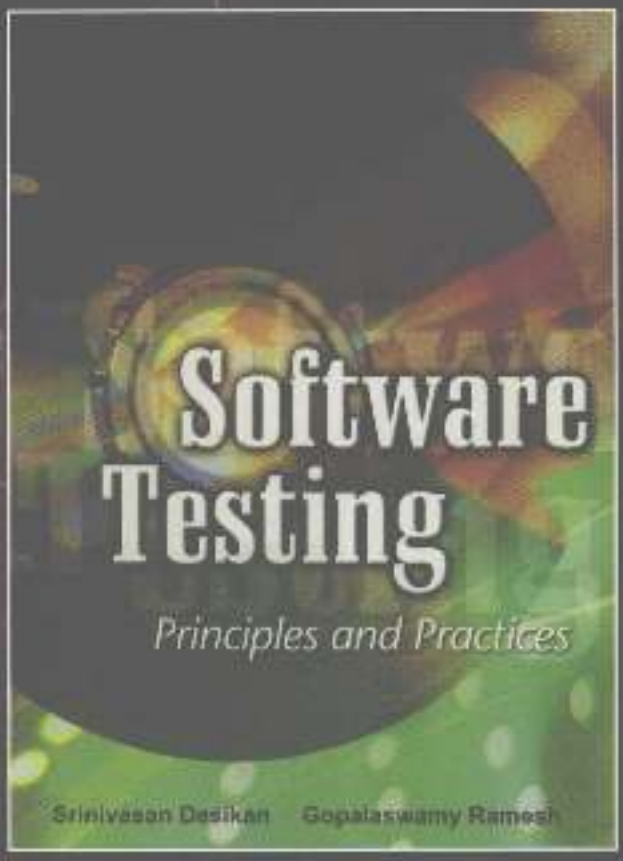
计算机科学丛书

软件测试
原理与实践

（印度）Srinivasan Desikan　Gopalaswamy Ramesh　著　韩柯　李娜　等译



Software Testing
Principles and Practices

机械工业出版社
China Machine Press

软件测试原理与实践

本书从实用的角度对软件测试进行了全面的阐述，讨论了像极限测试和即兴测试这类新兴的领域。

本书特色

- 关注分散在全球地域的团队。讨论全球化团队的人员、组织结构和模型问题。
- 提供印度在测试方面的丰富经验。越来越多的产品测试工作是在印度完成的，但是研究印度经验或印度业务模型的专著却很少，本书通过实例讨论了印度的最佳测试实践。
- 在保持完整的理论体系基础上，强调实践经验。本书在介绍诸如等价类划分和圈复杂度等传统方法的同时，还讨论了测试的一些实际问题，例如国际化测试和回归测试。

作者简介

Srinivasan Desikan 印度班加罗尔市的西贝尔系统公司质量工程部主任，具有16年的产品测试经验，所测试的产品正在被全世界数以百万计的客户使用。Srinivasan在测试自动化、测试管理、测试过程和测试团队建立等方面有丰富的经验。他经常在国际测试会议上发表演讲，定期在多所大学举办讲座。Srinivasan是印度技术学院的客座教授，并担任多家印度测试公司的荣誉董事主任。

Gopalaswamy Ramesh 独立咨询师，班加罗尔信息技术国际学院的副教授，在印度和海外具有25年的业界经验，曾经担任过Oracle印度开发中心的高级主任。他还是获得过印度国家奖的畅销书《管理全球化软件项目》的作者。目前，他在为多家印度和海外公司提供项目管理和相关领域的咨询服务。

Authorized for sale and distribution in the People's Republic of China Exclusively (Except Taiwan, Hong Kong SAR and Macau SAR)

本书仅限在中华人民共和国境内（不包括中国香港、澳门特别行政区及中国台湾）销售。

www.PearsonEd.com

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站：http://www.hzbook.com

网上购书：www.china-pub.com

ISBN 978-7-111-25506-2

9 787111 255062

定价：45.00元

计 算 机 科 学 丛 书

软件测试
原理与实践

（印度）Srinivasan Desikan Gopalaswamy Ramesh 著 韩柯 李娜 等译

Software Testing

Principles and Practices

机械工业出版社
China Machine Press

本书全面论述了软件测试的基本原理和最佳实践，介绍了最近出现的极限测试和即兴测试等新的测试方法，本书介绍了全球团队的个人、组织结构和模型等问题。在介绍综合性理论知识的同时，强调实际经验。本书在介绍黑盒测试和白盒测试等传统方法的同时，还介绍了测试的很多实际问题，例如国际化测试和回归测试等。

本书的突出特点是从工程实践的角度，比较全面地讨论棘手问题的具体应对方法和相应的风险，还在比较高的层次上讨论软件测试工程的整体把握方法。全书在各章附有许多实际问题的思考题，帮助读者更深刻地理解这些现实问题。

本书可作为高等院校软件工程和测试方面相关课程教材，对软件开发和测试人员解决实际问题也有较高的参考价值。

Authorized translation from the English language edition, entitled Software Testing Principles and Practices, 9788177581218 by Desikan, Srinivasan; Ramesh, Gopalaswamy, published by Dorling Kindersley (India) Pvt. Ltd., publishing as Pearson Education. Copyright © 2006 Dorling Kindersley (India) Pvt. Ltd.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2009.

This edition is manufactured in the People's Republic of China, and is authorized for sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

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。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2008-4823

图书在版编目（CIP）数据

软件测试原理与实践/（印）迪西甘（Desikan, S.）等著，韩柯等译. —北京：机械工业出版社，2009.2

（计算机科学丛书）

书名原文：Software Testing: Principles and Practices

ISBN 978-7-111-25506-2

Ⅰ.软… Ⅱ.迪… 韩柯… Ⅲ.软件—测试 Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2008）第173215号

机械工业出版社（北京市百万庄大街22号 邮政编码 100037）

责任编辑：王　千

三河市宏达印刷有限公司印刷

2009年2月第1版第1次印刷

184mm×260mm · 19印张（含0.25印张彩插）

标准书号：ISBN 978-7-111-25506-2

定价：45.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

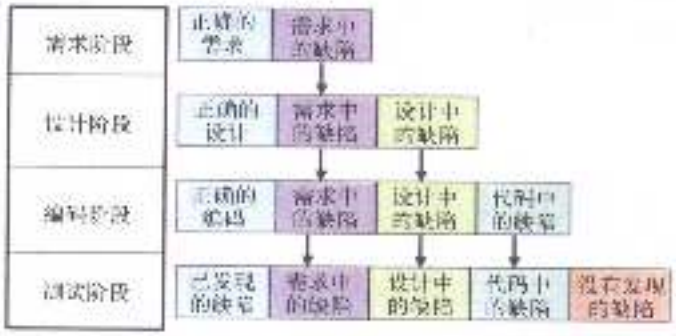


图1-2 早期阶段产生的缺陷如何使成本增加

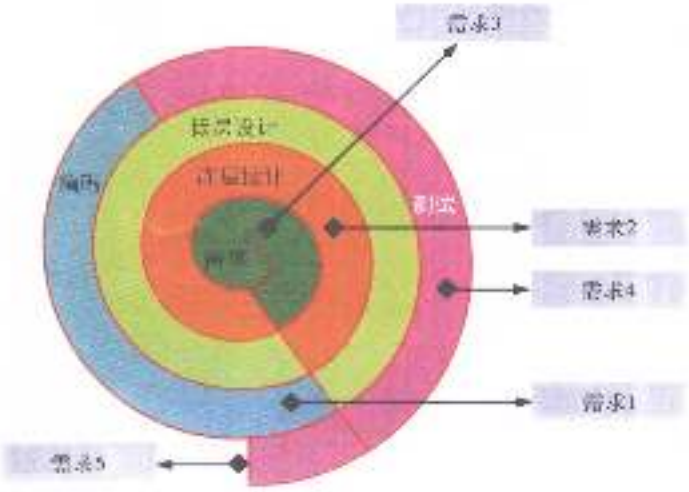


图2-3 螺旋模型

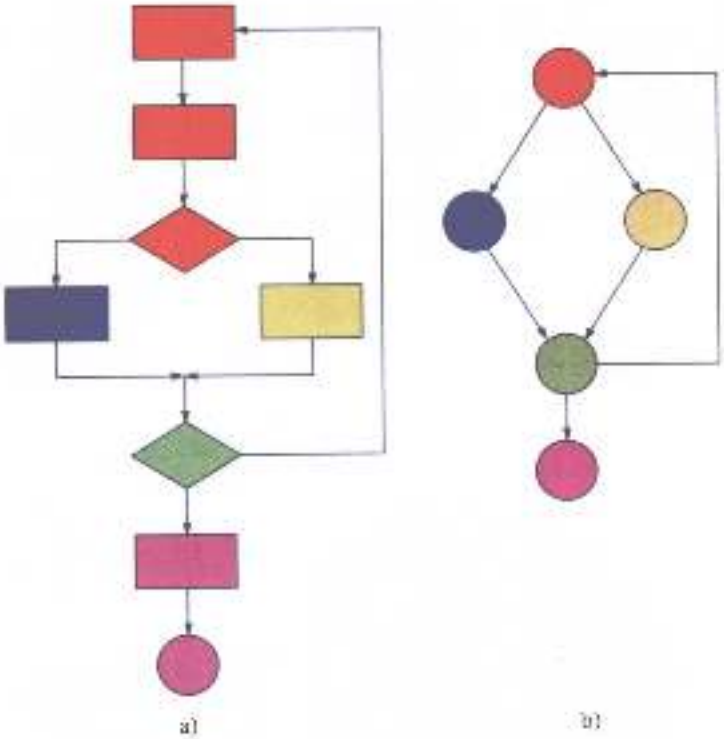
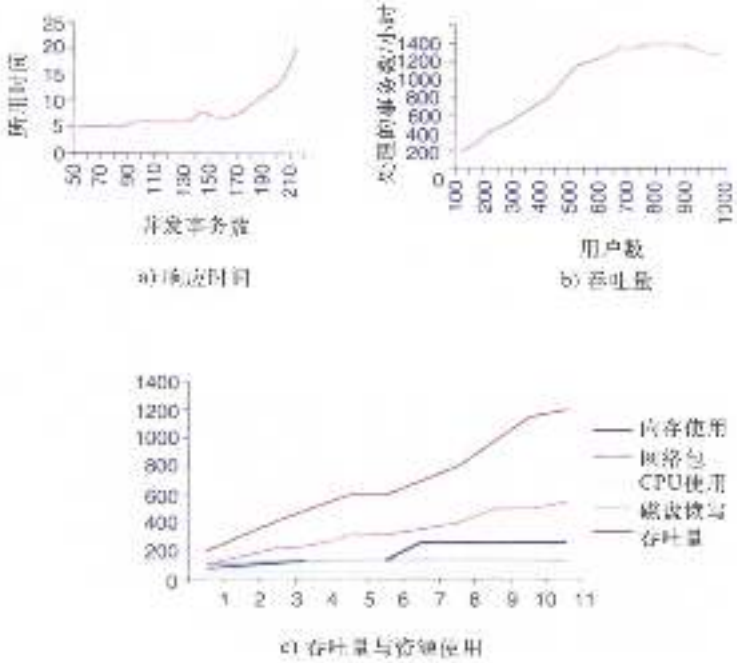


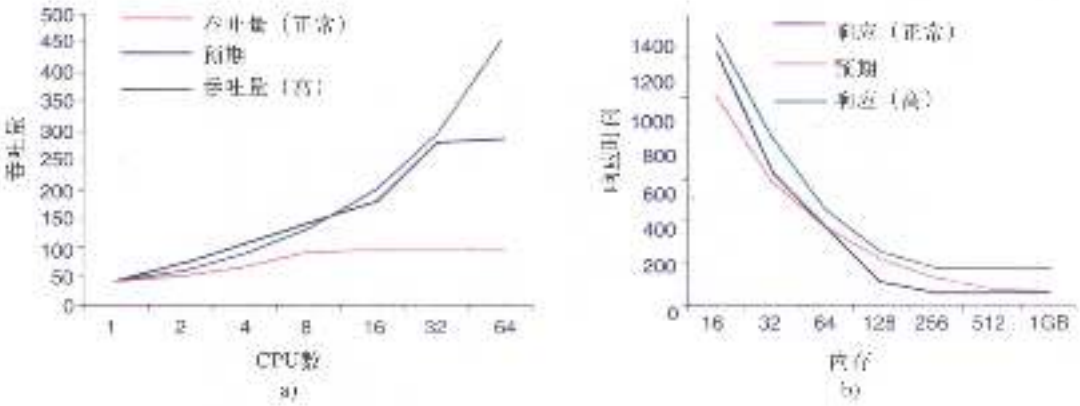
图3-4 将传统流程图转换为可计算复杂度的流程图



a) 响应时间　　b) 吞吐量

c) 吞吐量与资源使用

图7-3　图表绘制举例



a)　　b)

图7-4　性能调优结果曲线

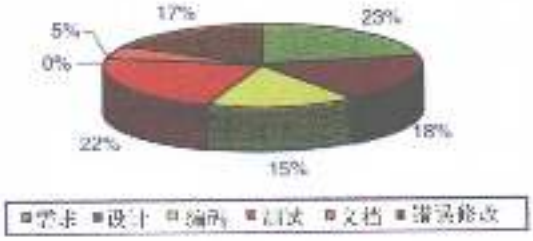


图17-5 实际投入分布

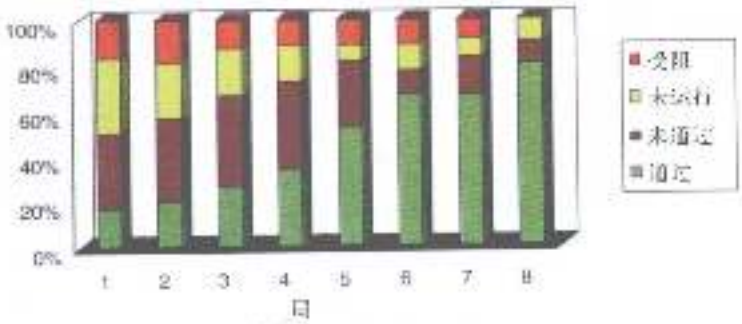


图17-6 测试用例执行进展

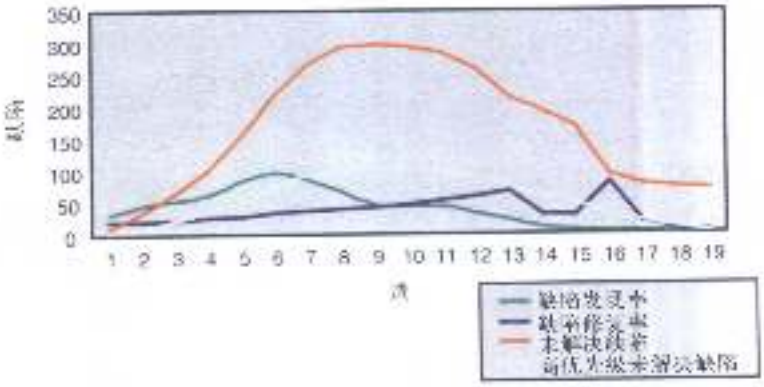


图17-8 缺陷趋势

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出，独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作，而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子信箱：hzjsj@hzbook.com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育

译者序

近年来，已经有不少关于软件测试方面的专著和教材介绍到我国，我国软件测试行业的学者和资深人士撰写的专著也已面市。但是，目前软件测试行业仍处于高度依赖个人经验和直觉的状态，教材上介绍的测试方法多种多样，而如何选择和组合运用来达到更好的效果，在很大程度上依靠个人的经验。此外，软件测试的成果并没有很可靠的、普遍适用的评价准则，经过测试的软件所残留的缺陷并没有可靠的方法验证。由于决定软件测试项目质量的因素有很多，因此不同的测试项目之间通常并不具备很强的可比性。所以，对于软件测试人员来说，关键的并不在于掌握多少测试技巧，而在于通过大量的实践和思考，对软件测试有怎样的理解和理念，在于怎样运用这些技巧。

本书的突出特点是不回避软件测试普遍存在的实际问题（例如时间压力、风险压力、人员管理问题等），从工程实践的角度，比较全面地讨论了这些棘手问题的具体应对方法和相应的风险，站在比较高的层次上全面地讨论了软件测试工程的整体把握方法。全书的主题设置和论述体现出作者在软件测试和软件工程领域有丰富的实践经验，每章所附的思考题大多是实际中遇到的问题，通过对这些问题的认真思考，相信读者会对这些现实问题有更深刻的理解。

在翻译过程中，除了对原文的个别错误进行了相应的更正外，我们力求忠实于原文。但是，由于译者的知识水平和实际工作经验有限，不当之处在所难免，恳请读者批评指正。参加本书翻译、审校和其他辅助工作的还有李沛沛、黄慧菊、耿民和周健。

译者

2008年3月

序

软件测试已经扩展到更宽的领域，并显现出其重要性。正如医药公司在新药发布之前被要求宣布他们如何测试该药品一样，由于顾客需要无缺陷的软件产品，相关管理部门也要对将发布的软件进行彻底的测试。市面上的大部分专著都是理论方面的，很少有针对实际问题的。本书供选择测试作为职业的学生学习及从业人员参考，他们需要务实和实用的测试视角，以及人员、过程和技术之间的正确平衡。两者结合形成了本书的基础——软件测试原理与实践。

顾名思义，我们强调原理和实践。本书的素材已被多所大学，比如印度钦奈的安娜大学和班加罗尔的国际信息技术学院进行"β测试"，班加罗尔还将本书用作信息技术学院学生的教材以及其软件工程卓越中心（Center of Excellence）的参考用书。本书的一些概念已通过国际会议和作者的客座演讲提供给从业者。

我们增加了管理地理上分散的团队的章节，这尤其适用于跨国公司，他们的团队分散在世界不同的大洲协调同步开发，测试和交付产品给全球的客户。本书也包括了像极限测试、即兴测试等顺应新趋势的测试。

本书的内容可供从业者理解测试行业的最新状态。新增的关于指标与度量、测试策划、测试管理及测试自动化等章节，帮助从业者在工作中采用这些概念。我们是在研究了不同大学的教学大纲后编制此书的，因此学术上是十分严谨的。

非常感谢印度钦奈eFunds公司的测试经理Gayathri Chandrasekar女士，本书许多章节都有她的贡献。

Srinivasan Desikan
Gopalaswamy Ramesh

致 谢

首先我要感谢我曾经工作过的各个公司（Wipro公司，Novell公司和Talisma公司），他们提供了机会和基础设施，使我在工作中学习到实用的测试技术。我还要感谢测试专业人士，他们为完成这本书，在全球各地为我提供见解和信息。最后也是最重要的，我想感谢我的妻子、儿子和女儿，感谢他们的牺牲和支持。大家可以通过srinivasan.desikan@gmail.com与我联系。

Srinivasan Desikan

我要感谢我的导师Mahabala教授，他是软件测试领域的佼佼者，是他一直激励我向前奋进。他是我过去二十年来灵感的来源。我还要感谢安娜大学计算机科学系和班加罗尔国际信息技术学院，他们让我讲授软件测试课程，这是我学习的源泉。最后，我想感谢支持我的家人，没有她们，我就不会一直努力到本书的面世。大家可以通过gopalaswamy_ramesh@yahoo.com与我联系。

Gopalaswamy Ramesh

前言

当今世界，软件变得无处不在，消费者对此的期望大幅度增加，“软件出错很正常，我们必须接受这个事实”的旧观点已经不再适用。如今，人们期望软件能每时每刻地正常运行，并且要满足客户不断变化的需求。早先，软件系统用于后台管理服务和非关键业务的操作；现在，越来越多的关键应用都在全球实现。对无差错运行软件期望的提高，导致了对软件供应商高品质产品需求的增长。反过来，在过去十几年中，市场关注的焦点已从单纯的编程和并发转向更全面的目标：生产的软件一直正常工作，因而更加关注对软件的测试。

在过去的十几年中，测试已经引起人们的极大兴趣。以下列举一些事实：

- 测试工作量的成倍增加提供了广阔的职业发展机会；
- 测试相关职位的薪资正在上升；
- 测试已经成为重要的外包机会；
- 过去五年，关于测试的会议和其他类似活动有了明显的增加；
- 越来越多的专业人士考虑把软件测试作为职业。

测试的工作方法为了跟上需求的增加也经历了彻底的改变。首先，全球化是持久的。现在的组织利用地域时差和全球的人才，向各大洲派遣开发和测试团队，这些团队共同无缝地工作。为了成功地推动这种新的全球化运作方式，公司必须掌握异地分布团队这项工作的艺术。其次，测试已经从即兴和偶然的尝试，转变为一项系统的、有计划的活动，以完成所有过程并且通过科学的度量。第三，现在成功的测试需要谨慎地利用各种技术，满足产品上市时间的要求。贯穿于测试生命周期的测试自动化也已经成为必需品而不是奢侈品。最后，人们对测试职业的看法也经历了一种巨变——成功的公司应为测试专业人士提供职业发展途径，鼓励他们发挥聪明才智，以保证他们能够长期供职于该公司和从事软件测试这个职业。

本书及时满足了测试从业人员的需要，同时鼓舞了有志投身测试行业的专业人士和学生。Ramesh和Srinivasan已经把他们40年的实际从业经验带给了大家。本书包含了某些对行业成功至关重要的方面，而这些方面在其他书中很少涉及，例如：

- 理论和实践之间的平衡：很多书都努力给出严谨的理论，试图把现实中的问题简单化。而本书作者坚持从现实世界存在的事实着手，提供解决现实问题所需的理论基础。
- 人员、过程和技术问题之间的平衡：成功的公司有条不紊地工作，充分利用技术并且提升人员的能力。作者成功地抵制住了以牺牲实用性为代价，只讨论“很酷的技术问题”的诱惑。例如，本书讨论了像自动化（技术密集）、人和组织结构（以人为本）、测试组织和报告（注重过程）这样的问题。
- 从从业者务实的观点展示出一个坚实的基础：作者广泛地讨论了各种不同类型的测试。例如，详细讨论了像国际化这样难懂的主题。
- 本书是我知道的第一本意识到并清楚提出全球化的重要性、明确讨论全球化的团队结构和因此带来的各种可能问题的书。这清晰地表明作者作为全球化软件测试团队的主要领导所积累的管理经验。
- 本书涵盖了一些公认的测试方法，这些测试方法已多次在世界各地的国际测试会议上提出并被接受。

除了以上这些，每章最后所附的实用练习向读者揭示了测试的真谛。作者全学期测试课程的教学经验在各种练习中得到充分体现。我坚信，随着本书的出版，这类课程会推广到各类高校，使软件测试形成一门学科，拓宽并发更具竞争力的测试职业网络。

我希望本书及其作者不断营造一种环境的努力获得成功。在这种环境中，软件测试被认为是任何软件开发生存周期的一个关键阶段，测试职业被认为具有很高价值，测试被发展为一门工程学科。

Vikram Shah

Vikram Shah目前是Silver Software和IT-People管理委员会的主任，是BiTES（Karnataka州政府成立的IT教育标准管理委员会）的一名积极成员。Vikram Shah具有30多年的业界经验，在多家公司（例如Mahindra British Telecom、Novell、Andiamo Software和Talisma Software）担任过CEO和M.D.。

目录

出版者寄语
译者序
序
前言

第一部分 写作线索

第1章 测试原理……2
1.1 生产软件中的测试背景……2
1.2 本章介绍……3
1.3 不完善的平……3
1.4 Dijkstra定律……4
1.5 及时测试……5
1.6 羊人和猫……6
1.7 首先测试测试用例……7
1.8 杀虫剂悖论……7
1.9 护航舰队与商船……8
1.10 桥上的警察……9
1.11 钟摆的终结……9
1.12 黑衣人……11
1.13 自动化综合症……12
1.14 小结……13
第2章 软件开发生存周期模型……15
2.1 软件项目的阶段……15
2.1.1 需求获取和分析……15
2.1.2 规划……15
2.1.3 设计……15
2.1.4 开发或编码……15
2.1.5 测试……16
2.1.6 部署和维护……16
2.2 质量、质量保证和质量控制……16
2.3 测试、验证和确认……17
2.4 表示不同阶段的过渡模型……18
2.5 生存期模型……19
2.5.1 瀑布模型……19
2.5.2 原型和快速应用开发模型……20
2.5.3 螺旋或迭代模型……22
2.5.4 V字模型……22
2.5.5 改进型V字模型……24
2.5.6 各种生存期模型的比较……26

第二部分 测试类型

第3章 白盒测试……30
3.1 白盒测试的定义……30
3.2 静态测试……30
3.2.1 人工静态测试……31
3.2.2 静态分析工具……33
3.3 结构测试……35
3.3.1 单元/代码功能测试……35
3.3.2 代码覆盖测试……36
3.3.3 代码复杂度测试……40
3.4 白盒测试中的挑战……43
第4章 黑盒测试……47
4.1 黑盒测试的定义……47
4.2 黑盒测试的意义……48
4.3 黑盒测试的时机……48
4.4 黑盒测试的方法……48
4.4.1 基于需求的测试……49
4.4.2 正面和负面测试……52
4.4.3 边界值分析……53
4.4.4 决策表……56
4.4.5 等价划分……57
4.4.6 基于状态或工作流的测试……59
4.4.7 兼容性测试……61
4.4.8 用户文档测试……63
4.4.9 领域测试……64
4.5 小结……66
第5章 集成测试……68
5.1 集成测试的定义……68
5.2 集成测试作为一种测试类型……68
5.2.1 自顶向下集成……70

5.2.2 自底向上集成 ……71
5.2.3 双向集成 ……72
5.2.4 系统集成 ……73
5.2.5 选择集成方法 ……73
5.3 集成测试作为一个测试阶段 ……74
5.4 场景测试 ……74
5.4.1 系统场景 ……74
5.4.2 用户场景 ……75
5.5 使用回归 ……77
5.5.1 选择缺陷回归的深度和持续时间 ……77
5.5.2 选择合适的产品版本 ……77
5.5.3 对使用回归的目标进行沟通 ……78
5.5.4 建立和监视实验室 ……78
5.5.5 采取行动解决问题 ……78
5.5.6 优化缺陷回归所涉及的工作 ……78
5.6 小结 ……79
第6章 系统测试和确认测试 ……80
6.1 系统测试概述 ……80
6.2 实施系统测试的原因 ……81
6.3 功能测试与非功能测试 ……82
6.4 功能系统测试 ……83
6.4.1 设计体系结构验证 ……84
6.4.2 业务垂直测试 ……84
6.4.3 部署测试 ……85
6.4.4 兼容测试 ……86
6.4.5 符合性的认证、标准和测试 ……87
6.5 非功能系统测试 ……88
6.5.1 设置配置 ……88
6.5.2 提供法人与出生准则 ……89
6.5.3 平衡关键资源 ……89
6.5.4 可伸缩性测试 ……90
6.5.5 可靠性测试 ……92
6.5.6 压力测试 ……95
6.5.7 互操作性测试 ……97
6.6 确认测试 ……98
6.6.1 确认准则 ……99
6.6.2 选择确认测试的测试用例 ……99
6.6.3 执行确认测试 ……100
6.7 测试阶段小结 ……101
6.7.1 多阶段测试模型 ……101
6.7.2 多个发布版本的矩阵 ……103
6.7.3 谁负责实施与何时实施 ……103
第7章 性能测试 ……106
7.1 引言 ……106
7.2 决定性能测试的要素 ……106
7.3 性能测试的方法论 ……108
7.3.1 收集需求 ……108
7.3.2 编写测试用例 ……110
7.3.3 自动化性能测试用例 ……110
7.3.4 执行性能测试用例 ……111
7.3.5 分析性能测试结果 ……112
7.3.6 性能调谐 ……113
7.3.7 性能基准测试 ……115
7.3.8 能力策划 ……116
7.4 性能测试工具 ……116
7.5 性能测试的过程 ……117
7.6 摘要 ……118
第8章 回归测试 ……120
8.1 回归测试的定义 ……120
8.2 回归测试的类型 ……120
8.3 回归测试的阶段 ……121
8.4 回归测试的方法 ……122
8.4.1 实施第一次"冒烟"或"烧瓶"测试 ……123
8.4.2 理解选择测试用例的准则 ……123
8.4.3 测试用例分类 ……124
8.4.4 选择测试用例的方法论 ……125
8.4.5 重新设置测试用例以进行回归测试 ……126
8.4.6 总结回归测试的结果 ……128
8.5 回归测试的最佳实践 ……129
第9章 国际化（I18n）测试 ……131
9.1 引言 ……131
9.2 国际化介绍 ……131
9.2.1 语言的定义 ……131
9.2.2 字符集 ……131
9.2.3 区域 ……132
9.2.4 本章使用的术语 ……132
9.3 国际化测试的测试阶段 ……133
9.4 有效性测试 ……134

9.5 黑盒测试 ……135
9.6 国际化确认 ……136
9.7 假语言测试 ……137
9.8 语言测试 ……138
9.9 本地化测试 ……138
9.10 国际化使用的工具 ……140
9.11 挑战与问题 ……140
第10章 探索测试 ……142
10.1 探索测试概述 ……142
10.2 失控测试 ……144
10.3 结对测试 ……145
10.4 探索式测试 ……146
10.5 迭代式测试 ……148
10.6 敏捷与极限测试 ……149
10.6.1 XP工作流 ……150
10.6.2 通过例子进行小结 ……152
10.7 缺陷播种 ……153
10.8 小结 ……153

第三部分 特殊测试专题

第11章 面向对象系统的测试 ……156
11.1 引言 ……156
11.2 面向对象软件入门 ……156
11.3 面向对象测试的类别 ……161
11.3.1 一组类的单元测试 ……161
11.3.2 将类组合在一起——集成测试 ……164
11.3.3 面向对象系统的系统测试与互操作 ……165
11.3.4 面向对象系统的回归测试 ……165
11.3.5 面向对象系统的测试工具 ……165
11.3.6 小结 ……167
第12章 可使用性与易获得性测试 ……169
12.1 可使用性测试的定义 ……169
12.2 可使用性测试的途径 ……170
12.3 可使用性测试的时机 ……171
12.4 实现可使用性的方法 ……173
12.5 可使用性的质量因素 ……174
12.6 美观测试 ……175
12.7 易获得性测试 ……175
12.7.1 基本易获得性 ……176
12.7.2 产品易获得性 ……177
12.8 可使用性工具 ……179
12.9 可使用性实验室的建立 ……180
12.10 可使用性的测试角色 ……181
12.11 小结 ……182

第四部分 测试中的人员和组织问题

第13章 常见人员问题 ……184
13.1 关于测试的错觉和错误概念 ……184
13.1.1 “测试没有什么技术挑战” ……184
13.1.2 “测试没有为我提供职业成长道路” ……185
13.1.3 “我被派来测试——我到底怎么了?!” ……186
13.1.4 “这些人是我的对手” ……186
13.1.5 “测试是如果我有时间最终会做的工作” ……186
13.1.6 “测试的拥有者毫无意义” ……187
13.1.7 “测试只是破坏” ……187
13.2 测试与开发工作的比较 ……188
13.3 为测试人员提供职业发展道路 ……188
13.4 生态系统的角色与行动要求 ……192
13.4.1 教育系统的角色 ……192
13.4.2 高级管理层的角色 ……193
13.4.3 测试界的角色 ……194
第14章 测试团队的组织结构 ……196
14.1 组织结构的要素 ……196
14.2 单产品公司的结构 ……196
14.2.1 单产品公司的测试团队结构 ……197
14.2.2 按组件组织的测试团队 ……199
14.3 多产品公司的结构 ……199
14.3.1 测试团队作为“首席技术官办公室”的一部分 ……200
14.3.2 针对所有产品的单一测试团队 ……201
14.3.3 按产品组织的测试团队 ……201
14.3.4 针对不同测试阶段的独立测试团队 ……201
14.3.5 混合模型 ……202
14.4 全球化与地域分散的团队对产品测试的影响 ……202

14.4.1 全球化的业务影响 ……202
14.4.2 全时区开发/测试团队模型 ……203
14.4.3 测试能力中心模型 ……204
14.4.4 全球团队面临的挑战 ……205
14.5 测试服务公司 ……206
14.5.1 测试服务的业务需求 ……206
14.5.2 测试作为一种服务与产品测试公司之间的差别 ……207
14.5.3 测试服务公司的典型角色和责任 ……208
14.5.4 测试服务公司面临的挑战与问题 ……209
14.6 测试公司的成功因素 ……211

第五部分 测试管理与自动化

第15章 测试策划、管理、执行与报告 ……214
15.1 引言 ……214
15.2 测试策划 ……214
15.2.1 准备测试计划 ……214
15.2.2 范围管理：决定要测试和不测试的特性 ……214
15.2.3 确定测试方法和策略 ……215
15.2.4 确定测试准则 ……216
15.2.5 确定责任、人员和培训计划 ……216
15.2.6 确定资源需求 ……216
15.2.7 确定测试的可交付产品 ……217
15.2.8 测试任务：规模与工作量估计 ……217
15.2.9 活动分解与进度估计 ……219
15.2.10 沟通管理 ……220
15.2.11 风险管理 ……220
15.3 测试管理 ……223
15.3.1 标准的选择 ……223
15.3.2 测试基础设施管理 ……225
15.3.3 测试人员管理 ……227
15.3.4 与产品发布集成 ……227
15.4 测试过程 ……228
15.4.1 把各种要素放在一起并确定测试计划基线 ……228
15.4.2 测试用例规格说明 ……229
15.4.3 可跟踪性矩阵的更新 ……229
15.4.4 确定有可能实现自动化的测试用例 ……229
15.4.5 测试用例的开发和基线确立 ……229
15.4.6 测试用例的执行与可跟踪性矩阵的更新 ……229
15.4.7 指标的采集与分析 ……230
15.4.8 准备测试总结报告 ……230
15.4.9 推荐产品发布准则 ……230
15.5 测试报告 ……230
15.6 最佳实践 ……231
15.6.1 与过程相关的最佳实践 ……232
15.6.2 与人员相关的最佳实践 ……232
15.6.3 与技术相关的最佳实践 ……232
附录A：测试策划检查单 ……233
附录B：测试计划模板 ……234
第16章 软件测试自动化 ……237
16.1 测试自动化的定义 ……237
16.2 自动化使用的术语 ……238
16.3 自动化所需的技能 ……239
16.4 自动化的对象与范围 ……240
16.4.1 确定自动化负责的测试类型 ……241
16.4.2 自动化不太可能变更的部分 ……241
16.4.3 自动化测试符合标准 ……241
16.4.4 自动化的管理问题 ……242
16.5 自动化的设计和体系结构 ……242
16.5.1 外部模块 ……243
16.5.2 场景与配置文件模块 ……243
16.5.3 测试用例与测试框架模块 ……243
16.5.4 工具与结果模块 ……244
16.5.5 报告生成器与报告/指标模块 ……244
16.6 测试工具/框架的一般需求 ……244
16.7 自动化的过程模型 ……250
16.8 测试工具的选择 ……252
16.8.1 选择测试工具的准则 ……253
16.8.2 工具选择与部署步骤 ……254
16.9 使用原型模型的自动化 ……255
16.10 自动化中的挑战 ……255
16.11 小结 ……255
第17章 测试指标和度量 ……258
17.1 指标和度量的定义 ……258
17.2 测试中指标的意义 ……260
17.3 指标类型 ……262
17.4 项目指标 ……262

17.4.1 投入偏差（计划投入与实际投入）……263
17.4.2 计划偏差（计划与实际）……264
17.4.3 不同阶段内的投入分布……265
17.5 进度指标……266
17.5.1 测试量与指标……267
17.5.2 开发缺陷指标……273
17.6 生产力指标……277
17.6.1 每100小时测试发现的缺陷数……277
17.6.2 每100小时的测试用例执行数……278
17.6.3 每100小时的测试开发测试用例数……278
17.6.4 每100个测试用例发现的缺陷数……279
17.6.5 每100个失败的测试用例缺陷数……279
17.6.6 测试阶段有效率……280
17.6.7 已关闭缺陷的分布……280
17.7 发布指标……281
17.8 小结……282
参考文献……284

第一部分 写作线索

本部分为全书制定的写作线索。第1章讨论最近几年软件行业的变化，以及这些变化对软件质量提出的要求。本书将这些要求转换为十一个基本原理，以此作为本书其他各章的中心。第2章定义验证、确认、质量保证和质量控制等关键术语，然后详细讨论各种生存周期模型，以及这些模型对验证和确认活动的意义。本部分还要介绍关于进入和退出准则的概念，这对于理解书中后面关于测试的不同阶段是非常必要的。

第1章 测试原理

1.1 生产软件中的测试背景

我们今天使用的几乎一切东西都包含软件。在软件发展的早期，软件用户的数量与大公司相比还是很少的。现在，一个典型的工作场所（或家里），差不多每个人都在使用计算机及软件。管理人员使用生产率很高的办公软件（代替以前的打字机）。会计师及财务人员使用电子表格软件和其他财务软件包，比使用计算器（甚至手工）要快得多。公司和家里的每个人都用电子邮件和互联网进行娱乐、教育、通信和交互，获取任何想要的信息。另外，"技术"人员使用程序设计语言、建模工具、仿真工具和数据库管理系统完成以前主要靠手工完成的任务。

除了上帝，我们都要测试。

上面只是说明软件的使用对于用户来说"很明显"的几个例子。但是，软件的无处不在和广泛普及远不止以上这些例子所揭示的那样，现在的软件就像20世纪初的电一样普及。我们在办公室和家庭所使用的几乎每一台设备都嵌入大量的软件，例如手机、电视、手表和冰箱以及厨房的每一件电器都有嵌入式软件。

另一个值得注意的现象是软件在任务关键场合的使用，在这些场合出现失效是根本不能接受的。对于心脏起搏器软件，决不能提议"请关机并重启系统"！我们离不开的几乎所有服务中都有软件。银行、航空管制、汽车等，驱动它们的软件都是绝对不能失效的。这些软件系统必须每时每刻、永久、可靠、可预见地运行。

这些无所不在、广泛使用和关键之处都对软件的开发和部署提出了一定的要求。

首先，开发软件产品或提供服务的公司必须尽全力减少，最好消除每件所交付的软件产品或服务中的缺陷。用户越来越不能容忍劣质的软件产品。从软件开发公司的角度看，发布有缺陷的软件产品在经济上也不是可行的。比如，在电视机发运给成千上万的用户后，其中的嵌入式软件被发现有一个缺陷。怎么可能发送"补丁"给这些用户，要求他们"安装补丁"？因此，唯一的解决方案是在产品交付用户之前就一次做好。

第二，缺陷不可能隐藏很久。当用户数量有限，并且使用产品的方式是可预期的（高度受限的），软件产品中存在的缺陷可能发现不了，并隐藏很长时间。但是，随着用户数量的增加，缺陷不被发现的几率越来越小。如果产品中存在缺陷，总有一天有人会发现它。

第三，一个软件产品或服务的使用特性正在变得越来越不可预知。如果软件是为一个特定公司的特定功能（例如工资管理软件包）专门开发的，那么产品的使用特性是可以预期的。例如，用户只能执行定制软件所提供的特定功能。此外，软件的开发者了解用户、了解用户的业务活动及操作。另一方面，请考虑驻留在互联网上的一般应用程序。开发者没办法控制用户的操作。用户可能运行没有测试过的功能，可能使用的是不合适的硬件或软件环境，可能没有受过完整培训就直接以不正确的方式或没有目的地操作软件。尽管存在所有这些"误操作"，软件应该仍能正确运行。

最后，对每一个缺陷的结果和影响都要进行分析，尤其是关键任务应用程序。软件发布时可以接受的是99.9%的缺陷修复率，只遗留0.1%的缺陷。看起来对于发布的软件这是一个不错的统计数字。可是，如果我们在关键任务应用软件中留下了0.1%的缺陷，那么统计数据将如下所示。

- 每周会有10 000个不合格的外科手术。
- 每天会有3架飞机坠毁。
- 每周有5个小时没有电。

确实，上述数据对于任何个人、组织或政府都是不可接受的。我们提出一种补救措施，例如"万一着火，请穿这件衣服"，或者在文档中写明失效，例如"假如飞机着陆有误，你可能只会损失部分器官"，这些对于关键任务应用程序来说都是不可接受的。

本书关注软件测试。在传统意义上，测试被狭义地定义为测试程序代码。我们更愿意采用一种更广义的测试定义：它包含所有生产优质产品的相关活动。生产软件产品必须有几个阶段（例如需求获取、设计和编码），除此之外还有测试（传统意义的测试）。虽然测试肯定是促成产品高质量的因素之一（也是一个阶段），但是测试本身却不能提高产品的质量。对于一个好的产品，测试与其他阶段恰当的交互是必不可少的。图1-1描述了这些交互及其产生的影响。

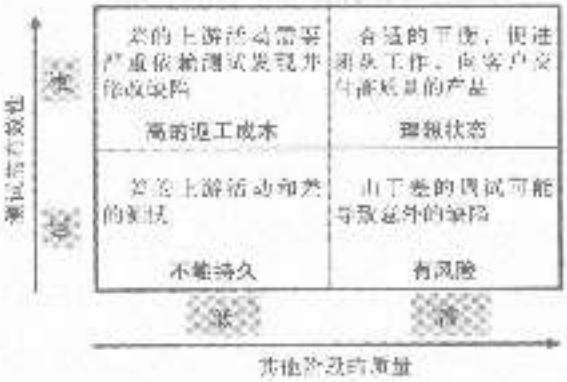


图1-1 测试的有效性与其他阶段质量的关系

如果其他阶段的质量较低，测试的有效性也很低（图1-1左下角的情况），这种情况是不能持久的。因为这样的产品将很快被淘汰。其他阶段质量很低而试图通过加强测试来弥补（图1-1的左上角）可能增加每个人的压力，尤其是产品将要发布时发现了缺陷。类似地，盲目地相信其他阶段的高质量从而进行较少的测试（图1-1的右下角），当最后一刻发现缺陷时将导致不可预见的危险情形。理想的状态是包括测试在内的所有阶段的高质量（图1-1的右上角）。这种情况下，用户会感到高质量带来的好处，这将激发团队更好地工作，公司获得成功。

1.2 本章介绍

本章讨论测试的一些基本原理。我们认为这些原理对于理解测试目的，给用户提供高质量的产品来说是十分重要的。这些原理同时构成本书其余部分的基础。本章是本书的核心。

测试的基本原理有：

1. 测试的目标是在用户发现缺陷之前找到它们。
2. 穷尽的测试是不可能的，程序测试只能说明缺陷的存在，决不能说明没有缺陷。
3. 测试贯穿于全部软件生存周期，并且不是周期结束前的最后一个活动。
4. 理解测试背后的原因。
5. 首先测试测试用例。
6. 测试用例需要逐步完善、不断修订。
7. 缺陷成群集中出现，因此测试应该关注这些缺陷簇。

8. 测试包括缺陷预防。

9. 测试是缺陷预防和缺陷检测之间的精心平衡。

10. 智能的和经过良好策划的自动化是实现测试效益的关键。

11. 测试需要具有天分、具有自信和信任团队的非常投入的人才。

我们将在随后的章节中阐述这些原理。我们在适当的时候将用一个信息技术领域以外的小故事阐明这些原理，以帮助读者理解。

1.3　不完善的车

汽车销售员："这辆车是完好的，你只需要刷漆即可。"

不管开发什么软件，最终都要满足客户的需要。除此之外的一切都是次要的。测试是确保产品满足客户需要的一种手段。

我们要给"客户"下个广泛的定义。客户不仅指外部客户，也包括内部客户。例如，如果产品是用公司内部不同小组开发的组件组成的，那么这些不同组件的用户就应该被认为是客户，即使他们来自同一个公司。这种客户视角可以提高包括测试在内的所有活动的质量。

我们把内部客户的概念作进一步拓展：开发团队认为测试团队是其内部客户。这样可以保证产品的构建不仅针对使用需求，而且针对测试需求。这种观念可提高产品的"可测试性"，还可改善开发团队和测试团队之间的交互。

销售代表/工程师："这部车有最好的变速器和制动器，它从静止加速到每小时80英里不到20秒！"

顾客："哦，这有可能是真的，但是不变的是当我踩刹车的时候车速却变得更快了！"

我们希望读者不要最后把这两个观点，即客户观点和质量观点，看作是附加的观点，而应该贯穿到本书从头到尾讨论的所有活动和组件中。

如果我们的工作是给客户提供一部完好的车（并且不要求用户刷漆），我们要确保车如用户所期望的那样工作，没有任何（大的）毛病，那么就要保证我们自己找到并改正车的所有缺陷，这就是测试的根本目标，我们在做任何测试工作时都必须记住这一点。

测试应该在用户没有发现缺陷前找到它们。

1.4　Dijkstra定律

假设一段接受六个字符密码的程序，并保证第一个字符必须是数字，其余字符是字符数字型的。如果我们的目标是穷尽测试，那么需要测试多少个输入数据的组合呢？

第一个字符可能有10种方式（数字0～9）。第二到第六个字符每一个字符都可能有62种方式（数字0～9，小写字母a～z，大写字母A～Z）。这就意味着总共有$10\times(62^5)$或9 161 328 320个有效组合要测试。假设每一个组合用10秒测试，测试这些所有有效组合将用2905年！

因此，在2905年后，我们可以得出结论：所有的有效输入都是可以接受的。但是这不是故事的结束。当我们输入无效数据，程序又会发生什么呢？继续上面的例子，如果假定有10个标点符号，那么我们将用44 176年测试所有有效和无效的输入组合。

所有这些仅是验证一个字段，并穷尽测试，显然，穷尽测试一个现实的程序是永远也做不到的。

上述这些都意味着只能选择测试用例的一个子集来测试。为了增加测试的有效性，应该选择最有可能发现错误的子集。本书第4章和第3章将讨论一些等价类划分、边界值分析、代码路径分析等技术，以帮助确定测试用例子集，从而增加发现缺陷的可能性。

然而，不管选择哪些测试用例子集，都不能100%地保证没有遗留缺陷。另一方面，引用一句老话，除了死亡和税收外没有任何事情是确定的，但是我们还是活着，并通过明智地把握不确定性来做其他事情。

> 测试只能证明缺陷的存在，但绝不能证明缺陷不存在。

1.5 及时测试

产品中的缺陷可能来自任何阶段。当获得最初需求时就可能存在错误。如果错误的或不完整的需求构成设计和开发产品的基础，那么最终产品的功能就不可能正确。类似地，当产品设计即构成产品开发（即编码）的基础是有缺陷的，那么缺陷设计产生的代码将不符合需求。因此，一个必要的条件是软件开发（需求、设计、编码等）的任何阶段都能发现并修正本阶段的缺陷，不能让缺陷传递到下个阶段。

我们看缺陷传递的潜在成本。如果在需求获取阶段所获取的需求是有错误的，而这个错误直到产品交付给用户时都没有发现，那么公司会遭受以下额外的损失：

- 基于错误的需求做出错误的设计；
- 在编码阶段将错误的设计传递到错误的代码中；
- 测试保证产品符合（错误）的需求；
- 发布的产品带着错误的功能。

在图1-2中，需求中的缺陷用深灰色方柱表示。从图1-2可以看出，这些深灰色方柱一直传递到随后的三个阶段，即设计、编码和测试中。

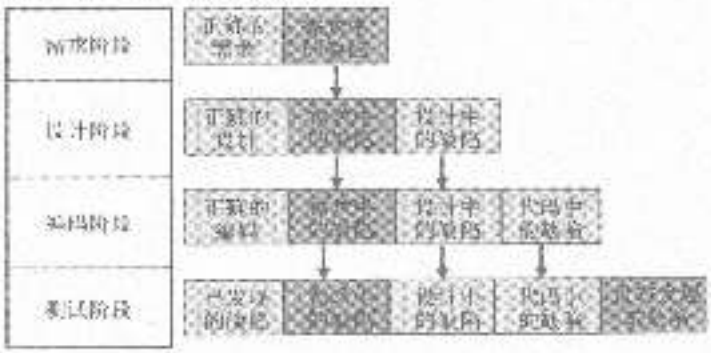


图1-2 早期阶段产生的缺陷如何使成本增加

当测试阶段完成后有瑕疵的产品送达用户手中，用户可能遭受停产带来的商业损失，反映到软件生产组织将是信誉的丧失。除了信誉损失，软件开发组织为了纠正问题，将不得不

重做列表中所有的步骤。

类似地，如果在设计阶段存在缺陷（尽管需求获取是正确的，彩图图1-2中用黄色表示），那么以后各阶段（编码、测试等）的成本会成倍增加。但是，这个成本比第一种情况在需求获取阶段出问题的成本要低，这是因为设计错误只能传递到编码和测试阶段。类似地，编码的缺陷只能传递到测试阶段。同样，可以预期编码阶段的缺陷影响较少（相对需求缺陷和设计缺陷，彩图图1-2中用绿色表示），导致的费用比前几个阶段要少。从上述讨论可以推断出，缺陷导致的费用随着检测出缺陷时间的拖延而增加。

传递产品的成本及产品中的缺陷数量随着缺陷数量传递到下一阶段而急剧增加。

因此，在缺陷注入（缺陷被引入）和检测缺陷（找到缺陷并改正缺陷）之间相隔的时间越短，不必要的费用就越少。因此，尽可能早地发现缺陷是非常重要的。行业数据已经再次肯定了这些发现。虽然关于缺陷检测延迟造成的成本并没有公认的结论，但是需求阶段引入的缺陷如果最终出现在发布的产品中，其导致的费用将是在需求阶段内发现并改正缺陷的上千倍，请参考图1-3。

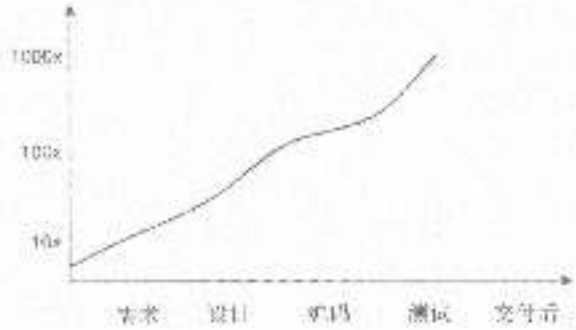


图1-3　缺陷对软件成本的综合作用

1.6　圣人和猫

一个圣人坐着思考，一只猫走来走去打断了他的沉思。于是，他叫门徒在他沉思时用篮子把猫扣住，此后，每次圣人思考都要把猫扣住，这变成了一种日常规程，一代代地传了很多年。一天，寺院里没有猫了，门徒们惊慌失措到处寻找，说："我们需要猫，有了猫，我们才能用篮子把它扣起来，这样圣人才能思考！"

测试首先要明白测什么，正确的结果是什么和为什么要执行这些测试。如果不明白为什么就执行测试，最后就会执行不合适的测试，不能检查产品所应该做的。事实上，有时为确保测试成功而修改产品，甚至产品被改得不能满足用户的需要！

理解为什么要测试某些特定功能会引出不同的测试类型，将在第二部分介绍。我们做白盒测试，检查代码中的各种路径，以确保代码的正确运行。知道给定测试用例会执行哪些代码，就能对其进行必要的修改，以确保覆盖合适的路径。知道产品的外部功能，就可以设计黑盒测试。集成测试用于确保不同组件能够组合在一起。国际化测试确保产品能够在世界各地多种语言环境中正常运行。回归测试用于确保变更与设计要求的一样，没有任何意外的副作用。

"为什么测试"的重要性同"测试什么"和"怎样测试"一样。

1.7 首先测试测试用例

一个耳科医生要检查她的病人，告诉她："我想测一下你能听多远，我从不同的距离问你的名字，你来回答，请转过身准备回答。"病人明白了需要做什么。

医生：（从30英尺外）：你叫什么名字？

……

医生：（从20英尺外）：你叫什么名字？

……

医生：（从10英尺外）：你叫什么名字？

病人：第三次了，我再说一遍，我的名字叫Sheela！

从上述例子看出，很明显医生而不是病人的听力有问题，不知道医生会不会认为病人不能在20英尺和30英尺听见问话而为病人开了治疗处方。

测试用例就像程序和文档一样，也是人类生产的产品，我们也不能确保测试用例是完美的，重要的是要在测试之前确保测试用例本身没有问题。确保测试用例本身已通过检测的一种方法是，对于一个特定的测试用例，把输入和预期的输出文档化，请专家确认这些文档的有效性，也可以通过一些外部方法反复检查测试用例。例如，给定一个已知的输入值，并且单独跟踪程序或进程的执行路径，就可以手工确定预期的输出。通过比较这"已知正确的结果"和产品产生的结果，可以增加测试用例和被测产品的可信水平。第3章将讨论评审和审查实践，以及测试策划，第15章将讨论测试测试用例的方法。

首先测试测试用例——有缺陷的测试用例比有缺陷的产品更危险！

1.8 杀虫剂悖论

每年，各种各样的害虫袭击田野和农作物，农业专家们要找到正确的对抗方法，用新改良的配方设计出杀虫剂。有趣的是，害虫适应了新的杀虫剂，产生了免疫力，使新杀虫剂失效。随后的几年里，老的杀虫剂只能用来杀死没有免疫力的害虫。同时还必须引入一些新的改良配方，用更厉害的新变异害虫作斗争。新旧杀虫剂的结合有时阻碍了旧杀虫剂发挥的作用。随着时间的流逝，旧的杀虫剂变得毫无用处，于是，害虫和杀虫剂之间不停地战斗，看最终谁占上风。有时杀虫剂赢，但是，有时害虫又可以成功地战胜最新的杀虫剂。这场斗争的结果是大自然和杀虫剂的不断发展进化。

缺陷就像害虫，测试就是设计正确的杀虫剂捕获并杀死害虫，测试用例就是杀虫剂。像害虫一样，缺陷发展免疫力抵抗测试用例。当我们写新的测试用例发现产品的缺陷时，其他"隐藏"在下面的缺陷就暴露出来。

有两种方式可以解释产品如何发展它的"免疫力"对付测试用例。一种解释是，最初的测试用例深入到代码一定程度，由于发现缺陷而停止进一步进行。一旦这些缺陷修复，测试继续进行，直到进入新的

测试就像杀虫剂——必须不停地改变其构成，以对付新的害虫（缺陷）。

以前没有处理的代码从而发现新的缺陷。这种“白盒”或代码方法可以解释为什么新的测试用例可以发现新的缺陷。

关于免疫力的第二个解释是，当用户（测试人员）开始使用（执行）产品时，最初的缺陷妨碍他们使用全部的外部功能。随着测试用例开始运行，发现缺陷，修复问题，用户得以继续探索新的没有使用过的功能，并发现新的缺陷。这种“黑盒”观点采用功能方法解释为什么“测试越多缺陷越多”的现象。

另一个解释这个问题的方法是，不是缺陷发展了免疫力，而是测试用例走得更深。更进一步诊断问题并且最终“杀死缺陷”。遗憾的是，由于软件很复杂，并且在多个组件之间交互，因此最终杀死的事情很少。测试后缺陷依旧存在，困扰着用户，造成数不清的破坏。

这种不断地修订要运行的测试用例，以识别新缺陷的需要，促使我们研究测试策划并且执行不同类型的测试，尤其是回归测试。回归测试承认新的修复（杀虫剂）可导致新的“副作用”同时引起一些旧缺陷出现。设计并执行回归测试的核心问题是设计正确的测试用例，同在先前测试中获得免疫力的缺陷作斗争。本书第8章将讨论回归测试。

1.9　护航舰队与破布

> 我们各个人都经历过交通堵塞，通常在交通堵塞时，我们可以看到护航舰队一样的效果。严重堵塞的道路向前延伸，车辆停起来像加入一支护航舰队，跟随着缓缓地向前航行（即驾驶），直到遇到下一个护航舰队。

测试只能发现存在于聚类中的一部分缺陷，修复一个缺陷可能给该聚类中引入另一个缺陷。

缺陷在程序中通常也会呈现出护航舰队现象，缺陷集中出现。Glenford Myers在关于软件测试的开创性著作[MYER-79]中提出，程序中某部分错误存在的可能性与此部分已经发现错误的数量成正比，参见图1-4。

听起来这可能不合理，但在逻辑上是可以证明的。通常一个缺陷的修复会引入一些不安定因素，因此有必要再修复。所有这些修复的副作用就是导致缺陷在产品的某些部分成群出现。

从测试策划的角度看，这就意味着如果我们发现在软件的某一部分存在缺陷，那么应该在更多而不是更少的时间测试这部分程序。这将增加测试的投入回报，因为测试的目的就是发现缺陷，这还意味着无论何时产品有任何变化，受到影响的部分就是容易出错的区域，这部分就需要测试。我们将在第8章讨论这方面的内容。

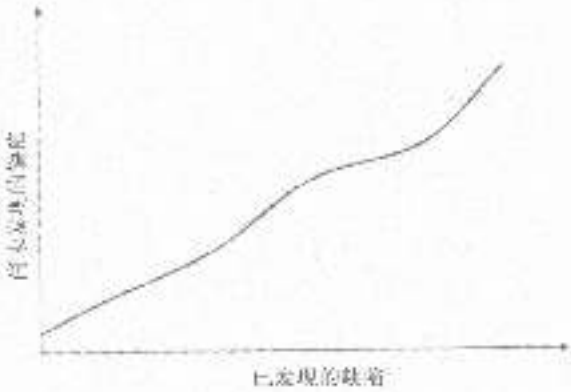


图1-4　尚未发现的缺陷随已发现缺陷数量的增加而增加

修复缺陷围绕特定几行代码进行，围绕同一段代码的修复会带来副作用。这会导致程序螺旋式变更，都是对特定部分的代码进行的。当我们审视这些修复了成群缺陷的代码，就像

看一块破床布！修补衬衣的一个破洞可能损坏另一个地方，一劳永逸的解决方法就是扔掉衬衣买件新的。这就相当于重新进行体系结构设计和重写代码。

1.10 桥上的警察

城市中有座桥，只要人们从桥上走过就会掉下去。为了解决这个问题，市长任命了一个强壮的警察站在桥下面接掉下的人。问题在一定程度上得以缓解，但仍不能彻底解决。

这个警察便休息，一个新警察接替了他的工作。在开始的几天里，新警察没有站在桥底接住掉落的人，他同一个工程师一起修好了桥上的洞。这是早先那个警察没有注意到的。从那时起，没有人再受伤，新警察也没有接住过一个人。（这使他现在的工作变得多余，他继续做其他对自己和人们都有益的事情……）

测试人员有可能能够很好地了解用户遇到的问题。就像上面故事所讲的第一个警察，他知道人们掉下来和人们为什么会掉下来。他们（宁愿冒着错过接住掉落人们的风险）不是简单地接住他们，而是调查导致掉落的根本原因并采取预防措施。测试人员本身可能难以采取预防措施。就像第二个警察需要工程师帮他堵上漏洞，测试人员也必须同开发工程师一起找到缺陷的根源并解决。测试人员不应认为解决问题与己无关。要像第二个警察一样，利用自己的缺陷检测经验，转换到缺陷预防，这既可以使自己的职业工作更丰富，也会使公司受益。

> 预防比治疗更有效——你可以把跟走就得更长远。

缺陷预防是测试人员工作的一部分。如果能够在缺陷预防和缺陷检测活动之间找到平衡，那么测试人员的职业工作就会变得丰富并会得到回报。这种职业发展道路包含在第13章将要介绍的一种三阶段模型中。下面我们讨论缺陷预防和缺陷检测之间的平衡问题。

1.11 钟摆的终结

任何软件公司最终的目的都是要确保用户能够得到没有多少缺陷的产品。达到这个目的有两个途径：一个是关注缺陷检测并改正缺陷，另一个是关注缺陷预防。这也称为质量控制关注和质量保证关注。

传统上测试被认为是一种质量控制活动，强调缺陷检测和更正。我们更愿意采用广义的测试观点，认为测试是面向测试预防的。例如，一方面第3章将讨论的白盒测试属于静态测试，包括桌面检查、代码走查、代码评审和审查。即使这些在传统上被认为是“质量保证”活动，整个测试活动的策划要以向用户交付高质量的产品为宗旨，这是难以做到的，除非全面地考虑通过质量保证能做什么、以及通过质量控制（即传统意义上的测试）能做什么。

质量保证通常部与过程模型例如CMM、CMMI、ISO 9001等关联，而质量控制通常与测试（测试是本书要讨论的主要内容）有关。这就导致了质量保证和质量控制之间的不自然分类。遗憾的是，公司都把这两种功能看作是互斥的，即“二选一”。我们甚至听到过“只要有好的过程管理，测试就是多余的”，或“过程是辅助的，测试可以发现一切”的说法。这就好像位于钟摆两端的两种思想流派：一个来源于缺陷预防（质量保证）关注，另一个来源于缺陷检测（质量控制）关注。经常会看到公司像钟摆一样从一个极端到另一个极端（如图1-5所示）。

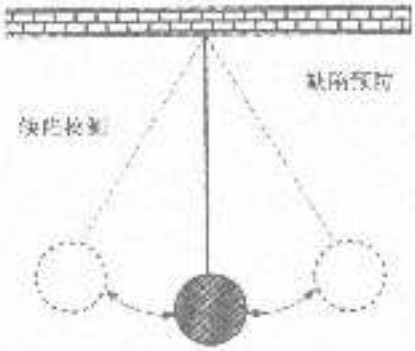


图1-5　质量控制和质量保证是提高质量的两种方法

我们认为，缺陷预防和缺陷检测不应看作是互斥的，也就是钟摆的两个极端，而应看作是两种相辅相成的活动，需要恰当地组合。图1-6给出了一种缺陷预防—缺陷检测网格，把这两种功能看作两个维度，两种活动正确的组合就对应于获取网格的的正确象限。

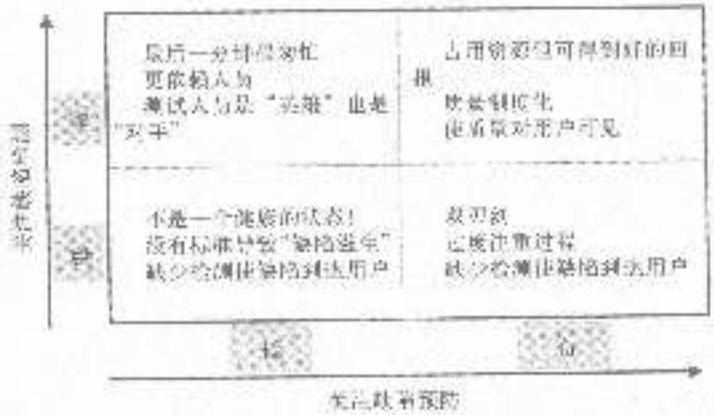


图1-6　缺陷检测关注和缺陷预防关注之间的关系

如果不太关注缺陷预防，就不会强调适合的标准、评审和过程，这种行为是缺陷理想的"温床"。保证产品质量的大部分工作都要靠测试及缺陷检测团队完成。如果也不太关注缺陷检测（图1-6的左下象限表示），公司就处于一种不好的状态。没有测试和缺陷检测活动就不能及时"杀死"缺陷，也就导致缺陷会达到用户。显而易见这不是一种健康的状态。

即使加强对缺陷检测的关注，但是仍然不注意缺陷预防（图1-6的左上象限），测试也会变成一种高度紧张、压力极大的工作。在产品发布前的最后一刻，还在检测大多数缺陷。测试人员于是成为及时找出所有缺陷从而扭转败局的超级英雄，同时他们也可能成为开发人员的敌人，因为他们总是找出开发人员所做工作的问题。这个象限比前一个好一些，但是很难维持，因为最后一刻玩的心跳会使人们筋疲力尽。

三个中国医生是兄弟。小弟是世界上非常有名的外科医生，他能发现人身上的肿瘤并切除。二哥能够在疾病的早期发现并开药治好它，他只在他所居住的城市非常有名。大哥不被外人所认识，但是两个弟弟总是听他的建议，因为他能够告诉他们在疾病没有出现时如何预防，大哥不是最出名的，但无疑却是最有效的。

预防疾病比治疗更有效。预防缺陷的人往往没有被关注。他们在公司中从未被称为英雄。灭火的人是大家看得见的人。不一定是一开始就确保火灾不发生的人。无论如何都不应该打击缺陷预防人们的积极性。

正如我们在上一节看到的，缺陷预防和缺陷检测不是互斥的，需要适当平衡才能提高产品质量。缺陷预防可改善产品生产过程的质量，而缺陷检测则可以捕获并改正过程中残留的缺陷。因此，缺陷预防就是关注过程，而缺陷检测就是关注产品。缺陷检测作为一种额外检查，可以放大缺陷预防的作用。

缺陷预防的提高在于能够建立评审机制、遵循先进的标准、完成工作遵循文档化的过程。从一开始就主动地关注把一切做好，会使测试工作（也就是缺陷检测）能够增加更多的价值，并能够在缺陷到达用户之前捕获残留的缺陷。质量就是始终如一地关注将缺陷预防与缺陷检测制度化。因此，一个公司必须分配足够的资源，以维持高水平的缺陷预防和缺陷检测活动（如图1-6所示的右上象限）。

缺陷预防和缺陷检测不能认为是互斥的，而应该是互为补充的。

但是，一个公司应该仔细避免过多依赖缺陷预防，而减少对缺陷检测的关注（如图1-6所示的右下象限）。过多关注缺陷预防，轻视缺陷检测，在研发和产品的质量管理层中会产生一种不舒服的感觉，因为内部发现的缺陷会很少。这种感觉会导致引入新过程，以改善缺陷检测的有效性。而太多的过程和缺陷预防最后可能被当作官僚形式，对于不同的情况没有灵活性和适应性。虽然过程带来了纪律约束，并降低了对个别人员的依赖，但是，如果没有实现其精神实质，过程也会成为双刃剑，会伤害到人们的积极性。如果公司既强调缺陷预防也强调缺陷检（图1-6中的右上象限），表面上看起来成本很高，但通过对内制度质量提升，对外，使用户能够看到这种变化，这种投入必定带来丰厚的回报。

公司应当为缺陷检测和缺陷预防适当地定位，也就是选择图1-6中的适合象限。对缺陷预防和缺陷检测强调的尺度要随着产品类型、发布日期的临近程度和可用资源的不同而调整。综合考虑不同因素，有意识地平衡缺陷预防和缺陷检测，将使公司生产出更高质量的产品。对于一个公司来讲，要避免过于强调一方面忽视另一方面，这一点很重要，这在下一节详细讨论。

1.12 黑衣人

从以上讨论可以看出，测试要求人员有多项才能。从事测试工作的人员应该以客户为焦点，从客户角度理解除令的要求。他们应具有很高的分析技能，以选择合适的测试用例子集，同时能正确应对杀虫剂悖论。他们应该提前考虑到缺陷预防，同时又能够识别并矫正出现的错误。最后，他们必须能够完成自动化工作（下一节将要阐述）。

为“测试”而自豪，就会处理好“其他一切”！

尽管所有这些都对所需的技术和人与人之间交流的技巧提出挑战，但测试仍然不是一种很受欢迎的工作。De Marco和Lister在他们的《人件》[DEMA-1987]一书中描述了一个有趣的实验。在测试团队中有目的地插入一些人，这些人“没有影响开发人员测试自己的程序会出现的认知问题”。这些人穿着特殊的衣服（黑衣服，以区别于公司中的传统工作服），并受到极大重视。所有的这些增加了他们的工作自豪感，使他们的成绩不可思议地飞跃增长，最初的团队成员离开并补充新成员很久以后，“黑衣团队”依然存在并且声名依旧。

选择测试作为职业的最大瓶颈在于缺乏自信。这种自信的缺乏和明显的对测试职业选择的不信任，使他们把测试看作是转到其他岗位的跳板（明显地，“开发”就是编写代码的委婉

提法）。结果是，测试人员没有能够在测试中寻找职业发展道路，而是增加了对测试这种职业的怀疑。

本书的第三部分将专门讨论职业期望和人们面临的类似问题。我们面对的一部分挑战是面对全球化，即合理利用全球资源保持竞争优势。第三部分的另一章将专门讨论由此引起的组织问题。

1.13　自动化综合症

一个农场主需要到一英里以外的井里提水用。于是，他雇了100个人从井里提水浇地。每人每天提一罐水，但这远远不够，庄稼枯死了。

在下一轮播种之前，农场主吸取了去年的教训，他想到自动化是提高生产率避免再次失败的关键方法。他听说摩托车可以更快地运水，于是，他买了50辆摩托车，解雇了50个工人，同时要求剩下的每人开摩托车运两罐水。看起来由于生产率的提高（由于摩托车的速度和便捷性），他需要的人更少了。遗憾的是，他是在庄稼生长期开始时，才选择使用摩托车的。因此，在开始的几周里，工人忙着学习使用摩托车，在学习摩托车的过程中，每天能运送的水罐数没有预想的那么多，加之工人的数量也减少了，运水能力实际上减少，庄稼又枯死了。

下一个播种季节又来了。现在除了一个人之外所有的工人都解雇了，农场主这次买了一辆卡车来运水。这次他也意识到需要训练工人学习驾驶，可是，从农场到井之间的路太窄，卡车不能帮他运来水，同样，这次庄稼也都枯死了。

经过这些经历后，农场主说："没有自动化，我的日子还会好过点！"

如果仔细提摸这个故事就会发现，庄稼枯死的多个原因都不是自动化引起的。农场主的失败不该直接归于自动化，而应该归于自动化所遵循的过程和不恰当的选择。第二年失败的原因是没有技能，第二年失败的原因是选择了不合适的工具。

自动化失败的例子比成功的多，自动化需要与产品开发一样的技能和关注。

第二年播种期到来时，农场主购买摩托车后立刻解雇工人，预期的金钱和时间都落空了，第二次他犯了同样的错误。自动化没有让他立刻得到回报。

上述故事的寓意同样适用于测试。自动化需要仔细地策划、评估和训练。自动化可能不会立刻产生回报，如果一个公司希望立刻通过自动化产生回报最终将会失败，并会错误地怪罪是自动化导致他们的失败，而不是客观地看待他们为自动化所作的策划、评估和训练等准备工作的水平。

大多数公司都因为自动化初期的失败而转向手工测试。遗憾的是，他们得出了错误的结论——自动化永远不能奏效。

测试的天性就包括重复性工作，也就是说，测试本身会自然地导致自动化。但是，自动化也是一把双刃剑，下面给出一些需要注意的关于自动化的要点：

- 在为了自动化而建议自动化之前，首先要知道为什么要采用自动化，以及想要自动化哪些内容。

- 在选择最适合需要的工具之前多做比较。
- 根据需要选择工具，而不是为配合工具的能力改变需要。
- 在指望测试人员提高生产率之前先进行培训。
- 不要期望自动化一夜之间就会产生回报。

1.14 小结

本章讨论了测试的一些基本原理，为后面内容的展开奠定了基础。本书由五部分组成。第一部分（包括本章）为本书其余部分确立了背景。下一章将讨论测试、验证和确认活动背景下的软件开发生存周期（SDLC）模型。

第二部分将讨论常见的测试类型。第3－10章涵盖了白盒测试、黑盒测试、集成测试、系统测试、确认测试、性能测试、回归测试、国际化测试和即兴测试的内容。

第三部分将讨论两个特殊且有些深奥的特殊测试问题，即第11章要讨论的面向对象测试、第12章要讨论的可使用性和易获得性测试。

第四部分将讨论一个经常被忽视的问题测试中的人员和组织问题。第13章将讨论常见的人员问题，包括一些误解、职业发展道路等问题。第14章将讨论当前流行的用于建立有效测试团队的不同组织结构，特别是在全球化背景下的组织结构。

最后的第五部分将讨论过程、管理和自动化等确保公司内部测试有效性的测试管理和自动化问题。第15章将阐述测试策划管理和执行，论述测试策划的内容，测试项目跟踪及相关问题。第16章将详述在测试领域正在出现且重要性日益凸显的重要问题，即实现测试自动化的好处、挑战和途径。第17章将详细说明衡量测试的有效性、产品的质量等，都需要采集什么数据，进行什么分析，以及如何使用这些信息实现可量化的持续改进。

虽然本书的各个部分都提供了必要的理论基础，但我们还是重点强调实践问题。通过阅读以上内容，读者可以概略了解本书其余部分的内容和背景关系。

问题与练习

1. 我们已经谈到了软件的普适性，它是迟早要对遗留在产品中的缺陷进行检测的一个原因。假设带有嵌入式软件的电视机可以在整个有线网络上自动下载、安装纠正错误的补丁，同时电视机制造商告诉你，这只需每周花5分钟的时间，而且“对用户免费。”你会同意吗？请给出一些不能接受的理由。
2. 你所在的公司已成功地开发出安装在几个客户那里的客户－服务器应用软件。你打算把它改成一个基于Web的应用，这样使任何人只需注册就能使用。请从质量和测试角度列出一些你认为这个经过修改的应用软件会面临的一些挑战。
3. 下面是一些来自产品开发公司的声音。找出这些观点中和本章相关原理不相符的错误。
   a. “本产品的代码是由CASE工具自动生成的，因此没有缺陷。”
   b. “我们已经通过了最新的过程模型认证，因此不需要测试。”
   c. “我们需要用针式打印机测试软件，我们从未没有发布过没有经针式打印机测试的软件。”
   d. “我已经运行了两个最新版运行过的所有测试用例，因此，不需要再运行任何其他测试用例了。”
   e. “我们的竞争对手用的就是这个自动工具，因此我们也应该用同样的工具。”

4. 假设需求获取时引入的每个缺陷到达客户的成本是10 000美元，设计缺陷和编码缺陷对应的成本分别为1000和100美元。还假设依据目前的统计数字，每个阶段要产生平均10个新的缺陷。此外，每个阶段的缺陷都会传递到下一阶段，那么现在的情况下，缺陷总成本是多少呢？如果采用了质量保证过程，每个阶段捕获50%的缺陷，使其不能传递到下一阶段，这可以节省多少成本呢？
5. 你要写一段两个两位整型数字相加的程序。你能穷尽测试这个程序吗？如果可以，需要多少个测试用例？假设每个测试用例的执行和分析需要一秒，运行所有的测试用例要花多长时间？
6. 我们认为程序中遗留的缺陷数量和检测出的缺陷数量成正比。为什么这个论点看似矛盾？同样，解释为什么在程序测试中会产生这种现象。

第2章 软件开发生存周期模型

2.1 软件项目的阶段

软件项目由一系列阶段构成。一般来说，大多数软件项目由以下阶段组成：

- 需求获取和分析
- 策划
- 设计
- 开发或编码
- 测试
- 部署和维护

2.1.1 需求获取和分析

在需求获取阶段，要收集构建软件的具体需求并形成文档。如果要构建的软件是预订软件，那么给出这些需求的是单一客户。如果要构建的产品是通用软件，那么要由软件产品公司内的产品市场开发团队通过汇集多种潜在客户的需求来描述需求。不管是哪种情况，最重要的都是要确保在每个环节获取的都是合适的需求。需求文档的形式是系统需求规格说明（SRS），这种文档是沟通客户和将构建该产品的设计人员之间的桥梁。

2.1.2 策划

策划阶段的目的是确定进度计划、项目范围和形成产品所需的资源。计划要描述如何满足需求，在什么时间满足。需要在这个阶段根据项目的范围、可以使用的资源以及产品的一组里程碑和发布时间考虑需求要满足什么和不满足什么这两个方面的问题。策划阶段既适用于开发活动，也适用于测试活动。策划阶段结束时，需要交付项目计划和测试计划文档。

2.1.3 设计

设计阶段的目标是确定如何满足系统需求规格说明文档所列出的需求。设计阶段要产生后续阶段即开发阶段将遵循的软件描述。这种软件描述应有两种用途：一是能够验证所有需求都被满足，二是能够为开发阶段进行系统的编码和实现提供足够的信息。设计通常分为两个层次，即高层设计和低层设计也叫做详细设计。设计阶段产生系统设计描述（SDD）文档，开发团队根据这种文档编写实现该设计的代码。

2.1.4 开发或编码

设计是实际编码的蓝图，这个开发或编码阶段要采用所选定的程序设计语言编写程序，产生符合设计确定的需求的软件。除了编程，设计阶段还要产生产品文档。

2.1.5 测试

程序以所选定的程序设计语言编写完成以后，还要进行测试。不仅如此，编码完成后，要按照计划进行测试。测试是使软件产品按照预先定义的方式运行，验证软件行为是否与预期的行为一致。通过测试软件产品，公司要在交付产品之前尽可能多地发现并清除缺陷。

2.1.6 部署和维护

产品测试完成后，要交付给将在其环境中进行部署的客户。用户在自己的环境中开始使用该产品后，可能会发现产品的实际行为和市场开发人员或通过产品文档确定的预期行为有差别。这些差别可能需要改正产品的缺陷。这时产品进入维护阶段，通过维护即修改产品满足客户预期，环境变化等引起的变更。维护工作包括更正性维护（例如解决用户报告的问题），适应性维护（例如使软件能够在操作系统或数据库的新版本上运行）和预防性维护（例如修改应用程序以避免操作系统代码中的潜在安全漏洞）。

2.2 质量、质量保证和质量控制

质量是对软件预期需求的一致和可预知的满足。

软件产品要满足一个或一批客户的特定需求。怎样刻画“满足需求”这个同呢？需求要转化成软件特性，所设计的每个特性都要满足一个或多个需求。对于每个特性，预期行为通过一组测试用例刻画。而每个测试用例又通过以下要素刻画：

1. 执行该测试用例所要求的环境；
2. 应该提供给该测试用例的输入；
3. 应该处理这些输入的方式；
4. 应该产生的内部状态或环境的变化；
5. 应该产生的输出。

给定软件对于给定测试用例在给定环境、给定内部状态和给定输入下的实际行为通过以下要素描述：

1. 这些输入的实际处理方式；
2. 实际产生的内部状态或环境的变化；
3. 实际产生的输出。

如果实际行为与预期行为对于上述要素都完全一致，那么就说该测试用例通过。否则，就说给定软件对于该测试用例存在缺陷。

如何增加产品一致和可预知地满足其预期需求的机会呢？有两种方法，即质量控制和质量保证。

质量控制试图构建一个产品，然后对其比对预期行为进行测试。如果预期行为与产品的实际行为不同，则对产品进行必要的修改，并重新构建该产品，这个过程反复迭代，直到产品的预期行为与测试场景下的实际行为一致。因此，质量控制是面向缺陷检测和缺陷更正的，针对的是产品而不是过程。

另一方面，质量保证试图通过关注生产产品的过程，而不是关注产品构建后的缺陷检测和更正来预防缺陷。例如，质量保证不是先产生再测试程序代码，比对合适的行为执行所构建的产品，而是在设计构建之前首先进行评审，改正设计错误。类似地，为保证生产出更好

的代码，质量保证过程还可以要求所有程序设计人员遵循编码标准。通过这个例子可以看出，质量保证通常更适用于采用某种过程的所有产品。此外，由于质量保证贯穿于产品的整个生存周期，因此是每个人的责任，是全员功能。而质量控制责任通常分配给质量控制团队。表2-1归纳了质量控制和质量保证之间的主要差别。

表2-1 质量控制和质量保证之间的差别

质量保证	质量控制
关注生产产品的过程	关注特定的产品
面向缺陷预防	面向缺陷检测和更正
通常贯穿整个软件生存周期	通常在产品构造后进行
通常是一种全员功能	通常是一种与产品线有关的功能
举例：评审和审核	举例：各种层次的软件测试

第3章将介绍质量保证方法，例如评审和审核的更多内容。不过本书主要关注的还是软件测试，软件测试本质上是一种质量控制活动。以下主要讨论软件测试。

2.3 测试、验证和确认

"测试"的狭义定义是编码之后、部署之前的阶段。过去测试专指测试程序代码。但是，编码是一种下游活动，而需求和设计则在产品生存周期中处于很靠前的位置。既然软件产品的目标是尽可能减少和避免缺陷，那么单靠程序代码测试是不够的。上一章已经谈到，缺陷会在任何阶段受诱，这些缺陷应该尽可能在接近引入点处检测，而不是等到程序测试时才检测。因此，如果每个阶段在结束时（更理想的情况是在阶段还没有完成时）都经过独立的"测试"，那么就可以更早地发现缺陷，从而降低总成本。

> 验证是评价系统或软件，以确定给定阶段的产品是否满足该阶段开始时确定的条件的过程。

及时测试可增加产品或服务满足客户需求的机会。当将在用户通过反映典型使用模式的恰当和现实的测试用例测试产品时，产品满足客户需求的机会就会增加很多。虽然测试不会保证零缺陷，但是有效的测试肯定会增加客户接受软件的机会。

测试的目的是发现系统中的缺陷（并叫别人改正这些缺陷）。测试由软件产品（或服务）公司内的一些人完成，其目标和宗旨是在产品到达客户手中之前发现产品中的缺陷（参见1.3节）。上一章已经介绍过，软件测试的目的不是证明产品没有缺陷，而是要发现软件产品中的缺陷。在本书后面介绍人员和组织问题的各章（第13、14章）将要讨论，成功的系统和组织结构应该营造并促进鼓励这种测试目的实现的一种环境。

测试并不取代其他质量保证方法（例如评审），它是检测软件产品缺陷的方法之一，还有可起到同样作用的其他方法。例如，本书后面将介绍的遵循定义完备的过程和标准可减少缺陷被引入软件的机会。本书还将讨论其他方法，这些方法实际上都试图防止缺陷被引入到产品中，为了更加有效，测试应该补充、完善、增强前一节提到的这些质量保证方法。

> 确认是在开发过程之中或结束时评价系统或软件，以确定其是否满足所描述需求的过程。

在每个阶段内部发现缺陷，避免缺陷到达测试阶段的理念又引出两个术语：验证与确认。

在需求获取阶段，需要如实地获取需求。SRS文档是需求阶段的产品。为了确保如实地获取需求，客户要验证SRS文档。设计阶段把SRS文档用作输入，并把需求映射到用来驱动编码

的设计上，SDD文档是设计阶段的产品。SDD要由需求团队进行验证，以确保设计如实地反映了SRS，而SRS在设计阶段的开始就确定了条件。

验证活动关注“我们是否恰当地构建产品”，确认活动关注“我们是否构建恰当的产品”。

为了恰当地构建产品，在生存周期的开始就要确定特定的活动、条件和规程。这些活动被认为是“主动的”，因为其目的是在缺陷形成之前进行预防。每个产品发布版本各个阶段执行的过程活动可称为验证。需求评审、设计评审和代码评审都是验证活动的例子。

为了构建恰当的产品，在各个阶段都要执行特定的活动，以确认产品是否按每条规格说明构建。这些活动被认为是“被动的”，因为其目的是发现影响了产品的缺陷，并在缺陷引入后尽快改正。确认活动的例子包括用来检验代码逻辑是否正确的单元测试，用来检验设计的集成测试以及用来检验是否满足需求的系统测试。

质量保证=验证，质量控制=确认=测试

总之，已经有两个相同或相似理念的不同方法论。出于本书的实际考虑，这里假设验证和质量保证是一回事，类似地，质量控制、确认和测试是一回事。

2.4　表示不同阶段的过程模型

过程模型是表示软件开发任意给定阶段的一种方法，有效地构成确认和验证概念，避免或尽可能缩短缺陷引入和缺陷检测（以及最后的更正）之间的延迟时间。在这种模型中，软件项目的每个阶段由以下要素刻画：

- 进入准则，描述该阶段能够开始的时间，还包括该阶段的输入。
- 任务，也就是需要在该阶段内执行的步骤，以及刻画该任务的度量。
- 验证，描述检查任务是否正确执行的方法。
- 退出准则，规定可以认为该阶段已经完成的条件，还包括该阶段的实际输出。

这个模型又叫做进入任务验证退出模型，或ETVX模型，在保证有效的验证和确认方面有以下几个优点：

1. 清晰的进入准则可确保给定阶段不会在不成熟的情况下开始。

2. 每个阶段（或每个阶段中的每个活动）的验证有助于防止缺陷出现，至少最大限度地缩短缺陷引入和缺陷更正之间的延迟时间。

3. 构成每个阶段的详细任务文档可降低指令解释的随意性，因此可最大限度地降低由不同个人重复执行这些任务所产生的差异。

4. 清晰的退出准则可提供阶段结束之后且进入下一个阶段之前的阶段确认的手段。

图2-1给出了将ETVX模型运用到设计阶段的一个例子。

图2-1　用于需求获取的ETVX模型

2.5 生存周期模型

ETVX模型刻画了项目的一个阶段，而生存周期模型则描述阶段如何组合在一起，形成一个完整项目或生存周期的形式。这种模型通过以下属性刻画：

所执行的活动 在任何给定软件项目中，除了最常见的活动和阶段，即需求获取、设计、开发、测试和维护，可能还有其他活动。这些活动有些可能是技术活动（例如移植），有些可能是非技术活动（例如招聘）。

每种活动的可交付产品 每种活动都会产生一组可交付产品，即该活动的最终产品。例如，需求获取阶段产生SRS文档，设计阶段产生SDD文档，等等。

可交付产品的确认方法 给定活动产生的输出标志着该活动要满足的目标，因此需要确定每种输出的恰当的确认准则。

活动序列 不同活动以一定的步骤序列协同作用，实现项目的总体目标。例如，需求获取的过程可以包括与客户的面谈、形成需求文档、和客户一起确认需求文档、冻结需求等步骤。这些步骤可能根据需要重复很多次才能达到最终的冻结需求。

每种活动的验证方法，包括活动之间的沟通机制 不同活动通过一定的沟通方法与其他活动交互，例如，当在一个活动中发现一个缺陷时，就会反向追溯较早活动的原因。从缺陷点回溯到该缺陷的原因需要恰当的验证方法。

以下介绍一些用于软件项目的常见生存周期模型，每个模型都要给出：

1. 模型的简要描述；
2. 模型与验证和确认活动之间的关系；
3. 生存周期模型适用的典型场景。

2.5.1 瀑布模型

在瀑布模型中，项目分成一组阶段（或活动）。每个阶段都是不同的，也就是说，在阶段之间存在清晰的分界线，每个阶段的功能都有非常清晰的划分。

项目从初始阶段开始，初始阶段完成，进入下一个阶段。这个阶段完成后，项目转向接下来的阶段，如此进行下去。因此，这些阶段在时间上是严格遵循顺序关系的。

图2-2给出了一个瀑布模型项目的例子。这个项目从需求获取的阶段开始，需求获取结束后生成系统需求规格说明文档。这个文档成为设计阶段的输入。在设计阶段，以系统设计描述（SDD）的形式生成详细设计。将SDD作为输入，项目进入开发或编码阶段。在这个阶段，程序员开发需要满足该设计的程序。程序员完成编码任务后，把产品提交给测试团队，由测试团队在发布之前对产品进行测试。

如果给定阶段没有问题，那么这种方法是有效的，会沿着一个方向进行下去（像瀑布一样）。但是，如果进入某个阶段后出现问题会怎么样呢？例如，进入设计阶段后发现采用目前的设计方法难以满足所提出的需求，可以采取哪些补救措施呢？如果可能，可以采用另一种设计，看是否能够满足需求。如果没有其他设计能够使用，则必须反馈到需求阶段修改需求。

瀑布模型有三个特征：

1. 项目分解为独立的不同阶段。
2. 每个阶段通过预先定义的输出与下一阶段发生联系。
3. 如果发现错误，则返回到上一阶段，一次跨一个阶段，直到在某个较早阶段改正该错误。

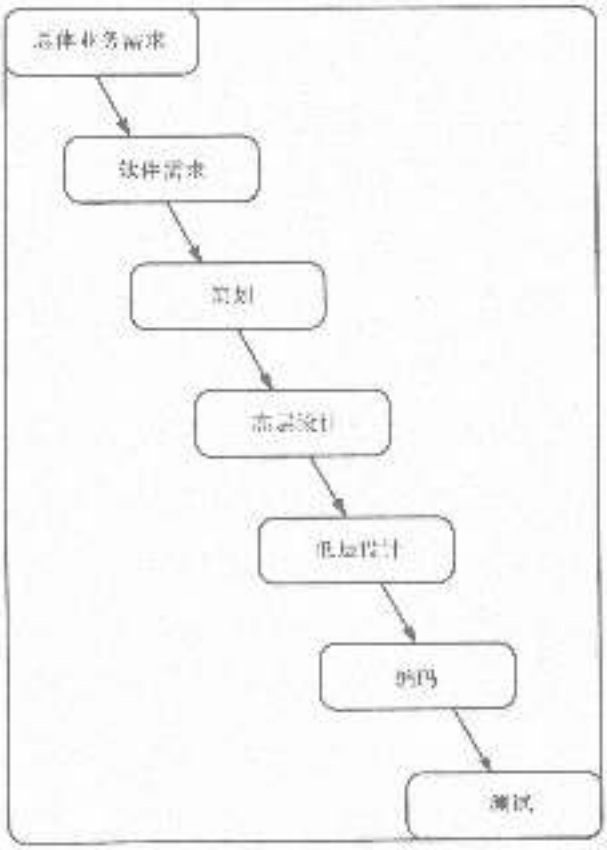


图2-2　瀑布模型

下面对前面的例子作进一步分析。假设针对给定的一组需求完成了一个设计，项目推进到程序设计/开发阶段。这时发现由于存在某些限制不能开发该设计对应的程序。这时该怎么办？一种方法是在开发阶段尝试其他策略，使该设计得到满足。另一种可能是设计存在问题，使得开发中产生冲突，这样需要重新研究该设计。当与前面的情况一样重新回到设计阶段时，可能发现该问题需要在需求阶段解决。这样，一个阶段的问题有可能追溯到前面任何一个阶段。

因为每个阶段都有输出，因此，输出可以比照一组准则进行确认。为了提高有效性，每种输出的完成准则都可以产生一种前提。在一个阶段开始前，可以检查上一个阶段的完成准则，并用作该阶段的验证机制。这样可以最大限度地缩短前面例子所提到的延迟。

瀑布模型的主要优点是简单。如果项目可以实际划分为独立部分，那么瀑布模型会非常有用。但是很少有软件项目可以这样划分。瀑布模型的主要缺点源自在阶段之间回溯产生的延迟，使得验证和确认活动的效率很低。一个阶段中的错误至少要到下一个阶段才能被检测出来。当某个阶段检测出错误时，只能与紧邻的上一阶段进行沟通。这种阶段之间沟通的顺序特征会导致问题的解决有很大的延迟。瀑布模型内部固有的应变能力低下，以及现实中很少有独立的阶段，这些都严重限制了瀑布模型的实际应用。

2.5.2　原型和快速应用开发模型

原型和快速应用开发（RAD）模型认识到并解决以下问题：

1. 尽早和频繁的用户反馈更有可能使软件产品满足客户的需求。

2. 变更是不可避免的，软件开发过程必须能够对自身进行改变以应对快速变化。

原型模型包含以下活动：

1. 软件开发组织与客户交互，理解客户的需求。

2. 软件开发组织生成原型，展示最终软件系统的外观。这个原型拥有可以说明输入屏幕和输出报告外观的模型，此外还有一些能够说明功能的机制，以演示工作流和处理逻辑。

3. 客户和软件开发组织不断评审该原型，以便在项目的一开始（即在需求获取阶段中）就不断获取客户的反馈。

4. 软件开发组织以客户反馈和所生成的原型为基础，生成系统需求规格说明文档。

5. 生成SRS文档后可以丢弃所生成的原型。

6. 将SRS文档用作进一步设计和开发的基础。

1. 原型模型早在需求获取阶段就通过不断的用户交互产生原型。

2. 使用原型导出系统需求规格说明，并可能在完成SRS后丢弃原型。

3. 在用户接受SRS后，选择合适的生存周期模型构建实际的产品。

因此，原型直接用作快速采集（恰当的）需求的手段。这种模型具有需求验证和确认的内在机制。在开发原型的过程中，客户的频繁反馈是确认机制。一旦SRS完成，可用作设计和后续步骤的验证机制。但是后续阶段的验证和确认活动要由获取SRS以后的生存周期模型实际描述。

在客户能够通过不断反馈进行参与方面，原型模型具有明显的优点。这种模型对于能够很容易地对客户反馈进行量化和集成，例如确定用户界面，预测性能等情况。

对于适用于很多客户的通用产品，没有单一的客户能够最终提供反馈。在这种情况下，产品提供商市场开发部的产品经理通常充当实际客户的角色。因此，原型模型在一定程度上不太适合通用产品。不仅如此，原型被用作获取需求的手段，以后不一定再使用。有时原型（或原型的一部分）会逐渐成为产品本身。这可能会产生没有预料到的后果，因为原型通常使用一些简化的方法、非结构化的方法和工具，以快速得到结果。这种简化方法在实际环境中是潜在的缺陷源。因此会给维护和测试带来很大的负担。

快速应用开发模型是原型模型的一种变种。与原型模型一样，RAD模型也依赖客户的反馈和交互获取最初的需求。但是，原型模型与RAD模型相比有两个不同点。

首先，在RAD模型中，构造的不是原型而是实际产品本身。也就是说，所构建的应用（对于原型模型来说就是原型）不会丢弃，快速应用开发模型也因此得名。

其次，为了保证获取需求的形式化并在设计和后续阶段正确反映需求，要从需求获取开始，在整个生存周期内使用计算机辅助软件工程（CASE）工具。这类CASE工具能够：

- 明确获取需求的方法论；
- 存储所获取需求以及所有源自下游实体的数据库，例如设计对象；
- 将存储在数据库中的需求自动转换为设计，并在所选定的程序设计环境中生成代码的机制。

CASE工具提供的方法论可以提供验证和确认的内置手段。例如，CASE工具能够自动检测和消除数据类型或依赖中的不一致。由于可以自动地通过需求生成设计（甚至可能还包括程序代码），确认会是非常完备的，一直延续到后续阶段，这一点与原型模型不同。

这种方法具有更宽的应用面，甚至包括通用产品。通过CASE工具自动生成设计和程序产品使这种模型更具吸引力。这种CASE工具的成本是组织在确定是否针对给定产品使用这种模

型时需要考虑的一个因素。此外，CASE工具和这种模型一般更适合应用项目，而不是系统类型的项目。

2.5.3 螺旋或迭代模型

在螺旋或迭代模型使用的过程中，要迭代地进行需求获取、设计、编码和测试，直到满足所有需求。采用这种模型，需求获取、设计、编码和测试活动之间会有大量重叠。很难确定产品所处的阶段，因为每个需求可能处于不同阶段。能够得出的唯一结论是单个需求所处的阶段。如果给定需求的任何阶段出现缺陷，会使该需求返回到前面的阶段，随着需求动态地加入，这种模型支持随着产品进化进行的渐进式开发。这使得开发人员能够在任何时间演示当时产品所具有的功能，还能够向客户提供“增量”版本以得到认可，产品的进化可以从项目的开始看到，因为模型每隔一定时间就会提供“增量”版本。即使采用这种模型很难策划产品发布日期，但是它可以使开发人员跟踪到项目的进展，定期得到客户的认可，因此可以降低在项目后期发现重大缺陷的风险。表2-2给出了产品的一些需求阶段的一个例子。

表2-2　一些产品需求和阶段

需求	当前状态/阶段
需求1	编码
需求2	设计
需求3	需求获取
需求4	测试
需求5	已发布

图2-3（参见彩图）描述了螺旋模型和对于表2-2所示例子模型中所包含的阶段。从图中可以看出，每个需求都随整个项目的进化，“螺旋式”地通过不同阶段。

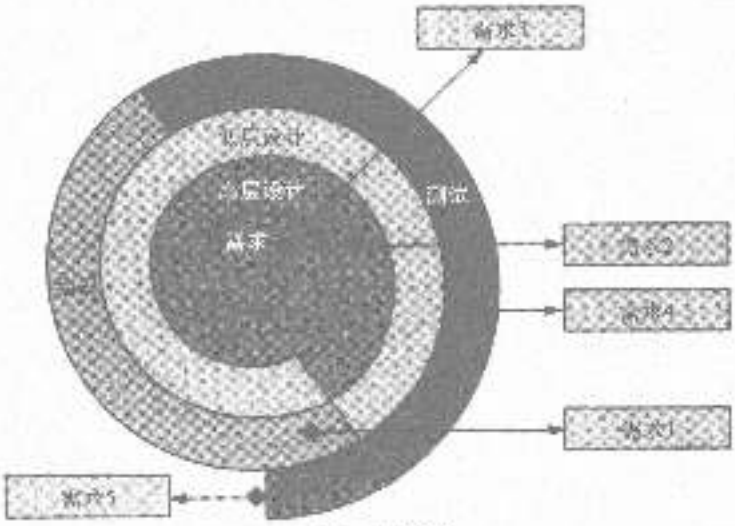


图2-3　螺旋模型

2.5.4 V字模型

瀑布模型将测试看作是一种开发后（即编码后）的活动。螺旋模型将测试看作是前进的一步，并试图将产品分解成增量版本，每个增量版本都可以单独测试。V字模型的出发点与瀑布模型类似，都把产品开发看作由若干阶段或层次组成。但是V字模型带来的新观念是不同类型的测试适用于不同层次。因此，从测试的角度看，每个层次需要完成的测试类型具有很大

不同。

例如，考虑前面介绍瀑布模型时给出的图2-2所表示的典型产品开发活动。系统从获取客户角度的总体业务需求开始，这些需求包括硬件、软件和运行需求。由于我们关注的是软件，因此从总体需求转移到软件需求就构成下一个步骤。为了实现软件需求，预期的软件系统被看作是一起协同的一组子系统。这种高层设计（将系统分解为具有一致接口的子系统）再转换到更详细的设计。详细设计解决诸如数据结构、算法选择、表单布局、处理逻辑、例外条件等问题，最后给出若干组件，每个组件都通过以合适的程序设计语言编写的代码实现。

有了这些层次后，针对每个层次需要采用什么类型的测试呢？首先对于总体业务需求，实际上所开发的任何软件都应该在这个总体框架中运行，并且被用户在其自己的环境中接受。这种最终证明的测试可做确认测试。但是，在产品部署到客户环境之前，产品提供商还应该把产品作为一个整体单元进行测试，以保证所开发的产品已经满足所有的软件需求。这种整个软件系统的测试可称为系统测试。由于高层设计将系统看作是由相互协同和集成的（软件）子系统组成的，因此，应该在能够测试整个系统之前，首先集成和测试每个单个子系统。这种对高层设计的测试就是集成测试。组件是低层设计的产品，需要在集成前进行独立测试。这种对应低层设计阶段的测试叫做组件测试。最后，由于编码会产生若干程序单元，在试图将这些程序单元组装为组件之前需要对其进行独立测试。这种对程序单元的测试就是单元测试。

图2-4给出了用于每个步骤的不同测试类型。为了简单起见，这里没有单独给出策划阶段，因为所有测试阶段都有策划阶段。但是直到产品实际构建出来以后才能够执行这些测试。换句话说，称为“测试”的步骤现在被分解为叫做验收测试、系统测试等不同的子步骤，如图2-4所示。所有测试只有在生存周期结束时才能执行相关的活动。

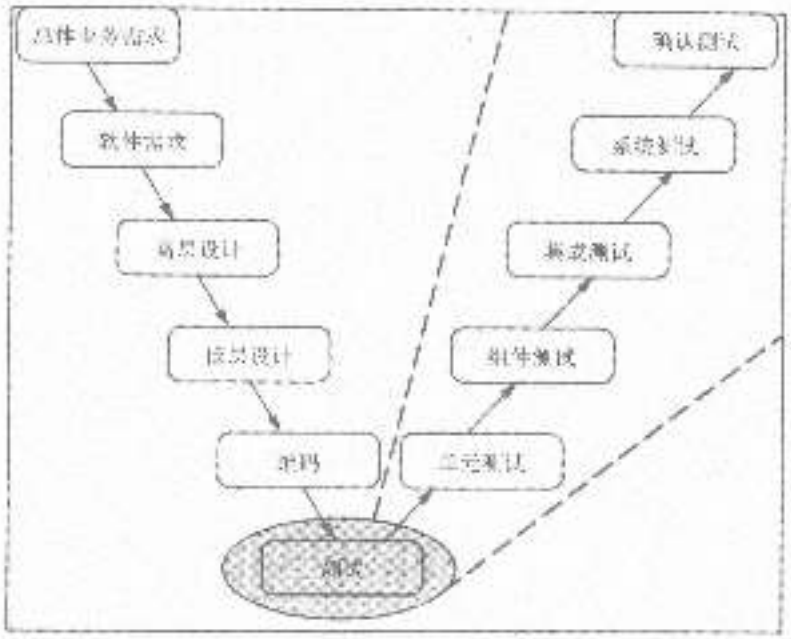


图2-4 不同开发阶段的测试阶段

即使测试执行要到产品构建完成后才能进行，测试的设计也可以在早期完成。事实上，

1. V字模型将测试分解为两个部分：设计和执行。

2. 测试设计在早期完成，测试执行在后期进行。

3. 生存周期的每个阶段都有不同类型的测试。

如果观察一下设计每类测试所要求的技能就会发现，设计所有这些测试的最佳人选是实际执行创建对应工作产品功能的人。例如，确定验收测试的最佳人选应该是形成总体业务需求的人（可能的话当然包括客户）。类似地，设计集成测试的最佳人选应该是知道系统如何分解为子系统以及子系统之间接口的人，也就是完成高层设计的人。同样，完成开发的人了解程序代码的内部结构，因此是设计单元测试的最佳人选。

不仅设计这些不同类型测试所要求的技能是不同的，而且没有理由得出要到最后才设计测试的结论。在V字型左侧的活动进行过程中，就可以着手设计对应类型的测试。通过进行测试的早期设计只在最后执行测试，可以得出以下三个重要结论：

- 首先，活动更并行化，可减少生存周期结束进行测试所需的时间。
- 第二，通过事先为每种活动设计测试，实际上是在进行更好的事先确认，同样可以降低最后一刻暴露严重问题的风险。
- 第三，测试由具有合适技能的人员进行设计。

这就是V字模型的基础，在验证和确认上具有很大的优势，如图2-5所示，对于每类测试，这里都把测试设计向前提，而测试执行活动仍然放在产品构建完成后进行。

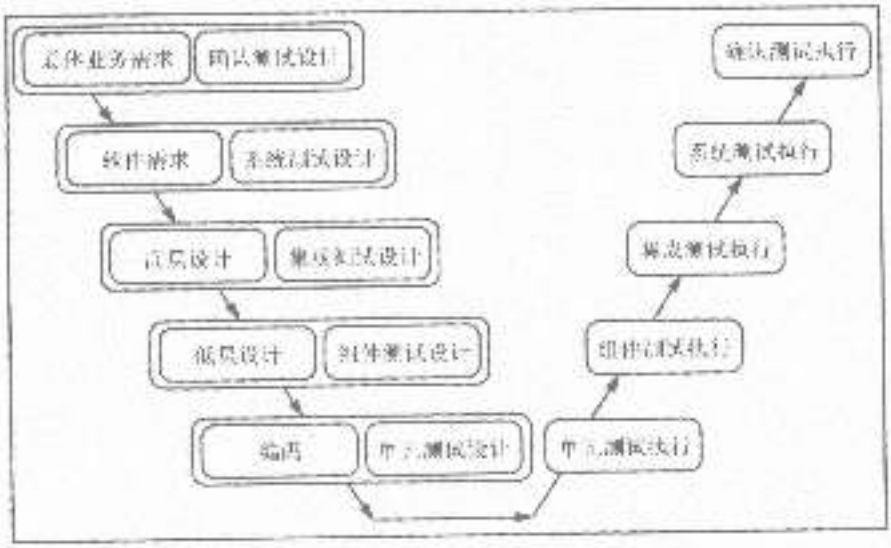


图2-5　V字模型

2.5.5　改进型V字模型

V字模型将各种类型的测试划分为设计和执行两个部分，并将测试设计部分附到对应的软件生存周期早期阶段。

假设即使测试执行活动分解为不同类型测试的执行，执行还是要等到整个产品构造完成后才能进行。对于给定产品，不同的单元和组件可能处于不同的进化阶段。例如，某个单元可能仍然在开发，因此处于单元测试阶段；另一个单元已经可以进行组件测试，而该组件本

身还不能进行集成测试。有些组件（即已经测试过的组件）可能已经能够进行集成测试（包括已经能够进行集成的其他模块，只要这些模块能够被集成）。V字模型并没有明确说明这种在产品开发中经常遇到的自然并行性。

在改进型V字模型中明确说明了这种并行性。如只每个单元、组件或模块明确地给出进入后续阶段的退出准则，那么满足给定测试阶段的单元、组件或模块只要有可能就会转向下一个测试阶段，不一定等到所有单元、组件或模块都从测试的一个阶段转到另一个阶段，如图2-6所示。

