中国系统分析员
www.csai.cn

软件工程知识体系指南（2004 版）

蒋遂平 翻译

蒋遂平，计算机应用专业博士，国家系统分析员，CSAI 专业顾问。曾从事过数据库、虚拟现实和人脸识别等方面的研究工作，先后参与和主持了多个系统的软件开发，主要感兴趣的领域包括软件工程，图象处理和数据库。

Guide to the Software Engineering Body of Knowledge

2004 Version

软件工程知识体系指南是IEEE计算机学会（IEEE Computer Society）职业实践委员会（Professional Practices Committee）主持的一个项目。®SWEBOK是IEEE的官方服务标记。
http://www.swebok.org

目录

//

第一章　指南简介

尽管全世界有数百万软件开发人员，软件在我们的社会中无处不在，软件工程在最近才达到了合理的工程学科和被认可的职业的状态。

一个职业在核心知识体系上达成一致，是所有学科的关键里程碑，IEEE计算机学会认为这是软件工程向职业状态演化的关键。本指南是在职业实践委员会的主持赞助下编写成的，它是一个被设计为达到这个一致的跨越数年的项目的一部分。

什么是“软件工程”？

IEEE计算机学会将“软件工程”定义为：“（1）应用系统化的、学科化的、定量的方法，来开发、运行和维护软件，即，将工程应用到软件。（2）对（1）中各种方法的研究”。（参见：IEEE Standard Glossary of Software Engineering Terminology。IEEE, Piscataway, NJ std 610.12-1990, 1990）

什么是被认可的职业？

软件工程要成为合理的工程学科和一个被认可的职业，在一个核心知识体系上达成一致就非常重要。Starr在定义什么将被认为是一个合理的学科和一个被认可的职业时，清楚地展示了这点。他在获得普利策奖的关于美国医学职业历史的书中，写道：

“专业人员威信的合法化涉及3个不同的需要：首先，专业人员的知识和能力能被其同行所确认；第二，这些被一致确认的知识依靠理性的、科学的基础，第三，专业人员的判断和建议要面向真实的价值，例如健康。这些合法性的各个方面对应于体现在术语“职业”上的各类属性：学院的、认知的和道德的。（参见：P. Starr, The Social Transformation of American Medicine: Basic Books, 1982. p15）。

什么是一个职业的特征？

Gary Ford和Norman Gibbs研究了几个被认可的职业，包括医学、法律、工程和会计等（参见：G Ford and N E Gibbs, “*A Mature Profession of Software Engineering*,” Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical CMU/SEI-96-TR-004, January 1996）。他们的结论是，一个工程职业由下列几个特征刻画：（1）由团体通过认证而确认的课程表的初始职业教育；（2）通过自愿认证或强制许可的适应实践的注册；（3）专门的技术培养和继续职业教育；（4）有职业团体的公共支持；（5）承诺遵从以伦理准则形式形成的规范。

本指南包括了这些成分的前面3个。清晰地指出知识体系是发展一个职业关键的一步，因为它代表了对于软件工程专业人员应该知道什么的一个广泛的一致意见。没有这样的一致，就不能确认任何职业许可的考试，就不能为专业人员参与考试准备课程表，也就不能形成一个认证一个课程表的准则。达成一致也是一个组织中采纳发展连贯技能和继续职业教育程序的前提。

什么是SWEBOK项目的目标？

不应当将指南与知识体系本身混淆，知识体系已经存在与发表的文献中，指南的目的是描述知识体系的哪些部分已经被普遍接受，将这些部分组织起来，提供一个使用它们的主题。对于“普遍接受”包含的附加信息可以在下面或附录A中找到。

建立软件工程知识体系（SWEBOK）指南有下面5个目的：（1）促进世界范围内对软件工程的一致观点；（2）阐明软件工程相对其它学科（如计算机科学、项目管理、计算机工程

和数学等）的位置，并确立它们的分界；（3）刻画软件工程学科的内容；（4）提供使用知识体系的主题；（5）为开发课程表和个人认证与许可材料，提供一个基础。

第一个目标，促进世界范围内对软件工程的一致观点，由指南的开发过程支持，来自42个国家的大约500名评审人员参与了石人阶段（1998年—2001年），产生了试用版本，来自21个国家的120多位评审人员参与了铁人阶段（2003年），产生了2004年版本。关于指南开发过程的更多信息可以在前言看到，或者在www.swebok.org 网站上看到。我们与涉及软件工程的职业和学术团体、公共机构等进行了接触，向它们告知了本项目，邀请它们参与评审过程。我们从北美、太平洋周边地区和欧洲，招募了指南的副编辑，在多个国际会议场合，做了本项目的演示报告，在以后，我们还安排了一些关于本项目的报告。

第二个目标，为软件工程确立边界，激发了本指南的基本组织。被认为是本学科的材料按照表1，组织为10个知识域（Knowledge Areas，KA），本指南对每个知识域分别用一章的篇幅来描述。

表1：SWEBOK知识域

软件需求	Software Requirements
软件设计	Software Design
软件构造	Software Construction
软件测试	Software Testing
软件维护	Software Maintenance
软件配置管理	Software Configuration Management
软件工程管理	Software Engineering Management
软件工程过程	Software Engineering Process
软件工程工具和方法	Software Engineering Tools and Methods
软件质量	Software Quality

在建立边界时，识别哪些学科与软件工程共享边界并有公共的交集，非常重要。最后，本指南识别出8个相关的学科，如表2所示（参见第12章，软件工程相关学科）。当然，软件工程具有来自这些学科的知识材料（知识域将参考它们），但是，SWEBOK指南的目标不是刻画相关学科的知识，而是刻画什么知识被认为对软件工程特别有用。

表2 相关学科

计算机工程	Computer Engineering
计算机科学	Computer Science
管理	Management
数学	Mathematics
项目管理	Project Management
质量管理	Quality Management
软件人类工程学	Software Ergonomics
系统工程	Systems Engineering

层次结构组织

知识域描述或章节Z的组织支持项目的第3个目标----刻画软件工程的内容。项目编辑组给各个副编辑提供的关于知识域描述的内容的详细规格说明，可以在附录A中找到。

本指南采用层次组织，将每个知识域分解为具有可识别标签的主题的集合。两级或三级

的结构分解，为找到感兴趣的主题提供了合理的方法。指南处理主题的方式与主流的思想学派的结构分解兼容，也与工业界和软件工程文献和标准的结构分解兼容。没有为主题的结构分解假设特定的应用领域、商业使用、管理哲学、开发方法等。每个主题的描述程度仅仅为主题的普遍接受特性需要的，并能让读者容易找到参考文献。总之，知识体系可以在参考材料中，而不是在指南中找到。

参考材料和矩阵

项目的第4个目标是提供按主题使用知识，本指南为每个知识域标明了参考材料，包括书记章节、论文和其它被认可的权威信息来源。每个知识域的描述也包括一个矩阵，将参考材料与主题联系起来。引用的文献的总量适合于大学本科毕业、具有4年工作经验的人员掌握。

在本版指南中，为所有知识域都分配了约500页的参考文献，这是对副编辑的规格说明。有人指出，一些知识域（如软件设计）需要更多的参考文献。本指南的以后版本将考虑这一点。

应该指出，指南并不打算在引用时包罗万象，没有包括很多适当的优秀材料，选择材料部分或主要是由于它覆盖了需要的主题。

处理的深度

从一开始，指南的深度问题就有了。项目组采纳了一个方法，以支持项目的第5个目标：为开发课程表、认证和许可提供一个基础。编辑组使用“普遍接受”的知识的准则，与高级研究知识（根据成熟性）和专门知识（基于应用的普遍性）进行区分，如图1所示。这个定义来自项目管理协会：“普遍接受的知识在大多数时间应用于大多数项目，广泛的一致意见确认其价值和效力”。（参见：*A Guide to the Project Management Body of Knowledge*, 2000 Edition, Project Management Institute, Newport Square, PA. www.pmi.org.）

但是，术语“普遍接受”并不意味着，指定的知识应该一律地应用于所有软件工程任务（每个项目需要确定需要的知识），但它意味着，有能力的软件工程师，为了胜任潜在的应用，应该具有这些知识。更确切地说，普遍接受的知识应该包括在软件工程师的许可考试的学习材料中，本这是本科毕业并工作4年后应该参加的考试。尽管这个准则是针对美国风格的教育的，并不适合于其它国家，但我们认为它很有用。但是，普遍接受的知识的两个定义应该被视为互补的。

<table>
<tr><td rowspan="2">专门的

仅用于某些特定类型软件的实践</td><td>普遍接受的

许多组织推荐的已经建立的传统实践</td></tr>
<tr><td>高级的研究性的

被某些组织测试和使用的新的实践、研究组织正在发展和测试的概念</td></tr>
</table>

图 1　知识的分类

与本书格式有关的局限

本版本采用的书记格式有其局限。如果采用超文本结构，内容的本质将会更好，因为主题可以相互链接，而不是按顺序排列。

有时，知识域之间、子知识域之间等的边界多少有些武断。没有过多强调这些边界的重要性，我们尽可能地在相关或有用的地方，使用指示器和链接。

知识域之间的链接并不是输入/输出类型，知识域是对一个人应该拥有的与知识域有关的软件工程知识的观点。知识域内的学科结构分解，以及描述知识域的顺序安排，都没有吸收任何特定的方法和模型。指南中适当的知识域中描述了一些方法，指南本身并不是任何一种方法。

知识域

图2和图3从总体上描述了11章，以及各章中包括的主题。首先按传统的瀑布生命周期顺序描述前5个知识域，但是，这并不表示指南采纳或鼓励瀑布方法，或任何其它模型。随后按字母顺序描述各个知识域，最后一章描述相关的学科。这相当于它们出现在本指南中的顺序。

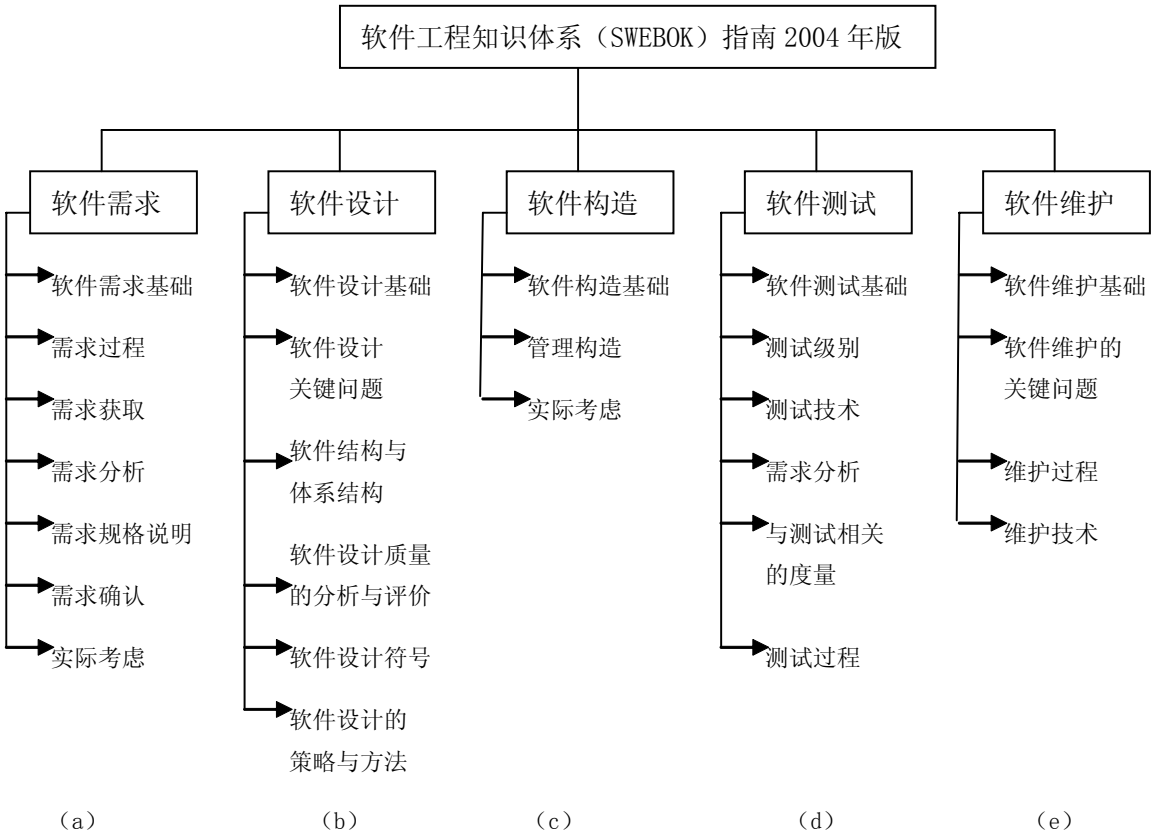

图 2　前 5 个知识域

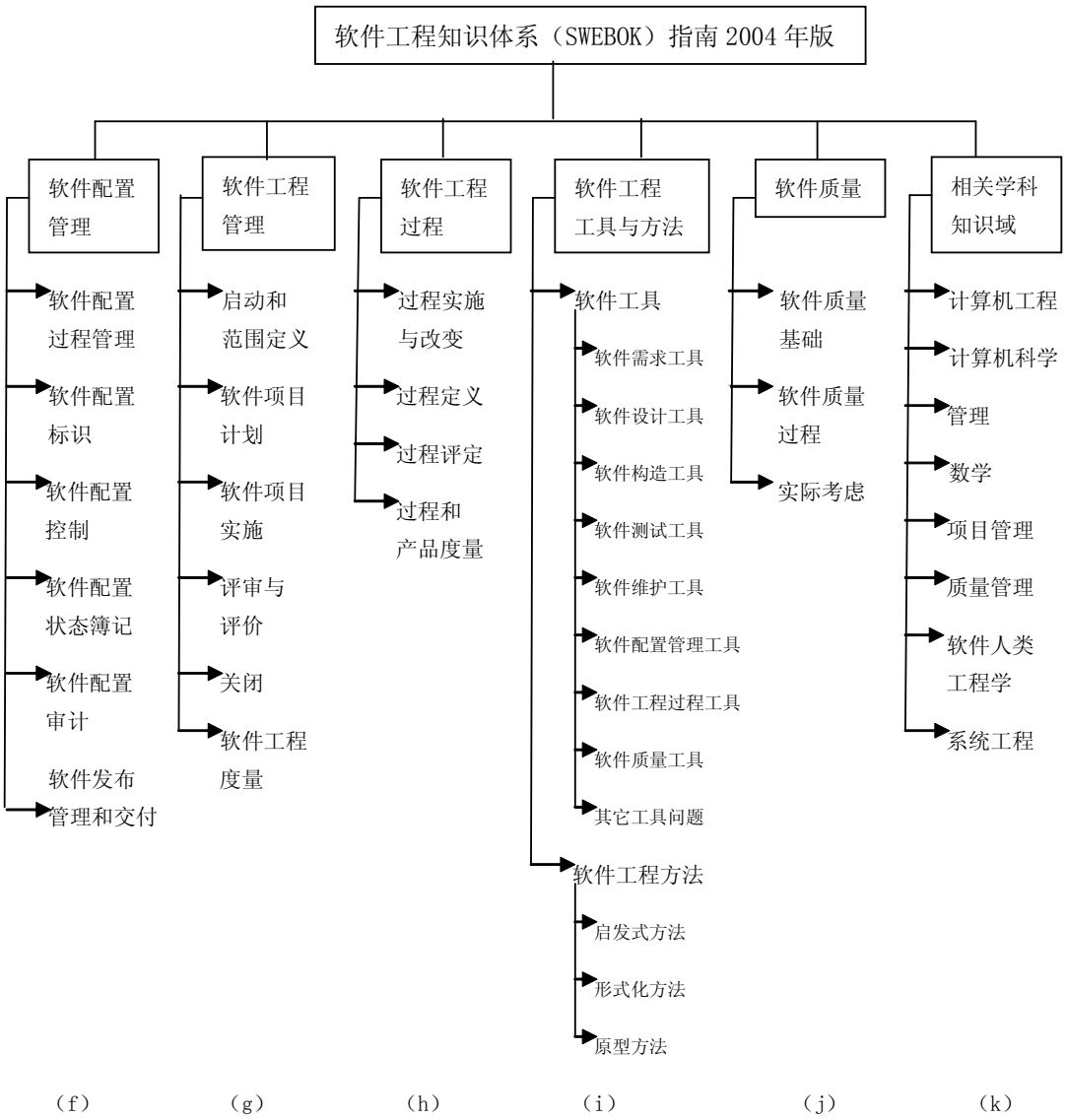

图 3　后 6 个知识域

知识域描述结构

知识域描述的结构如下所述：

在简介中，给出知识域的简要定义、其范围的总体视图、与其它知识域的关系。

主题的结构分解组成每个知识域描述的核心，它描述了将知识域分解为子域、主题和子主题。对于每个主题或子主题，给出简要描述，然后是一篇或多篇参考文献。

选择一个参考材料主要是因为认为它构成了与主题相关的知识的最佳表述，并考虑了对选择参考文献的限制（见前）。我们使用一个矩阵来联系主题和参考材料。

知识域描述的最后一部分是推荐的参考文献列表。每个知识域的附录A为希望了解知识域主题更多内容的读者，列出了深入读物，附录B列出与知识域最相关的标准。注意，方括号[]中的引用表示推荐的参考文献，圆括号（）中的引用表示对于编写或验证文本有用的参考文献，前者可以在对应的知识域章节中找到，后者可以在知识域的附录A中找到。

最后给出了知识域描述的简要总结和附录。

软件需求知识域(图2，列a)

需求定义为解决真实世界问题而必须展示的特性。

第一个知识子域是软件需求基础,它包括软件需求本身的定义、主要的需求类型的定义，如产品与过程、功能与非功能、突发性（emergent）特性等。子域也描述了可量化需求的重要性，并区分了系统的和软件的需求。

第二个子域是需求过程，它介绍过程本身，面向余下的5个子域，说明需求工程如何与其它软件工程过程吻合。它描述了过程模型、过程参与者、过程支持与管理、过程质量与改进。

第三个子域是需求获取，它涉及软件需求来自何方？软件工程师如何收集这些需求，它包括需求来源和收集技术。

第四个子域是需求分析，涉及分析需求的过程：（1）检测和解决需求之间的冲突；（2）发现软件的边界，以及软件如何与外界交互；（3）详细描述系统需求和软件需求。需求分析包括需求分类、概念建模、体系结构设计与需求分配、需求协商。

第五个子域是需求规格说明，一般是指产生一份（电子）文档，这样可以系统地评审、评价和批准需求。对于复杂系统，特别是涉及大量非软件组件的系统，至少需要产生3类不同的文档：系统定义、系统需求规格说明、软件需求规格说明。子域描述了这3类文档，以及产生它们的活动。

第六个子域是需求确认，目标是在分配资源给需求之前，发现任何潜在的问题。需求确认涉及检查需求文档，以保证它们定义了正确的系统（即，这是用户期望的系统）。这个子域进一步划分为需求评审的引导、快速原型、模型确认和接收测试的描述。

第七个和最后一个子域是实际考虑，它描述在实践中需要理解的主题。第一个主题是需求过程的迭代本质，后面3个主题是关于处于实际反映了要建造或已经建造的软件的状态的需求的变更管理和维护。它包括变更管理、需求属性、需求追踪。最后一个的主题是需求度量。

软件设计知识域（图2，列b）

根据IEEE的定义[IEEE610.12--90]，设计既是“定义一个系统或组件的体系结构、组件、接口和其它特征的过程”，又是“这个过程的结果”。软件设计的知识域分为6个子域。

第一个子域是软件设计基础，它是理解软件设计作用和范围的基础，这些是：一般的软件概念、软件设计上下文和软件设计的使能（enabling）技术。

第二个子域将软件设计的关键问题聚集在一起，它们包括：并发性、事件的控制和处理、组件的分布、错误和异常处理、容错、交互与表现、数据持久性。

第三个子域是软件结构与体系结构，它的主题是体系结构与视点、体系结构风格、设计模式、程序与构架族。

第四个子域描述软件设计质量的分析与评价。虽然有一个完整的软件质量知识域，这个子域描述与软件设计质量特别有关的主题。这些方面包括：质量属性、质量分析和评价技术与度量。

第五个子域是软件设计符号，它分为结构与行为描述两部分。

最后一个子域是软件设计策略与方法。首先描述一般策略,然后是面向功能的设计方法、面向对象的设计方法、以数据结构为中心的设计、基于组件的设计和其它方法。

软件构造（图2，列c）

软件构造指通过编码、验证、单元测试、集成测试和排错的组合，详细创建一个可以工作的、有意义的软件，其知识域分为3个子域。

第一个子域是软件构造的基础，前3个主题是：复杂性最小化、变更预见和为验证进行构造。最后一个主题讨论软件构造的标准。

第二个子域描述构造的管理，主题包括：构造的模型、构造的计划、构造的度量。

第三个子域覆盖实践考虑，主题包括：构造的设计、构造的语言、编码、构造的测试、复用、构造的质量和集成。

软件测试（图2，列d）

软件测试由在有限测试用例集合上，根据期望的行为，对程序的行为进行的动态验证组成，测试用例是从实际上是无限的执行域中适当的选择出来的。软件测试包括5个子域。

第一个子域是软件测试基础，首先介绍与测试有关的术语，然后描述测试的关键问题，最后是测试与其它活动的联系。

第二个子域是测试级别，这些是根据测试对象（target）和测试目标来划分的。

第三个子域是测试技术。第一个范畴包括基于测试人员直觉和经验的测试，第二组是基于规格说明的技术组成，然后是基于代码的技术、基于错误（fault）的技术、基于使用的技术和与应用本质有关的技术。最后讨论如何选择和组合适当的技术。

第四个子域是测试相关的度量，度量划分为：与被测试的程序的评价有关的度量、与测试本身的评价有关的度量。

最后一个子域是测试过程，包括了测试时的实际考虑和测试活动。

软件维护（图2，列e）

软件一旦投入运行，就可能出现异常，运行环境可能发生改变，用户回提出新的需求。生命周期的维护阶段从软件交付时开始，但维护活动出现得还要早。软件维护知识域划分为4个子域。

第一个子域是软件维护基础：定义和术语、维护的本质特征、维护的必要性、维护成本的大份额性、软件的进化、维护的分类。

第二个子域将软件维护中的关键问题聚集在一起，这些是：技术问题、管理问题、维护成本估算和软件维护度量。

第三个子域是维护过程，其中的主题包括各种维护过程和维护活动。

第四个子域是维护技术，包括程序的理解、再工程和逆向工程。

软件配置管理（图3，列f）

软件配置管理（Software Configuration Management，SCM）是为了系统地控制配置的变更和维护配置在整个系统的生命周期中的完整性和可追踪性，而标识软件在时间上不同点的配置的学科。这个知识域包括6个子域。

第一个子域是SCM过程管理，包括的主题有：SCM的组织结构上下文、SCM的约束和指导、SCM计划、SCM计划本身和SCM的监管。

第二个子域是软件配置标识，它识别要控制的项目，为各个项目及其版本建立标识方案，确定在获取和管理被控制项目中要使用的工具和技术。子域中第一个主题是识别要控制的项目和软件库。

第三个子域是软件配置控制，它管理软件生命周期中的变更。其中的主题包括：（1）软件变更的请求、评价和批准；（2）实现软件变更；（3）偏离和放弃（deviation and waiver）。

第四个子域是软件配置状态簿记，其主题有软件配置状态信息和软件配置状态报告生成。

第五个子域是软件配置审计，包括：软件功能配置审计、软件物理配置审计、软件基线

的过程内部（in-process）审计。

最后一个子域是软件发布管理和交付，覆盖软件建造和软件发布管理。

软件工程管理（图3，列g）

软件工程管理知识域处理软件工程的管理与度量，虽然度量是所有知识域的一个重要方面，但在这里涉及的是度量程序的主题。软件工程管理游个子域，前5个覆盖软件工程管理，第六个描述软件度量的程序。

第一个子域是启动和范围定义，它由需求的确定与协商、可行性分析、需求的评审和修订过程组成。

第二个子域是软件工程计划，包括过程计划、确定可交付成果、工作量、进度与成本估算、资源分配、风险管理、质量管理、计划本身的管理。

第三个子域是软件项目实施，它的主题是：计划的实现、供应商合同管理、度量过程的实现、过程的监理、过程的控制、报告生成。

第四个主题是评审与评价，主题有：确定需求的满足程度、评审和评价项目性能。

第五个主题描述项目的关闭：确定关闭项目、关闭涉及的活动。

第六个子域描述软件工程度量，特别是度量本身的程序。产品和过程的度量在软件工程过程知识域中描述，许多其它知识域也描述其特定的度量。这个子域的主题包括：建立和维持度量工作、度量过程的计划、进行度量过程和评估度量。

软件工程过程（图3，列h）

软件工程过程的知识域涉及软件工程过程本身的定义、实现、评定、度量、管理、变更和改进。它分为4个子域。

第一个子域是过程实现与变更，其主题有：构成基础结构、软件过程管理周期、过程实现与变更的模型、实际考虑。

第二个子域处理多成定义，主题有：软件生命周期模型、软件生命周期过程、过程定义符号、过程适配（adaptation）和自动化。

第三个子域是过程评定，主题有：过程评定模型和过程评定方法。

第四个子域描述过程与产品度量。软件工程过程覆盖一般的产品度量，以及过程度量。特定于各知识域的度量在相关的知识域描述。子域的主题有：过程度量、软件产品度量、度量结果的质量、软件信息模型和过程度量技术。

软件工程工具和方法（图3，列i）

软件工程工具和方法知识域包括软件工程工具、软件工程方法。

软件工程工具子域使用了与指南相同的结构，为其它9个软件工程知识域各分配一个主题，附加一个主题是其它工具问题，例如工具集成技术，这些问题可能应用于所有类型的工具。

软件工程方法子域分为4个小节：处理形式化途径的启发式方法、处理基于数学的途径的形式化方法、处理基于各种原型的软件开发途径的原型方法、？？。

软件质量（图3，列j）

软件质量知识域处理跨越软件生命周期过程的软件质量的考虑，由于软件质量在软件工程中无处不在，其它知识域也涉及质量问题，读者可以注意到本知识域到其它知识域的指示器。本知识域覆盖4个子域。

第一个子域描述软件质量基础，例如软件工程文化和伦理学、质量的价值与成本、模型

和质量特征和质量改进。

第二个子域覆盖软件质量管理过程，主题有：软件质量保证、验证和确认、评审和审计。

第三个即最后一个子域描述与软件质量有关的实际考虑，主题包括：软件质量需求、缺陷特征、软件质量管理技术、软件质量度量。

软件工程相关学科（图3，列k）

最后一章是软件工程相关学科。为确定软件工程的范围，有必要鉴别与软件工程有公共边界的学科，这一章按字母顺序，鉴别这些相关学科。对每个相关学科，使用我们找到的基于一致认可的来源，进行以下内容的鉴别：（1）资料性定义（可行时）；（2）知识域列表。

