微型计算机实用大

第14篇 计算机新技术

14.1 软件工程

14.1.1 软件与软件工程

软件工程 按照IEEE软件工程术语标准词汇的定义：软件是与计算机系统的操作有关的计算机程序、规程、规则以及可能有的文件及数据。软件工程是软件开发、运行、维护和退役的系统方法。它的主要目的是提高软件产品的质量、增加软件产品的数量、改善软件制造环境。

软件工程这一概念是把人们从一种自发、繁琐、重复的工作状况纳入一种规范化的生产管理方式中来。

软件工程是一项人力密集的产业，它需要技能和管理。它是一门集计算机科学、管理科学、经济学及所有有关问题求解工程的科学。软件工程包括了三项要素：方法、工具和规范。

随着软件工程的不断发展，也有许多新的概念产生。"软件工程经济学"是把微观经济学的概念与软件工程结合在一起，运用目标管理的方法来分析和控制软件工程的行为。"软件质量工程"则把软件质量的概念进行了延伸与强化。提到了工程的角度。随着软件事业的不断发展，软件工程的概念涉及的领域也越来越多了。

世界已经进入了一个以全范围应用计算机为标志的信息时代，以软件生产为代表的信息工业生产在工业生产中占有越来越重要的地位。软件工程的出现，一方面标识着人类对软件特性有了更成熟的认识，另一方面使人们能够更有效、更经济地开发软件产品，有力地促进了软件生产。

软件危机 是60年代提出的概念，直至今日还在被人延用。它旨在说明随着计算机事业的不断发展，硬件水平连续提高，硬件技术日趋成熟，而对软件的需求日显紧促，越来越满足不了人们的需要。这种技术与需求的差距一点点增大的趋势，就形成了在计算机系统中硬件成本逐渐下降、软件成本逐渐上升的发展状况。它表示：①人员不足；②软件制作费用高；③维护困难等问题越来越突出。

软件危机这个概念一经提出，人们就没能彻底摆脱它。软件产业始终在为提高软件的工作水平而努力。

从某种角度讲，软件的危机是一种人的危机。它是对人的智力的一次挑战。它是对人终究能否真正驾驭机器的考验。

软件危机实际上也是质量危机。它反映了软件企业的生成效率和软件质量还达不到应有的水平，因而需要人员水平和工具水平的提高。

正是为了克服这一危机，人们提出了软件工程这一概念，所以说，它有力地促进了软件工程理论和实践的发展。

软件生存周期 过去人们往往将软件开发理解为编制程序。随着计算机应用越来越广泛和深入，软件越来越复杂，人们清楚地认识到软件产品和其它工业产品一样，必须经过设计、测试、检查等。根据这一概念，逐步形成了"软件生存期"的概念，即它是软件产品从形成开始，经过开发、使用和不断增订修改，直到最后被淘汰

的整个过程。

软件工程方法提出了技术上构造一个软件的途径。它包含一系列的工作：项目规划与估计、系统与软件的需求分析、软件设计(包括：数据结构设计、程序结构设计和算法过程设计)、编码、测试和维护。这些过程被概括为软件的生存周期。从生存周期各个阶段来看，所用的方法有系统分析方法、设计方法、编程方法、调试方法等。随着软件工程的发展，软件开发过程模式也在不断更改，但是这种典型的生存周期图的基本思想作为软件工程的过程模型依然被广泛采用。

软件工具 是这样一类程序，它可以用来帮助开发、测试、分析、维护其它计算机程序及其文档资料，实现软件生产过程自动化并提高软件的劳动生产率和可靠性。今天，软件工具已经为所有软件工程方法提供了前所未有的支持。一个新的学科分支－计算机辅助软件工程(CASE)也已经形成。CASE 建立了一种新的集成的软件工程环境，对软件在生存周期中各个步骤提供了连接和支持。过去，人们认为，也许程序设计自身恰恰是自动化的最后难点。软件工程尤其是CASE 的出现，改变了这一局面。

软件标准 力图使软件生产过程中作业标准化，确定作业和文档的格式，归纳出软件的开发准则和制度。不同的国家、不同的企业通常都有自己的规范，以保证软件产品具有可靠性、可维护性及交互性。

软件标准可分为产品标准和过程标准。产品标准是对产品所应有的特性的规定，如：程序设计语言标准。过程标准是对软件生产过程行为的规定。可以说软件标准囊括了最好的，或至少是最合适的实践方法。它们都是经过实际检验的有效的行为方法。同时，制定标准总是在某一组织范围内进行的，它们也是具有普遍性即要求组织内的所有人都采用的方法。

一些生产标准，如：程序风格和形式、文档设计、文档结构等运作起来确实是沉闷乏味的工作。一些标准的制定也有很多有些脱离实际。为避免这类问题，应采取下列步骤：

(1)制定软件产品开发标准要有软件工程师参加。它们应理解标准制定中内含的动机同时也应让大家理解。

(2)适时评审和修改标准以适应技术发展和研究环境的变化。

(3)尽力提供软件工具来支持标准化开发活动。

研制软件工程项目标准是一项困难多、工作量大的工作。大的国家和国际标准化机构有：ISO，ANSI，NATO，US DoD，CCITT 和 IEEE 等。其中如 ISO(国际标准化组织)是世界性标准化组织。它的 OSI(开放系统互连)标准在计算机界有着广泛和重大的影响。这些标准指导着人们生产和研究的实践。我国的标准化活动也在不断地进行和深化。

软件工程环境 软件工程环境是指软件开发、维护和管理人员在进行软件开发、维护和管理时所涉及的整个工程化的工作环境。它主要包括人员、计算机系统、工具和开发设施。

软件工程环境的主要目的是把环境中的诸要素有机地组织起来，以软件生存期中的各阶段为线索，把现有的软件技术集成在一起，形成一个连续的、转换自然的软件开发和维护的环境。

14.1.2 软件工程的需求分析

项目规划 制造产品必须先了解产品，有一个合理的规划。软件项目规划主要完成分析和估算工作。这些工作包括：

(1)确定软件的应用范围。根据功能、性能、接口、可靠性来制定出系统规范。

(2)对设备(环境)、人力、时间要作出基本估计。人员包括软件开发各阶段、各层次的人员。硬件资源、软件资源也要有完整的预算，然后做出一个合理的安排。

(3)对软件产品和质量的度量。开销、工时、工作量(代码量)、进度、错误率都要有个衡量。

尤其是在没有系统规范说明的情况下，计划制定者起到系统分析员的作用，以确定将影响估算任务的软

件功能、性能、接口和可靠性等。

项目规划使管理人员对项目的人、财、力能有一个合理的估计。以此来对项目进度作出规划、管理。这里也包含着不确定性和风险性。因而需要有比较完备的计划来应付不测。

软件成本估算 软件成本在计算机系统中的成本估算中占有越来越大的比重。因此，软件成本估算在计算机行业中(特别是在软件产业中)对项目成败已起到很大作用。这一问题已引起管理者们的广泛重视。

