HZ BOOKS

软 件 工 程 技 术 丛 书

测试系列

软件测试经验与教训

Lessons Learned in Software Testing

（美） Cem Kaner
James Bach
Bret Pettichord 著

韩柯 等译

机械工业出版社 China Machine Press

中信出版社 CITIC PUBLISHING HOUSE

Lessons Learned in Software Testing

软件测试经验与教训

"本书中充满对软件测试实践富于启发性的新观点，会使你重新思考作为软件测试基础的各种假设和传统理论。"

—— Ed Yourdon

"这些经验中的任何一个，都抵得上这本书的价钱。"

—— Tom DeMarco

优秀的软件测试团队不是天生的，而是造就的，是通过大量艰苦工作和有效沟通造就的。在这个过程中，有很多陷阱，这些陷阱会使精心制订的计划出现偏差，使项目不能按进度完成。

本书的三位作者具有多年的测试经验，知道成功的测试都需要什么。在这本革命性的新书中，他们汇总了293条测试经验建议，阐述了如何做好测试工作，如何管理测试，以及如何澄清有关软件测试的常见误解。读者可直接将这些经验用于自己的测试工作中。这些经验中的每一条都是与软件测试有关的一个观点，后面是对运用这条经验的方法、时机和原因的解释或例子。

为了满足不同层次的软件测试员、开发人员和管理人员的需要，本书还提供以下内容：

- 根据世界顶级软件测试专家多年的测试经验总结出的有用实践和问题评估方法。
- 在所有关键问题上积累的经验，包括测试设计、测试自动化、测试管理、测试策略和错误报告。
- 如何将本书提供的经验有选择性地运用到实际项目环境中的建议。

作者简介：

Cem Kaner 是美国佛罗里达技术学院的教授，同时为软件开发界提供技术和管理等方面的咨询工作。他是世界级的测试技术权威，《Testing Computer Software》（中文版即将由机械工业出版社出版）和《Bad Software》这两本书的第一作者。

James Bach 是Satisfice公司创始人和首席顾问，这是一家软件测试和质量保证公司。他具有在顶级硅谷公司（例如苹果和Borland公司）从事软件开发的经验，这使他在"足够好的"质量、基于风险的测试、探索测试和其他对技能和判断能力要求很高的技术等方面拥有丰富经验。他还是软件测试实验室的首席科学家。

Bret Pettichord 是独立顾问，并为www.testinghotlist.com编辑很流行的"软件测试活动表"。他经常演讲并发表文章，并且还是测试自动化Austin研讨会的创始人。

适用水平：中级

封面制作 锡彬

ISBN 7-111-12975-X

9 787111 129752

HZ BOOKS

华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259，68995264

读者服务信箱：hzedu@hzbook.com

http://www.hzbook.com

ISBN 7-111-12975-X/TP · 2907

定价：35.00 元

测试系列

（美） Cem Kaner
James Bach
Bret Pettichord
著
韩柯 等译

软件测试经验与教训

Lessons Learned in Software Testing

机械工业出版社 China Machine Press
中信出版社 CITIC PUBLISHING HOUSE

本书汇总了293条来自软件测试界顶尖专家的经验与建议，阐述了如何做好测试工作、如何管理测试，以及如何澄清有关软件测试的常见误解，读者可直接将这些建议用于自己的测试项目工作中。这些经验中的每一条都是与软件测试有关的一个观点，观点后面是针对运用该测试经验的方法、时机和原因的解释或例子。

本书还提供了有关如何将本书提供的经验有选择性地运用到读者实际项目环境中的建议，在所有关键问题上所积累的经验，以及基于多年的测试经验总结出的有用实践和问题评估方法。

Cem Kaner, James Bach, Bret Pettichord: Lessons Learned in Software Testing (ISBN: 0-471-08112-4).

Authorized translation from the English language edition published by John Wiley & sons, Inc.

本书版权登记号：图字：01-2002-1057

图书在版编目（CIP）数据

软件测试经验与教训/（美）凯纳（Kaner, C.）等著；韩柯等译. -北京：机械工业出版社，2004.1

（软件工程技术丛书　测试系列）

书名原文：Lessons Learned in Software Testing

ISBN 7-111-12975-X

Ⅰ.软…　Ⅱ.①凯…②韩…　Ⅲ.软件-测试　Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2003）第080767号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：杨　文

北京中加印刷有限公司印刷·新华书店北京发行所发行

2004年1月第1版第1次印刷

787mm×1092mm 1/16·16.25印张

印　数：0 001-5 000册

定　价：35.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

本社购书热线电话：(010) 68326294

译者序

应该承认，这是一本很吸引人的书。它的精彩之处在于它使各类软件测试人员，甚至是与测试人员打交道的人，都能得到很好的启发；在于它能够引导读者对自己在实际工作中的得与失做比较全面的思考，重新审视过去很多深信不疑的软件测试规则；在于它直接面对令大家感到困惑的问题，从更实际的角度进行诠释。

作者结合自己丰富的软件测试实践经验，大胆地对软件测试界很多人多年来鼓吹的所谓最佳实践、关键活动、甚至国际标准进行了深刻的反思，令人信服地提出了自己的观点，对一些关键问题做了哲学思考，内容涉及与软件测试有关的各个方面，很适合有一定实际经验的软件测试、项目管理、软件开发、软件工程等方面的工程技术人员阅读。

在翻译过程中，我们力求忠实原文。但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：原小铃、李津淳、王威、屈健、黄惠菊、韩文臣、朱军、杜蔚轩、解冀海、付程、孟海军、耿民、王强等。

译　者

序

请想像自己拿着一瓶50年的陈年波尔多红葡萄酒。这里有一种品味的方法，虽然这并不是惟一的方法，但是有多年饮用波尔多经验的人大都会发现，有些经验有助于更好地享用波尔多。以下就给出几条经验：

经验1：不要直接用瓶子喝。如果没有玻璃杯，也没有其他可用的容器，可把少量波尔多倒在自己的手掌中吸吮。在吸吮的同时，应该嗅一嗅波尔多的芳香，让波尔多在舌头上来回流动，不要一口咽下。

经验2：不要整瓶喝下去。如果喝波尔多是为了解渴，那么还是请把波尔多放下，而喝一大杯水吧。每次饮用少量的波尔多，会使整瓶波尔多带来的享受达到极致。

经验3：不要污染波尔多。如果有人建议尝试一下用橙汁、海水和波尔多勾兑的鸡尾酒，那么请礼貌地拒绝他，带着绚烂的微笑说："可是我想享用的是一杯波尔多。"

经验4：不要独贪波尔多。把波尔多藏起来，就意味着再也不能享受一边与朋友轻松地闲谈，一边品味波尔多的乐趣。波尔多最好还是与喜欢来一杯的朋友共享。请记住，他们可能也会有一瓶波尔多。

读者手中拿的并不是一瓶波尔多，而是本书，这是一本有关软件测试的非常有价值的专著。它通过作者多年经验的酿造，已经至善至纯。波尔多是你味觉感官的朋友，而本书则是你头脑的朋友。我想读者还会发现其他值得注意的差别。我已经体验了本书，并总结出以下经验，希望这些经验能够帮助读者通过本书得到最大收益。

经验1：不要直接用瓶子喝。阅读本书时要带上自己的容器。也就是说，带上自己所有的软件开发和测试经验。如果读者从来没有参加过正式的软件开发工作，则本书对你可能度数太高了。它会使你头晕，在一段时间内浑身无力。如果读者有一定的经验，则可以结合自己项目背景，享用本书的内容。

经验2：不要整瓶喝下去。不要一口气把它读完。读上一两条经验，合上书，想想你对Kaner、Bach和Pettichord的话有何反应。读者会发现，他们把自己的方法叫作"语境驱动"的测试。只有读者才会知道自己工作的背景，必须靠自己确定本书所给出的经验在哪些方面适合自己的具体工作。

经验3：不要污染波尔多。有人要为本书的293条经验加上标题，并制成表。但愿读者没有这种想法。本书的核心是每条经验的解释。还要当心有人会立即尝试把本书内容ISO化或CMM化。我现在可以看到"使用这本书将达到CMM第293级"这样的文章标题。哈！正如作者所说："……我们不相信'最佳实践'，我们相信在一定条件下，一些实践比另一些更有用。"这些话代表了达到软件测试大师级专家境界的思想精华。

经验4：不要独贪波尔多。如果有什么书要和同事一起读的话，本书就是。买上一箱，送给搞测试或管理测试员的人每人一本。一次有选择性地读上几条经验，然后聚在一起边讨论，边喝咖啡、吃午餐，甚至享用波尔多！阅读、反思、享受。夫，干杯！

Tim Lister
美国纽约市
大西洋系统协会
2001年8月17日
lister@acm.org

前　言

提出软件工程知识体系（Software Engineering Body of Knowledge，SWEBOK）的目标是为颁发政府执照、规范软件工程师以及开设软件工程大学课程提供一种恰当的基础。SWEBOK文件声称以大部分人的意见为基础，希望这样的文件能够包容这个领域已经积累的知识和智慧（已经积累的经验教训）。

有关探索式测试，SWEBOK只有如下的叙述：

也许，最广泛采用的手段还仍然是专门定制的测试，即依靠测试员的技能和直觉（“探索”测试），依靠测试员在类似工程上积累的经验所导出的测试。虽然人们建议采用更系统化的方法，但是专门定制测试方法对于确定形式化手段不能轻易“捕获”的特殊测试用例还是很有用的（但是只有当这些测试员都是真正专家的时候）。此外还必须指出，这种手段的有效性会有大幅度的变动（SWEBOK 0.95，2001，第5~9页）。

SWEBOK怎样看待这种在测试领域中被最广泛地采用的手段呢？它丝毫没有涉及这种手段的使用方法，只是说应该由真正专家进行探索，建议使用其他方法，认为其他形式化的手段会使有效性更加稳定。

哈！

我们并不打算向读者正式描述反映测试领域大多数人观点的所谓知识体系，但是我们确实要大量谈论这个领域中最常见的实践。本书不是要排斥探索式测试，而是要站在使用探索法（以及很多其他方法）在现实条件下达到最佳测试效果的人员立场上，来描述测试应该是怎样的。

欢迎阅读本书

本书是我们实际软件开发经验的总结。我们几个人加起来已经有了50~60年的开发经验（当然这要看怎样计算了），其间，我们既看到了大量出色的工作，也看到了大量不那么出色的工作。

本书并不打算讨论软件工程在更有规则、更受控的环境中怎样运用，而是要讨论我们所要面对的现实世界。

在我们的世界中，软件开发团队常常要在时间很紧张的情况下工作，要在探索该做什么的同时，探索该怎样做。所使用的方法，有时比较正式，有时就不那么正式。这取决于千差万别的实际环境。

我们采用的是软件测试中的**语境驱动法**（context-driven approach）。我们认为在某些环境中很有效的方法，在另外一些环境中就没有效果。我们不谈论最佳实践，而是谈论最适合当前特定环境的实践。本书的最后要讨论语境驱动法，但是从本质上看，语境驱动的

测试要从“谁”、“什么时候”、“哪里”、“为什么”和“如果这样会出现什么”五个方面，来研究测试的“内涵”（手段、工具、策略等）。

我们的目标是将所选择的实践与当前具体环境匹配起来，以取得理想的测试效果。我们不认为通过接管项目，或通过跺着脚告诉项目经理（或执行经理）“真正的专业人士”该怎样管理项目，就能够取得理想的测试效果；不认为通过强迫程序员，或通过奉承他们，就能够取得理想的测试效果；不认为通过填写数以千计的小卡片（或对应的电子记录），或通过在没有必要的过程上浪费其他所有人的时间，就能够干好测试工作。

我们并没有严格的条条框框，也没有理想化的有关测试的灵丹妙药。

这种灵丹妙药根本就不存在！

我们认为好的测试包括要求具有很高技能的技术工作（搜索缺陷）和准确、有说服力的沟通。

技艺高超的搜索毕竟也是一种探索。有无限多的测试工作要做，但是只有少量时间，只能完成其中很小一部分测试。要完成的每个测试，要写的每份文档，要参加的每个会议，都要占用可能会发现关键缺陷的测试时间。面对这种限制，我们要优化测试过程，以便充分利用不断积累的产品及产品市场、产品应用和产品弱点等方面的知识。我们今天所学的知识，会在明天更好的测试中体现出来。即使：

- 产品被相当完备地描述。
- 规格说明准确地反映了需求文档。
- 需求文档准确地表达了产品项目相关人员的实际需要。

（读者是否确实见过具备所有这些条件的项目），我们在测试这样的产品时，仍然会学到很多这种产品的测试方法。具体地说，当发现错误时，我们知道这组程序员会犯怎样的错误。规格说明告诉我们，程序如果编写正确，应该怎样完成功能，但是不会告诉我们预期会出现的错误，也不会说明应该如何设计测试以发现这些错误。对于这件关键工作，我们要从头至尾地在一个项目中逐渐改进，一个项目一个项目地逐渐改进。

不管外界怎么看，只要我们在用我们的头脑进行测试，我们的工作就是探索。

本书的读者对象

本书针对测试软件和管理测试员的所有读者，针对在其软件开发项目中要与测试员打交道的所有读者。

我们主要针对的读者应该已经从事了几年的测试工作，可能最近被提升到管理岗位。我们希望这样的读者能够从本书找出大量与自己的经历一致的内容，从而对我们的经验能够有新的理解，能够找出可以向自己的经理转述的经验，能够很喜欢我们的一些阐述，以至于将我们所说的话放大贴在办公室中的显著位置。读者也许会对我们的至少一句话反应如此强烈，以至于将它贴在自己的飞镖靶盘上。（我们要启发读者的思想，而不只是想让读者

赞同我们的观点。）

刚开始从事测试工作的读者（以及正在申请从事测试工作的读者）不会有很多机会感觉到已经经历过我们所谈论的问题。对于这样的读者，本书可以提供一些早期认识和忠告，对测试员要面临的问题有一个很好的体验。

> **提示** 如果读者对测试绝对是新手，正在找一本书来研究，以便为一次工作面试做准备，那么本书是不合适的。如果这是你能够找到的惟一一本书，那么请仔细研究第3章和第10章中的内容。如果有多本书可以挑选，我们建议阅读《测试计算机软件》（Testing Computer Software）（Kaner等，1993）的前5章。

对于必须与测试员打交道的程序员、项目经理和执行经理，本书会有助于这些人正确期望测试小组的工作。我们希望本书有助于这些读者在不赞同测试小组的政策，或感到他们没有明智地利用自己的时间时，能够评估和讨论对测试小组的意见。

本书包含的内容

多年以来，我们学会了很多有用的实践和很有帮助的问题评估方法。我们得出结论的基础是经验。本书归纳了我们的很多经验，将这些经验组织为接近300个短小、可读性好的经验系列描述。

选择这些经验时，我们遵循了以下一些准则：

- 经验应该是有用的，或有自己的观点。
- 经验应该通过90分钟独立思考的考验。如果是什么人漫不经心地花90分钟就可以提出观点的话，那么这样的经验不值得收入本书。
- 经验必须基于我们的实际经历。我们中至少有一人（最好是三人）曾经成功地运用过我们给出的建议；我们中至少有两人在尝试我们所批判的实践时，遭受过挫折。（请注意：有时我们会根据自己不同的经历，分别得出不同的结论。偶尔读者会发现我们决定提供两种观点，而不是一种。即使只提供一种观点，读者也不能就认为我们三人都完全赞同这种观点。当出现不同意见时，我们三人很可能会服从对该问题有最多经验的一两个人。）
- 经验应该根据我们同事的经历进行调整。通过“测试自动化Los Altos研讨会”、“软件测试经理圆桌协会”、“启发式与探索测试技术研讨会”、“测试自动化Austin研讨会”、“软件测试模式研讨会”、“基于模型的自动化测试研讨会”、“系统有效性管理集团”等几十个软件测试会议，以及不太正式的同级相互培训（例如在Crested Butte举行的一年一度的顾问营），我们收集了详细的经验报告。
- 经验应该简短，切中有害，但又易于理解。
- 经验可以长一些，但是只限于解释如何做，或提供有用的工具。很长的解释和详细背

景信息不适合本书。

- 经验应该是自完备的，读者可以从书中任何地方开始阅读。
- 经验合在一起，应该使读者对我们怎样做测试以及考虑测试，能有一定的认识。

本书不包含的内容

本书并不是软件测试的综合指南。

本书并不是永远都正确的经验汇集。这些经验是我们自己的经验，依据的是我们的经历。我们相信这些经验可以广泛运用，但是有些在我们的经历中被证明是很有用、很重要的经验，可能并不适用于读者。读者要自己进行判断。应该指出，本书适用性的一个具体限制因素是，我们更多从事的是面向大众市场的软件开发，以及以具体合同为基础的软件开发，自用软件开发的经验较少。我们在开发生命关键软件和嵌入式软件方面的经验也不多。

本书并不是最佳实践的汇集。事实上，我们并不相信“最佳实践”。我们相信在一定环境中，有些实践比另外一些更有用。我们注意到有很多东西被作为最佳实践，不加批判地向并不适合的场合推广（并应用）。

如何使用本书

本书的结构设计旨在便于读者浏览或有选择性地阅读，而不是从头读到尾。有时（我们希望）读者会发现“金块”，发现对自己非常有吸引力的思想。我们建议读者不要不加批判地运用这些思想。（我们的经验并不是最佳实践。）相反，我们强烈希望读者能够评估这些经验对于自己实际环境的适合性。

以下是一些有助于读者进行评估的问题：

- 在什么条件下，自己的公司运用这个经验会收到效果？
- 在什么条件下，运用这个经验不会收到效果？
- 你认识什么人以前尝试过类似的经验吗？效果怎样？为什么会有这种结果？你当前的项目与那个人的项目有哪些不同？这些差别重要吗？
- 通过尝试运用这个经验，谁最有可能受益？
- 通过尝试运用这个经验，谁的利益最有可能受到影响？
- 通过尝试运用这个经验，你可以学到什么？
- 尝试任何新东西都会增加风险。通过试点，即不是投入太多地尝试该经验，或在低风险环境中研究该经验运用于自己的公司有多大作用，这有意义吗？在公司中引入试点，对当前项目（或对其服务）尝试该经验是否可行？
- 你怎么知道运用该经验是否有作用？如果有作用，你怎么知道之所以取得成功，更多的是因为该经验的内在价值，而不是靠尝试该思想时的最初热情？如果继续运用该经验，这种成功会持续下去吗？

- 作为尝试运用该经验的结果，对你会出现什么最好和最坏的事情？
- 会对其他项目相关人员，例如一个用户或开发团队的其他成员，出现什么最好和最坏的事情？
- 如果公司中有一位关键人物不赞同你要尝试的经验会出现什么情况？你怎样克服他们的反对，并向他们推销要尝试的经验？

我们希望本书能够引出不同意见和争论

本书显示出我们观点和其他观点之间的鲜明对比。我们认为这种对比有助于引起在测试领域展开本来就应该展开的论战。我们认为在软件测试或作为整体的软件工程中，还没有一种已经被普遍接受的范型，这是为什么我们不赞同通过标准化一种知识体来推动政府颁发执照、规范软件工程师的原因。根据我们的经验，对于要推行的最佳方法，还有一些有显著不同的有说服力的观点。

我们要非常明确地提出这些观点。我们经常对我们非常尊重的人们的工作提出批评。在很多情况下，我们所批评的工作是由我们很好的朋友完成的。不要把对某种思想的批判，错误地当作对该思想的支持者或提出者个人的攻击。

我们认为通过直接比较、对比这些观点，会使整个测试领域受益。对于测试领域来说，重要的是对我们的方法展开讨论。我们倡导进一步发展每一个主要方法范畴内的实践，最终我们都会认识到不同方法最适用的实际环境，到那时才会出现百花齐放的局面。

一些词汇的解释

以下是本书用到的一些关键词及其在本书中的含义：

我们。作者。

你们。读者。

缺陷（fault）是程序实现或设计中的错误、失误。

错误（error）在本书中就是缺陷。

失效（failure）是程序的错误行为，由包含缺陷的程序产生。

失效在一定条件（condition）下发生。例如，当程序试图进行除数是0的计算时，就会出现崩溃。在这种情况下，缺陷是允许被0除的代码，失效是崩溃。但是只有在除法中的一个关键变量取值为0时，才会看到失效。那个变量取值为0是失效发生必须满足的关键条件。

征兆（symptom）与失效类似，但是不那么严重。例如，如果程序存在内存泄漏，则程序在给出内存耗尽错误消息之前很久就开始逐渐慢下来。这种降速就是不能直接看得到的内部问题（内存短缺）的征兆。

程序错误（bug）是含义很广的词，可以指任何软件问题。报告程序错误的人描述的可能是缺陷、失效或使程序对项目相关人员的价值减弱的局限性。

如果把质量（quality）定义为对某人的价值（Weinberg，1998，2.i.），那么**错误报告**（bug report）就是一种陈述，说明有人认为由于被描述为程序错误的东西使该产品的价值降低。

过失（defect）有一种法律指责含义，表示“产品肯定出现了问题”。有些公司不允许这样的词出现在错误报告或与程序错误有关的备忘录中。

有些公司喜欢用**异常**（anomaly）、**问题**（problem、issue）替代程序错误。

当报告一个程序错误之后，程序员（或“变更控制委员会”）会将其更正，或决定不更正。他们会把这个程序错误标识为**已解决**（resolved）（已更正、推迟、不可重现、设计如此等）。

黑盒测试（black box testing）。通过向程序送入输入，并检查其输出，来测试程序的外部行为。在一般的黑盒测试中，测试员没有代码内部知识，但是更熟悉显式和隐式产品需求，例如使用程序的方式，有可能输入程序的数据类型，与软件试图解决或帮助解决的问题有关的任何管理问题，以及软件将使用的硬件和软件环境。

行为测试（behavioral testing）。测试程序的外部行为，类似黑盒测试，但是尽可能利用测试员能够得到并与测试有关的程序内部知识。

功能测试（functional testing）。黑盒或行为测试。

白盒或玻璃盒测试（white box或glass box testing）。利用程序内部知识的测试。

结构测试（structural testing）。关注程序内部结构的白盒测试，例如从一个决策或行动到下一个决策或行动的控制流。

冒烟测试或版本确认测试（smoke testing或build verification testing）。用于软件新版本的标准测试包。这种测试检查版本基本功能是否稳定，或检查是否遗漏关键功能，是否满足基本要求。如果版本没有通过这些测试，则不再进行进一步的测试，而是继续测试老版本，或等待新版本。

项目经理（project manager）。负责按时在预算限度内交付恰当产品的人。有些公司将上述工作划分给程序经理和开发经理完成。

最大整数（MaxInt）。用户平台或程序员开发语言可以使用的最大整数。大于最大整数的数不能作为整数存储。

客户（client）。公司要满足其利益要求的人。在一定程度上可能包括与项目有关的所有人，以及产品的最终用户。

致　谢

如果没有很多人的支持和帮助，本书是不可能出版的。我们要感谢Lenore Bach、Jon Bach、Becky Fiedler、Leslie Smart和Zach Pettichord，感谢他们在此书出版工作中对我们的支持、理解和帮助。我们要感谢Pat McGee在关键时刻给予我们的帮助。

评阅人对初稿提出了仔细、深刻的反馈意见，我们受益不小。我们在书中补充了评阅人给出的一些例子和观点。我们要感谢Stale Amland、Rex Black、Jeffrey Bleiberg、Hans Buwalda、Ross Collard、Lisa Crispin、Chris DeNardis、Marge Farrell、Dorothy Graham、Erick Griffin、Rocky Grober、Sam Guckenheimer、George Hamblen、Elisabeth Hendrickson、Doug Hoffman、Kathy Iberle、Bob Johnson、Karen Johnson、Ginny Kaner、Barton Layne、Pat McGee、Fran McKain、Pat McQuaid、Brian Marick、Alan Myrvold、Hung Nguyen、Noel Nyman、Erik Petersen、Johanna Rothman、Jane Stepak、Melora Svoboda、Mary Romero Sweeney、Paul Szymkowiak、Andy Tinkham、Steve Tolman和Tamar Yaron。

在“测试自动化Los Altos研讨会”、“启发式与探索测试技术研讨会”、“软件测试经理圆桌协会”、“测试自动化Austin研讨会”、“软件测试模式研讨会”，以及很多其他研讨会、会议、课堂上和工作中，本书得益于与很多热心探索更好软件测试方法的人的讨论。我们在这里一并向他们致谢。

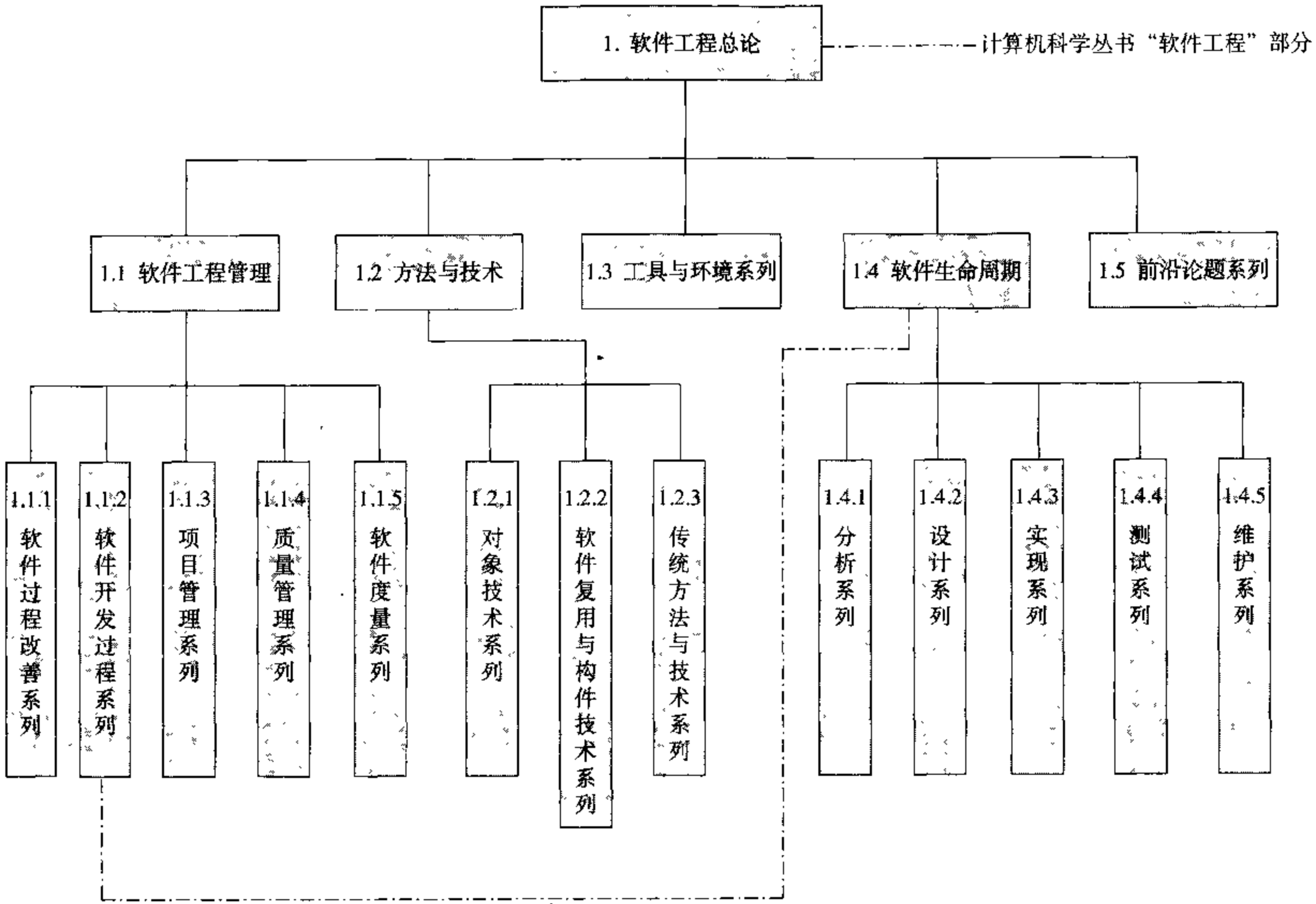
软件工程技术丛书书目

丛书编号	英文书名	中文书名	作　者
1	Object-Oncnted and Classical Software Engineering, 5E	面向对象与传统软件工程(原书第5版)	Stephen R. Schach
1	Object-Oriented Software Engineering	面向对象的软件工程	Ian Sommerville
1	Software Engineering: A Practitioner's Approach, 5E	软件工程:实践者的研究方法(原书第5版)	Roger S. Pressman
1	Software Engineering, 6E	软件工程(原书第6版)	Ian Sommerville
1	Software Engineering with Java	软件工程:Java语言实现	Stephen R. Schach
1	Project-Based Software Engineering:An Object-Oriented Approach	基于项目的软件工程:面向对象研究方法	Evelyn Stiller
1.1.1	Software Process Improvement: Practical Guidelines for Business Success	软件过程改进	Sami Zahran
1.1.1	Making Process Improvement Work	软件过程改进简明实践	Neil S. Potter
1.1.2	The Road to the Unified Software Development Process	统一软件开发过程之路	Ivar Jacobson
1.1.2	The Unified Software Development Process	统一软件开发过程	Jacobson/Booch/Rumbaugh
1.1.2	The Rational Unified Process: An Introduction , 2E	Rational统一过程引论(原书第2版)	Philippe Kruchten
1.1.2	UML and The Unified Process: Practical Object-Oriented Analysis & Design	UML和统一过程: 实用面向对象的分析和设计	Jim Arlow
1.1.3	Managing Global Software Projects: How toLead Geographically Distributed Teams,Manage Processes and Use Quality Models	全球化软件项目管理	Gopalaswamy Ramesh
1.1.3	Software Project Management: A Unified Framework	软件项目管理:一个统一的框架	Walker Royce
1.1.3	How to Run Successful Projects III The Silver Bullet	成功的软件项目管理:银弹方案(原书第3版)	Fergus O'Connell
1.1.3	Successful Software Development, 2E	成功的软件开发(原书第2版)	Scott E. Donaldson
1.1.3	Six Sigma Software Development	6σ软件开发	Christine B. Tayntor
1.1.3	IT Project Management: On Track from Start to Finish	IT项目管理: 从开始到结束的历程	Joseph Phillips
1.1.3	Successful IT Project Delivery: Learning the lessons of Project Failure	IT项目成功交付的秘诀	David Yardley
1.1.3	Software Project Management, 3E	软件项目管理(原书第3版)	Bob Hughes
1.1.3	Architecture-Centric Software Project Management	软件项目管理实用指南: 以体系结构为中心	Daniel J. Paulish
1.1.4	Handbook of Software Quality Assurance, 3E	软件质量保证(原书第3版)	Gordon G. Schulmeyer
1.1.4	Software Reliability Engineering	软件可靠性工程	John Musa
1.1.4	Implementing ISO 9001:2000 The Journey from Conformance to Performance	ISO 9001:2000 实施指南	Tom Taimubg
1.1.4	CMMI Distilled: A Practical Introduction to Integrated Process Improvement	CMMI精粹:集成化过程改进实用导论	Dennis M. Ahern
1.1.4	CMM Implementation Guide	CMM实施与软件过程改进	Kim Caputo
1.1.4	Implementing the Capability Maturity Model	CMM实施指南	James R.Persse
1.1.4	Object-Oriented Defect Management of Software	面向对象的软件缺限管理	Houman Younessi
1.1.4	Metrics and Models in Software Quality Engineering	软件质量工程:度量与模型	Stephen H. Kan
1.1.4	Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software	软件性能工程	Connie U. Smith

丛书编号	英文书名	中文书名	作　者
1.1.4	Solid Software	Solid Software	Shari Pfleeger
1.1.4	Peer Reviews in Software: A Practical Guide	同级评审	Karl E. Wiegers
1.1.5	Practical Software Measurement	实用软件度量	John McGarry
1.1.5	Software Metrics: A Rigorous and Practical Approach, 2E	软件度量(原书第2版)	Norman E. Fenton
1.1.5	Software Assessments, Benchmarks, and Best Practices	软件评估、基准测试与最佳实践	Capers Jones
1.2.1	The Object Primer: The Application Developer's Guide to Object Orientation and the UML, 2E	面向对象软件开发教程(原书第2版)	Scott W. Ambler
1.2.1	UML and C++: A Practical Guide to Object-Oriented Development, 2E	C++面向对象开发(原书第2版)	Richard C.Lee
1.2.1	Object-Oriented Methods: Principles & Practices, 3E	面向对象方法:原理与实践(原书第3版)	Ian Graham
1.2.1	Principles of Object-Oriented Software Development, 2E	面向对象软件开发原理(原书第2版)	Anton Eliëns
1.2.1	Object Solutions: Managing the Object-Oriented Project	面向对象项目的解决方案	Grady Booch
1.2.1	An Introduction To Object-Oriented Programming, 3E	面向对象程序设计导引(原书第3版)	Timothy Budd
1.2.1	The Object Advantage: Business Process Reengineering with Object Technology	对象优势:采用对象技术的业务过程再工程	Ivar Jacobson
1.2.1	The Unified Modeling Language User Guide	UML用户指南	Booch/Rumbaugh/Jacobson
1.2.1	The Unified Modeling Language Reference Manual	UML参考手册	Rumbaugh/Jacobson/Booch
1.2.1	Applying UML and patterns: An Introduction to Object-Oriented Analysis and Design, 1E	UML和模式应用(原书第1版)	Craig Larman
1.2.1	Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process , 2E	UML与模式应用(原书第2版)	Craig Larman
1.2.1	Object-Oriented Analysis and Design with Applications, 2E	面向对象分析与设计(原书第2版)	Grady Booch
1.2.2	Software Reuse: Architecture, Process and Organization for Business Success	软件复用:结构、过程和组成	Ivar Jacobson
1.2.2	Software Reuse Techniques: Adding Reuse to the Systems Development Process	软件复用技术:在系统开发过程中考虑复用	Carma McClure
1.2.2	Practical Software Reuse: Strategies for Introducing Reuse Concepts in Your Organization	实用软件复用方法	Donald J. Reifer
1.2.2	Large-Scale Component-Based Development	大规模基于构件的软件开发	Alan W. Brown
1.2.2	Component-based Product Line Engineering with UML	基于构件的产品线工程:UML方法	Colin Atkinson
1.4.1	Object-Oriented Analysis & Design	面向对象的分析与设计	Andrew Haigh
1.4.1	Analysis Patterns: Reusable Object Models	分析模式:可复用的对象模型	Martin Fowler
1.4.1	Requirements Analysis and System Design: Developing Information Systems with UML	需求分析与系统设计	Leeszek A. Maciaszek
1.4.1	Systems Analysis and Design in a Changing World	系统分析与设计	John W. Satzinger
1.4.1	Advanced Use Case Modeling, Vol I: Software Systems	高级用例建模 卷I:软件系统	Frank Armour
1.4.1	Requirements Engineering: A Good Practice Guide	需求工程	Ian Sommerville
1.4.1	Software Requirements and Estimation	软件需求与估算	Swapna Kishore
1.4.1	Effective Requirements Practices	有效需求实践	Ralph R.Young
1.4.1	Applying Use Cases. A Practical Guide, 2E	用例分析技术(原书第2版)	Geri Schneider
1.4.1	Managing Software Requirements	软件需求管理:统一方法	Dean Leffingwell
1.4.1	Writing Effective Use Cases	编写有效用例	Alistair Cockburn
1.4.1	Problem Frames Analyzing and Structuring Software Development Problems	Problem Frames Analyzing and Structuring Software Development Problems	Michael Jackson

丛书编号	英文书名	中文书名	作　者
1.4.2	Object-Oriented Software Construction, 2E	面向对象的软件结构(原书第2版)	Bertrand Meyer
1.4.2	Pattern-Oriented Software Architecture, Vol 1: A System of Patterns	面向模式的软件体系结构 卷1:模式系统	Frank Buschmann
1.4.2	Pattern-Oriented Software Architecture, Vol II Patterns for Concurrent and Networked Objects	面向模式的软件体系结构 卷2:用于并发和网络化对象的模式	Douglas Schmidt
1.4.2	Server Component Patterns	面向模式的软件体系结构 卷3:服务器组件模式	Markus Volter
1.4.2	DesignPatterns: Elements of Reusable Object:-Oriented Software	设计模式:可复用面向对象软件的基础	Gamma/Helm/Johnson /Vlissides
1.4.2	Patterns of Enterprise Application Architecture	企业级应用体系结构模式	Martin Fowler
1.4.2	AntiPatterns and Patterns in Software Configuration Management	软件配置管理中的模式与反模式	William J Brown
1.4.2	Software for Use, A Practical Guide to The Models and Methods of Usage-Centered Design	面向使用的软件设计	Larry L Constantine
1.4.2	Software Architecture: Organizational Principles and Patterns	软件架构: 组织原则与模式	David Dikel
1.4.2	The UML Profile for Framework Architectures	框架体系结构的UML档案	Marcus Fontoura
1.4.2	Software Architect's Profession, An Introduction	软件架构师入门必读	Marc Sewell
1.4.2	Business Modeling with UML: Business Patterns at work	UML业务建模: 实用业务模式	Hans-Erik Eriksson
1.4.3	Software Fabrication:Automating Application Development	软件构造 自动化软件开发	Jack Greenfield
1.4.3	Building J2EE Applications With The Rational Unified Process	用RUP构建J2EE 应用程序	Peter Eeles
1.4.3	Programming from Specifications	从规范出发的程序设计	Carroll Morgan
1.4.4	Testing IT, An Off-the-Shelf SoftwareTesting Process Handbook	实用软件测试过程之路	John Watkins
1.4.4	Lessons Learned in Software Testing	软件测试经验与教训	Cem Kaner
1.4.4	Testing-Compute: Software: The Bestselling Software Testing Book Of All Time, 2E	计算机软件测试(原书第2版)	Cem Kaner
1.4.4	Software Testing in the Real World, Improving the Process	软件测试过程改进	Edward Kit
1.4.4	Effective Methods for Software Testing, 2E	有效的软件测试方法(原书第2版)	William E Perry
1.4.4	Beta Testing for Better Software	软件Beta测试	Michael R. Fine
1.4.4	A Practical Guide to Testing Object-Oriented Software	面向对象的软件测试	John D. McGregor
1.4.4	Managing the Testing Process, 2E	测试过程管理(原书第2版)	Rex Black
1.4.4	Software Testing, A Craftsman's Approach, 2E	软件测试(原书第2版)	Paul C. Jorgensen
1.4.4	Just Enough Software Test Automation	软件测试自动化	Daniel J Mosley
1.4.4	The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing	软件测试实用技术	Brian Marick
1.5	JAVA Tools for Extreme Programming: Mastering Open Source Tools, Including Ant,JUnit, and Cactus	极限编程的JAVA工具	Richard Hightower
1.5	Agile Software Development Ecosystems	敏捷软件开发生态系统	Tom DeMarco
1.5	Agile Modeling: Effective Practices For eXtreme Programming and The Unified Process	敏捷建模:极限编程和统一过程的有效实践	Scott W. Ambler
1.5	Model-Driven Development: Automating Component Design, Implementation, and Assembly	模式驱动的软件开发	David Frankel
1.5	A Practical Guide to Feature-Driven Development	特征驱动开发方法 原理与实践	Steve R. Palmer

软件工程技术丛书结构图



目　录

第 1 章

测试员的角色

测试员要在项目中起什么作用？这是本章要讨论的问题。像有关测试的很多问题一样，这个问题初看起来答案很明显、很平凡，但其实不然。

一个角色就是一种关系。这意味着人们不能控制自己的角色，但是可以协商。别人期望从测试员那里得到的可能并不合理。当测试员由于低质量的产品而受到指责时（这种事时有发生），不管是谁指责，可能都存在分不清角色的问题。也许他们认为测试员的工作，就是在产品交付之前使用“质量魔术棒”敲打产品，他们也许认为测试员敲打得还不够狠。

当测试员清楚了自己的角色之后，在协商角色内容时，就有了在可能出现的任何情况下确立对自己预期的基础。但是，即使是清晰和恰当的测试角色也是一种苛求。

测试员是项目的前灯

一个项目就像是一次陆上旅行。有些项目很简单、很平常，就像是大白天开车去商店买东西。但是大多数值得开发的项目更像是夜间在山里开越野卡车。这些项目需要前灯，而测试员要照亮前面的道路，使程序员和经理尽管还在拿着地图争吵，但是至少可以看清他们在哪儿，要从什么样的路面上开过去，离悬崖峭壁有多远。每个公司测试团队的具体使命都不尽相同，不过在这些细节背后的要素都是一样的。测试就是要找到信息，有关项目或产品的关键决策都是根据这些信息做出的。

测试员的使命决定要做的一切

测试员的使命，可能要取决于自己的行业、公司、项目或团队的个性，测试项目也千差万别。把测试作为一种工艺发展的挑战，一直是建立测试实践对话所面临的困难，这种测试实践要跨越我们之间的文化和技术差异。这些差异中的很多内容，决定了测试团队的不同使命。例如，在有些测试机构中，测试计划只是用来为测试员提供帮助的工具。测试计划可能写在餐巾纸上，且仍然有效。而另一些机构作为产品来编写测试计划，必须随软件一起交付。他们的测试计划必须遵循严格的格式和内容要求。

以下任何要求都可能决定测试员的使命。读者期望的是哪种要求呢？

- 快速找出重要软件问题。
- 对产品质量提出总体评估。
- 确认产品达到某种具体标准。
- 帮助客户改进产品质量和可测试性。
- 保证测试过程能够达到可分清责任的标准。
- 就测试和与测试员协作方式培训客户。
- 采用特定的方法集或遵循特定的规则集。
- 帮助预测和控制支持成本。
- 帮助客户改进其过程。
- 以最小化成本、时间或尽可能减少副作用的方式，完成自己的工作。
- 为满足特定客户要求，完成所有必要的工作。

如果测试员将时间和精力都投入到客户并不关心的需求上，就会冒做无关工作或生产率低的风险。测试员要与自己的经理协商使命问题，并明确使命。如果不能就使命达成一致意见，就不会有做任何工作的好基础。

*如果测试员不知道该做什么怎么办？*一种回答是评审使命。这样做可以找出自己的核心问题。如果测试员明确自己的测试使命，就可以为自己的工作辩护，并明确地确定下一步要做什么。测试员还可以用简单的描述，向其他人解释自己的角色。如果由于某种原因不能完成自己的使命，应该立即把这个问题汇报给管理层。

*如果测试员确切地知道要做什么该怎么办？*经常重新考虑自己的使命，保证自己的计划不会由于过于偏重测试问题的一个方面，而忽略其他方面。

测试员为很多客户服务

测试是一种服务角色，要乐于接受这种角色，因为测试员提供的服务是至关重要的。服务就意味着有客户，即要被服务的人。测试员是否成功，主要是看其是否很好地满足了客户的要求和最佳利益。这不会太难，不过测试会有很多客户，这些客户都有自己的需要，而且他们的各种需要不一定一致。

- 项目经理。项目经理有资格了解测试员的工作进展并施加影响。测试员根据要求向其报告工作状态，迅速报告重要问题，并不要成为项目的瓶颈，从而为项目经理提供服务。指挥项目是项目经理的特权。测试员的责任就是告诉项目经理自己能做什么，不能做什么，有关项目的决策和条件会对测试产生什么影响。
- 程序员。通过尽可能迅速地提供好的错误报告，使得程序员的工作更容易一些。努力提高自己的技能并了解产品，以免用错误的或用毫无意义的报告浪费程序员的时间。如果测试员可以做到这一点，就可以赢得更多的信任，而这种信任又可以转化为支持和影响。
- 技术文档编写员。与测试员一样，负责编写文档和在线帮助的技术文档编写员也不能得到产品的完整信息。测试员可以帮助他们理解产品到底怎样发挥效能，并为其指出文档中的错误。技术文档编写员也会帮助测试员。当技术文档编写员研究产品，以及必须阅读文档的用户会怎样使用产品时，会了解到一些测试员不知道的信息。如果测试员与技术文档编写员有很好的关系，编写员就会告诉测试员有关产品的新特性、新用法、测试计划中的漏洞和他们所发现的软件问题。这些问题中的一部分永远也不会被报告，除非某个文档编写员知道哪个测试员关心这些程序问题。
- 技术支持员。遗留在产品中的任何问题都会为技术支持员带来负担。测试员通过告诉技术支持员可能会给用户带来麻烦的产品问题，向其提供服务。如果测试员在开发期间与技术支持员一起工作，有时技术支持员会帮助测试员找出应该更正的软件问题。测试员也应该通过研究现场发现的难题，为技术支持员提供帮助。通过这种方式，能够把测试员与技术支持员拉得更近，进而与客户也更近了。
- 市场开发员。市场开发员需要了解产品中任何与产品应该提供给客户的关键利益不一致的地方。对于程序员来说是很小的程序问题，对于市场开发员来说可能会是至关重要的问题。他们也许能意识到这种程序问题会使客户较难完成某种重要任务。此外，通过评审市场开发计划文档或描述，测

试员可以帮助市场开发员对产品能力有更精确的认识。

- 管理层和项目相关人员。测试员服务于公司业务，这也是为什么测试员必须小心，不要像个质量狂，而不是通情达理的人的原因。特别是到了项目要结束的时候，测试员要以兼顾公司短期和长期利益的方式完成自己的职责。要以明确、简洁的词汇编写测试状态报告，以便执行经理能够感到有做出决策的依据。
- 用户。在测试员的心中，要想着将要使用该产品的人。当然，用户的满意是项目的最高利益。但是还要考虑满足主要用户对项目团队的特殊要求。

以上列出的各条没有什么特别顺序，不过在实际项目中可能有一定顺序，因此要认真研究，找出对项目最重要的人，找出要服务的人。这是做好测试工作的第一步。

测试员发现的信息会“打扰”客户

测试团队的使命包括（或应该包括）根据客户对价值的定义，通知客户有关威胁产品价值的任何信息。如果发现即使产品能够按意图运行，但是仍然达不到所要求的价值，则测试员有责任报告。如果客户忽视报告，那是他们的权力。

迅速找出重要程序问题

测试员的使命很可能包括找出重要的（与无意义相反）程序问题，而且要迅速找出。如果是这样，那么这对测试员所执行的测试意味着什么呢？

- 首先测试经过变更的部分，然后测试没有变化的部分。修改和更新都意味着新的风险。
- 首先测试核心功能，然后测试辅助功能，测试产品所完成的关键和常用功能，测试完成产品基本任务的功能。
- 首先测试能力，然后测试可靠性。先测试每个功能是否完全能用，然后再深入检查任何一个功能在很多不同条件下表现如何。
- 首先测试常见情况，然后测试少见情况。使用常用的数据和使用场景。
- 首先测试常见威胁，然后测试罕见威胁。用最有可能出现的压力和错误情况进行测试。
- 首先测试影响大的问题，然后测试影响小的问题。测试在出现失效的情况

下会产生大量破坏的产品部件。

- 首先测试最需要的部分，然后测试没有要求的部分。测试对团队其他人有重要意义的任何部分的任何问题。

测试员如果对产品、产品必须与之交互的软件和硬件以及将使用软件的人越了解，越有可能更快地找出重要问题。应好好研究这些方面的内容。

跟着程序员走

为程序员提供支持，很可能是测试员使命的关键部分。在测试员测试程序员正在编写或刚刚完成的程序时，测试员的反馈有助于提高程序员的工作效率。程序员交付软件后，应该马上测试；程序员修改代码后，应该马上测试所做的变更。尽可能建立最短、最快的反馈环路。当程序员正在苦苦地思索测试员刚刚发现的程序问题时，测试员又开始寻找更多的程序问题。（对于测试员来说，）理想情况是，程序员为了修改测试员找出的程序问题忙得团团转，使程序员，而不是测试员，成为项目的瓶颈。

询问一切，但不一定外露

不问问题当然可以测试，但是不可能测试得好。问问题是测试员对项目发挥作用的基础。不问问题，测试就没有目标，就是呆板、机械的。不过很直白的问题会刺激别人，常常使人产生顾虑。

问题就像是一剂猛药，最好采用低剂量，或与饭一起吃（即结合其他沟通形式）。幸运的是，这样问问题的价值并不低于直白地发问。测试员所想到的任何问题都会有助于启发自己的思想，最终产生对问题的至关重要的认识。

如果测试员在测试时发现对产品提不出问题，那么还是先停下来。

测试员关注失效，客户才能关注成功

测试是项目团队中惟一不直接关注成功的角色。其他所有人都在创造什么，或创造性地指导创造。但测试员却是消极的。测试会是一种沉闷的工作，几乎像希腊神话所说："测试者在孤岛上，注定要不停地寻找不会存在、也不应该存在的东西，深信成功会为神带来不幸。"

重新定义比较积极的测试员使命是错误的，例如确认程序正常。即使“确认程序正常”作为使命交给测试员，测试员也要忠告客户，这样的确认是不可能的。这种确认成本极高。除非运行所有可能的测试，否则就不能证明程序正常。测试员只能够说：“就我所执行的测试来说，没有发现产品不正常。”但是反过来的确认就非常经济了：只需一个测试，就可以说明产品不正常。

测试员关注失效，是因为这可以增加发现失效的机会。用自己全部的创造力和技能，寻找产品中的关键问题。如果测试员没有找到关键问题，程序员就不能改正，以后用户可能会替测试员找到。通过发现程序中客观存在的问题，测试员能够帮助项目团队更加了解自己的技能和产品风险，帮助他们将产品做得更好，更具可支持性，在市场中有可能更成功。

不会发现所有程序问题

测试员的任务就是找出并报告重要的程序问题，但是不会发现所有的程序问题。为了发现全部程序错误，测试员必须检查所有可能有问题的地方，要在所有可能发生的不同条件下观察这些地方，还需要一种十分可靠的方法，当所有类型的程序错误发生时，都能够识别出来。如果测试员认为自己能够做到这些，那么要么产品非常简单，要么测试员的想像力太差。

知道并承认自己不能做所有的事之后，测试员必须选择如何使用自己的时间。

当心“完备的”测试

有一些测试员承认自己不知道是否发现了产品中的全部问题，但仍然不准确地讨论结束测试的含义。“对这个产品我需要测试5天”可以解释为，他可以在5天之内对产品进行完备的测试，也可能意味着他会在5天之内发现所有问题。完备性常常是隐含地表示出来的，而不是明说出来的。不管是哪种情况，这都是必须小心对待的概念。请考虑完备测试可能的含义：

- 完全发现了产品中每个问题。
- 完全检查了产品的每个部分。
- 完成了自认为是有用和经济的测试。
- 尽自己所能，完全达到了项目团队制定的目标。

- 完成了约定的测试。
- 完成了在一定条件下人所能够测试的所有内容。
- 完成了自己知道如何测试的所有内容。
- 完成了自己所承担的测试部分，不考虑其他人的工作。
- 完成了对产品很广、但是不深的测试。
- 完成了对产品的一种测试。
- 用完了分配给测试的时间。

如果测试员小心地澄清自己的意思，不要有“完备”、“完成”、“结束”等含义，则可能会很安全，由于有些工作没有做而受到的责备可能更少，在受到责备时可以更好地为自己辩护。请注意，“完备”的定义并不是在项目一开始就能够最终确定的，随着测试项目的进展，随着新测试任务的突然出现，需要重新考虑。

为了解决在完备性上的普遍沟通问题，可让客户详细了解测试过程。总结自己实施的测试，以及为什么值得实施这些测试，并告诉客户自己没有做的其他值得做的测试，以及为什么没有做这些测试。

通过测试不能保证质量

测试员太容易把自己看作是质量卫士了。但是测试员既不会提高质量，也不能降低质量。测试员表现出好像是自己“制服”了产品，但实际上产品到测试员手中时已经被制服。质量来源于构建产品的人，有时这对他们来说是要背负的沉重负担。测试员使命中的很大一部分，就是帮助他们更有效地对付真正的负担。如果测试员自认为是项目团队中惟一关心交付好产品的人，就不能很好地完成这部分使命。

测试小组的任务可能叫做“质量保证”，但是测试经理可别真的也这样认为。测试员的测试和错误报告提供促进项目质量保证的信息，但是这种保证要来自整个项目团队。

永远别做看门人

有些测试员梦想对产品的发布具有否决权。不妨给他们点教训，让他们梦想成真。问题是当测试员获得控制产品发布的权力之后，同时也承担了产品质量

的全部责任，团队其他成员可以为此放松一点，也许还会大大地放松。如果问题逃过测试，走出项目团队大门，团队其他人员会（并且也将）耸耸肩膀，并责备测试员。谁让测试员交付有问题的产品来着。另一方面，如果测试员延误发布，作为这种质量狂，会被严厉追究并承担很大压力。

最终，要由控制项目、条件最好的人承担发布产品的责任。但是我们所看到的大多数非常有效的项目团队，都采用某种方式的集体决定是否发布产品。如果读者被授权决定产品的发布，我们建议毫不犹豫地立即坚持与项目团队的其他角色一起分担这种权力。

当心测试中的不关我事理论

测试是如此复杂，与其他项目活动如此密切关联，以至于测试员总想通过采用狭隘的测试使命观进行更好的控制。有些测试员认为，自己的使命就是找出产品和规格说明之间的差别。任何超出这个范围的问题，例如可使用性问题、需求问题、数据质量和可支持性问题，都“不关我的事”。我们强烈要求这样的人把眼光放宽一些。所有其他事情都一样，测试员的使命应该是尽其所能，通知团队可能会对产品的价值产生消极影响的所有问题。因此，优秀的测试团队应由理解项目总体问题的不同类型的人员组成：产品将怎样设计、制造、市场开发、销售、使用、服务和升级。

说“不关我的事”的另一个原因是测试员处于困难的测试环境。负责程序设计的同事编写的规格说明可能很差，可能交付代码太迟，以至于没有时间执行合理的测试过程。他们可能声称测试员发现的重要问题实际上是测试员自己的主观臆想。在这种环境下，很想拒绝测试。测试员可以说，解释模糊的规格说明或在这样短的时间内进行某种测试不是自己的事。如果情况很严重，这样做也许不错，但是首先应该考虑一下自己的期望是否实际，考虑一下是否有其他方式能够得到自己所需要的条件。如果测试员接受这样的哲学，认为自己的工作就是付出合理的努力去适应和灵活应对各种情况，则程序员更有可能把测试员当作恩赐，而不是负担。而这反过来又会促使他们把帮助测试员看作是自己的事。

当心成为过程改进小组

有时测试员会对查找问题感到厌烦，并考虑是否最好能够防止问题的发生。

也许程序员更仔细一些，问题就会更少一些。就问题而言，这确实很有意义。但是同样有意义的是，满怀好意地告诉自己所爱的人怎样生活得更好。如果尝试一下就会知道，好的忠告并不总能被真正接受。问题不在于是否能够认识到，而在于感情。不管过程改进要干什么，它永远都会涉及感情。

即使测试员通过测试着手推进质量改进有管理层强有力的支持，但是团队其他成员同时会有很多方法避开测试员的努力，并使测试员看起来无能。是的，如果过程改进是整个团队的工作，则测试员可以成功地参与工作，并获得成功，但是我们强烈要求测试员不要试图把测试小组“提升”到扮演过程批评角色。这样做是愚蠢的。

别指望任何人会理解测试，或理解测试员需要什么条件才能搞好测试

作为测试员的你在读本书时，别指望其他人也会读它。让客户了解为了有效地完成测试工作都需要什么条件，完全要靠测试员自己。测试员要受管理层和程序员决策的很大影响。如果他们的计划不明确，或设计出的产品很难测试，测试工作就会很难进行。测试员也许不会得到想要的一切，但是测试员可以向管理层和程序员提供帮助自己的机会。

管理层和程序员并不是不关心测试或质量，他们也许只是不理解自己的行动会对测试过程产生的影响。测试工作的一个重要部分就是向客户解释测试。测试员的解释就像是流感疫苗，有利于健康而又不那么痛苦，但是疫苗的作用会逐渐衰退，必须一遍又一遍地解释。

第 2 章

按测试员的方式思考

测试员有很多不同的背景，测试团队是多元化的集体，但是大多数人都同意：测试员的思考方式是不同的。怎么不同？有人说测试员是“消极”思维者。测试员会抱怨这种说法，认为自己喜欢征服，他们在报告坏消息时有一种特别的兴奋感。这是一种普遍观点。我们提出另一种观点。测试员并不抱怨，他们提供的是证据。测试员并不喜欢征服，他们喜欢打破产品没有问题的幻觉。测试员并不喜欢发布坏消息，他们喜欢把客户从虚假信念中解放出来。我们的观点是，按测试员的方式思考意味着实践认识论。测试运用的是认识论，不是靠傲慢或谦卑。

