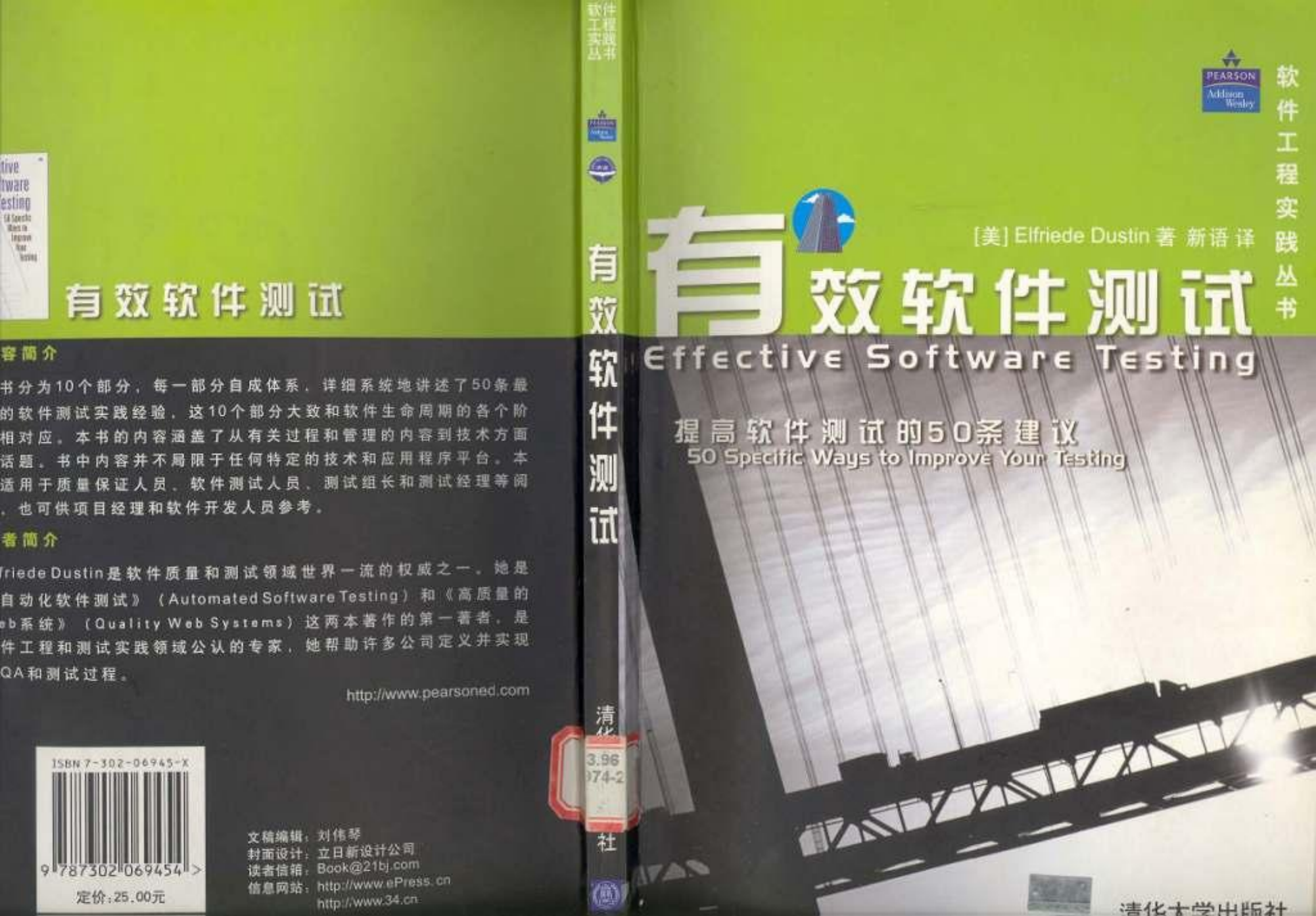

软件工程实践丛书
有效软件测试
清华大学出版社

有效软件测试

内容简介

本书分为10个部分，每一部分自成体系，详细系统地讲述了50条最佳的软件测试实践经验，这10个部分大致和软件生命周期的各个阶段相对应。本书的内容涵盖了从有关过程和管理的内容到技术方面的话题。书中内容并不局限于任何特定的技术和应用程序平台。本书适用于质量保证人员、软件测试人员、测试组长和测试经理等阅读，也可供项目经理和软件开发人员参考。

作者简介

Elfriede Dustin是软件质量和测试领域世界一流的权威之一。她是《自动化软件测试》（Automated Software Testing）和《高质量的Web系统》（Quality Web Systems）这两本著作的第一著者，是软件工程和测试实践领域公认的专家，她帮助许多公司定义并实现了QA和测试过程。

http://www.pearsoned.com

ISBN 7-302-06945-X
9 787302 069454 >
定价：25.00元

文稿编辑：刘伟琴
封面设计：立日新设计公司
读者信箱：Book@21bj.com
信息网站：http://www.ePress.cn
http://www.34.cn

软件工程实践丛书

有效软件测试

[美] Elfriede Dustin 著

新 语 译

清华大学出版社

北 京

内 容 简 介

本书的内容涵盖了从有关过程和管理的内容到技术方面的话题。书中内容并不局限于任何特定的技术和应用程序平台。本书适用于质量保证人员、软件测试人员、测试组长和测试经理等读者阅读，也可供项目经理和软件开发人员参考。

北京市版权局著作权合同登记号 图字：01-2003-2088

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

有效软件测试 /[美]达斯汀著；新语译.—北京：清华大学出版社，2003
（软件工程实践丛书）
书名原文：Effective Software Testing
ISBN 7-302-06945-X
Ⅰ.有… Ⅱ.①达… ②新… Ⅲ.软件—测试 Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字（2003）第 064036 号

出 版 者：清华大学出版社　　地　　址：北京清华大学学研大厦
http://www.tup.com.cn　　邮　　编：100084
社 总 机：010-62770175　　客户服务：010-62776969

文稿编辑：刘伟琴
封面设计：立日新设计公司
印 刷 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：170×230　印张：12.5　字数：234 千字
版　　次：2003 年 8 月第 1 版　2003 年 8 月第 1 次印刷
书　　号：ISBN 7-302-06945-X/TP・5132
印　　数：1～4000
定　　价：25.00 元

译 者 序

我曾经在国内一家知名的软件企业工作了 8 年，在此期间一直是核心项目组的成员之一。我们负责开发一个专业性很强、技术含量很高的产品，这一点和本书中提到的财务系统有些接近。虽然我并没有真正从事过软件测试工作，但是作为一名开发人员，经常需要和测试人员打交道，所以我有机会感受到了测试工作乃至整个软件工程环境的变化。

这家企业的软件工程环境的发展大体可以分为 3 个阶段：

精英软件工程环境

我刚进入这家公司时，计算机工业在中国处于超高速发展期，这个行业的待遇和福利相对较高。而且当时计算机还没有现在这么普及，计算机行业对于普通的大学毕业生来说充满了神秘感和吸引力。

我们公司的前身是一所国内著名大学的校办产业，创业者基本上都是这所学校的职工。当时有相当部分的员工一个人有两个身份，既是公司的职员，又是学校内某个部门的职工。因此在项目开发中不可避免地继承了一些学校办事的风格，例如：重技术、轻管理。

我刚进公司的时候，企业仍然保持着强劲的上升势头，虽然规模不大，但是效益很好。职工的进取心也很强。

当时我们公司的主要用户是一些大客户，这些大客户的共性很强（从后来的结果来看，他们对产品的功能需求非常片面，所以对开发人员和测试人员的行业知识要求不高）。并且当时卖方市场的竞争不激烈，用户对产品质量的要求也不是太高。

由于以上的大环境、小环境和其他一些因素，当时的软件工程环境有以下一些突出特点：

- 人员素质很高。无论是开发人员还是测试人员大都是名牌大学的优秀毕业生。

- 开发人员和测试人员都不是行业专家。
- 校园文化对企业文化影响至深，现代企业制度不完善，制度的执行也缺乏力度。

当时负责我们项目测试工作的测试人员一共有3位，个个都是以一当十的强人。虽然他们缺乏基本的行业背景，对计算机技术也缺乏了解，但是由于他们的素质很高，学习能力很强，再加上当时对行业知识和计算机技术的要求不高，所以他们很快就掌握了测试工作所需的各种知识和技术。

虽然当时制度有漏洞，但是由于企业的小环境很好，职工大都责任感很强，所以不完善的制度并没有过多地影响开发、测试工作。

测试人员的高素质特别体现在当他们再次从事同一产品的后续版本的测试工作时，尽管他们并没有正式的渠道了解软件技术、软件结构，但是通过与开发人员的非正式交流和经验，他们已经能够猜到哪些部分容易出问题，并且一旦某个问题暴露了以后，他们立刻会想到哪些地方也可能出现类似的问题。因此测试效率比较高。

从现在的观点来看，当时的软件工程环境的最大问题是非“结构化”的。但是职工出色的个人能力和敬业精神在很大程度上弥补了制度上的不足。

标准化的软件工程环境

随着时间的推移，企业的扩张，第一代测试人员由于种种原因，有的走上了领导岗位，有的转向其他性质的工作（可能是厌倦了测试工作吧），公司迎来了第二批测试人员。要求也不是那么高了，他们的专业素质和个人素质与第一代测试人员相比有所下降。由于这些人毕业时间较晚，所以他们的知识结构和第一代职工有一定的差异，但对软件工程有了比较系统的认识。

企业为了发展，需要拓展产品的生存空间。与此同时，国外的产品开始进入国内市场，市场竞争日趋激烈。我们的产品在新市场上遇到了挫折，用户不再总是唱颂歌，而是开始有了不少的抱怨。在这种背景下，我们公司出现了第一次有关软件工程环境的激烈争论，争论的主题是：谁应该为软件产品的质量负责？

整个事件的导火索是在一次产品会议中，一位技术骨干对测试负责人说：“产品的质量出了问题，全部是测试部门的责任。”而测试负责人显然对软件工程的理论有一定的了解，所以当然对这个在软件工程领域中常见的错误观点给予了猛烈的反驳。会议以后，

争论的方式改成了邮件，并且相继有多人加入了讨论的行列。在争论渐渐平息之后，公司开发部门的总负责人认为这个讨论很好，于是把经过多人多次回复的邮件转发给所有开发人员和测试人员，当时邮件的总长度可能已经超过了一万字。

此时企业的规模几乎达到了顶峰，在经营上也开始出现了一些不好的苗头：经过了多年的高速发展，增长的速度已经放缓，员工创业的激情逐渐消退；市场竞争开始加剧，公司的利润率开始下降。各种内忧外患使得公司高层忧心忡忡，他们迫切地想找到一条企业发展的长久之计。与此同时，国家开始推广现代企业制度，并且在政策上给予了强有力的支持。最终公司高层决定实施ISO 9000标准。

ISO 9000的实施在客观上极大地提高了测试部门的地位，系统测试的重要程度也达到了一个前所未有的高度：产品的测试报告可以决定一个项目的命运，可以一票否决一个产品的发行。

此阶段软件工程环境的特点是：

- 初步建立了标准化的软件工程环境。
- 开发人员和测试人员的素质整体上有所下降。
- 由于市场的拓展，相关人员在行业知识方面的欠缺变得突出。

ISO 9000质量体系虽然在公司内建立起来了，但是由于观念上的问题，核心的开发人员都不愿意从事质量保证工作，所以建立的ISO 9000体系本身和实施过程都非常教条，开发人员对ISO 9000认可程度不高。此外，测试部门的权利过大，造成了开发人员和测试人员的矛盾。当测试人员的决定影响了公司或者部门的产品策略时，也造成了测试人员和市场人员的矛盾。

由于测试人员和开发人员分别属于不同的部门，他们之间的交流不多，因此测试工作的有效性很成问题，另外，测试人员的基本素质和行业知识方面的欠缺渐渐成为影响测试工作质量的重要因素。

总而言之，虽然公司在软件质量保证方面下了很大的力气，但是标准化的软件工程环境并没有起到应有的作用，所以此阶段开发的软件产品的质量也没有多大的提高。

务实的软件工程环境

到了这个阶段，公司的内部和外部环境更加恶劣。为了重新在市场上取得优势，市场人员要求开发部门对来自市场上的信息快速做出反应，因此各个开发部门的压力越来越大。在这种情况下，开发人员和市场人员对测试部门“轻易”否决他们负责的产品怨言很多，认为根本无法实施他们的产品战略。

为了扭转这种不利的局面，公司高层决定调整公司构架。调整的重点之一是把开发人员、测试人员合并到各个事业部，这种调整在客观上提高了市场人员的权利，但是削弱了测试部门的权利。测试部门重新回归到一个从属的地位。在这个阶段也发生了一起标志性的事件：有一位坚持原则的测试经理，坚决不同意发行一个他认为问题较多的产品。产品经理和开发组长万般无奈，只好找到上级领导向这位测试经理施压。在压力下，这位测试经理最终还是在产品的质量合格证上签了字。从此以后，所有的测试评审工作均由一个评审委员会完成，不再是一票否决，也不再由测试经理独立完成。而测试经理们也学乖了，他们总是把产品的遗留问题和对产品的担忧一一列出，不再对产品是否应该发行发表正式的意见（事实上，在新的公司构架下，他们的影响力实在有限）。

随着公司员工正常的新陈代谢和质量体系潜移默化的影响，软件工程的理念逐渐深入人心，开发人员不再对软件工程环境所涉及的各种规章制度有过多的抵触情绪，并且开始从中受益。

测试部门的领导已经意识到了行业知识的重要性，并且在员工培训和招聘工作中力争弥补这个欠缺。此阶段招收的第三代测试人员开始向相关专业的毕业生倾斜。

由于公司经营状况不佳，所以公司高层对费用的控制越来越严格。他们认识到一旦到了测试阶段，开发费用实际上已经花得差不多了，到这个时候再决定枪毙一个项目对公司的风险太大。因此公司高层开始考虑对整个软件生命周期进行控制，他们决定实施更符合软件企业的 CMM 规范。

此阶段的软件工程环境的特点是：

- 软件工程环境逐步完善，大家对软件工程环境的建设从理想主义转向务实，对整个软件生命周期各个阶段的理解也逐渐加深。
- 重视测试人员的行业背景。
- 对市场人员的意见空前重视。

到了这个阶段，公司终于开始从结构化的软件工程环境中受益了。虽然公司的内部和外部环境没有起色，开发人员和测试人员的基本素质也呈下降的趋势，测试部门不再具有否决一个产品发行的权利，但是由于整个公司的软件工程环境不断完善，所以产品的质量还是有所上升。

然而，此阶段软件工程环境的缺点也是非常明显的。主要问题是：

- 由于企业经营状况不佳，开发资源和测试资源空前紧张。
- 经过多年的积累，测试用例库的规模增长很快。同时，由于市场领域的不断扩展，需要测试的内容越来越多，测试的周期越来越长。
- 测试人员的行业知识不再成为问题，但是他们的基本素质和技术背景却成了制约测试工作有效性的主要原因。

怎样阅读本书

从上面的例子可以看出，这家公司用了多年时间才把软件开发工作初步纳入正轨。可见，一个好的想法在付诸实施以前，还应该考虑许多非技术的因素。在下面的观点中，有些在本书中已经提到了，有些是我多年实际工作的总结，与想在本企业中掀起一场质量保证革命或者改良运动的各位朋友分享。

- 任何软件工程环境的改进活动，其目标、实施策略必须和企业或者组织文化的精神一致。
- 必须获得公司高层、部门高层的支持。这一点甚至是 CMM 实施过程中的第一要素。
- 任何工作都是由人完成的，高素质的人才可以弥补制度上的缺欠，而且有些想法只有高素质的人才能成功实施。
- 软件工程环境的完善程度，取决于公司或者部门的投入。在改进活动中一定要量力而行，不要追求根本不可能实现的目标。

虽然本书是一本实用性很强的书籍，但是本书还是给出了大量测试生命周期中的原则。我认为，深刻地理解这些原则对读者的帮助会更大，因为它们适用的期限会远比具体的方法长得多。例如：测试人员应该及早介入软件开发周期中的需求阶段，这条原则

至少 30 年不会动摇。我希望本书的读者都能以这种态度阅读本书。

致　谢

我要感谢 Jane 和 Run 为本书作了认真细致的校对工作。他们的意见为本书增色不少。

我更要感谢清华大学出版社的尤晓东先生和刘伟琴女士。感谢尤晓东先生给了我这次宝贵的学习机会，感谢刘伟琴女士为本书所作的具体工作。

最后，我还要特别感谢本书的读者，如果你们能从本书中汲取一点有用的营养，得到一些帮助，那么我将会感到无限欣慰。这是我从事翻译工作的初衷。

由于时间仓促以及水平有限，错误之处在所难免，敬请读者批评指正。

前　言

在大多数软件开发组织中，测试工作是应用程序的最后一道“质量关”，它允许或者拒绝应用程序从理想的软件工程环境进入真实的现实世界。伴随着这个角色的是巨大的责任：应用程序的成功、甚至可能是组织成功的赌注全部押在软件产品的质量上。

众多小规模的测试任务必须由测试组来完成和管理，事实上，由于测试任务的数量过多，所以测试工作关注的只是测试一个应用软件的功能，很少顾及测试工作需要的外围工作。获得合适的测试数据、应用程序需求和构架的可测试性、适当的测试过程标准和文档，以及硬件和设备等诸如此类的问题即便不是没有考虑，也往往会到项目生命周期的后期才予以考虑。对于规模较大的项目，单纯使用测试脚本和测试工具是不够的，大多数有经验的软件测试人员都会认同这一点。

一般情况下，我们只有通过具体实践才能熟知使测试工作从头到尾都获得成功所需要的各个环节的有关知识。在项目生命周期中，如果某些任务完成得早，那么测试工作的效率就会高得多，这是一条有价值的经验。当然，在我们意识到这一点时，当前的项目通常已不可能从这条经验中获益了。

本书提供了一些实践经验和关键理念，组织可以利用这些实践和理念成功地实现有效的测试工作。本书的目的是为读者提供各种经过精心挑选的技术和建议，软件从业人员能够直接用它们来改进产品，同时也能避免损失惨重的过失和疏忽。本书详述了 50 条最好的软件测试实践，分为 10 部分进行讲述，这 10 部分大致和软件生命周期的各个阶段相对应。这种结构本身就阐述了软件测试工作中一个关键理念：为了获得最大的有效性，测试工作必须完全融入软件开发过程。我们必须避免把测试工作当成“项目流程”中一个独立的步骤（在软件开发周期的最后阶段），这是一个很常见的错误。

本书的内容涵盖了从有关过程和管理的内容（例如：管理变化的需求和测试组的构成）到技术方面的话题（例如：提升系统可测试性的方法和把单元测试融入开发过程）。虽然在需要的地方本书中也出现了一些伪代码，但是本书的内容并不局限于任何特定的技术和应用程序平台。

但是，还有很多测试工作以外的因素也会强烈地影响项目的成败，认识到这一点非常重要。虽然包含测试工作在内的完整软件开发过程会确保与软件工程有关的工作成功，但是所有项目还要处理有关商业用例、预算、时间表和组织文化方面的问题。在某些情况下，这些问题会和高效的工程环境的需要发生冲突。应用本书中的建议的前提是：为

了测试工作的成功，组织能够适应测试工作的需要，并且为这些需要提供支持。

本书的组织结构

本书由 50 个独立的条目组成，它们覆盖了 10 个重要的方面。这些经过挑选的最优实践出现的顺序和系统开发生命周期的各个阶段的先后顺序一致。

读者可以一条接一条、一章接一章地顺序阅读本书。当需要获得或者了解有关一个特殊问题的信息时，也可以直接跳到特定的条目。虽然在每一章中会引用其他章节或者其他书籍的内容，它们会有助于向读者提供更多的信息，但是每一章的内容基本上是自成体系的。

第 1 章描述了测试工作在需求阶段需要考虑的问题。在需求阶段，包括测试组代表在内的所有涉众必须参与需求工作，并且必须收到需求变更通知，这是非常重要的。此外，对于任何大型项目来说，基于需求开发测试用例都是一个最基本的理念。测试组参与此阶段工作的重要性怎么强调都不过分，只有在这个阶段才能获得对系统和它的需求的全面理解。

第 2 章描述了测试规划活动，其中包括：对测试工作目标的了解方法，确定测试策略的方法，以及有关数据、环境和软件本身需要考虑的问题。在软件生命周期中，规划工作必须及早开始，这是因为我们需要为成功地实施测试工作预留时间。及早规划使得我们可以对测试进度和预算进行估计，并且使之获得批准和加入整个软件开发计划。我们必须不断地监控这些估计，并且和实际情况进行比较，这样就可以根据需要对它们进行修正，并且实现预期的目标。

第 3 章主要讨论测试组的人员构成。所有成功的测试工作的核心是执行它的人。一个成功的测试组需要同时具备技术和行业两方面的知识，还要有结构化和简明的角色与职责划分。为了确保测试工作成功完成，在整个测试过程中，必须不断地评估每个测试组成员的有效性。

第 4 章讨论了有关测试系统构架方面的考虑。为了保证系统本身是可以测试的、能够进行灰箱测试和有效进行缺陷诊断，考虑这些因素是非常重要的，但是它们经常被忽视。

第 5 章详细描述了如何有效地设计和开发测试过程，其中包括在测试创建和文档化方面需要考虑的问题，还讨论了最有效的测试技术。随着时间的推移和系统开发迭代的

继续，需求和设计会不断精化，因此测试过程也要不断精化，它们需要加入新的和修改后的需求以及系统功能。

第 6 章讨论了在整个测试策略中，开发人员进行单元测试所扮演的角色。在实现阶段中，单元测试会显著地提高软件质量。如果全面地执行了单元测试，以后的测试阶段会更成功。但是，基于对问题的了解的、随意的单元测试和基于系统需求的、结构化的、可重复的单元测试是有区别的。

第 7 章讲解了有关自动测试工具的问题，其中包括：在项目中使用恰当的工具类型、有关定制开发还是购买的决定和为组织选择恰当的工具需要考虑的因素。本章描述了多种类型的测试工具，它们可以用于开发生命周期的各个阶段。此外，本章还讲到了开发定制工具方面的问题。

第 8 章讨论了为自动测试选择最佳实践方面的问题。本章描述了如何正确地运用记录/回放工具、自制测试工具和回归测试。

第 9 章提供了测试一个应用软件非功能性方面的信息。如果满足了应用程序的非功能性需求（包括性能、安全性、可使用性、兼容性和并发性测试），那么会提升应用程序的整体质量。

第 10 章提供了测试执行的管理策略，其中包括：追踪测试过程执行和缺陷生命周期的正确方法以及收集用于估计测试进程的度量。

本书面向的读者

本书面向的读者包括：质量保证人员、软件测试人员、测试负责人和测试经理。本书中提供的大部分信息对于项目经理和软件开发人员也是有价值的，他们可以利用这些信息改进软件项目的质量。

致 谢

感谢所有在本书撰写过程中向我提供帮助和支持的软件专业人士，其中包括参加了本人讲授的自动软件测试（Automated Software Testing）、高质量的 Web 系统（Quality Web Systems）和有效测试管理（Effective Test Management）这几门课程的所有学生，各个公

司中协助本人完成各种测试工作的合作者，还有本人各种著作的合著者。他们提出的问题、见解、反馈和建议直接或者间接地增加了本书内容的价值。特别感谢 Douglas McDiarmid 为本书做出的杰出贡献。他提供的信息极大地丰富了本书的内容，同时也极大地提升了本书素材的整体质量。

还要感谢下列人员，他们提供了非常有价值的反馈。他们是：Joe Strazzere，Gerald Harrington，Karl Wiegers，Ross Collard，Bob Binder，Wayne Pagot，Bruce Katz，Larry Fellows，Steve Paulovich 和 Tim Van Tongeren。

还要感谢 Addison-Wesley 的工作人员对本书的支持，特别是 Debbie Lafferty，Mike Hendrickson，John Fuller，Chris Guzikowski 和 Elizabeth Ryan。

最后，还要感谢 Eric Brown，他为本书设计了有趣的封面。

Elfriede Dustin

目 录

第1章 需求阶段

最有效的测试工作应该始于项目的开始阶段，远远早于程序代码的编写阶段。首先需要检验的是需求文档，只有如此，在项目的后续阶段测试工作才能专注于保证应用程序代码的质量。在项目生命周期的早期——详细设计和编码工作之前，消除需求工作中的缺陷能够使昂贵的返工工作降到最低。

软件应用程序或者系统的需求说明书必须非常详细地描述它的功能。与提供需求的人员进行交流是确定需求的工作中最具挑战性的工作之一。每条需求必须阐述得准确和明确，这样它的读者对需求的理解才会完全相同。

如果使用了一致的方法来撰写需求文档，那么任何需求收集人员就有可能有效地参与需求过程。一旦某条需求浮出了水面，那么通过详细咨询相关人员这条需求就可以被测试和澄清。我们可以利用各种各样的**需求测试**来保证每条需求是恰当的，并且大家对它的含义有相同的理解。

第 1 条：测试人员及早介入

测试人员需要从项目生命周期之初就开始介入，这样他们才能准确地理解测试的对象并且和其他涉众一起生成可测试的需求。

缺陷预防是指在各种错误遗留到后续开发阶段之前，运用各种技术和过程来发现和避免这些错误。缺陷预防工作在需求阶段的效率最高，此时修正缺陷所需的改动很小：需要改动的仅仅是需求文档，可能还需要相应地修改此阶段制定的测试计划。如果测试人员（和其他涉众一起）从开发生命周期之初就开始参与项目，那么他们就能够协同发现遗漏、矛盾、含糊的地方和一些其他问题，这些问题可能会影响项目需求的可测试性、正确性以及其他测试质量。

对于一条需求，如果可以设计出一个过程来执行所测试的功能，若输出结果是可以预先知道的，并且能够通过编程或者人工方式加以验证，那么我们就称这条需求是**可测试的**。

测试人员需要彻底地了解产品，只有这样他们才可能设计出更出色和更全面的测试计划、测试设计、测试过程和测试用例。测试组及早介入，就可以避免在项目生命周期中的后续阶段对产品的功能行为不理解。此外，测试组及早介入，还可以了解到应用程序的哪些方面对最终用户来说最关键以及哪些元素的风险最大。这样测试组就可以首先把精力集中在应用程序中最重要的部分，避免对不经常使用的部分过度测试而对重要的部分又测试不充分。

有些组织不是促使测试人员及早介入，而是把测试人员完全当作需求和其他软件开发工作产品的客户，当软件交给测试人员的时候，才要求他们学习应用程序的使用及行业背景知识。这种做法对于小项目还可以接受，但是在复杂的环境中，如果测试人员在需求、分析、设计甚至是部分实现阶段之后才接触到应用程序，那么就不能指望他们发现应用程序中所有的重要缺陷。测试人员所要掌握的不仅仅只是软件的“输入和输出”，他们还需要掌握更深入的知识，而这种知识只有了解撰写产品功能说明书的思考过程才能获得。这种理解不仅能够提高测试人员开发的测试过程的质量和深度，而且测试人员也可以针对需求提出反馈意见。

在生命周期中发现缺陷越早，那么修正缺陷的代价就越小。表 1.1 列出了在项目生命周期的不同阶段，修正发现的缺陷所需付出的代价。[①]

表 1.1 预防比纠正付出的代价小：修正错误的代价在系统开发的生命周期中急剧增长

阶段	相应的纠正代价
系统定义	$1
概要设计	$2
详细设计	$5
编码	$10
单元测试	$15
集成测试	$22
系统测试	$50
发行以后	$100+

① B. Littlewood 编辑，*Software Reliability：Achievement and Assessment*（Henley-on-Thames，England：Alfred Waller，Ltd.，November 1987）。

第 2 条：验证需求

在确定系统需求的工作中，Christopher Alexander[①]认为应该为每条需求建立一个**质量测度标准**："为每条需求建立一个质量测度标准的想法，可以把一条需求的所有解决方案分成两类：认为满足需求的解决方案和认为不满足需求的解决方案。"换句话说，如果为一条需求指定了质量测度标准，那么我们会接受任何满足测度标准的解决方案，同时拒绝任何不满足测度标准的解决方案。质量测度标准用于对照需求对新系统进行测试。

为一条需求定义质量测度标准有助于使模糊的需求明确化。例如：每个人都会认同类似于"系统必须值得购买"这样的陈述，但是可能每个人对"值得购买"都会有不同的解释。为了设计一个标准来测量"值得购买"，就必须先确定"值得购买"的含义。有时要求涉众用这种方式来考虑需求，这样就能够设计出大家意见一致的质量测度标准。但是在有些情况下，大家可能难以就质量测度标准达成共识。此时，解决问题的一个办法是用若干明确的小需求替换一个过于模糊的大需求，其中每个小需求都有自己的质量测度标准。[②]

在项目启动时定义需求开发和需求文档工作的方针非常重要。除了一些最小的程序以外，为了保证正在开发的系统的正确性，我们必须谨慎地进行分析。**用例**是一种把功能性需求文档化的方法，它能够使以后的系统设计和测试过程更加全面。（在本书的大部分内容中，粗体术语**需求**表示任何类型的规格说明书，无论是用例还是其他类型的系统功能性表述。）

除了功能性需求以外，在开发过程的早期考虑非功能性需求（例如性能和安全性）也是非常重要的：这些非功能性需求会决定技术方案的选择和存在风险的区域。非功能性需求并不赋予系统任何特殊的功能，但是它们会约束甚至定义系统如何完成给定的功能。功能性需求应该和与之关联的非功能性需求一起进行说明。（第 9 章会讨论非功能性需求。）

① Christopher Alexander，*Notes On the Synthesis of Form*（Cambridge，Mass.：Harvard University Press，1964）。

② Tom Gilb 定义了一套指定这种需求质量的符号，称作 Planguage（用于编制计划的语言）。他即将出版的著作 *Competitive Engineering* 中讲述了 Planguage。

下面是测试人员在需求阶段用来验证需求质量的检查列表。[①②]运用这一检查列表是及早发现需求中的缺陷的第一步，它使得这些缺陷不会遗留到后续阶段。一旦缺陷遗留到了后续阶段，那么发现这些缺陷就会更困难，而且发现和修正这些缺陷所要付出的代价也会更大。负责需求的所有涉众都应该用下面几个属性来验证需求。

- **正确性** 根据用户的需要来进行检验。例如：标准和规则描述得是否正确？需求是否准确地反映了用户的需要？在需求阶段，最终用户或者恰当的用户代表的介入是非常必要的。正确性也可以基于标准进行检验，需求是否遵从某个标准？
- **完整性** 用于保证需求中没有遗漏任何必须的元素。其目的是为了避免只是由于没有人提出恰当的问题或者没有人检查所有相关的原始文档而引起的需求遗漏。

测试人员应该坚持与每个功能性需求一起描述非功能性需求，非功能性需求包括性能、安全性、可使用性、兼容性和可访问性。[③]需求文档中对非功能性需求的描述通常分成两级：

1．在系统级的规格说明中定义作用于整个系统的非功能性需求。例如："Web 系统的用户界面必须兼容 Netscape Navigator 4.x 或以上版本和 Microsoft Internet Explorer 4.x 或以上版本。"

2．每条需求的描述应该包含一个标题为"非功能性需求"的部分，这一部分罗列了此需求需要的、不同于整个系统非功能需求规格说明的、特殊的非功能性需求。

- **一致性** 验证的是工作产品的内部元素之间或者工作产品之间没有内部或者外部的矛盾。通过提出问题，"规格说明书中是否定义了用到的所有重要的主题术语？"我们就可以判断需求中使用的元素是否清晰和准确。例如：需求规格说明书中的许多地方都使用了术语"观察者"，而这一术语在不同位置根据上下文有着不同的含义，那么这样会给后面的设计和实现阶段带来问题。没有清晰一

① Suzanne Robertson，其论文 *An Early Start To Testing: How To Test Requirements* 发表在 EuroSTAR 96，Amsterdam，December 2-6，1996，1996 The Atlantic Systems Guild Ltd.版权所有。未经许可，不得使用。

② Karl Wiegers，*Software Requirements*（Redmond，Wash.：Microsoft Press，Sept. 1999）。

③ Elfriede Dustin 等人，"Nonfunctional Requirements"，文章发表在 *Quality Web Systems: Performance, Security, and Usability*（Boston，Mass.：Addison-Wesley，2002），Sec. 2.5。

致的需求定义，就无法确定需求的正确性。

- **可测试性或者可验证性** 保证了测试一条需求的可能性，同时测试的结果是预先知道的并且能够通过编程或者人工来加以验证。如果一条需求不能被测试，或者不能通过其他方法来验证，那么我们就必须注明这个事实以及相关的风险，我们应该尽可能调整需求以使它可以测试。

- **可行性** 保证了需求可以在给定的预算、进度表、技术和其他可用资源的情况下实现。

- **必要性** 验证规格说明书中的每条需求是否都与系统有关。为了检查相关性或者必要性，测试人员需要对照系统的既定目标来检查需求。这条需求对实现系统目标有价值吗？去掉这条需求会影响系统实现它的目标吗？其他需求依赖这条需求吗？有些与系统无关的需求并不是真正的需求，只是建议的解决方案。

- **优先级** 让大家了解每条需求在需求涉众心中的价值。Pardee[①]建议从 1 到 5 划分 5 个等级，来奖励某条需求存在时的正面影响和惩罚这条需求不在时的负面影响。如果一条需求对整个系统的成功是至关重要的，那么它的惩罚分和奖励分都是 5。如果一条需求有了会更好但并不是至关重要的，那么它的惩罚分是 1，奖励分是 3。一条需求的整体价值或者在需求涉众心中的重要性等于惩罚分和奖励分之和——在第一个例子中价值为 10，第二个例子中价值为 4。到系统设计的时候，就可以根据这一打分结果来确定需求的优先级并对需求做出取舍。这种方法需要在用户（一类涉众）的观点和实现用户建议的需求所需承担的成本和技术风险（开发人员（另一类涉众）的观点）[②]之间取得平衡。

- **明确性** 保证了需求的陈述使用了精确的和可测量的方法。下面的例子就是一个不明确的需求："系统必须快速地响应用户的查询请求。""快速"是模糊的和主观的，因此这条需求是不可测试的。客户可能认为"快速"是指在 5 秒种之内做出响应，而开发人员则认为它是指在 3 分钟之内。反过来，开发人员也可能认为它是指在 2 秒钟之内做出响应，这样就会为不必要的性能目标花费过多的精力。

① William J. Pardee，*To Satisfy and Delight Customer：How to Manage for Customer Value*（New York，N.Y.：Dorset House，1996）

② 详细信息请参见 Karl Wiegers 的著作 *Software Requirements* 中的第 13 章。

- **可追溯性** 保证了每条需求可以通过下面的方法来确定，那就是能够找到所有引用它的系统部分。对于需求的任何变化，我们能确定系统中受这种变化影响的所有部分吗？

从这种观点出发，每条需求都被认为是一个能够单独确定的、可测的实体。同时为了正确地了解一条需求对其他需求的影响，考虑需求之间的联系也非常必要。肯定有方法能够处理大量需求以及它们之间的复杂联系。Suzanne Robertson[①]建议同时处理所有关系的方法是不足取的，较好的方法是把需求分成几个容易管理的组。这是一种把需求分配到子系统，或者按照优先级分期分批发行的方法。一旦完成了分配工作，那么需求之间的联系就可以从两个方面来考虑：首先是每个组内所有需求的内部联系，然后是不同组之间的联系。如果划分需求的原则是使不同组之间的联系最小，那么追溯需求之间联系的复杂度就会降到最低。

如果一条需求发生了变化，那么可追溯性还能够保证找到受这种变化影响的需求以及系统的其他部分；例如：设计、编码、测试、联机帮助等等。当测试人员收到需求变化的通知时，他们就会保证所有受影响的部分都经过了适应性的调整。

一旦某条需求经过评审已经确定了，那么我们就可以按照前面讲述的几个特性来测试这条需求。尽早发现需求工作中的缺陷能防止错误的需求进入设计和实现阶段，一旦到了这些阶段，那么发现和修正缺陷将会更加困难，而且需要为此付出更大的代价。[②]

为了更好地组织、规划、追踪和测试每项功能，我们应该遵从上述步骤。这样正在开发的应用程序的功能集合才是清晰的和可量化的。

① 前面引用的 Suzanne Robertson 的著作，*An Early Start to Testing*。

② T. Capers Jones，*Assessment and Control of Software Risks*（Upper Saddle River，N.J.：Prentice Hall PTR，1994）

第 3 条：需求就绪后立即设计测试过程

和软件工程师根据需求撰写设计文档一样，测试组也需要根据需求来设计测试过程。在一些组织中，设计测试过程的工作被推迟到软件版本交付给测试组以后，原因不是时间紧张就是缺乏合适的、详细说明的文档来设计测试过程。这种做法本身就有缺陷，这些缺陷包括可能遗漏需求或者在生命周期的后期才发现错误；软件实现上的问题，例如：未能满足需求；不可测试；设计的测试过程不完备。

为了让测试过程的设计工作有助于需求分析活动，它应该安排在项目过程中更靠近需求阶段的位置，而不是一直等到软件开发阶段。在设计测试过程的过程中，因为测试人员会用测试数据集合作为输入非常仔细地走查系统的每个交互操作，所以可能会发现需求文档中的某些疏忽、遗漏、错误的流程和其他错误。通过这个过程会使需求适应各种变化的场景，也会把在各种情况下的需求描述成一条由交互操作组成的清晰路径。

如果在需求中发现了问题，那么需求工作会因此返工。在整个过程中错误改正得越早，修正对软件设计和实现的影响可能就越小。

我们在第 1 条中已提到，早发现就等于低代价。如果某个需求的缺陷在整个过程的较晚阶段才发现，那么所有涉众必须修改需求、设计和编码，这些工作会影响预算、进度以及项目组成员的士气。但是如果在需求阶段就发现了这些缺陷，那么改正工作只是修改和评审需求文本的内容。

通过测试过程的定义来确认一条需求中的错误和遗漏的过程就是**验证需求的可测试性**。如果需求规格说明中没有足够的信息，或者提供的信息过于含糊不清，不能用相关的测试用例为有关的路径创建完整的测试过程，那么就不能认为需求规格说明书是可测试的，可能也不适用于软件开发。确定是否能够为一条需求设计出测试方案，这种检查总是有价值的，并且可以认为是检查一个需求是否完备的确认过程的一部分。也会有一些例外情况存在：无论是通过编程还是手动执行一个测试，一条需求均不能马上得到验证。那么我们就必须明确列出这些需求。例如："为了保存记录，所有数据文件需要存储 3 年"这条需求的实现就不能马上得到验证。但是它也需要审核、坚持和跟踪。

如果一条需求不能验证，那么它的实现正确性就得不到保证。通过确认没有遗漏重要的需求信息、标记出那些非常困难甚至是不可能正确实现和测试的需求，我们就能设计出一个能够保证需求完整性的测试过程，它包括每条需求对应的数据输入、验证需求的步骤和已知的预期输出。早为需求设计测试过程有利于尽早发现不可验证的问题。

如果将软件版本交付给测试组之后再设计测试过程，那么由于完成产品测试周期的时间紧张，测试组也容易设计出不完整的测试过程。这种风险有几种表现方式，例如：根本就没有测试过程；或者测试过程定义不完整，遗漏了会产生不同测试结果的某些路径或者数据元素。最终的结果是可能漏掉了缺陷。或者如前所述——需求可能是不完整的，并且不能为定义必须的测试过程甚至正常的软件开发工作提供足够的支持。不完整的需求经常导致不完整的实现。

对一个应用需求的可测试性的早期评估是定义测试策略的基础。例如：评审需求的可测试性时，测试人员可能会决定使用记录/回放工具，这样就可以自动执行一些测试工作。早作这种决定可以为评估和实现自动测试工具预留充足的时间。

再讲一个例子：在早期评估阶段，可能确定了某些涉及复杂和多变计算的需求更适合使用自制的测试工具（请参见第 37 条）或者专门的脚本进行测试。自制测试工具的开发和其他类似的测试准备活动，也需要在测试工作开始之前为其预留时间。

但是，把测试过程移到更接近一个**迭代**[①]的需求定义阶段的位置，还要承担一些额外的责任，包括根据需求划分测试过程的优先级、分配足够的人手和理解测试策略。由于时间、预算和人员的限制，马上为每条需求设计出所有测试过程，即使不是不可能，也是太奢侈了。理想情况下，作为需求定义工作的一部分，需求测试小组和由行业专家组成的测试小组共同负责创建测试场景示例，这里的测试场景包括场景的结果（预期的输出）。

测试过程的设计必须根据迭代的实现计划来划分优先级。如果有时间限制，测试设计人员应该首先为先实现的需求设计测试过程，然后再为后完成的需求设计测试过程"草案"。

需求常常会以迭代方式通过评审和分析而不断得到精化。新的需求细节和场景在设计和开发阶段才逐渐澄清是非常普遍的事。纯粹主义者认为所有的需求细节应该在需求分析阶段全部搞定。但是，实际情况是截止期限的压力要求开发人员尽可能提早启动开发，预先全部完成需求活动的情况是非常少见的。如果需求在随后的过程中得到精化，那么相应的测试过程也需要细化。不管发生了什么样的改动，这些需求必须保持处于最新状态：我们必须把它们看作"动态"的文档。

---

① **迭代开发过程**中的一个**迭代**包括需求分析活动、设计活动、实现和测试活动。一个迭代开发过程中有许多次迭代。整个项目只有一次迭代的开发方法就是**瀑布模型**。

为了有效地管理这些不断完善的需求和测试过程，我们需要定义一个完善的过程，在这个过程中测试设计人员必须参与需求过程。第 4 条会详细论述把需求的变化传达给所有涉众的重要性。

第 4 条：确保需求变化的传达

当测试过程已经根据需求定义好了以后，在需求发生变更时把这种变化通知到测试组成员是非常重要的。这看起来好像是显而易见的，然而常有这样的情况：需求的变化引发应用程序的实现也随之发生了变化，导致测试过程与更新后的应用程序并不匹配，这种情况出现的频率是令人吃惊的。在很多情况下，负责设计测试过程和执行测试过程的测试人员并没有收到需求变化的通知，这种情况会导致错误的缺陷报告，并且失去了必需的研究机会和宝贵的时间。

导致这种过程崩溃现象的原因有以下几种：

- 变更没有形成文档。某人（可能是产品或者项目经理、客户或者系统分析员）在未经其他涉众同意的情况下通知开发人员功能发生了变化，而开发人员既没有和任何人进行交流、也没有形成文档，就实现了这个变化。这就需要设立一个规程，使开发人员明白需求发生了怎样的变更和需求何时发生了变更。这个问题一般可以通过一个**变更控制委员会**（Change Control Board，CCB）、一个**工程评审委员会**（Engineering Review Board）、或者一些类似的机制解决，这些内容我们会在后面讨论。
- 过期的需求文档。测试人员的疏忽或者糟糕的配置管理可能会导致某个测试人员根据需求文档的老版本来设计测试计划和测试过程。需求的更新需要形成文档、在配置管理控制下纳入**基线**，并且通知所有涉众。
- 软件缺陷。虽然需求文档和测试文档都是正确的，但是开发人员可能错误地实现了某个需求。

在最后一种情况下，我们应该写缺陷报告。但是如果没有遵守需求变更规程，就很难判断出现的现象究竟属于上述哪种情况。问题究竟出在软件、需求、测试过程还是上述所有情况？为了避免无端的猜测，所有需求变化必须要公开地进行评估、形成一致的意见并且传达给所有涉众。这可以通过设置一个**需求变更规程**来实现，这个规程有助于把需求的所有变化传达给所有涉众。

