高等院校计算机应用系列教材

软件工程实用教程

(微课版)

和孟佯　主　编
赵国桦　副主编

清華大學出版社
北　京

内 容 简 介

本书系统全面地介绍了软件工程的基本原理与核心技术，章节安排合理有序。每章均设有学习目标、主要内容、小结以及思考与练习，旨在帮助读者理解软件工程的关键知识，并初步掌握基本的软件开发方法。全书共分为 10 章，内容包括软件工程概述、软件过程、需求分析与软件需求规约、结构化分析、结构化设计、面向对象分析、面向对象设计、统一建模语言、编码与测试及软件项目管理。

本书力求语言通俗易懂，采用大量案例分析并配备教学 PPT 和教学视频，帮助读者快速掌握软件工程的基础知识与项目管理技能，为实际应用打下坚实基础。本书适合作为高等院校计算机与信息类相关专业的教材或教学参考书，也可作为研究生及软件工程从业者的参考资料。

本书配套的电子课件和习题答案可以通过 http://www.tupwk.com.cn/downpage 网站下载，也可以扫描前言中的二维码获取。扫描前言中的视频二维码可以直接观看教学视频。

图书在版编目（CIP）数据

软件工程实用教程 : 微课版 / 和孟佯主编.
北京 : 清华大学出版社, 2025. 4. -- (高等院校计算机应用系列教材).
ISBN 978-7-302-68218-9

Ⅰ. TP311.5

中国国家版本馆 CIP 数据核字第 20254F3V22 号

责任编辑：胡辰浩
封面设计：高娟妮
版式设计：恒复文化
责任校对：马遥遥
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：河北盛世彩捷印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**14.25　　**字　　数：**346 千字
版　　次：2025 年 4 月第 1 版　　**印　　次：**2025 年 4 月第 1 次印刷
定　　价：69.00 元

产品编号：104843-01

前　言

软件工程(Software Engineering，SE)是一门通过工程化方法构建和维护高效、实用且高质量软件的学科。它涉及程序设计语言、数据库、软件开发工具、系统平台、标准及设计模式等多个领域。掌握先进的软件开发工具及软件工程管理理论与方法，不仅能显著提高工作效率，还能增强软件系统设计与项目实施的能力。

本书从软件危机的起源讲起，遵循“从简单到复杂”和“从抽象到具体”的原则，系统介绍软件开发过程的基本原理及核心技术。本书共分为10章，第1章是软件工程概述，介绍软件危机的起源及软件工程的基本原理；第2章介绍软件过程中涉及的不同软件生命周期模型；第3章讲解软件需求分析中的相关知识及常用的需求记录与分析方法；第4章介绍常见的结构化分析工具；第5章讲解结构化设计中的常用规则；第6章介绍面向对象分析方法；第7章讲解面向对象设计的基本原则；第8章重点介绍统一建模语言；第9章讲解编码与测试方法；第10章介绍常见的软件项目管理方法。

本书内容丰富、结构合理、思路清晰、语言简练流畅、示例翔实并配有教学视频。每章开篇提供内容概述和学习目标，引导读者了解本章的学习方向。正文部分结合每章的知识点和关键技术，使用通俗易懂的语言进行介绍。每章的末尾设有本章小结，总结本章的内容、重点和难点。除此之外，每章还安排了针对性的思考题和练习题，帮助读者巩固所学知识。同时，配套的教学视频对每章的知识点进行了讲解和总结。

本书主要面向计算机相关专业的初学者，适合作为高等院校计算机与信息类相关专业的教材或教学参考书，也可作为研究生及软件工程从业者的参考资料。

除封面署名的作者外，参与本书编写的人员还有陈超、贾中海、刘璐豪、穆阳茜、杨超霞(排名不分先后)。

由于作者水平有限，书中难免存在不足，恳请专家及广大读者批评指正。在编写本书的过程中参考了相关文献，在此向这些文献的作者深表感谢。我们的电话是010-62796045，信箱是992116@qq.com。

本书配套的电子课件和习题答案可以通过 http://www.tupwk.com.cn/downpage 网站下载，也可以扫描下方左侧的二维码获取。扫描下方右侧的视频二维码可以直接观看微课视频。

扫描下载

配套资源

扫一扫

看视频

作　者

2024 年 10 月

目　录

∞ 第1章 ∞

软件工程概述

在过去的半个世纪里，计算机在商业领域的应用不断扩大和深化。最初，计算机的运行速度较慢，结构也较为简单。然而，随着科技的持续进步与创新，计算机的运算能力显著提升，结构变得更加复杂和精密。与此同时，生产成本的降低和市场竞争的加剧，使得计算机的价格逐渐变得更加亲民。多次重大的技术突破为计算机的发展注入了强大动力，不仅显著提升了运行速度，还大幅降低了制造成本，推动了计算机的广泛应用和快速发展。

随着计算机性能的不断提升，其处理复杂程序的能力也在不断增强。因此，软件工程师们必须以最高且最具成本效益的方式来应对愈发庞大和复杂的挑战。值得钦佩的是，软件工程师们不仅通过创新来应对这些挑战，还积极借鉴过去的编程经验和其他专业领域的规范化工程方法，从中汲取智慧。在追求最佳成本效益和最高效率的过程中，积累的创新经验以及编写高质量程序的心得，逐步被系统化并整合为一个完善的知识体系，构成了软件工程方法论的核心思想。具体叙述如下：

- 应依托正确扎实的理论依据，确保每一项决策都建立在科学的依据之上。同时，在面临理论知识支撑不足的情况下，应充分吸收并借鉴以往的经验教训，并对其进行系统整理与归纳，从而形成一套切实可行的指导准则。
- 在系统设计过程中，需要全面权衡各个目标之间的利益，确保所选方案既能满足系统需求，又能兼顾各方的利益诉求。
- 为实现成本效益的优化，应采取务实的策略，深入考虑经济成本因素。在提升整体效益的同时，确保资源的合理分配与高效利用。

总之，软件工程是一种用于开发软件的系统化、规范化的工程方法，其主要目的是提高软件产品的质量和开发效率，并减少维护的难度。本章将从工程学科的发展历程开始，全面而详细地描述软件工程的起源及其基本原理。