本章旨在把测试员的大脑开发成经过仔细调谐的推理机器。请记住：要用精神力量做好事，而不做坏事。

经验 16

测试运用的是认识论

读者看到这个题目会说：嘿，回来！我们在这里不是要讨论对电影明星的新崇拜。请相信我们，认识论是能够帮助测试员更好测试的一个哲学分支。

认识论研究如何认识所了解的东西：研究证据和推理。这是科学实践的基础。研究认识论的人有科学家、教育家和哲学家，当然还有精英级的软件测试员。学习认识论的学生研究科学、哲学和心理学，目标是了解怎样才能改进我们的思维。我们使用的术语比经典定义要宽，以便能够更多地利用批评性思维的最新成果。将认识论运用于软件测试，要问与以下类似的问题：

- 怎么知道软件足够好？
- 如果软件并不是足够好，怎样才能知道？
- 怎么知道已经完成了足够的测试？

苏格拉底早在2400年前就提倡并描述了对信念的批判性观察，因此我们把他看作是最早的认识论者。直到今天，哲学家、科学家和心理学家都还在继续研究认识论。作为测试员，这就是我们的遗产。

研究认识论有助于更好测试

直接与软件测试有关的认识论问题包括：

- 如何收集和评估证据。
- 如何进行有效的推论。
- 如何使用不同逻辑形式。
- 拥有合理的信念意味着什么。
- 形式和非形式推理之间的差别。
- 非形式推理的常见谬误。
- 自然语言的含义与模糊性。
- 如何做出好的决策。

从来也没有研究过这些问题的很多人也能测试得很好，但是如果要做得比很好还好，就要研究这些问题。研究认识论可帮助测试员设计有效的测试策略，更好地意识到自己工作中的错误，理解自己的测试能够证明什么、不能证明什么，并编写出无懈可击的测试报告。

以下是三本具有很高可读性的入门书：

- 《批判性思维的工具：心理学的元思想》（Tools of Critical Thinking: Metathoughts for Psychology）（Levy 1997）。这本书是针对精神病医生写的，但是对测试员也很有用。书中每一章都是有关更好思维的不同思想。不一定把它全读完，可以挑选任何一章阅读。
- 《思考与决策》（Thinking and Deciding）（Baron 1994）。这是讨论思维世界的一本可读性很高的普通教科书，是很好的入门书。
- 《研究的技巧》（The Craft of Research）（Booth、Colomb和Williams 1995）。这是一本有关批判性阅读和写作的很好的书籍，包括如何组织有说服力的论据。主要针对大学生读者。

认知心理学是测试的基础

如果说认识论告诉我们的是应该怎样思考，那么认知心理学告诉我们的是我

们是怎样思考的。与测试有关的一些问题包括：

- 人的感觉和记忆可靠性。
- 信念从哪里来。
- 信念如何影响人的行为。
- 做出决策所使用的偏见和捷径。
- 如何了解并分享所知道的信息。
- 如何考虑复杂事情。
- 在压力下如何思考。
- 如何识别模式。
- 如何把想法和事物分类。
- 如何注意事物之间的差别。
- 记忆事件中的失真。
- 如何重新构建部分记忆的事件（例如不可再现的程序错误）。

从来也没有研究过这些问题的很多人也能测试得很好，但是如果要做得比很好还好，就要研究这些问题。研究认知心理学有助于理解影响测试员工作成绩的因素，以及影响人们解释自己工作方式的因素。

开始研究认知心理学，不能不看《旷野中的认知》（Cognition in the Wild）（Hutchins 1995）。Hutchins研究海军航海团队，以及他们怎样协同工作。这本书的很多内容也都与软件项目和测试团队有关。

有关思考心理学的一本有用的书是《理论与证据：科学推理的能力的开发》（Theory and Evidence: The Development of Scientific Reasoning）（Koslowski 1996）。在这本书中，Koslowski研究了人们如何使用因果关系理论进行系统推理。这可以解释为什么测试不只是查看外部行为，并对照简单的预期描述进行检查。

经验 19

测试在测试员的头脑中

优秀测试和平庸测试之间的差别在于测试员如何思考：测试员的测试设计选择，解释所观察到的现象的能力，以及非常令人信服地分析描述这些现象的能力。测试的其他工作大部分是一般的办公室工作。如果看到两个测试员并排工作，不一定能看出谁的测试更好。他们工作中能够看得到的部分外表相同，这说明：

- 很多人认为测试很容易，因为可以很容易地模仿优秀测试员的外表看得到

的行为，并且他们没有好的测试的其他标准。

- 如果要成为优秀测试员，就要学会像优秀测试员那样思考，而不是模仿他们的行为。

测试需要推断，并不只是做输出与预期结果的比较

流行的观点认为，测试员只是执行测试用例，并对照预期结果比较执行结果。这种观点把测试看作是简单的比较活动，没有看到一些聪明人必须设计测试，并确定预期输出。想想看，测试设计人员几乎从来没有得到过应该测试什么的权威指导，更不要说应该期望什么了。可以得到的指导是要解释的主体。在现实生活中，大多数测试设计都是基于推断，或基于与测试员的推断有关的经验。不仅如此，这些推断还要随时间发生变化。像测试员那样思考，就是要掌握探索式推断的艺术。

探索式推断听起来可能像是奇怪的想法，这意味着要以一种不能事先预测的方式，通过一种思想引出另一种思想，然后再引出下一种思想。有关探索式推断的一本很好的书是《证明与反驳：数学发现的逻辑》(Proofs and Refutations: The Logic of Mathematical Discovery)(Lakatos，1976)。关于这本书需要注意的是，Lakatos如何说明数学和科学推理过程是探索式的，而不是脚本化的。甚至数学家也是积极探索地推理，而不是通过运用枯燥的公式。他们像测试员那样思考！

优秀测试员会进行技术性、创造性、批判性和实用性地思考

各种类型的思考都要考虑测试的实施。但是我们认为需要提出四种主要思考：

- 技术性思考。对技术建模并理解因果关系的能力。这包括诸如相关技术事实的知识和使用工具并预测系统行为的能力。
- 创造性思考。产生思想并看到可能性的能力。测试员只能以能够想像得到的方式进行测试，只能寻找猜想会存在的问题。
- 批判性思考。评估思想并进行推断的能力。这包括在自己的思考中发现并消除错误的能力，将产品观察与质量准则关联起来的能力，以及针对特定信念或所建议的行动过程构建有说服力的测试用例的能力。

- 实用性思考。把想法付诸实施的能力。这种能力包括诸如运用测试工具，并使测试手段和力量与项目范围适应的技能。

总之，像测试员那样思考，会最终导致相信事物可能不像外表看起来那样。不管事物是怎样的，都可能有差别。我们发现，当测试过程以最具破坏性的方式失败时，根本原因最有可能是视野狭窄。换句话说，这不是运行了一万个测试，而本来应该运行一万零一个的问题；问题是没有想像出测试的总体大纲，没有做即使有两倍时间和资源也不会做的测试。

黑盒测试并不是基于无知的测试

黑盒测试意味着产品内部知识在测试中不起重要作用。大多数测试员都是黑盒测试员。为了做好黑盒测试，就要了解用户，了解他们的期望和需要，了解技术，了解软件运行环境的配置，了解这个软件要与之交互的其他软件，了解软件必须管理的数据，了解开发过程，等等。黑盒测试的优势在于测试员可能与程序员的思考不同，因此有可能预测出程序员所遗漏的风险。

黑盒测试强调有关软件的用户和环境知识，这一点并不是所有人都喜欢的。我们甚至把黑盒测试描述为基于无知的测试，因为测试员自始至终都不了解软件内部代码。我们认为这反映出对测试团队角色的根本误解。我们不反对测试员了解产品的工作原理。测试员对产品了解得越多，了解产品的方式越多，越能够更好地测试它。但是，如果测试员主要关注的是源代码，以及能够从源代码导出的测试，则测试员所做的工作也许就是程序员已经做过的，并且测试员关于这些代码的知识要少于程序员。

测试员不只是游客

测试员对产品做的大量不是测试的事，有助于测试员对产品的了解。测试员可以浏览产品，看看产品由什么组成，怎么工作。这样做有很高价值，但这不能算是测试。测试员和游客之间的差别在于，测试员把精力放在评估产品上，而不只是见证产品。虽然不必事先预测产品应该表现出的行为，但是试验产品能力的活动还没有成为测试，除非而且直到测试员运用某种如果问题存在就能标识的原理或过程时，这种活动才能成为测试。

所有测试都试图回答某些问题

所执行的所有测试，都是要回答有关现实的产品和应该得到的产品之间关系的某个问题。有时测试员完全没有意识到自己在回答问题。如果测试员只是在寻找明显的问题可能还好，但是在很多情况下，问题并不会闪烁着“请报告我”的提示自己跳出来。产品的有些错误行为用户可能一眼就会看出，尽管测试员可能没有注意到。在任何测试活动中，都要问自己什么样的问题应该推动自己评估测试策略，否则就会更像是游客，而不是测试员。

所有测试都基于模型

测试员在设计测试时，头脑中可能会有一个想像的图景，也可能有功能清单或某种图表。测试员会有谁是用户、用户关心什么的一些概念。所有这些都是模型。不管模型是什么，测试都主要基于产品模型进行，而不是实际产品。有缺点的模型会产生有缺点的测试。学会一种对产品建模的新方法，就像是学会了观察产品的一种新方法。

要研究建模问题。测试员对建模艺术越精通，越能够更好地测试。有关需求分析和软件体系结构的教科书和课程会有所帮助。获得各种建模技能的一种很好方法就是研究系统的思考。请参阅《通用系统思考引论：25周年版》（An Introduction to General Systems Thinking: Silver Anniversary Edition）（Weinberg 2001）。

直觉是不错的开始，但又是糟糕的结束

测试员很想根据自己的直觉使用具体的测试数据，或判断具体的输出，即测试员自己知道的“本能感觉”，即使说不出来使用这些知识的合理性的理由。我们认为这是有用的感觉，但是只是在开始时更有用，而不是在其他时候。

除了直觉有很强的偏见这个事实之外，真正的问题还在于测试员试图让其他人（例如程序员和经理）认真地对待自己的错误报告和质量评估。除非这种发现是基于大家都有的直觉，否则测试员的工作建议很可能不被采用。

因此，我们建议把直觉用作指南，但不能用作合理性证明。当有“这是问题，因为它显然是问题”的想法时，可考虑换一种方式：“这是问题，因为我观察到

产品行为与需求X、Y和Z矛盾，而我的客户很看重这些需求。”

为了测试，必须探索

为了很好地测试产品，测试员就必须研究它，必须深入它。这是一种探索过程，即使已经有了产品的完美描述。直到通过想像或接触产品本身而探索规格说明之前，所得到的测试都会是肤浅的。即使充分研究了产品，对产品有了很深的了解，仍然要探索问题。因为所有测试都是采样，而且样本永远也不可能完备，探索式思考要在整个测试项目过程中，在寻求最大化测试价值时起作用。

这里所谓的探索，是指有目的的漫游：带着一般使命在某个空间中漫游，但没有预先确定的路线。探索包括不断学习和实践。常常需要返回、重复或在没有经过培训的人看来是浪费的其他过程。也许正因为如此，探索对于测试和对软件工程的重要性，常常没有得到重视，甚至受到这个领域的文献作者和顾问的嘲笑。

证明我们关于探索核心重要性的论断超出了本书的主题范围。生动地体验探索的一种方法，就是观察自己在不看印在盒子上的完成图的情况下，如何拼接拼图板，或玩“20点问题（Twenty Questions）”或“策划（Mastermind）”游戏。请注意，通过预先严格确定的步骤进行，在这些活动中会怎样遇到更多的困难，得到更少的回报。

有关把科学的探索用于与测试很相似的另一个领域，即社会学的讨论，请参阅《基本理论的发现：定性研究策略》（The Discovery of Grounded Theory: Strategies for Qualitative Research）（Glaser和Strauss 1999）和《定性研究的基础》（Basics of Qualitative Research）（Strauss、Anselm和Corbin 1998）第2版。如果读者喜欢统计学，可参阅《探索式数据分析》（Exploratory Data Analysis）（Tukey 1977）。

探索要求大量思索

探索就是侦查，是没有边界的搜索。可把探索看作是在太空中遨游，需要前向、后向和侧向思索。

- 前向思索。根据已知探索未知，从所看到的探索还没有看到的，注意支流和副作用。例如，看到一个打印菜单项，点击看看会发生什么。

- 后向思索。从怀疑或想像的东西返回到已知，尝试证实或否定自己的推测。例如：怀疑是否有打印这个文档的方法，于是打开菜单并检查是否有打印菜单项（Solow 1990）。
- 侧向思索。让自己的工作由于新冒出的想法而转移，然后再将探索主题返回到主线索上（de Bono 1970）。例如：这个图很有意思。嘿！我想该打印一些更复杂的图，看看会怎么样。

即使没有要测试的产品，也可以探索。可以使用同样的思索过程探索一组文档，或与程序员面谈。通过构建更丰富、更具想像力的产品模型，探索也会不断取得进展。这些模型以后会使测试员设计出有效的测试。

使用诱导推断逻辑发现推测

诱导推断（abductive inference）又叫做假设归纳（hypothetical induction），是一种测试员每天都要使用的关键推理形式的有些怪的术语：最佳解释的推理。其主要内容是：

1. 收集一些数据并要找出其中的意义。
2. 构造可能说明这些数据的各种解释。
3. 收集更多的数据，以确定或否定每种解释。
4. 从候选解释中，选择能够最一致地说明所有重要数据的解释，如果没有足够证据证实任何结论，则继续搜索。

诱导推断是科学和测试的基本方法。医生在为病人诊断时就要使用这种方法，测试员在判断产品是什么和不是什么、产品应该怎样运行或不应该怎样运行时，也要使用这种方法。如果要更好地进行诱导推断，则：

- 收集更多的数据。
- 收集更重要的数据。
- 收集更可靠的数据。
- 理解应用于数据的原因和效果。
- 找出可以说明数据的更多、更好的解释。
- 收集更多否定每个解释的数据。
- 收集更多区分解释的数据。
- 除非某个解释能说明所有重要数据，并且明显得到比其他解释更好的解释，否则不要确定解释。

诱导是寻找好的解释的一种系统化方法。尽管诱导推断过程并不提供绝对确

定性，但是在很多情况下，这都是最佳手段。

使用猜想与反驳逻辑评估产品

20世纪初，哲学家Karl Popper引入了猜想与反驳（conjecture and refutation）方法（Popper 1989），用于如何区分宗教和科学问题。这种方法基于科学家永远也不能绝对肯定任何具体事实，或关于自然的理论这个前提。任何东西都是猜想。有些猜想很强，例如重力的存在。它是猜想而不是绝对肯定的事实，是因为能够想像出新的信息，如果这种信息存在，就会使人们拒绝该猜想。Popper注意到，虽然我们不能证明猜想是真，但是却可以证明猜想是假的。因此，他提出给定猜想的置信度只能来自反驳它，但又反驳不了的努力。

这种给出猜想并尝试反驳的方法，以二种重要方式应用于测试：

- 测试的目的是显示产品失效，要比显示产品正常更有力。如果想知道产品是否能够正常运行，寻找方法反驳其正常运行，这样的测试可能更好。
- 有关软件（软件有怎样的行为、如何好等）已经牢固形成的信念应该受到质疑。这意味着应该能够想像出新的与已有信念矛盾的信息。否则，我们的信念只不过就是信仰。信仰在私人生活中是有益的，但是对测试是有害的。
- 警惕声称以超过测试员所运行的具体测试的方式，检验或确认了产品的测试。任何量的测试都不能提供产品质量的确认性。

需求是重要人物所关心的质量或条件

可以从很多种“需求”定义中选择适合测试员的定义。作为测试员，必须认识到谁的关于质量的观点最重要（并不是每个人的观点都同等重要）。然后了解对于产品他们要什么，不要什么。这种需求与软件工程的“需求”（在“需求文档”中发布的，并由有批准权限的人批准的一组陈述）和所有种类的规格说明没有差别。至于测试，产品应该具备或满足的任何质量或条件都是需求。

不同客户通过产品要得到不同的东西，他们不一定知道要什么，而且所要的东西会随时发生变化。这使测试员的工作更有意义。欢迎测试。

通过会议、推导和参照发现需求

如果期望得到一迭印刷精美、文件袋封条盖有全球有效印章的需求，那还是另找工作吧。我们所经历的最好情况，需求文档（包括所有种类的产品规格说明、用例、多媒体文档等）是不完整、不准确的，尽管需求文档提供了信息并且是有帮助的。我们所看到的最差情况，文档是不完整、不准确、没有提供信息并且是没有帮助的。

测试员把项目文档（产品的显式规格说明）看作是惟一需求来源会影响其测试过程。在我们所管理的所有测试团队中，坚持这样要求会招致反驳。

需求信息到达测试员主要有三种途径：

- 会议。找出其有关质量的意见具有影响力的人，与他们交流，了解他们最关心什么。
- 推导。通过外推已知的项目和产品其他信息，确定什么需求最重要。
- 参照。既发现显式，也发现隐式规格说明，并以此作为测试的基础。

在很多项目中，优秀测试员所使用的大多数需求要么来自推导，要么来自隐式规格说明的参照。搜寻测试所需的信息，是测试员的工作。

有一本关于这个问题的好书：《探索需求：设计之前的质量》（Exploring Requirements: Quality Before Design）（Gause和Weinberg 1989）。

既要使用显式规格说明，也要使用隐式规格说明

并不是包含测试所依赖重要信息的所有参照都是显式地提供给测试员的：

- 显式规格说明是一个有用的需求信息源，经过客户的权威确认。（“是的，这就是规格说明，是产品描述。”）
- 隐式规格说明是没有经过客户权威确认的一个有用的需求信息源。（“这不是规格说明，但是有意义。”）

隐式规格说明的威信来自其内容的说服力和可信性，而不是客户的批准。在大多数情况下，只有部分隐式规格说明与当前产品有关。隐式规格说明有很多种形式：

- 竞争对手的产品。
- 相关产品。
- 同一产品的老版本。

- 项目团队之间的电子邮件讨论。
- 顾客意见。
- 杂志文章（例如，有关产品老版本的综述）。
- 相关主题的教科书（会计书籍适用于会计应用程序）。
- 图形用户界面（GUI）风格指南。
- 操作系统（O/S）兼容性需求。
- 测试员自己的丰富经验。

当产品与显式规格说明冲突时，测试员的报告任务相对简单一些："产品违反了规格说明，因此产品也许错了。"当违反的是隐式规格说明时，测试员的报告必须详细一些："在Microsoft Office中，功能键F4固定用于重复命令。除非我们也这样定义，否则在日常工作中也使用Office的用户会感到困惑。"虽然没有人说Microsoft Office是这个产品的规格说明，但是客户可能会同意采用与Office一致的用户界面会提高可使用性。如果是这样，则Office就是这个产品的一个隐式规格说明。

有些测试员可能会问，为什么设计人员不把所有有用的信息都放入显式规格说明，使得他们不必再从隐式资源中分辨规格说明。回答很简单：虽然这样做方便了测试员，但是很昂贵，而且没有必要。客户相信测试员能够使用所需的各种参考资料快速找出重要的问题。

经验 34

"它没有问题"真正的含义是，它看起来在一定程度上满足部分需求

任何时候听到有人说"我试过了，它没有问题"、"我保证它没有问题"或"它现在更好了"，我们建议把"它没有问题"解释为"它看起来在一定程度上满足部分需求"。测试员应该立即想到的一些问题包括：

- "它"是什么？我们正在谈论的是产品的哪个部分？
- 外观是什么情况？到底观察到了什么？
- 检查了哪些需求？正确性如何？性能如何？
- 为了通过测试要在多大的程度上满足该需求？只是刚刚通过，还是超过指标很多？
- 它什么时候没有问题？测试覆盖了多大范围的条件？通过这些条件可以安全地推广到多远？

如果不愿意，可以默默地问自己。关键是如果对"它没有问题"没有进一步地限定，它就会是模糊的。测试员所认为的"它没有问题"的意义，可能与别

人的定义不同。

最后，测试员所能得到的只是对产品的印象

不管测试员对产品的质量有什么看法，都是猜想。不管猜想有多么好的支持，也不能肯定自己是对的。因此，任何时候报告产品质量状态时，都应该用有关测试方法和测试过程的已知局限性的信息，对报告进行限定。

不要将试验与测试混淆起来

试验的含义是什么？可能表示测试员执行一段探索式测试，产生一些没有文档或试验产品的临时性试验；也可能表示测试员编写一套可执行测试程序，或一套显式的测试过程；也可能表示某种高水平的测试矩阵、测试大纲或一套测试数据。

试验的概念是自包含的、实在的，与其他方便（我们通篇使用方便（convenient）概念，因为它是测试界的标准行话）试验不同，但也是受限的。关键还是测试，而不是如何将测试打成被称为试验的包。测试是任何至少包含以下四种活动的活动：

- 配置。准备要测试的产品，将其置于正确的起始状态。否则测试结果会受到不良变量的影响。
- 运行。向产品输入数据，向产品发命令，以某种方式与产品交互。否则，产品只是放在那里，测试员能够做的只是评审，而不是测试。
- 观察。收集有关产品行为信息、输出数据、系统整体状态、与其他产品的交互等方面的信息。测试员不能观察所有事物，但是没有观察的任何事物都可能使测试员看不到问题所在。
- 评估。运用规则、推理或可检测所观察到数据中存在问题的机制。否则要么不能报告问题，要么只是把数据提交给客户，由客户自己进行评估。

试验产生的可能有很多形式，不要过于关注形式，要保证有这四种活动发生。要关注执行这些活动的思考者，关注试验是否很好地完成了预想的策略和测试使命。

当测试复杂产品时：陷入与退出

有时复杂性可能是无法抗拒的。测试员的意志可能会被击垮。因此，当要测

试复杂和使人畏惧的功能集合时，可间歇进行。人的头脑具有处理复杂问题的惊人能力，但是不要指望马上就能理解复杂产品。可试着先研究复杂产品30分钟或一个小时，然后停下来干点别的。这就是陷入与退出（plunge in and quit）法。不要担心在这段不长的时间内效率不高，如果觉得问题太多，则尽快退出。

这种方法的优点是，除了选择产品的一部分并研究外，绝对不需要计划。经过几个轮次的陷入与退出，就会开始明白产品的模式和轮廓，很快就会在头脑中更系统、更具体地测试和研究策略。这种方法很神奇。最终，会掌握足够的知识以设计全面的测试计划，如果认为这些计划能够完成自己的任务。

运用试探法快速产生测试思路

试探法（heuristic）是一种经验规则，是一种基于经验做出猜测的方法。这个词源自希腊语，表示“开始发现”。试探法并不能保证得到正确的答案或最佳答案，但是很有用。最早运用试探法的著作是《如何解决它》（How to Solve it）（Polya 1957）。

由于可能的测试用例数量是无限的，因此肯定要选出在所面临的时间和预算约束条件下有效的少量测试用例。有经验的测试员会收集并共享能够改进其猜测质量的测试试探方法。一组好的试探方法有助于很快地生成测试。以下是采用试探法测试的一些例子：

- 测试边界。边界更有可能暴露规格说明的模糊问题。
- 测试所有错误消息。错误处理代码与主流功能代码相比，一般比较弱。
- 测试与程序员的配置不同的配置。程序员已经偏信自己的配置没有问题。
- 运行比较难设置的测试。在其他条件相同的情况下，易于设置的测试更有可能已经被执行过。
- 避免冗余测试。如果某个测试实际上是重复其他测试，就不会产生新价值。

为了明智地运用试探法，请注意：试探法中并没有智慧，智慧来自测试员。试探法所能够做的，只不过就是为测试员的思考提出建议。盲目使用自己并不了解的试探法并不是好的测试实践。在收集测试方法时，要了解每个方法背后的原理，以及更适用和不太适用的条件。

测试员不能避免偏向，但是可以管理偏向

测试员是有偏向的，这使得测试员选择一部分测试的可能性要比其他测试

大。如果有一个很长的编辑字段，测试员也许更可能输入诸如1111111111，而不是3287504619，因为输入字符重复的字符串，要比从0到9随机选择数字更容易。也许这是一种很小的偏向，但仍是一种偏向。更糟的偏向是，大多数测试员倾向于测试最可视的功能，不管是不是最重要的功能。此外，大多数测试员还倾向于考虑认为与自己类似的用户，倾向于使用非常简单、非常荒谬的输入，而不是具有中等复杂度的现实输入。

以下是一些常见偏向：

- 同化偏向。更有可能把未来的测试结果解释为总体上证实自己对产品的看法。
- 证实偏向。更有可能关注确实会证实自己对产品看法的测试结果。
- 可用性偏向。如果头脑中已经想到一种用户以某种方式操作的场景，则更有可能认为这种操作更常出现。
- 最初印象偏见。更信任所做的第一次观察。
- 最新印象偏见。更信任所做的最近一次观察。
- 框架效应。对错误报告的反应与措辞有很大关系，不管其真正含义如何。
- 知名偏向。把碰巧认识的用户意见放在更重要的地位。
- 表达偏向。期望较小的问题也许有较小的原因，而严重问题会有大原因。

测试员不能避免这些偏向，因为这些偏向在很大程度上已经固化在头脑中。测试员能够做的是管理偏向。例如，只需通过研究偏向并在实践中注意，这样在思考时就可以更好地进行补偿。多样化也可以抵御过强的偏向。如果测试员集体谈论测试问题，可以将一个测试员的偏向降低到最低限度。

根据定义，试探法也是一种偏向。使用试探法，是因为其偏向可以以比较好的方式引导测试员。

如果自己知道自己不聪明，就更难被愚弄

骗子说，最容易上当的人，是绝对自信不会被愚弄的人。作为测试员也可以把这条定律用于自己的工作中。证明自己容易被愚弄。做到这一点并不难，只需仔细看看自己在测试中犯的错误。任何时候都要注意其他测试员所发现的自己本来也可以发现，但是没有发现的问题。

如果真心认为自己容易被愚弄，也会比较谨慎一点，在考虑自己的测试策略细节时就会更认真一点。这是一种新测试员能够提高的最快的方法之一，因为知道自己可以被愚弄是一种态度，并不是特殊技能或知识。新测试员的问题是，对于他们来说，这个定律只是一条信仰（“人家告诉我，我应该认为我可以被愚

弄……”），而有经验的测试员的感觉和反应，会通过实际教训唤起和加强（“我还觉得1994年的那次大教训。我们怎么也没有想到病毒会感染我们自认为性能非常可靠的磁盘。我的声誉在这一天全给毁了”）。

如果遗漏一个问题，检查这种遗漏是意外还是策略的必然结果

如果掷币猜边时，猜的是国徽面，出现的却是字面，这是否意味着做出了差的决策？以任何理性的观点看都不是这样。除非在硬币上做了手脚，否则出现任何一面的机会都是50%。出现字面没有什么可奇怪的，只是不够幸运罢了。决策策略没有问题。

在测试过程中没有发现程序错误时也存在同样问题，同样也会困扰客户。在研究测试策略出现了什么问题之前，先不要自责。出现遗漏，是否因为忠实地执行了好的测试策略，并只是碰巧没有发现那个特定的问题？如果是这样，可保持原有方针不变。确实有这种情况。但是，如果遗漏程序错误是因为测试策略关注了错的问题类型，可利用这个机会改进测试策略。

困惑是一种测试工具

当测试员感到困惑时，这可能是某种重要的预示。

- 规格说明不清楚吗？规格说明中的模糊点，常常是为了掩盖有影响力的项目相关人员之间的重要分歧。
- 产品不清楚吗？产品可能有严重问题。
- 用户文档不清楚吗？产品的这个部分可能太复杂，有太多的特例和不一致性要描述。
- 内部问题只是难以理解吗？我们试图自动化的有些系统具有内在的复杂性，或包含复杂的技术问题。程序员也认为它们复杂、困难，并导致自己犯遗漏、误解和过于简化的错误。

测试员对产品、技术和一般测试问题了解得越多，自己的困惑就会成为更有力的指南针，指出重要问题所在。

在测试过程中，如果对产品一无所知，那么至少知道自己在困惑。在这种情况下，困惑可以成为最佳交付内容，即提出也许其他人没有勇气提出的问题。

清新的眼光会发现失效

理解事物，是把新信息吸收到已知信息中，同时修改已知的信息以适应新信息的高智力过程。测试员在理解了产品或功能部件之后，会在头脑中形成映射，并且头脑不再那么努力工作。对于测试员来说这可能是个问题。当非常了解产品后，会对产品做出更多的假设，并更少检查这些假设。

这种情况对于测试至少有三点提示：

- 第一次接触产品或功能时，要特别注意使自己困惑和烦恼的地方。这可能说明用户也会有类似反应。
- 当与团队的新成员一起工作时，与他们一起测试。观察他们在了解产品时的反应。
- 警惕陷入测试惯例。即使没有遵循严格的测试脚本，也可能对特定功能太熟悉，以至于以越来越窄的方式进行测试。在任何可能的地方引入多样性，或改由其他测试员负责。

测试员要避免遵循过程，除非过程先跟随自己

警惕其他人的过程。测试用例和过程的描述，常常不提测试的内部设计目标。这非常容易使测试员在执行测试时并不太理解如何建立测试，或寻找什么。换句话说，测试员并没有真正跟上过程。一般来说，测试过程的编写和设计都比较差，因为没有多少优秀测试员像擅长计算机那样擅长程序设计人员的工作。如果要遵循测试过程，最好采用自己设计、自己拥有或已经完全了解的过程。

为了得到最好结果，测试员必须掌握自己的测试，而不是自己的文档。要使过程跟随自己。

如果确信那些过程很好，也至少要研究一下过程的工作原理。请参阅《使我们聪明的事物：机器时代的人性保护》(Things that Make Us Smart: Defending Human Attributes in the Age of the Machine)(Norman 1993)和《信息的社会寿命》(The Social Life of Information)(Brown和Duguid 2000)。

在创建测试过程时，避免“1287”

我们中的Bach曾经见过一位测试员编写的测试过程包含“在字段中输入

1287个字符。”这1287是从哪里来的？测试员解释说，她的测试想法只不过是在一个小输入字段中，输入非常多的字符。因为她听说测试过程必须具体，因此小心地数了自己已经输入的字符数，1287。这就是过程中的这个数的由来——一个任意数，现在却被永远供奉起来，就像是印在水泥人行道上的猫的脚印一样。

过于详细没有什么好处。当编写测试过程时，要避免与测试概念无关的细化。包含所有必要的信息和设计与解释测试所需的细节，但是要让未来的测试员有创造性和判断力地执行，让未来测试员引入变化以使现在所编写的测试过程新鲜、高效。

测试过程的一个重要成果，是更好、更聪明的测试员

我们经常听到反对产生很少或不产生文档的测试的理由，好象测试的惟一价值就是通过测试产生的文档。这种观点忽略了测试的一种意义深远的重要产品：测试员本身。

好的测试员永远都在学习。随着项目的进展，他们不断加深对产品的了解，逐渐从各个方面提高对产品的反应能力和敏感性。了解产品并已经经历过一两次产品发布的有经验的测试员，即使没有任何指示，但与有一套如何测试产品的书面指示的没有经验的测试员相比，他们测试的有效性也要高得多。

软件测试领域内的一些顾问和论文作者看起来相信，只要提供测试过程，测试效果差的测试员就可以变成测试效果好的测试员。在我们看来，这是一种差的实践，这反映出他们对测试和进行有效测试的人的根本误解。

在评估测试过程时，首先要看项目测试员的素质，要看他们怎么思考，以及这种思考怎样对其行为产生影响。只有掌握了这些信息才能评估他们的工作产品。

除非重新发明测试，否则不能精通测试

不要彻底改造轮子。请等一下，难道轮子不是历史上被重新发明次数最多的吗？这不是好事情吗？毕竟我们的汽车是装在充气轮胎上，而不是花岗岩圆盘上。轮子如果说不是数以百万计的话，也有数以千计的变种。也许我们可以从中得到启发。在我们看来，重新发明东西至少有两个原因：通过改造使其适应新条件，了解其工作原理。而要掌握它，这两个方面都需要。

我们有的同事忠告测试专业的学生不要重新发明测试，也不要重新发明测试

思想。我们不同意这种观点。这就像是学习科学又不想做实验一样。通过其他思想家进行学习是没有错的，我们认为这样学习是很重要的。但是如果这是学习的惟一方式，那永远也不会成为测试技艺的行家。这样的人将是技术员，但不会再进一步了。按照指示做不会掌握任何东西，就像要沿高速路上火星一样。我们提倡要像伟大的机械师和伟大的程序员那样地学习测试：把东西分解，琢磨其工作原理，再以新的方式组装到一起。不要把自己限制为只是接受智慧的服务者，而应该使自己成为智慧的创造者。

在学习过程初期，要重新发明不是非常好的测试、想法、手段和文档。这是正常的。要永远使头脑运转，观察其他测试员，研究和不断评估如何筛选自己的思想。如果想要善于做到这一点，就必须实践。

我们这样做已经有很多年了，我们仍然在重新发明，仍然在反思老的想法。我们所尊敬的每个同事都是这样走向精通之路的。

第3章

测试手段

测试员应该做什么？在前两章中，我们的回答是观察和学习，但是这个回答相当抽象。现在该更具体一些了。测试从哪里来？测试看起来像什么？本章谈论测试手段问题，但是不会详细定义每种手段。要想得到这类信息，读者必须阅读有关测试的主要教科书。我们推荐Kaner、Falk和Nguyen（1993），Jorgensen（1995），Beizer（1990），Marick（1995），Collard（即将出版）以及Hendrickson（即将出版）。Whittaker和Jorgensen的论文（1999和2000）以及Whittaker（2002）也提出了很有用的思想。

本章的结构与本书其他几章不同，这有两个原因。

- 首先，本章给出的观点基本上是结构化的，要介绍其余内容的分类系统。我们把这个分类系统放在第一条经验中。接下来的五条经验列出若干手段，但是列出的主要目的是支持分类系统。给出这些细节可使读者更容易地将分类系统用于自己的工作中。

这个分类系统综合我们单个地使用和教授的方法。使用这种结构确定哪些手段是可用的，并且适用于给定问题，或产生将这些手段结合起来有效地解决问题的思想。

手段清单有时包含超出快速描述的细节，不过我们把这部分内容当作选读材料。细节的详细程度有意不均匀。我们期望读者能够通过其他专著或课程了解大多数手段的细节。

- 第二，尽管本章并不讨论主要如何使用手段，但是不可能在讨论测试手段时不足够详细地描述一些读者会实际使用的手段。因此本章附录将采用我们在专业级研讨会和软件测试大学课程上受到学生欢迎的方法，来介绍我们认为很有用的五种手段。

经验48

关注测试员、覆盖率、潜在问题、活动和评估的组合测试手段

本章的主要目标是提出一种测试手段的分类系统，我们把它叫做“五要素测试系统（Five-fold Testing System）”。人们可以做的所有测试都可以在五个方面进行描述：

- 测试员。进行测试的人。例如，用户测试是由目标市场的成员、通常使用该产品的人进行的专项测试。
- 覆盖率。测试了哪些内容。例如，在功能测试中，要测试每个功能。
- 潜在问题。测试的原因（要测试什么风险）。例如，测试极值错误。
- 活动。如何测试。例如探索式测试。
- 评估。怎样判定测试通过还是不通过。例如，与已知正确结果的比较。

本章还要详细描述几个手段，并就另外几个手段的使用提出自己的观点，不过我们的主要目标还是解释分类系统。

所有测试都包括所有这五个要素。测试手段将测试员的关注点集中的一个或几个要素上，把其他要素留给测试员自己判断。可以把关注一个要素的手段与关注另一个要素的手段结合起来，以得到想要的结果。可以把这种结合的结果叫做新手段（有人确实这样叫），不过我们认为思考过程要比增加另一个名字更重要。在测试领域中，正在使用的定义不一致的手段清单正在不断膨胀。我们的分类模式有助于读者有意识地理智深刻地理解这种结合。

测试任务常常按一个要素分配，但是完成任务时要涉及所有五个要素。例如：

- 有人可能要求测试员做功能测试（彻底测试每个功能）。它说明的是要测试什么，测试员还必须决定谁来测试，以及要寻找什么问题、如何测试每个函数、如何确定程序是否通过。
- 有人可能要求测试员做极值测试（测试在向变量输入极值条件下的错误处理）。它说明的是要找出什么问题。测试员还必须决定谁来测试、要测试哪些变量、如何测试这些变量、如何评估结果。
- 有人可能要求测试员做β测试（让市场的外部代表做测试）。它说明的是谁来测试，测试员还必须决定告诉外部代表什么（告诉他们多少）、试用产品中的哪一部分、他们应该查找什么问题（应该忽略什么问题）。在有些β测试中，测试员还要具体地告诉他们如何识别特定类型的问题，可能要求他们以特定的方式执行特定的测试。在另外一些β测试中，可能会由他们

决定要完成的活动和评估。

手段不一定只涉及一种要素，也不应该是这样。所有测试都涉及所有五个要素，因此我们应该期望跨多个要素的更综合的测试手段。以下是多要素手段的一个例子：如果有人要求做“基于需求的测试”，则可能是表示以下三种想法的任意组合：

- 覆盖率（测试在这个需求文档中列出的所有内容）。
- 潜在问题（测试不满足这个需求的各种方式）。
- 评估（设计测试的方式，要使得测试员能够使用需求规格说明确定程序是否通过）。

在说到“基于需求的测试”时，不同的测试员会有这三种想法的不同组合，对这个词并不存在惟一的正确解释[⊖]。

尽管存在模糊性（并且在一定程度上正是因为有这种模糊性），但是我们认为这种分类系统是一种很有用的思想生成器。

测试时要时刻想着所有五个要素，就可能做出更好的组合选择。在β测试中，可能决定不描述这些要素中的一个或多个，可以决定不确定如何评估测试结果或测试员该怎样做。但是我们的建议是，要有意识地做出类似上面提到的决定，而不是采用只关注一种要素的手段，而不注意到还要做出其他决定。

关注测试员的基于人员的测试手段

以下是一些通过执行测试的人来区分的常见手段举例。

用户测试（user testing）。由将使用该产品的典型人员进行输入的测试。用户测试可以在开发期间任何时候进行，可以在开发场地，也可以在用户场地，可以在精心指导下进行，也可以根据用户的意愿进行。有些类型的用户测试，例如任务分析，更像是联合探索（涉及至少一名用户和至少一名公司测试小组成员）；而不是由一个人完成的测试。

α测试。由测试小组（可能还包括其他感兴趣的、友善的内部人员）执行的内部测试。

⊖ 基于需求测试的多种含义，提供了软件工程中一个重要普遍问题的一个例子。定义在测试领域是不固定的。定义的使用在不同子领域和个人之间有很大不同，即使存在期望能被看作是参考标准的文档。我们稍后讨论使很多人无视标准文档的一些因素。请注意，我们在这里不是要声称提供权威定义，或测试领域手段的描述。有些人会使用同样的词汇表示不同的含义。其他人可能会同意我们的描述，但是却以不同的方式表达。不管哪种情况都是合理的、有说服力的。

β测试。利用不属于开发机构并且是产品的目标市场成员的测试员实施的用户测试。待测产品一般非常接近完成。很多公司都将让客户试用代码看作是β测试，他们把所有β测试都归结为叫做“β”的里程碑。这是个错误。实际上有很多不同类型的β测试。设计β，用于要求用户（特别是有关领域的专家）评价设计，应该尽可能早地实施，以便有根据评价意见进行修改的时间。市场开发β，用于再次确认在该产品推出并投放自己的大型销售网上时会有大量客户购买，实施时间相当晚，要等到产品相当稳定之后。兼容性测试β，客户在开发机构自己不容易测试的硬件和软件平台上运行该产品。这种测试不能太晚，否则难以确定和解决兼容性问题。对于所管理的任何类型β测试，在确定进度和实施方法之前，都要确定测试目标。

强力测试（bug bash）。利用秘书、程序员、市场开发人员和可以找到的任何人所实施的内部测试。强力测试一般持续半天，在软件接近投放市场时进行。（请注意，我们列出这种手段是为了举例，并不表示赞同。有些公司由于种种原因认为它很有用，有些公司则认为没用。）

有关领域的专家测试（subject-matter expert testing）。向软件目标领域内的专家提供产品，并寻求反馈意见（错误、批评和赞扬）。专家可以是，也可以不是预期使用产品的客户，公司看重的是专家的知识，而不是其市场代表性。

成对测试（paired testing）。两个测试员在一起发现程序错误。一般情况下，他们共用一台计算机，在测试时轮流操作。

自用测试（eat your own dogfood）。全公司使用并依靠自己软件的试用版，通常要等到软件足够可靠能够实际使用时，才向市场销售。

关注测试内容的基于覆盖率的测试手段

可以根据在使用这些手段时已经掌握知识的不同，把这些手段按所关注的问题进行多种不同的分类。例如，如果把功能集成测试用于检查每个功能与所有其他功能组合在一起时是否能够正常运行，则这种测试就是面向覆盖率的测试。如果有针对功能相互交互的错误理论，并想进行跟踪，则这种测试就是面向问题的测试。（例如，如果意图是想发现功能在相互传递数据时出错，就是面向问题的测试。）

我们将在本章末尾补充解释这些定义中的一些领域测试，因为与领域有关的手段在软件测试中使用得非常普遍、非常重要。读者应该对其有所了解。

功能测试（function testing）。逐个测试每个功能。彻底测试功能，直到可以

确信该功能没有问题。白盒功能测试通常叫做单元测试，集中测试可以看到代码的功能。黑盒功能测试关注命令和特性，以及用户可以做或选择的事情。在做涉及多个功能的更复杂的测试之前，最好先做功能测试。在复合测试中，第一个出现问题的功能可能会使测试停下来，阻止通过这个测试发现多个其他功能也出现问题。如果依靠复合测试而不是单个测试功能，可能要到很晚才会知道有一个功能出现问题，可能要花费大量工作在复合测试中定位，最后却发现问题出现在一个简单功能上。

特性或功能集成测试（feature or function integration testing）。一起测试多个功能，以检查功能在一起执行的情况。

菜单浏览（menu tour）。遍历GUI产品中的所有菜单和对话框，使用每个可用的选项。

域测试（domain testing）。域是一个（数学）集合，包含所有可能的函数变量取值。在域测试中，要标识函数和变量。变量可以是输入或输出变量。（输入域和值域之间的数学区别在这里无关，因为这两种域的测试分析都是一样的。）对于每个变量，都要把其可能取值集合划分为等价类，并从每个类中选择少量代表（一般是边界值）。这种方法假设如果用类中的少量好的代表值进行测试，就可以发现用类中所有成员测试所能够找出的大多数或全部问题。请注意，与功能测试形成对比的是，感兴趣的要素是变量而不是功能。很多变量被多个功能使用。进行域测试时必须分析变量，然后再根据分析，以这个变量作为输入或输出，测试涉及这个变量的每个功能。

等价类分析（equivalence class analysis）。等价类是测试员认为是等价的一组变量取值。如果相信一组测试用例：（a）测试的都是相同的东西；（b）如果其中一个捕获到一个程序错误，其他测试用例也可能捕获到；（c）如果其中一个不能捕获到某个程序错误，其他测试用例可能也不能捕获到，则这些测试用例是等价的。一旦找出一个等价类，可只测试其一两个成员。

边界测试（boundary testing）。等价类是一组取值。如果可以把成员映射到一列数字上，则边界值就是类的最小和最大值。在边界测试中，要测试这些值，还要测试相邻类的边界值，这些值比要测试的类的最小值略小，比要测试的类的最大值略大。例如，请考虑一个接受10~50整数值的输入字段。感兴趣的边界值是10（最小整数）、9（小于10的最大整数）、50（最大整数）、51（大于50的最小整数）。

最佳代表测试（best representative testing）。等价类的最佳代表是在暴露软件中的错误的可能性方面至少与类中其他值一样的值。在边界测试中，边界值几乎总是最佳代表。但是有时不能将等价类映射到一组数字上。例如，兼容惠

普PCL-5的打印机是（或应该是）一个等价类，因为这些打印机的工作方式相同。假设对于一个具体任务，其中一种打印机与其他打印机相比，略微更可能出现问题。那么这种打印机可以作为这个类的最佳代表。如果对它测试没有发现问题，那么可以比较可靠地认为其他打印机也没有问题。

输入字段测试大纲或矩阵（input field test catalogs or matrices）。对于每种输入字段，可以开发一组相当标准的测试用例，在这个产品和后续产品中的类似字段中重用。本章稍后还要给出这种方法的例子。（请参阅“如何创建针对输入字段的测试矩阵”。）

用各种方式映射和测试编辑字段（map and test all the ways to edit a field）。常常可以以多种方式改变某个字段中的值。例如可以把数据输入到该字段，直接在字段中输入数据，通过程序将计算好的结果复制到字段中，通过程序将再次计算好的结果复制到字段中，等等。字段是有限制的（限制字段可以取哪些值）。有些限制是不变的，有些限制要依赖于其他字段的取值。例如，如果J和K是无符号整数，其限制就是0一直到MaxInt。这些都是不变限制，依赖于程序设计语言对无符号整数的定义。但是，如果N也是无符号整数，N = J + K，N = 5。在这种情况下，J = 5 - K，J不能大于5（N的值）。这是可变限制，其所允许的取值范围取决于N的值。为了检查J是否在所允许的取值范围内（5 - K），可以使用各种能够把数据输入到J中的方法改变J的取值。

逻辑测试（logic testing）。变量在程序中有关系。例如，程序可能有这样一个决策规则：如果PERSON-AGE大于50，并且如果SMOKER是YES，则OFFER-INSURANCE必须是NO。这种决策规则表达了一个逻辑关系。逻辑测试试图检查程序中的所有逻辑关系。因果图（cause-effect graphing）是一种用于设计大量基于逻辑测试的手段。

基于状态的测试（state-based testing）。程序的状态要发生转换。在给定状态中，有些输入是有效的，其他输入被忽略或拒绝。对于有效输入，被测程序要完成它可以做的事，并且不尝试做它不能做的事。在基于状态的测试中，每次都要通过经过大量状态迁移（状态改变）并仔细检查结果来检验程序。

路径测试（path testing）。一条路径包含测试员所执行的所有步骤，或程序为了得到正确状态所通过的所有语句。路径测试包括测试通过程序的很多路径。通过非平凡程序的所有路径是不可能的。因此，有些测试员进行子路径测试（subpath testing），测试很多部分路径。例如，基本路径测试（basis-path testing）包括测试一定类型（基本路径）的大多数或全部假设，这里采用的假设是如果测试了所有基本路径，那么几乎没有更长的路径会找出这些测试所遗漏的问题。

语句与分支覆盖率（statement and branch coverage）。如果测试执行了程序中的所有语句（或代码行），则达到100%的语句覆盖率。如果执行了所有语句和一个语句到另一个语句之间的所有分支，则达到100%的语句和分支覆盖率。设计自己的测试，达到高的语句与分支覆盖率，有时叫做“基于覆盖率的测试（coverage-based testing）”。（达到覆盖率目标后，可以停止测试，或停止设计更多的测试）。把它叫做*语句与分支覆盖率*，是为了与关注其他类型覆盖率的测试相区别。配置覆盖率就是一个很好例子，这种手段执行同一条语句很多次，但是潜在产生非常不同的结果。其他例子还有很多很多（Kaner 1995a）。关注达到高语句与分支覆盖率的测试往往遗漏很多类型的问题，例如（但不限于）与以下因素有关的程序错误：遗漏代码、边界值处理不正确、时序问题、硬件和软件配置兼容性问题，诸如指针越界、内存泄漏或栈破坏等最终导致栈溢出的滞后暴露问题、可使用性问题，以及其他方面没有满足客户需求的问题。这种手段在标识不完备测试方面（哪些代码还没有测试过）要更有价值得多，而不是在所需测试量的最低标准方面。的确，让测试员停止测试只是因为达到了X%的覆盖率，这样做很危险（Marick 1999）。

配置覆盖率（configuration coverage）。如果必须测试100台打印机的兼容性，并且已经测试了10台，就达到10%的打印机覆盖率。更一般地，配置覆盖率度量测试员已经运行（并且程序已经通过）的配置测试占计划运行的配置测试总数的百分比。为什么要把它叫做测试手段？一般我们只是将它看作是已经完成了多少一定类型测试的度量。但是，有些测试员完成的一系列特殊测试，可以更快、更容易地完成大量配置测试。对于他们来说，优化工作以达到高的覆盖率，是一种测试手段。

基于规格说明的测试（specification-based testing）。这种测试关注验证在规格说明中所做的有关产品的每个事实声明。（事实声明是可以用真或假表示的任何语句。）常常包括手册、市场开发文档或广告、技术支持人员寄给客户的印刷品中的所有声明。

基于需求的测试（requirements-based testing）。测试关注证明程序满足需求文档中的所有需求（或关注逐个需求地证明某个需求没有被满足。）

组合测试（combination testing）。相互组合测试两个或更多变量。本章最后的“测试手段附录”还要讨论这个问题。组合测试很重要，但是很多测试员对这种测试研究得还很不够。通过程序得到的大部分好处都基于很多变量的交互。如果不在测试中一起改变这些变量，就会遗漏由不同的组合（而不是不同的单个取值）触发的错误。

关注测试原因（针对风险测试）的基于问题的测试手段

基于风险的测试（risk-based testing）至少有两个主要含义。

Amland（1999）提出了一种基于风险测试管理的很好描述。根据这种观点，进行风险分析是为了确定下一步要做的测试。要根据程序中某个功能失效的可能性，以及如果失效确实发生可能造成的损失，确定测试的优先级。昂贵失效的可能性越高，尽早、尽可能仔细地测试该功能就越重要。

另一个含义是我们更关注的，即为了发现错误进行风险分析。当研究产品的某个功能时，我们要研究它会怎样失效。这个问题又可以分解为很多额外问题，例如：失效外观是怎样的？这个功能为什么会失效？什么样的风险因素可能影响这个功能？本章“测试手段附录”将描述我们的基于风险测试方法。

这两种基于风险测试的方法也在《James Bach论基于风险的测试》（1999c）中做了描述。

Whittaker和Jorgensen（1999和2000）做了精彩的讨论，给出了涉及约束冲突的各类错误的例子：

输入约束（input constraint）。限制程序可以处理的内容的约束。例如，如果程序只能处理32位数字（或小于32位数字），则程序员应该提供保护性例程检测并拒绝超出32位数字限制的输入。如果没有这样的保护，当程序试图处理它不能处理的输入数据时就会失效。

输出约束（output constraint）。输入是合法的，但是会导致产生程序所不能处理的输出值。当程序试图显示、打印或保存输出值时会失效。

计算约束（computation constraint）。输入和输出没有问题，但是在计算某个值（会产生一个输出）时，程序失效。例如，将两个很大的数乘在一起，积太大，程序不能处理。

存储（或数据）约束（storage (or data) constraint）。输入、输出和计算都是合法的，但是操作使程序耗尽内存，或产生的数据文件太大，程序不能处理。

Whittaker（2002）针对这些约束提出了详细建议。

以下是基于风险测试设计的一些补充提示：

- 如果进行基于风险的测试，还必须做相当量的非基于风险的测试，以针对了解还不够，还不能做出正确决策的风险进行测试。
- 针对时序的测试。令人奇怪的是，很多接受美式教育的测试员都没有想到时序问题。一些经典时序问题包括竞争条件和其他事件意外发生顺序。
- 在创建测试时，总要创建测试过程，以强制程序使用测试员输入的测试数

据，从而能够确定程序是否不正确地使用了这些数据。

关注测试方法的基于活动的测试手段

回归测试（regression testing）。回归测试涉及相同测试的重用，使得在软件变更以后可以重新执行（这些测试）。共有三种回归测试。报告了程序错误，在该程序错误得到更正之后，这时的测试叫做*程序错误更正回归*（bug fix regression）。其目标是证明该更正有误。*老程序错误回归*（old bugs regression）测试用来证明对软件的变更使老的程序错误更正变成未更正。*副作用回归*（side-effect regression）测试又叫做*稳定性回归*（stability regression）测试，要重新测试产品的很大一部分，其目标是证明该变更使得曾经没有问题的东西现在有问题了。

脚本测试（scripted testing）。手工测试，采用由更高级的测试员编写的测试过程步骤，一般由低级程序员执行。

冒烟测试（smoke testing）。这种副作用回归测试的目标，是证明新版本不值得测试。从一个软件版本到下一个版本，冒烟测试常常是自动化、标准化的。这种测试检查预期没有问题的内容，如果不是这样，就要怀疑该程序是用错误文件或基本上有问题的东西构建的。

探索式测试（exploratory testing）。我们期望测试员在整个项目过程中，都要了解产品、产品的市场、产品的风险和怎样没有通过以前的测试。不断创建并使用新测试。新测试比老测试更有力，因为新测试是建立在测试员持续增长的知识基础之上的。

游击式测试（guerilla testing）。对程序快速、有力的攻击。这是一种探索式测试，通常有时间限制，并由有探索式测试经验的测试员承担。例如，高级测试员花一天时间测试否则会被忽略的部分。测试员要进行最有力的攻击。如果发现严重问题，则对这部分要重新做预算，并且可能会影响整个测试计划。如果没有发现严重问题，这部分可以忽略，或只做少量测试。

场景测试（scenario testing）。场景测试（顾名思义）一般要涉及四个属性。（1）测试必须是现实的，应该反映客户实际要做的事。（2）测试应该是复合的，要以能够对程序构成一定挑战的方式包含多个功能。（3）应该能够容易并且快速地显示出程序是否通过测试。（4）如果程序没有通过测试，有关人员会强烈要求修改程序。具有这四种属性的测试会很有说服力，如果程序不能通过测试，一般会导致修改程序错误。但是，测试员可能需要花几天的时间开发出色的场

景测试。

场景测试（scenario testing）。通过用例导出的测试，又叫作测试试验（Jacobson 1992，Collard 1999）或用例流试验（use case flow test）。（很多人把这些测试归到基于覆盖率的测试中，关注重要用例的覆盖率。）

安装测试（installation testing）。以各种方式，在可以安装该软件的不同类型系统上安装该软件。检查在磁盘上增加或修改了哪些文件。安装后的软件能够正常运行吗？再卸载时会出现什么情况？

负载测试（load testing）。通过在面临很多资源要求的系统上运行，攻击被测程序或系统。在足够高的负载下，系统可能会失效，但是导致这种失效的事件模式会指出被测软件或系统中的弱点，这些弱点可能在被测软件的更一般的使用中暴露。Asbock（2000）对负载测试做了精彩介绍。

长序列测试（long sequence testing）。测试要持续一夜，甚至几天、几周，目标是发现短序列测试遗漏的错误。这种测试经常发现的错误包括越界指针、内存泄漏、栈溢出、超过两个特性之间的错误交互等。（长序列测试有时叫做持久测试（duration testing）、可靠性测试（reliability testing）或耐力测试（endurance testing）。）

性能测试（performance testing）。运行这些测试通常要确定程序运行有多快，以便确定是否需要优化。不过这种测试也可以暴露很多其他程序错误。如果与前一个版本相比出现显著性能变化，则这可能是编码错误作用的反映。例如，在试验今天运行一个简单功能测试需要多长时间，明天在同一台机器上运行同样的测试需要多长时间时，如果两次运行时间相差三倍以上，则可能需要与程序员进行核对，或编写错误报告。更慢和更快两种情况都要警惕，因为程序的基本内容发生了变化[⊖]。

关注测试是否通过的基于评估的测试手段

评估手段描述确定程序是否通过测试的方法。这些手段不说明应该完成什么测试，也不说明如何收集数据，而是要说明如果能够采集到一定的数据该如何评估。

⊖ Sam Guckenheimer指出，“性能差距还可能反映第三方组件或配置的变化。例如，JDK不同版本的JVM变化，会导致性能显著变化。由于这是客户可更新的组件，因此即使代码根本就没有改变，性能测试也可能产生令人惊讶的结果”！

自校验数据（self-verifying data）。所使用的数据文件带有使测试员能够确定输出数据是否被破坏的信息。

与已保存的结果进行比较（comparison with saved result）。回归测试（一般是，但并不总是自动化的）是否通过，通过将当前得到的结果与以前的结果进行对比确定。如果以前的结果是正确的，而现在有所不同，这种差别可能就是新缺陷的表现。

与规格说明或其他权威文档比较。不符合规格说明的都（可能）是错误。

启发式一致性（heuristic consistency）。一致性是评估程序的重要评判准则。不一致性可能是报告程序错误的一个理由，也可能反映有意的设计差异。我们介绍七种主要的一致性：

1. 与历史一致。现在的功能行为与以前的行为一致。
2. 与我们的想像一致。功能行为与机构的项目预期一致。
3. 与可比较的产品一致。功能行为与可比较产品的类似功能一致。
4. 与所声明的内容一致。功能行为与承诺提供的功能一致。
5. 与用户的预期一致。功能行为与我们认为是用户想要的功能一致。
6. 产品内部一致。功能行为与产品内部可比较的功能或功能模式的行为一致。
7. 与用途一致。功能行为与明确的用途一致。

