前言

理论研究与实践的桥梁

自从 1968 年 NATO 会议首次提出“软件工程”概念以来，它经历了一条漫长的道路。在几十年前，“软件”这个概念本身还不能被多数人接受。因而软件工程理论研究和实践必须建立一个坚固的统一标准使得人们懂得在我们现今生活中如何建立良好软件和怎样评价软件的风险、概率。本文融合了当前两种软件工程的潮流：从实践者角度，实践者的焦点在于建立高质量的软件产品，提供实用的功能；从研究者角度，侧重于寻找提高质量途径，提高实践者的生产效率。

本书用于研究生软件工程教材，描绘了实用的软件工程理论和实践概况，由于学生的经历有限，本书中所举的例子可能是超出我们的经验，但这些例子足以清楚地阐述大型软件项目从设计到实现的整个开发过程。

此书还可作为本科生软件工程概念和实践的入门教材，或用于软件开发人员扩充该领域知识。本书中涵盖的各种样例：大型项目，小型项目，面向对象和面向过程，实时处理，事务处理，开发案例，维护，适合各种读者群。12 章、13 章和 14 章提供的材料用于激励学生启发思想，培养研究兴趣。

核心特征

与其他书相比本书具有如下特征：

- 本书将许多评价标准综合运用于软件工程，测量标准是软件工程策略的完整部分，不能孤立看待。这种综合看待软件工程测量标准的办法可以使学生学会如何将定量分析，定量改进运用到日常活动中。可以评价在个人方面、团体以及项目基础上的进步。
- 本书将许多概念，如：重用、风险管理、质量工程融于软件工程中，而非分裂处理。
- 每章用两个实例说明该章中的主要概念，两个例子均来源于实际的项目。信息系统实例描述了一个软件系统怎样确定一家英国大型电视公司广告时间价格，实时系统实例给出 Ariane-5 火箭控制软件；在这些实例的问题报告中，我们还可以探索软件工程中的技术怎样定位问题所在及如何解决、避免这些问题。学生可以从这些实例中学到如何把软件工程技术运用到实际的系统中。
- 每章末尾，给出该章主要内容对于小组开发的意义、个人开发意义、研究意义。学生可以选择阅读，查找相关部分。
- 本书给出相关的网址，文献，网上相关的工具，方法和学习指南。从网上学生可获得许多实际的需求文档、设计、代码、测试计划等相关信息。一些声誉较好的网站上还有进一步深入的信息。
- 本书包含许多实例和文献中的样例。其中的简略例子详细内容可在相关网页上查询。从中可了解理论概念是怎样运用于实践的
- 每章末尾给出启发式问题，这些问题涉及到软件工程的合法与伦理等方面。学生可以从社会、政治环境出发考虑这些问题。和其他科学一样，必须从他给人们生活带来的后果角度看待软件工程决策。
- 面向过程和面向对象两种思想方法在每章中都有体现。此外将有一章专门阐述面向对象的发展过程，面向对象的开发过程。此处使用 UML 描述通用概念。面向对象开发的每

一步均有实例说明。

- 本书给出注解文献的出处，网址，讨论小组以及专业领域如：软件可靠性、容错、计算机安全等的相关联接。
- 本书给出解决方案手册，可以在 Prentice Hall 得到，Power Point 格式。
- 每章介绍一个项目，比如抵押处理软件系统开发，老师可以针对这些项目介绍，项目变体作为课堂作业。
- 每章后给出概念索引。

内容与组织

本书分为三部分：第一部分（第一章至第三章）启发读者阐述软件工程知识对于实践者和研究人员的重要性，讨论了问题理解，项目计划意义；第二部分（第四章至第十一章）详细阐述开发维护主要步骤，可以不考虑创建软件的处理模型：需求检查、需求获得、设计问题解决方案，代码编写和测试、提交用户；第三部分（第十二章到第十四章）集中讨论评价与改进。这里将阐述我们如何看待软件产品的质量和怎样提高质量。

第一章：为何需要软件工程

在本章中，我们首先说明每种关键问题均出现在后面的那些章节中。然后参考 Wasserman’s 的核心因素给出软件工程的定义：抽象、分析、方法设计、专用符号、模块和体系结构，软件生命周期、出版，重用、测量，工具，环境集成，用户界面。接着讨论计算机科学和软件工程之间的差别，解释一些可能遇到的问题，给本书其它部分打下地基。最后阐明了实用系统方法建立软件的必要性，给出的两个实例是各章中都将用到的，同时给出这些实例的工程背景。

第二章：过程模块与生命周期

给出各种不同类型的处理和生命周期模块概要，包括：瀑布模式，V 模式，螺旋模式以及其他原型。我们还将讨论几种建模技术，工具，包括系统动力，SADT 和常用方法。对于两个实例我们都给出模块分析。

第三章：项目计划与管理

本章主要讲解项目计划和进度安排。引入几个概念，比如：工作量，里程碑，进度安排表，任务图，风险管理，成本估算。同样我们将用估算模型评价两个实例的成本代价。集中于 F-16 飞行器软件开发系统和 Digital’s alpha AXP 项目的软件开发与管理的成本估算。

第四章：需求分析

本章讲解需求分析和需求说明书，阐明功能需求与非功能需求的差别，分别用几种不同的方式说明他们之间的差别，讨论如何建立需求原型。并且使用各种正式的方法说明和评价需求。此外还包括需求文档书写，需求文档回顾，需求质量及评价，需求可测性。

第五章：系统设计

本章主要考虑系统结构问题。首先讨论 Shaw 和 Garlan 的软件体系结构框架。接着描述概念设计和技术设计的区别。讨论负责设计的人员的角色，两种基本设计方法：组合法与分解法。然后给出良好设计特征，介绍几个设计策略，给出若干系统设计技术的实例，工具。在本章中读者还将学到客户-服务器体系结构，可重用设计组件，人机接口设计，安全与可靠性设计（包括出错处理和容错技术），设计模式，正式的设计方法，设计协议评价。在解释了如何评价设计质量和正确性证明，怎样书写结果文档，我们转向代码设计阶段。

代码设计分别用模块化设计和独立设计用两种方法：自顶向下，自底向上解释，并给出逻辑设计和物理设计的区别。针对并发与安全性要求较高的系统，我们检查其设计上的因差错而导致的 Therac-25 的功能故障。举出若干设计工具，彻底讨论设计质量以及怎样衡量。最后结合信息系统和时实系统两个实例给出软件设计的实例。

第六章：关于对象

第六章从间接的角度考虑面向对象开发的特殊性质。我们先给出使用案例的背景，讨论如何从需求中获得对象、对象特征。其次要检查系统设计。接着扩充系统设计，加入非功能性需求，编程设计的代码细节。使用 UML 和构造图，我们可以产生面向对象的系统说明和系统设计，这里所用的实例是空军服务站系统。

对于面向对象开发的评价，我们使用普通的面向对象规则评价服务站系统。可以从中学到如何在规则中加入适当的改变有助于我们决定如何分配资源，寻找错误。

第七章：编写代码

在本章中将讲解如何编写高质量的代码实现系统设计。将着重讨论代码编写标准、编写过程、提倡使用简单实用的编程指导。在这里给出两种类型语言的编程实例：面向对象和面向过程。并讨论代码文档的必要性，错误处理措施。

第八章：程序测试

本章将从不同侧面考虑程序测试，比较两种方法，确认软件系统。给出软件问题定义，分类。分类方法怎样使数据采集，数据分析更加有效。解释单元测试和整体测试的区别。引入若干软件自动测试工具和技术，测试生命周期的必要，以及如何将这些工具、技术集成到系统中。

第九章：系统测试

首先给出系统测试的原则，包括测试和数据的重用性，配置管理。所引入的概念还包括：功能测试、性能测试、确认测试、安装测试。同时分析了面向对象系统的特殊测试需求。这里给出几个测试工具，测试小组的成员讨论内容。接下来介绍软件可靠性模型，可靠性问题，软件可维护性，适用性。读者可从中学会如何使用测试结果评价提交产品可能具有的特征。

第十章：系统递交

本章讲解培训与文档记录的必要性。

第十一章：系统维护

本章我们涉及到系统变化问题，系统变化在系统生命周期中怎样产生，随之而来的系统设计、代码、测试处理、文档变化。并讨论典型的系统维护问题配置管理的必要性。并彻底讨论了对可能出现变化及变化所带来后果的评测。

第十二章：产品，过程，资源评估

因为许多软件工程决策涉及现存组件集成与整合，那么就需要一种方法评价过程与产品。在这里我们给出经验法评估以及若干评价策略。这些规则用来建立质量和生产力的基线。在这里使用几个质量模型，评价系统可重用性，后期使用，理解信息技术投资的回报。

第十三章：预测，处理和资源的改进

本章建立在第十一章基础上，包括几个比较深入的实例用来表示预测模型，检测技术，可以扩充软件工程的其他方面的理解并有助于提高投资技术水平的提高。

第十四章：软件工程的前景

在最后一章，我们探索几个软件工程领域的若干公开问题。重温 Wasserman 的几个概念重新看待将软件工程作为一门学科我们在相关行业做得如何。此外还要讨论在研究成果转化成实际应用时若干技术转移问题和决策制定的改进与提高。

致谢

感谢朋友和家人给我的技术与情感上的支持。对于不能够将所有在编写本书过程中曾给过我帮助和支持的人的名字列在这里深表遗憾和歉意。Carolyn Seaman (马里兰大学-巴尔的摩校区)出色的评论家，提出简化澄清的种种方法，帮助我写出更加紧凑，更易于理解的内容，此外她还给出了绝大多数习题的答案，并给该书分配了网址。在此衷心感谢她的友善和帮助。Forrest Shull (Fraunhofer Center) 更新答案使它们能够反映最新的习题与材料。Yiqing Liang 修复了网上的连接错误，添加了新的材料。Carla Valle，来自里约热内卢联合大学，更新网址并更新了网上资源。

尤其要感激的是 Guilherme Travassos，他允许我们使用在马里兰大学-College Park 共同开发的项目材料。感激 Manny Lawrence，实时空军服务站系统经理，以及他的簿记员 Bea Lawrence，感谢他们和我们这些共同的工作和生活经历。

习题

1 怎样用处理模型的相关概念描述一个系统？例如：如何确定一个用处理模型描述的系统的上届？

2 对本章中的每个模型，分别列举其优点和缺点。

3 本章中的每个模型是如何处理未来需求分析的改变。

4 请画出一个商务旅行的机票订购处理图。

5 画出 Lai 公司人造用品表，要求包括人造用品没经测试状态、部分经过测试状态，完全经过测试状态。

6 用自己选择的符号画出软件系统的开发过程，给出三种不同的处理原型，从中选择一个最佳的。

7 仔细考虑 2.4 节处理模型的特征，这里的特征对于问题和解决方案还没经彻底理解的项目非常重要。

8 在本章中，我们强调软件开发是一个创造性的过程，而非一个完全的生产过程。讨论一现代有软件开发的生产的特征，并解释为何软件开发是一个创造性的过程。

9 一个开发组织可以采用一个单一的处理模式处理所有的软件开发，讨论它的正面和反面作用。

10 假如与某用户的合同指明要求使用某个特殊的软件开发工具，那么这项工作应该怎样管理？

11 考虑本章对处理的介绍。哪一个对于需求变化反映的灵活性最大。

12 假设 Amalgamated 公司在签订创建一个系统时合同中指明要使用某给定的模型。你答应使用事先规定的这些人力、物力资源和软件资源。在软件交付和安装后，系统遇到了一个灾难性的错误。当 Amalgamated 公司开始调查错误原因时，你被指责没有做代码检查，这些检查可以发现递交时问题所在。你反驳：城代码检查并不在所要求得的处理中。那么在这场争论中法律问题和伦理问题的焦点是什么？

第 1 章 为什么进行软件工程?

在本章，我们来看一下：

- 软件工程意味着什么
- 软件工程的发展
- “好软件”意味着什么
- 一个系统方法为什么重要
- 自 20 世纪 70 年代来软件工程的变化

软件遍及我们的世界，我们有时把软件在使我们的生活更舒适、有效率的过程中所扮演的重要角色视为当然。例如，考虑为早餐准备烤面包这样的简单任务。烤箱中的代码控制面包变褐的程度及何时成品出箱。程序调控屋子的供电，软件为我们的能源消费记帐。实际上，我们可能使用自动程序付电费、定购更多杂货甚至买新的烤箱。事实上，现在软件或者显式的或者在幕后为我们生活的各方面服务，包括那些影响我们健康和财富的重要系统。因此，软件工程显得比以往更加重要。好的软件工程的实施必须确保软件在我们引导我们的生活中做出积极的贡献。

本书着重于软件工程中的关键问题，描述了我们所知道的有关技术和工具方面的东西，以及它们是怎样影响我们建造和使用的那些产品的。我们从理论和实践上来看一看：我们知道的东西和它如何应用于一个典型的软件开发或维护工程中的。我们也检查一下我们还不知道的东西，而这些东西将有助于使我们的产品更可靠，安全、有用和易于理解。

我们从考虑我们怎样分析问题和提出解决方案方面着手。然后我们研究一下计算机科学问题和工程问题间的差异。而最终的目标是得到生成高质量软件的方案，并且我们还考虑为这种质量做出了贡献的特征。

我们也着眼于我们作为软件系统开发者已有的成败。通过检查几个软件失败的例子，我们来弄明白我们已经走得多远、及为了掌握质量软件开发的艺术我们还将必须走更远。

接着，我们考虑一下与软件开发相关的人。在描述顾客、用户和开发者的角色和职责后，我们研究系统本身。我们知道一个系统可被视作与一活动集相关并用某种边界封装的一组对象。作为选择，我们用一个工程师的眼睛来看待一个系统；开发一个系统恰似修建一座房屋。在确定了建立系统的系列步骤后，我们讨论开发小组在每步中的角色。

最后，我们讨论一些已经对我们实施软件工程的方式产生影响的那些变化。我们介绍 Wasserman 的八条思想把我们的实践联系在一起以形成一个连贯的整体。

1.1 什么是软件工程？

作为软件工程师，我们运用计算机和计算的知识来帮助解决问题。通常要处理的问题与计算机或现存计算机系统有关，但有时问题中潜在困难与计算机没有关系。因此，首先理解问题的本质是很重要的。特别地，我们必须要小心，不要把计算机械和技术强加于我们遇到的每个问题上。首先必须解决这个问题。然后，如果需要，使用技术作为工具来实现我们的解决方案。在本书其他部分，我们假定我们的分析已经表明某些种类的计算机系统对解决手头一类特别问题是必须且使人满意的。

解决问题

大多数问题是庞大的，有时处理起来棘手，特别是如果它们提出了某些以前从没解决过的新东西。因此我们必须通过分析来对它开始进行调查，也就是把问题分解成我们能理解并尽量能处理的问题片。因而我们能够把这个大问题用小问题集和它们间的相互关系来描述。图 1.1 解释了如何分析。重要的是要记住关系（图中的箭头，及子问题的相对位置）跟子问题本身一样重要。有时，正是关系保持着怎样解决大问题的线索，而不简单是子问题的本身特性。

一旦分析了问题，我们必须从表述问题各方面的部件来构建解决方案。图 1.2 解释了这个相反的过程：综合(Synthesis)，就是把从小建筑块装配成一个大建筑。随着分析，单个解决方案的合成可能跟寻找这些方案本身一样富有挑战性。为了弄清为什么，考虑写一本小说的过程。字典包含了所有你可能想在作品里使用的词。但是，写作最困难的部分就是决定怎样组字成句，同样的，组句成段、章以形成这部完整的书。因此，任何解决问题的技巧必须有两部分：分析问题以弄清它的本质，然后基于分析综合/合成出一个解决方案。

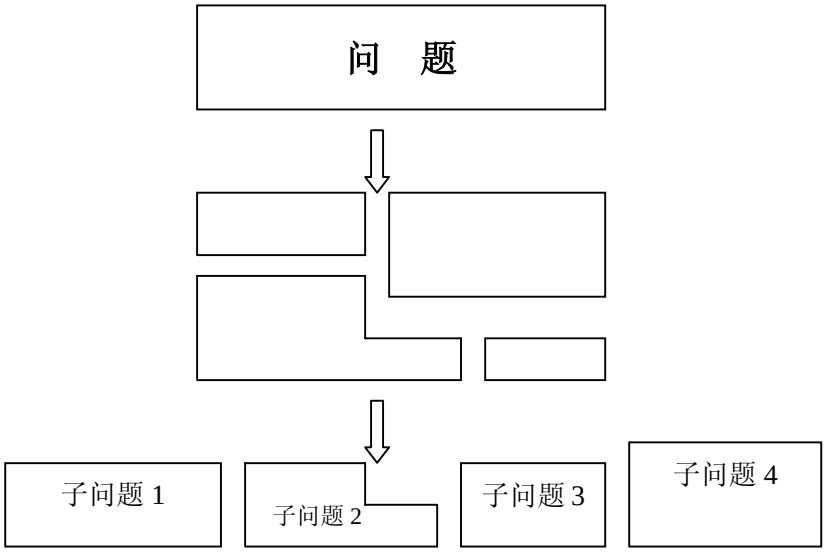

图 1.1 分析过程

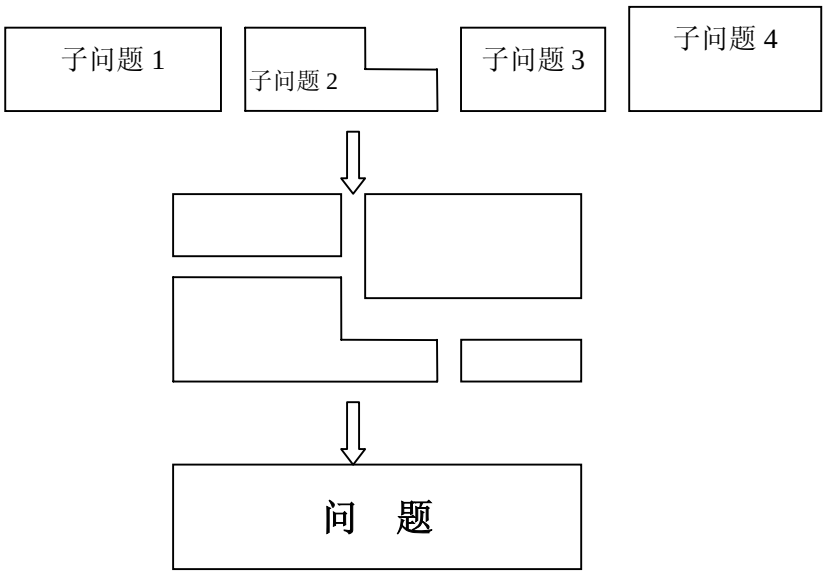

图 1.2 综合过程

为了有助于解决问题，我们使用各种方法（method）、工具（tool）、程序（procedure）及范例（paradigm）。一种方法（method/technique）就是一种用于产生某种结果的正式的程序（procedure）。例如，一个厨师准备调料时，他使用一系列成分并结合一套仔细排好的时、序步骤以便调料变浓但又不凝固或分散。准备调料的程序与时间选择和成分有关但可能并不依赖于使用何种类型的烹饪设备。

一件工具（tool）是一件能以更好的方式完成某件事情的设备或自动化系统。在这里“更好方式”可能意味着这件工具使用我们更准确、更有效率、或更多产、或增强了所得产品的质量。例如，我们使用打字机或键盘和打印机来写信是因为所得文档比手写更易于阅读。或者我们用一把剪刀作为一种工具是因为比起撕一张纸使用剪刀会剪得更快更直。然而，工具并不总是更好地做某件事所必须的。例如，一个烹饪方法能做出更好的调料而不是厨师所使用的罐也非勺。

一个程序（procedure）就象一个秘诀：是一致地产生特别产品的工具和方法的组合。打个比方，就象将在后面章节看到的，我们的测试计划描述了我们的测试程序；它们告诉我们在何种情形下用何种工具作用在何种数据集上以便让我们检查出软件是否满足要求。

最后，范例就象一种烹饪风格；它提供了一个特别的构建软件的方案或哲学。恰就象我们能区别出法国烹饪和中国烹饪一样，我们也能区别出象面向对象开发和过程化开发这样的范例。一种范例比不比另一个好；每一种都有它自己的优势和劣势，一个比另一个更恰当是有条件的。

软件工程师使用工具、方法、程序和范例来增强他们软件产品的质量。他们的目标就是使用有效的富有成果的途径来产生有效的问题解决方案。在后续几章中，我们将突出一些支持我们所述的开发维护活动的特别途径。

最新的一套工具和方法的指示器本书相关 WWW 主页上。

软件工程师的位置

为了理解一名软件工程师是怎样适合计算机世界的，让我们举例看一下另一学科。考虑化学研究和它对解决问题的作用。化学家调查化学制剂：结构、相互的作用及在行为后的原

理。药剂工程师把化学家的研究运用于各种问题。化学被化学家视为研究对象。另一方面，对药剂工程师来说，化学是一门工具，被用于解决一般的问题（可能甚至在本质上不是“化学的”）。

我们可同样地看待计算。我们可能关注于计算机和程序语言本身，或者我们视它们为工具被用于设计、实现一个问题解决方案。软件工程是从后者角度来看的，就象如图 1.3 所示。一名软件工程师集中于把计算机作为一个解决问题的工具，而不是调查硬件的设计或证明有关算法如何工作理论。我们会在本章后面看到一名软件工程师把计算机的功能作为一般解决方案的一部分，而非计算机自身的结构或理论。

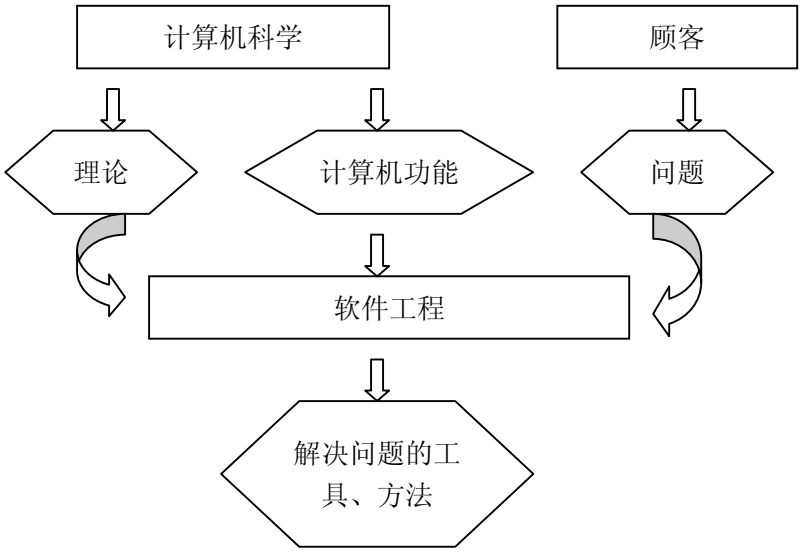

图 1.3 计算机科学和软件工程间的关系

1.2 我们已经取得的成功

写软件既是一门科学也是门艺术，作为一名计算机科学的学生理解这是为什么是很重要的。计算机科学家和软件工程研究员研究计算机机制，和理论化有关怎样使计算机多产有效。然而，他们也设计计算机系统并写程序以便在那些系统上能执行任务，这是一项很有艺术性、独创性、技巧性的实践工作。在一个特定的系统上或许有许多执行任务的方式，但一些比另外的更好。一种方式可能更有效率、更精确、更易修改、更易使用或更易理解。任何黑客能写出使用某物工作的代码，但是要产生健壮的、易于理解和维护、其工作起来尽可能最有效率和效果的代码，是需要有专业软件工程师的技能和理解力的。从而，软件工程将设计和开发高质量的软件。

在调查出完成高质量软件系统要具备的东西前，让我们回顾一下我们已经取得的成功。用户对现存的软件系统满意吗？是也不是。软件使我们能比以前任何时候更快更有效的执行任务。想一下，在，例如字处理软件、电子表格软件、电子邮件、精巧电话，出现前的生活。软件在医学、农业、运输和其他大部分行业支持着生命维系或生命救治方面的进步。另外，软件已经使我们能做我们以前从没做过的事情：显微外科、多媒体教育、机器人技术……。

然而软件不是说没有自身的问题。通常系统运行却不是严密地按我们所期望的那样。我们所有人都听说过有些系统几乎就不能工作的事情。我们都写过不完善的程序：一些仍然包含失误的代码，而在偶然的评定或在只是演示一个方案可行性的时候这样的代码又是够好的。显然，在开发一个交付给用户使用的系统的时候，此种行为是不可接受的。

一类工程的一个错误与一个大的软件系统中的一个错误二者间是有很大差别的。实际上，（人们）在文献里或走廊里频繁讨论软件故障和生产无故障软件面临的困难。一些故障仅仅有点让人讨厌；而别的将花费大量的时间和金钱。还有些甚至威胁生命。补充栏 1.1 解释了故障（fault）、错误（error）和失败（failure）间的关系。让我们看一些失败的例子，注意发生了什么故障及为什么会这样。

补充栏 1.1 描述臭虫（Bug）的术语

通常我们说起软件里“臭虫”这暗示着有一个它所依赖的上下文环境。

一个 Bug 可能是理解需求时的一个误会、一片代码里的语法错误，或者（尚未知的）系统崩溃的原因。为了描绘软件产品中的 Bug，IEEE 已经提出了一套标准术语（IEEE 标准 729，IEEE 1983）。

因人误解时产生一个故障（fault），而它称为错误（error）当在执行某个软件活动而产生时。例如，一个设计者可能误解了一条需求而做了与实际的分析员和用户的需求真实意图不符的设计。这条设计故障便是一个错误的前奏，并且它会导致别的故障，如不正确的代码和用户手册中一条不正确的描述。因此，一个简单的错误会产生许多故障，并且一个故障能出现在任何开发或维护产品的过程中。

失败（failure）是一个对系统要求行为的偏离。它能在系统交付之前或之后（在测试、操作或维护过程中）被发现。因为需求文档可能包含故障，一个失败指示出系统没有按需运行，即使它可能按指定的步骤执行。

因此，从开发者的角度来看，故障是系统内部视图，而失败是外部视图：用户能看到的问题。并非每个故障对应一个失败；例如，如果不完善代码从不被执行或从不进入特别状态，那么故障将从不会引起代码失败。图 1.4 显示了失败的起源。

图 1.4 人为错误（error）怎样引起失败（failure）

在 20 世纪 80 年代，美国国税局（IRS，the United States Internal Revenue Service）雇请 Sperry 公司建立联邦收入税表格自动处理系统。按照华盛顿邮报（Washington Post）的说法，“已经证明系统不足以应付工作量，并且花费了预期的近两倍的费用，不久必须被替换”（Sawyer 1985）。在 1985 年，为了增强原本价值 1 亿零 3 百万美元的 Sperry 设备又追加了 9 千万美元。另外，因为那些问题妨碍了在最后期限里向纳税者返还退款，IRS 被迫支付了 4 千零 2 百万美元的利息及 2 千 2 百 3 十万美元的雇员的加班费（雇员一直努力跟上工作进度）。在 1996 年，形式仍没有改观。洛杉矶时报在三月二十九号报到说对于 IRS 的计算机现代化工作仍然没有任何高明计划，仅仅是一份 6000 页的技术文档。国会议员 Jim Lightfoot 把这项工程称为“因为计划不充分导致的仍在挣扎着的 40 亿美元的大惨败”（Vartabedian 1996）。我们将会在第二章看到为什么项目计划对质量软件生产是重要的。

多年来，公众在日常生活中没有异议地接受软件的灌输。但里根总统提出的“星球大战（SDI）”提升了公众对生产无故障软件系统的认识。流行报纸杂志报到表达出了在计算机科学领域的怀疑。现在，20 年后，当请求美国国会为建立类似系统配发资金的时候，许多计算机科学家和软件工程师仍然相信没有任何保证充分可靠性的编写及测试软件的办法。

例如，许多软件工程师认为一个反弹道导弹系统将至少需要 1 千万行代码；有些估计要达到 1 千万行。比较起来，支持美空间航天飞行的软件至少由三百万行代码组成，包括地面上控制发射和飞行的计算机；在 1985 年，航天飞机（Rensburger 1985）自身有约 10 万行代码。因此一个反导软件系统将需要测试惊人数量的代码。而且，可靠性约束将没法测试。为了弄清为什么，考虑一下临界安全（safety-critical）软件的概念。典型地，我们说某物是临界安全（safety-critical）（例如，某物失败将对生活或健康产生威胁）的那么它至少应当有 10（-9）的可靠度。象我们将在第 9 章看到的，这条术语意味着在 10(9)小时运行中失败不能超过一次。为了观测这种程度的可靠度，我们必须运行系统至少 10(9)小时以验证它没有失败。但是 10(9)小时超过了 114,000 年——作为测试期太太长了。

我们也将在第 9 章看到，如果软件设计和编写不适当，有用的技术也会变得致命。例如，当 Therac-25，一台化疗和 X 射线机，发生故障并杀死了几名病人时，医学界一片震惊。软件设计师没有预料到几个箭头键的非标准使用方式；结果，当想使用低水平剂量时，软件保持了高水平剂量的设置并发出了一束非常集中剂量的辐射。

飞行员杂志（Pilot）发表并在风险论坛（Risks Forum,Pilot 1996）报告了一个不曾预料到的使用和带来的后果的类似而危险的例子。两名苏格兰洛锡安边境区警官正在贝里克郡的荒野用雷达枪识别超速驾车者。突然他们的雷达枪锁定在速度指示超过 300 英里/小时处，数秒后，一架低空飞行的“鹞”式飞机飞来。这架“鹞”式飞机的目标搜索器已经识别出雷达并认为它是“敌人”的；幸运的是，这架“鹞”式飞机是非武装操作的，因为正常的行为是发射一枚自动报复性导弹！

系统不曾预料的使用应该在整个软件设计活动中考虑到。这些使用至少以两种方式处理：运用想象思考系统会怎样被滥用（跟正确使用一样），假定系统将被滥用并设计软件处理这种滥用。我们将在第八章讨论这样的方案。

尽管许多商家致力于无漏洞软件，但实际上大多的软件产品并不是远离故障的。市场力量鼓励开发者迅速交付产品，几乎没有时间彻底测试。典型的，测试小组仅能测试出最可能使用的功能，或最可能会危及、激怒用户的功能。因此，许多用户提防安装第一版代码，知道直到第二版出来前“臭虫”是不能被解决的。此外，对已确知的故障的修改有时是如此的困难以致于使得重写整个系统比修改现存代码要容易。我们将在第十一章研究软件维护的相关问题。

尽管生活中软件取得了一些非凡的成功得到了全面的认可，我们所生产的软件的质量仍然还有很大的改进空间。例如，质量缺乏变得很昂贵；故障不被觉察的时间越长，纠正起来就越昂贵。特别地，据估计，在项目初始分析过程中纠正一个犯下的错误的代价仅是在系统交付给顾客后再纠正类似的一个错误所费代价的十分之一。不幸的是我们没有在早期就抓住大部分的错误。用于测试维护期间发现的错误的纠正费用半数来源于系统生命的早期。在第十二、十三章我们考虑评价开发活动效果、改进尽早捕错过程的方式。

我们将提议的简单但强有力的技术之一就是回顾与检查的运用。许多学生习惯于独自开发、测试软件。但他们的测试效果可能比他们想象的要差。例如，Fagan 研究了过去检测错误的方法。他发现通过给定测试数据并运行程序所检测出的错误仅是系统开发过程中所发现错误的五分之一。然而，同样回顾起来，同事间检查评价彼此的设计和编码，能揭开所有已发现错误的其余五分之四（Fagan，1986）。因此，仅请同事来回顾一下你工作，你的软件质量就会有戏剧性的提高。在后面的章节中，我们将学到更多这样的东西：怎样将回顾和检查

过程运用于每一主要的开发步骤之后以尽可能早地发现、确定错误。我们将在第十三章了解怎样改善检查过程本身。

1.3 什么是好的软件？

正象厂商想办法保证他们生产的产品质量一样，软件工程师也必须找到确保产品具有合意的质量和效用的方法。因此，好的软件工程必须总是包括生产质量软件的策略。但在我们想出一个策略前，必须理解质量软件意味着什么。补充栏 1.2 显示设想是怎样影响“质量”的意思的。在本节，我们研究是什么把好软件和坏软件区分开来的。

补充栏 1.2 展望质量

Garvin（1984）写过有关人们对质量的不同理解。他从五个不同方面描述质量：

- 先验视角，质量是我们能意识但不能定义的东西
- 用户视角，质量是对目标的适切性
- 制造业视角，质量就是规格的一致
- 产品视角，质量依赖于固有产品特性
- 价值视角，质量依赖于消费者为之付款的量

先验视角很象柏拉图（Plato）对理想的描述，或亚里斯多德（Aristotle）形态的观念。换句话说，正象每个实际的桌子是一个理想桌子的近似一样，我们可视软件质量为我们为之奋斗的理想；然而，我们从来就不可能完全实现它。

先验视角是空气般的，相比于更具体的用户视角。当我们衡量产品特性（如缺乏密度和可靠性）以理解产品全面的质量时，我们运用用户视角。

制造业视角在生产中和交付后都考虑质量。特别地，它检测头次产品是否正确生成，避免返工固定了的被交付了的故障。因此，制造视角是一个过程视角，提倡与好过程一致。然而，没有证据表明过程一致会在实际中产生较少故障和失败的产品；过程可能的确会导致高质量产品，但它可制度化了普通产品的生产。我们在第十二章研究一些此种问题。

用户和制造业的视角是从外面考虑产品，而产品视角从内部窥视并评估一件产品的内部特性。此视角是软件衡量制专家所提倡的视角之一；他们假定好的内部质量指标将导致好的外部质量指标，如可靠性和可维护性。然而，需要更多的研究来验证这些假设及确定质量的哪些方面对实际产品用途产生影响。我们可能必须开发出联结产品视角和用户视角的模型。

消费者和市场商人经常采用质量的用户视角。研究人员有时运用产品视角而开发小组运用制造业视角。如果观点上的差别没有弄清，那么混淆和误解可能导致有害的决策与可怜的产品。价值视角能将这些质量全异的情形联结起来。通过把质量和消费者愿意付出的金钱等同起来，我们能看到花费和质量间的交换，并且我们也能管理他们出现时引发的冲突。类似地，购买者将产品花费和潜在利益相比较，把质量看作金钱价值。

Kitchenham 和 Pfleeger（1996）在一份《IEEE 软件》的质量特刊的介绍中研究了此问题的答案。他们提出上下文（环境）有助于确定答案。在字处理软件中所能容忍的错误在临界安全或临界事务系统中就不能接受。因此，我们必须至少以三种方式考虑质量：产品质量，导致产品的过程的质量，处于它将适用的商业环境上下文中的产品的质量。

产品质量

我们可以请人们说出对软件全面的质量作出贡献的特征，但我们很可能会从我们所问的所有人那里得到不同的答案。产生这样的差别是因为特征的重要性依赖于谁在分析软件。如果软件以一种易于理解易于使用的方式完成了使用者想做的事那么使用者认为它是高质量的。然而，有时质量和功能是互相盘绕的；如果某软件难以理解或难以使用，但因它的功能而值得这样，那么它仍然被认为是高质量的。

我们设法测度软件质量，因此我们可能将一个产品和另外一个相比较。为此，我们指定一些对全面质量作出贡献的特别的方面。从而，在测度软件质量时，用户评定诸如失败次数和失败类型这样的外在特征。例如，他们可能把失败确定为次要、主要和灾难性的，并希望发生的任何失败仅是次要的。

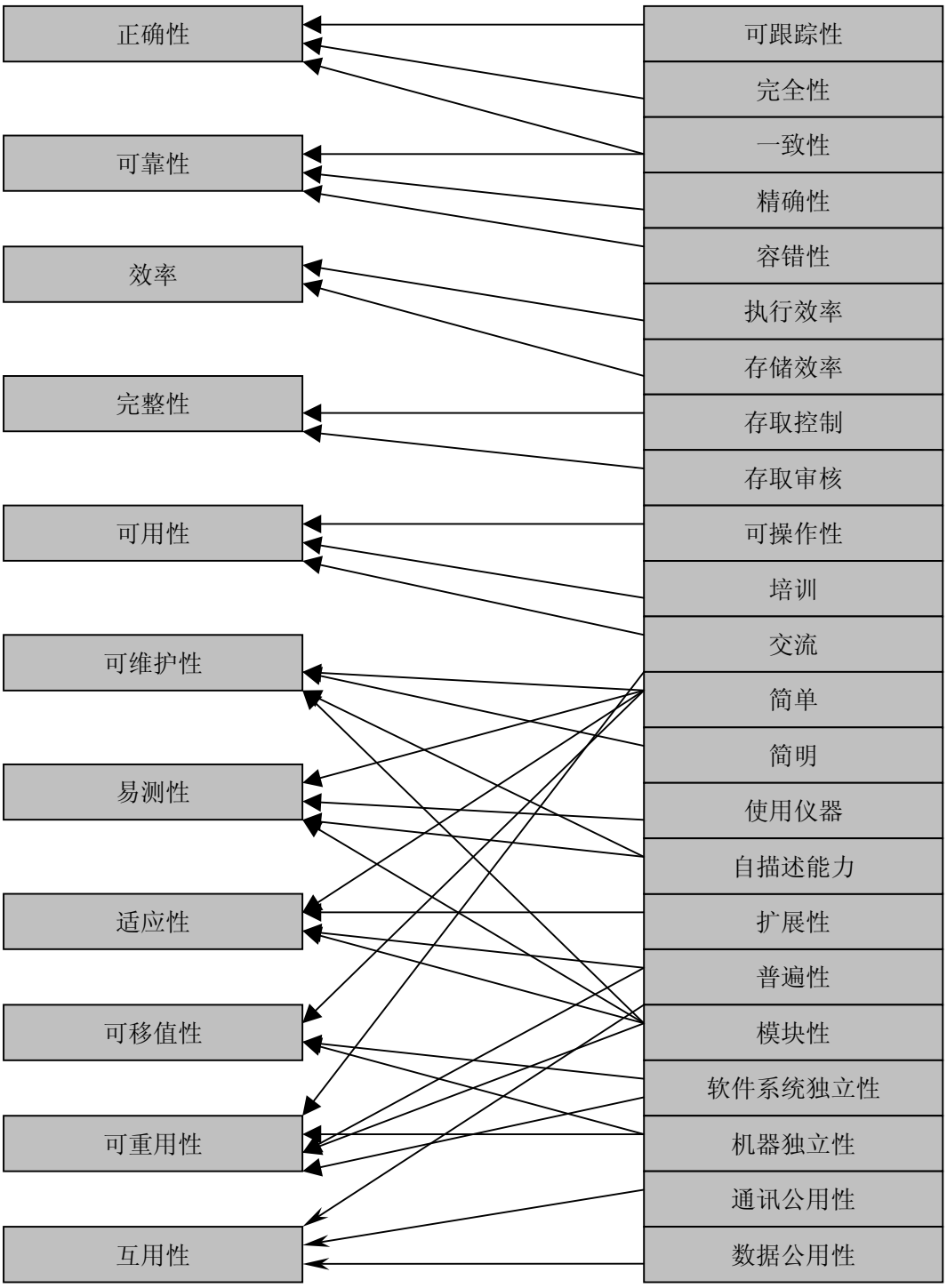

图 1.5 McCall 的质量模型

软件也必须由设计和编写代码的人以及程序完成后负有维护职责的人来评价。这些从业人员倾向于考虑产品的内部特征，有时甚至是在产品交付给用户之前。特别地，从业者考虑故障数量和类型作为产品质量（或缺乏）的论据。例如，开发者跟踪在需要、设计、编码检查中出现的故障的数量并用它们作为最终产品可能的质量的指示器。

因此，我们经常建立将用户外部视角和开发者的内部视角关联起来的模型。图 1.5 是早

期质量模型的一个例子，这个模型由 McCall 和他的同事们建立的，显示了外部质量因素（左侧）是怎样与产品质量标准（右侧）相关的。McCall 将每个右侧标准与一个指示着所表达的质量元素的程度的指标联系起来（McCall，Richards，和 Walters 1977）。我们将在第十二章研究几个产品质量模型。

过程质量

有许多活动影响最终产品质量；如果任何一些活动走偏，产品质量将受到损害。因此，许多软件工程师觉得开发和维护过程的质量跟产品质量同等重要。为该过程建模的优点之一就是我们能检查它并想法子改进它。例如，我们可以问如下的问题：

- 何时何地我们可能发现一个特定种类的故障？
- 在开发过程中怎样较早地发现故障？
- 我们怎样固定故障容忍度以便我们最小化一个故障成为失败的可能性？
- 有替代活动使得过程更有效（率）确保质量？

这些问题可适用于整个开发过程或子过程，象配置管理、重用或测试；我们将在以后章节研究这些过程。

在 20 世纪 90 年代，在软件工程中过程建模和过程改进是极力宣扬的焦点。受 Deming 和 Juran 的工作所启示，由 IBM 之类的公司实现，过程指导方针——如能力成熟度模型（CMM），ISO 9000 和软件过程改进和能力检测（SPICE）——建议通过改进软件开发过程，我们就能改进因而产生的产品的质量。在第二章，我们将了解怎样确定相关过程活动及如何将他们对中间和最终产品的影响建立模型。第十二、十三章将深入地检查过程模型和改良结构。

商业环境上下文中的质量

当质量评估的焦点放在产品和过程上时，我们常常用涉及故障、失败和时间选择的数学表达式来度量质量。而范围扩展到包括商业观点的情况是很少的，而在商业方面，是通过软件所内嵌的商业环境提供的产品和服务来衡量质量的。就是说，我们考虑的是产品的技术价值而不是更广泛地考虑它们的商业价值，并且我们仅仅是基于生产产品的技术质量来做决策的。换言之，我们假定改进技术质量将自动转化为商业价值。

几位研究人员已经对商业价值和技术价值间的关系作了紧密考虑。例如，Simmons 已经会见了许多澳大利亚商家以确定他们是如何做有关信息技术的商业决策的。她提出了一个理解“商业价值”对公司的含义的构架。在 Favaro 和 Pfleeger（1997）的一份报告中，Steve Andriole，Cigna 公司（一家大型美国保险公司）的首席信息官，描绘了他的公司是怎样区别技术价值和商业价值的：

我们衡量（我们的软件）质量是通过显然的衡量尺度的：根据时间先后，维护费用、修改费用等。换言之，我们基于代价参数范围里的操作性能来管理开发。我们对卖家如何提供划算的性能的关心要少于对这种努力所取得的结果的关心……商业和技术价值问题在我们内心也是亲密和宝贵的……并也是一个我们在上面集中了大量注意力的问题。我猜我惊讶地了解到公司愿意为了别的公司的技术价值而以相对的商业价值为代价与他们缔约。If anything,我们在另一方面犯了错！如果没有清晰（可预料）的商业价值（在数量表示为：被处理的主张数，等等）那么我们不能发起这项工程。我们非常郑重地工程的“有目的的”需要阶段，那时，我们问：“我们为什么需要这个系统？”及“我们为什么在意？”

已经有几次想以定量和有意义的方式将技术价值和商业价值关联起来的尝试。例如，Humphrey，Snyder，和 Willis（1991）指出，通过按照 CMM“成熟”规模（将在第十二章

讨论）改进开发过程，Hughes Aircraft 将生产率提高了 4 倍，节省了几百万美元。类似地，Dion（1993）报告说，在过程改进后，Raytheon 生产率加倍，且每投资一美元就得到 7.7 美元的回报。而位于俄克拉荷马州的 Tinker 空军基地的技术人员说生产率有 6.35 倍的改进（Lipke 和 Butler 1992）。

然而，Brodman 和 Johnson（1995）对过程改进的商业价值做了更紧密的思考。他们调查了 33 个进行了某种过程改进活动的公司，并调查出了几个关键的问题。在另外的一些事情中，Brodman 和 Johnson 问公司他们怎样定义投资回报率（ROI），一个商界清楚定义了的概念。他们指出，投资回报率的书面定义衍生于金融界，描述了根据为达到别的目的所进行的投资。也就是说，“投资必须不仅要返还原来资本而且要更多，至少要等于这笔资金投资于别的方面所赚利润加上一笔风险金（Putnam 和 Myers 1992）。通常，商界使用三个模型之一来评估 ROI：偿还模型，会计学返回率模型和折扣兑现流模型。

然而，Brodman 和 Johnson（1995）发现美国政府和美国工业界以很不相同的方式解释 ROI，彼此不同，且二者也不同于标准商校的方法。政府以美元来看 ROI，考虑减少动作费、预测美元储蓄及计算采用新技术的费用。政府的投资也以美元来表示，如引进新技术或发起过程改进的费用。

另一方面，工业界以业绩来看待投资而不是花费或美元。也就是说，公司感兴趣的是节省时间或使用更少的人力，并且他们对有关投资回报的定义反映在临死业绩这样的焦点上。在被调查的公司中，投资回报包括这样的项目：

- 培训
- 进度表
- 风险
- 质量
- 生产率
- 过程
- 消费者
- 花费
- 商业

包括在定义中的花费问题涉及满足预测、改善花费效能及不超出预算，而不是减少运作费用或使工程或组织简化并更有效率。图 1.6 概括了许多组织在他们的 ROI 的定义中包括一条投资项的频率。例如，会见公司中约%5 在 ROI 业绩计算中包括了一个质量组的业绩，约%35 在考虑投资费用时包括软件费用。

观点间的差异是烦扰的，因为它意味着组织间 ROI 的计算是无法比较的。但是这此不同的观点的存在却有很好理由。来自缩减的进度、更高质量和增加的生产率中的美元节省被返回政府而不是订约人。另一方面，订约人经常期待竞争优势和增强的工作能力和更大的利润；因此，订约人的 ROI 是更有业绩——比起基于花费。特别地，更精确的费用和进度估计可能意味着消费者满意及不断的生意。并且，对于市场来说，缩减的时间跟改进了的产品质量一样也被理解为提供商业价值。

即使不同的 ROI 计算可对每个组织是合理的，也有人担心软件技术投资回报与金融上的 ROI 不同。从某种观点来看，程序成功必须向更高层次的管理汇报，很多与软件无关而是与公司的主要业务有关，例如通讯或银行。有很大内在意思上差异的相似术语的使用将引起很多混淆。因此，我们的成功标准必须不仅对软件工程和过程有意义，而且也应对软件所支持的更一般商业实践有意义。我们将在十二章的更多细节中调查这个问题并考虑使用几个通用商业价值指标来在技术选项中进行选择。

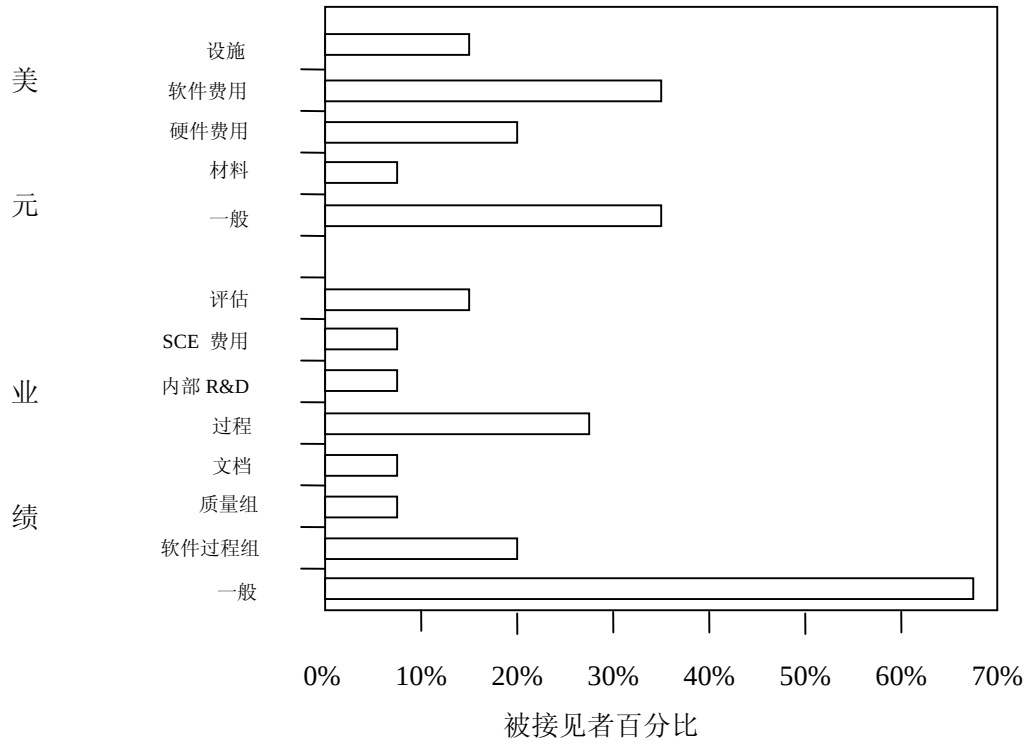

图 1.6 工业投资回报率中的术语

1.4 谁实施软件工程？

软件开发的一个关键部分就是顾客和开发者之间的交流；如果失败，系统也将失败。在建立一个系统来帮助顾客解决问题之前，我们必须理解顾客想要和需要的东西。为了做好这一步，让我们把目光转向软件开发过程中涉及的人。

从事软件开发的人数依赖于工程规模和困难程度。然而，无论涉及多少人，在整个工程的生命中扮演的角色是有区别的。因此，对一个大型工程，一个人或一个组可能会被安排为确定的每个角色；在一个小工程中，一个人或一个组可能一次担当几个角色。

通常，一个项目的参与者属于三类：顾客，用户，或开发者。顾客就是为软件系统开发支付资金的公司、组织或个人。开发者就是为顾客开发软件系统的公司、组织或个人。这类人包括任何协调、指导程序员和测试人员所需要的管理人员。用户就实际使用这个系统的人员：坐在终端前或提交数据或读输出。尽管对一些工程顾客、用户和开发者是同一个人或同一个组，但通常他们是不同集合中的人。图 1.7 显示了三类人间的基本关系。

顾客，控制着资金，通常谈判合同和签署接受文件。然而，有时顾客不是一名用户。例如，假定 Wittenberg Water Works 与 Gentle Systems 公司签署为前者建立计算机化记帐系统的合同。Wittenberg 的总裁向 Gentle Systems 的代表正确地描述要求，并且签署合约。然而，这位总裁将不直接使用这套记帐系统；用户将是簿记员和财务工作人员。因此开发者正确地理解顾客和用户两者想要的和需要的东西是很重要的。

另一方面，假定 Wittenberg Water Works 是如此庞大的以致于有自己的计算机系统开发部门。这个部门可能决定需要有一个自动工具来跟踪它自己的工程的花费和进度。在自身建

立工具的时候，这个部门同时是用户、顾客和开发者。

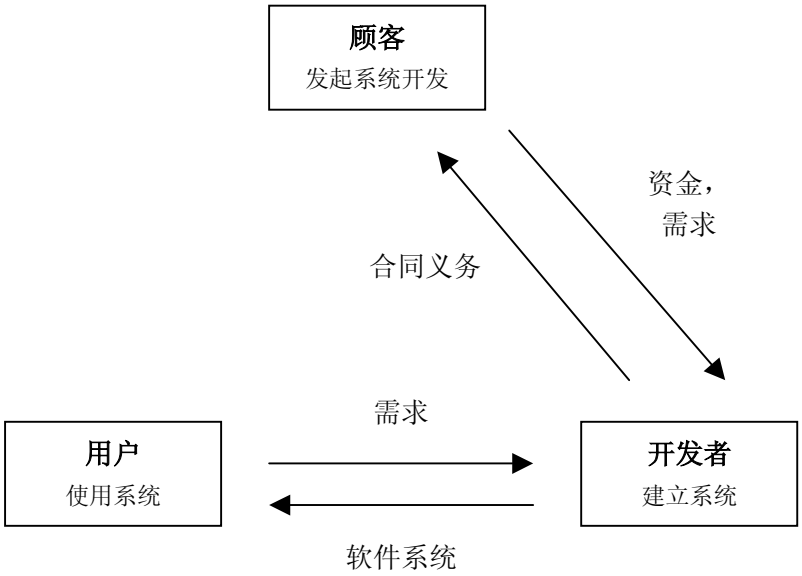

图 1.7 软件开发的参与者

在最近几年，顾客、用户和开发者间简单的区别已经变得更加复杂。顾客和用户已经以各种方式涉入开发过程中。顾客可能决定购买 Commercial Off-The-Shelf（COTS）软件以并入开发者将供应和支持的最终产品中。当这样的事情发生时，顾客涉及了系统体系决策，并且开发工作也有了更多的限制。类似地，开发者可以选择使用别的被称为次承包商的开发者，次承包商建立子系统并将它交付给开发者以纳入最终的产品中。次承包商可能与主开发者并肩工作，或者在不同的场所工作，同时保持他们的工作与主开发者协调一致并在开发过程后期交付子系统。子系统可能是一个一切齐全可立即使用的系统（turnkey system），在这个系统中代码被合并成一个整体（没有额外的代码以求完整）；或者子系统需要一个单独的完整性处理以建立主系统到子系统间的联接。

因此，“系统”的概念在软件工程是很重要的，我们装配起来的硬件和软件必须与用户交互，与别的软件作业交互，与别的硬件交互，与现存数据库（例如，数据库拥有仔细确定的数据集和数据关系），甚至与别的计算机系统。所以，对任何工程，了解这个工程的边界以为其提供一个上下文（环境）是很重要的：什么需要包括在这个工程中，什么不需要。例如，假设上司要求你写一个替你办公室里人员打印付薪水的支票的程序。你必须要知道程序是否只是简单地从别的另外的系统读工作小时数并打印结果，或者是否也还要计算支付信息。类似的，你必须知道是否由程序来计算税金、养老金和退休金，或者是否这些项目的一份报告将附随于每一张付薪支票。你真正问的是：这个工程开始哪里又在哪里结束？同样的问题适用于任何系统。一个系统是一个对象和活动的集合再加把对象和活动联系在一起的关系的一个描述。典型地，我们的系统定义中，对每个活动，包括所要求的输入、所要采取的行为、和所产生的输出的一个列表。因此，为了开始，我们必须知道是否任何对象或活动已经被包含于系统中。

系统元素

我们通过为系统各部分命名然后确定组成部分和另外组成部分如何关联来描述它。识别是分析摆在我们面前的问题的第一步。

活动和对象

首先，我们区分一下活动和对象。一个活动（Activity）就是发生于系统中的某事。活动通常被描述为一件由触发器发起的事件，它通过改变一个特征来将一件事物转换为另一事物。这种转换可能意味着一个数据元素从一个位置移到另一个位置，从一个值变为另一个值，或者与另的数据合起来为另外的活动提交输入。例如，一个数据项可能从一个文件移到另一个文件。在这种情况下，发生变化了的特征就是位置。或者这个数据项的值被增加。最后，这个数据项的地址可能和几个另外的数据项的地址被包括在参数列表中以便别的例行程序立即被调用来处理所有这些数据。

活动中涉及的元素被称为对象（object）或实体（entity）。通过，这些对象以某种方式相互关联。举例，对象可被排列在一张表或矩阵中。通常，对象聚合成记录，每个记录以一种指定的格式排列。例如，一条雇员的历史记录（称为域）可能包括着描述每位雇员的对象，如下：

First name	Postal code
Middle name	Salary per hour
Last name	Benefits per hour
Street address	Vacation hours accrued
City	Sick leave accrued
State	

在记录中，不仅定义了每个域，而且也指明了每个域的尺寸及与别的域间的关系。因此，记录描述规定了每个域的数据类型、在记录中的开始位置和域的长度。依次地，因为有了每个雇员的记录，记录可被合并成一个文件，并且可以指定这个文件的特征（例如最大记录数）。

有时，对象的定义会稍有不同。在一条更大的记录中，对象被视为是独立的，而不考虑作为一个域的每一项。对象描述包括了每个对象的特征列表和使用这个对象或影响这个对象而发生的所有的活动的列表。例如，考虑“多边形对象（polygon）”。对象的描述可能说诸如边数和每边边长这样的特征。行为可能包括面积或周长的计算。甚至可能有一个称为“多边形类型（polygon type）”的特征，以便“polygon”的每个实例能被识别出何时它是，举例来说，“菱形（rhombus）”或“矩形（rectangle）”。 类型自身可能有一个对象描述；例如，“rectangle”可以由“正方形（square）”和“非正方形(not square)”组成。我们将在第四章研究需求分析时探究这些概念，并在第六章讨论面向对象开发时深入探究。

关系和系统边界

一旦实体和活动确定下来，我们将实体和它们的活动匹配。实体和活动间的关系是被清晰而仔细定义。一个实体定义包括实体来源的描述。一些项在已存在的文件中；另外的在活动期间建立。实体的目的地也是重要的。一些项仅由一项活动使用，但另一些被预定地输入别的系统。也就是说，一些来自一个系统的项被处于受检查的本系统之外范围时的活动使用。因此，我们认为我们所见的系统是有边界或分界线的。一些项跨过边界进入我们的系统而别的项是我们的系统的产品并跑出去为另外系统使用。

运用这些概念，我们可在将系统（System）定义为如下事物的集合(collection)：一个实体集、一个活动集、一个实体与活动的联系的描述，和一个系统边界的定义。这条系统的定义不仅适用于计算机系统而且也适用于在其中对象以某种方式与别的对象交互的任何事物。

1.5 一个系统的方法

为了弄明白系统定义如何工作，考虑让你身体可以吸收氧气排出二氧化碳和水的那个部

分：呼吸系统。你可以很容易地定义它的边界：如果你叫出你身体的一个特别器官，你能说出它是否是呼吸系统的一部分。氧气和二氧化碳分子以明确定义了的方式通过系统的对象或实体。我们也能根据实体的交互描述系统中的活动。如果需要，我们能通过显示进入和离开它的东西解释系统；我们也能提供表格来描述所有的实体和他们所涉及的活动。图 1.8 解释了呼吸系统。注意到每个活动都涉及实体，并且能通过描述哪个实体充当输入、它们怎样被处理和产生了什么结果（输出）来定义。

图 1.8 呼吸系统（略）

我们也必须清晰地描述我们的计算机系统。我们和预期的用户来定义系统的边界：我们的工作开始和终止于哪里？另外，我们需要知道什么是处于系统边界上并因此确定输入的来来源和输出的目标。举例，在一个打印付薪支票的系统中，支付信息可来自公司计算机。系统输出可能是一组将送往邮箱以交付给适当收件人的付薪支票。在图 1.9 所示的系统中，我们能看到这样的边界并能理解实体、活动和他们的关系。

相关系统

边界概念是重要的，因为很少有系统是独立于其他系统的。例如，呼吸系统必须与消化系统，循环系统、神经系统及其他系统交互。没有神经系统呼吸系统不能工作；没有呼吸系统循环系统也不能工作。相互依赖可能是复杂。（的确，许多环境问题出现并加剧就是因为我们没有认识到生态系统的复杂性。）然而，一旦系统边界得到描述，我们就很容易明白什么在里面什么不在里面、什么穿越了边界。

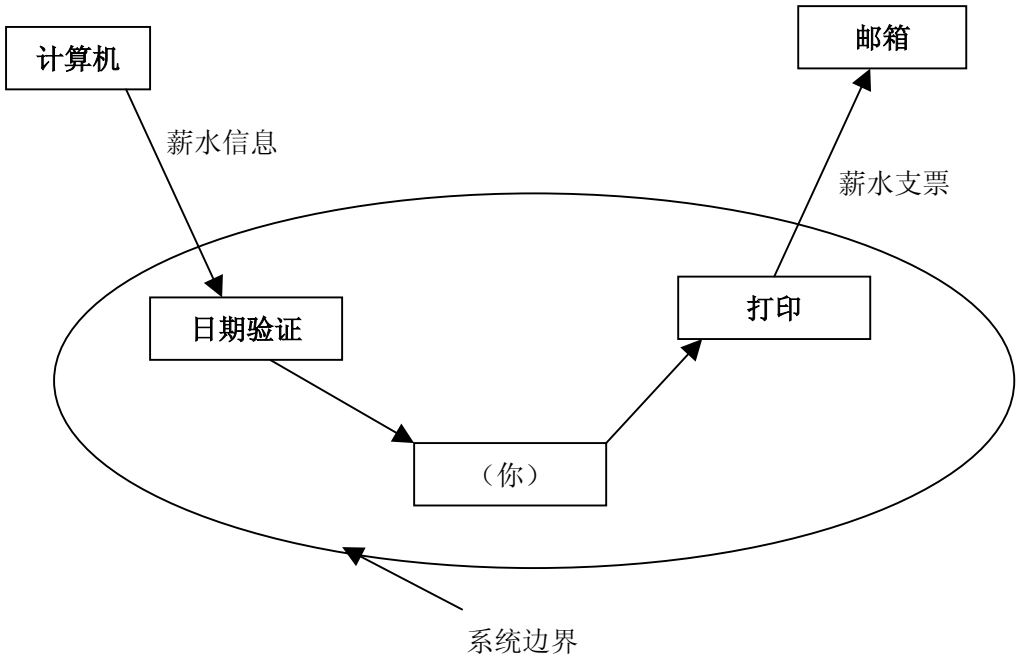

图 1.9 付薪支票产生系统的定义

依次地，一个系统存在于另一个系统内部也是可能的。当我们描述一个计算机系统时，我们常常把焦点集中于实际上更大的一个系统的一小片上。如此的焦点可让我们定义和建立一个复杂性相对要小一点的系统包含于大系统内部。如果我们仔细地文档化影响我们系统的

那些系统间的交互，那么当我们把专注于一个较大系统的这个小片时就不会丢失什么东西。

让我们看一个怎样进行这样的工作的例子。假定我们正开发一个水文监视系统，它的数据来源于整个流域的许多观测点。在收集处进行几样计算，而结果被传到一个中央位置以得到完整的报告。这样的一个系统可以通过中央位置的一台计算机和远端的几十台较小的计算机通信来实现。许多系统活动必须考虑到，包括数据收集的方式、远端计算机执行的计算、和中央位置的信息通讯、数据库或共享文件中通讯数据的存储、从这些数据产生报告。我们可以把这个系统当作一个系统集，每一个系统完成一个特定的目标。特别地，我们可只考虑大系统的通讯方面并开发一个通讯系统来从一批远端向中央位置传输数据。如果我们小心的定义通讯和大系统间的边界，通讯系统的设计和开发工作能独立于大系统进行。

整个水文监视系统要比通讯系统复杂得多，因此我们分别的小片的处理使得我们的工作简化不少。如果边界定义详细且正确，从较小系统建立大系统就相对容易。我们能通过分层考虑大系统来描述建立的过程，就象图 1.10 所描述的水文监视的例子。一个层本身就是一个系统，但每层和它所包含的也形成一个系统。图中的圈代表了各个系统的边界，整个圈集合并成整个水文监视系统。

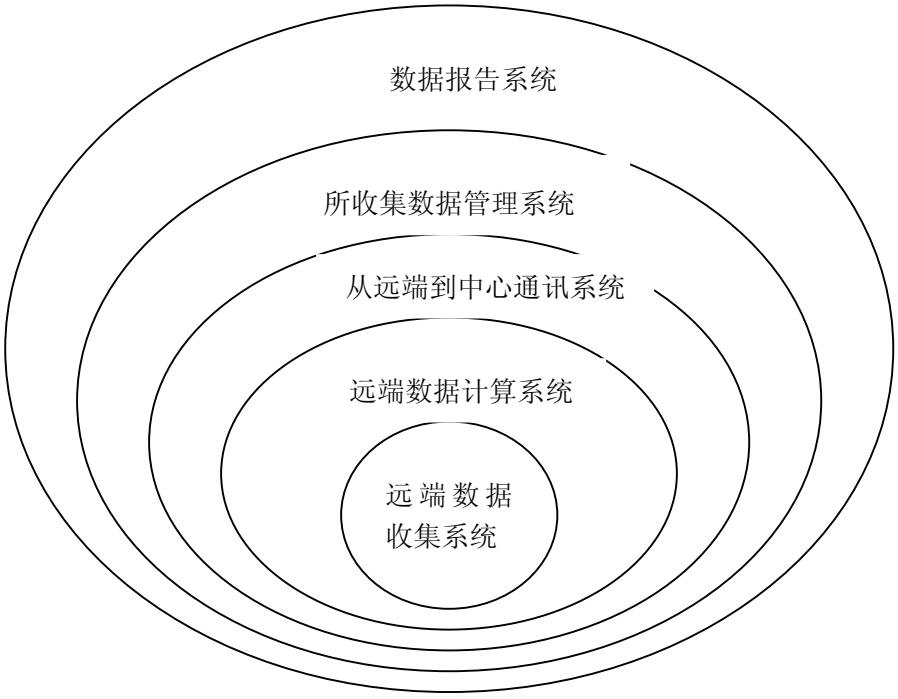

图 1.10 水文监视系统的层次

认识到一个系统容纳另一个系统是重要的，因为它反映这样的事实：在一个系统中的对象或活动是外层代表的每个系统的一部分。因为每层都引入了更多的复杂性，在每个包含更多圈的系统中理解任何一个对象或活动都变得更困难。因此，我们通过首先尽可能关注最小的系统来达到最佳的简单度和随之而来的对系统的最好的理解。

我们用这样的观念来建立取代较老的系统的系统，或者手工或者自动的。我们需要尽可能理解新老系统是如何工作的。通常，两个系统间的差别越大，设计和开发就越困难。这种困难不仅在于人们倾向于抵制变化，而且因为差别使得理解困难。在建立或综合重要系统中，一个递增系列中间系统对构造新系统是有戏剧性帮助的。我们可能从系统 A 到 A’到 A”再到 B 而不是直接 A 到 B。例如，假如 A 是一个手工系统，它包含了三个主要功能，而 B 是一个 A 的自动系统版本。我们可以定义系统 A’是一个功能 1 自动而功能 2 和 3 仍然手动的新

系统。然后 A”有自动的功能 1 和 2，而 3 仍是手工的。最后，B 有三个自动功能。通过将从 A 到 B 的“距离”分割，我们得到一系列比整体更易处理的小问题。

在我们的例子中，两个系统是非常相似的；功能相同，但他们实现的风格不同。然而，目标系统与现存系统常有很大不同。特别地，通常目标系统没有现存软硬件强加的限制。一个增量开发方法可能合并一系列阶段，每个阶段在前一个阶段的基础上摆脱了另外的一个限制。例如，阶段 1 可能增加了一块新硬件，阶段 2 可能替换了一套执行一个特定功能的软件，等等。系统从旧的软硬件中被慢慢拉走直到它反映了新的系统设计。

因此，系统开发首先可被合并成一个对实际系统的改变集合，然后加入一系列变化以产生一个完整的设计计划，而不是一下子从当前跳到将来的系统。用如此方法，我们必须同时以两种不同的方式来看待系统：静态地和动态地。静态视图现在的系统正如何工作，而动态视图向我们指出系统正怎样变成它最终的样子。一个视图没有别一个是不完整的。

1.6 一个工程方法

一旦理解了系统的本质，我们就准备开始构建。在这点上，软件工程的“工程”部分变得贴切并补足迄今为止我们已做的事情。回顾一下，我们是以“承认写软件是一门科学也是一门艺术”开始本章的。生产系统的艺术涉及软件产品的工艺。作为艺术家，我们利用已经被证明对生产有益的，高质量的产品有帮助的技术和工具。例如，我们可能使用一个优化编译器作一个工具来生成在我们使用的机器上运行得很快的程序。或者我们可能包括特别种类或搜索例程作为在我们的系统中省时省空间的技术。这些基于软件的技术被使用正象技术和工具用在精心制作优质家具或建房中。的确，一套受欢迎的的编程工具集被称为程序员工作台，因为程序员依赖它们就象一位木匠依赖于一个工作台。

因为建立一个系统类似于建造一幢房屋，所以我们能指望房屋建造作为别的为什么“艺术的”方法对软件开发重要的例子。

建造一幢房屋

假定 Chuck 和 Betsy Howell 雇用了某人为他们建一座房屋。因为它的尺寸和复杂性，一幢房屋常常需要不止一个人在建设小组里；因此，Howell 家雇用了 McMullen 建设公司。房屋建设涉及到的第一件事就是 Howell 家和 McMullen 间召开一次会议以便 Howell 家能解释他们想要什么。这次会议探究的不仅是 Howell 家想这房屋看起来的样子，而且也探究它应包括的特征。然后，McMullen 草拟房屋的地基计划和建筑透视图。在 Howell 家和 McMullen 讨论了细节后，变化开始了。一旦 Howell 家认可了 McMullen，建设就开始了。

在建设过程中，Howells 很可能视查建设场所，考虑他们喜欢的变化。在建设过程中几个那样的变化可能产生了，但最终房子完工了。在建设过程中和 Howell 家迁入之前，会检测房子的几个组成部分。例如，电工检测电线，管道工确保管道不会漏水，木匠调正木料以便地面光滑水平。最后，Howell 家搬进去。如果有某物没有正确的建好，McMullen 可能被召来修整，但最终，Howell 家对这房屋负全部责任。

让我们更紧密地考虑这个过程中涉及的东西。首先，因为许多人同时参与这幢房子的建设工作，文档是重要的。不仅是地基计划和建设草图是必需的，而且也必须写下细节以便象管子工和电工之类的技术人员在这幢房子成为一个整体的时把它们的产品组合在一起。

其次，在这个过程开始时就期待 Howell 家描述他们的房子及直到房子完工才走开都是不合情理的。的确，Howell 家可能会在建设过程中几次修改房屋的设计。这些修改可能产生于许多条件：

- 起初指定的材料不再有用。例如，某些类型的顶瓦可能不再生产了。
- 在 Howell 家看到房屋成形时可能有了新想法。例如，可能意识到额外花一小笔费用就可为厨房安上天窗。
- 可用性或资金限制可能要求 Howell 家改动需求以满足进度或预算。例如，Howell 家想订购的特定窗户在冬天房屋完成前将不能及时备好，因此库存窗户被用来替代。
- 当初想好的项目或设计结果可能证明是不可施行的。例如，土壤浸透测试可能揭示房屋周围的土地不能支持 Howells 家所要求的浴室数。

McMullen 也可能在建设已经开始后提议一些改动，或许因为一个更好的主意或者因为建设队伍的一名关键成员不能利用。McMullen 和 Howell 家可能对房屋的一个特征改变了想法，甚至是在这个特征已经完成之后。

第三，McMullen 必须提供蓝图，配线和配管道图，器具使用手册，和别的文档以便 Howells 迁入后能作修改或维修工作。

我们可能将建设过程概括为以下几步：

- 确定、分析需要
- 产生及归档房屋的全面设计
- 生成房屋的详细说明
- 标识、设计组成部分
- 建立房屋的各组成部分
- 测试房屋的各组成部分
- 结合和组成部分、在住户已经迁入后做最后的修改
- 由房屋居住者继续维护工作

我们已经看到参与者必须如何保持灵活性并允许建设过程中在原来的规格说明中的多个不同点上做更改。

房屋是在它所处于的社会、经济和政府结构的上下文环境中建设的，记住这点很重要。正象图 1.10 的水文监测系统描绘了子系统的依赖，我们必须把房屋当作一个处于更大配置中的一个子系统。例如，房屋建设是在城市或乡村的建设法规或规章的上下文环境中进行的。McMullen 雇员要由城市或乡村许可，并且他们被指望按照建设标准行事。建设检查员巡视建设场所以确保建设标准得到遵守。并且建设检查员设定质量标准，这样检查成为这项工程质量保证检查点（checkpoint）。也可能有社会或惯常的限制提出了公共的或可接受的行为；例如，让前门直接对着厨房或卧室开放违反常情的。

同时，我们必须认识到我们不能正确的规定建房中的活动；我们必须为基于经验所作的决定、为处理料想不到或不标准条件而留下余地。例如，许多房屋从先存在的部件塑造而来；门已经被提供在框架里了；浴室使用预制淋浴隔间等等。但标准建屋过程可能不得不更改以适应一个不平常的的特征或要求。假定架子搭好了、干燥墙建好，底层地板被铺好，接着的步骤便是在浴室地板上铺瓷砖。建立者发现在，非常令他们沮丧的是，墙和地面都不恰是正方形的。这个问题可能不是过程欠缺的结果；房子建自于有一些自然的和制造方面变异的部分，因此.不精密的问题是会发生的。地板砖，由小正方形组成，如果以标准方式铺设将突出这种不精密。在这里，艺术和专门技术发挥了作用。修建者很可能从衬背里去掉瓷砖，每次铺一板，彼此间作小的调整以便全部的变动是对所有人（除了最有辨识能力的眼睛）觉察不到的。

因此，房屋建设是一个复杂的任务，在此期间，过程、产品或资源中有许多改变的时机，这些都通过适量的艺术和专门技术来调节。建房过程能被标准化，但总是需要内行的判断和创造力。

建立系统

软件项目也以相似于建房的过程发展。在我们的例子中，Howells 是顾客和用户，而 McMullen 是开发者。如果 Howells 请 McMullen 建房给 Howell 先生的父母居住，那么用户、顾客和开发者将是有区别的。同样，软件开发涉及用户、顾客和开发者。如果我们被邀请为一个顾客开发一个软件系统，第一步就是和顾客召开会议以确定需求。需求描绘了系统，象我们在前面看到的一样。不知道边界、实体和活动，描述软件及它将怎样与环境交互是不可能的。

一旦确定了需求，我们将建立一个满足指定需求系统。就像将在第五章看到的一样，系统设计反映了从顾客的角度看起来系统的样子，因此，正象 Howell 家看到楼层计划和建筑草图，我们向顾客提供将采用的录象显示屏图片，生成的报告，以及任何别的用来解释用户如何与完工后系统交互的描述。如果系统有帮助手册或优先的过程，也应得到描述。开始，Howells 只对房子外观和功能感兴趣，晚一些他们必须决定诸如使用铜质还是塑料管子这样的事项。同样，软件项目的系统设计阶段仅描述外形和功能。

然后顾客复查设计。得到认可后，整个系统设计用于产生要涉及的单个程序的设计，注意这一步是在提到程序的才开始进行。除非外形和功能确定了，否则考虑编码通常没有任何意义，在房屋的例子中，我们现在将准备讨论管子类型和电线质量了。因为我们已知道在此结构中水需要流向的地方了，所以可以决定使用铜墙铁壁质还是塑料管子。同样，系统设计得到各方认可后，我们准备讨论程序，我们讨信纸的基础就是：一份很好定义了的这个作为一个系统的软件项目的描述；系统设计包括一份对涉及的功能和交互活动的完整描述。

程序写好后，在能被联接到一起前，他们被当作单纯代码来测试。测试的第一阶段我称为模块式单元测试，一旦确信这些代码工作满意，我们便把他们放于一起，并确保他们与别的代码联接起来后也能正确工作。这便是第二测试阶段，被称之为集成 C 或集成 B 测试，就像建系统时将一块片断补充到下一个片断之上。一直到整个系统是可操作的。最后测试阶段称为系统测试涉及的是整个系统的测试以确保当指定的功能和交互都被正确实现，在这个阶段，系统将和指定需求进行比较；开发者，用户核查系统是否达到了预期目标。

最后，最终产品被交付。在使用过程中，矛盾和问题揭示出来。如果我们开发的一个“交钥匙型”系统，那么交付后顾客自己对系统负责。然而，许多系统不是那样，如：如果哪里出错，或需求变化了。开发者或别的组织提供维护。

因此，软件开发包含下列活动：

需求分析和定义
交流设计
程序设计
写程序（程序实现）
单元测试
整体测试
系统交付
维护

理想条件下一次进行一项活动，当到达列表尾部时，人已经完成了一个软件项目。而实际上，其中许多步将被重复。例如，在得查系统设计时，你和顾客发可能发现某些需求还没有写入文档。人与顾客共同工作补充需求并可能重新设计系统，类似地，在编写和测试代码

时，你可能发现一个设备并不像文档中描述的那样发挥作用。人可能不得不重新设计代码，重新考虑系统设计。或甚至回头与顾客讨论怎样满足需求。因而。我们这样来定义软件开发过程（goftware development precess）：是任何包含了上面列下来的被组织起来的产生测试后代码的凡个活动的软件开发的描述。在第二章我们将讨论建立软件时使用的几种不同的开发过程。后面章节就将考查每个子过程和他们的活动——从需求分析到维护。但在此之前，让我们看一下谁开发软件及过去的时间里的软件开发的挑战是如何变化的。

1.7 开发组的成员

在本章较前部分，我们看到顾客、用户和开发者在与新产品的定义和建立中扮演着主要角色。开发者是软件工程师，但每位工程师可能专攻于开发的一个特定方面。让我们更详细地考虑一下开发组中成员的角色。

许多开发过程的第一步就是弄清楚顾客想要什么并文档需求。我们已经看到，分析是一个将事物分成组成部分以便于我们能更好理解的过程。因此，开发组包括一名或多名需求分析员，他（们）与顾客一起工作，把顾客想要的东西分解成离散的需求。

一是需求已知并被文档化，分析员与设计员一起工作以产生一个“系统将要做什么”的系统级描述。依次地设计员与程序员一起以一种程序员能写下实现指定需求的代码行的方式描述系统。

生成的代码必须被测试。通常由程序员自己做第一步测试；有时额外的测试员也被用来帮助程序员捕捉程序员忽视了的差错。当代码单元被集成到功能组时，一组测试真与实现小组一起在通过合并代码片建立系统时验证系统正确并与规格一致地工作。

当开发组对系统的功能和质量满意后，注意力便转向顾客。测试组和顾客一起工作以验证整个系统是顾客想要的东西；他们通过拿系统和当初的需求集进行比较一进行这项工作。然后，培训员向用户演示怎样使用系统。

对很多软件系统，顾客的认可并不意味着开发者工作的结束。如果系统被交付后发现了故障，一个维护组一维护 它。此外，随着时间流逝，顾客的需求可能发生变动，相应的系统也会改动。因此，维护可能涉及分析员——他决定要增加可改动什么需求，设计员——决定系统设计中哪里应改动、程序员——实现改动测试员——确保改动后的系统仍能正确运行，和培训员——向用户解释是怎样影响系统的使用的。图 1.11 解释了开发组各种角色是怎样与开发步骤对应的。

学生常独自或与小组组成项目的开发组。导师要求的文档是最少的，学生通常不需写用户手册或培训文档。而且，通常相对稳定；整个项目生命期是需求不发生变化。然后，学生建立的系统很可能在课程结束时被遗弃；他们的目的是演示能力面，不是替真正顾客解决问题。因此，对课堂项目占程序大小，系统复杂性，对文档的要求，对可维护性的要求相对要小。

图 1.11 开发组角色

然而，对一个真实的顾客，系统的大小和复杂性都很大，也极要求文档和可维护性，对一个泼汲 4 千行代码和更多的开发组成员间的交互活动的项目来说，项目各方面的控制可能是很困难的，为了支持开发组中的每个人，有几个人可能在开发一开始就参与其中与系统打交道，并在过程中一直这样，

图书管理员准备并保管整个系统生存期要用到的文档，包括需求规格说明书，设计描述，

程序文档，培训手册，测试数据，进度等等，与图书馆管理员一道工作的是一个配置管理组。配置管理涉及到维护需求设计实现，测试间的一阶，这种交叉参考可在需求中某处变动时哪些程序需要更改，或者提议的某类改动会影响程序中哪些程序中哪些部分配置管理工作人员也协调已建立成支持的系统的不同版本。例如，一个软件系统可能支持不同的平台，或者可能交付了一系列的发行版。配置管理确保从一个平台到另一平台上功能一致。不随系统发行版而恶化。

开发角色可由一个或几个人来承担，对小项目两三个人可以共同担负所有的角色，然而对较大项目开发组基于开发中的功能常分为不同的组。有时，维护系统的哪些人不同于当初设计或写这个系统的哪些人。对一个非常庞大的项目，顾客甚至雇用一个公司最初的开发而雇另一个公司从事维护。在后面章节中讨论开发和维护活动等。

我们将看一看每种类型的一切角色需要什么技巧。

1.8 软件工程如何变化的

我们已经把建立软件比作建房。第三年全国有数百的房屋建起来，满意的顾客搬进去每年开发者也建立起灵敏百的软件产品，但顾客常常对结果不满意。为什么会有这样的差异？如果枚举系统开发的步骤是如此容易的，为什么作为软件工程师的我们会花如此艰难的时间来建立生产质量软件？

回顾我们建房的例子，建设过程中，Howell 一家不断地复查计划。他们也有很多机会来改米对他们所想要的东西的想法。同样地软件开发允许顾客复查每一步中的计划，及在设计上作出改动。毕竟如果开发者生产出了不满足顾客要求的不可思议的产品，最终系统将浪费了每个人的时间和努力。

因此，灵活地运用软件工程工具和技术是重要的。过去作为开发者我们作交定顾客从一开始就知道他们想要的东西。那种稳定性通常不是事实。当项目各个阶段渐渐显露时，开始没有预料到的限制出现了。例如，为特定项目选定好软硬件后，我们可能发费顾客需求的一个改变使用一特定 DBMS 正确地产生普通向顾客承诺过的菜单变得困难。或者我们发现我们的系统要与之接口的系统已经改为了加工或期整数据的格式。我们甚至发现软件硬件不能工作的象卖家文档中已经承诺过的那样好。因此，我们必须记位每个项目是独特的，工具和技术必须有所选择的，把映放置于这个单独项目之上的限制。

我们也承认，大多系统不能独自运行。它们与别的系统接口，或者接收或者提供信息。开发这样的系统是复杂的仅仅因为也要和与与之通讯的系统进行大量的协调。这种复杂在同时开发这些系统的表现的大为真切。过去开发者难以保证系统间接口文档的精确和完全。在后续章节，我们将谈一下控制接口问题。

变化的本质

这些问题是许多影响软件开发项目成功的因素的一部分。无论采取什么方法，我们必须前后都看看。就是说，向后看一看以前的开发项目以明白我们已领会了什么，不仅是有关确保软件质量方面的，而且也有关我们的技术和工具的有效性方面的。我们必须向前看，软件开发和软件产品的使用将可能改变我们在未来的实际的方式。Wassermane197.5 指出自 20 世纪 70 年代以来这些变化已经是富有戏剧性的。例如，早期应用程序设计的运行于单个处理器上，常常是一个主机。较为是一次性的，通常为一幅卡片或一个输入带，而输出是代数。系统以两种基本方式之一来设计：作为一种转换系统，输入转换为输出；或者作为一个事务，转入规定了哪个功能方没执行。现在的基于软件是远远不同并更加复杂的。典型地，他们运

行在每个系统上，有时配成具有分布功能的客户一般服务的结构。软件不仅执行用户需要的主要功能，而且还有网控。安全，用户接口的表示和处理，及数据或对象管理。系统分布模型开发方法，它采用一种线性发的开发活动，一个活动只须它的前导活动完成才开始我们将在第二章研究，而这种开发方法对今天系统来说不再具有伸缩性或者说不适合，在Wasserman(1996)的 Stevens 演讲中，他通过标识已经影响软件工程实线的七个关键因素（在图 1.12）中说明来概括了这些变化：

1、对商品而推向市场时间上的危急性
2、计算方面经济上转移；较低硬件费用和较高的开发和维护费用。
3、能有有桌面计算的利用
4、广泛的局域和广域网络互联。
5、面向对象技术的可用性和实用
6、运用窗口、图标、菜单和指针的图形用户接口（GUI）
7、软件开发的分布模型的不可能性预测性。

图 1.12 已经改为软件开发的关键因素

例如，市场压力意味着商家必须在竞争对手行运之前准备好新产品和服务。否则，商业自身的变化可能产生巨大的利害。因此，传统的复查和测试技术如果需要大量不能扣除的时间以减少故障损失，效率的话将不能使用，同样，以前为改善速度或减少空间而花在代码优化上的时间可能不再是英明的投资；一块额外的硬盘或内存卡可能是此问题更便宜的解决方案。

而且，桌面计算把开发权力放在了用户手里，他们现在用他们系统来开发电子表格和数据库应用，小程序，甚至特化用户接口和模示的实验。开发职责的转移意味着作为软件工程师的我们，很可能将建立更复杂的系统。类似地，巨大网络互联将有能力对大多用户和开发者可以利用，这使得用户在没有特别应用程序时也能更容易地查找信息。举例，搜索 WWW 是快速、容易并有交互的；用户再也不需要写一个数据闸应用程序来查找他或她需要的信息了。开发者也发现他们的工作加强了。而向对象技术，结合网络和用仓储库，使所开发者在新应用程序中将更大集合的可重用模块直接快速地包容进来加以利用。图形比用户接口（常用专门的工具开发）有助于在复杂应用中放置一个友好的桌面。因为我们分析问题的方工变化很复杂，我们能划分一个别系统以便我能并列地开发其子系统，这需求一种非常不同于“瀑布模型”的开发过程。我们也将在第二章中看到。对此过程我们有许多选择，其中包括一些允许我们建立原型（为与顾客用户验证需求的正确及许估设计上的灵活性，在活动中反复的过程。这些步骤有助于我们确保需求和设计在以代码实例化它们前尽可能没有毛病。

Wasserman 的软件工程原则。

Wasserman（1996）指出七个技术变化任何一个都对软件开发过程产生重大意义的影响。它们一起转变了我们工作的方式。在他的陈述中 Demaero 描述一种极端的转移，也说我们首先解决容易的问题；而那意味着留待解决的问题现在比以前更困难。Wasserman 谈论了这种挑战，提议软件工程中有八个概念形成了软件工程有高级原则的基础。在这里我们简单介绍它，在以后面章节会回到它上面看它们应用于我们工作中哪些地方以及如何运用。

抽象。有时，从问题的“自然状态”CVK ：象顾客或用户表达的一样一考虑也是件很

让人气馁的工作。我们有分到一个以一种有效归纳层次上对问题的一个描述，这让我们可专注于问题是关键方面，而不致于陷于细节。此概念不同于转换（transforhation），转换是指我们把一个问题计成另一个我们更好理解的环境；将问题从真实世界转移到数学世界，因此我们能使用数字来解决问题。

典型地，抽象包括确认对象的类以及更让我们将项目分组到一起；通过此方式，我们可处理更少的事务并把注意力放在每个类各项的共性上，我们可以读者论论在一个类中项的特性或展性并栓查在特性和类中的关系。例如假如我们被邀请建立一条大面复杂的河流的环境监测系统。监测设备可能得知空气质量，水质、温度、速度和别的环境特片的感尖器，但是，为达到目的，我们可选择定义一个名为“sensor”的类；此类中每项都有某种特性，不管它正监测的特征：高度、速度、电气要求，维护计划等等因了解问题内容和设计一个解决方案时我们处现这个类而非它的元素。这种方工有助于我们简化问题表述并专注于问题的本质元素或特征。

我们也可形成抽象的层次。例如，感应的是一型电子设备，我们可能有两型感应器：水感应器和空气感应器，因此我们可形成图 1.13 所示的简单层次。通过隐藏一些细节，我们可将注意力集中在对象的本质上，我们必须就对象来处理并找出简单而优美的解决方案。在第 5.6 和 7 章我们将紧密地看一下抽象和信息隐藏。

分析设计方法和概念。当你设计一个程序作为课堂作业时，你通常独自工作。你生成的文档是一份正式的给你作注解的描述：为什么会选择一个特定方法、表示名的意思、实现的是哪种算法，当和一个组来共同工作时，你必须与许多开发过程的参与者交流。大多工程师，讨论他们做哪类工程，都使哪一个标准概念帮助交流，并文档化决策。例如，一名建筑师画一张图表或蓝图，别的建筑师都能理解。更重要的是，通用概念让承继人理解建筑师的意图和想法。象我们将在第 4、5、6 和 7 章看到的，软件工程中没有相同标准并由此导致误解是今天软件工程中关键问题之一。

分析与设计方法提供给我们不止一种交流媒介。他们允许我们建模并检查完全性和一致性。而且我们可更容易地从以前对象应用需求和设计组件，这样相对容易的提高生产率和质量。

但在决定一套公用方法和工具前，还有许多公开的问题要解决。象我们在以后章节中将看到的，不同的工具和技术用于一个问题的不同方面，我们需要确定建模原语，以使我们用科学—技术便可抓住问题的所有重要方面。或者我们需要研究出一种能与所有方法使用的，可能以某种方式裁剪的表示技术。

用户接口原型。原型意思是建立一个系统的一个小版本，通常只具有有限的功能，原型用于：

- 帮助有户或顾客确定一个系统的关键需求
- 论证一个设计或方法的可行性

图 1.13　监测设备的简单层次

通常，原型反复的：我们建立一个原型，评估已（就用户和顾客的反馈）考虑变动可能怎样改善产品或设计，然后建另一个原型。当我们和顾客都认为我们有了问题的一个满意的解决方案在手边时，这个反复过程便结束了。

原型通常用于设计一个好的用户界面：User lwterface,交流中与用户交互的那部分。然而也有别的便用原型的机会，甚至在嵌入式系统（embedded system），例如，在某种意义上

用户界面是应用领域和软件开发者之间的一座桥梁，原型能产生表面问题和设想，而使用别的需求分析方法可能已经不是很清楚。我们将在第 4 和第 5 章考虑用户界面原型的角色。

软件体系结构。一个系统的整体体系不仅对发现和测试系统的难易度而且也对维护和改变的速度和效力很重要。体系结构的质量能构造成破坏系统；的确，象 Shaw 和 Garlan（1996）把体系结构单独作为一条原则来介绍，它的影响在整个开发过程中都能感觉得到。一个系统的体系结构应该反映了好的设计原则，我们将在第 5、7 章研究。一个系统体系结构以一套体系单元和单元间如何联系的图来描述系统。单元越独立，体系越模块化，我们就越容易设计和开发分片 Wasserman（1996）指出至少有五种方式将系统分成单元：

1、模块分解：基于模块被指定的功能
2、而向数据分解 S：基于外部数据结构
3、面向软件分解：基于系统必须处理的事件，
4、外部设计（side-in design）：基于用户给系统的输入
5、面向对象形设计，基于确定对象的其和相互关系

这些方法相互并不排斥，例如，我们能用面向事件分解设计一个用户接口，用面向对象或面向数据设计数据库。我们将在后面章节中更详细地考查这些技术。这些方法的重要性在于它们对我们设计经验的捕捉，通过应用我们已做的工作和在做时学到的东西来利用过去的项目。

软件过程（softwre process）自 20 世纪 80 年代，许多软件工程师已经仔细注意软件开发的过程（process）,和由此产生的产品开发活动中的组织和原则对软件质量和开发速度的贡献已经得到承认。然而 Wasserman 提到：

应用类型和组织文化的巨大变化使得不可能说明过程本身。因此似乎软件过程不象抽象的模块化那样是软件工程的基础。（wasserman1996）

代替地，他建议不同类型的软件需要不同的过程。特别是，Wasserman1996 建议企业范围应用程序需要大量的控制，而个体或部门应用可采用快速开发如图 1.14 所示。

用今天的工具，许多小或中等系统由一或两个开发者来建立，每个开发者必须担当多个角色。这个工具可能色括文本编辑与编程环境、测试支持，或许还有一个小数居库来捕获产品和过程本身的关键数据元素。因为项目的风险相对低，仅需少量的管理支持或复查。

然而，大而复杂的系统需要更多的结构，栓查和平衡，这样的系统通常涉及许多顾客和用户，开发持续一个长的时期。而且，开发者不总是控制了整个开发，因为一些重要子系统可能由别人提供或以硬件实现。这种高风险系统需要分析和设计工具，项目管理配置管理，更完善的测试工具，更严格的系统复查和原因分析，在第二章，我们将仔细看一下几个过程选择以弄明白怎样改变过程以用于不同目的。然后，在第 12、13 章，我们评估一些过程的效力并看一看改进的方法。

图 1.14 开发的不同(Wasserman1996)

复用（Reuse）。在软件开发和维护中，我们常通过应用以前开发中的某些共性来利用跨越应用程序的共同特征。例如，我们从一个项目到另一个项目都使用不同的操作作系统和 DBMS，而不是每次都建一个新的。类似地，在我们建立和以前做过的相似但不相同的系统时我们应用需求集，设计部分，和测试脚本或数据组。Baranes 和 Bollinger(1991)指出，应

用不是一个新观点，并提供了许多关于如何应用（不仅仅是代码）的有趣例子。

Priteo-Diaz(1991)介绍可应用组件的概念为一种商业资产。公司和组织投次于可应用项目（item），然后在这些项目一次又一次在后来的项目里使用时获得可以计量的利润。然而，建立一个长期，有效力的应用程序是困难的，因为有几个障碍：

- 有时建立一个小组件比可应用组件仓储库中搜索更快
- 使组件足够通用以使未来别的开发者很容易地应用它，这需花额外的时间
- 难以文档化保质程度和已做的测试，以便一名潜在应用者难对组件质量放心
- 如果一个应用组件失败或城要更新，谁来负责是不清楚的
- 理解和应用别人写的组件是费钱费时的
- 通常在能家长性和特殊性之间存在冲突

我们将在第 12 章考查应有和的更多细节，考查几个成功应用的例子。

度量。改良是软件工程研究的推动力：改进过程、资源和方法以便我们产生和维护更好的产品，但有时我们只是泛泛地改进目标，而没有从量上描述我们到了哪里和我们想到哪里。因此，软件度量已成为好的软件工程实践的一个关键方面，通过量我们能到哪里及我们能做什么，我们以一种允许估进展的通用数学语言找述了我们的行为和他们的结果。另外，一种数量的方法使我们可比较不同项目的进展，例如，在 John Young 担任 Hewlett-Packard 的 CEO 时，他设定了“IOX”目标，对 Hewlett-Packard 的每个工程，不管它的应用类型或领域，在持量和生产率上都有十倍改进。（Gray and Cas well 1987）

在较低层次的抽象中，度量有助于使过程和产品的特定特征更具可见性。反我们对其实观察的世界的理解转换成正式的数学世界中的元素，和关系常常是很有用的，因为在数学世界中我们能进一步操作元素和关系以获得更深的理解。象图 1.15 所示，我们能用数学和统计来解决头问题寻找趋势，或特征化一个条件（运用媒介和标准偏差）然后，新信息能映射回真实世界并被用作解决我们正在努力要解决的经验问题的方案的一部分。整个本书，我们将看到如何运用度量支持分析和作决策的例子。

图 1.15 使用度量来帮助找到解决方案

工具和集成环境。很多年来，卖主以 CASE 工具招来顾客。在那里，标准化的集成开发环境能增强软件开发。然而我们已经看到开发者使用各种过程，方法或资源是多么的不同，因此，一个统一思想一的方法说来容易做却难。

另一方面，研究人员已经推荐了向个基准体系，让我们可比较对照现存的和推荐的环境。这些基础标准体允许我们考查每个软件软件工程环境提供的服务并决定哪一个环境对给定问题或应用开发是最好的。

比较工具一个主要难题就是卖主很少谈起整个开发生存周期。代替地，他们专注于小的活动集，如设计或测试，并且还取决于把被选择工具合并到开发环境的使用者。Wasswemen(1990)指出了任何工具集成中必须谈到的王个问题：

1、平台集成：工具在一个异类网络上互操作的能力；

2、表示集成：用户里面的共同特征

3、过程集成：工具和开发过程间的关联

4、数据集成：工具共享数据的方式

5、控制集成：一个工具通知和启动另一工具中行为的能力在本书后面每一章，我们将考查支持本章陈述活动和概念的工具。你可把这里所述 8 个概念当作 8 条线过织成了本书的

结构，把我们所称的软件工程的分离的活动拴在一起。当我们对软件工程了解更多之后，我们将再访这些观念来看看它们是如何统一并提升软件工程作为一门科学纪律的。

1.9 信息系统的例子

注：Piccadilly 皮克迪利，英国的一个地区

整部书我们将用两个例子结束每一章，一个是信息系统的另一个是实时系统的我们将章节里所描述的概念应用于每个例子的某些方面，因此你能看到概念在实践中的意义，而不仅仅是在理论上。

我们信息系统的例子自 James 和 Suzanne Robertson，完全系统分析：工作手册，教科书和答案。它涉及到一个替 Piccadilly 电视台（一个英国地区性电视权拥有者）销售广告时段的系统的开发。图 1.16 说明了 Piccadilly 电视台收视区，象我们将看到的对电视时间的价格上的约束有很多且是变化的，因此问题即有趣又困难。在本书中，我们突出问题的各方面和它的解决方案；Robertson 的书向你揭示了捕获和分析系统需求的详细的方法。

在英国，广播委员会颁发 8 年特许权给一家商业电视公司，给他独有的在这个国家仔细划定的区域广播它的节目。作为回报“特权者”广播时必须使戏剧、喜剧、体育、儿童的和其他的节目达到规定了的平衡。而且，对在哪些时段播放哪些节目都有限制，对节目和商业广告的内容也规定。

一名商业广告客户有几个触及 Midlanels 的观众的选择：Piccadilly 电缆通道和卫星通道。然而，Piccadilly 吸引了大多数观众。因此，Piccadilly 必须设定阶格来吸引部分广告客户的预算。吸引广告客户注意力的方式之一是用观众收视率。（反映了全天不同时间观众的数量和类型）收视率根据节目类型、观众类型、时段、电视公司等来汇报。但广告价格不仅仅只依赖于收视率。例如，如果广告客户买了大方时数，每小时价格就会便谊些。而且，对在某些时间段广告的类型和对某类节目也有限制。例如：

图 1.16 Piccadilly Television 特权区

酒精饮料的广告仅在 9，PM 后才可播出

如果一名演员出现在电视上了，哪么这名演员有关广告在他出现的 45 分钟里不能广播

当我们更详细操控究这个例子时，我们将提到客户外的有关广告和费用的规章。图 1.17 中系统情况图显示了系统边界及它如何与这些规则关联。阴影圆就是作为信息系统例子我们所关心的 Piccadilly 系统；系统边界就是圆周。箭头和方框显示了可能影响 Piccadilly 系统工作的项，但我们仅就它们的资源和目标把它们视为一个输入、输出集。

在后面章中，我们将使阴影圆中活动和元素可见，我们将使用每章所描述的软件工程技术来考查系统的设计和开发。

1.10 实时例子

我们的实时例子是基于 Ariane-5,一种属于欧洲航天局（ESA）的空间火箭，的内嵌软件。1996 年 6 月 4 日，在它的处女飞行中，Ariane-5 被发射并正确运行约 40 秒。然后，它开始转离航向，在一个 Ariane 地面控制的指挥下，火箭被遥控摧毁。这颗未投保火箭的报毁不

仅失去了火箭本身，而且失去了它所容纳的四颗卫星，这场灾难的代价是 5 亿美元。

图 1.17

软件包含于这个系统的各个方面，从火箭的导航系统到组件部分的内部运转。火箭的失败和随后的报毁引发了很多关于软件质量问题，就象我们在后面章节看到的，调查问题原因的调查委员会专注于软件质量和确实性上。在本章我们我们从火箭的商业阶价值来看质量 Ariane-5 的成功得到许多组织的资助：区分为航天局（ESA）CNES（法国的航天局，全面指挥 Ariane）计划的一系列延期和问题的另一起，1995 年一次发动机测试时氮泄露杀死了两名工程师。然而，这次六月事故是第一起将原因直接归究于软件故障的事故。

这次事故的商业冲击在装备上超过了 5 亿美元，1996 年 Ariane-4 火箭和其前期变种囊括了世界发射合同的一半以上，领先美、俄、中的发射者。因此，Ariane 火箭的失败使计划的可信性和潜在生意十分危险。

将来的生意部分建立在新火箭比前述发射者运送更重的有效载荷入轨的能力。Ariane-5 设计成运载达 6.8 吨的单颗卫星或合重 5.9 吨的双卫星。到 2002 年止进一步的开发工作就是希望在发射能力上再增加一吨。增加了的运输能力拥有明显的优势；通常，经营者通过共享发射来减少费用，因此 Ariane 能一次提供几个公司有效负载的发射。

考虑在这个例子的上下文环境里质量的意思。Ariane-5 证明是顾客错误指定的需求的结果。既然这样，开发者可能会申称系统是质量的它仅仅按照了错误原规格建立。代替地，被组成调查事故原因，寻求解决灾难办法的调查委员会提到：

委员会的裁决速建立在东西 Ariane-5 项目组彻底坦率的阿述和文档的基础上，从一秀贩工程工作和文档的完整性和可跟综性考虑，文档已经证明了 Ariane-5 计划的高质量。(Lions et al 1996)

但从用户和顾客的观点，规格过程本应该是规格缺陷：在伤害发生前迫使顾客纠正规格（这两方面）是够好的。调查委员会承认：

SRI（问题原因最终被定位的子系统）的提供者仅仅遵从提供给它的规格说明，这个子系统保证只要一检测到故障处理器将停止工作。产生的异常不是由于随机失败而是设计错误。故障检测到了，但没有恰当处理，因为人们已经认为：软件只有在有故障时才应考虑纠正。委员会有理由相信此观点，也在 Ariane-5 软件设计中被接受。委员会赞同相反的观点。软件应该被假定是不完善的，直到应用普通认可的最好的实践方未能证明它是正确的为止。（Lions et al.1996）

在后面的章节，我们将更详细地研究这个例子，考虑全开发者和顾客的决策上有关设计测试和维护上的暗示。我们将看到开发之初可怜的系统工程是如何导致一系列可怜的决策。依次导致灾难的。另一方面，对所有受关注东西的公开，包扩 ESA 和调查委员，与高质量文档及一个最诚挚的快速寻求真相的愿望，导致了直接问题的快速解决和一份有效防止未来的此类问题的计划。

系统观点让调查委员会，在开发者配合下，把 Ariane-5 视为一子系统集。这个集合反映了问题的分析（下如本章所述）以便开发者能从事于带有明显不同功能的子系统中。例如：

火箭在空中资态和运动由 SRI 系统测定。它有自己内部计算机，在哪里，在来自一个“strop-down”惯性平台（带有激光陀螺仪和加速计）的信息的基础上计算角度和速率。一自 SRI 的数据通过数据总线传到板上计算机（On –Bccrd ,OBC）,OBC 执行飞行程序，通过侍服阀和液压激励的控制坚固的后推器的喷嘴和 Vulcain 低温发动机。

但是，方案的集合必须色括对所有组件部分的总体认识，这些部分被放于一起观察的确

定把它们粘于一起的胶水是否充和适当。在 Ariane-5 情况里，调查委员会暗示顾客与开发者本应一起工作的找到临界软件并确保它不仅能处理预料而且非预料的行为。

这意味着临界软件——软件故障使任务处于风险之中——必须在非常详细的层次上确定下来，异常行为必须限定，一个合理的备份策略必须将软件故障考虑进来。（Lions et al 1996）.