本章的学习目标：

- 了解工程学科的发展历程
- 了解软件危机出现的原因
- 掌握软件工程的基本原理

1.1 软件危机

1.1.1 工程学科的发展历程

在本节中，我们将简要回顾软件工程学科的发展历程，探索其如何从大约半个世纪前的微小起点，逐步发展成为如今的庞大体系。同时需要指出，尽管软件工程学科仍存在诸多不足之处，但它无疑是应对软件危机最有效的解决方案之一，其重要性和价值不容忽视。

在过去的半个世纪里，得益于众多研究人员和软件专业人士的不懈努力，软件工程理论取得了显著的进步与发展。早期的程序员往往采用探索性的编程方式。在这种风格下，每个程序员独立探索软件开发技术，依靠直觉、经验、灵感和个人偏好进行操作。对于那些对软件工程原理了解不足的人来说，他们在编写程序时也多倾向于采用这种试探性的编程风格。我们可以将这种“试探性”程序开发风格类比为艺术创作，因为艺术在很大程度上也是由直觉引导的。过去，众多关于程序员的故事展现了他们如同艺术家般，能够运用一些深奥的知识创作出卓越的程序。

回顾并分析过去 50 年软件开发风格的演变，可以发现编程工作已从早期的一种高深的“艺术形式”逐渐转变为一种更为普遍的“工艺形式”，并最终发展为一门工程学科。实际上，这种演进模式与其他工程学科相比并没有显著区别。无论其所涉及的领域如何，技术的发展普遍遵循着相似的路径和模式。图 1-1 展示了各种技术发展的普遍演进规律。

以玻璃生产技术的发展来举例说明：在远古时期，与玻璃制造相关的技术鲜为人知且被高度保密，那些掌握技术奥秘的人往往将其视为家族秘密，代代相传。然而，随着时间的推移，玻璃生产逐渐从“艺术”转变为“工艺”，技工与学徒之间开始分享知识与经验，从而使得知识库得以不断积累和扩展。再经过一个阶段的发展，通过系统的知识整合，以及科学原理的融入，最终形成了现代玻璃制造技术，促进了技术的飞跃与发展。

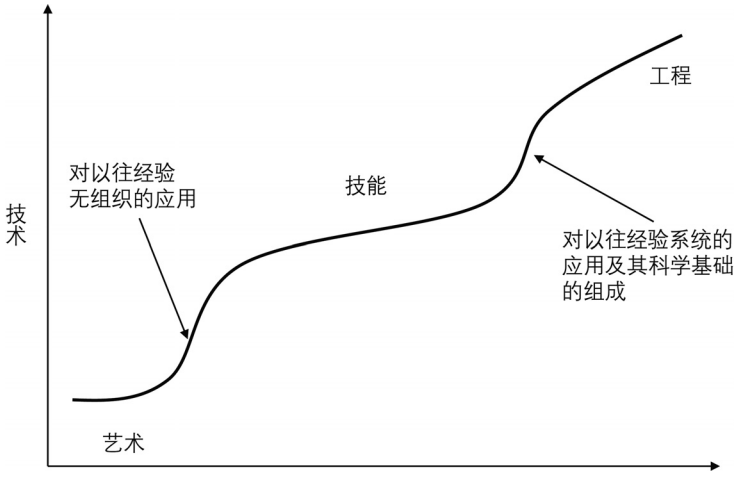

图 1-1　技术发展普遍规律

当然，时至今日，依然存在一些不同的声音对软件工程学科中的某些方法论和指导方针提出质疑，认为它们缺乏充分的科学依据且过于主观。然而，必须指出的是，这些质疑并未否定采用软件工程技术在实际应用中的价值。实际上，软件工程技术对于低成本、高效率开发高质量软件具有显著的促进作用。软件工程实践已经被广泛证明是大型软件开发项目中不可或缺的一环，它为项目的顺利开展提供了有力的保障。虽然在某些小型项目的开发中，“探索性”的开发方式仍然能够取得成功，但软件工程技术的运用无疑为这些项目的成功提供了更多的保障和支持。因此，不能简单地将软件工程学科中的方法论和指导方针视为缺乏科学依据和主观性过强的存在，而应客观辩证地看待其在软件开发实践中的重要作用。

1.1.2　软件危机的介绍

自第一台计算机问世以来，软件的开发应运而生。随着计算机技术的迅猛发展和应用领域的不断拓宽，自 20 世纪 60 年代中期起，软件需求激增，软件的数量也大幅增加。这种趋势推动了软件产业的发展。为了更好地理解软件产业的演变过程，可以将软件开发的发展历程划分为以下三个时代。

1. 程序设计时代(1946—1956 年)

在程序设计时代，软件的开发主要依赖于个体的手工操作。程序设计者使用机器语言和汇编语言作为工具，专注于提升编程技巧和优化程序运行效率。

然而，在程序设计的过程中，他们并未充分重视其他辅助手段的作用，导致所设计的程序往往难以阅读、理解和修改。在这个时期，软件的特征主要表现为单一的程序和程序设计概念，而对程序设计方法的重要性并未给予足够的重视。随着技术的进步，这种生产方式逐渐暴露出其局限性和不足，为后续的软件发展带来了新的挑战与机遇。

2. 程序系统时代(1956—1968 年)

随着计算机技术的广泛应用，其应用领域不断扩展，软件需求呈现出持续增长的趋势。随着软件所需处理的问题领域不断扩大，程序的复杂性也随之增强。面对这些挑战，软件设计者逐渐放弃了传统的个体手工开发方式，转而采用小团队协作的生产模式，这标志着软件开发正式进入作坊式生产方式的程序系统时代。

这种生产方式虽然在一定程度上提高了软件开发效率，但仍然面临诸多挑战和限制，需要不断探索和改进。随着大型程序的复杂度不断提升，合作开发逐渐成为程序设计中的关键环节。与此同时，软件行业的飞速发展吸引了众多其他行业人才涌入，程序员数量急剧增加。在这一热潮中，一方面，软件开发的需求日益旺盛，软件的规模不断扩大，结构日益复杂；另一方面，当发现错误必须需要修改程序时，软件维护所需的资源耗费变得极为严重，甚至导致许多软件最终无法维护。除此以外，由于缺乏统一的开发规范和方法论指导，开发人员的技术水平参差不齐，难以应对大规模、结构复杂的软件开发任务。这些尖锐的矛盾直接引发了软件危机。

