决旧系统中存在的问题。此外，从经济角

度看，如果新系统不能增加收入或减少使用费用，那么新系统就不如旧系统。

分析员应该画出描绘现有系统的高层系统流程图，并请有关人员检验他对现有系统的认识是否正确。

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

3. 导出新系统的高层逻辑模型

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

通过前一步的工作，分析员使用数据流图描绘数据在系统中流动和处理的情况，使用数据字典定义系统中使用的数据。数据流图和数据字典共同定义了新系统的逻辑模型，以后可以从这个逻辑模型出发设计新系统。

4. 进一步定义问题

分析员应该和用户一起再次复查问题定义、工程规模和目标，这次复查应该把数据流图和数据字典作为讨论的基础。

可行性研究的前 4 个步骤实质上构成一个循环。分析员定义问题，分析这个问题，导出一个试探性的解；在此基础上再次定义问题，再一次分析这个问题，修改这个解；继续这个循环过程，直到提出的逻辑模型完全符合系统目标。

5. 导出和评价供选择的解法

分析员应该从建议的系统逻辑模型出发，导出若干个较高层次的物理解法供比较和选择。

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

最后为每个在技术、操作和经济等方面都可行的系统制定实现进度表。

6. 推荐行动方针

如果分析员认为值得继续进行这项开发工程，那么他应该选择一种最好的解法，并且说明选择这个解决方案的理由。

7. 草拟开发计划

分析员应该为所推荐的方案草拟一份开发计划，除了制定工程进度表之外还应该估计对各类开发人员和各种资源的需要情况，应该指明什么时候使用以及使用多长时间。此外还应该估计系统生命周期每个阶段的成本。最后应该给出下一个阶段的详细进度表和成本估计。

8. 书写文档提交审查

应该把上述可行性研究各个步骤的工作结果写成清晰的文档，请用户、客户组织的负责人及评审组审查，以决定是否继续这项工程及是否接受分析员推荐的方案。

2.3 系统流程图

系统流程图是概括地描绘物理系统的传统工具。它的基本思想是用图形符号以黑盒子形式描绘组成系统的每个部件(程序，文档，数据库，人工过程等)。系统流程图表达的是数据在系统各部件之间流动的情况，而不是对数据进行加工处理的控制过程。

2.3.1 符号

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

图 2.1 基本符号

2.3.2 例子

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

该装配厂使用一台小型计算机处理更新库存清单主文件和产生定货报告的任务。零件库存量的每一次变化称为一个事务，由放在仓库中的 CRT 终端输入到计算机中；系统中的库存清单程序对事务进行处理，更新存储在磁盘上的库存清单主文件，并且把必要的定货信息写在磁带上。最后，每天由报告生成程序读一次磁带，并且打印出定货报告。图 2.3 的系统流程图描绘了上述系统的概貌。

图中每个符号用黑盒子形式定义了组成系统的一个部件，箭头确定了信息通过系统的逻辑路径。

系统流程图的习惯画法是使信息在图中从顶向下或从左向右流动。

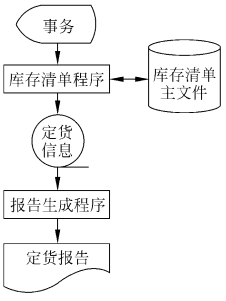


图 2.3 库存清单系统的系统流程图

2.3.3 分层

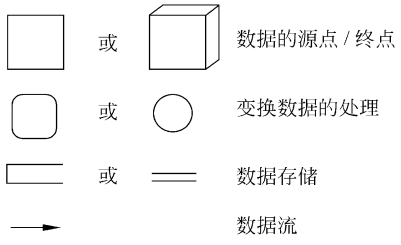
面对复杂的系统时，一个比较好的方法是分层次地描绘这

个系统。首先用一张高层次的系统流程图描绘系统总体概貌，表明系统的关键功能；然后分别把每个关键功能扩展到适当的详细程度。

2.4 数据流图

数据流图(DFD)是一种图形化技术，它描绘信息流和数据从输入到输出的过程中所经受的变换。在数据流图中没有任何具体的物理部件，它只是描绘数据在软件中流动和被处理的逻辑过程。数据流图是分析员与用户之间极好的通信工具。此外，设计数据流图时只需考虑系统必须完成的基本逻辑功能，所以是进行软件设计很好的工具。

2.4.1 符号



(a)

(b)

2.4.1 符号

数据流图有四种基本符号：

正方形(或立方体)表示数据的源点或终点；

圆角矩形(或圆形)代表变换数据的处理；

开口矩形(或两条平行横线)代表数据存储；

箭头表示数据流，即特定数据的流动方向。

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

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

2.4.2 例子

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

数据流图有 4 种成分：源点或终点，处理，数据存储和数据流。因此，第一步可以从问题描述中提取数据流图的 4 种成分： 首先考虑数据的源点和终点，从上面对系统的描述可以知道“采购部每天需要一张定货报表”，“通过放在仓库中的 CRT 终端把事务报告给定货系统”，所以采购员是数据终点，而仓库管理员是数据源点。接下来考虑处理，再一次阅读问题描述，“采购部需要报表”，显然他们还没有这种报表，因此必须有一个用于产生报表的处理。事务的后果是改变零件库存量，然而任何改变数据的操作都是处理，因此对事务进行的加工是另一个处理。注意，在问题描述中并没有明显地提到需要对事务进行处理，但是通过分析可以看出这种需要。最后，考虑数据流和数据存储：系统把定货报表送给采购部，因此定货报表是一个数据流；事务需要从仓库送到系统中，显然事务是另一个数据流。产生报表和处理事务这两个处理在时间上明显不匹配——每当有一个事务发生时立即处理它，然而每天只产生一次定货报表。因此，用来产生定货报表的数据必须存放一段时间，也就是应该有一个数据存储。

表 2.1 总结了上面分析的结果，其中加星号标记的是在问题描述中隐含的成分。

数据流图是系统的逻辑模型，然而任何计算机系统实质上都是信息处理系统，也就是说计算机系统本质上都是把输入数据变换成输出数据。因此，任何系统的基本模型都由若干个数据源点/终点以及一个处理组成，这个处理就代表了系统对数据加工变换的基本功能。对于上述的定货系统可以画出图 2.5 这样的基本系统模型。

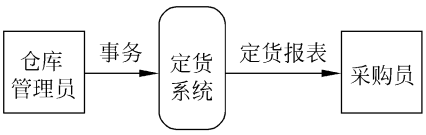


图 2.5 定货系统的基本系统模型

从基本系统模型这样非常高的层次开始画数据流图是一个好办法。在这个高层次的数据流图上是否列出了所有给定的数据源点/终点是一目了然的，因此它是很有价值的通信工具。

然而，这张图太抽象了，从图上对定货系统所能了解到的信息非常有限。下一步应该把基本系统模型细化，描绘系统的主要功能。从表 2.1 可知，“产生报表”和“处理事务”是系统必须完成的两个主要功能，它们将代替图 2.5 中的“定货系统”(图 2.6)。

此外，细化后的数据流图中还增加了两个数据存储：处理事务需要“库存清单”数据；产生报表和处理事务在不同时间，因此需要存储“定货信息”。除了表 2.1 中列出的两个数据流之外还有另外两个数据流，它们与数据存储相

同。这是因为从一个数据存储中取出来的或放进去的数据通常和原来存储的数据相同，也就是说，数据存储和数据流只不过是同样数据的两种不同形式。

在图 2.6 中给处理和数据存储都加了编号，这样做的目的是便于引用和追踪。

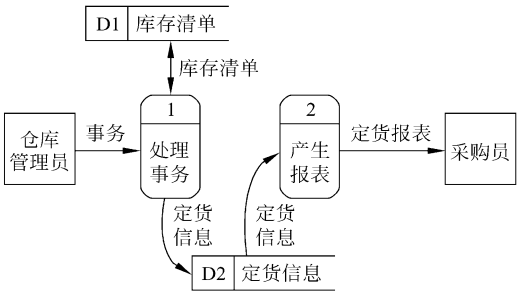


图 2.6 定货系统的功能级数据流图

接下来应该对功能级数据流图中描绘的系统主要功能进一步细化。考虑通过系统的逻辑数据流：当发生一个事务时必须首先接收它；随后按照事务的内容修改库存清单；最后如果更新后的库存量少于库存量临界值时，则应该再次定货，也就是需要处理定货信息。因此，把“处理事务”这个功能分解为下述 3 个步骤，这在逻辑上是合理的：“接收事务”、“更新库存清单”和“处理定货”(图 2.7)。

当对数据流图分层细化时必须保持信息连续性，也就是说，当把一个处理分解为一系列处理时，分解前和分解后的输入输出数据流必须相同。

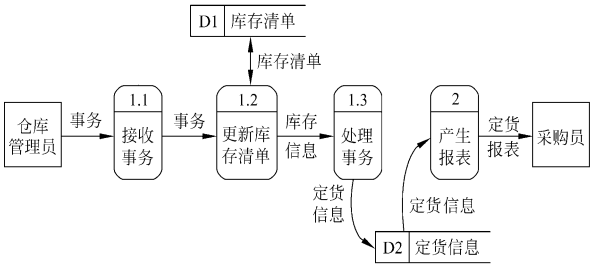


图 2.7 把处理事务的功能进一步分解后的数据流图

2.4.3 命名

数据流图中每个成分的命名是否恰当，直接影响数据流图的可理解性。

1. 为数据流(或数据存储)命名

(1) 名字应代表整个数据流(或数据存储)的内容，而不是仅仅反映它的某些成分。

(2) 不要使用空洞的、缺乏具体含义的名字(如“数据”、“信息”、“输入”之类)。

(3) 如果在为某个数据流(或数据存储)起名字时遇到了困难，则很可能是因为对数据流图分解不恰当造成的，应该试试重新分解，看是否能克服这个困难。

2. 为处理命名

(1) 通常先为数据流命名，然后再为相关联的处理命名。

(2) 名字应该反映整个处理的功能，而不是部分功能。

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

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

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

2.4.4 用途

画数据流图的基本目的是利用它作为交流信息的工具。分析员把他对现有系统的认识或对目标系统的设想用数据流图描绘出来，供有关人员审查确认。

数据流图应该分层，并且在把功能级数据流图细化后得到的处理超过 9 个时，应该采用画分图的办法，也就是把每个主要功能都细化为一张数据流分图，而原有的功能级数据流图用来描绘系统的整体逻辑概貌。

数据流图的另一个主要用途是作为分析和设计的工具。分析员在研究现有的系统时常用系统流程图表达他对这个系统的认识，这种描绘方法形象具体，比较容易验证它的正确性；

数据流图着重描绘系统所完成的功能而不是系统的物理实现方案。

当用数据流图辅助物理系统的设计时，以图中不同处理的定时要求为指南，能够在数据流图上画出许多组自动化边界，每组自动化边界可能意味着一个不同的物理系统，因此可以根据系统的逻辑模型考虑系统的物理实现。例如，考虑图 2.7，事务随时可能发生，因此处理 1.1(“接收事务”)必须是联机的；采购员每天需要一次定货报表，因此处理 2(“产生报表”)应该以批量方式进行。问题描述并没有对其他处理施加限制，例如，可以联机地接收事务并放入队列中，然而更新库存清单、处理定货和产生报表以批量方式进行(图 2.8)。当然，这种方案需要增加一个数据存储以存放事务数据。

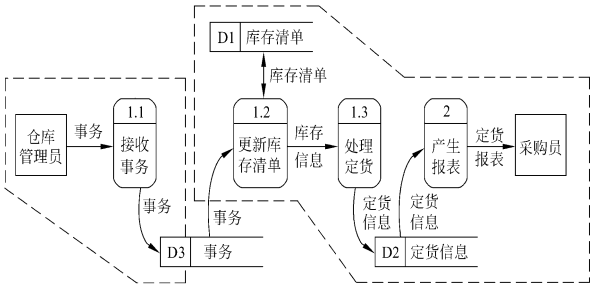


图 2.8 这种划分自动化边界的方法暗示以批量方式更新库存清单

改变自动化边界，把处理 1.1，1.2 和 1.3 放在同一个边界内(图 2.9)，这个系统将联机地接收事务、更新库存清单和处理定货及输出定货信息；然而处理 2 将以批量方式产生定货报表。还能设想出建立自动化边界的其他方案吗?如果把处理 1.1 和处理 1.2 放在一个自动化边界内，把处理 1.3 和处理 2 放在另一个边界内，意味着什么样的

物理系统呢?

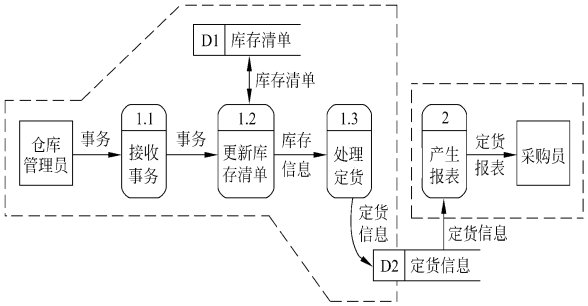


图 2.9 另一种划分自动化边界的方法建议以联机方式更新库存清单

2.5 数据字典

数据字典是关于数据的信息的集合，也就是对数据流图中包含的所有元素的定义的集合。

数据字典的作用是在软件分析和设计的过程中给人提供关于数据的描述信息。

数据流图和数据字典共同构成系统的逻辑模型，没有数据字典数据流图就不严格，然而没有数据流图数据字典也难于发挥作用。只有数据流图和对数据流图中每个元素的精确定义放在一起，才能共同构成系统的规格说明。

2.5.1 数据字典的内容

一般说来，数据字典应该由对下列 3 类元素的定义组成:

(1) 数据流

(2) 数据流分量(即数据元素)

(3) 数据存储

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

数据元素的别名就是该元素的其他等价的名字，出现别名主要有下述 3 个原因:

(1) 对于同样的数据，不同的用户使用了不同的名字;

(2) 一个分析员在不同时期对同一个数据使用了不同的名字;

(3) 两个分析员分别分析同一个数据流时，使用了不同的名字。

虽然应该尽量减少出现别名，但是不可能完全消除别名。

2.5.2 定义数据的方法

数据字典中的定义就是对数据自顶向下的分解。

一般说来，当分解到不需要进一步定义，每个和工程有关的人也都清楚其含义的元素时，这种分解过程就完成了。

由数据元素组成数据的方式有下述四种类型:

(1) 顺序 即以确定次序连接两个或多个分量;

(2) 选择 即从两个或多个可能的元素中选取一个;

(3) 重复 即把指定的分量重复零次或多次。

(4) 可选 即一个分量是可有可无的(重复零次或一次)。

为了说明重复次数，重复算符通常和重复次数的上下限同时使用(当上下限相同时表示重复次数固定)。

为了更加清晰简洁，建议采用下列符号:

=意思是等价于(或定义为);

+意思是和(连接两个分量);

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

{ }意思是重复(即，重复花括弧内的分量);

( )意思是可选(即，圆括弧里的分量可有可无)。

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

下面举例说明上述定义数据的符号的使用方法：某程序设计语言规定，用户说明的标识符是长度不超过 8 个字符的字符串，其中第一个字符必须是字母字符，随后的字符既可以是字母字符也可以是数字字符。使用上面讲过的符号，我们可以像下面那样定义标识符:

标识符 = 字母字符 + 字母数字串

字母数字串 = 0 {字母或数字} 7

字母或数字 = [字母字符 | 数字字符]

字母字符 = a .. z

数字字符 = 0 .. 9

2.5.3 数据字典的用途

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

数据字典是开发数据库的第一步，而且是很有价值的一步。

2.5.4 数据字典的实现

目前，数据字典几乎总是作为 CASE“结构化分析与设计工具”的一部分实现的。在开发大型软件系统的过程中，数据字典的规模和复杂程度迅速增加，人工维护数据字典几乎是不可能的。

如果在开发小型软件系统时暂时没有数据字典处理程序，可采用卡片形式书写数据字典，每张卡片上保存描述一个数据的信息。这样做更新和修改起来比较方便，而且能单独处理描述每个数据的信息。每张卡片上主要应该包含下述这样一些信息:

名字、别名、描述、定义、位置。

2.6 成本/效益分析

开发一个软件系统是一种投资，期望将来获得更大的经济效益。经济效益通常表现为减少运行费用或(和)增加收入。但是，投资开发新系统往往要冒一定风险，系统的开发成本可能比预计的高，效益可能比预期的低。效益分析的目的正是要从经济角度分析开发一个特定的新系统是否划算，从而帮助客户组织的负责人正确地作出是否投资于这项开发工程的决定。

为了对比成本和效益，首先需要估计它们的数量。

2.6.1 成本估计

软件开发成本主要表现为人力消耗(乘以平均工资则得到开发费用)。成本估计不是精确的科学，因此应该使用几种不同的估计技术以便相互校验。简单介绍 3 种估算技术。

1. 代码行技术

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

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

2. 任务分解技术

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

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

典型环境下各个开发阶段需要使用的人力的百分比大致如表 2.2（见书 40 页）所示。当然，应该针对每个开发工程的具体特点，并且参照以往的经验尽可能准确地估计每个阶段实际需要使用的人力。

3. 自动估计成本技术

采用自动估计成本的软件工具可以减轻人的劳动，并且使得估计的结果更客观。但是，采用这种技术必须有长期搜集的大量历史数据为基础，并且需要有良好的数据库系统支持。

2.6.2 成本/效益分析的方法

成本/效益分析的第一步是估计开发成本、运行费用和新系统将带来的经济效益。运行费用取决于系统的操作费用(操作员人数，工作时间，消耗的物资等等)和维护费用。系统的经济效益等于因使用新系统而增加的收入加上使用新系统可以节省的运行费用。因为运行费用和经济效益两者在软件的整个生命周期内都存在，总的效益和生命周期的长度有关，所以应该合理地估计软件的寿命。

虽然许多系统在开发时预期生命周期长达 10 年以上，但是时间越长系统被废弃的可能性也越大，为了保险起见，以后在进行成本/效益分析时一律假设生命周期为 5 年。

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

1. 货币的时间价值

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

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

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

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

例如，修改一个已有的库存清单系统，使它能在每天送给采购员一份定货报表。修改已有的库存清单程序并且编写产生报表的程序，估计共需 5000 元；系统修改后能及时定货将消除零件短缺问题，估计因此每年可以节省 2500 元，5 年共可节省 12500 元。但是，不能简单地把 5000 元和 12500 元相比较，因为前者是现在投资的钱，后者是若干年以后节省的钱。

假定年利率为 12%，利用上面计算货币现在价值的公式可以算出修改库存清单系统后每年预计节省的钱的现在价值，如表 2.3（见书 41 页）所示。

2. 投资回收期

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

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

3. 纯收入

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

4. 投资回收率

把资金存入银行或贷给其他企业能够获得利息，通常用年利率衡量利息多少。类似地也可以计算投资回收率，用它衡量投资效益的大小，并且可以把它和年利率相比较，在

衡量工程的经济效益时，它是最重要的参考数据。
已知现在的投资额，并且已经估计出将来每年可以获得的经济效益，那么，给定软件的使用寿命之后，怎样计算投资回收率呢?设想把数量等于投资额的资金存入银行，每年年底从银行取回的钱等于系统每年预期可以获得的效益，在时间等于系统寿命时，正好把在银行中的存款全部取光，那么，年利率等于多少呢?这个假想的年利率就等于投资回收率。

2.7　小结

可行性研究进一步探讨问题定义阶段所确定的问题是否有可行的解。在对问题正确定义的基础上，通过分析问题，导出试探性的解，然后复查并修正问题定义，再次分析问题，改进提出的解法……。经过定义问题、分析问题、提出解法的反复过程，最终提出一个符合系统目标的高层次的逻辑模型。然后根据系统的这个逻辑模型设想各种可能的物理系统，并且从技术、经济和操作等各方面分析这些物理系统的可行性。最后，系统分析员提出一个推荐的行动方针，提交用户和客户组织负责人审查批准。
在表达分析员对现有系统的认识和描绘未来的物理系统的设想时，系统流程图是一个很好的工具。系统流程图实质上是物理数据流图，它描绘组成系统的主要物理元素以及信息在这些元素间流动和处理的情况。
数据流图的基本符号只有 4 种，它是描绘系统逻辑模型的极好工具。通常数据字典和数据流图共同构成系统的逻辑模型。
成本/效益分析是可行性研究的一项重要内容，是客户组织负责人从经济角度判断是否继续投资于这项工程的主要依据。

第 3 章　需求分析

需求分析是软件定义时期的最后一个阶段，它的基本任务是准确地回答“系统必须做什么?”这个问题。
需求分析的任务还不是确定系统怎样完成它的工作，而仅仅是确定系统必须完成哪些工作，也就是对目标系统提出完整、准确、清晰、具体的要求。
在需求分析阶段结束之前，系统分析员应该写出软件需求规格说明书，以书面形式准确地描述软件需求。
在分析软件需求和书写软件需求规格说明书的过程中，分析员和用户都起着关键的、必不可少的作用。用户必须把他们对软件的需求尽量准确、具体地描述出来；分析员在需求分析开始时他们对用户的需求并不十分清楚，必须通过与用户沟通获取用户对软件的需求。
需求分析和规格说明是一项十分艰巨复杂的工作。用户与分析员之间需要沟通的内容非常多，在双方交流信息的过程中很容易出现误解或遗漏，也可能存在二义性。因此，不仅在整个需求分析过程中应该采用行之有效的方法，必须严格审查验证需求分析的结果。
目前有许多需求分析的结构化分析方法，这些分析方法都遵守下述准则：
(1) 必须理解并描述问题的信息域，根据这条准则应该建立数据模型。
(2) 必须定义软件应完成的功能，这条准则要求建立功能模型。
(3) 必须描述作为外部事件结果的软件行为，这条准则要求建立行为模型。
(4) 必须对描述信息、功能和行为的模型进行分解，用层次的方式展示细节。

3.1　需求分析的任务

3.1.1　确定对系统的综合要求

1. 功能需求

这方面的需求指定系统必须提供的服务。通过需求分析应该划分出系统必须完成的所有功能。

2. 性能需求

性能需求指定系统必须满足的定时约束或容量约束，通常包括速度(响应时间)、信息量速率、主存容量、磁盘容量、安全性等方面的需求。

3. 可靠性和可用性需求

可靠性需求定量地指定系统的可靠性。
可用性与可靠性密切相关，它量化了用户可以使用系统的程度。

4. 出错处理需求

这类需求说明系统对环境错误应该怎样响应。“出错处理”指的是当应用系统发现它自己犯下一个错误时所采取的行动。但是，应该有选择地提出这类出错处理需求，对应用系统本身错误的检测应该仅限于系统的关键部分，而且应该尽可能少。

5. 接口需求

接口需求描述应用系统与它的环境通信的格式。常见的接口需求有：用户接口需求；硬件接口需求；软件接口需求；通信接口需求。

6. 约束

设计约束或实现约束描述在设计或实现应用系统时应遵守的限制条件。常见的约束有：精度；工具和语言约束；设计约束；应该使用的标准；应该使用的硬件平台。

7. 逆向需求

8. 将来可能提出的要求

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

3.1.2　分析系统的数据要求

任何一个软件系统本质上都是信息处理系统，系统必须处理的信息和系统应该产生的信息在很大程度上决定了系统的面貌。分析系统的数据要求通常采用建立数据模型的方法。

为了提高可理解性，常常利用图形工具辅助描绘数据结构。常用的图形工具有层次方框图和 Warnier 图。

3.1.3　导出系统的逻辑模型

综合上述两项分析的结果可以导出系统的详细的逻辑模型，通常用数据流图、实体-联系图、状态转换图、数据字典和主要的处理算法描述这个逻辑模型。

3.1.4　修正系统开发计划

根据在分析过程中获得的对系统的更深入更具体的了解，可以比较准确地估计系统的成本和进度，修正以前制定的开发计划。

3.2　与用户沟通获取需求的方法

3.2.1　访谈

访谈是最早开始使用的获取用户需求的技术，也是迄今为止仍然广泛使用的需求分析技术。

访谈有两种基本形式，分别是正式的和非正式的访谈。正式访谈时，系统分析员将提出一些事先准备好的具体问题。在非正式访谈中，分析员将提出一些用户可以自由回答的开放性问题，以鼓励被访问人员说出自己的想法。

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

在访问用户的过程中使用情景分析技术往往非常有效。所谓情景分析就是对用户将来使用目标系统解决某个具体问题的方法和结果进行分析。

3.2.2　面向数据流自顶向下求精

结构化分析方法就是面向数据流自顶向下逐步求精进行需求分析的方法。通过可行性研究已经得出了目标系统的高层数据流图，需求分析的目标之一就是把数据流和数据存储定义到元素级。为了达到这个目标，通常从数据流图的输出端着手分析，这是因为系统的基本功能是产生这些输出，输出数据决定了系统必须具有的最基本的组成元素。沿数据流图从输出端往输入端回溯，能够确定每个数据元素的来源，与此同时也就初步定义了有关的算法。

通常把分析过程中得到的有关数据元素的信息记录在数据字典中，把对算法的简明描述记录在 IPO 图中。通过分析而补充的数据流、数据存储和处理，应该添加到数据流图的适当位置上。

从输入端开始，分析员借助数据流图、数据字典和 IPO 图向用户解释输入数据是怎样一步一步地转变成输出数据的。这些解释集中反映了通过前面的分析工作分析员所获得的对目标系统的认识。

用户应该注意倾听分析员的报告，并及时纠正和补充分析员的认识。复查过程验证了已知的元素，补充了未知的元素，填补了文档中的空白。

随着分析过程的进展，经过问题和解答的反复循环，分析员越来越深入具体地定义了目标系统，最终得到对系统数据和功能要求的满意了解。图 3.1 粗略地概括了上述分析过程。

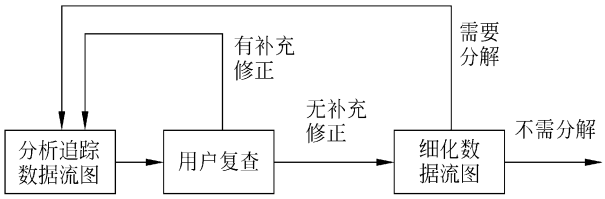


图 3.1　面向数据流自顶向下求精过程

3.3　分析建模与规格说明

3.3.1　分析建模

所谓模型，就是为了理解事物而对事物做出的一种抽象，是对事物的一种无歧义的书面描述。通常，模型由一组图形符号和组织这些符号的规则组成。

结构化分析实质上是一种创建模型的活动。为了开发出复杂的软件系统，系统分析员应该从不同角度抽象出目标系统的特性，使用精确的表示方法构造系统的模型，验证模型是否满足用户对目标系统的需求，并在设计过程中逐渐把和实现有关的细节加进模型中，直至最终用程序实现模型。

根据结构化分析准则，需求分析过程应该建立 3 种模型，它们分别是数据模型、功能模型和行为模型。

3.3.2　软件需求规格说明

需求分析阶段还应该写出软件需求规格说明书，它是需求分析得出的最主要的文档。

通常用自然语言完整、准确、具体地描述系统的数据要求、功能需求、性能需求、可靠性和可用性要求、出错处理需求、接口需求、约束以及将来可能提出的要求。

3.4　实体-联系图

为了把用户的数据要求清楚、准确地描述出来，系统分析员通常建立一个概念性的数据模型(也称为信息模型)。概念性数据模型是一种面向问题的数据模型，是按照用户的观点对数据建立的模型。它描述了从用户角度看到的数据，它反映了用户的现实环境，而且与在软件系统中的实现方法无关。

数据模型中包含 3 种相互关联的信息：数据对象、数据对象的属性及数据对象彼此间相互连接的关系。

3.4.1　数据对象

数据对象是对软件必须理解的复合信息的抽象。所谓复合信息是指具有一系列不同性质或属性的事物。

数据对象可以是外部实体、事物、行为、事件、角色、单位、地点或结构等。总之，可以由一组属性来定义的实体都可以被认为是数据对象。

3.4.2　属性

属性定义了数据对象的性质。必须把一个或多个属性定义为“标识符”，也就是说，当我们希望找到数据对象的一个实例时，用标识符属性作为“关键字”(简称为

“键”)。

应该根据对所要解决的问题的理解，来确定特定数据对象的一组合适的属性。

3.4.3　联系

数据对象彼此之间相互连接的方式称为联系，也称为关系。联系可分为以下 3 种类型：

(1) 一对一联系(1 : 1)

(2) 一对多联系(1 : N)

(3) 多对多联系(M : N)

3.4.4　实体-联系图的符号

使用实体-联系图(entity-relationship diagram)来建立数据模型。用 ER 图描绘的数据模型称为 ER 模型。

ER 图中包含了实体(即数据对象)、关系和属性等 3 种基本成分，通常用矩形框代表实体，用连接相关实体的菱形框表示关系，用椭圆形或圆角矩形表示实体(或关系)的属性，并用直线把实体(或关系)与其属性连接起来。例如，图 3.2 是某学校教学管理的 ER 图。

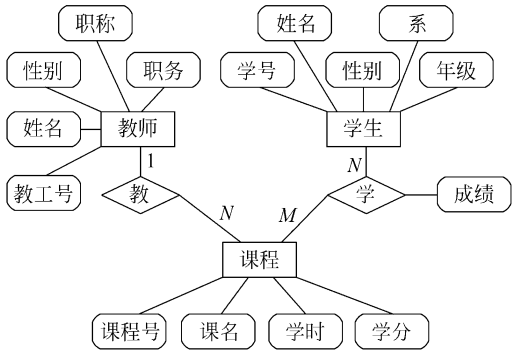


图 3.2 某校教学管理 ER 图

3.5　数据规范化

软件系统经常使用各种长期保存的信息，这些信息通常以一定方式组织并存储在数据库或文件中，为减少数据冗余，避免出现插入异常或删除异常，简化修改数据的过程，通常需要把数据结构规范化。

通常用“范式(normal forms)”定义消除数据冗余的程度。第一范式(1 NF)数据冗余程度最大，第五范式(5 NF)数据冗余程度最小。从实用角度看来，在大多数场合选用第三范式都比较恰当。

(1) 第一范式每个属性值都必须是原子值，即仅仅是一个简单值而不含内部结构。

(2) 第二范式满足第一范式条件，而且每个非关键字属性都由整个关键字决定(而不是由关键字的一部分来决定)。

(3) 第三范式符合第二范式条件，每个非关键字属性都仅由关键字决定，而且一个非关键字属性不能仅仅是对另一个非关键字属性的进一步描述(即一个非关键字属性值不依赖于另一个非关键字属性值)。

3.6　状态转换图

状态转换图(简称为状态图)通过描绘系统的状态及引起系统状态转换的事件，来表示系统的行为。

此外，状态图还指明了作为特定事件的结果系统将做哪些动作(例如，处理数据)。因此，状态图提供了行为建模机制，可以满足分析准则的要求。

3.6.1　状态

状态是任何可以被观察到的系统行为模式，一个状态代表系统的一种行为模式。状态规定了系统对事件的响应方式。系统对事件的响应，既可以是做一个(或一系列)动作，也可以是仅仅改变系统本身的状态，还可以是既改变状态又做动作。

在状态图中定义的状态主要有：初态(即初始状态)、终态(即最终状态)和中间状态。在一张状态图中只能有一个初态，而终态则可以有 0 至多个。

状态图既可以表示系统循环运行过程，也可以表示系统单程生命期。当描绘循环运行过程时，通常并不关心循环是怎样启动的。当描绘单程生命期时，需要标明初始状态(系统启动时进入初始状态)和最终状态(系统运行结束时到达最终状态)。

3.6.2　事件

事件是在某个特定时刻发生的事情，它是对引起系统做动作或(和)从一个状态转换到另一个状态的外界事件的抽象。例如，内部时钟表明某个规定的时间段已经过去，用户移动或点击鼠标等都是事件。简而言之，事件就是引起系统做动作或(和)转换状态的控制信息。

3.6.3　符号

在状态图中，初态用实心圆表示，终态用一对同心圆(内圆为实心圆)表示。

中间状态用圆角矩形表示，可以用两条水平横线把它分成上、中、下 3 个部分。上面部分为状态的名称，这部分是必须有的；中间部分为状态变量的名字和值，这部分是可选的；下面部分是活动表，这部分也是可选的。

活动表的语法格式如下：

事件名(参数表)/动作表达式

其中，“事件名”可以是任何事件的名称。在活动表中经常使用下述 3 种标准事件：entry，exit 和 do。entry 事件指定进入该状态的动作，exit 事件指定退出该状态的动作，而 do 事件则指定在该状态下的动作。需要时可以为事件指定参数表。活动表中的动作表达式描述应做的具体动作。

状态图中两个状态之间带箭头的连线称为状态转换，箭头指明了转换方向。状态变迁通常是由事件触发的，在这种情况下应在表示状态转换的箭头线上标出触发转换的事件表达式；如果在箭头线上未标明事件，则表示在源状态的内部活动执行完之后自动触发转换。