1.11 本章对你意味着什么？

本章已经介绍了许多对好的软件工程研究和实践很重要的概念。你，作为单独软件开发者，能以发下方式运用这些概念：

在有人给你一个问解决时（无论方案是否涉及软件），你能通过把它分解组件部分和组件部分间的关系来分析之。然后，你能通过解决单个子问题，合并子问题成一整体一综合出一个解决方案。

你必须理解，需求是会变的，即使人正分析问题，正建立一个方案的时候。因此你的方案应好好文档化并有伸缩性，你应该文档化你的设想和使用的算法（以便于以后很容易地更改）

你必须从向个方面看质量，理解技术持量和商业质量是非常不同的。

你必须使用抽象的度量来帮助确定问题和方案的本质方面。

记住系统边界，以便你的系统不会与别的同你正建的系统交互的相关系统重叠。

从本页起“Deveyeopment Team”秋为开发团队

1.12 本章对你的开发团队意味着什么？

你的很多工作是在作为一个较大开发团队的一员来做的。象我们在本章看到的，开发涉及需求分析，设计、实现、测试、配置管理、质量保证等等。团体的一些人可能看你一样同时担当几个角色，项目的成功在很大程度上依赖于团队成员间的交流和协调。我们已在本章看到你可通过如下选择来帮助你成功：

对你团队的规模、风险水平、应用领域恰当的开发过程

被很好集成并支持项目法要求的交流类型的工具

能提供给你尽可能多的可见度和理解的试题 和支持

1.13 本章对研究人员的意义

本章所讨论的问题都是很好可进一步研究的主题。我们已经提到软件工程中一些公开的问题、包括：

简化问题处理的抽象的正确层次

使问题和方案的本质属性可见和有帮助的正确度量

一个可使子问题能解的恰当的问题解决

使工具集成容易有效，使项目参与者最效交流的一个公共框架或符号

在后面章节，我们将描述许多技术。一些使用了并被很好的证明了的软件开发实践，而别的虽被推荐但还仅在一引起小的，“玩具”的，学生项目中得到验证。我们希望引导你如

何去改善人现在正做的工作，并同时激发你对在未来尝试新技术和过程的创造力和思考。

1.14 学期项目(TERM PROJCT)

不参与开发一个软件项目（与同事一道），你不可能学会软件工程。因此本书每一章将描述一个学期项目的信息，你可和一个同不团队来执行此学期项目的开发。此项目基于一个真实机构的一个真实系统，让你经历一些非常真实的挑战、分析、设计、实践、测试和维护。此外，因为你在和一个团队共同工作，所以你将安处理团队分歧和项目管理的问题。

这个学期项目涉及的是（在人买房进可能要和银行商议的）贷款种类。银行通过多种途径产生收入，通常是以通过以低利润从储户借钱，然后以较高的利率把把钱以银贷款方式借出去。然而，长期财产贷款（如抵押）曲型地有长达 15.25 或甚至 30 年的其限。也就是说你有 15.25 或 30 年的时间来偿还贷款：年金（你最初借的钱）十按指定率的利息，尽管来自这些贷款利息的收入是有利的，但这些贷款长时间占用资金，妨碍了银行将资金用于其它交易。因而，银行常将贷款卖给稳固的机构，取得较短时期利润一换取资本以其他方式使用的自由。

你的学期项目的应用程序被称为 Loan Arranger（贷款安全系统），也用一配合一个（虚构的）金融联全组织(Financial Consolidotion Orgahization,Fco)处理买自很行的贷款。FCO 通过从银行购买贷款再出售给投资者来赚钱。银行出售贷款给 FCO，作为回报获得本金。然后，象我们将看到的，FCO 出售贷款给那些愿意比银行等给更长时间获得回报和投资者。

为弄明白交易如何进行，考虑你如何获得一笔购房贷款（被称为“代押”）

你可通过付$50,000 作为初次支付（称为“down payment”）并支付余下$100，000 的一笔贷款一购买一幢价值$150，000 的房子。你从第一国家银行（First National Bamk）得到的贷款的期限可能是 30 年（以 5%的利息），这个术语（注 terms）意味着第一国家银行给你 30 年（贷款期限）来偿还你借的数量（本金）加上利息（无论什么你都不用马上返还）例如，你可能通过 30 年里每月支付一次一偿还这$100，000，不家未付余额的利息（也就是，360 次分期付款或每月一次的支付）。如果初始余额是$100，000 银行使用本金量利率，你必面还清的时间及每月支付额应相同的假是未计算的支付额。

例如，假如银行告诉你每月支付额将是$536.82，第一月的利息是（1/12）×（0.5）×($100,000),或$416.67,此支付的余下部分（$53682416.67）用来减少年金: $120.15，因此利息减少到(1/12)×(0.5)×($100,000)，或$416.67。此支付的其余部分($536.82 416.67)用来减少本金：$120.15。第二个月，你先用$100,000 减去$120.15，因此利息减少到（1/12）×（0.5）×（$100,000-120.15)，或$416.17。因此，第二个月支付额中仅$416.17 是利息而余下部分$120.65 用于减少其余本金。整个时间上，你支付越来越少的利息而更多的用于减少本金额，等到你付清了所有本金和拥有了你的财产，摆脱清除了银行施于的累赘。

在你正在支付期间的某个时间，第一国家银行可能将你的贷款买给了 FCO。每一国家限银行与 FCO 商议价格。依次他，FCO 可能将你的贷款卖给 ABC 投资公司。你每月依然“支付抵押，但你的支付进入 ABC，而不是第一国家银行。通常，FCO 以”色（bwndles）“出售贷款，而不是学独的贷款”，以便投资者基于风险、涉及的本金和期望的返还来购买一个贷款的收集（collection)。换句话说，一名投资者（如 ABC）能和 FCO 签约，并详细说明也希望投资多少钱，多长时间、愿意承担多大风险（基于偿还贷款的人或机构的历史），期望的利润。

Loan Arranger 是一个让 FCO 分析员以选择一色贷款来匹配一名投资者乐意的投资特征的应用程序。应用程序取 FCO 从借出机构购买的贷款的信息。当一名投资者详细说明了投

资标准后，系统选择满足标准的最佳贷款色。在系统允许一些高级优化——诸如从那些可用贷款的子集（例如，从所有属萨诸塞分的贷款，而非全部可用贷款中）选择最好的贷款色——的时候，系统将仍然允许一名分析员手工为客户选取择在一色中的贷款。除了色选择外，系统也自动化信息管理活动，如更新银行信息，更新贷款信息，和在银行每月提供贷款信息的增加此信息）

我们能总结这样的信息：Loan Arranger 系统允许一名贷款分析员存取有关由 FCO 从多名借出机构购买的、打算重新色装贷款未出售给其它投资者的抵押（家庭贷款，这里简单找述为“贷款”）的信息。由 FCO 购买用于投资和再出售的贷款信息仓储库中的投资历组合贷款信息。另外，系统允许贷款分析员创建贷款色来卖给投资者，Loan Arranger 的一名使用者是跟踪 FCO 购买的抵押的贷款分析员。

后几章，我们将更深处地探究系统需求。

图 1.17 皮克迪利显示系统边界的上下文图

第 2 章 过程建模与生命周期

本章，我们来看一下

- “过程”的意思
- 软件开发产品
- 过程和资源
- 几个软件开发过程模型
- 过程建模的工具和技术

在第 1 章我们看到做软件工程既是一个有创造力又是一个逐步的过程，通常涉及到许多人生产许多不同的产品。本章，我们将详细地考查这些步骤，看一看组织我们的活动方式以便协调我们做什么和什么时候做。本章我们从定义过程的意义开始，以便在我们理解在软件开发建模的时候必须要包括什么东西。接着，我们考查几个典型的软件过程模型。一旦我们知道了希望使用的模型的类型，我们便紧密地看一看两种类型的建型技术：静态的和动态的。最后我们应用这些技术中的几个到我们的信息系统和实时系统的例子中。

2.1 过程的意义

当我们提供一项服务或建造一个产品，无论它是开发软件、写一份报告、或者作一次商务旅行，我们总按照一序列的步骤来完成一套任务。这些任务总是每次按同样的次序来执行；例如，通常，在房屋的线路安装好之前你不会建清水墙，或者，在所有的成分混和在一起前烤面包。我们可以把一个有序任务集合看作是一个过程（process）：一个用来产生某类想要的产品所涉及的活动、约束和资源的步骤序列。

一个过程常常涉及一套工具和技术，象我们在第 1 章所定义的一样。任何过程有如下的特征：

- 过程规定了所有的主要处理活动
- 过程使用资源，服从一套约束（如时间进度），并产生中间和最终产品。

- 过程可能由许多子过程（它们通过某种方式联接在一起）组成。过程可能被定义为一个过程层次，这样组织起来以便每个子过程有它们自己的过程模型。
- 每个处理活动都有其入口和出口，以便我们知道活动何时开始与结束。
- 活动以一定次序组织起来，以便一个活动相对其他活动应该何时开始清楚。每个过程有一套指导原则来解释每项活动的目标。
- 约束或控制可以应用到一项活动、资源或产品。例如，预算或进度可能限制一个活动可占时间的长度，或者，一件工具可以限制资源利用的方式。

当过程涉及某种产品的建立时，我们有时称这个过程为一个生命周期（life cycle）。因此，软件开发过程有时称为软件生命周期（software life cycle），因为它描述了一件软件产品的生命：从它的概念到实现、交付、使用、和维护。

过程是重要的，因为往活动集上加入了一致性和结构。当我们知道怎样把事做好且希望确保别人也以同样的方式做的时候，这些特征是有用的。一个软件开发的过程可以用灵活的方式来描述以允许人们用拿手的技术和工具来设计和建立软件；一个过程模型要能要求设计在编码前，但允许使用许多不同的设计技术。因而，过程有助于将多人一切生产的产的产品或服务维护持一定水平的一致和质量。

过程(cprocess)不仅仅是程序（procedure）。程序（procelure）象食谱：一种组合工具和技术的生产一件产品的结构化方式。过程（process）是程序（proceduers）的收集，组织起来以便于我们建立满足一套目标或标准的产品。事实上，过程（process）可能建议我们从几个程序（protedures）中挑选，只要我们要求的目标达到就行。

过程结构（process structure）通过让我们考察理解控制改进组成过程的活动来引导我们的行为。为了弄清楚如何进行，考虑制作带糖衣的巧克力糕点的过程。此过程可能含几套程序（procedwes），如买成分和寻找适当的烹饪工具。食谱描述了实际混和与烘烤糕点的程序。食谱包括活动（如“与别的成分混和前敲鸡蛋”）约束（如在“与糖沸和前加热巧克力还是沸点中心”温度要求）和资源（如糖、面粉、鸡蛋、巧克力）。假设 Chank 按这份食谱烤一只巧克力蛋糕。蛋糕做好后，他尝了一个样品，觉得太甜，他看食谱型 明白哪个成份与甜度有着：糖，然后他烤了另一只蛋糕。但是这次他减少了新食谱中糖的份量。他再一次品尝，但现在他发现没有够多的巧克力味道，他补充可可粉到第二份修正样品中并再次品尝。在几次这么反复（每次改变一种成份或活动）后，chunk 得到了一个投其喜好的蛋糕。如果没有文档化这部分过程的食谱，Chund 可能就不能容易地做改动并评价结果。

过程也有助于我们赢得我们的经验并将它传给其他人。。

同样，我们要从过去的开发工程中学习，将生产高质量软件的最好的实践文档化，并遵从一个软件开发过程以便我们能理解，控制和改进我们替顾客建立产品时所发生的事。软件开发通常包括以下几个阶段：

- 需求分析和定义
- 系统设计
- 程序——
- 写程序（程序实现）
- 单元测试
- 综合测试
- 系统测试
- 系统交付
- 维护

每个阶段本身就是能描述为一个活动集合的过程（或过程收集）。每项活动涉汲约束，与交出和资源。例如，南非求分析和定义需求用户所表达的有关想要的功能和特征的叙述一

作为初始输入，此阶段的教后输出是一个需求集合，但当用户和开发者问的对话导致变更或可选取事物时也可能有中间产品存在。我们也有约束，象用于产生需求文档的预算和进度对各类需求标准。

本书会讨论每一阶段。对每一阶段，我们都紧密地看一看处理过程（processes）资源、活动和转出，学习它们如何为最终的有用产品的持量作出贡献。

每一过程都能用多种方式（文本、图片或者二者结合）一描述。

2.2 软件过程模型（SOFTWARE PROCESS MODEL）

在软件工程文献中已经描述了许多过程模型。一些是对软件开发应该如何前进的方式的规定：别的是对实际中软件开发如何进行的描述。理论，这两类模型应相似或相同，但实际上不是。建立一个过程模型并讨论它的子过程有助于团队理解“应该是什么”和“实际是什么”之间的差别。

有几个别的应进行过程建模的原因：

当一个群体写下它的开发过程的描述等，就形成了对软件开发所涉及的资源约束的共同理解。

建立一个过程模型帮助团队发现在过程中的不一致，多余，遗漏

模型应该反映开发的目标，辟如建立了模型后，开发团队评估候选活动

每个过程应该因地而剪裁。建立一个过程模型有助于开发团队理解在何处剪裁。

每个软件开发过程模型包括作为输出入的系统需求和作为输出的最后产品，很多那样的模型已经提出多年了。我们提出几个流行的模型以理解其共同特征和差异。

瀑布模型（Waterfall Model）

如图 2.1 各阶段名胜瀑布一样由一个阶段向另一阶段落下。

图 2.1 瀑布模型

与每个过程活动相联系的便是里程碑和可交付物，以便项目经理能评估项目及时在某点完成有多近，例如，“单元和综合测试”是随里程碑“已写被测试，集成的代码模块”而结束的，中间交付产品是已测试代码的副本。接着，代码被交给系统测试员以便与别的系统部件硬件或软件合并，并作为更大的整体来测试。

瀑布模型在帮助开发者安排他们需要做什么方面非常有用。它的简易性使它易于向不熟悉软件开发的顾客解释说明；也使得“为开始下一阶段需要哪些中间产品”更为明确，许多别的更复杂的模型实际上只是瀑布模型的一个修饰，加入了反馈循环和客外活动。

瀑布模型的最大问题在于没有反映代码等实际的开发方式。开发软件能常带有大量重复。实际的软件开发过程，如果失控，可能看起来如图 2.2

图 2.2 现实中的软件开发过程

软件开发过程通过包括增强理解的活动和子过程来控制这种打击。“原型”就是那样一个子过程；原型（prototype）是一个局部地开发的产品，使顾客和开发者来考察目标系统的某些方面，决定它是否对完成的产品合适或恰当，例如，开发者要能建立一个系统实现一些关键性需求的一小部分以确信需求是一致、灵活、实际的；如果不，则可在需求阶段就做修改。类似地，设计部分也可被原型化，象图 2.3 设计原型帮助开发者评估备用的设计策略。并决定哪个对选定项目是最好的。确认（vaidation）确保系统已实现了所有需求；验证（verification）确保每一功能正确地工作。也就是说，确认（vaidation）确保开发者正建立正确的产品（按照规格），而验证（vaidation）检查实现的质量。“原型法”对确认（vaidation）和验证（vaidation）是有用的。但这些活动也发生在开发过程的别的部分。

图 2.3 带原型的瀑布模型

V 模型（VModel）

V 模型是瀑布模型的变种，示范了如何将测试和分析设计关系起来。如图 2.4,编码为 V 的顶点,分析和设计在左,测试和维护在右。

V 模型建议单元和综合测试也应被用来验证(verify)程序设计。也就是说，学员的综合测试期间，编码员和测试小组成员应确保程序设计的系统设计方面已经在代码中正确实现。类似地，系统测试应验证系统设计，确保所有系统设计方面被正确地实现。验收测试（由成客方指导面解开发者）通过将一个测试步骤和规格中每全个元素联系起来确认需求。

V 模型左侧与右侧的联接暗示如果在验证和确认期间发现问题，那么 V 的左侧能被重新执行来修改并改进需求。设计和编码。

图 2.5 原型模型

原型模型（prototyping Model）

我们已看到如何用原型活动来修改瀑布模型以增进理解。但原型法不需要只是瀑布模型的附属物；它自身就是一个有效的过程模型原基础，如图 2.5 原型模型容许快速地建设起系统的全部或部分的理解或澄清。总目标是减少开发中的风险或不确定。

例如，系统开发可能从顾客和用户能提供的一个需求的空间集合开始。然后，请有兴趣的参与者来看一看直接向顾客和用户来使用的可能的屏幕、图表、报告和别的系统作业。当用户和顾客决定了他们所需的东西时，需求被修正。一是对需求应该是什么有了共识。开发者继续设计。再次地，备选设计被探索出来（通常要和服客、用户协商）修正、初始的设计

直到开发者、用户、和顾客对结果满意。的确有时在考虑设计备选方案的揭示出需求上的问题，并且开发者回到需求活动来重新考虑并更改需求规格。最终系统被编码，备选方案被讨论，并有可能再次作需求到设计的反复。

操作规格（Operational Specification）模型

对许多系统，需求的不确定导致以后开发中的变和问题。Zave 提出一种过程模型，它也许开发者和顾客在开发早期检查需求和它们的暗示，从而可讨论并解决一些不确定。在操作型的规格模型（Operational Specification Model）中，通过演示系统的行为的方式来评价或执行系统需求。也就是说，一旦需求指定后，它们可使用一软件来饰演，“以便在设计开始之前能计允许评价其暗示。例如，如果规格需要目标系统处理 24 个用户，一规格的一个可执行形势能帮助分析员决定是否这样的用户数综合系统施加了太多的性能员理。

这种类型的处理所需不用于传系统模型（诸如瀑布模型）。瀑布模型把系统的功能和设计分开，想保持顾客需要与实现分离。而操作规格模型允许功能和设计合并。图 2.6 解释了一个操作规格如何工作。注意它和原型法类似；此过程使得用户和开发者可在早期就考查需求。

转换模型.（Transformation Model）

Balzer 的转换模型通过削减几个主要的开发步骤来减少出错的机会。通过使用自动化支持，转换过程运用一系列转换来将规格转换为一个可交付系统。

样本转换包括：

- 改变数据表示
- 选择算法
- 优化
- 编译

因为从规格到交付的系统可取许多路径，转换序列和它们反映出的决策被保留为一个正试的开发记录。

此转换方法有很好的前途。但是一个最主要的阻碍就是要使用它就需要精确表达的形式规格以例子转换能操作它，如图 2.7 所示。当形式规格方法变得更加流行时，转换柳模型可能获得更广的承认。

图 2.7 转换模型

分阶段的开发：增量的反复（Phesed Developinent:Increments and terations）

早期软件开发，顾客愿意在软件系统准备好之前等待一个长的时间。然而现在的商业环境不再容忍长时间的延期。软件帮助区分市场上的产品，顾客总是在寻求新的功能。从而，新过程模型开发出来帮助减少周期。

一种减少周期的方法就是使用分阶段开发如图 2.8，系统被设计成能分片交付，使得用户有某些功能而其余的正在开发之中。因而通常有两种系统同时运转：产品系统和开发系统。产品操作型成产品系统是当前正由顾客和用户使用的系统。而开发系统是准备用来代替当前产品系统的下一版本系统。通常我们根据版本号提及系统：开发者建立发行版 1，测试它，

将它提交给用户作为第一运作发行看到（产品发行版）。然后，在用户使用发行版 1 时，开发者正建立发行版 2。因而，开发者总是从事于发行版 n+1，而发行版 n 是运作的版本

有两种最流行的方法就是增量开发和反复开发。

增量开发（Incremental Developmevt）中，需求文档中说明的系统按功能划分子系统。通过从一个小的功能子系统开始确定发行版，然后在每一新发行版中增加功能。图 2.9 上部显示增量开发是如何随每一新发行版建立起全部功能的。

然而，**反复开发（iteraeive development**）在一开始就交付一个完全系统，然后随每一发行版改变每一子系统的功能。图 2.9 的底部解释了在一次反复开发中的三个发行版。

图 2.8 阶段开发模型

增量开发

反复开发

图 2.9 增量和反复模型

为理解二者的差别，考虑一个字处理包。假设此包将交付三种类型的功能：建立文本，组织文本（如剪切和粘贴），和格式化文本（如使用不同类型的尺寸和风格）使用增量开发时，我们可能在发行版 1 中仅提供建立功能，然后在版本 2 提供建立和组织功能，然后最终在版本 3 中提供建立组织、格式化功能。

而在反复开发中，我们在版本 1 中提供所有三种类型功能的最初形式。例如，我们能创建文本并能剪切，粘粘它，但剪切、粘贴功能可能是笨拙或慢的。因此在版本 2 中，我们拥有同样的功能，但已增强了质量；现在剪切和粘贴是容易且快速的。每一发行版的同样的方式改进了前一版。

实际上，许多机构使用反复与增量开发的一个组合。新版本可能包括了新功能，而现存功能可能也得到增强。

螺旋模型（spiral mldes）

Boehm 从所涉及的风险来看待软件开发过程，提议螺旋模型将风险管理结合到开发活动中，以最小化并且控制风险。如图 2.10，螺旋模型在某种意义上象图 2.9 所示的反复开发。这个过程以需求和开发初始计划包括预算、约呸……开始，在一个操作概念文档生成的描述系统应如何工作前插入了一个评估风险原型化备用方案析步骤。

从那份文档中，需求集合被说明并细至以保证需求尽可能的一致。因而，操作的概念是第一次反复的产品，需求是第二反复的主要产品，在第三次反复中，系统开发产生设计，第

四次反复使得能进行测试。

随着每一次反复，风险分析根据需求和约束来衡量不同备选项，并且，在特定备选项中选取取，原型用来验证灵活性和适宜性。当确定出风险时，项目经理必须决定如何消除或最小化风险。

图 2.10 螺旋模型

无论使用何种过程查型，许我活动对所有来说是公共的。

2.3 过程建模的工具和技术

一旦你决定了你要在过程模型中获得的东西，你就有许多种有关建模工具和技术的选择，恰当的技术依赖于你的目标和你喜爱的工作风格。特别地，你对表示法的选择依赖你想在模型中获得的东西。

在本章，表示法（notation）相对模型的类型是次要的，我们主要关注两个主要种类：静态的和动态的。静态模型（statio model）描绘处理（process）,显示输入如何转换为输出。动态模型（dynamic model）能扮演处理（process），以便用户能看到中间和最终产品在整个时间上是如何转换的。

静态建模型：Lai 表示法（stotic modeling:Lai Notation）.

有很多静态地进行过程建模的方式。在 20 世纪 9 年代初，Lai 提出了一个完整的过程表示法，打算使人们能给任何过程在任意的细节级建模。它建立在一个范例上，在那里人饰演角色，而资源饰演活动，导致产品的生产。这个过程模型显示了角色，活动和产品间的关系，状态表显示了给定时刻每个产品的完全性信息。

特别地，一个过程的活动按 7 种类型来刻画：

1．活动（Activity）将在一个过程中将发生的事。此元素能相关于：前后发生了什么，需要什么资源，什么触发了活动的开始，什么规则控制活动，如何描述算法和学到的教为止，如何将活动和项目团队联系起来。

2．次序：活动的顺序。

3．过程模型：对系统的感兴趣部分的视图。因而，过程的部分能被表示为分离的模型，或者预测过程行为或者考查些特征。

4．资源：一个必需的项、工具或人，资源可包括设备、时间、办公空间、人，技术等。过程模型确定每个活动对每种资源各需多少。

5．控制：对过程扮演的外来影响。

6．政策：指导原则。可能包括一个指定的开发过程，必须要使用的工具。

7．组织：过程代理的层次结构，物理分组对应逻辑分组和相关角色。

过程描述本身可公为几个抽象层次，包括在构建特定模块中指导资源的使用的软件开发过程，及类似于螺旋或瀑布模型的普通模型。Lai 表示法包括几个模板，如产品定义模板（Artifact Definition Template），它记录有关特定产品的信息。

Lai 的方未能可应用于软件开发过程建模。

动态建模（trynamic Modeling:Synamic）

过程模型的一个可取的属性就是扮演不定过程的能力，因此活动发生时我们能观察资源和产品都发生了什么？换句话说，我们想描述一个过程模型，观察软件显示除我们资源如何流过活动变成输出。动态过程视图使我们能模拟出过程，并在资源实际消耗之前是作更改。

2.5 信息系统例子

让我们考虑哪种开发过程用于支持我们的信息系统的例子，皮克迪星电视（Picadily Television）广告程序。回顾一下，对什么时候卖何种类型的广告有很多约束，且规章可能随广告标准局（Avertising Standard Authority）和别的规章委员会的裁定而改变。因此，我们想建立一个能维护和修改的软件系统。甚至有可能在我们正搭建这个系统时约束发生变化。

对我们的系统来说瀑布模型太古板，因为需求分析阶段完成后，容许很少的灵活性。。原型对建用户界面可能是有用的。因此我们在模型中包括某种原型。但大部分的不确定在于广告规章制度和商业约束。我们想使用一个能随系统进化而使用并重用的过程模型。一个螺旋模型的变种是很好的候选者。

Boehms 的螺旋模型的表示是高层次的，没有够多的细节指导分析员，设计员，编辑员和测试员的行动。然而，有许多技术和工具可以选择以在较好细节层上表示模型。技术或工具的选择部分依赖个人经验和喜好，部分依赖于对被表示过程的类型的适宜性。让我们看一下 Lai 的表示法如何用于表示部分皮克迪星系统的开发过程。

因为我们想使用螺旋模型帮助管理风险，我们必面在我产的过程模型中包括风险特征。也就是说，风险是我们必须描述的产品，以便我们能度量和跟综，每一次反复中的风险，我们从两个层面考虑风险：可能性和严重性。可能性(prodoebility)是特定问题将发生的可能性，严重性（severity）是它将常给系统的冲击。

我们在 Lai 产品表中表示这些风险条件，如表 2.2。在此，风险是带有子产品可能性和严重性的产品。为简化起见，对每一子产品我们仅选择两个状记：可能性的高低，和严重性的大小。

2.6 实时例子

Ariane-5（阿丽亚娜 5）软件涉及到来自 Ariane-4 的软件包应用。应用复用（reuse）是想减少风险，增加生产率，及提高质量。因此，任何用来开发 Ariane 软件的过程模型应该包括应用活动。特别地，过程模型必须包括检查可重用组件的质量的活动，并且确保应用软件正确地工作于新系统的设计的上下文中。

这样的一个过程模型可能看起来像图 2.17 的简化模型。模型中框代表活动，从左指向框的箭头是资源，在框右边离开的箭头是输出。从顶端进入的箭头是控制或约束，诸如时间表，预算可标准。从下面进入的箭头是机械（mechanisms）用来协助活动，如工具、数据库或技术。

Ariane-4 应用过程开始于软件的任务（即控制一枚新火箭），此外还有来自以前机体的软件，未满足需要和别的来自期限他来源（如购买的软件或来自其他工程的重用仓储库）的可利用的软件组件。基于空间建立者的商业战略，开发者能确定可用子过程，描述它们（或许带上相对于过去经验的注释）。并把它们放入一个库中以便由需求分析员来考虑。可重用过程经常涉及可应用组件（例如，可应用需求，设计或代码组件，或者甚至是测试用例、过

程描述和别的文档和产品）

接着，需求分析员考查给新机体的需求和库中可利用的可重用组件。他们产生一个修正了的需求集合由新的和重用的需求的混和体组成，然后，设计员使用那些需求来设计组件来证明他们的设计是正确的，并且和设计的新部分及需求和新描述的系统的总目标一致。最后，已证实的组件被用来构建或改变软件及生成最终系统。象我们在后面章节看到的，那样的一个过程可能已防止了 Ariane-5 的坠毁。

2.7,2.8,2.9　　（略）

2.10 学期项目

对给 FCO 的贷款安排系统（Loan Arranger System）,现在还处于其开发过程的早期。你还没有系统的一个完全的需求集合，你还只是纵览系统功能以及有一个系统如何用来支持 FCO 的业务的感觉。概要中的术语对你不是熟悉的，因此你请顾客代表准备一个术语表。他们给你的描述如表 2.3

表 2.3Loan Arranger 的术语表

借入方（Borrower）：是从贷方 clender 那里得到钱的接收方。借入方可能联合接收贷款；也就是说，每笔贷款有多个借入方。每一借入方有相关名字和一个唯一借入方标识号。

借入方风险：与任何代入方相联系的风险因子是基于借入方的偿还历史。没有任何贷款负担的代入方被分配一个名义上的借入主风险因子：50。当借入方及等偿还时风险因子减小，但当借入方延期偿还或拖欠时，风险因子增加借入方风险用发下公式计算：

公式：

例如，一借入主可能三笔贷款。每一笔两年前进行，所有偿还及时完成。这笔贷款处于“好”级别已经两年了。第二笔，第三笔分别是 4 年 5 年了。二者均处于“好”级别。直到近期。因此，两笔“迟”级别贷款已经处于“迟”级别仅一年。因而，风险是

$$50-[10\times 2]+[20\times(1+1)]+[30\times 0]=70$$

最大风险值是 100，最小风险值是 1

包（Bundle）：一包是贷款的收集，这些代款联系起业被当作单一单位卖给一名投资方。与每一包相联系的是包中贷款总额，包中代款处于活动的时期（也就是说，对这一包，借入方仍在偿不贷款），涉及购买此包的风险许估，借入方偿还所有贷款时获得的利润。

包风险（Bundle risk）一个贷款包的风险是色中贷款的风险的加权均值，为了计算那笔贷款的加权平均值，假设每笔 Li 有本 Pi 和风险 Ri，加权平均那么就是 $\frac{\sum_{i=1}^{n} PiRi}{\sum_{i=1}^{n} Pi}$

扣除额（Discount）：是 FCO 愿意卖给一笔贷款给一名投资方的份格。其计算可按公式：

$$\text{扣除额}\left(\text{剩余本金}\right)\times\left[\left(\text{利息}\right)\times\left(0.2+\left(0.005\times\left(101-\left(\text{贷款风险}\right)\right)\right)\right)\right]$$

利率类型（Interest rate type）：一种利率或者是因定的或者是可调的。固定率贷款（称为 FM）

在抵押期间有相同的利率。一种可调利率贷款（称为 ARM）有一个每年可变的率，（基于美国财政部提供的政府指标）

投资方（Investor）：一名投资方是有兴趣从 FCO 购买一色贷款的人或机购。

投资请求（Investment request）：一名投资方作一份投资请求，指定投资将以之进行的最大程度风险，一包中所要求的最小利润，及包中贷款偿还的最大其限

贷方（Lender）贷方是一个向借方贷款的机构。一名贷方能有 0.1 或许多贷款。

贷方信息（Lender Information）：是从应用程序外部导入的描述性数数。……不能改变或删除，下列信息与每名贷方相关：贷方名称、贷方关系人、联系人电话、贷方唯一标识号。一旦加入了系统，贷方入口能编辑但不能删除。

贷出机构：贷方的同义词。见“贷方”

贷款（Loan）：一笔款是一个信息集合，描述了家诞贷款和借入方信息，以确定与贷款相关的信息。下列信息与每笔贷款相联系：贷款号，利率，利率类型。决定日期限借方原来从贷款借款日期，期（表达为年数），期（表达为年数），借入方贷方贷款类型（巨额或正常），和资产。一笔贷款必须有一个相联贷方和借入方。另外，一笔贷款与一个贷款风险和一个贷款状态关联，贷款分析员：贷款分析员是一 FCO 的专业雇员，被培训使用 Loan Arranger 系统来管理，和给贷款打包。贷款分析员对贷款和借贷的术语熟悉，但手头并没有用来评价单笔贷款收集的相关信息。

贷款风险（Loan risk）：每笔贷款与一个风险级别指示为一个 1 到 100 间的整数相联系。1 代表最低风险贷款；

也就是说，借入方不可能推迟或拖欠贷款。100 代表最高风险，就是说，几乎可以肯定借入方会拖欠贷款。

贷款状态（Loan Status）：一笔贷款能具有三种状态指派之一：好（good ）,推迟(late)或拖欠(defanlt)。一笔贷款处于“好”状态，如果借入方到当前已偿不字全部。

最后偿不字但不是在支付预定期。

最后的支付在预定日期的 10 天内还未收到

贷款类型（Loana type）：一笔贷款或是巨额低押（财产阶值>$275，000）或是正常抵押（财产价值<=$275，000）

抽资组合（Portfotio）：FCO 购买的贷款收集，用于包含于一包（bundle）中，Loan Arranger 维护的仓储库中容纳有投资组合（Portfolio）中所有贷款的信息。

图 2.17　新机体软件的应用过程模型

第 3 章 项目计划与管理

- 跟踪项目进展
- 个人与团体
- 效果与进度评估

- 使用项目计划处理模型

软件开发周期包括许多步骤。有些步骤反复出现直到系统完成，客户用户满意。但是在发放系统开发维护资金时，用户往往希望获悉这个工程项目将需要多少资金和时间。本章将讲述人力资源计划和软件开发项目管理的必要性。

3.1 进展跟踪

只有一个软件提供所要求的功能或所需要的服务时才被称为有用的软件。当一个用户使你开始讨论感性上的需求时，一个项目开始了。比如，一家大型的国内银行希望在你这里寻求帮助建立一个信息系统，允许银行客户得到他们的账户信息，跨越不同的地理位置。也许你和海洋生物学家签订一份合同希望建立一个系统联系他们的水生管理设备和行为统计分析数据。普通情形下客户有以下几个问题需要解答：

- 你理解我的问题和我的需要吗？
- 你能设计一个系统，它能够解决我的问题或能够满足我的需要？
- 开发这样的一个系统需要多久？
- 开发这样的系统需要多少资金？

回答后面的两个问题需要一份周密详细的进度安排表。项目进度描述某个项目的软件开发的整个生命，通过枚举该项目的每个阶段，江每个阶段分解成离散的任务。进度也描述了任务间的相互关系，评估时间。因此进度表实际上是个时间表，显示了任务的开始和结束的时间，以及相关干开发的产品准备完备时间。

第一章我们已经学到系统分析方法包括分解和合成：将一个问题分解成若干组成部分，给每部分设计解决方案，最后将这些组合成完整的解决方案。这是一种确定进度表的方法。首先和客户、用户共同探讨理解他们的需求。列出客户所需要的所有要提交的产品，也即他们期望了解的软件开发的信息。这些信息包括：

- 文档
- 功能说明
- 子系统说明
- 正确度说明
- 可靠性、安全性和性能说明

接下来，我们要确定为了完成这些要提交的信息哪些工作是必须做的。我们可以使用第二章学到的模型处理方法，列出所有会发生的事情和相关任务，用到的产品或资源。某些事件被指定成里程碑，用它们说明一个阶段的进步。

在分析项目时，我们必须明确的分清里程碑和工作任务的差别。工作任务是一个项目在某个阶段的部分工作，而里程碑则指一段任务的结束------一个特殊的时间点。工作任务是一个过程，有开始有结束；里程碑则是一个某指定工作的结束时间点。

用这种方法仔细检查项目，我们能将开发的项目分成一系列子阶段。每个阶段包含若干步骤，每个步骤如有必要可进一步划分，如图 3.1 所示。

考虑这个房屋建筑项目的阶段、步骤、工作任务表。首先，我们考虑两个阶段：环境设计和盖房。然后将两个阶段花分成若干步骤，比如，清扫，挖掘，植草皮，种树，剪枝。如有必要可将这些工作进一步划分。比如，内部装修还可分成：管道设计，电器设备设计，壁纸，室内装饰图，地板铺设，门窗设计和家具设计。每个工作都是一定量评价的。
这些分析可以给用户一种想法：建造一幢房子包括哪些工作。同样，分析一个软件开发系统和工程维护同样用到这样的分析方法。

任务图

任务图用来描述一个项目的工作任务结构。在这个图中里程碑和任务都用于描述整个项目进展。用箭头描述任务流向，开发者可遵循这些箭头，明确下一项工作任务，遇到里程碑，则知道此部分工作结束了。但这种图不能表现工作之间的相互依赖关系，已不能体现那些可以同步执行的项目任务。

我们可以用四个参数描述一项工作任务：前辈，持续时间，截止日期和终点。前辈指某个任务在执行前必须执行的任务，他表示了一个任务可以被执行的前提条件。持续时间则指完成这个任务所需的时间长度。截止日期表示任务应该在哪个时间点之前完成，这个日期一般有双方的合同确定。终点则是一个里程碑，表示可交付客户。

一个项目的许多可视化特征都可以体现在任务图中。比如从图 3.2 一个项目的许多可视化特征都可以体现在任务图中。比如从图 3.2 在里程碑 2.2 没经实现前，管道工作则不能开始实施。此外从这个图中我们可以看到哪些工作是可以同时进行的。比如室内装修和室外装修是独立的两部分工作，在左路径上的任务初始工作时间不依赖于右路径的工作，因此可以同时进行。注意结点“需求通过”1.2 与“测量”1.1 之间的破折号，它表示这两个任务必须在挖掘活动（指向里程碑 1.3 的活动）开始之前进行。但是从里程碑 1.2 到里程碑 1.1 没有实际的任务，因而这个破折号说明任务之间没有伴随关系。

需要认识到任务图是建立在对任务并行特征的理解基础上。如果所作的工作不能并行进行，那么任务图在描述工作之间的相互协调关系时所起的作用不大。此外任务图必须如实反映工作之间的并行性。在我们这个例子中，某些任务，比如管道工作可以有承担其它工种的人来完成，比如电器工程人员。但在软件开发项目中，许多人掌握多种技巧，然而理论上的并行性却不能反映现实。同样工作组的人员往往要参与许多工作。

任务结束评价

在每项任务上附加结束评价信息，可使任务图更加有效。例如对于表 2.1 的第二个阶段，可以加在图 3.2 种工作天数参数。这样每个任务的可能结束的时间有了衡量标准。
这样的结果体现在图 3.3。可以注意到里程碑 2.7，2.8，3.4，3.6 都是终点的前辈。也即，在项目完成之前，这些里程碑必须实现。节点链接符号上的 0 表示结束之前这些任务无需额外时间。

这种项目的图表描述可以告诉我们许多项目进度的信息。比如，由于我们初步估计第一个任务可能需要 3 天完成，那么就不能希望在第三天结束时遇到里程碑 1.1。同样也不能在第 15 天结束时遇到里程碑 1.2。这主要是由于挖掘工作的开始必须在里程碑 1.1 和 1.2 之后，直到第 16 天挖掘工作才可以开始。

分析任务图中的里程碑路径被称为关键路径法（CPM）。这条路经可以表示完成一个项目所需的最短时间，每个任务所需的持续时间。CPM 还能表明哪些任务对于一个项目而言按时完成是非常关键的。

再来考虑我们的盖房实例，首先，我们可以注意到那些到达里程碑 1.1 土地测量和里程碑 1.2 需求允许的任务。由于到达里程碑的 1.3 的挖掘工作直到 16 天后才可以开始，土地测量则需要 15 天后才能开始，尽管它只用 3 天就可以完成。同样，对于活动图中的每个任务，我们都可以计算它们的两个时间参数：实际时间和实际消耗时间。可以使用时间则指该任务在项目完成过程中可能分配的进度时间。

松懈时间则指可得到时间与可能时间的差，即

松懈时间 = 可以使用时间 — 实际时间

另一种衡量松懈时间的方法是比较一个任务的最初开始时刻与最末开始时刻在没有延迟情况下的时间差。

松懈时间 = 最末开始时间 — 最初开始时间

从上述分析我们可以知道，最长的路径对于它的每个节点都有零延迟，主要因为这条路经可以确定该项目能否被完成。出于这种原因，这条路经被称作关键路径。因此在关键路径上，每个节点的松懈时间均为零。从本章的实例中可以看出，关键路径不唯一。正是由于关键路径的零延迟特征，执行该路径上的任务时不允许失误出现。

考虑关键路径上的任务如果开始出现开始延迟，那么这个延迟将推后所有的该路径上的任务开始时刻，导致整体进度后移。而对于非关键路径上的任务，他们可能丢掉松懈时间。通过者章图我们可以了解到进度滑移。

如果在进度图中出现循环会怎样？但任务需要重复时，循环就会出现。比如在我们的房屋建造例子中，设计监察人员可能要求管道工返工他们的工作。在软件开发过程中，设计检查可能要求从新细致的改动。这些循环的出现使得关键路径出现往复。在此种情况下，进度评估难度增加。

图 3.4 时一个软件开发的工作任务分栏图表，该表中包含最早开始日期，最晚开始日期，这种分栏任务图是由项目管理工具自动生成的。水平栏表示活动的持续时间，包含星号的栏表示关键路径。

项目进展的关键路径分析可以表示出那些任务要等待当前项目进展经过的任务，以及那些任务为避免延迟必须在指定时间内完成。我们可从不同方面提高图表的分析效果。比如在房屋建造过程中，假设我们知道每项任务需要的时间长度，尽管事实上并非如此，但可以根据以往的经验和相似的项目给出估计时间。因此可根据概率分布给每项任务一个持续时间。因而每项任务就有了一个期望值和方差，也即我们不必知道准确的持续时间，只要分析在一个时间窗内，或一段时间内，实际所耗时间下落的概率，在这里期望值是一段时间内的点，方差则描述时间段的宽度。你可能对标准的正态概率分布非常熟悉，其分布曲线呈钟形。程序评估与审查技术（PERT）是一种常用的关键路径分析技术，以正态概率密度分布为前提。（参看 Hiller 和 Lieberman[1967]有关 PERT 的内容）PERT 可以确定一项任务最早起始时间距离进度要求时间的概率。使用概率分布知识，最早最晚起始时间，任务图和 PERT 可以计算关键路径以及可能造成瓶颈的任务。许多项目经理使用 PERT 和 CPM 方法检查他们的项目。但是这种方法只在由若干个任务可以同时进行的稳态项目中有价值。当项目多数任务为序列顺序时，那么几乎所有的任务都在关键路径上，他们都可能成为瓶颈。此外如果一个项目需要多次需求改进，或重新设计，那么任务图和关键路径在开发过程中往往有较大的改动。

跟踪进展的工具

有许多工具都可用来跟踪项目进展状况。这些工具可能是手工的，电子制表软件还可以是一些复杂的图形界面工具。可以根据图 3.5 所描述的工作分解图确定哪些工具对项目的跟踪有益。在这里总的目标是创建一个系统包含通讯软件，项目经理需要描述工作的五个步骤：系统计划，系统设计，代码编写，测试和提交。为简单起见，我们着重考虑前两个步骤。步骤一又被分成四个部分：考察规范，分析预算，分析进度和开发项目计划。同样系统设计也可分为：顶层设计，原型设计，用户界面接口设计和详细设计。

许多项目管理软件系统提供工作分解功能，帮助项目经理跟踪步骤任务的进展。例如项目经理可以画一个甘特（Gantt chart）图表，描述项目任务，用平行方式表示，用颜色或特殊标记表示任务完成情况。这样的图表可帮助经理理解哪些任务可以同时进行，哪些可以并行进行。

图 3.6 时一个甘特图，描述图 3.5 的工作分解结构。该项目从一月份开始，标有“today”的垂直虚线表示当前项目进展到的时间，正值五月。垂直栏表示每个任务进展情况，用三种颜色跟别表示完成，正在持续和关键时刻。钻石符号表示那里有进度滑移，三角表示活动开始或结束。干特图比图 3.4 的 CPM 图要包含更多的信息。

在本章后面，我们将见到如何评估开发代价。项目管理软件工具可以跟踪实际代价，用于比较估计代价，有了这些帮助，整个进展都可得到恰当的评估。

3.2 项目人员

为了确定项目安排和与费用和努力有关的项目估算，我们必须了解大概有多少人参加这个项目，还有他们要执行的任务是什么，他们的能力和经验如何。在这一节中，我们要看看怎样决定人员的组织。

员工角色和特征

在第二章中，我们审查了几种软件过程模型，每一个描述软件开发的几个活动联系的方式。无论模型怎样，对于任何的软件项目总是有其必须的活动。比如，每一个项目要求与用户的交互，来确定用户要什么。其他项目人员设计系统，还有一些编写程序和测试程序。主要的项目活动可能包括：

1. 需求分析
2. 系统设计
3. 程序设计
4. 程序实现
5. 测试
6. 运行
7. 维护
8. 质量保证

然而，不是每一个任务都是由同一个人或同一个组完成的；任务的人员分配取决于项目的大小，人员的专业化程度和人员的经验。将不同的任务分配给不同的人群有很多优点，提出“检查和平衡”可以在开发过程中尽早地发现故障。比如，假设负责测试任务的和负责设计系统与编码的人员分开。测试新的软件或修改过的软件设计一个系统测试，就是开发人员要向用户证明系统是如规格说明中的一样。测试人员必须定义和书写测试进行的方式，还有连接需求规格中的被证明的功能和执行特征的标准。测试人员要在不知道系统内部是怎样的情况下，得出一个测试计划。因为测试人员没有关于软件和硬件如何工作的事先的看法，它将注意力集中在系统的功能上。这个方法使得测试人员更容易不活错误，和由设计者和程序员造成的漏洞。部分由于这个原因，在 Cleanroom 方法中用一个独立的组进行测试工作。

基于同样的原因，将程序设计人员和系统设计人员发开业很有用。程序设计者深入地涉及代码的细节，有时候忽视了系统应该怎样工作的大的描绘。我们在后面的章节中会看到像走查，审查和复查这样的方法，可以在实现编码之前为两类设计者带来双重的检查，还有在开发过程中提供连续性。

我们在第一章中看到在开发或维护的过程中有很多其他角色的人员。在以后的介绍开发的主要任务时，我们会描述项目组的成员都完成哪些工作。

一旦我们决定了项目组成员的角色时，我们必须确定每一种角色我们需要什么类型的人。项目人员在很多方面可能不同，而且不能确定地就说一个项目需要一个分析员，两个设

计员，五个程序员。两个有相同头衔的人也可能不同，至少在下面这些方面之一：

- 完成工作的能力
- 对工作的兴趣
- 关于类似应用的经验
- 关于类似的工具或语言的经验
- 关于类似方法的经验
- 关于类似开发环境的经验
- 培训
- 他人交流的能力
- 与他人共同承担责任的能力
- 管理技巧

这里的每一个因素都会影响一个人出色地完成工作的能力。这些也可以帮助解释为什么一个程序员在一天内就可以写出特定的例程，而不是别人要求的一周呢。这些差别不仅对计划评估，还对整个项目的成功起关键作用。

为了了解每个人员的表现，我们必须知道他们完成手头的工作的能力。有的善于纵观全局，而不擅长追究细节问题。这样的人适合系统设计或测试，不擅长程序设计和编码。如果人们对自己的能力有信心的话，那么完成起来会更高效。

对工作的兴趣也可以决定一个人的成功与否。虽然一个人可能擅长做某事，但是它可能更有兴趣作一些新鲜的事儿，而不是重复以前的工作。这样工作的新颖有时候是产生兴趣的一个因素。但是有的人更愿意做他们熟知的事情，而不愿意冒险尝试。重要的是谁被选择完成某项工作，而不是什么原因。

假定两个人能力兴趣相同，也可能在类似地应用、工具或是方法上的经验和培训不同。也即项目人员的选择不仅要看个人的能力和技术，还要看他的经验。

每一个软件开发或维护项目中，开发人员都要与其他交流，比如用户。项目的进展不仅会受交流程度的影响，还会受与他人交流的人的能力的影响。软件的失败可能是由交流和理解中的失误造成的，因此，和他人交流的人数会影响最终的产品质量。图 3.9 说明了交流线是怎样增长的。人员由两个增长到三个的时候，可能的交流的线路数增长了两倍。通常，如果一个项目有 n 个人的话，那么有 n(n-1)/2 对人可能需要交流，有 2^n-1 可能的组可以被建立进行更小项目的开发。这样，一个项目涉及 10 个人的话，就有 45 条交流线路可用，有 1023 个可能的人员组合去处理子系统的开发！

很多项目涉及几个必须共同负责完成一个或更多活动的人员。他们之间必须互相信任。在共同承担工作的时候，不仅要互相进行口头交流，而且要将你的想法和所作的工作写成文档。很多人在这方面上都有困难。

控制是管理项目中的一个问题。有些人擅长指挥其他人的工作。那些不能使它的伙伴感到舒服的人并不是开发工作中管理人员的好的候选者。

这样，人员的背景的几个方面能够影响到项目组的质量。一个项目管理人员应该了解每一个人的喜好和能力。Sidebar3.1 解释了怎样组织他们可以增强或妨碍项目进度。正如我们会在后面的一节中看到的一样，雇员的背景和交流也能够对项目的费用和进度起到影响。

Sidebar3.1 开会可以增快项目进度

软件项目的很多交流是发生在会议上的，或是电视会议。然而，会议会浪费很多时间，而又没有完成那么多的任务。它浪费人力和财力，却没有收到良好的效果。有关会议通常的抱怨是：

- 会议的目的不明确。
- 出席者没有准备好。
- 必要的人物缺席或是迟到。
- 谈话总是跑题。
- 一些会议的参加者不讨论实在的问题。
- 会议的决定在会后没有颁布。

好的项目管理涉及计划所有的软件开发活动，包括会议。这里有几种方法可以保证会议质量。首先，管理者必须明确谁要出席会议，什么时候开始和结束，会议要完成什么。其次，会议要有议程，可能的话，提前发放。第三，有专人负责进行讨论和解决冲突。第四，有专人负责会议上的决定能够被确实执行。最重要的是，最小化开会的次数，和出席会议的人员数。

工作风格

不同的人有不同的与人交流和理解问题的喜好，在工作的过程中。比如，你可能愿意在做出决定前将所有可能的信息都分析一遍，而你的同伴可能依赖于感觉做出重要的决定。你可以根据两个部分考虑你的风格：你的思想交流和想法组织的方式，和你用感情决定问题的程度。在交流思想的时候，有的人喜欢告诉别人它的想法，而有的人在提出观点前喜欢先征求别人的意见。Jung(1959)称前者为性格外向者，后者为性格内向者。显然，你的交流风格影响了你和项目中其他人交流的方式。类似地，感性的人作决定时，基于他们的情感；而理性的人作决定时主要依赖事实和仔细考虑所有的方面。

我们通过图 3.10 来描述各种各样的工作风格。共分为四种：感性内向的人，感性外向的人，理性内向的人，理性外向的人。

感性内向的人：作出决定前先搜集足够的信息，并且作决定的时候基于对他所了解的东西的感觉来做出判断。感性外向的人：作决定时有很多的感情因素，愿意告诉别人他们的想法，而且感性的特点使他们经常提出一些不是常规的解决问题的方法。理性内向的人：作决定时候要考虑所有可能的方面，并且搜集所有的信息。理性外向者：作决定的时候不会让感情影响他们，而且也很少在做之前询问更多的信息。

图 3.10 说明了他们的倾向和喜好。

交流是项目成功的关键，工作风格决定了交流的风格。理解工作风格可以帮助你在和其他开发人员或用户的交流时更灵活。特别地，工作风格可以给你一些有关他人兴趣的信息。比如，假设 Claude 是你的客户，而你为他准备了一个有关项目状况的报告。如果 Claude 是一个内向的人，那么你知道他愿意搜集信息。这样，你可以组织你的演讲，来告诉它大量的有关项目的结构和进展情况。如果 Claude 是一个外向的人，那么你可以允许他提出有关他的需要的问题。这样，工作风格直接决定了客户，开发者和用户之间的交流。

工作风格还涉及一个特定任务的人员的选择问题。比如，理性的人可能喜欢设计和编程维护的开发（需要新的思想）。

项目组织

软件开发与维护项目组不包含彼此互相独立的或没有任何合作关系的人员组织。成员之间都用一种可以提高产品质量的方式组织在一起。选择哪中项目人力资源组织方式取决于以下几方面因素：

- 小组成员的背景和工作风格
- 小组成员数
- 客户和开发者的管理风格

优秀的项目管理者应该清醒的认识到这些问题，而且应该搜寻那些能够和所有参与者和睦共处的小组成员。

一种广为人们喜爱的组织方式是首席编程团队，第一次在 IBM 使用。在一个首席编程团队中，由一名成员完全负责整个系统的设计和开发。所有的其他成员向总程序开发者汇报情况，该负责人有最终的决策权。首席程序员指导所有其他成员，设计所有的编程工作分配代码任务给所有的其他成员。首席助理是一个替角工作，在必要情况下代替首席。图书管理员辅助整个团队，负责维护所有的项目文档。该项工作还包括遍及和连接代码，执行豫交付给信息管理库的所有模式的测试。这种分工科室程序员集中精力做编程工作。

这种首次编程团队组织可用图 3.11 说明。把决策的责任权加在首席编程人员身上，整个团队将项目中必要的交流时间缩短到最小。每个小组成员都必须经常和首席交流，但组间成员没有必要这样做。当一个团队包括 n-1 个小组成员，和 1 个首席时，整个团队只有 n-1 个交流通道，而不是 n(n-1)/2 条通道。比如在解决一个问题时，小组成员可以简单的和首席交流，而不是一个人孤立的去做这件事。

尽管这种团队具有层次结构，可以形成工作组来完成某些特殊任务。比如一个或多个组成员可以成立一个执行小组负责项目当前代价和进度的状态报告。

简言之，这种首席编程团队比须能够快速做出决策，因而首席从性格上讲必须是外向型的。但是如果组成员多数是内向型性格，那么这种层次结构就不是最佳的。另一种替换策略基于无私编程想法，Weinberg(1971)给出这种思想的描述。取而代之这种单一责任制，“无私”妨时实每个人都担负相同的责任。此外所有的工作在这些组员间划分，工作结果由最终的产品质量表现。

还有许多其它人力资源组织方法，前面列举的两种情况都属极端。那种组织结构更好呢？一个项目所需的人越多，则越需要一个正式组织形式。显然一个只有三四个人的团队不一定需要详细复杂的组织。但是对于由几十个人组成的团队必须有任务明确的组织形式。事实上，你的客户往往从过去成功的经验出发，要求你必须有一定的组织形式。比如你的客户要求你的团队组织独立于程序实际和开发。研究人员也在探索项目团队结构是如何影响产品的最终结果，如何在一定条件下选择最合适的组织。国家研究基金会（National Science Foundation 1983）调查发现具有较高成都的确定性，稳定性，统一性和重复性的组织可以通过层次组织结构，比如首席编程团队，实现更高的效率。这些项目对项目成员间的交流需求小，因而他非常适合那些强调规则，规范，手续合组织层次的清楚定义。

从另一方面看，当一个项目包括许多不确定因素时，一个民主的办法则显得更好。比如，当在开发过程进行时需求发生改变，那么这个项目具有一定的不确定性。同样地，假如你的客户正在制作一个新的硬件，想成为系统的接口，而且该硬件相关的规范都未知，那么这个项目的不确定性就特别高。那么参与决策就使预先定义的层次结构界限变的模糊。那么开放的交流就非常必要。

表 3.5 总结了项目特征和针对这些特征的组织结构。

两种组织结构可以联合在一起。比如程序员可以根据自己的情况设计一个子系统，在同一层次之间可采纳无私模式，或一个较为松散的团队结构可以强加上层次结构。或相反，可以在无私模式上强化一个人的责任权限。

3.3 效率分析

项目计划和管理的一个重要方面是理解一个项目可能的花销。所预计的花销过大可能使用户取消这个项目的建设，花销预计太低可能使整个团队的人员停下来等待资金到位。正如图 3.3 所描述的那样，有许多导致花销预算的估计错误。在项目生命周期内早期的准确的花销预算可帮助项目经理确知究竟有多少开发工作必须的，合理安排全体成员的工作。

项目预算涉及以下几方面开销：公用设备，人员开销，工具开销。公用设备开销包括：硬件，办公家具，电话，调制解调器，取暖设备和空调，网线电缆，磁盘，办公用纸，文具，影印机以及其他所有开发过程所用的物理设备。对于一些项目，所需环境可能已经存在，那么所需费用容易估计。但对于其他项目所需的物理环境需要从头创建，比如一个新的项目，可能需要一个安全保护的棚顶，增加了高度的地板，温度和湿度的控制器或者特殊的设备。在这里费用可以精确的估计出来，但他们可能和最初的估计有较大的差别。

对于经理和开发者还有许多隐性费用不易发掘。比如研究小组需要一个安静的、足够大的个人空间，才能有效工作。McCue（1978）提出在 IBM 公司内，编程工作空间至少是 100 平方英尺的地面面积和 30 平方英尺的水平工作面积。这些空间还需一个从地板到天棚的隔墙，保证没有噪音干扰。

项目费用还包括软件购买费用，系统支持工具费用。此外设计工具，编码环境和项目本身可能也需要购买一些软件来满足需求，组织文档，测试代码，跟踪动态的改变，产生测试数据和支持小组会议等消费项目。这些工具有时被称为计算机辅助软件工程（CASE），一般是客户或公司标准软件开发过程的一部分。

对于多数项目而言，最大的费用在于工作计划效率。我们必须确定为了完成一个项目所需工作日。工作效率具有最大的不确定性。我们可以看到工作风格，项目管理，能力，兴趣，经验，培训和雇员特征对于完成一项任务都有非常大的影响。此外当工作组必须互相交流，向顾问咨询时，则需要额外的开会培训时间。

费用，进度和工作效率评估在项目生命周期内必须尽早进行。主要因为他们影响资源的分配和项目可行性分析。但是评估工作也需要在整个生命周期内反复进行。

图中的星号表示从实际项目中获得的尺寸估计和费用估计。漏斗型线指向右侧表示随着我们更多了解一个项目时我们先前估算的准确程度。当项目的规范还没有完全确定下来时，估计费用和实际费用差值是 4 的倍数。随着决策制订，许多规范细节完善下来，这个因子将减少。许多专家努力使他们的估计误差控制在 10%内，但是 Boenm 的数据表明只有当整个项目接近尾声时，这种小误差的估计才存在。

为了强调估计准确性的重要，软件工程时已经开发出许多技术用来确定工作效率，员工性格特征，项目需求以及其他因素之间的关系，还有许多影响时间，效率和开发费用的因素。

专家评论

许多效率评估方法都以专家评论为基础。某些评论是非正式的技术，给予管理者对相似项目的经验。因此，预测准确性以能力，经验，客观情况和估计人的感知为基础。最为简单的形式就是的对建立整个系统或子系统所需工作的估计。完整的估计可以是从顶向下或从底向上。

在许多情形下，我们都可以用相似行衡量费用评估。例如，如果我们已经建立了一个非常相似系统，那么我们可以在这个已有的系统基础上评价新系统。例如，A 系统和 B 系统非常相似，那么 A 系统的费用就和 B 系统的费用非常接近。可以进一步讲，如果 A 系统的复杂程度是 B 的二倍，则 A 系统的费用是 B 系统费用的两倍。

可以通过向专家咨询三个预测：悲观因素 x，乐观因素 y 和最可能的猜测 z，给出相似处理的形式化描述。我们的估计是 β 概率分布的均值，由（x+4y+z）/6。通过使用这种技术，我们可以把个人评价正太标准化。

Delphi 技术用一种不同的方法利用专家评论。专家被要求根据他们的经验，他们所采用的处理方式给出个人的保密建议。每个专家都有机会审视他们的估计

Wolverton(1974)建立了软件效率模型。他的软件费用矩阵使用了他在 TRW 软件开发公司的项目费用。如表 3.6 表示，行名称代表软件类型，列名称代表难度。难度取决于两个参数：该问题是否属于旧问题（O），或新问题（N），容易的问题（E），难度适当的（M），难的（H）。矩阵的每个元素代表每行代码的代价，由在 TRW 项目中的历史数据为参考。在使用该软件前，首先将提出的软件系统分解成若干模块。然后估计每个模块可能包含的代码行数。使用矩阵，计算每个模块的代价，最后求和。例如有一个系统包含三个模块：一个输入/输出模块，旧问题而且简单，一个算法模块，新且难，一个数据管理模式，旧而且适中。那么所有这些模块在有 100,200 到 100 行代码情况下，则 Wolverton 模型估计所需代价为（100×17）+（200×35）+（100×31）= $11,800。

由于这个模型基于 TRW 而且使用了 1974$，因而对于今天的软件开发不太适用。但是这种技术可以灵活的移植到你自己的开发项目中。

总之，经验模型，依据过去的开发经验，易于产生较多的不准确估算。他们依据专家对过去相似项目的经验产生结论。然而即使非常相似的项目，他们的代价也相去甚远。
即使我们知道他们差别所在，也不能确定那些影响费用的差别。比率策略也是不可靠的，因为费用不是线型关系。两个程序员不可能完全一样的编码速度，可能需要额外的时间用来交流，合作或容纳对方的兴趣，能力经验。

专家经验不仅受可变性影响还受主观影响，以及所依赖的现行数据影响。这些专家评论所依据的数据必须反映现在的实际情况，那么它们必须及时更新。

算法模型

研究人员已经创建了一个模型，表示效率和影响效率的因素之间的关系。该模型用方程表示，在这里效率是自变量，其它的几个因素（比如：经验，规模和应用类型）是因变量。这些模型多数承认项目规模是方程中最具影响的因素。

这里S是系统的规模后记值，a,b,c是常数，X是费用因子级、组成的向量，从x_1到x_n，m是基于这些因子的调整乘数，也即效率主要由该系统的规模大小确定，受其它因子影响，这些因子涉及别的其它项目，过程，产品，和资源等。

Walston 和 Felix（1977）开发出了这样类型的一个系统，发现 IBM 公司从 60 多个项目中获得数据满足下列形式的方程。

提供这些数据的项目建立了一个规模从 4000 行代码到 467,000 行代码的系统，用 28 个不同的高级语言，使用 66 台计算机，耗去从 12 到 11,758 人月的效率。规模用代码行数衡量，包括代码注解。

这个基本的等式又由受 29 个因子影响的生产力索引做了补充，由表 3.7 表示。可以注意到这些因素都紧紧的连系在某个特定的开发类型，包括 2 个平台：操作有计算机和开发用计算机。这个模型反应了 IBM 联合，系统组织 20 特殊开发风格。

每个因素都经过衡量，标上权值 1 表示该因子可提高生产力，0 则表示不能提高生产力，-1 表示降低生产力。

Bailey 和 Basili（1981）提出了一种建模技术，称为元模型，用来建立一个评估方柱反应你自己的组织特征。他们用一包含 18 个科学研究项目的数据库描述他们的技术，这些项目代码均用 Fortran 编写，产生于 NASA 的 Goddard 空间飞行中心。首先它们最小化了标准估算的错误，产生一个较为精确的等式。

根据错误的比例调整初始估论值。如果 R，是实际效果 E 和预计效果 RE ´ 间的比例，则调整效果被定义为

$$ER_{adj}=$$

然后调整 E

最后，Bailey 和 Basili（1981）还考虑了其他影响效果的因素，如图表 3.8 所示。对于不中的每一项，该项目分值标记从 0 到 5，这些值依赖于项目经理的判断。因此整个 METH 的分值可高达 45，CPL×可达 35，EXP 高达 25。他们的模型描述了多线性最小平方回归的过程。

显然，这种类型模型的一个问题在于规模是一个关健变量。评估工作要于有关规模的准确信息给出了前进行，而且这种评估也是在代码行数给出之前进行的。因此该模型反效率问题转化成规模问题。Boehm 的建设性代价模型承认了这个问题的存在，并将了种规模评估技术纳入他们的最后版本，COCOMOⅡ中。

Boehm 在 70 年代开发了 COCOMO 的原型系统，他们利用 TRW 项目中的信息创建一个扩展数据库。由于软件设计要考虑工程因素和经济条件，Boehm 将规模当作主要的决定因子，再用其它的代价影响因子 调节估计值。90 年代，Boehm 更新了原有的 COCOMO 模型，创建了 COCOMOⅡ，标志了软件开发日趋成熟。

COCOMOⅡ评估处理所采用方法反应了软件开发的三个阶段。最初的 COCOMO 模型将提交的源代码行数作为关健输入的模型认为在开发周期早期代码的行数不可能为人所知。首先，项目往往建立一个原型，用来解决一些高风险问题，可能包括用户接口，软件，系统交互，性能。经二步，为了进一步开发我们需要制定一个决策，但是决策制定人必须明确系统体系结构，操作运用行方面的概念。这些信息又难的满足细粒度划分的持续时间分析，但这些信息确比第一步提供的内容要多。

第三部系统开发，在这一阶段系统规模才能确定下来，规模大小用代码行数或功能点数描写。

COCOMⅡ还考虑了模型重用性，维护和破损（比如随时间推移需求发生改变），利用原妨的 COCOM，该模型包括费用因子用于调整初始费用估算。因为 COCOMⅡ是一个新的模型，还没关于它的准确性数据分析。

下面我们详细地讲解一下 COCOMⅡ。基本模型有如下形式：

$$E=bs^{c}m(x)$$

Bs^{c}是初始基于规模的分析，由代价向量调整。

第一步，应用点提供规模测量标准。这种规模平测是 Kawffman 和 Kamar（1993）提出的对象一点方法的扩展。为了计算应用点，首先计算荧幕数，报告数，第三代语言组件数。假定所有元素定义为集成的 CASE 环境下一部分的标准方法。下一步你可以将每种应用元素分类成简单，中等或难。表 3.10 给出分类指导。

用于描述应用点的个数是一个综合权值。这个值反应了需要实现一个屏报告所需的相对难度。

然后将这些权值求和。如果 r%的对象可重用，则新的应用数可计算为

新应用点数=（应用点数）×（100+r）/%

用这个数评估效率，你可以用一个调整参数。称为生产雍率比值，基于开发者经验和能力，并参考 CASE。比如，如果开发者的经验少，但 CASE 成熟度非常高，那么生产率是两

个值的平均值，16。同样当队伍中每个开发者的经验互不相同时，则可以用加权求均恒方法计算生产率。

在第一阶段费用影响因子对效率估记起不了什么作用，但是在第 2 阶段效率估记基于功能点数计算结果，用重用程度调整，需求变化验室和维护代价。在第一阶段规模设为 1.0，第 2 阶段规模从 0.91 到 1.23。这些数据依赖于系统的新旧程度，一致性大小，早期体系统结构和风险解决能力，团队合作情况和处理手法成熟否。

第 2 阶段和第 3 阶段的影响因子均属于调整因子，用作效率乘数，其范围从“极低”到“极高”比如，一个开发团队的某种类型应用的熟悉程度可以从以下几个方面考虑：

- 极低　　经验不足 3 个月
- 非常低　　至少 3 月但不到 5 个月
- 低　　至少 5 月但不到 9 个月
- 普通　　至少 9 个月但不到 1 年
- 高　　至少 1 年但不到 2 年
- 非常高　　至少 2 年但不到 4 年
- 极高　　至少 4 年经验

同样，分析者的能力也可根据百分大的顺序级别衡量。比如，比例“非常高”指分析者在 19%，普通则指 15%—50%。相应地 COCOMⅡ给“低”分配小 42，“高”则为 0.71。这些乘数反应了一个概念，能力比较低的分析者比平均分析者高 1.42 倍。表 3.13 给出了费用影响因子范畴表，示数从 1.17（非常低）到 0.78（非常高）。

可注意到 COCOM 的不同组成部分在应用于你自己的组织时需要针对项目特征加的改变，量体裁衣。在本章后面的章节里，我们将应用机器学习方法。

在过去大多数效率和费用模型技术都以算法方法为基础。也即，研究人员检查过天一项目中的数据，从中得出许多数学等式，然而一些学者求助于机器学习指导产生良好的评测结果。比如，神经网络可用来表示许多互相连接，互相依赖的单元，因此他们是一种能够表示软件开发不同活动之间的有利工具。在一个神经网络中每个单元（称为神经元，是网络中的一个结点）代表一个任务，每任务有输入和输出。每个单元都有相应数值和输入相关，可用来计算权值。如果该权值和超出某个阀值，则该单元产生输出结果。这个输出同时成为别的单元的输入，直到整个网络的最后输出产生。神经网络是活动图方法的一种扩展。

神经网络有多种办法产生输出结果。一些技术包括回看其它结点产生的结果，称为向后传播技术。这个技术和我们在活动图中使用的回溯技术确定路径懈怡相类似。其它们的一些技术采用前向法，等待即将发生的事情。

神经网络是通过培训方法培养起来的。训练所用数据来自以往的项目，网络用前向和后向算法学习识别数据中具有的模式。比如关于过去项目的历史数据一定包括开发者的经验中经网络可能识别不同级别经验之间的关系，完成一个项目的效率，工作数量。

图 3.13 说明了 shipperd（1997）如何利用神经网络产生效果估计。网络有三层，没有循环。四种输入为影响项目的因子，网络使用这些输入做为单一输出。最初网络中每个单元赋给随机初始值，然后将从历史中得到训练结果作为新的权值给单元。模型的用户指定一种训练算法，该算法规定了训练数据的使用，此种产生于历史的算法一般包括后向传播。经过训练的网络，可用来估算新项目的效率。

几位研究人员曾使用后向传播方法研究相似的神经网络，用其预测开发效率，开发软件可能使用第四代语言。这个研究表明（shepperd1997）该类型模型准确性和神经网络术语认误用相关，此外还和学习阶段，初始权值有关。为了获得良好的预测值，神经络需要许多训练集合。也即他们必须基于大量经验数据，而不是若干代表性项目。这种数据往往来之不易。尤其是持续的得到大量数据。因此数据的缺少限制了这种技术的推广。此外用户对神经网络

理解上存在困难。查是如果这种技术产生更多准确切的估计，那么开发团体是乐于为它收集大量数据的。

总之，这种“学习”法已经从不同方面被人们尝试了。

一种机器学习技术称作基于案例推理可用于相似案例评估。被有于人工智能领域的CBR 根据输入的组合建立了一个决策算法。同其它技术一样，CBR 的 2 个明显的特点。第一，CBR 只处理那些实际发生的事件，而不是众的可能事件。第二理解 CBR 对于用户而言要比神经网络容易得多。

用 CBR 评估需要 4 个步骤：

①用户指定一个新问题为一个案例

②系统从历史信息中选取一个相似案例

③系统从先前案例中重新使用知识

④对于这次新案例给出解决方法

但在创建 CBR 过程中有两个障碍：案例特征化和相似性确定。

案例特征由可得到信息得来。一般，请教专家，从而获得所描述案例的显著特证，尤其在案例相似时，确定他们的相似性。实际上，相似性可用一个代表几个特征的一个向量表示。结合你自己的实际情况选取模型

今天有许多评估模型可供使用：多种多样的基于经验或复杂开发模型的商用工具，以及那些基于历史数据的自行研制工具。模型正确性证明是一项艰巨工作。此外如果说一个模型适合较大而且不同数据集时，那行支持立的数据库也需要这种环境。

即使找到好几个适合自己开发系统的模型，你也必须学会评价哪个模型最精确。有两种统计方法可以帮助你评估准确性，PRED 和 MMRE。PRED（x/00）是估计值在实际值 x% 内的项目所占比例，MMRE 是平均相对错误程度。因此我们希望 MMRE 对于给定模型而言越小越好。一般研究人员认为 0.25 的 MMRE 相当，Boehm（1981）则建议 MMRE 应该是 0.1 或更小。表 3.14 列出了文献里各种模型的最佳 PRED 和 MMRE 值。可见，大多数统计数据都会人沮丧，这说明没有一个模型可以捕获项目核心特征和开发关系。关系间代价是因子十分复杂，模型必须足够灵活才可以处理，使得某些人和方法被灵活应用。

此外，Kitchenham,Mou Donnell,Pickan 和 shepperel(2000)指出 MMRE 和 PRED 不是评测准确性的统计数据。他们建议使用最简单的比值，估计值/实际值。这种方法可直接反应准确率。而 MMRE 和 PRED 则告诉我们分布的有关特征。

即便评估模型给我们提供准确的估计值，我们也必须理解开发过程中努力是必须的。因此，当你建立自己的数据库评估效率时，还应该记录曾经花非费在项目上的功夫和活动，任务上的效率。

3.4 风险管理

可以看出，许多软件项目经理采取措施来确保他们的项目及时做完，并且保证在费用预算消耗内。管理者们还必须确定任何不受欢迎的事件发生时该采邓哪些措略来维护系统。减少损失。风险则指那些不希望发生的事件。项目经理必须学会风险管理，学会在项目进行过程中，控制风险。

什么是风险？

在软件开发过程会有许多事件发生；表 3.4 制出了 Boehm 认为是风险的一些实例。我们可以从三方面区别是风险和项目的其它事件。

1．与事件相关的损失。一个事件肯定会对一个项目产生负面影响：时间流失，质量。金钱、控制、理解等。比如，当设计完成时，需求又有了戏剧性变化，那么该项目将忍受控制的理解上的负面影响。尤其是新的需求所要求的功能不是过去所熟悉的功能时。而有需求的根本改变还可能导致时间浪费，资金浪费。这些称为风险后果。

2．事件发生的可能性。对于事件发生的概率我们应该有所了解。比如一个项目正在开发，并即将把又给测试过的都分转移制新的系统，假如新系统是我们所不熟悉的另一种模型，我们就必须估记新系统不能正常工作的概率。风险的概率，从 0 到 1，是风险发生的可能性，当概率为 1 时，称之为问题。

3．结果改变极度。对于每风险，我们必须确定，为减少或避免影响所能做的努力工作。风险控制包括一系列减少风险所带来损失的策略。

我们可以用风险后果乘上风险概率的值来量化风险的，产生风险。

有 2 种引起风险的原因：遗传风险和项目特有风险。遗传风险则指对所有软件项目共同拥有的，比如误解需求，人员流失，测试时间不充裕。项目特有的风险指给定项目的脆弱性带来的。

风险管理活动

风险管理包括几个重要步骤：每一步由图 3.15 说明。首先要充分估计一下你项目中的风险。这个语汇估包括 3 个活动：确定哪些活动是风险，分析他们给每个风险指派优先级。确定见险可通过许多不同技术来完成。

如果你现在制做的系统和原来曾经设计的非常接近，你就可能有了可能发生事情的列表。这次可以重新审视这些问题。而对于一个新系统。则这些问题需要讨论才能确定哪些是开发周期内的活动；通过分解过程，就可以发现问题。

最后分析所确定下来的风险，这样可以尽可能理解他们。有各种方法提高理解能力，这些方法包括动态方法，代价分析法，性能方法和网络分析法。

现在你已经列出所有的风险，必须利用已有的知识给每个风险赋上权值。这种优先赋值策略可以使你将有限的给那些最具威胁力的风险。

风险可通过计算风险结果和风险概率得到。于是你必须评价这些风险的方方面面。可以参见图 3.16 学习如何评价这此风险。假设你已经分析了系统，你将创建出一系列的版本，每个版本都包含一定的功能。由于这个系统设计的功能相对独立而且你正在考虑测试这些新功能，并假定已有的功能正常运作。这时可以预测会产生影响测试继续进行的风险，即隐性测试，能否保证现存的功能正常工作吗？

对每个产生的结果，你估计两方面因素：一个不希望出现的结果的了现概率 100，该结果所常来的损失（LVO），例如，对于隐性测试有三种结果：找到一个关键性错误，存在但没找到关键错误，确定没有关键性错误。如图所示，我们已经估计他们的三种情况分别是 0.75，0.05 和 0.20。不希望出现的结果发生的可能性估计为 0.05 百万美元的损失，如果找到一个关键性错误，那么风险是 0.375 百万美元。

风险　可以帮助我们给风险排序，最受关注的风险具有最高优先级。下一步我们必须采取措施控制风险。控制并不表示我们能够消除所有风险。我们只能减少风险所带来的损失。风险控制包括：风险缩减，风险计划和风险解决。

风险缩减有三步：

。避免风险，通过改变性能需求或功能需求。

。转移风险

。承担风险

为了缩减风险，我们还必须考虑减少风险的代价。我们称风险/减少风险所带来的带价为风险杠杆，换勿话说，风险杠杆是

（减少前的风险　—减少后的风险　）/（减少风险本身带米的代价）

如果杠杆值还高，到可以调节活动的程度，我们就可以考虑更换言之一种减小风险的方法。

在某些情况下，我们可以通过先选择开发过程帮助减少风险，比如从第二章我们可以看到原型法可以提高对需求和设计的理解，因而选择恰当原型处理方法可以减少许多项目风险。

在风险管理接计划中记录你的决策是非常有益的，这样客户和开发小组都可以重新检查文档使问题得以避免。于是我们就该在开发过程当中管控制项目，定期地重评估风险发生的概率，和可能带来的后果。

3.5 项目计划

我们通常用一份项目计划记录交流风险的分析和管理，项目代价分析，进度安排，客户组织等文档。该计划含有用户的需要以及我们要满足他们所需工作。客户也可以参考这份计划得到有关各项同任务的信息。我们还能用这份计划使客户相信我们所教的工作，尤其有关费用和进度。

一份好的项目计划应包含

1．项目所涉及范围；
2．项目进度；
3．项目团队管理组织；
4．技术描述；
5．项目标准、过程、所需技术和工具；
6．质量认证计划；
7．配置管理计划；
8．文档计划；
9．数据管理计划；
10．资源管理计划；
11．测试计划；
12．培训计划；
13．安全计划；
14．风险管理计划；
15．维护计划；

范围定义了一个系统可能涉及到哪些领域。这能够体现我们是否了解客户对我们所提的期望。进度可以用工作分解图表示。

项目计划还列出了开发小组成员名单，及组织形式，每人的任务。该计划也包含资源分配情况。

书写计术报告可以使我们集中精力回答和解决开发过程中的问题。这份技术描述包含硬件，软件，编译器，接口和一些特殊的设备。任何有关电缆，执行时间，反应时间，安全和其它方面功能，性能都应列在计划中。该计划还包含任何所涉及到的标准，方法。如

- 算法
- 工具
- 审查成检验工具

- 设计语言
- 测试技术

对于大型项目，有几份质量保证计划也非常适用，他们描述怎样检查代码，检验成测试其它技术。同样大型项目还需要配置管理计划，尤其在于有多种不同版本发布时，该计划非常有用。在第十章可以着到配置管理可以帮助我们控制软件的考贝。可以让客户了解我们是如何跟踪需求变化，设计，代码编写或文档。

许多文档都是开发过程中产生的，尤其对于大型项目，有关设计信息对于小组的每名成员都是可得到的。项目计划列出了所有可能得到的文档，解释谁将负责记录文档，什么时间和配置管理计划沟通，描述文档是如何变化的。

由于每个软件系统涉及数据输入，计算和输出，项目计划必须解释数据是如何聚集、存储控制和存档的。该计划还要解释资源是如何使用的。比如，如果硬件配置包括可移动磁盘，那么该项目计划的资源管理部分将解释每个磁盘上有何种数据，磁盘是怎样组织的，分配和备份。

测试程序则要示详细设计才能达到有效性，项目计划描述了项目的整个测试方法。尤其是该计划应该描述测试数据是如何得到的，每个程序模块怎样测试的，各模块间如何集成在一起，整个系统是如何测试的。当增添新功能时，那么必须用回归方法，确保以有的功能依旧工作。

在开发过程中训练课程与文档也需要准备，而不是在系统开发结束后准备。安全计划描述了系统是如何保护数据的，用雇用硬件的。即然安全性涉及机密性，可得到性，完整性，该计划必须解释每个安全限制是如何影响系统开发。

最后，如果一个项目在递交给用户后还需要维护，则项目计划还需讨论代码改变的责任分配，硬件维修，更新支持文档和训练材料。

3.6 处理模型和项目管理

我们已给看到一个项目的不同方面是怎样影响效率，代价和进度要求的，经及风险。那些能够在指定时间内完成高质量产品的成功的项目经理往往能够将项目管理技术量体裁衣适用到实际的资源需求中，过程和人力资源上。

想要理解工程下步要做什么，就要求助于项目管理技术。本章用到数字 Alpha AXP 项目和 F-16 飞行器软件。

注册管理

数字设备公司用许多年开发了它的 Alpha AXP 系统，一个新的体系结构并包含许多产口，这些形成了数字公司的历史。整个软件部分网络系统，编译器，数据库，集成木架，应用。与其它开发工作没的是，Alpha 公司主要问题是太早遇到里程碑。这样有关指导项目管理信息只能在最后阶段得到。

在开发过程中，项目开发经理开发了一个模型包括 4 个信念，称为注册管理模型：

①创建一个恰当大小的共享版本

②从参与者中选出特指的一些责任

③认真栓查并提供支持性反溃

④承认每次进步和随程序进展的学习

图 3.17 说明了这个模型。版本用于“注册”相关的程序，因而被称为共享公共目标。每个小组根据这个总的目标定义自己的目标。下一步随着经理们开发计划，他们给小组分配任务，反映明责任，进度限制。每个结果都是可以衡量的。而且有专门的所有人负责递交。这个所有者未必是做实际工作的人，但他是负责人。

经理继续检查项目以确保递交能够及时。项目组成负被要求明确责任，当风险威胁该组的承诺实现时，经理宣布该项目为一个 cusp：一个关键事件。这种宣告则表明组员需要做一些改变以便项目顺利进展。对于每个项目，经理应该承认个人和团体的进步。他们要记录所学的，并寻问问题的解决是如何进展的。

让所有的硬件和软件小组协同合人是件困难事，经理要认识到他们得还见到技术和项目上可能事件。也即，技术焦点在技术设计和策略上，而项目焦点强调责任和交付。图 3.18 说明了一种组织形式，该形式允许 2 种 foci 对整个系统起到一定作用。

模型与组织的简洁并不表示对 Alpha 和管理也是简单的。几个关键问题影响着项目而且他们曾被用几种不同方法解决。

当一个关键任务被宣布要落后进度几个月时，另一个关键问题出现了。管理层用几种可操纵的进展检查方法解决了这个问题。这个检查包括一页长短的报告，在这列出几个关键点。

- 进度
- 还没解决的问题及附属问题
- 里程碑
- 关键路径事件（上个月）
- 沿关键路径的所有任务（下个月）
- 解决的问题与附属问题

Alpha 成功的最重要一个方面是它的经理意识到工程师更容易认可的是物质鼓励而不是财政目标。取而代之奖金鼓励，他们强调进步并且确信经理对他们有多么信任。

Alpha 灵活和集中的管理所带来的结果是它完全吻合进度安排尽管这期间有许多挫折。注册管理使小项目组意识到问题的存在，并采取步骤逐渐解决它。

记帐模型

美国空军和 Lockheed Martin 形成了一个完整的产品开发小组来创立一个模块化软件系统，用来提高能力，提高功效，减少代价。最终该软件系统包括至少 4 百万行代码，本是要用来满足战斗飞行的实时要求的。F-16 开发还包括设备驱动器。Ada 运行系统的实时扩展，软件安装接口。

整个飞行软件的需求分析的非常好，并且十分稳定，尽管这里面有 8 个产品小组，一个工程负责人，外加项目经理，并且拥有一百万行代码。但是整套软件所需且备的能力要用另一种不为人熟悉的方法：使用 Ada 和面向对象模块化软件设计方法。项目管理约束规范还包括强硬的需求日期，开发三个版本的责任制。

随着资金提供的水平下降，最后期限的不切实际，该项目的压力增加。此外，这个项目的管理组织与以往大大不同。参与者使用一种矩阵组织方法。每个工程师依据个人技能划分到各个切功能模块小组。同即每个雇员可以在矩阵中找到他/她的位置。决策是通过功能结构的层次关系不制定出来的。然而 F-16 合同上要求该项目用集成产品开发系统组织。

为了让项目小组成员能够应付这种变化，F-16 项目使用了图 3.19 的计账模式。在此模型中一个团队就是由负责某项任务的所有人组成的。这个记录过程包含了一系列内容：你曾经做的，正在做的，计划要做的。

该模型用于系统管理设计，管理团队运作过程，取代独立特行的行为。强调要做的好而不是看上去好。

几种实践活动成为了必须。为强调职责的概念，每一个人的活动都有名确的界限，直到跟踪该任务完成为止。

由于这个团队有多重的，相互重叠的活动，因而我们需求一个活动图来描述整个项目中进行。图 3.20 表示该项目活动图的一部分。从中可以看出每个栏目是如何表示一项活动的。栏目上的点表示此处应该给出详细的活动指导，“Todas”线不示当前状态。

对于每项活动，都采取一定方法评价其进展情况。有时这种方法包括代价估计，关键路径分析或进度跟踪。Earned 值作为一种普通方法比较不同活动时展情况。该值计算包括权值计算，每一步的百分比。同样每项任何也有一个比值，表示占总任务的大小。

一旦一部分任务完成时，它的进展跟踪停止。它的性能则有所记载，问题也被记录。优先级表列出每次工作讨论中出现的问题，以及这些问题在将来该如何避免。

项目经理，用计账模型发现了一个主要问题：在不同队伍之间没有任何合作信息可得到。结果他们建立了归类目录和队与队之间的结果接口，这样每个队可以了解谁在等待他们队的产品。这种传递模式用做计划，删去不必要的模式或说明。这样检查这种 hand-off（脱手传递）模式成了检验过程的一部分。

显然可以看出计账模型，和与其配合的脱手传递模型是如何解决项目管理的若干方面的首先它提供了一种交流与合作的机制，其次它提揭晓风险管理，尤其是要求大家在讨论会上检查问题的存在，第三，它将问题解决融入进步报告中。这样该模型实际上描述了 F-16 项目管理过程。

借助里程碑

在第二章我们已经考查了许多描述软件开发过程技术进步的模型。那么在本章，我们看看几种组织项目法的方法。这个 Alpha AXP 和 F-16 例子向我们表明项目管理不只和进步跟踪相联，还和开发过程紧密联系。Boehm（1996）已经确认了对所有软件工发都具有的三个里程碑：

- 生命周斯目标
- 生命周期体系
- 初始远作能力

我们可以更详细地说明每个里程碑

生命周期目标里程碑是用来确保总负责人同意系统目标。总负责人在队伍中的任务是确定系统所涉及和范围，系统运作的环境，与之交互的外部系统。然后总持有人开始针对系统进行任务书做工作，该份任务书可能以用原型，布局，数据统，或者其它表示。如果该系统属于商务处理型或安全第一型，该任务书还须包括系统失败的实例，这样设计者可以有针对的解决这些问题，或绕过问题设计系统。初始生命周期计划列出以下的计划（1996）

- 目标：系统为何被开发
- 里程碑与进度：该做什么
- 责任分配
- 方法：工作如何做
- 资源：每种资源需要多少
- 可行性：该任务能被完成吗？

生命周期体系和生命周期目标相结合。生命周期体系里程碑目标在于定义系统和软件体系，我们在第五章，5 章和第 7 章的组成部分。体系方面的选择必须同解决风险管理带来的项目风险强调系统在长期内的进展，以及短期内需求。

初始运作能力的核心因素就是软件本身。Boehm 指出不同的处理方法都可用来实现初始运作能力，不同的评估技术还可用在同阶段。

为了实现三个里程碑，Boehm 建议使用 Win_Win 螺旋模型，图 3.22 描述。该模型鼓励参与者获得共同理解系统的下一阶段目标，方法和限制。

Boehm 把 Win_Win，称为 W 理路方法，应用到国防部的 STARS 项目，该方法焦点在于开发一套，软件开发原型系统。Win_Win 模型引出向个关键的协商，有公共开放接口协商，这些接口用来使厂商以较为低廉价可知获得市场份额。Boehm 报告指出空军项目费用从$140/每行代码到$57/每行代码。其它的几个项目也达到了同样的成功。第一次增长包括

将代码分布成生命周期体系的一部分，该项目要求说明其它能满足项目需求的种种能力。

第 4 章 需求获取

本章，我们来看一下

- 从顾客诱导需求
- 需求类型
- 需求获取的符号和方法
- 复查需求以保证质量
- 文档化需求以供设计和测试团队使用

前几章，讨论了系统开发的阶段。成功的软件开发都有几个关键的步骤，每个软件开发过程模型都包括捕捉软件需求的活动：理解顾客和用户期望这个系统能做什么。因而对系统的意图和功能的理解就从考查需求开始。在本章将看到需求分功能的和解功能的需求两类；我们会先研究每一类型的特征。然后讨讨需求集合的特性，如完全性和一致性。我们会看一个定义需求的方式，研究静态描述和动态描述间的差别。各种需求规格方法和符号将给予详细说明，既有自动的也有手工技术的例子。定义需求时，我们也学习怎样文档化需求，然后在需求复查进复查需求的正确性和完全性。

项目在尺寸和范围上各不相同。在本章结尾，我们学习如何为手边的项目选择适当的需求规格方法。分析需求活不仅仅是写下顾客想要的东西。我们必须找到我们和顾客都赞成的，且我们用它能建立测试程序（procedure）的需求。首先，确切地检查一下需求是什么以及我们如何同用户、顾客一同定义并文档化需求。

4.1 需求过程

顾客在请我们建立一个新系统时，已经对系统将做什么有了些概念。通常，新系统代替旧系统或做事情的方式。有的新系统是当前（手工或自动）系统的增强或扩展。例如，电话记费系统可能负责顾客每月一次的存取，现在可能被更新呼叫转发、呼叫等待和别的新服务计费。频繁地，提议的系统被计划用来做以前从未过的事：按顾客兴趣剪裁电子新闻，改变飞行着的飞机机翼的外形或监测糖病患者的血糖并自动控制胰岛素剂量。无论功能是新还是旧，每个基于软件的系统都有目标，表达为系统能做什么，需求就是系统的特征或者说是系统为完成系统的目标所能做的某事的描述。

图 4.2 说明了确定一个基于与软件的系统的需求的过程。首先我们通过问问题、论证相似系统，甚至开发目标系统的全部或部分原型同顾客共同工作以引出需求。接着，我们捕捉在一份文档或数据库中的需求。需求首先被写下来以便于我们和顾客对系统应做什么达成一致。需求常被以一种更加数学化的表示方法重写，以便设计员能更好地将系统需求转换为好的系统设计。一个查核步骤保证需求是完全的、正确的、一致的。一个确认步骤，确保我们已经描述了顾客在最产品中看到的东西。补充栏 4.1 描述了为什需求过程对好的软件开发是重要的。本章其余部分更详细地探究了此过程。

需求诱引<requirement elicitation>

需求诱解是此过程中一个相当重要的部分，我们必须使用各程技术来确定是用户和顾客正想要什么，有时我们将自动化一个手工系统，因而很容易确定已往做了什么。但是，当一个方案还未被找到时，我们通常要和用户、顾客工作理解问题。在考虑任何解决方案前必须分析问题，通常通过把问题分成小的能理解的问题来完成分析。进行问是分析的一程方式就是确定涉及的人、处理和资源，然后将它们间关系文档化。我们问涉及的用户、顾客并努力确定系统边界。我们弄明白哪些数据项从一个角色传到另一个角色、哪些过程将数据从一种形式或状态转换为另一种。整个需求诱引过程中，我们以多种方式问同样的问题，以使我们确信我们理解了顾客和用户想要和需要的东西。

将需求分成三类是有帮助的

1．绝对要满足的需求

2．很可取但不是必需的需求

3．可能但可以削减的需求。

例如：一个信用卡计帐系统必须能列出当前赊账，计算其总额，并要求到某个日期止偿还；这是第 1 类需求，但也可能将赊账按购买类型分开以方便买方理解，这是一条第 2 类需求，最后，计账系统可能可能以黑字打印存款并以红字打印负债，但此需求很可能处于第 3 其中。按类需求分析助于各方来理解什么是真还需要的。这种按类需求分析在软件开发项目受时间或资源限制时也是有用的；如果被定义的影响花费太多或耗时太长而不能开发的话，那么更可删去第 3 类需求，并分析第 2 类需求看能否削减或延期。

每一条系统需求处理对象或实体，它们可能处于的状态，和被执行改变状态或对象特征的函数。例如，假定我们在为顾客的公司建立一个产生付薪支票的系统。一条需求可能就是：支票每两星期发布一次。另一条可能是：允许所有处于某薪水级别或其上级别的雇员的支票的直接储存。顾客可能从公司几个不同位置请求存取付薪支票系统。所有这些需求是特定的追求系统要普遍的总目标的函数和特征的描述。因此，我们寻找定义系统对象的需求（“一名雇员就是一个由公司发给其薪水的人”），限制对象（“一名雇员每周能付薪的时间不超过 40 小时”）、或定义对象间关系（“如果雇员 Y 有权变要 X 的薪水那么 X 受 Y 监督”），补充栏 4.2 解释了我们也如何保证我们能测试以确定需求是否已经得到满足。

注意这里的需求没有一条指定系统将如何实现。换句话说，不提用什么 DBMS、计算机有多少内存，或必须用何种语言开发系统。我们认为这些特征实现的描述不是需求，除非顾客命令了。就是说，一需求说明了系统的目标而不考虑实现。根据这来看，很容易看到一些系统特征是不切题的。特别地，那些与系统目标没有关系的特征应从需求规格中删除。我们说需求确定了系统“是什么”；设计确定“怎样做”。

如果我们牢记了需求诱引和分析的目标，这条区别就会变得更清晰。图 4.2 显示了在更广的系统开发上下文中这些活动适合于哪些地方。需求在开发之初就被诱出，我们的目标就是确定顾客问题的本质。问题未被清晰定义之前，任何解决方案都是过早的。而且，问题最容易以顾客的业务术语来表达。两种类型需求文档。

因为焦点是在顾客的问题上，需求诱引和分析为两个分开但相关的目标服务。一方面，需求诱引使我们能写下一份需求定义文档；也是以顾客能理解的术语写下的，需求定义完全地得出了顾客期望系统要做的每件事。也代表了顾客和开发者对顾客需要成想要的东西的理解，且通常由顾客和开发者 联合书写。另一方面，需求规格重新以适合开发系统设计的技术术语来重新表达需求定义，也是需求定义文档的技术副本，是由需求分析员写的。

补充栏 4.2 使需求可测试

Alexander(1979)、写了有关好的设计的文章，他鼓励我们使需求可测试。他这样做的意思是：一旦需求被表达后，我们能考重新有可能满足需求的实践和活动，然后把他们分为两类：满足需求的和不满足需求的，这种划分必须则可重复的；就是说，类中成员不随谁来分类而变化。

Robertson 和 Robertson 指出可测试性（他们称之为“可度量性”）能在需求一抽出来时便追求它。我们可对能测试一个潜在方案是否满足给定需求的方式进行量化。这些合适的标准形成了需求意义的客观描述；当那样的标准不容易表达时，那么需求很可能是含混、不完全或不正确的。

例如，一个顾客可能如此表达一条需求：

水质信息必须能即刻存取。

顾客可能很明白“即刻”是什么意思，并且那样的需求必须马上被捕捉，我们能更正确地重述它：

水质记录必须在请求后 5 秒里被返回。

它能被客观地测试的是第二种形式的需求：一系列请求产生、并且，对每系请求，系统必须在 5 秒提供记录。

Robertsons 提议了帮助使需求可测试的三种方式：

。为每个副词和形容词指定一个数量上的描述以便质量的意思清晰、

不含混；

。用特定的实本名替换代词；

。确保每个名词在需求文档某恰当位置被定义。

有时单独一个文档服务于两个目标，顾客、需求分析员和设计员的共同影响。但通常两型文档都需要，并且在将需求定义重新解释为规格时，特别要小心信息丢失或变化。

定义和规格中的需求要有直接对应关系。用在整个生部周期的配置管理方法就从这里开始，配置管理是一个程序(procedures)集合跟踪：

- 定义系统应做什么的需求
- 产生自需求的设计模块
- 实现设计的程序(program)代号
- 查核系统功能的测试
- 描述系统的文档

在某种意义上，配置管理提供了把系统各部分捆在一起的线索，使已被分开开发的组件一致；这些线索元件我们协调开发活动，正如图 4.2 实本水平“线索”一样。特别地，在需求诱引和分析期间，配置管理详细说明了需求定义中元素和需求规格中元素问的对应关系以便顾客视图被以一种组织了的，可更深入的方式联系到开发者视图。如果我们没有定义这些联接，我们没任何办法设计测试案例来决定代码是否满足需求。在后面的几章中，我们将看到配置管理也是怎样让我们确定变化的冲击，及控制并引开发的影响的。

功能的和解功能的需求

需求描绘系统行为。当系统作用于数据或指令上时，对象或实践从一种状态迁移到另一种状态：例如由空到满、由忙到静，或由发送到接收，就是说，在任何给定状态，系统满足一个条件集；当系统行动时，它可以通过改变一个对象的状态来改变它的整体状态。需求表研究院系统和对象状态和从一种状态到另一种状态的转变。特别地，需求描述了系统的活动，诸如：对转入的反应和活动发生后系统中各实体的状态。例如，在工资系统中，雇员能以至

少两种状态存在：尚未付薪的雇员和已付薪雇员。需求描述了发放付薪支票时一名雇员如何由第一种状态变为第二种状态。

为有助于我们描述需求，我们能以两种方式考虑它们：功能的和解功能的。一条功能性需求描述了系统和环境的相互作用。例如，为确是功能的需求，我们决定什么状态是可接受的，系统将处于的状态。而且，功能的需求描述了系统对给定激励应如何举动。举例，对一个打印一周一次付薪支票的系统，功能的需求必须回答定于付薪支票何时发布的问题，要打印一张付薪支票，必需的输入是什么？什么条件下支付数量可改变？什么促使了一名雇员从薪水册列表中删除？我们可使用各种技术来确定功能的需求，包括使用补充栏 4.3 新描述的案例，更详细的将在第 6 章解释。

功能的需求遇到的问题都有独立于顾客问题解决方案实现的答案。我们描述系统将做什么，而不讨论：我们可能用的特定的计算机、用的编程语言、涉及的内部数据结构，或用来打印支索的纸张类型。不告诉我们系统将做什么，而只是对系统增加限制的需求，就是解决功能的需求。也就是说，一条解功能的需求或约束描述了施加于系统上的一条限制，限制了我们构建问题解决方案的选择。例如，我们可能被告知系统必须在一台 Aardvark 计算机上开发或者读取初始数据后不超过 4 小时时间里付薪支票必须被分发到雇员手中。类似地，我们被告知对系统的查询必须在 3 秒内得到回答。这些限制通常限制了我们对语言平台或实现技术或工具的选择，然而，直到需求已被指定后的设计阶段才做出这样的选择。

补充栏 4.3 用例（UseCases）
确定系统的功能必需的需求的一个方便的方式就是确定它的用例。用例将系统分为逻辑的，最小相关的片的集合，每一片描述了某条系统将起作用的方式。我们将在第 6 章更深入探究用例，因为它们是进行面向对象分析，不可缺的一部分 。Robertson 指出从用例角度看待一个系统的几点优势。 。因为用例间几乎没有联系，可分别考查每个用例而不必知道更大 的系统的细节，特别地，一种类型的用能理解提出的功能而不必了解别的用户的功能 。我们能用用例来作为评估设计，编写系统新要花费的时间和努力的依 据。 。系统开发能根据用例跟踪。也就是说，管理真能在设计、编程、测试 系统时。 跟随上用例进度。

功能的和解功通报需求二者以正式的、仔细的方式众顾客那里引出。正式的需求诱引是必需的，因为顾客不是总善于正确地描述他们想要的或需要的，而我们也不是总善于理解别的某人的业务事务。顾客知道业务，但他们不是总能够向外人描述清楚业务问题；描述充满了我们不熟悉的专用术语和假设。同样，作为开发者的我们知道计算机解决方案，但不是总能知道可能的方案将如何影响顾客的业务活动。我们出有我们的专有术语和假设，有时我们认为在说同样的语言而实际上我们对同样的情有不同的意思。因此，如果不仔细组织和鼓励，我们和顾客间的交流能导致误解和不完全的规格。

4.2 需求的类型

需求定义和规格文档描述了单位将如何与它的环境交互的每件事。包括如下项：

物理环境

。设备在哪里发挥作用？

。有一个位置还是几个？

。有一些环境限制等，如温度、湿度或磁声干扰？

接口

。转达入来自一个或多个别的系统？

。转业将去至一个或多个别的系统？

。有用于格式化数据的规定的方式？

用户和人为因素

。谁使用系统？

。有几种类型的用户？

。每种类型用户的技术水平怎样？

。对每型用户需要什么样的培训？

。用户理解、使用系统的难易度怎样？

。用户误用系统的困难程度怎样？

功能

。系统将做什么？

。系统将在何时做？

。有几种操作方式？

。系统能在何时、怎样被改变或增强？

。对执行速度，响应时间，或转出有限制？

文档

。需要多少文档？

。它应该是联机的、以书本格式或者二者？

。每种文档面向的哪些读者？

数据

。数据应以什么格式转入、转业？

。数据收或发的频度？

。数据的精确度/

。有多少数据流过系统？

。一些数据必须在任何时期都要予以保存？

资源

。建立使和维护系统都要些什么材料理、人员或其他资源？

。开发者必须具有哪些技术？

。系统占用多少物理空间？

。对开发规定了时间表了？

。对用于开发或软硬件上的钱数有限制？

安全

。必须按制对系统或信息的存取？

。一个用户的数据将如何同另一个实现隔离？

。用户程序如何和别的程序及和操作系统隔离？

。系统如何备份？

。备份副本必须被存于一个不同的位置？

。应采取措施防火，防水防盗？

质量保证

。对可靠性、有效性、可维护性、安全性和别的质量属性（第一章介绍了的）有什么要求？

。如何向别人示范了系统特征？

。系统必须检测并隔离故障？

。失败间的平均时间规定为多少？

。对一次失败后重启系统有一个最大时间？

。系统如何将变化合并到设计？

。维护将仅仅是纠正错误码率。还是色括改进系统？

。对资源使用和响应时间都可应和些什么有效的度量？

。将系统从一个位置移到另一个位置或从一种类型计算机移到另一型计算机的容易程度如何？

Volere 需求过程模型提议了几个需求来源，如图 4.3 所示。这个模型显示需求能以多种方式被诱引。我们通过有创造性地弄清顾客想要什么能扩充这些概念。

例如我们能

。复查当前状态

。与用户一起当学徒来理解上下文环境，问题和关系

。会见当前和潜在用户

。做一段录象以显示新系统可能将如何工作

。探现存文档

。和当前及潜在用户集体研讨

。观察结构和模式

4.3 需求的特征

需求不仅描述：进出系统的信息流和系统所进行的数据转换，而且也描述了对系统运行所施加的束缚。因此，需求服务于三个目标。首先，让开发者解释他们对顾客想要系统如何工作的理解，其次，它们告诉设计者结果系统将具备的功能和待点。第三，需求告诉测试团队怎样验证以使顾客确信所交付的系统的确是新要求的系统。

为确保我们和顾客二者都能正确理解并使用需求，需求应该是高质量的，这很重要。为达到那样的目的，我们检查需求有下列特征：

1．需求是正确的么？我们和顾客都应复查它们以确保它们被无错表达

2．需求是一致的么？也就是说，没有任何冲突或含糊的需求？例如，一条需求说系统在一个时间里最我只能有 10 个使用者而另外一条需求说在某条件下可能时有 20 名使用者，我们说这些需求是不一致的，换句话说，如果两条需求不能同时满则说二者是不一致的。

3．需求完全么？如果所有可能的状态、状态变化、转入、产品和约束都在某条需求描述了，那么说这个需求集合是完全的。因此，一个工资系统应该描述如下情况出现时要发生的事：一名雇员未领薪就走人了，提升了，或需人加薪了。我们说一个系统描述是外在完全的，如果描述包含所有的顾客要求的正克服关系。我们说一个系统描述是内在完全的如果在需求中没有任何不明确的作用。

4．需求是实际的么？系统真的能做顾客所请求做的事呢？例如，假定一个系统请求用户存取位于几千英里外的主计算机，并且对远程用户的响应时间要和对本地用户（那些工作站区接与主计算机相联的用户）的响应时间相同。这条需求可能是不实际的，因为通信线路

上的传转需要额外的时间。类似地，有时，当开发时间很长时，顾客试图进行技术改良，要求艺术化的需求。所有的需求应该队员证它们是可能。

5．每条需求描述的事物是顾客需要的么？有时，一条需求多条地限制了开发者，或者包括了转头问题不直接相关的功能。例如，一位将军可能决定一：一辆坦克的新的软件系统应该让士兵能收发电子邮件，尽管坦克的主要目的是辗过不平坦地带，我们应该复查需求来确定哪些直接与顾客的问题有关的需求。

6．需求是可检验的么？我们必须能写出测试来验证已被满足了需求。

7．需求是可跟踪的么？每一系统功能都能被跟踪到要求它的需求集合么？容易找到处理一个系统特定方面的需求集合么？例如，为了复查通讯需求，将需读所有的需求呢？

我们能用清单细查每个可能的需求，尽我们所能地会改进、更改。例如，考虑我们可如何来测试需求：

系统应该对查询实时响应

我们不知道“实时响应”是什么。然而，如果需求将说：

系统将在不超过两秒的时里对查询作出响应。

那么我们确切知道了怎样测试系统对查询的反应了。

假定一顾客对一个卫生控制系统提议这样一条需求：

精确性将足以支持任务计划（mission planning）

怎样测试系统看它是否满足这条需求？需求没有告诉我们都有些什么任务计划要支持，我们可以讨论任务计划的意思并重新记录这条需求：

在确定卫生的位置时，位置误差沿轨道不多于 50 英尺，隔离轨道不超过 30 英尺。

在这种情况下，我们能测试位置误差，并知道我们是否已满足了需求。补充栏 4.4 提供了一个例子，是关于我们的一些需求转为如何被自动化。

补充栏 4.4，检查完全性和一致性
Heimdaht 和 Leveson 已经开发出了一种在层次性的、基于状态的需求中检查完本性的一致性的方法，他们的体系使用的是需求规格建模语言，这种语言是在 lrvine 的加利福利亚大学开发的。 在 RSML 中，正在被建立的系统被认为是一台有限状态机，并且这种语言给状态，转换及事件解到建模。这种印恻的 RSML 规格定义了一个数学“下状态”函数，理想地，这个函数应定义在所有可能的系统状态；这条特性被称为 d-完代表性一。函数也应该没有冲突的需求，意思系统是一致的。 RSML 被用于分析 TCAS Ⅱ，一程用于美国空域的冲突规避系统，通过用“下状态”方法描述 TCAS，Heimdahl 和 Leveson 揭示了一种意外的写此规格时它是不明显的。

4.4 怎样表达需求

象许多计算机科学中的活动一样，最好自上而下来做需求定义。换句话说，我们先在最高层次上表达系统总的属性；然后，在其次的层次，属性被更具体化，许多年来，需求是以顾客的自然语言（使用正常的句子或词组）来指定的。然而仅使用自然语言有几个问题：首先，如果需求将是有用的，则用它的所有参与者必须以相同的方式解释它。如果顾客以一种方式考虑一个对象或实体而我们以另一种方式思考之，这样的需求将导致混淆；我们和顾客不可能对我们所有的话有相同的理解。例如，“可用性”对顾客来说可能有多得多的技术意义。在系统正进行备份导致用户不能使用终端时，顾客可能认为系统，是“不能供用户使用

的”。然而，我们可以说系统仍然在执行任务因而是“可用的”，因此，自然语言可能不是表达系统功能和它所相关部分关系所需要的精确、不含糊的中介。其次，需要不总是很晚地按照需求处理的系统元素被分开。有时从系统特征回头跟踪定义或影响它的那些需求是困难的（如果不是不可能的话）。自然语言的使用在这 里出能增加混淆。

因此，与文件工程师已经研究了许多方式来以更有力且受控形式定义需求。他们的方法通常是使用形式符号来描述将建立的系统。此方法的一个优势就是随同工具能被开发出来以检查规格完全性和一致性，并使跟踪管理更加容易。

任何需求集合应该描述系统的所有部分，包括边界。我们需要知道将包括哪些对象或实体、他们象什么（通过定义其属性）他们如何彼此相关，当它们进入，穿过或离开系统进会在它们身上发生什么。所需求都是用这些元素来描述系统。一旦系统元素被确定了，便可使用更详细的表示技术产生系统特定需求。让我们来研究这样的一些技术。

静态描述

一个系统描述列出了系统实体成对象，它们的属性（包含能在它们上运行或面它们来运行的功能），及相关间关系。因此，我们认为要求是相关的，也就是说，需求定义了对象或实体被此间的关系。这种视图是静态的，因为它不描述关系如何随时间变化。当系统的操作中时间不是一个主要因素时，那样的一份描述是很用且足够了。有几种静地描述系统的方式。

间接引用。能用对问题和它的解决方案用间接引用来描述系统。例如，假设问题是开发一个解一系列风个变量 KP 介方程的计算系统。对解决方案的实际算法已涵在其中而没直接表达。同这种类型定义一起，已经给出解决方案的特征而没有表述解方案的方法。因而，其它不能保证解决方案存在。

递归关系。一个相以的系统描述使用一个递归关系。在这种描述中，先定义一个初始条件，再根据前面定义的条件来描述一个条件到下一个条件的转换。例如你可能对斐博拉（Fibonacci）数熟悉，它定义了贝类转变的方式或急子繁殖的方式，斐博拉数可如下产生：

F（0）=1

F（1）=1

F(n+1)=F(n)+F(n-1)

类似地假定我们被邀请建一个系统来跟踪疾病在人群的传播。疾病的初次爆发以及这种疾病从一个从群传播到另一人群的方式者得到描述是为目标的目动系统将即时地产生任何地点对此种疾病事故的描述。

公理定义。看待系统的一个可选方式就是从公理角度。这种方法指定基本系统特性，而系统的行为就是从它他（指基本系统）产生新特性，称作定理。这种公理方法要求一个完全、致的公理集合；否则，产生的定理将不会表达出系统的真实情况，这种类型的需求定义尤其适合专家系统的开发，因为这样的一个系统涉及从有关特定义问题的基本知识的句子来产生新信息。

公理定义常用于明确说明抽象数据类型。系统被描述成是一个对象及在其上被允许的操作的集合，而公理说明了对象及操作之中的关系。本章后面将进一步研究数据抽象方法在说明需求时的用法。

作为一种语言的表达式。当一个系统处理一个数据串的集合时，我们有时把可接受的串描述成表达式，而这些表达式构成了一种可接受语言。例如一门语言的编程器读字符串、决定哪些是有效的哪些又是无效的。另外一个系统可能处理来自数据采集设备的信息串，在继续传递串以作进一步处理前检查每个串的有效性。

在这些情况下，需求成为串的语法的一种规格。一种特别种类的的语言能为有阴状态机

所识别。通过将需求描述为一种正则语言的串，并把有效串视为有效正则表达式，我们能够自动地检查需求的完全性和一致性。

例如，来自 Scenic Computer Systems(Redmond,Washington)的 Sprinter-2 文本文理系统是从以种正则语言表达的需求开发的。这个文本文理器读字符带，解释之，并格式化结果。Sprinter-2 语法条件是以 Backus-Naur 范式写成一个字符集合，如下所示：

ASCⅡ characters
Expressions(<expr>)
terms
factor
scale factors(<seale>)
funetions(<func>)
digits
letters
addition cperators(<addop>)
mwltiplication operators(<mpyop>)

字符问关系被表达为下列串：

<condition> =

（见原文 P149 面相应部分，此处略）

<underscore> =

这种方式来表达文本处理的条件，我们能将每个义和其它的比较（使用一个自动化处于，如果能用得上的话）以保证每个定义中的术语自身在别的某处被定义。

数据对象。在许多情况下，系统所操作的数据决定了所采取的引为的种类。把焦点放在数据而非功能上来定义需求是有用的。数据抽象是一门用来描述“数据是用来做什么”而不是“他们看起来如何”或“他们被称作什么”的技术。

我们通过形成一个数据字典来描述数据，中心思想就是数据分组并把相似元素分组在一起。每一类型这样的数据称为一个对象并赋予一个名字，字典以字母表充来包含这些名字。数据元素通过类型（type）或类（class）联系起来；如两个数据、元素如果有相同的一般形式和内容则是同一个数据据类型（data type）或处于同一个类（class）中。每个对象便被视为是所属类的一个实例（instance）。例如，我们要定义一个 student 类，在那里一个人如果在当前学期注册到少 6 个学分小时便认为是一名学生；如果你当前登记为一名学生，你就是 student 类的一个实例。

要决定数据类型是如何定义的，考虑你的学生记录。大学记录办公室跟踪每名学生在每学期取得的学分数，你的档案包含了你的信息（你的学生记录），包括你的姓名和学号，你已经记录的学期号，及那学期相关信息，通过知道数据的这 4 个种类后，你已经领会了记录中都有些什么，即使不是每个数据元素都被列出。

我们能以下列方式定义学生记录：

semester record
semester type
semester date
Grade-pomt awarage
Compteted hours
Semes type
(Fall,spring,summer)

Address information
Telephone number
Scteet address
city
stato
Pesstol code
Name
Stadent number
Address information
Number of semesters
Semster record

学生记录类型的实际定义是到表的最后。除的学生记录包含你的姓名和学号，随后是你的地址信息。地址信息自身是一种类型（type），并由包含地址、电话号码的五个元素来定义。在学期后，你的记录包含自加入此大学后每学期的学期记录。{ }指示学期记录类型可以被重复。

注意，学期 semester 数据类型是通过到举三个可能的入口来定义的。当一个数据类型能以这种列出所有选择项的方式完全定义时，我们用圆括号并描述每个选择。因此，一个 semester 数据类型对一个被描述的学期来说只能取 fall spring 或 summer 之一。当我们不能板举出所有可能选择时，我们常使用一种规则，这种规则描述数据能如何派生，或可能的取值范围是{1、2、…k}或{化和手}或{余额≧$10,000 的帐户}。

数据（对象）和数据类型（类）通常按它们彼此的关系来组织，以利用共享的特征。有很多表示这些关系的方式，而图常用于与顾客的讨论之中。图 4.4 描绘了一所大学中学生的类。因为这所学对来自国内学生的收费要较来自国外学生低，故我们可显示两型学生之间的关联系。图中三角形代表 UML 统一建模语言中的概括（generalization）；顶部 student 框是它下面各框的概括。

在使用数据抽象说明需求时，我们必须说明与数据和数据类型一起的被元语的行为（actions）我们描述为我们操作数据的引为或方法，而不是直接操作数据。因此，我们定义方法来告诉我们能使用数据的方式。通常每种方法涉及三型信息：数据能处于的状态，（states）建立新状态的操作（operations）及汇报一个状态信息的探针（probes）。数据抽象有助于我们抽取数据本质，并处理它而不致陷入数据表示操作的细节。图 4.4 中，每个框的顶部三分之一显示类或对象的名字；第二个三分之一列出属性；底部三分之一列出了与对象或类相关的操作或方法。这 student 中的 comprte tuition 操作是抽象的；这个图表显示实际费用计算依赖不同的费本。

动态描述（Dyhamic Description）

按照实体的关系描述系统时，难以说明系统在一段时期里是如何对改变系统行为的事性作反应的。因此软件工程师开发了根据随时间发生的变化来观察系统。系统被认为处于一种特定状态直到某些激励促使它改变状态，用这种方式说明系统使我们和顾客很容易描述所有可能的状态和激励；从而产生的需求更可能有完全。本节，我们研究从状态和激励的角度来描述系统的技术。

决策表（Decicion Table）。有时，我将系统描述为一个系统在一给定可满足的可能的条件：（某些条件集合被满足时对激励伯出反应的）规则和从而采取的行为集是便利的。

例如，假定的所大学的招生办正开发一个确定接受谁为新生的系统。表 4.1 显示如何做此决策。

条件列在表的左侧，每列代表一个条件集合因而是新的一个状态。每列下面的行为说明了系统处于列所代表的状态时接着的规则。入口“T”意味着行所表示的条件为真。“F”意味着条件为假，表示条件真假都无所谓。可能采取的行为显示在表的底部，用“×”表示。因此如果一名学生有高的标准考分，将发送他录取表格，而不管等级、保外活动和推荐。类似地如果一名学生有高能，出送也录取表格。在所有的其它情况下，学校将邮寄拒绝信。

决策表表示出当系统处于示状态之一时将采取的行为。这种表示会产生很大的表，因为状态数等于条个的组合数；如果有几个条个，则有工几何能的条件上组合。然而，注意，规则 3.4 和 5 是冗系的；我们能削减 4 和 5，因为它们的条件已经在了中条件覆盖了，通过这种方式检查决策表，我们能减少其尺寸，使它们易于理解。

我们能判断出有关需求规格的别的东西吗？那当然，我们能得到，如果每个可能的条件集合都导致了行为，那些么此规格是完全的。我们出可考查表来确定一致性并清除任何冲突情况。

函数描述和迁移图。我们能以一种类似的方式把系统看作一个状态集合，在那里系统对某些可能事件做出反应。例如，假如系统处于状态S，且事件×发生。事件×可能引起以某种方式行动：变到另一状态，维持在状态S里转出一个字符等等。系统的行为被解释成一系列函数，它的输入是一个条件集合和一个事件，而函数的输出是一个导致系统迁移到状态S_2的行为。我们可通过画出系统从一个状态到另一状态的运动图来描绘一迁移（transition）。首先，为系统每一状态画椭圆。然后果对每一状态和可能的输入，我们画一有向等头指示从一个状态到另一个的迁移，如图 4.5（a）。

当一个输入不引起状态改变时，我们能通过画一条开始，终止同一状态的箭头来指示。如图 4.5(b)所示，从状态S_1，一个转入 1 没有改变状态，而一个输入 0 引起也变为S_2。

一般地，我们能如下表达每个系统迁移：f(si,cj)=sk，指示当在 si 态时，条件 cj 的产生促使系统变为状态 sk。因而，系统的状态改变能以一个表格方式来（通过显示一份状态列表和系统处于此态时对转入的反应）表示。