如果一条需求需要更正，那么变更规程必须考虑到发生在设计、编码以及包括测试文档在内的所有相关文档上的连锁效应。为了有效地管理这一规程，任何变化都应该在配置管理系统中纳入基线和形成版本号。

变更规程会概括地记录变更发生的时间、变更的内容、谁做的变更和在哪些地方发生了变更。变更规程可以明确地规定变更请求可以在整个生命周期的任何阶段发起，这些阶段不仅包括需求分析过程中各种类型的评审、走查和检查阶段，而且包括设计、编码、错误跟踪或者测试活动、以及其他任何阶段。

每次需求变更都可以通过**变更请求表**形成文档，变更请求表是一个模板，它罗列了有助于简化变更请求规程的所有信息。变更请求表随即会被传递到**变更控制委员会**。建立 CCB 有助于确保任何需求变更和其他变更请求遵从一个详细而精确的规程。CCB 能够保证变更请求已经形成了正确的文档、变更请求已经过评估并已一致通过、受此变更影响的所有文档（需求和设计文档等）都已修改、并且所有涉众都收到了变更通知。

CCB 通常由各种管理小组的代表组成，例如：产品管理者、需求管理者和 QA 组，还有测试经理和配置管理经理。CCB 会议可以根据需要召开。所有涉众需要通过分析建议变更的优先级、风险和折衷方案来评估变更建议。

此外，还必须对变更建议带来的一些相关影响和主要影响进行分析。例如：一个需求变更可能会影响整套测试文档；需要大量增加测试环境并且测试工作将延期好几个星期。或者某个实现的变化会影响整个自动测试工具套件。在批准一个变更之前，我们必须要确定、讨论和处理这样的影响。

CCB 确定变更请求的合法性、后果、必要性和优先级（例如：变更是应该马上实现，还是应该只在项目的中心文档库中增加一个文档）。CCB 必须保证所有建议的变更、相关的风险评估和决策过程都已经形成文档并且已经传达下去。

让各方成员了解每一条变更建议是非常重要的，这样他们可以对风险分析和减缓变化带来的影响做出贡献。保证这一点最有效的方法是使用**需求管理工具**[①]，需求管理工具不仅可以用来跟踪需求的变化，而且还可以保持从需求到测试过程的可追溯性（关于可追溯性的讨论请参见第 2 条中的可测试性的检查列表）。如果需求发生了变化，那么这种变化应该在需求管理工具中加以体现并进行更新，并且它还会标出受影响的测试元素（还有受影响的其他元素，例如：设计、代码等等），这样各个小组会分别相应地更新他们的工作成果。随后所有涉众都会通过这一需求管理工具得到最新的信息。

使用需求管理工具来管理变更信息使得测试人员能够重新评估变更后的需求的可测试性，以及重新评估需求变化对测试元素（测试计划，测试设计等等）或者测试进度造成的影响。为了适应需求的变化和实现的变化，我们必须重新审视和更新测试过程。以前确定的缺陷必须要重新进行评估，因为需求的变化可能已经使它们不再成为问题。如果脚本、自制测试工具和其他测试机制已经建立，那么它们也可能需要更新。

一个定义良好的规程可以有助于需求变更的传达，它不仅是高效的测试工作的需要，而且还是决定项目实施效率的关键。

① 市场上有许多优秀的需求管理工具，例如 Rational 公司的 RequisitePro、QSS 公司的 DOORS 和 Integrated Chipware 公司的 RTM：Requirement & Traceability Management。

第 5 条：注意在现存系统上进行开发和测试

在许多软件开发项目中，遗留版本的应用程序已经存在，虽然它只有很少或者根本就没有需求文档，但它却是重新设计构架和进行平台升级的基础。在这种情况下，大多数组织会坚持新系统的开发和测试工作只需基于对现存应用程序的不断研究，并不愿意花时间分析应用程序的运作并形成文档。当需要通过现存的应用程序自身来反推必需的需求时，表面上这种做法好像会使发行日期提前，因为这种做法在需求再工程、或者分析现存应用程序并形成文档方面只花了很少精力或者根本没有“浪费”任何精力。

遗憾的是，除了一些最小的项目，把现存的应用程序当作需求基线的策略有许多缺点，并且经常会导致只有很少（如果有的话）的需求形成了文档、不正确的功能和不完全的测试。

虽然应用程序某些方面的功能是可以通过自身发现的，但是因为那些依赖于输入数据的商业逻辑非常容易被忽略，所以许多与领域相关的功能是很不容易反推出来的。因为通过所有可能的数据输入来了解现存系统通常是不切实际的，所以往往会遗漏一些复杂的功能。在某些情况下，我们很难确定特定输入产生特定输出的真正原因，此时软件开发人员会对应用程序的行为进行猜测。更糟糕的是，即使找到了真正的商业逻辑，但是这些发现也经常没有形成文档；相反，它会直接在新应用程序中编码实现，从而使猜测过程周而复始、万世不竭。

除了商业逻辑方面的问题以外，上述做法还可能造成对用户界面中某些域的误解，甚至完全遗漏整个用户界面。

在很多情况下，现存的基线应用程序仍然在使用和开发中，它可能既使用了新构架也沿用了旧技术（例如：桌面和 Web 版本）；或者它仍然在生产或持续的维护之中，通常包括每个新版本发行时都会有的缺陷修正和新功能添加等工作。这会引起“目标迁移”的问题：即使在开发人员和测试人员为新产品反推原来产品功能的时候，作为新产品需求基线的应用程序也在不断地更新和加入新功能。这种做法的结果是当新产品开发完成时，它可能变成现存应用程序不同版本的混合物。

在一个“目标迁移”的环境中进行分析、设计、开发和测试活动的最终结果就是难以准确地估计整个软件开发生命周期的时间、预算和需要的人员。因为没有明确的需求来澄清开发和测试的目标，所以负责新应用程序的团队就不能有效地预测相关的工作量。大多数估计只能根据对应用程序功能的肤浅理解得出，而这些理解可能是完全错误的，

或者当现存应用程序升级时需要马上改变。即使是基于详细描述的需求，完成估计任务也是非常困难的，但是当所谓的“需求”包含在一个遗留的或者目标迁移的应用程序中时，估计任务几乎就是不可能完成的。

表面上，如果假设随时间的流逝而变“旧”的应用程序和新实现的应用程序二者产生的输出是相同的，那么我们可能会觉得在现存应用程序基础上构建新应用程序的好处之一是测试人员可以比较二者的输出。但是，这种做法可能并不安全：如果在一段时间内“旧”应用程序在某些场景下的输出是错误的，但却没有人注意到这种情况，怎么办？如果新应用程序的执行是正确的，但是旧应用程序的输出是错误的，那么测试人员就会将一个非法的缺陷记入文档，并且修正工作就会把现存应用程序中出现的错误加入到新应用程序中。

即使测试人员知道不能依靠“旧”应用程序比较输出的结果，问题也仍然存在。如果测试人员在执行测试过程时发现两个应用程序的输出有差异，那么他们还是不知道到底哪一个结果是正确的。如果需求没有形成文档，那么测试人员怎么能断定哪个输出是正确的呢？本来应该在需求阶段完成的确定预期输出的分析工作，现在就落到了测试人员身上。

虽然在现存的应用程序基础上进行新软件的开发工作可能是困难的，但是还是有办法来解决这些问题。第一个步骤是选择可供使用的应用程序版本。项目团队的成员应该注意在现存应用程序上进行新的开发工作会出现的问题。下面列表概述了一些值得思考的要点。

- 使用确定的应用程序版本。所有涉众必须明白新应用程序为什么必须基于现存软件的某一个固定版本，并且必须认同这一点。开发团队必须选择现存应用程序的一个版本作为新开发工作的基础，并且只把这个版本作为开发工作的起点。

  在应用程序的一个固定版本上开始工作，使得缺陷跟踪更加直接，因为经过挑选的现存应用程序的版本本身就能够确定是否是新应用程序的缺陷，而不用考虑现存应用程序的升级版本或者修正版本。当旧应用程序存在缺陷的时候，确定新应用程序是否正确是非常重要的，因此，利用领域专业知识来验证新应用程序是否真的正确仍然是非常必要的。

- 把现存应用程序文档化。下一个步骤是请一个领域专家或者应用专家把现存应用程序文档化，每个功能至少写一段，其中包括各种测试场景及其预期输出。当然，最理想的做法是彻底分析现存的应用程序，但是实际上这种做法可能会增加很多时间和人员投入，这种投入可能是不切实际的，而且一般也不可能得

到资金上的保证。现实的做法是为每项功能写一个段落，并且只对需要详细文档的复杂的交互才细化需求。

仅仅撰写当前应用程序用户界面的文档通常还不够。如果界面的功能并不能反映应用程序内部隐藏的功能性行为的复杂性，也不能反映这种复杂性是如何与界面进行交互的，那么这样的文档就是不充分的。

- 对现存应用程序的更新也要形成文档。当新应用程序准备升级时，从现存基线应用程序的当前状况以后所做的变更（也就是添加或者修改的需求）都应该形成文档供以后参考。这样有助于稳定可靠地分析现存的功能，并且生成正确的设计和测试文档。如果同时适用于两个产品的话，那么需求、测试过程和其他测试资料就可以同时用于新产品和旧产品。

  如果更新的内容没有形成文档，那么新产品的开发工作将会出现“倒退”：新旧产品之间的不一致会表现得非常随机：有些是正确的，但是有些是错误的；有些问题可以预先知道，但是还有些问题只能在测试中发现，甚至更糟的是在产品中才发现。

- 从此实现有效的开发过程。虽然遗留系统可能已在没有需求、设计或者测试文档，或者某个系统开发过程的情况下开发完成了，但是只要有机会为原系统或者新系统开发一个新功能，那么开发人员就一定要定义一个系统开发过程，并且讨论、遵守、根据需要调整这个过程，这样就可以避免永远陷入恶劣的软件工程实践中。

为了更好地组织、规划、追踪和测试每个功能，我们应该遵从上述步骤。这样正在开发的应用程序的功能集合才是清晰的和可量化的。

第 2 章　编制测试计划

测试纲要成功的基石是有效的测试计划。为了适应开发过程或者提出对过程的改进意见，编制合适的测试计划需要对企业文化和企业软件开发过程有所理解。

考虑到必须为测试纲要的实施预留时间，所以在软件生命周期中，计划的编制应该尽早开始。为了估计所需的资源，以及得到必要的采购、通过雇用人员计划和获得测试工具、测试所需的支撑软件和硬件，及早地了解手头的任务是非常重要的。早编制测试计划，我们就能够进行测试进度和预算的估计和通过工作，并最终将其加入软件开发的整体计划。

为了建立和准备测试环境，同时也为安装将要测试的系统、测试工具、数据库和其他组件预留时间，计划的编制工作必须及早考虑。

任何两项测试任务的工作量都是不同的，只有彻底地了解可能对测试目标造成影响的所有部分，才能编制出有效的测试计划。此外，为了选择最有效和最恰当的测试策略，还必须具有测试经验并对测试规律有所了解，其中包括最好的实践经验、测试过程、技术和工具。

在设计测试策略期间，我们必须考虑风险、资源、时间和预算上的限制。为了估计所需的资源和它们所起的作用（其中包括人员数量、专业类型、角色和职责、进度和预算），还必须对估计技术和估计技术的实现有所了解。

估计测试工作量有好几种方法，其中包括比率法和与以前相似任务的工作量进行比较。如果完全从零开始，那么要完成估计工作是相当不容易的，但是正确的估计有助于组成一支高效的测试队伍，并且使得项目发行时间与测试组的工作进度精确地统一。

第 6 条：了解手头的任务和相关的测试目标

一般来说，测试工作就是用来验证软件是否满足某种特定的标准和最终用户的需求。有效的测试增加了被测的软件在任何环境下正确运行并满足预定的需求的可能性，最终目的是使应用程序的最终用户满意。无论是通过检查、走查、测试还是杰出的软件开发实践，实现这个目标的关键是发现和消除软件中的缺陷。

当下面的条件满足时，我们称一个程序是功能正确的：

- 给定合法的输入，程序会根据软件规格说明书的定义产生正确的输出。
- 给定不合法的输入，程序会正确而优雅地拒绝这种输入（并且适当地显示错误信息）。
- 无论是合法的还是不合法的输入，程序既不挂起也不崩溃。
- 程序能够在预定的时间内一直正确地运行。
- 程序实现了它的功能性和非功能性需求（关于非功能性需求的详细讨论，请参见第 9 章）。

对输入进行各种组合和变化来进行测试，以验证在各种场景下，应用程序已满足了功能性和非功能性需求，这是根本不可能的。

此外，大家都明白测试本身并不能产生出高质量的产品——测试的质量是不可能成为产品质量的。检查、走查和高质量的软件工程过程，对于增强产品的质量都是必须的。

但是，测试可以看作是保证软件质量的最后一关。

由于**测试策略**（请参见第 7 条）有助于用最有效的方式达到测试目标，所以必须定义测试策略。在项目交付的最终期限内，修正已发现的所有缺陷，通常是不可能的，所以我们必须为缺陷划分优先级，因为在发行之前修正所有缺陷，既无必要也不划算。

不同的测试工作有不同的测试目标，不同的测试阶段也有不同的测试目标。例如：程序功能性测试的目标与性能或者配置测试的目标是不同的。测试计划人员（通常是测试经理或者测试负责人）必须清楚：此次测试工作的目标是什么？这个目标基于系统必须满足的标准。

制订测试计划的第一步是了解手头的任务、它的范围和与之关联的测试目标。编制测试计划，必须对实现测试目标过程中起作用的每个细节都有清楚的了解。

那么测试组如何完成这种了解呢？测试经理的第一反应可能是拿来所有的需求文档，并据此制订一个包括测试策略在内的测试计划，然后把需求按照功能进行分解，最后为测试组下达设计和开发测试过程的任务。但是，如果不能对手头的任务进行全面的了解，那么这种方法就不是明智之举，因为测试组必须首先理解测试目标的所有组成部分。对测试目标的了解需要通过下面的途径来完成：

- 理解系统。测试组必须从整个系统的高度来了解正在测试的系统必须满足的功能性和非功能性需求（请参见第 2 条）。测试组必须理解这些需求是有效的。因为在单独和孤立的陈述中，场景和功能性流程通常并不清楚明了，所以从需求规格说明书中孤立地提取“系统会……”的陈述，几乎不能为测试组提供对系统的整体了解。我们应该利用涉及整个系统的讨论会和文档，来帮助我们形成对系统的整体了解。这些文档包括对系统要解决的问题的有关讨论建议，其他一些有助于加深对系统理解的文档可能包括高层次的商业需求陈述、产品管理的案例研究和商业案例。例如：与某些具有较高容错性的商业系统相比，有些不允许有错误的系统（例如：其错误会危及生命的医疗设备）会采用与之不同的方法。
- 及早介入。为了深入了解手头的任务，测试经理、测试负责人和其他必要的测试组成员应该在系统的开始阶段（有关系统的第一个决定做出的时候）介入。这种介入能够增加对客户需求、客户问题、潜在的风险以及最重要的方面——功能的理解。
- 理解企业文化和过程。为了适应开发过程或者提出对过程的改进意见，了解企业文化和企业软件开发过程是必需的。虽然每个小组成员都应该努力改进过程，但是在一些组织中，过程改进主要是 QA 和过程工程组（Process Engineering Group）的职责。

  为了适应一个能够实现预定目标的测试策略的要求，测试经理必须了解支持这种测试策略的过程类型，例如：

  - 是让负责测试工作的测试组和开发组相互独立，还是让测试工程师和开发组一体化？

- 测试方法是否适应“极限编程[①]”的开发方法？
- 测试组是否是保证产品质量的最后一关——无论是否满足测试标准，测试组是否都会绿灯放行？

- 实现的范围。为了确定相应的测试范围，除了理解系统要解决的问题和企业文化以外，测试组还必须了解实现的范围。
- 测试期望。管理层对测试的期望是什么？客户期望的测试类型是什么？例如：是否需要用户验收测试这一阶段？如果需要，那么必须遵从何种方法？如果指定了方法，那么预定的里程碑是什么，可交付使用的含义是什么？预定的测试阶段是什么？此类问题的答案通常应该出现在项目管理计划中。在测试计划中应该给出所有这些问题的答案。
- 吸取教训。从以前的测试工作中学到东西了吗？在确定测试策略和设定实际的测试预期时，这一点是非常重要的。
- 工作量大小。生成当前系统所需的工作量范围是什么？需要雇用多少开发人员？这些信息的用处很多，例如：根据开发人员和测试人员之间的比例，可以得到项目的复杂度和测试工作量的范围。第 12 条建议将讨论对测试工作量进行估计。
- 解决方案的类型。最终是实现了一个最复杂的解决方案，还是实现了只需要较短开发时间的、更划算的解决方案？了解这些信息有助于测试计划的制订者了解需要什么样的测试类型。
- 技术选择。系统的实现会选择什么样的技术，这些技术会引起什么问题？系统采用了什么样的构架？它是一个桌面应用程序、客户端-服务器应用程序还是一个 Web 应用程序？这些信息有助于确定测试策略和选择测试工具。
- 预算。包含测试工作在内，实现当前产品或者系统的预算是多少？给定的预算水平有助于确定可能的测试类型。遗憾的是，预算常常是在没有对测试费用进行估算的情况下而确定的，此时，测试经理只能根据预定的预算来调整测试工作量。

① 关于极限编程请参见第 29 条建议的脚注。

- 时间表。为系统开发和系统测试分配的时间分别有多长？截止日期是什么时候？遗憾的是，截止时间常常是在没有对测试时间进行估算的情况下而确定的，此时，测试经理只能调整测试时间表来适应预定的截止时间。
- 分阶段的解决方案。实现是**分阶段**进行，也就是通过发行许多版本，使系统功能不断得到增强呢，还是直接发行一个大版本？如果发行是分阶段的，那么测试人员就必须了解各个阶段以及需要优先解决的问题，这样测试的开发工作才能匹配当前的阶段并根据当前的迭代进行实施。

完成了对系统的全面了解以后，测试经理就知道了系统的规模和相应的工作量、客户的问题和潜在的风险。通过这些信息，测试经理就能够了解手头的测试任务并且建立测试目标和测试目标相关的测试框架，并最终就测试计划或者测试策略形成文档并进行讨论。

此外，确定预算和时间表时，需要预留一定的时间来满足一些其他需要，例如，获得测试环境所需的硬件和软件，以及评估、购买和实现测试工具。越早确定测试工具方面的需求，那么获得正确的测试工具并且有效运用的可能性就越大。

第 7 条：考虑风险

在制订有效的测试策略之前，必须理解测试计划中的**假定**、**先决条件**和**风险**。这包括不利于按照进度实现和执行测试纲要的任何事件、行为和环境，例如：预算批准得太晚、测试设备延期到货或者应用程序延期交付给测试部门。

为了实现预定目标，**测试策略**一定要包括一些必须由测试组才能完成的、详细和精确的活动。制订测试策略时必须考虑许多因素。例如：如果应用程序的构架由几个层次组成，那么设计测试策略时必须要考虑这一点。

测试策略通常必须综合考虑，以便使费用超支、进度延迟、软件中的关键错误和其他失败的风险降到最低。在设计测试策略期间，我们必须考虑手头的任务所面临的各种限制（请参见第 6 条），其中包括风险、资源、时间限制和预算限制。

测试策略最好通过缩小测试任务的方式来确定，如下所述：

- 理解系统构架。把系统分成几个层次，例如：用户界面或者数据库访问层。对系统构架的理解，有助于测试人员为每个层次或者层次和组件的组合制订测试策略。关于系统构架的深入讨论，请参见第 4 章。
- 确定需要使用 GUI 测试、后端测试还是二者同时使用。一旦理解了系统构架，也就能够确定最好的测试方法——通过图形用户界面（Graphics User Interface，GUI）、还是后端进行测试或者二者同时使用。因为 GUI 经常包含必须要执行和验证的代码，所以大多数测试工作会同时涉及这两个层次。

  在确定测试策略时，测试人员必须牢记：商务层（后端）测试的整体复杂度和需要的专门技术等级，远远高于应用程度的前端部分所需的用户界面测试的复杂度和专门技术等级。这是因为编写访问商务层的测试代码需要更复杂的语言和技术能力，例如：如果商务层是用 C++编写的，那么我们可能需要自信的、有经验的 C++程序员。而另一方面，GUI 工具和 GUI 测试并不需要很多的编程背景，这种测试工作所要求（根据测试人员的类型）的只是普通的编程技巧或者行业专业知识。

  确定采用什么类型的测试策略时，测试人员应该考虑对哪些部分的测试是多余的，并且使测试的有效性降低。例如：假设一个正在开发的应用程序，约 75%左右的功能可以通过 GUI 测试来完成，同时，约 25%的功能需要使用面向商务层的测试方法（以应用程序中风险高的区域为测试目标）。商务层上其余 75%的测试工作可能是不必要的。商务层的测试需要大量有经验的开发资源，并且需要增加费用和花费更多的时间。因为剩下 75%的功能可以通过 GUI 进行检验（事实上这些功能 GUI 已经测试过了），所以对它们进行深入测试就是多余的和不必要的。

  开发出来的 GUI 脚本最好是几乎不间断的。如果做不到这一点，那么就需要完成深入的商务级测试。

  在上述例子中，GUI 测试和商务层测试的比例为 75:25，可能需要两个 GUI 自动测试人员（关于自动测试的讨论，请参见第 8 章）和一个商务层测试人员。除此以外，还可能需要几个领域专家作测试人员，他们会根据应用程序的规模和复杂度完全运用手动的测试技术。在上述例子中，根据应用程序层次的数量或者逻辑上的层次，在商务层完成整个应用程序的测试可能需要 5 个或者更多的有经验的 C++开发人员。因为基本的商务层接口在两个版本之间可能发生变化，也经常会有这样的变化，而这种变化并不影响 GUI，所以测试人员必须紧跟开发组的步伐。

  由于应用程序 75%的功能的实现模式是直接读取记录和更新记录，所以对它们的底层测试工作是不必要的。但是，为了保证 GUI 本身没有数据处理问题或者其他细微错误，GUI 测试仍然必不可少。

  前面的例子证明了考虑风险、复杂度和必要性对确定测试策略的重要性。但这并不是一成不变的，每个项目都需要具体问题具体分析。

- 选择测试设计技术。缩小使用的测试技术类型的范围，有助于减少大量的输入组合和变化。可供使用的测试设计技术有很多，但是究竟使用何种技术，则必须作为测试策略的一部分而定下来。第 5 章中讨论了一些测试设计技术。
- 选择测试工具。当制订测试策略时，如果厂商提供了测试工具，那么测试人员必须根据手头的测试任务，确定将使用什么类型的测试工具；如何使用这些工具；哪些小组成员使用这些工具。也可能需要自主开发一个测试工具，而不是直接购买。第 7 章将讨论自动测试工具，其中包括如何在开发和购买之间做出决定。
- 开发内部自制测试工具或者脚本。通过研究手头的任务，并了解市场上有哪些类型的自动测试工具，测试设计人员可能决定开发一个内部自制的自动测试工具（请参见第 37 条），也可能决定使用普能的测试工具。当厂商提供的自动测

试工具对测试任务毫无帮助时，我们可能需要开发内部的测试脚本和自制测试工具。

- 确定测试需要的人员和专门技术。根据测试策略纲要，我们必须确定需要的测试人员和专门技术。如果需要开发自制测试工具，那么测试组必须包括一名开发人员（也就是有开发技能的测试人员）。如果部分测试策略涉及到了使用记录/回放工具来进行自动操作，那么我们还需要自动测试的技巧。我们还需要具备领域专业知识的人员，他们具备被测试的软件的行业知识。如果测试组不能掌握正确的技能，那么会对测试工作的成功造成极大的威胁。第 3 章讨论有关测试组的内容。

- 确定测试覆盖率。测试人员必须了解要求的测试覆盖率。例如：在某些情况下，合同中会列出所有需要测试的功能需求，或者列出代码覆盖率的要求。而在另外一些情况下，测试人员必须根据给定的资源、时间表、工具、手头的任务和忽略某项测试带来的风险，来确定测试覆盖率。测试人员在开始阶段就应该把测试覆盖的内容和不覆盖的内容写入文档。

- 建立发行标准。确定测试覆盖率，与定义发行标准或者结束标准有着紧密的联系。**发行标准**表示什么时候可以认为测试完成了，因此预先把发行标准文档化是非常重要的。例如：发行标准可能规定若应用程序中包含美观上的缺陷也可以发行，但是对所有紧急的和致命的缺陷，则必须修正以后才能发行。另一种发行标准可能会规定某个优先级高的功能在发行之前必须能够正常工作。

- 设置测试时间表。测试策略必须根据分配给测试工作的时间进行剪裁。为了避免去实施一个无法满足时间进度要求的策略，测试策略包含一个详细的时间表是非常重要的。

- 考虑测试阶段。不同的测试阶段需要运用不同的测试策略。例如：在功能测试阶段，测试工作是为了确定是否满足了功能需求，而在性能测试阶段，测试工作是为了保证是否满足了性能需求。在测试计划中是否安排了系统 α（Alpha）测试和 β（beta）测试？为了针对每个阶段制订足够的测试策略，我们必须了解各个测试阶段。

这些只是制订测试策略时需要考虑的一部分问题。因为无论多长时间都不足以彻底测试一个系统，所以在确定测试策略时了解项目风险是非常重要的。风险场景必须通过规划来评定等级和进行管理。最后，测试组必须确定需求的测试优先级，并且评估每条需求内在的风险。测试组必须评审系统中已经确定的重要功能和高风险的元素，并且在

确定测试需求优先级顺序时考虑这些信息。

当确定测试过程开发顺序时，为了保证需求已经由产品管理小组或者需求小组按照重要程度从高到底确定了优先级，测试人员必须要参加需求评审。在确定功能相对的重要性时，必须要考虑最终用户的意见。测试组必须能够拿到经过优先级评定的需求列表。

除了功能的优先级评定外，把需求按照相关的功能路径、场景或者流程进行分组也是非常有益的。这样就很容易把测试任务分配给不同的测试工程师。

下面列出的标准是 Rational Software Corporation [①]公司的推荐标准，它们用来确定需求分组的原则：

- 风险等级。在完成风险评估以后，为了确保降低影响系统性能的巨大风险或者减小使公司陷入债务危机的潜在威胁，我们为测试需求评定了优先级。高风险的问题可能包括诸如禁止数据输入等功能，也可能包括损坏数据或者导致违规的商业逻辑。

- 操作特性。对于一些频繁使用的功能，或者对用户缺乏了解的部分的测试需求，应该分配较高的优先级。属于技术资源或者内部用户的功能和不常使用的功能，可以分配较低的优先级。

- 用户需求。有些需求对用户接受程度来说是致命的。如果测试方法没有强调对这些需求的验证，那么最后的产品就可能违反合同中的义务，或者给公司造成财务上的损失。评估任何潜在的、对最终用户产生影响的问题是非常重要的。

- 可用的资源。排列测试需求的优先级时，要考虑的一个因素是资源的可用性。如前所述，测试纲要必须在各种限制下形成，这些限制包括有限的人员可用性、有限的硬件可用性和与项目需求之间的冲突。这里需要痛苦的权衡折衷过程。

大多数风险是由下面几个因素造成的：

- 短时间面市。对于软件产品来说，短时间面市使得工程资源的可用性变得更加重要。如前所述，测试预算和时间表经常在项目的开始（开发建议阶段）就确定了，当时并没有参考来自测试人员的、根据以往的经验或者利用其他有效的估计技术得出的结果。一名优秀的测试经理会马上做出判断：短时间面市不允

① Rational Unified Process 5.0（Rational Software Corporation，Sept. 7, 2002，从地址 *http://www.rational.com product/rup/index.jsp* 获得）。

许进行充分测试。测试策略必须根据可用的时间进行调整。尽快指出这样的问题是非常重要的，只有这样才能调整时间表、确定快速开发的风险并制订降低风险的策略。

- 新的设计过程。引入新的设计过程会增加风险，新的设计过程包括使用新的设计工具和设计技术。
- 新技术。如果采用了新技术，那么新技术能否像我们期望的那样运转、会不会被错误地理解和实现、或者是否需要补丁，这些方面都可能存在很大的风险。
- 复杂度。我们应该进行一些分析工作来确定哪个功能最复杂、哪个功能最容易出错、错误会对系统的哪些地方造成巨大的影响。测试组的资源应该集中在这些区域。
- 使用频率。应用程序中最常使用的功能（应用程序的“核心”）其潜在的失败会造成很大的风险。
- 不可测试的需求。不可测试的功能性和非功能性需求会对系统的成功构成巨大的威胁。但是，如果测试组在需求阶段（请参见第 1 章）已经验证了需求的可测试性，那么此类问题会减到最少。

尽管采取了所有的预防措施，但是风险还是可能发生。因此当确定了风险以后，我们必须要评估它们带来的影响，并且通过测试策略来减轻和消除它们的影响。为了针对某个特殊的应用程序制订有效的测试和降低风险的策略，测试人员必须仔细地检查风险。

风险评估必须要考虑到风险成为现实的可能性和导致问题发生的使用模式，还有降低风险的策略。我们应该评估潜在问题的严重程度和影响。风险降低策略应该针对那些风险最有可能发生的系统需求制订，同时还要权衡使用的频率和需求的重要性。通常情况下，一个应用程序最重要的部分即使允许也只能包含少量缺陷，而很少使用和不重要的部分则可以有更大的自由度。

非常详细地评估风险是非常困难的。这个过程需要投入大量的人力，例如：客户代表（产品经理）、最终用户和需求组、开发组、测试组和 QA 组。

我们应该讨论风险的等级，这样做出的决定才可能是正确的。如果风险太大，我们可能会建议暂停项目、修改或者完全中止项目。

风险分析提供的信息可以帮助测试经理或者测试组负责人做出困难的决定，例如：根据技能的高低、需要的工作量、以及风险和质量目标来分配测试人员。如果在执行风险评估工作中发现某项任务的风险很高，并且如果这项任务失败会有很大的负面作用，那么此时可以做出决定，指定有经验的测试人员负责这个高风险的任务。

一个有助于降低风险的测试策略，就是把测试工作的重点放在系统中可能会引起绝大多数问题的那些部分。测试工程师或者测试经理必须确定一个实现周期中风险最大的部分和最可能出问题或者失灵的功能。

另一方面，对于一个版本中风险低和影响小的功能，我们只需执行为了这个特定的版本发行所必须的测试工作。风险低和影响小的任务还能为缺乏经验的新手提供积累经验的机会，并且这种做法的风险很低。

仔细地检查目标和风险对制定一组恰当的测试策略是必要的，而它反过来又会产生更可预测的、更高质量的测试结果。

第 8 条：根据功能优先级安排测试工作

软件功能的实现必须划分优先级，分期分批地在功能不断增强的软件版本中实现。我们应该根据客户的需要或者必要性首先交付一些高风险功能。但是，测试过程的规划和开发不仅应该依赖于第 7 条中论述的优先级和风险，还要根据软件功能的实现时间表，因为后者决定了可供测试的功能交付的顺序。

及早确定软件开发的时间表（包括功能的实现顺序）并且通知测试组是非常重要的，这样测试组才能制订相应的测试计划。我们不必为那些现在还不可用的、下个版本才实现的功能设计测试方案，这样可以避免浪费时间，当项目时间有限时尤其应该这样做。我们应当避免频繁地修改实现功能的时间表，因为每次修改都需要修改开发计划和测试计划以及开发时间表和测试时间表。

评定功能的优先级对于分阶段发行来说尤其重要。在大多数情况下，开发工作的安排应该努力首先交付最需要的功能。相应地，应该首先测试这些功能。

功能列表可以根据各种不同的标准划分优先级：

- 风险最高到风险最低。我们在第 7 条中已经描述过，当确定一个项目的开发时间表和相关的测试策略时，考虑风险等级的高低是非常重要的。将开发和测试的精力首先集中在风险最高的特性上，这是划分优先级的一种方法。
- 复杂度最高到复杂度最低。根据复杂度划分优先级，首先开发和测试最复杂的功能，这种做法可以最大限度地减少进度的延迟程度。
- 客户的需要。对于大多数项目，一般都是根据客户的需要来确定功能交付的优先级，这通常是为了推动产品的市场和销售活动。
- 预算的限制。绝大多数软件项目在实施中都超出了预算。当为一个指定版本划分功能的优先级时，顾及分配给测试工作的预算是非常重要的。有些功能对程序的成功比其他功能更重要。
- 时间限制。当为一个指定版本的功能划分优先级时，考虑时间限制是非常重要的。
- 人员限制。当划分功能的优先级时，规划者必须考虑人力的可用性。由于预算或者其他问题，开发团队中可能还缺少实现特殊功能所需的关键人员。在划分功能优先级时，我们需要考虑的不只是“做什么”，而且还要考虑“谁来做”，认识到这一点是非常重要的。

为了制订一个有效的功能时间表，我们通常需要综合使用上述方法。为了得到一项功能的总体价值或者“分量”，我们可以为它的风险、复杂度、用户需要和其他类似的因素进行“打分”。一旦完成了为每个功能的打分工作，那么把功能列表按照得分进行排序，就得到了功能的优先级列表。

第 9 条：牢记软件方面的问题

当制订测试计划的时候，测试组应该了解影响项目开发和交付的一些软件问题，这是非常重要的。其中包括：

- β（Beta）或者预发行产品。开发组可能正在某一特殊技术或者操作系统的 β 版本上实现新功能。当开发组使用某些技术的 β 版本时，那么可能会遇到这样的情况：自动测试工具不能支持全部的 β 技术。测试计划必须要考虑这种因素，并且为了确定当前的测试工具是否满足需要，我们应该在生命周期中尽早完成对开发组用到的所有技术的评估。

- 新技术和不完善的技术。使用新技术不可避免地会给开发工作造成混乱，潜在的连锁反应甚至贯穿软件开发生命周期中的多个阶段，其中也包括测试活动。有时由于一项新技术引起的问题太多，甚至可能需要对产品中的一部分进行再工程。例如：假设开发组使用操作系统的 β 版本实现了一个解决方案，但是在正式发行的版本中它的构架发生了变化，此时要求开发组重写部分实现。相应的，测试组也必须再执行一次详细的回归测试，重新修改灰箱[①]测试的工作成果并且被迫重新开发已经完成的自动测试脚本。测试计划制定中必须考虑诸如此类的问题。

- 分阶段实现。在系统的第一个版本完成之前，功能是一个个交付使用的。测试工作必须与交付使用的功能相协调。例如：应用程序可能为输入一组数据提供了一个用户界面，但是查看这些数据的界面可能很晚才能使用。测试计划必须要适应这样的情况，具体的做法是提供一种可替代的测试方法，来测试应用程序是否正确地处理并存储了数据。

- 缺陷。缺陷在很多方面会阻碍测试工作的进行。测试过程可能无法完全执行：系统可能在完成所有的测试步骤之前已经停止了运转。在软件构建的早期尤其如此，原因是此时缺陷太多。为了彻底解决这些问题，我们必须通知开发组：某些缺陷使测试工作无法继续进行，这样这些缺陷的修正工作会被赋予更高的优先级。

① 灰箱测试是通过用户界面来检验系统或者直接考验底层的组件，同时监视内部组件的行为来确定一项测试成败的测试方法，请参见第 16 条建议。

- 补丁和服务包。操作系统提供商和其他第三方软件供应商，通常会提供产品的升级服务来修正缺陷和推出新功能。如果正在测试的应用程序受这种升级的影响，那么把这些情况写入测试方案是非常重要的。例如：当一个流行的 Web 浏览器的新版本发行时，许多用户会升级到这个新版本。测试计划必须为尽快获得新浏览器作好准备，并且在这个浏览器的新版本和旧版本下测试应用程序的兼容性。类似地，当一个操作系统的服务包发行后，许多用户会安装这个服务包。测试人员必须尽早地测试操作系统升级之后应用程序的兼容性，而不能依赖于服务包应该承诺的向后兼容。

第 10 条：获得有效的测试数据

在建立详细的测试设计期间（在第 5 章中讨论），测试用例中会加入对测试数据的需求，这样测试用例就成为测试过程的一部分。[1]有效的测试策略要求细心收集和准备测试数据；如果测试数据很糟糕，那么就会损害功能测试。反过来，好的测试数据有助于提高功能测试的质量。优秀的测试数据还能够帮助理解测试工作和提升可测试性。正确地选择测试数据的内容还可以减少维护的工作量。如果需求不明确，那么准备高质量的测试数据有助于关注商业逻辑。

在设计数据样例时，好的**数据字典**和详细的**设计文档**非常有用。数据字典除了提供数据元素的名字以外，还描述了数据结构、元组个数、使用的规则和其他有用的信息。设计文档——特别是数据库模式，也有助于确定应用程序与数据的交互方式以及数据元素之间的关系。

考虑一下各种可能性，我们通常不可能针对所有可能的情况来对输入和输出的各种组合和变化都加以测试，来确认应用程序的功能性和非功能性需求满足所有条件，但是可以通过各种各样的测试设计技术，来减少数据输入和输出的组合和变化。**数据流覆盖**就是这样一种测试技术。它是为把数据流并入经过选择的测试过程的步骤而设计的，它有助于在所有适用的测试路径中确定满足某些数据流特点的测试路径。

边界条件测试 是另一种测试技术。我们会为每个**边界条件**（数据的数量或者允许的内容有限制，它们是在软件设计中设置的）准备测试数据。系统的行为经常在它们的边界上发生变化。错误集中在边界上发生，因此，在边界上测试系统的行为是一种非常有效的测试技术。

边界条件应该在需求陈述中列出，例如："系统支持从选择列表控件中选择 1 项到 100 项这样一个范围"。在这个例子中，边界测试可以是 100+1（边界之外）、100-1（边界之内）、100（在边界上）；还有 1（在边界上）、空（null 没有输入）和 0（边界之外）。当系统需求中没有指定边界条件时，必须将实际发现的边界测试加入到测试计划中。这类测试是非常普遍的，因为在通过集中测试确定边界之前，开发人员经常也不知道边界是什么。

① Per ANSI/IEEE Standard 829-1987，测试设计确定测试用例，测试用例又包含测试数据。

理想情况下，确定需求的可测试性时，这些问题就应该已经解决。但是当今技术的复杂度如此之高，在需求阶段就确定系统的边界通常是不可能的。确定边界的一个替代方法是开发和测试软件的原型。

测试数据的设计必须使得每个系统级的需求都能经过测试和验证。测试数据的需求评审应该关注数据的几个关键方面，其中包含：[1]

- 深度。测试组必须考虑支持测试工作所需的数据库记录的数量和规模。测试组必须确定数据库中或者某一数据表中只存储了 10 条记录就已经足够了，还是必须有 10 000 条记录才行。在生命周期早期的测试（例如单元测试和集成测试）应该使用小规模和手工建立的数据库，这样可以最大程度地控制测试工作并且测试的针对性最强。当测试工作的进程已经完成了不同阶段的测试和不同类型的测试时，为了适应某些特殊的测试，我们必须适当增加数据库的规模。例如：如果生产环境中的数据库可能容纳 1 000 000 条记录，而测试执行时数据库只包含 100 条记录，那么性能测试和容量测试就毫无意义。有关性能测试和在开发生命周期早期完成性能测试的重要性的进一步讨论，请参见第 9 章。

- 宽度。测试工程师必须研究数据数值的变化（例如：10 000 个不同的账目包含不同的数值，或者许多不同类型的账目包含不同的数据类型）。一个精心设计的测试会考虑测试数据的变化，反之，所有测试数据都相似的测试，其输出结果是有限的。

  对于包含不同数值的数据记录，举例来说：测试人员需要考虑的可能是：有些账目可能包含负的余额或者小额账目（例如 100 美元）、中等额度（例如 1 000 美元）、大额账目（例如 100 000 美元）和超大额度账目（例如 10 000 000 美元）。而对于包含不同数据类型的记录，举例来说：测试人员必须考虑的可能是：把银行客户的账目分成几类，其中包括存款、支票、贷款、学生、联合和商业账目。

- 范围。测试数据的范围与数据的精确度、相关程度和完整程度是有关的。例如：通过查询操作来确定银行中各种不定额总数大于 100 美元的账户，要对这一查询进行测试时，测试数据中不仅应该包含大量符合上述标准的账户，而且测试也必须反映其他的数据，例如：理由码、相关历史和账目所有者的统计数据。

① 改编自 Ivar Jacobson，Grady Booth 和 James Rumbaugh 的 *SQA Suite Process*，*Rational Unified Process 5.0*，*The Unified Software Development Process*（Reading，Mass.：Addison-Wesley，Feb. 1999）

使用完整的测试数据，可以使测试过程全面地验证和检查系统，并且支持对结果的评估。测试工程师还必须验证，对于特定目的的查询（例如：不定额总额大于 100 美元的账户），查询结果中的记录的内容是合法的，并且还要验证查询结果既没有遗漏数值，也没有包括错误的数值。

- 测试执行期间的数据完整性。当执行测试时测试组必须保持数据的完整性。测试组在测试过程中，必须能分离数据、修改所选的数据并且能使数据库恢复到它的初始状态。测试组必须保证当几个测试工程师同时执行测试时，一项测试不能干扰其他测试需要的数据。例如：如果一个测试组成员正在修改数据数值时，另一个测试组成员正在执行一次查询，那么查询的结果可能与预期结果不符。如果切实可行并且经济允许的话，那么为防止一个测试人员的工作影响另一个测试人员的工作，可以采用为每个测试人员分配一个独立的测试数据库或者数据文件的办法。另一种办法是为每个测试人员分派不同的测试任务，使每个人集中测试一个特定的功能领域，不和其他人测试的功能领域发生交叉。

- 条件。创建的数据集应该能够反映应用程序所在领域的特定“条件”，这就是说特定模式的数据并不需要等到一定的时间之后才能通过执行特定的操作来获得。例如：财务系统一般都在年底出清存货。按照年终的条件存储数据就能够测试系统在年终出清存货时的状态，而不必先输入全年的数据。创建已经处于出清存货状态的测试数据可以简化测试工作，因为测试人员可以简单地装载出清存货的测试数据，而不用执行许多操作来使数据进入出清存货状态。

当确认测试数据需求时，制作一张表格，一列是测试过程、另一列是测试数据需求。这一作法很有好处。在这些需求中，标明所需数据集的规模和产生测试数据所需的时间是非常重要的。虽然一个小规模的数据子集已经足够用来进行功能测试，但是性能测试却要求一个生产规模的数据库。获得生产规模的数据可能需要很长的时间，有时需要长达几个月。

测试组也必须设计获得、生成或者开发测试数据的方法。为使得包括回归测试在内的所有测试活动能够顺利进行，把测试数据库恢复到最初状态这一机制也必须在项目的测试计划中进行设计并文档化。测试人员必须确定要使用的测试数据库和资料库的名字和位置，这些测试数据库和资料库是检查和测试应用软件所必需的。

测试数据通常必须在测试之前准备好。**数据准备**应该包括对原始数据文本或者文件的处理、一致性检查和深入分析数据元素等过程，其中后者又包括定义数据到测试用例的映射标准、澄清数据元素定义、确定主键和定义可接受的数据参数。为了准备数据，也为了开发出环境的安装脚本和试验台脚本，测试组必须获得并修改所需的全部测试数

据库。最理想的情况是使用客户现成的数据，因为它们是实际环境中出现的数据场景的真实组合和变化。使用真实的客户数据的一个好处是它们可能包含测试设计人员没有考虑到的数据组合或者使用模式。通过真实的客户数据来测试应用程序，对应用程序来说是一种实用的真实环境下的检查。