事件表达式的语法如下：

事件说明［守卫条件］ / 动作表达式

其中，事件说明的语法为：事件名(参数表)。

守卫条件是一个布尔表达式。如果同时使用事件说明和守

卫条件，则当且仅当事件发生且布尔表达式为真时，状态转换才发生。如果只有守卫条件没有事件说明，则只要守卫条件为真状态转换就发生。

动作表达式是一个过程表达式，当状态转换开始时执行该表达式。

图 3.3 给出了状态图中使用的主要符号。

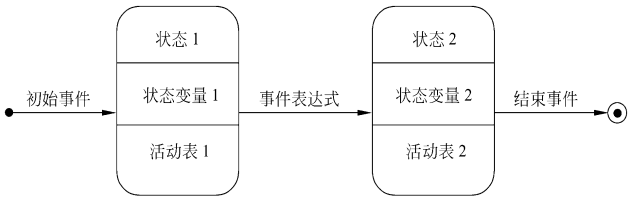
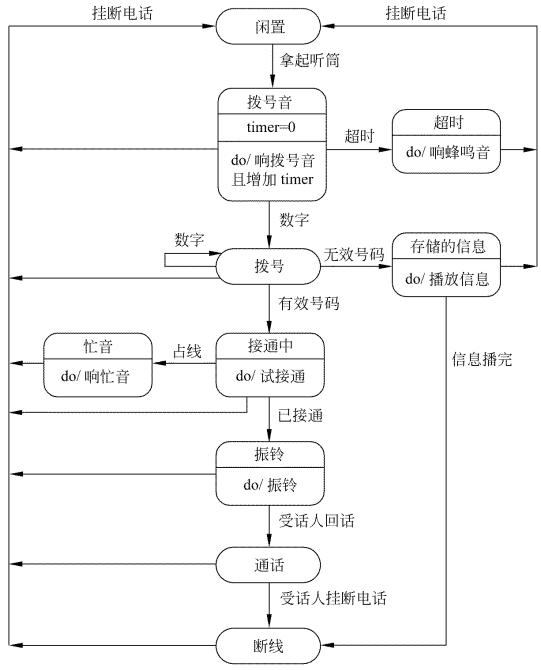


图 3.3 状态图中使用的主要符号

3.6.4 例子

图 3.4 是电话系统的状态图。

图中表明，没有人打电话时电话处于闲置状态；有人拿起听筒则进入拨号音状态，到达这个状态后，电话的行为是响起拨号音并计时；这时如果拿起听筒的人改变主意不想打了，他把听筒放下(挂断)，电话重又回到闲置状态；如果拿起听筒很长时间不拨号(超时)，则进入超时状态；……。



3.6.4 例子

3.7 其他图形工具

3.7.1 层次方框图

层次方框图用树形结构的一系列多层次的矩形框描绘数据的层次结构。

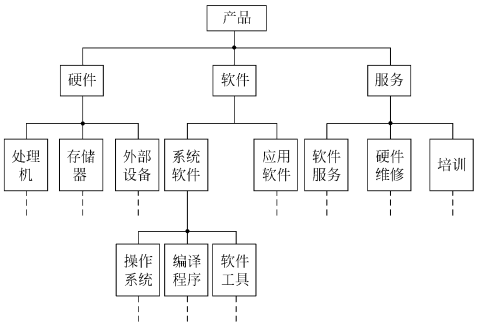


图 3.5 层次方框图的一个例子

3.7.2 Warnier 图

法国计算机科学家 Warnier 提出了表示信息层次结构的另外一种图形工具。Warnier 图也用树形结构描绘信息。

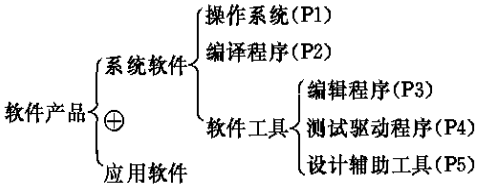


图 3.6 Warnier 图的一个例子

3.7.3 IPO 图

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

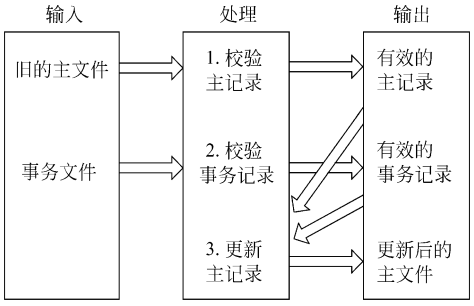


图 3.7 IPO 图的一个例子图

IPO 图的基本形式是在左边的框中列出有关的输入数据，在中间的框内列出主要的处理，在右边的框内列出产生的输出数据。处理框中列出处理的次序暗示了执行的顺序，在 IPO 图中用箭头清楚地指出数据通信的情况。图 3.7 是一个主文件更新的例子。

一种改进的 IPO 图(也称为 IPO 表)，这种图中包含某些附加的信息，在软件设计过程中将比原始的 IPO 图更有用。在需求分析阶段可以使用 IPO 图简略地描述系统的主要算法(即数据流图中各个处理的基本算法)。

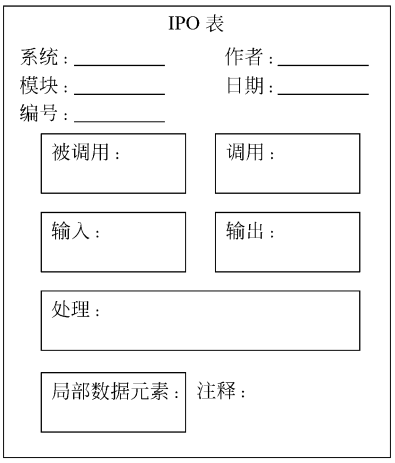


图 3.8 改进的 IPO 图的形式

3.8 验证软件需求

3.8.1 从哪些方面验证软件需求的正确性

需求分析阶段的工作结果是开发软件系统的重要基础。为了提高软件质量，确保软件开发成功，降低软件开发成本，必须严格验证需求的正确性。一般说来，应该从下述 4 个方面进行验证：

(1) 一致性：所有需求必须是一致的，任何一条需求不能和其他需求互相矛盾。

(2) 完整性：需求必须是完整的，规格说明书应该包括用户需要的每一个功能或性能。

(3) 现实性：指定的需求应该是用现有的硬件技术和软件技术基本上可以实现的。

(4) 有效性：必须证明需求是正确有效的，确实能解决用户面对的问题。

3.8.2 验证软件需求的方法

1. 验证需求的一致性

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

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

2. 验证需求的现实性

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

3. 验证需求的完整性和有效性

只有目标系统的用户才真正知道软件需求规格说明书是否完整、准确地描述了他们的需求。因此，检验需求的完整性，特别是证明系统确实满足用户的实际需要，只有在用户的密切合作下才能完成。

3.8.3 用于需求分析的软件工具

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

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

(2) 使用这个软件工具能够导出详细的文档；

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

(4) 使用这个软件工具之后，应该能够改进通信状况。

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

1977 年美国密执安大学开发了 PSL/PSA(问题陈述语言/问题陈述分析程序)系统。这个系统是 CADSAT（计算机辅助设计和规格说明分析工具）的一部分，它的基本结构类似于 RSL。

3.9 小结

传统软件工程方法学使用结构化分析技术，完成分析用户需求的工作。需求分析是发现、求精、建模、规格说明和复审的过程。需求分析的第一步是进一步了解用户当前所处的情况，发现用户所面临的问题和对目标系统的基本需求；接下来应该与用户深入交流，对用户的基本需求反复细化逐步求精，以得出对目标系统的完整、准确和具体的需求。具体地说，应该确定系统必须具有的功能、性能、可靠性和可用性，必须实现的出错处理需求、接口需求，必须满足的约束条件，并且预测系统的发展前景。

为了详细地了解并正确地理解用户的需求，必须使用适当方法与用户沟通。访谈是与用户通信的历史悠久的技术，至今仍被许多系统分析员采用。从可行性研究阶段得到的数据流图出发，在用户的协助下面向数据流自顶向下逐步求精，也是与用户沟通获取需求的一个有效的方法。为了促使用户与分析员齐心协力共同分析需求，人们研究出一种面向团队的需求收集法，称为简易的应用规格说明技术，现在这种技术已经成为信息系统领域使用的主流技术。实践表明，快速建立软件原型是最准确、最有效和最强大的需求分析技术。

为了更好地理解问题，人们常常采用建立模型的方法，结构化分析实质上就是一种建模活动，在需求分析阶段通常建立数据模型、功能模型和行为模型。

除了创建分析模型之外，在需求分析阶段还应该写出软件需求规格说明书，经过严格评审并得到用户确认之后，作为这个阶段的最终成果。通常主要从一致性、完整性、现实性和有效性等 4 个方面复审软件需求规格说明书。

多数人习惯于使用实体-联系图建立数据模型，使用数据流图建立功能模型，使用状态图建立行为模型。读者应该掌握这些图形的基本符号，并能正确地使用这些符号建立软件系统的模型。

数据字典描述在数据模型、功能模型和行为模型中出现的数据对象及控制信息的特性，给出它们的准确定义。因此，数据字典成为把 3 种分析模型粘合在一起的“粘合剂”，是分析模型的“核心”。

为了提高可理解性，还可以用层次方框图或 Warnier 图等图形工具辅助描绘系统中的数据结构。为了减少冗余、简化修改步骤，往往需要规范数据的存储结构。

算法也是重要的，分析的基本目的是确定系统必须做什么。概括地说，任何一个计算机系统的基本功能都是把输入数据转变成输出信息，算法定义了转变的规则。因此，没有对算法的了解就不能确切知道系统的功能。IPO 图是描述算法的有效工具。

第 5 章　总体设计

总体设计的基本目的就是回答“概括地说，系统应该如何实现?”这个问题，因此，总体设计又称为概要设计或初步设计。通过这个阶段的工作将划分出组成系统的物理元素——程序、文件、数据库、人工过程和文档等等，但是每个物理元素仍然处于黑盒子级，这些黑盒子里的具体内容将在以后仔细设计。总体设计阶段的另一项重要任务是设计软件的结构，也就是要确定系统中每个程序是由哪些模块组成的，以及这些模块相互间的关系。

总体设计设计软件结构。设计出初步的软件结构后还要多方改进，从而得到更合理的结构，进行必要的数据库设计，确定测试要求并且制定测试计划。

在详细设计之前先进行总体设计的必要性：可以站在全局高度上，花较少成本，从较抽象的层次上分析对比多种可能的系统实现方案和软件结构，从中选出最佳方案和最合理的软件结构，从而用较低成本开发出较高质量的软件系统。

5.1　设计过程

总体设计过程通常由两个主要阶段组成：系统设计阶段，确定系统的具体实现方案；结构设计阶段，确定软件结构。典型的总体设计过程包括下述 9 个步骤：

1. 设想供选择的方案
2. 选取合理的方案
3. 推荐最佳方案
4. 功能分解

通常分为两个阶段完成：首先进行结构设计，然后进行过程设计。

结构设计确定程序由哪些模块组成，以及这些模块之间的关系；

过程设计确定每个模块的处理过程。

结构设计是总体设计阶段的任务，过程设计是详细设计阶段的任务。

为确定软件结构，首先需要从实现角度把复杂的功能进一步分解。分析员结合算法描述仔细分析数据流图中的每个处理，如果一个处理的功能过于复杂，必须把它的功能适当地分解成一系列比较简单的功能。一般说来，经过分解之后应该使每个功能对大多数程序员而言都是明显易懂的。功能分解导致数据流图的进一步细化，同时还应该用 IPO 图或其他适当的工具简要描述细化后每个处理的算法。

5. 设计软件结构

通常程序中的一个模块完成一个适当的子功能。应该把模块组织成良好的层次系统，顶层模块调用它的下层模块以实现程序的完整功能，每个下层模块再调用更下层的模块，从而完成程序的一个子功能，最下层的模块完成最具体的功能。软件的逻辑结构可以用层次图或结构图来描绘。

6. 设计数据库

对于需要使用数据库的那些应用系统，软件工程师应该在需求分析阶段所确定的系统数据需求的基础上，进一步设计数据库。

7. 制定测试计划

在软件开发的早期阶段考虑测试问题，能促使软件设计人员在设计时注意提高软件的可测试性。

8. 书写文档

应该用正式的文档记录总体设计的结果，在这个阶段应该完成的文档通常有下述几种：

(1) 系统说明：主要内容包括用系统流程图描绘的系统构成方案，组成系统的物理元素清单，成本/效益分析；对最佳方案的概括描述，精化的数据流图，用层次图或结构图描绘的软件结构，用 IPO 图或其他工具简要描述的各个模块的算法，模块间的接口关系，以及需求、功能和模块三者之间的交叉参照关系等等。

(2) 用户手册：根据总体设计阶段的结果，修改更正在需求分析阶段产生的初步的用户手册。

(3) 测试计划：包括测试策略，测试方案，预期的测试结果，测试进度计划等等。

(4) 详细的实现计划

(5) 数据库设计结果

9. 审查和复审

最后应该对总体设计的结果进行严格的技术审查，在技术审查通过之后再由使用部门的负责人从管理角度进行复审。

5.2 设计原理

5.2.1 模块化

模块是由边界元素限定的相邻程序元素（例如，数据说明，可执行的语句）的序列，而且有一个总体标识符代表它。

按照模块的定义，过程、函数、子程序和宏等，都可作为模块。面向对象方法学中的对象是模块，对象内的方法也是模块。模块是构成程序的基本构件。

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

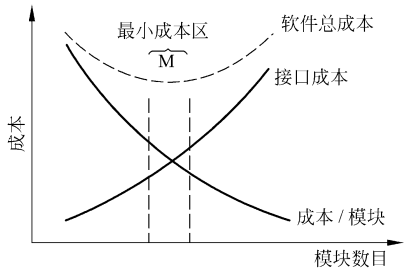


图 5.1 模块化和软件成本

采用模块化原理可以使软件结构清晰，不仅容易设计也容易阅读和理解。因为程序错误通常局限在有关的模块及它们之间的接口中，所以模块化使软件容易测试和调试，因而有助于提高软件的可靠性。因为变动往往只涉及少数几个模块，所以模块化能够提高软件的可修改性。模块化也有助于软件开发工程的组织管理。

5.2.2 抽象

抽象是通过对现实世界中一类事物或概念经过高度概括总结抽取出其共性的过程。

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

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

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

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

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

5.2.3 逐步求精

逐步求精是人类解决复杂问题时采用的基本方法，也是许多软件工程技术的基础。可以把逐步求精定义为：“为了能集中精力解决主要问题而尽量推迟对问题细节的考虑。”

求精方法的作用在于，它能帮助软件工程师把精力集中在与当前开发阶段最相关的那些方面上，而忽略那些对整体解决方案来说虽然是必要的，然而目前还不需要考虑的细节，这些细节将留到以后再考虑。

抽象与求精是一对互补的概念。抽象使得设计者能够说明过程和数据，同时却忽略低层细节。求精则帮助设计者在设计过程中逐步揭示出低层细节。这两个概念都有助于设计者在设计演化过程中创造出完整的设计模型。

5.2.4 信息隐藏和局部化

信息隐藏指出：应该这样设计和确定模块，使得一个模块内包含的信息对于不需要这些信息的模块来说，是不能访问的。

局部化是指把一些关系密切的软件元素物理地放得彼此靠近。局部化有助于实现信息隐藏。

局部化的概念和信息隐藏概念是密切相关的。

5.2.5 模块独立

模块独立的概念是模块化、抽象、信息隐藏和局部化概念的直接结果。

开发具有独立功能而且和其他模块之间没有过多的相互作用的模块，就可以做到模块独立。

为什么模块的独立性很重要呢?主要有两条理由：第一，有效的模块化软件比较容易开发出来。这是由于能够分割功能而且接口可以简化，当许多人分工合作开发同一个软件时，这个优点尤其重要。

第二，独立的模块比较容易测试和维护。这是因为相对说来，修改设计和程序需要的工作量比较小，错误传播范围小，需要扩充功能时能够“插入”代码。总之，模块独立是良好设计的关键，而设计又是决定软件质量的关键环节。

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

1. 耦合

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

在软件设计中应该追求尽可能松散耦合的系统。易于开发、测试或维护这样的系统。此外，由于模块间联系简单，发生在一处的错误传播到整个系统的可能性就很小。因此，模块间的耦合程度强烈影响系统的可理解性、可测试性、可靠性和可维护性。

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

根据模块间联系的紧密程度，把耦合分为：

a. 数据耦合
b. 控制耦合
c. 特征耦合
d. 公共环境耦合
e. 内容耦合

如果两个模块彼此间通过参数交换信息，而且交换的信息仅仅是数据，那么这种耦合称为数据耦合。如果传递的信息中有控制信息，则这种耦合称为控制耦合。

数据耦合是低耦合。系统中至少必须存在这种耦合，因为只有当某些模块的输出数据作为另一些模块的输入数据时，系统才能完成有价值的功能。一般说来，一个系统内可以只包含数据耦合。

控制耦合是中等程度的耦合，它增加了系统的复杂程度。控制耦合往往是多余的，在把模块适当分解之后通常可以用数据耦合代替它。

如果被调用的模块需要使用作为参数传递进来的数据结构中的所有元素，那么，把整个数据结构作为参数传递就是完全正确的。但是，当把整个数据结构作为参数传递而被调用的模块只需要使用其中一部分数据元素时，就出现了特征耦合。在这种情况下，被调用的模块可以使用的数据多于它确实需要的数据，这将导致对数据的访问失去控制。

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

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

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

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

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

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

一个模块访问另一个模块的内部数据；

一个模块不通过正常入口而转到另一个模块的内部；

两个模块有一部分程序代码重叠；

一个模块有多个入口(这意味着一个模块有几种功能)。

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

总之，耦合是影响软件复杂程度的一个重要因素。应该采取下述设计原则：

尽量使用数据耦合，少用控制耦合和特征耦合，限制公共环境耦合的范围，完全不用内容耦合。

2. 内聚

内聚标志一个模块内各个元素彼此结合的紧密程度，它是信息隐藏和局部化概念的自然扩展。简单地说，理想内聚的模块只做一件事情。

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

根据模块内联系的紧密程度，把内聚分为：

a. 偶然内聚
b. 逻辑内聚
c. 时间内聚
d. 过程内聚
e. 通信内聚
f. 顺序内聚
g. 功能内聚

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

低内聚有如下几类：

如果一个模块完成一组任务，这些任务彼此间即使有关系，关系也是很松散的，就叫做偶然内聚。有时在写完一个程序之后，发现一组语句在两处或多处出现，于是把这些语句作为一个模块以节省内存，这样就出现了偶然内聚的模块。

如果一个模块完成的任务在逻辑上属于相同或相似的一类，则称为逻辑内聚。

如果一个模块包含的任务必须在同一段时间内执行，就叫时间内聚。

在偶然内聚的模块中，各种元素之间没有实质性联系，很可能在一种应用场合需要修改这个模块，在另一种应用场合又不允许这种修改，从而陷入困境。

在逻辑内聚的模块中，不同功能混在一起，合用部分程序代码，即使局部功能的修改有时也会影响全局。因此，这

类模块的修改也比较困难。

时间关系在一定程度上反映了程序的某些实质，所以时间内聚比逻辑内聚好一些。

中内聚主要有两类：

如果一个模块内的处理元素是相关的，而且必须以特定次序执行，则称为过程内聚。使用程序流程图作为工具设计软件时，常常通过研究流程图确定模块的划分，这样得到的往往是过程内聚的模块。

如果模块中所有元素都使用同一个输入数据和(或)产生同一个输出数据，则称为通信内聚。

高内聚也有两类：

如果一个模块内的处理元素和同一个功能密切相关，而且这些处理必须顺序执行(通常一个处理元素的输出数据作为下一个处理元素的输入数据)，则称为顺序内聚。根据数据流图划分模块时，通常得到顺序内聚的模块，这种模块彼此间的连接往往比较简单。

如果模块内所有处理元素属于一个整体，完成一个单一的功能，则称为功能内聚。功能内聚是最高程度的内聚。

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

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

过程内聚 5 分

设计时力争做到高内聚，并且能够辨认出低内聚的模块，有能力通过修改设计提高模块的内聚程度降低模块间的耦合程度，从而获得较高的模块独立性。

5.3 启发规则

人们在开发计算机软件的长期实践中积累了丰富的经验，总结这些经验得出了一些启发式规则。

1. 改进软件结构提高模块独立性

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

2. 模块规模应该适中

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

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

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

3. 深度、宽度、扇出和扇入都应适当

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

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

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

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

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

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)。

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 5.2 中模块 A 的控制域是 A、B、C、D、E、F 等模块的集合。

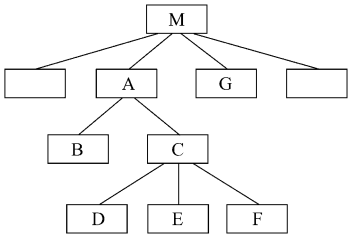


图 5.2 模块的作用域和控制域

5. 力争降低模块接口的复杂程度

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

接口复杂或不一致，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. 设计单入口单出口的模块

这条启发式规则警告软件工程师不要使模块间出现内容耦合。当从顶部进入模块并且从底部退出来时，软件是比较容易理解的，也是比较容易维护的。

7. 模块功能应该可以预测

模块的功能应该能够预测，但也要防止模块功能过分局限。

如果一个模块可以当做一个黑盒子，也就是说，只要输入的数据相同就产生同样的输出，这个模块的功能就是可以预测的。

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

5.4 描绘软件结构的图形工具

5.4.1 层次图和 HIPO 图

层次图用来描绘软件的层次结构。层次图中的一个矩形框代表一个模块，方框间的连线表示调用关系。图 5.3 是层次图的一个例子。

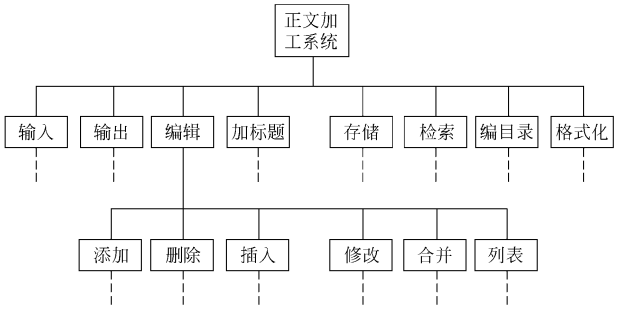


图 5.3 正文加工系统的层次图

层次图很适于在自顶向下设计软件的过程中使用。

HIPO 图是 IBM 公司发明的“层次图加输入/处理/输出图”的缩写。为了能使 HIPO 图具有可追踪性，在层次图里除了最顶层的方框之外，每个方框都加了编号。

图 5.3 加了编号后得到图 5.4。

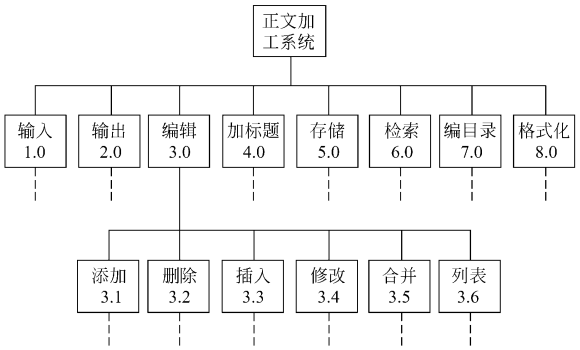


图 5.4 带编号的层次图(H 图)

5.4.2 结构图

Yourdon 提出的结构图是进行软件结构设计的另一个工具。结构图和层次图类似，也是描绘软件结构的图形工具，图中一个方框代表一个模块，框内注明模块的名字或主要功能；方框之间的箭头(或直线)表示模块的调用关系。因为按照惯例总是图中位于上方的方框代表的模块调用下方的模块，即使不用箭头也不会产生二义性，为了简单起见，可以只用直线而不用箭头表示模块间的调用关系。

在结构图中通常还用带注释的箭头表示模块调用过程中来回传递的信息。如果希望进一步标明传递的信息是数据还是控制信息，则可以利用注释箭头尾部的形状来区分：尾部是空心圆表示传递的是数据，实心圆表示传递的是控制信息。图 5.5 是结构图的一个例子。

以上介绍的是结构图的基本符号，也就是最经常使用的符号。此外还有一些附加的符号，可以表示模块的选择调用或循环调用。图 5.6 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 5.7 表示模块 M 循环调用模块 A、B 和 C。

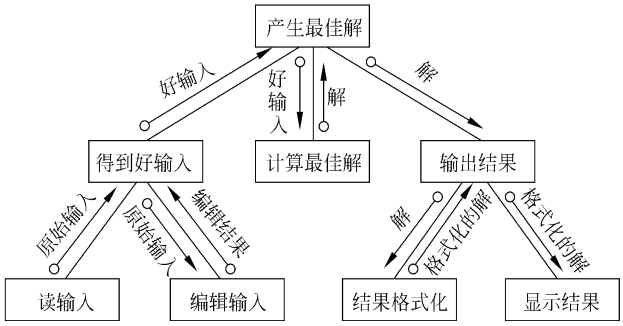


图 5.5 结构图的例子——产生最佳解的一般结构

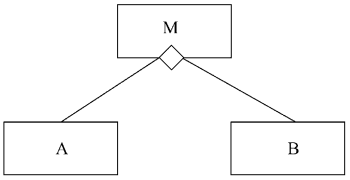


图 5.6 判定为真时调用 A，为假时调用 B

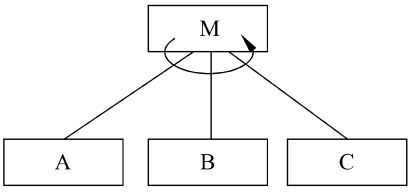


图 5.7 模块 M 循环调用模块 A、B、C

5.5 面向数据流的设计方法

面向数据流的设计方法的目标是给出设计软件结构的一个系统化的途径。

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

5.5.1 概念

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

1. 变换流

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

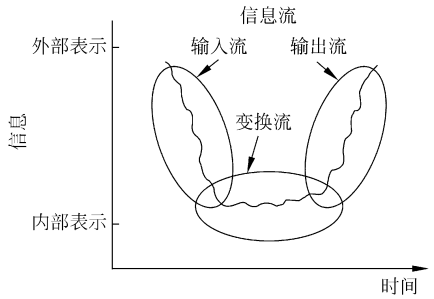


图 5.8 变换流

2. 事务流

数据沿输入通路到达一个处理 T，这个处理根据输入数据的类型在若干个动作序列中选出一个来执行。当数据流图具有类似的形状时，这种数据流是“以事务为中心的”，称为事务流。图 5.9 中的处理 T 称为事务中心，它完成下述任务：

(1) 接收输入数据(输入数据又称为事务)；

(2) 分析每个事务以确定它的类型；

(3) 根据事务类型选取一条活动通路。

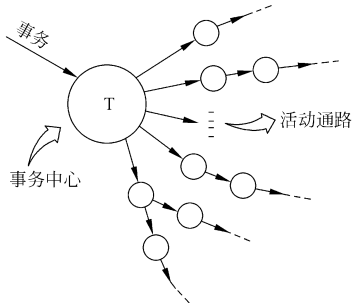


图 5.9 事务流

3. 设计过程

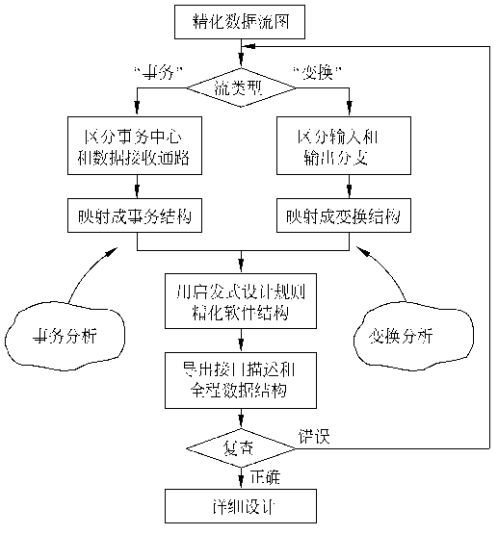


图 5.10 说明了使用面向数据流方法逐步设计的过程。

注意，任何设计过程都不是机械地一成不变的。

5.5.2 变换分析

变换分析是一系列设计步骤的总称，经过这些步骤把具有变换流特点的数据流图按预先确定的模式映射成软件结构。

1. 例子

考虑汽车数字仪表板的设计。

假设的仪表板将完成下述功能：

(1) 通过模数转换实现传感器和微处理机接口；

(2) 在发光二极管面板上显示数据；

(3) 指示每小时英里数(mph)，行驶的里程，每加仑油行驶的英里数(mpg)等等；

(4) 指示加速或减速；

(5) 超速警告：如果车速超过 55 英里/小时，则发出超速警告铃声。

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

2. 设计步骤

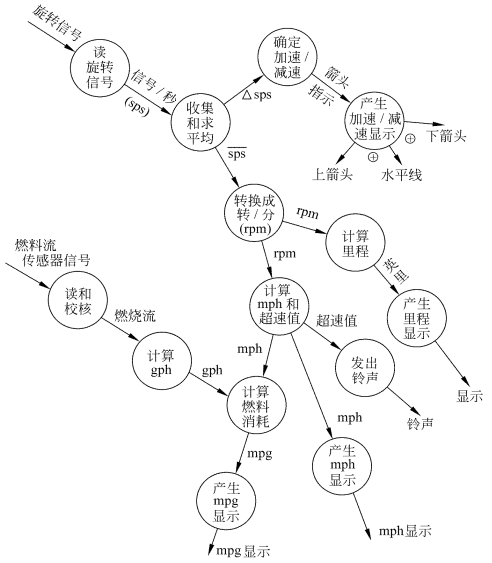
第 1 步 复查基本系统模型。

复查的目的是确保系统的输入数据和输出数据符合实际。

第 2 步 复查并精化数据流图。

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

假设在需求分析阶段产生的数字仪表板系统的数据流图如图 5.11 所示。



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

第 3 步 确定数据流图具有变换特性还是事务特性。

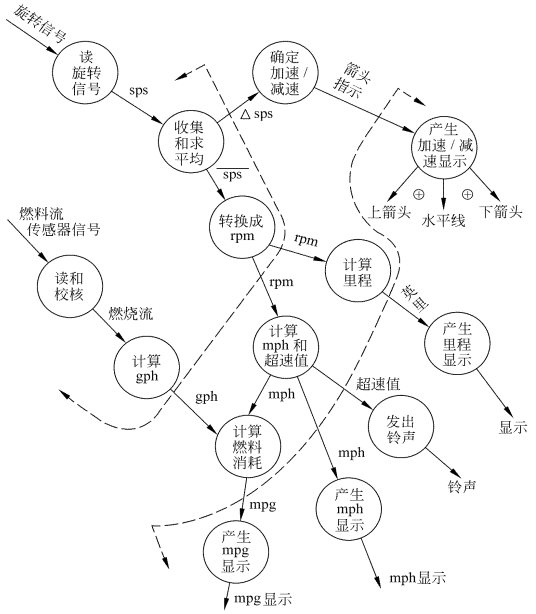
一般地说，一个系统中的所有信息流都可以认为是变换流，但是，当遇到有明显事务特性的信息流时，建议采用事务分析方法进行设计。在这一步，设计人员应该根据数据流图中占优势的属性，确定数据流的全局特性。此外还应该把具有和全局特性不同的特点的局部区域孤立出来，以后可以按照这些子数据流的特点精化根据全局特性得出的软件结构。

从图 5.11 看出，数据沿着两条输入通路进入系统，然后沿着 5 条通路离开，没有明显的事务中心。因此可以认为这个信息流具有变换流的总特征。

第 4 步 确定输入流和输出流的边界，从而孤立出变换中心。

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

对于汽车数字仪表板的例子，设计人员确定的流的边界如图 5.12 所示。



第 5 步 完成“第一级分解”。

软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。

对于变换流的情况，数据流图被映射成一个特殊的软件结构，这个结构控制输入、变换和输出等信息处理过程。图 5.13 说明了第一级分解的方法。位于软件结构最顶层的控制模块 Cm 协调下述从属的控制功能：

输入信息处理控制模块 Ca,协调对所有输入数据的接收；

变换中心控制模块 Ct,管理对内部形式的数据的所有操作；

输出信息处理控制模块 Ce，协调输出信息的产生过程。

虽然图 5.13 意味着一个三叉的控制结构，但是，对一个大型系统中的复杂数据流可以用两个或多个模块完成上述一个模块的控制功能。应该在能够完成控制功能并且保持好的耦合和内聚特性的前提下，尽量使第一级控制中的模块数目取最小值。

对于数字仪表板的例子，第一级分解得出的结构如图 5.14 所示。每个控制模块的名字表明了为它所控制的那些模块的功能。

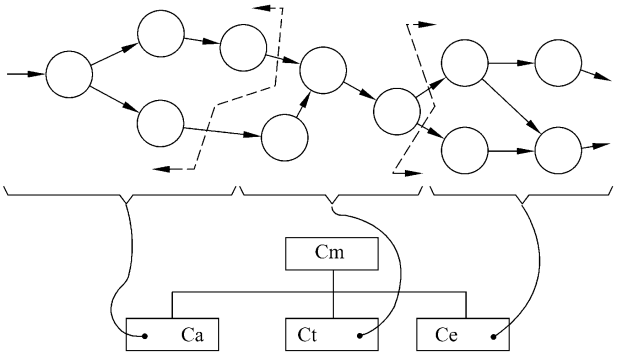


图 5.13 第一级分解的方法

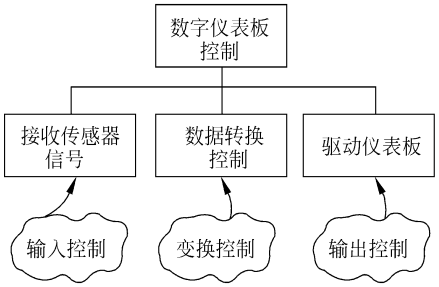


图 5.14 数字仪表板系统的第一级分解

第 6 步 完成“第二级分解”。

所谓第二级分解就是把数据流图中的每个处理映射成软件结构中一个适当的模块。完成第二级分解的方法是，从变换中心的边界开始沿着输入通路向外移动，把输入通路中每个处理映射成软件结构中 Ca 控制下的一个低层模块；然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受模块 Ce 控制的一个低层模块；最后把变换中心内的每个处理映射成受 Ct 控制的一个模块。图 5.15 表示进行第二级分解的普遍途径。