1. 改进型V字模型承认产品的不同部件处于进化的不同阶段。

2. 每个部件在满足合适的进入准则时进入合适的测试阶段（例如单元测试、组件测试等）。

就像V字模型引入各种测试类型一样，改进型V字模型引入各种测试阶段。测试阶段与测试类型有一对一的映射关系，即有单元测试阶段、组件测试阶段等。一旦单元完成单元测试阶段，就成为组件的一部分并进入组件测试阶段，然后再进入集成测试阶段，等。改进型V字模型不像V字模型那样把产品看作要走过不同的测试类型，而是把产品的每个部件看作要走过不同的测试阶段。这实际上是一个问题的两个方面，因此提供了不同的视点。改进型V字模型带来的主要优点是承认产品不同部件的并行性，并把每个部件分配到可能最合适的测试阶段。在图2-6中，表的列表示V字的一边，行（测试阶段）表示V字的另一边。

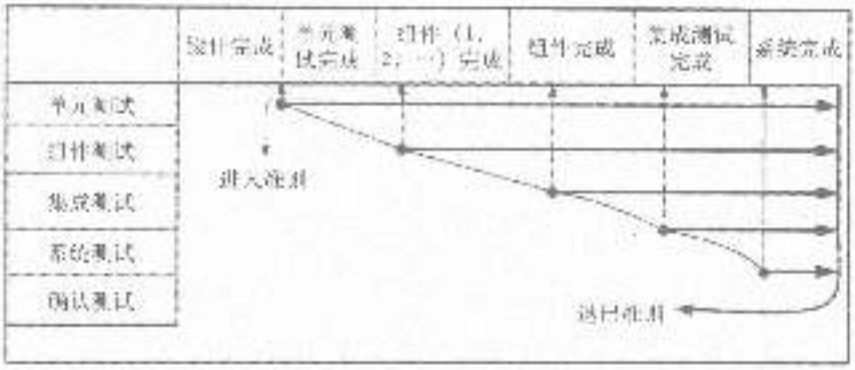


图2-6 改进型V字模型

从图2-6中可以看出，不同的测试阶段是并行完成的。在开始一个测试阶段时，注意该产品是否已经测试就绪是很重要的。这需要通过一组进入准则确定。最早开始下一个测试阶段的可能条件是由进入准则描述的。启动下一个测试阶段较早的阶段不一定结束。每个测试阶段还有一组退出准则，这些准则决定测试阶段的结束。每个阶段的进入和退出准则可确保交付给测试的产品已经达到合适的质量水平，以及交付测试后的产品完成了合适的测试量。尽管图2-6给出的所有测试阶段都同时结束，但实际上是可以不同时结束的。最长的阶段决定发布日期。

在图2-6中还有两种以前没有讨论过的活动，即“组件（1，2，…）完成”和“组件完成”。生存周期中没有额外的活动，这些文字只是用来表示集成测试可在两个组件完成后开始，以及当所有的组件都经过集成和测试，下一个测试阶段即系统测试才能开始。

2.5.6　各种生存周期模型的比较

从前面的讨论可以看出，每种模型都有其优点和缺点，适用于不同的场景。每种模型为验证和确认带来不同的问题、挑战和机会。表2-3归纳了每种模型关于适用性和相关验证和确认问题的突出点。

表2-3　模型适用场合及与验证和确认的相关性

模　型	适用场合	相关验证和确认（V&V）问题
瀑布	存在非常明确的阶段界限 每个阶段的可交付产品能够在进入下个阶段前冻结	测试是V&V推后至少一个阶段 过分强调属于最下游的活动 纠错错误的成本（即更正错误的时间消耗）很高
原型	有能够提供反馈的客户（或产品经理）	提供高水平的内外反馈机制 采用原型（甚至是丢弃原型）会使验证和确认很困难，并可能产生出乎预料的后果
RAD	有能够提供反馈的客户（或产品经理） 有关CASE工具和设计模板工具	除了需求和设计方面的内在反馈机制 CASE工具可以生成有准确文档，可用来实现V&V
螺旋	产品增量进化 能够进行中间阶段的检查和更正	将V&V推广到所有开发版本 将V&V推广到所有阶段（即需求阶段以外的其他阶段） 可以在打印开发形成产品的文档，能够实现迅速发布
V字模型	测试的设计能够与实际实现并行分离	测试的早期设计通过缩短开发和测试之间的延迟可缩短延迟 测试的早期设计可以更好地、更及时地对各个阶段进行确认
改进型V字模型	产品可以分解为不同的部件，每个部件可以独立测试	通过使每个部件更迅速进化进一步提高V字模型的并行性 通过引入测试活动之间的并行性进一步缩短总体延迟

问题与练习

1. 哪种软件开发生存周期模型最适合以下哪种情况？
   a. 产品是为具体客户订制的，客户随时可以提供反馈。
   b. 同上，只是开发人员拥有可以自动生成程序代码的CASE工具。
   c. 通用产品，但是产品市场开发团队非常强，他们非常了解并能够很好描述客户的总体需求。
   d. 产品由若干能够独立，增量地提供的功能组成。
2. 根据本书讨论的判断准则，以下哪种产品可以认为是“高质量的”？对自己的答案进行说明。

a. 产品的连续三个版本分别被发现0、79和21个缺陷。

b. 产品的连续三个版本分别被发现85、90和79个缺陷。

3. 从测试角度给出以下每种模型的三种或更多挑战：

a. 螺旋模型。

b. V字模型。

c. 改进型V字模型。

4. 从V字模型转移到改进型V字模型会遇到哪些挑战？

5. 图2.1给出了软件项目设计阶段的ETVX图。为编码阶段也绘出一幅类似的ETVX图。

6. 本书讨论的螺旋模型适用于能够增量进化的产品。试讨论V字模型怎样用于这种可增量进化的产品。

第二部分 测试类型

这一部分要讨论各种测试类型。各章安排从更接近代码到更接近用户的顺序排列。第3章将讨论利用程序代码的内部知识测试程序的白盒测试，第4章将讨论只了解需求规格说明描述的外部行为就可以测试产品外部行为的黑盒测试。软件是按模块进行开发的，模块要集成到一起，第5章将介绍集成测试。第6章将介绍的系统和确认测试要在类似客户部署的环境中，从用户的视角完备地测试产品。第7章将讨论的性能测试要测试系统是否有能力经受住典型和超常的工作负载冲击。有些软件需要不断变更，由于变更不能废弃已经有效的功能，因此第8章将讨论的回归测试就显得非常重要。由于软件必须以世界多种语言部署，因此，第9章将讨论国际化和本地化测试问题。最后在第10章中讨论的即兴测试是一种以一般不可预测的方式测试产品的方法，可以供最终用户使用。

第3章 白盒测试

3.1 白盒测试的定义

每个软件产品都是通过程序代码实现的，白盒测试是一种通过考察并测试实现外部功能的程序代码，测试代码的外部功能的一种方法，也叫做透明盒测试、玻璃盒测试或开放盒测试。

白盒测试考虑程序代码、代码结构和内部设计流。第4章要讨论的黑盒测试则不考虑程序代码，只从外部视角观察产品。

有些缺陷的起因是没有正确地将需求和设计转换为程序代码。另外一些缺陷是由程序设计错误和程序设计语言特性引起的。本章将要讨论的不同的白盒测试方法有助于缩短缺陷引入程序代码和将其检测出来之间的延迟。此外，由于程序代码代表产品实际要完成的事情（而不是期望产品要完成的事情），通过检查程序代码进行测试能够使测试人员更接近产品实际要完成的事情。

如图3-1所示，白盒测试分为"静态测试"和"结构测试"。3.2节将详细讨论静态测试，3.3节讨论结构测试。

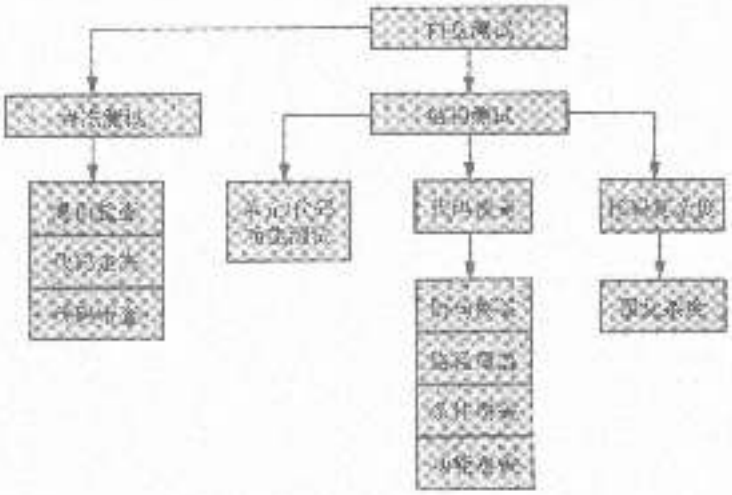


图3-1 白盒测试分类

3.2 静态测试

静态测试只要求提供产品的源代码，不要求提供二进制代码或可执行程序。静态测试不在计算机上执行程序，而是由人阅读代码，以确定：

- 代码是否能够满足功能需求；
- 代码是否与项目生存周期初期开发的设计一致；
- 是否遗漏功能代码；

• 代码是否恰当地处理错误。

静态测试可以人工完成，也可以借助专门的工具。

3.2.1 人工静态测试

人工静态测试依靠人通过阅读程序代码发现错误，而不是依靠计算机执行代码来发现错误。这种过程有以下优点：

1. 有时人工可以发现计算机发现不了的错误。例如，如果两个变量具有类似的名称，而程序员在表达式中“误”用了变量，计算机不能发现这样的错误，只是执行该语句并得出错误的结果，而人则可以识别出这样的错误。

2. 通过多人阅读并评价程序，可以得到多个视点，因此可以发现的错误比计算机发现得多。

3. 人工评价代码可以对比规格说明或设计，因此可以确保程序完成期望其完成的工作。而计算机运行测试并不总能做到这一点。

4. 人工评价一次可以检测很多问题，甚至可以尝试确定引出这些问题的根源。往往发现同一个根源可以解决多个问题。典型的情况是，在被动测试中，一个测试用例一次发现一个问题（最多发现若干问题）。这种测试一般只揭示现象，不能给出根源。因此，通过人工评价，可以显著缩短解决所有问题所需的总时间。

5. 通过在执行之前人工测试代码，可以节约计算机资源。当然，这是以增加人工成本为代价的。

6. 像静态测试这样的主动测试可以最大程度地缩短发现问题的延迟时间。第1章已经介绍过，发现、修改缺陷越早，解决缺陷的成本越低。

7. 从心理学角度看，在生存周期的后期（例如代码已经编译并正在集成系统）发现缺陷会对程序员产生很大的压力，他们不得不用更短的时间改正缺陷。在这种压力作用下，更有可能引入其他缺陷。

有多种人工静态测试方法，（按形式化程度排列）包括：

1. 代码的桌面检查
2. 代码走查
3. 代码评审
4. 代码审查

由于人工静态测试要在代码编译执行前进行，这些方法中的一些可以看作是面向过程，面向缺陷预防或面向质量保证的活动，而不是纯粹的测试活动。尤其是随着这些方法越来越形式化（例如Fagan审查），这些方法一般归为“过程”范畴。在形式化过程模型，例如ISO 9001，CMMI中有了人工静态测试的位置，很少归为“测试”范畴。但是正如本书前面已经提到的，我们从宏观的角度看待“测试”，任何提高产品质量的活动都纳入“测试”范畴。将人工静态测试方法纳入本章内容是因为这些方法要观察程序代码。

下面依次详细介绍这些方法。

桌面检查

桌面检查是一种检查代码各部分正确性的方法，通常由代码的编写者手工完成。这种验证通过将代码与设计或规格说明进行对比，保证代码有效地完成所期望的工作。这种桌面审查工作大多数程序员在编译执行代码之前都会做。一旦发现错误，代码编写者会立即对错误

实施修改。这种检查和改正错误的方法的特点是：

1. 没有可以保证完备性的结构化或形式化方法。

2. 不维护记录或检查单。

实际上，这种方法完全依靠代码编写者的投入、勤奋和技能，没有过程或结构保证或验证桌面检查的有效性。这种方法对于改正“明显”编程错误是有效的，但是在涉及由于错误理解需求或遗漏需求而引起的错误方面不是很有效。这是因为开发人员（更确切地说是执行桌面检查的程序员）可能没有充分理解需求所需的领域知识。

桌面检查的主要优点是程序员非常了解代码和所使用的程序设计语言，很容易阅读和理解自己编写的代码。另外，桌面检查是由个人实施的，没有多少进度安排或辅助支持开销。不仅如此，桌面检查还能以最短的时间延迟发现并改正缺陷。

桌面检查的缺点包括：

1. 开发人员不是检查自己编写的代码的最佳人选。他们可能受惯性思路的限制，难以看出某些类型的问题。

2. 开发人员一般更喜欢编写新代码，而不是做任何形式的测试。（本书在谈论挑战，第13章在讨论人员问题时将更深入地分析这种综合症。）

3. 桌面检查基本上是依靠人的，是非形式化的，因此可能很难在所有开发人员中产生一致的效果。

由于桌面检查有这些缺点，以下介绍两种其他主动测试方法。代码走查和正式审查（代码审查）的基本原则包括审查过程中的多人参与。

代码走查

代码走查和下一节将要讨论的正式评审都是面向小组的方法。代码走查比代码审查的形式化程度低。走查和审查之间的形式化界限并不非常清楚，各个组织的做法也有很大差异。代码走查相比桌面检查的优点是引入了多视角。在代码走查中，由一组人员检查程序代码并向代码编写者提出问题。程序编写者解释代码的逻辑，回答大家提出的问题。如果代码编写者对有些问题不能给出回答，则需要带回去找出答案。代码走查的完备性受走查组所提问题范围的限制。

正式审查

正式评审代码审查又叫做Fagan审查（以代码审查最早提出者的名字命名），一般具有很高的形式化程度。代码审查的关注点是检测所有缺陷、违规和其他副作用。通过以下措施代码审查可以增加所检测的缺陷数：

1. 要求在审查小组审前作充分准备；

2. 列出多个不同的视角；

3. 为多个参与者分配具体的角色；

4. 以结构化的方式顺序走查代码。

正式审查只有在代码编写者已经确认代码经过基本的桌面检查和代码走查，可以进行审查时才能进行。如果代码处于这种合理的审查就绪状态就可以安排审查会议了。审查有四种角色。首先是代码的编写者，其次是协调员，负责审查能按形式化地按照过程进行；第三个角色是审查员，对代码实际给出评审意见，一般有多位；最后是记录员，负责在审查会上详细记录，并在会后将会议记录提供给与会人员。

审查组由代码编写者或协调员选定。被选定的人员要具备尽可能多地发现缺陷的能力，

在预备会上，评审员应得到待审查代码的硬拷贝或软拷贝，以及其他支持性文档，例如设计文档、需求文档和所有可使用的标准。代码编写者还要说明待审查程序预期完成的工作，以及其希望审查组需要特别注意的问题。协调员通知审查组召开审查会的时间和地点。审查员要有足够的时间通读文档和程序，以确保文档和程序符合需求、设计和标准。

审查组在规定的时间召开审查会（又叫做缺陷记录会）。协调员带领审查组从头到尾顺序地审查程序代码，询问每个审查员在那部分代码中是否发现缺陷。如果有审查员提出发现了缺陷，则审查组集中讨论该缺陷，确定所提出的问题确实是个缺陷，并按两个要素将其分类：即轻微缺陷/严重缺陷和系统性缺陷/偶发性缺陷。顾名思义，偶发性缺陷是指由于编写者自身的错误或疏忽引入的缺陷。偶发性缺陷不太可能再在本工作产品或其他工作产品中反复出现。偶发性缺陷的一个例子是在语句中使用了一个错误的变量。而系统性缺陷需要在不同层次上采取措施。例如，使用了某种与具体机器有关的方法引起的错误。解决这样的问题可能需要修改编码标准。类似地，轻微缺陷不会严重影响程序，而严重缺陷需要特别关注。

记录员要将审查会上发现的缺陷正式形成文档。代码编写者负责修改这些缺陷。如果所发现的缺陷很严重，审查组可以召开评审会，对缺陷修改情况进行审查，以保证问题得到解决。不管是否召开评审会，通过审查发现的缺陷都需要进行跟踪，直到审查组的某个成员已经验证所发现的问题已经得到妥善解决。

组合各种方法

以上讨论的方法并不是互斥的。为了更好地达到尽早发现缺陷的目标，需要合理地组合使用这些方法。

实践证明正式审查在尽早发现缺陷方面是非常有效的。进行正式审查需要关注的挑战包括：

1. 很费时间。正式审查过程除举行正式审查会外，还要进行充分的准备，这是很费时间的。

2. 由于涉及多人参加，支持条件和会议安排可能成为问题。

3. 要记住多个参数及其组合才能确保程序逻辑、副作用和错误处理的正确，因此并不总能通读每行代码。可能不必对全部代码进行正式审查。

为了应对这些挑战，需要在策划阶段确定正式审查要处理的部分代码。可以根据关键级别或“高”、“中”、“低”复杂度将各部分代码分类，高或中复杂度的关键代码应该作为正式审查的对象，而复杂度低的代码可以作为代码走查甚至桌面检查的对象。

桌面检查、走查、评审和审查不仅用于代码，也可以用于项目生存周期内的所有其他可交付产品，例如文档、二进制代码和介质。

3.2.2 静态分析工具

前面介绍过的评审和审查机制需要大量人工工作。现在市场上已经有一些静态分析工具可以减少人工工作，并对代码进行分析，以找出以下列出的错误：

1. 是否有不可达的代码（使用GOTO语句有时会出现这种情况。可能还有其他原因）
2. 定义的变量没有使用
3. 变量的定义和分配算类型不匹配
4. 变量非法或易出错的赋值
5. 使用了不可移植或依赖体系结构的程序设计结构
6. 分配了内存，但是没有相应的语句释放
7. 计算圈复杂度（在3.3节介绍）

这些分析工具还可以看作是编译器的一种扩展，因为静态分析工具在定位错误方面使用了与编译器系统相同的概念和实现。好的编译器也是静态分析工具。例如，大多数C编译器都提供不同“层次”的代码检查，以捕获以上给出的各种类型的程序设计错误。

有些静态分析工具也可以检查像POSIX这样的标准所描述的编码标准的符合性。这些工具还可以检查与编码指南的一致性（例如，命名规则、允许的数据类型、可以使用的程序设计结构等）。

不论采用哪种人工检查方法，包括桌面检查、走查、正式审查，列出代码评审检查单都是很有用的。以下给出的是包含常见问题的检查单。每个组织都应该制定自己的代码评审检查单，检查单应该不断更新，以反映最新发现的问题。

对于有多个产品的公司，检查单可能要分成两个层次。公司级检查单包括诸如全公司的编码标准、文档标准等内容。其次，产品或项目级检查单描述与具体产品或项目有关的内容。

代码评审检查单

关于数据项定义

- 变量的名称有意义吗？
- 如果程序设计语言允许在名称中混用大小写字母，那么变量名称的大小写字母使用是否会产生误解？
- 变量初始化了吗？
- 有发音很近似的名称吗（特别是单复数的使用）？[这类情况可能产生意外错误。]
- 所使用的公共结构、常量和标志都是在头文件中而不是在各个独立的文件中定义的吗？

关于数据的使用

- 是否将数据类型正确的值赋给变量？
- 所有标准文件或数据库对数据的方法都是通过公开支持的接口吗？
- 如果使用了指针，指针是否被恰当地初始化？
- 数组下标边界和指针是否被恰当地检查？
- 相似操作符的使用是否检查了（例如C语言中的=和==，以及&&和&&）？

关于控制流

- 所有条件路径都是可达的吗？
- 复合条件中的所有单个条件是否都独立地计算过？
- 如果有嵌套IF语句，是否恰当地分开了THEN和ELSE部分？
- 对于多向分支，例如SWITCH/CASE语句，是否给出了默认条件？每个CASE后是否恰当地给出退出处理？
- 有不可达的代码吗？
- 有永远也执行不了的循环吗？
- 有最终条件永远满足不了，使程序进入死循环状态的循环吗？
- 嵌套条件语句的层次是多少？能对代码进行简化以降低复杂度吗？

关于语句

- 代码遵循了组织的编码规则吗？

- 代码遵循了与具体平台有关的编码规则（例如Windows或Swing专用的GUI调用）吗？

关于风格

- 程序中使用了不良的程序设计结构（例如C中的全局变量、COBOL中的ALTER语句）吗？
- 是否使用特定机器体系结构或下层产品指定版本的特殊性质（例如使用"没有列入产品文档中"的功能）？
- 在可读性问题上，例如遵循代码缩进格式规则，给予足够重视了吗？

其他

- 是否检查了内存泄漏（例如分配了内存但没有明确释放）？

关于文档

- 对于代码，特别是逻辑很复杂或对于整个产品正常运行非常关键的代码，是否有足够的注释说明？
- 是否很好地记录了变更历史？
- 文档中很好地说明接口和参数了吗？

3.3 结构测试

结构测试要考虑代码，代码的结构、内部设计以及设计是如何转化为代码的。结构测试和静态测试之间的基本区别是结构测试实际上是由计算机在已构建的产品上运行，而在静态测试中，产品是通过人工只使用源代码，而不是可执行代码或二进制代码进行测试的。

结构测试需要采用一些预先设计的测试用例运行实际产品，考察尽可能多或有必要执行的代码。如果在运行测试用例时，测试用例使程序执行给定部分的代码，那么这部分代码被认为是经过考察的。

本章开始已经说过，结构测试可以进一步分为单元/代码功能测试、代码覆盖和代码复杂度测试。

3.3.1 单元/代码功能测试

单元/代码功能测试是结构测试的基本部分，包括开发人员在将代码提交更深入的代码覆盖测试或代码复杂度测试之前所进行的一些快速检查。单元/代码功能测试有多种方法：

1. 第一步，由于开发人员了解输入变量和对应的预期输出变量，可以执行一些容易的测试。这可以是快速测试，检查所有明显的错误。通过以不同的输入变量取值重复这些测试，开发人员的自信心会不断提高。这种测试甚至可以在静态测试的正式审查之前开展，使正式评审人员不会在处理明显错误上浪费时间。

2. 对于包含复杂逻辑或条件的模块，开发人员可以构造产品的一种"调试版本"，加入一些中间打印语句，以保证程序通过恰当的循环和迭代合适的次数。重要的是一定要在修改了缺陷之后删除这些中间打印语句。

3. 执行初始测试的另一种方法是在调试器下或集成开发环境（IDE）中运行被测产品。这些工具使开发人员能够单步执行语句（使开发人员能够在执行每条语句后停下来，观察或修改变量内容，等），在任意函数或指令处设置断点，并观察各种系统参数或程序变量值。

以上所述活动更像“调试”活动而不是“测试”活动。不管怎样，这些活动都与代码结构知识密切相关，因此本书把这些活动称以“白盒测试”，这与我们的测试包含所有检测和更正产品缺陷的活动的理念是一致的。

3.3.2　代码覆盖测试

由于产品是通过程序代码实现的，如果可以运行测试用例考察代码的不同部分，那么由这些代码实现的产品的那一部分就是经过测试的。代码覆盖测试包括设计和执行测试用例，并确定由测试覆盖的代码百分比。测试的代码覆盖百分比是通过采用叫做代码插桩的技术确定的。插桩重新构建产品，将产品与由工具提供商提供的一组数据库关联起来。插桩后的代码可以监视并记录被覆盖的代码。这种工具还可以列出被频繁覆盖的代码，由此可以确定关键或最常使用的代码。

代码覆盖测试确定以下几类覆盖：

1. 语句覆盖
2. 路径覆盖
3. 条件覆盖
4. 功能覆盖

语句覆盖

大多数传统程序设计语言的程序结构都有以下特征：

1. 串行的控制流
2. 两个方向的判断语句，例如if then else
3. 多个方向的判断语句，例如switch
4. 循环，例如while do、repeat until和for

面向对象的语言具有所有以上特征，此外还有一些其他结构和概念，这些问题将在第11章讨论。这里只讨论传统语言。

语句覆盖就是编写测试用例执行每行程序语句。可以假设覆盖的代码越多程序功能测试得越好，因为程序功能是由代码实现的。根据这个假设，代码覆盖可以通过确定以上每种类型的语句覆盖率实现。

对于由顺序执行（即没有条件分支）的语句组成的代码，可以设计测试用例自顶向下贯穿运行，从头开始的测试用例一般会执行到这段代码的最后。但是，并不总是这种情况。首先，如果程序遇到异步例外（例如除零），即使测试用例是从这段程序的开始部分执行的，该测试用例也可能不会覆盖这段代码的全部语句。因此，即使是在串行语句代码中，也有可能不能实现所有语句的覆盖。其次，一个程序段可能有多个入口点。尽管这并不符合结构化程序设计的要求，但是在早期的程序设计语言中，这种情况很常见。

在考虑像if语句这样的双向判断结构时，我要覆盖所有语句，还要覆盖if语句的then部分和else部分。这意味着对于每个if　then　else，都必须（至少）用一个测试用例测试then部分，（至少）用一个测试用例测试else部分。

对于像switch语句这样的多向判断结构，可以归结为多个双向if语句。因此，为了覆盖所有可能的switch分支，需要多个测试用例。（把这个问题留给您作练习。）

循环结构有更多的问题需要考虑。循环有很多形式，例如for、while、repeat等，都要重复执行一段语句直到一定的条件被满足。很大一部分程序缺陷是由于循环设计错误引起

的。常见的情况是循环没有处理好所谓"边界条件"。一种常见的循环错误是没有描述好循环的终止条件。为了确保提高循环内部的语句覆盖率，所设计的测试用例需要：

1. 完全跳过循环，以测试启动该循环之前循环终止条件就为真的情况。

2. 执行循环的次数为一次到最大次数之间，以检查该循环所有可能的"正常"操作。

3. 尝试覆盖循环的"边界"附近，即如果循环n次，则分别测试循环恰好低于n、等于n和恰好高于n次。

程序的语句覆盖用一组测试实际执行过的语句百分比的指标表示，可以用以下格式计算。

语句覆盖率=（所考察的总语句/程序中可执行的总语句）×100

从以上讨论不难看出，语句类型从简单的串行语句到if…then…else语句，再到循环，实现语句覆盖所需的测试用例数在增加。从第1章介绍的Dijkstra定律我们知道，就像不可能穷尽地测试所有可能的输入数据一样，穷尽地覆盖程序的所有语句在实践中也是不可能的。

即使可以得到很高的语句覆盖率，也并不意味着程序就没有缺陷了。首先，考虑一个假想的例子，假设已经达到百分之百的代码覆盖率。如果程序执行了错误的需求，而且实现这个错误需求的代码经过"彻底测试"，达到百分之百代码覆盖率，然而这仍然是一个错误的程序，因此，百分之百代码覆盖率并不能说明一切。

其次，考虑以下例子：

```
total = 0; /*把total设置为零*/
if (code == "M") {
   语句1;
   语句2;
   语句3;
   语句4;
   语句5;
   语句6;
   语句7;
}
else percent = value/total*100; /*除以零*/
```

在上面的例子中，如果用code = "M"测试，则可以得到80%的代码覆盖率。但是，如果在现实世界中数据的分布为在90%的时间code取值不是="M"，则这段程序有90%的概率出错（由于最后一条语句出现除以零的情况）。因此，即使代码覆盖率达到80%，仍然会留下出错概率90%的缺陷。以下要讨论的路径覆盖要解决这个问题。

路径覆盖

在路径覆盖中，把程序分为若干不同的路径。程序（或程序的一部分）可以从头开始执行，并选择任何一条路径到达结尾。

路径覆盖率=（所考察的总路径/程序中的总路径）×100

下面看一个日期确认的例程，日期是用三个字段接收的，即mm，dd和yyyy。假设对这个例程输入之前，已经确认输入值的类型是数字。为了简化讨论，假设有一个叫做leapyear的函数，如果给出的年份是闰年，则函数返回TRUE。还有一个叫做DayofMonth的数组，包含每个月的天数。图3-2给出了这个例程经过简化后的流程图。

从图3-2中可以看出，程序中有不同的路径可以走，每条路径用花体字标出。路径包括：

- A
- B-D-G
- B-D-H
- B-C-E-G
- B-C-E-H
- B-C-F-G
- B-C-F-H

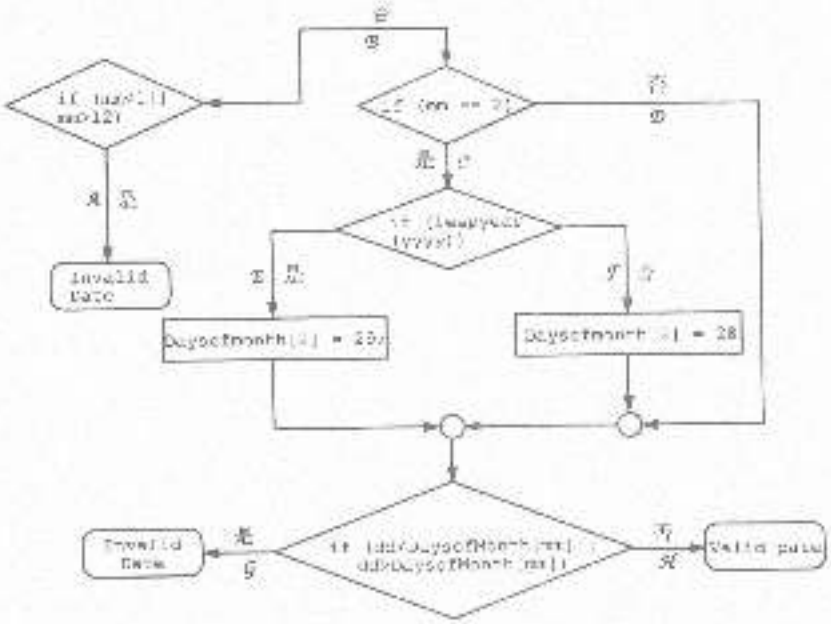


图3-2　日期确认例程的流程图

不考虑每条路径中的语句数。如果可以执行这些路径，就说覆盖了大多数典型的场景。

与语句覆盖相比，路径覆盖给出了更强的覆盖条件，因为路径覆盖要考虑程序中的各种逻辑路径，而不只是程序语句。

条件覆盖

在上面的例子中，即使覆盖了所有可能的路径，仍然不能说程序已完全测试了。例如，可以通过为mm赋一个取值小于1的数（例如0）使程序走路径A，发现可以覆盖路径A，程序检测出该月份值非法。但是，这样还没有测试其他条件，即mm>12的情况。不仅如此，大多数编译器都进行优化，以减大限度地压缩布尔操作的数量，可能没有计算所有的条件，即使选择了正确的路径。例如，如果有一个OR条件（以上例子的第一个IF语句），一旦发现IF语句的第一部分（例如mm<1）为真，就完全不会再计算IF语句的第二部分，因为总的布尔值为TRUE。类似地，如果布尔表达式中有AND条件，当第一个条件为FALSE时就完全不需要计算表达式的其他部分。

条件覆盖率=（所涉及的总判断/程序中的总判断）×100

由于所有这些原因，路径测试可能不是充分的。需要设计测试用例执行每个布尔表达式，既要测试TRUE路径，也要测试FALSE路径。显然，这意味着要设计更多的测试用例，而且测试用例的数量会随着

布尔表达式数量的增加而指数性地增加。但是在实际工作中，这种情况并不是很糟糕，因为这些条件通常存在相互依赖。

如边栏公式定义的条件覆盖率给出了由一组测试用例覆盖的条件百分比指标。条件覆盖率是比路径覆盖率强得多的准则，路径覆盖率又是比语句覆盖率强得多的准则。

功能覆盖 这是一种新的结构测试，用来指示测试用例覆盖了多少程序功能（类似C语言中的函数）。

产品的需求在设计阶段映射到功能，每个功能构成一个逻辑单元。例如，在数据库软件中，"在数据库中插入一行"可以是一个功能，而在工资管理应用系统中，"计算税额"可以是一个功能。每个功能反过来又可以通过其他功能实现，通过功能覆盖率，可以设计测试用例考察代码中的每个不同功能。与其他类型的覆盖率相比，功能覆盖率的优点有：

1. 功能在程序中很好确定，因此更容易编写测试用例，增加功能覆盖率。

2. 由于与代码相比功能处于高得多的抽象层次，因此，达到百分之百的功能覆盖率比达到百分之百的其他覆盖率更容易。

3. 功能对需求的映射更有逻辑性，因此可以给出产品测试覆盖率更直接的关联性。下一章将讨论需求跟踪矩阵，用来完成设计、编码和测试阶段对需求的跟踪。功能提供了实现这种跟踪的一种手段，功能覆盖率提供了测试这种跟踪性的方法。

4. 由于功能是实现需求的一种手段，因此可以根据其所实现的需求重要性确定功能的优先级，从而更容易确定功能测试的优先级。前面介绍的覆盖率方法不一定能做到这一点。

5. 功能覆盖提供到黑盒测试的自然过渡。

我们还要度量给定功能被调用的次数，这个指标可以指示哪个功能最常使用，因此该功能会成为所有性能测试和优化的目标。例如，如果在一个网络软件中发现最经常使用的是数据包封装和解封的功能，那么花费额外的精力改进这个功能的质量和性能是合适的。因此，功能覆盖除了能够帮助改进产品的质量，还能够帮助改进性能。

> 功能覆盖率=（所执行的总功能/程序中的总功能）×100

小结

代码覆盖测试包括采用预先编写的测试用例执行产品，并找出对代码的覆盖情况的"动态测试"。如果要求更高的代码覆盖率，则可能需要多轮的测试，对于每轮测试，测试人员都要仔细研究上一轮的统计数据，并编写新的测试用例，以覆盖没有被以前的测试用例覆盖的代码。为了完成这类测试，测试人员不仅需要了解代码和逻辑，还要了解如何编写能够覆盖更多代码的有效的测试用例。这类测试也可以叫做"灰盒测试"，因为这类测试为了提高有效性，综合使用了"白盒和黑盒测试方法论"（白+黑=灰）。

因此，更高的代码覆盖率源自对代码流的更好理解和编写有效的测试用例。一般可以达到40%－50%的代码覆盖率，超过80%的代码覆盖率就要花费大量的精力和理解代码。

本书到这里为止介绍过的多种代码覆盖技术并不是相互排斥的，而是相互补充、完善。虽然语句覆盖可以提供一种基本指标，但是路径、判断和功能覆盖通过考察各种逻辑路径和功能可以提供更具参考价值的指标。

以上讨论研究了针对各种功能需求的代码覆盖测试的使用，这些测试方法还有其他用途。

- **性能分析和优化** 代码覆盖测试可以确定最常执行的代码区域，这样就可以更关注这部分代码。如果再作努力也不能再提高其性能，那么可以考虑使用其他策略，例如缓存。代码覆盖测试可以提供有助于做出这种面向性能决策的信息。

- **资源使用分析** 白盒测试特别是通过插桩后的代码进行的白盒测试，有助于确定资源使用方面的瓶颈。如果发现特定资源，例如RAM或网络是瓶颈，那么经过插桩的代码可以帮助确定瓶颈的具体位置，从而提出可能的解决方案。
- **检查关键代码或与并发处理有关的代码** 关键代码是不能由多个进程同时执行的代码。采用插桩代码进行的覆盖测试是确定这种关键代码并发约束冲突的最佳手段之一。
- **确定内存泄漏** 由进程申请分配（例如通过C语言中的malloc）的每块内存都需要显式地释放（例如通过C语言中的free）。如果不这样，所分配的内存就会"丢失"，可使用的内存就会逐渐减少。经过一段时间后，会没有内存可分配给新提出的内存申请，各个进程会因此而出现问题。各种白盒测试方法有助于确定内存泄漏。大多数调试器或插桩代码可以检查内存的分配和释放操作是否成对。
- **动态生成的代码** 白盒测试有助于有效地确定安全漏洞，特别是动态生成的代码的安全漏洞。在动态生成和执行的代码中，需要动态测试生成代码的功能。例如，在使用Web服务时，可能出现从用户处接受一些参数并生成html/java代码，再传送给远程计算机执行的情况。由于执行了这种事务或服务后生成代码就马上执行，因此测试生成代码要求测试人员具有编码知识。因此，本章讨论过的各种白盒测试技术就派上了用场。

3.3.3　代码复杂度测试

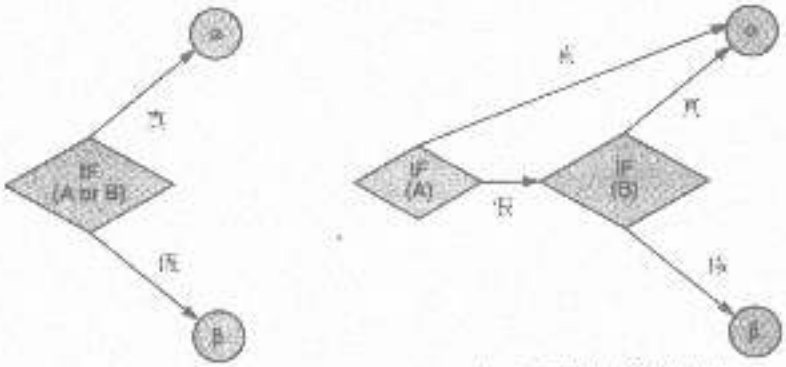
前面几节讨论了可用来测试程序的不同类型的覆盖，但是运用这些覆盖有两个问题：

1. 哪些路径是独立的？如果某条路径不独立，那么就有可能减小化测试数量。
2. 为了保证所有指令至少执行一次，是否存在必须运行的最少测试数的边界？

圈复杂度是一种量化程序复杂度的指标，可以回答上面两个问题。

程序可以用流程图表示。流程图由节点和边组成。为了把标准的流程图转换为一种可以计算圈复杂度的流程图，可以采用以下步骤：

1. 找出程序中的谓词或判断点（一般是布尔表达式或条件语句）。
2. 确保这些谓词是简单的（即在每个谓词中没有and或or等）。图3-3给出了把包含or的谓词分解为简单谓词的方法。类似地，如果有循环结构，要把循环终止检查分解为简单的谓词。



a）带有布尔OR的谓词　　b）一组等效的简单谓词

图3-3　将OR流程图转换为简单谓词

3. 将所有顺序语句合并为一个节点，因为一旦开始这些语句都要执行。
4. 如果一组串行语句后面接简单谓词（如以上第2点所述），把所有串行语句和谓词检查

合并成为一个节点，并通过这个节点引出两条边。这种拥有两条边的节点叫做谓词节点。

5. 确保所有边终止于一个节点上，在程序结尾处增加一个节点表示所有串行语句。

采用以上转换规则将传统流程图转换为如图3-4所示的流程图。参见彩图，读者可以看得更清晰。图3-4a中用深浅色区分的流程图元素映射到图3-4b中对应颜色的流程图节点元素。

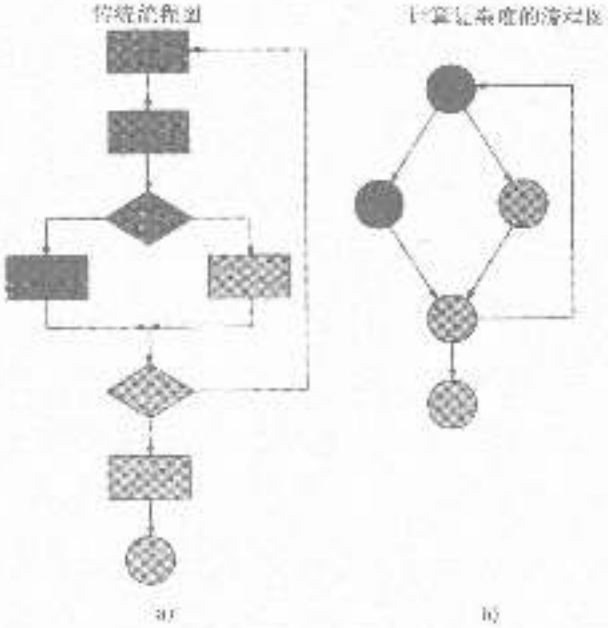


图3-4 将传统流程图转换为可计算复杂度的流程图

开始，流程图和圈复杂度直观地反映了程序逻辑流程的复杂性和程序中的独立路径数量。对于复杂度和独立路径的主要贡献是程序的判断点。考虑一个没有判断点的假想程序，这种程序的流程图如图3-5所示，拥有两个节点，一个对应代码，一个对应终止节点。由于所有串行步骤都合并为一个节点（第一个节点），因此只有一条边将两个节点连接起来，这条边是唯一的独立路径。因此，对于这个流程图来说，圈复杂度等于1。

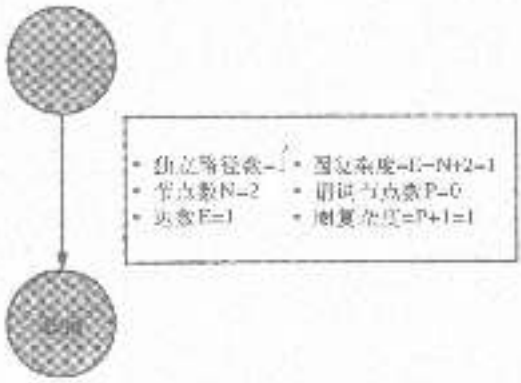


图3-5 没有判断点的假想程序

这个流程图没有谓词节点，因为没有判断点。因此，圈复杂度还等于谓词节点数(0)+1。

圈复杂度=谓词节点数+1

圈复杂度=E−N+2

请注意，在这个流程图中，边(E)=1，节点(N)=2，圈复杂度还等于1=1+2−2=E−N+2。

如图3-6所示，如果在流程图中增加谓词节点，明显就会有两条独立路径，一条对应布尔条件为TRUE，一个对应布尔条件为FALSE。因此，图3-6的圈复杂度是2。

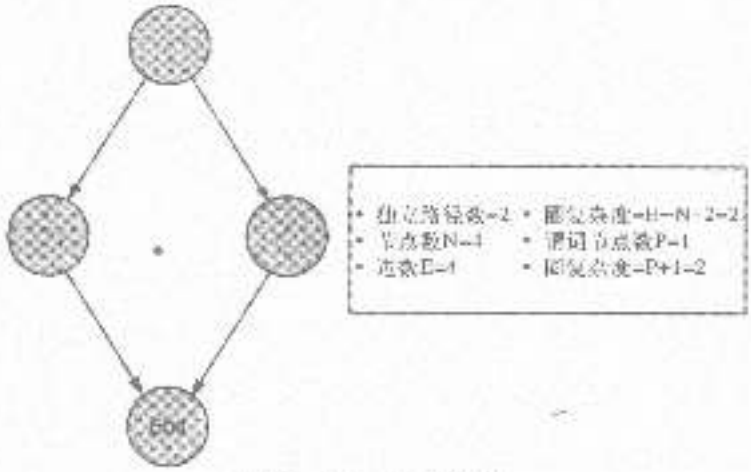


图3-6　增加一个判断节点

这个独立路径数是2，恰好又等于谓词节点数（1）+1。如果增加一个谓词节点（一个有两条边的节点），圈复杂度就增加1。因为E−N+2公式中的“N”不变，“E”加1，结果，使用E−N+2这个公式也可以算出圈复杂度。

从以上解释读者可能对圈复杂度两种不同的计算方法和对应的确定程序独立路径的圈复杂度有了一定的认识。本书归纳了这些公式，但没有进行证明。可以肯定的是这些公式非常有用。

以上两个公式提供了通过流程图计算圈复杂度的一种简单方法。实际上第一个公式甚至在没有流程图的情况下也可以使用，因为可以直接统计基本谓词的数量，根据图论基本理论还可以导出其他圈复杂度公式，这里就不介绍了，感兴趣的读者可以参考本章后面的参考文献。

使用流程图，独立路径可以定义为流程图中至少有一条没有被其他路径遍历过的边的路径。一组覆盖所有边的独立路径叫做基本集。一旦形成基本集，就可以编写测试用例，考察基本集中的所有路径了。

计算和使用圈复杂度　小程序可以手工计算其圈复杂度，但是在工程中，有数千行代码的程序需要使用自动化工具。为大型程序手工绘制流程图非常困难，市场上已经有多种工具可以计算圈复杂度。但是需要注意，在构建了模块之后才计算其复杂度并进行测试可能已经太晚了，因为复杂模块测试之后就可能难以重新设计了。因此，在启动测试阶段之前（甚至在编码阶段之前）必须进行某些基本的复杂度检查，这种检查可以是代码审查中的一个内容。根据通过工具得到的圈复杂度数，可以采用如表3-1列出的复杂度度量的一些对应措施。

表 3-1

复杂度	可能的含义
1-10	代码编写良好，有很高的可测试性，维护所需的成本和工作量很低
10-20	比较复杂，可测试性中等，维护所需的成本和工作量中等
20-40	非常复杂，可测试性很低，维护所需的成本和工作量很高
>40	不可测，不管你愿意投入多少人力，物力都不够

3.4 白盒测试中的挑战

白盒测试要求测试人员具有很好的程序代码和程序设计语言知识。这意味着开发人员要密切参与白盒测试。一般来说，开发人员不喜欢进行测试，不仅对于诸如评审这样的静态测试是这样，对于结构测试也是这样。此外，由于进度压力很大，程序员可能"抽不出时间"进行评审（一种花更多时间编写代码的托辞）。第13章在讨论人员问题时，还要深入讨论测试和开发之间的关系问题。

- **由于人的本性，开发人员不能发现自己代码中的缺陷** 正如前面已经讨论过的，我们大多数人都难以发现自己产品中的错误。由于白盒测试需要编写代码的程序员参与，因此他们不能最有效地检测自己工作产品中的缺陷的可能性相当大。引入独立视角可能会有所帮助。
- **在现实中可能很难进行完全的代码测试** 程序员一般不那么热心引入外部（客户）视角或领域知识可视化产品在实际环境中部署的情况。这可能意味着即使经过大量测试，也可能遗漏一些常见的用户场景，可能会造成缺陷遗漏。

这些挑战并不意味着白盒测试就是无效的，而是说如果在进行白盒测试时通过其他测试手段应对这些挑战，则有可能使测试更有效。下一章要讨论的黑盒测试可以在一定程度上应对这些挑战。

问题与练习

1. 本书讨论了如何将布尔or转换为可以用于导出流程图的简单调词。按照类似的方法将表达式if A and B转换为流程图。
2. 以下是一段简单的C语言程序，用于接受一组输入并计算标准差。这段程序中藏有故意设置的缺陷。请对这段程序进行白盒测试，找出程序的缺陷。此外，列出找出这些缺陷所使用的方法。

```
用于计算标准差的代码——白盒2和白盒3
# include <stdio.h>
void main (argc, argv)
   {
     int no_numbers;  /* 标准差计算的数字个数*/
     float mean, sd&av;  /* 临时变量 */
     int i, total;  /* 循环变量 */
     int numbers[100]; /* 实际数字 */
     printf ("请输入数字个数\n");
     scanf ("%d", &no_numbers);
     /* 读取数字，然后计算均值 */
```

```
    for ( i=0; i<no_numbers; i++)
        {
            scanf ( "%d", &numbers[i]);
            total += numbers[i];
        }
    mean = total/no_numbers;
    /* 以下开始计算标准差 */
    total=0;
    for(i=0; i<no_numbers; i++)
       {
          total +=
                ((mean - numbers[i])*(mean - numbers[i]))
       }
       stdev = total/no_numbers;
       printf("标准差是 %d\n", stdev);
    return (0);
    }
```

3. 针对上面的程序画出流程图，并计算该程序的圈复杂度。
4. 以下是一段C语言程序，用于从一个链表中删除一个元素。请设计一组测试数据覆盖这段程序的每条语句。

```
用于删除双向链表中的元素的代码　　问题4的答案
/* 假设链表的定义由以下结构给出:
struct llist {
                int value;
                llist *next;
                llist *prev;
                }
使用多重指针传递一个指向llist开始的指针和要删除的值。假设最多只能删除一个值//
void delete_list (llist ** list, int value_to_be_deleted)
   {
    llist* temp;
for (temp = list; temp != NULL, temp=temp->next)
{
  if (temp->value==value_to_be_deleted)
      if(temp->prev != NULL)
           temp->prev->next = temp->next;
      if(temp->next != NULL)
           temp->next->prev = temp->prev;
      return(0);
       }
  return(1);/* 要删除的值到搜索结束时也没有找到 */
  }
```

5. 在问题4，请说明即使测试数据提供了100%的语句覆盖，仍然有尚未发现的缺陷。
6. 以下给出的是由两位学生实际编写的用于日期确认的程序。这两位学生使用了两种不同的方法编写了代码，分别如程序6a和6b所示。你认为从白盒测试的观点看，哪段代码更容易测试？在本章讨论的白盒测试技术中，哪种技术更适合6a，哪种技术更适合6b？对于两段

代码，如何从更不容易引入缺陷和检测缺陷方面提高其有效性？

程序6a 用于确认日期的代码——问题6

```
int month_days [13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
valid = TRUE;
if (is_leap-year(yy))
     month_days [2] = 29;
if (mm < 1|| mm > 12) valid = FALSE;
else
    if (day < 1 || day > month(month_length[month]))
        valid = FALSE;
    else if (year < 1)
        valid = FALSE;
return (VALID);
/* 前面使用的函数is_leap-year定义如下 */
int is_leap-year (int year)
{
  int result;
  if ((year%4) != 0) result = FALSE;
  else (if (( year %100) == 0) result = TRUE;
  else if (( year %100) == 0) result = FALSE;
return (result);
}
```

程序6b 用于确认日期的代码——问题6

```
if (!(dd > 0 && dd < 32))
   printf ("只输入1到32之间的数字\n");
   return (1);
if ((mm = 4) || (mm == 6) || (mm == 9) || (mm == 11)) && day > 30)
   {
      printf ("非法日期，最多天数是30");
      return (1);
   }
if (( I = ( year%100 != 0) && (year%4 == 0) || (year%400 == 0))||
            &&   (day <= 29)|   || (day < 29))
   {
    printf("合法日期\n");
    return (0)
   }
else
    {
    printf("非法日期\n");
    return (1);
}
```

7. 对于程序6b，采用表P1给出的测试数据测试对闰年的确认。请计算这些测试数据对于代码闰年确认部分的各种覆盖率。

表P1　用于测试程序8b的测试数据

数　据	选择理由
29 Feb 2003	检查非闰年的天数大于28
28 Feb 2002	检查2003年之闰年，天数等于28是合法的
29 Feb 2000	检查闰年天数范围
30 Feb 2000	检查闰年天数范围
14 Feb 2000	检查闰年天数范围的天数
29 Feb 1896	检查对于闰年天数等于29是合法的

8. 请编写一段简单的矩阵相乘程序，并考虑尽可能多的有效和无效条件验证问题。确定在测试这段程序要使用的测试数据，并给出理由。
9. 嵌入代码是不修改自身的程序，而非嵌入代码是要修改自身的程序。请讨论从白盒测试角度和未来这种程序的维护角度看，会面临什么问题。
10. 请讨论从白盒测试的角度看以下结构存在什么负面作用：
    a. GOTO语句
    b. 全局变量
11. 请比较功能覆盖和其他形式覆盖的优缺点。

第4章 黑盒测试

4.1 黑盒测试的定义

黑盒测试使用规格说明，不要求考察程序代码。黑盒测试以客户的观点进行，进行黑盒测试的测试工程师只知道输入信息和预期的输出，不知道这些输入是如何被软件转化为输出的。

实施黑盒测试不需要被测系统的内部知识。

黑盒测试对于管理来说是很方便的，因为测试人员使用完全完成了的产品，不要求产品结构的知识。独立测试实验室可以管理黑盒测试，保证功能和兼容性。

拿锁和钥匙打个比方。我们并不知道锁内的机关是怎样起作用的，只知道输入（钥匙的数量、使用钥匙的具体顺序以及每把钥匙的转动方向）和预期的输出（上锁和开锁）。例如，如果钥匙顺时针转动就开锁，如果逆时针转动就上锁，使用锁不需要了解锁内部的结构，也不需要了解锁的工作原理。但是必须知道锁和钥匙系统的外部功能。以下给出使用锁需要知道的一些功能。

功　能	用户须知
锁的特性	锁是金属制品，有一个用于上锁的钥匙孔，能够插入钥匙，并使其可以顺时针或逆时针旋转
钥匙的特性	钥匙是金属制品，可以和特定的钥匙孔匹配
要执行的动作	上锁时，插入钥匙并顺时针转动 开锁时，插入钥匙并逆时针转动
状态	已上锁 已开锁
输入	钥匙顺时针或逆时针转动
预期输出	锁上 打开

有了以上这些知识，就可以在购买锁之前用合理的方法测试锁和钥匙，而不必是了解锁、钥匙和内部机关的机械专家。这个概念可以扩展到软件测试中的黑盒测试。

因此，黑盒测试要求有关被测产品的功能知识，不一定了解系统的内部逻辑，也不一定了解构建该产品所使用的程序设计语言。前面介绍的测试锁的例子关注的是测试产品（锁和钥匙）的特性、不同的状态，以及已知的预期输出。可以检查锁是否能够被其他钥匙（而不是原配钥匙）打开和锁上，还可以尝试用发卡或其他细小金属片检查上锁和开锁。以下将进一步详细介绍可以针对给定产品实施的不同种类的测试。

4.2 黑盒测试的意义

黑盒测试有助于对被测产品进行总体功能验证。

黑盒测试基于需求实施 黑盒测试除了可以发现各种有关系统整体上的问题，还有助于发现不完全、不一致的需求。

黑盒测试既检查已描述的需求还检查隐含需求 并不是所有需求都经过明确描述，有些需求是隐含的。例如：需要在报表上给出日期、页眉和页脚可能没有在报表生成需求的规格说明中明确给出，但是，这些功能在向客户交付产品时应该提供，以提供更好的可读性和可用性。

黑盒测试要包括最终用户视角 由于要从外部视角测试产品的行为，因此，最终用户视角是黑盒测试的一个组成部分。

黑盒测试采用有效和无效输入 用户在使用产品时出错是很正常的，因此，黑盒测试只采用有效输入是不够的，从最终用户视角实施的测试包括测试这些操作错误或无效条件，这可以确保在有效情况下产品行为与期望的相同，并且在无效输入条件下不会挂起或死机。这些称做正面和负面测试用例。

测试人员可以知道也可以不知道产品使用的技术或内部逻辑。但是，如果了解这些技术有助于构建针对特定易出现错误区域的测试用例。

规格说明就绪后就可以生成测试场景。由于需求规格说明是黑盒测试的主要输入，因此测试设计可以在生存周期的早期开始。

4.3 黑盒测试的时机

不管项目选择的是什么软件开发生存周期模型，黑盒测试活动都要求测试团队从软件项目生存周期的开始就介入。

测试人员可以从被测系统的需求获取和分析阶段就介入，在软件处于设计阶段时，就可以启动测试生存周期的测试构建阶段，准备测试场景和测试数据。

一旦代码完成并交付测试就可以执行测试。在测试构建阶段开发的所有测试场景都要执行，通常选择这些测试场景的一个子集进行回归测试。

4.4 黑盒测试的方法

黑盒测试以一种系统化的方式依据规格说明生成测试用例，以避免出现冗余并提供更高的覆盖率。

第1章已经介绍过，不管产品如何简单，要想穷尽地测试也是不可能的。因为黑盒测试是测试软件的外部功能，因此需要在尽可能少的时间内，编写出测试尽可能多的外部功能的测试用例，发现尽可能多的缺陷。虽然这看上去像是空想，不过本节将要介绍的技术可促进达到这些目标。本节介绍各种测试场景生成技术，以有效实施黑盒测试。

这里要讨论的各种技术包括：

1. 基于需求的测试
2. 正面测试和负面测试
3. 边界值分析
4. 决策表
5. 等效类划分

6. 基于状态的测试
7. 兼容性测试
8. 用户文档测试
9. 领域测试

4.4.1 基于需求的测试

基于需求的测试确认软件系统的软件需求规格说明（SRS）给出的需求。

前面几章已经提到过，并不是所有需求都明确给出，有些需求是隐含的。明确描述的需求是写入需求规格说明文档中的需求，隐含需求是没有写入文档但是预期要包含到系统中的需求。

需求测试的前提是需求规格说明要经过仔细评审。需求评审要确保需求是一致的、正确的、完备的和可测试的，这个过程要保证一些隐含需求被转换为明确需求，并形成文档，从而使需求更清晰，使基于需求的测试更有效。

有些公司对这种方法进行了修改，为需求增加更多细节。汇集所有明确需求（通过系统需求规格说明得到）和隐含需求（由测试团队导出），并形成“测试需求规格说明（TRS）”的文档。基于需求的测试还可以以TRS为基础实施，因为TRS反映了测试人员的视角。但是为了简化起见，这里把SRS和TRS看作是同一份文档。

前面提到的锁和钥匙例子的需求规格说明可以形成表4-1给出的文档。

表4-1 锁和钥匙系统的需求规格说明样本

序号	需求标识	描　述	优先级（高、中、低）
1	BR-01	插入号码为123-456的钥匙并顺时针转动，应能上锁	高
2	BR-02	插入号码为123-456的钥匙并逆时针转动，应能开锁	高
3	BR-03	其它号码为123-456的钥匙可以用来上锁和开锁	高
4	BR-04	其他东西都不能用来上锁	中
5	BR-05	其他东西都不能用来开锁	中
6	BR-06	即使受到生物指纹锁也不能被上开	中
7	BR-07	钥匙和锁都必须是金属制品，重量必须为150克左右	低
8	BR-08	开锁和上锁的钥匙转动方向应能够改变，以便于左和手人士使用	低

像上面这样的需求由“需求跟踪矩阵”（RTM）进行跟踪，RTM实施从开始贯穿设计、开发和测试的所有需求的跟踪。这个矩阵会在产品生存周期内不断进化。首先，每个需求除了都提供简要描述外还要赋给一个唯一标识。需求标识和描述可以来自需求规格说明（表4-1），或任何其他列出产品要测试的需求的可用文档。在上面的表中，命名规则使用了前缀“BR”，后接一个两位数字。BR表示测试的类型，即“黑盒-需求测试”。两位数字是需求的顺序号。对于更复杂的系统，代表模块和模块内连续序列号的标识号（例如INV-01、AP-02等）可用来标识需求。每个需求都被指派一个需求优先级，分为高、中、低。优先级不仅可以确定功能开发的资源保障顺序，还可以用于确定测试用例的执行顺序。针对高优先级需求的测试应该在低优先级需求对应的测试之前实施。这可以保证最高风险的功能在生存周期内更早测试，这样在测试中发现的缺陷能够尽早得到更正。

随着产品生存周期和测试阶段的向前推进，在RTM中记录需求和后续阶段的交叉引用。在前面给出的例子中，只给出需求和测试的映射，在更完整的RTM中，会有表示需求到设计

和代码映射对应的列。

“测试条件”这一列给出了测试需求的不同方法。测试条件可以通过本章介绍的技术得出，所有测试条件的标识使人可以感觉到没有遗漏任何在最终用户环境中可能产生问题的场景。这些条件可以合并在一起，构成一个测试用例，也可以每个测试条件映射到一个测试用例。

“测试用例标识”列可以用来完善测试用例和需求之间的映射。测试用例标识应遵循一定的命名规则，以提高其可用性。例如，在表4-2中，测试用例顺序编号，前缀是产品名称。对于由多个模块组成的更复杂的产品，测试用例标识可以由模块代码和顺序号组成。

表4-2　需求跟踪矩阵样本

需求标识	描　述	优先级（高、中、低）	测试条件	测试用例标识	测试阶段
BR-01	输入号码为123-456的钥匙并顺时针转动，应能上锁	高	使用号码为123-456的钥匙	Lock_001	单元、组件
BR-02	插入号码为123-456的钥匙并逆时针转动，应能开锁	高	使用号码为123-456的钥匙	Lock_002	单元、组件
BR-03	只有号码为123-456的钥匙可以用来上锁和开锁	高	使用号码为123-456的钥匙上锁 使用号码为123-456的钥匙开锁	Lock_003 Lock_004	组件
BR-04	门由两侧都不能用来上锁	中	使用号码为789-001的钥匙 使用发卡 使用螺丝刀	Lock_005 Lock_006 Lock_007	集成
BR-05	门从两侧不能用来开锁	中	使用号码为789-001的钥匙 使用发卡 使用螺丝刀	Lock_008 Lock_009 Lock_010	集成
BR-06	即使受到撞击和振动，锁仍能打开	中	使用有关测试	Lock_011	系统
BR-07	锁扣的尺寸必须是金属的[illegible]，至少必须为150克力	低	使用适当仪器	Lock_012	系统
BR-08	开锁和上锁的钥匙转动方向应能修改为，以便于左撇子人士使用	低			未实现

编写完测试用例，RTM还有助于标识需求和测试用例之间的关系。可以使用以下组合：

- 一对一——每个需求对应一个测试用例（例如BR-01）
- 一对多——每个需求对应多个测试用例（例如BR-03）
- 多对一——一组需求由一个测试用例测试（表4-2中没有这种情况）
- 多对多——多个需求由多个测试用例测试（这种测试用例在集成和系统测试中会出现，但是RTM不适合这种组合）
- 一对空——需求没有测试用例，由于没有实现或需求的优先级低，测试团队可以决定不测试一些需求（例如BR-08）

需求制约着多个阶段的测试，即单元、组件、集成和系统测试。对应的测试阶段可以是需求跟踪矩阵中的一列，这一列指示需求测试的时机，以及需求需要考虑进行测试的测试阶段。

RTM在基于需求的测试中具有重要作用：

1. 不管需求有多少，理想情况下每个需求都必须测试。如果有大量需求，可能难以手工跟踪每个需求的测试状态。RTM提供了一种工具，可跟踪每个需求的测试状态而不会遗漏任何（关键）需求。

2. 通过确定需求的优先级，RTM使测试人员能够优先执行优先级高的测试用例，以尽早发现高优先级区域内的缺陷。需求优先级还可用于判定高优先级需求的测试用例是否充分，并调整低优先级需求的测试用例数。此外，如果测试时间非常紧张，优先级的划定也可以确保选择合适的功能进行测试。

3. 测试条件可以合并一起，创建一组测试用例，或对应一个测试用例。从RTM中可以看出对应特定需求的测试用例清单。

4. 测试条件/用例可以用作测试工作量/进度估计的输入。

需求跟踪矩阵为各种测试指标提供大量信息。有些指标可以通过需求跟踪矩阵采集或导出，例如：

- 按优先级分类的需求数——这个指标有助于了解基于需求的覆盖率。覆盖高优先级需求的测试用例数与针对低优先级需求的测试用例数形成对比。
- 按需求分类的测试用例数——为每个需求创建的测试用例数。
- 已设计的测试用例数——为所有需求设计的测试用例总数。

执行这些测试用例后，测试结果可以用来采集指标，例如：

- 已通过的测试用例（或需求）总数——一旦执行完成，就可以得到已通过的测试用例总数及其对应的需求总数。
- 未通过的测试用例（或需求）总数——一旦执行完成，就可以得到未通过的测试用例总数及其对应的需求总数。
- 需求中的缺陷总数——每个需求对应的缺陷清单（需求的缺陷密度）。这个指标有助于进行影响分析：哪个需求的缺陷多，对客户有什么影响。低优先级需求出现相对较高的缺陷密度是可以接受的。高优先级需求中出现高缺陷密度则认为风险很大，可能不能允许产品发布。
- 已完成的需求数——已经成功地完成，且没有任何缺陷的需求总数。
- 挂起的需求数——由于存在缺陷而挂起的需求数。

基于需求的测试要确保产品符合需求规格说明的符合性。

第17章将详细讨论可采集的指标，以及对这些指标的采集和分析方法。

为了说明如何进行指标分析，假设采集到如表4-3所示的测试执行数据。

表4-3 测试执行数据样本

序号	需求标识	优先级	测试用例	关联测试用例数	通过的测试用例	未通过的测试用例	通过率%	缺陷数
1	BR-01	高	Lock_01	1	1	0	100	1
2	BR-02	高	Lock_02	1	1	0	100	1
3	BR-03	高	Lock_03,04	2	1	1	50	3
4	BR-04	中	Lock_05,06,07	3	2	1	67	5
5	BR-05	中	Lock_08,09,10	3	3	0	100	1
6	BR-06	低	Lock_11	1	1	0	100	1
7	BR-07	低	Lock_12	1	1	0	100	0
8	BR-08	低		0	0	0	0	1
合计	8			12	10	2	83	12

从表4-3可以看出，关于需求可以得到以下结论：

- 占总数83%的已通过测试用例对应71%已满足的需求（7个需求有5个满足，有一个需求没有实现）。类似地，从未通过测试用例一列也能得知这些缺陷影响了29%（100-71）的需求。
- 有一个高优先级需求BR-03没有通过测试。对应共有三个需要研究的缺陷，其中有些缺陷需要改正，测试用例Lock_04需要再次执行，以验证该需求是否得到满足。请注意，并不是所有三个缺陷都需要改正，因为有些缺陷可能是无关紧要的或影响不大的。
- 有一个中优先级需求BR-04没有通过测试。对应这个需求的缺陷（共有5个）改正后需要再次执行测试用例Lock_06。
- 需求BR-08没有满足，不过对于这个发布版本这个问题可以忽略。尽管对应这个需求存在一个缺陷，但是这个需求的优先级很低。

以上讨论的指标可以通过图表示。这种图直观地给出大量信息，如图4-1所示，显然需求BR-03和BR-04有未通过的测试用例，因此需要修改。

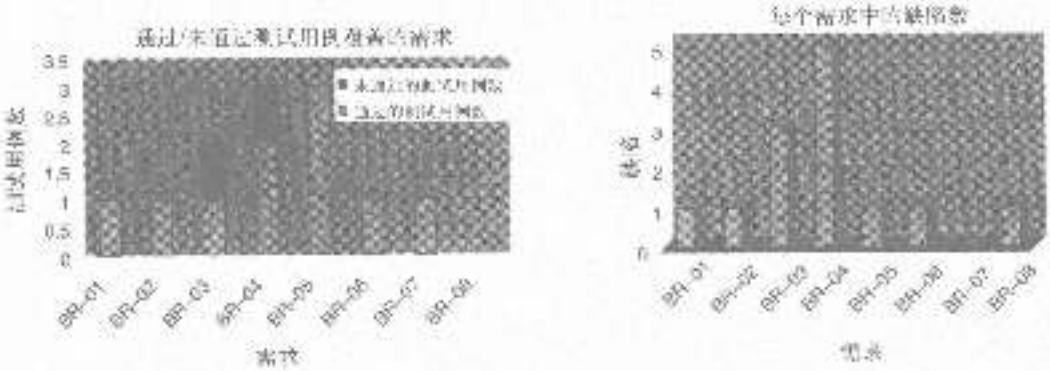


图4-1　表示测试用例执行结果的图

4.4.2　正面和负面测试

正面测试试图证明给定产品可以完成所期望完成的工作。如果测试用例通过一组预期输出验证产品的需求，则称为正面测试用例。正面测试的目的是证明产品对于每条规格说明和预期都能通过。如果预期产品给出一个错误时它确实给出一个错误，这也是正面测试的一部分。

因此，可以说正面测试要检查产品针对正面和负面条件的行为是否与需求所描述的一致。

对于锁和钥匙这个例子下面给出了一组正面测试用例。（有关需求规格说明请参阅表4-2。）

请看表4-4的第一行。这时锁处于开锁状态，使用号码为123-456的钥匙顺时针转动。预期的输出是将其锁上。在执行测试时，如果测试结果是锁上，则这个测试用例通过。这是正面测试的"正面测试条件"的一个例子。

表4-4　正面测试用例举例

需求标识	输入1	输入2	正面状态	结果状态
BR-01	号码为123-456的钥匙	顺时针转动	开锁	上锁
BR-01	号码为123-456的钥匙	逆时针转动	上锁	不变
BR-02	号码为123-456的钥匙	逆时针转动	开锁	不变
BR-02	号码为123-456的钥匙	顺时针转动	上锁	开锁
BR-04	发卡	顺时针转动	上锁	不变

请看表4-4的第五行，锁处于上锁状态，使用发卡并顺时针转动不会引起锁的状态改变，也不会给锁造成损坏。执行测试时一切没有改变，则这个正面测试用例通过。这是正面测试的“负面测试条件”的一个例子。

负面测试展示当输入非预期输入时产品没有失败。负面测试的目的是尝试使系统崩溃。负面测试使用产品没有设计和编码的场景。换句话说，输入值可能没有在产品的需求规格说明中描述。就规格说明来说，这些测试条件可以归纳为产品的未知条件，但是对于最终用户，会遇到多种产品需要考虑的场景，了解可能在最终用户层面上出现的负面情况，以便测试并采取预防措施，这一点对于测试人员来说甚至更重要。负面测试检验产品是否在应该提供错误信息时而没有提供，或不应该提供错误信息时却提供了。

正面测试用于验证已知测试条件，负面测试用于通过未知条件把产品崩溃。

表4-5给出了锁和钥匙例子的一些负面测试用例。

表4-5 负面测试用例

序号	输入1	输入2	当前状态	预期状态
1	某个其他锁的钥匙	顺时针转动	上锁	上锁
2	某个其他锁的钥匙	逆时针转动	上锁	上锁
3	铁丝	逆时针转动	上锁	上锁
4	用手头打上		上锁	上锁

在表4-5中，与我们已经看到的正面测试不同，这里没有需求标识。这是因为负面测试关注的是需求规格说明之外的测试条件。由于所有测试条件都在需求规格说明之外，因此不能按正面测试条件和负面测试条件分类。有人认为所有这样的条件都是负面测试条件，从技术角度看这种看法是正确的。

正面测试和负面测试的差别在于它们的覆盖率计算方法。对于正面测试，如果覆盖了所有形成文档的需求和条件，那么其覆盖率就认为是百分之百。如果规格说明很清楚，那么可以得到清晰的覆盖率指标。形成对比的是，负面测试是没有穷尽的，百分之百的负面测试覆盖率是不现实的。负面测试需要测试人员具有高度的创造性来覆盖尽可能多的“未知”条件，以避免产品在客户场地出现故障。

4.4.3 边界值分析

上一节已经提到过，条件和边界是软件产品中的两个主要缺陷源。本节将详细讨论条件问题。大多数软件产品中的缺陷都与条件和边界有关。所谓条件，是指基于各种变量取值需要采取一定行动的情况。所谓边界，是指各种变量值的“极限”。

本节要研究边界值分析（BVA），这是能够有效捕获出现在边界处的缺陷的一种测试方法。边界值分析利用并扩展了缺陷更容易出现在边界处的概念。

为了说明在边界处出现错误的概念，这里举一个能够为客户提供大宗购买折扣的记账系统。

我们很多人都熟悉购物时的大宗购买折扣概念，例如买1件要价1.59美元，但是买3件只需付4美元。大宗购买已经成为购物者的获利原则。从销售商角度看，大宗卖出也是更合算的，因为需要支出较少的存储成本并保持更好的现金流。假设有一家出售各种商品的

商店，它为购买不同数量商品的客户提出不同的价格，也就是说按购买量的不同"分段"计价。

购买数目	单价（美元）
买10件（即从第1件到第10件）	5.00
第二个10件（即从第11件到第20件）	4.75
第三个10件（即从第21件到第30件）	4.50
超过30件	4.00

从上表可以清楚地看出，买5件需要支付5×5=25美元。如果买11件，第一个10件需要支付10×5=50美元，第11件需要支付4.75美元。类似地，如果买15件，需要支付10×5+5×4.75=73.75美元。

从测试角度看，对于这个例子什么类型的数据最有可能暴露出程序的更多缺陷呢？人们通过总结发现，大多数缺陷出现在边界数据附近，例如购买9、10、11、19、20、21、29、30、31件和类似数量的商品时。虽然出现这种现象的原因还没有完全弄清楚，不过还是可以给出以下一些可能的原因：

- 程序员使用合适比较操作符的习惯，例如在进行比较时使用<=操作符，还是<操作符。
- 由于实现循环和条件检查有多种方式而产生的困惑，例如，在像C这样的程序设计语言中，有for循环、while循环和repeat循环。这些循环有不同的终止条件，在确定要使用的操作符时会产生一定程度的困惑，因此会在边界条件附近引入缺陷。
- 可能没有清楚地理解需求本身，特别是对边界附近需求的理解，因此使得即使是正确编码的程序也不能进行正确地处理。

表4-6给出在上面的例子中，应该执行的测试和预期的变量输出值（所购商品的总价）。表4-6只包含了正面测试用例，没有包括诸如非数字输入的负面测试用例。画圈的行是边界值，这些行与其他部分相比更有可能发现缺陷。

表4-6　大宗购买折扣例子的边界值

要测试的输入值	选择测试的理由	预期输出（美元）
1	第一个计价段的开始	5.00
5	第一个计价段中的值，没有考虑边界	25.00
9	正好低于第一个计价段，或正好在第一个计价段的末尾	45.00
10	第二个计价段的极限	50.00
11	正好高于第一个计价段，正好进入第二个计价段	54.75
16	第二个计价段中的值，没有考虑边界	78.50
19	正好低于第三个计价段，或正好在第二个计价段的末尾	92.75
20	第二个计价段的极限	97.50
21	正好高于第二个计价段，正好进入第三个计价段	102.00
27	第三个计价段中的值，没有考虑边界	129.00
29	正好低于第四个计价段，或正好在第三个计价段的末尾	138.00
30	第四个计价段的极限	142.50
31	正好高于第四个计价段	146.50
50	高出第四个计价段极限很多	182.50

边界值测试对于发现缺陷极为重要的另一个例子是特定资源、变量或数据结构的内部极限值。举一个在具备内存区缓存最近使用过的数据块的数据库管理系统（或文件系统）的例子。通常这种缓存区受到用户在启动系统时指定的参数限制。假设数据库在启动时指定缓存最近50个数据库缓存区，那么当这些缓存区满而第51个缓存区需要缓存时，第一个缓存区也就是最先使用的缓存区，就需要转存到次级存储器上。可以看出，类似插入新的缓存区和释放第一个缓存区，都出现在“边界”上。

> 如果输入（或输出）数据有可以明确区分的边界或范围，那么边界值分析对于生成测试用例是非常有用的。

如图4-2所示，有以下四种情况需要测试。第一，当缓存区完全空着（看起来像一个在逻辑上矛盾的句子），第二，当插入缓存区，缓存区还有空余时，第三，当插入最后一块缓存区时，第四，当缓存区已满再插入缓存区时。后两种测试比前两种更有可能发现缺陷。

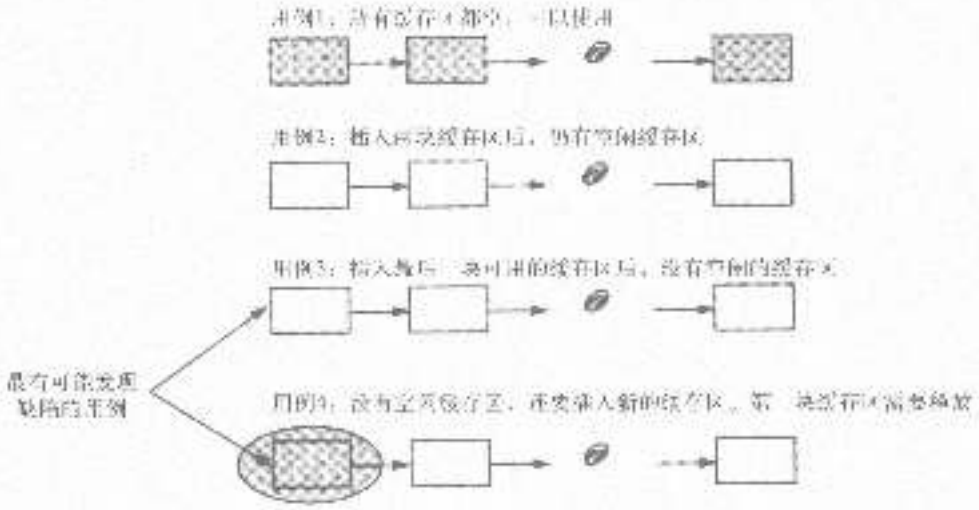


图4-2 缓存区管理的各种测试用例

总结一下边界值测试：

- 检查数据值对计算有影响的级差或不连续点，不连续点就是边界值，需要彻底测试。
- 检查内部极限，例如资源极限（如上面给出的缓存例子），产品处于这类极限的行为也应该是边界值测试的内容。
- 包含在边界值测试内容中的还有在文档中已说明的对硬件资源的限制。例如，如果文档说明产品将运行在4MB以上的RAM，那么一定要设计测试最低RAM（这里是4MB）的测试用例。
- 前面给出的例子讨论的都是对于输入数据的边界条件，对于输出值也要进行同样的边界值分析。

针对黑盒测试讨论的边界值分析也适用于白盒测试。像数组、堆栈和队列这样的内部数据结构也需要检查边界或极限条件。如果内部使用了链表结构，那么就应该彻底测试链表开始和结尾的行为。

边界值和决策表有助于确定最有可能发现缺陷的测试用例。这些概念的汇总就是本章中下面要讨论的等价类概念。

4.4.4 决策表

为了更好地说明测试条件（和决策表）的使用，来看一个计算收入纳税标准减免的简单程序。这个例子只是用来说明决策表的使用，不能用作任何国家的纳税建议或实际的税场景。

大多数纳税人都可以选择采用标准减免的方法或者采用逐项计算减免的方法。标准减免就是扣除要纳税的收入量。这种方法的好处是很多纳税人不必再逐项计算实际应该减免的税额，例如医药支出、慈善捐赠和税款。对于65岁以上的老人或盲人，标准减免更高。如果可以做出选择，就应该选择能够降低税额的计算方法。

1. 决定标准减免的第一个因素是申报状态。针对不同申报状态的基本标准减免是：

单身	4 750美元
已婚，联合申报	9 500美元
已婚，单独申报	7 000美元

2. 如果一对已婚配偶单独申报且一人不选择标准减免，那么另一人也不能选择标准减免。

3. 如果申报人是65岁或更年长，或其配偶是65岁或更年长（后一种情况适用于“已婚”且“联合”申报的情况），那么可以额外减免1000美元。

4. 如果申报人是盲人，或其配偶是盲人（后一种情况适用于“已婚”且“联合”申报的情况），那么可以额外减免1000美元。

从以上描述可以看出，标准减免的计算取决于三个因素：

1. 申报人的申报状态
2. 申报人的年龄
3. 申报人是否盲人

此外，在一定条件下以下因素在计算标准减免时也有影响：

1. 配偶是否采用标准减免方法
2. 配偶是否盲人
3. 配偶的年龄是否超过65岁

决策表列出各种决策变量、每个决策变量的条件（取值）以及每种条件组合要采取的行动。影响决策的变量作为决策表中的各个列，决策表的最后一列是决策变量取值组合对应的行动。如果决策变量很多（比如5、6个以上），变量的不同取值组合数较少（比如4、5个），决策变量在决策表中也可以作为各行，而不是各列。标准减免计算对应的决策表如表4-7所示。

表4-7　用于计算标准减免的决策表

状　态	配偶状态	年龄>65?	配偶年龄	是盲人?	配偶是盲人?	标准减免额（美元）
单身	—	否	—	否	—	4 750
单身	—	否	—	是	—	5 750
单身		是	—	否	—	5 750
单身	—	是	—	是	—	6 750
已婚，单独申报	标准减免	否	—	否	—	7 000
已婚，单独申报	标准减免	否		是		8 000

（续）

状　态	配偶状态	年龄>65?	配偶年龄	是盲人?	配偶是盲人?	标准减免额（美元）
已婚，单独申报	标准减免	是	—	否	—	8 000
已婚，单独申报	标准减免	是	—	是	—	9 000
已婚，单独申报	不采用标准减免	—	—	—	—	0
已婚，联合申报	—	否	否	否	否	9 500
已婚，联合申报	—	是	—	否	否	10 500
已婚，联合申报	—	是	—	是	—	11 500
已婚，联合申报	—	—	是	否	否	10 500
已婚，联合申报	—	—	是		是	11 500

注：严格说来，决策表各列排列次序并非必要。例如，第二列的问题应该是"配偶选择标准减免吗？"为了清晰起见，这里采用"配偶状态"作为标题。

读者会注意到，表4-7中有一些项给出"—"。在这些情况下，对应决策变量的取值不影响决策输出。例如，配偶状态只有当申报状态是"已婚，单独申报"时才有意义。类似地，配偶的年龄及其是否盲人，只有在申报状态是"已婚，联合申报"时才有意义。这种取值称为无所谓（有时用希腊字母φ表示）。这些"无所谓"显著降低了要设计的测试用例数。如果没有这些"—"，对于"单身"要设计8个测试用例：4个针对配偶选择标准减免，4个针对配偶不选择标准减免。除了这一差别，对于标准减免量的预期结果没有实际影响。这里把不允许使用"无所谓"，必须明确地给出所有情况的情形留作练习，供读者思考。有一些形式化的工具，例如卡诺（Karnaugh）图，可以用来导出表示决策表各种布尔条件的最小布尔表达式。本章最后的参考文献讨论了这些工具和技术。

因此，决策表是设计黑盒测试，检查产品在输入变量各种逻辑条件的行为的宝贵工具。形成决策表的步骤如下：

1. 确定决策变量。
2. 确定每个决策变量的可能取值。
3. 枚举每个变量允许值的组合。
4. 确定变量（或一组变量）对于其他输入变量组合没有的取值，并用"无所谓"符号表示。
5. 对于每个决策变量取值组合（通过"无所谓"场景进行恰当的最小化），列出行动或预期结果。
6. 形成表格，在除最后一列以外所有列内给出决策变量，在最后一列给出各行变量组合对应的行动项（包括在合适的地方填入"无所谓"）。

一旦形成决策表，表中的每一行就是一个测试用例的规格说明。确定决策变量使这些测试用例虽不是穷尽的也是有很大覆盖面的。通过使用"无所谓"对决策表进行剪裁，最小化测试用例的数量。因此，对于取决于决策变量取值的场景，决策表对于有效地编写测试用例非常有用。

如果输入和输出数据可以表示为布尔条件（TRUE，FALSE和无所谓），则决策表是非常有用的。

4.4.5 等价划分

等价划分是一种软件测试技术，用于确定少量能够产生尽可能多的不同输出条件的有代表性的输入值。这种方法可以减少用于测试的输入、输出值的排列组合，从而提高覆盖率，降低测试工作量。

产生同一个预期输出的一组输入值叫做一个划分。如果软件的行为对于一组值来说是相同的，那么这组值就叫做等价类或划分。在这种情况下，可从每个划分中选取一个有代表性的样本（又叫做等价类成员）进行测试。从一个划分中取一个样本对于测试已经足够。因为从划分中再取更多的值得到的测试结果会是相同的，不会产生额外的缺陷。由于所有值都产生相等和同样的输出，因此这些值叫做等价划分。

采用这种技术进行测试包括两个步骤：(a) 针对产品的输入和输出取值的完备集，确定所有划分，(b) 从每个划分中取一个成员进行测试，以最大化覆盖率。

通过等价类或划分的一个成员得到的测试结果，这种技术推断该划分所有取值的预期结果。使用这种技术的优点是采用少量的测试用例就能实现不错的覆盖率。例如，如果某个划分中的值有错误，那么可以外推到那个特定划分的所有取值。使用这种技术，通过不重复同一个划分中的相同测试，可以最大限度地降低测试的冗余。

下面举一个例子，某个保险公司根据年龄段实行以下保险费率。

寿险保险费率

某个寿险公司对所有年龄实行0.50美元的基本保险费。根据年龄段的不同，每月还要按下表支付额外的保险费。例如，一个34岁的人要支付的保险费=基本保险费+额外保险费=0.50+1.65=2.15美元。

年龄段	额外保险费（美元）
35岁以下	1.65
35－59	2.87
60岁以上	6.00

根据等价划分技术，可以给出基于年龄的等价划分：

- 低于35岁（有效输入）
- 35－59岁（有效输入）
- 60岁以上（有效输入）
- 负年龄（无效输入）
- 0岁（有效输入）
- 任何三位数字的年龄（有效输入）

从以上每个划分中选择一个代表值。读者可能已经注意到，即使只有很少的有效值，仍然应给出无效值输入样本，必须给出无效值输入样本，以免这些输入引起没有预见到的错误。可以看出，这个例子包含了正面和负面测试输入值。

表4-8给出了这个例子基于等价划分的测试用例。这个等价划分表有以下几列：

表4-8　寿险保险费例子对应的等价类

序号	等价划分	输入类型	测试数据	期望结果
1	低于35岁	有效	26，12	月保险费=（0.50+1.65）=2.15美元
2	35－59岁	有效	37	月保险费=（0.50+2.87）=3.37美元
3	大于60岁	有效	65，90	月保险费=（0.50+6.0）=6.50美元
4	负年龄	无效	-23	警告信息——无效输入
5	0岁	无效	0	警告信息——无效输入

- 划分定义
- 输入类型（有效/无效）
- 该划分对应的有代表性测试数据
- 预期结果

表4-8的每一行都是一个要执行的测试用例。例如，当一个人的年龄是48时，就使用行号为2的测试用例，预期结果是3.37美元。类似地，如果给出一个负数年龄，则应显示一条错误信息，告诉用户此输入是无效的。

上面这个例子导出的等价类使用的是范围值。还有一些方法可以确定等价类。例如，对于实数集合，一种划分的方法是：

1. 质数
2. 和数
3. 带小数点的数

这三类把实数集合划分为三个有效类。此外，考虑到用户可能给出的所有输入，必须增加一个无效类，即字母数字串。与上一个例子一样，表4-9也给出这个例子的等价划分。

表4-9 实数集合对应的等价划分样本

序号	等价划分	输入类型	测试数据
1	质数	有效	7, 29
2	和数	有效	444
3	带小数点的数	有效	78.67, 85.91
4	字母数字	无效	ABC23RTF

因此，与前一个保险费例子一样，这里也把潜在无限输入数据空间压缩为有限空间，又没有损失测试的有效性。这就是使用等价类的优点：选择能够真正代表整个输入空间的很小一组输入值并发现更多的缺陷。

准备等价划分表的步骤如下：

- 选择等价划分的判断准则（范围、取值表等）
- 根据以上判断准则确定有效等价类（允许取值的个数范围等）
- 从划分中选择一个样本数据
- 根据给定需求编写预期结果
- 确定可能有的特殊取值，并加到表中
- 检查是否为所有的测试用例给出了预期结果
- 如果对任何具体的测试用例不能给出明确的预期结果，可将其标出来，并进行更正。如果不能回答某个问题，或发现一个不合适的答案，可以考虑是否把问题记录下来并与大家一起沿着模糊的需求。

如果输入数据可以分为不同的集合，且产品对于集合的每个成员的行为或输出相同，那么等价划分对于最大限度地降低测试用例数量是非常有用的。

4.4.6 基于状态或基于图的测试

基于状态或图的测试对于以下情况非常有用：

1. 被测产品是一个语言处理器（例如编译器），语言的句法自动构成一个状态机，或一种由状态图表示的与上下文无关的语法。

2. 工作流建模。根据当前状态和合适的输入变量组合，执行具体的工作流，产生新的输出和新的状态。

3. 数据流建模。系统建模为一组数据流，从一个状态转换到另一个状态。

以上的2和3有些类似。以下针对1给出一个例子。再针对2给出一个例子。

考虑一个要求根据以下简单规则确认数字有效的应用程序：

1. 数字可以由一个可选符号开始。
2. 该可选符号可以后接任何位数的数字。
3. 这些数字可以有选择地后接用英文句号表示的小数点。
4. 如果有一个小数点，则小数点后应该有两位数字。
5. 任何数字，不管是否有小数点，都应该以空格结束。

以上规则可以用如图4-3所示的状态转换图表示。

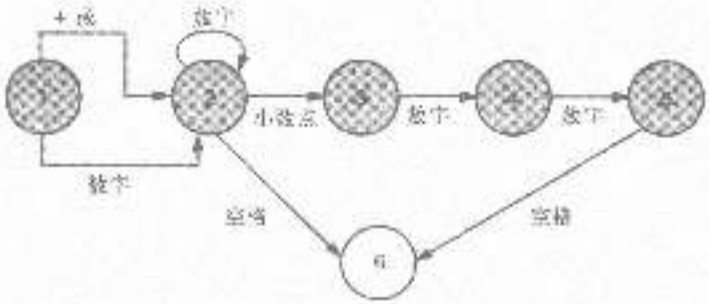


图4-3　状态转换图的例子

这个状态转换图可以转换为如表4-10所示的状态转换表，列出当前状态、当前状态允许的输入和对每个输入的下一个状态。

表4-10　图4-3对应的状态转换表

当前状态	输入	下一个状态
1	数字	2
1	+	2
1	−	2
2	数字	2
2	空格	6
2	小数点	3
3	数字	4
4	数字	5
5	空格	6

以上状态转换表可以用来导出测试用例，测试有效和无效数字。有效测试用例可以通过以下方法生成：

1. 从起始状态开始（本例是状态1）。
2. 选择一条到下一个状态的路径（例如+/-/数字，从状态1到状态2）。
3. 如果在给定状态遇到无效输入（例如在状态2遇到一个字符），则产生一个错误条件测试用例。
4. 重复以上过程，直到达到最后状态（本例就是状态6）。

基于图的测试方法适用于为诸如语言翻译器、工作流、事务流和数据流选择的状态机生成测试用例。

有关语言处理器使用基于状态的测试方法可归纳如下：

1. 确定场景的语法。上面的例子把状态图转换为状态机。在有些情况下，场景可以是与上下文无关的语法，可能需要用更复杂的"状态图"表示。
2. 对于每个有效状态-输入组合设计测试用例。
3. 对于最常见的无效状态-输入组合设计测试用例。

基于图的测试第二个适用情况是事务流或工作流。考虑一个员工请假申请的简单程序。典型的员工请假申请过程可以用以下步骤表示：

1. 员工填写请假申请，给出自己的员工标识号，以及希望离开的起止日期。

2. 自动系统确认该员工有资格在这段时间内离开。如果未通过确认，则拒绝该申请；如果有资格离开，则控制流会转到下一个步骤。

3. 此信息传给该员工的经理，由这个经理确认该员工在那段时间离开没有问题（例如在要求请假的那段时间没有特别重要的任务）。

4. 根据请假的可行性，经理对请假申请作出最终批准或拒绝。

以上事务流也可以使用简单的状态图给出，如图4-4所示。

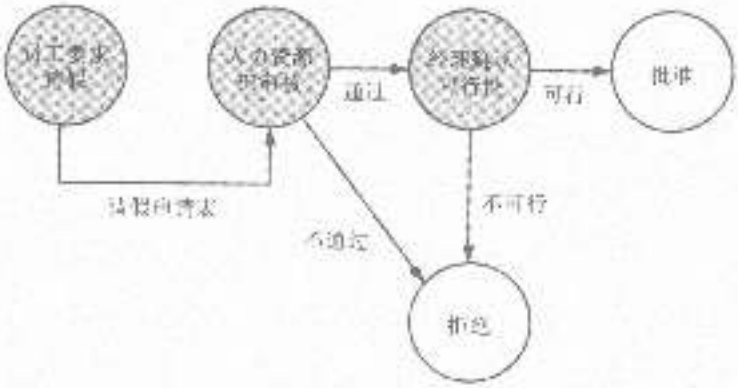


图4-4 表示工作流的状态图

在以上例子中，每个用圆圈表示的状态都是一个事件，也就是决策点。而状态之间的箭头或连线表示数据输入。与上一个例子一样，也可以从起始状态开始按照状态图走通各个状态迁移，直到达到"最终"状态（用没有阴影的圆圈表示）。

像以上这样的状态图还可以利用上个例子介绍的同样的标记和方法转换为状态转换表。像本例这样的基于图的测试适用于：

1. 应用程序可以用一组状态刻画。

2. 已经很好理解引发从一个状态到另一个状态迁移的数据值（屏幕、鼠标点击等）。

3. 已经很好理解每个状态内部对所接收数据的处理方法。

4.4.7 兼容性测试

前面几节介绍了几种测试产品功能和需求的技术。还提到了测试用例执行结果要与预期结果进行比较，以确定该测试用例是否通过。测试用例执行结果不仅取决于产品的功能是否合适，还同样取决于提供功能的基础设施。如果基础设施的参数改变，期望产品的行为仍然正确，并产生所要求或期望的结果。基础设施参数可以是硬件、软件或其他组件的参数。这些参数对于不同的客户是不同的，黑盒测试不知道这些参数对测试用例执行结果的作用，可能不一定是完备的、有效的，因此可能没有真正反映产品在客户场地的行为。所以需要进行兼容性测试。兼容性测试确保产品能够在不同的基础设施组件上正常运行。以下介绍兼容性测试所使用的技术。

确保产品功能对于不同基础设施组件能够一致地发挥作用的测试叫做兼容性测试。

通常会影响产品兼容性的参数有：

- 计算机中处理器（CPU）的类型（Pentium III、Pentium IV、Xeon、SPARC等）和数量
- 计算机的体系结构和特性（32位、64位等）
- 计算机上的可用资源（RAM、磁盘空间、网卡）
- 产品预期会适用的设备（打印机、调制解调器、路由器等）
- 操作系统（Windows、Linux等及其变种）和操作系统服务（DNS、NIS、FTP等）
- 中间件基础设施组件，例如Web服务器、应用服务器、网络服务器
- 后台组件，例如数据库服务器（Oracle、Sybase等）
- 要求特殊硬件软件解决方案的服务（聚类计算机、负载平衡、RAID阵列等）
- 用于生成二进制代码的软件（编译器、连接器等及其合适的变种）
- 用于生成组件的各种技术组件（SDK、JDK等及其合适的不同版本）

以上只是一部分参数，还有很多会影响产品特性行为的其他参数。在上面的例子中，我们描述了10种参数，如果每一个参数都有4个取值，那么就有40个不同的取值要测试。但是还不止这些，不仅是单个参数取值会影响产品特性，这些参数的排列组合也会影响产品特性。如果考虑这些组合，测试特定功能的次数会达到数以千计甚至百万计。以上假设有10个参数，每个参数有4个取值，那么要测试的组合数就是4^{10}，这个数太大了，不可能进行穷尽测试。

为了得到可行的测试参数组合，可创建一种兼容性矩阵。兼容性矩阵的列表示要测试的不同参数组合，行表示具体参数取值集合的唯一组合。表4-11给出了一个邮件应用程序的示例兼容性矩阵。

表4-11　邮件应用程序的兼容性矩阵

服务器	应用服务器	Web服务器	客户端	浏览器	MS Office	邮件服务器
Windows 2000 Advanced Server with SP4 Microsoft SQL Server 2000 with SP3a	Windows 2000 Advanced Server with SP4 and .Net framework 1.1	IIS 5.0	Win 2K Professional and Win 2K Terminal Server	IE 6.0 and IE 5.5 SP2	Office 2K and Office XP	Exchange 5.5、2K
Windows 2000 Advanced Server with SP4 Microsoft SQL Server 2000 with SP3a	Windows 2000 Advanced Server with SP4 and .Net framework 1.1	IIS 5.0	Win 2K Professional and Win 2K Terminal Server	Netscape 7.0、7.1、Safari and Mozilla	Office 2K and Office XP	Exchange 5.5 and 2K
Windows 2003 Enterprise Server Microsoft SQL Server 2000 with SP3a	Windows 2003 Enterprise Server and .Net framework 1.1	IIS 6.0	Win XP Home and Win XP Professional	IE 6.0 and IE 5.5 SP2	Office XP and Office 2003	Exchange 2K
Windows 2003 Standard Server Microsoft SQL Server 2000 with SP3a	Windows 2003 Standard Server and .Net framework 1.1	IIS 6.0	Win XP Professional and Citrix	IE 6.0、IE 5.5 SP2 and Mozilla	Office XP and Office 2003	Exchange 2003

（续）

服务器	应用服务器	Web服务器	客户端	浏览器	MS Office	邮件服务器
Windows 2003 Enterprise Server Microsoft SQL Server 2000 with SP3a	Windows 2003 Enterprise Server and .Net framework 1.1	IIS 6.0;	Win XPProfessional and Win 2003 Terminal	IE 6.0 IE 5.5 SP2 and Safari	Office XP and Office 2003	Exchange 2K

表4-11只是一个例子，并没有覆盖所有参数及其组合。关于执行兼容性测试使用兼容表的适用常用技术有：

1. 横向组合 产品同时存在的所有执行测试用例集的参数值，在兼容矩阵中组成一行。可以同时存在的参数取值一般属于不同的基础设施层次或类型，例如操作系统、Web服务器等。为每行设置计算机或环境，并逐一使用这些环境对产品的功能集合进行测试。

2. 智能采样 在横向组合方法中，必须用兼容性矩阵中的每一行测试产品的每个特性。这需要大量的时间和工作量。为了解决这个问题，基础设施参数的组合智能地与特性集合组合在一起进行测试。如果由于任何组合引起问题，那么就执行测试用例，检查各种排列组合。选择智能样本的基础是根据产品与参数的依赖关系集合得到的信息。如果产品对参数集合的依赖较弱，就从智能样本表中将其删除，合并并测试所有其他参数。这种方法能够显著降低测试用例的排列组合数量。

兼容性测试不仅与产品外部的参数有关，还与产品内部的参数有关。例如，给定数据库的两个版本可能依赖同一个数据库的一部分API集合。这些参数是兼容性矩阵的附加部分，也要进行测试。这类涉及产品本身部件的产品兼容性测试可以进一步分为两类：

1. 反向兼容性测试 客户会有同一个产品的很多不同版本，对于客户非常重要的是，采用产品老版创建的对象、对象属性、模式、规则和报告等，对于同一个产品的当前版是否仍然有效。保证产品的当前版对于同一个产品的老版仍然有效的测试叫作反向兼容性测试。反向兼容性测试所需的产品参数要补充到兼容性矩阵中，并进行测试。

2. 正向兼容性测试 有时产品要与该产品和其他基础设施组件的新版本一起运行，这时必须适应未来的需求。例如，IP网络协议第6版使用128位的编址模式（IP第4版只使用32位）。可以定义数据结构适应128位地址，并且采用还没有成为完全实现的产品的IPv6协议栈的原型实现进行测试。IPv6的部分特性对于最终用户还不能使用，但是这种针对未来的实现和测试有助于避免以后的大量变更。这种需求要作为正向兼容性测试的一部分实施。采用操作系统测试版对产品进行测试，较早使用开发人员的软件包等，都是正向兼容性测试的例子。这类测试可保证最大限度地降低产品在满足未来需求方面的风险。

实施兼容性测试和使用前面介绍的技术可能不需要对产品的内部知识有深入了解。兼容性测试要在对产品在基本环境上确认之后进行。兼容性测试需要大量的工作，因为有大量的参数组合。依照前面介绍的技术有助于更有效地实施兼容性测试。

4.4.8 用户文档测试

用户文档包括所有手册、用户指南、安装指南、设置指南、产品说明、软件发布说明以及与软件产品一起提供意在帮助最终用户了解软件系统的在线帮助。

用户文档测试有两个目的：

1. 检查文档所描述的是否在产品中提供。

2. 检查产品中有的是否在文档中作了正确解释。

当产品升级时，对应的产品文档必要时也应该升级，以反映会对用户产生影响的变更。但是，这种情况不一定总发生。一种原因是文档编写组和测试/开发组缺少充分协同。经过一段时间，产品的文档会偏离产品的实际行为。用户文档测试关注确保文档的内容是否确切地与产品行为一致。坐在计算机前对屏幕、事务和报表进行逐一验证。此外，用户文档测试还要检查文档中的诸如拼写和语法错误等语言使用问题。

用户文档测试要保证文档与产品的相互一致。

测试这些文档是很重要的，因为用户开始在自己的场地使用软件时会参考这些手册、安装和设置指南。用户常常不了解软件，需要帮一把，自到感到使用自如。由于这些文档是用户接触软件的第一种交互，他们会产生第一印象。编写不好的安装文档会失掉用户，使他们对产品产生偏见，尽管产品可提供丰富的功能。

实施用户文档测试的好处包括：

1. 用户文档测试有助于发现评审时忽略的问题。

2. 高质量的用户文档可保证文档与产品的一致性，因此，可以最大限度地降低客户报告缺陷的可能。好的用户文档还可以缩短每次电话支持所需的时间，有时处理帮助电话的最好方式就是告诉客户参考手册的相关章节，这样可以降低品的支持费用。

3. 使技术支持工作变得更容易。当用户忠诚地遵循文档给出的指示却不能得到所需要（或所承诺）的结果时，用户会感到灰心，并把这些情绪发泄给技术支持人员。保证每份文档都是有效的，正确的，可以使客户感到更满意，对技术支持人员的态度会更好。

4. 新加入项目组的程序员和测试人员可以使用文档了解产品的外部功能。

5. 如果文档的质量很高并且与产品一致，客户需要的培训更少，可以更快地进入高级培训和产品使用。这种高质量的用户文档可以为用户组织降低总的培训成本。

与一般软件缺陷一样，在用户文档中发现的缺陷也要跟踪到最后。为了使文档编写者能够提供完整的缺陷信息，应该向其提供缺陷描述、页码和段落号、文档版本号、审查人姓名、编写者姓名、审查人的联系电话、优先级和严重等级等文档缺陷信息。

由于好的用户文档有助于减少投诉电话，可减少组织的支出经费。在用户文档上投入的人力物力对于组织来说是一项很有价值的长远投资。

4.4.9　领域测试

白盒测试要求检查程序代码，黑盒测试执行测试时不需要检查程序代码，而要检查规格说明。领域测试可以看作是下一层测试，甚至不检查软件产品的规格说明，纯粹根据应用领域专门知识和经验测试产品。这种测试方法要求对软件所针对的日常业务活动有很好的理解，要求业务领域知识而不是软件规格说明包含了什么、或软件是怎样编写的。因此，领域测试可以看作是黑盒测试的扩展。当我们从白盒测试转到黑盒测试，再转到领域测试时（如图4-5所示），我们对于软件产品的细节了解得越来越少，越来越多地关注软件产品的外部行为。

实施领域测试的测试工程师要有关于业务领域的深入知识。由于深入的业务领域知识是领域测试的前提，因此从业务领域（例如银行、保险等）聘用测试人员并对其进行软件方面的培训，而不是让软件专业人员接受业务领域知识培训，有时会更容易一些。这可以减少测

试人员在领域知识方面的培训工作量和时间，还可以提高领域测试的效率。

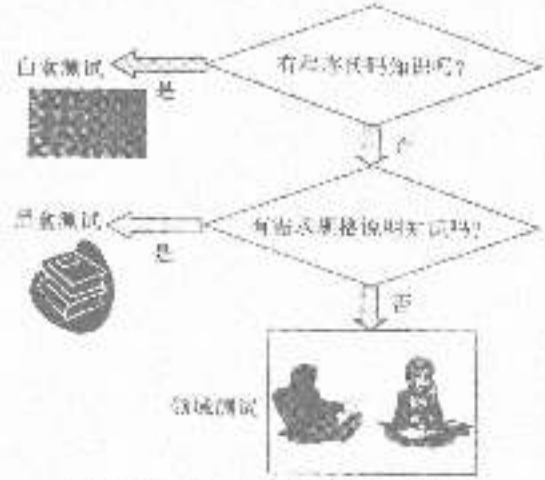


图4-5 白盒、黑盒和领域测试的内容

例如，对于银行软件，了解银行的账户开设过程能使测试人员更好地测试功能。在这种情况下，负责账户开设的银行人员了解开设账户的人员特性、面临的一般问题和实际解决方案。进一步地说，银行人员可能遇到过开设账户的人不能使用所需的支持文档或不能正确填写客户申请表的情况。在这种情况下，银行人员可能要采取不同行动开设账户。虽然这些情况的大多数会明确地在业务需求中描述，但是有时银行人员在测试中会观察到需求规格说明中没有明确描述的问题。因此，当银行人员测试软件时，所设计的测试用例往往更全面、更实际。

领域测试要求设计并执行与购买和使用该软件的人有关的测试用例，这有助于理解他们试图解决的问题，以及他们使用软件解决这些问题的方式。领域测试是否成功还取决于测试工程师个人对系统运行和系统应该支持的业务过程的理解程度。如果测试人员不理解系统或业务过程，那么他们使用软件以及通过测试脚本和测试用例测试软件，都是很困难的。

领域测试要测试的是产品，而不是全面检查产品的逻辑，业务流程而不是被测软件决定测试步骤。领域测试又叫做“业务纵向测试”。编写测试用例的依据是软件用户的日常工作。

> 领域测试是测试人员运用自己的领域知识测试产品对于用户日常工作适合性的测试。

下面再通过在ATM机上提取现金功能的例子来进一步理解领域测试，这个例子扩展了前面银行软件的例子。用户执行以下步骤：

步骤1：来到ATM前面。

步骤2：把卡插入ATM。

步骤3：输入正确的个人标识号。

步骤4：选择提取现金。

步骤5：输入金额。

步骤6：拿钱。

步骤7：退出并取卡。

在上面的例子中，领域测试人员关注的并不是测试设计中的所有细节，而是关注测试业务流程的所有细节。在上面的流程中有一些设计逻辑步骤不一定测试。例如，如果要进行其

他形式的黑盒测试，可能要测试是否使用了合适的币值名称。测试币值名称的典型黑盒测试方法是检查是否给出所需的名称，而且所要求的金额是否能够分解到已有的币值。而在作为最终用户进行领域测试时，所关心的只是能否得到所要数额的现金。（毕竟没有ATM能提供选择币值的便利。）在编写领域测试的测试用例时，会发现规格说明中省略了这些中间步骤。省略步骤并不意味着这些步骤不重要。这些"省略步骤"（例如检查币值）应该在启动领域测试之前进行。