3. 软件工程时代(1968 年至今)

1968 年，在联邦德国召开的国际会议上，软件危机问题成为讨论的焦点。这次会议不仅正式提出并使用了“软件工程”这一术语，还标志着这一新兴工程科学的诞生。软件工程时代的生产方式引入了工程领域的概念、原理、技术和方法，并结合数据库、开发工具、开发环境、网络、分布式系统以及面向对象技术，以实现更加高效的软件开发。

尽管软件的特性推动了开发技术的显著进步，但尚未取得突破性进展，导致软件价格持续攀升，软件危机的问题仍未得到根本解决。

在软件发展的第二阶段末期，得益于计算机硬件技术的飞速进步，计算机的运行速度、存储容量和可靠性显著提升，生产成本则大幅度下降，这为计算机的广泛应用奠定了坚实基础。随之而来的是一系列复杂且规模庞大的软件开发项目的涌现。然而，尽管软件开发技术不断进步，却始终未能完全满足日益增长的发展需求。在软件开发过程中，许多难题难以解决，这些问题不断积累起来，最终形成了尖锐的矛盾，最终引发了软件危机。所谓软件危机，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峻挑战。这些挑战远非 “无法正常运行”那么简单。实际上，几乎所有的软件都或多或少地存在某种问题。软件危机主要聚焦于软件开发的策略，如何满足日益增长的软件需求，以及如何有效维护数量庞大的现有软件。

软件危机的具体表现主要集中在以下几个方面。

(1) 软件危机在经费预算和完成时间方面表现为：预算经常超出预期，项目完成时间不断推迟。这主要是因为缺乏丰富的软件开发经验和数据积累，导致计划制订困难。主观制订的计划往往与实际执行情况存在较大偏差，从而使得开发经费频繁超出预算。同时，由于对工作量以及开发难度的估计不足，进度计划往往难以按时完成，开发时间不断延迟，给项目带来了巨大的压力。

(2) 软件危机在满足用户需求方面表现为：开发的软件常常无法满足用户的期望。这主要是因为在开发初期，对用户的需求了解不够明确和具体，导致需求描述不精确。随着开发工作的推进，软件开发人员与用户之间的沟通不充分，使得问题无法及时得到解决，最终导致开发的软件与用户的实际需求差距较大，甚至导致项目开发失败。

(3) 软件危机在软件可维护性方面表现为：开发的软件往往难以维护。由于缺乏统一的、公认的开发规范，软件开发人员各自为政，导致开发过程缺乏完整且规范的文档记录。这使得在软件出现问题时难以进行有针对性的修改。此外，程序结构的不合理也增加了维护的难度，一旦在运行时发现错误，通常难以进行有效的修复，从而严重影响了软件的可维护性。

(4) 软件危机在软件可靠性方面表现为：开发的软件往往缺乏足够的可靠性。这主要源于开发过程中缺乏有效的质量保障体系和措施，导致软件质量无法得到有效的保障。同时，在软件测试阶段，由于缺乏严格、充分且全面的测试，使得提交给用户的软件存在大量潜在问题。这些问题在软件运行过程中逐渐暴露，严重影响了软件的可靠性和稳定性。

1.1.3　软件危机的原因

软件作为逻辑部件而非物理部件，其进度和质量的控制与维护难度较高。由于软件规模庞大且需要多人协作开发，实施严格、科学的管理至关重要。然而，软件开发过于依赖于个人的智力劳动与经验，且往往在没有完整、准确地理解用户需求的情况下匆忙展开，进一步增加了软件开发的复杂性和风险。在软件的开发和维护过程中之所以存在诸多问题，一方面与软件本身的特性有关，另一方面与软件开发和维护的方法不当有关。客观原因包括软件需求量大、规模庞大等；主观原因则涉及软件本身的特性以及开发、维护方法的不足。综上所述，软件危机的原因可归结为多方面因素共同作用的结果。

(1) 随着技术的进步，软件的规模持续扩大，结构日益复杂。举例来说，1968 年美国航空公司的订票系统已包含 30 万条指令，IBM 360 OS 的第 16 版更是达到 100 万条指令。到 1973 年，美国阿波罗计划的软件规模更是突破了千万条指令。这些大型软件不仅功能丰富，还需适应多样性化的运行环境。随着计算机应用的日益普及，软件开发的规模不断膨胀，软件的结构也日趋复杂。相比硬件设计，软件设计的逻辑复杂度估计高出 10 至 100 倍。对于如此庞大的软件规模，其调用关系、接口信息及数据结构都异常复杂，这种复杂程度已经超出了人类所能轻松处理的范围。

(2) 软件开发管理是一项既困难又复杂的任务。与硬件有着显著不同，软件作为计算机系统中的逻辑部件，具有无形性。在代码编写和计算机试运行之前，由于其庞大的规模和复杂的结构，使得开发进度的度量与质量评估变得异常困难。这些因素共同导致了管理难度的提升，使得进度控制、质量控制以及可靠性的保障都面临巨大挑战。此外，软件在运行过程中不会因为长期运行而损坏。如果在运行过程中出现问题，通常是由于在开发阶段引入的缺陷且在测试阶段未被发现所导致的。因此，软件维护通常需要对原有设计进行改进，以确保其稳定性和可靠性。

(3) 软件开发费用的持续上涨是业界的一大难题。作为资金和人力密集型的产业，大型软件的开发需要投入大量的人力资源，且开发周期较长，导致费用快速上升。由于软件生产仍主要依赖手工方式，且软件作为高度密集化的知识和技术的综合产物，现有的人力资源难以满足社会对该领域迅速增长的需求。因此，软件开发成本上升的趋势预计将持续下去。

(4) 软件开发技术相对滞后。在 20 世纪 60 年代，计算机理论问题的研究受到人们广泛关注，包括编译原理、操作系统原理、数据库原理、人工智能原理、形式语言理论等。然而，相比之下，软件开发技术的研究进展较为缓慢，这导致用户所需的软件复杂性与软件技术解决复杂性的能力之间存在巨大差距，并且这一差距仍在不断扩大。