但是，软件成本估算实际上是一项非常复杂的工作，很多时候凭借的还是经验，因而它是一项风险性很大的工作。

软件产品的成本，一方面要考虑设备的折旧，因为计算机是更新率很高的设备，另一方面，最主要的是由于人的脑力和体力消耗。这种耗费是很难直接估算的。

一般来讲，软件成本估算至少要考虑两个量：软件的源代码行数(即软件的规模)和软件的生成率(即软件的单位成本)。

软件成本估算的七个基本步骤是：①建立目标；②对所需的数据和资源进行规划；③确定软件需求；④尽可能拟定所有可行的细节；⑤运用多种独立的技术和原始资料；⑥比较并迭代各估算值⑦事后跟踪；

软件成本估算也有很多方法。它们包括：①算法模型；②专家判定；③类比；④成本估算与现有资源相应；⑤为获胜而制定价格；⑥自顶向下法；⑦自底向上法。这些方法各有千秋，应适时、适地而用。

进度安排 软件开发进度的安排也就是对软件开发所需时间的估算。通常是用时间表来表示(如 Cantt 图等)。

进度安排首先是把一个复杂的工程项目分解成一系列比较容易管理的作业，然后安排这些作业的顺序。软件开发进度时刻表上要指明完成每项任务的具体时刻、任务完成的标志、各项任务所需时间长短。

软件项目的进度安排做法上与其它行业差不多。最常用的也是：PERT(计划评审技术)和CPM(关键路径)技术。这两种方法本质上是相同的。它们都使软件计划者能够：

(1)确定关键路径，即找到决定项目工期的作业链；

(2)应用统计模型对每个作业估计最可能的时间花费；

(3)决定最迟开始时间及闲散时间，并根据实际要求和可能压缩整个期限。

软件工程经济学 最主要的目的是把如何把经济学中的微观经济分析用于软件工程的决策。它是把软件工程与微观经济学结合在一起的一门科学。

随着软件工程不断发展，人们越来越意识到在软件的制作与购买过程中、性能与成本及利润之间的关系分析中、在软件成本与管理工作之间的关系中越来越需要比较严格的、摆脱单纯依靠经验的工作方法。这样软件工程经济学就应运而生了。它的主要分析方法包括：成本效益分析、多目标决策分析和不确定性(风险性)及其它统计分析。它们被广泛用于信息处理领域中，以支持软件开发、信息获取和维护工作中各种主要与次要的决策问题。因此，软件管理者和软件工程师掌握这些分析技术是非常有用的。

软件需求 是系统的特征或者说是对系统能力的描述。需求的概念可从两方面来考虑：定义和规范。定义是用户对系统所期望的活动；规范是对系统特征的技术描述。需求定义描述了系统与环境的交互关系。它包括：

(1)物理环境：设备、位置、环境状况、地理资源；

(2)接口：输入、输出的要求、方式、格式、媒体；

(3)用户特征：使用者、方式、理解力、熟练程度；

(4)功能：能力、限制；

(5)文档；

(6)数据：格式、精度、使用者、使用频度、保护方式；

(7)安全性；

(8)质量保证：故障率、恢复时间、效率度量、可更改性。

需求不仅描述了系统的信息流、数据变换，还对系统的性能做出了规定。因而需求可用于三个目的。首

先，开发者用它们来解释用户对系统的要求；同时，它们规定了设计系统所应有的特征和功能；另外，还提出了对测试组所递交的程序的要求。特别是，性能特征描述了对量化的指标要求。

有效的需求检验准则是：①正确性；②一致性；③完整性；④可行性；⑤必要性；⑥可证明性；⑦可跟踪性。

软件需求分析 是从系统规划到软件设计过渡的桥梁。它为设计阶段提供了软件需求规格说明书。需求分析细化了软件的功用、给软件设计者提供了信息和功能的具体表示、并提供了质量保证的手段。

它的工作过程是：依据在软件计划阶段确定的软件作用范围，进一步对目标对象和环境作细致深入的调查，了解现实的各种可能解法，加以分析评价，作出抉择，确定配置及划分出各个软件元素，建立一个目标系统的逻辑模型并写出软件规格。

软件需求分析的主要工作分为4个方面：1)问题研究。从系统规范和项目计划中找到基本问题要素。2)细化软件功能，分析信息流向、结构，明确系统设计要求；3)用规范的形式来表示整个软件需求；4)审查以消除错误。

需求分析是软件工程过程的第一步。正是在此处，软件项目的笼统的概框变成了具体的规范。需求分析集中在信息和功能范围内。为更好地理解需求，工程问题从逻辑和物理上被分解和表示。这种人为的划分和细化一直进行到描述出问题的各个层次为止。它侧重于软件要完成的是什么而不必去考虑具体完成过程。所以它是一个基本模型和具体实现的中间媒体。

需求分析方法 同软件设计方法一样主要也有3类：面向数据流的方法、面向数据结构的方法和面向对象的方法。面向对象的程序设计在14.2中有详细的论述，但面向对象的研究方法在软件需求阶段用于问题的定义和划分也是同样值得强调的。

在这里，对象可以看成是信息项，而操作可看成是用于一个或多个对象的过程或函数。面向对象的分析方法提供了一种简单而又强有力的机制来识别对象和操作。这个过程大致如下：

(1)首先用(自然)语言把问题的解决过程力求一致地描述出来。

(2)把描述中的每一名词和名词短语列一清单(标出那些同意词)作为对象。如果这一对象与实现相关则作为解空间的一部分；如果只是用于描述这个解则作为问题空间的一部分。

(3)把所有形容词作为对象的属性标识出来并把它们与相应的对象对应起来。

(4)操作是由动词和动词短语来确定。把每一操作归属于合适的对象。

(5)把所有的副词作为操作的属性标识出来并把它们与相应的对象对应起来。列整个清单时要把解空间与问题空间分开。

这种简单的办法就把信息域中的关键项和功能域中的关键项表示出来了。这种分析步骤可以重复地做，因而表中的这些项和功能也就可以随之进一步地被划分。更详细和进一步的过程处理可以参见14.2章。

规范说明 规范可以看成是对过程的描述。它对软件实现是否成功起到重要作用。在书写过程中应保证：①将功能与实现分开；②需要一种面向过程的系统描述语言；③规范必须指明软件所属的系统；④规范必须包括系统所操作的环境；⑤系统规范必须是一可认知的模式；⑥规范必须是可操作的；⑦系统规范必须能处理不完整性及可扩充性；⑧规范必须局部化及具有松耦合性。

软件需求可以以多种形式来描述。因而也需要适当的原则来约束。这包括：①表示格式和内容必须与问题相关。②规范内的信息必须是嵌套的。③表述格式必须以编号限制，且与随后的使用一致。④表示必须是可更新的。

世界上许多标准化组织对软件需求规范做了大量的分析和研究工作。IEEE，NIST和US DoD都对此做了一些规定。下面给出规范的轮廓：

1. 引言
   - a. 系统参考
   - b. 商业目标
   - c. 软件项目约束

2. 信息描述
   a. 信息流表示
   b. 信息内容表示
   c. 信息结构表示
   d. 系统接口描述
3. 功能描述
   a. 功能划分
   b. 功能描述
      (1)过程说明
      (2)约束/限制
      (3)性能需求
      (4)设计约束
4. 验证准则
   a. 性能限
   b. 测试类
   c. 期望的软件应答
   d. 特殊的考虑
5. 文献
6. 附录