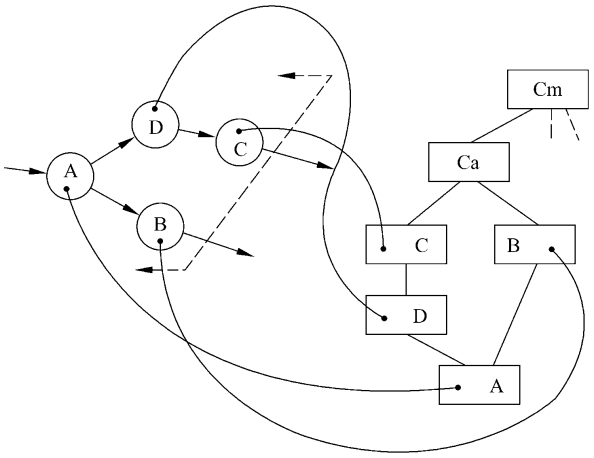


图 5.15 第二级分解的方法

虽然图 5.15 描绘了在数据流图中的处理和软件结构中的模块之间的一对一的映射关系，但是，不同的映射经常出现。应该根据实际情况以及“好”设计的标准，进行实际的第二级分解。

对于数字仪表板系统的例子，第二级分解的结果分别用图 5.16，5.17 和 5.18 描绘。这 3 张图表示对软件结构的初步设计结果。虽然图中每个模块的名字表明了它的基本功

能，但是仍然应该为每个模块写一个简要说明，描述：
进出该模块的信息(接口描述)；
模块内部的信息；
过程陈述，包括主要判定点及任务等；
对约束和特殊特点的简短讨论。
这些描述是第一代的设计规格说明，在这个设计时期进一步的精化和补充是经常发生的。
第 7 步 使用设计度量和启发式规则对第一次分割得到的软件结构进一步精化。

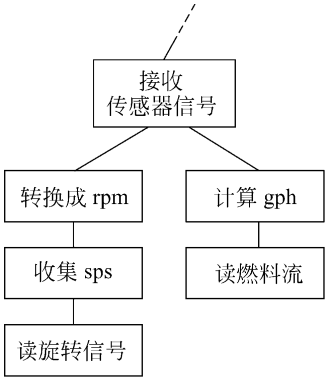


图 5.16 未经精化的输入结构

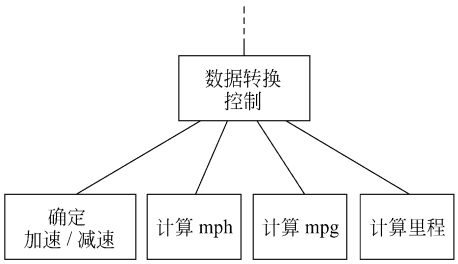


图 5.17 未经精化的变换结构

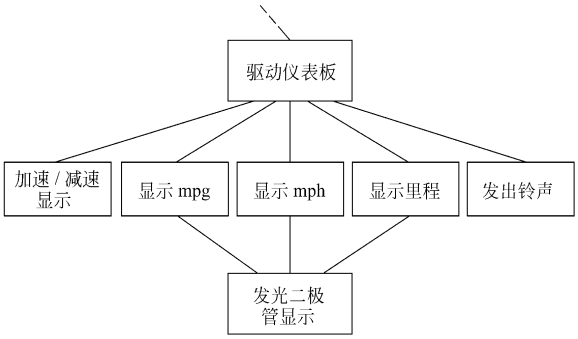


图 5.18 未经精化的输出结构

对第一次分割得到的软件结构，总可以根据模块独立原理进行精化。为了产生合理的分解，得到尽可能高的内聚、尽可能松散的耦合，最重要的是，为了得到一个易于实现、易于测试和易于维护的软件结构，应该对初步分割得到的模块进行再分解或合并。
具体到数字仪表板的例子，对于从前面的设计步骤得到的软件结构，还可以做许多修改。下面是某些可能的修改：
输入结构中的模块“转换成 rpm”和“收集 sps”可以合并；
模块“确定加速/减速”可以放在模块“计算 mph”下面，以减少耦合；
模块“加速/减速显示”可以相应地放在模块“显示 mph”的下面。
经过上述修改后的软件结构画在图 5.19 中。
上述 7 个设计步骤的目的是，开发出软件的整体表示。也就是说，一旦确定了软件结构就可以把它作为一个整体来复查，从而能够评价和精化软件结构。在这个时期进行修改只需要很少的附加工作，但是却能够对软件的质量特别是软件的可维护性产生深远的影响。

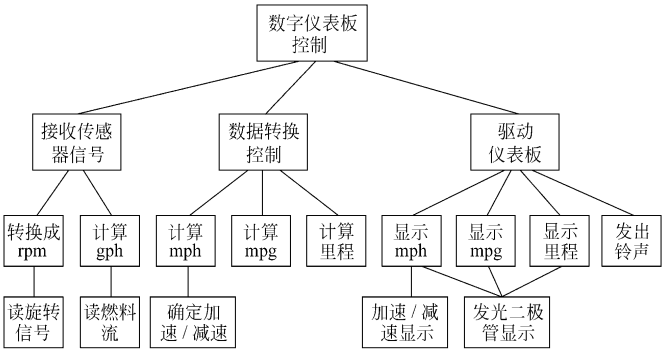


图 5.19 精化后的数字仪表板系统的软件结构

5.5.4 设计优化

软件设计人员应该致力于开发能够满足所有功能和性能要求，而且按照设计原理和启发式设计规则衡量是值得接收的软件。
应该在设计的早期阶段尽量对软件结构进行精化。可以导出不同的软件结构，然后对它们进行评价和比较，力求得到“最好”的结果。
结构简单通常既表示设计风格优雅，又表明效率高。设计优化应该力求做到在有效的模块化的前提下使用最少量的模块，以及在能够满足信息要求的前提下使用最简单的数据结构。
(1) 在不考虑时间因素的前提下开发并精化软件结构；
(2) 在详细设计阶段选出最耗费时间的那些模块，仔细地设计它们的处理过程(算法)，以求提高效率；
(3) 使用高级程序设计语言编写程序；
(4) 在软件中孤立出那些大量占用处理机资源的模块；
(5) 必要时重新设计或用依赖于机器的语言重写上述大量占用资源的模块的代码，以求提高效率。
上述优化方法遵守了一句格言：“先使它能工作，然后再使它快起来。”

5.6 小结

总体设计阶段的基本目的是用比较抽象概括的方式确定系统如何完成预定的任务，也就是说，应该确定系统的物理配置方案，并且进而确定组成系统的每个程序的结构。因此，总体设计阶段主要由两个小阶段组成。首先需要进行系统设计，然后进行软件结构设计，确定软件由哪些模块组成以及这些模块之间的动态调用关系。层次图和结构图是描绘软件结构的常用工具。
在进行软件结构设计时应该遵循的最主要的原理是模块独

立原理，也就是说，软件应该由一组完成相对独立的子功能的模块组成，这些模块彼此之间的接口关系应该尽量简单。

抽象和求精是一对互补的概念，在进行软件结构设计时一种有效的方法就是，由抽象到具体地构造出软件的层次结构。

软件工程师总结出一些很有参考价值的启发式规则，对如何改进软件设计给出宝贵的提示。在软件开发过程中既要充分重视和利用这些启发式规则，又要从实际情况出发避免生搬硬套。

自顶向下逐步求精是进行软件结构设计的常用途径；但是，如果已经有了详细的数据流图，也可以使用面向数据流的设计方法，用形式化的方法由数据流图映射出软件结构。这样映射出来的只是软件的初步结构，还必须根据设计原理并且参考启发式规则，认真分析和改进软件的初步结构，以得到质量更高的模块和更合理的软件结构。

在进行详细的过程设计和编写程序之前，首先进行结构设计，其好处正在于可以在软件开发的早期站在全局高度对软件结构进行优化。在这个时期进行优化付出的代价不高，却可以使软件质量得到重大改进。

第 6 章　详细设计

详细设计阶段的根本目标是确定应该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把这个描述直接翻译成用某种程序设计语言书写的程序。

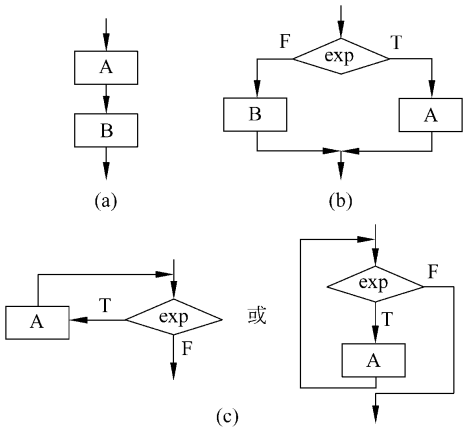
详细设计阶段的任务还不是具体地编写程序，而是要设计出程序的“蓝图”，以后程序员将根据这个蓝图写出实际的程序代码。因此，详细设计的结果决定了最终的程序代码的质量。

在软件的生命周期中，设计测试方案、诊断程序错误、修改和改进程序等等都必须首先读懂程序。实际上对于长期使用的软件系统而言，人读程序的时间可能比写程序的时间还要长得多。因此，衡量程序的质量不仅要看它的逻辑是否正确，性能是否满足要求，更主要的是要看它是否容易阅读和理解。详细设计的目标不仅仅是逻辑上正确地实现每个模块的功能，更重要的是设计出的处理过程应该尽可能简明易懂。

6.1　结构程序设计

结构程序设计的概念最早由 E.W.Dijkstra 提出。他在一次会议上提出取消 GO TO 语句。

1966 年 Bohm 和 Jacopini 证明了，只用 3 种基本的控制结构就能实现任何单入口单出口的程序。这 3 种基本的控制结构是“顺序”、“选择”和“循环”。



6.1 3 种基本的控制结构

结构程序设计本质上并不是无 GO TO 语句的编程方法，而是一种使程序代码容易阅读、容易理解的编程方法。在多数情况下，无 GO TO 语句的代码确实是容易阅读、容易理解的代码，但是，在某些情况下，为了达到容易阅读和容易理解的目的，反而需要使用 GO TO 语句。因此，下述的结构程序设计的定义可能更全面一些：

“结构程序设计是尽可能少用 GO TO 语句的程序设计方法。最好仅在检测出错误时才使用 GO TO 语句，而且应该总是使用前向 GO TO 语句。”

为了实际使用方便，常常还允许使用 DO-UNTIL 和 DO-CASE 两种控制结构。

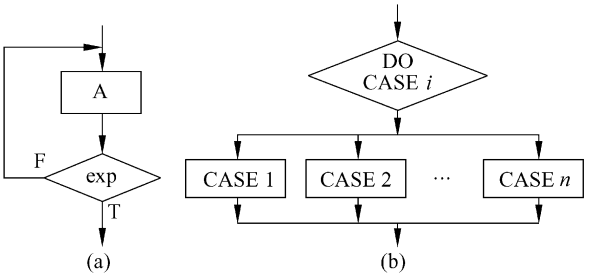


图 6.2 其他常用的控制结构

有时需要立即从循环(甚至嵌套的循环)中转移出来，如果允许使用 LEAVE(或 BREAK)结构，则不仅方便而且会使效率提高很多。LEAVE 或 BREAK 结构实质上是受限制的 GO TO 语句，用于转移到循环结构后面的语句。

6.2　人机界面设计

人机界面设计是接口设计的一个重要的组成部分。人机界面设计通常包括计算机通过输出设备向人提供信息及提示，以及人通过输入设备向计算机输入有关信息、问题问答等。近年来，人机界面在系统中所占的比例越来越大。人机界面的设计质量，直接影响用户对软件产品的评价，从而影响软件产品的竞争力和寿命，因此，必须对人机界面设计给予足够重视。

人机交互技术综述

人机交互的三元素

- 一个人机交互的计算机系统，通常必须考虑三个元素：交互设备，交互软件，以及人的因素。

- 交互设备:交互设备构成了交互计算机系统人机对话的基础；
- 交互设备可以分为许多类，主要有数字和字母输入输出设备（字符显示终端、打印机、键盘)；图形和图像输入输出设备（图形图像显示器，摄影机、扫描仪、绘图仪)；声音、触感及专用输入输出设备等。

人机交互技术综述

交互软件

交互软件是交互计算机系统的核心，向用户提供各种交互功能，以满足系统预定的要求；

交互软件分为系统软件和应用软件。系统软件包括操作系统、解释/编译程序、数据库查询语言、以及图形软件包。

人的因素

人的因素指的是用户操作模型，它与用户的各种特征有关；

首先，用户是人，人有许多弱点，如操作时经常出错。其次，用户的年龄、文化程度、工作经历对操作使用的要求也各不相同。

6.2.1 设计问题

在设计人机界面的过程中，几乎总会遇到下述 4 个问题：系统响应时间、用户帮助设施、出错信息处理和命令交互。不幸的是，许多设计者直到设计过程后期才开始考虑这些问题，这样做往往导致出现不必要的设计反复、项目延期和用户产生挫折感。最好在设计初期就把这些问题作为重要的设计问题来考虑，这时修改比较容易，代价也低。

1. 系统响应时间

系统响应时间指从用户完成某个控制动作(例如，按回车键或点击鼠标)，到软件给出预期的响应(输出信息或做动作)之间的这段时间。

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

易变性指系统响应时间相对于平均响应时间的偏差，在许多情况下，这是系统响应时间的更重要的属性。即使系统响应时间较长，响应时间易变性低也有助于用户建立起稳定的工作节奏。例如，稳定在 1 秒的响应时间比从 0.1 秒到 2.5 秒变化的响应时间要好。

2. 用户帮助设施

几乎交互式系统的每个用户都需要帮助，当遇到复杂问题时甚至需要查看用户手册以寻找答案。大多数现代软件都提供联机帮助设施，这使得用户无须离开用户界面就能解决自己的问题。

常见的帮助设施可分为集成的和附加的两类。

集成的帮助设施从一开始就设计在软件里面，通常对用户工作内容是敏感的，因此用户可以从与刚刚完成的操作有关的主题中选择一个请求帮助。可以缩短用户获得帮助的时间，增加界面的友好性。

附加的帮助设施是在系统建成后再添加到软件中的，在多数情况下它实际上是一种查询能力有限的联机用户手册。集成的帮助设施优于附加的帮助设施。

具体设计帮助设施时，必须解决下述的一系列问题。

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

(2) 用户怎样请求帮助?有 3 种选择：帮助菜单，特殊功能键和 HELP 命令。

(3) 怎样显示帮助信息?有 3 种选择：在独立的窗口中，指出参考某个文档和在屏幕固定位置显示简短提示。

(4) 用户怎样返回到正常的交互方式中?有两种选择：屏幕上的返回按钮和功能键。

(5) 怎样组织帮助信息?有 3 种选择：平面结构，信息的层次结构和超文本结构。

3. 出错信息处理

出错信息和警告信息，是出现问题时交互式系统给出的“坏消息”。出错信息设计得不好，将向用户提供无用的甚至误导的信息，反而会加重用户的挫折感。

一般说来，交互式系统给出的出错信息或警告信息，应该具有下述属性。

(1) 信息应该用用户可以理解的术语描述问题。

(2) 信息应该提供有助于从错误中恢复的建设性意见。

(3) 信息应该指出错误可能导致哪些负面后果(例如，破坏数据文件)，以便用户检查是否出现了这些问题，并在确实出现问题时及时解决。

(4) 信息应该伴随着听觉上或视觉上的提示，例如，在显示信息时同时发出警告铃声，或者信息用闪烁方式显示，或者信息用明显表示出错的颜色显示。

(5) 信息不能带有指责色彩，也就是说，不能责怪用户。

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

4. 命令交互

现在，面向窗口的、点击和拾取方式的界面已经减少了用户对命令行的依赖，但是，许多高级用户仍然偏爱面向命令行的交互方式。在多数情况下，用户既可以从菜单中选择软件功能，也可以通过键盘命令序列调用软件功能。

在提供命令交互方式时，必须考虑下列设计问题。

(1) 是否每个菜单选项都有对应的命令?

(2) 采用何种命令形式?有 3 种选择：控制序列(例如，Ctrl+P)，功能键和键入命令。

(3) 学习和记忆命令的难度有多大?忘记了命令怎么办?

(4) 用户是否可以定制或缩写命令?

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

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

6.2.2　设计过程

用户界面设计是一个迭代的过程，也就是说，通常先创建设计模型，再用原型实现这个设计模型，并由用户试用和评估，然后根据用户意见进行修改。

6.2.3　人机界面设计指南

用户界面设计主要依靠设计者的经验。总结众多设计者的经验得出的设计指南，有助于设计者设计出友好、高效的人机界面。下面介绍 3 类人机界面设计指南。

1. 一般交互指南

一般交互指南涉及信息显示、数据输入和系统整体控制，因此，这类指南是全局性的。

(1) 保持一致性。应该为人机界面中的菜单选择、命令输入、数据显示以及众多的其他功能，使用一致的格式。

(2) 提供有意义的反馈。应向用户提供视觉的和听觉的反馈，以保证在用户和系统之间建立双向通信。

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

(4) 允许取消绝大多数操作。UNDO 或 REVERSE 功能曾经使众多用户避免了大量时间浪费。每个交互式系统都应该能方便地取消已完成的操作。

(5) 减少在两次操作之间必须记忆的信息量。不应该期望用户能记住在下一步操作中需使用的一大串数字或标识符。应该尽量减少记忆量。

(6) 提高对话、移动和思考的效率。应该尽量减少用户击键的次数，设计屏幕布局时应该考虑尽量减少鼠标移动的距离。

(7) 允许犯错误。系统应该能保护自己不受严重错误的破坏。

(8) 按功能对动作分类，并据此设计屏幕布局。下拉菜单的一个主要优点就是能按动作类型组织命令。

(9) 提供对用户工作内容敏感的帮助设施。

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

2. 信息显示指南

如果人机界面显示的信息是不完整的、含糊的或难于理解的，则该应用系统显然不能满足用户的需求。下面是关于信息显示的设计指南。

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

(2) 不要用数据淹没用户，应该用便于用户迅速吸取信息的方式来表示数据。例如，可以用图形或图表来取代庞大的表格。

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

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

(5) 产生有意义的出错信息。

(6) 使用大小写、缩进和文本分组以帮助理解。人机界面显示的信息大部分是文字，文字的布局和形式对用户从中提取信息的难易程度有很大影响。

(7) 使用窗口分隔不同类型的信息。利用窗口用户能够方便地“保存”多种不同类型的信息。

(8) 使用“模拟”显示方式表示信息，以使信息更容易被用户提取。

(9) 高效率地使用显示屏。当使用多窗口时，应该有足够的空间使得每个窗口至少都能显示出一部分。此外，屏幕大小应该选得和应用系统的类型相配套。

3. 数据输入指南

用户的大部分时间用在选择命令、键入数据和向系统提供输入。下面是关于数据输入的设计指南。

(1) 尽量减少用户的输入动作。最重要的是减少击键次数，这可以用下列方法实现：用鼠标从预定义的一组输入中选一个；用“滑动标尺”在给定的值域中指定输入值；利用宏把一次击键转变成更复杂的输入数据集合。

(2) 保持信息显示和数据输入之间的一致性。显示的视觉特征应该与输入域一致。

(3) 允许用户自定义输入。

(4) 交互应该是灵活的，并且可调整成用户最喜欢的输入方式。用户类型与喜好的输入方式有关。

(5) 使在当前动作语境中不适用的命令不起作用。这可使得用户不去做那些肯定会导致错误的动作。

(6) 让用户控制交互流。用户应该能够跳过不必要的动作，改变所需做的动作的顺序(在应用环境允许的前提下)，以及在不退出程序的情况下从错误状态中恢复正常。

(7) 尽量对所有输入动作都提供帮助。

(8) 消除冗余的输入。尽可能提供默认值；不要要求用户

提供程序可以自动获得或计算出来的信息。

6.3 过程设计的工具

6.3.1 程序流程图

程序流程图又称为程序框图，它是历史最悠久、使用最广泛的描述过程设计的方法，然而它也是用得最混乱的一种方法。

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。至今仍在广泛使用着。

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。

6.3.2 盒图(N-S 图)

Nassi 和 Shneiderman 提出了盒图，又称为 N-S 图。它有下述特点：

(1) 功能域(即，一个特定控制结构的作用域)明确，可以从盒图上一眼就看出来。

(2) 不可能任意转移控制。

(3) 很容易确定局部和全程数据的作用域。

(4) 很容易表现嵌套关系，也可以表示模块的层次结构。

图 6.4 给出了结构化控制结构的盒图表示，也给出了调用子程序的盒图表示方法。

盒图没有箭头，因此不允许随意转移控制。坚持使用盒图作为详细设计的工具，可以使程序员逐步养成用结构化的方式思考问题和解决问题的习惯。

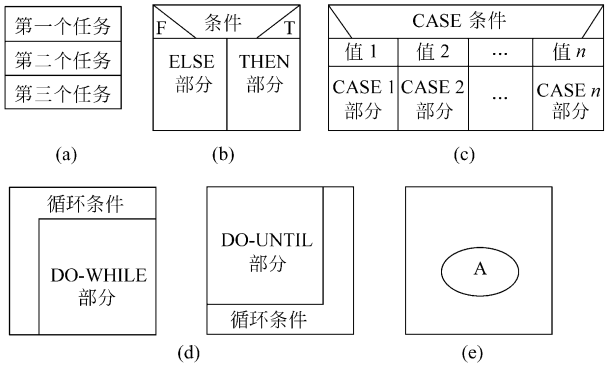


图 6.4 盒图的基本符号

6.3.3 PAD 图

PAD 是问题分析图(problem analysis diagram)的英文缩写，1973 年由日立公司发明。它用二维树形结构的图来表示程序的控制流，将这种图翻译成程序代码比较容易。图 6.5 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。

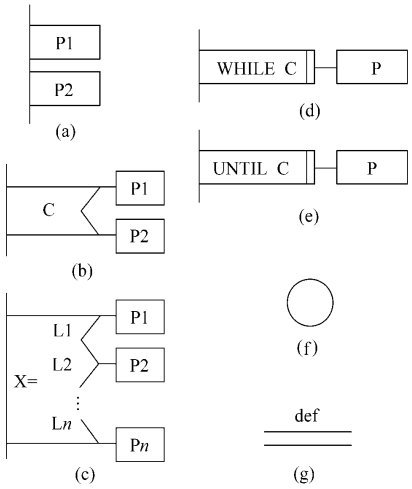


图 6.5 PAD 图的基本符号

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

(3) 用 PAD 图表现程序逻辑，易读、易懂、易记。

(4) 容易将 PAD 图转换成高级语言源程序。

(5) 既可用于表示程序逻辑，也可用于描绘数据结构。

(6) PAD 图的符号支持自顶向下、逐步求精方法的使用。

PAD 图是面向高级程序设计语言的，为 FORTRAN，COBOL 和 PASCAL 等每种常用的高级程序设计语言都提供了一整套相应的图形符号。

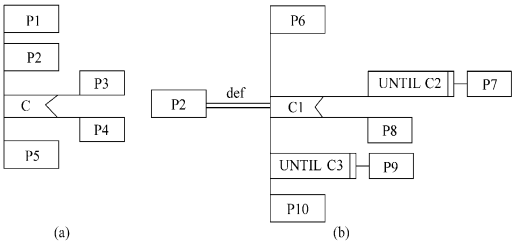


图 6.6 使用 PAD 图提供的定义功能来逐步求精的例子

6.3.4 判定表

当算法中包含多重嵌套的条件选择时，用判定表却能够清晰地表示复杂的条件组合与应做的动作之间的对应关系。一张判定表由 4 部分组成，左上部列出所有条件，左下部是所有可能做的动作，右上部是表示各种条件组合的一个矩阵，右下部是和每种条件组合相对应的动作。判定表右半部的每一列实质上是一条规则，规定了与特定的条件组合相对应的动作。

下面以行李托运费的算法为例说明判定表的组织方法。假设某航空公司规定，乘客可以免费托运重量不超过 30kg 的行李。当行李重量超过 30kg 时，对头等舱的国内乘客超重部分每公斤收费 4 元，对其他舱的国内乘客超重部分每公斤收费 6 元，对外国乘客超重部分每公斤收费比国内乘客多一倍，对残疾乘客超重部分每公斤收费比正常乘客少一半。用判定表可以清楚地表示与上述每种条件组合相

对应的计算行李费的算法，如表 6.1 所示。

	1	2	3	4	5	6	7	8	9
国内乘客		T	T	T	T	F	F	F	F
头等舱		T	F	T	F	T	F	T	F
残疾乘客		F	F	T	T	F	F	T	T
行李重量 W<=03kg	T	F	F	F	F	F	F	F	F
免费	X								
(w-30)X2				X					
(w-30)X3					X				
(w-30)X4		X						X	
(w-30)X6			X						X
(w-30)X8						X			
(w-30)X12							X		

从上面这个例子可以看出，判定表能够简洁而又无歧义地描述处理规则。当把判定表和布尔代数或卡诺图结合起来使用时，可以对判定表进行校验或化简。但是，判定表并不适于作为一种通用的设计工具，没有一种简单的方法使它能同时清晰地表示顺序和重复等处理特性。

6.3.5 判定树

判定表虽然能清晰地表示复杂的条件组合与应做的动作之间的对应关系，但其含义却不是一眼就能看出来的。此外，当数据元素的值多于两个时(例如，6.3.4 例子中假设对机票需细分为头等舱、二等舱和经济舱等多种级别时)，判定表的简洁程度也将下降。

判定树是判定表的变种，也能清晰地表示复杂的条件组合与应做的动作之间的对应关系。判定树的优点在于，它的形式简单到不需任何说明，一眼就可以看出其含义，因此易于掌握和使用。判定树是一种比较常用的系统分析和设计的工具。图 6.7 是和表 6.1 等价的判定树。

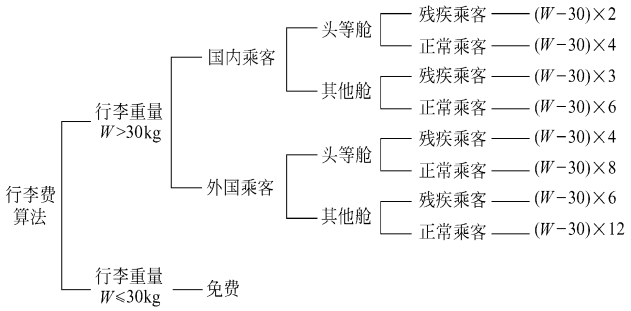


图 6.7 用判定树表示计算行李费的算法

6.3.6 过程设计语言

过程设计语言（PDL）也称为伪码，是用正文形式表示数据和处理过程的设计工具。

PDL 具有严格的关键字外部语法，用于定义控制结构和数据结构；另一方面，PDL 表示实际操作和条件的内部语法通常又是灵活自由的，可以适应各种工程项目的需要。因此，PDL 是一种“混杂”语言，它使用一种语言的词汇，同时却使用另一种语言(某种结构化的程序设计语言)的语法。

PDL 应该具有下述特点：

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

(2) 自然语言的自由语法，它描述处理特点。

(3) 数据说明的手段。应该既包括简单的数据结构(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，应该提供各种接口描述模式。

PDL 作为一种设计工具有如下一些优点：

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

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

(3) 已经有自动处理程序存在，而且可以自动由 PDL 生成程序代码。

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

6.4 面向数据结构的设计方法

数据结构既影响程序的结构又影响程序的处理过程，重复出现的数据通常由具有循环控制结构的程序来处理，可选择的数据要用带有分支控制结构的程序来处理。

面向数据结构的设计方法的最终目标是得出对程序处理过程的描述。这种方法最适合于在详细设计阶段使用，也就是说，在完成了软件结构设计之后，可以使用面向数据结构的方法来设计每个模块的处理过程。

Jackson 方法和 Warnier 方法是最著名的两个面向数据结构的设计方法。

使用面向数据结构的设计方法，当然首先需要分析确定数据结构，并且用适当的工具清晰地描绘数据结构。

6.4.1 Jackson 图

虽然程序中实际使用的数据结构种类繁多，但是它们的数据元素彼此间的逻辑关系却只有顺序、选择和重复 3 类，因此，逻辑数据结构也只有这 3 类。

1. 顺序结构

顺序结构的数据由一个或多个数据元素组成，每个元素按确定次序出现一次。图 6.8 是表示顺序结构的 Jackson 图的一个例子。

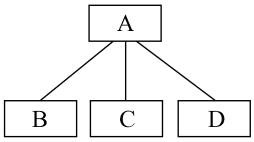


图 6.8 A 由 B、C、D 3 个元素顺序组成

2. 选择结构

选择结构的数据包含两个或多个数据元素，每次使用这个数据时按一定条件从这些数据元素中选择一个。图 6.9 是表示 3 个中选 1 个结构的 Jackson 图。

3. 重复结构

重复结构的数据，根据使用时的条件由一个数据元素出现零次或多次构成。图 6.10 是表示重复结构的 Jackson 图。

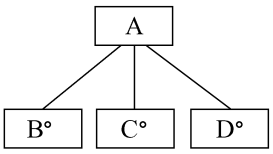


图 6.9 根据条件 A 是 B 或 C 或 D 中的某一个

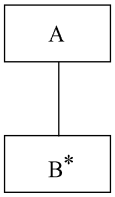


图 6.10 A 由 B 出现 N 次(N≥0)组成

Jackson 图有下述优点：

便于表示层次结构，而且是对结构进行自顶向下分解的有力工具；

形象直观可读性好；

既能表示数据结构也能表示程序结构(因为结构程序设计也只使用上述 3 种基本控制结构)。

6.4.2 改进的 Jackson 图

Jackson 图的缺点是，用这种图形工具表示选择或重复结构时，选择条件或循环结束条件不能直接在图上表示出来，影响了图的表达能力，也不易直接把图翻译成程序，此外，框间连线为斜线，不易在行式打印机上输出。为了解决上述问题，人们又提出了改进的 Jackson 图。

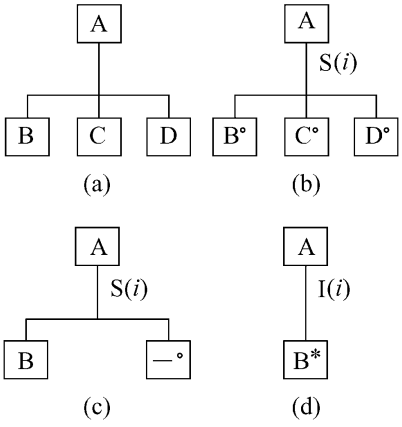


图 6.11 改进的 Jackson 图

Jackson 图实质上是对第 3.7 节中介绍的层次方框图的一种精化。虽然 Jackson 图和描绘软件结构的层次图形式相当类似，但是含义却很不相同，即，层次图中的一个方框通常代表一个模块；而 Jackson 图即使在描绘程序结构时，一个方框也并不代表一个模块，通常一个方框只代表几个语句。层次图表现的是调用关系，通常一个模块除了调用下级模块外，还完成其他操作；而 Jackson 图表现的是组成关系，也就是说，一个方框中包括的操作仅仅由它下层框中的那些操作组成。

6.4.3 Jackson 方法

Jackson 结构程序设计方法基本上由下述 5 个步骤组成：

(1) 分析并确定输入数据和输出数据的逻辑结构，并用 Jackson 图描绘这些数据结构。

(2) 找出输入数据结构和输出数据结构中有对应关系的数据单元。所谓有对应关系是指有直接的因果关系，在程序中可以同时处理的数据单元(对于重复出现的数据单元必须重复的次序和次数都相同才可能有对应关系)。

(3) 用下述 3 条规则从描绘数据结构的 Jackson 图导出描绘程序结构的 Jackson 图：

第一，为每对有对应关系的数据单元，按照它们在数据结构图中的层次在程序结构图的相应层次画一个处理框(注意，如果这对数据单元在输入数据结构和输出数据结构中所处的层次不同，则和它们对应的处理框在程序结构图中所处的层次与它们之中在数据结构图中层次低的那个对应)；

第二，根据输入数据结构中剩余的每个数据单元所处的层次，在程序结构图的相应层次分别为它们画上对应的处理框；

第三，根据输出数据结构中剩余的每个数据单元所处的层次，在程序结构图的相应层次分别为它们画上对应的处理框。

总之，描绘程序结构的 Jackson 图应该综合输入数据结构和输出数据结构的层次关系而导出来。在导出程序结构图的过程中，由于改进的 Jackson 图规定在构成顺序结构的元素中不能有重复出现或选择出现的元素，因此可能需要增加中间层次的处理框。

(4) 列出所有操作和条件(包括分支条件和循环结束条件)，并且把它们分配到程序结构图的适当位置。

(5) 用伪码表示程序。

Jackson 方法中使用的伪码和 Jackson 图是完全对应的，下面是和 3 种基本结构对应的伪码。

和图 6.11(a)所示的顺序结构对应的伪码，其中‘seq’和‘end’是关键字：

```
A    seq
     B
     C
     D
A    end
```

和图 6.11(b)所示的选择结构对应的伪码，其中‘select’、‘or’和‘end’是关键字，cond1、cond2 和 cond3 分别是执行 B、C 或 D 的条件：

```
A    select    cond1
     B
A    or        cond2
     C
A    or        cond3
     D
A    end
```

和图 6.11(d)所示重复结构对应的伪码，其中‘iter’、‘until’、‘while’和‘end’是关键字，cond 是条件：

```
A    iter until(或 while) cond
     B
A    end
```

下面结合一个具体例子进一步说明 Jackson 结构程序设计方法。

［例］一个正文文件由若干个记录组成，每个记录是一个字符串。要求统计每个记录中空格字符的个数，以及文件中空格字符的总个数。要求的输出数据格式是，每复制一行输入字符串之后，另起一行印出这个字符串中的空格数，最后印出文件中空格的总个数。