(5) 生产方式落后，亟待改进。软件开发仍然依赖于个体手工操作，缺乏统一的标准和规范，缺少明确和系统性的指导。开发过程中主要依赖于个人的习惯和喜好，流程无章可循，无规范可遵循。此外，传承方式主要依赖言传身教，缺乏系统性和科学性。

(6) 开发工具亟待更新，生产效率提升缓慢。软件开发工具相对落后，缺乏高效率的开发工具的支持，因而导致软件生产率低下，难以满足市场需求。在 1960 年至 1980 年间，计算机硬件的生产因为采用计算机辅助设计、自动生产线等先进工具，生产效率显著提高，提升幅度高达 100 万倍。然而，同期软件生产率仅提升了 2 倍，两者相差悬殊。这种情况显然不利于软件产业的健康发展。

1.1.4 消除软件危机的途径

软件工程逐渐成为应对软件危机的关键解决方案之一。图 1-2 展示了硬件成本与软件成本的比值随时间变化的曲线，从图中曲线可以看出，越来越多的企业将更多的预算用于软件的购买之上。软件产品不仅价格日益昂贵，而且给用户带来了一系列问题，如修改、诊断错误和增强软件产品功能变得异常困难，难以满足用户需求，可靠性差，甚至延迟交付等。在这些问题中，软件成本的持续增长无疑是软件危机最为严重的表现形式之一。过去在购买硬件时附带的软件通常是免费提供的。然而，如今软件成本已远超硬件，这不仅改变了市场格局，也凸显了软件危机带来的深远影响。

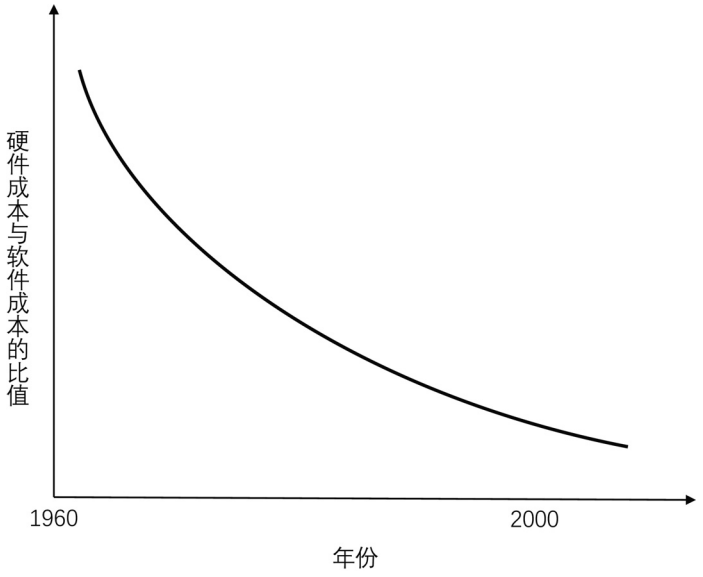

图 1-2　硬件与软件成本比值随时间变化的曲线

1.2 软件工程

软件工程是一门指导计算机软件开发和维护的工程科学，研究并应用系统化、规范化和可量化的流程化方法来开发和维护软件。它涉及的知识领域十分广泛，包含程序设计语言、数据库、软件开发工具、系统平台、标准及设计模式等多个方面，旨在提高软件生产效率、提升软

件质量并降低开发成本。为了应对软件危机，软件工程借鉴了其他产业的工程化生产经验，运用工程的概念、原理、技术和方法，并结合经过时间考验而验证正确的管理技术与方法，共同推动软件工程的发展。1968 年，“软件工程”这一术语在北大西洋公约组织的一次计算机学术会议上首次正式提出。这次会议聚焦于探讨软件危机问题，在软件发展史上具有重要的里程碑意义。

1.2.1　软件工程的出现

基于持续的创新和编写高质量程序所累积的丰富经验，近年来软件工程技术不断进步与发展。逐渐形成以下创新成果和编程经验，并共同推动了软件工程学科的发展。

1. 早期“探索式”编程

以当今的技术标准来衡量，早期的商业计算机运行速度和功能都显得相对滞后。在那个时期，即便是简单地处理一些紧急任务，也需要耗费大量的计算时间。因此，当时的程序规模都较小，复杂度也相对较低。这些程序通常采用汇编语言编写而成，代码的长度往往仅限于几百行的汇编指令。每位程序员在编程过程中都会展现出自己独特的编程风格，甚至会根据不同程序进行调整。这种个性化的编程方式被称为“探索式”编程风格。

2. 高级语言编程

随着半导体技术的应用，计算机的运行速度显著提升，使得人们能够利用更强大的计算机来解决更大、更复杂的问题。与此同时，FORTRAN、ALGOL 和 COBOL 等高级语言的问世，显著降低了开发软件产品所需的工作量，并帮助程序员编写更大规模的程序。然而，这类程序的源代码通常达到数十万行，反映出当时的软件开发风格仍处于较为初级的阶段。

3. 基于流程控制的设计

随着程序规模和复杂性的不断增长，传统探索性编程风格已无法满足日益增长的需求。程序员逐渐发现，编写低成本且正确的程序变得愈发困难，而理解和维护他人编写的程序也变得更具有挑战性。为了解决上述问题，资深程序员建议其他程序员，要特别重视程序的控制流结构的设计。一个程序的控制流结构能够清晰地展示程序的执行顺序，对于提高程序质量和可维护性至关重要。为了帮助开发出具备优良的控制流结构的程序，流程图技术应运而生并不断发展。时至今日，流程图技术仍然被广泛应用于算法的描述和设计中。图 1-3 展示了为同一问题编写程序代码的两种不同方法。相比图 1-3(a)，图 1-3(b)中的程序段具有更为简洁的控制流结构。

```
if(customer_savings_balance>withdrawal_request){
100:              issue_money=TRUE;
              GOTO 110;
            }
            else if(privileged_customer==TRUE)
                        GOTO 100;
          else GOTO 120;
110:          activate_cash_dispenser(withdrawal_request);
              GOTO 130;
120:      print(error);
130:      end-transaction();
```

(a)

```
if(privileged_customer(customer_savings_balance>withdrawal_request)){
            activate cash dispenser(withdrawal request);
}.
else print(error);
end-transaction();
```