还有别的描述状态，迁移的符号。图 4.6 是一个栅栏图，显示了一个旅馆房间预订系统的状态迁移图。每条水平线代表一个状态，箭头显示了可允许的从一个状态到另一个状态的迁移。如果说需要的话，箭头也可被标注上引起这种状态改变的条件名。

在 VML 中，描述状态变迁一下通过引起改变的条件或事件，二是状态变化时所可取的行为（动），象图 4.7 所示。

那样的表方式法对只有几个状态的小系统是有用的。例如，图 4.8 用 VML 符号来表示旅馆预订系统。行为（动）涉及到可用房间数量的变化和处于等待可利用房间列表上顾客的数目。注意，这些行为暗示了计数等待列表的存在；补充栏提醒我们使这些假设明显化，以便需求不会引起让人误解的解释。

补充栏 4.5，隐含意思和让人误解的假设
Jackson 在他的关于软件需求和规格的书中，创刊细考虑了阅读一份软件描述的过程。他鼓励我们对描述中的每件事提出问题，特别是对假设和隐含意思。例如，考虑图 4.9 所示的有限状态机。我们有几种方式来解释这张图： 。在暗（dark）状态，下推(push down)是不可能的，而在亮（lit）态，上推(push down)是不可能的。 。你能在 dark 态下推(push down)、在 lit 态上推，但没有任何影响。状态不变 。你能在在 dark 态下推(push down)、在 lit 态上推，但图中没有给出其影响。 有意义的名字引导我们对系统如何操作作出假设。例如，因为行为是上推(push down)，

我们期望下个行为是下推(push down)，——一种没被文档化或打算的假设。我们可能描绘成一个二位操作杆，而事实上图表中可能描述的是一个装有弹簧的操作杆，也在一推或下失真后便回到中间位置。因此，把期望或假设同需求实际所说的意思分开是很重要的。

事件表（Event Tables）。我们可以以一种不同的表格形式来显示系统的状态和迁移。象以前一样，我们先确定系统分解，然后形成一张表，其纵轴由状态或条件集合组成。沿水平轴，我们放置能发生的事件。表格元代表了系统处于行位置所指示的状态 F，在顶部列中事件出现时发生的行为。

为了明白怎样构建那样的一张表，假事实上Milly's Manufoctuhing正建一个服务于三个目标的自动系统。在另一模式下，系统为Milly的客户产生蓝图和结构图。第三种模式是本地计算机模式，允许程序员按需开发应用程序。我们在表 4.3 的左边列出三种操作模式。顶部放各种能的事件上。然后在每一行列交叉处对应格，我们指示出在行所示模式和列所发生的事件下产生的结果。

为了看表如何工作，假没事件工是“在终端上按下 help 键”。不同模式下发生不同的行为。在绘图模式下，按下帮助键将使用户移到一个显示着被执行函数的信息的荧前（A 行为 8）。在结构模式，显示给用户一个概括了最近被执引的一些函数的屏幕，跟着一个菜单问用户是否取消前面函数还是继续（行为了）。在本地计算机模式，可能显示给用户下一条所要求指令的提示（行为 4）。“×”意味着那条配置不可能；列所代表的事件在行代表的模式下从不会发生。“0”指示没有任何状态改变也没有发生任何行为。那样的条件可能存在，举例来说，用户处于结构模式而击了一个为图形模式而不是绘图模式而定义的功能键。

Petri 图（petri Nets）。迄今为止所描述的技术对那些状态和事件按序发生的系统是最有用的，当一次有几个事件发生时，有时一个系统必须执行并行处理，并且，特定的计算机被用来同时处理许多事。表示同时处理的一主要问题就是需要同步事件。几个事件可能并列发生面执行次序不可预料。

为描述同步和并行处理，我们可扩展前面使用的，**迁移（transition）**的概念。象简单的情况中，系统处于状态 A，一事件发生后状态变为 S：f(stateA,Event)→sate S 代替他，在系统离开状态 A 移于状态 S 前几件事必须同时发生。例如，一个用于打印公司付薪支票的系统在产生支票之前必须要求打印机里有适当的纸张，结算帐户的有足够的钱，一个标志支付结束的信号。三件事件能以任何顺序发生；而重要的事情就是在系统能转入书写支票状态前所有三件事件都要发生。在此形势下，几个事件触发了系统由一个状态向下一个状态的转移。我们可如下表示，一般的情况：f(stateA,Event1,Event2…EventN)→stateS。在取一般的情况，开始状态的迁移需要几个事件。然而，一旦迁移启动后，系统并行移入几个状态。我们表示这种迁移 f(stateA,Event1,Event2…EventN)→state1，state2, …stateM。为理解这种情况，考虑医院里的急救宝。在病人被救活前，几个事件必须发生。工作人员必须尽力弄清病人姓名，地址。还必须确定病人的血型。某人必须同看病介否有呼吸，也必须确定病人失血伤口。事件可不必按特定顺序发生，但在医生团队开始彻底的检查前所有这些事件必须发生。一旦治疗开始（例如，一旦从初始检查向更毛底的检查变迁），医生们就进入了新状态。整形到科医生检查损坏了的骨头，血液医生进行血液测试，外科医生缝合失血伤口。医生们的状态彼此独立，但是直到从初始检查开始的变迁发生这些事方能发生。

在这里，重要因素就是需要协调。活动正并行发生，我们需要栽种方式来控制改变状态的事件的收集（onection）。前面提到的技术中没有一样的适合这种同步的。Petri 网是一种

很适合表达并行处理需求的选择。Petri 网用图形化表示系统，也为每个状态画一个节点，画一个箭头来标记迁移。图 4.10 显示了一个例子：前面讨论的三种类型迁移可能性如可显示。

为了处理事个和状态的协调，petri 网的每一状态与一个令牌(tokens)集合相关联，如图 4.11 所示。令牌代表发生的事件。一旦一事件发生，一个令牌可能从一个状态移到另一状态。迁移由一个点火规则(firing rules)的集合来描述。每条点火规则解释了令牌如何与一个状态联系起来;当令牌的正确号码和类型被呈现在一个状态中时,令牌被释放以向另一状态转移。因此，点火规则对应于定义迁移条件的函数。令牌和点火规则的概念让 petri 能表示并同步化可能同时发生的活动。在图的上部分，一落千丈个令牌与垂直栏左侧的两个状态每个相联系。当每状态 的条件都满足了时，令牌将被点火，在这种情况中，令牌同步了两个事件。在令牌被点火后其结果如图下部分，其中一个令牌出现在垂直栏的右侧。

面向对象规格。迄今为止的技术都是从功能的角度考查系统。尽管数据的许多方面也被表示，并且尽管系统的实体被视为常有属性的对象，规格或表示描述的时是输入如何转换为输出，每个转换如何被分解为步骤也就是说函数被组在一起，如果他们是一个更高层次函数的执行组成步骤的话；这些步骤能操作不同的数据轴象。

然而，有时将需求写作面向对象而不是功能的规格是更恰当的。面向对象方法能（object-oriented approach）将焦点放在涉及的实体上而非输入输出转换上。较早所考查的数据对象的概念形成了面向对象的基础。但他们能被扩展，问：

- 怎样的数据结构定义一个实体？就是说，实体看起来象什么？
- 实体的状态如何能随时间进化？也就是说，我们能对实体会做什么？
- 实体和处理哪些方面是在整个时间上持久的？

系统中每个实体是一个对象（cobject）。一个方法（method）或操作(operation)，（对象定我的一部分）是一个行为，也或者由对象执行，或者能发生在对象上。只有这些方法能改变对象有的状态，并且一个方法仅能通过发给对象一条消息来激发。

象我们将在第 6 章更详细地看到，有几个要领将面向对象同其他表示形式区分开：封装（encapsulation），类推层次（class hierarchies），继承（inheritance）和多态(polymorphism)封装是这样一种方式，在此方法形成了一道绕着对象的保护性边界是，将对象和发生在其他对象的事物隔离开。也就是说，对象仅能通过他们的方法来操作。

对象能被组织成类层次，以便同一类的所有对象有相同的操作和属性列表。个体对象可能有没的属性，但一个个体对象被认为他所代表的类的一个实例。例如考虑大学生类。每位大学生有一个学号和一个所赚学分小时的集合就象图 4.4 中所看到的。一名特定学生如 FrancecsUKU 是 college student 类的一个实例。

图 4.12 显示了一个类　如何利用概括（gengralization）并让处于较低层次的类从其上在面的类进行继承。这 college student 类是 person 类的一部分，结果每个大学生是一个人，但不是每个人都必须是大学生。任何大学生都继承一个人的属性和方法，而实际上，每一大学生可能有某些其他人没有的属性，如学生学时（credit hours）或学生平均分（grade point average），在 college student 类下面有两个层次。左侧类的集合集中于费率。右侧类集合反映了本科生对研究生身份，这两个集合能结合，以便一个特定的类从两个其他类继承。这称为多重继承（multiple inheritance）。因此，一个为外本科生继承了基侧集合的为外费用属生和方法而从大侧继承了本科生属性和方法

最后，一个方法（method）是多态的（polymorphicvk）如果它为不止一个对象所定义。例如，计算几何图形面积的操作对每种图形都不同；一个三角开面积原计算和圆面积的计算是很不一样的。然而，我们能形成一个几何图形的层次，在那里每个图形（三角形、平么四边形、圆、椭圆等等）都有为它而定义的方法 area。

使用向对象方法捕获需求的优势之一就是对象和方法的描述紧紧地与将建立的软件的应用领域联系起来。也就是说，我们可这样捕获要求：请我们的顾客和相关用户来告诉我们他们操作的对象和操作能以之发生的方式。几种需求符号和技术，如 OMT（对象建模技术），使用仔细地建模来获得这种信息。例如，MOT 要求我们建立三个模型：对象模型，动态模型和一个功能模型。动态建模类似于本章已看见过的状态图解。功能建模使用下节将研究的数据流图。很多 00 技术利用象这样的被很好证明了的概念，而应用时侧重点不同。在第 6 章我们通过一个详细的例子来看 00 是如何用于设计和开发、以及需求的整个文本书我们将看到 00。对组织问题和设计的解决方案是一个吸引人的选择

4.5 额外的需求表示法

我们已经考查了几种表达需求的方式。有更多的方式，并且用于定义说明需求的方法的详细尽的讨论已超出了本书范围。但本节，我们提供几个额外表示法，以展示给你，特是方法已经如何被设计以反映某此种类系统的特别属性。每种方法帮助组织和标准说明需求的方式。

层次技术（Hierarchical Teehnipues）

有几种方式描述与我们正尽力捕捉的需求相关的层次。我们已见过几个涉及用方框和箭头来显示数据或功能间关系的描述的例子 Warnier 图包含型一个类似技术，使用的是一颗树，权值项用共括号（{）和特定符号这连接。尽管大多层次数据结构图是自上至下显示数据层次。一幅 Warnier 图通过从左到右显示数据层次。

左一幅 Warnier 图中，当一个名字出现在花括号的左侧时，那么花括号右部的部分就定义了这个数据的整个结构，操作符指示数据类型或者是串接的或者是相互排斥的。例如，图 4.13 中 Warnier 图显示了不同药品的信息。“ ”指示，药方的集合和未经医生处方可以买到的产品集合相互是排斥的。一个实心圆（没有显示）通常用作指示器来显示两种类型数据被串接。当一种数据类型或可能有多分拷贝时，可能的重复数通常放在此数据名字后的圆括号中，例如，图中圆括号中的数字显示药物类别重点的次数。

数据流图（Data Flow Diahrams）

到此为止，我们已讨论的技术已被显示如何组织数据和处理。然而，我们没有解释数据怎样流进，通过、流出系统。为了展示数据的流动，我们能使用数据流图。象许多别的技术，通过分层来表达层次，以便不同级的细节显示在不同层。我们开始是把系统看作一个数据转换机。图显示了流进系统的数据，它们如何被转换，如何离开系统。重点总是放在数据的流动上而非控制的流动上。象图 4.14 所示，输入是有指向的箭头，

输出是离开泡的箭头。因此，这个过程描述了输入到输出的转换，并且也由一个泡来表示，箭头显示了数据路径。有时，数据放于一个数据放于一个数据存储里——一个正式的仓储库或者信息数据库中。在此情况下，数据存储由两条平行栏代表，如图 4.14(b)所示。

图 4.15 显示了一个在一次典型的看病过程中所涉及的数据流。正如你能看到的，大部

分数据路径既是一个转换的输出，也是另一个转换的输入。在三种情况中，需要来自外部数据源即数据存储的数据，而不是一个处理的结果。此图也在图中引入了新项：在加工中表示行动者的矩形。一个行动者（actor）是一个提供或接收数据的实体；在我们的例子中，医生和病人是行动者。因为这个数据流图描绘了一次看医生的高层视图，这样处理是恰当的：每个处理泡将表示为一个分离的图以显示数据转换是如何发生的。

软件需求工程方法论（Software Requirements Engineering Methodology）

一个实时系统是生成需求最困难的系统之一，因为通常要有许多约束要文档化并被跟踪。为了描述这种需要，TRW 公司开发了一个软件需求工程方法论（SREM）

图 4.14 数据流图中的符号

图 4.15 看病的数据流图

SREM 有两部分，一个用于规格，一个用于分析和汇报。

要使用 SREM，我们从用需求陈述语言（Reqruiement Statement Language,RSL）写下需求开始。然后，这份陈述由软件工程确认系统（Software Engineering Validaation System）来分析。一个系统从对象和它们的关系来描述，RSL 描述处理流，根据什么事件激发了哪些处理。这些流的表示成网络，既使用图片也使用书面描述。每个网络或 R-网，说明了一个特定状态和单输入是如何被转换为一个带有输出信息集合的新状态的。通过用网络的格式，SREM 让我们能描述在给定时间上不止一个处理发生时网络里所发生的事；然而，在一个时刻只有一个 R-网是活动的。图 4.16 显示了对一个涉及联机银行系统的过程。网络看起来可能的样子 “ ”指示了处理分支的条件。在我们的例子中，或者右边或者左边的路径被选取。“ ”指示接着的处理能并行或以任何顺序执行。三角形指示了同步点，在那里，所要并行处理在下一处理开始前必须要完成。

图 4.16 需求网络

一旦我们画出了网络图，我们能将每幅图的组成部分译成相应的 RSL 陈述。例如，图 4.16 所描绘的 R-网能用 RSL 写成如下：

P 系 R-的定义外，我们能定义别的无素。Alpha 是一份 R-网中函数的规格，对每个函数，我们都包括了函数输入、输出、及转换的描述。类似的，alpha 也有元素（包括每个无素的域），每个元素来源于哪里，有一般的描述。R-网展示的仅仅是功能的系统需求。我们将非功能的需求考虑为沿各条路径，放置于注解上的约束的描述。例如，在图 4.16 中，顾客可能要求帐户余额应在帐户一位后的五秒里被打印出来。为说明这种需求，我们用确认点

（Validation point）标记 R-网。

一个确认点（Validation piont）是图中用于表示一个度量开始和结束的位置。在我们的例子中，我们标记“查找帐户记录”块为一个确认点，“打印余额”块为另一个。然后，我们“五秒”需求就是从一个确认点到另一个的说明符号。我们在 SREM 中使用这种普通方法，并将非功能的需求表达为能过 R-网的路径的说明符号。

RSL 无素让我们通过设置一个从初始需求到所产生的数据和处理的指针将需求与数据元素，与需求定义联系起来。这个特征对配置管理尤其有用，当需求变化时，可以通过看相应的必须改变的特定系统对象来评估这种变化的冲击。

在我们已使用 RSL 将需求翻译成一份关于数元素处理步骤及它们相关的功能的和非功能的需求的精确描述之后，REVS 系统读取 RSL 陈述作为输入。REVS 翻译 RSL 陈述并从它们形成一个数据库。这个数据库称为抽象系统语义模型（bstract System Semantic Model ASSM）,还辅有一套工具用来分析各组成部分并产生各种报告。报告有两种类型。首先，REVS 产生总结报告好让我们考虑后备方案，并评估它们之中的机会。其次，REVS 模拟系统的临界处理需求，让我们分析开发时系统的可伸缩性。用一个图形色来描述通过系统的数据流，而一个模拟器建立并运行系统的模拟模型。表 4.4 显示了有关使用 SREM 的步骤。

SREM 把系统看作一台有穷状态机。SREM 的一个增强（被称为系统需求方法，Systems Requi）向这个概念中补充了一个时间维。时间的补充让我们能说明事件或同时事件的顺序。我们也能根据对激励的响应来描述性能和功能。SYSREM 将 SREMR 扩展成为一种有用的设计工具。

SYSREM 和 SREM 二者有很多优点，首先，使用 RSL 来将需求翻译成一个详细的活动和数据描述的集合是相对容易。其次，因为系统被分成无关联的片，对每一片的接口都可进行完全性检查。REVS 评估后备方法并摸拟后备需求集合来考查系统的灵活性。这种方法很适合将内嵌于别的更大的系统的系统。

表 4.4SREM 步骤

阶段	焦点	标准
定义一个内核	确定输入输出，R-网，转换	得到反处理的所有输入信息
建立基准	清除数据库，设计 R-风	所有输出信息都产生了的有部分都一
确定数据	确定每个转换的输入输出	在赋值前没有任何数据被使用
建立可跟踪性	产生一致的可跟踪性需求	所有顶层需求都得到满足
模拟功能性	模拟被执行的子系统功能	确认所有被正确处理
确定性能需求	确定要跟踪，可测试的性能上子系统需求	每条路戏爱到响应时间和精确性的限制
险证灵活性	所有临界算法的快速原型	精确的需求由原型算法满足

结构化分析和设计

许多要求定义工具涉及一个系统的图形表示，它也能用来获取设计。典型的表示法就是结构分析和设计技术（SADT），也作为美国防部的 IDEFO 而出名。

这项技术真正地由两部分组成：结构分析（SA），接着设计（DT）。SA 用两种类型的图来说胆需求，然后 DT 的说明如何解释结果。

SA 用一个有序图形集合来表示系统。每个图代表一个转换，最后 6 个图被用来描述一项功能。（如果需要多于 6 个图，那么这个功能应重新定义为一套子功能）。图有 2 个因素：

输入，控制机制，和输出。输入是被转换为输出的数据项。控制是那些诸如预算和进度的用来限制项描述的处理的类型或程度项；机械是对处理的外来帮助，诸如用来执行转换的工具和技术。因此，每个图画成有三个进入前头号一个进出箭的块。如图 4.17 所示。象许多其他方法一样，这些图被按排在一个层产供销体中以便在更低层次显示更多细节。因为图形指示了系统的结构和关系，我们说 SADT 产生了一个系统蓝图。

为了弄清 AS 如何工作，考虑图 4.18，也说明了一个税金计算系统的顶层视图，输入包括一个人的年收入，扣除额（如不用付税的慈善捐助）和任务史（如前些年一直执行的扣除额）。约束包括税代码，告诉我们要付税的东西，缴税最后期限。机械可能包括一个税金数据库，告诉我们要付税的东西，缴锐最后期限。机械可能包括一个税金数据库，库中含有不同义务级别的信息（如在收入的前$10000，0%税金，在第二个$10000：15%税金等等）还包括一个表格数据库，含有各种政府所需不同标准的表格。输出是完成的表格和应付数量，以便人们能安排付款补充表格。

高层图然后又重写为几个低层图。例如，这个标有“计算应付税”的框可能分为三个附属框：计算来自各种来洙的收入，计算扣除额，计算税金。三个活动中每一个都与输入，输出，控制和机械相联系。同样，计算扣除额用这种方式，我们建起了一个活动层次体系，描述了系统需要采取的步骤。这种系统描述对和顾客交流很有帮助。这种表示法易于理解，并且能在任保对顾客恰当的层次一讨论需求。例如，相比于只仅仅看自己该付的税金的典型公民，会计师可能在较低的细节层看系统。SADT 在描述软件开发过程中活动方面也是有用的。

有几种称为形式规格语言的语言，以数学方式来表达需求，以便能使用证明和有时自动技术来评价之。那样一门语言提供一种表示法（它的语法域）一个又像范围（它的语法域）和一个精确的定义哪些对象满足每一规格的规则。规格是根据语法域的元素所写的句子。我们本章前面所见的 Back-Naur 范式就是一个形式规格的例子。

Z 是一种形式需求规格语言，也把抽象数据建模与集合理论，一阶谓词逻辑结合起来。也能用来说明系统状态和有效的状态变化，并有自动工具来检查规格的不完全和不一致。工具发现可达状态检查，死锁和不确定因素，并产生一个实现规格的有穷状态机。下面的例子显示了一个以 z 语言说明的符号表。未有“,”号的变量（unprimed variables）代表执行操作前的状态，加“,”号的代表执行操作后的变量。

形式规格被许多建立安全临界系统（如，那些其失败将事关用者或者说附近人士安危的系统）的软件工程师鼓励的使用。

例如，一份有安全临界系统的草拟的英国标准要求使用形式规格和设计。提倡使用表式规格暗示着可形式化强迫分析使需求一致，数学证明技术已经揭示了需求中的重大问题，而在这里纠正比在代码实现后纠正要容易得多。

需求表示方法的其他特征

有许多需求获取技术。一些包括了将不确定或风险的程度和每一需求联系起来的设备。另外的允许跟踪别的系统文档，如设计或编码，或别的系统。（如当应用需求时）。大多的规格技术已在一定程度上实现自动化了，使得一些工作容易：画图、将规格一数据字典捆绑、检查明显的不一致。随着一些辅助软件工程活动的工具不断开发出来，文档化跟踪需求将更加容易。然而，需求分析最困难的部分——理解顾客需求——仍然是人为努力。

4.6 原型化需求

当顾客和我们一同确定需求时，有时他们也不确定真正需要什么。需求分析员可能产生

一个顾客想看到的东西的希望列表，但列表是否完全不清楚。在一些形势下，顾客或用户直接涉及分析和设计过程，因而我们能提供可用选项，当顾客对选项作了反应时，我们再修改需求，在别的情况下，顾客知道需求什么，但我们不能确定顾客的问题是否有一个灵活的解决方案。有两种原型化方法：进化（evolutionary）和抛弃（throw-away）抛弃原型（throw –away prototype）是这样一种软件：开发出来以更多地了解问题或探究可能的方案的灵活性或合理性。一个抛弃原型是一个探索物，并不用于被交付软件的实际部分，另一方面，进化原型（evdueionary prdotype）开发了来了解问题，并形成被交付软件的部分或全部的基础。例如，如果顾客不能确定对他们系统想要何种用户界面，你可建几个进化原型一旦顾客选定一种，那么这个原型便可开发成实际界面并与产品其余部分一同交付。

两种技术有时被称作：“快速原型法（rapid prototy ping）”，因为他们建立起目标的一部分的以确定需求的必要性，合理性工灵活性。“ropid”将原型和另一工程中所用的原型区分开。在那里设计完成后，子系统的小系统建立起来。在快速原型法中，在设计建立前选择已被评估，快速原型的目的是帮助我们理解需求，决定一个最终设计。

让我们考虑一个原型如何起帮助作用的例子。假定我们正在建立一个工具跟踪每天用户锻炼情况。顾客就是一锻炼生理学家和教练；工具将帮助他们同客户工作并记录进展。用户界面重要的因为用户不总是熟悉计算机。

系统将要求用户输入每次例行煅炼的日期，教练不能确定怎样说明界面，因此我们建立快速原型证实入口屏幕看起来可能的样子。图 4.19 描绘了这一屏幕；你能看一通知用户将输入年：月：日。

然而界面可能性做得更加有趣灵巧。图 4.20 显示了一各含有日历的界面；在这里，用户使用鼠标从左至右滑动每个条，屏幕框中变化的显示选中的年，月，日这种界在提供了最快的选择，尽管它和用户习惯所见的不同。

三副图难以用单词或符号来描述，它们显示了一些需求用图或原型表示时是如何发现更好的。原型法帮助我们为系统与用户的交互选择正确的“look and feel”。然而，性能和效率问题仍要必须提及；这些功能的或行为的需求也能涉及原型，例如，有时界面的选择影响到系统响应的速度，或用于实现解决方案的算法。（图 4.19,图 4.20,图 4.21,见 P170）

4.7 需求文档

无信纸选择什么方法定义需求，必须要有一套文档记录结果。我们和顾客参照需求，贯穿整个开发和维护。必须写下需求因为对顾客对设计员都有意义。

书写需求的水平也很重要，就正如补充栏 4.6 所说明的。而且需求也必须以整个系统开发阶段可跟踪的方式来组织。随文档的清晰准确的图解图示应和文档保持一致给需求编号让我们可通过数据字典和虽的支持文档来交叉引用它们。编号机制对管理团队也是很重要的。如果在其他开发阶段需求做了一些变动，则能从需求文档，经设计过程和所有途径，一直到测试程序跟踪这些变致力。理想情况下，系统任何特征或功能能被跟踪到它的源头需求，反之亦然。

需求定义文档

包含的是以顾客的术语说明的需求记录。描述了顾客想看到的东西。

首先，简述总体目标。包括对别的相关系统的引用，并加入任何有用的术语、略语。接着，描述系统开发及背景和目标。例如，如果一个系统将替代现存方法，我们要解释为什么现存系统让人不满意。当前的方法和程序（proceelure）被概括时要够详细，以便我们将

顾客满意的和不满意的元素分离开。

如果顾客有一个解决此问题的推荐的新方法，我们概括要描述此方未能，然而要记住需求文档的目标是讨论问题而不是解决方案；焦点要放在系统将怎样满足顾客需要上。特别地顾客对开发有什么约束或做了特定假设，定义文档要列下它们。

一旦记录了问题的概要，我们要描述目标系统的详细特征。我们定义系统边界和接口解释系统功能。完全地引出数据元素类及它们的特征。我们细化数据和函数间的关系，还有每个处理或函数的输入输出不冤屈括特定性能需求，如时间，精确度，对失败的反应。

最后，讨论系统运作的环境。那些关于支撑、安全、隐私、和任何特别软硬件的约束的需求都应得到表达。

4.8 需求过程的参与者

有很多人都是需求集合的贡献者。他们从各自观点来看待系统，这些观点常有冲突。分析员的众多技巧之一就是能理解每个观点，获取需求时采取一种反映每个参与者的关注点的方式。

需求过程中可能包含以下参与者：

- 合同监督人员，提出里程不写和进度
- 顾客和用户，必须理解需求，确信满足了他们的需要。
- 业务经理必须理解建立和使用系统的可能的后果
- 设计员，使用需求作为开发可接受解决方案的依据
- 测试员，开发测试数据和测试事务的确保软件系统满足每一条需求

4.9 需求确认（Requirement Validation）

需求分析员有两种目的：1、提供一种方式一让顾客和开发者对系统将要做些什么达成一致；2、规格为设计员提供指导原则。因此，在需求交付设计员前，我们和顾客都要绝对了解对方的意图和意思。为建立起这种肯定，我们确认需求确保需求满足了顾客需要。

确认通常有两步，都是确保二种需求文档间的可跟踪性。首先，确保每种规格能跟踪到需求定义文档中的一条需求；其次，检查定义文档看是否每一条需求能跟踪到规格。表 4.6 列出了一些能用于执行确认的技术；你选择哪一技术依赖于你的经验，喜好与文档定义规格技术的适合性。

确认不仅仅检查可跟踪性。为确保系统将做顾客和用户其望它做的事，我们还必须证实顾客和用户的目的已经达到。否则，设计员将建立起一个非顾客想想要的系统。检察需求的一个简单方法就是进行需求复查。在复查中，来自我方和顾客的代表检查需求列表。代表包括将操作系统的人员，准备输入的人员，使用输出的人员。我们提供的是设计小组，测试小组，和配置管理组的成员

需求复查要做些什么？

1．我们复查系统的确定的目标

2．把需求同系统目标比较以验证所有需求的必要性

3．描述完全的，然后能再次复查系统信息流和结构以确保需求精确的地反映了顾客的意图，系统功能应与顾客的范围和意图一致。而且，功能和约束应是实际的，并且在我们的开发能力之内。再次检查漏遗，不完全处，不一致处。

表 4.6

4. 对任何开发或运行中将出现的风险进行评估并文档化，我们讨论并比较备选取方案，并和顾客在使用何种方法上达成一致。

5. 探讨关于测试系统的问题。在开发进行中并且需求变化或增加等，如何来继续实验确认系统？测试小组如何测试来弄清是否所有需求已正确实现？谁提供测试数据？如果系统将有一阶段实现，在中间阶段如何检查需求？

无论何时发现问题，复查小组记下它，考查也它的原因，在设计之前采取措施解决它。

也可使用工具一帮助需求复查。

当需求复查完成后，我们和顾客都应对需求规格觉得舒适。理解了顾客想要的东西后，我们能继续系统设计。需求复查完成时，顾客手头已有了一份正确描述系统将做什么的文档。

4.10 度量（测量）需求

有很多度量需求特征的方式，以便收集的信息告诉我们许多关于需求过程和需求质量方面的东西。度量常关注三个方面：产品、过程、资原。常常通过需求的数目一评估需求产品，当需求集合增长时，我们能很好地感受到所一切系统可能有多大。

类似地我们能度量需求变更的数目。大量变动，暗示了我们对系统应做什么的理解还不稳定；我们可采取措施以保将来更数目维持在尽可能实际的水平。整个开发中也能继续跟踪变化；当系统需求变化是那些变化的影响也能评估出来。

在可能的部分，需求尺寸度量能按需求类型记录下来。这样归类让我们了解到需求中的变更或不确定是广泛存在的，还是只是涉及某类需求（如用户界面或数据库需求）。这样的信息允许我们采取步骤增加理解并减少特定类型需求中的不确定。

因为设计员测试要使用需求，我们利用度量以反映何时将需求转交给他们。例如：我们请设计员来为每一需求定一级（共五级）。

如果你定设计员

1、意思是你完全理解了需求，你以前从类似的需求进行过设计，

2、此需求有新的元素，但与以前没有大的差别

3、有一些与以前所见的需求大一相同的元素，但你理解他并认为你能开发出很好的设计。

4、有部分你不理解，你不能确信你能开发出好的设计

5、你根本不理解需求，你不能开发设计

如果你是测试员

1、意味着，你完全理解需求，以前测实过该类似需求，现在再测试编码，应该没问题。

2、有新东西，但与以前没什么大的不同。

3、意味着有与以前大不一样的元素，但你理解并认为能测试它。

4、有部分需求你不理解，你不能确定你能设计一个测试

5、根本不理解需求，你不能测试

在每种情况下，评价完成后再看一下需求的与描写。如果设计员和测试员产生了大中分是第 1 第 2 种，如图 4.22(a),那么需求是好的,并能继续传递给设计小组,然而,如果许多都是第 4 第 5 象图 4.22(b)所示,就不应进行设计,而要重新评估.

我们也能提出每一需求何等实现设计,实现编码,，实现测试.这些度量告诉我们正向完成

迈进的进展,测试中也能度量他们的测试案例的彻底性.我们能度量每一测试案例所覆盖的需求的数目,以及已被确认的需求的数目。

图 4.22 度量需求稳定性

4.11 选择一个需求规格技术

没有一种技术对所有项目都是最好的。因此，要有一个标准用来确定对某次项目，哪个技术是最合适的。

我们考虑在那样的标准集合中应该抓住的问题。假设我们正在建立一个调用算机系统来监测一个大流域或湖泊河流的水质。在要监测的位置安装监测设备。某些处理就地执行，但收集的数据和就地处理的结果被传到一个中央位置作进一步分析这个水质监测系统的关键特征之一就是它是一个内嵌系统，就是说，系统的数据处理部分被内嵌于一个数据收集和分析设备的复合体中。另外系统包含大量功能，其处理分析分布于几台计算机中。这个系统的复杂性使得正确地完全地说明需求，非常重要。必须仔细定义接口，需求应提供足供的信息，以便测试小组知道怎样验证系统正确地执行任务。需求规格阶段原任何含糊将导致测试开始时的巨大困难。在这里，一些技术可能比另一些工作得更好。自动方法可能优于手工方法。而且，那些检查系统一致性和完全性的技术可能会捕捉到规格中的错误码率，而这些错误在别的地方就不容易认出。如果系统有实时需求，那我们必须寻找允许我们在规格中包含时间角色的技术。而且任何对分阶段开发的需要告诉我们将要经历几个中间系统跟踪需求。这不仅仅增加了跟踪需求的因难，而且也增加了系统生命中修改需老谋深算的可能性。

当用户与中间版本系统工作时，他们可能看到需要增加新项、改变功能、加入约束。因此，我们需要一个能很容易地处理变化的灵巧的方法。为了确保需求具备本章之初所列的那些合意的特征我们寻找这样一处方法，该方法让我们可修正需求，跟踪变更、交叉参照数据和功能项，并能分析需求的尽可能多的特征。模拟系统或一个子系统的能力也是一份可取的，因为开发实际涉及一系列不断增加功能的子系统的开发。

Ardis 和他的同事已构建了一个用了评价规格方法的标准集合。每一条标准与一相问题列表相联系，这些问题应予以回答以确定是否达到了这条标准。

应用性：此技术能以一种自然实际的方式描述真实世界问题和解决方案么？技术的假设合乎情理么？此技术能与用于本工程的其他技术兼容？

可实现性：规格能容易地提炼或转化成一个实现么？这种转换难度如何？是自动的么？如果代码是从规格自动产生，生成一代码效率如何？在机器生成的代码和手工生成的部分有细致定义了的接口么？

可测试性模拟：能使用规格试实现么？规格中每条句子都能由实现来测试么?

可检查性：理解被解决的潜在问题的某人能检查的情况性么？规格对领域专家而非技术专家来说可读么？有自动规格检查器么？

可维护性：规格对维护活动是有用？当系统进化时改变规格容易么？

可模块化：此方法允许大规格被分解成易理解的较小部分么？能够在不重写整个规格的情况下修改较小部分？

抽象层次 / 可表达性：从用户的视点，规格元素能多严密，多明白地描述用户域中实对象行为，和环境？

健全性（Sowndness）：我们能或者手工地或在工具支持下，检察规格的不一致或含糊之处？规格语言义被正确地定义了么？

可险证性：我们能正式地论证需求满足每个抽象屋次所描述的特性么？验证过程能是自动的么？如果是，自动化容易么？

运行时安全：如果代码能从规格自动产生，那么代码会在没有预料条件下不会急剧恶化吧？

工具成熟能力：对任何支持规格技术的工具来说，它们是高质量的么？有培训可用来帮助学习使用它们么？工具用户要多大规模？

松散度：规格能是不完全或容许不确定么？

学习曲线：新用户能快学会此技术概念，语法语义和启发式的论据？

技术成熟度：技术已发给证书或标准了？有一个用户群或大的用户基础么？

数据建模：技术表示数据，关系，或抽象？表示是统一的么？

纪律：技术强迫用户写结构清晰，易理解的规格么？

一个特定系统能通过运用这些问题并定其级别为强，充分弱或不能应用，、来达到评估。例如，对于一个开关系统应用来说， A rdis 和同事给 z 的评级如下，可模块化，抽象，可验证性，松散度，技术成熟度和数据建模为强（strong）；可应用性，可检测情形，可维护性，健全性和可测试性／模拟为弱（weak）。他们也指出了哪些标准对开发生产周期的阶段重要。如表 4.7。K 是需求阶段， D 是设计阶段， I 是实现阶段， T 是测试阶段， M 是维护阶段，O 表示其了阶段。

（4.7 见 P 18 1 ）

因为没有一种方法普遍适用于所有系统，在一些情况下必须组合运用几种方法来完全地定义需求。

4.12 信息系统例子

回顾皮克迪星（Piccadilly）例子，涉及 Piccadilly 电视在其特权区出售广告时用间。第 1 章部分内容图包括广告活动。我们可用几个规格技术来描写有关运的需求，首先画数据流图表示一个典型运动中的事件和响应结果可能看起来象图 4.23，注意此图显示了数据存储和数据流。这种描述抓住了很多系统的本质关系，但我们对图中每一项需要更多的信息。

我们能使灵数据字典描述过程模型中实体。例如，广告活动本身要能在数据字典中记为：

活动编号=实体。记录一次某产品的广告活动的条件和援助。

活动预测收视率+活动开始日期+活动结束日期

+目标观众+目标收视百分率

+活动预测收视率+活动预测算总额

+ Piccadilly 预算总数+活动持续时间

+城要的场地持续时间

而目标观众（target audience）可能看起来如下：

目标观众=数据元素一次活动瞄准的观众。见观众类型的值。

观众类型=数据元素　　用于分类收视率图

星号间是关于数据，包括一个表示被描述对像类型的标签：数据元素关系，实体，数据存储数据流，或数据元素分组的注释。

代替我们可能使用面向对规格。许多数据元素可能与我们早先数据分析中描述的相同或

类似，但我们将对象的特征行为传递的消息与对象联系起来。图 4.24 描述了一个典型的 OO 描述。

图关于 PiccadillyOO 规格

4.13 实时例子

回顾起 Ariane 与爆炸是由于头自 Ariane-4 代码部分的重用引起。Nuseibeh 从需求重用的观点分析了问题。也说是说，许多软件工程师觉得从以前开发的系统重用需求规格（和他们相关设计、代码和测试用例）能获得很大利益。通过寻找功能或行为相同或相似的需求来确定修改规格，然后在需要的地方修改。在 Ariane-4 案例中，参照系统 SRI 执行了许多 Ariane 与所需要的功能。

然而，Nuseibeh 提出，尽管所需功能与 Ariane-4 中的相似，但是 Ariane-5 有一些大不相同的方面。特别地，在 Ariane-4 需求。也就是说，需求确认在防止火箭的爆炸中本已经能扮演一个极重要的角色。

再一次考虑由 Ardis 和同事提出的用来选择规格语言的标准列表。这个列表了对确定诸如 Ariane-5 这样的系统特别重要的两项：可测试性模拟和运行时要全（run-tine safe）。在 Ardis 的研究中考察了 7 种规格语言对每一条标准的适合性：Modechart,VFSM,Esterel,Lotos,Z,SDL 和 C。只须 SDL 在可测试性模型和运行时安全方面被评为“强壮（strong）级。SDL 是一个包含面向对象概念的成熟的形式方法。商业工具可用于支技 SDL 语名的设计、调试和维护。一个典型 SDL 模型看起来如下：

STATE SEL_WORKING：

- （详细见 P185）

ENDSTATE.SEL_WORDING

SDL 模型能以几种方式构建来描述一个特定行为。例如，过程状态能用一个状态标识和由过程存储的永久性数据来表示。事件也有一个标识和数据参数。额外信息能被建成标识或数据。为确认（validate）一个 SDL 模型，系统需求能写成临时逻辑常量，象 Ardis 和同事所做的：

CLAIM；

- （见 P186）

ENDCLAIM；

因此；一个可能的防护技术辅以工个支技将已经是象 SDL 这样的规格方法的用途。

另一个防护措施可能是需求的模型。一个模拟器将已显示出了起飞后 40 秒发生的故障；然后 Ariane-5 的设计将在重用代码嵌入于新火箭中前被改变。需求分析期间采取的措施将导致对 Ariane-4 和 Ariane-5 之间的差异的更大的理解，也导致从根本上解决问题。

4.14.4.15.4.16 省略

4.17 学期项目

FCO 客户已准备了下面对贷款安排系统（Loan Arranger）象大多数其它的需求集合一样，需要以几种方式来检查这个需求集合是否正确，完全，一致。用这里的需求和以前几章关于 Loan Arranger 的补充材料，来评估改进这个需求集。这用本间提供的许多技术——包括需求度量和 Ardis 的列表。必要的时候，用一门需求语言或建模技术来表达需求，以保证系统静态，动态特性被很好地表达。

前提和假设

Loan Arranger 系统假定这几者都存在：贷方（Lenders）、借方(Borrowers)可供选择的贷款(Loans)、有兴趣购买或打贷款的投资方(Investors)

Loan Arranger 系统包含一个一自各种贷方(Lendlers)的贷款（Loans）信息仓储库要能为空。

每隔一定间隔，每个贷方（lender）提供列出了它的贷款(loans)的报告(reports)已经 FCO 购买的贷款将在这些报告上标示出来。

Loan Arranger 仓储库中每笔贷款（Loan）代表着一份和别的贷款打捆扎在一起并被卖出的投资。

Loan Arranger 系统可以供多大 4 名贷款分析员同时使用。

功能需求

高层功能性描述

1．Loan Arranger 系统接收新贷款的贷方发布的每月一次的报告。最近被 FCO 购买了的用于投资业务的贷款将会在报告中作标记。Loan Arranger 系统将使用这份报告信息来更新，可用贷款仓储库。

2．Loan Arranger 系统将接收来自每个贷款方的每月一次的报告，提供有关此贷方发放的贷款的状态的更新，更新信息包括：削减利率（对可调利率的低押）期望贷款的借方的状态好（good），迟（late），或拖欠（default）对在 FCO 业务中的贷款。

Loan Arranger 将更新仓储库中的数据。不在 FCO 投资组中的贷款也被检查以决定一个借方的信誉是否应被更新。FCO 将提供贷方报告的格式，以便所有报告共享一共公格式。

3．贷款分析员能改变单个的数据记录，就象“数据操作（Date Operations）”中描述的一样。

4．所有新数据入仓储库前必须验证有效性（按数据约束“Date Constrins”）中描述的规则

5．贷款分析员能用 Loan Arranger 来确定贷款来卖给特别的投资者。

数据约束（Date Constraints）

1．单个借方可有多笔贷款

2．每个贷方必须有唯一标识

3．每个借方必须有唯一标识

4．每笔贷款必须有至少一名借方

5．每笔贷款量必须在 1000 美元——500000 美元范围

6．基于贷款量，有两型贷款：正常的（<=275，000 美元）和巨大的（>275,000 美元）

7．如果一名借方的所有贷款处于好的荣誉那么此借方被认为处于好信誉。

拖久	拖久（defonlt）
迟	迟（late）

8．一笔贷款或借方能如下变化：good→late, good→default,late →good

或 late→defanlt，一旦一笔贷款或借方处于"defanlt"就不能变为其他。

9．一笔贷款能由 ARM 变为 FM，并可由 FM 变为 ARM

10．一名投资方所要求的利润是一个 0——500 之间的数字。0 代表一打贷款中元任何利润。非零利润表示这笔代款的回报率。如果利润是 X，那么投次方期望在贷款偿清时得到原本投资加上 X%原来投资。因此，如果一打贷款为 1000 美元，投资者期望 40 的回收率，那么投资方希望这打中所有贷款偿清时收 1400 美元。

11．每笔贷款不能在多包中出现。

数据操作（Date Cperations）

1.一名贷款分析员能复查仓储库中一个特定贷出机构、一笔特定贷款或一名特定借方的信息。

2．一名贷款分析员能从一个业务或一打贷款中建立、查看、编辑或删除一笔贷款。

3．当系统读贷方提供报告时一笔贷款被自动加入业务中，只有指定了相关贷方后，系统才能读报告。

4．贷款分析员能建立一个新贷方。

5．贷款分析员仅在某贷方没有任保相关贷款时删除他。

6．贷款分析员能改变贷方联系方式，电话号码，但不能改变他的姓名和标识号。

7．贷款分析员不能更改借方信息。

8．贷款分析员能要求系统以某种标准（常量，利率，清偿日期，借方，贷款类型或贷款是否已标记为包括于某一打中）来排序，搜索或组织贷款信息。组织标准了应包括范围。组织标准也应包括排除，例如所未作标记的贷款，如在 1999 年 1 月 1 日间的贷款。

9．贷款分析员应能以三种格式（文件、屏幕、作为打印报告）的每一种

10．贷款分析员应能在一份报告中请求如下信息：任何贷款属性，贷方，或借方以及这些属性的概括统计（平均，标准偏差，离散图，组织图（histogram））.在一份报告中的信息能被限制到全部信息的一个子集，就像贷款分析员的组织标准所描述的一样。

11. 贷款分析员必须能有 Loan Arranger 建立满足规定投资要求特征的色，

贷款分析员能以手工方式确定这些色。

手工确定必须被包括进此包的贷款的子集，或者通过指明行定贷款或和属性或范围描述他们。提供投资标准给 Loan Arranger，并允许 Loan Arranger 运行贷款包优化请求未选择满足标准的最好的贷款集。组合使用权用上面两方式：先手工或自动化地选一个贷款子集，然后按投资标准优化所选子集。

12．建立一个包有两步。首先，贷款分析员与 Loan Arranger 共同按标准建立一个包，象上面所描述的一样。然后候选包可能被接受，推绝修改。修改一个色意味着贷款分析员可能接受一些而非全部的 Loan Arranger 所提议的包的贷款可能在接受向一包中补充特定贷款。

13．贷款分析员必须能标记那些可能被包括进一个贷款包的贷款，一旦一笔贷款被那样标记了，它就不能用来包括进任何其他包中。如果分析员标记了一笔贷款而并未决定将这包括在这个包中，那么这个标准必须被删除，并且这笔贷款对别的打包决策是可用的。

14．一个候选取包被接受后，不再考虑将他的贷款用于别的包。

15．贷款分析员退出 Loan Arranger 前，所有当前事务必须予以解决。

16．贷款分析员能存取一个投资请求仓储库。这个仓储库可能为空。对每条投资请求，分析员使用请求约束（关于风险、利润和期间）来定义一个包的参数。然后，Loan Arranger 系统确定贷款打包来满足请求约束。

面（接口）和报告需求

1．Loovn Arranger 系统应工作于 UNIX 系统。

2．贷系分析员一次应能看到多条贷款、贷出机构或借方的信息

3．贷系分析员必须能够向前向后移动表示在屏幕上的信息。当信息有太多卷而不适合单屏时，必须通知用户能看到更多信息。

4．系统显示搜索结束时，必须通知用户能看到更多信息。

5．单条输出记录或行从不能在中间中断。

6．一条搜索请求不恰当或非法时必须通知用户。

7．遇到错误时，系统应该将用户返回前一屏。

非功能性需求

1．在一给定时间上顶多 4 名贷款分析使用系统。

2．对任何被显示信息的更新，在添加，更新或删除信息后五秒内刷新信息。

3．从贷款分析员的请求提交后，系统必须在少于 5 秒时间里对请求作出响应。

4．系统必须在 97%的业务工作日里对一名贷款分析员可用。

第 5 章 系统设计

主要内容：

- 总体设计和技术设计
- 设计风格、方法、工具
- 好的设计所具有的特征
- 合理设计
- 设计文档

需求分析过程的结果就是产生两个文档：一个是提供给用户的，以捕获他们的需求，另一个是提供给设计者，以技术形式来解释问题。开发的下一个步骤就是将这些要求转化成解决办法：一个满足用户需求的设计。这一章中，我们要研究做什么和怎样做。

5.1 什么是设计？

设计是将一个实际问题转换成相应的解决办法的主动过程。所谓设计也可以是对一种解决办法的描述。

通过一个实例区分需求与设计，体系结构的设计者可能根据实际需求设计出几种不同的解决方案，都能满足用户需求。这里可能没有最好的设计，用户选择那种设计完全取决于他们自己的喜好。

软件设计也遵循同样的思路。我们根据需求规格说明来定义问题，然后，说明问题的解决办法如果它满足规格说明中的所有需求。在许多例子中，可能的解决方案是无穷多的。

一开始决定下来的解决方案在具体实施的过程中可能要有所变动，根据用户的反馈意见或新的需求还要对最初的规格说明进行修改。同样地，在开发的过程中系统的描述也会有所改变。实际上，用户在开发者的协助下，可以很好地修改最初已经完成的需求分析。

总体设计和技术(technical)设计

设计者必须同时满足用户和系统建立者的要求才能将实际需求转换为可以运行的系统。用户要清楚系统要做什么，而系统建立者要清楚系统是怎样运作的，基于这样的原因，设计实际上是一个两部分的迭代过程。总体设计(或系统设计)：告诉用户系统具体将要做什么。一旦用户同意了这个总体设计，我们会将这个总体设计转换为更加详细的文档。技术设计：让系统建设者了解要解决用户的问题所需要的硬件和软件。确定设计的过程是一个迭代的过程，这是因为设计者对于需求的理解、提出解决办法、测试办法的可行性和为程序员提供设计文档始终处于一个反复的过程中。

如图 5.1，因为要针对不同的用户，所以要用两个设计文档来描述相同的系统。总体设计重点在描述系统的功能上，而技术设计主要是描述系统要采用的形式(form)。

总体设计描述系统时，不采用计算机方面的专业术语，用户完全能够理解。比如：通过系统所列出的一个菜单，用户可了解系统所具备的功能。但是用户并不知道数据是以怎样的形式在系统中存储的，也不会知道系统采用的是何种数据库管理系统。也就是说，用户从总体设计中只能知道系统做什么，而不能获得有关系统如何工作的网络协议和拓扑的信息。总之，总体设计是解决系统做什么的问题，而技术设计则是解决系统怎样做的问题。

有的时候，用户恰巧就是软件开发者，他了解相关的知识。这个时候，可以将总体设计和技术设计合为一个综合设计文档。此外，将两类文档分开而有使他们相关联是很有好处的，这样，可以使一个文档的改动在另一个文档中反映出来。

一个优秀的总体设计应该包含以下特征：

- 它是由用户语言书写的
- 不包括用户不熟悉的专业词汇
- 它描述系统功能
- 独立于实现过程
- 与需求分析文档相关联

与总体设计相比，技术设计主要描述系统的硬件配置、软件需要、人机界面、输入和输出，和网络体系结构等。也就是说，技术设计是系统说明的一个技术层面上的描述，它至少应该包括以下方面：

- 主要的硬件组成的描述和它们的功能
- 软件组成的层次和功能
- 数据结构和数据流

5.2 分解和模块化

设计一个系统就是要确定一组满足特定需求的组件，以及各组件间的接口关系。具体的设计方法由设计者自身的喜好、或是系统所要求的结构或数据所决定。然而每一种设计方法都要涉及某种分解方法：从系统关键元素的顶层描绘开始，然后建立较低层次，看系统的特征和功能将怎样相互适应。

Wasserman(1995)提出的确定设计的五种方法：

1. 模块化分解：在把功能分配到各个组成部分(component)的基础上进行构造。设计者开始于功能的顶层描述上，然后在较低的层次上说明各组成部分是如何组织的以及各部分是如何关联的
2. 面向数据的分解：这种设计是基于外部数据结构的。顶层描述总体的数据结构，而底层描述所包含的数据元素及它们之间的关系这些细节。
3. 面向事件的分解：这种设计是基于系统必须处理的事件和有关事件是怎样改变系统状态的信息。顶层描述是将各种各样的状态分类，而较低层次是描述状态转换是怎样发生的。
4. 有外向内的设计：这种黑盒方法是基于用户对系统的输入。也就是说，顶层描述列出所有可能的用户输入，而较低层次的描述是对于用户的这些输入(及可能产生的输出)，系统要做什么的描述。
5. 面向对象的设计：这种设计将对象类和它们之间的相互关系关联起来。顶层是对每一个对象类型的描述，而在较低层次上是对象的属性、行为的描述和对象是怎样和另一个对象相关联的解释。

这样，从系统的数据描述、事件、用户输入或是顶层的功能描述就能得出最后的设计。下面以面向数据的分解方法为例说明，这种逐层的向下分解是不同于需求描述的，因为它包括有关数据元素的表示和相互联系的信息，而不仅包括数据怎样被操作的信息。

无论是那种设计方法，每一种分解方式都是把设计分成它的合成部分，这就是所谓的模块或组件。我们所说的系统是模块化的，意思是系统的每一个行为都由一个组件完成，并且每个组件的输入输出是非常明确的。我们所说的组件是定义明确的，意思是对组件的所有输入对系统的功能都是必要的，而且所有产生的输出都必定是某个行为的结果。也就是说，如果有一个输入被遗漏了，那么这个组件不能完成所有的功能。此外，“定义明确“还意味着没有不必要的输入，并且每一个输入都用于产生一个输出。最后，组件是定义明确的还意味着每一个输出都是系统功能的一个结果，并且没有一个输入是在不经过组件转换就成为一个输出的。

分解在很多方面都是很有用的。这就是说一个软件体系结构不可能只有一个对总体特征的顶层设计，而没有细节。

如图 5.4 是在一个开发环境中集成了多种不同软件工具的体系结构，即所谓的“烤箱模型”，因为它的各组件部分看起来很像一个面包烤箱。这个参考模型表明，它所包括的几种类型的服务：储存库、数据集成、进程管理和用户界面，还有，它允许已有的工具可以无修改的插入。很明显，这样一个没有更多的分解的顶层的描述对于程序员实现系统是不够的，但是用于和用户、工具销售商进行交流是足够的。

5.3 体系结构风格和策略

Shaw 和 Garlan（1996）建议一个软件的设计从软件的体系结构开始。他们区分了三个设计层次：体系结构、代码设计、和可执行的设计。

1. 体系结构将在需求规格说明中规定的系统性能和将要实现它们的系统组件相联系起来。组件通常即模块，而体系结构是描述它们之间相互联系的，此外，体系结构定义了从子系统中建造系统的操作符。
2. 代码设计涉及到算法和数据结构，其组件是设计语言基本要素，例如：数字、字符、指针和控制线程。依次地，原始的操作符还包括语言的算术和数据操作基本元素，和组成机制（例如：数组、文件和过程）。
3. 可执行的设计是在一个更低的层次上说明代码设计的。它讨论的是内存分配、数据格式、位模式等问题。

自上而下的方法是很有用的，通常先是体系结构的设计，然后是代码设计，最后是可执行的设计。然而，一些研究表明，开发者们总是在当他们对解决办法有更深入的了解时，在几个分解层面上来回反复。比如：一组开发人员开始时决定系统采用表驱动的形式，但是在建立了部分功能的原型后发现，这种表驱动不能满足实时相应的需求，于是他们决定重新设计系统为数组形式，来代替原来的表的形式。类似地，设计者在开发系统的某个方面的时候，他们可能通过和测试者或是程序员的交流，而改变一开始的一些设计来加强系统的维护性等。这样，随着设计者理解的加深和不断的创造性，注定他们要在体系结构、代码设计和可执行的设计者三个层面上来回的反复。

软件的体系结构的风格，在结合它所有组件的时候，涉及它的组件、连接器和约束条件。Shaw 和 Garlan(1996)曾经指出通常有 7 种常用风格：管道和过滤器，对象，隐式请求，层次化，知识库，解释程序和过程控制。理解他们的特征有助于我们确定对于一个给定的系统那一种风格才是最适合的。

管道和过滤器

在一个管道--过滤器系统中，组件是由管道和过滤器组成的。所谓管道就是传送数据流的，包括输入和输出。所谓过滤器就是完成数据从输入到输出的转换的。这种类型的系统中，过滤器是独立的，每一个过滤器不知道系统其他的过滤器的功能。此外，系统输出的正确性不依赖于过滤器的顺序。如果你使用这样一个管道--过滤器系统，那么无论你什么时候编译一个程序，过滤器都是以一个线性的顺序工作：词法分析、句法分析、语义分析和代码生成。

管道--过滤器系统所具有的几个重要的特征是：

- 设计者可以把整个系统对输入输出的影响看成是过滤器的组成部分。
- 既然任意两个过滤器可以连接在一起，那么它们可以很容易地应用于其它系统。
- 系统发展演变很容易，因为新的过滤器可以很容易地加进来，而旧的过滤器也可以很容易地删减下去。
- 由于过滤器的独立性，设计者可以模拟系统行为和分析系统特性，比如吞吐量。
- 允许过滤器的并行执行。

然而，管道和过滤器也有一些缺点，首先，他们鼓励批量处理，且不适合处理交互的应用程序。其次，当两个数据流是相关的时候，系统必须维持它们之间的通讯。第三，过滤器的独立性意味着一些过滤器必须要复制一些由其他过滤器完成的准备功能，这样会影响性能，并使代码十分复杂。

面向对象的设计

在第四章中我们已经看到，需求可以通过对象和他们的抽象类型组织起来。这种设计也可以围绕抽象的数据类型来建立它的组件。也就是说，每一个组件是一种抽象数据类型的实例。因此，一个基于对象的设计必须具备两个重要的特征：一个对象必须保持数据表示的完整性，还有就是数据表示对于其他对象是隐藏的。

与管道--过滤器系统不同的是，一个对象必须能识别其他对象以便于它们之间的通讯，而过滤器是完全独立的。对象间的这种依赖意味着一个对象的改变将影响到所有和它有联系

的其他组件的改变。

隐式请求

隐式请求的设计模型是事件驱动的。基于广播的概念。它不同于直接调用一个过程，而是由一个组件宣布一个或多个事件要发生了。然后，其他组件将一个过程与这些事件联系起来（称这样的过程为注册过程），再由系统调用这些注册过程。在这种类型的系统中，数据交换是通过储存库中的共享数据完成的。这种类型的设计也经常用于包交换网络中或是基于施动者的系统中，或用于数据库中以保持一致性，还用于用户界面中以便分开数据和管理数据的应用程序。

下面举例说明事件驱动：假如在一个调试程序中，发现用户漏掉了一个变量，这时系统宣布“漏掉变量”这个事件，同时调用文本编辑器（这个文本编辑器已经注册到这个事件中），让用户在漏掉的那个变量的那一行进行编辑。

在这种风格的设计中，当一个组件或是系统宣布有一个事件发生时，它并不知道那一个组件将会受到这个事件的影响。基于这样的原因，隐式请求的系统中通常还有一些显式请求。

这种风格的设计对于从其他系统中重用设计组件是特别有用的。因为，任何组件都可以注册到这个事件中。一个重用的组件也可以被加载到系统中，进行注册，且独立于其他组件。类似地，当系统发展或是升级的时候，老的组件可以很容易地被删除，新的组件也可以很容易地加进来。另一方面，这种设计风格的最大的不利就是：缺乏一个组件一定会相应事件的保证。

层次化

层次是分级的，每一层都为它外面的一层提供服务，而且对于它里面的一层来说是一个客户。在一些系统中，每一层都有权进入某些或其他所有层；而在另一些系统中，一个指定的层次只有进入相邻层的权利。这个设计还要包括关于各层间是如何通讯的协议。

让我们来看看，这种类型的系统是如何工作的。如图 5.6 是描绘了一个提供文件安全保障的系统。最内层是密码层，包括加密和解密的功能，用于系统最基本的加密策略。第二层是文件级的接口，即加密和解密一个文件。第三层是密钥管理，它允许一个组件在一份文件上签名，证实签名后，计算出一个散列码作为访问这个文件的权限限制。最后，第四层提供验证，这一层管理一个密码文件，这个文件以加密形式存储，并且要求用户输入身份验证（比如用户名）和密码。

在设计中，用户可以访问系统的不同层次，这完全依赖于需求说明中的要求。比如：如果用户不需要知道有关加密策略的问题，那么他只需访问最外层，提供用户名和密码即可。

这种层次化的体系结构最大的好处就是抽象概念。也就是说，每一个层次都被认为是一个可扩展的抽象级，设计者可以用这些层次将一个问题分解成一系列更抽象的步骤。此外，由于各层次间通讯的限制，当需求改变的时候，很容易增加或修改一个层次。通常，这样的改变只影响到相邻的两个层次。同样，重用一个层次也是非常简单的，只需要在相邻层此间做一些改变。

然而，另一方面，像这样层次化地建立一个系统并不是很容易。

知识库

在知识库中有两种类型的组件：中央数据存储型和集合型组件，通过对它们的操作实现信息的存储、检索和更新。设计这类知识库的难点在于怎样实现两种类型的组件的相互作用。在传统数据库中，处理方法是以输入流的形式，触发进程执行。而在黑板法中，是中央存储控制进程的触发。

图 5.7 是一个典型的黑板体系结构，它包括三个方面：黑板自身、知识来源和控制。黑板包含一个问题解决状态数据的层次，这是依赖于正在设计的应用程序的。知识源是分隔开的知识片，也可以是应用程序，他们仅能通过使用黑板相互作用。控制是由黑板的状态决定

的。当黑板的状态改变的时候，知识源就有所反映。这种黑板体系结构式的设计已经出现在很多系统中，特别是信号处理和模式识别。

很多其他的系统也是依据这种数据库组织的：可重用构件库，大型数据库，搜索引擎等。这种体系结构的好处就是开放性；销售商很容易获得数据表示，可以开发很多工具来访问知识库。但是这个特点也造成了它的一个不利就是，共享数据必须以所有知识源都接受的形式存在，即使知识源本身是极端不同的。

解释程序（翻译机）

一个解释程序就是将伪码转换成实际能执行的代码。它不仅可以用于将程序语言转换成机器代码，还可以用于将任意一种形式编码转换成更明白有用的形式。一个解释程序由四部分组成：

1. 一个用于存放要被解释的伪码的存储器
2. 一个解释引擎，用于转化伪码和模拟它所代表的程序的
3. 解释引擎目前的状态
4. 被模拟的程序目前的状态

虽然这种设计风格只适用于特殊的一类问题，但是我们还是要介绍它，因为它与其它的设计方法有显著的不同。

过程控制

Shaw 和 Garlan(1996)指出过程控制系统与基于功能或基于对象的设计非常不同，它是以在设计中所出现的组件的种类，及它们之间的关系为特征的。过程控制系统的目标就是将过程输出的指定特性维持在某个指定值左右。

大部分基于软件的控制系统，都涉及两种形式的闭循环：反馈和前馈。如图 5.9,一个反馈系统测量出一个控制变量，比如说是温度，然后调整过程使控制变量值在设定点附近。而在前馈循环中，系统通过测量其他过程变量值希望对控制变量以更多的影响。当受控变量的变化推迟时，一个前馈系统比一个反馈系统更适合。

当设计一个过程控制系统时，有很多问题需要讨论：要监控那个变量，要使用什么传感器，要怎样标定它们，以及要怎样处理检测和控制的时钟。此外，Shaw 和 Garlan 推荐此体系结构将控制循环分成三个部分：

•计算元素，从控制策略中将过程分开。这个过程的定义应该包括改变过程变量的机制。而控制算法应该解释出如何决定什么时候和怎样做这些改变。

• 数据元素，包括过程变量（输入、控制和操纵），设定点，和所用的传感器。

• 控制循环策略：开环或闭环，反馈或前馈。

这种体系结构风格的一大优点就是它将功能和对外部干扰的反映割裂开。

其它设计风格

还有一些其他体系风格的设计方案。分布式系统体系结构处理的是一组系统如何进行相互作用的情况。他们通常是根据它们所配置的拓扑结构进行描述的。比如：图 5.10 的环形拓扑结构和星型拓扑结构。

一个流行的分布式系统体系结构就是客户机/服务器形式，由客户机系统提出一个服务的请求，然后由服务器系统响应这个请求。在这种体系结构中，服务器端并不知道有多少客户机将会对它提出请求，也不知道客户机的标识。但是客户机知道服务器的位置，它通过一个过程调用来向服务器发送信息。这种客户机/服务器系统的优点是用户仅在他们需要信息时得到信息。此外，这种设计还详述了表示细节，使得不同的客户机可以以不同的方式察看数据。但是，缺点是这种系统通常需要更复杂的安全保证，系统管理和应用程序的开发，所以他们需要更多的资源去实现和支持。Sidebar5.1 描述的是世界杯客户机/服务器系统的实现。

还有一种特殊领域体系结构，它是利用一些特殊的应用领域，比如航空电子学和自动化制造。这些体系结构利用了应用领域所提供的共同特征，使得一些基本的假定和关系不用再被表示。其长远目标是创建可以被直接执行的设计，或者至少也可以利用那些可重用的组件。

还有一些体系结构是结合了几种不同风格设计的一些方面，来用于特定的子系统。比如说在管道—过滤器系统中，第一层的分解是按管道—过滤器的形式组织的，而在每一个过滤器的设计可能又采用了不同的体系结构方案。

5.4 关于设计的几个问题

对于每种情况而言，没有一种设计风格或方法可以说是最佳的。事实上，当新方法和技术出现的时候，我们必须将它们和现有的方法进行比较。在这一节中，我们就要讨论在选择恰当的设计方法和得出细节时所涉及的几个问题。

模块化和抽象层次

很早的时候我们就注意到，模块性是一个好的设计所应具备的特征。在模块化设计中，每一个组件都被清楚地定义输入和输出，及确定的用途。也就是说，独立地测试一个组件是否完成所要求的功能是很容易的。因此，在设计软件时要尽可能地使用模块化的思想。

抽象层次，是能够帮助我们更好的理解问题，它在较抽象的高层次上隐藏了功能的细节，而在较低的抽象层次上是问题解决办法的详细描述。这样的抽象层次可以改善设计。如Sidebar5.2，就是用抽象的概念来扩展我们的设计的。

Sidebar5.2 的内容如下：

我们用抽象的概念进行设计。假设系统的功能是为列表 L 的元素排序。设计的初始描述如下：

Rearrange L in nondecreasing order

抽象的下一步可能是一个算法：

DO WHILE I is between 1 and (length of L)-1：

…

ENDDO

这个算法提供了很多额外的信息。它告诉我们这个过程用于在 L 上执行排序的操作。然而，他们也可以更细化。第三步和最后一步告诉我们重排操作确切是如何工作的。

DO WHILE I between 1 and (length of L)-1：

…

ENDDO

抽象的每一个层次都是有目的的。如果我们只是关心 L 在重排前后的情况，那么第一层抽象就是我们必须知道的。通过给予我们更多的细节，第二个算法提供了执行重排操作的过程的总的概貌。如果我们只是关心算法的速度，那么第二层抽象就足够了。然而，如果我们要编写重排操作的代码，那么第三层抽象告诉我们确切要做什么。

如果我们现在只有第三层抽象，那么我们可能不能立即看出这个过程是描述了一个重排。有了第一层抽象，过程的本质就很明显了。第三层抽象使你的注意力从过程的本质上移开了。这样，信息隐藏和抽象是我们把注意力放到了一个组件或算法的目的上。

抽象是面向对象设计和信息隐藏的基础。比如，不是直接执行一个栈操作，而是定义一个称为栈的对象，还有执行栈操作的方法，进栈和出栈。我们使用对象和方法，而不是栈本身取操纵栈中的元素。我们也可以定义探针来为我们提供有关栈是空还是满，栈顶是什么的

信息，同时不用改变栈的状态。

同样地，模块化思想也隐藏了细节。所谓信息隐藏，是每一个组件的内部细节和过程，对其他组件来说是不可访问的。

通过结合模块化组件和几个不同抽象层次，我们可以从不同的角度来看系统。最高层次的组件使我们可以在整体上看这个解决办法，它隐藏了那些可能扰乱我们视线的细节。在一个功能分解中，我们所看到的就是一个系统所要完成的主要功能。在一个面向对象的设计中，我们看到的是系统的抽象类型和各种各样的系统对象是如何关联的，而看不到每一个实例。当我们需要有关系统某部分的更多的细节时，我们只需要转向更低的抽象层次即可。

和通常思考问题的方式不同的是，将一个问题分解成几个部分的方法，不能用到将一个复杂的问题转换成一系列简单的问题中来的。但是，模块化允许我们将问题最难处理的部分分离出来；而抽象层次允许我们在细节内容逐渐增多的各层次上慢慢理解问题。通过这种分隔问题的方法，我们可以避免被那些无关的功能和数据所混淆，而走上歧路。

另外，模块化允许我们以不同的方式设计不同的组件。

共同设计

大部分的项目，是由一组人员共同工作进行设计的，通常不同的人被分配整个设计中的不同的部分。但是一些问题是必须注意的，包括谁最适合系统设计的某一部分，怎样写文档才能使得每一个组员都能清楚其他人的设计，怎样协调设计组件使他们像一个统一的整体。设计人员们必须清楚的是哪些原因可能导致设计失败（Sidebar5.3），并用集体的力量克服它。

共同设计时，一个主要的问题就是设计人员个人经验、理解力和喜好的差别。另一个就是，人们有时候在团队中的行为有别于他们个人的行为方式。

Sidebar5.3：设计失败的原因

Guindon,Krasner 和 Curtis(1987)研究了 19 个项目设计者的习惯，得出了引起设计过程失败的主要原因。他们发现主要有三类：缺乏知识，认识的局限，和两者的结合。

设计失败的原因

- 缺乏专业化的设计策略。
- 缺乏一个有关设计过程中导致不佳的资源分配的各种设计行为的元模式。
- 导致多种解决办法中的不佳选择的不好的优化法。
- 在定义一种解决方案时要考虑到所有确定的或是猜想的约束条件的困难。
- 用测试用例或是步骤进行智力模拟的困难。
- 在继续追踪和返回已经延迟的解决办法的自问题上的困难。
- 扩展或合并从单个的子问题中得出的解决办法和整个解决方案的困难。

软件行业追求的正是能够减缩经费和又能在位于世界各地的合作小组的共同开发软件的前提下最大限度地提高生产效率，对于这种分布式开发环境，Yourdon(1994)提出了四个阶段：

1. 第一阶段，一个项目在一个地点进行开发，并有一名来自国外的现场开发人员。
2. 第二阶段，现场分析决定系统需求。然后，将这些需求提供给不在场的分析人员们和程序员继续进行开发。
3. 第三阶段，由不在场的开发人员开发出可在全球使用的通用产品和组件。
4. 第四阶段，不在场的开发人员利用他们个人领域的专业经验开发出最后的产品。

需要注意的是，这个模型同前面章节引用的发现是相反的。当这个模型实现的时候，问题可能出现在第二阶段上，在这个反复的设计过程中，必须一直保持交流。

以注释、原型，图形和更多的形式进行反馈可以增强交流的效果。然而，这些需求与设计的直白的表示必须是无歧义的，并且能够捕获有关系统应该怎样工作的所有假设。Polanyi(1966)指出任何一种语言都不能完整的表述这些意图；有一些细微差别是不易察觉的。这样，如果一个人根据他自己的理解解释信息的时候就会使组员在交流的时候出现问题。

当我们使用多于一种语言进行交流的时候，这个困难还会增加。因为某一事物可能有多种不同的说法。要解释清楚一些细微的差别是极困难的事情。这也就是说，要想产生一个好的软件设计的一个最大的挑战，就是使所有组员在对系统和环境的各自不同的理解方式的基础上达到一个共识。这不仅涉及技术问题的复杂性，还有就是当用户和开发人员进行设计、开发时可能存在的社交问题。

我们必须意识到软件的设计是一个合作的过程，也是一个反复的过程。我们建立的不仅是一个产品，还是与客户、使用者、应用领域和环境，甚至更多方面的共识。

设计用户界面

用户界面是设计中的棘手问题，因为不同的人有不同的认识、理解的风格。而且，不同的人干一件事的顺序也不尽相同，还有它们使用屏幕和键盘的程度也不同。

Marcus(1993)讨论了在界面设计上所设计的一些问题。主要有以下几个关键要素：

- 隐喻：可以被识别、学习的最基本的术语、形象和概念。
- 一个智力模型：数据、功能，任务和作用的组织和表示。
- 模型的定位规则：怎样在数据、功能、动作和作用间移动。
- 看：用于给用户传达信息的系统的表面特征。
- 感觉：为用户提供上诉经验的交互技术。

设计用户界面的目标就是帮助用户尽快获得有关复杂系统的内容。

用户界面可以包含各种各样的技术：代理，超文本，声音，三维画面，视频和虚拟世界。这些的实现可以采用不同的硬件配置。但是为了设计出舒适，高效的界面，我们必须考虑两个主要问题：文化和喜好。

文化问题

在第四章中我们看到，原型在帮助用户和客户决定那一个界面使所需要的方面上是很有用的。同样，原型在设计阶段也可用于测试喜好和决定那种界面类型是可行的和满足实际需求的。但是，使用原型也要考虑到文化差异和可能的用户群。当我们的软件是全球发行的时候，我们不得不考虑可能使用我们这个系统的人的信仰、价值观、传统和其他方面的问题。有些界面的设计者已经在菜单、图标或是数字各始中为用户提供了他们的语言选项。然而，语言的转换不等于保证文化也转换了。况且，有些用户是有多种文化背景的人。因此，用户的文化背景又加大了设计出优秀的用户界面的难度。

为了使我们的系统具有多种文化的特性，我们设计用户界面的时候，分两个步骤：第一步，我们排除一些特有的文化参考或是倾向于尽可能的使我们的界面国际化。Jones et al.(1991)提出了在系统的很多方面减少文化偏见的方针，包括键盘、图标、标签和图例等等。第二步，采用没有偏见（bias-free）的设计，对设计进行加工，使其符合要使用这个软件的用户的文化背景。比如：图标使用的颜色、对象流动的方向等问题。

还有一点要记住的就是，文化不仅仅是由民族决定的，还由地区、性别、年龄、职业等多因素决定的。系统的用户很可能和系统的开发者来自不同的文化背景，因此，开发者如果假设他们很清楚用户的需要，就是十分危险的事情。最好就是由可能使用这个系统的用户对界面进行测试。

用户喜好

设计的很多方面都依赖于用户的喜好。实际上，由于用户的年龄、性别、职业等都不尽相同，所以应用于任何一种文化的用户界面都没有一个统一的标准，也很难描述能够使所有用户都满意的设计方针。一些研究的结果重点放在正被设计的系统中针对特殊目的的观众时原型的重要行。

决定用户界面特征的方针

选择一个用户界面设计的特征涉及到很多综合因素。如图 5.11 表示的是数据输入的方式不同时，用户使用的容易程度也不同。

表 5.1 列出了一些范围，由 Lane 推荐的，这是我们所要考虑的综合因素。在决定用户界面应该被怎样构造和它将怎样完成功能时，表中的每一个条目都要细致地讨论。比如：如果我们选择一个基于菜单界面形式，那么用户可以重复的选择，并且每次选择都会被系统显示出来。而一个命令行形式的界面将涉及一个人造的符号语言，与自然语言相对的。最后，某些系统可能允许用户直接操纵数据。

表 5.1 在平衡分析中要考虑的问题

功能	结构
外部事件处理	应用界面抽象层
• 没有外部事件	• 单块程序
• 当等待输入时处理事件	• 抽象装置
• 外部事件取代用户命令	• 工具包
	• 与固定数据类型的交互管理
	• 与可扩充数据类型的交互管理
	• 可扩充的交互管理
用户可定制性	抽象装置的可变性
• 高	• 理想装置
• 中	• 参数化装置
• 低	• 可变的操作装置
	• 特定装置
跨装置的用户接口适应性	用户接口定义符号
• 无	• 共享用户接口代码中的隐含
• 局部行为改变	• 应用程序代码中的隐含
• 整体行为改变	• 外部说明的符号
• 应用语义改变	• 外部过程的符号
	• 内部说明的符号
	• 内如说明的符号
计算机系统组织	通讯的基础
• 单机处理	• 事件
• 多重处理	• 纯状态
• 分布式处理	• 提示状态
	• 状态加事件
基本接口类	控制线程机制
• 菜单选择	• 无

- 填表
- 命令语言
- 自然语言
- 直接操纵

- 标准程序
- 轻量级程序
- 非抢先程序
- 事件处理程序
- 中断服务程序

跨用户接口风格应用程序可移植性

- 高
- 中
- 低

并发性

很多系统中，行为都是并发的，而不是按顺序地发生。

顺序系统通常用一串简单的执行来控制事件，但是并发系统必须有更复杂设计。并发系统中一个最大的问题就是要保证统一时间执行的组件中的共享数据的一致性。

处理并发的一种方式是给任何一种行为分配访问时间段。这样，准确的时间控制可以保证一种行为不会打扰另一种行为。然而这种定时不总是在系统的控制之下的。特别是对实时的系统，这种系统是对外部事件作出反应的。

针对这种类型的问题，我们采用技术去同步并发过程。同步是一种允许两种行动并行发生，而又不互相干扰的方法。

互斥是使过程同步的一个常见的方式：它保证了当一个过程正在访问一个数据元素的时候，没有其他过程可以影响这个元素。

通常，如果两个操作可以影响一个共享对象的状态，那么他们应该用互斥的策略来执行。同样地，如果一个操作测试了一个对象的一个状态值，那么这个对象应该被锁住，这样，对象的状态在被测试和被采取某行为的期间就不会改变了。

过程或组件的优先级也可以用于解决并行冲突。两个过程或组件中具有较高优先级的一个可以“赢得”战争，将另一个锁在外面，直到这个具有较高优先级的完成它的操作。

定时、同步和过程的优先级策略用并发的正确性来换决定，因为上锁和定时机制取决于要执行哪些要求的顺序。我们希望所设计出来的系统可以独立于定时的要求。幸运地是，还有两种方法可以做到这一点，监视器和监护器。

监视器：是一个抽象的对象或组件，用于控制特定过程的互斥。举例如下：假设 M 是一个指定的过程 P 的监视器，组件 A 要执行 P，则它必须通过 M 获得 P 的访问权。当 A 执行 P 的时候，组件 B 也想执行 P，M 则将 B 挂起直到 A 执行完，然后，M 激活 B，允许它访问 P。为了防止时间控制的问题，通常给监护器补充一个状态检查装置，以保证在状态正确时激活所要求的过程。

监护器：是一项一直在工作的任务；它唯一的目的就是控制对一个内嵌资源的访问。就像一个监视器一样，一个监护器也配有一个状态监测装置，以协助它做出访问控制决定。如果 G 是资源 R 的监护器，任务 T 如果想使用 R 必须向 G 发出一个请求；T 和 G 的交互即所谓的会合。G 立即挂起 T 直到状态检查装置报告说状态满足要求；然后，G 对要求做出相应。这样 G 连续不断地对有效的任务要求做出反应。从 T 的角度看，R 的行为是隐藏的，G 是不确定的。设计的正确性能够独立于时间控制，而被检测出来，直到我们执行这个设计时，我们才不得不决定怎样处理这个不确定的因素。

设计模式和重用

通常，我们现在设计和建立的系统和我们以前建立的那些系统在很多方面有相似之处。或者我们可以设计和建立一系列的有相似功能的应用程序，他们可以在不同的环境中运行。如果我们利用系统的这些共同特征，我们就不用每次都从头开发了。

要识别出这些共同点的一个通用的方法就是查看设计模式。这样当我们进行下一个类似的系统的开发的时候，就可以复用这些模式，还有代码，测试和相关的文档。

一个设计模式可以命名，抽象，识别出一个通常的设计结构的关键方面，这有助于建立一个可复用的设计。设计模式可以识别出包含的类和实例，及他们的作用，合作关系和责任的分布。

正如我们在以后的章节中将要看到的那样，在理想的情况下，我们想要无任何改动的复用设计。但是，通常我们要改变设计以适应当前系统的需求。这样，建立设计模式是很重要的，并且，不要使所设计出来的设计模式太束缚于特定的系统。也就是说，我们要最大程度地复用潜在的而且满足当前系统需求的设计。

5.5 好的设计所具备的特征

当我们设计系统时，我们要保证嵌入一些优良的特性。高质量的设计应该具备哪些导致优良产品的特征，比如：易于理解、易于实现、易于测试、易于修改和从需求规格的正确转换。可修改性是特别重要的，因为需求的修改有时候要导致整个设计的修改。

组件独立性

在整个设计中，抽象和信息隐藏允许我们检验与其它组件相关联的组件。Sidebar5.4 描述的就是在我们的设计中那些组件控制的问题需要被考虑。这样，不仅容易理解一个组件是怎样工作的，更容易修改一个独立的组件。同样地，当发现系统错误的时候，独立的组件可以帮助我们定位原因。

Sidebar5.4 的内容：

在大部分的设计中，我们不得不决定，一个特定的组件到底要控制多少个组件的问题。然后介绍了组件的控制域和作用域的概念，并提出组件的作用域应该在控制域之内，否则，不能保证对这个组件的修改将不会破坏整个设计。还介绍了组件的扇入和扇出的概念。在图 5.18 中给出了同一个系统的两个设计方案，想要讨论一下什么样的设计才是较好的。通常，我们要使具有高扇出的组件个数尽可能的少。另一方面，我们希望在设计中，增加层数，有的时候，某一个特定的组件可能在系统重要被用到不止一次。总的来说。就是希望组件有高扇入数和低扇出数。

我们使用耦合和内聚这两个概念来识别和测定一个组件独立的程度。

耦合：如果我们说两个组件是高耦合的，那么就是说他们之间有很强的依赖性。而松散耦合的组件间虽然有联系，但是程度比较弱。而无耦合组件间根本没有相互联系，他们是完全独立。如图 5.12。

耦合程度取决于几件事情：

- 一个组件对另一个组件的访问。比如：组件 A 可能要调用组件 B，所以说组件 A 要依赖于组件 B 完成它的功能。
- 组件间传递的数据量。
- 一个组件所传递给另一个的控制信息。

• 组件间接口的复杂程度。

图 5.13 是一个耦合度量的级别。

事实上，不可能一个系统是由完全无耦合的组件组成的。那么我们的目标就是使组件间的耦合程度尽可能的低。这就是说，我们要最小化组件间的依赖程度。总的来说就是，如果我们一定要对系统设计进行修改的话，那么松散的耦合会使得所涉及到的要改动的组件数量尽量少。

有一些类型的耦合是我们不希望看到的。比如内容耦合，当一个组件要修改另一个组件的内部数据或是一个组件要进入另一个组件的分支时，内容耦合就发生了，如图 5.14 所示。

我们可以通过对一个公共的数据存储区域的访问来减少耦合的程度。但是这种情况下，依赖关系仍然存在。公共数据的改变就意味着所有对它进行访问的组件都将受到影响。这就是所谓的公共耦合，如图 5.15 所示，表示公共耦合是怎样工作的。

控制耦合是指一个组件对另一个组件传递的参数是用于控制第二个组件的。这就是说，对于受控的组件来说，如果没有第一个组件的指示，它是不可能完成功能的。没有控制耦合的系统的一个优点就是，每一个组件只完成一个功能或只执行一个过程。这个限制使得在组件间传递的控制信息量最少，而且使对形成好的接口的固定的可识别的参数集的控制局部化。

当数据结构本身作为信息在两个组件件传递的时候就发生了标记耦合。如果仅仅是数据被传递的话，就是数据耦合。在标记耦合中，数据的值，格式和组织都必须在两个相互作用的组件中完全符合。如果耦合必须存在的话，那么数据耦合是最理想的。

在面向对象的设计中，组件通常有比较低的耦合，因为每一个对象组件的定义都包括了该组件要实施的行为和施加其上的行为的定义。也就是说，在面向对象的方法中，低耦合是自然而然的事情。

内聚：是指一个组件内部的粘合程度。一个组件的内聚程度越高，它内部元素互相联系的就愈紧密。换句话说，如果一个组件内的所有元素都是为了完成统一任务的话，那么这个组件是内聚的。

一开始哪些层次的定义是基于功能分解的概念。通常设计的目标是使每一个组件尽可能的内聚，这样组件过程的每一个部分都是和组件的单一功能相关联。我们可以改善这个思想以适用于任何类型的分解。如图 5.16 是内聚的几种类型。

最低程度的内聚是偶然内聚，即组件中的某些部分是完全无关的。这种情况下，不相关的功能、过程或是数据被放在同一个组件中。

逻辑内聚的程度稍微高点儿，但也是不希望看到的，即几个逻辑上相关的功能或是数据被置于同一组件中。

一个组件被用于初始化一个系统或一组变量。这样的一个组件要按顺序完成几个功能，但是功能又仅和时间安排有关系，这就是时间内聚。时间内聚和逻辑内聚组件都很难修改，因为这种组件同时要完成几个不同的功能，当要修改其中一个功能的时候，你必须搜索所有的组件找到与这个功能相关的部分。

通常，功能都是以一个确定的顺序执行的。当一个组件中的几个功能被分组的时候，就是为了保证能按这个顺序执行，这就是过程内聚。如果一些特定的功能都是对相同数据集合进行操作或是产生相同的数据集合的话，我们可以把它们关联起来，组件的这种组织方式就是通讯内聚。然而，通讯内聚经常破坏设计的模块化和功能的独立性。

如果组件中一个部分的输出正好是另一个部分的输入的话，就称为顺序内聚。而最理想的还应该是功能内聚，在组件中，每一个处理元素都是为了完成一个单一的功能，并且所有必需的元素都包含在这一个组件中。如图 5.17 是各种类型的内聚的例子。

内聚的概念可已扩展到面向对象中和基于数据和事件的组件设计中，但是要记住的就

是，只有当对象和行为具有共同的目的时，才将它们放到一起。所以，面向对象的设计中，组件通常都有高内聚的特点。

异常识别和处理

我们应该进行防御性设计，预言出可能导致系统问题的情况。

防御设计并不容易。需求规格说明只是告诉我们系统假定要做的事儿，并没有告诉我们系统没有被假定要做的事儿。

所谓异常，是指那些和我们真正想要系统作的事情相反的情况。在设计中我们将包括异常处理部分，这样系统可以处理每一个异常，而不会降低系统的特性。

典型的异常包括：

- 提供服务失败
- 提供错误的服务或数据
- 破坏的数据

对于我们能够发现的每一个异常，我们用以下三种方式中的一种进行处理：

1. 重试：我们让系统恢复到前一个状态，然后用不同的策略重新执行这个服务。
2. 更正：我们让系统恢复到前一个状态，改正系统的一些方面，然后使用相同的策略再执行一次这个服务。
3. 报告：我们让系统恢复到前一个状态，将问题报告给一个错误处理组件，不用再提供相应的服务。

这样，对于我们要系统执行的每一个服务，我们必须识别出它可能出错的方式，还要将它从错误中拯救出来。在后面的章节中我们会看到，我们可以用故障树分析法和失效方法分析来帮助我们识别出这些异常情况。

下面举例说明，怎样将一个异常处理嵌入到设计中。

当代码运行的时候，为了在设计中捕获异常可以用如下几个技术：

- 和检查和数字检查：对数据和运算的双重检查
- 冗余联接：包括向前和向后的指针
- 定时器

故障预防和故障容错

Sidebar5.5 提醒我们软件安全并不是想当然的事情。在设计中，应该尽量预料到故障，并且处理他们，才能最大程度地保证安全，减少混乱。目标就是通过在设计中加入故障预防和故障处理尽量避免故障的出现。我们已经知道如何使用异常处理来解决我们可以事先预料到的不经常出现的情况，同样地，我们也可以防范组件中故障的出现，包括由其他组件、系统和界面带来的故障。

故障是怎样发生的呢？我们犯的一个错误就可能导致软件产品中的一个故障。设计初期的一个对用户需求的误解可能会连续产生几个系统的缺陷。

先区分一下故障和失效。失效可以在系统发布前发现，也可以在发布后发现，这是因为它可能在测试阶段发生，也可能在操作时发生。故障代表开发人员所能看到的问题，而失效代表用户所能看到的问题。

应该说不是每一个故障都相当于要发生失效，因为一个故障导致系统失效的情况不是总能遇到。

好的设计的一个特征就是它防止故障的发生，或是能容忍故障存在。而不是等着系统失灵后再定位问题，设计者能够预料会发生什么，然后设计系统对这些可能发生的事情做出反

应。

主动发现故障。如果我们设计系统时，只是等待系统出现失效的问题再解决，那么就是被动发现故障。然而，如果我们定期检查系统，或是事先预料可能会发生的问题，那就是主动发现故障。比如说：我们采用一个相互怀疑的策略。让两个系统都现假设另一个系统有故障。每一个组件都检查自己输入的正确性和一致性。在这个方式下，一个工资单程序将保证在计算周工资前小时工资数是非负的。此外，故障一经发现就会被处理，而不是等到过程都完成以后再处理。这样及时的故障处理减少故障的危害程度，而不是让故障演变成严重的问题而带来一连串的破坏。

另一个主动发现故障的方法是使用冗余设计，将两个处理的结果对比然后判断他们是否相同。比如：一个计算的程序可以将所有的行加起来，然后将所有的列加起来，以保证总和是相同的。有的系统使用多个计算机，执行完全相同的设计。理论上，如果两个功能相等的系统由两个不同的设计组在不同的时间使用不同的技术开发，那么相同的故障同时发生的可能性就很小。这就是所谓的 n-方案编程，但是由于设计者们开发时都遵循类似的模式和技术，使得它的可靠性并不像前面说的那样。

在主动发现故障中，系统设计通常要合并与第一个系统并行运行的第二个系统。第二个系统要询问第一个系统，并检测系统的数据和查看可能显示问题的标记。这种技术用于一些“从不失灵”的交易处理系统，两个系统并行工作。

如果两个冗余系统的第二个系统仅通过检测相关数据不能发现某种故障，那么它就开始诊断处理。这个技术涉及到第二个系统在第一个系统中产生错误的但是无害的处理，这样第二个系统就可以确定第一个是否正确工作。

故障改正。一旦发现故障，它就必须立即被改正。故障改正是系统对故障存在的补偿。通常，故障改正稳定故障所造成的损害，也修改产品以消除故障。这样我们的设计必须包括故障的处理策略。通常，当故障影响系统时我们终止系统运行或简单地记录下来存在的问题，和当时的系统状态，以后再稳定所造成的损害。

系统的危急程度决定了我们要选择的策略。然而，在医学装置或航空系统中相同的策略是不能考虑的。

在决定故障改正的策略时，系统维护是要考虑的另一个因素。系统活动突然停止比系统继续运行更容易发现问题的来源；继续运行可能会产生效果，它掩盖了潜在的故障。

故障容错。改正一各故障有的时候太昂贵、有风险或是不便的。取而代之，我们可以将故障带来的损坏降低到最低程度，然后使它几乎不给用户带来干扰。故障容错是将一个故障所引起的损害分隔开来，这在很多情况下很便利，甚至是用户所希望的。

一些故障容错的策略取决于预测故障位置和失效时间的能力。为了使系统能一直工作，设计者们必须能够猜测出什么可能出现问题。一些故障很容易预测，但是一些复杂的系统就比较难于分析了。同时，比较复杂的系统更可能存有大的故障。而且实现故障容错的代码可能自身还包含一些问题。这样，也要有故障容错策略用于分离可能的故障区域，而不仅是预测出故障就行了。

5.6 改进设计的手段方法

对于特定的需求，我们可以使用相同的手段。状态表，数据流图，SADT 图等可以用来表达怎样设计而不是需求是什么的记法。开始设计时，我们可以用到上一节中为设计质量所建立的目标的特征，满足这些目标的设计通常就容易建立、测试、改正和维护了，所以我们要尽可能早地将特性加入到系统中去。

减少复杂度

设计时，我们要再没有改变解决办法本质的前提下，尽可能简化结构。随着设计的高质量特性被逐步转化通过开发，软件和硬件就更容易设计、构造和维护了。

比如：系统交互图的复杂程度降低可以使他们更易于理解。我们可以重绘图以减少截面的数量，使他们更简单易于理解。保持关系不变而最小化截面的个数可以证明图形的可平面化。

有时，描述系统行为的决策表也可以降低其复杂程度。下面一个例子就是通过简化表达式来降低复杂度。

类似的方法可以被用于硬件的逻辑设计中，这样既可以增强理解性，也可以降低硬件消耗，并使测试更容易。

合同设计

Meyer(1992a)提出了一个设计方法，称为合同设计，是为了保证一个和设计满足它的规格说明的。他一开始将一个软件系统视为一组通讯组件，他们的交互是基于规格说明的精确定义的。而这些规格说明就称为合同，用于管理每一个组件是怎样和其他组件系统进行交互的。这样的规格说明不能用于保证一个组件的正确性，但是它是进行测试和验证的基础。

Meyer 将一个合同的概念应用于软件。一个软件组件，被称作客户，采用某策略去执行一系列任务 t1,t2,…,tn。当客户调用另一个组件来执行某任务 ti 时,另一个组件就称为供应者。这就是说，在执行某个任务的时候，两个组件间存在一个合同。每个合同涉及双方的责任（称为先决条件），所得到的益处（称为后续条件），和一致性限制（称为不变量）。所有合在一起，这些合同的特性称为断言。

例如，假设客户组件有一张表，表内的每一个元素都是用一个字符串标识的，作为一个关键字。我们的供应者组件的任务就是将表中的一个元素插入到一个有限长度的字典中。我们可以用如下方式描述这两个组件间的合同：

1. 客户组件保证字典是不满的，并且关键字是非空的。
2. 供应者组件纪录表中的元素。
3. 客户组件访问修改过的表。
4. 如果表已满或是关键字为空，则不采取任何行动。

下面一段程序使用面向对象的语言描述这个合同。

可以为所有类型的设计构造合同。假设我们有数据抽象来设计一个面向对象的系统，这个系统是用于控制流入水库和流出大坝的水流的。我们有对象类：大坝、水库和河流。我们可以询问对象问题，比如：水库是否为空，是否为满。

一个描述类的特征的断言称为类不变量，这些特征总是应用于类，甚至当发生改变的时候。所以，当要改动的时候，有必要测试这些改变的正确性。

类和断言可以和随后的执行相比较，在数学中证明他们两个是一致的。此外，断言提供了测试的基础：测试者可以决定每一个的结果。另外，当设计改变的时候，可以检查断言是否削弱或增强了。

原型设计

用户经常提出一些要求，而我们并不能确定我们是否能够实现。比如：一个用户可能要求一个数据库管理系统可以在一个特定的平台上运行；我们不能确定这个平台是否能处理并行访问一类的要求。而开发一个数据库的原型，可以先实现那些要求我们回答的必要的功能。

通过可行的原型，我们可以发现一开始提出的解决办法是否能够真的解决我们手边的问题。这样，原型可以促进开发人员之间的，还有和用户之间的交流。通过这种方式，我们可以在编码之前就解决很多问题，而且还可以避免测试中的一些问题。

一个原型通常遗漏功能上的很多细节问题，以致于我们经常狭隘地将注意力放在系统的某个特定方面。比如说，一组开发人员要使用原型先解决问题的几个方面：一个负责界面，一个负责性能，还有一个负责安全等等。最终的系统是几个单个原型的综合。这样，一个单一的原型通常会充满漏洞，需要后期完善。

如果一个原型仅仅是要证明设计的可行性，那么我们不用给予太多的关注在上面。基于这样的原因，我们不用将功夫花在填补原型的漏洞上，而应该围绕原型进行讨论，来建立实际的系统。这个就是抛弃原型法，意思是，原型的开发仅仅是为了证明系统某些特征的可行性。只要用户能认识到原型只是一个用于探索的模型就行了，它并不是最终的产品，原型对于开发人员和用户之间的交流是非常有帮助的。事实上，Brooks（1975）就提出了先建立一个系统，然后抛弃它，再建立一个，这样开发第二个系统时就可以吸取前一个的经验。

最近的软件工程研究鼓励使用快速原型法，即保留部分原型用于实际的系统中。通过定义和开发较高层次的组件，快速原型法可以回答有关设计的问题，同时它还为最终的系统提供组块。快速原型法将规格说明、设计、实现和测试都具体化了。其缺点就是这个原型的建立过程必须是迅速的，否则不能达到它预期的目标。

当考虑你的项目是否适合用原型法来开发时，有几个因素是要权衡的。Boehm,Gray,和Seewaldt(1984)研究了项目是否适合用原型来开发的问题。他们发现用原型法开发项目，可以少花费 45%的努力，还可以减少 40%的代码。而且，开发出的产品的速度和效率与用传统方法开发出的差不多。

故障树分析法

我们已经知道有关故障识别、改正和容错的重要性。这里有几种手段是用于识别故障和定位它们的。故障树分析法，最初是由美国 Minuteman 导弹程序发展的，可以帮助我们找出可能导致失效的情况。在这个意义上，这个名字可能起错了，因为我们不是要分析故障，而是要找出失效可能的原因。我们建立故障树是要展示从结果到原因的逻辑路径。然后，根据我们所采用的设计策略，这些树要用于支持故障改正或容错。

通过识别可能的失效问题开始我们的分析。我们可以用一组指示词来帮助我们理解系统是怎样背离一开始的初衷的。表 5.3 列出了一些可能会用到的指示词。

表 5.3 识别可能的失效问题的指示词

指示词	解释
no	没有数据或控制信号被发送或接受
more	数据量太大或太快
less	数据量太小或太慢
part of	数据或控制信号不完整
other than	数据或控制信号又另一个分量
early	信号在预定时钟之前到达
late	信号在预定时钟之后到达
before	信号在预定顺序之前到达
after	信号在预定顺序之后到达

下面要画一张图，其节点是失效，或者是单个组件，系统功能，整个组件。图的边表明节点间的关系，每一条边都有一个逻辑描述：“与”表示两个组件都失灵；“或”表示其中一

个必须失灵。有时，边被标记上“n of m”的字样，表示系统涉及 m 个冗余的组件，而其中 n 个失灵的组件导致某指定的失效。画图时一个关键的问题就是每一个节点代表一个独立的事件。

一旦这个图建立起来，我们就可以搜索一下设计中的存在的几类弱点了：

- 失效的单个点，在这里，系统的安全性和完整性依赖于者一个组件。
- 不确定性，对变量值或是转移的情况没有足够的限制
- 多义性
- 漏掉的组件

比如：图 5.19 代表一个核电厂控制系统。这个图中，制冷系统可能溢出，它代表一个危险的失效。如果在“开”位置上控制水的阀门被卡住或者是在满的模式下系统仍继续，那么这个情况就可能发生。而只有当测试满状态的传感器失灵和控制水流动的定时器不切断这两个基本事件都发生时，系统在满的模式下仍继续运行这件事才会发生。

从这个故障树，我们还可以构造另一棵树，就是割集树。这个树帮助我们找到单个的失效点，特别是当故障树很复杂难于分析的时候。规则如下：

1. 从上到下搜索，将故障树顶端第一个逻辑门分配给割集树顶端的节点。
2. 如果遇到一个或门，就将割集树劈成两半；如果遇到一个与门，就包括一个由子节点构成的复合节点。
3. 继续，直到所有的叶节点都是基本事件或者是由基本事件组成的复合节点。

割集是割集树的叶结点的集合。如图 5.20，左边是一个故障树，G1 是第一个逻辑门，它是或门，因此在割集树中被分成 G2,G3。依次地分解成右图的割集树。在这个图中，割集就是{A1,A3},{A1,A4},{A2,A3},{A2,A4},{A4,A5}.含义是，只有当 Ai,Aj 都发生的时候{Ai,Aj}才发生。这样我们就可以推断出所有可能的原因了。

到目前为止，应用于任何系统、硬件或是软件的概念我们已经描述过了，软件故障树分析的设计分解可以更精确，因为我们可以根据它的次序、决定和迭代进行任何形式的设计。这样每一种次序、决定和迭代结构都可以转换成都可以转换成相等的割集树表示和一个很大的割集。下一步，通过假设一些失效可能发生要详细检查这个设计，发现可能产生它的事件集合。如果没有发现。我们就假设这个失效不会发生。

一旦我们在设计中发现失效点，我们就要重新设计以减小系统的脆弱性。当在设计中发现故障的时候，我们有如下选择：

- 去掉它
- 增加一个组件或状况以防止导致这个故障的可能的输入
- 增加一个可以恢复失效所引起的损害的组件

虽然第一个选择是最佳的，但是它总是不可能的。

故障树通常用于计算给定的效率可能发生的可能性。但是也有一些缺点。首先，构造图要花费一定的时间。其次，很多系统都有很多依赖，所以很难发现不一致性，就像很难将注意力集中在设计的最关键的部分一样。此外，每一个失效必需的先决条件的数量和种类都不少，而且不总那么容易就能确定，同时也没有度量方法可以帮助我们将他们整理好。然而，研究人员们仍在继续寻找能够自动建立树和分析过程的方法。在美国和加拿大，故障树分析法被用于关键的航空和核应用，在这些领域里，失效的危险率值得花费相当的努力去建立和评估故障树。

5.7 设计评估和确认

当我们设计一个系统时，可以从两个不同的角度检查它。首先，我们要确保设计满足用户的需求规格说明。这个过程称为设计的确认。然后，我们要说的是设计质量：验证是否像我们一开始描述的那样将一个好的设计所具备的特征都具体实现了。自动化工具可以帮助我们进行验证和确认。然而，这些工具不能帮助整个工作。此外，我们可能有几种设计或是设计决定要选择，我们必须决定那一个是最理想的。在这一节中，我们就要看看可以帮助我们进行验证和确认的技术。

数学验证

理想情况是，我们希望在形式上验证设计是正确的，系统处理可以正确地将输入转化成所希望的输出。一些研究者们已经通过在系统中加入一些处理过程是这种类型的确认具有数学精度。和每一个处理相关联的是一个输入集，一个所希望的输出集和一个关于处理的断言集。然后，对于每一个这样的处理，我们都证明：

- 如果输入集被制定正确，那么它们会转化成所希望的输出集。
- 处理是没有错误地终止的。

这个过程“证明”设计是正确的。然而，使用这种方法去证明数学上每一个小的转换都是正确的，是要很费时的。所以，有的时候只在系统的最关键的部分使用它。形式设计符号可以使这个过程简单化，研究者们继续构造工具来支持他们。我们通常使用其它较少的公式方法来验证设计。

度量设计质量

一些研究人员正在开发用于评价设计质量的关键方面的度量方法。Chidamber 和 Demerer(1994)提出了一套用于面向对象设计的度量法，Bieman 和 Ott(1993)提出了用于面向对象系统的度量内聚法。Briand,Morasca 和 Basili(1994)提出了用于高层次设计的通用的度量法，包括内聚和耦合，Briand，Devanbu 和 melo(1997)在这些思想的基础上提出了度量耦合的方法。

让我们来看一看，这些度量方法是怎样揭示关于设计的信息的，以耦合度量法为例。Briand et al.指出在像 C++这样的设计中，耦合基于如下三个不同的特征：关系（友，继承，或都不是），迹（一个改变的流是指向一个类还是从一个类出来）和类型（类-属性交互，类-方法交互，方法-方法交互）。每个类的设计中都定义一个规格来计数类之间的或类和方法之间的交互。然后用以往的有关设计和导致系统故障和失效的经验来分析耦合类型和被发现的故障种类之间的关系。比如，当发现一个类与不是它父类、子类或友元的类发生频繁的属性交互的时候，则代码可能更容易发生故障。用这种方式，设计信息可用于预测故障可能会在那里被发现；然后我们可以在设计策略中采取步骤来防止故障，或进行故障容错。

Card 和 Glass（1990）讨论了软件设计复杂度的度量方法的导出方法。一开始是寻找一个预测故障的好的方法，他们提到了，由 Card,Church 和 Agresti（1986）发现的故障的分布是不依赖于设计中耦合的类型的，而且，有大跨度的控制的组件会花费较高和有较高的出现故障的可能性。基于对组件内聚的客观评估，他们还发现高内聚的模块有较低的故障发生率和开发费用。

Card 和 Glass（1990）指出设计复杂度实际涉及两个方面：组件内部的复杂度和组件间的复杂度。他们定义了包含这两个方面的系统复杂度的度量方法，还讨论了将用数据进行测试的时候如何提炼这个定义。他们最终的度量复杂度的公式是一个和 C=S+D，其中，S 是组件间的结构复杂度，D 是组件内部的数据复杂度，f(i)是组件 I 的扇出，V(i)是组件 I 的输

入输出变量个数，n 是组件个数。C 和每千行代码中的故障个数的关系入图 5.21 所示。统计数据显示这种度量方法捕获的设计复杂度占故障率变化的 69%，而且，一个单位的复杂度的增长会引起每千行代码中 0.4 个故障的出现的增长率。

需要记住的是，这些度量方法是从特定的开发程序中得出的，如果移植到你自己的项目中的话，不一定很确切。也就是说，你可以根据以前的项目的数据得出描述变量间关系的公式，但是你的公式和前面列出可能会有所不同。

比较设计

度量各种不同的设计特性不仅对预测故障有用，而且对比较两个设计也很有帮助。比如，我们可以用故障容错或是改正故障树分析中的一个问题来改善我们的设计；基于对比的度量能够保证新的设计的确比旧的好。此外，我们可能根据不同的体系风格的出不同的设计方案，这个时候我们必须决定那一个更适合系统的目标。

一个规格说明，多种设计

不同的设计风格可以被用来解决同一个问题。举例：KWIC（上下文中的关键字）检索系统接受一个有序的行集；每一行是有序的词集合，每一个词是有序的字符集合。任何行都可以重复地删除第一个词，然后在行后追加一个。这个系统输出一个按字母顺序的所有行的所有循环变动的列表。

Shaw 和 Garlan(1996)提出了四种不同的体系结构设计来实现 KWIC 系统：共享数据，抽象数据类型，隐式请求和管道过滤器。如图 5.22 是共享数据的解决方案，将问题分解成四个功能部分：输入，循环变动，按字母顺序排列和输出。由一个主程序按顺序调用它们。数据集中存放，被组件分开计算。然而，Parnas 指出这种设计方法很难修改，因为一个数据的改动会影响所有组件，因为设计不是复用的。

图 5.23 介绍了一个不同的设计方法。数据不再集中存储，但是处理分解是相同的。和上一个方案相比，数据和算法都很容易改动，也容易复用，因为组件间的依赖成分降低了。但是也很难加进新的功能。

图 5.24 介绍了另一个共享数据方案，但是这一次，数据的界面很是不同。第一个设计中，每一个组件对数据形式都可以直接的访问；而在这种方法中数据被抽象的访问。此外，当数据修改时，计算才发生。这种方法使得功能增强更容易了，因为可以通过改变数据的事件标记。同时，因为数据被抽象地处理，数据表示的改变将不会影响到设计的计算部分。这些特征都使设计较前两个易于复用，因为所有的行为都依赖于事件触发。

但是另一方面，我们不能简单地控制处理过程的顺序。基于这样的原因，我们看一下管道-过滤器方案。如图 5.25，处理的顺序由过滤器的顺序控制。因为每一个过滤器都是独立运行的，这个设计易复用，且每个过滤器易修改。然而，很难实现这个设计，使它成为交互系统。

我们看到每一种设计都有优点和缺点。这样，我们需要有一种方法来比较这些设计，以便我们选出最适合我们系统的设计方案来。

比较表

Shaw 和 Garlan（1996）通过建立一个含有重要性质的表格（如表 5.4），比较了四种定义方式。每一行代表一个性质，一列代表一种定义类型。表中的减号表示这一行的性质不是该列代表的定义方式拥有的；加号表示该定义有这个性质。从表中我们发现选择并不十分明

显。因此如果我们想选择一个最佳定义方式来满足我们的要求，我们必须给每个性质分配优先权，从而形成加权值。

表 5.4

	共享数据	抽象数据类型	隐式请求	管道过滤器
易于修改算法	-	-	+	+
易于修改数据示	-	+	-	-
易于修改功能	+	-	+	+
高性能	+	+	-	-
易于复用	-	+	-	+

其它一些我们需要选择的特征包括：

- 模块性
- 可测试性
- 可维护性
- 效率
- 是否易于理解
- 是否易于修改
- 一致性

一旦我们列出了所有需要评估的性质，我们就通过给每个性质分配一个与其优先权对应的加权值来显示它的重要性。例如：如果定义被要求在一些其它的产品中可被重用，我们就给“可重用性”这个性质分配“5”这个加权值。在这里我们的取值范围是 1 到 5，5 是最高优先权。

下一步，我们形成一个如表 5.5 的矩阵。矩阵的每一行的含义与表 5.4 相同。矩阵的第一列列出了我们在前一步中定义的加权值。第二列列出了我们对每种性质定义的优先权。在剩下的列中，我们根据已经列出的标准来评价每种定义方式。如果总共要比较 n 种定义方式，从第三列到第 n+2 列就代表了这些定义方式。我们给这每一列都分配一个 1 到 5 之间的评价值，因此表中的第 j 列的位置就可以评价第 j 列的定义方式是怎样满足第 I 行的性质的。

表 5.5

	优先级	共享数据	抽象数据类型	隐式请求	管道过滤器
易于修改算法	1	1	2	4	5
易于修改数据表示	4	1	5	2	1
易于修改功能	3	4	1	4	5
高性能	3	5	4	2	2
易于复用	5	1	4	2	5

最后，我们给每个定义方式计算一个分值。计算方法是：将每个形式的优先权值与该定义方式对它定义的加权值相乘，再将这些值相加得出结果。例如，pipe 和 filter 定义方式的分数的计算是这么进行的：1*5+4*1+3*5+3*2+5*5＝55。因此，根据以上定义的优先权与评价值，我们计算出了这四种定义方式的分值。他们是：

共享数据 37

抽象数据类型 57

隐式请求 40

管道过滤器 55

在这种情况下我们选择了抽象数据类型。需要注意的是，优先权、评价值和性质的选择都是十分主观的，是依赖于我们的客户、用户的需要和我们建立与维护系统的偏好的。不同的方法会得出不同的分值，从而使我们做出不同的选择。通过深入地学习如何测量定义的性质以及其它的系统特性之间的关系，我们会从这个方法中削弱一些主观因素。但是由于我们每个人的需求和观点都不相同，主观因素实际上是不可避免的。

设计复查

当设计结束时，我们要在开始下一步工作以前和我们的客户一起对设计进行复查。复查过程分为三步，与设计过程相对应。首先，我们要和客户、用户一起，采用概要设计复查的方法来检查我们在概念上的设计。然后，在关键设计审查中，我们要向其它开发者描述我们在技术上的设计来检查设计的细节。最后，我们进行程序设计的复查。因此，在实现以前，编程人员可以从他们的设计上得到反馈。每个复查过程的目标都是相同的，即确保我们的设计与实现正是用户想要的。

概要设计复查

在概要设计复查中，我们和客户、用户一起来验证概念上的设计，即我们要确保我们的设计包含了用户需求的所有方面。为了做到这一点，我们要邀请几位关键人员来参加复查：

- 参与定义系统需求的客户
- 参与定义系统需求的分析员
- 系统用户
- 系统设计者
- 会议主席
- 秘书
- 一些没有参与这项工程却对其感兴趣的系统开发人员

参与复查的人员数量不仅依赖于所开发系统的规模和复杂程度，还依赖于用户的数量和种类。在复查小组中的每个成员都有权力成为他或她公司的代表，而且有权做出决定和承诺。人员总数不应太多，以免妨碍我们进行讨论和做出决定。

会议主席领导讨论，但是他必须在这项工程中没有既定的利益。他要促使讨论进行下去，像一个在相对意见中进行调解的调解人，而且要维护过程的目的和平衡。

因为参与讨论、记录主要观点和结果都是十分困难的，所以我们需要一个秘书。他不参与问题的讨论，他或她的工作只是一个记录者。然而，除了必须的速记技巧以外，秘书还需要有足够的技术知识去了解讨论过程和记录相关的技术信息。实际上，秘书经常会向发言者提问，要求他们说清楚他们的观点，从而他们可以记录的更明白。

没有参与到这个工程的开发人员要提供一个局外人的观点。因为他们没有个人感情在这之中，所以在评论提出的设计时，他们可以保持客观公正。实际上，他们可能会有一些新的观点，从而促进设计的发展。他们也像一个小型质量保证小组，确保能采取合适的步骤来实现问题的正确性、一致性和好的设计实现。通过在复查中的实现，他们假定每个设计者对设计有相同的责任。这个共有的责任强迫复查中的每个人都要仔细检查每个设计细节。

在复查过程中，我们像我们的听众提出概念设计。在这个过程中，我们陈述了一哦那个中拥有的在需求文档中要求的结构、函数、特点。我们证实了提出的设计包括了必需的硬件、与其它系统的接口、输入与输出。客户来批准提出的菜单、对话框、报告格式，以及对错误的处理。如果系统是分阶段建设的，我们还要描述各阶段的特点和功能。

任何发现的分歧都要被秘书记录下来，再在大会上被讨论。当问题出现时，我们就要

解决它。然而，如果主要错误和误解出现了，我们就要修改设计。在这种情况下，我们要安排一个新的时间进行概要设计复查来评价新的设计。

关键设计复查

一旦客户对提出的设计感到满意，我们就要进行一个关键设计复查了。在这里，我们对技术设计提出一个概述。关键设计复查的参与者包括：

- 参与定义系统需求的分析员
- 系统设计者
- 会议主席
- 秘书
- 这个工程的程序设计者
- 系统测试员
- 写系统文档的分析员
- 一些没有参与这项工程却对其感兴趣的系统开发人员

注意，这个小组中的人员比在概要设计复查小组中的人员在技术性上要强一些。这是因为在关键设计复查中我们要陈述设计的技术细节。正像前面的一样，主席控制讨论的顺序来保证复审的焦点主要集中在两个问题上：设计是否实现了所有的需求？设计是否是高质量的？程序设计者们不仅要评论设计，而且要理解它。因此以后他们可以从中得到一些程序设计的思想。

通过使用图和数据，我们解释了各种设计策略以及我们怎样和为什么做的设计决定。如果我们使用了设计工具，复审小组就会容易的得到输出来检验正确性。正像前面的一样，如果我们发现了严重问题，设计就必须要重新来过；如果必要的话，关键设计复查或概要设计复查和关键设计复查就要重新进行。

程序设计复查

当我们对技术设计感到满意时，程序设计人员就要将这个技术设计解释为一系列的设计描述，从而建立用来编码和测试的实际组件。在程序设计完成之后，开始编码之前成学设计者要向一个由其它设计者、分析员、编程人员组成的小组提出他的设计方案，从而得到评论、获得建议。复审小组应包括：

- 提出系统需求的分析员
- 系统设计员
- 程序设计员
- 一个系统测试员
- 写系统文档的分析员
- 开发人员
- 会议主席
- 秘书
- 一些没有参与这项工程却对其感兴趣的系统开发人员

正像前面的一样，参与的人数决定于工程的规模和复杂程度，参与人员必须有权力做决定。主席要平衡并使讨论进行下去，秘书要记录技术讨论和其导致的决议。旁观者继续保持复审的客观性、详细检查设计、帮助控制质量。

设计复审的价值

在每种设计复审中，听众都要提出几个重要问题，包括：

1. 此设计能解决这个问题吗？
2. 此设计是模块化的、有良好结构的。易于理解的吗？

3. 可以采取一些措施去改善结构、增加设计的可理解性吗？
4. 此设计可移植到其它平台上吗？
5. 此设计是可重用的吗？
6. 此设计是易于修改和扩充的吗？
7. 此设计易于测试吗？
8. 此设计拥有最佳的性能吗？在何处适用？
9. 此设计重用了其它工程的一些组件吗？哪里是重用的？
10. 算法合适吗？还可以被改进吗？
11. 如果系统是分阶段建设的，那么各阶段之间的接口是否是足够的？这是为了确保各阶段之间的参数传递是容易的。
12. 文档是否完备？应包括设计的选择和理由。
13. 设计的组件和数据是否互见于需求？
14. 设计是否采用了合适的技术处理错误、阻止故障？

设计复查过程的一部分要集中于发现错误而不是改正他们。需要牢记的是参与人员是要调查设计的完整性而不是设计者的完整性。因此在复审时，所有人都要按着相同的目标去工作。在设计复审中的评语和讨论必须是无私的，这是因为评论是赋予过程和产品而不是参与者的。复审过程加强了小组之中不同人员间的交流。

此外，如果我们在问题和错误都十分简单、易于解决的时候发现他们，这将对我们是十分有利的。在抽象的和概念的阶段解决一个问题要比在实现后再去解决容易的多。在开发过程中许多困难和固定错误的花费是与发现错误的地点有关的。如果在设计复审中我们发现了一个问题，我们就知道了这个问题是在设计的什么部分出现的。然而，如果当系统已经运行后我们再发现一个问题，这个问题的根源就可能出现在很多方面：硬件、软件、设计、实现或是文档。因此越早发现一个问题，我们所需检查的地方就越少。

5.8 记录设计

设计过程的一个重要产品是一系列描述将要建立的系统的文档。正如我们已经看到的，文档的一部分要用自然语言告诉客户和用户系统是做什么的；另一部分就要用技术术语来描述系统的结构、数据、函数。因此这两部分的内容可能会重叠，但是描述的方式就不一定会重叠了。

设计文档应该包含一个叫做设计理由的部分，概述重要问题和在产生设计时考虑的一些平衡因素。这样就帮助了客户和其它开发人员理解如何和为什么设计的某些部分会组合在一起。

设计也包括了对系统组件的描述。一个部分将会描述用户是怎样与系统相互作用的，这包括：

- 菜单和其它显示格式
- 人机界面：功能键，触摸屏描述，键盘安排，鼠标和操纵杆的使用。
- 报告格式
- 输入：数据的来源、格式，以及存储媒介
- 输出：数据的目的、格式，以及存储媒介
- 普遍的函数特点
- 性能约束
- 档案的程序
- 处理错误的方法

通常一些符号和图表描述了系统的所有组织和结构，包括抽象的各个级别。

如果系统是分布式的，设计的结构就要详细描述用来展示网络的拓扑结构，网络节点间的相互访问，以及节点上的函数分配。如果系统需求包括时间上的限制，或是网络上的节点需要同步，那么在设计上就要描述定时是怎么工作的。相似地，设计还要包括控制信息和路由信息。他也可能包括网络完整性地规定：要确保数据的准确或是在失败后，数据可以得到恢复。

如果客户需要，设计还应包括监视系统性能的元素。此外，系统还应该包括一个手工控制系统的方式，在设计中要描述出这是怎么工作的。设计文档的其它部分应包括错误定位与分离，系统的重新配置，专门的安全措施。

最后，设计和需求是互见的。这个一致性就强迫大家要仔细检查。此外，这种互见性将使系统增强和修改在以后更加容易。例如，如果需求改变了，只要改变与其对应的设计就可以了。

5.9 信息系统实例

设计可以用各种方式记录。我们可以采用正式的语言、状态机、数据流图、数据字典、面向对象的方法，或是一些其它的符号和技术。选择适合你的设计和你正在研制的系统目标的技术和符号是十分重要的。例如，如果你正在设计一个将要被重新使用或是在其寿命之后将被大量修改的系统，你就必须选择一种对于广大设计者来说是易于理解的符号或是技术。

让我们看一下 Piccadilly 系统，看看一种设计技术的结合是十分有用的。图 5.26 描述了需求的一部分，即追踪当 Piccadilly 在电视上做广告的同时，另外一个电视台正在播放什么。这个事件我们采用数据流图来描述，而且很容易看出对方的时间表将在 Piccadilly 的节目计划中得到记录。（图 5.26 追踪对方时间表的最初模型）

Robertson 和 Robertson（1994）对这个系统提供了大量的信息，包括出自数据字典的数据元素的描述。对方的时间表在数据字典中，将以这种方式得到记录：

Opposition schedule= * Data flow *

Television company name

+{ Opposition transmission data

+ Opposition transmission time + Opposition program name

+ (Opposition predicted rating)}

花括号表示他所包含的元素可能多次发生，圆括号表示此元素是可选的。

在设计这个需求的系统时，我们可以将其它电视台都看成是“对方”。每个对方电视台都会产生“对方节目”的事例，我们可以用一个数据模型来表示他们之间的关系。根据需求说明，每个节目都有一个时间和数据；这个信息允许我们的系统定位那些用来展示相对的 Piccadilly 数据。我们叫每个 Piccadilly 节目的事例为一个事件。图 5.27 展示了这个事件的一个完整的数据模型。每个方框或是菱形框中的符号表示了想要的访问类型（创造、恢复、更新、删除）。线上的 N 连接表示在方框之间可能会有大量的数据拷贝进行传递，“1”表示只有一个数据拷贝传递。

（图 5.27 追踪对方时间表的数据模型）（图 5.28 追踪对方时间表的重要过程模型）

我们能通过一个高层次的事件的描绘更新原始的需求模型，正如在图 5.28 中所示。另外，我们能提供用自然语言描述我们想让软件作什么的算法：

Input：Opposition schedule

For each Television company name,crate Opposition company.