相关学科包括：计算机工程、计算机科学、管理、数学、项目管理、质量管理、软件人类工程学、系统工程。

附录

附录A　知识域描述规格说明

这个附录描述了编辑组向副编辑提供的规格说明：内容、推荐的参考文献、格式、知识域的描述风格。

附录B　指南的演化

第二个附录描述项目建议的指南进化。2004年版指南只是指南的当前版本，指南将继续演化，以适应软件工程团体的需要。演化的计划还没有制订，但本指南附录提供了一个完成暂时的过程轮廓。在本次编写时，这个过程已经被项目的工业界顾问委员会认可，并向IEEE计算机学会的主管委员会发送了简报，但还没有得到资助或实施。

附录C　标准到知识域的分配

第三个附录是大多数相关标准的注释表，这些标准主要来自IEEE和ISO，注释表将标准分配到SWEBOK指南的各个知识域。

附录D Bloom级别

作为支持项目的第五个目标的辅助手段，特别是对于课程表开发人员（和其它用户），第五个附录对每个主题按照一个教育学目录集，划分等级，这个目录主要由Benjamin Bloom提出。其概念是，教育目标可以分类6类，分别表示递增的深度：知识、理解、应用、分析、综合和评价。对所有知识域的等级分类结构在附录D中。但是，这个附录不能认为是确定性的分类，而应该被认为是一个起点。

//

第2章　软件需求

术语与缩写

DAG：Directed Acyclic Graph，无环有向图

FSM：Functional Size Measurement，功能规模估算

INCOSE：International Council on Systems Engineering，国际系统工程理事会

SADT：Structured Analysis and Design Technique，结构化分析与设计技术

UML：Unified Modeling Language，统一建模语言

引言

软件需求知识域涉及软件需求的获取、分析、规格说明和确认。在软件工业界，人们广泛认为，如果这些活动完成得不好，软件工程项目很容易上失败。

软件需求表达了施加在要解决真实世界问题的软件产品上的要求和约束[Kot00]。

术语“需求工程”在需求中被广泛使用，表示系统地处理需求。但是出于一致性，本指南将不使用它，正如本指南要避免使用术语“工程”来表示除了软件工程活动以外的活动一样。

基于同样原因，本指南也不使用出现在一些文献中的术语“需求工程师”，我们使用“软件工程师”，有时也使用“需求专家”，后一个术语表示问题中的角色通常由个人完成，而不是软件工程师完成。但是，这并不表示软件工程师不能去完成这个任务。

知识域的结构分解与IEEE12207中关于需求活动一节广泛兼容(IEEE12207.1-96)。

在我们提出的结构分解中有一个固有风险:隐含了类似瀑布模型的过程。为澄清这一点，我们设计了子域2（需求过程）来给出需求过程的高层视图，它确定过程进行需要的资源和约束，以及使其具有一定形式而需要的行动。

另外一个可能的分解是使用基于产品的结构（系统需求、软件需求、原型、用况等）。基于过程的分解结构反映了一个事实：需求过程要成功，就必须作为一个涉及复杂、紧密耦合的多个活动组成的过程考虑，而不是作为在软件开发项目开始就进行的离散的、单个的活动组成的过程来考虑。

软件需求知识域与软件设计、软件测试、软件维护、软件配置管理、软件工程管理、软件工程过程和软件质量等的知识域紧密相关。

软件需求主题的结构分解

1. 软件需求基础

1.1. 软件需求定义

软件需求的最基本含义是一个为了解决真实世界问题而必须展示的特性。指南将需求与软件联系起来，因为需求涉及的是软件要解决的问题。因此，软件需求是一个为解决特定问题而必须由被开发或被修改的软件展示的特性。这个问题可能是使用软件的某人的任务中的一个自动化部分，或是支持委托开发软件的组织的业务过程，或修正当前软件的缺点，或是控制一个设备等等。用户、业务过程和设备的功能通常很复杂，因此，特定软件的需求在外延上通常是来自一个组织不同层次的不同人员的需求和来自软件将要在其中运行的环境的需求的复杂组合。

所有软件需求的一个基本特性就是：可验证。验证某些软件需求可能很困难或则成本很高，例如，验证呼叫中心的吞吐量需求就需要开发模拟软件。软件需求和软件质量人员都必须保证，可以在现有的资源约束下，需求可以被验证。

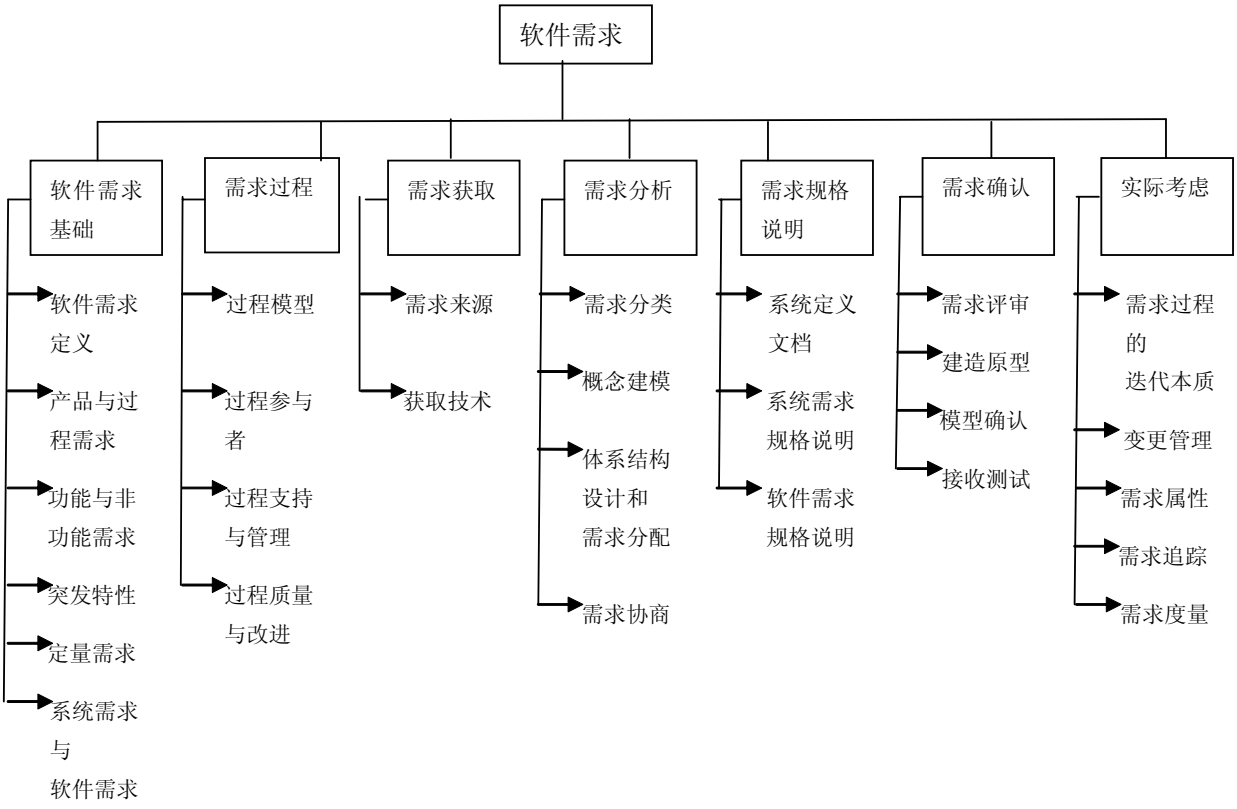

图 1　软件需求知识域主题的结构分解

除了其表达的行为特性外，需求还有其它的属性。普遍的例子是一个优先级别，以便在资源有限时进行权衡，或者一个分情况的价值，以便监控项目的进展。通常，软件需求要唯一地标识，才能在整个软件生命周期中，进行软件配置控制和管理[Kot00; Pfl01; Som05; Tha97]。

1.2 产品和过程需求

可以区别产品参数与过程参数，产品参数是需要开发的软件上的需求（例如，“软件应该验证一个学生在注册一门课程时，已经满足了所有的前提条件”），过程参数本质上是对软件开发的约束（例如，“软件应该用Ada编写”），过程参数有时也叫过程需求。

一些软件需求可以产生隐含的过程需求，一个例子就是选择验证技术，另外一个例子可能是要求使用特别严格的分析技术（如形式化规格说明方法）来减少可能导致不适当可靠性的故障。开发组织、客户或第三方（如安全管理人员）也可能直接提出过程需求[Kot00; Som97]。

1.3 功能与非功能需求

功能需求描述软件要执行的功能，例如，将某些文本格式化或调制一个信号，功能有时叫做能力。

非功能需求是限制解决方案的需求，非功能需求有时叫做约束或质量需求，可以进一步划分为：性能需求、可维护性需求、安全性需求、可靠性需求，以及其它类型的软件需求，这些主题也在软件质量知识域中讨论[Kot00;Som97]。

1.4 突发特性

一些需求表现了软件的突发（Emergent）特性，即，这些需求不能由单个组件完成，根据其满足程度，依赖于所有软件组件之间的互操作。例如，呼叫中心的吞吐量需求可能依赖于电话系统、信息系统和接线员在实际运行条件下的相互作用。突发特性特别依赖于系统体系结构[Som05]。

1.5 定量需求

软件需求应尽可能清楚地无二义性地陈述， 可能时，应该定量地陈述。避免含糊和不可验证的需求非常重要，因为这些需求依赖于主观判断的解释（“软件应该可靠”、“软件应该用户友好”）。这对于非功能性需求特别重要。下面是两个定量需求的例子：呼叫中心软件必须增加20%的吞吐量；在运行的任一小时内，系统产生致命错误的概率应该小于$1.0*10^{-8}$。吞吐量需求是一个高层次的需求，可以导出许多详细需求，可靠性需求将紧密约束系统体系结构[Dav93; Som05]。

1.6 系统需求与软件需求

在这个主题中，系统的含义是“多个元素相互作用的组合，以实现为其定义的目标，元素包括：硬件、软件、固件、人员、信息、技术、设施、服务和其它支持元素”，这是国际系统工程理事会的定义(INCOSE00)。

系统需求是对整个系统的需求，在包含软件组件的系统中，软件需求是从系统需求中导出的。关于需求的文献有时将系统需求称为“用户需求”。本指南将“用户需求”严格定义为系统客户或终端用户的需求。这样，系统需求包括：用户需求，其它干系人的需求（如管理层），以及没有可标识的人类来源的需求。

2 需求过程

本节介绍软件需求过程，它面向剩余的5个子域，说明软件需求过程如何与总的软件工程过程吻合[Dav93; Som05]。

2.1 过程模型

这个主题的目的是提供对需求过程的如下理解：（1）需求过程不是软件生命周期中的一个离散的从头到尾（front-end）的活动，而是一个过程，开始于项目的起始阶段，并在整个生命周期中需要持续精化。（2）需求过程要将软件需求标识配置项，并使用与其它软件生命周期过程的产品相同的软件配置管理实践来管理软件需求。（3）需求过程需要与组织和项目上下文保持适应。

这个主题也包括为需求过程提供输入的活动，如市场和可行性研究等[Kot00; Rob99; Som97; Som05]。

2.2 过程参与者

这个主题介绍参与需求过程的人员的角色。需求过程本身是交叉学科的需求专家需要在干系人和软件工程师之间进行协调。除了需求专家外，有许多人员参与，每个人都与软件有某种干系。随项目不同，干系人也不同，但总是包括：用户/操作人员和客户（二者可以不同）[Gog93]。软件干系人的典型例子包括（但不局限于）：（1）用户：这个群体由将要操作软件的人员组成，通常是一个杂合群体，包含有不同角色和需求的人员。（2）客户：这个群体由委托软件开发的人员和代表软件的目标市场的人员组成。（3）市场分析师：大众市场产品不止一个委托方，因此需要市场营销人员来确定市场的要求和作为代理客户。（4）

管制人员：许多应用领域，如银行和公共运输，是被政府管制的，这些领域的软件必须遵从管制当局的需求。（5）软件工程师：这些人员对从软件开发中通过复用其它产品的组件而获利感兴趣，在这种情形下，如果特定产品的用户有复用组件的特殊需求，软件工程师必须权衡自身的利益和客户的利益。

满足所有干系人的需求，通常是不可能的，软件工程师的任务就是协调各种权衡，使之能被主要的干系人接受，并能满足预算、技术、管制规则和其它约束。作到这一点的前提是，已经标识了所有干系人，分析了他们的利益本质，获得了他们的需求 [Dav93;Kot00; Rob99; Som97; You01]。

2.3 过程支持与管理

这个主题介绍需求过程需要和消耗的管理资源，它为软件工程管理知识域的第一个子域（启动和范围定义）建立上下文，它的主要目的是建立2.1中标识的过程活动与成本、人力资源、培训和工具等问题之间的联系[Rob99; Som97; You01]。

2.4 过程质量与改进

这个主题涉及需求过程的质量评定和改进，其目的是强调需求过程在软件产品的成本和准时性方面、客户对需求过程的满意度方面所起的关键作用[Som97]。它有助于需求过程适应软件和系统的质量标准和过程改进模型。过程质量和改进与软件质量知识域和软件工程过程知识域都密切相关，特别是软件质量属性和度量的问题，以及软件过程定义的问题。这个主题覆盖下列问题：（1）过程改进标准和模型覆盖的需求过程范围；（2）需求过程度量与基准；（3）改进的计划与实现[Kot00;Som97; You01]。

3 需求获取[Dav93; Gog93; Lou95; Pfl01]

需求获取涉及软件需求来自何方，软件工程师如何收集它们。这是理解需要软件解决的问题的第一阶段，这是本质是一个人类活动，活动中要标识干系人，建立开发小组与客户之间的联系。需求获取又称为“需求捕获”、“需求发现”和“需求获得”。

良好的软件工程的一个基本原则就是，软件工程师与软件用户之间有良好的沟通。在开发活动开始前，需求专家必须为这个沟通建立渠道，他们必须协调软件用户（及其它干系人）的业务领域和软件工程师的技术领域。

3.1 需求来源[Dav93; Gog93; Pfl01]

在典型的软件中，需求有许多来源，有必要标识所有潜在的来源，并评价其对软件的影响。设计这个主题是为了促进对软件需求的各种来源的意识和管理这些来源的框架的意识。要点如下：

（1）目标：术语“目标”（有时称为“业务关注”或“关键成功因素”）指的是软件总体的、高层次的目标。目标为软件提供了动机，但通常被含糊地表达。软件工程师需要特别注意评定目标（相对与优先权）的价值和成本，一个低成本的评定方法是可行性研究[Lou95]。

（2）领域知识：软件工程师需要获得或具有关于应用领域的知识，这样，能够推导出干系人没有清楚说明的隐性知识，能够评定相互矛盾的需求之间的必要的权衡，并作为“用户”支持者。

（3）干系人（参见2.2过程参与者）：由于过分强调了一部分干系人的需求而牺牲了其它干系人的需求，许多软件的结果不令人满意。因此，交互的软件难以使用，或违反了客户组织的文化或政策结构。软件工程师需要标识、描绘和管理很多不同类型的干系人的视点

[Kot00]。

（4）运行环境：需求能够从软件将要运行的环境中导出。例如，实时软件的定时约束、办公环境中的互操作约束。必须主动地寻找这些需求，因为它们可能对软件的可行性和成本有重大影响，并限制设计的选择[Tha97]。

（5）组织环境：通常要求软件支持某个业务过程，业务过程的选择受组织的结构、文化和内部策略的限制。软件工程师对这些事情应该敏感，因为，一般要求新的软件不能对业务过程施加没有计划的变更。

3.2 获取技术[Dav93; Kot00; Lou95; Pfl01]

一旦标识了需求来源后，软件工程师就开始从需求来源获取需求。这个主题主要是让人类干系人清晰地表达他们的需求的技术。这是一个非常困难的领域，软件工程师需要记住这个事实（例如）：用户可能难以描述他们的任务，可能遗漏重要的信息，可能不愿意或不能够合作。特别重要的是，需要理解，获取不是一个被动的任务，即使干系人愿意合作并能清晰表达其任务，软件工程师也必须努力工作，以获取正确的信息。有大量的需求获取技术，下面是主要的几个[Gog93]。

（1）面谈：这是一个“传统”的获取需求的方法。了解面谈的优点和限制，以及如何引导面谈，非常重要。

（2）场景：这是一个为获取用户需求提供上下文的有价值的工具，场景让软件工程师为有关用户任务的问题提供一个框架，允许提出“如果…那么…”和“这是如何做的”之类的问题。场景最普通的类型就是用况，这里提供一个到主题4.2（概念建模）的指示器，因为用况和用况图等场景符号在概念建模中很普通。

（3）原型：这是一个阐明不清楚的需求的有价值的工具，原型与场景的作用类似，向用户提供一个可以更好理解他们需要提供的信息的上下文。原型技术的范围很广，从纸面的屏幕模拟设计到软件产品的β版本，原型用于需求获取和用于需求验证，二者之间有很多重叠（参见主题6.2建立原型）。

（4）恳谈会：目的是试图达到一个综合效果，因为一群人对其软件需求的洞察比与向单个人了解的要深，他们可以采用头脑风暴，精化其想法，这些想法是使用面谈难以得到的。另一个优点是，相互冲突的需求出现得早，让干系人能认识到存在冲突。如果组织得好，这个技术可以比其它技术得到更丰富更一致的需求。但是，需要仔细控制会议（因此，需要一个推动者），以避免出现小组中批判的能力被对团体的忠诚消蚀，或者出现赞同少数人说出的需求而损害其他人员。

（5）观察：在组织环境中软件的上下文的重要性，导致修改观察技术来获取需求。软件工程师通过将自己投入到环境中，观察用户如何与其软件交互或相互交互，来学习用户的任务。这些技术成本相对高，但却是有益的，因为它们能阐明许多参与者不能轻易描述清楚的微妙和复杂的用户任务和业务过程。

4 需求分析[Som05]

这个主题涉及分析需求的过程，目的是：（1）检测和解决需求之间的冲突；（2）发现软件的边界，以及软件与其环境如何交互；（3）详细描述系统需求，以导出软件需求。
需求分析的传统观点是：需求分析被精简为使用多个分析方法中的一个来进行概念建模，这些方法包括：结构化分析与设计（Structured Analysis and Design Technique，SADT）。尽管概念建模很重要，我们将需求分类也包括进来，以帮助需求之间的权衡的告知（需求分类）和建立这些权衡的过程（需求协商）[Dav93]。

描述需求时，必须仔细，应该精确到能确认需求，验证需求的实现，估算需求的成本。

4.1 需求分类[Dav93; Kot00; Som05]

需求可以在多个维度上分类，下面是一些例子：

（1）需求是功能性的还是非功能性的（参见主题1.3功能与非功能需求）。

（2）需求是从一个或多个高层次需求或突发性特性（参见主题1.4突发特性）导出的，还是由干系人或其它来源直接施加在软件上的。

（3）需求是针对产品的还是过程的。对过程的需求可以约束合同商的选择、软件工程过程的采用、要遵循的标准。

（4）需求的优先性。一般，优先性越高，需求对于满足软件的总体目标就越基本。通常用固定尺度进行分类，如：强制的、特别需要的、需要的、可选的。优先性通常要与开发和实现的成本进行平衡。

（5）需求的范围。范围指的是需求影响软件和软件组件的程度。一些需求，特别是一些非功能性需求，具有全局范围，因为对这些需求的满足不能分配到某一个离散的组件。因此，全局性需求可能强烈地影响软件的体系结构和许多组件的设计，范围小的需求对于满足需求没有多少影响。

（6）易变性/稳定性。一些需求在软件生命周期内将发生变化，甚至在开发过程本身中发生变化。估计需求可能发生的变化概率，是有用的。例如，在银行应用中，对客户帐户计算和除息功能的需求可能比支持特定类型的免税帐户的需求更稳定，前者反映了银行领域的一个基本特征（帐户可以有利息），后者可能由于政府的法规而被废止。标记潜在的易变需求，有助于软件工程师建立一个更容许变化的设计。

也可以采用其它适当的分类，这取决于组织的平常实践和应用本身。

需求分类和需求属性之间有较大的重叠（参见主题7.3需求属性）。

4.2 概念建模[Dav93; Kot00; Som05]

开发真实世界问题的模型是软件需求分析的关键，模型的目的是帮助理解问题，而不是启动方案的设计。因此，概念模型由来自问题域的实体模型组成，实体模型反映了它们的真实世界联系和依赖。可以开发几类模型，包括：数据和控制流、状态模型、事件追踪、用户交互、对象模型、数据模型，以及其它模型。影响模型选择的因素有：

（1）问题的本质：一些类型的软件要求某些方面的分析特别严格，例如，控制流和状态模型对于实时软件来说，就比管理信息软件重要得多，后者通常使用数据模型。

（2）软件工程师的技能：采用软件工程师有经验的建模符号或方法，具有更高的生产率。

（3）客户的过程需求：客户可能要求使用其喜欢的符号或方法，或者禁止使用其不熟悉的方法。这个因素与前面一个因素是冲突的。

（4）方法和工具的可用性：尽管适合于某些特定的问题，培训和工具不常支持的符号或方法可能不被广泛接受。

注意，几乎所有情况下，从构造软件的上下文模型开始，是有用的。软件上下文提供了打算开发的软件与其外部环境之间的联系，这对于理解软件运行环境中的软件上下文，对于标识软件与环境的接口，都很关键。

建模问题与方法紧密联系，对于实用目的，方法是由引导符号应用的过程支持的一个符号（或一组符号）。几乎没有经验迹象表明一个符号对另一个符号的优越性。但是，特定方法和符号的广泛接受可能导致工业界的技能和知识的聚集，这就是目前UML（统一建模语言）情况(UML04)。

可以追溯到逻辑推理的、使用基于离散数学符号的建模，已经对某些特定领域产生影响。

客户或标准可能要求使用这些方法，并可能对某些关键功能或组件的分析提供引人注目的优点。

本主题并不寻求“教授”一种特定的建模风格或符号，而是提供建模目的和意图的指南。两个标准提供了可能在进行概念建模时有用的符号：IEEE1320.1功能建模的IDEF0标准和IEEE1320.2信息建模的IDEF1X97(IDEF对象)标准。

4.3 体系结构设计与需求分配[Dav93; Som05]

在某些时候，必须导出方案的体系结构。体系结构设计是需求过程与软件或系统设计重叠时进行的，这说明将二者截然分开是不可能的[Som01]。本主题与软件设计知识域的软件结构与体系结构子域紧密联系。在许多情况下，软件工程师要作为软件体系结构师，因为分析和详细阐明需求的任务，要求标识负责满足需求的组件。这是一个需求分配：将满足需求的责任分配到组件上。

分配对于需求的详细分析很重要，例如，一旦将一组需求分配给一个组件，就可以进一步分解单个需求，以发现有关组件需要如何与其它组件交互以满足分配的需求的更深入的需求。在大型项目中，分配刺激新一轮的对子系统的分析，例如，对轿车的特定刹车性能的需求（刹车距离、恶劣驾驶条件下的安全性、应用的平滑性、需要的踏板压力等），可以分配给刹车硬件（机械和液压组件）和一个反锁定的刹车系统（ABS）。只有在标识了一个反锁定的刹车系统，并将需求分配给它后，才能使用ABS的能力、刹车硬件和突发特性（如轿车重量）等来标识详细的ABS软件需求。

体系结构设计与概念建模差不多，从真实世界领域实体到软件组件的映射并不总是显然的，因此将体系结构设计标识为一个独立的主题。对于概念建模和小、体系结构设计，二者对于符号和方法的需求广泛地相同。

IEEE1471-2000标准（软件密集系统的体系结构描述的推荐实践）建议使用多视点的途径来描述系统的体系结构及其软件项(IEEE1471-00)。

4.4 需求协商

本子主题中另一个普遍使用的术语是“解决冲突”。这涉及需求冲突的问题，冲突发生在两个干系人需要不兼容的特征，或者发生在需求与资源之间A，或者在功能与非功能需求之间[Kot00,Som97]。多数情况下，软件工程师作出单边的决定是不明智的，与干系人协商达成适当的权衡，就有必要。从合同的原因上，这样一个决定应该追溯到客户。我们将需求协商分类为一个软件需求分析主题，是因为问题是作为分析的结果出现的。但是，也可以将其作为需求确认的一个主题。

5 需求规格说明

对于许多工程职业，术语“规格说明”指的是将数值或限制分配给产品的设计目标(Vin90)。一般的物理系统的值的数目相对少，而通常软件却有大量的需求，在完成数量化和管理大量需求的复杂的相互作用之间有相同的重点。在软件工程术语中，“软件需求规格说明”一般指产生一个文档或其电子版本，以便系统地评审、评价和批准。对于复杂的系统，特别是涉及许多非软件组件的系统，最多要产生3类文档：系统定义、系统需求和软件需求。对于简单的软件产品，只需求软件需求文档。下面描述所有3类文档，需要指出，它们应该适当组合。对于系统工程的描述可以参见第12章“软件工程相关学科”。

5.1 系统定义文档

这个文档（有时称为用户需求文档或操作概念）记录系统的需求，它从领域角度，定义

高层系统需求，其读者包括系统用户/客户代表（对于市场驱动的软件，市场人员可以担任这些角色），因此，它的术语必须采用领域术语。文档需要列举系统需求，以及有关系统总体目标的背景信息、系统的目标环境，以及对约束、假设和非工程需求的陈述。它也可以包括设计用于说明系统上下文的概念模型、使用场景和主要的领域实体，还包括数据、信息和工作流。IEEE1362标准“操作文档概念”为准备这种文档的内容提出了建议(IEEE1362-98)。

5.2 系统需求规格说明[Dav93; Kot00; Rob99; Tha97]

开发包含许多软件和非软件组件的系统（例如，现代民航客机）的人员，通常将系统需求描述与软件需求描述分开。按这个观点，首先规格说明系统需求，软件需求从系统需求导出，然后为每个软件组件的需求进行规格说明。严格地讲，系统需求规格说明是一个系统工程活动，不属于本指南的范围。IEEE1233标准是一个开发系统需求的指南(IEEE1233-98)。

5.3 软件需求规格说明[Kot00; Rob99]

软件需求规格说明为客户和合同商或供应商（对于市场驱动的项目，这些角色可以有市场人员和开发部门担任）之间达成的一致建立了基础：软件产品要做什么，软件产品不应该做什么。对于非技术领域读者，除了软件需求规格说明文档，通常还需要软件需求定义文档。

软件需求规格说明允许在设计开始之前，对需求进行严格的评定，以减少以后的重新设计工作，它也为估算产品的成本、风险和进度提供一个现实的基础。组织结构也可使用软件需求规格说明文档来更有效地开发自己的确认和验证计划。

软件需求规格说明为将软件产品传递到新用户或新机器提供了一个信息基础。最后，它为软件的增强也提供一个基础。

软件需求通常用自然语言编写，但是对于软件需求规格说明，需要用形式化或半形式化描述离开补充。选择适当的符号，可以更精确更简明地描述软件体系结构的特定需求和特定方面。一般的规则是，使用的符号应该尽可能简明地描述需求，这对于某些安全性要求很高的软件，以及某些对其依赖性强的软件，特别重要。但是，符号的选择通常受培训、技能和文档的作者与读者的限制。