软件需求规范是作为分析过程来进行的，评审在这里起到关键作用，它用来保证开发者和用户对系统有同一的理解。但最难解决的还是用户对问题的描述经常是变化的。

原型 Prototype(快速原型) 原型是开发者为开发的软件创建的一个模型。它作为定义需求的机制，建立了一个初级的系统。快速原型方法是用来建立系统需求的有效工具。

所有的软件工程项目都是从用户的要求开始的。这种需求可以以下列之一种形式出现：①描述问题的备忘录；②定义了一组行为目标集的报告；③外界提出的形式化要求；④作为大的计算机系统中一部分确定了功能和性能的软件的系统规范。这些需求要按下面步骤来构造软件原型：

(1)评价软件需求来确定开发的软件对原型是否是合适的。对那些特别复杂的功能应取掉或重新划分。

(2)对已简化的需求表示进行分析。把问题划分在合适的信息域和功能域中。

(3)评审那些需求表示后，写出原型的规范。

(4)创建原型并进行测试和细化。

(5)测试原型后，把它交给用户了解、评价并提出修改建议。

(6)重复(4)至(5)步直到所有的需求都形式化或原型已转化成系统产品。

在软件生命周期中的需求分析和定义阶段使用原型系统的好处有：

(1)可以通过演示来发现软件开发者和用户之间的理解偏差；

(2)可以找到被忽略的用户需求；

(3)可以发现并改进难以使用或混淆的用户服务；

(4)软件开发人员在研制原型中易找出不完备或不一致的要求；

(5)原型系统可迅速演示说明应用的可行性和它的益处；

(6)原型可用来作为书写高质量产品化系统规范的基础。

通常，一个原型用来勾画出系统的规范并进行实验、修改直到客户对功能表示满意为止。这样，人们还可以从原型中得到系统的规范并把它们在最后的产品版本中重新实现。

当然，有时一个实现的原型本身可能就被当做系统的规范。用户可能会说："这就是我要的"。但这样做是

危险的。这是因为：

(1)原型系统实现中会丢掉许多系统的重要特征。系统的一部分要求可能根本无法在原型中实现；

(2)原型很难成为一项合同的所有依据；

(3)系统的许多非功能要求如：可靠性、安全性难以在原型中表达；

(4)原型系统和具体实现有可能在操作系统、实现方式、使用的机器上有差异。因此在性能要求方面也不一致。

反对采用原型方法的主要说法是研制原型要占用整个系统开销的很大一部分。但是，使用原型系统的主要原因恰恰就在于它能节省在系统实现后返工、修改和维护的费用。从长远看，它节省了开发费用。

快速原型技术 如果原型的实现采用同最终产品相同的工具和标准的话，软件原型的开销就会很昂贵。但原型主要是为了演示系统的功能部分，所以它同最终产品相比可以只需要很少的开销。下列方法可用来减少系统原型的开发时间：

(1)使用高级语言来实现；

(2)去掉那些非功能化需求。如速度、空间等；

(3)忽略掉错误处理；

(4)减少对程序可靠性和质量的要求。

曾有一种看法是，原型的用户界面可比实际生成的系统简单。现在人们已认识到用户界面是系统重要的一部分。

由于错误检测和恢复在多数系统中占有很大比重，因此它们的减少使整个系统的开销大为减轻。这样就使原型系统的开发时间比最终系统的开发时间要少得多。同时，减少系统的规模和标准，软件原型可以用任何程序语言来构造，也使原型的开发时间减少许多。这种减少是以性能的丧失为代价的。

形式化规范 从软件生存周期来看，软件设计可看成是对需求定义和需求规范的细化。因而，规范被划分成了从需求定义－＞需求规范－＞结构设计－＞精确规范的过程。设计与规范必须反复多次修改，以便更深入地细化。这样形式化规范就成为最有效的手段来支持详细分析。

形式化软件规范是用某种语言来描述的规范。它的词汇、语法和语义都是形式化定义的。这种精确定义的规范有许多优点：

(1)这种精确定义使人们能更好地理解软件规范和设计；

(2)人们可以利用形式化系统规范来证明程序与规范的一致性；

(3)形式规范可以自动来处理。因而可以使用软件工具来开发、理解和调试；

(4)有些形式化规范语言甚至可用来生成原型系统；

(5)作为一种数学实体，它可用数学方法来研究；

(6)测试者可从形式规范中直接生成测试实例。

但是，形式化描述技术并未得到广泛的使用。这主要原因是：

(1)软件管理者因循守旧、误解或有些迟疑。开发形式化系统规范相对来讲开销较大，还无法一下子说服管理者相信它可大大减少整个系统的开销；

(2)大多数软件工程师未受到足够的训练来开发形式化软件规范；

(3)现有的一些形式化规范系统描述有些问题还不很方便；

现在还只有少数系统在开发中使用形式化描述技术。随着形式化规范的概念被越来越多的人理解和接受、越来越多的工具来支持人们使用，形式化规范势将被更多的人所掌握、喜爱。

14.1.3 软件设计技术

软件设计的基本概念 软件设计的概念几十年来有了很大的发展。它们为软件开发者提供了软件设计方法学的基础，它们主要帮助软件工程师了解：①把软件划分成各个不同成份的原则是什么；②如何把软件的概念化表示与功能和数据结构分离；③软件设计的质量要求是否是前后一致的。

这些概念包括：

(1)细化。逐步求精技术是一种自顶向下策略。在每一步中，程序的指令被更详细的指令所细化，直到整个程序完全用程序设计语言来实现。

(2)程序结构。程序结构表示了程序模块(各个成份)的层次关系。有许多标记方法来用于程序的结构。其中最典型的方法是树表示方法。其它的表示如 Warnier－Orr 和 Jackson 图也是非常有效的方法。

(3)数据结构。它表示了不同数据成份的逻辑关系。因为信息结构必将影响最终的过程设计，所以数据结构也是软件构造的重要表示形式。

(4)软件过程。程序结构定义了控制结构但它并未考虑程序的处理次序。程序过程主要用于处理每个不同模块的处理细节。过程要提供处理的精确描述。

(5)模块化。所谓模块就是程序对象的名称的集合。模块化就是把程序划分成若干个模块，每个模块完成一个子功能，它们构成的整体用来满足问题的需求。采用模块化原理可以使软件结构清晰，容易设计、实现和理解。

(6)抽象。抽象使得人们在某种级别上关注问题而不用关心低层的细节。在软件工程过程中的每一步都是对软件解法的抽象层次的细化。这种自顶向下由抽象到具体的方式简化了软件的设计和实现，提高了软件的可理解性和可测试性，并且使软件更容易维护。过程抽象、数据抽象和操作抽象并都在软件设计方法之中普遍运用。

(7)信息隐蔽。它的含义是对模块的描述和操作应保证在模块内的信息(过程和数据)不应被不相干的其它模块所访问。信息隐蔽保证了模块的独立性、局部性。尤其对调试、测试程序很有好处。

模块独立 是指每个模块完成一个相对独立的特定子功能，并且和其它模块之间的接口很简单。模块独立的概念是模块化、抽象、信息隐蔽和局部化概念的直接结果。模块的独立程度可以由内聚和耦合来度量。耦合衡量不同模块彼此间互相连接的紧密程度；内聚衡量一个模块内部各个元素彼此结合的紧密程度。