对于这个简单例子而言，输入和输出数据的结构很容易确定。图 6.12 是用 Jackson 图描绘的输入输出数据结构。

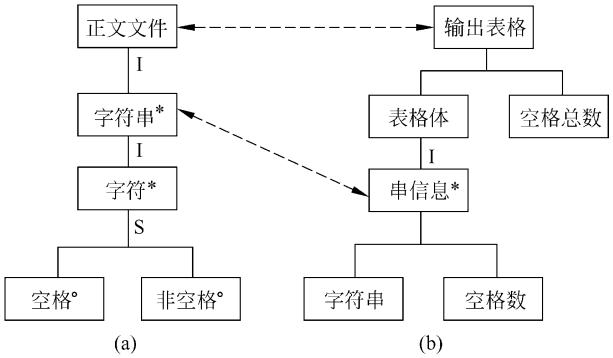


图 6.12 表示输入输出数据结构的 Jackson 图

确定了输入输出数据结构之后，第二步是分析确定在输入数据结构和输出数据结构中有对应关系的数据单元。在这个例子中哪些数据单元有对应关系呢?输出数据总是通过对输入数据的处理而得到的，因此在输入输出数据结构最高层次的两个单元总是有对应关系的。这一对单元将和程序结构图中最顶层的方框(代表程序)相对应，也就是说经过程序的处理由正文文件得到输出表格。因为每处理输入数据中一个“字符串”之后，就可以得到输出数据中一个“串信息”，它们都是重复出现的数据单元，而且出现次序和重复次数都完全相同，因此，“字符串”和“串信息”也是一对有对应关系的单元。

还有其他有对应关系的单元吗?为了回答这个问题依次考察输入数据结构中余下的每个数据单元。“字符”不可能和多个字符组成的“字符串”对应，和输出数据结构中其他数据单元也不能对应。“空格”能和“空格数”对应吗?显然，单个空格并不能决定一个记录中包含的空格个数，因此没有对应关系。通过类似的考察发现，输入数据结构中余下的任何一个单元在输出数据结构中都找不到对应的单元，也就是说，在这个例子中输入输出数据结构中只有上述两对有对应关系的单元。在图 6.12 中用一对虚线箭头把有对应关系的数据单元连接起来，以突出表明这种对应关系。

Jackson 程序设计方法的第三步是从数据结构图导出程序结构图。按照前面已经讲述过的规则，这个步骤的大致过程是：

首先，在描绘程序结构的 Jackson 图的最顶层画一个处理框“统计空格”，它与“正文文件”和“输出表格”这对最顶层的数据单元相对应。但是接下来还不能立即画与另一对数据单元(“字符串”和“串信息”)相对应的处理框，因为在输出数据结构中“串信息”的上层还有“表格体”和“空格总数”两个数据单元，在程序结构图的第二层应该有与这两个单元对应的处理框——“程序体”和“印总数”。

因此，在程序结构图的第三层才是与“字符串”和“串信息”相对应的处理框——“处理字符串”。在程序结构图的第四层似乎应该是和“字符串”、“字符”及“空格数”等数据单元对应的处理框“印字符串”、“分析字符”及“印空格数”，这 3 个处理是顺序执行的。但是，“字符”是重复出现的数据单元，因此“分析字符”也应该是重复执行的处理。改进的 Jackson 图规定顺序执行的处理中不允许混有重复执行或选择执行的处理，所以在“分析字符”这个处理框上面又增加了一个处理框“分析字符串”。最后得到的程序结构图为图 6.13。

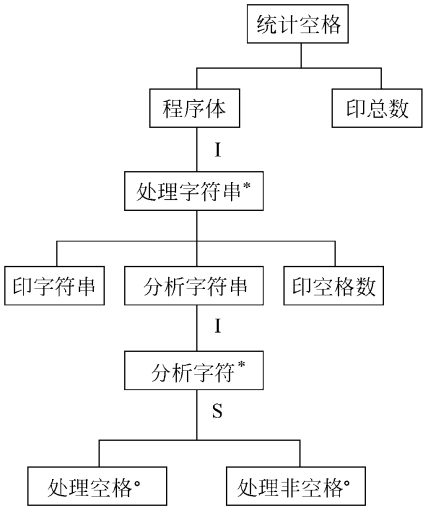


图 6.13 描绘统计空格程序结构的 Jackson 图

Jackson 程序设计方法的第四步是列出所有操作和条件，并且把它们分配到程序结构图的适当位置。首先，列出统计空格个数需要的全部操作和条件。

经过简单分析不难把这些操作和条件分配到程序结构图的适当位置，结果为图 6.14。

Jackson 方法的最后一步是用伪码表示程序处理过程。因为 Jackson 使用的伪码和 Jackson 图之间存在简单的对应关系，所以从图 6.14 很容易得出下面的伪码：

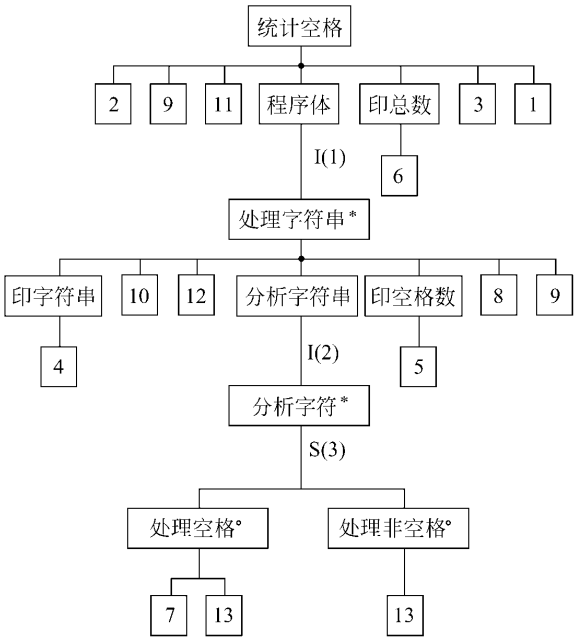


图 6.14 把操作和条件分配到程序结构图的适当位置

```
统计空格 seq
        打开文件
        读入字符串
        totalsum：=0
        程序体 iter until 文件结束
            处理字符串 seq
            印字符串 seq
                印出字符串
            印字符串 end
            sum：=0
            pointer：=1
            分析字符串 iter until 字符串结束
                分析字符 select 字符是空格
                处理空格 seq
                    sum：=sum+1
                    pointer：=pointer+1
                处理空格 end
                分析字符 or 字符不是空格
                处理非空格 seq
                    pointer：=pointer+1
                处理非空格 end
                分析字符 end
            分析字符串 end
            印空格数 seq
                印出空格数目
            印空格数 end
            totalsum：=totalsum+sum
            读入字符串
            处理字符串 end
        程序体 end
        印总数 seq
            印出空格总数
        印总数 end
        关闭文件
        停止
统计空格 end
```

以上简单介绍了由英国人 M.Jackson 提出的结构程序设计方法。这个方法在设计比较简单的数据处理系统时特别方便，当设计比较复杂的程序时常常遇到输入数据可能有错、条件不能预先测试、数据结构冲突等问题。为了克服上述困难，把 Jackson 方法应用到更广阔的领域，需要采用一系列比较复杂的辅助技术，详细介绍这些技术已经超出本书的范围。

6.5 程序复杂程度的定量度量

详细设计阶段设计出的模块质量如何呢?人们希望能定量度量软件的性质。本节将要介绍的程序复杂程度定量度量方法是其中比较成熟的一种。

定量度量程序复杂程度的方法很有价值：把程序的复杂程度乘以适当常数即可估算出软件中错误的数量以及软件开发需要用的工作量，定量度量的结果可以用来比较两个不同的设计或两个不同算法的优劣；程序的定量的复杂程度可以作为模块规模的精确限度。

下面介绍使用得比较广泛的 McCabe 方法和 Halstead 方法。

6.5.1 McCabe 方法

1. 流图

McCabe 方法根据程序控制流的复杂程度定量度量程序的复杂程度，这样度量出的结果称为程序的环形复杂度。为了突出表示程序的控制流，人们通常使用流图(也称为程序图)。所谓流图实质上是“退化了的”程序流程图，它仅仅描绘程序的控制流程，完全不表现对数据的具体操作以及分支或循环的具体条件。

在流图中用圆表示结点，一个圆代表一条或多条语句。程序流程图中的一个顺序的处理框序列和一个菱形判定框，可以映射成流图中的一个结点。流图中的箭头线称为边，它和程序流程图中的箭头线类似，代表控制流。在流图中一条边必须终止于一个结点，即使这个结点并不代表任何语句(实际上相当于一个空语句)。由边和结点围成的面积称为区域，当计算区域数时应该包括图外部未被围起来的那个区域。

图 6.15 举例说明把程序流程图映射成流图的方法。

用任何方法表示的过程设计结果，都可以翻译成流图。图 6.16 是用 PDL 表示的处理过程及与之对应的流图。

当过程设计中包含复合条件时，生成流图的方法稍微复杂一些。所谓复合条件，就是在条件中包含了一个或多个布尔运算符(逻辑 OR，AND，NAND，NOR)。在这种情况下，应该把复合条件分解为若干个简单条件，每个简单条

件对应流图中一个结点。包含条件的结点称为判定节点，从每个判定结点引出两条或多条边。图 6.17 是由包含复合条件的 PDL 片断翻译成的流图。

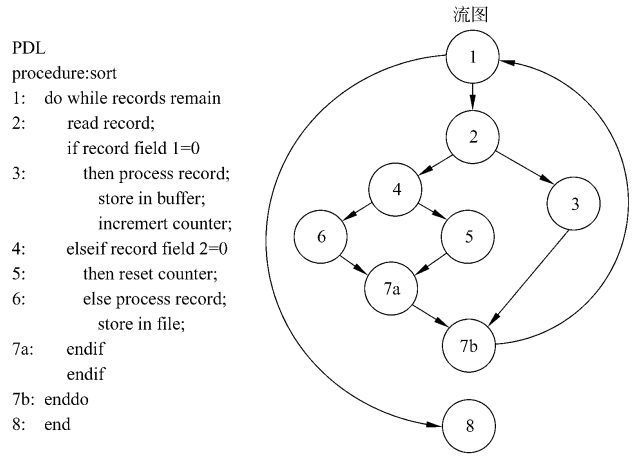


图 6.16 由 PDL 翻译成的流图

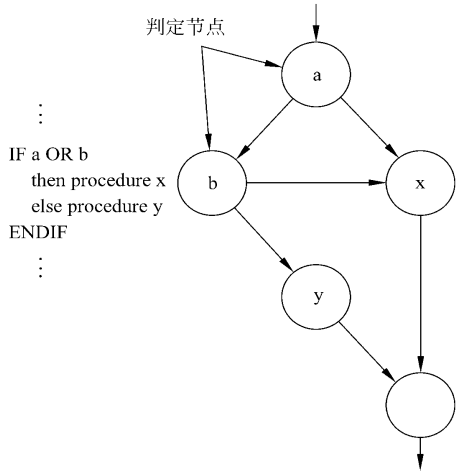


图 6.17 由包含复合条件的 PDL 映射成的流图

2. 计算环形复杂度的方法

环形复杂度定量度量程序的逻辑复杂度。有了描绘程序控制流的流图之后，可以用下述 3 种方法中的任何一种来计算环形复杂度。

(1) 流图中的区域数等于环形复杂度。

(2) 流图 G 的环形复杂度 V(G)=E-N+2,其中，E 是流图中边的条数，N 是结点数。

(3) 流图 G 的环形复杂度 V(G)=P+1，其中，P 是流图中判定结点的数目。

3. 环形复杂度的用途

程序的环形复杂度取决于程序控制流的复杂程度，也即是取决于程序结构的复杂程度。当程序内分支数或循环个数增加时，环形复杂度也随之增加，因此它是对测试难度的一种定量度量，也能对软件最终的可靠性给出某种预测。McCabe 研究大量程序后发现，环形复杂度高的程序往往是最困难、最容易出问题的程序。实践表明，模块规模以 V(G)≤10 为宜，也就是说，V(G)=10 是模块规模的一个更科学更精确的上限。

6.5.2 Halstead 方法

Halstead 方法是另一个著名的方法，它根据程序中运算符和操作数的总数来度量程序的复杂程度。

令 N1 为程序中运算符出现的总次数，N2 为操作数出现的总次数，程序长度 N 定义为：N=N1+N2

详细设计完成之后，可以知道程序中使用的不同运算符(包括关键字)的个数 n1，以及不同操作数(变量和常数)的个数 n2。Halstead 给出预测程序长度的公式如下：

H=n1 log2 n1+n2 log2 n2

多次验证都表明，预测的长度 H 与实际长度 N 非常接近。

Halstead 还给出了预测程序中包含错误的个数的公式如下：

$$E=N \log_2 (n1+n2)/3000$$

有人曾对从 300 条到 12000 条语句范围内的程序核实了上述公式，发现预测的错误数与实际错误数相比误差在 8%之内。

6.6 小结

详细设计阶段的关键任务是确定怎样具体地实现用户需要的软件系统，也就是要设计出程序的“蓝图”。除了应该保证软件的可靠性之外，使将来编写出的程序可读性好、容易理解、容易测试、容易修改和维护，是详细设计阶段最重要的目标。结构程序设计技术是实现上述目标的基本保证，是进行详细设计的逻辑基础。

人机界面设计是接口设计的一个重要的组成部分。人机界面的质量直接影响用户对软件产品的接受程度，因此，对人机界面设计必须给予足够重视。在设计人机界面的过程中，必须充分重视并认真处理好系统响应时间、用户帮助设施、出错信息处理和命令交互等 4 个设计问题。总结人们在设计人机界面过程中积累的经验，得出了一些关于用户界面设计的指南，认真遵守这些指南有助于设计出友好、高效的人机界面。

过程设计应该在数据设计、体系结构设计和接口设计完成之后进行，它的任务是设计解题的详细步骤(即算法)，它是详细设计阶段应完成的主要工作。过程设计的工具可分为图形、表格和语言 3 类。

在许多应用领域中信息都有清楚的层次结构，在开发这类应用系统时可以采用面向数据结构的设计方法完成过程设计。本章以 Jackson 结构程序设计技术为例，对面向数据结构的设计方法做了初步介绍。使用环形复杂度可以定量度量程序的复杂程度，实践表明，环形复杂度 V(G)=10 是模块规模的合理上限。

第 7 章 实现

通常把编码和测试统称为实现。

所谓编码就是把软件设计结果翻译成用某种程序设计语言书写的程序。程序的质量主要取决于软件设计的质量。但是，所选用的程序设计语言的特点及编码风格也将对程序的可靠性、可读性、可测试性和可维护性产生深远的影

响。
无论怎样强调软件测试的重要性和它对软件可靠性的影响都不过分。
人的主观认识不可能完全符合客观现实，与工程密切相关的各类人员之间的通信和配合也不可能完美无缺，因此，在软件生命周期的每个阶段都不可避免地会产生差错。力求在每个阶段结束之前通过严格的技术审查，尽可能早地发现并纠正差错；但是，经验表明审查并不能发现所有差错，此外在编码过程中还不可避免地会引入新的错误。
测试的目的就是在软件投入生产性运行之前，尽可能多地发现软件中的错误。
但是，发现错误并不是最终目的。通过测试发现错误之后还必须诊断并改正错误。
目前软件测试仍然是保证软件质量的关键步骤，它是对软件规格说明、设计和编码的最后复审。
通常在编写出每个模块之后就对它做必要的测试(称为单元测试)，模块的编写者和测试者是同一个人。在这个阶段结束之后，对软件系统还应该进行各种综合测试，这是软件生命周期中的另一个独立的阶段，通常由专门的测试人员承担这项工作。
大量统计资料表明，软件测试的工作量往往占软件开发总工作量的 40%以上。因此，必须高度重视软件测试工作。

7.1 编码

7.1.1 选择程序设计语言

程序设计语言是人和计算机通信的最基本的工具，它的特点必然会影响人的思维和解题方式，会影响人和计算机通信的方式和质量，也会影响其他人阅读和理解程序的难易程度。因此，编码之前的一项重要工作就是选择一种适当的程序设计语言。
适宜的程序设计语言能使根据设计去完成编码时困难最少，可以减少需要的程序测试量，并且可以得出更容易阅读和更容易维护的程序。由于软件系统的绝大部分成本用在生命周期的测试和维护阶段，所以容易测试和容易维护是极端重要的。
使用汇编语言编码需要把软件设计翻译成机器操作的序列，由于这两种表示方法很不相同，因此汇编程序设计既困难又容易出差错。
高级语言使用的符号和概念更符合人的习惯。因此，用高级语言写的程序容易阅读，容易测试，容易调试，容易维护。
总的说来，高级语言明显优于汇编语言，因此，除了在很特殊的应用领域(例如，对程序执行时间和使用的空间都有很严格限制的情况；需要产生任意的甚至非法的指令序列；体系结构特殊的微处理机，以致在这类机器上通常不能实现高级语言编译程序)，或者大型系统中执行时间非常关键的(或直接依赖于硬件的)一小部分代码需要用汇编语言书写之外，其他程序应该一律用高级语言书写。
为了使程序容易测试和维护以减少软件的总成本，所选用的高级语言应该有理想的模块化机制，以及可读性好的控制结构和数据结构；为了便于调试和提高软件可靠性，语言特点应该使编译程序能够尽可能多地发现程序中的错误；为了降低软件开发和维护的成本，选用的高级语言应该有良好的独立编译机制。
在实际选择语言时不能仅仅使用理论上的标准，还必须同时考虑实用方面的各种限制。下面是主要的实用标准：
(1) 系统用户的要求。如果所开发的系统由用户负责维护，用户通常要求用他们熟悉的语言书写程序。
(2) 可以使用的编译程序。运行目标系统的环境中可以提供的编译程序往往限制了可以选用的语言的范围。
(3) 可以得到的软件工具。如果某种语言有支持程序开发的软件工具可以利用，则目标系统的实现和验证都变得比较容易。
(4) 工程规模。如果工程规模很庞大，现有的语言又不完全适用，那么设计并实现一种供这个工程项目专用的程序设计语言，可能是一个正确的选择。
(5) 程序员的知识。如果和其他标准不矛盾，那么应该选择一种已经为程序员所熟悉的语言。
(6) 软件可移植性要求。如果目标系统将在几台不同的计算机上运行，或者预期的使用寿命很长，那么选择一种标准化程度高、程序可移植性好的语言就是很重要的。
(7) 软件的应用领域。所谓的通用程序设计语言实际上并不是对所有应用领域都同样适用。因此，选择语言时应该充分考虑目标系统的应用范围。

7.1.2 编码风格

源程序代码的逻辑简明清晰、易读易懂是好程序的一个重要标准，为了做到这一点，应该遵循下述规则。
1. 程序内部的文档
所谓程序内部的文档包括恰当的标识符、适当的注解和程序的视觉组织等等。
选取含义鲜明的名字，使它能正确地提示程序对象所代表的实体，这对于帮助阅读者理解程序是很重要的。如果使用缩写，那么缩写规则应该一致，并且应该给每个名字加注解。
注解是程序员和程序读者通信的重要手段，正确的注解非常有助于对程序的理解。
通常在每个模块开始处有一段序言性的注解，简要描述模块的功能、主要算法、接口特点、重要数据以及开发简史。
插在程序中间与一段程序代码有关的注解，主要解释包含这段代码的必要性。
用空格或空行清楚地区分注解和程序。
注解的内容一定要正确。
程序清单的布局对于程序的可读性也有很大影响，应该利

用适当的形式使程序的层次结构清晰明显。

2. 数据说明

数据说明的次序应该标准化。有次序就容易查阅，因此能够加速测试、调试和维护的过程。

当多个变量名在一个语句中说明时，按字母顺序排列这些变量。

如果设计时使用了一个复杂的数据结构，则应该用注解说明用程序设计语言实现这个数据结构的方法和特点。

3. 语句构造

构造语句应该遵循的原则是，每个语句都应该简单而直接，不能为了提高效率而使程序变得过分复杂。下述规则有助于使语句简单明了：

不要为了节省空间而把多个语句写在同一行；

尽量避免复杂的条件测试；

尽量减少对“非”条件的测试；

避免大量使用循环嵌套和条件嵌套；

利用括号使逻辑表达式或算术表达式的运算次序清晰直观。

4. 输入输出

在设计和编写程序时应考虑下述有关输入输出风格的规则：

对所有输入数据都进行检验；

检查输入项重要组合的合法性；

保持输入格式简单；

明确提示交互式输入的请求，详细说明可用的选择或边界数值；

当程序设计语言对格式有严格要求时，应保持输入格式一致；

设计良好的输出报表；

给所有输出数据加标志。

5. 效率

效率主要指处理机时间和存储器容量两个方面。程序的效率和程序的简单程度是一致的，不要牺牲程序的清晰性和可读性来不必要地提高效率。下面从三个方面进一步讨论效率问题。

(1) 程序运行时间

源程序的效率直接由详细设计阶段确定的算法的效率决定，但是，写程序的风格也能对程序的执行速度和存储器要求产生影响。在把详细设计结果翻译成程序时，总可以应用下述规则：

写程序之前先简化算术的和逻辑的表达式；

研究嵌套的循环，以确定是否有语句可以从内层往外移；

尽量避免使用多维数组；

尽量避免使用指针和复杂的表；

使用执行时间短的算术运算；

不要混合使用不同的数据类型；

尽量使用整数运算和布尔表达式。

(2) 存储器效率

在大型计算机中必须考虑操作系统页式调度的特点。

如果要求使用最少的存储单元，则应选用有紧缩存储器特性的编译程序，必要时可以使用汇编语言。

提高执行效率的技术通常也能提高存储器效率。

(3) 输入输出的效率

简单清晰是提高人机通信效率的关键。

所有输入输出都应该有缓冲，以减少用于通信的额外开销；

对二级存储器(如磁盘)应选用最简单的访问方法；

二级存储器的输入输出应该以信息组为单位进行；

如果“超高效的”输入输出很难被人理解，则不应采用这种方法。

这些简单原则对于软件工程的设计和编码两个阶段都适用。

7.2 软件测试基础

在测试阶段测试人员努力设计出一系列测试方案，目的却是为了“破坏”已经建造好的软件系统——竭力证明程序中有错误不能按照预定要求正确工作。

当然，发现问题是为了解决问题，测试阶段的根本目标是尽可能多地发现并排除软件中潜藏的错误，最终把一个高质量的软件系统交给用户使用。

7.2.1 软件测试的目标

(1) 测试是为了发现程序中的错误而执行程序的过程；

(2) 好的测试方案是极可能发现迄今为止尚未发现的错误的测试方案；

(3) 成功的测试是发现了至今为止尚未发现的错误的测试。

正确认识测试的目标是十分重要的，测试目标决定了测试方案的设计。如果为了表明程序是正确的而进行测试，就会设计一些不易暴露错误的测试方案；相反，如果测试是为了发现程序中的错误，就会力求设计出最能暴露错误的测试方案。

由于测试的目标是暴露程序中的错误，由程序的编写者自己进行测试是不恰当的。因此，在综合测试阶段通常由其他人员组成测试小组来完成测试工作。

此外，即使经过了最严格的测试之后，仍然可能还有没被发现的错误潜藏在程序中。测试只能查找出程序中的错误，不能证明程序中没有错误。

7.2.2 软件测试准则

为了能设计出有效的测试方案，软件工程师必须深入理解并正确运用指导软件测试的基本准则。下面讲述主要的测试准则。

（1） 所有测试都应该能追溯到用户需求。正如上一小节讲过的，软件测试的目标是发现错误。从用户的角度看，最严重的错误是导致程序不能满足用户需求的那些错误。

（2） 应该远在测试开始之前就制定出测试计划。实际

上，一旦完成了需求模型就可以着手制定测试计划，在建立了设计模型之后就可以立即开始设计详细的测试方案。因此，在编码之前就可以对所有测试工作进行计划和设计。

（3） 把二八原则应用到软件测试中。

（4） 应该从“小规模”测试开始，并逐步进行“大规模”测试。通常，首先重点测试单个程序模块，然后把测试重点转向在集成的模块簇中寻找错误，最后在整个系统中寻找错误。

（5） 穷举测试是不可能的。所谓穷举测试就是把程序所有可能的执行路径都检查一遍的测试。

但是，精心地设计测试方案，有可能充分覆盖程序逻辑并使程序达到所要求的可靠性。

（6） 为了达到最佳的测试效果，应该由独立的第三方从事测试工作。

7.2.3 测试方法

测试有两种方法：

黑盒测试:如果已经知道了产品应该具有的功能，可以通过测试来检验是否每个功能都能正常使用；

白盒测试:如果知道产品的内部工作过程，可以通过测试来检验产品内部动作是否按照规格说明书的规定正常进行。

对于软件测试而言，黑盒测试法把程序看作一个黑盒子，完全不考虑程序的内部结构和处理过程。也就是说，黑盒测试是在程序接口进行的测试，它只检查程序功能是否能按照规格说明书的规定正常使用，程序是否能适当地接收输入数据并产生正确的输出信息，程序运行过程中能否保持外部信息的完整性。黑盒测试又称为功能测试。

白盒测试法与黑盒测试法相反，它的前提是可以把程序看成装在一个透明的白盒子里，测试者完全知道程序的结构和处理算法。这种方法按照程序内部的逻辑测试程序，检测程序中的主要执行通路是否都能按预定要求正确工作。白盒测试又称为结构测试。

7.2.4 测试步骤

测试过程分步骤进行，后一个步骤在逻辑上是前一个步骤的继续。大型软件系统通常由若干个子系统组成，每个子系统又由许多模块组成，因此，大型软件系统的测试过程基本上由下述几个步骤组成。

1. 模块测试

模块测试通常又称为单元测试。

在设计得好的软件系统中，每个模块完成一个清晰定义的子功能。因此，把每个模块作为一个单独的实体来测试，容易设计检验模块正确性的测试方案。模块测试的目的是保证每个模块作为一个单元能正确运行，在这个测试步骤中所发现的往往是编码和详细设计的错误。

2. 子系统测试

子系统测试是把经过单元测试的模块放在一起形成一个子系统来测试。模块相互间的协调和通信是这个测试过程中的主要问题，因此，这个步骤着重测试模块的接口。

3. 系统测试

系统测试是把经过测试的子系统装配成一个完整的系统来测试。在这个过程中不仅应该发现设计和编码的错误，还验证系统确实能提供需求说明书中指定的功能。在这个测试步骤中发现的往往是软件设计中的错误，也可能发现需求说明中的错误。

不论是子系统测试还是系统测试，都兼有检测和组装两重含义，通常称为集成测试。

4. 验收测试

验收测试也称为确认测试。

验收测试把软件系统作为单一的实体进行测试，测试内容与系统测试基本类似，但是它是在用户参与下进行的，而且可能主要使用实际数据进行测试。

验收测试的目的是验证系统确实能够满足用户的需要，在这个测试步骤中发现的往往是系统需求说明书中的错误。

5. 平行运行

关系重大的软件产品在验收之后往往并不立即投入生产性运行，而是要再经过一段运行时间的考验。所谓平行运行就是同时运行新开发出来的系统和旧系统，以便比较新旧两个系统的处理结果。这样做的具体目的有如下几点：

(1) 可以在准生产环境中运行新系统而又不冒风险；

(2) 用户能有一段熟悉新系统的时间；

(3) 可以验证用户指南和使用手册之类的文档；

(4) 能够以准生产模式对新系统进行全负荷测试，可以用测试结果验证性能指标。

7.2.5 测试阶段的信息流

图 7.1 描绘了测试阶段的信息流，这个阶段的输入信息有两类： (1)软件配置，包括需求说明书、设计说明书和源程序清单等； (2)测试配置，包括测试计划和测试方案。所谓测试方案不仅仅是测试时使用的输入数据(称为测试用例)，还应该包括每组输入数据预定要检验的功能，以及每组输入数据预期应该得到的正确输出。实际上测试配置是软件配置的一个子集，最终交出的软件配置应该包括上述测试配置以及测试的实际结果和调试的记录。

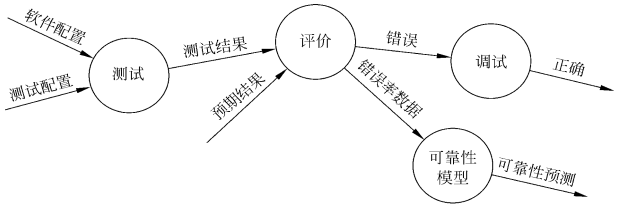


图 7.1 测试阶段的信息流

比较测试得出的实际结果和预期的结果，如果两者不一致则很可能是程序中有错误。设法确定错误的准确位置并且改正它，这就是调试的任务。与测试不同，通常由程序的编写者负责调试。

7.3 单元测试

单元测试集中检测软件设计的最小单元——模块。在编写出源程序代码并通过语法检查之后，就可以用详细设计作指南，对重要的执行通路进行测试，以便发现模块内部的错误。单元测试主要使用白盒测试技术，而且对多个模块的测试可以并行地进行。

7.3.1 测试重点

在单元测试期间着重从下述 5 个方面对模块进行测试。

1. 模块接口

首先应该对通过模块接口的数据流进行测试。

在对模块接口进行测试时主要检查下述几个方面：参数的数目、次序、属性或单位系统与变元是否一致；是否修改了只作输入用的变元；全局变量的定义和用法在各个模块中是否一致。

2. 局部数据结构

对于模块来说，局部数据结构是常见的错误来源。应该仔细设计测试方案，以便发现局部数据说明、初始化、默认值等方面的错误。

3. 重要的执行通路

在单元测试期间选择最有代表性、最可能发现错误的执行通路进行测试是十分关键的。应该设计测试方案用来发现由于错误的计算、不正确的比较或不适当的控制流而造成的错误。

4. 出错处理通路

好的设计应该能预见出现错误的条件，并且设置适当的处理错误的通路，以便在真的出现错误时执行相应的出错处理通路或干净地结束处理。当评价出错处理通路时，应该着重测试下述一些可能发生的错误：

(1) 对错误的描述是难以理解的；

(2) 记下的错误与实际遇到的错误不同；

(3) 在对错误进行处理之前，错误条件已经引起系统干预；

(4) 对错误的处理不正确；

(5) 描述错误的信息不足以帮助确定造成错误的位置。

5. 边界条件

边界测试是单元测试中最后的也可能是最重要的任务。软件常常在它的边界上失效，例如，处理 n 元数组的第 n 个元素时，或做到 i 次循环中的第 i 次重复时，往往会发生错误。使用刚好小于、刚好等于和刚好大于最大值或最小值的数据结构、控制量和数据值的测试方案，非常可能发现软件中的错误。

7.3.2 代码审查

人工测试源程序可以由编写者本人非正式地进行，也可以由审查小组正式进行。后者称为代码审查，它是一种非常有效的程序验证技术，对于典型的程序来说，可以查出 30%～70%的逻辑设计错误和编码错误。审查小组最好由下述 4 人组成：

(1) 组长，应该是一个很有能力的程序员，而且没有直接参与这项工程；

(2) 程序的设计者；

(3) 程序的编写者；

(4) 程序的测试者。

如果一个人既是程序的设计者又是编写者，或既是编写者又是测试者，则审查小组中应该再增加一个程序员。

审查之前，小组成员应该先研究设计说明书，力求理解这个设计。可以先由设计者扼要地介绍他的设计，其他成员倾听他的讲解，并力图发现其中的错误。审查会上进行的另外一项工作，是对照类似于上一小节中介绍的程序设计常见错误清单，分析审查这个程序。当发现错误时由组长记录下来，审查会继续进行(审查小组的任务是发现错误而不是改正错误)。

审查会还有另外一种常见的进行方法，称为预排：由一个人扮演“测试者”，其他人扮演“计算机”。会前测试者准备好测试方案，会上由扮演计算机的成员模拟计算机执行被测试的程序。在大多数情况下，通过向程序员提出关于他的程序的逻辑和他编写程序时所做的假设的疑问，可以发现的错误比由测试方案直接发现的错误还多。

代码审查比计算机测试优越的是：一次审查会上可以发现许多错误；用计算机测试的方法发现错误之后，通常需要先改正这个错误才能继续测试，因此错误是一个一个地发现并改正的。也就是说，采用代码审查的方法可以减少系统验证的总工作量。

实践表明，对于查找某些类型的错误来说，人工测试比计算机测试更有效；对于其他类型的错误来说则刚好相反。因此，人工测试和计算机测试是互相补充，相辅相成的，缺少其中任何一种方法都会使查找错误的效率降低。

7.3.3 计算机测试

模块并不是一个独立的程序，因此必须为每个单元测试开发驱动软件和(或)存根软件。通常驱动程序也就是一个“主程序”，它接收测试数据，把这些数据传送给被测试的模块，并且印出有关的结果。存根程序代替被测试的模块所调用的模块。因此存根程序也可以称为“虚拟子程序”。它使用被它代替的模块的接口，可能做最少量的数据操作，印出对入口的检验或操作结果，并且把控制归还给调用它的模块。

例如，图 7.2 是一个正文加工系统的部分层次图，假定要测试其中编号为 3.0 的关键模块——正文编辑模块。因为正文编辑模块不是一个独立的程序，所以需要有一个测试驱动程序来调用它。这个驱动程序说明必要的变量，接收测试数据——字符串，并且设置正文编辑模块的编辑功能。因为在原来的软件结构中，正文编辑模块通过调用它的下层模块来完成具体的编辑功能，所以需要有存根程序简化地模拟这些下层模块。为了简单起见，测试时可以设置的编辑功能只有修改(CHANGE)和添加(APPEND)两