For each Opposition schedule

Locate the Episode where Episode schedule date = Opposition
transmission data AND Episode start time = Opposition
transmission time
Create instance of Opposition program
Create the relationships Planning and Coppeting
Output: List of Opposition programs

这一描述在一个高水平上是清晰的，并且更详细水平的摘要需要精确弄清程序要做什么。例如，设计需要关于节目开始时间的相似的但不是确切的信息（例如某一从下午 9 到 10：30 的 Piccadilly 节目可能与从 8 到 10 和从 10 到 11 的节目有些交迭）。我们必须也决定如何处理故障；在这个事例中，对方时间表可能包含一个无效的数据，诸如二月 31 日，或者一个无效的时间，例如 2900 小时。我们的最低水平细节应该包括组件描绘以便程序能被分配到独立的组件给代码和测试使用。尽管如此，这一例子向我们展示了我们能使用的互补设计技术或者额外的要求技术规格，来开始我们的设计程序。

5.10 实时事例

在这一章中，我们已经学习了异常处理和在设计时包含异常条件的必要性。调查 Ariane-5 故障的小组承认“不是对所有的 Ariane-4 换算都进行了保护，这是因为最大工作量的 80％都被用于 SRI 计算机（Lions et al.1996）。换句话说，由于性能需求，Ariane-5 的设计者们放弃了一些处理在 Ariane-4 中可能出现的例外情况的编码。这种类型的分析经常在设计过程中实现。我们通常使用折中的办法来满足一个非功能性的需求，这不仅时因为性能，而且是要在编码和测试时节省时间。

“为了决定易受伤的不被保护的代码，一个分析者执行关于每一个操作将引起一个运算对象故障。这导致七个变量中的四个将被进行保护。然而，剩下了三个不受保护的变量”（Lions et al 1996）。设计者感到在特定的情况下，一个溢出情况能发生，他们的错误分析用在了 Ariane－4 的假定，而不是 Ariane－5；实际上，Ariane－5 弹道参数是不同的，并且实际上溢出会引起火箭毁坏。

J 和 Meyer（1997）指出由合作者设计应该捕获 Ariane－5 在早期发展过程中的问题。他们还指出对于从 Ariane－4 再利用的组件没有精确的说明。一个系统任务说明的模糊部分陈述了代表水平和斜线的变量应该用 16 位，但是代码不能对那种情况检测。从下面可以看出这种情况已经弄清了。

```
Convert (horizontal_bias: DOUBLE): INTEGER is
  require
    horizontal_bias <=Maximum-bias
  do
   . . .
  ensure
   . . .
  end
```

当输入必须满的情况下，要注意以前的情况、要求、精确状态。

大体上，在测试阶段中合同里的断言应该已经被自动转化了，在 Ariane 的最初飞机之前显示了故障。或者断言能在执行阶段得到了保留，违反断言可能引起一个例外。所有中最好的是这些断言文档。是否由合作者或从设计的其他人而来的一部分将支持一个彻底的设计评论。在评论中，评论者将可能有细查断言，确保设计每一个完全的地址。在最后这个例子

中，问题可能已经在早期捕获，允许它被快速并廉价固定嵌入在执行中。

5.11 本章对你的意义

在这一章中，我们已经了解了对于设计一个系统它意味着什么。我们发现设计是在一个很高的水平上以一些关于达到想要的设计性质、系统的长期用途（如重用或是可修改）、基于系统需求的系统结构等的重要决定开始的。在你进行一个设计时，你要牢记以下几个方面：模块化、抽象级别、耦合、内聚、容错、原型法和用户界面的设计。同时，测量在评估组件质量，预计哪个组件在建立或维护过程中会耗费很高都是十分有用的。

5.12 本章对你们开发小组的意义

小组的许多活动都涉及到了设计。因为设计是依赖于组件的，所以组件和数据的相互关系必须被好好记录下来。设计过程的一部分就是和小组里的人员进行经常性的讨论，不仅是关于协调不同组件之间的相互作用，而且是更深入地了解需求和你在编码上做的每一个设计决议地含义。

你可以和用户一起工作来决定怎样设计系统界面。你可以设计几种版本来给用户观看，从而决定哪一种满足了性能需求，或是对你来说是最好看，感觉最好地。

你的设计方法地选择，即使是很私人化的，也要被记录下来。这样是为了让观看你设计的人们能够了解设计。在某些情况下，你的个人设计在设计全部完成后被翻译为一个普通的符号，这样是为了在以后修改设计是能够从整体上了解所有组件。在别的情况下，互见性被用来帮助了解设计的每个部分影响什么组件和数据。在任何情况下，在你记录你的设计时，将你对各个观点的讨论情况、你所作的选择完全、清楚地记录下来都是十分重要地。

作为小组中的一员，你也要参加设计复审，在几个阶段评估设计，对改进提出意见。记住，你只是对设计发表评论而不是对设计师，而且软件开发工作只有在技术讨论时完全放弃自负的情况下才能够做的最好。

5.13 本章对研究人员的意义

软件设计在开发过程中是一个很重要的部分，而且在其中还有很多有待探索的研究领域。一些研究人员致力于了解设计的本质：什么是关键活动？什么样的人是最佳设计者？是什么区分了一个好的和差的设计者？一些研究人员则集中研究符号和技术，希望能找到一种在保持可懂性和易修改性的同时能允许自动设计检查和互见性的新方法。

Shaw 和 Garlen（1996）主要研究了系统结构。他们引发的问题将会产生一个使设计更易使用，更易理解，更易修改的重要进展。

硬件设计比软件设计更先进。在这里有很多自动化技术和工具用来检验硬件的设计质量。研究人员接下来会将一些硬件技术应用与软件设计上；故障树分析、故障模式分析和其它方法可以被扩展，以使我们采用一种系统的方法而不是将硬件和软件分离。

最后，研究人员将观察一个好的设计者需要的素质。很明显的，设计需要很多的创造性。但是研究人员发现经验也很重要；因此我们需要决定那种经验是最佳的，从而将设计者训练到最佳状态。

5.14 学期工程

设计和管理一样需要艺术性和创造性。不同的设计者会采取不同的方法进行他们的设计，但是他们的结果都是固定的。我们认为设计是在一个从“以任务为中心的设计”到“以用户为中心设计”的连续集上进行的。在做“以任务为中心的设计”时首先要进行的是设计者要考虑一个系统必须完成什么功能。相反的，“以用户为中心设计”首先要考虑的是用户是怎么与系统相互作用来实现他们的任务的。这两种设计不是相互排斥的，通常他们是一种相互补充的关系。然而，一种设计思想通常会统治支配另一种思想。作为你学期工程的一部分，你要考虑一下：两种设计方法，一种是以任务为中心的设计，另一种是以用户为中心设计。你会选择什么样的设计风格？比较比较他们的结果。那种设计更容易转换？测试一下？

5.15 重要的参考书

有许多关于软件设计的好书。第一本就是 Shaw 和 Garlen（1996），这本书为你学习如何设计提供了一个结构框架。其它有用的书包括：Ward 和 Mellor(1986)，Hatley 和 Pirbhai（1987），Shumate 和 Keller（1992），McConnell（1993）和 Gomaa（1995），他们中有很多讲述了一些协作系统和实时系统的专门需要。你也可以参考以下书籍：界面设计（Hix 和 Hartson 1993；Shneiderman1997）和数据库（Weiderhold1998）。

一些期刊也提供了用户界面设计的专刊。Communications of the ACM 的 1993 年 4 月号讨论了图形用户界面。1996 年的 4 月号讨论了以初学者为中心的设计。IEEE Software 的 1990 年和 1997 年的 7 月号也讨论了用户界面的设计。

John McDermid 在 YORK 大学领导了一个将冒险、故障树、故障分析应用于在软件设计中出现的问题的研究小组。

在 Risks Forum 讨论了许多设计中出现的问题，这些内容我们可以在线获得或是在 ACM Software Engineering Notes 的每一期上找到相应的摘要。论坛的参与者们描述了一些与软件有关的冒险与故障的情况，并分析了原因。

5.16 习题

1. 在图 5.4 的 NIST/ECMA 模型中提出了什么类型的结构风格？
2. 在 Ariane-5 的设计方案中，对于 7 个可能事件中的 3 种情况，开发者有意的避开了异常处理。那么对于此有什么合情合理的解释呢？谁对由此而引起的后果负责？难到测试者没有发现这个设计的瑕癖吗？
3. 回顾 Shaw 和 Garlan 提出的结构风格。对于任意一种，它的高层成分有可能有或高或低的内聚和耦合吗？
4. 对于任一种内聚，写一个相应的描述。
5. 对于任一种耦合，举一个两个成分之间以相应方式耦合的例子。
6. 对一个已开发的工程，用多层交互成分为你的软件画一个系统示意图。系统的模型性好吗？有耦合吗？能重新构建你的系统以减少耦合增加内聚吗？
7. 一个系统能完全做到内部无耦合吗？也就是说，能否将该系统的耦合减到最低以至于它的各部分间没有耦合？
8. 试问，有没有这样的系统，各部分间不能做到功能性内聚？请说出你的理由。

9. 对于第一节中每一个质量属性，试解释好的设计怎样有助于提高产品的质量。例如，内聚、耦合、模块化是怎样引起不可信和跟踪能力等问题的。
10. 我们知道布尔变量的总和和它的有效成分是 1。用该特性验证表 5.2 的公式和前面提到的简化是等价的。
11. 在可视化的终端上设计一个全屏编辑器。该编辑器允许文本的插入、删除和修改。可以对文件进行剪切和粘贴操作。用户可以指定一个文本串，编辑器可以找到这个串的下一个出现。通过这个编辑器，用户可以实现对边框、页长度、标签的设置。然后，评估你的设计的质量。
12. 设计一个简单的翻译机。你的系统会接收一个字符串，然后判断他是不是一个合法的命令。写一条错误的信息。看看你的设计是怎么处理故障的。讨论你对不同故障的发现、防止，容错策略的设计的优缺点。
13. 我们叫一个在某种方式下调用自己的组件为一个递规组件。如果在这章中我们提到了它，你认为它是一个好的还是一个坏的思想？为什么？
14. 给出一个开发原型机时不会节省大量时间的例子。
15. 列出一个拥有原型是不合适的系统的特点。
16. 解释一些为什么模块化和应用程序产生器是不可分的概念。给出一个你曾经工作过的应用程序产生器的实例。
17. 解释一下为什么 Shaw 和 Garlen（1996）在图 5.22 中设计的解决方式是不容易重用的。
18. 列出你设计一个评价矩阵时可能会考虑的特性。确定你对以下系统你可能使用的加权值：一个操作系统、一个字处理系统、一个卫星定位系统。
19. 许多班级的工程要求你自己开发程序。集合一小组同学来为这样的工程设计写一个评论。由若干名同学作为消费者和拥护。开始设计评论时，一定要用非技术的方法把所有的需求和系统特性都表示出来。然后写出一个严格的设计评论。列出设计过程中所有的变化。比较在设计状态下改变所需的时间和它在已经存在的程序中改变所需的时间。
20. 你已经被一个计算机查询公司雇佣来开发一个为某账目公司计算收入税的软件包。我们已经根据用户的需求设计了系统并用设计评论的方式来表示它。下面的问题中哪一个应该在初始的设计评论中提及？在严格的设计评论中呢？在两者中的？解释你的答案。
    (a) 它将运行在什么计算机上？
    (b) 输入的屏幕是怎样的？
    (c) 将产生什么样的报告？
    (d) 会同时有多少个用户？
    (e) 你将使用一个多用户的操作系统吗?
    (f) 这个算法的细节是什么？

第 6 章 关于对象

这一章的主要内容：

1. 面向对象发展的特殊属性
2. 应用举例
3. 使用 UML
4. 面向对象系统的设计
5. 面向对象程序的设计
6. 面向对象的度量

既然我们已经知道怎样获取需求和设计系统，我们可以做一个迂回来更仔细地检验有关面向对象开发的概念。为什么面向对象的开发值得特殊的关注呢？因为现在正在被开发的新系统中的许多都包含一些或全部有关面向对象的概念。在这一章中，我们系统的描述了面向对象的开发怎样有别于其它类型的开发。我们介绍了面向对象过程中的概念，并且展示了他们是如何被应用于需求分析，系统设计，程序设计的。为了说明这些概念，我们应用 UML（统一模建语言）来为皇家服务站系统产生一个设计，需求在 Sidebar6.1 中列出。

Sidebar6.1 皇家服务站需求：

1． 皇家服务站为顾客提供三种类型的服务：加油，车辆管理和停车。也就是说，一个顾客

可以向他自己车辆的槽中加油，可以进行车辆维护，或者能够在停车场中停放车辆。一个顾客可以选择在购买时自动付账或是按月账单付账。无论是那种情况，顾客都可以用现金，信用卡或个人支票支付。皇家服务站燃料出售时，根据每加仑价格，而这个价格又依赖于燃料是否是柴油的，标准的或是高级的。服务的价格是根据部件和劳动力的费用。停车的价格根据天，周和月的比率。燃料，服务维护，部件和停车的价格都可能变动；只有站长 Manny 可以改动价格。根据他的决定权，可以指定给某一个顾客以打折的待遇；这个折扣也可能根据不同的顾客有所变动。而且所有的购买活动要缴纳 5%的税。

2. 系统必须跟踪每月的账单情况和由油站提供的产品和服务情况。跟踪的结果要报告给站管理员。
3. 站管理员使用系统控制清单。系统会在库存少的时候提出警告，并自动定购新的部件核燃料。
4. 系统将跟踪信用卡的历史，并且对其过期的用户发出警告信。账单在顾客购买后的每个月的第一天发给顾客。任何的账单只要在期限后 90 天没有付，就会取消顾客的信用卡。
5. 系统仅用于常规的重复顾客。常规的重复顾客意味着一个顾客可以通过姓名，地址和生日识别，并且他至少每个月在服务站消费一次。
6. 系统必须处理和其他系统的接口的日期要求。一个信用卡系统用于处理产品和服务的信用卡交易。信用卡系统要使用如下信息：卡号，姓名，到期日期和购买量。在接受这些信息后，信用卡系统证实交易使被允许还是拒绝。部件订购系统接受部件码和数量。他返回部件发送的日期。燃料订购系统要求一个燃料定购的描述，包括燃料类型，加仑量，站名和站的识别码。它返回燃料将被发送的日期。
7. 系统必须纪录税和相关信息，包括每一个顾客付的税，和每项的税。
8. 站长必须能够按照要求复查税纪录。
9. 系统会为用户提供定期信息，提醒他们车辆到时候该维修了。通常，维护需要每六个月一次。
10. 顾客可以在站停车场中租车位。每位用户必须从系统中要求一个可获得的车位。站长能够查看有关车位的占用情况的月纪录。
11. 系统维护一个统计信息库，可访问统计数和顾客姓名。
12. 站长必须能够按照要求复查统计信息。
13. 系统能够按要求报告价格的打折分析情况给站长。
14. 系统可以自动通知休眠账目的所有者。也就是，如果顾客超过两个月没有在站内进行消费的话，会被通知。
15. 系统不可用时间不能超过 24 小时。
16. 系统必须对顾客的信息进行保密。

本章中的许多例子是由Guilherme Travassos教授最初开发的，这些例子的更多细节和其它例子可在他的网站中找到：http://www.cos.ufrj.br/~ght.

6.1 什么是面向对象

面向对象是一种把问题和解决办法作为一个独立对象的集合组织在一起的软件开发方法；数据结构和操作都包含在实型中。我们可以通过七个特性来识别一个面向对象类型：等同，抽象，分类，封装，继承，多形性和持久性。一些实型只应用这七条中的一部分；虽然它们被叫做面向对象的，但是几乎所有的面向对象的实体都被叫做是基于对象的。

等同是指数据被组织成离散的，区别得出的，被称为对象的实体。一个单独的实体具有和它关联的状态和动作。例如，在一个控制河坝的系统中，坝本身可以是一个实体。河坝

会具有若干状态，包括全部打开和全部关闭。与河坝相关联的动作可以包括声音警告 ：当河坝将要从一个关闭的状态转换到开放的状态时，它将发出一个警告给顺流的、把河流作为娱乐的使用者，警告他们河水的视为将要升高。在面向对象的系统中每一个对象通常都有一个名字，也叫做一个参考或名称；名字用来区别不同的对象。

我们已经在第 1，4 和 5 章中看到，抽象对于建立任何一个系统都是很重要的，无论它是不是面向对象的系统。面向对象的系统中的抽象用来表示被开发的系统中不同的具体的观点。总的来说，抽象形成了一个表示不同的系统全貌是怎样关联起来的分级结构。

面向对象应用分类来把具有属性和操作的对象进行分组。例如，图 6.1 展示了一个不同对象的分类。有两个大象的对象被分在一组，四个飞机对象分在一组。一个单个的自行车在一个单独的类中，和一个茶杯一样。

大象的组可以被考虑为一个类。我们可以将大象类和属性联系起来，比如，颜色，大小和位置。此外，我们将于这个类相关的操作和它联系起来。如图 6.2，是用一个盒子来代表一个类。

注意，图 6.1 的分组反映了个体或组成队列的成员的全貌。例如，图中包括自行车和飞机的单独的类，然而，一个等价的，明智的分类方法可以是把自行车和飞机房子一起形成一个交通工具的类。记住类的定义和分层是用来代表某一个问题和它的解决方法：两种不同的表示方式可以同样正确和有用。

可以说，任何一个对象都是一个类的实例。每一个实力都有它自己的属性，但是和这个类中的其它实例共同拥有相同的属性名和操作。我们可以讨论一下大象这个类并且注意到诸如象牙和耳朵，这些属性的名称是相同的。我们也可以用属性来区别类中的不同成员。这样，一个类描述了一系列拥有相同的结构和相同的操作的对象，但其中的属性可以帮助我们区分不同的对象。

面向对象的系统用到了封装的概念。一个类封装了对象的属性和操作，把具体的细节隐藏了起来。然而，封装并不等于信息隐藏，正如 Berard 所说。封装的保护边界实际上是透明或半透明的。也就是说，有时你可以在保护的边界内看到，但又是它就被隐藏了起来。Berard 注意到“抽象是帮助我们识别那些信息应该可见，那些信息应该隐藏的一种技术。于是封装就是一个用把该隐藏的信息隐藏，该可见的信息可见的方法将信息包装起来的技术。

很明显，一些属性在一个给定的类中是被其成员共享的。例如，非洲象和亚洲象都有相同数目的象牙，但它们在其它特性上有差异，比如耳朵的形状。我们可以根据不同类之间的相同点和不同点把类按分级地组织起来：这个层次展示了面向对象类的继承结构。为了建立这个层次，我们从广泛的定义一个类开始，然后把它提炼成更多的专项子类。一个子类可以继承父类的数据结构，操作和属性。举个例子，图 6.3 展示了两种燃料是怎样被合并成一个燃料类的分级结构。有时我们用一个抽象类来简化这个层次结构，抽象类中不定义对象，除非作为子类的一个实例。

`一个行为是一个对象操作或被操作的动作或变换。一个对象的行为有一个特定信息的接收来触发，或者进入一个特定的状态。有时，同样的行为可以被不同的展示在不同的类或子类，这个性质成为多形性。例如，考虑一个多边形的类。每一个多边形有一个面积，但计算一个三角形的面积有别于计算一个矩形的面积。这样，面积的计算就是该类对象的特性。

某个类中一个操作的实现被称为一个方法。在一个具有多形性的系统中，一个操作可以有一个以上的方法来实现它。例如，可以有一个方法来计算三角形的面积，有另一个方法来计算正方形的面积。一个面相对性的设计语言被设计成自动选择正确的方法来实现一个操作，与操作有关的作为参数的数据，和对象所属类的名称。在我们的面积的例子中，对象将选择正确的面积方法，所根据的描述多边形的参数。多形性允许在不改变现有的代码的情况下增加新类。所以，例如，一个新的燃料比如天然期或碳可以被很容易的加进图 6.3 的层次

中。
与面向对象系统相联系的第七个属性是持久性：一个对象的名字，状态和行为的超越时间和空间的能力。换句话说，对象的名字，状态和行为在对象被改变时保存。例如，我们可能想让系统能保存每加仑柴油机燃料的价格。价格每天都可能改变，所以一个人今天付了 1.5 美元，明天可能付 1.55 美元。但是我们想要一个稳定的日价来比较去年和今年的价格，或者画图来表示价格随时间推移的变化。在这个例子中，对象是恒定的，即使它的属性改变了。

6.2 面向对象开发的过程

面向对象开发的一个优点是语言的一致性。我们可以用同样的方式来描述问题和解决方法：类，对象，方法，属性和行为。在整个开发过程中，我们应该具有术语和全局的一致性。例如，无论在过程的哪一部分表示一个对象，都应该包括对象的名字，属性和行为。在下一个更高的层次，用面向对象表示法来描述类有三个概貌：静态，动态，限制。静态的视图包括对象的描述，属性，行为和相互之间的关系。动态的视图描述了通信，控制/定时，和状态及状态的变化。限制描述了结构上的限制和动态行为。

这个跨过程的一致性是较多的传统的开发过程和面向对象的开发过程的关键不同点。面向对象开发过程用数据的封装和行为来形成独立的单位（对象）。相同的语义结构从需求到应用实现和测试表示了系统。这样，面向对象是有关问题和其解决方法的一门学问，并不是一个软件自身的生命周期。实际上，面向对象可用于许多不同的软件生命周期，从瀑布式到螺旋式。而且，像我们将在 12 章看到的那样，因为面向对象有封装和数据隐藏的优点，许多面向对象开发者喜欢用他们喜欢的方式来考虑对象和类的重用。一般来说，面向对象处理需求分析，高层次的设计，低层次的设计，编码，和测试，但是没有必要是一个顺序的方式；其顺序由生命周期来决定，不是由面向对象表示法。

表 6.1 列出了一个软件产品或工程的各种特性。右边的一列说明一些特性不可能改变，而其他的再开发过程中有很大的几率会发生本质的改变。这个表中的条款会帮助你决定那一种开发过程最适合你所遇到的情况。

表 6.1

软件产品或项目的特征	改变的可能性
从应用中得到的对象	低
长期的信息结构	低
被动的对象属性	中
行为的顺序	中
与外部世界的接口	高
功能	高

面向对象的需求分析

不管生命周期是怎样的， 一个面向对象开发周期都需要以下几个步骤：描述需求，设计系统，设计程序，编码和测试。面向对象的需求分析通常使用用户的语言来描述并且讨论了在应用领域中的概念和情况。这些概念包括了信息，服务和责任。主页领域只是是开发者能够理解系统将被用于的上下文，并且可以以一种用户理解的方式描述需求。需求的表达是同样的情况，无论开发者打算怎样实现系统。也就是说，需求的定义可以作为一个对象来独立表示。

面向对象设计

正常来说，就像我们在第 4 章看到的那样，需求说明书应该用一种面向对象的方法来表示，以便设计者可以从问题的说明中产生出它们的设计。然而，在许多例子中，面向对象的问题描述都相同于或类似于发现面向对象的解决办法的第一个步骤。所以在面向对象的开发过程中，需求说明书的步骤世纪上可以作为系统设计的第一步。也就是说，在两种情况中，对象都必须被定义，它们的关系也要被描述。基于这个原因，面向对象开发经常把写一个有别于需求文档的需求说明书省略掉。

用面向对象的方法表示一个系统设计有两个方针。首先，定义和表示类和对象是很重要的。我们必须不但了解问题领域中的对象，而且要知道每个对象的属性和行为的细节。第二，我们必须定义对象和类之间的交互作用和关系：它们的联系，组成，聚合和继承关系。系统设计被当作一个最终将成为程序设计的高层次的抽象。一系统设计开始，程序设计者采取若干步为实现提供重要的细节。

1. 他们在模型中插入计算的特征。
2. 他们插入了一些类库的细节，通常使用自底向上的方法。
3. 他们考虑了非功能的需求，比如操作和安全性，并适当的增强设计。

面向对象的编码和测试

一旦程序设计完成以后，系统用对象的模型，属性和行为在一个非常低的层次上被描述。编码通过把模型翻译成一种面向对象变成语言来进行。这个步骤不能敷衍。通常提炼分级结构并且随着需求的增长和成熟做调整对实现者来说是必要的。在一些情况下，实现者还要寻找机会来使系统的某部分更普遍，以便对象和类更适合在将来重用。

测试一个面向对象系统包括一些在测试任一类系统都要进行的动作。编码者对他们自己的工作进行单元测试，然后再联合起来在测试组进行完整性，系统性，适应性的测试。图 6.4 说明了抽象层是怎样和各种不同的测试类型联系起来的。这些测试类型的每一个都将在第 8 章和第 9 章更详细的阐述。然而，有一些面向对象的特性要求在测试过程中特别注意；我们将在第 8 章讨论它们。

6.3 用例

为了用面向对象的方法开发一个系统，引出需求并且用用例的技术描述它们。一个用例描述了一个系统通过建立对话操作或展示一个用户，外部系统或其他实体在所开发的系统中的函数性。例如，一个用例可以描述银行的储户与 ATM 交互来从一个账号中取钱的方法。或者它可以详细说明 ATM 是如何与记录用户将打算取多少钱的数据库交互的。与系统交互的实体叫做参与者，它可以是一个用户，一个容器或其它系统。

每一个用例描述了实体可能与系统交互的一种可能的情况。用例经常被表示城有关对象的绘图，加上一个简短的关于函数如何使用的说明。对于每一个情况，用例都定义了所有可能发生的事件和系统的反应。用例在整体上构造了一个所有可能的实体在用所有可能的方法使用系统的一个完整的描述。这样，用例的集合描述了系统的一个完整的功能图。

用例对于消费者，系统设计者和测试者之间的通信是很有用的。消费者通过用例可以看到系统的设计是否包括想要功能。系统设计者可以用用例设计被操作和数据将被存储在何时位置的对象。测试者可以把用作为系统测试的基础。当系统完成时，消费者也可以以用例为基础建立验收测试计划，以便功能中的每一个单位都能包括在至少一个测试中。这样，用

例存在于生命周期中的每一个阶段，作为通信的一个终结。基于这个原因，用例在许多面向对象通信中被采用作为在系统分析中建立对象的更正式的方法的一个补充。

用例图表有四个元素：参与者，事例，扩展名和用途。就像上面提到的，参与者是一个被有关系统的实体操作的角色。事例是系统功能的某些方面的描画，它对于被用例反映全貌的参与者是可见的。扩展名扩展用历来说明一个不同的或更深入的全貌。一个用途通常是一个已经定义的用例的重用。

例如，考虑一个实现皇家服务站实现的系统。一个高层次的视图可能看起来像图 6.5 的图表。这里，服务站有三种类型的服务：补充燃料、停车、和维修。消费者在这里是个参与者，而且每种服务都回乡消费者提供一个单子。

用例帮助我们理解消费者和他们的问题。对每一个用例图表，我们都写一个情况处方来描述系统将提供的功能。当我们文问题的时候，我们会发现在开始的用例描画中没有发现活不清楚的问题。例如，我们可以扩展图 6.5 中用例图表的一部分，修改它并把图 6.6 中的元素加进去。在新的描画中，我们有一个附加的参与者，这个站的管理者，和一个新的服务：预维修。这里，预维修扩展了维修的概念而成为另一种服务。

其次，我们考虑到许多消费者肯能用信誉卡付账。信誉卡系统必须与我们的系统相联系，所以我们在第二层扩展加入它的作用，在图 6.7 中描画。

最后，我们扩展系统把所有的参与者都包括进去。为了做到这一点，我们在与完成一个任务或提供一个服务的系统相联系的环境中寻找所有的实体。实体可以是类型的参与者：使用者，替她系统，外部组织，或外部容器。我们看到参与者做了什么和任务是在系统的那个过程中完成的。例如，一些参与者初始化事件，其他的作为事件的结果与系统相作用。在每一个事例中，我们把参与者看成一个角色，而不是特殊的人。John 和 Jane 可以是两个人，但是对系统来说每个的作用都是一样的。

我们可以用一系列的问题来帮助我们定义参与者：

1. 什么用户和组用该系统来完成一个任务？
2. 为了系统能发挥它的作用，什么样的用户和组被需要？
3. 什么外部系统用该系统的什么来完成任务？
4. 什么外部系统，用户或组发信息给系统？
5. 什么外部系统，用户或组从系统接收信息？

用这些问题，我们决定在皇家服务站系统包括一个打印账单的打印系统，一个定燃料的系统以便在供应量低时订购更多的燃料，和一个零件订购系统订购零件来支持维修和修补。我们也可以在整个系统中增加两个子系统：一个用来处理税务，另一个控制站上现有的存货来处理燃料和零件。结果可以看图 6.8.土上不是全部的用例。就像我们将在下一节中看到的那样，对图表中的每一条来说，我们都有一个描述：这个条目是做什么的，有哪些数据和行为。结果关于系统怎样工作的理解性的描述，包括它与超出它的范围的系统的联系。

从不同的角度分析系统的一步一步的过程产生了超过一巨大系列的用例。通过像我们表述系统功能的不同表述，用例给我们提供了怎样考虑系统需求的选择方法。除了扩展和阐明需求以外，用例在发现需求错误方面也是有用的。如果用自然语言写的需求完成得很好，那么把需求转换成用例的任务也相对直观些。然而，有时用自然语言描述的问题需求被隐藏了；当我们读需求定义的时候不能直观的看到。当我们翻译成用例时，这些问题就出现了。这样。用例翻译本身就是对需求分析质量的一个很好的检验。

一旦我们有了用例，我们可以更深入的检验它们来发现已经存在的和潜在的问题。例如，我们可以问一下类型的问题：

• 例是否用了正确的术语？也就是，是否有两个或更多的条目来描述需求中的相同的

实体或参与者？

- 动作是否与角度相匹配？也就是说，一个用例是否用和参与者本身的描述相一致的方法描述系统和参与者之间的联系？
- 动作和参与者的描述是否清晰，完整？有没有多于一种的方法来解释描述？
- 是否有被描述的联系丢失了它的附加参与者描述？
- 是否有的一个外部系统或一系列用户被描述但是机上没有包括在某个动作中或与系统相联系？
- 每个动作的开始和结束是否清楚？

这些问题重点强调了我们在第 1 章中讨论的系统边界的重要性。检验用户用例给我们一个更好的理解系统边界以及什么于系统有联系，什么与系统完全没有关系的机会。

同时，我们可以给功能本身一个附加的安全性。用一个适合把描述应用于现实中的方法描述系统行为不总是那么容易的。考虑写一个需求来教某人其自行车。首先，看起来好象我们需要描述的全部就是腿蹬踏板的的动作，和手握方向盘的动作，用车闸，或许还有更换齿轮。直到后来我们才意识到还有微妙的全身动作帮助我们平衡和转弯难以描述。用同样的方法，我们必须把所有那些细小但重要队是系统功能正确的事情包括在用例中。我们会发现一些失误可以严格的描述，但另一些则不能。随着我们理解的增长，或许通过建立照相制版活想象来设计想象，我们可返回用例使它们更严格。

一种增加理解的方法是证明用例中的跟踪能力。也就是说，我们注意那些用例独立于其它？每个实体都是从哪里来的？是在哪创建的？什么时候声明终止？我们寻找每一个用例中有哪些参与者，并且要弄清楚多有的用例都包括了它所需要的所有参与者。类似的，我们检验用例之间的独立性：在一个参与者开始之前哪个参与者必须结束？有没有冲突和不相容？一个跟踪能力的巨针对变化影响的古选也是非常有用的，消费者，设计者或测试者应该建议需求或设计是可被修改的。就像我们将在第 9 和 11 章看到的那样，跟踪能力和组态控制不仅对建立系统，而且对维护它也是很重要的。

6.4 表示面向对象：一个用统一建模语言的例子

统一建模语言是描述面向对象解决问题的办法得很重要的方法。它可以倍加工成适合不同的开发环境和软件生命周期。实际上，像对象管理组这样的组织已经使用统一建模语言作为面向对象的重要的标准。在这一节中，我们解释了统一建模语言的结构并用统一建模语言为皇家服务站问题做了一个高层次的设计。对统一建模语言的全面的介绍，请看larman(1998)。

统一建模语言可以被用于显示、详细说明、证明一个问题。它对于描述不同的设计选择，和最终证明人工制品是特别有用的。UML 图表包括系统动态的视图，静态的视图，限制，和形式化。动态的视图用用例、动作清单、显示顺序和写作的相互作用图表和说明状态及其变化的状态机器来描述。静态视图用类图表来描述，展示了它们的联系和可扩充性。另外，静态视图展示了打包和展开。限制和形式化用 OCL（对象限制语言）来表示。在这一节的剩余部分中，我们用同一建模语言和它的一些结构探讨了皇家服务站问题。

过程中的统一建模语言

因为面向对象应用在开发的所有部分，统一建模语言可以应用于软件开发的全过程。图 6.9 展示了统一建模语言是怎样应用在规格说明书，设计和编码中的。在需求过程中，工作流程图表通过描述工作所需要完成的动作定义了整个工作过程。就像我们看到的那样，用

例图表也能描述通过描述整个系统需要完成的过程所建立的系统。另外用例能够包含不同的描述系统怎样工作和用户怎样与系统联系的情况。这些图表可以用使用它们的函数来定义对象的类的对象模型作补充。类和它们之间的关系开始是非正式定义的，随着用户产生的各种情况和设计的进展逐渐正式起来。

一旦需求有了一个比较好的情况，设计就以同一建模语言和行为图表开始。行为图表展示了在系统中随着对象属性的改变所能发生的所有情况。例如，一个对象可能会对类型 A 的输入产生一个短报告，而为类型 B 的输入产生一个常报告；报告对象根据输入产生正确的输出。与动作图表一致，我们开发状态图表来显示对象可能具有的所有状态。从一个状态到另一个状态的改变又在两个对象之间传送的信息来触发。这样，状态图表当一个类可能有许多状态改变才需要。

其次，设计通过访问对象的静态结构来继续。一系列结构和对象图表描述了一个类是怎样和其它类联系的，包括继承关系。类之间的相互作用用两种作用图表来说明：顺序和协作。顺序图表，像第 4 章的围墙图表，展示了消息是怎样从一个对象到另一个对象之间流动的，把需求中事件的不正规描述规范化。协作图表使用对象和顺序信息来展示对象之间事件的流动。

最后，设计可以用封装、组件、和展开图表来实现。封装图表显示出类是怎样逻辑的分成若干模块的。组件图表经常和封装相同，但它们反映了实际上的最终的系统模块。因为最终的模块经常是一个巨大的网络工作的一部分，所以展开图表显示了与正在建立的应用程序有关的网络连接。

6.5 面向对象系统设计

来看一下 UML 是怎样应用的，考虑皇家服务站系统的顶层设计情况。从 Sidebar6.1 的需求我们要得出一个系统设计。这就是说我们根据有关这个系统的描述要设计出一个解决办法。一开始，我们用 UML 类图。这些类图描述对象类型和他们的关系。特别地，当我们要描述对象之间的联系，还有类-子类之间的关系。我们用图表示每一个对象的属性，他们的行为和施加在每一个类或对象上的限制。

设计过程从需求的陈述开始。我们提取出名词，寻找特殊的条目：

- 结构
- 外部系统
- 装置
- 角色
- 操作过程
- 位置
- 组织
- 要建立的由系统操纵的事情

比如考虑系统的第一个需求部分：

一个顾客有权利选择购买时自动付账或是被寄一张月账单。无论那种情况，顾客都可以付现金，信用卡或是个人支票。皇家服务站的燃料是根据每加仑的价格出售的，而每加仑的价格又取决于燃料是否是柴油机的，标准的或是高级的。服务的价格是根据地区和劳动的花费。停车出是根据天，周和月的利率出售的。燃料、维护服务，地区和停车的价格都可能变动；只有 Manny，站的管理员，能进入或改变价格。他的权利有，可以指定某个顾客购买时打折；这个折扣根据不同的顾客也可以有所变动。所有的购买都必须上缴 5%的地方税。

从这个需求的阐述中，我们可以提取出几个暂时的类，包括：

- 个人支票
- 账单
- 信用卡
- 顾客
- 站管理员
- 购买
- 燃料
- 服务
- 折扣
- 税
- 停车
- 维修
- 现金
- 价格

你可以以如下问题为指导，判断那些要包括在你的候选类里：

1. 那些需要以某种方式处理？
2. 那些条目有多个属性？

3. 你什么时候有不止一个对象在类里？
4. 那些是基于需求本身的，不是从你对需求的理解中得到的？
5. 那些属性和操作对一个类或对象总是适用的？

这些问题可以帮助我们组织候选的类，如表 6.2 所示。

下一步，我们要检查其他需求，看看有什么要加进来。比如：第五个需求说，系统只用于标准重复顾客。一个标准的重复顾客意味着用名字、地址和生日可以完全识别的，并且每个月至少享受一次站的服务。类似地，第九个需求说系统会定期为顾客提供信息，提醒他们车辆到时候该维护了。通常，维护每六个月一次。

这样，我们又有了新的候选类，比如标准重复顾客，姓名，地址和定期信息。我们修订表 6.2 为表 6.3。然而，标准重复顾客是冗余的，因此去掉它。

对于整个的需求，我们将包括所有的类，如表 6.4

表 6.4 第一组属性和类：第三步

属性：	类：
个人支票	顾客
税	维护
价格	服务
现金	停车
信用卡	燃料
打折	账单
生日	购买
姓名	定期信息
地址	站管理员
	警告信
	部件
	账目
	清单
	信用卡系统
	地区-订购系统
	燃料-订购系统

接下来，我们要识别出在我们的设计中必须要描述的行为。从需求陈述中，提取动词。寻找能使人想到行为的特定条目：

- 命令动词
- 被动词
- 动作
- 事情或提醒事件
- 作用
- 操作过程
- 由一个组织提供的服务

这些行为会变成由一个类或对象执行的动作或责任，或是对类或对象施加的动作。比如：登记一个顾客是一个行为。这个行为是由整个系统的一部分执行的，而且登记会影响到顾客。

为了更容易地管理对象，类和行为，我们使用 UML 图来描述它们的关系。图 6.10 是一个 UML 盒，用于描述一个类的组件的。三层中的顶层包括类的名字，中间层是属性，底层是操作的描述。每一个属性都由它的名字，类型和初始值描述。类似地，每一个操作都有它的名字，参数列表和返回值描述。

UML 盒被放置的位置表示他们说代表的类的关系。当一个盒在另一个的上面的时候，并且如图 6.11 箭头所指的方向，则表示上面的盒子所表示的类是下面的那个的超类。属于继承关系，有时，称为 is-a 关系，允许下面的类继承上面的类的属性和行为。

关系通常包括四种类型：归纳，关联，聚集和组合。继承关系实际就是超类将子类归纳的关系。比如：燃料类可能就是柴油机燃料类的归纳。

当两个类一起出现的时候，且一段时间内必须保持联系，则这两个类是关联关系。比如，如图 6.12 所示，每一个订货单都与一个售货员相关联。这个关联用一个直线描述，两端的数字表示参与关联的对象个数，在这个例子中，一个售货员可能收到一个或多个订货单。

当一个类是另一个类的一部分时，我们称为聚集关系。比如，在图 6.12 中，订购条目是订货单的一部分，我们用一条线和末端一个实心的菱形来表示。菱形指向较大的实体，另一端联接的类是一部分。同样用两端的数字表示基数。如图中表示一个订货单可能包括多个订货条目。

一个空心的菱形表示一个聚集关系，但是没有继承的关系。在这个例子中，一个顾客有一个订货单，但是顾客类不是订货单类的一个子类。图 6.13 表示了类间关系的其他几种方式。

UML 包括类与其它类联系的其他方式，包括一个指定类的注释。图 6.14 是如何用注释和限定词来加强类的联系。如果想知道有关类的更彻底的讨论和 UML 注释请看 Larman(1998)。

用这些记法，我们可以画出皇家服务站设计的第一步，如图 6.15 所示，是用面向对象解决方案所需的 15 个类。用 UML 记法来表示对类间关系的理解。

通过比较设计和 Sidebar6.1 中的需求，我们发现这个类图可以被进一步改善。首先，我们加入一个信息类，这样警告信和定期的信息是不同类型的信息。然后，我们注意到燃料类应该和清单类相关联，因为我们必须跟踪燃料和订货单，不超过所允许的最小量。

我们可以通过删除账目类来化设计，因为它只有一个属性，然后把账目数加入到顾客类中。此外，要添加三个新类：加油，停车空间和服务。这样服务类可以处理每种类型服务的价格和打折。通过使用服务类，我们可以利用面向对象提供的多态性；价格和打折是不同的，取决于需要的是什么服务。我们还可以从燃料类中去掉打折率，把它加入到部件类中。

通过删除燃料类中的价格，并把它加入到加油类中，则燃料类现在只跟踪数量，而加油类处理与一个特定的销售相联的加仑和价格。类似地，我们将停车空间的价格归入停车类中，位置归到停车空间类中。这样停车空间类跟踪那一种车辆可以停放的不同的位置，可以提供我们必需的使用趋势报告。最后的设计如图 6.16 所示。

接下来，通过前面的细查，我们发现站管理员类和其他的类并不关联；因此删除它。还发现，需要添加一个附加的类，服务站，用于处理待用账目。然后，我们为每一个类添加操作，以反映 Sidebar6.1 中的功能需求。比如，账单类需要一个计算整个价格的操作，还有一个计算购买税的操作。类似地，清单类需要一个和燃料、部件订购系统交互的操作：订购燃料，定购部件和停车空间的操作，还有一个告知是否有可用空间的操作。最后，我们添加图的基数，是我们对类间的关系有更好的理解。如图 6.17。

其他 UML 图

为了补充设计，我们还需要各种其他类型的图。首先，用类描述模板对每一个类进行更详细的描述。这个模板告诉我们整个层次中类的位置（根据继承的深度），出口控制，基数（就是，类中有多少对象），和其他类的联系。它也指明了类中的操作，和类的公共接口。这里有一个对加油类的类描述模板。

注意到类描述模板将基础工作放在了程序设计上。换句话说，类描述模板包括程序员实现设计所需的基本信息。比如，模板在类的操作中包含公式，还将公共接口和私有接口区分开。一个类的私有接口是一种限制对类的成员的访问权限的机制；这个类的属性和方法对其他的对象是不可见的。一个公共接口允许对方法进行访问，但是不能访问属性，因为一个公共的属性违反封装的原则。

UML 也包括包图，这样系统就可以被看成是很多包的一个小集合了，还可以被扩展成一个更大的类集合。包图表示出了当类属于不同的包时，它们之间的依赖关系。说当一个定义的改变会引起另一个的改变时，我们就所这两个条目是依赖的。比如，如果一个类给另一个类发消息，或是一个类需要另一个类的数据，或是一个类在执行一个操作的时候需要另一个类提供参数，这些情况下我们都说两个类是依赖的。特别地，如果两个包中有类是依赖的，我们就说这两个包是依赖的。好的设计中，我们要将不必要的耦合降低到最低；这样，我们可以将依赖程度降低到最低。

在测试的时候，包和它们之间依赖程度特别重要。在第 8，9 章会看到，我们必须设计测试用例来说明组件间各种可能的交互行为。UML 的包图有助于我们理解依赖关系和创建测试用例。图 6.18 就是皇家服务站的一个 UML 包图的例子。在图中，有四个主要的包：外部系统，顾客，产品和服务。每一个包都是由图 6.18 设计中的类组成的。比如，服务包是由 5 个主要的类组成：服务，加油，停车，停车空间和车辆维护；在图中它用来说明类和包间的联系的，但又不总是包括在包图中的。虚线的箭头表示包的依赖。比如，服务包取决于顾客包。正如你所看到的，包图给出了系统的一个顶层的概貌，并标注了顶层的依赖关系。

UML 还包括交互图，用于描述对象是怎样处理操作和行为的。我们通常从一个用例中得出一个交互图，来识别代表系统整个功能的信息和操作。

有两种交互图：顺序和协作。一个顺序图表示活动或行为发生的顺序。一个对象用一个盒子描述，且盒子位于一条垂直线的顶部，称为对象的生命线。生命线上的一个窄盒子表示信息的开始或结尾。两条生命线间的箭头代表两个对象间的信息，并且标上信息的名称和信息被发送必须满足的条件。箭头上的星号表示信息被发送很多次，并且是几个不同的接收对象。当信息箭头循环回它的起始的那个盒子的时候，这个对象正向它自己发信息；这种类型的信息称为自代理。如图 6.19 是皇家服务站的一个顺序图，表示的是加油类的一个用例。

一个协作图也是基于用户用例的，它表明的是对象的静态联接。就像顺序图那样，对象用图标表示，箭头是用来描述信息的。然而，不像在顺序图那样，协作图中信息的顺序是用编号来表示的。比如，图 6.20 中是一个停车用例协作图。停车信息被标号“1”，从顾客类到服务站类。然后下一个可用的停车位的信息由服务站类发送给停车空间类;它被标注“2”.在这个用例中总共有五中类型的信息要发送。

到目前为止，我们使用 UML 捕获了系统的静态特征。UML 还支持两类图：状态和活动，可以描述系统的动态模型。一个状态图表示一个对象所有可能的状态，触发状态转移的事件，和状态改变时所采取的行动。通常，一个对象状态和属性值的集相联系，事件发生在消息被发送或接受的时候。

一个状态图的注释类似于第四章中介绍的状态转移图。如图 6.21 所示，开始节点用一个黑点表示，而结束节点用一个小一点而的黑点外套一个白圈表示。一个矩形代表一个状态，一个箭头表示一个状态转移。条件是用中括号匡起标在箭头旁。比如，购买就不用在授权中考虑除非授权功能产生了一个“已付“的信息。

我们能可以建立皇家服务站的状态图。比如，如图 6.22 是燃料和部件类的状态图。我们注意到部件类的状态图和燃料类的状态图是相同的。这种类似可以用来简化设计。在一个类中包含两个相似的功能，我们把这些功能放在清单类中。

如图 6.23，是清单类，定购部件或燃料；不同的条件触发每一个状态。这个类追踪储量是否在正常水平或者是否低于指定的最小值。这样，可以使问题简化。正如我们说看到的，从不同的角度看待问题可以改善设计。

UML 用一个活动图来在一个类中建立过程或活动流的模型。当用条件决定去激活那一个活动时，活动图用一个决定节点来表示选择。图 6.24 说明了在 UML 中用到的活动图。开始节点和结束节点的表示方法和状态图中的一样。一个矩形表示一个状态，用箭头表示从一个

状态到另一个状态的转换。此例中，B 执行完后，必须做出一个选择。如果条件满足，将 X 输出到其他类。否则，C 或 D 可能被触发。C,D 上的长的水平线表示由 B 发出一个消息，以广播形式可以发送到 C 和 D。然后，C 和 D 可以被单独地，连续地或并行地触发。

我们可以为清单类画一个活动图，如图 6.25 所示。这个活动图可能有两个决定：一个是证实燃料是足够的，另一个是证实零件有现货。如果清单上他们中的任何一个短缺的话，这个活动图就触发定购部件或燃料的活动。我们注意到水平条允许这两个订购活动的初始化。这个可能会出现，如果一个顾客既需要燃料又需要部件的话。比如，一个顾客可能会访问皇家服务站，抱怨燃料质量低。Manny 和他的员工们可能判断出在燃料槽里有个洞。这样，他们可能选择装一个新槽，然后注满燃料。在这种情况下，燃料槽和更多的燃料可能要通过系统自动地定购，如果这两样的清单上面的量足够的话。

6.6 面向对象程序设计

一旦我们进入到系统设计中，我们可以用它们的加工建立我们的程序设计：编码者实现系统的设计。我们程序设计的努力从系统设计中的对象和类开始，但我们必须增改它们以包括更多的项目：

- 非功能性的要求，例如性能和输入输出限制。
- 从以前建立的系统的重用组件。
- 为更多系统而不仅限于当前系统的可重用的组件。
- 用户界面要求。
- 数据结构和管理细节。

可能在程序设计中比在系统设计中有更多的对象。例如：很多程序设计中的对象对于用户是不可见的，同时在真实世界中没有对应的实体。系统设计可以用高层描述数据组织层次和获得的方式，但底层的系统设计包括程序员必须知道的数据结构的信息（例如链表和数组）。在需求和系统设计阶段，我们不应该与用户讨论数据结构；这些数据结构在对问题的理解和产生高层的解决问题的描述中既不可见也不是必须的。但是在程序设计中，我们必须对数据结构做更多的具体的决定。也就是说，在程序设计中的抽象程度不同于在系统设计中。

在程序设计中，我们必须明确每个对象同世界上其余对象的接口的特征。尤其是，我们需要知道每个操作中的操作符（operation signature）,也就是说，我们命名每个操作，操作中作为参数的对象和操作的返回值。在很多情况下，我们可以从顺序图中得到这些信息，一个对象的接口是所有操作符的集合，一旦我们定义了这些接口，我们可以按类型分类，同时可以建立一个借口类型的的组织层次表,表示出某一接口从其他接口继承的部分，这个层次表有特殊的意义，以为对象之间指通过接口互相访问。

这种接口也允许我们利用面向对象的多态特征。正如我们看到的，一个对象可以得到一个请求并为不同的实例做出不同的解释。例如，一个多边形对象对于三角形和四边形可以计算不同的面积，当需求与对象在运行时的特殊操作相联系时，我们可以用带有自动绑定(dynamic binding)的功能的面向对象系统。类似的，不同的对象可以收到同种请求但用不同的方法处理。例如，一个皇家服务站的零部件预定对象的重新预定请求与其对燃料预定对象的重新预定请求是不同的。

一旦我们明确了接口，我们就可以开始进行实现，每一个对象是某个类的一个实例，当一个特定的对象被实例化时，编译器为对象所需的内部数据分配存储空间，然后将数据和操作相关联，尽管实现的大部分工作是程序员的，但必须由程序设计者选择构成类是否利用继承设计和如何利用。例如，程序设计通常在对象构造和类继承之间选择。当一个类从另一个类中继承了某些属性，父类的内在特点对子类是可见的。由于这种原因，继承有时被称为白

盒重用，但是，类也可以由组件类建立起来，很象小时候搭积木的过程。在这里，由构造建立的过程经常称为黑盒重用，因为对构造者没有构造模块的内在特点可见。

每一种构造模块均有优缺点。继承的方式在编译时给我们一个对系统的静态观察，同时可见性是修改变得容易且直接。但是，继承在运行时并不能简单改变，父类的任何改变都影响到子类。实际上，大范围的改变可以导致相关类的内在层次的变化以致使之难以跟踪和维护。

在另一方面，构造方法使用了在系统设计中构造的封装方式。也就是说，应该隐藏的内容保持隐藏。另外，构造方式使我们很容易替换构造模块，只要替换的类型是相同的这种技巧很象搭积木中用一个红块换一个绿块，只要他们是相同的大小和形状。因此，和继承的方式相比较，构造的方式的关联很少。但构造的方式有其自身的缺点，构造的方式系统在运行时自动定义所以它不能弄清那个对象引用其他的对象。对象间的接口必须非常详细的设计才可以使接口限制不妨碍在需要时使用特定的对象。

一种折衷的方式是构造方式更象继承方式的方法是，允许一个对象代表另一个对象的操作，例如，假定 A 对象向 B 对象发送一条消息，作为消息的接收者，B 可以把它自己发送给授权的对象 C，然后授权操作返回 B，这种结构听起来有些麻烦，并且很难懂，但实际上它考虑在运行时行为可以被构成，同时如果需要可以被很容易的改变。

设计辅助工具

没有适用于任何情况的设计方法。对所有系统都适用的唯一的指导方针是：为改变而设计。不管你设计的是什么系统，都可能在某些时候发生变化，有很多与面向对象相联系的技巧帮助你使系统变得灵活和易维护。另外，选择一个灵活的设计策略，你可以用工具包构造你的系统。一个工具包是一组互相关联的可重用的类，提供设计好的功能，工具包很象结构化语言中的子程序库，例如，你可以用一个工具包从按钮和窗口中构造一个用户界面，而不用重新自己写代码。

框架和模型也是设计中的辅助工具，它们不同与共举报以为他们更着重于设计重用，而不是代码重用。一个模型式一个抽象设计元素集（亚历山大　１９７９）的模板用来引导你进行自己的设计。原本用于构思描述建筑物和城市的构造要素，模板现在在实践中已经被认为是很好的设计模式，用类似于亚历山大的方法，２０世纪９０年代中期软件设计者认识到特定的同样软件设计问题并建议用模型的方式解决它。例如，假如你要构造一个存储数值间交互的演示系统，Gamma et al（１９９５）建议采用模型－视－控制（　model—view—controller）模型。模型是一个包括存储信息的类的集合，实施一个实现用户接口的类的集合，控制类控制用户接口和存储信息的交互。

每种模型都有一个上下文和驱动与之同时定义，上下文解释模型所适合的情况，驱动是上下文中在某种程度上变化的要素，如果你所处的条件与某一模型的上下文和驱动相匹配，那么这个模型就适合你的应用，但是，如果你的环境要求灵活，模型就无法使用。图６。２列举了几种产生模型的方法。

一个框架是一个明确域的设计的部分重用。它比一个设计模式更特殊化，同时它可能在它的规格说明中包含模式，框架必须在你创建的系统设计中扩充，扩充包括创建其他的类，用继承或构造的方式去补充在框架中已存在的功能。因此，模型是一个同类的骨架结构，而框架更象一个自己必须提供主要模块的自建工程。

Sidebar6.2　创建模型的方式

Gamma et al(1995)建议了几种创建模型的方法： 抽象工厂：这种技术提供互相关联或互相依赖的对象家族间的接口，它不明确指出具体

的类

构建者：这种方式把对象的构造和表示分开，同样的构造过程可以产生不同的表示方法。

工厂方式：这种方式定义一个产生对象的接口，但子类决定哪个类实例化，我们认为工厂方式延迟了类的实例化。

原型：这种技术用原型实例，并且用拷贝原型的方式创建新类。

单独创建：用这种方法，一个类只有一个实例，并有一个获得它的指针。

用户界面设计

让我们对面向对象设计中的几个具体方面进行研究，从用户界面开始。为实现一个程序设计的用户界面部分，我们必须考虑几个问题：

- 定义谁用系统进行交互。
- 为系统完成一项工作的途径提供说明。
- 为用户请求设计一个组织层次图。
- 改善用户与系统交互的次序。
- 设计在层次图中相关的类以实现用户界面设计。
- 在整个系统的类层次图中合并用户界面的类。

在用户界面设计的第一步，是将交互的情况在纸面上列出。为此，你必须决定在现存系统中的流程。文件中可以提出系统中的合理的用户界面，例如，图 6.26 左图显示纸面上的皇家服务站用的账单，那么，为自动完成记账过程，可以提出象右图的屏幕界面。

如果 Manny 同意屏幕设计，下一步包括设计一个或多个类实现屏幕界面，你可以用图 6.27 中的设计，注意设计中包括一个为 OK 按钮和文字框设计的类。这些对象不能体现你第一次与 Manny 进行需求分析时所得到的信息，他们体现出你的决定而不是她的。当你从生命周期的问题理解阶段发展到问题解决阶段时，对象和类的集合扩展是正常的，类和层次树的扩展是你作决定时考虑灵活设计的又一个重要原因。

数据管理设计

程序设计必须提出存储和恢复持续对象的方法，数据管理考虑系统性能和存储空间的要求。从我们理解数据要求和限制的角度，我们必须列出一个为对象及其操作的设计，我们可以用四步完成这个任务：

1．确认数据，数据结构及它们之间的关系。

2．设计为管理数据结构及其互相关系而作的服务工作。

3．找到工具，例如数据库管理系统，来实现某些数据管理任务。

4．设计类和类的层次树，来监督数据管理功能。

另一个面向对象的解决方式使用传统的文件或关系数据库，但它最容易与面向对象的数据库接口。举例来说，考虑一个跟踪车辆维护和拆卸的问题，如图 6.28 所示，如果我们用传统的文件方法，我们必须为每个类和程序连接建立文件以完成系统要求的工作，用关系数据库使我们的工作变得容易一点，如图，我们必须建立表，同时需要附加的表表示类之间的关系。例如，我们不仅需要一个拆卸和维护的表，同时，还需要一个车辆和拆卸部分之间关联的表，这必须在我们已定义的类之外增加，一个面向对象的关系数据库是最简单的，因为它用我们解决问题中现存的类实现数据管理。

任务管理设计

任务管理是程序设计中核心的部分，我们必须认真考虑需求并决定如何协调系统执行的任务，一个任务是指一个系统中的一个进程，它可能是事件驱动或时间驱动的，事件驱动任务是当某一特定事件发生时被启动，例如当鼠标移动时，按钮按下时，一个传感器发送或接收一个消息时。一个时间驱动任务是在某一特定时间时被触发，例如每 n 分钟执行一次。

我们可以用四个步骤来设计任务管理：

1．确认需要执行的任务，并分为事件驱动和时间驱动。

2．决定任务的优先级，也就是对每对任务，决定当两个同时触发时，那一个具有优先性。

3．创建一个任务，协调其他任务。

4．为每个任务设计对象，并且定义它们之间的关系。

每个任务必须正规定义，以便程序员理解如何适当的实现它，为帮助程序员，我们应该包括任务名，描述名，优先级，服务通信机制和在层次表的位置。例如定购零部件的任务可以如下定义：

任务名：定购零件

描述：目的是当没有库存的零件是自动定购新的零件。

优先级：高。当存货清单警告零件库存很少时必须激发该任务。

包括的服务：检查存货清单

管理者 / 管理：服务站系统

通信：与定购零件系统相连的调制解调器。

Gamma et al（1995）描述了几个可以帮助我们决定如何管理任务的模型，它们被列在 Sidebar6.3。让我们详细分析一个模型，以便理解模型如何用于程序设计。

观察者模型是当一个摘要至少有两方面，同时某些依赖于其他时可以使用。这种模型提出我们在对象间定义一个一对多的依赖关系，以便当一个对象动态改变时，它的相关联的对象被通知并自动更新。封装彼此分离的对象的各个方面有很多好处。当一个对象的改变需要其他不定数目的对象的改变时，封装使它容易改变并独立的重用对象。因此，对象间不存在强耦合；一个对象可以在不知道哪些对象需要通知时通知其他的对象。

Sidebar6.3　　gamma et al 为任务管理构造的模型

责任链：避免因为给不止一个对象机会去处理请求以致使请求的发送者和接收者相耦合。链接接收的对象并将请求沿着链传送知道某对象处理它。

命令：将一个请求作为一个对象封装，让系统对顾客的不同请求参数化，请求包括排队或登录请求，并支持不能执行的任务。

解释器：对特定的语言，按照语法定义一个表示符，可以是一个用表示符解释语言中的句子的解释器。

重复器：在不暴露基本的表示符的情况下，顺序获取集合对象中的元素。

仲裁者：定义一个封装一组对象如何互交的过程的对象，这个仲裁者对象通过不允许对象间明确互相引用的方式促进了弱耦合，它也允许对象间独立的改变互交。

回忆功能：不违背封装原则，捕获并外部化一个对象的内在状态，一时对象在以后的进程中可以被恢复。

观察者：定义一个一对多的依赖关系，以使当一个对象改变状态时，它的所有关联对象可以自动被通知并更新。

状态：当一个对象的内部状态改变时，允许它改变行为，这个对象将在改变类时出现。

策略：定义一个算法族，并封装每一个，使它们可以互相作用而改变对方，策略可以使算法独立改变，不受使用它的用户的影响。

模板法：在一个操作中，定义一个算法的框架，并将一些步骤交给子类完成，这种方式使子类可以重新定义一个算法中的某些步骤并不改变算法结构。

访问者：代表在一个对象结构上成员上所执行的操作，访问者允许再不改变操作的成员所在类上的情况下定义一个新的操作。

观察者模型包括四个主要的构造：一个主体，一个观察者，一个具体的主体和一个具体的观察者。主体认知它的观察者并提供一个链结合分离观察者对象的接口，它可以被若干观察者对象观察到。观察者的主要作用是定义一个为对象建立的更新接口，可以在主体发生变化是得到通知。一个具体的主体存储一个或多个具体的观察者所关心的状态，并在状态改变时通知观察者，同样，一个具体的观察者对象维护了一个对具体主体的引用，它存储了与主体状态相一致的状态，并相应更新其接口。图 6.29 显示了所有四个构造之间的关系。

我们可以花一个如图 6.30 所时的序列表显示四个构造之间如何互相作用，观察者对象观察一个或多个主体，我们要避免观察者纪录的主体状态与实际主体状态不一致的情况，相应的图示向我们显示一个具体的主体当使其状态与观察者状态不一致时的变化时，将通知其观察者。一旦一个具体的观察者被通知到，它可以向主体查询更多的信息，这些信息被具体的观察者保证纪录的和实际状态一致。

这种模型允许在主体和观察者之间的抽象耦合，并且支持广播通讯。但是，主体和对象必须频繁的监视它们的状态，以避免不可预料的更新导致不一致。

6.7 面向对象的测量

一旦我们实现了一个面向对象的系统，我们如何测量它的属性，例如，我们在第五章中希望得到的特征如低耦合和高内聚，同时我们也可以度量如设计复杂度之类的属性，如何度量在面向对象系统中的这些特点？这些测试方法是否对理解，控制，预测系统有效？这一部分，我们探讨一些在面向对象度量中的新的领域。

面向对象系统大小的度量

正如我们所注意到的，面向对象的度量在我们从需求分析到设计到编码到测试的过程中系统的规模逐渐变大，在这方面，它们与用其他设计方法的系统没有区别，但是，不象其他模式，用面向对象的方法我们在生命周期的每一步都用相同的或相似的语言，是从系统开端到交付使用到维护的整个过程，度量系统的规模变得方便。

研究者在测试系统规模时利用普通的词汇，并在系统规模的基础上作预测，例如，Pfleeger(1993),在他致力于预测方法适用对象和方法作为基本的测量单位，她发现，它的方法在预测建立几个面向对象系统所需的作为结果的工作量时，比 COCOMO 更精确。Olsen(1993)将这种技术应用到商业工程中，估测结果非常准确。用相同的度量单位在开发的全过程评估系统的规模的优点是很清楚的，在开发过程中估测的技术可以重用，同时，估测的输入是直接可以比较的，换句话说，很容易知道初始大小并在开发中追踪它的增长。

Lorenz 和 Kidd（１９９４）扩展了面向对象的系统规模的测试方法，生成一个更高水平上的详细理论。它们定义了系统规模的九个方面，不仅体现了通常意义上系统中的容量，而且体现出类的特征如何影响系统。每个用例与一个描述系统执行任务的说明书相关，Lorenz 和 Kidd 计算在用例中的说明书脚本的个数(NSS),它们报告说这个测量与测试范例中的应用的大小和数目有关，由于这个原因，NSS 在至少两方面有作用，作为一个测试尺度，

它可以输入一个估测的模型预测工程的效果或持久程度，作为一个测试范例的估测，它帮助测试组测试范例，并为将来的测试行动分配资源。

下一步，Lorenz 和 Kidd 历数了系统中关键类的数目，这个测试指标的目的是估测高层设计，提出构造系统需要多少工作量，它们也记录了支持类的数目，这个指标指向底层设计和工作量。每个关键类所需的支持类的数目在跟踪开发系统的结构中都会用到。

它们也根据每个独立的类来定义测量单位，类的大小是类中所有操作和属性数目的和；这里，我们将继承的特征和类中特有的特征一起计算，为估算继承的影响，它们计算了子类中重载操作的数目（NOO），也计算了子类中加入的操作的数目。通过这些，它们定义了一个特定索引 SI，定义如下：

SI=(NOO*等级) /（所有类的方法）

每个方法都可以在开发的不同阶段应用，如表 6.5 所示，用例脚本的数目和关键类的数目可以在开发早期测量，即在需求分析阶段，这种测量尺度比传统上为便于工程经理尽可能早的分配资源而采用的计算需求数量的方案更具体更精确。

表 6.5

度量	需求描述	系统设计	程序设计	编码	测试
脚本个数	X				
关键类的个数	X	X			
支持类的个数			X		
每个关键类的支持类的平均个数			X		
子系统的个数			X	X	
类的大小		X	X	X	
可被子类重载的操作的个数		X	X	X	X
可被子类附加的操作的个数		X	X	X	
特定索引		X	X	X	X

我们可以将这种测试方法用于皇家服务站的问题，图 6.31 显示了我们的面向对象分析问题的全貌，回顾 Sidebar6.1 中的图例，可以看到六个椭圆形图代表六个关键的系统中的部分，用 Lorenz 和 Kidd 的方法，说明脚本的数目也是六个。

我们可以通过在本章中创建系统的类的层次树来进行下面的测量计算。图 6.32 回顾了层次树图，并列举了 Lorenz 和 Kidd 方法中的五个值中每个值得最大值和最小值。我们可以一次作为基础评估随着系统扩展的增长情况。这些测量值也可以告诉我们特定改变对系统的影响，同时我们可以判断是否会使测试和理解变得更困难。

面向对象设计度量

Chidamber 和 Kemerer(1994)也为面向对象的开发设计了一套测试方法，他们的工作更着重于设计而不是系统的规模，所以它们补充了 Lorenz 和 Kidd 的工作。除了测量单位例如每个类的方法，继承的深度和子类的数目，Chidamber 和 Kemerer 测量对象间的耦合程度，以及类的响应，和方法中缺乏内聚性的程度。表 6.6 显示了每个测试参数在开发中如何获得和使用。因为他们广泛使用并成为同其他面向对象的测试指标相比较的标准，让我们进一步看一下 Chidamber－Kemerer 测试尺度的内容。

表 6.6

度量	系统设计	程序设计	编码	测试
每个类的加权方法	X	X		X
继承的深度	X	X		X
子类个数	X	X		X
对象间的耦合		X		X
一个类的响应		X	X	
内聚的缺乏		X	X	X

在第五章中无门探讨过设计质量的几方面问题，例如复杂度，正如我们在第十一章中的讨论，也有其他的方法测试一段代码的复杂度，Chidamber 和 Kemerer 没有明确测复杂度的方法，所以可以用自己认为恰当的方法。Chidamber 和 Kemerer 用复杂度作为计算每个类的加权方法的数值。

每个类的加权方法＝所有复杂度的和

也就是说，我们将所有方法的复杂度求和来确定类的加权方法数值，如果每个方法的复杂度为 1，那么每个类的加权复杂度就是系统中方法的数目。方法的数目和方法的复杂度揭示了构造一个类所需的时间和工作量，方法的数目越多，工作量也越多，对子类的影响也越大。

它们定义了一个类的继承深度为在类的层次树中从继承树的根到叶子的最长路径。一个类层次树的继承深度越深，越多方法被继承，则类变得越难理解和维护。类似的，子类的数目是指定类的直接继承者的数目，这个测量尺度也指示了维护，测试，重用的难易度。

我们可以将这些概念应用于皇家服务站系统，在图 6.33 中，我们可以看到与账单，购物，消息有关的部分设计。对于账单类，加权方法测量的值是 4，继承深度和子类数目均为 0。账单类的耦合对象为 2，因为账单类与购货类和警告信息类间耦合，但购货类的耦合对象为 4，正如我们看到的有四个箭头指向或离开代表这个类的方框，在第五章中谈到，在设计中过多的耦合是不合理的，我们要使每个类都最大程度的独立，以便其更容易开发，维护，测试和重用。

计算方法中缺乏内聚的程度更为复杂，假定一个类 C 有 N 个方法，从 M1 到 Mn，假定{Iij}是方法 Mi 用到的实例变量，可以有 n 个这样的集合,每个方法一个，我们定义 P 为 （Ir，Is）的数对集合，Ir 和 Is 没有相同成员，Q 是(Ir,Is) 数组对,Ir,Is 至少有一个相同的成员，正规定义如下：

P={(Ir,Is)|Ir∩Is=　}

Q={(Ir,Is)|Ir∩Is=　}

那么，我们定义 C 方法中缺少内聚的度量值为｜P｜－｜Q｜,（如果｜P｜比｜Q｜大），否则，值为 0。在更普遍的语言中，该值用没有互交集合的数目减去有互交集合的数目，这个测试指标基于如下观点：如果两个方法用同一个实例变量则这两个方法相似，相似的方法越多，类的内聚性越强，因此，LCOM 测量值是测量一个类中完全不同的方法的情况的一个方法。

类的响应是一组在类中的实体接收到一个消息时执行的方法，如果接收一个消息时，许多方法被激活，那么测试和修改将很困难，所以设计者需要经这个值保持在低水平，用较高的值指出类及其实现的附加检查。

我们也可以将 Chidamber－Kemerer 测量标准应用于程序设计，图 6.34 显示了皇家服务站系统的部份程序设计，我们可以从途中看出账单类的对象间耦合由 2 变成 3，但其他的测量值保持不变。我们可以用改变测量值的方法作为代码中相关问题的指示，例如，每个测量值作为指出代码中类似问题的一个指标：

。每个类中的加权方法值越大，在代码中越容易出错误（因为系统更复杂，也更难测

试）。

。子类的数目越大，代码中越容易出错误

。继承深度越深，代码中越容易出错误

。类响应值越大，代码中越容易出错误。

这些指标由 Basili,Briland 和 Melo(1995)估测 c++代码时得出,Li 和 Henry(1993)研究了这些测量标准，增加了两条：

消息传递耦合：在一个类中激活方法的数目。

数据抽象耦合：在一个被测类中使用，在另一个类中定义的抽象数据类型的数目。

Li 和 Henry 显示了这些测量标准如何在维护阶段用来预测类的大小的改变。

Travossos（１９９９）提出了面向对象系统的其他可能的测试标准，操作发送信息的数目可以计算平均的操作的大小，类似的，每个操作的平均参数的数目也是有作用的，每个操作自身都可以用一组复杂度测试估测自身的大小，就象在传统开发中的应用一样。这些测试方法在图 6.35 中列出。它是在皇家服务站系统中的一个顺序图，图中只有一个操作，这个操作有五个信息，最大和最小的操作大小是相同的，均是 5，我们可以看到第四步需要四个参数，而第一个操作需要 0 个参数，所以平均的操作树从 0 到 4 。

我们可以以一个类的描述中得到一个测试标准，例如，考虑一个重新加油的类的模板：

类名：重新加油

类别：服务

外部文件：

出口控制：公共

Cardinality : n

类层次：

上级类：服务

关联：

无名：与加油关联（未命名）

操作名称：价格

公共成员：重新加油

文件：／／计算加油最后价格

预定条件：加仑数 〉 0

对象表：未确定

语义：最终价格：加仑数＊价格

对象表：未确定

一致性：顺序

公共接口：

操作：价格

私有接口：

属性：加仑数

价格

实现：

属性：加仑数

价格

状态机：无

一致性：顺序

持续性：短暂

从这个模板中，我们可以发现 Lorenz 和 Kidd 的类的大小，用操作和属性相加的和来计算（包括继承的特征），这里，类的大小为 4 ，公共和保护类的比例为 0 ，公共获得的数据成员为 0 。我们也可以用传统的方法估计一个类的大小，例如，扇入 / 扇出法，这些方法可以帮助我们做出基本的估测和追踪系统开发的全过程，以便我们可以观察系统的增长和估测设定的改变。

在何处完成测量

有许多测量面向对象系统的方法，但最好的方法仍未发现，重要的是只有我们用它来扩充我们对系统的理解，预测和控制时这些标准才有作用。这里列出的标准或其他的标准以及你自己设计的标准都可能有所帮助，只有看获得这些数据的方便程度和你要解决的问题的相关程度，我们可以检验本章列出的这些测试标准是我们描述与面向对象有关的文档中最恰当的标准。

如表 6.7 所示，这些标准与六种文件有关：用例，类表，交互表，类描述，状态图和软件包表。这些测量标准通过使问题更明显支持系统开发，在本章中的皇家服务站的例子，我们可以见到图中的每个部分都揭示出一个问题或一个方面，能够增强我们对问题的理解。同样，每个测试的指标可以帮助我们，但多重的指标就可以提供综合的观点来帮助我们预测和解决在软件开发和维护过程中的诸多问题。

表 6.7

	用例	类图	交互图	类描述	状态图	包图
脚本个数	X					
关键类的个数		X				
支持类的个数		X				
每个关键类的支持类的平均值		X				
子系统的个数						X
类的大小		X		X		
每个子类重载的操作的个数		X				
每个子类附加的操作的个数		X				
指定索引		X				
类的加权方法		X				
继承深度		X				
子类个数		X				
对象间的耦合		X				
类的响应				X		
方法中内聚的缺乏				X		
操作大小的均值			X			
每个操作参数个数的均值			X			
操作的复杂度				X		
公共和保护的百分比				X		
数据成员的公共访问				X		
根类的个数		X				
扇入/扇出		X				

6.8 信息系统举例

第四章中 Piccadilly 系统中包括个广告促销活动的软件开发过程，在本章中，用一个事件的响应过程的模型，一个数据字典和一个数据流图来描述广告促销的过程，并明确处理广告促销中的需求。第四章的数据流图如图 6.36 所示，可以作为该例子的基础。

但是，从面向对象的角度，这个图并不能提供足够信息完成开发的过程，例如，广告活动中，需要预测电视收视率，这个收视率预测包括在流程图中，它从数据仓库中得到。但实际上，系统必须从历史纪录中预测，例如，收视率预测应该由某一机构从当时或以前的广告中和销售情况做出，上午 1 1 点比下午 7 点播出广要便宜，但收视率要低。所以我们要设计类去处理收视率问题。图 6.37 显示了一个处理收视率预测的两个类之间的情况，广告机构类得到消息去总结几个特定的促销活动，这个类依次激活预测类，让它计算平均收视率，并返回值给广告机构类，练习 6.4 检查了这个图和类去激活它们，但用对象的方法第一次试图定义这种操作就使问题比第四章的数据流和事件的响应图更为清晰，面向对象的方法使我们在一个更好的细节层次上考虑处理过程，并理解在某一特定时间，信息如何在系统中流动。用面向对象的方法，我们可以从很多过程的自然顺序中分离出来，取而代之以用关联数据激活一系列动作。

6.9 实时举例

主张采用面向对象的开发者通常强调对象的重用的潜力。如我们所见，Ariane—5 系统重用 Ariane— 4 系统的 SRI，如果 Ariane—5 系统采用面向对象的方法，重用将采用构造或继承的方法，用构造的方法，SRI 将被视作黑盒，并由主系统调用，返回在 Ariane—5 系统中另外的部分使用所产生的适当的值，用继承的方法，SRI 结构和行为被公开化，从父类继承。一个构造的方式不会避免问题的发生，因为黑盒不能使流量过大的问题可见。另一方面，继承方法可以暴露出 Ariane— 4 系统的问题，使 Ariane—5 的设计者采取措施。

6.10 本章对你的意义

本章详细描述了面向对象的开发过程，我们看到用例如何从需求分析中产生，提出了从各个不同角度看到的系统的功能，用例被用来产生系统设计，描述系统如何用来解决当前的问题，程序设计实现非功能性的要求，用其他细节帮助程序员实现系统。

我们提出了 UML 的基本概念，即联合模型语言，它是事实上的描述面向对象的系统的标准。同时，我们提出了一些测试指标用来测试系统规模和面向对象加工的设计特征，这些标准帮助我们不仅预测开发中的资源分配，测试和维护；它们也作为当系统变得复杂时的警告。

6.11 本章对开发小组的意义

面向对象方法对开发小组提供了一个从系统开发开始直到交付使用全过程的统一语言。需求，设计和编码均可以采用对象和方法来描述。术语的一致性是小组可以在设计完成前预测系统的规模和结构，它简化对加工过程的跟踪。例如，你可以看到在需求分析阶段的一个

对象在设计和编码阶段的情况

CML 支持定义的一致性，使你在创建自己的系统时，可以使用其它系统的面向对象的组件。定义的一致性使你的小组可以互相理解特定对象和类所指的内容，而且，一致性有助于维护和测试，使建立测试和监控更容易。因为需求，设计，编码都用同一方式表示，使估测设定的改变对系统的影响变得容易。

6.12 本章对研究者的意义

面向对象的方法对软件工程研究是个很广阔的领域，传统上衡量一个系统的方法对于面向对象的方法并不恰当或不全面。基于一行行代码的工程被预测的方法所代替，以利用类和层次树的特点。尤其是，错误预测的研究已经产生了更多的面向对象的方法来预测在设计和编码阶段存在的问题。

另一些研究者研究开发面向对象的最好的特点的方法，层次树的深度应该为多少？在什么情况下构造比继承更容易重用？如何最好的利用继承和多态性？与这些研究领域相结合的是找到测试面向对象开发过程和产品的方法。

6.13 小组工程

回顾为贷款分配系统所作的需求分析，如何将他们改变来实现面向对象的需要？这些改变如何能使系统变得更容易修改和改变？下一步，回顾面向对象的系统的七个特征，这些特征中哪些对贷款分配系统有利？最后，为贷款分配系统设计一个面向对象的设计，并比较以前你的设计，那个设计更容易测试和改变？

6.14 主要参考资料

有一些很好的能够提供详细指导如何用面向对象的方法来思考系统的书。Gamma 等人.(1995)提供了如何按照设计模式来思考系统的概括方法。像 Jacobson 等人 .(1995)，Rumbaugh 等人（1991），Booch(1994)提供了如何设计、实现面向对象系统的详细的方法，书中包含很多好的例子。Larman(1998)，Fowler、Scott（1999）提供了如何使用 UML 的初级读本。

经典的面向对象方法的参考书包括 Lorenz 和 Kidd(1994),Li 和 Henry(1993),Chidamber、Kemerer(1994)。Travassos、Andrade(1999)把面向对象方法的目的同面向对象开发的一些有效的指导规范结合起来。

6.15 习题

1. Manny，皇家加油站经理，打算把洗车也扩展到服务项目中来。它将是一个自动计算系统。顾客选择洗车方式并且给出车的款式。系统计算出洗车费并把数量显示在控制屏上。这样顾客就可以直接交付洗车费用。付完费后，如果洗车设备正处繁忙，那么顾客必须等待。否则系统会提示顾客应该驾车进入洗车间。使用本题对皇家加油站的描述，为洗

车设计设计一组用例。

2. 图 6.23 体现了皇家加油站的管理状态图。图 6.25 是其行为图。解释每个图对设计者、编码者、测试者所能看到的各是什么。为什么我们需要两种类型的图？
3. Lorenz 和 Kiddzh 的专门索引的重要性是什么？高专门索引的含义是什么？低又是什么意思？当产品升级了体现在索引上的主要变化是什么？
4. 按照使用例子的方法重写图 6.36 的数据流图。如何让图 6.37 的类适用到你的方案中？图 6.37 是本章所讲述的几种类型图的混合图。巴图 6.37 改写成用适当的 UML 符号表示。怎么使用 UML 图才可以使设计更好？
5. 面向对象的方法可以适用开发任何系统么？面向对象（OO）方法的优势是什么？它的缺点是什么？给出一个 OO 不使用的开发策略。

第 7 章 编写程序

本章，我们着眼于:

- 编写程序的标准
- 重用为指导
- 用设计组织代码
- 内部的和外部的文档

迄今为止，我们已经学习帮助我们理解客户和使用者的问题并且为它设计出高水平的解决方案。现在，我们必须专注于把方案转化为软件。换句话说，我们必须编写程序来实现我们的方案。这个任务因为一些原因或许是让人畏缩的。第一，设计者可能对平台和程序的运行环境的特性没有完全的了解；使用流程图和表容易表示的结构和关系并不一定易于编写成代码。第二，我们写出的代码不仅当我们进行复、查测试时可以理解，而且对于后来的系统升级者也要能够理解。第三，当我们编写代码时，我们必须充分利用设计者的组织，数据结构和编程语言的构成的特性,以编写出易于重用的程序。

显然，我们有很多途径实现我们的设计，有多种语言和工具可供选择；但是本书不可能涵盖所有。本章，我们提供了几种常见语言的例子，但是对于任一种实现，指导原则是相同的。换句话说，本章不是教你如何去编写程序，而是告诉你在编写程序的过程中要牢记的一些软件工程的实践经验。

7.1 编程标准和过程

在你的职业生涯中，你可能会开发很多不同的软件项目，使用不同的工具和技巧编写代码。你的一部分工作或许包含着评价现存的代码，因为你想替换、修改或是在其他应用中重用它们。你也许会参与正式或非正式的测试自己或他人代码的工作。这些工作同你在课堂上编写程序大不相同。课堂上，你的工作是独立的，所以你的老师可以判断它的质量和建议如何改进。然而，就更广阔的范围来看，大多数软件是由小组开发的，并且每一种工作都要求做出高质量的产品。甚至写代码本身，通常也牵扯很多人，所以协作是非常需要的。因此，不但

要求他人能够理解你写了些什么，而且要求理解你为什么写它们以及它是怎么适应你的应用。

因为这些原因，你必须在开始编写代码之前了解你们组织的标准和过程。许多公司坚持让他们的代码遵循一定的风格、规范和标准，所以代码和与之相关的文档对于每个阅读它们的人来说都是清晰的。

对于你的标准

标准和过程可以帮助你去组织你的想法和避免错误。某些过程包括证明你的代码是简洁和易于采纳的方法。这些文档是你不会因为长时间为接触工作而丢失已经做完的工作。标准的文档还可以帮助你定位错误并作修改，因为它阐明了程序中各个部分的函数的作用。

附注 7.1 微软的编程标准
在微软 Cusumano 和 Selby(1995,1997)研究了软件开发。他们指出微软在软件开发过程中，在保留了黑客们展示自己的创造力和个性的同时要融入一些软件工程概念。因此，微软一定找到了 “组织和调整成员工作的同时允许他们最大限度的发挥创造性，并且引出产品各个开发阶段的细节” 的方法。因为市场的压力和变化的需要，微软各小组重申在设计组件中，建立它们并测试它们。例如，小组成员在了解产品的功能时还要对产品的特色和细节提出修改。 然而，弹性不能排除标准。几乎所有的微软开发小组都在各自独立的地方工作，使用常用的开发语言（通常是 C 和 C++），公共的开发风格，标准的开发工具。这些标准有助于交流，讨论问题，解决问题。微软还要求各自的小组收集一系列测试结果，包括出错信息和发现的潜在和修改了的错误。当要继续开发核发行产品的时候，这些测试结果指导如何决策。

标准和过程还有助于把设计转化为代码。按照标准把代码结构化，你可以保留代码和设计中的相同部分。因此，设计的变化很容易就在代码中体现。同样的，因为硬件或是接口说明的改变而引起的代码的变化是简单的，并且出错的可能也会降到最低。

其它的标准

一旦你的代码完成了，其他人有可能在各种场合使用它。例如，我们将会在后续章节中看到，另外一个独立（不是编写代码的）的小组可能会测试这些代码。或者另外一些人或许会结合你的软件和其他的程序建立、测试一个子系统或是最终的整个系统。甚至在系统完成并在运行之后，因为发发现错误或是用户要求修改界面或功能，系统还需要修改。你也许不是维护和测试小组成员，所以组织、规范、文档化你的代码以使他人很容易的理解它是如何开发和工作的就变得非常重要。

例如，假设你们公司开发的每个产品都以描述程序的功能和与其它程序的接口为开始，那么文档的开始部分如下：

```
COMPONENT TO FIND INTERSECTION OF WO LINES
COMPONENT NAME:FINDPT
PROGRAMER: E.ELLIS
VERSION :1.0 (2 FEBRUARY 2001)

PROCEDURE INVOCATION:
CALL FINDPT(A1,B1,C1,A2,B2,C2,XS,YS,FALG)
```

```
INPUT PARAMENTRS:
    INPUT LINES ARE OF THE RORM
        A1*X+B1*Y+C1=0 AND
        A2*X+B2*Y +C2=0
    SO INPUT IS COEFFICIENTS A1,B1,A2,B2,C2
OUTOUT PARAMENTERS:
    IF LINES ARE PARALIEL,FLAG SET TO 1.
    ELSE FALG =0 AND POINT OF INTERSECTION IS (XS,YS)
```

这段注释告诉读者这段代码做什么并且给出了代码的总的描述。对查找可重用组件的人，这段给出的信息足以决定是否需要。对于查找代码错误源的，这段说明也给出足够的细节来决定它是否是出错源或是出错的参与者。

程序的维护者读了这段说明会发现修改也变得更加容易。一旦组件定位了，如果数据的名称是清晰的并且接口定义的好，维护者能够确定被修改部分不会对其他部分带来不可预料的影响。

自动化工具能够分析这些代码以决定哪个过程被此组件调用，哪个过程，调用它。因此，自动化工具产生的文档指出了那些组件调用它和他调用了那些过程。有了这些信息，修改系统是简单的。在本章的最后，我们做了一个标准和过程的实验，以测试它给我们的程序带来的效率。

匹配设计和实现

最关键的标准是指导程序设计和代码实现小组之间合作的需要。如果设计的模块化没有考虑到变为代码，设计过程只能获得很少的益处。设计要求，例如低耦合，高聚合和良好的接口定义，数据结构易于转化为代码等。

系统的基本目的在软件生命周期中要维持不变，但是随着客户的要求的变化，系统的特性也会变化。例如：假设你是计算机辅助汽车显示设计小组的成员。你的系统应该是汽车的一部分，但是菜单和输入设置或许会改变，或者新的特性会被加入。这些变化将导致高层设计和底层代码改变。因此，设计和编写代码的合作是非常关键的。后续章节，我们会看到测试、维护和配置管理没有这些相互关联的标准是不可能的。

7.2 编码方针

编码包括很大的创造性。设计是组件的功能或是目的指导，但是编程人员把设计实现为代码时时有很大弹性的。特殊的设计或要求或许建议一种编程语言，因为这要由设计者或是客户来直接决定。具体语言不在这里说明，因为有很多关于者方面的书。取而代之，我们讨论如何把一些规则应用到一般的程序设计中，而与语言无关。

无论使用什么语言，每个程序组件至少包括三个主要方面：控制结构、运算法则和数据结构。我们详细的测试每方面。

控制结构

许多组件的控制结构是由计算机体系结构建议或是设计的，我们想保持设计至代码。在这些体系结构中，像面向对象的设计，控制是系统状态的基础。另外，更多的程序设计，控制取决于代码本身的结构。对于任一种设计，程序代码结构反映设计控制结构是很重要的。

读者无需大范围的在代码中跳来跳去。他们应该集中注意于程序作了些什么，而不是在程序的控制流。因此，很多指导原则和标准建议写出的代码要能够从上至下很容易读懂。

我们来看一个如何重组代码以便于理解的例子。考虑下面的程序，其控制在程序的段与段间跳来跳去，使得很难清楚他的流程。

```
    Benefit=minimum;
    If (age<15) goto A;
    Benefit = maximum;
    Goto C;
    If (age<65) goto B;
    IF(age<55) goto C;
A: IF (age<65) goto B;
   Benefit=benefit *1.5+bonus;
    Goto C;
B:  IF (age<55) goto C;
  Benefit=benefit*1.5 ;
C: next statement
```

我们可以重新组织代码通过另外一种形式来完成这件事而使其流程容易读懂：

```
if (age<55) benefit =minimum;
elseif (age<65) benefit=minimum+bouns;
elseif(age<75)benefit=minimum*1.5+bouns;
else benefit=maximum;
```

当然，并不是总能够有一个完美的自上而下的流程。例如，到达一个循环的末尾或许会打乱这个流程。

我们看到在以前的章节中，模块化是一种很好的设计特性。把它引入代码中也会有同样的优点。模块化的建立一个程序，我们能够不同程度的隐藏部分实现细节，使整个系统易于理解、测试和维护。换句话说，我们可以把每个程序的组成部分进行模块化，我们可以使用宏，过程，子程序，方法和继承来隐藏细节同时提高程序的可读性。越多的模块化，越易于维护和重用；模块化和详细的宏，子程序或者其他子组件无关。

因此，编写代码时，牢记通用性是个优点；不要让你的代码过于有针对性。例如，要求你写一个在一段文字中查询长度为 80 的字符串的程序，你可以编写成包括两个参数字符串长度和字符串的程序。这样，这段代码就可以在查询任意长度和任何字符串的程序中得到重用。但同时，也不要因为过于一般化而降低了你的程序的可读性。

其他的设计特性，例如耦合和聚合。当你编写代码时，记住使用参数名称和注释来表现组件间的耦合度。例如，假设你编写组件来计算收入税。用总的收入扣除其他组件供给的数据。用 Reestmate TAX based on values of GROSS_INC and DEDUCTS 作为标记比用 Reestimate TAX 要好。Reestmate TAX based on values of GROSS_INC and DEDUCTS 说明了计算涉及的其他的组件提供的数据项。

你的代码必须是读程序者能够分辨出输出、输入参数。否则，测试和维护可能是件很困难的事情。换句话说，组件间的关联关系必须是可见的。出于同样的原因，作为系统的组件对其他组件隐藏了实现细节，你的子程序应该隐藏自己的计算细节。例如，前面的查找字符串的程序，搜索文本的子程序应该包含怎么搜索字符串的信息，但是调用它的子程序只需要知道字符串是否被搜索到或是在哪里而无需知道怎么查询。这些信息隐藏，允许你在修改查找算法时而无需修改其他代码。

算法

程序设计常常指定了一系列你在程序中要用到的算法。例如，设计时或许会告诉你使用快速过滤法，或是可能给你列出了快速过滤算法的逻辑步骤。然而，除受到语言表达和硬件的限制，你有足够的弹性实现算法。

对于程序的功能和效率很大程度上取决于你的决定。你的直觉告诉你要使程序尽可能的运行的快些。然而，运行快的代码会带来如下隐藏的代价：

- 编写运行快速的代码，可能会很复杂所以会花费更长的时间来写代码
- 测试代码的时间会更长，复杂的程序要求更多的测试用例和数据
- 用户会花费更多的时间去理解代码
- 必要的修改会花费更多的时间

因此，实现仅占了成本的一小部分。考虑到设计质量，标准和用户的要求，你必须平衡实现所占的时间比。特别地，千万不可牺牲正确性和清晰性来换取速度。

如果速度对你的程序是很重要的，你必须学会优化编译你的程序。然而，优化或许表面上加速了你的程序，但实际上可能使它变得更慢。为了清楚这种矛盾的情况为什么会发生，假设你编写的代码使为了实现一个三维数组。为了提高效率，你决定不用已有的一维数组的变址计算方法而自己编写所有这些方法。因此，你的计算代码如下：

```
index=3* i + 2* j + k;
/////////
```

数据结构

在编写程序时，你应该格式化并存储数据以使得数据管理和操纵更直接。这里有一些使用数据结构的技巧，建议你程序该如何组织。

保持程序简单

程序的设计阶段或许已经说明了实现函数中用到的一些数据结构。通常，之所以选择了这些结构是因为他们适合整个结构，能够提高信息隐藏和组件接口的控制。在组件范围内操纵数据能够影响你对数据结构的选择。例如：重新组织数据能够简化你的程序的计算。为了看清楚为什么，假设你正在编写的一个程序是计算税务的金额。作为输入，你应该给出应纳税的收入，如下：

1．第一个$10,000，税收 10%
2．第二个$10,000，税收 12%
3．第三个$10,000，税收为 15%
4．第四个$10,000，税收为 18%
5．多余$40,000 的金额，税收为 20%

这样，如果一个纳税人的收入为$35,000，那么他交第一个$10,000 的 10%（也就是$1，000），第二个$10,000 的 12%（也就是$1,200），第三个$10,000 的 15%（也就是%1,500），剩下金额的 18%（也就是$900），总税金为$4,600。为了计算这个税务，你可以在你的程序中包含下面的代码

……

然而，我们能为纳税义务中的每一个“括号”定义一张如表 7.1 的税表，其中每个“括号”都带有一个基础值和一个百分率。

通过这张表，我们可以相应的简化算法：

……

注意是如何通过改变数据的定义来简化计算的。这钟简化是程序更利于理解，测试和修改。

通过数据结构来确定程序结构

在税表这个例子中，定义数据的方法指导了我们如何进行必要的计算。通常，数据结构能影响程序的组织和流程。在一些例子中，数据结构还能影响编程语言的选择。例如，LISP 被设计成表处理机，它结构使其在处理表上比其它语言更有优势。类似的，Ada 和 Eiffel 语言中含有可以处理异常的结构。

如果一种数据结构先定义了一个初始元素，然后通过先前定义的元素的函数来产生后继元素， 我们就说它是递归的。例如，树一种由点和线组成的图，它的定义条件如下：

1.树上确定的一个点被指定为根.

2.如果将从根发出的线擦除,剩余的部分是一组不连通的图,其中的每一个图都是一棵树

图 7.1 为一棵树,图 7.2 展示了将根移除后得到了一组更小的树.每一棵小树的根都是一个节点,这些节点在大树中被原来的根所连结.因此,树通过根与子树被定义:一个递规定义.

像 pascal 之类的语言允许用递归过程来处理递归数据类型。既然处理这种数据类型的负担被从应用程序转嫁到编译器，你可能会愿意使用这种数据类型。因此，通常当你决定用何种语言来实现设计时，就应该仔细考虑数据类型。

总体指导方针

为了在代码中保持设计时的优点，几个总体策略是有用的。

输入输出本地化

程序的输入和输出部分是高度专业化的，且必须反映底层软硬的特点。由于这种依赖，有时程序的输入，输出部分功能是难于测试的。实际上，如果硬件或软件被修改，这部分是最可能发生改变的。因此，在组件中本地化这部分以和其余的代码相分离是适合的。

一个附加的优点是使总体系统具有普遍性。其它作用在输入上的系统级的功能（如从新初始化或类型检查）能被包含在特殊的部件中，以减轻其它组件的负担，减少重复。同样的，把输出功能放在一起能使系统易于理解和改变。

包含伪代码

设计通常为每一个程序组件提供一个框架。然后，你用自己的专业知识和创造性来编写代码以实现设计。例如，设计可以是相对语言独立的，在使用特殊语言结构上给你多种选择，怎样使用，数据如何被表示，等等。既然设计是概述在程序组件中要做什么，在从详细设计到编码中加入这个阶段要好于直接将设计翻译成代码。

伪代码被用于使设计适应于所选的语言。通过采用结构和数据表示，而不是直接将其转变成特殊的命令，可以试验并确定那一种实现是最适合的。用这种方法，代码能以最小的改变量被重排和重构。例如，假定设计一个文本处理系统的组件。

……

根据所选的语言和程序员的偏好，这个设计可以有多种实现方法。在从设计到编码的第一步中，可以检查每个命令的类型以确定在编码中什么需要被包含其中。通过使用中间伪代码，可以用下面的方法描述想让代码作些什么。

然而，重新考察这些伪代码，可能发现有些步骤可以被从新组织以使特定的通用函数可以被依次排列。

……

用这种方法描述命令后，会发现在第一个和第三个命令集合中应用了相同的一组命令。另外，应该注意到除了在命令 PARAGRAPH 中，line pointer 都在 margin 的左侧。有了这些

信息，我们可以写出更详细的伪代码：

……

最后准备写代码来实现设计：

……

因此，伪代码充当一个框架的角色，在其中构建代码。简述通过这种方法从设计到编码，值得注意的是设计的组织被改变了几次。这些改变必须通知设计者，并经过其同意，以使需求分析，设计和编码中的联系能被记录并被保持。

修订，重写而非修补

编码时，就像写一篇学期报告，或完成一件艺术品，通常要先写个大概的草稿。然后修订，重写直到对结果满意为止。如果发现控制流混乱，或是判定过程难以理解，或是有无条件的分支难以消除，应该回到设计中。重新检查设计，确定所遇到的问题是重设计中继承下来的，还是在编码时产生的。重看一下数据的表示与结构，算法的选择和分解过程。

重用

有两种重用：过程重用，我们设计的过程可以在以后的应用程序中被重用；用户重用，使用以前为其他工程开发的组件（Barnes and Bollinger 1991）。依照某种评价和改变组件的标准，一些公司有部门间或公司间的重用程序。Sidebar7.2 描述了在 Lucent 中的一个这样的程序。这些程序的成功表明提出一个重用的指导方针。如果你是当前工程的一个用户，有四个关键的特点来检验你说要重用的部件：

1.这个部件是否提供你所需要的功能和数据

2.如果需要较小的修改，修改的费用是否少于从新编译部件

3.该部件是否有完备的文档，是你不必逐行的阅读代码就可以理解它

4.是否有该部件测试的完整记录和历史版本，是你可以确信其没有缺陷。

同时还必须估计为使系统能与可重用部件相结合所需写的代码量。

另一方面，如果你开发可重用部件，必须清楚以下几点：

1.使用参数和可预计的条件，使系统可以激活你的部件。从而提高部件的通用性

2.将可能改变的部分与可能不变的部分相分离。

3.明确定义部件接口并保持其通用性。

4.记录任何发现的缺陷即改正的信息。

5.使用清楚的命名规则。

6.记录数据结构和算法。

7.保持通信与错误处理的隔离和易修改性

7.3 文档

许多公司和组织都关注伴随程序搜集的描述。我们认为程序文档是一个向读者解释程序要做什么，如何去做的字面描述的集合。内部文档是指在代码中书写的注释，其余的为外部文档。

内部文档

内部文档包含的信息直接面向读程序源代码的人。因此提供概述信息描述数据结构，算法和控制流。通常，这些信息被放在部件开始部分的名为头注释区注释内。

头注释区

就像一篇优秀的新闻报道，包含一个故事的时间，地点，人物，起因，经过，结果。在每个部件的头注释区中必须包含以下信息:

1．部件的名称
2．部件书写者
3．部件适用于通用系统的那部分设计
4．部件的书写时间和修改时间
5．部件存在的原因
6．部件如何使用它的数据结构，算法和控制流

我们进一步的看一下这几条信息

首先部件的名字必须在文档中被显著的指出来。其次标明书写者，应有他的电话号码或是 e_mail 地址，以使保存和测试小组在出现问题或征求意见时可以与之联系。

在系统的生命周期中，由于修改错误或是需求发生变化，部件经常要被更新或修改。如在 11 章中所提到的，跟踪以前的版本是很重要的，因此在程序文档中应该记录修改及由谁修改的日志。

由于部件是大系统的一部分，这个区域应该表明这个部件是如何适应部件层次的。这个信息有时被一个图所表述，在其它时候，应该由简单的描述。头注释区还应该解释部件是如何被激活的。

还需要更细的信息来解释部件是如何完成它的目标的。区域应该列出如下信息：

- 名字，类型，每个主要数据类型和变量的目的
- 逻辑流，算法和错误处理的简要描述
- 需要的输入和可能的输出
- 测试的帮助和如何使用
- 期望的扩展和以前的版本

你的组织的标准通常规定了头注释块的顺序和条目。下面是一个典型的头注释：

……

其他的注释.