耦合分为数据耦合、环境耦合和内容耦合。在软件设计中应该追求尽可能松散耦合的系统。尤其是应坚决避免使用内容耦合。以保证信息隐蔽。

内聚和耦合是密切相关的。内聚大致可分为：偶然内聚、逻辑内聚、时间内聚、通信内聚、过程内聚、顺序内聚和功能内聚。设计时应该力求做到高内聚，从而获得较高的模块独立性。

面向数据流设计方法 每一种设计方法最终都是建立一个程序结构。面向数据流的设计方法定义了一些不同的映射把信息流转换成程序结构。这种方法是从基本系统模型入手，把信息表示成一种连续流。这种信息流经过一系列的转换完成了从输入流到输出流的转化。数据流图在这里用来作为刻划信息流的工具。

面向数据流的设计方法是基于模块化、自顶向下、结构程序设计等程序设计技术基础之上发展起来的，又称为结构化设计法。其特点是：面向软件体系结构是在比模块更高一级的层次上讨论结构问题，它是许多方法的综合应用。面向数据流的设计方法是已有应用中使用最广泛的方法。

面向数据流的设计方法首先要在需求分析阶段求出数据流图。数据流是由软件规格说明提供的，按照数据变换性质它们可以分为两类：一类是“中心变换型”数据流图，另一类是“事务处理型”数据流图，不同的类型流图有不同的转换方法。

在设计阶段主要过程有：

(1)细化数据流图,如果不够详细,则应进一步求精。

(2)确定数据流图类型。按照数据变换性质数据流图可以分为两类:"变换"型数据流图,另一类是"事务处理"型数据流图,不同的类型有不同的转换方法。

(3)把数据流图转换成软件结构图,建立软件结构基本框架。

(4)进一步分解结构中的模块。

(5)用试探法细化得到的软件结构。

(6)开发接口描述和全局数据结构。

(7)评审,然后进入细节设计阶段。

面向数据结构设计方法 设计的基本过程是:①评价数据结构的特征;②把数据表示成元素的形式;③把数据结构映射成软件的控制层次图;④细化软件的层次;⑤最终导出软件的过程描述;这类方法最主要的有:Jackson 系统开发方法(JSD 方法),Warnier 图方法,即程序逻辑构造方法(LCP 方法)和 Warnier—Orr 方法,即数据结构系统开发方法(DSSD 方法)。

JSD 方法包括:①标识出实体和行为;②用 Jackson 图来表示影响每个实体的行为并按时间排好;③把实体和行为用过程模型来表示;④描述与活动所对应的功能;⑤描述时间对系统的限制;⑥用序列、条件和重复框架来把问题分解成层次结构。

LCP 设计方法的主要依据是 Warnier 图。

(1)它首先是用 Warnier 图描述输入数据和输入数据的结构。Warnier 规则强调信息的任何集合必须划分为子集合。它根据元素出现次数的附加说明来完成这种划分。此外有条件出现的数据,用"0"或"1"来表示。

(2)根据数据结构的 Warnier 图导出程序的层次处理系统;

(3)根据层次处理,变换成更常见的程序流程图;

(4)进一步具体设计给出伪码表示。

一般来讲,面向数据结构的方法要比面向数据流的方法复杂一些。但面向数据结构的方法有更强的软件设计能力。它们之间的共同点是:都是由信息驱动的,都是把信息转换成软件的结构表示,都是基于某种概念来进行。

但面向数据结构的设计未明确地使用数据流图,因而变换和事务处理的流分类与这种方法并不相干。特别是,面向数据结构的设计最终生成的是对软件的过程描述,因而它与模块这一概念没有直接的关系。面向数据结构的设计用层次图来表示信息结构,所以这一点在项目的需求分析阶段就应有所考虑。

面向对象设计作为第三种重要的设计方法近年来得到了越来越广泛的重视,这一方法借鉴于前两种方法对抽象、信息隐蔽和局部化的思想有了更深一步的处理方法并增添了许多重要的新的概念。

面向对象设计方法(OOD) 也是面向信息的设计方法。它与传统方法最显著的区别在于把数据与处理分离开来。在其演化过程中,OOD 方法已能把三类设计方法,即数据设计、结构设计和过程设计的各种元素综合起来,通过确认客体来建立数据抽象。通过定义操作来指定模块和建立软件的结构。通过开发使用这些客体的方法来描述软件的接口。

OOD 方法包括如下各步:

(1) 问题的定义。

(2) 开发出现实中软件实现的非形式化策略。

(3) 用下列各子步来形式化该策略:①确定客体及其属性。②确定可施加于客体的操作。③显示客体和操作的关系。建接口。④对数据和过程抽象提供实现的细节。

(4) 反复应用(2),(3),步直到得到完备的设计。

14.1.4 软件质量工程

软件质量 软件质量可从4个角度来考虑：

(1)从需求规范等内部属性方面来看，软件质量为软件产品能满足给定需要的性质和特性的总和。

(2)从顾客角度来看，软件质量为软件满足其综合期望的程度。

(3)从外部属性来看，软件质量为软件具有所期望的各种属性的组合的程度。

(4)从软件自身来看，则为软件在使用中将满足顾客预期要求的程度。

一般来讲，软件质量的各方面特性主要从软件的可靠性和可维护性这二方面来衡量。一种有代表性的定义包括了：正确性、可靠性、效率、完备性、实用性、可维护性、易测试性、灵活性、可移植性、可重复使用性、连接性。

软件危机在很大程度上是质量危机。因此，软件质量工程学将软件工程所研究的质量、工期、开销这些质量要素，通过一系列观测数据逐步用一些模型来描述和控制。以质量管理为核心，来研究软件生成。

软件质量保证(SQA) 是为使软件产品符合已建立的技术需求提供足够的可信度，而必须采取的有计划和系统的全部活动的模式。软件质量保证与软件生命周期中每一阶段的证明与给验证活动关系很大。在许多组织中，未对这些活动予以区别。但是，质量保证和证明与验证实际上是不同的活动。在软件开发过程中，质量保证为管理功能，而证明与验证属于技术上的一部分。

在一个单位里，质量保证工作是由独立的软件质量保证部门来负责的。它直接向在项目负责人之上的管理人员报告情况。质量保证部门并不与哪一个专门的开发组相关，它要对单位的所有项目组的质量保证工作负责。

有必要进一步来区分的是，验证和证明只关心发现错误，而质量保证还要考虑诸如软件的维护性、可移植性和可读性等。这里给出的质量保证定义是：

软件质量保证是由过程、技术和工具构成。专家们以此来保证产品满足或超过在产品开发周期中所采纳的标准；或者是达到了工业或商业可接受的要求。

尽管质量保证关心的只是最终的产品，但软件的生产过程也必将影响制造出的软件质量。人们已经认识到，计划周密、管理完善的生产过程才能生产出高质量的产品来。

这一思想是从制造工业中得来的。但在软件生产中，生产的软件产品总是具有独有的特性不具备批量性，因而过程与产品的关系更加微妙复杂。

软件质量管理 软件质量管理包括了三方面：①软件质量管理的目标；②为达到这一目标而进行的有计划、有系统的活动；③维护软件质量的措施、方法。质量管理与质量保证含义不同，质量管理包括质量保证，它是一个更广泛更综合的概念。