种，用控制变量 CFUNCT 标记要求的编辑功能，而且只用一个存根程序模拟正文编辑模块的所有下层模块。下面是用伪码书写的存根程序和驱动程序。

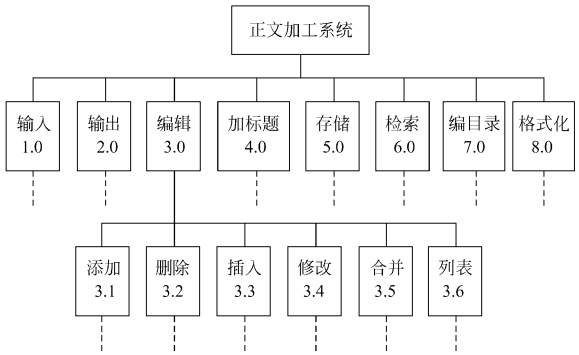


图 7.2 正文加工系统的层次图

```
Ⅰ. TEST STUB(*测试正文编辑模块用的存根程序*)
        初始化;
        输出信息“进入了正文编辑程序”;
        输出“输入的控制信息是”CFUNCT;
        输出缓冲区中的字符串;
        IF    CFUNCT=CHANGE
            THEN
            把缓冲区中第二个字改为***
            ELSE
            在缓冲区的尾部加???
        END IF;
        输出缓冲区中的新字符串;
END TEST STUB
Ⅱ. TEST  DRIVER(*测试正文编辑模块用的驱动程序*)
        说明长度为 2500 个字符的一个缓冲区;
        把 CFUNCT 置为希望测试的状态;
        输入字符串;
        调用正文编辑模块;
        停止或再次初启;
END   TEST   DRIVER
```

驱动程序和存根程序代表开销，也就是说，为了进行单元测试必须编写测试软件，但是通常并不把它们作为软件产品的一部分交给用户。许多模块不能用简单的测试软件充分测试，为了减少开销可以使用下节将要介绍的渐增式测试方法，在集成测试的过程中同时完成对模块的详尽测试。

模块的内聚程度高可以简化单元测试过程。如果每个模块只完成一种功能，则需要的测试方案数目将明显减少，模块中的错误也更容易预测和发现。

7.4 集成测试

集成测试是测试和组装软件的系统化技术，例如，子系统测试即是在把模块按照设计要求组装起来的同时进行测试，主要目标是发现与接口有关的问题。例如，数据穿过接口时可能丢失；一个模块对另一个模块可能由于疏忽而造成有害影响；把子功能组合起来可能不产生预期的主功能；个别看来是可以接受的误差可能积累到不能接受的程度；全程数据结构可能有问题等等。

由模块组装成程序时有两种方法。

一种方法是先分别测试每个模块，再把所有模块按设计要求放在一起结合成所要的程序，这种方法称为非渐增式测试方法；

另一种方法是把下一个要测试的模块同已经测试好的那些模块结合起来进行测试，测试完以后再把下一个应该测试的模块结合进来测试。这种每次增加一个模块的方法称为渐增式测试，这种方法实际上同时完成单元测试和集成测试。

非渐增式测试一下子把所有模块放在一起，并把庞大的程序作为一个整体来测试，测试者面对的情况十分复杂。测试时会遇到许许多多的错误，改正错误更是极端困难，因为在庞大的程序中想要诊断定位一个错误是非常困难的。而且一旦改正一个错误之后，马上又会遇到新的错误，这个过程将继续下去，看起来好像永远也没有尽头。

渐增式测试与“一步到位”的非渐增式测试相反，它把程序划分成小段来构造和测试，在这个过程中比较容易定位和改正错误；对接口可以进行更彻底的测试；可以使用系统化的测试方法。因此，目前在进行集成测试时普遍采用渐增式测试方法。

当使用渐增方式把模块结合到程序中去时，有自顶向下和自底向上两种集成策略。

7.4.1 自顶向下集成

自顶向下集成方法是一个日益为人们广泛采用的测试和组装软件的途径。从主控制模块开始，沿着程序的控制层次向下移动，逐渐把各个模块结合起来。在把附属于主控制模块的那些模块组装到程序结构中去时，或者使用深度优先的策略，或者使用宽度优先的策略。

参看图 7.3，深度优先的结合方法先组装在软件结构的一条主控制通路上的所有模块。选择一条主控制通路取决于应用的特点，并且有很大任意性。而宽度优先的结合方法是沿软件结构水平地移动，把处于同一个控制层次上的所有模块组装起来。

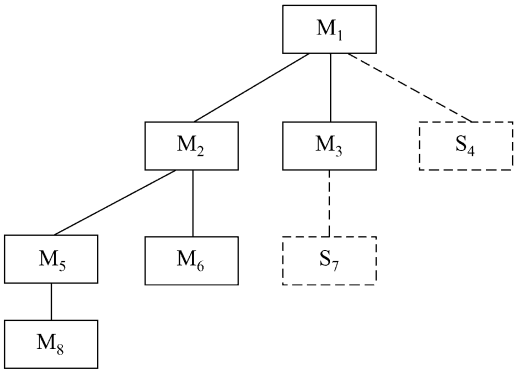


图 7.3 自顶向下结合

把模块结合进软件结构的具体过程由下述 4 个步骤完成：

第一步，对主控制模块进行测试，测试时用存根程序代替所有直接附属于主控制模块的模块；
第二步，根据选定的结合策略(深度优先或宽度优先)，每次用一个实际模块代换一个存根程序(新结合进来的模块往往又需要新的存根程序)；
第三步，在结合进一个模块的同时进行测试；
第四步，为了保证加入模块没有引进新的错误，可能需要进行回归测试(即，全部或部分地重复以前做过的测试)。
从第二步开始不断地重复进行上述过程，直到构造起完整的软件结构为止。图 7.3 描绘了这个过程。
自顶向下的结合策略能够在测试的早期对主要的控制或关键的抉择进行检验。在一个分解得好的软件结构中，关键的抉择位于层次系统的较上层，因此首先碰到。如果选择深度优先的结合方法，可以在早期实现软件的一个完整的功能并且验证这个功能。
自顶向下的方法讲起来比较简单，但是实际使用时可能遇到逻辑上的问题。这类问题中最常见的是，为了充分地测试软件系统的较高层次，需要在较低层次上的处理。然而在自顶向下测试的初期，存根程序代替了低层次的模块，因此，在软件结构中没有重要的数据自下往上流。为了解决这个问题，测试人员有两种选择： 第一，把许多测试推迟到用真实模块代替了存根程序以后再进行；第二，从层次系统的底部向上组装软件。第一种方法失去了在特定的测试和组装特定的模块之间的精确对应关系，这可能导致在确定错误的位置和原因时发生困难。后一种方法称为自底向上的测试，下面讨论这种方法。

7.4.2 自底向上集成

自底向上测试从“原子”模块(即在软件结构最低层的模块)开始组装和测试。因为是从底部向上结合模块，总能得到所需的下层模块处理功能，所以不需要存根程序。
用下述步骤可以实现自底向上的结合策略：
第一步，把低层模块组合成实现某个特定的软件子功能的族；
第二步，写一个驱动程序(用于测试的控制程序)，协调测试数据的输入和输出；
第三步，对由模块组成的子功能族进行测试；
第四步，去掉驱动程序，沿软件结构自下向上移动，把子功能族组合起来形成更大的子功能族。
上述第二步到第四步实质上构成了一个循环。图 7.4 描绘了自底向上的结合过程。
随着结合向上移动，对测试驱动程序的需要也减少了。事实上，如果软件结构的顶部两层用自顶向下的方法组装，可以明显减少驱动程序的数目，而且族的结合也将大大简化。

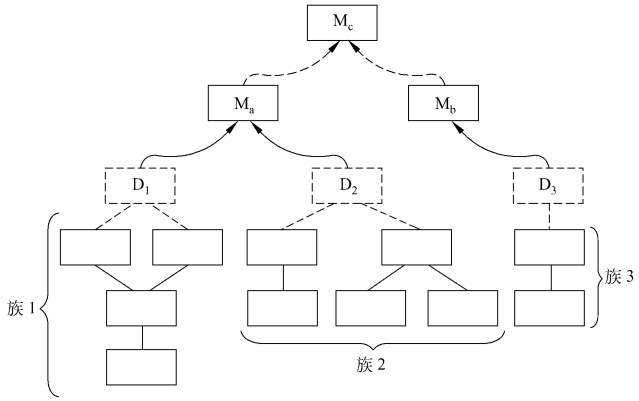


图 7.4 自底向上结合

7.4.3 不同集成测试策略的比较

一般说来，一种方法的优点正好对应于另一种方法的缺点。自顶向下测试方法的主要优点是不需要测试驱动程序，能够在测试阶段的早期实现并验证系统的主要功能，而且能在早期发现上层模块的接口错误。自顶向下测试方法的主要缺点是需要存根程序，可能遇到与此相联系的测试困难，低层关键模块中的错误发现较晚，而且用这种方法在早期不能充分展开人力。可以看出，自底向上测试方法的优缺点与上述自顶向下测试方法的优缺点刚好相反。
在测试实际的软件系统时，应该根据软件的特点以及工程进度安排，选用适当的测试策略。一般说来，纯粹自顶向下或纯粹自底向上的策略可能都不实用，人们在实践中创造出许多混合策略：
(1) 改进的自顶向下测试方法。基本上使用自顶向下的测试方法，但是在早期使用自底向上的方法测试软件中的少数关键模块。一般的自顶向下方法所具有的优点在这种方法中也都有，而且能在测试的早期发现关键模块中的错误；但是，它的缺点也比自顶向下方法多一条，即测试关键模块时需要驱动程序。
（2） 混合法。对软件结构中较上层使用的自顶向下方法与对软件结构中较下层使用的自底向上方法相结合。这种方法兼有两种方法的优点和缺点，当被测试的软件中关键模块比较多时，这种混合法可能是最好的折衷方法。

7.4.4 回归测试

在集成测试过程中每当一个新模块结合进来时，程序就发生了变化：建立了新的数据流路径，可能出现了新的 I/O 操作，激活了新的控制逻辑。这些变化有可能使原来工作正常的功能出现问题。在集成测试的范畴中，所谓回归测试是指重新执行已经做过的测试的某个子集，以保证上述这些变化没有带来非预期的副作用。
回归测试就是用于保证由于调试或其他原因引起的变化，不会导致非预期的软件行为或额外错误的测试活动。
回归测试可以通过重新执行全部测试用例的一个子集人工地进行，也可以使用自动化的捕获回放工具自动进行。利用捕获回放工具，软件工程师能够捕获测试用例和实际运行结果，然后可以回放（即重新执行测试用例），并且比

较软件变化前后所得到的运行结果。

回归测试集（已执行过的测试用例的子集）包括下述 3 类不同的测试用例：

（1） 检测软件全部功能的代表性测试用例；

（2） 专门针对可能受修改影响的软件功能的附加测试；

（3） 针对被修改过的软件成分的测试。

在集成测试过程中，回归测试用例的数量可能变得非常大。因此，应该把回归测试集设计成只包括可以检测程序每个主要功能中的一类或多类错误的那样一些测试用例。一旦修改了软件之后就重新执行检测程序每个功能的全部测试用例，是低效而且不切实际的。

7.5 确认测试

确认测试也称为验收测试，它的目标是验证软件的有效性。

什么样的软件才是有效的呢？软件有效性的一个简单定义是： 如果软件的功能和性能如同用户所合理期待的那样，软件就是有效的。

需求分析阶段产生的软件需求规格说明书，准确地描述了用户对软件的合理期望，因此是软件有效性的标准，也是进行确认测试的基础。

7.5.1 确认测试的范围

确认测试必须有用户积极参与，或者以用户为主进行。用户应该参与设计测试方案，使用用户界面输入测试数据并且分析评价测试的输出结果。为了使得用户能够积极主动地参与确认测试，特别是为了使用户能有效地使用这个系统，通常在验收之前由开发单位对用户进行培训。

确认测试通常使用黑盒测试法。应该仔细设计测试计划和测试过程，测试计划包括要进行的测试的种类及进度安排，测试过程规定了用来检测软件是否与需求一致的测试方案。通过测试和调试要保证软件能满足所有功能要求，能达到每个性能要求，文档资料是准确而完整的，此外，还应该保证软件能满足其他预定的要求（例如，安全性、可移植性、兼容性和可维护性等）。

确认测试有下述两种可能的结果：

（1） 功能和性能与用户要求一致，软件是可以接受的；

（2） 功能和性能与用户要求有差距。

在这个阶段发现的问题往往和需求分析阶段的差错有关，涉及的面通常比较广，因此解决起来也比较困难。为了制定解决确认测试过程中发现的软件缺陷或错误的策略，通常需要和用户充分协商。

7.5.2 软件配置复查

确认测试的一个重要内容是复查软件配置。复查的目的是保证软件配置的所有成分都齐全，质量符合要求，文档与程序完全一致，具有完成软件维护所必须的细节，而且已经编好目录。

除了按合同规定的内容和要求，由人工审查软件配置之外，在确认测试过程中还应该严格遵循用户指南及其他操作程序，以便检验这些使用手册的完整性和正确性。必须仔细记录发现的遗漏或错误，并且适当地补充和改正。

7.5.3 Alpha 和 Beta 测试

如果软件是专为某个客户开发的，可以进行一系列验收测试，以便用户确认所有需求都得到满足了。

如果一个软件是为许多客户开发的，那么，让每个客户都进行正式的验收测试是不现实的。在这种情况下，绝大多数软件开发商都使用被称为 Alpha 测试和 Beta 测试的过程，来发现那些看起来只有最终用户才能发现的错误。

Alpha 测试由用户在开发者的场所进行，并且在开发者对用户的“指导”下进行测试。开发者负责记录发现的错误和使用中遇到的问题。总之，Alpha 测试是在受控的环境中进行的。

Beta 测试由软件的最终用户们在一个或多个客户场所进行。与 Alpha 测试不同，开发者通常不在 Beta 测试的现场，因此，Beta 测试是软件在开发者不能控制的环境中的“真实”应用。用户记录在 Beta 测试过程中遇到的一切问题，并且定期把这些问题报告给开发者。接收到在 Beta 测试期间报告的问题之后，开发者对软件产品进行必要的修改。

7.6 白盒测试技术

设计测试方案是测试阶段的关键技术问题。所谓测试方案包括具体的测试目的（例如，预定要测试的具体功能），应该输入的测试数据和预期的结果。通常又把测试数据和预期的输出结果称为测试用例。其中最困难的问题是设计测试用的输入数据。

不同的测试数据发现程序错误的能力差别很大，为了提高测试效率降低测试成本，应该选用高效的测试数据。因为不可能进行穷尽的测试，选用少量“最有效的”测试数据，做到尽可能完备的测试就更重要了。

7.6.1 逻辑覆盖

有选择地执行程序中某些最有代表性的通路是对穷尽测试的惟一可行的替代办法。所谓逻辑覆盖是对一系列测试过程的总称，这组测试过程逐渐进行越来越完整的通路测试。测试数据执行(或叫覆盖)程序逻辑的程度可以划分成哪些不同的等级呢?从覆盖源程序语句的详尽程度分析，大致有以下一些不同的覆盖标准。

1. 语句覆盖

为了暴露程序中的错误，至少每个语句应该执行一次。语句覆盖的含义是，选择足够多的测试数据，使被测程序中每个语句至少执行一次。例如，图 7.5 所示的程序流程图描绘了一个被测模块的处理算法。

为了使每个语句都执行一次，程序的执行路径应该是 sacbed,为此只需要输入下面的测试数据(实际上 X 可以是任意实数)：A=2，B=0，X=4

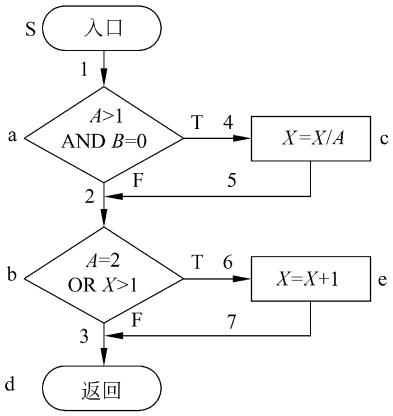


图 7.5 被测试模块的流程图

语句覆盖对程序的逻辑覆盖很少。此外，语句覆盖只关心判定表达式的值，而没有分别测试判定表达式中每个条件取不同值时的情况。

语句覆盖是很弱的逻辑覆盖标准，为了更充分地测试程序，可以采用下述的逻辑覆盖标准。

2. 判定覆盖

判定覆盖又叫分支覆盖，它的含义是，不仅每个语句必须至少执行一次，而且每个判定的每种可能的结果都应该至少执行一次，也就是每个判定的每个分支都至少执行一次。

判定覆盖比语句覆盖强，但是对程序逻辑的覆盖程度仍然不高，例如，上面的测试数据只覆盖了程序全部路径的一半。

3. 条件覆盖

条件覆盖的含义是，不仅每个语句至少执行一次，而且使判定表达式中的每个条件都取到各种可能的结果。

条件覆盖通常比判定覆盖强，因为它使判定表达式中每个条件都取到了两个不同的结果，判定覆盖却只关心整个判定表达式的值。

4. 判定/条件覆盖

既然判定覆盖不一定包含条件覆盖，条件覆盖也不一定包含判定覆盖，自然会提出一种能同时满足这两种覆盖标准的逻辑覆盖，这就是判定/条件覆盖。它的含义是，选取足够多的测试数据，使得判定表达式中的每个条件都取到各种可能的值，而且每个判定表达式也都取到各种可能的结果。

有时判定/条件覆盖也并不比条件覆盖更强。

5. 条件组合覆盖

条件组合覆盖是更强的逻辑覆盖标准，它要求选取足够多的测试数据，使得每个判定表达式中条件的各种可能组合都至少出现一次。

显然，满足条件组合覆盖标准的测试数据，也一定满足判定覆盖、条件覆盖和判定/条件覆盖标准。因此，条件组合覆盖是前述几种覆盖标准中最强的。但是，满足条件组合覆盖标准的测试数据并不一定能使程序中的每条路径都执行到。

以上根据测试数据对源程序语句检测的详尽程度，简单讨论了几种逻辑覆盖标准。在上面的分析过程中常常谈到测试数据执行的程序路径，显然，测试数据可以检测的程序路径的多少，也反映了对程序测试的详尽程度。从对程序路径的覆盖程度分析，能够提出下述一些主要的逻辑覆盖标准。

6. 点覆盖

图论中点覆盖的概念定义如下：如果连通图 G 的子图 G′是连通的，而且包含 G 的所有结点，则称 G′是 G 的点覆盖。

在第 6.5 节中已经讲述了从程序流程图导出流图的方法。在正常情况下流图是连通的有向图。满足点覆盖标准要求选取足够多的测试数据，使得程序执行路径至少经过流图的每个结点一次，由于流图的每个结点与一条或多条语句相对应，显然，点覆盖标准和语句覆盖标准是相同的。

7. 边覆盖

图论中边覆盖的定义是：如果连通图 G 的子图 G″是连通的，而且包含 G 的所有边，则称 G″是 G 的边覆盖。为了满足边覆盖的测试标准，要求选取足够多测试数据，使得程序执行路径至少经过流图中每条边一次。通常边覆盖和判定覆盖是一致的。

8. 路径覆盖

路径覆盖的含义是，选取足够多测试数据，使程序的每条可能路径都至少执行一次(如果程序图中有环，则要求每个环至少经过一次)。

7.6.2 控制结构测试

1. 基本路径测试

基本路径测试是一种白盒测试技术。使用这种技术设计测试用例时，首先计算程序的环形复杂度，并用该复杂度为指南定义执行路径的基本集合，从该基本集合导出的测试用例可以保证程序中的每条语句至少执行一次，而且每个条件在执行时都将分别取真、假两种值。

使用基本路径测试技术设计测试用例的步骤如下：

第一步，根据过程设计结果画出相应的流图。

例如，为了用基本路径测试技术测试下列的用 PDL 描述的求平均值过程，首先画出图 7.6 所示的流图。注意，为了正确地画出流图，我们把被映射为流图结点的 PDL 语句编了序号。

```
PROCEDURE average;
/* 这个过程计算不超过 100 个在规定值域内的有效数字的平均值；同时计算有效数字的总和及个数。*/
        INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;
    TYPE value［1…100］ IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
```

```
                    minimum,maximum, sum IS
SCALAR;
      TYPE i IS INTEGER;
1:    i=1;
          total.input=total.valid=0;
          sum=0;
2:    DO WHILE value［i］ <> -999
3:          AND total.input<100
4:    increment total.input by1;
5:    IF value［i］>=minimum
6:          AND value［i］<=maximum
7:    THEN increment total.valid by 1;
                  sum=sum+value［i］;
8:       ENDIF
               increment i by 1;
9:    ENDDO
10:   IF total.valid>0
11:   THEN average=sum/total.valid;
12:   ELSE average=-999;
13:   ENDIF
            END average
```

第二步，计算流图的环形复杂度。

环形复杂度定量度量程序的逻辑复杂性。有了描绘程序控制流的流图之后，可以用第 6.5.1 小节讲述的 3 种方法之一计算环形复杂度。经计算，图 7.6 所示流图的环形复杂度为 6。

第三步，确定线性独立路径的基本集合。

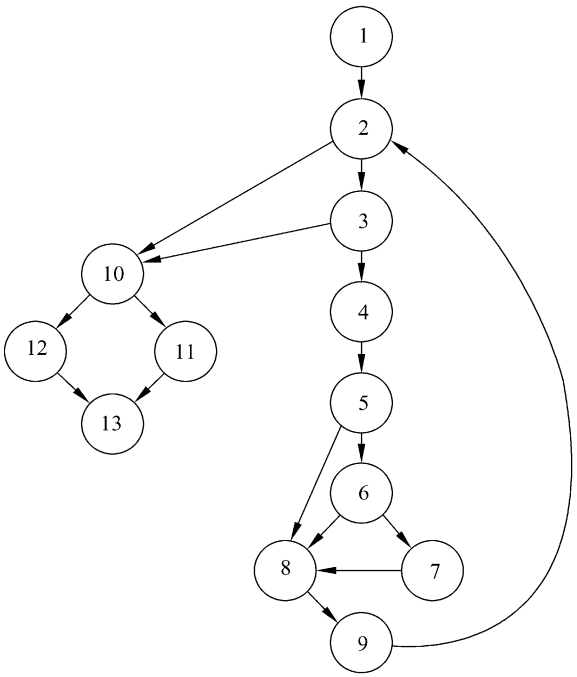


图 7.6 求平均值过程的流图

所谓独立路径是指至少引入程序的一个新处理语句集合或一个新条件的路径，用流图术语描述，独立路径至少包含一条在定义该路径之前不曾用过的边。

使用基本路径测试法设计测试用例时，程序的环形复杂度决定了程序中独立路径的数量，而且这个数是确保程序中所有语句至少被执行一次所需的测试数量的上界。

对于图 7.6 所描述的求平均值过程来说，由于环形复杂度为 6，因此共有 6 条独立路径。

通常在设计测试用例时，识别出判定结点是很有必要的。本例中结点 2、3、5、6 和 10 是判定结点。

第四步，设计可强制执行基本集合中每条路径的测试用例。

应该选取测试数据使得在测试每条路径时都适当地设置好了各个判定结点的条件。

在测试过程中，执行每个测试用例并把实际输出结果与预期结果相比较。一旦执行完所有测试用例，就可以确保程序中所有语句都至少被执行了一次，而且每个条件都分别取过 true 值和 false 值。

应该注意，某些独立路径不能以独立的方式测试，也就是说，程序的正常流程不能形成独立执行该路径所需要的数据组合。在这种情况下，这些路径必须作为另一个路径的一部分来测试。

2. 条件测试

用条件测试技术设计出的测试用例，能够检查程序模块中包含的逻辑条件。

一个简单条件是一个布尔变量或一个关系表达式，在布尔变量或关系表达式之前还可能有一个 NOT 算符。关系表达式的形式如下：

E1<关系算符>E2

其中，E1 和 E2 是算术表达式，而<关系算符>是下列算符之一：“<”，“≤”，“=”，“≠”，“>”或“≥”。

复合条件由两个或多个简单条件、布尔算符和括弧组成。布尔算符有 OR（“|”），AND（“&”）和 NOT。不包含关系表达式的条件称为布尔表达式。

因此，条件成分的类型包括布尔算符、布尔变量、布尔括弧（括住简单条件或复合条件）、关系算符及算术表达式。

如果条件不正确，则至少条件的一个成分不正确。因此，条件错误的类型如下：

布尔算符错（布尔算符不正确，遗漏布尔算符或有多余的布尔算符）

布尔变量错

布尔括弧错

关系算符错

算术表达式错

条件测试方法着重测试程序中的每个条件。

条件测试策略有两个优点： ①容易度量条件的测试覆盖率； ②程序内条件的测试覆盖率可指导附加测试的设计。

条件测试的目的不仅是检测程序条件中的错误，而且是检测程序中的其他错误。如果程序 P 的测试集能有效地检测 P 中条件的错误，则它很可能也可以有效地检测 P 中的其他错误。此外，如果一个测试策略对检测条件错误是有效的，则很可能该策略对检测程序的其他错误也是有效的。

人们已经提出了许多条件测试策略。分支测试可能是最简单的条件测试策略：对于复合条件 C 来说，C 的真分支和假分支以及 C 中的每个简单条件，都应该至少执行一次。

域测试要求对一个关系表达式执行 3 个或 4 个测试。对于形式为 E1<关系算符>E2 的关系表达式来说，需要 3 个测试分别使 E1 的值大于、等于或小于 E2 的值。如果<关系算符>错误而 E1 和 E2 正确，则这 3 个测试能够发现关系算符的错误。为了发现 E1 和 E2 中的错误，让 E1 值大于或小于 E2 值的测试数据应该使这两个值之间的差别尽可能小。

包含 n 个变量的布尔表达式需要 2n 个测试。这个策略可以发现布尔算符、变量和括弧的错误，但是，该策略仅在 n 很小时才是实用的。

在上述种种条件测试技术的基础上，K.C.Tai 提出了一种被称为 BRO(branch and relational operator)测试的条件测试策略。如果在条件中所有布尔变量和关系算符都只出现一次而且没有公共变量，则 BRO 测试保证能发现该条件中的分支错和关系算符错。

BRO 测试利用条件 C 的条件约束来设计测试用例。包含 n 个简单条件的条件 C 的条件约束定义为（D1，D2，…，Dn），其中 Di(0<i≤n)表示条件 C 中第 i 个简单条件的输出约束。如果在条件 C 的一次执行过程中，C 中每个简单条件的输出都满足 D 中对应的约束，则称 C 的这次执行覆盖了 C 的条件约束 D。

对于布尔变量 B 来说，B 的输出约束指出，B 必须是真（t）或假（f）。类似地，对于关系表达式来说，用符号>，=和<指定表达式的输出约束。

3. 循环测试

循环是绝大多数软件算法的基础，但是，在测试软件时却往往未对循环结构进行足够的测试。

循环测试是一种白盒测试技术，它专注于测试循环结构的有效性。在结构化的程序中通常只有 3 种循环，即简单循环、串接循环和嵌套循环，如图 7.7 所示。下面分别讨论这 3 种循环的测试方法。

（1） 简单循环。应该使用下列测试集来测试简单循环，其中 n 是允许通过循环的最大次数。

跳过循环。

只通过循环一次。

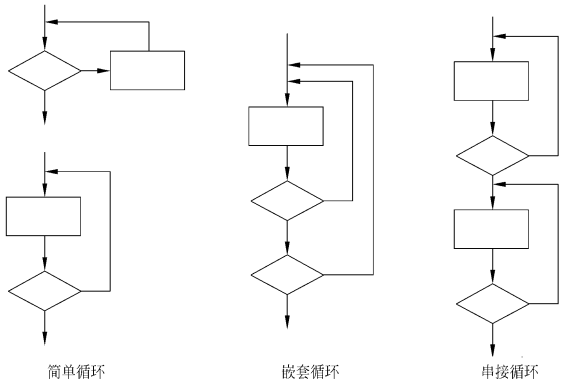


图 7.7 3 种循环

通过循环两次。

通过循环 m 次，其中 m<n-1。

通过循环 n-1,n,n+1 次。

（2） 嵌套循环。如果把简单循环的测试方法直接应用到嵌套循环，可能的测试数就会随嵌套层数的增加按几何级数增长。B.Beizer 提出了一种能减少测试数的方法：

从最内层循环开始测试，把所有其他循环都设置为最小值。

对最内层循环使用简单循环测试方法，而使外层循环的迭代参数（例如，循环计数器）取最小值，并为越界值或非法值增加一些额外的测试。

由内向外，对下一个循环进行测试，但保持所有其他外层循环为最小值，其他嵌套循环为“典型”值。

继续进行下去，直到测试完所有循环。

（3） 串接循环。如果串接循环的各个循环都彼此独立，则可以使用前述的测试简单循环的方法来测试串接循环。但是，如果两个循环串接，而且第一个循环的循环计数器值是第二个循环的初始值，则这两个循环并不是独立的。当循环不独立时，建议使用测试嵌套循环的方法来测试串接循环。

7.7 黑盒测试技术

黑盒测试着重测试软件功能。黑盒测试并不能取代白盒测试，它是与白盒测试互补的测试方法，它很可能发现白盒测试不易发现的其他类型的错误。

黑盒测试力图发现下述类型的错误： ①功能不正确或遗漏了功能； ②界面错误； ③数据结构错误或外部数据库访问错误； ④性能错误； ⑤初始化和终止错误。

白盒测试在测试过程的早期阶段进行，而黑盒测试主要用于测试过程的后期。设计黑盒测试方案时，应该考虑下述问题：

（1） 怎样测试功能的有效性？

（2） 哪些类型的输入可构成好测试用例？

（3） 系统是否对特定的输入值特别敏感？

（4） 怎样划定数据类的边界？

（5） 系统能够承受什么样的数据率和数据量？

（6） 数据的特定组合将对系统运行产生什么影响？

应用黑盒测试技术，能够设计出满足下述标准的测试用例集：

（1） 所设计出的测试用例能够减少为达到合理测试所需要设计的测试用例的总数；

（2） 所设计出的测试用例能够告诉我们，是否存在某些类型的错误，而不是仅仅指出与特定测试相关的错误是否存在。

7.7.1 等价划分

等价划分是一种黑盒测试技术，这种技术把程序的输入域划分成若干个数据类，据此导出测试用例。一个理想的测试用例能独自发现一类错误。

穷尽的黑盒测试通常是不现实的，只能选取少量最有代表性的输入数据作为测试数据，以期用较小的代价暴露出较多的程序错误。等价划分法力图设计出能发现若干类程序错误的测试用例，从而减少必须设计的测试用例的数目。

如果把所有可能的输入数据(有效的和无效的)划分成若干个等价类，则可以假定：每类中的一个典型值在测试中的作用与这一类中所有其他值的作用相同。因此，可以从每个等价类中只取一组数据作为测试数据。这样选取的测试数据最有代表性，最可能发现程序中的错误。

使用等价划分法设计测试方案首先需要划分输入数据的等价类，为此需要研究程序的功能说明，从而确定输入数据的有效等价类和无效等价类。在确定输入数据的等价类时常常还需要分析输出数据的等价类，以便根据输出数据的等价类导出对应的输入数据等价类。

划分等价类需要经验，下述几条启发式规则有助于等价类的划分：

（1） 如果规定了输入值的范围，则可划分出一个有效的等价类(输入值在此范围内)，两个无效的等价类(输入值小于最小值或大于最大值)；

（2） 如果规定了输入数据的个数，则可以划分出一个有效的等价类和两个无效的等价类；

（3） 如果规定了输入数据的一组值，而且程序对不同输入值做不同处理，则每个允许的输入值是一个有效的等价类，此外还有一个无效的等价类(任一个不允许的输入值)；

（4） 如果规定了输入数据必须遵循的规则，则可以划分出一个有效的等价类(符合规则)和若干个无效的等价类(从各种不同角度违反规则)；

（5） 如果规定了输入数据为整型，则可以划分出正整数、零和负整数等 3 个有效类；

（6） 如果程序的处理对象是表格，则应该使用空表，以及含一项或多项的表。

为了正确划分等价类，一是要注意积累经验，二是要正确分析被测程序的功能。

一般说来，不需要设计测试数据用来暴露编译程序肯定能发现的错误。

划分出等价类以后，根据等价类设计测试方案时主要使用下面两个步骤：

(1) 设计一个新的测试方案以尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步骤直到所有有效等价类都被覆盖为止；

(2) 设计一个新的测试方案，使它覆盖一个而且只覆盖一个尚未被覆盖的无效等价类，重复这一步骤直到所有无效等价类都被覆盖为止。

注意，通常程序发现一类错误后就不再检查是否还有其他错误，因此，应该使每个测试方案只覆盖一个无效的等价类。

下面用等价划分法设计一个简单程序的测试方案。

假设有一个把数字串转变成整数的函数。运行程序的计算机字长 16 位，用二进制补码表示整数。这个函数是用 Pascal 语言编写的，它的说明如下：

function strtoint (dstr:shortstr):integer;

函数的参数类型是 shortstr,它的说明是:

type shortstr=array［1..6］ of char;

被处理的数字串是右对齐的，也就是说，如果数字串比 6 个字符短，则在它的左边补空格。如果数字串是负的，则负号和最高位数字紧相邻(负号在最高位数字左边一位)。

考虑到 Pascal 编译程序固有的检错功能，测试时不需要使用长度不等于 6 的数组做实参，更不需要使用任何非字符数组类型的实参。

分析这个程序的规格说明，可以划分出如下等价类：

有效输入的等价类有

(1) 1～6 个数字字符组成的数字串(最高位数字不是零)；

(2) 最高位数字是零的数字串；

(3) 最高位数字左邻是负号的数字串；

无效输入的等价类有

(4) 空字符串(全是空格)；

(5) 左部填充的字符既不是零也不是空格；

(6) 最高位数字右面由数字和空格混合组成；

(7) 最高位数字右面由数字和其他字符混合组成；

(8) 负号与最高位数字之间有空格；

合法输出的等价类有

(9) 在计算机能表示的最小负整数和零之间的负整数；

(10) 零；

(11) 在零和计算机能表示的最大正整数之间的正整数；

非法输出的等价类有

(12) 比计算机能表示的最小负整数还小的负整数；

(13) 比计算机能表示的最大正整数还大的正整数。

因为所用的计算机字长 16 位，用二进制补码表示整数，所以能表示的最小负整数是-32 768，能表示的最大正整数是 32 767。

7.7.2 边界值分析

经验表明，处理边界情况时程序最容易发生错误。例如，

许多程序错误出现在下标、数据结构和循环等等的边界附近。因此，设计使程序运行在边界情况附近的测试方案，暴露出程序错误的可能性更大一些。

使用边界值分析方法设计测试方案首先应该确定边界情况，通常输入等价类和输出等价类的边界，就是应该着重测试的程序边界情况。选取的测试数据应该等于、刚刚小于和刚刚大于边界值。