第 11 条：规划测试环境

测试环境由支持测试工作的所有物质元素组成，例如：测试数据、硬件、软件、网络和设备。测试环境计划必须确定访问测试环境的人员的数量和类型，并且必须为这些人（关于测试组成员的讨论请参见第 3 章）分配足够数量的计算机。我们还应考虑所需环境的安装脚本和试验台脚本的数量和种类。

在本章中，术语**生产环境**是指最终软件的运行环境。它可能小到一台单独的最终用户的计算机，大到一个连接到 Internet 并且作为一个完整的 Web 站点的计算机网络。

虽然单元测试和集成测试通常由开发人员在开发环境中完成，但是，系统测试和用户验收测试最好在独立的测试实验室完成，测试实验室的设置和生产环境相同，或者至少是生产环境的简装版本。为了发现所有可能影响应用程序的、与配置有关的问题，例如：软件不兼容、群集问题和防火墙问题，测试环境必须能够反映生产环境中的基线配置，因此测试环境必须能代表生产环境。但是由于费用和资源的限制，完全复制生产环境常常是不现实的。

在完成上述问题的收集和文档化工作之后，测试组必须结合下面的信息和资源的准备状况来设计测试环境：

- 获得客户环境样本的描述，包括支撑软件、商用现货（Commercial off-the-shelf，COTS）工具、计算机硬件和操作系统等一系列列表清单。硬件的描述应该包含下面的要素：显示器分辨率、硬盘空间、处理器速度、内存的特性以及打印机特性，打印机特性包括打印机类型、处理能力、以及打印机是专供用户的机器使用还是连接在网络服务器上。
- 确定测试环境是否需要一个归档机制来存储测试后产生的大文件（特别是客户端-服务器系统中的日志文件），例如：磁带机或者可刻录 CD（CD-R）驱动器。在当今的测试环境中，归档几乎是必不可少的。
- 确定客户环境中的网络特性，例如：使用的线路是租用线路、调制解调器还是 Internet 连接，使用的协议是以太网、IPX 还是 TCP/IP。
- 对于客户端-服务器或者基于 Web 的系统，我们需要确定服务器的操作系统、数据库和其他组件。

- 确定测试组需要的自动测试工具的许可证数量。
- 确定执行某些测试过程需要的其他软件，例如：字处理软件、电子制表软件和报告写作器。
- 如果可行，最好在确定硬件测试环境时考虑对测试数据的需求，其中包括测试数据库的规模。保证机器有足够的处理能力并保证安装数据所需的资源（例如：磁带机或者网络连接）是非常重要的。
- 考虑配置测试需要的特殊资源，例如：活动硬盘和图像库。

遵照这些准备活动，测试组就完成了测试环境的设计工作，这个设计由测试环境的构架表示图和支持这一构架所需的组件列表组成。我们必须检查这一组件列表，确定哪些组件已经到位；哪些可以在组织内部调配；哪些必须外购。必须外购的组件列表组成了**测试设备采购清单**。清单中会列出需要的数量、单价信息以及维护和支持的费用。为了确保测试工作在硬件失灵的情况下也能够继续进行，测试组也可以在清单中加入少量的备用组件。

第 12 条：估计测试准备和执行所需的时间[①]

在实施理想的测试计划和最好的测试策略之前，我们必须估计测试的准备和执行所需的时间。当制订整个软件开发的时间表时，记住包含对测试时间的估计是非常重要的。

从历史上来看，软件开发的估计工作，主要关注的是开发工作量和整个项目的工作量。软件质量保证工作（例如：软件测试）所需的时间，往往是通过与预期的开发工作量或者整个项目工作量进行比较，然后粗略地估算出来。但是，因为测试草案的可变性，仅用这种方法估计测试时间通常是不够的，还必须考虑到影响特定测试工作的各种因素。

一个特定项目需要的测试工作量依赖于许多变量。它们包括：组织文化或者组织的“测试成熟度”、正在测试的应用软件的复杂度、为项目制订的测试需求的范围、执行测试的个体的技能水平以及承担测试工作的测试组织的类型。为了对测试组资源进行估计，可以把这些变量作为自动估计工具支持的复杂模型的输入。

但是，因为估计公式往往不够准确，[②]所以即使给出对测试工作量有影响的大量变量，使用复杂的方程一般也没有什么用处。用这些方法估计的测试工作量，其结果一般并不能真正反映实际付出的工作量。另外，这些方法也过于麻烦。但是，简单的估计模型反而比较有效。

理想情况下，测试估计工作应该从**工作分解结构**（work breakdown structure，WBS）开始，或者也可以从测试任务的详细列表开始，“测试工作分解结构”是根据测试任务的详细列表而生成的。

为了获得最好的效果，这里列出的方法必须根据组织的要求和规程进行调整。

1. 开发比例法

由于软件工业关注的焦点是估计软件开发的工作量，所以测试工作量的估计一般是基于开发比例方法得到。这是一种快速简单地测量测试工作所需的工作量等级的方法，

① Elfriede Dustin 等人，*Automated Software Testing*（Reading，Mass.：Addison-Wesley，1999），第 5-节（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

② 关于这个话题的更多内容，请参见 J.P. Lewis，*Large Limits to Software Estimation*，ACM Softwa Engneering Notes 26:4（July, 2001）:54-59。

其核心是计划投入的软件开发的人员数量。测试组的规模可以根据项目中开发人员和测试工程师的比例计算得出。在这里术语**开发人员**包括设计人员、开发人员和单元级测试人员。

如表 12.1 所示，根据这一比例得到的结果依赖于软件开发工作的类型。当测试工作的范围包含集成测试和系统测试级别上的功能测试和性能测试时，表 12.1 中的比例是恰当的。注意测试组成员与开发人员之间的比例依赖于正在开发的软件的类型和复杂度。例如：当一个商业软件提供商开发的软件面向的是全球用户时，测试的重要性就增加了，所以测试工程师与开发人员之间的比例也要上升。但是，对于市场较小的软件产品，尽管其他因素相同，测试工程师与开发人员之间的比例就会低一些。

另外，在开发生命周期的不同阶段，测试人员和开发人员之间的比例也会发生变化。在测试阶段的后期，由于时间紧张并且截止期限临近，开发人员、临时人员和产品管理人员都可能被分配去协助完成测试工作。此时，测试人员和开发人员之间的比例就会升高，测试人员的数量甚至可能会超过开发人员的数量。

测试计划中制定的这一比例，只能看成是近似的或者理想的。实际执行时，这一比例会发生变化，这决定于可供支配的预算、有关测试工作的采购决定、应用程序的复杂度、测试工作的效率和许多其他因素。

表 12.1 使用开发比例法计算测试组的规模[①]

产品类型	开发人员的数量	开发人员和测试人员之间的比例	测试人员的数量
商业产品（大市场）	30	3:2	20
商业产品（小市场）	30	3:1	10
为单个客户的开发和大量的 COTS 集成	30	4:1	7
政府（内部）应用程序开发	30	5:1	6
企业（内部）应用程序开发	30	4:1	7

① 根据系统复杂度或者构建的系统对错误的容忍度（例如：医疗设备系统或者飞机控制系统是不能容忍错误的），比例会有所不同。

2. 项目人员比例法

另一种快速估计完成测试工作所需的人员数量的方法是项目人员比例法，表 12.2 给出了其详细内容。这种方法根据历史上的经验，利用整个项目计划投入的人数（包括需求、配置、过程、开发和 QA 等各类人员）来计算测试组的规模。当实施开发工作的人数频繁变化或者难于确定数量时，这种方法特别有价值。表 12.2 计算测试组规模时，假设了测试工作的范围包括集成和系统测试级别上的需求评审、配置测试、包含的过程以及功能和非功能测试。为了简化在"测试人员数量"列中相应数值的计算，我们选择"项目人员数量"列中的基数为 50。当然，一个项目的人数可能会比 50 多或者少。

表 12.2 使用项目人员比例法计算测试组的规模

开发类型	项目人员数量	测试组大小因子	测试人员数量
商业产品（大市场）	50	27%	13
商业产品（小市场）	50	16%	8
为单个客户的应用程序开发	50	10%	5
为单个客户的开发和大量的 COTS 集成	50	14%	7
政府（内部）应用程序开发	50	11%	5
企业（内部）应用程序开发	50	14%	7

3. 测试过程法

另一种估计测试工作所需人数的方法是采用测试工作的规模估计，其具体形式指是计划在项目中实施的测试过程的数量。由于这种方法只考虑需要设计和执行的测试过程的数量，而忽视了对测量测试工作量有影响的其他重要因素，所以这种方法在某种程度上是有局限性的。为了使估计更加全面准确，这种方法只能和此处列出的其他方法合使用。

为了采用测试过程法，首先，组织必须获得已经完成的各种开发项目的历史记录，这些记录中需要包含有关开发规模的度量值，例如：功能点、使用的测试过程数量、"人·小时"为单位的测试工作量的测量结果。开发规模的度量值可以按照代码行（LOC）、等价的代码行数、功能点或者产生对象的个数来进行统计。

然后，测试组确定需求以及历史上开发规模的度量值和那些项目中使用的测试过程的数量二者之间的关系，并且计算出新项目中需要的测试过程数量的估计值。接着，测试组根据历史上相似项目的经验来确定历史上测试过程的数量和所需的测试人·小时数之间的关系。最后，根据这个结果来估计新项目的测试工作所需的人·小时（或者等价的全职人员）的数量。

为了使这种估计方法获得最大的成功，用来比较的项目必须在性质、技术、需要的专业技术、解决的问题以及其他因素方面是相似的，这一点会在后面的"其他考虑"部分详细地加以描述。

表 12.3 显示了利用测试过程法得到的示例数据，图表中假设测试组估计新项目需要 1 120 个测试过程。测试组回顾了历史上两个或多个与目前项目相似的项目的测试工作量记录，它们的测试工作平均需要 860 个测试过程和 5 300 个人·小时。在从前的这些测试工作中，在整个测试活动生命周期中（从启动和规划到设计和开发再到测试和生成报告的全过程），每个测试过程需要的人·小时数大约为 6.16。假设 5 300 小时的工作占用的时间跨度大约为 9 个月（1 560 小时），那么项目需要雇用 3.4 个全职的测试工程师。测试组计划为这个新项目开发 1 120 个测试过程。

表 12.3 使用测试过程法计算测试组的规模

	测试过程的数量	因子	人·小时的个数	完成的时间跨度	测试人员的数量
历史记录（两个或者两个以上相似项目的平均值）	860	6.16	5 300	9 个月 (1 560 小时/人)	3.4
新项目的估计	1 120	6.16	6 900	12 个月 (2 080 小时/人)	3.3

当历史数据来自组织的测试文化成熟后承担的项目时，使用测试过程法得到的因子才最可靠。

重视一个项目需要的测试过程数量和测试需求范围二者之间的关系也很重要。成功地运用这个方法要求预先确定需求和项目范围。遗憾的是，我们经常在需求定稿之前就要求有关测试的估计数据，事实上也确实需要这样做。

4. 任务规划法

另一种估计测试工作所需工作量的方法需要回顾相似类型的测试工作在人·小时数方面的历史记录。这种方法和"测试过程法"的不同之处在于它关注的焦点是测试任务。

两种方法都要求高度结构化的环境，这样就可以追踪和测量这个环境中的各种细节，例如：下面要介绍的“其他考虑”部分中描述的各种因子。历史上的测试工作应该包括对工作分解结构中每项任务耗费时间的记录，这样历史记录就记录了完成各种任务所需的工作量。

在表 12.4 中，一个估计需要 1 120 个测试过程的新项目与历史上的基数进行比较。历史记录表明平均需要 860 个测试过程的项目需要 5 300 个人·小时，其因子为 6.16。假设用这个因子来估计完成 1 120 个测试过程所需的人·小时数（这里的历史数据和表 12.3 中测试过程法使用的历史数据相同）。

表 12.4 使用任务规划法计算的测试的人·小时数

	测试过程的数量	因子	测试所需的人·小时数
历史记录（相似项目）	860	6.16	5 300
新项目的估计	1 120	6.16	6 900

接下来测试组从历史记录中得到在测试阶段中各项测试任务需要的时间。表 12.5 描述了每个阶段所需的时间（单位为小时）。

表 12.5 任务规划法中每个测试阶段需要的时间（单位为小时）

阶段	历史数据	占项目的百分比	初步估计	调整后的估计
1 项目启动	140	2.6	179	179
2 早期项目支持（需求分析等等）	120	2.2	152	152
3 关于自动测试决定	90	1.7	117	-
4 测试工具的选择和评估	160	3	207	-
5 引入测试工具	260	5	345	345
6 制订测试计划	530	10	690	690
7 测试设计	540	10	690	690
8 测试开发	1 980	37	2 553	2 553
9 测试执行	870	17	1 173	1 173
10 测试管理和支持	470	9	621	621
11 测试过程改进	140	2.5	173	-
项目总计	**5 300**	**100%**	**6 900**	**6 403**

为了得到这些数据，测试组首先要用历史上的百分比因子初步估算出每个阶段所需

时间。（如果没有历史数据，那么测试人员就只能凭过去的经验来估计这些值）。右边的一列是针对当前具体情况修正后的数据。在这个例子中，要求新项目使用一个特定的自动测试工具，所以第 3 项和第 4 项就是多余的了。同时测试组接到通知没有资金来保证测试过程的改进活动，因此修正后的估计时间一列中将第 11 项也删除了。

下一个步骤是根据调整后的人·小时估计（6 403 小时）来计算测试组的规模，如表 12.6 所示。测试组规模的计算结果是在为期 12 个月的项目中需要 3.1 个测试工程师。倘若在整个测试工作期间给测试组正好配备了 3 名全职的测试人员，那么要在给定的时间表内完成测试工作，如果可能的话测试组的工作效率就需要比以前的测试组稍高一些。更常见的情况是，测试组最初必须只关注高风险的方面，然后再考虑低风险的内容。因此测试策略必须进行相应的调整，并且必须通报调整的理由（例如：人手不足）。

测试组也可以使用不同的用人方法，例如用两个全职的测试人员再加上两个部分参与的测试工程师。部分参与的时间分配也有许多方法，例如两个兼职人员都投入 80%的精力。

表 12.6 从人·小时估计得到的测试组规模

	测试过程数量	人·小时估计	调整后的估计	完成的时间跨度	测试人员数量
新项目的估计	1 120	5.71	6 403	12 个月（2 080 人·小时）	3.1

5. 其他考虑

无论选用哪种估计方法，有经验的专业测试人员应该考虑到任何特殊情况，在这些情况下就不能完全根据以往的经验来估计测试工作量。例如：如果一个组织以前根本不重视测试工作，因此也没有形成成熟的测试过程，那么在这个组织中使用开发比例法时就需要相应地调整比例因子。相似的，当使用任务规划法时，如果测试组最近经历了重大的人员调整，那么也需要谨慎地调整得到的因子。当估计测试工作量时，需要考虑如下一些问题：

- 组织。组织的测试文化和测试成熟度。
- 测试需求范围。必须完成的测试工作可能包含功能测试、服务器性能测试、用户界面测试、程序模块性能测试、程序模块复杂度分析、程序代码覆盖率测试、系统负载性能测试、边界测试、安全性测试、内存泄漏测试、响应时间性能测试和可使用性测试，以及其他测试。

- 测试工程师的技能水平。执行测试的个人的技术能力和水平。
- 使用测试工具的熟练程度。自动测试的使用使项目测试人员面临从未经历过的新的复杂度水平。学习工具需要有个过程，而编写测试脚本需要有编程方面的专业技术。这对测试组来说可能是新东西，而且测试组可能也缺乏编写代码的经验。即使测试组具有某种自动测试工具的使用经验，但是新项目可能使用不同的工具。
- 商业知识。测试组成员对应用程序商业领域的熟悉程度，也称为“领域知识”。
- 测试组的组织结构。因为有些测试组的组织结构要优于其他形式的结构，所以测试组的组织类型可能也是一个因素。
- 测试程序的范围。一个有效的自动测试程序本身就需要一定的开发工作量，其中包括规划策略和目标、测试需求定义、分析、设计和编码。
- 测试工作的启动。测试计划和测试活动应该在项目早期开始。这意味着为了防止分析和设计错误，测试工程师必须参与分析和设计的评审活动。测试人员及早介入，就可以更透彻地了解需求和设计，因此能够构建更合理的测试环境和生成更全面的测试设计。测试人员及早介入不仅有利于有效的测试设计（当使用自动测试工具时尤其如此），而且还有利于及早发现错误和防止错误从需求规格说明阶段进入设计阶段、从设计阶段再进入编码阶段。
- 软件计划升级的版本个数。许多业界软件专业人士都有这样的经验：使用自动软件测试工具可以显著地减少测试工作的人·小时数，或者降低测试计划和执行的复杂度。使用自动测试工具可以节约资金，使得时间变得充裕。但是事实上，测试组最初使用某种特定的自动测试工具时，这种节约几乎体现不出来，而在应用软件的后续版本中才会获得好的效果。
- 过程定义。使用已编档的过程会提高测试工程的实施效率。而缺乏已定义好的过程，其结果会完全相反，这会使初级测试工程师的学习曲线变长。
- 高风险的应用程序。如果软件失灵会对人的生命产生威胁，那么这种软件的测试工作所要求的深度和广度要远远大于那些不会产生很大风险的应用软件。

测试开发和执行的时间不足会减低测试工程实施过程的效率，并且还需要额外的测试工作去纠正错误或疏忽。

当一个应用软件项目所需的测试工作量决定于很多变量时，有些简单、通用的测试

工作量估计方法非常有效，因为这些简单的估计方法反映了所有这些因素对测试工作量所产生的影响效果的标准分布。

根据开发组的规模来确定合适的测试组规模的开发比例法可能是最通用的测试工作量估计方法。根据项目的总规模确定测试组规模的项目人员比例法通常是确定测试组规模最容易的方法。测试过程法和任务规划法需要维护历史记录，才能根据组织的实际经验来估计测试组规模。

最后说明一点，当估计测试工作量时，要考虑到异常情况，所以审查测试工作量化因子列表是非常重要的。

为了有助于未来的测试规划工作，记录当前项目每项任务实际花费的人员和时间常是一种行之有效的方法。这些数据在未来的测试工作量估算中是相当有用的。但是这里介绍的所有方法在实施中都面临同一个问题，那就是很少有两个软件开发项目是相似的。对于大多数项目，复杂度、开发人员的经验、技术和无数的其他变量是不会相同的

第3章 测 试 组

测试组能力的高低会极大地影响测试工作的成败。一个高效率的测试组应该同时具备与目前软件问题相关的技术知识和领域专业知识。对测试组来说，只精通完成测试任务所需的测试技术和测试工具还不够，根据软件问题域的复杂度，测试组还应该包括对问题域有深入了解的人员。他们可利用自己的知识创造制作出测试制品和数据，并且能够高效地实现测试脚本和其他测试机制。

除此以外，测试组还必须结构合理，每个成员必须被赋予角色和职责，分配的原则是使得测试人员在执行测试工作时相互之间重叠的职责最小，并且对哪些成员应该履行哪些义务必须分工明确。分配测试资源的一种方法是细化应用程序的功能性部分和非功能性部分。测试组需要的角色数量可以多于成员数量，这是测试经理应该考虑的问题。

对任何测试组来说，不断地评估每位成员的有效性对保证测试工作的成功是非常重要的。对测试人员进行评估需要调查他们在多方面的表现，其中包括发现缺陷的类型与遗漏缺陷的个数和类型。只根据发现缺陷的数量来评价一个测试工程师的表现绝不是一个好办法，因为这个度量本身并不全面。在评估中应该考虑许多因素，例如：所测功能的复杂度、时间限制、测试工程师的角色和职责、经验，等等。只有用合理的标准对测试组成员进行规范地评估才能提高整个测试工作的有效性。

第 13 条：定义角色和职责[①]

测试工作很复杂，测试组需要具备多种专门技术才能理解测试工作的范围和深度，并为测试工作制订测试策略。

为了使测试组的每一位成员都了解什么是必须完成的任务和谁是某项任务的负责人，我们需要定义并文档化测试组成员的角色和职责。这些角色和职责应该用口头或者书面的方式通知测试组的每位成员。为项目中每位测试组成员分配角色就使得每个人都清楚地了解什么人负责项目中的哪些部分。特别是当发生问题的时候，这种做法能够使团队的新成员快速地确定应该和谁取得联系。

为了确定由哪些人来完成某一项任务，我们应该提供一个任务描述。如果了解了任务涉及的范围，那么为这项任务分配特定的小组成员就变得容易了。

为确保测试任务的成功执行，我们应该开发**工作包**，并把它们发给测试组的成员。工作包一般包括任务的构成、技术方法、任务时间表、预计的费用、每个人的时间分配以及适用的标准和流程列表。

在一个项目中测试工程师角色的数量可能比测试组成员的人数还多（第 2 章中介绍过，需要角色的数量依赖于手头的测试任务）。在这种情况下，一个测试工程师可能有多个“头衔”，承担多个角色。

表 13.1 是一些测试工作中的角色相应的职责和需要的技能。

根据测试人员的技能和他们的其他长处，把适当的角色分配给合适的人是非常重要的。每个测试人员可以专门研究应用程序的特定部分和非功能性部分。为测试人员分配应用程序中的特定部分可以使他们成为这些特定部分的专家。除了应该关注特定的功能测试部分以外，测试组成员还应该专门研究特定的非功能性测试部分，例如：性能测试、兼容性测试、安全性测试和并发测试。

① 改编自 Elfriede Dustin 等人的 *Automated Software Testing*（Reading，Mass.：Addison-Wesley，1999），表 5.11，第 183~186 页（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

表 13.1 测试工作角色

试经理

责	技 能
负责部门之间的交往联络；测试组的代表 如果需要的话，与客户进行交流 招募人员、管理人员和培训人员 确定测试预算和时间表，包括对测试工作量的估计 测试规划，包括制订测试目标和测试策略 与供应商之间的交流 测试工具的选择和引进 联系测试和开发活动之间的纽带 获得测试环境需要的硬件和软件 测试环境和测试产品的配置管理 测试流程的定义、培训和持续改进 利用各种度量支持持续的测试流程改进 测试工作的监督和进度跟踪 协调测试前和测试后的各种会议	• 了解测试流程和方法论 • 熟悉测试工作的相关内容，包括测试环境和测试数据的管理、问题的发现和解决，以及测试设计和测试开发。 • 了解手动测试技术和自动测试的最佳实践。 • 了解应用商业领域和应用需求 • 精通测试目标、测试目的和测试策略的制订 • 熟悉各种测试工具、缺陷追踪工具和其他支持测试的 COTS 工具，并熟悉它们的使用 • 善于各方面的规划工作，包括对人员、设备和进度的规划

试负责人

责	技 能
测试工作（包括测试方法）的技术领导者 负责客户接口、人员招募、测试工具的引进、测试计划的制订、人员管理、费用和进度状态的报告 验证需求的质量（包括对可测试性的验证）、需求定义、测试设计、测试脚本和测试数据的开发、自动测试、测试环境配置，测试脚本的配置管理和测试的执行。 为了在项目中最充分地利用测试工具，负责和测试工具提供商进行交流	• 了解应用商业领域和应用需求 • 熟悉测试工作的相关内容，包括测试数据管理、问题的发现和解决，以及测试设计和测试开发 • 掌握各种技能的专业知识，其中包括编程语言、数据库技术和计算机操作系统 • 熟悉各种测试工具、缺陷追踪工具和其他支持测试生命周期的 COTS 工具，并熟悉其使用

职　责	技　能
• 跟踪最新的测试方法和工具，并且把有关知识传授给测试组 • 指导测试设计和测试过程的走查和检查 • 根据经验和教训对测试流程进行改进 • 测试追溯矩阵（从测试过程到测试需求的对应关系） • 测试流程的实现 • 确保被测试产品的文档的完整性	

可使用性[1]测试工程师

职　责	技　能
• 设计和开发可使用性测试场景 • 可使用性测试流程的管理者 • 定义可使用性测试的完成标准、分析测试结果，并且把结果呈现给开发组 • 撰写被测试产品的文档和报告 • 定义可使用性需求，和客户一起完善这些需求 • 参与测试过程的走查	• 精通测试套件的设计 • 了解可使用性方面的问题 • 熟练的测试技巧 • 出色的人际交往能力 • 精通 GUI 设计标准

手动测试工程师

职　责	技　能
• 根据功能性和非功能性需求设计和开发测试过程和测试用例（包括相关的测试数据） • 手动执行测试过程 • 参与测试过程走查 • 指导测试并准备测试进展和回归方面的报告 • 遵守测试标准	• 对 GUI 设计有很好的了解 • 精通软件测试 • 精通测试套件的设计 • 精通正在测试的应用程序的商业领域 • 精通测试技术 • 了解各种测试阶段 • 精通 GUI 设计标准

① 术语可使用性是指应用程序的用户界面的直观性和易使用程度。

自动测试工程师（自动测试人员/开发人员）

职　责	技　能
• 根据需求设计和开发测试过程和用例 • 设计、开发和执行可重用和可维护的自动测试脚本 • 使用记录/回放工具对 GUI 进行自动测试或者在适当情况下用编程语言或脚本开发测试工具 • 遵守测试设计标准 • 指导/参与测试过程的走查 • 执行测试并准备测试进度和回归方面的报告 • 为了了解测试工具的能力参加测试工具用户组及其相关活动	• 对 GUI 设计有很好的了解 • 精通软件测试 • 精通测试套件的设计 • 精通测试工具的使用 • 编程技巧 • 精通 GUI 设计标准

网络测试工程师

职　责	技　能
• 完成网络、数据库和中间件的测试 • 研究网络、数据库和中间件的性能监视工具 • 开发负载测试和强度测试的设计、用例和过程 • 协助负载测试和强度测试的过程的走查和审查工作。 • 根据实际的测试工作实现性能监视工具 • 指导负载测试和强度测试	• 网络、数据库和系统的管理技能 • 各种技能的专业知识，其中包括编程语言，数据库技术和计算机操作系统方面的知识 • 产品的评估和集成技巧 • 熟悉网络嗅探器和用来进行负载测试和强度测试的工具

环境测试专家

职　责	技　能
• 负责安装测试工具和建立测试工具环境 • 负责运用环境设置脚本建立并控制测试环境 • 创建并维护测试数据库（包括添加、恢复和删除数据库等等） • 在测试工具环境内维护层次化的需求	• 网络、数据库和系统的管理技能 • 各种技能的专业知识，包括编程语言和脚本语言，数据库技术和计算机操作系统方面的知识 • 测试工具和数据库经验 • 产品的评估和集成技能

安全性测试工程师

职责	技能
• 负责应用程序的安全性测试	• 了解安全性测试技术 • 安全性方面的背景 • 安全性测试工具方面的经验

测试库和配置专家[①]

职责	技能
• 测试脚本的变更管理 • 测试脚本的版本控制 • 维护测试脚本的重用库 • 在某些情况下需要建立各种测试版本	• 网络、数据库和系统的管理技能 • 各种技能的专业知识，其中包括编程语言，据库技术和计算机操作系统方面的知识 • 配置管理工具方面的专门知识 • 测试工具方面的经验

使用自动测试工具的小组应该包括具备软件开发技能的人员。自动测试要求开发、执行和管理测试脚本。由于这个原因，完成手动测试工作所需的技巧和活动与完成自动软件测试所要求的技巧和活动有所不同。因为这种差异，执行手动测试的角色应该在角色和职责定义列表中单独列出。这并不是说“自动测试工程师”任何时候都不能执行某些手动的测试任务。

如果要同时测试多个产品，那么让固定的测试组成员在一段时间内稳定地从事相同产品领域的工作可以使测试人员的工作连贯。对测试人员来说，在区别很大的技术和问题域之间频繁地进行工作切换是非常困难的。测试人员应该根据自己的特长一直从事同类型的产品（例如：Web 或桌面应用程序）或者相同的问题域方面的工作。最高效的测试人员除了技术上的专长以外，还应了解行业需求和系统需求。

表 13.2 是一个测试组组织结构的例子。表中假设测试人员已经具备了基本的测试技能，因此表中没有列出。例如：如果是在 Windows 环境下工作，那么假设测试组成员已经具备了 Windows 的技能；如果是在 UNIX 环境下工作，那么假设测试组成员已经具备了基本的 UNIX 技能。

① 这种类型的配置工作通常由独立的配置管理部门来完成。

表 13.2 测试组分工示例

职位	产品类型	责任/技能	角色和职责
测试经理	桌面应用程序 Web 应用程序	负责测试工作、客户接口、测试工具的引进与人员的招募和管理 技能：管理技巧、MS Project、Winrunner、SQL、SQL Server、UNIX、VC++、Web 应用和测试工具的使用经验	管理测试工作
测试负责人	桌面应用程序 Web 应用程序	人员管理，费用/进度/测试状态的报告以及测试的规划、设计、开发和执行 技能：TeamTest，Purity，Visual Basic，SQL，Winrunner，Robot，UNIX，MS Access，C/C++，SQL Server	[这里引用相关的测试需求] 为功能性测试过程开发自动测试脚本
测试工程师	桌面应用程序 Web 应用程序	测试工作的规划、设计、开发和执行 缺陷的识别和追踪 技能：测试工具的使用经验、财务系统的相关经验	[这里引用相关的测试需求] 开发自制测试工具
测试工程师	桌面应用程序 Web 应用程序	测试工作的规划、设计、开发和执行 缺陷的识别和追踪 技能：测试工具的使用经验、财务系统的相关经验	性能测试 [这里引用相关的测试需求]
测试工程师	桌面应用程序	测试规划、设计、开发和执行 缺陷的识别和追踪 技能：财务系统的相关经验	配置测试和安装测试 [这里引用相关的测试需求]
测试工程师	Web 应用程序	负责测试工具环境、网络和中间件测试 完成所有其他的测试活动 缺陷的识别和追踪 技能：Visual Basic，SQL，CNE，UNIX，C/C++，SQL Server	安全性测试 [这里引用相关的测试需求]
初级测试工程师	桌面应用程序	完成测试规划、设计、开发和执行 缺陷的识别和追踪 技能：Visual Basic，SQL，UNIX，C/C++，HTML，MS Access	[这里引用相关的测试需求]

表 13.2 确定了测试组的职位和他们在项目中的角色分工，以及正在测试的产品。表中列出了每个职位的人员必须履行的责任，以及这些职位所需要的技能。值得注意的还有为测试组的每个职位分配的产品。

本书的第 1 条强调了测试组从产品生命周期之初就介入项目的重要性。如果测试人员的及早介入已经成为一个组织的确定做法，那么在生命周期的每个阶段内定义和文档化测试组的各种角色就是可能的（也是必要的），其中包括测试组在每个阶段结束时期望交付的产品。

第 14 条：测试技巧、行业知识和经验三者缺一不可

最高效的测试组应该由具备各种专门技术的成员组成（例如：具备行业知识、技术知识、测试技术方面知识的人员），同时也应该由具备各种经验等级的成员组成（例如：新手和测试专家）。了解应用程序功能细节的行业专家（subject-matter expert，SME）在测试组中起着重要作用。

下面详细地阐述这些观点：

- 行业知识。一个专业测试人员可能认为深入地了解行业知识并不难，但是如果软件面向的行业非常复杂，那么情况就没有这么简单。彻底地了解某些问题域（例如：税法、劳动合同和物理学）可能需要花费几年的时间。我们可以讨论详细和明确的需求是否应该包含所有可能的细节，这样开发人员可以正确地设计系统，测试人员可以正确地规划测试工作。但是，实际情况中预算和时间都有一定的限制，这经常会导致需求不够详细，结果是需求的内容被随意解释的空间很大。甚至是详细的需求也经常会出现内部的不一致，这些问题必须要识别并排除。

  由于这些原因，在工作中每个 SME 必须与开发人员和其他 SME（例如：税法专家、劳动合同专家、物理学家）紧密协作来剖解需求的复杂性。如果有两名 SME，那么他们的意见必须达成共识。如果两名 SME 不能达成一致意见，那么就需要第三名 SME 的加入。经过适当地测试之后，测试 SME 会对最终的实现签字认可。

- 技术知识。虽然测试人员完全掌握问题域确实有价值，而且我们也期望这样，但是如果对软件工程（包括测试工作）缺乏一定程度的了解，那么也会降低测试人员的有效性。最高效的行业专家测试人员是那些对技术感兴趣并有相应的实践经验的人员——也就是那些上过一门或几门编程课或者有相关技术经验的人员。行业知识还必须由技术知识（包括对软件测试科学方面的知识）来补充才行。

  但是要保证测试工作取得成功，专业测试人员还必须深入了解技术平台和组成系统的构架。为了更好地进行测试的准备工作，一个专业测试人员应该掌握如何编写自动测试脚本、如何编写自制测试工具，以及理解诸如兼容性、性能和系统安装之类的技术问题。虽然对于 SME 来说掌握一些这样的技术知识是有

益的，但是，他们的技术知识水平不如专业测试人员水平高，这当然也是合情合理的。

- 经验等级。测试组完全由具有多年经验的专家级测试人员组成，这种情况很罕见，我们也没有必要指望这样。与所有工作一样，初学者也有其用武之地，但需要高级职员对他们进行培训和指导。为了确定需要培训和改进的方向，测试经理必须要评审需要的技能和个人实际技能之间的差距。

  我们可以安排初级测试人员测试风险较低的功能，或安排测试一些诸如 GUI 界面控制之类的外在特性（如果我们认为这个部分风险不高）。如果要分配一个初级测试人员测试风险较高的功能，那么应该安排一名高级测试人员与其一同工作，这样可以对初级测试人员提供指导和帮助。

虽然专业测试人员和行业专家测试人员侧重于测试工作的不同方面，但是应该鼓励这两种类型的测试人员互相协作。正如一个专业测试人员需要很长时间才能了解相关行业的所有细节一样，问题域专家或者行业专家要熟悉测试中需要考虑的技术问题也需要花很长时间。测试组应该提供交叉培训，使专业测试人员熟悉相关行业知识，也使行业专家了解技术问题。

有些测试任务可能需要技术方面或者行业知识方面的特殊技能。例如：可使用性测试工作应该由具有这方面经验或者至少对这方面较熟悉的测试人员来承担，而一个对这方面不熟悉的测试人员却只能猜测究竟是什么使得一个应用程序容易使用。本地化测试工作的情况与此相似，一个讲英语的人只能猜测 Web 站点在翻译成其他语言以后是否正确。一个更高效的本地化测试人员应该是站点所翻译成的语言为其母语的测试人员。

第 15 条：评估测试人员的有效性[①]

要保持测试工作的有效性，就需要对实现测试工作的各个元素（例如：测试策略、测试环境和测试组的组成）不断地进行评估，并且根据实际需要不断地改进这些元素。测试经理负责确保测试工作正在按照预定的计划实施，同时也要确保特定的任务正在按照预期情况执行。为了实现这个目标，测试经理必须跟踪、监督和评估测试工作的实现，这样他们才能在必要的时候对测试工作进行改进。

执行测试工作的核心力量是测试工程师。测试人员应正确地设计、撰写文档和执行高效的测试、准确地解释测试结果、记录缺陷、追踪缺陷直至最终解决问题，他们的这些能力是决定测试工作有效性的关键因素。测试经理设计的测试过程可能很完美，选择的测试策略也很理想，但是，如果测试组成员不能有效地执行这个测试过程（例如：有效地参与需求审查和设计走查），或完成指派的所有关键测试任务（例如：执行特殊的测试过程），那么有些重要缺陷就可能直到开发生命周期的晚期才被发现，并最终会导致费用增加。而更糟糕的情况是，缺陷可能根本就没有被发现，并且最终进入了软件产品。

测试人员的有效性很大程度上会影响和项目中其他小组的关系。例如：虽然应用程序运行一切正常，但是由于测试人员错误地理解了需求，他们可能会频繁地发现伪错误、报告“用户错误”，或者最糟糕的是经常遗漏关键的缺陷，这样测试人员就会失去测试组其他成员和其他小组的信任，并最终会损害整个测试工作的声誉。

评估测试人员的有效性是一项困难的任务，并且经常是主观的任务。除了评价所有雇员表现时需要考虑的典型因素（例如：出勤率、注意力、态度和主动性）以外，还有专门用于评估测试人员的有关测试的度量方法。例如：无论他们是技术型的测试人员、行业专家、安全性测试人员，还是可使用性测试人员，所有的测试人员必须要明确工作的方向和具备分析能力。

评估的过程应该从招聘工作就开始。评估工作的第一步是为每个职位雇用其角色和职责所需技能的测试人员（关于角色、职责和技能的讨论，请参见第 13 条）。

当测试组是“继承”而来的，而不是专门为项目招募而来的时候，评估工作更加复

① 改编自 Elfriede Dustin 的“Evaluating a Tester's Effectiveness”，Stickyminds.com（Mar.11，2002）。参见 *http://www.effectivesoftwaretesting.com*。

杂。在这种情况下，测试经理必须熟悉各类测试人员的背景，这样就可以根据他们的经验、专业知识和背景为他们分配任务并进行评估。进一步了解了他们的能力之后，可能需要调整一些测试组成员的角色。

如果没有指定的角色和职责、任务、时间表和标准，就无法评价测试工程师的表现。测试经理必须首先明确地表明在什么时间对测试工程师的期望是什么。

下面列出了对测试人员的期望，这些期望必须要和测试人员达成共识：

- 遵守测试标准和测试过程。测试工程师必须了解需要遵守的标准和过程，并且必须就过程进行充分的交流。标准和过程在第 21 条中讨论。
- 保持进度。测试人员必须了解测试时间表，其中包括必须交付测试计划、测试设计、测试过程、脚本和其他测试产品的时间。除此以外，所有测试人员都应该知道要测试的软件组件的交付时间表。
- 达到目标和完成指派的任务。为每个测试人员分配的任务必须形成文档并且需要和他们进行充分交流，同时还必须确定截止期限。测试经理和测试工程师必须就分配的任务达成共识。
- 控制预算。当测试人员评估必须购买的测试工具和其他测试技术时，他们必须了解可供支配的预算，这样测试人员才会在预算允许的范围内开展工作，而不会浪费时间去评估过于昂贵的产品。

由于手头的任务和测试人员具备的技能有所不同，所以对测试工作的期望和任务分配也会有所不同，希望采用的测试类型、测试方法、测试技术以及测试结果也不相同。

一旦设定好预期情况，测试经理就可以对照预先确定的目标、任务和进度来比较测试组的实际工作，这样可以测量实现的有效性。下面是评估测试人员有效性时需要考虑的一些要点：

- 行业专家和技术专家。行业专家具备的专门技术与应用程序的行业领域有关，而技术型的测试人员只对应用程序的技术问题感兴趣。

  当一个技术型的测试人员履行自动测试职能的时候，就应该根据测试工程师必须遵守的预定标准来评估自动测试过程。例如主管可能会问：测试工程师开发的自动测试脚本是可维护的、模块化的和可重用的，还是用于系统的每个新版本时都必须修改脚本？测试人员是否遵从了最好的做法，例如：是否确保把测试数据库纳入了基线，并且当需要重新运行测试脚本时能够恢复基线数据库？

如果测试人员正在开发自定义的测试脚本或者自制的测试工具，那么评估这种测试人员时还应该使用评估开发人员的一些标准，其中包括代码的可读性和可靠性。

如果一个专门研究如何使用测试工具的测试人员却不了解应用程序功能的复杂性和基本概念，那么这种测试人员的工作一般是低效的。源于对应用程序肤浅理解的自动测试脚本一般只能发现不太重要的缺陷。要成为一名高效的测试组成员，自动测试人员也要了解系统功能，这一点非常重要。

另一个需要评估的方面是技术能力和适应能力。测试工程师能否无师自通地掌握新工具并且熟悉其功能？如果测试人员尚未完全熟悉测试工具的各种功能，那么应该对他们进行培训。

- 有经验的测试人员与初学者。我们在前面已经提到过，测试人员的技能水平必须要受到重视。例如：初学者可能会忽视一些缺陷，或者对缺陷视而不见，根本没有意识到它们是缺陷。因此为测试新手分配风险低的测试任务是很重要的。

  并不是只有缺乏经验的测试人员才会遗漏缺陷，有经验的测试人员由于过去的经验（“这个产品总是能完成某项工作”）或者由于反复从事相似的工作可能也会忽略某些类型的缺陷。不管是否应该，测试人员可能会对某些熟悉的缺陷麻木了，并且不再报告那些他们认为不重要但是最终用户却不能接受的缺陷。

- 功能性测试与非功能性测试。测试经理应该评估的还有：测试人员对各种可利用的测试技术的了解程度，以及对哪种技术能够提高手头测试任务的工作效率的了解程度。如果测试人员不了解各种测试技术并且错误地运用了某种技术，那么就会对测试设计、测试用例和测试过程产生负面影响。

  功能性测试还应该建立在对测试过程的评审基础上。一般来说，测试人员会根据分配的需求获得测试指定功能部分的测试过程。对测试过程的走查和审查应该接受包括需求、测试和开发组在内各个小组的指导。在走查过程中，应该确保所有小组对应用程序行为的理解是一致的。

  在评估功能测试过程时，应该考虑下列问题：

  - 测试过程中的步骤是否完全映射为需求的步骤？是完全可追溯的吗？
  - 测试的输入、步骤和输出（预期的结果）正确吗？
  - 在测试过程的功能流中遗漏了重要的测试步骤吗？

- 在设计有效的测试场景时是否经过了一个分析、思考的过程？
- 是否遵从了测试过程创建标准？
- 在认定测试过程有效和完整之前，由于误解和缺乏交流导致的修改次数是多少？
- 在制作测试用例过程中是否运用了有效的测试技术？

在对测试过程走查过程中，必须验证测试过程的“深度”或者是否全面。换句话说，测试过程测试的内容是什么？它是只测试了较高层面上的功能，还是确实触及到了应用程序的实质性功能？

在某种程度上，这个问题和需求的深度有关。例如：一个功能性需求可能规定“系统应该允许添加 A 类型的记录”，一个较高层面的测试过程只是验证通过 GUI 可以添加记录。而更有效的测试过程还应该包括对应用程序中由于插入记录而受到影响的那些部分的测试。例如：通过一条 SQL 语句就可以验证新记录是否已正确地插入到数据库表中，而通过其他的步骤还可以验证记录的类型是否为 A，另外还需要考虑许多其他测试步骤，例如：当添加多条 A 类型的记录时验证系统的行为，比如说是否允许存在重复记录。

如果编写的是较高层面的测试过程，那么要确认对应的需求也处于同样的层面，而且要确认没有遗漏相关的细节，这一点是非常重要的。如果在测试过程中遗漏了需求中的细节，那么测试工程师就应该接受培训，学习如何编写有效测试过程。当然也可能是测试工程师对需求的理解不够充分。

评估功能性测试和非功能性测试要运用不同的标准。例如：非功能性测试的设计方法和文档化方法与功能性测试过程使用的方法不同。

- 测试阶段。在不同的测试阶段（α 测试、β 测试、系统测试、用户验收测试，等等），测试人员要完成不同的测试任务。

在系统测试阶段，测试人员要负责本书中描述的所有测试任务。包括开发和执行测试过程、追踪缺陷直至改正，等等。其他测试阶段可能不需要这么全面的内容。

例如：在 α 测试期间，测试人员的任务可能只是简单地重现和记录独立的“α 测试组”报告的缺陷，α 测试组一般是公司中独立的测试（独立验证和确认）（Independent Verification and Validation，IV&V）组。