由于软件产品的复杂性，软件质量管理变得比通常的质量管理更为复杂。差异性更大。这表现在：

(1)软件质量的度量比较复杂，因而规定质量需求、通过质量度量进行监督、跟踪和控制就有一些偏差；

(2)软件设计评审无法完全定量化，因而不论是评价与检查都有不完全性。

软件质量管理活动大致可分为质量控制和质量设计。质量控制是通过计划、规程和产品评价来完成对生产过程的管理工作。质量设计则是在一开始设计软件时就考虑软件质量指标。软件质量指标是通过软件开发组织在软件设计、开发、测试和维护中所需的质量政策和规程来实现的。

软件可靠性 可靠性－软件可靠性是软件最重要的特性。质量保证的最主要工作就是要保证软件产品的可靠性。

软件可靠性的衡量一般是采用统计方式。所以常用的定义是：规定时间内正常使用系统时的故障率；或者说，在规定的时间周期内在所述条件下执行所要求的功能的程序的能力。这一概念实际是一种工程概念。

它与程序错误的概念是不相同的。

根据软件测试理论，“enough is not enough”(充分性永远是不够的)。程序的复杂性决定了我们不可能进行穷尽测试，来发现程序中的错误。所以需要对软件的可靠性作出度量。在这里，用户们更关心的是感觉到的可靠性而不是程序的可靠性。因此，质量保证过程强调产品操作的可靠性而不是整个软件的错误数目。一项研究表明，消除产品的60%的错误也许只对可靠性有了3%的改进。

软件复杂性 从广义来讲软件的复杂性是对软件的可理解性、结构性、简单性、清晰性和模块性的度量。它反映了：①理解程序的难度；②纠错、维护程序的难度；③向他人解释程序的难度；④按指定方法修改程序的难度；⑤根据设计文件编写程序的工作量；⑥执行程序时所需的资源。

软件复杂性是通过对结构复杂性、算法复杂性、数据结构复杂和文本复杂性的度来进行度量的。

软件复杂性度量可用于：①软件质量的属性评价；②软件的可视性管理；③软件开发成本、开发进度的预估等。

软件质量的度量 软件的复杂性使得人们注意到有必要从整体上评价软件的质量，指导软件的开发，在软件开发过程中进行软件质量控制。

为了在国际范围内广泛应用SQM(软件质量度量)技术，各标准化组织纷纷建立了关于软件质量评价的标准。按照ISO的定义，软件质量度量模型由三层组成。高层：软件质量需求评价准则；中层：软件质量设计评价准则；低层：软件质量度量评价准则。高层和中层是在国际范围内推广应用。而低层可由各使用单位视实际情况制定。

度量的最根本方法还是定量评价。只有能够定量评价，才能测定其特性，进行分析、评价，提出改进措施。

在软件的各种特性中，既有能够进行量的评价的特性、也有不能进行量的评价的特性。对于不能进行量的评价的特性，进行质的评价，即评价特性的“有或无”。对于能够进行量和评价的特性，也可以进行质的评价，即看出它是否越出了特定的基准。

质的评价的基本点有两个：①检查特性的“有或无”。②对特性的重要程度加权。极为重要的特性是必不可少的特性。有的特性，即使不是必不可少的，但若没有它，将对产品整体价值带来很大影响。把重要程度分级加权评价较为方便。

14.1.5 软件测试

程序正确性 从用户方面来看软件的正确性含义有两方面：

(1)软件符合规定的需求的程度。这表明了程序是否能正确地执行、全面地完成整个过程。

(2)软件满足用户期望的程度。即程序所表现的功能是否为用户所满意。但在测试过程中，程序的正确性主要是从程序员方面来考虑的，即主要是检查软件有无设计缺陷和编码错误。

软件规范包括了对程序输入数据的描述。这些数据被称为程序的“域”，它经常用D来表示。规范也提供了对程序在D域内的预期行为的描述，用f(d)来表示输入数据d的期望行为。实际上，f(d)可能非常复杂。

类似地，一个程序P可以用一些计算活动来表示，当输入数据已提供给程序时，这些活动就会出现。虽然这些活动可能相当复杂，但可以用P*来简单地表示程序P的行为。因此，p*(d)是程序P在数据项d的理想化的行为。

简写的f(D)和p*(D)分别用来表示所有输入的期望行为和P对所有输入的反应。

只有在f(D)=p*(D)时，即P的行为在所有的输入都符合预期的行为，才能说程序P“正确”遵守规范f。

这样，在数学理论的实际应用中就产生了这样一个问题：是否可能确定在一个域中对某些特定的数值d存在(或不存在)f(d)=p*(d)。如果规范文本是完全抽象化的，它就能回答这个问题。但规范文本是很难完全形式化的(即使形式化了，f(d)也难以确定)。当规范不是形式化时，f(d)就可能要从人工计算、需求文本或对

应用的模拟估计中来得到。

程序测试理论经常忽略这些问题。假定存在着神谕(oracle)。它能判定对任意的d，是否有f(d)=p°(d)。这个神谕的理想化是软件测试的精髓。不同的测试策略以不同的方式来处理这个神谕问题。例如：在某种策略中，要求规范是正规的。这样，规范文件必须提供计算f(d)的方法。在某些情况下，这是靠规范是可执行的来保证。另一方面，其它一些策略没做出任何关于神灵是如何干预的假定。这些策略简单地用p°(d)来表示测试规范。而对p°(d)=f(d)的确定就留给神谕来解决了。

可靠性与有效性 正确性问题是要确定p°(D)是否等于f(D)。在程序测试中，确定过程是基于对测试数据d0,d1,…,dn 的有限的程序执行。如果：

$$f(d0) = p°(d0)$$
$$f(d1) = p°(d1)$$
$$\cdots$$
$$f(dn) = p°(dn)$$

那么，一般就能得出结论p°(D)=f(D)。

显然，如果不对测试数据做出某些限制，这个过程是不可行的。例如：如果D是一个无限的，P°可能只是在f 集上假装"垄断"所有的测试数据，而实际上在其它地方出错。

如果p(T)=f(T)=>P(D)=f(D)，则说这个测试数据T 对P 是可靠的。这就是说，只有当观察P 在测试数据执行的结果使人们能得出P 是正确的结论时，才能说这个测试数据是可靠的。

如何选择可靠的测试数据呢？令C 是对程序P 选择测试数据集的过程。为使C 选择可靠的测试数据，这个过程必须满足二个条件：C 必须是可靠的和有效的。C 是可靠的仅当，无论何时C 选择T1 和T2，P 要么是同时符合T1 和T2 的规范，要么T1 和T2 的规范对不符合。这就是说，p°(T1)=f(T1)当且仅当p°(T2)=f(T2)。一个选择过程被称为有效的仅当：如果P 是不正确的，那么，C 至少选择出一个测试数据使得P 未能满足它的规范。更精确地说，当(d∈D)(p°(d)≠f(d)) => (t∈T)(p°(t)≠f(t))时，则说C 是可靠的。

可靠和有效的选择过程实际上同程序的正确性是一致的。即：一个对P 选择测试数据T 的有效和可靠的测试数据选择过程C 使得p°(T)=f(T)当且仅当p°(D)=f(D)。为了说明这一点，首先假定P 是正确的，即假设p°(D)=f(D)。由于选择空的测试数据集是有效和可靠的，所以所要求的过程存在。相反的，假定C 存在着某种特性。先假设P 是不正确的，再说明它是正确的。即如果P 是不正确的，则对一些d，有p°(d)≠f(d)。因为C 是有效的，它将选择数据T，使得p°(T)≠f(T)。由于C 是可靠的，它将只选择使P 失败的测试数据。这同对C 的选择矛盾，所以p°(D)=f(D)。