通常设计测试方案时总是联合使用等价划分和边界值分析两种技术。

7.7.3　错误推测

有时分别使用每组测试数据时程序都能正常工作，这些输入数据的组合却可能检测出程序的错误。因此必须依靠测试人员的经验和直觉，从各种可能的测试方案中选出一些最可能引起程序出错的方案。对于程序中可能存在哪类错误的推测，是挑选测试方案时的一个重要因素。

错误推测法在很大程度上靠直觉和经验进行。它的基本想法是列举出程序中可能有的错误和容易发生错误的特殊情况，并且根据它们选择测试方案。

应该分析程序规格说明书，找出其中遗漏或省略的部分，以便设计相应的测试方案，检测程序员对这些部分的处理是否正确。

选择输入组合的另一个有效途径是把计算机测试和人工检查代码结合起来。

例如，通过代码检查发现程序中两个模块使用并修改某些共享的变量，如果一个模块对这些变量的修改不正确，则会引起另一个模块出错，因此这是程序发生错误的一个可能的原因。应该设计测试方案，在程序的一次运行中同时检测这两个模块，特别要着重检测一个模块修改了共享变量后，另一个模块能否像预期的那样正常使用这些变量。

7.8　调试

调试（也称为纠错）作为成功测试的后果出现，也就是说，调试是在测试发现错误之后排除错误的过程。软件工程师在评估测试结果时，往往仅面对着软件错误的症状，也就是说，软件错误的外部表现和它的内在原因之间可能并没有明显的联系。调试就是把症状和原因联系起来的尚未被人深入认识的智力过程。

7.8.1　调试过程

调试不是测试，但是它总是发生在测试之后。调试过程从执行一个测试用例开始，评估测试结果，如果发现实际结果与预期结果不一致，则这种不一致就是一个症状，它表明在软件中存在着隐藏的问题。调试过程试图找出产生症状的原因，以便改正错误。

调试过程总会有以下两种结果之一：①找到了问题的原因并把问题改正和排除掉了；②没找出问题的原因。在后一种情况下，调试人员可以猜想一个原因，并设计测试用例来验证这个假设，重复此过程直至找到原因并改正了错误。

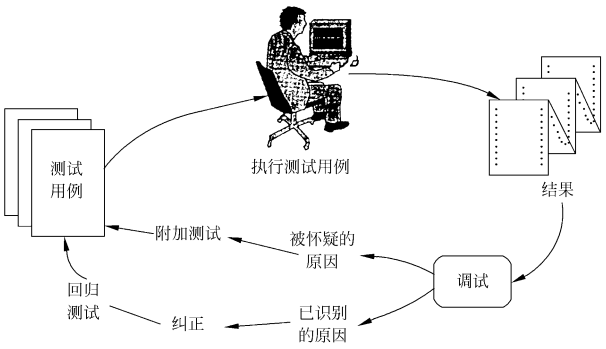


图 7.8 调试过程

软件错误的特征：

（1）　症状和产生症状的原因可能在程序中相距甚远，也就是说，症状可能出现在程序的一个部分，而实际的原因可能在与之相距很远的另一部分。

（2）　当改正了另一个错误之后，症状可能暂时消失了。

（3）　症状可能实际上并不是由错误引起的（例如，舍入误差）。

（4）　症状可能是由不易跟踪的人为错误引起的。

（5）　症状可能是由定时问题而不是由处理问题引起的。

（6）　可能很难重新产生完全一样的输入条件。

（7）　症状可能时有时无。

（8）　症状可能是由分布在许多任务中的原因引起的，这些任务运行在不同的处理机上。

7.8.2　调试途径

无论采用什么方法，调试的目标都是寻找软件错误的原因并改正错误。通常需要把系统地分析、直觉和运气组合起来，才能实现上述目标。一般说来，有下列 3 种调试途径可以采用：

3. 原因排除法

对分查找法、归纳法和演绎法都属于原因排除法。

对分查找法的基本思路是，如果已经知道每个变量在程序内若干个关键点的正确值，则可以用赋值语句或输入语句在程序中点附近“注入”这些变量的正确值，然后运行程序并检查所得到的输出。如果输出结果是正确的，则错误原因在程序的前半部分；反之，错误原因在程序的后半部分。对错误原因所在的那部分再重复使用这个方法，直到把出错范围缩小到容易诊断的程度为止。

归纳法是从个别现象推断出一般性结论的思维方法。使用这种方法调试程序时，首先把和错误有关的数据组织起来进行分析，以便发现可能的错误原因。然后导出对错误原因的一个或多个假设，并利用已有的数据来证明或排除这些假设。

演绎法从一般原理或前提出发，经过排除和精化的过程推导出结论。采用这种方法调试程序时，首先设想出所有可能的出错原因，然后试图用测试来排除每一个假设的原因。如果测试表明某个假设的原因可能是真的原因，则对数据进行细化以准确定位错误。

2. 回溯法

回溯是一种相当常用的调试方法，当调试小程序时这种方法是有效的。具体做法是，从发现症状的地方开始，人工沿程序的控制流往回追踪分析源程序代码，直到找出错误原因为止。但是，随着程序规模扩大，应该回溯的路径数目也变得越来越大，以至彻底回溯变成完全不可能了。

1. 蛮干法

蛮干法可能是寻找软件错误原因的最低效的方法。仅当所有其他方法都失败了的情况下，才应该使用这种方法。这种方法印出内存的内容，激活对运行过程的跟踪，并在程序中到处都写上 WRITE 语句，希望能发现错误的线索。

上述 3 种调试途径都可以使用调试工具辅助完成，但是工具并不能代替对全部设计文档和源程序的仔细分析与评估。在使用任何一种调试方法之前，必须首先进行周密的思考，必须有明确的目的，应该尽量减少无关信息的数量。

如果用遍了各种调试方法和调试工具却仍然找不出错误原因，则应该向同行求助。把遇到的问题向同行陈述并一起分析讨论，往往能开阔思路，较快找出错误原因。

一旦找到错误就必须改正它，但是，改正一个错误可能引入更多的其他错误，以至“得不偿失”。因此，在动手改正错误之前，软件工程师应该仔细考虑下述 3 个问题：

(1) 是否同样的错误也在程序其他地方存在？在许多情况下，一个程序错误是由错误的逻辑思维模式造成的，而这种逻辑思维模式也可能用在别的地方。仔细分析这种逻辑模式，有可能发现其他错误。

（2） 将要进行的修改可能会引入的“下一个错误”是什么？在改正错误之前应该仔细研究源程序（最好也研究设计文档），以评估逻辑和数据结构的耦合程度。如果所要做的修改位于程序的高耦合段中，则修改时必须特别小心谨慎。

（3） 为防止今后出现类似的错误，应该做什么？如果不仅修改了软件产品还改进了开发软件产品的软件过程，则不仅排除了现有程序中的错误，还避免了今后在程序中可能出现的错误。

7.9 软件可靠性

7.9.1 基本概念

1. 软件可靠性的定义

对于软件可靠性有许多不同的定义，其中多数人承认的一个定义是：软件可靠性是程序在给定的时间间隔内，按照规格说明书的规定成功地运行的概率。

在上述定义中包含的随机变量是时间间隔。显然，随着运行时间的增加，运行时出现程序故障的概率也将增加，即可靠性随着给定的时间间隔的加大而减少。

按照 IEEE 的规定，术语“错误”的含义是由开发人员造成的软件差错（bug），而术语“故障”的含义是由错误引起的软件的不正确行为。在下面的论述中，将按照 IEEE 规定的含义使用这两个术语。

2. 软件的可用性

通常用户也很关注软件系统可以使用的程度。一般说来，对于任何其故障是可以修复的系统，都应该同时使用可靠性和可用性衡量它的优劣程度。

软件可用性的一个定义是： 软件可用性是程序在给定的时间点，按照规格说明书的规定，成功地运行的概率。

可靠性和可用性之间的主要差别是，可靠性意味着在 0 到 t 这段时间间隔内系统没有失效，而可用性只意味着在时刻 t，系统是正常运行的。

如果在一段时间内，软件系统故障停机时间分别为 td1，td2，…，正常运行时间分别为 tu1，tu2，…，则系统的稳态可用性为：

$$Ass=Tup/(Tup+Tdown) \quad (7.1)$$

其中 $Tup=\sum tui$，$Tdown=\sum tdi$

如果引入系统平均无故障时间 MTTF 和平均维修时间 MTTR 的概念，则(7.1)式可以变成

$$Ass=MTTF/(MTTF+MTTR) \quad (7.2)$$

平均维修时间 MTTR 是修复一个故障平均需要用的时间，它取决于维护人员的技术水平和对系统的熟悉程度。

平均无故障时间 MTTF 是系统按规格说明书规定成功地运行的平均时间，它主要取决于系统中潜伏的错误的数目，因此和测试的关系十分密切。

7.9.2 估算平均无故障时间的方法

软件的平均无故障时间 MTTF 是一个重要的质量指标，往往作为对软件的一项要求，由用户提出来。为了估算 MTTF，首先引入一些有关的量。

1. 符号

在估算 MTTF 的过程中使用下述符号表示有关的数量：

ET——测试之前程序中错误总数；

IT——程序长度(机器指令总数)；

τ——测试(包括调试)时间；

Ed(τ)——在 0 至 τ 期间发现的错误数；

Ec(τ)——在 0 至 τ 期间改正的错误数。

2. 基本假定

根据经验数据，可以作出下述假定。

(1) 在类似的程序中，单位长度里的错误数 ET/IT 近似为常数。通常 $0.5\times10\text{-}2\le ET/IT\le 2\times10\text{-}2$，也就是说，在测试之前每 1000 条指令中大约有 5～20 个错误。

(2) 失效率正比于软件中剩余的(潜藏的)错误数，而平均无故障时间 MTTF 与剩余的错误数成反比。

（3） 假设发现的每一个错误都立即正确地改正了(即调试过程没有引入新的错误)。因此

$$Ec(\tau)=Ed(\tau)$$

剩余的错误数为

$$Er(\tau)=ET-Ec(\tau) \quad (7.3)$$

单位长度程序中剩余的错误数为

$$\varepsilon r(\tau)=ET/Ir - Ec(\tau)/IT \qquad (7.4)$$

3. 估算平均无故障时间

经验表明，平均无故障时间与单位长度程序中剩余的错误数成反比，即

$$MTTF=1/[K(ET/IT-Ec(\tau)/IT)] \qquad (7.5)$$

其中 K 为常数，它的值应该根据经验选取。美国的一些统计数字表明，K 的典型值是 200。

估算平均无故障时间的公式，可以评价软件测试的进展情况。此外，由(7.5)式可得

$$Ec=ET-IT/(K\times MTTF) \qquad (7.6)$$

因此，也可以根据对软件平均无故障时间的要求，估计需要改正多少个错误之后，测试工作才能结束。

4. 估计错误总数的方法

程序中潜藏的错误的数目是一个十分重要的量，它既直接标志软件的可靠程度，又是计算软件平均无故障时间的重要参数。显然，程序中的错误总数 ET 与程序规模、类型、开发环境、开发方法论、开发人员的技术水平和管理水平等都有密切关系。下面介绍估计 ET 的两个方法。

(1) 植入错误法

使用这种估计方法，在测试之前由专人在程序中随机地植入一些错误，测试之后，根据测试小组发现的错误中原有的和植入的两种错误的比例，来估计程序中原有错误的总数 ET。

假设人为地植入的错误数为 Ns,经过一段时间的测试之后发现 ns 个植入的错误，此外还发现了 n 个原有的错误。如果可以认为测试方案发现植入错误和发现原有错误的能力相同，则能够估计出程序中原有错误的总数为

$$N^=n/ns\times Ns \qquad (7.7)$$

其中 N^即是错误总数 ET 的估计值。

(2) 分别测试法

植入错误法的基本假定是所用的测试方案发现植入错误和发现原有错误的概率相同。但是，人为地植入的错误和程序中原有的错误可能性质很不相同，发现它们的难易程度自然也不相同，因此，上述基本假定可能有时和事实不完全一致。

如果有办法随机地把程序中一部分原有的错误加上标记，然后根据测试过程中发现的有标记错误和无标记错误的比例，估计程序中的错误总数，则这样得出的结果比用植入错误法得到的结果更可信一些。

为了随机地给一部分错误加标记，分别测试法使用两个测试员(或测试小组)，彼此独立地测试同一个程序的两个副本，把其中一个测试员发现的错误作为有标记的错误。具体做法是，在测试过程的早期阶段，由测试员甲和测试员乙分别测试同一个程序的两个副本，由另一名分析员分析他们的测试结果。用 τ 表示测试时间，假设

τ=0 时错误总数为 B0;

τ=τ1 时测试员甲发现的错误数为 B1;

τ=τ1 时测试员乙发现的错误数为 B2;

τ=τ1 时两个测试员发现的相同错误数为 bc。

如果认为测试员甲发现的错误是有标记的，即程序中有标记的错误总数为 B1,则测试员乙发现的 B2 个错误中有 bc 个是有标记的。假定测试员乙发现有标记错误和发现无标记错误的概率相同，则可以估计出测试前程序中的错误总数为

$$B0^ =B2/bcB1 \qquad (7.8)$$

使用分别测试法，在测试阶段的早期，每隔一段时间分析员分析两名测试员的测试结果，并且用(7.8)式计算 B0^。如果几次估算的结果相差不多，则可用 B0^的平均值作为 ET 的估计值。此后一名测试员可以改做其他工作，由余下的一名测试员继续完成测试工作，因为他可以继承另一名测试员的测试结果，所以分别测试法增加的测试成本并不太多。

7.10　小结

实现包括编码和测试两个阶段。

按照传统的软件工程方法学，编码是在对软件进行了总体设计和详细设计之后进行的，程序的质量基本上取决于设计的质量。但是，编码使用的语言，特别是写程序的风格，也对程序质量有相当大的影响。

程序内部的良好文档资料，有规律的数据说明格式，简单清晰的语句构造和输入输出格式等等，都对提高程序的可读性有很大作用，也在相当大的程度上改进了程序的可维护性。

软件测试是保证软件可靠性的主要手段。测试阶段的根本任务是发现并改正软件中的错误。

软件测试是软件开发过程中最艰巨最繁重的任务，软件测试至少分为单元测试、集成测试和验收测试 3 个基本阶段。

设计测试方案是测试阶段的关键技术问题，基本目标是选用最少量的高效测试数据，做到尽可能完善的测试，从而尽可能多地发现软件中的问题。

软件测试不仅仅指利用计算机进行的测试，还包括人工进行的测试如代码审查。两种测试途径各有优缺点，互相补充，缺一不可。

白盒测试和黑盒测试是软件测试的两类基本方法，这两类方法各有所长，相互补充。通常，在测试过程的早期阶段主要使用白盒方法，而在测试过程的后期阶段主要使用黑盒方法。

设计白盒测试方案的技术主要有，逻辑覆盖和控制结构测试；

设计黑盒测试方案的技术主要有，等价划分、边界值分析和错误推测。

在测试过程中发现的软件错误必须及时改正，这就是调试的任务。

测试和调试是软件测试阶段中的两个关系非常密切的过

程，它们往往交替进行。

程序中潜藏的错误的数目，直接决定了软件的可靠性。根据测试和调试过程中已经发现和改正的错误数，可以估算软件的平均无故障时间；反之，根据要求达到的软件平均无故障时间，可以估算出应该改正的错误数，从而能够判断测试阶段何时可以结束。

第 8 章　维护

在软件产品被开发出来并交付用户使用之后，就进入了软件的运行维护阶段。这个阶段是软件生命周期的最后一个阶段，其基本任务是保证软件在一个相当长的时期能够正常运行。

软件维护需要的工作量很大，平均说来，大型软件的维护成本高达开发成本的 4 倍左右。

软件工程的目的是要提高软件的可维护性，减少软件维护所需要的工作量，降低软件系统的总成本。

8.1　软件维护的定义

所谓软件维护就是在软件已经交付使用之后，为了改正错误或满足新的需要而修改软件的过程。可以通过描述软件交付使用后可能进行的 4 项活动，具体地定义软件维护。

改正性维护：在程序的使用期间，用户发现程序错误，并且把问题报告给维护人员。把诊断和改正错误的过程称为改正性维护。

适应性维护：为了和变化了的环境适当地配合而进行的修改软件的活动，是既必要又经常的维护活动。

完善性维护：在使用软件的过程中用户往往提出增加新功能或修改已有功能的建议，还可能提出一般性的改进意见。为了满足这类要求，需要进行完善性维护。这项维护活动通常占软件维护工作的大部分。

预防性维护：当为了改进未来的可维护性或可靠性，或为了给未来的改进奠定更好的基础而修改软件时，这项维护活动通常称为预防性维护，目前这项维护活动相对比较少。

8.2　软件维护的特点

8.2.1　结构化维护与非结构化维护差别巨大

结构化维护

维护工作从评价设计文档开始，确定软件重要的结构特点、性能特点以及接口特点；估量要求的改动将带来的影响，并且计划实施途径。然后修改设计并且对所做的修改进行仔细复查。接下来编写相应的源程序代码；使用在测试说明书中包含的信息进行回归测试；最后，把修改后的软件交付使用。

8.2.2　维护的代价高昂

在过去的几十年中，软件维护的费用稳步上升。1970 年用于维护已有软件的费用只占软件总预算的 35%～40%，1990 年上升为 70%～80%。

软件维护的无形的代价：

可用的资源必须供维护任务使用，以致耽误了开发的良机；

当看来合理的有关改错或修改的要求不能及时满足时将引起用户不满；

由于维护时的改动，在软件中引入了潜伏的错误，从而降低了软件的质量；

当必须把软件工程师调去从事维护工作时，将在开发过程中造成混乱。

软件维护的最后一个代价是生产率的大幅度下降，这种情况在维护旧程序时常常遇到。

8.2.3　维护的问题

（1）　理解别人写的程序通常非常困难，而且困难程度随着软件配置成分的减少而迅速增加。

（2）　需要维护的软件往往没有合格的文档，或者文档资料显著不足。

（3）　当需要对软件进行维护时，往往原来写程序的人已经不在该项目组中了。

（4）　绝大多数软件在设计时没有充分考虑将来的修改。

（5）　软件维护不是一项吸引人的工作。形成这种观念很大程度上是因为维护工作经常遭受挫折。

软件工程至少部分地解决了与维护有关的每一个问题。

8.3　软件维护过程

维护过程本质上是修改和压缩了的软件定义和开发过程。

首先必须建立一个维护组织，随后必须确定报告和评价的过程，而且必须为每个维护要求规定一个标准化的事件序列。此外，还应该建立一个适用于维护活动的记录保管过程，并且规定复审标准。

1. 维护组织

通常并不需要建立正式的维护组织。

每个维护要求都通过维护管理员转交给相应的系统管理员去评价。系统管理员对维护任务做出评价之后，由变化授权人决定应该进行的活动。图 8.1 描绘了上述组织方式。

在维护活动开始之前就明确维护责任是十分必要的，这样做可以大大减少维护过程中可能出现的混乱。

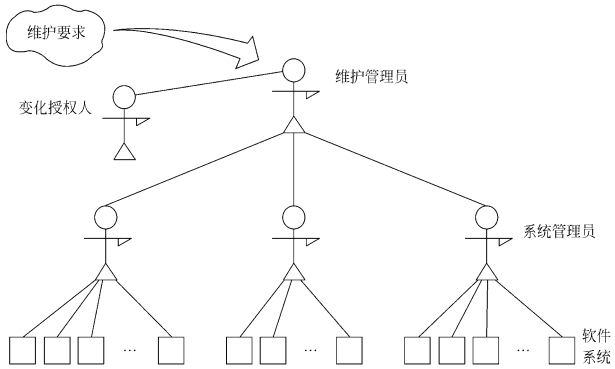


图 8.1　维护组织

2. 维护报告

应该用标准化的格式表达所有软件维护要求，称为软件问题报告表，这个表格由用户填写。

如果遇到了一个错误，那么必须完整描述导致出现错误的

环境。对于适应性或完善性的维护要求，应该提出一个简短的需求说明书。

由维护管理员和系统管理员评价用户提交的维护要求表。维护要求表是计划维护活动的基础。

软件组织内部应该制定出一个软件修改报告，它给出下述信息：

(1) 满足维护要求表中提出的要求所需要的工作量；

(2) 维护要求的性质；

(3) 要求的优先次序；

(4) 与修改有关的事后数据。

在拟定进一步的维护计划之前，把软件修改报告提交给变化授权人审查批准。

3. 维护的事件流

不管维护类型如何，都需要进行同样的技术工作。这些工作包括修改软件设计、复查、必要的代码修改、单元测试和集成测试(包括使用以前的测试方案的回归测试)、验收测试和复审。不同类型的维护强调的重点不同，但是基本途径是相同的。

维护事件流中最后一个事件是复审，它再次检验软件配置的所有成分的有效性，并且保证事实上满足了维护要求表中的要求。

4. 保存维护记录

保存维护记录需要保存下述内容：

①程序标识； ②源语句数； ③机器指令条数； ④使用的程序设计语言； ⑤程序安装的日期； ⑥自从安装以来程序运行的次数； ⑦自从安装以来程序失效的次数； ⑧程序变动的层次和标识； ⑨因程序变动而增加的源语句数； 因程序变动而删除的源语句数； 每个改动耗费的人时数； 程序改动的日期； 软件工程师的名字； 维护要求表的标识； 维护类型； 维护开始和完成的日期； 累计用于维护的人时数； 与完成的维护相联系的纯效益。

应该为每项维护工作都收集上述数据。可以利用这些数据构成一个维护数据库的基础，并且对它们进行评价。

5. 评价维护活动

(1) 每次程序运行平均失效的次数；

(2) 用于每一类维护活动的总人时数；

(3) 平均每个程序、每种语言、每种维护类型所做的程序变动数；

(4) 维护过程中增加或删除一个源语句平均花费的人时数；

(5) 维护每种语言平均花费的人时数；

(6) 一张维护要求表的平均周转时间；

(7) 不同维护类型所占的百分比。

8.4 软件的可维护性

把软件的可维护性定义为： 维护人员理解、改正、改动或改进这个软件的难易程度。

提高可维护性是支配软件工程方法学所有步骤的关键目标。

8.4.1 决定软件可维护性的因素

在维护的修改之后应该进行必要的测试，以保证所做的修改是正确的。

如果是改正性维护，还必须预先进行调试以确定错误的具体位置。因此，决定软件可维护性的因素主要有下述 5 个：

1. 可理解性

软件可理解性表现为理解软件的结构、功能、接口和内部处理过程的难易程度。模块化、详细的设计文档、结构化设计、程序内部的文档和良好的高级程序设计语言等等，都对提高软件的可理解性有重要贡献。

2. 可测试性

诊断和测试的容易程度取决于软件容易理解的程度。良好的文档对诊断和测试是至关重要的，此外，软件结构、可用的测试工具和调试工具，以及以前设计的测试过程也都是非常重要的。维护人员应该能够得到在开发阶段用过的测试方案，以便进行回归测试。

对于程序模块来说，可以用程序复杂度来度量它的可测试性。模块的环形复杂度越大，可执行的路径就越多，因此，全面测试它的难度就越高。

3. 可修改性

耦合、内聚、信息隐藏、局部化、控制域与作用域的关系等等，都影响软件的可修改性。

4. 可移植性

软件可移植性指的是，把程序从一种计算环境（硬件配置和操作系统）转移到另一种计算环境的难易程度。把与硬件、操作系统以及其他外部设备有关的程序代码集中放到特定的程序模块中，可以把因环境变化而必须修改的程序局限在少数程序模块中，从而降低修改的难度。

5. 可重用性

所谓重用（reuse）是指同一事物不做修改或稍加改动就在不同环境中多次重复使用。

使用可重用的软件构件来开发软件，可靠性比较高，且在每次重用过程中都会发现并清除一些错误，随着时间推移，这样的构件将变成实质上无错误的。因此，软件中使用的可重用构件越多，软件的可靠性越高，改正性维护需求越少，适应性和完善性维护也就越容易。

8.4.2 文档

文档是影响软件可维护性的决定因素。软件在使用过程中必然会经受多次修改，所以文档比程序代码更重要。

软件系统的文档可以分为用户文档和系统文档两类。用户文档主要描述系统功能和使用方法，并不关心这些功能是怎样实现的；系统文档描述系统设计、实现和测试等各方面的内容。

总的说来，软件文档应该满足下述要求：

(1) 必须描述如何使用这个系统；

(2) 必须描述怎样安装和管理这个系统；
(3) 必须描述系统需求和设计；
(4) 必须描述系统的实现和测试，以便使系统成为可维护的。

1. 用户文档

用户文档是用户了解系统的第一步，它应该能使用户获得对系统的准确的初步印象。文档的结构方式应该使用户能够方便地根据需要阅读有关的内容。

用户文档至少应该包括下述 5 方面的内容：

(1) 功能描述，说明系统能做什么；
(2) 安装文档，说明怎样安装这个系统以及怎样使系统适应特定的硬件配置；
(3) 使用手册，简要说明如何着手使用这个系统；
(4) 参考手册，详尽描述用户可以使用的所有系统设施以及它们的使用方法，还应该解释系统可能产生的各种出错信息的含义；
(5) 操作员指南(如果需要有系统操作员的话)，说明操作员应该如何处理使用中出现的各种情况。

上述内容可以分别作为独立的文档，也可以作为一个文档的不同分册，具体做法应该由系统规模决定。

2. 系统文档

所谓系统文档指从问题定义、需求说明到验收测试计划这样一系列和系统实现有关的文档。描述系统设计、实现和测试的文档对于理解程序和维护程序来说是极端重要的。和用户文档类似，系统文档的结构也应该能把读者从对系统概貌的了解，引导到对系统每个方面每个特点的更形式化更具体的认识。

8.4.3 可维护性复审

可维护性是所有软件都应该具备的基本特点，必须在开发阶段保证软件具有可维护性。在软件工程过程的每一个阶段都应该考虑并努力提高软件的可维护性，在每个阶段结束前的技术审查和管理复审中，应该着重对可维护性进行复审。

在需求分析阶段的复审过程中，应该对将来要改进的部分和可能会修改的部分加以注意并指明；应该讨论软件的可移植性问题，并且考虑可能影响软件维护的系统界面。

在正式的和非正式的设计复审期间，应该从容易修改、模块化和功能独立的目标出发，评价软件的结构和过程；设计中应该对将来可能修改的部分预作准备。

代码复审应该强调编码风格和内部说明文档这两个影响可维护性的因素。

在设计和编码过程中应该尽量使用可重用的软件构件。

在测试结束时进行最正式的可维护性复审，称为配置复审。配置复审的目的是保证软件配置的所有成分是完整的、一致的和可理解的，而且为了便于修改和管理已经编目归档了。

在完成了每项维护工作之后，都应该对软件维护本身进行仔细认真的复审。

维护应该针对整个软件配置，不应该只修改源程序代码。当对源程序代码的修改没有反映在设计文档或用户手册中时，就会产生严重的后果。

每当对数据、软件结构、模块过程或任何其他有关的软件特点做了改动时，必须立即修改相应的技术文档。不能准确反映软件当前状态的设计文档可能比完全没有文档更坏。在以后的维护工作中很可能因文档不完全符合实际而不能正确理解软件，从而在维护中引入过多的错误。

8.5 预防性维护

当初开发这些老程序时并没有使用软件工程方法学来指导，文档不全甚至完全没有文档，对曾经做过的修改也没有完整的记录。

怎样满足用户对上述这类老程序的维护要求呢？为了修改这类程序以适应用户新的或变更的需求，有以下几种做法可供选择：

（1） 多次地做修改程序的尝试，以实现所要求的修改；
（2） 通过分析程序尽可能多地掌握程序的内部工作细节，以便更有效地修改它；
（3） 在深入理解原有设计的基础上，用软件工程方法重新设计、重新编码和测试那些需要变更的软件部分；
（4） 以软件工程方法学为指导，对程序全部重新设计、重新编码和测试，为此可以使用 CASE 工具来帮助理解原有的设计。通常人们采用后 3 种做法。其中第 4 种做法称为软件再工程。

预防性维护方法是由 Miller 提出来的，他把这种方法定义为：“把今天的方法学应用到昨天的系统上，以支持明天的需求。”

8.6 软件再工程过程

典型的软件再工程过程模型如图 8.3 所示，该模型定义了 6 类活动。在某些情况下这些活动以线性顺序发生，但也并非总是这样，例如，为了理解某个程序的内部工作原理，可能在文档重构开始之前必须先进行逆向工程。

在图 8.3 中显示的再工程范型是一个循环模型。这意味着作为该范型的组成部分的每个活动都可能被重复，而且对于任意一个特定的循环来说，过程可以在完成任意一个活动之后终止。下面简要地介绍该模型所定义的 6 类活动。

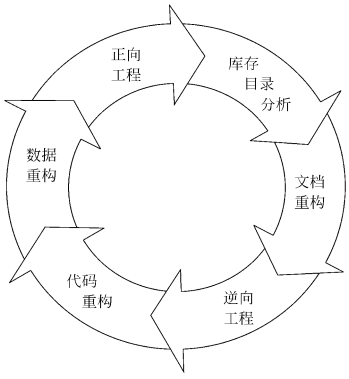


图 8.3 软件再工程过程模型

1. 库存目录分析

每个软件组织都应该保存其拥有的所有应用系统的库存目录。该目录包含关于每个应用系统的基本信息（例如，应用系统的名字，最初构建它的日期，已做过的实质性修改次数，过去 18 个月报告的错误，用户数量，安装它的机器数量，它的复杂程度，文档质量，整体可维护性等级，预期寿命，在未来 36 个月内的预期修改次数，业务重要程度等）。

每一个大的软件开发机构都拥有上百万行老代码，它们都可能是逆向工程或再工程的对象。但是，某些程序并不频繁使用而且不需要改变，此外，逆向工程和再工程工具尚不成熟，目前仅能对有限种类的应用系统执行逆向工程或再工程，代价又十分高昂，因此，对库中每个程序都做逆向工程或再工程是不现实的。下述 3 类程序有可能成为预防性维护的对象：

（1） 预定将使用多年的程序；

（2） 当前正在成功地使用着的程序；

（3） 在最近的将来可能要做重大修改或增强的程序。

应该仔细分析库存目录，按照业务重要程度、寿命、当前可维护性、预期的修改次数等标准，把库中的应用系统排序，从中选出再工程的候选者，然后明智地分配再工程所需要的资源。

2. 文档重构

老程序固有的特点是缺乏文档。具体情况不同，处理这个问题的方法也不同：

（1） 建立文档非常耗费时间，不可能为数百个程序都重新建立文档。如果一个程序是相对稳定的，正在走向其有用生命的终点，而且可能不会再经历什么变化，那么，让它保持现状是一个明智的选择。

（2） 为了便于今后的维护，必须更新文档，但是由于资源有限，应采用“使用时建文档”的方法，也就是说，不是一下子把某应用系统的文档全部都重建起来，而是只针对系统中当前正在修改的那些部分建立完整文档。随着时间流逝，将得到一组有用的和相关的文档。

（3） 如果某应用系统是完成业务工作的关键，而且必须重构全部文档，则仍然应该设法把文档工作减少到必需的最小量。

3. 逆向工程

软件的逆向工程是分析程序以便在比源代码更高的抽象层次上创建出程序的某种表示的过程，也就是说，逆向工程是一个恢复设计结果的过程，逆向工程工具从现存的程序代码中抽取有关数据、体系结构和处理过程的设计信息。

4. 代码重构

代码重构是最常见的再工程活动。某些老程序具有比较完整、合理的体系结构，但是，个体模块的编码方式却是难于理解、测试和维护的。在这种情况下，可以重构可疑模块的代码。

为了完成代码重构活动，首先用重构工具分析源代码，标注出和结构化程序设计概念相违背的部分。然后重构有问题的代码（此项工作可自动进行）。最后，复审和测试生成的重构代码（以保证没有引入异常）并更新代码文档。

通常，重构并不修改整体的程序体系结构，它仅关注个体模块的设计细节以及在模块中定义的局部数据结构。如果重构扩展到模块边界之外并涉及软件体系结构，则重构变成了正向工程。

5. 数据重构

对数据体系结构差的程序很难进行适应性修改和增强，事实上，对许多应用系统来说，数据体系结构比源代码本身对程序的长期生存力有更大影响。

与代码重构不同，数据重构发生在相当低的抽象层次上，它是一种全范围的再工程活动。在大多数情况下，数据重构始于逆向工程活动，分解当前使用的数据体系结构，必要时定义数据模型，标识数据对象和属性，并从软件质量的角度复审现存的数据结构。

当数据结构较差时，应该对数据进行再工程。

由于数据体系结构对程序体系结构及程序中的算法有很大影响，对数据的修改必然会导致体系结构或代码层的改变。

6. 正向工程

正向工程也称为革新或改造，这项活动不仅从现有程序中恢复设计信息，而且使用该信息去改变或重构现有系统，以提高其整体质量。

正向工程过程应用软件工程的原理、概念、技术和方法来重新开发某个现有的应用系统。在大多数情况下，被再工程的软件不仅重新实现现有系统的功能，而且加入了新功能和提高了整体性能。