在 β 测试期间，除了重现和记录其他 β 测试人员（客户经常会成为 β 测试人员）发现的缺陷以外，测试人员还需要负责把要执行的 β 测试过程文档化。

- 开发生命周期的各个阶段。有一个观点贯穿本书的始终，那就是测试人员应该从生命周期的开始就介入项目。对测试人员表现的评估也应该按阶段进行。例如：在需求阶段可以根据缺陷预防工作来评估测试人员，比如说要识别出可测试性问题和需求不一致问题。

虽然对测试人员的评估可能是主观的，但是我们还是应该重视和各个测试阶段有关的因素，而不是凭第一印象得出表面的结论。例如：在需求阶段评估测试人员时，考虑需求本身的质量非常重要。如果需求写得非常糟糕，那么即使一个普通的测试人员也能发现许多缺陷。但是如果需求写得很好并且质量上乘，那么只有水平高的测试人员才可能发现细微的缺陷。

- 服从命令和关注细节。关注一个测试工程师遵照指示和注意细节的程度是非常重要的。测试经理对测试人员的可依赖程度和贯彻指示的持久性要做到心中有数。如果为了保证产品质量必须要更新和执行测试过程，那么测试经理必须对完成这项任务的测试人员有信心。如果测试是自动执行的，那么测试经理应该对工作进展有信心。

周例会是一种跟踪和测量工作进度的有效方法，在会上测试工程师报告他们的工作进度。在测试阶段的最后时期，类似的会议应该每天召开。

- 缺陷类型、缺陷率和缺陷文档。在评估过程中还必须考虑测试工程师发现的缺陷类型。当使用这个度量来评估测试人员有效性时，必须要牢记一些因素，其中包括：测试人员的技能水平、正在进行的测试类型、处在什么测试阶段以及正在测试的应用程序的复杂度和成熟度。发现缺陷不仅依赖于测试人员的技能，而且还依赖于编写代码、调试代码和对代码进行单元测试的开发人员，同时还依赖于负责评审需求、设计和编码的走查和审查小组。理想情况下，最好在测试工作正式开始之前，他们已纠正了绝大多数缺陷。

在这种情况下，还有一个影响评估的因素是确定测试工程师发现的缺陷是复杂的、与行业领域相关的缺陷，还是简单的外观缺陷。外观缺陷相对来说比较容易发现，在可使用性测试期间它们的优先级较高。例如：窗口文字丢失或者控件移位。但是，与数据有关或者在应用程序的元素之间有因果关系的问题就比较复杂，相对来说更不容易发现，发现这些缺陷要求对应用程序有更进一步的了解，在功能测试期间它们的优先级较高。但是，由于外观缺陷大多数是明显

的，所以它们会更直接地影响客户的满意度。

测试经理还必须考虑测试人员负责的具体区域。即使测试人员所负责的部分中大多数缺陷是在产品中才发现的，也不能就此断定这位测试人员表现不佳。如果测试人员所负责的这部分非常复杂并且容易出错，而且产品发行得又非常匆忙，那么遗漏了一些缺陷也情有可原。

在产品中发现的缺陷类型也是非常重要的。如果某个缺陷可以通过现成的测试过程套件中的基本测试发现，并且测试人员也有足够的时间来执行这些测试过程，那么这就是负责此部分的测试人员的重大疏忽。但是，在下结论之前，我们还应该考虑下列问题：

- 测试过程是需要手动执行的吗？手动测试人员可能已经厌倦了一遍又一遍地执行相同的测试过程，并且因为应用程序的被测部分以前一直运转良好，所以在多次例行公事以后就得出肯定的结论而不再执行测试过程。
- 软件是在截止期限的压力下发行的吗？此时即使完全取消整个测试周期也无法改变软件发行日期。尽管有时间上的压力，但不满足发行标准就不应该允许软件发行。
- 测试是自动执行的吗？自动测试脚本遗漏的测试步骤中是否可能存在缺陷？如果出现了这种情况，那么必须重新评估自动测试脚本。
- 缺陷是通过一些很少执行的基本功能的组合发现的吗？发生这种类型的缺陷也情有可原。

除此以外，当测试工作启动时，还需要对测试目标、项目的风险和所作的假设进行评审。如果由于时间限制或者风险很低而决定放弃某类测试，那么测试人员就不应该对此承担责任。只有对可能发生的问题有充分的了解，才能接受这种风险。

有效性也可以通过检查缺陷的文档化方式进行评估，缺陷报告中是包含了足够的细节以使开发人员能够重现问题呢，还是让开发人员花费大量时间才能重现测试人员所发现的缺陷呢？必须制订一个详细描述在缺陷报告中需要哪些信息的标准，同时还要充分交流和理解缺陷追踪的生命周期。所有测试人员必须遵从这些标准（关于缺陷追踪生命周期的讨论，请参见第 50 条）。

在对测试人员进行评估的过程中，对发现的每个问题都应该确定问题的原因并找到解决方案。对一个测试人员的能力下结论之前，必须先要仔细地评估每个问题。在认真地评估了所有因素并且提供了针对性的附加培训以后，才能对测试人员的具体发展方向、分析能力和有效性等方面做出评价。如果我们确定了一个测试人员不注意细节、缺乏分析能力或者不善于沟通，那么这个测试人员的表现就应该受到严密的监视和检查，并且还可能需要对他进行更多的指导和培训，或者采取其他恰当的步骤。

要确保测试工作成功完成，就必须不间断地对测试人员的有效性进行评估。

测试工程师自我评价

测试工程师有责任评估他们自己的有效性。下面列出的问题可以作为制订测试工程师自我评价过程的起点。在这里，我们假定测试工程师已经理解了分配的任务、角色以及职责：

- 关注发现的缺陷类型。发现的缺陷重要吗？或者还是大部分是外观上的、低优先级的缺陷？如果测试人员总是只能发现低优先级的缺陷（例如：在功能测试中，发现快捷键工作不正常，或者 GUI 上的文字排列有缺陷），那么就需要重新评估测试过程的有效性。请记住，在其他测试阶段（例如：这里提到的可使用性测试），这些缺陷的优先级会发生变化。
- 测试过程足够详细吗？是否覆盖了发现高优先级的缺陷所必需的深度以及数据和基本功能的组合与变化？是否同时包含了对非法数据和合法数据的测试？
- 是否听取了来自需求人员、开发人员和其他测试人员对测试过程的反馈意见？如果没有，那么测试工程师应该请求这些小组对测试过程进行评审、审查和走查。
- 为了挑选最有效的测试过程，测试工程师是否对可以利用的测试技术（例如：边界值测试、等价类划分和正交排列）有足够充分的了解？
- 测试工程师是否对应用程序功能的实质和行业领域具有足够深入的了解？如果不是这样，那么测试人员应该加强总体了解并另外接受培训。技术型的测试人员可以向行业专家（SME）请教。
- 是否主要缺陷在测试生命周期中发现得太晚了？如果这种情况经常发生，那么应该考虑下面的要点：

  测试工作的开始阶段是否集中在低优先级的需求上？测试工作的初始阶段应该关注高优先级、高风险的需求。

测试工作的开始阶段是否集中在对已经交付的功能的回归测试上，而这些功能此前一直运转正常、几乎没有失败过？测试工作的开始阶段应该关注代码修改、缺陷修正和新功能。回归测试应该晚些时候再进行。理想情况下，回归测试工作最好自动执行，这样测试工程师就可以集中精力测试其他新交付的部分。

- 是否某些正在测试的部分中发现的缺陷少得令人惊奇？如果是这样，那么应该重新评估这些部分来确定：
  - 测试覆盖的内容是否足够全面。
  - 正在执行的测试类型是否是最高效的，是否遗漏了重要的步骤。
  - 正在测试应用程序部分的复杂度是否很低，缺陷确实很少。
  - 功能的实现方式是否不可能留下重要缺陷，例如：编码工作由最高级的开发人员完成并且已经通过了单元测试和集成测试。

考虑缺陷处理的工作流程：

- 每个缺陷都应该及时（也就是缺陷一旦被发现并且得到了验证以后）形成文档。
- 必须遵守生成缺陷报告的标准。如果还没有制定缺陷报告的标准，那么应该向测试经理索取。标准中必须列出生成缺陷文档所需要的全部信息，以确保开发人员能够重现缺陷。
- 如果拿到了一个新版本，那么测试工作应该先集中返测以前出现的缺陷。尽可能早地返测纠正的缺陷是非常重要的，这样开发人员才能了解他们的更正工作是否成功。
- 应该不断地评估开发组对缺陷报告质量的意见。如果开发人员经常表示缺陷报告缺乏必要的信息（例如：对重现缺陷的操作步骤缺少详细描述），那么测试人员应该努力地提供更好的缺陷报告。
- 测试人员应该积极地追踪缺陷直至解决。

- 检查附在缺陷文档上的意见，这样可以确定开发人员和其他测试人员对它的接受程度。如果缺陷报告上经常标注着“工作正常”或者“不能重现”，那么可能也表明存在下面一些问题：
  - 测试人员对应用程序的理解可能不够充分。在这种情况下，他们需要接受更多的培训。也可能需要行业专家（SME）的帮助。
  - 需求可能不够明确。如果是这样，那么必须澄清需求（一般来说，这个问题会在需求或者测试过程的走查和审查过程中发现）。
  - 测试工程师撰写文档的技能可能离要求还有差距。文档不够详细可能会导致对已经识别的缺陷的误解。缺陷描述中可能还需要一些详细步骤才能使开发人员重现缺陷。
  - 开发人员可能误解了需求。
  - 开发人员可能缺乏耐心来按照详细的缺陷报告重现缺陷。

- 当产品转入生产阶段以后，测试人员应该密切注视自己负责的部分是否发现了缺陷。如果有这样的问题，那么必须要对它们进行评估来确定其漏网的原因：
  - 测试人员是否放弃执行了一个可能发现漏网缺陷的测试过程？如果是这样，那么为什么会忽略这个测试过程？回归测试是自动进行的吗？
  - 是否没有测试过程能够发现这些缺陷？如果是，那么原因是什么？是不是认为这个部分风险很低？此时应该重新评估测试过程的创建策略，并且应该在回归测试中添加一个用来发现这种问题的测试过程。测试人员应该和同行或者测试经理讨论如何建立更加有效的测试过程，包括测试设计、测试策略和测试技术。
  - 是否由于时间的关系没有执行一个现成的测试过程？如果是这样，那么应该在应用程序发货之前，而不是发货之后把情况通知管理层。此类问题也应该在测试结束以后，用户安装之前的会议上进行讨论，并且应该写入测试报告。
- 其他测试人员在他们的工作中是否发现了属于某位测试人员负责的缺陷？如果是，那么这位测试人员应该查找原因并且进行相应的调整。

根据测试阶段和手头的测试任务、专业知识的类型（技术知识还是行业领域知识）、测试人员的经验等级，测试人员还可以提出许多与测试有效性相关的问题。

自动测试人员渴望熟悉自动测试标准和最好的自动测试实践。而性能测试人员可能会要求更多的培训，来了解使用的性能测试工具和采用的性能测试技术。

测试人员能力的自我评估和随后的改进措施是高效测试工作的重要组成部分。

第4章 系统构架

要正确地测试应用程序，所需要的不仅仅是简单地对模拟和重现的用户动作加以验证。不了解系统的内部构架和组件，只通过用户界面进行测试的方法就是典型的**黑箱测试**。黑箱测试本身并不是最有效的测试方法。要全面地验证应用程序在功能方面的正确性，就需要设计和实施最有效的测试策略，这就要求测试组必须对系统内部（例如：构成系统的主要组成部分）有一定程度的了解，基于这种了解，测试组就能够设计出更有效的测试方案，并能更高效地发现缺陷。在测试一个系统或者应用程序时，如果直接把系统的各种模块和层次作为测试目标，那么这种测试方法就叫做**灰箱测试**。

了解了组成整个系统的组件和模块，就可以减轻测试组的工作量，而且测试组可以把注意力集中在有缺陷的特定区域或者层次上，从而提高开发人员修正缺陷活动的有效性。黑箱测试人员只能报告缺陷的结果或者症状，原因是他们必须依赖用户界面显示的错误消息或者其他信息，例如："无法生成报告"，同时黑箱测试人员也很难确定缺陷的漏识别和缺陷的误识别，而灰箱测试人员不仅能通过用户界面看到错误信息，而且还掌握了诊断错误的工具并能报告缺陷产生的原因。了解系统构架还能使测试人员抓住测试工作的重点，针对性地测试应用程序中在构架上的敏感区域，例如：数据库服务器或者核心计算模块。

第 1 章中已经提到：测试组必须参与需求文档的编写过程，与此同等重要的是，测试组还必须参与应用程序系统构架的评审。这样测试组在项目生命周期的早期就能发现潜在的可测试性问题。例如：一个应用程序的系统构架中过多地使用了第三方的产品，这会给系统测试和缺陷诊断带来麻烦，因为测试组所属的组织并不能控制这些组件的源代码并对它们进行修改。测试组必须尽早发现这类问题，以便在确定有效的测试策略时把这些因素考虑在内。过于复杂的构架（例如：利用了许多松散连接的现成产品）也会导致缺陷难以定位和重现。再强调一次，为了制定更好的测试计划，测试组必须及早地发现这些问题。

如果系统实现得很好，那么系统本身就会在许多方面简化测试过程。日志和**跟踪**机制在开发和测试工作中对追踪应用程序的行为帮助极大。此外，即使是在应用程序发行以后，不同的工作**模式**（如调试模式和发行模式）也有助于发现和诊断应用程序中存在的问题。

第 16 条：了解系统构架和基本组件

如果测试工程师对应用程序的构架和基本组件有所了解，那么这些知识会有助于他们查明产生特定测试结果的应用程序中的各个部分。这种了解可以指导测试人员进行灰箱测试，而灰箱测试是黑箱测试的有力补充。在灰箱测试过程中，测试人员可以识别应用程序失败的特定部分，例如：测试人员能够确定系统中容易出错的那些部分，确定的理由是由于这些部分的复杂性或者只是由于它们是不稳定的“新”代码。

下面这些例子表明了对系统构架的全面了解可以对测试工程师有所帮助：

- 提高缺陷报告的质量。测试过程一般都依赖于需求，因此它会沿着一条相对固定的路径经过系统。当在这条路径上发生错误的时候，如果测试人员能够在缺陷报告中反映与系统构架相关的一些信息，那么系统的开发人员势必会极大地从中受益。例如：如果某个对话框不能显示，那么测试人员通过调查确定错误的根源是从数据库中检索数据时出了问题，还是应用程序不能和服务器建立连接。

- 提高进行探索性测试的能力。如果某项测试失败了，测试人员通常需要进行一些集中测试，也许是通过修改原始的测试场景来确定应用程序的“断裂点”，断裂点指的是导致系统崩溃的因素。在寻找断裂点的过程中，测试人员所具备的被测系统的构架方面的知识会给他们的工作带来极大的帮助，他们可以进行更有用的和更有针对性的测试，如果对基本组件的了解能够获得足够的有关问题的信息时，他们甚至可以完全跳过多余的测试工作。例如：如果确定了应用程序遇到的是数据库连接方面的问题，那么测试人员就不必试着用不同的数值进行操作，而是关注数据库连接方面的问题。

- 增加测试的精确度。灰箱测试是为了通过用户界面或者直接面对基本组件检验应用程序而设计的，它通过监视基本组件的行为来确定测试是成功还是失败。因此灰箱测试理所当然地会提供与缺陷发生原因有关的信息。

下面是在测试中遇到的一些最普遍的错误类型：

- 一个组件遇到了某些类型的错误，导致操作中止。用户界面一般会显示：某个错误发生了。

- 执行测试后产生的结果不正确，与预期的结果不符。系统某处的组件对数据的处理不正确，导致了错误的结果。

- 组件执行失败了，但是并没有通知用户界面错误发生了，这称之为漏报。例如：输入的数据没有存储在数据库中，但是没有向用户报告任何错误。

- 系统报告了错误，但是事实上所有处理都是正确的，这称之为误报。

在第一种情况下（即错误导致操作中断的情况），显示一些实用的和描述性的错误提示信息是非常重要的，但是实际情况往往并不是这样。例如：在操作过程中发生了数据库错误，用户界面一般会显示诸如“未能完成操作”之类的模糊信息，却并没有任何有关错误原因的内容。有价值的错误消息会提供更多的信息，例如：“由于数据库错误未能完成操作”。应用程序内部的错误日志可能会提供更丰富的信息。对系统组件有所了解，测试人员就能利用所有可以利用的工具（其中包括日志文件和其他监视机制）更精确地测试系统，而不是完全依赖用户界面所报出的信息。

测试组了解系统构架有好几种方法。最好的方法莫过于在开发人员提出备选的构架时，测试组参与对构架和设计的评审。应该鼓励测试人员评审系统构架和设计文档，并且向开发人员提出问题。每次版本升级后测试人员参与对系统构架变化的评审也非常重要，这样他们才不会遗漏任何影响测试工作量的因素。

第 17 条：确认系统的可测试性

许多大型系统是由许多子系统组成的，而这些子系统又是由处于一个或者多个层次的代码和其他支持组件（例如：数据库和消息队列）组成的。用户和系统边界或者用户界面进行交互，随后系统再和其他子系统进行交互来共同完成一项任务。系统中的子系统、层次和组件越多，那么测试这个系统时定位一个问题可能就越难。

考虑下面这个例子。用户界面接受来自用户的输入，并且利用应用程序各个层次的代码提供的服务，最终把输入的数据写入数据库中。随后另一个子系统（例如：报告子系统）读取这些数据，并另外进行一些处理来生成一个报告。如果在这个过程的任何环节中发生错误，可能是由于用户输入的数据错误，也可能是并发性的问题，那么就很难对产生的缺陷进行隔离定位，并且错误也很难重现。

在应用程序的构架最初形成的过程中，测试人员就应该有机会提出以下问题：怎样才能通过系统内的一条路径来跟踪输入。例如：如果某个功能导致了一个其他服务器上的任务启动，那么如果测试人员可以通过一种方法来验证远程任务确实按照要求启动了，这会大有好处。如果提议的构架使得追踪这类交互非常困难，那么可能就需要重新考虑系统构架了，也许需要使用一个更可靠、可测试性更好的构架。测试策略必须考虑诸如此类的这些问题，并且在某些情况下可能需要和开发人员，包括第三方组件的提供商，一起来进行集成测试。

测试组必须要关注所提议构架的所有方面，以及这些方面对应用程序测试工作的有效性和效率可能产生的正负两方面影响。例如：虽然使用第三方的组件（例如：现成的用户界面控件）会节省构架中重要组件的很大一部分工作，从而缩短开发时间，但是它们的使用会对可测试性造成负面影响。除非能够获得可以修改的源代码，否则，由于第三方组件没有提供跟踪和其他诊断信息，所以追踪经过它们的路径可能会非常困难。如果在应用程序的构架中准备使用第三方的组件，那么实现一个原型来验证确实有一些方法可以监视经过这个组件的控制流，这是非常重要的。第三方的组件可能还会妨碍一些测试工具的使用，这会对测试工作产生负面影响。

当应用程序的构架还只是停留在文档上而没有付诸实施时，对构架可测试性的研究可以极大地减少随后的测试工作中可能出现的意外。如果不能确定系统构架的一方面或多方面的问题是否会对测试工作造成影响，那么测试组应该坚持用一个原型来使测试人员实验各种测试技术。从这种实验得到的反馈信息能够确保用这种方式开发的应用程序，其质量是可以进行验证的。

第 18 条：使用日志增加系统的可测试性

增加应用程序可测试性最常用的方法就是实施日志机制即跟踪机制，这种机制提供的信息有：组件正在做什么（包括它们正在操作的数据）、应用程序的状态或者应用程序运行中遇到的错误。在执行测试过程期间测试工程师可以利用这些信息来跟踪处理流程，他们还可以利用这些信息来对系统中发生的错误进行定位。

在应用程序执行过程中，所有组件写入的日志条目详细描述了它们正在执行的方法（也称为函数）和正在操作的主要对象。在执行了一个或者多个测试过程之后，日志条目通常会写到一个磁盘文件或者数据库中，为了方便以后的分析或者调试工作，这些条目应该具有适当的格式。在一个复杂的客户端-服务器或者 Web 系统中，日志文件可能在多台主机上生成，所以日志中包含足够的信息以便确定主机之间的执行路径就显得非常重要。

为了有助于分析和调试工作，跟踪机制必须把足够多的信息写入日志，但是也不能写入大量多余的和无用的信息，因为这样反而会使从中确定重要条目变得很困难。

一个日志**条目**只是一条具有一定格式的消息，它包含了可以用于分析工作的关键信息。一个形式良好的日志条目会包含以下一些信息：

- 类名和方法名。如果一个函数不是任何类的成员，那么也可以是函数名。在确定一条经过若干组件的执行路径时，这类信息非常重要。
- 主机名和进程 ID。 如果日志条目是由不同主机的事件或者同一主机的不同进程的事件所产生的，那么通过主机名和进程 ID 就可以对日志条目进行比较和跟踪。
- 条目的时间戳（最好精确到毫秒）。如果多个事件是同时发生的或者是在不同主机上发生的，就可能会导致这些条目不按顺序写入日志中，那么给每个条目一个精确的时间戳就使得事件之间可以相互关联。
- 消息。在条目中最重要的内容之一就是消息。它是一个由开发人员编写的、有关应用程序在某个时刻发生了什么事件的描述，也可以是有关执行期间遇到的错误的描述或者是一个操作的结果码。其他类型的消息还包括主域对象的永久实体的 ID 或键值，这样就能够在测试过程执行期间追踪经过系统的对象。

通过逐个检查系统中的每个组件的每个方法或者函数写入日志文件的条目，我们可以跟踪测试过程在系统内的执行过程，并且可以与测试过程正在操作的数据库（如果用到了数据库）中的数据相互关联。当系统严重失灵时，日志记录会指出哪个组件是罪魁祸首。如果是计算方面的错误，那么日志文件会列出所有参与测试过程执行的组件，还有所有实体使用的 ID 或键值。再加上数据库中的实体数据，这些信息对于开发人员确定源代码中的错误来说应该是足够了。

下面是某个应用程序的日志记录的示例，这个应用程序负责从数据库中检索一个客户对象：

```
Function:    main (main.cpp, line 100)
Machine:     testsrvr (PID=2201)
Timestamp:   1/10/2002 20:26:54.721
Message:     connecting to database [dbserver1, customer_db ]

Function:    main (main.cpp, line125)
Machine:     testsrvr (PID=2201)
Timestamp:   1/10/2002 20:26:56.153
Message:     successfully connected to database
             [dbserver1, customer_db]

Function:    retrieveCustomer (customer.cpp line 20)
Machine:     testsrvr (PID=2201)
Timestamp:   1/10/2002 20:26:56.568
Message:     attempting to retrieve customer record
             for customer ID [A1000723]

Function:    retrieveCustomer (customer.cpp line 25)
Machine:     testsrvr (PID=2201)
Timestamp:   1/10/2002 20:26:57.12
Message:     ERROR: failed to retrieve customer record,
             message [customer record for ID A1000723
             not found]
```

这个日志文件摘录展示了应用程序日志的一些主要方面，它们有助于提高测试工作的有效性。

在每个条目中，都标明了函数名、产生这个条目的应用程序源代码的文件名和代码行号、主机名、进程 ID 以及条目产生的时间。每条消息还包含了有关参与当前活动的所有组件的标识信息，例如：数据库服务器是“dbserver1”，数据库是“customer_db”，客户 ID 是“A1000723”。

这个日志明确地表明应用程序不能成功地检索指定客户的记录。在这种情况下，测试人员应该检查服务器 dbserver1 上的数据库，并且用 SQL 工具在 customer_db 数据库中查询 ID 为 A1000723 的客户记录来确认它是否确实不存在。

测试人员可以把这些细节信息作为缺陷报告的一部分交给开发人员，所以这些信息大大增强了测试工作的缺陷诊断能力。这样测试人员不仅能够报告缺陷的表面“症状”，而且也能够准确地指出导致问题发生的应用程序的内部行为。

第 19 条：验证系统支持调试和发行两种执行模式

如果软件系统正处于开发过程中，那么很明显问题会很多。在开发人员自测试阶段，特别是在单元测试和集成测试阶段以及测试组负责的正式测试阶段中，问题随时会出现。

当发现缺陷时，必须有办法能够诊断问题，而且最好是在不需要修改应用程序本身的前提下进行诊断。为了使这个过程有效，应用程序必须支持不同的运行模式，尤其是调试模式和发行模式，**调试模式**的作用是当遇到问题时帮助开发人员和测试人员诊断问题，而**发行模式**是去掉大多数或者全部与调试相关的特性，并经过优化的系统版本。交付给最终用户的应用程序一般是发行模式。

根据应用程序开发工作中使用的语言、工具和平台，有下面几种调试方法：

- 源代码检查。如果问题不太复杂，那么用眼睛检查源代码可能就足以确定问题了。例如：如果应用程序窗口的标题栏不正确，那么开发人员只需在源代码中查找窗口的标题栏文本的定义，并且对它做相应的改变就可以了。
- 输出日志。如果应用程序具备第 18 条中描述的日志机制，那么开发人员可以运行应用程序，重现错误，然后检查生成的日志条目。而且日志机制更大的好处是测试人员可以在缺陷报告中提供日志文件的摘要，如果摘要中包含的信息足以使开发人员确定问题发生的原因，那么缺陷就可以在源代码中得到改正。否则，开发人员必须考虑这里列出的其他调试技术。
- 实时调试。这是功能最强的调试技术，开发人员把一个调试器“附到”应用程序正在运行的实例上，并且在和应用程序直接交互的同时监视调试环境中的变量和程序流。通过执行缺陷报告中描述的步骤来重现缺陷，在确定导致缺陷的原因时，运行应用程序的同时要检查应用程序的代码。

虽然使用哪种调试技术是开发人员的事情，但是根据缺陷报告中的信息经常可以很容易地选择其中的一种技术。例如：如果问题只是简单的外观错误，那么只选用快速地检查源代码这一技术就足够了，而更深入和更复杂的错误一般需要选用实时调试技术。

但是实时调试也并不是总能行得通。例如：在 C++中，实时调试需要系统的**调试版本**（在系统的开发过程中，调试代码还能完成许多其他的检查工作，例如：栈和堆是否损坏）。前面已经提到过，当应用程序正在运行时，开发环境可以“附到”应用程序的调试版本上，使开发人员在和应用程序交互的同时可以监控代码。这样，应用程序的行为

和变量可以得到彻底的检查——有时这是诊断并修正缺陷唯一适用的方法。

此外，应用程序的调试版本会提供更详细的有关遇到问题的信息。这些信息涉及内存和其他底层系统功能。例如：若应用程序运行时遇到了底层错误或者意外条件（断言，那么一般会显示一个描述问题的对话框。这种底层的、用于诊断的信息对最终用户没有什么意义，所以通常系统的发行版本中并不包含这些信息，这样，如果在发行版本中发生了问题，那么系统只会崩溃或者表现出不可预测的行为。

当准备发行系统时，需要关闭或者删除有关调试的特性。有些语言中，调试特性的去除要求重新生成系统。这种做法是必要的，因为调试特性（例如：记录日志）通常会导致性能下降和占用更多的内存，并且会在无人注意的情况下耗尽系统的资源，例如磁盘空间。此外，因为过于详细的日志信息可能会给系统的入侵者暴露太多的信息，所以它们会造成安全性方面的风险。

因为生成应用程序的发行模式时可能会产生新的缺陷，所以对于测试组来说，了解正在测试系统的生成模式是非常重要的：一般来说，只有开发人员才应该使用调试版本，而测试组则应该在发行版本下工作，这样会保证软件的发行版本（将要交付给客户的系统版本）受到足够的重视。

在系统的发行版本中保留日志记录机制也必须受到重视，日志机制可以从应用程序中完全删除，但是更好的方法是通过配置记录少量日志或者干脆不记录日志。虽然好像应该从应用程序的发行版本中将日志全部删除，但是保留某些形式的日志还是值得考虑的。如果在应用程序的产品中恰当地加入了一个可配置的日志机制，那么即使在应用程序发行以后缺陷仍然可以得到有效的诊断。在日志文件的帮助下，客户发现的缺陷也可以得到重现和诊断，与只根据用户的输入和应用程序的用户界面上报出的错误信息来试图重现问题相比较，这种方法更加可靠。

应用程序的日志机制可以通过配置文件来进行配置，在配置文件中可以设置日志信息记录的级别。一个简单的日志系统应该包含两个级别：**错误**级别和**调试信息**级别。在应用程序的代码中每个日志条目应该对应一个级别，同时配置文件会指定日志的级别。一般来说，在发行模式中只把错误记录到日志中，而在调试模式中同时把错误和调试信息都记录到日志中。下面的伪代码说明了在这两种日志级别（错误和调试）下分别产生的日志条目：

```
write_debug_log_entry "connecting to database"
result = connect_to_database_server "dbserver"
if result == connect_failed then
```

```
        write_error_log_entry "failed to connect to database"
else
        write_debug_log_entry "connected to database"
```

上面的伪代码同时展示了调试和错误消息的日志。写入的调试信息反映出应用程序的流程和数据，而当遇到错误时（在这个例子中指的是连接数据库失败），写入的是错误信息。

当应用程序为调试模式时，日志文件中包含了所有有关流程、数据和错误的消息。而在发行模式下，则只有错误被写入日志文件。因为在发行模式下调试信息没有写入日志文件中，并且应用程序经过了几个完整的测试周期后错误已经不会频繁地发生了，所以在发行模式下日志文件不会变得很大。

为了保证日志机制不会给应用程序带来缺陷，在把应用程序变成产品之前必须要执行回归测试，这个回归测试用来验证系统的行为在完全启用日志机制（错误和调试信息都要记录），或者部分启用日志机制（只记录错误，或者什么都不记录）的情况下，并不会发生变化。

如果日志机制从应用程序的发行版本中删除了，但是因为错误仍然可能发生，所以系统必须具备其他的诊断能力。如果没有日志机制，那么诊断可以通过系统提供的机制来完成，例如：Windows 系统的事件日志（Event Log）或者 UNITX 系统的日志服务（syslog service）。

将调试模式和发行模式分离开，我们就能够根据实际情况确定诊断和调试应用程序的不同级别。当应用程序正在开发时，为了帮助定位问题和修正缺陷，应用程序最好具备完全的诊断日志记录能力和调试能力。当应用程序已经成为产品时，需要的诊断信息会少一些，但是为防止最终用户遇到错误，所以少量的诊断机制仍然必不可少。调试和发行版本、以及可配置的日志机制使得这种灵活性成为可能。

第 5 章　测试设计和测试文档

测试设计和测试文档是测试组的主要工作。第 1 章中已经讨论过，这些活动不应该等到软件版本交给测试组时才开始启动，而应该在第一条需求被认可和纳入基线的时候就启动。因为需求和系统设计会随着时间和系统开发迭代的进展不断完善，所以测试过程也需要不断改进，以便能够覆盖新增的和修改过的需求和系统功能。

结构化的方法有利于测试过程，这种方法定义了测试过程操作的级别——黑箱测试还是灰箱测试、测试过程文档的格式、选择的测试技术，例如：检查边界条件和探索性测试。另外测试脚本和其他测试软件开发工作也能从结构化的方法中受益，因为这种方法能够把数据和逻辑分开，并能实现公认的软件设计准则。

在需求和软件设计的评估过程中，通过了解构架及评审原型和现存的应用程序，测试组能够在保持预算和进度的情况下，把各种测试任务“分而治之”。测试工作中的“测试什么”、“如何测试”、“何时测试”和“由谁测试”等要素可以通过以下的活动来决定：确定测试策略、分解任务、使用特殊的技术以及用并行和迭代的方式划分测试过程的优先级。

第 20 条：分而治之

手中有了需求规格说明书、再加上一些设计知识，测试人员已经作好设计和开发测试过程的准备了。若一个项目的需求非常多，要开始测试过程的开发工作，看起来好像无从下手。那么，本条为您提供了一种有效的分解测试任务的方法。

在开始测试设计之前，必须先搞清将要执行的测试是处于什么测试阶段。针对可使用性测试、性能测试、安全性测试和其他测试等不同的测试阶段，以及功能性测试和非功能性测试，分别会有不同类型的测试。非功能性测试将在第 9 章中讨论。

为了理解用来定义当前测试项目的测试目标、测试范围和测试方法的环境和框架，设计测试过程的第一步是回顾测试计划（如果还对测试计划不熟悉的话）。测试计划还列出了可以利用的资源（例如：测试设备和工具）、必须遵循的测试方法和标准、以及测试时间表。如果测试计划还不够有效（有效的测试计划的定义请参见第 2 章），那么必须通过其他途径获得这些信息。

为了分解测试任务，需要回答下面几个问题：

- 应该测试什么？在测试计划阶段，应该确定哪些内容需要测试，哪些内容不需要测试，并且把它们作为测试范围的一部分写入文档。
- 何时开发测试过程？在第 3 条中我们建议：一旦得到需求，就应该马上开发测试过程。如果测试内容已经确定下来，那么测试工作的顺序也必须马上确定。哪些内容需要先测试？测试计划的制订者应该了解测试的优先级，也应该熟悉版本和发行的时间表。用于测试高优先级功能的测试过程应该优先开发，但是有一个例外：有些功能是完成其他功能的基础和前提，那么不管这些功能的优先级是否很高，都必须首先运行这些前提功能（关于划分功能优先级的更多信息，请参见第 8 条）。

  此外，为了帮助划分测试过程的优先级，应该使用风险分析（请参见第 7 条）。如果不可能测试所有内容，那么测试人员只能关注最重要的元素。风险分析提供了确定这些元素的一种方法。
- 测试过程应该如何设计？任何一种测试方案都不可能有效地覆盖系统的所有部分。针对系统的不同部分，其测试过程所使用的设计方法，对于测试这些特定部分来说必须是最恰当和最有效的。

为了设计最恰当和最有效的测试，我们必须考虑系统的各个组成部分以及这些部分的集成方式。例如：为了通过用户界面验证系统的功能性行为，最合适的测试过程应该基于功能需求陈述，同时还要包含执行各种路径和场景的测试用例。另一种方法是用典型的合法数据和非法数据从测试用户界面的每个域入手，来验证系统在不同输入情况下行为的正确性。这将包含一个执行路径序列，例如：当填写了一个域或者一个界面时产生了另一个也需要输入数据的 GUI 界面。

根据先前确定的测试策略，必须为 GUI 测试或者后端测试进行测试设计工作，或者需要为二者都进行测试设计工作。GUI 测试当然会与后端测试不同。

- 谁应该负责测试开发工作？一旦确定了必须测试的内容、何时开始测试工作和完成测试工作的方式，那么根据各类测试人员预定的角色和职责，确定执行各种测试任务的人员就容易了。关于测试人员角色和职责的讨论，请参见第 13 条。

除上面这几个问题外，在测试过程的设计和开发过程中，还会出现下列疑问和问题：

- 如果没有书面的需求文档怎么办？如果没有可以用来导出测试用例的需求，那么为了更好地了解正在开发的应用程序，可能有必要向客户代表、技术人员当面咨询，同时还有必要阅读任何可以利用的文档，例如：当前系统或者遗留系统的用户手册（如果有的话）。有关基于遗留系统的测试工作的风险的讨论，请参见第 5 条。如果存在遗留系统，那么可能会得到用于遗留系统开发过程的设计文档。但是，记住设计文档不一定能正确地描述用户需求。在某些情况下，为了获得需求可能需要使用逆向工程和探索性测试（有关探索性测试我们将在第 27 条中加以讨论）。
- 黑箱和灰箱测试。主要通过用户界面进行各种系统调用来检验系统的测试方法称为黑箱测试。例如：如果用户通过在用户界面上输入数据向应用程序的数据库添加了一条新记录，那么各种层（例如：数据库层、用户界面层和商务逻辑层）都要执行这一操作。而黑箱测试人员只能通过观察用户界面的输出来验证系统行为的正确性（有关黑箱测试和灰箱测试的更多内容，请参见第 4 章）。

  但是，我们还必须意识到，由于软件的错误报告机制存在缺陷，有时错误并没有在用户界面中报告出来。例如：应用程序在向数据库插入一条记录时失败了，但是并没有向用户界面报告错误，用户界面从底层代码收到了一个“假正确”的消息，并且在没有向用户报告任何错误消息的情况下继续执行。

因为用户界面或者黑箱测试并不能暴露所有缺陷，所以还必须运用灰箱测试[①]，灰箱测试要求测试设计人员掌握组成系统的基本组件方面的知识，有关灰箱测试的讨论请参见第 16 条。

- 需要开发自制测试工具吗？系统中的某些组件只能用自制测试工具来测试。例如：考虑一个可以处理成千上万种输入组合的计算引擎。这种测试要求的测试设计不同于用户界面或者黑箱测试：这种测试输入的组合和变化数量太多，根本不可能通过用户界面进行测试。出于时间和其他因素的考虑，我们需要开发一个自制测试工具直接对计算引擎进行测试，它可以输入大量的数值来验证输出结果。关于自制测试工具的进一步讨论请参见第 37 条。

- 应该使用什么类型的测试技术？根据要求的测试数据和输入数值，可以选择不同的功能性测试技术。**正交排列**就是一种验证大量数据组合的技术，使用正交排列测试选择测试参数组合的好处是：能够用最少的测试用例获得最大的测试覆盖率[②]。

  其他可能有助于大量减少输入数据组合的测试技术还有等价类划分和边界值分析法（请参见第 25 条）。综合使用这些技术能够缩小测试输入集合，但是仅是在平均意义上。

  另一个需要不同类型的测试方法的例子是使用**商用现货**（commercial off-the-shelf，COTS）工具。我们必须验证 COTS 能否和自主开发的代码正确地集成。例如：虽然一种 COTS 工具擅长执行 SQL 查询并且已经经过了验证，但是我们仍然需要执行测试来验证专用的 SQL 查询的内容的正确性。

- 应该使用记录/回放工具或者其他测试工具吗？如果说从战略上来说，对计算引擎和其他后端功能的测试可以使用自制测试工具，那么自动测试（记录/回放）工具已经证明适用于 GUI 回归测试。

  测试策略的定义需要确定程序的哪些部分应该使用自动的记录/回放工具来测试：哪些部分需要用自制测试工具来测试；哪些部分必须手动测试。在测试过程中应避免过分依赖自动测试工具的记录/回放功能，原因请参见第 36 条。

- 哪些测试应该自动进行？当测试工程师确定哪些测试工作使用自动测试、哪些应该手动执行时，他们应该仔细地分析应用程序。当某些测试自动执行比手动执行代价更大时，这种分析工作就可以避免不必要的费用。

  下面是一些可以自动执行的测试示例：

  - 需要多次执行的测试。相反，只执行一次的测试工作一般不值得使用自动测试。
  - 风险高的测试项目。低风险的元素不值得使用自动测试方法。考虑对一个用户极少使用的功能进行的测试。例如：正在测试的系统可能允许这一用户完成一些从商业角度看来意义不大的动作。因为这种功能的失败可能影响的用户数量很少，所以这项功能失败的风险很低。当设计测试时，针对这样的问题使用自动测试就没有意义，相反，应该关注高风险的部分。
  - 运行有规律的测试。这样的例子有**烟雾**（生成验证）测试、回归测试、平凡测试（包含许多简单和重复的步骤、并且必须反复执行的测试）。
  - 用手动测试不可能完成的或者代价过大的测试。例如：在性能测试和强度测试、内存泄漏测试或者路径覆盖测试中，需要模拟 1 000 个用户并发访问系统来验证计算引擎的输出，此时可能需要使用自动测试。
  - 用多种数值对同一动作的测试（数据驱动的测试）。
  - 在不同配置下运行的基线测试。
  - 结果可预测的测试。如果某项测试的输出不可预测，那么使用自动测试就不划算。
  - 对基本稳定的系统的测试。功能、实现和技术都不轻易发生变化。否则，维护的代价会超过自动测试带来的好处。

- 需要什么样的测试数据？一旦了解了测试任务的细节，我们就可以定义一组测试数据作为测试的输入。认真地选择测试数据是非常重要的，不正确的和过于简单的测试数据都可能会导致缺陷的漏识别或误识别，引起一些不必要的工作和测试覆盖率的降低（关于获得测试数据的更多内容，请参见第 10 条）。

回答本条中列出的这些问题有助于测试计划制订者分解测试任务。同时考虑可供支配的预算和时间表也是非常重要的。

① 请记住，即使综合使用黑箱测试和灰箱测试，但对于产生高质量的产品来说仍然是不够的。测试流程、测试过程、审查和走查，以及单元测试和集成测试都是有效测试工作的重要组成部分。只用黑箱测试和灰箱测试并不能发现软件程序中的所有缺陷。

② 更多关于正交排列的信息，请参见 Elfriede Dustin 的“Orthogonally Speaking”STQE Magazine 3：5（Sept.-Oct.2001），也可以参见网址 *http://www.effectivesoftwaretesting.com*。

第 21 条：使用测试过程模板和其他测试设计标准[①]

为了测试过程的可重复性、一致性和完备性，在适用的时候应该运用测试过程模板。如下图 21.1 是一个测试过程模板的例子：

测试过程 ID（TP ID）：

TP ID 遵从的命名规范来自需求中的用例，但是以 T（Test）打头而不是 R（Requirement）

测试名称：

对正在测试的内容的概述

执行日期：

测试工程师的名字：

测试过程的作者：

测试目标

——简述测试过程的意图：

相关的用例/需求编号

——列出在测试目标范围内被测试的所有用例的名称和编号：

前置条件/假设/依赖

——列出在测试过程能够执行之前必须满足的前置条件、假设和依赖。这可以用公告列表的方式来完成。通常情况下这里的前置条件和所用的用例的前置条件相同：

① 摘自 Elfriede Dustin 等人的 *Quality Web System: Performance, Security, and Usability*（Boston, Mass.: Addison-Wesley, 2002），52 一书中的“Web Engineering Using the RSI Approach”。

验证方法

功能性测试步骤					自动/手动（下划线选择）		
步骤	用户动作（输入）	预期结果	追踪日志信息	实际结果	需要的测试数据	通过/失败	用例的步骤号或者需求编号

图 21.1 测试过程模板

图 21.1 中展示的标准测试用例中有下面这些关键元素：

- 测试过程 ID：符合命名规范的测试过程 ID。
- 测试名称：提供对测试过程的描述。
- 执行日期：说明测试过程的执行时间。
- 测试工程师的名字：执行测试过程的测试工程师的名字。
- 测试过程的作者：确定测试过程的开发人员。
- 测试目标：概述测试过程的目标。
- 相关用例/需求编号：给出测试过程验证的需求的标识编号。
- 前置条件/假设/依赖：给出在测试过程运行之前必须满足的标准或者先决条件，例如：特殊的数据安装需求。如果当前的测试过程依赖于前一个测试过程，那么也应该填写这一项。如果同时执行两个测试过程会发生冲突，那么这一项也需要填写。注意一个用例的先决条件往往会成为相应测试过程的先决条件。
- 验证方法：此项可能包括：标准、自动测试还是手动测试、审查、示例和分析等内容。
- 用户动作（输入）：测试过程的目的和期望应该在这里明确地定义。这里以文档形式提供了创建一个测试所需的步骤。这一项中的条目和软件开发中的伪代码非常相似。此项内容会澄清和记录验证用例所需的测试步骤。
- 预期的结果：定义执行一个特定的测试过程所产生的预期结果。
- 跟踪日志信息：将后端组件的行为文档化。例如：在测试过程执行期间 Web 系