(b)

图 1-3　非结构化程序示例与结构化程序示例

图 1-4 展示了图 1-3 中的两个程序段的流程图。通过为多个问题绘制流程图，可以得出一个结论：如果一个程序的流程图表示起来很复杂，那么该程序的理解和维护难度必然增加。复杂的流程图导致其所表示的程序难以理解的主要原因在于，通常人们在理解一个程序时，需要追踪其所有执行路径上的执行序列，这显著增加了理解的难度。以图 1-3(a)中的程序为例，需要追踪多条执行路径，如 1-2-3-7-8-10、1-4-5-6-9-10 和 1-4-5-2-3-7-8-10 等，导致程序的理解过程变得复杂。如果程序的控制流结构比较混乱，那么确定其所有的执行路径就会异常复杂，同时以此为基础追踪这些路径上的执行顺序则更加困难。因此，混乱的流程图会使要表示的程序难以理解和维护。此外，滥用 GOTO 语句是导致程序的控制结构变得混乱的主要原因，因为 GOTO 语句可以随意改变控制流。因此，一个优秀的程序应该具备清晰的控制结构。

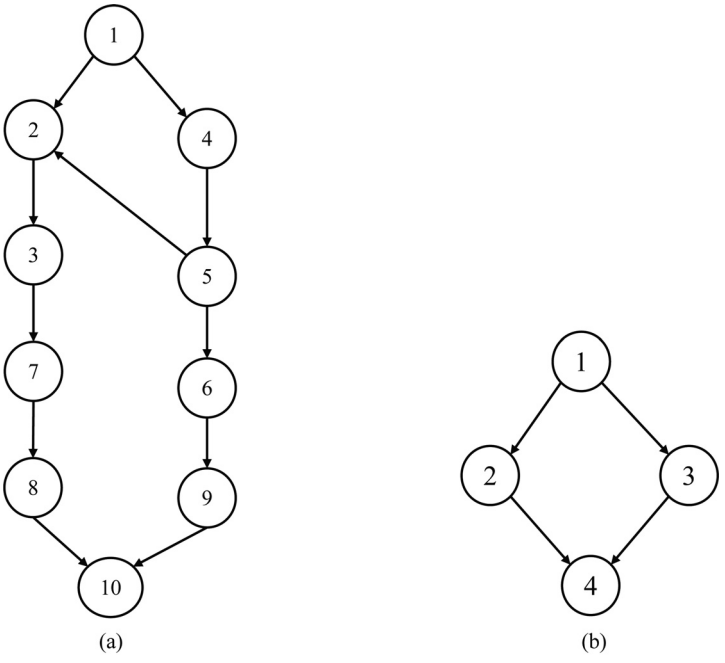

图 1-4　图 1-3(a)和图 1-3(b)中程序的控制流图

因此，高效的程序应当尽量限制 GOTO 语句的使用。然而，许多程序员仍然使用汇编语言。在汇编语言中，跳转指令是常用的分支工具。因此，具有汇编语言编程背景的程序员常认为在编程中使用 GOTO 语句是不可避免的。针对这一问题，Dijkstra 发表了一篇著名的论文“*GOTO*

Statements Considered Harmful”，指出“GOTO 语句是有害的”。正如预料的那样，在阅读了 Dijkstra 的文章之后，许多程序员发表文章进行反驳，强调 GOTO 语句的优势及其必要性。然后，随着研究的深入，逐渐达成共识：只需要顺序、选择和迭代这三种编程结构就足以表达任何程序逻辑。随着时间的推移，越来越多的程序员开始认同一个观点，即不使用 GOTO 语句同样也可以解决编程中的任何问题，并且应当尽量避免滥用 GOTO 语句。这一理念逐渐成为结构化编程方法的核心基础，推动了编程实践向更加规范、清晰的方向发展。

当一个程序完全采用顺序、选择和迭代等结构化逻辑来构建时，它即被视作结构化程序。结构化编程强调限制 GOTO 语句的使用，以此避免非结构化的控制流，从而提升程序的清晰性和可维护性。如果编程语言支持单入单出结构的程序构建，如 if-then-else 和 do-while 等控制结构，那么它将极大促进结构化编程的实现，有助于编写逻辑清晰、易于维护的程序。因此，结构化编程的原则着重于为程序构建精巧且高效的控制结构方案。以图 1-4 为例，该图清晰地揭示了结构化和非结构化程序之间的本质区别。除此以外，结构化程序这一术语并不仅限于控制结构方面，它还涵盖其他编程特性，这些特性共同构成了结构化编程的丰富内涵。例如，结构化程序的核心特性之一在于其模块化设计。通过将模块化的程序分解为多个相互独立且依赖度低的模块，能够显著提升代码的可读性、可维护性及可重用性。这种模块化结构不仅有助于降低程序复杂性，提高开发效率，还使程序更易于理解和调试。

与非结构化程序相比，编写结构化程序的主要优势表现在以下几点。

(1) 显著提升代码质量并减少错误率。研究表明，程序员在使用结构化的 if-then-else 和 do-while 语句时，通常比使用 test-and-branch 代码构建时犯的错误更少。

(2) 显著提升代码可读性和开发效率。除显著减少错误率外，结构化程序还具备诸多优势。其高可读性使得代码更易于理解，从而提升了开发的整体效率。

(3) 显著提升代码代码的可维护性，并降低后期修改和扩展的难度。与非结构化程序相比，结构化程序通常所需要的工作量更少，这得益于其清晰的控制结构和模块化的设计，使得开发过程更加高效有序。

尽管结构化编程的诸多益处已得到人们的广泛认可。但在实际应用中，有时出于特定需求(例如处理异常情况或支持非正常循环退出等)的考虑，程序员可能无法完全严格遵循结构化编程的原则。这体现了在实际编程过程中，灵活性和适应性同样重要，需要根据具体场景权衡和选择最合适的编程方法。

为了支持结构化编程，PASCAL、MODULA 和 C 等语言应运而生，这些语言的出现极大地推动了结构化编程的普及和应用，旨在提供结构化编程所需的特性和工具。这些编程语言不仅支持编写模块化程序，还具备出色的控制结构特性，从而有效解决了混乱的控制结构所带来的常见问题。随着编程技术的发展，关注点逐渐从优化控制结构转向设计优质的数据结构方案，以进一步提升程序性能和代码质量。