基于理念的测试（oracle-based testing）。理念是一种评估工具，它会告诉测试员程序是否通过测试。在大规模的自动化测试中，理念也许是另一种产生结果或检查被测软件的结果的程序。理念一般要比被测软件更可信赖，因此值得花时间和精力检查理念所给出的提示。

根据自己的看法对测试手段分类

读者可能不明白我们为什么要这样分类给出具体测试手段。如果读者有这种疑问会很有好处，这说明读者在思考。请注意，所有测试都要包括“五要素测试系统”中的所有五种要素。我们分类列出测试手段，只是使读者能够体会到不同测试手段强调思考的某种方式。读者的感觉会是不同的。例如，有读者与我们辩论，认为负载测试应该划归基于问题（或基于风险）的测试，而不是基于活动的测试。我们的回答是，既可以把负载测试看作是基于问题的测试，也可以看作是基于活动的测试。

下面从面向问题的观点讨论这个问题：

- 可以通过服务拒绝攻击效果来考虑负载测试。攻击者可以尝试通过创建过

多的连接或用户，或通过使用很多内存（使每个用户同时发出非常占内存的命令），或通过利用占用过多处理能力的任务，来使服务器拒绝服务。针对这些风险中的每一种实施不同的负载测试。

下面从面向活动的观点考虑：

- 使用工具攻击客户活动模式。客户最常使用的命令是哪些？客户最常使用的任务有哪些？进行每种活动的客户百分比是多少？当测试员在自己的场地建立了使用模式模型后，找一个负载测试工具，并编程使其看起来像每类使用的场景。让工具随机选择场景，最后产生代表不同类型用户的不同会话。不断增加会话，观察系统性能和可靠性随负载的增加而下降的情况。根据实际情况修改软件。

从风险方面思考时，会考虑到程序可能存在的弱点，研究如何设计一系列测试暴露这些弱点。当知道想要运行什么类型的测试时，考虑一下运行该测试将采用的方式。如果要测试的是某种电话系统，则可能需要一种工具，或能够拉上10个朋友发出一批电话呼叫。这种测试设计的评判准则必须有强度，即可以检测出所寻找的缺陷。

形成对比的是，在从活动方面考虑时，研究的是如何完成负载测试。用什么工具？这些工具能做什么？等等。期望是，如果恰当地使用这些工具，并完成与好的负载测试关联的其他活动，就可能发现负载测试有可能暴露的那些类型的程序错误。

这两种分类都是准确的，但是分类本身的帮助也就是这么多了。不管怎样对类似负载测试这样的手段分类，在进行实际测试时，仍然需要在五个要素方面进行决策：

- 谁来测试？
- 要测试程序的哪些方面？
- 要寻找什么类型的问题？
- 具体要完成什么任务？
- 如何确定测试是否通过？

测试手段附录

以下更详细地介绍几种我们认为特别有用的关键测试手段：

- 如何针对输入字段创建测试矩阵。
- 如何针对重复问题创建测试矩阵。
- 如何为基于规格说明的测试创建可跟踪性矩阵。
- 如何使用全对偶（all pairs）测试手段进行组合测试。

- 如何分析与程序的某个部分或方面关联的风险。

如何针对输入字段创建测试矩阵

首先，“什么是针对简单整数字段的有意义的输入测试？”以下是我们看作为这种字段例程的一些测试：

- 不输入。
- 清空字段（清空默认值）。
- 超出上限位数或字符数的数字或字符。
- 0。
- 有效值。
- 下限值-1。
- 下限值。
- 上限值。
- 上限值 + 1。
- 远远低于下限值。
- 远远高于上限值。
- 下限位数或字符数的数字或字符。
- 下限位数或字符数的数字或字符 - 1。
- 上限位数或字符数的数字或字符。
- 上限位数或字符数的数字或字符 + 1。
- 远远高于上限位数或字符数的数字或字符。
- 负数。
- 非数字，特别是/（ASCII字符47）和:（ASCII字符58）。
- 错误的数据类型（例如把小数输入整数字段）。
- 表达式。
- 在其他数据前加一个空格。
- 在其他数据前加很多空格。
- 在其他数据前加一个0。
- 在其他数据前加很多0。
- 在其他数据前加一个 + 号。
- 在其他数据前加很多 + 号。
- 非打印字符（例如Ctrl + 字符）。
- 操作系统文件名保留字符（例如，“*.:”）。
- 程序设计语言保留的字符。

- ASCII（又叫作ANSI）上半区（128 – 254）字符。
- ASCII 255（常常解释为文件尾）。
- 大写字符。
- 小写字符。
- 修饰键（例如，Ctrl、Alt、Shift-Ctrl等）。
- 功能键（F2、F3、F4等）。
- 不输入，但是在等待很长时间后才输入回车键或制表键，点击OK，或其他等效于退出字段的键。是否有超时控制？有什么效果？
- 输入一个数字，但是在等待很长时间后才输入另一个或多个数字，然后输入回车键。如果系统有超时控制，那么超时控制的时间有多长？对所输入的数据怎样处理？对以前输入的其他数据怎样处理？
- 输入多个数字并使用删除键编辑，再删除这些数字，并使用箭头键（或鼠标）跳到已经输入的数字上，以便能够插入或覆盖新的数字。
- 在系统正在响应不同类型的中断（例如打印机活动、时钟事件、鼠标移动和点击、文件存盘等）时输入数字。
- 输入一个数字，切换到另一个应用程序，再返回到这个应用程序。光标位置在哪里？

这样的列表常常叫做测试大纲（catalog of tests）（Marick 1995定义的这个词，并给出了例子。）我们发现将列表转换为矩阵形式很有用，如表3-1所示。

表3-1 数字输入字段测试矩阵

数字输入字段	不输入	清空字段（清空默认值）	0	下限值-1	下限值	上限值	上限值 + 1	远低于下限值	远高于上限值	上限位数数字	上限位数数字 + 1	远高于上限位数数字	负数	非数字（/ ASCII 47）	非数字（: ASCII 58）	错误的数据类型	表达式	其他数据前加空格	非打印字符	操作系统文件名	ASCII上半区	大写字符	小写字符	修饰键（Ctrl、Alt等）	功能键

矩阵中的第一行是测试员要反复使用的测试，每一列是要测试的字段。例如，在典型的“打印”对话框中，有一个字段是“打印份数”。打印份数的有效值范围一般是1~99，或1~255（取决于打印机）。在测试矩阵的一行中写上“打印：打印份数”，然后对该字段中运行部分或全部测试，并填写相应的结果。（我们喜欢用绿色和粉色突出对应通过和没有通过测试的小格。）

这个矩阵提供了一种易于委托的结构。当以后项目增加或修改了功能，可以为多个相对不太熟悉项目（但是有测试经验）的测试员分配多张这样的标准矩阵。测试员的工作就是检查新的基本功能是否正常，或认为会受变更影响的老功能是否仍然正常。

整数输入字段只是一个例子，读者会发现针对（具有不同精度的）有理数、（各种长度的）字符字段、文件名、文件存放位置、日期等创建图表很有价值。如果要一个程序接一个程序地测试输入字段，或反复针对被测程序测试输入字段，那么花些时间制定可重用的矩阵还是值得的。

如何针对重复问题创建测试矩阵

输入字段的文字矩阵只是测试员可以创建的各类有用矩阵中的一个例子。输入字段并不是惟一可以被标准化的对象。只要某种情况在项目内部或项目之间反复出现，都要花时间和精力制定一个测试大纲的基础。有了大纲，总可以用矩阵形式表达。

以下是不涉及输入变量的一个例子。

在这个例子中，大纲列出程序尝试把文件写入磁盘的各种失败方式。有多种情况程序会试图写文件，例如：

- 保存新文件。
- 覆盖同名文件。
- 在结尾处续接文件。
- 用同名文件的新版本取代正在编辑的文件。
- 转换到另一种文件格式。
- 打印内容存盘。
- 消息或错误日志存盘。
- 保存临时文件。（很多程序都把这当作例程的一部分，因此在用户界面测试时可以不考虑。但是如果磁盘满，程序仍然会失效。）

这些情况中的每一个在矩阵中都有自己单独的一行。类似地，如果被测软件能够输出不同格式的数据，则每种格式在测试矩阵中也有自己单独的一行，等等。

以下是不成功尝试文件存盘的一些重要测试用例大纲：

- 保存到一个已满的本地磁盘。
- 保存到一个几乎已满的本地磁盘。
- 保存到一个写保护的本地磁盘。
- 保存到一个已满的局域网磁盘。
- 保存到一个几乎已满的局域网磁盘。
- 保存到一个写保护的局域网磁盘。
- 保存到一个已满的远程网磁盘。
- 保存到一个几乎已满的远程网磁盘。
- 保存到一个写保护的远程网磁盘。
- 保存到没有权限写的文件、目录或磁盘。
- 保存到一个已损坏（I/O错误）的本地磁盘、局域网磁盘或远程网磁盘。
- 保存到一个未格式化的本地磁盘、局域网磁盘或远程网磁盘。
- 打开文件后把本地磁盘、局域网磁盘或远程网磁盘从驱动器中移走。
- 等待本地磁盘、局域网磁盘或远程网磁盘在线时的超时控制。
- 在保存到本地磁盘、局域网磁盘或远程网磁盘期间创建一个键盘或鼠标I/O。
- 在保存到本地磁盘、局域网磁盘或远程网磁盘期间生成某个其他中断。
- 在保存到本地磁盘、局域网磁盘或远程网磁盘期间（本地计算机）断电。
- 在保存到本地磁盘、局域网磁盘或远程网磁盘期间（驱动器或连接到驱动器上的计算机）断电。

为了创建像这样的大纲，建议至少要召开两次有同事参加的集体讨论。在第一次讨论会上，努力想出针对被测对象（例如输入字段）或任务（例如保存文件）的例行测试过程。花一个小时的时间写满很多张纸，然后自己花一天时间单独整理通过讨论会得到的材料。

为了组织材料，可重新用几张纸，在每张纸上都写上主题标题，例如“磁盘能力”、“在写操作期间被中断”等。在每个标题下，将适合的条目抄过来。最后，每个条目都会归在一个主题下，或被放弃。（随意放弃不明智的提议。）

次日的讨论会讨论已经分类的测试项。大家可能会为“磁盘能力”、“在写操作期间被中断”等补充更多的测试项，并提出一些新的主题。在第二次讨论会上得到的测试项可能会翻一番，这种情况并不少见。

在第二次讨论会之后，将测试项排序，把基本测试项写入测试矩阵中，把不常使用的测试项写入第二个列表，也可能写入主矩阵中，也可能放弃。

Nguyen (2000)提供了测试矩阵的更多示例。

如何为基于规格说明的测试创建可跟踪性矩阵

可跟踪性矩阵使测试员能够正向跟踪每个测试用例到规格说明中的一项（或多项），并反向跟踪每个规格说明项到测试该规格说明项的测试用例。表3-2给出了一个例子。

表3-2 规格说明可跟踪性矩阵

	规格说明项1	规格说明项2	规格说明项3	规格说明项4	规格说明项5	规格说明项6
测试用例 1	X		X			X
测试用例 2	X	X		X		X
测试用例 3			X	X		X
测试用例 4			X	X		X
测试用例 5	X				X	X
测试用例 6		X				X
合计	3	2	3	3	1	6

每列包含一个不同的规格说明项。一个规格说明可以指一个功能、一个变量或变量的一个取值（例如边界值）、所承诺的好处、所声称的兼容驱动器以及其他可以证明真或假的任何其他承诺或陈述。

每行是一个测试用例。

每个小格指示哪个测试用例测试哪个规格说明项。

如果功能出现变化，可以从中迅速看出哪些测试必须重新分析，并有可能重写。一般来说，可以从给定的规格说明项跟踪覆盖它的测试。

这个矩阵并不是完美的测试文档。它没有说明测试，只是将测试用例映射到规格说明项。从这个矩阵不能看出测试是强的还是弱的，也不能看出测试用该功能（或其他规格说明项）做了很重要的事，还是大家都不太关心的事。此外，从这个矩阵还不能看出针对没有被在规格说明中说明的功能的测试，也不能看出用于规格说明不正确而经过调整的测试。尽管存在这些问题，这样的矩阵仍然有助于测试员了解：

- 几乎永远不会测试的功能，而另外的功能得到极为经常的测试。
- 针对某个规格说明项的变更（例如表3-2中的规格说明项6），会导致对系统中的大量测试重新进行考虑。（在合同推动的开发中，这是一种关键问题，因为如果客户进行显然是被要求进行的变更，就要支付大量测试经费，在做出变更之前应该得到警告。）

可跟踪性矩阵不仅能够用于基于规格说明的测试中，还可以用于其他测试。只要有一个要测试内容的清单（规格说明项、功能、用例、网卡等），就可以放在矩阵的第一行，把测试用例作为列，然后检查哪个测试用例测试哪项内容。

通过这种方式几乎肯定会找出测试漏洞。如果测试是自动化的，也许能够自动生成可跟踪性矩阵。

如何使用全对偶测试手段进行组合测试

组合测试要求一起测试多个变量。组合测试的第一个关键问题是测试用例的数量。请想像三个变量一起测试，每个变量有100个可能的取值。变量1加变量2加变量3的可能取值个数是100 × 100 × 100 = 1 000 000个测试用例。压缩测试数量是一个关键问题。

从域划分开始

第一步是压缩每个变量要测试的取值数。最常见的方法包括域测试。将变量1的取值划分为子域，并选择子域最好的代表值。通过这种方式有可能把变量1的测试数压缩到5个。如果对变量2和变量3也能做同样的处理，则现在就只有5 × 5 × 5 = 125个测试了。在实践中这还是太多，不过比1百万还是少多了。

Ostrand和Balcer（1988）、Jorgensen（1995）对打印划分做了精彩的讨论。Jorgensen给出了划分和多个划分后变量组合测试的很好的例子。我们给出一种与他的方法不同的组合方法，我们认为这种方法很有用。

获得全单值

最简单的一组组合测试，要保证覆盖每个变量的每个重要取值（我们要测试的5个取值中的每一个）。这叫做“全单值（all singles）”（与全对偶和全三元组（all triples）形成对比），因为要保证要测试每个变量中的每个取值。可以通过以下步骤做到这一点：

1. 设V1、V2和V3代表三个变量。

2. 设A、B、C、D和E是V1中的5个重要取值。特别地，设V1是操作系统，A是Windows 2000，B是Windows 95，C是Windows 98最初版，D是带第一个服务包的Windows 98，E是Windows ME。

3. 设I、J、K、L和M是V2中的5个重要取值。特别地，设V2是浏览器，I是Netscape 4.73，J是Netscape 6，K是Explorer 5.5，L是Explorer 5.0，M是Opera 5.12 for Windows。

4. 设V、W、X、Y和Z是V3中的5个重要取值，分别表示系统中的5个不同的磁盘驱动器。

为了测试这些变量取值的所有组合，会得到5 × 5 × 5 = 125个测试。

表3-3给出了一种达到“完备测试”的组合测试表，其中完备性的评判准则是，每个变量的每个取值都必须出现在至少一个测试中。

表3-3 全单值——所有值都出现至少一次

	变量1	变量2	变量3
测试用例 1	A（Windows 2k）	I（Netscape 4.73）	V（磁盘1）
测试用例 2	B（Windows 95）	J（Netscape 6）	W（磁盘2）
测试用例 3	C（Windows 98）	K（IE 5.5）	X（磁盘3）
测试用例 4	D（Windows 98 SP1）	L（IE 5.0）	Y（磁盘4）
测试用例 5	E（Windows ME）	M（Opera 5.12）	Z（磁盘5）

这种方法常用于配置测试中，以把被测配置的数量压缩到可管理的程度。

这种方法的一个严重问题，是会遗漏可预知的重要配置。例如，有很多人会在Windows ME上使用Explorer 5.5，但是这种测试没有列出，给出的是在Windows ME上测试Opera 5.12。

这种问题的常见解决办法是，确定包含关键变量值对偶（例如Windows ME加Explorer 5.5）的额外测试用例，或两个以上变量值的其他关键组合（例如Explorer 5.5，加Windows ME，加HP 4050N打印机，加256M RAM，加分辨率为1600 × 1200的21英寸彩色监视器。）市场开发或技术支持人员会提出这些组合，可能会提出10或20种额外的关键配置，需要总数为15或25的测试。

获得全对偶

在全对偶方法中（Cohen等，1996和1997），一组测试用例包含每个变量的所有取值对偶。因此，E（Windows ME）不仅与M（Opera）配对，还要与I、J、K和L配对。类似地，E也要与V3的每个取值配对。

表3-4给出了满足全对偶准则的一组组合。每个变量的每个取值都要与至少一个测试用例中的每个其他变量的每个取值配对。与全单值相比，这是更彻底的标准，不过仍然把测试用例从125（全组合）压缩到25，压缩量相当大。

表3-4 全对偶——取值的所有对偶都至少出现一次（25个测试，不是125个测试）

	变量1	变量2	变量3
测试用例 1	A	I	V
测试用例 2	A	J	W
测试用例 3	A	K	X
测试用例 4	A	L	Y
测试用例 5	A	M	Z
测试用例 6	B	I	W
测试用例 7	B	J	Z
测试用例 8	B	K	Y
测试用例 9	B	L	V
测试用例 10	B	M	X
测试用例 11	C	I	X
测试用例 12	C	J	Y

（续）

	变量1	变量2	变量3
测试用例 13	C	K	Z
测试用例 14	C	L	W
测试用例 15	C	M	V
测试用例 16	D	I	Y
测试用例 17	D	J	X
测试用例 18	D	K	V
测试用例 19	D	L	Z
测试用例 20	D	M	W
测试用例 21	E	I	Z
测试用例 22	E	J	V
测试用例 23	E	K	W
测试用例 24	E	L	X
测试用例 25	E	M	Y

为了说明如何创建一个全对偶测试集，以下一步一步地介绍一个简单些的例子。

一个详细的例子

请想像一个有三个变量的程序：V1有三个可能的取值，V2有两个可能的取值，V3有两个可能的取值。如果V1、V2和V3是独立的，那么可能的组合是12（3×2×2）个。

为了构建全对偶表，可按以下步骤进行：

1. 在列上标出变量名称，（按可能取值的数量）降序排列变量。

2. 如果第一列是V1的可能取值，第二列是V2的可能取值，则表中至少有V1×V2行（以这种方式画出表，但是在第一列的每组重复值之间留出一两行空行）。

3. 填写表格，一次填写一列。第一列每次重复其元素V2次，空出一行，然后再重复给出下一个元素。例如，如果变量1的可能取值是A、B和C，且V2有两个可能取值，则第一列包含A、A、空行、B、B、空行和C、C、空行。留出空行是因为很难知道会需要多少测试，留出空行给额外的测试。

4. 在第二列中，列出变量的所有取值，空出一行，再列出取值，等等。例如，如果变量2的可能取值是X和Y，则到目前为止的表如表3-5所示。

5. 增加第三列（第三个变量）。

第三列的每个段（两个AA行构成一个段，两个BB行构成另一个段，等等）要包含变量3的所有取值。这些取值的顺序要使该变量3与变量2也构成全对偶。

假设变量3可以取0或1。第三段可以任意填写。可以在矩阵中对做出的选择做标记，以便在必要时可以将其反过来。选择（比方说1，0）是任意的。参见表3-6。

表3-5 创建全对偶矩阵的第一步

变量1	变量2	变量3
A	X	
A	Y	
B	X	
B	Y	
C	X	
C	Y	

表3-6 创建全对偶矩阵的第二步

变量1	变量2	变量3
A	X	1
A	Y	0
B	X	0
B	Y	1
C	X	1
C	Y	0

现在就完成了这个三列练习。下面再增加变量，每个新增变量都有两个取值。

为了增加有多于两个取值的变量，必须从头来，因为表中的变量必须从取值个数最多的变量开始降序排列。（也可以不这样做，但是根据我们的经验，如果不明智地尝试其他方法会出现很多错误。）

给出第四列很容易。首先保证得到第四列和第二列的所有取值对偶（可以在AA和BB段中完成），然后保证得到第四列和第三列的所有取值对偶。参见表3-7。

表3-7 在全对偶矩阵中增加第四个变量

变量1	变量2	变量3	变量4
A	X	1	E
A	Y	0	F
B	X	0	F
B	Y	1	E
C	X	1	F
C	Y	0	E

第五列需要注意（参见表3-8）。第五列得到GH与第一、二和三列的所有对偶，但是没有得到与第四列的对偶。

表3-8 在全对偶矩阵中增加第五个变量

（这个矩阵不行，不过它可以说明先做出猜测，如果猜测不正确如何恢复）

变量1	变量2	变量3	变量4	变量5
A	X	1	E	G
A	Y	0	F	H
B	X	0	F	H
B	Y	1	E	G
C	X	1	F	H
C	Y	0	E	G

最近的任意选择是BB段中的HG。（确定了BB段中的先H后G的顺序后，对于CC段HG就是必要的顺序，以便H能够与第三列的1配对。）

过去关于HG对BB段是好顺序的猜测是错误的。为了恢复，可将其擦掉重新尝试：

1. 颠倒最新任意选择（第五列，BB段，HG改为GH）。

2. 擦掉CC段，因为HG的选择是根据BB段是HG做出的，需要擦掉。

3. 通过检查遗漏的对偶，重新填写CC段。GH、GH得到两个XG、XG对偶，因此CC段翻转为HG。这使得第二列X对应第五列的H，第二列的Y对应第五列的G，这正是组成所有对偶所需要的。（参见表3-9。）

表3-9 在全对偶矩阵中成功地增加第五个变量

变量1	变量2	变量3	变量4	变量5
A	X	1	E	G
A	Y	0	F	H
B	X	0	F	G
B	Y	1	E	H
C	X	1	F	H
C	Y	0	E	G

如果尝试再加一个变量，这六个对偶就不行了。尝试任何顺序的IJ（第六列的值），总也不会成功。（参见表3-10。）

不过这个问题很容易解决。只需要再增加六个测试用例，如表3-11所示。如果考虑第二张表，所需要的是把G与J配对和把H与I配对的两个测试。其他

变量的任何取值都无关（就得到所有对偶而言），因此任意填写其他变量值。如果还要不断增加变量，可以先不确定这些任意变量值，以后（在尝试使变量7和变量8适应同样的八个测试用例时）再决定在这些行中填写方便的值。

表3-10 这六个变量不适合全对偶矩阵中的六个测试

变量1	变量2	变量3	变量4	变量5	变量6
A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	J
B	Y	1	E	H	I
C	X	1	F	H	J
C	Y	0	E	G	I

变量1	变量2	变量3	变量4	变量5	变量6
A	X	1	E	G	I
A	Y	0	F	H	J
B	X	0	F	G	I
B	Y	1	E	H	J
C	X	1	F	H	J
C	Y	0	E	G	I

表3-11 有八个测试用例的六个变量全对偶

变量1	变量2	变量3	变量4	变量5	变量6
A	X	1	E	G	I
A	Y	0	F	H	J
				G	J
B	X	0	F	G	I
B	Y	1	E	H	J
				H	I
C	X	1	F	H	J
C	Y	0	E	G	I

如果要测试这些变量的所有组合，则需要3 × 2 × 2 × 2 × 2 × 2 = 96个测试。我们使用全对偶，把测试集从96压缩到8，效果显著。

如果仅仅使用全对偶测试用例是有风险的。与全单值一样，测试员可能知道被广泛使用的特定组合，或可能有麻烦的组合。能够做的就是在表中补充这些测试用例。我们已经把测试用例从96降到8，把测试用例扩展到10或15个，以覆

盖重要的特殊情况，这样做是合理的。另一个经过处理的例子，请参阅（Cohen等1997）。

如何分析与程序的某个部分或方面关联的风险

假设测试员正在测试产品的某个功能。（也可以是在测试某个变量，并不限定是特征。）

这个特征可能有问题，即它不能通过产品质量的某种重要度量。

为了确定特征是否有问题，请考虑问题的起因，即使特征更可能出错的因素。

质量属性

如果没有或与以下属性冲突的特征，也许应该写入错误报告：

- 可获得性（accessibility）。
- 能力（capability）。
- 兼容性（compatibility）。
- 并发性（concurrency）。
- 标准符合性（conformance to standards）。
- 效率（efficiency）。
- 可安装与可卸载性（installability and uninstallability）。
- 可本地化（localizability）。
- 可维护性（maintainability）。
- 性能（performance）。
- 可移植性（portability）。
- 可恢复性（recoverability）。
- 可靠性（reliability）。
- 可伸缩性（scalability）。
- 安全性（security）。
- 可支持性（supportability）。
- 可测试性（testability）。
- 可使用性（usability）。

为了确定某个特征是否有缺陷，可以问自己如何证明它没有或与这些属性冲突。

以可使用性为例。如何证明被测功能是可使用的？不可使用性的特征是什么？可以使用什么传统可使用性测试来研究这个特征？可问自己类似这样的问题（并运行合适的测试）。

这些测试受测试员想像力的约束，但是测试员很多思想可能来自问题起因清单。

问题起因

以下是一些反映错误可能性的因素。测试员可以把这些因素中的每一个看作是或大或小（根据自己的判断）警告，并设计测试，以确定程序是否有这些因素所提示的脆弱性。

新功能。较新的功能可能失效。

新技术。新概念带来新错误。

新市场。不同的客户群以不同的方式看待和使用产品。

学习曲线。由于无知而犯的错误。

变更后的功能。变更可能破坏老代码。

后期变更。匆忙的决策，匆忙或士气受影响的员工产生的错误。

贸然的工作。有些任务或项目习惯性地资金不足，会影响工作质量的所有方面。

差的设计或没有可维护性的实现。有些内部设计决定，使得代码过于难维护，以至于错误更正总产生新问题。

疲倦的程序员。超过几个星期或几个月的长时间加班会导致效率低下和错误。

其他人员问题。酗酒问题、健康问题、家人去世……两个程序员相互不说话（他们编写的代码也不沟通）……

意外溜入。程序员私人插人的（但是是计划外的）特征也许不能很好地与其他代码交互。

外来品。外部组件会引起问题。

预算外。没有预算的任务质量可能很差。

模糊。（规格说明或其他文档中）模糊的描述，会导致不正确或有冲突的实现。

矛盾的需求。模糊常常隐藏着矛盾，其结果是使产品对某人没有价值。

未知需求。在整个开发过程中，新需求不断浮现。不满足合理需求，对于该产品的项目相关人员来说就是质量问题。

需求变化。随着产品开发的推进，人们逐渐意识到自己想要什么。坚持按产品开始时的需求表开发可能会满足合同，但是会使产品失败。（请参阅www.agilealliance.org。）

复杂性。复杂的代码容易有问题。

问题过多。有很多已知存在问题的功能，可能还有很多未知问题。

依赖性。失效可能触发其他失效。

不可测试性。缓慢、低效的测试风险。

没有什么单元测试。程序员发现并修改大多数自己的程序错误。走这样的捷径意味着风险。

到目前为止没有什么系统测试。未测试过的软件可能失效。

依赖狭窄的测试策略。例如，狭窄的回归和功能测试，会导致大量错误在多个版本中一直存在。

弱的测试工具。如果没有工具帮助标识和隔离某类错误（例如越界指针），则错误更有可能不被发现。

不可修改性。不能修改错误的风险。

程序设计语言类型错误。例如C语言中的越界指针问题。有关例子，请参阅《面向对象开发的不足》（Pitfalls of Object-Oriented Development）（Webster 1995）和《Java的不足：改进程序的省时解决方案与技巧》（Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs）（Daconta等2000）。

使用错误大纲

《测试计算机软件》（Testing Computer Software）（Kaner等 1993）给出了480个常见缺陷。可以利用这个缺陷表开发自己的缺陷表。以下是缺陷表的使用方法：

1. 在缺陷表中找到一个缺陷。
2. 研究被测软件是否会有这种缺陷。
3. 如果在理论上有存在这种缺陷的可能，研究如果存在如何发现。
4. 研究在程序中出现这种错误的可能性，如果存在会产生怎样严重的失效。
5. 如果可以，针对这类错误设计一个或一系列测试。

但是，有太多的人都以《测试计算机软件》中给出的缺陷表开始和结束。这个表已经过时，在本书出版时就已经过时了。而且它也没有涵盖特定系统的问题。构建缺陷表是一种不断处理、补充的工作。以下是一个来自Hung Nguyen（个人通信）的例子。

这个问题是在一个客户/服务器系统中发生的。系统向客户发送一个名称表，以校验客户输入的名字是否是新的。

客户1和客户2都要输入一个名字，客户1和客户2都使用同一个新名字。相对于本地比较表来说，这两个名称实例都是新的，因此被接受。这样就有了同一个名字的两个实例。

发现这种问题之后，我们建立了一个问题库。发现方法是探索式的，并要求精细的内部技术。确定主要问题的测试计划，或要自动化正在开发的测试脚本。

当测试员发现新的问题（在时间限定内）后，将其放入数据库中，使下一个

项目的测试员也能够看到。

利用最新资源补充自己的错误大纲

有针对常见平台中常见失效的大量资源。以下是一些我们认为很有用的信息源样本：

www.bugnet.com

www.cnet.com

Nguyen（2001）

Telles和Hsieh（2001）。

第 4 章

程序错误分析

不能报告程序错误的测试员，很像是只有当冰箱门被关上时才亮的冰箱灯。冰箱内是被照亮了，只是没有亮在点子上。测试员要提供信息服务，不过为了提供有效服务，测试员不光要填写报告模板，并假设报告能够被完全理解。要了解如何编写和表达自己的测试结果，以便读者能够真正得到结果。我们把这个过程叫做“程序错误分析”。

经验 55 文如其人

错误报告是大多数测试员的主要工作产品。测试员的读者通过这些文档认识测试员。报告写得越好，测试员的声誉越高。

程序员通过测试员的报告得到关键信息。重要问题的良好报告会为测试员带来良好声誉，差的报告会为程序员带来额外（在程序员看来是不必要的）的工作。如果测试员浪费了程序员太多的时间，程序员就会躲避测试员及其工作。

程序员并不是测试员的惟一听众。项目经理和执行经理有时也阅读错误报告。人事管理问题会很快引起他们的注意，并刺激他们的神经。这些错误报告看起来像是解释不清、研究不充分或提出过于追究小问题的建议，这些都会使负责奖励和提升测试员的人产生消极印象。

如果下功夫研究并写好报告，所有人都会受益。

经验 56 测试员的程序错误分析会推动改正所报告的错误

测试员写的错误报告是要求改正错误的分析文档。

有些错误永远也不会被改正。测试员的责任不是保证所有错误都得到改正，而是准确报告问题，使读者能够理解问题的影响。

深入研究并写出好的报告，常常对错误改正的可能性产生巨大的影响。

使自己的错误报告成为一种有效的销售工具

不管测试员是否这样想，他们的错误报告都是一种推销工具，它劝导人们付出宝贵的资源来换取测试员所建议的好处。对于程序错误，资源就是时间和资金，好处就是通过改正这个具体错误而带来的质量改进。

销售策略一般包括两个目标。

陈述种种好处，使得潜在客户想要它。测试员的错误报告应该使读者明白为什么要改正这个错误。例如，可以解释问题会怎样影响产品的正常使用，会破坏什么数据，或人们如何经常遇到这个问题。测试员可以利用杂志上的评论或其他出版物中的有关批评，指出类似的问题给竞争对手带来的麻烦。可以引用技术支持统计数据，说明其他产品中的类似问题所带来的资金损失。还可以说明这个程序的以前版本通过了这个测试。（在有些公司中这是一个关键问题。）有人喜欢自行在产品中引入一些特征，这些特征可能有问题，要保证报告会引起这些人的注意。在很多情况下（参见下面的讨论），从看起来相对较小的错误开始，通过后续测试能发现更严重的后果，不应该报告所看到的错误的第一个版本。

向销售人员说明预期存在问题，并反驳他们。有些问题会由于多种原因不被考虑：问题太小、不可重现、不能理解、在实际环境中不太可能发生、问题只出现在没有人有的非常特殊的设备配置上、改正错误风险太大、不会影响产品的实际用户等。通过养成良好的报告编写习惯，测试员可以例行地说明这些潜在的问题：文字明确而简单，核实（并报告）程序错误出现在多种配置上。其他问题会随程序错误的不同而不同。测试员可以在错误报告中预测某种问题，并提供相关信息。也可以等一等，看看大家最初对报告有什么反应，在评审该程序错误时再提供补充信息。我们并不是说测试员要在报告中为程序错误辩解。尽可能避免这样说，“我知道你们在考虑不改正这个错误，但是，如果你们是因为某某原因而认为它不重要，那么你们应该知道……”相反，如果认为这是一个关键问题，则应该提出建议，并给出一些有关事实，例如“产品的第二版也有类似的问题。技术支持经理估计，由于这个问题，造成10万美元的技术支持开销。”

错误报告代表的是测试员

当别人收到并阅读错误报告时，测试员一般都不在场。写错误报告，就是在要求程序员（他们并不归测试员管）做一些补充工作。程序员很少有足够时间改正每个错误。改错常常要占用程序员自己的很多时间，例如下班之后或周末。这也就是说，测试员在要求程序员放弃自己的时间来修改所发现的错误。

在有些公司（特别是当项目接近尾声时），由各种经理确定要改正什么。决策小组可能叫做“变更控制委员会”。（在另外一些公司，这种决策小组叫做项目小组（project team）、筛选小组（triage team）、作战小组（war team）或程序错误评审小组（bug review team）。）这些人知道每个变更都需要资金、时间投入，以及带来影响其他部分的风险。

为了改正错误，测试员必须说服变更控制委员会批准变更，或说服程序员自己来改正错误（也许要在变更控制委员会不监视的深夜进行[⊖]）。错误报告是程序员说服别人改正错误的主要（常常是惟一的）途径。测试员在变更控制委员会会议上可能有辩解的机会，但是在很多公司中，只有一个测试员（可能是测试小组领导或测试经理）参加这种会议。这个人会怎样为测试小组所发现程序错误辩解，取决于测试员所写的错误报告。

努力使错误报告有更高的价值

由于有那么多的人要阅读并依赖错误报告，因此要下功夫丰富每个错误报告的信息，提高报告的可理解性。这会为公司带来价值。

在大多数公司中，错误报告有多种用途。例如：

- 报告提示大家注意缺陷，并帮助程序员定位内部问题。
- 报告提示大家注意规格说明和（取决于公司政策）测试文档、用户文档或开发工具中的问题。
- 为技术文档编写员提供背景信息，编写员要编写手册或公司网站中的问题定位部分。
- 报告提示需要通过客户培训解决的问题。

⊖ 让程序员背着变更控制委员会改进产品，对有些公司来说是正确的策略，而对另一些公司来说则是错误的策略。请读者注意自己的企业文化。

- 报告为客户售后支持人员提供关键信息，他们会遇到还没有被解决（真糟！）或没有完全解决的问题。
- 报告向管理层提供正在开发的产品的状态和质量信息。
- 报告在开始计划产品下一个版本时，提供初始改进建议。

产品的任何项目相关人员都应该能够报告程序错误

产品的项目相关人员是对产品的成功有既定利益的人，可以是开发该产品公司中的员工，也可以严重依赖该产品的客户或用户。

整个产品开发公司都要坚持在预算限度内，按时生产有合适功能的高质量产品。公司任何部门的员工发现质量有问题或功能有问题，都应该能够记录自己的看法，以某种方式提交给设计和实现团队。测试小组永远也不要干扰这种沟通，而应该促进这种沟通，即使是——尤其是——抱怨来自产品开发团队外部（包括测试小组）。

测试员不应该把程序错误跟踪记录给所有人看。如果把源代码和其他开发决策文件作为机密，那么也应该把程序错误跟踪数据视为机密。但是，不同类型的人应该能够得到通过测试员的记录加工的信息。

引用别人的错误报告时要小心

如果没有得到允许，可以补充评论，但不能编辑别人的材料。不要在不是自己写的、不是自己批准的文件上署名。对于公司中其他人的错误报告也要这样，即使错误报告很糟糕也不要擅自修改。

在没有得到允许的情况下擅自引用还有遗漏重要信息的风险。

任何时候要在错误报告，特别是其他人的报告中做补充，都要注明自己的姓名和日期。例如，一般要在报告中另起一行补充信息，并以类似“[CK 12/27/01]”这样的标记开头。这还有助于别人更容易地找到合适的人，进一步了解有关补充细节。

将质量问题作为错误报告

质量对于个人来说就是价值（Weinberg，1992）。

不同的人对于产品有不同的期望，但是如果产品的合法项目相关人员由于产品做的或没做的事而对产品感到失望，感到产品不那么有价值，那么应该作为错误报告写出来。

测试员的任务就是帮助这样的人写出报告，清晰、有效地表达他们的意见。

有些产品的项目相关人员不能报告程序错误，测试员就是他们的代理

成品（off-the-shelf）软件的客户（例如消费者）在开发期间不能报告程序错误，因为他们还没有得到软件。产品的其他的项目相关人员也可能不在现场。测试员要顶替不在的项目相关人员。为了站在他们的立场上理解要报告的内容，并使报告具有说服力，测试员必须研究用户使用产品的方式，以及他们喜欢这种产品的什么，不喜欢什么。

将受到影响的项目相关人员的注意力转移到有争议的程序错误上

如果测试员认为很难说服程序员改正错误，但是测试员希望改正，可以考虑公司中如果这个错误被改正能够受益的其他人。例如，用户界面的不一致对于程序员来说可能是个小问题。试图与一些项目经理理论这类问题是浪费时间。

不一致性会增加文档编写、培训和技术支持的成本。不仅如此，如果在产品演示时不能避免出现这种问题，销售量也会受到影响。如果这个问题影响到可获得性，则要求与ADA兼容的政府机构就不能采购该产品。如果产品评论员发现这个问题，会通过媒体对产品造成不好的影响（把该产品描述为设计差或难使用的产品）。这些损失都不会影响程序员的预算。

写入错误报告（或附在错误报告后面的备忘录），以引起其预算会受到这个程序错误影响的人的注意。他们会为测试员争辩必须改正这个错误。如果他们认为这个错误产生的影响不大，就不会倡导改正，也许测试员应该撤掉对这个错误的修改要求，转而要求修改其他错误。

决不要利用程序错误跟踪系统监视程序员的表现

测试员非常想报告某个程序员有大量程序错误没有修改，或程序员修改代码的时间太长，或程序员总试图推迟修改，这种愿望太强烈了。但是，使用这种

跟踪系统时间估计程序员，或使程序员感到为难之后，其他程序员就会防备这个系统。最有可能的结果是，程序员争辩设计问题并不是程序错误，类似的错误是重复的，不应该报告不可重现的程序错误，测试员不胜任或测试不公平。这并不是没有道理的。只要测试员把程序错误跟踪系统用于行政或人力资源管理，而不是技术管理，人们就会这样对待它。

决不要利用程序错误跟踪系统监视测试员的表现

如果测试经理根据测试员所报告的程序错误数奖励测试员，就会以测试经理所不希望的方式影响测试员。例如，为了提高程序错误数，测试员也许报告容易发现、更肤浅的程序错误，也许更愿意报告同一个程序错误的多个实例。他们会不那么愿意花时间指导其他测试员，不那么愿意花时间使用程序错误跟踪系统或其他测试小组的基础设施。程序员更有可能不承认测试员为了提高程序错误数而统计在内的设计错误为程序错误。

及时报告缺陷

不要等到第二天或下周才报告程序错误，不要等到忘记了一些关键细节才报告。拖延的时间越长，程序错误被解决的可能性就越小。

拖延报告的另一种风险是：如果经理知道测试员在测试产品的某一部分，而且没有看到错误报告，他们会错误地认为测试员没有发现程序错误，那么这部分功能一定很稳定。

永远不要假设明显的程序错误已经写入报告

别人也会做同样的假设，结果是一些明显的程序错误一直没有报告，直到被β测试员发现。如果认为某个程序错误可能已经被报告，可利用跟踪系统来察看是怎样描述的，以及反应是什么。也许自己可以通过补充注释或增加信息来强化错误报告。如果老的错误报告写得太弱或被拒绝，则有必要写新的错误报告。

对于一些严重程序错误，我们见到过这种情况发生。很多人都知道该程序错误，但是所有人都假设其他人已经报告了。有一次，直到产品被交付后才真正修改。

报告设计错误

计算机程序为人们带来好处，那是它们的工作。程序是包括设备、其他软件和人员的系统的组成部分。如果软件产品难以使用，使用时容易搞乱，不能与其他软件协同工作，或只捆绑在范围很窄的硬件上，那么软件产品的价值就会降低。测试员可能是开发团队中看到或评估设计错误及其对现用系统影响的惟一成员，如果测试员不报告设计错误，谁还会报告呢？

有人（包括一些测试顾问）认为测试员不应该在错误报告中包含设计错误，因为程序的设计在项目初期就应该已经定下来，测试员没有专门知识评估设计。我们认为对于这两个方面，这都是一种错误观点。

关于设计的后期评判。即使产品事先已经完全设计好（在现实世界的真实项目中不太可能），人们也要到系统构建完成时才会充分理解系统的含义。只有当开始使用系统时才会发现系统设计错误，这完全是合理的[⊖]。当发现设计问题时必须报告，很多设计问题要到系统几乎完成时才会被发现。

关于专门知识的问题。的确，测试员理解和评估设计决策的能力是不同的。另一方面，测试员是公司中为数不多的在软件销售或正式使用之前，从头到尾考虑完整系统的人。如果测试员没有相关的专门技术，则需要小心。例如，测试员在批评程序的用户界面之前，也许应该核对一下针对该系统编写的用户界面设计指南[⊖]，并向比自己更了解设计问题的人咨询。不过如果确信自己发现了一个错误，可将其放入跟踪系统中。

测试小组通过聘用具有不同背景的员工，可以增强自身评估设计错误的能力。具有领域专门知识的测试员（了解该产品将怎样使用、为什么这样使用的深入知识）能够将测试和解释集中到会为合理用户带来不便的问题上。这只是对测试小组有帮助的与主题有关的专门知识的一个领域。如果一个测试员了解数据库设计，另一个测试员了解网络安全，还有一个了解用户界面，等等，测试小组作为一个整体，就可以对产品的不同设计方面做出有见识、有价值的评估[⊜]。

⊖ 这也是现代开发方法，例如极限编程（Extreme Programming）和Rational统一过程（或更一般地，把自己称为迭代式、适应式、进化式或敏捷式的方法）被引起关注的原因。这些方法都假设会出现后期需求变更，并寻求最小化与变更关联的风险。请参阅Beck 1999、Kruchten 2000和www.agilealliance.org。

⊖ 苹果、微软、Sun和其他一些公司，都发表了针对图形用户界面的用户界面设计指南。

⊜ 在项目需要的时候，总是不可能把合适的人都集中到测试小组中。可能不得不让外界，即承包商或顾问培训员工，或进行某种类型的评估。

看似极端的缺陷是潜在的安全漏洞

通过因特网的绝大多数入侵，都利用了缓冲区溢出（buffer overrun）缺陷（Schneier 2000a和2000b）。只要测试员不是忙于检查他们“知道”程序员会忽略的极端情况，在产品交付之前本来是可以发现这些缺陷的。可以影响程序操作或导致数据被破坏的任何问题，都是一种伺机利用。

如果输入存储区的数据比所分配的尺寸大，就会出现缓冲区溢出。超出的数据会溢出到其他存储空间中。会出现什么情况，取决于这是什么数据以及该存储空间的使用。有时什么也不会发生，有时在屏幕上会看到乱七八糟的东西，有时程序运行很怪异，有时导致崩溃。熟练的攻击者可以利用这种漏洞作为后门，得到对程序或运行该程序的系统的控制。

假设通过在预期接受一个1~99的字段中，输入65536个9会导致程序崩溃。会有人真的这么干吗？是的，有人当然要这样做。大概不会是自己的朋友，也不会是放言“如果有人愚蠢到会这样做，程序崩溃会教训他”的忽略该缺陷的程序员。但是白痴并不是惟一滥用程序的人。

任何会产生严重后果的问题都应该解决，不管其多么“不可能”发生。当熟练的攻击者利用程序中的缺陷得手后，会写下这个消息并广为传播，使得所有脚本生手都可利用。

使冷僻用例不冷僻

为了提高效率，常常要用到极值。测试员所发现的第一个错误常常是在边界处。极值测试的思想是，如果程序能够在这种条件下存活，那么在不那么极端的情况下也可以存活。因此，可以通过少量极端测试了解很多东西。例如，为了测试期望接受一个0~999的输入字段，也许要检查-1、0、999和1000。

有些程序员把涉及极值或极值组合的测试结果抱怨为“冷僻用例（corner case）”。这些程序员把这种测试看做是不现实的，在现实世界中不太可能发生。一般来说，冷僻用例都被用来作为不修改缺陷的借口。

假设处理0~999数字输入的程序拒绝999。如果出现这种结果时停下来，有些程序员会忽略它（“谁会输入999？那是最大的可能值。不会有人输入999的！”）。

出现这个结果时不要停下来，尝试一下较小的数字，直到确定预期可接受取

值的范围。如果程序拒绝所有大于99的数字，则可以报告说程序在100~999整个范围内都是失效的。这个结果就会引起关注。如果程序只在999上失效，那也是有用的信息。要明确地写入报告。

与所有后续测试一样，测试员没有多少时间可以花在这样的问题上。如果认为这种检查占用的时间太长，得不到回报，则停止测试并寻找新的缺陷。如果公司内的程序员总是认真对待极值测试，则可以直接报告问题，不必再确定范围。

小缺陷也值得报告和修改

小错误会使客户感到困惑，并降低客户对产品其他部分的信心。被认为是很小的缺陷可能包括拼写错误、小的屏幕格式问题、鼠标遗迹（鼠标箭头移开之后留下的小点）、小的计算错误、图形比例不太准、在线帮助错误、不适当地灰掉了（或应该灰掉而没有灰掉）的菜单选项、不起作用的快捷键、不正确的错误信息、极值处理错误、超时处理错误，以及其他程序员认为不值得花精力修改的缺陷。

几年前，我们中的Kaner与David Pels合作研究了小缺陷对技术支持成本的影响问题。Kaner曾经是软件开发经理，Pels负责技术支持。Kaner和Pels选择了一种销售量很大的大众市场产品进行研究，审阅了来自客户的所有信件和电子邮件、杂志评论、客户电话记录以及缺陷跟踪系统的记录。他们把能够在4个小时（不包括测试时间）之内修改的缺陷叫做“低成本修改”。修改可以是代码变更、驱动器或磁盘数据文件替换、对用户手册或产品包装上的文字修改。只要修改能够在4个小时之内完成，并能够杜绝客户对这个问题的后续抱怨，就可以算作一个低成本修改。根据这种标准他们得出结论，（小缺陷的）低成本修改可以避免该产品一半以上的技术支持电话。（他们在Kaner和Pels 1997中简要讨论了这部分工作。）在下一年中，Kaner的开发小组解决了这些问题中的很多，每个新客户的技术支持成本下降一半。这并不是一种完美的因果关系，因为Kaner的开发小组还修改了一些重要错误，并增加了一些新特征（以及新程序错误），但是（Kaner和Pels认为）很明显，这类一般性的修改对客户满意度有重要影响。该产品当年的市场份额显著增加。

任何产品都会残留一些小缺陷。但是随着小缺陷数量的增加，客户信心会下降。更糟糕的是容忍这些缺陷的腐蚀作用。当甚至连小缺陷的报告都停止后（有些公司要求其员工只报告“值得报告”的错误），测试员和公司就会容

忍越来越多的严重缺陷。长此以往（除非在市场中处于垄断地位），最终会使产品失败。不断容忍更大的错误是造成挑战者号航天飞机空难的一个重要因素，Richard Feynman在他的《你在什么方面在乎别人想什么》（What Do You Care What Other People Think）（Feynman 89）一书中，对这个问题做了生动说明。

时刻明确严重等级和优先等级之间的差别

严重等级（severity）表示程序错误的影响或后果，优先等级（priority）表示什么时候公司要求修改错误。除非了解到有关隐藏后果的更多信息，否则严重等级是固定不变的。优先等级随项目的进展而发生变化。一般来说，更严重的问题有更高的修改优先等级，小的装饰性问题有低的优先等级。但是，情况并不总是这样。严重问题可能不值得解决，例如假设发现某个程序错误会损坏1999年12月以前新创建的记录。几年前这会是高优先等级的错误（那时1999年12月还是未来的日期），现在它仍然是严重的，但是很多公司都不再修改了。

另一方面，在启动软件时出现的屏幕背景中有公司标志，但是公司名称写错了，这纯粹是装饰性问题。但是大多数公司都会把它当作高优先等级缺陷。

失效是错误征兆，不是错误本身

当看到失效时，看到是程序的错误行为，而不是代码中的内部错误。缺陷看起来可能很小（例如鼠标遗迹），但是内部问题可能会严重得多（例如指针越界）。如果在略有不同的情况下遇到了相同的错误代码，则所看到的错误行为可能会糟糕得多。

因此，任何时候看到看起来很小的错误，都要：

- 执行后续测试，以发现更多严重征兆，最终产生更引人注意的报告。
- 执行后续测试，以发现在更宽范围并且会被很多人看到的条件下，该问题的出现情况。

如果问题很难重现，可：

- 执行后续测试，以确定使该问题重现的关键条件，然后执行后续测试，以充分暴露现在已经可重现的问题。

针对看起来很小的代码错误执行后续测试

当发现编码错误时，程序所处状态是程序员不想要或可能没有料到的。数据现在也可能是预期不可能的值。如果在第一次失效后继续使用这个程序，任何情况都有可能发生。

如果看到的是小失效，不要只是重现该失效并写入报告。程序处于脆弱状态，尝试利用这一点，继续测试，并可能发现内部缺陷的实际影响是糟糕得多的失效，例如系统崩溃或数据损坏。

我们建议尝试三种后续测试：

变化自己的行为（通过改变操作方式改变条件）。例如，让程序不断反复进入失效状态。有累积影响吗？尝试与该任务相关的失效操作。例如，当在屏幕上增加两个数字时，如果程序意外地稍微上移一点显示，则尝试测试这种缺陷是否影响加法或影响数字。尝试与该失效相关的操作。如果失效是在做了加法处理后显示意外上移，可尝试先上移显示再做加法。尝试刷新屏幕然后再做加法。尝试重新定义数字显示的尺寸后再做加法。也可以尝试更快速地输入数字，或以某种方式改变测试活动的速度。当然，还可以尝试各种其他探索式测试手段。有时这种方法很有用：运行很多完全无关的其他测试，直接在不重新启动或重新设置的情况下继续使用程序。

变化程序选项和设置（通过改变被测程序改变条件）。典型的变化是使用不同的数据库、改变持久变量的取值、改变程序使用内存的方式，或改变程序允许测试员修改的任何其他选择（任何其他选项或偏好或设置）。在屏幕上移例子中，也许可以改变程序的窗口尺寸、要显示的数字精度或拼写检查器的背景活动设置。

变化软件和硬件环境（通过改变与被测程序一起运行的软件或正在使用的硬件改变条件）。这不是配置测试，不是要检查标准配置上的缺陷，而是要调查怎样变更配置才会使这个失效暴露充分。例如，如果认为时序可能是个因素，就使用不同速度的处理器或视频显示或通信连接。如果认为内存可能有关，就使用内存较少（或较多）的计算机，或改变虚拟内存设置。

后续测试可以一直进行下去。应该做多少呢？对于每个认为反映了编码错误的失效（没有体现程序员意愿的代码）至少要做几分钟的后续测试。

对于有些失效要多做一些后续测试，可能需要一天的时间。要相信自己的判断。如果认为再做一些测试可能会发现有价值的新信息，就继续测试。如果认为现在对该缺陷的了解，通过继续测试也不会再深入了，就停止这种测试，并编写错误报告。要相信自己的判断。

永远都要报告不可重现的错误，这样的错误可能是时间炸弹

不可重现的错误会是公司能够交付的最昂贵的缺陷。有时程序的表现没有办法重现。看到程序错误一次，但不知道如何使其再次出现。如果产品交付客户后还出现这种情况，会影响客户对产品的信心。如果技术支持人员需要很长时间评估客户的数据或环境，客户就会更加厌烦。

程序员拥有测试员所没有的工具。如果测试员清晰地报告征兆，程序员常常通过研究测试员怎样得到的特定消息，或当测试员查看对话框或点击特定控件时可能出现的情况，从而能够跟踪到代码。相信程序员能够改正报告给他们的“不可重现”缺陷中的20%，并不是不合理的[⊖]。

即使所在的公司把忽略不可重现的缺陷作为政策，我们也仍然建议测试员报告不可重现的缺陷。不要报告还没有尽力重现的程序错误。但是如果尽力之后仍然不能重现该程序错误，仍然值得报告。定期浏览数据库，检查不可重现缺陷的模式。同样的问题可能出现多次，并（如果坚持报告不可重现的程序错误）被报告多次。这些报告都会略有不同。将其联系起来分析，可能会得到内部条件的强烈提示。

在报告不可重现的程序错误时，要明确说明自己不能重现这个程序错误。有些跟踪系统有专门针对这种情况的字段（能够重现这个程序错误吗：是/否/有时可以/不知道）。有些公司是靠约定，例如在程序错误的小结行或描述的第一行中加NR（不可重现）标记。

使用PrintScreen、类似Spector的屏幕记录器，甚至摄像机，都有助于注明一些程序员可能会说决不会发生的“不明飞行物（UFO）”的存在。

不可重现程序错误是可重现的

程序错误要在特定条件下出现。如果测试员知道这些条件，就可以重现程序错误；如果测试员不知道关键条件，也许就不能重新产生该失效。

如果遇到自己不能重现的程序错误，一定是有什么重要的东西自己没有注意到。如果可以找出条件是什么，就能够重新产生该失效。以下是一些我们认为有助于思考的条件例子（人们常常没有给予足够重视，只是在回忆时才意识到）：

⊖ 显然，这种百分比会随产品、程序员可用的工具包的不同而不同。我们见到过程序员在商业软件中达到这样的成功率。可靠人士告诉我们，有的程序员和项目团队能够达到80%的成功率。

- 程序错误可能有延迟效应，例如内存泄漏、指针越界或栈被破坏。如果怀疑不断严重的问题，可以考虑请程序员使用Bounds Checker（www.numega.com）、Purify（www.rational.com/products/purify_unix/index.jsp）或类似的工具，也可以利用各种方法连续监视软件行为变化（将程序输出录像，使用实用工具每秒钟捕获一次屏幕，监视内存使用情况，等等）。
- 程序错误可能只是在安装、使用产品或使用产品的特定功能时出现一次。使用Drive Image（www.powerquest.com/driveimage/）、Ghost（www.symantec.com/sabu/ghost）或类似工具，有助于创建干净系统的确切拷贝（即还没有使用过该应用程序的系统）。恢复干净系统，重新装载该应用程序，检查现在是否能够重现该问题。
- 程序错误可能依赖于特定的数据取值或被破坏了的数据库。
- 程序错误可能只在特定的时间或日期内发生。检查日末、周末、季度末和年末程序错误。
- 缺陷可能依赖于以特定顺序执行一系列相关的任务。在执行这个失效任务前还做什么了？
- 程序错误可能是前面失效的残余。例如，上一次出现GPF后重新启动计算机了吗？
- 程序错误可能是由被测应用程序与后台运行的其他软件，或和与被测应用程序竞争设备访问的软件的交互引起的。也许程序错误反映的是与其他设备交互时产生的没有预料的问题。

还有一些其他想法，但是对于本书来说这些想法太多、太细了。我们在《测试计算机软件》（Testing Computer Software）（Kaner等，近期出版）新版本的第一卷（测试员卷）对这方面的问题做了全面综述，同时，也可以参考Nguyen（2000）、Kaner等（1993）、Telles和Hsieh（2001）中的有用思想。

注意错误报告的处理成本

我们建议读者报告所有小问题和不可重现的问题。下面我们必须退一步，在判断报告的内容和如何报告上做一点讨论。

错误报告有处理成本。编写报告需要时间，接下来程序员和项目经理要阅读报告。在有些公司中，全体变更控制委员会要评审程序错误，以确定处理意见。在另外一些公司中，只有当程序员或项目经理决定不改缺陷时，报告才会被提

交给变更控制委员会。另外还会有很多读者。把所有人的时间加起来，会发现公司需要2~8个人时评审和拒绝小缺陷。在项目后期，如果向公司报告很多小缺陷，就会大大影响程序设计和项目管理小组的生产率。通过在项目后期报告非常小的问题，或编写不清晰（浪费时间）的错误报告，测试员都可能会引起很多怨恨或愤怒。