已经有了大量的质量指标，用于将软件需求规格说明的质量与产品的其它属性联系起来，诸如成本、可接收性、性能、进度、可重现性等。单个软件需求规格说明语句的质量指标包括：祈使语气、引导语句、弱短语、选项和连续性。整个软件需求规格说明文档的指标包括：篇幅、可读性、规范、深度和文本结构[Dav93; Tha97] (Ros98)。

IEEE为软件需求规格说明的制作和内容发布了一个标准，称为IEEE830标准[IEEE830-98]。另外，IEEE1465标准（类似ISO/IEC12119标准）是一个处理软件包中质量需求的标准 (IEEE1465-98)。

6 需求确认[Dav93]

需求文档需要经过确认和验证过程。确认需求以保证软件工程师已经理解了需求，验证需求文档遵从公司标准，并且是可理解的、一致的和完备的，也很重要。形式化符号在（严格地说，至少）证明后面两个特性方面具有很强的优点。不同的干系人，包括客户和开发人员代表，应该评审文档。需求文档与软件生命周期过程中其它可交付产品一样，要由软件配置管理来管理(Bry94, Ros98)。

通常需要在需求过程中明确一个或多个点，以进行需求的确认。目标是分配资源给需求之前，发现存在的问题。需求验证涉及检查需求文档，以保证它确实定义了正确的软件（即，这是用户期望的软件）[Kot00]。

6.1 需求评审[Kot00; Som05; Tha97]

最普通的确认手段可能就是检查或评审需求文档，分配一组评审人员简要地检查是否存在：错误、不正确的假设、阐述不清楚、与标准不一致等。引导评审的小组的组成很重要（例如，对于市场驱动的产品，至少应该包括一个来自客户的代表），小组可以提供检查清单中应该出现的内容。

在系统定义文档、软件需求规格说明文档、新发布版本的基线规格说明或过程中任何其它步骤完成时，应该进行评审。IEEE1028标准提供了如何进行评审的指导(IEEE1028-97) 。软件质量知识域中也覆盖了评审，参见主题2.3评审与审计。

6.2 建立原型[Dav93; Kot00; Som05; Tha97]

原形是确认软件工程师对软件需求的解释、获取新需求的常用手段。就获取需求而言，有大量的原型技术，在过程中也有大量的检查点可以使用原型确认。原型的一个优点是：它使得解释软件工程师的假设变得容易，并能在需要的时候，给出有用的反馈，说明一些假设为什么不正确。例如，使用动画原型比文本描述或图形模型，能更好地理解用户界面的动态行为。但是，原型也有缺点。由于原型的外观或质量问题，它可能将用户的注意力从基础的核心功能转移到其它方面。因此，一些人推荐不使用软件的原型，例如基于翻动图表的模拟。开发原形的成本可能会高。但是，如果原型能够避免由于试图满足错误的需求而浪费的资源，这些成本也是值得付出的。

6.3 模型确认[Dav93; Kot00; Tha97]

在分析时，通常有必要确认开发的模型的质量。例如，在对象模型中，进行静态分析，验证对象之间存在通信路径（在干系人领域，就是交换数据），就有用处。如果使用了形式化规格说明符号，就有可能使用形式化推理来证明规格说明特性。

6.4 接收测试[Dav93]

软件需求的一个本质特性就是，应该能够确认完成的产品满足需求。不能被确认的需求实际上仅仅是“愿望”。因此，一个重要的任务就是计划如何确认每个需求，在多数情况下，设计接收测试来完成这个任务。标识和设计非功能需求（参见主题1.3功能与非功能需求）的接收测试可能比较困难，为了确认它们，必须分析它们，用定量的形式描述它们。

额外的信息可以在软件测试知识域的子主题2.2.4遵从性测试中找到。

7 实际考虑

本知识域的第一级分解看起来描述的是线性的活动序列，这是一个过程的简化视图[Dav93]。需求过程跨越了整个软件生命周期。在某个状态中的需求的变更管理和维护，是软件工程过程成功的关键，它们精确地反映了待构造的软件或已经构造的软件[Kot00; Lou95]。

并非每个组织都有将需求文档话和管理需求的文化，在动态启动的公司中。受强烈的“产品视觉”驱动和资源的限制，常常将需求文档看成是不必要的额外开销。但是，随着这些公司的扩展、客户的增加、公司产品的进化，他们经常发现，自己需要恢复那些促进产品特征的需求，以评定建议的变更的影响范围。这样，需求文档化和变更管理是任何需求过程成功的关键。

7.1 需求过程的迭代本质[Kot00; You01]

软件工业的普遍压力是要求更短的开发周期，特别是竞争非常激烈的市场驱动的部门。此外，许多项目多少受环境约束，很多是升级或修订给定了体系结构的现存软件。在实践中，

将需求过程实现为线性的、确定性的过程，是不现实的。在线性的、确定性的过程中，从干系人获得需求，确定基线，分配需求，交付给软件开发小组，顺次完成。大型软件项目的需求被完美理解或完美规格说明，这只是一个神话[Som97]。

相反，需求通常是反复地向一个详细和质量水平前进的，这个水平允许人们作出设计和采购决定。在一些项目中，这样可能导致需求在所有其它特性被充分理解之间，被作为基线。如果问题在以后的软件工程过程中出现，就会有推倒重来的高成本风险。但是，软件工程师应该受项目管理计划的约束，并采取措施，确保需求的“质量”在给定的资源条件下，尽可能高。例如，他们应该清楚说明所有支持需求的假设，以及其它任何已知的问题。

几乎在所有情况下，对需求的理解会随着设计和开发的进行而继续演化，这常常导致在生命周期后期修改需求。也许，理解需求工程中最关键的一点是：相当大一部分需求将回改变。有时，这是由于分析中的错误，但它经常是“环境”改变的一个不可避免的后果：例如，客户的运行或业务环境、软件将要在其中销售的市场环境。无论什么原因，认识到变化的不可避免性和采取措施来缓和它的影响，非常重要。必须管理这些变更，要求提出的变更必须经过一个事先定义的评审和批准过程，并使用仔细的需求追踪、后果分析和软件配置管理(参加软件配置管理知识域）。因此，需求过程不仅仅是软件生命周期中个一个简单的任务，而是跨越整个软件生命周期。在一个典型的项目中，软件需求活动从获取到变更管理，都随时间演化。

7.2 变更管理[Kot00]

变更管理是需求管理的中心。这个主题描述了变更管理的角色、需要经过的程序、应该对提出的变更进行的分析。它与软件配置管理知识域有较强的联系。

7.3 需求的属性[Kot00]

需求不仅仅由需要的事物的规格说明组成，还包括帮助管理和解释需求的辅助信息。这应该包括需求的各种不同的分类维度（参见主题4.1需求分类）和验证方法或接收测试计划。它还可能包括每个需求的概要性原理、每个需求的来源、变更历史等附加信息。但是，最重要的需求属性是一个标识符，它允许需求被唯一地和无二义地被标识。

7.4 需求追踪[Kot00]

需求追踪涉及恢复需求的来源和预测需求的效果。追踪是完成需求变更时的效果分析的基础。一个需求应该能够追踪回溯到原始需求和提出需求的干系人（例如，从软件需求回溯到其需要满足的系统需求）。一个需求还应能够向前追踪到满足的需求和设计实体（例如，从系统需求追踪到详细描述它的软件需求，追踪到实现它的代码模块）。典型项目的需求追踪将形成需求的一个复杂的有向无环图(DAG)。

7.5 度量需求

作为一个实用问题，对于一个特定的软件产品，有一个需求的“量”的概念是有用的。这个数在评价需求变更的“规模”、估算开发或维护任务的成本时，很有用，也可简单用于其它度量的分母。功能规模度量（Functional Size Measurement，FSM）是评价功能需求体系的规模的技术，IEEE14143.1标准定义了FSM的概念[IEEE14143.1-00]。ISO/IEC的标准和其它标准描述了特定的FSM方法。

有关规模度量和标准的其它信息可以在软件工程过程知识域中找到。

主题与参考资料矩阵

	[DAV93]	[Gog93]	[IEEE830-98]	[IEEE1414 3.1-100]	[Kot00]	[Lou95]	[Pfl01]	[Rob99]	[Som97]	[SOm05]	[Tha97]	[You01]
1软件需求基础												
1.1软件需求定义					*		*			C5	C1	
1.2产品与过程需求					*				C1			
1.3功能与非功能需求					*				C1			
1.4紧急特性										C2		
1.5定量需求	C3 S4									C6		
1.6系统需求与软件需求												
2需求过程	*									C5		
2.1过程模型					C2 S1			*	C2	C3		
2.2过程参与者	C2	*			C2 S2			C3	C2			C3
2.3过程支持与管理								C3	C2			C2 C7
2.4过程质量与改进					C2 S4				C2			C5
3需求获取	*	*				*	*					
3.1需求来源	C2	*			C3 S1	*	*				C1	
3.2获取技术	C2	*			C3 S2	*	*					
4需求分析	*									C6		
4.1需求分类	*				C8 S1					C6		
4.2概念建模	*				*					C7		
4.3体系结构设计与需求分配	*									C10		
4.4需求协商					C3 S4				*			
5需求规格说明												
5.1系统定义文档												
5.2系统需求规格说明	*				*			C9			C3	

5.3软件需求规格说明	*		*		*			C9			C3	
6需求确认	*				*							
6.1需求评审					C4 S1					C6	C5	
6.2建造原型	C6				C4 S2					C8	C6	
6.3模型确认	*				C4 S3						C5	
6.4接收测试	*											
7实际考虑	*				*	*						
7.1需求过程的迭代本质					C5 S1				C2			C6
7.2变更管理					C5 S3							
7.3需求属性					C5 S2							
7.4需求追踪					C5 S4							
7.5需求度量				*								

推荐的软件需求参考文献

[Dav93] A. M. Davis, *Software Requirements: Objects,Functions and States*: Prentice-Hall, 1993.

[Gog93] J. Goguen and C. Linde, "Techniques for Requirements Elicitation," presented at International Symposium on Requirements Engineering, San Diego, California, 1993

[IEEE830-98] IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements Specifications*: IEEE,1998.

(IEEE14143.1-00) IEEE Std 14143.1-2000//ISO/IEC14143-1:1998, *Information Technology-Software Measurement-Functional Size Measurement-Part 1: Definitions of Concepts*: IEEE, 2000.

[Kot00] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*: John Wiley and Sons, 2000.

[Lou95] P. Loucopulos and V. Karakostas, *Systems Requirements Engineering*: McGraw-Hill, 1995.

[Pfl01] S. L. Pfleeger, "Software Engineering: Theory and Practice," Second ed: Prentice-Hall, 2001, Chap. 4.

[Rob99] S. Robertson and J. Robertson, *Mastering the Requirements Process*: Addison-Wesley, 1999.

[Som97] I. Sommerville and P. Sawyer, "Requirements engineering: A Good Practice Guide," John Wiley and Sons, 1997, Chap. 1-2.

[Som05] I. Sommerville, "Software Engineering," Seventh ed: Addison-Wesley, 2005.

[Tha97] R. H. Thayer and M. Dorfman, Eds., "Software Requirements Engineering." IEEE

Computer Society Press, 1997, 176-205, 389-404.
[You01] R. R. You, *Effective Requirements Practices*: Addison-Wesley, 2001.

附录A 深入阅读的文献

(Ale02) I. Alexander and R. Stevens, *Writing Better Requirements*: Addison-Wesley, 2002.
(Ard97) M. Ardis, "Formal Methods for Telecommunication System Requirements: A survey of Standardized Languages," *Annals of Software Engineering*, vol. 3, 1997
(Ber97) V. Berzins and al, "A Requirements Evolution Model for Computer Aided Prototyping," presented at Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, 1997
(Bey95) H. Beyer and K. Holtzblatt, "Apprenticing with the Customer," *Communications of the ACM*, vol. 38, iss.5, 45-52, May, 1995
(Bru95) G. Bruno and R. Agarwal, "Validating Software Requirements Using Operational Models," presented at Second Symposium on Software Quality Techniques and Acquisition Criteria, Florence, Italy, 1995
(Bry94) E. Bryne, "IEEE Standard 830: Recommended Practice for Software Requirements Specification," presented at IEEE International Conference on Requirements Engineering, 1994
(Buc94) G. Bucci and al, "An Object-Oriented Dual Language for Specifying Reactive Systems," presented at IEEE International Conference on Requirements Engineering, 1994
(Bus95) D. Bustard and P. Lundy, "Enhancing Soft Systems Analysis with Formal Modeling," presented at Second International Symposium on Requirements Engineering, 1995
(Che94) M. Chechik and J. Gannon, "Automated Verification of Requirements Implementation," presented at Proceedings of the International Symposium on Software Testing and Analysis, Special Issue, 1994
(Chu95) L. Chung and B. Nixon, "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach," presented at Seventeenth IEEE International Conference on Software Engineering, 1995
(Cia97) P. Ciancarini and al, "Engineering Formal Requirements: An Analysis and Testing Method for Z Documents," *Annals of Software Engineering*, vol. 3, 1997
(Cre94) R. Crespo, "We Need to Identify the Requirements of the Statements of Non-Functional Requirements," presented at International Workshop on Requirements Engineering: Foundations of Software Quality, 1994
(Cur94) P. Curran and al, "BORIS-R Specification of the Requirements of a Large-Scale Software Intensive System," presented at Requirements Elicitation for Software-Based Systems, 1994
(Dar97) R. Darimont and J. Souquieres, "Reusing Operational Requirements: A Process-Oriented Approach," presented at IEEE International Symposium on Requirements Engineering, 1997
(Dav94) A. Davis and P. Hsia, "Giving Voice to Requirements Engineering: Guest

Editors' Introduction," rements Engineering: Guest Editors' Introduction," *IEEE Software*, vol. 11, iss. 2, 12-16, March, 1994
(Def94) J. DeFoe, "Requirements Engineering Technology in Industrial Education," presented at IEEE International Conference on Requirements Engineering, 1994
(Dem97) E. Demirors, "A Blackboard Framework for Supporting Teams in Software Development," presented at Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, 1997
(Die95) M. Diepstraten, "Command and Control System Requirements Analysis and System Requirements Specification for a Tactical System," presented at First IEEE International Conference on Engineering of Complex Computer Systems, 1995
(Dob94) J. Dobson and R. Strens, "Organizational Requirements Definition for Information Technology," presented at IEEE International Conference on Requirements Engineering, 1994
(Duf95) D. Duffy and al, "A Framework for Requirements Analysis Using Automated Reasoning," presented at Seventh International Conference on Advanced Information Systems Engineering, 1995
(Eas95) S. Easterbrook and B. Nuseibeh, "Managing Inconsistencies in an Evolving Specification," presented at Second International Symposium on Requirements Engineering, 1995
(Edw95) M. Edwards and al, "RECAP: A Requirements Elicitation, Capture, and Analysis Process Prototype Tool for Large Complex Systems," presented at First IEEE International Conference on Engineering of Complex Computer Systems, 1995
(ElE95) K. El-Emam and N. Madhavji, "Requirements Engineering Practices in Information Systems Development: A Multiple Case Study," presented at Second International Symposium on Requirements Engineering, 1995
(Fai97) R. Fairley and R. Thayer, "The Concept of Operations: The Bridge From Operational Requirements to Technical Specifications," *Annals of Software Engineering*, vol. 3, 1997
(Fic95) S. Fickas and M. Feather, "Requirements Monitoring in Dynamic Environments," presented at Second International Symposium on Requirements Engineering, 1995
(Fie95) R. Fields and al, "A Task-Centered Approach to Analyzing Human Error Tolerance Requirements," presented at Second International Symposium on Requirements Engineering, 1995
(Gha94) J. Ghajar-Dowlatshahi and A. Varnekar, "Rapid Prototyping in Requirements Specification Phase of Software Systems," presented at Fourth International Symposium on Systems Engineering, Sunnyvale, California:
National Council on Systems Engineering, 1994
(Gib95) M. Gibson, "Domain Knowledge Reuse During Requirements Engineering," presented at Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), 1995
(Gol94) L. Goldin and D. Berry, "AbstFinder: A Prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology and Evaluation," presented at IEEE International Conference on Requirements Engineering,

1994
(Got97) O. Gotel and A. Finkelstein, "Extending Requirements Traceability: Lessons Learned from an Industrial Case Study," presented at IEEE International Symposium on Requirements Engineering, 1997
(Hei96) M. Heimdahl, "Errors Introduced during the TACS II Requirements Specification Effort: A Retrospective Case Study," presented at Eighteenth IEEE International Conference on Software Engineering, 1996
(Hei96a) C. Heitmeyer and al, "Automated Consistency Checking Requirements Specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 5, iss. 3, 231-261, July, 1996
(Hol95) K. Holtzblatt and H. Beyer, "Requirements Gathering: The Human Factor," *Communications of the ACM*, vol. 38, iss. 5, 31-32, May, 1995
(Hud96) E. Hudlicka, "Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods," presented at Second IEEE International Conference on Requirements Engineering, 1996
(Hug94) K. Hughes and al, "A Taxonomy for Requirements Analysis Techniques," presented at IEEE International Conference on Requirements Engineering, 1994
(Hug95) J. Hughes and al, "Presenting Ethnography in the Requirements Process," presented at Second IEEE International Symposium on Requirements Engineering, 1995
(Hut94) A. T. F. Hutt, Ed., "Object Analysis and Design - Comparison of Methods. Object Analysis and Design - Description of Methods." John Wiley & Sons, 1994.
(INCOSE00) INCOSE, *How To: Guide for all Engineers, Version 2*: International Council on Systems Engineering, 2000.
(Jac95) M. Jackson, *Software Requirements and Specifications*. Reading, Massachusetts: Addison Wesley, 1995.
(Jac97) M. Jackson, "The Meaning of Requirements," *Annals of Software Engineering*, vol. 3, 1997
(Jon96) S. Jones and C. Britton, "Early Elicitation and Definition of Requirements for an Interactive Multimedia Information System," presented at Second IEEE International Conference on Requirements Engineering, 1996
(Kir96) T. Kirner and A. Davis, "Nonfunctional Requirements for Real-Time Systems," *Advances in Computers*, 1996
(Kle97) M. Klein, "Handling Exceptions in Collaborative Requirements Acquisition," presented at IEEE International Symposium on Requirements Engineering, 1997
(Kos97) R. Kosman, "A Two-Step Methodology to Reduce Requirements Defects," *Annals of Software Engineering*, vol. 3, 1997
(Kro95) J. Krogstie and al, "Towards a Deeper Understanding of Quality in Requirements Engineering," presented at Seventh International Conference on Advanced Information Systems Engineering (CAiSE '95), 1995
(Lal95) V. Lalioti and B. Theodoulidis, "Visual Scenarios for Validation of Requirements Specification," presented at Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, 1995
(Lam95) A. v. Lamsweerde and al, "Goal-Directed Elaboration of Requirements for a

Meeting Scheduler: Problems and Lessons Learnt," presented at Second International Symposium on Requirements Engineering, 1995
(Lei97) J. Leite and al, "Enhancing a Requirements Baseline with Scenarios," presented at IEEE International Symposium on Requirements Engineering, 1997
(Ler97) F. Lerch and al., "Using Simulation-Based Experiments for Software Requirements Engineering," *Annals of Software Engineering*, vol. 3, 1997
(Lev94) N. Leveson and al, "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, vol. 20, iss. 9, 684-707, September, 1994
(Lut96a) R. Lutz and R. Woodhouse, "Contributions of SFMEA to Requirements Analysis," presented at Second IEEE International Conference on Requirements Engineering, 1996
(Lut97) R. Lutz and R. Woodhouse, "Requirements Analysis Using Forward and Backward Search," *Annals of Software Engineering*, vol. 3, 1997
(Mac96) L. Macaulay, *Requirements Engineering*. London UK: Springer, 1996.
(Mai95) N. Maiden and al, "Computational Mechanisms for Distributed Requirements Engineering," presented at Seventh International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, 1995
(Mar94) B. Mar, "Requirements for Development of Software Requirements," presented at Fourth International Symposium on Systems Engineering, Sunnyvale, California, 1994
(Mas97) P. Massonet and A. v. Lamsweerde, "Analogical Reuse of Requirements Frameworks," presented at IEEE International Symposium on Requirements Engineering, 1997
(McF95) I. McFarland and I. Reilly, "Requirements Traceability in an Integrated Development Environment," presented at Second International Symposium on Requirements Engineering, 1995
(Mea94) N. Mead, "The Role of Software Architecture in Requirements Engineering," presented at IEEE International Conference on Requirements Engineering, 1994
(Mos95) D. Mostert and S. v. Solms, "A Technique to Include Computer Security, Safety, and Resilience Requirements as Part of the Requirements Specification," *Journal of Systems and Software*, vol. 31, iss. 1, 45-53, October, 1995
(Myl95) J. Mylopoulos and al, "Multiple Viewpoints Analysis of Software Specification Process," *IEEE Transactions on Software Engineering*, 1995
(Nis92) K. Nishimura and S. Honiden, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach," *IEEE Transactions on Software Engineering*, December, 1992
(Nis97) H. Nissen and al, "View-Directed Requirements Engineering: A Framework and Metamodel," presented at Ninth IEEE International Conference on Software Engineering and Knowledge Engineering, Skokie, Illinois: Knowledge Systems Institute, 1997
(OBr96) L. O'Brien, "From Use Case to Database: Implementing a Requirements Tracking System," *Software Development*, vol. 4, iss. 2, 43-47, February, 1996
(UML04) Object Management Group, "Unified Modeling Language," 2004, available at

http://www.uml.org
(Opd94) A. Opdahl, "Requirements Engineering for Software Performance," presented at International Workshop on Requirements Engineering: Foundations of Software Quality, 1994
(Pin96) F. Pinheiro and J. Goguen, "An Object-Oriented Tool for Tracing Requirements," *IEEE Software*, vol. 13, iss. 2, 52-64, March, 1996
(Pla96) G. Playle and C. Schroeder, "Software Requirements Elicitation: Problems, Tools, and Techniques," *Crosstalk: The Journal of Defense Software Engineering*, vol. 9, iss. 12, 19-24, December, 1996
(Poh94) K. Pohl and al, "Applying AI Techniques to Requirements Engineering: The NATURE Prototype," presented at IEEE Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence, 1994
(Por95) A. Porter and al, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Transactions on Software Engineering*, vol. 21, iss. 6, 563-575, June, 1995
(Pot95) C. Potts and al, "An Evaluation of Inquiry-Based Requirements Analysis for an Internet Server," presented at Second International Symposium on Requirements Engineering, 1995
(Pot97) C. Potts and I. Hsi, "Abstraction and Context in Requirements Engineering: Toward a Synthesis," *Annals of Software Engineering*, vol. 3, 1997
(Pot97a) C. Potts and W. Newstetter, "Naturalistic Inquiry and Requirements Engineering: Reconciling Their Theoretical Foundations," presented at IEEE International Symposium on Requirements Engineering, 1997
(Ram95) B. Ramesh and al, "Implementing Requirements Traceability: A Case Study," presented at Second International Symposium on Requirements Engineering, 1995
(Reg95) B. Regnell and al, "Improving the Use Case Driven Approach to Requirements Engineering," presented at Second IEEE International Symposium on Requirements Engineering, 1995
(Reu94) H. Reubenstein, "The Role of Software Architecture in Software Requirements Engineering," presented at IEEE International Conference on Requirements Engineering, 1994
(Rob94) J. Robertson and S. Robertson, "Complete Systems Analysis," vol. 1 and 2. Englewood Cliffs, New Jersey: Prentice Hall, 1994.
(Rob94a) W. Robinson and S. Fickas, "Supporting Multi-Perspective Requirements Engineering," presented at IEEE International Conference on Requirements Engineering, 1994
(Ros98) L. Rosenberg, T. F. Hammer and L. L. Huffman, "Requirements, testing and metrics," presented at 15th Annual Pacific Northwest Software Quality Conference, Utah, 1998
(Sch94) W. Schoening, "The Next Big Step in Systems Engineering Tools: Integrating Automated Requirements Tools with Computer Simulated Synthesis and Test," presented at Fourth International Symposium on Systems Engineering, Sunnyvale, California, 1994
(She94) M. Shekaran, "The Role of Software Architecture in Requirements

Engineering," presented at IEEE International Conference on Requirements Engineering, 1994
(Sid97) J. Siddiqi and al, "Towards Quality Requirements Via Animated Formal Specifications," *Annals of Software Engineering*, vol. 3, 1997
(Span97) G. Spanoudakis and A. Finkelstein, "Reconciling Requirements: A Method for Managing Interference, Inconsistency, and Conflict," *Annals of Software Engineering*, vol. 3, 1997
(Ste94) R. Stevens, "Structured Requirements," presented at Fourth International Symposium on Systems Engineering, Sunnyvale, California, 1994
(Vin90) W. G. Vincenti, *What Engineers Know and How They Know It - Analytical Studies form Aeronautical History*. Baltimore and London: John Hopkins University Press, 1990.
(Wei03) K. Weigers, *Software Requirements*, 2nd ed: Microsoft Press, 2003.
(Whi95) S. White and M. Edwards, "A Requirements Taxonomy for Specifying Complex Systems," presented at First IEEE International Conference on Engineering of Complex Computer Systems, 1995
(Wil99) B. Wiley, *Essential System Requirements: A Practical Guide to Event-Driven Methods*: Addison-Wesley, 1999.
(Wyd96) T. Wyder, "Capturing Requirements With Use Cases," *Software Development*, vol. 4, iss. 2, 36-40, February, 1996
(Yen97) J. Yen and W. Tiao, "A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements," presented at IEEE International Symposium on Requirements Engineering, 1997
(Yu97) E. Yu, "Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering," presented at IEEE International Symposium on Requirements Engineering, 1997
(Zav96) P. Zave and M. Jackson, "Where Do Operations Come From? A Multiparadigm Specification Technique," *IEEE Transactions on Software Engineering,*, vol. 22, iss. 7, 508-528, July, 1996

附录B 有关标准

(IEEE830-98) IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements Specifications*: IEEE, 1998.
(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*: IEEE, 1997.
(IEEE1233-98) IEEE Std 1233-1998, "IEEE Guide for Developing System Requirements Specifications," 1998
(IEEE1320.1-98) IEEE Std 1320.1-1998, *IEEE Standard for Functional Modeling Language-Syntax and Semantics for IDEF0*: IEEE, 1998.
(IEEE1320.2-98) IEEE Std 1320.2-1998, "IEEE Standard for Conceptual Modeling Language-Syntax and Semantics for IDEFIX97 (IDEF Objetct)," IEEE, 1998.
(IEEE1362-98) IEEE Std 1362-1998, *IEEE Guide for Information Technology-System Definition-Concept of Operations (ConOps) Document*: IEEE, 1998.
(IEEE1465-98) IEEE Std 1465-1998//ISO/IEC12119:1994, *IEEE Standard Adoption of*

International Standard ISO/IEC12119:1994(E), Information Technology-Software packages-Quality requirements and testing: IEEE, 1998.
(IEEEP1471-00) IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Description of Software Intensive Systems*: Architecture Working Group of the Software Engineering Standards Committee, 2000.
(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, IEEE, 1996.
(IEEE14143.1-00) IEEE Std 14143.1-2000//ISO/IEC14143-1:1998, *Information Technology-Software Measurement-Functional Size Measurement-Part 1: Definitions of Concepts*: IEEE, 2000.