值得注意的是：可靠和有效的选择过程能选择出可靠的测试数据集。即如果C 是可靠和有效的，那么由C 选择的任何测试数据集是可靠的。另一方面，如果T 是可靠的测试数据集、T 是从过程C 选择的，则C 是有效的。

这些定义有大量的概念问题。首先值得考虑的是，可靠和有效的选择过程是与程序正确性等价的。这样，如果P 是正确的，则证明某一C 是有效和正确地同证明程序是正确的是一样的。基于同样的考虑，如果P 已经是正确的，则有效和可靠的过程使测试集对P 是正确的并未有什么有效的证实因为P 在任何测试集上都能正常工作。最后，有效性和可靠性不是一个独立的概念(Weyuker 和 Ostrand 发现)，每一个测试过程要么是有效的，要么是可靠的。

在许多环境下，人们可能要寻找测试数据的条件使得测试者得出正确的结果，而不必形式化地等价于正确性。如果一个测试数据集T 有p°(T)=f(T)，并且对所有Q 有Q°(D)≠f(D)，则有Q°(T)≠f(T)。换句话说，如果P 在T 的行为是正确的，而所有不正确的程序在T 的行为都是不正确的，则T 是充分的。这样自然可以得出：如果T 是充分的，那么它就是可靠的。另一方面，可靠的不一定就是充分的。因为如果P 是正确的，则任何数据集是可靠的。

不存在一种通用的有效和可靠的测试选择过程。用技术术语来说就是，没有一个有效和可靠的测试数据

过程是可计算的。这样测试研究的目标就只是把注意力放在有限的特定的错误集中。而这样的错误集是可计算的。例如，可以构造这样的测试数据，它对于程序A的一个集合是充分的，而对在A中的所有程序Q，有Q*(D)≠f(D)=>Q*(T)≠f(T)。如A也许表示为某种可能引入程序中的错误集。这样，一个充分的测试数据集的存在可证实P不存在A类错误。如果对于P在A类T是充分的，而T是可靠的并未能说明什么。假定T不是可靠的，则p*(D)≠f(D)。但对所有Q中的A，如果Q不是正确的，则Q*(T)≠f(T)。因为P*(T)=f(T)，P不可能在A中。

程序正确性证明 确定P*(D)是否等于f(D)的一种方法是证明这个等式成立。这个方法的基本策略是：如果P是正确的，那么经过对P和它的规范严格的数学化分析，证明了对D内的所有输入数据x(也就是说数据满足P的输入规范)，如果P在x上操作，则有P*(x)=f(x)(也就是说，P满足它的输出规范)。换句话说，如果P是不正确的，则要找到证实，使得错误能被发现。

证明一个程序正确同测试一个程序检验它的工作是不同的。正确性证明是一种演绎行为，而测试并不是。

在正确性证明中，人们要论述所有的输入满足程序输入的规范，而论述的结论，即P是正确的，在理论上是有效的。在测试中，人们观察程序的执行实例，从观测的角度判断P是正确的。然而，这种判断在理论上是无效的(除非这种观测是从一个可靠测试集中得出)。由于可靠测试集在实际中不一定能得到，因此，测试者也许对测试数据的选择就不那么有说服力。人们也许在"充分"测试集上观察程序的执行，在这种情况下，人们只能说P是正确的这一可能性很大。然而，即使程序的正确性证明可以对P是否正确给予明确的回答，但在实践中仍然很少应用。

可靠性统计模型 为定量化评价一个系统的行为，当直接测量已不可能时，就需要有一个可靠性模型。对这个模型的关键要求是运行时间和测试发现的错误数量。在软件可靠性分析中使用了许多可靠性度量的概念，其中最重要的是：失败率和可靠性函数。

一般来讲，软件的可靠性是由预先确定的总运行次数(N)和运行成功次数决定的。软件的可靠率可以用R=S/N来标记。它的失败率或不可靠率可用U=F/N来表示。在实时系统中，考虑到连续的数据流，失败间隔的平均时间(Mean Time Between Failure)表示成MTBF=t/F。失败率为u=1/MTBF。对失败率的测量就用于建立错误模型。

可靠性函数是另一个软件可靠性的度量标准。它应用了大量古典的可靠性统计理论于软件之中。从硬件方面的经验得到的启发是可靠性应定义成时间间隔[0,t]内系统的性能统计的概率。另一个要考虑的因素是硬件环境。一个常用的定义是：

软件的可靠性为软件系统在一个时间周期内软件没有错误的概率，运行该系统的机器是在设计限制范围内运行的。

这个定义的数学表达式是：

$$R(t) = \exp[\int Z(t)dt]$$

这里，Z(t)是错误的估计率。MTBF可以表示成：

$$MTBF = \int R(t)dt$$

如果没有Z(t)的值，这个模型并没有更多的意义。许多模型试图解决这个问题。对可靠性的测量的不同方式形成了二种类型模型：错误模型和可靠性模型。它们在项目的可靠性估计中也可以相互使用。

错误模型首先假设在集成测试之前程序中的错误总数(Et)已知，错误亦能在发现时立即得到纠正，它主要预测剩余的错误数目(Er)。这样在集成测试中在时间t内剩余的错误数目为Er(t)=Et(t)-Ec(t)。这里，Ec是已改正的错误数。

Weibult的可靠性模型是在指数可靠性模型(Shooman)的基础上提出的新的模型：

$$Z(t) = a.b.t^{b-1}$$

显然，错误率依赖于b的增长或减少。

当然，从实际中人们也还得到许多其它模型。但对所有的模型，即使有些实验数据符合这些模型的函数，下列的假设仍使它们对软件的错误的估计有问题：①初始的程序错误可以可靠地估计；②程序的长度在它的生命周期内是恒定的。

也有人对这些程序错误的估计结果是持有怀疑和否定态度的。从混沌学的角度出发，可以认为程序的错误一开始是处于一种混沌状态，因此：①程序的错误总数是未知的；②对程序错误的"改正"有可能带来更多的错误，即程序错误的总数是变化的；③错误在不同的程序"尺度"下有着某种比例关系；④错误在程序中是随机的，但它们是以聚群方式出现的。

不过，即使错误模型是错误的，概率统计分析对测试理论似乎还是合适的。因为它给出了比较的方法，对成功的测试给予了肯定。它是一个有研究希望的测试基本理论。

测试原则 实际上可以有不同的测试原则。从不同的角度要注意以下问题：

从人的心理考虑，测试原则应包括：①确定测试例样的预期输入；②程序员（程序设计机构）应尽量避免测试自己的程序；③仔细检查每个测试结果；④检查非法输入。

从程序的错误分布考虑可得出：①先前的测试例样应尽可能在后来继续使用；②在一段中发现了错误意味着这一段错误可能较多；③程序测试应在多"尺度"下进行，应贯穿于程序生命周期的始终；④测试不能保证程序无错。

从外部环境来看，应该注意的问题有：①不要做过头的测试；②要更多地注意测试工具的开发和使用；③提交来的文件经常是不充分的；④测试工作的结果依赖于人的素质；⑤程序的外部设置对测试结果有很大影响。