头注释块起到介绍你的程序的作用，就像是介绍一本书的写作目的。附加注释在读者浏览程序的时候起到提示的作用，帮助他们理解你在头注释中提出的目的用代码是如何实现的。如果代码的结构反映了一个结构良好的设计，语句是结构清晰的，标志，变量的名称和数据名称的描述易于理解，那么必需的附加注释是很少的。也就是说，遵循简单的格式和结构规则的代码是其本身信息的来源。

甚至对于结构清晰书写规范的代码，注释也是不可缺少的。尽管对于结构清晰的代码可以减少必需的附加注释的数量，但是附加注释无论写在什么地方都是有用的。另外逐行的解释程序做什么，能够把代码分成体现其主要动作的小段。每个动作又可以分解成更小的，长度仅有几行的小步。程序设计中的伪码可以起到此作用。当代码修改时，程序员应该更新注释一反映其变化。这样，注释对程序的每个修改版本都建立了纪录。

最基本的注释要反映代码的作用。另外，要保证注释反映的是新的信息，而不是从代码中可以直接看出的诸如标志和变量名称。例如：这样的注释是毫无意义的：

```
//Increment i3
i3=i3+1;
```

你可以通过写下面的注释来添加一些重要的信息：

```
//Set counter to reader next case
```

i3=i3+1
理想的情况是，变量的名称可以反映其作用：
case_counter = case_counter + 1;

通常，你通过把设计转化为伪码来编码时，此过程提供了你的代码的最后的框架和最基本的注释。一定要在携代码的同时写注释，不要在写完代码之后，因此你可以描述你的设计和你的意图。要注意难写注释的代码，通常这个困难表示你的设计应该变得简单一些。

有意义的变量名称和声明标志

我们选则变量和标志的名称要反映其意义。例如写
weekwage=(hrrate * hours) + (0.5)* (hrrate) * (hours-40)
比下面这种写法要有意义的多
z=(a*b)+(0.5)*(a)*(b-40)
事实上，weekwage 的例子根本不需要注释。

同样地，按照字母顺序的声明标志应该告诉读者你的标志的目的。如果标志一定要是数字的，要使其是按升序排列的并且目的相关的要放在一起。

格式化以提高可理解性

注释的格式能够帮助读者理解代码的目的和目的是如何实现的。代码的缩进和空格能够反映基本的控制结构。下面的就是未缩进的代码：
.....
通过缩进和重新组织空格能够更清晰：
代码略(p324)

另外使用格式表现控制结构。Weinberg(1971)建议这样格式化你的代码，注释放一边，代码放在另一边。这样， 当测试代码时可以忽略注释，这样不会受误导。例如，下面的代码（摘自 Lee 和 Tepfenhart 1997）即使仅看左面的代码也可以理解。
代码略(p324)

文档数据

对于程序阅读者来说，最困难的事情之一就是理解数据是如何组织和使用的。一个数据映像在理解代码的作用时是很有用的，特别当要处理多种类型的文件而且有很多标志为和参数的时候。这个映像应该和数据字典在外部文档中结合起来，阅读程序者可以通过代码的要求和设计来跟踪数据操作。

面向对象的设计使这些问题的重要性减小了，但是有时候这些信息的隐藏更难正确的理解数值是怎么变化的。所以，内部文档应该包括数据结构和使用的描述。

外部文档

鉴于内部文档是简洁而且是为相应水平的程序员写的，所以外部文档是为那些甚至是没有读过代码的。例如，设计者在要修改或者升级程序的时候也许会阅读外部文档。另外，外部文档给你一个合理的而且相对在程序注释中可以更详细解释问题的机会。如果你把头注释看作程序的概要，那么外部文档则是对程序的详细介绍。它回答了—谁、为什么、何时、何地和如何使用此系统而不是一个组件或展望。

因为软件系统是由互相关联的组件构成的，外部文档通常包括系统中的组件的概述，或是一些组件组（向用户界面组件，数据库管理组件，或者地速计算组件）。图，与之相关的描述每个组件的注解语句，表明了数据是怎么被一个多个组件使用和共享的；通常综述描述信息是怎么从一个组件传递到另一个组件的。对象类和他们的继承层次也在这里说明，因为也是属于类型定义和数据范畴。

外部组件文档是系统文档概述的一个部分。在组件被编写的时候，相关的组件的结构和数据流已经在设计文当中进行了详细的描述。从某种意义来说，设计是外部文档的框架，细节由描述讨论组件的特性提供。

描述问题

代码文档的第一部分，应该介绍这个组件将要解决什么问题。这一部分是描述哪些方法被考虑作为解决方案和为什么选择这个解决方案的基础。问题描述不是对需求文档的重复，应该是综合的讨论设备，介绍什么时候被调用和为什么需要他。

描述算法

一旦清楚为什么该组件应该存在，就应该介绍算法的选择。应该解释组件中用到的每个算法，包括公式、边界或是特殊的条件，甚至是它的派生或是对参考书或参考论文的说明。

如果算法处理特殊情况，一定要讨论每一种情况并且要阐述是怎么处理的。如果因为他们不希望被遇到的特殊的情况，从而没有给出解决方法，那么就要给出合理的出错处理代码。例如算法可能包括一个含有一个变量出另一个变量的公式。文档就应该描述出除数是零的情况，指出当这种情况发生时，怎么处理。

描述数据

在外部文档中，使用者或是程序员应该能够看到组件层次的数据流。数据流图应该应该和相应的数据字典索引互相补充。对于面向对象的组件，对象和类的概述应该介绍其接口。

7.4 信息系统实例

回忆第 5 章研究的 Piccadilly 系统的设计。设计的一方面包括如何竞争电视节目频道上，所以我们应该使用广告信息战。部分设计描述是这样，确定程序的时间表就像 Piccadilly 系统中趣事一样。

…….

这部分的设计是由图表提供，包括数据字典定义：

…….

编写代码实现 Piccadilly 系统时，我们必须决定组件的级别。我们仅研究它们中的一个：

假设我们想用面向对象的语言就像 C++实现系统。我们创建类层次，包括类的 Opposition Schedule，依照数据字典 Opposition 表应该具有 Television company name, Opposition transmission date 和 Opposition predicted rating 这些特性。我们想创建匹配和 Episode 敌对节目的类 Opposition Schedule 的方法。其中一个决定就是我们必须搞清楚怎么把关于 Episode 的信息传回请求它的组件去。这里有一些参数传递机制可供我们选择，包括传值、引用、指针或是数组。

传值，Episode 的值实际上没有改变。相反，只是在本地栈做了一个数据拷贝。一旦方法终止，本地值对方法是不可访问的。使用这种方法的一个优点是调用组件时不需要保存参数的值。然而，如果参数很大的话，传递参数要花费很长时间和很大的空间。此外，如果参数必须改变的话，这种方法是不可行的。如果我们使用传值来实现组件的方法，在 C++中如下：

……..

另一种传递参数的方法如指针。这里对参数不是拷贝，相反，这种方法接受的指向 Episode 的实例。此种情况下，参数的实际值是可以改变的，程序可以调用 Episode 的方法。其参考代码如下：

…….

最后，参数还可以以引用来传递。这里也不是对参数做一个简单的拷贝，相反函数接收的是参数的地址。这种情况下，参数的实际值使可以改变的。此种情况的示例代码如下：

……

一旦你决定你的传递参数的方式，你应该在你的外部文档和程序注释中都体现出来。这样，其他的程序员如果修改或是升级程序时会很好的理解你的意图。

7.5 现实的例子

我们已经看到 Ariane-5 软件的主要问题是它需要正确的处理错误。这种情形可能和选择的实现语言有关系。Coleman 等人（1994）讨论一些面向对象语言和他们处理出错的能力。

一个很流行的处理出错的方法是发现异常。一个异常是一种情况，当发现时，处理该情况的部分叫异常处理者。依次，异常处理者调用以设计的代码去整理隐含的错误或是使系统到一个正常的状态。

Eiffel 语言（Meyer 1992b）包含了外在的异常处理机制。当函数执行时，如果异常激发了，有特殊的代码，称为援救代码，它来调用处理问题。我们必须设计决定如何让援救代码工作。有时援救代码修复了问题并尝试重新调用此方法。另一种情况，援救代码完成了其工作接着把控制权交给另外的异常处理者；如果问题不可能被修复，那么系统处理交给一个可以安全终止程序的状态。在一个设计合同中，合同包括预处理，声明和预定义环境。为了处理异常，Eiffel 语言还包含其他的解释声明出错时系统的预定义环境。这样，Ariane-5 代码，已经在 Eiffel 语言中实现了。

另一方面，C++编译器没有标准的异常处理机制。Coleman 等人(1994)指出 Eiffel 风格的异常代码可以实现如下：

……

无论什么语言和异常处理策略的选择，最重要的原则是在同一个系统中不可以有两种不同的方法。

7.6 本章对你的意义

本章中，我们讨论了程序实现的一些原则。我们已经看到编写代码时，你应该注意以下的原则：

- 组织的标准和指导原则
- 代码的可以在其他项目中重用
- 代码可以在更进步的项目中重用

- 使用最初的设计作为初始框架，可以直接从设计过渡到代码
- 同一系统中使用同一种出错处理策略
- 注意内部文档，外部文档解释程序的组织，数据、控制、函数要像解释设计目的一样
- 代码重要保留设计的特性
- 根据设计特性选择实现语言

7.7 本章对你的开发小组的意义

尽管很多编程是个人的努力，但是所有的程序都要由小组来控制。你适用信息隐藏时你可以仅暴露关于你的组件的最重要的信息，所以你的同伴可以调用和重用他们。使用标准，提高小组成员的交流。使用普遍的设计技巧和策略是你的代码更容易测试、维护和重用。

7.8 本章对研究者的意义

编程的很多方面是值得我们研究的

- 我们需要好的程序设计者特征的更多的信息。生产率可以和 10 个因素有关，质量同样是个变数。理解关于他们更多的信息可以帮助我们训练出高效率的开发人员
- 判定组件是否具有很好的可重用性是困难的。评定标准需要帮助理解什么样的组件有好的可重用性
- 我们继续需要更多的研究语言的特性和他们对产品质量的影响。例如：Hatton(1995)介绍了 C 语言中一些不可理解的方面应该避免以使软件更加安全可靠。
- 自动化工具在生成一致性代码、管理代码库、加强设计约定和提供标准代码结构模板方面有很大帮助。研究者不仅要继续开发新工具还要评估现有工具在实际项目中使用价值。

7.9 条件方案

前面的章节中，你已经测试了对 Loan Arranger 的要求，做出了一些设计以实现你所描述的问题。现在是实现你的系统的时候了。一些方面是值的特别注意的，因为他们比要求描述的看起来更加困难。一个是要求同时有四个借款分析员使用此系统。这个要求可以这么实现，当他们同时使用时可以锁住某些特定的借款分析员。实现这个要求的最好方法是什么？

第二个实现上的困难是实现束算法时。你的代码怎么发现最好的束去适用标准？

最后，语言的选择也影响实现要求的性能。

7.10 主要参考文献

Kernighan and Plauger(1976,1978);Hughes,pfleeger,and Rose(1978);Barron and Bishop(1984);Bentley(1986,1989),abd McConnell(1993)

世界范围的可重用库的一些如下：

- ASSET(Asset Source for Software Engineering Technology)
- CARDS(Central Archive for Reusable Defense Software)
- COSMIC(Computer Software Management and Information Center)

DSRS（防御软件知识库系统）

希尔空军基地软件技术支持中心

欲需这些或其他重用资源的联系信息可在书的电子版找到

7.11 习题

1. 任何一们编程语言中，如果某个语句根据变量的值能够分支到一些区域中的一个，我们把此语句称作计算类型语句。探讨一下计算类型语句正负方面。特别是，它是怎么样影响控制流的？可维护性？重用性？

2. 如果某人编写了一个组件，其他人修改了，组件出现故障谁将负责？重用别人组件在法律和伦理上意味着什么？

3. 列表是一种能够递归定义的数据结构。给出一个列表的递归定义。如果你熟悉一种带有递归过程编程语言（例如 LISP 或 PL/I），解释一下在语言中如何加入删除元素。

4. 给出一个实例，说明为递归设计的语言在哪些方面比没有这种设计的语言在处理列表上容易理解。

5. 有人要求你编写一个打印全年日历的程序。用户输入希望的年份，输出该年的日历。论述一下内部数据的表示如何影响到程序的编写。给出几个程序中可能遇到困难的数据结构实例（提示：数据结构是否是累积的？闰年如何处理？）

6. 常见的二次方程求根算法需要考虑几个特殊情况。书写恰当的算法注释能够容易辨别不同的情况和处理方式。书写外部辅助文档解释该算法。

7. 找到你熟悉的操作系统页面算法。书写外部算法文档，为用户解释页面转换是如何实现的。

8. 换一种角度研究你上交的程序。它能否用本章中介绍的建议改进？如果这样，怎么改？实施这些建议使你的程序高效与否？

9. 在你的组织机构中用同一个标准语言和工具贯穿这个工程，有那些优势和劣势？

10. 当代码组件由工具自动执行或从程序库中重用时，怎样执行代码，文档和设计标准？

11. 怎么在文档中书写面向对象程序的控制流？

第 8 章 测试程序

在本章中，我们将研究：
错误类型和如何分类错误
测试的目的
单元测试
集成测试策略
测试计划
何时停止测试

一旦你编好你的程序组件，就是测试它们的时候了。有许多种方法测试，这张和下一章将为你介绍几种测试方法，它们能够使你为用户发布一个高质量的系统。发现错误的发生并不是对于测试来说已经不是第一次了；我们已经看到了在开发过程中怎样在需求和设计的复查中帮助我们探索错误。但是测试是以发现错误为核心的，有许多种方法使我们在测试上下的功夫更有效率更有效果。在本章中，我们了解单独测试组件和组合组件来测试接口。然后在第九章，我们集中技术测试系统的整体。

8.1 软件的错误和失败

在一个理想的情况下，我们程序员都很在行，以至于我们编得程序每次运行都工作良好。不幸的是，理想不是现实。两者的区别是如下的条件导致的。首先，许多软件系统处理大量的状态、复杂的公式，活动和算法。另外，当用户对需求不是十分确切的了解时，我们经常用工具来为用户的系统下定义。最后，工程的大小和参与的人员增加了复杂性。这样，错误的出现不仅是软件而且也是用户和客户期望所导致的。

当我们说我们的软件出错了是什么意思呢？通常，我们指的是软件不能按照需求描述的工作。例如，需求描述指出了系统仅当用户有权浏览数据时才能进行特定的查询。如果程序允许非授权用户查询，我们就说系统失败了。失败可能是如下几条原因导致的结果：

需求描述可能错了或者有遗漏。需求描述可能没有完全阐述客户想要的或需要的。在我们这个例子中，客户可能实际需要几类的授权，每类的授权带有不同的访问权限，但需求描述没有详尽的阐述。

需求描述可能存在对于给定的硬件软件难于实现的需求。

系统设计可能存在错误。可能数据库和查询语言设计不能授权用户。

程序设计存在错误。组件描述中可能存在有不能正确处理此类情况访问控制算法。

程序代码可能错误。代码不能完整或者适当实现算法。

这样，系统一些方面上的一个或者多个错误导致了失败。

不管我们写程序多么出色，有一点可以肯定，从可能错误多样化上我们应该检查组件确保我们正确编写代码。许多程序员把测试看成对程序恰当执行的一个论证。然而，论证正确性的想法却是和测试的内涵背道而驰的。我们测试程序是要发现错误的存在。因为我们的目的是发现错误，当我们发现错误或者测试过程中导致了失败的发生，我们才看作是一个成功的测试。**错误鉴定**是确定什么错误导致失败的过程，**错误纠正**或者**清除**是修改系统除去错误的过程。

当我们编写完代码开始测试程序组件时，我们希望需求描述是正确的。而且，运用了前面几章描述的软件工程技术，我们尽量保证系统和组件的设计反应需求搭建了实现的基础。然而，软件开发周期阶段不仅涉及计算技巧而且涉及人与人之间交流和协作的技巧。完全有可能软件中一个错误的产生是由于早期开发活动中误解而导致的。

首要记住的是，软件错误不同于硬件的错误。桥梁、建筑物和其他工程建筑可能由于劣质的材料、差强的设计、构件的老化而失败。但是循环在上百次的迭代后不会老化，参数不会从一个构件到另一个构件传递中丢失。如果一段特殊的代码没有恰当的工作，如果一个劣质的硬件不是错误的根源，我们可以确信代码中存在错误。由此，许多软件工程师拒绝用 bug 这个词来描述软件错误。把错误叫做 bug 意味着在代码与代码之间已经使开发者无法控制此错误。在搭建软件中，我们运用软件工程经验来控制代码的质量。

在前一章中，我们进行了多次实践来最大限度减少需求分析和设计时错误的出现。在这一章里，我们练习最大限度减少编码中错误发生的技术。

错误的类型

在编写完程序组建后，我们通常检查代码，查出错误，马上清除。当没有明显的错误出现时，我们就在一些非计划条件下测试程序，看看能否孤立出更多的错误。这样，对于我们搜索错误的类型时非常重要的。

当一个算法组件由于运算步骤的错误对于给定的正确输入不能产生正确输出时一个 产生了。这些错误有时容易挑出来，仅仅通过通读程序（**desk checking**）或者在程序固定的运行时输入各类的输入数据。典型的算法错误包括

分支过快

分支过慢

测试错误条件

忘记初始化变量或者使循环递归

忘记测试特殊条件（例如当除以零时发生）

比较两个不恰当数据类型的变量

当查找算法错误时，我们也搜索语法错误。这里我们要明确的是我们已经正确的运用了程序语言的结构。有时，一个看似平常的错误将导致灾难性的结果。例如,Myers(1976)指出

第一次美国 Venus 任务的失败就是因为在 Fortran do 循环中缺少了一个逗号。幸运的是，编译器为我们捕捉了很多语法错误。

计算和**精度**的错误在公式的执行或者不需计算所需精度结果的情况下发生。例如，把整型和浮点型变量放在一个表达式里可能产生意料之外结果。有时，不恰当运用浮点数，不期的截断，操作的顺序都可能导致不可接受的精度。

当文档和程序实际运行不匹配，我们说程序有**文档错误**。经常文档来源程序的设计，文档提供了一种清晰的描述——程序员想让程序作什么的，但是实现这些功能时产生错误。这些错误在程序的生命周期中将增加其他的错误，因为我们大多数人在检查代码修改错误时都是相信文档的。

需求描述通常详细记录了系统中用户和设备的数量，通讯的需求。有了这些信息，设计者经常仅仅考虑系统的特性而不考虑可能超过需求中描述的最大负载情况。一些特性诸如队列长度、缓冲区大小、表的维度等等将贯穿程序设计始终。当这些数据结构超过其负载大小时将发生**超载错误**。

同样，当系统运行达到规定上限时，系统的行为不可接受，**容载**、**边缘错误**发生。例如，如果需求描述规定系统必须处理 32 设备，程序必须经过所有 32 个设备运行时监测系统的行为的测试。而且，系统还要在可能的配置下测试超过 32 个设备运行时系统的运行。通过测试和文档记载系统对超载的反应，测试小组能够帮助维修小组确定在未来增加系统容载的可能性。容载状况跟磁盘访问的数量，中断的数量，并发运行任务的数量，诸如此类的系统相关指数相关。

开发实时系统时，首要考虑的是协调一些进程并行执行还是按照预定义顺序执行。当代码不足以协调这些事件时，**时钟**和**协调错误**发生。造成这种错误难于鉴别和纠正出于两种原因：首先，设计者和程序员难于预料系统可能的所有状态。第二，由于诸多因素牵扯时钟和运行，不可能在程序发生错误后复制错误进行纠错。

当系统没有按照需求规定的速度运行，**吞吐**或**性能错误**产生。时钟错误有不同种：用户需求在系统行为上强制时钟而不是协作。//不动

正如我们在设计和实现时所看到的，我们下很大的功夫保证系统能够从多种的失败中恢复过来。当失败产生时，系统没有按照设计者期望或者用户需求运转下去，**恢复错误**产生。例如，系统运行时关闭电源，系统应该按照一个可以接受的方式恢复，诸如恢复所有文件到失败前的状态。对于一些系统来说，一些恢复意味着系统将运用一个恢复电源源继续所有的系统的运算；对于另一些来说，恢复意味着系统记录下处理的日志，当电源恢复时继续运算。

对于许多系统来说，一些硬件和相关系统软件在需求分析中都是规定好的，组件是根据那些重用或者购买的程序的规范而设计的。例如，一个调制解调器用于通讯，调制解调器的驱动程序行使调制解调器的命令和解析调制解调器发来的命令。然而，当配备的硬件和系统软件不能在文档规定的条件和过程下工作，**硬件和系统软件的错误**产生。

最后，代码应该复查，确信组织性标准和程序正确运行。(不懂)。**标准和程序错误**可能不会影响到程序的运行，但它们为系统测试修改时错误的产生铸造了温床。由于对于需求标准的理解错误，一个程序员可能使另一个程序员难于理解代码逻辑或者难于找到数据描述解决问题。

正交错误分类

分类和跟踪我们发现的错误类型是有益的，不是在代码中，而是在任何一个软件系统中。历史信息能够帮助我们预测我们代码中可能存在的错误类型（这对我们测试导向有帮助），多次发生的同一类型的错误能够警告我们是不是该重新考虑我们的设计甚至我们的需求。许多组织依靠错误类型的数量和分布进行错误统计模型建立，定期分析。例如,IBM 错误预防程序追踪和文档纪录每次发生错误的根源；这些信息用来帮助给那些寻找错误类型的

人提出建议，这些已经减少了在软件中错误的数量。

Chillarege 在 IBM 开发出一种方法能够追踪错误，叫做**正交错误分类**。此方法针对那些能够产生许多错误的开发程序共同地以图表的形式对其进行分类。因此，分类策划必须是与产品，机构无关的，在开发的所有领域都是可用的。表 8.1 列出了由 IBM 分类构成的错误类型。在分类过程中，开发者要不仅识别错误的类型，还有确定分类是错误的冗长还是委任（不懂）。**冗长的错误**是由一些关键方面的代码遗漏所导致的；例如，变量没有初始化导致的错误。**委任的错误**是由代码不正确导致的；例如，变量初始化为错误的值。

表 8。1 IBM 正交监测分类

错误类型	意义
功能	影响性能，综级用户界面，产品接口，与硬件结构的接口或全局数据结构错误
接口	组件间通过调用，宏，控制阻塞和参数列表交互的错误
检查	数据和值在应用前验证失败产生的逻辑错误
分配	数据结构或代码块初始化错误
时间/连载	涉及时间分配和实时资源的错误
构造/打包/合并	存在于程序库，管理变化和版本控制出现的问题
文档	影响发布和维护注释的错误
算法	涉及算法效率和正确性或没有设计的数据结构

工具栏 8。1 惠普的错误分类

格雷迪(1997) 描写惠普的错误分类的方法。在 1986 年，惠普的软件度量学委员会鉴定在跟踪错误的几种分类。计划发展成现实，图 8.1 种详细描述。 那些开发者使用这个模型，即通过为发现的每个错误选择 3 个描述符： 错误(即错误在一个产品内注入) 的起源，和方式，错误类型(即，信息丢失，不清楚，错误，改变，或者可能更好的方法)。

每个惠普划分分别跟踪它的错误，并且做出如图 8。2 的摘要统计饼形图上报有关的情况。不同的划分经常有非常不同的错误轮廓，并且轮廓的本质帮助开发者想出需求，设计，代码，和测试活动，处理分类通常分辨出的特别错误。总的效果是降低错误的数量。

正交错误分类的一个关键特征是它的**正交性**。就是说，每一个分类元素仅属于一个分类。换句话说，我们希望在系统中清楚地查出每一个错误。因此，每个分类中错误数量的摘要信息是有实在意义的。如果一个错误同时属于多个分类将失去分类尺度的意义。同时，分类必须简洁明了，任何两个开发者可能对于一个特殊的错误分到同一个分类中。

错误分类能够通过告知我们在开发过程中发现的错误类型提高整个开发过程。例如，对于每一个在搭建系统时用到的错误鉴定或测试技术，我们能够构造一个错误类型的轮廓。可能不同的方法产生不同的轮廓。然后我们就能根据我们在系统中预料的错误类型构造错误预防错误监测的策略，然后动手把它们根除掉。Chillarege 演示了 IBM 公司采用此概念，在设计复查中的错误轮廓不同于代码检测。

8.2 测试论坛（不准）

在我们自信系统正常工作发布给用户时，许多类型的测试工作需要做。一些测试依靠

已经测试的：组件，组件组，子系统，整个系统。另一些测试依靠我们想了解的：系统是否按照设计，需求，用户期望运作？让我们考虑如下一些论点

测试机构

在开发一个大的系统时，测试通常涉及许多步骤。首先，每个程序组件要孤立与系统其他组件进行自我测试。已知的这样的测试有**模块测试**，**组件测试**或**单元测试**，这种测试校验组件在设计中期望的输入条件下功能正常。单元测试尽可能在一个受控的环境下进行。因此测试小组输入一些预定的数据给组件测试，观察输出情况和产生的数据。另外，测试小组要检查内部数据结构，输入和输出数据的逻辑和边缘条件。

当组件的集合进行了单元测试时，下一步是确保组件间的接口正确定义和处理。**集成测试**是一个确保系统各组件按照系统描述和设计需求统一工作的过程。

一旦我们确信组件按照设计的要求工作，我们测试系统保证它实现预期的功能。**功能测试**评估一个系统是否能够作为一个完整的系统实现需求分析中描述的功能。结果是一个功能系统。

回忆一下需求通过两种方式纪录：一，用户的术语，二，开发者用到的一系列软件硬件需求。**功能测试**按照需求描述的功能比较系统功能。然后，性能测试按照其余的软件和硬件需求比较系统。当在用户实际工作环境测试成功执行，就产生了一个**正确的系统**。

当性能测试完毕，我们开发者确信系统根据我们对系统的了解工作。下一步是与客户协商，确认系统按照用户期望在工作。我们加入到用户中进行**接受性测试**，核对系统按照用户需求描述工作。接受性测试完毕，接受的系统安装到将被用到的环境下，最后的**安装测试**确保系统仍能按照工作。

图 8.3 演示了这些测试步骤的关系。不管被测试的系统有多大，每一步描述的测试类型都是必须确保正确的工作。在本章中，我们主要注重单元和集成测试，就是组件单独测试，然后组合在一个更大的工作系统进行测试。在第 9 章，我们会关注测试过程中余下的步骤，全称**系统测试**。在随后的几步中，系统看作一个整体，作为一个整体进行测试，而不是分离的部分。

对待测试的态度

程序员新手不习惯于把测试看待成发现的过程。作为一名学生，你根据老师给你的规定编写程序。设计完程序，你编写代码，编译代码，确定是否存在语法错误。当为你的程序评估等级时，你通常给老师一个程序列表，测试输入数据，经过处理的输出数据。代码，输入，输出证明了你的程序正确运行。你通常选择你的输入，劝说老师代码实现了课堂作业要求的功能。

你可能仅考虑了你的程序是解决问题的一种方法；你并没有考虑问题的本身。如果这样，你的测试数据仅选择了某些用例中正确的结果，忽略了错误的存在。在这种方式下写出和演示的程序仅代表你的编程技巧。因此，心理上你可能认为对于你程序的批评是对你的能力的批评。测试你的程序正确工作是你对老师展示能力的一种方法。

然而，当你为客户开发一个系统时，他们不关心系统在特定条件下的正确工作。相反，他们关心的是系统能够在任何条件下工作。所以作为一个开发者你的目标是尽可能的消除过多的错误，不管错误在那里发生，由谁产生。惨痛的教训不允许在开发过程中发现错误。因此，许多软件工程师采纳了一种叫做**忘我编程**的态度，这种态度要求程序被看作大系统的组件，是谁编写的并不重要。当错误发现或失败产生，忘我的开发小组关心的是纠正错误，不是给那一位开发者带来耻辱。

谁进行测试？

就是当我们采用忘我的方法开发系统，有时仍难于排除个人情感进行测试过程。因此，我们常采用一个独立的测试小组测试系统。这样，我们排除了个人出错责任的冲突，尽可能发现更多的错误。

另外，其他一下因素也证明了独立小组的正确性。首先，我可能不经意间在解释设计，确定程序逻辑，书写描述文档，实现算法时掩盖了错误。很明显，我们不会认为代码不符合规范就提交进行测试。但我们需要贴近代码客观的挑出一些微小的错误。

而且，一个独立的小组能够参与到开发过程中的复查阶段。这个小组可以是需求设计复查的一部分，可以单独地测试代码组件，可以测试完整的演示给用户的系统。这样，测试能够与编码并行运作，测试小组能够测试完整的组件，并且开始合并组件。同时程序员继续编写其他组件。

测试对象的见解

在我们单元测试之前，让我们考虑我们测试后面的道理。我们测试一个组件，一组组件，子系统，系统，你对测试对象的见解（在这里是一个组件，一组组件，子系统，系统）影响到测试进行的发展。如果你把测试对象从外边看成的**闭盒**或**黑盒**，就是说内部细节不考虑，测试输入到闭盒指出产生的输出。在这种情况下，测试的目的是保证每提交一个输入，输出匹配期望的结果。

这种测试有利也有弊。明显的优点是闭盒不受内部结构和测试对象逻辑的约束。然而，这种方式不能进行全面的测试。例如，假想一个简单组件接受 3 个数字 a,b,c 产生方程的两个根或消息“没有解“。不可能提交所有可能的三个一组数字（a,b,c）测试组件。在这种情况下，测试小组可能选择具有代表性的测试数据来说明所有组合能够正确处理。例如测试数据可能选择对于每个 a，b，c 正数，负数，零的组合：27 种可能。如果我们知道如何解决二次方程，我们宁愿选择数值确保判别式在三个分类中的每一个：正数，负数，零（在这种情况下，我们猜测组件的实现）。然而，如果在每一类的测试都没有出现错误，我们不能保证组件就是不存在错误的。组件可能仍由于 round-off 错误或不匹配数据类型而失败。

对于一些测试对象来说，测试小组不可能运行测试用例覆盖所有可能的情况。回忆一下第 7 章的组件，接受调整后的总收入，计算联盟收入税的数量作为输出。我们可能有一个每一个给定的输入相对应的税收表，但我们不必了解税收是如何算出来的。计算税收的算法依赖于税收分类，分类限制和相关百分比是组件内部运算的一部分。把此组件看成一个闭盒，我们不能选择有代表性的测试用例，因为我们不了解其内部的运算机制。

为了克服这些困难，我们把测试用例看成**开盒**（有时叫**白盒**或 **clear box**）；于是我们能够用测试对象的结构进行不同方式的测试。例如，我们能够设计执行组件所有语句或所有控制路径的测试用例，以确保测试对象正确工作。然而，正如我们在本章以后所看到的，采用此方法并不切实际。

例如，一个带有许多分支许多循环的组件有许多路径需要测试。就是一个简单的逻辑结构，一个具有迭代递归的组件也难于全面测试。假想一个组件结构是循环 nm 次，如图 8.4 所示。如果 n 和 m 每个都等于 100000，测试用例就需要循环 100 亿次遍历所有逻辑路径。我们可以用一个测试策略仅循环几次，用一小部分相关用例检测所有的可能。在这个例子中，我们选择变量 I 的值小于 n，等于 n，大于 n，同样，我们选择 J 小于 m,大于 m，等于 m，把 J 的组合和 I 的组合联系起来。一般说来，这种策略基于数据，结构，函数，其他一些标准。

当确定如何测试时，我们不必唯一地要么选择白盒要么选择黑盒。我们可以考虑黑盒测试作为一个测试的极端，而白盒测试作为另一种极端。任何一钟测试哲学都在两者间寻找平衡。工具条 8.2 告诉我们如何在不同方面运用测试方法联合几项指数生成一个联合体（不懂）。一般说来，测试方法的选择依赖于许多因素，包括

可能的逻辑路径数量

输入数据的种类

涉及的计算数量

算法的复杂度

工具栏 8.2 盒子结构

盒子结构方法与信息系统紧密联系在一起，扩展对开盒和闭盒观点的概念(Mills, Linger, and Hevner 1987:Mills 1988)。这技术从黑盒的观点开始，逐步扩展到状态盒子然后提炼到白盒。

一个对象的黑盒观点是在全部可能的情形里的它的外表行为的描述。对象(组件，子系统或者完全系统) 描述了接受的刺激和刺激历史——也就是说，它过去对刺激的反应的记录。 我们能把每个反应描述为以下转变，

(刺激，刺激历史-)反应）

接着，状态盒描述源自黑盒描述，只是增加状态信息。每种转变写为

（刺激，旧状态-）反应，新状态）

最后，白盒描述增加实现状态盒的程序；即，它描述刺激和老状态怎样转换到反应和新状态：

（刺激，旧状态-）反应，新状态）通过程序

程序依据顺序，交替，迭代和并发编写。从黑盒到白盒的过程不仅对测试过程，而且对设计组件过程有帮助，帮助把高级描述变成更低的水平，更仔细设计描写。

8.3 单元测试

如果我们的目的是在组件中找到错误，我们怎样开始？这个过程类似于你测试一个明确到类的程序。首先，你通读代码，试着挑出算法，数据和语法错误。你可能甚至把你的代码和需求陈述，设计进行比较以便保证你考虑了所有相关的情况。接着，你编译代码，清除剩余的语法错误。最后，你开发用例，显示输入正确的转换成期待的输出。单元测试也遵循这些步骤，我们每次检测它们一次。

检查代码

因为设计描述帮助你编码和文档纪录每个程序组件，你的程序反映了你的设计。文档用文字和图片解释了程序在代码需要做的。因此，邀请一些客观的专家复查你的代码和文档，找出误解，不一致或其他错误是有益的。这个过程，叫做代码复查，与前几章讨论的需求复查设计复查类似。一个小组由你程序员，3 至 4 个技术专家组成；这个小组有组织地研究程序寻找错误。技术专家可以是其他的程序员，设计员，技术作家或工程管理员。尽管设计复查小组包括用户代表，但代码复查小组不包括任何用户组织人员。客户表述需求，赞同提议的设计；他们仅在我们根据他们的描述演示系统时才关心实现。

代码预排。有两种类型的代码复查：预排和检查。在预排中，你要提交你的代码和你的合作文档给复查小组，小组注释上他们的改正。在预排中，你领导和控制讨论。气氛是非正式的，注意力的中心集中在代码，不是编写者。虽然监督人员会在场，预排不会影响到你的

性能评估，专心测试，寻找错误，但不必修正。

代码检查。代码检查，最早由 Fagan 在 IBM 首先介绍，它类似于预排但更正式。在代码检查中，复查小组认真核对预先标注的代码和文档。例如，小组可能检查数据类型和数据结构的定义和应用，确认它们与设计和系统标准和程序是否一致。小组能够复查算法和计算的正确性和有效性。注释与代码进行比较，保证它们的准确和完毕。同样，组件间的接口也要求正确性测试。小组甚至评价代码在内存利用上，运算速度上的性能特征，以满足性能需求上的要求。

检查代码通常涉及几步。首先，小组全体开会商议对于代码的总体看法和检查目的的描述。然后，小组成员各自为第二次团队会议作准备。每个检察员研究代码和相关文档，指出发现的错误。最后，在团队会议中，小组成员汇报它们所发现的，记录在探讨各自发现中找得多余错误。有时个人发现的错误被看成“false positive”：元素看上去是错误实际对于团队来说不是真正的错误。

检查小组成员被选做用来检查代码的目的，有时一个特殊的小组成员身兼数职。例如，如果为了检查接口的正确性，小组需要包括一个接口设计者。因为目的集中在检查，小组主持人，不是程序员，是会议的领导者，对于一些关键问题进行回答。和预排一样，检查批评的是代码，不是编写代码的人。结果不反映到成绩评估上。

成功的代码复查。你可能会因为小组的检查代码而感到不安。然而，复查在检测错误上演示了超凡的成功，经常作为强制的或最好的实习加入到组织列表中。记住开发过程中错误越早被找出，纠正起来越容易越省力。当困难的根源还不是很清楚的时候，在组件级发现毛病要好于后来的测试环节。实际上，由此 Gilb 建议早期检查开发成果，诸如需求描述和设计而不是代码。

一些研究员已经发现复查鉴别错误的广度。Fagan 实验得出 67%的系统错误最终在单元测试前通过检查发现。在 Fagan 的研究中，第二组程序员写相同的程序，采用非正式的预排而不是检查。检查的组比预排的组在第一个 7 个月中少 38%的错误。在第二个 Fagan 实验中，所有在系统开发中发现的错误，82%是在设计和编码检查中发现的。早期的错误发现为开发者赢得了大量的时间。其余的调查者报告也证明了他们采用了检查方法。例如，Ackerman,Buchwald and Lewski 指出在一个 6000 行的商业软件中全部错误 93%是由检查发现的。

Jones 广泛的研究程序员的生产力，包括错误的查找和修正的本性。查阅了历史上一千万行的代码，他发现代码检查纠除了 85%多的错误。Jones 研究的其他方法从未如此成功，实际上，没有一个能够除去一半以上的错误。Jones 最近更多的调查建议典型的准备时间和会议时间，表 8.2 显示。

表 8。2　典型的准备时间和会议时间（1991）

开发产品	准备时间	会议时间
需求文档	每小时 25 页	每小时 12 页
功能描述	每小时 45 页	每小时 15 页
逻辑描述	每小时 50 页	每小时 20 页
源代码	每小时 150 行代码	每小时 75 行代码
用户手册	每小时 35 页	每小时 20 页

工具栏 8。3 检查的最好小组大小

韦勒(1993) 在公牛信息系统 3 年来的检查检查数据。 来自几乎 7，000 个检查会议的尺度包括大约 11557 个错误信息和 14677 页的设计文献。他发现，一个 3 人低准备率测试

小组和一个 4 人高准备率小组工作效率一样；他提出准备率，并非小组大小，确定测试效率。他也发现小组的有效性和效率取决于他们对产品的熟悉： 越熟悉，越好。

另一方面，韦勒发现好代码检查结果能建立错误的信任。在一项 12000 行 C 语言的工程上，需求和设计没被复查； 检查从代码开始。 但是需求继续在单元和集成测试期间逐步形成，此期间代码量翻番。比较代码检查数据和测试数据，韦勒发现代码检查鉴定主要编码或者低水平设计错误，但是测试主要发现需求和结构的错误。因此，代码检查不在系统里处理源代码的变化性，它的结果没代表真实的系统质量。

表 8。3 发现活动中发现的错误（Jones 1991）

发现活动	每千行代码发现的错误
需求复查	2.5
设计复查	5.0
代码检查	10.0
集成测试	3.0
接受性测试	2.0

Grady 在惠普解释到，为代码检查做计划一般需要 2 个小时，接着 30 分钟的小组会议。然后个人准备 2 个小时发现错误和 90 分钟汇报错误。小组花费 30 分钟集体讨论错误的发现并推荐采取的措施。当错误修正后，代码检查会议的主持人在花费半个小时书写和发布摘要文档。工具栏 8.3 描述了 Bull Information Systems 软件开发者在维持效果条件下探索减少资源需要的方法。

Jones 总结了他的工程信息的数据，画了一个不同的图，比较复查，检查和其他发现方法的不同。因为产品在尺寸上变化广泛，表 8.3 显示了在发布的产品中错误相对于千行以上的代码数量发现率。此表清楚演示了代码检查比其他的技术发现很多的错误。然而，研究员继续研究那些形式的行为比其它的更容易发现不同类的错误。例如，代码检查善于发现代码错误，但是原型系统更容易鉴别需求的错误。

Fagan 在 IBM 发布检查代码的方针后，许多其它的组织机构，包括惠普，ITT 和 AT&T 都采纳了代码检测作为推荐或标准实施。代码检测的成功应用继续出现在文献中出现，一些在网站上有参考文献。

证明代码正确

假想你已经编完你的组件，自己检查完毕，小组复查完毕。下一步测试是，以更加结构化的方式详细审查确保它的正确性。为了单元测试，一个程序是正确的，当它正确实现设计中说明的功能和数据和它与其它组件正确接口。

一种研究程序的正确性方式是把代码看成语句的逻辑流。如果我们能够用正式的逻辑系统（诸如一系列关于数据的语句和暗示）复写程序，我们就能测试新的语句的正确性。我们在设计的方面解释正确性，我们希望我们的表达遵循数学逻辑的规则。例如，我们用一系列断言和定理表达程序，我们证明了定理的正确就证明了代码的正确。

公式证明技巧。让我们看看公式证明是怎么工作的。我们按照一系列步骤转换代码为逻辑副本：

1．首先，我们写断言来描述组件的输入和输出条件。这些语句由逻辑变量组合（每个要么是 true 要么是 false），由表 8.4 列出的逻辑连接符号连接。

表 8.4 逻辑连接

连接	例子	意义
联合	X&Y	X and Y

分离	X v Y	X or Y
取反	-X	Not X
蕴含式	X->Y	If x then y
相等	X=y	X equals y
通用	V x P(x)	对于所有 x,条件 P(x)是真
存在	E x P(x)	至少有一个 x，P(x)是真

例如，假想一个组件接受一个长度为 N 数组 T 作为输入。作为输出，组件制造了一个等同的数组 T'，数组 T'由 T 的元素升序排列。我们可以写输入条件作为一个断言：

同样，我们写输出作为一个断言

2．接着，我们画一个流程图描述组件的逻辑流程。在图中，我们指示转换发生在那一点。

图 8.5 演示了一个用于起泡法升序排序的组件的流程图。在图中，两点高亮显示转换的发生。用一个括号标记的点用如下断言描述：

同样，有两个括号标记的点断言写成这样：

3．从断言中，我们可以证明一系列定理的证明。从第一个断言开始，有一个转换到另一个转换，我们完成推理，保证如果一个正确，那么下一个也正确。换句话说，如果第一个断言 A1 和第一个转换点 A2，于是我们的定理是 A1->A2

如果 A3 是下一个转换点，下一个定理是 A2->A3

按照这种方式，我们阐述定理 Ai->Aj

其中 Ai,Aj 是流程图中相邻的转换。最后的定理指出最后转换点的条件真暗示了输出断言的真 Ak->Aend

交替一下，我们在流程图中倒溯，由 Aend 开始，寻找前一个转换点。我们首先证明 Ak->Aend 然后证明 Aj->Aj+1 以此类推，直到我们证明 A1->A2

每种方法的结果是相同的。

接着，我们在流程图中定位循环语句，制定每一个 if then 断言。

到了这一步，我们已经鉴别所有可能的断言。为了证明程序的正确性，我们定位所有从 A1 开始到 Aend 结束的路径。通过跟随每一条路径，我们同时跟随了正确输入产生正确输出的条件的每条路。

鉴别了所有的路径，我们必须严格证明输入断言根据路径转换暗示了输出断言，从而校验了每个分支的真实性。

最后，我们证明程序结束。

程序证明的优势和劣势

通过按照前面讲述的方法构造一个自动的或手动的证明，我们能够发现代码算法错误。另外，证明方法让我们对于程序有个正式的了解，因为我们检查了潜在的逻辑结构。经常用此方法能使我们在规定数据，数据结构，算法规则上更严格更精确。

然而，严格是需要代价的。许多工作牵扯到提出和完成程序的证明。例如，起泡法排序组件的代码要比逻辑描述证明少。在很多情况下，更多的时间花在程序的证明上，而不是程序编写上。而且，大的更复杂的组件涉及巨大的逻辑流程图，许多转换和许多路径的确认。例如，非数字化的程序要比数字化的程序难于用逻辑表示。并行计算难于驾驭，复杂的数据结构导致复杂的转换语句。

注意到证明方法是根据逻辑规则基于如何使输入断言转换成输出断言。在逻辑认识上，证明程序的正确性并不意味着软件就没有错误。事实上，这种方法不能挑出在设计，与其它组件接口，解释描述，程序语言的语法语义或文档上的错误。

最后，我们必须承认不是所有证明都是正确的。历史上许多的数学证明已经被接受了许多年后来显示出谬误。对于一个特殊的证明进行如此争论是得不偿失的。

其它证明方法

逻辑证明方法忽略编程语言结构和语法的。在某种意义上说，这种方法证明了组件设计是正确的，但对于实现不是必需的。其余方法将编程语言考虑进去。

一个这样的方法，**符号执行**，用符号而不是数据变量模拟代码的执行。测试程序看成由输入数据和条件决定的输入状态。每行代码的执行，该方法都检测状态的变化。每个状态的变化都记录下来。程序的执行就是一系列状态的变化。因此，程序的每个逻辑路径都对应着一系列有序的状态变化。每条路径的最终状态就是输出状态。如果每个可能的输入状态都生成正确的输出状态，程序就是正确的。

我们用一个例子看看符号执行是怎么工作的。假想我们测试这几行代码：

```
a=b+c
if(a>d) taskx();//perform taskx
else tasky();//perform tasky
```

符号执行工具会指出条件 a>d 要么是 true 要么是 false。虽然常规的代码执行需要给 a,d 赋予特定的值，但符号执行工具记录了两种可行的状态 a>d 是 false a<d 是 true。不用测试过多的 a，d 的可能值，符号执行仅考虑两种情况。由此，大量的数据被分为等价类，代码仅考虑关于每一类数据的反应。由于用符号表示，仅考虑等价类的数据，在程序证明中大大斤少了用例的数量。

然而，这种方法在逻辑定理证明中同样存在劣势。证明定理要比编写代码本身耗时，正确性的证明不能保证不出现错误。而且这种方法依赖于贯穿程序始终细心跟踪变换条件。虽然在某种程度上这种方法可以实现自动化，但过多的过复杂的代码仍需要耗费时间检测许多状态和路径的。对于自动符号执行工具来说难于对循环进行跟踪执行。另外，当下脚标和指针在代码中出现后，分等价类将变得更难。

自动定理证明

一些软件工程师试着开发工具进行程序的自动证明，这些工具读入输入，诸如

输入数据和条件

输出数据和条件

检测组件的代码行数

自动工具输出要么是组件的正确性，要么是一个展示一些数据说明组件没有正确转换成输出的反例。自动定理证明器包括组件编写语言信息，因此，语法，语义的规则也是必需的。跟随编程的步骤，定理证明器用一些方式鉴别路径。如果常规推理和演绎用起来略显笨拙，有时就采用一些启发式证明。

这样的定理证明软件难于开发的。例如，工具软件必须能够检验一元和二元（例如，加法，减法，求反）操作的正确应用，也包括相等和不相等比较。更多的复杂规则例如交换性，分配性，结合性都要考虑进去。用一些基本原理推导出定理，而仅采用程序语言是困难的。

假想这些困难能够克服。用试验和报错构造一个定理对于任何一个微不足道的组件是很耗时的。因此，一些人机交互期望能够引导定理证明器。开发一个专家系统，运用经常用到的方法，一个交互式的定理证明器可以在用户的帮助下选择转换点和跟踪路径。这样，定理证明器没有真正地证明，相反，它是在用户的驱使下检查证明。基于符号执行的工具目前也

只能在小的程序评估代码，根本就没有通用的，语言无关的，自动的符号执行系统。

设想，随着机器运行速度的加快，编程语言能够处理更复杂的问题，理想中的定理证明器能否制造出来？理想中的定理证明器会读入任何程序生成输出要么是程序的正确性要么是出错的位置。定理证明器还要决定是否有任意的代码来处理任意输入数据。不幸的是，这种定理证明器永远不会制造出来。Pfleeger and Straight 指出一个程序的构造等同于一个中止问题的图灵机。中止的问题不可解决，也就是说，不仅没有方法解决问题，而且不可能找到解决问题的方法。我们可以使我们的定理证明器仅应用于解决没有分支的代码，但此限制导致了工具软件仅能应用与程序的一个子集。因此，虽然众望所归，自动程序证明器大概是个幻想吧。

测试程序组件

证明程序正确是软件工程师期望的目的；因此，许多相关的研究用来开发方法和自动工具上。然而，在最近的未来，开发小组更可能去测试软件而不是证明程序。

测试正确证明

在证明一个程序是正确的，测试小组或者程序员仅考虑代码和输入输出条件。程序看成是数据的分类和设计描述中的条件。因此，证明可能不涉及执行代码而是了解程序内部的运行机制。

然而，客户带有另一种观点。为了演示给他们一个正确工作的程序，我们必须演示代码在程序以外是如何工作的。在此情况下，测试变成了一系列在给定条件下程序如何反应的实验。然而程序证明告诉我们程序在设计和需求描述的假定环境下如何工作，测试告诉我们在实际操作的环境下程序的运行情况。

选择测试用例

测试一个组件，我们选择输入数据和条件，允许组件操作数据，得出输出。我们选择输入，因此输出证明了代码的一些行为。**测试点**或**测试用例**是用来测试程序的输入数据特殊选择。测试是无数测试用例的集合。我们怎样选择测试用例定义测试才能向我们自己和用户保证程序不仅在测试用例，而且在所有输入的情况下都正确运行。

我们从确定测试对象开始。然后，我们选择测试用例，定义测试设计来迎合特定的对象。一个对象可能证明了所有的语句工作正常。对象决定我们怎样为选择测试用例而分类输入。

我们根据测试的对象，把代码看成要么白盒要么黑盒。在黑盒中，我们提供了所有可能的输入，根据需求与预期比较输出。然而，如果代码看成白盒，我们用细致的测试策略检测代码的内部逻辑。

回想一下计算二次方程根的那个组件。如果测试对象证明代码功能正确，我们会选择系数 abc 代表正数负数零的一些组合的测试用例。或者我们选择一些相对大小的系数组合：

a 大于 b 大于 c

b 大于 c 大于 a

c 大于 b 大于 a

等等。然而，如果我们确认代码内部的运作，我们知道程序逻辑依赖于判别式的值。然后，我们选择代表判别式为正数，负数，零的测试用例。我们也可以包括一些非数字化的测试用例。例如，我们可以输入字母 F 作为系数，确定系统对于非数字的反应。包括 3 种数字的用例，我们有 4 种相互独立的测试输入。

在这种方式下，我们用测试用例将输入分成等价类。也就是说，数据的分类要满足这些规则：

1。每一个可能的输入属于一个分类。也就是说，分类覆盖整个输入数据。

2。没有那个输入数据属于两个以上的类。也就是说，分类是不相交的。

3。如果输入的一个特定分类中的成员导致了执行代码的错误，同样的错误在该分类中的其他成员同样发生。也就是说，每一个分类中的元素代表整个分类。

第 3 条约束不容易或者不可能碰到。我们放宽约束条件，如果一个分类中的元素导致了错误，该分类中的其他元素导致错误的概率也很高。

闭盒测试带来的烦恼是不容易确定所选择的测试用例会揭露特殊的错误。在另一方面，开盒测试不得不把注意力更多放在代码的内部运行上。我们可能在程序的做了什么上就结束了测试，而不是程序应该做什么上。

我们可以联合开盒和闭盒测试运行测试数据。首先，把程序考虑成一个闭盒，我们用程序的外部规定运行初始的测试用例。这些用例应该合并期望的输入数据，输入输出的边缘条件，除了一些不合法的数据。例如，如果程序希望输入一个正数，我们可能包括如下一些用例：

一个很大的整型数

一个正整型

一个正的负点数

一个比 0 大比 1 小的数

零

一个负数

一个非数字的字符

有些数据故意选择的不恰当。我们可以测试程序驾驭不正确数据的完整性。

下一步，通过研究程序的内部结构，我们添加额外的测试用例。例如，我们增加测试用例，测试所有的分支，运行尽可能多的路径。如果牵扯到循环，我们包含了让循环运行一次，多次，或不运行的测试用例。我们也要检查算法的实现。例如，如果程序进行三角计算，我们包含了一些三角函数极值的如 0，90，270，360 的测试。我们也可以输入一些导致分母为零的用例。

有时一个系统需要记住前一个用例的条件，所以测试用例的顺序也是必须的。例如，当系统实现无限状态自动机时，代码必须回忆前一个系统状态；前一个状态加上现在的状态决定下一个状态。同样，实时系统是中断驱动的，测试需要一系列的用例，而不是单单的一个。

测试完全

为了进行测试，我们决定如何证明以一种令人信服的方式让测试用例遍历所有的行为。让我们看看我们拥有那些选择。

为了完全测试代码，我们选择测试用例，是基于代码操纵的数据运用以下至少一种方法来选择的。

语句测试：在某些测试中组件中的每个语句至少执行一次。

分支测试：在某些测试中代码中的确定点，每个分支至少选择一次。

路径测试：在某些测试中代码中每一个不同的路径至少执行一次。

定义用途路径测试：在某些测试中每个变量每个用途的定义的路径进行测试。

所有用途测试：测试包括此定义下从定义到应用的每一条路径。

所有谓词应用/一些计算用途测试：对于每个变量和变量的定义，测试包括定义到谓词应用的至少一条路径；如果定义不涵盖描述，为了涵盖所有定义，也要包括一些计算应用。

所有计算应用/一些谓词应用测试：对于每个变量和变量定义，测试包括从定义到计算应用至少一条路径；如果定义不能涵盖描述，为了涵盖所有定义，也要包括一些谓词应用。

也有一些其它的类似的测试，例如：全定义，全谓词应用，全计算应用。Beizer 描述了这些测试策略相对力度。如图 8.6。例如，测试所有路径要强于测试所有路径从定义到应用。一般说来，策略越强，需要越多的测试用例；我们必须考虑在测试可用资源上和策略力

度上交替使用。

我们可能采用策略比随机测试效果更好。例如，Ntafos 在 7 个已知错误数学程序比较了随机测试，分支测试和全用途测试。他发现随机测试发现 79。5%的错误。分支测试发现 85。5%，全用途测试发现 90%。

为了说明策略影响测试用例的数量，考虑图 8.7 的例子，图中是要测试的组件逻辑流程图。每一条语句由菱形或矩形表示，都标好数字。语句测试需要测试用例从 1 到 7 执行语句。选择 X 大于 K 产生一个正数的结果，我们按照 1234567 的顺序执行语句，因此一个测试用例足够了。

对于分支测试，我们必须鉴别所有判别点，图 8.7 用菱形表示。有两个判别：一个关于 X 和 K 的关系，另一个关于结果是否为正数。两个测试用例执行路径 1234567 和 124561, 经过每条路径至少一次。第一条路径在第一判别点为 yes，第二条路径为 no。同样，第一条路径在第二判别点用 yes，第二条路径用 no

如果我们测试程序中所有的可能路径，我们需要更多的测试用例。路径 1234567 1234561 124567 124561 覆盖了所有可能情况：两个判别点在每个分支两种选择。

在我们这个例子中，语句测试比分支测试需要少的测试用例，分支测试比路径测试需要少的测试用例。一般说来，都是这样的情况。而且，一个程序越复杂，测试用例需要的路径越多。习题 4 研究了判别点的结构和顺序与代码路径的关系。

在单元测试中也有许多测试策略。例如安全应用经常测试每一种可能的事务进行到底，这种策略叫**事务流测试**。希望了解全面的测试策略，参考 Beizer.

比较方法

Jones 已经比较了一种类型的错误发掘方法，以便确定哪一种更可能发现同一类的错误。表 8.5 列出了开发过程中产生错误的调查结果。例如，如果一个存在于代码中的错误，但是由需求分析导致的错误，它被列在了需求这一列。

表 8.5 由错误原因计算出的错误发现百分比（Jones1991）

发现方法	需求	设计	编码	文档
原型	40	35	35	15
需求复查	40	15	0	5
设计复查	15	55	0	15
代码检查	20	40	65	25
单元测试	1	5	20	0

Jones 也研究了那一类除错技术在某一类错误上应用地更好。表 8.6 显示了复查和检查对于设计和代码错误更有效，但是原型是在鉴别需求问题上是最好的。工具栏 8.4 说明了为什么采用多样的方法进行纠错。

表 8.6 错误发现方法有效性（Jones 1991）

	需求错误	设计错误	编码错误	文档错误
复查	一般	极好	极好	好
原型	好	一般	一般	不适用
测试	不好	不好	好	一般
正确性证明	不好	不好	不好	不好

工具栏 8。4CONTEL IPC 错误发现效率

奥尔森(1993) 描述一个 184000 行代码，使用 C，对象 C ，汇编系统的开发，为一个金融团体提供办公自动化的程序。他跟踪在各种各样测试活动期间发现的错误，找出了方法的不同： 17.3% 的错误在对系统设计的检查期间发现， 19.1% 在组件设计部分期间发现，

15.1%在代码检查期间， 29.4% 在集成测试期间，16.6%在系统和回归测试期间。在系统在一定领域内发布之后， 仅仅 0.1% 的错误被发现。 因此，奥尔森的工作显示在开发期间使用不同的技术查出不同种类的错误重要性；依赖单个方法捕捉错误是不够。

8.4 集成测试

当我们对于单独的组件正常工作迎合我们的目的感到满意时，我们联合组件到一个工作系统中去。集成是有计划的和平等的，以至于当出现失败时，我们知道是什么导致的。另外，组件的测试顺序影响到我们测试用例和工具的选择。对于大的系统，一些组件可能还在编码中，其它的还在单元测试中，仍有些作为集合一起测试。组件融合到工作系统中的原因和方式说明了我们的测试策略。这个策略影响到的不仅是定时和代码顺序的集成，还是处于测试成本和全面的考虑。

系统再一次地被看成层次结构的组件，每个组件属于设计中的一层。我们自顶向下测试，自底向上，或者联合这两种方法。

自底向上集成

一个常用的方法是合并组件测试一个大的系统，叫做**自底向上测试**。用此方法，系统层次的最底层组件首先单独测试。然后，调用前一些已经测试组件的组件进行测试。这种方法重复进行，直到所有的组件都已经进行测试。在面向对象设计和由许多独立的重用组件组合的大系统，对于许多底层的组件是通用目的的用来被其他组件调用的工具程序，自底向上测试方法是有益的。

例如，考虑图 8.8 中的组件和层次结构。为了自底向上测试系统，我们首先测试最底层 EF 和 G。因为我们没有调用最底层组件的组件，我们编写特殊代码帮助集成。**组件驱动**是用来调用一个特殊的组件，传递测试用例的常规方法。由于它不需要复杂的计算，驱动不能编写。然而，需要细心的是，保证驱动的接口与测试组件的定义要吻合。有时测试数据是自动配备的，因为某个特定目的语言中使数据的定义变得便利。

在我们这个例子里，我们需要为 EF 和 G 构造组件驱动。当我们对于三个组件正确工作满意时，我们转移到更高的层次。不同于最底层的组件，下一层的组件不能单独测试。相反，他们需要和别调用的组件（已经测试过了）一起测试。在这个例子中，我们要一起测试 BE 和 F。如果出现错误，由于 E 和 F 各自正确的工作，我们知道错误是要么是由 B 产生，要么是 B 和 E 接口或者 B 和 F 的接口。如果我们测试 BE 和 F，而没有单独测试 E 和 F，我们不会这么容易就孤立出错误的根源。

同样，我们一起测试 D 和 G。因为 C 没有调用任何组件，我们单独测试它自己。最后，我们一起测试所有的组件。图 8.9 显示了测试的顺序和依赖。

经常有人抱怨，自底向上测试是对系统功能的分解，最上层的组件往往是最重要的，却被最后测试。最上层引导主要系统的活动，然而最底层的通常完成一些平庸的任务，例如输入输出或者重复的计算。最上层越通用，然而最底层越特定。因此，一些开发者感到先测试底层，对于主要错误的发现将会推迟到测试结束。而且，有时顶层的错误反映了设计上的错误。很明显，这些错误应该尽早在开发中纠正，而不是等到结束时。最后，顶层的组件通常控制或影响时间因素，当许多的系统运算依赖于时间因素时，自底向上很难测试一个系统。

在另一方面，自底向上测试对于面向对象程序最敏感。对象每次都是由已经测试过的对象或对象集合组合而成。消息从一个发到另一个，测试保证对象对消息做出正确反应。

自顶向下集成

许多开发者更喜欢**自顶向下方法**，此方法在很多地方于自底向上相反。最高层，通常作为控制的组件，自身测试。然后，所有被已测试组件调用的组件组合在一起作为一个大的单元测试。这种方法多次应用直到所有的组件都被组合。

一个正在测试的组件可能调用另一个还没有测试的组件，所以我们写一个**存根**，一种用来模拟遗漏组件活动的特殊目的程序。存根程序回应调用顺序，返回输出结果，让测试过程进行下去。例如，如果一个组件用来计算下一个可用的地址，但该组件并未测试，于是存根程序返回一个固定的地址，让测试进行下去。和组件驱动一样，存根程序不必复查和逻辑完整。

图 8.10 显示了自顶向下测试在我们的例子系统中如何工作的。只有顶层组件 A，在 B，C，D 的存根下，自己测试。一旦测试完毕，A 就与下一级组件组合，A,B,C,D 一起测试。此时的测试则需要 EF 和 G 组件的存根，最后，整个系统测试。

如果最底层的组件执行输入输出操作，该组件的存根程序要与替代的实际组件大致相同。在这种情况下，需要改变一下集成的顺序，以便使输入输出组件在测试顺序中在一些组合。

许多自顶向下设计和编码的优势同样适用于自顶向下测试。当特定的组件中的函数通过自顶向下设计而局部化后，从上到下的测试允许测试小组每次测试一个函数，这样可以跟随命令次序从最高层控制到最适当的组件。这样，测试用例可以定义到检测函数这一级别。而且，任何设计的错误或函数可行性的主要问题在测试的开始都能显露出来，而不是在最后。

同时也要注意，驱动程序不必用自顶向下的测试。在另一方面，编写存根是很难的，因为它们允许测试所有可能的条件。例如，假想组件 Z 是一个地图绘制系统，其中计算用到了由组件 Y 输出的精度和纬度。设计描述中指出了 Y 的输出经常在北半球。由于 Z 调用 Y，当 Z 进行自顶向下的部分测试时，Y 可能并没有编写代码。如果编写一个处理从 0 到 180 之间的数存根程序，让 Z 的测试继续下去，当设计改变成允许南半球定位时，存根程序也必须改变。也就是说，存根程序是测试中的一个重要部分，它的正确可能直接影响到测试的正确性。

自顶向下测试的劣势是可能需要大量的存根程序。在系统最底层包含一些通用的程序时，这种情况就会产生。一种避免这一矛盾的方法是轻微的改变策略。一个修改后的自顶向下测试方法先单独测试每一级组件，然后在合并组件，而不是一次就组合整个一层的组件。例如，在我们的例子系统中，采用修改后的自顶向下测试方法，先测试 A，然后测试 B,C,D，合并第一层和第二层这 4 个组件进行测试。然后是 EF 和 G 自身的测试。最后，整个系统联合起来测试。图 8.11 显示。

单独测试每一层的组件同样有困难。每个组件都需要存根程序和组件驱动，导致了更多的编码和更多错误的隐患。

大爆炸集成(Big Bang)

当所有的组件都独立测试通过，应该试着将组件整合在一起作为一个最终的系统进行初次运行，Myers 把这个称作**大爆炸测试**，图 8.12 显示了在我们的例子系统上是如何工作的。许多程序员在小的系统用大爆炸方法，但它并不适用于大的系统。实际上，由于大爆炸测试有诸多劣势，不是所有的系统都推荐用此方法。首先，它需要存根程序和组件驱动来测试组件的独立性。第二，因为所有的组件立刻合并，难于找出任何失败的根源。最后，接口的错误不能像其他错误那样容易辨别出来。

三明治集成

Myers 组合自顶向下策略和自底向上生成了**三明治测试方法**。系统被看成三个层次，就像三明治：目标层在中间，高于目标层，低于目标层。自顶向下方法用在上层，自底向上的方法用在下层。测试聚集在目标层，目标层的选择基于系统特征的和组件层次结构。例如，如果底层包含通用工具程序，目标层要高于该层，因为大多数组件要用到此工具。这种方法允许在测试的开始用自底向上测试检验工具程序的正确性。然后不必编写工具的存根程序，因为实际的工具已经可用了。图 8.13 描述了一个根据例子组件的层次结构可行的三明治集成顺序，目标层是中间级，组件 B,C,D。

三明治测试允许在测试过程中过早的集成测试。它也在自顶向下测试的优势上融入了自底向上在开始可以测试控制和工具的优点。然而，此方法在集成前没有彻底地单独测试各个组件。一个变化的修改后的三明治测试允许上层组件先测试后和其他组件合并。图 8.14 显示。

比较集成策略

选择一个集成策略不仅依靠系统的特征，还要依靠用户的期望。例如，用户先尽快看到一个能够运行的版本，所以我们采用集成进度表，这会在测试过程中生产一个基本的工作系统。在这种情况下，一些程序员编写代码，同时其余的在测试。以便测试和编码过程并行进行。Myers 构造了一个矩阵，如图 8.7 所示。根据一些系统属性和用户需求，比较了一些测试策略。工具栏 8.5 说明了在市场压力下，微软的策略。

表 8.7 集成测试策略比较(Myers 1997)

	自底向上	自顶向下	修正自顶向下	大爆炸	三明治	修正三明治
集成	早	早	早	晚	早	早
基本运行系统	晚	早	早	晚	早	早
需要组件驱动	需要	不需要	需要	需要	需要	需要
需要存根	不需要	需要	需要	需要	需要	需要
开始平行工作	中等	低	中等	高	中等	高
测试特定路径能力	容易	难	容易	容易	中等	容易
计划和控制顺序能力	容易	难	难	容易	难	难

工具栏 8。5 微软的构建

基于尽快有一个工作产品的需要(Cusumano 和 Selby 1995，1997)，微软公司的集成策略是市场的驱动。它使用很多小，平行小组(每组 3 到 8 个开发者) 实现一个"同步和稳定" 方法。 开发过程在设计，构造，测试组件中迭代，并且使用户深入测试的过程。一个产品的全部部分经常集成起来，确定什么工作什么不工作。

当开发者了解这个产品能做并且应该做的更多信息，微软公司的方法允许小组改变特征的说明。有时那些特征变化 30%或者更多。这个产品和工程基于特征分成许多部分，不同

的小组对不同的特征负责。然后，特征划分成非常关键，一般关键和最小关键，由此确定开发中的里程碑。特征小组在每日的构建产品和发现纠正错误的基础上，协调他们的工作。图8.15显示。因此，最重要的特征首先开发并集成，每个里程碑包括“缓冲时间”处理意料外的复杂因素或延迟。如果时间表一定要缩短，最不重要的特征从这个产品取消。

不管选择什么策略，每一个组件仅为测试合并一次。而且每个组件应该立即修改简化测试。存根程序和组件驱动是分离的新的程序，不是已有程序的临时修改。

8.5 测试面向对象系统

我们阐述的许多测试系统技术应用于所有类型的系统，包括面向对象系统。然而，你应该额外下些功夫，确保在你的测试技术上存有面向对象编程的特征。

测试代码

Rumbaugh 建议你在测试你的面向对象系统时考虑一下一些问题：

当你的代码需要一个唯一的值，存在一条生成唯一结果的路径么？

当有许多种可能的值，存在着一种方法选择唯一的结果么？

存在着不能驾驭的可用用例么？

接着，确保你检查对象和类的冗余和不足：遗漏对象，不需要的对象，遗漏或不需要的关联或不正确关联和属性的放置。Rumbaugh 提供了一些准则，帮助你在测试中鉴别这些条件。准则中指出，对象可能遗漏，假设：

你发现不均匀的关联或概括

你发现在一个类中全异的属性和操作

一个类行使两个或更多的任务

某个操作没有理想的目标对象

你发现两个有同样名字和作用的关联

一个类如果没有属性，没有操作，没有关联可能不是必需的。同样，一个关联如果带有多余的信息或没有操作路径也不是必需的。如果任务名在其所处的位置上过宽或过窄，那么可能有个关联放错了位置。或者你需要通过一个属性值来访问对象，你可能错误的定义了属性。对于每一种情况，Rumbaugh 建议你改变你的涉及，补救现状。

Smith 和 Robson 建议你的测试具有许多不同的层次：函数，类，群（相互合作对象的交互群体），整个系统。传统的测试方法对于函数有良好的应用，但许多方法没有考虑到测试类需要的对象状态。最起码，你应该开发测试，能够跟踪一个对象的状态到另一个状态的转换。在测试过程中，提防并行和同步的难题，确保相应的的事件是完整的一致的。

面向对象和传统测试的不同

Perry 和 Kaiser 谨慎地测试面向对象组件，特别是哪些在其他应用程序中重用的组件。面向对象的特性通常被认为对减少测试有助，但事实并非如此。例如，封装孤立了组件，使其可以单独开发。轻易的认为如果一个程序员重用了一些没有改变的组件，重用了一些改变的组件，仅仅是改变的代码需要测试。然而，“一个单独经过足够测试的程序在组合测试上可能是不够的”(Perry 和 Kaiser1990)。实际上，他们想说明的是，当我们增加一个子类或修改一个现有的子类，我们必须重新测试从其祖先父类继承的所有方法。

他们也检查是否有足够的测试用例。对于过程语言，我们能用一系列测试数据测试系统；

当系统作过改动时，我们测试改变是正确的，用现有的测试数据检验额外剩余的功能是一样的。但 Perry 和 Kaiser 这种情况在面向对象系统是不同的。当一个子类替代了一个局部化同名的继承方法，重载的子类必须用可能不同的测试数据重新测试。Harrold 和 McGregor 描述了一种用一个面向对象系统的例时测试用例减少额外测试数量的方法。他们首先测试没有父类的基类；测试策略是单独测试每个函数，然后测试函数间的交互。接着，他们提供了一种算法，更新父类测试历史的增量；仅新的属性或影响到继承设计的被测试。

Graham 在两个方面摘要了面向对象和传统测试的不同。首先，她指出，哪些方面是的面向对象测试变得更容易那些变得更难。例如，对象趋于变小，通常存在于组件中的复杂度增加了组件间的接口的复杂度。这种不同意味着单元测试不太难了，但集成测试必须更广泛。正如我们看到的，封装通常认为在面向对象设计中是好的特性，但在集成测试中需要广泛的领域。

同样，继承引发了更多测试的需要。一个继承函数需要额外的测试，假设

它重新定义

它在继承的子类有特定的行为

这个类中其他函数被假定为一致的

图 8.16 描述了 Graham 对于这些不同的看法

Graham 也在研究测试过程中面向对象影响的步骤。图 8.17 中的图表是 Kiviat 或者雷达曲线图，比较了面向对象和传统测试的不同。灰色多边形表示需求分析和确认，测试用例的生成和特殊需求的覆盖分析。灰线离图表的中心越远，面向对象与传统的测试区别越大。

需求可能在需求文档中表述，但很少有工具支持对象和方法需求的确认。

同样，大多数帮助生成测试用例的工具却没能驾驭对象和方法表述的模型。

多数源代码评估是为过程性代码定义，不是为了对象和方法。在评估面向对象系统的尺寸和复杂性上传统的方法诸如计转器将毫无用处。随着时间的过去，研究员建议和测试有用的面向对象评估，在此点上两者的不同将减小。

由于对象的交互是复杂性的根源，代码覆盖评估和工具在面向对象测试中要比传统测试用处小。

8.6 测试计划

正如我们所看到的，测试组件和集成组件搭建系统涉及了太多东西。谨慎的测试计划帮助我们设计和组织测试，因此我们有信心进行恰当地彻底地测试。

测试过程的每一步都要计划好。实际上，测试过程在开发周期中有其自身的生命周期，它可以与其它开发行为并行进行。特别是，我们必须计划一下每个步骤：

1．确定测试对象
2．设计测试用例
3．编写测试用例
4．测试测试用例
5．执行测试
6．评估测试结果

测试对象告诉我们生成那些种类的测试用例。而且，测试用例的设计是成功测试的关键。如果测试用例不具有代表性，不能彻底地执行函数，显示系统的正确性和合法性，余下的测试过程是徒劳的。

因此，测试首先从复查测试用例开始，证明它们是正确的，可行的，提供预计覆盖的等级，演示预期的功能。一旦这些检查做到，我们才实际执行测试。

计划的目的

我们用测试计划来组织测试行为，测试计划考虑测试对象，合并任何由测试策略或工程最后期限要求的时间安排。系统的开发生命周期需要几个级别的测试，从单元和集成测试开始，到演示所有系统的功能。测试计划描述的方式如同我们演示给用户软件正确工作（例如软件错误少，严格按照需求定义履行功能）。因此，一个测试计划不仅要求单元和集成的测试，还要求系统的测试。测试计划是整个测试行为的向导。它解释了谁行使测试，为什么行使测试，测试是怎样执行的和什么时候进行测试。

开发测试计划，我们必须了解需求，功能描述和系统设计编码中的模块层次。当我们开发每一个系统元素时，我们用我们所了解的选择测试对象，定义测试策略，生成一系列测试用例。随之发生的是，测试计划随着系统的开发而开发。

计划的内容

测试计划从测试对象开始，进行每一种类型的测试，从单元到功能到接受和安装测试。因此，系统的测试计划是一系列测试计划，每个测试种类都要求严格保证。下一步，测试计划考虑测试怎样运行，应用那些标准决定测试结束。知道什么时候测试结束通常不是很容易的。我们已经看到一些代码的例子，在它们身上执行每一个输入数据和条件的组合是不可能或不切实际地。选择所有可能数据的子集，我们要承认可能增加了遗漏测试特殊错误类型的可能性。在完备性和现实中成本时间的折衷是我们的最终目的。在本章后面，我们考虑如何估计遗留在代码中的错误数量和鉴别有错误倾向的代码。

当测试小组承认一个接触过的测试对象，我们说这个测试对象是**完整定义的**。然后到了我们决定如何集成组件生成一个工作系统的时候了。我们在组件级别考虑语句，分支，路径覆盖，同样在集成级别考虑自顶向下，自底向上和其他策略。合并组件生成一个整体，得出的结果计划有时叫做**系统集成计划**。

对于测试的每个过程，测试计划详细描述行使每个测试用到的方法。例如，单元测试可以由非正式的预排和正式的代码检查组成。其后是分析代码结构和分析代码实际性能。计划中指出每一个自动操作需要的支持，包括需要工具用到的条件。这些信息帮助测试小组计划测试行为和制定测试时间表。

详细的测试用例列表辅助每个测试方法或技术。测试计划也解释了测试数据如何生成，如何捕捉每一个输出数据和状态信息。如果需要一个数据库来跟踪测试，数据和输出，数据库和它的用途也要描述出来。

因此，当我们阅读测试计划时，我们对于如何进行测试和为什么进行测试有了全面的了解。在我们设计系统的同时，编写测试计划，我们身不由己明白了系统的全部目标。实际上，有时，测试的看法鼓励我们对于现实的问题和设计的恰当性产生疑问。

许多用户在需求分析中规定测试计划的内容。例如，美国保安局在系统的开发中，为开发者提供了自动化的数据系统文档记录标准。该标准这样解释测试计划的：

测试技术是一个能够指导测试的工具，包含了事件的顺序时间表和物资列表，这些东西会影响一个完整的自动化数据系统的广泛测试。指导职员的文档应该用非技术语言描述，指导操作员的文档应该用恰当的术语描述。

我们在第 9 章深入探讨测试计划的细节。

8.7 自动测试工具

有许多自动化工具帮助我们测试代码组件，我们在本章已经提过很多回了，例如自动定理证明和符号执行工具。但是一般说来，不必要的话，在测试过程中，仅有几处地方工具是有用的。

代码分析工具

有两类的代码分析工具。当程序实际上没有执行时，需要**静态分析**工作。程序真正运行时，需要**动态分析**工作。每种类型的工具反馈代码自身的信息或者正在运行的测试用例。

静态分析 一些工具能够在源程序运行前分析它。这些工具能够研究出程序或一组组件的正确性。分为四类：

1．**代码分析** 自动根据语法为组件评估。如果语法错误，构造有错误趋势，元素没有定义，语句高亮。

2．**结构检查** 该工具根据提交的输入从组件中生成一个图表。图表描述了逻辑流程，该工具检查结构缺陷。

3．**数据分析** 该工具复查数据结构，数据声明和组件接口，然后指出组件中不恰当的连接，数据定义的冲突，不合法数据应用。

4．**顺序检查**：该工具检查事件的顺序；如果是错误顺序的编码，事件高亮。

例如，代码分析能够生成一个符号表，纪录一个变量的定义和什么时候用到。例如定义用途测试采用这样的策略。同样，一个结构检查读入一个程序，决定所有循环的位置，标志没有运行过的语句，指出一个循环中间的分支等等。一个数据分析能够在分母为零时警告我们；它还能检查子程序的参数是否传递正确。系统中的输入输出组件提交给顺序检查确定事件是否在正确的顺序下编码。例如，一个顺序检查可以保证所有文件在修改前都被打开。

从许多的静态分析工具，尺度和结构特征暗含在输出中，以至于我们能够更好的了解程序特征。例如，流程图通常补充一系列所有可能程序路径的清单，允许我们计划测试用例进行路径测试。我们也配备扇入扇出的信息，程序中操作者和操作数，决策点的数量和一些代码结构复杂度的尺度。在图 8.18，我们看到一个静态分析程序的输出例子。图中比较了一段特殊代码的发现和历史信息的大型数据库。比较中不但包括嵌套深度，连接的尺度和决策的数量，还包括潜在错误和无用变量的信息。这样的描述告诉我们测试是多么容易，警告我们在正式测试运行前我们可以修改的那些可能错误。

动态分析 许多时候，由于一些并行的操作，系统是难于操作的。这种情况在实时系统容易发生。在这些情况下，很难找到测试条件和生成有代表性的测试用例。自动化工具能够在程序的执行中通过保留条件的暂时瞬间，让测试小组抓住事件的状态。因为这些工具检测和汇报程序的行为，有时被称为**程序检测器**。

检测器可以列出组件被调用和一行代码执行的次数。这些统计结果告诉测试者测试用例的语句和路径覆盖情况。同样，检测器能够汇报一个决策点是否在所有的方向上都分支过，由此提供分支覆盖的情况。摘要统计也要上报，他们统计测试用例覆盖在语句，路径和分支百分比。当测试对象考虑到覆盖方面，这些信息是重要的。例如，伦敦航空交通控制系统需要 100%语句覆盖测试的合同。

附加的信息也会帮助测试小组评估系统的性能。例如，统计特殊变量的情况：他们初始值，最后值，最小值，最大值。通过在系统中设定断点，测试工具会报告一个变量获得或者超过一个特定的值的出现情况。当达到断点，一些工具会停下来，允许测试者检查内存中的

内容或者特定数据元素的值；有时可以在测试的进行期间改变值。

对于实时系统，在执行过程中，捕捉尽可能多的关于特定状态或条件的信息可以在执行后为测试提供附加信息。控制流可以在某个断点向前或向后跟踪，于是测试小组就可以根据数据的变化进行检查。

测试执行工具

迄今为止我们描述的工具都集中在代码上。其他的工具能够用来自行计划和运行测试。在当今给定大多数系统的大小和复杂度，自动执行工具在处理大量的测试用例是必需的，以便达到彻底测试整个系统的目的。

捕捉和回放

当计划好测试，测试小组必须在测试用例中规定提供什么样的输入和期望要检测行为产生什么样的输出。**捕捉和回放**或者**捕捉和播放**工具捕捉按键，输入，对其做出反应作为测试的运行。该工具比较期望的和实际输出。之间的差异报告给小组，捕捉来的数据帮助小组跟踪差异找到根源。这种类型的工具在发现错误改正后特别有用；经常用来检验修正了错误，没有带来新的错误。

存根和驱动

我们前面指出了在集成测试中存根和驱动的重要性。商业工具能够自动帮助你生成存根和驱动。但是检测一个特定的组件，测试驱动要比简化程序更广泛。这个驱动能够

1. 使所有恰当的状态变量为一个给定测试用例做准备，然后运行测试用例
2. 模拟键盘输入和其他相关数据对于条件的反应
3. 比较实际输出和期望输出，报告差别
4. 跟踪贯穿执行过程的路径
5. 重置变量为下一个测试用例做准备
6. 于一个调试包交互，这样在测试过程中，错误能够跟踪并及时修复。

自动化测试环境

测试执行工具可以和其他工具联合起来搭建一个广泛的测试环境。通常我们所说的工具连接着测试数据库，尺度工具，代码分析工具，文本编辑，模拟和模型工具，尽可能的使测试过程自动化。例如，数据库跟踪测试用例，为每个测试用例存储输入数据，描述期望的输出，记录实际的输出。然而，找到错误的证明不同于错误的定位。测试需要手工的努力去跟踪难题到根源；自动机可以帮助但决不能替代必要的人类职责。

测试用例的生成

测试依赖于谨慎的彻底的测试用例定义。由此，自动生成一部分测试用例是有益的，这样我们确信我们的用例覆盖了所有可能的情况。有一种类型的工具帮助我们做这些工作。**结构测试用例生成器**基于源代码的结构生成测试用例。它们为路径，分支，语句测试列出测试用例。通常它们也包括启发式测试帮助我们达到对程序的最好覆盖。

其余的测试用例生成器基于数据流，功能测试（也就是执行所有可能的状态，影响一个给定函数的完成），或者在输入域中每个变量的状态。其余的工具能够生成随机几组测试用例，通常用来支持可靠性模型

8.8 什么时候停止测试

我们在前面的几章指出软件的质量可以由许多方式衡量。一种评定一个好的组件方式是它含有的错误数量。自然会这样的设想，软件错误难于查找，也难于纠正。代码首次检查时，查出越容易纠正的错误，在测试过程就会寄存越难的错误，这看上去也合情合理。然而,Shooman 和 Bolsky 发现事实并非如此。有时花费大量的时间寻找无谓的错误，许多这样的错误在测试过程中容易忽视，或者没有显示出来。而且,Myers 汇报到随着检测的错误数量的增长，越有可能存在没有检测出来的错误。图 8.19 显示。一个组件存在许多错误，我们希望在测试过程中尽可能的发现它们。然而，此图告诉我们如果我们在开始发现大量的错误，我们可能仍存在着大量的未检测出来的错误。

除了和我们的直觉相反，这些结果也使我们难于确定何时停止查错。我们必须估计仍存的错误，不仅知道何时停止我们对更多错误的研究，而且在某种程度上给我们所编写的代码一些自信。在系统发布后，如果错误需要遗留下来检测，错误的数量也暗示了我们需要维护工作。

错误种子

Mill 开发了一种叫做**错误播种**或**毛病播种**来估计程序中的错误数量。基本前提是测试小组的一个成员故意在程序中插入或（播入）已知数量的错误。然后，其它的组员尽可能的找出错误。没有发现的种子错误数量作为程序中所有错误（包括杂乱的，非播种的错误）数量的指示器。也就是说，找到的种子错误于所有错误的比率等于检测出的非播种的错误于所有非播种错误的比率。

因此，如果程序中播种了 100 个错误，测试小组仅找到了 70 个，代码中可能存在 30% 未被发现的错误。

我们可以更正式的表达这个比率。让 S 表示程序中播种的种子错误数，让 N 表示非播种的错误数。如果 n 是测试时检测的非播种错误实际数量，s 表示测试时检测出的播种错误数。估计所有的未发现的错误数是

虽然简单有效，此方法假设程序中播种的种子错误是同一类的，复杂性与实际错误相同。但是我们在找到错误前不知道错误的类型。因此很难让种子错误代表实际错误。一种方法可以增加典型性的可能，就是基于过去相同的项目中历史纪录过的种子错误。然而，这种方法仅适合于我们过去构造过相同系统。正如第 2 章所指出的，事物并不如我们意识中那样，看似相同，其实是不同的。

为了克服这些障碍，我们采用两个独立的测试小组测试相同的程序。叫它们小组 1 和小组 2。让 x 代表小组 1 发现的错误数，y 表示小组 2 发现的错误数。一些错误两个小组都会发现。用 q 表示这样的错误数，因此 q<=x,q<=y，最后，让 n 表示程序所有的错误的数量，我们希望估计出 n。

每个小组测试的效力可以通过计算每个组的错误分数来衡量。因此，小组 1 效力 E1 表示为

小组 2 的效力 E2 为

小组的效力标志了一个组从现有错误中检验出来的能力。因此，如果一个组找到程序中一半的错误，效力是 0.5。考虑小组 1 和小组 2 检测错误。如果我们假设小组 1 在任何一段程序任何一部分都有如此的效力，我们考虑小组 1 从小组 2 发现的错误中找到错误的比率。

就是说，小组 1 在小组 2 的 y 个错误中找到了 q 个错误。因此，小组 1 的效力就是 q/y。换句话说，

然而，我们知道 E2 是 y/n，因此我们为 n 得出下列公式

我们已知 q 的值，我们通过 q/y 估计出 E1, 通过 q/x 估计出 E2，所以我们有足够信息估计出 n。

来看看这种方法如何工作的，假想两个测试小组测试程序。小组 1 找到 25 个错误，小组 2 找到 30 个错误。两组发现相同的错误数是 15。这样，我们就有，

x=25

y=30

q=15

根据小组 1 的效力 q/y 或 0.5 估计出 E1，由于小组 1 在小组 2 的 30 个错误中找到了 15 个错误。同样，根据小组 2 的效力 q/x 或 0.6 估计出 E2。因此，我们估计出 n，整个程序的错误是 15/(0.5*0.6)，是 50 个错误。

测试计划定义的测试策略引导测试小组决定何时停止测试。测试策略可以用估计方法决定何时测试停止。

软件的信心

我们用错误估计来告诉我们我们在软件测试中寄予了多少信心。**信心**，通常用百分比表示，表明了软件无错的可能性。因此，如果我们说一个程序 95%级信心无错，意味着软件无错的可能性是 0.95。

假想我们在程序中播种了 S 个错误。我们宣称代码中仅有 N 个实际错误。我们测试程序知道发现所有的 S 个播种错误。如以前，如果 n 表示测试中发现的实际错误数量，信心级别可以通过公式计算，

例如，假想我们宣布一个组件是无错的，意味着 N 是零。如果我们播种了 10 个错误，找到了所有 10 个不包括未发现的错误，然后我们用 S=10 和计算信心级别。这样，C 等于 10/11，信心级别是 91%。如果需求或合同上需要信心级为 98%，我需要播种 S 个错误种子，以使 S/(S-0-1)=98/100。解这个方程，我们发现我们必须播种 49 个错误，继续测试发现所有 49 个错误(不包括找出未发现错误）。

这种方法面临一个重要的难题：在所有种子错误检测出来之前，我们不能预测出信心的级别。Richards 建议修改此方法，使得信心级别不管所有的错误是否被找出，根据找出种子错误数量，估计结果。C 代表：

这些估计结果假定所有的错误检测出的概率是相同的，事实上是不太可能的。然而，许多其他的估计把这些因素考虑进去。这样的估计方法不仅给了我们一些程序中信心观念，而且提供了一方的利益。许多程序员试着下结论，每次发现的错误就是最后的错误。如果我们估计仍存的错误数，或如果我们知道需要发现多少错误才能满足信心需求，我们就会有动力继续测试寻找错误。

这些方法在重用的组件上评估信心同样可用。我们研究一个组件的错误历史，特别是错误种子发生时，用诸如这些方法决定在不用再次测试组件条件下，重用组件中存在多少信心。或者我们给组件播种，用这些方法确定信心的基线级别。

其它停止标准

测试策略本身能够用来定义测试停止标准。例如，当我们做语句，路径，分支测试，我们能够跟踪需要执行的语句，路径，分支数量，根据测试中遗留的语句，路径，分支的数量来决定我们的测试进度。

许多自动化的工具为我们计算覆盖范围。考虑从 Lee 和 Tepfenhart 实现一个计算机游戏的代码。

程序代码

工具可能添加到清单上一个符号标记，分支的位置。如程序所示。因此，当 i 在语句 6 的循环参数内时，经过分支 1 路径，当 i 不在循环参数内时，经过分支 2 的路径。当 v.X 在语句 13 为零时，经过分支 3，当 v.X 不为零时，经过分支 4 等等。某些自动化的工具计算测试覆盖的所有路径。在这种情况下，有 8 中可能。随之测试的进展，工具能够生成一个象表 8.8 和 8.9 的报告，以便我们清楚还需经过多少路径才能达到路径或分支的覆盖。

表 8.8 路径遍历摘要

		本例测试：			累积：		
测试用例	路径数	调用	路径遍历	覆盖率%	调用	路径遍历	覆盖率%
6	8	1	4	50	5	6	75

表 8.9 未执行路径

测试用例	遗漏路径	总计
6	1 2 3 4	4

鉴别有错误趋势的代码

很多方法基于相同应用程序的错误历史，鉴别有错误趋势的代码。例如，一些研究员在开发和维护中跟踪每个组件的错误发现数量。他们也收集每个组件例如大小，决策点的数量，操作数和运算符的数量，修改次数等信息。然后，他们执行方程计算，建议你最有可能出现代码错误的组件的属性。这些方程可以用来给你建议，那些组件应该首先测试，那些在复查和测试时应该仔细检查。

Porter 和 Selby 建议用分类树鉴别有可能出错的组件。分类树分析是一种静态方法，通过对大量的尺度信息排序，创建了一个决策树说明那个尺度对于特定的属性是最好的预测。例如，假想我们收集组织机构中每个组件的尺度数据。我们收集包括大小（每行代码），代码中不同路径的数量，运算符的数量，嵌套的深度，耦合和内聚的程度（在以 1 为最低 5 为最高的比例），编写代码时间，组件中发现错误数量。我们用分类树分析工具分析带有五个或多于五个错误的组件的属性，将它与少于五个错误的组件相比。结果是如图 8.20 的决策树。

分类树用来帮助我们决定那些组件在我们现有的系统中有可能有大量的错误的。根据树的情况，如果一个组件有 100 到 300 行的代码，至少有 15 个决策点，他就有可能有错误趋势。或者如果组件超过 300 行代码，没有进行设计复查，至少改动过 5 次，他也是有出错趋势的。我们用这种分析方式帮助我们在有限的测试资源下定位我们的测试。或者我们定期为组件检查帮助在测试开始前纠错。

8.9 信息系统的例子

假想我们产生测试用例来测试 Piccadilly 系统组件，我们选择一个测试策略，计划执行组件中每条路径。我们可以决定编写测试描述输入和期望输出的测试脚本。测试过程涉及为每个测试用例比较实际输出和期望输出。如果实际输出等于期望输出，是否意味着组件是无错的？不确定。在组件中我们可能有 Beizer 称作的

一致纠正。我们考虑有如下结构的组件，来明白为什么会这样。

CASE 1:Y:=X/3;

CASE2:Y:=2X-25;

CASE3:Y:=X MOD 10;

ENDCASE;

如果我们的测试用例用 15 作为 X 的输入，期望 5 作为 Y 的输出，实际上产生的 Y 等于 5。我们不知道那些路径执行了；每一条用例生成 Y 为 5X 为 15！由此，测试用例必须补充有标志物，帮助测试小组鉴别代码中那些路径实际运行过。在这个例子中，路径覆盖工具在每个测试用例运行时，有效的跟踪了执行的语句。

因为 Piccadilly 系统是一个信息系统，我们实际上更愿意用数据流测试策略而不是结构策略。我们能够用数据字典轻易地鉴别出每一个数据元素，然后考虑他们每个可能的值。象定义用途测试策略可能是最恰当的；我们跟随每个数据项深入到组件中，寻找数据项变化的情况，检验变化是正确的。许多自动化的工具支持这样的测试：数据库仓库，测试用例生成器和测试执行监测器。实际上，我们的测试小组希望用其他工具连接存有数据字典的数据库。

8.10 实时系统

Ariane-5 系统经历了许多复查和测试。根据 Lionset，飞行控制系统采用 4 种方法测试：

1。设备测试

2。板上计算机软件测试

3。程度集成

4。系统合法性测试

所有 Arian-5 测试哲学是在每一级检查前一级不能达到的地方。在这种方式下，开发者希望提供每个子系统和集成系统的测试完整覆盖。让我们考虑后展开研究，看看在软件质量评定时为什么测试在实际飞行前没有发现 SRI 错误。（我们会在第 9 章讨论集成和合法性）

调查员报告，“关于倒数，飞行时间顺序和 Ariane-5 的弹道，没有行使测试验证 SRI 正确工作。”实际上，SRI 软件的描述在功能需求中不含有 Ariane-5 的弹道数据。换句话说，没有需求文档讨论了 Ariane-5 与 Ariane-4 弹道的不同。调查员指出，“每个临界设备强制要求的限制声明，服务于鉴别任何 Ariane-5 弹道的服从。”

因为根源处在需求上，在开发过程中，很早就被注意到。的确，复查是设计和编码的一个完整部分。调查员指出复查“在所有级别上实施，牵扯项目中所有主要的参与者”他们得出结论“很明显，SRI 软件的限制没有在复查中完全分析出，在测试上没有意识到覆盖的不足够，以显露出这样的限制。也不可能在飞行过程中，允许修改软件（不清楚）。处于这些考虑，复查过程是软件失败的一个过失因素”。

因此，Ariane-5 开发者带有错误的自信，使软件信赖于不足的复查和测试覆盖。

有几种方法提高完整复查和测试覆盖的可能性。一个是在复查过程中请一位非专家。这样的参与者会问一些许多其它复查者认为想当然的错误的假设。另一个是检查测试用例的完

整性。要么请一个外面的参与者复查测试用例或用正式的方法评估覆盖程度。正如我们要在第 9 章看到的，Ariane-5 开发者在测试期间有许多其它机会发现 SRI 软件的错误，但都在安全网络下溜过了。

8.11 本章对于我们的意义

本章描述了许多方法，你可以用来单独测试你的组件，或者与同伴把方法综合起来应用。分清楚错误（需求，设计，编码，文档，测试用例的问题）和失败（系统功能上的错误)是很重要的。测试寻找错误，有时通过迫使代码失败寻求错误根源。单元测试

单独执行每个组件的开发活动。集成测试在规定的方法下综合组件，联合组件一起测试，帮助你孤立错误。

测试的目的是找到错误，不是证明正确性。事实上，没有错误不能保证正确性。有许多手工和自动的方法帮助你在代码中找到错误，同样测试工具会显示已经了测试了多少，什么时候停止测试。

8.12 本章对于你的开发小组的意义

测试是一个个人和集体的活动。一旦一个组件编写完，开发组的一部分人或所有人检查寻找对于编写者不明显的错误。研究文献说明了在开发过程中代码检查在查错上是非常有效的。但是，很明显，其他方法能够找到代码检查遗漏的错误。开发过程中，需要运用你建议的许多方法，尽可能的提前查出错误，所以你和你的工作小组忘我的工作是非常重要的。

集成测试也是一个小组的活动，你必须和其他组员协作，选择集成策略，计划测试，生成测试用例，运行测试。自动化的工具在这些活动中可能是有用的，它们能够帮助你和你的组员谨慎检查测试结果，验证错误和原因。

8. 13 本章对于研究员的意义

研究员继续在研究测试的同时研究大量的重要内容：

代码检查是有效的，但是不同方式的运用会更有效。研究员正在考虑用最好的方法选择代码检查小组成员，复查开发产品，与小组开会交换意见，查找尽可能多的错误。一些研究员比较有小组会议和没有小组会议两种模式，看看小组会议确实能达到什么样的效果。

研究员继续试着理解哪一种方法在查找某类错误上是最佳的。

我们今天搭建的系统要比以往几年搭建的系统远远复杂远远庞大。因此，用自动化的工具支持我们的测试变得越来越重要了。研究员正在考虑一些方法，来定义测试用例，跟踪测试，评估覆盖完整性，最终在这些活动中达到自动化的目的。

测试资源通常是有限的，特别是在市场驱动产品的规划下。研究员继续寻找方法鉴别有错误趋势的组件，以便测试能首先针对这些目标。同样，对于一个安全苛刻系统，研究员搭建模型和工具，保证最苛刻的组件彻底的测试过。

8.14 小组项目

借贷协商器需要大量的测试。因为系统对于 FCO 经济健康发展很重要，用户希望软件

尽可能的早发布。设计一种测试策略，用最少的资源测试借贷协商器。验证你的策略，解释如何知道何时停止测试，把系统转交给客户。

8.15 关键参考

测试已经是版刊杂志上特殊内容的主题，包括 1991 三月 IEEE SoftWare 和 1988 六月的 Communication of the ACM。1994 九月的 Communication of the ACM 讨论了在测试面向对象系统中的特殊想法。2000 年一月二月 IEEE SoftWare 指出为什么测试是困难的。另外，IEEE Transactions on Software Engineering 经常有针对某类错误比较不同测试方法的文章。

有几本好书详细介绍了测试。Myers 是杰出的著作，同其它几种特殊的方法一起，描述测试哲学。Beizer 在该领域有许多关键论文的参考文献，提出了对于测试考虑和方法的全面观点。他的 1995 年的书特别集中在黑盒测试。Hetzel（1984）和 Perry,Kit,Kaner,Faulk,Nguyen(1993)也是很有用的参考文献。Binder(2000)是测试面向对象系统的广泛向导。

有许多好的论文，描述代码检查的作用，包括 Weller(1993 和 1994)，Grady 和 van Slack(1994)。Gilb 和 Granham 的书(1993)在代码检查上是一个好的广泛的实用的向导。研究员继续提炼检查的方法，试图扩展到如需求这样的过程产物中去。此类工作的例子，请看 Porter et al.(1998)和 Shull et al.(2000)。

有许多自动的测试工具能够用在程序中。在软件质量工程方面，Jacksonvill，Florida 每年出版了 Testing Tools Reference Guide。有关某个工具的信息可在卖方的官方网站上找到，例如,Reliable Software Technologies 和 Rational Software。RST 网站上还有测试资源的数据库。Fewster 和 Graham(1999)的书讨论了软件自动化测试内容。

Software Research 开发并出售测试工具。每年五月在 San Francisco 和欧洲的每个秋季也有个叫做 Quality Week 的会议，参与者在会上汇报测试经验。

其他的与测试相关的会议由 IEEE Computer Society 主办和 ACM 在其网站上发布。

另外，在 Washington Dc 有关于测试计算机软件的国际会议和展出。每年的主题集中在测试的不同的方面，例如测试的自动化或强制测试。需要更多信息,联系 G.Houston-Ludlam, ginger@fron-tech.com。

8.16 习题

1. 研究惠普的错误分类计划，如图 8。1 是正交分类么？如果不是，解释为什么，使其是正交分类。
2. 让 P 为程序组件，它读入 N 个纪录的清单和记录键的转换条件。前七个记录特征构造了记录键。P 读入记录键，生成一个仅包含在规定范围内记录键失败纪录的输出文件。例如，如果范围为”JONES”到”SMITH”，输出文件由根据词典编撰方式，记录键在 JONES”和”SMITH”之间，所有记录组成。
3. 完成图 8。5 中的程序证明。换句话说，写出对应于流程图的断言。然后，找到输入条件到输出的路径。证明此路径正确。
4. 假设一个程序包含 N 个决策点，每个有两个分支。测试这样的程序，需要多少的测试用例完成路径测试？如果每个决策点有 M 个选择，需要多少个测试用例完成路径测试？此程序的结构能减少用例的数量么？举例证明。
5. 考虑一个程序的流程图，菱形和盒子在指示图中为节点，节点之间的逻辑箭头表示指示

边界。例如图 8。7 的程序能够用图 8。21 显示。证明一个程序的语句测试等同于在图中寻找一条包含所有节点的路径。证明分支测试等同于寻找一组路径，能够联合覆盖边缘。最后，证明路径测试等同于通过此图寻找所有可能路径。

6. 编程难题：编写一个程序，接受一个指示图的节点和边界作为输入，打印该图的所有可能的路径。你的程序中首要的设计考虑是什么？图的复杂度（就是分支，循环数量）怎样影响你用到的算法？
7. 图 8。22 显示了一个软件系统的组件层次。描述用自底向上的方法集成组件测试顺序，自顶先下的方法，修正的自顶向下方法；大爆炸方法，三明治方法和修正的三明治方法。
8. 解释一下为什么图 8。19 能够说明，如果在编译时代码中找到许多错误，你需要抛弃代码重新编写。
9. 在图 8。19 中有哪些可能的行为解释？
10. 如果一个程序播种了 25 错误。在测试时，18 个错误被找到。其中的 13 个是种子错误，5 个是非发现错误。程序中仍存的非发现错误 Mill 估计数量是多少？
11. 你宣称你的程序是 95%信心级别无错。你的测试计划要求你测试直到找到所有播种的错误。测试前需要播种多少种子才能正式你的宣称？如果由于某种原因，你不能找到所有播种的错误，Richards 公式需要多少种子错误？
12. 讨论一下测试商业苛刻系统，安全苛刻系统和失败不会严重影响系统生命周期，健壮和事务的系统有什么不同？
13. 给出一个面向对象系统，该系统需要对同步困难进行谨慎测试。
14. 如果一个独立的测试小组集成测试，在测试后仍存在一个严重的错误，谁要对该错误导致的损失负法律和伦理责任？
15. 假想你正在构造一个税务准备系统，需要三个组件。第一个在屏幕上创建界面，允许用户输入姓名，地址，税收证明和金融信息。第二个组件用税务表和输入信息计算当年应付税款。第三个组件用地址信息打印包括所缴税款的联盟，周（省），城市税务报表。描述你要用到的测试策略，在测试计划中给出测试用例的轮廓。

第 9 章 测试系统

在本章中, 我们研究:

功能测试
性能测试
接受性测试
软件可靠性，可用性，可维护性
安装测试
测试文档
测试安全苛刻系统

测试系统与集成测试是有很大区别的。当你单元测试你的组件时，你已经完全掌握了测试过程。你可以创造你自己的测试数据，设计你自己的测试条件，进行自身测试。当你集成组件时，你有时是自己工作的，但是你通常会和测试小组和开发小组中的一小部分合作。然而当你测试一个系统时，你将和整个开发小组合作，协调你所作的东西，并接受测试小组领导的指导。在这一章里，我们将看一看系统测试的过程：它的目的，步骤，参与者，技术和工具。

9.1 系统测试的原则

单元和集成测试的目的是确保代码能够正确的实现设计；那就是：程序员写代码，实现

设计者的意图。在系统测试中，我们有一个很难的问题：确保系统能够做顾客想要做的事。为了理解怎样实现这个目标，我们首先必须理解系统的错误来自那里。

软件故障的来源

要消除软件的毛病所引发的错误需要伴随恰当的条件。那就是：可能有一个错误存在代码里，但是如果这段代码没有被执行，或者是这段代码没有被执行的足够长，或者是适当的条件所引发的问题，那时我们将看不到软件的故障。因为测试不能检测到每一种可能的条件，我们本着发现错误的目的，希望在程序中我们能够消除所有在实际系统的使用中所能导致失败的错误。

软件错误能够被插进一个任务，一个设计，或是代码组件，或是在文献中，在开发或维护中的任何一点。图 9.1 说明了在每一个开发中所可能引起的错误。虽然我们可以尽可能早的发现和纠正错误，但是在集成测试后系统测试的错误信息可能还是会出现。

在开发过程中错误迟早会引入系统，例如当纠正一个最新的错误时就有可能。例如，检测软件能导致来自任务的错误信息。如果需求是模棱两可的，就由于顾客不能确定他的需要，或是由于我们误解了顾客的意图，将导致同样的结果：系统不能执行顾客想要做的工作。

在系统设计的过程中可能会产生同样的交流错误。我们可能误解一项任务，写出一个错误的设计说明书。或者我们理解了这个任务但是说明书中的措辞太贫乏以至于后来读到它和第一次使用的人不能理解。类似的，我们可以假设许多特征和关系不能被其他的读者接受。

相似的事件能导致程序设计的错误。当系统设计在程序设计说明书中被翻译成更低级的描述时，错误的解释是经常出现的。程序员起初和顾客讨论系统目标和系统功能性时就将程序分为几个等级。用一个“树“而不是用“森林“表示出来，可以看到程序员并不能在开发周期的初步阶段就将存在的所有错误都找出来。出于这个原因，任务和设计的复查对于保证系统结果的质量是非常必要的。

开发小组的程序员和设计员很可能不能使用恰当的语义和语法来记录他们的工作。编译器或者是汇编程序能够在运行前找到一些错误，但是他们不能找到这样的错误，描述意图是正确的但是与程序员和设计员的意图不符的错误。

一旦程序组件测试开始，错误将会无意识的增加并影响其他问题的纠正。这些错误经常是很难查出来的。因为他们经常是只有当某个函数被执行时，或是在某种条件下才会出现。当有一个新的错误无意中被加入时，如果那些函数已经被检测，那么这个新的错误可能很久以后才会找到，因为它的源头并不明确。这种情况经常出现在我们使用从别的程序中取来的代码再经过修改以适合我们现有的程序这种情况中。代码设计的微小差别是不明显的，而我们的改动实际上可能并不能使情况变得好起来而是更糟糕。

例如，假设你正在测试组件 A,B,C。对其进行单独测试。当将三个组件一起测试时，你会发现 A 传递给 C 一个错误的参数。在修改 A 的过程中，你确定现在的参数传递是正确的了，但是你增加的代码设置了不恰当的指针。因为你不能再回去单独的测试 A 代码，所以你只有在已有的测试中才能找到新的错误的踪迹，以至于使 A 时错误的源头变得模糊。

同理，维护也能引进新的错误。系统的提高需要改变需求，系统结构，程序设计和自身的执行，随着系统提高的描述，设计和编码，将引进许多种错误。另外，系统并不能执行其正常的功能，因为使用者并不知道系统是怎样执行工作的。如果文献不清楚或不正确，将得出错误的结论。人文因素，包括使用者的直觉，在理解系统，解释信息和所要求的输入信息等方面都起了很大的作用。对于不适应这个系统的使用者是不能够正确或者说是充分使用本系统的。

测试过程将是彻底对系统函数进行操作达到每个人都满意：使用者，客户，和开发者。

如果测试时不完全的，那么错误将仍然不会被查出。正如我们所知越早查出错误越好。错误查出的越早越容易所花费的代价也越少。因此，尽早彻底的进行测试不仅能帮助我们尽早的查出错误，而且更容易找出原因。

图 9.1 表示了错误的原因，没有进行证明。因为测试的目的是尽可能多的发现错误，所以很关心错误在哪里。知道错误是怎样产生的能够给我们提供当系统检测是哪里需要注意的线索。

系统测试过程

在测试系统中有以下几步：

1. 功能测试
2. 执行测试
3. 协议测试
4. 安装测试

图 9.2 说明了其步骤。每一步都有一个不同的中心，每一步的成功都依赖于它的目标或说是目的。因此，复查测试系统每一步的目的是很有益处的。

过程目的 起初，我们用系统测试功能的执行情况。我们开始对一组组件进行单个测试然后再结合起来。功能测试检验了集成系统是否按照需求中描述那样执行它的功能。例如，一个银行统计包的功能测试是核实这个包是否能正确提供贷款，输入一个回车，计算利润，打印一个空格，等等。

一旦测试组认为功能运作能够向所说明的一样，**性能测试**将把集成组件与非功能性系统需求进行比较。这些需求，包括安全性，准确性，速度，可靠性，都限制了系统功能的执行。 例如，一个银行统计包裹的性能测试其速度的增加就要依靠一些统计量，计算的精度，防御的安全性，用户咨询的反馈时间。

就这一点来说，系统能够按照设计者的意图运行。我们称其为**校验系统**；它是设计者对需求说明的解释。其次，根据复查需求定义的说明文档，我们将系统同顾客的期望作一下比较。如果我们对系统表示满意，我们就已经满足需求，那么我们得到一个**有效的系统**；也就是说，我们已经证明了这个系统的有效性。

迄今为止，所有的测试都是有开发者根据他们对系统的理解和系统的目标进行的。客户也对系统进行测试，确保满足他们对需求的理解，这一定与开发者有一定的区别。这个测试称为**接受性测试**，保证客户要求的系统就是他们所构建的。**接收性测试**有时在实际环境中运行，但是他也经常用作一种测试工具，这就与原本的目的有一定的区别了。由于这个原因，我们可以运行一个最终得到的安装测试，可以使使用者检验系统的功能并且可以验证在实际运行环境中所能导致的额外的问题。例如，一套海军系统从开发者的角度可以设计，构造，和测试，构建一艘轮船，但是这并不是一艘实际的轮船。一旦开发测试完成，另外一套安装测试将用于甲板上的每一种类型船只的系统，并最终采用本系统。

构建或集成计划。 理想的情况，在程序测试后，你能够将所选择的组件看作为单个的实体。然后，在系统测试的第一步期间，集成性的选取将是对先前描述的一个提高。然而，当测试一个巨大的组件时，庞大的系统有时是难以处理的。实际上，这样的系统经常是在阶段性开发使用的，仅仅是因为他更容易构建和测试小型的组件。因此，你可以选择阶段性的系统测试。我们在第一章已经看到系统可以被看作一系列嵌套层或是子系统。每一层至少负责他所包含的子系统的功能。相似的，我们可以划分测试系统为一系列的嵌套子系统并每次在一个子系统上执行系统测试。

子系统是在预先定义的标准上定义的。通常，划分的基础是功能性。例如，我们在第8

章看到微软划分一个产品为三个子系统的基础是最标准的功能，所期望的功能，和所需的最小功能。相似的，一个电讯通话线路系统可以被分为以下的子系统：

1 交换站通话线路

2 区域代码通话线路

3 州，省，或地区范围的通话线路

4 国家范围通话线路

5 国际通话线路

每一个更大的子系统都包括所有他前面的小的子系统。我们从第一个系统的功能开始测试。当所有的交互站功能测试成功时，我们进行测试第二个系统。相似的我们轮流测试第三个，第四个，第五个系统。最终成功的测试整个系统，额外的测试使错误的检查和纠正相对于我们只关注整个大的系统来说变得容易了。例如，一个问题出现在州，省，或地区的通话功能测试中，只可能是州的代码错误导致的而不应该是区域代码或交互信息的错误。因此，我们能缩小我们的搜索范围只在子系统3中了影响3的1，2中的代码。如果这个问题出现在所有的子系统合并的情况下，我们就不能这么容易的精确定位。

增量测试需要精心的策划。测试组必须创造一个**构建计划**，或**集成计划**以定义子系统的测试和描述方法，定位，时间，和由谁操纵这个测试。我们在集成测试中所讨论的许多问题都必须由构建计划决定，包括集成的顺序和存根和驱动的需要。

有时，一个构造计划的一级或者子系统称为一个自旋。旋转顺序编号，低级的叫做旋零。对大系统来说旋零通常是一个最小的系统；它甚至通常只是宿主计算机上的操作系统。

比如说，电信系统的构建计划也许会包含一个与表 9.1 类似的日程安排。构建计划用数字，功能内容和测试安排来描绘每一个旋转。如果对第 n 个旋转测试成功但在第 n+1 个旋转上出现了问题，那么最有可能出问题的原因就是旋转 n 与旋转 n+1 之间有所差别，也就是从一个旋转到另一个旋转出现的额外的功能。如果在两个连续的旋转之间的差别很小，那么相对来说我们就会很少有机会能查出出现问题的所在。

表 9。1 电信系统的构件计划

旋转	功能	测试开始	测试结束
0	交换站	9 月 1 号	9 月 15 号
1	地区编码(类似邮政编码	9 月 30 日	10 月 15 号
2	州/省/地区	10 月 25 日	11 月 5 号
3	国家	11 月 10 日	11 月 20 日
4	国际	12 月 1 日	12 月 15 日

旋转的数量以及它们的定义主要依靠我们的资源和我们客户的资源。这些资源不但包括硬件和软件，也包括时间和人力资源。一个微小的系统被放在一开始的旋转里，同时接下来的旋转尽可能的及早的利用下面最重要的和最关键的功能来联系到一起。比如说，考虑一下图 9.3 所出示的星际网络。 星际中心是一个从几个小型计算机中接受信息的计算机，每一个小型机捕捉来自传感器中的数据然后把它们进行传输以用于工作。这样，中央计算机的主要功能就是翻译并吸收来自外部计算机中的信息。由于这些功能对于整个的系统来说是很重要的，因而要包括早期的一个旋转。事实上，我们可以用以下的方法来定义旋转：