//

第3章　软件设计

术语和缩写

ADL：Architecture Description Languages，体系结构描述语言
CRC：Collaboration Responsibilities Card，协作责任卡
ERD：Entity-Relationship Diagrams，实体联系图
IDL：Interface Description Languages，接口描述语言
DFD：Data Flow Diagram，数据流图
PDL：Pseudo-Code and Program Design Languages，伪码与程序描述语言
CBD：Component-based design，基于组件的设计

引言

在文献[IEEE610.12-90]中，设计被定义为“定义一个系统或组件的体系结构、组件、接口和其它特征的过程”和“这个过程的结果”。作为过程看待时，软件设计是一种软件工程生命周期活动，在这个活动中，要分析软件需求，以产生一个将作为软件构造的基础的软件内部结构的描述。更精确地说，软件设计（结果）必须描述软件体系结构（即，软件如何分解成组件并组织起来）和这些组件之间的接口，它必须在详细的层次上描述组件，以便能构造这些组件。

软件设计在软件开发中起着重要作用:它让软件工程师产生形成要实现的方案的蓝图的各种不同的模型,我们可以分析和评价这些模型,以确定使用它们能否实现各种不同的需求,我们可以检查和评价各种不同的候选方案，进行权衡，最后，除了作为构造和测试的输入和起始点外，我们可以使用作为结果的模型，来规划后续的开发活动。

在《IEEE/EIA 12207软件生命周期过程》[IEEE12207.0-96]等软件生命周期过程的标准列表中，软件设计由两个处于软件需求和软件构造之间的活动组成：（1）软件体系结构设计（有时叫做高层设计）：描述软件的搞成结构和组织，标识各种不同的组件。（2）软件详细设计：详细地描述各个组件，使之能被构造。

对于软件设计知识域的范围，目前的知识域描述并没有讨论每一个其名字包含“设计”一词的主题。在Tom DeMarco的术语中(DeM99)，本章讨论的知识域主要处理D设计（decomposition design：分解设计，将软件映射到组件上）。

但是，由于日益增长的软件体系结构领域的重要性，我们也将讨论FP设计（family pattern design：族模式设计，目标是在一族软件中建立可利用的公共部分）。但是，软件

设计知识域没有包含I设计（invention design：创造性设计，通常在软件需求过程中完成，目标是将软件概念化，并说明软件能满足已经发现的要求和需求），因为这个主题是需求分析和规格说明的一部分。

软件设计知识域描述与软件需求、软件构造、软件工程管理、软件指令和软件工程相关学科特别相关。

软件设计主题分解结构

1 软件设计基础

这里介绍的概念、符号和术语形成里理解软件设计的作用和范围的基础。

1.1 一般设计概念

软件并不是有设计的唯一领域。一般而言，我们可以将设计看作是一种问题求解[Bud03:c1],例如,难题(没有确定性答案的问题)对于理解设计的限制就很有趣[Bud04:c1]，一般意义上，其它许多符号和概念对于理解设计也有趣：目标、约束、候选方案、代表、答案[Smi93]。

1.2 软件设计上下文

为理解软件设计的作用，理解其所处的上下文—软件工程生命周期--很重要，这样，理解软件需求分析与软件设计与软件构造与软件测试，就非常重要。[IEEE12207.0-96; Lis01:c11; Mar02; Pfl01:c2;Pre04:c2].

1.3 软件设计过程

通常认为软件设计是一个两步的过程[Bas03; Dor02:v1c4s2; Fre83:I; IEEE12207.0-96]; Lis01:c13; Mar02:D]：

1.3.1 体系结构设计

体系结构设计描述软件如何分解和组织成组件（软件体系结构）[IEEEP1471-00]。

1.3.2 详细设计

详细设计描述这些组件的特定行为。

软件设计过程的输出是一组模型和人造物品，它们记录了采用的主要决策[Bud04:c2; IEE1016-98; Lis01:c13; Pre04:c9]。

1.4 使能技术

根据牛津英语词典，“原理”是“一个基本的真理或普遍的法则，用来作为推理的基础或行动的指南”。软件设计原理又称为使能技术[Bus96]，是针对许多不同的软件设计方法和概念的关键观念。使能技术包括[Bas98:c6; Bus96:c6; IEEE1016-98; Jal97:c5,c6; Lis01:c1,c3; Pfl01:c5; Pre04:c9]：

1.4.1 抽象

抽象是“遗忘一些信息，使得不同的事物可以当作同样的事物来处理”[Lis01]。在软件设计上下文中，有两个关键的抽象机制：参数化和规范。规范抽象主要由类：过程抽象、

数据抽象和控制（迭代）抽象[Bas98:c6; Jal97:c5,c6; Lis01:c1,c2,c5,c6; Pre04:c1]。

1.4.2 耦合与聚合

耦合定义为模块之间相互联系的强度，聚合定义为组成一个模块的各个元素如何相互联系[Bas98:c6; Jal97:c5; Pfl01:c5; Pre04:c9]。

1.4.3 分解与模块化

将大型软件分解和模块化为大量小规模的独立模块，一般目标是将不同的功能或责任放置到不同的组件中[Bas98:c6; Bus96:c6; Jal97 :c5; Pfl01:c5; Pre04:c9]。

1.4.4 封装与信息隐藏

封装与信息隐藏意味着将元素和抽象的细节分组打包，使得外部不能访问细节[Bas98:c6; Bus96:c6; Jal97:c5; Pfl01:c5; Pre04:c9]。

1.4.5 接口和实现的分离

接口和实现的分离涉及通过规格说明一个公共接口（称为客户）来定义一个组件，并将如何实现细节分离出来[Bas98:c6; Bos00:c10; Lis01:c1,c9]。

1.4.6 充分性、完备性和原始性

要实现充分性、完备性和原始性，就要保证一个软件组件包括了一个抽象的所有重要特征，并且没有多余的特征[Bus96:c6; Lis01:c5]。

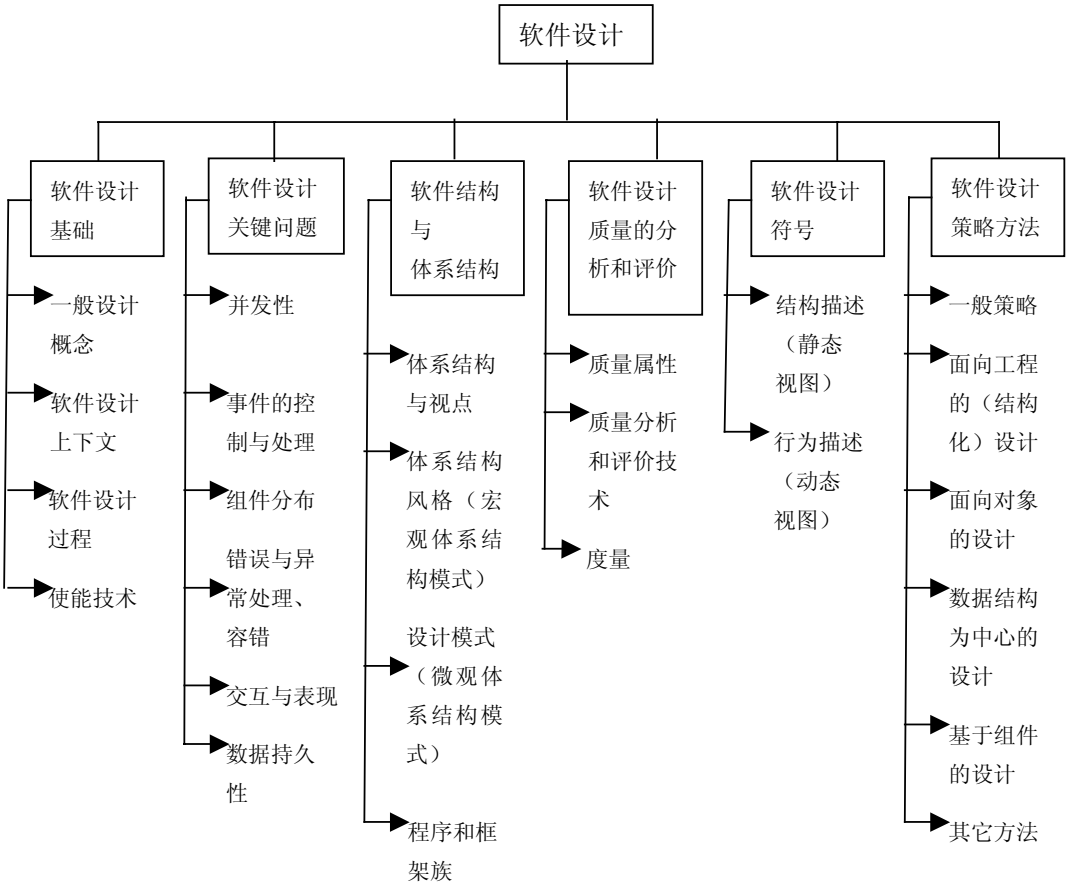

图 1　软件设计知识域的主题分解结构

2 软件设计关键问题

设计软件时，必须处理许多关键问题。一些是所有软件都必须处理的质量问题，例如，性能。另一个重要问题是软件组件的分解、组织、打包方法，这个问题很基本，所有设计方法都必须以某种方式处理它（参见主题1.4使能技术和软件设计策略和方法子域）。其它的问题“处理软件行为的某些不在应用领域内的方面，而这些行为涉及支持领域”[Bos00]。这些问题通常与系统的功能性横断相交，被成为剖面（aspect），“剖面一般不是软件功能分解的单元，而是以系统的方式影响组件的性能和语义”(Kic97)。下面是一些关键的、横断问题：

2.1 并发性

如何将软件分解为多个进程、任务和线程，以处理相关的效率、原子性、同步和调度问题 [Bos00:c5; Mar02:CSD; Mey97:c30; Pre04:c9]。

2.2 实践的控制与处理

如何组织数据和控制流，如何通过不同的机制（如隐含调用和回调）处理交互和临时事件 [Bas98:c5; Mey97:c32; Pfl01:c5]。

2.3 组件的分布

如何将软件分布到各个硬件上，组件如何通信，如何使用中间件来处理异构软件

[Bas03:c16; Bos00:c5; Bus96:c2 Mar94:DD; Mey97:c30; Pre04:c30]。

2.4 错误和异常处理、容错

如何阻止和容许故障，如何处理异常条件[Lis01:c4; Mey97:c12; Pfl01:c5]。

2.5 交互和表现

如何设计结构和组织来实现与用户的交互、信息的表现形式（例如，使用模型-控制器-视图方法，将表现与业务逻辑分离）[Bas98:c6; Bos00:c5; Bus96:c2; Lis01:c13; Mey97:c32] 。应该指出，这个主题不是关于如何规格说明用户接口细节的，后者是用户接口设计（软件人类工程学的一部分）的任务，参见软件工程相关学科。

2.6 数据持久性

如何处理需要长时间生存的数据[Bos00:c5; Mey97:c31]。

3 软件结构与体系结构

严格地说，软件体系结构是“一个描述软件系统的子系统和组件，以及它们之间相互关系的学科”(Bus96:c6)。体系结构试图定义结果软件的内部结构（根据牛津英语词典，“结构”是某个事物被构造和组织的方式）。但是，在1990年代中期，软件体系结构开始作为一个更广泛的学科出现了，它涉及以更一般的方式研究软件结构和体系结构[Sha96]，这引发了大量有趣的关于在不同抽象层次上设计软件的思想，其中一些概念在特定软件的体系结构设计中有用（例如，体系结构风格），也在详细设计中有用（例如，低层的设计模式）。但它们也可以用于设计一般的系统，导致了程序族（也称为产品线）的设计。有趣的是，可以认为，多数这些概念是试图描述或复用一般的设计知识。

3.1 体系结构和视图

可以描述软件设计的不同高层剖面，并形成文档，这些剖面通常称为视图：“一个视图表示了软件体系结构的显示软件系统某个特定特性的某个特定方面”[Bus96:c6]。这些不同的图点与关联软件设计的不同问题相关，例如：逻辑视图（满足功能需求）与进程图点（并发问题）、物理视图（分布问题）与开发视图（设计如何分解为实现单元）。其它作者使用了不同的术语，例如，行为、功能、结构和数据模型视图。总之，软件设计是多剖面的产品，它由设计过程产生，并由一些相对独立和正交的视图组成[Bas03:c2; Boo99:c31; Bud04:c5; Bus96:c6; IEEE1016-98; IEEE1471-00]。

体系结构风格（宏观体系结构模式）：体系结构风格是“施加在一个体系结构上的、定义一组或一族满足它们的体系结构的一组约束”[Bas03:c2]。体系结构风格可以被认为是提供软件高层组织（宏观体系结构）的元模型。不同的作者已经标识了大量的主要体系结构风格[Bas03:c5; Boo99:c28; Bos00:c6; Bus96:c1,c6; Pfl01:c5]：（1）一般结构（例如，分层、管道线、过滤器、黑板）；（2）分布式系统（例如，客户/服务器、3级结构、代理）；（3）交互式系统（例如，模型-视图-控制器、表现-抽象-控制）；（4）自适应系统（例如，微内核、反射）；（5）其它（例如，批处理、解释器、过程控制、基于规则）。

3.2 设计模式（微观体系结构模式）

简洁地讲，模式是“给定上下文中普遍问题的普遍解决方案”(Jac99)。体系结构风格可以被认为是描述软件高层组织的模式（宏观体系结构），其它设计模式可用于描述较低层次的、更局部的细节（微观体系结构）[Bas98:c13; Boo99:c28; Bus96:c1; Mar02:DP]：（1）

创建型模式（例如，构造者、工厂、原型、单件）；（2）结构型模式（例如，适配器、桥接器、组合、装饰器、剖面、蝇量、代理）；（3）行为型模式（例如，命令、解释器、迭代器、协调器、备忘录、观察者、状态、策略、模板、访问者）。

3.3 程序和框架族

另一个复用软件设计和组件的途径是设计程序族，又成为软件产品线。标识族中成员的公共特性，使用可复用可裁剪组件来解决族内成员之间的可变性问题，就可以实现程序族[Bos00:c7,c10; Bas98:c15; Pre04:c30]。

在面向对象编程中，一个关键概念是框架：可以通过适当的特定插件（又成为热点）实例化的、部分完成的软件子系统[Bos00:c11; Boo99:c28; Bus96:c6]

4 软件设计质量的分析与评价

本节包含大量的与软件设计特别相关的质量和评价主题，多数主题也包含在软件质量知识域中。

4.1 质量属性

对于得到一个高质量（可维护性、可移植性、可测试性、可追踪性、正确性、健壮性、目的的适应性)的软件设计，多种质量属性都认为是重要的[Bos00:c5; Bud04:c4; Bus96:c6; ISO9126.1-01; ISO15026-98; Mar94:D; Mey97:c3; Pfl01:c5]。一个有趣的区分是在运行时间可区别的质量属性（性能、安全性、可用性、功能性、可使用性）、运行时间不可区别的质量属性（可修改性、可移植性、可复用性、可集成性、可测试性）、与体系结构本质质量相关的质量属性（概念完整性、正确性、完备性和可构造性）[Bas03:c4]。

4.2 质量分析与评价技术

有多种工具和技术来帮助人们确保软件设计的质量。

（1）软件设计评审：有正式的和半正式的，通常是以小组方式进行，来验证和保证设计结果的质量（例如，体系结构评审[Bas03:c11]、设计评审和检查[Bud04:c4; Fre83:VIII; IEEE1028-97; Jal97:c5,c7; Lis01:c14; Pfl01:c5]、基于场景的技术[Bas98:c9; Bos00:c5]、需求追踪[Dor02:v1c4s2; Pfl01:c11]）。

（2）静态分析：正式或半正式的静态（不可执行的）分析技术，可以用于评价一个设计（例如，故障树分析或自动交叉检查）[Jal97:c5; Pfl01:c5]。

（3)模拟与原型:这是评价设计的动态的技术(例如,性能模拟或可行性原型[Bas98:c10; Bos00:c5; Bud04:c4; Pfl01:c5]）。

4.3 度量

使用度量可以评定或定量估计软件设计的不同方面：规模、结构、质量。已经提出了许多度量，它们多数依赖产生设计的方法。这些度量可以分为两类。

（1）面向功能（结构化）设计的度量：通过功能分解得到的设计结构，通常表示为结构图（有时称为层次图），可以计算其多种度量[Jal97:c5,c7, Pre04:c15]。

（2）面向对象设计度量：设计的总体结构通常表示为类图，可以计算多种度量，也可以计算每个类内部的内容的度量[Jal97:c6,c7; Pre04:c15]。

5 软件设计符号

有许多表达软件设计成果的符号和语言，一些主要用于描述设计的结构组织，另一些用

于描述软件行为。某些符号主要在体系结构设计中使用，另一些则主要用于详细设计，少部分可以用于这两个步骤。另外，一些符号通常用于特定方法的上下文中（参见软件设计策略与方法子域）。这里，我们将符号分类为描述结构（静态）视图和行为（动态）视图两类。

5.1 结构描述（静态视图）

下面的符号，主要是（但不总是）图形，描述和表示软件设计的结构方面，即，它们描述主要的组件和组件间的联系（静态视图）：

（1）体系结构描述语言（Architecture description languages，ADL）：文本（通常是形式化的）语言，以组件和组件间相互联系的方式描述软件体系结构[Bas03:c12]。

（2）类图和对象图：用于表示类或对象的集合，以及它们之间的联系[Boo99:c8,c14; Jal97:c5,c6]。

（3）组件图：用于表示组件（[遵从和提供]一组接口的实现的系统的物理的和替代的部分）集合和组件间联系[Boo99:c12,c31]。

（4）协作责任卡（Collaboration responsibilities cards，CRC）：用于表示组件（类）的名称、责任和协作组件名称[Boo99:c4; Bus96]。

（5）部署图：用于表示一组（物理）节点，及其相互联系，表示了系统的物理外观[Boo99:c30]。

（6）实体联系图：用于表示存储在信息系统中数据的模型[Bud04:c6; Dor02:v1c5; Mar02:DR]。

（7）接口描述语言（Interface description languages，IDL）：类编程语言，用于定义软件组件的接口（输出的操作的名字和类型）[Bas98:c8; Boo99:c11]。

（8）Jackson结构图：以顺序、选择和重复的方式描述数据结构[Bud04:c6; Mar02:DR]。

（9）结构图：用于描述程序的调用结构（一个模块调用的其它模块，调用某个模块的其它模块）[Bud04:c6; Jal97:c5; Mar02:DR;Pre04:c10]。

5.2 行为描述（动态视图）

下面的符号和语言，一些是图形的，一些是文本的，用于描述软件和组件的动态行为。多数符号用于详细设计。

（1）活动图：用于表示从活动（在一个状态机内进行的非原子执行）到活动的控制流[Boo99:c19]。

（2）协作图：用于表示发生在一组对象之间的交互，重点是对象、对象的链接、对象在链接上交换的消息[Boo99:c18]。

（3）数据流图：用来表示数据在一组处理过程之间的流动[Bud04:c6; Mar02:DR;Pre04:c8]。

（4）决策表和决策图：用于表示条件和行动的复杂组合[Pre04:c11]。

（5）流程图和结构化流程图：用于表示控制流和要完成的对应活动[Fre83:VII; Mar02:DR; Pre04:c11]。

（6）序列图：用于表示一组对象之间的交互，重点在按时间顺序的消息交换[Boo99:c18]。

（7）状态变迁与状态图：用于表示状态机中，状态之间的控制流[Boo99:c24; Bud04:c6; Mar02:DR; Jal97:c7]。

（8）形式化描述语言：文本语言，使用来自数学的基本符号（例如，逻辑、集合、顺序）来严格和抽象地定义软件组件接口和行为，通常形式是前置条件和后置条件[Bud04:c18; Dor02:v1c6s5; Mey97:c11]。

（9）伪码和程序设计语言：结构化的、类编程语言，通常在详细设计阶段，用于描述一个过程或方法的行为[Bud04:c6; Fre83:VII;Jal97:c7; Pre04:c8, c11]。

6 软件设计策略与方法

有各种一般的策略来帮助指导设计过程[Bud04:c9, Mar02:D]。与一般的策略不同，方法则更为专门，方法通常建议和提供了与方法一起使用的一组符号，并描述了遵循方法时要使用的过程，以及使用方法的指南[Bud04:c8]。这些方法作为传递知识的手段和作为软件工程师小组的公共构架，很有用处[Bud03:c8]。可以参见软件工程工具与方法知识域。

6.1 一般策略

常常被引用的设计过程中有用的一般策略是分而治之和逐步求精[Bud04:c12; Fre83:V]、子顶向下与自底向上[Jal97:c5; Lis01:c13]、数据抽象与信息隐藏[Fre83:V]、使用启发式规则[Bud04:c8]、使用模式和模式语言[Bud04:c10; Bus96:c5]、使用迭代和增量方法[Pfl01:c2]。

6.2 面向功能（结构化）设计[Bud04:c14; Dor02:v1c6s4; Fre83:V; Jal97:c5;Pre04:c9, c10]

这是软件设计的一个经典方法，分解的中心集中在标识主要的软件工程上，然后以自顶向下的方式，不断详细描述和精化这些功能。结构化设计通常在结构化分析后进行，产生数据流图对应的过程描述。研究人员提出了各种策略（例如，变换分析和事务分析）和启发式方法（例如，扇入/扇出、影响范围/控制范围）来将一个数据流图变换为通常用结构图表示的软件体系结构。

6.3 面向对象的设计[Bud0:c16; Dor02:v1:c6s2,s3; Fre83:VI; Jal97:c6;Mar02:D; Pre04:c9]

已经提出了许多基于对象的软件设计方法，这个领域从1980年代中期的基于对象的设计（名词=对象，动词=方法、形容词=属性）发展到面向对象的设计，其中，继承和多态性起着关键的作用，在发展到基于组件的设计，其中，可以定义和访问元信息（例如，通过反射）。虽然面向对象设计源于数据抽象，人们提出了责任驱动的设计作为面向对象设计的另一个选择。

6.4 数据结构为中心的设计[Bud04:c15; Fre83:III,VII; Mar02:D]

数据结构为中心的设计（例如Jackson方法、Warnier-Orr方法）从程序要操纵的数据结构开始，而不是从程序要完成的功能开始。软件工程师首先描述输入输出的数据结构（例如，使用Jackson的结构图），然后基于这些数据结构图来开发程序的控制结构。人们提出了各种启发式规则来处理特殊情况，例如，当输入结构与输出结构不匹配时的情况。

6.5 基于组件的设计

一个软件组件是一个独立的单元，具有良定义的接口和可以独立组合和部署的依赖性。基于组件的设计要解决为了改进复用而提供、开发、集成这些组件有关的问题[Bud04:c11]。

6.6 其它方法

还有一些其它有趣但非主流的方法：形式化和严密的方法[Bud04:c18; Dor02:c5; Fre83; Mey97:c11; Pre04:c29]和变换方法[Pfl98:c2]。

主题与参考资矩阵

	[Bas03]{Bas98}	[Boo99]	[Bos00]	[Bud03]	[Bus96]	[Dor02]	[Fre83]	[IEEE 1016-98]	[IEEE 1028-97]	[IEEE 1471-00]
1软件设计基础										
1.1一般设计概念				C1						
1.2软件设计上下文										
1.3软件设计过程	C2s1, c2s4			C2		V1c4s3	2-22	*		*
1.4使能技术	{c6s1}		C10s3		C6s3			*		
2软件设计关键问题										
2.1并发性			C5s4.1							
2.2事件的控制和处理	{c5s2}									
2.3组件分布	C16s2,c16s4		C5s4.1		C2s3					
2.4错误和异常处理、容错										
2.5交互和表现	{c6s2}		C5s4.1		C2s4					
2.6数据持久性			C5s4.1							
3软件结构与体系结构										
3.1体系结构与视点	C2s5	C31		C5	C6s1			*		*
3.2体系结构风格	C5s9	C28	C6s3.1		C1s1-c1s3, c6s2					
3.3设计模式	{c13s3}	C28			C1s1-c1s3					
3.4程序和	{c15s	C28	C7s1,		C6s2					

框架族	1,c15 s3}		c7s2, c10s2 -c10s 4,c11 s2,c1 1s4							
4软件设计质量的分析和评价										
4.1质量属性	C4s2		C5s2. 3	C4	C6s4					
4.2质量分析与评价技术	C11s3 ,{c9s 1,c9s 2,c10 s2,c1 0s3}		C5s2. 1,c5s 2.2,c 5s3,c 5s4	C4		V1c4s 2	542-5 76		*	
4.3度量										
5软件设计符号										
5.1结构描述（静态视图）	{c8s4 }C12S 1,C12 S2	C4,C8 ,C11, C12,C 14,C3 0,c31		C6	429					
5.2行为描述（动态视图）		C18,c 19,c2 4		C6,c1 8		V1c5	485-4 90,50 6-513			
6软件设计策略与方法										
6.1一般策略				C8,c1 0,c12	C5s1- c5s4		304-3 20,53 3-539			
6.2面向功能（结构化）设计				C14		V1c6s 4	328-3 52			
6.3面向对象设计				C16		V1c6s 2,v1c 6s3	420-4 36			
6.4数据结构为中心的设计				C15			201-2 10,51 4-532			
6.5基于组				C11						

件的设计										
6.6其它方法				C18		181-192	395-407			

	[IEEE 12207.0-96]	[ISO9126-01]	[ISO15026-98]	[Jal97]	[Lis01]	[Mar02]*{Mar94}	[Mey97]	[Pfl01]	[Pre04]	[Smi93]
1软件设计基础										
1.1一般设计概念										*
1.2软件设计上下文	*				C11s1	D		C2s2	C2	
1.3软件设计过程	*				C13s1,c13s2	D			C9	
1.4使能技术				C5s1, c5s2, c6s2	C1s1, c1s2, c3s1-c3s3, 77-85, c5s8, 125-128, c9s1-c9s3			C5s2, c5s5	C9	
2软件设计关键问题										
2.1并发性						CSD	C30		C9	
2.2事件的控制和处理							C32s4,c32s5	C5s3		
2.3组件分布						{DD}	C30		C30	
2.4错误和异常处理、容错					C4s3-c4s5		C12	C5s5		
2.5交互和表现					C13s3		C32s2			
2.6数据持久性							C31			
3软件结构										

与体系结构										
3.1体系结构与视点										
3.2体系结构风格								C5s3		
3.3设计模式						DP				
3.4程序和框架族									C30	
4软件设计质量的分析和评价										
4.1质量属性		*	*			{D}	C3	C5s5		
4.2质量分析与评价技术				C5s5, c7s3	C14s1			C5s6, c5s7, c11s5		
4.3度量				C5s6, c6s5, c7s4					C15	
5软件设计符号										
5.1结构描述（静态视图）				C5s3, c6s3		DR			C10	
5.2行为描述（动态视图）				C7s2		DR	C11		C8,c11	
6软件设计策略与方法										
6.1一般策略				C5s1.4	C13s13			C2s2		
6.2面向功能（结构化）设计				C5s4					C9,c10	
6.3面向对象设计				C6s4		D			C9	
6.4数据结构为中心的设计						D				
6.5基于组件的设计										

6.6其它方法							C11	C2s2	C29	

推荐的软件设计参考文献

[Bas98] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*: Addison-Wesley, 1998.