当报告小缺陷时，要特别注意使报告简明，并经过充分分析（请参阅本章稍后后续测试部分）。如果要在项目后期报告一组小缺陷，可与测试经理讨论变通策略。也许可以把错误报告放在次级数据库中，将作为新版本的输入而正式评审，但是当前版本不予考虑。这样做可以保护测试员的工作，但是只在新版本中考虑。

当要报告一个不可重现的程序错误，特别是在项目后期时，需要格外注意努力重现该程序错误。如果做了大量后续测试仍然不能重新产生该程序错误，则明确说明这个程序错误是不可重现的，并描述所做的定位工作，以及所遇到的程序错误征兆。

特别处理与工具或环境有关的程序错误

如果由于操作系统，或者另一个应用程序或被测程序要与之通信的已知弱点，而造成程序失效，而失效完全在程序员的控制之外，决定不报告这样的程序错误是合理的。另一方面，测试员也可能发现通过加一些适当的错误处理代码就可以很容易改正的缺陷。

但是，由于与操作系统（或其他产品或系统）中的已知弱点密切交互，程序可能会失效。例如，也许程序员可以修改程序，以便避开关键的系统调用，因此不会使系统崩溃。这种程序错误部分是操作系统程序错误，但是由于可以在应用层上避免出现失效，因此也是应用程序错误。如果测试员认为是这种情况，就可以这样报告。在错误报告中，总是要首先报告重现该问题的步骤，但是要说明已知这种程序错误看起来涉及应用程序和操作系统之间的交互，并建议应用程序能够避开操作系统级缺陷。以这种方式报告很多程序错误之前，可考虑先与对下层系统很了解的人交流。

还可以考虑使用适当的工具对失效所出现的配置做快照。例如，Rational公司发放一种免费的安装分析器、可记录Windows安装详细信息，并与一个参考版本进行比较（VeriTest-Rational Install Analyzer，通过以下网站可得到：www.rational.com/products/testfoundation）。类似工具还有InCtrl5，可通过

zdnet.com获得。

类似地，程序员常常把失效归罪于集成到应用程序中的第三方代码。在所有情况下都要报告这些程序错误，因为它是公司交付给客户的代码包中的错误引起的失效。如果有理由认为程序员能够解决他们所使用的第三方代码中的问题，则应该提倡修复。

在报告原型或早期个人版本的程序错误之前，要先征得同意

有时程序员会给测试员一个个人版本，并要求进行单独测试。如果公司允许这样做，则要尊重这种隐式规则。在评审早期代码时所发现的问题，也许在程序错误跟踪系统中没有报告，而是通过与程序员谈话、通过向程序员提供程序错误笔记或发电子邮件来报告程序错误。如果测试员公开程序员早期、私下提供给测试员的代码版本程序错误，就会丧失程序员的信任。最有可能的结果是，测试员不再有机会看到这些早期版本了。

到一定时候，程序会在公司内部公开，准备进行常规测试。可使用在个人评估程序时记录的笔记。以前发现但没有被修改的问题，都应该进入跟踪系统。

我们必须提醒读者注意不要滥用这种安排。我们曾经遇到过有程序设计小组试图通过隐藏其缺陷来掩盖其进度问题。他们把软件的最终版本之外的所有测试版本都称作原型或个人版本。除了最终版本，不想将任何错误报告录入跟踪系统（管理层会通过跟踪系统查到错误报告）。

如果测试员遇到这种情况，要与自己的经理沟通。测试经理可能不得不与项目经理的经理或其他关键项目相关人员沟通。问题是公司把早期版本的程序错误跟踪数据，看做是主要供程序设计小组使用的工具，还是看做是关键项目相关人员有权查阅的公司资源。努力合作并提供帮助，不要做公司会认为是试图掩盖事实的事。

重复错误报告是能够自我解决的问题

当然，通常更好的作法是把数据加到现有的开放错误报告中，而不是写一份自己认为是相同程序错误的新报告。如果知道正在报告的是与另一个程序错误类似的程序错误，可做交叉引用。

在有些公司，尤其是过于重视程序错误统计和程序错误缺陷的公司中，项目

经理和程序员会对重复错误报告感到不满，或与管理层争辩，认为程序错误统计会由于重复而膨胀。在这些公司中，可快速搜索一下相同程序错误的报告。但是，

- 不要让搜索时间失控。如果数据库很大，可能要花大量非生产时间搜索重复程序错误。要控制自己的时间。
- 编写良好的相同程序错误的所有报告都包含新信息，有助于更容易地清除该程序错误。
- 两个报告是否重复还需要统一认识。两个失效可能是由同一个缺陷引起的，多个缺陷可能涉及到一个失效。要保留所收集到的所有信息，直到认识统一并修改了缺陷（Pettichord 2000a）。

对重复程序错误使程序错误数膨胀感到不满的项目经理，应该通过修改这些缺陷很容易地解决这个问题。

每个程序错误都需要单独的报告

不要努力地把不同的程序错误合并到同一份报告，来减轻项目经理或程序员对重复错误报告的不断抱怨。如果把多个程序错误写到一份报告中，有些程序错误就可能得不到修改。当把数据库中的程序错误与要报告的程序错误进行比较时，可使用这个准则：如果要运行不同的测试检查要报告的程序错误和数据库中的程序错误是否被修改，则将两个程序错误作为不同错误报告。

Pettichord建议以稍微不同的方式管理类似程序错误：

有时为了提高效率，可能会在一份报告中包含多个相似小程序错误，假设这些程序错误可以一次全解决。如果当该报告被打上已修改标记之后，还有没有修改的残余程序错误，可以为这些没有修改的程序错误写一份新报告，并引用已修改标记的老报告。

归纳行是错误报告中最重要的部分

归纳行有时又叫做主题或标题，是至关重要的，因为项目经理、其他经理和执行经理检查还没有修改或不打算修改的程序错误表时，要看到归纳行。较弱归纳行中的程序错误在程序错误筛选会议上会被放弃。（在程序错误筛选会议上，项目团队决定要以什么优先顺序修改哪些缺陷。）执行经理和程序设计小组之外

的其他经理更有可能花更多的时间关注带有重要归纳行的程序错误。归纳行是向这些经理推销程序错误的最佳工具。

好的归纳要向读者提供足够的信息，以帮助读者决定是否索取更多信息。归纳行应该包含：

- 简要描述，要足够具体，使读者能够想像出该失效。
- 简要指出程序错误的局限性或依赖关系。（涉及这个程序错误的环境有多宽或多窄？）
- 简要指出程序错误的影响或后果。

不能把所有这些信息都放入归纳部分，因为归纳只能有一行长（也许只有65个字符）。（可以输入更长的归纳，但是列出多个程序错误的管理报告通常只显示一行归纳，所输入的超过一行的信息部分不可见。）挑选对于报告最重要的信息，把其他内容放入报告的信息描述部分。

不要夸大程序错误

测试员的可信性是影响力的基础。如果测试员使所报告的程序错误看起来比实际情况更严重，就会失去影响力。

公司都有程序错误严重等级模式，例如小程序错误、严重程序错误和关键程序错误。我们不想在这里定义严重等级，因为各个公司的定义有很大不同。不管自己公司的规范是怎样定义的，都要按要求使用。不要只是为了提高别人的注意力而把普遍认为的“小程序错误”定为“严重程序错误”。如果认为公司的严重等级分类模式不能正确地用于所发现的程序错误，可以使用自认为是合适的定级模式，但是要在问题描述的最后解释清楚。（“我知道通常这样的缺陷会被定为小程序错误，但是我认为这个具体程序错误应该定为严重程序错误，因为……。”）

清楚地报告问题，但不要试图解决问题

测试员的工作是报告问题，而不是确定根源，不是奋力争取具体的解决方案。

如果不研究内部代码，就不可能知道失效的根源。我们常常看到错误报告过多地关注测试员（不正确）的原因理论，他们没有报告足够的使程序员马上就

能够理解测试员所看到的东西的实际数据。

有些程序员可能会拒绝自己认为无效“解决方案”的报告，而没有考虑其中的问题。不要让人轻易地拒绝报告。

决定适用于特定程序错误的解决方案是产品设计师的事。例如，假设测试员发现一条当用户一移动鼠标就立即消失的错误信息，这使得很难阅读消息内容。测试员很想在错误报告中说：“这个错误消息应该出现在对话框中，在显式地关闭之前一直在顶层显示。”如果这样做了，当设计师返回一个便条，告诉测试员管好自己的事时，也没什么好奇怪的了。事实上，关于错误消息问题有多种解决方案，设计师会挑选自认为最合适的解决方案。

面向解决方案错误报告的另一个问题是，很多测试员对提出解决方案太感兴趣，以至于没有提供有关失效本身的清晰、准确的信息。如果测试员错误地理解了失效的内部原因，所提出的解决方案最多是无用的，有时由于测试员无意地省略关键信息，或通过报告把程序员的注意力吸引到错误的细节上，这样的报告比无用还糟。类似这样的错误报告有时会成为程序设计小组的笑柄。没有人想让别人嘲笑自己的无知建议。

报告信息消失问题的一种更好方式是，“出现一个错误消息，但是我没能看清其内容，因为我一移动鼠标这个错误消息就消失了。”如果测试员与设计师有不错的关系，还可以加上一句，“如果这是模态对话框，也许能够解决这个问题。”请注意，这条建议听起来不像是命令。

如果项目团队都知道程序的某些部分以惟一一种方式运行，则报告问题和建议解决方案可能就是一回事。否则就是不务正业。

注意自己的语气。所批评的每个人都会看到报告

错误报告带有责备或居高临下的语气是没有益处的。称程序员不够专业、思想保守或愚蠢都是得不偿失的。测试员也许会舒服一时，但是会失去可信性，招致过于关注测试员的工作细节，修改所发现的很多缺陷的可能性降低。还要注意报告格式。例如，如果全部采用大写字母，则读起来像是撰写人在尖叫。如果测试员不清楚别人会怎样读自己写的报告，可请别人读，并仔细听取他们的意见。

如果语气对于测试员，或对于公司内的程序员是个问题，可尝试自己对着自己朗读报告。使自己的语言听起来带威胁、讽刺或冷漠语气。另一种检查语气

的方法是把报告草稿交给所信任的人审阅。

使自己的报告具有可读性，即使对象是劳累和暴躁的人

有大量缺陷是出为了完成项目而加了很多班的程序员在项目的最后几周修改（并延迟）的。对于困难的项目，程序员常常睡眠不足，承受很大压力，并受到太多的咖啡刺激。要编写使这些程序员能够理解的报告。

要使重现步骤的描述简洁：

- 一次只走查该程序错误一步。
- 为每一步编号。
- 不要跳过重现问题所需的任何步骤。
- 列出使读者能够重现失效的最少步骤集。
- 通过空行使报告更容易浏览。
- 使用简短句子。
- 说明实际发生了什么，自己预期会发生什么。
- 如果后果很严重，但是测试员有理由怀疑程序员不理解为什么很严重，可解释为什么自己认为很严重。
- 如果可以便于程序员意识到问题，或便于自己在修改之后重新测试，可补充注释。
- 对于复杂产品或问题，考虑使用描述的头三行专门小结该问题，然后给出问题细节。
- 要始终保持中立语气。
- 不要开玩笑，别人会产生误解。

提高报告撰写技能

研究跟踪系统中的错误报告，找出提高报告撰写技能的思路。例如：

- 比较已经修改了的程序错误和没有修改的程序错误，研究两者报告方式的差别。如果想让自己发现的程序错误也被修改，可像以前那些被修改的程序错误那样报告。
- 研究程序员（或其他人）对错误报告的反映。什么使他们困惑不解、不能接受、能够接受？

如果合适，使用市场开发或技术支持数据

如果可能，可把自己产品的表现与领先竞争对手的产品进行比较。这有助于描述用户预期，有助于市场开发经理（根据他们的视点）评估问题的严重等级。

调查销售人员和销售工程师在与客户接触时都遇到过什么问题，他们如何说明产品，他们要演示什么，他们想怎样演示。在报告中使用调查结果。

如果可能，将所报告的程序错误与相关或相似程序错误的技术支持记录联系起来。为了收集这部分数据，可能需要与公司技术支持人员密切合作。如果可能，可估计延迟修改程序错误的支持成本，或如果程序错误留在产品中将会面临的人员（客户或技术支持人员）的不满。

相互评审错误报告

有些测试小组在将错误报告提交给程序员之前，要安排第二个测试员评审所报告的缺陷。这第二个测试员要：

- 检查是否清楚地提供了关键信息。
- 检查是否可以重现该程序错误。
- 研究该报告是否能够简化、概括或加强。

测试员可以评审：

- 所有缺陷。
- 自己发现的所有缺陷。
- 同事发现的所有缺陷。

如果发现问题，可以找原始报告者讨论该程序错误。

- 如果报告者是测试员，可以指出该测试员深入培训目标的问题。
- 如果报告者不属于测试小组，可以向其核对基本事实。

这是一种培训员工、提高报告质量的很有用的方法，但注意不要过多地增加评审测试员的负担。评审过程需要时间。不仅如此，还应该决定什么时候评审者应该重现每个程序错误，什么时候只检查每个报告的严谨性和可理解性。

与将阅读错误报告的程序员见面

如果测试员认识将要阅读错误报告的人，就更有可能用关心、亲切的词汇写

错误报告，并研究如何能使报告更易于理解。如果不知道程序员是谁，测试员更有可能把程序员看做是傻瓜，而不是努力干好工作的容易出错的人。

最好的方法可能是向程序员演示所发现的程序错误

有些测试员一发现程序错误，就立刻找到编写这段代码的程序员，向其说明或演示该程序错误。根据讨论，测试员会进一步定位或客观写人报告。有些公司鼓励这种实践，有些公司不鼓励。我们总的说来喜欢这种实践，不过除非测试员已经与程序员建立了很强的合作关系，我们还是强烈建议在直接找程序员之前，还是要对所发现的程序错误下一点功夫（使其可重现，也许还要做一点后续测试）。与程序员越不熟悉，在与他见面之前越应该做更好的准备。

在测试复杂产品，并需要提供测试员不知道的数据时，这种方法特别有效。测试员可以从程序员那里了解情况，而程序员在系统失效时能够接触系统。

如果测试员找程序员时程序员很忙，则不要打扰程序员。但可以发送电子邮件，告诉他自己发现了重要问题，并希望在将问题放入程序错误跟踪系统之前先尽快与他谈谈。让他在有空的时候进行讨论。如果他太忙了，可不用再谈，直接写错误报告。

当程序员说问题已经解决时，要检查是否真的没有问题了

在很大的时间压力下，很多程序员都会采用最快的技术路线消除在错误报告中所描述的程序错误征兆。这种消除可能会也可能不会完全解决该程序错误。在重新测试声称已经改好的程序时，在略有不同的环境下可能还会发现它失效，例如采用不同的数据。也许还会发现这种修改会带来新问题。应该进行后续测试，以保证该征兆不会在其他地方意外出现。

尽快检验程序错误修改

当测试员被告知所报告的程序错误被清除时，要尽可能迅速地检验。随时注意检验程序错误时可表现出对程序员的尊重，也使程序员将来更有可能对测试员所报告的程序错误迅速做出响应。

如果迅速发现更改问题并迅速报告给程序员，他可能仍然记得自己是怎么改

的代码，并能够立即处理该问题。测试员等待的时间越长，程序员所记得的内容越少。（有关类似测试小组基本运作的有用建议，请参阅DeNardis 2000。）

如果修改出现问题，应与程序员沟通

如果程序错误修改反复失败，或在开发阶段的后期失败，不要仅仅通过错误报告反馈信息并在跟踪系统中归档，而应该直接找程序员。如果大家在同一座办公楼中工作，可以直接到程序员的工作间去。如果程序员离得很远，可打电话。（请注意，如果程序员在另外的公司工作，测试员需要得到项目经理的允许才能给他打电话。在这种情况下，首先把程序错误交给项目经理。）测试员的语气和态度应该友好、热心。不能对程序员这样说："真糟！糟糕的程序员！"测试员要帮助程序员立即得到信息，并准备随时澄清报告中的任何不清楚的地方，如果程序员想看，应准备随时演示该程序错误。

错误报告应该由测试员封存

当程序错误被标为已解决时，测试员应该进行检查。如果程序错误被标为已修改，测试员应该尝试检查修改是否彻底。如果错误报告由于不可重现或不能理解而被拒绝，测试员应该修改报告。如果程序错误被推迟修改，或作为非程序错误而被拒绝，测试员应该决定是否收集补充数据以对推迟和拒绝提出质疑。如果程序错误因为冗余而被拒绝，测试员应该决定是否赞同。有些项目小组通过把程序错误标为冗余而隐瞒程序错误。

在一般情况下，报告程序错误的测试员应该再次进行测试，但是如果程序错误是非测试员报告的，则应该由最熟悉程序这部分的测试员进行评估。如果可能，该测试员也应该咨询原始报告者沟通。

除非经过测试员的评审和封存，否则任何程序错误都不能标示为已封存。

不要坚持要求修改所有程序错误，要量力而行

有时不修改特定程序错误有很好的理由。最重要的理由之一是风险。每个程序错误修改（以及对代码的其他变更）都会产生新的程序错误。当程序员修改一个小程序错误时，有可能产生更严重的程序错误。如果在项目邻近结束时修

改程序错误，测试员在修改完成和产品交付之间，可能没有足够的测试时间找出更大的问题。由于害怕不能发现修改产生的副作用，有经验的项目经理会对代码变更很谨慎。

另一个很好的理由是客户不愿意为修改付费，定制、合同软件和内部软件开发尤其是这样。客户实际上要为测试所花的时间付费。有些程序错误的处理成本要低于修复成本。即使是远方客户也有价值权衡问题。请想像一下修复了删除客户数据的关键错误更新版本。包含缺陷的程序还在用户场地运行。假设推迟更新版的安装，以便修改拼写错误。对于这个更新版的答案是很明显的（不要修改拼写错误）。对于不太明显的决策，项目小组负责想像和评估这种价值权衡，以便在时间、成本、可用功能和可靠性几个方面找到合适的平衡点。

如果不能举出某个程序错误很重要的情况，或发现产品项目相关人员足够积极地支持测试员关于推迟特定程序错误修改的请求，我们建议还是把那个程序错误放在一边，先考虑其他问题。

不要让延迟修改的程序错误消失

延迟意味着所报告的程序错误确实存在，但不会在这个版本中修改。因此，在这个版本中由于延迟修改而封存的程序错误，在下一个版本的一开始就要启封。很多项目团队都设置程序错误跟踪系统，以便自动重新启封老的已推迟的程序错误，或将其（作为未解决程序错误）转到下一个版本产品的新文件中。如果在新的版本中，相关设计决策要重新审查，则他们还常常重新启封由于“符合设计要求”而被拒绝的程序错误。

已经有很多版本的产品，会重新启封一些令人烦恼的从来没有导致代码修改或文档改进的错误报告。有些项目团队由一些项目经理进行评审，并把一些程序错误作为INWTSTA（“我永远不要再看到它”）而永久封存。最好在项目一开始就做这种评审，那时的进度压力最轻，而且项目经理也处于最清醒、最理智的状态。

测试惰性不能成为程序错误修改推迟的原因

如果测试经理要求程序员不要修改程序错误（编码错误或设计错误），只因为修改会影响太多的检查单、脚本或其他测试工作制品，因此要占用太多的管

理时间，那么说明测试过程存在致命缺陷。

立即对程序错误延迟决定上诉

如果所发现的程序错误被推迟修改或被拒绝（程序符合设计要求），就要确定是否对这种决策上诉。有些公司允许把上诉当作项目小组会议过程的一个正常部分（程序错误筛选会议等）。在另外--些公司中，上诉采用测试员或测试经理与执行经理单独面谈的方式。

如果测试员决定对延迟或拒绝上诉，就要尽快。不要在决策制定的一个月后再提反对意见。除非有特殊情况，否则你是不会得到同情的。

如果决定据理力争，就一定要赢！

如果测试员确实要上诉，不要依赖自己最初错误报告中的语言和信息。报告是不可动摇的。如果测试员不列举更有效的例子，则不仅是浪费自己的时间，而且会损害自己的信誉。

为了准备自己的上诉，测试员应该：

- 与其他产品项目相关人员沟通，例如技术支持人员、文档编写员、销售人员等。找出如果程序错误留在产品则预算影响最大的人，确定该程序错误会增加他们多少成本，或给他们带来多大麻烦。
- 补充做一些后续测试，寻找该程序错误更严重的后果，或寻找比在错误报告中所描述的更广环境下出现的情况。
- 开发一些场景、情节，说明合理的用户在合理地使用产品时会遇到该程序错误。
- 寻找一些讨论与所报告程序错误类似问题的报刊文章。公司的竞争对手可能交付了带有类似程序错误的产品。如果这种程序错误在任何新闻报告中披露，就会成为应该认真对待该缺陷的很强的证据。

我们强烈建议读者要遵守的原则是，所做的每个上诉都必须是有说服力的。即使不能赢得所有上诉（当然不会赢得所有上诉），也应该得到自己的所有上诉理应获胜的好名声。

第 5 章

测试自动化

靠机器人做早餐，靠飞行汽车代步。那是科学幻想。但是软件可以做任何事。因此，为什么不让软件测试软件呢？按照同样的逻辑，既然一台计算机可以完成三百万数学家使用小棍在沙子上计算所完成的工作，那么一台计算机也肯定能完成许多测试员的工作。的确，测试自动化是一种大有前途的激动人心的想法。但是请注意，使一部分测试工作自动化可能有帮助，也可能没有帮助。自动化可以节省时间，加快开发，拓展测试员的能力，并使测试更有效。自动化也可以分散测试员的注意力，并浪费资源。

在测试自动化上投入的价值，是帮助测试员完成自己的使命。测试工作就是收集信息。自动化会提供什么信息？

有些人的自动化努力获得了巨大成功，而另外一些人则受到挫折。有些失败的人误导自己及其管理层，使其认为实现自动化的工作能够得到有用的东西。

如果自动化能促进测试使命的完成，就利用自动化。评估利用自动化是否成功的标准，是看它在多大程度上帮助测试员完成自己的使命。

本章初稿的审阅人分别强烈建议我们强调一些相反的观点：

在决定要自动化的内容时，首先设计自己的测试。这样可以避免落入自动化测试的陷阱：易于自动化，但是在找缺陷上很弱。

采用与设计手工测试不同的方法设计自动化测试。自动化测试的力量，很大程度上来自利用计算机完成人所不能做的事。寻找机会，例如能够对数以千计不同的数据文件重复测试。这样可以避免落入这样的陷阱：只通过现有（手工）测试计划自动化测试，并错失测试自动化的大好机会。在设计手工测试时，测试员不太可能考虑对数千文件重复进行的测试，只因为工作量太大。

这些矛盾消息源自以下两条重要的经验：

- 没有好的测试设计的自动化可能会产生大量活动，但没有什么价值。
- 没有很好理解自动化可能性的测试设计，可能会低估一些最有价值的自动化机会。

我们认为，为了可靠地获得成功，必须既有优秀测试设计者，也有能够选择和设计自动化测试的优秀自动化设计者。不过这说起来容易做起来难。

加快开发过程，而不是试图在测试上省钱

目标是降低测试成本的自动化工作，很少能够得到为了获得成功所需要的关注和合作。

如果要得到支持，就要把精力集中到降低开发失效的风险上。

测试用例在帮助测试员和测试小组获得关于被测系统的有用信息上，是强有力的。通过帮助测试员迅速收集和传播信息，迅速得到程序员的反馈，自动化测试能够增强测试员的能力。最成功的公司通过自动化测试实现其开发灵活性。他们的自动化测试部分目标是：

- 迅速检测出新版本中的不稳定的变更。
- 尽可能迅速暴露回归程序错误。
- 快速报告问题，因为这会使程序错误修改更容易。

快速修改会使代码稳定，使代码稳定会节省时间（不会有多人在相同程序错误上浪费时间），并促进通过重新分析或其他工作改进代码结构，并解决不稳定代码问题。如果代码基础大体是稳定的，并有很强的自动化测试包，则程序员可以尝试以较低的风险做更大的变更。项目团队还可以通过调整产品的范围和发布时间，迅速抓住市场机会。

以下是针对支持开发节奏的手段的两个例子：

- **自动化冒烟测试**。“冒烟测试”这个词来自硬件测试。测试员插入一块新板子并接通电源。如果看到板子冒烟，立即切断电源。不必再做进一步的测试。冒烟测试（又叫做“版本确认测试（build verification test）”）在有限的时间内（一般是一中午或一夜），广泛检验产品的功能。如果关键功能不能正常运行，或关键程序错误还没有被清除，测试小组就不再浪费时间安装或测试该版本了。解决这些问题对程序员来说也是同等紧急的事。
- **自动化单元测试**。这些测试也会使测试过程流畅、避免回退，并保持开发动力。这些测试有更大的测试集，针对的是被测产品中的下层功能和类。

自动化冒烟测试和单元测试的最大价值，在于可以由任何人在任何时候运

行。作为开发过程的一部分，自动地运行这些测试。开发这样的测试有助于个体程序员创建一些只包含了一个或少量变更的微版本。如果其中一个微版本失效，程序员会知道该检查什么地方。如果微版本没有问题，会向程序员提供更大的版本，其中包含了所有程序员所做的变更，这种版本出现问题的可能性也更低。这对于项目非常有好处，项目经理肯定欢迎。

创建这类自动化测试，需要时间、精力、技能和资金。单元测试可能要由程序员创建，尽管测试员也可以通过与程序员结成程序设计对子，鼓励程序员并加快测试开发进度。有了这些优势，与只是关注节省手工测试时间相比，测试员在确保协作方面要容易得多（Beck 1999，第20章）。

拓展测试领域，不要不断重复相同的测试

利用自动化拓展测试领域，使测试员能够看到更多、做到更多。

离开自动化简直就不能完成某些测试，而另外一些自动化测试则会大大扩展测试范围。以下是几个例子：

负载测试。200人同时使用被测软件会出现什么情况？2000人呢？需要自动化来模拟这些场景。

性能基准测试。系统的性能是越来越好还是越来越差？测试员可以通过自动化测试，在每次运行时都捕获时间度量参数。通过收集这些度量参数，并按时间顺序观察，就会发现性能退化现象。资源利用基准，例如内存或外存的使用，也可以通过同样的方法获得。

配置测试。软件常常必须在不同平台、不同配置上运行，挂接不同的外部设备。怎样才能测试所有情况呢？自动化测试有助于提高覆盖率。为了做到这一点，必须保证测试是可跨平台移植的。

耐力测试。被测产品使用几周或几个月会出现什么情况？出现内存泄漏、栈破坏、指针越界和类似的错误时可能不太明显，但是最终会带来麻烦。一种策略是中间不重新设置系统地长时间运行一系列测试用例，例如几天或几周。这要求测试自动化。

竞争条件。有些问题只有在一定顺序条件下才会发生。竞争同一资源的两个线程或进程的时序重叠，可能会产生叫做竞争条件的错误。这些问题常常很难发现，也很难重现。自动化测试会提供很大帮助，因为测试员可以利用很多略有不同的时序关系，反复进行测试。

组合错误。有些错误涉及多个功能之间的交互。可将大量复合测试自动化，

每个复合测试都以不同方式使用多种功能。

这些方法都集中于利用自动手段创建新的测试，或以旨在发现新程序错误的方式重复产品的使用。这些测试的实现都不那么简单，测试员必须通过自动化测试的不同部分，并开发辅助工具，逐渐实现。但是，我们认为这常常是自动化测试努力的更好目标，而不只是简单地一遍又一遍地重复相同的功能测试。

根据自己的背景选择自动化测试策略

自动化测试策略随测试需求、软件产品体系结构和测试人员现有技能的不同而不同：

测试需求。软件产品会有很多功能，但是往往只有少量功能是关键的。这些功能必须可靠，可能要求值得将其自动化的大量测试。另外，测试员也可以关注自动化测试能够如何帮助控制产品的主要风险。手工测试对产品其他功能来说可能已经足够好。

软件产品体系结构。分析产品体系结构以确定测试自动化的可能性。主要软件组件是什么？这些组件如何通信？使用了什么技术？有哪些可用接触点？体系结构描述软件组件及其与系统不同部件之间的关系。语言、环境和组件都会影响测试工具的适用性。所使用的接口决定自动化测试的机会（Hoffman 1999b）。

测试人员技能。有些自动化测试方法适合非程序设计测试员，有些则要充分利用测试员-程序员技能。我们中的Pettichord曾经被要求为某个在多个地点开发多种产品的公司推荐一种测试自动化方法。他没有推荐。有些开发地点的测试员能够编程，而另外一些地点的测试员不能。没有单一的方法能够适用于所有人。即使产品很相似，但是员工的定位和能力差别太大。

不要强求100%的自动化

有些软件经理错误地认为自动化测试总是意味着更好的测试，他们强制要求所有测试都应该自动化。这种要求对很多项目造成损害。

如果项目经理或执行经理容易上当，也许当别人给他演示最新测试工具，或讲解最新杀手锏测试方法时，应该有一位高级测试员在场。

如果有顾问或工具推销员声称行业领先的公司通常会达到100%自动化测试

率时，应该引起怀疑，让他们举出证据。很多行业领先公司成功地使用了混合测试技术，包括不可重复的提供最低限度文档的探索式测试。而且他们也应该是这样，自动化测试通常意味着更经常地运行更少的测试。很多测试只值得运行一次。

应该打破的另一个神话是，按下按钮测试就会开始，并自动地报告所发现的所有程序错误。提供其测试工具与跟踪工具间集成的工具提供商往往鼓励这种不负责任的幻想。我们从来没有看到过这会改进测试、报告或程序错误跟踪。如果在报告程序错误之前没有人来分析程序错误，就会浪费程序员的时间，并会引起程序员的反感，会由于不得不对付大量虚警和冗余而感到不满。

也就是说，在以下特定情况下，创建范围最大、最彻底的自动化测试包可以是合理的目标：

- 由于合同或一般原因，必须证明产品的最终版本通过很强的攻击测试。
- 通过对产品的测试确认客户能够自己运行。（这种实践在电信业很常见。）
- 产品对后向兼容性有严格要求，要求兼容很多版本。
- 某些产品只能通过执行自己编写的程序来测试。

但是，即使是对这些情况，我们也认为测试员总是可以进行一些非正式手工或单一使用、没有编制成文档的测试。不要对测试施加人为限制。

测试工具并不是策略

测试工具并不能教测试员如何测试。如果测试出现问题，则测试工具会加重问题。在实现测试过程自动化之前，应首先解决测试过程中的问题。

有些测试工具带有测试策略的基本建议。但是这种建议很少能够描述得很清楚，并不能针对具体情况，而且往往过于强调他们那种自动化测试的重要性。

我们接触过一些经理，他们想采购测试工具，而不是聘用测试员。（我们也接触过一些助长这种思想的工具提供商。）我们从来没有见到这种想法行得通的情况。如果采购这样的工具，解聘测试员，最终只能会有很多繁忙的工具使用者，有大量的活动，找到很少或根本找不到程序错误。这样做的价值何在？

不要通过自动化使无序情况更严重

如果测试设计得很差，则自动化会使差的测试执行得更快。如果测试组织得

很差，则自动化计划会使情况更糟。如果测试员不知道或不想说自己在测试什么，则他们创建的是没有人知道如何运用的自动化测试。

以下是我们和其他顾问都曾经遇到过的场景：公司要求顾问进行软件测试自动化。最初的问题是能够采用GUI测试工具测试被测软件。这种方法需要资金和时间投人，但是最终可解决问题。当自动化测试计划没有多少进展时，顾问发现根本问题还是没有测试过程。于是他们把工具放起来，首先关注解决内部问题。只有在这个时候才会看到积极结果。

如果检查单组织得很差，计算机并不会替人组织。如果测试组织得很差，则自动化测试会把人的注意力从真正需要做的工作移开。

不要把手工测试与自动化测试等同起来

当手工执行测试时，可以充分利用人的能力。可以临时想出新测试，也可以注意到没有或不能预测的现象。自动化测试则是高智能过程微弱的、很小的反映。这也是为什么说把自动化测试看做自动化的人员测试是错误的原因。

自动化并不能使计算机完成测试员所做的事，自动化测试执行测试员明确描述的测试，不能利用测试员隐含的知识和认识。自动化测试每次运行都以同样的速度、同样的顺序、完全一样的鼠标移动和键盘操作做同样的事。但是手工测试员在每次运行测试时都必须对测试做变动。这些变动可以发现未看到的程序错误。

自动化测试结果检验也有局限性。以任何测试过程为例，这种检验能告诉测试员来自扬声器的奇怪噪声吗？能告诉测试员检查屏幕上的异常雪花吗？能告诉测试员渐进性性能退化吗？也许不能。但是即使没有得到指示，优秀测试员也会注意到并报告这些问题。经过专业培训的人的头脑是最好的测试工具，要超过任何可能想像的自动化测试。在测试员说“怎么了？这不可能正确”时，可以临时注意到成百上千以前甚至想都没有想到的问题。

自动化测试有很多优点，但是测试员和测试是有很大区别的。这意味着问题可能被触发但是没有被注意。与自动化测试不同，手工测试员可以立即联系当时的背景条件，调查研究所出现的异常。测试员还可以识别并过滤很大范围内的虚警，而这些虚警会对自动化测试带来麻烦。

因此，不要拿手工测试与自动化测试相比，而应该把自动化测试看做是对测试员能力的扩充，能够完成手工测试所不能完成的工作。

不要根据测试运行的频率来估计测试的价值

测试的价值在于它所提供的信息。估计测试的价值是很困难的。好的测试就是要很好地进行这种判断。

常常有人建议测试员估计自动化测试与手工运行同样测试的成本相比，投资回报有多少（例如Linz和Daigl 1998a和1998b；Dustin等 1999，第52页；Fewster和Graham 1999，第519页）。

我们认为，这种方法以错误的方式度量了错误的东西（Hoffman 1999a）。

以下是我们认为从根本上有错的等式：

手工测试成本 = 手工测试准备成本 +（N × 手工测试执行成本）

自动化测试成本 = 自动化测试准备成本 +（N × 自动化测试执行成本）

……其中，N是测试执行的次数

作为一种成本的简单近似，这些公式比较公正，反映出自动化测试一般具有较高前期成本，而执行成本较低的基本共识。

我们基于以下两点反对使用这些公式判断自动化测试的成果：

1. 测试本身是不可比较的。不要把手工测试等同于自动化测试（请参阅经验108）。这两种测试不能提供相同类型的信息。

2. 比较自动化测试的运行成本，比方说50次，与手工50次运行同样测试的成本是没有意义的。谁会手工运行50次同样的测试？这样做是不值得的。测试的价值，即所提供的信息，不会抵偿执行开销的！

是的，自动化使得可以更容易地经常运行测试，但是只有当本来每次都要手工执行这些测试时才会节省资金。

与所有测试活动一样，决策的背后自动化某些测试是成本和效益分析。如果分析有错，就会不恰当地分配资源。

自动化的回归测试发现少部分程序错误

非正式调查显示，由自动化测试发现的程序错误百分比令人惊讶地低。设计良好的重要自动化测试项目报告说，回归测试发现的程序错误占总报告程序错误的15%（Marick，在线）。

自动回归测试在测试开发阶段发现的程序错误比以后执行测试时多。但是，如果执行回归测试，并在不同环境中发现重用这些回归测试的机会（例如不同

硬件平台或不同软件驱动器），则测试更有可能发现问题。结果，这些测试实际上不再是回归测试，因为测试的是以前没有测试过的配置。测试员报告说，这种自动化测试一般可以发现接近30%~80%的程序错误。

老功能的新测试也与老测试一样可能发现回归程序错误，并且增加了发现以前没有发现过的程序错误的优势[⊖]。

在自动化测试时考虑什么样的程序错误没有发现

在计算测试自动化的成本时，建议关注机会成本。花在自动化测试的时间可以用来做什么呢？现在没有运行什么测试？现在没有发现什么程序错误（Marick 1998）？

很多自动化计划的直接影响是拖延测试和拖延程序错误发现。这也是使测试员能够在软件测试就绪之前准备测试的领先自动化测试技术要解决的问题。例如，数据驱动策略使测试员能够很早定义测试输入。在这种情况下，测试自动化可帮助测试员加快测试和程序错误发现。

差的自动化测试的问题是没有人注意

已经开发出一段时间的产品常常有自动化测试包。这些自动化测试包是以前创建的，现在仍然用来测试该产品。

测试员常常庆幸有这样的测试包。但是我们想提出一些警告：

测试可能并不完成所想像的工作。测试员怎么知道这些测试会完成期望完成的工作？测试自动化开发人员有时编写与描述不同的测试过程。很久以前，这样做也许有很好的理由，要么就是自己搞乱了。现在的测试员怎么会知道？重新检查这些测试并阅读代码有多容易？

测试可能已不再重要。评论员Douglas Hoffman报告说："我最近发现一个计算机制造商使用称作'孤独的位测试'的自动化内存测试。这个测试用于发现上个世纪70年代以前的核心内存的特定连线错误。最大的核心内存是16Kb，测

⊖ 有两种例外：（1）如果项目团队的配置管理很差，老代码可能会被重新引入，并带来老程序错误。但是，良好的配置管理通常是比回归测试更便宜、更有效的解决方案。（2）设计很差的代码会导致对一个问题的解决引起另外的问题。当以后发现并解决第二个问题时，第一个问题又重新出现。

试运行需要几分钟。现在对于很大的内存配置要运行几个小时。测试所寻找的错误在当前内存系统中已不可能。我最近评审的另一个自动化测试是专门设计用来测试嵌入式处理器系统状态机的。我看到该测试的文档还是1986年编写的，是七代以前的产品，而且当每一代产品的状态机都被更改时，没有人想到要更新该测试。”

覆盖率可能很差。人们常常关注测试包中的测试个数。增加大量相同测试的小变种，可以很容易地使测试包的尺寸膨胀。检查测试包的实际覆盖率，可能会对有这么多的语句没有被测试而感到惊讶。

虚警可能很常见。随着时间的推移，不够仔细的自动化测试编程和被推迟的维护，会由于自动化测试编程中的程序错误导致测试失败。当然，测试员不能肯定哪些失效是产品程序错误，哪些是测试失效，因此需要研究所有失效。这些测试中的很多是有问题的，不能正常执行已经有很长时间了，但是没有人修复。从测试包中删除有问题的测试。这样的测试有什么用？这些是死测试。承认这一点并将其埋葬，要么现在就修复这些测试。但是不要把这些不好的内容留在有用的测试包中。

测试结果可能有错。有些测试不检查任何结果。除非产品崩溃，否则测试都会通过。有些测试包含程序错误，可跳过部分测试，或不能报告所检测出的失效。其他测试可能有无效的“黄金”输出文件：当测试失败时，黄金输出文件被更新，而不是报告程序错误。我们没有夸大。根据我们的经验，这种问题一般出在相对复杂的测试包中。为了避免出现这种问题，自动化测试时必须简化问题，或使用保守的程序设计手段。这类问题相当糟糕，但是我们还看到和听到过有很多测试把通过条件固定在代码中。这样的测试只给出“结果 = 通过”，这是自动化测试开发人员甚至没有想过的。我们仔细想过出现这种情况的原因，但是没有找到任何有说服力的解释[⊖]。我们所能够说的，就是这种情况会在很多不同地方出现。我们不断听到有人又看到同样的情况发生。甚至在被测系统上还没有要测试的产品，测试怎么会通过呢？

好的测试包是活的。要补充新测试，要修复或删除老测试。如果没有出现这种情况，测试包就会开始僵化。不久，开发人员就会转到其他工作上，测试包就会开始进入神话般的老橡树状态，即动画片中的那种给出建议的角色。随着橡树越来越老，就会赢得更好的名声。聪明的老橡树一定是正确的，因为它已经存在这么长的时间了，这种想法对于动画片角色很合适，但是我们认为测试

⊖ 出现这种情况的很多Sweency报告案例，都是因为自动化测试开发人员开始使用了包含“结果 = 通过”模板，然后忘记将其删除。

员应该更警觉。我们把这种盲目相信创建了出色（现在已经老了）测试包的老测试员的智慧和编码的现象，叫做老橡树综合症。

捕获回放失败

最流行的测试工具包括各种记录器。这些记录器看起来提供一种不用编程创建自动化测试的方法。在测试时只需打开记录器，记录器会捕获所有用户事件，包括所有击键、鼠标移动和鼠标点击事件，并将这些事件存储在可供以后回放的脚本中。测试员还必须指示检查点。这些就是点击热键以通过脚本保存屏幕信息的测试步骤。在回放期间，脚本将当前屏幕（或与当前屏幕有关的某种设计好的属性）与所保存的信息比较。如果两者不同，则测试失败。

关键问题是，这种脚本与用户界面和系统配置的微小细节捆绑得太紧。在实践中，测试常常因为有意实现的产品设计变更，或由于无意产品回归程序错误而失败。测试员发现，他们花在分析测试失效和重新记录测试上的时间，要比以前手工执行测试所需的时间还多。

假设有100个涉及创建账户的测试脚本，并且产品被修改，现在账户创建工作流程要通过一个额外的对话框。所有这些测试都会失败，必须重新记录测试。再假设屏幕上的一个标签从“登录名”改为“用户名”。包含这个屏幕检查点的所有测试现在都会失败。

如果测试失败，会给分析带来更大的困难。是设置程序错误，自动化测试中的程序错误，还是产品程序错误呢？从计算机动态生成的脚本很难区分这些程序错误。如果不读脚本，怎么知道测试怎样运行呢？

更结构化的测试自动化策略使用同样的工具，但需要更长时间编码并要求程序设计技能。当用户界面变更后，更新所产生的测试要容易得多。

根据我们和大多数测试自动化设计人员的经验，捕获回放自动化测试是覆盖着冰的下面有砖墙的坡地。有太多的小组花费大量时间记录自动化测试脚本，发现程序只修改几次这些测试就不能再用了，需要全部重新记录。

要在构建能够持久或与手工测试结合的自动化测试的技能和规划上投入。与捕获回放相比，这两种测试通常更高效、更有效。

（我们发现捕获回放工具对于学习工具很有用，并且有助于手工编辑测试脚本。我们反对将捕获回放单独作为解决方案。）

测试工具也有程序错误

当测试员发现花了很高的价钱采购的质量工具本身有很多程序错误时，会很愤怒。的确，与可比较（但是便宜一些）的开发工具相比，测试工具常常程序错误更多。要计划测试工具，并花时间找出解决程序错误的方法。

有些工具专门支持特定的组件技术。这些组件技术中缺少可测试性，是导致测试工具存在程序错误的一个原因。

- 工具程序员必须首先等待能够获得这种技术。
- 然后必须对组件实施逆向工程，确定如何增加测试支持。
- 在此期间，产品程序员又开始使用最新的组件技术，而工具提供商还不能提供所需的工具更新。当测试员需要时，提供商会让工具匆忙通过测试，以尽快交付给测试员。
- 奇怪！匆忙工作产生大量程序错误。

只要必须对组件技术实施逆向工程，而不是从头开始设计，这种循环就会永远进行下去。我们满怀希望并听到过改进承诺，但是很难说什么时候会改进。这是一个全行业范围的问题。

有时测试工具会受其他组件中的程序错误的影响。我们中的Pettichord使用过一种执行不可靠的执行工具，有时不能生成鼠标事件。最终发现是操作系统驱动器的问题。没有工具这种不可靠是可重现的。为什么手工测试员没有发现呢？因为他们没有注意到偶然的鼠标移动或点击没有起作用。如果他们注意到，也可能会怪罪自己或鼠标硬件。我们向操作系统提供商报告了这个缺陷，当时他们看起来还不知道这个缺陷。对可测试性的严重影响看起来不重要。我们最终使用不同的鼠标驱动器解决了这个问题[⊖]。

有些工具可能会对正在测试的产品产生很大的影响，以至于不能使用这样的工具。覆盖率和内存监视器必须对软件插装，要占用相当多的内存。这样的工具会使产品执行速度降低过大，以至于不能执行测试。

由于测试工具有这样的坏名声，因此程序员可能会在认真考虑自动化测试发现的程序错误之前，要求测试员在工具之外重现自动化测试所发现的所有程序错误。这会进一步增加分析自动化测试所捕获失效的难度。

⊖ Bob Johnson报告了使用不同技术的类似问题："鼠标驱动器实际上没有参与所要求的应用程序测试。但是为了自动化测试，我们不得不跟踪这个问题，花了我们测试员和开发人员几天的时间。为了自动化测试，付出了几千美元的隐性成本。"

用户界面变更

保持与用户界面设计变更同步，是GUI自动化测试的一个主要困难。如果要自动化GUI测试，要把这一点考虑在内。

告诉程序员他们应该冻结用户界面是没有用的。用户界面的早期版本经常需要修改。不要把自己置于反对改进的位置上。GUI总是要修改的。

要抽象测试自动化设计的界面。当用户界面变更时，只需升级抽象层，而不是升级访问修改后界面的所有测试。

以下是提供产品GUI抽象的一些手段：

窗口映射。GUI测试工具支持各种手段来标识窗口控件，例如内部名称、各种属性、邻近标签和顺序位置。不是将标识手段嵌入到对控件的所有引用中，而是使用窗口映射将名称与该控件的标识手段关联。如果用户界面变更迫使变更具体手段，则只需更新窗口映射。有些测试工具包含对创建和使用窗口映射的支持，又叫做GUI映射（GUI map）或窗口声明（window declaration）。如果工具没有提供内置支持，通常可以不太麻烦地创建窗口映射支持。窗口映射提供小GUI变更支持，例如标签重命名，或在屏幕上重新定位控件。窗口映射还可以用来支持对已经移植到其他语言的用户界面的测试重用。

数据驱动的自动化测试。（经验127：数据驱动的自动化测试更便于运行大量测试变种。）这种手段提供某种抽象，使测试员能够变更测试过程，并仍然能够使用所创建的测试数据。

任务库。（经验126：不要只是为了避免重复编码而构建代码库。）将测试用例分解为要素任务。每个任务在概念上都应该是不同的。要特别注意任务的起始和结束状态。为这些任务创建可以在测试脚本中使用的函数。如果任务的用户界面出现变化，则只需更新任务，而不是使用任务的测试。这种手段提供了显著的抽象，但是开始时会要进行大量的设计和创建工作。

关键词驱动的自动化测试。（经验128：关键词驱动的自动化测试更便于非程序员创建测试。）

基于API的自动化测试。（经验132：利用编程接口自动化测试。）完全避免GUI。

根据兼容性、熟悉程度和服务选择GUI测试工具

经常有人要我们推荐测试工具。最佳选择取决于具体环境。有些GUI测试工

具与特定开发环境不兼容，或支持得很差。一些微妙的因素也会引起麻烦。很难事先预测什么工具可用，什么工具不可用。这实际上就是试错法。那么从哪里开始呢？

确定团队已经知道的工具，或是否已经知道某种工具所使用的语言。工具使用的培训和学习费用（是正式课程、自学还是试错）常常相当高。熟悉程度可能是工具成功的决定因素，当然假设该工具被实际充分使用。

另一个重要的因素是提供商支持工具的能力。即使工具针对今天的被测产品运行得很好，来年必须在新的平台上测试时会出现什么情况？在购买工具时，投资的是提供商维护工具并使其与新技术同步的能力。可检查提供商的在线支持论坛，与其他用户讨论提供商的历史情况。

需要花一些时间测试工具与产品的兼容性，并检查提供商的服务记录。要有一个试用期（30~90天），或至少30天的定金返还保证。在这期间可能会遇到问题。测试提供商的支持以检查能够得到及时、有效的响应。还应该要求购买工具时捆绑提供的培训，这样会更合算。往往只有在培训时才会得到没有写在手册中的重要提示。

对于工具中的有用功能我们确实有很强的看法，但是对于大多数开发团队来说，上面介绍的问题会把这些看法分解为一种选择。有关进一步建议，请参阅Hendrickson（1999）。

经验 117

自动回归测试消亡

自动回归测试所面临的最大问题是退化和过早消亡。测试员设计回归测试以检测已经通过测试的功能中的问题，即由于修改错误或增加新功能时的程序员失误所产生的问题。

回归测试退化有多个原因，例如：

用户界面或输出格式的变更。这类变更是导致退化的主要原因。由于这些变更，已经通过的很大一部分测试都会失败，即使并没有做什么会影响要测试的功能的明显修改。这些变更可能非常微小，以至于手工回归测试员根本不会注意。但是自动化测试是敏感和脆弱的，不能将改进和程序错误区分开。

有关测试环境的设计假设。当把测试包用于不同的计算机或重新分配必要的资源时，测试包会出现问题。

维护中的错误。修改测试的自动化测试开发人员犯错误，将问题引入测试包中。回归测试包因此会自己产生回归程序错误。

变更操作员。测试包的使用和维护可能要求特殊技能和知识。当重新指派自动化测试操作员时，有价值的知识会流失。例如，由于产品中的问题，Sheila临时关闭磁盘I/O测试，并把编程工作留给Topeka完成。Sheila的接替者可能假设该测试已经完全作废，并将其归档，永远不再看了。

测试员在回归自动化上投入，是期望这种投入会随着时间的推移得到回报。遗憾的是，他们常常发现这些测试不能使用的时间要比期望的早。测试跟不上产品的节奏，需要修改，对发现程序错误不再有帮助。通过更新测试测试员可以做出合理的响应，但是这会比预想的困难。很多自动化测试开发人员都发现，在诊断测试失效和修复退化测试方面所花的时间太多。

在选择要自动化的测试时需要小心，因为所自动化的任何内容都要维护或放弃。失控的维护成本也许是自动回归测试要解决的最常见的问题。

仅仅是维护成本这一点，就足以使自动化会使测试员没有事可做的神话破灭。

测试自动化是一种软件开发过程

测试自动化项目常常由于缺乏约束和项目管理而失败。很多测试员没有意识到自动化测试实际上就是在开发软件。

Weinberg（1992）提出了一种评估机构开发级别的方法。与类似的分级一样，第一级是无序机构，第二级是重复级，一直到最高的第五级。但是他在最下层还加了一级：第零级是忘却（oblivious）机构，即甚至没有意识到自己在开发软件的机构。很多测试小组都没有意识到测试自动化是软件开发，因此属于这一级。

任何成功的软件开发项目都要求遵循某些基本规则。当程序员没有遵循规则时，第一个提出意见的就是测试员。如果走了捷径后测试自动化计划遇到麻烦，测试员不应该感到意外。

规则是什么？规划项目，并建立里程碑和可交付制品，定义需求，对工具、自动化代码和测试进行源代码控制。在编写测试代码前先设计，并对代码进行评审和测试。将测试自动化程序错误存入程序错误数据库中。将自动化测试的使用写成文档，并准备提供给非测试自动化开发人员使用。

没有把测试自动化作为软件工程会导致代价昂贵的自动化失败，例如测试小组发现由于没有有用的自动化的测试使进度受到很大影响。

我们不想说要遵循哪种开发过程，但是要遵循某种过程。

测试自动化是一种重要投资

自动化测试需要时间和成本。设计良好的使用GUI测试工具的自动化测试，所需的时间是手工执行GUI测试工具的10倍。虽然这个数字会受到很多因素的影响，但是很多有经验的测试自动化开发人员都用10倍作为初步估值（Kaner 1998a）。我们也看到有人提出只需2或3倍的时间就可以自动化测试。我们发现得出这种结果是因为所自动化的是一次性测试、没有全部考虑所有自动化工作，或者是运气好。如果比较谨慎，在掌握自己具体环境的具体证据之前，最好还是不要使用较乐观的估值。

有效地使用测试工具，要求熟练的编程和设计技能。与自动化任何其他工作一样，测试自动化也是软件工程问题。

如果有很好的理由重用很多次相同的测试，那么测试自动化会有助于降低成本，有助于创建和运行手工运行昂贵或不可能的测试，有助于不能手工完成的程序操作（时序和内存使用）度量，有助于运行大量测试序列，查找只有经过几个月的正常使用才会表现出来的缺陷，以及手工运行太费时间的测试。通过在测试自动化上的一定投入得到这些好处中的一部分是值得的，但是如果资源和预算计划是建立在几乎不花钱就可以用部分时间完成的假设基础上，那么希望就会落空。

在编写本书时，很多GUI测试工具的价格是每个席位5 000美元，负载测试工具的价格是50 000美元甚至更高。测试工具的价格已经很高了，但是我们发现培训（不管是正式培训还是在岗培训）和保障方面的人员成本会很快使工具成本相形见绌。不要把测试自动化的所有预算都投到测试工具上，那只是冰山一角。

测试自动化项目需要程序设计、测试和项目管理方面的技能

同时擅长测试设计和测试自动化的人并不常见，指派不同的人完成测试设计和测试自动化是很重要的。这两种工作都是全职的。使用专家可以得到最佳结果。

测试。明确自动化的测试目标。测试的用途是什么？怎样帮助发现程序错误？发现什么程序错误？测试需要理解产品的用户领域吗？对测试不了解但非常投入的程序员，很容易创建有意思但没有价值的测试包。由理解测试以及产

品将会怎样使用的人提供指导。

编程。测试自动化就是编程。使测试员不编程就能创建测试包的工具策略是行不通的。不要单纯依靠初级程序员或不合格的程序员。管理、安装、配置和维护工具都需要编程技能。所有自动化测试都是更大自动化测试应用的一个程序或功能。测试自动化并不容易，不遵循软件工程原则是不会成功的。

项目管理。如果不重视管理，自动化项目就可能不会实际达到最初预想的目标。不要让自动化成为副产品，不要让兼职人员承担。

技能全面对于创建长时间有用的测试项目特别重要。需要让员工接受语言和所使用工具的培训。所采用的自动化方法将决定具体需要哪些技能。

通过试点验证可行性

通过测试自动化项目试点验证所使用的方法，确认所选用的工具适合被测产品，并确定通过自动化预期可以得到的回报。计划用一个月左右的时间显示结果，然后全面推广。

试点有助于展示测试小组的能力；使保证得到全面实施测试自动化的成功所需的资源和协作更容易一些。

自动化的测试会改变开发和测试过程。越早自动化部分测试，越可以更早进行变更。测试小组使用手工测试过程的时间越长，越难最大限度地利用自动化测试包。

请测试员和程序员参与测试自动化项目

如果有产品测试员和产品程序员参加，测试自动化项目会受益。

产品测试员。请产品测试员定义测试自动化的需求，并请他们检验所开发的自动化测试是否有用、可理解和可信赖。自动化测试光好还不够，还必须得到测试员的信任。如果测试员不相信自动化测试，就不会使用。自动化测试必须服务于测试员。

产品程序员。产品程序员是软件开发专家，应该评审自动化测试的体系结构。请他们参与开发团队的工作，并提供加入产品可测试机制的更多机会。

考虑以下这些常常被忽视的关键域，确立清晰的目标并定义需求（Pettichord 1999）：

可评审性。谁需要能够评审测试？做到这一点难度有多大？

可维护性。谁将维护测试？他们必须了解什么？

完整性。怎么知道测试被别人信任？

经验123

设计自动化测试以推动评审

要测试软件是因为有编码错误。编码错误在测试代码中也存在。怎样处理测试代码中的错误呢？

应该测试自己的测试代码。想法不错。这种测试也应该自动化吗？应该测试自动化针对自动化测试的代码吗？想想一层套一层的测试就叫人头疼。

另一种方法是评审测试代码。设计测试框架，以便于不同人员评审测试。这有助于他们信任自动化测试开发小组所编写的测试，信任对自动化测试的测试。要使代码简单，并优化测试数据格式，以满足测试员的需要。使用标准的程序设计语言。

测试自动化会在很多方面出现问题。通过评审可使出现重大失误的可能性大大降低。还要帮助测试小组成员相互学习，在他们学习评审测试代码时，也就学习了如何开始评审产品代码。

鼓励评审的企业文化会促进每个人合理依赖的测试自动化。

我们没有多少自动化测试的成对编程经验。这是另一种形式的评审，预计（但不能根据我们的经验确认）这种方法也会很有效。

经验123

在自动化测试设计上不要吝啬

假设手工测试会由高素质和负责任的人完成是很合理的，但是对于自动化测试不能做这种假设。

以下是在设计自动化测试时应该明确考虑的问题：

- 保证测试已经被正确地设置。
- 描述预期结果。
- 发现潜在错误和副作用。
- 从潜在测试失效中恢复。
- 防止测试相互干扰。

将测试设计形成文档，以便以后使用该测试的人能够从现在的设计思想得到

一些启发。

避免在测试脚本中使用复杂逻辑

测试脚本中的条件逻辑会使测试更难理解，也更容易出错。更成问题的是包含发出和获取错误信号的代码。

可能需要逻辑控制来处理测试的设置、响应经过检查的输出，或处理定制控件。把这些逻辑放在单独的功能中。可以单独测试该功能（这样做很好），也使测试更容易评审（这样做也很好）。