统组件产生了日志条目，这些条目详细地记录了这些组件执行的功能，和与这些组件进行交互的主要对象。在测试过程内可以获取这些日志条目。

- 实际结果：这一项可以有默认值，例如："和预期的结果一致"，如果测试过程的执行失败了，那么这一项记录实际的结果。
- 需要的测试数据：指的是支持测试过程执行所需的测试数据集合。更详细的介绍请参见第 26 条。

在许多测试工作中，经常是事后才考虑非功能性测试。然而，从测试生命周期的开始阶段就考虑非功能性测试工作是非常重要的。注意图 21.1 中描述的测试过程文档格式分成 5 个部分，其中只有一个部分用于功能性测试步骤，而其他 4 部分都是用于诸如安全性、性能和可伸缩性、兼容性以及可使用性等非功能性测试方面。有关非功能性测试需要考虑的更多事宜，请参见第 41 条。

测试设计的标准必须要形成文档、认真传达并付诸实施，这样每个相关人员才能遵守设计方针并且提供所要求的信息[①]。无论是开发手动测试过程还是自动测试过程，都需要遵循测试过程的创建标准或者原则。手动测试过程的设计标准应该包含一个范例，它说明了测试过程应该包含哪些细节。详细的程度可能只是要求简要地列出操作步骤，例如：

步骤 1：点击 "File（文件）" 菜单。

步骤 2：选择 "Open（打开）"。

步骤 3：在计算机的本地磁盘上选择一个目录。

根据正在测试的应用程序的规模以及给定的时间和预算限制，详细的测试过程的设计工作可能过于消耗时间，在这种情况下概括地描述测试过程可能就足够了。

测试过程的标准还应该包括对期望的测试结果文档化的一些原则。这样的标准应该包含以下问题：测试结果是否需要屏幕打印？是否需要另一个观察测试过程的执行和结果的人员来判定某项测试的结束？

自动测试过程的设计标准应该遵照最好的编码习惯，例如：模块化、松耦合、简明

① Elfriede Dustin 等人的 *Automated Software Testing*（Reading，Mass.：Addison-Wesley，1999），第 7.3.4 部分（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

的变量和函数的命名规范等等。这些做法一般和普通的软件开发工作相同。

在某些情况下，并不是所有的测试场景都需要按照模板的要求以相同的细致程度来进行文档化。在复杂系统（例如：财务系统）中的一些测试过程需要数以万计的用于计算的输入数据，此时可能需要采用不同的文档化方法，例如：使用数据表。

第 22 条：根据需求得到有效的测试用例[①]

对一个应用程序进行功能性测试的目的是为了发现与最终用户需求不一致的地方。功能性测试活动是大多数软件测试工作的中心；功能性测试阶段最重要的目标是评估系统的行为是否和需求指定的行为一致。

在功能性测试阶段设计测试过程的最佳做法是建立在功能性需求的基础上。此外，某些为功能性测试阶段创建的测试过程经过修改后，可以用于应用程序某些方面的非功能性测试，例如：性能测试、安全性测试和可使用性测试。

在第 1 章中我们已经讨论了获得可测试的、完整的和详细的需求的重要性。但是在实际工作中，测试人员拿到完美需求的情况是非常罕见的。为了创建有效的功能测试过程，测试人员必须理解应用程序的细节和实质。当这些细节和实质在需求文档中没有充分体现时（这是常见的情况），测试人员就必须对文档进行分析。

即使提供了详细的需求，完成一条需求的流程和一条需求对其他需求的依赖也往往并非一目了然，所以测试人员必须对系统进行研究才能对系统的行为有足够的了解，这样才能创建最高效的测试过程。

一般来说，测试人员必须分析每一部分应用程序的任何一种变化对应用程序其他部分产生的影响，例如：一个变量或者某个区域的变化对应用程序其他部分产生的影响。只验证这种变化本身还不够，例如：利用输入数据的组合和变化的一个子集，系统出现了要求的改变，并且这种改变的结果也已经正确地进行了保存，但是，有效测试还必须要覆盖到这种变化所影响的其他所有部分。

例如：考虑下面的需求陈述：

“系统必须保证用户能够在客户记录屏幕上编辑客户名称。”

客户名称字段和关于它的限制已经形成了文档（最好在数据字典中）。下面这些测试步骤用来验证这条需求是否已得到满足：

1. 通过在用户界面上点击客户记录屏幕，确认客户名称可以编辑，以此来验证系统

① Elfriede Dustin 等人的 *Automated Software Testing*（Reading，Mass.：Addison-Wesley，1994），第 1~114 页（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

允许用户编辑客户名称。

2. 测试客户名称允许的所有正面组合和负面组合，例如：所有等价类划分、所有的边界。在数据字典的限制范围内测试可能的组合和变化的一个子集。

3. 运行一个 SQL 查询语句来验证更新后的内容是否正确地保存在相应的表中。

上面的步骤组成了一个有效的基本测试，但是为了全面验证这条需求，还需要一些其他内容。

除了性能和其他非功能性问题以外，还需要澄清以下问题：“当客户名称发生变化时，系统还受到哪些其他影响？”是否有其他使用或者依赖客户名称的界面、功能或路径？如果答案是肯定的，那么下一步就有必要确定这些应用程序的其他部分受到了什么样的影响。下面举一些例子：

- 验证订货模块中的“创建订单”功能是否使用了修改后的客户名称。
- 增加一条订单记录并且验证新记录中保存的是新的客户名称。
- 执行任何其他使用新客户名称的功能，来验证是否对其他以前运转良好的功能产生了负面影响。

分析和测试必须持续进行，一直到所有受影响的部分都得到了确认和测试。

当需求文档不够详细时，虽然这种方法能够显著地提高软件功能性测试的有效性，但是它往往被忽略。事实上，大多数情况下需求文档都没有详细到明确地定义了需求和功能路径之间关系的地步，而这种关系才是测试过程开发的关键。因此，有效的测试过程经常不能只根据需求陈述来设计。

有效的测试设计所包含的测试过程很少出现重叠，相反它们在重复劳动最少的情况下提供了最大的测试覆盖率（虽然有时为了保证百分之百的测试覆盖率，重复并不能完全避免）。如果两位测试工程师用两个不同的测试过程来测试同一个功能，那么这种做法就不是高效的，除非是为了达到要求的功能路径覆盖（例如：两条路径在某些地方使用重复的步骤），否则这种做法没有必要。

在测试执行期间，为了确保测试执行的顺序是正确的、测试工作没有不必要的重复、测试人员没有使其他测试人员的测试成果白费、没有在重复劳动或者错误地发现缺陷方面浪费时间，分析测试流程是非常重要的。如果错误地发现了缺陷，那么不仅会浪费开发人员的研究时间和测试人员重复进行测试的时间，而且在不能正确追踪的情况下使缺陷度量遭到曲解。为了达到下面的目的，测试组应该重新评审测试计划和测试设计：

- 确定若干事务使用的相似动作或者事件的模式。一旦确定了这些信息，测试过程就应该使用模块化方式来开发，这样测试过程就可以通过复用和重组来执行各种功能路径，从而避免了建立测试工作中的重复劳动。
- 为了满足执行测试过程所需的前置条件（例如：数据库配置以及由控制流或工作流所产生的其他需求），必须确定特定事务的测试顺序或者序列。
- 创建一个测试过程关系矩阵，这个矩阵根据执行一个测试过程所需要的前置条件和后置条件组成了测试过程的流程。一个测试过程的关系图表示了各种测试过程之间的交互作用（例如：在测试设计期间建立的测试过程关系概要图），它们可以显著地改进测试工作。

上面的分析工作有助于测试组确定正确的测试设计和开发工作的顺序，这样，模块化的测试过程就可以正确地链接在一起，并且它们的执行顺序能够保证测试的连续性和有效性。

为了在开发时间表中尽早地安排对最重要功能的测试以及对它们进行更深入的测试，建立有效测试过程另外需要考虑的问题是确定和评审关键的和高风险的需求。在没有为高风险或者使用最频繁的功能建立测试过程的情况下，还投入大量精力去创建用来验证用户很少使用的功能的测试过程，这种做法就是浪费时间。功能性测试过程建立的优先级必须根据风险和使用频率的高低来划分。更多的内容请参见第 8 条。

要设计高效的测试用例，就需要对系统的变化、流程和场景有所了解。为了理解各种联系、流程和相互关系而逐页地通读需求文档是一件苦差事。在复杂的系统中理解原因与结果之间的联系需要分析思考和关注细节。只是设计和开发对系统的外在功能进行测试所需的高层测试用例还不够，在更深入的、灰箱层次上设计测试过程也同样重要。

第 23 条：把测试过程当作“动态”的文档

通常测试工程师努力开发出来的测试过程只能执行一两次，原因是需求、设计或实现的变化使它们成为废物。在必须完成测试任务的压力下，测试人员只能不停地工作，没有机会重新审视测试过程。他们希望只通过直觉和分析能力就能够覆盖到系统中最重要的功能，这种方法存在的问题是如果测试过程过时了，那么当初创建这些测试所付出的心血也就白费了，而更糟的是其他没有测试过程的手动测试工作根本不能再重复执行。为了避免测试过程成为“摆设”，把它们当成“动态的”而不是静态的文档是非常重要的。

大多数软件项目采用了迭代式和渐进式的开发方法，测试过程的开发也经常采用这种方法。在项目的开始阶段，开发一套全面而详细的测试过程，不仅不能达到预期的目标，而且也是不切实际的[①]，原因是在每个迭代中的开发生命周期都会带来变化。此外，时间上的限制通常也不允许测试组开发一套全面的测试过程。

在一个迭代式和渐进式的开发生命周期中，根据工作版本开发计划时间表，需要交付大量的软件工作版本，所以想让测试过程保持不变是很困难的。在第一个工作版本完成并交付测试时，我们希望对它的测试过程也能同期完成。当开发组在等待一个工作版本（系统的一个子集）的测试结果时，并不需要完成整个系统的完整的测试过程的开发工作。在这种情况下，测试过程必须根据当前工作版本的需求针对这个版本进行开发。

在一个项目中，所有的需求、设计细节和场景很早就形成了文档并且可供利用，这种情况是非常罕见的，有时甚至可能还没有形成概念；罗列了系统内所有功能路径和场景的文档也是非常罕见的。因此测试过程必须在开发流程的每个阶段中不断发展。在进行系统构架和设计阶段更多的细节才会浮出水面，甚至有时在实现阶段才会发现本该早就发现的问题。为了适应这些新信息，测试过程也必须增加或者修改。当需求发生变化时，测试人员必须了解这种变化，这样他们才能相应地调整测试过程。

在开发生命周期中，系统的功能可能会发生变化或者增强。这可能会影响一些测试过程，测试新功能之前必须要重新设计测试过程。当新功能交付使用时，必须包括该新功能的测试过程。

① 这里强调的是详细的和全面的测试过程。一旦获得需求，就应该开始测试过程的设计工作了，并且由于需求文档经常是动态的，所以必须认为测试过程也是动态的文档。

在发现和修正缺陷的时候，我们必须更新测试过程来反映系统的这种变化和增加。与增加新功能一样，缺陷的修正有时也会改变系统的运作方式。例如：有时测试过程可能要绕过系统中的一个严重错误。一旦错误改正了，那么必须要修改测试过程来适应这种变化，并且验证当前的功能实现正确。

测试人员自我感觉有充分的时间来开发一套全面的测试过程只是偶然现象，这里全面的测试过程指的是对于所有已知的需求都有相应的测试，并且完成了从需求到测试的可追溯矩阵。不管测试覆盖的范围看起来是多么的全面，但是即使是对于中等复杂度的系统，也经常能够找到从未考虑过的新测试场景。一个系统可能执行的场景是无限的。因此如果遇到了新的场景，那么必须要评估它、给它指定一个优先级并且添加到“动态”的测试过程集合中。

和任何动态的或者发展中的文档一样，测试过程也应该存储在一个版本控制系统中。

第 24 条：利用系统设计和系统原型

原型有多种用途。它们可以让用户预先看到某个功能的实现结果，并且让用户预先对应用程序有所体验。通过原型用户可以提出对正在开发的软件的反馈意见，这些意见可以用于改进原型并且使最终的产品对用户来说更满意。

原型有助于检测需求中的不一致性。在定义详细的需求时，如果它们在一个或者多个文档的许多不同的页中出现，或者一个以上的人员负责这些需求的开发和维护时，那么有时矛盾是很难发现的。虽然严格的步骤和人工检查能够使不完整性和矛盾降到最低程度，但是这些工作的成效在实际工作中受到了限制。原型的创建和设计有助于发现不完整性和不一致性，并且为开发定制工具奠定了基础，定制工具是为了在开发过程中及早发现其他额外的问题。

如果及早地为高风险和高复杂性的部分制作原型，那么我们就能够在整个过程的早期开发出正确的测试机制（例如：自制测试工具），并且不断加以改进。在生命周期的后期才开始努力地构建复杂的测试机制，一般不会成功。在开发生命周期中及早准备好原型，我们就可以把原型当作更接近实际的应用程序的模型，并且据此开始定义自制测试工具的概要和方法。

设计和原型有助于测试过程的完善，同时也是对需求进行添加和修改的基础。因此它们也是创建更好的、更详细的测试过程的基础。使用原型还能提供用来增强测试过程的其他信息。原型能够提供静态的需求文档无法提供的细节。

原型还有助于利用功能性测试工具的自动测试工作的设计：原型可以帮助测试人员确定哪些自动测试工具会遇到兼容性问题。这样使得在软件工作版本交付之前，我们有时间调查工作区和调整自动测试设计。有时原型能够及早地暴露自动测试工具的问题，从而为从工具提供商那里获得补丁预留了充足的时间。

第 25 条：设计测试用例场景时采用经过检验的测试技术

第 10 条讨论了预先规划测试数据的重要性。在测试设计阶段，我们肯定会发现能够用于测试过程输入的测试数据的组合和变化是无穷无尽的。因为对数据进行穷尽测试通常是不可能的，所以必须要利用有效的测试技术来减少测试用例和测试场景的数量，用最小的工作量达到最大的测试覆盖率。为了设计出这样的测试方案，了解可供利用的测试技术是非常重要的。

许多书籍中都讲到了各种白箱和黑箱测试技术[①]。虽然有关测试技术的文档已经把它们描述得非常详尽了，但是很少有测试工程师使用一种结构化的测试设计技术。在测试设计期间，我们有必要了解应用最广泛的应用程序测试技术。

经验表明：综合使用多种测试技术比使用单一的测试技术更有效。如果向系统专家咨询：在测试一个程序时如何确定一套测试用例是否够用，那么他们很可能会指出：在平均意义上，只有一半测试用例是有效的。当测试人员选择要执行的测试用例时，那么这种方法往往是不可靠的，其中包括测试覆盖率不足的风险。

在可能采用的许多测试技术中，可以减少测试用例集合的测试技术有：功能分析法、等价类划分、路径分析法、边界值分析法和正交排列测试。下面是这些方法的一些要点：

- 功能分析法在第 22 条已经详细讨论过了。这种方法需要根据功能规格说明书分析系统预期的行为，并且据此为系统的每个功能或者特性设计一个或者多个测试过程。如果需求规定系统具有功能 x，那么测试用例必须有足够多的手段证明系统提供了功能 x。第 22 条讨论了一种指导功能性测试的分析方法。在功能性测试方案已经确定，并且已经得出了覆盖应用程序的许多测试路径以后，必须运用其他测试技术来减少测试期间每一步功能步骤的输入数据的数量。
- 等价类划分用来确定产生相同结果的输入范围和初始条件。等价类划分与期望系统运行所处的各种情形的相同点和不同点有关。如果这些情形彼此之间是等价的或者本质上是相似的，那么只测试其中一种情形就足够了，而无需全部测试。虽然等价性通常在感觉上是不言自明的，但是在判断等价性时还是需要小心从事。

① 例如：Boris Beizer 的 *Software Testing Technique*（Hoboken，N.J.：John Wiley & Sons，1995）。

考虑下面的例子，它演示了在边界值上如何应用等价性：

一个密码域只允许 8 位数字，其他位数的数字都是非法的。由于大于边界值或者小于边界值的值都是同一个“等价类”的成员，所以没有必要使用同一个等价类的多个成员进行测试（例如：10 位数字、11 位数字和 12 位数字的密码），原因是它们产生的结果相同。

- 路径分析法用于测试产品内部的路径、结构和连接。它可以在两个层次上运用。一个是基于代码或者**白箱**测试，单元测试通常就是在这个层次上完成的。**单元测试**经常在代码开发完成或者修改后由作者或者程序员马上完成（更多关于单元测试的内容，请参见第 6 章。）

  路径分析也可以在第二个层次上运用：功能或者黑箱层次上。此时可能没有源代码，并且黑箱测试通常由系统测试人员或者用户验收测试人员完成。即使有源代码，在黑箱级别上的测试工作仍然按照预定的需求进行，并不评估在源代码中实现的细节。

- 边界值（BV）分析法是路径分析法的补充。BV 测试主要用于输入编辑逻辑的测试。**负面测试**是 BV 测试的变种，用来检查对非法数据的过滤处理工作是否正常。因为许多错误是在边界上发生的，所以确定边界条件应该是设计有效 BV 测试用例的一部分。

  边界把数据分为 3 组或 3 类：正确的、错误的和处于边界上的（也分别称为边界内、边界外、边界上）。

  例如：考虑一个检查输入值确保其大于 10 的应用程序。

  - 13 是一个边界内的值，因为它大于 10。
  - 5 是一个边界外的值，因为它小于 10。
  - 10 事实上是一个边界外的值，因为它不大于 10。

  除了边界内或边界上的值（例如：端点），BV 测试还会使用最大值/最小值、刚刚大于最大值/最小值、刚刚小于最大值/最小值、以及 0 或空这样一些输入值。例如：在定义对输入值只能是数字的测试时，我们应该考虑以下问题：

  - 本区域是否像定义的那样只能接受数字，还是也接受字母输入？
  - 输入字母后发生了什么事？系统接受它们吗？如果接受，那么系统报告错

误消息了吗？

- 如果输入域接受了应用程序或者某种特殊的技术保留的字符，那么发生了什么事，例如：在 Web 应用程序中输入了特殊字符‘&’会发生什么呢？用户输入这些保留的字符后是否引起了系统崩溃呢？

系统不应该允许用户输入边界以外的字符，或者也可以通过显示一条适当的出错消息来优雅地处理它们。

- 正交排列法能够用最小的测试过程集合获得最大的测试覆盖率。当可能的输入数据或者这些输入数据的组合数量很大时，由于不可能为每个可能的输入组合都创建测试用例，所以这种方法就特别有效。[①]

最好用例子说明正交排列的概念。假设有 3 个参数（A、B 和 C），每个参数有 3 个允许的取值（1、2 和 3）。测试 3 个参数所有的组合需要 27 个测试用例（3^3）。有必要全部测试这27个用例吗？如果测试人员怀疑某个错误依赖于全部3个参数的某一组值（例如：某个错误仅在 A=1，B=1 和 C=1 的情况下发生），那么答案是肯定的。但是错误的发生很可能只依赖于其中两个参数的取值。对于上面的情况，错误在使用 3 个测试用例中任何一个时都会发生：（A=1，B=1 和 C=1）；（A=1，B=1 和 C=2）；和（A=1，B=1 和 C=3）。因为在这个例子中参数 C 的取值似乎与这个特定错误的发生无关，所以 3 个测试用例中有一个就够了。基于这种假设，下面的排列列出了发现所有这样的问题所需要的 9 个测试用例：

Case#	A	B	C
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

① 更多关于正交排列的内容，请参见 Elfriede Dustin 的“Orthogonally Speaking”，STQE Magazine 3:5（Sept.-Oct.2001），也可以访问 *http://www.effectivesoftwaretesting.com*。

因为每对参数所有取值的组合只出现了一次，所以排列是正交的，也就是说，每个可能的参数对（A 和 B）、（B 和 C）和（C 和 A）都只出现了一次。由于只考虑了每对参数的组合，所以这个排列的强度是 2。因为没有出现所有的三维组合——例如：组合（A=1，B=2 和 C=3）就没有出现，所以它的强度不是 3。这里重要的是它覆盖了任何可能的两个参数的取值。

选择真正能代表接受值范围的具有代表性的数据样本要靠测试设计人员的判断。当各种取值之间关系错综复杂时，这是非常困难的。在这种情况下，测试人员可以考虑在已经生成的数据集合中加入可能取值的随机样本。

使用所有测试参数的所有排列来测试一个系统一般是不可能的、不切实际的和不划算的。因此综合运用边界值分析法和其他测试技术（例如：等价类划分和正交排列法）来得出一个适度的测试数据集合是非常重要的。

第 26 条：在测试过程中避免包含限制和详细的数据元素

在第 21 条中，我们已经讨论了使用测试过程模板来定义和文档化测试过程步骤的重要性。

为了保持测试过程模板的可维护性，应该用通用的方式来编写模板。在测试过程中包含详细的测试数据不是一种好做法，因为这样会带来不必要的重复：对于每一个测试数据场景，都必须重复定义所有的测试过程步骤，其中唯一的不同是数据输入和预期的结果。这样不必要的重复对于维护工作来说是灾难性的，例如，如果一个数据元素发生了变化，那么所有测试过程都必须对这一数据元素进行修改。假设所有测试过程都引用了一个 Web URL，如果这一 URL 发生了变化，那么必须改变所有的测试过程文档，这是一项既浪费时间又容易出错的工作。

为了使测试过程容易维护，应该把特定的测试数据输入和相应的预期输出结果单独放在一个场景文档中。测试数据场景信息可以保存在数据表或者数据库中，其中每行定义一个单独的测试用例。这些场景可以被测试过程引用，同时引用的还有反映具体数据的、用于每个测试过程的场景示例。在得到具体数据的时候，为了参考对数据的限制应该查阅数据字典。确保这些测试过程是可重用的以后，就可以避免维护困难的问题。造成测试过程维护困难的主要原因是：测试用例中出现了非常底层的细节以及把数据绑定到特定场景的测试过程的硬编码。

图 26.1 展示了一个数据表格形式的简单的测试数据场景的示例，它用于用户登录系统时对用户的 ID 和密码进行验证[①]。假定已经存在这样一个测试过程，它记录了登录所需的所有步骤。在这个测试过程中，这些步骤包含数据元素，也就是用户 ID 和密码。只要引用了数据元素，测试人员就应该查阅这个场景数据表。注意：在实际应用中，这个数据表应该包含“实际结果”这样一列（图 26.1 中没有显示），这样测试人员可以在不使用其他文档的情况下记录测试执行的结果。

为全部测试过程形成如图 26.1 描述的测试数据场景文档需要巨大的工作量。如果只是把主要的测试过程引用的测试数据保存在单独的数据表或数据库中，那么不仅使维护工作变得容易，还会使测试文档更有效。

① 注意这里没有列出所有可能的测试数据组合。

测试过程 ID （交叉引用测试数据关联的测试过程）			
用户 ID	预期结果	密码	预期结果
比允许的位数多一位——满足其他所有限制	拒绝	比允许的位数多一位	拒绝
比允许的位数少一位——满足其他所有限制	接受	比允许的位数少一位	接受
和允许的位数相同——满足其他所有限制	接受	和允许的位数相同	接受
使用允许字符的组合——满足其他所有限制	接受	使用允许字符的组合	接受
使用不允许的字符	拒绝	使用不允许的字符	拒绝
使用零输入	拒绝	使用零输入	拒绝
使用空输入	拒绝	使用空输入	拒绝

图 26.1 测试场景数据表的示例

如果数据元素没有包含在主要的测试过程中，是单独放在其他分开的位置，那么这样的测试过程文档对于自动测试尤其有益，并且还能成为其他测试过程的基础。无论是手动还是自动测试过程，都应该把数据元素和它们的测试脚本分开保存，就像第 8 章中讨论的那样，在脚本中应该使用变量而不要使用硬编码的数值。

第 27 条：运用探索性测试

第 22 条已经讨论过，在当今软件开发环境下，定义了所有关系和依赖的完整而详细的需求是非常罕见的。因此经常要求测试人员运用分析能力来确定应用程序的本质。有时为了获得高效的测试设计工作所需要的知识，我们需要进行探索性的测试工作。

对功能进行探索研究的原因是在测试开始之前不能很好地理解它们。如果对正在测试的系统缺乏了解（例如：根本没有功能说明书或者是没有正式的功能说明书，或者没有充分的时间进行详细设计和文档化测试过程），探索性的测试工作就非常有效。第 22 条已经说明了探索性的测试可以增强测试工作的效果。

探索性测试用迭代的方法确定测试条件。在探索性的测试工作早期发现的问题模式有助于指明以后测试工作的方向。例如：如果早期的研究表明系统的某个部分错误成堆，而另一个部分则相对错误较少，那么测试工作将根据这些回馈信息调整工作的重心。

如果在探索性测试中确定了某些部分容易出错，那么应该对它们进行评估来确定它们和软件中复杂部分的关系。探索性的测试与经过深思熟虑的、计划好的测试过程有所不同，它们并不能预先定义或者精确地按照一个计划来执行。

无论测试过程是基于最详细的需求，还是根本就没有需求，所有的测试工作都需要探索性的测试。当一个测试人员执行一个测试过程、发现了一个错误并且试图重现并分析它的时候，为了有助于确定问题的原因，必然会使用探索性测试。另外，在发现缺陷的过程中，需要探索和研究缺陷与应用程序其他部分之间的关系。这通常需要采取测试过程定义的步骤中所没有的措施，因为不是所有的测试场景都会写入文档（当然，这样做在经济上也是不可行的做法）。

另一个需要探索性测试的情况是测试人员必须确定某个特定功能的阈值（最大值和最小值）的时候。我们考虑下面的例子：

有一条需求表明："这个域必须允许创建选择列表项。"

假设数据字典定义了术语"选择列表项"和与之相关的约束和限制，那么测试人员的第一个问题应该是"允许的选择列表项的数量是多少？"这个例子中，在需求阶段忽略了这个问题（甚至在走查和检查期间也遗漏了这个问题。）

无论是对手头需求的研究还是向开发人员咨询，测试人员都没有得到答案。同时客户代表也没有明确的意见。因此测试人员就有必要设计和执行一个测试过程，来确定这个域允许的选择列表项的数量。随着测试人员不断地增加选择列表项，在某个时刻应用程序的性能开始下降，或者是不再允许加入新的选择列表项了。这样测试人员就可以确定阈值了。经过客户代表认可这个数字是可以接受的以后，测试人员应该把这个阈值公布给所有涉众。测试人员现在就可以继续进行原来的工作了——测试这个域（以及它的选择列表项）与应用程序之间的关系，具体请参见第 22 条中的描述。

由于探索性测试的指导思想是具体问题具体分析，所以它们是不能预先规划的。但是为了实现测试的可重复性、可重用性以及达到足够的测试覆盖率，在执行前或者执行后把探索性测试文档化，并且把它们加入回归测试（回归测试在第 39 条讨论）中绝对是一个好办法。

探索性测试使测试工作更加圆满：它增强了功能性测试（详细描述请参见第 22 条），但是探索性测试本身并不是非常强大的测试技术。使用探索性测试通常并不能确定或者测量测试覆盖率，并且会遗漏重要的功能路径。利用探索性测试获得的新知识，可以显著地增强功能性测试过程，就像第 23 条讨论的那样把测试过程当成"动态"的文档。

最强有力的测试工作不仅包括计划明确、定义明确的测试策略，而且还包括通过功能分析和其他测试技术获得的测试用例，这些技术包括：等价类划分、边界测试和正交排列测试（请参见第 25 条）。此外，还要通过深思熟虑的探索性测试来进一步加强测试。

在测试周期中，无论多长的时间都不足以对测试过程所有可能的测试场景、变化和组合进行文档化。为了保证在测试生命周期中能够覆盖最重要的问题，探索性测试是一种很有价值的测试技术。

第6章 单元测试

单元测试是通过检查单独的一部分代码来确定其功能是否正确的过程，单独的一部分代码也称为一个**组件**。在认为一个组件或者一段代码完成之前，几乎所有的开发人员都会进行某种程度的单元测试。单元测试和集成测试是交付高质量软件产品的有力保证，但是它们经常被忽略，或者实施得非常草率。

如果全面地完成了单元测试，那么后续的测试阶段就会更成功。但是，基于对问题的了解所进行的随意的和特殊的单元测试，与基于系统需求的可重复和结构化的单元测试之间是有区别的。

为了实现结构化的和可重复的单元测试，我们必须要在软件开发工作之前或者与此同时开发可执行的单元测试软件程序，这些测试程序会用各种必要的手段来检查代码，最终确定它提供的功能是否和需求一致、工作的方式是否和设计相符。一般把单元测试认为是开发项目的一部分，并且随着项目的进展单元测试需要根据需求和源代码的变化而不断更新。

单元测试保证了在集成测试和系统测试之前，软件至少已经具备了基本的功能。如果组件仍然处于开发阶段，那么此时发现缺陷对于时间和费用的节省都是非常显著的：不需要把缺陷放入缺陷追踪系统、也不需要重现缺陷和研究缺陷，相反，可以只由一个开发人员在软件组件交付之前对缺陷进行修正。

这里讨论的单元测试方法是适用于大多数软件项目的、简单而实用的测试方法。其他更复杂的单元测试方法（如路径覆盖分析法）可能对于风险非常高的系统是必要，但是，大多数项目既没有时间也没有预算来执行这种级别的单元测试。

这里提到的单元测试方法不同于**纯粹的单元测试**，纯粹的单元测试是一种把一个组件和它可能调用的所有外部组件分离开，使得当前单元完全依靠自己就能完成测试的方法。纯粹的单元测试要求所有的基本组件具有**存根**[①]，这样就提供了一个孤立的环境，这种方法非常浪费时间并且需要很高的维护代价。因为这里讨论的单元测试方法并不要

---

① **存根**是为子例程或者其他程序元素创建临时替身的方法。存根通过返回预先确定的（逻辑确定的）结果使得其他程序继续运行。用这种方法，程序元素可以在它们调用的所有例程的逻辑完成之前进行测试。

求分离基本组件和正在测试的组件，所以此类的单元测试产生了一些集成测试的效果，原因是正在测试的单元在测试中会调用更底层的组件。只有当这些底层组件已经经过了单元测试并且证明工作正常的情况下，这种单元测试方法才是可以接受的。如果底层组件的单元测试工作已经完成，那么导致单元测试失败的原因很可能在于正在测试的组件，而不是更底层的组件。

第 28 条：用结构化的开发方法来支持有效的单元测试

软件工程师和程序员必须为他们的工作成果的质量负责。但是有很多人认为他们只是代码的编制者，不必正式地为代码的测试工作负责，在他们的印象中，那应该是系统测试组的工作。事实上程序员必须要承担起按照规定的需求制造高质量的初始产品的责任。把包含大量缺陷的代码交给测试组经常会导致很长的改正周期，通过恰当地运用单元测试，代码中的错误会显著地减少。

虽然对代码的了解可能导致低效的单元测试[①]，但是如果组件是按照系统的需求文档来完成某项特定功能的，那么对需求的了解一般不会出现上面的情况。即使一个组件只完成了功能需求的一小部分，单元测试也能直接确定当前组件是否正确实现了指定的需求部分。

除了编写单元测试程序，开发人员还必须用其他工具来检查代码和组件，例如：用内存检查软件来发现内存泄漏。让若干开发人员一起检查源代码和进行单元测试可能会使单元测试过程的效果更好。

除了第一次编写单元测试代码，当代码发生变化时，组件的开发人员应该对单元测试进行更新。这些修改可能只是为了适应一般的改进和重构、为了修正一个缺陷、或者为了适应一条需求的变化。让负责编写代码的开发人员同时也负责对它的单元测试，这是使单元测试始终处于最新和最有效状态的一条有效途径。

根据单元测试的实现方式，如果单元测试程序是软件生成的一部分，那么单元测试可能会导致软件生成的中断——这可能是编译失败，也可能是生成可执行模块失败。例如：假设开发人员从一个组件的 C++接口中删除了一个函数或者一个方法。如果单元测试尚未更新，它仍然需要这个函数才能正确编译，那么编译将会失败。这样就会阻止继续生成系统的其他组件，除非更新单元测试。为了解决这个问题，开发人员必须调整单元测试程序代码来适应接口中某个方法的删除。这个例子表明当代码发生变化的时候，开发人员必须更新单元测试程序的重要性。

① 由于代码的作者对内部的工作机理和设计意图非常了解，所以他们可能会仅对可能没有问题的场景编写测试代码。如果让另一个人实施这段代码的单元测试工作，那么他会有不同的看法，可能会发现代码的原作者发现不了的问题。

除了编译通过，有些软件项目还要求成功执行单元测试，这才能认为软件生成是成功的。关于这个话题的讨论请参见第 30 条。

单元测试必须用能够测试存在疑问的代码或者组件的语言来编写。例如：如果开发人员为了解决某一特殊的问题或者为了满足某一特殊的需要而编写了一套纯 C++的类，那么用于检验这些类的单元测试程序很可能也必须用 C++编写。而对其他类型的代码（例如：COM 对象）的测试，则可以通过 Visual Basic 或者也可能是脚本（例如：VBScript，JScript 或者 Perl）来编写测试。

在一个大系统中，通常都采用模块化的方法来编写代码，具体来说就是：把功能划分成若干层次，每个层次负责系统的一个方面。例如：一个系统的实现可以包含下面几个层次：

- **数据库抽象** 对数据库操作的抽象，把和数据库的交互封装[①]进一组类或者组件（根据使用的语言）中，处于其他层次的代码通过这些类或者组件和数据库进行交互。
- **域对象** 一组表示系统的问题域中实体的类，例如："账目"或者"订单"，域对象一般和数据库层进行交互。域对象只包含少量的代码逻辑，并且可以用一个或者多个数据库表来表示。
- **商业处理** 它指的是实现商业功能的组件或者类，它们利用一个或者多个域对象来达成某种商业目的，例如："下订单"或者"创建客户账目"。
- **用户界面** 应用程序中用户可见的部分，它们提供了和系统进行交互的手段。这个层可以有多种实现方法。它可以是包含若干控件的窗口、Web 页面或者简单的命令行接口，以及其他的实现方法。用户界面一般处于系统各个层次的顶层。

上面列出的只是一个层次化实现的简单示例，但是它说明了从"自底向上"的层次角度来看如何将功能划分为各个层次。每个层次由若干代码模块组成，它们共同完成当前层次的功能。

在层次化系统的开发过程中，最有效的做法通常是把一个开发人员只分配到一个固

① 封装是指把代码和数据放到某一个地方，需要的时候可以引用它们。这样如果需要改动，那么只需要改动一处，而不需要在使用它们的每个地方都进行改动。

定的层次上工作，每个层通过书面形式的预定义**接口**来设计和其他层次的组件进行交互。在一个典型的交互中，用户通过用户界面选择执行某个动作，随后用户界面（UI）层调用商业处理（BP）层来完成动作。在内部，BP 层用域对象和其他逻辑来处理来自 UI 层的请求。在处理过程中，域对象会和数据库抽象层进行交互，以检索或者更新数据库中的信息。这种方法有许多突出的优点，其中包括：分离了各个层的工作、一个用来完成工作的详细定义的接口，还有增加了层次和代码的重用潜力。

根据每个层次的规模和组织方法，它们一般会包括一个或者多个单元测试程序。在上面的例子中，域对象单元测试程序在执行时就会像 BP 层操作它那样操作每个域对象。例如：下面的伪代码概述了一个对订单处理系统中域对象层的单元测试，这个系统有 3 类元素："客户"、"订单"和"帐目"。单元测试试图创建一个客户、一个订单和一个帐目，并且对它们进行操作：

```
// create a test customer, order, and item
try
{
        Customer.Create("Test Customer");
        Order.Create("Test Order 1");
Item.Create("Test Item 1");
        // add the item to customer
        Order.Add(Item);
        // add the order to customer
        Customer.Add(Order);
        // remove the order from the customer
        Customer.Remove(Order);
        // delete the customer, order, and item
        Item.Delete();
        Order.Delete();
        Customer.Delete();
    }
catch(Error)
{
        // unit test has failed since one of the operations
        // threw an error - return the error
        return Error;
```

```
}
```

相似地，BP 层也包含一个和用户界面工作方式相同的、检验它自身功能的单元测试，就像下面的伪代码：

```
// place an order
try
{
    OrderProcessingBP.PlaceOrder("New Customer",ItemList);
}
catch(Error)
{
    // unit test has failed since the operation
    // threw an error - return the error
    return Error;
}
```

如果不隔离底层组件，那么在某种程度上，对一个层次化系统的单元测试结果其实就是：正在测试的组件和与之相关的底层组件的集成测试，原因是为了完成使命，较高的层次会调用较低层次提供的服务。在前面的例子中，BP 组件使用了客户、订单和账目等域对象来实现下订单的功能。这样当单元测试执行下订单的代码时，它也间接地测试了客户、订单和账目等域对象。因为这种做法能够把单元测试失败的原因定位到发生错误的具体的层次上，所以这是我们希望的单元测试的效果。例如：假定域对象单元测试成功了，但是 BP 测试失败了，那么这很可能预示着 BP 逻辑本身有问题，或者是两个层次集成有问题。如果没有对域对象进行单元测试，那么我们就很难定位出现问题的层次。

前面已提到，我们应该根据系统的详细需求，把用例或者其他文档作为单元测试工作的指南。一项功能需求一般由系统的多个层次的实现来共同完成，所以为了满足这条需求，每个层次都会增加一些必要的代码，这是在设计阶段确定的。当需求的实现涉及多个层次时，为了保证每个层次都正确地实现了它们负责的需求部分，每个受影响层次中的组件必须通过针对它们的单元测试。

例如：前面提到的订单处理系统有一个名为“废止账目”的需求。为了满足这条需求，系统需要一个能够加载一项账目并且废止它的 BP 组件，这个组件还能检查是否有尚未完成的订单包含这个账目。再往下层，这会要求域对象和数据库层允许废止账目对象，实现方式也许是通过一个 discontinue（）方法，同时订单对象支持在订单中用 ID 查找账目。通过提供方法或者实现，所涉及的每个层次都参与了对这条需求的满足。

更好的方法是：针对每个需求的代表性测试包含在每个涉及的层的单元测试程序中，用来证明该层提供了满足需求的必要功能。在前面的例子中，每个层的单元测试应该包含一个“测试废止账目”的方法，它用来测试组件功能与需求相关的这样一些组件的废止账目的处理。

除了测试符合需求的情况，还应该测试**错误情况**（也称为**异常**）来确认组件能够优雅地处理错误的输入和其他异常条件。例如：某一条需求要求用户必须提供全名，并且按照数据字典的定义全名不能超过 30 个字符。单元测试除了应该用一个长度合法的名字进行测试，还应该用 31 个字符的名字进行测试，以便验证当前组件能否把输入限制到 30 个字符或者更少，其中边界是在数据字典中指定的。

第 29 条：在实现之前或者与实现同时开发单元测试

随着名为**极限编程**（extreme programming，XP）[①]的开发方式的流行，在实际软件开发之前开发单元测试程序的观念也被证明是有效的。在这种方法中，因为要用需求来指导单元测试的开发，所以它们必须在单元测试开发之前定义。在系统中一条需求经常隐含了多个单元测试，这些单元测试必须检查正在测试的组件是否和它需要实现的需求部分一致。关于在一个层次化的系统中，需求影响单元测试的方式，请参见第 28 条。

在实现软件组件之前开发单元测试有许多好处。首先，最明显的好处是单元测试强迫软件的开发方式能够满足每一条需求。当软件提供的功能都成功地通过了单元测试时，我们才能认为软件完成了，而在此之前还不能认为软件已完成；并且通过单元测试需求也得到了增强和检查。第二个好处是把开发人员的精力集中在解决某一专门的问题上，而不是开发一个也能满足需求的、巨大的解决方案。这经常导致更少的代码量和更直接的实现。第三个好处更加精妙，通过单元测试可以推测开发人员的实现目标（对照需求陈述的内容）。如果开发人员对需求的理解有问题，那么这种误解会在单元测试代码中有所反映。

为了正确地利用这种技术，大部分需求文档必须在开发之前准备好。这样做的原因是：在对某一特定功能进行详细的需求规格说明之前就开发软件可能是有危险的。需求应该定义得尽量详细一些，这样开发人员才能容易地确定需要的对象和功能[②]。开发人员可以根据需求文档为组件制订单元测试的总策略，其中包括对成功情况和失败情形的测试。

为了简化单元测试的开发工作，开发人员必须重视用**基于接口**的方法来实现组件。优秀的软件工程做法是围绕接口，而不是围绕组件内部的机理来设计软件。注意组件或者软件接口不同于用户界面，后者是通过图形化或者文本化的途径呈现给用户或者从用户取得信息。组件的接口通常由函数组成，这些函数可以被其他组件调用，并且能够根

① 总的来说，**极限编程**（XP）是一种软件开发方法。XP 改变了程序员协同工作的方式。XP 中一个主要的部分是程序员配对工作，并且测试工作是编码过程不可分割的一部分。更多的内容请参见 Kent Beck 的 *Extreme Programming Explained: Embrace Change*（Boston，Mass.：Addison-Wesley，2000）。

② 无论是从用户的角度还是从系统的角度，用于用例分析的 RSI（Requirements-Services-Interfaces，需求服务接口）方法是一种文档化需求的有效方法。更多关于需求定义和 RSI 的信息，请参见 Eltfiede Dustin 等人的 *Quality Web System*（Boston，Mass.：Addison-Wesley，2002）第 2 章。

据一组输入值完成特定的任务。如果函数的名称、输入和输出已经确定并且达成了一致，那么就可以开始实现组件了。首先设计组件和上层功能之间的接口，就使开发人员能够从高层开始设计组件，把注意力集中在组件和外部世界的交互上。因为组件的接口可以用存根模块来代替；所以它还有助于单元测试的开发。存根是指通过接口调用的每个函数可以实现成只是返回一个预定义的、硬编码的结果，而没有真正的内部逻辑。例如：考虑下面的接口：

```
class Order
{
        Create(orderName);
        Delete();
        AddItemToOrder(Item);
}
```

接口中出现的函数是根据需求描述“系统必须允许创建订单、删除订单和向订单添加项目”确定的。为了编写单元测试代码或者其他用途，接口可以像下面的示例这样用存根模块来代替。

```
Create(orderName)
{
        return true;
}
Delete()
{
        return true;
}
AddItemToOrder(Item)
{
        return true;
}
```

空函数或者接口的存根，并没有真正地执行任务，他们只是返回“true”。把一个组件用存根模块来代替的好处是根据接口就可以编写（如果需要的话，还可以编译）单元测试代码了，并且在这些函数真正实现之后这些代码仍然能够正确运转。

单元测试也能辅助接口本身的开发工作，因为它有助于认识组件在代码中实际的使

用方式，而不只是停留在设计文档中。单元测试的实现会精化组件和使组件易于使用，因为在按照接口实现代码的过程中容易发现接口设计的不足。

在实际工作中，总是在第一步就创建单元测试是很困难的。在某些情况下，单元测试的开发和软件实现必须（也是可以接受的）同时进行。采用这种方法有很多原因：从需求出发不能立即设计出最好的组件接口；或者由于突出的问题或其他非需求的因素（例如：时间限制）而没有全部完成需求。在这种情况下，还是应该尽量预先为组件定义完整的接口，并且针对接口中确定的部分开发单元测试。组件剩余部分和相关的单元测试可以随着组件开发工作而逐渐进行完善。