[Bas03] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Second ed: Addison-Wesley, 2003.

[Boo99] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, First ed: Addison-Wesley, 1999.

[Bos00] J. Bosch, *Design & Use of Software Architectures: Adopting and Evolving a Product-line Approach*, First ed: ACM Press, 2000.

[Bud04] D. Budgen, *Software Design*, Second ed: Addison-Wesley, 2004.

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*, First ed: John Wiley & Sons, 1996.

[Dor02] M. Dorfman and R. H. Thayer, Eds., "Software Engineering." (Vol. 1 & vol. 2), IEEE Computer Society Press, 2002.

[Fre83] P. Freeman and A. I. Wasserman, *Tutorial on Software Design Techniques*, Fourth ed: IEEE Computer Society Press, 1983.

[IEEE610.12-90] IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*: IEEE, 1990.

[IEEE1016-98] IEEE Std 1016-1998, *IEEE Recommended Practice for Software Design Descriptions*: IEEE, 1998.

[IEEE1028-97] IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*: IEEE, 1997.

[IEEE1471-00] IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Descriptionos Software Intensive Systems*: Architecture Working Group of the Software Engineering Standards Committee, 2000.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, vol. IEEE, 1996.

[ISO9126-01] ISO/IEC 9126-1:2001, *Software Engineering-Product Quality-Part 1: Quality Model*: ISO and IEC,2001.

[ISO15026-98] ISO/IEC 15026-1998, *Information technology-- System and software integrity levels*: ISO and IEC, 1998.

[Jal97] P. Jalote, *An Integrated Approach to Software Engineering*, Second ed. New York: Springer-Verlag, 1997.

[Lis01] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, First ed: Addison-Wesley, 2001.

[Mar94] J. J. Marciniak, *Encyclopedia of Software Engineering*: J. Wiley & Sons, 1994.

The references to the Encyclopedia are as follows:

CBD = Componenent-based Design

D = Design
DD = Design of the Distributed System
DR = Design Representation
[Mar02] J. J. Marciniak, *Encyclopedia of Software Engineering*, Second ed: J. Wiley & Sons, 2002.
[Mey97] B. Meyer, *Object-Oriented Software Construction*, Second ed: Prentice-Hall, 1997.
[Pfl01] S. L. Pfleeger, *Software Engineering: Theory and Practice*, Second ed: Prentice-Hall, 2001.
[Pre04] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, Sixth ed: McGraw-Hill, 2004.
[Smi93] G. Smith and G. Browne, "Conceptual foundations of design problem-solving," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, iss. 5, 1209-1219, Sep-Oct, 1993.

附录A　深入阅读的文献

(Boo94a) G. Booch, *Object Oriented Analysis and Design with Applications*, Second ed: The Benjamin/Cummings Publishing Company, Inc., 1994.
(Coa91) P. Coad and E. Yourdon, *Object-Oriented Design*: Yourdon Press, 1991.
(Cro84) N. Cross, *Developments in Design Methodology*: John Wiley & Sons, 1984.
(DSo99) D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML - The Catalysis Approach*: Addison-Wesley, 1999.
(Dem99) T. DeMarco, "The Paradox of Software Architecture and Design," *Stevens Prize Lecture*, August, 1999
(Fen98) N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, Second ed: International Thomson Computer Press, 1998.
(Fow99) M. Fowler, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.
(Fow03) M. Fowler, *Patterns of Enterprise Application Architecture*, First ed. Boston, MA: Addison-Wesley, 2003.
(Gam95) E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.
(Hut94) A. T. F. Hutt, *Object Analysis and Design - Comparison of Methods. Object Analysis and Design - Description of Methods*: John Wiley & Sons, 1994.
(Jac99) I. Jacobson, G. Booch and J. Rumbaugh, *The Unified Software Development Process*: Addison-Wesley, 1999.
(Kic97) G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, "Aspect-oriented programming," presented at ECOOP '97 - Object-Oriented Programming, 1997
(Kru95) P. B. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, iss. 6, 42-50, 1995
(Lar98) C. Larman, *Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design*: Prentice-Hall, 1998.
(McC93) S. McConnell, *Code Complete: A Practical Handbook of Software Construction*:

Microsoft Press, 1993.
(Pag00) M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*: Addison-Wesley, 2000.
(Pet92) H. Petroski, *To Engineer is Human: The role of failure in successful design*: Vintage Books, 1992.
(Pre95) W. Pree, *Design Patterns for Object-Oriented Software Development*: Addison-Wesley and ACM Press, 1995.
(Rie96) A. J. Riel, *Object-Oriented Design Heuristics*: Addison-Wesley, 1996.
(Rum91) J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*: Prentice-Hall, 1991.
(Sha96) M. Shaw and D. Garlan, *Software architecture: Perspectives on an emerging discipline*: Prentice-Hall, 1996.
(Som05) I. Sommerville, *Software Engineering*, Sixth ed: Addison-Wesley, 2001.
(Wie98) R. Wieringa, "A Survey of Structured and Object: Oriented Software Specification Methods and Techniques," *ACM Computing Surveys*, vol. 30, iss. 4, 459-527, 1998
(Wir90) R. Wirfs-Brock, B. Wilkerson and L. Wiener, *Designing Object-Oriented Software*: Prentice-Hall, 1990

附录B　有关标准

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*: IEEE, 1990.
(IEEE1016-98) IEEE Std 1016-1998, *IEEE Recommended Practice for Software Design Descriptions*: IEEE, 1998.
(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*: IEEE, 1997.
(IEEE1471-00) IEEE Std 1471-2000, *IEEE Recommended Practice for Architectural Descriptionos Software Intensive Systems*: Architecture Working Group of the Software Engineering Standards Committee, 2000.
(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, vol. IEEE, 1996.
(ISO9126-01) ISO/IEC 9126-1:2001, *Software Engineering Product Quality-Part 1: Quality Model*: ISO and IEC, 2001.
(ISO15026-98) ISO/IEC 15026-1998, *Information technology-- System and software integrity levels*: ISO and IEC,1998.

//

第4章 软件构造

术语和缩写

OMG：Object Management Group，对象管理集团
UML：Unified Modeling Language，统一建模语言

引言

术语“软件构造”指的是通过编码、验证、单元测试、集成测试和调试的组合，详细地创建可工作的、有意义的软件。

软件构造知识域与其它所有知识域都有联系，特别是软件设计和软件测试，这是因为，软件构造过程本身涉及大量的软件设计和测试活动，软件构造也使用了设计的输出，并为测试提供一个输入，这时，设计和测试都是活动，而不是知识域。设计、构造和测试之间的详细边界（如果存在的话）取决于项目使用的生命周期过程。

尽管某些详细设计可以在构造之前进行，但多数设计是在构造活动内部完成的，这样，软件构造知识域与软件设计知识域紧密相关。在整个构造阶段，软件工程师要进行单元测试，并将其工作集成，这样，软件构造知识域也与软件测试紧密相关。

软件构造一般产生需要在软件项目中管理的最多数量的配置项（源文件、内容、测试用例等等），这样，软件构造知识域也与软件配置管理知识域相关。由于软件构造强烈依赖工具和方法，并且可能是工具最密集的知识域，它与软件工程工具和方法知识域有联系。尽管软件质量对所有知识域都重要，但是，代码是软件项目的最终可交付产品，因此，软件质量与软件构造密切相关。

在软件工程相关学科中，软件构造知识域与计算机科学最接近，因为软件构造使用了计算机科学的算法知识和详细编码实践经验，通常认为二者属于计算机科学领域。软件构造也与项目管理，软件构造的管理对项目管理提出了相当大的挑战。

软件构造主题的分解结构

下面给出软件构造知识域的主题分解结构，以及与知识域相关的主题的简要描述。对每个主题，我们给出了适当的参考文献。图1是这个知识域的分解结构的顶层图形描述。

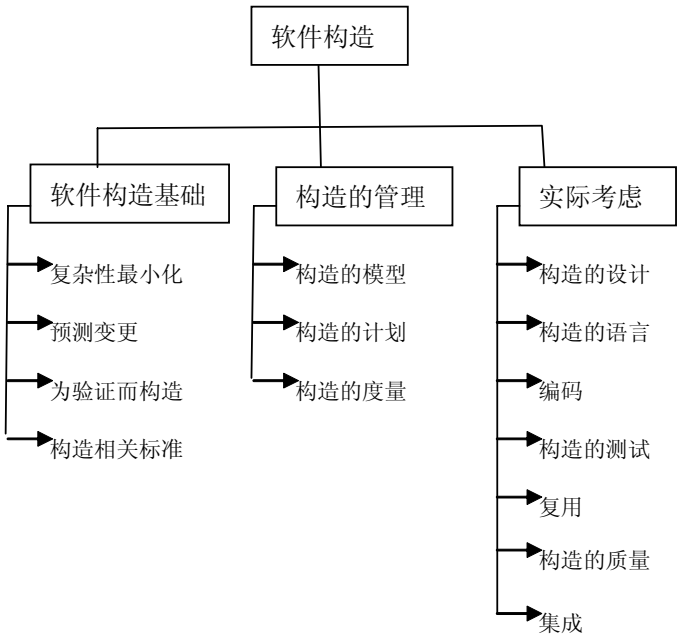

图 1　软件构造知识域主题的分解结构

1 软件构造基础

软件构造基础包括：复杂性最小化、预测变更、为验证而构造、构造中的标准。前3个概念也适用与设计。下面的小节定义这些概念，并描述如何将其应用于构造。

1.1 复杂性最小化[Bec99; Ben00; Hun00;Ker99; Mag93; McC04]

在人们如何将意图传递给计算机中的一个主要因素是，人们在工作记忆中保持复杂结构和信息的能力受到严重限制，特别是长时间保持记忆。这导致了软件构造的一个最强烈的驱动因素：将复杂性最小化。降低复杂性的要求几乎存在于软件构造的每个方面，特别是软件构造的验证和测试过程。在软件构造中，通过创建简单、可读的代码，而不是“聪明”的代码，可以达到降低复杂性的目标。

复杂性最小化可以通过采用标准来实现，这将在主题1.4构造中的标准中讨论，可以采用主题3.3编码中的大量特定技术。也可以用以构造为中心的质量技术来支持复杂性最小化，这将在主题3.5构造的质量中，进行总结。

1.2 预测变更[Ben00; Ker99; McC04]

多数软件要随时间变化，对变更的预测推动着软件构造的许多方面的发展。软件不可避免的是不断变化的外部环境的一部分，外部环境的变化以多种方式影响着软件。可以使用主题3.3编码中的许多特定技术来预测变更。

1.3 为验证而构造[Ben00; Hun00; Ker99; Mag93; McC04]

为验证而构造指的是，用这种方式来构造软件：编写软件的软件工程师、独立的测试活动和运行活动能容易地找出其中的故障。支持为验证而构造的特定技术包括以下：支持代码评审的编码标准、单元测试、代码结构能支持自动测试、限制使用复杂和难以理解的语言结构等。

1.4 构造的相关标准[IEEE12207-95; McC04]

直接影响构造问题的标准包括：编程语言（例如，Java和C++的语言的语言标准）、通信方法（例如，文档格式和内容标准）、平台（例如，操作系统调用的程序员接口标准）、工具（例如，UML等符号的图示标准）。

应用外部标准：软件构造依赖于应用构造语言的外部标准、构造工具、技术接口和软件构造与其它知识域之间的交互。标准有不同的来源，包括硬件和软件接口规范，诸如对象管理集团OMG和国际标准组织IEEE或ISO。

应用内部标准：可以在一个公司级别或特定项目的组织的基础上，创建标准，这些标准支持小组活动的协调、复杂性最小化、预测变更和为验证而构造。

2 构造的管理

2.1 构造的模型[Bec99; McC04]

人们为开发软件创建了很多模型，其中一些模型强调软件构造。从软件构造的观点，一模型是线性的，如瀑布模型和分阶段交付的生命周期模型。这些模型将构造作为仅仅发生在重要的前提工作（详细需求工作、广泛的设计工作和详细的计划）已经完成后的活动来处理。更线性化的方法倾向于强调构造前面的活动（需求和设计），并倾向与在活动之间创建更多的分隔。在这些模型中，构造的重点是编码。

其他模型是迭代的，如进化式原型、极限编程和结对编程（scrum），这些方法倾向于将构造作为一个活动处理，构造与其它软件开发活动（需求、设计和计划）并行地或重叠地进行。这些方法倾向于将设计、编码和测试活动混合起来，并将这些活动的组合称为构造。这样，构造的含义多少取决于采用的生命周期模型。

2.2 构造的计划[Bec99; McC04]

构造方法的选择是构造计划活动的一个关键方面。构造方法的选择影响着构造前提工作的完成程度、这些前提工作的完成顺序、以及它们在构造工作开始前的完备程度。构造的方法影响着项目降低复杂性、预测变更和为验证而构造的能力。这些目标可以在过程、需求和设计层次上处理，但它们也受选择的构造方法的影响。

构造的计划也要根据选择的构造方法，定义创建和集成组件的顺序、各种软件质量管理过程、将任务分配到特定的软件工程师，以及分配其它任务。

2.3 构造的度量[McC04]

可以度量很多构造活动和产品：开发的代码、修改的代码、复用的代码、销毁的代码、代码复杂性、代码检查的统计数据、故障修正和发现的比率、工作量和进度情况。这些度量对于管理构造、构造中的质量保证、改进构造过程以及其它方面，很有用处。有关度量的更多内容，可以参见软件工程过程知识域。

3 实际考虑

构造是一种活动，其中的软件要遵守任意的和凌乱的真实世界约束，并能精确做到这点。由于与真实世界约束的逼近性，构造比其它知识域更受实际考虑的推动，在构造领域，软件工程也许是最接近工艺的。

3.1 构造的设计[Bec99; Ben00; Hun00; IEEE12207-95; Mag93;McC04]

一些项目为构造分配了更多的设计活动，另一些项目有一个明显的设计阶段。无论实际分配如何，一些详细设计工作将发生在构造的层次上，设计工作倾向于受软件要解决的真实世界问题施加的不可改变的约束。

正如建造物理结构的建筑工人必须进行小规模的修改，以适应建设者计划中的没有预料到的差距，软件构造人员也必须作出或大或小的改动，以在构造中补充软件设计的细节。

设计活动在构造层次上的细节与软件设计知识域中描述的相同，只是应用于更小的规模上。

3.2 构造的语言[Hun00; McC04]

构造的语言包括人类向计算机规格说明一个可执行的问题解决方案的所有形式的通信交流。

构造语言的最简单类型是配置语言，这里，软件工程师从事先定义好的有限选择集合中选择需要的东西，创建新的或定制软件装置，在Windows和Unix操作系统中使用的基于文本的配置文件就是一个例子，某些程序生成器的菜单风格的选择列表是另一个例子。

工具箱语言用于从工具（特定应用可复用组件的集成集合）中构造新的应用，比配置语言更复杂。工具箱语言可以显式地定义为应用编程语言（例如，脚本），也可以由工具箱的接口集合隐含定义。

编程语言是最灵活类型的构造语言。它们也包含了特定应用领域和开发过程的最小公共信息，因此，为有效使用它，需要最多的培训。编程语言使用的3类符号包括：语言学的、形式化的、可视化的。

语言学符号特别显著之处是使用类似词的文本串来表示复杂的软件构造，并将类似词的串组合为具有类似句子语法的模式。使用得当时，每个这样的串将具有较强的语义内涵，为其表示的软件构造被执行时要发生的事情的直接、直观的理解。

形式化符号较少依赖于单词和文本串的直观的、日常意义，而是更多地依赖于由简明的、无二义的、形式化（或数学）的定义。形式化构造符号是多数系统编程形式的核心部分，在这里，精确性、时间行为、可测试性比易于到自然语言更重要。形式化构造也使用组合符号

的简明定义方式，以避免许多自然语言构造的二义性。

可视化符号比语言学的和形式化的构造更较少依赖于面向文本的符号，而是依赖于表示软件的可视化实体的直接的视觉解释和放置。可视化构造多少受到仅仅在显示器上移动可视化实体来构造复杂语句的困难的限制，但是，如果主要的编程任务是建造或调整一个详细行为已经定义了的程序的可视化接口时，它就是一个强有力的工具。

3.3 编码[Ben00; IEEE12207-95; McC04]

下面的内容可以应用于软件构造的编码活动：（1）构造可理解的源代码的技术，包括命名和源代码编排；（2）使用类、枚举类型、变量、命名约束和其它类似实体；（3）使用控制结构；（4）处理错误条件：预计的错误和异常（例如，错误的输入数据）；（5）预防代码级的安全泄露（例如，缓冲区超限或数组下标溢出）；（6）通过排斥机制和顺序访问可复用资源规则来使用资源（包括线程和数据库锁）；（7）源代码组织（语句、例程、类、包或其它结构）；（8）代码文档；（9）代码调整。

3.4 构造的测试[Bec99; Hun00; Mag93; McC04]

构造设计通常由编写代码的软件工程师完成的两类测试：单元测试和集成测试。构造测试的目的是减小故障被带入到代码时的时间与检测到故障的时间之间的差距。有时构造测试在代码完成后进行，有时，测试可以在代码编写前进行。

构造测试一般涉及测试类型的一个子集，测试类型在软件测试知识域中描述。例如，构造测试一般不涉及系统测试，α测试、β测试、强度测试、配置测试、可用性测试，以及其它专门的测试。

本主题有两个已经发布的标准：IEEE829-1998标准“IEEE软件测试文档标准”和IEEE1008-1987标准“IEEE软件单元测试标准”。对于更专门的参考文件，请参见软件测试知识域的相应子主题2.1.1单元测试和子主题2.1.2集成测试。

3.5 复用[IEEE1517-99; Som05].

在IEEE1517-99的引言中指出：实现软件复用比创建是使用程序库需要做更多的工作，它需要通过将复用过程和活动集成到软件生命周期中，来将复用的实践经验形式化。但是，复用在软件构造中很重要，因此这里将其作为一个主题处理。

在软件构造的编码和测试中，与复用相关的主题有：（1）选择可复用的单元、数据库、测试过程或测试数据；（2）代码或测试的可复用性评价；（3）在新代码、测试过程或测试数据上的复用信息的报表制作。

3.6 构造的质量[Bec99; Hun00; IEEE12207-95; Mag93;McC04]

有很多保证在构造代码时的质量保证技术，构造使用的主要技术包括：（1）单元测试和集成测试（主题3.4构造的测试）；（2）测试优先的开发（参见软件测试知识域，主题2.2测试目标）；（3）代码逐步求精（stepping）；（4）使用断言；（5）调试；（6）技术评审（参见软件质量知识域，子主题3.3技术评审）；（7）静态分析（IEEE1028）（参见软件质量知识域，主题3.3评审与审计）。

特定技术或选择的技术依赖于被构造的软件的本质，以及完成构造的软件工程师的技能水平。构造质量活动的重点与其它质量活动的重点不同。构造质量活动的重点在代码和与代码紧密相关的产品，与代码相关的产品是小规模的设计，它们与需求、高层设计和计划等其它与代码不密切相关的产品是相对的。

3.7 集成[Bec99; IEEE12207-95; McC04]

构造中的一个关键活动是集成单独构造的例程、类、组件和字系统。另外，一个特定的软件系统可能需要与其它的软件或硬件系统集成。与构造集成相关的东西包括：计划集成组件的顺序、创建支持附加组件来支持软件的过渡版本、确定集成组件前要完成的测试和质量工作的程度、确定项目中测试软件过渡版本的时间点。

主题与参考文献矩阵

	[Bec99]	[Ben00]	[Hun00]	[IEEE1517]	[IEEE12207.0]	[Ker99]	[Mag93]	[Mcc04]	[Som05]
1软件构造基础									
1.1复杂性最小化	C17	C2,c3	C7,c8			C2,c3	C6	C2,c3,c7-c9,c24,c27,c28,c31,c32,c34	
1.2预测变更		C11,c13,c14				C2,c9		C3-c5,c24,c31,c32,c34	
1.3为验证而构造		C4	C21,c23,c34,c43			C1,c5,c6	C2,c3,c5,c7	C8,c20-c23,c31,c34	
1.4构造有关标准					X			C4	
2构造的管理									
2.1构造的模型	C10							C2,c3,c27,c29	
2.2构造的计划	C12,c15,c21							C3,c4,c21,c27-c29	
2.3构造的度量								C25,c28	
3实际考虑									
3.1构造的设计	C17	C8-c10,p	C33		X		C6	C3,c5,c24	

		175-6							
3.2构造的语言			C12, c14-c20					C4	
3.3编码		C6-c10			X			C5-c19,c25-c26	
3.4构造的测试	C18		C34, c43		X		C4	C22,c23	
3.5复用				X					C14
3.6构造的质量	C18		C18		X		C4,c6,c7	C8,c20-c25	
3.7集成	C16				X			C29	

软件构造的推荐参考文献

[Bec99] K. Beck, "Extreme Programming Explained: Embrace Change," Addison-Wesley, 1999, Chap. 10, 12, 15, 16-18, 21.

[Ben00a] J. Bentley, "Programming Pearls," Second ed: Addison-Wesley, 2000, Chap. 2-4, 6-11, 13, 14, pp. 175-176.

[Hun00] A. Hunt and D. Thomas, "The Pragmatic Programmer," Addison-Wesley, 2000, Chap. 7, 8 12, 14-21, 23, 33, 34, 36-40, 42, 43.

[IEEE1517-99] IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*: IEEE, 1999.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, vol. IEEE, 1996.

[Ker99a] B. W. Kernighan and R. Pike, "The Practice of Programming," Addison-Wesley, 1999, Chap. 2, 3, 5, 6, 9.

[Mag93] S. Maguire, "Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Software," Microsoft Press, 1993, Chap. 2-7.

[McC04] S. McConnell, "Code Complete: A Practical Handbook of Software Construction," Microsoft Press, 1993, Chap. 2-32.

[Som05] I. Sommerville, "Software Engineering," Seventh ed: Addison-Wesley, 2005.

附录A 深入阅读的参考文献

(Bar98) T. T. Barker, *Writing Software Documentation: A Task-Oriented Approach*: Allyn & Bacon, 1998.

(Bec02) K. Beck, "Test-Driven Development: By Example," Addison-Wesley, 2002.

(Fow99) M. Fowler and al, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

(How02) M. Howard and D. C. Leblanc, *Writing Secure Code*: Microsoft Press, 2002.

(Hum97b) W. S. Humphrey, *Introduction to the Personal Software Process*:

Addison-Wesley, 1997.
(Mey97) B. Meyer, "Object-Oriented Software Construction," Second ed: Prentice-Hall, 1997, Chap. 6, 10, 11.
(Set96) R. Sethi, "Programming Languages: Concepts & Constructs," Second ed: Addison-Wesley, 1996, Parts II -V.

附录B 有关标准

(IEEE829-98) IEEE Std 829-1998, *IEEE Standard for Software Test Documentation*: IEEE, 1998.
(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*: IEEE, 1987.
(IEEE1028-97) IEEE Std 1028-1997 (R2002), *IEEE Standard for Software Reviews*: IEEE, 1997.
(IEEE1517-99) IEEE Std 1517-1999, *IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes*: IEEE, 1999.
(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, vol. IEEE, 1996.