旋转 0：测试中央计算机的普通功能

旋转 1：测试中央计算机的信息传输功能

旋转 2：测试中央计算机的信息吸收功能

旋转 3：在独立方式方面测试每台远离中心(外围)的计算机

旋转 4：测试外围计算机的信息传输功能

旋转 5：测试中央计算机的信息接收功能

以下依此类推。

旋转的定义也依赖于系统组件以独立方式运行的能力。与在旋转里包含一个部件相比，模拟一个系统中丢失的部件会更难，这是因为在部件中的相关性有时会需要像通常的代码一样多的模拟代码。记住我们的目的是为了测试系统。建构和应用测试工具所需的时间及代价在测试通常的系统时也许会更值得一些。这种权衡与在单元测试和整体测试期间所涉及到的选择一个测试原则是相似的：因为能够测试这些模拟的原始部件。所以在编程期间需要大量的时间来开发许多的存根和驱动程序。

配置的管理：

我们经常在一个阶段或是部分中测试一个系统，依靠的是旋转(如前所说)或是子系统，功能，或是其它使得测试更容易掌握的分解方法。(在本章的后面就会看到这些测试策略)无论如何，系统测试必须考虑用于开发的几个不同的系统配置。一个**系统配置**就是将发送给特殊客户的系统部件的集合。比如说，一个数学计算包也许会包含在一个用于 UNIX 机器中进行出售，或是在另一个 Windows 的机器，或是 Solaris 系统中进行出售。通过那些运行在芯片中的某类东西或是利用手头那些特殊的设备就可以大大的区别开这些配置。通常，我们开发运行在每一个机器中的软件内核，并且我们用在第 5，6，7 章中描述的规则利用少量独立部件的配置中的差异来进行隔离。比如说，核心的功能也许包含在组件 A，B 和 C 中；那么，配置 1 就包括了 A，B，C 和 D，并且配置 2 包含了 A，B，C 和 E。

开发和测试这些不同的配置需要**配置管理**，它用来空值系统的差异以减少错误的出现。在以前的章节中我们已经看到了配置管理组是如何确保需求，设计，或是在文献中所反映出来的代码的变化的，以及由于变化而带来的组件的影响。在测试期间，培植的管理是特别重要的，这在测试人员和开发人员中是同样重要的。

版本和发布。对一个特定的系统来说，一个配置通常就叫做一个**版本**。这样，一个软件包的初始化传递可能包括了几个版本，软件将会被用在每一个平台或是每一种情况下。比方说，也许会创建一个飞机软件，其中版本 1 用于海军的飞机，版本 2 用于在空军飞机上，版本 3 用于在商业飞机上。

由于对软件进行了测试和使用，那么发现的错误就需要进行更正或是对原来的功能进行细微的改变。一个软件新的**发布**就是一个用于代替旧有的更好的系统。通常，是用版本 n，发布 m，或是版本 n.m 来描述一个软件系统，并且随着系统的完善和成熟，这些数字能够反映出这个系统的状况。版本 n 通常使用来代替版本 n-1，而发布 m 则是代替发布 m-1。(单词"版本"可以有两个不同的意思：它是每一种平台或是操作系统的版本，或是一个阶段产品的相关顺序。这个术语通常可以在它所使用的上下文中去理解。比如说，一个销售商可以提供产品的版本 3 用在 UNIX 平台上，版本 4 用在 Windows 平台上，它们每一个版本都提供相同的功能。)

配置管理组就是要确保在每一个版本或是版真正使用之前是正确的或是稳定的，并且发生的变化也应是准确和及时的。准确性是很重要的，因为在纠正存在的错误时，我们要避免生成新的错误。同样的，及时性也是重要的，因为在检测组查找额外的错误时，同时会有错误的检测和更改。这样，那些试图处理系统错误的人就会用反映现在系统状态的组件和文档来进行工作。

跟踪和测试版本在我们进行阶段的开发是显得格外的重要。就像我们在以前的章节中说到的那样，一个产品系统就是一个版本，它是指根据客户需要的一部分而进行的测试并执行。下一个具有更多特征的版本，在用户执行产品系统是会进行开发。这样的一个开发系统

会被创建并被测试，当测试完成以后，开发系统就会取代产品系统而变成了一个新的产品系统。

比如说，假如一个发电厂在控制间里正进行着工作。发电厂里的操作人员已经训练的可以手工的做每一件事情，但用计算计工作却显得很不自然，因此我们决定建立一个阶段系统。第一个阶段与手工系统基本上是一样的，但是可以让操作人员去做一些自动纪录的工作。第二个阶段在第一个阶段的基础上又添加了几个自动化的功能，但控制室里一半的功能还是手工的。紧接着的阶段继续自动选择功能，在以前的阶段上创建，直到所有的功能都是自动化为止。通货这种方式扩展自动化系统，我们可以让操作人员在新系统面前逐渐感到习惯并适应。

在阶段开发的任何时间，发电厂的操作人员都要用到完整的测试过的产品系统。与此同时，我们还要致力于下一个阶段，去测试开发系统。当开发系统测试完毕并且发电厂的操作人员准备利用的时候，这个开发系统就会成为产品系统同时我们转向下一个阶段。当工作在开发系统时，我们就在当时的产品或是操作系统中加入了一些额外的功能从而形成了新的开发系统。

然而在应用一个系统的时候，也许会出现一些问题并把它们提交到我们面前。这样，一个开发系统经常就会具有两种功能：它会给下一个阶段增加功能，并且还就纠正在以前版本中出现的问题。一个开发系统因此也能够增加新的组件，还能对存在的组件进行改变。但不管怎么样，这个过程容许已经测试过的组件出现差错。当我们写一份构建计划和测试计划时，我们应该考虑到控制从一个版本或是发布到下一个它们之间的变化，使之成为必要。而额外的测试应能够确保开发系统的职能至少和当时的产品系统是一样的。但是，从一个版本到下一个版本的代码记录应该保持着准确的变化，以便于我们跟踪它们出现的问题。比方说，如果产品系统中的用户汇报出一个问题，我们就必须要知道用的是代码的哪一个版本和发布。一个版本与另一个版本之间的代码可能会有很大的不同。如果我们在错误的清单下工作，我们或许就不会找出问题产生的原因。更可怕的是，我们或许会认为已经找出了错误的原因，在根本没有纠正以前的错误的情况下又会引来一个新的错误！

回归测试。就像我们在第八章看到的那样，测试的目的就是识别出错误，而并非改正它们。然而，想发现引起问题的原因并在发现后尽可能的改正它，这种想法是很自然的。否则，测试小组就不可能判断出是否系统在正确的发挥功效，紧接下来出现的异常就会阻碍以后的测试。这样的话，任何测试计划都要包括一组向导用来纠正错误。但是，在测试期间纠正错误会带来新的问题，就像前面所说的那样。

回归测试能够识别出新的问题，这些问题也许会随着当时问题的纠正而出现。一个**回归测试**就是一个应用到一个版本或是发布上的测试，它能够确保和以前的版本和发布一样，仍然具有同样行为的同样功能。

例如，假设对于版本 m 的功能测试是成功的并且正在测试版本 m+1，那么版本 m+1 就会具有版本 m 的所有功能和一些额外的功能。在早期的测试当中，在第 m+1 行的代码线上需要进行改变，这是为发现错误所作的准备；代码现在也必须改变为的是使 m+1 的测试能够继续。如果小组能够严格的遵循回归测试的要求，那么这个测试就会包括以下几个步骤：

1. 插入你的新代码
2. 要知道对新代码所施加的测试功能对其的影响
3. 对 m 的重要功能进行测试来确保它们仍然是正确工作的(这是通常的回归测试)
4. 对 m+1 继续进行功能测试

这些步骤能够确保新增加的代码不会否定以前测试所带来的影响。注解 9.1 就说明了不进行回归测试所带来的危险。

通常，回归测试涉及到从以前的级别测试中重复利用最重要的测试；如果你在你的测

试计划中对回归测试进行具体的描述，你就也能够解释哪种测试应该被重复利用。

三角洲，分离文件，和环境编辑。这里有三种主要的方式来控制版本和发布，并且每一个在测试期间都有管理配置的提示。一些开发计划更愿意对每一个不同的版本或是发布利用**分离文件**。比如说，一个安全系统就会有两种配置说明：为机器服务的版本以能够存贮在主存中所有的数据，而版本二具有很少的内存，这样数据在某种的环境下就必须存贮在硬盘里。系统的基本功能或许是一样的，通过由 A1 到 Ak 控制，但是存储管理也许由版本一的 B1 和版本二的 B2 来完成。

假设在 B1 中发现了一个错误，同时在 B2 中也有，那么查询的工作在两者之间是一样的。或者是假设功能既添加到 B1 也添加到 B2 中。保持现行的版本并进行更正是困难的。所需的变化似乎是不一样的，但它们的结果在用户的眼中一定是一样的。为了着重说明这个困难，我们可以给主要的版本指定一个特殊的版本，并且定义所有的版本都与主版本不同。然后，我们只需要存储它们之间的差异，而并非对其他版本来说的每一个组件。这个不同的文件就叫做一个**三角洲**，它包含了描述如何把一个主版本转变成一个不同版本的编辑命令。我们说我们"应用了一个三角洲"来把主版本转变成了不同的版本。

工具栏 9.1 不进行回归测试的结果

不正确的进行回归测试可能会有一些严重的结果。比如说，Seligman(1997)和 Trager(1997)报告说由于 Northern Telecom 公司软件出现的一个问题导致了 167000 位加利福尼亚的居民要求交纳 667000 美元，原因就是当地电话的不可靠性。在 New York City 的顾客也曾经历过类似的问题。

在软件中一个错误所导致的问题升级到 DMS-100 电话开关上。这个引起的差错就是在收费界面上电话公司的办公人员用到了错误的地址代码，他用到了不止一个的代码。结果，本地电话的收费按照了长途电话来收费。当顾客抱怨时，当地的电话公司告诉他们这个问题正在用长途搬运公司来处理，然后长途搬运公司把顾客踢回到了当地的电话公司！这花费了当地电话公司大约一个月的时间来寻找这个错误。如果 Northern Telecom 在软件升级的时候就进行回归测试的话，包括察看地区代码汇报的正确性，这个收费错误也许就不会发生了。

利用三角洲的优点就在于普通功能的改变只作用于主版本。甚至，三角洲与 fullblown 版本相比，需要很少的存储空间。但是，本质上三角洲也还有一些缺点。如果主版本丢失或是被破坏，那么所有的版本就都丢失了。更为重要的是，作为一个从主版本转换过来的每一个变体通常是很难的。例如，考虑一下一个包含下面代码的主版本：

```
26    int total=0;
```

一个三角洲文件定义了一个用新代码取代第 26 行的变体：

```
26    int total=1;
```

但不管如何，假如在主版本文件中出现了一个变化，即在第 15 行和第 16 行之间增加了一行。那么第 26 行就变成了第 27 行，同时应用三角洲改变了错误的命令。这样的话，为了在主版本和它的变体之间进行通讯就需要一些复杂的技术，这就要正确的应用三角洲。

三角洲对于维护版来说是特别的有用。最开始的版被认为是主系统，这以后的版就被认为是三角洲到版 1 的子集。

第三个可以控制文件差异的就是用**环境编辑**。也就是说，一个单独的代码组件代表了所有的版本。环境的状态用编译器来决定哪一种状态适合于哪一种版本。由于共享代码只能应用一次，我们在应用所有的版本时可以做一次更正。但是，如果在版本中的变量非常的复杂，源代码就会很难阅读和理解。甚至，对大量的版本来说，环将编辑会变得无法控制。

环境编辑只能代表编码。然而，分离文件和三角洲不但在控制代码上有用处，而且在控制其它开发的产品上，如需求，设计，测试数据和文献方面。工具栏 9.2 说明了三角洲和分离文件在组织和改变大型系统中时如何有用的。

工具栏 9.2　三角洲和分离文件

源代码控制系统，利用 AT&T's UNIX 的大多数版本进行传播，目的是打算控制一个工程的软件界限。它也能够用于其它相关的工程文献，并且它们是以文本的形式出现。利用三角洲，SCCS 容许多个版本和发布，同时一个程序人员可以要求在一个给定的时间内来自系统中的任何一个版本或是发布。基准系统是用转换方式存储的。那也就是说，对一个给定的组件，SCCS 在一个文件中存储，也就是那个组件版本 1.0 的基准代码，而把它转换成版本 2.0 的是三角洲，同时三角洲也把它转换成版本 3.0。同样，SCCS 可以存储各种的发布，或者是版本和发布的结合体。这样，任何给定的发布或是版本通常是可用的也可以改变；SCCS 仅仅是应用合适的三角洲把它从基准中导出来。但是，对中间的版本或是发布进行改变可能会引起问题，因为接下来用于版本或是发布的三角洲是基于上一个版本内容的。另一方面，在处理多个版本和发布时的SCCS的可变性意味着一个可以用SCCS的出口来同时的支持许多的版本和发布。

一个程序员在被 SCCS 生成的版本或是发布时需要用到"get"命令。如果程序人员指明一个"-e"代表编辑过的组件，SCCS 就会把所有以后的用户的组件锁住，直到被改变了的组件检测以后才行。

Ada 语言系统(ALS)是一个程序环境，它是用配置管理设计的，用来作为一个主要的设计因素(Babich 1986)。它并不包括一个特殊的配置管理策略。相反。它含有类似于 UNIX 的命令，用于支持配置管理工具。它不像 SCCS，ALS 作为分离的，不同的存储这些修订。此外，ALS 出了当时的以外，固化所有的修订和发布。也就是说，一旦一个新的修订版适用于用户后，以前的修订版就不会改变了。

ALS 允许收集所有的相关的修订版并把它们分组形成变量集。这些变量可以基于一个产品版本加上几个开发版，或者是带有几个连续发布的版本。ALS 也利用属性信息来标志每一个文件，比如创建日期，持有者名字，最后测试日期，甚至是那些文件的目的也是这样。这个系统也对关联进行跟踪，为的是在系统中的所有文件或者是变量集中的所有文件能够进行识别。

对 ALS 的控制模式来说，包括锁住命名的人，这些人容许读，覆盖，添加，或是执行文件中的数据。系统也为那些访问或是与文件交互的某类工具指定了许可。

修改控制。结构管理小组和测试小组紧密联系工作，控制所有方面的测试。在系统的任何一部分上提出的任何改动都要首先得到结构管理小组的同意。所有的组件和文档都要做改动，小组通知所有受影响的人员。例如，如果测试导致了需求的改变，那么需求描述，系统设计，程序设计，代码，所有相关的文档，甚至测试计划本身都可能需要改变。因此，改变系统的一部分，可能影响到系统开发中的所有人。

当多于一个开发者在同一组件上作过改动后，修改控制变得更加复杂。例如，假想测试时出现了两个失败。Jack 被委派找出并修正第一个失败的原因，Jill 被委派找出和修正第二个。虽然失败起初看作不相关，Jack 和 Jill 都发现错误的根源是代码组件中的初始化。Jack 可能从系统库中移走初始化，作些改动，把正确的版本放回库中。Jill 从最初的版本开始工作，做了她的修正，替换了 Jack 的修正，从而取消了 Jack 的工作。复原测试可能发现 Jack 的委派的错误仍存在，但努力和时间已经浪费了。

为了定位问题，结构管理小组进行修改控制。小组监视库中的代码和文档，开发者必须在作修改时检查拷贝。在我们这个例子中，Jill 只有 Jack 用正确的测试过的版本替代旧版本后才能拿到初始化的拷贝。或者结构管理小组会做一些额外的工作，将 Jack 和 Jill 的版本完善成一个版本；完善后的版本进行复原测试，确定失败都清除了。

另外一个方法能够保证所有工程成员工作在最新的文档上，这就是让文档在线。屏幕显示文档，及时更新，我们会避免由于打印分发新的或修改页而造成的时间滞后。然而，结构

管理小组仍要在某种程度上维护修改控制，以保证文档的改变镜像到设计和代码中。我们仍不得不检查版本以便修改。如果某人在修改文档，我们会被告知一些文档不可用或加锁。工具栏 9。3 描述了微软公司在每天构造中合并改变前用私有源码拷贝允许开发者各自测试改变。

工具栏 9。3 微软的构造控制

Cusumano 和 Selby(1997)做过报告，微软的开发员必须在下午特定的时间把代码输入到产品数据库中。然后工程小组编译源码，在第二天早晨生成一个新的发展中产品构件。任何代有错误阻碍构件编译和运行的代码必须理解纠正。

构件过程本身分几个步骤。首先，开发者从中心存放主版本的地方检查自己的私有有码拷贝。接着，他或她修改修改私有拷贝来实现和改变特征。一旦做完改变，带有新的或改变特征的私有构件进行测试。当测试成功完毕时，具有新的或修改特征的代码放到主版本处。最后，复原测试保证开发者的改变没有无意间影响到其他功能。

个别的开发者可能在必需的时候联合他们的改变（有时每天，有时每个礼拜，看需要情况），但一个构件主管要每天用主版本源码生成一个完整版本。对于每个产品和市场每天的构件都是必需的。

测试小组

正如我们看到的，开发者要对功能和性能测试负首要责任，但是用户在接受性和安装测试中起更大的作用。然而，进行所有测试的测试小组却不在这两组成员中。经常，没有工程中的程序员深入到系统测试中；他们太熟悉结构的实现和意图了，他们可能在接受实现和需求功能和性能上的差异有困难。

因此，测试小组经常独立于实现的小组。理论上，一些测试小组的成员已经是有经验的测试者。通常，这些“职业”的测试者是前任的分析员，程序员，设计者，现在将所有时间投身于系统测试。测试者不仅对系统描述熟悉，还精通测试方法和工具。

职业测试员组织和进行测试。他们随着工程的进展从开始到设计测试计划和测试用例一直插手。职业测试员与结构管理小组合作，提供文档和连接需求，设计组件和编码的其他机制。

职业测试员集中精力在测试开发，方法和过程。因为测试者可能没有撰写需求人员那样的文采，测试小组要包括熟悉需求附加人员。**分析员**插手于原始需求定义和描述，他们在测试中是有用的，因为他们懂得客户定义的问题。许多系统测试比较新系统和原始需求，分析员很好的掌握了用户的需要和目标。虽然他们和设计者一起工作，寻求办法，分析员对于系统应该怎样解决问题还是有一定想法的。

系统设计者为测试小组增加了目的观点（不懂）。设计者明白我们解决方案的提议和方案限制。他们也明白系统如何分成功能或数据相关的子系统，系统怎样工作。当设计测试用例和保证测试覆盖时，测试小组邀请设计者帮助列出所有可能。

因为测试和测试用例于需求和设计紧密联系，测试小组需要一个**结构管理代表**。当出现失败或需要改变时，结构管理专家安排影响到文档，需求，设计，编码，其他开发产品的变化。实际上，纠正错误的改变可能导致其他测试用例或大部分测试计划的修改。结构管理专家执行这些修改，协调测试的修正。

最后，测试小组需要用户。他们经过良好的训练，评估界面，易用和其他人为因素的恰当性。有时，用户在工程的初始阶段很少参与。参与需求分析的用户代表可能不会用到此系统，但会于用此系统人员有业务往来。例如，代表可能是用到此系统人员的管理者或是发现问题与本职工作有非直接问题的技术代表。然而，这些代表可能远离实际问题，以致需求描

述是不准确的或不完整的。用户可能没有意识到重新定义或加需求的必要。

因此，提议系统的用户是重要的，特别是如果系统需求首次定义是他们没在场。用户可能因为平日的接触熟悉问题，在评估系统，证实其解决问题上是无价的。

9.2 功能测试

系统测试由功能测试开始。鉴于以前的测试集中在组件和交互上，系统测试第一步忽略系统结构，集中在功能上。我们从现在起更多地用倒闭盒，而不是开盒。我们不必知道那个组件正在执行；相反，我们必须知道系统假定要做什么。因此，功能测试是基于系统功能需求的。

目的和任务

每个功能是通过联合系统组件来完成的。对于一些功能来说，部分可能包含在整个系统。一组关联于某功能的动作叫做一个**线程**，因此功能测试有时叫做**线程测试**。

逻辑上，在小组的组件中查找错误的原因要比大组更容易。因此，功能测试的顺序要求谨慎选择，这会给测试带来方便。功能可能定义成嵌套方式，就像旋转定义成级别。例如，假想一个需求描述了一个水位监测系统要在 4 个特征上识别大的变化：溶解的氧气，温度，酸度，放射能。需求描述可能把变化确认作为整个系统许多功能的一个功能。然而，为了测试，我们可能把监测看成四个独立的功能：

确认溶解氧的变化

确认温度的变化

确认酸度的变化

确认放射能的变化

然后，我们逐个测试每一个。

有效的功能测试应该很有可能检测出错误。我们运用在单元测试中用到的同一方针测试功能。也就是说，一个测试应该

很有可能检测出错误

用一个独立于设计者和程序员的测试小组

了解期望的动作和输出

测试合法不合法的输入

不为测试容易而修改系统

有测试停止标准

功能测试应该在谨慎的可控制的条件下进行。而且，由于我们一次测试一个功能，实际上功能测试可以在这个系统构造完之前开始，如果有必要的话。

功能测试比较系统的实际性能和需求，因此功能测试的测试用例由需求文档开发而成。例如，一个字处理系统能够通过检查系统处理

文档创建

文档修改

文档删除

方式进行测试。

在每个分类中，不同的功能接受测试。例如，可以考虑通过

增加字母

增加词

增加段落
删除词
删除段落
改变字体
改变类型大小
改变段落格式等等进行测试。

因果图

如果我们能够从需求中自动生成测试用例，测试会变得很容易。IBM 做的一些工作是转换原始需求定义语言到正式的描述，这些描述能够用来列举测试用例用来功能测试。生成的测试用例不是冗余的；也就是说，一个测试用例不在测试已经被其他用例测试过的功能。另外，转换过程还要找出需求中存在的不完整和模糊方面。

转换过程检查需求的语义，重申输入和输出或输入和转换的逻辑关系。输入叫做**原因**，输出和转换叫**结果**。结论是一个反映这些关系的布尔图，叫做**因果图**。

我们将信息加载到初始图中，这样就可以指定语法规则和反应环境的约束条件。然后我们将这个图转化为决策表。决策表的每一栏对应功能测试的一个测试。

创建一个因果图需要以下几个步骤，首先需划分需求，使得每一个需求，只定义一个功能。然后描述所有的条件和结果。这些条件和结果将成为图中的节点。将条件放在图的左边，结果放在图的右边。我们用符号的方式将图中的逻辑关系画出来。如图 9。4。额外的节点可以用来简化图形。

让我们通过一个例子来看如何建立这种图。假设我们在为一个防洪部门的水位检测系统进行测试。对于系统功能的一个需求定义如下：

系统通知水库管理员水库的水位安全性

对应这个需求以下是设计描述

输入：这个功能为 LEVEL(A,B)

A 是水库中的水位高度（单位米），B 是过去 24 小时的降雨量（单位厘米）

处理：计算水位是否在安全范围内，太高或是太低

输出：在屏幕上显示下列信息之一

1. ”LEVEL=SAFE ” 当结果是安全或低时
2. “LEVEL=HIGH” 当结果为高时
3. “INVALID SYNTAX”取决于计算的结果

我们将这些需求分成 5 个条件

1. 五个条件中的第一个为命令”LEVEL”
2. 指令包括两个参数，用逗号隔开，并用括号括起来
3. 参数 A，B 为实数，水位计算为 LOW
4. 参数 A，B 为实数，水位计算为 HIGH
5. 参数 A，B 为实数，水位计算为 SAFE

我们也可以描述三个结果

1. 消息”LEVEL=SAFE”显示在屏幕上
2. 消息”LEVEL=HIGH”显示在屏幕上
3. 消息”INVALID SYNTAX”打印输出

这些条件和结果将成为我们图中的节点。但是需要对参数进行检查，以确保它们合法。因此我们创建了两个中间节点

1. 指令语法正确
2. 操作数合法

我们可以在条件和结果之间画出它们的逻辑关系了，如图 9。5。注意，在结果的左边的虚线，这些线意味着同时只有一个结果发生。因果图中其他符号提供了额外的信息。图 9。6 解释了这些可能的情况。

因此，通过看图，我们能归纳以下几种情况

- 条件集 中有且仅有一个被激活
- 条件集 中至多只有一个被激活
- 条件集 中至少有一个被激活
- 一个结果掩盖了另一个结果
- 激活一个结果之前需要激活另一个结果

现在我们准备使用因果图来生成决策表。在表中为每个条件和结果生成一行表项。在我们的例子里，决策表需要 5 行条件的，3 行结果的。决策表的列对应测试情况，我们通过检查所有结果及所有条件的组合产生每个结果的情况生成列。

在我们的 LEVEL 例子中，我们可以通过检查流向结果的线来决定决策表的列的数目。在图 9。5 中，有两条独立的线指向 E3，每条线对应一列，有四条线指向 E1，但只有两条组合才产生结果，每一个组合就是表中的一列。最后只有一个线的组合指向 E2，因此我们产生了表的第 5 列。

决策表的每一列代表条件和结果的一个集合。当特定条件组合激活后，我们还需要跟踪其他条件的状态。当条件被激活或为 TRUE 时，我们指定这个条件的状态为 I,当未被激活或 FALSE 时，为 S。如果我们不关心这个条件是激活还是未被激活，我们用 X 表示“无所谓”状态。最后，我们指明是否一个特定的结果未产生（A）或产生（P）。

对于 LEVEL 功能的测试，表 9。2 的 5 列显示，条件激活和结果产生之间的关系。如果原因 1 和 2 为 TRUE,（也就是指令和参数都合法），那么，哪个条件产生就依赖于条件 3,4,5 的正确。如果条件 1 正确，但条件 2 不正确，结果就已经确定了，这是我们的并不用关心条件 3,4,5 的状态。同样，如果条件 1 不正确，我们就无需关心其他条件了。

表 9。2 因果图的决策表

	测试 1	测试 2	测试 3	测试 4	测试 5
条件 1	I	I	I	S	I
条件 2	I	I	I	X	S
条件 3	I	S	S	X	X
条件 4	S	I	S	X	X
条件 5	S	S	I	X	X
结果 1	P	P	A	A	A
结果 2	A	A	P	A	A
结果 3	A	A	A	P	P

注意，理论上我们可以产生 32 个测试用例，5 个条件每个条件 2 个状态，就有 2^5 个可能性，因此，我们可以使用因果图来减少测试用例的数目。

通常，通过我们对条件的理解可以进一步减少测试用例的数目。例如，如果测试用例的数目很多，我们可以给每个条件组合分配一个优先权，因此我们可以消去低优先权的组合。同样的，我们也可以消除那些出现机率很小的组合，使得在经济上更加合理。

除去能减少测试用例的数目，因果图还能帮助我们预测系统可能的结果，还能找出无法预测的副作用。但是因果图对系统来说，还不是完全实用的，因为它无法表示延时，迭代，循环。

9.3 性能的测试

一旦我们通过需求定义确定了系统执行的功能，我们开始致力于这些功能的实现。因此，功能测试强调的是功能需求，而性能测试强调非功能需求，

目的和任务

系统的性能测试对应着用户在非功能需求中对性能目标的描述，举个例子，功能测试是用来证明一个测试系统可以依据火箭的推力，气候状况和相关的传感器及系统信息，计算出火箭的轨迹。性能测试做的是这个计算做的有多么好，对用户命令的响应速度，结果的精确度，相对于用户性能描述的检查数据的访问程度。

性能测试由测试小组设计并管理，并将结果提交给用户。因为性能测试常常不但包括软件部分也同时包括硬件部分，因此硬件工程师也是测试小组的一部分。

性能测试的类型

性能测试基于需求，因此非功能需求的种类决定测试的类型。

//第 8 章翻译成压力测试

极限测试（Stress Tests）在系统在其极限停留短暂时间来估测系统性能。当需求阐明一个系统可以处理特定数目的设备和用户，极限测试是在测试所有这些设备和用户同时激活时的系统性能评估。这项测试对那些平时工作在最大容量值下但在出现峰值时负在极大的系统特别重要。

容量测试（Volume Tests）用来测试系统中大容量数据的处理。例如，我们检查是否数据结构（队列或堆栈）被定义的足够大，以处理各种可能的情况。另外，我们检查域，记录和文件来看是否它的大小可以容纳所有的数据。我们还应确定当数据集达到最大尺寸时，系统是否工作正常。

配置测试（Configuration Tests）分析在需求中定义的不同的软件和硬件配置。有时，系统被创建为多类用户服务。系统的配置比较复杂。例如，我们定义一个最小系统来为一个用户服务，其他的配置建立在这个最小配置之上来为其他用户服务。配置测试是用来测试配置的所有可能以确保每一条满足需求定义。

兼容性测试（Compatibility Tests）当一个系统需要于其它系统互操作时，就需要兼容性测试。我们检查是否接口执行了需求指定的功能。比如，如果系统和一个大数据库通信，兼容性测试检查检索数据的速度和精确性。

//回归测试第 8 章翻译成复查测试

回归测试（Regression Tests）当被测试的系统是用来代替现存系统时，就需要回归测试。回归测试保证新的系统的性能至少和老的一样好。回归测试总是使用在阶段开发中。

安全测试（Security Tests）安全测试保证安全需求。我们来测试相关于数据和服务的可用性，一致性和确定性系统性能。

时间测试（Timing Tests）时间测试评估处理响应用户和执行某项功能所需时间的需求。如果一个事务必须在一特定时间内发生，测试就执行那项事务并检查需求是否满足了。是检测试通常和极限测试合用来测试当系统在极限条件下时间需求是否满足。

环境测试（Environmental Tests）环境测试关注系统在安装点的执行能力。如果需求中包括了对温度，湿度，运动，是否有化学物品，便携性，电磁场，电击穿，及其他安装地环境特性，我们的测试就是保证系统在这些条件下能正常工作。

质量测试（Quality Tests）质量测试评估系统的可靠性，可维护性及可用性。这些测试包括计算发生错误的平均时间，恢复的平均时间及找到和改正错误的平均时间。对管理员来说，质量测试很难。例如，如果一个需求要求发生错误之间的间隔时间要长，有时让系统运行足够长时间来验证这个时几乎不可能的。

恢复测试（Recovery Tests）恢复测试关注数据丢失，掉电，设备出故障，服务不可用时的响应，我们让系统的资源受到破坏看它是否能恢复。

可维护性测试（Maintenance Tests）它强调对发现资源问题的诊断工具的需求。我们需要提供诊断程序，内存映像，业务跟踪，电路图和其他辅助工具。我们需确定辅助工具功能正常。

文档测试（Documentation Tests）它保证我们已经编写了所需的文档。因此，如果需要用户指南，维护指南和技术文档，我们能确认这些资料存在且内容一致，准确，易读。进一步说，有时需求还指定了文档的格式和用户，我们为应用来评估文档。

人为因素测试（Human factors Tests）它用来测试系统的用户接口需求，我们验证屏幕显示，消息，报告格式和易用性的其他方面。另外，用户程序也检测是否它们满足使用需求。这些测试有时又称为**可用性测试**（Usability Tests）。

以上测试中有很多测试都比功能测试困难。需求必须详细具体。需求质量常常反映在性能测试中。除非一个需求按照第 4 章定义是很明白，容易测试的，测试小组很难知道需求什么时候满足了，实际上如果测试是否成功没有很好的定义的话，很难执行一个测试。

9.4 可靠性，可用性和可维护性

性能测试一个最关键的问题是保证系统的可靠性，可用性及可维护性。因为在系统发布之前无法直接测量，这个测试特别困难。我们只能间接地评估系统可能的特征。由于这个原因，在这一部分我们仔细研究系统的可靠性，可用性和可维护性。

定义

为了了解我们所指的可靠性，可用性和可维护性，可以想想汽车。如果一辆车在大多数情况下都工作正常，我们就认为汽车性能可靠。我们知道，有些功能会停止工作，有些出问题部件需要修理或替换，但是我们希望一辆可靠的车在需要维修之前工作一段很长的时间。因此，在两次维修之间能正常工作的汽车就是稳定的。

稳定性指一段时间的行为，但可用性是描述在某一给定时间上发生的事。一辆汽车可用是指当你需要它时，可以使用它。这辆车可能已使用 20 年了，且只维修了 2 次，我们称这辆车具有很高的稳定性。但是恰巧它在修理铺中你却需要它，它仍是不可用的。因此有些东西有很高的稳定性，当在某一时刻仍不可用。

假设你的车即可靠又可用，但生产它的厂商已经关闭了。这意味着一旦你的车出了故障，维修商将很难找到可替换的部件。因此你的汽车在修好返回给你需要很长的时间，在这种情况下，你的车具有很低的可维护性。

这些概念同样可以用在软件系统上，我们希望我们的软件长时间工作一致和正确。当你需要它时它可用。当它出故障时能得到和很快很容易的修理。我们讲的**软件可靠性**是指系统在给定条件，一段时间内出故障的概率。我们用 0~1 之间的数表示可靠性。高可用性的系统的可靠值接近 1。不可靠的系统可靠值接近 0。可靠性是在执行时间上衡量的，而不是在真实时间（非时钟时间），这样它能更精确的反映系统的使用情况。

同样**软件的可用性**是在一特定时间点上，系统操作成功的概率。更正式的定义是：一个

功能在指定的时间点且外部资源都可用的情况下能完成的概率。一个系统总是能工作，可用性是 1，不能使用的系统可用性是 0。可用性是以时钟时间衡量的，而不是执行时间。

同样**软件可维护性**是指特定情况下，给定时间间隔和给定资源，过程下，维护活动完成的概率。同样它的范围也在 0 与 1 之间。和硬件维护有很大不同，硬件维护在执行时总是要求系统不可用，而软件维护在系统工作时也能做。

由于可靠性，可用性和可维护性是基于错误（故障）定义的，一旦系统完成切开始工作，就应该衡量这些指数。软件工程师总是要明确新产生的错误，也就是说，在判定可靠性时，我们只考虑新的故障，而不是我们已知但没有修正的故障。

另外，我们常为故障划分严重级别来描述它们对系统的影响。例如，美国军队标准 MIL-STD-1629A 划分了错误严重性的几种级别：

1. 灾难性的：错误将造成系统死机或丢失。
2. 致命的：错误能造成系统严重损害，会导致服务丢失。
3. 较严重的：错误能造成系统较小的损害，延时，丢失可用性或服务降级
4. 很小：错误没有损害系统，但需要额外的维护。

数据错误

在我们获得软件错误信息时，我们对软件本身做 3 个假设，特别的，我们假定当软件出现故障时，我们能找到故障根源并更正它。其实，这种更正本身就有可能引发新的错误。它们可能是以前没有出现过的故障现在无意产生的。在长时间运行中我们期望看到软件可靠性的提高（也就是说，我们希望两次故障之间的时间越长越好），但在短时间运行中，我们可能有时发现更短的故障间隔时间。

我们可以监视一个系统并纪录故障间隔时间来显示系统的可靠性是否提高。例如，在表 9。3 中我们列出了在一个命令控制系统的室内测试故障之间的执行时间（单位秒），这个测试都通过使用实际操作环境系统仿真（Musa 1979）来实现的。图 9。7 画出了这些数据可以明显看到长期稳定性的提高，因为错误间隔时间在增长。

表 9。3 故障时间间隔（按行从左往右读）

//此表全是类似数字，省略很多部分

3	30	113	81	115	9	2	91	112	15
138	50	77	24	108	88	670	120	26	114

注意在短时间内时间长度变化很大，这种情况甚至发生在数据集的末尾。这些数据告诉了我们的关于系统稳定性的什么内容呢？我们如何利用这些数据去测试到下一次故障发生时的时间长度？在我们能回答这些问题之前，我们需要理解不确定性。

在错误数据中都包含了可现的不确定。即使我们完全了解存在软件中的所有错误，我们也无法说出下一次错误将在什么时候发生。我们无法预测下一次故障是因为我们不能确知这个软件如何使用。我们不知道对这个软件输入和执行顺序。因此我们不能测试哪个错误会触发下一个故障，我们称这个为**第一型不确定性**，反映系统如何被使用的不确定性。因此在任一时间点，下一次故障发生都在不确定。我们可以将这个时间作随机数处理。

第二个模糊区域称为**第二型不确定性**，它反映了我们缺乏对错误去除产生的后果的全面认识。当我们更正一个错误我们无法知道是否我们的修正是完全的和成功的。甚至当我们的修正完一个错误时，我们都不知道在故障发生之间的时间长度是否会增加，换句话说我们不确定我们的修正能对软件的稳定性提高多大。

可靠性，可用性和可维护性的测量

我们希望能将可靠性，可用性和可维护性作为软件的参数来描述。测量值在 0（不可靠，不可用，不可维护）和 1（完全可靠，总是可用，完全可维护）。为了得到这些测试值我们需检查错误数据的参数。假定我们正在记录错误数据且我们已经看到了 i—1 个错误。我们可以记录错误之间的时间或发生错误所需时间，设为 t1，t2…t i—1。这些值的平均就是**错误平均时间**。Mean Time to Failure**(MTTF)**

假定每个错误都被修正，系统又可开始运行。我们可以使用 Ti 表示下一次错误发生的时间，Ti 是个随机变量。当我们在对一个软件可靠性进行断言时，我们实际上是对 Ti 进行断言。

有几个其它的和时间有关的数据对计算可用性及可维护性很重要。一旦错误发生，就需要额外的时间来找到发生错误的原因并修正它。**平均修复时间（MTTR）**告诉我们修正软件组件的一个错误需要的时间平均值。我们可以将这个值与平均故障时间结合起来告诉我们一个系统会有多长时间不可用。也就是说，可以通过检查**错误之间的平均时间(MTBF)**来测量可用性。计算如下：

MTBF=MTTF+MTTR

一些业界人士和研究者主张通过这些平均值来测量可靠性，可用性和可维护性。例如，可以考虑，系统可靠性和平均错误时间的关系。当系统变的更加可靠时，平均错误时间应该增加。在测量过程中我们希望使用 MTTF。当 MTTF 很小时，可靠性它的值接近 0，MTTF 的值越来越大时，值接近 1。利用这个关系，我们可以定义系统的可靠性为：

R=MTTF/(HMTTF)

它在 0 到 1 之间取值，同样我们也可以推导可用性公式

A＝MTBF(HMTBF)

当软件可维护时，我们希望减少 MTTR，因此我们推导可维护性的公式

M＝1/(HMTTR)

其它的研究者通过使用其代理方法来描述可靠性。例如错误密度（也就是说：每千行代码中错误的个数或每个功能点上错误的个数）这是在他们不能直接计算出错误时采用的办法。有一些学者如 Voas 和 Friedman（1995）认为不是错误总数对软件可靠性起举足轻重作用的，而是一个系统作为一个整体隐藏未被发现的错误的能力。

可靠性的稳定和增长

我们希望我们的稳定性测试能告诉我们当我们找到并改正错误之后是否软件的**稳定性在提高**（也就是说发生更少的错误）。如果的确提高了，我们就有了**稳定性的增长**。但是预测一个系统什么时候失效是很困难的。从工具条 9.4 中可以看到。对于硬件的预测要比软件的相对容易。硬件故障是几率性的，我们无法知道出错误的确切时间，但我们可以说一个硬件在一个给定的时间段内可能会出问题。例如，如果我们知道一个轮胎 10 年就会损坏，我们就知道在第 3653 天时轮胎损坏的概率为 0 到 1。当我们买新轮胎时出故障的概率从 0 开始慢慢增长，当我们使用了 10 年之后，出故障的概率就接近 1。我们可以画出这段时间出故障可能性的增长图，曲线的形状取决于轮胎的原料，设计和我们在何种路面驾驶，汽车的含量等等很多原因。我们使用这些参数对出故障概率来建模。

当我们通过定义时间的函数，**概率密度函数** f（t）来对软件错误建模时，我们使用类似的方法。例如，假设我们知道软件的某个组件在下一个 24 小时的某一时刻会出现问题（可能是偶然的缓冲溢出）若在任何一小时间隔内发生错误的概率都相同。24 小时共有 86400

秒，如果我们以秒来计算概率密度函数就是在 0－－86400 中的任一秒的概率都是 1/86400，对于大于 86400 秒的 t 时间，概率为 0。我们在时间段 t＝0 到 86400 中概率密度函数呈**一致分布**。见图 9.8。

工具条 9.4 硬件和软件稳定性的区别

Meuor(1992)解释了为什么硬件故障和软件故障本质不同.当一个组件出了问题不能正常工作了,这个复杂的硬件就失效了,举个例子可以停在 0 或 1,或短路时的一个电阻,造成这个问题的原因是物理性的(比如:腐蚀或氧化)故障在特定的时间点发生,为了修理故障需要或更换一些部件,系统就能回复到以前的状态.

软件故障却可以在系统中长期存在,只有在特定的条件下,错误在特定条件下错误才会被激活产生故障软件错误是潜在的而且系统可能一直出错(在相同的条件下)除非改善软件的设置以更正那个错误.

由于错误产生的结果不同,软件可靠性定义必须和硬件的不同.软件可靠性定义必须和硬件的不同.当硬件修复后,它能达到以前的稳定状态,硬件的可靠性也就恢复了,但当一个软件被修复后,它的可靠性可能提高也可能降低.因此硬件工程可靠性目标是稳定,而软件可靠性目标是可靠性增加.

但并不是每一个密度函数都是一致分布，要使用合适的概率密度函数来解释和测量可靠性是很难困难的一件事。因此我们应定义一个函数 f（t），用它来计算在给定时间[t1,t2]之间出故障的可能性。因此这个可能性是在曲线起始点和结束点之间的下面面积，那么在时间 t1 和 t2 之间出现故障的概率为

//公式 408 页

定义**分布函数** F(t)为从 0 到 t 的积分.F(t)是软件在时间 t 之间出故障的概率.我们定义**可靠函数** R(t)为 1-Fi(t).它是软件一直工作到时间 t 仍正常的概率

稳定性预测

我们可以利用故障和故障时间的历史信息来建立一个对可靠性预测的简单模型.例如,使用 Musa 的数据(见表 9.3),通过平均以前两次故障时间预测第三次故障的方法来预测下一次故障的时间,从表中我们观察到 t1=1 和 t2=30.因此我们预测下一次发生故障的时间 T3 是 32/2=15.5.我们可以对每个观察值 ti 继续这样的计算,因此我们可以得到

当 i=4 时 t2=30,t3=113 则 T4=71.5

当 i=5 时 t3=113,t4=81 则 T5=97

如此计算下去,我们就得到了图 9.9 标为”av2”的灰色曲线,我么可以使用更多的历史数据来扩展这个技术.图中曲线”av10”是使用了前 10 个故障时间,”av20”是使用了前 20 个.

研究人员提出了更加复杂的方法来描述当我们找到并修改了错误之后我们对软件行为的假设.例如,有些模型假定修正一个错误和修正另一个错误系统行为的变化是一样的,但其实模型认为错误是不同的,对系统的影响也是不同的.我们可以对每一个修正得到不同的概率密度函数特别是稳定性在增长时.正如 Fenton 和 Pfleeger(1997)指出的.一个系统的预测度应包括三个因素:

- 预测模型:对随机过程进行完全的描述(例如:函数 Fi(Ti)和成功次数的独立假设)
- 参考过程:对于基于 t1,t2…….ti-1 的未知数的参数过程
- 预测过程:通过综合模型和参考过程对未来的错误行为作出预测

在这部分,我们研究两个常用的测试模型,对于其它的更多的模型和更详细的信息请参考 Fenton 和 Pfleeger(1997)

好的稳定性模型可以描述关于稳定性的两类不确定.第一类型不确定性是假定每个错误

发生都是随机性的.因此我们用指数分布来表示下一个错误发生的时间.例如,在摩托罗拉公司的零故障测试方法中(工具条 9.5)就使用了指数分布.因此我们可以通过他们处理第二型不确定性的方法来区分不同的可靠性模型.

工具条 9.5 摩托罗拉零故障测试

摩托罗拉使用一个成为零故障测试的简单模型.这个模型来源于错误率函数(Brettsvhneider 1989).模型认为错误产生的数量和时间的关系为

//公式 410 页

a,b 为常数,这个模型可以告诉我们为了达到可靠性目标,系统需要测试多少小时.模型需要三个输入:目标项目的平均故障数目,到目前为止测试发现故障的总数(test-failure)和到最后一个错误为止已经测试的时间(hours-to-last-failure)零故障测试时间的计算公式为:

//公式 410 页

例如:假定我们在测试一个 33000 行的程序在测试 500 小时后发现了 15 故障,在随后的 50 小时测试没有发现错误,你的目标是保证没千行程序出现 0.03 个错误.基于以上信息.3300 行程序出现的错误大致为 1.通过使用前面的公式,为达到目标需作测试的时间为

//公式 410 页

因此如果在发生一个错误后,你连续测试 77 小时内没有发现错误系统的可靠性要求就达到了.由于你已经测试了 50 小时,你仅还需要测试 27 个小时,但是如果在那 27 小时中发现了一个错误你就需要重新测试计算.

Jelinski-Moranda 模型.这个模型是最简单可能也是最有名的可靠性模型(Jelinski,Moranda1972)它假定不存在第二型不确定性.也就是说,这个模型认为问题的修正是很完美的(他们能修正错误且不会因此而引入新错误).同时它认为对任一个错误的改进对系统稳定性的贡献都是一样的.我们来看看 Jelinski-Moranda 模型是否真是地描述了故障假设我们的测试包含 15 个软件.更正每个错误都能使系统稳定性提高 0.003,.表 9.4 描述到第 I 次故障所经历的平均时间及错误次数的仿真集合(错误次数由模型的随机数产生)当它达到 15 时,故障间隔时间越来越大.换句话说,第二栏告诉我们基于历史信息的第 i 次故障的平均时间.第三栏告诉我们基于 Jelinski-Moranda 模型预测的时间.

常用的 **Musa** 模型就是建立在 Jelinski-Moranda 模型基础上的.通过执行时间获得错误间隔时间.当目标的稳定性达到后,它也引进日历时间来估算(Musa,Lannino 和 Okumoto 1990).Musa 模型将稳定性和项目管理联系起来.它鼓励管理人员将可靠性模型用于各种环境中特别是远程通信。

Littlewood 模型

Littlewood 模型比 Jelinski-Moranda 模型更实际一些,因为它认为每个错误更正对系统稳定性的贡献是随机独立的.这种贡献被认为具有 Jamma 分布特性. Littlewood 在他的模型中使用了两类不确定性,因此我们称他的模型为**双随机模型**(doubly stochastic) Littlewood 倾向于及早找到并更正对系统稳定性影响更大的错误.在错误发生时间中 Jelinski-Moranda 模型使用指数分布,而 Littlewood 使用 Pareto 分布.

表 9。4Jelinski-Moranda 模型的故障间隔时间

I	第 i 次平均时间	第 i 次模拟时间
1	22	11
2	24	41
3	26	13
4	28	4

5	30	30
6	33	77
7	37	11
8	42	64
9	48	54
10	56	34
11	67	183
12	83	83
13	111	17
14	167	190
15	333	436

操作环境的重要性

在第13章中，我们比较、对比了几种可靠性模型的准确性。设应用条件相同，这里假设过去模型准确则将来也准确。通常，预设是建立在测试中的失败的基础上。但我们的测试环境不一定反映实际的典型应用系统的使用。

用户什么时候有不同的系统使用、不同的实验级别、不同的操作环境就现实来讲是无法掌握的。如，初学表格或会计软件包者不可能象熟练者一样使用快捷方式；每一个人的错误模式都是唯一的。

Musa 预测典型的用户交互来发现问题，将其放入用户可能的**操作轮廓**中。理想地，操作轮廓是对输入的概率描述，反映概率分布。当测试策略基于操作轮廓时，测试数据反映几率的分布。

经常通过分割输入空间成不同的类并为其分配概率来产生操作轮廓。例如，一个程序允许你运行三个菜单选项：增、删、改，增;我们从用户测试中决定选择增操作是删，改操作的两倍。我们分配增0。5几率，删0.25，改0.25，随机选择输入就会得到该结果。

该**统计测试**方案有如下优点：

1. 测试集中于最有可能用到的系统部分，使得系统更可靠
2. 可靠性预测根据测试结果，能够量化

然而，做数据统计测试是不容易的，没有简单或者可重复的方法定义操作轮廓，后面将介绍cleanroom软件开发集成统计测试如何建立高质量软件。

9.5 接受性测试

当功能和性能测试完成，我们确信系统满足在软件开发的初始阶段所有需求，下面是咨询用户和用户

目的和规则

但目前，作为开发者我们设计了测试用例并监督所有测试。现在用户领导测试并定义测试用例。目的是使得用户和用户确定系统是否满足需要。于是，接受性测试由开发者回答用户技术问题，由客户完成。通常，用户的意见起十分重要的作用，因为他们知道要什么样的系统。

接受测试的种类

有三种方法评估系统。**基准测试**：用户准备典型的测试用例，这些用例代表系统要安装的环境，用户对于每个测试用例评估系统性能。BenchMark测试由实际用户或特殊小组训练系统功能。另一种情况，测试员熟悉需求能够评估系统实际性能。

BenchMark测试在用户有特殊需求时常用。两个或两个以上的开发小组根据需求描述开发系统；此系统在BenchMark测试下会选择出售。例如用户要求两个通讯公司安装语音数据网络，每个系统都是经过BenchMark测试的。两个系统都迎合需求，但一个比另一个更快或更早。用户确定选择哪一个是基于BenchMark标准的。

导航测试是在实验的基础上安装系统。用户训练系统就像永久安装一样。尽管BenchMark测试包括一组用户的特殊测试用例，导航测试依赖于每天的系统功能测试。用户经常准备每个用户要用到的功能清单。然而，导航测试没有BenchMark测试更正式化结构化。

有时，我们在系统发布到用户前在自己的公司和用户测试系统；我们在用户运行实际导航测试前导航系统。我们内部的测试叫做**alpha测试**，用户的导航测试叫**beta 测试**，这种方法在系统发布给多样化的用户是经常用到的。例如，一个操作系统的新版本在我们内部进行alpha测试，然后到一组特殊用户那里进行beta测试。我们选择beta测试的用户代表所有系统可能的用户。

即使一个系统只为了一个人开发，导航系统仍牵扯一小部分潜在的用户。我们选择用户，这些用户能够代表将来用到此系统的其他用户。选择一个地点或组织测试系统，而不是允许所有人参与。

如果一个新系统代替了以存在的系统，或是开发中的部分阶段，第三种方法用于接受性测试。**并行测试**：新系统于旧系统是并行工作的。用户逐渐习惯新系统，但继续用旧的复制新的。这种逐步转换允许用户比较和对比新系统和旧系统。它也允许持疑问观点的用户比较结果，对新系统树立信心，确信新系统于旧系统一样有效和高效。在某种意义上，并行测试结合了兼容性和功能性测试。

工具栏9。6 beta版的不适当用途

在1997年7月，美国NASA经历了带有寄居号的寻路者号在探测火星上遇到的问题。寻路者号的软件能够使其在火星着陆。脱离寄居者号，管理地球与登陆者号的通讯。然而，由于故障出现在栈，任务转换中的管理和指定，寻路者号定期重启，打断了定期的工作。

寄居者号包含一个简单的连续任务80C85控制器，它工作良好。但是NASA需要更复杂的软件管理更复杂的寻路者号功能。在设计期间，NASA首先选择了目标处理器，然后找到软件在其上运行。因此，NASA选择IBM新的R6000处理器，与PowerPC的处理器相同。32位处理器很有诱惑力，因为在其上的商用实时操作系统避免了开发自己软件的花销。这样，NASA的下一步就是为寻路者号选择一个操作系统。

有一些操作系统是可用的，NASA选择了Wind River System(Alameda,California)的VxWorks。当选定后，VxWorks在PowerPC上进行测试和商业应用。然而，为R6000准备的版本还没有到位。因此，Wind River System利用C语言的可移植性移植PowerPc的版本到R6000上，移植版本在1994年发布到NASA。

当R6000的VxWorks版本到了NASA,NASA停止了寻路者的配置版本到5。1。1，即使操作系统的重要毛病还没有解决。这样，寻路者号的软件实际上运行在beta测试版的操作系统上，而不是鲁棒的完全测试的系统。

接受性测试的结果

测试系统的方式和用户的爱好决定了接受性测试方法的选择。实际上，可以混合使用这些方法。接受测试可以让用户检验是否系统如合同的需求一样。换句话说，接受性测试是用户证实所要是否所得。如果用户满意的话，系统就可以象合同那样接受。

事实上，接受性测试不仅是需求差距分析，还允许用户决定它们真正想要什么，是否按照需求描述的那样。记住需求分析是客户解释问题需要什么办法，系统设计是我们建议的解决方案。直到客户和用户实际用建议的方案在系统上工作，用户才可能知道它们想要的是否已经被满足了。实际上，在我们的系统上工作可能帮助客户发现问题中我们没有意识到的特征。

在前面的章节中，快速的原型系统可能被用来在开发实施前帮助用户理解系统解决方案，但该原型系统常是不实际的或很昂贵。另外，在建立大系统的时候，往往存在初始描述与系统开始阶段观点的巨大差异，这时候，用户的需求可能会有所改变。例如，联合规划、主要人员，甚至用户的日常事务可能变化，同样影响原始问题的本质。因此，系统需求变化可能不仅由于开始没有给他们描述好，也可能它们又需要新的解决方案了。

在接受测试之后，用户告诉我们那些需求没有达到，那些由于需求变化要删掉，修改或增加。配置管理工作组识别变化，记录结果，开发、测试。

9.6 安装测试

最后一轮回测试包含安装系统测试，如果接受性测试在现场，就不用了。否则，还需要这一步。开始安装测试，我们在用户的环境中配置系统，建立和其它系统的通讯。定位文件，恰当分配函数和数据。

安装测试需要我们与用户一同工作来决定需要那些现场测试。回归测试来证实系统在实地安装工作正常。测试用例是用户确信系统完整并且所有的文件设备可用。测试集中于两件事：系统安装的完整性、核实功能的和非功能的性能不受现场影响。例如，安装在船上的系统必须测试来显示其不受恶劣天气和船行进的影响。

当用户满意以后，测试完成，系统交接成功。

9.7 自动系统测试

许多在第8章介绍的测试工具很有用，其它的被设计用来测试大的组件组或者辅助对软硬件的同时测试。

模拟仿真让我们专心于通过评估系统的一部分去描述系统的其他部分特征。一个**模拟仿真器**可以赋予一个系统的设备或者在没有可用设备的系统或可用系统的全部特征。正如一个飞行模拟器可以让你在没有实物飞机的情况下让你学会飞行，一个设备仿真器可以允许你再没有设备的情况下去控制设备，以上几种情况很常见，特别是当软件发展到可以脱离设备地点工作的时候和设备和软件并驾齐驱开发的时候。

举个例子，假设一个卖主正在建造一个包括软件和硬件的新的通信系统，与此同时软件工程师们正在编制这个通信系统的驱动程序。这时候测试这个卖主的还未完成设备是不可能的，所以我们可以利用设备的详细说明书来建造一个模拟仿真器，这样我们就可以测试期望的交互作用了。

同样地，当一个特殊的设备被建造在用户或者使用者这一地点，而测试在另一个地点

进行时，模拟仿真就显出其重要性了。进一步地，如果你正在建造一个汽车导航系统，你不太可用到实物汽车去测试软件，你可以让你的系统和汽车模拟仿真器交互，进行测试。实际上，有时候设备仿真器比设备本身还更有帮助，原因是在测试的每一阶段，仿真器可以存储显示设备状态的数据。当出现错误时仿真器可以报告出错信息，这就帮助你找到错误和错误产生的原因。

当测试系统必须和其他的系统接口时，仿真器还可以用来模仿系统的外部特征。如果消息通讯或者数据库访问，一个模拟仿真器就可以在不复制全部其他的系统的情况下为测试产生必要的信息。模拟仿真器还可以模拟重量和体积测试，因为它可以通过编程，装载大量的数据，请求信息和用户。

通常，模拟仿真器让测试脱离测试环境进行控制，这种控制允许你进行危险的或触及不到的测试。例如，通过使用模拟仿真器，导弹测试系统可以被构建得简单而且安全。

自动化还可以用来设计测试用例，例如，Cohenetral.（1996）描述了一个由Bellcore设计开发自动化测试仿真器（简写为AETG），可以用来使用组合设计技术产生测试用例，在他们的组合设计方法里，他们进行了一种测试，这种测试可以囊括全部对等，三个一组或者n-条组合的测试参数。例如，覆盖全部的参数组合，如果x_1是一个有效的参数值，x_2是另一个有效的参数值，有一个测试用例中，它的第一个参数是x_1, 第二个参数是 x_2。在一个测试，为了最后的版本，测试需要 75 个参数，和带有 10^{29} 个可能的测试组合。使用AETG，研究人员执行了仅 28 种测试覆盖所有的对等参数组合，在其他的试验当中，采用这种技术的测试相对于随机测试服从较好的块和决策覆盖。第三个研究表明自动系统显示了重大的需求和代码错误，这是在使用其他的测试方式所不会出现的。工具栏 9.7 描述了另一个如何自动化加速测试进程的例子。

工具条 9.7 发动机保险报价系统的自动测试

Mills（1997）描述了这个公司是如何使用自动化测试发动机保险报价系统。每一个系统包括了风险轮廓图，这个轮廓图有将近 90 个保险者和生产者，以及一个经纪人提供关于汽车和它的驾驶信息，接受一个保险报价图。输入的有 50 个域，包括年龄，驾驶经验，驾驶区域，使用类型，机车类型，驾驶员的数量。这些信息帮助把申请人放进 20 区之一，作为多于 20 个交通群体之一，五个使用类别，三种类型保险，15 个年龄群之一。报价系统跟踪 14 个保险系统的产物，每一个系统至少每月更新一次。

这样，需要彻底地测试报价系统的测试用例的数量非常庞大，而且测试过程的一个大的部分是决定到底有多少个测试用例才算足够，Bases（1997）产生计算表明在 National Westminisrer 银行为一个系统测试 5000 个环境需要 21，000 个脚本，由于每一个正本需要 3 分钟的时间手工测试，在一个平台上一个人需要花 7.5 个月的时间测试，对于 Mills 描述的保险系统这个环境是不可行的。因为它需要太多的环境和测试正本。发明家预测在分批模式中他们可以测试最多 100 到 200 个情形，保险公司指出开发者运行 100 个自由模式测试报价。但是通过使用自动化测试，三分之一的参与者每个月在自己的报价系统上运转每个客户的 30，000 计划测试用例。测试过程只花了少于一周的时间就全部完成。Mills 公布自动和人工测试的最大不同点除了速度外还有就是许多错误可以在测试进行中被发现，在系统发布的另一个版本之前留给了很多时间处理错误。

9.8 测试文档

测试是复杂和困难的。系统的软件和硬件可以导致困难，还有系统中使用的程序也会。除此之外，分布式或实时系统在跟踪和定时数据中需要格外的谨慎，得出性能结论。最后，当系统很大时，在发展和测试中所涉及到的人数如此众多，致使协调变得很复杂。为了控制

这种测试的复杂性和困难度，我们使用完整的认真设计的测试文档。

有如下几种类型的文档：**测试计划**文档：描述了系统本身和测试所有的函数和特征的计划；**测试说明和评估**文档：详细记述了每一个测试并定义了为评估每一个测试所具有的特征的标准；第三个：**测试描述**文档：记录了每一个单独的测试的数据和过程；最后一个：**测试分析报告**文档：描述了每一个测试的结果，图 9.10 描述了测试进程的几个文档之间的关系。

测试计划；

在第八章，在对所有可能的测试行为规划测试样式的时候，我们讲述了测试计划的任务，现在我们就看看一个测试计划是如何用来指导系统进行测试的。

图 9.11 出示了测试计划的组成部分。计划开始与规定它的目标状态 。包括：

- 指导测试的管理
- 支配测试中所需要的技术支持
- 建立测试计划和进度，包括制定所需的设备、组织的需要、测试方式、预期成果、使用这方向。
- 解释每一个测试的内涵和外延
- 解释测试是如何完全地评估系统的功能和性能
- 文件测试输入，详细的测试步骤，预期结果。

下一步，测试计划参考其他主要在开发过程中产生的文档。特别的，计划解释需求文档，设计文档，代码组件和文档，测试程序之间的关系。举个例子，假如有一个命名或者编号方式的图表，这个图表连接着所有得文档，所以需求 4。9 就反映在设计部分 5．32 节，5.6 节和 5.8 节被程序 12.3 所测试。

接下来，这些初步的部分就是一个系统的摘要。由于测试计划的读者可能没有参与前一阶段的设计，系统摘要把测试时间表和事件联系了起来。摘要不必太详细；它可以是一个轮廓，描述主系统的输入和输出，连同主要转换的描述。

一旦测试被放入了一个系统关系中，计划描述主测试和用到的测试方法。举个例子，测试计划辨别功能测试，性能测试，接受性测试，安装测试。如果性能测试根据某一标准可以被进一步分割（例如子系统测试），测试计划花费了测试的全部组织过程。

解释完了组件测试之后，计划居于事件的安排。安排包括测试场所和时间框架。通常被描述成阶段图表或者行为图。测试时间表包括

- 全面测试阶段
- 测试地进一步细分，以及开始和停止的次数
- 任何预期需求（比如定向或者熟悉系统，用户训练，或者测试数据的产生）和每一个测试的时间需要
- 准备和复查测试报告的时间需求

如果测试是在好几个地方进行的，每一个测试都有一个测试计划。一个图表说明了硬件、软件、和每一个地点进行测试的个人需求，以及每一个源测试需要的时间。还得纪录特殊训练和维护所学的东西。

根据可交付使用（例如：用户或者操作者，样品，磁带）和地点提供的素材（特殊测试设备，数据库工作平台，存储媒介），计划来鉴别测试素材。举个例子，如果测试是用于一个数据库的管理系统以建立一个样本数据库，测试需要在职使用者在测试组到达前定义数据元素。类似地，如果测试组需要某个安全或者保密措施，测试点的人员可能要在测试开始前需要建立密码或者特殊访问权限。

测试规范和评估

测试计划描述了如何将测试全面分解为针对特定单元的单元测试。例如，如果待测试系统分布在几台计算机上处理，那么功能和性能测试可以被进一步分解为针对每个分系统的单元测试。

对于每个单元测试，我们书写测试规范和评估。规范起于列示通过测试证明要满足的需求。参考需求文档，规范解释了测试的目的。

可以用图表来查看需求和测试之间的对应关系，例如表 9.5。注意需求通过在需求文档中的编号列示在顶端；需求要求的功能在左端，在需求对应列上放置 X。

TABLE9.5 Test-requirement Correspondence Chart

Test	Requirement2.4.1 Generante and Maintain Database	Requirement2.4.2 Selectively Retrieve Data	Requirement2.4.3 Produce Specialized Reports
1. Add new record	X		
2. Add field	X		
3. Change field	X		
4. Delete record	X		
5. Delete field	X		
6. Create index		X	
Retrieve record with			
A requested		X	
7. Cell number		X	
8. Water height		X	
9. Canopy height		X	
10. Ground cover		X	
11. Percolation rate			X
12. Print full Database			X
			X
13. Print directory			X
14. Print keywords			
15. Print simulation Summary			

测试中涉及的系统功能列在表中。性能测试可以使用类似的方法，取代功能需求，表中列出速度、数据库安全性等需求。

通常，单元测试是一组小的测试的集合，其数量表示了对需求的满足情况。从这个意义上讲，测试规范描述了小测试和需求之间的关系。

每个测试都在某种测试哲学的指导下，采用一套方法。然后，哲学和方法可能受限于其他的需求或测试条件这一事实。规范使这些测试条件明确。这些条件可能如下：

- 系统是否在使用实际的用户或设备输入，还是特定的由程序或代理生产的用例？

- 测试覆盖标准是什么？
- 数据如何被记录？
- 有没有时间、接口、设备、人员、数据库或其他的测试限制？
- 如果是一系列小的测试，他们应该以什么顺序进行？

如果测试数据在评估前被处理，测试规范讨论处理过程。例如，当一个系统产生大量的数据时，数据缩减技术通常被应用于输出以产生适于评估的结果。

伴随着每个测试的是如何得知测试结束的方法。因此，规范伴随着关于我们如何知道何时测试结束、相关需求被满足的讨论。例如，计划解释了哪一范围的输出满足需求。

评估手段在完成标准之后。例如，在测试中产生的数据可以被手工收集和比较，然后由测试小组检验。交替的，小组使用自动化工具来评估一些数据，然后形成报告或进行逐项的预期结果对比。Sidebar9.8 中描述的效率和有效性的度量可以应用于测试的全过程。

测试描述

对于每一个测试规范中定义的测试都要书写测试描述。我们用测试描述文档作为测试工作的向导。这些文档一定要详细、明确，包括：

- 控制方法
- 数据
- 过程

一个对测试的综合描述可以作为文档的开始，然后，我们定义测试是否被初始化并通过人工或手工方法来控制。例如，数据可能通过键盘手工输入，然后使用自动设备执行要测试的功能。或者是，整个过程都是自动化的。

工具栏 9.8 估量测试有效和效率

测试计划和报告的一个方面是估量测试效率。 Granham(1996) 建议测试效率可以通过测试方面发现测量除以全部错误数量(包括那些在试验之后发现)来衡量。例如，假定集成测试找到 56 个错误，并且测试过程的总数是 70 个错误。然后，Granham 的测试效率尺度是集成测试 80% 是有效的测试。但是，假定系统在 70 个错误被发现之后交付，并且 70 个另外错误在操作的前 6 个月期间发现。然后，集成测试要为找到 140 个错误中的 56 个，仅仅 40% 的测试效果负责。

这种评价特别测试阶段或者技术影响的方法可能用几种方式调整。例如，失败可能分配给严厉一级，并且测试效率可能以级别计算。用这种方法，集成测试可能是在挑出关键的失败的错误上效率为 50%，但是在挑出引起较小故障失败效率是 80%。或者，测试效率可能与根源分析相结合，因此我们要在尽早在开发内描述挑出错误效率。例如，集成测试可能发现错误 80%，但是那些错误的一半可能在早些时候发现的，例如在设计期间复查，因为那时他们可能在设计错误。

测试效率通过用错误数量除以执行测试付出的努力计算出来的，一人员小时有一个错误数量值。效率尺度帮助我们了解挑错的成本，和在测试的过程的不同的阶段过程中找到他们的有关的费用。

有效和效率尺度在测试计划过程中有用；我们希望基于过去测试的历史使我们的有效性和效率最大化。因此，当前测试的文档应该包括那些我们计算有效性和效率尺度。

那些测试数据可能被看成几个部分：输入数据，输入命令，输入状态，输出数据，输出状态，和通过系统生产的消息。每一个都是详细描述的。例如，输入命令提供给小组如何初始化测试，停住或者暂停测试，重复或者恢复一个不成功或者不完全的测试，或者结束试验。类似，小组必须解释消息，理解系统状态并且控制测试。我们要说明小组怎样区别起因于输入数据失败，从不恰当的测试过程，或者从硬件故障(只要有可能)。

例如，给排序程序测试可能是如下的测试数据：

输入数据

输入数据将由 LIST 程序提供。程序随机产生一个 N 个文字数字列表；每个词长度是 M。程序在测试驱动器中通过 RUN LIST(N，M)调用。输出放置在全局的 LISTBUF 数据区里。

这次测试使用的那些测试数据集如下：

Case 1:Use LIST with N=5,M=5

Case 2:Use LIST with N=10,M=5

Case 3:Use LIST with N=15,M=5

Case 4:Use LIST with N=50,M=10

Case 5:Use LIST with N=100,M=10

Case 6:Use LIST with N=150,M=10

输入命令：

排序程序由 RUN SORT(INBUF, OUTBUF)或 RUN SORT(INBUF)调用。

输出数据：

如果用到两个参数，排序后的列表放置在 OUTPBUF。否则，放置在 INBUF。

系统消息：

在排序过程中，显示了如下消息：

“Sorting ..Please wait..”

排序结束后，SORT 在屏幕上显示如下消息：

“Sort completed”

结束消息显示前，在键盘上，按 CONTROL-C，挂起和中断测试也会显示消息。

一个测试过程经常叫作一篇**测试脚本**，因为怎样执行试验，它给我们一个循序渐近的描述。一组严格定义的步骤给了我们驾驭测试的控制，因此如有必要，我们能够复制条件并且再创造失败，努力找到问题的原因。如果测试是被一些原因中断，我们必须能继续测试不用必须返回到开始。

例如，修改某个域的内容的函数（表 9。5 中列出）部分测试脚本可能是这样的：

Step N: Press function key 4:Access data file.

Step N+1: Screen will ask for the name of the data file. Type ‘sys:test.txt’

Step N+2:Menu will appear, reading

***delete file**

***modify file**

***rename file**

place cursor next to ‘modify file’ and press RETURN key.

Step N+3: Screen will ask for record number. Type ‘4017’.

Step N+4: Screen will fill with data fields for record 4017;

Record number:4017 X:0042 Y:0036

Soil type:clay Percolation:4 mtrs/hr

Vegetation:kudzu Canopy height:25 mtrs

Water table:12 mtrs Construct:outhouse

Maintenance code:3T/4F/9R

Step N+5:Press function key 9:modify

Step N+6:Entries on Screen will be highlighted. Move cursor to VEGETATION field. Type ‘grass’ over ‘kudzu’ and press RETURN key.

Step N+7:Entries on screen will no longer be highlighted. VEGETATION field should now read ‘grass’.

Step N+8:Press function key 16:Return to previous screen.

Step N+9:Menu will appear,reading

***delete file**

***modify file**

***rename file**

To verify that the modification has been recorded, place cursor next to 'modify file ' and press RETURN key.

Step N+10:Screen will ask for record number .Type '4017'

Step N+11:Screen will fill with data fields for record 4017:

Record number:4017 X:0042 Y:0036

Soil type:clay Percolation:4 mtrs/hr

Vegetation:grass Canopy height:25 mtrs

Water table:12 mtrs Construct:outhouse

Maintenance code:3T/4F/9R

那些测试脚本被编号，并且每个步骤相关的数据被旁注。如果我们没有在别处描述它们，我们说明怎样为测试准备那些数据或者场所。例如，设备需求设置，数据库定义，通讯连接要详细描述。下一步，脚本解释在测试期间将发生什么。我们报告键盘输入，屏幕显示，生成输出，设备反应，和任何其它显示。我们说明期望的结果或者输出，我们给操作者或者用户讲解如果期望结果不同于真实的结果做什么。

最后，测试描述说明所需行为顺序来结束测试。这些行为可以涉及读入或者打印关键的数据，终止自动化的程序，或者关闭设备的一部分。

测试分析报告

当测试已经执行，我们分析结果，确定是否测试的功能或者性能合乎要求。有时，一个功能简单示范是足够的。可是，大多数时间，性能限制了功能。例如，知道一个栏分类或者合计是不够的。我们必须衡量运算速度并且注意它的正确性。因此，一份测试分析报告对于几个原因是必要的：

它用文档记录了测试的结果。

如果发生失败，如有必要，报告提供复制失败需要的信息，定位和修正问题的源头。

它提供必要的信息，决定是否开发的工程是完整的。

它对系统性能建立信心。

测试分析报告可能被那些不参加测试过程但是熟悉其它方面系统和开发的一部分的人们阅读。 因此，报告包括工程，工程对象和相关参考的概要。例如，测试报告提到需求，设计和实现的文档，文档中描述了在测试过程中运行的函数。报告也注明该测试部分测试计划和说明文档。

一旦按这种方法确定测试阶段，测试分析提供在测试中要论证和描述的实际结果的功能清单和工作特性的报告。结果包括功能，性能，和数据评估，指出是否目标需求已经被满足。如果一个错误或者缺陷已经被发现，报告中讨论它的影响。有时，我们依据一个严格的尺度标准评估那些测验结果。这一措施帮助测试小组决定是否继续测试或者等待，直到错误已经被改正。例如，如果失败在显示屏幕的上部出现一种伪造的特性，当我们定位错误并且改正它时，测试能够继续。但是，如果一个错误导致系统崩溃或者数据文件被删去，测试小组可以决定中断测试，直到错误被修改。

问题报告报表

回忆第 1 章系统中的一个错误导致了系统的失败；开发者看见的是错误，用户经历的是失败。在测试期间，我们在**问题报告报表**中俘获大约错误和故障。**差异报告报表**是一个问题报告，当真实系统性能或者属性与我们期望的不匹配时，它描述问题的出现。它说明了什么是期望的，什么是实际上发生，和导致失败的情形。**错误报告报表**经常作为差异报告报表附属物，说明了一个错误怎样被发现并且修正的。

每个问题报告报表应该能够回答以下一些问题：

定位：问题何处发生

时间：问题何时发生

症状：发现了什么

最终结果：推论是什么

机制：问题怎样发生的

原因：为什么会发生问题

严重：影响到多少用户或事物

成本：花费多少

图 9。12 是一个英国工具中的实际错误报告的例子。注意到每个错误号码分给每个错误，开发者记录通知问题发生的日期和找到问题原因并且修正的日期。因为开发者不能立刻处理每个问题，(例如，因为其它问题有高优先权)，开发者还有记录修理某个特定错误需要的实际的小时数。

但是，错误报告报表丢失很多数据。通常，一张错误报告报表应该记录问题清单的详细信息(Fenton 和 Pfleeger 1997)：

定位：系统内部识别，诸如模块或文档名

时间：在某个错误产生，检测和纠正期间开发阶段。

症状：报告错误消息的类型或者显示错误（如测试，复查，检查）的行为

最终结果：由错误导致的失败

机制：源代码如果产生，删除和纠正。

原因：导致失败的人为错误类型

严重性：指导致和潜在失败的严重性

成本：定位和检查花费的时间或者努力；包括在开发早期鉴别出来的错误成本分析

注意，错误的每个方面反映出开发者对错误影响系统的理解。另一方面，差异报告应该反映出用户对由错误引发失败的看法。问题是相同的，但是答案确是非常不同：

定位：错误发现位置的安置

时间：CPU 时间，时钟，其他相关时间的度量

症状：错误消息和失败迹象的类型

最终结果：失败的描述，例如，操作系统崩溃，服务退化，数据丢失，错误输出和没有输出。

机制：事件链，包括键盘命令和状态数据导致的失败

原因：涉及可能导致失败的错误

严重性：在用户和事务上的影响

成本：修复花费加上丢失潜在事务的花费

图 9。13 实际是一个差异报告报表，错误的叫成错误报告。它指出在我们清单中的大多数问题和详细描述失败。但是，它仅仅包含清单中改变的那些项目，并非包含导致失败潜在原因的描述。理论上，这个报表应该旁注一份或更多错误报告，以便我们能说出哪个错误引

起哪个失败。

我们在问题报告报表需要更完整的信息，这样我们能够评价我们的测试和开发实践的有效和效率。 特别是当我们限制资源，问题报告捕捉的历史信息帮助理解哪些活动很可能引起错误， 哪个实践擅长发现和修正他们。

9.9 测试安全苛刻系统

在第 1 章，我们研究了几个系统出错就能使人们受害或死亡的例子。由于失败的结果是严重的，这样的系统叫做**安全苛刻**。Anthes(1997)汇报了许多其它的例子，软件的失败导致了不可接受的伤害。例如，从 1986 到 1996 年之间，450 件报告充满了美国食品药品管理局在医疗装置方面描述软件故障。这些报告中的 24 个涉及到导致死亡或者伤害的软件，在报告中的问题有如下几方面：

某一静脉内的药物泵枯竭，注射气体进入病人体内

当病人的心脏停止跳动时一个监控器没有正确发出的警报

某一呼吸器不定期的给病人输送氧气

数字显示某一病人名字和从其他病人而来的医疗数据

这些问题未必预示着软件质量下降。相反，它们反映了软件在安全苛刻系统下应用的增长趋势。

不幸的是，我们通常不清楚软件的发展过程如何影响我们构造的产品特征，因此，对于我们来说很难确保安全苛刻系统是足够安全的。尤其是，我们知道的很少关于软件可靠性的每一特征或者技术属性。与此同时，我们的客户要求我们达到更高水平或者超高可靠性。

例如，空中巴士 320 是一个通过金属线飞行的航空器，也就是说软件控制着它的绝大多数主要功能。因为飞行器不能承受它的软件故障，这一系统的软件可靠性要求是大约每小时 10^{-9}的失误率（Rouquet 和 Traverse 1986）。软件需求必须更严格。

规定的失误率是指系统在 10^9小时内最多能承受 1 个故障。换句话说，系统要在 100000 年至多失误一次。我们认为某一系统当它在 10^9小时内最多有一个故障是**超高可靠性**的。很明显，我们不能运用普通的评估工具评估这种情况的可靠性。因为这样做将意味着把系统至少运行 100，000 年，并且追踪故障行为。因此，我们必须寻找其它的实际办法以帮助评估软件可靠性。

图 9.14 指出了评估高可靠性软件的另一种方式。我们是从一个实际应用系统中得出图中的故障数据的，软件和硬件设计变化用来修正故障的原因。在图中，Littlewood－Verrall 模型是用作计算随着测试进行，在变化时间内，故障发生周期率（ROCOF）。由手工修正的虚线显示了一个清晰的返回减少规律，因为虚线范围变得平坦，表示故障率越来越小，也就是说我们必须测试一段更长时间以便系统再一次失误。因此可靠性会增加。

然而，我们不十分清楚最终可靠性将会怎样。因为当我们修正一个原始的故障时也同时引入了新的故障，我们也无法分辨是否曲线是渐近到零点或者到达一个非零可靠性状态。即使我们对系统有信心，能达到超高水平的可靠性，我们仍然需要测试直到非常长的时间直到我们的信心得以论证。

实际上，即使我们长期测试一个没有故障的程序，我们仍没有达到所需的保险水平。LittleWood 指出如果一个系统已经工作在无故障情况下 x 小时，那么下一个故障发生在 x 小时内的可能性存在大约是 50：50，为了维持这种信心，一个诸如 A320 的飞行器，它要求无故障软件性能达到几十亿小时（LittleWood 1991）。因此，即使系统达到它的预期的可靠性，我们仍不能保证它是在一个可接受的时期内是可靠的。

如果我们希望继续在安全苛刻系统内使用软件，确保比较高水平的可靠性是困难的但是

也是要面临的严峻挑战。许多软件工程师建议我们使用正规证实的需求，设计，编码方法，但是正规的评估自然语言是不可能的，并且如果我们将自然语言翻译成数学符号会有一些重要信息可能会丢失，即使正式的描述和设计证明不是极不牢固的，因为错误有时会在证明阶段出现。由于这个原因，像 Baltimore 电气公司（工具栏 9.9）依赖于质量评估方法的结合以极力达到或者阻止尽可能多的故障。工具条 9.10 列出了另外的增加系统安全工业方法。与此同时，研究人员已经寻找其它的方法以帮助发明者理解并保证可靠性，我们研究这些方法中的三个：设计差异、软件安全用例、cleanroom。

工具栏 9.9 软件质量练习在黑牒电气公司

在 Maryland，黑牒电气公司（BG&E）在其发展安全苛刻软件来控制它的两个核反应没有使用特殊的工具或技术。然而，管理者希望通过检查需求定义，质量复查，谨慎测试，完善文档，完整的配置控制来确保高可靠性。

为了保证所有的问题在早期发现，系统复查两次。信息系统组和核设计工程组进行设计评论，代码复查和系统测试。

美国政府已经为 Nuclear Power Plants 起草联邦质量评估标准，来自于 BG&E 的内部信息系统组的软件必须与这些规则一致。而且，当一个销售商提供软件，用作控制系统的一部分，BG&E 往销售商开发环境派遣了一个审计小组，以确保销售商有一个满足规则要求软件质量保证程序（Anthes 1997）。

设计差异

设计差异，在第 5 章中介绍过，是基于一个简单的哲学体系。同样的系统是按照相同的需求描述但是以几种独立的方式建立的，每一个都按照不同的设计方式。每一系统和其它系统以平行的方式运行，当一个系统的结果与其它的不同时，采取投票协调方式调节。根本的假设是不可能出现对于一个特定的需求，一组开发人员中的五分之三人员写出不正确的软件，所以说高可靠性是可能的（Avizienis 和 Kelly 1984）。几个系统已经用这个技术来建立了，包括美国航空航天飞机和空中巴士 A320 使用的软件（Rouquet 和 Traverse 1986）。然而，根据经验的例子指明独立的软件版本并不独立出现故障；多种多样的设计并不总是比单一版本提供更高的可靠性。

例如，Knight 和 Leveson（1986）做了一个实验，在每 27 个软件系统独立开发版本中。他们实验发现了每一个系统故障，并且发现一个普通情况的高事故率。Knight 和 Leveson 推测出，因为我们训练软件开发者使用普通的方法和技术进行设计。我们能预期不同的软件设计者和开发者可能犯同种错误。Eckhardt 和 Lee（1985）讨论一个理论化的想法，就是基于多种不同的难度的输入概念的，用于支持实验的发现。

Miller（1986）指出，即使我们建立多余的故障独立系统，我们必须仍旧尽力去估计在任何两个版本之间的依赖性。他还指出这一示范同用黑盒测试结论系统面临一样的困难，并且声称这是本质上不可能的任务。

工具栏 9.10 对于建立安全周期软件的建议

Anthes （1997）建议对于构造和测试安全苛刻系统的几个步骤，如工业顾问提出的建议：

- 认识到测试不能清除所有的错误或冒险
- 不要分不清安全、可靠性和责任，一个系统 100%的可靠性仍可能既不安全也不保险
- 密切联系你的组织机构的软件和安全组织
- 建立并使用一个安全信息系统
- 滴注管理文化安全
- 假定用户将能就范每一个错误

• 不要假设低可靠性、高影响性的事件不会发生
• 强调需求定义，测试，代码和描述复查，结构控制
• 不允许因短期的花费考虑而影响长期的冒险和花费

软件安全用例

关联软件设计和执行测试是必须的。我们能检查设计帮助我们生成测试用例，考虑我们能想到的所有可能假设。Fenelon et al（1994）建议我们通过列出系统的目标研究安全苛刻系统的质量，调查如何能使设计满足这些目标并且确保这些实现与设计匹配。总体上，我们想使系统**安全**，也就是说，免受事故或者失败。我们能分解安全目标和分配故障率或者限制到每一个设计组件，以便满足每一个低级的目标，“积累”直到我们对整个系统满足安全目标。在这种方式下，我们为系统作一个**安全用例**，寻找以什么方式使我们的软件满足安全苛刻系统的性能目标。

我们能从 4 个不同的方面来分析一下系统：知道原因或不知道原因，知道结果或不知道结果。在每一种情况下，我们希望在导致正常的行为情况和那些导致潜在故障之间的建立联系。表 9.6 解释了在每一种情况下我们能采取的步骤。在设计期间使用的分析帮助我们计划避免故障的方法。在测试期间使用的分析帮助我们鉴别重要的测试用例。

表 9.6 安全分析观点

	已知原因	未知原因
已知影响	系统行为描述	包括故障树分析的演绎分析
未知影响	包括故障模式和影响分析的归纳分析	包括冒险和操作研究（operability studies）的探测分析

我们在第五章中可以看到分析故障树如何允许我们调查可能的结果并追溯到他们的可能的根本原因。**故障模型和有效分析（FMEA）**通过从已知的故障模型到未知的有效系统分析补足故障树。我们说一个**冒险**是一个系统状态，伴随正确的情况将导致某一事故，一个**故障代码**是一个将给出危险的状态。例如，在 Ariane－4 SRI 中的溢出是一次危险；它不能导致 Ariane－4 达到失败，因为起作用的条件没发生（但是他们确实在 Ariane－5 起作用了）。对于 Ariane－5 的故障代码是在 SRI 运行的比它设计之处预期的时间长的情况下产生的。

FMEA 是高度劳动密集型并且基于分析家的经验。它通常涉及我们的软件设计进行原始分析，概要可能导致故障的模型。我们研究如何结合基本的故障模型才可能导致实际的故障。工具栏 9.11 描述了如何 FMEA，与其它的技术一起，可能改善 Therac－25 的安全。

工具栏 9.11 安全和 therac－25

在 1985 年 6 月和 1987 年 1 月之间一个叫做 therac－25 放射性治疗法机器认定涉及 6 个事故，导致了大量死亡和严重的伤害。Leveson 和 Tumer（1993）很详细的描述了这一机器的事故和软件问题，对于设计和构建安全苛刻系统的软件工程师，应该阅读一下他们的文章。

软件是由单个人写的。用 PDP—11 汇编语言，在一台早期的叫做 Therac-6 的机器上编写。软件一部分在模拟器上测试，但是绝大多数是作为一个大系统的部分来进行系统集成测试。（也就是说微小单元测试和软件测试）。

加拿大限定的原子能（AECL）运行一个关于 Therac—25 系统的安全分析，AECL 开始用一个失败的模型和影响分析以便鉴别单一失败导致的危害。然而，为了鉴别多种失败和结果质量，它运行一个故障树分析系统。最终，它雇佣一个外来的顾问去运行与大多数严重危害相关功能的详细代码，检查电子束扫描，能量选择，束关闭，剂量标准。AECL 最终报告

了对于 Therac—25 硬件推荐的 10 个改变，包括能量选择和电子束扫描的软件控制后退的互锁。

Leveson 和 Tunner（1993）描述如果问题的原因是一个时间错误，重新产生是如何困难。他们指出，大多数与计算机相关的故障来自于需求故障，不是编码故障。并且他们列举了几个 Therac—25 违反的基本的软件工程原理：

- 文档应随着发展过程而撰写，不应推后
- 软件质量保证练习应作为发展进程的一个完整部分，这些练习应包括标准的早期设置，在评估中间产品时使用
- 简单设计，编码和测试比复杂的容易理解。
- 软件应该被设计成能预期故障并捕获他们的信息。
- 假想系统测试将捕获软件问题是不够的。软件应被广泛测试，正如在硬件集成之前到系统和组件级前的正式分析问题。

冒险和操作研究（HAZOPS）潜心于预期系统冒险一个结构化的分析并且建议避免或者处理它们的方法。他们是基于皇家化学的工业（UK）在十九世纪六十年代分析设计一个新的化学植物的技术发展而来的。HAZOPS 用向导词作为部分广泛复查过程，在运行组件期间，联合控制和数据流分析，帮助分析员鉴别冒险。表 9.7 提供了一个系统中的向导词例子，该系统中，由数据和信号控制的事件时间对于任务协调是重要的。

表 9。7HAZOP 向导词

向导词	意义
No	没有数据和控制信号发送和接受
More	数据容量过高或过快
Less	数据容量过低或过慢
Part of	数据和控制信号不完整
Other than	数据和控制信号由额外的组件
Early	信号对于系统时钟来的过早
Late	信号对于系统时钟来的过晚
Before	信号比预期队列来的早
After	信号比预期队列来的晚

Fenelon et al（1994）已经采纳了 HAZOPS，对于软件环境，被称作 SHARD 方法。他们将他们的向导词基于以下对冒险的三点看法：

1. 服务错：当软件不应该提供服务时它提供一个服务或者当它应该提供时而没提供：冗长/委任。
2. 即时错：服务提供的或者太快或者太慢：早/晚。
3. 值错：服务是不正确的并且它是或者容易看出错误或者相反：粗糙错/微妙错。

这一框架被延伸到一个巨大关键词设置上，将在表 9.8 中展示。

表 9。8SHARD 向导词

Flow		Provision		Failure Categorization Timing		Value	
Protocol	Type	Omission	Commission	Early	Late	Subtle	Coarse
Pool	Boolean	No update	Unwanted update	N/A	Old data	Stuck at…	N/A
	Value	No update	Unwanted update	N/A	Old data	Wrong tolerance	Out of tolerance

	Complex	No update	Unwanted update	N/A	Old data	Incorrect	Inconsistent
Channel	Boolean	No data	Extra data	Early	Late	Stuck at…	N/A
	Value	No data	Extra data	Early	Late	Wrong tolerance	Out of tolerance
	Complex	No data	Extra data	Early	Late	Incorrect	Inconsistent

一旦鉴别了一个故障模型，我们寻找可能的原因和结果。当我们发现一个有可能的原因和结果时，我们接着寻找策略来要么避免原因要么缓和一下影响。在测试阶段，我们能选择测试用例以实验每一个失败模型，以便我们能观测到系统的在运行是否正确反应（例如，不能导致灾难的失败）。

Cleanroom

在二十世纪八十年代中期，在 IBM 的研究者提出一个新的软件发展过程建议，为一个高效率的小组生产高质量的软件产品而设计。他们的过程叫 **cleanroom**，反映了在芯片生产上保持故障最小的思想。（Mills，Dyer，和 Linger 1987）

Cleanroom 原理和技术。Cleanroom 方法居于两个基本原理：

1. 证明软件描述，而不是等到单元测试时来找到故障
2. 产生零故障或者接近零故障的软件

这一原理被在这一章和前几章讨论的几种技术中混和运用。首先，软件明确的用盒子结构（在第五章介绍过）。系统被定义为一个黑盒，被提炼为一个状态盒，并且再次提炼为一个白盒。盒子结构在生命周期早期时是比较容易并且廉价修正的，它鼓励分析者去寻找一个冗长的需求。

下一步，白盒规范用自然语言或者以数学方式适合描述，转化到一个想要的功能。一个正确的定理定义一个关系，叙述了三分之一的正确条件，描述每一个关于控制结构的假定功能的正确性。

例如，对于普通结构的正确性能以问题报表叙述（Linger 无日期）

控制结构：	正确性情况：
序列	所有的争论
[f]	
DO	
g	Does g followed by h do f?
h	
OD	
Ifthenelse	
[f]	
IF p	Whenever p is true
THEN	dose g do f ,and
g	Whenever p is false
ELSE	dose h do f?
h	
FI	
Whiledo	
[f]	Is termination guaranteed ,and
WHILE p	Whenever p is true

```
DO                                  Does g followed by f do f ,and
 g                                  Whenever p is false
OD                                  dose doing nothing do f?
```

项目小组复查了这些关系并且证实了用正确性证明证明正常的条件。例如，一个程序和它的子证明可以从下面看出（Linger 无日期）

```
程序：                                              子证明：
[f1]                                              f1=[DO g1,g2,[f2] OD]?
DO
   g1
   g2
   [f2]                                        f2=[WHILE p1 DO   [f3] OD]?
  WHILE
   p1
  DO [f3]                                      f3=[DO g3;[f4];g8 OD]?
   g3
    [f4]                                    f4=[IF p2 THEN [f5] ELSE [f6] FI]?
   IF
    p2
   THEN [f5]                                 f5=[DO g4;g5 OD] ?
    g4
    g5
   ELSE [f6]                                f6=[DO g6,g7,OD]?
    g6
    g7
  FI
  g8
 OD
OD
```

这个证明代替了单元测试，这是不被允许的，在这一阶段，软件是通过规范证实的。

最后一步包括统计使用测试，正像我们在这一章前一部分看见的，测试用例是随机的，基于使用可能性的。测试结果是用在一个质量模型中来决定期待打算故障时间和其它的质量衡量尺度。尽管统计测试在改善软件可靠性方面更有效，在 IBM 的研究者感到传统的覆盖测试寻找故障是以随机的顺序。Cobb 和 Mills（1990）报告了统计使用测试在扩展 MTTF 有效性方面比覆盖测试超过了 20 倍。

Cleanroom的承诺 已经有许多经验评估Cleanroom了。例如，Linger和Spangler（1992）记录了首次 cleanroom 小组以高产率在 IBM 和别的地方已经编写了超过 300，000 行代码，其中故障率每千行代码大约有 2.9 个故障。他们宣布与传统代码开发的每一千行代码有 30 到 50 个故障相比，是一个急剧减少趋势。而且，“经验证明在传统的开发中，滞后于正确性验证而发现的错误在统计测试中变得容易发现和修正，但不容易碰到深层设计和接口的错误”（Linger 和 Spangler 1992）。报告结果是是基于 3 到 50 人的用基于过程和面向对象语言开发的许多钟程序的小组。

在 NASA Goddard 空间飞行中心的软件工程实验室将 cleanroom 用到一个严格的测试。它进行了一系列控制实验和事例研究以决定是否一些cleanroom方法的关键的元素能达到像预期的那样工作。我们能从表 9.9 看出结果，Cleanroom 看起来在小工程项目上工作的很好，

但是不是大项目。因此，SEL cleanroom 处理模型应运而生。尤其，目前的 SEL 处理模型被运用到少于 50，000 行代码但是正准备加大的工程项目中。另外 SEL 开发者不在使用可靠性模型和预知模型，因为他们有基于他们的项目的很少的数据。Basili 和 Green（1994）指出了这些在动态飞行环境做过的研究，已经证实了 cleanroom 一些关键的特点导致低的故障率、高的生产率，一组更完整并且一致的代码注释和一个开发工作重发布，然而，他们的警告大家，SEL 环境不同于 IBM，并且 cleanroom 一定在它使用的环境量体裁衣。

表 9。9NASA 的 SEL 研究结论(Basili 和 Green 1994)1996IEEE

特征	实验	用例研究 1	用例研究 2	用例研究 3
小组大小	三人开发小组（10 个实验小组，5 个的控制小组），普通的独立测试员	三人的开发小组；两人的测试小组	四人的开发小组；两人的测试小组	14 人的开发小组；4 人测试小组
工程大小和应用	1500 行的 Fortran 代码；大学实验室课程-消息系统的	40000 行的 Fortran 代码；动态飞行，地面支持系统	22000 行的 Fortran 代码；动态飞行和地面支持系统	160000 行的 Fortran 代码；动态飞行，地面支持系统
结论	Cleanroom 小组用少的计算机资源，更成功的满足了需求，预期发布高百分比	工程在设计下了大功夫，用了少的计算机资源，做到了比环境基准更高的生产力和可靠性	工程在维持基准的生产力，继续提高可靠性	工程可靠性仅强于基准，但生产力低于基准

关于 cleanroom 的警告。虽然在文章上有许多建议说 cleanroom 改善软件质量，但是 Beizer（1997）建议我们仔细阅读结论，他声称 cleanroom 的在单元测试上的缺陷促进了危险的营私舞弊。与“已知测试理论和常见感觉”相抵触。按照 Beizer 所说“你不能找到一个 bug 除非你执行的代码有 bug“，正交 cleanroom 只依靠于统计测试去证实可靠性，回避单元测试的任一种形式，而且，统计使用测试本身被误导，正如工具栏 9.12 的所示。

工具栏 9.12 统计使用测试什么时候误导

操作测试设想最显著的故障是在最频繁发生的操作和输入数据中。Kitchenham 和 Linkman(1997)指出这一假定，在一个特定的操作内是正确的，但是不能在整体系统中的操作是正确的。为了了解原因，他们描述了发送打印文件请求给 4 个打印机中的一个的例子。当请求被接受时，不是所有的打印者可以有效利用，三种情况可能发生：

1. 一个打印机可利用，没有内部的打印队列。这种情况称为非饱和条件
2. 没有打印机可利用，没有打印队列。一个内部队列一定要初始化，打印请求放到打印队列。这种情况称为过渡条件。
3. 一个打印机是可利用的，一个打印队列已经存在，并且打印请求存在于打印队列中。这种情况称为饱和条件

从过去的历史我们可以知道饱和情况占 79%时间，非饱和情况占 20%，过渡情况只占 1%。假设对于这三种情况的任意一种故障情况发生的可能性，同样是 0.001，那么每个模型对于全部故障中非饱和情况分配(0.001)*(0.20)或 0.0002 的可能性，对于饱和情况分配(0.001)*(0.79)或 0.00079，对于过渡情况分配(0.001)*(0.01)或 0.00001。假定我们有三个故障，每一个和每种情况相联系。Kitchenham 和 Linkman(1997)指出有 50%的机会检测到每一个故障，我们必须运行 0.5/0.0002=2500 个测试用例去测试非饱和情况下的故障，

0.5/0.00001=500000 个测试用例去测试过渡情况下的故障，0.5/0.00079=633 个测试用例去测试饱和情况下的故障。因此，按照此操作能检测出很多错误。

然而，他们指出过渡情况通常是最复杂和最容易出现故障的。例如，虽然起飞和着陆占有一个航空公司操作轮廓的很小比例。但是这些操作性模型占总故障的很大比例。因此，假定选择一个故障输入状态的原因对每一个饱和情况和非饱和情况是是小机率的，大约 0.001。但对于过渡情况是 0.1。每一个模型对于全部可能故障中非饱和情况分配(0.001)*(0.20)或 0.0002 的可能性，对于过渡情况是(0.001)*(0.79)或 0.00079，对于饱和情况(0.1)*(0.01)或 0.001。如前所示，转换成测试用例，检测非饱和情况需要 2500 个测试用例，检测饱和情况只需要 633 个测试用例，但检测过渡情况需要 500 个测试用例。换句话说，当我们实际应该专心于过渡情况测试时，用操作轮廓却专心于测试饱和模式。

Beizer 指出 cleanroom 从来不曾被很好的衡量。

- 合适的单元测试通过覆盖目标完成
- 测试由受过测试技能训练的软件工程师完成
- 测试在那些使用测试设计和测试自动化技术的组织来进行
- 合适的集成测试

此外，cleanroom 假设我们善于衡量软件的可靠性。然而，从 Lyu 的可靠性手册中总结出来的可靠性文献表明，在可靠性工程中存在很多的问题。特别是，正如我们所看到的，我们不能保证操作轮廓是精确而有意义的，而这对于好的模型是必需的。

Beizer 描述了 24 个评估 cleanroom 的实例研究的结果，包括 Basili 和 Green 的（1994），指出他们有一些致命的缺点。

- 课题参与者知道他们在参与实验，所以，Hawthorne 效应会影响结果。即，参与者知道他们的产品会被评估这一事实会导致质量上升，但这并不是 cleanroom 技术本身引起的。
- 测试者的控制组本身没有受过合理的测试方法的训练和经历。
- 没有使用覆盖工具或自动化工具来支持测试。
- 欺骗没有得到控制，因此，很有可能 cleanroom 已经被应用于调试代码。

通过对比，Beizer 指出没有任何研究表明现有的测试理论是数学上不完备的。既然他发现经验测试本身是不足以令人信服的，他建议对于以往的经验分析的回顾可以消除某些偏见（Vinter1996）。为了比较两种方法，我们可以使用第一种方法开发软件，在开发过程和实用的第一年里报告所有发现的错误。然后，第二种方法可以被应用于代码以观察它是不是能发现这些错误。相似的，一个用第二种方法开发的软件系统可以应用第一种方法来回顾。
这场争论的胜负还有待确定，然而，Beizer 提出一些重要的问题并引导我们去怀疑，不仅包括用于解决软件质量和生产率的软件工程方法，还包括用于证明一种方法优于另一种方法的评估技术。我们必须象重视软件一样重视我们的测试理论和假设。

9.10 信息系统举例

本章中所讨论的概念对于毕卡第利系统的开发者有实际意义。测试者必须选择一种方法来决定如何测试系统，当停止测试时，期望有多少错误和失败。这些问题不容易回答。例如，在文献中，哪种错误密度是可期望或可接受的并不清楚：

- Joyce(1989)公布美国航天局的航天飞机电子设备系统的错误密度为每千行代码有 0.1 个错误。
- Jones(1991)声称 leading-edag software 公司的错误密度为每千行 0.2，或每个函数点的用户反馈错误不大于 0.025。

- Musa,Iannino 和 Okumoto(1990)在一次可信度调查中发现重大系统中的错误密度平均为每千行 1.4。
- Cavano 和 LaMonica(1987)对军用系统调查表明其其错误密度在每千行 5.0－55.0 之间。

因此为错误密度或计算停在条件设定目标是相当困难的。

工具栏 9.13　为什么 SIX-SIGMA EFFORTS 不能应用于软件

当我们考虑高质量系统时，通常会使用硬件分析，以将一些成熟的硬件技术应用到软件上。但是 Binder(1997)解释了为什么有些硬件技术对软件并不适合。尤其，考虑到这样做，开发软件就得符合‘six sigma’质量限制的概念。加工部分在一定程度上要迁就它的设计目标。例如，想要一个重 45mg 东西，实际上 44.9998 到 45.0002 之间都可以接受。如果不在这个范围内，我们就说出现了错误。‘six sigma’质量限制是说在 10 亿个部分中，我们期望只能有 3.4 个在可接受的范围之外。（即每 10 亿个中有多于 3.4 个就是错误）产品中组成部件数量的增加，得到一个无缺陷系统的几率就随之下降。就像一个由 100 个部分组成的产品（每个部件都被设计成符合‘six sigma’限制）无故障的几率是 0.9997。针对这种在质量上的下降，我们可以减少产品组成部分的数量，减少每部分限制条件的数量，简化个独立部分之间的连接。

Binder 指出了这种硬件分析不适用于软件的三个原因：过程，特点和唯一性。首先，因为人们是可变的，软件过程从一部分到另外一个部分内在地包含着很大程度无法控制的变化。其次，软件或是符合，或是不符合，不存在某种程度上符合的概念。符合性是二元的，甚至不能和一个缺陷相联系起来。有时，多个缺陷共同组成了个错误，通常我们不能准确的知道一个系统中到底有多少个缺陷。而且，引起错误的原因可能是另外一个与之交互的应用程序（例如，一个外部系统给被测系统发了一个错误的消息）。第三，软件不是大规模生产的产品。“试图用相同的开发过程开发出数以千计的相同的软件组件是不可想象的，只有一些样品是一致的，并且然后，由于它产生了许多不符合需求的系统而修改它。我们能通过一个机械过程生成大量的拷贝，但这与我们说的软件故障并不相关……作为标语，‘six sigma’简单地意味着一些低层的缺点（主观上讲）.精确的数据意义被失去。（Binder 1997）

正如工具栏 9.13 中提到的，硬件分析有时不适合对质量作出判断。相反，毕卡第利系统的开发者从某些方面对已有的与之相似的系统的错误与失败记录进行检测是明智的，其中包括语言，函数，开发组成员和设计技术。他们能按照差错和失败行为设计一个模型，评定可能的风险，并且基于作为测试捕获的数据作出判断。

有许多因素于毕卡第利系统相关。就像广告的价格与播出时间，竞争强度，重复播出的次数等许多因素有关。因此要考虑许多不同的测试用例，自动测试工具对于产生和追踪测试用例及其结果是有用的。

Bach(1997)对于选择测试工具提出了几点建议。

- 容量：工具是否包含所需的全部关键内容，尤其是对测试结果的验证和对测试数据和教本的管理。
- 可靠性：工具是否可以长时间的无故障运行。
- 能力：工具是否能在大规模的工业的环境无故障工作。
- 易学性：工具是否可以在段时间内被学会。
- 易操作性：工具是否易于使用，它的特征是否繁琐难懂。
- 性能：在测试，开发，管理过程中，是否可以节省时间和费用。
- 兼容性：是否可以在你的开发环境中工作。
- 不可中断性：是否可以模仿现实中真正的用户。

每一条都提醒我们不要只依赖于用户手册中的描述，或商用逻辑中的函数描述。对于提

出的每一点，了解它是如何在我们的开发环境中工作，对于我们在一个真实工程中使用它是重要的。毕卡第利系统的开发者应该在他们的环境中评估若干工具并且选择一个能减轻他们对所有产生的测试用例过程乏味的工具。然而，没有一个工具能使他们轻松决定在区别不同测试用例时哪一个因素是重要的。

9.11 实时系统举例

在前面的章节中，我们看到了需求和调查不充分方面的问题造成的 Ariane-5 软件的失败，SRI。评估失败的委员会也考虑了那预防角色可能起到模拟作用。他们注意到在实际中分离和测试是不可能，然而，软件或硬件模拟能产生当一个转台提供了角运动时，被联系到预测的飞行参数的信号。假如这个方法在接受测试时被应用,调查者认为失败条件将被揭示。

另外，带有限制，测试与模拟在功能模拟设备上被应用

- 向导，导航，控制系统
- 传感器的冗余性
- 导弹每个阶段的功能
- 机载计算机软件顺从飞行控制系统

在这台设备上，工程师实施了完全的闭环模拟，包括地面操作，气流遥感勘测和动态发射。他们期望标定的弹道是正确的，也包括使用内部参数降级弹道，大气参数和设备失败。在这些测试期间，不能使用实际的 SRIs，相反，SRIs 被特别开发的软件所模拟。只有一些开环测试使用真正的 SRIs，像电子集成和通信适应。

调查者注意到

子系统的所有部分在给定的层次上的测试都在场并不是强制，即使更好。有时在物理上这是不可能的，有时不可能完全或是在某个特定方面测试他们。在这些情况下，在逻辑上，用模拟器来代替它们，但只有在确认以前的测试完全覆盖所有情况时.(Lions et al.1996）

实际上，这个调查报告中描述了两种 SRIs 能使用的方法。第一个是提供精确的模拟，但很昂贵。第二个便宜一些，但它的精度依赖于模拟的精度。但在两种情况下，电子设备和所有的软件都必须在实际环境中经过测试。

为什么 SRIs 没有使用闭环测试呢？首先，觉得 SRIs 在设备级已经被测试过了。其次，机载导航软件的精度依赖于 SRIs 的度量。然而，这个精度不能通过使用测试信号所获得；模拟失败模态被认为用一个模型会更好。最后，SRI 是基于一毫秒为单位操作的，但功能模拟设备是采用六毫秒为单位，这更降低了模拟的精度。

调查者发现了这些原因在技术上是有效。但他们也指出系统模拟测试的目的不止是验证接口，也要把系统作为对应于一个特定的应用的整体。他们得出结论：

重要设备（像 SRI）自身通过质量验证，或是以前在 Ariane－4 上使用过。这样的假定都有明确的风险。当模拟的高的精确性是合乎需要的时，在 FSF 系统测试它是明显的比在精确性上妥协但是完成所有的另外的目要好一些的,在他们之中例如 SRI 证明设备的合适的系统集成。指导系统的精度能有效地被分析和计算机模拟所表明。

9.12 本章对你的意义

本章讨论了许多有关软件测试的主要问题，包括与可靠性和安全相关的内容。作为一个单独的开发者，你应该有从系统生命周期开始就作测试的准备。在需求分析阶段，应该考虑能捕获状态信息和数据的函数，如果软件出现故障，这些函数将帮助你发现错误的根源。在

设计阶段，你应该使用故障树分析法，故障模式和作用分析，以及其他可以帮助避免错误或调节影响的方法。通过设计和代码复查，你能构造一个安全的测试用例是你和你的同事确信你们的软件具有高度的可信性并会生成一个安全的系统。在测试期间，应该十分注意所有可能的测试用例，适当的运用自动测试，保证你的设计遇到所有可能的危险。

9.13 本章对开发小组的意义

作为一个单独的开发者，你可以按照规格逐步的保证你对组件的设计，开发，测试工作。但是通常组件之间的接口会产生问题。集成测试对于测试组件之间的连接是有用的，但系统测试会发现更多的现实问题，有更多的机会让问题暴露出来。你的工作组在作这种测试时，应该保持通信信道的开通，使各种假设明确。你的小组必须仔细检查系统的二元条件和异常处理。

例如 cleanroom 技术在开发测试盒结构和设计实施数据测试时需要大量的小组计划与合作。可接受测试中的动作需要同用户紧密协作；当他们测试发现问题时，你必须快速的找出原因并改正以使测试可以继续进行。因此尽管开发的一些部分是单独的个人任务，但测试系统是一个需要协作的群体任务。

9.14 本章对研究者的意义

除了在此介绍的几种，还有许多测试方法。经验主义的研究还将继续进行，它将帮我们理解那种测试方法可以找出那种问题。经验工作与测试理论相结合会使我们的测试工作更有效率。

Hamlet(1992)提出几点建议供研究者参考：

- 在测试系统的可靠性时，将系统分块来指导测试方式并不比随机测试好。
- 我们需要更好的理解软件的可靠性意味着什么。状态爆炸（大量的状态产生了更大数目的测试用例）可能并不像我们想象的那么严重。我们可以对相关的状态进行分组，然后从分组中选择测试用例。
- 我们可以刻画那些测试用例的数量不是令人恐惧的程序与系统；我们说这些系统比那些测试用例数量大到几乎不可能的系统易于测试
- Voas 和 Miller(1995)提出了一种用来检测状态对数据变化敏感性的技术。在不敏感的状态中的错误可能难于通过测试来探测到。研究者必须调查敏感性与测试难度之间的关系

另外，研究者应该区别为纠错而进行的测试和为增加可靠性而进行的纠错。一些研究者（例如 Frankl et al.1997）已经论证了为第一种目的而进行的测试通常不能满足第二个目的。

9.15 小组计划

我们已经讨论了如何使设计与想象中的相同。把你的设计与你的同学，或是班上的其他组相比较。看看有那些相同，有那些不同。这些异同对于代码中错误的发现是否有利？

9.16 关键参考

IEEE 软件部分有几期专刊是关于我们本章中所讨论的内容。1992 年 6 月刊和 1995 年 5 月刊关于可靠性，1991 年 3 月刊重点讨论测试。

年度软件工程国际会议的学报中通常有在测试理论领域中的最新的优秀论文。例如，Frankl et al.(1997) examine the difference between tesing to improve reliability and testing to find faults.测试方面的优秀的参考书包括 Beizer(1990);Kaner,Falk 和 Nguyen(1993);Kit(1995)。每本中都基于工业经验提出了现实的观点。

有几家公司对测试工具进行了评价，并出版了对他们可靠性的概述。例如，Ovum Ltd. (info@ovum.mhs.compuserve.com) 提供了对几十种测试工具的二十多页详细的评价。这些工具从需求分析，有效性，计划与管理，模拟，测试开发，测试执行，范围分析，源代码分析，测试用例生成几个方面描述。