4. 面向数据结构的设计

随着集成电路技术的不断发展，计算机性能显著提升，使得解决复杂问题变得更加高效。

这一进步为软件工程师们提供了更广阔的发展空间，使他们能够开发出规模更大、功能更复杂的软件产品，这往往需要编写数万行甚至更多的源代码。传统的基于控制流的软件开发技术在处理大规模、复杂软件产品的问题时显得力不从心。因此，迫切地需要寻求更为高效的软件开发技术，来应对日益增长的软件规模和复杂性带来的种种挑战。

在开发程序的过程中，人们逐渐意识到，相比于其控制结构的设计，程序的核心数据结构设计显得更为重要。良好的数据结构设计能够显著提高程序的效率和可读性，从而优化整个软件系统的性能。遵循这一原则的设计技术被称为面向数据结构的设计技术。它强调以数据结构为核心，通过精心设计数据结构来优化程序性能并提升代码质量。在应用这些技术时，我们首先关注于设计程序的核心数据结构，然后根据这一数据结构来开展程序设计。

面向数据结构的设计技术中，JSP (Jackson Structured Programming)技术是一个典型案例，它为程序设计提供了新的视角和方法。在 JSP 方法中，首先利用顺序、选择和迭代符号来精心设计一个程序的数据结构。该方法的亮点在于能够从数据结构表示中巧妙地推导出程序结构，为开发者提供了一种新颖且高效的程序设计思路。随后，基于数据结构的设计技术得到了进一步发展，多种新方法应运而生。其中一些技术因其高效性和实用性而广受欢迎，并在实际项目中得到了广泛应用。由于篇幅限制，本文不会深入探讨论这些技术细节。随着技术的不断进步，这些方法在工业界的应用逐渐减少，并被更先进的技术所取代。

5. 面向数据流的设计

随着超大规模集成电路技术的革新及新结构概念的引入，计算机性能得到了显著提升。为了应对日益复杂和有挑战性的问题，开发更复杂、更先进的软件产品来满足日益增长的需求变得尤为重要。因此，软件工程师们一直在探索更高效的技术来优化软件产品的设计。很快，就有人提出了面向数据流的技术，这一创新性的方法引起了广泛关注。这些技术强调在设计系统时，应首先明确主要的数据项，然后确定对这些数据项的处理流程，以实现预期的输出。在这个过程中，函数(或称为过程)与数据项在不同的函数间进行交换，并通过数据流图(Data Flow Diagram，DFD)进行可视化表达。最终，根据数据流图的设计，构建出相应的程序结构。

DFD 已被广泛验证为一种通用性极强的技术，它不仅可以用于模拟软件系统，还适用于各种不同类型的系统模拟。例如，图 1-5 直观地展示了自动化汽车装配厂的数据流动情况。对于那些未曾亲身体验过自动化汽车装配厂的人而言，对其运作流程进行简要介绍是非常有必要的。在自动化汽车装配厂中，输送带(也称为装配线)沿侧设有多个处理站(也称为工作站)，每个工作站都有其特定任务，如加装轮子、装配发动机和喷漆等。随着部分组装完成的汽车沿着流水线移动，各个不同的工作站便在其上依次执行各自相应的工作任务。图 1-5 中的 DFD 模型将每个过程(以圆形表示)视作一个工作站，这些工作站接收特定的投入项目，并产生相应的输出项目。即使对 DFD 一无所知，理解如图 1-5 所示的汽车装配厂的 DFD 模型也是相当直观的。这正是 DFD 的一大优势——其简洁性使得用户能够轻松理解复杂的系统过程。一旦建立了一个问题的 DFD 模型，面向数据流的设计技术便能够提供一种有效的方法，直接根据 DFD 模型推导出相应的软件设计。

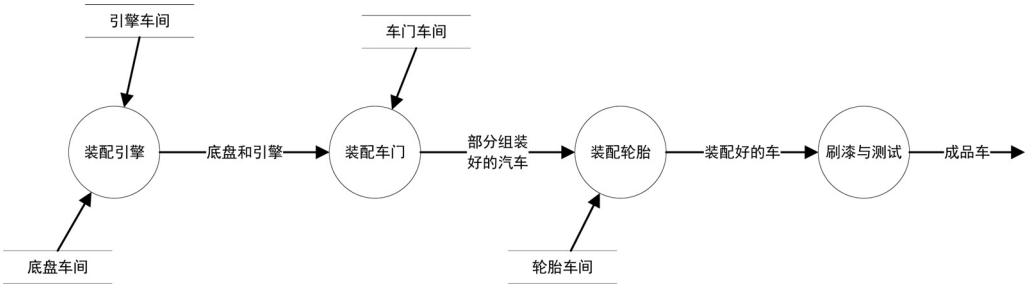

图 1-5　汽车装配厂的数据流模型

6. 面向对象的设计

随着技术的不断进步，面向数据流的技术逐渐演变为面向对象的设计技术。面向对象的程序设计技术提供了一种直观易懂的方法，其核心在于首先识别出一个问题中的自然对象，然后确定这些对象之间的组成、引用和继承等关系。每个对象本质上都是一个数据隐藏或数据抽象的实体。由于其自身的简洁性、代码和设计重用的广泛可能性、缩短的产品开发周期、降低的开发成本、增强的代码健壮性以及维护的便捷性等因素，面向对象技术已经得到了广泛的认可和应用。关于面向对象设计(OOD)技术的深入讨论将会在后续章节中展开。

7. 其他发展

在本节中，我们已经深入探讨并回顾了自早期编程时代以来，软件设计技术是如何逐步发展并经历深刻变化的。图 1-6 展示了软件设计技术的发展历程，总结了该领域的演进轨迹。回顾过去 40 年，软件设计技术飞速发展，并指明了未来发展的方向。除在软件设计技术方面取得显著进展外，还引入多个新的概念和技术，以确保软件开发的高效性和有效性，包括生命周期模型、规范技术、项目管理技术、软件测试技术、调试技术、质量保证技术及计算机辅助软件工程技术等。这些技术的综合应用为软件开发的整个过程提供了有力的支持和保障，极大地推动了软件工程学科的成熟与进步。在本书后面的章节中，我们将对这些技术进行深入讨论和探索，以掌握它们在软件工程实践中的应用和价值。