需求的更新应该按照如下步骤进行处理：首先，单元测试需要按照新需求进行修改，此时可能需要为组件接口新增一些函数、提供新的输入或者返回新的输出。为了配合单元测试的开发工作，接口更新时要提供新增部分的存根实现，这样使得单元测试能够进行。最后，组件自身通过更新来支持新的功能——每当开发人员完成了一个满足新需求的新组件时，必须同时提供更新后的单元测试。

第 30 条：使单元测试的执行成为生成过程的一部分

大多数大型软件系统是由源代码组成的，这些代码必须经过生成或**编译**[①]才能成为操作系统可以使用的可执行模块。一个系统通常包含许多可执行文件，为了完成使命，一个文件可能会调用其他文件。在大系统中，根据执行生成的硬件的性能，编译全部代码需要的时间可以长达从数小时一直到数天。

在许多开发环境中，每个开发人员也必须在他们自己的机器上生成软件的**本地版本**，然后添加和修改必要的代码来实现新的功能。需要编译的代码越多，每个开发人员每次花在生成系统的本地版本上的时间就越长。另外，如果系统的底层有缺陷，那么本地版本可能无法正常运转，这会让开发人员花很多时间调试本地版本。

第 29 条已经讨论过了，单元测试程序对于确保软件功能符合需求的要求是有价值的。单元测试程序还能在当前组件依赖的其他组件编译之前，验证它的最新版本的运转是否正常。这会节省生成的时间，同时也能使开发人员指出系统中失败的组件，并且立即开始检查那个组件。

第 28 条中已描述过，如果一个软件构架是层次化的，那么各个层次的生成是相互依赖的，较高的层次会向下调用较低的层次来达到目的和满足需求。编译一个层次化的系统，就需要准备好（经过编译并且可以使用）所有较低的层次，这样上一层次才能成功地编译和运行。这种自底向上的生成过程是非常普遍的，它能够使各种层得到重用，还能分清开发人员之间的职责。

如果针对每个层次中的组件的单元测试已经编写完成了，那么在一个层次生成之后，生成环境就能自动地执行单元测试程序。例如：单元测试的自动执行可以通过一个编译文件或者生成后的步骤来完成。如果单元测试的执行是成功的，那么这意味着在当前层次的组件中没有发现错误或者故障，可以继续生成下一个层次。但是如果单元测试失败了，那么生成过程就停留在失败的层次上。把生成成功建立在单元测试执行成功的基础之上，能够避免浪费大量的生成和调试时间，并且还能确保单元测试的真正执行。

单元测试代码在编写之后就不再定期地更新、维护和执行是相当普遍的现象。要求

① 在大多数开发环境中，术语**编译**描述了从一组源代码文件（例如：一组 C++文件）生成一个可执行模块的动作。

每次生成必须执行相应的单元测试，就避免了这些问题。但是这种做法是要付出代价的，当项目的进度表非常紧张（特别是在错误改正和测试周期内），快速纠正错误的压力非常大，有时要求的周期是几分钟。更新单元测试程序，使当前层次编译通过看起来是一件麻烦事，在特殊时刻简直就是浪费时间。但是我们必须认识到：磨刀不误砍柴功，用很短的时间来更新单元测试，就能在随后的调试和查找缺陷中节省许多时间。如果压力很大并且源代码修改频繁，那么这种节省尤其重要。

许多开发项目使用了**自动生成**来定期生成系统的正式版本，有时每天晚上都要生成，新版本包括了代码的最新改动。在自动生成情况下，如果组件不能正确编译，那么在源代码的问题被排除之前，生成过程就不会执行。当然，这种情况是不可避免的，因为只有在开发人员确定和改正了源代码中的语法错误以后，生成过程才能够进行。

把自动执行单元测试加到生成过程中，使得生成过程增加了另一种质量保证机制，不再是语法正确就能够通过编译。它保证了通过自动生成得到的产品是一个真正通过单元测试的系统。软件始终处于可测试的状态，并且由于单元测试已经发现了大多数错误，所以组件中不会再有重大的错误。

单元测试中的主要问题是不一致性。许多软件工程师都没有采用一种统一的和结构化的方法进行单元测试。标准化和流畅的单元测试减少了它们的开发时间，并且也避免了它们使用不同方法的现象。如果单元测试是生成过程的一部分，那么这一点尤其重要，因为如果所有人的行为是一致的，那么管理单元测试程序会更容易。例如：当遇到错误或者处理命令行参数时，单元测试的行为应该是可以预测的。如果在单元测试中使用了一些标准（例如：所有的单元测试程序在成功时都返回 0，失败时都返回 1），那么生成环境就能根据这个结果判断生成过程是否应该继续。如果没有使用标准，那么不同的开发人员可能会使用不同的返回值，这样就使情况变得复杂了。

一种完成这种标准化工作的方法是创建一个**单元测试框架**。这是一个处理命令行参数（如果有的话）和报告错误的系统。一般是在启动时用一个需要运行的测试列表对这个框架进行配置，随后框架会依次调用它们。例如：

```
Framework.AddTest(CreateOrderTest);
Framework.AddTest(CreateCustomerTest);
Framework.AddTest(CreateItemTest);
```

每个测试（也就是 CreateOrderTest、CreateCustomerTest 和 CreateItemTest）都是单元测试程序中的一个函数。框架通过依次调用这些函数来执行所有这些测试，同时处理它们报告的所有错误，最后返回整个单元测试的结果，通常是**通过**或者**失败**。框架会缩

短单元测试开发时间，因为只需要编写和维护每个层次中的单个测试，不需要它们都支持错误处理和其他执行逻辑。公共的单元测试函数只在框架中编写一次。每个单元测试程序只实现测试函数，所有其他功能（例如：错误处理和命令行处理）借用框架的代码。

因为单元测试程序和它们测试的源代码直接相关，所以它们应该驻留在与它们相关的源代码所在的工程或者工作区中。这使得单元测试能够和正在测试的组件一起获得有效的配置管理，避免了不同步的问题。由于单元测试非常依赖底层组件，所以如果它们不是当前层次的一部分，那么管理起来就会很困难。如果它们驻留在同一个工程或者工作区，那么在每次生成之后自动地执行它们也就会变得容易了。

但是，单元测试框架中可重用的部分应该存放在其他地方，需要时单元测试程序可以调用它们。

第 7 章　自动测试工具

自动测试工具能够增强测试工作的效率，但是要实现这个愿望，我们必须了解使用工具带来的问题，并且选择一个和当前系统环境兼容的工具。选择的工具要和当前的任务相匹配，并且能够帮助测试实现自动化，这些也是非常重要的。在开发生命周期中的各个阶段，有许多可供利用的测试工具，其中包括非常著名的记录/回放测试工具。

有时通过研究和评估得出的结论是：市场上没有合适的工具完全满足项目的需要。在这种情况下，我们有必要决定是否需要开发独特的解决方案。解决方案的形式可以是脚本、定制的工具或者只靠手动测试。

而还有另外一种情况是，有些商业工具可以用于测试工作，但是它们提供的功能比我们需要的多，导致费用也增加了许多。这是另一种需要开发定制工具的情况。但是我们需要认真分析，以确定自主生成工具的开发代价不会比购买工具更昂贵。

还有必须使用定制工具的一些情况，特别是当一个特殊的系统组件的风险很高，或者是专用的、不能用现成的工具测试的时候。

在某些情况下，特别是需要在自动测试工具方面大量投资的时候，我们必须要考虑整个组织的需要。无论是选择的工具还是自主开发的工具都必须在使用的技术上能够正常运转，同时还要确定在满足项目预算和时间限制情况下能否顺利地引入工具。我们必须建立评估的标准，在购买之前，工具的能力必须用这些标准进行验证。

在生命周期的早期检查这些问题将会避免在后期出现代价惨重的变更。

心理学。可使用性测试主要是通过手动的过程来确定一个系统界面的易使用性以及其他一些特征。但是有些自动化工具可以辅助完成这个过程，虽然它们绝不能替代对界面的人工确认[①]。

- 测试数据生成器。测试数据生成器通过自动生成测试数据来辅助测试过程。市场上有很多工具支持生成测试数据和填充数据库。无论测试数据是用于功能测试、数据驱动的负载测试，还是性能测试和强度测试，测试数据生成器都能够根据一组规则快速地填充数据库。

- 测试管理工具。测试管理工具支持对测试生命周期的所有方面进行计划、管理和分析。有些测试管理工具（例如：Rational 的 TestStudio）与需求管理、配置管理以及缺陷追踪工具集成在一起，这是为了简化对整个测试生命周期的管理。

- 网络测试工具。客户端-服务器或者 Web 环境的普及给测试工作带来了新的复杂度。测试工程师不能再像过去那样只是检验在单一系统上操作的单个的、封闭的应用程序。客户端-服务器构架包含 3 个独立的部分：服务器、客户端和网络。平台间的连接增加了出错的可能性。因此，测试过程必须要覆盖服务器和网络的性能、整个系统的性能和贯穿这 3 个组成部分的功能。利用网络测试工具，测试工程师能够在整个网络上对性能进行监控、测量、测试和诊断。

- GUI 测试工具（记录/回放工具）。市场上有很多自动的 GUI 测试工具。这些工具通常包含记录和回放功能，测试人员可以在不同的环境下创建（记录）、修改和运行（回放）自动化的测试。其中在用户界面控件或者“窗口部件”上（不是在位图上）记录 GUI 组件的工具最为常见。记录活动捕捉测试工程师输入的按键，自动地在后台用高级语言创建一个脚本。负责记录工作的是一个称为测试脚本的计算机程序。但是这种工具的记录/回放功能大约仅是其全部功能的 1/10。为了充分体现记录/回放工具的价值，工程师应该充分利用工具内置的脚本语言。关于为什么不能过于依赖记录/回放工具，请参见第 36 条。

记录好的脚本必须经过测试工程师的修改，才能成为一个可重用的和可维护的测试过程。脚本的运行结果成为测试的基线。然后脚本在软件的新版本上回放的结果可以和基线进行比较。

① Elfriede 等人的 *Quality Web System: Performance, Security, and Usability*（Boston, Mass.: Addison-Wesley, 2002）一书中的第 7.5 章“Usability”

具备记录功能的测试工具通常还带有一个比较器，它会自动地把实际结果与预期结出进行对比，并且将比较的结果记入日志。根据测试的比较类型，结果可以按像素、字符和属性进行比较，并且工具会自动指出预期结果和实际结果之间的区别。例如，Rationa的 Robot 和 Mercury 的 Winrunner 会在测试日志中把正确的结果记录成通过、并且在屏幕上用绿色显示，而错误的结果则用红色表示。

- 负载、性能和强度测试工具。性能测试工具使测试人员能够检查一个系统或者应用程序的响应时间和负载能力。此类工具可以在多台客户机上同时运行，这是为了测量客户端-服务器系统同时被多个用户访问情况下的响应时间。强度测试包括用高强度场景运行客户机来确定它们是否崩溃和什么时候崩溃。

- 专用工具。针对不同类型和构架的应用程序，需要对构架上的特殊部分进行专门测试。例如：对一个 Web 应用程序，就要求自动链接测试人员来验证是否有断开的链接，而对安全测试工作，就要求检查 Web 服务器的安全性问题。

第 31 条：了解各类测试支持工具[①]

虽然在测试工业中宣传最多的是功能性测试工具（也被称为“记录/回放工具”），但是了解其他可供利用的、用来支持测试生命周期的工具类型也是非常重要的。表 31.1 列出了可供利用的工具，它们分别适用于不同的测试阶段，本条将对它们进行简介。虽然大多数软件项目中还会使用其他工具（例如：缺陷跟踪工具和配置管理工具），但是表中只列出了自动测试专用的工具。

表 31.1 中列出的所有工具对改进测试生命周期都是有价值的。但是在组织做出购买某种工具的决定之前，必须要通过分析确定哪些工具（如果有的话）对特定的测试过程最有帮助。

表 31.1 测试工具

工具类型	描　述
测试过程生成器	根据需求/设计/对象模型生成测试过程
代码（测试）覆盖率分析器和代码测量器	确定未经测试的代码和支持动态测试
内存泄漏检测	用来确认应用程序是否正确地管理了它的内存资源
度量报告工具	读取源代码并显示度量信息，例如数据流、数据结构和控制流的复杂度。能够根据模块、操作数、操作符和代码行的数量提供代码规模的度量
可使用性测量工具	用户配置、任务分析、制作原型和用户走查
测试数据生成器	产生测试数据
测试管理工具	提供某些测试管理功能，例如：测试过程的文档化和存储，以及测试过程的可追踪性
网络测试工具	监视、测量、测试和诊断整个网络的性能
GUI 测试工具（记录/回放工具）	通过记录用户与在线系统之间的交互，使 GUI 测试自动化，这样它们可以被自动回放
负载、性能和强度测试工具	用于负载/性能/强度测试
专用工具	针对特殊的构架或技术进行专门测试的测试工具，例如：嵌入式系统。

① 改编自 Elfriede Dustin 等人的 *Automated Software Testing*（Reading，Mass.：Addison-Wesley，1999）第 3.2 部分（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

检查一个工具的能力和缺点所用的方法是：比较要解决的问题和预定解决方案、评估改进的潜力和进行成本-效益分析。在购买支持软件工程活动的工具之前，必须进行这样的评估，它们和第 34 条中描述的自动测试工具评估过程非常相似。

下面是各种类型的测试工具的要点。

- 测试过程生成器。一个需求管理工具可以和一个基于需求规格说明书的测试过程（用例）生成器联成一体。需求管理工具是用来捕捉需求信息的，随后这些信息会被测试过程生成器利用。生成器通过统计、计算或者探索式的方法创建测试过程。如果是用统计的方法生成测试过程，那么工具会按一个分布选择输入的结构和值，这个分布可能是统计上的随机分布，或者是和正在测试的软件的用户配置相匹配的分布。

  测试过程生成器最常使用的策略是动作、数据、逻辑、事件和状态驱动。这些策略用于检测不同种类的软件缺陷。当用探索式或者故障导向的方法来生成测试过程时，工具需要使用测试工程师提供的信息。测试工程师把以前频繁发现的故障输入到工具中。这样工具就具备了一定的知识，它利用对历史故障的了解来生成测试过程。

- 代码覆盖率分析器和代码测量器。测量结构上的覆盖率使开发组和测试组认识到测试和测试套件的有效性。此类工具能够量化设计的复杂度，测量限制设计所必需的集成测试的数量，有助于进行集成测试和测量还没有执行的集成测试的数量。有些工具还能用多种方式测量测试覆盖率，其中包含：代码段、分支和条件覆盖率。测量的方式是根据特定应用程序的具体情况而确定的。

  例如：整个测试套件可以通过代码覆盖率工具来测量分支覆盖率。然后再把没有覆盖到的分支和逻辑加到测试套件中去。

- 内存泄漏检测工具。此类工具用于特定的目的：验证应用程序是否正确地使用它的内存资源。这些工具确定一个应用程序是否释放了它申请的内存，并且还提供了运行时的错误检测。因为许多程序缺陷都和内存问题有关，其中包括性能问题，所以如果应用程序内存操作非常频繁，那么进行内存检测是值得的。

- 可使用性测试工具。可使用性工程学是一个范围很广的学科，其中包括：用户界面设计、图形设计、人类环境改造学、人性因素、民族学，还有工业和认知

第 33 条：了解自动测试工具对测试工作的影响[①]

自动测试工具只是解决方案的一部分，它们不能解决所有测试工作中的问题。自动测试工具决不能代替指导测试工作的分析技能，也不能代替手动测试。我们必须把自动测试看作是对手动测试过程的补充。

自动测试这一想法的出现使大家对它们抱有很高的期望。很多期望是对技术和自动化方面的渴望。有些人认为自动测试工具应该能够完成从测试规划到测试执行期间的所有工作，而且还不需要过多的人工干预。如果有这样的工具，那简直太好了，但是当今市场上还没有一种产品具备这样的能力。还有一些人错误地认为：无论处于什么样的环境参数（例如：操作系统或者编程语言），只用一个测试工具就可以支持所有的测试需求。但是环境因素的确会限制手动和自动测试工作。

有些人错误地认为自动测试工具会立即降低测试工作量和缩短测试时间。虽然自动测试被证明是有价值的，并且会对自动测试工具的投资产生回报，但是往往不能马上见效。由于存在不切实际的期望、进行了错误的实现或者是选错了工具，有时自动测试工作可能还会失败。

下面的列表纠正了一些软件业内部普遍存在的对自动测试的错误认识，并且提供了避免对自动测试期望过高的指导方针。

- 经常需要多个工具。一般来说，单个测试工具不能满足一个组织所有的自动测试需求，除非这个组织仅在一种操作系统上只开发一类产品。期望终归只是期望，我们必须明白：目前市场上，任何一个产品都不能与所有的操作系统和编程语言相兼容。

  这只是测试多种技术需要多种工具的一个原因。例如：有些工具在识别第三方的用户界面控件上有问题。为了实现一个应用程序，开发人员可以利用成百上千的第三方提供的插件。工具提供商可能会声称自己的工具和应用程序使用的主要编程语言兼容，但是我们不可能知道他们提供的工具能否和所有可能要使用的第三方插件兼容，除非提供商正式声明与它们兼容。开发一个和所有第三方插件都兼容的测试工具几乎是不可能的。

  如果已经有了一个测试工具，但是系统构架还没有开发，那么测试工程师可以给开发人员一个列表，它列出了测试工具支持的所有第三方控件。关于针对开发环境有效地评估工具和购买恰当工具方面的信息，请参见第 34 条。

- 测试工作量没有减少。在一个项目中引入自动测试工具最主要的动力可能就是希望减少测试工作量。但是经验表明在一个新项目中，运用自动测试需要一个学习的过程。节省测试工作量的效果并不会马上显现出来。第一次使用时，记录并编辑脚本使它们成为可维护的和可重复的测试，相比只是简单地手动执行其中的步骤需要花费更多的时间。但是有些测试经理和项目经理已经阅读了测试工具的文献，他们可能会迫不及待地想发挥自动测试工具的潜力，但是事实上这根本行不通。

  虽然看起来有些不可思议，但是在首次使用自动测试工具时，测试工作量增加的可能性很大。在一个新项目中引入自动测试工具会给测试工作带来更高的复杂度。测试工程师要精通自动测试工具的使用，需要有一个学习过程，此外，经理们千万不要忘记在一个项目中，任何工具都不能完全代替所有的手动测试。

- 测试时间没有缩短。对自动测试的另一个类似的误解是：在一个新项目中引入自动测试工具会立刻缩短测试时间。因为在首次引入自动测试工具时，测试工作量事实上会增加，所以也不能指望测试时间会缩短，相反必须对测试时间的增加有心理准备。毕竟，为了使用自动测试工具，必须要加强当前的测试过程或者重新开发和实现全新的测试过程。自动测试工具会增加测试覆盖率，但是它不会立刻缩短测试时间。

- 自动测试遵从软件开发生命周期。在首次引入自动测试时，为了确定应用程序中哪些部分可以进行自动测试，我们需要仔细地分析正在测试的应用程序。也需要仔细关注测试过程的设计和开发。自动测试工作可以看作有它自己的微型开发生命周期，也会有其他开发工作中出现的规划和协调问题。

- 需要相对稳定的应用程序。为了利用记录/回放工具进行有效的自动测试，应用程序必须要相当稳定。由于维护的原因，对软件中一直在变化的部分运用自动测试通常是不切实际的和不可能的。有时由于应用程序中的问题，自动测试不能完全执行，仅仅只是执行到中途就不能继续了。

- 不是所有的测试都应该实现自动测试。前面已经提到过，自动测试只是手动测试的补充，我们不能指望一个项目中的所有测试都能自动进行。分析确定哪些

① 改编自 Elfriede Dustin 等人的 *Automated Software Testing*（Reading，Mass.：Addison-Wesley，1999）的第 2 章（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

测试可以自动执行是非常重要的。有些测试是不可能自动执行的，例如：确认打印输出的结果。测试工程师可以自动地把一个文档发送给打印机，甚至会显示一条“打印成功”的消息，但是测试人员为了验证结果，他们必须走到打印机跟前确认文档确实打印下来了。(打印机可能处于离线状态或者缺纸状态。打印的结果可能位置不正确，或者数据元素可能在边缘被裁掉。)真正在纸上打印的内容必须要用眼睛来验证。第 5 章中讨论了哪些测试应该自动进行。

- 一个测试工具不会自动地测试所有可能的组合。一个被广泛接受的谬论是：对于任何给定的测试工作，测试工具都能够自动地验证 100%的需求。无论自动测试工具有多么强大的功能，但是因为当今应用程序中系统和用户动作可能的排列组合数量是无限的，所以我们没有时间测试所有的可能性。

  尽管从理论上来说是可能的，但是没有一个测试组有足够的时间和资源来 100%支持整个应用程序的所有测试工作自动完成。

- 使用工具的全部费用不只是购买费用。自动测试的实施代价不只包括工具的购买费用，另外还有培训费用、自动测试脚本的开发费用和维护费用。

- 即买即用的自动测试工具并不高效。许多工具的提供商为了销售它们的产品，会极力夸大工具的易用性。它们对使用新工具需要的学习过程视而不见。工具提供商指出他们的工具能够简单地捕捉（记录）测试工程师的按键，同时在后台创建一个脚本，然后可以简单地回放这个脚本。但是有效的自动测试不会这么简单。为了使测试脚本健壮、可复用和可维护，我们必须手动地修改测试工具记录的测试脚本，这需要对测试工具的脚本语言有所了解。

- 需要培训。如果测试人员没有接受任何新工具使用方面的培训，那么他们只能把测试工具束之高阁或者实现得非常低效。当在一个新项目中引入自动测试工具时，把工具的培训及早地纳入议事日程，并且把它作为重要的里程碑之一，这些是非常重要的。因为测试工作会贯穿系统的开发生命周期，所以对工具的培训应该在及早进行。这样使得测试阶段的前期和后期都能够应用自动测试工具，并且可以更早地发现和解决有关工具方面的问题。这样在项目的开始阶段，要测试的系统就已经具备了可测试性和自动化能力。

  有时在一个项目中工具的培训启动得太晚就没用了。例如：通常只用到了测试工具中的记录/回放部分，这样必须要反复建立脚本，从而浪费了大量的精力。及早培训工具的使用会杜绝大量这种浪费。

- 测试工具可能是插入式的。有些测试工具是插入式的，为了使自动测试工具正

常运转，可能需要向应用程序插入特殊的代码来与测试工具进行集成。开发工程师可能不愿意加入这些额外的代码。他们会担心这些代码会导致系统操作错误，或者需要复杂的调整才能使它们正常运转。

为了避免这种冲突，测试工程师应该和开发人员一起选择自动测试工具。如果某种工具要求增加额外的代码(并不是所有的工具都是这样)，那么开发人员需要事先知道此事。为了让开发人员了解工具不会产生问题，以便使他们放心，我们应该向他们提供其他已经使用这种工具的公司的反馈信息，还有厂商提供的相关文档。

由于测试钩子（专门用于测试而插入的代码）和使用仪器会带来缺陷，这些缺陷会干扰系统的正常运行，所以使用插入式的工具也有风险。为了确保没有因为使用工具而带来的缺陷，产品发行前的回归测试需要使用干净的代码。

- 测试工具可能是不可预测的。和所有其他的技术一样，测试工具也可能是不可预测的。例如：数据库损坏，基线不能恢复，或者工具的行为并不总是和预期一致。定位问题或者将损坏的库从备份中恢复往往需要花费很多时间。测试工具本身也是复杂的应用程序，所以它们也会有干扰测试工作的缺陷，并且也需要厂商提供补丁。在自动测试过程中，所有这些因素都会消耗额外的时间。

- 自动测试人员可能忘记测试目标。如果在一项测试工作中第一次使用一个新工具，那么在自动测试脚本上花费的时间通常比实际的测试工作还多。测试工程师会急于开发具体的自动测试脚本，从而忽视了真正的目标：测试应用程序。他们必须牢记编写自动测试脚本只是测试工作的一部分，并不能代替测试工作。并不是所有的测试工作都能够或者都应该实现自动化。我们在前面已经提到，评估哪些测试可以自动执行是非常重要的。当规划一个自动测试过程的时候，很重要的一点就是要做到责任明确。整个测试组把全部时间都花在开发自动脚本上是不必要的，只有部分测试工程师进行自动测试脚本的开发就够了。从事这项工作的工程师必须经过挑选，他们应该有从事软件开发的背景。

第 34 条：关注组织的需要

任何参与测试工程师用户组讨论[①]的人员都是会被问及下面的问题："市场上哪个测试工具是最好的？你推荐使用哪个测试工具？"

讨论组的成员会对这些问题给出许多不同的答案，就像在测试论坛上所发表的各种意见一样。通常最熟悉某种特殊工具的成员用户会说这种工具是最佳的解决方案。

但是这个常见问题的最佳答案是："它依赖于……"。到底哪种测试工具最佳，这取决于组织的需要和系统工程环境（也就是测试的方法论），它们在一定程度上决定了测试工作如何实现自动化。

在选择一个测试工具时，下面列出了值得考虑的最优实践：[②]

- 确定需要的测试生命周期工具类型。如果计划在整个组织范围内实现自动化，那么就需要倾听所有涉众的意见。它们希望把哪些内容实现自动化？例如：正在测试的系统的内部用户可能想把工具用于用户验收测试。只有确定了对自动测试的期望，那么才能早日达成这些愿望，正如第 33 条中所讨论的那样。

  有时，测试经理会受命寻找一种满足组织内大部分测试需求的工具（如果可能的话）。确定这样的工具需要考虑系统工程环境和其他组织的需要，还要列出一个工具评估标准清单。那么什么是系统工程环境呢？正在使用哪些应用程序开发技术呢？这些问题是工具选择过程的一部分，都应该是评估标准开发工作中所应该考虑的。

  而还有些时候，测试人员受命寻找的测试工具必须支持他们正在从事的特定项目，例如：一个 Web 项目需要 Web 测试工具，但是在总体上当前组织开发的是桌面或者客户端-服务器应用程序。最好不要把自动测试工具的选择标准限制在单个的项目上，因为虽然购买这样的工具可能对特定的项目是有益的投资，但是当这个项目结束之后这个测试工具便会无人问津。

① 两个这种讨论组的例子是网站 *http://www.qaforums.com* 和 Usernet 上的新闻组 *comp.software.testing*。

② 关于评估工具的更多信息，请参见 Elfriede Dustin 等人的 *Automated Software Testing*（Reading, Mass.: Addison-Wesley, 1999），67-103（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

确定了测试工具的类型之后，就应该进一步制定工具选择标准。例如：如果在整个组织都用一种工具，那么测试工程师必须要确认：它能够和尽可能多的操作系统、编程语言和组织其他方面的技术环境兼容。测试工程师必须要评审组织的系统工程环境，具体的方法是关注本章中提出的问题和注意事项，并且将发现的问题文档化。

- 确定各种系统构架。在调查系统工程环境期间，测试工程师必须确定应用程序在技术上的构架，其中包括整个组织或者一个特殊的项目应用最普遍的中间件、数据库和操作系统。测试工程师还要确定每个应用程序 GUI 的开发语言，以及要使用的第三方插件。除此以外，测试工程师还必须了解详细的构架设计，因为它们会影响性能需求。评审普遍的性能需求也是有益的，其中包括：强负载下的性能、复杂的安全性机制、以及有关系统可使用性和可靠性的度量。

  根据给定的测试工作，制定特殊的选择标准，这要依赖于正在测试的应用程序的系统需求。如果特定的测试工具不能适用于多个项目或者应用程序，那么可能有必要在组织内为正在测试的最重要的应用程序制订测试工具选择标准。

- 确定是否需要一种以上的工具。负责使用测试工具的测试组必须要确认：选择标准中包含了他们自己的期望。第 32 条中已提到，在一个组织内单个工具通常不会满足所有对测试工具的兴趣和需求。选择的工具应该至少满足更急迫的需求。随着自动测试工具工业的不断发展和成长，测试工具满足的需求会越来越多，并且单个测试工具提供大部分需要功能的可能性也会增大。最后，一种工具可能适合大多数 GUI 测试；而另一种工具可能会覆盖大多数性能测试；第三种工具则可能满足大多数可使用性测试的需要。但是现在，我们只能根据目前所处的测试阶段，考虑使用多种测试工具，并且满足对测试工具能力的期望。

  在给定的环境下，可能没有哪个工具能与所有的操作系统和编程语言都兼容。例如：如果需要测试的是 LINUX 客户机，它通过 3270 会话和 Java 客户端与一个 IBM 大型主机相连接，那么很难找到一个同时支持 LINUX，Java 和 3270 仿真终端的工具。而更常见的情况是需要若干工具测试各种不同的技术。在某些情况下，市场上没有和目标环境兼容的工具，此时就需要开发定制的自动测试解决方案。如果测试策略倾向于开发自主的测试工具，那么厂商提供的自动测试工具就会被排除（有关开发自制测试工具的讨论，请参见第 37 条）。

- 了解正在测试的应用程序管理数据的方式。测试组必须要了解目标应用程序管理数据的方式，并且确定自动测试工具如何支持对数据的验证。大多数应用程序最主要的目的是把数据变换成有意义的信息，并且用图形或者文本的方式把

它呈现给用户。测试组必须要了解应用程序是如何完成这种变换的，这样才能制定出验证变换正确性的测试策略。

- 评审帮助台的问题报告。当一个应用程序或者应用程序的一个版本正在测试时，为了研究由用户报告的应用程序中存在的最普遍的问题，测试组应该监视帮助台的问题报告。如果一个应用程序的新版本正在开发，那么测试组可以把测试工作的重点放在系统最频繁出现的问题上，其中包括确定一个支持此类测试的测试工具。这也应该成为选择标准的一部分。

- 了解从事的测试类型。任何一个项目都要经过很多种类型的测试阶段，因此有必要选择您感兴趣的测试类型。这一点应该在测试策略中定义，这样测试组才能明确需要的测试类型——回归测试、强度测试或者容量测试、以及可使用性测试等等。回答如下几个问题有助于确定使用的测试类型：要求工具提供的最重要的功能是什么？工具是否将主要用于强度测试？有些测试工具专门用于**源代码覆盖分析**。也就是，它们会确定在测试中必须验证的源代码路径。某个或者一组特殊的项目是否需要这样的功能？其他需要考虑的测试工具应用程序包括：支持处理自动化和从输入文件装载大量数据的测试工具。考虑测试组使用测试工具希望取得的效果。目的是什么？期望的功能是什么？

- 了解进度。在选择一个测试工具时，另一个需要关心的问题是它能否满足或者影响测试进度。在时间表的限制内，评审测试人员是否有足够的时间学习这种工具是非常重要的。当在时间表中没有足够的时间，那么建议不要引入自动测试工具，这是明智的选择。通过把某个测试工具的引入推迟到恰当的时机，测试组会避免匆忙引进的风险，或者避免为组织选择了错误的工具。无论是哪种情况，拿到的测试工具都不会是最好的，并且这些肯定不是最好的自动测试工具反而会妨碍最佳工具的引入。

- 了解预算。一旦确定了需要的工具类型，那么可能会出现片面追求最佳的倾向，但是，考虑可供支配的预算是非常重要的。评估最强大的工具可能需要数月的时间，结果只是发现它的价格超过了预算。此外，为使测试人员熟练使用工具，还需要培训预算。如果测试组中没有开发人员，则可能还需要额外的人力资源。

最重要的是测试人员必须记住，在市场上没有哪个工具在所有环境下都是最优的。所有的工具在不同的环境下都有他们的优点和缺点。到底哪种工具最佳，这依赖于系统工程环境以及组织特定的其他需求和标准。

第 35 条：在应用程序的原型上对工具进行测试

因为在软件开发工作中使用的技术太多了，所以验证备选的测试工具能够在正在开发的系统上使用是非常重要的。完成此项工作的最好方法是：让提供商在需要测试的应用程序上演示他们的工具。但是，这通常是不可能的，因为在测试工具评估阶段，需要测试的系统经常还不存在。

一个可以替代的可行的方法是：开发人员为评估一组测试工具创建一个系统原型。在评估过程中，可能会发现某些工具不能在经过挑选的开发技术上正常运行，例如：测试组不能正常地操作工具的记录/回放功能，或者其他重要的功能。

在系统开发的前期，原型的制作必须针对系统中那些关键的部分，这些部分的技术特点能够在某种程度上代表系统中使用的技术。完成了工具的兼容性测试对于 GUI 测试工具特别重要，因为这些工具可能很难在应用程序的用户界面中识别定制的控件。许多应用程序（尤其是在 Windows 平台上）中用到的日历（calendar）、表格（grid）和上下箭头（spin）控件在识别时就经常会遇到问题。这些控件或者窗口部件以前叫做 VBX，后来又叫 OCX，现在在 Windows 和 Web 界面中又称为 Active-X 控件。它们常常由第三方编写，很少有测试工具供应商能够兼容各种公司生产的成千上万种控件。

这种问题的一个例子是：考虑要测试一个用 Visual Basic 和 PowerBuilder 生成的应用程序，虽然测试工具可能兼容所有 Visual Basic 和 PowerBuilder 版本，但是如果应用程序使用了不兼容的第三方定制的控件，那么测试工具还是可能不能识别屏幕上的对象。特别是应用程序使用的第三方的表格，测试工具大都不能识别。测试工程师必须决定：是用工作区的方法对不能识别的部分使用自动测试，还是手动测试这些控件。

如果测试工程师从一开始就评估和选择与项目需求匹配的工具，那么就能解决这些不兼容的问题。

测试工具的评估人员必须要有合适的背景。一般来说要求一定的技术背景，这是为了确保测试工程师能够针对应用程序或者原型运用工具，这样才能保证测试工具能够在要求的技术上正常运转。另外，易用性、灵活性和第 34 条中讨论的其他特性也最好由技术人员来评估。

除了在应用程序的原型上观察测试工具的运行，没有其他替代办法。虽然提供商可

能会承诺工具和许多 Web 应用技术兼容，但是最好还是检验一下他们的说法。在软件工业中新技术不断涌现的情况下，没有一个提供商敢承诺他的工具能兼容所有的技术。每个组织必须努力保证选择的工具能满足需要。

第 8 章　自动测试：选择最好的实践

为了取得最好的效果，我们应该把自动软件测试当成一个软件开发项目。和应用软件开发一样，测试开发工作也需要根据详细的需求仔细地进行分析和设计。本章所讨论的内容是：使用经过挑选的、最好的自动测试实践。例如：在测试工作中如何才能最好地运用记录/回放技术，及与使用此类自动测试技术相关的缺陷。本章还讨论了其他自动测试的实践，例如：自动回归测试，自动软件生成和烟雾测试。

目前市场上的自动测试工具，并不总是适应所有环境或者所有测试工作的需要。无论是为了代替现货供应的测试解决方案，还是作为其补充，有些测试工作可能需要定制的测试工具。本章会讨论开发这样的工具的原因和时机，以及与这些工具有关的问题。

没有哪种自动框架解决方案对所有的自动需求都是最优的。自动测试框架必须要根据组织的测试环境和当前的任务进行剪裁。在开发一个自动测试程序时，选择正确的自动测试框架是非常重要的。

第 36 条：不要过分依赖记录/回放工具[①]

功能性测试工具（也称为记录/回放工具），只是无数可供利用的测试工具中的一种。记录/回放机制能够增强测试工作的效果，但是我们不应该只使用这一种自动测试方法。即使使用的是最好的记录/回放自动测试技术，它们还是有局限性。

记录/回放脚本在首次记录生成之后必须要进行修改。对功能性测试的修改主要集中在通过 GUI 进行验证的测试，也称为**黑箱**测试。为了取得最佳的效果，黑箱测试应该与针对内部组件的**灰箱**测试和针对代码的**白箱**测试结合起来使用。因此，除了记录/回放技术以外，最高效的自动测试方法还应该包括其他自动测试工具和技术。

记录/回放只是大多数自动测试工具的一个功能，它能够记录下用户和应用程序交互时的击键和鼠标的移动。击键和鼠标移动都被记录成了一个脚本，然后可以在测试执行期间“回放”。虽然这种方法对于特定的情形是有益的，但是只通过记录/回放直接创建的脚本有一些严重的局限和缺点：

- 硬编码的数值。记录/回放工具根据用户的交互动作来生成脚本，其中也包含了从用户界面输入或者接受的任何数据。让数值在脚本中“硬编码”会给以后的维护工作带来问题。如果应用程序的用户界面或者其他方面发生了变化，那么硬编码的数值会导致脚本非法。例如：在记录期间生成脚本时，输入值、窗口坐标、窗口标题和其他值可能也被记入生成的脚本代码中。如果在应用程序中，这些值中任何一个发生了变化，那么在测试执行期间这些固定的数值就成了罪魁祸首：测试脚本与应用程序的交互是错误的，或者彻底失败。另一个可能出问题的硬编码数值是日期戳。如果测试工程师在测试过程中记录了当前日期，那么几天之后再次运行这个脚本会导致失败，因为脚本中包含硬编码的数值不再和当前的日期匹配。
- 非模块化的、不易维护的脚本。测试工具产生的记录/回放脚本通常不是模块化的，维护起来非常困难。例如：可能许多测试过程都引用了一个 Web 应用程序中特殊的 URL，如果脚本中使用了硬编码的 URL，那么这个 URL 的改变会导致大量的脚本作废。在一个模块化的开发方法中，只有一个函数中引用，或者包装了这个 URL。随后各个使用它的脚本会调用这个函数，这样 URL 的任何变化只需要改动一处即可。
- 缺乏可重用性的标准。测试过程开发面临的最重要的课题之一是可重用性。如果测试创建专门的标准，明确地要求开发可复用的自动测试过程，那么就会极大地提高测试组的工作效率。如果把用户界面部分的测试封装进模块化的、可重用的脚本文件，供其他脚本调用，那么当用户界面经常不断地变化的时候，脚本的维护工作量就会大大降低。

  创建一个可重用的函数库时，最好把诸如数据读取、写入和确认、导航、逻辑以及错误检查功能分别归到不同的脚本文件中。自动测试开发的指导方针应该大量借用高效的软件开发工作所遵循的原则。遵循与测试工具生成脚本的开发语言最接近的开发语言的指导方针，这是一个很好的实践。例如：如果工具生成的脚本类似于 C 语言，那么应该遵从 C 语言的开发标准；如果工具生成的脚本类似于 BASIC 语言，那么则应该遵从 BASIC 语言的开发标准。

在自动测试开发中，只使用记录/回放方法生成的测试脚本是很难维护和重用的，这是明显的事实。虽然也有少数的情况，可以使用未经加工的脚本；但是对于大多数情况，如果不在记录之后修改脚本，那么在测试执行期间，测试工程师会由于正在测试的应用程序的变化而反复记录脚本。使用记录/回放工具可能带来的潜在收益，一定会被不断重建测试脚本的无奈所抵消。这会使测试人员产生很强的挫折感，并且会对自动测试工具感到不满。

为了避免未经加工的记录/回放脚本带来的问题，应该建立可复用的测试脚本的开发方针。未经加工的记录/回放脚本并不表示有效的自动测试，并且应该尽量避免只使用记录/回放功能。

遵从上述这此实践会提高记录/回放和自动测试的质量。但是，即使用本章中描述的最佳实践来实现记录/回放功能，记录/回放工具仍然不是测试自动化的唯一方法，认识到这一点非常重要。为了获得更大的测试覆盖和深度，有效的测试还需要使用其他的工具和技术。

① Elfriede Dustin 等人的 *Automated Software Testing*（Reading，Mass.：Addison-Wesley，1999）的 8.2.2 部分，“Reusable Test Procedures”（《自动化软件测试》（影印版）由清华大学出版社于 2003 年出版）。

第 37 条：必要时自制开发一个测试工具

自制测试工具用于对一个程序或者系统的核心组件进行自动测试。在这里，这个术语指的是用于测试应用程序基本逻辑（后端）的自主开发的代码。现成的自动测试工具有它们的局限性，有些局限性在第 32 条中已经讨论过了。此外，通过用户界面的自动测试通常会由于要检验成千上万的测试用例而变得太慢，特别是变化频繁的用户界面更是自动测试的障碍。

为了去除自动测试工具的局限性和对核心组件进行更深入的测试，可以自制开发一个测试工具。这种定制的测试工具一般用健壮的程序语言编写，例如：独立的 C++或者 VB 程序，定制的测试工具一般比自动测试工具生成的脚本运行的速度更快，也更灵活，因为这些脚本受限于测试工具的特定环境。

我们举一个适合用定制测试工具来测试任务的例子，假设一个应用程序的用途是根据用户提供的信息进行计算，并且把计算的结果生成报告。计算过程可能是复杂的，并且可能对各种输入参数的不同组合是敏感的。这可能会有数百万种潜在的变化，这些变化会产生不同的结果，因此，对计算过程进行全面的测试才能保证计算的正确性。

手工开发和验证大量的计算性测试用例是非常浪费时间的。在大多数情况下，通过界面执行大量的测试用例也是非常缓慢的。此时一个更高效的方法是自制开发一个测试工具，它会直接针对应用程序的代码（一般是直接针对用户界面层之下的核心组件）执行测试用例。

另一种使用自制测试工具的方法是：对照遗留组件或者系统来比较新组件。两个系统通常使用的数据存储格式是不同的，用不同的技术实现的用户界面也是不同的。此时，为了在两个系统上运行相同的测试用例并且生成比较报告，自动测试工具需要一种特殊的机制来复制自动测试脚本。在最坏的情况下，单个测试工具不能同时兼容两个系统，此时两套测试脚本必须用两种不同的自动测试工具来开发。一个更好的替代方案是自制生成一个定制的、自动测试工具，它把两个系统之间的差异封装进独立的模块，这样我们就能够设计出同时适用于两套系统的测试。自制的自动测试工具可以把遗留系统产生的测试结果作为基线，并且通过比较两套结果和输出它们之间的差异来自动地验证新系统产生的结果。

达到上述目的的一种方法是利用自制测试工具适配器模式。**自制测试工具适配器**是一个模块，它通过转换或者“改造”正在测试的每个系统使之和自制测试工具兼容，这样自制测试工具可以通过适配器在系统中执行预定义的测试用例，并且把结果存储为相互之间可以自动比较的标准格式。对于每个处于测试的系统，开发出来的适配器（例如：DLL 或者 COM 对象），必须能够直接和系统进行交互和针对系统执行测试用例。用一个自制测试工具测试两个系统需要两个不同的适配器并独立调用两次自制测试工具，每个系统调用一次。第一次调用产生的结果应该保存起来，随后将这个结果和第二次调用产生的结果进行比较。图 37.1 描述了一个能够针对遗留系统和新系统执行测试用例的定制测试工具。

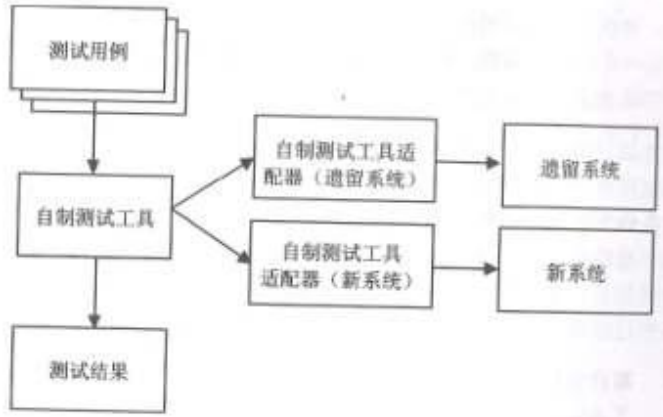

图 37.1 自制测试工具的基本构架

通过为每个系统使用不同的适配器，相同的测试用例可以用于多个系统。针对遗留系统的适配器用来产生一组基线结果，这个结果用于和新系统的结果进行比较。

为了完成它们的任务，自制的测试工具适配器首先获得一组测试用例，然后按照顺序执行这些用例，从而直接测试每个系统的逻辑，绕过了用户界面。绕过用户界面使性能得到了优化，使得测试用例的吞吐量最大化。它还具有更高的稳定性。如果自制测试工具依赖于用户界面，那么用户界面的任何变化（在开发生命周期中，用户界面经常会经过多次修改）都可能导致自制测试工具对缺陷的漏识别。检查这样的结果会浪费大量宝贵的时间。