使测试线性化有助于关注测试的目的（另一种好事），而不是自动支持。如果测试太复杂，就容易引入错误。要使测试简单，使测试线性化。

不要只是为了避免重复编码而构建代码库

标准程序设计格言建议，通过把重复代码放入函数中，在以前包含这些重复代码的地方调用，从而避免重复代码。在测试自动化中，这种方法经常带来麻烦。而相反的方法，即保留重复代码不动，叫做*开放编码*（open coding）。

测试的特性是大量重复。要通过不同场景、顺序或通过与其他功能的各种组合，来测试相同的功能。如果把所看到的重复代码都另行存放，最终会得到一种杂烩库。函数包含的常常是一个接一个的任务序列，即使任务是另一个任务的一部分。命名规则、结果分析和测试策略也可能会被绑在杂烩库上。很难为函数指定有意义的名字。在实际环境中，很难推断函数是做什么的。

使用这种库的测试很难评审，很难调试，很难修改。我们多次评审过用杂烩库构建的测试包，都没有得到比较理想的结果。

测试自动化有很多重复代码。有用的库应该遵循更强的设计原则，而不只是为了避免重复编码。有用的任务库关注的是封装用户可感知到的任务，特别注意函数的起始和结束状态。这种工作并不总能达到目的。如果出现这种情况，可采用开放编码方法。

数据驱动的自动化测试更便于运行大量测试变种

为了通过相同的测试过程测试不同的输入和输入组合，可利用数据驱动的自

动化测试。

将测试输入和预期输出组织为表，表中的一行对应一个测试。然后创建一个从表中逐行读入的自动化测试过程，执行每个输入步骤，并检验预期结果。电子表格对于存储测试数据是很方便的，可使数据录入更容易。大多数测试工具和程序设计环境都可以没有多少麻烦地访问电子表格数据，可以访问以电子表格内部格式存放的数据，也可以访问很容易输出的以隔断文本文件格式存放的数据（.CSV文件）。

当把数据驱动测试过程放在一起后，可以反复使用该过程来执行新测试。这种手段对于有很多不同数据选项的产品工作流来说最有效。利用更复杂的变种关键词驱动的自动化测试（keyword-driven automation），可支持由不同序列或多个不同路径组成的测试。

数据驱动的自动化测试支持非程序设计测试员。自动化测试开发人员创建数据驱动的测试过程，测试员创建测试数据。在有些情况下，可能很难自动化测试结果的检验。采用测试过程来收集测试结果，并在输入数据的语境中表示测试结果，这样可以简化手工结果分析。

数据驱动的自动化测试越来越常见，很多测试工具都包括对这种手段的直接支持（Dwyer和Freeburn 1999）。

经验128

关键词驱动的自动化测试更便于非程序员创建测试

关键词驱动的自动化测试建立在数据驱动手段上。但是表中包含指令（关键词），而不只是数据。

首先，这种方法要求既支持运行测试，又支持设置库、结果分析和报告的一般框架。这种框架要用于所有关键词驱动的测试。

第二，必须创建一个任务库，封装由被测产品支持的用户任务。标识可以在测试中使用的所有任务函数，对每个任务函数在任务库中都有一个记录与之对应。声明任务函数有效的起始状态，以及所产生的结束状态。这样可以分辨出哪个任务函数序列是有效的，以便捕获有问题的测试。

第三，增加对读取电子表格数据的支持，每次读入一行。通过声明把第一列解释为任务函数名称，后续列是函数的参数。使用函数的参数执行该函数。然后指向下一行。

其结果就是关键词驱动的自动化测试。利用这种方法可以避免在测试脚本中使用复杂逻辑（经验125）。由于测试存储在电子表格中，因此通常便于非程序

员创建和评审。由于要使用和测试的任务是测试员要描述的所有内容，因此测试员可以将注意力集中到测试，而不是控制语言上。

对于要求很多非程序员创建自动化测试的情况，我们认为这是一种比较好的解决方案。这种方法的缺点之一是，除非所要求任务的支持关键词已经存在，否则不能编写测试。定义和实现任务函数会变成主要任务。

评论员Hans Buwalda写道："与我自己的常用词一样，关键词驱动方法可以提供一种很好的基础。但是多年的经验说明，测试和测试自动化仍然是极具挑战性的领域，需要有经验专家的共同努力。"

我们看到过使用这种方法的项目取得很好效果，可以提前相当长的时间开发自动化测试。我们也听说有些项目团队认为这种方法要求做的工作太多，难以承受。

有关这种手段的进一步讨论，请参阅Pettichord（1996）和Buwalda及Kasdorp（1999）。

利用自动化手段生成测试输入

常用程序设计方法在以下情况下会有所帮助：

创建大文件。

创建大量测试输入。

设置测试床。在装载测试时，首先预装载数据量比较实际的数据库。要搜索的数据量会对数据库检索产生影响。

创建随机数据。特别适用于数据驱动和关键词驱动的测试。

覆盖所有输入组合。利用算法生成排列和组合。

但是，使用以上方法通常不能描述预期结果。这会要求进行大量手工评审或描述结果的工作。以下手段特别有用，因为这些手段要么可以在一定覆盖率的前提下减少所需的测试用例，要么可以描述预期结果。

覆盖等价类的所有代表对偶。有些研究提出，如果测试所有关键等价类成员的成对组合，可发现大多数交互程序错误。本书第3章中的"如何使用全对偶测试手段进行组合测试"一节讨论过全对偶组合方法。

覆盖逻辑条件中的交互。如果变量不独立，就不能使用类似全对偶组合测试的手段。因果图（cause-effect graphing）是一种更健壮的方法（Elmendorf 1973和Bender 1991）。我们还没有使用过这种手段，而听到过成功和失败的例子。

通过状态模型创建测试场景。状态模型测试手段经过学术界的深入研究，

在工业界也取得重要成果。状态模型，也就是状态图，标识系统的状态（文档变更或未变更，数据库连接或切断连接，事务挂起或完成）以及状态之间可能的迁移。熟练专业人员不必创建太多的状态就能够构建有意义的有用模型。有些人还对这种手段投入很大精力，生成大量状态模型，但没有得到收益。这种问题被称为状态爆炸。最成功的专业人员常常最快得到结果。如果读者要使用这种手段，我们建议创建一两个功能的状态模型，生成测试，然后对此进行重新评估。如果经过一周的努力没有得到回报，也许就不值得继续投入了（Robinson 1999和Nyman 2000）。

将测试生成与测试执行分开

将执行代码与测试数据分开的一种策略是数据驱动的自动化测试（经验127）。这种分离有利于测试生成，并具有以下优点：

- 测试易于理解和评审。
- 可以使用不同的测试工具或程序设计环境生成和执行测试。
- 独立的测试用例生成器比较容易测试。如果使用的是随机方法，应该知道程序设计环境提供的随机数算法往往很弱，数据的随机性可能不像所想像的那样强。参阅（Park和Miller 1988）。Kaner和Vokey（1984）提供了一种随机数生成器的经过全面测试的参数集，可以很容易地采用Java或任何其他能处理高精度整数计算的语言来实现。
- 如果预先生成数据，则更容易重复测试。我们见到过每次运行测试脚本都会改变测试的例子。如果不能预先生成数据，则需要采用其他度量以保证可重复性，例如记录数据或用于生成数据的种子。在测试脚本中加入指令以便使用这些记录数据，这样会增加复杂性。
- 报告所发现的程序错误更容易。程序员的第一本能就是对测试员的工具产生疑问。
- 不同的测试专业人员会各自关注自动化测试的不同方面，使用他们自认为最合适的任何工具或语言。

使用标准脚本语言

如果测试员希望了解更多程序设计方面的知识，我们建议学习Perl、Visual Basic、TCL、JavaScript、Python或周围程序员了解并使用的任何脚本语言

（Sweeney 2001）。有些脚本语言，例如Unix壳脚本或DOS批处理文件已经被使用很长时间了。脚本语言是便于使用而不是用来提高执行性能的高级语言。使用脚本语言而不是系统程序设计语言，例如C/C++或Java时，很多程序员的生产率都更高，更不容易出错。

对于大多数自动化测试来说，脚本语言都是最合适的。测试员可以使用脚本语言生成测试用例、访问编程接口以及检验结果。

很多测试工具都有内置脚本语言。有些工具明智地使用了标准脚本语言，有些工具创建了自己专用的脚本语言，我们叫做*提供商脚本*。我们认为使用提供商脚本并没有令人信服的理由，而且还有一些问题。

提供商脚本使编码更困难。这些脚本语言中的很多都以像C这样的标准语言为基础。如果可以看懂C，也许就能看懂基于C的提供商脚本，但编写有效代码需要很长的时间。这些提供商脚本不支持很多标准语言表达方式，这些表达方式使编写代码在一定程度上就像书写不用字母N的英文一样容易。

提供商脚本难以掌握。很难找到提供商脚本的培训课程或书籍，这增加了掌握提供商脚本的难度。即使学会了提供商脚本也不能用在其他地方。因此人们学习提供商脚本的积极性不高。如果要招募员工，也很难找到已经掌握提供商脚本的人。

提供商脚本影响测试员与程序员之间的协作。我们建议测试自动化项目要得到测试员和产品程序员的合作。如果使用不同的语言会使合作变得复杂。

很难在别人工作的基础上构建基本库。与可用于标准语言的库相比，可用于提供商脚本的库少得可怜。这意味着不能在别人工作的基础上构建，而是要花时间重新构建基本库。

我们建议避免使用采用提供商脚本的工具。现在有越来越多的工具使用标准语言。如果必须使用采用提供商脚本的工具，应尽量减少在工具中编写的代码量，并在独立语言环境中进行尽可能多的处理[⊖]。

经验132

利用编程接口自动化测试

今天，很多软件产品都提供可以用来实现测试的编程接口（公共API）。没有提供编程接口的软件产品可能有不公开的接口（私用API），如果提出要求设计者会提供。所以，如有需要应提出要求。

⊖ 有关进一步讨论，请参阅Pettichord（2001a）。

公共API作为软件产品的一部分要写入文档。公共API不会有很大变化，其稳定性对测试自动化很有吸引力。私用API会极为有用，但是要确定其稳定性如何。

由于没有注意到这些API，很多测试自动化设计人员都关注可见性最好的GUI。可悲的是，GUI通常是最困难的测试自动化接口。几乎所有编程接口都更容易使用，即更稳定、更可靠[⊖]。在前面介绍的经验中，已经讨论了有关GUI自动化问题，并提出了一些处理策略建议。但是最好的方法还是完全避免使用GUI。编程接口更可能提供稳定性，而且还有利于错误检测和隔离。

当我们检查所观察的很多产品测试工作时，发现是否提供编程接口与开发强有力的自动化测试包之间有很强的相关性[⊖]。编程接口包括API、命令行接口、COM接口、HTTP等，所以测试员需要学习不同的语言和技术。不要期望程序员会对测试工作指导，没有几个程序员是有耐心的，有时间的程序员就更少了。如果真想搞好测试自动化，就必须学习或找了解这些编程接口的人帮忙。

即使GUI测试自动化设计人员也发现需要学习GUI技术细节。在技术层上，GUI实际上比其他接口技术还复杂。设计人员总要通过某种方式学习自动化测试所使用的接口技术细节。设计人员要做出选择。我们认为GUI应该放在选择表的最后。

以下是一个简单例子。很多人要我们就使用GUI测试工具自动化安装器提出建议。最好的方法就是放弃这样的工具。很多安装器都有提供更好方法的脚本接口。例如，InstallShield是很流行的安装系统，用来为很多产品创建了安装器。很多测试员不知道可以运行InstallShield安装器，通过选项记录所选择的安装选项。这些数据记录在应答文件中。以后安装时，安装器可以使用应答文件自动确定安装选项。这很容易、很便宜，应答文件也很简单、易读、易编辑。这是自动化安装的一种效费比很高的方法（请参阅“创建无交互自动安装”，http://support.installshield.com/kb/display.asp?documents_id=101901）。

⊖ Paul Szymkowiak不同意这种观点，“我的经验与此不同。我发现很多用户界面比可用的编程接口更稳定——当然是从测试自动化的观点看。原因可能是用户界面对于客户是看得到的，并在培训材料中使用，印在用户手册中。因此，适应不断变更用户界面的成本更高。我曾经与一些项目经理交流过，他们会在等效的编程接口‘冻结’之前，首先要求很好地冻结用户界面。很多编程接口文档都编写得不好，因为程序员是预想客户，软件提供商/开发人员都不会在乎编程接口的不断改变。我还发现很多编程接口错误严重，与用户界面一般预期的‘可使用性门限’相比，编程接口的可使用性门限更低。”

⊖ Douglas Hoffman的经验与此类似，“我取得的最大成功是在一个著名的桌面出版软件包上。其引擎是不可见的，产品要通过其所见即所得GUI了解。由于计划要在n + 1版本上彻底检查用户界面，我们完全避免了自动化GUI测试，而是使用公共API实现功能测试自动化，并手工测试GUI。我们发现了大量缺陷（大多数都是在自动化测试时发现的），通过功能测试可以容易得多地确定GUI问题。”

但是现在测试的不是GUI！要把注意力集中到帮助最大的测试自动化上。有时可以有效地自动化GUI测试，有时则不能。不要让别人关于自动化测试应该是什么的先入之见束缚自己。

鼓励开发单元测试包

单元测试关注构成软件系统的最小单元：程序员所创建的函数、类和方法。大多数经理都要求自己的程序员做单元测试，而且程序员也声称做了单元测试。但是在实践中变数非常大，而且很难确认。

真正的单元测试要孤立地测试单元。要创建桩处理对外调用，创建驱动器提供对内调用。构建这些桩需要很大工作量。

自动化单元测试最常见的形式，通过在语境中测试单元来避免开发桩。我们也许可以把它叫做*单元集成测试*。对于自底向上构建的系统，这种形式的自动化会相当容易。

测试员需要一个像Junit或Xunit这样的框架来管理测试包的执行。这样做既不太困难也不太昂贵。代码通过该语言所支持的一般调用接口测试。程序员编写测试所使用的语言与产品软件语言一样。针对Java的测试用Java写，针对C语言的测试用C写。将单元测试用于回归测试、冒烟测试和配置测试。

应该在告诉程序员该做什么上十分小心。但是如果经理要求实现更多的测试自动化，则经理应该知道程序员和测试员可以有很多方法提供帮助。如果程序员显示出对单元测试感兴趣，我们建议测试自动化设计人员应该提供帮助。单元测试看起来是极端编程和其他敏捷方法的核心实践（Beck 1999和Beck等2001）。

小心使用不理解测试的测试自动化设计人员

自动化需要程序员。需要什么样的程序员呢？

很多程序员自认为对测试了解很多。除非程序员非常不负责任，否则会大量测试自己的代码，而且还常常发现不得不测试他们试图使用的其他代码。他们确实了解很多测试知识，只是与专职从事测试的人相比他们在这方面要差一些。

从测试员的观点看，很多程序员实际上并不了解多少软件测试（就像很多测

试员不了解程序设计一样）。程序员对测试的了解都以对自己代码的测试为基础，过于强调适合自己个性的策略。对测试要求了解得更深入的人应该提供输入和评审。

如果再加上很多程序员对测试员都报有的轻蔑态度，缺乏测试知识会成为更大问题。这种轻蔑源自测试员一般对软件不够了解的观点。因此，程序员会交付明显与所要求或所期望的不同的自动化测试。为了避免出现这种情况，应计划对测试自动化代码做评审。程序员习惯于别人给出他们并不完全理解的需求。他们经常使用的一种策略是做最好的猜测，编写程序，然后再根据反馈意见修改程序。对于常常造成需求和其他程序设计要求理解困难的模糊或混乱情况，采用这种策略是可以理解的。但当这种策略用于测试自动化时却是危险的。不要错误地认为直接运行测试包就会发现测试包是否有效。有些测试包程序错误会造成测试失败，这样的问题容易发现和解决。有些程序错误使测试放行了没有权力放行的问题，而测试员永远也不会知道。我们应运用评审、设计策略和测试来防止出现这种情况。

有些负责任的程序员可能意识到他们对测试的了解还不足以开发测试自动化项目，他们会把速度降下来，以便有机会学习。应该请他们开发试点项目，并请测试员评审。也可以请他们深入测试工作，以便获得测试员需求的第一手知识。

经验 135

避免使用不尊重测试的测试自动化设计人员

有些程序员把测试看做是低于其能力的下等工作。我们见到过这样的程序员被分配做测试。他们很自然地受测试自动化的吸引，并自信能够干好，因为他们认为自己是这样出色的程序员。

这样的程序员把测试看做是苦差，没有动力了解如何更好地测试。相反，他们寻求使工作更有意思，喜欢测试自动化工作，并着手开发没有多少价值的工具，或者重新设计程序错误跟踪系统，或创建运行测试用的新奇GUI，即任何能够避开测试的工作。

我们不懂这样做会有什么结果，不知道如何使用这样的程序员，建议如果可能还是避免使用他们。如果不能避免，要确保他们的工作不会对测试小组其他成员带来问题。也许可以让他们做技术支持。（不要忘了对这样的程序员说明自己认为跨部门工作经验对做好支持工作大有好处。）

可测试性往往是比测试自动化更好的投资

在很多情况下，可以利用可测试性（驻留在产品内部提供控制或可视性的测试代码）或测试自动化（测试支持代码在产品的外部）支持测试。可测试性常常以较低的代价提供更好的解决方案。以下是一些例子：

- 安装了产品之后，用户（和测试员）必须检查记录文件以查看是否有安装错误。如何自动化这种测试呢？第一种想法是编写一个脚本，浏览安装记录文件，查找可能的错误消息。而更好的思路是把这些作为产品的一个功能在产品内部实现。这样也许更可靠，能够得到更好的测试，实际上可直接使用户受益。
- 测试员需要对磁带备份软件模拟传输媒体错误，以测试该软件是否能够平稳地恢复。对于自动化来说很难模拟，可能需要创建一个磁带驱动模拟器。而与程序员协同工作的测试员可以通过下层磁带写入代码（虚假地）指示媒体有问题。
- 断言是代码中的语句，如果假设为假则指示出现错误。断言可以放入被测软件，以检查结果是否合理。与编写外部代码检验结果相比，这种方法往往更容易、更高效。

可测试性是可视性和控制

有助于提高测试员观察或控制软件操作能力的任何功能都是对可测试性的改进。有人常常要我们列出潜在的功能，以下就是这些功能：

访问源代码。很多公司不允许测试员修改源代码，但是可以查看源代码。能够检查源代码控制系统中的变更记录特别重要。

日志。记录错误消息、错误源、使用剖面、资源使用、系统消息和协议通信。可以定义记录的不同级别。在被测产品所使用的组件中，可能已经有了日志机制。程序员使用日志文件辅助调试，测试员使用日志文件更快地捕获程序错误，分析程序错误模式，在错误报告中提供详细信息，评估测试覆盖率，收集客户使用信息，并掌握更多被测软件的知识（DiMaggio 2000、Johnson 2001和Marick 2000）。

诊断。诊断可对潜在问题发出警告。断言就是一个例子。数据完整性检查检验数据内部是否一致。代码完整性检查检验程序是否被覆盖或修改。内存完整

性检查检验是否根据分配情况使用内存。与日志结合起来可以构成功能很强的工具，使测试员在检测到错误时能够进入调试模式，或卸出程序信息）（Kaner 2000b）[⊖]。

错误模拟。被测产品有内部状态，其中很多状态可能很难进入，特别是不可重现和系统问题。很多错误状态都是这种情况。软件应该能够从媒体错误、内存或外存不足、网络延迟和中断连接，以及类似的问题中恢复。测试员会发现很难创造这些条件，特别是不可重现和系统问题。可以在产品软件的下层替代错误状态触发器，以便于测试错误处理情况（也有在被测产品外部模拟这些错误的工具（Houlihan 2001））。

测试点。能够在系统的不同点上检查或修改数据（Cohen 2000）。

事件触发器。当内部任务开始和结束时发出通知，有助于帮助同步测试。

读入老的数据格式。在产品开发过程中，数据格式可能会发生多次变化。提供把数据转换为新格式的手段，以免重新生成数据。

测试接口。编程接口为可测试性带来重要好处。有些产品会为这个目的实际增加编程接口。

定制控件支持。使GUI测试工具能够使用定制用户界面控件，这也许是最需要的可测试性功能之一。

允许多实例。允许在同一台计算机上运行多个客户或代理，即使在现场并不支持这样的配置。这会使测试员能够在小实验室中模拟大网络。

很多软件产品都拥有没有写入文档的可测试性功能，程序员增加这些功能是为了辅助自己的测试和调试。可向程序员询问[⊜]。

经验 138

尽早启动测试自动化

测试自动化是一件艰苦工作，需要计划、研究和设计。如果计划实施大量自动化的测试，要在产品还在设计时就要开始。为了更有效地自动化，常常必须解决可测试性问题。由于很难事先描述，因此可测试性解决方案常常是通过试错法得到。

⊖ Noel Nyman写道："以下是Windows用来辅助发现错误的两个诊断'工具'。（1）在初始化缓冲区时填充已知模式。这有助于标识越界和指针错误等。（2）把缓冲区放在已分配堆的最后，并逆堆的方向处理。这样当缓冲区越界时，会强制触发Windows的内存错误通知。"

⊜ 有一个嵌入式软件产品，Kaner和Hoffman发现程序设计组已经编写了1100多个诊断命令，用来检查软件和设备状态。这些都可以提供给测试员使用，测试员只需把合适的命令加到其测试执行工具中即可。

程序员在项目的初期对可测试性建议更开放，在设计还没有定型时，增加可测试性功能更容易。程序员和项目经理可以将其列入进度和预算中，并进行计划。整个设计的不确定性会促进很多程序员对测试提供帮助，如果程序员知道代码会被很好地测试，其压力会减轻一些。

有些测试经理没有较早做好这些工作，而希望以后能够赶回来。我们发现，启动测试自动化越晚困难越大，原因如下：

- 当测试已经成型后，很难把资源转向自动化。
- 当测试人员和过程都集中于手工测试之后，他们会抵触变更。
- 设计完成之后，程序员在可测试性要求方面会变得不那么合作。

如果要成功地进行测试自动化，就不要拖延。但是我们强烈建议不要从一开始就尝试自动化所有东西。尽早建立基础设施，但是在选择要自动化哪些测试时应该慎重。

为集中式测试自动化小组明确章程

有些公司处理集中式测试自动化小组，以对多个产品的测试小组提供支持。这常常是一种明智之举。

如果有这样的小组，要确保有明确的章程来描述要提供的测试辅助种类、应该如何提出请求，以及如何平衡相互矛盾的要求。这一点很重要，因为遇到最大麻烦并要求提供最多帮助的测试小组，常常最不能通过测试自动化得到好处。他们更容易说自己需要自动化提高技术能力，而不是改进工作组织，就像是他们应该有能力自己做到这一点。不能很好地处理这个问题，就会卷入低效旋涡中。

我们建议要求接受帮助的测试小组指定专人参加测试自动化项目，这有以下几点好处：

- 可以评估他们所做出的承诺或所面临的实际问题。
- 通过培训其员工，可以减少后续维护和失效分析要求。
- 通过与请求提供服务的测试小组合作，可以从头至尾在项目中把他们的测试需求结合到工作中。

集中式测试自动化小组是一个很小的内部编程团队，直接与提出服务请求的人接触。请参阅Beck（1999）和Jeffries等（2000）。

测试自动化要立即见效

有太多的人认为测试自动化就是自动化手工测试，结果是过于强调GUI回归

测试。

不是考虑测试用例，而是应该选择容易取得的成果作为突破口会提高自动化效率。关注影响大、成本小的自动化任务。以下都是不错的切入点：

系统设置与准备。新测试员常常对花费那样长的时间安装和配置系统，使其达到测试就绪状态感到意外。这部分工作可以自动化。磁盘映像使系统能够复位到标准配置上。安装脚本可用来自动化设置过程。使用公共编程工具可以自动装载样本数据。

辅助诊断。有些程序错误很难表现出来。数据破坏和内存泄漏缺陷一般到该数据被访问或内存耗尽时才会被检测出来。诊断工具可以在缺陷出现时就检测出来。已经有检查内存的工具，构建检查被测产品数据完整性或内存使用的工具也不困难。与项目团队的程序员协作，他们也许已经使用这样的工具了。

会话记录。严谨的错误报告要求提供完整的配置信息。程序可以自动收集和报告必要的信息。这些信息有助于测试员确认系统是否是按要求配置的。

测试生成。利用自动化手段生成测试输入（经验129）。

不必从头至尾进行自动化。先实现一部分自动化，这有助于逐步实现更广泛的解决方案。

经验141

测试员拥有的测试工具会比所意识到的多

跑表是一个优秀测试工具的例子。度量系统的响应时间是一种重要的测试活动。跑表易于使用、灵活、准确。有些测试员和程序员可能认为，通过调用系统时钟实现的自动化方法应该更好，但是跑表常常是黑盒时间度量的最佳选择。我们建议测试员准备一只。

测试工具并没有贴上“测试工具”的标签。测试员会发现几十个其他工具都很有用。其中很多工具很便宜、很平凡。例如：

磁盘映像工具。使系统能够迅速恢复到某种已知状态。

依赖关系检查器。显示软件应用所使用的动态库。

文件扫描器。检查并分类系统中被修改了的文件。

内存监视器。跟踪内存使用情况。

宏工具。便于重复特定任务。

“小语言”，例如Sed、Awk、Grep和Diff。这些工具有助于自动编辑文件、处理输出、搜索数据和发现差别。这些小语言最初是为Unix开发的，目前在大多数平台上都可以得到。

很多系统实用程序和通用程序设计工具也适合测试，而且价格往往更合理。

在因特网上可以找到很多有用的免费软件、共享软件和廉价软件。www.zdnet.com、www.pcmagazine.com、www.cnet.com、www.qadownloads.com和www.softpanorama.org都提供了有用的程序。Elisabeth Hendrickson收集有用的工具集：www.bughunting.com（请参阅Hendrickson 2001b.）。

第6章

测试文档

本章的目的是帮助读者探索自己的测试文档需求。本章并不提供样本文档（有关表格和矩阵的例子，请参阅第3章的相关内容），而是提出有助于读者决定自己需要什么的问题。

本章首先详细评估IEEE标准829《软件测试文档》。我们觉得读者可能没有读过甚至没有听说过这个标准，因为IEEE销售该标准的定价很高，使大多数个人不能购买[⊖]。很少有人自己有一本标准829。我们的很多客户和雇主公司也没有购买IEEE标准。不过，在行业内流传的很多测试文档模板都是由标准829导出的。因此，即使读者不知道标准的名称，但只要在测试领域工作过一段时间，就可能会遇到这个标准829。

如果读者还没有接触过标准829，并且对测试文档模板不感兴趣，我们建议略去后面的内容，直接从“经验147：在决定要构建的产品之前先分析需求，这一点既适用于软件也同样适用于文档”开始阅读。

推动标准829的最新文件是我们要在这里讨论的《软件工程知识体系》。

本书前言已经提到过《软件工程知识体系》（SWEBOK 0.95版，2001），第10章还要进一步讨论。本章介绍SWEBOK的测试文档观点。

测试文档与工作产品

文档是形式化测试过程的一个重要组成部分。IEEE软件测试文档标准[829]提供了一种很好的测试文档描述，以及测试文档相互之间和

⊖ 2001年7月30日我们在www.ieee.org上核对了价格。印刷版标准829（64页简装）的每本售价为：公众65美元，IEEE会员52美元。公众电子访问售价为98美元，会员访问售价为78美元。对更多标准感兴趣的读者可订购4卷本《软件工程标准》1999版，订购价325美元（会员订购价260美元），也可以在线订购《软件工程标准》，每年850美元。

> 与测试过程之间关系的描述。测试文档包括“测试计划”、“测试设计规格说明”、“测试过程规格说明”、“测试用例规格说明”、“测试记录与测试差错或问题报告”等。描述了版本并标识了使测试可以开始的硬件和软件要求的被测软件，在文档中被称为“测试项”。应该编写测试文档并持续更新，与开发中的其他文档标准一样（《IEEE计算机协会》2001，第92页）。

我们曾经试图开发IEEE标准829风格的文档，也看到过不同行业中多家公司的努力结果。我们对结果并不乐观。事实上，根据我们的经验，标准829的缺点也许大于优点。

我们使用标准829遭到的部分挫折，是测试小组试图遵循标准829遇到麻烦的延伸。一个小组又一个小组根据标准829创建了测试计划模板，然后按照该模板创建没有价值的文档。最初我们认为问题出在人身上，一定是他们误用了标准。后来我们得出结论，问题一定出在更深层，因为问题的面很广，并且使我们尊敬的人落入陷阱。

我们看到的模式（或反模式）是测试小组创建或借用模板，从而将大量精力投入到填充格式的案头工作中，编写出并不特别有价值的大量原始材料。以后他们会遇到成本和这种文档效果的限制，并逐渐放弃。这使得很多测试小组做纯粹的临时测试，因为他们把所有做计划的时间和预算投入到不会使用的文档上。

放弃这种模式的努力并不意味着他们公开放弃他们的工作，通常是无声无息地停止阅读或更新这些文档。如果向他们提起测试文档，他们很可能会搬出大捆完整文档（没有人阅读或更新这些文档）。有些公司一遍又一遍地重复这个循环，总期望下一次会做得更好，抱怨自己这一次由于某种原因没有做好。

问题并不是这些公司没有以合适的方式遵循标准829。

问题是标准829并不是满足公司需求的合适方法。

为了有效地应用解决方案，需要清楚地理解问题

没有比使用和滥用测试文档更能表现出这个原则的了。因此，我们通过正反两个方面对测试文档标准进行辩论。我们希望读者首先考虑测试文档需要解决的问题，然后再运用一种适合解决方案的形式。

不要使用测试文档模板：除非可以脱离模板，否则模板就没有用

测试文档模板不能替代技能。

模板是创建测试文档的一种结构。填满所有内容或填空，就会得到自己的文档。

模板的问题是，模板使编写表面看起来不错但没有具体内容的文档更容易一些。对于一些人来说，这是额外的东西。但是为了使用模板编写好的测试文档，必须拥有模板，必须理解模板每一部分的含义，理解为什么要有这一部分，什么时候可以删除。如果测试员对这些都能够理解，就不再需要模板了。如果不理解这些，就不要受模板的影响。由于测试员不理解模板作者对需求和权衡所做的全面考虑，模板就会使测试员生产率降低。

我们看到过一些公司为自己创建了模板，但是没有什么使用效果，或生产率很低。使用模板的人必须能够根据自己的当前具体需求剪裁模板。

如果测试员不用模板就可以编写有效的测试文档，那么模板有助于这样的测试员更快地写出有效的测试文档。

使用测试文档模板：模板能够促进一致的沟通

也许测试员要把文档提交给第三方；也许作为定制工程产品的一部分，测试文档要交付给客户，客户根据这些文档接管维护和测试工作；也许另一家公司要用所编写的测试文档来测试某个产品；也许测试文档要供审计员或管理人员审查，或在使用公司产品出现意外事故后（包括人员伤亡），作为测试好坏的法庭证据。在这些情况下，测试文档要更有效地沟通，如果遵循标准，采用预先定义的格式，以标准的形式组织信息，涵盖一组标准化的问题，并使用标准术语，这样会让别人更容易理解。

使用IEEE标准829作为测试文档

IEEE标准829项目是由David Gelperin领导的。他很聪明，思想丰富、细心、开放，在促进软件测试领域内的多样性、创造性和技能的开发和运用方面，做了大量工作。他的公司叫“软件质量工程（SQE）”，负责组织STAR会议（软件测试分析与评审），这是软件测试领域最好、最成功的会议之一。SQE还组织了另外几个会议。我们在每次SQE会议上都发言。SQE还开设测试方面的课程，我们也为SQE组织的学习班讲过课。我们对标准829的评判，并不是对David的评判，David是我们的好朋友，我们也不是要批评SQE引以为豪的他们自己出资向

软件测试界提供的很多公共服务。

标准829的很多属性都反映出David的实力，标准中的各类信息都准确无误。当我们阅读标准时，可以理解为什么有人要知道这个标准所要求的所有信息。

标准中没有什么是强制的。如果测试员要创建过程脚本化的测试用例，标准829指出应该怎样说明这类测试用例，应该在文档的哪一部分描述。如果不想以这种方式编写测试用例，也不必仅仅为了满足标准还创建这种类型的文档。标准提供的是一种框架、一种结构和一组定义，而不是强制要求的一套内容。

这个标准已经经过广泛研究和讨论。软件测试领域的人都了解这个标准。这个标准已经成为在各个公司之间流传的很多（也许是大部分）测试文档模板的基础，成为很多公司内部和公司之间讨论测试计划和测试文档（或好或坏）的基础。

我们中的Bach和Kaner曾经担任过该标准对一些有缺陷产品的试用顾问（试用专家），我们的同事担任了另外一些产品的试用顾问，并向我们提供了详细情况。我们也遇到过一些测试文档成为导致法律诉讼问题一部分的情况。即使文档并不是关键问题，有些其产品被认定有缺陷，或被指控欺诈的公司，如果测试文档更清楚一些，组织得更好一些，在纠纷中的处境就会更有利一些。较差的测试文档影响了他们的辩护。IEEE标准829本来可以为这些公司节省大量金钱，而且本来也会为其客户节省大量金钱。

不要使用IEEE标准829

当我们最初阅读IEEE标准829时，每个人都曾经对其抱有很高热情。但是，我们发现实践中的一些问题。在我们评论这些问题之前，有时会被告知这些问题是由于滥用该标准造成的。毕竟这个标准很灵活。测试员不必以不适当的方式使用该标准。

对于我们来说，这种感觉就像是“枪并不会杀人，是人杀的人”这样的辩解。有时这是有效的辩解，而有时不是。

对于标准829，我们有很多次发现问题，问题出现在少数我们所尊重的人身上，出现在一些在多个项目中使用基于标准829模板的公司，因此我们认为这些问题反映出该标准的弱点，不应该都将其归结为人们使用上的问题。

如果枪有弹簧扳机，没有保险开关，在公共场合装上子弹，并加上有官方味道的告示“对所有项目都使用这把枪”，那么辩解“枪不会杀人”就会有另外的解释。

以下是我们在实践遇到的使用该标准的问题：

- 标准的前提假设看起来是瀑布方法，要在早期开发测试，仔细编写文档，以后就不再更改。变更成本（文档的维护成本）会对变更产生很大影响。在我们看来，随着程序越来越稳定，应该创建更复杂的测试，随着对程序认识的提高，应该创建能力更强的测试。由于测试文档的维护成本很高，促使测试员使用老的测试，而不是开发能力更强的新测试；坚持当前的测试策略，而不是随着知识的增多而加以改进，测试文档是问题的一部分，而不是过程的一部分。
- 大量的测试文档产生一种盲从心理。照着计划做。这与警觉心理有本质上的不同，挑剔的测试员会注意并利用可以利用的所有提示。
- 该标准没有提供测试文档需求分析，没有提供什么时候提供什么类型信息的建议或指南。
- 该标准既没有明显提到也没有讨论提供所有这些类型信息的（巨大）成本。花在文档上的很大一部分时间不能用在测试上。
- 该标准看起来强调文档的广度，好像越多越好，好像最好能够生成测试计划、测试设计规格说明、测试过程规格说明、测试用例规格说明等。
- 该文档没有提出判断测试文档特定实例好坏的准则。在实践中，文档量看起来替代了清晰性和覆盖率。我们曾经评审或审计了大量测试文档，作者考虑的是足够和完备，结果存在很大漏洞——完全忽略了关键测试策略和风险分析。当测试文档长达数百页甚至数千页时，即使是最明显的测试遗漏也太容易被忽略了。
- 这样大量的文档维护成本是很高的。当软件变更时，不仅要修改捆绑到软件被变更部分的那部分文档，而且还要搜索必须修改的所有内容，同时还要并行地搜索并修改实际测试文件（如果是自动化测试，就是代码文件）。如果文档和代码的对应关系不是1对1，就会出现匹配错误，这会对以后的测试本身带来问题，并增加时间开销。
- 将每个测试都写入文档会严重影响自动化的测试。如果每个测试用例需要一个小时编写文档（我们认为这是比较低的估计），并且要对项目做1万个自动化的测试（这肯定不是很多公司不同测试数的上限），那么在文档上就要花费1万个人时，还要支付准确按文档描述实现自动化测试的成本。当由于要维护被测软件，使测试发生变更时，测试文档也要变更，会花费更多的人时。这会为自动化测试带来很大负担。遵循该标准的公司会编写较少的测试，他们也许宁愿放弃一些测试，也不愿意面对维护测试和相关

文档的巨大开销。根据我们的经验，公司更有可能直接放弃文档，最终使到此为止的所有与文档相关的工作都变得没有价值（由于文档是不完整的，很快就会过时）。

- 要生成或通过随机数据或随机序列组合的大量测试（几百万）的自动化测试范例看起来完全脱离这个标准。我们可以想像把软件文档和相关的软件模型都硬塞到这个标准的大纲中，但是在实践中却行不通。相反，这些工作（模型、代码、预期结果等）都写入其他文档，或根本就不写入文档。
- 前面提到了我们的法律诉讼经历。这里列出的这些问题都是从不同的经历归纳出来的。一些公司在项目开始时作为合同条款，规定测试过程的文档工作要达到一定水平，要遵循的模板（基于标准829），要采取这样那样的测试策略。然后在项目推进过程中，这些条款会被放弃，以便在他们当时看来能够按实际进度要求完成实际的工作。他们的决定可能完全正确，但是在诉讼上会出现问题。他们开始有周密的计划和工业标准，然后无声无息地进行较少的实践，然后交付缺陷很多的产品。这看起来很糟、很糟、很糟。如果不打算遵循过于野心勃勃的计划，并且公司有被起诉的风险，就不要在项目开始时声称自己要遵循这样的计划。过于野心勃勃的计划的害处远远大于好处。

列出这些成本之后，我们再回到受益问题上。如果花了这么多钱，在自己的项目中增加了所有这些惰性，并鼓励员工在进行（而不是编写）测试时不要太动脑子，那么得到的回报是什么呢？

很多公司使用的纸张要少得多，他们使用简练的清单和表格、状态报告和定期团队会议跟踪项目状态。他们通过存储在运行良好的程序错误跟踪系统中的编写良好的错误报告跟踪项目中的问题。这样的公司通过遵循标准829会增加什么收益呢？在读者的实际环境中，这些收益是那么令人信服吗？

在很多情况下，所增加的收益并不令人信服。在这种情况下，由于开发和维护大量测试文档要增加大量成本和风险，因此我们认为创建标准829风格的文档是不负责任的。

在决定要构建的产品之前先分析需求，这一点既适用于软件也同样适用于文档

决定什么内容要包含到测试文档中，什么内容不包含，应该以项目需要为基础。IEEE标准829的格式和大纲可能很有用，也可能没用。在完成文档需求分析

之前就把IEEE标准829（或任何其他详细规格说明）作为自己的结构和用户界面，往低里说是不成熟的。

如果有人坚持在透彻地分析需求之前不应该编写代码，但是又要在没有进行相应的透彻需求分析之前就编写大量测试文档，则我们很难理解他们的心理。

请不要误解。我们并不是说IEEE标准829不适合读者的产品。IEEE标准829可能是完美的东西，就像COBOL可能是读者项目完美的语言一样。我们要说的是，在选定自己的程序设计语言和关键工具之前，应该考虑自己将要构建的是什么。

经验148

为了分析测试文档需求，可采用类似以下给出的问题清单进行调查

对于需求分析的全面介绍，以及可以用于任何需求分析的非常好的与语境无关的成套问题，请参阅Gause和Weinberg（1989）。Michalko（1991，第138页）提供了一组与语境无关的补充问题（美国联邦调查局的“不死鸟”问题）。此外，我们还收集了一些专门针对测试文档需求分析的问题。

测试小组的使命是什么，测试这个产品的目标是什么？如果测试文档不支持自己的使命也无助于达到自己的目标，这样的测试文档（与所创建的所有其他工作产品一样）是没有价值的。

自己的测试文档是产品还是工具？产品是给别人使用的东西。人们要为产品付费。生产者可能要按照客户的要求遵循任何标准，这是客户愿意为产品付费的条件。形成对比的是，如果文档只是内部工具，则不必太完整、有太多的组织要求、太整齐，能够在最低限度上有助于达到目标即可。

软件质量是受法律问题驱动还是受市场驱动？如果软件和测试要受到管理者的审查，那么也许要遵循像标准829这样的形式化文档格式。类似地，如果产品可能会引起人员伤害或财产损失，则测试文档可能会在诉讼中起作用。标准829形式化的结构，大量829风格的传统文档和“行业标准”状态，都使标准829成为一种好的选择。给人印象深刻的文档可能会也可能不会有助于提高软件质量，但是却有助于公司日后的辩护。另一方面，如果在市场中产品质量低的后果是降低了销售量，面不是法律诉讼，客户是不会看或关心测试文档的。

设计变更有多频繁？如果软件的设计变更很频繁，则不要把细节写入测试中，因为这些细节很快就会过时。不要编写大量测试文档，测试被修改或放弃的速度太快，不值得在文档上投入太多。

反映设计变更的规格说明变更有多频繁？如果规格说明长期不完整并且过

时，就不能采用规格说明驱动的测试，也不能把测试文档捆绑在这种规格说明的内容上。请注意：不要给别人带来过大影响。如果项目团队不按照最新的规格说明，那么项目可能会也可能不会需要更好的规格说明。给测试小组带来不便，并不是改变项目团队规格说明策略的特别有力的因素。如果不能得到好的规格说明，应计划修改测试策略，而不是修改项目团队政策。如果测试员要力争更好的规格说明，则应该以其他项目相关人员编写规格说明的成本和风险为基础，特别是对公司利益得失影响更明显的项目相关人员的成本和风险。

在测试时，是希望证明与规格说明一致，还是希望证明与客户预期不一致？如果要测试的是根据合同确定的规格说明市场的定制软件，则测试和文档可能关注与规格说明的一致性上。形成对比的是，如果产品是面向大众市场的，因此没有人会在规格说明书上签字，没有合同支持产品规格说明，利用手头现有的规格说明产生令客户满意产品的把握就较低。在这种情况下，更好的方法是通过测试证明客户不会喜欢该产品，而不是通过对照现有的任何规格说明检验产品来为项目作贡献。为达到这个目的的优秀测试文档，应该包含客户预期的证据，例如有关竞争对手产品的信息、将与本公司软件一起使用的常见设备、对同类产品或早期产品的批评性杂志评论、其他有关客户和平台的材料。

要采用的测试风格更依赖于预先定义的测试，还是探索式测试？如果主要重用测试用例，则需要每个测试用例的操作和维护文档。如果主要采用探索式测试，则更需要战略和策略文档（有关如何在某个领域测试的想法，但不是测试用例），以及能够使探索式测试更容易一些的已经购置或开发的任何工具的文档。

测试文档应该关注要测试什么（目标），还是应该关注怎么测试（过程）？我们倾向于关注目标的测试文档，但是为了向第三方描述测试过程，一步一步的详细过程描述当然也很有用。

文档应该控制测试项目吗？要让测试员从文档中查阅操作信息吗（例如下一步工作内容的计划进度信息）？

如果文档控制部分测试项目，那么是在项目的初期还是后期进行控制？测试应该主要通过参照测试文档完成吗？如果是这样，整个项目都是这样还是初期测试更多采用探索方式？或者后期测试更多采用探索方式（对看起来运行正常的产品做全面测试）？

谁是这些测试文档的主要读者？这些读者有多重要？如果想要测试员和程序员评审测试文档（例如寻找覆盖率漏洞），编写文档时就要强调设计，并便于看出测试所覆盖的内容。不要强调测试的过程步骤描述，否则评审人员是不会看的。

需要多强的可跟踪性？要反向跟踪哪些文档（规格说明或需求）？谁来控制跟踪？

测试文档要在多大程度上支持项目状态与测试进展的跟踪和报告？应该创建一个处理系统以便统计文档中描述和计划的测试用例数、运行的测试用例数、已通过的测试用例数和当前已发现的程序错误数？测试文档应该在这种系统中起作用吗？例如，测试员是否应该在测试时交互地处理文档，在运行文档所描述的测试时做标记？是否应该收集测试员所做的标记或状态标记，并加入状态报告中？

文档要在多大程度上支持向新测试员指派工作？为了有效地指派工作，必须告诉每个人应该做什么，详细程度要足以使其能够开展工作。有效的指派不一定要求具体步骤指示。我们强烈建议要教会测试员一些技能，为其分配入门性任务以使其熟悉产品文档（例如用户手册），然后下达与其技能和参考材料相适应的工作指示。不是减慢（并堆积大量测试文档）不能发展自己的技能并寻找自己解决方案的测试员的工作，而是用能够快速处理测试文档的员工替代做不到这一点的人。如果认为必须给出详细指示，则必须注意编写有效的详细指示需要很高的技巧。有关这个问题的进一步介绍，请参阅Wurman（1991）。指派工作的另一个关键问题是要能够确定某个人都完成了什么工作，完成得怎么样的能力。如果让某个人在非常简明的文档上做标记（例如我们在第3章的“测试手段附录”中给出的某个矩阵），与翻阅几十或几百页的做了标记的测试脚本相比，可以更容易地找出各种模式。

对新测试员的技能和知识做哪些假设？所有的文字都是给别人看的。读者知道的东西越多，要告诉他们的越少。

要使用测试文档记录项目团队的过程，以提供一套别人做测试可依据的产品模型或描述，或给出发现程序错误的结构吗？对于技能不同、兴趣不同的不同读者，这些都是非常不同的目标。

测试包应该提供预防、检测和预测收益。对于本项目来说，哪一种收益最重要？如果创建测试足够早，并且足够有效地与程序员一起进行过评审，程序员可能以保证其能够通过测试的方式设计程序。由于他们在开发时还考虑测试问题，因此就不会发生那种错误。仅仅通过开发这些材料，并询问程序员关键问题，就可以指出程序员所用方法的风险和弱点。这是测试计划中预防收益的一个例子。以后测试员会得到代码。如果计划能够对测试进行有效的指导，就会找出代码中的程序错误。这就是测试计划中的检测收益。最后，测试结果可能有助于计划项目的其余部分或以后的项目，可能提示出问题所在、常见程序错误类型以及有效和低效的策略。测试结果产生的统计数字，还有助于预测完

成某种任务需要多长时间，以及项目还有多少工作没有完成。测试员的计划工作会得到很多收益。在这三种收益中（预防、检测和预测），如果只能关注一种，那么应该是哪种？对这个问题不同的测试小组有不同的回答。

测试文档（及其测试用例）的可维护性有多高？这些测试文档在多大程度上能够保证测试变更能够跟上代码变更？有些产品规格说明是只读文档。这些文档帮助开发团队创建其初始计划，但是从来不更新。以后的文档根据需要帮助开发团队解决具体问题。有些公司创建不断更新的规格说明，并且规格说明的细节捆绑到正在构建的产品的每个方面。哪种方法适合自己的条件？

测试文档会有助于标识（并修改或重新组织）程序风险模式的永久转移吗？有一种老的看法，认为有一些程序错误的程序部位以后还会有程序错误。因此，更有重点地测试以前发现过程序错误的部位。但是到一定时候，产品的这个部位最终总会完善，而过去认为没有错的程序其他部分可能变得不稳定。是否要通过测试文档设计，帮助确定产品不同部位稳定性的转变？

当我们询问以上问题时，并不想请读者编写一份“需求文档”，并写出所有回答，而是建议读者考虑这些问题，写出自己所需要的东西，需要写多详细就写多详细，以帮助达到测试文档目标。读者可能需要写出有多页纸的报告，记录自己的选择并有助于获得管理层的批准。在有些公司中，如果项目严重拖延，或产品在现场表现出无法接受的质量水平，测试小组需要这种文档为自己的工作进行辩护。而有时一句话的使命描述也可能足够。

经验149

用简短的语句归纳出核心文档需求

以下是两种不同的归纳：

- 测试文档集主要支持我们自己找出这个产品版本中的程序错误、指派工作和跟踪工作状态。
- 测试文档集将支持当前产品开发和至少10年的测试维护，为新测试小组成员提供培训材料，并创建适合管理者或法庭检查使用的档案。

这些归纳会导致有很大不同的文档集。请与本项目所有感兴趣的项目相关人员一起评审自己的这种归纳描述。

第7章

与程序员交互

测试员与程序员的交互在很大程度上源自所报告的程序错误。第4章专门讨论了这个重要问题。本章重点讨论与程序员交互的其他问题。

由于程序员在计算机怎样思考方面是专家，因此程序员常常被看做是机器本身。不要陷入这种错误的思考模式。程序员不是编码机器，他们有感情，大多数人都非常在乎所做的工作。

很多程序员也误解测试员。我们认为避免或对付坏关系的最好办法，就是建立相互尊重基础上的人员关系。假设要与之打交道的人是值得尊重的，并基于这种假设行动。以会赢得他们尊重的方式工作。拒绝接受误解或不尊重。

作为程序员工作的正式批评者，测试员必须敏感、有鉴赏力并有外交手段。不要作为啦啦队长而置身度外，而是要让别人知道测试员理解程序员工作的价值。如果程序员的工作很差，不要为此使他们难堪。

在与程序员打交道时要开诚布公和诚实。先从谈话开始。测试员可与程序员一起讨论本章内容，听听他们的意见。

理解程序员怎样思考

我们三人在专门从事软件测试之前都当过程序员。我们现在仍然写代码。我们的经历影响着我们对程序员的理解，影响着我们作为测试员的工作方式，影响着我们与程序员的协作。

程序员和测试员是在不同条件下工作的，扮演不同的角色。在这些角色中，我们以不同方式思考自己的工作。如果能够考虑到测试员和程序员在观点和方法上的一般差别，会使自己的工作更有效。

测试员了解如何与程序员交互的最好方法，是成为一名程序员，并在一段时间内承担与其他程序员一样的工作。把自己编写的一些产品代码拿来测试，让测试员、用户、经理和其他程序员批评、责备和表扬。从这种体验中得到的认识是我们在本章中的任何讨论所不能提供的。

这里所做的归纳会对不同的人有不同的作用。我们迫切希望读者逐渐理解自己要与之打交道的人，而不是主要依靠以下这些观察。不过读者可能会发现以下归纳中的一些会对自己很有用：

大多数程序员都很专一。程序员常常关注子系统或模型，依靠的是其代码必须与之交互的其他系统要素的粗略信息。形成对比的是，对于要测试的系统，测试员常常是多面手。为了更好地测试，测试员必须理解程序的各个部分如何组合在一起，并必须能够向与之打交道的程序员提供完整系统的信息。

程序员关注自己的系统理论。程序员拥有说明系统组件如何关联，哪个组件是可靠的，以及错误如何传播的模型。他们必须使用头脑中的模型。当程序员告诉测试员所报告的程序错误不会发生时，他们并不是说不会有错误，而是说这种错误与他们的模型不匹配，他们相信模型是没有问题的。测试员关注的是观察和证据，要检验他们的模型。应仔细做记录，在报告中集中说明实际看到了什么，并让程序员根据他们自己的推断找出缺陷。

程序设计是一种复杂活动。程序员耗费了很大一部分精力只是为了理解他们所构建的系统。这种思想集中常常使程序员只关注他们认为是重要的东西，并会对别人的打扰感到不耐烦。

程序员常常要与各种困难做斗争。程序员要对付模糊和不断变更的需求，有很多问题的工具和组件技术，以及不断被打扰的工作环境。

很多程序员不喜欢例行工作，常常构建工具和脚本来自动化自己所面临的重复性工作。很多人都把测试看做是重复性工作，因此自然应该自动化。他们会提出如果测试员不自动化测试就会出现什么问题。不要被他们的话打动。不要把尝试自动化测试当作赢得程序员尊重的一种方法。还有一些更好的方法。测试员的正直和能力需要尊重（经验153）。

有关进一步讨论，请参阅Pettichord（2000b）的文章："测试人员与开发人员思考方式的差异"。

获得程序员的信任

不要与被测程序的开发人员产生不必要的敌对关系。如果要与之打交道的程

序员与测试员共享信息，例如程序员的计划、设计文档的早期草案和早期原型等，测试工作会更有效。找出程序员需要什么样的反馈信息，并为其提供。

测试员越早与程序员接触情况就越好。尽早与程序员接触要求测试员敏感并能提供帮助。在处理早期不成熟的代码时，程序员知道会有很多问题。程序员不想听他们已经知道的，例如需要还没有编写的错误处理代码等。程序员想要知道存在哪些严重问题。确定他们对严重性的定义，并暂时将注意力集中在他们所提供的信息上。作为项目开发人员，测试员有时间开发或影响更独立的准则。

如果测试员不同意某种情况是否会引起问题，可陈述自己的观点，但不要反复唠叨。测试员是否正确，以后就会很清楚。

提供服务

主动直接为程序员提供帮助。这样可以建立信任，并证明测试员是程序员应该与之合作的人。以下是测试员能够提供的一些服务：

- 测试第三方组件。共享测试结果，使得程序员能够决定是否可以在产品中使用第三方组件，以及怎样使用。
- 测试非正式个人版本和原型。
- 为程序员建立测试环境，以便程序员自己进行测试。
- 评审需求文档的可测试性。程序员会由于要求很模糊而感到头疼，他们会很欢迎测试员的介入。

作为测试员，所做的一切都应该是提供服务。以上给出的只是最直接、最明显的一些例子。提供服务使测试员有机会得到信任，也有机会展示自己的才能。

测试员的正直和能力需要尊重

测试员是客户利益的维护者。测试员的工作最终是报告用户可能会遇到的问题。程序员和经理承认这些问题可能会有困难。如果出现这种情况，测试员提供的就是令人不愉快的信息。测试员也许不愿意这样。但是如果测试员发现令人信服的问题，并准确、直接地报告，那么测试员应该得到尊重。当报告问题时，要注意下列问题：

要干脆地报告问题。即一步一步地将问题给出，没有（或很少）多余的步

骤。准确地描述失效征兆。使错误报告易读、易理解。测试员的工作会得到别人的尊重，因为测试员表现出对程序员时间的尊重（作为错误报告的读者，或依靠错误报告寻找线索的调查员）。

将自己的判断建立在产品行为的实际观察基础之上。测试员常常比任何其他人使用被测软件都多，使其成为被测程序外部行为的专家，但是对于程序内部并不是专家。因此测试员应只谈论所看到的现象，不要在猜测内部问题的性质上花费大量时间。

如果失效是不可重现的，要展示为了重现失效所做的各种尝试。当测试员提交一个不可重现的错误报告时，给别人的最好印象应该是已经做了彻底的调查，但是还需要比现有更好的工具和信息。给别人的最差印象是一遇到困难就放弃，并把工作推给程序员。应该展现对程序员时间的尊重。

直接报告坏消息。在向程序员的上司报告之前先向程序员提供使他们能够做出反应的机会，要事先告诉上司自己要上报一个问题。

不要假装了解自己并不了解的东西。例如，如果不知道某个问题的严重性，就不要假装知道。要么收集证据（例如通过技术支持人员或市场开发人员），要么不发表意见，要么明确说明自己是在猜测。

不要夸张错误报告。也不要缩小错误报告，不要在外界压力和引诱作用下忽略或隐瞒自己所发现的错误。如果发现问题，就要坚持自己的观点，报告该问题，如果感到合适还可以向上报告。得到耿直声誉就会赢得尊重。

如果测试员是正直的，就可以展示自己的能力。如果测试员丧失正直性，其能力就会变得毫无意义。

将关注点放在产品上，而不是人上

如果测试员发现程序错误，就要报告。不要报告说程序员Joe是个笨蛋。他也许很笨，但是如果是测试员说出来，就会破坏报告的效果。

很多有经验的测试员都利用对个人和组织弱点的观察，突出查找程序错误的重点。测试员在这一点初尝甜头之后可能不禁想到直接向机构报告观察到的问题会更简单、更有效。错！

一旦把报告问题当作自己的工作，所有程序员都会不再与测试员共享信息，也不会再邀请测试员参加开发小组会议。这会使测试员低效，并成为问题测试员。

不要低估管理层注意到这些问题的能力。人员问题的发现要比解决容易得

多。请注意，测试员并不是（表面上并且也许实际上）忽略问题员工的经理的经理。如果测试员把注意力放在不称职的程序员身上，就会限制管理层处理这个问题的选择。也可能测试员的行为会迫使经理面对他们努力回避的问题。输家还是测试员。

有些测试员走得太远，以至于认为惩罚程序员犯错误、遗漏截止日期、没有遵循过程要求等，都是自己的工作。想想看这样的测试员会有什么结果？会变成垃圾。有的可能会被立即撤职，有的还可能作为现成的坏警察留着，直到有真正的大傻瓜需要替罪羊。