8.7 小结

维护是软件生命周期的最后一个阶段，也是持续时间最长代价最大的一个阶段。软件工程学的主要目的就是提高软件的可维护性，降低维护的代价。

软件维护通常包括 4 类活动：改正性维护、适应性维护、完善性维护以及预防性维护。

软件的可理解性、可测试性、可修改性、可移植性和可重用性，是决定软件可维护性的基本因素，软件重用技术是能从根本上提高软件可维护性的重要技术。

软件生命周期每个阶段的工作都和软件可维护性有密切关系。良好的设计，完整准确易读易理解的文档资料，以及一系列严格的复审和测试，使得一旦发现错误时比较容易诊断和纠正，当用户有新要求或外部环境变化时软件能较容易地适应，并且能够减少维护引入的错误。因此，在软件生命周期的每个阶段都必须充分考虑维护问题，并且为软件维护做准备。

文档是影响软件可维护性的决定因素，因此，文档甚至比

可执行的程序代码更重要。文档可分为用户文档和系统文档两大类。不管是哪一类文档都必须和程序代码同时维护，只有和程序代码完全一致的文档才是真正有价值的文档。

目前预防性维护在全部维护活动中仅占很小比例，但是不应该忽视这类维护活动，在条件具备时应该主动地进行预防性维护。

预防性维护实质上是软件再工程。典型的软件再工程过程模型定义了库存目录分析、文档重构、逆向工程、代码重构、数据重构和正向工程等 6 类活动。上述模型是一个循环模型，这意味着每项活动都可能被重复，而且对于任意一个特定的循环来说，再工程过程可以在完成任意一个活动之后终止。

第 13 章　软件项目管理

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

软件项目管理先于任何技术活动之前开始，并且贯穿于软件的整个生命周期之中。

软件项目管理过程从一组项目计划活动开始，而制定计划的基础是工作量估算和完成期限估算。为了估算项目的工作量和完成期限，首先需要估算软件的规模。

13.1　估算软件规模

13.1.1　代码行技术

代码行技术是比较简单的定量估算软件规模的方法。这种方法依据经验和历史数据，估计实现一个功能所需要的源程序行数。当有以往开发类似产品的历史数据可供参考时，用这种方法估计出的数值还是比较准确的。把实现每个功能所需要的源程序行数累加起来，就可得到实现整个软件所需要的源程序行数。

可由多名有经验的软件工程师分别做出估计。

每个人都估计程序的最小规模(a)、最大规模(b)和最可能的规模(m)，分别算出这 3 种规模的平均值,和之后，再用下式计算程序规模的估计值：

$$L=\frac{\bar{a}+4\bar{m}+\bar{b}}{6}$$

用代码行技术估算软件规模时，当程序较小时常用的单位是代码行数（LOC），当程序较大时常用的单位是千行代码数（KLOC）。

代码行技术的主要优点是：代码是所有软件开发项目都有的“产品”，而且很容易计算代码行数。

代码行技术的缺点是：　源程序仅是软件配置的一个成分，用它的规模代表整个软件的规模似乎不太合理；用不同语言实现同一个软件所需要的代码行数并不相同；这种方法不适用于非过程语言。

为了克服代码行技术的缺点，人们又提出了功能点技术。

13.1.2　功能点技术

功能点技术依据对软件信息域特性和软件复杂性的评估结果，估算软件规模。这种方法用功能点（FP）为单位度量软件规模。

1. 信息域特性

功能点技术定义了信息域的 5 个特性，分别是输入项数(Inp)、输出项数(Out)、查询数(Inq)、主文件数(Maf)和外部接口数(Inf)。下面讲述这 5 个特性的含义。

（1）　输入项数：　用户向软件输入的项数，这些输入给软件提供面向应用的数据。输入不同于查询，后者单独计数，不计入输入项数中。

（2）　输出项数：　软件向用户输出的项数，它们向用户提供面向应用的信息，例如，报表和出错信息等。报表内的数据项不单独计数。

（3）　查询数：　查询即是一次联机输入，它导致软件以联机输出方式产生某种即时响应。

（4）　主文件数：　逻辑主文件（即数据的一个逻辑组合，可能是大型数据库的一部分或是一个独立的文件）的数目。

（5）　外部接口数：　机器可读的全部接口（例如，磁盘或磁带上的数据文件）的数量，用这些接口把信息传送给另一个系统。

2. 估算功能点的步骤

用下述 3 个步骤，可估算出一个软件的功能点数（即软件规模）。

（1）　计算未调整的功能点数 UFP

首先，把产品信息域的每个特性(即 Inp、Out、Inq、Maf 和 Inf)都分类为简单级、平均级或复杂级，并根据其等级为每个特性分配一个功能点数。

然后，用下式计算未调整的功能点数 UFP：

UFP=a1×Inp+a2×Out+a3×Inq+a4×Maf+a5×Inf

其中，ai(1≤i≤5)是信息域特性系数，其值由相应特性的复杂级别决定，如表 13.1（见书 297 页）所示。

(2) 计算技术复杂性因子 TCF

这一步骤度量 14 种技术因素对软件规模的影响程度。这些因素包括高处理率、性能标准、联机更新等，在表 13.2 中列出了全部技术因素，并用 Fi(1≤i≤14)代表这些因素。根据软件的特点，为每个因素分配一个从 0（不存在或对软件规模无影响）到 5（有很大影响）的值。

然后，用下式计算技术因素对软件规模的综合影响程度

$$DI=\sum_{i=1}^{14}F_i$$

DI：

技术复杂性因子 TCF 由下式计算：

TCF=0.65+0.01×DI

因为 DI 的值在 0~70 之间，所以 TCF 的值在 0.65~1.35 之间。

（3）　计算功能点数 FP

用下式计算功能点数 FP：

$$FP=UFP\times TCF$$

功能点数与所用的编程语言无关，看起来功能点技术比代码行技术更合理一些。但是，在判断信息域特性复杂级别和技术因素的影响程度时，存在着相当大的主观因素。

13.2 工作量估算

软件估算模型使用由经验导出的公式来预测软件开发工作量，工作量是软件规模（KLOC 或 FP）的函数，工作量的单位通常是人月（pm）。

支持大多数估算模型的经验数据，都是从有限个项目的样本集中总结出来的，因此，没有一个估算模型可以适用于所有类型的软件和开发环境。

13.2.1 静态单变量模型

这类模型的总体结构形式如下：

$$E=A+B\times(ev)^C$$

其中，A、B 和 C 是由经验数据导出的常数，E 是以人月为单位的工作量，ev 是估算变量（KLOC 或 FP）。下面给出几个典型的静态单变量模型。

1. 面向 KLOC 的估算模型

(1) Walston_Felix 模型　$E=5.2\times(KLOC)^{0.91}$

(2) Bailey_Basili 模型　$E=5.5+0.73\times(KLOC)^{1.16}$

(3) Boehm 简单模型　$E=3.2\times(KLOC)^{1.05}$

(4) Doty 模型（在 KLOC>9 时适用）

$E=5.288\times(KLOC)^{1.047}$

2. 面向 FP 的估算模型

(1) Albrecht & Gaffney 模型　$E=-13.39+0.0545FP$

(2) Maston,Barnett 和 Mellichamp 模型

$$E=585.7+15.12FP$$

从上面列出的模型可以看出，对于相同的 KLOC 或 FP 值，用不同模型估算将得出不同的结果。主要原因是，这些模型多数都是仅根据若干应用领域中有限个项目的经验数据推导出来的，适用范围有限。因此，必须根据当前项目的特点选择适用的估算模型，并且根据需要适当地调整估算模型。

13.2.2 动态多变量模型

动态多变量模型也称为软件方程式，它是根据从 4000 多个当代软件项目中收集的生产率数据推导出来的。该模型把工作量看作是软件规模和开发时间这两个变量的函数。

动态多变量估算模型的形式如下：

$$E=(LOC\times B^{0.333}/P)^3\times(1/t)^4$$

其中，E 是以人月或人年为单位的工作量；

t 是以月或年为单位的项目持续时间；

B 是特殊技术因子，它随着对测试、质量保证、文档及管理技术的需求的增加而缓慢增加，对于较小的程序（KLOC=5~15），B=0.16；对于超过 70 KLOC 的程序，B=0.39

P 是生产率参数，它反映了下述因素对工作量的影响：

总体过程成熟度及管理水平；

使用良好的软件工程实践的程度；

使用的程序设计语言的级别；

软件环境的状态；

软件项目组的技术及经验；

应用系统的复杂程度。

开发实时嵌入式软件时，P 的典型值为 2000；开发电信系统和系统软件时，P=10000；对于商业应用系统来说，P=28000。可以从历史数据导出适用于当前项目的生产率参数值。

从（13.2）式可以看出，开发同一个软件（即 LOC 固定）的时候，如果把项目持续时间延长一些，则可降低完成项目所需的工作量。

13.2.3 COCOMO2 模型

COCOMO 是构造性成本模型（constructive cost model)的英文缩写。

COCOMO2 给出了 3 个层次的软件开发工作量估算模型，这 3 个层次的模型在估算工作量时，对软件细节考虑的详尽程度逐级增加。这些模型既可以用于不同类型的项目，也可以用于同一个项目的不同开发阶段。

这 3 个层次的估算模型分别是：

（1） 应用系统组成模型。这个模型主要用于估算构建原型的工作量，模型名字暗示在构建原型时大量使用已有的构件。

（2） 早期设计模型。这个模型适用于体系结构设计阶段。

（3） 后体系结构模型。这个模型适用于完成体系结构设计之后的软件开发阶段。

下面以体系结构模型为例，介绍 COCOMO2 模型。该模型把软件开发工作量表示成代码行数（KLOC）的非线性函数：

$$E=a\times KLOC^b\times\prod_{i=1}^{17}f_i$$

其中，E 是开发工作量（以人月为单位），

a 是模型系数，

KLOC 是估计的源代码行数（以千行为单位），

b 是模型指数，

fi(i=1~17)是成本因素。

每个成本因素都根据它的重要程度和对工作量影响大小被赋予一定数值（称为工作量系数）。这些成本因素对任何一个项目的开发工作量都有影响，即使不使用 COCOMO2 模型估算工作量，也应该重视这些因素。

Boehm 把成本因素划分成产品因素、平台因素、人员因素和项目因素等 4 类。

表 13.3 列出了 COCOMO2 模型使用的成本因素及与之相联系的工作量系数。

（1） 新增加了 4 个成本因素，它们分别是要求的可重用性、需要的文档量、人员连续性（即人员稳定程度）和

多地点开发。这个变化表明，这些因素对开发成本的影响日益增加。

（2） 略去了原始模型中的 2 个成本因素（计算机切换时间和使用现代程序设计实践）。现在，开发人员普遍使用工作站开发软件，批处理的切换时间已经不再是问题。而“现代程序设计实践”已经发展成内容更广泛的“成熟的软件工程实践”的概念，并且在 COCOMO2 工作量方程的指数 b 中考虑了这个因素的影响。

（3） 某些成本因素（分析员能力、平台经验、语言和工具经验）对生产率的影响（即工作量系数最大值与最小值的比率）增加了，另一些成本因素（程序员能力）的影响减小了。

为了确定工作量方程中模型指数 b 的值，原始的 COCOMO 模型把软件开发项目划分成组织式、半独立式和嵌入式这样 3 种类型，并指定每种项目类型所对应的 b 值（分别是 1.05，1.12 和 1.20）。COCOMO2 采用了更加精细得多的 b 分级模型，这个模型使用 5 个分级因素 Wi(1≤i≤5),其中每个因素都划分成从甚低（Wi=5)到特高（Wi=0)的 6 个级别，然后用下式计算 b 的数值：

$$b = 1.01 + 1.01 \times \sum_{i=1}^{5} W_i \quad (13.4)$$

因此，b 的取值范围为 1.01~1.26。显然，这种分级模式比原始 COCOMO 模型的分级模式更精细、更灵活。

COCOMO2 使用的 5 个分级因素如下所述：

（1） 项目先例性。这个分级因素指出，对于开发组织来说该项目的新奇程度。诸如开发类似系统的经验，需要创新体系结构和算法，以及需要并行开发硬件和软件等因素的影响，都体现在这个分级因素中。

（2） 开发灵活性。这个分级因素反映出，为了实现预先确定的外部接口需求及为了及早开发出产品而需要增加的工作量。

（3） 风险排除度。这个分级因素反映了重大风险已被消除的比例。在多数情况下，这个比例和指定了重要模块接口（即选定了体系结构）的比例密切相关。

（4） 项目组凝聚力。这个分级因素表明了开发人员相互协作时可能存在的困难。这个因素反映了开发人员在目标和文化背景等方面相一致的程度，以及开发人员组成一个小组工作的经验。

（5） 过程成熟度。这个分级因素反映了按照能力成熟度模型度量出的项目组织的过程成熟度。

在原始的 COCOMO 模型中，仅粗略地考虑了前两个分级因素对指数 b 之值的影响。

工作量方程中模型系数 a 的典型值为 3.0，在实际工作中应该根据历史经验数据确定一个适合本组织当前开发的项目类型的数值。

13.3 进度计划

管理者必须制定一个足够详细的进度表，以便监督项目进度并控制整个项目。

一个有效的软件过程应该定义一个适用于当前项目的任务集合。一个任务集合包括一组软件工程工作任务、里程碑和可交付的产品。为一个项目所定义的任务集合，必须包括为获得高质量的软件产品而应该完成的所有任务。

项目管理者的目标是定义全部项目任务，识别出关键任务，跟踪关键任务的进展状况，以保证能及时发现拖延进度的情况。

软件项目的进度安排是这样一种活动，它通过把工作量分配给特定的软件工程任务并规定完成各项任务的起止日期，从而将估算出的项目工作量分布于计划好的项目持续期内。进度计划将随着时间的流逝而不断演化。在项目计划的早期，首先制定一个宏观的进度安排表，标识出主要的软件工程活动和这些活动影响到的产品功能。随着项目的进展，把宏观进度表中的每个条目都精化成一个详细进度表，从而标识出完成一个活动所必须实现的一组特定任务，并安排好了实现这些任务的进度。

13.3.1 估算开发时间

估算出完成给定项目所需的总工作量之后，接下来需要估算的就是项目开发时间。

实际上软件开发时间与从事开发工作的人数之间并不是简单的反比关系。

通常，成本估算模型也同时提供了估算开发时间 T 的方程。与工作量方程不同，各种模型估算开发时间的方程很相似，

(1) Walston_Felix 模型　　$T=2.5E^{0.35}$

(2) 原始的 COCOMO 模型　　$T=2.5E^{0.38}$

(3) COCOMO2 模型　　$T=3.0E^{0.33+0.2\times(b-1.01)}$

(4) Putnam 模型　　$T=2.4E^{1/3}$

其中，E 是开发工作量（以人月为单位），

T 是开发时间（以月为单位）。

用上列方程计算出的 T 值，代表正常情况下的开发时间。客户往往希望缩短软件开发时间，显然，为了缩短开发时间应该增加从事开发工作的人数。但是，经验告诉我们，随着开发小组规模扩大，个人生产率将下降，以致开发时间与从事开发工作的人数并不成反比关系。出现这种现象主要有下述两个原因：

当小组变得更大时，每个人需要用更多时间与组内其他成员讨论问题、协调工作，因此增加了通信开销。

如果在开发过程中增加小组人员，则最初一段时间内项目组总生产率不仅不会提高反而会下降。这是因为新成员在开始时不仅不是生产力，而且在他们学习期间还需要花费小组其他成员的时间。

综合上述两个原因，存在被称为 Brooks 规律的下述现象：　向一个已经延期的项目增加人力，只会使得它更加

延期。

下面让我们研究项目组规模与项目组总生产率的关系。项目组成员之间的通信路径数，由项目组人数和项目组结构决定。如果项目组共有 P 名组员，每个组员必须与所有其他组员通信以协调开发活动，则通信路径数为 P(P-1)/2。如果每个组员只需与另外一个组员通信，则通信路径数为 P-1。

因此，通信路径数大约在 P~P2/2 的范围内变化。也就是说，在一个层次结构的项目组中，通信路径数为 Pα，其中 1<α<2。

对于某一个组员来说，他与其他组员通信的路径数在 1~（P-1）的范围内变化。如果不与任何人通信时个人生产率为 L，而且每条通信路径导致生产率减少 l，则组员个人平均生产率为

$$Lr=L-l(P-1)^r$$

其中，r 是对通信路径数的度量，0<r≤1(假设至少有一名组员需要与一个以上的其他组员通信，因此 r>0)。

对于一个规模为 P 的项目组，从（13.5）式导出项目组的总生产率为

$$Ltot=P(L-l(P-1)^r)$$

对于给定的一组 L，l 和 r 的值，总生产率 Ltot 是项目组规模 P 的函数。随着 P 值增加，Ltot 将从 0 增大到某个最大值，然后再下降。因此，存在一个最佳的项目组规模 Popt，这个规模的项目组其总生产率最高。

让我们举例说明项目组规模与生产率的关系。假设个人最高生产率为 500LOC/月（即 L=500），每条通信路径导致生产率下降 10%（即 l=50)。如果每个组员都必须与组内所有其他组员通信（r=1），则项目组规模与生产率的关系列在表 13.4（见书 304 页）中，可见，在这种情况下项目组的最佳规模是 5.5 人，即 Popt=5.5。

Boehm 根据经验指出，软件项目的开发时间最多可以减少到正常开发时间的 75%。

13.3.2 Gantt 图

Gantt（甘特）图是历史悠久、应用广泛的制定进度计划的工具，下面通过一个非常简单的例子介绍这种工具。

假设有一座陈旧的矩形木板房需要重新油漆。这项工作必须分 3 步完成： 首先刮掉旧漆，然后刷上新漆，最后清除溅在窗户上的油漆。假设一共分配了 15 名工人去完成这项工作，然而工具却很有限： 只有 5 把刮旧漆用的刮板，5 把刷漆用的刷子，5 把清除溅在窗户上的油漆用的小刮刀。怎样安排才能使工作进行得更有效呢?

一种做法是首先刮掉四面墙壁上的旧漆，然后给每面墙壁都刷上新漆，最后清除溅在每个窗户上的油漆。显然这是效率最低的做法，因为总共有 15 名工人，然而每种工具却只有 5 件，这样安排工作在任何时候都有 10 名工人闲着没活干。

应该采用“流水作业法”，也就是说，首先由 5 名工人用刮板刮掉第 1 面墙上的旧漆(这时其余 10 名工人休息)，当第 1 面墙刮净后，另外 5 名工人立即用刷子给这面墙刷新漆(与此同时拿刮板的 5 名工人转去刮第 2 面墙上的旧漆)，一旦刮旧漆的工人转到第 3 面墙而且刷新漆的工人转到第 2 面墙以后，余下的 5 名工人立即拿起刮刀去清除溅在第 1 面墙窗户上的油漆，……。这样安排每个工人都有活干，因此能够在较短的时间内完成任务。

假设木板房的第 2、4 两面墙的长度比第 1、3 两面墙的长度长一倍，此外，不同工作需要用的时间长短也不同，刷新漆最费时间，其次是刮旧漆，清理(即清除溅在窗户上的油漆)需要的时间最少。表 13.5 列出了估计每道工序需要用的时间。可以使用图 13.1 中的 Gantt 图描绘上述流水作业过程： 在时间为零时开始刮第 1 面墙上的旧漆，两小时后刮旧漆的工人转去刮第 2 面墙，同时另 5 名工人开始给第 1 面墙刷新漆，每当给一面墙刷完新漆之后，第 3 组的 5 名工人立即清除溅在这面墙窗户上的漆。从图 13.1 可以看出 12 小时后刮完所有旧漆，20 小时后完成所有墙壁的刷漆工作，再过 2 小时后清理工作结束。因此全部工程在 22 小时后结束，如果用前述的第一种做法，则需要 36 小时。

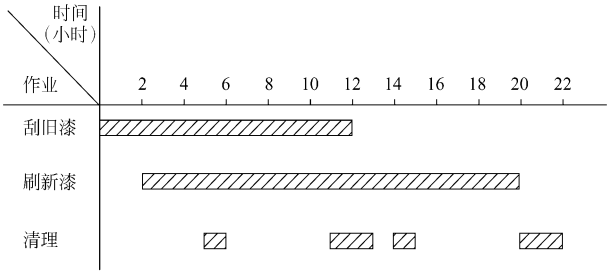


图 13.1 旧木板房刷漆工程的 Gantt 图

13.3.3 工程网络

上一小节介绍的 Gantt 图能很形象地描绘任务分解情况，以及每个子任务(作业)的开始时间和结束时间，因此是进度计划和进度管理的有力工具。它具有直观简明和容易掌握、容易绘制的优点，但是 Gantt 图也有 3 个主要缺点：

(1) 不能显式地描绘各项作业彼此间的依赖关系；

(2) 进度计划的关键部分不明确，难于判定哪些部分应当是主攻和主控的对象；

(3) 计划中有潜力的部分及潜力的大小不明确，往往造成潜力的浪费。

当把一个工程项目分解成许多子任务，并且它们彼此间的依赖关系又比较复杂时，仅仅用 Gantt 图作为安排进度的工具是不够的，不仅难于做出既节省资源又保证进度的计划，而且还容易发生差错。

工程网络是制定进度计划时另一种常用的图形工具，它同样能描绘任务分解情况以及每项作业的开始时间和结束时间，此外，它还显式地描绘各个作业彼此间的依赖关系。因此，工程网络是系统分析和系统设计的强有力的工具。

在工程网络中用箭头表示作业(例如，刮旧漆，刷新漆，清理等)，用圆圈表示事件(一项作业开始或结束)。注意，事件仅仅是可以明确定义的时间点，它并不消耗时间和资源。作业通常既消耗资源又需要持续一定时间。图 13.2 是旧木板房刷漆工程的工程网络。图中表示刮第 1 面墙上旧漆的作业开始于事件 1，结束于事件 2。用开始事件和结束事件的编号标识一个作业，因此“刮第 1 面墙上旧漆”是作业 1—2。

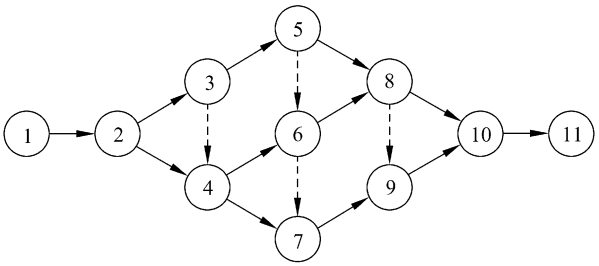


图 13.2 旧木板房刷漆工程的工程网络

在工程网络中的一个事件，如果既有箭头进入又有箭头离开，则它既是某些作业的结束又是另一些作业的开始。例如，图 13.2 中事件 2 既是作业 1—2(刮第 1 面墙上的旧漆)的结束，又是作业 2—3(刮第 2 面墙上旧漆)和作业 2—4(给第 1 面墙刷新漆)的开始。也就是说，只有第 1 面墙上的旧漆刮完之后，才能开始刮第 2 面墙上旧漆和给第 1 面墙刷新漆这两个作业。因此，工程网络显式地表示了作业之间的依赖关系。

在图 13.2 中还有一些虚线箭头，它们表示虚拟作业，也就是事实上并不存在的作业。引入虚拟作业是为了显式地表示作业之间的依赖关系。例如，事件 4 既是给第 1 面墙刷新漆结束，又是给第 2 面墙刷新漆开始(作业 4—6)。但是，在开始给第 2 面墙刷新漆之前，不仅必须已经给第 1 面墙刷完了新漆，而且第 2 面墙上的旧漆也必须已经刮净(事件 3)。也就是说，在事件 3 和事件 4 之间有依赖关系，或者说在作业 2—3(刮第 2 面墙上旧漆)和作业 4—6(给第 2 面墙刷新漆)之间有依赖关系，虚拟作业 3—4 明确地表示了这种依赖关系。注意，虚拟作业既不消耗资源也不需要时间。

13.3.4 估算工程进度

画出类似图 13.2 那样的工程网络之后，系统分析员就可以借助它的帮助估算工程进度了。为此需要在工程网络上增加一些必要的信息。

首先，把每个作业估计需要使用的时间写在表示该项作业的箭头上方。注意，箭头长度和它代表的作业持续时间没有关系，箭头仅表示依赖关系，它上方的数字才表示作业的持续时间。

其次，为每个事件计算下述两个统计数字： 最早时刻 EET 和最迟时刻 LET。这两个数字将分别写在表示事件的圆圈的右上角和右下角，如图 13.3 左下角的符号所示。

事件的最早时刻是该事件可以发生的最早时间。通常工程网络中第一个事件的最早时刻定义为零，其他事件的最早时刻在工程网络上从左至右按事件发生顺序计算。计算最早时刻 EET 使用下述 3 条简单规则：

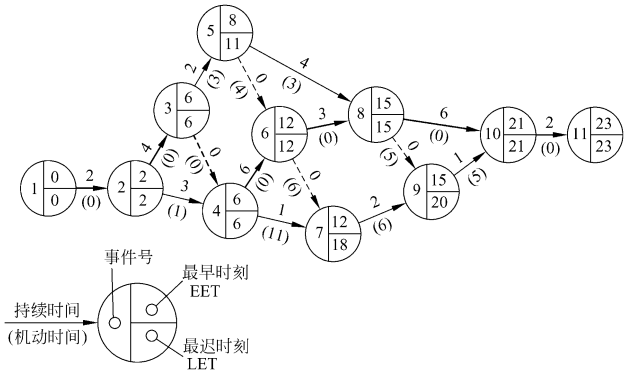


图 13.3 旧木板房刷漆工程的完整的工程网络

(1) 考虑进入该事件的所有作业；

(2) 对每个作业计算它的持续时间与起始事件的 EET 之和；

(3) 选取上述和数中的最大值作为该事件的最早时刻 EET。

按照这种方法，不难沿着工程网络从左至右顺序算出每个事件的最早时刻，计算结果标在图 13.3 的工程网络中(每个圆圈内右上角的数字)。

事件的最迟时刻是在不影响工程竣工时间的前提下，该事件最晚可以发生的时刻。按惯例，最后一个事件(工程结束)的最迟时刻就是它的最早时刻。其他事件的最迟时刻在工程网络上从右至左按逆作业流的方向计算。计算最迟时刻 LET 使用下述 3 条规则：

(1) 考虑离开该事件的所有作业；

(2) 从每个作业的结束事件的最迟时刻中减去该作业的持续时间；

(3) 选取上述差数中的最小值作为该事件的最迟时刻 LET。

图 13.3 中每个圆圈内右下角的数字就是该事件的最迟时刻。

13.3.5 关键路径

图 13.3 中有几个事件的最早时刻和最迟时刻相同，这些事件定义了关键路径，在图中关键路径用粗线箭头表示。关键路径上的事件(关键事件)必须准时发生，组成关键路径的作业(关键作业)的实际持续时间不能超过估计的持续时间，否则工程就不能准时结束。

工程项目的管理人员应该密切注视关键作业的进展情况，如果关键事件出现的时间比预计的时间晚，则会使最终完成项目的时间拖后；如果希望缩短工期，只有往关键作业中增加资源才会有效果。

13.3.6 机动时间

不在关键路径上的作业有一定程度的机动余地——实际开始时间可以比预定时间晚一些，或者实际持续时间可以

比预定的持续时间长一些，而并不影响工程的结束时间。一个作业可以有的全部机动时间等于它的结束事件的最迟时刻减去它的开始事件的最早时刻，再减去这个作业的持续时间：

机动时间=(LET)结束-(EET)开始-持续时间

对于前述油漆旧木板房的例子，计算得到的非关键作业的机动时间列在表 13.6 中。

在工程网络中每个作业的机动时间写在代表该项作业的箭头下面的括弧里(参看图 13.3)。

在制定进度计划时仔细考虑和利用工程网络中的机动时间，往往能够安排出既节省资源又不影响最终竣工时间的进度表。在图 13.4 中的 Gantt 图描绘了其中的一种方案。

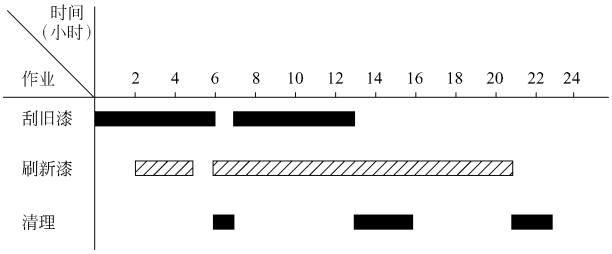


图 13.4 旧木板房刷漆工程改进的 Gantt 图之一

这个简单例子明显说明了工程网络比 Gantt 图优越的地方： 它显式地定义事件及作业之间的依赖关系，Gantt 图只能隐含地表示这种关系。但是 Gantt 图的形式比工程网络更简单更直观，为更多的人所熟悉，因此，应该同时使用这两种工具制订和管理进度计划，使它们互相补充取长补短。

以上通过旧木板房刷新漆工程的简单例子，介绍了制订进度计划的两个重要工具和方法。软件工程项目虽然比这个简单例子复杂得多，但是计划和管理的基本方法仍然是自顶向下分解，也就是把项目分解为若干个阶段，每个阶段再分解成许多更小的任务，每个任务又可进一步分解为若干个步骤等等。这些阶段、任务和步骤之间有复杂的依赖关系，因此，工程网络和 Gantt 图同样是安排进度和管理工程进展情况的强有力的工具。

第 13.2 节中介绍的工作量估计技术可以帮助我们估计每项任务的工作量，根据人力分配情况，可以进一步确定每项任务的持续时间。从这些基本数据出发，根据作业之间的依赖关系，利用工程网络和 Gantt 图可以制定出合理的进度计划，并且能够科学地管理软件开发工程的进展情况。

13.4 人员组织

软件项目成功的关键是有高素质的软件开发人员。必须把多名软件开发人员合理地组织起来，使他们有效地分工协作共同完成开发工作。

为了成功地完成软件开发工作，项目组成员必须以一种有意义且有效的方式彼此交互和通信。如何组织项目组是一个重要的管理问题，管理者应该合理地组织项目组，使项目组有较高生产率，能够按预定的进度计划完成所承担的工作。经验表明，项目组组织得越好，其生产率越高，而且产品质量也越好。

除了追求更好的组织方式之外，每个管理者的目标都是建立有凝聚力的项目组。一个有高度凝聚力的小组，由一批团结得非常紧密的人组成，他们的整体力量大于个体力量的总和。一旦项目组具有了凝聚力，成功的可能性就大大增加了。

现有的软件项目组的组织方式很多，通常，组织软件开发人员的方法，取决于所承担的项目的特点、以往的组织经验以及管理者的看法和喜好。下面介绍 3 种典型的组织方式。

13.4.1 民主制程序员组

民主制程序员组的一个重要特点是，小组成员完全平等，享有充分民主，通过协商做出技术决策。因此，小组成员之间的通信是平行的，如果小组内有 n 个成员，则可能的通信信道共有 n(n-1)/2 条。

程序设计小组的人数不能太多,否则组员间彼此通信的时间将多于程序设计时间。此外,通常不能把一个软件系统划分成大量独立的单元,因此,如果程序设计小组人数太多,则每个组员所负责开发的程序单元与系统其他部分的界面将是复杂的,不仅出现接口错误的可能性增加,而且软件测试将既困难又费时间。

一般说来，程序设计小组的规模应该比较小,以 2～8 名成员为宜。如果项目规模很大,用一个小组不能在预定时间内完成开发任务,则应该使用多个程序设计小组,每个小组承担工程项目的一部分任务，在一定程度上独立自主地完成各自的任务。系统的总体设计应该能够保证由各个小组负责开发的各部分之间的接口是良好定义的,并且是尽可能简单的。

小组规模小,不仅可以减少通信问题,而且还有其他好处。例如,容易确定小组的质量标准,而且用民主方式确定的标准更容易被大家遵守;组员间关系密切,能够互相学习等等。

民主制程序员组通常采用非正式的组织方式，也就是说，虽然名义上有一个组长，但是他和组内其他成员完成同样的任务。在这样的小组中，由全体讨论协商决定应该完成的工作，并且根据每个人的能力和经验分配适当的任务。

民主制程序员组的主要优点是，组员们对发现程序错误持积极的态度，这种态度有助于更快速地发现错误，从而导致高质量的代码。

民主制程序员组的另一个优点是，组员们享有充分民主，小组有高度凝聚力，组内学术空气浓厚，有利于攻克技术难关。因此，当有难题需要解决时，也就是说，当所要开发的软件的技术难度较高时，采用民主制程序员组是适宜的。

如果组内多数成员是经验丰富技术熟练的程序员,那么上述非正式的组织方式可能会非常成功。在这样的小组内组员享有充分民主,通过协商,在自愿的基础上作出决定,因此能够增强团结、提高工作效率。但是,如果组内多数成员技术水平不高,或是缺乏经验的新手,那么这种非正式的组织方式也有严重缺点： 由于没有明确的权威指导开发工程的进行,组员间将缺乏必要的协调,最终可能导致工程失败。

为了使少数经验丰富、技术高超的程序员在软件开发过程中能够发挥更大作用，程序设计小组也可以采用下一小节中介绍的另外一种组织形式。

13.4.2 主程序员组

美国 IBM 公司在 20 世纪 70 年代初期开始采用主程序员组的组织方式。采用这种组织方式主要出于下述几点考虑：

(1) 软件开发人员多数比较缺乏经验；

(2) 程序设计过程中有许多事务性的工作，例如，大量信息的存储和更新；

(3) 多渠道通信很费时间，将降低程序员的生产率。

主程序员组用经验多、技术好、能力强的程序员作为主程序员，同时，利用人和计算机在事务性工作方面给主程序员提供充分支持，而且所有通信都通过一两个人进行。这种组织方式类似于外科手术小组的组织。