软件的可靠性和高安全性系统正越来越多的受到关注，有许多这方面的专著和文章，包括 Leveson (1996,1997)。另外，the Dependable Computing Systems Centre in the Department of Computer Science,University of York,UK 正在开发评价软件可靠性的技术与工具。你能从 John McDermid,at jam@minster.york.ac.uk 中得到更多的信息。

可用性测试也非常重要：一个正确的可靠的但难于使用的系统实际上比一个易用的不可靠系统更糟糕。可用性测试和更多的可用性问题在 Hix and Hartson(1993)中被深入的介绍。

9.17 习题

1。考虑一下汇编程序的开发。列其中的函数，想一想如何将所有的函数依次测试，在下一个函数被检查之前，测试好已知的函数。提出一个开发计划，并解释如何同时设计计划与测试。

2。认证是外界对系统正确性的一个认可。它通常通过与一个预先确定的性能标准相比较，来确定是否通过认证。例如，美国国防部经过大量的功能指标测试才完成对 Ada 编译器的认证。考虑一下本章的术语，这是函数测试，性能测试，接受测试，还是安装测试？解释一下是与不是的原因。

3。当你开发一个构造计划时，你必须考虑开发者与用户的可用资源，包括时间，材料和钱。举一个由于资源限制而影响到为系统开发定义的构件的数量。解释一下这些限制是如何影响到构造计划的。

4。假定一个具有计算直线斜率与截距功能的计算器。文档中定义的需求是这样描述的：“计算器应该能接收型如 Ax+By+C=0 的方程的输入，并能输出该直线的斜率与截距。”这个需求的系统实现为语法为 LINE(A,B,C)的函数 LINE，其中 A 和 B 为 x 和 y 的系数，C 是方程中的常量。结果用 D 和 E 输出，其中 D 代表斜率，E 代表截距。写出需求的因果集合，并画出相应的因果图。

5。第四章中我们讨论了需求可测试性的必要。解释一下为什么可测试性对于性能测试是必须的。并举例说明你的观点。

6。对于文字处理系统需要那种性能测试？工资系统，银行自动出纳系统，水量监测系统，电厂控制系统分别需要那种性能测试？

7。设计一个空中交通控制系统，用来向一个或多个用户提供服务。解释一下如何实现系统的多样配置，并列出如何设计一组配置测试。

8。在一个机场要安装一个导航系统，在进行安装测试时要考虑那些问题？

9。CNN 发布了一条新闻：如果一个软件特征被正确实现，就可以避免 1997 年 8 月在关岛发生的韩国 801 次航班坠毁的事件。考虑那种类型的测试能确保这个特征可以在关岛机场附近正确的位置正常工作？

10。给出一个不使用设备模拟就无法测试的例子，再给出一个需要系统模拟的例子

11。按照上面的问题评论图 9.15 中的差异表，通过读这张表我们能回答这些问题。

12。一个工资系统被设计成为每一名在该公司工作的人员保存一条雇员信息纪录。雇员一周工作的小时数被每周更新一次。每两周出一次简要报表，显示从会计年度开始每人工作时间小时数。每月，雇员的当月工资被电汇到他们的银行账户。对于本章中所描述的各种性能测试，看是否应该应用到该系统上？

13。Willie's Wellies PLC 曾经委托 Robusta Systems 开发一个基于计算机的系统，用来测试所有橡皮鞋类生产线的强度。Willie's 在世界各地有九个工厂，每个系统按照工厂的大小进行配置。解释一下为什么 Robusta 和 Willie's 应该在接受测试完成之后进行安装测试。

14。为测试本章中描述的 Level 函数，写一个测试脚本。

15。本章中,我们就无故障运行时间提出了可靠性度量，就俩次失败之间的时间提出了有效性，就修复时间提出了可维护性。这些度量与本章中定义的一致吗？也就是，如果可靠性，有效性和可维护性被以概率形式定义,我们会得到与我们用公制定义的结果相同吗？如果不同，它们能互相转换吗？或是二者之间有本质的不可调和的差别？

16。一个高安全性系统出现错误以至死了几条人命。当 调查事故原因时，发现测试计划没有考虑可以引起这个错误的测试用例。谁应该对死者负责：是没有注意这个用例遗失的测试者？是没有制定出完整测试计划的设计这？是没有核对这个计划的管理者？还是没有彻底进行可接受测试的用户？

17。如果一个有超高可靠性要求的系统意味着可靠性永远无法被验证，系统应该不管如何都被使用吗？

18。有时，用户雇佣一个独立的组织（与开发组织相隔离）进行独立的验证和确认（V&V)。V&V 的职员检查开发的每个方面，包括过程和产品，以确定最终产品的质量。如果雇佣了 V&V，但系统还是出现了灾难性的故障，谁应该对此负责：是管理者，V&V 小组，设计者，代码书写者，还是测试者？

19。本章中，我们介绍了两个函数：分布函数，F（t），和可靠性函数，R（t）。当我们测试安装系统时系统的可靠性提高，函数图将如何变化？

20。Sidebar9。3 描述了 VxWorks 软件的两个版本：一个为 68000 芯片设计，另一个为 R6000 芯片设计。解释一下与构建一个支持两种不同芯片的系统相关的配置管理问题。配置管理策略是否对于卖方从 68000 版本导入到 R6000 版本有利？

21。测试神谕是一个假设的人物或机器，他能判断什么时候实际结果与预计结果相一致时。解释一下开发测试理论包含测试神谕的必要性。

22。列出构建一个 Piccadilly 系统测试计划的大纲。

软件错误困扰关岛机场雷达系统

1997 8 10

Web posted at 10:34a.m.EDT(1434GMT)

AGANA,关岛(CNN)——调查者周日说曾经警告过上周由于飞行太低而在关岛坠毁的韩国喷气式飞机的雷达是被软件错误所干扰。这个名为 FFA Radar Minimum Safe Altitude Warning 的系统能向地面官员发出警告，他随即通知飞行员飞机飞行过低。但是调查坠毁事件的联邦机构说这个位于岛上的美国军方系统最近被修改过，一个错误插入到软件中。……调查员称这个错误作为事件的一个原因并不是微不足道的，它结束了 225 个人的生命，因为它应该能提醒飞行员将飞机提升到一个更高的高度。

可能被阻止

NTSB 成员 George Black 称“这不是一个理由——它完全可能被避免”。 在一个道路控制员告诉他们飞机坠毁之前没有收到警告后，调查员开始调查这个系统。联邦航空管理组织探测到这个错误

系统设计发出高度警告的范围是覆盖一个半径为 55 海里的圆形区域（102 公里）。然而，自从软件被修改后，系统仅仅覆盖圆周为一英里的区域。801 航班坠毁时不在这个范围之内。Black 说修改软件的目的是阻止系统发出过多的错误警告。同时他也提到修改的幅度太大了。还不能立刻清楚这个错误已存在了多长时间和自从修改后有多少航班降落在这个机场。调查者宣布由于 FFA 为全美范围内的机场提供这种软件，他们将调查时候还有其它的机场收到了影响。

航空公司为飞行员辩护

韩国方面称这个飞行员经验丰富，有足够的能力驾驶飞机.当韩国航空公司的官员为波音 747 飞行员辩护时传来软件故障的消息。同时有新闻可能由于是飞行远的失误。韩国方面发表声明：“Park Yong－chul 是一个有着 9，000 小时飞行经历的经验丰富的飞行员。”声明还出示了出事当周 Park 的飞行计划。在他的最后一次飞行之前，他休息了 32 小时 40 分钟。周三上午，韩国航班 801 坠毁在关岛国际机场附近的山腰上。机上共有 254 人，包括 23 名机组人员，有 29 人获救。调查者称在坠毁时飞行员完全控制飞机，同时他们正在检查山的高度和飞行记录以得出他为什么飞得这么底。

调查员称即使没有警告系统，飞行员手头上还有其它的指示告诉他飞机距离山腰太近。首席调查员说“是的，系统会有帮助，但那只是一部分，并不是向我们所知道的因为它而引起了这起事故。”

存在的其它问题

这个警报系统并不是机场中 FAA 设备的唯一的故障。“倾斜滑行”是将飞机引导到跑道的着陆指令的一部分，它不在固有的维护服务之内。航空公司已经声称他们清楚这一点。在提出的声明中，航空公司声称各种设备之间的连接问题和恶劣的天气都可能引起坠毁。“我们不能排除由于暴雨引起的高度突变，倾斜滑行故障，连接等其他因素引起事故的可能性。”周六，飞机在接近关岛机场跑道时滑行过头，但它设法稳住自身，在第二次尝试时安全着陆。还不清楚为什么第一次接近跑道时出现误差。

事故中的尸体回收被难以接近的岩石所，多丘的出事地点和碎片状态的尸体所阻碍。通讯记者 Jackie Shymanski，The Associated Press 和 Reuters 联合报道。

第 10 章 系统提交

本章我们关注

- **培训**
- **建立文档**

我们现在接近开发的结束了。前几章告诉我们怎样认识问题，设计解决方案并且实现它，测试它。现在我们准备把这个系统移交给用户，确保系统可以继续被恰当地使用。

许多软件开发人员把系统提交看成是一项手续–象是剪彩或者给出计算机的钥匙。然而，即使仅仅是有人管理的系统，提交也包含了不仅仅是把系统放到那里那么简单。在开发阶段我们帮助用户理解和感受我们的产品的可用性。如果系统提交是不成功的，用户就不能恰当地使用系统还会对系统的表现不满意。另外，用户并不象我们想象的那样有能力，我们小心建立的一个高质量的系统可能会是一种浪费。

10.1 培训

有两种人使用系统：用户和操作员。我们可以把他们看成是司机和机械之间的关系。汽车的主要功能是提供运输功能。司机使用汽车从一个地方走到另一个地方。然而，机械服务于或者支持汽车使它能够被驾驶。这个机械从不会驾驶汽车，但是如果没有机械使用的辅助功能，汽车什么也干不了。

同样，用户通过系统提供的功能来解决需求描述文档中描述的问题。因此，用户是为消费者解决问题的人。但是，一个支持主要系统功能的系统通常有一个辅助的任务。例如，一个辅助功能可能定义谁可以访问这个系统。另一个为在系统失败时能够恢复而定期产生重要文件的备份。这些辅助的功能常常不直接由用户进行。相反，我们的操作员进行这些工作来支持主要的工作。表 10。1 包含了用户和操作员的任务。

表 10.1 用户和操作员的作用

用户作用	操作员作用
操作数据文件	授予用户权限
模仿行为	授予文件权限
数据分析	进行备份
数据通信	安装新的设备
画图形和线条	安装新的软件
	恢复损坏的文件

培训的类型

有时，同一个人既是用户又是操作员。然而，用户和操作员的任务的目标具有很大的不同，因此对不同工作的培训侧重于系统的不同的方面。

用户培训

对用户的培训主要是系统的主要功能和用户访问他们需要的东西。例如，如果一个系统管理律师事务所的规定，用户必须必须在管理功能上被培训：产生或者查询文档，改变或者删除纪录，等等。除此之外，用户必须浏览纪录来访问其中特定的一个。如果信息被用密码进行保护了或者防止意外删除，用户还应该学习特定的保护功能。

同时，用户不需要了解系统的内部功能。他们可以对一个纪录集进行排序而不需要关心这个排序是外部排序、泡沫排序或者快速排序。用户在访问系统时不需要知道别的谁同时在使用这个系统或者信息被存放在那个磁盘里。因为这些是支持功能不是主要的功能，只有操作员才关心这个问题。

用户培训介绍系统的主要功能以便于用户理解系统的功能是什么以及怎样去执行它。培训与系统的功能现在是怎样运行的以及他们以后会怎样运行。做到这一点是困难的，因为用户常常被迫封闭自己熟悉的东西以便于学习新的知识。新旧活动的相似的但是细微的差别会阻止学习，所以我们必须好好设计我们的培训并把这个困难考虑进去。

操作员培训

操作员培训集中于对系统支持功能的熟悉；这种培训强调系统是怎样工作的，而不是系统是用来做什么的。这里，任务的冲突较小，除非系统与其他系统在操作员的工作上非常相似。

操作员常常需要在两个层次上进行培训:怎样调出并运行新的系统，以及怎样对用户进行支持。首先，操作员学习怎样配置系统，怎样授予或者回收系统的访问权限，怎样分配任务大小和磁盘空间，以及怎样监控并提高系统的性能。然后，操作人员集中在开发的系统的特定的方面：怎样恢复丢失的文件和文档，怎样与其他系统交互，怎样调用各种支持过程。

特殊的培训要求

用户和操作员常常被在系统使用上进行集中的完全的培训。常常是，培训从基本问题开始：怎样佩置按键，菜单选项的具体作用等；其他的功能被逐渐地介绍并且仔细地研究。这个完整的过程是在系统被提交给用户时进行的。

然而，新的用户可能会取代被培训过的用户，常常由于工作分配的变化。这是必须能进行相应的培训告诉他们系统是怎样工作的。

有时用户想复习他们在最初培训时忘记的东西。即使开始的培训是易于理解的，但是让用户或者操作员理解所有的东西是困难的。用户常常喜欢复习在最初几堂课中的东西。

你可以感谢通过记忆类似你第一门程序设计语言一样你需要复习一些东西。你能够学习所有的合法命令，但是你比其他人记得更多的代码和含义。为了掌握这门语言，你经常复习笔记看一下常用命令。

相似的问题会在遇到对系统不熟悉的人员时遇到。培训时获得的东西很容易被忘记，如果他们并不常用。例如，看一个大公司的文字处理系统。系统的主要用户可能是打字员，他们每天都敲文章并把他们从一个地方移到另一个地方。使用这个系统是打字员对系统的大部分功能非常熟悉。但是公司经理可能只使用它一两次来产生文档或者备忘录；这个文档然后被作为最后形式的文档存放了。对于不常使用系统的用户培训与标准用户的培训是不同的。这个经理不需要了解系统的所有的特殊性质；培训只集中在文档建立和修改功能。

操作员也可能遇到相同的困难。如果一个系统功能是每半年进行一次放到另一个磁盘上的存储，一些操作员可能忘记了要怎样做。在没有复习培训时，拥护和操作员倾向于进行他们认为好用的功能；他们不会使用那些使他们更有效率的功能。

类似地，特殊培训课程可以为那些有特殊要求的人而设计。如果系统产生图表或者报告，有些用户可能需要知道怎样产生图表和报告，而其他人只需要直到怎样使用现有的就行了。培训可以限于有限的系统功能或者仅仅复习整个系统的某个部分。

培训的辅助工具

培训可以以许多方式进行。不管培训是怎样提供的，它必须在任何时候向用户和操作员提供信息，而不是仅仅在系统开始被提交时。有时，如果用户忘记了怎样访问一个文件或者使用新的功能，培训必须包含能找到并学习这个功能的方法。

文档

每一个系统和培训支持都伴有正式文档。这些文档包括用户正确、有效地使用系统所需要的信息。文档在用户使用系统时是可以使用的，存在于分离的手册里或者网上。系统手册可能是对机械主人的手册是很熟悉的。；他们是当出现问题时使用的参考。你可能在你用钥匙点火之前不会去仔细地看车子的使用说明；类似地，用户和操作员也不总是在使用系统时通览系统说明。事实上，研究表明只有 10%到 15%的用户在培训时完全地阅读文档。六个月以后，没有人再看用户手册了，以及文档化的修订信息。在这种或者其他条件下，用户可能关心设计得很好的图标，在线帮助，示例，和学习系统如何使用的课程。

图标和在线帮助

系统可能被设计使它的用户接口功能清晰、易于理解。很多基于计算机的系统都遵循

了 Apple 和 Xerox 的使用图标来描述用户对系统功能的选择的例子。单击图标选择，双击调用具体的功能。对于这样的系统的培训是很简单的，因为通过查找图标而不是记忆命令和语法而使功能调用变得非常简单。

类似地，在线帮助是用户操作更容易。用户可以浏览系统函数功能而不必在文档中查找。精炼的使用超级链接的在线帮助允许用户专研到函数内部。然而，如注解 11。1 所示自动化培训需要与其他系统同样的维护。

注解 10.1 培训系统也是软件

记住基于计算机的培训常常包含复杂的软件是很重要的，而且它有下降的趋势。Oppenheimer 指出当用户“集中注意力与软件的操作而不是手边的目标时，学习到的东西会变少而不是增加”。他解释说象仿真这样的技术的使用能隐藏我们所做的假设，而不是使我们更清楚或鼓励我们对它质疑。其结果是用户会致力于表面的东西，而不是意识到他们可以改变他们的工作环境。

示范和课程

示范和课程为培训提供了可视化，受到用户和操作员的肯定。示范和课程常常被安排成一个系列，每一堂课讲授系统的一个方面。

示范和课程可以比文档和在线帮助更灵活。用户可能喜欢交互式的方法，这里他们可以运行示范的功能。示范可以在正式的课堂中进行。然而，基于计算机的和网络培训已经在示范和教学系统上取得了很大的成功。

有些教学系统使用了各种各样的多媒体。例如，光盘、录像带，设定的网站可以用于实现这个功能，然后学生在自己的机器上试验这些功能。其他的教育软件和硬件，例如 Robotel，允许老师监督每个学生正在做什么或者控制学生机点击鼠标和键盘。

示范和课程常常包含多种形式的加强学生学习的方式。对于很多人，口头的表述比书面能更长时间地引起注意。

专家用户

有时仅仅看一两个示范或者参加课程是不够的。你需要一个角色模型证明自己能够掌握系统了。在这种情况下，指派一个或者多个用户或者操作员为“专家”是很有用的。这些专家优先于其他人受到培训。其他的人会感到很轻松，因为他们认识到他们的专家是掌握了这些技术的用户。专家可以指出遇到过哪些问题，但是已经解决了。因此专家说服用户那些他们认为不可能的事情是可能的。在正式的培训阶段结束后，专家用户也可以是不固定的。他们作为顾问，回答问题并使他们自己能解决别人提出的问题。许多用户会感到在课堂上问问题是不舒服的，而不会在向比自己更熟练的用户问问题时感到不舒服。

专家用户给系统分析员关于用户对系统的满意程度的反馈，进行额外培训的必要，以及出现的故障。用户有时在向分析员解释系统应该怎样加强时遇到困难。专家既学习了用户的知识，又学习了分析员的知识，所以他们可以帮助避免交流的问题。

培训的方针

只有当培训适应了你的需要以及你的能力时，培训才是成功的。人的偏爱，工作方式，以及

有组织的压力在这个成功中起到作用。一个不会打字的经理可能要求它的秘书会。一个工人可能在课堂上更正上级的错误时感到窘迫不安。由一些学生喜欢通过读来学习，另一些通过听，还有一些通过使用结合的手段。

个人化的系统常常能适应这个背景上、经验上、爱好上的不同。然而一个学生可能对某个概念很不熟悉，可能需要大量的时间去学习他，另一个学生可能很熟悉而直接跳过它。甚至键盘能力也是一个部分：一个需要熟练的打字能力的作业可能被熟练的打字员很快完成，而对计算机非常熟悉的人不需要学习某个外围设备是用来干什么的。复习是为了那些已经对某些函数很熟悉的人设计的。

在培训课堂或者演示中使用的材料应该被分成部分陈述，每个部分的范围应该是有限的。一次性的太多的材料可能会超载，因此很多短的会议会比一次很长的会议易于接受。

最后，学生所处的地点可能决定培训的类型。对全世界上百个地点的人进行培训需要一个基于 Web 的系统或者基于计算机的在实际的系统中运行的系统，而不是让每个预期的用户从各地飞到一个集中的地方进行培训。

10.2 建立文档

文档是培训中易于理解的一种方法。文档的质量和类型可能是很重要的，不仅仅是对培训，而且是对系统的成功。

文档的类型

在产生培训计划和参考文档时，需要考虑一些东西。每一个要考虑的问题可能在决定文档是不是成功的上是很重要的

考虑读者

一个基于计算机的系统被各种不同的人使用着。当出现问题时或者进行了建议的改变时，除了用户和操作员，开发小组的其他成员和用户也需要阅读文档。例如，加和几个分析员正在与用户一起工作以决定是否建立一个新的系统还是修改旧的系统。分析员阅读系统的概况来理解现在的系统能做什么和不能做什么。这种给分析员看的概况与给用户看的是不一样的；分析员必须知道计算的细节而这对用户来说并不值得关心。类似地，操作员需要的描述对用户来说是没有用的。例如。S-PLUS 包附带有几本书，首先包括*自述文件*。每本书是为不同读者看的，具有不同的目的。从文档标志开始，用四页简短的纸。每一本书包括下列文档

- 为计算机初学者提供的 *S-PLUS 系统简介*
- 为熟练使用计算机的用户提供的 *S-PLUS 系统详细教程*
- *S-PLUS 用户手册*，解释怎样使用系统，操作数据，使用其它的高级的图形
- *S-PLUS 统计和数学分析指南*，描述统计建模
- *S-PLUS 系统程序员手册*，解释 S 和 S-PLUS 编程语言
- *S-PLUS 程序员补充手册*，针对某个具体版本的软件信息
- *S-PLUS 光栅图形用户手册*，描述了系统的统计分析的特殊的图形特性
- *S-PLUS 系统概览*，提供了关于文档各版本的交叉引用

我们必须通过考虑想要的读者来设计和完成文档。手册和指南可以写给用户，操作员，系统支持人员以及其他人员。

用户手册

用户手册是系统用户的参考指南。这个手册应该是完整的而且可以理解的，因此它体现了系统的不同层次的用户，开始于普通的目标。首先，手册描述了它的目的和对其它具有更详细信息的系统文档的引用。这些预备知识对于向保证文档包含他们要查找的信息是很有帮助的。手册中使用的特殊的术语、缩略词、只取字首的缩略词被包含在容易找到的参考材料里。

其次，手册更详细地描述了习。系统的概述表现了下面的项目:

1. 系统的用途和目标
2. 系统的性能和功能
3. 系统的特性、性质和优点，包括系统完成情况的清楚的图形表示

概述只需要几段话。

例如，*S-PLUS 用户手册* 从一个叫做怎样使用本书的段开始的。第一段解释了 S-PLUS 系统的用途:“一个数据分析的强大的工具，向你提供了对于探测数据分析，现代的统计技术，以及产生你自己的 S-PLUS 程序的方便的特性”。他列举了用户将从书中学习到的关键性技术：

- 讨论 S-PLUS 系统的命令
- 产生简单的数据对象
- 产生 S-PLUS 系统函数
- 产生和修改图形
- 对数据进行操作
- 定制 S-PLUS 段

每一个用户手册都需要用图形来支持文本。例如，一个图形描述了输入以及输入的来源，输出以及输出去向，以及帮助用户理解系统功能的主要系统函数。类似地，图形可以和叙述一起使用来描述同一件事情。

系统功能应该一个一个地描述，与软件的本身上相独立的。也就是说，用户需要学习系统是用来做什么的，而不是怎样做到这一点的。

不管系统执行什么功能，用户手册的功能性描述至少包括下面的元素：

- 系统主要功能图和他们之间的关系
- 根据用户能看到的界面描述每个系统功能，以及每一个菜单选项和按键将引起的结果
- 每个函数的输入
- 每个函数的输出
- 可以被每个函数调用的特殊性质

例如，S-PLUS 系统的主要部分被通过 S-PLUS 系统既能做图形又能进行统计分析而在用户手册中得到解释。然后，每项功能被扩展开以便于用户能够理解它。S-PLUS 用户手册首先描述了图形功能：

- 散点图
- 一页内多条曲线
- 柱状图
- 盒图
- 图形集的隔离符号
- 图例
- 可能的常用曲线

- 曲线对
- 画笔
- 三维图形
- 更详细的图形
- 其他图形

然后是统计分析功能：

- 一俩个连续数据的例子
- 变量分析
- 线性归纳模型
- 其他归纳模型
- 局部衰退
- 树模型
- 残差分析

如果你不能迅速地和方便地找到你需要的东西，那么完整的、详细的用户手册是没有用的。一个书写很差的用户手册导致对系统感到不舒适的用户灰心而且不能尽可能有效地使用系统。因此，任何增强可读性和信息访问的方式都是有用的，例如，术语表，标签，编号，交叉引用，彩色编码，图形和索引等。例如，一个功能按键表，可能比叙述性的描述他们的位置更容易理解。类似地，例如表 10。2 中的简单图形可以帮助用户查找合适的组合键。

操作员手册

操作员手册以用户手册同样的方式向操作员提供了材料。预期的读者的不同是操作员手册与用户手册的唯一的不同之处：用户希望了解系统详细的功能和使用方法，操作员系统了解系统运行和访问的详细情况。因此，操作员手册描述了软件和硬件配置，给用户授予权限的方法，给系统增加或者减少外围设备的方法，以及复制或者备份文档的方法。

用户在系统中存在层次的差别，操作员也是一样。首先描述系统的概况，跟随着系统的目标和功能的更具体的描述。操作员手册可能与用户手册有些重复，因为操作员也要了解系统的功能。。例如，操作员从不生成电子数据表并从这里生成图形和曲线。但是，知道了系统具有这样的功能会使操作员对系统的功能有更多的了解。例如，操作员可能学习软件的过程名以及使用到的硬件。然后，如果用户报告了错误，操作员可能知道问题是否能够补偿或者是否应该通知维护小组。

通用系统指南

有时你想学习系统是用来干什么的，而不是具体的函数的功能。例如，作为一个大公司的审计部长。你可能阅读系统说明确定系统是否适合你的需要。系统说明不需要描述每一个界面。然而，描述必须能让你能判断系统对于你的公司的需要是不是完整的而且精确的。

通用系统指南强调了这个需要。它的读者是消费者，而不是开发人员。通用系统指南与系统设计文档类似。它按照用户能够理解的方式描述了一个问题的解决方案。除此之外，通用系统文档描述了系统的硬件和软件配置，也描述了系统的体系结构。

通用系统文档类似于汽车消费商给预期的消费者的一个平常的、非技术的小册子。描述了车的类型，发动机的大小、类型以及车体的大小，运行统计，节能性，标准的和可选的特性等。消费者不会关心一个自动化的系统是怎样抽象出来的。类似地，通用系统文档不需要描述算法等。相反的，这个指南只描述了需要建立或者访问新的纪录需要的信息。

一个好的通用系统指南提供交叉引用。如果指南的读者想要关于系统实现的具体方式，他们可以找到用户指南中的合适的页。另一方面，如果用户需要更多的关于系统支持的信息，他们可以求助于操作员指南。

自动化导师制的系统概览

一些用户喜欢在系统的实际功能中学习，而不是阅读系统文档看这个功能的详细的描述。对于这些用户可以开发自动指导系统。用户调用软件或者一步一步解释系统功能的程序过程。有时文档与特定的程序结合到一起；用户首先阅读某个功能，下一步在系统运行这个功能。

其他的系统文档

许多系统文档在系统提交时被提供。有些是系统开发的当前步骤的产品。例如，需求文档在需求分析完成后书写并且在必要时进行更新。系统的设计被纪录在系统的设计文档中，系统设计文档描述了程序的设计。

实现的细节是在我们在第 7 章中描述的程序设计文档中纪录的。然而，另外的一个文档会帮助那些将会维护和增强系统的人。程序员指南是用户手册的副本。正如用户手册以层次表现系统，从系统概述到系统功能说明，程序员表现了软硬件的配置情况的概况。这个概况后面伴随着系统组件的详细描述。以及他们相关的执行函数。为了帮助程序员定位执行特定功能的代码，或者由于发生了一个错误，或者由于一个功能必须被改变或者加强，程序员的指南与用户指南是交叉引用的。

程序员指南也强调了那些使维护小组能定位代码问题的系统的那些方面。它描述了系统支持功能例如运行诊断程序，已经执行的代码行或者内存内的值的显示，有问题的代码的定位以及其他的工具。我们将会在 11 章中更深入地讨论维护的技术。

用户帮助和故障解决

用户和操作员参考文档来分析问题的根源并且在必要的时候请求援助。可以提供集中帮助，包括参考文档和在线帮助文档。

错误信息参考指南

如果系统发现了一个错误，用户和操作员会被用一个统一的相容的方式通知。例如如果你两个名字或字母，例如 “3 x”中间没有操作符，S-PLUS3.0 为产生下面的出错信息：

```
>3 x
syntax error: name(x)used illegally at this point:
3 x
```

或者你会被通知表达式中没有函数调用。

```
〉.5(2,4)
Error:″0.5″ is not a function
```

回忆我们建立的系统设计建议使用一个用于发现、报告、解决错误的体系。各种系统出错消息都被包含在设计和用户文档里，列出了所有可能的消息以及他们的含义。任何可能的时候，错误消息指出了因此故障的错误。然而，有时这个问题无法得知，或者没有足够的空间来在屏幕上或者报告中显示完整的消息。因此，错误消息参考指南，作为报告的最后一个文档，必须被详细地描述。在屏幕上的错误消息可能包含下面的消息：

1. 当出错时正在运行中的代码组件
2. 正在运行的源代码行数
3. 故障的严重程度以及它对系统的影响
4. 任何相关的系统内存或者数据指针的内容，例如寄存器或者栈指针

5. 错误的性质，或者错误消息号

例如，一个错误消息可能在用户屏幕上表现为如下形式：

FAILURE 345A1:STACK OVERFLOW

OCCURRED IN:COMPONENT DEFRECD

AT LINE: 12300

SEVERITY: WARNING

REGISTER CONTENTS: 0000 0000 1100 1010 1100 1010 1111 0000

PRESS FUNCTION KEY 12 TO CONTINUE

用户使用错误号来查阅参考指南，其入口形式如下：

Failure 234A1: Stack overflow.

This problem occurs when more fields are defined for a record than the system can accommodate. The last field defined will not be included in the record.You can change the record size using the Record Maintenance function on the Maintenance menu to prevent this failure in the future.

注意错误消息反射到特定的系统体系，来解决故障和疏忽。改变后的系统将会自动解决这个错误，或者展现给用户，或者预防之。在给出的例子里，它是提交给用户去解决这个问题。

在线帮助

许多用户希望能在手头上找到自动的援助，而不是不得不去找那些类似参考指南之类的东西。常常，界面上又一个作为菜单项的帮助功能，来使用辅助的或者另外的信息。

更具体的帮助可以通过选择另外的菜单或者另一个按键来进行。一些也在你的支持文档中提供了页式的支持。因此，你可以从自动化的系统中直接找到你要的信息，而不是在文档中查找。

快速参考指南

一个即时的测量是快速参考指南。这个系统主要功能的摘要是设计用来作为用户可以在工作站保留的一两页纸。通过参考这个指南，你能够找到进行通常使用的方法或者功能而不必阅读长长的关于每个工作是怎样进行的解释。这个指南在你必须记住特殊的功能键定义或者使用代码以及缩写。

10.3 信息系统实例

Piccadilly 系统看起来有很多对于电视节目或者广告出售很熟悉的用户，但是他们是不必要的在计算机的概念中很好地组织的。因此，系统应该有广泛的用户文档和帮助功能。许多人喜欢在工作中学习，而不是参加几天的培训课程，因此 Piccadilly 培训可以在网上进行，使用实际的系统界面。

例如，用户必须学习如何改变广告的比率，选择一个比例来计算用户的广告费，浏览系统来向用户展示更多的信息。一个培训系统能用户展示 Piccadilly 界面，如图 10.1 所示。然后，培训软件能够增加以使用户能理解系统每个功能的性质和目的。假如一个用户正在阅读识别率界面，如图所示。通过单击文字“Spot Rates”用户能调出培训界面，该界面描述了条件的含义和指向相关的帮助文件。

这个培训目标不仅仅对一些用户是很有用的，而且对于断续的需要记住工作的改变率的用户也是很有用的，而且系统通常是怎么工作的。因此培训事实上可以被任何人在系统存活的任何时间中使用。

注意这种培训软件是很成熟的。它必须与 piccadilly 系统交互来允许通常的系统功能能够运行，而且能为培训活动提供更多的解释和处理。由于这个原因，培训和文档不能被后考虑；他们必须在系统的其他部分被设计完之前设计，并且在系统变大或者改变后进行维护。培训软件可能需要很多的开发时间，所有这些活动都必须计划好并且作为工程管理的常用的部分。事实上，培训和文档常常被看成是系统的特性，在需求说明文档中描述并跟系统的其他部分一起进行开发。

10.4 实时的例子

调查员检查了 Ariane-5 系统失败的原因。注意 Ariane-4 软件的一些软件没有被在文档中表现出来。

..SRI 系统说明没有表明系统的运行限制。这种限制的声明，它本应该对每一个紧急任务设备是强制性的，成为对于 Ariane-5 系统的不顺从的东西。

因此，Ariane-4 软件的重用强调了整个系统的文档化的需要。Ariane-5 的设计者在考虑 SRI 代码从 Ariane-4 中的重用时本应该能读懂所有的在继承的文档中潜在的假设。特别是，Ariane-4 文档应该清楚地表明它的代码不能在 Ariane-5 上运行。也就是说，Ariane-4 的设计说明应该能在 Ariane-4 的 SRI 设计文档中找到并且被 Ariane-5 的设计者仔细地察看。在文档中包含设计说明比让设计者通过读代码来了解系统的函数功能要容易得多。

10.5 本章对你的意义

在这一章里，我们察看了必要的培训和文档来支持系统提交。作为一个开发员，你应该记住：

- 培训和文档应该在工程开始的时候就计划好
- 培训和软件文档应该被与合格的系统软件相集成。
- 所有的培训和文件模块和文档应该考虑不同读者的不同需要：用户，操作员，顾客，程序员以及其他的对系统进行工作或者与系统交互的人

10.6 本章对于你的开发小组的意义

你的开发小组不应该在开发的最后时刻才进行培训和书写文档。培训和文档应该在需求分析结束后就开始。事实上，用户手册应该通过需求文档书写，同时测试者书写系统的可接受的测试脚本。因此，培训员和系统文档编制者应该与开发人员保持稳定的联系，以便于系统需求和设计的改变能够反映到文档和培训材料中来。

而且，培训和文档的更新能被更早地计划。开发小组能用适当的方式来从集中的地方访问他们的更新情况或者让用户直接从 Internet 或 WebSite 中下载。特别地，文档可以被保持最新的，能报告最新的项目，如错误、新特性、新的错误更正以及其他的错误相关的信息。这个更新可能提醒用户或者操作员即将到来的培训课程，复习课程，常见问题，用户群体会议，以及其他对用户、操作员、和程序员有用的信息。

10.7 本章对于研究人员的意义

有一个很大的关于教育和培训的开发主体，而且软件工程研究人员在为软件相关的系统设计

培训计划和文档时将会意识到学习它的重要性。我们需要更多地了解

- 用户对于培训和文档形式的偏爱
- 人机界面接口和对看法保持能力之间的关系
- 人机交互接口和人对于学习方式的偏爱之间的关系
- 获取信息的有效的方式以及对于需要它的人们的有效性

研究员也能用新的方式来鼓励用户之间的交互。例如，一个在线用户可能帮助用户学习新的窍门，交换关于系统和工作区的定制的经验，理解使用系统的更为有效的方式。

10.8 小组的目标

为借贷管理系统书写用户手册。读者是借贷分析员。在你的用户手册里，参考你认为有用的其他文档，例如在线帮助。

10.9 主要的参考文献

美国开发和培训会（ASTD）支持会议，培训课程，以及关于开发和培训的信息。ASTD 也发行杂志，*培训和开发.*

Price(1984)和 Denton(1993)是关于为计算机系统书写有用的书。

10.10 练习题

1. 原型系统允许用户在实际的系统完成之前能够试用系统工作模型。解释如果它在培训时产生一个冲突，原型系统怎样是达不到预期目标的。
2. 给出一个用户培训和操作员培训是相同的的系统的例子
3. 一个自动化的系统的用户不需要对计算机的基本概念很熟悉。然而，计算机的知识对很多操作员来说是非常有用的。在什么情况下自动化系统的用户不能意识到计算机系统的潜在的危险？这种缺乏意识的现象是一个好的系统设计吗？给出一个例子来支持你的答案。
4. 检查你的学校或者工作中一个计算机系统的用户文档。它是清晰的且易于理解的么？它对于一个对于计算机了解得很少的人来说是可以理解的么？错误消息是易于解释的么？有没有独立于用户文档的错误消息说明？在文档中找到主题是容易的吗？你怎样改变文档来提高它？
5. 假设系统的错误处理体系是对现场背后的问题的调停，没有一点用户知识。在一个安全鉴定系统里，在发生了错误时什么是合法的不告诉用户的暗示？系统是否应该报告错误以及正确的动作？
6. 表 10.3 包含了在实际的 BASIC 解释的参考指南中的一些错误消息。对清晰性、信息的总量、以及对用户和操作员的适当性进行评论。

表 10.3 BASIC 错误消息

号码	消息
23	行缓冲区溢出 试图键入具有太多字符的行

24	设备操作超时 你指定的设备在当时是不可用的
25	设备出错 进入了一个不正确的设备指派
26	FOR 后面没有跟 NEXT 进入 FOR 循环而没有遇到匹配的 NEXT 表达式
27	超出纸面 打印设备超出纸面可打印区域
28	不可知的错误 一个在现存条件下无效的错误消息
29	WHILE 后面没有跟 WEND 进入 WHILE 语句而没有与之匹配的 WEND 表达式
30	WEND 没有找到匹配的 WHILE 进入 WEND 表达式没有遇到与之匹配的 WHILE 表达式
31-40	不可知的错误 一个在现存条件下无效的错误消息

第 11 章 系统维护

Maintaining The System

在这一章，我们关注以下几个问题：

- **系统改进**
- **系统继承**
- **影响分析**
- **软件重组**

在前几章里，我们研究了如何建立一个系统。不过，系统的生命周期并没有因提交而结束，在第 9 章我们看到最终系统即使是在软件构建完之后也常常不断地变化。因此我们现在讨论不断地变化着的系统的维护。首先，我们回顾一下系统易于变化的各个方面。然后我们研究一下系统维护涉及的活动和人员。系统维护的过程是困难的：需要检查相关的问题，包括损失以及问题是如何扩大的。最后，我们看一下在系统的进化中能帮助我们改进系统质量的工具和技术。

11.1 系统的变化(The Changing System)

当系统能够为用户所用时，即当系统被用户在实际的生产环境中使用时，系统的开发就完成了，之后的改变系统的任何工作称为维护。许多人把软件系统维护等同于他们在进行硬

件系统维护时所做的工作：对损坏了的和不正常的部分进行修理和预防。然而，软件维护并不能以同样的方式对待。为什么呢？

软件工程的一个目标是揭露以下技术：精确地定义问题，把系统设计成一个解决方案，实现正确的有效的程序集以及调试系统的错误。这个目标与硬件开发相似：生成一个可靠的、没有错误的能按要求工作的产品。这种系统中的硬件维护致力于替换无法使用的部分或使用技术延长系统寿命。然而，Whiledo 结构没有在 10000 次循环以后终止，程序没有运行到表达式的结束。因此，软件系统与硬件系统不同，我们无法将在软件工程中硬件之外的各个方面成功地运用的方式用于硬件。

系统的类型：(Type of Systems)

硬件和软件系统的最大的不同是软件系统的构建体现了变化。除了最小的系统，我们开发的系统都是在不断地变化的。也就是说，系统的某一个或某一些特性常常在系统的生命周期中发生变化。Lehman(1980)提出了一种根据程序的变化对它们进行分类的方式。在这一节，我们也将看一下这种分类怎样应用于系统。

软件系统的变化不仅仅源于顾客决定用不同的方式做这件事儿，也源于系统本身的特性的变化。例如，我们看一下一个公司的薪水扣除计算和薪水支付的系统。这个系统依赖于税收法以及公司所处的城市、州、省和国家的税收调节。如果税收法改变了，或者如果公司迁到另一个地方，系统便需要修改。因此，即使系统在过去工作得很正常也可能需要修改。

为什么有些系统比其它的系统更易于变化？通常，我们可以根据它与其操作的相关的环境。不象抽象的程序，现实世界包含不确定因素以及我们还没有完全理解的概念。系统的需求越依赖于客观世界，就越倾向于改变。

S-系统　有些系统是有规格说明定义的或来源于规格说明。在这些系统中特定的问题是根据它应用的整个环境来描述的。例如。有人要求我们创建一个系统来完成对于给定的在某种限定下的矩阵集中的矩阵的加、乘和转置。这个问题的定义是完全的，而且有一个或多个正确的解决方法。这些方法众所周知，编程者不需要担心这个方法的正确性，而需要关心这个方法的实现的正确性。使用这种方法构建的系统称之为 S-系统。这样的系统是静态的。而且不容易发生改变。

如图 11．1 所示，用 S-系统解决的问题与现实世界相关，现实世界是容易变化的。然而，如果环境变化了，结果就成了一个必须重新定义的新的问题。

P-系统 计算机科学家常常使用 S-系统定义抽象问题，并开发出系统来解决。然而，完全地描述现实世界的问题并是容易的和可能的。在很多情况下，理论上的解决方法是存在的，但是方法的实现是不可能的或者不合实际的。

例如，看一下一个象棋系统。既然象棋规则定义的很完全，问题是完全地描述的。在游戏的没一步解决方法可能涉及对所有可能移动以及移动的结果进行计算来决定最佳的移动。然而，完全地实现这个方法在目前的技术水平下是不可能的。在有限的时间内移动的可能步数太大了以至于无法评价。因此，我们必须研究一种在实际中更可行的近似的解决方法。

为了开发这个解决方法，我们用抽象的方式描述这个问题。然后从我们的抽象的观点出发书写系统的需求说明。使用这种方法开发的系统叫 P-系统，因为它是机遇对于问题的实际的抽象，而不是完全地定义的规格说明。如图 11．2 所示，一个 P-系统比 S-系统动态性更强。这个方法产生可以跟问题对比的信息，如果这个信息无论如何都不合适，问题的抽象概念就会改变，需求也会修改以使最终的解决方案更实际。

因此，在 P 系统里，需求是建立在近似解的基础之上的，这个解决方案部分地依赖于对产生需求的分析的解释。即使正确的解决方法可能存在，由 P-系统产生的解决方法在它

产生的环境中得到优化 。在 S-系统中，如果规格说明是正确的则解决方案可以接受。而在 P-系统中，如果在问题所在的现实世界中，结果是有意义的，则解决方案是可接受的。

在 P-系统中，很多东西会发生变化。当输出信息与实际问题对比抽象结果可能发生变化，需求可能需要扩充，因此实现也受到影响。由于改变而产生系统不能作为一个新问题的新的解决方案。相反地，这是对旧的解决方案的修改去找到更合适的现存问题的解。

E-系统。就 S-系统和 P-系统而言，现实世界条件仍然是稳定的，然而第三类系统结合了现实世界的不断改变的性质。E-系统是嵌入到现实世界中的而且随着世界的变化而变化逐个解决方案建立在涉及的抽象过程的模型的基础上因此这个系统是它模拟的世界的一个完整的部分。

例如，一个预测一个国家的经济状况的系统建立在经济作用模型的基础上的。在问题所在的世界中不断地发生变化。然而，经济没有被完全理解，所以模型随着我们的理解的变化而变化 。图 11．3 描述了 E-系统的改变能力以及它对于它的现实世界的依赖性。虽然 S-系统不怎么喜欢变化，P-系统倾向于逐渐的变化，E-系统倾向经历不断的变化。更重要的是 E-系统的成功完全依赖于顾客对系统的评价。既然由 E 系统完成的问题不能被完全地描述，系统必须根据它在实际的操作条件下单独进行判断。

这些分类告诉我们系统的因素易于变化，可以变化的因素的数量越大，系统的维护就越必要。特别是，既然由于系统产生的问题可能发生变化，E-系统解决方案可能经历一个不断的改进。

系统生命周期中的变化(Changing During The System Lifecycle)

通过检验这几类系统（S,P,E），我们可以看到在什么地方开发可能变化以及这些变化将会怎样影响系统。从本质上讲，S-系统问题是完全定义的，而且不容易变化。对于相类似的问题通过修改 S-系统来解决，但修改系统是一个新的分析、解决的全新的问题。如果 S-系统不能正常工作，常常是因为它解决了一个错误的问题（即它不是用来解决这个问题的），我们的反应是重新定义问题，产生它的新的描述。我们开发新的解决方案而不是修改旧的系统。

P-系统是问题的近似解。当发现不符合或有遗漏时，系统可能改变。事实上当我们对比系统产生的信息和它所模拟的现实世界条件时，我们可能需要修改系统以保证它是经济的、有效的。

对于一个 P-系统，一个模型模仿一个特定问题的解决方案。因此在开发的所有的阶段都可能发生修改。首先，问题的抽象可能变化。换句话说，我们修改问题的抽象描述，然后据此改变需求说明。其次，我们修改系统设计，重新实现测试以并入这些变化。最后，我们修改这个近似的系统以及程序文档，并且可能需要重新的培训。

E-系统使用抽象化的概念和模型来模拟现实情况。因此，E-系统至少要经历 P-系统可能经历的变化。事实上（它们的本质）由于其属性也可能变化，它们的性质是更不稳定的。因为嵌入到不稳定的活动中，E-系统需要把特性建立到系统自身内以适应系统。

任何类型的系统的变化的例子列于表 11．1，例如，在需求分析阶段对需求修改可能导致规格说明的变化。技术的设计的修改可能需要系统设计或初始需求的改变。因此开发阶段的任何修改可能影响前期和后期的结果。

软件工程的规则建议系统开发在维护的早期变化。如果你已经在需求阶段组织了设计代码模块以及交叉引用部分，你就可以跟踪需求变化到受到影响的模块，以及必须重新进行

测试。相应地，如果发生了错误，你可以找到包含错误的模块，然后在所有的层次（设计、编码和测试）上进行更正而不仅仅在编码上进行修正。软件工程的原则不仅导致了好的设计和代码，而且导致了快捷的改变能力。

表 11.1 软件开发阶段的改变

导致变化的活动	相应需要改变的手工材料
需求分析	需求说明
系统设计	架构设计说明
	技术设计说明
程序设计	程序设计说明
程序实现	程序代码
	程序文档
模块测试	测试计划
	测试脚本
系统测试	测试计划
	测试脚本
系统提交	用户文档
	培训工具
	操作员文档
	系统指南
	程序员指南
	培训课程

系统的生命范围(The System Life Span)

当软件工程师试图建立可维护的产品时，我们必须问自己建立这样一个系统是否是可能的，换而言之，我们使用高内聚，低耦合的组件，文档是否是完全的而且最新的。整个系统的交叉引用的。我们需要维护吗？不幸的是，答案是是。原因在于系统本身的特色。正如所看到的那样，无法保证 P-系统和 E-系统不需要改变。事实上，我们必须假定它们将会发生变化，然后建立系统，以使他们很容易被改变。

然后，我们的下一个问题是：我们预期可以发生多少改变？答案取决于系统的性质。S-系统有一点儿或没有改变。P-系统有很多改变。E-系统倾向于不断地变化。由于这个原因，许多软件工程师喜欢把开发阶段的维护称作改进期。我们所说的原始系统是当我们需要的时候建立的。但正如我们所看到的，环境不同了，我们必须评估原始系统，帮助原始系统随着我们的技术和事务所需要发展而发展。基于以上观点，我们可以决定用一个全新的系统去代替原始系统或放弃原始系统，因为它不再需要了。

开发时间和维护时间( )

我们可以看一下其他工程的开发与维护需要的时间，从而了解改进的阶段需要多长时间。根据 Parilch 和 Zvegintzon（1983），开发这类型的工程用了一年到两年的时间，而需要额外的 5-6 年的维护时间。就效果而言，工程设计策略的一半以上花在维护上。由图 FieldStad and Hamlen(1979)做的调查报道了一个相似的问题，25 个过程结构数据中的说明。他们平均

有 39%的努力用在开发，61%用在维护上（更正，模拟和用户培训）。近期的调查报导了相类似的发现。，并且许多开发者认为 20%的努力用在开发上，而 80%用在维护上 。

系统的发展与系统的衰退，当一个系统需要显著的不断改变时，我们必须决定是否放弃原始系统而建立一个新的来代替他。为了决定怎样做，我们讨论下面几个问题：

1) 维护费用很高吗？
2) 系统的可靠性是不能接受吗？
3) 在合理的时间内，系统可能不再适合进一步的改变了吗？
4) 系统运作情况一直超过规定的限度吗？
5) 系统的有限的作用是无用的吗？
6) 其他的系统作同样的工作比原系统更好，更快或更便宜吗？
7) 维护硬件的成本很高，足以证明用便宜的更新的硬件取代它是有理的吗？

对于这些问题的答案，可以说出用新系统代替旧系统的时间。于系统的开发与维护相联系的，从创作到淘汰的一系列的全部耗费称为生命循环成本。通常，我们根据比较原始系统、修改系统和新系统生命循环。成本来决定是维护、重建还是取而代之。

软件改进的规则。了解在余下的时间里系统发生了什么,有助于我们的维护的决定。我们对于在大小，复杂性，策略和易于维护的改变问题感兴趣。通过检验大型系统看它们是怎样变化的我们可以了解许多发展趋势。例如，注解 11．1 描述了贝尔大西洋公司一个大型系统的发展。

通过自己的经历,Lehman 已经注意到系统的行为正如它们所预料的。它在发展计划中总结了它的发现（Lehman 1980）：

1) 继续改变。使用过的程序，经历了不断的改变或变得无用。改进的过程或衰退的过程持续到使用重新生成的版本替换这个系统，更能节约成本。
2) 不断增加的复杂性。当一个发展中的程序不断地被改变，它的结构就会变坏，反过来相应的改变，它的复杂性也就增加，除非你设法维护或缩减它。
3) 程序改进的规则〉程序的改进归功于这样的动力。它导致编码过程整体系统和系统属性的评价以及静态决定的自我约束。
4) 维护编制中的稳定性。在程序的存在的生命中，程序的全局活动的是稳定的持续不变的。
5) 保持普遍性。在程序的存在的生命中，发展中的系统的后继的发行的内容是稳定的。

注解 11．1　BELL ATLANTIC 使用一个改进的系统代替三个系统

在 1993 年，Bell Atlantic（现在是 Verizon）开发了售后服务交流系统（SSNS）来代替三个旧的系统，它支持操作员使用新的基于电话设备的服务。系统的最初目标是把错误最小化，并且缩小用户服务需要占用的电话时间。但是，当销售代表使用这个系统时，管理人员发现的系统具有可以提供屏幕上的暗示的潜力提醒产品部的代表这可能很适合消费者的需要。系统的目标因此而改变了，从定购方式到基于需求的销售。

SSNS 定购过程很象一个交互，由系统引导。当客户代表输入关于顾客的更多信息时，SSNS 向这个代表展示相关的产品。这样，当 Bell Atlantic 扩充它的产品和服务时，SSNS 系统也必须相应地增强。SSNS 已经被扩展用于处理账单信息，校验地址和信用卡，为用户产生校验的结果，并给出代表服务问题的答案。

系统使用简洁的英语替代了古老的命令，以前的 20 卷手册现在都放在网上。系统已经被定制成几种形式，每个形式处理不同的电话服务。一些调整的代理商在开始时要求公司提供关于产品信息的描述，因此系统就适合这项工作。

最初系统是用 C/C++语言书写的，在九十年代晚期系统使用 Java 改写以支持居域网访

问。当规章，产品，技术和商业需求改变时，SSNS 系统必须设法适应他们。（1997）

第一条规则表明大的系统永无休止，他们不断地变化发展。当我们增加特性增加限制，于其他系统交互，支持大量的用户等时。系统变大，他们也由于环境的变化而变化。他们转移到其他平台或用新的语言重新书写。

第二条规则告诉我们随着大系统的发展，他变得越来越复杂，除非我们采取措施减少复杂性。很多时候，这种复杂性的发生是由于我们必须匆忙地打补丁来解决问题。我们没有办法花时间来维护整个代码重的一贯性和好的设计。

根据第三条规则，软件系统证实了正常的行为以及我们能够测量和预言的趋势。事实上，很多软件工程研究者致力于发展软件开发和维护的”普遍真理”。

第四条规则表明，在编制的属性，如生产率中不存在广泛的相似性。Lehman 引用 Brooks 的观点支持这条规则，也就是说，从某种意义上讲，资料与输出达到最佳。增加更多的资源不能显著地改变输出。类似的第 5 条规则表明一段时间以后后继发行的软件对于整个系统的功能不会产生很大的改变。

11.2 维护的本质(The Nature of Maintenance)

当我们开发一个系统时，我们集中注意力于产生能实现需求而且能正常工作的代码 。在开发的每一个时期，我们不断地借鉴前后阶段产生的结果。系统设计与需求说明紧密相连，代码模块是交叉引用的。而且与设计一致，测试基于找到函数和规范是不是按照需求和设计的规定工作的。因此开发包括以一种仔细的可控制的方式复查。维护是不同的，作为维护人员，我们不仅回顾开发的产品而且建立用户和操作人员的关系。以弄清楚它们对于系统工作方式是否满意，我们也展望未来以提前发现将会出错的东西。考虑由于变化的商业需求而产生的功能性的改变并且考虑由于硬件软件或接口的改变而改变系统的需要。因此，维护的范围更广。有更多的事情需要去追踪和控制。下面让我门看一看为了使系统顺利地工作，并能区分操作人员而需要进行的活动。

维护活动和作用(Maintenance Activities and Roles)

维护的活动于开发类似：分析需求进行系统和程序设计，书写并检查代码。测试系统的改变并更新文档以便进行维护的人员如分析者编码者设计者就相似的角色。然而由于这些改变常常需要对代码结构和内容非常熟悉。编码者在维护中比在开发中起到的作用更大。

1. 维护调节系统日常功能
2. 维护控制系统的修改
3. 使现有的可接受的功能完全
4. 防止系统设计从衰退到不可接受的水平

校正维护措施(Corrective Maintenance)

为了调节系统的日常的功能，我们这些维护人员负责从错误中找出问题的结果，研究的这些问题称作矫正维护。由于失败，问题引起我们的注意。我们找到失败的原因然后矫正和改变需求、设计、代码、测试和文档。常常在开始的修理是暂时的，仅仅能使系统工作而不是最好的修补。长期的改变可能向我们展示一个有打印页面上面有太多行的报告范例。我们的程序员来分辨由于打印机驱动程序而引起的问题作为紧急的处理开发人员高用户怎样在

打印前通过在报告菜单上设置参数来设置每页行数。最后开发小组更新设计、编码、测试打印驱动程序以使它在没有用户交互的情况下能正常工作。

有时候系统的一个部分的改变需要改变其他的模块，自适应的维护是对这些次要的改变的实现，例如，看一下现存的数据库管理系统，它是一个更大的软硬件系统组成。被更新为新的版本。在这个过程中程序员发现磁盘访问过程需要一个参数，自主地改变被实施从而增加这个额外的参数。他们仅仅允许系统在变化的时间内改变。。

相类似地，一个编译器因为增加了调试工具而更强，我们必须扩充菜单、图标和快捷键来允许用户选择调试功能。自适应的维护也可以适用于硬件或环境的变化。如果一个系统最初被设计工作在干燥的稳定的环境下，当它被用于坦克或潜水艇里时系统必须能适应移动、磁场和潮湿。

完整的维护(Adaptive Maintence)：我们检查文档，涉及和编码测试，寻找改变的机会。例如我们在系统中增加一个函数时最初的透明的基于表的设计可能变得难以跟随。基于规则的更新，设计将来将会加强未来的维护，使我们增加新函数的工作变得更加容易。完全的维护包括实施改变以增强系统的某个方面。这些改变不是由于出错而一起的改变，改变文档以使条理更加清晰，改变测试集以提高测试的覆盖面，修改代码和设计来增强可读性都是自适应的维护的例子。

预防的维护于自适应的维护类似，预防的维护包括改变系统的某些方面以防止出错，它可能包含某一种维护的检测，检查方式。错误处理的增强或另外的给 Case 语句增加”Chtch-all”表达式等以确保系统能处理所有的可能。预防维护常常在程序员或代码分析员找到了还没有出错的实际的或隐藏的疏漏并且采取措施在产生危害前更正错误。

系统维护人员(Perface Maintence)：　开发系统的人员通常不被用于维护系统。常常要雇佣单独的维护队伍来保证系统的正常工作。这有好的一面，也有不好的一面。开发人员对代码设计和以及系统的关键函数熟悉，如果开发人员指导他们需要建立一些它们需要维护的东西，他们会以能使维护更容易的方式开发系统。

然而开发人员对于自己对于系统的了解程度非常自信，以至于它们倾向于不能够及时更新文档，他们对于书写和保留文档的忽略可能导致需要更多的人力和资源来解决这个问题，这种情况导致从问题的发生到问题的解决的时间更长。许多顾客不能容忍这种耽搁。

常常，一个单独的分析员、程序员和设计员的小组组成了维护团队。一个新的团体可能比原来的开发队伍更客观。单独的队伍更容易发现系统应该怎样工作以及它实际是怎样工作的。如果开发员知道别人是通过它们的文档开始工作的，他们就会在文档上和编码规范上更加细心。

维护小组的职责(Team Responsibilities)

系统维护包括所有的小组成员。主要是：用户、操作员或给维护小组提出解释和问题的顾客。分析员和程序员找出受到影响的代码模块，对系统设计的影响，以及相应的资源需要的变化。该小组需要进行许多工作：

1. 理解系统
2. 在文档中定位系统的信息
3. 保持系统的文档是最新的
4. 扩充现有函数以适应新的、变化中的需求
5. 给系统增加新的函数
6. 找到系统出现问题的根源
7. 找到并更正错误

8. 回答关于系统工作方式的问题
9. 重新进行设计和代码重组
10. 重写代码设计和代码组件
11. 删除无用的设计和代码组件
12. 管理系统发生的变化

除此之外，维护小组还与用户、操作员和顾客一同工作。首先，他们要从用户的角度理解问题。其次，问题转换成对修改的要求。需求的改变包括系统现在是如何工作的，用户希望系统如何工作以及需要进行那些修改。一旦系统的设计和编码改变了，维护小组就需要重新培训用户。因此系统既包含了与软硬件的交互，也包含与其他人的交互。

维护时间的使用：

维护，人员对于维护时间的使用是不同的。Lienta 与 Swanson(1981)调查了 487 个数据处理的组织，结果如图 11．4 所示.主要的精力花费在完全维护和自主维护上。后期的报告与之类似。但是不同组织的分布取决于很多东西，包括系统是不是 S-,P-,E-系统，以及商业上需要变化的速度。

11.3 维护中存在的问题(Maintencance Problems)：

系统维护是困难的。因为系统已经可用，维护人员需要权衡改动的需要和用户的易操作性。例如，更新系统将使用户在几个小时内不能使用该系统。然而，如果系统是紧急的，将不允许你几个小时停止使用系统。例如，一个对病人的生命支持系统不能断开。维护小组必须找到一种方式使系统的改变不会给病人带来不便。

维护人员的问题(Staff Problems)

有很多的维护人员和组织的问题使维护变得困难。职员必须在很短的时间内找到问题的解决方法，思考这个软件保证解决方法沿问题的路线。

理解的限度：为了平衡用户需求和软硬件需求，维护小组要向人类理解能力的极限挑战。但是人学习文档及相关的问题的效率是有限的。更重要的是，我们常常要找到比必要的更多的材料。

Parikh 和 Zvegintzow(1983)报告说 47%软件开发耗在对要修改的软件的理解上。例如，如果一个系统具有 m 个模块，我们需要改变其中的 k 个。将有

K*(m-k)+k*(k-1)/2

个接口需要检查是否有问题。因此，即使只有一行代码改变也可能导致上百的系统组件需要测试以保证改变对系统的其他组件没有直接或间接的影响。

对用户的理解也可以避免问题。Lientaz 和 Swanson(1981)发现一半以上的维护人员的问题是由于用户能力的低下和对系统的缺乏理解。例如，如果用户不能理解系统如何工作的，他们会向维护人员提供误导的数据。

这些结果说明了清楚、完整的文档的重要性。维护人员也要有较好的“人缘”。正如我们在第 2 章看到的那样，有各种不同的工作方式。维护小组必须理解不同类型的人员思考问题的方式的不同，必须善于与人交往。

维护的优先级：维护小组权衡顾客对于系统需求进行管理的渴望。管理常常重于技术。管理者常常认为维护比开发一个新的系统更为重要。换句话说，公司必须时常象平常一

样关注日常事务，而不是开发新的产品。可是当管理人员要求修理就的系统时，顾客就吵着要更新的功能或系统。类似地，市场对新产品的需求会激励我们去更快的更粗糙的实现，而不是遵循严格的、好的软件工程方法。结果是打满补丁的难以理解的系统，需要不断地进行修补。

士气：Lientz 和 Swanson 的研究（1981）表明 11.9%问题是由于士气低落。主要的士气低落的原因是他们被看成是第二位的。程序员常常认为设计和开发一个系统更需要能力。然而，正如我们看到的，维护的程序员能发现开发员不会发现的问题。维护不仅仅是写代码和文档的能力，还有与用户工作，预见更改，侦察等。找出问题，理解大系统的工作方式，修改系统结构、代码、文档需要更强的能力。

有些公司在开发和维护之间对换人员，给程序员学习不同东西的机会。这种对换避免了维护编程的侮辱。然而，程序员要同时进行不同的系统的工作。对程序员时间的需要导致程序员主次不分。在维护里 8%的问题由于程序员同时参与了太多的事儿而不能集中注意力。

技术问题(Technical Problems)

技术问题也会影响维护的效率。有时，这些也是开发员和维护人员以前的工作遗留下来的问题。有时，它们是实现中采用了得特殊的用例和程序的结果。

人为性和范例(Artifacts and Paradignk) 如果设计的逻辑不是一目了然的，那么我们就难以决定我们的设计会不会易于发生改变。不灵活的设计将会需要额外的时间来理解，修改和测试。例如，开发人员可能包含了一个仅限于磁带的输入和输出，那么主要的修改可能限于磁盘访问，因为磁盘访问与磁带的顺序的访问不同。类似地，开发人员可能没有考虑 2000 年问题，即使用两位数来表示年份，是一个关于使用简单但很局限的设计对维护产生巨大影响的很好的例子。

维护面向对象的程序也可能会有问题，因为这种设计常常包含由复杂的继承关系引起的高度的相互关联。一次次的增加必须非常小心，以为修改会导致包含易混淆的重复定义的对象。注解 11.2 更多地描述了当对面向对象的程序进行维护时需要进行的特殊的设计。

通常，不合适的设计说明和低质量的程序以及文档占用了大约 10%的工作量。硬件需求所占的比重大约相当：主要是合理的存储和处理时间。作为学生，你会理解当你遇到一个需要解决的问题但没有访问工作站的权限或者不得不不断地拨号以获取远程访问权限。当硬件、软件或数据不可考时，就会出现问题。

测试的困难：当找时间来进行测试会有困难时，测试就会成为问题。例如，航空预约售票系统必须根据时钟来判定其有效性。让用户放弃两个小时进行测试可能是困难的。当系统在进行紧急的任务时，如通信控制或病人监护，把它拿到飞机下解决是不可能的。在这些情况下，我们常常在另一个副本上进行测试，然后把这种改变转移到实际的产品中。

除了时间的问题，还有可能没有好的合适的测试用例来测试系统的改变。例如，一个地震预报系统可能被修改以调节传感器的信号。测试数据必须是相似的。因为科学家还没有完全弄清楚地震的发生机理，很难找到精确的测试数据。

注解 11．2 面向对象系统的维护的优缺点
Wilde,Matthews 和 Huitt(1993)研究了面向对象的系统与面向过程的系统之间的不同。他们纪录了面向对象的一些优点： ● 对于一个单独的对象类的修改不会对系统的其他类产生影响 ● 维护人员可以很容易地重用对象，只需要书写很少的新代码 然而，它也有一些缺点： ● 面向对象的技术可能是程序更难于理解。很难废除原来设计者的意图，因为设计是

分散的：系统的计划分散在系统的不连续的段中。

- 由于同样的原因，多个部分可能是对整个系统的行为的理解变的困难
- 继承关系可能使依赖关系难以跟踪
- 动态绑定使找出那些方法将会得到运行是不可能的，因此维护人员必须考虑所有可能的情况。
- 由于要隐藏数据结构，程序的函数常常分布在几个类中。找到并解释类的交互是很困难的

最重要的是，对于测试者来说，预报设计和编码可能的结果并预防之并不总是那么容易。尤其是在队伍中的不同人员处理不同的问题时。如果对一个组件进行修改来解决数据流的问题并相应地修改其接口时，这种改变结合起来可能引起新的问题。

折衷的必要(The Need to Compromise)

维护小组常常要在一组目标与另一组目标之间进行权衡。正如我们所看到的那样，混淆会发生在用户系统的有效性和系统编辑、更正、增强的实现上。因为问题发生在不可预知的时间里，维护人员可能会意识到这种混淆。

对于计算专业人员，另一个冲突在任何改变时都会发生。软件工程的规则与软件的方便性和成本之间互相竞争。常常，问题以下面两种方式之一解决：一个是快捷但是不好的解决方案。或者是一个实现完全的方便的。当我们注意前者时，程序员可能由于必须马上进行改变而被迫在方便性和规则之间权衡。

当我们采用这样的折衷时，相关的事件可能使未来的维护更加困难。首先，用户和操作员常常会抱怨维护人员。这个人不能从设计和编码的上下文捕捉到问题。其次，问题的解决仅仅包含了问题的及时的解决。不允许修改系统或程序设计来使整个系统跟易于理解，使系统的改变能被系统的模块包容。这两个因素使维护队伍将迅速的修复作为自己的有限的目标。他们不得不把注意力集中在它们还很不了解的问题上。

维护队伍必须解决另一个冲突。当系统被开发来解决一个初始的问题时，开发人员有时试图不改变设计和编码来解决相似的问题。这样的系统常常由于它们的通用的代码而运行缓慢，必须评估大量的情形和可能性。为了提高质量，系统可以合并特定的模块来牺牲通用性来换效率。这些特定的模块常常很小，以为它们不需要考虑每一种情况。改变后的系统更加容易改变。小组在分析怎样和为什么进行修改时必须衡量通用性和速度。

其他的维护小组实施的可能影响结果的途径有：

- 错误的类型
- 错误的危险程度和严重程度
- 必要的改变的难度
- 必要的改变的范围
- 改变的模块的复杂度
- 进行了相应改变的物理地址的数量

这里描述的所有的因素告诉我们维护人员进行双倍的工作。首先，小组理解系统设计、编码、测试体系结构。其次，它提出了关于维护和目标系统结构的哲学。在长期和短期的目标之间平衡，小组决定何时牺牲质量求速度。

注解 11．3　在 CHASE MANHATTAN 中平衡管理和技术需求
Chase Manhattan 的 Middle Market Banking Group 已经占了商业银行服务的对于纽约地区中小型企业的一半业务。为了弄清楚他们的顾客是那些人，他们主要用那个银行以及他们

将来可能更多地购买什么产品，公司开发了关系管理系统(RMS)，这个系统为销售人员提供了对市场主体客户的数据的多个类型的单一接口，这些数据包括：存取平衡和转换等。它使用了 PC/LAN/WAN 技术来跟 Chase Manhattan 的旧系统接口，强调了使顾客代表节省时间以及了解顾客的目标。

系统是作为 Chemical Bank 开发的一个应用在 1994 年出现的。当 Chemical 与 Chase 在 1996 年合并后，Chase 决定修改 Chemical 的系统以适应更大的银行。RMS 系统包括很多步骤。它与另一个系统,the Global Management System，合并了，后来又合并了几个排除冗余，连接硬件平台和商业事务的系统。RMS 开发成基于 Window 图形界面的，系统被修改允许使用支持电子表格和打印报表的微软公司的产品。然后，系统混入了 Lotus Notes，以使数据的改变只能够通过 Notes 应用进行。RMS 的一些部分是在 Chase Manhattan 银行的其他部门实现的，系统配置在 Intranet 上以提供远程访问支持。

维护开销(Maintenance Cost)

所有的维护问题归于软件维护的高成本。在二十世纪 70 年代，大部分的软件预算用于开发。在 80 年代，开发费用的比重开始降低，维护占了大约生命周期的 40-60% 。

影响成果的因素。除了已经讨论的问题，还有很多其他因素归于维护系统的努力。这些因素主要有：

- 应用类型. 实时的和高度同步的系统比那些对时间要求不强的系统更难以改变。我们必须非常小心以保证对于系统一个组件的修改不会影响其它模块的相应时间。类似地，以严格的数据形式改变程序需要对大量的数据访问过程进行修改。
- 系统的新特性。当一个系统实现一个新的应用或以新的方式实现相同的功能时（例如在注解 11．3 中描述的系统），维护人员不能简单地依赖于它们的经验和理解来找到并更正错误。它将需要更长的时间来理解设计，找到出现问题的代码，测试更正后的代码。在很多情况下，当旧的测试数据不存在时必须产生额外的测试数据。
- 流通量以及维护人员的有效性。需要足够的时间来理解系统从而改变它。如果小组成员不断地转换到其他组织，如果成员离开组织去进行另一个工程，或者如果小组成员被期望同时维护几个不同的系统，维护的努力就会受挫。
- 维护生命范围。 一个被设计来长时间工作的系统比一个工作时间短的系统更难于维护。快速的更正和文档的不及时更新在短期的系统里可能是可接受的。但是在长期的系统里，将是致命的，在这里他们将使其他小组成员进行后继的改变更加困难。
- 对于变化的环境的依赖。 一个 S-系统常常比 P-系统需要的维护要少，而 P-系统需要的维护比 E-系统要少。特殊地，一个依赖于硬件特性的系统在硬件发生改变时需要很大的改变。
- 硬件特性。不可靠的硬件组件或不可靠的供货商支持可能使找到问题的根源更加困难。
- 设计质量。如果系统是由独立的、粘着的组件组成，找到并更正问题的根源将会由于对其他模块产生不可预料的结果的改变而复合。
- 代码质量。如果代码不是结构化的或者没有它的系统结构的主要法则，定位错误将会很困难，语言本身就会使得找到并更正错误很困难。高级的语言常常增强可维护性。

- 文档的质量。没有文档化的设计或编码使查找一个问题的解决方法几乎是不可能的。类似地，如果文档难以读懂或甚至是不对的，维护人员就会找不到头绪。
- 测试的质量。如果测试是在不完全的数据下或没有预料到改变的反射进行的，修改可能产生其他的系统问题。

维护的模型(Modeling Maintenance Effort) 在进行开发时，我们想要估计一下维护系统需要的努力。Belady 和 Lehman(1972)是第一批试图使用预测模型掌握维护结果的人之一。他们考虑了大系统随着时间的退化。一系列的修改和增强常常导致系统活动的碎片，而且，通常，系统在每一次维护修理后变大。

在很大的系统内，维护人员必须称为系统的某些方面的专家。亦即，每个小组成员必须专攻特定的功能和运行领域：如数据库、用户接口、网络软件。这种专攻常使小组没有专家，没有一个个人对于系统如何工作、如何与需求挂钩由一个系统的了解。小组成员的专攻常常导致投入到维护中的资源指数增加。需要更多的人力处理这个不断增大的系统，还必须有设备和时间来支持。为了对其他系统组件或功能的工作进行二次检查，需要更多的通讯。

同时，作为这二者的结果系统常常变得更加复杂。首先，当一个错误被更正时，修正本身可能带来新的系统问题。其次，进行修正时，系统结构发生了相应的变化。因为许多修理是在有限的目标下来解决特定的问题的，相关的组件，象面向对象系统中的继承关系那样，也常常退化。

Belady 和 Lehman 使用下面的公式描述这些影响：

$$M = p + K^{c-d}$$

M 是花费在系统上的总的维护精力，p 表示总体的有效的精力：分析、设计、编码、测试。C 是由于设计和文档结构的不合理产生的复杂度；它随着 d 而减小，d 为维护小组对系统的熟悉程度。K 是由比较此模型与实际工程的关系而决定的常量；被称作经验常数。

Belady-Lehman 公式表达了系统维护中各因素的非常重要的关系。如系统不是按照软件工程法则建立的，c 的值会很高。如果，另外，它是在没有对软件本身理解的前提下进行维护的，d 的值会很低。其结果是软件的维护成本成指数增加。这样，为了在维护上节约，最好的方式是使用好的软件工程实践并给维护者时间来了解软件。现在的耗费和计划模型使用 Belady 和 Lehman 提出的很多相同的因素。例如，COCOMO II 计算机维护中使用一个 size 变量，计算如下(Boehm et al.1995)：

Size=ASLOC(AA+SU+0.4DM+0.3CM+0.33IM)/100

变量 ASLOC 测量源代码行数，DM 是设计修改的百分比，CM 是代码修改的百分比，IM 是外部代码集成的百分比，SU 是表示需要对软件的理解程度的度量，如表 11．2 所示。例如，如果软件是高度结构化的，清晰的，那么理解仅需要 10% 。如果没有文档，散乱的代码，需要 50%。

COCOMO II 也包含了为了访问代码和产生改变需要的代价的比率，如表 11．3 所示。越需要测试和文档，需要付出的代价就越大。

很多讨论关注开发中关于维护的估计。特别是，最好的估计是基于过去的对于相似的工程的整个历史的。另外，当工程或者产品的属性发生改变时，这个预测必须重新进行；既然遗留下来的系统不断地改变，这个估计也基于定期的改变。

11.4 测 量 维 护 性 质 (Measuring Maintenance Characteristics)

我们讨论了可以使软件易于理解、增强、更正的属性。当系统被提交后使用这些特性来衡量软件，我们可以预测我们的软件在整个生命周期内是可维护的。在维护过程中，这些测量可以引导我们的行为，帮助我们衡量改动的影响或评定一些提议的改变的优点。

可维护性不仅仅受编码的影响；它描述了许多软件产品，包括说明、设计和测试计划文档。因此，我们需要对我们需要维护的所有产品进行可维护性测量。

我们可以用两种方式看待可维护性，把外部的和内部的观点反应到软件本身。本书内定义的可维护性是软件的外部特性,因为它不仅依赖于产品，还依赖于进行维护的人、支持文档和工具，以及计划的软件使用方法。也就是说，我们不能在没有在特定环境下监控软件的运行的情况下进行度量。

另一方面，我们应该在软件还没有被提交前就进行可维护性测量，以确保我们具备必要的资源来解决可能遇到的所有的问题。对于这一类测量，我们使用软件的内部特性，确定它们能预测外部特性。因为这一目标不是直接的目标，我们必须衡量实际上这个实际的间接目标与外部目标的正确性是否真正一致。

可维护性的外部表现：

为了度量可维护性同时进行修理，我们需要对每个问题的下述信息进行详细纪录：

- 问题报告的时间
- 由于管理延迟的时间
- 分析问题需要的时间
- 详细列出修改需要的时间
- 进行修改需要的时间
- 测试修改需要的时间
- 对修改建立文档需要的时间

图 11.5 描述了英国某个大公司的软件系统的各子系统的修复所需要的平均时间。这个信息对于区分引起问题的子系统以及计划预防的维护是很有用的。对于修补这样的图形的平均时间的跟踪向我们展示了系统是不是可维护的。

如果有效的话，别的测量也可能是有效的：

- 总的改变实施时间与实际贯彻的修改时间的比
- 尚未解决的问题的数量
- 引起新的问题的改变的比率
- 在尚未解决的问题上花费的时间
- 实现一个改变需要修改的模块数

这些测量绘成维护程度和维护过程有效程度的图形。

可维护性的外部表现(External View of Maintenance)

很多研究者建议测量关于可维护性的内部特性的测量。例如，前面描述的复杂度的测量常常与维护的努力相关；也就是说，代码越复杂，需要付出的努力就越多。要记住这种关联与测量是不同的。但是在结构不清晰和文档不全的产品以及他们的可维护性上有直接的清晰的联

系。

圈数 。 在维护中最常用的测量方式是圈数，它是由 McCabe(1976)定义的。圈数是通过度量代码的线性独立性的数量捕获源代码的结构复杂度的测量制度，它是通过把代码转换成等价的控制流程图来计算的，然后使用图形来检查制度。

为了看一下圈数怎样计算的，我们看一下 Lee 和 Tefenhart(1997)写的 C++代码：

```
Scoreboard::drawscore(int n)
{   while(numdigits->0){
                score[numdigits]->erase();
    }
    //build new score in loop,each time update position
    numdigits=0;
    //if score is 0,just display “0”
    if(n==0){
        delete score[numdigits];
    score[numdigits] = new Displayable(digits[0]);
    score[numdigits]->moew(Point((700-numdigitrs*19),40));
    score[numdigits]->draw();
    numdigits++;
    }
    while(n){
                int rem=n%10;
                delete score[numdigits];
                score[numdigits]= new Displayable(digits[rem]);
                score[numdigits]->move(Point((700-numdigits*18),40));
                score[numdigits]->draw();
                n /= 10;
                numdigits++;
    }
}
```

控制流图画在图 11。6 的左侧。我们可以通过给每一个菱形或矩形分配一个节点然后用边把这些点连接起来而重新画图。其结果是由 n 个节点和 e 条边的图；在我们给出的例子里，n 是 6，e 是 8 。图形理论的一个结果告诉我们图形里的独立的路的条数为：

e-n+2

或者在我们的例子里为 4。McCabe 证明圈数也等于代码里的结果表达式的数量加 1。如果我们。我们看一下前面的代码段，我们看到两个 while 语句和一个 if 语句，因此圈数应该是比 3 大 1，即 4。一个简单的通过图形计算圈数的方法是看一下图形怎样分成小块的。在我们的例子里，右边的图形分成三个部分（三角形，半圆，不规则三角形）加上一个页面余下的额外的部分。因此，这个图形分成四个部分；这个数字就是圈数。

类似地可以通过组件的设计来计算，因此圈数常常用于衡量几个设计的选择，在编码开始之前。圈数在许多其他的条件下也是很有用的。它告诉我们有多少条路需要进行测试从而给我们提供了路径覆盖，因此它常常在测试策略中度量和使用。

在测试中，独立的路径数或者说条件路径加 1，告诉我们当我们检查和改变组件时我们需要理解多少东西。这样，许多研究人员和开发人员发现关注改变的结果或对组件或系统圈数的更正是非常有用的；如果这个改变或者圈数导致了圈数的戏剧性的增加，那么维护人

员可能要重新考虑改变或修补的设计。事实上，Lehman 的关于软件解决方案的第二条法则预测当系统增大时其圈数会增加。我们在使用这一点或其他的测量方式来代表所有软件的复杂性时必须谨慎。条件或其分支的增加使代码难以读懂。但是还有其他的源于复杂性的性质但不决定于代码结构。例如，一个面向对象的程序的继承层次可能是非常复杂的，我们在研究一个模块时其分支是很复杂的。研究人员不断地寻找更好的方式来定义这个复杂度，帮助我们使需求更为简单，系统更易于维护。

其他的产品测量。 有很多帮助我们理解可维护性并预测相应的问题根源的产品特性。一些机构使用基于组件例如大小的规则。例如，Moller 和 Paulish(1993)在西门子展示了小的组件出错的比率更大(注解 11．4)。其他的研究人员使用了嵌套深度的信息即操作符的数量，以及扇入和扇出来预测维护质量。注解 11．5 描述了在 Hewlett-Packard 用于产生可维护性标示的方法。

注解 11．4　故障行为模型
Hatton 和 Hopkins(1989)考察了 NAG Fortran 科学运算子程序库，包括 1600 个过程共 250000 行可执行代码。该库在 20 年内发行了 15 个版本，因此有一个广泛的历史供查。他们惊奇地发现小的模块比大的模块含有更多的错误。 Hatton 继续从其他研究者那里找到证明。他注意到 Moller 和 Paulish 在西门子报告了同样的现象，这里的大小是用行数来衡量的。Withrow(1990)找到了 Ada 的相同的行为。 然而，Rosenberg(1998)指出这些报告都是根据大小与错误密度的比较得出的。因为密度是用错误除以大小得到的，大小都是比较的因素。因此这两个因素之间与实际的错误和大小的关系之间有很强的负相关。Rosenberg 提醒我们在我们使用统计技术进行我们理解得还很有限的测量时要非常小心。

注解 11．5 HEWLEFT-PACKARD 的维护测量
Oman 和 Hagemeister(1992)建议可维护性可以使用三维建模：控制结构，信息结构以及系统维护时的排版、命名和注释。他们为每一维都定义了刻度，然后把他们结合到整个系统的可维护性索引里。 这个可维护性索引被 Hewlett-Packard(HP)的 Coleman et al.(1994)用于衡量一些软件的可维护性。首先，这个索引与大量的刻度校准，使用圈数，代码行数，注释数计算出一个合适的多项式。然后，这个多项式用户含有 236000 行的 C 代码的 714 个组件。这个分析产生可维护性排序的组件列表，帮助 HP 公司定位难以维护的组件。其结果与 HP 维护人员的感觉一致。这个多项式也可以用于比较两个大小、模块数，平台和开发语言相似的软件系统。其结果证实了 HP 工程师的感觉。在接下来的几个分析中，这个多项式与维护人员的直觉仍然一致。但是这个测量提供了额外的信息，这个信息支持了自己做还是购买的决定，确定预防和完善的模块，评估重新工程的效果。

Porter 和 Selby(1990)使用叫做分类树分析的统计学方法来区分那些最好的对在维护中可能遇到的接口错误的预测。从对数据分类分析而得出的判定树暗示了基于历史纪录的可测量的限制。

- 在设计时进行了 4 到 8 次修改而且至少 15 处数据约束表明接口可能出错。
- 接口错误可能发生在主要功能是文件管理并且在设计时已经进行了 9 次修改的组件里。

这些提议仅仅是对特定的数据集的，不会成为任何团体的通用的指导方针。但是，这个技术可以用于任何数据库的信息衡量。

对于本文中的产品，可读性影响了可维护性。最著名的可读性测量方法是 Gunning 的

$$F = 0.4 \times \frac{\text{单词数}}{\text{句子数}} + 3\text{个或多个音节词的百分比}$$

迷惑指数，F，定义为：

提出这些测量是为了大略地与一个在学中的学生需要的轻松地理解这一段落需要的时间相关联。对于大型文档，常常用文本的样本进行相应的测量。

其他的可信的测量方法仅仅是对软件产品的。De Young 和 Kampen(1979)定义源代码的信度系数 R 为：

R=0.195a-0.499b+0.13c

其中 a 时代码中变量名的平均长度，b 是包含该表达式的代码行数，c 是 McCabe 定义的圈数。这个公式源于个人评价的可读性的数据的衰退分析。

11.5 维护技术和工具(Maintenance Techniques and Tools)

一个降低维护工作量的方式是在开始时就注意质量。尽量把好的设计和结构加入到原有的在开始没有正确地设计的系统里。然而，除了良好的习惯，还有几个别的加强理解、提高质量的技术。

结构管理(Configuration Management)

维持改变莲和他们对于系统其他组件的影响不是一件很容易的工作。系统越复杂，一个改变影响的系统组件就越多。由于这个原因，在开发阶段很重要的结构管理在维护中是紧急的，或者说是很重要的。

结构控制部(Configuration Control Board)。因为很多维护的改变是客户或者用户提出来的，我们建立一个结构控制部来俯瞰改变的过程。该部门包括所有对它感兴趣的部门的代表，包括顾客，开发人员，用户。每个问题都是用下述方式处理的：

1. 由用户、开发人员、顾客提出的问题，他们以形式的改变控制方式纪录这些症状。作为选择，顾客、用户或者开发人员需要进行系统加强：定义新函数、旧函数的变异，或者删除现有的函数。它的形式，与我们在第 9 章中讨论的错误报告类似，必须包含系统工作方式的信息，问题或者加强的性质，以及预期系统怎样工作。
2. 建议的改变报告给结构控制部。
3. 结构控制部开会讨论这个问题。首先，讨论这个建议是否违背需求或系统增强的需要。这个决定将影响谁将支付进行改变需要的资源。
4. 对于报告上来的故障，结构控制部讨论可能的错误根源。对于要求的系统加强，该部讨论这个改变可能影响系统的那些部分。在每一种情况下，程序员和分析员会描述改变的范围以及预期的实现这一改变需要的时间。控制部给各个需求分配优先级或者紧急程度，程序员和分析员负责进行适当的系统改变。
5. 指派分析员或程序员定位问题根源或者改变涉及到的系统组件。使用一个测试副本进行工作，而不是现在正在运行中的版本，程序员或分析员实现并改变他们以确保系统能正常工作。
6. 程序员和分析员与程序管理人员一起控制改变后的系统的安装。所有的相关文档进行更新。
7. 程序员和分析员书写报告详细描述这些改变。

改变控制(Change Control)。 最重要的过程是步骤 6。在任何时候，结构管理组必须知

道系统的任何组件和文档的状态。因此，结构管理应该强调对于系统产生影响的人员之间的通讯。Cashman 和 Holt(1980)建议我们始终要知道以下问题的答案：

- 同步：什么时候发生改变？
- 鉴定：谁进行的改变？
- 命名：系统的哪些组件被改变了？
- 证明：改变是正确的么？
- 授权：谁授权进行相应的改变的？
- 安排：谁通报了这一改变？
- 取消：谁可以取消这一改变的要求？
- 委托：谁对这个改变负有责任？
- 评估：这个改变的优先级是什么？

注意这些问题是管理问题，而不是技术问题。我们必须采用相应的步骤来小心地管理这些改变。

我们可以用下面的几种习惯来帮助管理这些改变。首先，给每一个工作版本分配一个唯一的鉴定号码。当版本发生变化后，给每个改变了的组件分配一个修改后的号码。我们保留每个组件的版本和状态纪录，以及所有的改变的历史纪录。然后，在生命周期的任何点上，结构管理组可以区分当前的使用系统的版本和使用的各个组件的版本。他们还可以找出不同版本之间的不同之处，谁进行的这些改变以及为什么进行改变。

从学生的观点来看，这些结构管理习惯可能是不必要的。你们的工程可能是单独管理的或者由一个很小的程序员的组织管理的，使用口头的通讯来跟踪修改和增强。然而，想象一下由于对 200 个组件的程序使用相同的技术而引起的混乱吧。常常，大系统是由几个独立的小组同时在系统的不同方面进行工作的，有时这些系统存放在城市的不同地方或者甚至在不同的城市里。当不当的通讯引起系统故障时，结构管理小组必须能恢复系统到前面的稳定的情况下，这个仅仅当小组知道谁、什么时候对系统的那些组件进行了什么样的改变时才能正常进行。

影响分析(Impace Analysis)

传统的软件生命周期把维护描述成在软件投入使用后开始的。然而，软件的维护依赖于并且开始于用户需求。因此，好的软件开发法则将用于开发和维护过程。因为好的软件开发支持软件的改变，改变是在软件产品的生命周期中必须考虑的问题。而且一个看起来不重要的改变常常比预料的范围更广。影响分析是对许多改变可能产生的冒险的评估，包括对资源、工作量、工作计划的影响的评估。

多个部分的改变的影响可以在形成的不适当的过期的文档中找到，如对软件的不恰当或者不完全的修补、设计或者编码的结构不清晰、不符合标准的自己造的东西等等。问题随着复杂性的增加、开发人员理解改变的代码需要的时间的增加以及代码的改变对系统其他部分的负面影响的增加而复合化。这些问题增加了维护的开销，管理倾向于使这种开销易于控制。我们可以使用影响分析来控制维护开销。

Pfleeger 和 Bohner（1990）研究了测量计划的改变的影响以判断其风险度以及权衡几种选择。他们描述了一个包含了可测量的反馈的软件维护模型。图 11．7 描述了当需要进行改变时要进行的活动，图中标有箭头的地方表示提供管理者用来判断什么时候以及怎样进行改变的信息。

有效产品是其改变是有效的开发中形成的东西。因此，需求、设计和代码组件、测试用例以及文档都是有效产品；其中一种产品的质量能影响到其他产品的质量，因此改变它可能

产生重要的结果。我们可以评估改变对所有的有效产品的影响。对于每一个有效产品，纵向可追溯度表达了有效产品各部分之间的关系。例如，需求的纵向可追溯度描述了系统各项需求之间的依赖关系。横向可追溯度强调一个有效产品集中各部分之间的关系。例如，每一个设计组件被转化为实现这一部分的代码组件。我们需要这两种可追溯度来理解在影响分析中得到评估的关系的全集。

我们可以使用有向图来描述纵向和横向可追溯度 。这个有向图包含了一些对象（称之为节点），以及相关联的有方向的节点对（称之为边）。有向边表示有效产品内部或有效产品之间的关系。

图 11．8 描述了相互关联的有效产品之间的图形化的关系和可追溯性连接是怎样形成的。我们检查没一个需求并且在需求和实现这一需求的代码组件之间画一个连接。依次地，我们把每一个设计组件和实现它的代码组件连接起来。最后，我们用测试这些模块的测试用例连接每一个代码模块。最后就形成了描述有效产品之间的关系的有连接图形。

图 11．9 描述了整体的可追溯图。每一个主要的人为过程表示成一个包含了子节点的盒子。盒子内的实线边是盒子里组件的纵向联系。盒子之间的虚线边表示系统的横向联系。注解 11．6 描述这种方法怎样用于 Ericsson 的。

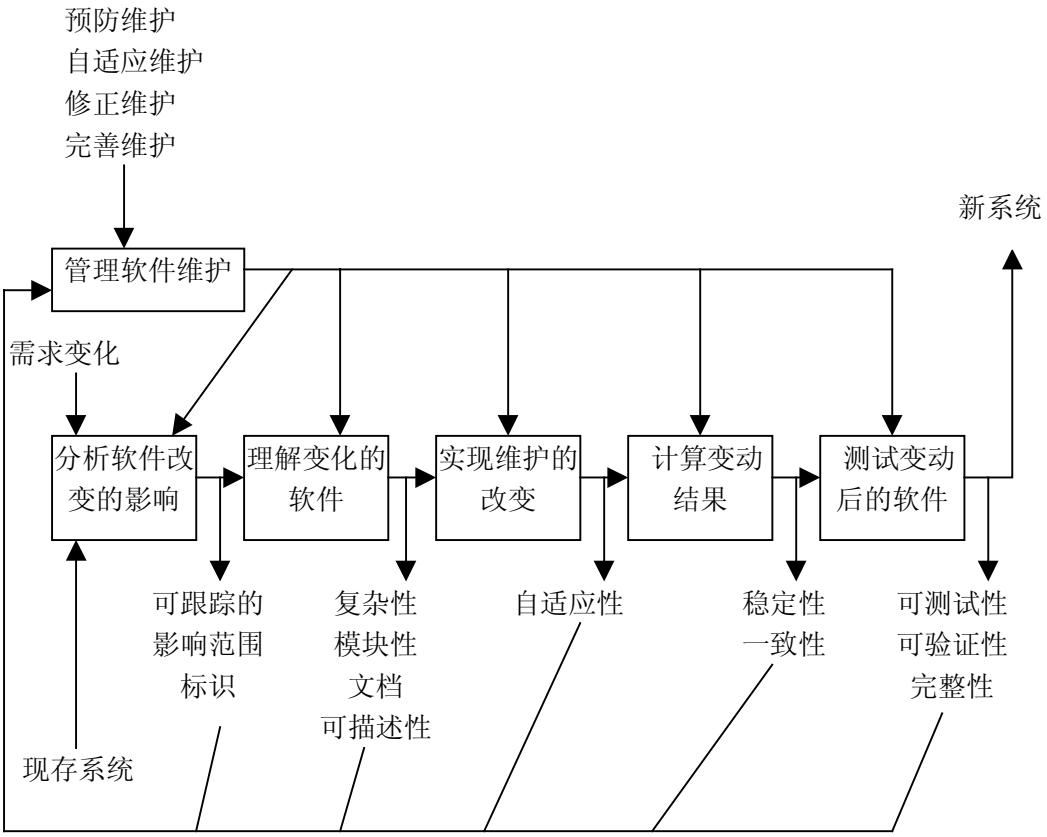

图 11.7 软件维护活动

注解 11．6 把可追溯度用于现实世界
Lindvall 和 Sandahl(19996)将 Pfleeger 和 Bohner 的可追溯度模型用于 Ericsson Radio System 的一个面向对象开发的工程。结果，他们建立了可维护度的二维模型。第一维描述了跟踪的项目。例如，他们实现了 5 种可追溯度：

- 对象与对象
- 关系与关系
- 用例与用例
- 用例与对象
- 二维对象与对象

第二维描述了怎样进行追溯：

- 使用清晰的链接
- 使用不同文档的文本上的引用
- 使用相同或者相似的名字和概念
- 使用系统知识和领域知识

这个链接的形成过程导致了问题和改正方法的发现，以及系统许多方面含义的清晰化。他们从他们的以下研究中得出结论：可追溯性在工程的开始是不是重点的质量因素，文档是不是会更清晰更坚固，能不能获得个人设计的更好的理解，系统的输入是不是更集中，维护是不是更少地依赖于个别的专家。然而，他们的可追溯性的度量工作需要很多工作 Lindvall 和 Sandahl 认为在以下两种情况下需要重要的努力：

- 在没有任何追溯链接的工具的条件下进行追溯。
- 在部分不固定的或者没有建立文档的模型中进行追溯。

有很多证据证明一些复杂性的度量是工作量和错误比率的良好的指示器。这些纪录可以扩展成可追溯性图以评价一个建议的修改产生的影响。例如，我们看一下图 11．9 中每一个盒子里的纵向联系。节点的总数、边的总数、以及类似圈数等测量数据，可以在改变前和改变后进行评估。如果图形的大小和复杂度看起来随着改变而增加，那么相关的有效产品的大小和复杂度也会相应的增加。使用这个信息，结构管理部可能决定用不同的方式实现或者根本就不进行实现。即使管理者决定实现建议的改变，潜在的危险也会由于这个基于测量的图形被更准确地理解。

纵向可追溯性测量是把改变结果反馈到维护的有效产品的产品测量。横向可追溯性图形的性质的度量代表了一个程序全部的改变过程。对于每一对有效产品我们可以形成他们之前的子图：一个是相关的需求和设计，另一个是相关的设计和编码，以及相关联的代码和测试用例。然后我们测量大小和复杂性的关系以测定不利的影响。而且，我们可以观察横向可追溯性总图看看在改变后总图是不是变得更加复杂或者简单了。Pfleeger 和 Bohner（1990）关注了图形范围的最小路径集；如果改变后覆盖路径变多了，那么系统就会更难以使用而且难以维护。类似地，如果节点的入度和出度明显增加，将来系统会更难以维护。

自动维护工具(Automated Maintenance Tools)

跟踪系统所有组件的状态和测试是一项艰难的工作。幸运的是，有一些可以帮助我们维护软件的自动化工具。我们在这里叙述部分工具；在本书的主页中有这些工具的卖主的主页链接和示例。

文本编辑器(Text Editors)。 文本编辑器在维护的很多方面都是非常有用的。首先，文本编辑器可以从一个地方拷贝代码或文本到另一个地方，防止复制时出错。其次，正如我们在第 9 章中看到的那样，一些文本编辑器提供可以跟踪从存于另一个地方的源文件的改变。许多文本编辑器还提供了文本入口的时间和日期标签用于把当前版本的文件恢复成旧版本的文件。

文件比较器(File Comparators)。 维护中的一个很有用的工具是文件比较器，它能够比较两个文件并报告他们的不同之处。我们常常使用它来保证两个系统是相同的。这个程序读出两个文件并指出其中的不同之处。

编译器和链接器(Compilers and Linkers)。 编译和链接器包含了简化维护和配置管理的特性。编译器检查代码和语法错误，用多种形式指出错误之处和错误的类型。一些语言的编译器，例如 Modula-2 和 Ada ，还检查分离编译的组件的一致性。

当代码被完全地编译了后，链接器把需要的其他的代码与之链接以运行程序。例如，一个 C 语言或者能识别子过程、库、和宏调用的链接器把 filename.h 与它所相关的 filename.c 链接起来，自动地选择必要的文件形成一个整体。有些链接器可以区分每个需要链接的组件的版本号，以便选择合适的版本来链接到一起。这种技术帮助排除测试系统时由于系统的错误的拷贝或错误的子系统而引起的问题。

调试工具(Debugging Tools)。 调试工具通过以下方式帮助我们进行维护：单步跟踪程序的运行逻辑，检查寄存器的内容和内存区的内容，指定标志和指针。

交叉引用产生器(Cross-Reference Generators)。 在本章前面，我们注意到了可跟踪性的重要性。自动进行系统生成和交叉引用生成为开发组和维护组进行系统修改提供了更强的支持。例如，一些交叉引用工具作为系统需求的仓库，也存储与每个需求相关的其他系统文档和代码。当我们提出对需求的改变时，我们可以使用这个工具告诉我们哪些其他的需求、设计和代码组件将受到影响。

一些交叉引用工具包含了一系列的称之为垂直条件的逻辑公式；如果所有的公式为真，系统满足产生它的要求。这个性质在维护中特别有用，确保我们改变的代码仍然遵守它的说明。

静态代码分析(Static Code Analyzers)。静态代码分析计算代码的结构属性信息，例如嵌套深度、覆盖路径数、圈数、代码行数，不可达的表达式。我们可以在我们维护系统建立新的版本时进行这些信息的计算，看看他们是不是变大了，变复杂了，变得难以维护了。这个测量帮助我们在几种可能的选择中选哪一个，尤其是当我们设计现存代码的一个部分时。

结构管理仓库(Configuration Management Repositories)。 没有控制改变的信息库，结构管理是不可能的。这些仓库存储问题报告，每个问题的信息，报告的小组，修改的小组等。有些仓库允许用户在他们使用的系统中给报告的问题加标签。其他的，例如在注解 11．7 中描述的工具，实行版本控制和交叉引用。

注解 11．7　PANVALET

Panvalet 是 IBM 主机上常用的工具。它合并了源代码，对象代码，控制语言，以及运行系统需要的数据文件。文件被分配成不同的类型，不同的文件可以相互关联。这个性质允许开发员在文件中扩充字符串，在给定文件类型的所有文件中，或者整个文件库。

Panvalet 不只控制一个系统版本，因此文件可以有多个版本。一个单独的版本被指定成产品版本，任何人都不能扩充它。为了修改这个文件，开发员需要生成文件的的新版本然后改变这个新的文件。

这个文件是按照层次组织的，而且这些文件是交叉引用的。每个版本的文件与这个版本相关的目录相联系：产品版本的状态关系，最后修改或者更新时间，文件中的表达式数，对文件进行的最后的操作的类型。当文件被编译时，Panvalet 自动地把版本号和最后修改时间放到编译列表和对象模块中。

Panvalet 也有报表、备份、恢复功能，加了三级安全访问控制。当文件长时间不被使用时，Panvalet 自动压缩他们.