测试员如果发现一些自己感到可能解决不了的问题，单独把证据提供给合适的经理，由经理来处理。测试员只做自己份内的事（Pettichord 2001c）。

程序员喜欢谈论自己的工作。应该问他们问题

很多测试员报告说从程序员那里收集信息有麻烦。我们发现程序员往往喜欢谈论自己的工作。

一个很好的切入点是讨论程序员所使用的设计文档。先做点准备工作，浏览能够得到的所有文档。如果可能，还可以看看代码。

程序员的文档会有很多地方不清楚，测试员要询问在程序员看起来重要但是自己还不明白的部分。有时可以通过电子邮件问问题，但是面对面的交流效果往往更好，特别是还有很多后续问题。如果程序员同意面谈，测试员应该做些准备，以免浪费程序员的时间。

如果程序员没有文档，可以向他们索要系统框图。大多数程序员在自己的头脑中都有整个系统的图像，并且很高兴与别人讨论。

测试员可以请程序员在白板上画出系统框图。一种方法是指着任意一个箭头或方框问："如果不这样会发生什么？"这样会发现遗漏错误处理或无异议的假设。对两个或更多程序员问这样的问题，会暴露出程序员之间很有意思的观点差别。

测试员为什么问这些问题呢？为了更多地了解在建系统，了解系统可能失败的方式，了解人们构建系统所做的假设。不要考他们、难为他们。如果程序员有这种感受，就不会与测试员合作了。

测试员得到答案后，应做好笔记，并与该程序员和其他测试员共享。程序员不喜欢一遍又一遍地对不同测试员回答同样的问题。

如果测试员理解程序员所使用的语言会很有帮助。如果程序员采用C++或

Java编程，则测试员应该对什么是类有所了解。如果软件运行在多线程系统上，则测试员应该知道线程是什么。

积极地聆听，这本身就会使测试员受益匪浅。在讨论时，每个人都会谈出自己的观点、经验或窍门。当测试员在积极地聆听时，应注意尝试帮助其他人说出必须说出的内容，包括用自己的话复述程序员所说过的内容，询问能够启发补充信息或条件的问题，做出推论等。

作为测试员，其工作就是考虑产品怎样才会失效。但是作为团队一员，测试员需要理解在建产品的价值所在。要让程序员知道自己理解他们正在从事的工作的价值。

测试员不要告诉程序员在自己能够开展工作之前，他们必须提供一定文档。如果程序员使用的文档初稿遗漏一些重要信息，测试员可向他们提问。如果需要信息，就向程序员索取。要向程序员解释为什么需要这些信息，以及这些信息会对自己的工作带来怎样的帮助。程序员不会知道测试员在想什么。（参见Gause和Weinberg 1989，第6章；Michalko 1991，第14章）。

经验156

程序员乐于通过可测试性提供帮助

大多数程序员都希望自己的程序能够得到很好的测试，他们努力把工作做得漂亮，知道自己会犯错误，并期望测试员能够找出这些错误。

对于测试员来说，可测试性是能够便于测试软件的任何功能。在与程序员交谈时，更实用的定义是可测试性是可视性和控制（经验137）。这个定义说明了有助于测试员的功能特性。要知道，程序员会提出测试员没有想到（或没有要求）但是很有用的功能。测试员应该提出什么功能要求？经验137给出了几个例子。

很多测试员在试图通过程序员得到可测试性功能时遇到挫折。我们认为，要使这种要求得到满足有三点需要注意：

用程序员的语言谈话。如果测试员能够阅读设计文档和代码会很有帮助。测试员必须以程序员能够理解的方式提出请求。如果能够确切地描述自己想要在哪部分代码中提供什么样的接口，程序员会相当公正地听取这类意见。的确，测试员可能会惊讶地发现程序员已经提供了自己所要求的功能，因为程序员要利用这些功能进行调试或完成其他工作。

尽早提出要求。请参见经验138：尽早启动测试自动化。

要现实。有些可测试性要求很小，可以与其他实现任务一起完成。有些可测

试性要求需要新的功能，像其他所有功能一样，也有预算和进度计划问题。测试员还必须向管理层说明这些问题。

大多数程序员都喜欢编程，如果测试员提出具体、合理的要求，这会给他们施展编程魔力的机会。程序员不喜欢努力猜想别人在想什么，不会欢迎新的模糊要求。

很多测试员告诉我们程序员会有各种借口回绝，例如“这会危及软件的安全”，“这会影响性能”。这些拒绝增加测试代码的借口很少站得住脚。但是我们认为，提出“我们不想考虑这个问题”这些借口常常是在编码阶段。根据实际情况，可能需要一点推销手段帮助程序员认识到帮助测试员最终就是帮助自己。可能需要寻找其观点和影响力会使测试员的目的达到的合适程序员。但是，如果确定了目标，要在合适的时间提出要求，以合适的语气提出，我们认为程序员会建设性地考虑测试员的要求。

我们曾经见过很多对产品测试带来很大好处的可测试性功能。有些是应我们中的一位的要求提供的，有些是团队中的测试员或程序员提出的。提出可测试性要求很难，但是很值得。我们建议读者要有不屈不挠的精神。

第8章

管理测试项目

管理测试项目在某些方面与管理任何其他类型的项目一样。但是测试项目至少有一个特点：测试项目是受编程项目驱动的。测试员的工作是对程序员工作的反映。这也是为什么使用像Microsoft Project这样的工具计划测试任务会遇到很大挫折的原因。把团队工作用小小的甘特图表示出来，需要费很大的劲。本章将概述推动以及控制测试项目方面的经验。

经验157

建设一种服务文化

项目团队开发软件是为了使客户受益。客户可以是内部的，也可以是外部的，可以是付费的，也可以是非付费的。客户可以与开发人员是同一批人（例如当开发工具时）。

测试员为整个项目团队提供服务。典型的服务就是发现并报告程序错误。其他服务取决于测试小组的使命（请参阅第1章的相关内容）。

贯穿测试文献和测试亚文化的基本问题之一是，测试员的角色究竟是服务还是控制：

- 服务提供者控制他尽更大努力所提供服务的质量和相关性，以取得最终结果。我们向需要服务的人们提供优质服务。
- 服务提供者不控制最终产品的质量，不控制其他服务提供者（程序员、文档编写员、市场开发人员）所使用的过程，不批准或拒绝产品的发布。服务提供者不是项目经理，而是要向项目经理提供服务。

不要尝试建立一种控制文化

测试员常常收到并给出有关项目应该如何管理的详细建议。我们觉得这些建议在很大程度上太天真。最差和最武断的建议都来自其开发经验只局限于测试（或度量和评估）其他人工作的测试员和顾问。

测试员常常从最坏方面看待项目。测试员要面临很差和不完整决策的结果和完成了一半的任务，没有注意到计划和使产品得以完成的工作。看起来很差和不完整的决策常常是测试员所不赞同的精心考虑的业务决策。测试员很容易自认为理解得更好，做得更好。

有些过程使测试员更容易一些，有些过程使测试员更困难一些。但是，胜任的测试小组可以向各种项目经理提供一致的服务，要在非常不同的项目管理风格下工作，包括不方便的风格、以及使测试成为低效项目开发一部分的风格。

有些过程看起来注定要产生差的产品。这是严重问题，应该有人进行管理。但是，要管理的最差的小组就是测试小组。测试小组没有资源、经验或行政力量来改正更广的开发过程，也不能管理被改正的过程。

我们并不是说测试员（即今天进行测试的人）应该了解自己的责任并不要越职。远远不是这样。我们鼓励测试员扩展自己在公司中的作用和影响。如果读者想要并有能力管理项目经理，那么就管理项目经理，但是要以合适的角色管理项目经理，即先当上项目经理的经理，测试经理并不是管理项目经理的角色。

要发挥耳目作用

测试员在公司中的权力建立在自己的调查技能和自如沟通的基础上，而不是建立在命令链上，因为测试员在项目团队的命令链中的位置并不很高。

在理想情况下，我们可以向公司中受影响的任何小组指出问题。没有解决的重要问题会使某些小组付出大量经费。如果他们认为问题相当严重，他们会努力给予解决。如果他们认为问题还不够严重，就可能不解决。并不是所有问题都要解决。

在有些公司（我们经历过一些）中，测试员向任何想了解的人演示所发现的程序错误并提供状态报告。项目经理会感到恼火，会抱怨，同事和经理会让他们别抱怨：测试员就是测试员。

有些公司层次性更强一些。在这样的公司文化中，测试员访问市场开发人员

或技术支持经理，讨论正在开发的产品的问题是不合适的。

如果测试员的任务是帮助公司针对所发现的缺陷做出正确的判断，那么若不针对这些缺陷与公司中受这些缺陷影响最大的人沟通，就不能有效地完成任务。如果不能直接就这些发现进行一对一的个别沟通，可以间接沟通。例如，可以授权其他人访问程序错误跟踪数据库，并教会他们如何使用该数据库发现他们感兴趣的问题。也可以分发暴露出关键问题的状态报告。由于其他小组的人认识到所提供的这些信息的价值，并依赖这些信息，因此他们（常常）倡导向他们提供信息，并为测试员的这种权力辩护。

测试员必须评估自己公司的文化，要在不被解雇或排斥的限度内发挥作用。在这个限度内，我们建议测试员要成为能够为别人带来价值的守信用、高度正直的信息提供者，以此来施展和扩大自己的影响力。根据我们的经验，与通过例行管理渠道，例如有权拒绝签署（批准）产品版本发布相比，通过提供信息能够为自己赢得更大的更实际的影响力。

测试经理管理的是提供测试服务的子项目，不是开发项目

测试工作是整个项目的一个子项目，要申请资源并提供服务。项目经理在如何运作测试项目上有很大控制权，应该仔细地选择自己的管理风格，从而与项目经理的管理风格协调。

有时项目经理会犯错误，有时会受益于测试经理的建议。测试经理要通过各种方式提出建议。要把自己的想法说出来。但是如何运作项目的决策权在项目经理手中。如果项目经理没有采纳自己的建议，那么接受现实。

我们不打算在本章过多讨论在强大进度压力下如何管理人员的问题，第9章会更多地讨论这个问题。现在我们只是指出，如果项目经理和执行经理做出差的决策，有时对测试员做不恰当的批评，要求连续加班，要求绝对服从他们。这已经超出项目经理合理的职权范围。作为测试经理，自己工作的一个很重要的部分就是要保护下属人员不被滥用。

所有项目都会演变。管理良好的项目能够很好地演变

在每个项目的整个过程中，测试经理都应该准备（正常情况下）对整个计划的大小细化或更正。

项目就是一组任务。随着时间的推移，项目团队会发现有些任务比预期的要困难，或要花费更长时间；有些任务还不能完成，因为这个任务的关键人物正在忙于其他事情；有些任务与前一周相比，对市场开发经理或客户来说更紧迫。此外，每次测试员提出一份错误报告时，都会在大堆的任务中再增加一项。

项目团队要为集成新信息、确定下一步（可能）做什么、项目完成前的迭代工作提供某种框架。要把项目看作是有关下一步应该做什么的正在进行着的结构化对话。

总会有很晚的变更

很多传统项目管理方法的目标都是限制和控制变更，但是有些方法则接受变更（例如，请参阅Weinberg 1992、Beck 1999、Beck等2001和Krutchen 2000）。但是所有项目管理方法都必须处理变更。

请想像造一把新椅子来取代破损的椅子。需要什么、谁需要、怎样使用、椅子的承重有多大，这些都很明确。还可以找出能够造出非常接近自己要造的椅子的人。

软件与此不同。在大多数软件项目中，没有人以前创建过与这个产品完全一样的产品，即使有人创建过，这个项目团队的成员也没有创建过。不仅如此，将使用这个软件的人以前也没有使用过这种产品。即使他们可能对自己想要的东西有很好的想法，也不知道如何确切地描述自己的需求，这是因为：

- 他们不知道自己的所有需求。
- 当他们试用软件的早期版本或竞争对手的产品后，其需求会发生变化。他们会发现试用该软件的新方法，并想出自己也可以利用但是现在还不能利用的新用法。
- 不同的项目相关人员具有不同的需要，这些需要常常是矛盾的。没有一份文档可以说清所有这些矛盾和潜在矛盾的需求，并进行综合考虑。

不仅如此，随着组件和工具的构建和技能的提高，提供给定功能的预期成本会发生变化，使满足该功能变得多少容易一些。

需求是在我们想要和我们能够得到的功能之间进行不断斗争的结果（Bach 1999a）。随着项目的展开，需求会变化。

项目涉及功能、可靠性、时间和资金之间的折衷

项目经理的任务就是按时并在预算限度内交付一组合适的功能，达到合适水

平的可靠性。这是一种具有挑战性的折衷。

功能。选择合适的功能集，交付所有项目相关人员所要求的一切功能太昂贵。

可靠性。使产品能够正常运转，但不能花费无限时间和资金保证在所有能够想像出的环境下都能完美地运行。

时间。尽可能迅速地将产品投入正式使用或销售。

成本。以最低的合理成本构建产品。成本包括资金和机会成本。当把关键资源（特别是技术熟练人员或独特设备）用于项目团队时，其他项目团队就不能使用该资源。

我们用“折衷”这个词来刻画这一点，因为项目经理可以通过增加或减少一种要素上的支出而获取另外要素上的更多收益。例如，如果花费更长的时间（和更多的资金）构建产品，就会有更多功能。

如果要理解项目经理看起来奇怪的决策，可尝试理解这种决策如何影响各种要素（功能、可靠性、到产品交付或投入正式运行为止的日历时间和成本）。

经验 164

让项目经理选择项目生命周期

生命周期模型是从最初考虑构建这种产品，到向公众发放并投入使用为止，产品设计和开发过程的描述。

我们反对软件开发生命周期单一最佳选择的想法。每种模型都确定项目的一些方面，而其他方面则留待修订。没有确定的部分的每种生命周期模型都不一样。

有些公司遵循标准生命周期模型，但是根据我们的经验，项目经理总是有一定的定制余地。明智的项目经理会挑选能够流畅地控制自认为很难管理的方而的方法，而预留出自己特别有能力解决的方面。每种选择都有风险和后果。所做的选择并不会总为测试小组提供方便，也不会总与测试经理的选择一样。做出决策的权力属于项目经理。

在后面的两条经验中，我们将研究瀑布和进化生命周期。经常被测试顾问大力提倡的瀑布生命周期并不总能解决我们所关心的问题。进化模型对于测试员常常更好一些，但是对于项目团队其他成员可能更困难。

我们并不是说应该认为进化方法比瀑布方法优越，也不是说要提倡任何生命周期。我们要指出的是，生命周期选择相互之间有很大不同，做出选择并不容易。

瀑布生命周期把可靠性与时间对立起来

瀑布模型是一种描述按阶段推进项目管理的具体生命周期方法。这些阶段包括：

- 问题定义（要构建什么，以及为什么构建）。
- 需求定义。
- 内部和外部设计。
- 编码。
- 测试。
- 安装。
- 安装后支持。
- 失望客户的法律诉讼。
- 问题定义（针对产品的下一个版本）。

瀑布模型因其阶段框图得名，因为在一些人看来这种阶段框图像瀑布。“瀑布”这个词描述了这种方法的串型特征，要从一个阶段转向下一个阶段，很难回到前一个阶段，就像瀑布的水不能向上流一样。

在实践中，会出现一定回溯（在我们尽力满足所描述需求的产品时，发现需求描述有错误或不能实现），但是不鼓励变更控制。

瀑布的一种变种描述了更宽的测试角色，即V字模型。在每个阶段的最后，测试员都要正式评估该阶段的工作产品。例如，测试员要在需求定义阶段结束时评审（并可能批准）需求文档。测试员还要编写一系列测试，在后续阶段结束以及系统的相关部件能够操作时运行[⊖]。

瀑布（V字或非V字）像是一种很清晰的过程，但是如果项目团队不能在恰当的阶段提交所需的一切会出现什么情况？如果项目明显滞后于进度计划会出现什么情况？

⊖ 对于V字模型我们还有其他一些考虑。在代码交付之前编写详细测试的过程是有风险的。如果产品的设计变更，测试就会过时。到代码完成时，测试员所编写的很多测试针对的都是没有编写的产品功能，或以与最初设想不同的方式实现。对于很多项目，所有这些案头工作都没有用，如果有测试代码和很多测试数据则更糟。这些案头工作在发挥作用之前就已经过时。在V字模型过程中，这些测试小组的工作产品可以供程序员使用，帮助他们在编码之前发现所提出的功能的模糊点和弱点。我们当然赞同设计评审可以改进设计并避免出现一些问题。但是如果这就是目标，我们认为与事先编写永远也不会运行的测试相比，设计审查和代码评审对产品质量的贡献要大得多，所用的时间要少得多。我们建议只要设计和代码已经可以提交评审就立即评审，不必等到阶段结束时。

大多数软件项目确实会明显滞后于最初的进度计划。我们所了解的有经验人士的忠告可归结为“精心组织的项目不会出现这种问题”。但是软件项目有大量风险。总会有很晚的变更。“这不会发生”只是一种愿望。

那么，如果项目明显滞后于进度计划会出现什么情况？

在瀑布模型下，到测试员得到代码时，所有功能都已经设计，且大部分或所有功能都已经编码，大部分软件开发经费已经支出。关键的折衷要素是时间和可靠性。要么修改错误而晚一些交付产品，要么较快交付产品，但是有较多错误。

这是项目经理和测试员之间的经典争持：是交付错误很多的产品，还是延迟交付（或投入正式运行）。对于这个问题，很多测试员都要求更严格地遵循瀑布项目管理模型。这不是一种解决办法。在瀑布模型中总不可避免地会存在可靠性和时间之间的折衷考虑。

在倡导瀑布或经过修改的瀑布模型（V字模型）时应该小心。

进化生命周期把功能与时间对立起来

在软件开发进化方法中，项目团队每次增加一个功能。他们设计该功能、编码、测试，并更正其错误。如果集成了这个功能的产品满足了项目团队的质量标准，他们就会增加下一个功能（Gilb 1997和Beck 1999）。

开发团队可以在任何时候发布该产品。（发布通过测试的最新版本。）今天的版本和下个月的版本之间的差别是下个月的版本有更多功能。这两个版本都能使用。不存在项目结束时的时间和可靠性之间的折衷考虑。

这种方法也有自己的难题。请想像自己是市场开发经理或技术文档编写员。找到项目经理，问他：“这个产品中有什么功能？”回答当然是：“功能不是一定的。我们什么时候交付的？”必须知道在产品发布时产品中都包含了哪些功能的任何人，都会发现进化方法的难题。有些人认为，与进化方法相比，瀑布方法更容易控制功能集合不确定和功能蔓延的风险。

愿意在开发初期将资源分配给项目团队

人们越来越普遍认识到，测试员应该介入开发周期的早期阶段。但是，测试小组一般都没有投入足够人力，而且工作负荷过大。在开发初期，能够得到什么重要成果值得将测试员撤离开发关键点呢？

- 如果只是想让测试员参加早期的项目团队会议，那也许是浪费测试员的时间。
- 如果测试员不了解所使用的程序设计语言，派他参加代码评审常常是浪费时间，也许更糟的是，他可能表现出无知，从而失去程序员的尊重。

测试员参加早期开发可以是很有价值的活动。以下举几个例子：

- 他们可以评审需求文档的可理解性、可测试性和模糊性。
- 随着其他项目工作制品（文档、代码等）的开发，再对它们进行测试。不要等整个产品完成才测试。当工作制品的作者说（或测试员认为）工作制品接近能够进行有用的测试和评审时，就应该开始研究。
- 推动代码评审。代码评审是收效很大的一种质量改进工作。通过搞好后勤工作（预定会议室，准备茶点）和主持会议（召集会议，明确需达成一致的意见），测试小组可以帮助公司搞好代码评审。通过代码评审会议，测试小组可以了解很多信息，但不会（而且不应该）对评审文件做评论。为了很好地推动代码评审，测试小组应该接受培训。
- 拟定硬件配置和准备购置或借用设备的初步清单。
- 要求提供可测试性功能。提供这类功能要占用设计和编程时间。如果在他们的预算和进度计划中不把提供可测试性功能考虑在内，功能就不会通过代码实现。
- 讨论代码覆盖率度量和使用其他开发支持工具（例如Purify或Bounds Checker）的可能性。为了更好地使用这些工具，测试员需要得到支持（时间和至少有一名编程小组成员的参与）。如果项目团队在预算中没有考虑这一点（也许占软件测试时间的一半），也做不到这一点。
- 准备测试自动化。这种准备包括就测试自动化的内容和测试自动化支持人员的水平达成共识。
- 研究测试工具。订购测试自动化支持软件和设备，学习如何使用。
- 如果可能，订购可用于被测软件的外部开发的测试包。更一般地，寻找可以用于预测的软件，以便于进行大量测试。
- 了解产品的市场和竞争情况。要成为这个市场熟练应用至少两种应用程序（非自己公司出品的）的熟练用户。

合同驱动的开发不同于寻求市场的开发

如果公司要根据合同进行软件开发，则合同要描述各方的责任。合同可能要描述一组功能，描述程序该如何编码、测试、编写文档以及提供技术支持。开

发公司的主要责任是根据合同完成自己的义务。如果提供了合同所描述的功能，客户就必须向开发公司支付合同款。（这里的“客户”是指支付软件费用的人或公司。）随着产品的开发，客户可能会改变有关产品某些功能的想法。很多项目团队会对这种变更进行控制，因为这些变更会影响项目成本。（不仅如此，可以使用变更控制过程弱化客户的谈判地位。）

软件工程教科书中给出的建议，很大程度上都可以很好地用于根据合同进行开发的大型定制软件开发项目。

寻求市场的开发与此不同。客户要到开发公司已经将产品开发出来以后才会购买。在整个开发过程中，项目团队的主要考虑是所发布的产品是否能销售给目标市场。如果竞争对手发布更有吸引力的产品，发布得更快，或市场开发工作做得更好，那么是否严格遵循规格说明就没有什么意义了。市场开发人员和销售人员在整个开发过程中，都会根据所搜集到的客户、竞争对手和媒体期望的最新信息，而要求修改设计。

在合同驱动的项目中，测试员的主要活动是对照一组规格说明测试软件。在寻求市场的项目中，测试员更可能根据不同客户的预期，来研究和测试产品。

要求可测试性功能

如果读者是产品测试小组的负责人，那么可能就是这个项目的第一个测试员。越早提出可测试性功能要求，程序员和项目经理越有可能把它列入预算和进度计划。如果可测试性功能没有被列入项目的预算和进度计划，则很可能得不到这些功能。

一般来说，测试经理要负责使整个项目团队了解测试小组的需要，以及使测试小组更有效、高效地发挥作用所需的信息和支持。

经验137“可测试性是可视性和控制”更详细地讨论了可测试性的思想。

协商版本开发进度计划

测试过程应该与软件修改的步调协调一致。公司每个月都开发出一个新版本吗？每周一个新版本吗？每天三个新版本吗？

测试经理也许不能处理太频繁的版本。检查版本（检验是否通过基本测试）然后再更新所有人的计算机都需要时间。如果每天都开发出一个版本，那么在

更新之前测试一个版本需要几天的时间。

有些程序员要求只报告当天版本的程序错误。他们认为，关于前一天版本的任何报告都没有什么价值，因为他们也许已经发现并解决了这些问题。这对程序员当然是方便的，但是对于测试员可能就是恶梦，因为每天都必须停下所有的事，更新计算机，对新版本重复完成一部分的测试后才可以继续前一天的工作。应该一整天完成的任务，可能要分两三天完成，因为增加了很多管理时间和干扰。

我们所知道的惟一解决方法是制定版本进度计划，内容包括测试小组接受软件的频度，对提交的被测试软件的要求，以及在最近版本中发现的程序错误如何在新版本中重现。

了解程序员在交付版本之前会做什么（以及不会做什么）

有些编程小组在向测试员交付新版本前做了大量单元测试，而有些小组则没有。有些编程小组把其冒烟测试作为其开发过程的一部分，而有些小组则没有。

与测试员协作的编程小组会按自己的过程工作，不要指望他们完成或不完成某种类型的测试，也不要指望对要交付给测试员的版本做各种准备。要了解他们的开发过程，并根据自己对他们的了解来确定他们所做的事。

为被测版本做好准备

当被测版本就绪时测试环境也应该就绪，这一点很重要。对于Web开发更是如此。在快节奏的项目团队中，没有管理良好的测试环境的测试小组是没有价值的。

有时测试员应该拒绝测试某个版本

测试员偶尔也会回绝某个版本，拒绝测试。这样做可以有充分的技术理由：

- 由于这个版本应该加入某个关键功能，因此很重要，而测试员发现这个版本中没有该功能，或马上失效，那么进一步测试就是浪费时间。
- 如果以前正常的关键功能现在不能正常使用了，则该版本也许用错了文件，或很快就会有替代的文件。在这个版本中发现的程序错误可能会被忽略。

（“对，这个程序错误是坏的某某文件造成的。任何问题都有可能由这个糟糕的文件引起。现在还出现那个程序错误吗？”）测试小组的冒烟测试应该发现这种失效。当冒烟测试失败后，一般都要拒绝该版本。

- 如果收到一个版本，并且已知另一个版本在几个小时之后就会完成，并且不会以任何方式受在这个版本中发现的问题所产生的任何影响，取决于检验和安装一个版本的成本，可能需要忽略该版本，在等待下一个版本的同时继续测试老版本。

一般原则是，如果会使测试工作在不能得到明显收益的情况下效率受到很大影响，或对某个版本的测试结果会被忽略，则应该拒绝测试该版本。

使用冒烟测试检验版本

冒烟测试（又叫作健全性检查（sanity check）或接受测试（acceptance into testing））是一种测试包，其目标是检查版本的基本功能。如果该版本没有通过测试，则可宣布该版本太不稳定，不值得测试。

在一般情况下，当某个新版本提交测试时，要有一名测试员运行冒烟测试（可能是自动化测试、手工测试或自动和手工测试的结合）。其他测试员要等到该版本通过冒烟测试后才投入测试。一般在冒烟测试中包含一系列标准的核心测试，以及少量对这个版本特别重要的程序错误或特别功能的临时测试（使用过几个版本后就不再使用）。

冒烟测试过程是公开的。在冒烟测试过程执行得好的公司，程序员很愿意复制描述冒烟测试的所有文档，以及运行冒烟测试的自动化测试代码。在一部分这样的公司中，程序员把运行冒烟测试作为版本过程的自动化部分。每个人都知道，如果程序不能通过冒烟测试，测试小组就会拒绝该版本。这个过程没有什么可奇怪的。在这些环境下，冒烟测试被看作是一项技术任务，而不是行政任务，而且项目团队中的大多数成员（包括大部分或所有经理）都认为冒烟测试过程是合理的。

有时正确的决策是停止测试，暂停改错，并重新设计软件

如果不管进行了多少次程序错误的修改，在同一位置总发现问题，或不管对用户界面做了多少小修改，仍然存在使用户困惑的问题，也许应该停止测试并

对有关代码进行调试。这部分内容可能需要重新设计或重写。

测试经理可以向项目经理提出这种建议。测试经理处于使用程序错误跟踪系统以显示支持自己建议的统计数据（很多报告、很多修复、很多新失效）的有利地位。

我们建议测试经理私下单独向项目经理进行这种说明或建议，并且要认识到有可能不能说服项目经理采取自己所推荐的行动。管理项目团队的是项目经理，测试经理要向其提供服务，包括好的建议，但是项目经理也会犯错误。他的好决策最初在测试经理看来也许是错误的。

与本书其他地方一样，在这里企业文化也要发挥作用。有些公司期望测试经理公开提出这类建议以及相关的数据，而我们的主要建议仍然适用于测试经理提出建议的语气：测试经理是在提出问题和建议，对此项目经理可以采纳也可以拒绝。

根据实际使用的开发实践调整自己的测试过程

有一位顾问在最近的一次软件测试会议上，建议测试员拒绝测试产品，除非程序员向测试员提供完整的规格说明。这是很糟糕的建议。

遗憾的是，类似这样的建议相当常见，而且这种状况已经持续了至少20年。我们认为采纳这种建议更可能使测试经理的诚信受到影响，或被解雇，而不会使公司的过程得到任何改进。

让我们记下并将这个信仰问题放在一边：通过大量文档对付后期更改（例如瀑布模型）的方法是不是真的惟一的“好工程”？这就是这类建议背后的假设（常常明确指出）。

在我们看来，这种拒绝引起的关键问题是：

- 测试小组是否应该设计自己的实践，要求公司内的程序员做他们现在不做、不愿意做并且不必做的事？
- 这是否现实？
- 从什么时候开始测试经理成为项目经理或开发副总裁的？如果测试经理要管理项目团队，则先要成为有权管理项目团队的人。

我们建议不要改变程序员的工作方式。销售人员和来来去去的顾问与测试经理的成功没有利害关系，也不会承担失败后果，他们在确定公司内程序员的恰当责任方面并没有权威性。

“项目文档是一种有趣的幻想：有用，但永远不足”

——Brian Marick

即使对于试图充分描述产品的项目团队，开发文档（例如规格说明和需求文档）也和想像有很大差别。不要对抗这个事实，这是一个基本问题。

请考虑错误处理。有人估计，在现代软件项目中，超过80%的代码用于实现错误处理，实现主要控制流的代码不足20%。但是即使完整的规格说明也只会用不足20%的篇幅描述错误处理。这意味着80%的代码是程序员边编码边设计的。

在使用规格说明时应要求提供特定补充信息以弥补这种差距。不要根据文档完备、一致或准确的假设设计测试或规划项目。

测试员除非要用，否则不要索要

测试员愿意说：“没有规格说明我怎么测试？”如果要求提供规格说明就要使用它。要保证项目经理和规格说明编写人员知道测试员要使用，并且知道将如何使用。否则以后他们就不会向测试员提供会给他们带来不便的任何材料。他们会说：“我为什么要给你这个？你让我花了那么长的时间为你编写规格说明，但你根本没有用。”

充分利用其他信息源

如果没有人向测试员提供规格说明，测试员并不会没有作为，还有很多其他信息源可以帮助测试员把握思考的方向。

例如，测试用例可以请教市场开发或开发人员。这些测试用例可以用来进行基于会话的测试，与系统化地推敲功能表相比，这种方法能够更快地发现严重设计错误。

以下是一些其他有用的信息源，可用来补充规格说明所没有提供的信息：

- 用户手册草稿（以及以前版本的手册）。
- 产品市场开发文献。
- 市场开发人员向管理层做的推销产品概念的演讲。
- 程序每个新的内部版本附带的软件变更备忘录。
- 内部备忘录（例如项目经理对工程师描述的功能定义）。

- 已出版的风格指南和用户界面标准（例如由苹果计算机公司和微软公司出版的指南）。
- 已出版的标准（例如C语言）。
- 第三方产品兼容性测试包。
- 以出版的规定。
- 错误报告（以及对错误报告的响应）。
- 程序逆向工程的结果。
- 与人员面谈，例如开发小组负责人、技术文档编写员、客户服务和技术支持人员、领域专家和项目经理。
- 头文件、源代码、数据库的表定义。
- 所使用的所有第三方工具的规格说明和问题清单。
- 原型以及针对原型的所有实验室记录。
- 与最新版本开发人员面谈。
- 前一个版本的客户电话记录（在用户现场发现了什么问题）。
- 可使用性测试结果。
- β测试结果。
- Ziff-Davis SOS CD和其他技术支持CD，针对本公司产品的问题，以及所使用环境或平台上的常见问题。
- 描述常见错误的BugNet和其他Web网站：www.bugnet.com、www.cnet.com，以及www.winfiles.com、邮递列表、新小组、Web讨论网站等的超链，讨论自己产品的问题和失效定位方针（可能有已经发表的针对所使用平台的程序错误定位方针）。
- 兼容产品（了解其功能、问题和设计，再与自己公司的产品进行比较）。请参阅listserv's、NEWS、BugNet等。
- 与要仿制的产品进行确切比较。
- 内容参考材料（例如交通图可用来检查自己的在线地理系统）。

向项目经理指出配置管理问题

程序错误修复损坏产品中的其他功能，或使以前解决了的问题重新出现的可能性有多大？对于不同的公司和项目团队，出现副作用的可能性有很大不同。我们见过在有的项目团队中几乎没有副作用，也见过受副作用严重影响的项目团队。

当代码很老、反复修改、代码之间严重相互依赖（很强的数据和/或代码耦合）以及文档问题严重时，很有可能出现副作用。但是即使是相对干净的代码有时也会有副作用问题。出现这种情况的根源之一，就是很弱的配置管理。

如果已经解决了的问题重新出现，再解决，再出现，再解决，反复多次，那么这时也许应该检查源代码控制问题。要么编程小组能够解决这个问题，要么测试小组不断反复测试在上一个版本中已经通过测试的代码。

有时问题不是出在配置管理上。为了说明这个问题，Rex Black告诉我们："有一个测试子项目在4个月的时间内发现了几百个程序错误，平均每个程序错误会再次出现2.5次。"他说："这不是配置管理问题，而是由于不切实际的进度计划所导致的错误修改太草率。"

不管是哪种情况，都要与项目经理讨论这个问题，并要求提出解决意见。

如果问题还得不到解决，在状态报告中要说明需要高级别的回归测试（以及为了实现这种回归测试所要完成的工作）。在一定程度上，这像是公司内部的一个递归问题，需要增加未来项目的预算，以便实现大量自动化回归测试。

程序员就像龙卷风

测试工具推销员试图说服某个公司通过使用一种捕获回放方法来创建用户界面层次上的测试脚本。公司告诉推销员这行不通，因为预期程序员会对软件的用户界面和功能集做后期修改。推销员回答说，测试员应该推动程序员实现真正工程化。推销员说，真正工程化意味着要在项目的相对初期冻结用户界面和功能集，然后测试员就可以采用这种工具进行自动化测试了。

类似这种推销的价值，就在于可以帮助测试经理认清这位推销员（及其公司）不是应该与其打交道的人。就像前面介绍的坏建议一样（除非测试员能够得到规格说明，否则拒绝测试，请参阅经验176），把测试方法建立在公司内部编程小组不会实施的实践基础上，是没有意义的。

在美国的中西部，住宅都有地下室。过去很长时间以来，地下室的用途之一就是躲避龙卷风。当然，并不一定非要花钱建地下室，也许可以不建地下室并宣布以后不会再有龙卷风。但是这种宣布不会有什么用，特别是当下一个龙卷风就要来临的时候。

程序员就像龙卷风，把他们看作是一种自然力量。程序员会按照自己的方式做，而且在不同的公司中程序员的工作方式也有很大不同。测试经理应该相应地设计自己的实践。

好的测试计划便于后期变更

如果后期变更是不可避免的，那么测试经理的责任是设计能够适应后期变更的测试过程。以下是一些建议：

- 不要在测试之前开发大的测试包，而是在需要测试包时再开发。如果产品的后期变更使这些测试过时，但我们至少还用过一段时间。
- 不要编写维护成本很高的大量测试文档，例如详细手工测试脚本。应该使自己的文档尽可能简洁。
- 不要把手工或自动化测试捆绑到特定的用户界面，除非想要专门测试该用户界面。即使进行的是必须通过用户界面的端对端测试，也不要将测试捆绑到用户界面的低粒度细节上，因为用户界面会变更。
- 通过最大化可维护性和跨平台可移植性来设计自动化测试。（请参阅第5章相关内容。）
- 构建一组能够在不同程序中重复使用的通用测试。这样可以节省规划时间，并在项目后期增加新功能或变更老功能时，能够容易地使用。
- 构建一组很强的冒烟测试，以快速检测被测软件中的基本失效。如果程序员做出重大变更，则很有可能经常重新构建软件，并愿意迅速给测试员送来新版本，每次集成少量变更，并尽可能迅速地将变更交付测试。冒烟测试可以以很低的成本剔除坏的版本，便于测试员处理频度很高的新版本测试（例如每天一个版本）。
- 慎重考虑使用极限编程（Extreme Programming）法开发自动化的测试（Beck 1999和Jeffries等2000）。具体地说，我们建议构建一种整体体系结构并设计一系列自动化测试，然后反复设计和交付代码，采用方法的优先顺序是：最小化项目风险（指测试子项目）、成对编程和与项目相关人员密切合作（其他测试员、程序员和项目经理），以确定下一步应该做什么。
- 开发一种产品用户和用户要通过产品获得效益的模型。通过这种模型导出复杂测试。这些测试中的大多数都不会随项目的推进而快速变化，因为这些测试关注的是收益，而不是实现细节。
- 帮助程序员开发大的单元测试包，以及相对简单功能的其他测试。每次重新构建代码时，在提交测试小组测试之前都可以自己运行。

这些建议的重要性还没有达到一般原则的程度。测试经理要分析自己的测试实践和环境条件，确定当软件出现后期变更时，会有哪些成本支出和效率降低的地方。然后寻找变更自己测试过程的途径，以降低成本，或把成本分摊到整

个开发周期中，而不是集中到项目后期。

经验183

只要交付工作制品，就会出现测试机会

程序并不是一下子写成的，要一个函数一个函数，一个功能一个功能地实现。对于规格说明、需求文档和使用手册也有类似的情况：也要一节一节、一章一章地写。任何时候产品的任何部分可以提交评审，测试机会都会出现。通过这种方式可以把测试溶入产品的整个开发过程中。只要工作制品已经能够测试，都要尽快测试。

经验183

做多少测试才算够？这方面还没有通用的计算公式

我们都希望能够有某种方式保证已经完成了足够的测试。事实上，人们已经提出了很多计算公式。我们认为所有这些公式都有严重问题。不仅如此，我们认为对的测试不确定性要比错的测试确定性好。关于多少测试算够的好的决策，必须依靠自己的技能和判断，考虑与这个问题有关的因素也是这样。

因此，不要费心寻找计算公式，还是应该多开动脑筋。

经验185

“足够测试”意味着“有足够的信息可供客户做出好决策”

由于测试是一种信息收集过程，当收集了足够的信息时就可以停下来。可以在发现了所有问题之后停下来，但这需要无限多的测试才能知道已经找到所有问题，因此这是行不通的。当有理由相信产品仍然有重要的未发现问题的可能性很低，就可以停止测试。

判定测试是否足够好（存在未发现重要问题的机会足够少）有多种因素：

- 测试员知道要发现的重要问题的种类，如果存在这种问题的话。
- 测试员知道产品的不同部件如何表现出严重问题。
- 测试员对产品做了与这些风险相应的检查。
- 测试策略具有合理的多样化，以避免视野太窄。
- 测试员使用了所有可用的资源进行测试。
- 测试员满足客户期望满足的所有测试过程标准。
- 测试员尽自己的可能，清晰地表示测试策略、测试结果和质量评估。

如果熟练、忠实地做到这些，那么在交付之后存在大的问题可能有以下三种原因之一：

- 测试员不想按照自己想像的那样了解风险的动态。现在知道得深入一些。
- 测试员在测试中出现错误。下一次会做得好一些。
- 测试员的风险评估是正确的。但是管理层决定冒风险，并造成损失。

测试经理对采集多少信息才算够的理解能力，会随着参加产品线的开发而增长。在测试中遗漏问题不算问题，粗心、不认真思索或不通过实践吸取教训才是问题。

不要只为两轮测试做出预算

有些测试实验室只为两轮测试做出预算。他们认为，第一轮会暴露所有问题，第二轮检查所有错误修改。只要不增加新的测试，除了这些错误修改外，产品不会再以任何方式变更，并且这些错误修改都很好，其他的功能运行得也很好。请回到现实世界中来，产品测试不得不进行的次数也许比两次多得多。

- 随着对产品了解的逐步深入（第二次测试会比第一次测试了解更多信息），测试员会考虑新的更好的测试，并找出新问题。如果测试经理采用两轮模型，则在第二轮测试中创建新测试就会受到很强的抑制。
- 即使所有程序员都试图解决在第一轮测试中发现的所有问题，他们解决了所有问题并且不引入新问题的概率微乎其微（除非只发现一两个问题）。
- 测试员经常发现在第一次测试中有些测试甚至不能运行，因为缺陷妨碍了测试的运行。这些是阻断式缺陷。被阻断的测试以后才能进行第一次有效运行。那时这样的测试发现问题怎么办？

我们经常发现项目团队只为两轮测试做出预算，然后陷入困境。由于最初的预期是不现实的，因此进度变更是不可避免的，但是每个变更都被测试小组看作是一种失效和拖延。

为一组任务确定进度计划，估计每个任务所需的时间

测试员的工作由大量任务构成。请列出任务清单。（请参阅Kaner 1996b和有关工作分解结构估计的任何项目管理专著。）有些任务只能进行一般描述，有些任务可以分解得相当细。根据自己所能，对需要一天以上时间完成的任务单独

列出一项。估计（猜测）每个任务会占用的时间，然后累加起来，再加上25%（根据公司具体情况，可多可少）的会议、培训和其他非项目工作，并以此估计所需的总时间。

这种方法听起来容易但做起来难。列出任务并不是一件简单的事，因为很容易遗漏任务或低估任务范围。

尝试采用其他估计方法进行收敛估计：

- 如果测试员完成过与此类似的项目，那么可以根据以前所用的时间估计这次所需的时间。
- 如果了解程序的长度和复杂度，了解以当前公司将程序长度和复杂度与测试时间关联起来的数据为基础的模型，则使用这种模型导出估计值。
- 如果了解与这个项目有关的风险，则估计针对这种风险测试需要什么（时间和任务），最终估计出整个产品的测试任务。
- 一些其他因素也会影响测试员的估计。例如，如果测试员知道程序员特别擅长这种应用程序，则他们的代码可能需要较少的测试。如果个别程序员引入的问题比往常多，则项目也许需要更多测试。如果已经编写了用户文档，或用户提供了清晰、详细的输入，则测试会进行得更快、更容易。

采用类似方法，测试经理可以猜测出项目进展中任何时刻的测试员人数。越到项目后期（掌握的信息越多），估计也就越准确。

承担工作的人应该告诉测试经理完成该任务需要多长时间

测试经理常常低估委派给其他人的任务。如果要使估计更有意义，可收集承担该任务的员工所做的估计并进行统计。如果测试员的估计比测试经理的估计长得多，先不要试图让测试员改变估计，而是尽力理解测试员对任务范围的看法，测试员还做了什么，以及还有什么因素使测试员得到看起来很高的估计。这可能是帮助测试员更明智地工作的一个绝好机会。也许测试经理要为测试员重新定义任务，也许测试经理不得不修正自己的估计。强迫测试员改变自己的估计可以得到较低的估计，但是任务还是需要那么长的时间，所制定的进度计划到后来才能符合实际。

讨论到这里，读者也许想听听Pat McGee很有见地的回答：

“将承担该工作的人应该告诉你需要多长时间……”我不太喜欢这种实践。我认为很多人都没有足够地注意为了给出合理的估计应该怎样

做。我曾经见过有的程序员估计要用两周做一件事，而经理认为不应该超过两天，而实际上只用一天半就完成了，并且还一次通过评审。但是我也见过实际使用时间要长得多的情况。我认为做出估计的合适人选应该是最注意应该花多长时间的人。有时这个人是经理，有时是实际完成工作的人员，有时谁也不是。不管这个人是谁，我认为最初的估计都会过于简化而到了没有用的程度。

我们之间的观点差别是，Pat建议由掌握最佳知识的人进行估计，我们建议由在估计有误时要承担责任的人进行估计（因此有得到最好估计的动力）。在我们看来，这两种方法都是合理的，都要比根据希望而进行估计这种更常见的方法好得多。

经验169

在测试员与开发人员之间没有正确的比例

有人常常问我们，测试员与其他开发人员的合适比例是多少。我们认为这个问题提得不对（Kaner等2000和Hendrickson 2001a）。

首先，什么是一比一的比例？对于不同的人可能有不同的含义。因此，在统计谁、什么时候开始统计、在将测试与其他工作比较时统计什么任务这些问题上，各个公司之间是有差别的。这使得不能在公司之间做这种比例的比较。

其次，这种比例关注的是自身，而不是所完成的工作。假设在最近的项目中，程序员花费16个人月设计并编写代码，测试员花费24个人月查找错误。比例（24比16，也就是3比2）是精确的，但是没有意义。改变新代码与第三方代码代码的比例，改变重用以前项目的代码量，改变对错误报告中错误定位的要求，使得测试比上一次更多（或更少）地介入程序错误分析定位，以及其他很多可变因素，上一个项目的比例都有可能不适用于当前项目。并不是不能讨论比例问题，而是要讨论必须做什么工作，以及每项工作需要多少人完成。

经验169

调整任务和不能胜任的人员

擅长测试数据质量的测试员，在计划高效测试各种配置上的产品时可能会感到困难。不同的测试员都有不同的强项。要鼓励测试员承担风险并扩展自己的能力，测试经理要尽可能提供指导，但是要关注测试员的进步。如果测试员完成某项任务时特别慢，或总也写不出有效的错误报告，这说明这个工作是不适

合他的（太困难，测试员不熟悉的领域的专业性太强，太枯燥，或与开发人员存在个人矛盾）。不要让测试员承担不适合的工作。在理想情况下，测试经理可以找到另一个愿意接替这个任务并放弃其他任务的测试员。机会总是有的，就在于你是否去发现。

轮换测试员的测试对象

不要让测试员自始至终对某个项目的同一组功能进行测试。首先，这会使大多数测试员感到厌烦。其次，一段时间之后会使得测试员太专，使这样的测试员的价值还不如经验不那么多的多面手。第三，如果手下的专家离开，会给测试小组留下很大的知识漏洞。第四，也是最重要的一点，两个测试员会以不同方式分析同样的功能。他们会使用不同的查错理论，创建不同的测试，并发现不同的缺陷。当一个测试员开始对程序的一个重要部件的质量有信心时，可把他调离当前的任务。所指派的新测试员会发现前一个测试员从来没有想到过的能够发现的缺陷。

Fran McKain建议对背景条件进行评估：

> 在这个问题上需要注意。有些功能组相当大，需要很长的时间了解相关的信息才能进行有效的测试。工作轮换过于频繁会使测试员的工作效果变差。

我们认同专业化的重要性，但是我们也担心如何管理专业化带来的风险。也许降低这种风险的一种方式是成对测试，让专家和测试小组的其他人结成工作对子。

尽量成对测试

测试员结成对子，一起工作，常常等于（或优于）熟练的查错高手。测试对子可以是稳定的（两个人一般在一起工作），也可以有相当高的流动性，就像在极端编程。在这种情况下，负责给定部分的测试员会寻找拥有与该部分有关技能和知识的短期伙伴。

成对测试与很多其他类型的成对工作不同，因为测试是一种思想生成活动，而不是计划实现活动。测试是一种在一种开放的多维空间中启发式搜索过程。成对测试有利于每个测试员解释思想，并对其做出反应。当一个测试员必须向

另一个测试员说明自己的思想时，简单的说明过程看起来能够把思想更好地集中，并很自然地启发更多思想。如果忠实地执行，我们相信这种过程会产生促进测试的更多、更好的思想。

我们强烈建议在开始测试之前，每个测试对子要先有个约定。离开计算机，花上5~10分钟（可能利用一下白板）时间考虑接下来的一两个小时的工作方向。例如，可以关注要调查的风险，预测要发现的问题，要测试的功能，或要使用的工具。这是一种总体指南，而不是测试用例的详细列表。测试员可以随便偏离约定，以探索新的机会（例如，跟踪刚刚注意到的有疑问的行为，但是该行为属于另一个功能）。但是，在这个方向前进了一段时间后，应该检查一下约定，以确定下一步该做什么。如果没有约定，我们认为测试对子有时会失去重点。

我们还认为成对测试有助于两个测试员始终关注任务，更容易重现和分析程序错误，并使一个测试员进行工作，而另一个测试员应付中间插入的情况，或离开主任务以抓住所需的东西。我们还认为其他人用小问题打扰他们的可能性也会更小。这种方式还会有更多的乐趣。

我们认为测试员成对工作与单独工作的效率至少一样。测试对子可以用更短的时间内完成更多的工作。这是一种新想法：尝试成对测试，看看在自己的公司中是否有效。可以选择对某些类的任务尝试成对测试，或指定一部分人结成对子，其他人还是单独工作。

为项目指派一位问题查找高手

问题查找高手是经验丰富、热心探索的测试员。以下是我们使用问题查找高手的一些方式：

- 对有怀疑的部分进行初步探索式测试，形成更细致地跟踪的想法，这可以让经验不足的测试员执行。
- 探索被认为是风险很低的部分——问题查找高手能够快速找到导致重新评估风险的程序错误吗？
- 定位看起来很容易引起不可重现问题的关键部分。
- 找出关键程序错误，以说服项目经理推迟（不成熟的）发布日期。

确定测试的阶段计划，特别是探索式测试的阶段计划

在操作计算机之前，测试员（或测试对子）应该很清楚自己要做什么，在接

下来的60~90分钟内要做什么。我们把这叫作阶段计划（session charter）。虽然我们还正在试验这种方法，但是这种方法看起来很有效。

这种方法的好处是有助于测试员集中注意力，避免在与工作无关的事情上分心。测试员并不锁定在这种计划上，如果发现值得怀疑的现象或有了很好的想法，测试员可以随便按新思路进行。但是当测试员净下心来时，应该检查自己的计划，如果没有明显更有吸引力或有更大压力的事情，则应该回到原定的计划上来。

我们认为当两个测试员结成对子在一起工作时，达成明确的计划特别有价值。

为了完成计划，测试员可能会指定详细项目大纲，选择一个一天或几天内可以完成的任务，也可以在白板上列出这个阶段的工作，或以后几个阶段的工作。

阶段目标可以包括要测试什么，要使用什么工具，使用什么测试战术，有什么风险，要寻找什么程序错误、要研究什么文档，需要什么结果等。

分阶段测试

一个阶段是一个不受干扰的时间块，长度为60~90分钟。在一个阶段中，测试员要进行专题测试。测试经理应该保护阶段的完整性——可以在测试员的隔板外挂上所有人（包括测试经理本人）都应该尊重的“请勿打扰”牌子，除非有严重问题需要紧急解决。

没有办法保证自己时间的测试员常常不得不间断式地工作，因为由于回答各种问题和参加会议使工作中断得太频繁。

通过活动日志来反映对测试员工作的干扰

如果认为不需要保护自己的测试免受频繁中断，可以记录一两周的活动。在日志中，列出每次电话交谈（以及电话所占用的时间），每次读突然来的电子邮件及回复，每次会议，每次有人伸进头问问题、说笑话或扔枚手雷。可以实际专心计划或执行测试的不被中断的最长时间是多少？在正常工作时间内，有没有合理的时间块？是否必须很早上班并加班到很晚，以便有不被中断的时间？（有关时间片段的进一步讨论，请参阅Weinberg 1992，第

284页。）

作为测试经理，为了解决看起来生产率有问题的测试员，了解需要大量加班才能完成任务的测试员的实际问题，活动日志是有助于将注意力集中到任务并对任务划分优先级（并且有助于测试经理了解怎样帮助测试员）的有效方法。我们不是建议读者把这种方法当作一种严格工具或正式的表现评估工具。对于人员问题，可以找公司的人力资源部征询与具体公司有关的建议。我们的建议是有时人们会遭遇挫折。作为遭遇了挫折的员工的经理，应该寻找一些疏导方法，帮助员工将注意力集中到现有的任务上，有效利用时间，清理不能完成的工作。而活动日志就是一种这样的工具。

定期状态报告是一种强有力的工具

测试小组的真正力量来自沟通，不是监管。测试经理要说服别人提供自己所需的资源，说服别人完成想让他们完成的工作，说服别人重新考虑把不能令人满意的产品交付给客户的问题。

状态报告是传递自己信息的很有用的工具。为了最大限度地发挥这种工具的效能，应该注意以下几点：

- 永远使用中性、客观的语气，不要使用感叹号、全大写词汇以及幽默。
- 不要针对具体的某人。只谈论可交付制品、程序错误和截止时间，但是要注意对事不对人。
- 采用所有项目都一致的格式。不要使（自己项目遇到麻烦的）项目经理感到测试经理以特别的方式对待自己的项目，因为测试经理不喜欢自己。
- 按照标准进度计划提交报告。在项目的早期，状态报告的频度可以是每两周一次。以后可以增加到每周一次。项目接近结束时，可能要增加到每天一次。对所有项目都要采用一样的报告频度。
- 仔细地选定状态报告的内容。状态报告要简练，要在几页纸中包含大量信息。
- 要把状态报告提交给项目团队之外的人看，要提交给项目经理的上司，也许还有项目经理上司的上司。要把状态报告送给公司中所有项目相关人员。项目经理可能会要求测试经理不要广为散发状态报告。我们建议用两句话回答这样的项目经理："我们已经把类似这样的状态报告送给这些人了"，"如果状态报告对于X不合适，可直接让他告诉我们他不需要这些状态报告。

我们很愿意按照其愿望停止向其提交状态报告。”[⊖]

经验198

再也没有比副总裁掌握统计数据更危险的了

在报告状态（或进行其他度量）时，应该知道自己在统计什么数据，谁将看到这些数据。具体地说，高层经理很可能利用这些自己还不理解的数据做出决策。根据定义，度量就是整个图片的一小片。将整个图片压缩为少量数据的度量是一种很大的简化。如果测试经理了解度量的条件背景，就有希望有效地利用度量。而高层经理往往并不知道这些条件背景。

我们鼓励读者进行度量实验。我们利用度量帮助了解项目的进展情况，以及产品各个方面的质量情况。根据我们的经验，执行经理有些令人担心，因为他们利用指标不是为了了解情况，而主要是为了对他们并不理解的东西进行武断的控制。

这是否意味着永远也不应该向执行经理提供数据？在很多公司中这并不现实。但是测试经理可以更谨慎地提供数据，可以预测会带来的误解并直接提出问题，并拒绝提供某些数据。例如：

- 不要跑到执行经理的办公室，抱怨程序员修复一个程序问题平均需要1.4周，而Joe用了5.3周。如果测试经理主动提供了根据程序错误跟踪系统得出的个人表现数据，以后会被要求大量提供这种信息。
- 如果提供的备忘录说，到交付至少还需要5周，因为还有200个程序问题没有解决，而编程小组每周平均只能解决40个问题，这时应该在数字旁加上注释。也许应该指出这些数据是大量程序问题的统计结果，但是如果没有解决的程序问题数目很少，很多其他项目因素对确定产品交付时间会比问题的数目产生更大的影响。

⊖ 本书的有些评审人建议，在他们的公司中这最后一点可能是危险的，并且会降低生产率。我们知道在一些公司中，信息流有更多限制，经理有更多的权力控制信息流。另一方面，如果测试经理不能使项目相关人员了解测试小组发现的问题，测试小组的有效性就会打折扣。也许以下建议会有帮助：

- 如果只是把测试状态报告提交给项目团队，可在公司文化氛围允许的范围内，尽可能广地传阅。利用非正式渠道使更多的人了解有这些报告，让这些人要求项目经理将自己列入传阅名单中。最终，如果测试状态报告是有用的，就会建立一个能够在不同项目中使用的传阅名单。到这个时候，广泛传阅状态报告就会成为公司实践。只有到这个时候测试经理才能对项目经理说：“我们一直向这些人提供状态报告。”
- 如果只向项目经理提供服务，没有责任为公司其他人提供服务，那么就没有理由为项目经理之外的人提供状态报告。另一方面，如果测试经理要承担不向公司其他人报告关键问题的责任，那么就需要向这些人提供状态报告的公开渠道。

- 当被要求提供个人统计数据时（例如每个测试员发现的程序错误数），可拒绝。如果出现这种情况可以向他们解释，一旦利用程序错误跟踪系统收集人力资源数据，数据库的性质就会发生变化。错误报告过程会变得越来越行政管理化，越来越受到抵触，准确性也越来越低。

要小心通过程序错误数度量项目进展

程序错误数是一种度量项目进展的很好参数。但是，程序错误数是不充分的，并且常常产生误导。

程序错误数可以用来说明项目离发布日期还很远。如果项目团队每周平均可以解决40个程序问题，应该在发布之前解决400个未解决的程序问题，可以有说服力地说，产品在下个月内不能交付。

程序错误数不能用来说明产品已经接近发布时所要求的质量。如果到了项目的最后阶段，未解决的程序错误数已经降低到接近要求的水平，这是意味着产品更稳定，还是测试小组花了太多的时间编写错误报告、运行回归测试（很少发现新程序错误）、在展示会上展示产品，以及不直接导致发现新程序错误的其他活动？从程序错误数中不能得到这些问题的答案。

我们特别不赞同把程序错误到达率（每单位时间会发现多少程序错误）作为项目管理依据的统计模型，因为我们没有理由相信这种方法核心的概率模型假设符合项目的实际情况。Simmonds（2000）提出了清晰、明确地描述这类模型之一的假设。