主程序员组核心人员的分工如下所述：

（1） 主程序员既是成功的管理人员又是经验丰富、技术好、能力强的高级程序员，负责体系结构设计和关键部分（或复杂部分）的详细设计，并且负责指导其他程序员完成详细设计和编码工作。如图 13.5 所示，程序员之间没有通信渠道，所有接口问题都由主程序员处理。主程序员对每行代码的质量负责，因此，他还要对组内其他成员的工作成果进行复查。

（2） 后备程序员也应该技术熟练而且富于经验，他协助主程序员工作并且在必要时（例如，主程序员生病、出差或“跳槽”）接替主程序员的工作。因此，后备程序员必须在各方面都和主程序员一样优秀，并且对本项目的了解也应该和主程序员一样深入。平时，后备程序员的工作主要是，设计测试方案、分析测试结果及独立于设计过程的其他工作。

（3） 编程秘书负责完成与项目有关的全部事务性工作，例如，维护项目资料库和项目文档，编译、链接、执行源程序和测试用例。

注意，上面介绍的是 20 世纪 70 年代初期的主程序员组组织结构，现在的情况已经和当时大不相同了，程序员已经有了自己的终端或工作站，他们自己完成代码的输入、编辑、编译、链接和测试等工作，无须由编程秘书统一做这些工作。典型的主程序员组的现代形式将在下一小节介绍。

虽然图 13.5 所示的主程序员组的组织方式说起来有不少优点，但是，它在许多方面却是不切实际的。

首先，如前所述，主程序员应该是高级程序员和优秀管理者的结合体。承担主程序员工作需要同时具备这两方面的才能，但是，在现实社会中这样的人才并不多见。通常，既缺乏成功的管理者也缺乏技术熟练的程序员。

其次，后备程序员更难找。人们期望后备程序员像主程序员一样优秀，但是，他们必须坐在“替补席”上，拿着较低的工资等待随时接替主程序员的工作。几乎没有一个高级程序员或高级管理人员愿意接受这样的工作。

第三，编程秘书也很难找到。专业的软件技术人员一般都厌烦日常的事务性工作，但是，人们却期望编程秘书整天只干这类工作。

我们需要一种更合理、更现实的组织程序员组的方法，这种方法应该能充分结合民主制程序员组和主程序员组的优点，并且能用于实现更大规模的软件产品。

13.4.3 现代程序员组

民主制程序员组的一个主要优点，是小组成员都对发现程序错误持积极、主动的态度。但是，使用主程序员组的组织方式时，主程序员对每行代码的质量负责，因此，他必须参与所有代码审查工作。由于主程序员同时又是负责对小组成员进行评价的管理员，他参与代码审查工作就会把所发现的程序错误与小组成员的工作业绩联系起来，从而造成小组成员出现不愿意发现错误的心理。

解决上述问题的方法是，取消主程序员的大部分行政管理工作。前面已经指出，很难找到既是高度熟练的程序员又是成功的管理员的人，取消主程序员的行政管理工作，不仅解决了小组成员不愿意发现程序错误的心理问题，也使得寻找主程序员的人选不再那么困难。于是，实际的“主程序员”应该由两个人共同担任： 一个技术负责人，负责小组的技术活动；一个行政负责人，负责所有非技术性事务的管理决策。这样的组织结构如图 13.6 所示。技术组长自然要参与全部代码审查工作，因为他要对代码的各方面质量负责；相反，行政组长不可以参与代码审查工作，因为他的职责是对程序员的业绩进行评价。行政组长应该在常规调度会议上了解每名组员的技术能力和工作业绩。

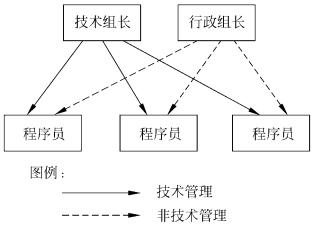


图 13.6 现代程序员组的结构

在开始工作之前明确划分技术组长和行政组长的管理权限是很重要的。但是，即使已经做了明确分工，有时也会出现职责不清的矛盾。例如，考虑年度休假问题，行政组长

有权批准某个程序员休年假的申请，因为这是一个非技术性问题，但是技术组长可能马上否决了这个申请，因为已经接近预定的项目结束日期，目前人手非常紧张。解决这类问题的办法是求助于更高层的管理人员，对行政组长和技术组长都认为是属于自己职责范围内的事务，制定一个处理方案。

由于程序员组成员人数不宜过多，当软件项目规模较大时，应该把程序员分成若干个小组，采用图 13.7 所示的组织结构。该图描绘的是技术管理组织结构，非技术管理组织结构与此类似。由图可以看出，产品开发作为一个整体是在项目经理的指导下进行的，程序员向他们的组长汇报工作，而组长则向项目经理汇报工作。当产品规模更大时，可以适当增加中间管理层次。

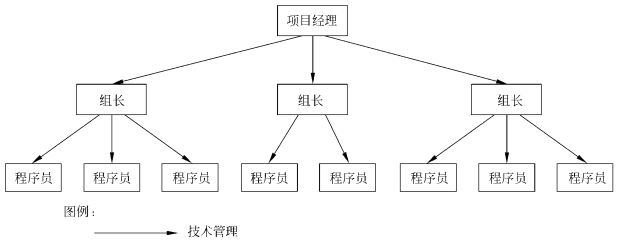


图 13.7 大型项目的技术管理组织结构

把民主制程序员组和主程序员组的优点结合起来的另一种方法，是在合适的地方采用分散做决定的方法，如图 13.8 所示。这样做有利于形成畅通的通信渠道，以便充分发挥每个程序员的积极性和主动性，集思广益攻克技术难关。这种组织方式对于适合采用民主方法的那类问题（例如，研究性项目或遇到技术难题需要用集体智慧攻关）非常有效。尽管这种组织方式适当地发扬了民主，但是上下级之间的箭头（即管理关系）仍然是向下的，也就是说，是在集中指导下发扬民主。显然，如果程序员可以指挥项目经理，则只会引起混乱。

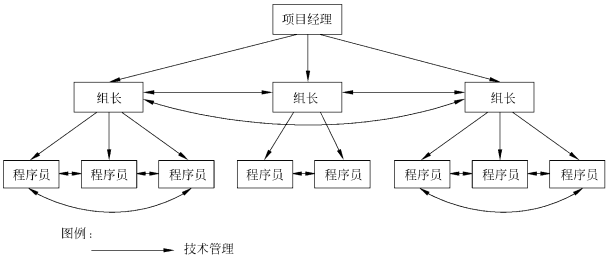


图 13.8 包含分散决策的组织方式

13.5 质量保证

13.5.1 软件质量

概括地说，软件质量就是“软件与明确地和隐含地定义的需求相一致的程度”。更具体地说，软件质量是软件与明确地叙述的功能和性能需求、文档中明确描述的开发标准以及任何专业开发的软件产品都应该具有的隐含特征相一致的程度。上述定义强调了下述的 3 个要点：

（1） 软件需求是度量软件质量的基础，与需求不一致就是质量不高。

（2） 指定的开发标准定义了一组指导软件开发的准则，如果没有遵守这些准则，几乎肯定会导致软件质量不高。

（3） 通常，有一组没有显式描述的隐含需求（例如，软件应该是容易维护的）。如果软件满足明确描述的需求，但却不满足隐含的需求，那么软件的质量仍然是值得怀疑的。

虽然软件质量是难于定量度量的软件属性，但是仍然能够提出许多重要的软件质量指标(其中绝大多数目前还处于定性度量阶段)。

影响软件质量的主要因素是从管理角度对软件质量的度量。可以把这些质量因素分成 3 组，分别反映用户在使用软件产品时的 3 种不同倾向或观点。这 3 种倾向是： 产品运行、产品修改和产品转移。图 13.9 描绘了软件质量因素和上述 3 种倾向(或产品活动)之间的关系，表 13.7（见书 315 页）列出了软件质量因素的简明定义。

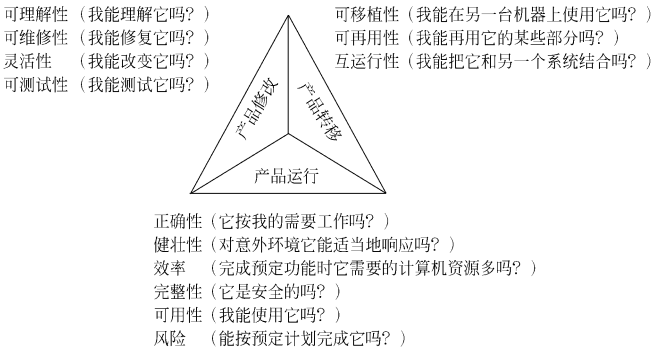


图 13.9 软件质量因素与产品活动的关系

13.5.2 软件质量保证措施

软件质量保证（software quality assurance,SQA）的措施主要有： 基于非执行的测试（也称为复审或评审），基于执行的测试（即以前讲过的软件测试）和程序正确性证明。复审主要用来保证在编码之前各阶段产生的文档的质量；基于执行的测试需要在程序编写出来之后进行，它是保证软件质量的最后一道防线；程序正确性证明使用数学方法严格验证程序是否与对它的说明完全一致。

参加软件质量保证工作的人员，可以划分成下述两类：软件工程师通过采用先进的技术方法和度量，进行正式的技术复审以及完成计划周密的软件测试来保证软件质量。SQA 小组的职责，是辅助软件工程师以获得高质量的软件产品。其从事的软件质量保证活动主要是： 计划，监督，记录，分析和报告。简而言之，SQA 小组的作用是，通过确保软件过程的质量来保证软件产品的质量。

1. 技术复审的必要性

正式技术复审的显著优点是，能够较早发现软件错误，从而可防止错误被传播到软件过程的后续阶段。

统计数字表明，在大型软件产品中检测出的错误，60%~70%属于规格说明错误或设计错误，而正式技术复审在发现规格说明错误和设计错误方面的有效性高达 75%。由于能够检测出并排除掉绝大部分这类错误，复审

可大大降低后续开发和维护阶段的成本。

实际上，正式技术复审是软件质量保证措施的一种，包括走查（walkthrough)和审查（inspection）等具体方法。走查的步骤比审查少，而且没有审查正规。

2. 走查

走查组由 4~6 名成员组成。以走查规格说明的小组为例，成员至少包括一名负责起草规格说明的人，一名负责该规格说明的管理员，一位客户代表，以及下阶段开发组（在本例中是设计组）的一名代表和 SQA 小组的一名代表。其中 SQA 小组的代表应该作为走查组的组长。

为了能发现重大错误，走查组成员最好是经验丰富的高级技术人员。必须把被走查的材料预先分发给走查组每位成员。走查组成员应该仔细研究材料并列出两张表： 一张表是他不理解的术语，另一张是他认为不正确的术语。

走查组组长引导该组成员走查文档，力求发现尽可能多的错误。走查组的任务仅仅是标记出错误而不是改正错误，改正错误的工作应该由该文档的编写组完成。走查的时间最长不要超过 2 小时，这段时间应该用来发现和标记错误，而不是改正错误。

走查主要有下述两种方式：

（1） 参与者驱动法。参与者按照事先准备好的列表，提出他们不理解的术语和认为不正确的术语。文档编写组的代表必须回答每个质疑，要么承认确实有错误，要么对质疑做出解释。

（2） 文档驱动法。文档编写者向走查组成员仔细解释文档。走查组成员在此过程中不时针对事先准备好的问题或解释过程中发现的问题提出质疑。这种方法可能比第一种方法更有效，往往能检测出更多错误。经验表明，使用文档驱动法时许多错误是由文档讲解者自己发现的。

3. 审查

审查的范围比走查广泛得多，它的步骤也比较多。通常，审查过程包括下述 5 个基本步骤：

（1） 综述。由负责编写文档的一名成员向审查组综述该文档。在综述会结束时把文档分发给每位与会者。

（2） 准备。评审员仔细阅读文档。最好列出在审查中发现的错误的类型，并按发生频率把错误类型分级，以辅助审查工作。这些列表有助于评审员们把注意力集中到最常发生错误的区域。

（3） 审查。评审组仔细走查整个文档。和走查一样，这一步的目的也是发现文档中的错误，而不是改正它们。通常每次审查会不超过 90 分钟。审查组组长应该在一天之内写出一份关于审查的报告。

（4） 返工。文档的作者负责解决在审查报告中列出的所有错误及问题。

（5） 跟踪。组长必须确保所提出的每个问题都得到了圆满的解决（要么修正了文档，要么澄清了被误认为是错误的条目)。必须仔细检查对文档所做的每个修正，以确保没有引入新的错误。如果在审查过程中返工量超过 5%，则应该由审查组再对文档全面地审查一遍。

通常，审查组由 4 人组成。组长既是审查组的管理人员又是技术负责人。审查组必须包括负责当前阶段开发工作的项目组代表和负责下一阶段开发工作的项目组代表，此外，还应该包括一名 SQA 小组的代表。

审查过程不仅步数比走查多，而且每个步骤都是正规的。

审查的正规性体现在： 仔细划分错误类型，并把这些信息运用在后续阶段的文档审查中以及未来产品的审查中。

审查是检测软件错误的一种好方法，利用审查可以在软件过程的早期阶段发现并改正错误，也就是说，能在修正错误的代价变得很昂贵之前就发现并改正错误。因此，审查是一种经济有效的错误检测方法。

4. 程序正确性证明

测试可以暴露程序中的错误，因此是保证软件可靠性的重要手段；但是，测试只能证明程序中有错误，并不能证明程序中没有错误。因此，对于保证软件可靠性来说，测试是一种不完善的技术，人们自然希望研究出完善的正确性证明技术。一旦研究出实用的正确性证明程序(即，能自动证明其他程序的正确性的程序)，软件可靠性将更有保证，测试工作量将大大减少。但是，即使有了正确性证明程序，软件测试也仍然是需要的，因为程序正确性证明只证明程序功能是正确的，并不能证明程序的动态特性是符合要求的，此外，正确性证明过程本身也可能发生错误。

正确性证明的基本思想是证明程序能完成预定的功能。因此，应该提供对程序功能的严格数学说明，然后根据程序代码证明程序确实能实现它的功能说明。

在 20 世纪 60 年代初期，人们已经开始研究程序正确性证明的技术，提出了许多不同的技术方法。虽然这些技术方法本身很复杂，但是它们的基本原理却是相当简单的。

如果在程序的若干个点上，设计者可以提出关于程序变量及它们的关系的断言，那么在每一点上的断言都应该永远是真的。假设在程序的 P1,P2,…，Pn 等点上的断言分别是 a(1),a(2)，…，a(n)，其中 a(1)必须是关于程序输入的断言，a(n)必须是关于程序输出的断言。

为了证明在点 Pi 和 Pi+1 之间的程序语句是正确的，必须证明执行这些语句之后将使断言 a(i)变成 a(i+1)。如果对程序内所有相邻点都能完成上述证明过程，则证明了输入断言加上程序可以导出输出断言。如果输入断言和输出断言是正确的，而且程序确实是可以终止的(不包含死循环)，则上述过程就证明了程序的正确性。

人工证明程序正确性，对于评价小程序可能有些价值，但是在证明大型软件的正确性时，不仅工作量太大，更主要的是在证明的过程中很容易包含错误，因此是不实用的。为了实用的目的，必须研究能证明程序正确性的自动系统。

目前已经研究出证明 PASCAL 和 LISP 程序正确性的程序

系统，正在对这些系统进行评价和改进。现在这些系统还只能对较小的程序进行评价，毫无疑问还需要做许多工作，这样的系统才能实际用于大型程序的正确性证明。

13.6 软件配置管理

任何软件开发都是迭代过程，也就是说，在设计过程会发现需求说明书中的问题，在实现过程又会暴露出设计中的错误，……。此外，随着时间推移客户的需求也会或多或少发生变化。因此，在开发软件的过程中，变化（或称为变动）既是必要的，又是不可避免的。但是，变化也很容易失去控制，如果不能适当地控制和管理变化，势必造成混乱并产生许多严重的错误。

软件配置管理是在软件的整个生命期内管理变化的一组活动。具体地说，这组活动用来： ①标识变化； ②控制变化； ③确保适当地实现了变化； ④向需要知道这类信息的人报告变化。

软件配置管理不同于软件维护。维护是在软件交付给用户使用后才发生的，而配置管理是在软件项目启动时就开始，并且一直持续到软件退役后才终止的一组跟踪和控制活动。

软件配置管理的目标是，使变化更正确且更容易被适应，在必须变化时减少所需花费的工作量。

13.6.1 软件配置

1. 软件配置项

软件过程的输出信息可以分为 3 类： ①计算机程序（源代码和可执行程序）； ②描述计算机程序的文档（供技术人员或用户使用）； ③数据（程序内包含的或在程序外的）。 上述这些项组成了在软件过程中产生的全部信息，我们把它们统称为软件配置，而这些项就是软件配置项。随着软件开发过程的进展，软件配置项的数量迅速增加。不幸的是，由于前述的种种原因，软件配置项的内容随时都可能发生变化。为了开发出高质量的软件产品，软件开发人员不仅要努力保证每个软件配置项正确，而且必须保证一个软件的所有配置项是完全一致的。

可以把软件配置管理看作是应用于整个软件过程的软件质量保证活动，是专门用于管理变化的软件质量保证活动。

2. 基线

基线是一个软件配置管理概念，它有助于我们在不严重妨碍合理变化的前提下来控制变化。IEEE 把基线定义为：已经通过了正式复审的规格说明或中间产品，它可以作为进一步开发的基础，并且只有通过正式的变化控制过程才能改变它。简而言之，基线就是通过了正式复审的软件配置项。在软件配置项变成基线之前，可以迅速而非正式地修改它。一旦建立了基线之后，虽然仍然可以实现变化，但是，必须应用特定的、正式的过程（称为规程）来评估、实现和验证每个变化。

除了软件配置项之外，许多软件工程组织也把软件工具置于配置管理之下，也就是说，把特定版本的编辑器、编译器和其他 CASE 工具，作为软件配置的一部分“固定”下来。因为当修改软件配置项时必然要用到这些工具，为防止不同版本的工具产生的结果不同，应该把软件工具也基线化，并且列入到综合的配置管理过程之中。

13.6.2 软件配置管理过程

软件配置管理是软件质量保证的重要一环，它的主要任务是控制变化，同时也负责各个软件配置项和软件各种版本的标识、软件配置审计以及对软件配置发生的任何变化的报告。

具体来说，软件配置管理主要有 5 项任务： 标识、版本控制、变化控制、配置审计和报告。

1. 标识软件配置中的对象

为了控制和管理软件配置项，必须单独命名每个配置项，然后用面向对象方法组织它们。可以标识出两类对象：基本对象和聚集对象（可以把聚集对象作为代表软件配置完整版本的一种机制）。基本对象是软件工程师在分析、设计、编码或测试过程中创建出来的“文本单元”，例如，需求规格说明的一个段落、一个模块的源程序清单或一组测试用例。聚集对象是基本对象和其他聚集对象的集合。

每个对象都有一组能惟一地标识它的特征： 名字、描述、资源表和“实现”。其中，对象名是无二义性地标识该对象的一个字符串。

在设计标识软件对象的模式时，必须认识到对象在整个生命周期中一直都在演化，因此，所设计的标识模式必须能无歧义地标识每个对象的不同版本。

2. 版本控制

版本控制联合使用规程和工具，以管理在软件工程过程中所创建的配置对象的不同版本。借助于版本控制技术，用户能够通过选择适当的版本来指定软件系统的配置。实现这个目标的方法是，把属性和软件的每个版本关联起来，然后通过描述一组所期望的属性来指定和构造所需要的配置。

上面提到的“属性”，既可以简单到仅是赋给每个配置对象的具体版本号，也可以复杂到是一个布尔变量串，其指明了施加到系统上的功能变化的具体类型。

3. 变化控制

对于大型软件开发项目来说，无控制的变化将迅速导致混乱。变化控制把人的规程和自动工具结合起来，以提供一个控制变化的机制。典型的变化控制过程如下： 接到变化请求之后，首先评估该变化在技术方面的得失、可能产生的副作用、对其他配置对象和系统功能的整体影响以及估算出的修改成本。评估的结果形成“变化报告”，该报告供“变化控制审批者”审阅。所谓变化控制审批者既可以是一个人也可以由一组人组成，其对变化的状态和优先级做最终决策。

为每个被批准的变化都生成一个“工程变化命令”，其描

述将要实现的变化，必须遵守的约束以及复审和审计的标准。把要修改的对象从项目数据库中“提取（check out）”出来，进行修改并应用适当的 SQA 活动。最后，把修改后的对象“提交（check in）”进数据库，并用适当的版本控制机制创建该软件的下一个版本。

“提交”和“提取”过程实现了变化控制的两个主要功能——访问控制和同步控制。访问控制决定哪个软件工程师有权访问和修改一个特定的配置对象，同步控制有助于保证由两名不同的软件工程师完成的并行修改不会相互覆盖。

在一个软件配置项变成基线之前，仅需应用非正式的变化控制。该配置对象的开发者可以对它进行任何合理的修改（只要修改不会影响到开发者工作范围之外的系统需求）。一旦该对象经过了正式技术复审并获得批准，就创建了一个基线。而一旦一个软件配置项变成了基线，就开始实施项目级的变化控制。现在，为了进行修改开发者必须获得项目管理者的批准（如果变化是“局部的”），如果变化影响到其他软件配置项，还必须得到变化控制审批者的批准。在某些情况下，可以省略正式的变化请求、变化报告和工程变化命令，但是，必须评估每个变化并且跟踪和复审所有变化。

4. 配置审计

为了确保适当地实现了所需要的变化，通常从下述两方面采取措施： ①正式的技术复审； ②软件配置审计。

正式的技术复审（见 13.5.2 节）关注被修改后的配置对象的技术正确性。复审者审查该对象以确定它与其他软件配置项的一致性，并检查是否有遗漏或副作用。

软件配置审计通过评估配置对象的那些通常不在复审过程中考虑的特征（例如，修改时是否遵循了软件工程标准，是否在该配置项中显著地标明了所做的修改，是否注明了修改日期和修改者，是否适当地更新了所有相关的软件配置项，是否遵循了标注变化、记录变化和报告变化的规程），而成为对正式技术复审的补充。

5. 状态报告

书写配置状态报告是软件配置管理的一项任务，它回答下述问题： ①发生了什么事？ ②谁做的这件事？③这件事是什么时候发生的？④它将影响哪些其他事物？

配置状态变化对大型软件开发项目的成功有重大影响。当大量人员在一起工作时，可能一个人并不知道另一个人在做什么。两名开发人员可能试图按照相互冲突的想法去修改同一个软件配置项；软件工程队伍可能耗费几个人月的工作量根据过时的硬件规格说明开发软件；察觉到所建议的修改有严重副作用的人可能还不知道该项修改正在进行。配置状态报告通过改善所有相关人员之间的通信，帮助消除这些问题。

13.7 能力成熟度模型

美国卡内基梅隆大学软件工程研究所在美国国防部资助下于 20 世纪 80 年代末建立的能力成熟度模型（capability maturity model，CMM），是用于评价软件机构的软件过程能力成熟度的模型。最初，建立此模型的目的主要是，为大型软件项目的招投标活动提供一种全面而客观的评审依据，发展到后来，此模型又同时被应用于许多软件机构内部的过程改进活动中。

在无规则和混乱的管理之下，先进的技术和工具并不能发挥出应有的作用。

改进对软件过程的管理是消除软件危机的突破口，再也不能忽视在软件过程中管理的关键作用了。

能力成熟度模型的基本思想是，由于问题是由我们管理软件过程的方法不当引起的，所以新软件技术的运用并不会自动提高软件的生产率和质量。能力成熟度模型有助于软件开发机构建立一个有规律的、成熟的软件过程。改进后的软件过程将开发出质量更好的软件。

软件过程包括各种活动、技术和工具，因此，它实际上既包括了软件开发的技术方面又包括了管理方面。CMM 的策略是，力图改进对软件过程的管理，而在技术方面的改进是其必然的结果。

CMM 在改进软件过程中所起的作用主要是，指导软件机构通过确定当前的过程成熟度并识别出对过程改进起关键作用的问题，从而明确过程改进的方向和策略。通过集中开展与过程改进的方向和策略相一致的一组过程改进活动，软件机构便能稳步而有效地改进其软件过程，使其软件过程能力得到循序渐进的提高。

对软件过程的改进，是在完成一个又一个小的改进步骤基础上不断进行的渐进过程，而不是一蹴而就的彻底革命。

CMM 把软件过程从无序到有序的进化过程分成 5 个阶段，并把这些阶段排序，形成 5 个逐层提高的等级。这 5 个成熟度等级定义了一个有序的尺度，用以测量软件机构的软件过程成熟度和评价其软件过程能力，这些等级还能帮助软件机构把应做的改进工作排出优先次序。成熟度等级是妥善定义的向成熟软件机构前进途中的平台，每个成熟度等级都为软件过程的继续改进提供了一个台阶。

CMM 对 5 个成熟度级别特性的描述，说明了不同级别之间软件过程的主要变化。从“1 级”到“5 级”，反映出一个软件机构为了达到从一个无序的、混乱的软件过程进化到一种有序的、有纪律的且成熟的软件过程的目的，必须经历的过程改进活动的途径。每一个成熟度级别都是该软件机构沿着改进其过程的途径前进途中的一个台阶，后一个成熟度级别是前一个级别的软件过程的进化目标。

CMM 的每个成熟度级别中都包含一组过程改进的目标，满足这些目标后一个机构的软件过程就从当前级别进化到下一个成熟度级别；每达到成熟度级别框架的下一个级别，该机构的软件过程都得到一定程度的完善和优化，也使得过程能力得到提高；随着成熟度级别的不断提高，该机构的过程改进活动取得了更加显著的成效，从而使软件

过程得到进一步的完善和优化。CMM 就是以上述方式支持软件机构改进其软件过程的活动。

CMM 通过定义能力成熟度的 5 个等级，引导软件开发机构不断识别出其软件过程的缺陷，并指出应该做哪些改进，但是，它并不提供做这些改进的具体措施。

能力成熟度的 5 个等级从低到高依次是：

1. 初始级
2. 可重复级
3. 已定义级
4. 已管理级
5. 优化级

五个级别的关键过程域

初始级

可重复级：需求管理、软件项目计划、软件项目跟踪与监督、软件分包合同管理、软件质量保证、软件配置管理

已定义级：机构过程焦点、机构过程定义、培训大纲、综合软件管理、组间协调、同行评审

已管理级：定量过程管理、软件质量管理

优化级：缺陷预防、技术更新管理、过程更改管理

1. 初始级

软件过程的特征是无序的，有时甚至是混乱的。几乎没有什么过程是经过定义的（即没有一个定型的过程模型），项目能否成功完全取决于开发人员的个人能力。

处于这个最低成熟度等级的软件机构，基本上没有健全的软件工程管理制度，其软件过程完全取决于项目组的人员配备，所以具有不可预测性，人员变了过程也随之改变。如果一个项目碰巧由一个杰出的管理者和一支有经验、有能力的开发队伍承担，则这个项目可能是成功的。但是，更常见的情况是，由于缺乏健全的管理和周密的计划，延期交付和费用超支的情况经常发生，结果，大多数行动只是应付危机，而不是完成事先计划好的任务。

总之，处于 1 级成熟度的软件机构，其过程能力是不可预测的，其软件过程是不稳定的，产品质量只能根据相关人员的个人工作能力而不是软件机构的过程能力来预测。

2. 可重复级

软件机构建立了基本的项目管理过程(过程模型),可跟踪成本、进度、功能和质量。已经建立起必要的过程规范，对新项目的策划和管理过程是基于以前类似项目的实践经验，使得有类似应用经验的软件项目能够再次取得成功。达到 2 级的一个目标是使项目管理过程稳定，从而使得软件机构能重复以前在成功项目中所进行过的软件项目工程实践。

处于 2 级成熟度的软件机构，针对所承担的软件项目已建立了基本的软件管理控制制度。通过对以前项目的观察和分析，可以提出针对现行项目的约束条件。项目负责人跟踪软件产品开发的成本和进度以及产品的功能和质量，并且识别出为满足约束条件所应解决的问题。已经做到软件需求条理化，而且其完整性是受控制的。已经制定了项目标准，并且软件机构能确保严格执行这些标准。项目组与客户及承包商已经建立起一个稳定的、可管理的工作环境。

处于 2 级成熟度的软件机构的过程能力可以概括为，软件项目的策划和跟踪是稳定的，已经为一个有纪律的管理过程提供了可重复以前成功实践的项目环境。软件项目工程活动处于项目管理体系的有效控制之下，执行着基于以前项目的准则且合乎现实的计划。

3. 已定义级

软件机构已经定义了完整的软件过程（过程模型），软件过程已经文档化和标准化。所有项目组都使用文档化的、经过批准的过程来开发和维护软件。这一级包含了第 2 级的全部特征。

在第 3 级成熟度的软件机构中，有一个固定的过程小组从事软件过程工程活动。当需要时，过程小组可以利用过程模型进行过程例化活动，从而获得一个针对某个特定的软件项目的过程实例，并投入过程运作而开展有效的软件项目工程实践。同时，过程小组还可以推进软件机构的过程改进活动。在该软件机构内实施了培训计划，能够保证全体项目负责人和项目开发人员具有完成承担的任务所要求的知识和技能。

处于 3 级成熟度的软件机构的过程能力可以概括为，无论是管理活动还是工程活动都是稳定的。软件开发的成本和进度以及产品的功能和质量都受到控制，而且软件产品的质量具有可追溯性。这种能力是基于在软件机构中对已定义的过程模型的活动、人员和职责都有共同的理解。

4. 已管理级

软件机构对软件过程（过程模型和过程实例）和软件产品都建立了定量的质量目标，所有项目的重要的过程活动都是可度量的。该软件机构收集了过程度量和产品度量的方法并加以运用，可以定量地了解和控制软件过程和软件产品，并为评定项目的过程质量和产品质量奠定了基础。这一级包含了第 3 级的全部特征。

处于 4 级成熟度的软件机构的过程能力可以概括为，软件过程是可度量的，软件过程在可度量的范围内运行。这一级的过程能力允许软件机构在定量的范围内预测过程和产品质量趋势，在发生偏离时可以及时采取措施予以纠正，并且可以预期软件产品是高质量的。

5. 优化级

软件机构集中精力持续不断地改进软件过程。这一级的软件机构是一个以防止出现缺陷为目标的机构，它有能力识别软件过程要素的薄弱环节，并有足够的手段改进它们。在这样的机构中，可以获得关于软件过程有效性的统计数据，利用这些数据可以对新技术进行成本/效益分析，并可以优化出在软件工程实践中能够采用的最佳新技术。这一级包含了第 4 级的全部特征。

这一级的软件机构可以通过对过程实例性能的分析和确定产生某一缺陷的原因，来防止再次出现这种类型的缺陷；通过对任何一个过程实例的分析所获得的经验教训都可以成为该软件机构优化其过程模型的有效依据，从而使其他项目的过程实例得到优化。这样的软件机构可以通过从过程实施中获得的定量的反馈信息，在采用新思想和新技术的同时测试它们，以不断地改进和优化软件过程。

处于 5 级成熟度的软件机构的过程能力可以概括为，软件过程是可优化的。这一级的软件机构能够持续不断地改进其过程能力，既对现行的过程实例不断地改进和优化，又借助于所采用的新技术和新方法来实现未来的过程改进。

一些统计数字表明，提高一个完整的成熟度等级大约需要花 18 个月到 3 年的时间，但是从第 1 级上升到第 2 级有时要花 3 年甚至 5 年时间。

13.8 小结

- 软件工程包括技术和管理两方面的内容，是技术与管理紧密结合的产物。因此，有效的管理是大型软件工程项目成功的关键。
- 软件项目管理始于项目计划，而第一项计划活动就是估算。为了估算项目工作量和完成期限，首先需要预测软件规模。

度量软件规模的常用技术主要有代码行技术和功能点技术。这两种技术各有优缺点，应该根据项目特点及从事计划工作的人对这两种技术的熟悉程度，选用适用的技术。

根据软件规模可以估算出完成该项目所需的工作量，常用的估算模型为静态单变量模型、动态多变量模型和 COCOMO2 模型。为了使估算结果更接近实际值，通常至少同时使用上述 3 种模型中的两种。通过比较和协调使用不同模型得出的估算值，有可能得到比较准确的估算结果。成本估算模型通常也同时提供了估算软件开发时间的方程式，这样估算出的开发时间是正常开发时间。

管理者必须制定出一个详细的进度表，以便监督项目进度并控制整个项目。常用的制定进度计划的工具有 Gantt 图和工程网络，这两种工具各有优缺点，通常，联合使用 Gantt 图和工程网络来制定进度计划并监督项目进展状况。

高素质的开发人员和合理的项目组组织结构，是软件项目取得成功的关键。比较典型的组织结构有民主制程序员组、主程序员组和现代程序员组等 3 种，这 3 种组织方式的适用场合并不相同。

软件质量保证是在软件过程中的每一步都进行的活动。软件质量保证措施主要有基于非执行的测试（也称为复审）、基于执行的测试（即通常所说的测试）和程序正确性证明。软件复审是最重要的软件质量保证活动之一，它的优点是在改正错误的成本相对比较低时就能及时发现并排除软件错误。

软件配置管理是应用于整个软件过程中的保护性活动，是在软件整个生命期内管理变化的一组活动。软件配置管理的目标是，使变化能够更正确且更容易被适应，在需要修改软件时减少为此而花费的工作量。

能力成熟度模型（CMM）是改进软件过程的有效策略。它的基本思想是，因为问题是管理软件过程的方法不恰当造成的，所以采用新技术并不会自动提高软件生产率和软件质量，应该下大力气改进对软件过程的管理。

事实上对软件过程的改进不可能一蹴而就，因此，CMM 以增量方式逐步引入变化，它明确地定义了 5 个成熟度等级，一个软件开发组织可以用一系列小的改良性步骤迈入更高的成熟度等级。