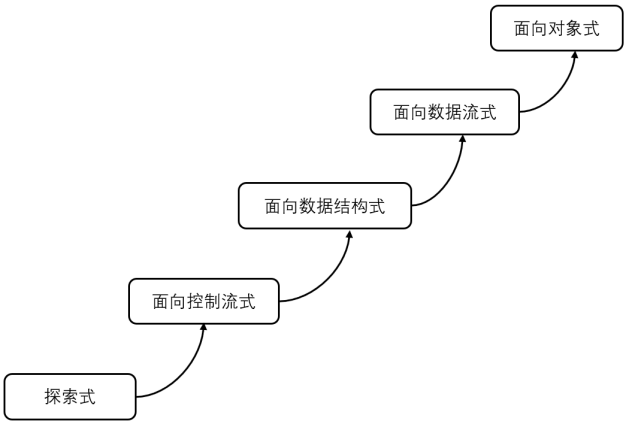

图 1-6　软件设计技术的发展历程

1.2.2 软件工程的基本原理

为了达到软件系统的开发目标，软件开发过程需要严格遵循软件工程的七大基本原理，以确保开发工作的规范性和高效性。这些原理包括：模块化、结构化、抽象化、可重用性、可维护性、可测试性及可靠性。这七大原理共同构成了软件工程的基础，它们对于提升软件开发的效率和品质起着至关重要的作用，确保项目能够顺利进行并达到预期目标。以下是这些原理的详细描述。

1. 模块化

模块化是软件工程中的关键步骤，其核心思想是将复杂的软件系统细分为一系列相互独立、功能明确的模块或组件。这些模块都拥有各自独特的接口和功能，使得它们可以独立开发、测试和维护，这种划分方法极大地优化了开发流程，为软件工程的顺利进行提供了有利的支撑。模块化提升了开发的并行性，使得多个团队或开发者能够同时处理不同的模块，从而显著加快了开发速度。此外，通过将系统细分为更小、更易于管理的模块，模块化简化了复杂问题，降低了开发难度。更重要的是，模块化促进了代码的复用。一旦某个模块经过验证并稳定运行，便可以被轻松地集合到其他项目中，几乎无需进行烦琐的修改或重构。这不仅显著减少了重复性劳动，还提高了程序开发效率，更有助于保障代码的质量和可靠性。这种可复用性的模块化为软件工程的长期发展奠定了坚实基础，并为未来的软件创新提供了有力支持。

2. 结构化

结构化是软件系统设计和实现的重要原则，它强调系统应该按照清晰、明确、有序的结构和规范进行组织。结构化的设计方法使得软件系统的各个组成部分之间的层次关系和交互方式更加明确，极大地增强了系统的可读性和可维护性。同时，结构化的编程方法要求程序员遵循一定的编程规范，确保代码逻辑清晰、结构紧凑，有助于减少错误和缺陷，提高代码的可靠性和稳定性，是软件开发中不可或缺的一环。此外，结构化思维在软件开发的完整流程中同样发挥着作用。从需求分析、构思设计、代码编写到测试验证，每一个环节都需要贯彻结构化的理念，以确保软件系统的整体协调性和内在一致性。这种全方位的结构化方法不仅提升了软件开发的效率和客观理性，还有效降低了项目面临的风险。

3. 抽象化

抽象化是软件工程中至关重要的过程，它涉及将系统中的复杂实体、行为和关系提炼为更为高级和简洁的概念和模型。这一过程犹如将纷繁复杂的现实世界简化为易于理解的地图，使我们能够专注于系统的核心特性和功能，而无须深究底层细节的复杂性。通过抽象化，我们不仅能屏蔽掉不必要的复杂性，简化复杂系统的设计和实现，还能提高开发效率，使开发者能够更快速地掌握系统的整体架构和运行逻辑。此外，抽象化还有助于增强代码的可读性，使得其他开发者能够更容易地理解和维护代码，从而促进团队协作和知识共享。在实际应用中，抽象

化表现为类、接口、函数等形式。这些抽象元素在构建模块化、结构化的软件系统中起到关键作用，通过将复杂的功能或数据结构进行抽象化，我们可以将其拆分成更小、更易于管理的模块，从而提升软件系统的可维护性和可扩展性。抽象化设计将系统的共性与个性部分进行分离，使系统在面对需求变更或功能扩展时更加灵活和便捷。因此，在软件设计和开发过程中，应当充分认识抽象化应用的重要性，通过合理的抽象设计，打造出更易于理解、高效、稳定且易于维护的软件系统，从而为用户提供更加优质、稳定的软件体验。

4. 可重用性

可重用性是软件工程中的关键特性，它指的是软件系统中的组件、模块或代码能够在不同的系统或项目中重复使用的能力。首先，通过重用已有的组件和代码，不仅可以显著减少开发工作量，提高开发效率，还降低了开发成本，提升项目的整体效益。开发团队可以避免重复开发，将更多精力聚焦于创新和解决新问题。其次，重用的组件和代码经过充分测试和验证，具有较高的可靠性。这意味着在重用这些组件和代码时，可以有效降低潜在风险，减少错误和缺陷的发生。这对于保证软件系统的质量和稳定性至关重要。此外，可重用性还促进了知识共享和团队之间的协同作业。通过重用已有的组件和代码，团队成员可以迅速了解并把握彼此的工作内容，减少了沟通障碍，增强团队协作的紧密度。同时，这种重用机制也推动了知识的传承与积累，为软件工程的持续进步和发展注入了新的活力。

因此，在软件设计和开发过程中，积极追求并实践可重用性至关重要，通过精心设计和构建合理的软件架构，打造出易于重用、易于维护的软件系统。可重用性不仅为软件系统的长期稳定运行和维护奠定了坚实基础，也推动了软件工程领域的不断发展和进步。