Hoffman（2000）给出了一些非常好的度量失败的例子，其中就包括这种度量。请参阅Austin（1996）和Kaner（2000a）。

使用的覆盖率度量越独立，了解的信息越多

可以在多个元素上度量产品已经完成的测试量：程序错误、需求、代码、配置、变更历史、开发人员、测试员、数据等。单独任何一个元素都是不够的。

例如，一种经常推荐的度量是测试所执行过的代码行百分比（语句行加分支，或几种其他考虑因素）。

与程序错误数一样，代码行度量在说明测试的充分性或完备性上是有用的。如果只测试10%的代码行，那么对软件的质量就不应该有多大把握。如果测试了

接近100%的代码行，这并不说明产品已经差不多可以发布，而只能说明根据这种度量产品离发布已经不远了。（有关这种度量的进一步谈论，请参阅Marick 1999。）

假设正在开发一个瞄准大众市场的产品，决定在10多种Windows变种上测试，使用10多个浏览器版本，10种不同连接（不同速率的调制解调器、以太网等），以及另外一些其他配置。可以把这些归纳为这个程序必须兼容的100种标准配置，并在所有这100种配置上测试该程序。

- 在头几个配置上测试了该产品之后，在其他配置上可能只多运行几行程序，或执行少量不同的分支。如果认为代码行是评估程序测试充分性的度量，则这种情况表明应该停止配置测试了。但是对于大众市场来说，这种决定可能是灾难性的。
- 如果有100种标准配置，已经测试了其中的30种，那么有一种独立于代码行但是非常重要的覆盖率度量，即配置测试的覆盖率达到30%。

覆盖率度量涉及给定类型可能的测试总集合、计划运行的测试总集合的子集，以及实际运行的测试子集。可以把已经执行的测试数与计划运行的测试数的比值作为覆盖率度量，也可以把已经执行的测试数与有意义的测试总数的比值作为覆盖率度量。两种百分比都很有用。覆盖率度量都只限于一类因素，不管是所执行的代码行，还是所实验过的配置，还是所检查过的数据流或经典风险（例如除零问题——没有代码预防这种除法出现的问题，这种遗漏在代码行计数中反映不出来），还是其他因素。读者还可以想出几百种可能的因素，每类可能的失效都会对应一种因素（Kaner 1995a和2000a）[⊖]。

经验201

利用综合计分牌产生考虑多个因素的状态报告

综合计分牌常常用于度量企业是否健康发展（Kaplan和Norton 1996，并请参阅Austin 1996）。比较简单的指标是不够的。

- 例如，如果将重点放在季度利润上，就有可能鼓励最大化季度利润的行为。短期内可以通过解雇研发（R&D）人员使季度利润大增，但是由于没有新产品可卖，公司在连续两年获得很大利润后，接着就会破产。

⊖ 即使这样也是做了很大简化。本书的一位评审Noel Nyman解释说："这意味着所有测试都有一个单位值，这可能不对。就像程序错误可以有不同的严重程度一样，测试也有不同的价值和不同的运行时间。我们可能运行了80%的测试，只完成所需的20%的测试小时；也可能完成了80%的测试，但是仍然有50%最重要的测试还没有执行。忽略这一点而给出'覆盖完成率'度量的任何指标都会给出虚假信息，通常使测试员产生虚假幻想。"

- 也可以统计新发布的专利数，不过也有可能在研究上的投入过度，而在可销售产品上的投入不足。

多个不同的数字看起来可以比较好地说明企业的状态，但是任何一个数字孤立地看都很容易产生严重的副作用。度量的一个关键点是鼓励员工在所度量的方面做得更好，因此希望做好工作的员工会想方设法提高度量值。他们使用的方法可能不是指标设计者曾经想到过的。例如，削减研究人员以提高短期效益，可能就是倡导提高季度效益的人所没有想到的。但是这种方法忠实地响应了提高季度效益的要求。

针对度量副作用问题的一种解决方案，是利用能够抵消副作用的多个参数。例如，报告短期效益、专利发布数和人员跳槽率的综合计分，很快就反映出很高短期收益之外的研发失败问题。

能够很好平衡的计分牌可以是企业健康状况的有效度量，即使这些计分因素单独考虑都不有效或不保险。

我们认为同样的推理也可以用于项目进展和测试进展度量。我们可以把进展（或停滞）报告看作涉及不同的因素，例如：

- 产品已经完成了多少测试？
- 计划进行的测试已经完成了多少？
- 发现了多少问题？其中有多少问题还没有解决？
- 我们对测试质量有多大把握（例如，如果β测试员发现以前的测试员遗漏的很明显的错误，那么对质量的把握就比较低）？
- 由于未解决缺陷、或缺乏设备，或没有做出决策，有多少测试工作受到阻碍？

我们所看到的最好的状态报告使用了反映以上各个因素的参数。不是通过单一、平凡的数字向管理层提供信息，但是信息模式的表示要足够简单，以便指导决策。

以下是周状态报告的推荐结构

可以把状态报告看作四页长的文档。第一页列出关键问题，例如：

- 所需的决策。（例如，应该如何确定这些功能的优先级？测试什么设备？是否吸收新测试小组成员？）
- 所需的程序错误修改。（对测试工作产生影响的所有程序错误，都应该优先解决。）
- 预期的交付的工作制品（例如承诺交付的文档、设备、功能和工具）以及

承诺的交付日期。在截止日期之前一点就要把这些工作制品列在清单上，过期没有交付的工作制品要留在清单上，删除已经提交的工作制品。这种清单表示遗漏的东西，而不是累积工作表。突出显示阻碍工作进展的因素，例如无法访问承诺提供的计算机或没有完成和交付功能清单。

- 意外问题。（例如，报告由于员工跳槽引起的某部分测试效率降低，某种关键工具不像预想的那样好用，员工需要培训等）。

第二页描述测试小组完成计划任务进展的情况。例如，可以列出不同测试工作进度计划，每项测试工作的预算（例如两周），每项测试任务的完成情况（例如完成了10%），每项测试工作使用的时间。在这个例子中，如果所使用的时间超过了计划两周的10%以上，那么这部分工作的进展就会落后进度计划。通过浏览这个清单，就可以得到进展（或停滞）对比计划和进度的总体印象。

第三页提供错误报告统计数字。报纸编辑都把体育部分放在报纸的内页，因为体育部分是被最多读者阅读的部分。同样，我们也把程序错误数放在报告的中间页。如果把程序错误数放在内页，就可以把自己认为很重要的新闻放在第一页，并在其他页中塞入其他广告（或要通报的其他内容）。

最后一页列出本周被推延的程序错误。清单可以只包括程序错误数、总计（或标题）行，以及测试员为该程序错误划定的严重程度。需要更多信息的读者可以进一步查找。

经验203

项目进展表是另一种展示状态的有用方法

进展表是一种图表，画在会议室的大白板上，向项目团队所有成员和与该项目有关的任何人公开。进展表直观地展示项目的进展情况。（有关进一步讨论与说明，请参阅Bach 1999b。）

典型的进展表会列出多个工作区，每个工作区对应一行。如果要使字体足够大，以便于阅读，那么即使是很大的白板也不会容下太多的工作区。对于每个工作区，白板显示当前工作量水平、这部分工作区的计划覆盖率、目前已经达到的覆盖率、测试员对质量的评估（分为高、中、低）。进展表还留出空白，使测试员能够补充一些关键注释，以便进行评估。

典型的进展表每周更新一次（在项目初期），当项目接近尾声时，可增加到每天更新一次，甚至两次。进展表提供的信息要足够新，使经过的经理也希望看一看。

各个公司的项目进展表细节各不相同。例如，有些公司需要每个工作区的其

他具体信息（其他列），或留出一两行可变行，用于说明本周的问题或表现状况。

进展表的具体形式无关紧要，关键是要使别人一眼就能看出项目的状态。

测试进展表				更新日期：2月21日 版本号：38
工作区	工作量	覆盖率	质　量	注　释
文件/编辑	高	1	☺	
视图	低	1+	😐	1345，1363，1401
插入	低	2	☺	
格式	低	2+	😐	自动化测试有问题
工具	中断	1	☹	崩溃：1406，1407
幻灯片	低	2	☹	动画内存泄漏
在线帮助	中断	0		没有交付新文件
标签	无	1	😐	测试需要帮助……
转换器	无	1	😐	测试需要帮助……
安装	3月17日开始	0		
兼容性	3月17日开始	0		已确定实验室测试时间
通用GUI	低	3	☺	

如果里程碑定义得很好，里程碑报告很有用

有些公司通过里程碑推动其项目，并在每个里程碑处对状态进行全面评估。有些公司没有采用这种方法。

如果公司要在每个里程碑处评估产品，那么测试经理需要知道应对照什么进行评估。公司怎样定义里程碑？产品需要与里程碑定义中的哪些方面进行比较？例如，如果里程碑定义说50%的功能已经完成编码，则应该判断产品是否达到要求，怎么知道是否达到要求了呢？

如果公司没有里程碑的标准定义，就不能很肯定地做出决策，例如产品已经能够进入β测试阶段。没有公司标准的判断更容易引起管理方面的不同意见，而不是认识上的不同。

如果测试经理要评估与里程碑相关的进展，并且没有可用的里程碑定义，我们建议测试经理首先得到有关高层经理对里程碑定义的认同，然后再根据里程碑定义进行评估。如果高层经理征询测试经理有关定义里程碑的建议，那么应建议他们把里程碑看作是一个项目迭代的完成。这个迭代的退出准则是什么？（或下一个迭代的进入准则是什么？）Lawrence和Johnson（1998）的研究工作

对解决这个问题很有帮助，他们描述得很细致，并且可以免费得到[⊖]。

不要签署批准产品的发布

要让项目经理或项目团队确定什么时候发布产品。测试经理的工作是使项目经理能够得到可以做出这种决策的最好的数据（测试员所能够提供的最好数据）。忠实、直接、准确地向所有项目相关人员报告测试发现。如果他们做出违背测试经理意愿的产品发布决定，则这是他们的权力。

不要签字承认产品经过测试达到测试经理的满意程度

如果有人坚持要测试经理签署产品发布表（授权项目经理把产品投入生产，或投入正式使用），要声明自己的签字只说明（在测试经理看来）产品已经经过充分测试，已经完成了约定程度的测试，或测试小组在可用的时间内尽力做了最好的测试。测试经理可以通过备忘录（向项目经理或项目团队提供）或在签署发布表时做简短声明。

如果测试经理要编写产品发布报告，应描述测试工作和结果，而不是自己对该产品的看法

测试小组对产品的整体质量或产品对目标市场的适合性做出评估是非常困难的。测试经理并没有相关的数据，他只掌握错误报告和发现程序错误的尝试（产品通过了的测试）。测试经理应只描述自己所知道的东西。

在产品最终发布报告中列出没有排除的程序错误

如果测试经理准备（或帮助准备）产品最终发布报告，应该附上产品中未排除的程序错误的清单。还应该在清单中列出自认为重要的被拒绝了的设计问题。事先传阅这份清单，以免最终报告使大家感到意外。

⊖ 多个项目里程碑详细定义的有用例子，请参阅www.coyotevalley.com/plc/builder.htm。我们并不能保证这些定义是“正确”定义，也不认为Lawrence或Johnson会把这些定义看作是“正确”定义。相反，他们说明了可以被用来定义里程碑的准则，这些定义对于正在创建自己里程碑定义的公司来说，是很好的切入点。

有用的发布报告应列出批评者可能指出的10个最糟糕的问题

如果要向代理商提供软件，可以考虑列出有可能被不留情面的杂志评论员提出的10大问题。如果这些问题很严重，那么公司的市场联络员（或其他市场开发人员）会暂停提供该版本。这也许是测试经理所能提供的最有用的质量评估。（请注意，这样做能使测试经理维持自己作为软件批评者的地位，而不是试图对软件做出公正的评估。）

第9章

测试小组的管理

本章集中介绍测试小组管理方面面对的挑战。本章与第8章之间的差别在于，第8章关注的是一个项目的问题，即测试经理如何与项目开发团队和其他项目相关人员协同工作，以帮助在预算内按时拿出合适的产品。本章考虑的是一个项目接一个项目、年复一年地一起工作的员工组成的小组。

从我们个人喜欢的经验中挑出这部分经验很困难，因为我们三人都有丰富的管理经验。我们都经营自己的公司，都想就技术人员管理问题进行很长的一般讨论。我们不打算这样做，建议读者参考以下文献：Weinberg（1992，1997a，1997b，1997c，1998和2001）、Humphrey（1997）、Deming（1986）、DeMarco（1997）、DeMarco和Lister（1999）、Brooks（1995）、Constantine（1995）、Black（1999）、Drucker（1985）、Wiegers（1996）。这里只提供一般经验，特别是有关招募方面的经验。我们在这里讨论这些问题，是因为有太多的人问我们有关招募和找工作的问题，以至于我们认为如果不涉及这些问题，测试小组管理的讨论就是不完整的。

经验210

平庸是一种保守期望

- 如果测试经理通过抽取测试过程的人性化因素来扼杀员工的创造性；
- 如果测试经理的目标是使自己的员工成为不能互换的齿轮；
- 如果测试经理要标准化自己员工的工作，使得每个人都确切地知道他们能够做什么，并以同样的方式做；
- 如果测试经理通过并不反映其创造性、可说服性、判断力或人际敏感性的数字评价自己的员工；

- 如果测试经理将自己的员工与其工作结果剥离，告诉他们其工作是报告程序错误，由别人在不需要他们进一步帮助的情况下决定要修改什么，阻止他们在程序错误分析会上对所发现的程序错误发表意见；
- 则在项目最需要他们的时候，不要指望他们能与测试经理同心协力；
- 不要指望他们在原则问题上发表意见；
- 在项目需要突击时，不要指望他们能够很高兴地通宵加班；
- 不要指望他们为困难而关键程序错误问题而设计漂亮的测试；
- 不要指望他们为了苦苦思索如何向仍然不能理解的经理解释某个关键问题而失眠……
- 在测试经理最需要他们的时候，不要指望他们能够表现出对测试经理和公司的忠诚。

如果认为在自己的公司内不会出现英雄，就不会得到英雄[⊖]。

经验211

要把自己的员工当作执行经理

Peter Drucker被很多人看作是现代管理理论的创始人。他在一本出色的专著《有效的执行经理》（The Effective Executive）中指出，执行经理就是管理自己的时间以及影响机构发挥作用的人。大多数知识工人都是执行经理。Drucker指出，人们交给执行经理的工作量，总是比其所能够完成的要多。有效的执行经理会挑出自己能够完成好的那一部分任务子集，并完全不考虑其他任务。低效的执行经理总是试图做所有的事，总是不成功，特别是不能取得自己所努力争取的结果。不同的执行经理都有不同的强项和兴趣。把一样的工作（不是任务，而是正在进行的工作）交给两个执行经理，他们的表现会有很大不同。

根据Drucker的观点，大多数测试员都是执行经理。要以此为基础对测试员进行管理。对测试员不要像对工厂工人那样地发号施令，也不要白费时间把测试员的工作重新设计为与工厂工作等价的工作。相反，应该接受测试员具有不同的强项和兴趣这种事实，并有针对性地进行管理。

以上并不是专指高级测试员。当成立或管理测试小组时，要录用具有不同经历和背景的员工。新测试员需要管理层更多地关注。应该在培训初级员工方面投入。但是我们的目标是产生执行经理，并且从第一天开始就要表现出对其学习、思考的尊重。我们期望他们能够迅速担起责任，表现出工作热情，建立自己的信誉，发展自己的技能。如果他们想要要求比较低的工作，那还是另谋高就吧。

⊖ 所谓英雄在这里的含义，请参阅Sims和Manz（1996）及Lebow（1990）。

阅读自己员工完成的错误报告

有些测试经理或测试管理人员阅读自己员工完成的所有错误报告（对于由7人组成的测试小组，这很容易，但是对于由70人组成的测试团队，做到这一点要困难得多）。这有助于测试经理了解产品情况、自己员工的强项和品德，以及影响自己员工的沟通和人际问题。

为了评估自己员工的工作质量，以下是针对测试员的错误报告可能提出的一些问题：

- 这些报告写得好吗？
- 报告直率地提出了问题吗？
- 报告留下迫切要求后续测试的漏洞了吗？
- 发现程序错误的测试看起来例行公事还是很有见地？
- 程序错误很难发现吗？程序错误出现在应用程序一般比较稳定的部分吗？如果是，这反映出测试员的坚韧性，还是好运气？
- 报告的语气怎么样？
- 程序员能理解报告吗？程序员对报告有什么反馈意见？这反映出程序员和测试员良好的协作，还是相互指责？

像评估执行经理那样评估测试员

程序错误数顶多就是测试员工作有效性的没有价值的度量。程序错误数会产生误导，减弱推动力，会使测试经理自认为所掌握的比其实际知道的多很多。在最坏情况下，会把做得差的测试员看作是做得好的，而把最好的测试员看作是能力不足。

程序错误数所以能够成为一种度量，是因为经理和执行经理认为他们需要某种方式评估员工工作的有效性，除了程序错误数之外，他们不知道还可以使用什么。我们并没有简单的指标可以推荐，但是我们可以建议评价员工工作时可以进行以下工作：

- 阅读其错误报告。
- 阅读其编写的代码。
- 阅读其编写的测试文档。
- 收集与其一起工作的程序员和其他有关人员的意见。

可考虑以下因素：

- 他卷入了什么争端，为什么？
- 在按期完成任务方面做得怎么样？
- 在信守自己的诺言方面做得怎么样？
- 他遗漏了什么类型的问题？
- 他对其他测试员和程序员提供了什么类型的帮助，以提高他们工作的有效性和生产率？
- 他在学习新技能吗？他在把自己的技能传授给其他测试员方面做得怎么样？
- 他站在公司的立场上处理过什么问题？这些在其对公司业务的判断和个人道德上是如何体现的？

通过这些工作采集到的信息，会使测试经理对员工的工作和所测试的项目状况有全面的认识。这些信息并不能压缩成简单的数字，但是也许能够利用这些信息建立报告卡。假设在每个方面（错误报告、代码等）给测试员打0~5分。最终的得分卡会显示出一种强项和弱点模式（与测试经理的感觉一样）。类似这样的多元素信息，有助于测试经理为每位测试员制定出培训和指导计划。

很多公司都把自我评估作为正式表现评估的一部分利用，并对评估过程进行投入。（在自我评估中，员工填写表格以对自己的表现进行评估。）这些方法会非常有用，但是有些经理过于依赖自我评估，没有花足够的时间更多地了解员工实际完成的工作。我们的另一种担心是正式评估的频度很低——每半年或一年进行一次。我们鼓励测试经理积极、随时地跟踪员工的工作。（有关进一步讨论，请参阅Deming 1986，第102页。）请注意，我们并不是在建议进行微观管理：了解自己员工做什么并据此进行指导，与告诉他们什么时候以及怎样完成自己的工作有很大不同（Drucker 1985）。

经验214

如果测试经理确实想知道实际情况，可与员工一起测试

测试经理可能经常听到这样的说法，“好的经理会派活。”当然，测试经理必须分配小组的大部分工作，并且可能在任何项目中都没有充当主要的有用角色。但是，如果测试经理不参加项目团队，一段时间之后就会丧失自己的很多技术能力，就会看不到测试员正在处理的真正问题，并且很难评估测试员的工作质量。测试经理参加项目团队可能并不实际，但是可以对自己的时间划分优先级，可以仔细考虑一次至少积极地参加一个项目团队的可能性，这项工作很重要，值得给出较高优先级。

如果测试经理甚至不能安装或运行测试员正在测试的软件，就会丧失可信性。我们殷切希望测试经理能够重现一些在产品中所发现的程序错误，要有足够的被测产品背景知识，使得测试员能够与自己深入地讨论该产品。

不要指望别人能够高效处理多个项目

如果测试项目比测试员多，测试经理往往想为每位测试员指派多个项目。请注意两个问题：

- 有些人会接受多项任务，但是在特定的一周（或一个月）内，他们只能承担这些工作中的一项，而其他任务会被抛在一边。
- 积极地承担所有被分配的任务的测试员会把时间分得很碎，要在管理方面花很多时间。最终，测试经理使某个测试员参与多个项目，参加很多会议，花大量时间了解各个项目（重新阅读笔记，核对数据库中的最新程序错误，阅读新功能报告，阅读大量电子邮件，等等），而对任何一个项目所做的实际测试都很少（DeMarco和Lister 1999）。

积累自己员工的专业领域知识

随着员工对制约产品设计的外部因素、用户如何使用（或将使用）类似的产品、什么样的问题对他们很重要、竞争对手如何解决的这些问题、关于使用这类产品都有哪些文章等了解得更多，他们工作的有效性会显著提高。

可以通过多种途径积累真正的专业领域知识：

- 阅读面向类似产品客户的杂志和书籍。
- 参加教授客户如何使用类似本公司正在开发产品的产品学习班（最好参加由非本公司员工授课的学习班）。
- 参加与产品的下层主题有关的学习班。例如，如果要销售房地产程序，就要学习如何成为房地产代理商、抵押贷款代理商或评估师。
- 在客户现场工作。可以采用全日派出的形式（例如，在安装和培训软件产品期间，公司可以向重要客户派出员工工作几周），也可以采用兼职的形式（必须征得公司的同意）。
- 推销自己公司或竞争对手的软件。例如，我们中的Kaner在客户的软件公司担任软件开发经理期间，花了半年的时间，利用星期六到最大的零售商

店推销软件，掌握了产品市场的很多信息。产品发行商和零售商都明白他的目标并鼓励他的这种实践。

- 每周利用几个小时回答公司技术支持热线电话。最初要在技术支持人员的指导下进行，最终能熟练解答问题，并且会了解到很多信息（如果能够认真对待这种工作，解答问题，分析研究客户对产品的一些批评）。

积累自己员工相关技术方面的专门知识

随着硬件和软件环境变得越来越复杂，自己的应用程序越来越多的问题会变成与其他应用程序、远程服务器或其他不受公司应用程序开发人员直接控制的软件或硬件的交互问题。Nguyen（2000）向测试员介绍如何测试Web应用程序的交互问题。为了成功地发现交互问题，测试员必须对其他硬件和软件有很多了解。（他提供了很多介绍材料。）

此外，如果部分员工理解程序员所使用的技术，则测试小组都会受益。例如，如果测试小组的一些成员对程序员所使用的组件库有很多了解，并知道如何编写程序以通过组件的API（应用编程接口）来测试组件，这会对测试很有好处。

积极提高技能

测试经理不要只是说："今天要掌握因果图测试法。"如果程序错误查找、错误报告或程序错误评估手段值得学习，则也许不得不学习一些子任务，并多次实践。由于技术不断变化、竞争市场的变化、开发工具的变化以及来自方方面面的产品，提高自己的技能会变得越来越困难。尽管并不容易，也必须做这件事，以免落伍，这对于强化测试小组在公司中的价值极有帮助。

浏览技术支持日志

为了理解软件安装之后应该正常运行时出现的问题，应该阅读公司的客户投诉记录（客户打来的电话记录，以及投诉信和电子邮件），思考为了使某类投诉减少自己可以做些什么；如果有这样的问题，思考什么类型的测试会有助于发现被测软件中类似的一般问题。

帮助新测试员获得成功

当录用测试员后，通过几个步骤使其头几周的工作能够成功：

- 为新测试员找好地方（办公室或小隔间），并在他上班之前安排好（提供两台计算机和其他所需的硬件和软件，以及必要的公司和项目文档）。
- 至少要安排一天的时间让新测试员与有关人员见面，带他参观公司部门，向他介绍将在一起工作的每一个人，通过面谈了解他的希望和预期。
- 为新测试员指派一位指导者。指导者要带新测试员各处参观，并回答各种问题，检查新测试员所完成的工作并提出建议，午餐请几次客，这些都会很有帮助。指导者并不是新测试员的上级，并不向测试经理报告有关新测试员的任何事，除非是很重要的问题。有些公司从新员工的部门之外指派指导者（例如，测试员的指导者可能是程序员或市场开发员）。这样做的优点是便于各个部门之间的沟通，缺点是指导者可能难以对测试员的工作技术细节进行指导。

让新测试员对照软件核对文档

所谓新测试员，是指新接触这个项目的测试员，可能有也可能没有其他软件的测试经验。

让新测试员了解产品的一种方式，就是让其针对软件的行为测试手册和在线帮助。在这样的测试结束之后，新测试员就会了解程序的所有部分。

全面测试手册和在线帮助是很值得的。当用户需要关于软件的权威信息或遇到问题时，就会查阅手册和在线帮助。此外，对于大众产品，文档要保证产品的行为与所描述的一致。这种基于制造商实际书面声明的保证，不能通过软件公司所使用的固定套话合约所取消（废除）。文档所叙述的任何内容都可以被用作举报产品有缺陷的证据（Kaner 1995b、Kaner和Pels 1998）。

测试员应该核对所有事实描述（任何可以证明真或假的陈述）和程序测试。此外，测试员还应该测试自认为可以通过文档合理地引出的所有推论。这可以为新测试员提供很好的培训和发现重要程序错误的机会（Pettichord 2001b）。

通过正面测试使新测试员熟悉产品

当新测试员核对完用户文档，并开始理解产品应该提供什么功能之后，应该

让其尝试以简单但是实际的方式使用该产品。让其从头安装产品，可能还要在新系统上安装。让其列出使用类似的产品有可能做的事，然后使用被测产品，尝试简单地照样做。

这样做的目的，是帮助新测试员理解该产品或产品版本的优点。有了这种体验之后，在开始寻找失效而不是成功时，就会成为更见多识广、更有效的批评者。

让测试新手在编写新错误报告之前，先改写老的错误报告

测试新手在这里指初次接触测试，并不只是刚加入课题组。新测试员必须学习测试基本技能，而不只是如何完成这个测试项目。

指导新测试员的一种方法，是让其重新测试老程序错误，并改写出更有效的错误报告（供测试经理审查）。这可能会涉及改写和后续测试（查找与这个程序错误有关的更严重的问题，或查找会出现所报告问题的范围更广的条件环境）。测试经理最后对后续测试错误报告中的语气、清晰性和创造性给出反馈。

如果测试经理对新测试员编写出的合理的错误报告感到满意，就可以让其编写新的错误报告了。在新测试员得到测试经理的批准之前，也许不应该允许其编写新的错误报告。

让新测试员在测试新程序错误之前，先重新测试老程序错误

使新测试员熟悉数据库中的程序错误的一种有用方法，就是让其重新测试老程序错误。我们利用三类任务，这些任务可以帮助测试员了解被测产品、产品会怎样失效、可以怎样测试产品、数据库中有哪些程序错误。

重现现在还没有关闭的程序错误。对于每个被测试的程序错误，新测试员都要报告该程序错误仍然能够重现，并标出报告日期、软件当前版本标识号、测试使用的硬件和软件配置。新测试员也可能报告该程序错误不能重现，同样也给出日期、版本和配置信息。

重新测试已经解决的程序错误。对于所选的程序错误，新测试员要核对曾经解决的程序错误是否又出现。这些程序错误也许已经关闭。在大多数公司中，除非程序错误重现，否则不留程序错误记录，在这种情况下（取决于公司策略），测试员会重新打开并更新现有错误报告，或存档交叉引用老报告但是描述当前错误行为的新报告。

重新测试已经解决但还没有关闭的程序错误。程序错误已经返回——解决、推迟或不可重现，或以其他方式拒绝。报告该程序错误的测试员要重新测试，并可能关闭。也要让新测试员调查该程序错误。经过调查之后，让新测试员与报告并重新测试该程序错误的人交谈。这个测试员和测试经理可对新测试员所使用的方法、定位策略，以及重新测试给定程序错误投入多少时间和精力等提出反馈意见。如果测试经理感到新测试员具有一定的判断水平和错误定位技能，就可以让其关闭报告。

不要派测试新手参加几乎完成的项目

如果测试经理要为某个接近完工的项目团队增加员工，则不要指派新手。有经验的测试员已经知道如何阅读测试矩阵、边界图以及其他一般测试文档（或很快学会）。只需要告诉这样的测试员要测试什么，不需要声明每个测试怎么做。测试员会需要时间了解产品，但是不需要占用别人太多时间就能很快提高测试效率。

但是对于测试新手，测试经理需要给出多得多的指示。有些测试小组要花大量时间编写测试详细步骤指示、供测试新手使用。我们认为这些测试小组最好还是把时间用在发现程序错误、使问题得到解决上。我们还认为使用这类指示的测试新手会遗漏大量程序错误，因为他们不知道要查找的缺陷范围，并且书面指示并不能传递必要的观察范围信息和解释技能。

如果有时间进行培训，或只需要测试经理很少时间就可以准备和安排测试新手任务，确实要录用新手。

如果测试经理不能安排测试新手的培训，则不要录用新手。不要为了能够让测试新手能够开展一般工作而浪费大量高级测试员时间的方式，来构造自己的测试设计和文档活动。

员工的士气是一种重要资产

Napoleon指出，“三分士气一分体力”。

认为自己的工作很重要，如果自己申请就可以完成任务，而且所完成的工作质量很高，这样的员工会取得显著成就。而这些员工的测试经理是其士气最重要的保护人：

- 礼貌地对待员工，尊重员工。
- 注意他们的工作。
- 称赞好的工作、热心和诚实努力。
- 如果员工加班，测试经理也要加班。不一定每晚，不一定每个周末，但是要足够经常，使得员工可以感到测试经理在观察他们加班情况。
- 如果可能，为员工指派他们感兴趣的任务和项目。鼓励他们说出自己的兴趣，并给予考虑。
- 如果测试员任务完成得不够顺利，可指派别人给予帮助、指导，如果有必要还可以接替（让其承担其他任务）。
- 提供培训机会。表现出测试经理很看重技能和专业发展。
- 测试经理要公平对待员工，并且要求别人也公平对待他们。
- 不要对任何一位员工产生误导。如果测试经理不知道问题的答案，就实话实说。如果只是凭感觉，则应明确表明自己是在猜测，而不是根据知识。
- 不要对员工叫喊，不要利用自己的权力强制别人接受自己的观点。
- 避免公开批评员工，但是必须在私下指出其错误和问题。
- 不要与员工私下议论其他小组内的员工。
- 测试经理不要与员工约会，并在接受员工个人提供的方便或礼物时应该特别小心，即使是小礼物。给的东西要多于接受的东西。

测试经理不要让自己被滥用

项目经理和执行经理有时会提出不合理的要求。

不要做自己做不了的事。很多经理都按惯例试图为每位员工再增加10%~20%的工作，不管员工已经多么努力工作。测试经理对于自己做不到的，也不要陷入责任圈套，不要为了过度工作而搞垮自己和整个团队。

软件开发并不总是一种朝九晚五的工作。测试经理应该支持自己的小组，有时这可能需要几天或几周的大量加班。有时项目经理会需要测试经理帮助使出现偏离的项目回到正轨，或使项目按时完成。如果测试经理能够提供帮助，一定要想方设法尽量提供。但是要选择自己的节奏，并做出自己的承诺。测试经理是志愿者，而不是牺牲者[⊖]。

⊖ 不要使自己陷入只能导致失败的工作上，不要在明显没有希望的项目上投入太长的时间。这样做对谁都没有好处，还会使自己的情绪受到伤害。请参阅Yourdon（1997）。

测试经理不要承诺不可能做到的事，在这样的事上不必说谎，不必掩盖问题。事实上，测试员工作的本质就是暴露问题，而不是将其掩盖起来。不必，也永远不应该在诚实上让步。

测试经理的力量来自向需要了解真相的人说明真相。如果在诚实上让步，就会使自己力量的一个核心支柱弱化。

有些老板不太讲理并滥用员工，测试经理不能改变这样的人。如果工作难以忍受，可以在找到更好的工作后离开。

不要随意让员工加班

很多测试小组都长期加班。在有些公司中，测试员的工作要比其他开发小组的工作紧张得多。

长期加班会使员工体力透支，跳槽率升高，不信任别人，关系复杂，低效，工作草率。我们强烈建议测试经理要避免让测试员长期加班。

很多项目在关键时刻都需要短期突击，这也是测试专业的正常情况。但是这种情况并不是我们所说的滥用加班。不过请注意，每个员工都有不同的极限。有些员工有家庭负担，或其他不太容易灵活处理的负担。测试经理需要妥善地处理好这种实际情况。

有些公司或项目经理，长期过紧地计划项目，并期望自己的员工能够弥补进度缺口：

- 在制定项目进度计划时，不要指望员工能够每天都8个小时集中在工作上。这不现实。测试员要参加会议，接受培训，要编写状态报告和备忘录，填写公司人力资源部门要求的繁琐表格。如果幸运的话，测试员可以每天有6个小时专心完成办公室工作，其余时间用于完成其他必须做的工作。
- 不要同意自己知道的不现实的进度计划，要尽可能准确地估计完成不同任务都需要多长时间。在面对比自己的任务时间表短的进度计划时，应该问清楚首先要完成哪些任务，哪些任务在时间有富余时才完成。如果经理坚持要求完成所有工作，则要问清楚哪些任务可以完成得不那么全面，并要求提供更多人手。有时也会遇到不太讲理的执行经理，坚持要求同意在不可能的时间内完成所有工作。好吧，测试经理可以提出这些要求和问题，但是这些要求并不一定能够奏效。现实就像压路机，最终会征服项目团队和员工，而不管人们做出怎样的承诺。面对不听别人说明理由的执行经理，测试经理要明智地管理好自己的员工，否则会失去他们。

长期加班的另一个原因是经理奖励加班（“坚守岗位”）的人。每周“坚守岗位”80小时的人，看起来比每周工作45小时但是工作完成得更多、更好的人更加专心和投入。测试经理的挑战是找出一种方法，使更有效率的员工的工作效果能够展现出来。我们不能通过书本告诉读者应该怎样做。我们的成功作法都是针对特定个人（经理和员工）的。

还有两类执行经理也会给测试经理带来与加班有关的管理问题。一类执行经理认为制定软件进度计划是不可能的，所有进度计划都会流产，因此能够做的事就是询问经理想要的一切，并尽可能努力地推动员工。另一类执行经理看起来认为只有引诱员工尽可能多地加班才能体现出自己的价值。这两类经理我们都见过。由于他们本质上是不大讲理的，因此很难给他们讲道理。好消息是软件公司的经理和执行经理经常跳槽，并会出现重组。傻瓜和外行来来去去，测试经理的挑战就是减轻这些人对自己员工的影响。

不要让员工被滥用

有时别人不能善待测试员，威吓，大声叫喊，把测试员称作傻瓜、骗子等。如果测试经理发现测试员遇到这样的困难，也许该给员工做一些规定，为员工提供精神支持，并解决各种不公正待遇。

成对测试也是一种使测试员能够相互支持的好方法。

在压力很大的环境中，测试员可能说出不恰当的话，做出不恰当的事。在管理上，应该礼貌地处理表现不当的测试员。私下向测试员明确说明这样的表现是不能接受的。如果屡禁不止，则应该按照公司规定进行处理。

创造培训机会

成立阅读小组，每周活动一次，或每月活动两次。目标是每周阅读一篇论文或专著中的章节，并进行讨论（测试员讨论，测试经理参加但不要说很多话）。这样的活动要体现自愿原则，所有人都不是被迫参加的。但是要寻找某种方法奖励经常参加活动并积极发言的人。（例如为说明已经读过部分章节的人赠送书籍。）

召开午餐培训会议（每周一次或每月两次）。有时测试经理要讲话，有时可请客座演讲人（可以是从其他公司请来的测试经理）。可以邀请愿意免费演讲的

外部人员，只要测试经理也愿意给对方员工演讲。有时某位员工可以介绍一种技术，或测试某种类型设备或应用程序功能所遇到的挑战。有时还可以给出一两个练习，供测试员实践。

如果公司统一安排培训，可搞清楚当地大学提供什么相关课程，有什么更好的与软件有关的在线课程。类似地，搞清楚都有哪些测试（和其他相关）会议，哪些会议最适合公司的项目。个别与有关员工交流，对他们的继续教育计划提供支持。这些讨论是研究测试员较长时期目标的很好机会。（在分配下一批任务时应该考虑这些目标。）测试经理不要告诉别人该选什么课，但是要指出自己认为教得比较好并与公司当前项目有关，或对员工职业发展有益的课程。

一般应该派出两人或三人参加课程（特别是职业培训研讨会）和会议，以便能够讨论所学到的内容。学习回来后，要向测试小组报告所学的内容。

录用决策是最重要的决策

录用不适合该工作的人，是经理能够做出的最差的决策，可能要长时间忍耐这种错误的后果。

要想方设法录用好的员工。

在招募期间利用承包人争取回旋余地

利用临时帮助，以便有充裕时间寻找合适的人。即使是对利用承包人有严格限制的公司，也常常允许这种特别的用法。

谨慎把其他小组拒绝的人吸收到测试小组中

有些没有在其他小组中干出成绩的人在测试中做出很大成绩，但是这样的人并不多。测试小组接受的失败者越多，就越会背上“失败者”的名声。应极力拒绝执行经理推荐的被其他小组拒绝而测试经理自己又没有把握的人。即使这是免费的“礼物”（这种调人并不会抢了现有员工的饭碗，也不会占用新员工录用名额），但是更多的人头数会影响下一次请求批准新员工的录用。（你刚刚得到人了，为什么还要？）周围的人也会认为测试小组的能力应该与员工人数成比例地增加。测试员增加会招来新工作。如果有了新工作，而新测试员又不能

承担，只会增加小组其他人的工作负担。

对测试小组需要承担的任务，以及完成这些任务所需的技能做出规划

不要根据简单的资历材料决定人员录用。测试员的真正资历并不只是合适的学历，或多少年的经验。录用有或没有传统资历材料，但是能够证明可以胜任测试工作的人。

测试团队成员要有不同背景

录用的测试员不要都是有计算机科学学士学位、两年测试经验、有特定自动化测试工具使用经验等条件的人。测试小组成员要都有自己的强项，且彼此有很大不同。我们曾经录用过的几个人从来没有干过测试，但是有其他相关的实践经验。如果使测试小组具有所需的能力，这样做会很成功。[⊖]

我们并不是说不应该录用程序员。确实要录用程序员。需要程序员承担很多工作。但是，应该要通过培训使其具有其他非程序设计方面的知识。其他测试员可以是拥有程序员不可能有的很多非程序设计领域知识的人。要使这些测试员的优点能够相互弥补各自的弱点。

例如（这些例子都基于我们个人的成功经验），考虑引入一个很聪明的人，其最新工作角色是律师，可以分析所给的任何规格说明，可以培养成代言人、销售和市场开发主任（我们录用其进行培训，以推广搜索并编写错误报告、引起市场开发部门注意的新方法）、硬件维修技师、库管理员（请考虑测试数据库或其他信息检索系统）、程序员、项目经理（非软件项目）、具有支持与被测软件类似产品的技术支持代表、翻译（如果公司以多种语言发行软件则特别有用）、秘书（请考虑所收集、存储和分发的信息，以及必须处理的测试经理及员工时间安排）、具有网络知识的系统管理员、被测软件的使用者。这些技能都使员工对项目很有用。

这条建议的适用性是有限的。对于某些产品，测试员必须能够编程。请想像测试编译器——为了编写有意义的测试，测试员必须能够编写代码。在有些公司

⊖ 尽管我们并不能总采用这种建议，我们原则上同意本书评审人之一Rex Black的意见："当我们进行需要利用专门技能的测试，例如自动化测试特别是性能测试时，我认为慎重的安排应该是保证测试小组中至少有两个人掌握这些技术。"

中，非程序设计测试员不管表现多么出色，总会被看作是二流员工。在这种情况下必须录用程序员。测试经理的挑战，是录用具有不同背景的程序员。根据我们的经验，录用具有不同背景的程序员并不容易，但是如果是好公司，并且业界不处于录用高峰期时，还是能够做到这一点的。

另一种多样性问题也值得提一下。偶尔有文章提到计算机界的种族主义、性别主义和年龄主义，我们也有同感。团队成员的录用、培训、工资和提升存在偏好，使得白人男性不必要地占据优势，这尤其会影响测试生产率。测试员的文化和经验越多样化，分析软件就会有更多的方法，就能够发现更多的问题。对于测试来说，多样性是一种关键资产，而不是要回避的东西。

录用其他渠道的应聘者

在非传统场合寻找测试员，尤其是在人力市场很紧张的时期。例如，律师和会计都有很强的分析技能，令人惊讶的是，其中很多人都希望花一年（或更长）时间学习如何进行软件开发。另一个例子是刚刚当上母亲的高级程序员或项目经理，需要放慢其工作节奏，但是她目前的公司不同意削减其工作时间。向其提供每周弹性工作35个小时的岗位，就能够获得有一两年特别经验的员工。另一个例子是正在寻找新的、也许压力较轻的岗位的退休执行经理。

根据大家意见决定录用

让测试小组中的任何全职员工以及与测试小组工作关系密切的任何人与测试候选人面谈。所有参加面谈的人都可以否决候选人，除非否决是基于性别、种族或其他与完成该工作能力无关的其他（非法）偏见。

抛弃基本情感，并同意接受候选人，这实在太容易了。我们犯过很多严重的录用错误，包括录用了性骚扰者。测试经理应细心、投人地倾听员工所描述的观点，并尊重员工的否决决定，除非员工的否决有歧视问题。

录用热爱自己工作的人

寻找热心的人，谨慎录用与过去的经理存在过节的人，尤其要小心看起来对其过去的工作不满的人。

录用正直的人

测试小组要提供有争议的信息。客户的信任是测试小组最重要的资产。测试员的人格会影响客户的信任程度。

在面谈时，让应聘者展示期望有的技能

例如，如果应聘者是有经验的测试员，则可让他写一份错误报告。（利用非本公司编写的公开程序，以避免使应聘者有测试小组想得到免费咨询的印象。）再举一个例子，如果要录用一位自动化测试结构分析员，可请应聘者分析产品及其条件，并提出高层自动化测试计划。（同样也应该考虑使用非本公司编写的公开源产品。）通过应聘者询问的问题、所进行的查询以及把各部分信息综合为整体方法的方式，对其做出判断。所给出的方法不一定是完备的，但是应该是合理的，并有很好的推理。（有关这方面问题的进一步讨论，请参阅DeMarco和Lister 1999。）

在面谈时，请应聘者通过非正式能力测验展示其在工作中能够运用的技能

有些测试小组确实通过逻辑或数字脑筋急转弯迷题，来实行某种非正式的能力测验。我们并不反对这样做，不过我们认为这样做并不像有些人想像的那样能够说明问题。

对于逻辑和数字脑筋急转弯迷题，通过做大量实际练习会有很大作用。Kaner曾经让他12岁的女儿练习逻辑和数字脑筋急转弯迷题，效果相当不错。但这并不意味着她更聪明，也不会使她成为更好的测试员，而只意味着她能够更好地解答脑筋急转弯迷题。这种大量练习是准备通过像SAT、LSAT、GRE和其他标准化大学人学考试的基础。实践作用（以前的经验）可以持续很长时间，对于速度测试效果更明显，对于非语言测试和性能测试效果也更突出（Jensen 1980）。能够高分通过这些考试的人，也许只是更熟悉这种考试而已。得分很低的人也许只是没有回答这些问题的经验，但是（根据我们的经验）却很聪明，并能够成为出色的测试员。

速度测验考察的是快速选择的能力，不一定需要全面思考，筛选出来的是脑

力兔子。而脑力乌龟有时却能够设计出更好的产品，或更好的产品测试策略。

在录用时，要求应聘者提供工作样本

并不是所有人，但是很多人都能够提供他们自己编写的代码样本、所起草的错误报告、所撰写的论文或报告。由于正常原因被解聘或离开公司的人，原公司常常允许其使用一些特定的经过标识的工作成果，以便潜在雇主了解其业绩。其他人可以在空闲时根据公开软件起草报告或编写代码。永远不要要求提供机密材料，并保证应聘者提供的材料是非保密的。

一旦拿定主意就迅速录用

要优先让录用材料迅速通过公司内部手续，否则好的应聘者会在这段时间内接受其他职位。

要将录用承诺形成文字，并遵守诺言

测试经理不兑现对别人就某事所做出的承诺，会使当事人感到不快，会认为测试经理经常改变主意，测试小组其他人也会受到警示，有可能使测试经理的信誉受到损害。即使从来没有做那样的承诺，但是由于承诺不明确，也会受到同样影响。永远不要承诺超出自己职权的事，例如职业发展或提升等。

第10章

软件测试职业发展

把软件测试作为职业有怎样的未来呢？回答是复杂的。测试职业很容易走进死胡同，因此必须控制好自己的职业发展，否则就不能有所作为。与其他开发职位相比，测试的回报往往相对较低，但是如果能够积极积累自己的技能并挑选好的雇主，回报也不一定低。测试员的岗位变换频率较高，与具有同样能力的开发人员相比，更经常被解聘。培养自己的找工作和谈判技能是很有用的。

我们提出的找工作和谈判建议，是专门针对美国人的，特别是硅谷的美国人。如果读者不能肯定我们的建议是否适用于自己的文化，可征求一些有经验的同事的意见。

有些测试员认为如果能够得到认证在工作应聘时会更具竞争力，并更能受到重视，如果有执照，工作就会更得心应手。认证和执照也有优势和弱点。以下将进行讨论。

确定职业发展方向并不懈努力

测试方面职业发展的两个一般方向是技术和管理。测试中技术工作的一些例子包括（在这些工作中都可以从初学者成长为专家）：

- 自动化测试程序员。
- 自动化测试结构分析员。
- 性能和可伸缩性测试员。
- 系统分析员。
- 用户界面和人员因素分析员和鉴定员。
- 测试计划设计员。

- 专题测试专家。
- 黑盒测试员。

假定读者在以上任何一个工作岗位上有5年的专门工作经验，那么我们的印象是，如果是自动化测试结构分析员，则应该拿最高的工资和名望，有更高的录用机会；如果是出色的自动化测试程序员，则应该拿第二高的工资。目前性能和可伸缩性测试员的工资也不低。我们的印象是，如果是出色的黑盒手工测试员或对专题知识的掌握多于测试技术掌握的专题测试专家（取决于专题内容），则工资是最低的。

这里有一种有趣的利益冲突现象。测试员通过学习范围相当宽的测试技术，并用领域知识加以补充，会使自己对当前雇主的价值得到提高。这样的测试员可以要求明显更高的工资。但是，如果是为对其员工没有诚信的公司工作，或由于其他原因期望改变工作，则最好还是广泛学习一些有价值的技能，即使是以损失领域知识以及为这个雇主工作的有效性为代价。一条很实用的指导原则是，确定雇主（或行业）最需要的技能，并努力掌握这种技能。

测试中管理工作的一些例子包括：

- 测试小组组长。
- 测试经理。
- 测试主任或质量主任。
- 内部顾问。
- 外部顾问。

测试员可能会发现测试之外的提升机会。与程序员相比，测试员对产品有更广、一般来说更浅的认识，并往往对整个产品有兴趣、认识和影响。测试员还和公司大多数其他与软件有关的部门经理和高级经理交互。结果，很多熟练的测试负责人或测试经理得到培训并调到其他管理岗位的机会，尤其是：

- 程序设计经理或项目经理。
- 技术支持经理。
- 产品经理（特别是有技术技能的领域专家）。
- 文档编写小组经理。
- 销售支持经理（对于高端产品，销售人员常常配有能够回答技术问题并与客户一起开发产品原型应用程序的技术人员。）

测试员的另一个发展方向是过程管理人员，包括：

- 软件指标专门人员。
- 软件过程改进专门人员。

我们建议在向过程管理方向发展时应该小心。这些岗位并没有直接捆绑到任

何产品的开发和收益上。因此在有些公司中，这些岗位更容易受到裁员的影响。此外，在有些公司中，这些岗位都由能力不强的人担任。我们建议读者在担任指标专员工作之前，首先全面掌握数学统计和度量理论（阅读并理解Zuse 1997和Austin 1996）；建议在担任过程改进专员工作之前，至少在两个部门，最好在三个部门工作过。

测试员的收入可以超过程序员的收入

传统上，测试员挣的钱要比具有类似经验的程序员少。并不是所有公司都是这样，有人说，在过去几年中，测试员的收入已经超过程序员。

收入比较很复杂。在网络公司蓬勃发展时期（1998~2000），各类软件工作的工资都上涨。在很多公司中，有经验的软件测试自动化结构设计员的工资比有经验的程序员高。只进行手工测试的有经验的黑盒测试员的工资也不低。在网络公司低潮期（2000~2001），工资都被削减，我们怀疑黑盒手工测试员的平均工资削减得更快。

我们的印象是，具有专门技术特长的测试员比具有管理技能的测试员更受重视（也更难找到），而具有管理技能的测试员比黑盒测试员更受重视，但不如有经验的测试自动化测试员受重视。

可大胆改变职业发展方向并追求其他目标

测试员并没有被锁定在测试上，也没有锁定在管理角色或岗位上。测试员并没有锁定在任何公司、任何职业发展方向或任何工作上。

行业都有发展周期。软件就业市场有供大于求的情况，也有供不应求的情况；有景气的时候，也有萧条的时候。当软件就业市场供不应求时，雇主很焦急，测试员可以相对更容易地大胆转换工作。但是，即使是软件就业市场供大于求的时候，对于那些已经做好准备的人来说，也是有大量机会的。

例如，如果要撤离测试领域而进入程序设计领域，可参加学习班。如果由于工作时间表的原因不能参加大学课程或大学拓展课程（有时又叫做职业教育），可以参加网上由自己决定学习节奏的课程，或“书加CD”课程。掌握了有关程序设计语言的基本技能之后，可以向一个程序设计小组提出用自己业余时间承担一部分工作。但可能会受到来自测试经理的一些阻力。我们

建议可寻找某种方法向测试经理保证每周可以进行40个小时的测试工作。40小时之外的时间应该由测试员自己支配。（如果测试员不能得到测试经理的同意，而且又想转向程序设计或其他新的发展方向，可以寻找其他地方的临时工作。）

不管选择走哪条路，都要积极追求

职业发展方向是自己的事。公司或经理也许愿意帮助员工规划职业发展方向，员工也可能会从中受益。也可能不是这种情况。如果有了发展方向，以及支持向这个方向发展的技能和决心，公司更有可能会为员工支付学习费用，使员工能够尝试新事物。

有些公司会支付员工的学习费用，有些公司不会。我们鼓励读者考虑自己的职业发展方向问题，搞清楚自己需要什么，自己对什么感兴趣，这会使自己对当前公司或下一个公司（更看重自己才能的公司）更有价值。

有些公司会让测试员全职或兼职参加其他开发小组的工作。有些公司不会，或允许测试员全职参加测试工作，兼职参加其他项目团队的工作。这第二种情况更常见。做好加班的准备，以便调换岗位。如果不能加班（也许家庭负担很重，不允许加班），仍然需要挤出晚上时间参加学习班，并参加非雇主的项目（例如参加开源软件的开发）。（注意不要打破业余夜生活习惯。例如，如果要在家里编写一些开放代码，并利用上班时间进行测试或润色，雇主也许会拥有开放代码的版权。还应该注意，不要编写会对雇主产品构成竞争的程序。）

超出软件测试拓展自己的职业发展方向

很多最得意、最成功的软件测试员经常转换职业。

花大量时间寻找并批评其他人的错误会变得俗套。尝试做一些其他工作，掌握一些软件开发的其他技能。自己也犯一下程序设计错误。如果有兴趣，还可以再回到软件测试。这会使自己见识更广，更灵活，饭碗更牢靠，更适合就业市场的需要，对项目团队所面临的挑战和约束会更有见地，会得到开发小组其他成员更大的信任，也许还会拿更高的工资。

与只承担一项任务的经理相比，能够发挥多种作用的经理常常能够按不同的

更高级别获得报酬。完成多种任务的经理在很多公司中都会得到赏识。

超出公司拓展自己的职业发展方向

职业发展方向是自己的事。如果公司明天出现问题，或削减员工，员工仍然还是原来的自己，职业发展方向仍然还是原来的方向。（只是必须找新工作，并且常常可以找到比原来更好的工作。）软件测试员群体很大，软件开发人员的群体更大。我们写文章、开会、争论，我们相互学到很多东西，相互提出忠告，并相互帮助寻找新工作。

我们可以以很多不同的方式参加这个群体。出席软件测试或软件开发会议。参加当地的计算机学会、美国质量学会、人员因素与人类工程学学会，或其他与软件质量有关的职业学会。把自己列入“软件测试讨论邮件列表”（www.testingfaqs.org/swtest-discuss.html），并且参与发表有价值的意见。如果读者赞同本书末尾给出的背景驱动测试原则条款，就可以签署（www.context-driven-software.com）并加入我们的邮件列表。有关联网方面的建议，请参阅Agre（2001）。

参加会议是为了讨论

参加有关软件测试或软件开发的会议时，不要只是坐在分组讨论会场中听别人发言。应该参加很多（或至少部分）分组讨论，但是还要花很多时间与参加会议的其他人见面，讨论所做的发言或本领域中的一些问题。

如果不认识很多的与会人员，就与部分人员见面。午餐时，与不认识的人坐在一起，听其谈话，寻找有兴趣并且有见地的人。利用其他机会寻找有兴趣的人，然后邀请其一起喝咖啡，或询问他是否接受自己早餐请客。在与这些人见面时，询问他们的工作内容，了解他们在会上所讨论的内容。他们发现了什么重要的东西？经过一段时间之后（需要参加多次会议），就会结交一些主要的在会议上见面的朋友，通过因特网与在会议上认识的人交流最新信息，也许还会与其合作发表论文或发表演讲。我们三个人就是这样走到一起的。

并不是所有人都有机会被公司派去参加会议。要提前足够长的时间向自己的经理说明想要参加的会议，使经理更加关注能够增加派自己参加会议的机会。参加了一两个会议之后，就可以申请发言。如果会议接受了发言申请，公司就

更有可能派自己参加会议，并且会在公司内外（如果发言很有见地）赢得声誉，并受到关注。

很多公司的问题并不比本公司的问题少

很多测试员会对本公司产品中的大量程序错误感到不可思议，并对本公司存有疑惑。这种情况并不少见，即使是在很好的公司中。测试员看到的是问题，而且问题还很严重。但这并不总是好事。

从统计角度看，测试员有可能在占总数10%的最差公司中工作，但也有可能不是，因为有些公司甚至没有测试员。与其他公司的测试员建立联系，有助于自己的未来发展。

如果不喜欢自己的公司，就再找一份不同的工作

在当前的岗位上干下去通常都是好主意，但是当前有工作几乎总是比当前没有工作更容易找到好工作。不要把自己的简历张贴在因特网上，除非要让自己的老板最终看到。只与少数招募人员打交道，只有同意对应聘者进行考核时，他们才会要求把简历寄到某个地方。

在与其他公司进行面谈之后，应聘者可能会接受新职位，也可能发现当前工作的公司也并不坏。决定继续干原来的工作，可以是寻找工作努力的非常合理和成功的结果。

如果现在的工作环境如此恶劣，以致于影响自己对自己的看法，以及面谈时对自己的描述，那么最好还是离职，休息一段时间，然后再寻找新工作。除了从差公司辞职之外，特别是当辞职损失过大时，还可以决定不管公司的底线和条件是什么，都继续干好自己的工作（正直，但不投人），也可以通过其他方式提高自己工作方面的声誉。如果可能，可参加夜校。如果可能，可多注意休息。目标是使自己处于很好的精神状态，以便更有效地为下一个潜在雇主工作。

为寻找新工作做好准备

我们中的Kaner总是关心就业市场信息，面且只要与测试小组合作，总要给

出自己的简历。这并不是因为他要不断换工作，而是防御性作法。他常常处于有问题的情况。知道自己有最新简历，与一些很好的招募人员有不错的关系，知道有些什么类型的招聘岗位、哪些公司要招聘以及相应的招聘待遇，他就能够对找到新工作很有信心，就能够相当迅速地找到工作。有了这些信息，他就能够承受否则不会承受的风险。

积累并维护希望加入的公司的名单

很容易收集公司信息，特别是通过网络、会议和关系。考虑自己希望加入的公司，与在那些公司中工作的人见面、交谈。

积累材料

掌握能够证明自己能力的一些代码、文档和其他工作样本，对于测试员来说非常容易。可以利用这些材料显示出其他应聘者所不具备的能力。但是积累这些材料并不容易。

在大多数公司中，测试员的成果是专属的，即由公司拥有，并且是保密的。但是，测试员可以编写不是特别敏感的文档或一些代码，也可能创建的部分工作产品今天是敏感的，但是5年之后就不敏感了。还是有一些可以用于积累的材料。有些经理允许员工合适地使用非敏感材料，但是有些公司很不痛快。测试员得到允许使用在公司中创建的材料的最佳时机是：