//

第5章 软件测试

术语与缩写

SRET：Software Reliability Engineered Testing，软件可靠性加固测试

引言

测试是为评价、改进产品质量、标识产品的缺陷和问题而进行的活动。

软件测试由一个程序的行为在*有限*测试用例集合上，针对*期望*的行为的*动态*验证组成，测试用例是从通常的无限执行域中适当*选取*的。

在以上定义中，斜体字对应与标识软件测试知识时的关键问题，特别是：

（1）动态：这意味着，测试总是隐含在经评价过的输入上执行程序，更确切地，输入值本身并不能充分地确定一个测试，因为一个复杂的、非确定性的系统可能对相同的输入作出不同的反应行为，这取决于系统的状态。尽管在这个知识域中，“输入”这个术语在需要它的情况下，仍将继续使用，并有隐含的约定：其含义也包括一个特定的输入状态。静态技术与动态不同，并作为动态技术的补充，将在软件质量知识域中描述。

（2）有限：即使是简单的程序，理论上也有很多的测试用例，需要若干年或若干月才能完成穷举测试，因此，在实践中，一般认为整个测试集合是无限的。测试总是隐含了有限的资源和进度与无限的测试需求之间的权衡。

（3）选取：人们提出的许多测试技术，本质上的区别在于如何选择测试集合，软件工程师必须意识到，不同的选择准则可能产生差别很大的效果。在给定条件下，如何标识最适当的选择准则，是一个非常复杂的问题。在实践中，需要使用风险分析技术和测试工程技能。

（4）期望：确定观察的程序执行的输出是否可以接受，这尽管不容易但也必须是可能的，否则测试工作就无用。观察到的结果可以与用户的期望对比（一般称为确认测试）、与规格说明对比（验证测试）、或与隐含的需求的期望行为或合理的期望对比，参见软件需求

知识域的主题6.4接收测试。

软件测试的视图已经演化为更具建设性，测试不再被认为是一种仅在编码阶段完成后才开始的活动，其目标只是检测失效。现在，软件测试被认为是一种应该包括整个开发和维护过程的活动，它本身是实际产品构造的一个重要部分。确实，测试的规划应该在需求过程的早期就开始，随着开发的进展，必须系统地和连续地开发或精化测试计划和过程。测试的规划和设计活动本身为设计者查找潜在的弱点（如设计中的忽略或矛盾之处、文档中的省略或二义之处）提供了有用的输入。

正是目前，人们认为，对质量的正确态度是预防措施之一：避免问题比更正问题好得多。这样，必须认为，测试不仅是检查预防措施是否有效的主要手段，而且是识别由于某种原因预防措施无效时的错误的主要手段。一个可能明显但有值得认识到的事是，在广泛的测试活动成功完成后，软件可能仍包含错误，交付后出现的软件失效的补救措施是由更正性维护，软件维护主题在软件维护知识域中讨论。

在软件质量知识域（参见主题3.3质量管理技术），软件质量管理技术明显分为：静态技术（没有代码执行）和动态技术（代码执行）。这两类都有用。软件测试知识域集中于动态技术。

软件测试也与软件构造有关（参见主题3.4构造测试），即使不是其中一部分，单元和集成测试与软件构造密切相关。

主题的分解结构

软件测试知识域的分解结构如图1所示。

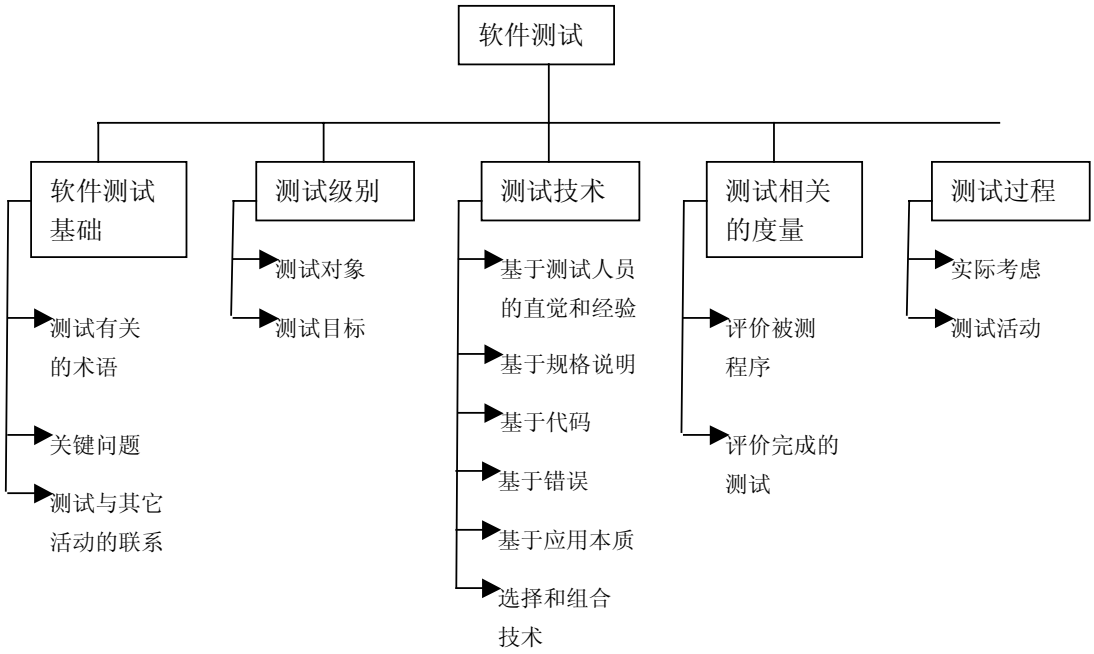

图 1　软件测试知识域主题的分解结构

第一个子域描述软件测试基础，它覆盖软件测试领域的基本定义、基本术语和关键问题、与其它活动的联系。

第二个子域是测试级别，有两个（正交的）主题组成：2.1是大型软件测试的传统划分级别，2.2考虑特定条件或特性的测试，称为测试目标。并非所有测试类型都适用于每个软

件产品，也没有列出每个可能的类型。

测试目的和测试目标一起确定如何标识测试集合，二者都与测试集合的一致性（达到规定的目标需要多少测试？）和组成（为达到规定的目标，需要选择哪些测试用例？）有关（尽管“达到规定的目标”通常是隐含的，且一般只提出前面两个问题中的第一个）。有关第一个问题的准则称为测试适合性准则，有关第二个问题的准则称为测试选择准则。

过去几十年来，人们开发了多个测试技术，并仍在提出新的测试技术。子域3覆盖了普遍接受的技术。

子域4讨论与测试有关的度量。

子域5讨论与测试过程有关的问题。

1 软件测试基础

1.1 与测试有关的术语

1.1.1 测试及相关术语的定义[Bei90:c1; Jor02:c2; Lyu96:c2s2.2] (IEEE610.12-90)

软件测试知识域的综合性介绍可以参见推荐的参考文献。

1.1.2 故障（Fault）与失效（Failure）[Jor02:c2; Lyu96:c2s2.2; Per95:c1; Pfl01:c8] (IEEE610.12-90; IEEE982.1-88)

软件工程文献中，有许多术语用于描述功能失调，如毛病（fault）、失效(failure)、错误(error)，以及其它。这个术语在IEEE610.12-1990标准“软件工程术语标准词汇(IEEE610-90)”中有精确定义，也在软件质量知识域中讨论。有必要明确地区分功能失调的原因（这里应该使用错误fault或缺陷defect）和观察到的系统提供的服务中的不希望的效果（称为失效failure）。测试可以揭露失效，但错误才能够并必须被消除。

但是，应该认识到，失效的原因并非总能毫不含糊的识别。没有理论准则来确定什么错误引发观察到的失效，必须指出，正是要修正错误才能消除问题，但其它方面的修改也能达到这一点。为避免二义性，一些作者使用引发失效的输入（failure-causing input）(Fra98)而不是错误（fault），即这些输入集合引发一个失效出现。

1.2 一些关键问题

1.2.1 测试选择准则/测试适当性准则（或停止规则）[Pfl01:c8s7.3; Zhu97:s1.1] (Wey83; Wey91; Zhu97)

测试选择准则是一种手段，用于确定适当测试用例应该是什么样。选择准则用于：选择测试用例、检查选择的测试组（suite）是否合适，即确定测试是否能停止。参见主题5.1管理问题中的子主题终止。

1.2.2 测试有效性（effectiveness）/测试目标[Bei90:c1s1.4; Per95:c21] (Fra98)

测试是对程序执行的一个样本的观察，可以用不同的目标来指导样本的选择：只有以追求的目标为样板，才能评价测试的有效性。

1.2.3 为标识缺陷而测试[Bei90:c1; Kan99:c1]

在为标识缺陷而进行的测试中，成功的测试是引起系统失效的测试，这与为展示软件满足起规格说明或其它需要的特性而进行的测试完全不同，在这类情况中，成功的测试是没有观察到（重大）失效的测试。

1.2.4 神喻（oracle）问题[Bei90:c1] (Ber96, Wey83)

神喻是任何决定一个程序是否在给定的测试中具有正确行为、并相应给出一个“通过”

或“失败”的裁定的（人或机械的）智能体。有多种不同类型的神喻，神喻的自动化非常困难和昂贵。

1.2.5 测试的理论和实践局限[Kan99:c2] (How76)

测试理论对将一个没有确定的置信度赋给一系列通过的测试提出了告诫。不幸的是，测试理论建立的多数结论是否定性的，因为它们指出什么是测试永远不能达到的，而不是其实际能达到的。关于这点的最著名问题是Dijkstra的格言：程序测试可以用于说明存在错误，但却不能用来说明没有错误。明显的原因是真实软件的完全测试是不可行的。因此，必须基于风险来驱动测试，并将测试作为风险管理策略看待。

1.2.6 不可行路径问题[Bei90:c3]

不可行路径是任何输入都不能到达的控制流路径，这是面向路径的测试中的一个重要问题，特别是在为基于代码的测试技术自动导出测试输入中。

1.2.7 可测试性[Bei90:c3,c13] (Bac90; Ber96a; Voa95)

术语“软件可测试性”有两个相关的但又不同的含义：一方面，它指一个软件完成给定测试覆盖准则的容易程度，见(Bac90)；另一方面，它被定义为如果一个软件是有缺陷的，这个软件在测试中显露失效的概率，可能在统计上度量，见(Voa95, Ber96a)。这两个含义都很重要。

1.3 测试与其它活动的联系

软件测试与静态软件质量管理技术、正确性证明、排错和编程有关，但又不同于它们。但是从软件质量分析员和认证员的角度，来考虑测试，会得到更多的信息。

测试与静态软件质量管理技术的关系可以参见软件质量管理知识域的子域2软件质量管理过程 [Bei90:c1; Per95:c17] (IEEE1008-87)。

测试与正确性证明和形式化验证的关系见[Bei90:c1s5; Pfl01:c8]。

测试与排错的关系见软件构造知识域主题3.4构造的测试[Bei90:c1s2.1] (IEEE1008-87)。

测试与编程的关系见软件构造知识域主题3.4构造的测试[Bei90:c1s2.3]。

测试与认证的关系见(Wak99)。

2 测试级别

2.1 测试对象(target)

软件测试随开发和维护过程，通常在不同的级别上进行，即，测试的对象是变化的：单个模块、一组（通过目的、使用、行为或结构而联系起来的）模块、或整个系统[Bei90:c1; Jor02:c13;Pfl01:c8]。可以在概念上区分3个大的测试阶段：单元、集成、系统。这里没有隐含任何过程模型，也没有隐含其中一个阶段比其它两个阶段更重要。

2.1.1 单元测试[Bei90:c1; Per95:c17; Pfl01:c8s7.3] (IEEE1008-87)

单元测试验证孤立的、可以分别测试的软件片的功能，在不同的上下文中，软件片可以是单个的子程序、或者是由紧密联系的单元组成的较大的组件。在IEEE软件单元测试标准(IEEE1008-87)中对单元测试进行了更精确的定义，这个标准也描述了系统化和文档化的单元测试的集成方法。一般，单元测试与访问被测试的代码一起进行，并需要排错工具的支持，可能涉及编写代码的程序员。

2.1.2 集成测试[Jor02:c13,14; Pfl01:c8s7.4]

集成测试是验证软件组件之间的交互的过程，经典的集成测试策略，如自顶向下、自底向上，与传统的层次结构软件一起使用。现代的系统化集成策略是由体系结构驱动的，它隐含了基于已标识的功能线索来集成软件组件或子系统。集成测试是一个连续的活动，在其每个阶段，软件工程师必须抽象较低层次的视图，并集中到正在集成的层次的视图上。除了小型、简单的软件，通常使用系统化的增量集成策略，而不是一次性将所有组件聚集在一起，后者被成为“大爆炸式（big bang）”测试。

2.1.3 系统测试[Jor02:c15; Pfl01:c9]

系统测试关注整个系统的行为。在单元测试和集成测试中，已经标识了多数的功能性失效，通常认为，系统测试适合于评价系统的非功能性需求，如安全性、速度、精确性和可靠性。这个阶段也要评价系统与其它应用、实用程序、硬件设备、运行环境等的接口。更多的功能与非功能性需求的信息可以参见软件需求知识域。

2.2 测试的目标[Per95:c8; Pfl01:c9s8.3]

测试是按多少有些明确陈述、精确程度变化的特定目标的观点进行的。用定量的术语精确地陈述目标，可以在测试过程上建立控制。测试可以针对不同的特性，可以设计测试用例来检查是否正确实现了功能规格说明，这在文献中被称为遵从性测试、正确性测试、功能性测试。但是，还应该测试其它几个非功能性特性，这包括：性能、可靠性、可使用性等。

其它重要的测试目标还包括（但不局限于）：可靠性度量、可用性评价和接收，接收可以采用不同的途径。注意，测试目标随测试对象而变化，一般，在测试的不同级别针对不同的目的。

前面为本主题推荐的参考文献描述了潜在的测试目标集合，下面列出的是在文献中最常引用的子主题。注意，某些类型的测试更适合于定制的软件包，安装测试就是一个例子，其它的测试则适合于一般的产品，β测试就是一个例子。

2.2.1 接收/认定测试[Per95:c10; Pfl01:c9s8.5] (IEEE12207.0-96:s5.3.9)

接收测试按照客户的需求来检查系统的行为，需求可以任意用任意方式表达。客户承担或说明典型的任务，以检查需求是否满足，或检查开发机构已经标识了软件的目标市场的需求。这个测试活动可能涉及系统的开发人员，也可能不涉及他们。

2.2.2 安装测试[Per95:c9; Pfl01:c9s8.6]

通常，在软件开发和接收测试完成后，可以在目标环境中通过安装来验证软件。可以将安装测试看作是根据硬件配置而再次进行的系统测试，并可以验证安装过程。

2.2.3 α与β测试[Kan99:c13]

在软件发布前，有时会让小规模、有代表性的潜在用户试用，可以在开发机构中进行（α测试），也可在用户处进行（β测试）。通常，不对α与β使用进行控制，测试也不总是针对一个测试计划。

2.2.4 遵从性测试/功能测试/正确性测试[Kan99:c7; Per95:c8] (Wak99)

遵从性测试目的是确认观察到的被测试软件的行为是否遵从其规格说明。

2.2.5 可靠性达到和评价[Lyu96:c7; Pfl01:c9s.8.4] (Pos96)

为帮助标识错误，测试是改进可靠性的一种手段。与之相反，根据运行大纲，随机地产生测试用例，可以导出可靠性的统计学度量。使用可靠性增长模型，这两个目标都能达到（参见子主题4.1.4寿命测试、可靠性评价）。

2.2.6 回归测试[Kan99:c7; Per95:c11,c12; Pfl01:c9s8.1] (Rot96)

根据IEEE610.12-90，回归测试是“系统或组件的选择性测试，以验证修改没有引起不需要的后果……”。在实际中，这个思想用于表明以前通过测试的软件，目前仍能通过。Beizer(Bei90)将其定义为重复的测试，目的在于表明，除了需要的外，软件的行为没有改变。

显然，必须在每当一次变更后进行的回归测试作出的断言和回归测试需要资源之间作出平衡。

在主题2.1测试对象中的每个测试级别上都可以进行回归测试，并可应用与功能和非功能测试。

2.2.7 性能测试[Per95:c17; Pfl01:c9s8.3] (Wak99)

性能测试特别针对验证软件满足规格说明的性能需求，例如，容量和响应时间。一种特殊的性能测试是容限测试(Per95:p185,p487; Pfl01:p401)，它要测试程序或系统的内部极限。

2.2.8 压力测试[Per95:c17; Pfl01:c9s8.3]

压力测试以设计的最大负载运行软件，并以超过最大负载运行软件。

2.2.9 背对背测试

在一个软件产品的两个实现版本上运行同一个测试集合，并比较结果。

2.2.10 恢复测试[Per95:c17; Pfl01:c9s8.3]

恢复测试的目的是验证软件在“灾难”后的重启动能力。

2.2.11 配置测试[Kan99:c8; Pfl01:c9s8.3]

开发的软件将服务于不同的用户时，配置测试用于分析软件在规格说明的不同的配置下的行为。

2.2.12 可用性测试[Per95:c8; Pfl01:c9s8.3]

这个过程评价终端用户学习和使用软件（包括用户文档）的难易程度、软件功能在支持用户任务的有效程度、从用户的错误中恢复的能力。

2.2.13 测试驱动的开发[Bec02]

测试驱动的开发本质上不是一个测试技术，它将测试作为一个需求规格说明文档的替代手段，而不是将其作为一个独立的检查软件是否正确实现需求的手段。

3 测试技术

测试的一个目标就是尽可能多地揭露潜在的失效，人们为此开发了很多的技术，这些技术试图通过运行一个或多个从被认为是等价的执行的已标识的类中导出的测试，来“中断”程序这些技术的主要原理是尽可能系统化的标识程序行为的代表性集合，例如，考虑输入域、

场景、状态和数据流等的子类。

找出分类所有技术的相同的基础比较困难，这里使用的分类基础是一个折中。分类基于测试用例是如何从软件工程师的直觉和经验、规格说明、代码结构、需要发现的（真实或人造）错误、使用领域或应用的本质中生成的。有时，如果测试依赖于有关软件如何设计和编码的信息，则这些技术被分为白盒或玻璃盒，如果测试仅依赖于输入/输出行为，则这些技术被称为黑盒，还有一类组合使用了两种或两种以上的技术。

显然，这些技术通常不会被测试人员平等地使用，下面是软件工程师应该知道的一些技术。

3.1 基于软件工程师直觉和经验

3.1.1 特定测试[Kan99:c1]

实践中最广泛采用的技术也许是特定测试：测试是根据软件工程师的技能、直觉和对类似程序的经验导出的。特定测试对于标识不容易被形式化技术捕获的特殊的测试有用。

3.1.2 探索性测试

探索性测试定义为边学习、边设计测试和边执行测试，即，没有在制定的测试计划中事先定义测试，而是动态的设计、执行和修改测试。探索性测试的效果依赖于软件工程师的知识，这些知识可以从不同的来源导出：测试中观察到的产品行为、对应用、平台和失效过程的了解、可能的错误和失效类型、与特定产品相关的风险等[Kan01:c3]。

3.2 基于规格说明的技术

3.2.1 等价类划分[Jor02:c7; Kan99:c7]

将输入域划分为一些子集合或等价类，根据一个指定的联系，这些类是等价的，从每个类中提取的测试的代表性测试集合（通常只有一个测试）。

3.2.2 边值分析[Jor02:c6; Kan99:c7]

在变量的输入域的边界上和边界附近选择测试用例，其基本原理是许多错误倾向于发生在输入的极端值附近。这个技术的一个扩展是强健性测试，在变量的输入域外部选择测试用例，以测试程序对于不期望或错误输入的强健性。

3.2.3 决策表[Bei90:c10s3] (Jor02)

决策表表示了条件（粗略地说是输入）和行动（粗略地说是输出）之间的关系。考虑条件和行动的每个可能的组合，系统性地选择测试用例。一个相关的技术成为原因-结果图（*cause-effect graphing*）[Pfl01:c9]。

3.2.4 基于有限自动机[Bei90:c11; Jor02:c8]

将一个程序看作一个有限状态自动机，可能根据对其状态和状态迁移的覆盖，来选择测试用例。

3.2.5 根据形式化规格说明的测试[Zhu97:s2.2] (Ber91; Dic93; Hor95)

用形式化语言形式给出规格说明，可以自动导出功能性测试用里，同时也为检查测试结果提供了一个参考输出或神喻。有许多方法可以从模型(Dic93, Hor95)和代数规格说明(Ber91)中导出测试用例。

3.2.6 随机测试[Bei90:c13; Kan99:c7]

测试完全是随机产生的，不要与子主题3.5.1操作大纲中描述的来自操作大纲的统计测试混淆。之所以将这类测试划分到基于规格说明的测试，是因为必须知道输入域，以便在其中选择随机点。

3.3 基于代码的技术

3.3.1 基于控制流准则[Bei90:c3; Jor02:c10] (Zhu97)

基于控制流覆盖准则旨在覆盖程序中所有的语句或语句块，或它们的任意组合。已经提出了几个覆盖准则，如决策条件覆盖。最强的基于控制流的准则是路径测试，其目的是执行流程图中所有从入口到出口的路径。由于循环的存在，路径测试一般是不可行的，实践中采用其它较弱的准则，如语句测试、分支测试、条件/决策测试。这些测试的适当度按百分比度量，例如，当测试将所有分支至少执行一遍时，可以认为达到了100%的分支覆盖。

3.3.2 机遇数据流的准则[Bei90:c5] (Jor02; Zhu97)

在基于数据流的测试中，用有关程序的变量如何被定义、使用和丢弃（取消定义）的信息来注释控制流图。最强的准则是使用所有定义的路径，它要求对于每个变量，每一个从变量的定义到使用定义的控制流路径段被执行。为减少需要的路径数目，可以使用较弱的策略，如所有定义或所有使用。

3.3.3 基于代码测试的参考模型（流程图、调用图）[Bei90:c3; Jor02:c5].

尽管这不是一个技术，在基于代码的测试技术中，程序的控制结构可以用一个流程图表示。流程图是一个有向图，节点和弧分别对应程序元素，例如，节点可以代表语句或不被中断的语句序列，弧代表节点之间的控制转移。

3.4 基于错误的技术(Mor90)

随形式化的程度不同，基于错误的测试技术特别针对揭露可能的或事先定义的错误类别，来设计测试用例。

3.4.1 错误猜测[Kan99:c7]

在错误猜测中，软件工程师设计特殊的测试用例，试图给定程序中最可能的错误。一个比较好的信息源是以前项目中发现的错误的历史，以及软件工程师的技能。

3.4.2 变异测试[Per95:c17; Zhu97:s3.2-s3.3]

变异是被测试程序的一个微小修改的版本，对程序进行一个小的、语法上的变更。在原始版本和所有变异版本上运行每个测试用例，如果一个测试用例能够成功分辨原始程序和变异版本，就认为后者被丢弃了。最初，变异测试用于评价测试集合（参见4.2），它本身也是一个测试准则：随机地选择测试用例，直到足够多的变异被丢弃，或者特别地设计测试，来丢弃幸存的变异版本。在后一种情况中，可以将变异测试分类为基于代码的测试。变异测试的基本假设是耦合效应，即，通过寻找简单的语法错误，可以发现更复杂的、真实的错误。为了让变异测试有效，必须系统化地自动生成大量的变异。

3.5 基于使用的技术

3.5.1 操作大纲[Jor02:c15; Lyu96:c5; Pfl01:c9]

在为进行可靠性评价而进行的测试中，测试环境必须尽可能相近地重现软件的运行/操作环境。其思想是从观察到的测试结果中，推测软件在未来实际使用时的可靠性。为此，根据其在实际操作中出现的情况，为输入分配一个概率分布或大纲。

3.5.2 软件可靠性加固测试[Lyu96:c6]

软件可靠性加固测试（Software Reliability Engineered Testing，SRET）是一个包括整个开发过程的测试方法，这里，测试是“根据可靠性目标、期望的相对使用、领域中不同功能的关键性来设计和进行的”。

3.6 基于应用本质的测试

上面的技术适用与所有类型的软件。但是，对某些应用，为导出测试，需要一些额外的知识，基于被测试应用的本质，下面提供一些专门的测试领域：（1）面向对象的测试[Jor02:c17; Pfl01:c8s7.5](Bin00)；（2）基于组件的测试；（3）基于Web的测试；（4）图形用户接口GUI测试[Jor20]；（5）并发程序的测试(Car91)；（6）协议遵从性测试(Pos96; Boc94)；（7）实时系统测试(Sch94)；（8）极端要求安全性的系统的测试(IEEE1228-94)。

3.7 技术的选择和组合

3.7.1 功能性与结构性[Bei90:c1s.2.2; Jor02:c2,c9,c12; Per95:c17](Pos96)

通常分别将基于规格说明的技术和基于代码的技术成为功能性的和结构性的。这两类选择测试的方法并不是相互可替代的，而是互补的，事实上，它们使用不同的信息来源，突出了不同种类的问题。根据预算的考虑，可以组合使用它们。

3.7.2 确定性的与随机的(Ham92; Lyu96:p541-547)

根据前面列出的各种技术，可以按确定性方式选择测试用例，也可以从输入分布中随机选择，就象可靠性测试那样。已经进行了一些分析性和经验性的比较，分析了一种方法比另一种方法更有效的条件。

4 测试相关的度量

有时，测试技术与测试目标混淆在一起，测试技术是手段，用来帮助确保测试目标的达到。例如，分支覆盖是常见的测试技术，要达到特定的分支覆盖度量在本质上不是测试的目标，它仅是一个通过系统地从一个决策点执行每个程序分支，以增加发现失效的机会的手段。为避免这种误解，应该在提供对被测试程序的评价的测试相关的度量和评价测试集合完备性的测试相关的度量之间，进行明确的区分。附加的程序度量信息在软件工程管理知识域的子域6软件工程度量中。有关度量的附加信息请参见软件工程过程知识域的子域4产品与过程度量。

通常认为，度量是质量分析的仪器手段，度量也可用于优化测试的计划和执行。测试管理可以使用几个过程度量来监控进展。与管理目的的测试过程相关的度量见主题5.1实际考虑。

4.1 评价被测试的程序(IEEE982.1-98)

4.1.1 辅助测试的计划和设计的程序度量[Bei90:c7s4.2; Jor02:c9] (Ber96; IEEE982.1-88)

可以使用基于程序规模（例如，源代码行或功能点）或基于程序结构（如复杂性）的度量来指导测试。结构度量也可以包括程序模块之间的度量（如一个模块调用其它模块的频率）。

4.1.2 错误类型、分类和统计[Bei90:c2; Jor02:c2; Pfl01:c8](Bei90; IEEE1044-93; Kan99; Lyu96)

有关错误的分类学和分类的文献有很多，为了让测试更有效，知道可以在被测试的软件中发现哪类错误，以及这些错误在过去发生的相对频率，很重要。这个信息在预测质量和过程改进中，很有用处。更多的信息请参见软件质量知识域的主题3.2缺陷特征。有一个分类软件异常的IEEE标准(IEEE1044-93)。

4.1.3 错误密度[Per95:c20] (IEEE982.1-88; Lyu96:c9)

通过按发现的错误类型来计数和分类，可以评定被测试的程序。对每个错误类型，错误密度定义为发现的错误数目与程序规模的比率。

4.1.4 寿命测试、可靠性评价[Pfl01:c9] (Pos96:p146-154)

可以从可靠性达到和评价（参见子主题2.2.5）得到软件可靠性的统计学估计，它被用于评价一个产品和决定是否可以终止测试。

4.1.5 可靠性增长模型[Lyu96:c7; Pfl01:c9] (Lyu96:c3,c4)

可靠性增长模型提供了基于在可靠性达到和评价（子主题2.2.5）下观察到的失效而对可靠性的预测，一般，这些模型假设引起观察到的失效的错误已经被改正（尽管一些模型也接受不完全的改正），这样，在平均上，产品的可靠性呈现增长的趋势。已经发表了十多个模型，许多模型基于一些公共假设，其它则不然。特别地，这些模型划分为错误计数模型和失效间隔时间模型。

4.2 评价完成的测试

4.2.1 覆盖/彻底性度量[Jor02:c9; Pfl01:c8] (IEEE982.1-88)

有几个测试适当度准则要求，测试用例系统性地程序中或规格说明中（见子主题3.3）标识的元素集合。为评价已经执行的测试的彻底性，测试人员必须控制被覆盖的元素，这样才能动态地度量已覆盖的元素和所有元素之间的比率。例如，有可能度量程序流程图中已覆盖的分支的百分比，或已执行的列在规格说明文档中的功能需求的百分比。基于代码的适当性准则要求适当的“仪器”来测量被测试程序。

4.2.2 错误种植[Pfl01:c8] (Zhu97:s3.1)

在测试前，可以在程序中人工引入一些，执行测试时，可能揭露一些种植的错误，以及一些程序中本来的错误。在理论上，根据发现的哪些人工错误及其数目，可以评价测试的有效性，并估计剩余的程序原来的错误数目。实际中，统计学家要考虑种植的错误的分布，以及其相对与程序原本错误的代表性，和外推基于的小样本规模。一些人指出，使用这个技术必须非常小心，因为将错误插入软件涉及到将其留在程序中的风险。

4.2.3 变异评分[Zhu97:s3.2-s3.3]

在变异测试（子主题3.4.2）中，被丢弃的变异对生成的总变异数目的比率可以用来度量执行的测试的有效性。

4.2.4 不同技术的比较和相对有效性techniques
[Jor02:c9,c12; Per95:c17; Zhu97:s5] (Fra93; Fra98;Pos96: p64-72)

已经进行了一些研究，来比较不同测试技术的相对有效性。必须精确定义技术评定的特性，这一点很重要，例如，什么是术语“有效性effectiveness”的精确含义？可能的解释是：发现第一个失效需要的测试的数目、通过测试发现的错误数目与测试中和测试后发现的错误的总数目的比率、或者可靠性增加了多少。根据前面列出的有效性的各个概念，对不同技术进行了分析的和经验的比较。

5 测试过程

需要将测试概念、策略、技术和度量集成到一个定义好的、受控制的由人来执行的过程中。测试过程支持测试活动，为测试小组提供从测试计划到评价测试输入的指导，这样，就提供了可证明的保证：测试目标可以有效地满足成本。

5.1 实际考虑

5.1.1 态度/无自我的编程[Bei90:c13s3.2; Pfl01:c8]

成功测试的一个重要成分是对测试和质量保证活动的合作态度。在培育对开发和维护中发现失效的良好接受态度方面，管理者起着关键作用。例如，阻止程序员之间对代码所有权的固有思想，程序员们就不会感到对其代码揭露的失效负有责任。