黑盒测试指导测试对于具体测试类型或阶段更有意义的区域。像等价划分和决策表这样的技术对于黑盒测试的早期阶段都是非常有用的。能够捕获一定类型的缺陷或测试条件。一般来说，领域测试要在所有组件都集成之后并在产品完成其他黑盒测试（例如等价划分和边界值分析）之后进行。因此，领域测试的关注点更多地集中在业务领域上，以保证软件具有领域所需要的功能。为了测试软件是否具有特定的"领域智能"，测试人员应具备业务流程方面的实际知识。这一点可以通过能够检查实际业务场景的更好、更有效的测试用例反映出来，可以满足领域测试的目标和目的。

4.5　小结

没有方法论指导的黑盒测试就像看地图时不知道自己在哪里，不知道目的地叫什么一样。

本章介绍了执行黑盒测试的多种技术。这些技术并不是相互排斥的。在实际产品测试中，应组合运用这些测试技术以最大限度地提高有效性。表4-12归纳了这些技术适用的场景。通过明智地组合使用这些不同的技术可降低下游其他测试，例如集成测试、系统测试等的总成本。

表4-12　黑盒测试技术的适用场景

要测试的场景	如有效使用黑盒测试技术
输出值可以根据输入变量取值由一定的条件算式	决策表
输入值可以分等级，每个取值范围对应一种确定的功能	边界值分析
输入值可以划分为类（例如范围、取值表等），每个类对应一种特定的功能	等价划分
检查错误和未期望的输入值	正面和负面测试
工作流、过程流或状态转换图	状态转换测试
保证需求被完全测试和合理满足	基于需求的测试
测试领域问题而不是产品规格说明	领域测试
保证文档与产品一致	文档测试

问题与练习

1. 请考虑本章介绍的锁和钥匙例子。我们曾假设锁只有一把钥匙。假设锁需要两把钥匙按一定顺序插入。请修改表4-1给出的样本需求，把这个新条件包含进去。相应地重建一个类似表4-2的跟踪矩阵。
2. 对于以下每种情况，确定可用来测试以下需求的最适合的黑盒测试技术：
   a. "性别代码的有效值是'M'或'F'"。
   b. "员工有资格每年请假的天数，对于工作未满3年的是10天，满3年未满5年的是15天，满5年的是20天"。

c. "每份采购订单都必须先经过员工经理和采购部负责人的批准。此外，如果金额超过10 000美元，还应经过CFO的批准"。

d. "文件名应以字母开头，可以有最多30个字母数字，可以有一个英文句号后接最多10个字母数字"。

e. "到银行开户的储户可能没有进行生日的英文确认，在这种情况下，银行柜员必须能够判断，人工取消输入生日确认码的处理"。

3. 请考虑第3章问题4已经介绍过的从链表删除一个元素的例子。这个例子的边界值条件是什么？确定要在边界值上和附近进行测试的数据。

4. 一个基于Web的应用软件可以部署在以下环境中：

a. 操作系统（Windows和Linux）

b. Web服务器（IIS 5.0和Apache）

c. 数据库（Oracle和SQL Server）

d. 浏览器（IE 6.0和FireFox）

如果要穷尽测试所有配置组合，需要测试多少种配置？对上表进行剪裁采用的准则是什么？

5. 产品通常提供不同类型的文档，包括安装文档（用于产品安装）、管理文档、用户指南等。测试这些不同类型的文档需要什么技能？对于每种文档什么人最有资格进行测试？

6. 某个仓库系统中的产品代码输入值应通过一个产品主表提供。请确定用于这些需求测试的等价类集合。

7. 本书讨论了将输入空间划分为多个等价类的例子。请举出等价类可以通过划分输出类的方法获得的情况。

8. 请考虑第3章给出的日期确认例子。假设不能得到程序代码，用本章讨论的任何技术导出一组测试用例，并用以下表格表示设计结果。

测试数据	选择该数字的理由	可使用的黑盒测试技术	预期结果

9. 以下给出创建SQL数据库表的一个样本规则：

SQL语句应该以如下句法开头：

```
CREATE TABLE <表名>
```

后面应该接一个左括号、一个用逗号分隔的标识列表和一个右括号。每个列标识都应提供一个强制的列名、一个强制的列类型（应该是NUMBER、CHAR和DATE之一）和一个可选的列宽。此外，还有以下规则：

a. 每个关键字可以缩写为3个或更多字符

b. 列名在表中必须唯一

c. 表名不能重复

请根据以上需求画出状态图并导出初始测试用例。采用其他合适的技术检查以上的规则a至c。

第5章 集成测试

5.1 集成测试的定义

系统是由可以包括硬件和软件的多个组件或模块组成的。集成定义为组件之间的交互。测试模块之间以及与其他外部系统交互的集成叫做集成测试。集成测试在完成了两个产品组件后就可以开始进行，直到所有组件齐面都测试完毕结束。最后一轮包含所有组件的集成叫做最终集成测试（FIT），或系统集成。

集成测试既是一种测试类型也是一个测试阶段。因为集成定义为一组交互，因此组件之间的所有已定义的交互都需要测试。体系结构和设计可以提供系统内部的交互细节，但是测试一个系统与另一个系统之间的交互要求对这些系统一起工作的方式有深刻理解。这种集成知识（也就是关于系统或模块在一起工作的知识）取决于很多模块和系统。这种千差万别的模块在与其他系统集成时可能有不同的工作方式。这会给集成测试流程和要完成的工作带来复杂性。由于认识到这种复杂性，因此人们专门用一个测试阶段测试这些集成，逐渐产生集成测试过程。这个阶段叫做集成测试阶段。

> 集成测试既是一个阶段也是一种测试类型。

由于集成测试的目标是模块之间的交互，因此这种测试就像白盒、黑盒和其他类型的测试一样，也有一套以下将要介绍的技术和方法。因此，集成测试又看作是一种测试类型（所以放在本书的第二部分中）。

下一节将介绍作为一种测试类型的集成测试，下节之后将把集成作为一个测试阶段。

5.2 集成测试作为一种测试类型

集成测试意味着测试接口。在说到接口时，需要注意集成测试要测试两种接口，即内部接口和输出接口。

内部接口是指在项目或产品内部两块模块之间提供通信的接口，是产品内部的，不对客户或外部开发人员开放。输出接口是产品之外第三方开发人员和解决方案提供商可以看到的接口。

提供接口的一种方法是提供应用编程接口（API），API使一个模块能够调用另一个模块，发出调用的模块可以是内部的，也可以是外部的。例如，JDBC就是Java程序通过使用API实现特定SQL调用的例子。尽管API和接口很相似，但是重要的是要认识到，集成是要达到的目的，而API是达到这种目的的一种手段，只是提供两个模块之间接口的手段之一。读者会想到各种模块之间的其他集成手段。比如简单的函数调用、公共函数，还比如程序设计语言结构的一些特性，如全局变量，还比如操作系统结构，如标志和共享内存。本章不打算详细讨论集成载体（这主要是开发问题），而是要研究究竟该怎样测试接口（这是测试关注点）。

并不是所有接口都可以同时提供测试，因为通常不同的接口是由不同的开发团队开发的，每个团队都会有自己的进度安排。为了在不能提供所涉及的组件全部功能的情况下测试接口，可使用桩模块。桩模块通过以合适的格式提供合适的值，就像要集成的实际组件一样，模拟接口。

集成测试通过历遍内部和输出接口并测试软件功能的测试用例完成。内部接口是供组织内部其他开发人员使用的，输出接口是供第三方开发人员或组织外其他用户使用的。内部接口的测试需要完全理解体系结构和高层设计（HLD），及其对软件功能的影响。对于软件提供的输出接口，需要理解这些接口的用途，提供这些接口的理由，以及开发人员和解决方案集成提供商实际使用这些接口的方法。因此，设计、体系结构和用法知识是实施集成测试必需的。

最初的输出（或外部）接口是通过API和软件开发包（SDK）提供的。SDK的使用要求理解提供API/SDK的程序设计语言。以后，接口可以通过脚本语言提供，不需要SDK。（一些流行的脚本语言包括Perl、Tcl/Tk等）。这些脚本语言的使用者不必或不需要深入学习编写API所使用的语言，接口能够被与最初编写该接口不同的程序设计语言环境调用。这极大地简化了输出接口的使用，为了测试接口，可以通过点击鼠标，动态地创建脚本，并可以在运行时进行修改。

所有这些都促使接口广泛使用，接口的用途也越来越多。这些接口已经变得越来越通用，不再局限于特定的应用或语言。这导致越来越多地出现接口排列组合地使用的情况。因此，集成测试的复杂性，即测试接口各种使用场景的复杂性，也是要增加。

> 集成测试作为一种测试类型，关注测试"隐式和显式"、"内部和外部"的接口。

在讨论接口时，需要记住并不是模块之间的所有交互都是已知的，都是通过接口解释的。有些接口写入文档，有些则不写入，这就带来另一种接口分类，即隐式接口和显式接口。显式接口是写入文档的接口，隐式接口是软件工程师内部知道，但是没有写入文档的接口。测试（白盒和黑盒）既要检查显式接口也要检查隐式接口，测试所有的交互。

测试人员经常会想到的一个问题是，究竟集成测试是白盒测试还是黑盒测试。在大多数情况下，最恰当的回答是集成测试是一种黑盒测试。但是有时体系结构或设计文档没有解释清楚组件之间的所有接口，那么集成测试也会包括检查代码以生成额外的测试用例，并与通过黑盒测试方法生成的测试用例一起使用。这种方法可以叫做"灰盒测试"方法。

下面通过例子解释。图5-1给出了一组模块和与其相关的接口。根据对体系结构、设计或用法的理解，可得到实线表示的显式接口和虚线表示的隐式接口。

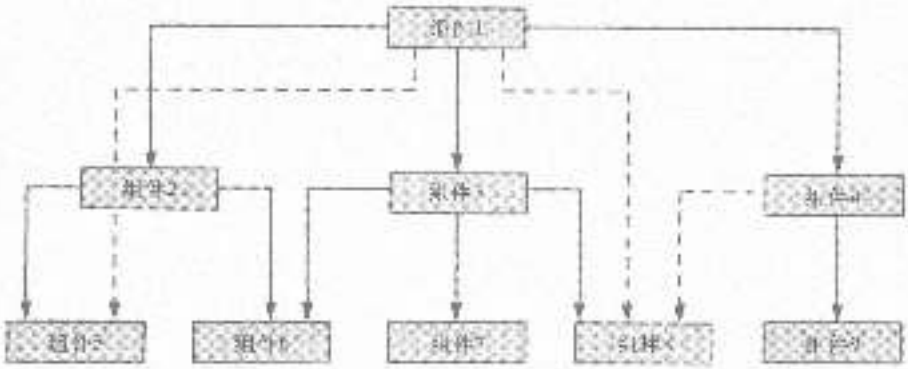


图5-1 一组模块和接口

从图5-1可以清楚地看出，模块之间至少有12个接口需要测试（9个显式接口和3个隐式接口）。下一个问题是需要以什么顺序测试这些接口。已有多种方法可用来确定集成测试的顺序，包括：

1. 自顶向下集成
2. 自底向上集成
3. 双向集成
4. 系统集成

5.2.1　自顶向下集成

集成测试首先测试最上层的组件与下层组件的接口，从上到下依次测试，直到覆盖所有组件。

为了更好地理解这种方法，假设某个新产品软件开发按图5-2给出的编号顺序逐个完成各个组件。集成测试首先测试组件1和组件2之间的接口，为了完成集成测试，需要一起测试图5-2中覆盖所有箭头的全部接口。接口测试的实施顺序如表5-1所示。

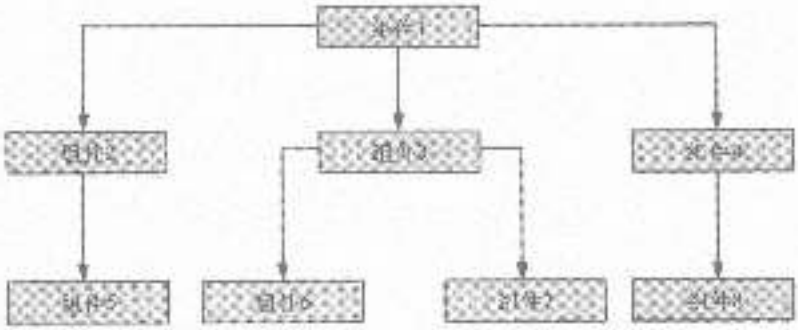


图5-2　自顶向下集成举例

在增量产品开发模型中，每个增量都要在产品中增加一两个组件，集成测试方法要求只对所增加的新接口和产品中受变更或增加影响的有关接口进行测试。因此，在这种情况下并不是以上所有集成测试步骤都要执行。例如，在图5-2中假设一个组件（组件8）是新增加到这个版本上的，所有其他组件都在前一个版本中测试过。如果这种增加会对组件5的功能产生影响，那么当前版本的集成测试只需要包含步骤4、7、8和9。

表5-1　图5-2所示例子的接口测试顺序

步骤	测试的接口
1	1-2
2	1-3
3	1-4
4	1-2-5
5	1-3-6
6	1-3-6-(3-7)
7	(1-2-5)-(1-3-6-(3-7))
8	1-4-8
9	(1-2-5)-(1-3-6-(3-7))-(1-4-8)

为了减少集成测试的步骤数，步骤6和步骤7可以合并为一个步骤执行，类似地，步骤8和9也可以合并为一个步骤执行。合并步骤并不意味着减少要测试的接口数量，只意味着可以缩短等待时间，因为不必等待步骤6和8完成就可以分别开始步骤7和9。

如果一组组件及其相关的接口不需要其他组件先完成就可以提供功能，或在软件产品中只有很少的接口需求，那么这组组件及其相关的接口叫做“子系统”。产品中的每个子系统不管是否有其他子系统存在都能够独立运行。这使得集成测试更容易一些，并使关注点集中到

所要求的接口上，而不必考虑组件的各个组合。在表5-1中，属于步骤4、6和8的组件可以认为是子系统。

以上介绍的自顶向下集成测试假设组件1提供了其他组件所需的所有接口，即使其他组件已经就绪，并且以后阶段（也就是说，其他组件开发完）也不需要修改。这种方法反映了瀑布或V字软件开发模型。

如果高层组件在每次增加下层组件后都需要修改，那么对于增加的每个组件，集成测试都要从步骤1开始重复。这可能是迭代式软件开发模型的一个要求，因此，不管使用什么软件开发模型，自顶向下的方法经过适当重复都可以在集成测试中使用。

如果使用不同的遍历方法，接口的测试顺序会产生一点变化，与表5-1给出的顺序略有不同。广度优先方法测试组件的顺序是1-2、1-3、1-4等，深度优先方法测试组件的顺序是1-2-5、1-3-6等。广度优先方法在表5-1得到使用，因为假设组件的完成顺序同组件的编号。但是遵循这种集成方法并不需要采用同样顺序。

5.2.2 自底向上集成

自底向上集成正好与自顶向下集成相反，即新产品开发的组件反过来就绪，从最下层开始。在图5-3中，组件的完成顺序与图中的编号相同。

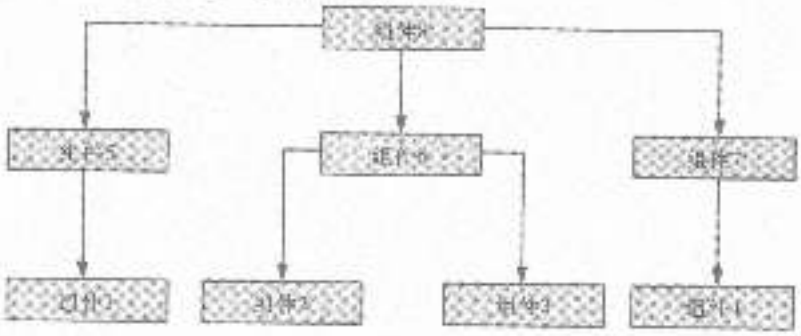


图5-3 广度向上集成举例。黑箭头方向表示逻辑流，黑箭头方向表示集成路径方向

注：图5-3给出箭头的两个方向同时表示了组件的逻辑流和集成方向。逻辑流从上到下，集成路径自底向上。

自底向上集成的遍历从图5-3中的组件1开始，覆盖所有的子系统，直到到达组件8。表5-2给出了接口测试的顺序。

表5-2 图5-3所示自底向上例子的接口测试顺序

步骤	测试的接口
1	1-5
2	2-6、3-6
3	2-6-(3-6)
4	4-7
5	1-5-8
6	2-6-(3-6)-8
7	4-7-8
8	(1-5-8)-(2-6-(3-6)-8)-(4-7-8)

通过合并步骤2和3、步骤5至8，表5-2所示集成步骤数可以优化到4步，如表5-2所示。参见彩图。

与前面在介绍自顶向下集成时提到的一样，对于增量式产品开发，只有受影响和增加的接口需要测试，覆盖所有子系统和系统组件。

图5-2和图5-3的一点不同是，后者的箭头是两个方向的。自顶向下的方向表示交互或控制流方向，自底向上的方向表示集成方法或集成路径方向。这意味着产品的逻辑流可以与集成路径的方向不同。在这个例子中，逻辑流表交

互自顶向下，而集成路径自底向上。这种方法使得同一个产品的集成路径可以组合。

迭代和敏捷模型是一个解释逻辑流和集成路径不同的例子。在这种模型中，产品开发组织需要定期向客户展示产品功能，以得到客户反馈。提高交付速度的一种办法就是使每个组件都独立。不同组件之间有一部分代码重复，以便向客户单独展示组件。重复的代码不仅是为了展示产品，也是为了独立测试。在上面的例子中，组件5～8作为带有重复的公用代码独立组件开发，经过客户的测试和认可之后，公用代码就会推到高层组件中，比方说组件5～7。经过这轮迭代，组件5～7仍然有重复的代码，会在下一轮迭代中通过将这些代码转移到组件8中而消除。这是一种进化模型，使产品经过多轮客户反馈和集成后最终完成。接口定义只在把代码转移到高层组件时才完成。公共代码在组件之间转移，只在转移时才定义接口。因此在这种模型中，集成测试的每轮迭代都从表5-2中的步骤1开始。

读者可能会说，自顶向下的集成方法最适合瀑布和V字模型，而自底向上方法适用于迭代和敏捷模型。从过程的观点看，这只对上面的例子是对的。在实践中，集成方法的选择更多取决于产品的设计和体系结构，取决于对应的优先级。此外，合适方法的选择需要考虑多种其他视角，例如组件的就绪情况、所使用的技术、过程、测试技能和可用的资源。

5.2.3　双向集成

双向集成组合了自顶向下和自底向上的集成方法，组合导出集成步骤。

如图5-4所示，假设软件组件的就绪顺序与图中的编号顺序相同。单独测试单个组件1、2、3、4和5。双向集成最初通过桩模块和驱动模块进行。驱动模块用于提供上游的连通性，而桩模块提供下游的连通性。驱动模块是一种函数，用于将请求转移到某个其他组件，而桩模块模拟尚未就绪的组件行为。当测试了这些被集成组件的功能之后，要丢弃这些驱动模块和桩模块。一旦组件6、7和8就绪，这种集成方法就会只关注这些组件，因为这些组件是新的和需要关注的。这种方法又叫做"三明治集成"。

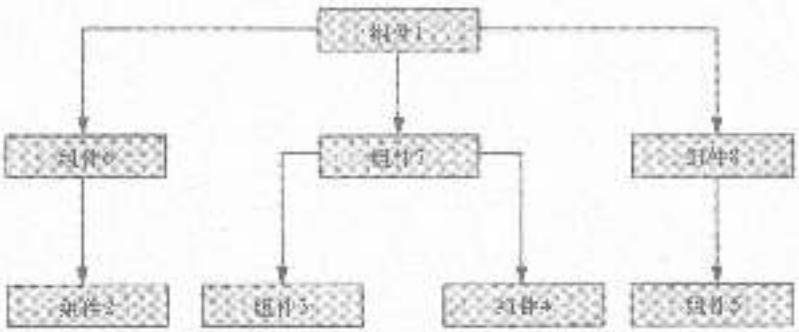


图5-4　双向集成

使用双向集成方法的操作步骤如表5-3所示，参见移图。

从表5-3可以看出，在这个例子中，步骤1～3采用了自底向上的集成方法，步骤4使用了自顶向下的集成方法。

这种方法适用于从两层到三层体系结构环境迁移的情况。在产品开发阶段，当需要从两层体

表5-3　采用三明治测试方法的集成步骤

步骤	测试的接口
1	6-2
2	7-3-4
3	8-5
4	(1-6-2)-(1-7-3-4)-(1-8-5)

系结构向三层体系结构迁移时，中间层（组件6～8）作为通过取自下层的应用和上层的服务代码的一组新组件创建。

5.2.4 系统集成

系统集成意味着将系统的所有组件集成起来作为一个单元。测试接口的集成测试可分为两个类：

- 组件或子系统集成
- 最终集成测试或系统集成

在研究以上各个集成方法论的步骤时，可以发现完整的系统集成也包含在最后一个步骤中。因此，系统集成实际上是以上介绍的各种方法论的一部分。

这种方法的突出特点是优化问题。这种方法不是逐个组件地集成和测试，而是等到所有组件就绪后进行一轮集成测试。这种方法又叫做大爆炸集成。这种方法可以减少测试工作量，并节省测试中的重复工作。

采用大爆炸方法的系统集成最适合绝大多数组件已经就绪且稳定，要增加或修改的组件非常少的产品开发场景。在这种情况下，不是逐个测试组件接口，而是集成了所有的组件后进行一次测试，可比分多个步骤进行组件集成节省工作量和时间。

大爆炸集成适合接口稳定且缺陷很少的产品。

虽然这种方法可以节省时间和工作量，但也并不是没有缺点。对产品的发布时间和质量有影响的一些主要缺点有：

1. 如果在系统集成过程中出现失效或缺陷，则很难在产品中定位，以找出缺陷所在的接口，可能又要回到调试阶段，重新关注特定的接口并进行测试。

2. 确定由谁来消除缺陷报源可能会很困难。

3. 集成测试是在最后进行的，发布日期临近的压力非常大。作用在工程师身上的这种进度压力会影响产品的质量。

4. 如果有些组件的就绪时间推迟，就不能先行测试其他接口，导致浪费时间。

由于有这些缺点，选择集成测试的方法就变得极为重要，需要将以上方法合理地组合，以达到提高集成测试时间效率和质量的目的。

5.2.5 选择集成方法

表5-4给出了一些大概层次的集成方法选择指南。正如前面提到过的，集成方法不仅取决于过程和并发模型，而且取决于各种其他因素。

表5-4 集成方法选择指南

序号	原 因	建议使用的集成方法
1	需求和设计很清楚	自顶向下
2	需求、设计和体系结构不断改变	自底向上
3	需求结构改变，设计稳定	双向
4	对现有体系结构或设计很有信心，时间较小	大爆炸
5	以上各种情况的组合	经过精心分析后选择以上一种方法

5.3 集成测试作为一个测试阶段

本章开头已经介绍过，集成测试作为一个测试阶段，从有两个组件可以一起测试开始，到所有组件可以作为一个完整系统一起运行提供系统产品功能为止。在集成测试阶段，关注点不仅要放在组件的功能是否有效，还要放在组件是否能够一起运行，提供了系统和系统功能。

集成测试阶段关注发现由于各种被测构建的组合引起的缺陷，而不是关注组件或少量组件。集成测试作为一种类型，关注接口的测试，这是集成测试阶段的一个子集。当把下系统或系统组件放到（即集成到）一起时，不仅接口会暴露缺陷，各种其他原因，例如用法、对产品领域的不完备理解、用户错误等，都会暴露缺陷。因此，集成测试阶段既要关注接口，也要关注使用流，必须注意这一点，以避免将集成测试类型和集成测试阶段相混淆。

集成测试作为一个阶段包括在这个阶段内完成的不同活动和不同类型的测试。这个测试阶段要保证测试的彻底性和功能覆盖率，为了达到这个目标，不仅要关注按计划执行的测试用例，而且要关注没有列入计划的测试，即所谓“即兴测试”。第1章在介绍测试原理时说过，测试是没有尽头的，质量不仅取决于预先写好的测试用例，即兴测试对于集成测试阶段非常重要。即兴测试有很多不同的叫法，例如探索式测试、猴子测试、意外测试等（请参阅第10章）。所有这些测试都在集成测试阶段发挥同样的作用，即发现执行计划内的测试用例没有发现的缺陷。即兴测试有助于对某些测试团队很难发现、在开始很难想象的问题进行定位，还有助于系统的所有内部用户对软件产生一种不错的感觉，对产品有一种总体的接受感。

> 从将两个组件集成到一起开始，到所有系统组件在一起运行为止的所有测试活动，都可以认为是集成测试阶段的一部分。

集成测试阶段包括开发和执行覆盖多个组件和功能的测试用例，当不同组件或功能组合在一起测试相关操作的序列时，叫做场景。场景测试是一种经过策划的活动，用来检查不同的使用模式，并将其合并为测试用例，叫做场景测试用例。以下详细讨论场景测试。

5.4 场景测试

场景测试是：用来评价产品的一组实际用户的活动。场景测试还定义为包含客户场景的测试。

有两种方法可用来开发场景：

1. 系统场景
2. 用例场景/基于角色的场景

5.4.1 系统场景

系统场景是一种方法，用于场景测试的一组活动涵盖系统的多个组件。以下方法可以用于开发系统场景：

故事线 开发一个故事线，将最终用户可能执行的各种产品操作活动组合到一起。用户进入自己的办公室，登录系统，检查电子邮件，给一些邮件写回信，编译程序，执行单元测试等。所有这些在日常工作中完成的典型活动组合在一起构成一个场景。

生存周期/状态转移 考虑一个对象，导出对该对象进行的各种转换/修改，导出覆盖这些迁移的场景。例如，在银行储蓄账户中，首先可以用一定数额的钱开一个账户，然后存一笔钱，取一笔钱，计算利息，等。所有这些活动都作用于“钱”这个对象，不同的转换也作用

于“钱”这个对象，构成不同的场景。

来自客户的部署/实现故事 通过已知客户部署/实现细节开发场景，由该实现中的各种用户创建一组活动。

业务垂直线 可视化产品/软件用于不同纵向业务的方式，并创建一组行动作为创建，描述具体的纵向业务。以采购功能为例，不同业务垂直线上，例如药房、软件公司和政府组织的采购功能是不同的，可视化这些不同类型的测试，能使产品具有“多种用途”。

战场 创建一些场景证明“产品有效”，并创建一些“尽力使系统崩溃”的场景，证明“产品无效”。这样可以为以上提到的场景增加内容。

如果组合使用以上提到的绝大多数方法，而不是孤立地使用，可以更有效地创建一组场景，场景不应是一组相互没有联系的活动，场景中的任何活动永远是前一个活动结果的继续。有效的场景要综合考虑当前客户实现和产品的未来使用，以及开发即兴测试用例。只考虑一个方面（例如只考虑当前客户的使用或未来客户需求）会使场景的效果降低。如果只考虑测试当前客户的使用，就可能没有充分测试新特性；如果场景只考虑未来市场，可能使场景测试只涉及新特性，一些已有的功能可能没有测到。采用以上介绍的各种方法恰当组合创建场景，对于场景测试的有效性是非常关键的。

覆盖率总是场景测试中有关功能的一个大问题。这种测试并不要求覆盖产品特性和使用的不同排列组合。但是，通过运用某种简单技术，可以得到对场景测试活动覆盖率比较满意的理解。表5-5通过一个例子解释这种概念。

表5-5 场景测试的活动覆盖率

如实用户活动	频率	优先级	适用环境	覆盖的次数
1. 登录应用	高	高	W2000，W2002，XP	10
2. 创建一个文档	高	中	W2000，XP	7
3. 修改文档	中	中	W2000，XP	5
4. 列出对象参数	低	中	W2000，XP，W2002	2
5. 写电子邮件	中	中	W2000，XP	6
6. 加附件	低	低	W2000，XP	2
7. 发送邮件	高	高	W2000，XP	10

从表5-5可以看出，系统场景测试的场景集已经很好地覆盖了重要活动。这种活动表还有助于确保所有活动都根据在客户现场的使用频率和客户使用确定的相对优先级进行了覆盖。

5.4.2 用例场景

用例场景是关于用户按照不同角色和相关的参数，预期使用系统按步骤描述的规程。用例场景可以包括故事，图片和部署细节。用例对于无任何歧义地解释客户问题和软件解决其问题的方式非常有用。

一个用例可以包含多个角色或根据角色执行不同活动的用户类。有些活动是各角色公用的，有些活动是非常特殊的，只能由特定角色执行。用例场景把属于不同角色的用户叫做参与者，把产品应该针对特定活动作出的响应叫做系统行为，在参与者和系统之间交互的特定角色的用户叫做代理。

为了解释用例场景的概念，举一个从银行提取现金的例子。储户填写一张支票并交给银行工作人员。银行工作人员通过计算机检验该账户的余额，并将所要求的现金交给储户。这

个例子中的储户就是参与者，银行工作人员是代理，计算机给出的响应，即给出账户余额的行为是系统响应，如图5-5所示。

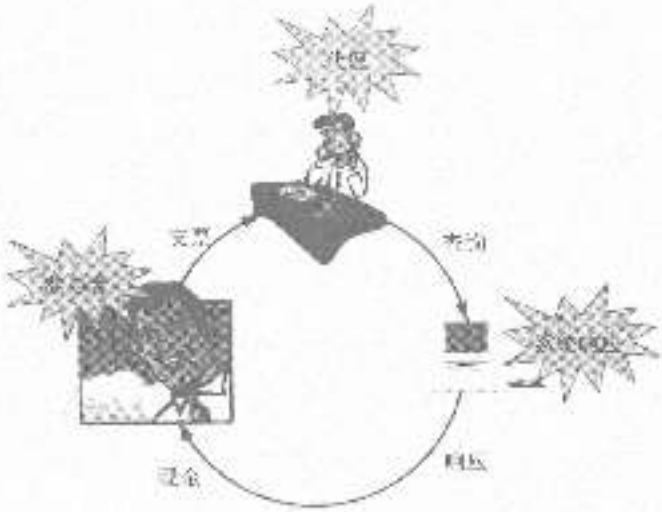


图5-5　银行用例场景示例

在测试用例中描述不同角色的方法有助于不深入产品细节就测试产品。在上面的例子中，参与者（储户）不需要知道银行工作人员做什么和自己与计算机交互使用了什么命令。参与者只关心得到现金。代理（银行工作人员）不关心计算机工作的逻辑和方式，只关心通过计算机了解是否可以付给储户所要求的现金。但是，需要在使用代理活动和参与者活动序列前测试系统行为（计算机逻辑）。在这个例子中，参与者所执行的活动和代理可以由对产品没有多少知识的测试人员测试。对产品有深入了解的测试人员可以进行系统行为部分的测试。他们需要了解代码工作的逻辑和方式，了解系统响应是否准确。

前面已经介绍过，参与者和代理对应不同类型的用户角色，模拟不同类型的用户也需要清楚地理解业务和系统响应，因为每个用户都要清晰地理解产品是怎样实现的。因此，使用用例模型的测试人员，一个人测试行动，另一个人测试系统响应，相互补充彼此的测试，同时测试产品的业务和实现问题。

用例的代理部分并不是在所有情况都需要，对于包含客户和系统的完全自动化的系统来说，编写用例时不需要考虑代理。把前面的例子扩展为使用ATM机提取现金，表5-6说明了参与者和系统响应如何通过用例描述。

表5-6　ATM机现金提取用例中的参与者和系统响应

参与者	系统响应
用户要提取现金并将卡插入ATM机	要求输入密码或个人标识号（PIN）
用户输入密码或个人标识号	确认密码或个人标识号，给出包含账户类型的列表
用户选择账户类型	要求用户输入要提取的金额
用户输入要提取的现金金额	按要求正确的现金，更新账户余额，吞没卡片，付给现金
从ATM机取走现金	打印收据

这种记述场景和测试的方法，使客户使用起来更容易也更实用。用例不只是用于测试，在有些产品实现中，用例在设计和编码阶段之前就已准备。所有开发活动都根据用例文档进

行。在极限编程模型中，用户叫做用户故事，形成体系结构设计和编码阶段的基础。因此，用例对于合并业务视角和实现细节，并将其一起测试是非常有用的。

5.5 缺陷围歼

缺陷围歼是一种即兴测试，由人员装扮成组织中的不同角色在一起同时测试产品。这种做法被应用程序开发公司广泛使用，他们的产品可以由承担不同角色的人员使用。在实施缺陷围歼时所有参与者所进行的测试不是基于已写好的测试用例，要测试什么内容是由参与测试的每个人的判断力和创造力决定。他们还可以尝试一些超过产品规格说明的操作。缺陷围歼总结了很多在测试界很著名的很好的实践：

1. 使人员"跨越边界并跨越已分配领域测试"
2. 让在组织中充当不同角色的人一起进行测试——"测试不只是测试人员的事"
3. 让组织中的每个人在产品交付前使用产品——"各尽其责"
4. 引入新人来发现新的缺陷——"新人没有多少偏见"
5. 引入对产品有不同理解程度的人在一起随机地测试产品——"软件的用户是不同的"
6. 使测试不等待文档完成——"测试要等到所有文档都完成后才能开始吗？"
7. 既要允许人们说"系统运行正常"，也要允许人们说"使系统崩溃"——"测试并不是获得出系统运行正常或不正常的结论"

尽管缺陷围歼被认为是一种即兴测试，但并不是缺陷围歼的所有活动都是没有计划的。缺陷围歼的所有活动，除被测试的内容外都是有计划的。下面给出了一些步骤：

步骤1：选择缺陷围歼的频度和持续时间

步骤2：选择合适的产品版本

步骤3：与每一位参与人员就每次缺陷围歼的目标进行沟通

步骤4：建立和监视用于缺陷围歼的实验室

步骤5：采取行动解决问题

步骤6：优化缺陷围歼所涉及的工作

5.5.1 选择缺陷围歼的频度和持续时间

缺陷围歼是一项包含大量工作（因为很多人参加）和需要大量策划工作（从上一节讨论可以看出）的活动。频繁缺陷围歼的投入回报很低，而过少次数的缺陷围歼又不能达到发现所有缺陷的目标。持续时间也是一个重要因素。缩短有很多人参加的缺陷围歼持续时间是一项很大的节省。另一方面，如果持续时间很短，要完成的测试量可能又达不到目标。

5.5.2 选择合适的产品版本

由于缺陷围歼需要大量的人员、工作量和策划，因此缺陷围歼需要一个高质量的版本。回归测试版本（请参阅第8章）是很理想的，因为在这种版本中，所有新功能和缺陷修改都已经过测试。编码还在进化的中间版本和没有经过测试的版本都会降低缺陷围歼的效果。对于有很多人参加的缺陷围歼，高质量的版本会使大家增强对产品和工作进展的信心。此外，如果缺陷围歼的测试人员发现过量的缺陷或非常严重的缺陷，则测试人员的信心就会下降，产品不稳定的印象会持续很长时间。

5.5.3 对缺陷围歼的目标进行沟通

尽管缺陷围歼是一种即兴活动，但是其目的和目标还是非常清楚的。由于缺陷围歼包含充当不同角色的人，因此必须要求他们的工作满足缺陷围歼的目的和目标。目标应该是发现大量未发现的缺陷，或发现系统需求（CPU、内存、硬盘等），或发现不可重现或随机缺陷，这些缺陷通过其他有计划的测试很难发现。测试工程师很容易发现的缺陷，不应该是缺陷围歼的目标。在会前告知与会者这些之后，他们就会更好地实现目标要求。

5.5.4 建立和监视实验室

由于缺陷围歼是经过策划、短暂和耗费大量资源的活动，因此需要建立并监视实验室。在缺陷围歼正式开始之前，需要精心策划合适的配置和资源（完成缺陷围歼的硬件、软件和人员）。由于所涉及的工作量很大，确保能进行合适的配置，使每个人都能在被测软件上完成自己的工作是非常关键的。绝大多数失败的缺陷围歼都是由于不充足的硬件、错误的软件配置和与软件的性能和可伸缩性有关的理解问题。在缺陷围歼期间，需要监视由于产品参数和系统资源（CPU、RAM、硬盘、网络等）引起的缺陷并进行处理，以便用户可以继续使用系统完成整个缺陷围歼。

有两种缺陷会出现在缺陷围歼中，一种是产品中的缺陷，由用户报告，叫做功能缺陷。在监视系统资源时发现的缺陷，例如内存泄漏、过长的转换时间造成堵塞、占用大量系统资源或有很大影响等，叫做非功能缺陷。缺陷围歼是发现功能和非功能缺陷的唯一测试方法。但是，如果没有很好地建立实验室或没有恰当地监视，一些非功能缺陷有可能根本就没有被注意到。

5.5.5 采取行动解决问题

最后一个步骤是在缺陷围歼后采取必要的改正行动。通过用户发现大量缺陷是缺陷围歼的目的，也是缺陷围歼的正常结果。很多缺陷都是重复的，但是不同用户对相同缺陷的不同解释，相同缺陷的影响会在不同的地方以不同的形式表现出来，都使这些缺陷很难叫做重复的。由于可能会有大量缺陷，因此解决缺陷围歼发现的问题不能逐个进行。如果逐个问题地修改代码，会很难解决所有问题，需要在更高层次上把问题分类，以免同样的问题再出现在将来的缺陷围歼上。有时一个缺陷就是一类，有时多个缺陷算作一类。例如“在所有的组件中，所有输入的员工编号都应该在业务逻辑使用之前进行确认。”这样就使不同组件的所有缺陷分成一类。缺陷围歼发现的所有问题都要经过完备的代码和设计审查、分析，并一起从根源上解决。因此，缺陷围歼的结果还可以用于未来缺陷围歼的缺陷预防。

5.5.6 优化缺陷围歼所涉及的工作

由于缺陷围歼涉及大量人员，实施缺陷围歼常常需要耗费很大工作量。如果记录了缺陷围歼的目标和输出，则有多种方法可以优化工作量。前面提到的采用经过测试的软件版本、提供合适的环境、明确目标等，都可以压缩工作量并达到目标。压缩缺陷围歼工作量的另一种方法是在组织大型缺陷围歼前先组织“小型”缺陷围歼。在小型缺陷围歼上会发现一些明显缺陷。由于缺陷围歼是一种集成测试阶段的活动，所以在吸收其他人参加之前，可以由集成测试团队先进行缺陷围歼。为了防止组件级缺陷出现在集成测试阶段，还可以先组织小型缺陷围歼找出功能级缺陷，然后再对产品进行集成。因此，缺陷围歼可以进一步分成：

1. 功能/组件缺陷围歼
2. 集成缺陷围歼

3. 产品缺陷围歼

为了解释缺陷围歼分类对工作量的压缩，假设要举行有100人参加、持续2个小时的三次缺陷围歼，总工作量是3×2×100=600人时。如果各有10人的功能组件团队和集成测试团队可以进行两轮小型缺陷围歼，可以发现三分之一的缺陷，那么根据以下计算可以得出压缩20%工作量的结论：

A. 两轮产品缺陷围歼的总工作量——400人时

B. 两轮功能缺陷围歼的工作量（2×2×10）——40人时

C. 两轮集成缺陷围歼的工作量（2×2×10）——40人时

节省的工作量=600−(A+B+C)=600−480=120人时，也就是20%

缺陷围歼是一种即兴测试，由人们在集成测试阶段的同一段时间扮演不同角色，发现有计划的测试可能遗漏的所有类型的缺陷。

这只是一种大致计算，因为本节前面提到的步骤（步骤1－6）所包含的工作量也要计入每次缺陷围歼。不管是功能级、集成级还是产品级缺陷围歼，这些步骤对于每轮缺陷围歼都要重复进行。

5.6 小结

集成测试既是一种测试类型也是一个测试阶段。集成测试在每个组件经过测试并交付后开始。采用第4章介绍的黑盒测试方法，所有组件都经过测试并能正常运行并不意味着这些组件放到一起经过集成后还能够正常运行，这是很多公司常常忽视的一个重要的测试类型/阶段。由于项目的巨大压力和开发进度的拖延，集成测试由于处在组件测试和系统测试阶段之间而被淡化。近来一些公司已经建立了专门关注集成测试的测试团队，这说明长期关注集成测试问题是值得的。如果恰当实施，集成测试会减少一些在系统测试阶段会发现的缺陷，下一章将专门介绍系统测试。

问题与练习

1. 对于以下给出的每种系统，请给出最适合的集成测试类型：
   a. 产品的版本不断演化，每个版本都引入与其他已有功能关系不大的新功能，也就是说，现有功能很稳定，受新加功能的影响不大。
   b. 产品由接口定义很清楚的组件构成。因此，桩模块和驱动模块从一开始就可以使用。
   c. 产品提供标准的数据库查询和显示功能，查询功能实现标准的JDBC接口，显示功能提供与多平台和显示系统（例如Windows、Macintosh等）接口的灵活性。
2. 请考虑一个典型的大学学术活动信息系统，确定这个系统用例的各种典型的代理、参与者和预期系统行为。
3. 请考虑一个产品，其版本不断演化，每个版本大约需要半年才能上市。对于每个版本，在什么时候进行每种类型的缺陷围歼？明确说明作出决策的各种假设。
4. 本章给出的哪种方法适合针对以下情况导出系统场景：
   a. 客户说，“使用上一个版本时，我很难把数据从老版本中转移出来——不能对客户主机备份数据，不能为折扣表加新的列。”
   b. 产品部署在FMCG行业和零售行业，为每个行业进行定制。
   c. 一个测试工程师进行单元测试，更新缺陷数据库，向组件的编写者发送电子邮件，与他们交换意见并更新日志。
5. 请举一个例子，说明集成测试在什么场合可以作为一种白盒测试方法（即需要检查代码）使用，什么场合可以作为黑盒测试方法使用。

第6章 系统测试和确认测试

6.1 系统测试概述

系统测试是对完整集成后的系统进行测试的阶段，用来评价系统对具体需求规格说明的符合性。系统测试在单元、组件和集成测试阶段之后进行。

对完整集成后的产品和解决方案进行测试，用来评价系统对具体需求规格说明的功能和非功能的符合性的测试叫做系统测试。

系统是完整的一套经过集成的组件，一起提供系统功能和特性。系统还可以定义为一起提供系统功能和解决方案的一组硬件、软件和其他部件。为了测试整个系统，需要理解整个产品的整体行为，系统测试有助于发现可能难以直接与模块或接口关联的缺陷，发现产品整体的设计、体系结构和代码的基础问题。

系统测试是既测试产品的功能也测试非功能的唯一测试阶段。对于功能测试，系统测试关注实际客户对系统和解决方案的使用，系统测试模拟客户部署。对于通用系统，系统测试还意味着测试产品不同的业务级深和应用领域，例如保险、银行、资产管理等。

对于非功能方面，系统测试有不同的测试类型（又叫做质量因素），包括：

1. **性能/负载测试** 评价系统完成所要求的功能所用的时间或响应时间，并与同一产品的不同版本或不同的竞争产品进行比较，称为性能测试。下一章将介绍性能测试。

2. **可伸缩性测试** 通过大量资源确定系统容纳最大能力的测试。本章后面将详细介绍这种测试。

3. **可靠性测试** 评价系统或系统独立组件在给定时间段内反复执行所要求功能的测试。本章后面将详细介绍这种测试。

4. **压力测试** 使系统超过已描述需求或系统资源（例如硬盘空间、内存、处理器使用）的限制，确保系统不会意外崩溃的测试。本章后面将详细介绍这种测试。

5. **互操作测试** 保证两个或更多产品能够交换信息、使用信息和密切协同的测试。本章后面将详细介绍这种测试。

6. **本地化测试** 验证经过本地化的产品能够以不同的语言运行的测试。第9章将详细介绍这种测试。

根据背景的不同，系统测试的定义会不断变化，以覆盖更宽和更高层的问题。提供给客户的解决方案可能是多个产品的一种集成。每个产品又可能是多个组件的一种组合。产品组件的提供商可以把独立组件作为系统进行组件的系统测试。从产品组织的视角看，集成这些组件是子系统测试。当由不同组件开发商提供的所有组件由产品组织装配到一起时，是作为系统整体测试的。在下一个层次上，解决方案集成商将多个来源的产品进行组合，为客户提供完整集成的解决方案。把很多产品放在一起叫做系统，对这种集成解决方案进行系统测试。图6.1描述了不同组织从自己的视角和全局视角进行系统测试。

系统测试根据通过详细体系结构/设计文档、模块规格说明和系统需求规格说明提取的信

总编写的测试用例为基础实施。系统测试用例在研究了组件和集成测试用例后创建，同时在设计中用来测试系统整体的测试用例，也可以根据用户故事、客户讨论和观察典型客户使用来开发系统测试用例。

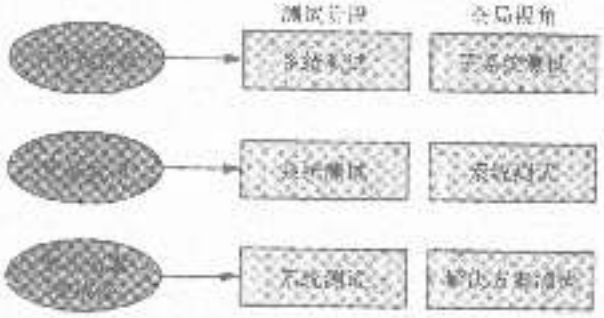


图6-1 系统测试的不同视角

系统测试可以不包含很多负面场景验证，例如测试不正确和负面取值。这是因为这种负面测试应该已经在组件和集成测试中完成，并且可能不反映真实客户的使用。

系统测试可以在单元、组件和集成测试完成后进行，这样可以保证更基础的程序逻辑错误和缺陷已经被更正。除了验证产品的业务需求，系统测试还要保证产品已经可以转移到用户确认测试层次。

6.2 实施系统测试的原因

系统测试通常由独立测试团队承担。这种独立测试团队与承担组件和集成测试任务的团队不同。系统测试团队一般要向高层经理而不是向项目经理报告，以避免产生利益冲突，测试人员能够自由进行系统测试。从独立视角测试产品，并结合客户视角，使系统测试变得唯一、不同和有效。由独立团队实施系统测试可以消除对产品的偏见。通过独立系统测试团队引入的“新目光”有助于发现组件和集成测试遗漏的问题。

在系统测试期间要验证完整产品的行为，包括涉及多个模块、程序和功能的测试，测试完整产品的行为是很关键的。因为很多人错误地认为经过单独测试的组件放到一起后仍然能够正常运行。

系统测试有助于在其部署中客户发现缺陷之前，尽可能多地发现缺陷。这是测试团队在将产品交付客户之前发现各种残余产品缺陷的最后机会。

系统测试的目标是在产品向客户发布之前发现产品级缺陷，并建立信心。组件和集成测试阶段关注发现缺陷。如果系统测试的关注点也与组件和集成测试的关注点相同，并且发现很多缺陷，人们就会认为产品不稳定（特别是系统测试比组件或集成测试更接近产品发布）。与此相反，如果系统测试没有发现多少缺陷，人们又会对系统测试阶段的有效性和价值产生怀疑。因此，系统测试总是要在产品发布之前发现缺陷和建立信心这两个目标之间寻找平衡。

由于系统测试是发布前的最后测试阶段，又因为开发和测试所需的时间和工作量问题以及最后时刻变更所带来的潜在风险，并不能修改代码中的全部缺陷，所以要对这些缺陷进行影响分析，以降低发布带缺陷产品的风险。如果客户受这些缺陷影响的风险很高，那么在发布前需要修改缺陷；否则原封不动地发布产品。对缺陷进行分析并分类还可以使大家了解产品发布后客户发现缺陷的可能性。这类信息有助于策划采取变更回滚和文档修改等选择的措施。

因此系统测试有助于降低产品发布的风险。

系统测试对于其他测试阶段有很强的互补性。组件和集成测试把功能规格说明和设计作为输入。这类测试的主要关注点是技术和产品实现，而客户场景和使用模式是系统测试的基础。因此，系统测试阶段通过明确关注客户对早期测试阶段形成补充。系统测试阶段有助于将关注点从产品开发团队转移到客户及其对产品的使用上。

总之，实施系统测试有以下理由：

1. 在测试中引入独立视角
2. 在测试中引入客户视角
3. 引入"新目光"，发现以前的测试没有发现的缺陷
4. 在一种正式、完备和现实的环境中测试产品行为
5. 测试产品功能和非功能方面的问题
6. 建立对产品的信心
7. 分析和降低产品发布的风险
8. 保证已满足所有需求，产品已具备交付确认测试的条件。

6.3 功能测试与非功能测试

功能测试要测试产品的功能和特性。非功能测试要测试产品的质量因素。系统测试由功能和非功能测试验证构成。

功能测试验证系统的预期，要测试产品的特性或功能。就需求满足而言只有两种情况——满足或不满足。如果需求没有很好地写清楚，则对功能需求的理解就可能有多种方式。因此功能测试应该采用描述产品行为的方式写清预期结果的文档。功能测试由执行测试用例的简单方法和步骤组成。功能测试的结果一般取决于产品，而不是环境。功能测试使用预先确定的一组资源和配置，但有少数测试类型除外，例如第4章所述，在兼容性测试中配置起很大作用。功能测试不仅需要深入的客户和产品知识，还要求领域知识，以开发不同的测试用例发现严重缺陷，因为功能测试的关注点是找出缺陷。功能测试中出现的失效通常会导致对代码的修改，以得到正确的行为。功能测试在测试的所有阶段实施，例如单元测试、组件测试、集成测试和系统测试。前面已经提到过，系统测试阶段中的功能测试（功能系统测试）关注产品特性，而早期测试阶段关注组件特性和接口特性。

非功能测试用于验证质量因素（例如可靠性、可伸缩性等）。这些质量因素也叫做非功能需求。非功能测试要求以定性和定量的形式将预期结果写入文档。非功能测试需要大量资源，对于不同的配置和资源，结果也是不同的。非功能测试是非常复杂的，需要收集和分析大量数据。非功能测试的关注点是评判产品，而不是发现缺陷。非功能测试的测试用例包括明确的通过/不通过评判准则。但是，测试结果既要依据通过/不通过定义，也要依据运行测试的经历。

除了验证通过或不通过状态，非功能测试结果还受执行测试所用工作量和执行测试期间所遇问题的制约。例如，如果经过10轮迭代性能测试满足了通过准则，那么这个过程就不够好，测试结果就不能认为是通过。产品或非功能测试过程都需要修改。

非功能测试要求理解产品行为、设计和体系结构，还要了解竞争产品的情况。由于需要仔细分析大量生成的数据，所以需要分析和统计技能。没有通过非功能测试对设计和体系结构的影响比对产品代码的影响大得多。由于非功能测试是不可重复的，需要稳定的产品，因此要在系统测试阶段实施。

表6-1归纳了以上讨论的要点。

表6-1 功能测试与非功能测试的比较

测试角度	功能测试	非功能测试
包含	产品特性和功能	质量要素
测试	产品行为	行为与结果
得出结论	用以与预期结果的简单比较	需要进行复杂分析或复杂数据
影响结果的因素	产品实现	产品实现、资源和配置
测试关注点	被客户看到	产品的质量参数
所需知识	产品和领域	产品、领域、设计、体系结构、统计技能
未通过的常见原因	代码	体系结构、设计和代码
测试阶段	单元、组件、集成、系统	系统
测试用例可重复性	可重复多次	只在未通过时和针对不同配置和环境
配置	针对一组测试用例一次建立	针对每个测试用例改变配置

表6-1给出的一些要点像是主观判断。例如，功能测试也需要设计和体系结构知识。因此，表6-1给出的要点仅供参考，不是严格的规则。

由于系统测试阶段要测试功能和非功能内容，因此可能会提出的问题是“这两类测试在系统测试中测试用例的合适比例是多少？”由于功能测试从单元测试阶段开始就是测试的关注点，而非功能测试只有到系统测试阶段才进行，因此最好将很大一部分系统测试的工作量放在非功能方面。非功能和功能测试三七开应该是比较理想的情况，五五开也是不错的起点。但是，这只是一种建议，合适的比例更多地取决于背景情况。发布类型、需求和产品。

6.4 功能系统测试

正如前面解释过的，功能测试在不同的测试阶段实施，关注点是产品级特性。由于功能测试在不同测试阶段实施，因此会出现两个明显的问题。一是重复，二是灰色区域。重复是指多次实施相同的测试。灰色区域是指在所有阶段都遗漏的测试。各阶段之间存在少量的重复是不可避免的，因为参与的团队不同，在编写系统测试用例之前进行交叉评审（邀请以前测试阶段的团队参与评审），检查以前阶段的测试用例，有助于最大限度地降低重复率。有少量的重复是明智的，来自不同团队的不同人员以不同的视角进行测试会发现新的缺陷。

出现测试的灰色区域是因为缺少产品知识、缺少客户使用知识、缺少测试团队间的协同。这种测试灰色区域使缺陷溜网，并影响用户的使用。实施某个具体阶段测试的团队可能假设某个特定测试会在下个阶段实施，这是出现这种灰色区域的一个原因。在这种情况下，需要在尽可能早的阶段制订明确的团队测试交互指南。将测试用例从较后阶段移到较前阶段，比从较前阶段推迟到较后阶段好，因为测试的目的就是尽可能早地发现缺陷。这些工作要在当前阶段的所有测试工作完成后进行，不能妨碍当前阶段的测试。

实施系统功能测试有很多方法，功能测试产品级测试用例的导出也有很多方法，以下给出的是一些常见方法。

1. 设计/体系结构验证
2. 业务垂直测试
3. 部署测试
4. Beta测试

5. 符合性的认证、标准和测试

6.4.1 设计/体系结构验证

在这种功能测试方法中，要对照设计和体系结构开发和检查测试用例，检查这些测试用例是否实际的产品级测试用例。把这一点与集成测试作一下比较，集成测试的测试用例是对照接口进行创建的，而系统级测试用例首先创建和对照设计和体系结构验证，检查其是产品级还是组件级测试用例。集成测试用例关注模块或组件之间的交互，而功能系统测试关注整个产品的行为。这样做的一种额外好处是可保证产品实现的完备性。这种技术有助于确认根据客户场景编写的产品特性，并用产品实现进行验证。如果有对应客户场景的测试用例采用这种技术没有通过确认，那么就将其转移到合适的组件或集成测试阶段。由于功能测试要在各个测试阶段实施，拒绝这种测试用例并将其转到较早测试阶段以尽早捕获缺陷，避免在后期阶段造成重大问题是很重要的。以下给出了一些拒绝功能系统测试测试用例的建议：

1. 这个测试用例关注的是代码逻辑、数据结构和产品单元吗？（如果是，则应属于单元测试。）

2. 这个测试用例在任何组件的功能规格说明中描述了吗？（如果是，则应属于组件测试。）

3. 这个测试用例在集成测试的设计和体系结构规格说明中描述了吗？（如果是，则应属于集成测试。）

4. 这个测试用例关注的是不能被客户看到的产品实现吗？（它关注的是实现，应该属于单元测试或组件测试，或集成测试。）

5. 这个测试用例恰当地综合了客户使用和产品实现了吗？（客户使用是系统测试的前提。）

6.4.2 业务垂直测试

像工作流自动化系统这样的通用产品可以供不同的商业公司和服务公司使用。使用和测试针对不同业务纵深的产品，例如保险、银行、资产管理等，并验证业务运作和使用，叫做“业务垂直测试”。对于这种类型的测试，要改变产品中的规程以适应业务过程。例如，在贷款处理中，贷款首先由负责人认可，然后转给银行工作人员。在理赔处理中，索赔申请首先由工作人员进行整理，然后转给负责人认可。产品要创建像工作人员和负责人这样的用户对象，然后为其分配操作。这是定制产品以适应业务的一种方法，有些操作只能由某些用户对象完成，这叫做基于角色的操作。重要的是产品要理解业务过程，把定制作为一个特性，以便不同的业务纵深可以使用产品。在定制特性的帮助下，可以改变系统的一般工作流，以适应特殊的业务纵深。

另一个重要的方面叫做术语。为了解释这个概念，先举一个电子邮件的例子。在保险行业内，发送的电子邮件可能叫做索赔申请，而发送给贷款处理系统中的电子邮件却要叫做贷款申请。用户会很熟悉这种术语，而不是适用的“电子邮件”这个词。用户接口应该反映出这种术语，而不是使用通用术语电子邮件，否则会阻碍产品的针对性，用户也可能难以清楚地理解。发送给血库服务公司的电子邮件不能以与员工之间内部传递的电子邮件一样的优先级处理。产品要通过发送者的背景信息和邮件内容做出这类有差别的处理。有些电子邮件或电话呼叫需要由产品进行跟踪，以检查是否达到所约定的服务水平（SLAS）。例如，发送给血库服务公司的电子邮件需要以最快的速度响应。有些电子邮件甚至可以根据电子邮件管理系统确定的规则自动应答，以达到所约定的服务水平。因此产品的术语特性可以把电子邮件

恰当地叫做索赔或交易，并且可以与背景信息和属性关联起来，使得特定的业务纵深领域能够更好地使用。

与业务垂直测试有关的另一个问题是联合体。并不是业务纵深所需的所有工作都由产品开发组织完成。解决方案集成商、服务提供商会向产品组织支付产品使用许可费用，并以自己的名称和形象销售产品和解决方案。在这种情况下，产品名称、公司名称、技术名称和版权都属于解决方案提供商，产品组织会改变产品的名称。产品应该为这类联合体提供特性，也要作为业务垂直测试的一部分。

业务垂直测试可以用两种方法进行，即模拟和复制。在业务垂直测试模拟中，客户或测试人员要测试需求和业务流。在复制中，要获取客户数据和过程，对产品进行完全定制和测试，并把定制后经过测试的产品发布给客户。

在讨论集成测试的那一章已经介绍过，要通过场景测试测试业务纵深。场景测试只是进化场景和思路的一种方法，并不是穷尽的，场景测试更多地是从接口及其交互的观点实施。通过集成测试创建一些业务纵深场景可确保系统测试阶段进展更迅速。采用诸如以上所描述的定制、术语和联合体，在实际客户环境中对业务纵深领域进行完备的测试。

6.4.3 部署测试

系统测试是产品交付前的最后阶段，到这个时候应该知道预期客户及其配置，有时产品公司还会做出销售的承诺。因此，系统测试是针对这些正在等待产品发布的客户进行测试的恰当时候。特定产品版本是短期成功还是失败，主要根据这些客户需求的满足程度判断。这种在产品开发组织内进行以确保满足客户部署需求的（模拟）部署测试叫做离场部署。

部署测试也通过使用客户场地中已有的资源和环境在产品发布后，由产品开发组织和将使用该产品的组织联合实施，叫做现场部署。尽管现场部署不在系统测试阶段实施，但是为了保持内容的完整性还是在这里讨论这个问题。通常由系统测试团队完成现场部署测试。现场部署测试被认为是将在本章后面介绍的确认测试的一部分，是离场部署测试的一种扩展。

现场部署测试分两个阶段进行。第一个阶段（Stage 1）要采集实际系统的真实数据，建立类似的机器和配置镜像环境，在镜像部署机器上重新执行用户操作，这样可以了解增强或类似的产品是否能提供现有功能，且不影响用户。这样可以降低产品不能满足现有功能的风险，因为部署没有经过充分测试的产品可能给组织造成重大业务损失。有些部署测试采用智能记录器记录在真实系统中发生的事务，并在镜像系统上做这些操作，然后再将结果与真实系统进行比较。记录器的目标是暂时镜像系统和真实系统在业务事务上保持一致。在第二阶段（Stage 2），在成功完成第一阶段工作的基础上，在镜像系统上建立一个运行新产品的正式系统，采用定期备份和其他方法记录从镜像系统正式运行以来所增加的事务。在第一阶段使用过的记录器在第二阶段也可以继续使用，但是，建议采用不同的方法记录增加的事务，因为还会由于记录器的缺陷产生失效。这个阶段有助于避免出现重大失效，因为有些失效只有在运行一段时间后才能观察得到。在这个阶段，以前使用的实际系统和镜像系统正式运行以来记录的事务都要保留起来，如果在这个阶段出现重大失效，要能够回到老系统上。如只在这个部署第二阶段有一段时间（比方说一个月）没有发现失效，那么就可以认为现场部署是成功的，并用新系统替代老的实际系统。图6-2给出了部署测试的第1阶段和第2阶段。

在图6-2的第1阶段中，可以看到记录器插在用户和实际系统之间记录所有事务。通过实

际系统记录的所有事务，事后都要在测试工程师的监视下（用虚线表示）在被测产品上回放。在第2阶段，测试工程师采用记录器和其他方法记录所有事务，并在老的实际系统上回放（用虚线表示）。

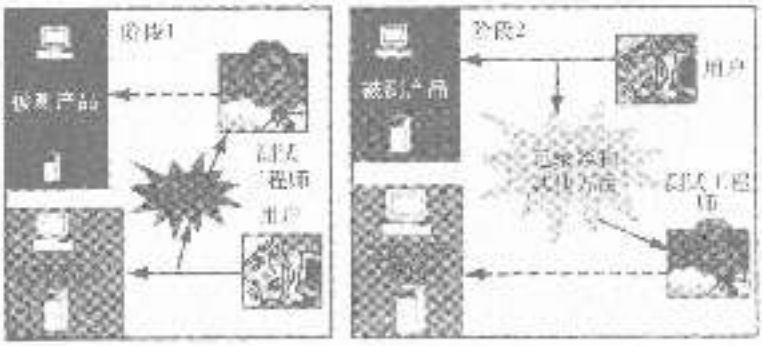


图6.2　部署测试的阶段

6.4.4　贝塔测试

开发产品需要大量工作和时间。推迟产品发布时间、产品不能满足用户需求是很常见的事情。产品在交付之后被客户拒绝对组织意味着重大损失。产品不能满足客户需求有很多原因，包括：

1. 产品有隐式和显式需求，产品没有满足隐式需求（例如易用性）可能导致客户拒绝。

2. 由于产品开发需要大量时间，到交付产品时，一些在项目初期给出的需求可能已经作废或改变，客户的业务需求不断变化，没有在产品中反映这些变化造成以后变得过时。

3. 需求是高层描述，相当模糊。尽管认识到模糊区域但是却没有和客户一起解决，导致对产品的拒绝。

4. 对需求的理解可能是正确的，但是实现会出错。这可能意味着重新设计和编码，以满足客户所要求的实现方面的特性。如果不能及时做到这一点就可能导致产品被拒绝。

5. 可使用性和文档方面的不足使客户很难使用产品，也可能导致产品被拒绝。

以上给出的只是一部分原因，可能还有更多的产品拒绝原因。为了达到系统测试的目标，即降低风险，需要定期搜集对产品的反馈信息。一种机制可以是把待测产品交给客户收集反馈意见。这叫做贝塔测试。贝塔测试由客户完成，并由产品开发组织提供帮助。在整个贝塔测试实施期间，要根据具体进度安排策划并执行各种活动，叫做贝塔程序。贝塔程序所包含的一些活动包括：

1. 采集客户及其贝塔测试需求列表，以及他们对产品的期望。

2. 写出贝塔程序进度安排并通知客户。并不是列表中的所有客户都要有统一的开始时间和结束时间。贝塔程序的结束时间应该在产品发布之前，以便在贝塔测试中发现的问题在发布前能够得到修改。

3. 提前提供一些文档供客户阅读，并就产品的使用对客户进行培训。

4. 测试产品，确保产品满足"贝塔测试进入条件"。客户和提供商的产品开发/管理小组一起确定一组贝塔测试进入和退出准则。

5. 将（具有一定质量水平的）贝塔版产品提供给客户，让他们进行自己的测试。

6. 定期收集客户的反馈信息，对要修改的缺陷指派优先级。

7. 通过修改产品和文档回应客户的反馈，及时与客户形成沟通闭环。

8. 分析并确认贝塔测试是否满足退出条件。

9. 向客户通报进展和采取的措施，正式结束贝塔程序。

10. 在产品中合并恰当的修改。

决定产品贝塔测试的进入准则和贝塔测试时间安排需要做出一些权衡考虑的选择。提供产品太早，不充分的内部测试会使客户不满意，会对产品的质量产生不好的印象。提供产品太晚，意味着贝塔缺陷修改时间太短，不能得到贝塔测试的目的。集成测试的后期和系统测试的前期开展贝塔测试是不错的时机。

客户有可能在启动贝塔测试后没有能够继续下去，或一直被动进行，没有充分使用产品并提供反馈。从客户的观点看，贝塔测试往往只是其很多活动之一，可能并没有很高的优先级。需要经常与客户保持沟通，推动他们使用产品，只要他们在使用产品上遇到问题马上提供帮助。应该让客户看到参加贝塔测试的好处，他们可以很早接触新技术，贝塔测试客户是采用新技术的先锋的形象会对客户产生很深的印象。一旦客户把贝塔测试看作是双赢的活动，既对提供商有好处，也对自己有好处，而不是只看作产品的试验末，他们参与测试的积极性就会提高。在贝塔测试中报告的缺陷应该指定与正常支持电话处理相同的优先级，唯一不同的是产品开发管理部门可能与贝塔测试客户有更直接的交互。不能达到贝塔测试目标或及时修改错误，意味着有些客户会拒绝产品。因此，一旦报告了问题要尽快把经过修改的产品提供给客户，并一定保证所作的修改满足客户的需求。

贝塔测试的一个挑战是选择合适的测试客户量。如果数量过少，则产品可能没有经过足够多样化的测试场景和测试用例。如果数量过多，则工程组织可能难以及时解决所报告的问题。因此，贝塔测试的客户数量需要在产品使用场景的多样性和有效处理其报告问题的实际管理能力上存在折衷。

最后，贝塔测试的成功严重依赖于贝塔测试客户充分了解产品中会存在缺陷，以各种方式使用产品的意愿。这可不是一件容易的工作。正如前面已经介绍过的，必须鼓励贝塔测试客户看到自己可以得到的好处，只有当客户积极地在产品进化过程中充当可信赖伙伴，他们才会参与贝塔测试。

6.4.5 符合性的认证、标准和测试

产品需要通过主流硬件、操作系统、数据库和其他基础设施构件进行验证，这叫做确认测试。不能在主流硬件、软件或设备上运行的产品，可能不适合当前和将来的使用。产品的销售取决于其是否经过主流系统的验证。产品必须在主流系统上运行，因为客户已经在主流系统上投入很多，不仅产品要能够在这些主流系统的当前版本上运行和共存，而且产品组织还应该以路线图的方式作出文字承诺，继续开发能在主流系统未来版本上运行的产品。对于这种类型的测试，产品开发组织、客户和验证产品的认证代理都同样感兴趣，但是对确认负责任的还是产品开发组织。确认代理通过开发自动测试包帮助产品开发组织，产品开发组织运行这些确认测试包，解决产品中的问题，保证测试的成功。一旦测试运行成功，测试结果会交给确认代理，由确认代理出具产品的确认。确认代理可能会重新运行测试包以验证测试结果，在这种情况下，被测产品应该与测试结果一并提交。

每种技术领域都有很多标准，产品可能需要符合这些标准，符合这些标准使得产品能够很容易地与其他产品交互，这一点非常重要。这还有助于客户不再过于担心产品与其他产品

的未来兼容性。每个技术领域都有很多标准，产品开发组织要在产品生存周期的一开始就要选择要实现的标准。可能很难遵循所有标准，有时可能由于没有实现一定的标准而产生一些非功能问题（例如性能影响）。标准也会进化（例如，网络中的Ipv6和移动技术的3G），在实现开始或进行过程中，标准细节可能才会制定出来。有些标准由开放社区负责维护，并作为公共领域标准发表（例如Open LDAP标准）。可以免费使用与这些开放标准相关的工具验证标准的实现。保证这些标准被恰当实现的测试叫做标准测试。产品完成一些标准的测试后，要写入提供给客户的发布文档中，以便让他们了解产品实现了哪些标准。

产品会有很多约定和法律方面的要求。不满足这些要求会导致业务损失，引起针对开发组织及其高层管理的法律行动。这类需求的一部分可能是合同义务，有些则是法律要求。不能满足这类需求会严重限制产品市场。例如，如果不满足可使用性指南（508可获得性指南）就不能参加英国政府组织的竞标。对于产品针对合同、法律和法规符合性的测试是系统测试团队的一个关键活动。以下是符合性测试的一些例子：

- 符合FDA 食品和药品管理的这个法案要求对像化妆品、药品和医疗科学方面的产品进行充分测试，还要求保存每个测试周期的所有测试报告、全套测试用例、执行记录文档、以及监管部门的批准，供FDA检查测试的充分性。
- 508可获得性指南 这套可获得性指南要求产品满足残疾人士的某些需要，要求残疾人能够像非残疾人一样地使用产品。
- SOX（Sarbanes-Oxley法案）这个法案要求审计产品和服务，以防止组织中的经济诈骗。要求软件检查所有事务，并列出怀疑有问题的事务进行分析。这个法案的测试者要帮助最高管理者保存有关经济事务及其确认的记录。
- OFAC与爱国法 这个法案要求审计银行中的事务，以免资金被恐怖分子利用。

认证、标准和符合性测试这些词可以互换使用，术语的混用不会产生问题，只要满足其测试目标。例如，帮助组件满足标准要求的认证代理商可以叫做认证测试，也可以叫做标准测试（例如Open LDAP既是认证也是标准）。

6.5　非功能系统测试

在6.3节中，已经介绍了非功能测试与功能测试之间的差别。非功能测试涵盖的过程与功能测试的过程类似，只是在复杂性、所需知识、所需的工作量和测试用例重复次数等方面有所不同。由于重复执行非功能测试用例需要更多的时间、工作量和资源，因此，非功能测试过程比功能测试有更强的弹性性，以尽可能减少重复。这要求有更严格的进入/退出准则，更好的策划，并事先准备好测试用配置和数据。

6.5.1　设置配置

所有类型的非功能测试都要面临设置配置这个最大的挑战。有两种办法可以完成设置，一种是模拟环境，一种是真实客户环境。由于客户类型、可以使用的资源、建立环境所需的时间等很不相同，建立完全真实的场景是很困难的。即使使用真实客户环境对于这种测试的成功是至关重要的因素，但由于多种复杂性，在难以得到实际配置的场合，还是将模拟环境用于非功能测试。设置配置是一个挑战，这是因为以下原因：

1. 由于环境具有高度多样性和客户的不同，很难预测客户通常将使用什么类型的环境。

2. 通过配置的不同排列组合测试产品效率可能很低，因为客户可能不会使用相同的环境

组合，测试多种组合很耗费人力和时间。不仅如此，由于配置的多样性，要测试的配置会出现组合爆炸。

3. 建立这种环境的成本相当高。

4. 环境的有些组件可能来自竞争公司的产品，可能很难得到。

5. 人员可能没有建立环境的技能。

6. 很难准确预测客户可能使用的数据特性。由于客户所用数据的保密性，这样的信息不会告诉测试团队。

为了建立“接近真实”的环境，需要事先收集有关客户硬件设置、部署信息和测试数据的详细信息。测试数据要根据所给出的样本数据构建。如果是新产品，需要采集类似或相关产品的信息。这类输入有助于建立接近客户环境的测试环境，以便更准确地验证系统的各种质量特性。

6.5.2 提出进入与退出准则

提出进入和退出准则是非功能测试的又一个关键要素。表6-2给出了针对一组参数和各种类型的非功能测试如何确定进入和退出准则的一些例子。满足进入准则是前一测试阶段（即集成测试阶段）的责任，或是由系统测试团队在接受产品进行系统测试前执行的演练测试的目标。

表6-2 非功能测试的典型进入与退出准则

测试类型	参数	样本进入准则	样本退出准则
可扩展性	最大数据	产品可扩展到1万条记录或1000个用户	产品可扩展到1千万条记录或5000个用户
性能测试	• 响应时间 • 吞吐量 • 延迟	查询1000条记录时，响应时间小于3秒	查询10 000条记录时，响应时间小于3秒
可靠性	• 每周失败 • 每个测试周期失败	对1000条记录作24小时的查询时，失败率低于2%	对1000条记录作48小时的查询时，失败率低于0.1%
压力	当压力超过系统极限	在只能接受20个客户的配置上，产品应能容纳25个客户同时登录5小时	在只能接受100个客户的配置上，产品应能容纳100个客户同时登录5小时

6.5.3 平衡关键资源

本节将讨论非功能测试与四种关键资源，即CPU、磁盘、内存和网络的关系问题。这四种资源相互关联，需要完整理解它们之间的关系，以实现非功能测试的策略。

计算机中的这四种资源需要同等重视，因为需要精心平衡以提高产品的质量因素。所有这些资源都是相互依赖的。例如，如果系统中的内存需求满足了，对CPU的需求可能就相应提高了。而这又会随着客户需求的不断增长导致新一轮升级。在发布产品的新版本时对这些资源的需求一般会增长，因为软件变得越来越复杂，软件不仅是针对计算机的，而且还针对设备，例如手机，因此升级资源不再那么容易。

当告知客户需要增加CPU、内存和网络带宽以提高性能、可伸缩性和其他非功能因素时，他们常常对这种变动感到很困惑。产生这种印象，是因为没有提供描述当资源调整后所预期的性能、可伸缩性的合理和可度量的指南。因此，当要求客户升级资源，需要清楚地分析投入所得到的一种重要回报。客户头脑中还会存在的一个问题是，"升级以后我会得到什么？"如果产品经过充分测试，那么对于资源的每次升级，都会伴随产品在非功能方面的改进，因而客户可以理解增加资源的要求。

重要的是理解和承认客户关于产品大量使用的资源和对产品使用非常关键的资源的观点。这样告诉客户很容易，"产品A大量使用CPU，产品B需要更多内存，产品C需要更高的带宽。"等。不过有些运行在特定服务器上的产品可能来自多个供货商，但是却要求运行在同一台计算机上。因此很难要求客户增加服务器上的所有资源，因为所有产品都要求同时运行在同一台服务器上。类似地，多个应用会运行在共享资源的客户机上。

同样重要的是从产品组织的观点分析软件变得越来越复杂，因为使用不同和最新技术对每个新版本都增加了越来越多的特性。除非为产品分配了合适的资源，否则很难看到产品的改进。

因为有很多资源观点，有很多资源之间的关系和对非功能测试的不同需求，测试团队必须对资源作出假设，并在开始测试前得到开发团队和客户的确认。如果没有经过确认的假设，就不可能得出非功能测试的任何合理结论。以下是对资源和配置作出基本假设的一些例子。

1. CPU可以被充分使用，只要当高优先级的作业进入时CPU能够被释放。

2. 可用的内存可以被产品完全使用，只要当另一个作业请求内存时内存可以被释放。

3. 增加CPU或内存的成本并不像以前那样高。因此可以很容易地增加资源，以得到更好的性能，只要能够量化并分析每增加一项资源会得到的好处。

4. 产品会生成很多网络包，只要有可用的网络带宽和延迟，并且成本不高。这种假设中有一点不同，所生成的大多数网络包是在LAN而不是在WAN上传输的。对于WAN和包含多跳的路由器来说，产品生成的网络包需要降低。

5. 产品可以占用更多磁盘空间或完全I/O带宽，只要可以使用。硬盘的成本在不断降低，而I/O带宽则没有降低。

6. 只有优化使用CPU、硬盘、内存和网络，客户在这些资源的投资回报才能最大化。因此，软件需要有能力理解服务器的配置及其使用。

7. 当计算机中的资源还被服务器中的其他活动使用时，可以预期非功能特性会有温和的降低。

8. 同一个产品在不同的配置上有可预测的性能或可伸缩性变化是可接受的。

9. 当调整一些参数后，性能或可伸缩性的变化是可接受的，只要我们知道调整这些可调整参数的影响。

10. 对于不同的配置，例如低端和高端服务器，产品的非功能特性可能表现出不同，只要其与投资回报一致。这实际上会激励客户将其资源升级。

一旦这些样本假设被开发团队和客户确认，就可以开始非功能测试了。

6.5.4　可伸缩性测试

可伸缩性测试的目标是确定产品参数的最大能力。由于要确定最大能力，因此，这种测试需要的资源一般非常多。例如，某个可伸缩性测试用例可能是确定多少个客户机可以同时

登录到服务器上执行某些操作。对于互联网，有些服务会要求对服务器的上百万个访问，因此，尝试模拟这种真实可伸缩性参数是非常困难的，但同时又是非常重要的。

开始执行可伸缩性测试时，关于系统的最大能力可能没有明显线索，因此，要选择高端配置一步步地增加可伸缩性参数，直到达到最大能力。

设计和体系结构会给出理论值，客户提出的需求会提到期望的最大能力，可伸缩性测试首先验证这两者较低的参数。如果客户需求比设计和体系结构所提供的高，则应暂停可伸缩性测试，并对设计进行返工，然后再重新启动可伸缩性测试验证可伸缩性参数。因此，需求、设计和体系结构一同对待测试的参数值提供可伸缩性测试的输入。

与其他类型的参数不同，可伸缩性测试并不在满足需求后就结束。可伸缩性测试要持续进行，直到对于特定配置可伸缩性参数达到最大能力。考虑客户未来需求的高度可伸缩系统会有很长的使用生命周期，再更，每次提出新需求后，就要对产品进行大量重新设计和审查工作，有些稳定的特性由于这些变更也不能正常使用，因此产生质量顾虑。这种产品开发所需的成本和工作量是很大的。

可伸缩性测试中的失效包括系统不能响应或系统崩溃等，但是究竟这些失效是否能够接受，要根据业务目标和目的决定。例如，如果产品不能对100个并发用户作出响应，而其目标是同时支持至少200个用户，那么这就认为是一个失效。如果产品预期只需经受住100个用户，在增加到200个用户时出现问题，那么就认为该测试用例通过，出现的情况是可接受的。

可伸缩性测试有助于确定产品中的主要瓶颈。如果发现资源是瓶颈，那么在确认了6.5.3节提到的假设后可以增加资源。如果瓶颈在产品内部，则需要修改产品。但是，有时基础设施，例如操作系统或技术也可能成为瓶颈。在这种情况下，产品组织应与操作系统和技术提供商交涉解决问题。

可伸缩性测试要在不同的配置上实施，以检查产品的行为。对于每种配置都要采集并分析数据。以下给出了一个数据采集模板。

给定配置
RAM：512MB
缓存区：200MB
用户数：100个（可伸缩参数）
是否有索引：有索引

记录数	开始时间	结束时间	每个记录平均所需时间	所使用的硬盘	所使用的CPU	内存	每秒记录数	服务器配置
0-1百万条记录								
1-10百万条记录								

测试结束后，对收集到模板中的数据进行分析，并采取合适的措施。例如，如果CPU使用达到100%，则可以启用另一台服务器分摊负载，或在原服务器上增加一个CPU。如果结果成功，则可以针对200个或更多用户重复测试，直到确定配置的最大极限。

有些数据需要在更细的粒度上进行分析。例如，如果最大CPU使用达到100%，但是时间很短，如果测试共使时间CPU使用要持续40%，增加一个CPU就没有意思了。但是就出现的CPU短暂使用高峰还是要对产品进行分析，并解决问题。这种测试要求定期采集使用数据放

入以上给出的实例中，通过研究高峰出现的频度，分析产生高峰的原因，会找出瓶颈所在，并确定要采取的措施。因此，仅仅增加资源并不能取得更好的可伸缩性效果。

在可伸缩性测试中，对资源的需求随着可伸缩参数的增长呈指数增长。可伸缩性达到一个特定点后，超过这一点就不可能够改进了（如图6-3所示）。这一点叫做可伸缩性参数的最大能力。尽管还有可用资源，但是产品限制会不允许再扩展。这叫做产品的瓶颈。在测试阶段找出瓶颈并尽早解决，是对继续开始可伸缩性测试的一个基本要求。

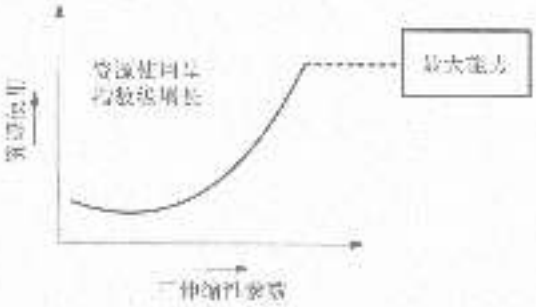


图6-3　用可伸缩性参数检验资源

前面已经解释过，可伸缩性测试可能还需要对资源升级。如果没有产品瓶颈，则需要对资源升级以完成可伸缩性测试。在升级资源时，要进行投资回报研究，以确定所得到的回报与升级资源的成本相比是否值得。以下是在进行这种回报分析时要注意的一些假设。假设中用到的数字和百分比只是指导性的数值，必须结合产品和背景进行修改。

- CPU以最低需求为基准增加一倍时，可伸缩性应该增加50%。以后CPU每增加一倍，可伸缩性应该增加40%，直到增加到制定数量的CPU。这种测试适合大量使用CPU的产品。
- 内存以最低需求为基准增加一倍时，可伸缩性应该增加40%。以后内存每增加一倍，可伸缩性应该增加30%。这种测试适合大量使用内存的产品。
- NIC卡或网络带宽增加一倍时，可伸缩性至少应该增加30%。大量使用网络的产品应进行这种测试。
- I/O带宽增加一倍时，可伸缩性至少应该增加50%。大量使用I/O的产品应进行这种测试。

在可伸缩性测试期间也会遇到瓶颈，需要调谐一定的操作系统参数和产品参数。"打开文件数"和"产品线程数"是可能需要调谐的参数例子。在进行这种调谐时应该作好记录。包含调谐参数和其他产品的推荐值，以及达到可伸缩性数值的环境参数的文档叫做伸缩指南。这种指南是可伸缩性测试必备的文档。

改正可伸缩性缺陷可能会对产品的其他非功能特性产生影响。在这种情况下，可靠性测试（将在下一节讨论）要关注监视参数，例如响应时间、吞吐量等，并采取必要的措施。可伸缩性不能以牺牲其他质量因素的前提下获得。因此建议测试工程师与执行其他功能和非功能测试的人员一起汇总讨论测试结果。

可伸缩性另一个重要方面是相关的完成过程。可伸缩性测试需要大量资源而且开销很大。最好能够在短时间内经过少数几轮测试就能达到可伸缩性要求。如果在可伸缩性测试期间需要对产品、可调谐参数和资源进行大量"调整"，就说明计划不够，对产品行为也不够了解。需要预先仔细研究产品和预计出现的问题及解决方案，才能够得到对这种测试好的体验和感受。

6.5.5　可靠性测试

正如前面定义过的，可靠性测试用于区分产品在给定条件下在给定时间段内或很多轮迭代内，执行所要求功能的能力。可靠性的例子包括持续不断地查询数据库48小时和执行登录操作10 000次。

产品的可靠性不能与可靠性功能混淆起来。生产可靠产品需要很好的技

术、很好的原则、健壮的过程和强有力的管理，以及产品组织内各个角色或功能的一整套活动。产品的可靠性研究生产高质量产品的不同方法，通过关注产品开发和过程的所有阶段使产品没有什么缺陷。这里所说的可靠性是一种无所不包的术语，指产品的所有质量因素和功能。这个观点更多地与产品开发的总体方法有关，与测试的直接联系较少。这种产品可靠性通过关注以下活动实现：

1. **已定义的工程过程** 软件可靠性可以通过遵循清晰定义的过程实现。要求团队要从一开始就理解可靠性需求，并首先关注创建可靠的设计。所有活动（例如设计、编码、测试、文档编制）都经过策划，都要考虑软件的可靠性需求。

2. **在每个阶段评审工作产品** 产品开发生存周期的每个阶段结束时，所产生的工作产品都要经过评审。这可以保证尽早发现错误，一出现错误就立即得到纠正。

3. **变更管理规程** 产品中存在的很多错误都是因为没有很好地对产品变更进行影响分析。实践证明在产品开发生存周期的后期进行变更是很有害的。可能没有足够的时间进行回归测试，因此产品很可能存在由于变更引入的错误。因此，必须清晰地定义变更管理规程，以便交付可靠的软件。

4. **评审测试覆盖率** 为不同阶段和类型的测试分配时间，有助于在产品开发过程中而不是在产品开发出来后发现错误。所有测试活动都要评审时间分配的充分性、测试用例和每类测试所用的工作量。

5. **持续监视产品** 产品交付以后，要主动分析所有可能遗漏的错误。在这种情况下，要针对遗漏的缺陷，就修改产品也修改过程，以可以避免同样类型的缺陷再次出现。

另一方面，可靠性测试是指在一定时间段内持续不断地测试产品。实施好的可靠性测试不保证产品本身是可靠的，因为正如前面已经介绍过的，产品是可靠的还有各种其他需求。可靠性测试交付的只是“经过可靠性测试的产品”，而不是可靠的产品。可靠性测试主要考虑的因素是缺陷。通过测试发现的缺陷被密切监视，要与在前面阶段清除的缺陷进行比较，分析以前没有被发现的原因。跟踪缺陷既要指导产品开发过程，也要指导测试过程，还要确定软件发布的可行性。在项目的一开始就要建立允许最大缺陷数的可靠性准则，与在各种执行测试时间段内发现的实际缺陷数进行比较，比照可靠性准则判定产品的可靠性状况，如图6-4所示。

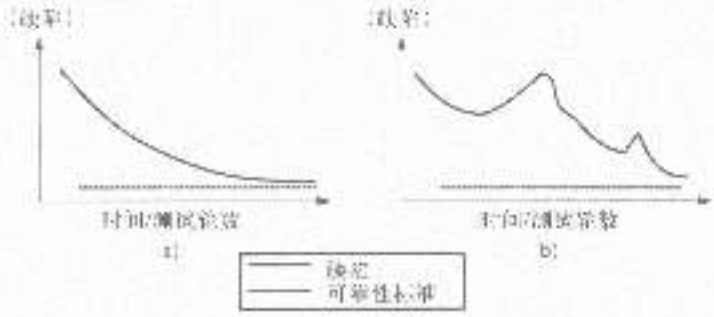


图6-4 可靠性准则示例 a) 平稳进展 b) 不平稳的进展

从图6-4中可以看出，产品最终非常接近满足可靠性准则。图6-4a说明满足可靠性准则的进展是平稳的，而图6-4b进展过程中有在高峰。这些高峰意味着可靠性测试时产品的缺陷时多时少。这可能说明对缺陷的修改在系统中引入了新的缺陷。从过程和产品两个角度分析高峰并采取措施防止高峰出现，有助于以有效和可重复的方式满足可靠性准则。

可靠性不应以牺牲其他质量因素为代价获得。例如，在重复操作时，可能会由于竞争条件而出现失效，解决这个问题可以在语句之间插入“睡眠”。这样做肯定会影响性能，因此，采集和分析可靠性测试数据还应该包括其他质量因素，以便从整体上分析所产生的影响。图6-5给出了一个可靠性影响性能的例子。

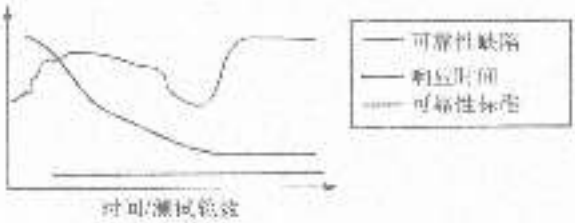


图6-5　可靠性对性能的影响

从图6-5可以看出，虽然可靠性看起来在接近所要求的值，但是响应时间却是不稳定的。有效分析系统测试结果不仅要考虑会聚可靠性准则所要求值的（正面）影响，也要分析为什么性能是不稳定的。可靠性测试发现的错误是由于重复执行某些操作引起的。内存泄漏通常是可靠性测试会发现的一种问题。重复执行操作后，有时CPU可能没有被释放，磁盘和网络活动可能还在继续。因此，采集有关系统在执行可靠性测试之前、之中和之后使用的各种资源数据，并对其进行分析是很重要的。完成了可靠性测试后，产品的资源使用应该降低到测试前的资源使用水平。这可以检验产品使用资源后是否释放。下表说明如何采集可靠性数据。

配置细节

内存：1GB

处理器：2850MHz

网络带宽：100Mbps

系统数据	25个客户	60个客户	100个客户
总迭代次数：			
总失效数：			
峰值内存使用（MB）：			
平均内存使用（MB）：			
峰值CPU使用：			
平均CPU使用：			
任务管理器的包数：			
服务器发送的包数：			

定期采集上表的数据，并绘图分析每个参数的变化情况。将可靠性图表显示的失效数据与可靠性准则进行比较。采集资源数据，例如CPU、内存和网络数据，分析资源对可靠性的影响。

图6-6说明了在可靠性测试执行一段时间后的资源使用情况。

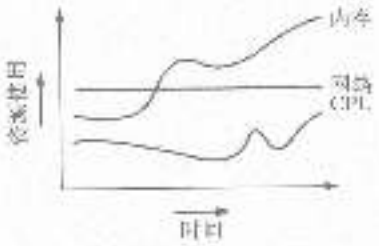


图6-6　可靠性测试执行一段时间后的资源使用情况

图6-6说明在可靠性测试期间，网络的使用是稳定的，而内存和CPU使用却上升。这需要分析，并从根本原因上

解决。CPU和内存使用必须在测试执行期间保持稳定。如果不断上升，就会影响机器上的其他应用程序，该机器甚至会用光内存，出现挂起或崩溃现象，在这种情况下机器需要重新启动。由内存引起的问题对于服务器软件很常见，这类问题需要大量时间和工作来解决。

有多种方法可以通过图表表示可靠性缺陷。

1. 平均失效间隔时间 指产品连续失效之间的平均间隔时间。必须记录产品失效时间的细节，以便理解特定时间扫内的产品可靠性。比方说，如果每72小时失效一次，那么就需要作出适当的提醒，取出当前代码，解决出现的问题，并原样封装。

2. 失效率 是单位时间发生的失效数。失效率是给定单位时间内出现的失效率密度函数（上图给出了这个指标）。

3. 平均发现下K个缺陷的时间 是预测遇到下K个缺陷的平均时间。如果产品不稳定，发现K个缺陷所需的时间就短；产品逐渐稳定后，发现K个缺陷所需的时间就趋于变长。

使用真实场景会得到更准确的可靠性结果。既然可靠性定义为在一定时间内执行的操作，那么就不能保证需要重复所有操作。有些操作，例如配置、备份和恢复很少执行。不应选作可靠性测试操作。应该考虑把大量使用的操作和能够反映客户日常实际活动的操作（场景）组合作为可靠性测试操作。例如，登录—退出操作应该是进行可靠性测试的重要操作。但是实际用户不会不断地连续登录退出，一般中间会插入一些操作。典型情况下，用户会检查一下电子邮件，发送几条即时消息，等。这种活动组合反映了典型的真实客户用法。正是这种活动组合必须纳入可靠性测试中。当多个用户通过不同的客户机使用系统，不断重复一些操作时，可反映服务器端的场景。因此选择可靠性测试的测试用例应该考虑与产品真实使用更接近的场景。

小结一下，"经过可靠性测试的产品"具备以下特征：

1. 重复执行事务操作没有错误，或错误极少。
2. 零宕机。
3. 资源的优化使用。
4. 对于重复执行的事务操作，在给定的时间区间，产品具有一致的性能和响应时间。
5. 重复执行事务操作没有副作用。

6.5.6 压力测试

压力测试用来评价系统超过所提交的需求或资源限制时的情况，保证系统不崩溃。实施压力测试是为了发现在极端条件下或没有必要的资源时产品行为的退化情况。故意制造产品过载情况，模拟资源出现问题，观察产品的行为。期望随着负载的增加产品性能平稳地下降。但是在压力测试期间，系统在任何时刻都不应该崩溃。

压力测试有助于了解系统在极端（不足的内存、不足的硬件）和现实环境中的行为。系统资源耗尽就会出现这种情况。这有助于了解这些测试用例不通过的条件，以便了解像并发用户数、搜索条件、大量事务等方面的最大极限。

也可以模拟像资源不可用这样的极端情况。有些工具可以模拟"挤占内存"，产生巨量通信包挤占网络带宽，创建占用所有CPU周期的进程，不断读写硬盘，等。如果在产品上运行这些工具，可以减少进行压力测试所需的机器数。但是，使用这类工具可能不能发现产品所有与压力有关的问题。因此，在通过工具模拟极限条件进行测试后，最好能够在不使用工具

的情况下重复原来的测试。

压力测试所需的过程、数据采集和分析与可靠性测试的非常接近。唯一的不同是运行测试用例的方式。执行可靠性测试时要保持恒定的负载条件，直到测试用例执行完。而对于压力测试，负载要通过各种方式逐步增加，例如增加客户机、用户和事务处理的数量，直到超过资源被完全使用的情况。随着负载的不断增加，产品会到达一个压力点，有些事务会开始由于资源不可用而失败，超过这一点失效率会增加。为了继续进行压力测试，负载要略微降低到这个压力点以下，观察产品是否会相应地恢复到正常状态。失效率也随之降低，如图6-7所示。这种增加/降低负载操作要执行两三次，以检查产品行为与预期的一致性。

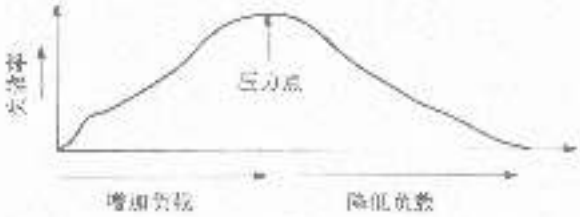


图6-7　负载变化下的压力测试

有时产品不会在降低负载后立即恢复。原因包括：

1. 有些事务可能处于等待队列，这会延迟恢复时间。
2. 有些被拒绝的事务可能需要清除，这会延迟恢复时间。
3. 由于失效，产品可能要执行一些清理操作，这会延迟恢复时间。
4. 有些数据结构可能被破坏，可能不能从压力点恢复。

产品从这些失效迅速恢复所需的时间由MTTR（平均恢复时间）表示。不同的操作对应的恢复时间也可能不同，需要给出计算。前面已经解释过，要在压力点附近做多次测试和不同的操作，并计算平均MTTR，如图6-8所示。

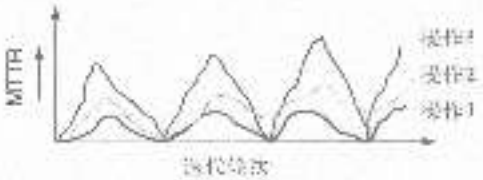


图6-8　不同操作的MTTR

在图6-8中，MTTR随着负载围绕压力点的上升和下降而变化。从图6-8中还可以注意到，每轮操作的操作1恢复时间都是一致的（当负载降低时操作1的恢复时间为0，不同轮次的恢复时间相同），而操作2在第一轮后没有完全恢复，而操作3的恢复时间随着轮次的增加而略有增加。

不是为每个操作都画一张MTTR图，所有操作的平均恢复时间可以画在一张图上。

前面已经提到过，可靠性测试所使用的测试用例、数据采集单和过程也可以用于压力测试。唯一的不同点是负载是可变的。还有就是可靠性测试组合使用测试用例和操作。选择使用资源的不同测试用例在压力测试中使用。因此，在压力测试时要对系统运行很多不同类型的测试用例，但是，在系统上运行产生压力点的测试用例要接近真实场景。以下给出的是选择压力测试的测试用例的一些建议。

1. **重复性测试用例**　重复执行的测试用例可以保证所有数据代码都能按预想的方式运行。

有些操作是由客户反复执行的，设计压力测试时可考虑恰当组合这些操作和事务。

2. 并发性 并发测试用例可保证代码以多路的方式同时执行。设计压力测试时选择由多个用户使用的操作并同时执行。

3. 量级 施加载到产品的负载量要对系统形成压力。可以是一个供很多用户执行的操作，也可能是不同用户执行的操作组合。设计压力测试时要策划并执行可产生负载量的操作。

4. 随机变化 前面已经解释过，压力测试取决于可变负载的增加和降低。在设计压力测试时要选择一部分随机输入（包括用户数量、数据规模）、随机实例和随机量级的方式对系统产生压力的测试用例。

压力测试中遇到的缺陷通常不会在其他任何测试中发现。像内存泄漏这样的缺陷很容易被检测出来，但是由于负载的变化和组合执行不同类型的测试用例，很难对内存泄漏进行分析。因此，压力测试一般在可靠性测试之后进行。为了检测与压力有关的错误，需要重复执行测试用例很多次，使资源使用最大化，并发现重要的错误。压力测试有助于发现像死锁、线程泄漏这样的并发和同步问题。

6.5.7 互操作性测试

互操作性测试用于保证两个或多个产品可以交换和使用信息，并恰当地在一起运行。

系统可以单向（以一个方向交换信息）或双向（以两个方向交换信息）互操作。例如，文本编辑器中的文本可以使用“插入 > 文件”操作选项输出到Microsoft Word应用程序中。但是Microsoft Word中的图片不能输出到文本编辑器中。这是一种单向互操作。电子邮件管理器（Microsoft Outlook）和Microsoft Word之间的信息交换是一种双向互操作，在两个方向上都可以剪裁粘贴信息。

“互操作性”和“集成”这两个词常常互换使用，不过这是错误的。集成是一种方法，而互操作性是术语结果。集成是关于一个产品的，为两个或多个组件定义接口。除非两个或多个产品在设计时就考虑了交换信息，否则是不能实现互操作性的。在集成测试一章已经解释过，各种类型测试之间的界限并不明显，表6-3给出了进一步的解释和具体的背景。

表6-3 不同类型测试的划分

测试描述	测试类型
测试产品组件之间的接口	集成测试
测试两个或多个产品之间的信息交换	互操作性测试
在不同类型设备间，例如操作系统、数据库和网络测试产品	兼容性测试
测试产品新版本生成的目标码或二进制文件是否能在当前版本上使用	后向兼容性测试
测试产品接口是否能在基础设施部件的未来版本上使用	前向兼容性测试
测试产品的API接口是否能在客户开发的软件上使用	API集成测试

互操作性测试对于有医特网背景的应用软件来说更显重要，因为这样的软件具有需要多个计算机和多个软件部件的无缝共存和互操作的特点。因此，越来越多的产品必须具备互操作性，以便能够与几乎所有的操作系统、浏览器、开发工具、编译器、应用软件等进行通信。需要证明产品具备尽可能高等级的互操作性，以便能够与其他系统集成。

互操作性测试还没有实际的标准方法。一个系统对另一个系统的互操作性测试、一个系统对多个系统的互操作性测试和多维互操作性测试是有差别的。遵循技术标准，例如SOAP

（简单对象访问协议）、可扩展标记语言（XML）以及其他W3C（万维网财团），一般会有助于开发使用公共标准和方法的产品。但是符合标准对于互操作性测试还是不够的，因为只有标准并不能保证一致的信息交换和工作流。有些流行产品可能没有实现所有标准，但是由于业务需要，被测产品需要与这些流行产品进行互操作。

以下是改进互操作性测试的一些建议：

1. **跨系统信息流的一致性** 当向产品提供输入时，要被所有系统一致地理解，这样才会向用户返回一种平稳、正确的响应。例如，如果使用数据结构跨系统传递信息，那么这些数据结构的结构和解释就应该在每个系统内保持一致。

2. **按照系统需求修改数据表示** 如果将两个不同系统集成对用户提供响应，第一个系统以特定格式发送的数据必须修改或调整，以适应第二个系统的需求。这有助于请求被当前系统理解，只有在这个时候，才能向用户发送合适的响应。例如，当小尾端机器向大尾端机器传递数据时，字节顺序必须改变。

3. **消息的关联互换与接收合适的应答** 如果某个系统以消息的形式发送一个输入，则第二个系统处于等待模式或监听模式准备接收该输入。如果有多台机器参与信息交换，就会出现死锁、错误响应、死锁或通信延迟。这些问题应该在产品的体系结构和设计中考虑，而不应该留到后期阶段。

4. **通信与消息** 当消息从系统A传递到系统B时，如果消息丢失或损坏，应该测试产品，看其如何对这类错误消息作出响应。产品绝不能崩溃或挂起，应该为用户给出有用的错误信息，要求其等待一段时间，直到恢复连接。由于包含多个产品，因此像“来自远程系统的错误”这样的错误消息会产生误解，没有增加价值。用户不需要知道该消息来自哪里，需要知道的是产生该消息的原因以及必要的改正措施。

5. **满足质量要素** 当两个或多个产品放在一起时，会对相互之间的信息交换产生额外要求。这种需求不能影响单个产品已经满足了的各自质量因素。互操作性测试要从这个角度进行检验。

互操作性的责任更多在于该领域包含的各种产品的体系结构、设计和标准。因此，只有当需求被开发活动，例如体系结构、设计和编码满足后，互操作性测试才能取得更好的结果。互操作性测试应该严格限制在验证信息交换范畴，而不是发现缺陷并逐个修改。

产品之间的互操作性是一种团体责任，需要很多产品组织的努力。所有产品组织都应该协同工作，以满足互操作性目标。有一些关注互操作性标准的标准制定组织，可以帮助产品组织越大限度地减少协同方面的工作。可以辅助定义、实现和验证针对互操作性的标准实现。

6.6　确认测试

确认测试是系统测试之后的一个阶段，通常由客户或客户代表实施。客户定义一组要执行的测试用例，以验证和接受产品。这组测试用例要向客户自己执行，用以在购买产品之前，迅速判定产品的质量。确认测试的测试用例一般数量很少，目的也不是为了发现缺陷。更详细的测试（以发现缺陷为目的的测试）应该在组件、集成和系统测试阶段完成，在将产品交付给客户之前完成。有时由客户和产品组织一起联合开发确认测试的测试用例。在这种情况下，产品组织应该完整理解客户在确认测试时要测试的内容，产品组织要作为系统测试周期的一部分领先执行这些测试用例，以避免在客户执行这些测试用例时产生意外结果。

对于确认测试由产品组织单独完成的情况，执行确认测试要检验产品是否满足在项目的

需求定义阶段定义的确认准则。确认测试的测试用例属于黑盒类型，用于检验一条或多条确认准则。

确认测试也要在近似实际场景下执行。除了检验功能需求外，确认测试还要检验系统的非功能需求。

确认测试的测试用例如果在客户现场没有通过，就会导致客户拒绝产品，会意味着经济损失或耗费人力物力对产品进行返工。

6.6.1 确认准则

确认准则——产品确认

在需求阶段，每个需求都与确认准则关联。有可能一条或多条需求映射形成确认准则（例如，所有高优先级的需求都必须100%地通过），只要需求发生变更，确认准则都要相应地进行修改和维护。

确认测试并不是要执行以前没有执行过的测试用例。因此，要对现有的测试用例进行研究，并对测试用例分类以对应形成确认准则（例如，所有性能测试用例都必须满足所要求的响应时间）。

确认准则还要包含测试对特定法律或合同条款的符合性。对特定法律，例如Sarbanes-Oxley法的符合性测试可以是确认准则的一部分。

确认准则——规程确认

确认准则可以根据交付规程进行定义。规程确认的一个例子是文档和发布介质。这类确认准则的一些例子还包括：

1. 用户文档、管理文档和排错文档应该是发布版本的一部分。
2. 除了二进制代码，产品的源代码包括构造脚本也应该通过CD交付。
3. 在部署之前，要提供至少20人的产品使用培训。

这些规程确认准则的检验和测试都是确认测试的一部分。

确认准则——服务等级约定

服务等级约定（SLA）可以是确认准则的一部分。服务等级约定通常是由客户和产品组织签署的合同的一部分。要选择重要的合同条款作为确认测试的一部分进行检验。例如，在服务等级约定中可能会包含某些解决缺陷的时间限制：

- 部署头二个月内发现的所有重要缺陷应免费修改；
- 系统不能工作的时间不能高于0.1%；
- 从报告时间算起，所有重要缺陷的修改时间不得长于48小时。

对于像上面的这些准则（除了第二条），看起来好像没有什么可测试或检验的。但是这里的确认测试是要保证具有满足这些服务等级约定所需的资源。

6.6.2 选择确认测试的测试用例

前面已经提到过，要从不同测试阶段现有的测试用例集合中选择确认测试的测试用例。这一节给出选择确认测试测试用例的一些建议。

1. 端到端的功能验证 确认测试应该选择涉及产品端到端功能的测试用例。这可以保证所有业务事务都可以作为一个整体进行测试，并且这些事务可以成功地完成。在进行端到端的