每个测试用例的执行结果被存储在一个或者多个结果文件中，存储格式是相同的（例如：XML 文件），与正在测试的系统无关。保存结果文件是为了与随后运行的测试产生的结果进行比较。比较可以由一个定制生成的结果比较工具来完成，这个工具按照一定

的规则读取和评估结果文件，并且输出发现的所有错误或者差异。如果我们把结果格式化，那么也可以通过标准的“file diff（比较文件差异）”工具对它们进行比较。

与所有测试类型一样，自制测试工具使用的测试用例可能也是相当复杂的，特别是当自制测试工具所测试的组件是用于数学计算或者科学计算时尤其如此。因为有时计算过程可能涉及各种参数的数百万种组合，所以可能的测试用例数量也会以百万计。在时间和预算都有限的情况下，测试所有可能的用例是不现实的，但是利用自制测试工具执行数千个测试用例还是可行的。

随着建立和执行的测试用例增加到上千种，测试用例的管理就成了一个重要问题。下面的内容是一个通用的策略，根据这些策略可以开发和管理定制测试工具所用的测试用例。这个策略也适用于其他测试工作。

- 创建测试用例。定制测试工具所用的测试用例的开发方法和手动测试一样，必须运用各种**测试技术**。测试技术是一种选择测试条件的正规方法，它选择的测试条件发现缺陷的概率很高。测试技术用严格和系统的方法来获得测试条件，而不是瞎猜应该选择什么样的测试用例。很多测试方面的书籍都讲述了诸如等价类划分、边界值分析、因果图和其他一些测试技术。这些测试技术在第 25 条中已经详细讨论过了，这里只是简单地进行回顾：
  - **等价类划分** 确定期望产生相同输出的输入范围和初始条件。等价性依赖于不同系统的工作环境之间的共性和差异。
  - **边界值测试** 主要用于测试输入编辑逻辑。边界条件应该总是测试场景的一部分，因为许多缺陷都是发生在边界上。边界把数据分成 3 组或 3 类：正确的、错误的和在边界上的（边界内、边界外和边界上）数据。
  - **因果图** 提供了逻辑条件和相应动作的简明表示，它是一种图形化的表示方法，图中原因在左边、结果在右边。
  - **正交排列测试法** 选择的测试参数组合的特点是：用最少的测试用例提供最大的测试覆盖率。正交排列测试的测试用例可以用自动化的方式生成。
- 建立公共起点。所有测试工作必须从一个定义良好的起点开始，自制测试工具每次执行的起点是相同的。这通常意味着：在每次测试中，每个系统使用的数据必须是相同的，这样才能正确地比较结果。当每个模块化的测试组件被重新运行时，为了运行下一个测试组件，它应该能够把应用程序的状态恢复到初始状态。如果做不到这一点，那么因为假设的起点是错误的，所以第二个测试组件运行总是会失败。
- 管理测试结果。测试脚本为每套所执行的事务产生结果。这些结果一般会写入一个文件。虽然对于大多数的情况，一个文件就足够了，但是一个测试脚本也可以根据测试人员的意愿把结果写入许多文件。当运行一系列的测试用例时，创建了若干测试结果文件。一旦纳入了基线，那么每次执行任何给定的测试用例都应该产生相同的结果，测试结果文件可以用简单的文件比较例程或者用定制开发的测试结果比较工具来直接进行比较。为了确定、文档化并跟踪引起这些差异的缺陷，直至问题解决，在比较过程中发现的任何差异必须要进行评估。

定制的测试工具可以提供比自动测试工具脚本更强有力的测试手段。虽然生成一个定制的测试工具可能是很耗时间的，但是它也具有很多优点，例如：对应用程序的敏感部分可以有更深的覆盖、具有比较两个应用程序的能力，而这些是单个现成的测试工具无法完成的。

第 38 条：使用经过考验的测试脚本开发技术

测试脚本的开发本身就可以当作一个软件开发项目。为了得到高效的、可维护的和可重用的测试脚本，我们应该使用经过考验的技术。这可能要求部分测试组成员做更多的工作，但是最终测试工作会从一个更有效的自动测试产品中受益。

在使用功能性测试工具开发测试脚本时，我们应该考虑下面这些技术：

- 数据驱动框架。**数据驱动**意味着数值是从电子数据表或者工具提供的数据池中读取，而不是通过硬编码写进测试过程脚本中。第 37 条中已经提到过，记录/回放工具生成的脚本中包含了硬编码的数值。因为硬编码的数值常常会使得测试过程不能重用，所以我们一般推荐测试工程师对这些脚本进行编辑。应该用变量来代替硬编码的数值，并且在可能的情况下从外部数据源读取这些数据，例如：ASCII 文本文件、电子数据表或者数据库。

  测试组确定各种测试过程使用的数据场景、数据元素、数值的名字或者数据变量名。在测试过程脚本通过记录/回放工具创建之后，向脚本添加的关键数据是变量定义和从中读取数值的数据源定义（文件、电子数据表及数据库的名字和位置）。

  把输入数据外置还能降低数据维护的工作量和增加灵活性。如果需要用不同的数据集执行一个测试过程，那么可以根据需要更新或修改数据源。

- 模块化的脚本开发。一个通用的软件开发技术是把代码分成逻辑上的**模块**，每个模块完成一项独立的任务，这种技术可以增加源代码的可维护性和可读性。然后这些模块可以在脚本的多个地方被调用，这样它们的代码就不必在不同脚本中每次都重写或者修改。当一个模块发生变化的时候，这种技术显得格外重要，原因是只需要改动一处，降低了维护的代价。

- 模块化的用户界面导航。模块化的测试脚本的一个特别重要的功能是应用程序用户界面的导航。为了增加测试脚本的可重用性，测试工程师应该在脚本中使用导航的方法，这样可以将来自应用程序的用户界面的影响最小化。在测试重用库内部，导航函数应该隐藏在一个独立的模块中。测试需求应该说明：用一个测试工具记录器开发的测试过程是否通过使用 TAB 键、快捷键（热键）或者鼠标点击来对应用程序进行导航；测试过程是否需要捕捉 x-y 坐标（不推荐）

或者特殊对象的对象名，例如：窗口名或者 Web 页面上的 HTML 对象的名字。

所有这些选择都可能受应用程序设计变化的影响。例如：如果测试过程脚本在开发中利用 TAB 键完成了对象间的移动，那么生成的测试脚本是依赖于 TAB 顺序的。如果 TAB 顺序发生了变化，那么可能需要改正所有的脚本，这样的修改可能是经常性的工作。如果脚本中使用了加速键，但是后来它们在应用程序中发生了变化，那么脚本同样需要修改或者替代。如果用户界面上的控件被删除了或者换成另一种类型的控件，那么回放鼠标点击动作就可能陷于混乱。

无论使用何种导航方法，引入一种新的控件将会给大多数脚本造成负面影响[①]。此时把与控件的交互模块化是有益的。另一种避免此类问题的办法是：通过对象名或者特殊的开发工具分配的其他唯一标识来确定控件和进行导航。利用这种方法，无论焦点从一个控件移到屏幕上的任何位置，脚本也不受影响，因为焦点的导航是依赖于控制的名字而不是位置。利用这种方法，记录的脚本受代码修改的影响更小、复用性更强。

与此相似，在可能的情况下，测试过程应该利用 Windows 的对象名。比起用标题标识窗口，对象名保持稳定的可能性更大。根据脚本设计的方法，窗口信息或者对象信息可以保存在一个电子数据表中。导航功能可以通过一个驱动程序来完成，这个驱动程序从上述电子数据表中取得各种导航信息块。有些工具把对象信息保存在一个特殊的 GUI Map 文件或 GUI Frame 文件中。在这种情况下，就没必要再把信息放入电子数据表中了。

- 可重用的函数库。一般来说，针对同一个产品，测试脚本执行许多相似的动作，甚至对于同一类型的不同产品（例如：Web 应用程序）也是如此。我们应该把这些公共的动作剥离出来，形成一个共享的脚本库，供组织内部所有测试人员使用。这样可以极大地提高测试工作的效率。

- 现成的库。Internet 上提供了专用于某些测试工具的现成函数库。例如：使用 WinRunner 测试工具的一组测试工程师创建了一个可重用的脚本库，并且把它

① 但是工具本身可以“更智能”。例如：Rational Software 的 RobotJ（请参见 *http://www.rational.com/products/tstudio/robot.jsp*，retrieved Sept.8，2002）。Rational 的新技术利用了模式匹配。为了在一个交互式的应用程序中确认数据的变化，测试人员可以设置一个可接受的数据范围，这样在测试回放过程中，会比其他相对简单的软件工具出现的问题少。

放到 Web 上供所有测试工程师使用[①]。库中包含了测试脚本的模板、测试脚本语言的扩展和使用 WinRunner 进行 GUI 测试的标准测试过程，WinRunner 是 Mercury Interactive 公司的产品。表 38.1 列出了用于 WinRunner 的可重用的函数示例。

这个库包含了 3 个基本部分：脚本模板、函数和 GUI 检查。这 3 部分的设计目标都是为了简化测试开发人员的工作，以及使测试结果标准化。脚本模板为测试工程师提供了用于编写他们自己脚本的框架。可以先拷贝脚本函数，然后开发成完整的测试脚本或函数集。通过启动相应的安装脚本，可以将 GUI 检查和 WinRunner 环境进行集成，然后就可以与 WinRunner 内置的 GUI 检查有同样的使用方法。

表 38.1 WinRunner 脚本的可复用的函数

模 块	描 述
Array	和数组有关的函数
File	用于文件访问的函数
General	各种杂类函数
Geo	检查对象位置对齐的函数
List	简化对列表对象访问的函数
Menu	处理菜单的函数
Standard	标准测试
StdClass	对象类的标准测试，例如：单选按钮
String	处理字符串的函数
Tab	访问 tab 卡控件的函数
Tooltips	访问工具提示和工具条的函数

- 版本控制。虽然本质上版本控制不是一种脚本开发技术，但是它是任何软件项目的重要组成部分。为了防止不同测试工程师的并发修改和维护每个文件的修改历史，所有的测试脚本应该存储在一个版本控制工具中，例如：Microsoft Visual SourceSafe。如果没有版本控制工具，那么无法控制不同的脚本开发人员

① Tilo Linz and Matthias Daigl, "How to Automate Testing of Graphical User Interface" (Mohrendorf, Germany: imbus GmbH, n.d., retrieved Sept.5, 2002 from http://www.imbus.de/engl/forschung/pie24306/gui/ aquis-full_paper-1.3.html [text]and http://www.imbus.de/engl/forschung/pie24306/gui/gui_folien.html [accompany slides])。

修改同一个文件的情况，最后迫使开发人员手工合并他们的修改内容，而这项工作是很容易出错的。此外，当使用版本控制工具时，我们可以通过加标签来将脚本进行分组，这样追踪哪些脚本应该和哪些软件版本配合工作就变得更容易了，而且在将来任何时候都可以方便地取得它们。

第 39 条：尽量使回归测试自动化

回归测试确定的是：在修改先前的错误或者向应用程序添加新功能时，是否引入了新的错误，这些错误影响以前运行正常的功能。如果升级过程破坏了用户已经成功使用了好几年的功能，那么为尊贵的客户提供的大量崭新和激动人心的功能也就没有意义了。回归测试应该发现这些新引入的缺陷。尽管我们已经认识到了回归测试的重要性，但是，在计划测试活动时，回归测试常常是最不受重视的一个测试阶段。

缺乏规划和手动测试的方法会导致回归测试效率低下和测试不充分，并且对资源的利用也是低效的。为了得到一个计划周密、更高效的回归测试计划，必须要回答下面这些问题：

- 什么时候应该执行回归测试？软件的每次改动都可能会影响已经纳入基线的软件功能，所以需要进行回归测试。它包含了所有以前运行的测试过程。如果以前有缺陷的软件得到了改进，那么也应该进行回归测试。
- 回归测试应该包含什么内容？首先，回归测试应该集中在高风险的功能和执行最频繁的路径上。测试了这些元素以后，才能检查更细节的功能。对高风险的部分和修正缺陷可能潜在地影响的某些代码部分，回归测试可以有针对性地对这些部分进行自动测试，当然也可以重新运行整套测试。

  在理想的测试环境中，当新功能提交给测试时，所有必需的测试过程最终都会自动成为回归测试套件的一部分（见下）。
- 如何优化和改进回归测试套件？测试工程师可以按照下面的步骤来优化回归测试套件。

  1. 运行回归测试集合。
  2. 如果出现的情况是：回归测试集合运行成功了，但是错误在以后暴露出来了，那么把确定这些错误的测试过程和其他相关场景加入到回归测试集合。（缺陷可能在回归测试阶段以后才被发现，例如：在系统测试或者 β 测试期间，甚至通过技术支持电话才发现。）
  3. 重复这些步骤，用质量测量法不断地优化回归测试脚本套件。（在这种情况下，相应的度量应该是测试过程错误的数量和类型。）

除了遵从这些步骤，由于实施新功能、改错和修改其他代码可能会直接影响应用程序的某些部分，所以开发人员还应该提供有关这些部分的信息。开发人员提供的这些信息应该记录在一个回归冲击矩阵中。测试人员可以利用这样一个矩阵来评估回归测试套件、确定计划中的测试是否覆盖了足够的受影响区域，并且相应地增加和修改测试过程。

除了变动的软件部分以外，回归冲击矩阵还应该考虑其他部分。错误可能在最出人意料的地方出现。典型的回归错误例子是：因为一个模块发生了变化，所以会对它进行系统测试，但是，若还有另一个模块使用了这一模块生成的数据，那么这个模块的变化就会影响另一个模块。但如果回归测试只覆盖了直接修改的这个模块，那么就会遗漏很多回归错误。

我们需要牢记的一个要点是：在迭代开发和增量开发模型中，回归测试的开发也应该是渐进式的，并且必须在上一个迭代的回归测试的基础上开发。另外，当应用程序的一部分发生变化的时候，某些回归测试也应该修改或者删除。回归测试不是静态的，对回归测试的改动是必要的。

我们在第 22 条中已经讨论过，基本的测试过程集合应该基于需求。因此，为了保证把所有需求的变化直接传达给测试组，我们必须制定一个规程。如果当前需求规格说明书发生了变化，那么为了适应应用程序的新功能，回归测试过程也需要进行相应的修改。测试组和需求组之间必须有直接交流的途径。在测试组不知情和未认可的情况下，需求不应该发生变化。

回归测试用例应该能追溯到原始需求。这样当需求发生变化或者被删除时，就能知道需要更新的相关测试用例。回归测试阶段一般只是简单地重新运行现有的测试套件，希望这样就能发现所有问题。但是，随着被测系统的发展，添加新的测试用例是非常重要的，这样才能为处于持续改进状态下的软件提供覆盖的深度和广度。

- 为什么需要自动回归测试？当面对一个庞大、复杂的系统时，如果以前版本的功能测试套件也加入到下一个版本回归测试工作中时，那么最终可能需要执行成千上万的回归测试。因此，我们有必要自动地执行回归测试套件，并且还必须在一个能够快速地运行所有这些测试的、不占用太多资源的稳定测试环境中执行。如果这些测试不能自动执行，那么回归测试就会成为漫长和单调的一个过程。更糟糕的是，有些回归测试可能永远不会执行，使测试工作存在重大的漏洞。

此外，手工执行回归测试不仅单调乏味，容易出错，并且难以达到足够的测试覆盖率。由于某项测试在以前的版本已经执行过了，或者某个改正是发生在软件的其他部分、不是正在检查的部分，测试人员经常会觉得即使不重复执行这项测试也是安全的。还有些时候，产品的时间表不允许运行整套的手动回归测试。因为回归测试是冗长和单调的，并且很容易发生人为的错误，所以应该使用一个自动化工具或一个自制测试工具，或者同时使用这两种工具来完成回归测试。

有些工具允许创建测试过程组。例如：在 Robot 中，一个**测试外壳**由若干测试过程组成，并且会按照一个明确的、预定的顺序回放它们。在回归测试期间，执行一个测试外壳或包含整套系统测试脚本的**测试包**是对自动化脚本的充分利用。通过这样一个过程，测试工程师就能创建并执行一个全面的测试套件，然后把测试结果存储在单个输出日志中。

对于自动测试工具方面的投资来说，回归测试能够提供最大的投资回报。当为了修正缺陷系统发生变化的时候，为了验证是否引入了新的错误，我们可以不断地执行以前开发的所有脚本。因为脚本的运行可以不需要人工干预，所以为了发现错误，我们可以根据需要运行任意多次。自动测试为测试提供了简单的可重复性。

大量的测试工作是操作应用程序基本的用户界面。当应用程序具备相当的可操作性以后，测试工程师就可以对应用程序的商业逻辑和其他方面的行为进行测试。我们可以用部分回归测试套件完成一个简单的**烟雾测试**，第 40 条中将要讨论到，烟雾测试是回归测试套件的一个精简版本。综合使用手动回归测试和自动回归测试，测试组反复执行了相同的基本可操作性测试。例如：对于每个新版本，测试组需要验证以前工作正常的部分是否仍然运转良好。

除了会影响其他测试的进度，测试人员对手动执行回归测试的厌倦会耗尽他们的精力和激情。手动测试会因为重复执行这些测试而推迟，结果是牺牲了其他要求的测试的进度。自动测试在给定的时间限制下，为快速完成回归测试和执行更全面的测试提供了机会。自动回归测试释放了测试资源，使测试组的创造性和注意力转向更复杂的测试问题和关心的内容。

因为大多数自动测试工具都提供了在预定时间执行脚本的选项，不需要进行人工干预，所以自动回归测试还可以在下班以后执行测试。例如：测试工程师可以在早上设置自动测试工具在晚上 11 点执行一个测试脚本程序。第二天，测试组就可以检查测试脚本的输出日志，并且对结果进行分析。

- 如何分析回归测试的结果？当发现一个以前运转正常的系统功能出现了错误时，测试组必须确定最可能影响出错功能的其他功能部分是哪些。基于这种分析结果，回归测试可以对受影响的部分更具有针对性。如果错误改正了，那么为了保证缺陷已经修正，并且没有引入新的缺陷，我们需要针对有问题的部分再次运行回归测试。

  测试组还必须要确定相对产生较多错误报告的组件或者功能。基于这种分析，需要针对这些组件投入更多的测试过程和测试工作。如果开发人员指出一个特定的功能已经改好了，但是回归测试还是在不断发现软件中的相关问题，那么测试工程师必须要确定是由于环境的问题，还是软件修正的实现有问题。

  通过分析测试结果，我们还能确定特定的测试过程在发现错误方面是否有价值。即使一个回归测试用例从未发现任何错误，我们也不能肯定在以后的运行中它不会发现错误。但是当引入修改过的功能或者增加新功能时，评估每项测试的有效性和合理性是非常重要的。测试结果分析给我们提供了另一种方法，来确定哪些功能暴露的缺陷最多，因此，下一步应该集中对这些功能进行测试和改正工作。

上面这些要点有助于测试组进行有效的回归测试工作。与所有测试技术和测试方法一样，确保自动回归测试对当前项目的意义是非常重要的。例如：如果系统在不断的变化中，那么自动回归测试带来的好处可能就不多，这是因为在一个不断发展的系统上应用自动测试需要巨大的维护工作量。因此，在使一个回归测试套件自动化之前，确定系统是否稳定非常重要，也就是它的功能、底层技术以及实现是否总在变化。

第 40 条：实现自动生成和烟雾测试

自动生成通 常每天执行一次或者两次（可能在晚上），它们会使用最新的和稳定的代码。开发人员可以每天从负责生成的机器上取得组件，或者在迫不及待的情况下在本地重新生成。

烟雾测试 是回归测试套件的精简版本。它主要是对应用程序关键的高层功能进行自动测试。当拿到一个软件的新版本时，烟雾测试不是手动地反复执行所有测试，而是有针对性地验证软件中的主要功能是否仍然能够正常运转。

利用一个自动测试工具，测试工程师可以记录验证一个软件版本另外还需要的手动测试步骤。一般来说，这种验证在系统编译、单元测试和集成测试之后进行。它应该视为这个版本进入测试阶段的“入口标准”。

自动生成的实现，可以很好地连接起开发组和配置管理组的工作。大型软件系统一般是在下班后生成的，这样第二天就会有一个新版本[①]。但是，任何在无人值守的生成过程中遇到的问题必须尽可能快地得到解决，这样才不会延误开发的进度。如果软件生成或者回归测试失败了，那么有些组织采用诸如发送电子邮件或者文本页面等自动通知技术来和责任人取得联系。根据自动生成或者回归测试时间的长短，为了使错误在第二个工作日之前得到改正，有时改错的任务是非常紧急的。

除了软件生成自动化以外，自动进行烟雾测试能够进一步优化开发和测试环境。因为烟雾测试会自动地验证生成的版本，所以软件在生成之后总是处于可用的状态，这是普遍认可的理想状况。如果第 6 章论述的有关单元测试的实践也是生成过程的一部分，那么开发人员和测试人员就会确信在软件提交测试时，版本中的缺陷数量已经大大减少了。

如果完全遵从了前面提到的实践，那么典型的软件生成顺序会是这样的：

1. 软件生成（编译），可能包含第 6 章中描述的单元测试。

① [在大型项目中，为了完成频繁生成新版本的工作，通常要求配备复杂的配置管理或者过程管理工具。这些所谓的企业级工具提供的功能远比简单的、低成本的工具要强大得多，后者一般只能提供基于文件的版本控制。它们几乎随时都允许生成软件的工作版本，甚至在若干开发人员正在进行较大修改时。

2. 烟雾测试

3. 回归测试

如果在这个过程中任何一步发生了错误，那么整个过程就中断了。在问题改正之后，生成序列会在适当的位置重新启动。

要生成烟雾测试：测试组首先必须确定应用程序的那些部分是高层功能，然后针对系统的这些**主要**部分开发自动测试过程。在这里，**主要**这一词指的是使用最频繁的基本操作，通过执行这些操作可以确定软件中是否有重大缺陷。例如，主要功能包括：登录、添加记录、删除记录和生成报告。

烟雾测试也可以由一系列测试组成，这些测试能够验证：数据库指向了正确的环境、数据库版本是正确的、会话可以启动、所有的界面和菜单都是可访问的以及数据可以输入、选择并编辑。当测试应用程序的第一个发行版本时，为了尽早地开始测试开发工作，而不必等到整个系统稳定下来，测试组可能会希望对系统每个可用的部分进行烟雾测试。

如果结果和期望相符，这意味着处于测试状态的应用程序已经通过了烟雾测试，软件就正式地进入测试环境。否则，我们不能认为当前版本可以开始测试。

第 9 章　非功能性测试[①]

一个应用程序或者系统的非功能性方面会给测试和完善工作带来相当大的工作量，例如：性能、安全性和可使用性。是否满足非功能性需求决定了应用程序是勉强实现了它的功能，还是不仅最终用户和技术支持人员对它的接受程度很高，而且系统管理员也愿意对它进行维护。

在应用程序的开发生命周期中，过晚关注非功能性系统需求是错误的。为了满足非功能性需求，系统的架构和实现可能会发生变化。为了避免这种变化造成的影响，在开发生命周期的早期考虑非功能性需求是非常必要的。最好把非功能性需求当成应用程序的重要特性。因此，要想开发能被广泛接受的应用程序，准备好合适的非功能性需求文档是至关重要的。

在第 1 章中，推荐的做法是在需求阶段定义非功能性需求，并且把它们和对应的功能性需求关联起来。如果尚未定义非功能性需求，那么作为一个整体我们不能认为需求是完备的；如果不及早定义，那么系统的这些重要组成部分可能直到开发生命周期的后期才会得到应有的重视。

---

① 非功能性需求并不赋予系统更多的功能，但是会限制或进一步定义系统完成给定功能的方式。

第 41 条：不要事后才考虑到非功能性测试[①]

一般来说，软件项目中的需求、开发和测试工作总是过于关注系统的功能，直到非常晚的时候才开始关注软件的非功能性问题。当然，如果一个应用程序或者系统不具备正确的功能，那么就不能认为它是成功的。我们也可以讨论：非功能性问题是否可以在晚些时候再考虑，例如：通过版本升级或者补丁。

遗憾的是，这种方法会给系统的实现带来问题，甚至增大产品失败的风险。例如：一个只顾及功能而忽视了安全性的 Web 系统，可能会遭到恶意的 Internet 用户的攻击，进而会导致系统瘫痪、丢失客户数据和失去公众对站点的兴趣，并最终导致经济损失。再举一个例子：假设一个应用程序功能完备，但是不能处理大量的用户数据。虽然该应用程序提供了正确的功能，但是由于它没有满足客户的需要，所以还是没有价值的。类似的问题还会导致对应用程序口碑很差，并会因此失去客户。这种问题经常会使实现功能的努力付之东流，并且还要付出巨大的努力进行改正。

有关非功能性的事宜，最好早在应用程序的构架和设计阶段就开始研究。如果不能及早地关注这些方面的实现，那么以后为了满足非功能性需求而添加或修改组件就会很困难，甚至是不可能的。考虑下面的例子：

- Web 应用程序的性能。Web 应用程序一般在一个较小的环境中开发，例如：只由一个 Web 服务器和一个数据库服务器组成的系统。此外，当系统第一次投入生产时，最省钱的方法是为它配备最低的硬件配置，它们的性能只能为最初少量的客户提供服务。但是，随着时间的推移，Web 服务器的负载可能加重了，需要相应地提高站点硬件的处理能力来解决负载问题。如果不增加硬件的处理能力，那么用户会遭受性能问题，例如：载入页面时间过长甚至最终用户的浏览器可能访问超时。一般来说，向 Web 站点增加若干台机器，可以合理地**分散** Web 应用程序，以获得更高的性能，这样 Web 处理能力就得以提高。如果应用程序最初的架构和实现没有考虑这种扩展，那么为了获得这种可伸缩性，设计和实现需要进行很大的调整。结果是需要更多的费用，并且可能最糟糕的是：当工程师正在开发和完成经过改进的站点产品时，已经出现了重大的延期。

① 关于非功能性测试实现的详细讨论，请参见 Elfriede Dustin 等人的 *Quality Web Systems*（Boston, Mass.: Addison-Wesley, 2002）。

- 使用不兼容的第三方控件。有一种方法在增强系统应用程序功能的同时能够缩短开发时间，那就是在部分用户界面中使用第三方的控件，例如：ActiveX 控件。但是这些控件不可能与所有平台和设备兼容。如果一个应用程序在发行时，尚未在所有用户环境下验证其组件的兼容性，但是发行后发现了兼容性的问题，那么可能别无选择，我们只能删除这个第三方的控件，同时提供一个替代品或者定制实现，这就需要额外的开发和测试时间。在此期间，组织无法为某些用户提供服务，直到应用程序的新版本开发完成。

- 多用户客户端-服务器应用程序中的并发性。多个用户访问一个应用程序是客户端-服务器构架最重要的功能之一。在客户端-服务器系统中，并发性或者同时访问应用程序的数据是一个主要值得关注的问题（例如：两个用户试图修改同一个数据条目），原因是此时可能会出现意外的行为。为了保证在多用户环境下的数据一致性，我们需要一种同步数据处理的策略。这种策略一般在应用程序的底层来实现，有时也可以在用户界面实现。如果在应用程序的构架和设计中没有考虑并发性问题，那么为了适应这条需求，应用程序的组件以后必须要进行修改。

无论正在开发的软件产品是何种类型，任何组织必须认识到：在开发生命周期中，过晚关注应用程序的非功能性方面会带来风险。虽然经过判断某些风险可能无法马上解决，但是我们可以对它们进行了解和规划，这总比当问题暴露时再表示惊讶要好。

当规划一个软件项目时，需要考虑下列非功能性风险：

- 糟糕的性能。糟糕的程序性能可能只是使用不便，或者也可能是向最终用户提交了没有价值的应用程序。当评估一个应用程序的性能风险时，要考虑应用程序处理大量用户或者数据的需要（第 42 条将讨论大数据集合）。在基于服务器的系统中（例如：Web 应用），缺乏对性能的评估会让我们无法确定系统的可伸缩能力，并且导致对扩展系统的费用估计不准确或者根本就没有估计费用。

- 不兼容性。在最终用户各种不同的系统配置下，应用程序中各功能都能够正常运转，这是大多数软件开发活动最关心的问题。不兼容性会导致大量技术支持电话和退货。与糟糕的性能一样，兼容性问题可能只是让人感到有些厌烦，或者它们也可能会全面地妨碍应用程序的正常运转。

- 缺乏安全性。虽然所有的应用程序都应该重视安全性，但是基于 Web 的软件项目要特别关注这个问题。对安全性重视不够会导致危及客户数据的安全，甚至会出现法律问题。如果一个应用程序负责管理敏感的客户信息，那么安全性方

面的过失会使产品的名誉受损，并且可能卷入法律纠纷。

- 缺乏可使用性。由于糟糕的可使用性会使用户感到应用程序难以使用，或者不能完成必要功能，所以对应用程序的可使用性方面重视不够，会导致它在最终用户中间接受程度不高。这会导致技术支持电话繁忙，并且会对用户接受程度和应用程序的销售带来负面影响。

为了防止这些问题，在开发生命周期中及早对非功能性方面进行评估是非常重要的。

但是，同样重要的是也不要过度关心非功能性方面的问题。当解决非功能性方面的问题时，一般采用折衷的策略。例如对于应用程序的性能：有些软件项目过度强调了开发性能优异的应用程序，致使他们忽视了设计，这两个方面通常是矛盾的。过于重视性能而忽视设计会导致代码难于维护或者以后难于扩展。安全性是另一个需要折衷的例子：一个提供了最大安全性的应用程序架构会导致应用程序的性能低下。对诸如安全性或者性能之类的非功能性需求的关注会牺牲应用程序其他方面的问题，权衡这种风险是非常重要的。

对测试组来说，如果一个应用程序的非功能性需求，能在开发过程的需求阶段和功能需求一起考虑，那么这是最理想的情况。需求文档会详细说明每个用户交互的性能和安全性限制，这不仅能够使得开发人员保证在应用程序的实现过程中满足非功能性需求，而且也能让测试组为这些方面设计测试用例。第 21 条已经讨论过，对一个给定的需求，测试过程应该包含对每个非功能性部分的测试。

除了特定的详细需求的非功能性信息，总结出一组作用于所有需求的全局非功能性限制也是非常有益的。这样就避免了在每个需求文档中重复地描述相同的非功能性问题。

非功能性需求通常用两种方法形成文档：

1. 为系统中的所有用例建立一个用于定义非功能性需求的、系统级的规格说明书。例如："Web 系统的用户界面必须与 Netscape Navigator 4.x 或者更高版本，和 Microsoft Internet Explorer 4.x 或者更高版本兼容"。
2. 每条需求描述包含一个题目为"非功能性需求"的部分，这个部分记录了所有当前需求需要的、与系统级规格说明书不同的非功能性需求。

第 42 条：用产品级数据库进行性能测试

如果一个应用程序的用途是管理数据，那么随着应用程序中存储的数据的增加，测试组必须了解其性能的下降程度。数据库和应用程序的优化技术能够极大地降低这种性能的退化。因此，测试应用程序是否成功地采用了优化技术是非常关键的。

为了把应用程序在不同规模的数据集合下的性能制成图表，通常需要在不同的数据集合下进行测试。例如：为了调查当数据量增长时应用程序性能的变化，我们分别在 1 条、100 条、500 条、1 000 条、5 000 条、10 000 条、50 000 条和 100 000 条记录的情况下对其进行测试。通过此类测试，还可以获得应用程序数据处理能力的"上限"，也就是在应用程序的性能可以接受的前提下，数据库的最大规模。

在开发生命周期中，尽快开始应用程序的性能测试是非常关键的。这种做法使得在应用程序还处在开发状态时就开始性能的改进工作，而不是等到应用程序的重要部分开发和测试完成之后。在早期，需要关注只是主要的性能测试，而不是细微的性能调整。在开始阶段，任何明显的性能问题都应该得到更正，精细的调整和优化可以在开发周期的后期完成。

在项目早期，通常会制作一组普通的样例数据。由于时间上的限制，即使不是所有的开发和功能测试工作都使用这些数据，但是也会是大部分。遗憾的是，这些数据集合的规模和变化与实际情况相距甚远，当真正的客户试图使用应用程序时，由于他们使用的数据远比样例数据集合中的数据多，所以他们可能会对应用程序的性能表示惊讶。由于这个原因，我们推荐在开发应用程序的时候，测试组（或者还可能包括开发组）就应使用包含了范围广泛的数据组合和场景的产品级数据库。虽然这样需要预留更多的时间来创建数据库，但是当应用程序成为产品时，它对于避免即将出现的危机很有价值。

在项目中使用实际规模的数据库还有第二个好处。大型的数据库可能难于处理和维护。例如：生成一个很大的数据集合可能需要巨大的磁盘空间和强大的处理器能力。存储这种规模的数据集合需要更多的资源，例如：大容量的磁盘驱动器或者磁带。根据具体的应用程序，对大型的数据集合我们可能还需要考虑数据库的备份和恢复。如果数据需要在一个网络或者 Internet 上传输，那么支持人员可能还要为应用程序和其他对数据的处理过程考虑带宽的问题。尽早地在实际的数据环境中工作，有助于及早暴露这些问题，此时我们还有时间解决问题和找到更划算的解决方案。如果应用程序或者系统已经成为产品，那么解决这些问题必须在预算和时间两个方面都付出巨大的代价。

为大型的数据集合确定和填充数据的方法有几种，其中许多方法在第 10 条中已经描述过了。为了确定那些数据部分最关键，以及哪些数据部分最富于变化，我们需要和开发组、产品管理组和需求组进行协商。例如：一个订单处理系统所用的数据库会包含许多充满信息的表，但是这些表中变化最大的是存储订单和客户详细信息的表。

了解了必要的数据元素以后，就可以随机地产生（通过脚本或者其他程序）大量记录来增加数据库的规模。但是要记住：因为随机产生的数据并不能反映由用户输入的、真实的数据场景，所以它们可能对功能测试是无用的。通常有必要为功能测试和性能测试单独使用针对特殊目的的数据集合。

获得实际数据的一种方法是：从潜在客户或者现有客户那里拿到他们在应用程序中使用的或者计划使用的数据。这项工作应该在需求阶段完成，具体做法是通过调查最大的数据集合有多大、平均的数据库规模有多大等。对于前面的例子，就是确定订单处理系统的潜在客户拥有的数据集合范围是从 1 000 到 100 000 个订单。这些信息应该放入应用程序的商业用例，以及全局非功能性需求规格说明书中。在做出了支持这个订单数量范围的决定以后，就可以创建反映最小的预期订单集合和最大的预期订单集合的测试数据库，以及二者之间的中间规模的数据库。

如果正在开发的是一个 Web 或者客户端-服务器应用程序，那么一般来说，与用于测试和开发的硬件相比，最终运行软件产品的硬件功能更强大。一个产品级的主机很可能装备了更快和更多的处理器、更快的硬盘和更多的内存（RAM）。由于预算的限制，大多数组织不会为开发和测试环境配备产品级的硬件。

这种情况就需要我们根据测试硬件平台和产品级硬件平台的差异来推测性能。这种估计通常需要确定两种平台之间性能比例的基线。估计的两种平台的性能倍数确定了以后，我们就可以在处理能力稍逊的硬件上进行性能测试，以此来验证应用程序是否有能力处理大量数据。但是应该记住：导致两种系统性能出现差异的因素有很多，所以上述推测只能是粗略的估计。

第 43 条：为预期受众定制可使用性测试

对发布一个令用户满意的应用程序来说，可使用性测试是一个困难但是必不可少的步骤。可使用性测试的首要目标是：验证对应用程序有意向的用户是否能够和应用程序正确地进行交互，同时感到使用起来明确而方便。它需要检查应用程序界面的布局，其中包括：导航路径、对话框控件、文字、以及定位和可访问性等其他元素。支持组件（例如：安装程序、文档和帮助系统）也需要进行调查。

为了正确地开发和测试应用程序的可使用性，我们需要了解目标受众和他们的需求。这些需求应该出现在应用程序的商业用例和其他高层文档的显著位置。

从可使用性角度来说，下面几种方法可以确定目标受众的需求：

- 行业专家。在一个复杂的应用程序开发中，拥有一些同时也是领域专家的成员也非常有必要。这些能够不断地为需求组、开发组、测试组提供咨询的成员是重要的资源。因为对相关领域的规则和过程的意见经常会不一致，所以事实上大多数项目都需要有若干名行业专家组。
- 专题组。为了获得潜在的客户所提供的他们希望的界面，最好的办法是召开专题组会议，这样就能获得最终用户对备选用户界面的意见。原型和截屏在专题组讨论中是有力的工具。为了保证足够的代表性，挑选能代表产品各类最终用户的专题组是非常重要的。
- 调研。虽然不如行业专家和专题组有效，但是通过调研我们也能了解到有价值的信息：潜在的客户想如何使用一个软件产品来完成他们的工作。
- 对同类产品的研究。通过研究同类产品，我们能够获得这样的信息：在相同的和不同的问题域中，其他小组是如何解决问题的。虽然不能照搬其他产品的用户界面，但是研究其他小组或者竞争对手用户界面的实现方法是有益的。
- 观察用户的操作。通过观察用户和应用程序界面的交互，我们能够得到关于其可使用性的大量信息。这既可以通过用户使用应用程序的同时简单地做笔记这种方法完成，也可以通过录像完成。我们可以根据录像来分析、观察用户的动作，这样可使用性测试人员就能够知道哪些地方用户难于使用，哪些地方直观易用。

和大多数非功能性需求一样，比起以后再改进应用程序来，早关注可使用性问题，其结果要好得多。例如：有些应用程序设计和架构与用户界面需求不符；如果很晚才发现应用程序的架构不能支持优秀的可使用性，那么问题将非常难于解决。此外，制作应用程序的用户界面需要耗费许多时间和精力，因此在整个过程中及早确定正确的界面是明智之举。

在开发易于使用的应用程序时，用户界面原型是一个有效的工具。这种原型使得潜在的客户、需求人员和开发人员可以通过讨论来确定应用程序界面的最佳呈现方法。虽然上述工作可以在纸上完成，但是因为原型是交互式的，并且为应用程序的外观提供了更真实的预览，所以原型的效果要好得多。正如第 24 条中描述的，配合使用需求文档和原型还能为测试过程提供早期的开发基础。在原型的制作期间，有关可使用性方面实现的变化不会对开发进度造成很大的影响。

在开发周期的后期，最终用户的代表和行业专家应该参与可使用性测试。如果应用程序的最终用户有几类，那么每类至少应该有一人参加测试。参与人员可以使用开发组织的站点上的软件，或者按照有关可使用性评估指导书，把预发行版本发送到最终用户手中。最终用户测试人员应该记录那些不容易理解和难于使用的地方，并且提出改进建议。因为在开发生命周期中的这个阶段，大幅修改应用程序的用户界面一般是不切实际的，所以对反馈的要求应该定位在细微的改进上。

对于已经成为产品的应用程序也可以采用相似的办法。为了确定下个版本的应用程序在可使用性方面的改进方案，反馈和调研也是有力的工具。如果反馈是来自那些购买应用程序的已有既得利益的用户，他们希望通过改进应用程序来他们的需要，那么他们的这些反馈是非常有价值的。

第 44 条：特定需求和整个系统都需要考虑安全性

和其他非功能性需求一样，安全性需求应该和每条功能性需求联系起来。除了作用于整个系统的系统级安全性需求以外，每条功能性需求都可能包含一组与安全相关的问题，这些问题将在软件实现中解决。例如：在一个客户端-服务器系统中，必须指定允许登录的重试次数、登录失败时应采取的行动等等。其他功能需求有他们自己的有关安全性的需求，例如：允许用户输入的最大长度[①]。

有了关于安全性需求的正确文档，我们就可以创建测试过程来验证系统是否满足了它们。有些安全性需求可以通过应用程序的用户界面进行验证，例如：输入长度检查。对于其他情况，可能需要使用第 16 条描述的灰箱测试来验证系统满足特定的需求。例如：登录功能的需求可能指定用户名和密码必须用加密的格式传输。此时必须要使用一个网络监视程序来检查客户端和服务器之间发送的数据包，这样才能验证登录信息确实是加密的。还有些需求可能要求分析服务器硬盘上的数据库表或者文件。

除了和特定需求直接相关的安全性问题以外，软件项目还有全局的安全性问题，并且这些安全性与应用程序的构架以及整体实现相关。例如：一个 Web 应用程序有一条全局需求——各种客户私有数据都必须以加密的形式存放在数据库中。因为这条需求毫无疑问地对系统中的许多功能需求都起作用，所以它必须针对每条需求进行检查。另一个系统级安全性需求的例子是：用 SSL（Secure Socket Layer）来加密客户端浏览器和 Web 服务器之间传送的数据。测试组必须验证所有的这种传输都正确地使用了 SSL。建立这种需求一般是出于第 41 条中描述的风险评估的需要。

许多系统，特别是 Web 系统，都利用了第三方资源来实现特殊的功能性需求。例如：一个电子商务网站可能使用了第三方的支付处理服务器，那么就必须仔细评估这些产品，确定它们是否安全，并且保证对它们的使用是正确的，否则就会导致安全漏洞。对于测试组来说，尤其重要的是：验证所有经过这些组件的信息，是否遵从了系统的全局安全性需求规格说明书。例如：测试组必须验证第三方的支付处理服务器，不会把机密的信息写到一个日志文件中，否则当系统遭到入侵时，这些文件可能会被入侵者读取。

对于第三方的产品来说，紧跟和安装厂商提供的最新补丁是非常重要的。其中包含

① 检查输入长度对防止应用程序的缓冲区溢出攻击是至关重要的。更多的信息请参见 Elfriede Dustin 等人的 *Quality Web Systems*（Boston，Mass.：Addison-Wesley，2002）第 76～79 页。

Web 服务器、操作系统和数据库的补丁。例如：当发现安全性问题时，Microsoft 会频繁地为它的 Internet Information Server（IIS）提供安全性补丁。虽然验证这些系统级的补丁是否改正了发现的问题通常并不必要，但是安排人了解系统中使用的第三方产品的安全性补丁却非常重要。与所有的更新一样，每次安装了补丁之后，都应该对系统进行回归测试来保证没有引入新的问题。

如果已经确定了一个应用程序的安全性风险确实存在，那么外购安全性测试是值得考虑的。对于电子商务站点、在线银行和其他处理客户敏感数据的站点来说，安全性的优先级很高，因为一次非法进入对站点和组织来说可能就意味着灭顶之灾。对于采用最先进的工具和技术所产生的安全缺陷，有许多第三方的安全性测试公司可以对它们进行研究、甚至是对它们实现的研究。要发布一个安全的应用程序，那么将外部采购和内部测试结合起来是一种非常好的、针对安全性的非功能性需求的测试方法。

第 45 条：研究系统对并发性测试计划的实现

在一个多用户系统或者应用程序中，**并发性**是开发组需要面对的一个主要问题。在应用软件领域，并发性指的是对多个用户试图同时访问相同数据的处理。

例如：考虑一个多用户的订单处理系统，它允许用户添加和编辑客户的订单。因为每个新订单都会生成一条独立的记录，所以增加订单倒不是问题。若干用户可以互不干扰地同时向数据库增加订单。