11.6 软件重组(Software Rejuvenation)

在很多有大量的软件的部分，维护这些系统是一个挑战。为了看一下其原因，考虑一个提供新的人身保险的保险公司，。为了支持这个产品，公司开发软件来处理保险政策、政策制订信息，以及账目信息。这种政策可能持续 12 年，有时这个软件不能废弃，直到过去的政策死亡，新的所有权产生。结果是，保险公司要支持多种系统平台的多种语言开发的软件。其选择可能是增强系统或完成新的系统；每一个选择都是要保持或提高软件的质量。

软件重组通过试图增加现有系统的整体质量强调这一维护的挑战。它回顾系统的有效产品从中获取更多的信息或者重新把他们格式化成更容易理解的方式。软件重组需要考虑几个方面，包括：

- 重建文档
- 重新构造结构
- 回退工程
- 重新进行工程化

当我们为系统重建文档时，我们对源代码进行静态的分析，产生新的信息来帮助维护人员理解和参考源代码。这种分析不会转化为实际的代码；它只产生信息。然而，当我们重新构造结构时，我们实际上通过转变非结构化的代码为结构较好的代码而改变了代码。这两种技术主要集中在源代码上。要对系统回退工程，我们回过头来从源代码到先于它的产品，重新进行设计和说明信息到编码。更宽泛地讲就是重新进行工程化，这里我们回退现有系统的工程然后进行软件工程过程以改变需求说明和设计，完成完整的系统逻辑模型；然后我们从修订的说明和设计出发生成一个新的系统。图 11．10 说明了这四种重组之间的关系。

当然，想要从一个给定的源代码重新生成所有的有效产品是不可能的。这个工作与从成人的绘画作品中重新产生儿童作品类似。然而，一些有效产品的关键的性质可以进行加强或者修饰。信息可以从最终产品提取的程度取决于下面几个因素（Bohner 1990）：

- 使用的语言
- 数据库接口
- 用户接口
- 系统服务接口
- 与其他语言的接口
- 该领域的成熟程度和稳定程度
- 可用的工具

维护人员的能力，知识面和经验也在信息可以被成功地解释和使用的程度起到巨大的作用。

重新编写文档(Redocumentation)

重新编写文档包括从代码的静态分析到产生系统文档的工作。我们检查使用的变量，调用的组件，控制路径，组件大小，调用参数，测试路径以及其他帮助我们理解代码能做什么和不能做什么的工作。由静态代码产生的信息可能是图形的或者文本的。

图 11．11 说明了重新编写文档的过程。典型地，维护人员从向分析工具提交代码来开始文档的重新编写工作。这个输出包括：

- 组件调用关系
- 类的层次
- 数据接口表

- 数据字典信息
- 数据流表和图
- 控制流表和控制流图
- 伪代码
- 测试路径
- 组件和变量的交叉引用

这些图形的、文本的和表格的信息可以用于评价一个系统是不是需要进行重新构造。然而，既然在需求说明和重新构造的代码之间不存在一一映射，其结果文档反馈给现在的而不是将会有的。

重新组织结构(Restructuring)

我们重新结构化软件，以使它更易于理解和改变。有工具通过解释源代码并从内部表现它来帮助我们实现这个任务。然后，使用转换规则简化内部表示，其改变以结构化的代码形式出现。尽管一些工具之产生代码，其他具有产生这样结构、版本、复杂性以及其他信息的支持函数。这些测量用于决定代码的可维护性和评价重构的结果。例如，我们希望复杂性测量在重构后降低。

图 11．12 描述了重构包含的三个活动。首先，静态分析提供了我们用于以语义网或有向图表示代码的信息。这种表示不必对人们易读，它常用于自动化工具。。

其次，这种表示在接下来的基于转换技术转化下精化。最后，精化的表示被解释并用于产生系统的结构化的等价的代码。

逆向工程(Reverse Engineering)

回退工程象文档的重新编制一样通过它的代码向软件系统提供需求说明和设计信息。然而，回退工程要做的工作更多一些，复原基于软件说明和设计方法的软件工程信息；这个信息存于一个允许我们操作的表单中。提取出来的信息不必是完全的，因为许多源代码组件常常与一个或多个设计组件相关。由于这个原因，回退后工程化的系统可能含有的信息量比原始系统的信息要少。

多亏了有图形化的工作站和存储管理工具，回退工程的大部分工作可以自动进行，这些工具解释结构和命名信息并且用文档重建相同的方式构造输出。标准的结构化分析和设计方法是实现回退工程中的信息，如数据字典，数据流，控制流，以及实体联系图等的结合的好的通讯机制。

回退工程的关键是从具体的代码实现中抽象出需求说明的能力。然而，在回退工程可以被广泛地使用以前存在一些障碍。实时的系统代表了一个问题；实现于设计的联系很小，因为频繁的运行优化。第二问题发生在以简洁的或不可理解的约定实现一个复杂的系统时。当回退工程工具用于这些类型的系统时，模型信息的价值很小。

当期望值很小时，回退工程时成功的。也就是说，这些工具在判断所有的相关的数据元素和对特定的组件的调用上做得很好。他们能演示复杂的系统结构，能识别矛盾或与设计标准的违背。

再工程(Reengineering)

重新工程化是对回退工程的扩展。然而，回退工程抽象信息，重新工程产生新的软件代码但不改变整个系统的功能。图 11．14 表明这个过程的步骤。首先，使用现有的说明方法和设

计方法对系统进行回退工程并为人和计算机重新生成其内部表示。其次，矫正并完成软件系统模型。最后，针对这个新的需求说明和设计生成新的系统。

重新工程化的输入包括源代码文件，数据库文件，界面生成工具以及类似的系统相关的文件。当这个过程结束后，他产生所有的系统文档，包括说明，设计和新的代码。

由于在不远的将来全自动的重新工程是不可能的，这个过程必须包含一系列的转换和人的交互。我们可以手工地完成这个要求，一个有经验的设计者能在新系统产生之前增强系统的设计。注解 11．8 讨论了对系统进行重新工程的工作，以及哪些需要手工处理。

软件重组的未来(The Future of Rejuvenation)

由于软件维护作为一种新的软件开发方法并不总是吸引人的，类似重组的话题并不能引起很多注意。然而，一些改进更多地注意了软件的重组。商业的回退工程的工具部分地解决了软件系统的设计问题；他们可以区分，表示和分析源代码信息，但是不能进行重构，表达在源代码中不明确的设计的抽象。

注解 11．8　重新工程化要做的工作
美国国家技术和标准研究会(NIST) 研究了 COBOL 源代码的 13131 行代码的重新工程化的结果。这个系统，包括一批没有任何商业作用的过程，被使用自动转换工具进行了重新工程。Ruhl 和 Gunn(1991)报告说整个重新工程过程需要 35 人月。 Boehm et al.(1995)指出原始的 COCOMO 模型估计需要 152 人月来进行重新工程化相同类型的系统，显然不可接受。结果是，COCOMO II 被以其他的重新工程的方法修改了，以使它包含了能自动转换的因素；这个模型计算出自动转换以 2400 行每人月的转换率发生。 NiST 的研究表明可以被自动转换的代码量随着系统类型的不同而不同。例如一批过程的 96%可以被自动转换，而带有数据库管理系统的一批过程只有 88%可以自动转换。相反，Ruhl 和 Gunn 发现交互式的应用只有 50%可以自动地转换。

源代码信息不包含很多关于原始设计的信息，因此剩下的部分就必须通过推测来重新构建。因此，在例如信息系统等被广泛理解的领域最需要小心。这里，典型的系统是标准化的，语言非常简单而且结构很好，并且有很多该领域的专家。

在其他的领域中，设计恢复仅仅对代码信息，现有设计文档，个人经验和问题领域的知识是可能的。在能够理解和重构之前，需要对于问题领域的非正式的语言知识。因此，当技术和方法能够把握规则、策略、设计结果、术语、命名习惯以及其他的非正式的信息时，软件重组将会得到加强。正如我们将在 12 章看到的那样，死亡检查会帮助我们纪录这一信息。

同时，设计符号的规范化和对领域内模型的介绍将会扩大维护系统的可用信息的量。我们可以预期转换技术的加强以支持更多的应用领域，而且更完整的表示允许更多的重新设计自动化。

11.7 信息系统示例(Information system Example)

让我们看一下 Piccadilly 软件与现实世界的关系，判断一下 Piccadilly 是 S-,P-或者 E-系统。如果 Piccadilly 是一个 S 系统，它的问题应该是完全定义的；因此系统是静态的而且不易适应产生它的问题的变化。然而，显然问题本身会常常发生变化。例如，英国政府颁发的广告调节税可能在通过新的立法时发生变化。或者电视公司的价格策略会发生变化。因此软件不是 S-系统；它的不变性将会要求在现实世界的条件变化时生成一个全新的系统。

如果 Piccadilly 是一个 P-系统，它的解决方案应该是对问题的抽象。事实上，任何判断广告时间的成本的方法基于电视节目的不同性质，包括在一天中的时间，在一周的那一天，以及其他的广告的数量。然而，P-系统要求一个稳定的抽象。也就是说，系统不会发生变化；只有系统模型的信息会发生变化。显然，我们的系统中模型可能发生变化，因为会出现新的广告政策。

在 S-或 P-系统中，现实世界的条件是稳定的，而 E-系统随着现实世界的改变而改变；这个系统是它所模拟的现实世界的一个内部的部分。显然这适合 Piccadilly 软件，因为特定广告策略的成功会影响模型本身。在现实世界和软件的抽象之间有不断的联系，因此 Piccadilly 是 E-系统。

与维护相关的显然是设计的关系。初始的 Piccadilly 的设计一定是合适的，设计部允许系统变化时系统变坏。而且电视公司的主管可能要通过模拟能力增强 Piccadilly，以便于他们能计划策略的改变。

11.8 实时的例子(Real-Time Example)

可维护性根据出错的平均间隔时间测量，因此减少错误是维护小组的主要任务。一种强调这个目标的方式是细察系统出错策略。也就是说，我们必须问自己已经假设了最好的解决错误的方式，并且检查我们是否降低了出错率。

Ariane-5 发生后调查小组指出开发应该致力于减少随机的错误。也就是说，给习惯于参考系统的开发员的指导是如果发生了错误就中止运行。当这个系统出现错误，这个错误是由于设计的错误产生的，而不是一个随机的错误，因此根据要求停止运行是正确的，但是根据根据　任务的实时性是不当的。

调查部在报告中强调了这一点。报告中说：

> 这个异常被检测到了，但是没有处理好，因为观点被限于软件是正确的，直到软件出现了错误。调查部有理由相信这个观点在 Ariane-5 软件设计的其他部分也是存在的。该部支持相反的观点，软件应该被看成是有错的，直到它在实际的使用中确实没有出错。

因此，Ariane 软件的下面最紧要的一步是改变这个错误的策略实现一系列的防卫措施。事实上，该部明确地描述了这个任务：

> 这意味着这个软件必须在一个非常具体的水平上进行区分，它的异常处理必须受到限制，而且合理的备份策略必须考虑出错的情况。

Ariane-5 也向我们提供了一个测试和改变控制的困难的例子。显然欧洲航空代理处不能在每次软件需要进行测试时就发射一枚火箭，因此测试维护需要在一系列复杂的模拟下测试新的改变的影响。

11.9 本章对你的意义

本章向你介绍了软件的维护的关键技术。我们看到了软件维护怎样解决技术上的和与人们相关的问题的。维护时不仅仅要理解软件的现状，还要了解他以前和以后是什么样子的。主要的课程如下：

- 系统与现实世界的联系越紧密，就越容易改变从而难以维护
- 除了软件的开发，维护还有很多工作。他们不断地与顾客和用户打交道，他们必须理解商业需求以及软件的开发技术。他们还需要是好的侦探，能仔细地测试系

统查出源代码中的错误。

- 软件的测量是复杂的。为了得到软件的真正的可维护性的度量结果，我们必须衡量系统的外部特性同时跟踪系统的错误。但是等到系统出错就已经太晚了，因此我们使用代码的内部性质，例如大小和结构，来预测将会出错的系统的某个部分。我们使用静态代码分析来帮助我们区分过程。
- 影响分析建立并跟踪了需求、设计和编码以及测试用例之间的联系。他帮助我们评价系统的一个组件的变化给其他组件带来的影响。
- 软件重组包括重新设计文档、重新结构化、回退工程化、以及重新工程化。总的目标是使隐藏的信息明朗化，因此我们可以用它来提高设计和代码结构。尽管完全的重组在最近的将来不大可能实现，它已经在一些成熟的的领域中成功地运用，例如信息技术。

11.10 本章对你的开发小组的意义

维护完全是团队行为。需要大量的协调检验组件是否合格，改变并测试组件，把修订过的组件放回正在工作的系统中。而且许多故障易由于组件之间的复杂的交互引起的，因此，你必须与你的小组成员交换信息以获得关于软件怎样与环境融合的信息。

你的与人交往的能力在维护中非常重要。当你找到了问题，你必须告诉你的伙伴、用户，顾客以及其他具有不同工作角色的人。因此你必须学会怎样提取你需要的信息，使用你的工作方式。

11.11 本章对研究人员的意义

维护是研究的一个成熟的领域。我们许多的维护活动可以随着我们对错误根源的预测能力的增强而变得更容易、更有效。研究人员正寻找更好的基于产品信息测量可维护性的方式；他们正开发新的模型向我们展示产品的，过程的和资源的联系。类似地，这个模型也会帮助我们理解维护一个系统需要花费多大的精力。

我们的工作继续在建立帮助我们进行维护工作的工具上。重新开发的工具，改变的结构管理库，以及工程历史数据库在用户建立基于经验数据建立原型时变得更成熟。

最后，研究人员将继续关注软件维护的普通法则。他们渴望知道软件工程理论是否能巩固实际中的经验观点：软件系统的改进是一致的和可预测的。

11.12 小组的目标(Team Project)

看看你的为了借贷管理做的所有的手工产品（需求，设计，编码，测试计划，文档）。他们的可维护性有多强？如果你又一次设计并实现了借贷关系系统，你必须做那些不同的事儿以使产品易于维护？

11.13 主要参考资料

有很多关于软件维护的最新的书本；大部分来自杂志和会议记录。*IEEE 软件杂志* 1990 版中含有维护，回退工程，和设计恢复的只是；1995 版集中于遗留系统，1993 版有

Wilde,Matthews 和 Huitt 关于面向对象的系统的特殊的维护问题的好文章。*Communications of the ACM* 的 1994 年 5 月刊是一个关于回退工程的特刊。*软件维护：研究与实践* 是一个完全是讨论维护的刊物。

IEEE 计算机学会提供了一些关于维护的好的指南，包括 Arnold 的一个关于软件重新工程化的文章，另一个是 Bohner 和 Arnold 关于影响分析的文章。

Samuelson 研究了回退工程的合法的暗示，询问这个实践是否等于窃取他人的观点。

国际软件维护会议每年召开一次，由 IEEE 和 ACM 主办。你可以从 IEEE 计算机学会订阅这个会议记录。

11.14 练习题

1. 把下列系统分成 S-,P- 或 E-系统。对于每个系统，解释为什么属于这个分类。指出系统的那些方面会发生变化。
   a) 空运控制系统
   b) 微机的操作系统
   c) 符点运算加速系统
   d) 数据库管理系统
   e) 找出一个数的素数因子的系统
   f) 找到大于给定的数的最小素数的系统
2. 解释组件之间的高耦合可能使系统维护非常困难
3. 解释为什么系统的成功与否很大程度上依赖于系统开发是产生的文档的质量
4. 一些计算机课包括建立一个开始很小的系统并且不断地增强这个系统直到其结果是完全的。如果你工作于这样的一个系统，回顾你的纪录。你需要多长时间理解这个问题？需要多长时间实现编码？对比你的系统和表 11.2 中的各类系统的时间估计，标注这个不同是好还是坏。
5. 解释为什么维护系统可能比开发系统更具有挑战性。为什么很多好的维护程序员具有好的“人缘”？维护程序员还需要哪些品质？
6. 从你的课程工程里选择一个大的程序。你会给程序加多少文档以使别人能维护你的系统？当你的程序结束后讨论你的文档中有多少正面的和负面的因素。
7. 从你的朋友那里借一个大的程序（多于 1000 行代码）。尽量选择你不熟悉的程序。文档有多大作用？比较代码和文档；文档有多精确？如果你被分派维护这个程序，你希望再看到什么文档？程序的大小对于你维护它的能力有多大影响。
8. 同前一个问题，检查你的朋友的程序。假设你要对程序代码进行修改，而且你必须进行衰退测试。测试数据和测试脚本对你有用吗？讨论保留正式的测试数据集和脚本对于维护的意义。
9. 解释一下为什么单入口单出口的模块使维护中的测试更容易
10. 回顾一下好的软件设计的性质。对于每一点，它是否会有助于或妨碍软件重组
11. Ariane-5 是 S-,P-还是 E-系统
12. McCabe 圈数允许我们根据组件质量形成顺序吗？也就是说，我们可以说一个组件比其他组件更复杂吗？举出不能由圈数解释的软件复杂性的几个方面。
13. 假设你正在维护一个大的安全鉴定系统。你使用了一个模型，如 Porter 和 Selby 设计的那样，来预测系统那个组件最容易出问题。然后，你仔细检查了那些系统组件，并进行了预防维护。不久以后，系统遇到了灾难性的错误，将对生命财产安全产生巨大的影响。错误的根源证明是由于你的系统模型中没有考虑的一个组

件引起的。你忽略了其他的组件是错误的吗？

14. 下面是英国代理处的结构管理工具的版本列表和控制标准。解释他们对于维护简便性的作用。
    a) 参考和版本纪录
    b) 需要时查找任何版本
    c) 纪录关系
    d) 纪录工具控制访问的和不访问的版本之间的关系
    e) 控制安全性，记录权限
    f) 纪录文件的改变
    g) 纪录版本状态信息
    h) 帮助配置一个系统版本
    i) 与一个工程控制工具的关系
    j) 产生报告
    k) 控制发行
    l) 控制自身
    m) 对常用文件建档查询

第 12 章 评估软件产品、过程和资源

在前面的章节中，我们学习了在开发与维护软件系统的过程中所进行的各种不同的行为。从工厂和政府中应用的实例中我们可以看出软件开发者们应用不同的方法和工具来确定用户的需求、设计、系统实现、测试和维护。为了从这些繁杂方法和工具中找到一种能应用于具体的现实环境下的最有效、最适当的加以应用正是本章要研究的目的。下一章将介绍基于这些技术的改进实例。

12.1 关于评估

作为专业的软件开发人员，我们通常会关心我们所生产的软件产品和我们生产他们所应用的方法。我们所熟悉的评估技术就是指估算我们的产品和产品的开发过程以及开发过程所耗费的资源，应用估算出来的结果信息来判断我们的产品是否在功能、质量以及其他方面达到了预期的目标。

本书中介绍的评估技术分成以下四类：

1. 特征分析（feature analysis）
2. 纵览（survey）
3. 事例研究（case study）
4. 形式实验（formal experiment）

特征分析

特征分析是一种形式最为简单的评估。用来将不同的产品的各种属性分类、分等级，以便于我们选择合适的工具和方法。

比如说，我们想买一种设计使用的工具，那么我们首先要列出我们所需要的工具所应该具备的五条属性。

1. 友好的用户界面
2. 具备面向对象的功能
3. 拥有一致性验证功能
4. 能够操作用户事例
5. 能够在 UNIX 系统中运行

然后，我们选定 3 个可能的工具并将每个工具的属性的重要性标准从 1 到 5 排列，如下表 table 12.1。 最后根据该表中的属性的得分结果，选择我们需要的适当的工具。
用特征分析的方法能够缩小我们选用工具和方法的范围，但是它不能评估出真正的基于原因和结果的行为。

调研（survey）

调研一般是指一种为了找到在一个给定的条件下各种因素的相互关系和最后结果而进行的反复的研究。软件工程中的调研与上述方法相类似，我们通过记录下相应的数据信息来决定项目参与者将如何对项目的某一特定的方法、工具或是某种技术做出反应。或是如何决定我们以后工作方向和协调各方面的关系。同时，我们还可以得到有关工程项目的信息来规划工程组成模块的规模、可能出现的错误的数目以及需要付出的工作量等等。

事例研究（case study）

在事例研究中，我们先确定出可以影响到一个行为结果的事物的关键性属性（输入、限定条件、使用资源、输出），然后记录下他们。事例研究侧重于从两种不同的方法的比较中选优。相比之下，形式化实验则是一种严格的基于控制的调查方式。在形式化实验中，一个行为的关键性属性先被确认，然后证明这种属性在输出中将产生怎样的影响。
事例研究和形式化的实验都包含一系列的步骤：提出概念、设定假设、设计、准备、运行、分析、推广、决策制定。其中尤其重要的阶段是设定假设阶段，它决定我们将如何去最后衡量和分析我们的结果。值得注意的是我们应该认真选择要处理的项目，选择的项目必须具备典型性。

形式化实验（formal experiments）

在形式化实验中，首先处理一些独立的变量，然后我们通过对相关变量的变化的观察来决定在输入端的变化将会对输出端产生怎样的影响。

在一个形式化的实验中，有许多种技术来防止产生有偏差的和有混淆性的结果。例如，可以通过随机选择实验主题的方法来防止产生偏差，而且我们也可以通过重复实验的方法通过得到的多样的数据集来增加结果的可信度。

形式化的实验必须经过认真的设计，这样我们观察研究的实例才会有代表性。

准备进行评估

无论我们用什么样的评估方法都有几个关键的步骤来帮助我们确定评估方向和选择适当的变量。

设定假设

当我们开始决定我们希望对什么进行调查时，我们需要将我们想要做的工作先用一种前

提假设的形式表达出来，即我们将调查什么？我们都想知道哪些东西...
例如，下面的一段话可以作为一种前提假设
“应用 cleanroom 方法可以产生比应用 SSADM 方法产生质量更高的软件。”
如果可能的话，我们在设定假设当中将尽量的多用一些可数的名词，这样可以使该假设易于证明和正否。这些可数的名词包括“错误数”等等。

确定对变量的控制

当我们确定了一个鲜明的假设之后，我们必须找出有哪些变量与该假设的正确性相关，然后对于每个变量，我们观察对该变量我们拥有多大的控制能力。
例如：假设我们想对一种程序设计方法对最后软件产品质量的影响进行研究。但是，我们无法控制谁具体使用哪种方法，然而我们可以做一次事例研究来记录结果，只有我们能够直接的、精确的、系统的掌握行为的时候实验才能够进行下去。这样，我们就可以控制谁应用 cleanroom 方法，谁应用 SSADM 方法，以及何时何地应用，来进行我们的实验。

在实验当中，我们在独立变量上抽取典型元素，因此我们可以代表所有可能的事例。但是在事例研究中，我们从相关变量中抽取元素，选择那些典型的代表参与者、组织和项目的值。举例来说，一个研究编程语言对结果产生影响的实验将选择一个项目的集合来覆盖所能覆盖的一切编程语言。相比之下，使用事例研究的方法将只选择一种在大多数的项目中使用的编程语言进行研究。

使调查有意义

在软件工程中，有许多领域可以应用调研、事例研究和实验的方法来进行分析。能够应用一个形式化的实验的方法而不是用调研或是事例研究的方法的主要因素是该实验的结果应该具备一般性的特点。即用事例话实验方法得到的结果将适用于更广泛的范围。而且，软件工程的实验并不像是化学中的实验或是生物学中的实验，在考虑我们研究的结果是否适应新的环境时，我们还必须考虑所使用控制变量的局限性和不足性。

12.2 选择评估技术

kitcheham、pickark 和 pfleeger 意识到研究使用方法的不同将同时在研究的规模上有所体现。从本质上讲，因为形式化的实验要求对属性变量有大量的控制，因此，这些形式化的实验趋向于小型化，包括小数量的人和事件。于是，我们可以把实验看成是一种小范围的研究。事例研究通常是注重典型的项目，而不是试图找到所有相关事例的信息。调研的方法则是一种对大范围的事物对象的研究的方法。
关键性选择因素

有许多种因素可以指导我们具体选择哪种评估技术，在这些因素当中，控制因素是一种关键的因素。如果我们对那些能对输出结果产生重要影响的变量有较强的控制能力的话，那么，我们就可以考虑用一个形式化的实验来进行评估。反之，如果我们没有这种变量的控制能力，我们则只有用事例研究的方法。但是，对变量的控制是一件较为困难的工作。不仅因为会相应的增加代价，而且还会冒一定的风险。
另一种要考虑的因素是我们能够复制我们正在研究的基本环境条件的程度。例如，假设我们正在考察不同的编程语言对所生成的软件的影响。那么我们是否能够每次应用不同的语言多次运行于同一个环境中呢？假如该环境是不可复制的话，那么我们就无法进行一个形式化的实验。然而，即便该环境允许复制，但是复制过程的代价也将是一种制约因素。

我们应该相信什么

通常的研究报告应该包括用事例研究、调研和形式化实验方法产生的推导信息，但是对于你来讲，辨别出哪个结果使用于你的具体环境将会是困难的。

尤其是结果产生冲突的时候，我们又怎样知道应该信任哪种评估方法呢？我们可以用下面的图板上的一系列问题来理解如何将这些研究分类。问题的答案将会告诉你什么时候你将拥有足够的信息来推出两个因素之间关系的有效推论。首先，假设你的项目小组正致力于提高代码的质量，你想找出决定质量提高的因素。第一，你想通过用计算每千行代码所产生的错误数的办法;这样一个好的系统意味着在每一千行的代码中的错误数将不超过 5 个。第二，你试图通过对人员的研究来发现还有那些因素影响软件质量。

12.3 评估与预测

评估过程通常包括度量(measurement)，我们搜集信息来辨别独立和相关变量的不同值，并且通过整理这些信息来增加我们对自身的理解。度量可以帮助我们区分典型的环境条件，确定起点与最后目标。

在形式上，度量可以看作是现实世界中的实际事物和属性到数学模型的一种映射。例如，现在我们考虑人类这个集合和人类的属性如：身高、体重和头发的颜色。然后我们可以定义映射来掌握人的属性，并保持这种关系。例如，我们可以说“Hughie 2 米高”，“Dewey 2.4 米高”“Louis 2.5 米高”等等，然后我们可以对这些符号数字用数学的方法进行分析以得到更多的信息来帮助我们反过来更好的理解现实的世界。，在我们所举的例子中，我们可以通过最后分析知道，所研究对象的人群的平均的身高为 2.3 米，我们的研究集合中有两个对象的身高在平均身高之上。

大量的关于软件的度量确定了有关产品、处理过程或资源属性的大量信息，但是研究发现确定一个适合你的目标的度量通常是困难的。因为那些可选的度量在度量和预测相同的属性时，会采用完全相异的方法。

为了理解软件度量的有效性，可以考虑下面的两个系统。

1. 度量系统：用显著的特征和属性来评估存在的实体
2. 预测系统：用来预测未来实体的一些属性

一般的讲，如果一个度量能够真正的精确刻画出它想要描述的属性，则这个度量就使有效的。

有效化预测系统

我们可以通过经验的建立系统的精确度的方法来有效化一个预测系统。即，我们在一个给定的环境中比较模型的功能和已知的数据。先声明一个关于该预测的假设，然后审查数据信息看该假设是否成立。

在有效化模型的时候，一个合理的精确度与几种因素有关包括执行该评估的人是谁。一个初级的评估者在评估的精确度上要逊于有经验的评估者。我们也可以就一个给定模型考察基于它们建立的确定系统和不确定系统之间的不同。

在一个不确定的模型中我们围绕着实际的值提供一个错误窗口，窗口的宽度是可变的。完成软件代价估计、工期时间表估计、可信度估计的预测系统会存在许多错误，我们称之为是不确定的。例如，假如你发现在某一环境条件下，你的组织的可信度预测是精确在 20%的范围内的；即预测下一次错误出现的时间的偏差不会超出实际下一次错误出现的时间的 20%。 我们再用一种可接受范围方式来描述这种窗口：一种在预测与实际值之间的的极限最

大的偏差。因此，上例中的 20%就是该模型的可接受范围。(acceptance range)。在你应用一个预测系统之前，你必须先确定你所要求的可接受的范围有多大。

当我们设计一个实验或是事例研究的时候，模型将被用来代表特别的、比较困难的问题，因为它们的预测结果将直接影响最后的输出结果。这样一来预测便成了目标，于是开发者们则有意无意的努力去满足这个目标。因此，实验评估模型有时候被设计成为双向不可见的实验。在该模型中，直到实验进行的时候才能够知道实验的目标是什么，而在这之前对于参与者这个目标是不可见的。从另一个方面讲，一些像可信度模型之类的模型将不对输出的结果产生影响。也就不会产生这样的问题。

最后要指出的是预测系统不必要建得太复杂。

有效化度量

有效化一个软件的度量与有效化一个预测系统之间存在较大的差别。在有效化度量中，我们希望度量所得到的属性值正是他所期望得到的。代表条件说明度量得到的数字值之间的关系一定要与现实世界的相应属性之间的关系相对应。假如我们度量高度，那么如果在现实世界中 James 比 Suzanne 高的话，对 James 度量得到的值一定要比对 Suzanne 度量的值要大。为了有效化一个度量，必须要求代表条件同时支持度量和度量所对应的相应属性。

12.4 评估产品

软件的开发分成若干的步骤：需求分析、设计、编码、测试实例、测试代码等等。在每个步骤中，我们能够通过检验某个阶段产品的方法来察看是否存在我们需要的属性。即，我们查询文档、文件或是系统来确定其中是否存在于诸如软件的完整性、一致性、可靠性等有关的属性信息。

产品质量模式

产品的质量模式将不同的产品质量属性相联系起来，软件产品的每种质量模式都能够帮助我们理解几种不同的属性是如何作用于整个产品的。
下面我们介绍几种软件产品的质量模式。

Boehm's Model

图 12.2。

Boehm's Model 指出高质量的软件是那些能够同时满足用户的需求与用户的期望的软件产品，而且在 Boehm's Model 中还包含了硬件功能的特征。

Boehm's Model 开始于软件的一般应用。Boehm 和他的同事们指出首先软件系统应该是有用的。否则软件开发就是一种资源的浪费。我们可以结合以下几种使用者的类型来考察软件的有用性。

第一类用户是指最初的用户。如果系统满足该用户所期望的要求，该用户则将满意该软件的使用。第二类系统的用户是在软件升级和系统改变后继续应用软件的那些用户。最后第三类用户是那些维护系统的编程人员。他们能够按照用户的需求或在使用中新发现的错误而对软件系统作相应的改动。所有三种的用户都希望软件是可靠有效的。同时系统还应该对用户的需求能够在允许的时间范围内给出结果。适时的满足用户。最后，对于用户和程序员来说，系统还应该是易学易用的。

因此，Boehm's Model 声明真正的高质量的软件应该是具备下面这些特征的软件：

* 能够完成用户想做的事
* 正确高效的利用计算机资源
* 易学易用
* 进行了优化的设计，优化的编码，易于测试和验证

ISO 9126

SO 9126 是一种在 90 年代初,集中各种软件质量方面的观点于一体的能过反映出国际范围内的衡量软件质量标准的模型。
它与 Boehm's Model 的区别在于它的体系定义是严格的，在体系定义图中，每个右边的属性只严格对应一个左边的部分。

Dromey's Model

Dromey 提出了一种建立软件质量模型的一般方法。Dromey 认为软件产品的质量大都是由所选择的组成软件产品的组成部分决定的，因此，他定义 4 种属性为现实的软件产品质量分类。

* 正确性属性
* 内部属性
* 上下文相关属性
* 描述性属性

接下来，Dromey 还提出高等级的质量属性还必须是那些用有较高优先权的质量属性。在一个实例中，他指出了 8 种高等级的质量属性：

* 独立性
* 可分离性
* 可配置性
* 面向用户性
* 易于定义性
* 确定性
* 有效性

为了使这些特征更加现时，Dromey 将这些特征属性与他设计的体系框架相结合抑或的它们的联系如图 12.4 所示。
Dromey's Model 的建立是基于下面的五个步骤：

1. 确定一个高等级的质量属性集合
2. 确定软件产品的组成部分
3. 为每个组成部分确认和分类出最重要的、现实的雨量有关的属性
4. 提出能够联系软件产品质量和质量属性的公理
5. 评估该模型，在评估中发现它的不足之处并改进它

建立基准起点和目标点

另一种评估软件产品质量的方法是用某一基准起点进行比较。基准起点描述的是在一个组织或一定的范围内的那些通常的、典型的某些结果。基准起点在管理预测期望的时候有很大的用处。当某一个值接近于该基准起点的时候，它并不向我们显示出任何异常。相反，当某些值偏离该基准起点的时候，我们将要调查一下产生该结果的原因。通常我们会找到合理的原因，但是有时的结果则需要我们对系统作相应的改动。

目标是指基准起点的变动。管理者通常使用定义最小的可接受的行为的方式来设定目

标。例如一个公司声称它将直到找出整个系统中 95%的错误时它才会最后的发布产品。

当目标能够反映出是对一个基准起点的正确理解的时候，则该目标是有意义的。该目标是可以达到的。

软件重用

软件重用要求从现在和从前的项目出发评估软件的质量，来决定是否在我们下一个将要建立的项目中它们依旧适用。

我们所建立的许多的软件系统彼此间是相似的，对于有着相似的目标的软件系统来说，这种相似的特性尤其明显。因此在我们新创建系统的时候，我们需要考虑一下过去的应用模块甚至整个过去的系统，看看是否适合我们将要创建的新系统。软件的研究人员确信，软件复用在提高软件的产量将低成本上有着巨大的潜力。

软件复用的类型

我们所说的软件复用意味着对软件系统的任意组成部分的重复使用：分析文档、编码、设计、需求、测试实例等等。而且从更广的角度来讲，软件的生成过程也可以被看作是可以复用的。

从软件复用者的角度来讲，复用分成两种类型。生产复用，用来创建复用所使用的模块；消费复用，在以后的系统中使用该可使用的复用模块。作为软件复用中的消费者，可以无改动的使用原来模块（black-box reuse）；也可以对模块加以改动适应新系统的需求（clear-box reuse）。

几种软件工程中的先进的技术正促进着软件复用的发展。OOD 技术就是其中之一。OOD 将软件复用变得容易实现，因为 OOD 允许软件工程师围绕着不变的组件进行设计，许多的设计成分和编码模块都能应用在其他的综合的系统当中。另一种软件复用的促进工具是在软件开发过程产生影响的软件开发工具组合技术的发展。一种为某一系统开发的能够允许组件转移的工具将使组件适用于更大的软件系统。

软件复用可分成合成复用和再生的复用。合成复用将复用的组件被看成是一个构建模块的集合，开发软件的过程被看成是，基于可得的软件复用模块从底部作起的组装建设过程。组件被存放在组件库中。而且组件应该是被明确的归类的。在选择组件组成新系统时还需要一个组件的检索系统。

再生复用专门服务于一个特殊应用的域。即，组建被专门设计用于满足某个应用，或是以后应用于相似的系统当中。

两种复用方式的根本的行为就是域分析。一个分析应用域的过程，用以决定所存在的共同应用的区域和描述它的方法。组合式复用通过域分析得到一个大范围系统中的一般性的低级别的功能集合。再生复用通过域分析定义一个原始的域体系结结构，同时还确定该结构组件的使用接口。

这些观点直接引导出水平复用和垂直复用的观点。垂直复用包括在同一域或范围内的复用，而水平复用则是指可以跨域的复用。

复用技术与复用检索

复用中最大的障碍就是解决从大量的软件产品集中选择适合某一需求的最恰当的产品加以利用。这个问题的解决办法就是组件的分类，根据一定的复用准则将组将进行分类规划。可以通过分等级的办法将组建分类，最高级别的下面可以继续分成低级的子级别。如图 12.6 所示。但是这种结构是固定的，不易改动的，新的分类只能简单的加在较低的级别上。

解决这个问题的方法是 Prieto-Diaz 的多面分类技术。每个组件被一组安排好的称之

为面的特征所描述。一个面是一种能够帮助确定组件的描述方法，一个面的集合能够确定出几种特征。例如：可复用代码的面可以是：

* 一个应用区域
* 一个函数
* 一种编程语言
* 一个操作系统

这种分类系统由一种自动的能根据用户需求找到适当的组件的检索系统（或叫做存储库）来支持。这种存储库还有记录用户需求的功能。对于一种新的应用原来的库中信息无法满足的需求，该库可以记录请求，并创建新的适应这种请求类型的组件。

软件复用经验

有很多的公司和组织机构已经在软件开发的过程中成功的运用了软件复用的方法。

最初的报告使用复用技术的例子是在 Raytheon。导弹系统信息区分处理系统组织发现在它的应用设计和编码中，有 60%的信息是冗余的。这就为复用提供了理想的复用源。复用程序随之被建立起来，超过 5000 个 COBOL 程序被测试并被分成了三类：编辑、更新和结果报告。Raytheon 还发现大多数的商业应用都是和这三种逻辑结构相关，这些逻辑结构是标准的。可复用组件的库也被建立起来。软件工程师被培训应用该库，他能够在当一个结构能够在新的应用中复用时做出相应的确认。用这种方式运行六年之后，Raytheon 指出创建一个新的包含 60% 复用代码的系统的效率将比原来提高 50%。

一个软件复用方面最成功的例子就是 GTE'S 的数据服务实例。它建立了一个复用资源的管理程序来达到公司内软件资源复用的目的，该管理程序从分析已存在的系统并从中找到复用资源开始，任何可以被部分或是全部重复使用的软件都看成是可复用的资源，将被分类、分目的存入复用库中。GTE 建立了一套用户自主和有回报机制的软件复用制度。使用重用软件的人应该相应的向软件的提供者做出回报（以现金购买的方式）。

软件复用的优点、代价、成功点

实践证明，软件复用技术能够在软件工程的整个开发过程中提高软件开发的质量和效率。可以缩短软件工期，相应缩短项目编码、测试等过程的时间。而且，提取的程序复用组件还可以提高软件执行的效率和可靠度。因为，在组件存进组件库之前组件就经过了严格的测试与认证，所以具有较高的稳定性。

从长期来看，复用的好处还在于能够组织系统间的相互协调的能力。软件复用当中所建立的复用标准直接导致了一套系统中各个部分之间统一接口的建立，使得建立相互协调合作的系统变得更加容易。

Pfleeger 总结出几条成功使用软件复用技术的经验

1. 复用的目标一定是可度量的，以便最后检验我们应用复用得到的结果是否能够达到我们的要求

2. 由不同角色的人制定的复用的目标可能会产生冲突，但是对于这些冲突必须尽早的解决

3. 不同的理解将引起不同的问题，对不同问题的从不同角度的回答则会反映出复用程序中不同角色的参加者的不同观点。

4. 每个组织必须确定自己的组织应该在哪个等级上提出问题和解决问题。

5. 复用过程应该在开发过程中进行集成，不然的话，项目的参与者将不知道什么时才可以复用。

6. 应该将度量与复用过程相联系，以便于你可以随时的度量你的过程块并加以提高和改进。

Pfleeger 还提出了几个复用要取得成功所必须回答的几个问题

1. 你是否拥有一个正确的复用模型？
2. 现存的模型怎样适应未来的项目集合，而不是单个项目？
3. 正规的项目总评的概念如何适应复用？
4. 谁将对复用组件的质量负责？
5. 谁将对处理过程的质量和维护过程负责？

12.5 评估过程

本书中所提到的许多软件工程实践中都包含处理过程为了以某种方式提高我们的软件产品。有些处理过程通过直接的方式影响产品如：inspections 和 cleanroom 而有些则通过间接的方式 如： configuration management 和 project management。
有些处理过程存在于整个软件生存周期中，而有些则在软件生存周期的某一小部分存在。但是这两种情况的共同点就是我们都希望我们的处理过程有效、有影响力。本章中，我们将研究几种评估这些作用于产品和开发者上的处理过程的新方法。

Postmortem Analysis（事后分析）

每个项目都是由一系列的处理过程组成的。每个过程都用于完成一个特定的任务。一种评估这些处理过程的方法就是收集大量的开发中和项目结束后的数据，然后对这些数据加以分析看处理过程是否达到了预期的目标。

事后分析是一种项目执行后对项目的各个方面包括产品、处理过程和资源等等各个方面的评估。目的是找到以后对项目进行改进提高的方向。
事后评估分成五个步骤：

1. 设计和发布项目调查来收集信息
2. 收集对象项目信息，例如：资源使用代价、程序边界条件、预定完成工期、错误数目等等。
3. 召开交流汇报会议收集调查过程中遗漏的信息
4. 与部分参与者组织项目进度回顾，来回顾项目中的事件、数据，发现关键性的问题。
5. 发布结果学习总结经验。

调查

调查是整个评估处理过程的起点，因为调查的结果将直接引导下面的事后分析的进行。调查能够决定最后分析的范围，使我们获得项目成员所需要的信息。
调查所应该遵循的三个原则：

1. 不问超出需求范围的问题
2. 不问超前的问题
3. 保护匿名者

客观信息

客观信息用来对调查过程提出的意见进行补充。同样的，我们想通过简单的方式来收集这些信息以使项目之间的比较易于实现。Collier,DeMarco,Fearey 提出从三个方面收集这些衡量信息：花费的代价、完成工期表、产品质量。例如花费的代价可以包括：

* 花费项目成员的工作日

* 由函数执行的代码的总行数
* 由函数执行的代码的总的改动的行数
* 接口的数目：所有的、新增的、更改的、删除的

理想的情况下这些信息应该是从开发和维护当中能够得到的。当意识到对于关键问题的回答不需要花太大的力气来收集所需信息的时候，事后处理能够鼓励项目组在下一个项目中做得更多、更好。而且，被重复使用的度量可信度比那些一次性的更加可靠。因此，即使事后分析过程不能找到所有他想找到的信息，他也会对以后的项目加以改进。

交流汇报会议

在交流汇报会议上，允许项目组的成员汇报在项目中哪些工作完成得好，哪些工作完成得不好。这种交流汇报会议应该以自由的形式组织。而且，对于较大的项目，其中项目成员众多的情况，可以采用分成若干项目小组的方法。

项目历史回顾

项目历史回顾的参加成员的数目是受限制的。历史回顾的目的在于发现项目当中的关键性问题的根本成因。所以，参加历史回顾的人将仅仅包括那些知道为什么在项目工期表、产品质量和使用资源中存在缺口的人。具体的讲，这些人可以来自开发部、市场部、消费者、项目管理者和硬件工程师等等。

结果发布

结果发布是事后分析的最后一步。在该步骤中主要是将以上步骤的新发现向项目组的其他成员发布，不是通过在开一次会议的形发布，而是由参加历史回顾的成员向其他的开发者和管理者发布公开的信件。该信件包括四个部分：

1. 一个关于项目描述的解释项目类型和项目唯一性的介绍
2. 总结事后分析的所有新的有益的发现
3. 归纳出制约组织达到目标的最大的影响因素
4. 建议改进的措施

处理过程完备模型

1980 年受 IBM 的刺激，有些组织开始将软件的整个开发过程作为一个整体来研究。一些研究人员也开始寻找能使一个处理过程变得更有效的特征。从这些工作中产生了处理过程完备的概念。

能力完备模型

能力完备模型（CMM）由 SEI 开发用以帮助美国国防部估测特的合作者的能力。CMM 起源于处理过程完备模型，在他当中每个组织被分成等级从 1 到 5 的 5 个等级。这些等级是由他对提出的开发过程的 110 个问题的回答而确定的。图 12.9 显示了从低级到高级的图示。在表 12.10 中列出了第 2 等级评估所需的问题，如果其中的一个问题的回答是“no”则它将被降到第 1 个等级。

CMM 模型使用对问题的回答来评价一个开发项目的等级。模型描述出能够产生更优秀的软件产品应遵循的原则与实践经验并且用五个等级来组织它们。模型可以以两种方式使用：第一，由潜在的用户使用，来确定提供给他们产品的优点与不足；第二，由软件开发者使用，评估它的真实能力，找到一条用以提高的途径。

每个能力等级都与一个关键的处理过程区域相联系，在这些区域中，组织应该集中精力在他要提高的行为功能部分上。第一个等级是 initial 级别，该级别的软件开发过程状况

是混乱的。从输入到输出的过程是无定义的，无控制的。相似的项目在最后的产品结果上由于组织结构和控制的不同，将会产生完全不同的结果。

如表 12.11 所示，在这个等级上根本没有关键的处理过程，开发工作的成功将依靠与开发者的努力。而不是整个工作组的合作。在第一级别上的组织应该集中力量建立对处理过程的结构化和控制，以便于进行有意义的度量工作。

下一个等级是可重复等级，可重复等级用于确定处理过程的输入和输出，确定限定条件（如：完成工期安排）和用于得到最后产品所应用的资源。基本的项目管理过程将对项目花费、工作工期等进行跟踪。在成员组中还有一些所必需遵守的规则以便于早期成功的项目能够在应用到以后相似的项目中。在此等级中，主要的处理过程是一些主要的管理活动。

处理过程是可重复的可以理解成某个子程序是可重复的： 适当的输入将产生适当的输出，但是从输入到输出的过程是不可见的。如果要具体描述这样的处理过程的话，可以用图 12.10 来表示。简单的对于一个项目来说我们可以将软件的需求看作是输入，最后产生的软件系统就是输出，所有成员的工作消耗看作是资源而完成工期就可以看作是限制条件。

既然可见的部分是可以进行度量的，因此在可重复处理过程图中所有由箭头代表的部分都将是可以度量的。即：输入、输出、资源和控制四个部分。

对可重复的处理过程进行改进可得到一个确定的处理过程。在确定的处理过程中，管理和软件工程过程中的行为都将被文档化，标准化，一体化。处理的结果对于组织中的每个成员来说都是一种标准的过程。虽然一些项目可能存在一些区别，但是这些标准的处理过程都将适应它们的需求。同时由管理过程不断提高它们的适应性。在这个等级上，关键的过程集中在组织集中上。

第三等级区别于第二等级的关键在于这种确定的处理过程将“construct system”部分可视化了，而且增加了从输入到输出之间的中介处理过程，这些中介过程的增加使得进入中介的输入和从中介出来的输出都可以被验证。图 12.11 显示了这一结构。

在这部分我们可以度量产品的属性。还可以跟踪每种类型产品的错误发掘，用预计的值来比较每种产品的错误密度。特别指出的是，这种早期的产品度量将为后期的产品度量提供暗示。例如：对可以通过对早期的需求和设计阶段进行度量以预测后期编码的质量。

受管理的处理过程主要集中在产品质量上。随着引进对处理过程可软件产品质量的详细的度量，一个组织可以集中应用定量的信息来使问题可见，和对可能的解决方案的最后结果进行估测。因此关键的处理过程区域集中在定量的软件管理和对软件质量管理者两个方面。

如图 12.12 所示，我们可以用从早期的项目行为的反馈信息来为现在的和将来的项目行为设定优先级。因为我们可以进行比较对照，所以一个行为变化的结果会影响到其他的行为。在第 4 级别上，反馈信息将决定资源的配置，但是项目中组基本的行为活动是不可变的，在此级别，我们可以评估处理过程行为的结果。可以用收集的度量信息来是处理过程趋于稳定化。因此，产品的效率和质量将会逐步趋向期望的值。

级别 3 和级别 4 的一个重大的区别就是级别 4 的度量反映了所有处理过程和一些重大的行为活动相互作用的特征。管理上的疏漏依靠一个记录资源分布、工作的效率、软件的质量等的数据库来弥补。

能力完备的最理想的级别就是最优化级别。在最优化处理过程中，定量的反馈信息不断的与处理过程在结合产生连续不断的对处理过程进行改进。而且他还监测新的技术和工具是怎样产生影响的。关键的处理过程区域包括错误保护、技术改变管理和处理过程管理。

图 12.5 显示出级别 5 的原理。一系列的交错的框格代表处理过程的级数，用 T0,T1..... 代表第一个框格是在 T0 时刻运行的处理过程。在某个给定的点上从行为活动中得到的度量信息被用来通增、删处理过程中的行为或是动态的改变处理过程的结构的方式来提高目前的处理过程。最后的结果导致向图中其他处理过程的移动。因此，处理过程的改变最后可以影

响到组织和项目的改变。用此种方法得到的结果同样可以应用到未来不同的开发处理过程中。

例如，我们用标准的瀑布方法开发软件。当需求确定后开始设计的时候，经过度量的和口头上的反馈信息可能暗示着在需求上存在着较大程度上的不确定因素。基于这些信息，我们可能会决定要改变这种处理过程，以使这些不确定的因素在我们真的进行设计之前得到解决。按照这种方法，能够使我们的处理过程在开发过程中有最大的适应性。

CMM 还有一个没有列出的级别：每个处理过程区域都组成了一个关键实践集合。它的存在暗示出开发者已经应用了处理过程区域。这种关键实践用以证明处理过程区域是有效的，可重复的和可持续的。

关键实践集由以下组成：

* 执行的承担者：确定哪些行为确保了处理过程的建立和连续使用
* 执行能力： 确定什么样的先决条件能够是组织有能力应用处理过程
* 执行的行为： 什么样的角色和过程能够用于实现一个关键的处理过程区域
* 度量和分析： 度量处理过程可分析度量结果需要什么样的过程
* 检验应用： 什么能够保证行为与已建立的处理过程相一致

SPICE

CMM 提供了一种处理过程的评估方法，SPICE 是由 UK Minstry of Defince 提出的一种新的度量处理过程的标准。即（Software Process Improvement and Capablity determination）与 CMM 相似，SPICE 的目的在与改进软件的处理过程和能力决策。他的框架是建立在定义友好的实践和处理过程的体系结构之上的。

有两种不同类型的实践

1. 基本实践，它是那些明确的处理过程中的行为活动所必需的
2. 一般的实践，它用一般的方式建立一个处理过程

图 12.14 解释了 SPICE 是如何将这两者联系在一起的。图的左面显示的是软件开发过程的功能性的实践，这种功能性包含 5 种类型的行为：

1. 支持用户的行为类型：能够直接影响用户，支持产品的开发和将产品发送到用户的手中，并保证正确的操作与应用。
2. 工程类型：指那些用于说明、实现和维护系统和系统文档的行为
3. 项目类型：能够建立项目，合作管理资源活为用户提供服务
4. 支持类型：允许支持其他存储过程功能。
5. 组织类型：能够建立商业目标并且进行达到这些目标的评估

图的右边展示了管理的画面，一般的可用于所有的处理过程的实践被安排成 6 个等级。

0. 不可执行：执行失败或是没有可确认的工作产品
1. 非正式的执行：没有在计划之内的执行而且没有跟踪纪录
2. 计划之内和被跟踪的执行：根据特定的执行过程进行检验，工作产品与特定的标准和需求相吻合。
3. 合理确定的执行：被合理确定的处理过程将使用经过审核的，合适的以被记录的标准处理过程版本进行执行
4. 定量的和受控的执行：详细的执行度量，预测能力级别，进行客观的管理最后进行产品的质量评估。
5. 后续的改进：接收基于商业目的的发来的有关结果和效率的目标信息， 从已确定了的处理过程得到的定量的信息反馈，在加上新的思想。

一份评估报告就是一个简要的归档;每个处理过程区域都将被评估最后被归结到六个能力水

平的其中的一个当中。图 12.5 显示了如何去填写这样一张建要归档表。

ISO 9000

International Standards Organization 建立了他的一套标准 ISO 9000。这个标准制定了当系统存在质量上要达到的目标时所应该采取的行为。而且，ISO 9000 应用在当系统的购买者需要系统的支持者示范某一水平的专门技术的时候。它不仅可以用在不同组织中，还可以用在相同组织中。

在 ISO 9000 的标准中，标准 9001 是最适合我们看法和维护软件的方式的。他解释了购买方必须做些什么才能够使软件的提供方能够符合设计、开发、安装和维护等方面的需求。表 12.12 列出了 ISO 9001 的条款。

12.6 评估资源

许多的研究人员认为使用资源的质量比我们所得到的某些技术上的突破更加重要。例如，DeMaro 和 Lister 声称创造性、连续的工作时间和友好的合作在软件开发过程中是十分必要的。他们强调，一个紧密合作的团体才能生产出优秀的软件。

类似的，在 Boehm's Model 中加入了基于个人属性（如：开发经验）作为调节工作工期和个人努力程度的参数。这种最初的研究显示出，工作能力高、低的两个工作组将对项目的开发产生重大的影响。因此，研究人员们强调多重视些开发人员因素而少重视些技术因素。

同时，软件多是在商业的环境中开发的，开发人员被提供给一些必要的资源如：时间、金钱等等，然后被要求解决提出的商业上的或是社会上的问题。在本章中，我们考察两种结构来评估这些种类的资源：人员完备模型和投资回报模型。

人员完备模型

很明显，CMM 并没有考虑与人和它们的生产能力相关的因素。CMM 设计成为是一种以度量处理过程的能力为主而不是以衡量组织的人员为主。Curtis，Hefley,Miller 共同提出了人员能力完备模型，用以加深对组织成员对软件产品的影响。

同 CMM 一样，人员能力完备模型也分成五个等级。每个等级与一些关键的实践活动相联系，这些关键的实践活动能够反映出组织结构发生怎样的变化和改进。表 12.13 显示了这些级别与相对应的实践活动。

最低的级别代表了一个起点，它有许多待完善的地方。在 initial 阶段，管理的技巧来自过去的经验，和个人的合作技巧而不是来自正式的训练。一些与人相关的行为是不与长期的目标相联系的。

在像级别 1 那样的不成熟的组织中，管理者并不将整体的能力看作是一种关键的资源，开发者们只追求它自己的目标。因为系统的雇员无计划的从一个工作转移到另一个工作上，这样就严重影响了知识和技术的进步。

级别 2 是向提高工作组能力走出的第一步。管理者们开始将人员整体的增长和开发看成是一种重要的职责行为。但是，这是只有当他们发现在单个成员的技术首先的时候，才会想到这种组织的整体职能。因此，可重复级别的组要任务就是在给定组织的不同雇员中建立基本的工作实践。

在级别 3 中，组织开始将它的工作实践适用于工作上。他从创建一个长远的用与开发它所需要的能力的计划开始进入第 3 等级。这种需求来自实际工作中需要的知识和技能。被看作是组织的核心能力。翻过来，当组织全体成员掌握了这种核心能力后，他们将会得到最后

的收获。

指导将在第 4 个级别上其中要的作用，不仅单个成员要学习核心技巧；而且还围绕着知识和技巧建立一些互相补充的小组。建组活动导致了组的兴起和组织内部的凝聚。许多组织的任务集中在了动员可发展成员小组上。

在这个级别，组织还为核心能力的增长确定了一定的目标。而且通过检验一些趋势来确定实践活动为关键性的技术带来进步的程度。因为基于这种定量的理解，组织全体成员的能力变的可预测，从而使得管理工作易于进行。

最由级别是第 5 个级别，也是级别最高的一个。在这个级别上，个人、管理者、整个组织都将注意力集中到提高小组和个人的技能上来。组织能够找到提高成员素质的机会，而不用等待某一个问题的反映。那些能够产生优化结果的实践活动被应用到整个组织当中。一般来讲，在最优的级别环境下，每个组织成员都集中注意于能够达到提高效果的所有方面。

投资回报模型

在一个正在进行的提高软件开发的尝试中，我们从推荐的方法和工具中进行选择。通常，受限的资源会限制我们的选择结果。因此我们只能选择那些标准的最可能帮助我们达到我们目标的选项。因此我们有必要注重开发者和管理者所做出的技术选择。

那些出名的商家在产品发布后通常要对技术投资问题进行分析。例如，最近的一篇文章曾提到通过从几个方面对技术方面的投资进行评估最后建立一个计平衡计分卡的办法。这些方面包括：从消费者的观点；从操作的观点；和最终结果的观点等。

然而，还有许多不同的方法可以获得环境的值。我们必须确定哪种投资分析是最适合与软件相关的投资决策的。投资分析应该与物质资源和人员资源的分配方式相联系。

并不是所有的投资分析都会如是的反映现实的情况。采取金融分析家的观点 Favaro 和 Pfleeger 将注意力集中到了最普通的方法上来： Net Present Value、payback、avage return on book value、internal rate of return、profitability index.。其中他们指出 NPV 对于评估与软件相关的投资有重要的意义。

NVP 根据整个项目的生存周期给出了一些经济上的数值。因为投资计划包括未来项目中的花销。因此我们将投资的 present value 看作是现在的花销。NVP 应用 discount rate 和 opportunity cost 并参照从资本市场像等价的投资中得到的期望值进行计算。这个值会随时时间而变。Discount rate 表示同样的资金如果投放到银行或是纯粹的金融领域辉等到多少回报。

Net present value 现在的利润值减去最初的投资值所得到的结果。接收 NPV 的条件是简单的：投资一个 NPV 值大于 0 的项目。为了看 NPV 是如何起作用的，请看下面的具体情况。一个公司可以用两种方式创建一个新的产品生产先：

1. 基于 COTS 这种选择包括较大的初期生产花费，但是可以得到后期的大的回报。但是 COTS 产品将可能会过时，因此必须在三年后更换。

2. 用复用的方式生产产品。复用需要大量的先期的设计和写文档方面的投资。

从表 12.14 可以比较二者的 NPV 值大小。

NPV 对资金流动的适时性较为敏感。返回的约晚损失的将会越大。因此，投入市场的时间将会对最后结果的分析产生重大的影响。项目的规模也会影响 NPV 的值。因为 NPV 是附加的，因此对于评估许多项目的情况，我们可以将它们的 NPV 相累加。而且不同的技术间得到的结果可以相互覆盖，因此评估每种投资时，我们最好还采取分别评估的办法。现时的世界中，NPV 并不用于单个的项目评估，一般应用在比较复杂的金融和决策制定的项目中。

12.7 信息系统实例

Piccadilly 系统确实增加了畏途它的那些电话广播者的通信量。做广告的时间变得更快了；为适应竞争的需求可以提供适当的功能等等。但是，我们怎样才能有效的组织管理这些信息？假如收入增加的话，我们可以把它和在系统开发式投入的预算作以比较，我们会惊奇的发现我们正处于一个我们的收入正停留在某一水平，但是如果没有该系统这个收入就会下降的情况下。因此，为了在市场上保持我们已有的地位，我们必须进行一些必要的技术投资。

这些问题应该在以后研究，除了本章中谈论的技术问题。换句话说，事后的分析不仅要回顾一下商业本身，还应回顾一下应用的技术。在回答问题“这个系统有使用价值么？”时应该考虑上述两种因素。有些时候，新的技术人员被接收并不是因为他们真的适用于该工作，而是因为好的雇员如果没有受到最近的技术培训的时候就将离开。管理者一定要明白开发人员和维护人员不止应该被工资所驱动。他们应该有对新技术的一种向往。所以，投资的回报不仅仅包括从前的角度的回报，还应该包括用户的满意。

12.8 现时实例

Ariane-5 是一个典型的事后分析的例子。调查组沿用由 Collier 所推荐的相似的过程。DeMaro 和 Fearey 集中精力于那些易见的能够决定产生错误的原因的需求上。这个报告避免了批评与抱怨，相反它列出了在开发过程应该采取的注重初始的问题（需求纵览、设计决策、测试技术、模拟执行等等）的几个步骤。

在下一步没有计入报告的步骤中用报告推荐的方式改变下一个进步的设计、建立和测试的途径。我们在 13 章将会看到，用 Ariane-5 中的数据与接下来的几步相比较以确定是否我们真的有所提高。因此从一系列的事后分析中，我们倾向于建立一个历史记录。当我们解决了一个问题的时候，直到大多数的主要的关键条件得到满足的时候，我们才会面对下一个主要的问题。

12.9 本章对我们的启示

在本章中，我们研究了进行产品评估、处理过程评估和资源评估的途径。本章从介绍几种评估的方法包括，特征分析、调研、事例研究和形式化实验。我们看到度量在区别评估和预测之间的差别时是十分必要的。然后，继而又研究了如何有效化度量的方法，我们应该保证我们正在度量我们想度量的对象，并且我们的预测是精确的。

产品评估通常基于一个与属性相关的模型上。其中，我们研究了三中的质量模型。接着又介绍了软件的复用技术。再复用之前，我们需要先评估一下被复用的组件。

处理过程评估可以以多种方式进行，事后分析回顾完成了的过程丽查找我们产生错误的根本原因。处理过程模型像能力完备模型、SPICE 等等对于关于问一些有关我们对过程的控制和返回的信息是有用的。

CMM 领导了其他的完备模型。我们在项目中进行多种的投资，投资回报决策帮助我们理解是否对人员、工具和技术上的投资在商业商会得到有利的回报。

12.10 本章对开发组的意义

在本章中谈到的许多评估模型都与组织的合作有关。处理过程完备模型要有组织的和作与交流。这些模型帮助开发组控制它们将做些什么，并且对将来有发生的情况能够做先期的预测。

特征分析，事例研究，调研，和形式化实验鼓励个小组之间信息共享以便于确定产品、处理过程和资源之间的关系。工作小组必须相互合作的工作。摒弃哪种由个人决定能够影响未来的重大问题的产生原因的做法。

12.11 本章对研究者的意义

在软件工程实践和产品中需要大量的经验评估。研究者必须将标准的调查研究技术应用到软件工程实践当中。我们可以做一个项目两次，一次用一些技术和工具，而另一次则不用。我们的研究方法的最终目的必须是能得到最优的效果。

一些模型和框架结构能够帮助我们理解我们正在研究的事物的关系，研究者还在不断的提出能够观察产品、处理过程、资源不同方面的新方法。通过对模型和框架结构的研究评估来看它们是如何与我们已知的那些结果相吻合的。

第 13 章 改进预测、过程和资源

我们已经研究了多种软件工程的技术和工具，他们的共同点就是帮助我们用最便捷的方法产生最优秀的产品。在 12 章中，我们学习了评估产品、处理过程和资源的方法决定开发和维护中的哪些方面将影响到软件的质量。在本章中，怎样将细致的评估和技术采纳相结合来帮助我们改进对新技术的使用。例如，只采用那些观察来的结果是不够的，我们还应该实际的检验对这些结果的实际使用以确定什么因素使它们更好，我们怎样才能使他们更加完善。

我们集中研究软件工程的四个方面：预测、软件产品、处理过程、资源。对于每个方面我们都基于实际的研究提出几种改进策略。像通常所说的那样，我们建议你使用那些最适合你的开发和为环境的技术和工具。

13.1 改进预测

在本书中我们已经看到对诸如：人员的努力、完成工期表、软件中的错误数、新系统的可靠度、测试产品的时间限制等等因素的预测。对于每种预测，我们都希望他尽量的精确。在本节当中，我们将要看到集中能够提高预测过程精确程度的方法。

预测的精确性

在第 9 章中，我们介绍了几种预测软件系统可能的可靠度的预测模型。在 12 章中，我们还研究了对有效化预测系统的需求，从而知道必须将预测的精确度与实际的可靠度值相比较。在本小结中，我们将集中研究可靠度精确性模型，并找到一些能够改进精度的技术。

Abdel-Ghaly，Chan，Littlewood 在相同的数据集上将集中可靠性的模型作以对比，发现在不同的预测中存在巨大的差异。如图 13.1 所示。途中的每条曲线代表一个模型。每个模型将产生 100 个连续的可靠度估计值，虽然每种模型都展现出一种再次数据集上可靠度不断增加的势头。但在每个预测的行为上还存在着相当的区别。有些优于另一些，而还有些则表现出存在“噪声”，随着测试的进行，预测结果会产生预测之外的抖动。

如果预测存在下面的两种情况，则视为是不精确的。

1. 如果预测的值总与实际产品的可信度存在着差异，则是这种预测是由偏差的。

2. 如果纪云某种度量的一系列的连续的预测值与实际值之间存在较大摆动，责成这种预测是存在“噪声”的。

处理偏差：使用 u-plot 方法

我们可以使用实际观测到的错误数目小于预测的数目的出现频度的方法来处理偏差。即：当一个给定的模型预测下一个错误将要出现的时候，我们需要测量一下下一个错误实际出现的时间，然后将二者作比较。例如，假设开始时间定为 t0，第一个错误出现的时间是 t1，下一个错误出现的时间是 t2，如此下去，直到在 tn 我们发现了软件中的 n 错误。我们将这些时间与预测中错误出现的时间 T1....Tn 相比较。如果 ti 小于 Ti 的数目严格小于 n/2。那么，我们就可以说我们的预测中存在着偏差。

从直觉上看，我们希望所有测试的进行，会找到更加精确的结果随着我们发现的错误数的增多，我们对数据的了解也将增多。

我们可以通过建立一系列的数值{ui}的方法形式化的来表现偏差。其中,ui 代表 ti 小于 Ti 的一种估计。例如，现在看一下第 9 章提到的预测系统，我们要通过先前发生的两个错误的平均间隔时间来预测的下一次错误出现的间隔时间。我们可以用这种技术来在 Musa 数据上生成表 13.1 所显示数值。

从这个数据集上，我们可以用计算分布函数的方法来得到 u 的值。然后画一个称之为 u-plot 的图：将 ui 值设成横坐标，按照 ui 的变化做出每步长为 1/（n+1）的阶梯图形。图 13.2 参照该图，如果我们的预测是较为精确的话，那么我们所画的阶梯线就应该去进与对角线。因此可以看出，预测与实际的值之间的差距就可以用接替下与对角线的接近程度来表示。

处理噪声：用 Prequential Likelihood 方法

仅估计模型中的偏差是不够的。假如我们拥有的是没有偏差但是存在噪声的模型，那么这个模型同样没有实用价值。

有时候，与噪声有关的预测结果的抖动会反映出实际可靠度抖动的方式。例如，在我们改动系统和引进新的错误时我们会得到这种系统的抖动。一种当现实可靠度中不存在抖动，而仅在估计值中出现抖动的现象叫做不合理的噪声。我们可以应用一个一般的方法 Prequential Likelihood 来同时处理噪声和偏差，帮助我们选择一个好的模型。

Prequential Likelihood 函数允许我们在同一数据源上比较集中不同的预测，因而我们可以从中选择出最优的的一种。应用 Musa 数据集和计算由先前得到值的平均值，我们可以为每个观察的结果计算一系列的 Prequential Likelihood 函数，得到表 13.2 中的值。

我们用这些值来比较来自两个模型 A\B 的预测结果。A、B 的 Prequential Likelihood 函数分别表示成 Pla 和 PLb 如果 Pla/PLb 的结果随着 n 增加而增加，那么我们就说 A 的预

测结果要比 B 的预测结果更精确。

Recalibrating Predictions

我们现在由许多种方法对模型进行评估，来帮助获取最适合我们自己的环将条件的模型。

* 检验每个模型的基本功能
* 寻找偏差
* 寻找噪声
* 计算 Prequential Likelihood 率

然而，这些技术都无法值出最好的模型。而且，在不同的数据集上会有不同的表现。甚至在一个相同的数据集上也会得到不同的结果。观察下面的 switching system data 。我们将用几种可靠度模型作用于它，同时规划一些预测结果。如图 13.4 所示。

图 13.5

图 13.6

图 13.7

13.2 改进产品

我们已经研究了许多软件开发和维护的例子。软件工程的目的之一就是应用适当的技术来改进提高这些产品是它们易于使用，自由的实现错误发现，能够更加有效的完成预定的工作。

在本节中，我们将研究两种能够改进产品的策略检查和复用，来显示通过这些技术策略的引进是如何在产品中产生了可度量出的改进提高。通常，我们是通过对错误数的观察来检验这些技术的作用结果的。

检验

Barnard, Price 指出 AT&T 就致力于应用代码检验的方法来提高软件的质量。然而，对过去使用经验的文献的研究表明用检验方法查处的错误比率可达 30%到 70%。在为提高查出错误百分率的努力尝试中，Barnard, Price 使用从商业需求、基于目标的规划、监控、控制和改进检测产生的 9 种度量方式的集合。最后的结果使得 AT&T 明白，检测的结果不仅提高了代码的质量，而且还展示了在准备和检验代码的过程中，全体成员组的重要作用。表 13.5 展示了对两个实例项目的一些度量主题和得到的对应值。

对于第一个实例项目来讲，观察者们发现由 41%的检测速度快于 Fagan 所推荐的每小时 150 行代码的速度。在第二个项目中，检测速度低于 125 的检测到每千行代码中的平均 46 的错误，高于那些快速度的检测。这种发现表明，或是由于低速而导致了更多错误的发现，或是由于发现了更多的错误而使得检测变的低速。这种发现指导了 AT&T 调整它的检测过程，从而使得它能够发现更多的错误，更好的改进它的产品。

Weller 说明了一个在 Bull HN 信息系统中应用的相似的经验。Bull 的软件工程师们跟踪在开发过程中发现的错误并且与他们所期望的估计的错误相比较，如图 13.8 所示。

但发现的错误密度低于期望的值时，开发组会认定由于以下的几个原因：

1. 检测并没有检测它应该检测的所有错误
2. 设计没有满足要求
3. 项目比计划的要小

4. 软件产品的质量比预计的要好

相似的如果错误密度高于期望值时

1. 产品比预计的要大
2. 检验正在出色的完成检错任务
3. 产品质量低下

例如，假设错误密度的期望值是每页应该有 0.5~1.0 个错误，如果检测的实际结果低于每页 0.5 这个值的话，那么项目小组就应该进行调查以确定是否检测者花费了足够的时间准备这次检测。

Weller 指出项目中引入的错误数目存在着 7:1 的区别。即：一些项目中在发布产品时所带的没有处理的错误数会使其他项目的 7 倍。但对于同一个开发组进行的相似的项目来说，仅仅错误的比率大致相同。用比较期望错误和实际发现的错误的方法 Bull 组能够知道在开发中怎样事先的找到错误，以及怎样使他们的检验过程更加有效。最终会使产品质量慢慢的提高。

复用

复用技术长时间以来一直被人们认为是一种能够提高产品质量的一种好的方法。用使用一个已经经过了测试、发布和在其他地方正在使用的产品可以使我们不会两次犯同样的一个错误，我们可以直接利用其他开发者的成果。我们通过参考设计和组件编码阶段所发生的错误和改动的历史记录来确保在新的项目中它会良好的运行。

令人奇怪的是，关于复用对于软件质量的影响几乎没有经验上的信息。Lim 从他在 Hewlett-Packard 的工作中向我们显示出复用是如何提高代码的质量的。他进行了两个实例研究过程来确定是否复用真的能够降低错误的密度。图 13.9 显示出在新的代码的原来的代码中错误密度的不同。而且将新代码和如用代码中的错误密度结合起来看是十分重要的。

Moller, Paulish 研究了错误密度与在 Siemens 中的复用的关系。他们发现一个将原来的模块的 25%进行了改动的复用模块将产生比原来那些每经过改动的模块多 4 到 8 倍多的错误。因此，应用复用所得到的质量的提升还与我们对复用模块所作的改动有关。但是有些情况下我们是无法避免对复用的模块作相应的改动的，在发生这种情况是我们就应该使用一些如检验和附加测试等一些补救的措施来尽量减少因为改动而可能引入的错误。

13.3 改进处理过程

我们已经看到软件开发中的各个处理过程能够对最后产品的质量产生影响。处理过程完备模型的建立是基于改进处理过程本身将自动的改进最终的产品，尤其是软件方面。许多应用面较小的处理过程例如：prototyping, cleanroom 也将目标定在了能够减少花销，提高质量和缩短开发周期和维护周期上来。在本小结中我们将研究对这些处理过程的改进。

处理过程和能力完备性

本书的第 12 章介绍了提高所有开发过程中的处理过程的完备性的几种模型。例如：CMM, ISO 9000, SPICE。这些模型中有些已经得到了开一些开发人员的认可，但是还有一些开发人员不愿意自觉的使用他们。事实上，这些完备模型和他们中的评估方法在一些组织中正在成为固定的标准。例如一些美国空军软件开发合同中就明确指明使用最小化的 CMM。

但是自从引进了处理过程完备模型，同时也产生了应用它们的过程中的问题。例如，

Bollinger 和 McGowan 提出了使用最初的 SEI 处理完备性的问卷中所遇到的问题。而且他们还指出这些限制性的问题只能够一个好的软件的时间应用中的一小部分属性。他们还指出这种处理过程的完备模型为软件假设了一个制作范例。但是从第 1 章我们所学的知道，制作范例和复制过程在软件的开发当中是不同的。

他们也提出，这种向处理过程完备性趋近的行为并没有深入的挖掘软件的开发实践式样进行的。例如，在第 2 级别的确认中需要对问题“是否第一线的管理者进行了定制工作周期表和工作花销的工作？”得到“yes”的回答。这个问题代表在第 2 级别，项目必须由那些真正能够承担估测责任的人管理。但是，这个问题并没有说明关于是否管理浙江队它估测到的结果的精度作进一步的分析以及根据他们从处理过程和反馈而得到的信息对估测进行改进。而得到这些不精确的关于关键处理过程和实践的的估测结果的危险在于可能会绘出于项目和管理不相符合的图画。从这种意义上讲，那些不恰当的模型和不精确的评估应该被忽略掉；管理也不应该基于只有等级 2 的估测结果。

既然存在这些弊端，那么这些完备框架结构还能够使用么？将这种完备性排除会给软件开发带来利处么？例如，Pfleeger 和 McGowan 建议改进了的完备性可以提高处理过程内部处理功能的可视程度。SEI 自从能力成熟模型提出以后已经承担了对处理过程的改进对结果将产生如何影响的研究工作。在 SEI 的支持下，Herbsleb 已经收集了从 13 个代表不同级别的能力完备性的组织中说几了大量的信息。通过对随着因能够提高软件处理过程的行为的作用而带来的执行功能的改变的观察，研就人员最后确认了完备性模型最终会对早期的错误检验、推向市场的时间周期和产品的质量等方面产生积极的影响。结果如表 13.7 所示。

这项研究结果展现了软件处理过程的该上所带来的积极效应，但是我们还不能够由此将这个结论一般化。进行试验的组织都是那些自愿参加进来的，因此这就不满足实验选择的随机性。而且项目本身也没有表明会允许我们进行横向的比较。处理过程该进所产生的结果从一个项目到另一个项目也存在很大的区别。虽然我们可以看到处理过程的改进在这个实例中产生了有利的影响，但是我们无法保证这种结果在其他的实例中也同样成立。

这些报告的结果将在大范围内被怎样对待是不清楚的。在 Herbsleb 实验中得到的“反馈值”看起来是根据早期的错误预测、投放市场时间的缩短和减少操作中的错误的过程中度量而得到的。但是这些特征并不能满足用户的需求。因为用户仅仅会注重这些改进方面的某一些他们所感兴趣的部分。因此我们就能够仅根据这些信息就判断完备模型适合某个商业上的应用。

加入这些模型和度量是不正确的或是被误导了，那将产生资源的错误分配和丢掉商机的结果。

Card 指出不同的开发小组从 CMM 对相同的组织的评估产生的结果会有不同。因此，应用之前我们必须清楚，CMM 的可靠程度究竟为多少。这里，可靠程度是指重复的相同的度量过程能够得到相同的结果的近似程度。

关于处理过程框架的改进而引起的问题不仅出现在 CMM 模型中，Seddon 在他的关于使用 ISO 9000 模型的作用时提出了同样的问题。

因此，在考虑应用这些处理过程和组织的框架时，必须的考虑一些度量上的问题。我们必须清楚什么样的度量和模型是有效的、可靠的。知道那些属性应该被度量。测试完备值和支持完备的行为的关系。

维护

由 11 章我们知道，维护过程的费用一直在攀升而且经常会出现超出软件开发费由的情况。因此，对于研究出一种能够改进软件维护处理过程从而降低维护费用，提高软件质量的

方法是十分必要的。为了选择一个恰当的模型，Henry 考虑了这个问题，提出在做出适当的选择前英回答的三个问题：

1. 我们怎样才能够定量的评估维护处理过程
2. 我们怎样能够评估处维护过程的提高改进
3. 怎样才能评估任何处理过程的改进而得到的结果

使用借助一般的统计学测试的方法，定量的研究维护行为处理过程和产品特征之间的关系，他们研究了维护处理过程，而且，他们研究了需求的变化怎样影响到产品的属性。

例如，研究小组想找到一种简单的软件组件分类方法以使他们能够预测出那些组件有出现错误的倾向。他们基于被纠正了的错误数和对组件产生影响的升级改动数目的中值创建了一个附加表。最后发现在改正了的错误和升级的影响之间有一种相关性。他们用这种关系来标识组件，通过细致的选择组件；通过与影响组件的升级项目的关系，他们确定了 93%的错误数高于中值的组件。

软件工程的测试目前集中在衰亡测试上，而且现在软件工程测试小组在监测承包商所进行的测试。

通过研究人员对软件维护过程的研究指出当要改进维护过程时应该注意以下几个方面：

1. 认真应用统计的方法，因为单一的使用某种技术在某些情况下将无法争取的评估出处理过程的改进
2. 在某些情况下，如果最后的影响结果能够在统计结果中显示出来，处理过程的改进将会是十分生动易于理解的。
3. 处理过程的改进以不同的方式影响返回的结果。

Cleanroom

NASA's 软件工程实验室已经从事 20 年的评估和改进处理过程方面的研究

Basili, Green 描述了 SEL 如何引进一种新技术。首先评估一下它的影响能力，然后用那些新技术新工具来得到有关方面的重大改进，SEL 同时考虑了使用新技术的风险因素，只有当新技术是恰当的时候才会被用于通常的使用环境中。

这些非规定内的研究工作通常都以一种十分正式的可控制的实验或是实例研究的形式进行。SEL 首先通常用一个小的允许方便控制变量的实验，当这个实验的结果显示有意义时，接着就要进行实例研究已确定这种小范围内的结果是否适合于现实的环境。当 SEL 认定某一项技术将在 NASA's 开发和维护环境下正常工作时，他会将着整个过程积累的经验存进库中作为保存，便于其他人能够理解和使用这种技术。

Basili, Green 研究了 cleanroom 中的关键的处理过程看他们是否对 NASA 有益，他们将他们的研究组成五个部分

1. 一个比较读和测试的受控制的实验
2. 一个比较 cleanroom 和 cleanroom-plus-testing 的实验
3. 一个检查在 3 人开发组和 2 人测试组中的 cleanroom 的事例研究
4. 一个检查在 4 人开发组和 2 测试人组中的 cleanroom 的事例研究
5. 一个检查在 14 人开发组和 4 测试人组中的 cleanroom 的事例研究

实验

在第一个实验中，Basili 和 Green 用云找错误的办法来比较读来的结果。相互比较的结果如表 13.8 所示。

他们也考虑了结果的可信程度，在这个实验结束之后，读程序的人认为他们找到了所有

错误的一半，而测试的人认为他们几乎找到了所有的错误。但是这是不正确的，Basili 和 Green 认为经过了大量的验证练习，是测试者产生了某种信心上的错觉。

读程序的人同样能够发现各种类型的错误，包括接口错误等。

在第 2 个实验中，Basili 和 Green 理解了传统的 SEL 在测试上的依赖准则。他们认为将测试至开发者的控制之外将会产生很大的风险。因此，他们将传统的 cleanroom 与允许测试的 cleanroom 相比较得到如下结果：

* cleanroom 的开发这对于作脱机的度程序的工作是很在行的。
* 允许测试的 cleanroom 开发组更加重视那些功能上的测试而不是读
* cleanroom 小组花费很少的时间在线测试，他们更易于达到他们的目标
* cleanroom 的产品并不复杂，有许多全局的数据和许多的注释
* cleanroom 的产品能够完全的满足需求，而且他们也拥有许多独立的成功测试事例
* cleanroom 的开发者并不使用一些正式的方法
* 几乎所有 cleanroom 的参加者都会对它产生好感

因为两种组织间的主要不同点影响了一些额外的测试的进行，Basili 和 Green 建议允许测试的 cleanroom 成员不要在花费时间在学习其他的技术了。

事例研究

所有三个事例研究都包括飞行动力软件的开发。第一个研究的目的是在不增加成本的情况下提高软件的质量和可靠性。因为 SEL 已经建立了一个关于飞行动力软件的基准，因此研究者可以比较 cleanroom 的结果研究它们的不同点。这种 cleanroom 的处理过程基于前两种实验而建，包括：

* 开发预测试组的区分
* 依赖于同组织人的观点而不是某一单元的测试结果
* 用非正式的行数来定义系统设计
* 统计的测试基于操作上的设想

Basili 和 Green 发现当启用 cleanroom 时，有 6%的项目工作会从、编码转移到设计上来，对于传统的开发者来说，将有 85%的时间用来编写代码，15%的时间来读，但是 cleanroom 可以至少缩短一半用在这些上的时间。然而由于在应用这种正式的方法时会有一段困难时期，因此，他们将统计测试与功能测试结合起来。

对于 cleanroom 小组来说，检错的速度明显的得到了改善。但是同时也存在一些弊端，cleanroom 的成员大多数不愿意用设计抽象和盒式结构。从而使得与代码的编译过程和开发者与测试者的合作交流。

第三个事例研究主要是从前两个获得经验。因为大多数的 cleanroom 都支持人员的培训，因此这种经验尤其重要。

结论

SEL 的 cleanroom 实验教给我们以下几点：第一，Basili 和 Green 已经向我们显示出如何将实验于事例研究结合来比较一种新技术和以存在的技术。第二，他们的研究工作还在继续，随着新技术和工具的不断采纳，这种议程的组织将不断的完善。

13.4 改进资源

产生一个优秀的软件将会需要许多的资源，我们必须对系统要求的各种资源及时的满足。有些资源提供以后就没有在改进的余地了，但是有些资源是高度可变的。正确的理解他们的边环境会帮助我们对他进行适当的改进。例如，进行了同样培训的软件工程师会有不同的工作结果。我们都知道有些善于编码的软件开发者在测试上干得很糟。这就要求我们在软件工程过程中要优化对资源的利用。

工作环境

不幸的是，已有的文献中有很少介绍到了软件工程中人员角色对项目的影响。DeMaro 和 Lister 对工作环境对工作质量的影响问题作了研究，创建了“peopleware”这个词来区分开发者间存在的差异，以强调通过提供给开分院良好的工作环境一样能够提高产品的工作质量。其中对这些工作环境的要求中涉及工作环境中的噪声要求，工作空间应该尽量保持安静；还有对工作小组规模的要求，一般认为，一个规模小的小组回避大规模的小组更加有效，因为小的工作组成员之间便于交流沟通。

DeMarco 还特别强调工作小组的团结性，工作组的成员必须能够和睦的一起工作，互相户作，尊重对方。因此他建议以前工作良好的工作小组在新的项目中还要继续保持存在。因此，整个项目工作小组将视自己为一方而视问题为另一方。

代价、目标和折衷安排

时间可以看作是一种关键的资源。拥有足够的时间开发者可以通过详细的设计、开发和测试生产出高质量的软件。但不幸的是，由于来自市场竞争等方面的压力，一个项目被给与的时间通常都不会是很充足的。因此这就需要我们对开发的代价、目标和产品质量的关系等有一个深刻的理解和认识，以便我们能够正确的协调各方面的关系，达到最终的目的。

现在有许多代价和完成工期的估测模型都包含一种折衷安排的分析。例如，COCOMO 描述了代价与完成工期之间的相互关系，同时提出了一种名义上的基于项目参数的代价与完成工期的综合度量。还有其它的一些模型像 Putnam's SLIM 说明对于来自项目工期方面的压力的结果导致开发全体人员的自身需求的不断的增加。类似的，Lister 也指出，在重大压力下的开发人员不一定会使项目更快的完成，相比之下，规定一个项目的最终完成期限将更加有用。

Abdel-Hamid 应用系统动态模型来研究项目完成工期压力所带来的影响。他注意到，项目的工期是一个连续的过程，项目的管理者制定关于项目的工作工期，这样最后真正的完成时间和会费的总的费用将由这个最初的估计和现实的实现程度和所需资源的满足程度所确定。

Abdel-Hamid 将它的模型应用在了 NASA Goddard Space Center 的数据中。来观察管理者的策略对项目花费和完成工期的影响。如图 13.11 所示，显示了分别采取两种开发人员完成项目的工作日的策略最后得到的不同结果。第一种策略用圆圈表示得部分，表示管理者将为了保证工作的按时完成将适当的调整劳动力的分配；第二种策略用方框表使得部分，表示解决完成工期的压力的办法是通过适当的延长工期的办法解决的。

13.5 一般的改进方法

为了保持成功，组织应该灵活并能够不断的提高改进自己，像其他的对于一个公司重要的因素一样，一个基于技术的程序也同样要求策略上的长远计划。这种计划不仅要集中在技术被怎样应用上，而且还应该集中在怎样应用它来提高产品、改进过程和资源上来。

因为事物会随着时间而变化，因此一个策略的规划应该适时而定，管理者和开发者应该问一下关键性的问题：

* 目标是否相同？假如商业上的目标改变了，那么程序的目标也应该随之改变

* 目标的优先权是否相同？因为一种类型的改进就是能够正确的执行，所以满足这个条件的其它的可选择类型同样可以成为下一次的初始优先级。

* 问题是否相同？与程序的第一阶段相关的问题可以被一些后期的问题所取代。

* 度量是否相同？当一个开发和维护过程趋于完备的时候，系统中丰富的度量要求我们能够理解、控制他们。

* 完备性相同么？开发和维护过程的完备性是可以提高和改进的，对于它的应用需要对可视化的理解和度量新的对象。

* 处理过程是否相同？开发的处理过程随着时间会产生巨大的变化。应用反馈信息的循环可以对处理过程作相应的变动与改进。

* 使用对象是否相同？许多技术程序开始的时候很小，但是会慢慢的扩展到各程序间的相互合作之中，这是能够理解他们的使用对象就从程序员、开发者和管理者向各个分块的总管和合作的执行者转变。即面向的对象随着技术影响力的增大而扩大。

13.6 信息系统实例

在本书中，我们已经举了 Piccadilly 系统的实例。假设系统正在正常的运行，同时对系统的大多数的维护反映了广告公司为满足商业需求而进行的改变的需求。那么，Piccadilly 应该在维护中遵循哪些原则以使他们能够快速的作出变化而不在软件中引入错误。

一个执行维护功能的关键性的原则是维护人员能够检验软件的设计过程来确定是否软件是易于更改的。参照过去更改的历史，他们可以确定最有可能被变化影响的模块。

另一个原则是检验其它在 Piccadilly 系统中的其他的相似的软件系统。在本章和前些章中已经说明了 Bell Atlantic 和 Chase Manhattan 如何用一个大的更复杂的系统来替换几个遗留的系统。

13.7 实时实例

改进策略在欧洲航天局同样得到了重视。Lions et 的报告列出了以下几种改进策略：

* 工作小组应该执行一个详细的需求回顾，以确定 Arian-5 和 Arian-4 需求上的区别。而且，规格要求应该将 Ariane-5 的轨迹数据作为一种功能需求来看待

* 工作组应该在对预测飞行参数的测试中引入模拟的加速符号，用一个转盘来模拟发射时有角运动的参数。

* 导航系统的精确性应该有分析和计算机模拟来证明

* 回顾探讨应该作为设计和质量处理过程的重要的组成部分。回顾的过程包括各方面的专家和主要的项目参与者。

这些步骤将会帮组我们确定那些在过去监测当中被忽略的错误，在将来的监测中将不会被再次的忽略。

13.8 本章对你意味着什么

在本章当中我们学习了几种能够改进预测、产品、处理过程和资源的技术。我们知道应用 u-plot、prequential likelihood 的方法如何能够改进预测过程。应用 recalibration 如何能够降低噪声和偏差。产品的改进可以通过复用程序和引入一个检验过程的办法。处理过程能够被改进提高通过评估他们的作用结果和决定他们在提高产品质量和效率方面的关系的办法。类似的，处理过程的完备框架结构能够帮助组织进行易于改进产品质量的行为。最后，随着我们对于人员的不同素质的理解的加深和对项目花销代价与完成工期的相互关系的深入研究知道，在资源分配的改进上也将有广大的远景。

作为一名开发者或是维护人员。这些结果可以直接影响你。为了真正提高你的专业水平，你必须真正的加入从事实例研究、和实验测试的工作中来，并且将你得到的回馈信息给那些正在努力寻找什么能够达到提升改进的结果的人。你必须取得你的用户的信任，这样你对你所开发的系统才会有信心。而且你必须工作在一个小组当中，争取用最一般通用的方法来解决你的问题。

13.9 本章对你的开发组有什么启示

本章研究的结果将对你的开发组未来的变化起到深刻的影响。好的预测依赖于对于影响你小组的项目花销代价和完成工期的得关键性问题的一般公共的认识。好的产品和优秀的处理过程依赖于你的工作组的协同一致的工作，同时优化的资源配置将士你的工作组能够正常工作的重要保证。

一些在这里报告的结果将会直接影响你的工作组。假如某些处理过程的完备框架在改进软件产品的质量方面如果被真的证明是有效的话，那么他们中所推荐的某些实践活动就将被一些工作小组直接定为一种制度。

另外，本章所报告的研究结果也强调组织之间互相的检查其它的组织的工作的重要性。Inspections, cleanroom, reuse, 以及其他的一些与质量相关的处理过程都包括对一个组织对另一个组织和个人的工作的详细审查。

13.10 本章对研究人员的意义

对于改进项目方面的研究正在不断的扩展，本章中详细说明了对更多的调研、事例研究和实验的需求。Basili 和 Green 的例子显示以前研究的结果即是如何被组织在一起的。

在改进的框架结构中同样需要基于证明技术的正确性和对于特殊实际应用的实用性的研究过程。而且，在研究中这些框架结构集和最好被联系在一起，以使得实际进行应用的人能够选择出哪种技术最为实用。

最后，软件工程的研究人员还应注意人为的因素。我们能够从已经存在的社会科学中的理论理解像开发组的规模、组织形式、和工作环境等资源的作用。然后可以在进一步估计以

下哪些的因素可以应用到软件工程人员身上。许多的研究人员都同意人员能力的区别是工程能否达到质量目标和按时完成工作的关键因素。对于这种因素的理解将会帮助我们设计适当的技术和工具达到既定目标。

第 14 章 软件工程的未来

在本书中我们详细的介绍了一些能够开发出高质量的软件的方法。我们看到了如何应用恰当定义的处理过程来建立满足用户需求、提高质量和生存周期的软件。但是，要确定我们以后的方向还要回过头来看看在目前相对短的时间内，我们都取得了哪些成果。

14.1 我们都做了些什么？

自从 1968 年软件工程这个词首次被 NATO 使用以来，直到今天他被许许多多的公司机构认定为是它们开发和使用软件的依据，我们已经建立了许多的数据集、理论和实践，而由于他们的建立也正在几乎影响到了所有人的生活。由于软件在世界上的广泛的应用，我们随之扮演了学生、项目参加者、研究者等角色。我们的部分职责落在了如何明确我们应该做什么和为什么要这样做，用这些知识来最后提高改进我们的时间和产品。

我们现在正在不断的朝我们的目标努力，我们用复杂的语言来指导我们的纯属数字的系统。我们已经弄清了建立可复用的产品和新的设计方法的模式和抽象的过程。我们已经将正规的方法应用到了困难的、不直接提出的问题当中来帮助我们将他们的复杂程度和冲突情况可视化。我们还建立了一系列的工具来帮助我们的工作进行。

我们已经设计了诸如面向对象和白盒测试等软件工程的方法，但是，在我们的面前还有新的问题有待解决。我们总是试图从大的方面确定事物的精确性，我们可以预测宇宙飞船什么时候会到达火星；相反在小的方面我们无法给事物以精确性：我们无法预测一个软件产品下一次崩溃时间等。

Wasserman's 趋向完备的步骤

Wasserman 通过 8 个步骤最后达到一个较为完备软件工程的阶段。

* 抽象

我们已经看到了抽象过程在问题提要提取方面的重要作用，除了设计和编码我们还得继续将抽象过程应用在需求、用户使用倾向、测试事例等方面。抽象过程学习和解决问题过程的重要根基。

* 分析和设计方法

我们使用一系列不同的方法和标识符号来表示我们所遇到的问题与对问题的解决。由于我们理解和学习这种方式的程度不同，因此每个人在具体的应用中都会有自己的倾向。一些人倾向于用图示的方法，然而另外一些人倾向于使用文本的方式。多媒体软件的出现导致了人们应用声音、图像和动画和其他的多样的形式。而解决问题中我们的目标是将这些被代表

了的问题转化成一般的形式，一种为讨论和归档所有用的形式。因此在各种可选择的表达方式中，我们可以找到一种表达我们的需求和设计的一般的方法。

* 用户接口规范

随着软件被引入到我们生活的关键领域，用户的作用显得越来越重要。我们必须用户是怎样提出他们的问题和予以解决的。因此软件应该支持有正当的用户行为的功能。通过对用户需求和现实的商业需求的研究理解，我们将开发出更加实用、可靠的软件。

* 软件体系结构

Shaw 和 Garlan 已经告诉我们不同的体系结构是如何对相同的问题产生不同的解决方案的。对于每种体系的解决方案，都会有支持者和反对者。我们必须用解决问题的最优化的特征来指引我们对某种体系结构的选择。我们一定要扩展对体系结构方面的研究以使我们对软件模式、组件和软件类型的含义有更深刻的理解。

* 软件处理过程

当软件的处理过程变的更加可视化和可控制的时候，它将一定会对软件的质量产生重大的影响。但是，这种可视化和控制具体是怎样影响软件的质量的将是以后研究的问题。在第 1 章中我们看到，软件开发像其他科学一样也是一门艺术。是一个创造和组织的过程。我们必须学习应用软件处理过程来简化软件的开发，以及哪种处理过程在那种条件下将会最有效的工作和在处理过程的选择上有哪些产品和人为的因素是最有参考价值的。

* 软件复用

过去，复用技术集中在了复用代码和创建新的能够复用在多个产品中重复使用的新的代码模块。未来我们必须要开阔我们的眼界，从开发到维护过程研究复用。而且在研究复用需求时还应该将我们对抽象和软件体系的理解相结合。同时，我们需要一种能够帮助我们理解一种组件的评估技术，在应用某个复用模块时，我们应该对它能够完成我们预定的任务有一定的信心。这些复用的技术不仅应用在家庭式的软件开发，也能应用在大范围的商业应用中。

* 度量

在本书的几乎所有对行为的描述种都包含度量。我们需要了解我们的产品是否达到了质量要求以及我们希望我们进行的实践是有意义和有效的。在未来，我们一定会以一种平常的、有用的、及时的方式来度量产品、处理过程和资源。我们在度量产品的时候应该使用除了代码规模的那些特征，在软件编码之前我们就应该通过度量过程了解我们的开发的产品将会正确的工作。

* 使用工具和集成环境

许多年以来，开发者们一直在寻找一种能够使软件工程过程更加多产和有效的工具和环境。现在，开发者们正在开发和使用一些具有更现实的期望的工具。我们的工具和环境将会自动的完成现实的工作，完成预定计算和进行跟踪等等。未来，我们必须找到一种能够自动追踪产品之间的联系的工具以使我们在进行某一项改动之前可以先对后果进行以下分析。我们还需要开发一种度量工具能够对后台的环境进行度量，为开发者提供所需的关于产品和过程的信息。我们需要能够模拟和帮助理解问题的工具。需要能够支持我们进行复用的工具，这样就可以从初期的开发者中提炼出对我们有用的需求，然后再现在的产品中将他们组织在一起。

现在的情形

Wasserman 的原则至少是我们现在应该注重两方面的重要问题。一个是我们如何将软件工程的思想转移到现实应用当中；另一个我们研究和实践的结果将怎样应用到对处理过程、资源和产品的决策中去。

14.2 技术转移（transfer）

假设你将要进行一次软件开发或是软件维护的工作，那么将选用一种已经成熟的并且经过了时间验证的方法还是使用一种看起来更加有前景但是现在还并不成熟的方法呢？你将为新技术找到合理的适用证明还是仅仅相信你的直觉、你的同事的建议或是卖主的说明？你的最后选择可能要依靠于你是一个技术的创建者还是一名技术的使用者，还依靠于你注重解决问题的过程还是注重最后解决问题的结果。例如，IBM 就充当了技术消费者的角色，因此它一般集中于问题是否能够被正确稳定的解决。但是 Silicon Graphics 公司则注重新技术的采纳使用。技术转移指的就是技术的创建者创建使用新的技术，同时技术的消费方能够在生产和服务中使用这些新技术。

考察以中西技术是否成功的关键在于看他是否能够正确的解决问题，看他是否有商业上的价值以及从它的开发到他在市场上成为有用的实际应用需要多长的时间等因素。

如今怎样实现技术转移

如今的经验告诉我们，现在我们必须以比以前更快的速度将新技术推向市场。市场不可能为某项技术的革新而等上 10 年或是 20 年的时间，因此许多的机构、公司在一项新技术还没有找到明朗前景的根据时就对它予以采用。

从另一个角度讲，技术采纳就是当一种新技术看起来已经具有十分明朗的利益前景，而不需要对其进行严格的适用性评估时而进行的采纳。

在技术决策制定中使用证据

为了理解技术决策是如何制定出来的，让我们回到商业学校，来到他们的市场研究的课堂。Rogers，一个在许多组织中研究技术转移过程的专家，基于“扩散研究”（diffusion research）的方法提出，技术转移过程有着其明确的形式与速度。如图 14.1 所示，最先接收一项新技术的人叫做“革新者”；他们占整个可嫩人群的 2.5% ，他们大胆的而略带轻率的接受新技术，而他们所基于的出发点通常是从一个系统范围外的观念。革新者通常是按照它个人对新技术的接触而使它的同事们去接受这项新的技术。

早期的接受者

早期的接受者也属于能够较早的采纳新技术的成员，他们对新技术的采纳使基于新技术可能会带来的利益潜力而不是基于新技术本身。这种早期的技术采纳者还让其他人先尝试这种技术，当他们发现新技术在实用中所获得的成功时，他们就会基于这种成功将新技术引入他们所在的组织。通过明智的技术决策的制定早期的采纳者们会减少新技术效率和影响的风险。

早期采纳主体

早期的采纳主体一般都经过了详细的决策制订过程。他们基于现实确定最初的技术革新者所做出的决策是正确的。他们愿意接受一项被其他人证明可行的技术。而且这种采纳时不需要技术被证明在其他的地方也是正确的。

后期的采纳主体

后期的采纳主体是多疑的。他们对新技术的采纳通常是在商业上的压力和同行的竞争压力下而进行的。他们需要在这种新观念的大多数的不确定因素得到确认的情况下，并且新技

术被正式的建立，同时有足够的支持新技术有效的条件时才接受新的技术。

落后的采纳者

最后，那些落后的采纳者都是些拒绝接受新事物的人，由于经济或是个人偏见的原因。当他们确认了一种新技术的使用不会带来任何失败和错误时或是基于管理者和用户方面的压力才会最后采用这种技术。

因此说，成功的技术转移过程不仅仅需要新的观念而且还需要能够理解和采纳者中新观念的群体。

支持技术决策的证据

研究者通常进行调查研究来落实最后支持技术决策的证据。一项由 Zelkowitz, Wallance 和 Binkley 进行的研究表明了在试图采纳的参加者的接收方法上存在的巨大差距。通过对 90 名研究人员和涉及项目人员关于不同的实验方法对于验证一项新技术的有效性的价值的调查，Zelkowitz 发现实际的项目工程人员更加注重方法与他们的应用环境的相关性。即，像实例研究、域研究、复制控制实验等方法对于决策的制定过程是十分重要的。

另一方面， 研究人员一般都乐于那些包括可重复性的验证有效性方法的研究因为她们可以独立的实验室中使用。例如，理论的证明、统计分析和模拟等等。他们忽视了需要与项目实践人员交互的方法。换句话说，研究人员常常避免实践人员所推崇的那些方法。这样，研究人员试图建立一种能够评估新技术的证据体系，而是检人员也同时建立了另外一套完全不同的体系。这项研究表明，成功的技术转移需要对各种目标人群的透彻理解，并且需要收集一系列的相关的可信的证据来证明它的正确性。

证据的详细分析

卖技术性的观念需要一个基于对人群理解的有效的市场决策。如我们见到的，这种人群可以包括面向新技术的五种属性，研究人员应该知道基于每种属性所寻找的证据的类型。但是我们怎样将各条证据合成一个完整的论点呢。幸运的是，现在我们可以参考一下合成证据的规程。Schun 详细的描述了从两点：证据源和可信性对证据的分析，我们可以将一条证据分成如下五类：

* 切实的证据： 可直接用来减压它所显示的东西。例如：对象、示例、文档、模型、图表。

推荐式证据： 这种类型的证据由能够知道如何获得这种证据的人来提供 Schum 指出了三种类性的这种证据：直接观察的、经由其他资源得到的、弹出的一种观点。

* 可疑的推荐证据： 证据的提供者不能清楚的记得这个证据或是用可模糊词来提供支持证据的信息。如：可靠度的估计。

* 丢失的证据： 所期望的证据无法找到。例，用已经无效的系统维护人员提供的报告

* 已接受的事实： 那些不需要证明的被接受的信息。例，某些数学公式和一些物理的常量。

证据的可信度是通过对它的真实性和准确性的观察而得到的。例如，一些估计证据的可信度直接与他们的提供者的可信度相关联。反过来，提供者的可信度将依靠我们对它的信任程度、客观程度和观测的灵敏度。

当我们观察每条证据的时候，我们还可以将几条证据合起来一起看。是否他们能够结合的整体能够为技术提供一条证明推论。我们先得到某种信任程度的一条推论，然后经这条推论和它的信任等级作为一条新的证据发布出去。假如要合并的证据之间存在着冲突，那么这

种冲突则需要在信任等级上反映出来。

因此，证据应该是对研究人员和实践人员有意义而且还应该是可信的。但是为了达到这些目，这些证据都应该回答些什么问题呢？Rogers, Schum, Moore 给出了什么样的问题适合哪些人的几条规则。例如，Rogers 指出一项技术被采纳的速度应该基于以下几种重要因素：

* 用于增加理解新技术的交流渠道的特性

一些类型的技术采纳者发现某些交流渠道将会比其它的更加有效。例如，革新者多依赖于技术性的期刊和来自研究人员和专家的报告;而早期的采纳者则是通过向革新者征询技术的可信度信息。

* 潜在用户所在社会的属性

革新者一般会求助于专门的技术人员，但是新技术所面对的主流群体还得参考社会上的商业交流，看看竞争者们的反映。

* 向一个组织传播新技术的努力程度

早期的市场仅需要较少的技术传播方面的努力，但是随着市场主流的参与，将不仅仅关于纯粹的技术方面的信息，而且还需要诸如：产品包装、用户支持和当前采纳者和用户的推荐。

* 技术的属性

Rogers 指出技术的属性可以包括：

* 相对比的进步性： 新技术优于以存在技术的程度的特性。
* 兼容性： 新技术予以存在技术的相互包容程度的特性。
* 复杂性： 新技术的易学易用程度。
* 可试验能力： 新技术是否能够带着有限的偏差被验证
* 可观察性： 应用新技术的结果对于其他人是否可见

图 14.2 显示了应用于不同群体的证据的类型。

左下方的方格反映了革新者所感兴趣的内容，他主要注意技术因素。早期的采纳者注意了一些稍宽的产品方面的内容。等到了市场的主流的采纳者就主要集中注意公司和市场的因素了。这个图显示了不同的用户群体将对不同的证据感兴趣。

技术转移的新模式

采纳新技术的最重要的目标就是至少以某种方式使得一种新产品、过程和资源得到改进提高。证据能够帮助我们判断是否这种新技术能够实现这种提高改进。我们向通过调查知道新技术的应用与最后产品的一种或几种的利益变化之间的关系。甚至如果这种得到的利益与应用原来存在的技术或是其它的相类似的技术是相同的，那么我们照样可以选择一种新的技术如果它仅仅能够减少原因到结果的不确定性。换句话说，我们希望我们开发和维护过程的结果是可预测的。

为了这一目的，我们可以将为技术转移的成功而建立的模块（blocks）放到一起，来更好的理解技术转移过程是如何在我们的组织中工作的。通常，这些模块包括对一项新技术的初步的评估、它将由那些第一次使用这种技术以检验它是否完成了预计完成的任务的人来执行。该步骤应该根据不同组织的特征而制定。同时我们还得记录下对于此技术的支持者和反对者。以总结谁对这种技术的采纳是促进的，谁是起阻碍作用的。例如，对于支持采纳新技术的人对于决定某一特定的技术的接收时将被赋予大的决策力。

不断增加的证据将被用来支持更加高级的评估，我们不再仅仅检验技术本身，而且我们还可以来检验证据本身。即，评估一下证据是否能够结合成使用技术的一个论点。我们比较新技术用老的方式应用的条件和用新的方式应用的条件，来决定是否这些证据之间存在冲

突、一致性等等。

然而，光用这种合成的证据来确定技术采纳的正确性也是不够的，现在的市场条件还要求我们对技术进行包装以及增加一些附加的支持技术，以使它面对用户变的更加“友好”，易于理解易于使用。

事实上，我们需要能够发布新技术的多样的模式，这些模式应该增加本节介绍的市场和证据的因素。为我们提供理解影响技术采纳的发布的成功的关键因素的依据。

改进技术转移的下一步

我们不会为一种无意义的，不会成功的技术进行投资。实践者可以一步步的帮助研究者理解成功的技术转移的本质，然后用这种新的理解来建立支持有效的评估和采纳的模型。特别的，优先现存的对软件工程技术的经验的研究并没有经过周密的规划。假如软件工程的研究人员能够组织一种研究使得每一步都能够清晰的为最后结果服务的化，那么最后得到的证据结果集将会是有说服力的。

同时，实践者倾向应用新的技术。技术转移的支持者和反对者集中讨论三个范畴：技术、组织、可信度。例如，技术转移会受到缺乏包装、缺乏与一个紧迫的技术或是商业问题的关联、不易于理解和掌握等因素的限制。相反，技术转移同样受到一个易于理解的上下文说明、和一个对未来存在的利益潜力的明确理解等因素的支持。

我们无法知道我们所需要了解的关于软件工程技术在那个条件下会得到最好的执行的全部信息。我们也不能去等上几年已获得这些信息。但是有一些来自软件工程研究和其他的技术评估和转移中所遵循的规则的经验我们是可以借鉴的。现在，我们就可以一步步的来加速我们进行技术采纳的步伐，建立一个关于我们实验结果的可信的证据集合。

14.3 软件工程中的决策制定

伴随着对新技术的需求而来的是作出最后决策的需求。我们怎样能够知道下一个项目中我们应该使用那些技术呢？我们怎样能够将资源正确的分配到工作小组呢？我们怎样来衡量使用一种技术的风险会小于其它的呢？像应用技术转移一样，我们可以研究一下技术决策上的规则来帮助我们做出能够做出正确的选择。

决策参考

有时候，软件工程就像一串由决策选择和估计联系起来的被强迫执行的行为的集合。我们通过估计资源和风险来规划项目；通过判定我们的过程是否有效和资源是否分配恰当以及产品是否满足要求来评估项目；为了测试产品，我们从中抽样。

我们不需要制定那种纯粹以上的决策，现在已经有了从两种观点出发支持决策制定的理论：说明性理论和规定性理论，说明性理论为决策在现实中如何制定提供证据；规定性理论为决策制订者改进问题发现和问题解决过程提供标准的框架结构和方法，并给他们一些限制条件。图 14.3 详细说明了二者的区别。

通常，决策制定包括两个明显的步骤。第一，我们分别进行选择。第二，经我们各自的发现应用到整个群组的决策制定当中。例如，为了估计出现一个特别类型的软件我们需要付出的努力，我们每个参加者可以先做自己的预测然后在将各自预测的结果联合起来得到整个项目所需要付出的整体努力到底有多大。这里的群组我们可以理解成我们的项目、整个组织或甚至是我们的整个社会。

图 14.4 显示出有许多因素将影响我们做出的决定。上下文相关的条件将限制我们的理解和观点。在这些上下文条件中，我们必须在解决问题之前将它理解和表示出来。每一种观点必须以多种方式隐藏起来，以决定它合理性和现实性。“合法化”非常重要，但是它又有时候在软件工程过程中被忽略。可以想象，一种被估测者和决策制订者制定的估计和选择的优化将被容易的验证。这种优化体现了一种为达到特殊的目的而采取得行为上的偏见。就象用算法库模型来代替专家的判断一样。同时，可能出现的解决问题的方式必须经过我们的价值和信任的过滤。

为了明白我们如何在决策制定中使用这些因素，让我们看看选择办公空间的例子。表 14.1 种显示了 5 种可能的选择意见。每一种选择的方法都由每月的租金、离家的距离、办公室的面积、和空间的质量来标识。

有许多我们可以使用从中选出最优的规则。 例如，我们可以选择租金最低的房间；或是可以选择离家最近的办公室。这些规则将反映出我们侧重的价值。选用第一种规则的人重视钱的因素要大于时间。除此之外，我们还可以用一些其他得更复杂的规则。例如，我们可以将租金和大小合起来定义“办公室的价值”，同时我们用最短的从家到办公室的时间来平衡二者。或者我们可以用多步趋近的方法，第一步参考租金与距离，然后下面的步骤中继续平衡其它的参数。

当然我们的选择过程还可以变得更加高级一些。例如，我们可以用控制过程的方法。在控制过程中我们评估各个被控制的选项。然而这种规则可能会使我们得到一种次优的结果。假如我们设计的选择过程是任意设计或是没有经过周密的设计那么我们就无法得到最优的解。或者我们还可以用关联的方法（可选项的每个参考量都必须满足一个定义了的标准）；非关联的方法（可选项的每个参考量多选足够高的选项）。这样，当特征值接近极限的时候就会产生一种疏漏；例如，我们抛弃那些租金大于$500 的选项，但是对某些人来讲，有些租金是$501 的可能会是最好的。

另一个策略是应用通过外表的消去法。这里，每一个属性都有一个预先设置的值。而且每个属性还被制定了一个优先级。然后属性将被依据他们各自的重要性被一一评价。我们可以用一个附加的值模型（value model）来形势化这个过程。在这个模型中，我们首先为每个属性（xj）标定优先级（wj），最后将各个属性值和优先级的乘积相加求和。如公式所示：每一个这种选择的过程都说自己是正确的选择，但是有些时候某些选择事实上并不是最好的选择。在实际应用中我们可以选用启发式趋近的方法来得到一个较好的方法。

群体决策制定

到现在为止，我们已经与问题本身相关的特征。群体决策制定在某种程度上更加难以实现。因为群体行为的每个方面都将影响最后的决策制定结果。图 14.5 说明了当几个人想在可选的项目中做出选择时所必须考虑的因素。例如：相互信任、相互交流、合作。在这些因素中没有一个是在个体决策制定时需要考虑的因素。

也有集中群体决策制定方案来处理这些关心的因素。例如，辩证的方案。它允许一方面提升论点；另一方面讲。第三方将被引入用以调节分歧。一个智囊团将被用来为一系列的不确定的诸如：选择时机和威胁等问题做出最后的确认。

当群组是一个组织的时候一定要从传统的常规的决策中区分出基于策略的方案。战略决策制定将会影响到组织的本性。他们包括新产品、服务和市场。而且高级的管理讲起到重要的作用。代价估计也是战略决策制定的一部分。尤其是在他们被用于将产品推向市场的时候。传统的决策制定可以影响价格、雇员的分配等但是它不能影响到组织的根基或者大的商业方向。

日程的决策制定是较为平常的：本质相同的一种重复活动、在本地的范围内、由组织的规则原则来指引。例如，假设一个公司支持他自己产品的复用，软件工程是将会因为向复用库存进一个可复用的使用模块而得到回报，同样的，他也会因为使用复用库中的产品而被收取一定的费用。决定这些复用组件的价格就将是由公司指引的一项日常的工作。

我们是怎样制定决策的

决策制定科学和操作研究著作是由一些决策制定的例子来创建的。但是当我们在制定决策的时候究竟会用哪些技术呢？由 Forgionne 作的一项研究指出我们一般都倾向用统计和仿真的方法。只有很少数的人会用到较为复杂的过程。

有许多我们尽量避开应用复杂技术的原因。最大的在应用他们中遇到的困难就是计算庞大。与那些只在假设种分析最有的决策制定相比，Klein 已经观察了现实工作中的决策制订者。在进行的 156 次观察中，他发现没有人用制定预先选择的方法。18 个人使用了比较评估的方法。先选定一个初始的选择观点，然后用其它的与之相比较，最后选择最优的。有 11 个人自创了新的选择方法。其它的人都应用了 Simon 称之为满意策略的方法：他们用自己确定的价值来评估每种方法，直到找到第一个合适的方法为止。

在观察研究了战斗机飞行员、紧急医疗技术和士兵等必须在经常在压力下做出决策的人，Klein 提出了一种“初始识别决策制定模型”（如图 14.6 所示）来描述我们是怎样制定决策的。他指出，我们现在脑子里村进了大量的实例，当我们面对一个要做出的选择时，首先我们先将它与我们脑子里的事例进行对比，看看有没有相类似的事例。最后我们在头脑中找到一个我们认为是最接近的一个实例来作为本次选择的根基。然后，我们用我们的意识模拟来判断在当前的条件下这种选择是否是正确的。如果不正确，我们在返回上述的寻找过程，直到找到一个合适的为止。

Hershey, Kunreuther, Schoemaker 指出一个决策的上下文描述将会影响到选择果，为什么？让我们先看看下面的两个问题。

1. 你现在就有 50%的可能丢失$200 元，和%50 的可能不丢$200 元。那么你是否愿意付$100 来避免丢失发生呢？

2. 你有支付$100 的能力来避免这种丢$200 的情况发生，那么你愿意支付这笔钱么？

在他们的研究中，他们分别问两个相同的组这两个问题，并且要求他们用“yes”或是“no”来回答。在回答第一个问题的组中，有 6%的人回答 yes.
在回答第二个问题的组中 32%的人回答 yes。

Tversky 和 Kahneman 分析了在风险分析决策者决定过程中的这种相同类的偏见。他们描述了一个环境，在这个环境中一种疾病正威胁着村子中 600 人的生命。公共的医疗室目前只有供 200 人使用的疫苗，他们想象了两种不同的可能。一种是将 200 支疫苗只给 200 人使用，从而救这 200 人；另一种可能，他们将 200 支疫苗放在水里，希望可能会救出 600 人的生命。研究人员问了一群人来在下面两种可能下确定它们的选择。

A: 只有 200 人获救

B: 有三分之一的可能救出 600 人，三分之二的可能所有人都死掉

同样，研究人员问了与上组等价的一个组从下面的选相中作出选择。

C: 将有 400 人死亡

D: 有三分之一的机会每人死亡；但是有三分之二的可能 600 人死亡。

虽然两组问题实质上是一样的，但是，对选择的结果却大不相同。

对于第一组，有 72%的人选择 A; 但是对于第二组有 22%的人选择 C。

在相似的研究中也发现，人们总是将过去的行为预先在的选择联系在一起。所以对于上

下文的详细斟酌是十分必要的。不同的问题提出方式将会得到不同的选择结果。

除此之外，还有其它的一些在决策制定中常常产生偏见的因素。例如，人们往往过高的估量他们所拥有的东西。这种现象会使对于一种在相似的环境中开发的产品的估计过于乐观。类似的可能性和产品回报也是起作用的因素。

一般的人都显示为乐于接受一些实例性和单一的信息而反对一些统计和分类性的信息。Busby 和 Barton 在研究那些服务于一个有严密的组织机构的组织或正在进行一种对预测途径进行细目分目的工作的评价者时集中研究了这种偏爱倾向。不幸的是，这些途径不适应那些没有被规划的行为。因此，那些预测途径一直被低估 20%。从定义上讲，每个项目的特殊实例的证据不能够说明没有规划的行为。然而，跨越个项目之上的统计的证据可以说明没有规划的行为是可以发生的。

另外，我们必须记住回想的结果通常是由我们记忆中的实验的崭新程度和鲜明程度所决定。如果一个过去发生的实验越久远，回想者就越倾向于忽略它的重要性。

最后的确认和调节

最后的确认和调节是由评价者要掌握另外一项技术。这里，评价者选择一个相似的条件，然后将它适用在新的环境中。然而，这个过程会有相当多的证据证明评价者的行为是不够细致的。这种倾向由评价者的某种倾向所影响，因为大多数的更加适合的类似情况将被忽略，而仅仅因为他们不是最近发生的。

一个群组现实怎样制定决策

许多组织机构应用群组决策者定的技术来决策重要的项目。但是组织的动态性将影响组织作出决定或是评价质量。例如，Foushe 发现组织成员要共同达到多产的目的将需要一定的时间。成员小组在最后的工作要优于在开始的时候的工作。因为在整个过程中他们逐步的学会了合作在一起有效率的工作。

组织的动态性也会带来不好的影响。Asch 验证了在一项个人决策制定的过程中，其他同事对结果的影响。他画了一个图，如图 14.7 所示。当被问图中的 A,B,C 哪条线预测试线一样长时，几乎所有的人会说是 B, 但是当在同一个组里的其他人有人给了一个错误的答案时，将会产生不同的结果。如表 14.2 显示。

一种适度的观察性研究

为了检验软件工程中的群体决策制定。Pfleeger，Shepper 和 Tesoriero 展示了一组影响估测结果的可能的趋向。他们在 Bournemouth University 展开了一项调查。12 名研究生被按照人数从 2 到 4 分成 4 组。作为研究的一部分，这些小组被要求发现需求并开发出一个简单的信息系统原型。他们然后要求小组继续原型用代码行数表示的大小。该题目不限制用任何技术，虽然实际当中他们都倾向于应用主观判断的方法。很快这个参加了 Delphi 会议题目引出了另外的两种评价。

表 14.3 显示无论是中间的错误还是扩展的错误和范围上的错误在初始的评价中都占有很大的比例。在接下来的轮回中我们可以看到在预测值和实际原型的值之间的差距将逐步的减小。如图 14.9 所示，4 个小组中的 3 个已经明显的表现出改进的趋势。但是第 4 组的预测与实际值之间始终存在着一个偏差，这个偏差可能是因为组中的某一成员的决断而造成的。Delphi 技术允许匿名估计，因为那些有利的决策者将常常受到攻击。如我们所看到的，一个决策制定的结果使决策制定学这技术以及群组成员如何参考专家的意见最终作用的效

果。

经验总结

从对 Delphi 评估技术的研究中我们可以学习到许多经验。第一个就是所选择的主题必须对这种技术有一种正确的积极态度。实例中也证明，使用这种技术可以提高评价结果的质量。

第二个经验是人的个性可以支配最后的结果，甚至当这个占支配地位的决策者并不正确的时候。而且，个人的带有偏见的建立初始评价结果的假设与接下来的群组的讨论结果不相关。这种结果与在群组会议关于评论和检查结果的调查报告所体现的内容相一致。在此报告中，许多个人的发现，当它们不能得到群组中其他人的认可时就将被最后抛弃。

我们通常会认为，那些经验丰富的人会深刻的影响群组讨论的结果。指引出一种更加现实的评价结果。但是在 Pfleeger, Shepperd, Tesoriero 的研究中，甚至最有经验的群体也要通过中间的结果作重复的确认。因此，在 Delphi 的讨论中，个性是可以支配经验因素的。

大多数的当前的关于评价技术的研究都集中在个人评价的精确确定方面。然而，大多数的参与个人都是在一个群组的环境下做出的评价。或者是依赖于像 Delphi 这样的技术或从其它的同事的观点中得到暗示而得出的。因为这个原因，能够认识到在评价过程中群组的动态变化特性是十分重要的。我们通常假定一些专家的观点和经验的因素将是起支配性的因素，拥有了高质量的历史相似项目记录数据使得我们能够勾画出一个模拟的评价结果，以及如何确认出两个项目的相似性。不幸的是，许多的这种假设在某种程度上是错误的。参与者必须依靠个人和群组的工具来生成项目的决策。虽然我们强调客观性的必要性，事实上在实际当中我们则常常用到主观的因素。基于观察的研究结合上对群组动态特性的完整理解竟会帮助我们制定正确的决策制订过程，提高我们对最后结果的信心。

14.4 软件工程的未来

软件工程已经经历了一个长期的发展过程。但是，如果要将软件工程的成熟标准按照其它的工程的原则来制定的化，那么软件工程还将有一段较长的路要走。美国的一些州现在仍在继续软件工程师的培训和认证工作。有很多组织号召确立一个属于软件工程范畴的知识体，确立正式的软件工程大学课程和认证考试。这些努力的结果会使一些目前的难题得到解决，例如：软件工程是多少创新和工程的结合体？

为了回答这些问题，我们学习与我们经历了相似任务的其他软件工程师所使用的方法，这样我们才可以从其它人的经验中获得知识。同时，我们还得学习与我们经历不相似任务的软件工程师们使用的一些方法，这样我们才可已将我们的处理策略、技术和工具适应到我们遇到的实际问题中去。更一般的讲，我们必须在更宽的领域中来研究软件工程。认识到高质量的软件产品和处理过程是由有创意的人在工作小组中开发出来的。不是一个简单的制造过程生产的。我们还应该遵循一些其它的原则包括，社会科学等等。这样我们的处理过程可以利用每个软件工程师提供的最好的方法；我们的软件产品可以为用户提供最优秀的应用、服务。最后，我们还得注意软件工程决策制定的结果。谁将在我们软件错误的时候担起责任？认证和培训将扮演一个什么样的角色？谁将在我们的需求、设计、应用和测试出现错误时担负起道德上的和法律上的责任？像其它的成熟的原则一样，我们必须学会在我们的行为和产品中负起职责。