产品测试时，应选用实际测试场景。

2. **领域测试用例** 由于确认测试关注的是业务场景，因此要包含产品领域测试。应选择反映业务领域知识的测试用例。

3. **用户场景测试用例** 确认测试要反映真实用户场景的验证，因此，要选择反映用户场景的测试用例。

4. **基本的完备性测试用例** 要选择检验产品基本的已有功能的测试用例。这些测试用例保证系统能够执行所要求的基本操作。对于产品经过变更或修改的情况，更应该关注这些测试用例的选取。需要检验这些现有功能是否能够不间断地持续保持。

5. **新功能** 对于产品经过变更或修改的情况，确认测试还应选择关注检验新功能的测试用例。

6. **少量非功能测试用例** 在确认测试中应包括并执行一些非功能测试用例，以再次验证产品的非功能特性满足要求。

7. **符合法律义务和服务等级约定的测试用例** 选择检验确认测试准则中包含的产品对特定法律义务和服务等级约定符合性的测试用例。

8. **确认测试数据** 确认测试要选择使用客户真实数据的测试用例。

6.6.3 执行确认测试

前面已经介绍过，有时客户自己要执行确认测试。在这种情况下，产品公司的任务就是辅助客户完成确认测试，并解决确认测试中出现的问题。如果确认测试由产品组织完成，则组织确认测试团队就成为最重要的活动。

确认测试由客户或客户代表完成，以检查产品是否已经具备在真实环境中使用的条件。

确认测试团队通常由参与过产品日常使用活动或熟悉用户使用场景的人组成。具有很好客户知识的产品管理层、支持团队和咨询团队，可以参与确认测试的定义和执行。他们可能不熟悉测试过程或软件的技术问题，但是了解产品应该做什么。确认测试团队中90%的成员应该具有所需的产品业务过程知识，10%的成员是技术测试团队的代表。测试团队中需要执行确认测试的成员很少，因为与其他测试阶段相比，确认测试的范围和工作量都不大。

前面已经介绍过，确认测试团队成员可以了解也可以不了解测试或过程。因此，在确认测试之前需要对团队提供产品和过程方面的恰当培训。这种培训也可以向不参与确认测试的客户和其他支持功能的人员提供，因为所花费的工作量是相同的。确认测试团队可以得到从事过开发和测试软件的团队成员的帮助，获取所需的产品知识。可能还会通过提供内部培训材料达到同样的目的。

测试团队成员在确认测试之前和之中的角色是至关重要的，因为需要与确认测试团队成员不断交互。测试团队成员帮助确认团队成员采集所需的测试数据、选择确定测试用例和分析确认测试结果。在确认测试执行期间，确认测试团队定期报告进展，并定期产生缺陷报告。

在确认测试期间报告的缺陷有不同的优先等级。测试团队要协助确认测试团队报告缺陷，使系统不能继续运行和高优先级的缺陷需要在发布软件前修改。如果在确认测试期间发现重要缺陷，就有可能推迟产品发布日期。如果缺陷修改涉及产品范围或需求变更，那么要么推迟产品发布日期以在当前版本提供所需特性，要么推迟到下一个发布版本提供。所有这些缺陷的修改（和不修改）都要与确认测试团队讨论，只有经过他们的认可，确认测试才能结束。

6.7 测试阶段小结

本节要归纳前面已经介绍过的所有测试阶段和测试类型。

6.7.1 多阶段测试模型

本章和前几章讨论了各种测试阶段，如果由不同测试团队执行这些测试阶段的工作，这种模型的有效性就会提高。但是，这种模型的一个大问题是什么时候每个测试阶段该开始和结束。本节提出可以用来启动和结束每个测试阶段的一些建议。每个测试阶段的转换由一组进入和退出准则决定。进入和退出准则的目的是能够并行地实施测试，同时给出产品质量的重要程度，以确定是否能够进行阶段转移。太松或太严的进入准则都有缺点。进入准则太松，极端条件下所有测试阶段都可以同时开展，这时同样的缺陷会被不同的测试团队报告，提高缺陷发现的重复率，造成多个团队等待缺陷的修改。这会导致所发布的产品质量降低，不易跟踪处理问题。在新版本出来时，还会出现各个阶段测试用例重复的情况。实践证明，如果一个测试阶段没有满足质量要求，会影响下一个测试阶段的生产率。太严的进入准则可以避免出现这类问题，但是不具有并行性，会造成产品发布时间推迟。图6-9给出了这两种极端情况。

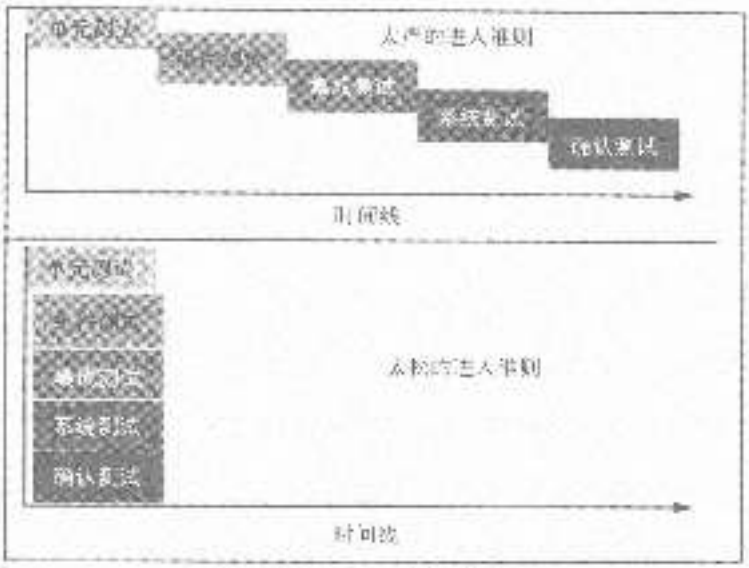


图6-9 进入准则与时间线的关系

正确的方法是让产品质量决定启动某个测试阶段的时间。进入准则应该提供特定测试阶段的质量要求和开展某个特定阶段的最早时机。执行前一个阶段的团队负责满足下一个阶段的进入准则。图6-10说明了这种情况。

表6-4、表6-5和表6-6给出了一些样本进入和退出准则。请注意单元测试没有进入和退出准则，因为单元测试在代码一结束编译就开始，而组件测试的进入准则可以用作单元测试的退出准则。但是，单元测试的回归要一直持续到产品发布。以下给出的准则用于同时确定启动和结束测试阶段的产品质量评判，并使测试阶段能够并行展开。

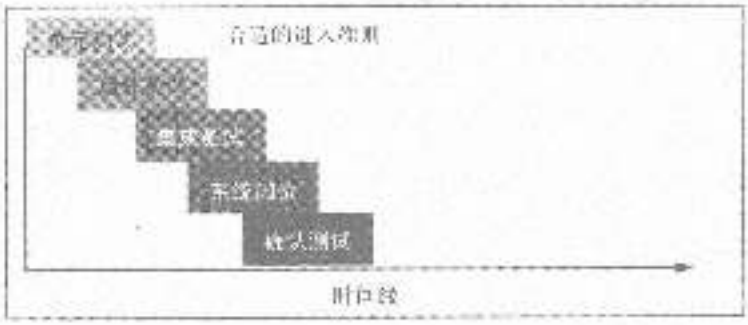


图6-10　综合考虑并行性和质量的进入准则

表6-4　组件测试的样本进入和退出准则

进入准则	退出准则
组件测试 完期单元测试已报告至少70%的完成率	功能中没有关键和严重缺陷
能够提供基本功能，版本稳定（可安装）	执行了100%的组件测试用例，且通过率至少达到98%
已经准备好组件测试用例	完成了组件测试结果报告（定期），根据功能对缺陷趋势和状态进行了分析 完成了组件级性能和负载测试报告，并进行了分析

表6-5　集成测试的样本进入和退出准则

进入准则	退出准则
集成测试 定期组件测试进展报告显示至少有50%的完成率，至少有70%的通过率	功能中没有需要修改的关键和严重缺陷
集成了全部特性，基本稳定（可安装，可启动）	执行了100%的集成测试用例，且通过率至少达到98%
缺陷分析显示缺陷发现趋势在下降	集成测试进展报告显示进展顺利，缺陷分析显示一致的下降趋势 性能和负载测试报告显示所有关键特性都在可接受的范围内 产品处于发布状态（包括文档、介质等）

表6-6　系统和确认测试的样本进入和退出准则

进入准则	退出准则
确认测试 根据定期集成测试结果报告，开始系统测试至少要有50%的通过率，开始确认测试至少要有90%的通过率	执行了100%的系统测试用例，且通过率至少达到98% 执行了100%的确认测试单元，且通过率达到100%
集成了全部特性，版本稳定（产品冻结）	测试总结报告显示所有阶段的测试都完成得很好，并经过分析，缺陷趋势至少已经持续4周
缺陷分析显示缺陷发现趋势在下降	（质量和进展）指标显示产品已经处于待发布状态
没有关键和严重缺陷	性能和负载测试报告覆盖了所有关键特性和整个系统

6.7.2 多个发布版本的处理

正如前面所介绍过的那样，每个测试阶段安排单独的测试团队可以提高有效性，这还为测试团队创造了同时处理多个发布版本的机会，这样测试团队可以充分发挥。例如，组件测试团队满足了退出准则，可以继续进行第二个发布版本的组件测试，而同时集成和系统测试团队则可以关注当前版本的测试。这使得在对当前版本测试的同时，一部分测试团队可以处理下一个发布版本。这是一种缩短各个发布版本占要的总时间，利用测试阶段重叠和并行性的一种方法。图6-11表示了这种概念。图中只考虑了测试团队的一些主要活动，例如自动化、创建测试用例和执行测试用例，以解释概念。

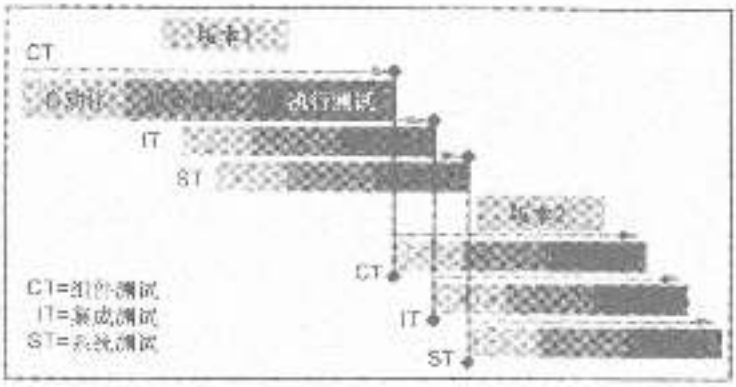


图6-11 利用测试阶段期间的并行性同时处理多个发布版本

6.7.3 谁负责实施与何时实施

表6-7回答了什么时候实施什么测试的问题。

表6-7 什么时候实施什么测试

测试类型	单元测试	组件测试	集成测试	系统和确认测试
静态分析(内存泄漏/代码复杂度分析	★	☆		
国际化	★	☆		
兼容性（向上与向）		★		
本地化测试				★
可靠性			☆	★
API/接口测试			★	
性能测试				★
负载测试				★
可靠性				★
功能/可使用性	☆	★	☆	☆
白盒测试	★	☆		
黑盒测试	☆	★	☆	☆
向后版本兼容测试	★			
相互测试	★			
缺陷再现			★	
场景测试			★	☆
确认测试				★

（续）

测试类型	单元测试	组件测试	集成测试	系统/确认测试
回归测试		★		★
探索式测试	★	★		★
结对测试	★	★		
即兴测试	★	★	★	★
可回验性测试				★

注：★——完全　☆——部分

问题与练习

1. 哪种系统测试适用以下每种情况：
   a. 客户已经给出事务负载和吞吐率需求，需要为客户推荐一种合适的硬件和软件配置。
   b. 产品基于Web，具有很强的季节性使用模式，需要了解产品行为和性能，甚至包括负载远远超过预期使用最大值的情况，以便为未来扩展留下余地。
   c. 产品需要持续运行，功能不能中断。
   d. 有一个在Oracle数据库上运行的特殊报表软件，最近Oracle发布了新版本，需要测试该软件是否能在这个新版本上运行。
   e. 需要度量和发布产品的性能特性指标。
2. 将以下情况按功能和非功能测试分类：
   a. 测试产品的文档，确定是否与产品行为一致。
   b. 验证工资系统满足当地的税收法。
   c. 测试屏幕的用户友好性。
   d. 确定产品的性能。
   e. 保证产品代码覆盖率达到一定百分比。
3. 以下哪些是文中所讲的"产品级测试用例"？如果不是，请说明应该属于哪一级。
   a. 验证定制能够在ERP包之上运行，而ERP包在数据库的一个特定版本上运行。
   b. 在工资系统中，仅对税收计算模块进行测试。
   c. 在数据库包中有不同的算法选项，验证各个算法的行为是否正确。
4. 考虑一个革命性开创产品，采用用户从来没有使用过的全新使用方法。在这种情况下，如何制画"确认测试"？预见在确认测试中会遇到什么挑战？如何应对这些挑战？
5. 假设向客户提出建议："根据测试结果，需要把内存增加一倍、把网络带宽增加三倍。"应该给出怎样的客观数据支持上述建议？如果客户要求通过图表的方式进行说明，说明所作的所有假设，应该怎样做？
6. 最初对产品的测试发现CPU使用存在高峰，但是在一般情况下，CPU使用低于平均水平，需要进一步做什么测试，减少并确定引起CPU使用高峰的原因？
7. 以下哪种测试场景可以用作压力测试：
   a. 用户对数据库作即兴查询。
   b. 学生在咨询日内通过网络登录查询考试成绩。
   c. 客户服务中心的用户录入各种客户的订单、输入信用卡信息、处理投诉等。

8. 在说到互特网时常会提到"e服务"，但是因特网上的"服务"是不同的（比如旅行服务、酒店预定服务等）。请考虑由这些原子服务组成的"更大"服务给测试带来的挑战。例如，请考虑一个综合旅行预定服务，使用（现有）旅行服务的航空公司订票功能和（现有）酒店预定服务的酒店预定功能，可以预见到对这种组合服务的测试会遇到什么挑战？请考虑所有的测试阶段和类型，假设购买综合服务的用户将提供端到端功能的责任放在测试团队的身上。

9. 请考虑某个作为像电视机这样消费品部件的嵌入式软件，需要实施本章介绍的哪种系统测试？什么时候开始实施？与测试比方说传统财务应用软件或数据库、操作系统这样的系统软件相比，会遇到什么挑战？

第7章 性能测试

7.1 引论

在这个因特网的时代，越来越多的业务通过在线方式完成事务处理，人们对所有应用程序都能尽可能快地运行抱有很大期望，这是可以理解的。如果应用程序运行很快，系统就能迅速满足业务需求，还能够扩展业务处理未来需要。由于性能低下而不能对业务事务处理提供服务的系统或产品是产品公司，产品客户和产品客户的客户的一大损失。例如，据统计40%的美国在线消费品销售集中在11－12月。这个时间段出现的系统缓慢或缺乏响应意味着公司要损失数以百万计的美元。再举一个例子，当在因特网上公布考试成绩时，成千上万的人会在很短的时间内访问教育网站。如果网站要用很长时间才能完成查询处理或显示页面的速度很慢，就意味着可能损失业务机会，因为人们会转到其他网站查询成绩。因此，性能是任何产品的基本需求，也迅速成为测试界极为关注的一个主题。

7.2 决定性能测试的要素

决定性能测试的要素有很多。在理解系统测试和测试结果分析方法之前理解这些要素的定义和目的是至关重要的。

上一节已经介绍过，预期产品在给定的时间内处理多个事务，系统或产品处理多个事务的能力由叫做吞吐率的要素决定。吞吐率表示在给定时间内请求数量与产品处理事务的比值。重要的是理解吞吐率（即单位时间产品所服务的事务数）随施加到产品上的负载强弱变化。图7-1给出了在不同负载条件下系统吞吐率的情况。施加到产品上的负载量可以通过增加用户数或增加对产品的并发操作数提高。

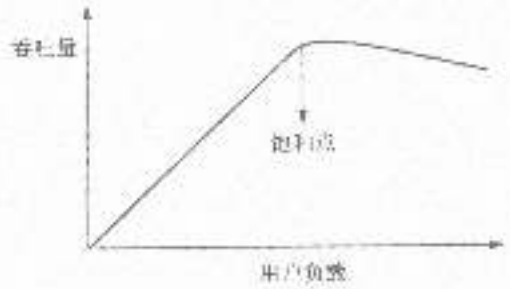


图7-1 不同负载条件的系统吞吐率

从上面的例子可以看出，吞吐率开始随着用户负载的增加而不断上升，这对于很多产品来说是一种理想情况，说明产品能够在更多用户尝试使用产品时完成更多的工作。在图7-1的第二部分，到达一定用户负载条件后（拐点之后），吞吐率开始下降。这时的系统用户会发现系统的响应时间不能令人满意，系统开始用更多的时间完成业务事务处理。“最佳吞吐率”用

饱和点表示，代表产品的最大吞吐率。

吞吐率表示在给定时间区间和给定负载条件下产品可以服务多少个业务事务，确定每个事务的完成时间也同样重要。正如上一节所介绍的，如果这个网站或应用处理具体请求需要的时间更长，客户就可能会去其他网站或应用，因此，度量“响应时间”就成为性能测试的一项重要活动。响应时间可以定义为从请求到产品作出第一个响应之间的延迟。在典型的客户-服务器环境中，吞吐率表示服务器能够处理的事务数，响应时间表示请求和响应之间的延迟。

在现实生活中，并不是所有的请求和应答之间的延迟都是由产品引起的。在网络场景中，网络或共享网络资源的其他产品都会引起延迟。因此，重要的是知道产品引起的是什么延迟，环境引起的是什么延迟。这又引出性能的另一个要素，即反应时间。反应时间是由应用程序，操作系统和环境引起的需要单独计算的延迟。为了解释反应时间，下面举一个Web应用的例子，这个应用通过与由网络连接的Web服务器和数据库服务器通信提供服务，如图7-2所示。

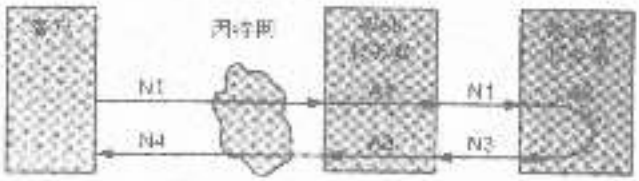


图7-2 各种层次——网络和应用上的反应时间

在上面的例子中，反应时间可以对运行在客户机上的产品进行计算，也可以表示产品可以用于基础设施。对于图7-2，反应时间和响应时间可以计算为：

网络反应时间＝N1－N2－N3＋N4

产品反应时间＝A1－A2－A3

实际响应时间＝网络反应时间＋产品响应时间

以上关于性能中反应时间的讨论很重要，因为对产品的任何改进都只能通过改进A1、A2和A3降低响应时间。如果网络反应时间与产品反应时间相比更重要，且更影响响应时间，那么改进产品性能就没有多少意义。在这种情况下，研究改进网络基础设施是值得的。对于网络反应时间很长且不能再改进，则产品可以使用一些聪智的方法，包括缓存和通过一个包发送多个请求并成批地接收应答。

性能测试评价响应时间、吞吐率和系统的使用情况，执行所要求的功能以时间一产品的不同版本或不同的竞争产品进行比较。

决定性能测试的下一个要素是调谐。调谐是一种通过设置不同的产品、操作系统和其他组件的参数（变量）来改进产品性能的规程。调谐不需要接触产品源代码，就可以改进产品性能。每个产品都可能有可以在运行时间设置的参数或变量，以获得优化的性能。这类产品参数设置的默认值对于具体的部署，不一定总会给出优化性能。因此，需要修改参数或变量值，以适应部署或特定的配置。实施性能测试时，调谐参数是在采集实际数字之前需要完成的一项重要的活动。

性能测试需要考虑的另一个要素是竞争产品的性能。产品性能经过非常充分的改进，如果不能与竞争产品相当也是没有商业意义的。因此，将产品的吞吐率和响应时间与竞争产品进行比较是非常重要的。这种与竞争产品进行比较的性能测试叫做基准测试。没有两个产品

在特性，成本和功能上是相同的，因此确定从哪些参数对两个产品进行比较并不容易。需要精心分析才能列出要在产品之间比较的事务清单，才能进行对等的比较。这类规程的目的是参照竞争对手的产品改进自己产品的性能。

影响性能测试的一个最重要的要素是资源的可用性。需要恰当的配置（包括硬件和软件）导出性能测试和部署的最佳结果。确定要使用的资源和配置的活动叫做能力规划。能力策划的目的是帮助客户在安装或升级产品之前落实硬件和软件资源。这种活动还要确定对于可用的硬件和软件资源客户预期的性能。

归纳一下，性能测试的目的是保证产品：

- 在给定的时间段内处理所需的事务数（吞吐率）；
- 可以在不同的负载条件下运行（可用性）；
- 对于不同的负载条件响应足够快（响应时间）；
- 有不错的资源（包括硬件和软件）投入回报，确定不同的负载条件产品所需的资源种类（能力策划）；
- 在不同参数上与竞争对手的产品相当或更优。

这种对性能测试及其要素的理解有助于确定以下性能测试的定义。

7.3 性能测试的方法论

由于有大量资源和时间需求，性能测试是很复杂、很费钱的。因此，需要精心策划和健壮的方法论。性能测试是模糊的，因为充当各种角色的不同的人都有不同的预期。此外，在性能测试期间发现的很多缺陷可能需要修改设计和体系结构。最后，对性能缺陷的修改甚至会使一些功能出现问题，因此回归需要更多的工作量。所以，本节集中讨论以方法论的方式实施性能测试的各种步骤和指南。性能测试方法论包含以下步骤：

1. 收集需求
2. 编写测试用例
3. 自动化性能测试用例
4. 执行性能测试用例
5. 分析性能测试结果
6. 性能调诊
7. 性能基准测试
8. 向客户推荐合适的配置（能力策划）

7.3.1 收集需求

策划性能测试的第一步是收集需求。在典型情况下，功能测试有明确的输入和输出集，有明确的预期结果定义。而性能测试一般需要详细描述的文档和环境设置，并且可能难以预先很清楚预期结果。由于有这些差别，收集性能测试的需求就成为特有的挑战。

首先，性能测试需求应该是可测试的——并不是所有特性和功能都可以测试性能的。例如，涉及手工干预的特性不能测试性能，因为测试结果取决于用户对产品的输入速度。只能完全自动化的产品执行性能测试。

其次，性能测试需求需要清楚地描述需要度量和改进的要素。正如上一节讨论过的，性

能有多种要素，例如响应时间、反应时间、吞吐率、资源使用等。因此，需求需要把一个或多个要素的度量和改进作为性能测试的一部分。

最后，性能测试需求要与所需的实际与预期改进量或百分比关联起来。例如，如果业务事务，比方说ATM机提取现金，应该在两分钟内完成，那么形成文档的需求就要说明预期的实际响应时间。只有这样才能确定性能测试的通过/不通过状态。如果没有合适参数（响应时间、吞吐率等）的预期数字，就会使性能测试完全失去意义，因为没有对成功的定量度量，最终就不会有什么结论或改进。

天为有以上挑战，所以，关键问题就是如何导出性能测试的需求。导出性能测试有多个来源，包括：

1. 与同一产品的以前版本进行性能比较 性能需求可以类似“ATM机取款事务处理速度将比前一版提高10%”这样的描述。

2. 与竞争产品进行性能比较 性能需求可以类似“ATM机取款事务处理速度将与竞争产品XYZ一样或快”这样的描述。

3. 与根据实际需要导出的绝对数字进行性能比较 性能需求可以类似“ATM机应能每天处理1000个事务，每个事务的处理速度不超过1分钟”这样的描述。

4. 从体系结构和设计中导出的性能数字 产品的体系结构师和软件设计师通常比任何人都能更好地说出产品的预测性能如何。体系结构和软件设计的目标是根据特定负载下预期的性能确定的。因此，可以预期源代码的编写也要满足这些性能要求。

有两类性能测试需求需要关注，即一般需求和特殊需求。一般需求是产品领域内所有产品都共有的需求。这个领域内的所有产品都要达到这些性能预期。有些产品需要满足服务水平约定和标准。装入一个页面所需的时间、对鼠标点击的初始响应时间和在屏幕之间漫游所需的时间都是一般需求的例子。特殊需求取决于具体产品的实现，给定领域内不同产品的特殊需求也是不同的。特殊性能需求的例子包括在ATM机上提取现金所需的时间。性能测试时一般和特殊需求都要测试。

前面介绍过，性能测试还包括负载模式和可用的资源，以及在不同负载条件下产品的预期行为。因此，在将预测响应时间、吞吐率或任何其他性能要素写入文档时，同样重要的是与不同的负载条件对应起来，如表7-1所示。

表7-1 性能测试需求举例

事 务	预期响应时间	负载模式和并发	机器配置
在ATM机提取现金	2秒	最多10 000个用户同时访问	Pentium IV/512MB RAM/宽带网
在ATM机提取现金	40秒	最多10 000个用户同时访问	Pentium IV/512MB RAM/拨号网
在ATM机提取现金	4秒	10 000～20 000个用户同时访问	Pentium IV/512MB RAM/宽带网

任何产品在超过一定的负载后都会呈现一定的性能下降。虽然理解这种现象很容易，但是，如果不知道负载条件下的下降程度会很难实施性能测试。性能下降过大用户是不能接受的。例如，在ATM机提取现金超过一小时才能完成，不管有什么原因或负载条件都是不能接受的。这种情况下，请求提取现金的客户只能等待，也可能离开ATM机，结果现金给了下一个到达ATM机的客户！当负载增加时，可接受的极限性能常用“平稳性能下降”表示。确认产品需要的这种有效平稳下降也是性能测试的一种需求。

7.3.2 编写测试用例

性能测试的下一步是编写测试用例。前面简单讨论过，性能测试的测试用例应该有以下细节：

1. 要测试的操作或业务事务列表；
2. 执行这些操作和事务的步骤；
3. 影响性能测试及其结果的产品和操作系统参数列表；
4. 负载模式；
5. 资源及其配置（网络、硬件、软件配置）；
6. 预期结果（即预期响应时间、吞吐率、反应时间）；
7. 要比较的产品版本或竞争产品及其相关信息，例如对应的字段（上面给出的步骤2～6）。

性能测试用例本质上可以重复执行。这些测试用例通常针对不同的参数值、不同的负载条件等重复执行。因此，针对什么条件重复什么测试等细节，需要在测试用例文档中说明。

在针对不同的负载模式测试产品时，重要的是逐渐增加负载或可伸缩性，以避免在出现失效时做不必要的工作。例如，如果ATM提款对10个并发操作失效，则尝试10 000个并发操作就没有意义。测试10 000个并发操作所需的工作可能是测试10个并发操作的许多倍。因此，系统化的方法应该是逐步增加并发操作数，比方说10、100、1000、10 000地增加，而不是在第一轮尝试10 000个并发操作。测试用例文档应该明确说明这种方法。

性能测试是一种很费时费工的过程，并不是所有操作和业务事务都可以包含到性能测试中，因此，要为性能测试的所有测试用例分配不同的优先级，以便高优先级的测试用例能够优先执行。客户提出的优先级可以采用绝对优先级，性能测试的测试用例也可以采用相对优先级。绝对优先级由需求说明，测试团队通常分配相对优先级。在执行测试用例时，需要考虑绝对和相对优先级，并对测试用例排序。

7.3.3 自动化性能测试用例

在性能测试方法论中，自动化是很重要的一个步骤。由于以下特点，性能测试本身很自然地会要求自动化测试用例。

1. 性能测试需要重复。
2. 性能测试用例如果不自动化就不会是高效的，在大多数情况下，不自动化几乎不能执行性能测试。
3. 性能测试的结果需要精确，手工计算响应时间、吞吐率等可能不够精确。
4. 性能测试要考虑多种要素，这些要素之间有太多的排列组合，如果手工测试这些要素，很难记住并使用所有的排列组合。
5. 性能测试结果和失效的分析需要考虑相关信息，例如以一定时间间隔采集资源的使用、日志文件、跟踪文件等，很难在执行性能测试时手工记录好所有这些相关信息并进行分析。

第16章将要讨论，在性能测试的自动化脚本中不加入任何固定编码的数据。这种固定编码会影响测试用例的可重复性，可能需要更多的时间和精力对自动化脚本进行修改。

性能测试需要端到端的自动化。不仅测试用例本身，而且测试用例所要求的设置、设置不同的参数值、创建不同的负载条件、设置并执行竞争产品的操作和事务，也都需要作为自

动化脚本的一部分。在自动化性能测试用例时，重要的是要使用标准工具和实践。由于有些性能测试用例涉及与竞争产品的比较，测试结果高度敏感，因此测试结果需要一致、可重复和精确。

7.3.4 执行性能测试用例

一般来说，执行性能测试的工作量小于策划、数据采集和分析的工作量。前面已经讨论过，性能测试需要100%的端到端自动化，如果做到这一点，那么执行性能测试用例不过就是调用一定的自动化脚本，但是，执行中通常工作量最大的还是数据采集。在执行性能测试时需要采集与以下有关的数据。

1. 测试用例执行的起止时间。
2. 产品和操作系统的日志和跟踪审计文件（为了以后的调试和可重复）。
3. 定期的资源使用统计（CPU、内存、硬盘、网络使用等）。
4. 所有环境参数的配置（硬件、软件和其他组件）。
5. 在测试用例文档中规定的固定时间间隔的响应时间、吞吐率、反应时间等。

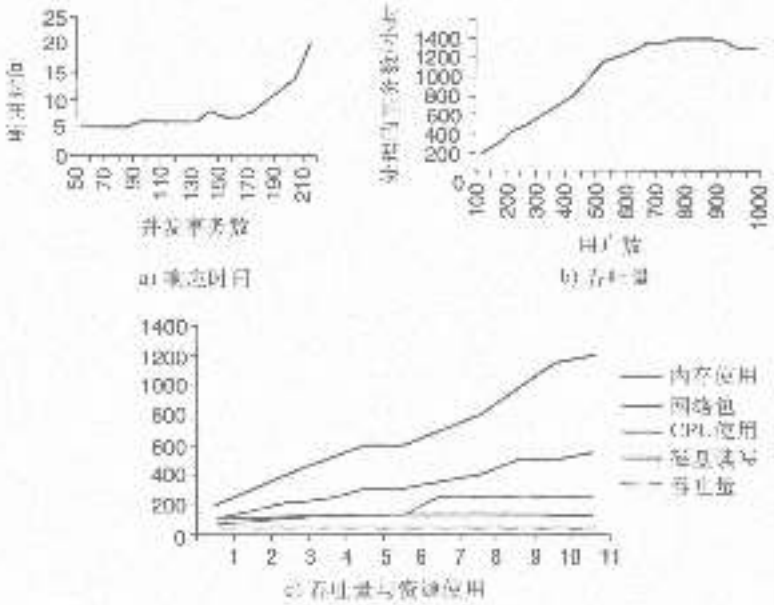
与性能测试执行有关的另一个问题是场景测试。通常由用户执行的一组事务和操作构成性能测试的场景。场景测试要保证不同用户和机器并发使用的各种操作和事务组合能够满足性能准则要求。在现实世界中，并不是所有用户都一直执行同样的操作，因此要进行这样的测试。例如，并不是所有用户都从ATM机提取现金，有些用户要查询账户余额，有些要存钱，等。在这种情况下，要用现有的自动化脚本执行这个场景测试（不同的用户执行不同的事务），并使用现有的工具采集相关的数据。

产品针对硬件和网络环境的不同配置表现出的性能，是在执行中应该考虑的问题。这要求对不同的配置重复执行测试用例，这种测试叫做配置性能测试。配置性能测试保证产品的性能与不同的硬件协调一致，利用这些配置的特性，最大限度地得到最好的性能。对于给定配置，产品要表现出最好的性能，如果配置改善，产品的性能也应随之提高。表7-2给出了配置性能测试的一个例子。性能测试用例针对表7-2中的每一行执行一次，记录并分析诸如响应时间和吞吐率这样的要素。

表7-2 样本配置性能测试

事 务	用户数	系统环境
在ATM机查询账户余额	20	RAM 512 MB，P4双核处理器，操作系统：Windows NT Server
在ATM机提取现金	20	RAM 128 MB，P4单核处理器，操作系统：Windows 98
在ATM机查询用户信息	40	RAM 256 MB，P5四核处理器，操作系统：Windows 2000

一旦执行完性能测试并采集了各种数据点，下一步就是把数据绘成图。前面介绍过，性能测试用例要针对不同的配置和不同的参数值重复执行。因此，将这些测试用例分成组，用图表方式表示是很有意义的。将数据绘成图有助于快速分析，否则对原始数据加工非常困难。图7-3给出了绘制性能数据图表的例子，参见彩图。



a) 响应时间　　b) 吞吐量

c) 吞吐量与资源使用

图7-3　图表绘制举例

7.3.5　分析性能测试结果

分析性能测试结果需要多维思考。这是性能测试中最复杂的工作，绝对需要产品知识、分析技能和统计背景知识。

在分析数据之前，需要对数据进行计算和组织，包括：

1. 计算性能测试结果数据的平均值。
2. 计算标准差。
3. 去除噪声，并重新绘制图表和重新计算平均值和标准差。
4. 由于产品实现的缓存和其他技术，需要将来自缓存区的数据和产品处理的数据区分开来。
5. 区分资源完全可用的情况和背景活动正在运行中的性能数据。

为了公布性能数据，需要满足一个基本预期，即性能数据是能够为客户复现的。为此，需要多次重复所有性能测试，并取这些值的平均值。这样可以增加能够在客户场地上对于同样的配置和负载条件复现性能数据的机会。

可重复性不仅取决于对性能数据取平均值，还取决于产品性能数据的一致性。标准差对此会有帮助。根据标准差可以判定性能数据是否能够在客户场地复现。标准差反应的是数据偏离平均值的程度。例如，如果100人在ATM机上提取现金的平均响应时间为100秒，标准差是2，那么与标准差是30相比，有更大的机会这个性能数据能够复现。标准差接近零表示产品性能具有很高的可复现性，性能值很一致。标准差值越大，产品性能的可变性越大。

如果有一组性能数据来自相同测试用例的多次运行，有可能在几轮重复中，脚本、软件或人员出现错误。取这些执行过程中的错误数据是不妥的，需要舍弃这样的数据。不仅如此，如果在图上表示这些数据，超出范围的一两个值会使图表出现发散，不易进行有意义的分析。可以舍弃这种值，以得到平滑的曲线或图。清除一些不想要的取值的过程叫做噪声清除。从采集数据中清除一部分取值后，要重新计算平均值和标准差。

绝大多数客户—服务器、因特网和数据库应用程序在查询时都会把数据存储在高速缓存区

中，以使再次进行相同查询时能够快速地提供数据。这种方法叫做缓存。要根据结果的来源，来自服务器还是来自缓存又来区分性能数据。这些数据可以保存在两个不同的数据集中，一个对应缓存区，一个对应服务器。把数据保存在两个不同的数据集中，使以后可以根据部署中的预期选中率，对性能数据进行外推。例如，假设缓存区中的数据会产生1000微秒的响应时间，服务器访问占用1微秒，请求的90%可以通过缓存区满足。这样平均响应时间就是：$0.9\times1000+0.1\times1=900.1$微秒。这样平均响应时间就根据权重而不是简单平均算出。

产品的一些"定时启动的活动"或操作系统和网络的后台活动会影响性能数据。一个这种活动的例子是操作系统或编译器内存管理中的垃圾收集和去除碎块活动。当这种活动在后台启动后，就会造成性能下降。确定这些后台事件，将这些数据分离并进行分析，有助于得到正确的性能数据。

组织了数据集（经过前面介绍过的恰当噪声清除和精细化）后就要分析性能数据，归纳以下内容：

1. 当执行多次测试时，产品的性能是否一致。
2. 对于什么类型的配置（包括硬件和软件）和资源可以预期怎样的性能。
3. 什么参数影响性能，能够如何利用这些参数提高性能（请参阅性能调谐一节）。
4. 对于性能要素，包含多个混合操作的场景有什么作用。
5. 像缓存这样的产品技术对性能改进有什么作用（请参阅性能调谐一节）。
6. 负载增加到什么程度性能参数是可接受的，产品性能是否满足"平稳下降"准则。
7. 对于一组负载、资源和参数这类的要素，产品的优化吞吐率和响应时间是多少。
8. 满足了哪些性能需求，与以前的版本或较早确定的预期或竞争产品的比较情况如何。
9. 有时可能不能使用高端配置进行性能测试。在这种情况下，使用通过性能测试能够得到的现有性能数据集和图表，外推或预测高端配置的预期性能参数。

7.3.6 性能调谐

分析性能数据有助于逐步减少对性能结果产生影响的参数，提高性能。一旦参数压缩到少数几个后，要对这些参数的不同取值重复执行性能测试用例，进一步分析其作用，获得更好的性能。这种性能调谐活动需要确定参数和其对性能的贡献方面的很高技能。理解每个参数及其对产品的影响还不足以进行性能调谐，参数的组合也会引起性能变化。各种参数之间的关系及其影响对于性能调谐也是非常重要的。

推进性能调谐包括两个步骤：

1. 调谐产品参数。
2. 调谐操作系统和参数。

与产品关联的有一组参数，产品的管理员或用户可以设置不同的值以获得优化性能。有些通用的实践包括提供一些子过程执行并行事务处理、缓存和内存规模、创建后台活动、推迟例程检查的时间点、对大量使用的操作和事务提供更高的优先级、暂停低优先级操作、改变一组操作的顺序或组合以适应可用的资源等。对这些参数设置不同的取值可提高产品性能。独立和组合的产品参数都对产品性能有影响。因此完成以下工作是很重要的：

1. 针对影响性能的每个参数的不同取值重复性能测试（在改变一个参数时，可能要保持其他参数值不变）。
2. 有时当一个特定参数值被改变时，需要改变其他参数（因为有些参数相互关联）。对参

数组及其不同取值重复性能测试。

3. 对所有参数的默认值重复性能测试（叫做工厂设置测试）。

4. 对每个参数及其组合的高值和低值重复性能测试。

在产品参数调谐时有一点非常重要。性能测试只能对特定的配置和特定的事务得到更好的结果，可以达到较好的性能目标，但是可能会对功能或某些非功能特性产生副作用。因此，调谐对其他情况或场景可能会使生产率下降，需要分析这种调谐产品参数的副作用，这种副作用也应该包含在性能调谐分析活动中。

调谐操作系统参数是获得更好性能的另一步骤。操作系统提供不同类别的各种参数集，使用随操作系统一起提供的工具（例如可以使用regedit.exe编辑修改MS-Windows注册表）可以改变这些取值。操作系统中的这些参数在不同的大类下分组，解释这些参数的影响：

1. 与文件系统有关的参数（例如允许打开的文件数）。

2. 硬盘管理参数（例如硬盘同时读/写）。

3. 内存管理参数（例如虚拟内存页面尺寸和页面数）。

4. 处理器管理参数（例如启用/禁用多处理器环境中的处理器）。

5. 网络参数（例如设置TCP/IP超时）。

前面介绍过，不仅单个参数，而且参数的组合都会对产品性能产生不同的作用。与以前一样，性能测试要对操作系统单个参数和参数组合的不同取值重复性能测试。在重复测试时，需要在完成应用程序/产品的调谐后再调谐操作系统参数。

调谐操作系统参数提高产品性能时还有重要的一点需要注意。在调谐参数的机器上，可能有多个产品和应用程序在运行。因此，调谐操作系统参数可能使被测产品得到更好的结果，但是有可能对运行在同一台机器的其他产品产生严重影响。因此，只有当完全了解操作系统参数对运行在该机器上所有应用程序的影响后才能调谐操作系统参数，否则除非对大幅度提高性能绝对必要，不要调谐操作系统参数。如果调谐操作系统参数只能得到性能少量提高，则不要调谐。

通常有多个平台可以支持产品。因此，性能调谐规程应该考虑操作系统参数及其对支持产品的所有平台的影响。

图7-4给出的图表是性能调谐活动取得结果的例子。在图7-4中，预期结果（性能需求）、调谐前的性能结果（正常线）和调谐后的性能结果（较高的线）画在了一起，有助于分析针对各种可用的资源，例如CPU和内存调谐的效果。这些测试用例也可以对不同负载条件重复执行，再画另外一组性能数据点。

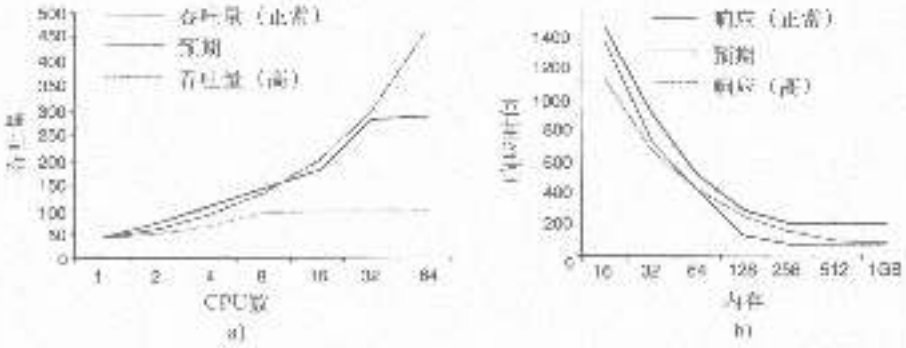


图7-4　性能调谐结果曲线

从图7-4a的吞吐率曲线可以看出，对性能的预期在调谐到32个CPU配置后得到满足，到了64个CPU配置时产品出现问题。从图7-4b的响应时间曲线可以看出，调谐前后都满足预期，只要内存达到128MB或以上。如果不能满足预期，则需要分析并解决存在的问题。参见彩图。

性能调谐结果通常以一种叫做"性能调谐指南"的形式发布，以便客户从性能测试活动中受益。这种指南详细解释每个产品和操作系统参数对性能的作用，并给出一组参数组合的指导值，以及在什么情况下必须调谐什么参数，以及关于错误调谐可能带来问题的警告。

7.3.7 性能基准测试

性能基准测试将产品事务处理的性能与竞争对手的进行对比。没有两个产品能够用同样的体系结构、设计、功能和代码，客户和部署类型也会不同。因此，在这些方面比较两个产品是非常困难的，最终用户事务或场景可以是一种比较方法。一般来说，由独立测试团队或与被比较产品的组织没有关系的独立公司实施性能基准测试，这样可以避免将偏见引入测试。为了使测试成功，实施性能基准测试的人需要被比较产品的专业知识。性能基准测试的步骤是：

1. 确定事务或场景以及测试配置。
2. 比较不同产品的性能。
3. 公正地调谐被比较产品的参数，达到最佳性能。
4. 发布性能基准测试结果。

前面已经介绍过，作为第一步，选择可比较（对等）的事务和场景进行性能基准测试。通常配置细节早就预先确定，因此测试用例不针对配置重复执行。一般来说，所有被比较产品的测试用例要在同一个测试床上执行。但是要考虑用两三套配置完成性能基准测试，以保证测试能够提供覆盖现实场景所需的宽度。

一旦测试执行后，下一步就是比较结果。这时对被比较产品的理解成为关键。完成这种测试的人需要对所有产品具备同样层次的专业知识。各种产品的可调谐参数可能完全不同，理解这些参数及其对性能的影响，对于得到公正的比较结果是非常重要的。经过精心调谐的产品A可能会与没有经过参数调谐的产品B进行比较，以证明产品A的性能比B好。重要的是在性能基准测试中，所有产品的调谐都应该在同一层次上。

从具体产品的视点看，性能基准测试可以有三种结果。第一种结果是正面的，在一组事务和场景上超过竞争对手的产品。第二种结果是中性的，在一组事务上与竞争对手的产品相当。第三种结果是负面的，在一组事务和场景上劣于竞争对手的产品。第三种结果对于产品的成功是有害的，因此要由产品公司在内部使用同样的配置针对这组事务实施上节已经介绍过的性能调谐活动。如果在这种情况下调谐有效，则至少有助于缓解失败危机，否则就要解决性能缺陷，并再次执行性能基准测试测试用例的一个子集。即使在第三种结果时需要重复实施调谐活动，但是并不只限于第三种结果的情况，性能调谐可以对正面、中性和负面所有情况重复实施，以得到最佳的性能结果。重复性能调谐并不总是可能的。如果中立代理参与，他们可能只进行对等比较测试，可能不进行调谐。在这种情况下，测试团队需要重复执行测试用例。

要发布性能基准测试的结果，有两种类型的发布。一种是内部发布，只对产品团队秘密发布，包括以上提到的所有三种结果，以及建议采取的措施。性能基准测试的正面结果通常作为间接市场开发手段发布，用作产品的销售工具。由独立公司得出的基准也作为比对对象发布。

7.3.8 能力策划

如果要对多种配置实施性能测试，那么有大量数据和分析结果可以用来预测特定一组事务和负载模式所需的配置。这种逆向过程就是能力策划的目标。性能配置测试要针对不同的配置实施并获得性能数据。在能力策划中，性能需求和性能测试结果是输入需求，并导出满足这组需求所需的配置。

能力策划需要清楚地理解事务和场景对应的资源需求。产品有些与一定负载条件关联的事务包括大量使用硬盘，有些大量使用CPU，有些大量使用网络，还有些大量使用内存。有些事务可能需要这些条件的组合才能得到较好的性能。这种对每种事务需要什么资源的理解是实施能力策划的前提。

如果能力策划要确定事务和特定负载模式的合适配置，那么下一个问题就是如何确定负载模式。负载可以是客户当前（短期）需要的实际需求，也可能是下几个月（中期）的需求，也可能是未来几年（长期）的需求。由于负载模式随未来需求变化，因此在做能力策划时考虑到这些需求是至关重要的。对应短期、中期和长期需求的能力策划分别是指：

1. 最低要求配置
2. 典型配置
3. 特殊配置

最低要求配置表示低于这种配置，产品甚至都不能运行。因此，一般不支持低于最低要求配置的配置。典型配置表示在这种配置下，产品能够不错地运行，满足所要求负载模式的性能需求，并且可以应对该负载模式略有加强的情况。特殊配置表示能力策划结论是在考虑了未来所有需求的情况下得出的。

在能力策划中，有两种技术起主要作用，即负载平衡和高可用性。负载平衡保证能够平等使用多台机器，为事务提供服务。这样，可以通过增加更多的计算机，产品可以承担更多的负载。计算机簇可以用来提供可用性。在计算机簇中，在多台计算机之间共享数据，使得一台计算机出现问题时，事务可以由计算机簇内的另一台计算机处理。在进行能力策划时，要考虑负载平衡和可用性要素，以描述所需的配置。

绝大多数能力策划活动只是对数据作解释，并就已有信息进行外推。对性能测试结果分析或外推中的微小错误都会导致产品部署后实际使用预期的偏差。不仅如此，能力策划的基础是在实验室采集的性能测试数据，这只是模拟环境，在实际部署中，还有一些其他参数会影响产品性能。由于有这些不可见的因素，除了前面提到的技能，完成能力策划还需要真实世界数据和使用模式方面的经验。

7.4 性能测试工具

有两类工具可以用于性能测试，一类是功能性能工具，一类是负载测试工具。

功能性能工具用于记录并回放事务，获取性能数据。这类测试一般需要很少的计算机。

负载测试工具为性能测试模拟负载条件，不需要使用很多用户或计算机。负载测试工具简化了创建负载的复杂性。如果没有这类负载测试工具，可能就不能实施这类测试。正如前面介绍过的，这只是模拟的负载，现实情况可能与模拟情况有出入。

以下给出一些流行的性能测试工具：

- 功能性能工具

- Mercury公司的WinRunner
- Compuware公司的QA Partner
- Segue公司的Silktest

• 负载测试工具
- Mercury公司的Load Runner
- Compuware公司的QA Load
- Segue公司的Silk Performer

有很多销售这些性能工具的供应商。本书最后的参考文献给出了一些流行的工具。

性能和负载工具只能帮助获得性能数据。资源的使用是另一个需要采集的参数。Windows的任务管理器和Linux的top都是可以帮助采集资源使用信息的工具。今天的几乎所有操作系统都提供能够采集网络数据的网络性能监视工具。

7.5 性能测试的过程

性能测试的过程与所有其他测试类型的过程一样，唯一的差别是采集的信息详细和更多的分析。前面已经介绍过，性能测试需要的工作量更大，一般需要重复测试多次。所增加的工作量反映成本的增加，因为性能测试所需的资源相当多。性能测试的一个主要挑战是使用合适的过程以最大程度地降低工作量。图7-5试图通过一个简单的性能测试过程说明这个问题。

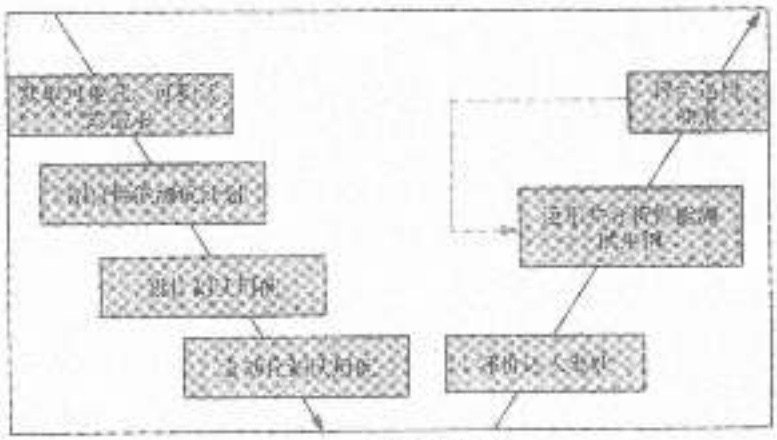


图7-5 性能测试过程

不断变化的性能需求是对产品的一种严重威胁，因为修改代码只能在一定程度上提高性能。前面已经介绍过，绝大多数性能问题需要修改体系结构和设计，或排例需求。因此，重要的是在生存周期的早期采集并传达性能需求，因为在生存周期的后期修改体系结构和设计代价极高。在采集性能测试的需求时，重要的是要确定其是否可测试，也就是说，要保证性能需求是量化的，能够客观地确认。如果是这样，就将量化了的性能预期写入文档，使需求可测试、可度量是性能测试成功所需的第一个活动。

性能测试过程的下一个步骤是编写测试计划。测试计划需要包含以下内容：

1. 资源需求 需要将性能测试所需的特殊额外资源列入计划并落实。通常，先获取这些资源，在性能测试中使用，在性能测试结束后释放这些资源。因此，资源应该是策划工作的一部分，并进行跟踪。

2. 建立测试床（模拟的和真实的）。测试实验室 要在执行之前建立具有所有所需设备和软件配置的测试实验室，性能测试需要大量资源和特殊配置。因此，建立模拟和真实的环境是很耗费时间的，建立测试床中出现的错误可能意味着整个性能测试都得重新进行。因此，建立测试床必须是策划工作的一部分，并进行跟踪。

3. 责任 前面说过，性能缺陷可能会造成体系结构、设计和代码的修改。此外，面对客户的团队一般需要就性能进行沟通。性能测试执行的成功涉及多个团队，如果要满足性能目标，所有团队和不同岗位的人员要一起工作。因此，必须将编制责任矩阵作为性能测试计划的一部分，并对全体团队进行宣传和提倡。

4. 建立（内部和外部的）产品跟踪和审计 性能测试结果要形成跟踪记录，以便分析测试结果和缺陷。要预先策划应跟踪的内容，并作为测试计划的一部分。这些工作要预先完成，因为跟踪的内容过多可能会开始影响性能结果。

5. 进入和退出准则 由于性能测试有复杂性和准确性要求，因此需要稳定的产品，对产品的变更会影响性能数据，可能意味着必须重新进行测试。在产品稳定之前或还在修改当中，执行性能测试用例效率很低，因此，性能测试的执行一般要在产品满足一组准则之后进行。这组要满足的准则作为性能测试计划的一部分，预先定义并形成文档。类似地，还要定义一组退出准则，以总结性能测试结果。

设计和自动化测试用例是性能测试过程的下一个步骤。自动化问题值得单独讨论，因为如果没有自动化就几乎不能执行性能测试。

进入和退出准则在性能测试执行过程中有重要作用，在产品开发期间，要定期评估进入准则，如果达到准则要求就启动测试。每个性能测试用例会有独立的准则，需要定期评估进入准则，因为启动测试过早效率很低，启动过晚可能意味着在发布之前难以达到性能目标。性能测试执行结束时，要对产品进行评估，看其是否满足所有退出准则。如果有些准则没有满足，则要对产品进行改进，重新执行对应退出准则的测试用例，目标是弥补差距。这个过程反复进行，直到所有退出准则都得到满足。

由于涉及的要素（即成本、工作量、时间和有效性）都很重要，因此，以上介绍的每个性能测试过程步骤都很关键。保持强有力的性能测试过程能获得很高的回报。

7.6 挑战

性能测试在测试界是一个没有被很好理解的课题，对于性能测试有多种解释，有些公司将性能测试与负载测试分开，在不同的测试阶段实施，在有些情况下这样做可能会取得成功，但是有时将其分开会引起复杂化。当需要与负载测试数据比较功能性能数据时，由于所用的版本不同，且由于在两个不同的阶段实施测试造成时间过度的不同（由于时间进度的不同，产品的质量也会不同），完成这种比较是很困难的。在这种情况下，进行对等比较是不可能的。

是否具备所需的技能是性能测试面临的一个重要问题。本章多次提到，产品知识、竞争对手产品的知识、工具的使用、测试用例自动化、过程、统计知识和分析技能都是实施性能测试所需要的。到目前为止所介绍的任何测试类型所需的技能都没有这样多。工程师在这些技能上着眼于性能测试长远发展的培训，有助于满足技能方面的要求。

性能测试需要大量资源，例如硬件、软件、工作量、时间、工具和人员。这大公司也会发现缺少满足性能测试目标需要的资源，即使有这些资源，也只是短时间的。这是性能测试的另一个挑战。研究可用的资源，努力满足尽可能多的性能测试目标是执行性能测试的团队

所期望的。

性能测试结果需要反映现实环境和预期。由于工具只能模拟环境，建立能够在受控状态下运行的测试实验室，创建如客户所做的那样并不是所有字段都有内容的数据包，在真实的客户部署中复现性能测试结果，是很大的挑战。创建与客户部署接近的测试床是性能测试另一项预期。

选择合适的工具进行性能测试是另一个挑战。现在有很多性能测试工具，但并不是所有工具都满足所有需求。不仅如此，性能测试工具很昂贵，并且需要额外的资源安装和使用。性能测试工具还要求测试工程师学习额外的元语言和脚本，这对性能测试又提出一个挑战。

性能测试的另一个挑战是包括客户在内的不同团队的相互接口，不仅客户，技术人员和开发团队也会提出性能测试需求。实施性能测试是为了满足客户、体系结构设计师和开发团队的预期。从商业意义看，产品性能需要赶上竞争对手的产品性能。随着预期从各个方向增长，很难一次满足所有预期。如果要满足其绝大多数性能预期，需要持续努力。

管理层和开发团队对性能测试不够重视是另一个挑战。产品的所有功能都正常运行后，大家就认为产品已经可以交付了。由于已经介绍过的各种原因，性能测试需要在功能稳定以后实施。性能测试发现的缺陷需要管理层非常认真地进行研究。由于已经太迟，不能修改其中的一些缺陷，或由于发布时间的压力，或需要对设计和体系结构进行修改使回归测试工作量增大，或其他各种原因，一般一些性能缺陷会推迟到后续版本解决。这与性能测试的目的背道而驰。成功执行性能测试需要得到高层管理对在产品发布之前修改性能缺陷的承诺和指示。

问题与练习

1. 在以下情况中，哪种性能要素可能最重要？
   a. 每天白天下班后银行要处理所有事务，夜里完成现金对账。
   b. 到银行的用户应该在5分钟内完成其事务处理。
   c. 用户通过没有处理能力的客户机访问远程文件系统，并输入一个文件名。由于处理是在远程服务器上完成的，用户输入的每个字符都要发送给服务器并返回过来。
2. 每个产品都有不同的调谐参数。对于以下每种情况，请确定重要的调谐参数，应该从哪种文档/资源找出这些参数？
   a. 操作系统（例如Windows XP）
   b. 数据库（例如Oracle）
   c. 网卡（例如无线LAN卡）
3. 采集性能测试需求与采集黑盒功能测试需求有什么不同？通过所使用的资源、方法、工具和所需的技能区分。
4. 软件大多数类型产品像数据库、网络等，都有标准的性能基准。例如，TPCB基准规定了事务率。请查询因特网，就各种类型的数据库事务、编译器和其他自己感兴趣软件的行业标准基准整理一份报告。
5. 市场上有一些性能测试自动化工具。从自己的公司角度考虑，请准备一份针对自己对性能测试自动化工具预期的检查单。使用这份检查单对各种可以得到的自动化工具进行比较。
6. “人工进行性能测试是最困难的工作”——请根据所需的技能、态度和其他方面对这句话作出判断。

第8章 回归测试

8.1 回归测试的定义

医生：恭喜你！给你造成痛苦并影响你消化的胃溃疡现在已经彻底治愈了！

病人：好的，医生。可是我的口腔溃疡太严重了，什么都不能吃，所以还是什么也消化不了。

软件在不断修改，这些修改是必要的，因为要改正缺陷、要增强现有功能、要增加新的功能。任何时候做这种修改时，重要的是要保证：

1. 按照设计进行修改或补充；
2. 修改或补充不能对已经有效的内容造成破坏，而且继续有效。

回归测试就是要达到以上两个目的。首先举一个简单的例子进行说明。

> 回归测试要保证增强型或改正型修改使软件正常运行，并且不影响已有的功能。

假设某个产品的给定版本中有三个缺陷D1、D2和D3。报告这些缺陷后，开发团队将修改这些缺陷，测试团队将进行测试以保证这些缺陷确实已经修改。当客户开始使用该产品（经过修改以消除缺陷D1、D2和D3）时，可能会遇到新的缺陷D4和D5。开发和测试团队再次进行修改、并测试这些新缺陷的修改情况。但是，在修改D4和D5过程中，由于无意识的副作用，D1可能又出现了。因此，测试团队应该不仅保证修改需要修改的缺陷，还要关注不能影响其他所有已经有效的内容。

回归测试使测试团队能够满足这个目标。回归测试在今天是很重要的，因为软件的发布非常频繁，以满足竞争的要求，提高客户的认可度。迅速和频繁地发布并交付稳定的软件是对软件公司最基本的要求。回归测试要保证任何引入现有产品的新特性不会反过来影响当前的功能。

回归测试采用有选择性的重测技术。每次完成对缺陷的修改，测试团队都要选择一组需要运行以检验缺陷修改情况的测试用例。要进行影响分析，以确定由于这些缺陷修改会受影响的区域。根据影响分析，再选择更多的测试用例来关注受影响的区域。由于这种测试技术关注已经执行过的现有测试用例的重用，因此叫做有选择性的重测。有时还可能需要开发新的测试用例，以关注一些受影响的区域。但是，回归测试主要还是重用已有的测试用例，因为回归测试关注测试已经可用并至少已经测试过一次的特性。

8.2 回归测试的类型

在介绍回归测试的类型之前，首先要理解“版本”的含义。当内部或外部测试团队或客户开始使用产品时会报告发现的缺陷。这些缺陷要经过每个负责修改缺陷的开发人员的分析，然后开发人员再进行单元测试，并将缺陷修改装入配置管理（CM）系统。然后对完整产品的

源代码进行编译，这些缺陷修改以及现有特性都纳入版本中。因此，版本就是产品所提供的所有缺陷修改和特性的集合。

实践中有两种类型的回归测试：

1. 常规回归测试

2. 最终回归测试

常规回归测试在测试周期之间进行，保证缺陷修改已经完成，以前测试周期已经有效的功能继续有效。常规回归测试可以对多个产品版本执行测试用例。

"最终回归测试"要在发布前确认最终的版本，配置管理工程师提交要交付给客户的版本，包括介质和其他内容。最终回归测试周期要在开发和测试团队共同约定的具体时间段内进行，这叫做回归测试的"烹饪时间"。烹饪时间对于持续测试产品一段时间是必要的，因为有些缺陷（例如内存泄漏）只有在用过一段时间后才能发现。产品要在烹饪时间内持续运行，以保证这类与时间有关的缺陷能够被发现。有些测试用例重复执行，以发现就要交付给客户的最终产品是否存在缺陷。发布版的所有缺陷修改都应该在最终回归测试周期版本中完成。最终回归测试比任何其他类型或阶段的测试都关键，因为这是保证到达客户的是经过测试的版本的唯一测试。

以上讨论的回归测试类型可用图8-1说明。彩图中的回归1和回归2是常规回归测试周期，最终回归测试用最终回归表示。

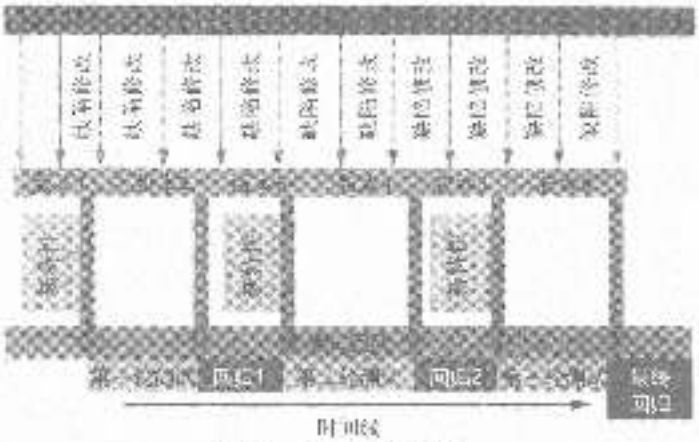


图8-1 回归测试的类型

8.3 回归测试的时机

只要对软件作了修改就要进行回归测试，以确保这些修改不会反过来影响现有的功能。常规回归测试可以对多个版本运行测试用例，但是，我们还是强烈建议对没有变更的版本作最终回归测试。由于缺陷而没有通过的测试用例应该纳入以后的回归测试。

回归测试在测试周期之间进行，以确定所提交的软件是否与以前收到的版本一样或更好。由于测试需要大量资源（硬件、软件和人员），因此需要快速测试，以评估版本的质量和对软件的变更。最初只在回归测试中投入少数几个人和很少的机器，这样可以避免在缺陷修改或版本处理时，影响到现有功能，造成质量下降的情况或工作量的浪费。这种副作用需要在大量人员投入测试之前立即得到纠正。在任何时候出现这种需求时就要进行回归测试。

当出现以下情况时要实施回归测试：

1. 相当多的初始测试已经完成。

2. 已经修改了大量缺陷。

3. 关注了能够产生副作用的缺陷修改。

回归测试也可以作为一种主动方法定期实施。

正如第15章将要介绍的那样，使用缺陷跟踪系统在所有有关人员之间就需求修改的状态进行沟通。当开发人员修改一个缺陷后，该缺陷就会通过缺陷跟踪系统返回给测试工程师进行验证。如果缺陷被修改，测试工程师需要采取恰当的措施关闭该缺陷。如果还没有被恰当地修改，就把它重新打开。在这个过程中，可能忽略副作用，即新版本修改好了特定的缺陷，但是有些以前正常的功能现在却不能用了。进行了一批缺陷修改后就要进行回归测试。为了确保没有副作用，回归测试周期还要选择更多的测试用例验证缺陷修改情况。因此，在测试人员能够关闭修改的缺陷之前，重要的是保证执行合适的回归测试用例，并且修改没有产生副作用。启动回归测试验证缺陷修改永远都是一个好的实践，否则，如果以后通过测试发现有副作用或发现功能损失，就很难确定是哪个缺陷修改导致的。

不管产品处于哪个测试阶段都可以实施回归测试。

从以上讨论可以清晰地看出，回归测试既是一种有计划的测试活动，也是一种基于需要的活动，并且要在各个版本和测试周期之间进行。因此，回归测试适用于软件开发生存周期（SDLC）的所有阶段，也适用于组件、集成、系统和确认测试。

图8-2归纳了本节讨论的内容。

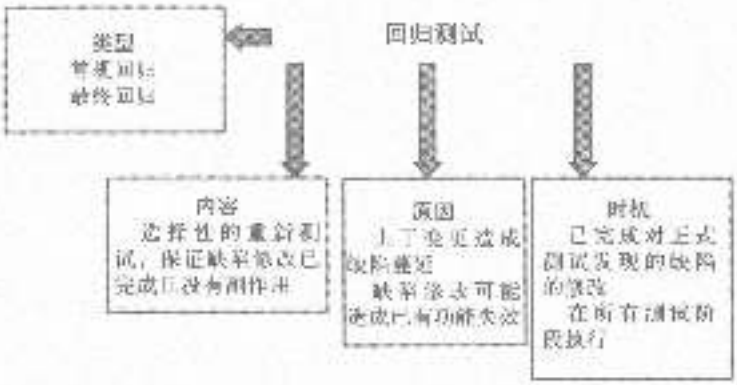


图8-2　回归测试小结

8.4　回归测试的方法

回归测试定义完备的方法论是非常重要的，因为回归测试是最后一种测试，通常最后在发布前实施。如果回归测试做得不好，就会使缺陷漏掉，可能会使客户面临测试团队没有发现的严重问题。

回归失效要在周期的较晚期才能发现或被客户发现。回归测试定义完备的方法论能够避免这种情况出现。

有一些回归测试方法论正被不同的公司使用。本节的目的是解释一种包含了绝大多数方法论内容的方法论。这里介绍的方法论包含以下步骤。

1. 实施第一次“冒烟”或“完备性”测试。

2. 理解选择测试用例的准则。
3. 划分测试用例的优先级。
4. 选择测试用例的方法论。
5. 重新设置测试用例，以进行回归测试。
6. 总结回归测试的结果。

8.4.1 实施第一次"冒烟"或"揉底"测试

只要对软件作了修改，首先就要保证没有破坏什么东西。例如，如果在构建一个数据库，那么该数据库的任何版本都应该能够启动，执行基本的操作，比如查询、数据定义、数据处理，并可以关闭数据库。此外，可能还需要保证与其他产品的关键接口也能正常工作。这些都是应该在对产品进行更详细的测试之前要做的。例如，如果给定版本不能启动数据库，则这个版本毫无用处。必须首先修改代码，解决这个问题，然后才能考虑测试其他功能。

冒烟测试包括：

1. 确定产品必须满足的基本功能；
2. 设计测试用例，确保这些基本功能是有效的，并将其装入冒烟测试包；
3. 保证每次构建产品时，都能在运行其他测试之前成功地运行这个测试包；
4. 如果测试包没有通过，请开发人员确定该变更，并可能修改或将该变更回退到冒烟测试包成功的状态。

要保证最先检测冒烟测试，有些公司要求任何时候开发人员作了修改，都必须在将该版本纳入配置管理库之前，对该版本成功地运行冒烟测试包。

产品中的缺陷不仅可以通过代码引入，也有可能通过用于编译和连接程序的构建脚本引入。冒烟测试可以发现由构建规程引入的这类错误。这个工作是很重要的，因为研究表明，有15%的缺陷是由配置管理或与构建有关的规程引入的，如图8-3所示。

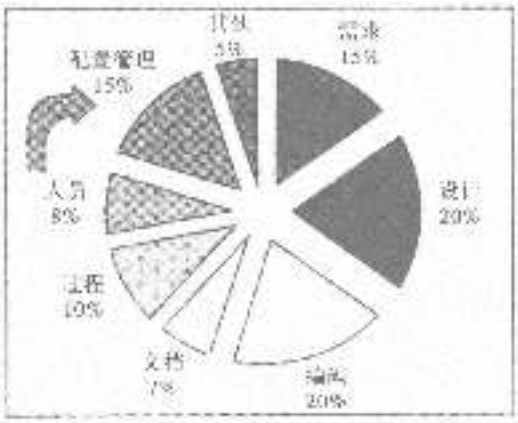


图8-3 配置管理缺陷

8.4.2 理解选择测试用例的准则

执行了冒烟测试以后，产品就值得进一步详细测试了。现在的问题是，应该运行什么测试用例，以达到保证修改有效和没有引起无意识的副作用这双重目的。

选择回归测试的测试用例有两种办法。首先，公司可以选择一组不变的回归测试用例，对每个版本和变更运行。在这种情况下，确定要运行的测试用例很简单。但是这种方法很可能不是最优的，这是因为：

1. 为了覆盖所有修改，不变测试用例集将包含所有特性，可能每次都要执行一些没有要求的测试用例；

2. 缺陷修改或变更的一个给定集合可能会引入的问题，在不变测试用例集中可能还没有现成的测试用例能够检测。因此，即使运行了全部回归测试用例，所引入的缺陷会依然存在。

第二种方法是通过评判测试用例，为每个版本动态地选择测试用例。选择回归测试的测试用例需要具备以下知识：

1. 对当前版本所作的缺陷修改和变更；
2. 测试当前变更的方法；
3. 当前变更对系统其他部件可能产生的影响；
4. 测试其他受影响部件的方法。

以下是选择回归测试测试用例的一些准则：

1. 包含以前发现缺陷最多的测试用例。
2. 包含针对修改功能的测试用例。
3. 包含上次发现问题的测试用例。
4. 包含测试客户要求必须提供的产品基本功能或核心特性的测试用例。
5. 包含测试应用软件或产品的端到端行为的测试用例。
6. 包含测试正面测试条件的测试用例。
7. 包含用户能够大量直接接触的区域。

在选择测试用例时，不要过多选择肯定不会通过和对缺陷修改没有多少关系的测试用例。对最终回归测试周期要更多地选择正面测试用例，而不是负面测试用例。选择负面测试用例——即故意使系统崩溃的测试用例——在确定失效根源上会产生一些困惑。这里还建议回归测试前的常规测试应包含正面和负面测试用例。

回归测试应更多地关注缺陷修改的影响，而不是缺陷本身的危险性。

选择回归测试的测试用例更多地取决于缺陷修改的影响，而不是缺陷本身的危险性。轻微缺陷可能导致严重副作用，对严重缺陷的修改可能没有或有很小的副作用。因此，测试工程师在选择回归测试的测试用例时需要综合考虑这些问题。

选择回归测试的测试用例是一种持续的过程，每次要执行一批回归测试用例（又叫做回归测试集）时，都要根据以上条件对测试用例就其适用性进行评估。

8.4.3　测试用例分类

每次回归测试动态地选择了测试用例后，从项目一开始，甚至在测试周期之前就开始策划回归测试是很值得的。为了对一轮回归测试选择合适的测试用例，可以根据其重要性和客户使用情况，把测试用例按优先级分类，如图8-4所示。

对于成功的测试执行，了解测试用例的相对优先级是很重要的。

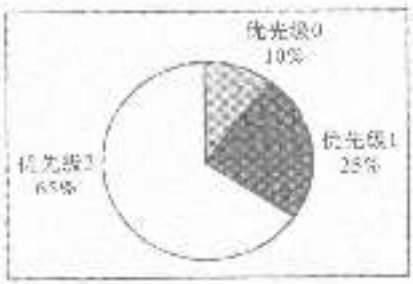


图8-4　测试用例的分类——一个例子

- **优先级0** 这些测试用例可以当做摸底测试用例，用来检查基本功能和进行进一步测试。当对产品做了重大变更后也要执行这些测试用例。这些测试用例对于产品开发团队和客户都具有很高的项目价值。
- **优先级1** 利用基本的和正常的配置环境，对于产品开发团队和客户都具有高的项目价值。
- **优先级2** 这些测试用例具有中等项目价值。作为测试周期的一部分执行，并根据需要为回归测试选用。

8.4.4 选择测试用例的方法论

把测试用例划分为不同的优先级后，就可以选择测试用例了。有多种回归测试的合适方法，需要具体情况具体分析。业界已有多种选择回归测试用例的方法论。本节介绍的方法论要在按上节讨论的把测试用例划分为多个优先级后，考虑缺陷修改的关键性和影响程度。

情况1 如果缺陷修改的关键性和影响程度很低，那么测试工程师从测试用例数据库（TCDB）中选择少量测试用例并执行就足够了。（测试用例数据库保存可用于测试产品的所有测试用例。有关测试用例数据库的更多信息，请参阅本书的第15章。）这些测试用例可以是任意优先级（0、1或2）。

情况2 如果缺陷修改的关键性和影响程度中等，则需要执行所有优先级0和优先级1的测试用例。如果要补充（少量）优先级2的测试用例，也可以选择并用于回归测试。在这种情况下，选择优先级2的测试用例是建议做的，但不是必须的。

情况3 如果缺陷修改的关键性和影响程度很高，则需要执行所有优先级0和优先级1的测试用例，并从优先级2中仔细选择一部分测试用例。

图8-5显示了以上讨论的几种情况。

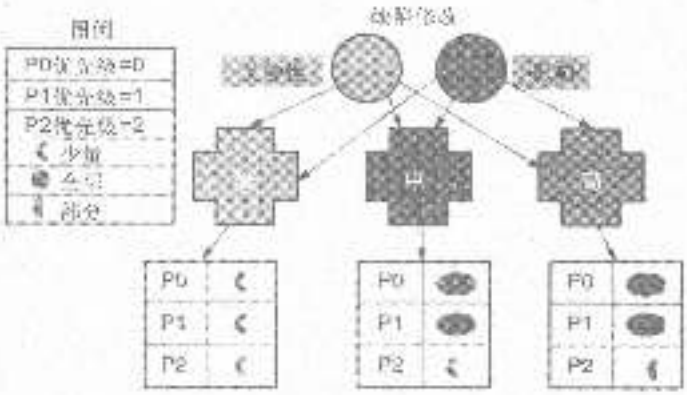


图8-5 选择测试用例的方法论

以上方法论要求分析所有缺陷的缺陷修改影响程度。这个过程可能很费时。如果有某种原因，没有足够的时间，且不进行影响程度分析的风险很低，那么可以考虑采用以下替代方法论：

- **全回归** 对于回归测试，运行全部优先级0、1和2的测试用例。这意味着要执行回归测试束/包中的全部测试用例。
- **基于优先级的回归** 基于全部优先级0、1和2的回归测试要根据合适的时间，按顺序执行测试用例。回归测试的终止要取决于合适的时间。

- **回归变更** 使用这种方法论的回归测试，要与上一测试周期进行代码变更比较，并根据其对代码的影响程度选择测试用例（灰盒测试）。
- **随机回归** 这种回归方法论随机地选择和执行测试用例。
- **基于背景的动态回归** 选择少量优先级0的测试用例，并根据这些测试用例执行后的情况和结果分析（例如，发现了新的缺陷、边界值），选择补充有关的测试用例，继续进行回归测试。

有效的回归策略通常是所有以上方法论的组合，并不一定孤立使用。

8.4.5 重新设置测试用例以进行回归测试

采用以上方法论选择了测试用例后，下一步就是为执行准备测试用例。为了执行这个步骤，需要"测试用例执行结果"记录。

在包括多轮测试的大型产品版本中，记录在哪个测试周期执行了哪些测试用例，执行结果和相关信息是非常重要的。这叫做测试用例执行结果记录，是第15章要讨论的测试用例数据库的一部分。

在很多公司中，并不是所有类型的测试和所有测试用例都要在每个测试周期中执行。前面介绍过，测试用例执行结果记录提供什么时候执行了什么测试用例的有用信息。使用测试用例执行结果记录，确定要选择的回归测试测试用例的方法或规程叫做重新设置规程。重新设置测试用例只不过就是在测试用例数据库（TCDB）中设置"不运行"或"再执行"的标志。这种重新设置规程还隐藏了以前版本测试用例的执行结果，使得执行这些测试用例的测试工程师不会受测试结果记录的影响产生偏见。

重新设置测试用例迫使测试人员检查所有测试用例，并根据缺陷修改的影响程度选择恰当的测试用例，降低测试缺陷修改的风险。如果是发布前的缺陷修改则风险更大，因此必须选择更多的测试用例。

重新设置测试用例一般不经常实施，出现以下情况时可以考虑实施：

1. 当产品中有重大修改时。
2. 当构建规程出现影响产品的变化时。
3. 大的发布周期，其间有些测试用例长期没有执行。
4. 当产品要用少量的测试用例进行最终回归测试时。
5. 当测试用例的预期结果与以前测试周期有很大不同时。
6. 与缺陷修改和生产问题有关的测试用例需要对每个发布版进行评估。如果这些测试用例的评估情况良好，则可以重新设置。
7. 只要删除了现有应用程序的功能，就要重新设置相关的测试用例。
8. 可以删除总产生正向结果的测试用例。
9. 可以删除与少数负面测试条件（不产生任何缺陷）相关的测试用例。

如果没有出现以上情况可以重新运行测试用例，而不是重新设置测试用例的执行结果。重新运行和重新设置测试用例的状态之间只有少量差别，这两种情况都要执行测试用例，但是对于重新设置可以预期得到与以前测试周期的不同结果。对于重新运行，预期测试用例会得到与以前执行一样的测试结果，因此管理层不必过度担心，因为执行这些测试用例主要是走程序，不期望会发现什么重大问题。

属于"重新运行"状态的测试用例，可通过再次测试增强大家对产品的信心。大家预期

这些测试用例不会不通过，或影响发布。属于“重新设置”状态的测试用例说明测试结果可能会与以前不同，只有当这些测试用例执行后才知道回归的结果和发布版本的状态。

例如，如果在产品安装中有一个变更，这个变更不会影响产品功能，那么该变更可以通过重新运行一些测试用例独立地测试，并且这些测试用例不必“重新设置”。类似地，如果有某个功能要进行重大修改（在设计、体系结构或代码上的修改），那么该功能所有相关的测试用例都需要“重新设置”，并必须再次执行这些测试用例。通过重新设置测试用例，测试工程师没有办法知道以前的执行结果。这样做可以消除偏见，迫使测试工程师提取并执行这些测试用例。

测试用例的重新运行状态表示发布版本有很低的风险，重新设置状态表示发布版本有中到高的风险。因此，对于马上就要发布的产品，最好能在执行“重新运行”测试用例之前，先“重新设置”测试用例。

是否进行重新设置还取决于功能的稳定性。如果要执行优先级1的测试用例，要先达到优先级0的测试用例执行情况良好（比方说有95%的通过率），那么就不要重新设置优先级0的测试用例。除非有重大变更，在优先级2测试阶段时对优先级1的处理也一样。

下面介绍“重新设置”标志在各个阶段的回归测试中的使用。

组件测试周期阶段

组件测试周期之间的回归测试只使用优先级0的测试用例。对于每个提交测试的版本，要选择版本号并重新设置所有优先级0的测试用例。这个测试周期只有在所有优先级0的测试用例全部通过后才开始。

集成测试阶段

组件测试后，如果在集成测试周期之间进行回归测试，则执行优先级0和优先级1的测试用例。优先级1的测试可以使用多个版本，在这个阶段，只有当缺陷修改和补充特性的关键性和影响程度很高时，才对这些测试用例进行重新设置。这个阶段的重新设置规程可能影响所有优先级0和优先级1的测试用例。

系统测试阶段

当所有优先级1的测试用例已执行，且根据测试计划制定的标准已达到可接受的通过率，就要开始执行优先级2的测试用例。在系统测试阶段，只有当缺陷修改和补充特性的关键性和影响程度很高时，才对这些测试用例进行重新设置。这个阶段的重新设置规程可能影响所有优先级0、优先级1和优先级2的测试用例。

重新设置测试用例的理由

回归测试要使用大量已经执行过的测试用例，要与一些执行结果和执行结果的假设关联。“重新设置”规程清楚地指示还有多少测试工作没有完成，并反映回归测试的状态。

如果测试用例没有重新设置，则测试工程师会倾向于根据以前的版本报告完成率和其他结果。这是因为人们会有在每个测试阶段使用多个版本的基本假设，会感觉如果在过去的版本通过也会在新版本中通过。回归测试不能有“未来是过去的延伸”这种假设。重新设置作为一种规程可以消除关于测试用例的任何偏见，因为重新设置测试用例执行结果可以防止测试工程师看到测试用例执行历史。

8.4.6 总结回归测试的结果

每个人都关心回归测试的结果，因为这对测试不仅说明缺陷情况，还说明缺陷的修改情况。

在测试周期结束后，有时在马上就要发布产品的时候，除了测试团队，公司中的很多人也会关注回归测试的结果。开发人员也关注回归结果，因为他们想知道他们的缺陷修改对产品的作用如何。因此，需要理解总结回归结果的方法。

由于回归测试使用已经执行多次的测试用例，如果缺陷修改做得好，预期对于同一个版本会有100%的测试用例通过率。如果通过率没有达到100%，测试经理会与未通过测试用例的以前执行结果进行比较，以确定回归测试是否成功。

- 如果特定测试用例的执行结果对于以前的版本通过，而当前版本未通过，则回归未通过。需要提交新版本，并对测试用例重新设置后从头重新测试。
- 如果特定测试用例的执行结果对于以前的版本未通过，而当前版本通过，则可以有把握地说缺陷修改有效。
- 如果特定测试用例的执行结果对于以前的版本未通过，而当前版本也未通过，如果对于这个特定的测试用例没有针对缺陷修改，则可能意味着这个测试用例的执行结果应该未通过，而且在回归测试中不应该选择这类测试用例。
- 如果特定测试用例的执行结果对于以前的版本未通过，但是通过某种写入文档的迂回方法可以通过，而且测试人员也对这种迂回方法感到满意，则可以认为对于系统测试周期和回归测试周期都通过。
- 如果测试人员对迂回不满意，则可以认为系统测试周期是未通过，但是回归测试周期是通过。

以上内容通过表8-1进行归纳。

表8-1　总结回归测试周期的结果

回归测试的当前结果	以前的结果	结　论	说　明
未通过	通过	未通过	需要改进回归测试程序进行长期保障
通过	未通过	通过	这是好的回归预期结果，说明缺陷修改有效
未通过	未通过	未通过	需要分析为什么缺陷修改没有成功。"修改有问题吗？"还要分析为什么要在回归中重新运行这个测试用例
通过（通过迂回方法）	未通过	分析迂回方法，如果满意，则通过	迂回也需要进行认真的评审，因为可能产生副作用
通过	通过	通过	这类结果给人们以没有出现由于缺陷修改而引起副作用的良好感觉

8.5 回归测试的最佳实践

实践1：回归可以用于所有类型的发布。

回归测试方法论可以用在以下场合：

1. 需要评估测试周期之间的产品质量（包括计划内的评估和基于需要的评估）；

2. 正在准备产品的一次重大发布，已经通过了所有测试周期，正在策划针对缺陷修改的回归测试；

3. 正在准备产品只有缺陷修改的一次小发布（支持包、补丁等），可以策划回归测试周期关注这些缺陷修改。

可以为每次发布策划多个回归测试周期。这种情况适用于多个阶段的缺陷修改，或要检查某些缺陷修改是否对某个具体版本无效的情况。

在测试执行期间，如果将未通过的结果指派给测试用例，最好也能够输入该缺陷的标识（通过缺陷跟踪系统获得），以便了解在缺陷修改提交时需要运行哪些测试用例。请注意，一个特定测试用例可能对应多个缺陷，一个特定缺陷可能影响多个测试用例。

实践2：将缺陷标识映射到测试用例上可改进回归质量。

即使在理想情况下已经在测试用例和缺陷之间作了映射，但是选择要执行的测试用例检查缺陷修改的副作用时，可能仍然在很大程度上是个手工过程，因为这要求了解各种缺陷修改之间的相互依赖关系。

随着时间的流逝和产品各个版本的发布，要执行的回归测试用例数目会增加。人们已经发现，过去客户报告的一些缺陷是由于最后时刻的缺陷修改产生的副作用引起的。因此，选择回归测试的测试用例确实是门艺术，并不容易。大部分人都想在回归测试上用最小的投入获得最大的回报，这使问题更加复杂。

为了解决这个问题，在对产品进行修改时，要在现有的测试用例包中增加或删除测试用例。这种叫做回归包或回归测试床的测试用例包，在为应用程序或产品引入新变更时运行。回归测试床中的自动化测试用例可以在每天晚上生成的版本上执行，以保证产品的质量在产品并发阶段持续保持。

实践3：每天都创建并运行回归测试床。

前面介绍过，缺陷、产品及其相互依赖关系的知识和结构化程度很高的方法论，对于测试用例的选择都是非常重要的。这些都强调需要选择合适的人做合适的工作。团队中最有经验的人或能力最强的人，在选择合适的测试用例进行回归上可能要比缺乏经验的人做得好。经验和能力会得到产品中薄弱部分的知识，并会影响缺陷分析。

请观察后面的图片。在第一张图片中，老虎被关在笼子里防止它伤害人。在第二张图片中，家庭成员躲在蚊帐内防备蚊子的侵扰。

实践4：使用最好的测试工程师挑选测试用例。

对于回归也采用同样的策略。就像笼子中的老虎，产品中的所有缺陷都必须标识和修改。这就是“检测产品中的缺陷”的含义。前面各章讨论过的所有类型的测试和回归测试都采用这种技术发现每个缺陷并修改。

蚊帐内家庭成员的图片表示“保护自己的产品不受缺陷侵扰”。这个策略是缺陷预防。有很多验证和质量保证活动，例如评审和审查（请参阅第3章）都试图预防缺陷。

与回归测试有关的另一个问题是“保护自己的产品不受缺陷修改的侵扰”。前面介绍过，被定为轻微缺陷的缺陷在代码中进行修改后可能会对产品造成重大影响。这与蚊子侵扰（影响）人的情况类似，尽管蚊子的个头很小，但其造成的危害却很大。因此，不管缺陷有多少和严重性有多轻，在通过代码对缺陷进行修改前，最好都要对缺陷修改的影响进行分析。对缺陷修改造成的影响进行分析是很困难的，因为时间很紧且产品很复杂。因此，最佳实践是，在临近产品发布的时候，限制对产品进行修改的量。这可以保护产品不引入由于缺陷修改造成的缺陷，就像蚊子可以通过小洞侵入蚊帐。如果在蚊帐上开个门赶走蚊子，那么这也会为新蚊子进入蚊帐打开了大门。不进行影响分析的问题改正会在产品中引入大量缺陷。因此，不仅要将产品与缺陷隔离，而且还要将产品与缺陷修改隔离，这是非常重要的。

实践5：检测缺陷，保护产品不受缺陷和缺陷修改的侵扰。

如果发现了缺陷，而且产品已经受到保护，防止缺陷和缺陷修改的侵扰，那么回归测试就是有效的和高效的。回归测试实际上提供的是蚊帐。

问题与练习

1. 一个用于工资自动化管理的产品定于三月发布。以下是该产品的一些特性。对于这些特性，标识出哪些应纳入发布前，三个月内的回归测试。假设九月还要发布另一个维护版。
   a. 输入员工信息的屏幕
   b. 法定年终报告
   c. 计算每月工资单的详细内容
   d. 维护员工地址的屏幕
2. 一个产品有一些很少运行的测试用例，表面上是“由于没有时间”运行。请讨论作为回归测试的一部分运行这类垃圾测试用例的优缺点。
3. 以下哪个测试用例应纳入给定的回归测试周期内？
   a. 正准备发布产品的版本7，有一些测试用例针对的是引入这个版本的新特性。
   b. 版本6在缓存区管理模块中有一些缺陷，而版本7是在这个模块上构建的。
   c. 版本6的用户界面模块相当稳定（即缺陷很少），版本7对这个模块的修改非常少。
   d. 版本7已完成贝塔测试阶段，并且在贝塔测试期间，一些重要的客户使用场景发现了产品中未发现的缺陷。
   e. 该产品是一个数据库软件，有一些诸如数据库启动、基本SQL查询和多场地查询的特性。
4. 本章讨论了划定测试用例优先级的方法。利用本章给出的建议，请为以下测试用例分配优先级P0/P1/P2。
   a. 针对网络产品的测试用例测试基本的流程控制和错误控制。
   b. 针对数据库软件的测试用例测试所有连接查询的选项。
   c. 针对文件系统的测试用例检查不连续的空间分配。
   d. 测试用正常参数启动操作系统的测试用例。
   e. 测试以“安全模式”启动一个应用程序的测试用例，这种情况不经常出现。
   f. 测试用例针对的特性于在当前版本作重要变更。
   g. 测试用例针对的是很稳定的特性，并且从没有发现任何严重缺陷。
5. 如果执行回归测试的时间非常紧张，请讨论本章给出的各种回归测试方法论（全回归、回归变更等）的优缺点。
6. 一个回归测试工程师提供了以下测试结果。请讨论选择回归测试和开发过程的效率：
   a. 针对打印功能的测试在以前的版本中没有发现缺陷，也没有在当前版本中发现缺陷。
   b. 针对多币种的测试发现的缺陷在当前版本中依然存在。
   c. 一个要交付的特性在当前的版本中有新缺陷。
7. 请讨论针对产品大发布版本和小发布版本的回归测试的差别。
8. 本章和第15章详细讨论了测试用例数据库、配置管理库和缺陷库。请讨论如何在这三个库之间建立联系，以自动选择给定发布版本的最优回归测试。

第9章 国际化〔$I_{18}n$〕测试

9.1 引言

软件市场已经真正地全球化。因特网的发展已经消除了影响软件产品广泛使用的一些技术障碍，并且简化了软件产品在全球的发行。但是，软件产品适用不同国家语言的能力，会严重影响产品的接受率。因此，一项既苛刻又合理的软件需求，是要支持不同语言（例如中文、日文和西班牙文）并遵循地区的习俗。本章从开发和测试角度介绍为了使软件产品能够以不同的语言在不同的国家使用所要做的工作。构建针对国际市场的软件，支持多种语言，既经济又高效的方法是在整个软件开发生存周期，即从需求获取到设计、开发、测试和维护都采用国际化标准。本章给出的指南有助于从一开始就构建具有足够灵活性以支持多种语言的软件产品，压缩构建软件产品支持多种语言所需的时间和工作量。

如果在针对国际化的软件生存周期中没有遵循一些指南，那么每支持一种新语言所需要做的工作和额外成本就会随时间的流逝显著增加。针对国际化的测试要保证软件没有强设任何特定语言或与某种特定语言相关的习俗。国际化测试要在软件生存周期的各个阶段实施。9.3节将详细讨论这个问题，以下首先介绍在国际化中使用的基本术语。

9.2 国际化介绍

9.2.1 语言的定义

为了解种国际化要用到的术语，需要理解什么是语言。语言是用于沟通国际化的一种工具。本章主要关注人类语言（例如日文、英文等），而不是计算机语言（例如Java、C等）。语言有一套字符（字母），一套由这些字符构成的（有效）单词，以及语法（关于如何构成单词和句子的规则）。此外，语言还有语态，即与句子关联的含义。对于同一种语态，口头使用、书面使用和语法会因国家而异。但是在大多数情况下，字符/字母还保持不变。单词对于使用相同语言的国家可以用一组不同的字符表示同样的含义。例如，“颜色”这个单词，在美国英语中的拼写是“color”，在英国英语中的拼写是“colour”。

9.2.2 字符集

本节归纳一些用于在计算机中表示不同语言字符的标准。本节的目的是介绍不同字符表示的思想。理解本章内容并不要求具有对每个字符如何表示和内部知识的深入理解。

ASCII　ASCII是美国信息交换标准代码。ASCII是在计算机中使用的字符的字节表示（8个位）。ASCII采用7个位表示在计算机中使用的所有字符。使用这种方法，可以以二进制的形式表示128（2^7=128）个字符。ASCII是最早的计算机二进制字符表示的方法之一。以后ASCII经过扩展，又包含了没有使用的第8个位，即可以以二进制的形式表示256（2^8=128）个字符。这种表示包含了更多的标点符号、欧洲语言字符和特殊字符。扩展ASCII还有助于表示

带重音的字符（例如ñáéíóú），带重音的字符是类似英语的字符，在欧洲和西方语言中具有特殊的含义。

双字节字符集（DBCS）　在ASCII中采用单个字节表示英语字符。有些语言，例如中文和日文拥有很多不同的字符，不能用一个字节表示。要使用两个字节表示每个字符，因此称做双字节字符集。在单字节表示法中，只能（以二进制的形式）表示256个不同的字符，而在双字节字符集表示法中，可以表示65 535（2^{16}）个不同的字符。

统一码　ASCII或DBCS也并是以对一种语言的所有字符进行编码，但是肯定不是以表示全世界所有语言的所有字符。所有语言的字符需要以标准形式存储，解释和传输。统一码有效地满足了这种需要。统一码为每个字符提供了唯一的数字，不管是什么平台，程序还是语言。统一码使用定宽16位世界字符编码。由统一码财团进行开发、维护和推广。统一码通过表示每种语言的字符来对每种语言进行唯一编码，因此能够以相同的方式处理每种语言，即存储、搜索和操作。统一码转换格式（UTF）规定了要转换为统一码的不同语言的字符映射算法。统一码财团吸收了世界上现有的很多语言和属地（将在下一节讨论）。每种语言的每个字符都被分配了唯一数字，极大地促进了软件的国际化。所有软件应用程序和平台都在转向统一码。例如，表9-1中的信息解释了Microsoft Windows不同版本向统一码发展的情况。

表9-1　Windows操作系统向统一码发展

微软操作系统	国际化道路
Windows 3.1	ANSI
Windows 95	ANSI和有限的统一码
Windows NT 3.1	第一个基于统一码的操作系统
Windows NT 4.0	显示统一码字符
Windows NT 5.0	显示和输入统一码字符

9.2.3　属地

商业化软件不仅需要记住语言，而且要记住使用该语言的国家。有一些软件需要考虑与特定语言相关的习俗。有可能两个国家使用相同语法、单词和字符集的同一种语言，但是仍然可能有变化，例如货币和日期格式。属地这个词用于区分这类参数和习俗。属地可以认为是一种语言的一个子集，该子集定义了在不同国家或地区的行为。例如，英语是在美国和印度使用的语言，但是两个国家使用的货币符号是不同的（分别为$和Rs）。数字中使用的标点符号也不同，例如1百万在美国表示为1 000 000，在印度表示为10 00 000。

为了正常运行软件，除了语言还需要记住属地。可能在使用同一种语言的国家里要使用多种货币（例如在法国使用欧元和法郎，而法国使用的语言是法语）。还有可能使用多种日期格式。属地这个词包含所有这些含义，以及其他一些可能很重要的含义。因此，一种语言可以有多个属地。

9.2.4　本章使用的术语

国标化（$I_{18}n$）是普遍使用的一个伞状术语，覆盖使软件能够推向国际市场所需的所有活动，包括开发和测试活动。国标化的简写形式是$I_{18}n$，下标18表示在“国际化”这个英文单词Internatsonalization中，在I和n之间有18个字符。在各个阶段实施的测试保证所有这些活动都已恰当地完成，称做国际化测试或$I_{18}n$测试。

全球化＝国际化＋本地化

本地化（$L_{10}n$）表示所有软件资源，例如消息向目标语言和习俗的转换工作。本地化的简写形式是$L_{10}n$，下标10表示在“本地化”英文单词Localization中，在L和n之间有10个字符。消息

和文档的转换要由懂得英文（默认所有软件消息都采用英文）又懂得软件要转换的目标语言的一组语言专家来完成。对于需要准确转换的情况，消息连同使用背景都包含在I18n术语中。

全球化（G11n）这个词使用得并不很普遍，表示国际化和本地化。一些公司使用全球化这个词，想把国际化和本地化活动区分开来。这种情况可能是有必要的，因为本地化活动由完全不同的一组语言专家考虑（按每种语言），并且这些活动一般是通过外购引入的。有些公司使用I18n只表示编码和测试活动，不包括转换。

9.3 国际化测试的测试阶段

国际化测试需要清楚地了解所包含的所有活动及其顺序。由于测试的任务就是要保证其他团队完成活动的正确性，因此本节还要讨论在测试团队之外的完成活动。图9-1使用不同灰度的方框描述国际化测试的各种主要活动。最上和最下一排共三个方框表示由开发人员完成的活动；左侧上部三个方框表示由明敏本地化团队完成的活动（本章稍后还要讨论），其他方框表示由测试工程师完成的测试活动。以下逐个详细介绍这些活动。

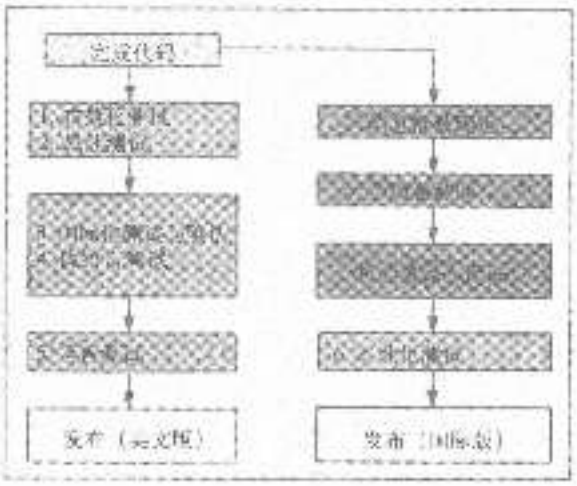


图9-1 国际化测试中的主要活动

国际化测试要在项目生存周期中的多个阶段内实施。图9-2进一步表示了第2章讨论过的软件开发生存周期的V字模型，给出了这种模型的不同阶段与各种I18n测试活动的关系。请注意，有效化测试要作为单元测试阶段的一部分，由开发人员完成。

国际化测试的一些重要问题包括：

1. 测试代码，检查其如何处理输入、字符串和排序；
2. 各种语言的消息显示；
3. 各种语言和习俗的消息处理。

这些问题是以下各节要讨论的要点。

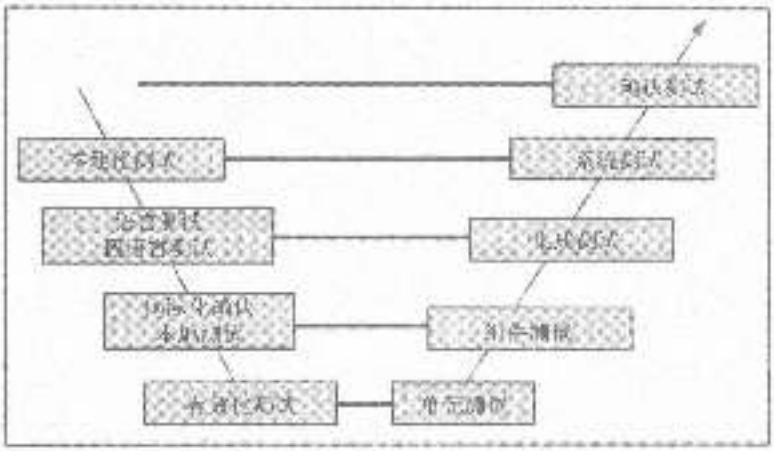


图9-2　软件开发生存周期V字模型的阶段与国际化活动的关系

9.4　有效化测试

有效化测试是一种白盒测试方法论，要保证软件所使用的源代码允许国际化。采用硬代码的方式表示货币和日期格式、采用固定长度的GUI屏幕或对话框，直接在媒介上读写消息，这样的源代码都不是有效性代码。代码评审或代码审查活动结合一些单元测试的测试用例，目标是找出国际化缺陷，这样的活动叫做有效性测试。2000年发现的绝大多数缺陷都是l18n缺陷，因为都使用硬编码格式。有效性测试使用检查单。有效性测试评审检查单的一些检查项包括：

- 检查API/函数调用的代码是否不是国际化API集的一部分。例如C语言中的函数printf()和scanf()就不是国际化有效的调用。要使用NLSAPI、统一码和GNU gettext定义的一些调用集。（有些公司使用预先编写的分析器/脚本捕获这些非国际化函数使用的明显问题。）
- 检查硬编码的日期、货币格式、ASCII代码或字符常量的代码。
- 检查代码是否没有对日期变量进行计算（加、减），或在代码中强制使用日期的不同格式。
- 检查对话框和屏幕是否留有至少0.5倍宽度的扩展空间（因为经过翻译的文本可能会占用更多空间）。
- 保证代码中没有基于地区文化的消息和俚语（基于地区文化的消息很难翻译成其他语言）。
- 保证没有在代码中执行字符串操作（子字符串搜索、拼接等），只使用系统提供的国际化API进行字符串操作。
- 验证代码没有为了执行功能，假设以特定的语言预先定义路径、文件名或目录名。
- 检查代码没有假设语言字符可以用8位、16位或32位表示（有些程序使用C语言的位移动操作读取下一个字符，这是不允许的）。
- 保证为缓存区和变量留有足够的空间容纳翻译后的消息。
- 检查位图和图表没有嵌入可翻译的文本（需要翻译的文本）。
- 保证所有消息及其使用背景都写入文档，供翻译者使用。
- 保证所有资源（例如位图、图标、屏幕、对话框）都与代码分离，并存储在资源文件中。

- 检查没有消息包含技术行话。所有消息对于即使经验很少的产品用户都是可理解的，这还有助于恰当地翻译消息。
- 如果代码所有文本滚动，那么屏幕和对话框必须提供恰当的滚动方向变更指示。例如从上到下、从左到右、从右到左、自底向上等，因为不同语言有不同的习惯。例如，阿拉伯使用“从右到左”阅读方向，“从左到右”滚动。[请注意，滚动方向一般与阅读方向相反，如图9-3所示。]

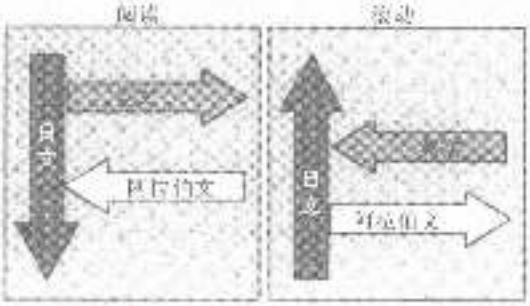


图9-3 阅读与滚动方向

对于国际化测试的所有阶段，如果做得好，有效化测试阶段一般会找出最大量的国际化缺陷。在设计和初步编码时，要注意保证满足检查单上的所有检查项和其他国际化需求。针对国际化的代码有效化测试不应推迟到编码阶段。如果产品开发团队先开发产品（代码），然后考虑为一个独立阶段考虑满足国际化要求，实践证明这是低效的，生产率很低。因为在这种情况下，必须针对国际化要求在很大范围内修改代码（在实现了基本功能以后）。因此，从单元测试开始到最后的确认测试的所有测试阶段，都需要重复，因为这类修改会影响软件的基本功能。

> 有效化测试的目标是在单元测试期间使用国际化标准验证代码。

9.5 属地测试

针对国际化要求验证了代码，完成了有效化测试后，下一步是确认产品中的属地变更效果。属地变更会影响日期、币种格式和屏幕、对话框和文本中的显示项。利用系统设置或环境变量改变不同的属地，并测试软件功能、数字、日期、时间和币种格式，这种测试叫做属地测试。如前所述，每种语言都可能有多个属地，在属地测试中需要考虑所有这种组合。对于每个组合，每次都要修改计算机中的属地设置，以测试功能和显示。

只要属地发生变化，测试人员就需要理解软件功能的变化。这需要代码有效化方面的知识。如果代码有效化做得不对，则必须测试产品每个属地的所有特性。但是在实际场景中，每个功能都分配了优先级（高、中、低），并根据功能的优先级进行测试。优先级的分配要考虑国际客户的功能重要性和影响。由于属地测试是软件开发生存周期V字模型（请参阅图9-2）中组件测试阶段的一部分，可以针对国际化测试为每个组件分配相对优先级。这样就可以更关注那些对客户很重要的功能，以及有效化不好的组件的属地测试。属地测试的一些检查项包括：

1. 适用于国际化的所有特性都要采用软件所针对的不同属地进行测试。不被认为是国际化测试的一些活动包括审计、代码调试、活动日志，以及只供使用英文的管理员和程序员使

用的特性。

2. 采用不同可用的属地测试热键、功能键和帮助示。（即检查属地变更是否影响键盘设置。）

3. 日期和时间格式与所定义的语言属地一致。例如，如果选择美国英文属地，则软件应该显示mm/dd/yyyy日期格式。

4. 币种与所选择的属地和语言一致。例如，如果语言是澳大利亚英文，则币种应该是AUS$。

5. 数字格式与所选择的属地和语言一致。例如，使用合适的十进制标点，并且标点放在合适的位置。

6. 考虑了时区信息和夏时制时间计算一致性（如果被软件使用），并且是正确的。

属地测试关注测试数字、标点、日期、时间和币种格式的表示习俗。

属地测试不像有效化测试或本章介绍的其他测试阶段，是一个专门的规程。属地测试关注点局限于属地的变更，并测试日期、币种格式和重要功能的键盘热键。请注意，可以通过操作系统的设置改变服务器或客户端的属地。例如，在Microsoft Windows 2000中，通过点击“启动->设置->控制面板->地区选项”改变属地。