可以说，测试好一个程序是比设计出这个程序更大的挑战。

模块测试 模块测试是测试一个程序的各个逻辑单元的过程，然后再把各个单元测试集成起来来评价整个系统。在模块测试中主要考虑的问题是测试例样设计和对多个模块测试的协调。测试例样可以从规范或从对模块代码的分析中得到。对应着这两种方法的测试策略分别称为黑箱方法和白箱方法。黑箱测试，又称功能测试，并不考虑程序的结构和行为，它的目的只是找出输入、输出行为不符合程序的规范。这种方法是从程序的规范中构造它的测试数据。白箱测试是检验程序的结构并从程序逻辑中提取测试数据，按照准则确定程序逻辑的覆盖。一种准则是每个程序语句至少执行一次。这项准则是必要的但不够充分，有些错误并不能发现。介于黑箱测试与白箱测试方法之间是灰箱测试方法。不言而喻，这种方法是前两种方法的综合运用。对模块组合分析也有着两种途径：非增量方法和增量方法。自顶向上和自顶向下是两种增量测试方法。其中自顶向下测试始于程序的顶模块，然后逐步测试较低的级别。为了模拟待测试的子进程模块功能要求有哑模块存在又称做桩模块。自底向上测试策略是从程序的最低开始测试。这样在测试过程中就需要有专门的驱动模块来调用正在测试的模块。当较低的模块测试好后，就再进行较高层模块的测试。而完整的测试策略要求在整个研制阶段来测试软件。

模块测试包括对一个程序的各个逻辑单元的测试过程，然后再把各个单元测试集成起来来测试整个系统。模块测试的目的是确定模块是否符合它的规范。

完成一个测试要考虑两件事情：测试例样设计和对多个模块测试的协调。测试例样可以从规范或从对模块代码的分析中得到。对应着这两种方法的测试策略分别称为黑箱方法和白箱方法。对模块组合分析也有两种途径：非增量方法和增量方法。非增量测试方法独立各个模块然后不再测试就构成整个系统。增量方法还要把先前测试的模块再组合起来测试。增量测试能更早地发现错误。增量测试中的两种策略是自顶向上和自底向下方法。这两种策略假定模块的调用过程是一个有向的生命周期图。

系统测试 实际上是一系列不同的测试。它的目的是完整测试整个计算机系统。下面是对这些方法的举例：

(1)恢复测试。它的目的是测试计算机系统在指定的时间内从故障中恢复并重新执行的能力。在许多情况下,计算机系统要有容错能力。另外,一些系统要能在指定的周期内识别错误。所以,恢复测试致力于测试软件在什么方式下失败并观查恢复是否正确执行。

(2)安全性测试。它是检查系统是否有合适的安全机制来防护非法入侵。在安全性测试中,测试者的角色就是一个入侵者,它应尽力去做各种尝试来破坏安全、防护和保密机制。设计这样一些测试情况的方法是先研究类似的其它系统中出现过的一些安全性问题,然后生成一些测试例样,来证明被测系统也有类似的问题。

(3)强度测试。它是让程序在高负荷下运行以测试系统的真正承受能力。强度测试中有些是程序在实际使用中可能遇到的情况,也有一些情况是永远不会发生的。它的目的是检查系统在一些极端情况下的应付能力。

(4)性能测试。它是用来检验系统运行的性能,说明在一定工作负荷下的响应时间及处理速度等特性。它的目的是观查系统的实际性能是否满足需求要求。

(5)验收测试。主要目的是验证在类似实际运行环境的测试条件下,软件是否满足系统的验收标准。

(6)其它的测试还包括:存储量测试、配置测试、兼容性测试、可靠性测试、功能测试、可安装性测试、文档测试等等。所有这些测试应按要求正确执行。

软件测试技术 软件质量是软件研制过程中一个首要考虑的问题。估计软件质量的一个传统方法就是软件测试。

软件测试是在实验室条件下用一些有代表性的数据来运行软件以便对软件质量作出估计。一种测试方法是证明软件的所有路径都已成功地走过;另一种模拟方法是用所有可能的输入测试一个程序,看它是否生成正确的输出。由于这两种方法都行不通,就提出了更实用的方法。例如把程序的输入空间划分成路径域。这样,程序输入域的子集就在每个路径上执行。因此,就可以从这些域中抽取测试数据构造程序执行的测试例样。这样的技术有输入空间划分、符号测试、随机测试、代数化程序测试、基于语法的测试、和数据流测试等。

(1)静态分析技术。是用人工或自动方式不实际执行代码来分析需求文件、设计文件和代码。使用这种技术对数组引用、指针和其它动态结构有很多限制。实验表明:代码审查与代码预运行(walkthrough)这些静态分析技术是非常有效的。它能找出一般程序的30%～70%的逻辑设计和编码错误。

(2)符号执行测试 是把输入数据和程序变量用符号值来表示。符号值可能是基本符号或者是表达式。程序的可能执行是由执行树来刻划的。程序的执行状态构成了PC(路径条件),它把在相应路径执行时输入所要满足的条件都收集起来。符号测试的执行是由称为"符号计算器"的系统完成的。它的主要部件是符号翻译器和表达式简化器。符号测试在程序包括循环或数组变量时会出现一些问题。同时,对应于大程序的符号执行树也会有毛病。

(3)程序检测测试。在程序检测测试中,软件测试每一次都有着大量的活动。这些活动的出现都被记录下来。记录过程的种类依赖于测量的要求和执行活动的类型。Ramamoorthy 描述了最小插入记录到程序中的方法。这种方法也保证了检测程序的充分性。记录的活动也许是某些变量的范围、或某些语句执行的次数及某个语句的条件是否被触发。记录过程有时又称做"探查"、"监控"或"软件检验"。

(4)程序变异测试。程序变异是度量测试数据充分性的技术。测试的充分性指的是数据在程序测试中保证某种错误不会出现的能力。在变异测试中,测试数据用来测试程序和它的变体(使程序含有某种错误)。如果程序通过了某种变异测试,则程序是正确的或者说程序包含了不可查错误。严格来讲,程序变异不属于测试技术。它是产生测试数据充分性的机制。

(5)输入空间划分。程序中的路径是由一些可能的控制流构成。程序中的路径把程序的输入空间划分成路径域。这里程序的输入域的子集使得在每个路径都执行。每条路径对应着路径域和路径计算功能。路径计算功能是用沿着路径执行语句的方法计算输入域的功能。路径上的条件分枝决定了路径域的边界。分枝断言的符号计算可用来构造路径域并用输入变量值的方式确定路径。路径分析测试技术强调的是把程序输空间

划分成路径域。这样，程序就从这些域中选取测试数据执行测试例样。而程序划分分析是使用程序形式化规范来产生类似的划分。它可用来与程序生成的结果比较。

(6)代数化程序测试。在代数化程序测试中，测试是把程序的正确性看作程序的等价问题。但对两个用很强的高级语言写的程序来讲，它们是否等价就难以确定。因而就必须对语言作出限制。例如去掉分枝语句、只允许整数变量等等。这样，被测程序限制在某些类中，小的测试数据集就可以充分证明程序的等价性。这个方法中存在着两个问题。首先很难充分定义程序P'的基本类及测试实例T'的相关类。其次，是与假定程序员已经了解Q的属性有关。这些假定必须在使用这个方法来证明P的正确性之前做出它包括有关P'的属性Q是正常的；从T中产生的测试例样满足Q的计算属性。