5. 可维护性

可维护性在软件工程中具有举足轻重的作用，是衡量软件系统在发布后是否便于修改、扩展和修复的重要指标。可维护性涵盖了代码的可读性、可理解性和可修改性等多个方面。具备良好可维护性的软件系统，其代码结构清晰、逻辑明确，使得开发人员能够轻松理解并快速定位问题。同时，这样的系统也更容易适应需求变化和技术更新，可以通过简单的修改或扩展来满足新的业务需求。提高软件系统的可维护性不仅能够降低维护成本，还能显著提高系统的可靠性和可用性。当系统出现问题时，维护人员能够迅速定位并修复故障，确保系统的稳定运行。此外，良好的可维护性对于软件系统的长远发展至关重要，它不仅能够延长软件系统的生命周期，还能显著减少因技术更新换代或需求变化所带来的系统重构风险。

因此，在软件设计和开发过程中，关注并致力于提升软件的可维护性具有重要意义。通过运用模块化、结构化等先进的设计原理，以及严格遵循良好的编码规范和标准，能够构建出易于维护的软件系统。这样的系统不仅具备更强的稳定性和可靠性，还能为软件的长期稳定运行和持续发展提供坚实的支撑和保障。

6. 可测试性

可测试性是指确保软件系统的代码和功能能够被有效且高效地测试和验证的能力。通过提高可测试性，能够在早期开发阶段发现并修复潜在问题，从而显著提升系统的质量和稳定性。具备良好可测试性的代码通常拥有清晰的模块划分和结构布局，这使得测试用例的编写和执行变得更加简便、直观。在实际开发中，重视可测试性对于降低后期维护难度、加速开发迭代和版本更新具有重要意义。通过引入自动化测试工具和方法，可以对软件系统进行全面的测试，确保每次代码变更都不会导致新的错误或问题产生。这种持续且自动化的测试流程极大地增强了软件系统的健壮性和可靠性，为用户提供了更加稳定、优质的软件体验。因此，在软件开发的每一个环节，都应该充分考虑到可测试性的重要性，并在设计和编码实施过程中予以充分保障。通过合理的架构设计和遵循严格的编码规范，确保软件系统的可测试性得到有效实现，从而为软件质量的不断提升奠定坚实的基础。

7. 可靠性

可靠性是软件系统的核心属性，反映了软件系统在特定环境下能够稳定运行并输出正确结果的能力。可靠性涵盖了系统的稳定性、容错性以及可恢复性等关键方面。稳定性确保了软件在长时间运行中不会出现无故崩溃或产生异常行为；容错性则使软件在面对异常情况或错误输入时能够保持正常运行，并尽可能减小错误对系统的影响；可恢复性则保证了软件在发生故障后能够迅速恢复到正常状态，避免数据丢失或业务中断。在软件开发过程中，通过采用合适的设计和实现方法可以显著提高软件系统的可靠性。这包括使用健壮的算法和数据结构，进行充分的测试和验证，以及实施有效的错误处理和恢复机制。同时，持续监控和收集软件的运行数据，能够帮助用户及时发现并解决潜在问题，减少系统故障和错误的发生，从而为用户提供更加稳定、可靠的软件服务。

1.3 本章小结

软件工程作为一门学科，其范畴广泛而深远。它不仅是收集几十年来优秀编程经验的结晶，更是研究者们为了以更低成本开发高质量软件而进行的不懈创新的体现。从最初的新手“探索式”编程风格，到如今的结构化编程、需求说明、测试和项目管理等技术的形成，软件工程始终在不断地发展完善中。

过去的近 50 年中，软件开发技术经历了翻天覆地的变化。这些变化不仅体现在技术革新上，更体现在对软件开发过程的深入理解和认知上。需求说明的精确化、测试的规范化、项目管理的科学化，都是推动软件工程学科得以发展的基石。同时，从过去的错误中吸取教训，不断修正和完善软件开发的方法论，也为软件工程学科注入了持续的创新活力。结构化编程是软件工程中的一个重要概念。通过将程序分解为多个模块，每个模块都能独立且正确地工作，极大地

提高了程序的可靠性和可维护性。同时，避免使用 GOTO 语句等可能导致程序流程混乱的指令，也是结构化编程的重要原则。计算机系统工程作为处理软硬件集成的系统开发学科，涵盖了软件工程的相关内容。这意味着在开发复杂的系统时，不仅需要关注软件的设计和实现，还需要考虑硬件的特性和限制，实现软硬件的协同工作。对于大型软件产品的开发来说，软件工程技术是不可或缺的。工程师团队需要协同工作，运用软件工程的原理和方法，确保软件的质量、进度和成本得到有效控制。即使是开发小型程序，软件工程的大多数原理依然具有显著的指导意义，能够帮助开发者更加高效地编写、测试和维护代码。然而，对于缺乏编程经验的学生来说，理解软件工程的原理和方法可能会存在一定的困难。因此，通过实践、学习和经验的积累，逐步加深对软件工程的理解和应用是非常重要的。在软件危机背景下，软件工程的出现和发展显得尤为重要。软件危机的成因复杂，包括需求不明确、开发过程不规范、缺乏有效的项目管理等。而消除软件危机的途径之一就是加强软件工程的理论研究和实践应用，提高软件开发的效率和质量。软件工程的基本原理包括模块化、结构化、抽象化、可重用性、可维护性、可测试性及可靠性等。这些原理为软件开发提供了指导，帮助开发者更好地应对开发过程中的各种挑战。

综上所述，软件工程是一门综合性、实践性很强的学科，其发展历程充满了挑战和机遇。通过不断地学习和实践，用户可以更深入地理解并应用软件工程的原理和方法，为开发高质量、高效率的软件产品做出贡献。

1.4 思考与练习

1. 什么是软件危机？
2. 软件危机有什么表现？
3. 软件危机产生的原因是什么？
4. 消除软件危机的途径是什么？
5. 软件开发的发展分为哪三个时代？
6. 软件工程的七条基本原理是什么？
7. 什么是软件工程？它是如何克服软件危机的？
8. 流程图是什么？流程图技术为什么对软件开发有用？
9. “结构化编程”是什么？PASCAL 和 C 等现代编程语言如何有助于编写结构化程序？与非结构化程序相比，结构化程序的优点在哪里？
10. 讨论面向对象设计方法对于面向数据流的设计方法的主要优势。
11. 简述软件工程在软件开发中的作用和意义。