9.6　国际化确认

国际化确认与国际化测试不同，国际化测试是本章讨论的所有类型测试的超集。实施国际化确认的目的是：

1. 用ASCII、DBCS和欧洲字符对软件功能进行测试。

2. 软件对所选择的语言和字符处理字符串操作、排序和队列操作。

3. 软件在GUI和菜单中非ASCII字符的显示要一致。

4. 软件消息处理正确。

这里，对测试的真正挑战是非ASCII字符和这些字符对软件的影响。为此，要使用与具体语言相关的键盘和能够生成非ASCII字符的工具。IME（输入方法编辑器）是Microsoft平台上的软件工具，可以用来向软件输入非英文字符。这个工具有助于生成非英文ASCII字符。在Unix/Linux平台上有大量类似工具可以免费下载。其中很多工具都带有“软键盘”，使用户能够以目标语言输入字符。图9-4显示了输入日文（片假名）字符的IME软键盘布局。

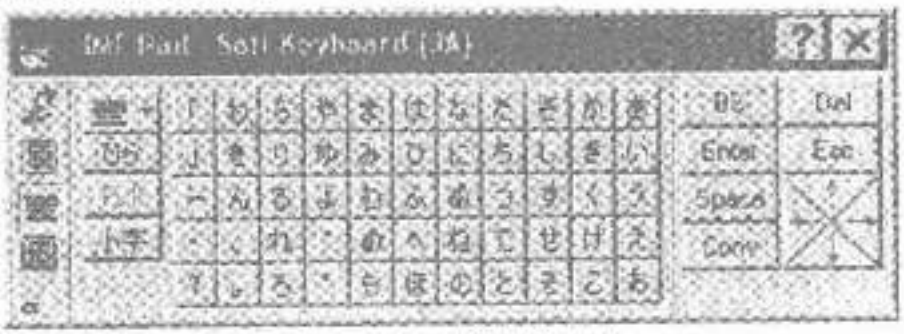


图9-4　针对日文的IME软键盘

国际化确认检查单包括以下检查项：

1. 对于所有语言和属地，功能不变。

2. 排序方法符合每种语言和属地习俗。

3. 能够使用工具，例如IME，将非ASCII或特殊字符输入软件，并且功能必须一致。

4. 名称中的非ASCII字符显示与所输入的一样。

5. 剪切或复制—粘贴非ASCII字符，在粘贴后保留其风格，软件能够正常运行。

6. 对于采用IME和其他工具生成的不同语言的单词和名称，软件能够正常运行。例如，采用英文用户名和带重音字符的德文用户名登录应该同样有效。

7. 文档具有一致的文档风格、标点，针对每位读者遵循所有语言和原地习俗。

8. 软件中的所有运行时消息都与语言、国家术语、用法和标点一致。例如，数字的标点对于不同的国家是不同的。货币量123456789.00在美国应该表示为123,456,789.00，在印度应该表示为12,34,56,789.00。

引入国际化确认阶段是为了关注所有组件中的消息和功能中所有与界面有关的问题，因为下一阶段将进行所有组件的集成。重要的是，以上测试要在实施下一级测试前对所有组件进行，下一阶段是假语言测试。

国际化确认关注组件功能的非英文消息输入和输出。

9.7 假语言测试

假语言测试采用软件翻译器尽早发现翻译和本地化问题，假语言测试还要保证语言之间能够正确切换，从合适的包含翻译过的消息目录中选择正确的消息。假语言测试有助于在产品本地化之前，主动地发现问题。为此，要从软件中取出所有消息，并通过工具转换假语言进行测试。假语言翻译器采用易于理解和测试的类英文语言。这类测试有助于使用英文的测试人员发现原本只能由语言专家在本地化测试期间发现的缺陷。图9-5展示了假语言测试。

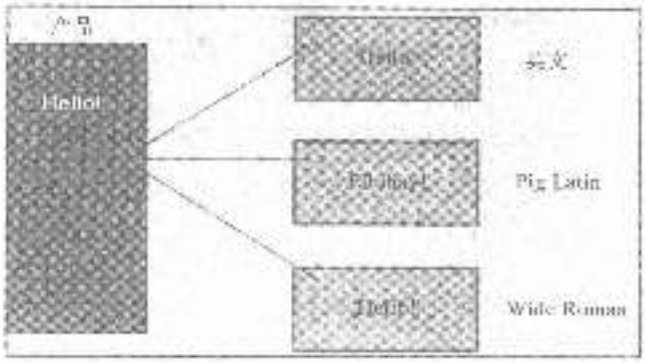


图9-5 假语言测试

在图9-5中使用了两种类英文的假语言（Pig Latin和Wide Roman），程序中的消息"Hello"以Pig Latin显示为"Ellohay"，以Wide Roman显示为"Hellolo"。这有助于确定采用系统设置动态地改变语言时，软件是否能够选择正确的语言。

以下是假语言测试检查单可以使用的检查项：

1. 保证软件功能至少被一种欧洲单字节假语言（Pig Latin）测试过。

2. 保证软件功能至少被一种双字节假语言（Wide Roman）测试过。

3. 保证所有字符串能够在屏幕上正确显示。

4. 保证弹出窗口和对话框的宽度和大小足以显示假语言字符串。

假语言测试通过软件翻译器，帮助模拟一种不同语言的本地化产品功能。

9.8 语言测试

语言测试是“语言兼容性测试”的简称。语言测试保证采用英文创建的软件在英文和非英文的不同平台和环境中能够正常运行。例如，如果软件应用程序是针对Microsoft Windows开发的，那么就要使用Microsoft Windows不同的可用语言设置测试该软件。由于这时软件还没有本地化，还会继续使用英文打印，使用英文消息进行交互。语言测试要保证在其他语言设置上软件不会崩溃，仍然具有兼容性。

语言测试关注用户产品的全球环境和非英文服务功能测试英文软件。

图9-6给出了语言测试和在客户—服务器体系结构中必须测试的各种属地组合。如图9-6所示，为了进行语言测试，要选择一种欧洲语言代表带重音的字符，选择一种双字节语言的代表，以及一种默认语言，即英文。

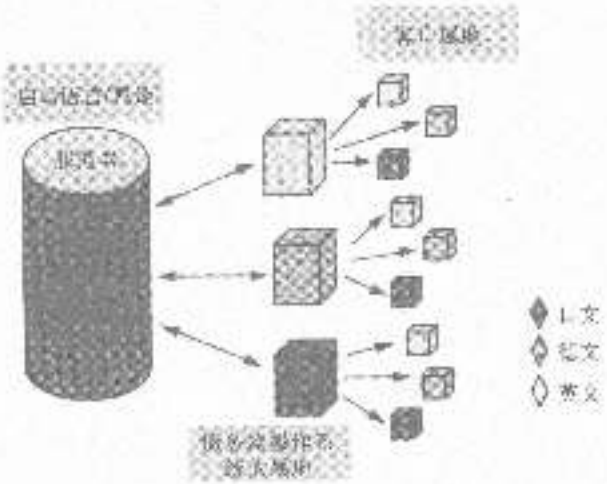


图9-6　语言测试和在客户—服务器体系结构中必须测试的属地组合

图9-6给出的属地组合的绝大部分测试要保证该英文软件对于不同语言平台的兼容性，在测试时，重要的是要检查与属地有关的问题，因为属地测试没有暴露的缺陷在语言测试中可能会显示出来。

还要注意理解当数据在计算机之间或软件和操作系统之间传递时，会有很多针对国际化的代码页、位流和消息转换，这一点很重要。这可能产生一些功能问题。因为要进行数据转换，最好能够设计一部分与语言测试相关的性能测试用例。不管软件、操作系统或其组合使用什么语言，都预期软件应该有类似的性能。这一点需要在语言测试中检验。

语言测试至少应该包含以下检查单：

1. 在英文、一个非英文和一个双字节语言平台组合上检查功能。
2. 在不同语言平台上和连接到网络上的不同计算机上检查关键功能的性能。

9.9 本地化测试

软件接近发布日期时，要将消息抽取到一个单独的文件中，并提交给多语言专家进行翻译。采用一套构建工具自动提取所有消息和其他资源（例如GUI屏幕、图片），并存入独立的文件中。用Microsoft的术语说，就是资源文件。在一些其他平台上，这叫做消息数据库。如

前所述，需要翻译的消息要与消息的使用背景说明一同提交，以便得到更好的翻译。多语言专家可能不了解软件或软件的目标客户，因此，最好消息中不要包含任何技术行话。

本地化是一种非常昂贵和费工的过程，并不是属于该软件的所有消息都需要翻译。有些消息可能是供管理员和开发人员看的，可能不是给目标用户看的。这类消息不需要翻译，应该排除在供翻译的文件外。在翻译时，不仅是消息，而且像GUI屏幕、对话框、图标和位图这样的资源也是本地化的对象，因为需要改变尺寸和定制与翻译后的消息匹配。定制是很重要的，因为滚动方向和阅读方向在一些语言中是不同的，图9-7和以下的解释有助于理解定制对于不同习俗的重要性。

图9-7 阅读方向的差别

当向讲英文的国家的人展示图9-7时，他们会理解左侧的脏圆领汗衫在放入洗衣机内后会变成右侧的干净圆领汗衫，这种理解是因为这些国家习惯从左到右阅读，如果把这幅图展示给阿拉伯国家的人看，他们会认为干净的圆领汗衫放入洗衣机后变成脏的，因为他们的阅读方向是从右到左！

> 国际化后产品就真正全球化了，能够以不同的语言运行，并能与其他全球化产品一起运行。

因此资源定制对于软件本地化是非常重要的，不仅软件，文档也需要针对目标语言和习俗进行本地化。

本地化之后的下一步是进行本地化测试，如前所述，翻译了消息、文档，定制了资源后，要启动一个测试周期，检查软件功能在本地化环境中是否能像预期的一样，这叫做本地化测试。这种测试需要懂英文和目标语言，并且具有软件和运行原理知识的语言专家，以下检查单有助于实施本地化测试：

- 所有消息、文档、图片、开幕都要本地化，以反映当地用户和国家、风俗和语言的习俗。
- 排序和大小写转换符合语言习俗。例如，英文排序顺序是A、B、C、D、E，而在西班牙文中，排序顺序是A、B、C、CH、D、E，如图9-8所示。

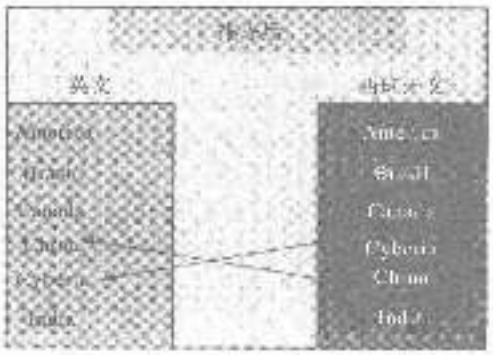


图9-8 英文与西班牙文的排序顺序

- 在翻译后的消息、文档和屏幕中，字体大小和热键仍然有效。
- 软件的过滤和搜索能力与语言和区域习惯一致。
- 本地化软件中的地址、电话号码、数字和邮政编码与目标用户的习惯一致。

9.10　国际化使用的工具

已经有一些针对国际化的工具，很大程度上取决于所使用的技术和平台。例如，针对客户-服务器技术的工具与针对使用Java的Web服务技术的工具是不同的。表9-2给出了一些已有的样本免费工具。产品和平台不同，对国际化测试的需求也会不同。因此，公司在选择合适的工具前，应该仔细地进行研究。

表9-2　样本国际化工具

序号	Microsoft操作系统工具样本清单	Linux操作系统工具名称	用　途
1	微软本地化工作室	GNU gettext()	有效化和有效化测试
2	http://BabelFish.Altavista.com	http://BabelFish.Altavista.com	图语言测试
3	微软地区设置	LANG和环境变量集	区域测试
4	IME	统一码IME（有来自不同公司的多个选项）	国际化确认
5	http://www.snowcrest.net/donnelly/pglatin.html	http://www.snowcrest.net/donnelly/pglatin.html	图语言测试（例如PigLatin）

9.11　挑战与问题

本章列出的活动有很多挑战和问题。第一个问题涉及归属。一些软件工程专家认为国际化和本地化是翻译团队的问题，这是不对的。使产品在市场上具备不具备国际性与开发产品的每个人都有关系。正如本章所介绍的，国际化工作散布在整个软件开发生存周期中。因此，国际化的责任并不只局限于翻译团队。很多公司都认为他们能够负担得起用一支单独的队伍，策划产品采用英文开发后对产品进行必要的国际化修改。本章已经说过，实践证明从长远看这是不经济和低效的。

国际化相信"在第一时间完成"。较后进行国际化或发布支持包或补丁，或以后进行修改，都不能认为是同一个软件发布版，甚至不可行。要让软件能够以不同语言运行的国际化工作，不能拖到下一个发布版本。类似地，如果产品的英文版已经发布，本地化测试也是唯一可以推迟到下一个发布版本的测试阶段。

对术语、活动和阶段有多种解释。只要满足本章所讨论的目标，这些解释的差别是微不足道的。在本章介绍的多种测试活动之间并没有鲜明的界限。测试工程师可以合并两种或三种国际化测试，一次完成。例如，区域测试、国际化确认和假语言测试可以合并为"国际化测试"。

国际化测试还没有像其他类型的测试那样得到人们的理解。国际化测试缺少标准工具和不够清晰，进一步使这个问题复杂化。开发界也面临类似的挑战，并为有很多变种，这对于研究开发解决这些问题的模型和方法论来说，既是挑战也是机遇。本章讨论的内容只是这个领域的冰山一角。

问题与练习

1. 请针对自底向上、从左到右和从右到左滚动，各确定至少一种语言。
2. 为什么国际化和本地化表示为I18n和L10n？
3. 配置管理在国际化中扮演什么具体角色？
4. 请给出两种俄语台以及能够支持其使用的相应工具。
5. 完成本地化测试需要什么基本技能？
6. 为什么统一码在国际化中的地位非常独特？
7. 请给出单元、组件、集成和系统测试阶段中的国际化活动。

第10章 即兴测试

10.1 即兴测试概述

前面各章已经介绍过，有很多测试技术和类型都可以帮助发现大量缺陷，从而提高产品的质量，最大限度地降低发布产品的风险。各章介绍的所有类型的测试都是计划测试的一部分，并采用一定的具体技术实施（例如边界值分析）。在本章中，将研究采用无计划形式实施的一组测试（所以叫做即兴测试）。本章将讨论即兴测试的特点，稍后介绍即兴测试活动与计划测试活动相比的相对位置，然后在后续各节讨论其他相关类型的即兴测试，即伙伴测试、探索式测试、结对测试、迭代式测试、敏捷与极限测试和缺陷播种。本章将比较所有这些方法，讨论每种方法适用（和不适用）的场景。

前面各章讨论的计划测试是由测试工程师及其在特定时间段内对产品的理解驱动的。假设被测产品以某个特定模式运行，正如需求文档和其他规格说明所描述的那样。这种假设的正确性可以在产品实现并准备测试时检验。测试工程师使用该产品越多，对它的理解也就越好。到了所实现的系统被完全理解时可能已经为时过晚，已经没有足够的时间完成详细的测试文档。因此，不仅需要根据所计划的测试用例进行测试，还需要根据使用产品所获得的更好理解进行测试。

计划测试面临的一些问题包括：

1. 需求和其他规格说明文档不够清晰。
2. 缺少实施测试的技能。
3. 缺少测试设计的时间。

首先，执行了一些经过计划的测试用例后，需求的清晰性得到改善。于是，早先编写的测试用例可能不能反映在这个过程中需求清晰性的改善。其次，经过一轮计划测试的执行，测试工程师的技能得到提高，但是测试用例可能并没有更新，以反映这种技能上的提高。最后，测试设计时间不够会影响测试的质量，可能考虑不够周全。

计划测试能够捕获一些类型的缺陷。虽然计划测试有助于提高测试人员的信心，但经常发现关键缺陷的，却正是测试人员的“直觉”。这是因为没有哪份规格说明能够完备地提供测试所需要的所有视角，正如第4章讨论领域测试时所提到的，有时没有规格说明的测试和计该领域专家测试产品会带来新的视角，有助于发现缺陷的新类型。当测试人员的思想“冲破牢笼”，这些视角就会不断动态地演化，有可能在计划测试中遗漏一些最关键的视角，以后又被测试人员发现。因此，测试人员的直觉起着很重要的作用。不仅如此，正如前面所提到过的，测试人员随着对产品理解不断清晰所获得的直觉和技能没有被已经设计好了的测试用例反映出来。

即兴测试试图为解决这些问题搭建了一座桥梁。即兴测试依靠直觉、以前使用产品的经验、平台或技术方面的专业知识以及测试类似产品的经验，探索产品未被发现的区域。即兴测试一般用于发现计划测试没有发现的缺陷。

即兴测试不使用任何测试用例设计技术，例如等价类划分、边界值分析等。

第1章的"杀虫剂悖论"介绍过，农场存活下来的害虫可产生对特定杀虫剂的抗药性。这种情况要求农场主每次在下一个栽培周期内使用一种不同类型的杀虫剂。类似地，产品缺陷逃过了计划测试用例，这些测试用例在下一个测试周期可能也不能发现缺陷，除非引入新的视点。因此，计划测试用例需要不断更新，有时更新间隔时间甚至只有一天，以获收新得到的信息。但是，非常频繁地更新测试用例是（或感觉上是）非常耗时和繁琐的工作。在这种情况下，人们采用即兴测试，则不要求正式和及时地变更测试用例地进行测试。

图10-1给出了即兴测试的各个步骤，并说明了即兴测试和计划测试之间的基本差别。计划测试和即兴测试之间最本质的区别是：即兴测试的测试执行和测试报告生成在测试用例设计之后完成。即兴测试的名称突出了执行先于设计的特点。

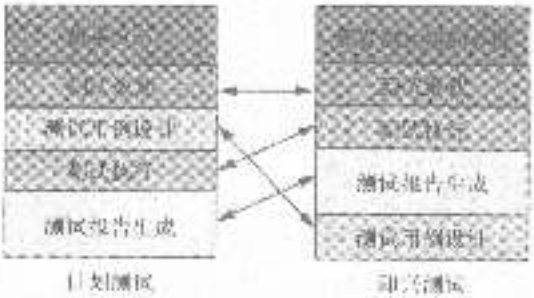


图10-1 即兴测试与计划测试的对比

由于即兴测试引入了新的视角，因此需要知道已有的计划测试活动所覆盖的内容，以及产品中的什么变化会引起测试功能的变化。

与开发人员和其他项目团队成员不断进行交互会更好地从不同视角理解产品。由于即兴测试要求对产品有更好的理解，因此"保持联系"是很重要的。项目评审会是另一个获取产品信息的好场所。

很多时候，由于缺乏沟通，需求变更可能没有通知给测试团队。如果测试工程师不知道需求变更，就有可能遗漏一些测试，导致有些缺陷没有被发现。到了被变更的需求被注意到时，可能已经为时过晚，不能将其在计划测试用例中反映出来。

与开发人员和其他项目团队成员进行交互可能只能帮助得到一组视角，有时这种交互可能使测试团队产生偏见，可能会被开发人员和其他团队成员所说的话牵着走。因此，重要的是要不断对测试用例提出问题，还要与公司外的人（例如客户）进行交互，以发现使用产品的不同方法，并在即兴测试中使用。

前面已经把以前的测试活动叫做"计划测试"。但是这并不意味着即兴测试是一种没有计划的活动。即兴测试是一种有计划的活动，只是测试用例开始时没有写入文档。即兴测试可以随时对产品实施，但是，如果在执行即兴测试之前首先执行计划测试用例，则即兴测试得到的收获会更多。即兴测试可以采用以下两种方式之一策划：

1. 在一定数量的计划测试用例执行以后。在这种情况下，产品可能处于更好的状态，因此可能发现更新的视角和缺陷。由于即兴测试不要求立即将所有测试用例写入文档，因此这提供了以最短的时间延迟获得多个遗漏视角的机会。

2. 在计划测试之前。这样能够更清楚需求，并能更好地评估产品的质量。

两种方式的策划目标都是要使即兴测试成为其他测试活动的补充和互补，以最小的时间延迟发现新的视角。

前面已经讨论过，即兴测试不要求将测试用例写入文档。这只适用于测试执行阶段。测试执行结束后，即兴测试要求将测试过的所有视角作为一组测试用例写入文档。这些测试用例将成为下一轮测试执行的一部分。遵循这种方法可保证两件事：第一，在一轮即兴测试中得到的视角被正式获得、不会丢失。第二，后续轮次的即兴测试将带来新的视角，不会重复已做过的事情。这可以保证即兴测试每次都是以直觉为基础的。显然，如果一次次地覆盖同样的视角，那就不再是即兴测试了。

即兴测试可以用来不断切换软件使用的背景，以在更短的时间内覆盖更多的功能。例如，不是端到端地测试给定功能，即兴测试可能要求测试人员在不同的功能和屏幕之间跳过。这就是所谓的"随机采样测试"。这种测试随机使用软件不同组件的功能，不考虑要测试的是哪个功能及其对每个组件的覆盖情况。由于这种技术类似猴子从一棵树上跳到另一棵树寻找更好的果子（希望如此！），认为给定树上的果子都是类似的，因此叫做"猴子测试"。

前面已经提到即兴测试一般要结合计划测试进行，因此下面的问题就是如何在计划测试和即兴测试之间分配工作量。完全依靠常规测试用例或完全依靠即兴测试都不是好主意，将两者有机地结合起来（如果需求相当稳定和清晰，可以举比方说95%的工作量放在常规测试上，5%的工作量放在即兴测试上）通常比只依靠计划测试或即兴测试更有效。对于需求不够清晰和丢失的情况，要相应地增加即兴测试工作量。

不采用任何形式化测试技术的测试叫做即兴测试。

即兴测试作为一种在即将发布前增加信心的手段，保证测试没有遗漏任何区域。即兴测试可以包括已经被常规测试测试过的区域，并以有计划的方式关注遗漏的区域。这仍然叫做即兴测试，因为测试用例没有写入文档。

虽然即兴测试有很多优点，但还是有一些缺点。表10-1列出了这些缺点及其解决方案。

表10-1　即兴测试的缺点及其解决方案

缺　点	可能的解决方案
很难保证能从即兴测试所了解到的信息能够在未来使用	• 测试后立即将即兴测试用例写入文档
在即兴测试中发现大量缺陷	• 安排一次会议，让缺陷修复 • 改进计划测试的测试用例
不了解即兴测试的理念	• 在缺陷报告时，将计划测试和即兴测试结合起来 • 需求全计划测试和即兴测试
很难跟踪测试步骤	• 以一步一步的方式编写详细缺陷报告 • 测试执行结束后将即兴测试用例写入文档
缺少用于搭桥分析的数据	• 为计划测试和即兴测试采集相关指标数据

即兴测试适用于所有测试阶段。在单元测试期间进行即兴测试可提高需求的清晰性，确定遗漏的代码，并尽早发现缺陷。在组件和集成测试阶段进行即兴测试可以发现以前的计划测试用例没有发现的缺陷。在系统和确认测试阶段进行即兴测试可以增强对产品的信心，发现可能会遗漏并造成损失的缺陷。

在即兴测试下有不同的变种和测试类型，以下分别进行描述。

10.2　伙伴测试

这种测试利用了"伙伴系统"实践，即两个团队成员组成伙伴。伙伴相互帮助，共同的目标是尽早发现缺陷并将其改正。

一个开发人员和一个测试人员通常可以结成伙伴，将人们组织起来形成良好的伙伴工作关系，克服理解上的问题，会具有优势。另一方面，如果把伙伴关系映射到一种两人完全一样的观点和方法，得不到两人之间所要求的多样性，这样会降低测试的有效性。具有良好的工作关系，同时又具有不同的背景，这样的伙伴可以提高尽早发现程序错误的机会。

在伙伴测试期间，伙伴不应感到相互威胁或感到不安全。他们要（如果要求）接受伙伴测试价值观和目标的培训（即尽早找出并改正产品中的缺陷）。应该使他们欣赏彼此的责任。在实际开始测试工作之前，他们还要就工作形式和条款达成一致，保持密切联系，能够执行所约定的计划。

在伙伴测试开始前，代码要经过单元测试，以保证代码完成所期望的工作。代码成功地通过单元测试后，开发人员要接近测试伙伴，在单元测试完成之前开始伙伴测试可能导致伙伴之间就没有满足所描述需求的代码进行长时间的讨论，这反过来会削弱伙伴的信心，造成不必要的返工。

伙伴可以检查编码标准的符合性、合适的变量定义、遗漏代码、足够的线内代码文档、错误检查等。伙伴测试即使用白盒测试方法也使用黑盒测试方法。测试以后，伙伴会编写一份具体的总结评论指出具体的缺陷。这些报告送给开发人员。反馈越具体，开发人员越容易修改缺陷。伙伴在指出工作产品中的错误时，还会就代码修改提出建议。

开发人员研究伙伴的评论。如果伙伴同意，就实现适当的修改。否则，两个人讨论评论并得出结论。在得出结论和采取措施时，不要以自我为中心或个人化地对待缺陷和总结评论。这就是为什么伙伴之间融洽的个人关系是伙伴测试成功的基础。

伙伴测试通过伙伴之间的不同视角或交互式交换思想，有助于避免出现遗漏、误解和沟通不够的错误。伙伴测试不仅有助于发现代码中的错误，还可以帮助测试人员了解代码编写的方式，并提供规格说明的清晰性，有助于作为伙伴的测试人员为后续计划测试制定出更好的测试策略。

> 开发人员和测试人员像伙伴一样地工作，在测试和理解规格说明上相互帮助，这种测试叫做伙伴测试。

伙伴测试一般在单元测试阶段进行，这时编码和测试活动同时进行。伙伴测试针对产品中新的或关键模块，其规格说明对于充当开发人员和测试人员不同角色的伙伴来说不够清楚。

10.3 结对测试

对于这种类型的测试，两个测试人员结成对子，在同一台计算机上测试产品的特性。结对测试的目标是在两名测试人员之间最大限度地交换思想，当一个人在执行测试用例时，另一个人记录，并提出建议或帮助提供额外的视角。

并不要求一个人在整个过程中始终充当一种角色，在测试过程中"测试人员"和"记录员"可以互换角色，可以协商确定具体做法。一个人在一天内的不同测试时点，可以和多个人结成对子。结对测试通常是大约一两个小时的专题讨论，由结对双方决定测试功能的不同方法。

> 举一个例子，两个人在一个新的地区开车寻找一个地方，一个人开车，一个人看着地图进行导航。
>
> 在上面的例子中，如果两个人把想法合在一起，并在两人之间分配具体的导航和开车角色，那么找新地方（就像查找产品待测查区域的缺陷）就会变得容易一些。

结对测试利用了以上刚才介绍的概念。资深成员参与也有助于结对测试。可以缩短学习产品所需的时间，还可以给团队成员提供更好的培训，可以充分理解需求的影响，并向经验较少的人解释。从以上讨论可以得出结论，结对测试的目标是通过交换思想找出缺陷。实践证明，如果两个成员能够很好协作，不断交流对产品的理解，那么结对测试就是很有效的。

在结对测试中，两个测试人员同时使用一台计算机查找产品中的缺陷。

结对测试可以在任何测试阶段开展。最好从需求分析阶段就产生想法，一直延续到设计、编码和测试阶段。在编码阶段，测试人员可以结成对子，产生各种测试代码和各种组件的想法。组件测试完成后，在集成期间，测试人员可以结对测试接口。系统测试期间的结对测试则要保证发现并解决产品级缺陷。缺陷复现对于结对测试变得很容易，因为两个测试人员可以讨论并复现测试步骤。

如果待测产品属于新领域，没有多少人具有所需的领域知识，那么结对测试就派上用场了。结对测试有助于通过结对，由有经验的测试人员对没有经验的成员进行指导。如果项目进度计划安排很紧，可能很难对所有成员进行培训，通过更有经验的成员对新成员提供经常持续的指导，结对测试可以解决这个问题。

当计划测试没有发现关键缺陷时，可以进一步测试产品探索潜在的缺陷。结对测试可以发现那些单人测试很难发现的缺陷。在这种结对测试中发现的缺陷可以由两个成员的代表进行更好的解释，这比单个成员判断缺陷及其背景更好。

结对测试是“结对编程”的扩展，结对编程概念在极限编程模型中使用。结对测试在实践中比结对编程早得多，但是实践证明通过结对概念得到的推动同样适用于编码和测试。

前面已经介绍过，结对测试要求在两个人之间交互和交换思想，正如格言“三个臭皮匠顶个诸葛亮”所说的，结对测试有更多的机会尽早报告重要的缺陷，使项目团队能够尽早修改产品中的重要缺陷。两个成员经常一起发现缺陷。在这种情况下，两个人对需求或产品功能的理解有助于不报告并不重要的缺陷。

由于团队成员在项目生存周期内要与其他人结对，整个项目团队都会更好地相互了解。还使对不同组件的认识在团队内更广泛传播，能够更好地促进团队工作。项目实施会议和团队会议现在有了新的含义，因为现在每个人都关注整个产品的交付（而不是这样的陈腔：“这是我的模块，我的程序，我的测试用例和测试计划”）。

不适合结对测试的情况

讨论了结对测试的优点，下面介绍不适合结对测试的情况。

在结对过程中，两个人中表现好的人会带来问题。有可能在协作过程中，一个人处于主导地位，另一个人处于被动地位，这样可能不会产生预期的结果。如果对子中的两个人不尽力理解和尊重对方，结对测试就可能变成支配关系，造成失败。

当一个成员在计算机上工作，另一个成员充当记录员角色时，如果他们的理解和执行速度不匹配，就会导致出现不和谐的情况，以后阶段也很难改变这种状况。由于花在测试人员交互上的时间过多，结对测试可能造成延迟（如果策划不周）。

有对新手和有经验的成员结对可能导致出现让新手干老手不想干的情况。到了合作结束时，不知道是谁负责主导的测试工作、掌握的方向和提交的测试结果。

10.4　探索式测试

即兴测试中发现缺陷的另一种技术是不断探索产品，覆盖更深更广。探索式测试试图通

过具体的目标，任务和计划做到这一点。探索式测试可以在任何测试阶段期间实施。

探索式测试人员可以根据以往对类似产品或类似领域的产品，或技术范围内产品的测试经验进行测试，还可以充分利用发现以前产品发布版本缺陷的经验，检查同样的问题是否还在当前版本中存在。

开发人员对于类似技术的知识会有助于在单元测试阶段探索该技术带来的限制或约束。探索式测试可以用来测试还没有测试过、未知或不稳定的软件。如果不清楚下一步该怎么测，或不清楚什么时候该深入测试，则可以利用探索式测试。探索式测试不仅针对功能，而且还针对不同的环境、配置参数、测试数据等。

由于在探索式测试中有大量创造性成分，由两个不同的人执行，类似的测试用例可能导致发现不同类型的缺陷。

探索式测试技术

实施探索式测试有很多种方式。常常使用了某些通用技术却没有意识到已经在使用这些技术。

> 举一个例子。有个人没有地图去一个新地区开车去某个地方。他会使用各种常见的开车技术到达那个地方，例如：
>
> - 获得那个地区的地图
> - 随机地向某个方向开以判断出那个地方的位置
> - 打电话向朋友问路
> - 向路过的加油站问路

同样类比可以扩展到探索式测试中，如图10-2所示，有多种方式可以执行探索式测试。

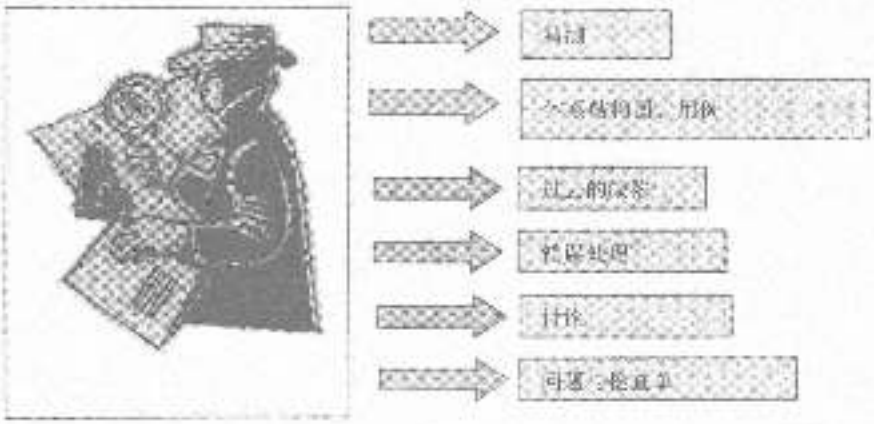


图10-2 探索式测试技术

猜测用来找出可能隐藏更多错误的程序部分。以前在类似产品、软件或技术上的经验有助于猜测，这是因为测试人员已经面对过测试类似产品或软件的情况。根据猜测得到的测试用例用于测试新产品，以检查类似的缺陷。

探索式测试的第二种技术是使用体系结构图和用例。体系结构图描述不同组件和模块之间的交互和关系，用例以使用角度直观地给出产品的使用方式。用例解释一组业务事件、所要求的输入、这些事件所涉及的人以及预测输出。探索式测试可以使用这些体系结构图和用

例测试产品。

探索式测试的第二种技术基于对过去缺陷的研究。研究在以前版本中报告过的缺陷有助于了解在产品开发环境中易出现错误的功能和模块。以前版本的缺陷报告用作进一步探索产品区域的路标。

产品中的错误处理是探索式测试的另一项技术。错误处理是代码的一部分，在出现失效时会输出合适的消息，或采取合适的措施。通过探索式测试可以检查各种场景下的良好的错误处理情况。例如，当出现重大错误时，程序中止时应给出有意义的错误消息。如果用户采取的行动是无效的或未预期的，系统可能出现不当行为，在这种情况下，错误处理会提供消息或更正行动。可以模拟这类情况进行测试，以保证产品的代码考虑了这些问题。

第五种探索式测试技术以通过讨论获得对产品的理解为基础。探索式测试的策划可以依据在项目讨论或会议上获得的对系统的理解。在这些会议上可以获得有关产品不同需求的实现方面的大量信息，可以记录这些信息并在测试时使用。还可以从各种产品实现介绍中获得信息，例如体系结构介绍和设计介绍，甚至包括对客户作的介绍。

所使用的最后一种技术是进行探索用的问卷和检查单。像“什么，何时，如何，谁和为什么”这样的问题可以引导到产品的探索区域。为了理解产品中的功能实现，可以问一些开放性问题，例如“这个模块是做什么的”，“什么时候调用或使用”，“输入如何处理”，“谁是这个模块的用户”等。这类问题可以提供下一步测试内容的线索。

为了进行探索式测试，可以制定详细的测试计划描述要测试的区域、目标，以及需要的时间和工作量。关注点还应该放在测试环境和系统配置参数上。

在执行测试时，要确定可能隐藏更多问题并需要进一步探索的区域。这种探索技术可以认为是输入、环境或系统配置参数的各种组合。

邀请领域专家参与探索式测试可能会得到很好的效果。一组项目团队成员（开发人员，测试工程师，商务代表，技术文件编写者）可以组成团队测试特定的模块或功能。利用他们不同的背景，通过探索式测试可能会发现更多的缺陷。只要有关产品功能没有写入文档的行为或没有答案的问题，以团队的形式进行探索式测试会更有效。

10.5 迭代式测试

第2章介绍过，迭代（或螺旋）模型适用于需求不断提出，产品开发针对每个需求迭代完成。与这种过程相关的测试叫做迭代式测试。

对这种模型测试的最大挑战之一，是要保证已经测试过的所有需求在提交新需求时仍然有效。因此，迭代式测试需要重复测试。当提出新的需求或完成对缺陷的修改时，可能会影响已经测试过的其他需求。由于新需求可能涉及产品的大量变更，这些测试用例的执行绝大多数是手工完成的，因为很难对这类测试用例自动化。迭代式测试针对产品所有需求的测试不考虑处在螺旋模型的哪个阶段。

与瀑布模型不同，所涉及的进度和工作量取决于该轮迭代引入的特性。每轮迭代结束时客户都会有一个可以使用的产品，可以在任何一轮特定迭代上停止产品的开发，将产品作为一个独立实体推向市场。由于每次产品都要经历生存周期的所有阶段，因此由于遗漏和误解产生的错误可以定期得到更正。

客户和管理层可以在每轮迭代结束时注意到缺陷的影响和产品功能。他们可以根据对上一轮的观察，决定是否进入下一轮迭代。

第一轮迭代要制定一份测试计划，后续每轮迭代都对计划进行更新。测试计划大致定义每轮迭代要完成的测试类型和范围。这是因为产品要经过一系列迭代，较后轮次迭代所用的测试时间会显著增长。此外，较后轮次迭代中执行的测试类型可能在较前轮次迭代中不能执行。例如，性能和负载测试只能在最后几轮迭代产品已经完成时进行。因此，测试计划准备成为开始阶段的一项重要活动。测试计划准备要使不同的小组并行工作，每个人都协调同步。每轮迭代后要更新测试计划，因为测试的范围、测试类型和涉及的工作量会发生变化（第15章将详细介绍测试策划）。

开发人员创建单元测试用例，保证所开发的程序经过完整的测试。通过黑盒角度生成单元测试用例，以更彻底地测试产品。每轮迭代结束后，要增加、修改或删除单元测试用例，以与当前阶段经过修正的需求一致。

至少每隔一轮迭代（如只不是每轮迭代）要重复一次回归测试，以保留当前的功能。由于迭代式测试需要重复以前轮次的迭代运行过的测试用例，测试人员会感到厌倦。为了避免这种单调，提高测试效率，只要可能，就应考虑将需要在每轮迭代执行的测试用例自动化。

> 举一个例子，有个人没有地图试图统计某个地区的饭店数量，当他走到多条道路的交汇处时，一次只能走一条路，并统计沿途的饭店，然后再回到交汇点，尝试走另一条路。他可以持续尝试下去，直到搜索完所有的路，并统计了沿途的饭店。

这样，迭代模型关注以短的有规律的时间间隔，少量增加交付产品。对于团队可能出现的一个问题是缺陷修改的协调。在一轮迭代中发现的缺陷根据客户定义的优先级，有可能在同一个版本中修改，也可能向后拖。假设缺陷是在第二轮发现的，直到第五轮迭代才修改。这时这个缺陷可能不再是有效的，由于在第三、第四和第五轮迭代期间需求已发生变化，这个缺陷可能已经不存在了。另一种可能是，在第二轮迭代能正常运行的功能到第五轮迭代已不能正常运行。因此，随着迭代轮次的增加，测试工作量也会增加。

在上面的例子中，统计饭店的数量从所经过的第一条路开始，每轮结束时都可以宣布统计结果并发布。

迭代式测试也使用了同样的概念，针对新需求小步增量地开发产品。下一节将要介绍的敏捷与极限测试也利用了迭代模型的概念。人们有意将这种模型与敏捷模型重叠起来。

10.6 敏捷与极限测试

> 接待员：我们的规定要求，只有证件的发放对象才能来领养系格。
> 来访者：我知道你们的规定，但我是要你们给我的祖父发死亡证明。

敏捷与极限（XP）模型把过程推到极限，以保证及时满足客户的需求。在这种模型中，客户与项目团队携手一步步地分阶段把项目完成。客户成为项目团队的一部分，以便澄清可能存在的任何疑问和问题。

敏捷与极限方法论强调全团队的参与，强调相互间的交互，以生产出能够满足给定特性集合的可使用产品。作为这种交互的结果，所有想法都得到交换。软件以小发布版本的形式

交付，以增量的形式引入特性。因为变更是逐渐引入的，因此对变更的响应就变得很容易。

敏捷测试对测试界的影响是深远的。对于敏捷测试，团队不再作为“一组测试人员”工作了，测试工程师不再需要提交测试文档和缺陷报告，等待来自项目组其他成员的输入。当测试人员和开发人员结合在一起后，他们会从技术角度关注程序的功能。当他们与客户结合时，他们的作用如同产品的领域专家。测试人员成为沟通客户（他们了解业务）和开发人员（他们了解技术）的桥梁，以解释他们不同的视角。这样，测试人员实际上就成了黏合剂，把客户的产品需求观点和开发人员的技术和实现视角结合起来。

典型的XP工作日是从所谓“站立会议”开始的。在每个工作日的开始，团队要碰头决定全天的行动计划。在站立会议上，团队要提出明确的要求和问题，并讨论解决，整个团队都了解每个团队成员所做的工作。测试人员报告测试结果向项目团队报告项目进展。要讨论的其他问题还可以是由于某个问题或前一日的各种活动花去的时间过多，没有达到估计的进度等，站立会议每天都能对变更作出快速响应。

尽管测试人员和开发人员的角色好像不同，但是在XP模型中，这两个角色之间并没有硬性的界限。人们超越边界充当XP模型中的不同角色。高度的沟通和团队协作使这种角色转换成为可能。

10.6.1　XP工作流

在遵循XP方法论的过程中有不同的步骤。本节讨论XP产品发布的过程。XP工作流包含的不同活动有：

1. 开发用户故事
2. 准备确认测试用例
3. 编码
4. 测试
5. 调整
6. 交付

图10-3给出了XP工作流。

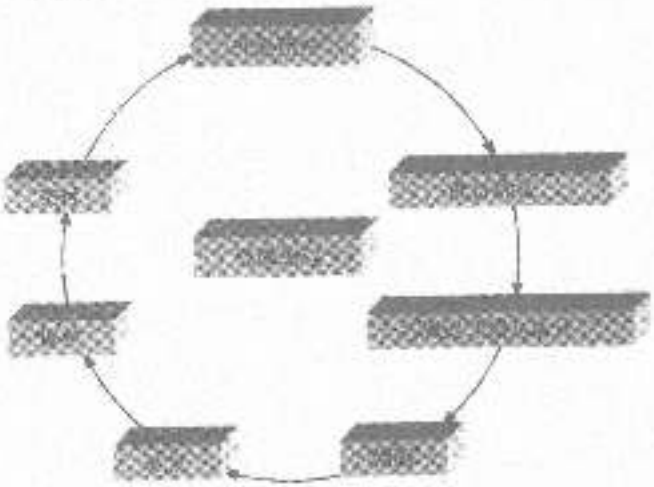


图10-3　XP工作流

开发和理解用户故事

XP项目中的客户需求以故事线的形式给出。这些故事线是描述要开发的特性和功能的短句，故事线写在叫做“索引卡”的小卡片上。

在讨论故事线时，测试工程师要准备关于故事的问题清单，使故事能被参会的所有人完全理解。在回答这些问题时要做记录，在准备确认测试时可能会用到这些记录。

如果要求要做性能和负载测试，单独的故事线有助于提供这些测试的需求小结。

准备确认测试

下一个步骤是根据已经理解了的用户故事准备确认测试。测试工程师与客户一起开发确认测试用例。在有些情况下，客户自己也可以准备确认测试用例。确认测试所需的数据可以由客户提供，以反映实际情况。确认测试可以包含测试场景，端到端地覆盖系统。测试工程师可以把对需求的分析提交给客户，并补充到确认测试用例中。测试人员还可以和客户结合开发负面场景。

开发人员可以评审这些测试用例，以便让其他人知道自己看待这些故事的实现视角。客户评审并认可确认测试用例。根据他们的反馈意见，测试人员细微调整确认测试。一旦这些测试用例形成基线，就用作每个人关于该发布版本需求的参考点。这些测试用例供团队所有成员共享。

在这种交换想法的过程中，测试人员从客户那里获得大量信息。通过与各种团队成员交互获得的知识可以在与开发人员合作时使用。

这个阶段的另一个活动是就该发布版本的确认准则达成共识。这种准则可以基于：

1. 所有缺陷已修改；
2. 执行了的所有测试用例中有x%的通过率，其中x是大家一致同意的数字；
3. 所有高优先级的缺陷已修改。

根据确认准则，可以决定开始该发布版本的下一轮迭代，还是停止工作流。

测试计划与估计

下一步是估计完成测试任务所需的工作量。测试任务所需的工作量要包含到整个项目工作量中。如果估计后发现遗漏或误解了需求，则要向客户解释需要增加工作量。客户可以决定把遗漏的需求留到下一轮迭代还是留到下一个发布版本。客户也可以同意将这些需求包含到当前迭代中，同时同意在当前发布版本中投入额外的工作量。

编码

这个过程的第二步是开始编码。但是在开始编码之前，测试工程师要与开发人员一起准备单元测试用例。测试计划和测试用例要在与不同小组讨论和交互的基础上开发。因为需求要发生变化，因此完成一轮迭代后要重新检查单元测试用例。编写代码时不仅要依据规格说明，还要依据单元测试用例。

测试

开发人员完成编码后，要执行测试用例。测试结果要形成文档，并跟踪在这轮迭代中发现的缺陷。只要缺陷一修改完，测试人员就要与开发人员一起检查缺陷修改情况。在站立会议上要讨论测试状态，以缺陷指标的形式介绍发现细节。缺陷指标更面向能够运行的功能和需要关注的功能，而不是缺陷统计数字。

每轮迭代后要根据修改后的需求更新测试用例，可以删除、修改和新增测试用例。如果必要，测试工程师还要进行性能、压力、安装和负载测试。可以实施回归测试，验证前几轮

迭代正常的功能在本轮迭代中是否依然能正常运行。

调整

在一个小发布版本内的不同迭代期间，要修改需求和变更优先级。为了应对这些变更，需要进行调整。调整是改正现有任何工作产品以反映当前变更的一种方法。在这个阶段，可能需要增加、修改或删除测试用例。最好能吸收客户参加调整测试用例的工作。

自动化

当开发人员忙于针对变化了的环境修改代码或进行缺陷修改时，测试人员可以关注能够自动化的测试用例，以便在下一轮迭代中运行。这会显著降低执行测试的工作量。需要确定适合该发布版本给定用户故事的恰当的自动化策略。

确认与交付

客户运行确认测试用例（与测试人员一起，或独立进行）并记录测试结果。根据测试结果，可以决定是否为下一轮迭代或发布版本开发新的故事。

如前所述，确认是以在项目开始时定下的确认准则为基础的。遗留的缺陷如果不是重要的，可以带入下一轮迭代。取决于修改缺陷所需的工作量，可能有少量没有修改的非重要缺陷能够带入下一轮迭代。客户认可当前迭代的所有没有修改的缺陷。

10.6.2　通过例子进行小结

举一个汽车制造厂的例子。传统的汽车有四个轮子、一个方向盘、刹车、一个加速汽车的踏板和一套传动系统。每种汽车都有一些基本特性。经过几年的发展，会以增量的方式引入新的特性，但是基本特性及其用途没有多少变化。但是，每个汽车制造商在一年内都会多次发布具有新特性的新款汽车。

这些频繁发布的新款汽车使客户喜买，他们不断更新自己的汽车，就这样，汽车工业不断成长、繁荣。

从技术上看，使用操纵杆驱动汽车可能会更容易，但是客户已经习惯使用方向盘开车。重点只能放在客户提出的需求上。

通过反馈机制，总是能从客户那里收集到对汽车的增量需求。客户要测试这些需求，在开发新特性时要融入这些需求。

极限编程和测试使用了以上例子的理念，频繁发布并以受控的方式吸收客户参加。对需求和软件只作小调整以降低成本、时间和工作量。极限编程和测试所采用的政策和理念是：

1. 跨越边界——开发人员和测试人员跨越边界担当不同角色。
2. 增量变更——产品和过程都以增量的方式进化。
3. 轮转前进——开发和测试的负担尽可能地小。
4. 沟通协调——特别关注沟通。
5. 在编码前编写测试——分别在编码和测试活动开始前编写单元测试和确认测试。所有单元测试用例都应该100%地运行。根据测试用例编写代码。
6. 频繁进行小发布。
7. 始终吸收客户参与。

10.7 缺陷播种

错误播种也叫做调查，用作产品发布版本的可靠性指标。

通常由项目组的一部分成员注入缺陷，由另一部分人进行清除。缺陷播种的目的是在发现已知播下缺陷的同时，发现不是播种播下的缺陷。播种的缺陷与实际缺陷类似，因此并不能明显、容易地发现缺陷。

可以播种的缺陷可以是严重或重大缺陷，也可以是轻微错误。这是因为缺陷播种可以用来预测缺陷类型的百分比，这样可以使审查团队很难区分播种的缺陷和实际缺陷。

缺陷播种可以用作检查审查或测试过程效率的指南，还可以用作了解缺陷清除百分比的手段，还可以用作系统中尚未发现缺陷数量的一种估计方法。

> 缺陷播种是一种通过有意向产品中引入缺陷，检查缺陷发现和残留比率的方法。

例如，假设把20个从严重到轻微的错误注入产品中，测试团队完成测试时，发现了12个播种下的缺陷和25个原缺陷。那么在产品中的缺陷总数就是：

缺陷总数＝（播种下的缺陷/已发现的缺陷）×已发现的原缺陷

根据以上公式可以得出估计产品中的缺陷总数为（20/12）×25＝41.67。

根据以上计算可以得出还没有找出的总缺陷数估计为17个。

如果测试小组知道系统中有播种的缺陷，尽可能多地发现这些播种的缺陷对他们是一种挑战。这为他们的测试注入了新的动力。对于手工测试，要在测试过程开始前播种缺陷。如果测试用例已经自动化，那么缺陷播种可以在任何时候进行。

实施缺陷播种时还要注意以下问题：

1. 实施缺陷播种过程时要很小心，要保证在发布产品前清除所有播种的缺陷。

2. 编写代码时要便于很容易地确定所引入的错误。尽可能压缩播种缺陷所增加的代码量，以减少清除播种缺陷的工作量。

3. 不仅需要估计发现播种缺陷的工作量，还要估计清除播种缺陷的工作量。由于一些缺陷的注入，还需要修改实际的缺陷。

值得注意的是，根据已有的缺陷注入缺陷可能不能得到所期望的结果，因为开发人员已经注意到了这类缺陷，并在以前的发布版本中修改过。

注入需求缺陷（例如不完整或遗漏需求）可能是难以做到的。而在很多实际情况下，需求缺陷占总缺陷的比例很大。缺陷播种法对于这类缺陷可能是低效的。

10.8 小结

本章介绍了即兴测试的不同方法。表10-2归纳了这些技术适用的场景。

表10-2 即兴测试的方法和有效性

适用的场景	要考虑的如有效的即兴测试技术
没有十足法测试用例就不适合机器测试产品	独立测试
由开发人员和测试人员共同早期发现编程错误	结伴测试
测试新产品领域技术	探索式测试
充分利用资深测试人员的经验，并后续新人去想法	经验测试
应对不断变化的需求	迭代式测试
频繁发布产品，降低客户与产品开发	敏捷/极限测试
对测试过程的有效性有一个大致了解	缺陷播种

问题与练习

1. 经过一段时间后，即兴测试如何转入计划测试用例？
2. 即兴测试可以是白盒测试吗？如果可以，确定在什么条件下，白盒测试是有用的。
3. 在测试的什么阶段，伙伴测试有可能捕获最多的缺陷？为什么？
4. 实施探索测试需要什么技能？
5. 伙伴测试和结对测试为两类即兴测试。请描述如何将伙伴测试和结对测试结合，达到更有效的效果。
6. 对于以下每种情况，本书介绍的哪种即兴测试方法最合适？
   a. 测试人员和开发人员工作紧密的小公司。
   b. 需要让多个测试人员了解给定模块的测试情况。
   c. 某个通过多个发布版本进化的产品，在发布版本上迭代地增加特性。
   d. 要开发和测试某个创新型产品。

第三部分 特殊测试专题

本书第三部分将讨论特殊的测试问题。第11章讨论面向对象系统的测试，将介绍如何对第二部分描述的测试类型进行调整，以适应面向对象系统测试的需要。第12章讨论可使用性和易获得性测试。满足用户的模糊用户界面预期变得越来越重要。另外，由于法律要求，产品必须适合残疾人士的使用。

第11章 面向对象系统的测试

11.1 引言

前几章讨论了测试的各种阶段和类型，所讨论的工具、技术和过程包括了大多数类型的软件。本章将讨论针对一种特定类型的软件系统——面向对象（OO）系统的适配问题。11.2节将介绍面向对象的术语和概念。本节并不想全面讨论面向对象的概念，而是要承上启下。已经熟悉面向对象术语的读者可以略过这一节。11.3节将从测试角度逐个讨论面向对象的基本概念，以及如何进行有差别地测试。

11.2 面向对象软件入门

本节讨论与测试有关的面向对象系统的一些基本概念。本节不打算全面介绍所有的面向对象概念，而是要突出一些对目前介绍过的测试方法进行修改和补充的主要概念。

较早的语言，例如C语言，叫做面向过程的语言，这可以通过20世纪70年代出版的一本书的书名被很好地表现出来：《算法＋数据结构＝程序》[WIRT-70]。

这些程序设计语言都是以算法为核心的，把程序看作是由算法驱动的，跟踪算法的执行从开始到结束，如图11-1所示。数据是一种由算法操作的外部实体。从本质上看，这类程序设计语言可以刻画为：

1. 数据被认为是与操作或程序独立的；
2. 算法是驱动者，数据是算法的补充。

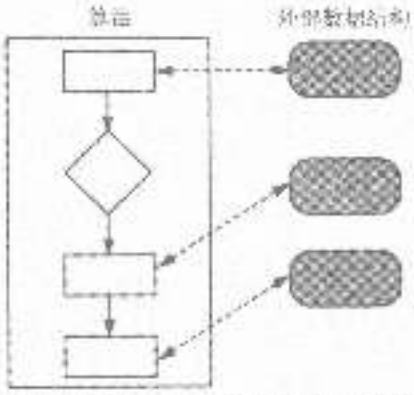


图11-1 传统以算法为核心的程序设计语言

从测试的角度看，测试传统面向过程的系统可归结为测试算法，把数据看作是测试算法流程的附属。

与此相反，面向对象语言和程序设计中有两个基本的范例转换：第一，语言是以数据，即以对象为核心的。第二，如图11-2所示，在数据和操作数据的方法之间没有分离，数据和

操作数据的方法一起构成一个不可见的单元。

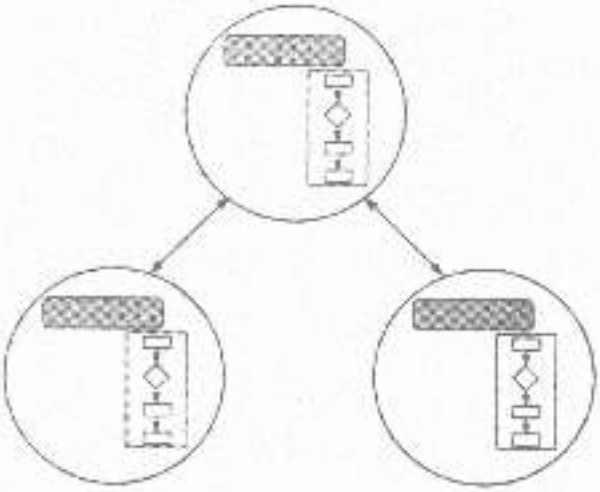


图11.2 以对象为核心的语言——数据和数据操作

类形成面向对象系统的基本构件块。类代表现实世界的对象。每个类（也就是类所代表的现实世界对象）由属性，也就是变量，和操作变量的方法组成。例如，叫做矩形的现实世界对象可用两个属性，即长和宽刻画，面积和周长是可以对矩形对象运算的两个操作，也就是方法。变量和方法合在一起定义一个rectangle类。

例11.1 简单的类定义

```
Class rectangle
  {
    private int length, breadth;
    public:
    new(float length, float breadth)
    {
      this -> length =length;
      this -> breadth = breadth;
    }
    float area()
    {
      return (length * breadth);
    }
    float perimeter()
    {
     return (2*(length+ breadth));
    }
  };
```

现在先不要过于关注例11.1给出的代码段的句法和语义。像例11.1中的类定义只是给出一个模板，指出一个对象的属性和函数。这些属性和方法（即模板）适用于那个类的所有对象。对象的一个具体实例由一个新的实例化创建。对象是类的动态实例化。使用一个给定（静态）类定义可实例化多个对象。这种具体的实例化通过使用构建器函数完成。大多数类都有一个方法（例如叫做new），用来创建类的新实例。创建了新的实例后，使用适当的参数通过向已实例化的对象传递消息，可以调用类的各种方法。例如：rect1.area()或rect2.new(l1,b1)。对于

第一个例子，叫做area的方法不需要任何参数，而在第二个例子中，创建一个新矩形需要描述其长和宽。

例11.2　构建器函数

构建器函数产生类的一个实例。在上面的例子中，new是一个构建器函数。每个类都可以有多个构建器函数，取决于所传递的参数，也就是函数的标志，使用正确的构建器。

因此，在程序中可以有很多rectangle类的实例，这为有不同的rectangle。这意味着实例rect1的length、breadth、area和perimeter与实例rect2对应的变量和方法不同。因此，引用的方式是（例如）rect1.length，rect2.area等。

有人可能会问，"面向对象"有什么好处呢？为什么不能使用传统程序设计语言，定义两个叫做area和perimeter的函数（或子例程），并从代码的任何其他地方调用这些函数呢？基本的差别是，对于面向对象的语言，函数area和perimeter（以及变量length和breadth）如果没有类rectangle的具体实例rect1是不能生存的。而对于传统程序设计语言，如果把area和perimeter定义为两个函数（把变量length和breadth定义为某个全局数据结构的部件），那么这两个函数的存在不依赖于对象rectangle的实例。这可能导致不可预知的结果。例如，变量length和breadth可能会以未料到的（并且是不正确的）方式操作，函数area和perimeter也有可能被不恰当的对象调用。而对于面向对象的语言，由于变量和方法是对象的特定实例专属的，因此可以获得对变量和方法的更好保护。

并不是所有数据、也不是所有方法都在类之外公开可见。有些数据、有些方法对象在类的内部。这保证对方法的实现是标准的数据和方法不能被外部世界访问。这样就提高了数据项转换的可预测性。此外，公共方法只提供能够用于对象内容的操作。这进一步降低了意外或恶意修改对象内容的机会。

方法是贴在对象上的，并不独立存在。因此，方法只提供可以对该对象进行的操作。换句话说，类的用户只知道面积或周长计算的外部接口，不知道有关方法实现的细节。也就是说，方法的实现对用户是隐藏的。这使得编写方法的人可以优化实现而不改变外部行为。这叫做封装性。

组成对象的方法和变量（或数据结构）都可以封装。当变量封装时，访问被封装变量的唯一方法是通过对象内部，这种封装变量在对象之外是看不到的。这进一步保护变量不会受意外修改。

现实生活中的不同对象可以刻画每个对象的不同特征。例如，矩形需要两个参数（长和宽）来刻画，正方形只需要一个参数即边长刻画。类似地，圆也由一个参数即半径刻画。但是所有这些对象，不管有什么差别，都有一些共同的特征——所有这些对象都是平面图形，都有两个叫做周长（即对象边界长度的总和度量）和面积（即图形占据的平面面积的度量）的参数。不同形状的对象有不同的面积或周长的计算方法，因此，叫做plane objects的通用对象（类）有两个叫做perimeter和area的适用于所有平面图形的属性，即使面积和周长的计算方法随平面对象的不同而不同。特定类型的平面图形（例如rectangle）从其父对象（平面对象）继承了这些函数，并根据需要进行修改，以适应其特殊需要。类似地，另一个平面图形圆也继承了两个函数，并进行了重定义以适应其需要（例如，周长=2πr，面积=r^2π）。

例11.3　封装

封装提供对外部世界合适修改实变量和方法的抽象。在上面的例子中，length和breadth（矩形的属性）是私用变量，即它们不能被从类以调用的程序直接访问。方法new、area和perimeter（用矩形描述）是公共方法，可

以从类的外部调用。

```
Class rectangle
  {
    private int length, breadth;
    public:
    new(float length, float breadth)
    {
      this->length=length;
      this->breadth=breadth
    }
    float area()
    {
      return (length*breadth);
    }
    float perimeter()
    {
      return (2*(length+breadth));
    }
};
```

面向对象系统的一个主要优点是能够通过已有的类定义新的类，新类的有些属性与已有的类类似，有些属性不同。这种能力叫做继承性。原来的类叫做父类（或超类），新类叫做子类（或导出类）。

例11.4 说明继承的例子

继承能够不同类间共享特性或从一个类导出另一个类。以下例子不是把rectangle、circle看作是不相关的图形，而是把它们看作从一个叫做plane objects的类中导出的。不管是什么对象，都有叫做area和perimeter的属性。

```
Class plane_objects
{
  public float area();
  public float perimeter();
  Class rectangle
    {
      Private float length, breadth;
      public:
        new(float length, float breadth)
        {
          this->length=length;
          this->breadth=breadth;
        }
        float area()
        {
          return (length*breadth);
        }
        float perimeter()
        {
          return(2*(length+breadth));
        }
    };
Class circle
    {
```

```
    private float radius;
    public:
      new (float radius);
      {
        this->radius=radius;
      }
      float area()
      {
        return (22/7*radius*radius);
      }
      float perimeter()
      {
        return (2*22/7*radius);
      }
  }
}
```

继承使对象（或至少对象的部件）可以重用。导出类继承了父类的性质——事实上，继承了所有父类的性质。因为类可以有很多层次，因此，由于这些可以被继承和原封不动使用的父类性质，开发和测试成本可以降低。

在上面的例子中，对象rectangle继承了通用类plane objects的性质，可以看出其两个导出类（rectangle和circle）都有叫做area和perimeter的方法，具体的输入和处理逻辑对circle的方法和rectangle的方法是不同的，即使方法的名字是相同的。这意味着即使方法名称对于circle和rectangle是相同的，其实际含义也要取决于类被调用的具体背景。两个方法的性质在不同的类中有相同的名称，但却引用不同的函数。这叫做多态性。

从以上讨论推论，方法要与对象关联。描述与方法关联的对象的一种方法是直接描述，例如rect1.area()。在这种情况下，很容易看出对哪个类调用哪个具体方法。一种更精细的变种叫做动态绑定。

假设有一个叫做ptr的变量，在运行时被分配了一个对象的地址。在这种情况下，仅看代码还不能确定要调用哪个方法，只有在运行时才能知道ptr的值，因此测试不能对这种代码进行静态分析。下一节介绍多态性和动态绑定对测试带来的挑战。

如前所述，面向过程的程序设计是以算法为核心的，“主”程序的控制流自顶向下，并对数据结构进行操作。但是，面向对象的程序集成了数据和方法，控制流被在不同对象之间传递消息所取代。消息不是别的，只不过是通过传递合适的参数调用类（即对象）实例的方法。

这种范例从以算法为核心，基于控制，面向过程的程序设计，转移到以对象为核心，基于消息，面向对象的程序设计，改变了编写程序或测试的方式。

第二点不同是，在基于消息的方法中，只有创建了实例之后，这些消息才能传递给实例。传递给没有实例化的对象的消息会产生一个运行时错误。本书已经介绍过的静态测试方法很难捕获这类错误。

控制流和面向对象系统测试的另外一点不同源自例外处理。每个类都可能有一组在出现错误条件下启动的例外，以应对错误的消息和条件。例如，如果传递给对象的参数是无效的，该对象就会启动一个例外。这种对例外处理代码的控制转移会造成程序流序列的中断，需要进行测试。如果类嵌套在其他类里，就有可能有嵌套例外。重要的是对象要通过沿嵌套层次执行对应正确例外的代码。如果程序员不知道多层嵌套之间的交互，那么这些例外可能产生不期望的结果。测试各种嵌套例外是非常重要的。

11.3 面向对象测试的差别

上一节介绍了面向对象程序设计的各种突出特点，下面讨论这些特点如何对测试产生影响。

从测试的角度看，这意味着测试面向对象系统应该将数据和算法紧密联系起来，推动面向过程语言测试类型的数据和算法的分离，必须被打破。

面向对象系统测试大致有以下主题：

1. 类的单元测试
2. 把类放在一起运行（类的集成测试）
3. 系统测试
4. 回归测试
5. 面向对象系统的测试工具

11.3.1 一组类的单元测试

由于类是在"发布"给其他程序使用之前构建的，因此要对类进行测试，检查类是否已经可以使用。类是整个面向对象系统的构件块。就像面向过程系统的构件块要在组装之前进行单独的单元测试一样，类也要进行单元测试。本节要介绍对这些面向对象构件块进行甚至更彻底的单元测试的特殊原因，然后介绍适用于面向对象系统的传统测试方法，最后介绍针对面向对象系统的独特测试技术和方法。

类必须首先进行单独测试的原因

对于面向对象系统，对构件块（类）进行彻底的单元测试甚至更重要（与面向过程系统相比）的原因是：

1. 类通常要大量重用。因此，类中的残留缺陷潜在地会影响重用的所有实例。

2. 在定义类（即属性和方法）时会引入很多缺陷。延迟发现这些缺陷会使其进入这些类的客户中。因此，对缺陷的修改会在多处反映出来，产生不一致性。

3. 类可能有不同的特性，类的不同客户可能使用类的不同片段。没有一个客户能够独自使用类的所有部分。因此，除非先把类作为一个单元测试，否则会有类的一些片段永远也得不到测试。

4. 类是数据和算法的一种组合。如果方法和数据不能在单元测试级同步工作，就有可能产生以后很难查找的缺陷。

5. 与过程语言构件块不同，面向对象系统具有像继承这样的特性，这些特性把更多的"背景"放入构件块中。因此，除非单独彻底地测试构件块，否则缺陷会在生存周期的后期从这些背景中露出来，并放大很多倍。

适用于类测试的传统方法

前面讨论过的有些单元测试方法可以直接用于类测试。例如：

1. 每个类都包含变量。在讨论黑盒测试时介绍的边界值分析和等价类划分都可以使用，以保证使用最有效的测试数据发现尽可能多的缺陷。

2. 前面已经介绍过，并不是所有方法都要由所有客户执行。可以使用在讨论白盒测试时介绍过的功能覆盖方法，以保证每个方法（功能）都能执行。

3. 每个类都拥有具有过程逻辑的方法。在讨论白盒测试时介绍过的条件覆盖技术、分支

覆盖技术、代码复杂性分析等都可以使用，以保证覆盖尽可能多的分支和条件，增加代码的可维护性。

4. 由于类要由不同的客户实例化很多次，第6章讨论的各种压力测试技术都可以实施，以尽早发现与压力有关的问题，例如内存泄漏，进行系统测试和验收测试。

第4章讨论过基于状态的测试，这种方法尤其适合测试类。由于类是数据和对数据进行操作的方法的组合，有些情况下可以把类可视化为通过不同状态的对象。传递到类的消息是触发状态转移的输入。在设计阶段获得这种视图是很有用的，因为测试可以更自然。可以用于测试的一些准则包括：

- 每个状态是否至少到达一次？
- 是否生成和测试了每个消息（即引起状态转移的输入）？
- 每个状态转移是否出现过至少一次？
- 是否测试过非法的状态转移？

类测试的特殊考虑

以上方法是来自面向过程系统的通用方法，针对被类实例化的对象性质（这些对象必须通过消息传递测试），如何在单元级测试这些实例？

为了测试经过实例化的对象，必须将消息传递给各种方法，以什么顺序把消息传递给对象？一种达到这个目的的有效方法是阿尔法－欧米嘎方法。这种方法遵循以下原则：

1. "从生到死"全面测试对象的生存周期（即从实例化到销毁）。实例通过构建器方法进行实例化，然后变量被赋值。在执行过程中，可能会修改这些值并且执行各种方法。最后，该实例被销毁器方法销毁。

2. 首先测试简单方法，然后测试更复杂的方法。由于构建面向对象系统的原理是构建一些可重用的对象，因此更复杂的方法很可能是在简单方法的基础上构建的。因此，在测试复杂方法之前首先测试简单方法是明智的。

3. 先测试私用方法，后测试公用方法。私用方法是不能被对象/类的外部看到的方法，因此，私用方法是面向实现的方法，负责处理方法的逻辑，是整个系统的构件块。另外，私用方法是与调用方（即客户）隔离的，这会降低测试的依赖性，使构件块在客户使用前更具健壮性。

4. 给每个方法发送消息至少一次。这可以保证每个方法至少被测试一次。

阿尔法－欧米嘎方法通过以下步骤可以达到以上目标：

1. 首先测试构建器方法。每个类都可能被多个构建器消息根据标志构建，这些是创建类的实例的不同途径。如果有多个构建器，则需要单独测试所有的构建器方法。

2. 测试get方法或accessor方法。accessor方法检索对象中的变量值供发出调用的程序使用。这个方法可以保证类定义中的变量可以被合适的方法访问。

3. 测试修改对象变量的方法。有些方法测试变量的内容，有些方法设置/更新变量的内容，有些方法循环处理各种变量。可以推测，这些方法越来越复杂，请记住前面说到过的原则。

4. 最后，对象必然要被销毁。当销毁对象后，不能对该对象进行意外访问。另外，被这个对象实例使用的资源都应该释放。这些测试结束了被实例化对象的生命。

还有一些挑战是类测试独有，而面向过程系统的单元测试所没有的。下面讨论这些挑战。

前面已经介绍过，封装是对类的客户隐藏类细节的手段。从实现和使用角度看这是很好的。但是从测试角度看却会增加难度，因为被封装部分的内部行为对测试人员的可视性降低。

对于面向过程的语言，人们可以进入实现的内部，能够更清楚地看到程序的行为。没有了这种便利，带有封装的类的白盒测试会变得很困难。

前面已经介绍过，类实际上可以是类层次的一部分，类可以：

1. 从父类继承一部分变量和方法；

2. 对父类的一部分变量和方法进行重新定义；

3. 定义自己专用、父类不能使用的变量和方法。

由于类由所有以上三类的变量和方法组成，因此严格地说，每个新类都必须测试所有的变量和方法。但是在现实中，更增量化的方法可能更有效、更高效。在第一次引入类的时候，必须使用已经讨论过的传统单元测试手段全面测试所有的变量和方法。以后，每次类被从父类导出时都需要测试以下内容，因为它们是第一次出现：

1. 对新类变量、方法和属性的修改必须再次测试，因为已经发生变化。

2. 引入到继承类的新变量和方法需要再次测试。

对于第一种情况，也就是经过修改的属性，用于父类的现有测试用例可能可以重用，也可能不能重用。在前面讨论过的平面图形例子中，即使圆的面积和周长已经测试过，但是对于针对矩形的同一个方法却不能说明任何问题，尽管两者都是从同一个父类导出的。

很显然，在子类中对于对类属性的所有修改和补充都必须进行独立测试，但问题是，如何处理从父类继承且没有被子类修改的属性？严格地说，不需要再次测试，因为理论上这些内容没有改变。但是这些没有改变的变量和方法在与一些变更混合后，可能有一些不期望的副作用，因此（没有修改过的）父类元素要和导出类一起进行有选择性的重新测试，那么如何确定应该重新测试的未改变的元素呢？以下给出的是一些可能的选择：

1. 只要未改变的变量在新的或已改变过的方法中引用，则针对该未改变变量的测试用例就是重新测试的候选测试用例，这可以发现任何未改变变量的非有意使用。

2. 只要未修改方法在新的或修改过的方法中调用，则可以考虑对这个未修改方法进行重新测试。如果新的或修改过的方法没有得到正确的结果，则可能说明未修改方法也许要在包含该新方法的子类中重新定义，也可能要生成原来的方法，以便适应新子类的需求。

在创建时彻底测试所有变更或新增内容，有选择性地重新测试其他未改变的属性，这种方法叫做增量式系统测试。这种方法既考虑了穷尽测试，也考虑了与风险关联的不测试（表面）没有改变的内容。

虽然继承使通过已有对象定义新对象更容易一些，但是继承也是一种潜在的缺陷源。请考虑一个具有多层嵌套的类，最为后的类可能只有很少的代码，但是可能继承类层次结构内上层类的大量变量和方法，这就味着有大量背景内容构成了这个子类，而这种背景内容不能通过孤立地检查这个类来确定，这与在面向过程语言中使用全局变量的情况类似。由于被嵌套的类可以随意访问其父类的方法和变量，因此测试嵌套类就需要访问其父类的信息。

还有其他两种形式的类和继承对测试带来特殊挑战——多重继承和抽象类。

到目前为止讨论过的例子都假设子类可以只从一个紧邻的父类导出。有些语言支持所谓多重继承，即子类通过两个父类导出，很像人类子女从双亲那里得到遗传。这种多重继承性质为测试带来很有意思的东西。例如，考虑一个通过两个父类P1和P2导出的子类A。很可能P1和P2都有名字相同但是完成不同功能的变量和方法，假设P1和P2都有叫做X的方法（完成不同的功能）。当子类从这两个父类继承时，子类可能：

1. 使用P1或P2的X，或

2. 自己修改X，使修改后的X作为X的默认含义，取代来自P1和P2的X。

第二种情况类似在单继承中经过变更的类，因此可能需要测试。对于第一种情况，X没有改变，但是可以考虑进行重新测试。由于多重继承，所以产生副作用的可能性成倍增加，缺陷的范围更大。因此从测试角度看，多重测试需要更彻底的测试。

有时必须存在特定的带有公开接口的方法，以重新定义类，但是这个方法的具体实现则完全留给实现者。例如，考虑对整数数组进行排序，并在另一个数组中返回排好序的列表的方法。对排序例程的接口明确定义：一个输入整数数组和一个输出整数数组。这种方法叫做*虚方法*。具有虚方法的类叫做*抽象类*。对从父类继承的每个新子类都必须重新实现虚方法。

抽象类和虚拟函数给测试带来什么意义呢？抽象类不能直接实例化，因为抽象类不完整，只有虚拟函数的位置。必须通过抽象类定义没有虚拟函数的具体类，要测试这些具体类的实例。由于对于不同的具体类向一个虚拟函数可以不同地实现，因此针对抽象类不同实现的测试用例一般来说不能重用。但是，虚拟函数和抽象类带给测试的好处是，提供了函数应该满足的接口定义。这种接口对于不同的实现应该是不变的，因此，这种接口提供了测试具体类的一个很好的切入点。

11.3.2　将类组合在一起——集成测试

以上所有讨论都是在类层次上进行测试。面向对象系统不是分立对象或类的集合，这些对象或类应该共存、集成并相互通信。由于面向对象系统在设计上有较小的针对重用（经过必要的重新定义）的组件或类构成，因此一旦基本类本身已经彻底测试，类是否能够在一起运行就成为测试的下一步。更多的情况是，不是一个单独的类作为一个单元进行测试，而是一组永远都在一起运行的有关的类，这与面向过程语言没有多大差别。对于面向过程语言，单元可能并不总是一个源文件，而是完成相关功能的一组相关文件作为一个单元测试。对于面向对象系统，由于测试重点是重用和类，因此测试这种集成单元是至关重要的。

对于面向过程的系统，测试是通过给出不同的数据检验控制流路径完成的。这些控制流路径自始至终是由程序调用的函数决定的。如前所述，在面向对象系统中，类相互之间通信的各种方式都通过消息。消息具有以下格式：

```
<实例名>.<方法名>.<变量>
```

对于有名称的实例调用具有指定名称的方法，或通过合适的变量调用（合适类的）对象。因此，不能通过列出执行要经过的函数名来描述要测试的流程。事实上，在测试面向对象系统时，方法名没有唯一地确定控制流。前面已经介绍过，这种函数或操作符的含义随背景的不同而变化，同一个操作在不同的条件下行为各异的性质叫做*多态性*。从测试的观点看，多态性是个很大的挑战，因为多态性推翻了代码覆盖和代码静态审查的传统定义。例如，如果有两个叫做square和circle的类都有一个叫做area的方法。即使函数在两个类中都叫做area，即使两个函数都只接受一个参数，但是取决于调用方法的背景，参数的含义是不同的（对于圆是半径，对于正方形是边长）。方法的行为对于这两个类也是完全不同的。因此，如果针对正方形测试了方法area，并不意味着area方法对于圆也是正确的。需要独立测试。

多态性的一种叫做*动态绑定*的变种也为测试带来很大挑战。在程序代码中，如果显式地引用square.area和circle.area，那么测试人员显然知道这是两个不同的函数，因此需要根据所使用的背景条件进行测试。对于动态绑定，要接收消息的类在运行时描述，这对于允许使用指针（例如C++）的语言来说，是对测试的一个很大挑战。假设指向一个特定对象的指针存

在叫做ptr的指针变量里，那么ptr->area(i)要在运行时解析由ptr指向的合适对象类型的area方法。如果ptr指向一个square对象，那么调用的就是square.area(i)（i就是这个正方形的边）。如果ptr指向一个circle对象，那么调用的就是circle.area(i)（i就是这个圆的半径）。这意味着像代码覆盖这样的白盒测试策略在这种情况下就没有什么用了。在上面的例子中，通过用指向一个square对象的ptr就可以达到对ptr->area(i)的代码覆盖。但是，如果ptr没有测试过指向circle对象的情况，那么计算圆面积的那部分代码就完全没有测试，尽管调用程序中的代码已经被测试用例覆盖。

除了封装和多态性，另一个问题是以什么顺序将类放在一起测试？这个问题与在面向过程系统集成测试中遇到的问题类似。像自顶向下、自底向上、大爆炸等各种集成方法对于面向对象系统都适用。在面向对象系统集成测试需要额外注意的几点是：

1. 面向对象系统本质上是要通过小的、可重用的组件构建，因此，集成测试对于面向对象系统来说更重要。

2. 面向对象系统下层组件的开发一般更具并行性，因此对频繁集成的要求更高。

3. 由于并行性提高，集成测试时需要考虑类的完成顺序，也需要设计桩模块和驱动器来模拟还没有完成的类的功能。

11.3.3 面向对象系统的系统测试与互操作

面向对象系统从设计上要通过较小的可重用组件（也就是类）构建。这种对现有构件块重用的强调使系统测试对于面向对象系统比传统系统变得更重要。这是因为：

1. 类可能有不同的部分，并不是所有部分都同时使用。当不同的客户开始使用某个类时，它们可能使用类的不同部分，这可能会在以后阶段（系统测试）引入缺陷。

2. 不同的类可以由客户组合在一起，这种组合可能导致还没被发现的新缺陷。

3. 实例化后的对象可能没有释放所分配的所有资源，导致内存泄漏和相关问题，这些问题只能在系统测试阶段才能暴露出来。

第5章介绍的不同类型的集成也适用于面向对象系统。重要的是要保证类和对象能互操作，并可以作为一个系统运行。由于类之间交互的复杂性可能是很微妙的，重要的是要保证在进行系统测试之前首先完成合适的单元和组件测试。因此，要在系统测试之前确定各种测试阶段合适的进入和退出准则，以最大限度地提高系统测试的有效性。

11.3.4 面向对象系统的回归测试

将集成测试的讨论再向前推进一步，回归测试对于面向对象系统非常重要。作为面向对象系统强调依赖可重用组件的结果，对任何组件的变更都可能对使用该组件的客户引入潜在的副作用。因此，对于面向对象系统测试来说，频繁运行集成和回归测试用例是很有必要的。此外，由于继承等性质导致的变更级联效应，尽可能早地捕获缺陷是很有意义的。

11.3.5 面向对象系统的测试工具

有一些工具可以帮助面向对象系统的测试，包括：

1. 用例

2. 类图

3. 序列图

4. 状态图

以下逐一介绍每种工具。

用例

第4章在讨论黑盒测试时介绍过用例。用例表示用户在与系统进行交互时将完成的各种任务。用例给出用户完成每个任务的具体步骤细节，以及系统对每个步骤的响应。这符合面向对象范例，因为任务和响应都是通过消息传递给各种对象的。

类图

类图表示不同的实体和实体之间的关系。由于类是面向对象系统的基本构件块，因此类图是以系统的类为基础的。类图有几个部件，这里从测试角度出发，列出几个重要的部件。

类图有以下要素：

方块　每个矩形方块表示一个类。构成类的各种要素在类矩形内的间隔处表示。

关联　通过连线表示两个类之间的关系。关系可以是"每个员工在且仅在一个部门工作"，或"员工可以参加零或多个项目"等。因此，关联可以是一对一、一对多、多对一等。任何一边的都显示在关联连线上。

通用化　表示从父类导出的子类，如本章前面讨论的那样。

类图在多个方面对测试很有用：

1. 类图确定类的元素，因此可以进行边界值分析、等价类划分等，以及对应的测试。

2. 关联有助于确定针对跨类的引用完整性约束的测试用例。

3. 通用化有助于确定类的层次结构，如果有新的变量和方法引入子类，通用化有助于递增的类测试。

序列图

前面介绍过，面向对象系统通过在各种对象之间传递消息来运行。序列图表示对象间传递消息完成给定应用场景或用例的序列。

序列图横向列出参与一个任务或用例的对象。对象的生存周期由自顶向下的纵向线条表示，顶部的虚线表示对象构造/激活，末端的X表示对象的销毁。

两个对象之间的横线表示消息。有不同类型的消息。消息可以是模块化的，也可以是非模块化的。有些消息的传递是有条件的，就像程序设计语言中的IF语句，条件消息要根据一定的布尔条件决定传递给不同的对象。

在序列图中，时间流逝方向为自顶向下。

序列图在以下方面对测试有帮助：

1. 确定各点的端到端消息。

2. 跟踪端到端事务中的中间点，因此能够更容易地缩小确定问题的范围。

3. 提供多种典型的消息调用序列，包括模块化调用和非模块化调用等。

序列图对于测试也有局限性：即使可能，描述复杂交互也会很混乱，很难表示动态绑定。

活动图

序列图表示消息序列，而活动图表示所发生的活动序列。活动图用于对应用程序中典型的工作流建模，描述手工和自动过程之间的交互要素。由于活动图表示活动序列，因此它与流程图类似，与传统流程图的大多数要素对应。

完整的工作流可以通过一组活动状态可视化，每个活动状态表示结果的一个对应用程序

有意义的中间状态。这与步骤序列的标准流程图要素一样，在步骤之间没有条件分支。与传统流程图一样，决策框采用菱形框表示，每个决策框有两个退出通路（一个对应决策布尔条件为TRUE，一个对应FALSE），在运行时只选择一个通路。由于对象是消息传递的目标和行动的起因，因此对象也在活动图中表示。活动通过控制流相互关联，控制从前一个活动流向下一个活动。活动也可以通过消息流相互关联，消息从一个活动状态发送给一个对象。由于判断或多控制流有可能产生多个分支，因此需要在以后同步。

由于活动图表示控制流，因此在以下方面有助于测试：

1. 通过执行导出各种通路的能力，与讨论白盒测试时介绍过的流程图类似，活动图也可以用于确定程序代码的代码复杂度和独立路径。

2. 确定活动和对象之间可能的消息流的能力，因此使前面介绍过的基于消息的测试更健壮、有效。

状态图

本章前面已经介绍了状态转移图对测试的帮助。如果对象可以建模为一个状态机，那么介绍白盒测试时讨论过的和本章前面介绍的基于状态的测试技术都可以直接使用。

11.3.6 小结

本章介绍了有关面向对象系统测试的相关概念，介绍了前面几章讨论的通用技术如何在面向对象测试中适配使用。表11-1归纳了本章的结论。

表11-1 针对关键面向对象概念的测试方法和工具

关键面向对象概念	测试方法与工具
面向对象	测试用例应基于标准继承和聚合原则
类的单元测试	边界值分析、等价类划分等方法用于测试类属性 代码覆盖方法用于测试方法 随机法——从类的方法用于测试方法 活动图用于测试方法 状态图用于测试类的状态 应为测试用例应反复实例化、销毁类，检验内存泄漏和类的状态
封装与继承	要求子类级进行单元测试，对父类进行增量式类测试 继承引入新的背景，必须重测不同背景的行为 由于有额外背景，应重新考虑单元测试的范围
抽象类	要求为抽象类的每个派生类进行单元测试
多态性	需要单独测试每个同名的多态方法 代码应可保护性公下降
状态测试	传统代码覆盖率测试方法不能用于状态测试 虽然日平预料的运行时顺序的可能性增加
通过消息进行的对象间的通信	消息序列 序列图
对象集成与对象的测试开发	需要多级集成测试来自顶向测试 与面向过程的一不同，要求测试单元测试之间的界限不分明 对象之间的接口错误在面向对象系统中更常见，因此需要进行接口测试

问题与练习

1. 不再分离数据和算法的思想被认为是以算法为核心的系统和以对象为核心的系统之间的一个主要差别。从测试角度看，这种差别会带来什么挑战？
2. 测试数据生成器在面向对象系统测试中起什么作用？
3. 在本章给出的平面图形例子中没有方法的重新定义，请考虑子类重新定义了父类的方法。在测试中应该考虑哪些问题？
4. "不支持使用指针的面向对象语言使测试更容易"——请对这句话作出评论。
5. 请考虑一个嵌套五层的类，每层只重新定义一个方法。测试这样的类和实例化对象会遇到什么问题？
6. 为什么集成测试和系统测试对于面向对象系统尤其重要？
7. 与传统面向过程的系统相比，面向对象系统的回归测试的方法有哪些不同？

第12章 可使用性与易获得性测试

12.1 可使用性测试的定义

可使用性测试试图从用户视点刻画产品的“外观和感觉”以及用法。前面讨论过的大多数测试类型都是客观的。有人认为可使用性测试根本就不属于测试范畴。这场争论的焦点因素主要有：

1. 可使用性和外观感觉问题本质上是主观的，并不总能客观地度量。

2. 对“高可使用性”的理解因人而异。例如，系统管理员或开发人员会觉得使用命令行和使用用户界面一样好，而最终用户会要求所有操作都要通过GUI要素，例如菜单、对话框等完成。

3. 用户界面可以看作是一种设计时的活动。如果某种用户界面不能满足用户要求，问题的根源在于没有采集到合适的需求，或没有把需求转换成合适的设计。

基于以上原因，有一种观点认为可使用性只能确认，不能测试。不管可使用性测试在语义上是测试活动还是确认活动，“可使用性测试”或“可使用性确认”的一些特性包括：

1. 可使用性测试从用户的视角测试产品。可使用性测试包括确定用户如何与产品交互，即使用产品的一系列技术。

2. 可使用性测试要检查各类用户是否能很容易地使用产品。

3. 可使用性测试是从使用愉悦性和美学角度确定产品的用户界面和人员用户需求之间的不适的过程。

如果把以上决定可使用性测试的各种要素特征合并，可得到以下共同线索：

1. 易用性

2. 速度

3. 使用愉悦性和美感

> 以用户视点确认产品的易用性、速度和美感的测试叫做可使用性测试。

可使用性测试从用户角度描述这三个方面的问题。综合这些特征可以正式定义可使用性测试。

根据以上定义可以很容易得出以下结论：

1. 由于一个产品可能有不同类型的用户，因此对一个用户很容易的产品对另一个用户却不一定。

2. 一个用户认为（比方说响应时间）快，可能另一个用户却可能认为慢，因为他们使用的机器和对速度的预期是不同的。

3. 有些人认为漂亮的产品另一些人却可能认为很丑。

4. 一个用户对产品的观点可能不是另一个用户的观点。

由于这些原因，可使用性像前面提到的那样仍然是主观的。但是，如果产品结合了整个用户群的不同观点和需求，就可能成为成功的产品。在整个软件界，可使用性测试变得越来越重要，因为产品可使用性的敏感性在增加，如果不能满足用户的可使用性需求就很难销售出产品。有一些标准（例如易获得性指南）、组织、工具（例如Microsoft Magnifier）和过程可以减弱可使用性测试的主观性，提高客观性。

12.2　可使用性测试的途径

在进行可使用性测试时，一些人员因素可以定量表示，并能够客观地测试。完成一个任务所需的鼠标点击次数，要选择的子菜单个数、键击次数，命令数量都可以度量和检查，并作为可使用性测试的一部分。显然，如果完成一个用户任务所需的鼠标点击次数过多或命令过多，就不能认为产品是易于使用的。

有时只提高一点可使用性就会得到很大好处，使产品的用户数增加。例如，Philips“十字”螺丝刀的发明，与“一字”螺丝刀相比，每转动一次螺丝刀只节省千分之几秒，如图12-1所示，但是有很多用户重复地使用，可以节约很多人年的工作。这就是为什么很小的可使用性提高对用户就会意味着很大的可使用性优势。我们把这个例子推广到帮助数据录入操作员采集用户订单的软件产品。如果可以降低每个订单处理的鼠标点击次数，比方说三次，那么如果要输入几十万个订单，那么可以节约的总时间就很可观了。

图12-1　Philips“十字”螺丝刀与“一字”螺丝刀的比较

可使用性测试不仅要针对产品的二进制代码或可执行代码，还包括文档和其他随产品一起提交的可交付产品，也要进行可使用性测试。例如，把发布媒介插入计算机后自动设置产品的典型AUTORUN脚本。当时这种脚本是特定操作系统版专用的，可能不能在其他操作系统版本上自动执行。即使用户可以通过手工点击鼠标来进行设置，但是这种额外点击（事实上产品并不能算是自动安装的）对于执行安装的用户来说可能就会有不好的印象。

一般来说，最适合实施可使用性测试的人是：

1. 将使用该产品的实际用户群的典型代表，以便获取典型的用户模式。

2. 第一次接触该产品的人，从一开始就没有任何偏见，能够找出可使用性问题。多次使用过该产品的人可能会看不到产品可使用性方面的问题，因为他们对产品已经“习惯”了产品的（潜在不合适的）可使用性。

因此，可以从测试团队之外的客户代表中选择一部分人进行可使用性测试。意图了解用

户想要什么和客户预期的面对客户的团队（例如客户支持、产品市场开发人员），可提高可使用性测试的有效性。

可使用性有时可以是执行其他类型测试的副产品并被注意到，很难针对可使用性编写测试用例。一般来说，要针对可使用性准备检查单和指南，并在测试期间遵守。在发现可使用性缺陷时关注细节、直觉和构建技能，与编写可使用性测试用例相比，会使可使用性测试更有效。在可使用性中有很多东西（将在后续各节中介绍）需要检查，公司应该培训和普及对可使用性方面专门知识的了解，以实现和测试可使用性。

> 可使用性的正确途径，是测试对用户有影响的所有工件产品，例如二进制代码、文档、媒介、包括图形和命令用户界面的使用模式是否适用。

可使用性的另一个问题是系统对用户给出的消息。消息分为三类——信息消息、警告消息和错误消息。可使用性测试内容取决于错误消息的类型。当出现信息消息时，要对消息进行检查，看最终用户是否能够理解这条消息，并将所完成的操作和背景关联起来。当出现警告消息时，要检查为什么会出现这条消息，如何避免出现这条消息。当出现错误消息时，要检查三个内容：错误是什么、为什么会出现这个错误以及如何避免或绕开这个错误。

可使用性并不只针对正确使用。当用户没有犯错误，产品能正确运行，而当用户犯了错误，产品运行不正常（或说用户界面不友好），就说产品是可用的都是不对的。系统应该明智地检测和避免错误的用法，如果错误用法不能避免，系统应该提供一个恰当和有意义的消息，把用户引导到正确的道路上。因此，可使用性应该通过正面和负面测试，即产品的正确和错误使用。

可使用性不要和图形用户界面（GUI）混淆。可使用性还适用于非GUI界面，例如命令行界面（CLI）。有大量Unix/Linux用户认为CLI比GUI更可使用。SQL命令是CLI的另一个例子，对于数据库用户来说，SQL更可使用。因此，可使用性还应该考虑用户使用的CLI和其他界面。

12.3 可使用性测试的时机

确保可使用性的最佳时机是实施两个阶段的可使用性测试。第一阶段进行设计确认，第二阶段作为测试周期中的组件和集成测试的一部分进行可使用性测试。

早在开发周期策划测试时，就应该并行地策划可使用性需求，这与任何其他类型的测试相似。但是一般来说，可使用性往往被忽视（至少分配的优先级很低），并没有从项目的开始就进行策划和执行。如果有两个缺陷，一个是功能缺陷，一个是可使用性缺陷，通常功能缺陷会有处理规程，这种方法是不对的，因为可使用性缺陷可能会使用户丧失使用软件的兴趣（即使执行的是所需的功能），如果用户拒绝产品，就意味着给产品公司造成巨大经济损失。此外，实践证明在测试周期中推后可使用性测试会有很高的代价。因为大量可使用性缺陷最终需要修改设计，需要修改多个屏幕，会影响不同的代码路径。如果尽早策划可使用性测试就可以避免这些情况出现。图12.2给出了可使用性测试的阶段和活动。

产品的可使用性要通过设计实现。只考虑功能的产品设计不能得到用户的认可。针对功能的产品设计还需要大量的培训，如果设计时甚考虑功能又考虑可使用性就会最大限度地降低培训要求。在可使用性测试的第一阶段，要确认可使用性设计的这个方面。

可使用性设计可以通过多种手段验证，包括：

- 风格单 风格单列出用户界面设计要素，使用风格单可保证多个屏幕设计要素的一致性。

测试风格单可保证测试基本的可使用性设计。风格单还包括帧，每个帧都被用户认为是一个独立的屏幕。要对风格单进行评审，检查风格单是否规定了字体大小、颜色模式等，这些可能会影响可使用性。

- **屏幕原型** 屏幕原型是可使用性设计的另一种测试方法。屏幕设计成与提交给客户的一样，但是没有与产品的其他模块集成，因此，这种用户界面不与其他功能模块集成，独立地测试。这种原型有模拟的其他用户界面功能，例如屏幕导航、消息显示等。原型使用户对产品发布后屏幕的确切外观和功能有直观印象。测试团队和一些真实用户测试这种原型，并把改进意见吸收到用户界面中。当这种原型测试以后，再与产品的其他模块集成。
- **书面设计** 书面设计利用确认可使用性设计的最早机会，早在产品的实际设计和编码还没有完成前就可以实施。屏幕、布局和菜单的设计被画在纸上，提交给用户等待反馈意见。用户可视化将书面设计和操作及操作顺序关联起来，得到使用产品的感觉，并提出反馈意见。使用风格单需要进一步的编码。原型需要二进制代码和资源来验证，但是书面设计却不需要任何其他资源。书面设计可以通过电子邮件发送或打印，并采集反馈意见。
- **布局设计** 风格单保证一套用户界面要素被组织起来，并一起反复使用。布局有助于动态地在屏幕上安排不同的要素，保证要素、空白、字体尺寸、图片、排列风格等在屏幕上的安排。这是可使用性设计的一部分，也需要进行测试。

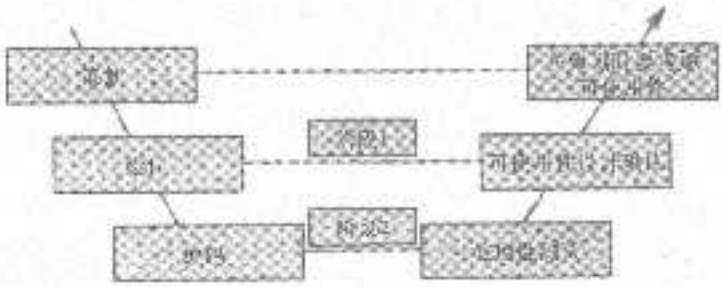


图12-2　可使用性测试的阶段和活动

如果对现有产品进行再设计或增强，通过使用现有的布局可以避免可使用性问题，因为用户已经熟悉了产品，会认为这样更可使用。对现有产品进行重大可使用性变更（例如重新组织屏幕上的按键顺序），会使用户产生困惑，导致出现用户错误。

在第二阶段，要运行产品，测试可使用性。在执行测试前，要挑选一部分实际用户（第一次接触产品和特性）使用产品，收集他们的反馈意见，并解决所提出的问题。有时很难请到产品的实际用户进行可使用性测试。在这种情况下，可从产品开发团队外部挑选一些用户代表，例如支持、市场开发和销售人员。

什么时候实施可使用性测试还取决于所开发的产品类型。例如，针对局域网环境设计的客户应用程序（Windows client32）通常有很丰富的特性，每个屏幕都试图完成各种任务，并提供大量信息。Web服务则不同，每个屏幕的信息量和任务很有限，以快速装载Web页面。对这些类型的应用程序要在不同阶段完成可使用性测试活动。表12-1对比了客户和Web服务的一般开发和测试方法。

表12-1 客户和Web应用程序的开发和测试

客户应用程序	Web应用程序
第1步：设计功能	第1步：设计用户界面
第2步：编写功能代码	第2步：编写用户界面代码（原型）
第3步：设计用户界面	第3步：测试用户界面（第1阶段）
第4步：编写用户界面代码	第4步：设计功能
第5步：将用户界面与功能集成	第5步：编写功能代码
第6步：结合易能测试用户界面（第1、2阶段）	第6步：将用户界面与功能集成
	第7步：结合功能测试用户界面（第2阶段）

如表12-1所示，Web界面要在设计功能之前设计。这为实施两阶段（第1和第2阶段）的可使用性测试留下了充足的时间。而在客户应用程序中，用户界面只能在确定了功能以后确定，因为用户界面编码是最后一项活动。设计确认没有多少工作要作为一个单独阶段完成。因此，将第1、2阶段合并。同时，没有硬性和严格规定客户应用程序不能在功能设计前先设计用户界面。随着越来越多地关注可使用性，这是唯一可以改变的公共实践。

只是测试本身并不能使产品完全可使用。产品必须针对可使用性进行设计、编码、测试，因此真实用户应该参加可使用性方面的评审才能取得最大效益。

前几章介绍了多个测试阶段。可使用性测试的责任散布在所有测试阶段和测试团队中，可使用性不能"外加"到产品上，应该预先策划，并必须测试测试。因此，可使用性作为一种需求可以间接地测试（直接测试由测试团队完成），从产品开发的第一天开始，一直持续到产品发布。在产品开发中的每个人都需要贡献可使用性的各种视点和需求。

"可使用产品"永远是所有相关利益方在整个项目开发期间共同协作的结果。

12.4 实现可使用性的方法

如果有人期望用户会明确地报告可使用性问题，那他的期望是会落空的。对用户界面失望的用户可能会逐渐总结出一些绕过方法（甚至不再使用产品的一部分！）。但是，主动吸收用户参加用户界面设计可以得到很好的结果。用户界面需求不能采用文字描述。用户使用产品时，必须有这种产品是可使用的感觉。这就是为什么要预先或在设计阶段（甚至是需求获取阶段）吸收客户、采集客户对所有用户界面需求的反馈意见的原因。当然这也并不总能行得通，因为：

1. 用户可能没有时间参加这种活动，

2. 有些用户会提出矛盾的需求，不可能满足每个人的愿望。

3. 对于突破性创新产品，用户甚至不能想象怎么使用产品。因此反馈意见可能不直接相关。

用户有不同的类别。有些用户可能是专家，有些可能刚入门，还有些可能是初学者。专家用户通常不会提出可使用性问题，他们会想出绕过办法，并自己适应产品，尽管他们也希望得到更好的产品。刚入门的用户通常会受到可使用性的影响，但仍然不会提出可使用性问题，因为他们渴望了解更多的产品知识。他们通常不知道是可使用性方面的问题还是自己不太会使用产品的原因。每一类用户对产品的期望也是不同的。初学者会提出大量可使用性问题，但是有些不会考虑，因为他们还没有足够技能使用产品。不考虑用户的分类，产品就要对所有用户都是可使用的。

除了以上类别的用户外，还有一些用户对产品的可使用性很具挑战性，即听力、视力或活动有障碍的用户。应对听力、视力和活动有障碍用户的可使用性测试，叫做易获得性测试，将在12.7节讨论。

为了使产品对于所有类型的用户都是可使用的，每类用户至少要吸收一人参加可使用性测试。前面介绍过，吸收每类用户参加可使用性测试并不总是行得通的。因此，可使用性测试团队的一部分人要从不同的团队挑选，代替不同类型的真实最终用户测试。

前面介绍过，很多类型的用户和客户并不会提出可使用性缺陷。问题并不是产品是否有可使用性缺陷，而是用户如何对产品使用作出反应。对这些用户反应（例如，愤怒的用户砸键盘表示系统的响应时间还不够快）会比他们实际提出的缺陷更有说服力。当用户在可使用性测试中使用产品时，他们的活动受到严密的监视，观察到的所有现象都要记录下来，并由测试团队提出缺陷，而不是期待所有问题都靠用户提出。记录操作序列、屏幕、用户反应和观察都需要建立可使用性实验室。12.9节将详细介绍这个问题。在进行观察前要提出一定的指南/检查单，在可使用性测试中进行检验。检查单中的一些检查项包括：

> 可使用性是一种习惯和行为，就算人一样，对于不同的用户及其预期，产品要有不同的和正确的行为。

1. 用户是否成功地完成了所指定的任务/操作？
2. 如果是，他们完成该任务/操作用了多少时间？
3. 产品的响应时间是否足够快，能够感到满意？
4. 用户在什么地方受阻？遇到了什么麻烦？
5. 他们在那里感到困惑？靠自己能继续吗？什么帮助他们继续下去？

除了检查单，产品还要针对其他可使用性成分进行检验，例如可理解性、一致性、响应和美观程度。12.5节和12.6节将讨论这些问题。

12.5　可使用性的质量因素

前面几节介绍了可使用性测试以及实施可使用性测试的途径和方法论。在实施可使用性测试时有些质量因素是非常重要的。前面介绍过，可使用性是主观性的，并不是可使用性的所有需求都可以清楚地写入文档，但是，关注以下给出的质量因素可以提高可使用性测试的客观性。

1. **可理解性** 产品应该有简单和有逻辑性的特性和文档结构，应该以用户场景和使用为基础。在场景早期执行的最常使用的操作应该通过用户界面首先给出。当把特性和组件集成进产品时，应该按用户的术语，而不是按技术或实现术语。

2. **一致性** 产品需要与适用的标准、平台外观感觉、基本设施和同一产品的早期版本一致。此外，如果同一公司有多个产品，那么这些产品在外观和感觉上最好有一定的一致性。在不同底层操作系统上的不同用户界面会使用户不满，因为用户在使用这些操作系统上的产品版本时，需要对不同的模板和规程感到舒服。例如，如果一个操作系统使用一种带有一套图标、错误编号、错误消息和到相关文档的链接的错误消息模板，遵循同样的格式会使用户对该操作系统上的产品感到更舒服。除非产品的用户界面有严重问题，否则当前的界面和用法要与相同产品的较早版本一致。遵循一些可使用性标准有助于满足可使用性的一致性要素要求。

3. **导航** 这有助于确定选择产品的不同操作有多容易。隐藏很深的选项需要用户经过多个屏幕或菜单选项才能执行该操作。执行一个操作所需的鼠标点击或菜单选择次数应该尽可能

减少，以提高可使用性。在用户受阻或感到困惑时，应该有容易的选项放弃或退回到前一个屏幕或主菜单，以便用户尝试选择一条不同的操作路径。

4. 响应 产品对用户请求的响应有多快是可使用性的另一个重要方面。不要把可使用性的响应与性能测试混淆起来。屏幕导航和直观显示应该几乎在用户选择了某个选项后立即给出，否则会给用户留下没有进展的印象，使他们不断重复尝试。只要产品在进行信息处理，就应该提供进展以及指示还剩多少时间的直观显示，使用户耐心等待，直到操作完成。用于指导用户的适当的对话框和弹出窗口也可以提高可使用性。不能用太多的弹出窗口和对话框处理响应。在很短的时间内出现太多的提示只会降低可使用性，事实上还会降低响应时间。

在可使用性检查单中增加一些以上讨论的质量因素，保证在设计和测试时考虑这些因素，有助于得到好的可使用的产品。

12.6 美感测试

可使用性中另一个重要方面是使产品"漂亮"。实施美感测试有助于进一步提高可使用性。美感测试很重要，因为很多公司的产品中与美感有关的很多问题都因为不属于功能缺陷而被忽视。产品中的所有美感问题一般都会映射到所谓"装饰性"缺陷类别中，其优先级是最低的。用一个单独的测试周期关注美感有助于建立预期，并关注改进用户界面的外观和感觉。

是否"漂亮"是一种主观判断（"情人眼里出西施"）。对一个人是可接受的东西，另一个人可能认为很丑。关注产品的美感问题可以保证产品是漂亮的，至少产品最后绝不能说是很丑。美观决定产品的第一印象，产品是否漂亮不重要的认识可能会影响用户对产品的接受。

但是，在很多产品公司中美学都要往后靠。美观不仅是外观，而存在于所有方面，例如消息、屏幕、色彩和图像。令人舒服的菜单外观、令人愉悦的色彩、小巧的图标等可以提高美观程度。一般认为这是镀金，但这种观点是不对的。镀金通常是在装饰完成后进行，而美观测试则不应该是最后一项测试活动，有些美学问题必须在设计阶段解决，不应该被看作是发布之前最后的低优先级的活动。

> 要让所有产品都像Taj Mahal那么漂亮是不可能的。但是针对美观的测试至少能保证产品看上去很好看。

美观测试可以由任何具有审美观的人承担，这意味着所有人。邀请美学家、艺术家和建筑师这些在日常生活中从不同方面创造美（不只是产品）的人在美观测试中担任专家。在设计和测试阶段邀请他们，听取他们的意见，使产品更美观。例如，如果由艺术家设计产品中使用的图标会更吸引人，因为它们不仅传递信息，而且使产品更漂亮。