5.1.2 测试指导[Kan01]

测试阶段可以由不同的目的来指导，例如：在基于风险的测试中，使用产品风险来对测试策略进行优先排序和聚焦，或者在基于场景的测试中，用基于规格说明的软件场景来定义测试用例。

5.1.3 测试过程管理[Bec02: III; Per95:c1-c4; Pfl01:c9] (IEEE1074:97; IEEE12207.0-96:s5.3.9,s5.4.2, s6.4,s6.5)

必须将在不同的级别（见子主题2测试级别）进行测试活动与人、工具、策略和度量组织到一个良好定义的过程中，这个过程是生命周期的一个完整部分。在IEEE/EIA标准12207.0中，测试不是一个孤立的过程，但是测试活动原则包含进了5个主要的生命周期过程和支持过程中。在IEEE1074标准中，测试与其它评价活动一起，是整个生命周期活动的一个完整部分。

5.1.4 测试文档与工作产品[Bei90:c13s5; Kan99:c12; Per95:c19; Pfl01:c9s8.8] (IEEE829-98)

文档是测试过程形式化的一个完整部分，IEEE软件测试文档标准(IEEE829-98)为测试文档、文档间相互关系、文档与测试过程关系，提供了很好的描述。测试文档可能包括：测试计划、测试设计规格说明、测试过程规格说明、测试用例规格说明、测试日志、测试事件或问题报告。被测试的软件作为测试项文档。测试文档应该以与软件工程其它类型文档一样的质量水平来生成和连续更新。

5.1.5 内部与独立测试小组[Bei90:c13s2.2-c13s2.3; Kan99:c15; Per95:c4;Pfl01:c9]

测试过程的形式化可能涉及测试小组组织的形式化。测试小组可以由内部成员（即项目组，可能或不涉及软件构造）组成，也可以由外部成员组成，以带来无偏见的、独立的观点，或者同时由内部和外部成员组成。对成本、进度、涉及到的组织的成熟度级别和应用的关键性的考虑，会影响决策。

5.1.6 成本/工作量估计，其它过程度量[Per95:c4,c21] (Per95: Appendix B; Pos96:p139-145; IEEE982.1-88)

管理人员常使用几个与测试耗费的资源、不同测试阶段的发现错误的相对有效性有关的度量，来控制和改进测试过程。这些测试度量可能覆盖这些方面：指定的测试用例数目、执行的测试用例数目、通过的测试用例数目、没通过的测试用例数目等。

测试阶段评价报告可以与原因分析组合在一起，以尽可能评价测试过程对于发现错误的有效性。这种评价可能与风险分析相关。还有，值得耗费在测试上的资源应该与应用的使用/关键性相称：不同的技术有不同的成本，因而产生产品可靠性的不同置信度。

5.1.7 终止[Bei90:c2s2.4; Per95:c2]

必须作出多少测试就足够了和什么时候终止测试的决策。彻底性度量（如达到的代码覆盖率或功能完备性）以及错误密度估计、运行可靠性估计，可以提供有用的支持，但这些本身并不充分。决策也涉及到对残余失效带来的成本和风险的考虑，以及继续测试的成本。参见子主题1.2.1测试选择准则/测试适当性准则。

5.1.8 测试复用与测试模式[Bei90:c13s5]

为了以有组织的、成本/效益比好的方式完成测试或维护，用于测试软件每个部分的手段必须系统地复用。这个测试材料的仓库必须处于软件配置管理的控制下，这样，软件需求或设计的变更可以反映到进行的测试的变更上。

用于在特定环境下的应用类型的测试解决方案，以及决策时的动机，形成了一个测试模式，它本身可以文档化，并在以后类似的项目中复用。

5.2 测试活动

在这个主题下，给出测试活动的一个简要总结，正如以下描述隐含的那样，测试活动的成功管理强烈依赖于软件配置管理过程。

5.2.1 计划[Kan99:c12;Per95:c19;Pfl01:c8s7.6] (IEEE829-98:s4; IEEE1008-87:s1-s3)

与项目管理的任何其它方面一样，必须对测试活动进行计划。测试计划的关键方面包括人员协调、可用的测试设施和设备（可能包括磁介质、测试计划和过程）的管理、对可能的不期望的结果的计划。如果维持了多于一个的软件基线，则主要的计划考虑是保证测试环境设置为适当配置需要的时间和工作量。

5.2.2 产生测试用例[Kan99:c7] (Pos96:c2; IEEE1008-87:s4,s5)

测试用例的产生基于要完成的测试级别和特定的测试技术。测试用例应该在软件配置管理的控制之下，并包括每个测试的期望结果。

5.2.3 开发测试环境[Kan99:c11]

用于测试的环境应该与软件工程工具兼容，它应该加快测试用例的开发和控制，以及期

望结果、脚本和其他测试材料的日志和恢复。

5.2.4 执行[Bei90:c13; Kan99:c11] (IEEE1008-87:s6,s7)

测试的执行应该科学实验的基本原理：测试中作的每件事都应该清楚地进行和记录，使其它人员可以重复结果。因此，测试应该根据用被测试软件的清晰定义的版本进行文档化的过程来完成。

5.2.5 测试结果评价[Per95:c20,c21] (Pos96:p18-20, p131-138)

必须评价测试结果，以确定测试是否成功。多数情况下，“成功”表示软件按期望运行，并且没有重大的非期望结果。并非所有的非期望结果必须是错误，而是被认为是简单的噪声。在消除一个失效前，需要进行分析和排错，以隔离、标识和描述失效。当测试结果特别重要时，需要召集一个正式的评审委员会来评价这些结果。

5.2.6 问题报告/测试日志[Kan99:c5; Per95:c20] (IEEE829-98:s9-s10)

测试活动应该记录到测试日志，以标识测试何时进行、由谁完成、测试的软件配置基础是什么，以及其它有关的标识信息。非期望的和不正确的测试结果应该记录到一个问题报告系统中，报告中的数据是以后进行排错和修改测试中观察到的失效的问题的基础。还有，异常没有分类为错误时，也应该记录到文档中，可能它们以后会比最初认为的更严重。测试报告也是变更管理请求过程（见软件配置管理知识域的子域3软件配置控制）的输入。

5.2.7 缺陷追踪[Kan99:c6]

测试中观察到的失效多数是由于软件中的错误或缺陷引起的。可以分析这些缺陷，以确认它们是什么时候引入软件的，它们是有余什么类型的错误造成的（定义不清的需求、不正确的变量声明、内存泄漏、程序语法错误等），它们是什么时候在软件中被首次发现的。缺陷追踪信息用于确定需要改进的软件工程的一些方面，以及前面的分析和测试的有效性。

主题与参考文献矩阵

	[Bec02]	[Bei90]	[Jor02]	[Kan99]	[Kan01]	[Lyu96]	[Per95]	[Pfl01]	[Zhu97]
1软件测试基础									
1.1测试相关的术语									
测试和相关术语的定义		C1	C2			C2s2.2			
故障与失效			C2			C2s2.2	C1	C8	
1.2关键问题									
测试选择准则/测试适当性准则(测试停止标准)								C8s7.3	S1.1
测试有效性/测试客观性		C1s1.4					C21		
为标识缺陷而测试		C1		C1					

神喻问题		C1							
测试的理论和实践局限				C2					
不可达路径问题		C3							
可测试性		C3,c13							
1.3测试与其它活动的联系									
测试与静态分析技术		C1					C17		
测试与正确性证明、形式化验证		C1s5						C8	
测试与调试		C1s2.1							
测试与编程		C1s2.3							
测试与认证									
2测试级别									
2.1测试对象		C1	C13					C8	
单元测试		C1					C17	C8s7.3	
集成测试			C13,c14					C8s7.4	
系统测试			C15					C9	
2.2测试目标							C8	C9s8.3	
接收/认定测试							C10	C9s8.5	
安装测试							C9	C9s8.6	
α和β测试				C13					
遵从性测试/功能测试/正确性测试				C7			C8		
通过测试达到可靠性和评价						C7		C9s8.4	
回归测试				C7			C11,c12	C9s8.1	
性能测试							C17	C9s8.3	
压力测试							C17	C9s8.3	
背对背测试									
恢复测试							C17	C9s8	

								.3	
配置测试				C8				C9s8.3	
可用性测试							C8	C9s8.3	
测试驱动的开发	III								
3测试技术									
3.1基于测试人员的直觉和经验									
特定的测试				C1					
探索性测试					C3				
3.2基于规格说明									
等价类划分			C7	C7					
边值分析			C6	C7					
决策表		C10s3						C9	
基于有限状态机		C11	C8						
基于形式化规格说明									S2.2
随机测试		C13		C7					
3.3基于代码的测试									
基于控制流的准则		C3	C10					C8	
基于数据流的准则		C5							
基于代码测试的参考模型		C3	C5						
3.4基于错误的测试									
错误猜测				C7					
变异测试							C17		S3.2,s3.3
3.4基于使用的测试									
运行大纲			C15			C5		C9	
软件可靠性加固测试						C6			
3.6基于应用本质的测试									
面向对象的测试			C17					C8s7.5	

基于组件的测试									
基于Web的测试									
GUI测试			C20						
并发程序的测试									
协议遵从性测试									
分布式系统的测试									
实时系统的测试									
3.7测试技术的选择和组合									
功能性与结构性		C1s2.2	C1,c11s11.3				C17		
确定性与随机性									
4测试相关的度量									
4.1被测试程序的评价									
辅助测试计划和设计的程序度量		C7s4.2	C9						
故障的类型、分类和统计		C2	C1					C8	
错误密度							C20		
寿命测试,可靠性评价								C9	
可靠性增长模型						C7		C9	
4.2评价已完成的测试									
覆盖/彻底性度量			C9					C8	
错误种植								C8	
变异分值									S3.2,s3.3
不同技术的比较与相对效果			C8,c11				C17		S5
5测试过程									
5.1实际考虑									
态度/无私编程		C13s3.2						C8	
测试指导	III				C5				
测试过程管理							C1-c4	C9	
测试文档和工作产品		C13s5		C12			C19	C9s8.8	

内部与独立测试小组		C13s2.2, c1s2.3		C15			C4	C9	
成本/效益估算和其它过程度量							C4,c21		
终止		C2s2.4					C2		
测试复用与测试模式		C13s5							
5.2测试活动									
计划				C12			C19	C87s7.6	
测试用例产生				C7					
测试环境开发				C11					
执行		C13		C11					
测试结果评价							C20, c21		
问题报表/测试日志				C5			C20		
缺陷追踪				C6					

软件测试的推荐参考文献

[Bec02] K. Beck, "Test-Driven Development by Example," Addison-Wesley, 2002.

[Bei90] B. Beizer, "Software Testing Techniques," International Thomson Press, 1990, Chap. 1-3, 5, 7s4, 10s3, 11, 13.

[Jor02] P. C. Jorgensen, "Software Testing: A Craftsman's Approach," Second Edition, CRC Press, 2004, Chap. 2, 5-10, 12-15, 17, 20.

[Kan99] C. Kaner, J. Falk and H. Q. Nguyen, "Testing Computer Software," Second ed: Wiley, 1999, Chap. 1, 2, 5-8, 11-13, 15.

[Kan01] C. Kaner, J. Bach and B. Pettichord, *Lessons Learned in Software Testing*: Wiley Computer Publishing, 2001.

[Lyu96] M. R. Lyu, "Handbook of Software Reliability Engineering," Mc-Graw-Hill/IEEE, 1996, Chap. 2s2.2, 5-7.

[Per95] W. Perry, "Effective Methods for Software Testing," Wiley, 1995, Chap. 1-4, 9, 10-12, 17, 19-21.

[Pfl01] S. L. Pfleeger, "Software Engineering: Theory and Practice," Second ed: Prentice-Hall, 2001, Chap. 8, 9.

[Zhu97] H. Zhu, P. A. V. Hall and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, iss. 4, 366-427 (Sections 1, 2.2, 3.2, 3.3), Dec., 1997

附录A 深入阅读的参考文献

(Bac90) R. Bache and M. Müllerburg, "Measures of Testability as a Basis for Quality

Assurance," *Software Engineering Journal*, vol. 5, 86-92, March, 1990
(Bei90) B. Beizer, "Software Testing Techniques," Second ed: International Thomson Press, 1990.
(Ber91) G. Bernot, M. C. Gaudel and B. Marre, "Software Testing Based On Formal Specifications: a Theory and a Tool," *Software Engineering Journal*, 387-405, Nov., 1991
(Ber96) A. Bertolino and M. Marrè, "How many paths are needed for branch testing?," *The Journal of Systems and Software*, vol. 35, iss. 2, 95-106, 1996
(Ber96a) A. Bertolino and L. Strigini, "On the Use of Testability Measures for Dependability Assessment," *IEEE Transactions on Software Engineering*, vol. 22, iss. 2, 97-108, Feb., 1996
(Bin00) R. V. Binder, *Testing Object-Oriented Systems Models, Patterns, and Tools*: Addison-Wesley, 2000.
(Boc94) G. V. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," presented at ACM Proc. Int. Symposium on Software Testing and Analysis (ISSTA'94), Seattle, Washington, USA, 1994
(Car91) R. H. Carver and K. C. Tai, "Replay and testing for concurrent programs," *IEEE Software*, 66-74, March, 1991
(Dic93) J. Dick and A. Faivre, "Automating The Generation and Sequencing of Test Cases From Model-Based Specifications," presented at FME'93: Industrial-Strength Formal Method, LNCS 670, 1993
(Fran93) P. Frankl and E. Weyuker, "A formal analysis of the fault detecting ability of testing methods," *IEEE Transactions on Software Engineering*, vol. 19, iss. 3, 202-, March, 1993
(Fran98) P. Frankl, D. Hamlet, B. Littlewood and L. Strigini, "Evaluating testing methods by delivered reliability," *IEEE Transactions on Software Engineering*, vol. 24, iss. 8, 586-601, August, 1998
(Ham92) D. Hamlet, "Are we testing for true reliability?," *IEEE Software*, 21-27, July, 1992
(Hor95) H. Horcher and J. Peleska, "Using Formal Specifications to Support Software Testing," *Software Quality Journal*, vol. 4, 309-327, 1995
(How76) W. E. Howden, "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering*, vol. 2, iss. 3, 208-215, Sept., 1976
(Jor02) P. C. Jorgensen, "Software Testing: A Craftsman's Approach," Second Edition, CRC Press, 2004.
(Kan99) C. Kaner, J. Falk and H. Q. Nguyen, "Testing Computer Software," Second ed: Wiley, 1999.
(Lyu96) M. R. Lyu, *Handbook of Software Reliability Engineering*: Mc-Graw-Hill/IEEE, 1996.
(Mor90) L. J. Morell, "A Theory of Fault-Based Testing," *IEEE Transactions on Software Engineering*, vol. 16, iss. 8, 844-857, August, 1990
(Ost88) T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of ACM*, vol. 31, iss. 3, 676-686, June, 1988

(Ost98) T. Ostrand, A. Anodide, H. Foster and T. Goradia, "A Visual Test Development Environment for GUI Systems," presented at ACM Proc. Int. Symposium on Software Testing and Analysis (ISSTA' 98), Clearwater Beach, Florida, USA, 1998
(Per95) W. Perry, *Effective Methods for Software Testing*: Wiley, 1995.
(Pfl01) S. L. Pfleeger, "Software Engineering: Theory and Practice," Second ed: Prentice-Hall, 2001, Chap. 8, 9.
(Pos96) R. M. Poston, *Automating Specification-based Software Testing*: IEEE, 1996.
(Rot96) G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, iss. 8, 529-, Aug., 1996
(Sch94) W. Schütz, "Fundamental Issues in Testing Distributed Real-Time Systems," *Real-Time Systems Journal*, vol. 7, iss. 2, 129-157, Sept., 1994
(Voa95) J. M. Voas and K. W. Miller, "Software Testability: The New Verification," *IEEE Software*, 17-28, May, 1995
(Wak99) S. Wakid, D. R. Kuhn and D. R. Wallace, "Toward Credible IT Testing and Certification," *IEEE Software*, 39-47, July-August, 1999
(Wey82) E. J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, vol. 25, iss. 4, 465-470, 1982
(Wey83) E. J. Weyuker, "Assessing Test Data Adequacy through Program Inference," *ACM Trans. On Programming Languages and Systems*, vol. 5, iss. 4, 641-655, October, 1983
(Wey91) E. J. Weyuker, S. N. Weiss and D. Hamlet, "Comparison of Program Test Strategies," presented at Proc. Symposium on Testing, Analysis and Verification TAV 4, Victoria, British Columbia, 1991
(Zhu97) H. Zhu, P. A. V. Hall and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, iss. 4, 366-427, Dec., 1997

附录B 有关标准

(IEEE610.12-90) IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*: IEEE, 1990.
(IEEE829-98) IEEE Std 829-1998, *Standard for Software Test Documentation*: IEEE, 1998.
(IEEE982.1-88) IEEE Std 982.1-1988, *IEEE Standard Dictionary of Measures to Produce Reliable Software*: IEEE, 1988.
(IEEE1008-87) IEEE Std 1008-1987 (R2003), *IEEE Standard for Software Unit Testing*: IEEE, 1987.
(IEEE1044-93) IEEE Std 1044-1993 (R2002), *IEEE Standard for the Classification of Software Anomalies*: IEEE, 1993.
(IEEE1228-94) IEEE Std 1228-1994, *Standard for Software Safety Plans*: IEEE, 1994.
(IEEE12207.0-96) IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes, vol. IEEE, 1996.

//

第6章 软件维护

术语和缩写

CMMi：Capability Maturity Model Integration能力成熟度模型集成
ICSM：International Conference on Software Maintenance软件维护国际会议
SCM：Software Configuration Management软件配置管理
SQA：Software Quality Assurance软件质量保证
V&V：Verification and Validation验证和确认
Y2K：Year 2000 2000年

引言

软件开发工作的结果是交付满足用户需求的软件产品。相应地，软件产品必须变更和演化。一旦投入运行，就会发现缺陷，运行环境可能会变化，用户会提出新的需求。生命周期的维护阶段从随后的保修时期或实现后的支持交付开始，但维护活动发生的更早。软件维护是生命周期的一个完整部分，但是，在历史上，它却没有得到与其它阶段一样的重视。在历史上，多数组织中，软件开发的地位比软件维护高得多。但目前发生了变化，各个组织正在通过让软件尽可能久地运行，以减少软件开发的投资。由于2000年问题，引起了对软件维护阶段的重大关注，而且，开放源码范例进一步加强了对于维护由其它人员或组织开发的软件的关注。

在本指南中，软件维护定义为需要提供好的成本/效益的软件支持的活动的全体。这些活动包括在交付前完成的活动，以及交付后阶段完成的活动。交付前完成的活动包括交付后运行的计划、可维护性计划、迁移活动的后勤决策。交付后活动包括软件修改、培训、帮助材料的操作和接口。

软件维护知识域与软件工程所有其它方面都相关，因此，这个知识域描述与本指南所有其它章节都有链接。

软件维护主题的主题分解结构

软件维护知识域的主题分解结构如图1所示。

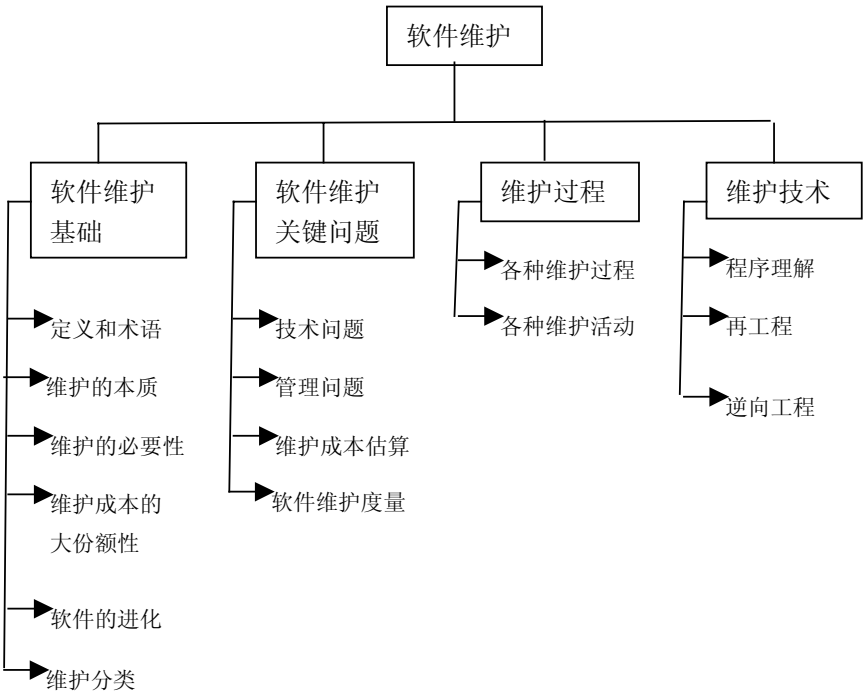

图 1　软件维护知识域主题的分解结构

1 软件维护基础

第一节介绍形成理解软件维护的作用和范围的基础的概念和术语,这个主题提供了各个定义，强调为什么需要维护。软件维护的类别对于理解其含义非常重要。

1.1 定义和术语[IEEE1219-98:s3.1.12; IEEE12207.0-96:s3.1,s5.5;ISO14764-99:s6.1]

在IEEE软件维护标准IEEE1219中,将软件维护定义为交付后,修改软件产品以更正错误、改进性能或其他属性、或适应修改后的环境。这个标准也论述了软件产品交付前的维护活动，但是在附录中给出的。

IEEE/EIA 12207软件生明周期过程标准将维护描述为生命周期过程的一个主要过程之一，并将维护描述为软件产品经历“由于一个问题或改进的需要而修改代码和相关文档，目标是修正现有的软件产品并保留其完整性”的过程。ISO/IEC 14764软件维护国际标准对软件维护的定义与IEEE/EIA 12207相同，并强调了交付前的维护方面，例如，计划。

1.2 维护的本质[Pfl01:c11s11.2]

软件维护在软件产品的整个运行生命周期维持软件产品，对于修改要记录到日志中，并能追踪，要确定建议的修改的影响范围，然后修改代码和相关的文档，进行测试，最后发布新的软版本。还要向用户提供培训和日常支持。Pfleeger在[Pfl01]指出，“维护（比开发）具有更广的范围，需要更多的追踪和控制”。

在IEEE/EIA 12207中，将维护者定义为一个完成维护活动的组织[IEEE12207.0-96]。在这个知识域中，有时，这个术语指的是完成这些活动的个人，这是相对于开发人员而言。IEEE/EIA 12207标识了软件维护的主要活动：过程实现、问题与修改分析、修改的实现、维护评审/接收、迁移、退出运行。这些活动在主题3.2维护活动中讨论。

维护者可以向开发人员学习软件的知识，与开发人员接触并尽早参与，可以减少维护工作量。有时，无法与软件工程师接触，或者他们参与了其它项目，这对维护者是一个巨大的挑战。维护者必须获得开发的产品、代码、文档，以立即支持它们，并在软件生命周期中进化和维护它们。

1.3 维护的必要[Pfl01:c11.s11.2; Pig97: c2s2.3; Tak97:c1]

需要维护来保证软件持续地满足用户需求。维护对于使用任何软件生命周期模型（如螺旋式）开发的软件都是必要的。系统由于更正性或非更正性软件行动而变更，必须进行维护，以便：（1）更正错误；（2）改进设计；（3）实现（性能的）增强；（4）与其它系统接口；（5）改编程序，以便能使用不同的硬件、软件、系统特征、电信设施；（6）迁移遗留软件；（7）让软件退出运行。

根据Pfleeger[Pfl01]，维护者的活动有4个关键特征：（1）维持对软件的日常功能的控制；（2）维持对软件修改的控制；（3）优化现有功能；（4）阻止软件性能下降到不可接受的水平。

1.4 维护成本的大分额性[Abr93:63-90; Pfl01:c11s11.3; Pig97:c3; Pre01:c30s2.1,c30s2.*2]*

维护消耗了软件生命周期财力资源的大部分。对于软件维护的一个普遍看法是它仅仅是更正错误，但是，多年来的研究和调查表明，软件维护工作量的大部分，超过80%，用于在非更正性活动上[Abr93, Pig97, Pre01]。Jones在(Jon91)中描述了软件维护管理人员在其管理报告中经常将增强和更正聚集在一起的方式。将增强请求包括在问题报告中，是由于更正成本高的误解，理解了软件维护的分类，有助于理解软件维护成本的结构。还有，理解影响一个系统中可维护性的因素，也有助于容忍成本。Pfleeger在[Pfl01]中给出了一些影响软件维护成本的技术性和非技术性因素：（1）应用类型；（2）软件新奇性；（3）软件维护人员的可获得性；（4）软件生命长度；（5）硬件特征；（6）软件设计、构造、文档和测试的质量。

1.5 软件的进化[Art88:c1s1.0,s1.1,s1.2,c11,s1.1,s1.2; Leh97:108-124],(Bel72)

Lehman在1969年首次指出软件维护和系统进化，在20年中，他的研究导致了6个“进化律的形成”[Leh97]。关键的发现包括：维护是进化性的开发，维护决策由对于系统（和软件）随时间发生的情况的理解来辅助。其他人指出，维护是持续的开发，只是有一个额外的输入（或约束）：现存的大型软件永远不会完备，需要连续进化。随软件的进化，如果不采取行动来降低其复杂性，软件将越来越复杂。

一些软件展示了有规律的行为和趋势，并且可以度量，已经有人试图开发预测性模型来估计维护工作量，这样，就开发出了有用的管理工具 [Art88], (Bel72)。

1.6 维护的类别[Art88:c1s1.2; Lie78; Dor02:v1c9s1.5; IEEE1219-98:s3.1.1,s3.1.2,s3.1.7,A.1.7; ISO14764-99:s4.1,s4.3,s4.10, s4.11,s6.2; Pig97:c2s2.3]

Lientz和Swanson最初定义了3类维护：更正性、适应性、完善性[Lie78;IEEE 1219-98] 。后来在ISO/IEC 14764软件工程标准—软件维护中进行了修改，以包括4类维护：（1）更正性维护：软件产品交付后进行的反应性修改，以更正发现的问题；（2）适应性维护：软件产品交付后进行的修改，以保持软件产品能在变化后或变化中的环境中可以继续使用；（3）完善性维护：软件产品交付后进行的修改，以改进性能和可维护性；（4）预防性维护：软件产品交付后进行的修改，以在软件产品中的潜在错误成有实际错误前，检测和更正它们。ISO/IEC 14764将适应性和完善性维护分类为增强，并将更正性和预防性维护分类为更正，如表1所示。如果安全性特别重要，软件产品中最常进行的维护是最新的预防性维护类别。