但是，编辑订单却会导致并发性问题。当一个用户打开一个订单对它进行编辑的时候，本地计算机显示了一个包含订单信息的对话框。为了实现这一功能，系统从数据库中检索数据，并且把它临时存储在本地计算机的内存中，这样用户就可以看见并修改数据。修改完成以后，数据被送回服务器，这条订单的数据库记录就被更新了。现在，如果两个用户同时打开了同一条记录的编辑对话框，那么他们都在自己的本地计算机内存上拥有一个数据拷贝，并且都能对数据进行修改。如果他们两人都选择保存数据，那么结果会怎么样呢？

问题的答案取决于是如何设计应用程序对并发性问题的处理方式。管理多用户对共享资源的访问是当年引入多用户大型机系统时带来的挑战。任何能够被一个以上用户访问的资源都需要软件逻辑提供的保护，这些逻辑能够控制多个用户同时访问和修改资源的方式。自从网络文件共享、关系型数据库和客户端-服务器计算机出现以后，此类问题就变得更加普遍了。

在应用软件中，处理并发性问题有若干种方法。其中包括下面的方法：

- 保守方式。这种并发性模型在数据上加了锁。如果一个用户已经打开了一条记录，那么在允许编辑的环境中，系统就会拒绝来自其他用户的读取数据的请求。在前面的例子中，第一个打开记录进行编辑的用户，就获得了加在订单记录上的锁。后来试图打开这条订单的用户会收到一条警告消息，消息的内容是当前订单正在被其他用户编辑，他们必须等到第一个用户把修改的结果保存，或者取消操作后才能打开这个订单。对于很可能出现一个以上用户同时编辑相同数据的情况，就最适合采用这种并发性模型。这个模型的缺点是当一个用户已经打开某个数据时，其他用户就不能访问它了，这样导致了系统在使用上有些不方便。由于系统必须管理这些记录锁，所以这种模型在实现上也会有一定的复杂度。

- 开放方式。在开放的并发模型中，总是允许用户读取数据，甚至还可能允许更新数据。但是，当用户试图保存数据时，系统会检查自从这个用户检索数据以后是否有其他人更新过数据。如果数据发生了变化，那么更新就失败了。这种方法比保守模型允许更多的用户查看数据，所以它一般适用于不太可能出现多人同时修改同一数据的情况。但是，如果用户花费了大量时间来更新一条记录，到头来却发现根本不能保存，那么他们会感到不便。此时必须要重新检索记录并重新完成修改。

- 没有并发保护："胜利属于最后一个用户"。这是所有模型中最简单的一种模型，这种方法并不对多个用户编辑相同的数据提供保护。如果两个用户打开同一条记录并且对它进行了修改，那么第二个用户的修改结果会覆盖第一个用户的修改结果，这就是一种"胜利属于最后一个用户"的情形。第一个用户的修改结果将会丢失。此外，根据系统的具体实现方式，当两个用户试图同时保存数据时，这种方法可能会导致数据损坏。

应用软件处理并发性的方式会影响系统的性能、可使用性和数据完整性。因此，为了验证应用程序在并发性处理方面是否正确，根据为项目选择的并发性模型设计相应的并发性测试是非常重要的。

测试应用程序的并发性可能很困难。为了正确地检验系统的并发处理技术，时机是一个问题。无论是在保守锁定方式中，采用建立锁的方法，还是在开放锁定方式中，采用在保存数据之前先检查数据是否更新过的方法，为了确定系统是否恰当地保护了数据，测试人员必须要模拟两个用户同时读取或者写入数据。

在并发性测试中，我们可以综合使用手动和自动测试技术。两个测试工程师可以在不同的计算机上通过用户界面访问数据，通过口头约定来保证他们同时对数据进行访问。为了发现所有关于并发性的问题，我们可以用自动化的手段（通常在用户界面层以下）来检验系统是否允许更大容量和更可靠的瞬间访问。要设计并执行正确的测试，其关键是要对系统实现有所了解。

当相同的数据可以通过不同的界面或者功能更新时，并发性测试会变得更加复杂。例如：当一个用户正在编辑一条记录时，另一个用户删除了这条记录，此时系统必须加强对并发性的控制。如果采用保守方式中的锁，那么一条记录正在编辑时系统不允许删除它。事实上，为了保证正在编辑的某条记录不被其他功能所修改，应该测试所有可能访问这条记录的功能。

系统级的非功能性需求规格说明书，通常应该详细说明系统处理并发性问题的方式。

如果某些特殊的系统功能与此有冲突，那么应该在描述这些特殊功能的需求文档中对它们进行详细说明。例如：如果系统级需求规格说明书可能会要求开放方式的并发性，但是系统中某个特殊的敏感部分要求的是保守方式的并发性，那么在描述这个特殊的系统部分的需求文档中必须记录这一点。

根据系统采用的并发性方法，以及（在前面的例子中描述过的）系统的特殊部分采用的并发性方法，测试过程会是多种多样的。如果缺乏对系统中并发处理方法的了解，那么实现正确的测试会非常困难，因此测试组掌握相关知识是极端重要的。不同并发模型之间的细微差别必须要在测试过程中体现出来。

下面的列表详细说明了对于不同的并发性模型，测试过程应该关注的要点：

- 保守方式。因为当应用程序正在编辑一个数据时，它会强行限制用户访问那个数据，所以在测试保守的并发性方法中，主要关心的是验证能否正确地取得、释放加在记录上的锁，并且能正确处理应用程序中所有可能更新这条记录的部分。这里主要关心的是以下3个方面：

  - 锁的获得。因为同一时刻只有一个用户能够进入一条数据记录或数据项的更新状态，所以关键是系统必须把锁正确地分配给第一个请求的用户。如果获得锁的代码实现有错误，就会导致多个用户都认为他们自己拿到了锁，但是当他们试图保存数据时，可能会遇到错误或者发生不可预料的行为。获得锁的操作是可以测试的，具体方式是：让两个用户试图同时进入编辑状态，或者也可以使用大量的请求。对于后者，我们可以使用一个脚本来产生比如说1 000个或者更多的编辑数据请求，这些请求是几乎同时的，以此来验证只有一个请求获得成功。

  - 锁的效用。如果一个用户取得了锁，那么系统必须确保其他任何用户不能用任何方法修改这个数据，其中包括对数据的更新和删除。具体的验证方法是：让一个用户打开一条记录（进入编辑模式并且保持这个状态），同时其他用户在应用程序的所有地方试图编辑、删除或者以其他方式更新数据。系统应该拒绝所有其他用户更新数据的企图。

  - 锁的释放。在保守的并发模型中，另一个需要小心对待的地方是锁的释放。测试人员必须要验证：当编辑数据的用户释放了这条记录以后（无论是更新完毕还是取消操作），系统能够成功地让其他用户使用这条记录。释放锁需要注意的一个重要方面是错误处理，也就是持有锁的用户遇到错误（例如：客户端系统崩溃了）的情况下，系统应该完成什么样的操作。锁是否

就失去控制了（处于无法释放的状态）？系统从释放锁的故障中重新恢复的能力需要重点考虑。

- 开放方式。在某种程度上，开放的并发性模型的实现复杂度稍低。因为允许所有用户检索并编辑数据，所以更新是唯一需要关注的要点。前面已经提到过，测试开放的并发性的最佳方式是综合使用手动和自动测试技术。在手动的方法中，两个测试人员编辑数据，然后试图同时保存数据。第一个用户更新操作应该是成功的，但是第二个用户应该得到一条消息，其内容是另一个人已经更新了数据。此时，他需要重新装载数据并且重新完成修改操作。

  除了手动测试以外，还应该使用自动和海量的测试方法。为了保证开放的加锁机制能够使得同一时刻只有一个用户更新了记录，我们可以利用脚本发起成百上千的、几乎同时发生的更新请求进行测试。其余的用户都应该收到错误消息，消息的内容是：因为另一个用户已经更新了记录，所以记录不能保存。

  与保守的并发模型一样，在开放的并发模型中，测试人员必须保证对所有可能修改数据的地方，开放的并发性都得到了验证，其中包括来自用户界面的不同部分的记录操作。

- 没有并发控制。最简单的也是最容易出错的方法是系统不具备任何并发性控制。测试这样的系统和测试开放的并发模型非常相似，区别只是：无论更新请求的顺序如何，所有用户都应该成功完成更新操作。在这里也应该综合使用手动和自动测试技术，但是在没有并发控制的情况下，应该关注的是数据的完整性。测试人员还应该验证是否正确地处理了更新错误，例如：当另一个用户试图更新某条记录时，它却被删除了。

测试组对系统实现计划的研究有助于他们正确地确定需要完成的并发性测试类型。因为并发性在大多数多用户系统中是一个普遍的问题，所以正确地测试并发性是非常重要的。

第 46 条：为兼容性测试建立高效的环境

对应用程序兼容性的测试可能是一项复杂的工作。现有的操作系统、硬件和软件配置是如此丰富，它们可能的组合数量非常之多。遗憾的是，兼容性测试问题是无法回避的——在任何给定的系统上软件或者能够正常运行，或者不能正常运行。但是一个适合兼容性测试的环境会使这个过程更加高效。

用于正在开发的系统的商业案例应该规定所有最终用户所使用的操作系统（如果需要，还有服务器操作系统），应用程序必须能够在这些操作系统上运行。另外，还需要明确规定其他兼容性。例如：正在测试的产品可能需要和某些版本的 Web 浏览器、某些硬件设备（例如：打印机）、或者其他软件（例如：查毒工具或者字处理软件）进行操作。

兼容性还可能扩展到从以前的软件版本升级。不仅系统本身必须能够正确升级以前的版本，而且还必须考虑以前版本的数据和其他信息。对一个应用程序来说，有必要向下兼容以前版本的数据吗？用户的首选设置和其他设置应在升级后保留吗？在全面地评估兼容性问题时我们必须要给出这些问题、还有其他一些问题的答案。

由于所有可能的配置和潜在的兼容性问题非常多，所以我们可能无法直接测试每种可能性。按照出现的频率的高低，软件测试人员应该把目标应用程序可能的配置进行排序。例如：如果大多数用户在拨号连接时运行的是 Windows 2000 操作系统下的 MS Internet Explorer 6.0 浏览器，那么这种配置应该出现在列表的顶部，并且在兼容性测试中受到重视。对于一些不常见的配置，时间和费用上的限制可能会把测试工作降格成快速的烟雾测试。

除了选择最重要的配置以外，测试人员还必须为兼容性测试确定恰当的测试用例和测试数据。因为为每种可能的配置运行全套测试用例会花费很长的时间，所以这种做法通常是不切实际的，除非应用程序的规模很小。因此我们有必要挑选最具代表性的测试用例集合，这些用例能够证实应用程序在一个特定平台上的正确机能。此外，测试人员还需要使用恰当的测试数据来支持这些测试用例。如果测试应用程序时还需要考虑其性能，例如：如果应用程序支持无限数量的数据，那么测试人员也需要使用充分大规模的数据集合来进行测试。

如果在应用程序中发现了兼容性问题，那么在开发过程中甚至发行以后，不断地更新兼容性测试用例和数据是非常重要的。当为兼容性测试套件选择需要加入的测试时，技术支持电话是一个很好的信息来源。这些信息还可能指出我们是否为兼容性测试选择

了最恰当的配置，还可能会使我们修改配置集合中元素的优先级。

另一个可能提供最终用户配置和兼容性问题信息的来源是 β 测试工作。在应用程序正式发行之前，分布广泛的 β 测试能够提供大量真实的、关于最终用户配置和兼容性问题的数据。

对兼容性测试环境的管理可能是一项主要的工作。为了正确地测试应用程序，最好完全重建最终用户环境。这种方法不仅需要安装实际的操作系统、硬件和软件配置，而且还要执行为每次安装挑选的测试过程。但是，正如前面讨论过的，用户可能的配置组合非常多，因此，为每种可能的测试配置都购买并安装用来表示这种配置的机器通常是不划算的；甚至为每项新测试从头开始重装一台或者几台计算机的代价也是很高的。因此，我们必须采取其他方法。

一个实用而经济的方法是综合使用活动硬盘驱动器和分区管理工具。这样我们就能用少量的计算机运行大量的配置。测试人员只需把正确的驱动器装入机箱、重启并选择需要的配置即可。市场上有许多分区管理工具可以在这种方法中使用，例如：Partition Magic。

另一种方法是使用驱动器映像程序（例如：Symantec Ghost）为需要的配置建立映像文件。以后可以用这些映像在目标计算机上重建配置。这种方法的缺点是：根据安装的规模，从映像重建配置可能会花费大量的时间。此外，映像文件可能很大，所以我们必须建立一种机制来管理它们。

无论使用的是何种安装管理技术，一旦在目标计算机上正确的配置环境已经启动了，那么我们就可以执行配置测试了，并且可以通过分析确定应用程序在当前平台上的兼容性。

在兼容性测试中，应用程序在目标配置上的安装方法非常重要。安装应用程序的方式和最终用户的安装方式完全相同是非常关键的，其中包括使用相同版本的组件和相同的安装过程。满足上述要求的最好方法是：在开发工作的早期完成安装程序，并且提供给测试组的是可安装形式的软件，而不是开发组制作的专用的包。如果使用的安装方法是特殊的和手动的，那么测试环境就不能反映最终用户的环境，并且兼容性测试结果可能也太不准确。不要在兼容性测试平台上安装开发工具也是非常关键的，因为这些工具可能“污染”测试环境，并使之不能准确地反映真实的操作条件。

第 10 章　管理测试的执行

测试执行阶段跟在前 9 章所讨论的各个阶段后面。我们已经有了测试策略、测试计划、设计并开发完成的测试过程以及可以操作的测试环境，前面几个阶段中建立起来的测试，现在到了该执行的时候了。

如果系统的开发工作还在进行，并且已经有了可供测试的软件版本，那么为了执行测试、跟踪发现的缺陷并提供关于测试工作进度的信息和度量，测试组必须制定一个详细的测试工作流程。

下面讨论的这几条涉及到组织中在测试执行期间的许多团队，其中包括测试组、开发组、产品管理人员和其他组。所有这些组的工作就是为了保证能够发现缺陷、为缺陷划分优先级和纠正缺陷。

第 47 条：明确定义测试执行周期的开始和结束

不管是哪个测试阶段，为软件测试执行周期定义**入口标准**（测试开始时间）和**出口标准**（测试完成时间）都是非常重要的。

入口标准描述了何时一个测试组可以开始测试一个特定的版本。在系统测试期间，为了接受一个软件版本，必须满足各种标准。例如：

- 所有的单元测试和集成测试已经成功完成。
- 软件的生成（编译）过程没有任何错误。
- 软件版本通过了第 40 条中描述的烟雾测试。
- 配套文档（**发行通知**）已经完成，文档的内容涉及软件版本的新功能和修改的内容。
- 缺陷已经修正并且准备重新测试（请参见第 49 条对缺陷追踪生命周期的讨论）。
- 源代码已经存储在版本控制系统中。

只有在满足了入口（接受）标准后，测试组才能准备接受软件版本并开始测试周期。

与定义测试阶段的入口标准一样，出口标准的确定也同样重要。出口标准描述了软件完成了充分测试的时间。和入口标准一样，出口标准依赖于当前的测试阶段。由于测试资源是有限的，测试预算和测试工程师的人数也有限，并且截止期限很快就到了，所以测试工作的范围也一定有自己的限制。测试计划必须明确地指出什么时候完成测试：如果出口标准定义得非常模糊，那么测试组就不能确定测试工作完成的时间。

例如：一个测试的完成标准可能是这样描述的：所有基于需求的、预定义的测试过程在执行过程中没有出现任何重大错误，也就是所有高优先级的缺陷都已经被开发人员修正了，并且由测试组的成员用回归测试进行了验证。如果遵从了本书中讨论的所有实践，那么满足这样一个标准会使我们充分相信系统满足所有需求，并且没有重大错误。

下面列出了几个样例，可以把这些陈述作为一个应用程序出口标准的一部分：

- 已经执行了用来确定系统满足指定的功能性和非功能性需求的测试过程。
- 在测试结果中记录的所有的 1 级、2 级和 3 级（导致运行中断的、紧急的和高优先级的）软件问题都已经解决。
- 报告的所有 1 级和 2 级（导致运行中断的、紧急的）软件问题都已经解决。
- 在测试结果中记录的所有的 1 级和 2 级（导致运行中断的、紧急的）软件问题都已经解决，同时报告的 90%的 3 级问题也已经解决。
- 软件发货时可能尚存在已知的低优先级缺陷（当然也会有若干未知缺陷）。

即使满足了出口标准，但是软件的成功也只是说明它对客户是有用的。因此，用户验收测试是测试计划中一个非常重要的因素。

开发人员必须了解系统验收标准。在把测试方案提交评审之前，测试组必须要及早和开发人员讨论入口和出口标准。如果可能的话，一个组织的入口和出口测试标准应该标准化，标准化的基础应该是经过若干项目考验的标准。

系统发行时可以有一些缺陷，这些缺陷将会在以后的版本或者补丁中加以解决。在系统成为产品之前，对测试结果的分析有助于确定哪些缺陷必须马上修正，哪些缺陷可以以后再解决。例如：有些“缺陷”的修正可以作为功能增强，在后续的软件版本中再处理。项目经理或者软件开发经理以及变更控制委员会的其他成员是确定马上修正一个缺陷，还是冒险带着某个缺陷发行产品的决策人。

其他的度量也必须作为出口标准的一部分进行评估。例如：

- 在回归测试中，从以前运转正常的功能中发现缺陷的比例有多大？换句话说，缺陷修正工作破坏以前运转正常功能的频率有多高？
- 缺陷修正失败的频率有多高？也就是我们认为一个缺陷已经改正了，但是其实并不是这样。
- 随着测试阶段的继续，新缺陷的发现率其走势如何？随着测试工作的进行，缺陷发现率应该呈下降的趋势。

虽然应用程序中很可能还有尚未发现的缺陷，但是当它处于可以发货的状态，或者满足出口标准时，我们就可以认为测试完成了。

现实情况中预算和进度都有一定的限度，那么必定存在一个测试必须结束和产品必须发行的时刻。在软件测试中，最困难的决定可能就是停止测试的时机。为了帮助测试组做出这种决定，我们必须建立软件完成和发行的质量方针。

第 48 条：隔离测试环境和开发环境

在测试组准备实施测试策略时，测试环境的建立是非常重要的。

测试环境必须要和开发环境分离开；这是为了避免测试期间的严重疏忽和无法追踪软件的变化，但是实际情况却经常不是这样：为了节省费用，测试组并不能获得一个独立的测试环境。

如果没有独立的测试环境，那么测试工作会遇到下面的一些问题：

- 环境的变化。一个开发人员可能在没有通知测试组的情况下，把缺陷修正的结果或者其他变化应用到开发配置中，或者向开发配置添加了新的数据。如果在此之前，测试人员刚刚记录下一个依赖于环境的缺陷，那么这个缺陷现在可能就不能重现了。缺陷报告只能标记为“关闭——不能重现”，从而导致了开发资源和测试资源的浪费。此外，如果开发人员没有通知测试人员：代码在一天内发生了变化，那么测试人员可能就会遗漏新的缺陷，他们不知道应该重新测试修改后的代码。

- 版本管理。如果开发人员和测试人员共享同一个环境，那么版本的管理就比较困难。开发人员可能需要为软件增加新的功能，同时测试人员却需要一个稳定的版本来完成测试工作。反过来，开发人员经常需要在他们自己的台式机以外的其他计算机上运行软件的最新版本。如果测试人员共享了开发环境，那么开发人员可以利用的环境中的资源就会减少。

- 操作环境的变化。为了在各种不同的条件下测试应用程序，可能需要对测试环境进行重新配置。这会造成环境的严重混乱，这种混乱会影响开发人员的活动。例如：机器需要重启，或者为了安装软件和硬件，机器几天不能使用。

虽然由于预算有限，经常无法建立一个独立的测试环境，但是采用某些节省费用的方法，我们还是能够建立起一个经济上可以负担的、独立的测试环境。例如：

- 降低性能和容量。在一个测试环境中，可能并不需要带有大容量的硬盘和内存的高性能的主机。大多数测试活动可以使用低配置的机器，如果需要，我们可以用外推法估计软件在高性能的硬件平台上的性能。但是因为外推法是一种不准确的性能测量方法，它只能得出在产品环境中实际性能的一个近似估计，所以只有当预算不允许配备产品级的测试机器时，我们才推荐采用这种方法。如果需要精确的数字，那么性能的度量应该来自产品级的硬件。

- 使用活动硬盘。测试不同的配置并不需要多台机器或者完全安装所有的配置。包含多个分区的活动硬盘使得一台机器能够容纳多个软件配置，这样就节省了费用和时间。关于在兼容性测试中使用活动硬盘的详细介绍，请参见第 46 条。

- 建立一个共享的测试实验室。为了分担测试环境的费用，若干软件项目可以共享一个测试实验室。为了用于不同的测试项目，实验室应该设计得容易重新配置，而不要只能专门用于某一个项目。

第 49 条：实现缺陷追踪生命周期

缺陷追踪生命周期是测试工作的一个关键部分。为系统环境评估并选择一个恰当的缺陷追踪工具是非常重要的。如果取得或者自主开发了这样一个工具，那么我们就应该制定、文档化并传达一个缺陷追踪生命周期。所有涉众必须了解缺陷处理流程，这个流程从发现缺陷直至缺陷被解决。

缺陷修正之后，必须对它进行返测。在缺陷追踪系统中，相应的元素应该标注为处于**返测状态**。开发经理必须了解并遵守用于追踪缺陷状态的过程。如果开发经理不遵守追踪过程，那么测试组就不知道应该返测哪些缺陷？

为了有助于开发组的缺陷修正活动，测试工程师必须将缺陷的细节和重现缺陷所需要的步骤文档化，或者引用发现问题的测试过程。为了使高优先级的缺陷首先得到解决，缺陷一般按照优先级进行分类。当变更控制委员会对重大的缺陷报告进行评审时，测试人员必须加入变更控制委员会。如果在软件的一个新版本中，以前发现的某个缺陷已经改正了，那么测试工程师会通过缺陷追踪工具得到通知。然后测试人员通过访问追踪工具得到那些可以返测的缺陷。缺陷修正记录最好还要包含对缺陷修正的描述以及可能受影响的系统其他部分，这些信息有助于测试人员确定哪些可能受影响的部分也需要返测。

每个测试组必须用一个详细的过程完成缺陷报告工作，其中包括下面几个步骤：

1. 分析和缺陷记录条目

这个过程应该描述如何评估不符合预期的系统行为。首先必须排除错误的结果。有时测试会产生缺陷的误识别，也就是正在测试的系统行为是正确的，但是测试却不正确地报告了一个错误。而有时测试（特别是当使用测试工具的时候）又会产生缺陷的漏识别，也就是虽然报告显示系统通过了测试，但是事实上系统中存在错误。为了确定测试过程输出的正确性，测试人员必须具备特殊的诊断能力。

假设测试人员的诊断结果所确定的不符合预期的系统行为确实是一个缺陷（没有漏识别缺陷、也没有误识别缺陷、也不是重复报告缺陷等等）。一般情况下，由测试组的成员，或者负责报告缺陷的其他小组成员将软件问题或者缺陷报告输入到缺陷追踪工具。

下面是缺陷文档中应该包含的各种属性的示例：

- 主标题。主标题的前缀应该是发现缺陷的系统部分的名字，例如：报表、安全性、用户界面或者数据库。

- 正在测试的工作版本。正在测试的工作版本号。软件产品版本号和软件测试的工作版本号应该有区别。例如：Build #3184，Test_Build_001。
- 操作系统和配置。列出发现缺陷的测试工作所用的操作系统和其他配置元素（UNIX，Win98，Win95，WinXP，WinMe，Win2000；I.E. 5.5，I.E. 6.0，Nescape 6 等等）。
- 附件。附件的例子有：有关错误的截屏图像和打印出来的日志。附件应该放入中心库中供所有涉众访问。
- 再现性。有时缺陷只是偶尔发生。这里需要指明缺陷是总可以重现、随机出现或根本不能重现。
- 重现错误的步骤。如果错误可以重现，那么这里应该包含开发人员能够重现缺陷的所有必要信息，例如：测试中使用的特殊数据。另外，还应该对错误本身明确地加以描述。避免诸如“程序显示了不正确的信息”之类的模糊语句。相反，应该具体指出程序显示的错误信息，以及正确的信息（预期的结果）。
- 返测失败。如果在返测中缺陷仍然存在，那么这里给出对返测失败的详细描述。
- 分类。需要报告的行为类别可能包括“缺陷”、“增强请求”或者“变更请求”。“增强请求”或者“变更请求”的优先级在通过评审之前应该保持在 N/A 状态。
- 解决方案。开发人员指出修正一个缺陷应该采取的正确措施。

在报告缺陷时，应该保持固定的**粒度**。每个报告应该只覆盖一个缺陷。如果若干缺陷是相关的，那么应该每个缺陷一个条目，但是可以交叉引用其他的条目。

2. 划分优先级

过程必须定义为每个缺陷分配优先级的方法。测试工程师最初必须评定：对于成功地使用系统来说，问题的严重程度有多大。最严重的缺陷可以导致软件失灵，并且阻止测试活动继续进行。缺陷一般提交到一个变更控制委员会（CCB），用于进一步评估和处理，这些内容在第 4 条中已经讨论过了。

下面提供了一个通用的缺陷优先级分类方法。

1. 导致运行中断——由于缺陷导致应用程序崩溃，预期的功能没有实现等原因，

测试工作无法继续进行。

2. 紧急——事件非常重要，并且需要马上给予关注。
3. 高级——事件是重要的，并且应该在紧急的事件处理之后尽快得到解决。
4. 中级——事件是重要的，但是由于解决问题需要一定的时间，所以可以用较长的时间解决。
5. 低级——事件不重要，可以在时间和资源允许的情况下再解决。
6. N/A——没有适用的优先级（例如：变更请求和增强请求）。

缺陷的优先级必须要根据项目和组织的需要进行剪裁。有些项目只用简单的“高/低”或者“高/中/低”优先级系统可能就足够了。而有些项目可能需要更多的缺陷优先级等级。但是如果使用的优先级等级太多，会使得对缺陷进行恰当的分类变得困难，因为这些等级已经失去了它的特征，并且开始混合在一起了。

3. 重复出现

过程应该定义如何处理重复出现的问题。如果某个缺陷以前曾经解决过，但是又不断地出现，那么应该如何处理？是应该重新处理以前的缺陷报告、还是应该增加一个新的缺陷条目？一般推荐的做法是：审查以前的缺陷报告来研究已经实现的修正，但是用一个新的缺陷报告来引用以前的这一“已经关闭——但是相关”的缺陷报告。此类引用提供了缺陷在改正之后又被重新引入的频率，可能会发现一个配置问题或者开发方面的问题。

4. 关闭

如果某个缺陷已经修正、确定是重复缺陷、或者发现其实不是缺陷，那么报告就应该结束了。但是它们应该保存在缺陷数据库中。任何人不应该从追踪系统中删除缺陷报告。这样就能保证正确地追踪所有的缺陷和它们的历史。除了可能会失去将来证明是有价值的历史信息以外，从缺陷追踪系统中删除记录会破坏编号系统的连续性。此外，如果有人有缺陷报告的硬拷贝，那么他们不会认同这个追踪系统文件。通过简单地把缺陷标注为“关闭——和预期相同”、“关闭——不是缺陷”或者“关闭——重复”来关闭一个缺陷，这一做法要好得多。

如果有一个规则能够部分地修正缺陷，那么它可能是有用的，例如：“如果一个缺陷只是部分改正了，那么缺陷报告不能作为改正来关闭”。通常这不应该是个问题。如果缺陷报告足够详细，并且粒度足够小，那么每个报告会只包含一个问题，这样就不会出现“部分改正”的情况。如果遇到了多个问题，那么不管问题可能是多么相似，每个问题都应该形成自己的缺陷报告。这样就不会出现把一个复杂的问题标注为“部分解决”的现象了，相反每个改正的元素和尚未改正的元素各自都有自己的记录。

如果一个缺陷已经被纠正，并且已经通过了软件开发组的单元测试，那么修改后的软件代码应该存储进软件的配置管理系统。当我们认为纠正的缺陷数量已经足够多了，或者只是改正了一个严重缺陷的情况下，我们就可以做出决定生成一个新的软件版本并且交给测试组。

在软件生命周期的各个阶段都可能发现缺陷。我们推荐测试组在生成软件问题报告时，按照在生命周期中发现缺陷的阶段对每个缺陷进行分类。表 49.1 是一个软件问题报告的类别示例：

表 49.1 缺陷类别和相关的测试生命周期元素

类别	问题发生的阶段以及适用元素	系统	软件	硬件
A	系统开发计划	√		
B	操作概念	√		
C	系统或者软件需求		√	√
D	系统或者软件的设计		√	√
E	完成编码的软件（正在测试的应用程序）		√	
F	测试计划、测试用例和测试过程报告		√	√
G	用户手册或者支持手册		√	√
H	项目中要遵守的过程		√	√
I	硬件、固件、通信设备			√
J	项目的其他方面	√	√	√

当使用一个缺陷追踪工具时，测试组必须定义缺陷生命周期模型，并且把它形成文档，模型也被称为**缺陷工作流程**。有些组织中，配置管理组或者过程工程组负责缺陷工作流程；而另有些组织中，测试组负责缺陷工作流程。下面是缺陷工作流程的一个例子。

缺陷工作流程

1. 当缺陷报告最初生成时，把它的状态设为“新”。

2. 赋予各个小组（测试组、文档组、产品管理组、开发组、培训组和需求组）打开问题的能力。他们能够选择的问题类型有：

—错误

—变更请求

—增强请求

1. 然后由缺陷发现者选择缺陷的优先级，再由变更控制委员会（CCB）进行修改或者批准。

2. 委派一个组或者个人（例如：软件经理或者 CCB）负责评估缺陷、为缺陷分配一个状态，并且可能修改问题的类型和优先级。

给合法的缺陷分配的状态是“打开”。给重复的缺陷或用户错误分派“关闭”状态。只有测试组有权关闭一个缺陷。必须将缺陷“关闭”的原因文档化：

—关闭：不能重现

—关闭：重复

—关闭：工作正常

如果已经确定缺陷报告事实上是增强请求，那么为这个报告分派“增强”状态。

1. 如果缺陷状态是“打开”，那么软件经理（或者其他被指派的人员）把缺陷分配给负责的涉众，并且把状态设置为“开发”。

2. 如果开发人员开始改正缺陷了，那么状态应该设置为“正在开发”。

3. 如果开发人员认为缺陷已经改正了，那么开发人员为了记录这次修改，他们会在缺陷追踪工具中把状态置为“修改完毕”。如果软件功能是正确的，那么也可以把状态设置为“工作正常”或者“缺陷不能重现”。同时开发人员把缺陷重新分配给缺陷发现者。

4. 如果创建了新版本，那么开发经理把当前版本中所有改正的缺陷的状态设置为“返测”状态。

5. 测试工程师返测这些改动和其他受影响的区域。如果缺陷在修改后已经被纠正了，那么测试工程师把状态置为“关闭——改正”。如果缺陷在修改后没有被纠正，那么测试工程师把状态设置为“返测失败”。如果缺陷本身被改正了，但是改动影响了软件的其他部分，那么就产生了一个新的缺陷。在这种情况下，需要重复执行上述缺陷工作流程，开始了一个新的循环。

第 50 条：追踪测试工作的执行

一个软件项目所涉及的所有人都希望知道测试何时结束。为了能够回答这个问题，需要对测试的执行进行有效的追踪。为了完成这项工作，我们需要收集数据或者度量来显示测试进度。为了确保软件成功，度量有助于确定何时必须完成改正工作。此外，利用这样的度量，测试组能够预测应用程序的发行时间。如果发行日期已经确定了，那么这些度量可以用于测量完成率。

在测试执行周期中，进度的度量需要反复收集。有关进度的度量包括：

- 测试过程执行状况（%）＝已经执行的测试过程数量/测试过程总数。这个执行状态度量由已经执行的测试过程数量和计划中测试过程的总数相除得出。通过检查这个度量值，测试组可以确定尚未执行的测试过程比例。这个度量本身提供的信息并不能反映应用程序的质量。它提供的信息只能反映测试工作的进度，不包括有关测试是否成功的信息。

测量已经执行的测试过程步骤数量是非常重要的，而不仅是已经执行的测试过程数量。例如：一个测试过程可能包含了 25 个步骤。如果测试人员成功地执行了步骤 1 到步骤 23，但是在步骤 24 遇到了导致运行中断的错误。如果只是报告整个测试过程失败，那么提供的信息未免太少；更有效的测量进度的方法要包括已经执行的步骤数量。具体到步骤一级来测量测试过程的执行状态是一个更精细的进度度量。

追踪测试过程执行的最好方法是制作一个包含正在测试的版本的标识符、所有测试过程的名字列表、为每个测试过程分配的测试人员以及完成比例的矩阵，这个矩阵每天更新，它测量出成功执行的测试过程的步骤与计划执行测试过程步骤的总数的比率。许多测试管理工具或者需求管理工具有助于使这个过程自动化。

- 缺陷持续时间＝从发现缺陷到改正缺陷的时间跨度。另一个确定进度状态的重要度量是改正一个缺陷所需时间，也被称为**缺陷持续时间**。缺陷持续时间是一个高级度量，它用于验证缺陷是否被及时地解决。利用缺陷持续时间的数据，测试组可以进行趋势分析。例如：假设一个项目记录了 100 个错误。如果文档中记载的以往的经验表明开发组每天能改正 20 个错误，那么改正这 100 个问题报告需要的时间估计为一个工作周。在这种情况下，缺陷持续时间平均是 5 天。如果缺陷持续时间延长到 10～15 天，那么开发人员对改错工作迟钝的反应，可能会影响测试组在计划的截止期限到来之前完成测试的能力。

标准的缺陷持续时间测量方法并不能适用于所有情况。根据正在实施的特殊缺陷修正工作的复杂度或者其他标准，有时我们必须对它进行调整。

如果开发人员不及时修正缺陷，那么这在项目中会引发连锁反应。测试人员可能在其他部分遇到相关的缺陷，创建重复的缺陷报告，但是及时的缺陷修正工作可能会防止此类问题的发生。此外，一个缺陷持续的时间越长，那么纠正缺陷的难度可能就越大，因为我们可能在错误代码的基础上又加入了新的代码。如果缺陷在最初发现的时候就被改正了，那么这样对软件的影响程度比以后改正要小得多。

- 从缺陷纠正到返测的时间＝从缺陷被修正并且在新版本中发布的时间到缺陷被返测的时间跨度。缺陷修正到返测的时间度量提供了一种衡量测试组的返测缺陷速度的方法。如果已经改正的缺陷不能得到充分而及时的返测，那么这样就可能会延缓进度，因为开发人员不能得到有关修正的保证：它没有引入新的错误或者原来的问题已经正确解决。其中后一点特别重要：正在开发的代码总是假设以前开发的代码已经被改正了，如果上述假设被证明是错误的，那么正在开发的代码也需要返工。如果缺陷不能得到及时返测，那么就必须提醒测试组返测缺陷修正的重要性，这样开发人员才能继续工作，才能知道他们所作的改动已经通过了返测。

- 缺陷趋势分析＝在测试生命周期内，随着测试工作的进行，发现缺陷的数量的变化趋势。缺陷趋势分析有助于确定所发现的缺陷的变化趋势。当系统测试阶段日益临近结束时，趋势是在变好（也就是发现的缺陷越来越少）还是在变坏？这个度量和“新发现的缺陷”密切相关。当系统测试阶段接近结束时，新发现的缺陷数量应该减少。如果情况不是这样，那么可能表明系统中存在的问题还很多。

  如果没有发布新的功能，并且正在测试的是相同的代码（只有修正缺陷所引起的代码的改变），但是发现的缺陷数量随着每个新的测试版本在增加，那么可能出现了下面的问题：

  - 对以前的缺陷修改不正确
  - 对以前的版本测试覆盖不完全（通过新的测试覆盖发现了新的缺陷）
  - 在某些缺陷被修正之前，有些测试根本不能执行，当这些测试可以执行的时候，就发现了新的缺陷。

- 修改的质量＝修改后剩余的缺陷数量（在以前已经工作正常的功能上新引入的或者重新出现的缺陷）。这个度量也称为**重现率**。它测量的是缺陷修正工作给以前运转正常的功能带来的新缺陷，或者破坏了以前运转正常功能的百分比。这种计算所得到的数值提供了一种度量方法，用来反映为响应错误报告而进行的软件缺陷修正工作的质量。

这个度量有助于测试组确定软件修正工作对以前运转正常的功能所产生的负面影响的程度。如果这个数值很高，那么测试组必须要让开发人员知道这一问题。

- 缺陷密度＝一个需求中发现的缺陷总数/为这个需求执行的测试过程数量。缺陷密度表示：某一特定的功能部分或者需求中，平均每一个测试过程所发现的缺陷数。如果某个功能部分的缺陷密度很高，那么需要通过下面几个问题认真分析其原因：
  - 功能是否非常复杂，并且因此预计缺陷密度会较高？
  - 功能的设计或实现是不是有问题？
  - 是否低估了它的难度，所以没有为这个功能分配足够的资源？

此外，当评估缺陷密度时，必须要考虑和缺陷相关的优先级。例如：应用程序的一个需求可能有 50 个低优先级缺陷，但是仍然满足验收标准。然而另一个需求可能只有一个尚未解决的高优先级缺陷，但是它却不满足验收标准。

为了对测试工作的执行进行测量，我们需要收集的度量很多。这里给出的只是很少一部分，还有很多其他可用的度量。为了确定缺陷修正活动的所有需要、突出高风险的部分和最终使测试工作成功执行，本条中描述的度量是需要追踪的核心度量集合。

术 语 表

第 1 章

必要性（necessity）

变更控制委员会（Change Control Board，CCB）

变更请求表（change-request form）

测试（tested）

迭代（iteration）

迭代开发过程（iterative development process）

工程评审委员会（Engineering Review Board）

基线（baseline）

可测试的（testable）

可测试性或者可验证性（testability or verifiability）

可行性（feasibility）

可追溯性（traceability）

明确性（unambiguousness）

瀑布模型（waterfall model）

缺陷预防（defect prevention）

完整性（completeness）

需求（requirement）

需求变更规程（requirement-change process）

需求测试（requirement test）

需求管理工具（requirement-management tool）

验证需求的可测试性（verifying the requirement's testability）

一致性（consistency）

用例（use cases）

优先级（prioritization）

正确性（correctness）

质量测度标准（quality measure）

第 2 章

边界条件测试（boundary-condition testing）

测试策略（test strategies）

测试环境（test environment）

测试设备采购清单（test equipment purchase list）

发行标准（release criteria）

分阶段（phased）

风险（risk）

工作分解结构（work breakdown structure，WBS）

假定（assumption）

开发人员（developer）

设计文档（design document）

生产环境（production environment）

数据流覆盖（data flow coverage）

数据准备（data preparation）

数据字典（data dictionary）

先决条件（prerequisite）

项目风险（project risks）

第 3 章

工作包（work package）

可使用性（usability）

行业专家（subject-matter expert，SME）

第 4 章

错误（error）

调试版本（debug build）

调试模式（debug mode）

调试信息（debug information）

断言（assertion）

发行模式（release mode）

方法（method，也称为函数，function）

跟踪（tracing）

黑箱测试（black-box testing）

灰箱测试（gray-box testing）

模式（modes）

条目（entry）

第 5 章

白箱（white-box）

边界内（in-bounds）

边界上（on-bounds）

边界外（out-of-bounds）

单元测试（unit test）

负面测试（negative testing）

前提功能（precursor function）

商用现货（commercial off-the-shelf，COTS）

烟雾（smoke）

正交排列（orthogonal array）

第 6 章

本地版本（local version）

编译（compiled）

层次（layer）

纯粹的单元测试（pure unit testing）

存根（stub）

存根（stubbed）

错误情况（error cases，也称为异常，exception）

单元测试框架（unit-test framework）

方法（method）

封装（wrap up）

基于接口（interface-based）

极限编程（extreme programming，XP）

接口（interface）

商业处理（business processing）

失败（fail）

数据库抽象（database abstraction）

通过（pass）

用户界面（user interface）

域对象（domain object）

自动生成（automated builds）

组件（component）

第 7 章

可测试性钩（testability hook）

源代码覆盖率分析（source code coverage analysis）

第 8 章

白箱（white-box）

包装（wrap up）

边界值测试（boundary-value testing）

测试包（wrapper）

测试技术（test techniques）

测试外壳（test shell）

等价类划分（equivalence partitioning）

过程管理（process-management）

黑箱（black-box）

灰箱（gray-box）

模块（module）

企业级（enterprise-level）

数据驱动（data driven）

烟雾测试（smoke test）

因果图（cause-effect graphing）

正交排列测试法（orthogonal-array testing）

主要（major）

自动生成（automated builds）

自制测试工具适配器（test-harness adapter）

第 9 章

并发性（concurrency）

分散（scale）

全局（global）

锁（locks）

第 10 章

出口标准（exit criteria）

发行通知（release note）

返测状态（retest status）

粒度（granularity）

缺陷持续时间（defect aging）

缺陷工作流程（defect workflow）

入口标准（entrance criteria）

重现率（recurrence ratio）

3. 其他软件工程类图书

- 《有效用例模式》(影印版) Paul Bramble 等著，定价：28 元
- 《有效用例模式》(中文版) Paul Bramble 等著，即将出版
- 《用例建模》(影印版) Kurt Bittner 等著，定价：28 元
- 《用例建模》(中文版) Kurt Bittner 等著，即将出版

4. Java技术丛书

- 《分布式 Java 2 数据库系统开发指南》 Stewart Birnam 著，定价：36 元
- 《JAVA 开发人员年鉴》(影印版，第 1.4 版，第 1 卷) Patrick Chan 等，定价：68 元
- 《JAVA 开发人员年鉴》(影印版，第 1.4 版，第 2 卷) Patrick Chan 等，定价：68 元

5. 其他系统与编程类图书

- 《MySQL 核心编程》 Leon Atkinson 著，定价：69 元
- 《Windows .NET Server 安全指南》 Cyrus Peikari 著，定价：39 元

6. Deitel编程金典

- 《C++编程金典》(第 3 版) Deitel 等著，定价：118 元
- 《Perl 编程金典》 Deitel 等著，定价：128 元
- 《C#高级程序员指南》 Deitel 等著，定价：118 元
- 《VB .NET 高级程序员指南》 Deitel 等著，即将出版
- 《Python 编程金典》 Deitel 等著，即将出版

7. 算法经典丛书

- 《Java 算法》(第 3 版，第 1 卷，影印版) Robert Sedgewick 著，定价：68 元
- 《Java 算法》(第 3 版，第 1 卷) Robert Sedgewick 著，即将出版
- 《C++算法》(第 3 版，第 2 卷) Robert Sedgewick 著，即将出版

8. 国外经典教材

- 《电子商务网站开发指南》 Vivek Sharma 等著，定价：59 元
- 《数据库系统原理》 Greg Riccardi 著，定价：55 元
- 《电气工程学概论》(第 2 版) John R. Cogdell 著，定价：88 元
- 《网络管理》 Mani Subramanian 著，定价：69 元
- 《微波晶体管放大器分析与设计》(第 2 版) Guillermo Gonzalez 著，定价：59 元
- 《C#解析教程》 Ira Pohl 著，定价：39 元