12.7 易获得性测试

很多人的视力、听力和移动存在部分或完全的障碍。不考虑他们的需求，产品可使用性会不被接受。对于这些用户，要提供使用产品的替代方法。有一些工具可以帮助提供替代方法。这些工具通常叫做易获得性工具或辅助技术。易获得性测试要测试这些使用产品的替代方法，结合易获得性工具测试产品。易获得性是可使用性的一个子集，应该包含到可使用性测试策划中。

> 检验产品对于行动不便的用户也具有可使用性的测试叫做易获得性测试。

产品的易获得性可以通过两种手段提供。

1. 利用基础架构（例如操作系统）提供的易获得性特性，叫做基本易获得性。

2. 通过标准和指南在产品中提供的易获得性，叫做产品易获得性。

12.7.1 基本易获得性

基本易获得性是硬件和操作系统提供的。计算机的所有输入和输出设备和其易获得性选项都属于基本易获得性。

键盘易获得性

键盘对于视力和移动有障碍的用户来说是最复杂的设备。因此，在易获得性上要特别注意键盘。有些易获得性改进是在硬件上完成的，有些是在操作系统上完成的。硬件易获得性改进的一个例子是在F和J键上加了一个小凸起，这种小凸起有助于有视力障碍的用户通过触觉调整敲键的手指。键盘上的键有不同大小，以及提供快捷键，都是通过硬件改进易获得性的例子。

类似地，操作系统提供商也在键盘上做了一些改进，包括使用粘贴键、开关键和箭头键操作鼠标。

粘贴键 为了说明粘贴键概念，以<CTRL><ALT><DEL>为例。对视力和移动有障碍的用户来说，最困难的序列之一就是<CTRL><ALT><DEL>。这种键盘序列有很多用途，例如登录、退出、加锁和开锁计算机、关机、调出任务管理器。有时对于没有障碍的用户来说，键序列也是很复杂的，因为它要求先按住两个键再按下第三个键。这种特定序列要求两只手上的三个指头必须协调，操作系统中的粘贴键设置消除了对按下键的要求。如果启用了粘贴键特性，用户可按下<CTRL>和<ALT>键一次，再释放，然后再按<DEL>键。这就可以用一个手指完成该序列操作。

过滤键 当键被按下并持续一定时间后，就认为是重复键入。有时，这对残疾用户是个挑战。有些残疾人不能像正常人一样快速按下和释放按键，过滤键可以完全停止重复或降低重复速度。

开关键声响 启用了开关键后，键入的信息可能与用户想要的不同。例如，在典型的字处理软件包中，<INS>键是一个开关。如果<INS>键的正常设置是插入字符，那么当键被按下一次然后释放，就会进入“替换”模式，这时所键入的字符替代已有的字符。有视力障碍的用户很难看清这些开关键的状态。为了解决这个问题，可以启用声响，启用和关闭开关键时会发出不同的音调。

声响键 为了帮助视力有障碍的用户，还有一种机制在用户敲击键盘时会读出所键入的字符。在有些操作系统中，这种特性作为朗读器实用程序（后面还要讨论）的一部分，但是很多易获得性工具都有这种特性。

用于控制鼠标的箭头键 移动有障碍的用户很难移动鼠标。通过启用这种特性，这类用户就能够使用键盘上的箭头键移动鼠标。鼠标上的两个按钮及其操作也可以通过键盘完成。

朗读器 朗读器是一种实用程序，提供声音反馈。例如，当用户执行事件时朗读器会朗读出来，读出所键入的字符，用不同的声音提示系统事件等。由于所提供的声音对应键盘和系统事件，因此这个特性被认为是对易获得性很重要的实用程序，可以提高产品的易获得性。

屏幕易获得性

很多键盘易获得性特性帮助了有视力障碍和移动障碍的用户。这些特性对有听力障碍的用户没有帮助，他们需要屏幕上的额外视觉反馈。下面这些易获得性的特性使用屏幕提高了可使用性。

可视声响 可视声响是声音的“声波形式”或“图形形式”。这些可视效果通过屏幕告诉用户系统发生的事件。

针对多媒体的字幕特性 所有多媒体语音和声音都可以用文字描述，在播放语音和声音的同时，在屏幕上显示文字。

软键盘 有些有移动障碍和视力障碍的用户认为使用点击设备比键盘更容易一些。软键盘通过在屏幕上显示键盘来帮助这样的用户。可以通过点击设备，例如用鼠标点击屏幕上的键盘画面键入字符。

采用高对比度的易读特性 有视力障碍的用户在识别菜单项中的一些色彩和字号上存在困难。操作系统一般会提供一个开关选项，切换到高对比度模式。这种模式在屏幕上的所有菜单使用舒适的色彩和字号。

其他易获得性特性

操作系统层还提供了很多其他易获得性特性，有视力或移动障碍的用户可能认为键盘和鼠标都很难使用。在这种情况下，应该提供使用其他设备的选项。例如，操纵杆可以用作点击设备的替换设备。这种点击设备可以结合软键盘使用，以满足用户的要求。

以上介绍的一些特性试图满足有多种障碍的用户需求。例如，粘贴键在消除按住键要求的同时，还在屏幕上的计算机工具条上显示所有粘贴键的状态。

12.7.2 产品易获得性

在为产品提供易获得性时，需要很好理解基本的易获得性特性，产品应该尽可能实现这些基本易获得性特性。例如，提供多媒体文件对应的详细文本可保证产品具有字幕特性。

> 需求举例1：必须为音频、视频和图片影像提供对应的文字字幕。

对基本易获得性特性和具有特殊需要的不同类型用户需求的很好理解，有助于编写一些关于产品用户界面设计方面的指南。在为产品提供易获得性的整个过程中，应牢记这些不同类别用户的不同需求，以及他们的能力和面临的挑战。

在各种网站上有很多关于易获得性标准和需求的信息，可以用于收集易获得性需求。可以参考易获得性标准，例如508和W3C，获得较完整的需求。本节通过几个需求为读者建立背景环境，并解释其概念。

这种需求说明提供图像影像对应文字、为音频部分提供字幕是很重要的。在播放音频文件时，提供字幕可为有听力障碍的用户提高易获得性。增加音频片段可为不能理解视频和图片的有视力障碍的用户提高易获得性。在尽力满足这类需求时，要了解已有的易获得性工具。例如，不了解易获得性工具的人可能，认为提供视频对应的字幕没有多大意义，因为有视力障碍的用户就看不到视频也看不到字幕。但是，如果用户使用像朗读器这样的工具，相关的文字就可以读出来，产生语音，从而使有视力障碍的用户受益。图12-3是从网站上获取的带有对应文字的图片样本。

> 需求举例2：支持和手段的组织要使得不需要特定的屏幕分辨率和模板（叫做网格单）也能阅读。

在上面的例子中，即使有视力障碍的用户不能看到图片，左侧给出的相关图片标签也会对使用朗读器的用户有帮助。

因此音频对应的文字（字幕）、图片和点观表示的音频描述，对于提高易获得性是很重要的。

图12-3　与白对比文字的图片

视力低的用户在使用产品时希望看到超大号字体。因此，期望用特定的分辨率和字号在屏幕上显示所有字段的产品会面临易获得性问题。字段和文本之间应该有足够的空白，使得增大字体时出现在屏幕上的消息不会聚在一起。

通常Web页面上的信息采用不同风格单的模板显示在屏幕上。有两种类型的风格单：内部风格单和外部风格单。内部风格单是硬编码的形式，指示字段、大小及其在屏幕上的位置。当用户希望调整窗口和字号大小时，这会带来问题。最安全的办法是使用外部风格单，产品的程序不应该影响用户定义的外部风格单。

不仅是视力低的用户，视力很好但是有色盲症的用户可能在识别各种色彩及其组合上也会出现问题。因此，使用彩色文本、彩色图片和彩色屏幕的产品不应该把色彩作为区分用户界面要素的唯一方法。例如，在如图12-4所示的图片中，色彩（绿和红）用作区分不同操作按钮的方法。但是，色盲用户可能难以选择正确的操作按钮。正确的方法应该是保留色彩并恰当地为按钮命名。在下面的例子中，把绿按钮命名为“继续”，把红按钮命名为“停止”就会保证易获得性。

需求举例3：所设计的用户界面应该使实用色彩传递的全部信息不利用色彩也能获得。

图12-4　把色彩作为区分方法

不同的人阅读速度是不同的。阅读速度慢的人会对一闪而过的文字感到不满，因为这会进一步影响其阅读速度。即使视力很好的用户也会觉得移动太快的文字不太舒服。一些易获

得性标准规定，文字闪烁的频率应该在2Hz和55Hz之间。

有些有移动障碍的用户和有神经方面问题的用户移动眼球的速度没有其他人快，还有其他一些健康问题使人不能像其他人一样快速阅读。移动和滚动文本时，文本移动的速度应该与最低的产品用户的阅读能力相适应，否则产品应该有调整文本移动速度的特性。

在设计用户界面时，应特别注意保证降低对使用产品所要求的身体移动，以帮助有移动障碍的用户。应该避免将界面元素散布到屏幕的角落，因为这要求用户把点击设备移动到屏幕角落，比较费力。图12-5是一个把四个字段散布到屏幕角落的例子。如果屏幕中有充足的空白，那么同样的用户界面如果设计为纵向排列会更好。

需求举例4：降低文字移动速度，避免使用闪烁文字。

图12-5 把四个字段散布到角落的屏幕

不仅单个屏幕，整个屏幕集在设计上也要考虑整体考虑最少移动要求。当使用多个屏幕完成一组操作（比方说确认多个屏幕的用户信息）时，需要提供相应的位置序列响应，例如“继续”、“确认返回”、“退出”按钮在不同屏幕上应该在相同的位置上，当屏幕改变时，点击设备不必再移动。

用户界面的设计要尽可能预期用户使用较少的设备完成常规操作。例如在图12.5中，使用键盘是最基本的要求，如果可以使用<TAB>键在字段之间跳转，则可以避免使用点击设备来获得输入信息。混合使用多种设备，例如键盘和鼠标，会增加用户界面的复杂性。

如果要求快速响应，则应该通知用户并用足够的时间告知需要更多的时间。有些用户需要更长的考虑和响应时间，因为可能有某些障碍。在用户界面上显示“如果信息正确，在5秒内点击OK键”会给用户带来压力，应该避免。

需求举例5：在设计用户界面时，应降低对用户的身体移动要求，并为用户响应留出足够时间。

以上给出的样本需求都是提高易获得性的例子，类似这样的需求还有很多。在定义易获得性的流行标准（例如508指南）中，这些需求一般按提供用户界面所采用的技术（例如基于Web的界面、基于客户机的界面等）分类。针对界面设计所使用的技术选择合适的标准会长远地提高产品的易获得性。

12.8 可使用性工具

在可使用性方面没有多少工具，因为评价可使用性具有很高的主观性。但是，有不少易

获得性测试工具。表12-2给出了一些可使用性和易获得性工具，前几节已经解释了其中的一些工具。

表12-2　可使用性和易获得性样本工具

工具名称	用　途
JAWS	用于测试产品的易获得性，涉及一些辅助技术
HTML确认器	根据可使用性和易获得性标准确认HTML源文件
风格单确认器	根据W3C建立的可使用性标准确认风格单（校验）
放大器	针对有视力障碍用户的易获得性工具（使他们能够放大显示在屏幕上的内容）
朗读器	朗读显示在屏幕上的信息，针对有视力障碍的用户到达有解释性的工具
按键器	通过在屏幕上显示键盘模板，使用户能够实现点击设备使用健全

采用黑白监视器而不是彩色监视器进行测试，可保证使用合适的色彩组合，也可保证通过色彩表示的信息没有色彩同样也能表达。有些彩色监视器有以灰度显示信息的特性，有些操作系统有黑白显示的设置。利用这种显示设置进行测试，可保证通过色彩表示的信息没有色彩同样也能表达。

邀请残疾用户参加产品评审也是一种确认产品易获得性的方法。

12.9　可使用性实验室的建立

前面已经讨论过，期望用户对可使用性提出意见是不大可能的。产品开发人员常常忽视可使用性缺陷。当用户面对类似的缺陷并逐渐习惯后，他们也会忽视可使用性缺陷。因此，单凭用户提出意见不会使产品具有可使用性。观察用户的"肢体语言"可以指出产品的可使用性缺陷，否则这些缺陷可能永远也不会被指出。

不仅是肢体语言，背景，即用户受阻的屏幕和使用户感到困惑的事件等，也需要与用户的反馈关联起来，以提取缺陷。

以上这些足以成为建立可使用性实验室的充分理由，以便记录、观察有关可使用性的所有信息，并修改缺陷。图12-6给出了可使用性实验室的一些关键元素。

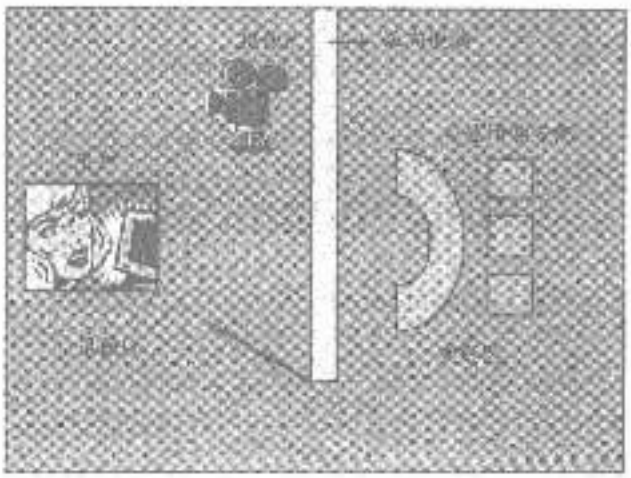


图12-6　可使用性实验室的关键元素

如图12-6所示，可使用性实验室有两个部分，即"记录区"和"观察区"。用户被请到实

验室来，在记录区执行一组预先确定的操作。预先向用户解释产品的使用方法，并提供文档，用户开始准备执行任务。在记录整个操作序列过程中，可使用性专家观察整个过程。

在观察区，一些可使用性专家坐着观察用户的肢体语言，并将缺陷与引起问题的屏幕和事件关联。观察通过“单向玻璃”进行，专家可以看到用户但是用户看不到专家。这有助于避免专家的身体移动对用户产生干扰，使用户处于自然状态。摄像机可以从不同角度捕获用户和计算机监视器的实时图像。观察了使用产品的不同用户后，可使用性专家提出对产品的改进建议。

被记录的可使用性测试可以在事后进行进一步观察，还可以针对可使用性缺陷修改得到更多的信息（如果有必要）。

因此，建立独立的可使用性实验室可以作为确定更多可使用性缺陷，并修改这些缺陷的一种工具。

12.10　可使用性的测试角色

可使用性测试在一些公司中不像其他类型的测试那样正式，不采用预先写好的测试用例或检查单。这些公司在进行可使用性测试时使用了以下不同方法：

1. 把可使用性测试作为一个独立的测试周期实施。

2. 聘请外面的顾问（了解人类工程学和特殊易获得性需求的专家）进行可使用性确认。

3. 建立独立的可使用性小组，使所有产品开发团队的可使用性实践制度化，建立公司级可使用性标准。

不管什么时候和怎样进行可使用性测试，都有一些技能对于执行完善的可使用性测试是非常重要的。表12-3归纳了可使用性测试的不同角色的技能和预期。

表12-3　可使用性测试的角色和责任

角　色	责　任
可使用性实践倡导者	• 在公司范围内制度化和改进可使用性 • 培训、推动和获得可使用性所需的资源和支持，作为公司的产品资产 • 提供解决整个公司在可使用性实践测试过程中出现的问题 • 在客户和公司之间建立沟通渠道，以获得可使用性工具的反馈意见
可使用性专家	• 提供执行可使用性测试所需的技术指导 • 确定产品可使用性测试的策略 • 对执行可使用性测试的所有人员提供技术指导 • 建立可使用性标准，并认证产品团队
人机因素专家	• 针对可使用性手册、屏幕和其他产品 • 保证多个产品之间的一致性 • 对改进用户产品的使用体验提出建议 • 从产品的一致性、易用性和使用效率方面提出意见
图形设计者	• 创建用户界面所需的图标、图形图象等 • 交叉检查图标和图形信息与其所处理的背景，保证这些图像是否传达了正确含义
可使用性管理者或顾问	• 评估、实施和监控所有可使用性测试活动 • 与可使用性专家密切协作，确定可使用性实验，并在项目中落实 • 与客户协作，在发布之前获得产品的当前版本的反馈意见
可使用性测试工程师	• 根据脚本、场景和测试计划执行可使用性测试 • 如果可能，不仅要从执行可使用性测试角度，还要从用户角度提出反馈意见

12.11 小结

软件界已经逐渐认识到可使用性测试的重要性。不久，可使用性测试就会成为一种工程学科、一种生存周期活动和一种职业。

但是，可使用性测试不是没有挑战。往往只是在可使用性测试结束前，为了赶过程才将可使用性测试工程师和顾问请来，这很难真正提高产品的可使用性。即使完成了可使用性测试，也是由在可使用性方面没有足够技能和接受过培训的工程师完成的。可使用性是正在发展的领域，培训对于帮助员工掌握最新技术是非常重要的。

有些公司从项目生存周期的开始就策划可使用性测试，并一直跟踪到结束。但是，在对工程师进行标准、工具、实验室建立等方面的培训上投入不足。只有当足够地关注这些方面，当在可使用性专题人员培训加强投入并落实到项目上，才能真正提高产品的可使用性。

可使用性不能只靠测试获得，可使用性更多地体现在设计上，体现在对产品作出贡献的人们的头脑中。可使用性是关于用户的产品体验。在项目开发时始终从用户的角度进行思考，将长期保证可使用性。

问题与练习

1. 确认可使用性设计可以有哪些不同方法？
2. 考虑不同障碍的用户进行分类，并总结易获得性测试和工具怎样帮助他们使用产品。
3. 什么是美观测试？为什么把美观测试纳入可使用性测试检查单很重要？
4. 请解释可使用性和易获得性可用的不同工具，这些工具对测试可使用性有什么作用？
5. 建立可使用性测试实验室的关键目标是什么？可使用性测试实验室与其他传统测试实验室有什么差别？
6. 请解释测试专业人员在可使用性测试中可以充当的不同角色。
7. 请解释V字模型，并解释可使用性测试的不同活动可以怎样映射到V字模型中。

第四部分　测试中的人员和组织问题

这部分将讨论在测试中经常被忽略的两个问题，实践证明这两个问题是开展测试项目的最大挑战。第13章首先讨论测试团队所面临的常见的人员问题。先列出对测试的一些道听途说的错误认识，然后对其进行分析，然后讨论人们在测试职业上的发展途径，讨论需要怎样做才能在这个生态系统中达到自己的目标。第14章将介绍如何组织测试团队并对团队成员分配工作，以提供更好的可审计性和有效性。这一章还将专门讨论全球化对团队结构和工作场所的影响。

第13章 常见人员问题

13.1 关于测试的感觉和错误概念

本节将讨论一些关于测试专业的常见的“道听途说”式说法。这些说法有的来自测试人员，有的来自管理层，有的来自学术界。本节将研究这些说法的出处，说法的错误所在和反驳这些说法的论据。重要的是在建立成功的有激情的测试团队之前，首先要澄清这些错误概念和认识。

13.1.1 “测试没有什么技术挑战”

如果进行招聘测试人员的面谈，通常会观察到应聘人员非常两极化的反应。第一类人通常占少数，他们非常自豪、投入和愉快地成为测试人员，很不幸，第二类人占多数，他们认为干测试是“因为别无选择”。他们常常认为测试没有什么技术挑战。出现这种认识有多种论据，其中之一是，“如果我做开发，可以得到给定程序设计语言的专业知识，这是很有价值的。而另一方面，测试只不过是例行公事和重复性的工作，并不需要什么特殊技能。”

这种论点在二十年前也许是对的，那时大部分测试工作是靠手工完成的，产品也比较简单。而今天的情况就不一样了，测试流与开发工作有很大的并行性，如表13-1所示。（下一节将介绍测试和开发工作之间的差别，这些差别是产生这些认识和错误概念的根源。）

表13-1 测试与开发工作的相似性

开发项目的功能	测试项目的相应功能	相似性
需求描述	测试描述	需要求对领域有透彻理解，有时需从需求要求从用户视点对整个系统有甚至更完整的理解
设计	测试设计	测试设计在构造测试系统时具有产品设计的所有属性，要考虑重用、形成标准、等等
开发/编码	测试脚本开发	包括使用测试用例开发和使用测自动化工具
测试	使测试可接受	需高要开发和测试团队之间的密切合作，以保证获得正确的测试结果
维护	测试回归维护	随着维护变更，保持测试用例的正确性（回归测试、功能测试等）

需要对整个产品而不是单个模块有完整的理解 开发工程师一般倾向于关注其他的模块，他们有可能在一定程度上忘记其他模块的功能。而测试工程师一般需要对产品有更完整的理解，而不是局限于对单个模块或组件的理解。这要求测试工程师成为领域专家。

需要全面理解多个领域 随着产品变得更复杂、更开放和无缝地相互集成，测试需要深入理解多个产品的领域。就像开发一个产品一样，测试也需要对应用领域的细微差别有深入的理解。事实上，扩展上一段给出的论据，大多数开发都是在封闭的环境下完成的，而测试则要求更深入地理解多个领域之间的交互和相互依赖关系，以便模拟真实场景。因此，测试看

来甚至更适合具有产品领域专门知识的人，更具有挑战性。

语言专门知识 大多数开发人员都坚持认为要在某些具体的程序设计语言或平台获得职业专门知识。在20世纪90年代初，大多数人都想成为C语言专家，后来是C++，再后来是Java。类似地，对于平台，流行平台从大型主机转移到客户—服务器计算环境，再转移到以网络为核心的计算环境。直到大约十年前，从事测试的人们对于这类"可丰富简历"的语言技能还只有很少的并行性，大多数测试要么是手工完成的，要么是用自己开发的工具完成的。在过去几年中，随着专门的测试语言和工具的出现，已经缩小了测试方面在语言专门知识上的差距。今天，"测试不需要任何程序设计"的论据再也站不住脚了。事实上，大多数测试工具的使用都要求程序设计语言知识，与开发人员使用的语言非常相似。

使用工具 从事软件生存期中其他活动的人能够使用各种工具，例如CASE工具、调试器、集成开发环境等。工具增加了他们工作的"魔力"，使用起来很"酷"。测试人员使用工具的机会很少。在过去的几年中这方面同样发生了很大变化，市场上已经有了标准的测试工具，不仅支持类似程序设计语言的测试语言，还提供机会更好地设计和与被测产品程序代码的集成。由于工具很复杂，实践证明，有时候测试用例自动化甚至比开发产品的代码更具挑战性！

概念化和开放思维的机会 由于测试被看作是"破坏性"工作，有人认为开发工作有比测试工作更多的概念化机会。前面介绍过，测试过程与开发过程有很强的并行性。即使测试被认为是破坏性的（从发现产品中的缺陷这一点上说），这些相似性足以说明测试工作也有大量的概念化和开放思维的机会。遗憾的是，人们在寻求测试工作时常常完全忽视了这种相似性。

今天在测试中有大量的投入——有时甚至超过开发 例如，测试工具是非常昂贵的，显然公司已经肯定对这些昂贵工具的投入会有足够的回报。因此，公司完全有可能把最好的员工投入到测试部门去。

今天的内部测试工具将可能是明天的一种非常昂贵的测试自动化工具！ 在很多公司中，最初是在小组里使用的内部工具最终发展为可调整规模的产品，派生产品和应用在多个小组中使用，甚至发展成兼容的测试自动化产品。从技术挑战方面看，这也许增加了测试工作更宽视野的需要。

从以上观点看，测试提供了足够的技术挑战，对测试感兴趣的专业人员能够持续学习和提高自己的知识水平，在提高产品质量的同时，也达到更高的个人满意度。

用一句话作为对比开发与测试的总结："所有的开发中都有测试，所有的测试中都有开发。"

13.1.2 "测试没有为我提供职业成长道路"

有些公司为进行开发和其他软件工程工作的员工定义了职业发展道路，但是测试工作的职业发展道路却不清晰。事实上，连他们的工作职务给的也是"开发工程师"、"高级开发工程师"等！从项目生存周期观点看，开发人员可能很自然地提升为设计人员、业务分析员和领域专家，甚可以成为受人尊重的"架构设计师"。测试公司并不总能提供这种明确的职业发展机会。这并不意味着对于测试专业就没有职业发展道路。由于我们深信对于测试专业也有同样有利的职业发展道路，下一节还要讨论测试专业职业发展道路的选择问题。

> 测试不是恶魔，开发也不是天使，但是测试和开发的机会是均等的。

13.1.3　"我被派来测试——我到底怎么了?!"

> 如果一个人不适合做开发，那么由于同样或类似的理由，他也同样不适合做测试。

由于测试是最靠近向客户发布产品的一项活动，有理由认为公司会把一些最好的人员部署在测试部门中，测试部门应该与其他部门一样看待。但是，与其他部门有关的（错误）"迹象"有时会向测试团队传递出并不清晰的信息。有时人们会感到他们做测试是因为他们不适合干别的！这种信息的原因和作用有：

1. 不能把测试部门作为二线部门看待。如果聘用和分配政策是所有"顶级"人物（或来自顶级学校的应聘者）都去开发部门，"剩下"的人去测试部门，那么管理层就是在传递错误的信号，强化那条错误信息。

2. 员工只有对测试所需的合适素质和态度才能分配到测试部门。他们应该看到测试的职业发展道路，就像开发人员寻找开发的职业发展道路一样。

3. 补偿模式不应与其他部门，特别是测试部门有差别。如果补偿和奖励机制有利于开发部门，员工肯定会把测试看作是"升到开发部门"的一种手段，而不是把测试本身看作职业。

4. 参与测试、维护和文档活动的工程师应该受到恰当的表扬。表扬应该不仅是奖金方面的奖励或补偿，而是一种承认。例如，如果产品完成并取得成功，我们有多少次能看到"测试设计师"站在舞台的中央而不是开发部经理？这些默默无闻的工作部门有时被认为干的是"得不到感谢的工作"，因此人们希望离开这样的部门。

事实是，由于有与测试部门性质有关的认识上的包袱，一些明显的常识被忽略掉了，这会削弱公司内的测试团队工作，最终影响产品的质量。

13.1.4　"这些人是我的对手"

> 测试和开发团队应该相互支持，而不是争吵。

由于测试的主要功能是发现产品中的缺陷，因此在测试团队和开发团队之间容易产生对立情绪。开发团队开始认为测试团队在对产品吹毛求疵，而测试团队开始认为开发团队没有成熟就交付产品。

13.1.5　"测试是如果我有时间最终会做的工作"

尽管不同的测试阶段散布在项目生存周期中，测试仍然被认为是一种"生存周期的结束"。当最后期限被认为是不能倒的后墙时，当开发人员认为"很自然"地要比计划"略微"推一点进度才能交付产品时，往往是测试团队要承担最后期限的压力。分配给测试的时间被压缩，测试人员常常不得不放弃周末，加班工作赶在最后期限前完成。还要确保产品经过"充分"测试。毕竟当客户发现问题时，第一个要问的问题就是，"为什么这个缺陷没有在测试中发现?！"

> 测试不应该安排在项目的最后，而应该贯穿到项目中，甚至一直持续到交付之后。

在高度紧张的状态下受到双重压力（随着产品生存周期越来越短，这种压力在不断增加），面临产品缺陷的打击，这些都对想加入测试部门的人的士气和信念产生消极的影响。解决这个问题的方法有：

1. 规定测试团队从开发团队接收产品的完成/确认准则。例如，公司可以规定产品能够交付测试的最低条件。

2. 赋予测试团队有强制产品在交付前必须达到最低质量要求的权力。例如，公司可以规

定产品在发布前还没有解决的最高问题数量和严重等级。

13.1.6 "测试的拥有者毫无意义"

在谈论产品开发的团队结构时，常常会提到像"模块负责人"这样的角色。这些人常常是开发模块的拥有者。即使是在维护阶段，模块级或组件级也有拥有者。这种拥有关系会产生一种自豪感，可推动人员提高质量。

> 像开发一样，产生也有可交付产品，因此测试人员也应该有同样的拥有者意识。

测试部门有时没有这种拥有者意识。一个可能的原因是测试部门看上去没有"可交付产品"。开发部门生产代码，开发人员觉得这种"看得见摸得着"的代码是他们的创造，产生一种拥有感。而测试人员会感觉自己只是使代码更好一些——或更差！他们没有把自己与任何具体的可交付产品关联起来。毫无疑问，测试人员拥有测试脚本、测试设计等，但是这些工作产品都不向客户提供，确实会对测试人员的拥有意识产生影响。

减少这个问题影响的一种方法是赋予测试团队确定产品就绪状态的权力，让他们成为质量的最终仲裁。第二，有些测试结束时产生随产品一起交付的用户确认测试用例和安装验证程序后，测试人员在产品中就有了更多可交付产品。最后一点，现在的产品相当复杂，不仅需要交付二进制代码和可执行程序，还要交付样本程序和应用。事实上，大多数产品提供商的网站都提供样本程序或应用，演示产品各种特性的使用。对产品使用有全面了解的测试工程师最适合编写这样的样本应用程序。在产品发布时包含这类样本，可提高测试过程可交付产品可视化的意识，从而提高测试过程的拥有意识。

13.1.7 "测试只是破坏"

有人认为测试工程师只是尽力"破坏软件"，或通过测试证明"产品不能有效运行。"这不完全对。测试的这种"悲观主义"（即要找出缺陷）只是测试工作的一个方面。与测试关联的悲观主义应该看作是职业上的悲观或建设性的悲观。这种建设性的悲观加上好奇心对于建设性的产品开发是必要的。测试工程师不仅会说"什么东西不能正常运行"，他们还会指出产品中的"什么东西能正常运行"。测试工作结合了指出产品中的哪些内容可以正常运行、"破坏软件"、分析发布带有缺陷的产品的风险以及提出综合考虑各种观点的缓解计划。综合考虑各种观点可树立测试工程师的积极形象。

> 测试既是破坏性的也是建设性的，就像一枚硬币的两个面。

仅仅因为测试工程师会带来有关产品的"坏消息"（不能正常运行的消息）并不意味着这个职业不好，或测试就是破坏性的。就是这同一个测试工程师也可以提出绕过缺陷的建议，当客户发现同样或类似的缺陷时，可以节约公司的经费，保护公司的形象。发现问题是解决问题的一半。通过用充分的数据从正面报告问题，测试工程师提供了解决问题的方法，否则在很多情况下很难解决这些问题。

因此，测试工程师的工作不仅是发现缺陷，还要通过从测试角度评审规格说明、设计、体系结构和代码，主动地预防缺陷。通过预先向开发人员提供测试用例，使代码可以结合所有的测试条件，避免缺陷。这样就使测试具有主动性、建设性，并节省发现和修改问题的工作量。测试对主动解决问题的贡献，使表面上的破坏性结合着建设性因素。

13.2　测试与开发工作的比较

上一节讨论的说法和错误概念是由于测试和开发工作有很大不同。以下讨论这些差别。

测试常常是在时间很紧的情况下进行的 尽管在整个软件生存周期有不同的测试阶段和测试类型，主要测试工作还是会集中到接近产品发布的时候。这种接近产品发布时间的集中测试工作会带来一些特有的策划和管理方面的挑战。

在项目的早期阶段一般更具"弹性" 开发工作拖延计划几乎成了正常的事！但是，由于产品发布的最后期限很少能够再商量，因此允许开发活动具有的进度弹性和灵活性通常不允许测试活动有。因此，策划测试项目通常比开发项目的灵活性小。

测试工作对于员工来说是最困难的 由于本章讨论过的所有原因，与其他部门相比，只有很少人愿意把测试作为职业，这使得测试部门很难吸引和保留最好的员工。但是，由于测试部门通常要经受很大的压力，人员方面的不确定性需要更全面的策划，以防出现突发情况。

与开发部门相比，测试部门更多地依赖外部 出现这种情况有两个原因。第一是测试要在项目生存周期的末尾进行。第二是测试活动通常有一些"暂停"情况，有些缺陷需要修改才能继续进行测试。当修改完缺陷再次测试产品时，可能还会发现其他严重缺陷，因此对外部的依赖更严重。

13.3　为测试人员提供职业发展道路

前面概要讨论了关于测试缺乏职业发展机会的认识误区。在积极的测试人员中有一种明显的对职业发展道路的不确定感。本节要分享我们关于测试可以是职业的想法，并再次强调对于有热情、有能力的人来说，测试职业也像其他职业一样，可以得到回报和满足。本节讨论测试人员寻求的职业发展道路，以及职业发展所需要的能力。

当人们探寻测试职业发展道路时（或已经选择了职业），所追求的发展方面包括：

1. 技术挑战；
2. 学习机会；
3. 不断增长的责任和权力；
4. 不断增长的独立性；
5. 对公司的成功发挥重要影响的能力；
6. 奖励和被承认。

为了在以上方面的价值链上向更高方向发展，我们提出一个测试人员可以遵循的分三个阶段的发展框架。

第一个阶段是服从阶段，从这个阶段开始，测试人员和公司之间的信任和信心刚开始发展。大多数专业人员在某个公司开始自己的职业生涯（特别是刚从学校毕业）时，就是按照所给的指示做。在这个职业发展阶段，关于自己必须完成什么任务和如何完成所定义的任务，会得到详尽的指示。这对于个人和公司双方试图相互适应对方的优点和发展协作方法是必要的。因此在这个阶段的工作是：

1. 相当面向任务的；
2. 有详尽的指示；
3. 几乎不需要作决策。

在服从阶段，技术人员通过所分配的任务展示自己的能力，既能使自己被认为是值得信

任的，也能使自己学到所执行任务的缺陷。技术人员要准备从仅通过执行指示完成任务的层次，上升到为自己和其他人进行规划的阶段。到了职业发展的这个阶段，需要更多地关注：

1. 独立地作出决策；
2. 为其他人分配工作方面的沟通；
3. 与其他小组的沟通。

当技术人员达到规划阶段，则已经展现出事前进行计划和协调、监视工作进展的能力。在这个阶段，专业人员承担执行部分工作的能力上升一个层次，从而进入下个发展阶段，承担影响角色，指导和培养其他新人。在这个影响阶段，自己成为角色的模范。公司内的其他人会愿意与这些角色模范接触，听取其对特定任务的建议和忠告。在这个阶段，这些角色模范已经能够在其他方面担负起更多的责任。承担更多责任后就开始在最从阶段开创新领域，可能会在这些新领域中已有角色模范的指导下开展工作。

下面从部门角度看典型测试工程师的职业发展道路。典型测试工程师的职业生涯可能会从执行一些手工或自动化测试开始。这类工作要求严格遵守给定的指示，按照所规定的过程进行反馈。经过一段时间后，测试工程师会逐渐从执行测试发展为使用自动化工具编写测试脚本。这会增加对测试工程师的挑战。下一步就是进行测试设计，这需要对产品和领域的更好理解。这反过来会导致形成测试设计和编码的标准，因此对测试产生更宽广的影响。最后，工程师可以到达影响阶段，参加像评估自动化工具、与外部产品小组交流等活动，包括参加上游的设计评审活动，从而提高测试的有效性。

图13-1给出了测试人员的一种可能的分三个阶段的职业发展道路。首先从测试工程师开始职业生涯。这时所承担的大部分角色都属于最从阶段。在像划分缺陷这样的领域中，测试工程师的工作一般是于规划阶段，这最接近现实。表13-2给出了测试工程师要面对的各种责任。

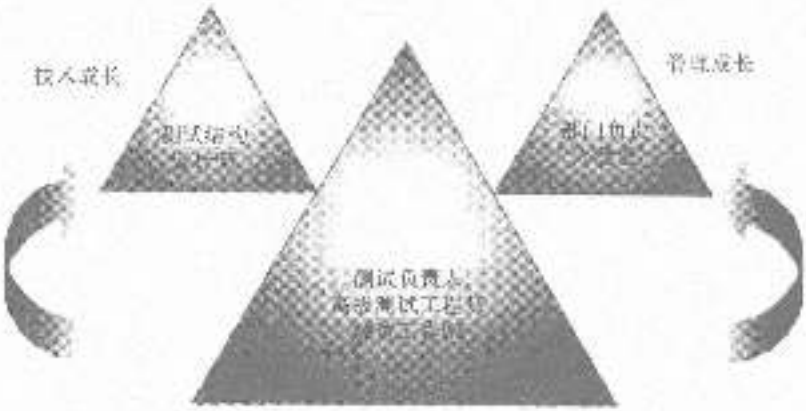


图13-1 测试人员的职业发展道路

表13-2 测试工程师的责任

任　务	技术	规划	影响
遵循测试计划执行测试用例，维护测试用例	✓		
编写高质量的缺陷报告，提示发现缺陷的人员	✓		
为缺陷分类		✓	
遵守所规定的进度计划	✓		
编写高质量的文档	✓		

测试工程师在积累了必要的专业知识，并展示出自己的能力与素质后，就成为高级测试工程师。在这个层次参加测试工程师的活动，更具独立性，较少接受指导。在大量活动中逐渐发展到规划阶段，此外还要承担一些新任务，并逐渐精通（通过服从阶段的积累）。表13-3给出了高级测试工程师要面对的各种责任。

表13-3　高级测试工程师的责任

任　务	服从	规划	影响
遵循测试流程执行测试日程，维护测试基准	✓	✓	
编写高质量的缺陷报告，使开发人员能够使用	✓	✓	
为缺陷分类		✓	
遵守所规定的测试计划	✓	✓	
编写高质量的文档	✓	✓	
帮助开发人员调试和确定问题	✓		
对设计测试过程提出意见	✓		
采集与测试有关的度量数据	✓		

下一个发展阶段是测试领导，主要负责产品的模块级测试。测试领导和模块开发领导密切协作，保证模块达到所期望的功能要求。测试领导要参加相当大量的影响阶段活动，并把工作交给测试工程师和高级工程师完成。对测试领导的沟通能力有更高的要求，包括组内的沟通能力和与其他组的沟通能力。测试领导要成为执行下游测试人员的基点，例如集成测试和系统测试。表13-4给出了测试领导要面对的各种责任。

表13-4　测试领导的责任

任　务	服从	规划	影响
保证模块级的测试日程，测试设计等		✓	✓
编写模块级测试策略和测试计划		✓	
分配测试任务		✓	
监督部分组的任务	✓	✓	
选择模块级测试技术和工具		✓	✓
指导团队成员开发技术上提供帮助			✓
解决技术问题并与开发团队交互			✓
解决测试和问题的协调与产品支持团队交互			
采集和分析与测试有关的模块级度量数据		✓	✓
对模块级的测试加监督全责		✓	✓

测试领导位置对于个人来说是个决策点。人们可以沿着技术方向发展，也可以沿管理方向发展。在管理方向上，可以承担测试经理或部门领导。这种角色的责任集中在人员问题和更高层次的策略问题。测试经理和部门领导的大部分工作属于影响和规划层。如果确定不想沿管理方向发展，而想搞技术，那么可以担当测试结构设计师的工作。这种角色基本上没有直接管理人员的职责，而是更集中在提供总体技术指导、公司价值方面领导者、帮助招聘并指导公司中的顶级有潜力的员工、在测试自动化工具的选择上发挥积极作用等。表13-5给出了测试经理或部门领导要面对的各种责任。

表13-5 测试经理或部门领导的责任

任　务	报从	规划	影响
管理产品或公司级测试策略		✓	✓
在产品或公司级生动质量工作		✓	✓
在组织内部分配资源		✓	✓
确定技术和工具选择		✓	✓
风险分析		✓	✓
团队协作		✓	✓
招聘并在公司内保留顶级有潜力的员工		✓	✓
与员工队或员进行职业发展规划		✓	✓
落实公司级政策	✓	✓	✓
在团队中阐释公司价值		✓	✓
有效管理会议		✓	✓
通过有效沟通使每个人都保持同步		✓	✓

实践证明，这些不同角色在实现测试工程师的成功职业发展中相当有效。为了在不同层次上完成所需的工作，需要很好的知识、技能和态度的组合。知识指知道在不同情况下做什么。技能指理解如何做所要求的事。态度指有内在动力想要做合适的事。为员工提供职业发展道路的能力强烈依赖于对知识、技能和态度的正确组合的认识。表13-6归纳了每个工作层次上所需的各种属性。

表13-6 不同工作层次上测试专业人员所需的属性

属　性	测试工程师	高级测试工程师	测试领导	测试经理	测试结构设计师
知识					
产品知识	低	中	高	非常高	非常高
业务知识	中	高	非常高	非常高	非常高
竞争信息	低	低	非常高但专用	非常高	非常高
测试过程知识	高	高	高	高	高
测试工具使用知识	高	非常高	高	中	高
测试行业最佳实践	低	中	高	非常高	非常高
测试设计评审	低	中	中	高	非常高
测试用例评审	低	中	高	高	非常高
过程方法	低	中	非常高	高	非常高
风险管理	低	中	非常高	非常高	中
测试自动化	高	非常高	非常高	高	中
特性需求和可测试性评审	低	低	高	非常高	非常高
采集测试指标数据	低	低	非常高	非常高	低
持续数据分析	低	低	高	非常高	非常高
评价工具	低	低	中	高	非常高
整体发布决策	低	低	高	非常高	中
技能					
分享知识能力	中	中	高	高	非常高
内部沟通技能	中	中	高	非常高	非常高
跨组沟通技能	低	中	非常高	非常高	非常高
寻求和利用反馈意见的能力	非常高	非常高	高	高	非常高
情况变更反应迅速直接管理	非常高	非常高	非常高	高	中

（续）

属　性	测试工程师	高级测试工程师	测试领导	测试经理	测试结构设计师
对分优先级的能力	低	低	高	高	高
主动提出问题的能力	非常高	非常高	非常高	高	高
客户观点	中	中	高	非常高	非常高
设定目标	低	中	非常高	非常高	高
独立工作的能力	低	中	高	非常高	非常高
有效的状态报告	非常高	非常高	非常高	非常高	非常高
说服与影响他人的能力	低	低	高	非常高	非常高
与团队交互中的形象意识	低	中	高	非常高	非常高
态度					
高输出工作自豪感	中	中	高	非常高	非常高
通过以身作则鼓励人	低	中	高	非常高	非常高
促进团队工作	低	低	中	非常高	非常高
为自己和团队设定和组织相关目标	低	低	中	非常高	高
向团队提供技术指导	低	高	非常高	高	非常高
特性或缺陷测试的决策策略	低	低	高	非常高	非常高
给人员分配任务	低	低	高	非常高	低

除了以上这些，无论公司中那个层次，都有一些共同特点，包括：

1. 做为团队成员，需要理解团队作为一个整体成功的重要性，而不是只关注个人兴趣。前面已经介绍过，测试团队的成功取决于在产品交付前发现缺陷。这不应看作与开发冲突的工作。整个公司的关注点都应该是按时发布高质量的产品，而测试团队的成员也应该记住这个总体目标。

2. 要主动。与大多数其他职业一样，成功的人都是主动、自觉奉献、积极实践并与其他人分享心得的人。测试职业也不例外。

3. 做个不断的学习者。随着技术的迅速发展，个人能够迅速适应新的技术、过程和工具是最基本的要求。

4. 能够对变化作出迅速响应，同时不放弃主动策划。测试需要团队成员具有很大的灵活性，特别是因为测试是一种下游活动。测试团队应该能够对优先级的变化、开发产品拖延提交和员工流动等快速作出响应。同时，测试团队还要尽可能早地预测出这类突发事件，以便为更好地应对这些变化作准备。

通过以上讨论，显然对测试职业发展没有出路的担忧是完全没有必要的。对于积极和有能力的专业人员来说，测试有足够的挑战，足够的回报和足够的职业发展机会值得奋斗。

13.4　生态系统的角色与行动要求

以上讨论的错误看法、概念和问题并不对每个公司都适用，还需要采取一些总体的，更高层次的行动。这些行动适用于涵盖教育系统、高级管理层和作为整体行业的整个生态系统。本节将讨论这些问题并提出满足生态系统要素所需的行动。

13.4.1　教育系统的角色

教育系统并没有对测试给予充分重视。以下是在大部分大学中普遍存在的一些现象：

- 关于程序设计有正式的核心课程，但是没有几所大学提供有关软件测试的核心课程；

大多数大学没有有关测试的正式的完整课程。即使有，最多也不过是人数不多的选修课程。

> 正确的价值只能更有效地被学生抓住，而不是被老师教授！

- 有针对各种开发工具的“实验课程”，但是没有或极少有针对常用测试工具的。
- 即使在像操作系统和数据库这样的课程中，对练习和实践工作的强调也只是放在程序设计而不是对所构建产品的有效测试上。结果，学生最终会误认为生产操作系统或数据库的工作到编码完成就结束了！
- 作为课程一部分的大多数“项目”（需要一个完整学期），从来不要求作测试计划，也不研究测试有效性问题，几乎所有重心都放在编码上。由于学生要受到回报的牵引，他们几乎完全忽视测试工作，在开始其职业生涯时就养成了必须“纠正”的习惯。
- 大多数课程和项目在鼓励学生获规方面很少考虑团队工作的培养，而团队工作对于开发和测试工程师的成功是非常重要的，最重要的是，对于测试工程师来说，绝对是立足基石的沟通和软技能在大学中从来不被强调或正式讲授。
- 现实世界的现象，例如不断变更的干扰和这种变更对产品质量的影响，以及对测试和质量保证方法的要求等，很少被强调。这降低了学生对变化的迅速反应能力。

需要尽快采取一些措施，提高对测试重要性的认识，并教授测试所需的知识、技能和态度，包括：

- 对计算机科学、软件工程和信息技术专业的所有学生开设有关软件测试的必修课；
- 在学位程序中更加关注沟通和软技能；
- 要求每门课程的实践部分不仅关注程序设计问题，还要关注系统地对所开发产品进行测试的问题；
- 在项目工作中，留出一定比例的分数给制定测试计划和采集合适的测试指标数据。

总之，学术界必须向学生灌输正确的价值体系，认识到测试工作的重要性和价值。这不仅需要适当修改评价准则等具体工作，还需要转变学术态度。

13.4.2 高级管理层的角色

公司的高级管理层对于发展测试职业具有至关重要的作用。只是说“质量是我们最关注的事”或“人是头等重要的”是不够的。这种承诺要转化成看得到的行动，高级管理层可以采取的一些具体措施包括：

> 公平对待并认可测试专业人员，不仅要做到，而且要做得被看到。

1. 保证把有发展潜力的员工公平地分配到测试部门。比如，通过把一些从“象牙塔尖”学校毕业的员工分配到测试部门，高级管理层发出对于质量的承诺和重视测试部门的信号。
2. 不允许开发工程师看不起测试工程师。要平等对待，他们都是平等的共同利益方，都对产品的成功做出了贡献。
3. 鼓励并有意识地保证开发、测试和支持部门之间的人员轮换。这种岗位轮换会提高公司内部的凝聚力，最大限度地降低测试人员被“看不起”的机会。
4. 在承认和奖励机制上展现对所有部门表现好的员工的平等。
5. 通过为测试专业人员提供进一步学习知识和技能的机会提高他们的能力。比如鼓励他们参加测试界的活动，例如会议，奖励他们参加各种认证培训等。这可以产生测试的角色榜样，在公司内得到高度重视，从而鼓励更多的员工把测试选作职业。

怎样发现好的测试人员？

寻找以测试为骄傲的人……如果眼睛里没有流露出兴奋，脑中也不大可能燃烧激情之火。

寻找没有编列学的简历……如果应聘者连自己的简历都测试不好，也不大可能测试好产品！

考验应聘者选开发还是选测试……踏实肯干的测试人员是不会上当的。

考察其对产品领域的总体理解……好的测试人员能更好地理解全局，并把这种理解直观地表现出来。

我们引用了阿加莎·克里斯蒂（Agatha Christie）《原告的证人》中的一段话，作为高级管理层在发展测试职业方面的角色总结。

13.4.3　测试界的角色

不管高级管理层或学术界做什么，测试界的成功总是从内部开始的。有一些事是测试界，也就是测试专业人员要做的，以展现和推动测试专业。

做为测试界的一员，你是否自豪并有平等意识？请记住，权威是争取的，不是赐予的！

测试人员首先要对自己的工作有自豪感，更全面地发挥自己的角色作用，对整个产品有全局观。缺少自豪感往往会产生自己不如开发人员的表象，从而陷入自我满足的境地。

对工作的自豪感可能适合用一个老故事解释。

有三个人在建筑工地上干活。

一个过路人问第一个工人，“你们在做什么？”

“哦！我在混日子，这就是我的命，”他回答说。

过路人又问第二个工人同样的问题，“我在加工石头，”第二个工人回答说。

于是过路人又问第三个工人同样的问题。

第三个工人充满热情地回答说，“我在帮助建一座很快就在这个工地上矗立起来的大教堂！”

第一个工人就像把测试作为权宜之计的测试人员，抱怨自己命不好，没有得到开发工作。第二个工人就像做测试，但看不到工作意义的人。

第三个工人看到了自己工作的价值，并理解全局。

测试人员自己应该把测试看成是一种职业，而不是权宜之计或转到开发部门的手段。希望前一节描述的职业发展道路能够激励人们把测试作为一种职业。

测试工程师不仅要遵循已有的技术，还可以在形成和使用新技术上起积极作用。测试工程师很难理解新技术，只有“开发人员”才能更好地理解的观点是错误的。有时测试工程师面临要同时学习技术和测试产品的压力，同时学习产品和测试可能会降低测试的有效性。这种状况必须改变。理想的情况是，在实现产品之前测试工程师就主动了解了新的技术领域。这可以通过成为产品开发团队的一部分并作出贡献，保证新技术在产品中被正确采纳，以使客户受益来实现这种目标。测试工程师主动参与了解新技术，可以引入更好的用户视角。从

而提高产品的市场潜在价值。

开发和其他软件工程领域有紧密联系的一个原因是存在分享经验和知识的社区和论坛，有许多涉及各种开发问题的会议和研讨会，而纯粹有关测试的论坛却非常有限。好消息是这种状况最近几年正在迅速发生变化，重要的是要保持创建和维护测试专业论坛的动力，可以采取的行动包括：

- 在承认和宣传测试重要性方面开展工业界和学术界的合作；
- 设立基金奖励测试专业的杰出贡献者；
- 组织关注测试的活动。

以上介绍的生态系统中各个方面的工作可以尽早地发现优秀测试专业人员，拓展他们的知识、技能和理念，使他们能够更好地满足实际需求，他们可以继续在测试职业上发展，理解测试职业的商业价值，并对整个行业的贡献感到自豪。

问题与练习

1. 有人说测试工作没有挑战性，你怎样反驳？
2. 本章只讨论了两种工作部门，分别叫做“开发部门”和“测试部门”。请考虑其他部门，例如支持和维护部门。
   a. 比较测试和这些部门的职业发展道路。
   b. 比较测试和这些部门的性质，指出相似和不同点。
3. 请讨论怎样在所有部门，即开发、维护、测试和支持部门之间实行轮换，才可以对组织有效益。
4. 你赞同以下高级管理层说的哪些话？请说出理由。
   a. “如只你做半年测试，我就让你做Java开发。”
   b. “我们的聘用政策是聘用本科生（刚毕业）去测试部门，本科生不能去开发部门。”
   c. “每个开发人员都必须去测试和支持部门轮换工作三个月。”
   d. “所有部门表现最好的员工所得到的奖金分配比例应该是一样的。”
5. 如何回答测试工程师的以下说法？
   a. “他和我都从同一所大学毕业，我的成绩比他好，为什么他去开发部门而我只能去做测试？”
   b. “我在测试岗位上表现得不好，因为我说过我对测试不感兴趣……把我调到开发部门，看我的表现吧。”
   c. “我喜欢测试工作，告诉我在测试中我可以学到哪些新东西吧。”
   d. “我已经选择了测试——难道我不应该得到更高补偿吗？”
6. 你有一个测试工程师团队，其中有些人你认为可以发展为高级测试工程师。你要为他们开列哪些培训内容？
7. 如果有人想从测试结构设计师调到测试经理岗位，
   a. 他（她）会遇到什么挑战？
   b. 需要接受什么培训？
   c. 为了在新岗位上取得成功，自身要作哪些改变？
8. 管理层的任务之一是“推动产品和公司级质量工作。”请进一步研究这个问题，并细分为多个活动。

第14章 测试团队的组织结构

14.1 组织结构的要素

本章将介绍典型测试组织中的各种组织结构。（我们只讨论直接进行开发和测试的组织结构，不包括公司的其他部门，例如财务、行政等。）

上一章讨论了直接与人员问题有关的组织结构问题。另外，组织结构不仅对有效性是非常重要的，因为设计良好的组织结构可提供对结果的可审计性。这种可审计性能够促进不同要素之间的团队协作，并能更好地关注工作。此外，组织结构还为团队成员的职业发展提供了路线图。

我们从两个方面讨论组织结构：一个是组织类型，一个是地域分布。

我们把公司类型大致分为两大类：产品公司和服务公司。产品公司生产软件产品，对整个产品有一至“从生到死”（或从设计、开发、测试和维护到产品退出使用）的责任。测试是其中的一个阶段或公司的一个组。本章要讨论的产品公司是多产品公司。服务公司没有完整的产品责任。在测试方面，他们是向有需求的其他公司提供测试服务的外部公司。从本质上看，测试服务是对这类公司的外包。这类测试服务公司有专门人员进行测试。他们还承担专门和特殊领域的测试工作，例如性能测试、国际化测试等。

决定组织结构的第二个关键要素是团队的地理分布。包含测试的产品或服务公司可以是单一场地也可以是多个场地。对于单一场地的团队，所有成员都在一个地方工作，而多个场地的团队则分散在多个地点。多个场地的团队有影响组织结构的文化和其他因素。

14.2 单产品公司的结构

产品公司一般都有与图14-1类似的高层组织结构。

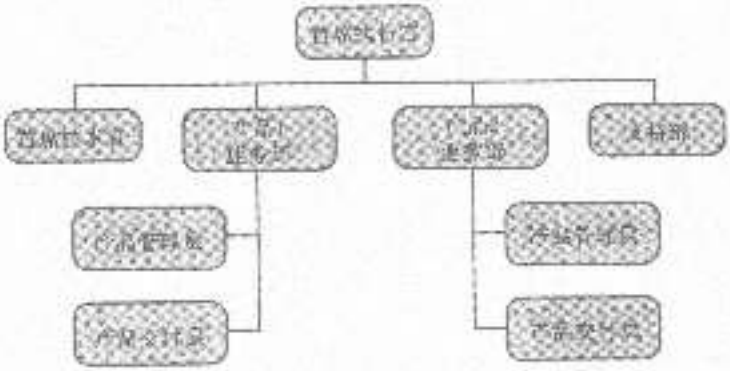


图14-1 单产品公司的组织结构

注：图14-1中只给出与本章的讨论直接有关的小组。这里没有包含像销售、财务和行政、硬件基础设施等小组。

首席技术官负责公司的高层技术指导。业务单元负责公司所生产的每个产品。（有时业务单元也处理产品的相关事务，以形成产品线。）产品业务单元又分成产品管理组和产品交付组。产品管理组负责结合特定的市场需要整理首席技术官的指示，形成产品路线图。产品交付组负责交付产品，并处理开发和和测试工作。这里使用“项目经理”这个词表示这个部门的领导，有时也使用“开发经理”或“交付经理”。

图14-1表示典型的多产品公司。交付团队的内部组织对于单产品和多产品公司的不同场景是不同的，这些将在以下讨论。

14.2.1 单产品公司的测试团队结构

大多数产品公司都是从单一产品开始的，在进展的初期，公司没有很多正式化的过程。产品交付团队成员把自己的时间分摊在多个任务上，常常身兼多种岗位。所有工程师都向负责整个项目的项目经理报告，测试工作和开发工作没有多大区分。因此，在“开发团队”和“测试团队”之间并没有很明显的界限。

如图14-2所示的模型适用于产品处于早期进展阶段的场景。在图14-2中，“项目经理”负责产品的一部分或全部。这里有意没有给出开发和测试部门的多层管理，因为在小公司的开发早期阶段，只有很少的管理层次，承担“经理”、“负责人”等岗位工作的人实际上也是“工程师”，也要完成工程工作。

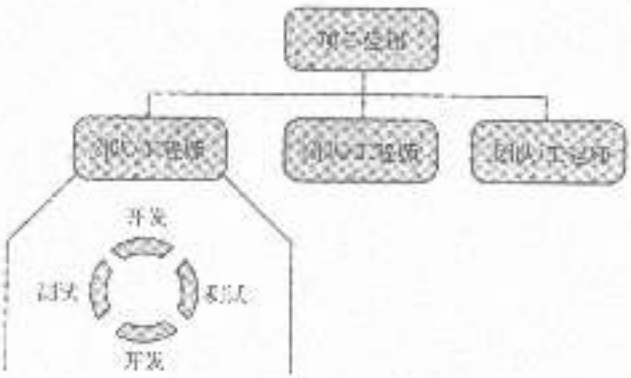


图14-2 产品早期阶段的典型组织结构

这种模型的一些优点非常适合小公司：

利用了测试活动主要在后期的特性 尽管测试活动分布在整个项目生存周期，但是测试的主要关注点和压力点还是在项目生存周期的后期。因此，这种组织结构提供了一种自动任务平衡机制，在项目的早期阶段，每个人都做开发，而在后期，他们又承担不同岗位的工作，转到测试上来。

使工程师获得生存周期所有方面的经验 由于工程师把自己的时间分别投入到开发和测试中，因此了解所有生存周期活动所包括的一切内容。这也使他们了解所有活动的重要性和困难，为日后划分不同小组后的更好的团队协作打下了基础。

尊重这样的事实，即公司主要使用非正式的过程 处于早期的公司通常只有非正式过程，可能没有定义完备的产品从一个阶段转入下一个阶段，比方说从开发到测试的进入和退出准则。因此，在开发和测试之间复用相同的资源是与公司的非正式性质相适应的。

会在早期发现一些缺陷 由于开发人员执行测试工作，因此有可能在时间上更早发现缺陷（很像白盒测试所起的作用）。

但是这种模型有严重的弱点，使公司很快发生偏离。这些弱点包括：

降低了测试和质量的责任 由于相同的人员从事开发和测试工作，使得测试和质量的责任受到挑战。开发人员（或项目经理）会把交付日期放在第一位，还是会执行开发的空做测试？他们更可能把时间用在开发上，增加一些新特性，甚至以短性彻底测试已有特性为代价。因此，测试的责任受到损害。

开发人员一般不喜欢测试，因此测试的有效性受到损害 第13章介绍过，开发人员一般不喜欢执行测试活动。因此，在这种模型中，他们承担测试任务是承担开发"酷任务"的"代价"。这种对承担测试工作缺乏内在动力，会对测试的有效性产生明显影响。

进度压力一般会使测试受到威胁 压力当前，最后期限决定生存！总会有很多最后期限必须达到，可能没有足够的时间进行测试。因此，与典型的工程师更愿意承担开发任务而不是测试任务相一致，测试的质量也会受到损害。

开发人员可能难以完成不同类型的测试 前面几章介绍过，测试有各种不同的类型。开发人员可能不具备完成所有类型测试的能力，因为有些测试需要专门的基础设施或技能。例如，性能测试就不是一般产品开发人员能够有效完成的，因为必须深入了解典型的工作负载，使用特殊的工具，等等。因此，有必要至少把这些特殊的测试工作独立出来，成立不同的小组。

随着产品的成熟和过程的进展，既做开发又做测试的匀匀单产品公司会分为两个不同的小组，一个做开发，一个做测试，如图14-3所示。这两支团队是对等的，都向负责整个产品的项目经理报告。在这种模型中可以去除上一个模型的一些弱点。

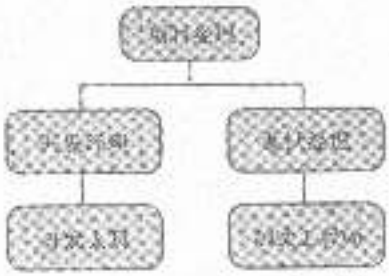


图14-3　分开的测试和开发小组

1. 测试和开发有明确的责任。可以更明确地确定和划分两个团队的结果和预期。

2. 测试提供一种外部的视角。由于测试和开发在逻辑上是分开的，因此不太可能出现前一个模型中出现测试人员要证明产品能够正常运行的偏见。这种外部视角会导致发现产品中的更多缺陷。

3. 考虑测试所需的不同技能。前面已经介绍过，测试工作所需的技能与开发工作所需的技能有很大不同，这种模型承认这种技能要求上的差别，并可以主动地应对。

有效运用这种模型有几点必须注意。首先，项目经理不应该屈服压力，忽视测试团队的发现和建议，发布没有达到测试准则的产品。其次，项目经理必须保证开发和测试团队不能相互对立。这会阻碍两个团队之间的协作，最终影响产品的进度和质量。最后，测试团队必须从一开始就参与项目决策，并对进度作出合适的安排，以免他们"仓促"地介入项目，面对不现实的进度要求和预期。

14.2.2 按组件组织的测试团队

即使公司只生产一种产品，产品也会由可以组装成一体的多个组件组成。为了更好地确定责任，每个组件都可能由一个独立的团队开发和测试，所有组件由向项目经理报告的单个集成测试团队进行集成。每个组件团队的结构可以是融合在一起的开发－测试团队（就像上面的第一种模型），也可以是将测试和开发责任分开的团队。这是因为并不是所有组件都具有同样的复杂性，并不是所有的组件都具有同样的成熟度。因此，针对不同组件的不同组织结构的非正式混合匹配，由核心权威保证总体质量更有效。图14-4描述了这种模型。

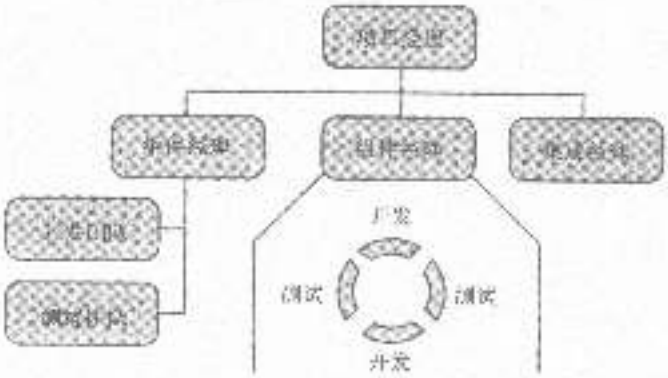


图14-4 按组件组织

14.3 多产品公司的结构

当公司作为单一产品公司取得成功后，可能决定开发其他产品。在这种情况下，每个产品看作一个独立的业务单元，负责产品的所有活动。此外，与以前一样，也会有像首席技术官这样的通用角色。

多产品公司中的测试团队组织大致可用以下要素描述。

与产品联系的紧密程度取决于技术 如果不同的产品使用类似的技术，那么对这些类似产品的测试也会有协作。例如，如果公司在做不同语言的编译器，则有可能把测试力量分散在多个编译器上。但是，如果公司产品类型的范围很宽，像操作系统或数据库这样的系统软件产品，到像工资系统这样的应用软件产品，那么就不大可能由各种产品共享测试资源，因为两种工作所要求的技能是不同的。

不同产品之间的依赖 如果产品之间的依赖很强，则产品的测试也要紧密结合。一个产品中的变化会对其他产品的功能产生严重影响。因此，测试团队可能有必要在不同产品上重叠，至少需要一个全公司层次的测试团队，以实施不同相关产品的集成测试。

产品发布周期的同步方式 如果不同产品的发布周期是同步的，则意味着两点：一，产品之间很可能有依赖关系；二，很可能同时需要测试资源。这看起来是矛盾的要求——一方面由于存在依赖关系需要测试团队具有共性（请参阅上一段），另一方面，由于不同产品会同时需要测试资源，资源可能是不同报告结构的不同团队的一部分。

每个产品的客户群以及不同产品客户群之间的相似性 由于测试类型和所要求的技能在很大程度上取决于产品的性质，而产品的性质又是由客户群决定的，客户群的性质对不同产品的团队是否相同有影响。

根据以上因素，多产品公司在组织测试团队方面有多种选择：

1. 一个集中式的“智囊团”团队，负责制定公司的测试策略。

2. 一个测试团队负责所有产品。

3. 每个产品（或一组相关产品）对应一个不同的测试团队。

4. 每类不同的测试对应不同的测试团队。

5. 以上所有模型的混合。

14.3.1　测试团队作为“首席技术官办公室”的一部分

有些情况下，测试团队在产品生存周期后期参与，而设计和开发团队在初期介入。但是产品的可测试性与产品开发一样重要（如果不是更重要的话）。因此，为测试分派与开发一样的重要性是有道理的。做到这一点的一种方法就是让测试团队直接向首席技术官报告，与设计和开发团队对等。这种模型带来的直接好处是：

1. 开发可测试的也就是适合测试的产品体系结构。例如，最好在体系结构和设计阶段描述非功能测试需求，通过把测试团队和首席技术官联系起来，使产品设计者更能够记住测试要求。

2. 测试团队会有更好的产品和技术技能，这些技能可以在产品生存周期中积累。事实上，测试团队甚至可以对产品和技术的选择作出有价值的贡献。

3. 测试团队可以清楚地理解设计和体系结构所针对的内容，并相应地策划测试。

4. 产品开发的技术路线图和测试包开发能够更好地同步。

5. 对于多产品公司，首席技术官的团队可以推广和优化公司内各种产品组织和业务单元的测试经验。

6. 首席技术官的团队可以制定针对测试自动化的一致、有效的策略。

7. 体系结构和测试的责任集中到首席技术官一个人身上，体系结构的端到端目标，例如性能、负载条件、可用性需求等，都可以无歧义地满足和预先策划。

在这种模型中，首席技术官只负责体系结构和测试团队。实际编写产品代码的开发团队可以向负责代码的另一个人报告，这样可以保证测试团队的独立性。

这种向首席技术官报告的小组解决了在公司层次上分割并需要主动策划的问题。让他们向首席技术官报告的理由是，这种团队可能是跨部门、跨工种的。这种报告结构提高了团队的可信性和权威。这样，他们的决定就可能被公司的其他部门更顺利地接受，提的问题也会更少，不会有“这种决策不适合我的产品，因为是其他人作出的决策”这样的反对意见。

这种结构还能解决一些顶级测试工程师的职业发展道路问题。人们常常在达到测试职业的顶峰后产生错觉，认为要再向前发展必须转去开发。而在这种模型中，测试角色要向首席技术官报告，具有高度可视性，可以激励测试人员树立好的发展目标。

为了使这种向首席技术官报告的团队更有效，应该注意以下几点：

1. 团队的数量要少；

2. 团队之间要平等，最多也只能有很少的层次；

3. 应该是全公司范围内的团队；

4. 应该有决策和实施权，而不只是一个只提建议的委员会；

5. 应该定期进行评审，以保证团队运转与所制订的策略一致。

14.3.2 针对所有产品的单一测试团队

在多产品公司中也可以使用单产品公司的单测试团队模型。本节讨论了组织测试团队的一些准则。根据这些准则，如果产品之间的界限不是很鲜明，那么有可能采用面向所有产品的单一测试团队模式。

这种模型类似于把单产品团队按组件划分，每个组件由一个独立的团队开发。这两种模型之间的主要差别是，前一种模型的测试团队要向项目经理报告，项目经理负直接的交付责任，而在多产品公司中，由于不同小组和个人对不同的产品负责交付，因此测试团队必须向不同层次的管理人员报告。这有两种可能性。

1. 单一的测试团队可以形成“测试业务单元”，并向这个单元报告。这类似下一节要讨论的“测试服务”模型。

2. 测试团队可以向前面讨论过的“首席技术官智囊团”报告。这可以更容易地落实标准和规程，但会降低首席技术官智囊团的功能，甚至降低战略性，更多参与操作层面的工作。

14.3.3 按产品组织的测试团队

在多产品公司中，如果产品之间相对独立，建立单一的测试团队会不太自然。单测试团队的责任、决策和进度安排都会成为问题。组织测试团队的最自然和有效的方法是把一个产品的所有方面的全部责任分配给对应的业务单元，让业务单元负责人策划如何组织测试和开发团队。这非常类似多组件测试团队模型。

取决于产品之间所要求的集成层次，可能需要一支核心集成测试团队，负责处理所有多个产品的集成问题。这种集成团队应该跨多个产品，因此最好向首席技术官智囊团报告。

14.3.4 针对不同测试阶段的独立测试团队

到目前为止，把“测试”看作是一种单一的同类活动，但是在现实中并不是这样。

- 我们看到，需要完成不同类型的测试，例如黑盒测试、系统测试、性能测试、集成测试、国际化测试等。
- 执行这些不同类型的测试需要不同的技能。例如，对于白盒测试，需要有丰富的程序代码和程序设计语言知识；对于黑盒测试，需要外部功能的知识。
- 每种不同类型的测试会在不同的时间点上实施。例如，在国际化测试中，有些活动（例如使能测试）要在生存周期的初期实施，而伪语言测试要在产品本地化之前进行。

由于以上因素，常常把测试工作分为不同测试类型和阶段。由于不同类型测试的性质不同，负责具体类型测试的人员不同，执行不同测试类型的人最终属于不同的小组。表14-1给出了一种组织执行不同类型测试的人员的方法。

表14-1 执行不同类型测试的人员组织

测试类型	承担测试的团队	理由
白盒测试	开发团队	白盒测试本质上与代码很接近，（原则）由开发人员自己编写并运行测试用例
黑盒测试	测试团队	这是产品“外部测试”的统一级，因此非常适合测试团队承担，理由同上

（续）

测试类型	承担测试的团队	理　由
集成测试	公司级测试团队	集成测试需要把多个软件或多个产品组装在一起，因此非常适合公司级测试团队承担
系统测试	产品管理或产品市场开发团队	系统测试需要在真实场景下测试，因此与确认测试一样，可以是产品管理或产品市场开发团队工作的一部分
性能测试	中心基准小组	（第7章介绍过），性能测试是一种很特殊的测试，可能与基准和行业标准有关。对于多产品公司，还会有影响性能的产品之间的依赖关系
确认测试	产品管理或产品市场开发团队	确认测试实际上是客户确认的一种代理，因此由产品管理或产品市场开发团队实施是很合适的
国际化测试	国际化团队和一些本地团队	国际化涉及不同层次的测试，（第9章讨论过），有些层次的测试甚至涉及本地语言和习惯的知识，因此责任应该分散
回归测试	所有测试团队	有些回归测试也属于冒烟测试的一部分，这部分测试由产品测试团队完成，公司必须建立在测试团队之间交接工作的规程

这种基于测试类型的组织有以下优点：

1. 有合适技能的人员参加特定类型的测试。

2. 可以更好、更及时地检测出缺陷。

3. 这种组织与V字模型一致，因此能够更有效地分配测试资源。

需要注意的挑战是，测试责任现在被分散了，因此没有一个统一的测试责任点。解决这个问题的关键，是为每个测试阶段或小组客观地定义指标，并跟踪直到结束。

14.3.5 混合模型

以上模型并不是相互排斥或不相交的。在实践中组合使用这些模型，并不时改变所选择的模型。例如，在临近产品交付的紧张时期，多个组件测试团队可以作为一个组件测试团队运行，当测试集成多个产品出现的问题时，不同的产品团队可以合成一个团队，在这个期间向首席技术官或首席执行官报告。可以把以上介绍的各种组织结构看作构件块，可以根据实际需要，以各种排列组合方式组装在一起。这种混合组织结构的主要目的，是在责任明确前提下提高有效性。

14.4 全球化与地域分散的团队对产品测试的影响

14.4.1 全球化的业务影响

全球化为软件产品的生产和维护方式带来了革命。本书前面已经介绍过，

1. 软件产品的市场正在全球化。因此，软件在全球各地的生产对于利用本地条件是很有必要的。

2. 由于市场是全球化的，产品必须满足的需要呈指数增长。因此，一个地点的资源不可能满足所有需求。

3. 世界上的多个国家拥有丰富的优秀人力资源，需要有效加以利用。

4. 很多国家不仅有丰富的优秀人力资源，还有成本优势，使他们具有很好的业务竞争优势。

软件产品团队分散在多个国家的现象越来越常见，他们协同工作，开发测试和交付单个产品，测试在全球化中受益很大，并推动最大限度地利用全球化的潜力。这方面的因素包括：

1. 在成熟的公司中，测试已经完备地定义为一种过程，因此，只要经过过程方面的良好培训，并配备必要的技术基础设施，就可以在任何地方的团队中执行。

2. 当测试的自动化程度很低时，测试只能是手工过程，因此很难在所有国家找到能够或愿意承担测试工作的人。

3. 随着测试自动化程度的提高，自动化开发通常是与要发布产品的测试并行的一种活动。这种程度的并行性有利于利用多个地点的资源。

4. 测试产品的较老版本被认为是一种低风险的活动，但是对于支持现有客户是必不可少的。可以选择成本低的地点完成这种活动。

跨多个地点的分布式测试团队大致有两种组织机构：全时区开发/测试团队模型和测试能力中心的模型。

14.4.2 全时区开发/测试团队模型

推动这种模型的主要因素有：

1. 开发和测试是交替、迭代进行的，即先进行代码开发，然后进行测试和清除缺陷。这个循环一直持续到通过测试认为结束。由于开发和测试交替进行，因此可以看作是两个独立的工作单元。

2. 由于开发和测试交替进行，因此可以在一天中的一个时段进行开发，在另一个时段进行测试。

3. 可以利用不同地点的地理时差"拓展"一天的时间，因此可以利用一个地点是夜里，另一个地点却是白天这一点有效地进行工作。

以一个团队的一部分在美国的加利福尼亚，另一部分在印度为例。由于印度和加利福尼亚存在时差，当加利福尼亚是白天的时候，在印度是夜间；当印度是白天的时候，加利福尼亚是夜间。如果把开发任务分配给加利福尼亚团队，把测试任务分配给印度团队，那么两个团队的工作时间就可以覆盖24小时，而每个团队并不连续工作24小时。图14-5给出了这种典型的工作流程。

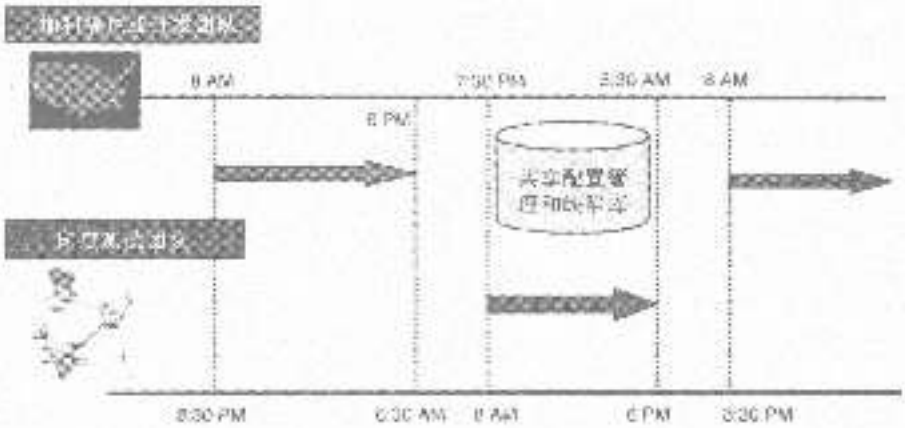


图14-5 加利福尼亚开发团队和印度测试团队之间的工作流程

1. 加利福尼亚的开发人员从当地时间上午8点到下午6点工作，对应的印度时间是晚上8点30分到（次日的）早晨6点30分。

2. 加利福尼亚的开发团队在下班之前，把最新的软件版本存入配置管理库。

3. 印度的测试人员上午8点来上班（此时是加利福尼亚前一日的晚上7点30分），从配置管理库中取出最新软件版本，运行任何需要运行（并能够运行）的测试用例，把发现的问题放入缺陷库。当印度的测试人员下午6点下班时，是加利福尼亚的早晨5点30分。开发人员已经能够在下一个版本中修改那些所发现的错误，并进入次日的下一个周期。

在上面的模型中，差不多把时间利用到了极致，一天24小时中只有大约三个小时没有活动。这种模型可以扩展到多个地点的多个团队，利用这部分时间。不过需要注意的是，通信负担是否能够抵消充分利用时间带来的效益。

对于以下情况，这种模型是自然和有效的：

1. 测试作为一种过程已经完备地建立，而且不需要测试人员介入就可以成功地运行测试用例。

2. 不管是白天还是夜间，任何时间在两个团队之间都有沟通渠道，以便一个团队需要另一个团队作出澄清时，不会在通信上浪费时间。

这种模型一般适合产品达到一定的成熟度和稳定性的情况。在产品开发的早期阶段，代码很可能不稳定，因此测试会经常遇到影响继续进行的缺陷。也就是说，当测试团队开始测试时，会很早就发现致测试不能继续进行的缺陷。例如，如果测试一个网络层产品，产品甚至不能建立连接，这样，像错误控制、流量控制等进一步测试都不能执行，必须等到连接缺陷被清除才可以继续测试。在这种情况下，开发团队必须立即介入，也就是说在他们的夜间工作，这样就失去了利用地理时差的意义。

解决这种问题的一种常见办法是采用一种“值班”系统，在两地团队内都设置24小时值班的代表。即使这样，由于产品不稳定引起的共同开销也使这种模型不适合产品周期的早期阶段。

14.4.3　测试能力中心模型

在这种模型中，建立一个叫做测试能力中心的逻辑组织，这种组织可以潜在地分布在多个地点，有效地利用时差和可以得到的技能资源。这种中心具有很高的测试专业性，对产品的质量和发布最终期限有最终发言权。这种测试能力中心模型有两种变种。

在第一种变种中，中心是一种共享（并且常常也是稀有）资源，被多个产品开发小组共享。前面介绍过，这种中心具有其他部门没有的测试能力，因此其他小组需要其服务。产品开发小组事先“预约”测试能力中心的时间，由于测试能力中心具有很高的技能和高度成熟的过程，管理层对他们的建议充分认可。测试能力中心可以是公司内部的，也可以是测试领域独立的外部公司。可以把测试能力中心看作一种“认证权威”，产品在发布之前需要得到其“祝福”。换句话说，测试能力中心的认证可提高对产品的信心。这种模型不能节省成本，事实上可能还很昂贵，因为测试能力中心具有专门性。这种模型没有利用时区优势，中心的位置完全取决于在哪里可以找到高级测试人员。这种模型既可以用于产品公司，也可以用于服务公司。

测试能力中心的另一种变种是拥有一组专门的测试队伍，从项目的早期阶段就参与全周期的测试活动。这种变种一般遵循以下过程和结构：

1. 这种中心的成员从产品概念的早期阶段就开始策划测试活动，涉及产品测试的所有活动都需要这个中心的成员参加，请注意，在这种模型中，中心本身的地理位置并不重要。

2. 在产品功能策划和发布时，产品团队和来自测试能力中心的测试团队确定被测产品的特性并指派优先级，以及要测试特性的地点。这个过程要保证最重要的特性得到优先测试，恰当地点的恰当资源能够在合适的时间测试恰当的软件特性。

3. 被测特性的选择和对不同地点测试中心的任务分配有以下几点需要考虑：

a) 使用类似特性的过去经验；

b) 完成该工作所需的技能，包括所需自动化工具的使用；

c) 是否有硬件和软件资源（包括当地是否有这些资源的支持）。

4. 在生存周期的早期，中心的测试团队成员花时间与开发团队（如果有必要，还包括产品管理团队）一起，理解被分配测试的产品特性细节。此时，他们要得到有关产品外部，必要时还有产品内部功能的培训。这构成测试策划和测试用例设计的基础。

5. 这样，测试用例设计由当地测试中心更大的团队转换为测试用例。他们还执行这些测试用例，验证测试用例是可用的（符合第1章所说的"首先测试测试用例"格言）。这就是测试用例的第一条基线。

6. 如果产品相当稳定，那么测试中心所在地的测试团队还可以承担随着代码的进展需要做的重复测试。如果产品不够稳定，和开发团队同在一地的小测试团队要在早期阶段执行重复测试。

7. 测试中心同时还要承担测试用例自动化的工作，逐渐把运行自动化测试作为构建软件版本的一部分工作。由于测试是自动完成的，服务所处的具体位置并不重要。

第二种变种最适合由于要增加或增强新特性，产品的版本经常需要变更的情况。在这种变种中，由于中心关注的是把测试作为一种能力开发，因此与本章稍后要讨论的"测试作为一种服务"的模型有些类似。

14.4.4 全球团队面临的挑战

文化挑战 不管全球团队在什么时候一起工作，文化和语言总会成为沟通的重要障碍。没有有效的沟通，开发和测试之间的界限只能加深。当在一个新的地点建立测试团队时，重要的是两地的团队要相互尊重对方的文化、语言口音等。可以参考以下办法：

1. 策划一个地方的成员到另一个地方的定期出差。测试提供了策划这种出差的自然方法，例如确认测试、多地点开发模块的集成测试、被测新产品的培训等。从长远观点看，把这种旅行列入项目计划中是很重要的，通过旅行可以保证测试团队成员能够体验得到其他地点的文化和语言。

2. 定期召开视频和电话会议。

工作地点挑战 当团队分散在各地时，重要的是要有明确的工作分配政策，所有团队都能一致地理解这种政策。不能产生这样的感觉，所有的"美差"都给了一地的团队，而"苦差"都给了另外一地的团队。整个团队都必须为自己的工作而骄傲，只有这样团队才能工作更有效，并能发展自己，使产品获得成功。

团队之间的对等 如果不同地理位置的团队承担类似的工作，他们会预期得到补偿和奖励系统规定的对等待遇。"对等"是一个有争议的词。首先，不同地点的团队应该有对等的工作内容。其次，团队应该在所有权、权威性、委派性和自由度等方面对等。最后，团队应该得

到补偿和奖励系统规定的对等。工作在美国的人得到的是美元工资，而工作在印度的人得到的是印度卢比工资，这样两地同工种员工工资的绝对价值是不同的。由于两个国家的消费水平和经济状况不同，这种差别是很自然的，也是合理的。但是，员工带来的价值可能是不同的，这可能会激励员工在有更高奖金补偿的国家工作（虽然这并不意味着一切）。解决方案就是不要给在印度的员工提供美国的工资！应对这种挑战的方法包括：

1. 提供具有挑战性的工作内容；
2. 增加测试的职业发展机会，例如采用能力中心的模式；
3. 提供长期工作激励机制，例如股票激励机制；
4. 提供旅行机会。

有效跟踪的能力 当团队分散在各地时，跟踪其工作进展变得很困难。已经完全没有像走廊聊天这样的内部交流机会。因此，跟踪机制必须更健壮，较少依赖人。提高跟踪有效性的方法包括：

1. 制订通信交流计划，例如每周的电话会议；
2. 明确的任务分工；
3. 一致地理解和落实报告机制；
4. 尽可能提高跟踪功能的自动化。

对通信基础设施的依赖 由因特网和内联网组成的通信基础设施是地理分布团队的生命线。对于测试人员每天都要提取新版的“全时区”模型的测试团队来说尤其是这样。这种模型必须有高带宽和高可靠性的通信基础设施的支持。通信技术已经相当健壮，这方面应该不是大问题，重要的是要有替代方案，以防备万一由于某种原因导致链路中断。降低这种风险的一种方法是每个地点的备份和镜像网站。这样做成本可能很高。另一种方法是在不同地点建立小的“在线”团队，需要时可作为“离线”团队的代理。

时差 在分布式工作和组织团队的不同模型中，时差通常是一种优势，但是时差也可以带来一些挑战。例如，在全时区测试模型中，当印度的测试人员发现一个影响测试继续进行的缺陷时，不得不在开发人员当地时间的夜间给开发人员打电话（假设开发人员在美国）。如果找不到该开发人员，印度测试人员的那个工作日就完全损失了。此外，当开发人员解决该问题时，测试人员也应该可以随时提供帮助，否则又会损失12小时！因此，为了解决一个问题，时差吃掉了36个小时，而对于单场地的团队来说，在几分钟内就可以解决问题。有效利用时差的关键是建立有效的过程和主动的质量保证（以使尽可能降低出现这种使测试不能继续进行的缺陷出现的可能性），并使各地的员工进行值班，以使任何时间任何地点都可以有人联系。

14.5　测试服务公司

14.5.1　测试服务的业务需求

到目前为止所讨论的大部分测试功能都隐含地假设测试活动是公司的一个部分，与产品的生产、销售和支持一样。但是，今天常常把测试活动外包给专门从事测试并提供测试服务的外部公司。这种把测试作为服务的模式有一些业务上的原因。

1. 前面介绍过，测试正在进一步细分，越来越专门化。
2. 测试自动化工具的多样性和复杂性增加了对测试的挑战。专业化的测试服务公司可以有效地满足这种专门化的要求。

3. 测试作为一种过程正在更好地定义，使测试（至少一些测试阶段和测试类型）更适合外包。

4. 在理解软件领域方面有经验的产品开发公司不一定在建立和运行有效测试环境上也有经验（特别是当需要使用复杂自动化工具时）。

5. 外包公司具有地域优势。在讨论多场地团队时提到过，位于不同时区的测试团队可以提供更好的全时区覆盖。

6. 外包公司具有成本优势。一些地区会比另外一些地区更经济，因此可以降低总成本，这个理由适用于所有功能的外包（包括开发和测试）。

因此，测试作为一种服务在业界已经得到普遍接受，结合时区和成本优势，测试服务一般外包给分布在各地的团队。

14.5.2 测试作为一种服务与产品测试公司之间的差别

测试服务公司中的组织结构和相关的人员问题受一些基本因素的影响。

测试服务公司一般与产品开发人员保持一定距离 在典型的产品开发公司中（测试是其一个部门），开发和测试团队之间有很紧密的联系，因为员工经常从一个团队调到另一个团队。不仅如此，测试团队检查产品代码时也没有知识产权（IPR）问题。当测试作为一种服务外包给外部公司时，外包公司的成员与产品开发人员可能达不到产品公司内部人员之间的融洽程度。如果外包团队在不同地点工作时情况更是如此，建立融洽关系的机会更少。产品代码的知识产权对于远离产品开发公司的测试团队也成为一个问题。

测试服务公司可能总是难以实施各种类型的测试 大多数测试服务公司都提供像黑盒测试、领域测试和一些专门测试，例如性能测试或测试自动化，通常不把白盒测试作为外包的一种测试类型。白盒测试不仅由于前面提到的知识产权问题，而且由于开发和测试团队之间密切协作对后勤支持有很高的要求，需要随时同步共享配置管理系统，以及对代码的深入知识，都使得将白盒测试外包给测试服务公司变得很困难。

测试服务公司的工作岗位和角色比产品公司中的测试团队更单一 在典型的产品公司中有各种工种——开发、维护、测试、支持等。由于测试服务公司只关注测试，因此角色定义和岗位设置更单一。这可能是一把双刃剑，寻求工作经验多元化（覆盖软件生存周期的所有方面）的人觉得测试服务公司从长远看没有意思，另一方面，通常被错误地认为测试是“没有意思”的工作，测试服务公司能够招募到对测试充满激情的员工，会更容易为其提供宽广的职业发展道路。表14-2归纳了产品测试和测试作为一种服务之间的主要差别。

表14-2 产品测试与测试作为一种服务之间的差别

内 容	产品测试	测试作为一种服务
测试类别	必须完成所有类型的测试，因为整个产品的责任都属于公司	公司可以专注特定类型的测试，例如性能测试或国际化测试
查看代码	产品测试团队可能要更深入地研究产品代码，特别是白盒测试团队	通常，测试服务公司关注产品的外部功能，因此不需要共享配置管理系统的同步要求。此外，因为知识产权或协议方面的原因，对源代码开放限制到最低程度，或不需要开放
接触开发人员	一般没有问题，因为他们属于同一公司	可能很少

（续）

内　容	产品测试	测试作为一种服务
职业发展问题	由于公司有多种角色和工作部门，可能导致测试人员脱离公司内部其他部门（例如开发部门），因此提供专门的测试职业发展道路更困难	主干整个公司都完全专注测试工作，因此更容易向员工提供测试职业发展道路，可以更好地适应和把握对未来的预测
工具的多样性	整个公司可能（经过一段时间的积累）对测试工具作了标准化，因此通常没有必要在多个工具上采购和培训上投资	测试服务公司要为拥有不同自动化工具的多个客户工作，因此不得不在多个工具上对员工进行培训和投资
领域专门性	由于产品一般集中在特定领域，可以更好地培养并从内部的领域专家	主干测试服务公司要为不同领域内的不同客户工作，公司要么招聘具有不同领域知识的员工，要么培养员工具有多个领域的知识

14.5.3　测试服务公司的典型角色和责任

图14-6给出了测试服务公司的典型组织结构。测试服务公司由多个部门组成，每个部门负责一个主要客户。每个部门都有一个部门经理，他是客户与测试服务公司的唯一接触点。部门经理有以下作用：

1. 是客户和测试服务公司之间就任何问题进行交涉的唯一接触点，也是扩展点。

2. 建立与客户的密切关系，负责保证按照承诺交付当前的项目，并从客户那里得到新（重复）的业务。

3. 参与客户和测试服务公司之间的所有战略（如果必要，还包括战术的）沟通。

4. 在测试服务公司内部起客户代理的作用。

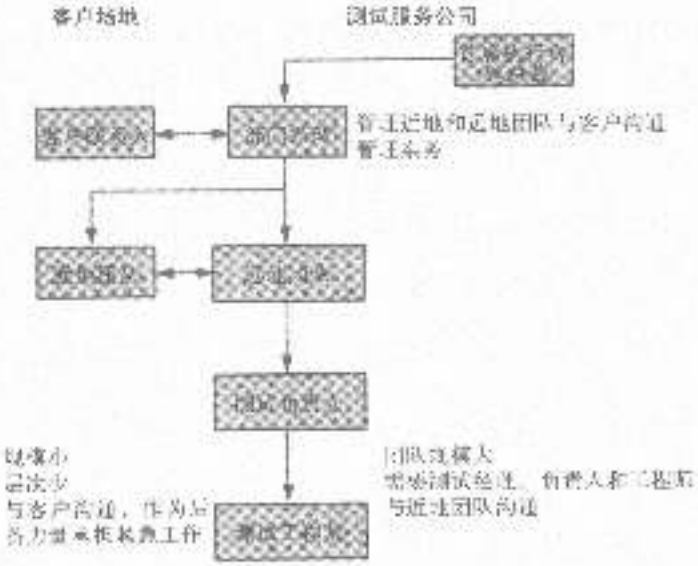


图14-6　测试服务公司的典型组织结构

部门经理可以常驻与客户场地很接近的地方，也可以在测试服务公司所在地。为了建立密切的关系，部门经理需要经常出差，与客户面对面地沟通。通常部门经理在客户公司高层

有唯一的接触点，以协调部门的多个项目。

由于测试服务公司与客户是不同的实体，一般不在同一个地点。（通常这些公司都在具有更有竞争力的成本结构的国家中。）

测试服务团队把其部门团队组织为近地团队和远地团队。近地团队通常很小，常驻客户场地或接近客户场地。近地团队进行以下工作：

1. 对于技术或紧急问题，与客户进行交涉的第一点接点。例如，如果需要快速研究系统以了解测试增量版本需要的内容，那么由于离客户很近，近地团队就可以完成这类工作。

2. 当出现不能出差、基础设施故障等紧急情况时，可作为代表远地团队的中转站。

3. 还可以密切客户的执行团队和测试服务团队之间的关系。

这种团队通常很小，没有必要建立经理－工程师结构。

远地团队常驻在测试服务公司所在地。远地团队的人数通常较多，并完成主要工作。远地团队的组织取决于测试项目的规模和复杂度。远地团队经理管理整个远地团队，与近地团队可以有对等的关系，也可以让近地团队向其报告，根据需要，远地团队可以进一步分组织层次，因为测试负责人和测试工程师与上一章讨论过的角色和责任相当接近。

近地团队和远地团队通常实行轮换制。例如，有些远地团队成员可以去做近地团队的成员。如果测试服务公司所在的国家与客户所在的国家不同，就有了出差机会——这也是保留人才的一种常见策略。此外，远地团队成员可能要定期去客户所在地进行近地团队不能独立完成的现场测试和类似的活动。

14.5.4 测试服务公司面临的挑战与问题

所有测试公司都面临一些共同的挑战。对于承接外包测试任务的测试服务公司，有些挑战更为突出，这主要是因为与开发团队有一定的距离。下面详细讨论这些挑战，以及应对这些挑战的方法。

局外人的效应与资源的评估

测试服务公司对开发公司来说是“局外人”，这意味着：

1. 他们不必像产品公司内部的测试团队那样多地接触到产品的内部或代码。

2. 他们不能接触到产品的历史，例如哪些模块以前更容易出现问题，因此可能难以得到策划并确定测试用例优先级所需的全部信息。

3. 他们不必像内部测试团队那样与开发团队建立密切的联系。

4. 内部开发和测试团队不一定需要有关所需的硬件和软件的资源信息，而测试服务公司则必须估计和策划硬件和软件资源。

这些挑战使测试服务公司估计进行性能测试所需的资源更加困难。对这些问题没有简单答案，但是服务公司肯定需要比产品公司更周密的策划和跟踪。

领域专门知识

产品公司中的测试团队可以在具体的领域积累领域专门知识。例如，ERP软件公司的测试团队可以积累有关ERP领域的专门知识。公司可以聘请几个领域专家（不一定具有计算机方面的专业知识），他们能够帮助测试团队。这些专家会发现加入产品公司很有意思，因为他们会发现从他们的领域到（显然）更有吸引力的IT领域，是很自然的过渡。例如，财务方面的专家可以很自然地进入IT产品公司。

但是，测试服务公司却可能要承接多个客户的项目，由于测试服务公司没有什么产品拥有权，因此争取领域专家加入测试服务公司会比较困难（但是测试服务公司的发展非常迅速，领域专家可以看到职业发展机会，因此这个弱点正在迅速消失。）不仅如此，领域的多样性使这个问题更加恶化。由于大多数测试服务公司都执行功能测试（黑盒测试、集成测试、确认测试等），这使领域专家要花很长时间改进测试质量。产品公司在这些资源上有更强的竞争力。

保密与客户隔离问题

为了生存，测试服务公司必须为多个客户工作。当公司为给定领域（例如医疗服务或财务服务）的客户工作时，不仅会积累一般的专业知识，还会积累特定领域的专业知识。公司不断积累领域专业知识，在向同一领域的其他公司提供测试服务时就有了竞争优势。类似地，当公司积累特定自动化工具的专门知识时，其优势可以显示在承接使用相同工具的其他公司的测试自动化项目。但是，由于IT与大多数业务的公司级战略有很强的连系，因此任何两个客户之间都必须画出清晰的界限。

出现这种主要挑战有两个因素。第一，测试服务公司拥有通用基础设施，因此将不同团队物理地隔离开来是很困难的。第二，员工会从一个项目（客户）转到另一个项目。当出现这种转移时，必须完全保证在原项目获得的与客户有关的知识不被带到另一个项目中。同时，测试服务公司为了持续改进，所学到的内容要提升抽象到合适的层次，同时又不能损害与客户有关信息的保密性。

前一节讨论过的组织结构可以解决以上问题。首先，每个客户都可以作为一个独立的实体对待，每个客户有一个与之对应的部门经理和专门的团队。在项目启动时，客户和测试服务公司要签订保密协议（NDA），这种保密协议提供一种不泄露保密信息的框架。在有些情况下，团队还进行了场地的物理分离，其次，特定客户的客户数据也要隔离和保密。这就引出了有关分配硬件和软件资源及成本的下一个问题。

分配硬件和软件资源及成本

当测试服务公司投标并为多个客户工作时，需要使用内部硬件和软件资源。有些资源可以直接确定，并划归特定的团队专用。有些资源需要多个项目复用。例如，像卫星链路这样的通信基础设施，像电子邮件服务器这样的通用基础设施，以及物理基础设施成本。

必须在不同的项目间划分成本，并进行项目成本核算。问题是很难确定每个项目应该分得多少资源。

维持“待命队员”

测试服务公司应该总有人员准备部署到突然来自客户的新项目中。此外，在正式投入工作之前，作一些初步研究或初步演示能力。这些也需要技术资源。有时客户可能想看看特定人员的简历，并希望从测试服务公司中挑选一些人员参加自己的测试项目。这些都要求测试服务公司有一些“待命队员”，即没有分配到任何项目组中，随时准备接手新项目，或促使客户增强信心。

质量资源（待命队员）是没有收入的，因此必须分摊到项目中。而且，待命人员通常会有失落感、不安全感，很容易产生怪癖。作为策划业务流程的一部分，要估计任何时候待命人员的数量，即质量深度。

14.6 测试公司的成功因素

不论是产品测试公司还是测试服务公司，都有一些共同的成功因素。这些因素包括：

沟通与团队协作 第1章和第13章已经讨论过，测试团队和开发团队应该有协作精神，不应该相互视为对手。但是，由于测试团队的工作就是找出开发团队工作中的缺陷，因此培养这种协作精神并不容易。对于产品公司来说，培养团队协作精神的一种方法是定期轮换工作。测试服务公司并不总能做到这一点，他们与开发团队有一定的隔离。团队之间应该有很多的沟通机会，有些沟通是在非工作环境和场景下进行的。对于地域分散的团队来说，沟通和团队协作问题更加突出，因为他们面对面交互的机会很少。

引入客户视角 测试团队应该发挥客户代理的作用。这要求逐步了解客户的视角。测试团队如何了解客户视角？一种办法是将领域专家或客户代表吸收到测试团队。另一种办法是测试人员和客户支持人员轮换。在产品公司中，客户支持人员是客户报告问题的接触点，他们通常熟悉客户所面临的问题、什么对客户的影响最大、客户可以接受什么。让客户支持人员参加设计测试用例或执行即兴测试（请参阅第10章）可以增加客户视角。另一种方法是测试团队，甚至开发团队与产品支持人员轮换，以便了解客户面临的第一手信息。

提供合适的工具和环境 为了提高工作的有效性，测试公司必须有合适的支持工具。两种最重要的工具是缺陷库和配置管理工具。缺陷库用于跟踪缺陷直到完成。前面已经讨论过，配置管理工具提供团队共享软件环境的机制，保证整个团队得到一致的产品源代码和目标代码，以及其他相关文件。（第15章还将详细介绍这些工具。）

提供定期技能升级 测试团队需要定期升级技能，使技能保持最新。要升级的技能包括：

1. 领域技能，升级到未来版本的新领域或新特性；
2. 技术技能，例如，新语言的使用；
3. 自动化技能，熟悉新的自动化工具；
4. 软技能，包括语言沟通、书面沟通、谈判能力，以及解决矛盾的能力，以更好地进行沟通。

重要的是在团队任务压度很紧张的情况下找到工作技能的时机。个人的职业发展计划需要安排时间进行技能升级，否则，测试团队的有效性会逐渐下降。

问题与练习

1. 当公司从单产品发展到多产品时，如何利用过去的经验？
2. 对于以下情况，开发和测试的哪类工作会有贡献？
   a. 使用了要求人员具有特殊技能的最新技术产品；
   b. 发布周期很短，版本频繁更新的产品；
   c. 同时有多个版本正在使用的产品；
   d. 在已经稳定的现有特性上渐进地增加新特性的产品；
   e. 公司的开发团队交付工作产品有明确的交接过程和方法的产品。
3. 最初成立一个测试小组，为银行的内部“客户”（应用程序开发人员）提供测试服务。不久，测试小组的能力得到发展，成为为其他银行和财务团体提供测试服务的独立公司。请列出值得这个新公司注意的组织结构和业务方面的挑战，以及他们应如何利用以前的经验。
4. 问题2中的哪种情况最适合地域分布的团队？

5. 某个公司在系统和应用程序领域有多个产品，请讨论成立独立的测试团队和公共测试团队各有什么优缺点。
6. 某个产品有几个月的发布周期，然后是十个月的维护期，什么时候可以建立地域分布的测试团队？在什么时候开展什么活动？请说明自己的理由。
7. 以下哪种测试活动可以外包给测试服务公司？为什么？
   a. 代码覆盖测试
   b. 性能测试
   c. 即兴测试
   d. 集成来自不同公司的多个产品
   e. 硬件和软件的集成
8. 上一章已经讨论了具体的人员问题，对于多团队情况，哪些人员问题会增加哪些新的挑战？

第五部分 测试管理与自动化

本书这一部分将讨论测试项目的策划和管理。第15章将讨论测试项目的公共项目管理问题，包括策划、风险管理、估计、跟踪和报告等。这一章还给出了涉及策划和执行测试项目各个方面的检查单和模板。第16章将讨论测试自动化工作的问题、挑战和一些解决方案。精心策划和组织的自动化是解决人员问题并提高团队效率的关键。本书最后一章——第17章将讨论与跟踪和测试项目改进有关的各种指标。这些度量使策划和自动化能够实现组织中的有效测试。

第15章 测试策划、管理、执行与报告

15.1 引言

本章将研究有关测试项目管理的一些问题。项目管理协会（PMI-2004）把项目正式定义为“创建某种特有的产品或服务的临时性工作”。这意味着每个项目都有一个明确的开始和一个明确的结束，而且产品或服务能够以某种方式与类似的产品或服务区分开来。

测试被集成到创建给定产品或服务的工作中，每个测试阶段和测试类型都有不同的性质，对每个版本的测试内容也可能不同。因此，测试完全符合这种关于项目的定义。

由于测试本身就可以看作是一种项目，因此也需要策划、执行、跟踪和定期报告。下一节将讨论测试策划问题，然后讨论推动测试项目的过程。接下来将研究测试的执行和在测试项目期间需要产生的各种类型的报告。本章最后将介绍有关测试管理和执行的最佳实践。

15.2 测试策划

15.2.1 准备测试计划

与任何项目一样，测试也要由计划推动，测试计划是整个测试项目的执行、跟踪和报告的锚点，它包括以下内容：

> 没有策划就是在策划失败。

1. 什么需要测试——测试的范围，需要清晰地确定要测试什么，不测试什么。
2. 测试应该如何实施——将测试分解为小的可管理的任务，并确定完成这些任务要采取的措施。
3. 测试需要什么资源——计算机和人力资源。
4. 测试活动展开的时间线。
5. 所有以上计划所面临的风险，以及合适的应对和处置计划。

15.2.2 范围管理：决定要测试和不测试的特性

前面已经介绍过，有各种测试团队完成不同测试阶段的测试工作。准备一份统一的测试计划可以包含所有阶段和所有团队的工作，也可以为每个阶段或每类测试准备一份计划。例如，需要有单元测试、集成测试、性能测试、确认测试等的计划。这些内容可以是一份统一计划的一部分，也可以包含在多份计划中。对于有多份测试计划的情况，应该有一份计划包含所有计划的公共活动，这种计划叫做主测试计划。

范围需要描述项目的范围。对于测试项目，范围管理包括：

1. 理解哪些内容构成产品的发布版本；
2. 将发布版本分解为特性；
3. 确定特性测试的优先级；

4. 确定哪些特性要测试，哪些不测试；

5. 收集数据，准备估计测试资源。

最好从确定项目的最终目标或产品发布版本视角开始，获得整个产品的完整图像，以确定测试的范围和优先级。通常，在发布版本的策划阶段要确定构成发布版本的特性。例如，仓库控制系统的某个特定发布版本可能引入新特性，以自动集成供应链管理，并向用户提供各种成本核算选项。测试团队应该在策划周期的初期参与策划并了解特性，了解特性并从使用视角进行理解，使测试团队能够确定测试的优先级。

选择并确定待测特性的优先级有以下考虑因素：

新特性和对发布版本至关重要的特性 发布版本的新特性确立了客户的预期，必须能够正常运行。这些新特性会引入新的程序代码，因此更容易引入和发生缺陷。不仅如此，这些新特性很可能对于开发和测试团队来说都有个熟悉的过程。因此，把这些特性放在优先测试的位置上是合理的，这可以保证这些关键特性有足够的策划和学习时间进行测试，不会出现测试不充分的现象。为此，产品营销团队和一些经过选择的客户应参与被测特性的选择。

失效会造成严重后果的特性 不管特性是否新创造，其失效有可能产生严重后果，或对商业带来负面影响的特性，都应该重点测试。例如，数据库的恢复机制品应该列为重点测试的特性。

有可能测试起来很复杂的特性 测试团队的尽早参与有助于确定很难测试的特性。这有助于尽早开展对这些特性的测试工作，并留出足够的资源。

对以前容易出现错误的特性的扩展特性 第8章已经介绍过，有些区域的代码很容易出现缺陷，需要彻底测试，以防老的缺陷再次蔓延。这些容易出现缺陷的特性应该在更稳定的特性之前进行测试。