(7)随机程序测试。随机测试是黑箱测试策略的重要方法。它是随机地抽取所有可能输入值的子集对程序进行测试。Duran和Ntafos收集的结果显示，随机测试对许多程序(包括实时软件)是有效的。程序可看做对错误的取样。即程序是在输入数据子集中执行。如果观察到程序不符合期望的行为，则认为出错。许多测试技术的目的是增加发现程序中存在的错误的样本值的可能性，但现在还没有一种技术能保证被测程序执行的正确性。随机选取或借助于抽样过程选择测试数据方法反映了输入序列中实际可能的分布。它使人们可估计出"操作的可靠性"。随机测试可以同端极/特别值测试一块使用。

(8)基于语法的测试。一些形式化规范系统，如：飞机订票系统、电话交换系统，可以用有限状态机模型来表示。这样，就可构造出由FSA可接受的正则文法的语言。测试策略可基于语法而产生系统的输入、输出。这种策略可扩展到测试使用上下文无关属性文法的一大类程序。

(9)数据流引导测试。数据流分析是用于优化编译分析程序的某种特性结构的技术。在数据流引导测试中，数据流分析用来从流图中抽取抽象的程序变量关系。它们主要有三种：块测试——它把单入单出块作为块变换的基本单元、树定义策略和数据空间测试。这些技术仅对建立数据变换有效。数据流分析是提取程序中结构化信息的技术。在这个方法中，程序被看成在程序的变量中建立了一种有意义的关系。这样，一个测试策略可以用一些或全部程序变量的数据变换的路径来定义。程序的控制流信息用来定义程序经历的路径集合。

(10)实时软件与测试。典型的实际软件测试类型是主计算机与目标计算机测试。大多数对主计算机测试使用技术同非实时应用的测试地样。在主机上对集成系统的测试需要运行一个环境模拟器，并适当控制外部进程。在端计算机测试中，先进行模块测试然后再进行系统集成测试，接着是整个系统测试。

软件测试工具 系统化测试的应用经常伴随着要求提供自动帮助。许多技术，诸如从带有注释的程序中研究证明条件或确定数据分析中变量的使用，需要大量的文字工作。这些工作用手工做起来令人乏味并易出错，所以最好有专门的程序来做。类似地，一些测试技术中测试例样的数量开销相当大，由程序员直接控制的大量测试例样的执行也是一项密集劳动。大多数情况下使用的方法是研制专门的软件来提取测试例样、启动程序的执行、记录测试的结果。当为使人工计算可行而把测试数据表示成程序部件的函数时，对测试数据自身的研究陷入了组合爆炸。在许多应用中，人工检查计算结果与期望值是否相等已不可行；输出文件可能大到已人工检查不了；预期的输出可能是从可执行规范独立执行得出；输出的正确性也许可用性能限制来确定；或者执行的数量非常庞大。所有这些情况都需要一个专门的程序来检查执行的结果自动确定对测试例样执行的正确性。测试文件、记录和文档的维护基于类似的原因也需要自动化。最后一点，集成系统也许需要模拟物理系统的软件。

一个真正的工具与专门用途的测试软件的区分必然是模糊的。下列是测试工具的表现特征：

(1)接口通用性。真正的工具应允许用户灵活说明测试要求和测试的程序。

(2)共享性。一个工具在它的公式中要有足够的通用性。这样，它可以在某个用户社团中共享。

(3)复用性。工具应能在一次使用中生存下来，即工具生存期应覆盖多个应用的生存期。

有效的测试工具的研究看起来是系统化测试方法成功应用的前提条件。这些工具的功能是尽可能按照它们相应的测试技术分类的。

按照测试工具执行中的分析方式，它们可分成两组：静态分析工具和动态分析工具。静态分析工具只对程序进行有限的分析。静态分析的重点在需求和设计文件及程序的结构外观。它判断的能力很有限。动态分析工具为待测试的程序直接运行测试提供了支持。工具提供的功能支持范围很大。运用程序检验技术提供的运行统计工具也已广泛使用。另外还有测试支持工具。一个实际的系统是多种分析工具的组合，它们并不直接属于哪一类。

14.1.6 软件维护与文档

软件维护 软件维护狭义地来看是指在软件产品交付之后对其进行修改，以改正缺陷。从更广的范围来看是指在软件产品交付之后对其进行修改，改进性能及其它属性，或使产品适应改变了的环境。

软件维护内容包括：

(1)校正性维护。是对软件错误的测试、查错、改正和重测试的过程。

(2)适应性维护。这是对软件系统适应新环境所做的更改。

(3)完善性维护。这是根据用户要求所做的改进、增添现有功能的工作。

(4)预防性维护。是指对软件所做的各种只带有预防性的活动。

软件维护的任务包括建立一个维护机构，描述维护报告和对维护进行评价，为每个维护请求规定响应的流程，保存和记录维护活动，并建立评审的评价标准。在维护请求发出之前，实际上已经开始了许多与维护有关的工作。

软件维护中的特点与问题 很长时间以来，软件维护一直是被人们所忽视的工作。但在一个完整的软件开发中，软件维护一直占有很大的比重。而且这个比重还要不断有所增加。比较保守的看法是它应占整个费用的50%，而许多统计表明它也许要占70%或者更多。

软件维护中碰到的大多数困难在于没有采用软件工程方法来管理和组织软件开发过程。这些问题包括：

(1)阅读和理解别人编写的尤其是没有说明文件的非结构化程序非常困难，清晰度和可读性都很差。

(2)由于人员变动或新软件开发需要，维护活动往往由没有参加该软件设计和开发的人员来执行因而缺少以往的经验。

(3)缺少适用的文档资料，或是文档资料与源程序代码不一致，可理解性差，因而无助于对程序结构、功能和接口性能的理解。

(4)绝大多数软件在开发设计时，没有考虑以后的修改，因而给修改带来了困难。所以任何一个小小的修改都可能孕育着很大的危险性。

软件的可维护性 是指软件能够被理解、校正、修改和完善，以适应新的环境和条件的难易程度。其中因素包括：①是否具有合格且训练有素的软件人员；②是否采用可理解的系统结构；③是否具有容易处理的系统；④是否采用标准化程序设计语言；⑤是否有标准化操作系统的支持；⑥是否采用标准化的结构文件；⑦测试例样是否有效；⑧是否有合适的纠错工具；⑨计算机本身的有效性等。

软件维护的工作量通常是用时间来度量的。

文档 是指对活动、需求、过程或结果进行描述、定义、规定、报告或确证的任何文字或图示信息。

所有大的软件系统、应用程序都有一套文档与之对应。文档可划分成用户文档和系统文档。用户文档描述了系统的功能，而系统文档则侧重于系统的设计、实现、测试。

文档是在系统生命周期中伴随着系统的各个阶段出现的。所有的文档需要有有效的目录。目录的好坏直接使一个文档大大地增辉或减色。

用户文档 文档实际上是系统用户与系统打交道中最先接触的东西。用户文档不必描述如何构造系统，也不应用华丽词藻或不实之词来渲染系统的功能。用户文档应该按适合于用户要求的程度来构造。