表1 软件维护分类

	更正	增强
前瞻性	预防性	完善性
反应性	更正性	适应性

2 软件维护的关键问题

为确保软件的有效维护，必须处理大量关键的问题。软件维护向软件工程师提供了独特的技术性和管理性挑战，这一点很重要。一个例子是，在不是软件工程师开发的有500K行源代码的软件中，试图发现一个错误，类似地，与软件开发人员竞争资源也是经常发生的事。计划未来的发布版本的同时，要为下一个发布版本编码，并为当前版本发送出紧急补丁，这些也是挑战。下面给出与软件维护有关的技术性和管理性问题，它们按主题标题聚集在一起：技术性问题、管理性问题、成本估计、度量。

2.1 技术性问题

2.1.1 有限的理解[Dor02:v1c9s1.11.4; Pfl01:c11s11.3; Tak97:c3]

有限的理解指的是，软件工程师能以多快的速度理解在软件中何处能作出变更或更正，且这个软件不是其本人开发的。研究指出，大约40%--60%的维护工作量用于理解要维护的软件，这样，软件理解的主题对于软件工程师有极大的兴趣。在面向文本的表示和源代码中，理解更加困难，例如，如果变更没有记录在文档中，而且没有开发人员来解释时，通过软件的发布/版本来追踪其进化，就很困难，事情也通常如此。这样，软件工程师最初可能对软件只有一个有限的理解，必须作很多工作来弥补这一点。

2.1.2 测试[Art88:c9; Pfl01:c11s11.3]

在软件的一个主要部分重复全部测试的成本在时间上和金钱上，都很大。对一个软件或组件进行回归测试或选择性重测试，以验证修改没有引起不需要的后果，这对于维护很重要。还有，找出测试的时间也很困难。当不同的维护小组同时处理不同的问题时，如何协调也是一个挑战[Plf01]。当软件运行关键功能时，可能无法在脱机状态下进行测试。软件测试知识域在子主题2.2.6回归测试中，对这个问题提供了附加的信息和参考文献。

2.1.3 影响（范围）分析[Art88:c3; Dor02:v1c9s1.10; Pfl01: c11s11.5]

影响分析描述如何以较好的成本/效益进行现有软件的变更的完备分析。维护者必须具有关于软件结构和内容的详细知识[Pfl01]，他们使用这些知识来完成影响分析，以标识软件变更请求将影响的所有系统和软件产品，并作出完成这个变更需要的资源的估计[Art88]。另外，要确定作出变更的风险。变更请求有时叫修改请求（modification request，MR），必须分析并将其翻译为软件术语[Dor02]。在变更请求进入软件配置管理过程后，就可进行变更。Arthur在[Art88]中指出，影响分析的目标是：（1）确定变更的范围，以便计划和实现这个变更；（2）对实现变更需要的资源作出精确的估计；（3）分析请求的变更的成本/效益；（4）与其他人员交流给定变更的复杂性。

通常使用问题的严重性来决定问题如何、何时解决。软件工程师然后标识受影响的组件，提出几个潜在的解决方案，建议一个最佳的行动路线。

在软件设计时就考虑可维护性，可以加速影响分析进程。可以在软件配置知识域中发现更多的信息。

2.1.4 可维护性[ISO14764-99:s6.8s6.8.1; Pfl01: c9s9.4; Pig97:c16]

如何在开发中就促进和遵循可维护性问题？IEEE在[IEEE610.12-90]中将可维护性定义为软件为满足规定需求而被维护、增强、适应修改、更正的难易程度。ISO/IEC将可维护性定义为质量的一个特征(ISO9126-01)。为降低维护成本，在软件开发活动中，必须说明、评审和控制可维护性的子特征。如果成功作到这点，就能改进软件的可维护性。由于在软件开发过程中，可维护性子特征不是重要的关注点，因此，很难达到可维护性。开发人员要关注其它事情，往往忽略维护者的需求，这通常导致，事实上也如此，缺乏系统性文档，这是程序理解和影响分析中困难的主要原因。人们已经注意到，系统性和成熟的过程、技术和工具，有助于增强系统的可维护性。

2.2 管理问题

2.2.1 与组织目标的一致[Ben00:c6sa; Dor02:v1c9s1.6]

组织的目标描述了如何表现软件维护活动的投资回报。Bennett在[Ben00]中指出，“最初的软件开发通常是基于项目的，具有确定的时间尺度和预算，主要重点是及时交付并在预算内，以满足用户需要。与之相反，软件维护的目标通常是尽可能地延长软件的寿命，另外，它也可以被满足用户对软件更新和增强的要求来驱动。在两种情况下，投资回报不很清楚，这样，高级管理层的观点是：软件维护是消耗大量资源、对组织没有清晰的可量化利润的主要活动”。

2.2.2 人员组织[Dek92:10-17; Dor02:v1c9s1.6; Par86: c4s8-c4s11](Lie81)

人员组织指的是如何吸引和保持软件维护人员，维护通常不被认为是一种有魅力的工作。基于调查数据，Deklava提出了与人员组织有关的问题的列表[Dek92]，结果是，软件维护人员通常被认为是“二等公民”(Lie81)，因而士气低落[Dor02]。

2.2.3 过程[Pau93; Ben00:c6sb; Dor02:v1c9s1.3]

软件过程是一组人们用来开发和维护软件及相关产品的活动、方法、实践和变换[Pau93]。在过程层次上，软件维护活动与软件开发有许多共同之处（例如，两种活动中，软件配置管理都很关键）[Ben00]。维护也需要软件开发中没有出现的几种活动（参加3.2节独特活动），这些活动对管理提出了挑战[Dor02]。

2.2.4 维护组织结果方面[Pfl01:c12s12.1-c12s12.3; Par86:c4s7;Pig97:c2s2.5; Tak97:c8]

组织结果方面描述了如何标识负责软件维护的组织和/或功能。一旦软件投入运行，就没有必要指定开发软件的小组来维护它。

在确定将软件维护功能放置在何处时，软件工程组织可以保持与原始开发者在一起，或转移到单独的小组（或维护者）。通常，选择维护者方式应保证软件正常运行和进化，以满足不断变化的用户需求。由于每种方式都有优缺点[Par86,Pig97]，决策时，应该具体问题具体分析，重要的是，不管组织的结构如何，应将软件维护责任委派和分配给一个小组或个人[Pig97]。

2.2.5 外包[Dor02:v1c9s1.7; Pig97:c9s9.1,s9.2], (Car94;McC02)

将维护外包已经成为一个潮流，大型公司将整个软件系统的工作进行外包，包括软件维护在内。通常的是，选择的外包部分是任务不太关键的软件，因为公司不愿意失去其核心业

务中使用的软件的控制。Carey在(Car94)中指出，一些公司只有在找到维持战略性控制的方法后，才会外包。但是，如何度量控制却很难。外包的一个主要挑战是确定需要的维护服务的范围和合同细节。McCracken在(McC02)中指出，50%的外包商在没有任何清晰的服务层协议就提供服务，外包公司花费几个月来评定软件，再签订合同[Dor02]。另一个已标识的挑战是如何将软件传递给外包商[Pig97]。

2.3 维护成本估算

软件工程师必须了解前面讨论的不同的软件维护类别，以能处理软件维护成本估算问题。对于计划而言，成本估算是软件维护的一个重要方面。

2.3.1 成本估算[Art88:c3; Boe81:c30; Jon98:c27; Pfl01:c11s11.3;Pig97:c8]

在子主题2.1.3影响范围中已经提到，影响分析要标识所有受软件变更请求影响的系统和软件产品，并估算完成变更需要的资源[Art88]。

维护成本的估算受到很多技术和非技术因素的影响。ISO/IEC14764 指出，“估算软件维护资源的两个最普遍的方法是：使用参数模型和使用经验”[ISO14764-99:s7.4.1]。通常，要组合使用几种方法。

2.3.2 参数模型[Ben00:s7; Boe81:c30; Jon98:c27; Pfl01:c11s11.3]

人们在应用参数成本模型来估算软件维护方面，做了一些工作[Boe81, Ben00]。需要来自过去项目的数据，才能使用模型，这一点很重要。Jones在[Jon98]中讨论了估算成本的各个方面，包括功能点(IEEE14143.1-00)，并为维护估算提供了详细的一章。

2.3.3 经验[ISO14764-00:s7,s7.2,s7.2.1,s7.2.4; Pig97:c8;Sta94]

专家的判断（例如，使用Delphi技术）形式的经验、类比和任务的结构分解，是用于增强来自参数模型数据的集中方法，很明显，维护估算的最好方法是组合经验 数据和经验，这些数据应该作为程序度量的结构来提供。

2.4 软件维护度量[IEEE1061-98:A.2; Pig97:c14s14.6; Gra87 ; Tak97:c6s6.1-c6s6.3]

Grady和Caswell在[Gra87]中讨论了建立公司范围内的软件度量程序，并描述了软件维护度量的表格和数据收集方法。实用软件和系统度量（Practical Software and Systems Measurement,PSM)项目描述了一个由许多组合子使用的问题驱动度量过程,很实用[McG01]。有许多对于所有感兴趣人员都普遍的软件度量，软件工程研究所（SEI）标识了以下度量：规模、工作量、进度和质量[Pig97]。这些度量为维护者提供了一个良好的起点。对于过程和产品度量的讨论在软件工程过程知识域中，软件度量程序在软件工程管理知识域中描述。

2.4.1 特定的度量[Car90:s2-s3; IEEE1219-98:Table3; Sta94:239-249]

Abran在[Abr93]中给出了比较不同内部维护组织的内部基准技术,维护者必须确定什么度量适合于其组织机构。[IEEE1219-98; ISO9126-01; Sta94]提出了更特定于软件维护度量程序的一些度量，这个列表包括可维护性的4个子特征的许多度量：（1）可分析性：度量维护者的工作量或耗费在试图诊断失效的缺陷或原因上的资源,或耗费在标识要修改部分的资源；（2）可变更性：度量与实现特定修改相关的维护者的工作量；（3）稳定性：度量软件的非期望行为，包括测试中遇到的非期望行为；（4）可测试性：度量维护者或用户在试图测试修改后软件上的工作量。

使用已有的商业工具，可以得到软件可维护性的某些度量(Lag96; Apr00)。

3 维护过程

维护过程子主题提供了勇于实现软件维护过程的参考文献和标准。维护活动主题将维护与开发分开，指出其与其它软件工程活动的关系。

软件工程过程的需要已经很好地文档化，CMMi模型可应用于软件维护过程，并与开发者的过程类似[SEI01]。文献(Apr03, Nie02,Kaj01)描述了处理软件维护的特有过程的软件维护能力成熟度模型。

3.1 维护过程[IEEE1219-98:s4; ISO14764-99:s8; IEEE12207.0-96:s5.5; Par86:c7s1; Pig97:c5; Tak97:c2]

维护过程为这些活动提供必须的活动和详细的输入输出，并由IEEE 1219和ISO/IEC 14764软件维护标准描述。软件维护标准(IEEE 1219)中描述的维护过程模型从交付后阶段的软件维护要求开始，讨论了维护计划等问题，这个过程见图2。

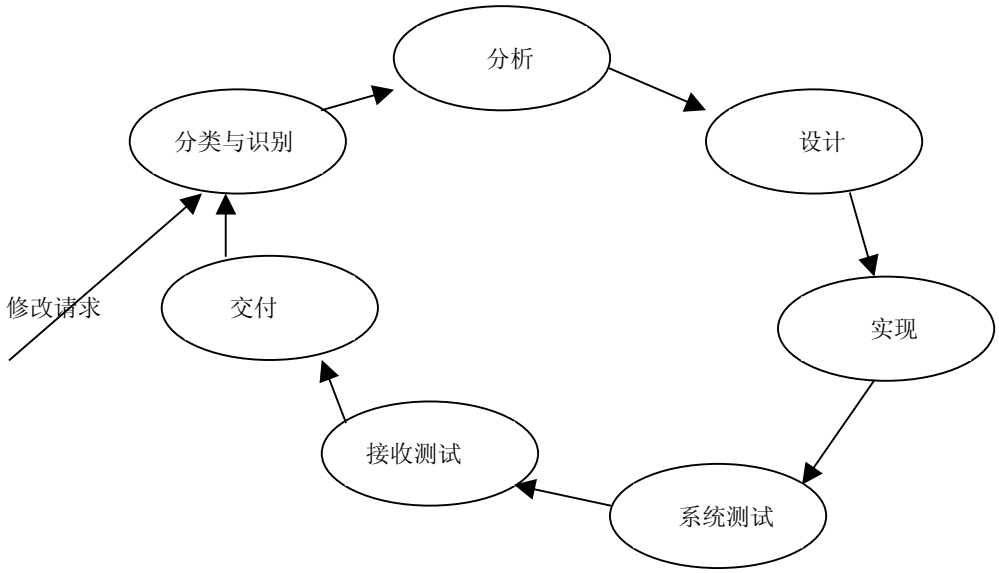

图 2　IEEE1219-98 维护过程的各种活动

ISO/IEC 14764 [ISO14764-99]是IEEE/EIA 12207.0-96维护过程的精心实现，ISO/IEC维护过程的活动类似于IEEE，只是它们聚集方式有点不同。ISO/IEC开发的维护活动如图3所示。

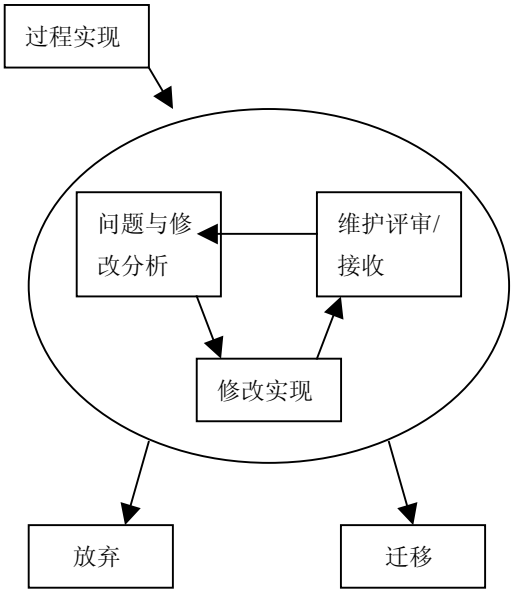

图 3　ISO/IEC14764-00 软件维护过程

ISO/IEC 14764的每个主要软件维护活动进一步分解为任务，如下：（1）过程实现；（2）问题和修改分析；（3）修改实现；（4）维护评审/接收；（5）迁移；（6）软件退出运行。

Takang和Grubb在[Tak97]中提供了维护过程模型进化到IEEE和ISO/IEC过程模型的历史。 Parikh 在[Par86]中也给出了一般维护过程的概述。最近，出现了敏捷方法，它提倡轻量级过程，这个需求来自不断增长的快速转变的维护服务要求，在(Poo01)中，给出了极限维护的一些实验结果。

3.2 维护活动

前面已经指出，许多维护活动与软件开发中的活动类似，维护者要完成分析、设计、编码、测试和文档。必须象在开发中一样，在活动中追踪需求，在基线改变时，要更新文档。ISO/IEC14764 推荐，当维护者引用一个类似的开发过程时，必须修改这个过程，以满足特定的需要[ISO14764-99:s8.3.2.1,2]，但是对于软件维护，一些活动涉及软件维护特有的过程。

3.2.1 特有活动[Art88:c3; Dor02:v1c9s1.9.1; IEEE1219-98:s4.1,s4.2; ISO14764-99:s8.2.2.1,s8.3.2.1;Pfl01:c11s11.2]

软件维护有许多特有的过活动和实践，例如：（1）迁移：是一个受控的和协调的活动序列，迁移中，软件从开发者逐步传递到维护者[Dek92, Pig97]；（2）修改请求的接收/拒绝：在一定规模/工作量/复杂性上的修改请求可能被维护者拒绝，并重新转移给开发者[Dor02], (Apr01)；（3）修改请求和问题报告的帮助桌面：这是一个终端用户支持功能，可以触发修改请求的评定、优先次序和成本分配[Ben00]；（4）影响分析（详细内容参见2.1.3）；（5）软件支持：帮助和向关心某个请求的用户得到信息（例如，业务规则、确认、数据含义和特定的请求/报告）(Apr03)；（6）服务级别协议（Service Level Agreements，SLA）和专门（特定领域的）维护合同：这是维护者的责任(Apr01)。

3.2.2 支持活动[IEEE1219-98:A.7,A.11; IEEE12207.0-96:c6,c7;ITI01;

Pig97:c10s10.2,c18] ;(Kaj01)

维护者可能也要完成一些支持活动，如软件维护计划、软件配置管理、验证和确认、软件质量保证、评审、审计和用户培训。还需要另一种支持活动：维护培训[Pig97;IEEE12207.0-96] (Kaj01)。

3.2.3 维护计划活动[IEEE1219-98:A.3; ISO14764-99:s7; ITI01;Pig97:c7,c8]

软件维护的一个重要活动是计划，维护者必须处理与大量计划观点相关的问题：（1）业务规划（组织层次上的）；（2）维护计划（迁移层次上的）；（3）发布/版本计划（软件层次上的）；（4）单个软件变更请求计划（请求层次上的）。

在单个请求层次上，计划是在影响分析中完成的（参见子主题2.1.3影响分析）。发布/版本计划活动要求维护者完成：（1）收集单个请求的可获得日期；（2）与用户达成对随后的发布/版本的一致；（3）标识潜在的冲突，提出替代方案；（4）评定给定发布的风险，并作出出现问题时的回收计划；（5）通知所有干系人。

软件开发项目一般持续几个月到几年，而维护阶段通常持续多年。估算资源是维护计划的关键因素，这些资源应该包括到开发人员的项目计划预算中。软件维护计划应该从决定开发一个新系统开始，并考虑质量目标(IEEE1061-98)。应该开发一个概念文档，然后作出维护计划。维护的概念文档应该包括[ISO14764-99:s7.2]：（1）软件维护范围；（2）软件维护过程的修改；（3）软件维护组织的标识；（4）软件维护成本估算。

下一步是开发一个相应的软件维护计划，这个计划应该在软件开发中准备完毕，并应该指明用户如何请求软件修改或报告问题。IEEE 1219 [IEEE1219-98]和ISO/IEC 14764论述了软件维护计划[Pig97]。

最后，在最高层次上，维护组织必须进行业务规划活动（预算、财务和人力资源），如同组织中所有其它部门一样。完成这些工作需要的管理知识可以参见软件工程相关学科一章。

3.2.4 软件配置管理[Art88:c2,c10; IEEE1219-98:A.11; IEEE12207.0-96:s6.2; Pfl01:c11s11.5; Tak97:c7]

IEEE软件维护标准IEEE 1219[IEEE1219-98]中，将软件配置管理描述为维护过程的关键因素。软件配置管理过程为软件产品的标识、授权、实现和发布需要的每一步提供验证、确认和审计。

简单地追踪修改请求或问题报告是不够的，必须控制软件产品和对其作的任何变更。通过实现和强制一个批准的软件配置管理过程，可以建立这种控制。软件配置管理知识域提供了软件配置管理的细节，讨论了提交、评价和批准软件变更请求的过程。软件维护的软件配置管理与软件开发的软件配置管理不同之处是：在运行的软件上必须控制的小变更的数量。软件配置管理过程的实现是：开发和遵循配置管理计划和操作过程。维护者要参与配置控制委员会，以确定下一个发布/版本的内容。

3.2.5 软件质量[Art98:c7s4; IEE12207.0-96:s6.3; IEEE1219-98:A.7; ISO14764-99:s5.5.3.2]

同样，简单地希望软件的维护可以提高软件质量也是不够的，必须进行计划并实现规定的过程，以支持维护过程。必须选择软件质量保证、验证和确认、评审和审计的活动和技术，于其它过程协同，以达到希望的质量等级。建议维护者适应性地修改软件开发过程、技术和可交付产品（如测试文档和测试结果）[ISO14764-99]。

详细内容参见软件质量知识域。

4 维护技术

这个子域介绍一些软件维护中普遍接受的技术。

4.1 程序理解[Arn92:c14; Dor02:v1c9s1.11.4; Tak97:c3]

程序员在阅读和理解程序上，花费大量时间，以便实现变更。代码浏览器是程序理解的关键工具。清晰和简明的文档有助于程序理解。

4.2 再工程[Arn92:c1,c3-c6; Dor02:v1c9s1.11.4; IEEE1219-98:B.2], (Fow99)

再工程定义为检查和修改软件，用新的形式重构软件。Dorfman和Thayer在[Dor02]中指出是修改的最激进（和昂贵）形式，其他人认为，再工程可用于少量的变更，通常不是用再工程来提高可维护性，而是用来替代老化的遗留软件。Arnold在[Arn92]中提供了一个综合性的主题概要，例如：与再工程有关的概念、工具和技术、实例研究、风险与效益。

4.3 逆向工程[Arn92:c12; Dor02:v1c9s1.11.3; IEEE1219-98:B.3;Tak97:c4, Hen01]

逆向工程是分析软件，标识软件的组件及其相互关系，以另一种形式或在更高的抽象层次上创建软件的表示的过程。逆向工程是被动的，它不改变软件，也不产生新软件。逆向工程的结果是来自源代码的调用图和控制流图。一种类型的逆向工程是重写文档，另一类型是恢复设计[Dor02]。重构（Refactoring）是程序变换，它重新组织程序而不改变其行为，也是一种逆向工程，目标是改进程序结构(Fow99)。

最后，在过去几年，数据逆向工程得到重视，它是从物理数据库中恢复逻辑模式。

主题参考文献矩阵

	[Abr93]	[Arn92]	[Art88]	[Ben00]	[Boe81]	[Car90]	[Dek92]	[Dor97]	[IEEE610.12-90]	[IEEE1061.98]
1 软件维护基础										
1.1 定义与术语	63-90			S4						
1.2 维护的本质										
1.3 维护的必要										
1.4 维护成本的大份额性	63-90									
1.5 软件的进化					C1s1.0-c1s1.2,c11s1.1,c11					

					s1.2					
1.6 维护的类别	63-90				C1s1.2	S5				V1c9s1.5
2 软件维护的关键问题										
2.1 技术性问题						S6c				
有限的理解						S11.4				V1c9s1.11.4
测试					C9					
影响分析					C3	S10.1-s10.3				V1c9s1.10.1-v1c9s1.11.3
可维护性		S3				S5,s9.3,s11.4				
2.2 管理性问题										
与组织目标的协调一致						S6a				V1c9s1.6
人员雇佣									10-17	V1c9s1.6
过程						S6b				V1c9s1.3
维护的组织方面						S6a,s7				
外包						S7				V1c9s1.7
2.3 维护成本估算										
成本估算					C3	S7	C30			
参数模型						S7	C30			
经验										
2.4 度量	63-90		A2			S9.1,s9.2,s10.1				

特定度量						S9.2		S2,s3		
3 维护过程										
3.1 各种维护过程								S8		
3.2 维护活动								S9.1		
特有活动	63-90				C3	S6b				V1c9s1.9.1
支持活动										
维护计划活动										
软件配置管理					C2,c10	S2,s6b,s9.2,s10.1,s11.4				
软件质量	63-90				C7s4	S7,s8,s9.2,s9.3,s11.4,s11.5				
4 维护技术										
4.1 程序理解				C14		S11.4				V1c9s1.11.4
4.2 再工程				C1,c3-c6		S9.2,s11.4,s11.5				V1c9s1.11.4
4.3 逆向工程				C12		S9.2,s10.3,s11.2,s11.3,s11.5				V1c9s1.11.3

	[IEEE1219-	[IEEE12207	[ISO14764-	[Jon98]	[Leh97]	[Par86]	[Pfl01]	[Pig97	[Pre04	[Sta94	[Tak97]

	98]	.0-96]	00]					]	]	]	
1 软件维护基础											
1.1 定义与术语	S3.1.12		S6.1								
1.2 维护的本质		S3,s5.5					C11s11.2				
1.3 维护的必要							C11s11.2	C2s2.3			C1
1.4 维护成本的大份额性							C11s11.3	C3	C31		
1.5 软件的进化					108-124						
1.6 维护的类别	S3.1.1,s3.1.2,s3.1.7,A1.7		S4.1,s4.3,s4.10,s4.11,s6.2					C2s2.3			
2 软件维护的关键问题											
2.1 技术性问题											
有限的理解							C11s11.3				C3
测试							C11s11.3				
影响分析							C11s11.5				
可维护性			S6.8,s6.8.1				C9s9.4	C16			
2.2 管理性问题											
与组织目标的协调一致											
人员雇佣						C4s8-c4s11					

过程											
维护的组织方面						C4s7	C12s12.1-c12s12.3	C2s2.5			C8
外包								C9s9.1-c9s9.2			
2.3 维护成本估算											
成本估算				C27			C11s11.3	C8			
参数模型				C27			C11s11.3				
经验			S7,s7.2,s7.2.1,s7.2.4					C8			
2.4 度量								C14s14.6			C6s6.1-c6s6.3
特定度量	Table 3									239-249	
3 维护过程											
3.1 各种维护过程	S4		S8			C7s1		C5			C2
3.2 维护活动		S5.5									
特有活动	S4.1-s4.2		S8.2.2.1,s8.3.2.1				C11s11.2				
支持活动	A7,A11	C6,c7						C10s10.2,c18			
维护计	A3		S7					C7,			

划活动								c8			
软件配置管理	A11	S6.2					C11s11.5				C7
软件质量	A7	S6.3	S5.5.3.2								
4 维护技术											
4.1 程序理解											C3
4.2 再工程	B2										
4.3 逆向工程	B3										C4

软件维护的推荐参考文献

[Abr93] A. Abran and H. Nguyenkim, "Measurement of the Maintenance Process from a Demand-Based Perspective," *Journal of Software Maintenance: Research and Practice*, vol. 5, iss. 2, 63-90, 1993

[Arn93] R. S. Arnold, *Software Reengineering*: IEEE Computer Society, 1993.

[Art98] L. J. Arthur, *Software Evolution: The Software Maintenance Challenge*: John Wiley & Sons, 1988.

[Ben00] K. H. Bennett, "Software Maintenance: A Tutorial," in *Software Engineering*, M. Dorfman and R. Thayer, Eds.: IEEE Computer Society Press, 2000.

[Boe81] B. W. Boehm, *Software Engineering Economics*: Prentice-Hall, 1981.

[Car90] D. N. Card and R. L. Glass, *Measuring Software Design Quality*: Prentice Hall, 1990.

[Dek92] S. Dekleva, "Delphi Study of Software Maintenance Problems," presented at Proceedings of the International Conference on Software Maintenance, 1992

[Dor02] M. Dorfman and R. H. Thayer, Eds., "Software Engineering." (Vol. 1 & vol. 2), IEEE Computer Society Press, 2002.

[Gra87] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing A Company-Wide Program*. Englewood Cliffs NJ, USA: Prentice-Hall, 1987.

[IEEE610.12-90] IEEE Std 610.12-1990 (R2002), *IEEE Standard Glossary of Software Engineering Terminology*: IEEE, 1990.

[IEEE1061-98] IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*: IEEE, 1998.

[IEEE1219-98] IEEE Std 1219-1998, *IEEE Standard for Software Maintenance*: IEEE, 1998.

[IEEE12207.0-96] IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, *Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-Software Life Cycle Processes*, vol. IEEE, 1996.

[ISO9126-01] ISO/IEC 9126-1:2001, *Software Engineering-Product Quality-Part 1: Quality Model*: ISO and IEC, 2001.