产品不只是这些特性的混合，它们要根据多种环境因素和执行条件，以各种组合协同运行。测试计划应该清晰地确定要测试的这些组合。

由于资源和时间的限制，很可能不能穷尽地测试所有的组合。在策划时，测试经理还应该精心确定不测试的特性或特性组合。作出这种决定时要综合考虑时间和资源需求，同时又不会使严重缺陷暴露给客户。因此，测试计划应该就不测试特定特性组合的理由，以及不测试所面临的风险作出明确的分析。

15.2.3 确定测试方法和策略

一旦有了带优先级的被测特性表，下一步就是深入分析需要测试的细节，估计规模、工作量和进度，包括确定：

1. 测试各个功能要使用什么测试类型？

2. 测试特性需要什么配置或场景？

3. 采用什么样的集成测试以保证这些特性能够协调运行？

4. 需要怎样的本地化确认？

5. 需要作什么“非功能”测试？

本书前面各章已经讨论了各种测试类型，每种类型测试的适用性和作用都有一定的条件。测试计划中测试方法和策略部分需要确定合适的测试类型，以有效地测试给定特性或特性组合。

测试策略和方法的确定应该导致确定每个特性或特性组合的合适的测试类型。此外还应该确定度量测试是否成功的客观准则，下一节将详细讨论。

15.2.4　确定测试准则

前面已经讨论过（特别是有关系统和确认测试的几章），各个测试阶段都必须有明确的进入和退出准则。各种特性和特性组合的测试策略确定应该如何对其进行测试。在理性情况下，必须尽早进行测试，以最大限度地降低开发拖延后给执行测试带来的巨大压力（请参阅以下“风险管理”一节）。但是，过早进行测试是没有用的。测试的进入准则描述每个测试阶段或类型的开始准则。整个测试活动的开始也可以有进入准则。完成退出准则描述什么时候测试周期或测试活动可以结束。如果没有客观的退出准则，测试就有可能继续下去，超过最佳回报点。

测试周期和测试活动不是一次就能完成的、孤立的持续活动。可能会在某个时间点上挂起，因为不能再继续进行。如果能够继续进行，就必须接着测试。挂起准则描述什么时候被挂起的测试可以接着执行。一些典型的挂起准则包括：

1. 遇到超过一定数量的缺陷，使测试活动频繁中断；

2. 遇到使测试不能继续进行的严重问题（例如，如果数据库不能启动，像查询、数据操作等这样的进一步测试根本无法执行）；

3. 开发人员提供了新版本，希望用它替代正在测试的版本（由于修改了某些严重缺陷）。

当这些问题都解决后就可以继续测试。

15.2.5　确定责任、人员和培训计划

范围管理确定要测试什么，测试策略确定如何测试，测试计划的下一个问题是谁参与测试，确定责任、人员和培训需求就是要回答这个问题。

测试项目需要不同的人起不同的作用。前两章已经讨论过，测试项目的角色包括测试工程师、测试负责人和测试经理。在被测模块和测试类型上也有角色定义。这些角色要彼此具有互补性，不同的角色定义应该：

1. 保证给定任务都有明确的职责定义，使每个人都知道自己要做什么；

2. 明确列出各类人员在各种工作中的职责，使每个人都知道自己的工作在整个项目中的作用；

3. 彼此要有互补性，不能相互影响；

4. 彼此补充，分配所有的任务。

角色定义不能只限于技术角色，还应该列出管理和报告责任，包括频度、格式和状态报告的接受者，以及其他项目跟踪机制。此外，在策划阶段还应该以查询服务等级约定的形式描述相应的责任。

要根据所涉及的工作量和发布前还剩下的时间分配人员。为了保证恰当的任务都被完成，要根据工作量、时间和重要性确定特性和任务的优先级。

为任务分配人员时，要尽可能兼顾任务需求和完成该任务所需的经验和技能，并不总能在需求和可利用的技能之间找到完美的契合，这样就需要制定合适的培训计划。重要的是要事先展开培训程序，因为在项目进度的压力下，培训往往排不到日程上。

15.2.6　确定资源需求

作为测试项目策划的一部分，项目经理（或测试经理）要对所需的各种硬件和软件资源

进行估计。估计时需要考虑以下因素：

1. 运行被测产品所需的机器配置（RAM，处理器、硬盘等）。
2. 如果有测试自动化工具的话，工具所需的开销。
3. 诸如编译器、测试数据生成器、配置管理工具这类的支持工具。
4. 必须提供的支持软件（例如操作系统）的不同配置。
5. 执行机器密集型测试，如负载测试和性能测试所必须满足的特殊需求。
6. 所有软件的合适数量的使用许可证。

除了以上因素，还有一些需要满足的隐含环境需求，包括机房空间，支持工作（例如人力资源）等。

对这些资源估计不足会严重影响测试工作的进展，会延迟产品发布时间，还会影响测试团队的积极性。但是，如果在估计这些资源上过于保守和"保险"，很容易产生不必要的开销，对这些资源的恰当估计需要不同小组，包括产品开发团队、测试团队和系统管理团队及高层管理之间的团队协作。

15.2.7 确定测试的可交付产品

测试计划还要确定测试周期或活动产生的可交付产品。可交付产品包括以下内容，所有这些产品都要经过合适人员的评审和批准：

1. 测试计划本身（主测试计划、项目的各种其他测试计划）。
2. 测试用例设计规格说明。
3. 测试用例，包括在计划中描述了的自动化。
4. 运行测试用例产生的日志记录。
5. 测试总结报告。

下一节将要看到，缺陷库给出产品生命周期内报告的缺陷状态。测试周期内的一部分可交付产品要保证缺陷库始终保持正确，包括在缺陷库中输入的新缺陷和验证缺陷修改后的状态更新。本章稍后还要介绍这些可交付产品的内容。

15.2.8 测试任务：规模与工作量估计

以上确定的范围大致确定了需要测试的内容，这些内容要在估计步骤中量化。估计大致分三个阶段：

1. 规模估计
2. 工作量估计
3. 进度估计

本节讨论规模和工作量估计，下一节讨论进度估计。

规模估计量化需要完成的实际测试量。测试项目的规模估计有多个影响因素。

被测产品的规模 这显然决定需要完成的测试量。一般来说，产品越大，测试的规模越大。被测产品的一些度量包括：

1. 代码行数（LOC）是一种有些争议的度量，因为代码行数依赖语言、程序设计风格、程序设计的紧凑性等。不仅如此，代码行数只表示编码阶段的规模估计，不能表示其他阶段，例如需求、设计等的估计。尽管有这些局限性，代码行数仍然是一种估计规模的常见度量。
2. 功能点（FP）是估计应用程序规模的流行方法。功能点表示应用程序的规模，与程序

设计语言无关，应用程序的特性（又叫做功能）按输入、输出、接口、外部数据文件和查询分类。这些功能的复杂性逐次升高，因此权重也逐次升高。功能的加权平均（每种类型的功能数乘以该功能类型的权重）得到规模或复杂度的初始估计。此外，估计规模的功能点方法论还提供了14个环境因素，例如分布式处理、事务处理率等。

这种应用程序的规模或复杂度估计方法论是综合性的，考虑了各种现实因素。这种方法的主要挑战是需要正式的培训，并且不易使用。不仅如此，这种方法还不直接适合系统软件类型的项目。

3. 表示应用程序规模的一种比较简单的办法是屏幕数、报表数或事务数。这些数据可以进一步细分为“简单”、“中等”或“复杂”。这种分类可以根据直观要素，例如屏幕上的字段数、要完成的确认数等确定。

所需的自动化范围 涉及自动化时，测试需要增加的工作规模。因为自动化需要首先进行基本的测试用例设计（运用条件覆盖、边界值分析、等价类划分等技术确定输入数据和预期结果），然后通过测试自动化工具的程序设计语言将其脚本化。

要测试的平台和互操作环境的数量 如果要在多个不同的环境或配置下测试特定产品，那么测试任务的规模就会增加。事实上，随着平台数量或跨不同环境的接触点的增加，测试量几乎呈指数增长。

以上规模估计考虑的都是“常规”测试用例开发；回归测试（请参阅第8章）的规模估计需要考虑产品变更和其他类似因素。

为了更好地进行规模估计，要完成的工作可分解为可做工作分解结构（WBS）单元的可管理的较小部分。对于测试项目，工作分解结构单元一般是给定模块的测试用例等。这种分解将问题域或产品划分为较为简单的部分，可以降低不确定性，减少未知因素。

规模估计可以采用以下任何形式表达：

1. 测试用例数
2. 测试场景数
3. 要测试的配置数

规模估计是对测试要覆盖的实际工作面的估计，是工作量估计的主要输入。工作量估计很重要，因为工作量对成本的影响往往比规模更直接。影响工作量估计的其他因素还有：

生产率数据 生产率指各种测试活动的完成速度，其基础是公司内部可以得到的历史数据。生产率数据可以进一步分解为每天（或某个单位时间）可以开发的测试用例数量、每天可以运行的测试用例数量、每天可以测试的文档页数等。有了这些细粒度的生产率数据就可以更好地进行策划，并提高估计的可信水平和准确性。

重用机会 如果在设计测试体系结构时考虑了重用问题，那么覆盖给定测试规模所需的工作量可以压缩。例如，如果所设计的测试用例能够重用以前的测试用例，那么测试开发的工作量就会下降。

过程的健壮性 重用是公司过程成熟性的一个具体例子。拥有定义良好的过程从长远看会减少完成任何活动所需的工作量。例如，如果某个公司的过程成熟性较高，就会拥有：

1. 针对编写测试规格说明、测试脚本等的形成规范文档的标准；
2. 针对完成评审、审计等活动的经过实践检验是很有效的过程；
3. 一致的人员培训方法；
4. 度量遵循过程有效性的客观方法。

所有这些都可以减少重复工作，从而降低所需的工作量。

通过把各个工作分解结构单元分成“可重用”、“修改”和“新开发”三大类来估计规模，再进行工作量估计。例如，如果有些测试用例可以重用现有的测试用例，那么开发这些测试用例所需的工作量就接近0。另一方面，如果需要从头开发测试用例，那么工作量估计就假设为测试用例规模除以生产率就是合理的。

工作量估计以人天、人月或人年的形式给出，然后再把工作量估计转换为进度估计，下一节将讨论进度估计问题。

15.2.9 活动分解与进度估计

活动分解和进度估计将所需的工作量转换为具体的时间帧，这种转换包含以下步骤：

1. 确定活动之间的外部和内部依赖关系；
2. 根据预期所需的工期和依赖关系确定活动的完成顺序；
3. 根据以上两个因素确定每个工作分解结构活动所需的时间；
4. 监视项目实际时间和工作量的耗费情况；
5. 必要时重新协调进度和资源。

在进行工作量估计时，已经确定了每个工作分解结构单元所需的工作量，并考虑了重用因素。工作量采用人月的形式表示，如果给定工作分解结构单元所需的工作量估计是40人月，则不能用“人”换“月”，即不能无限制地增加人手，并预期成比例地缩短所需时间。正如[BROO 74]所讨论过的，在一个已经被延误了的项目中增加人手肯定会造成进一步拖延！因为在项目中增加新人会增加沟通负担，新成员需要时间与团队已有成员进行磨合。不仅如此，这些工作分解结构单元不能以随机顺序执行，因为在这些活动之间有依赖关系。这些依赖关系可以是外部的，也可能是内部的。活动的外部依赖关系不受执行该活动的经理和员工的控制。一些常见的外部依赖关系包括：

1. 开发人员是否按时提交产品；
2. 招聘；
3. 培训；
4. 培训所需的硬件和软件资源的获取；
5. 是否有供测试的经过转换的消息文件。

内部依赖关系完全受执行该活动的经理和员工的控制。一些常见的内部依赖关系包括：

1. 测试规格说明的完成；
2. 测试用例的编码和脚本化；
3. 测试的执行。

测试活动也会面临并行性的约束。这进一步限制了可以同时开展的活动。例如，由于条件冲突（例如需要不同版本的被测组件）或需要在多个测试之间复用高端设备，有些测试用例不能执行。

根据依赖关系和可能的并行性，可以顺序地排列测试活动，这有助于在最短的时间内完成这些活动，同时又考虑到所有依赖关系。这种进度可以采用如图15-1所示的甘特图表示。

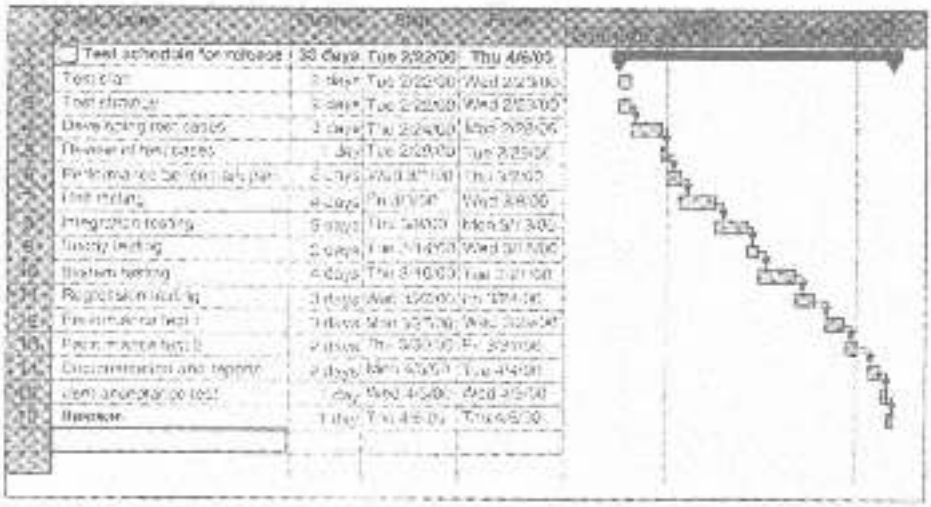


图15-1　甘特图

15.2.10　沟通管理

沟通管理包括制订并遵循沟通规程，保证每个人都在合适的细节层次上保持同步。由于同步与测试执行和测试项目的进展密切相关，因此，我们将在15.3节讨论测试周期内的各种类型的报告时详细介绍。

15.2.11　风险管理

与所有项目一样，测试项目也会面临风险。风险是会潜在地影响项目结果的事件。这些事件通常超出项目经理的控制。如图15-2所示，风险管理需要：

1. 确定可能的风险；
2. 对风险进行量化；
3. 策划如何缓解风险；
4. 风险真的出现时的应对措施。

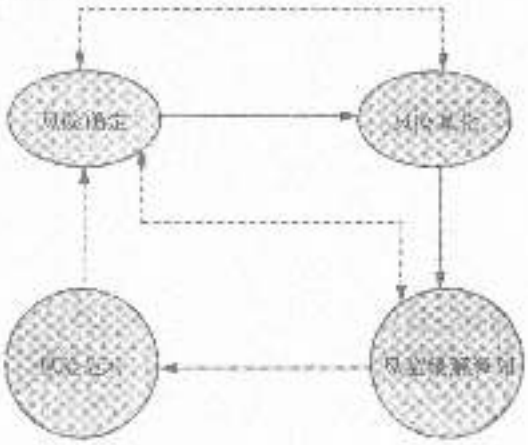


图15-2　风险管理的要素

随着一些风险的确定和解决，其他风险又可能出现。因此，随着风险的不断出现，风险管理实际上是一种循环，重复地执行以上给出的四个步骤。

风险确定要确定项目可能产生的风险。虽然项目可能有很多的潜在风险，但是风险确定应该关注更有可能发生的风险。以下是确定测试风险的常见方法：

1. **使用检查单** 经过一定时间的积累，公司在测试中会有一些新发现，归纳成检查单。例如，如果在安装测试中发现安装的特定步骤经常出现问题，那么在检查单中可明确列出需要检查该问题。如果把检查单用于风险确定，检查单本身也会有会过时的很大风险，因此，确定的不是真正的风险。

2. **利用公司的历史和指标** 如果公司收集并分析各种指标（请参阅第17章），那么这些信息对确定项目可能出现的风险很有价值。例如，过去的测试工作量估计偏差可以说明需要为策划出现问题的可能性有多大。

3. **整个行业的非正式网络** 整个行业的非正式网络有助于确定其他公司已经遇到过的风险。

风险量化是以数字的形式描述风险。风险量化有两个要素，一是风险发生的可能性，二是风险的发生带来的影响。例如，低优先级的缺陷可能有很高的可能性，但是影响很小。但是影响程序正常运行的缺陷的可能性很小（但愿如此！），但是影响很大（对于客户和提供商公司来说都是如此）。为了用一个数字表示这两个要素，通常采用风险指数。风险指数定义为风险可能性和风险影响的乘积，为了便于比较，风险影响采用金额的方式表示（例如以美元为单位）。

风险缓解系列确定如果风险出现的应对风险事件的替代策略。例如，缓解风险的一些替代策略是让多个人共享知识和建立公司级的过程和标准。为了更好地准备应对风险带来的影响，最好能有多种缓解策略。

如果及时系统化地完成以上三个步骤，即使风险真的出现，公司也会在应对风险方面处于更有利的位置。如果没有对这些步骤给予足够重视，项目组就会发现自己处于很大的风险压力下。在这种情况下，所作出的决策可能不是最优的，或不是审慎的，因为决策是在很大压力下作出的。

以下是一些测试项目的常见风险和特性。

不明确的需求 测试的成功在很大程度上取决于对被测产品的正确预期行为的了解。如果产品要满足的需求没有在文档中明确，对测试结果的解释就存在模糊性。这可能导致报告错误的缺陷，或遗漏真正的缺陷。这反过来又会导致开发和测试团队之间不必要的沟通浪费，从而损失时间。降低这种风险的一种方法就是保证测试团队预先参与需求阶段。

进度依赖性 测试团队的进度在很大程度上取决于开发团队的进度。因此，测试团队很难列出在什么时间需要什么资源。如果测试团队被多个产品共用，或在测试服务公司中（请参阅第14章），那么这种风险的影响会更大。应对这种风险的一种可能的策略是确定测试资源的后备项目。这种后备项目可以使用额外的资源加速活动的执行，但是，如果没有资源也不会有太大的影响，例如，加速测试自动化的工作就可作为后备项目。

测试时间不足 本书通篇都强调了不同类型的测试和不同的测试阶段。尽管有些类型的测试，例如白盒测试，可以在生命周期的早期进行，但是大部分测试一般还是要在接近产品发布的时候实施。例如，系统测试和性能测试只能在整个产品完成后，并接近发布的时候进行。通常这些测试很耗费测试团队的资源，而且所发现的缺陷也是开发人员比较难改的。在讨论性能测试时已经提到过，修改这类缺陷可能需要变更体系结构和设计。作这样的变更成本很

高，甚至不可能。开发人员修改了这样的缺陷后，测试团队完成测试的时间更少，面临的压力更大。采用V字模型至少把各种类型测试的测试设计部分提前到项目的较早阶段，有助于更好地预测各个层次上的测试失败风险，这反过来又可以降低最后时刻的压力。如果计算合适，“发布产品所需时间”（参见第17章）这个指标可有助于更好地策划所需的时间。

“影响测试继续进行”的缺陷 测试团队报告问题后，开发团队必须进行修改。有些影响测试继续进行的缺陷可能使测试团队在开发团队修改之前不能继续测试。遇到这类缺陷会对测试团队带来双重影响：首先，测试团队不能继续测试，因此造成空闲。其次，当缺陷修改后测试团队重新开始测试时，他们已损失了宝贵的时间，并面临最终期限迫近的巨大压力。这种影响测试继续进行的缺陷的风险会对测试团队的进度安排和资源利用带来很大挑战。这种风险的缓解措施与关于对开发进度依赖的风险缓解措施相同。

测试人员的技能和测试积极性 第13章已经介绍过，聘用和激励测试人员是很大的挑战。测试人员的聘用、保留和技能的不断提高对于公司是至关重要的，尤其是当员工一般都更喜欢从事开发工作时。

不能获得测试自动化工具 手工测试很容易出错，且占用大量人力。第16章讨论的测试自动化可以缓解一部分问题。但是，测试自动化工具很昂贵，公司可能面临买不起测试自动化工具的风险。这种风险反过来又导致测试的低效和更多矛盾。这类公司减低这种风险可以采用的一种方法是自行开发工具，但是这种方法会引入甚至出现更大的风险，因为工具的开发或文档的编写质量不高。

这些风险不仅有单独出现的危险，甚至还有可能多种风险接连出现的危险。遗憾的是，这些风险往往确实接连出现！测试小组依据开发进度制订自己的进度，一旦开发进度出现变更，测试团队的资源就会出现空闲时间，压力就会积累，测试进度也需要变更，于是恶性循环开始出现。重要的是要尽早或在风险对测试团队产生严重影响之前，确定这些风险，因此，我们需要确定每种风险的征兆。需要在整个项目期间密切跟踪这些征兆及其影响。

表15-1给出了典型的风险、风险征兆、影响和缓解应对计划。

表15-1　典型的风险、征兆、影响和缓解计划

风　险	征　兆	影　响	缓解应对计划
开发延迟	• 各个层次实施的工作进度经常出现延迟	• 测试使用开始时间更少 • 推迟产品发布时间	• 测试团队不断参加开发计划的评审 • 定期、及时沟通 • 采用V字模型（参阅第2章）调整测试活动
出现影响测试继续进行的缺陷	• 测试工作经常因性能缺陷而暂停	• 浪费测试资源 • 当缺陷修改后重新测试恢复进行时，给测试团队带来压力 • 有可能进度推迟	• 制订产品能够接受测试的开发退出准则 • 在等待时实施测试团队完成其他工作
需求不清楚	• 产品通过所有内部测试后客户发现缺陷	• 从需求缺陷开始进行返工 • 因为产品不能满足需求造成客户不满	• 用户与产品营销团队尽早参与项目开发，以更好理解需求 • 定义明确的确认准则 • 需求的严格控制程序

（续）

风　险	行　表	影　响	缓解应对计划
测试没有足够时间	• 测试工作量增加 • 花在测试上的时间在整个产品生命周期中所占比例很小	• 质量低劣，被客户发现 • 测试团队内部出现矛盾	• 把测试活动分散到整个产品生命周期 • 测试活动自动化 • 尽早使所有相关各方对时间表达成共识
测试工具保守	• 报告无关紧要的缺陷 • 测试团队成为产品发布的瓶颈	• 测试资源没有产生预期的效果	• 制订客观的测试准入准则
缺少高素质的测试人员	• 不断有人带着成见来测试部门（例如开发部门） • 与其他团队相比，开发团队高的不平衡	• 测试质量差，有很多缺陷漏网，被客户发现 • 使测试团队的信誉受到损害	• 定期培训提高技能 • 展现测试的职业发展道路并决定角色模型 • 在工资、测试和支持团队之间实行人员轮换制
缺少自动化工具	• 手工测试耗时太长	• 手工测试浪费人力 • 造成测试工程师不满	• 展示使用自动化测试工具获得成功的实例

15.3 测试管理

前一节已经把测试本身作为一种项目对待，并讨论了测试中的一些典型项目管理问题。本节将讨论策划测试项目应该考虑的一些问题。这些策划工作是对所有测试项目都有全面影响的主动手段。

15.3.1 标准的选择

在任何公司中标准都是策划的一个主要组成部分。标准有两类，一是外部标准，二是内部标准。外部标准是产品应该满足的标准，是外部都知道的，通常是由外部团体规定的。从测试的角度看，这些标准包括由外部团体提供的标准测试和客户提供的确认测试。符合外部标准通常是外部团体强制要求的。

内部标准是测试公司制订的标准，用于提高工作的一致性和可预测性。内部标准规范公司内部的工作过程和方法。这些内部标准包括：

1. 测试工作产品的命名和保管规定；
2. 文档编写标准；
3. 测试编程标准；
4. 测试报告编写标准。

测试工作产品的命名和保管规定　每种测试工作产品（测试规格说明、测试用例、测试结果等）都应该恰当地命名，并有一定的含义。这种命名规则应保证：

1. 能够很容易地确定一组测试用例所对应的产品功能；
2. 能够反向，以确定产品功能对应的测试用例。

举一个说明命名规则使用的例子，请考虑由模块M01、M02和M03组成的某个产品P。测试包可以命名为PM01nnnn.<文件类型>，其中的nnnn可以是运行序列号或任何其他字符串。对于一个给定的测试，可能需要不同的文件。例如，给定测试可能要使用测试脚本（描述要执行的具体操作的细节）、键盘输入记录文件、预期结果文件。此外，还需要其他支持文件（例如数据库的SQL脚本）。所有这些相关文件都有相同的文件名（例如PM01nnnn）和不同的文件类型（例如.sh、.SQL、.KEY、.OUT）。通过这种命名规则，人们可以找到：

- 与特定测试有关的所有文件（例如查找所有文件名为PM01nnnn的文件）；
- 与给定模块有关的所有测试（例如以PM01开头的文件对应模块M01的测试）。

这样，当对应模块M01的功能变更后，很容易找到这些为须修改或删除的测试。

通过恰当的命名规则实现的测试用例和产品功能之间的双向映射，使得当产品的功能发生变更时，能够很容易地找到需要修改并运行的对应的测试用例。

除了文件命名规则，标准还可能规定测试的目录结构规则。这种目录结构可以把逻辑相关的测试用例组织在一起（以及相关的产品功能）。这些目录结构要映射到配置管理库中（本章稍后要讨论）。

文档编写标准 有关文档编写和编码标准的大部分讨论针对的都是自动化测试。对于手工测试，文档需要规范在与测试人员技能水平一致的细节层次上对用户和系统响应的描述。

命名和目录标准描述测试实体如何对外表示。而文档标准描述在测试脚本本身内部如何获取关于测试的信息。测试脚本的内部信息与程序代码的内部文档类似，应当包括：

1. 文件开头部分的适当概述，描述测试的作用。

2. 分散在整个文件中的充分的行间注释，解释测试脚本每个部分的作用。对于测试脚本中很难理解的部分或有多层迭代和循环的部分，行间注释尤其重要。

3. 一直到当前的历史信息，描述对该测试文件所作的所有修改。

没有这样的详细文档，维护测试脚本的员工就只能依靠实际代码或脚本，猜测该测试的设计意图，以及对该测试脚本都作了哪些修改。这可能造成对测试的理解错误。不仅如此，还有可能造成项目组对最初编写测试脚本的员工不恰当的责骂。

测试编码标准 测试编码标准又深入了一层，用于规范如何编写测试用例本身。测试编码标准应该：

1. 在测试用例应该完成的初始化和清理中，强制采用恰当的类型，以使该测试用例的执行结果与其他测试用例的执行无关。

2. 规范脚本中的变量命名方法，以保证读者能够一致地理解变量的用途。（例如，不应该使用类似i、j这样的一般名称，名称应该有含义，例如network_init_flag）；

3. 鼓励测试工作产品的重用（例如，所有的测试用例都首先调用初始化模块init_env，而不是使用自己的初始化过程）；

4. 对操作系统、硬件等外部实体提供标准的接口。例如，如果测试用例要产生多个操作系统进程，不是让每个测试用例都直接产生进程，而是制订编码标准，规定测试用例都调用一个标准函数，比方说create_os_process。通过分别与外部接口隔离，使测试用例能够合理地不受底层变更的影响。

测试报告标准 由于测试与产品质量密切相关，因此，所有有关各方都必须一致、及时地

看到测试的进展。测试报告标准就是要解决这个问题。测试报告标准在报告的详细程度、格式和内容、报告的阅读对象等方面提供指南。本章稍后还要进一步讨论测试报告标准。

内部标准使测试公司具有一种竞争力手段，为避免出现返工返工和事后提供了最重要的保证。内部标准有助于新测试工程师的快速成长。当在全公司范围内都执行这类一致的过程和标准后，可以提高可预测性，提高大家对最终产品质量的信心。此外，任何异常都会及时暴露出来。

15.3.2 测试基础设施管理

测试需要预先策划建设的基础设施。这种基础设施由三种基本要素构成：

1. 测试用例数据库（TCDB）
2. 缺陷库
3. 配置管理库和工具

测试用例数据库保存有关公司内测试用例的相关信息。表15-2给出了这种测试用例数据库的一些实体和每个实体的属性。

表15-2 测试用例数据库的内容

实 体	用 途	属 性
测试用例	记录有关测试用例的所有"静态"信息	• 测试用例标识 • 测试用例名称（文件名） • 测试用例细目表 • 测试用例的关键文件
测试用例与产品特性交叉引用	每个测试用例和对应的产品特性之间进行映射，以确定给定特性对应的测试用例	• 测试用例标识 • 特性标识
测试用例运行历史	记录测试用例执行的时间和执行结果，给出回归测试的测试用例输入选择（请参阅第3章）	• 测试用例标识 • 运行日期 • 所用时间 • 运行状态（成功/失败）
测试用例与缺陷交叉引用	维护记录发现产品特定缺陷的测试用例，给出回归测试的测试用例输入选择	• 测试用例标识 • 缺陷引用号（指向缺陷库中的一条记录）

缺陷库保存所报告产品缺陷的所有细节。表15-3给出了缺陷库包含的一些信息。

表15-3 缺陷库中的信息

实 体	用 途	属 性
缺陷实体	记录有关缺陷的所有"静态"信息	• 缺陷标识 • 缺陷优先级/严重等级 • 缺陷描述 • 被影响的产品 • 相关软硬件信息 • 环境信息（例如操作系统版本） • 发现问题的客户（也可以是内部测试团队） • 缺陷报告的日期和时间
缺陷测试细节	记录给定缺陷对应的测试用例的详细信息。与测试用例数据库交叉引用	• 缺陷标识 • 测试用例标识

（续）

实　体	用　途	属　性
修改组件	记录给定缺陷对应的修改组件信息，与配置管理库相交叉引用	• 缺陷标识 • 修改组件（所修改的文件、被改的文件信息）
沟通	记录有关相关各方就该缺陷沟通的详细信息，包括测试团队与开发团队、客户等之间的沟通，以及沟通的有效性	• 测试用例标识 • 缺陷引用号 • 沟通细节

缺陷库是一种重要的沟通载体，影响着软件公司的内部工作流程。缺陷库还提供将出第17章将要讨论的多个指标的基本数据。具体地说，就是大部分测试缺陷类的指标和开发缺陷类指标都是从缺陷库导出的。

软件产品公司所需的（测试团队也需要）另外一种基础设施是软件配置管理（SCM）库。软件配置管理库又叫做配置管理库，用于跟踪构成软件产品的所有文件和实体的变更控制和版本控制。一种特定的文件和实体就是测试文件。变更控制要保证：

1. 对测试文件的修改要在受控条件下进行，并只能在经过批准后进行。
2. 测试工程师实行的变更不会意外丢失或被其他变更覆盖。
3. 每次变更要产生一个能够在任何时点上重新产生的唯一的文件版本。
4. 在任何时点上，任何人都只能访问最新版本的测试文件（特殊情况除外）。

版本控制保证与产品给定发布版本关联的测试脚本和产品文件建立了基线。基线就是对一套相关的文件版本建立了一种快照，并对这套文件分配唯一的标识。将来，当有人需要重新生成给定发布版本的环境时，这种标识能够保证可以重新生成。

测试用例数据库、缺陷库和软件配置管理库应该相互匹配，并以集成的方式协同运行，如图15-3所示。例如，缺陷库将缺陷、修改和测试用例关联起来，保存这些内容的文件放在软件配置管理库中，被修改测试文件的元数据又放在测试用例数据库中。这样，从给定缺陷开始，可以（通过测试用例数据库）跟踪该缺陷对应的所有测试用例，并通过软件配置管理库找出对应的测试用例文件和源文件。

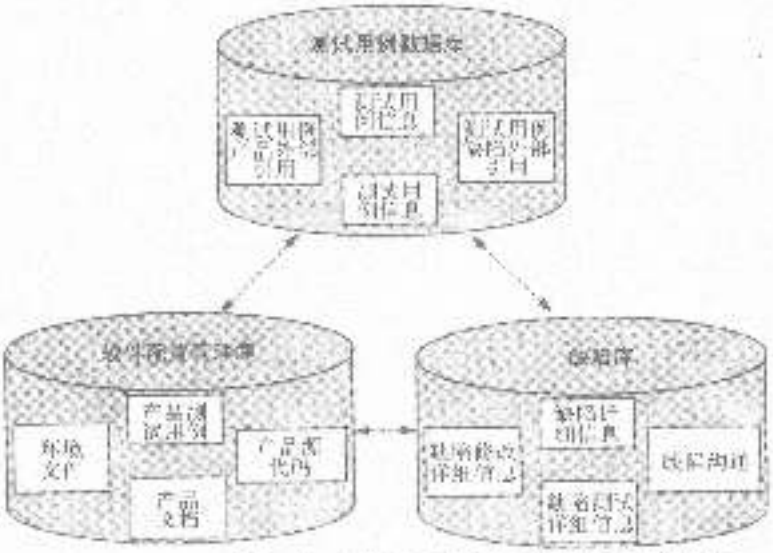


图15-3　软件配置管理库、缺陷库和测试用例数据库之间的关系

类似地，为了确定回归测试应该运行那些测试用例，

1. 可以通过缺陷库找出最近修改了的缺陷和对应的测试用例，从测试用例数据库中提取这些测试用例，组成回归测试测试用例。

2. 可以从软件配置管理库中找出自上次回归测试以来已经变更了的文件清单，并通过测试用例数据库跟踪到对应的测试文件。

3. 通过测试用例数据库可以得到最近没有运行的测试用例集，这些测试用例有可能用于按一定间隔时间进行的定期测试。

15.3.3 测试人员管理

开发人员：这些测试家伙……他们总在抱怨！
测试人员：为什么这些开发人员什么事都干不好？！
销售人员：什么时候我才能得到能够销售的产品？！

人员管理对于任何项目管理都是一个组成部分。人员管理对于从工程师转为经理的人员来说常常是一个难题，他们总依靠自己的技能来完成所分配的活动，他们没有受过必要的培训，不知道如何将需要做的写下来，以便由其他人完成。不仅如此，人员管理还要求有聘用、激励、保留合适人员的能力。这些技能很少有正式教授的（与技术技能不同）。项目经理常常在任务面前不得不以“摸着石头过河”的方式学习这些技能。

以上提到的大部分人员管理问题在所有类型的项目中都存在。测试项目还有一些其他挑战。我们认为测试公司（或测试人员个人）的成功非常依赖于明智的人员管理技能。由于人员和团队建设问题很重要，因此在第13章和第14章详细介绍。这两章讨论与构建和管理好的全球测试团队有关的问题，重点是如何有效地集成到产品开发和发布中。

应该强调的是，这些团队建设工作要持续进行，不能中断，不能一蹴而就。在产品交付最后期限和质量的压力下，团队建设工作会减弱，但是这种工作需要定期重温，重要的是，共同的目标和团队精神已经在所有相关人员的内部根深蒂固。一旦形成内部的东西，就不大会在项目执行过程中由于操作压力而左右摇摆。这种内在化和团队建设工作必须成为团队成功策划过程的一部分。

15.3.4 与产品发布集成

产品的最终成功取决于开发和测试活动集成的有效性。这两个部门之间必须密切协作，还要与产品支持、产品管理等部门密切协作。测试的进度计划必须直接与产品发布关联。因此，整个产品的项目策划应该全面，要包含测试和开发的项目计划。以下是这种策划的一些决策内容：

1. 开发和测试的同步点。什么时候可以开始不同类型的测试。例如，什么时候集成测试可以开始，什么时候系统测试可以开始等。什么时候开始是受每个测试阶段的客观的进入准则决定的。

2. 开发和测试之间的服务等级约定，即测试团队需要多长时间完成测试。这可以保证测试关注点只集中在发现相关的和重要的缺陷上。

3. 缺陷各种优先级和严重等级的一致定义。这涉及在开发和测试团队之间就所关注缺陷

的性质建立一致的认识。

4. 与文档编写小组的沟通机制，保证文档与产品在已知缺陷、避开方法等方面同步。

测试团队的目的就是找出产品中的缺陷，确定发布带有已知缺陷的产品所面临的风险。是否发布的最终决定权在管理层手里，他们要考虑市场压力，并权衡给公司和客户带来的商业影响。

15.4 测试过程

15.4.1 把各种要素放在一起并确定测试计划基线

测试计划要在一份文档中包含以上讨论过的所有要素。测试计划就是整个测试项目的锚点。本章附录B给出了一个测试计划的模板，附录A给出了对于形成测试计划很有用的检查单。

公司一般要制定在整个公司内使用的模板，每个测试项目都要根据该模板制定测试计划。如果需要对模板进行修改，那么必须经过仔细斟酌后进行（并经过合适的批准）。测试计划要经过公司中一些指定的有资格的人进行评审，然后由独立于直接负责测试的项目经理的相关负责人批准。然后，测试计划进入配置管理库，并建立基线。从此以后，成为基线的测试计划就成为推进测试项目的基础。测试项目中的任何重大变更都要在测试计划中反映出来，并对配置管理库内的测试计划重新建立基线。此外，有关各方还要定期就是否需要修改测试计划模板进行讨论，要保证模板对测试团队是最新的和适用的。

15.4.2 测试用例规格说明

以测试计划为基础，测试团队要设计测试用例规格说明。而测试用例规格说明又是准备测试用例的基础。本书通篇比较随意地使用测试用例这个词。实际上测试用例只不过就是在给定的环境中对产品进行的一系列操作步骤，使用预先定义的一组输入数据，预期产生预先定义的输出。因此，测试用例规格说明应该明确确定：

1. 测试的目的：说明测试针对的特性或部件。测试用例应该遵循与被测特性/模块一致的命名规则（前面已经讨论过）。

2. 被测项，以及合适的版本号。

3. 运行该测试用例所应建立的环境：可以包括硬件环境设置、支撑软件环境设置（例如操作系统、数据库等的设置）、被测产品的设置（正确版本的安装、配置、数据初始化等）。

4. 测试用例所使用的输入数据：所选择的输入数据将取决于该测试用例本身和测试用例所采用的技术（例如等价类划分、边界值分析等）。各个字段所使用的实际取值应该无歧义地定义（例如，不要只是说“输入一个二位数字的正整数”，而是说“输入789”）。如果测试用例要自动化，那么这些值应该收进文件中使用，而不应该是每次都手工输入数据。

5. 执行测试要遵循的步骤：如果采用自动化的测试，那么这些步骤要翻译为工具的脚本语言。如果要手工测试，那么这些步骤就是测试人员执行测试所使用的详细指示。重要的是要保证所编写步骤的详细程度要与执行测试人员的技能和专业知识一致。

6. 被认为是“正确结果”的预期结果。这些预期结果可以是用户可以看到的，例如GUI表单、报表等，也可以是对数据库或文件等固定存储的更新。

7. 将所得到的实际结果与预期结果的比较步骤：这个步骤应该对预期结果和实际结果进行“智能”比较，以发现可能的差异。所谓“智能”比较，是指应该考虑预期结果和实际结

果之间的"可接受的差异"，例如终端标识、用户标识、系统日期等。

8. 该测试与其他测试可能存在的关系：可以是测试之间的依赖关系，也可以是跨测试重用的可能性。

15.4.3 可跟踪性矩阵的更新

第4章已经讨论过，需求可跟踪性矩阵保证需求在后续生存周期阶段得到贯彻，不会中途被遗漏。具体地说，可跟踪性矩阵是确认每个需求都被测试过的一种工具。可跟踪性矩阵要在需求获取阶段本身建立，为每个需求确定唯一的标识符。以后，随着项目经历设计和编码阶段，设计特性和程序文件名称的唯一标识符要输入到可跟踪性矩阵中。当测试用例规格说明完成后，测试用例要测试的对应需求的标要用测试用例规格说明标识符更新。这要求在需求和测试用例之间存在双向映射关系。

15.4.4 确定有可能实现自动化的测试用例

测试用例设计构成编写测试用例的基础。在编写测试用例之前，需要考虑哪些测试用例要自动化，哪些要手工执行。第16章将详细介绍测试自动化问题，这里要说的，只是在决定要自动化的脚本时需要使用的准则包括：

1. 测试用例的重复性；
2. 自动化所需要的工作量；
3. 执行该测试用例所需要的手工干预量；
4. 自动化工具的成本。

15.4.5 测试用例的开发和基线确立

要根据测试用例规格说明和自动化测试用例的选择开发测试用例。测试用例的开发要把测试规格说明转换为能够执行该测试的形式。如果测试用例被确定为要自动化，那么需要用自动化工具语言编写测试脚本。如果是手工执行的测试用例，那么要写明执行测试和确认结果所需的详细的一步步的指示。此外，测试用例还要在文档中说明自最初开发以来所做的变更情况。因此，测试用例还要有变更历史文档，描述：

1. 做了什么变更；
2. 为什么要变更；
3. 谁做的变更；
4. 什么时间做的变更；
5. 简要描述变更的实现；
6. 这个变更对其他文件的影响。

测试用例的所有工作产品，即测试脚本、输入、预期输出等，都要按前面介绍的方式存入测试用例数据库和软件配置管理库中。由于这些工作产品要进入软件配置管理库，因此在形成基线之前，要经过评审和有关负责人的批准。

15.4.6 测试用例的执行与可跟踪性矩阵的更新

所准备的测试用例要在项目的合适时间执行。例如，对应冒烟测试的测试用例可能每天都要执行一次，而系统测试的测试用例只在系统测试阶段执行。

在测试周期中执行测试用例时，缺陷库也随之用以下信息更新：

1. 以前通过测试发现并在当前版本中已经修改的缺陷；

2. 在本次测试中发现的新缺陷。

缺陷库应该是测试团队和开发团队进行沟通的主要载体。前面讨论过，缺陷库包含有关测试发现的（以及客户报告的）缺陷的所有信息。所有有关各方都应该通过缺陷库了解所有缺陷的当前状态。这种沟通通过其他手段，例如电子邮件、会议通知等得到加强。

在讨论测试计划时已经提到过，测试在执行过程中可能由为出现一些影响测试继续进行的缺陷而被挂起。在这种情况下，被挂起的测试用例要等到恢复准则条件被满足后才继续运行。同样，测试用例的执行也要在进入准则条件满足后进行，在退出准则条件满足后结束。

在测试设计和执行期间，可跟踪性矩阵要随时更新。随着测试的设计和执行成功，要更新可跟踪性矩阵。可跟踪性矩阵本身也是配置管理的内容，即需要进行版本控制和变更控制。

15.4.7　指标的采集与分析

在执行测试时，要把有关测试执行的信息采集到测试日志记录和其他文件中。测试运行的基本度量再通过合适的转换和公式，变换为有意义的指标。第17章将详细讨论这个问题。

15.4.8　准备测试总结报告

测试周期结束时要产生测试总结报告。测试总结报告要向高层管理说明产品是否适合发布。本章稍后还要详细讨论测试总结报告。

15.4.9　推荐产品发布准则

测试的目的之一是确定产品是否适合发布。第1章已经介绍过，测试永远不能确定地证明软件产品中没有缺陷，测试所能提供的只是缺陷在产品中存在、严重程度和影响的证据。前面已经介绍过，测试团队的工作就是告诉高层管理和产品发布团队

> 产品在"被测"时，有可能面临"过测试"的风险，或因清除"最后一个缺陷"就是一种风险！

1. 产品中还有什么缺陷；

2. 每个缺陷的影响和严重等级；

3. 在已知缺陷没有修改的情况下发布产品会带来的风险。

然后高层管理会就是否发布给定版本作出合理的业务决策。

15.5　测试报告

测试需要测试团队和其他团队（例如开发团队）之间不断沟通。测试报告是实现这种沟通的一种手段。需要有两种类型的报告或沟通：测试事件报告和测试总结报告（又叫做测试完成报告）。

测试事件报告

测试事件报告是在测试周期内遇到缺陷时的沟通。我们在讨论缺陷库时提到过，测试事件报告只不过是缺陷库中的一条记录。每个缺陷都有唯一的标识，用于标识该事件。影响大的测试事件（缺陷）要在测试总结报告中指出。

测试周期报告

前面介绍过，测试项目以测试周期为单位实施。一个测试周期包括在周期内策划和执行测试用例，每个周期都使用不同的产品版本。随着产品经历了各种周期，产品预期越来越稳定。每个周期结束时的测试周期报告要给出：

1. 本周期内完成的活动总结；
2. 本周期内发现的缺陷，按缺陷的严重等级和影响分类；
3. 在缺陷修改方面前一个周期到当前周期的进展；
4. 本周期还没有修改的严重问题；
5. 工作量或进度上的任何偏差（下次策划时可以参考）。

测试总结报告

测试周期的最后一个步骤是对产品发布的适宜性提出建议。对一个测试周期的结果进行总结的报告叫做测试总结报告。

测试总结报告有两类：

1. 按阶段进行测试总结，在每个阶段结束时进行；
2. 最终测试总结报告（包含所有测试阶段和团队完成的所有测试，又叫做“发布测试报告”）。

总结报告应包含：

1. 本测试周期或阶段完成的活动总结；
2. 活动的实际执行和策划之间的偏差，包括：
   - 计划运行但是不能运行的测试（说明原因）；
   - 对原始测试规格说明的修改（在这种情况下，应该更新测试用例数据库）；
   - 运行的补充测试（没有列在原始测试计划中）；
   - 策划和实际工作量和时间的偏差；
   - 与计划的任何其他偏离。
3. 结果总结应包括：
   - 未通过的测试用例，并描述未通过的原因；
   - 测试发现的缺陷的影响严重等级。
4. 对发布的综合评估和建议应包括：
   - “发布适宜性”的评估；
   - 发布建议。

产品发布建议

公司可以根据测试总结报告就是否发布产品作出决策。

在理想情况下，公司希望发布零缺陷产品。但是，市场压力可能导致发布带缺陷的产品，只要高层管理确信不会有客户不满意的重大风险。如果残留缺陷的优先级和影响都很低，或出现这些缺陷的条件不现实，公司可能会决定发布带这些缺陷的产品。只有在征求了客户支持团队、开发团队和测试团队之后，高层管理才可以作出这样的决定，以便评估公司各个部门的总体工作量。

15.6 最佳实践

测试中的最佳实践可分为三类：

1. 与过程相关的最佳实践；
2. 与人员相关的最佳实践；
3. 与技术相关的最佳实践。

15.6.1 与过程相关的最佳实践

获得更好的可预测性和一致性要求具有很强的过程基础设施和过程文化。像CMMI这样的过程模型可以提供构建这种基础设施的框架。实现有业务意义的正式过程模型可以提供一致的全员培训，使任务的执行具有一致性。

实施员工友好的过程会造就过程友好的员工。

以明智的方式将过程与技术结合起来是公司成功的关键。过程数据库即有关各种过程的定义和执行信息，对于公司的有效运转是很有价值的。当这种过程数据库与其他工具，例如缺陷库、软件配置管理工具和测试用例数据库集成后，公司可以发挥最大效益。

15.6.2 与人员相关的最佳实践

本书第13章研究讨论了测试中与人员管理有关的最佳实践，以下总结一下这些测试管理方面的最佳实践。

成功管理的关键是保证测试和开发团队的密切配合。通过营造一种对产品总体目标认同感的氛围可实现这种配合。在开发团队和测试团队都有各自的目标情况下，对产品作为一个整体要定义总体目标，对这种总体目标的一致理解是非常重要的。测试团队通过参与这种确定目标的过程，通过早期参与整个总体的策划过程，能够实现这种密切配合。测试团队通过参与产品发布决策和发布准则的确定，有助于强化这种配合。

本章前面讨论过，支持、开发和测试人员的工作轮换也有助于强化团队之间的配合，有助于不同的团队更好地理解不同岗位面临的挑战，以便更好地相互协作。

15.6.3 与技术相关的最佳实践

前面介绍过，测试用例数据库、软件配置管理和缺陷库的完整集成有助于更好地实施测试自动化活动，有助于选择更可能发现缺陷的测试用例。即使没有满足各种要求的测试自动化工具，这三种工具的紧密集成也可以大幅度提高测试的有效性。第17章将要讨论每100小时测试发现的缺陷、缺陷密度、缺陷清除率等指标。通过这三种工具的紧密集成可以极大地简化这些指标的计算。

正如第16章将要讨论的，自动化工具承担了测试部门的枯燥和重复性工作，提高了测试部门的吸引力。尽管可能要作大笔的初始投资，但是自动化可以通过降低执行测试所需的工作量直接带来长期的费用节省。此外，在降低测试工程师之间的矛盾方面还有一些间接的好处，因为测试自动化工具不仅降低了重复性工作，还为测试岗位带来了一些工程师通常很喜欢的“编程”工作。

二十一世纪的工具与十九世纪的过程结合，只能产生十九世纪的生产率！

在使用测试自动化工具时，最好将这些工具与测试用例数据库、缺陷库和软件配置管理工具集成使用。事实上，大多数测试自动化工具在与商品化的软件配置管理工具集成时，都提供了测试用例数据库和缺陷库的功能。

关于最佳实践最后一点要说的是，最佳实践的三个方面不能孤立地实施。好的技术基础

设施需要有效的过程基础设施的支持，需要有高素质的人员执行，这些最佳实践是内部独立、自我支持和相互增强的。因此，公司需要全面考虑这些最佳实践，并在三个方面认真权衡综合考虑。

附录A：测试策划检查单

与范围有关的内容

- 确定了要测试的特性吗？
- 确定了不测试的特性吗？
- 产品管理层或高级管理层论证了选择不测试特性的理由及其影响吗？
- 确定了该发布版本的新特性吗？
- 在范围内是否包含了失效会造成灾难性后果的测试区域？
- 是否准备测试容易出现缺陷或很难测试的区域？

与环境有关的内容

- 是否有软件配置管理工具？
- 是否有缺陷库？
- 是否有测试用例数据库？
- 是否建立了更新这些库的制度化规程？
- 是否确定了设计和运行测试所需的硬件和软件？
- 是否确定了可能需要的测试制度化所需的经费和其他资源？

与准则定义有关的内容

- 是否为各个测试阶段定义了进入和退出准则？
- 是否为各种测试定义了挂起和恢复准则？

与测试用例有关的内容

- 是否公布了命名规则和其他设计、编写和执行测试用例的内部标准？
- 测试规格说明文档是否充分遵循了以上准则？
- 测试规格说明是否经过评审并得到合适人员的批准？
- 测试规格说明是否在软件配置管理库中形成基线？
- 测试用例是否根据规格说明编写？
- 测试用例是否经过评审并得到合适人员的批准？
- 测试用例是否在软件配置管理库中形成基线？
- 测试规格说明和测试用例形成基线后，是否更新了可跟踪性矩阵？

与工作量估计有关的内容

- 把范围转换为规模估计（例如测试用例数）了吗？
- 完成设计和构建测试所需的工作量估计了吗？
- 完成重复执行测试所需的工作量估计了吗？

- 工作量估计是否经过评审并得到合适人员的批准？

与进度有关的内容

- 制订利用了所有可用资源的进度安排了吗？
- 考虑并行开展工作的约束了吗？
- 考虑开发团队提交发布产品可能性方面的因素了吗？
- 考虑发现影响测试继续进行的缺陷方面的因素了吗？
- 在进度安排中对测试指派优先级了吗？
- 进度安排是否经过评审并得到合适人员的批准？

与风险有关的内容

- 确定测试项目中的可能风险了吗？
- 量化这些风险的可能性和影响了吗？
- 明确在风险发生之前就可发现的可能的征兆了吗？
- 明确这些风险的可能的缓解策略了吗？
- 考虑了不要把所有测试活动都挤在开发周期结束后的时间内吗？
- 采用什么机制（例如像在V字模型一样早期开始测试设计）将测试活动分散到生命周期各时点中？
- 考虑了由于测试被挂起时间空闲的风险吗？

与人员有关的内容

- 确定了所需的人员数量和技能水平要求吗？
- 确定了技能差距并安排了相关培训吗？

与执行有关的内容

- 按计划执行测试吗？如果出现偏差，更新计划了吗？
- 测试执行对测试用例设计做了一些必要的修改吗？如果修改了，测试用例数据库是否也被同步更新？
- 在缺陷库中记录了测试中发现的所有缺陷吗？
- 在缺陷库中更新了当前测试周期的所有缺陷修改吗？
- 可跟踪性矩阵是否也被同步更新？

与结束有关的内容

- 编写测试总结报告了吗？
- 在文档中清晰地写入了严重缺陷及其严重程度和影响吗？
- 对产品的发布提出了自己的建议吗？

附录B：测试计划模板

1. 引言

范围

哪些特性要测试，哪些特性不测试；测试哪些环境组合，不测试哪些环境组合。

2. 引用文档

（给出引用文档和链接，例如需求规格说明、设计规格说明、程序规格说明、项目计划、项目估计、测试估计、过程文档、内部标准、外部标准等）

3. 测试方法与策略

4. 测试准则

4.1 进入准则

4.2 退出准则

4.3 挂起准则

4.4 恢复准则

5. 假设、依赖关系与风险

5.1 假设

5.2 依赖关系

5.3 风险与风险管理计划

6. 估计

6.1 规模估计

6.2 工作量估计

6.3 进度估计

7. 测试交付产品与里程碑

8. 分工

9. 资源需求

9.1 硬件资源

9.2 软件资源

9.3 人力资源（人员的数量、技能要求、占用起止时间等）

9.4 其他资源

10. 培训需求

10.1 所需的培训细节

10.2 可能的被培训人员

10.3 约束条件

11. 缺陷记录与跟踪过程

12. 指标计划

13. 产品发布准则

问题与练习

1. 有人说，"测试严重依赖于开发，所以我不做测试项目策划。"如何反驳这种说法？

2. 请考虑一个新的操作系统版本，在确定该版本需要测试的范围时，必须测试以下哪些特性？

a. 操作系统引入对一种已经成为事实标准的新的网络协议的支持；

b. 操作系统经过修改，以便运行在火星载人飞船嵌入式系统中；

c. 操作系统支持长文件名的特性已经稳定，过去的五个版本都没有问题，也没有做过大的修改；

d. 操作系统的文件系统的缓存部分常常产生性能问题，每次微小变化都会引起大的麻烦；

e. 操作系统预期运行在不同的芯片和网络上，不同的计算机可能使用不同的芯片。

3. 请给出单元测试的一些典型的进入、退出、挂起和恢复准则。如果有区分单元测试中黑盒和白盒测试部分的准则，请考虑怎样对单元测试的白盒测试和黑盒测试分类。

4. 在讨论职责和人员需求时，考虑的都是测试团队的需求。测试计划还应该规定其他有关团队（例如开发团队）的什么服务等级约定或职责？应该怎样跟踪这类职责？

5. 为了估计测试项目的规模，请给出使用产品的代码行作为基础的优缺点。代码行只适用于白盒测试，还是也适合黑盒测试？

6. 估计涉及自动化的测试项目的规模和工作量需要哪些基本数据？

7. 假设已经有各种活动的历史数据，例如测试计划、设计、执行与缺陷修改，适配或修改这些历史数据以直接用于自己的项目时，需要考虑哪些因素？

8. 本章已经介绍过，风险指数被量化为风险发生的可能性与风险影响的乘积。如果该风险发生，如何估计这两个参数？如果没有这两个参数的量化数据，如何对风险进行量化？

9. 我们讨论了各种基础设施组件（测试用例数据库、缺陷库、配置管理库）。如何协调地利用以下工具提高有效性？

a. 为给定版本选择回归测试的测试用例；

b. 预测修改已发现缺陷后发布产品所需的时间；

c. 根据发现缺陷的有效性维护测试用例。

10. 修改本章给出的测试计划模板，以满足自己公司的需要。

11. 可以让客户访问缺陷库吗？如果可以，需要考虑哪些保密问题？

12. 测试事件报告需要包含哪些内容？注意我们的目标是在测试中发现的缺陷最终要被修改。

13. 公司在决定产品发布的适宜性时需要考虑哪些因素？

第16章 软件测试自动化

16.1 测试自动化的定义

以上各章已经讨论过多种测试类型，以及如何针对这些测试类型开发测试用例。经过这些测试用例的运行和检查，测试的覆盖率和质量（连同产品的质量）肯定会得到改进。但是这会带来一个问题，即运行这些测试用例需要额外的时间。解决这种问题的一个方法就是自动运行大部分需要反复运行的测试用例。

通过开发软件来测试软件叫做测试自动化。测试自动化有助于解决多种问题。

自动化可以节省时间，因为软件执行测试用例比人工执行快 这会有助于在夜间或无人介入控执行测试。这样节省下的时间可以被测试工程师有效用于：

1. 开发更多的测试用例以得到更好的覆盖率；
2. 执行一些复杂或特殊测试，例如即兴测试；
3. 执行额外的手工测试。

通过自动化节省的时间还可以用来开发额外的测试用例，因而提高测试的覆盖率。此外，自动化的测试可以在夜间执行，压缩测试时间跨度，因此能频繁发布产品。

测试自动化可以把测试工程师从人工任务中解放出来，使他们能把注意力放在更有创造性的任务上 第10章介绍过即兴测试，这种测试需要直觉和创造力，从测试用例没有涉及的视角测试产品。如果有太多的计划测试用例需要手工测试，就不会有多少自动化，那么测试团队就要把大部分时间花在测试执行上。这样就没有发挥测试团队直觉和创造力的余地，还会使测试团队感到厌烦和枯燥。自动化更多的琐碎工作使测试工程师有时间完成具有创造性和挑战性的任务。正如第12章所介绍的，激励测试工程师是一个重大挑战，而自动化从长远看有助于应对这种挑战。

自动化测试会更可靠 当工程师反复多次地手工执行某个特定的测试用例时，有可能犯错误或出现偏见，有些缺陷可能会因此而被遗漏。对于所有面向计算机的活动，可以预期自动化在任何时候都能够得到更可靠的结果，并能够消除厌烦和枯燥根源。

自动化有助于立即测试 自动化可以缩短开发与测试之间的时间间隔，因为产品一完成构建就可以执行脚本进行测试。通过设计，自动化测试可以在构建完成之后自动开始。自动化测试不需要等待有空闲的测试工程师。

公司通过自动化可以减少测试工程师的矛盾 自动化还可以用作针对产品进行测试工程师培训的知识转换工具，因为存储了产品的不同测试用例。对于手工测试，运行测试得到的任何特殊知识或“没有形成书面文档的方法”都会随着该测试工程师的离开而丢失。另一方面，自动化测试使得测试执行更少依赖人员。

测试自动化提供了更好利用全球资源的机会 手工测试需要测试工程师在场，而自动化测试可以一天二十四小时、一周七天地随时执行。这还使位于世界不同地方、不同时区的团队监视和控制测试，因此提供全时区的覆盖。

有些类型的测试不自动化不能执行 有些类型测试的测试用例，例如可靠性测试、压力测试、负载与性能测试，不自动化就不能执行。例如，如果要检查数千用户登录时系统的表现，那么不使用自动化工具是不能执行这样的测试的。

自动化用软件测试软件，使人能够把时间用在创造性测试上。

自动化意味着端到端，并不只是测试执行 自动化并不是开发了测试用例的程序就完了。事实上，开发完程序只是自动化的开始。自动化要考虑所有活动，例如选出正确的产品版本、选择正确的配置、执行安装、运行测试用例、生成合适的测试数据、分析测试结果、过滤缺陷库中的缺陷。在讨论自动化时，头脑中应该有一个全景图。

自动化应该有产生测试数据的脚本，以尽可能多地覆盖输入的排列组合，产生预期输出的脚本，以进行测试结果比较。这些脚本叫做测试数据生成器。确定所有输入条件的输出并不总是很容易。即使所有输入条件都已知，预期结果也满足，软件产生的错误和产品出错后的行为也可能是难以确定的。自动化脚本应该能够动态地映射错误模式，以得到测试结果。错误模式映射不仅要得出测试结果，而且要指出错误根源。自动化的定义包括这方面的内容。

以上已经提到过，自动化不仅要覆盖各种测试活动，而且当进一步采取的行动未知或不能自动判定（例如，如果测试结果不能由脚本判断是否通过）时，能够把控制返回给工程师。这一点也很重要。如果自动化脚本判断测试结果有误，这种结论会延迟产品的发布。如果自动化脚本被发现即使只有一两个这样的问题，那么不管其质量如何，都会失去信誉，都可能不能在后续的发布周期内使用，团队还会依赖手工测试。这可能会使公司认为自动化是所有问题的根源，有可能导致公司完全忽略自动化。因此，自动化不仅要尽可能覆盖所有操作活动，而且在必要时还要允许人的介入。

在自动化测试用例时需要注意所有的需求，人员转换或轮换运行测试用例会更容易地做到这一点。因为测试的目标是尽早捕获缺陷，因此经过自动化的测试用例可以交给开发人员，让他们作为单元测试的一部分运行这些测试用例。经过自动化的测试用例还可以交给配置管理工程师（构建工程师），以使版本构建完成可以执行测试。第8章已经讨论过回归测试。很多公司都实施“每天出一个版本和冒烟测试”策略，以保证版本可以供进一步测试，而且现有的功能没有因变更而被破坏。

16.2 自动化使用的术语

本书自由使用“测试用例”这个词。第15章已经正式把测试用例定义为执行测试操作的一组顺序步骤，其根据是为产生一定的预期输出而设计的一组预先定义的输入。有两类测试用例，一类是自动化的，一类是手工的。顾名思义，手工测试用例要手工执行，而自动化测试用例可自动执行。除非另外说明，本章所说的测试用例都指自动化测试用例。在执行时，测试用例永远都有一个与之关联的预期结果。

测试用例（手工或自动化的）可以用很多方式表示。可以在文档中写成一组简单的步骤，也可以是一个或一组判断语句。判断的一个例子是，“打开一个已经打开的文件应失败。”判断语句在自身中包含了预期结果，例如上面的例子，这便于自动化工程师编写测试步骤和判断该测试用例结果正确性对应的代码。

前面已经介绍过，测试包括多个阶段和多种测试类型。有些测试用例要在产品发布周期内重复执行多次，因为产品要构建多次。不仅测试用例要重复使用，在测试用例中所描述的一些操作也要重复执行。有些基本操作，例如“登录到系统”通常会在同一产品的很多测试

用例中执行。即使这些测试用例（以及这些测试用例中的操作）重复执行，每次测试的意图或关注的区域也在不断变化。这提供了针对不同测试目的和场景采用自动化代码的机会。

表16-1给出了登录例子中的一些测试用例，说明登录怎样在不同类型的测试中被测试。

表16-1 用于不同类型测试的相同测试用例

序号	用于测试的测试实例	涉及的测试类型
1	检查是否能登录	功能
2	登录操作连续执行48个小时	可靠性
3	10000个客户进行登录	负载与压力测试
4	在某个环境条件下登录操作所需的时间	性能
5	在运行日文的计算机上进行登录操作	国际化

表16-1可以针对其他类型的测试进行扩展。从以上例子可以看出，如果尝试对产品进行不同类型的测试，可以对产品重复进行的一定操作（例如登录）。如果记住这一点，为自动化登录操作编写的代码就可以在很多地方重用，因此可以节省工作量和时间。

如果仔细研究表16-1就会发现两个重要因素，一是"要测试什么操作"，二是"如何测试这些操作。"测试用例的"如何"部分与散场景（在表16-1中用楷体表示）。"要测试什么操作"是与具体产品有关的特性，"如何测试这些操作"是与具体组织有关的需求。与具体的框架/测试工具有关的需求不仅是针对测试用例或产品的，而是对公司内所有被测产品的一般需求。

自动化信念的基础是这样一种事实，即产品操作（例如登录）是可重复的，通过自动化基本操作且把不同场景（如何测试）留给框架/测试工具完成，就可以取得很大进展。这可以保证针对自动化的代码重用，并在"必须执行什么测试"和"应该补充什么框架或测试工具"之间画出明确的界限。如果场景通过产品的基本操作组合起来，就成为经过自动化的测试用例。

如果一组测试用例被组合并与一组场景关联，就叫做"测试包"。测试包只不过是被自动化的一组测试用例和与这些测试用例关联的场景。

图16-1给出了以上讨论的术语。

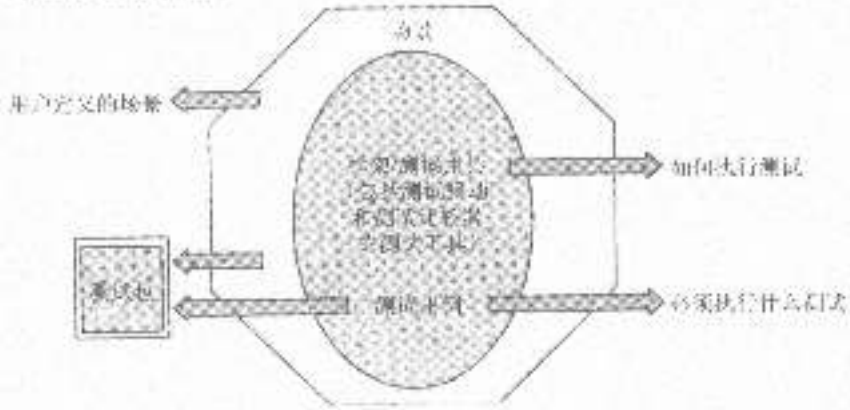


图16-1 测试自动化的术语

16.3 自动化所需的技能

有不同的"自动化代"。自动化所需的技能取决于公司所处的自动化代，或期望不远的将来将要达到的自动化代。

测试自动化可以大致分为三代。

第一代——记录与回放 记录与回放可避免重复执行测试用例。市场上几乎所有的测试工具都具有记录与回放特性。测试工程师记录键盘字符或鼠标点击的行动序列，并在以后按照录制的顺序回放这些所记录的脚本。由于所录制的脚本可以回放很多次，所以可以减少测试工作。除了可以避免重复工作，录制和保存脚本也很简单。但是这一代工具也有一些缺点。脚本中可能包含一些硬编码的取值。因此很难执行一般类型的测试。例如，如果报告必须使用当前的日期和时间，那么就很难使用已录制的脚本。错误条件的处理留给测试人员，这样回放脚本就可能需要很多人工介入来检测和更正错误条件。当应用程序变更后，所有脚本都必须重新录制，因此增加了测试维护的成本。所以，如果频繁出现变更，或没有多少机会重用或重新运行测试用例，那么这一代测试自动化工具的有效性就可能很低。

第二代——数据驱动 这种方法有助于开发生成输入条件集和对应预期输出的测试脚本。这使得测试可以针对不同输入和输出条件重复执行。这种方法所用的时间、工作量和产品开发相同。但是，只要输入条件和预期输出仍然有效，那么应用程序的变更就不要求自动化测试用例也作相应的变更。这一代测试自动化工具关注输入和输出条件，使用的是黑盒测试方法。

> 自动化在测试和开发所需的技能之间建立了桥梁，同时对测试团队提出了更高的技能要求。

第三代——行动驱动 这种技术使外行也可以创建自动化的测试用例。运行这样的测试用例不要求提供输入和预期输出条件。应用程序中出现的所有行动都会以为自动化定义的一般控件集为基础。自动测试，行动集表示为对象，采用这些对象，用户只需要描述操作（例如登录、下载等），其他所需的一切都会自动生成和使用。输入和输出条件会自动生成和使用。使用这一代测试自动化工具，测试执行的场景可以使用测试框架动态变更。因此，第三代自动化包含两个主要因素，测试用例自动化和框架设计。下一节将详细介绍框架设计问题。

通过以上各代测试自动化可以看出，所选的自动化代数不同，所需的技能级别也不同。第三代自动化引入了两级测试用例开发和框架技能，因此三代自动化所需的自动化技能可以分为四级，如表16-2所示。

表16-2　自动化技能的分类

自动化第一代	自动化第二代	自动化第三代	
测试用例自动化技能 脚本语言 记录-回放工具的使用	测试用例自动化技能 脚本语言 程序设计语言 数据生成技术知识 的测产品的使用	测试用例自动化技能 脚本语言 程序设计语言 标准产品的设计和体系结构 框架的使用	业务技能 程序设计语言 创建框架的设计和体系结构技能 各个产品的适用测试需求

16.4　自动化的对象与范围

产品开发的第一个阶段是需求获取，对于测试自动化也同样如此，因为自动化的输出也可以看作是一种产品（自动化的测试用例）。自动化需求定义需要自动化的对象，包括各个方面，具体的需求随产品的不同而不同，随时间的不同而不同。以下是确定自动化范围的一般提示。

16.4.1 确定自动化负责的测试类型

有些类型的测试本身自动进行自动化。

压力、可靠性、可伸缩性和性能测试 这些类型的测试要求在大量不同的计算机上以一定的持续时间运行测试用例，比如24小时、48小时等。让数百个用户天天使用产品简直就是不可能的，他们既不愿意承担重复性工作，也不可能找到那么多有所需技能的人群。因于这些类型测试的测试用例是自动化的第一候选者。

回归测试 回归测试是重复性的。这些测试用例在产品开发各个阶段要执行多次。由于这些测试用例具有重复性，因此自动化从长远看会显著节省时间和工作量。此外，正如本章已经提到过的，所节省的时间可以有效地用于即兴测试和其他更具创造性的测试。

功能测试 这类测试可能需要复杂的设置，因此可能需要当前还没有普通具备的特殊技能。利用专家的技能一次性自动化这些测试用例，使技能不那么高的员工也可以马上运行这些测试用例。

在产品开发场景中，很多测试需要重复，如果考虑了定期增强和维护发布版本，好的产品会有很长的生命期。这就提供了自动化测试用例在发布周期内多次执行的机会。根据一般经验，如果测试用例在不久的将来，比方说一年内需要执行至少十次，如果自动化工作量不超过执行这些测试用例的十倍，那么就可以考虑自动化这些测试用例。当然，这只是根据经验，具体选择哪些测试用例还有很多因素需要考虑，例如是否具备所需的技能、在强大的发布日期压力下是否有设计自动化测试脚本的时间、工具的成本、是否有所需的支持等。

作为自动化范围的总结，就是要选择自动化那些能够以最少的时间延迟换得最大投入回报的工作（根据以上给出的指南）。

16.4.2 自动化不太可能变更的部分

在产品开发过程中，需求的变更是很常见的。对于这种情况，要自动化的对象是很容易确定的。自动化应该考虑需求不变或没有变更的部分。需求变更一般会影响场景和新特性，不会影响产品的基本功能。第10章在讨论汽车制造的例子中已经解释过，汽车的基本组件，例如方向盘、制动和加速装置都已经很多年没有变化了。在自动化时，要首先考虑产品的这类基本功能，以便用作“回归测试集”和“日常构建和冒烟测试”。

用户界面在项目开发中通常会发生重大变化。为了避免对自动化的测试用例进行返工，需要进行恰当的分析，找出用户界面的变化部分，并只对变化相对很小的部分进行自动化。产品的非用户界面部分可以首先自动化。在自动化涉及面向用户界面和非用户界面（后台）要素时，给出明确的划分，以便测试用例既可以一起执行，也可以独立执行。这可以保证即使GUI发生变更，自动化的非GUI部分也能够被重用。

16.4.3 自动化测试符合标准

产品的一项测试是验证与标准的符合性。例如，提供JDBC接口的产品应该通过标准JDBC测试，这些测试变化相对较小，即使发生变更也具有后向兼容性，自动化脚本也可以继续运行。

提供标准的自动化有两个优点。针对标准开发的测试包不仅可以用于产品测试，而且可

以作为测试工具在市面上出售。市场上的很多工具最初都是针对内部使用开发的。因此，针对标准的自动化可以提供销售工具的机会。

如果市场上已经有能应这类标准的工具，就没有理由再重新发明轮子，再重新构建这些测试了。相反，注意力应该放在还没有工具的其他区域和向其他工具提供接口上。

针对标准的测试具有一定的法律和公司需求。为了认证软件或硬件，需要开发测试包并交付给不同的公司。每次在发布软件和硬件前，支持公司都要执行认证测试包，这叫做“认证测试”，每次执行测试时都要求完全兼容的结果。进行认证测试的公司可能对产品和标准了解不多，但要完成测试的绝大部分工作。因此，这个方面的自动化可以长久进行下去，肯定是自动化要关注的一个区域。例如，有些公司针对其软件产品开发了测试包，硬件制造商在发布新的硬件平台前要执行这些测试包。这使客户可以判定所发布的新硬件与市场上的流行软件产品兼容。

16.4.4　自动化的管理问题

在开始自动化前，需要花很大的精力取得管理层的承诺。自动化一般要耗费大量工作量，也并非一次性活动，自动化的测试用例还需要维护，直到产品退出市场。由于开发和维护自动化工具需要大量的工作量，因此取得管理层的承诺是一项很重要的活动。由于自动化在很长时间内都需要投入，因此管理层的批准是按阶段按部分进行的。所以，自动化工作应该要中于已经在在管理层承诺的区域。

要自动化哪些对象需考虑技术和管理因素，也要有长远眼光。

投入回报也是需要认真考虑的一个方面。自动化工作最后还要向管理层提供预期投入回报的明确结论。在启动自动化时，关注点应该放在好的组合区域上。这使自动化能够用较少的代码覆盖较多的测试用例。第二，自动化应该首先考虑需要较短时间易于自动化的测试用例。有些测试用例没有能够预先确定的预期结果，这类测试用例需要很长时间自动化，应该放在自动化的后期阶段，这可以满足管理层寻求自动化快速投入回报的要求。

为了符合Stephen Covey“重要事情优先做”[COVE-89]的原则，重要的是要首先自动化产品的关键和基本功能。为此，所有测试用例都要根据客户预期分为高、中、低优先级，自动化要从高优先级的测试用例入手，然后覆盖中、低优先级需求的测试用例。

16.5　自动化的设计和体系结构

设计和体系结构是自动化的一个重要问题。与产品开发一样，设计也要通过模块和模块之间的交互体现所有的需求。第5章已经介绍过，设计和体系结构需要说明内部接口和外部接口。在图16-2中，细箭头表示内部接口和流程方向，粗箭头表示外部接口。所有模块、模块用途和模块之间的交互将在下一节介绍。

测试自动化的体系结构包括两个主要要素：测试基础设施包括测试用例数据库和缺陷库，这在图16-2中表示为外部模块。利用这种基础设施，测试框架可以提供将测试用例的选择和执行捆绑在一起的骨干。

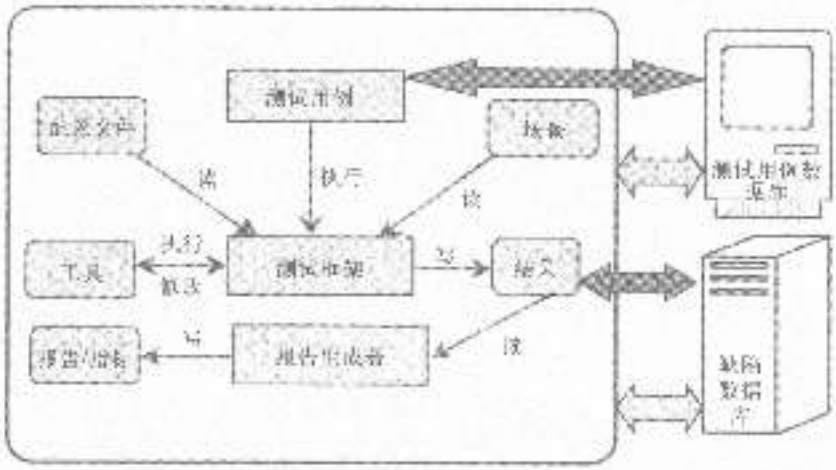


图16-2 测试自动化的组件

16.5.1 外部模块

测试自动化有两个外部模块，测试用例数据库和缺陷库。第15章详细讨论了这两个数据库。所有测试用例、执行测试用例的步骤，以及执行历史（例如特定测试用例的执行时间和是否通过等）都存储在测试用例数据库中。测试用例数据库中的测试用例可以是手工的，也可以是自动化的。图16-2中的扭曲头接口表示测试用例数据库与自动化测试用例的自动化框架之间的交互。请注意，手工测试用例不需要与框架和测试用例数据库交互。

缺陷库包含特定公司在各种被测产品中发现的所有缺陷的详细信息，包括缺陷和所有相关信息（发现缺陷的时间、分配给的员工、当前状态、缺陷类型、缺陷影响等）。测试工程师提交手工测试用例发现的缺陷。对于自动化测试用例，框架可以在执行期间自动地向缺陷库提交缺陷。

这些外部模块可以被自动化框架中的任何模块访问，而不只是一两个模块。在图16-2中，深绿色粗箭头表示特殊交互，浅蓝色粗箭头表示多重交互。

16.5.2 场景与配置文件模块

前面几节介绍过，场景只不过就是有关“如何执行特定测试用例”的信息。

配置文件包含一组在自动化中使用的变量。这些变量可以是针对测试框架的，也可以是针对测试自动化中的其他模块的。例如工具与指标，也可以是针对测试包、一组测试用例或一个特定的测试用例。配置文件对于测试用例在不同执行条件下的运行，以及针对不同输入和输出条件和状态的执行是很重要的。在这种配置文件中的变量取值可以动态改变，以实现不同的执行、输入、输出和状态条件。

16.5.3 测试用例与测试框架模块

图16-2中的“测试用例”表示来自测试用例数据库并由框架执行的自动化测试用例。测试用例是供体系结构中其他模块执行的对象，本身并不表示任何交互。

“测试框架”是将“要执行什么”和“如何执行测试”结合在一起的模块。测试框架从测

试用例数据库中提取要自动化的特定测试用例，提取场景并执行。测试框架提取变量及其定义的取值，并用这些取值执行测试用例。

测试框架被认为是自动化设计的核心，它为测试用例赋予不同的场景。例如，如果有场景要求48小时循环执行一个特定测试用例，那么测试框架就会循环执行测试用例，直到达到48小时。框架会监视每次迭代的执行结果，并把结果存储起来。测试框架包含交互、启动并控制所有模块的逻辑。下一节将介绍测试框架的各种需求。

测试框架可以由公司内部开发，也可以向提供商购买。本章混用测试框架和测试工具这两个术语。在本章中这两个术语的差别（如果需要区分）是，“框架”表示公司开发的内部工具，“测试工具”表示向提供商购买的工具。

16.5.4　工具与结果模块

当测试框架执行其操作时，可能需要一组工具。例如，当测试用例作为源代码文件存储在测试用例数据库中时，需要由构建工具提取并编译。为了运行编译过的代码，可能需要特定的运行时间工具和实用程序。例如可能需要IP分组模拟器、用户登录模拟器或计算机模拟器。在这种情况下，测试框架会调用所有这些不同的工具和实用程序。

当测试框架执行一组测试用例时，要使用由配置文件提供的不同取值下的一组场景，要将每个测试用例的执行结果，连同对应的场景和变量值一起存储，供进一步分析和处理。测试框架运行测试用例得到的结果不能覆盖以前运行测试用例得到的结果。应该记录所有以前运行的历史结果，并归档保存。归档结果有助于根据以前的测试结果执行测试用例。例如，测试工程师可以要求测试框架“执行所有以前运行没有通过的测试用例。”对所有测试的审计和相关信息存储在自动化模块中。正如第8章所介绍的，这也有助于选择回归测试的测试用例。

16.5.5　报告生成器与报告/指标模块

一旦得到测试运行的结果，下一步就是准备测试报告和指标。准备报告是一项很复杂的工作，很费时间，因此是自动化设计的一部分。应该有经过定制的报告，例如执行报告，给出非常高层的状态；技术报告，提供中等详细程度的测试结果；详细或调试报告，供开发人员调试未通过测试用例和产品时参考。报告的频度也是不同的，例如日报、周报、月报和里程碑报告。不同详细程度、不同频度的报告可以满足多种人员的需要，因此能够得到重要的反馈。

提取必要的输入并准备格式化报告的模块可称“报告生成器”。一旦有了测试结果，报告生成器就可以生成指标。

所生成的所有报告和指标都存储在自动化的报告/指标模块中，以备日后使用和分析。

16.6　测试工具/框架的一般需求

上一节介绍了测试自动化的一般框架，以下介绍这种框架及其使用应该满足的具有一定详细程度的准则。

> 需求1：在测试包中不要包含硬编码

在解释需求时，这里使用假想的元语言为例，以便于理解概念。读者应该验证自己所选自动化工具的可用性、句法和语意。

对测试包的一个重要需求，是把所有变量放到一个独立的文件中。

通过这种实践，测试包的源代码在针对变量的不同取值运行时不必每次都修改。这使不了解程序的测试人员也能修改变这些取值，并运行测试包。前面介绍过，测试包的变量叫做配置变量。保存所有变量名及其相关取值的文件叫做配置文件。很有可能配置文件中有多个变量，有些变量是针对测试工具的，有些变量是针对测试包的。属于测试工具的变量和测试包需要分开，使得测试包的用户不必考虑测试工具变量。此外，不知道测试工具变量的用途就无意地对其修改，可能会影响测试的结果。为每个变量提供行内注释会使测试包更好用。还可以避免不恰当地使用变量。以下给出一个编写良好的配置文件。

```
#测试框架配置参数
#警告：未咨询系统管理员前不要修改此设置
TOOL_PATH=/tools                          #测试工具路径
COMMONLIB_PATH=/tools/crm/lib             #公共库函数
SUITE_PATH=/tool_s/crm                    #测试包路径
#测试包中所有测试用例都适用的参数
VERBOSE_LEVEL=3                           #显示在屏幕上的消息级别
MAX_ERRORS=200                            #测试生成日前所允许的最大错误数
USER_PASSWD=hello123                      #系统管理员口令
#测试用例1参数
TC1_USR_CREATE=0                          #是否创建用户，1=是，0=否
TC1_USR_PASSWD=hello1                     #用户口令
TC1_USR_PREFIX=usor                       #用户前缀
TC1_MAX_USRS=200                          #最大用户数
```

在涉及多个发布版本、多个测试周期和映射修改的产品场景中，测试用例会发生很多变更，还会在测试包中增加新的测试用例。测试用例修改和新测试用例插入不能使现有测试用例失败，如果这种修改和新测试用例影响到测试包的质量，就违反了自动化的宗旨，测试包的维护需求成本就会增高。类似地，测试工具不仅用于拥有一个测试包的一个产品，而是面向有多个测试包的不同产品，在这种情况下，在框架中增加测试包而不影响其他测试包是非常重要的。归纳起来就是：

需求2：测试用例和测试工具有可扩展性

- 增加测试用例不能影响其他测试用例；
- 增加测试用例不能导致对整个测试包的重新测试；
- 在框架中增加新的测试包不能影响现有的测试包。

在“登录”例子中介绍过，产品功能在不同的场景下就成为不同类型测试的测试用例，这有利于在自动化中重用代码。通过遵循框架和测试包的目标分别包含自动化的“如何”和“什么”部分，可以提高测试用例的重用程度。代码的重用不仅适用于各种类型的测试，而且适用于自动化内的模块。多个测试用例需要的所有功能都可以分别包含在库中。在编写自动化代码时，通过提供函数、库和包含文件，可以使测试用例模块化。归纳起来就是：

需求3：针对不同测试类型和测试用例重用代码

1. 测试包只能做所期望做的事，测试框架需要处理“如何”方面的问题；

2. 测试程序需要模块化，有利于代码的重用。

对于每个测试用例都可能有一些在运行前要满足的前提。测试用例可能预期创建了一些对象，或产品的一部分已经以特定的方式配置。如果这类前提不能被自动化满足，那么在运行测试用例之前需要人工介入，如果测试用例预期特定的环境运行，会很难记住每个测试用

例所要求的环境，并以手工方式进行设置。因此，每个测试程序都应该有在执行该测试用例前创建必要的环境"设置"程序，测试框架应该有能力发现要执行哪些测试用例，以及需要调用的合适设置程序。

需求4：设置和清理自动化

一个测试用例的设置对于另一个测试用例可能是负面的。因此，重要的是不仅要创建环境，而且要能在该测试用例执行完立即"撤销"这些设置。因此，"清理"程序就变得很重要，测试框架应能在执行完测试用例后调用这种程序。

需求2讨论了测试用例的可扩展性。测试用例不仅要在设计阶段中独立，在执行阶段也要独立。为了执行特定的测试用例，不能期望以前已经执行过其他测试用例，也不能隐含地假设本测试用例执行完还要执行另外的测试用例。每个测试用例都应该单独执行，在测试用例之间不能存在依赖关系，例如测试用例2要在测试用例1之后运行等。这种需求使测试工程师能够以随机选择和执行任何测试用例，不必考虑其他依赖关系。

需求5：测试用例要独立

与需求5相反，有时测试用例也需要相互依赖。使测试用例独立能够随机选择和执行任何测试用例，使测试用例相互依赖，必须在选择执行一个依赖测试用例执行之前或之后执行一个特殊测试用例。测试工具或框架要提供这两种特性。框架要有助于动态地描述测试用例之间的依赖关系。

需求6：测试用例要有依赖性

通过环境隔离测试用例是对框架或测试工具的一个重要需求。在执行测试用例时，系统中可能有事件、中断或信号影响执行。例如Web浏览器中自动弹出的窗口。如果在执行测试用例时出现这种弹出窗口，就会影响测试用例的执行，因为测试包可能根据测试用例中前一步预期出现某个其他窗口。

需求7：在执行期间隔离测试用例

因此，为了避免测试用例由于不可预见的事件失败，框架要向用户提供屏蔽某些事件的选项。框架中必须有选项描述什么事件会影响测试包，什么事件不会。

将测试包的标准和合适的目录结构写入代码，有助于新工程师快速理解测试包和测试包的维护。将标准写入代码可提高测试包的可移植性。测试工具应有选项描述（有时要强制）将标准写入代码，例如POSIX、XPG3等。测试框架应提供选项或强制将目录结构写入代码，使多个程序员能够并行开发测试包和测试用例，不会通过重用部分代码复制测试用例的各个部分。

需求8：将标准和目录结构写入代码

一个框架可能有多个测试包，一个测试包可能有多个测试程序，一个测试程序可能有多个测试用例。测试工具或框架要能够使测试工程师选择一个特定的测试用例或一组测试用例并执行。测试用例的选择不需限制顺序，允许以任何方式组合，允许测试工程师选择测试用例可以节省时间，把注意力集中在要执行和分析的测试用例上。这些选择一般作为场景文件的一部分，可以通过编辑场景文件，在运行这些测试用例之前动态地选择。

需求9：测试用例的选择执行

例：

```
test program-name 2, 4, 1, 7-10
```

在上面的场景行中，选择了测试用例2，4，1，7、8、9、10，并按此顺序执行。用连字符（-）表示所选择的测试用例范围，即从7到10都执行。如果在上面的例子中没有提到测试用例数字，那么测试工具要有能力执行所有的测试用例。

需求10：测试用例的随机执行

需求8要求测试工程师在可用的测试用例中选择测试用例，同样的测试工程师有时会需要

从一组测试用例中随机地选择测试用例。给出一组测试用例，并期望测试工具选择测试用例叫做测试用例的随机执行。测试工程师从测试包中选择一组测试用例，由测试工具从给出的列表中随机选择一个测试用例。以下是说明随机执行的两个例子。

```
例1:
random
 test-program-name 2, 1, 5

例2:
random
 test-program1 (2, 1, 5)
 test-program2
 test-program3
```

在第一个例子中，测试工程师要测试工具从测试用例2、1、5中选一个并执行。在第二个例子中，测试工程师要从测试程序1、2、3中随机地选一个执行。如果程序1被选中，则随机地执行测试用例2、1、5中的一个。在这个例子中，如果程序2或3被选中，则该程序中的所有测试用例都要执行。

需求11：测试用例的并行执行

有些缺陷要在特定测试用例同时运行时才会暴露出来。在多任务和多进程操作系统中，有可能创建多个测试例程，并并行运行这些例程。并行执行可模拟多台计算机运行相同测试的行为，因此对于性能和负载测试非常有用。

```
例1:
instances, 5
   Test-program1 (2)

例2:
instances, 5
   test-program1 (2, 1, 5)
test-program2
   test-program3
```

在上面的第一个例子中，要创建5个测试程序1的测试用例3例程。在第二个例子中，要创建三个测试程序的5个例子。在每个例程中，都要创建5个用例，顺序执行测试程序1、2、3。

需求12：测试用例的循环

前面已经介绍过，可靠性测试需要循环执行测试用例。有两种循环，一种是迭代循环，给出特定测试用例要执行的迭代次数。第二种是定时循环，循环一直进行，直到所指定的时间停止。这些循环可以解决可靠性测试问题。

```
例1:
Repeat loop, 50
   Test-program1 (3)

例2:
Time loop, 5 hours
   test-program1 (2, 1, 5)
test-program2
   test-program3
```

在上面的第一个例子中，测试程序1的测试用例3要重复执行50次。在第二个例子中，测

试程序1的测试用例2，1，5和测试程序2和3的所有测试用例顺序循环执行5个小时。

需求13：对测试场景进行分组

前面介绍了测试执行的很多需求和场景，以下讨论如何将这些单个的场景组合为组，以结合很多测试用例长时间地运行。场景组使所选择的测试用例能够同时顺序、随机、循环执行，允许多个测试用例在预先确定的场景组合条件下执行。

以下是场景组的一个例子。

```
例:
group_scenario1
        parallel, 2 and repeat, 10 #scen1
scen1
        test-program1 (2, 1, 5)
        test-program2
        test-program3
```

在上面的例子中，创建了场景组循环10次执行单个场景"scen1"的两个例程。单个场景定义为执行测试程序1（测试用例2、1和5），测试程序2和测试程序3。因此在这个场景组中，两个例程要同时循环10次执行所有的测试用例。

需求14：根据以前的结果执行测试用例

第8章提到过回归测试方法论要求根据以前的测试结果和历史来选择测试用例。因此，如果选择测试用例时不考虑以前的测试结果，则自动化就不能提供多少帮助。不仅是回归测试，其他类型的测试也有这样的需求：一种有效的实践是选择没有执行过的测试用例，以及以前执行未通过的测试用例，并关注这些测试用例。依据以前的测试结果执行测试用例的常见场景有：

1. 重新运行以前执行过的所有测试用例；
2. 从上一次停下来的地方恢复运行测试用例；
3. 只重新运行未通过或未运行的测试用例；
4. 执行在"2005年1月26日"执行过的所有测试用例。

借助自动化，如果测试工具或框架能够帮助进行这类选择，那么完成这类任务非常容易。

大多数产品测试都需要使用多台计算机。因此需要有能力通过一个集中场地同时启动多台计算机的测试。这台中心计算机将测试用例分配给多台计算机，并协调其执行和采集结果数据。这种中心计算机叫做测试控制台或测试监视器。如果没有测试控制台，不仅在多台计算机的执行测试很困难，而且从这些计算机上采集执行结果也很困难。如果没有测试控制台，就需要人工收集和整理测试结果，因为引入人工步骤违背了自动化的宗旨，因此这是对框架的一个很重要的要求。归纳起来就是：

需求15：远程执行测试用例

- 能够通过测试控制台，执行或停止任何一台或一组计算机上的测试包；
- 能够通过测试控制台采集测试结果和日志记录；
- 能够通过测试控制台了解测试进展。

图16-3描述了测试控制台和多台测试计算机所充当的角色。

需求16：自动归档测试数据

需求14要求根据以前的结果重复执行测试用例。为了证实测试结果或重现问题，重复执行测试用例是不够的。测试用例必须以与以前一样的方式重复，具有同样的场景、同样的配置变量和取值等。这要求把测试用例的所有相关信息归档。因此，这种需求对于重复执行测试用例进行分析非常重

要。测试数据的归档必须包括：

1. 使用的什么配置变量；
2. 使用的什么场景；
3. 执行的什么程序，通过什么路径执行。

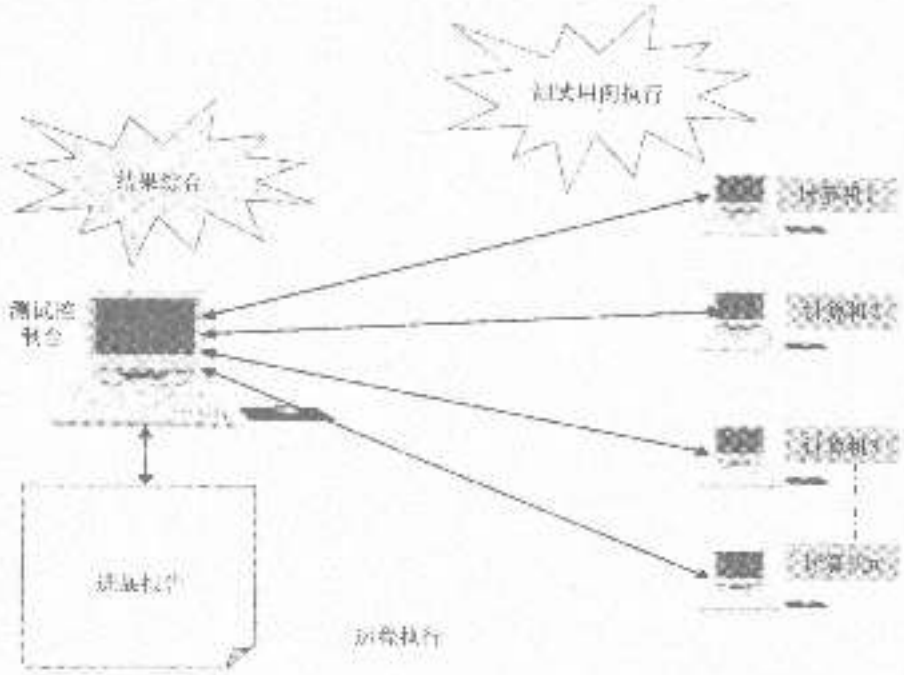


图16.3 测试控制台和多台执行计算机的角色

每个测试包都要有一个报告生成模式，通过这种模式可以抽取有意义的报告。讨论框架的设计和体系结构时已经介绍过，报告生成器应该有能力检查结果文件并生成各种报告。因此，尽管报告生成器用于动态生成报告，但是很难说清需要什么信息、不需要什么信息。因此，需要在结果文件中存储有关测试用例的所有信息。

需求17：报告生成模式

不仅是配置变量会影响测试用例及其执行结果，产品和操作系统的可调谐参数也需要归档，以保证可重复性和缺陷的分析。

审计日志记录对于分析测试包和产品的行为非常重要。日志记录保存每个操作的详细信息，例如什么时候调用的操作、变量的取值、操作完成的时间。对于性能测试，日志记录保存的信息有框架调用的时间、场景启动的时间、特定测试用例的启动时间及其对应的完成时间，这对于计算产品的性能很重要。因此，报告模式应该包括：

1. 框架、场景、测试包、测试程序和每个测试用例的启动和结束时间；
2. 每个测试用例的执行结果；
3. 日志记录消息；
4. 事件类别和事件记录；
5. 审计报告。

需求18：语言的独立性

在为自动化编码时，有些测试用例采用工具提供的脚本语言比较容易编码，有些工具采

用C，有些采用C++等，因此，框架或测试工具应提供在该软件开发领域流行的语言和脚本选择。不管自动化使用什么语言或脚本，框架都要以同样的方式运行，满足所有需求。很多测试工具都强制使用特定的语言，或强制使用专用的脚本语言编写测试脚本。应避免这种情况，因为要学习新的语言，而这会对自动化产生影响。归纳起来就是：

- 框架应该独立于程序设计语言和脚本；
- 框架应该提供程序设计语言、脚本及其组合的选择；
- 框架或测试包不应强制使用一种语言或脚本；
- 框架或测试包应能与采用不同的语言和脚本编写的不同的测试程序一起使用；
- 框架应该有输出接口以支持所有流行的标准语言和脚本；
- 框架使用的内部脚本和选项应该允许测试包的开发人员迁移到更好的框架上。

随着独立于平台的语言和技术的出现，市场上有很多产品支持多种操作系统和语言平台，产品是跨平台的，而测试框架不能在其中的有些平台上运行，这不利于自动化。因此，测试工具和框架要能够跨平台，能够在被测产品支持的各种平台和环境中运行，这是很重要的。

需求19：到不同平台的可移植性

因为测试工具需要跨平台，因此针对产品的测试包也跨平台，或通过少量的工作就可移植到其他平台上也是很重要的。

除了测试工具的跨平台能力检查单，检查平台体系结构也很重要，例如，市场上有64位操作系统和在64位体系结构中运行的产品，被测产品是64位，而测试包是32位的应用程序可能不会是大问题，因为所有具有兼容性的平台都支持运行在32位体系结构上的老应用程序。但是，这可能不能发现产品中的一些缺陷。在不提供后向兼容性的纯64位环境（例如Digital的Tru64）中，这些32位的测试包根本不能运行。归纳起来就是：

- 框架及其接口应该被各种平台支持；
- 对不同平台的兼容性是对测试工具和测试包的基本需求；
- 应该精心选择在测试包中使用的语言和脚本，以便在不同平台中使用；
- 针对测试包编写的语言和脚本不应包含与具体平台有关的调用。

16.7　自动化的过程模型

什么时候启动自动化，什么时候结束并没有硬性规则。自动化工作可以与产品开发同时进行，并且可以与产品的多个发布版本重叠。与多产品发布一样，自动化也有多个发布版本。对自动化的一个具体需求就是，自动化测试的交付应该在测试执行阶段之前，以便自动化可支持产品能够在测试被测产品的当前发布版本时使用。自动化的需求跨多个发布版本的多个阶段，这与产品需求一样，测试执行可以在发布产品后不久结束，但是自动化工作却在产品发布后仍然持续。

由于产品开发和自动化开发具有以上的相似性，因此，自动化遵循的过程和生存周期模型和产品开发的也非常类似。在绝大多数情况下，自动化也遵循产品的软件开发生存周期（SDLC）模型。本节集中介绍V字模型及其扩展，以介绍测试自动化的过程，首先研究产品开发所包含的阶段和如图16-4所示的自动化阶段，并理解两者之间的相似性。

本章已经介绍过，测试自动化生存周期活动与产品开发活动有很强的相似性，就像产品需要采集产品需求一样，也需要采集自动化需求，类似地，就像产品策划、设计和编码一样，在测试自动化过程中也有自动化策划、设计和编码。

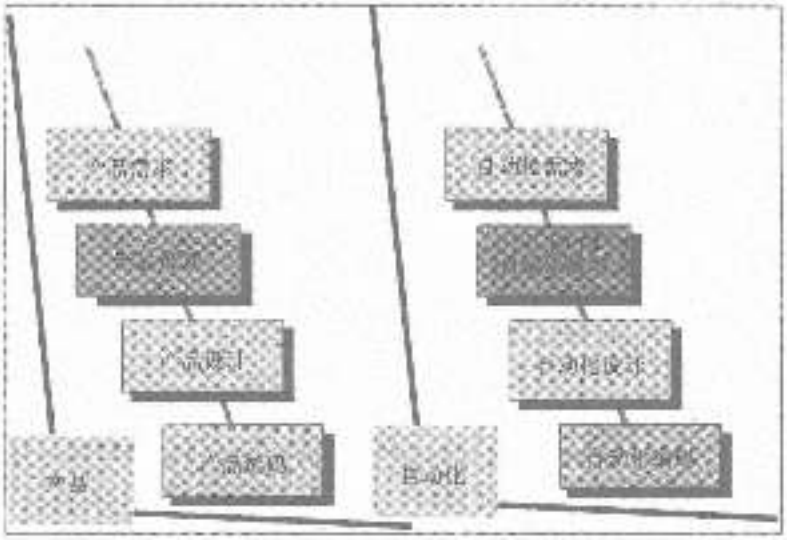


图16-4 产品开发与自动化之间的相似性

在测试产品时检验测试包很重要。如果测试包中有缺陷，就要花很大的精力搞清楚缺陷是来自产品还是来自测试包。第1章已经介绍过，自动化测试包在用于测试产品之前，首先要进行测试。如果测试包报告了错误的缺陷，就会严重影响测试自动化的信誉，影响下一步的自动化工作。因此，自动化测试包必须具有值得信任的质量。为了创建具有值得信任的质量的测试包，与产品开发一样，一些测试阶段也是很重要的。图16-5扩展了图16-4，引入了产品和自动化测试包的测试阶段。

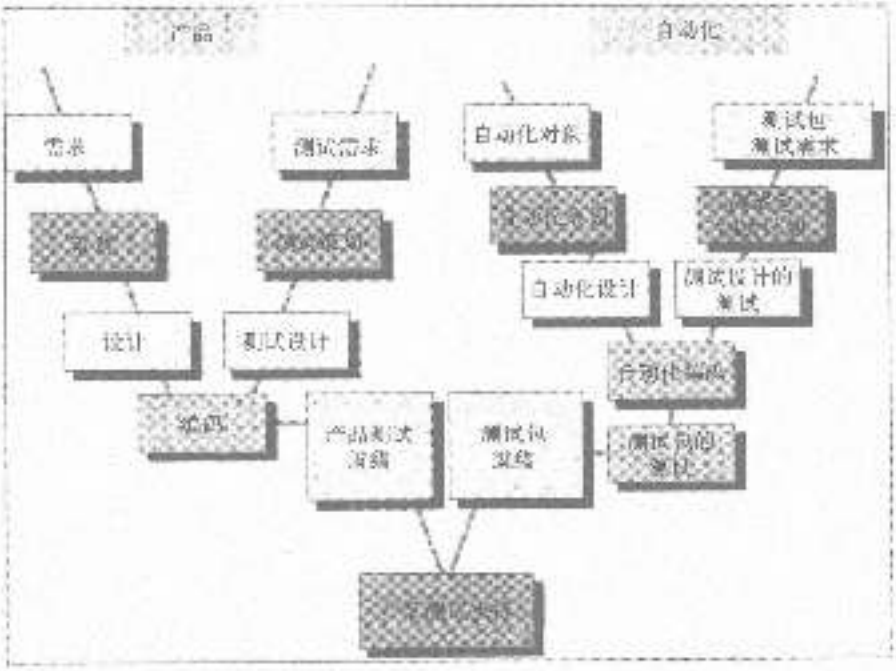


图16-5 W字模型——自动化包含的阶段

产品和自动化开发的每个阶段可以执行一组活动，包括四项。产品在需求阶段采集开发需求时，同时完成针对测试产品的测试需求、针对自动化开发的需求和针对自动化测试的需求。类似地，在策划和设计阶段也可以执行一组活动，也包括四项。针对产品和自动化的编码构成这种W字模型的编码阶段，这个阶段要交付产品和测试包。

产品和自动化就像铁路的两条钢轨，沿同一方向并行铺开，并具有类似的预期。

测试包构成自动化的一种可交付产品。此外，测试框架也可看作自动化的一种可交付产品，上一段讨论的测试阶段只针对测试包，因为测试包需要彻底测试，以便用于产品测试。

在产品和自动化中引入测试活动后，生命周期模型图中就包含了并行的两组开发活动和并行的两组测试活动。把这些活动合在一起就构成"W"字模型。因此，对于包含自动化的产品开发，遵循W字模型，既保证产品的质量，又保证所开发的测试包达到预期的质量要求，是一种很好的选择。

在讨论W字模型时，不能将其解释为出现在特定阶段内的所有活动都应该同时开始和结束。例如，图16-5中的"设计"、"测试设计"、"自动化设计"和"针对测试包的测试设计"都出现在同一个阶段（在图中并排给出）。针对产品和自动化的这些活动可以在不同时间开始和结束。W字模型只是保证活动的流程，并没有限定起止时间。产品开发和自动化可以有独立的进度计划，并作为两个不同的项目进行处理。

不需要同时开始和结束的另一个理由是，在很多公司中，要由同一个测试团队测试产品和开发测试包。在这种情况下，显然进度是不同的，活动的起止时间取决于由可用的资源和具体依赖关系决定的项目进度。

对于公司内有专门的自动化团队的情况，自动化进度可以独立于产品的发布。每次产品发布对应一些（经过测试的）可交付产品。这样，可以使用最新开发的测试包测试产品的当前发布版本。

16.8　测试工具的选择

确定了要自动化的对象后，一个相关的问题就是选择合适的自动化工具。尽管这里分成了两个顺序执行的步骤，但是这些步骤是密切关联的。

测试工具的选择是测试自动化的一个重要问题，这是因为：

1. 免费工具不能得到很好支持，很快就会被淘汰。由于测试工具的问题导致发布版本被推迟是极为危险的。

2. 开发内部工具很费时间。尽管内部工具会便宜一些并能更好地满足需要，但这类工具往往都是少数工程师凭兴趣开发的，通常文档写得很差，一旦开发工具的人离开公司，工具就不再能用。此外，在实际产品测试和发布期限的压力下，这种工具开发往往会往后放，因此人力不能得到保证。

3. 提供商销售的测试工具很昂贵。按绝对金额算，市场上的标准测试自动化工具是很昂贵的。很多公司，特别是中小企业，都会仔细评价作出这种重要投资的经济影响。

4. 测试工具需要很好的培训。除非使用工具的人经过恰当培训，否则测试自动化是不能成功的。这类培训通常包括熟悉随工具一起提供的脚本语言、工具的定制和为工具增加扩展件或插件。甚至经过这样的培训后，在使用工具上花费的时间也相当长。

5. 测试工具一般不能满足自动化的所有需求，因为工具要通用，因此很难充分满足特定客户的需要。这也是为什么定制和可扩展成为关键问题的原因。

6. 并不是所有工具都运行在所有平台上。为了降低自动化的成本，工具和自动化的测试脚本应能在被测产品运行的所有平台上重用。因此，工具和脚本对多平台的可移植性是选择测试自动化工具的一个关键因素。

由于以上原因，需要高度重视选择合适的自动化工具。前面已经介绍过，选择工具时理解自动化的需求是非常重要的。需求应该既包括短期需求，也包括长期需求。这些需求构成工具选择的基础。

16.8.1 选择测试工具的准则

上一节解释了评价测试工具的一些思路和需求采集对工具选择的作用。这些都会随着背景的不同而变化，不同的公司、不同的产品也会不同。下面讨论评价准则的大致分类。这些分类包括：

1. 满足需求；
2. 技术预期；
3. 培训与技能；
4. 管理问题。

满足需求

首先，市面上有很多工具，但是很少能够满足给定产品或给定公司的所有需求。针对不同的需求评价不同的产品需要付出大量人力、财力和时间。由于有太多的选择（每种选择都满足一部分需求），因此选择和引入测试工具会需要很长时间。

第二，测试工具通常落后一代，可能没有提供被测产品的后向或前向兼容性（例如JAVA SDK支持）。例如，如果产品使用了最新SDK版本，比方说版本2.0，测试工具所支持的SDK版本可能是1.2。在这种情况下，利用版本2.0编写的一些新特性就不能使用这种测试工具测试。测试工具跟上产品所用技术可能需要很长时间，这是产品公司不能忍受的。

第三，测试工具可能没有对新需求进行评价。例如，在2000年问题测试期间，有些工具就不能用于测试，因为工具本身也有与产品一样的问题（即不能正确处理日期）。与产品一样，测试工具也没有经过针对这些新需求的充分测试。第1章引用的成语“首先要测试测试用例”说得好，测试工具必须经过测试，验证其是否适合测试产品。

最后，有些测试工具不能区分产品失效和测试失效。这需要增加分析时间和人工测试。测试工具可能没有提供足够的定位、调试和错误消息，以帮助进行分析。这可能导致要在测试包中增加日志记录消息和审计，或导致人工进行测试用例检查。对于GUI应用程序的测试，测试工具可能根据运行时间的消息和屏幕排列判断测试结果。因此，如果产品的屏幕要素出现变化，就会要求测试包也随之变化。测试工具应有一定的智能，主动发现产品中出现的变更，并相应地分析测试结果。

技术预期

首先，测试工具一般不允许测试开发人员进行扩展和修改框架功能。因此，扩展功能需要由工具提供商完成，需要额外的成本和精力。测试工具可能没有提供与产品所提供的同样的SDK或扩展接口，市场上只有极少的工具提供源代码，用以扩展工具的功能或修正问题。可扩展性和可定制性是测试工具的一种重要预期。

第二，很多测试工具都要求把库与产品二进制代码连接在一起。当这些库与产品的源代码连接后，就叫做“插桩后的代码”。这使得在清除这些库之后还可以重复进行测试，成为一

定类型测试的结果在清除这些库后会发生变化和得到改善。例如，插桩后的代码对性能测试有重大影响，因为测试工具引入了额外的代码，执行这些代码会造成时延。

最后，测试工具不能100%地跨平台，只能支持一部分操作系统平台，这些工具生成的脚本可能在另外一些平台上不兼容。此外，很多测试工具只能测试产品，不能测试产品和测试工具对系统或网络的影响。如果要对产品对网络或系统的影响进行分析，首先怀疑的就是测试工具，在这种分析开始时要卸载测试工具。

培训技能

虽然测试工具需要大量培训，但是很少有提供商能够提供所需级别的培训。需要公司级的培训，以部署该测试工具，因为测试包的用户不仅是测试团队，还有开发团队和其他人员，例如配置管理。测试工具期望用户学习新的语言或脚本，并可能不使用标准的语言或脚本。这提高了自动化的技能需求，延长了公司内部的培训时间。

管理问题

测试工具增加了系统需求，需要升级硬件和软件。这会提高已经就很高的测试工具成本。在选择测试工具时，注意对系统的需求很重要，升级软件和硬件所涉及的成本需要包含在工具成本中。从一个工具到另一个工具的迁移可能很难，需要做大量工作，不仅是由于所编写的测试包不能供其他测试工具使用，而且还由于迁移所需的成本很高。因为工具很昂贵，除非管理层认为投入回报是值得的，否则一般是不会批准换工具的。

部署测试工具所需的工作量与在公司中部署产品的工作量一样。但是，由于项目进度压力，部署测试工具的工作会被淡化、被取消。以后，这又会成为产品发布拖延，或自动化不能达到预期要求的理由。在选择和部署测试工具时，对工具的支持是另一个需要考虑的问题。

表16-3归纳了以上讨论的内容。

表16-3　选择测试工具中的问题

满足需求	技术预期	培训与技能	管理问题
检查工具是否满足需求，包括工作量和经费	扩展测试工具是很困难的	很少测试工具供培训教程	测试工具需要系统升级
测试工具不能与产品完全兼容	有些测试需要删除插桩后的代码	测试工具要求用工学习新的语言或脚本	迁移到其他测试工具困难
测试工具没有针对部署需求进行过与产品一样的场景测试	测试工具不是跨平台的		部署工具需要大量预算和人力投入
增加了分产品问题和测试包问题，产品变更会要求变更测试包			

16.8.2　工具选择与部署步骤

本节要根据本章前面的讨论，提出选择和在公司中部署测试工具的七个简单步骤：

工具具有很高的引入、维护和退出成本，因此需要精心选择。

1. 在所讨论过的一般需求中要取测试包需求，补充其他需求（如果有的话）；
2. 保证已经考虑过前几节所讨论的经验；
3. 收集其他公司使用类似测试工具的经验；

4. 准备向提供商提出的有关成本、工作量和支持问题的检查单；
5. 列出满足以上需求的工具（优先列出提供源代码的工具）；
6. 评价并最后确定一个或一组工具，并对所有测试开发人员提供工具培训；
7. 培训工具的所有潜在用户后，为各个测试团队部署该工具。

16.9 极限编程模型的自动化

第10章已经讨论过，极限编程模型的基础是以下基本概念：

1. 单元测试用例要在编码阶段开始前开发；
2. 要针对测试用例编写代码，并保证测试用例通过；
3. 每次所有单元测试都要100%地运行；
4. 每个人都拥有该产品，员工之间经常跨越边界。

极限编程模型的以上概念使自动化成为产品开发的一个有机组成部分。为此，需要特别讨论极限编程模型。极限编程中的自动化不应看作是额外活动，得到产品需求的同时也就得到了自动化需求，这使自动化从产品开发的第一天起就启动了。

在极限编程中，测试用例要在编码阶段开始前编写。开发人员编写代码要保证这些测试用例通过。这使得代码和测试用例始终保持同步，这还使针对产品编写的代码能够被自动化重用。此外，由于代码的目标是保证测试用例通过，因此，开发人员在编写代码时会自己开发自动化的测试用例。

在极限编程中，所有单元测试用例每次都要以100%的通过率运行。这个目标使自动化具有额外的重要性，因为没有自动化，就不能每次使代码都能满足该目标。

开发技能和自动化编码技能之间的差距对于遵循极限编程模型的团队来说不成问题，因此，员工越过边界承担开发人员、测试人员等各种角色，在极限编程模型中，开发、设计和自动化技能并不是互不联系的。

在极限编程模型中，以上概念使自动化成为产品开发周期的一个有机的组成部分。

16.10 自动化中的挑战

通过以上讨论可以推论，测试自动化带来了一些特有的挑战。其中最重要的挑战要算是管理层的承诺。

第1章已经讨论过，自动化不能看作是所有问题的万能药，也不能看作是产品所有质量问题的快速解决方案。自动化需要时间和精力，长期坚持才能得到回报。但是，自动化需要相当大的初始资金投入，测试工程师需要经过很长时间的学习才能开始得到回报。管理层应该有耐心，并坚持自动化。这里的主要挑战是因为测试自动化需要很大的前期投入，而管理层寻求的是尽早得到回报。成功的测试自动化努力都有坚定的管理层承诺、明确的长远目标以及有能力确定不断向长期目标推进的近期目标等特点。如果管理团队没有这些特点，那么很可能自动化会中途夭折，更糟糕的是，人们可能对自动化开始存在错误认识，就像第1章中提到的农夫一样。

16.11 小结

第1章"首先测试测试用例"已经讨论了测试测试包的重要性。在把测试包用于测试产品

之前，要给当地测试测试包，并确保其满足对质量的预期。正如前面所讨论过的，如果对测试包的质量有怀疑，那么在确定究竟缺陷是源自测试包还是产品，就会花费精力并产生争论。为了打消这种怀疑，保证测试的平稳运行，测试包的所有质量需求和所遵循的过程步骤，都需要与产品开发的相同。

测试包的质量需求与产品的质量需求相当或更严格。

有人认为自动化意味着自动化测试执行的过程，自动化的范围应该扩展到产品开发所包括的所有阶段，涉及公司中的多个人员和多个角色。测试执行是自动化开始的地方，因为在测试执行阶段有很多重复性工作，并且手工工作更多一些。自动化并不止步于只是对测试用例执行的自动化。测试包需要与其他工具和活动（例如，测试用例数据库、缺陷过滤、自动发送有关测试和版本构建状态的电子邮件、自动生成报告等）关联起来，以提高有效性。说到自动化，并不是一切都可以自动化，100%的自动化并不总是行得通的。把一些测试用例自动化的测试包与富有创造力的人工测试结合起来，会使测试更有效。自动化不能止步于记录和回放用户命令。自动化测试包应该有足够智能，以确定预期的是什么，测试用例为什么没有通过、以及给出手工步骤以重现问题。自动化不能只用于节省工作量，还应该用于提高覆盖率，并提高可重复性。有时需要修改已经形成文档的手工测试用例，以适应自动化的需求。有自动化的地方，文档不需要很详细的描述，因为通过查阅自动化脚本就可以随时导出测试用例的步骤。

好的自动化有助于24×7地执行测试，从而节省人力和时间。

自动化的进展不仅会受到技术复杂性的阻碍，而且还会受到缺少人力资源的阻碍。即使分配了人力资源，在产品接近发布或进入关键状态时，也会被抽走。这种紧急措施是可以理解的，因为按时发布产品、兑现对客户作出的承诺与自动化相比，应该具有更高的优先级。但是，如果这种情况反复出现对于自动化是不利的。由公司中的一支独立团队承担研究自动化需求、工具评价和开发通用测试包等任务，会增加更多价值。独立的自动化团队可以实现公司长期的自动化目标。

随着测试自动化变得比产品开发还要复杂，自动化团队需要规划，以得到良好的开发和测试工程师。

工具的使用、管理和专业化是一种职业，有关测试工具的使用已经有认证课程，在自动化方面已经有很多职业机会，因此，把自动化看作是一种开发人员和测试人员可以成长的重要领域和岗位是很重要的。

自动化不能看作是当测试工程师有空闲时间时，让他们别闲着的权宜之计（"如果没有要执行的测试，就去实现测试自动化吧！"），使自动化更有用需要更宏的视野。需要制定计划维护测试包，就像产品维护一样，自动化应该看作是一种产品，而不是一个项目，就像产品路线图一样，自动化也应该有路线图如旁边的"空域图"所示）。

没有经过适当分析就选择测试工具，会导致把昂贵的测试工具束之高阁。可以把这种工具当做摆设架件。

选择合适的测试工具只是自动化成功的一个部分，另外需要考虑的因素是时机和测试包要如何以所需的质量满足测试需求。因此，测试包的开发要在测试执行开始前交付，针对被测产品的自动化目标应该是在产品发布和结束开发周期前交付。否则，只是自动化获得成功就会像谚语所说的，"手术虽然成功但病人却死了"。成功的自动化要在满足产品需求的同时，满足自动化需求。

讨论了自动化取得成功的各种因素后，该是讨论一些失败因素的时候了。一般来说，一

些调查和测试专著都指出自动化失败的一个原因是没有拥有关系。这种观点有以下事实支持：30%的测试工具都是库架件，第1章在讨论农夫抱怨自动化作物栽培失败时已经提到过"自动生综合症"。自动化本身没有失败，需要注意的是围绕自动化的各种观念。例如，有人认为自动生会减少公司的员工。自动化不会减少员工，而是使测试工程师能够将注意力放在直接与产品质量有关的更具创造性的任务上。

> 有一份公路图对于飞翔没有什么帮助，飞翔并达到目的需要"空域图"和"路线图"。

自动化使测试人员的工作更轻松，可以更好地重现测试结果，提高覆盖率，当然作为一种副产品还能降低工作量。自动化应该看作是产品开发周期的一部分，应该作为产品和公司成功的基本习惯保留。最后，通过自动化，可以得到更好、更有效的指标，能够以定量的方式帮助理解产品的健康状况，这就引出下一章的内容。

问题与练习

1. 本章和前面各章都提到了端到端的测试自动化，请考虑以下每种情况的自动化需求并评价工具的选项：
   a. 根据需求、设计和程序规格说明，设计测试用例；
   b. 测试数据的生成；
   c. 针对给定发布版本，根据代码变更情况选择测试用例；
   d. 自动分析测试正确性的工具；
   e. 自动分配合适的员工通过测试发现缺陷的工具；
   f. 性能测试工具；
   g. 测试报告生成工具。
2. 请设计一种简单模式，在测试用例数据库、配置管理工具和测试历史之间进行映射，以完成问题1列出的一些自动化需求。
3. 白盒测试的哪些方面适合自动化？需要什么工具？
4. 自动化测试应用程序中的GUI部分会遇到哪些挑战？这与后端测试自动化有什么差异？
5. 本书最后一章将讨论指标。研究所给出的各种指标，并说明在问题2中给出的模式可以用于生成这些指标。请给出采用例如SQL写出的能够生成以上指标的查询语句。
6. 请考虑编写自动化的测试用例测试数据库的各种查询选项。测试脚本需要什么类型的初始化和清理？
7. 在使用自动化脚本时，请导出可以用来实现和促进重用的标准。
8. 本章提到过不要在测试用例中采用硬编码的方式写入任何取值。还给出了一个例子说明不同层次的软件有不同的配置参数。请归纳维护这种保存不同层次参数的配置文件所面临的问题。如何解决这些问题？
9. 测试基于Web应用程序的一个共同问题是出现弹出窗口、广告等。请利用测试自动化工具已有的特性将应用程序与这些中断隔离。
10. 本章使用元语言描述了测试自动化需求，以和元语言还需要什么其他特性辅助实际的软件测试？

第17章 测试指标和度量

17.1 指标和度量的定义

这个时期我们注意到公司取得了很好的效益，而且，老板那时正在休假！

项目中的所有重大活动都要进行跟踪，以确保该项目按计划进行，同时可以决定是否采取纠正行动。关键参数的度量是跟踪不可缺的一部分。度量首先需要采集一些数据，但是，原始数据本身可能没有揭示为什么会发生特定的事件，所收集到的数据都必须综合分析以得出适当的结论。在上面的卡通画中，两个数据点分别表示，老板在度假和效益在过去的一个季度中飞速增长。但是，（希望如此，）这两个事件没有直接的联系。因此，从原始数据中得出的结论对决策是没有用处的。

不能度量就没有办法管理。

指标从原始数据中导出对决策有帮助的信息。这类信息对于理解以下内容会有帮助：

1. 数据点之间的关系；
2. 被观察数据点之间可能存在的因果关系；
3. 数据如何用于未来规划和持续不断的改进。

因此，指标是使用合适的公式或计算方法从度量中导出的。显然，相同的一组度量可以产生针对不同人员需要的不同指标。

从以上讨论可以看出，为了对项目的执行情况进行跟踪并对其进度进行有效监测，需要：

1. 必须度量合适的参数，这些参数要与产品或过程有关；
2. 必须对所度量的数据进行恰当分析，以得出关于项目组或公司内产品或过程健康状况的正确结论。
3. 分析的结果必须以适当的形式提供给利益相关各方，使他们能够对提高产品或过程质量（或任何其他相关的业务推动）做出恰当的决策。

由于本书焦点在于测试及被测产品，因此本章只讨论与测试及产品相关的指标，不考虑过程改进的方法。

当把孤立的数据点放在一起时，指标及指标的分析可以传递有关原因的信息。通过图表，图形的形式表示相关的数据点及其关系，可以简化分析工作，使于利用指标进行决策。指标程序的目的就是精确地达到上述目标，这也是本章的关注点。以下首先定义本章要使用的一些基本术语。

投入是用在特定活动或阶段的实际时间。逝去时间是从活动开始到活动结束之间的时间，例如，通过网络预订一种产品要花五分钟的投入和三天的逝去时间，人们预定时间很短，但是包装和运输等需要用较长的时间，但在这个计划中，潜在因素或耽搁必须记为三天。当然，

在这三天里，预定产品的人可以同时完成一些其他活动，一般来说，投入要从生产力数据导出，而逝去时间是要求完成一组活动的总天数，完成一系列活动的逝去时间构成项目的计划。

搜集和分析指标包含投入和其他步骤。这些步骤如图17-1所示。指标程序的第一步包括决定哪些度量数据是重要的，同时搜集这些相关的数据。用在测试上的投入、缺陷数、测试用例数都是度量的例子。根据使用的数据目的不同，度量的粒度会随之变化。

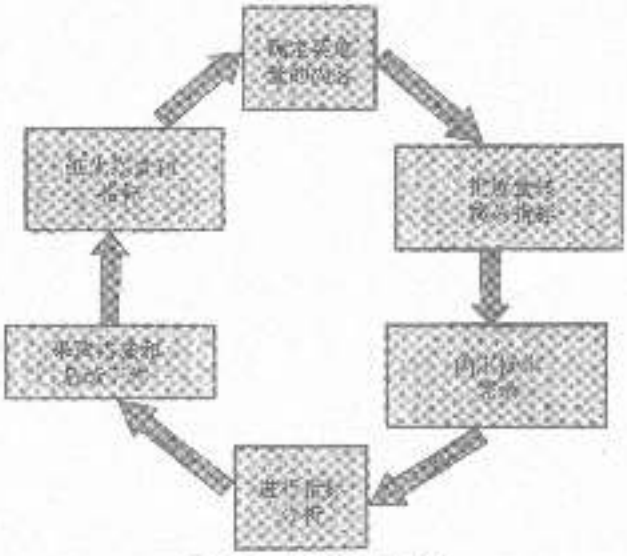


图17-1 指标程序的步骤

在决定度量内容时，需要注意以下方面：

1. 度量的内容应与要达到的目的相关。对于测试来说，显然我们感兴趣的是用在测试上的投入、测试用例的数量和通过测试用例报告的缺陷数目等。

2. 度量的实体应该是自然的，且不应带来过多的负担。如果度量的负担太大，或者测量没有自然地与实际完成的工作一致，那么提供数据的人就可能不愿意提供度量数据（甚至提供错误的数据）。

3. 度量的内容应在满足度量目标的情况下，选择合适的粒度水平。

让我们仔细看一下这最后一点：关于数据粒度的描述。不同的人使用度量数据可能会在不同的方面得出不同的推论。获得数据的颗粒度水平取决于特定人群对细节的要求水平。类比，指标及其来源必须针对不同的人群设定不同的标准。获得颗粒度细节的方法叫数据挖掘。后面给出了一个数据挖掘的练习。这就是发生在许多公司中的典型情况，提交的指标/测试报告显示不同粒度的数据，数据粒度取决于决策的不同层次。

测试人员：我们在这个测试周期定义发现了100个新缺陷。

经理：这些新缺陷都是哪个方面的？

测试人员：可使用性缺陷占了100个缺陷中的60个。

经理：哦！产品的什么组件又产生这么多缺陷？

测试人员：60个缺陷中的40个是由“数据迁移”组件产生的。

经理：是哪个具体的产品功能有这么多缺陷？

测试人员：40个缺陷中的35个是由包含不同模式的数据迁移产生的。

例子中的对话继续进行，直到全部问题都被回答，或关注的缺陷数目变得非常小，且可以追溯到根源。数据挖掘发生的深度取决于讨论或者需要关注的领域。因此，重要的是尽可能多地提供度量粒度。在上述例子中，度量的内容是"缺陷的数目"。

人们沟通时并不会像这个例子一样只涉及一个度量。本章稍后将介绍一些可以组合，以产生指标的度量。有关多个度量的一个典型问题是"在涉及不同模式的数据迁移里，多少个测试用例发现了40个缺陷？"这个问题包含两个度量：测试用例的数目和缺陷的数目。因此，指标收集的第一步骤是定义如何组合数据点或度量，从而提供有用的指标。一个具体的指标可以使用一种或多种度量。

了解了度量使用的方法和度量的粒度，就可以进行指标程序的第三步——决定度量的操作要求。制定指标计划的操作要求时，不仅考虑数据采集周期，而且考虑其他操作问题，譬如，谁应该采集度量数据、分析报告应该给谁等。这一步有助于为度量决定适当的周期，同时确定收集、记录和报告度量，并分发指标信息的操作职责。一些度量数据需要每天采集（例如，执行了多少个测试用例，发现并修改了多少缺陷等）。但包含上述类似问题的指标（"多少测试用例产生40个缺陷"）是在一段时间内需要监控的指标类型，这段时间可以是一周，也可以是到测试周期结束。因此，策划指标生成也要考虑指标的周期性。

指标程序的第四步是分析指标，以识别产品质量中的积极部分和待改善部分。通常，只关注指标提示的待改善问题，但赞誉和坚持产品的积极方面同样重要。这将确保最佳实践制度化，同时激励团队做得更好。

指标策划的最后一步是采取必要措施，并把行动贯彻到底。如果一个行动项目直到完成都没有遵守标准，那么指标程序的目的就没有达到。对于发布前最后一个阶段的测试来说，尤其是这样，分析及完成行动的任何耽误都会导致产品发布不必要地延误。

以上介绍的任何指标程序都是一个持续不断的过程。我们进行度量，把度量转换为指标，分析指标，采取纠正措施，其中摆在首位的就是进行度量。然后，继续重复进行下一轮指标程序，采集（可能的话）不同的度量，从而针对处理问题的不同（可能的话），导出更精确的指标，如图17-1所示。指标程序不断地经历上述不同度量或指标的步骤。

17.2　测试中指标的意义

由于测试是产品发布前的最后一个阶段，因此，很有必要度量测试进展和产品质量。跟踪测试进展和产品质量可以对发布版本有一个很好的认识，即发布版本是否及时达到所要求的质量。因此，有关确定测试进展的度量和所产生的指标就变得非常重要。

仅仅了解测试完成了多少，还不能回答什么时候完成测试和什么时候产品做好发布的准备这样的问题。要回答这样的问题，需要知道测试还需要多少时间。判断测试需要剩余的天数，需要两个数据点——尚未执行的测试用例数及每天去一天可执行的测试用例数。每人每天可执行的测试用例数基于一个叫做测试用例执行生产率的度量。生产率数值来源于以前的测试周期。边栏中是计算完成测试所需天数的计算公式。

完成测试需要的天数
=尚未执行的测试用例数/测试用例执行生产率

因此，指标需要知道测试用例执行生产率以估算测试完成日期。

不是只有测试能决定产品的发布日期，需要修改全部严重缺陷的天数也是一个关键的数据。缺陷修改所需天数需要考虑"待修改的严重缺陷"及"未来的测试周期将会发现多少个潜在的缺陷"的估计。收集了一段时间的缺陷趋势给出了一个粗略的估计。未来的测试周期

将出现多少缺陷。因此，指标有助于预测未来测试周期将会发现的缺陷数。

搜集了一个时期的缺陷修复趋势给出了团队的另一个估算值：缺陷修复能力。这个度量给出了开发团队在一段特别的时间内可以修复的缺陷数。结合缺陷修复能力的预测可以估算出发布产品需要的天数。边栏中给出的公式可以粗略估算缺陷修复需要的总天数。

因此，指标有助于估算缺陷修复需要的总天数。一旦测试时间和缺陷修复时间已知，发布日期就可以估算出来。测试和缺陷修复是可以同时进行的活动，只要回归测试计划已证实严重缺陷被修复同时不带来任何副作用。如果一个产品研发团队遵循开发团队和测试团队分离的模式，则发布日期取决于测试所需时间和缺陷修复所需时间哪一个更紧迫。边栏中给出的公式可计算发布日期。

在正常的测试周期结束后可以开始缺陷修复。在产品发布前这些缺陷的修复将由回归测试来验证。因此，边栏关于发布日期的公式需要修改。

由于现在的公式不包括各种其他活动（例如文档编制、会议等），上述公式可以进一步调整，以便提供更准确的估计。在这里讨论公式的构思是为了说明指标非常重要，它有助于得到产品的发布日期。

修复缺陷需要的总天数=（尚未修复的严重缺陷数+未来测试周期发现的缺陷数）/缺陷修复能力

发布需要的天数=最大值（测试所需天数，缺陷修复所需天数）

发布需要的天数=最大值（测试所需天数，（缺陷修复所需天数+回归测试修复严重缺陷所需天数））

在开发和测试周期内搜集的度量不仅可以用于产品发布，还可以用于发布后的活动。查看一段时期的缺陷趋势有助于得到近似的估计：发布后可能报告的缺陷数。这个缺陷趋势可以用来作为增加维护/支持团队的人数的参数之一，以解决发布后的缺陷问题。了解发布周期内发现的缺陷类型并且知道所有严重缺陷及其影响，有利于售后人员的培训，确保他们针对用户可能提出的缺陷报告做好充分准备。

指标不仅用于被动活动，指标及其分析还有助于主动预防缺陷，从而节省成本和人力。例如，如果一种类型的缺陷（比如编码缺陷）报告的数目较大，明智之举是执行代码审查以阻止这些缺陷的发生，而不是一个一个地寻找并修复这些代码，指标有助于发现这些机会。

指标用于人力资源管理，以发现合适的产品开发团队规模。由于人力资源管理是产品开发和维护的重要方面，因此指标在这个领域有很大作为。

还有其他方面的指标也很有用，例如测试用例发现缺陷的能力。在第8章讨论了测试用例的历史记录，这些历史记录结合项目的指标，可提供在当前周期里，哪些测试用例发现的缺陷多，哪些发现的少的详细信息。

总结一下，测试的指标有助于确定：

- 何时发布产品。
- 发布什么——根据各个模块的缺陷密度（稍后将给出缺陷密度的正式定义）、对顾客的重要程度以及这些缺陷的影响程度，可以决定按原计划发布产品时要发布的产品范围。指标有助于做出这种决定。
- 产品是否能以已知的质量发布——指标的作用不仅能确定发布产品的日期，而且能了解产品的质量，确认是否能按已知质量发布，还能确认产品能否在其应用领域内按预期的方式发挥作用。

17.3　指标类型

指标按照度量的内容和关注的领域不同可以分为不同的类型。从大的方面讲，指标可以分为产品指标和过程指标。如上所述，本章不讨论过程指标。

产品指标可细分为：

1. **项目指标**　说明项目是如何策划和执行的一组指标。

2. **进度指标**　跟踪项目中不同的活动是如何进行的一组指标。这些活动包括开发活动和测试活动。由于本书关注的是测试，因此本章只讨论适用于测试活动的指标。进度指标监控整个测试阶段，有助于查明测试活动的状态，同时也是产品质量的很好的指示器。测试出来的缺陷提供了丰富的信息，这些信息有助于开发团队及测试团队进行分析和改进。由于上述原因，本章的进度指标只关注缺陷。为了方便论述，进度指标可进一步分为测试缺陷指标和开发缺陷指标。

3. **生产力指标**　有关生产力数值的一组指标。采集这些指标可用于策划和跟踪测试活动。这些指标有助于策划和估计测试活动。

图17-2给出了上面讨论的所有类型的指标，以及本章将要讨论的具体指标。

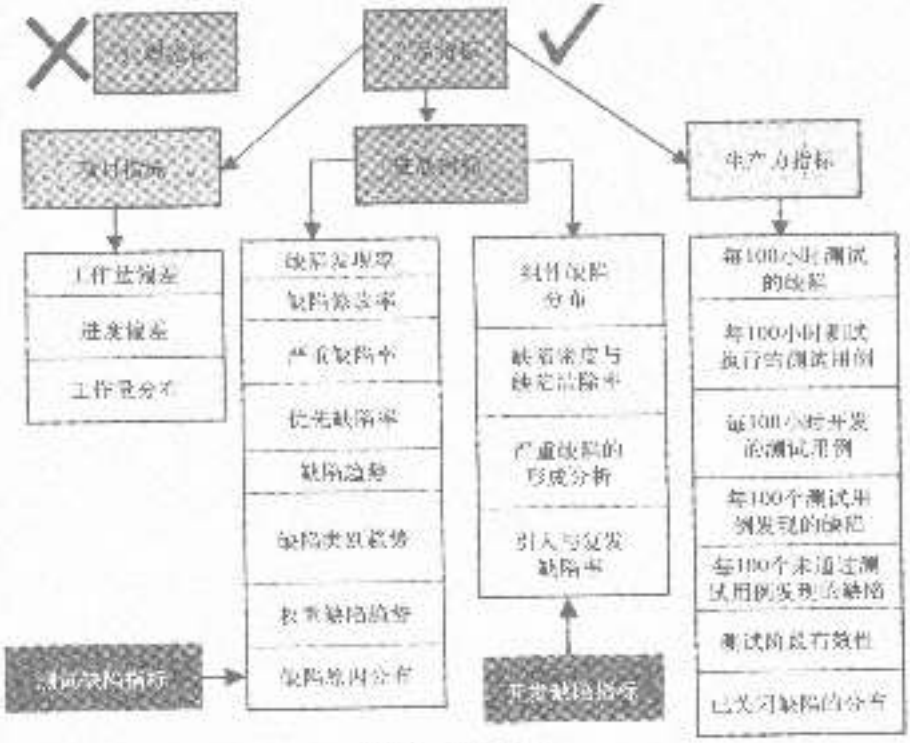


图17-2　指标类型

17.4　项目指标

典型的项目从需求获取开始到产品发布结束。这两点之间所有的阶段都需要策划和跟踪。在策划阶段，需要确定项目的范围，项目的范围转化为估算的工作量，得出要完成的工作。通过使用现有的生产力数据可将所估算的工作量转换为每个阶段和活动的投入估计。投入的

初值称为基准投入。

随着项目的进行，如果项目的范围改变或现有的生产力数据不正确，那么工作量估计就需要重估，重估的投入估计称为修订投入。根据需求及影响投入的其他参数的改变频率，估计也随之改变。

对于很多公司来说，投入的高估或低估是很正常的现象。错误估计的危险在于经济上的损失，延期发布也会在客户中留下不好的品牌形象。正确的估计来自于经验及团队的正确生产力数据。

投入和进度计划是跟踪任何阶段或活动的两个因素。跟踪软件开发生存周期阶段的活动有两个方法：投入和进度计划。在理想情况下，如果紧密跟踪投入并满足投入，那么进度计划也可以达到。通过向项目中添加更多的投入（额外的资源或请工程师加班）也可以实现进度计划。如果通过增加更多的投入来保证满足发布日期（也就是进度计划），那么项目策划和执行就被认为是不成功的。事实上，这种添加投入的想法并不总是可行的，因为公司不是每次都有可利用的资源，而且工程师加班超过一定程度也不会增加生产率。

同时，即使计划投入和实际投入是相同的，但计划没有满足，那么项目也是不成功的。因此，好的主意就是在项目指标里对投入和计划都进行跟踪。

非常自然，可直接捕获并形成指标输入的基本度量是：

1. 不同的活动以及最初的基准投入和进度计划，即项目或阶段开始的输入。
2. 各种活动的实际投入和时间，即活动过程中的输入。
3. 经过修改的投入和进度计划估计，即在项目过程中的合适时间进行重新计算。

17.4.1 投入偏差（计划投入与实际投入）

如果把软件生存周期所有阶段的基准投入估计、修正后的投入估计和实际投入都画在一起，可以得到有关估计过程的很多启示。因为不同阶段会涉及不同的人，因此最好分阶段画出这些投入的数值。通常在修正估算或到产品最后的发布日期时绘制这些偏差图。图17-3给出的是每个阶段的样例数据。

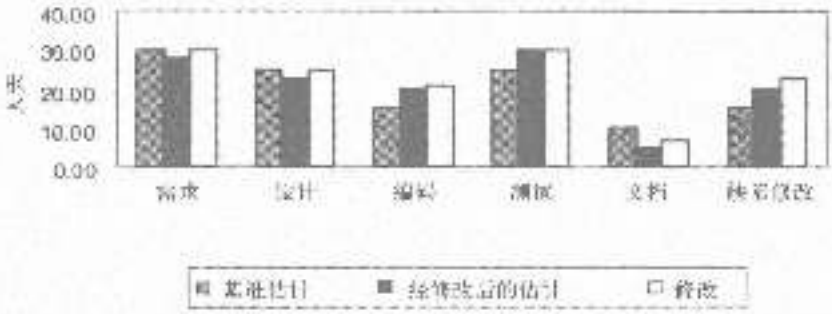


图17-3 分阶段的投入偏差

如果在基准投入和修正投入之间有很大差别，则说明初始估计不正确。计算每个阶段的投入偏差（如下面的公式计算）在修正投入和实际投入之间提供一个定量的偏差度量。

偏差%=[(实际投入-修正估计)/修正估计]*100

如果偏差只考虑修正估计和实际投入，那么问题出现了：基准估计还有什么用呢？正如前面提到的，投入偏差图为估计过程提供了输入。当估计错误（或正确）时，重要的是找出错误（正确）将在那里

发生。很多匆忙的修正估计是为了快速地对需求改变或不清楚的需求作出响应。如果是这种情况，偏差计算的合适参数就是基准估计。此时分析应该指出修正估计过程中存在的问题。类似地，基准估计过程中由于计算偏差也会带来问题。因此，每个阶段的基准估计、修正估计和实际投入应画在一起，偏差的统一表理显示在表17-1中。

表17-1　每个阶段样例偏差的百分比

投入	需求	设计	编码	测试	文档化	缺陷修复
偏差%	7.1	8.7	5	0	40	15

在软件开发生存周期的任何阶段，超过5%的偏差表明估计需要进行改进。在表17-1里，可接受的偏差只有编码和测试阶段。

偏差也会是负。负偏差表明高估。偏差数值连同分析有助于更好地估算下一次发布，或下一个修正估计周期。

17.4.2　计划偏差（计划与实际）

大多数软件项目不仅关注投入偏差，还关注进度计划的符合程度。这样就提出进度计划偏差指标。像投入偏差一样，进度计划偏差指的是实际进度和估计进度计划之间的差，但有一点不同。取决于项目所使用的软件开发生存周期模型，多个阶段可能同时进行，即软件开发生存周期中的不同阶段是相互重叠的，且共用一些相同人员。由于存在这种复杂情况，进度计划偏差只在项目层面的里程碑点综合计算，而不在软件开发生命周期的每个阶段分别计算。图17-4提供了绘制进度计划偏差的方法。

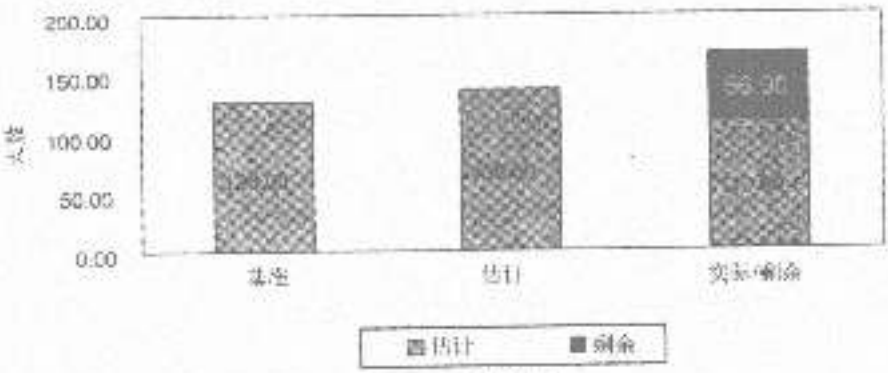


图17-4　计划偏差

使用图17-4中的数据，参考估计进度计划和实际进度，用上一节介绍的类似公式可以计算出偏差的百分比。

在每个里程碑点结束时计算进度计划偏差，了解项目遵照进度计划进展得如何。为了在项目进行过程中得到进度的真实情况，重要的是计算项目的“还剩余花费的天数”，并将这一数据同“实际计划天数”放在一起，如图17-4所示。把所有剩余活动加在一起就可以得到“还剩余花费的天数”。如果这一数据没有计算并标注出来，那么它对于项目进展中的图来说就没有任何价值，因为无法直观地从图中推测出偏差。当发布日期到达时，计划中的剩余天数就会变为零。

投入和进度计划偏差应该全面而不是孤立地进行分析。这是因为，投入是决定成本的一

个主要因素，而计划决定了产品能够抓住市场机遇的程度。偏差可以分为负偏差、零偏差、可接受偏差及不可接受偏差。通常情况下，0~5%属于可接受偏差的范围。表17-2给出了特定的场景、可能的原因以及结论。

表17-2 投入和进度计划偏差范围的解释

投入偏差	计划偏差	可能的原因/结果
零或可接受偏差	零偏差	好的执行项目
零或可接受偏差	可接受偏差	需要稍微改进投入或进度计划估算
不可接受正偏差	零或可接受正偏差	混乱，需要更进一步分析
不可接受负偏差	不可接受负偏差	投入和进度计划都被低估
负偏差	零或可接受负偏差	高估和拨时，投入和进度计划都需要改进
负偏差	负偏差	高估和投前，投入和进度计划都需要改进

尽管表17-2提供了各种情况下的可能原因和结果，但是它无法反映出所有的可能原因和结果。举例来说，一个阶段或模块的负偏差会抵消产品模块另一个阶段的正偏差。因此，应该研究指标中的"为什么和怎么做"，而不是仅仅把注意力放在实现了"是什么"上。前面讨论到的数据挖掘练习会对这种分析有所帮助。下面列出一些分析投入和阶段计划偏差时应该提出的典型问题。

- 投入偏差的出现是因为最初糟糕的估计还是糟糕的执行？
- 如果最初的估计被证明是不合适的，是不是缺乏做出好的估计所需的支持数据？
- 如果某些投入和进度计划与预先估计的不符，是什么原因导致了偏差？所测试的项目是不是出现了技术变化？是不是引入了新的测试工具？某些关键人物是不是离开了团队？
- 如果投入符合而进度计划不符合，那么对这个进度计划，是不是并行进行了合适的考虑？是否探索了合适的资源复用？
- 可否改进某些进程或者工具以改善并行度，并以此加快计划的实施？
- 一旦在投入和进度计划中遇到负偏差（也就是说，完成项目比原定的投入少，进度快），我们是否知道是什么提高了效率？如果知道，那么能否把这种效率制度化，以获得持续改进？

17.4.3 不同阶段内的投入分布

偏差计算有助于发现承诺是否及时兑现，以及估计方法是否切实有效。此外，如果能够获得并分析各个阶段的投入分布，那么就能得到有关产品质量的一些指示。例如：

> 为了得到高质量的产品发布，在软件开发生存周期的每一阶段都需要足够和合适的投入。

- 在需求上的投入很少就会导致频繁的需求变动，但也要为开发阶段和测试阶段留下足够的时间。
- 在测试上投入不足会导致在用户场地突然出现问题，但在测试上的投入比实际需要多又会使产品失去市场。

不同阶段的分布百分比在策划的时候就可以进行估计，这些数据可以在发布的时候与实际数据相比较，以便对发布和估计方法更有信心。图17-5给出了各阶段投入分布的例子。

成熟的公司至少会在需求阶段花费总投入的10%~15%，在设计阶段也会花费差不多同样的投入。测试投入比率取决于发布版本的类型和现有代码和功能的改变量。通常来说，公司

在测试上会花费总投入的20%~50%。

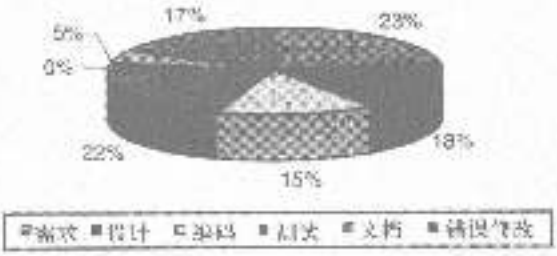


图17.5　实际投入分布

17.5　进度指标

任何项目都要从两个角度进行追踪。第一，项目在投入和进度计划方面做得如何。这也是本章到目前为止研究的问题。另一个同样重要的角度，是找出产品的发布版本达到质量需求的程度。按时并且投入也在估计偏差之内，但是产品的发布版本带有许多缺陷，没有什么意义，只会使产品无法使用。测试的一个主要目的，是在客户发现缺陷之前尽可能多地将其找出。产品中发现缺陷的数量是重要的质量指标，因此本节要讨论反映产品缺陷（因此是质量）的进展指标。

测试团队发现问题，而修复问题则要靠开发团队。根据这一思想，缺陷指标进一步分为测试缺陷指标（有助于测试团队分析产品质量和测试）和开发缺陷指标（有助于开发团队分析开发活动）。

已经发现了多少缺陷和将会发现多少潜在缺陷，这两个参数决定了产品质量和对产品质量的评估。对这种评估来说，测试的进展是能够了解的。如果只完成了50%的测试就发现了100个缺陷，假设这些缺陷均匀地分布于产品之中（同时保持其他参数相同），那么另外80~100个缺陷可以估计为残留缺陷。图17.6显示了通过标注测试执行情况和结果表示测试进展，参见彩图。

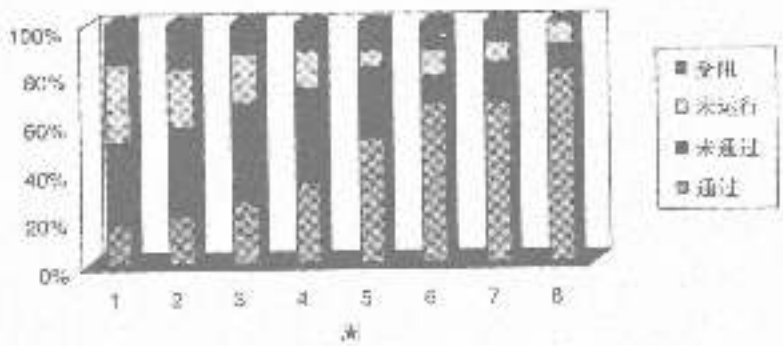


图17.6　测试用例执行进展

这一进展图给出了已执行测试用例的通过率和未通过率、挂起的测试用例以及等待修复缺陷的测试用例。以这种方式显示测试进度，可以更容易地理解测试状态并为进一步分析打下基础。在图17.6中，“未运行”的测试用例每周都在递减，这表示测试用例正在运行。从图中还能看到，通过百分比增加而未通过百分比减少，说明测试和产品质量的进展是积极的。图中受阻特定测试用例执行的缺陷也每周都在递减。因此，这样的进展图表明不仅测试进展

是好的，而且产品质量也在提高（也就是说测试是有效的）。从另一方面来说，如果进展图显示随着周数的增加，“未运行”用例的数目并没有减少，或是“失败”用例数量增多，“通过”用例数量没有增加，那么就清楚地表明产品有质量问题，并会阻碍产品的发布。

17.5.1 测试缺陷指标

上一节讨论的测试进度指标获得随时间推进发现缺陷的进度。下面的指标有助于了解如何利用所发现的缺陷改进测试和产品质量。不是所有的缺陷都具有相同的后果和重要性。一些公司通过指派缺陷优先级来进行分类（例如P1，P2，P3等）。根据缺陷的优先级可以排列出缺陷修复的先后顺序。例如，一个P1级的缺陷要比P2级的缺陷优先得到修复。一些公司使用缺陷严重程度（例如S1，S2，S3等）。缺陷的严重程度能够让测试团队掌握某个产品功能缺陷的影响程度。例如，严重度为S1级的缺陷表明产品的主要功能要么无法使用，要么软件崩溃。S2级的缺陷表明产品失效或是功能无法使用。表17-3给出了不同优先级和严重程度代表的意义。从以上例子中可以很清楚地看出，优先级属于管理范畴，是相对的。这意味着事先分配的缺陷优先级可以动态地改变。严重程度反映着产品的状态和质量，因此它是绝对的且不能改动。一些公司将优先级和严重程度结合起来对缺陷进行分类。

表17-3 缺陷优先级和严重程度——解释举例

优先级	含　义
1	修复具有最高优先级的缺陷，在下一次构建之前必须修复
2	在下一轮测试周期之前，修复具有高优先级的缺陷
3	在产品发布前，如果时间允许，修复具有中等优先级的缺陷
4	推迟到下一个发布版本进行修复，或接受这个缺陷
严重程度	**含　义**
1	产品基本功能失效或产品崩溃
2	由于意料的缺陷状态或者功能不能发挥作用
3	次要功能失败，或功能与预期不符
4	装饰性问题，并且对用户无影响

由于不同的公司使用不同的方法来定义优先级和严重程度，所以在表17-4中提供了通用缺陷定义和分类，因考虑优先级又兼顾严重程度，本章始终坚持这种分类。

表17-4 通用缺陷定义和分类

缺陷类别	含　义
至关重要	产品完全或不可用 需要立即修复
非常重要	产品的基本功能不能运行 需要在下一个迭代周期开始前进行修复
重要	产品的扩展功能不能运行 不影响测试进度 在产品发布前修复
次要	产品表现不一致 对测试团队或用户无影响 时间允许时进行修复
装饰性	最小的影响 本次发布不必修复

缺陷发现率

产品从开始到结束的开发周期里，依照一定的时间间隔（比如每天或每周），通过跟踪和画出所发现的缺陷总数，可以看出缺陷的出现模式。测试的指导思想是在测试周期的早期尽可能多的发现缺陷，然而，基于两个方面的原因，这种思路实现起来不太可能。第一，并不是产品的所有功能都可以在早期开发出来，由于资源的安排，产品功能是按某种顺序依次开发出来的。第二，本章前面已经提到，有些测试用例会由于出现比较严重的缺陷而被迫中止。如果晚提交的产品功能中有缺陷，或应该发现某类缺陷的测试受阻，那么尽早发现未知缺陷的目的将无法达到。一旦大多数模块被开发出来，同时受阻测试的缺陷被修复，那么缺陷出现率将会增加。经过一段时间的缺陷修复和检测后，缺陷出现率会趋于下降，这种现象的持续可以促进产品发布。这种结果就是如图17-7所示的“钟形曲线”。

测试的目的是在测试周期内尽早地发现缺陷。

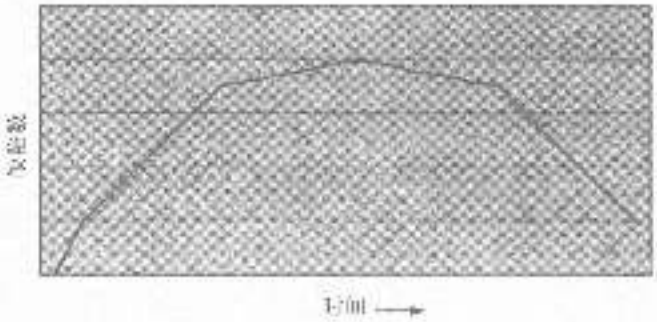


图17-7　产品缺陷发现的典型模式

对于适合发布的产品，重要的不仅是缺陷出现的模式，而且在特定的持续时间内出现的缺陷维持在很小数量也很重要。过去几天内的缺陷出现率呈钟形曲线，且发现的缺陷数达到最少，这表明即将发布的产品质量很可能是不错的。

缺陷修复率

如果测试目的是尽早发现缺陷，那么人们自然地期望开发的目的是一旦发现缺陷就立即修复，如果缺陷修复曲线与缺陷出现一致，则将再次产生如图17-7钟形曲线所示的结果，参见彩图。有一个原因可以解释为什么缺陷修复率应和缺陷出现率相同。如果在周期的后期修复了较多的缺陷，则不可能对可能产生的副作用进行充分的测试。正如回归测试中所讨论的，对产品缺陷进行修复的同时，也会产生新的缺陷。因此，应该尽早修复缺陷，并彻底测试这些缺陷，以找出所有引入的缺陷。如果不遵循这个原理，由缺陷修复所引入的新缺陷可能刚好在发布前出现，暴露出新缺陷或重新出现老缺陷，这会延误产品的发布，这种最后一分钟的修复通常不仅会导致在最后期限无法完工或产品质量出现问题，而且会使开发和测试团队置于强大的压力之下，进而影响产品的质量。

开发的目的是一发现缺陷就立刻修复。

未解决缺陷率

产品中未解决的缺陷数是产品中所发现的全部缺陷数减去已修复的缺陷数。如前所述，缺陷一发现，就应该立刻修复，而且缺陷出现呈钟形曲线。如果缺陷修复模式像直线一样恒定，未解决缺陷就像图17-7所示的钟形曲线。如果缺陷修复模式与缺陷出现率相匹配，那么

在执行良好的项目里，整个测试周期任何时候的未解决缺陷数都非常接近于零。

未解决缺陷曲线则看起来像一条直线。但是，当出现率达到钟形曲线顶端的时候，不可能修复所有的缺陷。因此，在很多项目中，未解决缺陷率的结果是钟形曲线。在测试进行中，未解决缺陷率应该保持接近于零，这样开发团队就可以在缺陷出现后很快地进行分析和修复工作。

高优先级未解决缺陷率

只关注缺陷发现率、缺陷修复率和未解决缺陷率，对于完整地理解缺陷数量还是不够的。如前所述，并不是所有缺陷的影响和严重程度都相同。有时测试产生的缺陷非常致命，要用巨大的投入去修复和测试。因此，重要的是看产品中揭示了多少严重问题。通过只画出高优先级缺陷，剔除低优先级缺陷，对未解决缺陷率进行的修正叫做高优先级未解决缺陷。由于越接近产品的发布，开发团队越不要修复低优先级的缺陷，以免修复产生难以预料的副作用，这是很重要的方法。通常在接近发布时期的阶段，仅追踪高优先级缺陷。

高优先级未解决缺陷对应于缺陷分类的关键重要和非常重要。一般公司认为，高优先级未解决缺陷包括重要缺陷（见表17-3缺陷分类术语）。

一些高优先级的缺陷可能需要在设计和体系结构上进行调整。如果软件开发生存周期的后期发现，产品的发布就要延期以便修复这些缺陷。但是，如果临近发布日期时仅发现低优先级的缺陷，而且需要更改设计，很可能管理层会决定不修复这些缺陷。如果可能的话，会提议实施临时的回避方案，并且在以后的产品发布时再修改设计。

有些缺陷修复相对需要较少的时间，但实际投入在测试的时间不少。修复相对简单，但重测是一件费时的事情。如果产品临近发布，那么对于这样的缺陷修复必须有明智的选择。对于高优先级的缺陷尤其如此。在执行这样的缺陷修复之前必须估计测试所需要的投入。如果设计修复的投入较少，一些开发人员会及时修复缺陷。如果缺陷修复需要做大量测试进行验证，那么需要进行仔细分析，在分析的基础上作出决策。在这种情况下，由于涉及的测试投入巨大，提供一个临时解决方案比修复缺陷更重要。因此，高优先级未解决缺陷应该进行单独的监控。

在画出高优先级未解决缺陷时，目的是在测试周期的任何时候都能看到未解决缺陷曲线趋近于零，这意味着高优先级缺陷需要立刻修复。

对发布关系重大的缺陷应给予额外的关注。

缺陷趋势

上面已经讨论了缺陷的单个度量方法，现在应该把上述所有内容统一为一张缺陷趋势图。图17-8给出了包含本节所讨论的所有图表的样本数据。

如果从缺陷发现率、缺陷修复率、未解决缺陷率和高优先级未解决缺陷率等多个观点分析，效果会更好。

从图17-8（参见彩图）可以观察到：

1. 缺陷发现率、缺陷修复率、未解决缺陷以及高优先级未解决缺陷均遵循钟形曲线模式。这标志着在第19周末产品发布将准备就绪。

2. 需要分析缺陷修复率的突然下降和突然增长（如图17-8所示的第13周和第17周）。

3. 在第19周结束时，有接近75个未解决的缺陷。而高优先未解决缺陷在19周结束时接近零，因此可以推断出所有的未解决缺陷属于低优先等级。这标志着发布已准备就绪。在发布前需要分析未解决缺陷。

4. 缺陷修复率和未解决缺陷率不相符。如果缺陷修复率提高了，新发现的缺陷从第14周可控，这样就能加快产品发布周期（可以将计划减少4-5周的时间）。

5. 缺陷修复率与缺陷发现率不在一个层次上。在第10周缺陷发现率高于缺陷修复率。使缺陷发现率等于缺陷修复率，可以避免第4~16周出现的未解决缺陷率峰值。

6. 平滑的高优先级未解决缺陷率表明高优先级缺陷被密切跟踪并得到修复。

通过分析上述的数据和关联每个数据及其序列，可以得到更深层次的结论。这样的分析有助于当前和将来的产品发布。

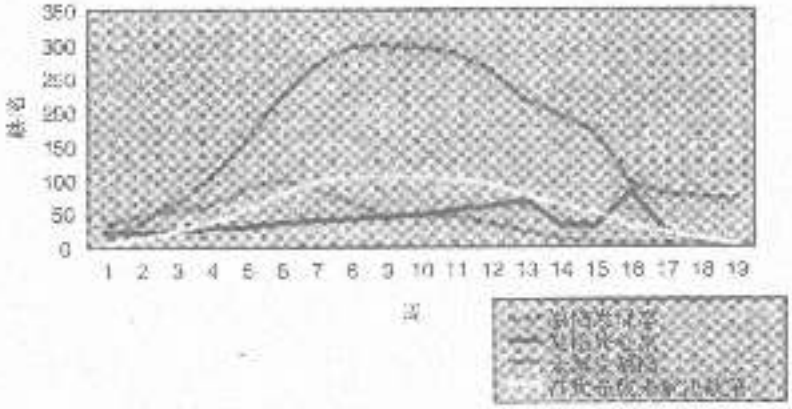


图17-8　缺陷趋势

缺陷分类趋势

通过图表形式提供缺陷分类视点，有助于确定产品发布的就绪程度。

图17-8中，缺陷分类只有二个级别（高优先级和低优先级缺陷）。一些数据挖掘或图表分析需要更详细的缺陷分类信息：至关重要的、非常重要的、重要的、次要的和装饰性缺陷。当讨论未解决缺陷的总数目时，可能会提出以下一些问题：

- 它们中有多少至关重要的缺陷？
- 多少是非常重要的缺陷？
- 多少是重要的缺陷？

这些问题需要根据缺陷分类分别绘出图表。至关重要、非常重要、重要、次要和装饰性缺陷的总数之和等于缺陷总数。在分别画出的每一类缺陷中，每个缺陷的顶部叠加得到缺陷总数的图称作"堆栈区域图"。这种类型的图有助于区分每种缺陷类型，同时展示缺陷如何累计对缺陷总量的贡献。图17-9中给出的样本数据解释了堆栈区域图的概念。

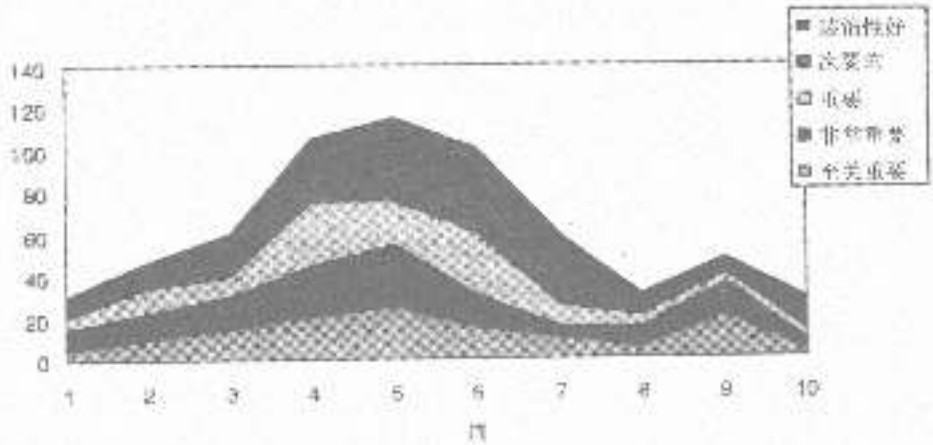


图17-9　缺陷分类趋势

从17-9图中可以观察到：

1. 由于所有类型的缺陷都有贡献，使第5周的缺陷数目达到峰值，其中至关重要和非常重要的缺陷起显著的作用。

2. 由于至关重要和非常重要的缺陷的作用，第9周缺陷数目出现高峰。

3. 由于第9周出现了峰值，因此需要再观察两周的时间，以确定产品是否可以发布。

图17-9中优先分布趋势可以按缺陷出现率和未解决缺陷率画出。然而，如果需要单独对出现缺陷进行分析，则缺陷分类趋势可以按缺陷出现率画出。

对于发布就绪程度的分析，重要的是考虑构成至关重要和非常重要缺陷中出现缺陷或未解决缺陷所占的百分比。至关重要和非常重要缺陷的低百分比，次要和表面缺陷的高百分比标志着发布准备就绪。为了进行分析，特定周的缺陷分类可画为饼形图。为了分析发布就绪程度，图17-10给出了第10周的饼形趋势图。

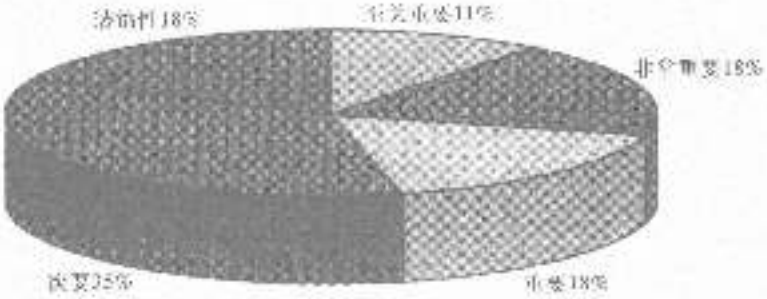


图17-10 缺陷分布的饼形图

从图17-10可观察到：接近29%的缺陷属于至关重要和非常重要的缺陷类别，这标志着产品达不到可接受的发布质量。

权重缺陷趋势

堆栈区域图提供了一些信息，这些信息表明不同级别和种类的缺陷如何影响缺陷总数。在这种方法中，所有缺陷处在相同水平。例如，非常重要的缺陷和装饰性缺陷被同等对待，都算作一个缺陷。这种相同的缺陷计数方法把至关重要和非常重要缺陷的严重性排除在外。为了解决这个问题，引入了叫做权重缺陷的度量。这个概念有助于快速地分析缺陷，而不用考虑缺陷的分类。在这种处理方法中，并不是所有的缺陷都采用同样的方式计数。严重的缺陷相比不严重的缺陷给予较高的权重。例如，使用类似下面给出的公式可进行所有带权重的缺陷计数。根据缺陷的优先级和严重程度，每类缺陷所得到的权重或增加的数字可由公司具体制定。

权重缺陷=（至关重要缺陷×5+非常重要缺陷×4+重要缺陷×3+次要缺陷×2+装饰性缺陷）

分析上述公式时，需要注意每个至关重要缺陷记为五个缺陷，而装饰性缺陷记为一个。通过这个公式不必再对堆栈区域和饼图进行分析，因为这个格式结合了所需的两种视点。根据图17-11给出的是图17-10对应的权重缺陷图。

从图17-11中可以注意到：

1. 第9周有较多的权重缺陷，这意味着存在"大量的小缺陷"或"相当多的大缺陷"，或两者兼而有之，这与使用堆栈区域图解释相同的数据是一致的。

"大缺陷"和"大量的小缺陷"都会影响产品的发布。

2. 第10周有相当多的权重缺陷（超过50个），这标志着产品发布还未准备好。

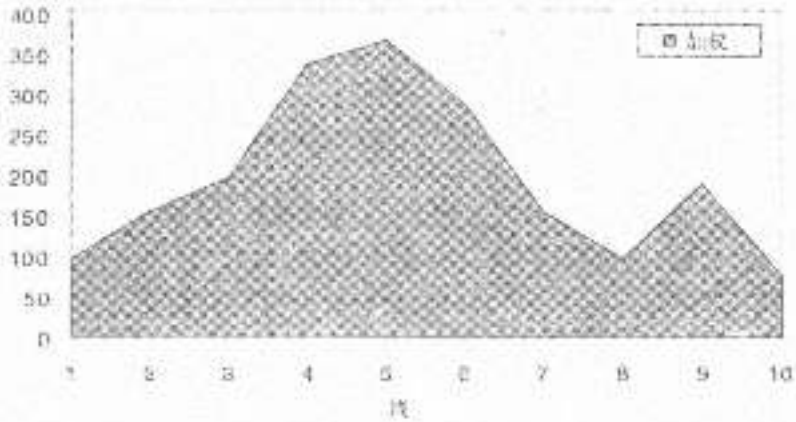


图17-11　权重缺陷趋势

因此，权重缺陷提供了一种综合的缺陷视点，不需要探究细节，这可以节省缺陷分析的时间。

缺陷成因分布

上面讨论的所有指标都有助于分析缺陷及其影响。这很自然地引出下面的问题：

1. 缺陷为什么发生？根本的原因是什么？

2. 为了通过测试捕获更多的缺陷，应该关注哪些区域？

发现缺陷根本成因有助于识别更多的缺陷，有时甚至能预防缺陷发生。例如，通过分析缺陷根本成因发现代码设计问题产生了最多的缺陷，那么测试重点应放在白盒测试和代码评审上，以防止产生更多的缺陷。这种发现缺陷成因的分析有助于回溯到软件开发生存周期阶段，重新执行这个阶段及影响区域，而不是发现和修复缺陷，因为这意味着在特定情况下"无休止的循环"。图17-12中展示了根据一些样本数据绘制的"缺陷成因分布"。

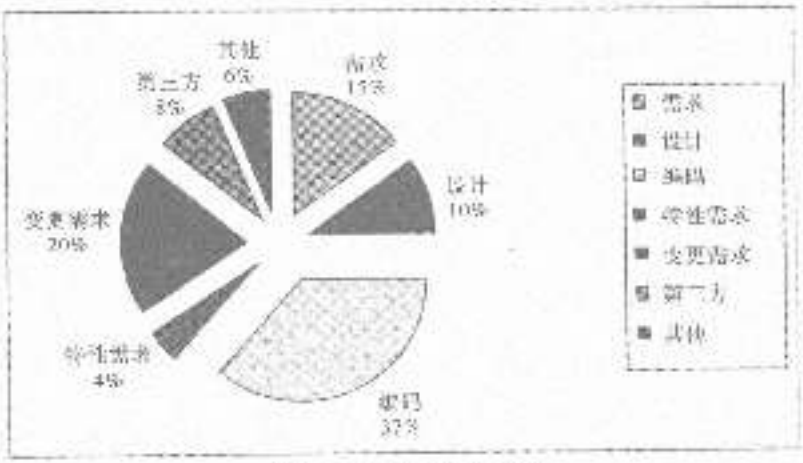


图17-12　缺陷成因分布图

如图17-12所示，编码占总缺陷的37%。如上所述，更多的测试应该集中到白盒测试方法上。下一个对总缺陷有较大影响的（占20%）是需求变更。这意味着当项目在进行时，需求始终在变更。根据所采用的软件开发生存周期模式（请参阅第2章），应该采取适合的行动，

比如，遵循螺旋或者敏捷方法的项目中20%的需求改变是可以接受的。但在使用V字模型或瀑布模型的项目里，这个比例是不能接受的，因为这些改变可能会影响产品发布的质量和时间。

在一定的时间周期间隔内，可以反复分析缺陷成因，同时可观察成因的趋势。在项目开始时，可能只有需求缺陷，随着进入后续阶段，开始出现其他缺陷。了解这些有助于分析在软件开发生存周期的哪个阶段发现什么缺陷，例如，与在需求阶段本身发现缺陷相比，在最后回归测试阶段发现需求缺陷需要更大的工作量和成本。这种分析有助于在软件开发生存周期模型的特定阶段识别问题，同时预防类似的问题出现在当前和将来的产品发布版本中。

了解缺陷成因有助于发现更多的缺陷，而且能在开发周期的早期阶段预防类似缺陷的发生。

17.5.2 开发缺陷指标

到目前为止，本章一直关注缺陷和缺陷分析，以帮助了解产品质量和提高测试的有效性。现在采取一种不同的视点看指标是如何用于改进开发活动的。本节讨论的缺陷指标有助于改进开发活动，因此称为开发缺陷指标。测试缺陷指标关注缺陷的数目，而开发缺陷指标则努力把这些缺陷映射到产品不同的组件和一些开发参数上，比如代码行。

按组件统计的缺陷分布

虽然统计产品中的缺陷数是重要的，但是对于开发来说，重要的是将缺陷映射到产品的不同组件，以便分配合适的开发人员修复这些缺陷。负责开发的项目经理要维护模块及对应开发人员的关系列表。在这张列表中，列举了所有的产品模块和开发人员。项目经理根据每一个模块中存在的缺陷数目、修复缺陷需要的工作量和每一个模块可用的技术分配资源。作为一个例子，图17-13给出了不同组件的缺陷分布。

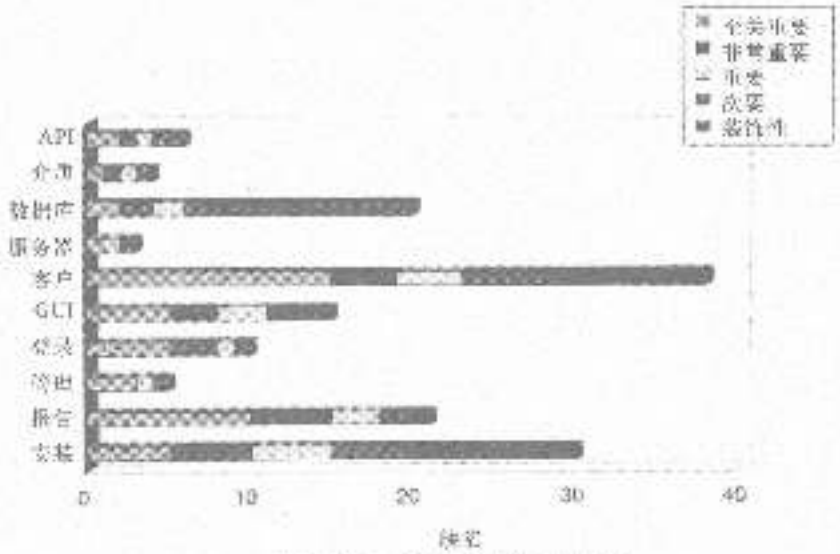


图17-13 按模块统计的缺陷分布

在图17-13中可以注意到：四个组件（安装、报告、客户和数据库）共有20个以上的缺陷，标志着这些组件需要更多的关注和资源。在图例中缺陷和分类使用不同彩色和阴影表示。缺陷分类以及对应产品每个部件的缺陷总数有助于项目经理分配并处理这些缺陷。

关于产品发布还有一个问题要考虑，就是要发布什么样的产品。假如产品中有一个独立、产生了大量缺陷的组件，如果其他所有组件都很稳定，那么减少发布版本的内容，剔除产生缺陷的组件，只发布比较稳定的组件，从而满足产品发布日期和发布质量的要求。这样做的前提是：剔除组件不会对发布产品的总体功能起决定性影响。以上把产品缺陷分类映射到产品组件的作法有利于做出这类决定。

了解产生更多缺陷的组件有助于制订缺陷修复计划，也有助于决定发布什么样的产品。

缺陷密度和缺陷排除率

质量高的产品在推度前会有较长的使用寿命，产品的使用寿命由它的质量决定，而不取决于发布产品的次数。对于已发布的产品，产品质量的合理度量是在测试中发现的缺陷数和产品发布后发现的缺陷数。如果跟踪不同版本的这些指标趋势，就可以了解到产品随着版本的不断发布而不断地得到（或没有）改善的情况。收集和分析这类数据的目的，是通过发布每个版本来提高产品质量。用户的期望值会随着时间的流逝而提高，因此这个指标非常重要。

然而在产品多次发布后只比较缺陷数目是不恰当的。通常，软件产品经历的发布（包括后来的版本升级），就编码行数或其他类似的度量而论，产品的规模也随之增加。尽管产品规模增加，客户仍然希望产品质量有所提高。

与源代码和缺陷相关的指标之一是缺陷密度。在产品中这个指标用产生缺陷的代码行数来映射缺陷密度。

计算缺陷密度有几个标准公式，其中每千行代码的缺陷数是进行估算和制订计划最简单有效的指标。KLOC代表千行代码。产品里每1000行可执行的语句计为一个KLOC。

每千行代码的缺陷数＝（产品中发现的总缺陷数）/（总的可执行代码千行数）

这样的信息可以按照每个里程碑或每个版本画出，以便了解随着时间进展的产品质量，这个指标与多个版本的产品质量度量有关。可以将当前版本的每千行代码缺陷数和以前的版本进行比较。这个指标有多个变种，适用于产品发布版本。其中之一是计算AMD（添加、修改、删除代码）来发现某次产品发布对产品质量有了什么影响。对于产品开发，构成产品的代码并不是每个版本中都完备。每次产品发布都要在现有产品中增加新功能，产品要进行修改，以修复存在的缺陷，从产品中删除不再使用的功能。因此，在这种情况下，通过计算程序的代码数量（行数）来分析产品质量的提升或下降是不合适的，因此，上面公式的分母需要修改，包含“在千行代码中所有的可执行的AMD代码数量”。修正过的公式为：

每千行代码缺陷=（产品中所发现缺陷的总数）/（千行代码中所有可执行的AMD代码数量）

每千行代码缺陷数可以作为产品发布的判断准则，也可以作为有关代码和缺陷的产品质量指示。由测试团队发现的缺陷需要由开发团队进行修复。因此，产品的最终质量由开发和测试活动共同决定。因此需要用来分析开发和测试阶段的综合指标，把它们映射到产品发布版本中。缺陷清除率（百分比）可用于这个目的。

缺陷清除率的计算公式如下：

（通过验证活动发现的缺陷－单元测试发现的缺陷）/（由测试团队发现的缺陷）×100

上面的公式有助于发现验证活动和通常由开发团队负责的单元测试的效率，并将此与测

试团队发现的缺陷进行对比，在多个版本中对这些指标进行追踪，并研究验证和质量保证活动的改进趋势。作为一个例子，图17-14给出了上面讨论的两种指标：每千行代码缺陷数和缺陷移除百分比。

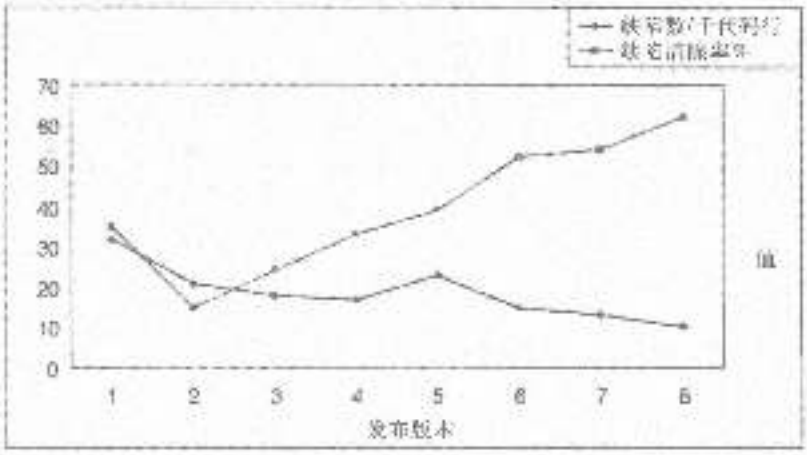


图17-14 每千行代码缺陷数和缺陷移除百分比

从图17-14可以看出，缺陷清除率提高时，每千行代码缺陷数却下降，这说明通过产品发布，产品的质量有所提高。第二版和第一版相比缺陷清除率下降了，第五版和第四版相比每千行代码缺陷数有所上升，这说明还需要对一些问题进行深入分析，找出问题的根源并加以修正。

未解决缺陷的寿命分析

以上讨论中的大部分型和指标，包括未解决缺陷趋势、未解决缺陷的分类及其修复率，只是说明了缺陷的数量，而没有反映出缺陷的寿命信息。寿命是指缺陷等待修复的时间长短。一些很难修复或者需要大量工作的缺陷可能会有较长的时间延期。因此，按这种方式，缺陷的寿命长短代表了需要修复缺陷的复杂程度。如果给定修复缺陷的复杂性和所需时间，就要进行实时跟踪，否则将可能延期直到临近产品发布，甚至耽误产品发布。跟踪缺陷的方法称为未解决缺陷寿命分析。

> 修复缺陷所需的时间与寿命成正比。

为了进行分析，要从发现未解决缺陷到现在的持续时间进行计算，每周在堆栈区域图中画出每个缺陷的重要程度。具体见图17-15。这个图有助于发现缺陷是否在发现后就立即修复，确保拉起很长时间的缺陷给予合适的优先级。前面讨论过的缺陷修复率只读到数量，但寿命分析考虑到持续时间。指标和相应图表的目的在于识别缺陷，尤其是高优先级缺陷，以及已经等了很久但还没有修复的缺陷。

从图17-15可以看出：

1. 第4周至关重要缺陷的寿命增加。
2. 第3~7周装饰性缺陷的寿命失控。
3. 缺陷的累积寿命在第2周里和第8周后都在掌控之中。

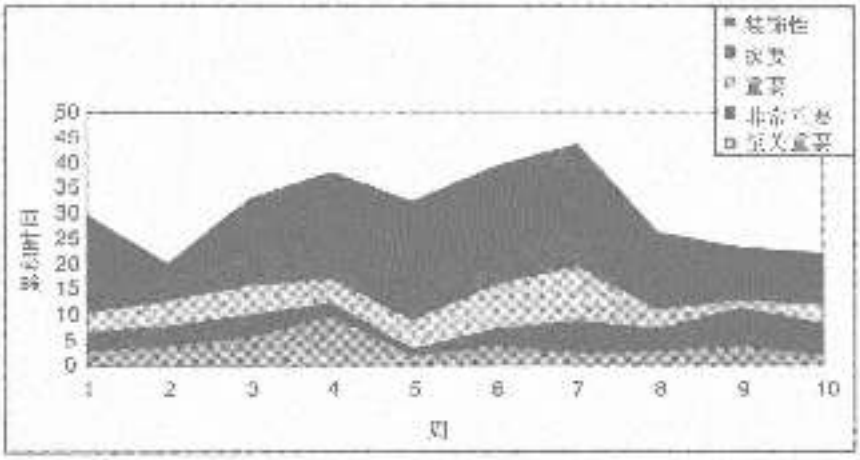


图17-15　未解决缺陷的趋势分析

引入和重新开放缺陷的趋势

当增加新的代码或者改变原有代码，以进行缺陷修复时，一些先前运行正常的程序可能停止运行。这称为引入缺陷。这些缺陷是在修复原有缺陷时注入到代码中或优化产品时生成的，这意味着这些缺陷以前并不在代码中，对应这些功能曾经很正常。

有时，通过代码修改没有彻底解决问题，而为一些修改可能再次生成以前修复过的缺陷，这叫做重新开放缺陷。因此，重新开放缺陷是指缺陷修复后不能正常运行，或者是以前修复过但又重新出现的缺陷。在没有完全理解缺陷产生的原因时修复缺陷就可能出现重新开放缺陷。这叫做缺陷的不完全修复。

所有上面提到的情况都要求在增加或者修改代码以增强产品功能或进行缺陷修复的时候，遵循编码原则和合适的评审机制。遵循这样的编码原则可以优化测试投入，并且有利于实施第8章所讨论的回归测试。太多的引入缺陷和重新开放缺陷意味着额外的缺陷修复和测试周期。因此，需要收集这些指标并定期加以分析。图17-16给出了通过样本数据绘制的引入缺陷和重新开放缺陷趋势。

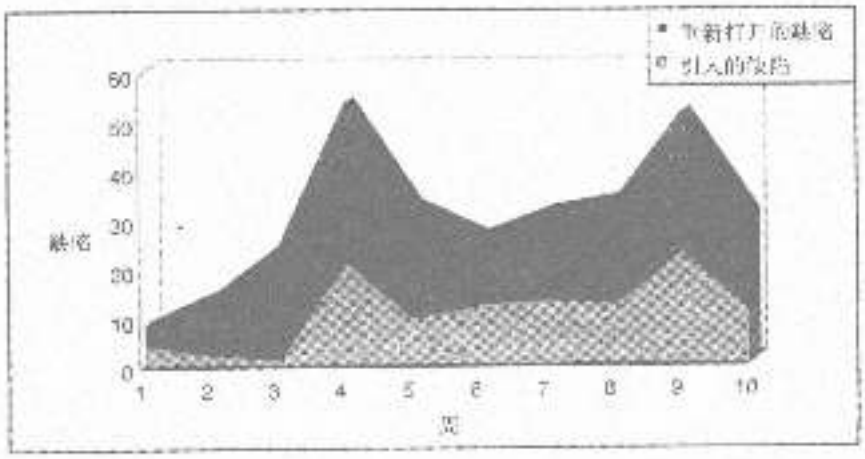


图17-16　引入缺陷和重新开放缺陷

从图17-16中可以观察到：

1. 重新开放缺陷从第4周起便维持在一个高水平上（图17-16所示与重新开放缺陷一致的区域图的厚度表明了这一点）。

2. 自第3周后，引入的缺陷处于高水平。在第4周和第9周发现更多缺陷。

3. 在第9周和第10周的所有引入缺陷与重新开放缺陷的数据表明代码不稳定。由于缺陷没有减少的趋势，产品不能发布。在这种情况下，开发团队应该检查本周改动的代码，并找出问题的根源。越是经常进行这样的分析，越可以更早地发现产生问题的区域。

测试的目的不是为了再次发现相同的缺陷，在评估产品发布就绪时需要考虑缺陷修复的质量。

17.6 生产力指标

生产力指标包括产品投入的多种度量和参数。这些度量和参数有助于发现整个团队的能力以及其他目的，比如：

1. 估计新的产品发布。
2. 了解团队的进展情况，弄清楚发生（正向和反向）偏差的原因。
3. 估算出可以发现的缺陷数。
4. 估算出产品发布的日期和产品的质量。
5. 估算出产品发布所需的成本。

17.6.1 每100小时测试发现的缺陷数

第1章提到过，程序测试只能证明缺陷的存在，而不能消除缺陷。因此，合理的推断是：测试没有尽头，而且测试越多，发现的缺陷就越多。但是进一步的测试没有发现任何缺陷时，就会造成收益的下降。如果产品中发现的缺陷在减少，也可能意味着以下情况：

1. 测试没有效果。
2. 产品的质量在改善。
3. 用在测试上的投入在减少。

前两个问题已经包含在本章讨论过的指标中。每100小时测试发现的缺陷数指标覆盖了第三点，按照投入把所发现的产品缺陷数规范化。每100小时测试发现的缺陷数的计算公式如下：

每100小时测试发现的缺陷=(一段时期内发现的所有缺陷/发现这些缺陷所用时间)×100

投入在判断产品质量方面起了重要作用。图17-17使用上面的公式通过样本数据说明了这种作用。

图17-17中的两个图都使用了相同的缺陷数据作为缺陷分类趋势，但花费的投入不同。在图17-17a中，假定在整个过程中投入恒定，该图产生钟形曲线，这表明产品发布就绪。

然而，在现实生活中，上述的假设不是成立的，每周的测试投入不可能相同。如果测试投入随着产品发布的临近而减少，图标及其分析有时会产生误导作用。这意味着，缺陷数量下降的原因不是因为产品质量的提高，而是对产品测试的关注下降了。在图17-17b中，假设在第9周和第10周花费了15个小时，而在其他各周都花费了120个小时。这种假设更贴近现实，实际表明产品的质量在下降。在第9周和第10周的测试中减少了投入，但发现了更多的缺

用所作的投入规范化缺陷数提供了产品发布质量的另一种视角。

陷，图17-17b的这个例子清楚地说明产品未做好发布的准备。

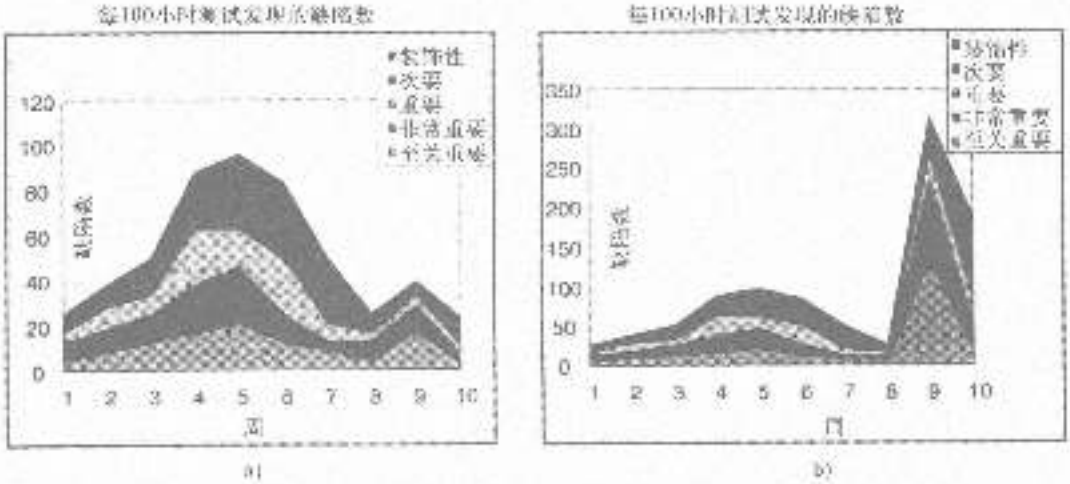


图17-17　缺陷分类趋势

不看投入就判断产品质量会产生误导，因为图17-17a所示的下降趋势假定各周投入都是相同的。从图17-17b可以看出测试人员退出测试或者用于测试的人员数量不足，是造成缺陷数下降的原因。每100小时测试发现的缺陷数量提供了重要视角，对于产品发布可做出正确决定。

17.6.2　每100小时的测试用例执行数

在一段时间内测试团队所执行的测试用例数取决于测试团队的生产力和产品的质量。必须准确地估计团队的生产力，以便对当前的发布版本进行追踪，估算产品下一次发布的时间。如果产品质量很高，因为不会有缺陷阻碍测试，可以执行更多的测试用例。同样，因为缺陷很少，用来进行文件归档、再加工和分析缺陷所需的投入就可以最小化。因此，每100小时的测试用例执行数有助于跟踪生产力，同时判断产品质量。每100小时的测试用例执行数使用下面的公式计算：

每100小时测试用例执行数=（一段时期内所执行测试用例总数/花费的时间）×100

17.6.3　每100小时的测试开发测试用例数

手动执行的测试用例和自动化的测试用例都需要估算和追踪生产率。对于产品场景，并不是每个版本的所有测试用例都重新编写。根据软件的新功能和以前没有测试过的功能添加新的测试用例，修改已有的测试用例反映了产品的变化。如果测试用例不再使用，或与之相关的功能从产品中移除，那么这些测试用例也删除。因此，测试用例开发的公式使用与增加、修改及删除测试用例有关的统计。

每100小时测试开发的测试用例数=（一定时期内开发的测试用例总数/在测试用例开发中花费的时间）×100

17.6.4 每100个测试用例发现的缺陷数

由于测试的目标是找出尽可能多的缺陷，所以有必要度量测试中的"缺陷产量"，即测试中发现了多少缺陷。这有两个参数：一是发现缺陷的测试有效性；二是选择测试用例发现缺陷的有效性。测试用例发现缺陷的能力取决于测试用例的设计和开发。但是在典型产品开发生命周期，不是每轮测试都能执行所有的测试用例。因此，最好选择能够发现缺陷的测试用例。对这两个参数进行量化的度量是每100个测试用例发现的缺陷数。但影响该指标的另一参数是产品质量。如果产品质量低劣，每100个测试用例中就会产生比优质产品更多的缺陷。用于计算该指标的公式是：

每100个测试用例发现的缺陷数=（一定时期发现的缺陷总数/
同时期执行的测试用例总数）×100

17.6.5 每100个失败的测试用例缺陷数

每100个失败的测试用例中的缺陷数是找出测试用例粒度度量的好办法。它表明：

1. 缺陷修复时需要执行多少个测试用例；
2. 需要修复什么样的缺陷，以便通过的测试用例数达到可接受的数量；
3. 在分析产品发布就绪程度时，测试用例的失败率和缺陷如何相互影响。

每100个失败的测试用例的缺陷数＝（一段时期发现的缺陷总数/
因为这些缺陷导致测试用例失败的总数）×100

在本节中所讨论的所有生产力指标除每100个小时测试的发现缺陷数之外，在图17.18中与使用的样本数据一起给出。

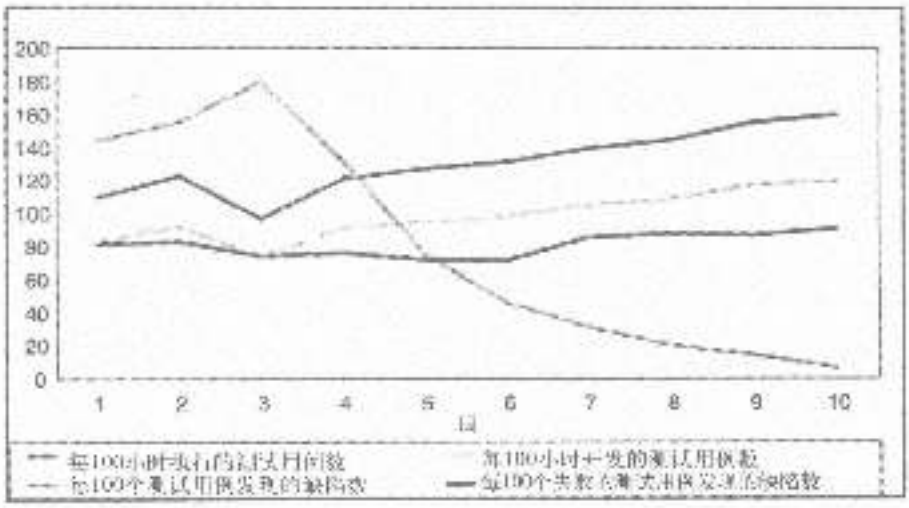


图17.18 生产力指标

从图17.18可以观察到下列现象：

1. 每100个测试用例中发现的缺陷数呈有下降趋势，表明产品发布准备就绪。
2. 每100小时执行测试用例数的上升趋势表明产品生产力和产品质量的提高（第3周数据

需要分析）。

3. 每100小时开发的测试用例数稍有上升，表明生产力的提高（第3周数据需要分析）。

4. 每100个失败的测试用例缺陷数在80~90之间，说明当缺陷修复时，相同数量的测试用例需要校验。这同时表明当缺陷修复后，测试用例通过率会得到提高。

17.6.6 测试阶段有效性

第1章已经讨论过，测试并不仅是测试人员个人的工作。开发人员执行单元测试，可能有多个测试团队一起完成部件、集成和系统测试阶段。测试的思想是在开发生存周期和测试的早些阶段发现缺陷。由于测试是由不同团队执行的，目标是在不同的阶段早期发现缺陷，因此在测试的各个阶段都需要指标来比较所发现的缺陷。例如单元测试（UT）、组件测试（CT）、集成测试（IT）和系统测试（ST）都要画出图表进行分析，具体见图17-19。

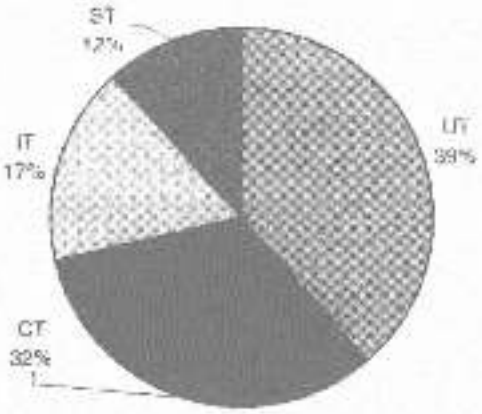


图17-19　测试阶段有效性

图17-19给出了每个测试阶段发现的缺陷总数，从中可以观察到：

1. 测试的早期阶段（单元测试和组件测试）发现很大比例的缺陷。

2. 产品质量随着测试阶段的深入而提高（后面的测试阶段（集成测试和系统测试）里显示所发现的缺陷所占比重较低）。

延伸这些数据，可以得到项目发布后的缺陷。组件测试发现的缺陷占32%，集成测试发现的缺陷占17%。缺陷数大约减少了45%。同样地，从集成测试阶段到系统测试阶段，缺陷数量大约减少了35%，现在假定产品发布后缺陷数减少了35%，占了缺陷总数的7.5%。保守的估计表明接近产品缺陷总数的7.5%是由用户发现的。这也许不是个准确的估值，但可用于培训后续支持活动的策划和配备人员。

17.6.7 已关闭缺陷的分布

测试的目的不仅是发现缺陷。测试团队也要确保测试发现的所有缺陷得到修复，以使用户从测试中受益，同时产品的质量得到提高。为确保大多数缺陷都到修复，测试团队需要追踪缺陷并分析这些缺陷是怎样关闭的。已关闭缺陷的分布有助于这种分析，如图17-20所示。

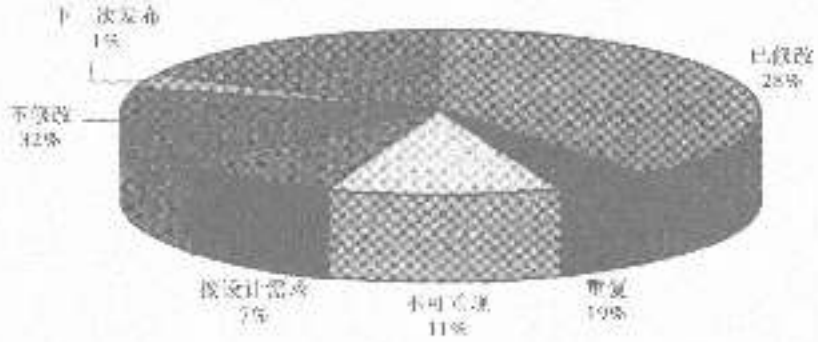


图17-20 已关闭缺陷的分布

从图17-20中可以观察到：

1. 在测试团队发现的产品缺陷中，只有28%得到了修复，这表明产品质量在发布前需要改进。

2. 归档的缺陷中有19%是重复出现的。这表明测试团队需要在新缺陷归档前对于现有的缺陷文档更新。

3. 不可重现的缺陷总数达到11%，这说明产品残留随机缺陷，或缺陷没有对应可重新执行的测试用例，这个领域需要进一步分析。

4. 接近40%的缺陷会由于“按照设计”，“不能修复”和“下次发布”等原因不能修复。这些缺陷可能会给客户带来影响，这需要进一步讨论，同时需要修复缺陷以提高发布产品的质量。

17.7 发布指标

本章讨论了多种指标以及如何用来决定产品是否可以发布。产品发布的决定需要考虑多个方面和多种指标。前面讨论的所有指标在做产品发布决定时都要综合考虑。本节的目的是提供一些指导原则以便于做出决定。这些只是些指导原则，而确切的数目和判断准则的性质会随着产品的不同、发布版本的不同、公司的不同而不同。表17-5提供为产品发布分析列出的一些视点和样本指南。

表17-5 为产品发布分析列出的一些视点和样本指南

指标	参考的方面	指南
执行测试用例	执行率 通过率	• 100%地执行所有测试用例 • 测试用例通过率最低应为95%
投入分布	没有开发人员投入太多投入	• 需求、设计和测试阶段各投入15%~20%
缺陷发现率	缺陷趋势	• 缺陷出现趋势显示为“钟形曲线” • 上周缺陷发生数接近于0
缺陷修复率	缺陷修复趋势	• 缺陷修复趋势与出现趋势相匹配
未解决缺陷趋势	未解决缺陷	• 未解决缺陷趋势显示“向下”趋势 • 临近发布前几周未解决缺陷接近0
高优先级未解决缺陷趋势	高优先级缺陷	• 临近发布前几周高优先级缺陷接近0
权重缺陷趋势	高优先级缺陷和次要的低优先级缺陷	• 权重缺陷趋势显示为“钟形曲线” • 临近发布前几周权重缺陷接近于0

（续）

指　标	参考观点方面	指　南
缺陷密度和缺陷清除率	每千行生码缺陷数 缺陷清除率	• 每千行代码缺陷数小于2 • 每千行代码缺陷数小于十次产品发布 • 缺陷清除率在50%以上 • 缺陷清除率高于上次产品发布
未解决缺陷寿命分析	缺陷寿命	• 缺陷寿命显示"向下"趋势
引入和重新开放缺陷	缺陷复原与清除 非正缺陷再次重现	• 未解决缺陷的数目和重新开放的缺陷数目都显示向下的趋势 • 引入未重新开放缺陷小于缺陷总数的5%
每100小时测试的缺陷数	缺陷率与引入的比例是否合适	• 每100小时测试的缺陷数应小于5 • 每100小时测试的缺陷趋势显示向下趋势
每100小时执行测试用例数	产品的质量提升是否允许更多的测试计划执行 测试用例执行与引入比例是否合适	• 测试用例执行显示向上趋势
测试阶段有效性	测试的每个阶段发现的缺陷数	• 在系统层确认测试阶段发现的缺陷比重很低（比如低于12%） • 缺陷分布与下一测试阶段相比，缺陷发生率减少 • 理想状态是单元测试占50%，组件测试占30%，集成测试占15%，系统测试占5%
已关闭缺陷分布	测试发现与缺陷已被修复的比例是否很高	• 至少50%关闭的缺陷被修复 • 不可重现缺陷小于5% • 转移到下次产品发布的缺陷应小于10%

17.8 小结

本章定义的指标提供了不同的视角并互相补充。因此，所有指标不能孤立地对待，必须综合分析。而且，不是所有的指标在所有情况下都是一样重要的。

管理层的承诺和促进捕获分析缺陷的文化是公司指标程序成功的重要方面。有一种观念：只要公司有丰富的项目管理经验，那么就不需要指标。有人会听到类似的话："我相信我的直觉，我为什么需要指标？"关键是除了确认直觉，指标还提供了许多新视点，甚至有经验的项目经理也很难看到这些新视点。还有其他一些有关指标的习惯看法，只有指标成为公司的习惯和文化的一部分才能纠正这些看法。

指标不仅用来改进过程和管理项目，而且可以估计并提高产品质量。指标和为了生成指标所采集的生产力数据决不能用来衡量团队或团队中个人的工作能力。这违背了指标的目的和本意。

如果公司的员工不能够一致地理解指标的目的和意图，那么就不能得到真实、准确、实际可行的数据。对这种错误数据的分析会产生误导。即使数据是正确的，人们为了使其符合自己的看法和需要也会将其扭曲。

生成指标和制作图表只是指标程序工作的一部分。分析指标、跟踪项目行动直到结束是耗时又很困难的活动。指标已生成，如果有忽视指标结果的倾向，团队成员依照所谓的直觉发布产品，那么有指标和没有指标没有什么不同。

自动收集数据和自动制作图表可以减少在指标上的投入。然而，指标的分析和付诸行动

依然要人工完成，因为分析需要人类的智慧与投入。组织安排合适的人去分析指标的结果，并确保各项活动及时地进行，这些有助于巩固指标程序的可信度并提高组织中指标的有效性。

问题与练习

1. 投入与计划的区别是什么？
2. 指标程序包含哪些步骤？简要解释每一步骤。
3. 在产品开发和测试中使用指标有什么重要益处？
4. 如果在投入和计划上有负偏差，这意味着什么？
5. 画出权重缺陷的附加价值是什么？和缺陷分类趋势相比，权重缺陷以何种方式提供额外的视角？
6. 请列举并讨论可以用作缺陷预防的指标，怎样预防？
7. 如何计算缺陷密度和缺陷清除率？为了得到好的产品质量，请讨论提高这些比率的方法。
8. 进行产品发布时，要考虑的视角和质量因素是什么？请以指标的形式，解释几个关于如何画出和分析质量因素的指导原则。

参考文献

ALBE-79	Albrecht A.J., *Measuring Application Development Productivity*, Proc IBM Application Development Symposium, October 1979, pp. 83–92.
ALBE-83	Albrecht A.J., and Gaffney J.E. Jr., *Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation*, IEEE Transactions on Software Engineering, SE-9, pp. 639–648 (1983).
BACH-2003	James Bach, *Exploratory Testing Explained*, http://satisfice.com (2003).
BCS-2001	British Computer Society, *BCS SIGIST Standard for Software Component Testing*, Draft 3.4, April (2001).
BEIZ-90	Boris Beizer, *Software Testing Technique, 2nd edition*, International Thomson Press, New York (1990).
BIND-2000	Robert Binder, *Testing Object-Oriented Systems: Models Patterns and Tools*, Addison Wesley (2000).
BOEH-81	B. Boehm., *Software Engineering Economics*, Prentice Hall (1981).
BOOC-94	Grady Booch., *Object Oriented Analysis and Design With Applications*, Addison Wesley (1994).
BOOC-99	Grady Booch, Ivar Jacobson and James Rumbaugh, *The Unified Development Process*, Pearson Education (1999).
BROO-75	F.P. Brooks, *The Mythical Man Month*, Addison Wesley (1975).
BURN-2004	Ilene Burnstein, *Practical Software Testing*, Springer (2004).
COVE-89	Stephen Covey, *The Seven Habits of Highly Effective People*, Pine cide, 1989.
CROS-80	Philip. B. Crosby., *Quality is Free: The Art of Making Quality Certain*, Mentor Book (1980).
DEMA-87	Tom DeMarco and Timothy Lister, *Peopleware—Productive Projects and Teams*, Dorset House Publishing Company, New York (1987).
DEMI-86	W.E. Deming., *Out of the Crisis*, MIT Center for Advanced Engineering Study (1986).
FAGA-76	Fagan., M.E., Design and *Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal. Vol 15, No 3 (1976).
FAIR-97	Richard. E. Fairley and Richard. H. Thayer., *Work breakdown structure, In: Software Engineering Project Management*, IEEE Computer Society (1997).
GOOD-75	Goodenough, J.B., Gerhart, S.L., *Towards a Theory for Test Data Selection*, IEEE Transactions on Software Engineering, pp. 156–173 (1975).
GRAH-94	Ian Graham, *Object Oriented Methods, 2nd Ed.*, Addison Wesley (1994).
HUMP-86	Watts Humphrey, *Managing the Software Process*, Addison Wesley (1990).
IEEE-1012	*IEEE Standard for Software Verification and Validation*, IEEE Std 1012–1998.
IEEE-1059	*IEEE Guide for Software Verification and Validation Plans*, IEEE Std 1059–1993.
IEEE-1994	*IEEE standards collection, Software Engineering* (1994).
IEEE-2001	*IEEE Software March/April 2001—Special Issue on Global Software Development.*
IEEE-829	*IEEE Standard for Software Test Document IEEE Std 829–1998.*
IFPU-94	*Function Point Counting Practices Manual*, Release 4.0., IFPUG., 1994.
KANE-2001	Cem Kaner, James Bach and Bret Pettichord, *Lessons Learned in Software Testing*, Wiley (2001).
KARL-E	Karl E. Wiegers, *Software Metrics Primer*, www.processimpact.com.

MARI-2001	Brian Marics, *Pair Testing*, http://www.testing.com, (2001).
MCCA-76	Thomas J. McCabe, *A Complexity Measure*, IEEE Transactions on Software Engineering, pp. 101–111 December, (1976).
MYER-79	Glenford Myers., *The Art of Software Testing*, Wiley (1979).
NIST-1	NIST Special publication 500–235, *Structured Testing (A Testing Methodology Using Cyclomatic Complexity Metric)*.
PMI-2004	*Guide to Project Management Book of Knowledge, 3rd Ed.*, Project Management Institute (2004).
PRES-97	Roger Pressman: *Software Engineering—A Practitioner's Approach*, 4th Ed., McGraw Hill (1997).
RAME-2002	Gopalaswamy Ramesh, *Managing Global Software Projects*, Tata McGraw Hill (2002).
SHNE-97	Ben Shneiderman, *Designing the User Interface—Strategies for Effective Human-Computer Interaction*, Addison Wesley (1997).
SRINI-2003	Srinivasan Desikan, *Building an Effective Test Organization*, SIEIA Software Journal, June (2003).
SRINI-2003A	Srinivasan Desikan, *A Test Methodology for Effective Regression Testing*, www.stickyminds.com (2003).
TET	*Test Environment Toolkit*, http://tetworks.opengroup.org.
UMES-2002	Umesh M.S. and Srinivasan Desikan, *Validating Mission Critical Server Software for Reliability*, February (2002).
UNI-2005	www.unicode.org *What is Unicode?* (2005).
UNI-2005A	www.Unicde.org/glossary, *Glossary of Terms* (2005).
YEWU-2001	Ye Wu, *A Presentation on Integration Testing*, George Mason University (2001).
YOGI-2002	Yogita Sahuu, *Looks Do Matter*, ASIASTAR-2002 conference proceedings (2002).
	Web references for Standards
MSXP	Microsoft XP Accessibility Help Document XXX Needs a Full Reference
508	www.access-board.gov/sec508/508standards.htm
W3C	www.w3.org/
	Web references for Testing Tools
Mercury	www.mercury.com
Compuware	www.compuware.com
Segue	www.segue.com

好书推荐

《软件工程：实践者的研究方法》（第6版）
作　者：[美] Roger S Pressman
译　者：郑人杰 等
中文版：7-111-49486-4　69.00元
英　文
原书版：978-7-111-22140-2　48.00元
英　文
原书版：978-7-111-24038-6　68.00元

《软件工程》（第8版）
作　者：[美] Ian Sommerville
译　者：[illegible]
中文版：7-111-[illegible]-8　[illegible]元
英文版：7-111-[illegible]-4　79.00元

《软件工程：面向对象和传统的方法》（第7版）
作　者：[美] Stephen R. Schach著
译　者：[illegible]
中文版：978-7-111-21762-2　49.00元
英文版：978-7-111-[illegible]-4　[illegible]元

《软件工程：面向对象和传统的方法》（第7版）
作　者：[美] Eric Gamma等
译　者：李英军 等
中文版：978-7-111-21723-4　68.00元
英文版：7-111-[illegible]-7　36.00元
原文版：7-111-[illegible]-7　38.00元

《UML和模式应用》（第3版）
作　者：[美] Craig Larman
译　者：李刚 等
中文版：7-111-[illegible]-8　36.00元
英文版：7-111-[illegible]-4　75.00元

《IT项目管理》（第4版）
作　者：[美] Kathy Schwalbe
译　者：韩松麟 郑秀芳
中文版：7-[illegible]-[illegible]-8　[illegible]元
英文版：7-[illegible]-[illegible]-4　[illegible]元

《软件测试》（第4版）
作　者：[美] Paul Ammann
译　者：[illegible]
中文版：7-[illegible]-[illegible]-8　[illegible]元
英文版：7-[illegible]-[illegible]-4　38.00元

《软件测试基础教程》
作　者：[illegible]
译　者：[illegible]
中文版：978-7-[illegible]-[illegible]-8　[illegible]元
英文版：978-7-111-24-[illegible]-6　40.00元

教师服务登记表

尊敬的老师：

您好！感谢您关注我们出版的 _____ 教材。

机械工业出版社华章公司本着为教育服务的宗旨，为广大高校教师能顺利与我们沟通，更好地为高校教师服务，特填此表，请您填妥后寄回给我们，我们将定期向您寄送华章公司最新出版图书目录。为您与我们沟通、参考选评教材提供一个良好的平台。欢迎您对我们的教材和服务提出宝贵意见，感谢您的大力支持！

个人资料（请用正楷完整填写）

教师姓名		□先生 □女士	出生年月		职务	职称：□教授 □副教授 □讲师 □助教 □其他	
学校			学院			系别	
联系电话	办公： 宅电： 移动：			联系地址及邮编			
				E-mail			
学历		毕业院校		国外进修及讲学经历			
研究领域							

主讲课程	现用教材名	作者及出版社	共同授课教师	教材满意度
课程： □专 □本 □研 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换
课程： □专 □本 □研 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换

样书申请			
已出版著作		已出版译作	
是否愿意从事翻译/著作工作 □是 □否		方向	
意见和建议			

填妥后请选择以下任何一种方式将此表返回：（如为复印件请留名片）

地 址：北京市百万庄大街22号 华章公司营销中心 邮编：100037

电 话：(010)88379079 88379895 传真：(010)68995293

E-mail:hzedu@hzbook.com marketing@hzbook.com 图书样章目录http://www.hzbook.com网站查询