- 在裁员涉及到自己时，经理或人力资源部的代表会解释裁员条件。如果自己手中有特定的工作样品，他们会乐意签署文件，允许员工向可能的雇主或客户出示这些（指定的）样本，以便获得工作。请他们签字，而不只是口头允许。
- 离开公司半年至一年后，如果自己的前老板还在原公司工作，则这个问题比较好处理。
- 在会议上用作例子。如果被允许在会议上公开披露秘密，而且也确实披露了秘密，则再次披露就不再需要进一步的批准了。这些秘密已经不再是秘密了。

如果利用员工自己的时间开发软件或撰写论文，使用的是自己的资源，没有使用公司的任何资源，并且所开发的材料与自己在公司中的任务没有直接联系，

则也许员工拥有所编写的材料，并且能够任意使用，有权在更广泛的条件下运用自己的工作产品。具体细节取决于员工与雇主的合同，以及控制这种合同的法律。

把简历当作推销工具

简历会把自己从一些工作岗位考虑中排除，并加入另外一些工作岗位的考虑中。我们的目标是写出针对恰当子市场的简历。

- 如果对有显著不同的工作岗位感兴趣，则写出不同的简历。这些简历的介绍重点不同，要突出最能够说明自己的技能、兴趣和背景。要保证自己不同简历的有关背景和成就的描述是一致的（永远假设雇主最终会看到自己的所有简历）。
- 准备一份历史简历。历史简历要以时间顺序描述自己的工作经历，突出介绍在每个工作中取得的成就。有些招募人员（以及一些雇主）喜欢功能性简历，说明应聘者运用每种技能的经验，包括在哪个公司中掌握的这些技能等细节。功能性简历没有什么不好，但是有人会利用这种简历夸大其经历，或隐瞒其失业时间段。因此，有些雇主经理（例如我们）会坚持要应聘者提供以前的工作描述。如果应聘者没有书面材料，我们只好通过电话询问，或进行面谈。如果有书面的工作描述，则在专门核实应聘者历史的电话审查时，应聘者可以向雇主经理发送按时间顺序排列的简历传真。有些经理会对这种细致的准备和考虑留下深刻印象。有些雇主经理不会考虑只提供功能性简历的应聘者，除非应聘者的技能非常适合应聘的岗位，并且简历是可信的。
- 准备一份功能性或关键词简历。有些公司将简历扫入计算机中，或由一些不理解招聘岗位的人快速浏览简历。如果应聘的是这样的公司，就必须在简历中包含合适的关键词或短语。功能性简历很适合这种处理。此外，在历史简历中，还可以考虑增加一段“关键词：供基于计算机的搜索使用”，然后在段落中列出所有程序设计语言应用程序、重要的技能等。只列出内容，不要解释。
- 决不要夸大自己的背景、技能或经验。诚信是自己的主要资产。如果有人强烈怀疑简历中有夸大或说谎情况，应聘者就不会被录用。甚至在被录用之后，别人还会看简历。在那个岗位上工作了两年之后，另一个小组调来的经理（或新经理）可能会翻阅简历，并发现应聘者显然并不了解其当时

声称具有的专门知识。这位经理不会和当事人谈论这个问题，但是可能会和其他经理谈论。这无助于自己的职业发展。

找一位内部推荐人

很多公司都给推荐被录用应聘者的员工提供奖励。可利用这一点。利用拟应聘公司中的内部员工推荐自己，这样也能了解该公司的内部信息，也许还能够得到有关面谈和协商过程的重要反馈信息。我们都这样做过，效果不错。

搜集薪金数据

通过像www.salary.com这样的数据库，可以很容易地得到现实的薪金预期值。了解这些信息，更好地准备面谈。

如果是根据招聘广告应聘，应根据广告要求调整自己的申请

如果招聘广告列出多项要求，应逐项回答。明确指出要求表中自己已经满足的条目，以及还不满足但很希望学习的条目。

充分利用面谈机会

当应聘者（或招聘人员）提交自己的简历后，有的公司可能会提出针对自己并没有期望接受的工作岗位的面谈。该工作岗位的薪金可能不够高，可能要到自己不想生活的城市上班，或由于其他原因对该工作不感兴趣。但是可以考虑接受这种面谈，并以确实想应聘工作的同样热情准备面谈。这种面谈是一个在没有危险且真实的环境下，实践自己面谈技能的机会。

可以有机会实践回答面谈问题，并听到自己事先没有准备的新问题。如果是新手，或没有应聘经验，在第一次面谈时会感到紧张。经过多次应聘面谈的实践之后，会更好地了解招聘人员期望什么，并更自如地应付面谈。参加即使是公司同意录用自己也不想接受的工作面谈，会帮助自己积累面谈经验，并增强自信心。

可能还会发现那份工作根本就不是自己想要的，或许公司也可能会提供自己感兴趣的其他工作机会。

了解准备应聘的招聘公司

如果没有工作，并寄出很多简历，就不会有时间研究每个公司，那么就别研究每个公司。但是如果发现一个特别感兴趣的岗位，则可以免费在因特网上做大量研究，并花少量的钱做进一步调查。访问该公司的网站，了解其人员，下载可能提供的演示软件，寻找他们所使用的程序设计工具信息。可以利用搜索引擎。如果是公开信息，可以访问投资者信息网站并询问问题。知道得越多，越可以对应聘信做更多的剪裁。

一旦了解到有对自己感兴趣（自己也对其感兴趣）的公司，就应该进行研究。知道得越多，与这些公司的讨论越有效。

如果确实有公司打来电话，要求进行电话交谈，可向其索取以下信息。

- 索取产品材料或公司介绍材料。
- 索取他们发行的软件演示版，或询问公司所使用的开发和软件测试工具，询问公司所开发的应用程序类型。
- 询问从哪里可以找到有关该公司的其他信息。

如果公司寄来一些材料，一定要在面谈之前看完。

公司参加面谈的人希望知道应聘者是否想在该公司工作，以及为什么要在该公司工作。很多招聘经理更喜欢迫切需要该工作岗位，并且对该工作岗位提供的机会很热心，但是刚刚达到应聘要求的应聘者，而不是超出招聘要求很多，但是不够兴奋的应聘者。对工作岗位很兴奋的人可以在工作中学习，并从中获得乐趣。（工作可能更努力。）只是说："噢，噢，我真兴奋"，不如显示出已经阅读过公司材料并对公司产品真正感兴趣的应聘者那么令人信服。工作业绩（在这里就是研究结论）胜于雄辩。

在应聘时询问问题

在应聘谈话（电话谈话或面谈）时询问问题也是一个很好的主意，尽管也许不想向一个人问太多的问题，并且应该针对不同的人挑选不同的问题。以下是一些常常产生有意义回答的问题例子：

- 贵公司提供什么产品和服务？
- 我可以看一下关键产品的演示吗？
- 贵公司提供的产品和服务有什么特别之处？有哪些主要优点和弱点？
- 贵公司如何开发主要产品？有些什么关键的开发综合考虑？
- 贵公司的客户有哪些？
- 贵公司的竞争对手有哪些？
- 贵公司如何了解自己的客户？
- 贵公司如何了解自己的客户对整个产品、设计和缺陷的满意程度？
- 请描述贵公司的组织图（你们在什么位置，我将在什么位置）。
- 贵公司的工作情况怎样？
- 你的工作是什么？你（与其谈话的人）提供什么产品和服务？
- 我可以看看例子吗？
- 你在产品开发过程处于什么位置？
- 你对你的工作有什么看法？
- 你想做一些改变吗？
- 你怎样安排时间与家人待在一起？
- 你对自己的工作有多大的控制力度？
- 谁来设计你所运行的测试？谁来运行你所设计的测试？
- 请谈谈你们的测试设计过程。
- 我可以看看一些测试计划和测试用例吗？
- 你对自己的待遇、老板、同事有什么看法？
- 去年你听过哪些课，出席过哪些会议？
- 你还接受过什么其他培训？
- 你怎样学习新技术？
- 请描述去年你所学到的三种关键技术。

在面谈时要集中注意力看着对方，认真地听。

- 面谈者累了吗？
- 面谈者和其他人对他们自己工作的看法是肯定的吗？
- 他们是否有很好的士气和团队精神？
- 他们的办公设备如何？他们在提高工程技术人员的环境舒适度和生产率方面的投资有多大？员工的办公设备与经理和执行经理的办公设备有多大不同？
- 测试员占有多大的工作面积、设备和照明，更高层人员占有多少？
- 寻找所声称的工作条件和实际条件之间的差别。例如，有些公司在其广告中声称他们实行每周4天、每天10小时工作制，但是他们却要求员工在其

他3天（别人认为应该放假的3天）完成很多其他工作。像这样的条件差别会产生怨恨和不满情况。这也是公司管理层存在潜在道德问题的早期警告。如果他们不直率地讲出其对员工的基本要求，那么他们还能直率地讲出什么呢？

就自己的工作岗位进行谈判

本书只能简单粗浅地讨论岗位谈判问题。谈判是一种技能，必须实践。可通过朋友进行实践，可通过潜在雇主进行实践。（会损失一些潜在工作岗位，但是会从自己的错误中学会谈判。）

在谈判期间得到的信息越多，谈判就越有效。以下是一些关键信息：

- 知道自己想要什么。
- 知道他们想要什么。
- 知道他们的想法。
- 知道他们作为潜在雇主怎样谈判。
- 知道自己的其他选择。除这个工作岗位之外的其他工作岗位是什么？

有其他选择对于谈判非常重要。如果现在有工作，那么最容易的其他选择就是继续留在现在的工作岗位上，直到出现更好的工作岗位。这也是已经有工作的人再找更好的工作，要比没有工作的人更容易的关键原因。与有工作的应聘者相比，没有工作的应聘者很少有可信的和明显的其他工作选择，他们常常有挫折感，即使没有挫折感，也看上去需要尽快得到工作，或就要用光积蓄（并开始有挫折感）。因此，与有工作的应聘者相比，公司常常感到可以提出比较低的待遇条件，并做出更少的承诺。如果是没有工作的应聘者，在与不错的公司接触时，不要放松继续找工作，而要加紧进行。找到另一家不错的公司。不管应聘者是否让各个公司知道这一点（在向一家公司提起另一家公司时应该小心，有些人认为这违反保密协议），在谈判时都会有所不同，因为知道自己还有其他选择。

以下是在谈判期间会出现并应该考虑的一些问题：

- 在会谈初期，要讲清楚自己想要什么（非报酬方面）以及自己对什么感兴趣。如果公司对于应聘者的兴趣也感兴趣，就会使谈话（以及接下来的谈判）以谋求应聘者加入其公司的方式进行。
- 应聘者应拒绝提供以前的报酬信息，在雇主对自己感兴趣之前，婉转地拒绝回答有关报酬期望方面的询问。解释应聘岗位与以前的报酬无关，因为

以前的岗位不同，常常有助于拒绝回答这种问题。自己对应聘岗位的待遇要求是合理的。然后询问公司对这个工作岗位设定的报酬是多少。有些招聘人员会告诉应聘者，有些会拒绝透露，这时应聘者可以说："我们以后了解了责任和机遇之后，再讨论钱的事。"

- 应该热心，但不要过于迫切。
- 应该使用他们的词汇讨论他们的产品。
- 应聘者应该发现并指出自己能够为招聘公司提供帮助的方面，提出建议，给出自己想法的例子。
- 应聘者要让招聘公司知道自己想要哪份工作。（对于有些文化，更合适的作法是，说明该工作很有意思，并且非常合意，但是一定要在准备接受该工作之后，才能说明自己想要那份工作。）
- 如果应聘者并没有完全达到该工作岗位的要求，但是又想得到这份工作，可作为一种拓展机会坦率提出来。拓展机会是一种挑战、是以前没有做过的工作，必须为此掌握新的技能和知识，但又是自己想做的事。要让招聘公司知道自己多么想得到这份工作，这份工作多么适合自己的兴趣和发展目标，以及自己如何在该岗位上施展才华。
- 当心潜在雇主员工的带有威胁味道的谈判风格。

请注意，在谈话和谈判时，公司经理都会极为检点。当应聘者开始为招聘公司工作后，他们不会更友善，不会减缓威胁口吻，不会更诚实，也不会更为员工着想了。

可参考一些很好的谈判专著：Chapman（1996）、Fisher和Ertel（1995）、Fisher等（1991）、Freund（1992）、Miller（1998）、O'Malley（1998）、Tarrant（1997）。有关谈判的一般建议，我们更喜欢Freund的书。

留意人力资源部

人力资源部并不做出录用决策，尽管他们确实常常决定录用经理该看哪些简历。各职能部门的经理（例如测试经理和开发经理）最终决定录用谁。要与决策者谈话，而不是与人力资源部谈话。

学习Perl语言

Perl是一种很方便的脚本语言，公司中的很多程序员也都了解Perl。（因此，

如果遇到问题可以请教他们。）采用Perl可以编写大量实用程序。例如，可以编写Perl程序分析日志文件，向设备馈入数据，或设计自动化测试并馈入数据。越经常使用Perl语言，就会有更多其他使用这种语言的新想法。（请注意：如果Perl不是本公司所选择的脚本语言，可以使用公司指定的脚本语言，也可以使用自己最喜欢的语言。Python和Ruby也是经常提到的脚本语言。如果读者不知道从哪里开始，并且也没有特别的理由选择其他脚本语言，我们建议从Perl开始。）有关脚本语言的进一步信息，请访问www.softpanorama.org/Scripting/index.shtml。

学习Java或C++

如果有机会积极运用所学到的知识，例如通过编写自己的测试工具，来学习本公司所使用的主要开发语言，并在实践中运用。学得越好，本公司内和新工作的选择机会就会越多。

下载测试工具的演示版并试运行

对于这些工作，测试员也许必须利用自己的时间在家中进行。我们建议应熟悉本公司（还）没有使用的工具。公司也许不会支持测试员进行这种研究，或让测试员把主要项目放在一边来完成这种工作。

了解了测试工具的内容和功能之后，可以说服公司购买并开始使用。如果工具很好，别人会赞赏所做的推荐。做出一些好的推荐之后，别人就会将其看作是小组的工具研究专家。

不管当前的雇主是否欣赏测试员跟踪最新测试工具，这些知识都有助于应付潜在雇主面谈。

提高自己的写作技巧

测试员的工作在很大程度上要涉及撰写报告和备忘录以劝说别人做事情。写得越好，报告的作用越大。参加有关技术和劝导写作方面的课程，并寻找其他途径实践并提高自己的写作技巧。

提高自己的公众讲话技巧

可以参加很多大学和社区学院都开设的公众讲话课程。此外，也可以考虑参加叫作致词人的组织（www.toastmasters.org）。这个组织提供安全、富有鼓动性的环境，以便学习和实践公众讲话技巧。很多“致词者”俱乐部都是建立关系网的好地方，参加俱乐部的人各式各样（执行经理、经理、其他工程师等），读者也许会对此感到惊讶。

考虑通过认证

Kaner是美国质量认证协会的工程师（CQE）。质量保证研究所提供软件测试和软件质量方面的认证。英国计算机学会正在建立针对测试员的新的认证模式。读者也可以通过其他组织的认证。

这值得吗?

我们的印象是，通过认证考试并不困难，考试的题目常常有一定可预测性。如果参加一个针对考试的复习课程，也许就能通过考试。有人不时将样本考题在网上公布。从这些考题可以看出，考试一般是多选题，有时还是开卷的。考试常常测试词汇和相对简单的概念知识和基本技能。单是通过认证并不意味着就是专家。

参加不只是复习课程，而是一系列课程后通过认证的很多人都认为自己是专家。他们把通过认证看作是一种拓展教育的机会。根据我们的经验，这些人很看重认证，并感到通过认证过程学到很多东西。

认证在应聘简历中很有用。它告诉未来的雇主自己对该领域是严肃认真的，并接受了足够的继续教育以通过认证考试。这能够使自己从其他申请测试工作的应聘者中脱颖而出。但是，如果自己通过认证（但不是程序员），而别人没有通过认证但是能够编程，最终自己没有得到那份测试自动化工作也没有什么奇怪的。通过认证能够带给自己的优势也就是这么大。

综合来看，我们认为软件测试和软件质量认证对于专业发展有好处，这些认证能够鼓励很多人研究该领域的经典专著和论文，学习新的技术，提高其处理与软件质量有关问题的分析能力。

我们不认可认证的知识体或观点，我们与认证机构的观点有差别，但是他们被授权表述自己的观点，并以他们的方法对别人的专业水平进行评判。认证

（我们并不反对认证）和许可（我们反对许可）的本质区别是，认证的地位更低。不管别人是否认证，我们都可以运用自己的技能，我们可以选择看起来最有用或对自己最有意义的认证。认证只是一种接受教育的载体，而不是一种规范（请参阅经验273：有关注册软件工程师作用的警告）。

应该指出，如果读者通过认证，并要成为独立顾问或独立承包商，则应该注意如何开发自己的服务市场。如果通过质量工程认证（或其他方面的认证），别人会认为你有一定程度的专业知识，你就要提供人们认为达到这个水平的人应该提供的服务。如果自己的工作达不到这种要求，客户就更容易起诉你失职或过失（Kaner 1996a）。因此，即使通过认证，也可能不选择向客户提供被认证级别的服务。

经验272

不要幻想只需两个星期就能够得到黑带柔道段位

人们会通过认证市场赚钱，从复习学习班、研究指南、磁带，到考试本身。很多人对认证都非常感兴趣，但是这与要成为成功的工程师没有多大关系。

在我们看来，有些认证没有意义。我们指的是声称只需几周就能够成为行家里手，或获得具有很高级别的培训认证。

要得到黑带柔道段位必须经过长时间的实际训练。第一个里程碑常常是黄带，当教练认为你最终对别人的威胁要比别人对你的威胁大时，就会授予黄带。即使得到黄带也要花比两周长得多的时间。

也许某种学科非常简单，只需两周就可以掌握。如果真是这样，那就心安理得地拿那黑带吧。但是不要抱任何幻想。

经验272

有关软件工程师认可制度的警告

软件工程师（包括测试员）应该得到许可吗？其他工程师都经过许可，所以也许软件工程师也需要。

实用工程的许可证是由政府颁发的。在美国，可以通过州政府得到许可证。因此，如果要在两个州行业，就需要得到两个许可证。注册工程师应恪守道德法则，否则就会由于过失而被起诉。

注册软件工程师的准备工作正在进行。这些工作正在美国的德克萨斯州、加拿大的不列颠哥伦比亚和安大略进行。有关最新报告，请参阅软件工程职业化

（Software Engineering Professionalism）的Construx网页：www.construx.com/profession/home.htm。Mead（2001）全面论述了认证制度的优点。

软件工程协调委员会（Software Engineering Coordinating Committee，SWECC）开发出一种“软件工程知识体系（Software Engineering Body of Knowledge，SWEBOK）”。SWECC是电气与电子工程师学会计算机分会（IEEECS）和计算机协会（ACM）联合组成的委员会。SWEBOK计划的目标，就是构建所有软件工程师都应该知道的基本知识库。

ACM不赞同软件工程师的认证制度，因此2000年6月从SWEBOK和SWECC退出。

> ACM认为，软件工程知识和实践的当前状态还很不成熟，还不能实施认可。此外，（ACM）委员会还认为发放许可证并不能有效地为软件质量和可靠性提供保证……。委员会进一步得出结论，最初针对土木工程的许可职业工程师框架，并不适合软件工程职业行业实践。实行这种许可制度并不能保证应具备的能力，即使知识体是成熟的，并且会使很多最有资格的软件工程师得不到许可……。因为SWECC已经在按照职业工程师模型发放软件工程师许可证方向上走得很远，所以ACM委员会决定退出SWECC（ACM 2000）。

SWEBOK是推动许可证制度发展的一个重要方面，应该引起读者的重视，因为软件测试是SWEBOK的领域之一，软件质量也是其另一个领域。通过www.swebok.org可以下载该文件。

为了得到软件工程师许可证，就必须通过考试。考试必须基于本领域广泛接受的知识和实践。得到许可证后，会由于过失而被起诉。如果注册工程师没有运用技能或运用该职业合理从业人员应具备的知识，而给客户造成损失（例如客户人身受到伤害，财产受到损失，或损失资金），会因过失而被起诉。如果SWEBOK被看作是软件工程知识体系，当立法机构、考试出题机构、法院、律师、陪审团和新闻记者要理解定义软件工程师应该知道什么和应该做什么的标准时，都会引用SWEBOK。

我们在前言和第6章中，已经尖锐地批评了SWEBOK中的陈述。我们并不孤立。有关更广泛的批评意见，请参阅Notkin等（2000）给ACM委员会的报告。

软件工程知识体系（IEEE计算机分会，0.95试用版）在前言部分这样说：

> 本指南的目的，是提供软件工程原理领域被广泛接受的有效刻画，并提供支持软件工程原理知识体的总体描述……本指南强调实践，因此与标准文献有很强的联系。大多数计算机科学、信息技术与软件工程文献都提供了对软件工程师很有用的信息，但是只有相对很少的文献是有

> 关标准的。标准化文档描述工程师在特定的环境下应该做什么，而不是只提供会有所帮助的信息。标准化文献要经过实践者形成一致意见的检验，并集中关注标准和相关文档。SWEBOK项目一开始被认为与软件工程标准文献有很强的关联……最终我们希望软件工程实践标准将包含可跟踪到SWEBOK指南的原则。

SWEBOK的前言把“形成一致意见”定义为：

> 这里所谓的“形成一致意见”，是指使这种陈述成立的惟一实用方法，是通过有关群体的所有重要组成部分的广泛参与并达成共识。

对于我们来说，很难想像已经被美国主流计算业界废弃的文档可以被认为是形成一致意见的文档。在我们看来，SWEBOK的很多标准陈述，并没有描述有理性的工程师在特定环境中应该做什么。

我们认为，把SWEBOK当作形成一致意见的文档是危险的。如果SWEBOK被看作是描述职业考虑和知识的标准，那么没有遵循SWEBOK推荐的方法的失败项目软件工程师会出现什么情况呢？这种失败会被解释为过失，即使该工程师采用的是实际上在那种情况下更适合的实践。

谁来决定在给定案例中没有应用SWEBOK是过失？不是其他工程师。做出决策的是法官、陪审团、律师和保险公司，是一些很少或没有工程经验的人。他们的决策会对行业带来很大影响。

假设某种知识体描述了当前并没有遵循的标准，（根据我们对很多大小软件公司的体验）这种知识体既没有被当作权威性的文件，大多数实际工作者甚至也都没有引用过、阅读过，知识体的优点往往看起来并没有得到实际检验，只不过得到参加了特定标准制定的业内少数人的认可，如果我们为非工程师提供这样的知识体会出现什么情况呢？我们料想过失裁决会随机地与被审查工程师的决策质量联系起来。

美国的法庭一直拒绝接受与计算机有关的过失诉讼，因为软件开发和软件工程并不是注册职业，如果不是注册职业，就不能因职业失职（过失）而提起诉讼（Kaner 1996a）。（1996年以来出现了一些法庭判决，但是都符合Kaner所做的归纳。例如，与计算机有关的最新法庭过失判决，是1999年Heidtman钢铁产品公司诉Compuware公司案。该判决驳回了过失诉讼，并引用了很多以前的驳回例子。）我们认为，除非就能够产生恰当的实践选择的领域技能和决策过程达成广泛、真正的一致，否则把某个领域宣布为注册职业，并把从业人员置于过失法体系的控制之下就是草率的。

顺便说一下，今天的软件工程师职务失职保险并不太贵，因为软件工程师不太可能因为职务失误而被成功地提起诉讼。在面临重大过失诉讼的领域中，每

年的保险费高达几千甚至几万美元。美国的很多州都要求如果许可从业者要在本州行业，必须购买过失保险。软件工程师到底愿意花多少保险费以便能够被准许成为独立承包商或顾问呢?

我们认为，过失有时被错误地解释为保护犯错误工程师的一种努力。我们想澄清这一点。我们非常关注软件质量问题，非常希望出台法律使开发人员，包括开发公司能够对差的工作负责。我们三人都积极提议“统一计算机信息交易法”，因为它能够保护软件开发人员和提供商不对其有缺陷的产品负责。Ralph Nader在审阅Kaner有关软件消费者保护的专著《坏软件》（Bad Software）时说，这是一本“信息时代消费者保护指南”（Kaner和Pels 1998，封底）。

我们不赞成实行软件工程师许可证制度，因为我们认为过失诉讼总体上是一种坏想法。在合适的条件下，过失诉讼是监督从业人员能力的有力工具。但是，这些条件目前还不适用于软件，不管我们怎么要求、怎么希望，都无助于问题的解决。我们赞成制定更严格的软件标准。举一个很简单的例子，我们赞成制定信息披露要求。

软件提供商（开发商、出版商或零售商）都应该被要求在将产品交付给软件客户时，告诉软件消费者提供商已知的所有产品缺陷。提供商要对自己已知的但是没有以这种产品的一般客户能够理解的方式披露给客户的缺陷所造成的损失负责。

我们认为这是一种适中的要求，会对投放市场中的产品质量产生显著影响。

我们赞成制定法律标准，将由于产品缺陷所造成损坏和损失与提供商的错误（失败）声明联系起来。如果建立不良后果责任追究体制，人们就会改进自己的过程以产生更好的结果。经过几年的努力，我们也许能够提出一套被普遍接受的工程实践。到那时，再考虑软件工程师职业化将是合适的。

第11章

计划测试策略

测试计划是指导自己测试过程的一套想法。我们使用测试策略这个词表示指导整个项目的测试设计。测试策略是好的测试计划的重要组成部分，是将测试与任务联系起来的桥梁。各种教科书已经大量谈论了测试逻辑和工作产品，但是都很少涉及测试策略，因此本章的重点放在测试策略。

经验274

有关测试策略要问的三个基本问题是“为什么担心？”、“谁关心？”、“测试多少？”

测试最终只有一个理由：某种重要的东西可能不正常。实行测试过程，就是要找出、调查并报告产品失效的风险。这也就是为什么测试经理要反复反省测试策略的三个问题：

- 为什么担心？测试是昂贵的。除非测试策略解决足够重要，需要花时间测试的问题，否则测试策略中不要包含活动。
- 谁关心？测试理由并不是自然规律，其根源在于重要人物的感觉和价值观。只在测试策略中包含与他们的利益有关的活动。
- 测试多少？有些策略说起来容易做起来难。“我们要测试所有打印机功能组合”是一句很简单的话，但是需要1千（或数百）个测试。到底打算实际测试多少？

经验275

有很多种可能的测试策略

测试策略就是一组选择。可以有很多种选择。以下是几种不同的测试策略：

- 我们经过简单的内部评审，找出所有特别明显的问题之后，将产品发放给友好的用户。这些友好用户会实际使用该产品、并告知希望项目团队做哪些修改。
- 我们定义以用户与产品交互动作序列表示的测试用例，这些测试用例合在一起，代表预期一般用户使用产品的各种方法。还在此基础上补充压力测试和异常使用测试（无效数据和错误条件）。首先要做的，是发现对比特定行为的基本偏差，但是也关注该程序与用户期望冲突的方式。还要考虑可靠性，但是我们还没有确定如何最有效地评估可靠性。
- 我们执行并行探索式测试，开发和执行自动化回归测试。探索式测试是基于风险的，并根据需要按覆盖区域分配。我们每周都要重新检查分配情况。自动化回归测试关注基本功能的检验（能力测试），以提供涉及主要功能失效的早期报警系统。我们还注意利用大量随机测试的机会。

上面给出的都是测试策略。请注意，这些策略都是不同的，每种策略都有不同的重点，都说明将如何进行测试。好的测试策略会给出要完成测试的令人信服的描述和论证。就像有很多种测试过程可以描述一样，也有很多种可能的测试策略。

请注意，以上每种策略都相当通用。对于实际项目，我们会运用自己关于该产品的具体知识，设计出针对性更强的测试策略。不过即使是这些通用策略也能够说明测试策略不只是测试技术清单，但是不如测试计划完整。

经验276

实际测试计划是指导测试过程的一套想法

测试计划是指导将要做什么的所有想法。有这样的想法很重要。是否记录，以及怎样记录这些想法则完全是另一个问题。

当听到有人坚持认为创建要做的事的文档非常重要时，请注意在“创建”和“文档”之间少了形容词：应该是好文档还是差文档？很多人都赞同，不创建差的文档很重要。问题是：好文档很难写，维护起来成本和难度都很高。

有很多测试文献指出，从根本上说，“不写测试计划就做不好测试。”根据我们的经验，这条建议的主要积极作用，就是扩大纸张和油墨制造商的产品销售。编写很差的测试计划实在太多了。我们也看到过很多没有遵循书面测试计划就完成得很好的测试。该是给出更好建议的时候了。

不要把自己的测试计划内容与沟通和管理该计划的方式混淆起来。除了正式的书面计划之外，还有很多选择：口头计划、列在白板上的计划、篇幅只有一页纸的计划、一系列电子邮件、一组大纲或问题清单。做能够完成任务的事。

经验277

所设计的测试计划要符合自己的具体情况

测试计划的一种直观表示如图11-1所示。这是Satisfice语境模型（Satisfice Context Model）。五角形各个角上的圆圈表示测试小组的具体条件：资源和约束。五角形的中央表示测试小组的选择。测试计划的目标，是所选的测试过程能够使测试控制在项目环境中，同时又能充分利用资源，完成自己的任务。

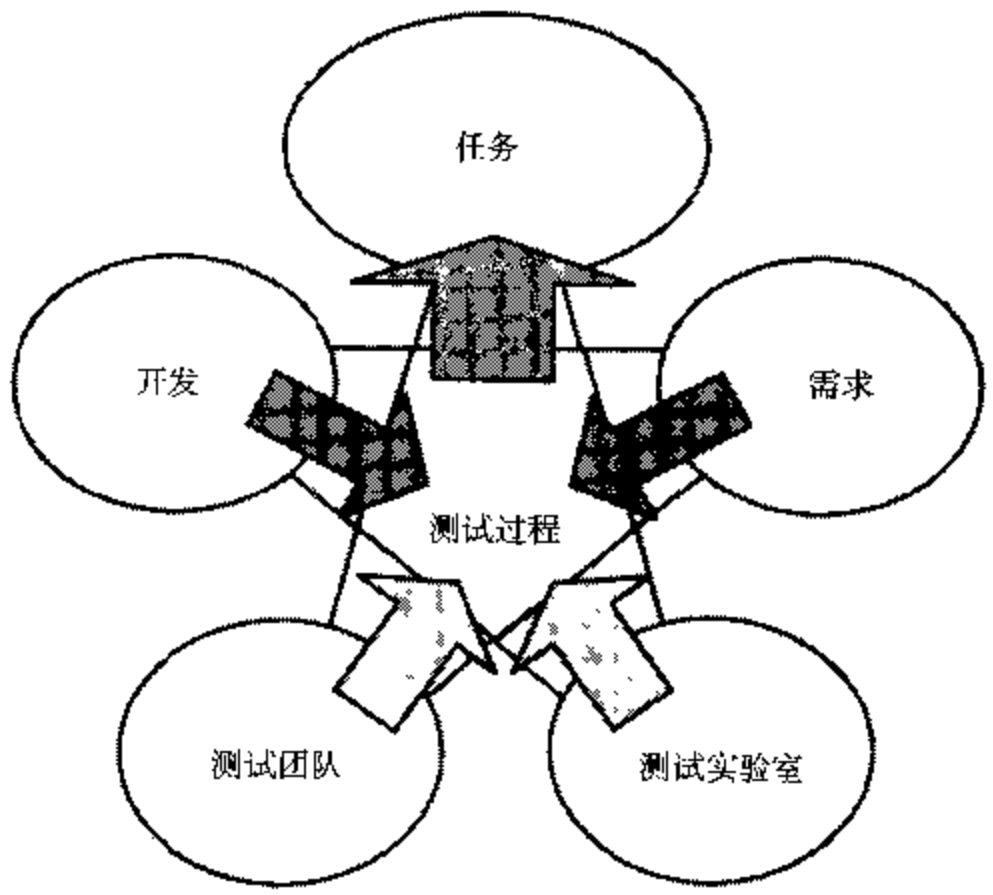


图11-1 Satisfice语境模型

给定五种资源和约束是：

- 开发。产生将要测试的产品的系统。如何接收该产品？该产品的可测试性如何？
- 需求。成功产品的评判准则。该产品的风险是什么？有关质量谁的意见最重要？
- 测试团队。能够投入该产品测试的人员。有合适的人选吗？能够及时完成任务吗？
- 测试实验室。使测试团队能够完成测试任务的系统、工具和材料。有合适的设备吗？程序错误跟踪系统的状态是否良好？
- 任务。测试团队必须要按照客户认可的成功标准解决的问题。快速找出重要问题？对质量做出准确评估？

测试经理可以协商争取更好的资源和约束。可以录用员工，或劝说程序员编

写更具可测试性的产品。但是，测试经理不要指望在这些问题上有很大控制能力，测试小组的控制能力在于如何应对这些资源和约束：自己要有什么样的测试策略、保障条件和工作产品？

利用测试计划描述在测试策略、保障条件和工作产品上所做的选择

好的测试计划，不管是不是书面的，都要描述有关测试过程的一套选择。测试计划必须描述三类选择：

- 策略。如何测试产品以快速找出重要问题？需要对哪些部分进行特殊测试？要运用什么手段创建测试？当程序错误出现时怎样识别？测试策略要规定测试项目与测试任务之间的关系。
- 保障条件。如何利用资源实现测试策略？谁来测试？什么时候测试？要想成功需要什么条件？
- 工作产品。怎样向客户提供工作产品？如何跟踪程序错误？需要编写什么测试文档？需要编写什么报告？

如果没有在测试计划中明确地做出这些选择，则要通过其他方式隐含地做出。测试小组不可能不做出这些选择，除非拒绝测试。

不要让保障条件和工作产品影响实现测试策略

测试策略常常被测试计划其他部分掩盖。我们看到过的测试计划文档详细给出进度计划、团队以及要产生的测试文档等大量信息，但是几乎没有谈如何测试该产品。这看起来是个缺点。这意味着除了要让测试员在合适的时间和场所坐在键盘前之外，没有什么要测试的。如果测试经理不告诉测试员该怎样测试，同事们就不会认为测试经理懂测试。

如何利用测试用例

如果拿出公司的所有箱子堆起来，并不会知道箱子所装东西的价值。如果公司有37只箱子，总重量是384磅，这能从什么方面说明公司的未来吗？不能。但是这些箱子所装的东西可能与公司的未来密切相关。回答这个问题的惟一方法，就是打开箱子，查看所装的东西。

测试用例就像是箱子。只是统计箱子个数而不管其中的内容是没有意义的。统计测试用例的通过与未通过比例说明不了任何问题：90%的通过率到底是好还是不好？如果不掌握测试内容的详细信息，谁都不能回答这个问题。统计所实现的测试用例与计划实现的测试用例比例也说明不了任何问题：也许最困难的测试用例被推到最后，最后10%的测试用例需要50%的时间完成。也许所计划的测试用例数还根本不足以覆盖重要的风险。

有时测试经理也赞同我们的这种观点，但是认为自己没有其他选择。当然有一种其他选择：不统计。我们认为，知道得很少但是正视这种现实，要比知道得很少但是假装知道得很多好。还有一种选择：讨论风险和覆盖率。换句话说，就是讨论测试用例的内容[⊖]。

测试员使用不合格的、未经过解释的测试用例指标向客户说明测试范围和完整性，不管是否是故意的，都是欺骗行为。

测试策略要比测试用例重要

测试策略包含要执行测试用例背后的理由。当被问及测试策略时，看起来最好的回答是指着测试用例说："这500个测试用例就是这个产品的测试策略。"这也许是一种准确回答，但是没有多少意义。这个回答包含的信息过多，但没有说明这些测试用例是否能够完成测试任务。应提供有意义的信息：总结产生测试用例的手段和目标概述。

测试策略要解释测试

在心中形成一种清晰的测试策略，并且在按照该测试策略执行测试之后，可以向任何关心这个问题的人快速、令人信服地解释自己的测试过程。这会赢得

⊖ Johanna Rothman认为我们的观点太极端。她认为："测试小组的另一种选择是讨论这些统计数字的含义。例如，当测试用例通过率从98%降到30%，这时项目才刚开始，应该为此感到担忧吗？大多数人都不会担忧。但是如果还有一周就要交付而进行β测试了，或更糟的是，就要交付产品了，这时大家都会担忧。为什么？在项目初期，一般不会运行很多测试用例。到项目快结束时，如果不是全部，一般也会运行大多数测试用例。可以利用这些数字讨论为什么预期没有达到。也许没有足够人手执行所有的测试，或编写错误报告、或检验已发现的问题是否已经解决。可以利用这些数字说明自己的考虑。"

Jeff Bleiberg写道："一般来说，需要使测试过程具有可视性。指标可以做到这一点。但是我发现不管我们认为指标多么'健壮'、多么能够'自我解释'，人们仍然会错误地解释。因此，我的一个原则就是，从来不散发报告，而是在会上拿出报告、并讨论所使用的指标及其含义。"

测试小组其他人对该测试过程的支持。如果自己不清楚自己的测试策略，那么对测试策略的解释也就更可能是模糊的，缺乏说服力的。

好的测试策略是：

- 与具体产品有关。不管通用测试策略有多么好，与当前具体产品和技术有关的测试策略总会更好。
- 关注风险。显示测试过程可以怎样描述重要问题。将测试过程与这个项目的任务关联起来。
- 多样化。在大多数情况下，多样化的测试策略优于单调的测试策略。多样化测试策略要包含各种不同的测试手段或方法。逃脱一种测试方法的问题还可能被另一种方法捕获。
- 实用。测试策略必须能够被执行。不要提出远超出项目团队能力的测试策略。

运用多样化的折衷手段

不太彻底但是更具多样性的测试策略，要优于更彻底但是不具多样性的测试策略。换句话说，执行达到相当水平的多种不同测试，要优于完美地执行一两种测试。我们把这种原则叫作多样化的折衷手段（diverse half-measures）。

这种测试策略原则来源于软件产品的结构复杂性。进行测试，就是对复杂空间采样。没有单一测试手段能够以快速发现所有重要问题的方式对这种空间采样。任何给定测试手段开始可能会发现大量问题，但是发现率曲线会逐渐走平。如果转而使用对不同种类问题敏感的测试手段，则发现率会重新上升。从问题发现率角度看，采用每种测试手段在发现率开始降低时，就转而使用一种新的手段。

多样性的另一个目的来源于一个迷题：在什么情况下会出现测试某个产品数月后交付用户，而用户在第二天就发现了测试员还不知道的大问题？有一些问题会使这种情况出现。一种主要原因就是很窄的视野。并不是测试员没有进行充分测试，而是没有执行合适种类的测试。我们曾经见过有公司运行数以十万计的测试用例，仍然会遗漏显而易见的问题，因为他们的测试缺乏多样性。

为了保证多样性，可使用五要素测试系统（第3章介绍的分类系统）从所有五种要素中选择手段。多样性可最大化发现率，最小化忽视重要问题的机会。

充分利用强有力测试策略的原始材料

可以用来执行测试策略的资源使测试策略成为可能。在整个项目开发过程以

及所有项目中，都要充分利用这些资源，以使策略选择达到最大化。以下列出其中的部分资源：

- 测试员运用各种测试手段的技能。
- 测试员有关产品内部技术的知识。
- 具有特殊测试或工艺技能的朋友。
- 原始测试数据库。
- 各种测试平台，包括多种操作系统和硬件配置。
- 各种测试工具。
- 实际用户数据。
- 植入产品中的可测试性功能（例如日志记录文件、判断和测试菜单）。

项目的初始测试策略总是错的

随着对被测产品及其失效模式认识的不断深入，测试策略也应该进化。我们建议要根据风险确定测试策略。这会带来一个问题：不知道有什么风险。在项目开始时，测试员知道的只是在哪里会有很多错误的传言。如果幸运，还可以运用有理论指导的猜测。因此，项目的初始测试策略很可能出现以下两个问题中的至少一种：没有关注风险，或关注的是看起来有风险但实际上没有的部分。

应避免过早固定一种且只有一种测试策略，这样可解决这些问题。随着对被测产品认识的不断深入，随着逐渐了解产品的弱点在哪里，随着想出测试该产品的新方法，测试策略也应该进化。

像V字模型这样的项目生命周期，都要求测试员在项目的一开始就知道应该采用什么测试策略。由于要想在项目初期就拿出合适的测试策略，测试员必须不仅是天才，而且还得是能够看透别人心思的天才，因此我们认为V字模型不是组织项目的好方法。如果读者坚持使用V字模型，可以考虑在项目初期制定测试计划，以后当所有人都不再注意问题，并一门心思努力按时交付好的产品时，再悄悄用新的和更好的测试来替换。

在项目的每个阶段，可自问“我现在可以测试什么，能够怎样测试”？

有时我们发现有人声称处于某个开发阶段的项目，例如系统集成，在该阶段只能进行特定类型的测试，例如基于需求的测试。这就像劳动节后不能穿白色服装那样令人困惑。为什么要遵循这种通用并且无所不包的规则呢？

测试策略要考虑进行测试的项目开发阶段以及测试结构层次（单元、子系统或系统），但并不是决定性的考虑因素。我们建议测试经理可以在任何开发阶段自问："我们在此时此地可以测试什么？怎样才能测试好？"

不要认为特定手段只在一定的开发阶段才是有用的。要使自己的测试策略能够抓住更多的机会。在任何时候，测试任何值得测试的东西，使用任何能够最好地满足客户要求的手段。

根据产品的成熟度确定测试策略

虽然通用项目开发阶段测试策略是不充分的，但是根据产品的成熟度确定不同的测试策略是很有意义的。

- 项目初期，同情地测试。在项目开发的初期，项目团队的工作还没有全面走上正轨，测试员对产品也了解不多。在这个阶段大可不必进行生硬的测试，因为即使简单的测试也会发现问题。此外，程序员也知道产品还不成熟，很担心测试员对新生产品过于不满。程序员想知道的是刚刚实现的功能是否基本上能够运行。
- 项目中期，积极地测试。随着产品逐渐成型，主要功能已经实现，并投入试用，简单测试已经没有什么作用了。程序员对产品也更有信心，已经把工作重点从功能设计和实现，转移到全时程序错误修复。现在可以实施要求更严格、更复杂的测试。层层深入测试产品值得测试的各个部分，尽可能多地发现并报告错误。可建立错误库供开发人员检查。

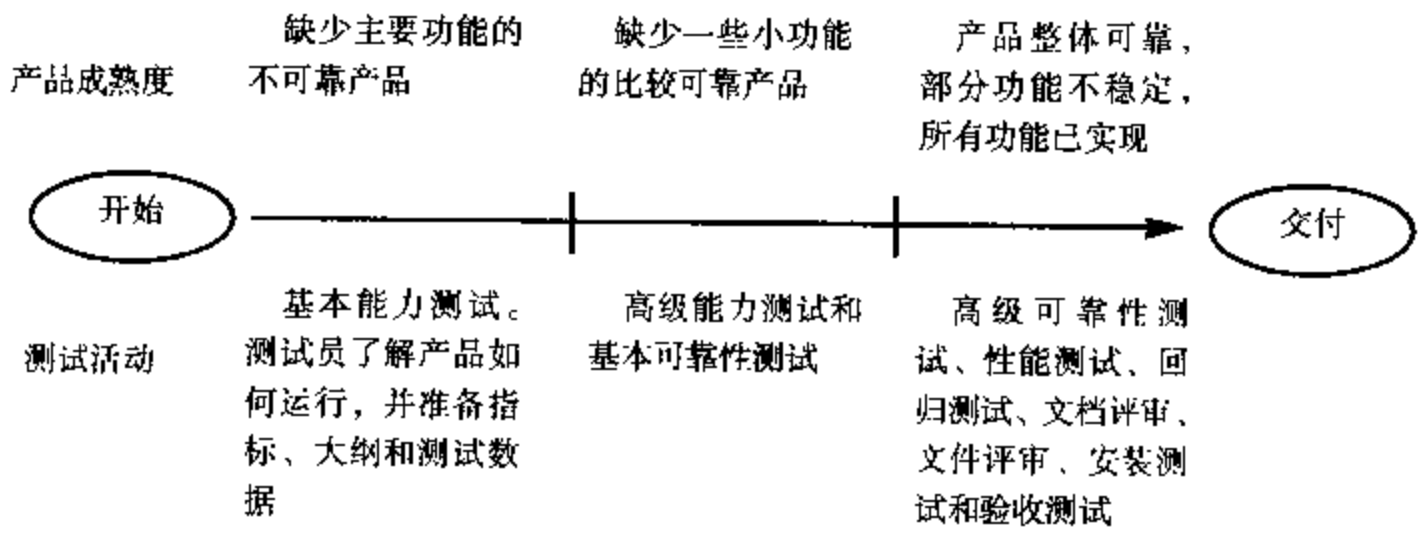


图11-2 产品周期

- 项目末期，多样地测试。找出成熟产品中的错误要更困难一些，因此需要更多的创造性。现在可以充分发挥自己的想像力和管理层所提供的支持，推行多样化测试。利用帮助人员、自动化测试工具、特殊测试活动（查错

奖励）、启发式测试、β测试人员——任何手段、所有手段。如果测试经理善于运用这些手段，错误发现率就会接近如图11-3所示的理想曲线。积极测试可提高错误发现率曲线，而使测试多样化、继续多样化可使曲线保持较高水平，直到想不出新的、更好的测试。

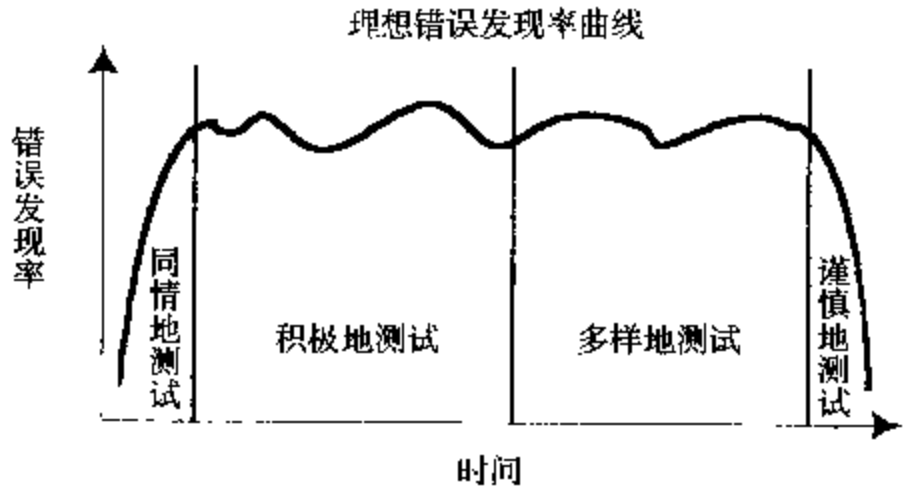


图11-3 持续积极测试的理想结果

- 项目最后，谨慎地测试。在最后的日子犯错误会为公司带来巨大损失。随着产品交付日期的邻近，测试的重点应该更具防御性。要仔细测试每个变更，检查这个版本要交付的所有文件。利用成对测试为每个测试再增加一层保障。

总体目标是随着产品开发的进展，不断调整测试策略，使得在产品开发整个过程中，重要错误的发现率都保持比较高的水平。

利用测试分级简化测试复杂性的讨论

为了在测试策略中简化测试复杂性的描述，很多测试项目团队都发现区分测试级别会有帮助。较低级别的是比较简单、功能较弱的测试，较高级别的是比较复杂、功能较强的测试。通过直接谈论测试类可简化测试策略的讨论。以下是测试分级结构的一个例子：

- 0级，冒烟测试。显示产品已经可以进行独立测试的简单测试。如果0级测试失败，可把产品打回程序员。
- 1级，能力测试。检验产品每个函数能力的测试。1级测试的目标，是保证每个函数都能够执行其任务。1级测试应该避免采用曲折的场景、富有挑战性的数据和功能交互。
- 2级，函数测试。检查产品各个个体函数和子函数的能力和基本可靠性。数据覆盖和复合测试结果评估方法是有意义的。使用边界、压力和错误处

理测试，避免采用曲折的场景和功能交互。

- 3级，复合测试。包含多组函数之间交互和控制流，以构成复杂场景的测试。评估的重点已经扩展，可能要包括性能评估、兼容性、资源紧张程度、内存泄漏、长时间可靠性或其他类型的质量评判准则，或由于现在产品更成熟，所有能够进行这些内容的测试。

这四级测试中的每一级，可能对应一些不同的测试技术或技术组合。总体思路是首先宽泛、同情地测试，随着测试不断成熟，逐步进行深度和边缘测试。对产品的早期版本执行第3级测试，而不首先执行第1、2级测试，可能会使程序员感到很反感。更有可能的是，根本不能执行这些测试。

测试灰盒

即使并不完全掌握待测产品内部情况，部分基于产品内部结构的测试策略也是很好的想法。我们把这种情况叫作灰盒测试。灰盒测试的概念很简单：如果了解一些产品内部工作情况，就能够从外部进行更好的测试。灰盒测试与白盒测试不同，白盒测试需要了解产品内部细节。在灰盒模型中，要从产品的外部测试，就像黑盒测试一样，但是所选择的测试反映出测试员对内部组件操作和交互的了解。

灰盒测试对Web和因特网应用程序尤其重要，因为因特网是通过松散集成起来的组件构建起来的，这些组件要通过相对定义完备的接口连接。除非了解因特网的体系结构，否则测试会很肤浅。Hung Nguyen的《测试Web应用系统》（Testing Applications on the Web）（2000）是用于Web的灰盒测试策略的一个很好例子。

在重新利用测试材料时，不要迷信以前的东西

在重用测试用例或任何测试材料时，不要将其当作黑盒使用。要先做一些了解，思考测试材料为什么采用这种方式设计。在很多情况下，已经归档的测试材料的质量很差，以致于重新编写新材料要比重用老材料好得多。很多测试员都倾向于相信老的测试材料，只是因为材料是老的，并可能有些神秘。我们经常发现老的测试材料虽然已经过时（有时已经是多年前的材料），却仍然被拿来进行例行测试。我们见到过一些聪明、熟练的测试员毫不怀疑地使用技能差很多的测试员所编写的测试用例。这是重用不好的一面。

我们知道的一个测试实验室有一条始终不变的规则“永远要在Compaq Presario机上测试，因为这种计算机的不兼容是众所周知的。”令人奇怪的是，这条规则是1996年订的。它今天还有效吗？有人重新考虑过这个问题吗？有人对以前提出这条规则的人的权威性提出过疑问吗？

如果对编写将被重用的测试用例想法感兴趣，可以自问别人怎么会知道该测试用例的含义，编写这个测试用例的理由，以及这个测试用例该在什么时候退役或修改。否则，别人很可能会把这个测试用例当作具有魔力的图腾，而不是按照原始创建者的意图使用。

两个测试员测试同样的内容也许并不是重复劳动

有些测试员担心可能出现重复劳动。放心，不要担心别人在同一个测试任务上重叠，或以同样的方法测试相同的产品。重复测试工作与重复测试不是一回事。两个测试员进行测试很有可能会发现不同的问题。这可能是因为两个测试员并非运行同样的测试用例。即使两人认为自己执行的是同样测试，两人肯定也会存在差异。此外，一个测试员可能会注意到被另一个测试员忽视的问题。这种情况很常见。

重复测试劳动几乎都不会是浪费。真正的问题不是浪费，而是产品的某一部分是否值得进行重复测试。

设计测试策略时既要考虑产品风险，也要考虑产品要素

好的测试策略不仅要根据产品风险制定，还要考虑产品内部要素。以下是一些基于项目的测试策略制定原则：

- 不要在测试员之间的缝隙中遗漏错误。除非要采用多样化折衷测试手段，或指派重叠测试，否则会面临测试遗漏的真正风险，这些遗漏会出现在两个测试员（或小组）任务之间的边界处。
- 经常测试客户要求测试的内容。测试员是在代表很多客户进行测试。他们认为测试员应该怎样测试？测试员要弄清他们的要求并至少完成一部分所要求的测试。
- 偶尔测试客户要求不要测试的内容。有时客户要求不要测试产品的特定部分。这是个微妙问题，我们也不能告诉读者该怎样做。但是，有时被要求不要测试的正是最需要测试的。

- 测试不够清晰和矛盾的内容。文如其人。任何地方有模糊和矛盾，就会有很多错误。如果程序员不太肯定某个功能应该完成什么任务，就测试这个功能。如果程序员是第一次使用这种技术，就测试这种技术。如果有两位程序员编写相互之间有接口的单元，就测试这个接口。测试员是不会一无所获的。
- 不要痛打落水狗。如果已经清楚某个功能看起来有很多错误，就不要继续测试了，除非要和开发人员一起检查。这可能是坏的版本或坏的配置。此外，如果组件的问题太严重，可能就要替代，而不是修改，所发现的所有错误都被一起归档，因此不必再费劲测试。
- 更多变更意味着更多测试。从理论上说，产品中最微小的变更也会产生大的全局效果。这意味着任何变更都可能潜在地使已经完成的该产品测试无效。在实践中，大多数变更都只有相当局部的影响。但是，测试员肯定必须测试所做的变更。产品中的变更越多，必须执行的测试越多。到项目快结束时，这部分的测试工作量很大。

把测试周期看作是测试过程的韵律

测试是按周期进行的。测试周期从一个版本开始，以以下两种情况之一结束：下一个版本完成，或确认进一步测试不再有意义。测试策略要通过测试周期来具体化。计划测试周期，以尽快向客户提供最佳信息。以下是组织测试周期的一种方法：

1. 接收产品。保证得到的是正确版本。

2. 对测试系统进行配置。清理测试系统。将磁盘恢复到原来的状态，或完全卸载产品的以前版本。

3. 检验可测试性。这是好版本吗？这个版本值得花时间测试吗？首先运行冒烟测试：说明该版本包含了所有主要功能，并且基本上能够操作的简单测试。

4. 确定哪些部分是新增加的或经过修改的。编写了哪些扩展或修改了产品的能力的新代码？

5. 确定修改了哪些程序错误。还要检查被拒绝的问题，并做出相应的响应。

6. 测试程序错误修改。趁程序员还能记得住，首先测试程序错误修改。如果程序错误修改没有成功，程序员都想迅速知道。

7. 测试新的或经过变更的部分。程序员头脑中最关心的下一个问题可能就是新代码了。

8. 测试其他部分（请注意，首先测试风险较大的部分）。下面测试所有其他

（有意义的）部分，直到时间用完，或完成了足够的测试。如果有自动化的回归测试包，应该运行。

9. 报告测试结果。在整个测试周期内，应该定期报告测试结果，至少每天报告一次。

如何制定语境驱动的测试计划

本指南旨在帮助读者制定测试计划。请注意，真正的测试计划是实际指导自己实施测试的一套想法。不管读者是否制定书面测试计划，我们设计的这个指南都会有所帮助。

本指南并不是一种模板，不是供读者填写的表格，而是一组旨在帮助读者思考的思想，用于降低读者遗忘重要内容的可能性。我们使用的是简洁语言和描述，有可能不太适合测试新手。本指南主要向有经验的测试员或测试组长提供支持。

以下分七个任务主题。这些主题没有一定顺序。实际上，读者可以按任何顺序阅读　只是需要注意，测试计划的质量与是否很好地执行了任务以及是否很好地考虑了像这里提出的问题相关。状态检查部分有助于读者确定是否制定了足够好的测试计划。但是我们建议读者要在整个项目开发过程中，重新检查并修改测试计划（至少要在心中修改）。

1. 监视影响测试计划的主要问题

确定影响制定实用、有效的测试策略中时间、工作量或可行性要素的风险、障碍或其他挑战。要把握计划的整体作用。在整个项目开发过程中，全程监视这些问题。

状态检查

- ❑ 是否有要满足的特别关键或很难度量的产品质量标准？
- ❑ 产品是否复杂或很难学会？
- ❑ 测试员是否需要特殊培训或工具？
- ❑ 是否有很难得到或配置的部分测试平台？
- ❑ 是否将测试未集成或半可操作的产品组件？
- ❑ 是否存在具体的可测试性问题？
- ❑ 项目团队是否缺乏产品设计、技术或用户群的经验？
- ❑ 测试是否必须很快开始？
- ❑ 是否有制定测试计划所需的信息还没有收集到？

- ❑ 是否能够评审被测产品的某个版本（甚至是演示版、原型版或老版本）？
- ❑ 是否有足够的难以录用或组织的测试人员？
- ❑ 是否必须遵循自己所不熟悉的测试理论？
- ❑ 项目计划的制定是否没有考虑测试需要？
- ❑ 计划是否要经过漫长的协商或批准？
- ❑ 测试员是否远离客户？
- ❑ 项目计划是否经常变动？
- ❑ 计划是审计的一个内容吗？
- ❑ 客户是否说不出测试员能够为他们做什么？

2. 明确任务

本节给出的任何一部分或全部目标都可能是具体测试任务的一部分。有些任务比另外一些更重要。根据对具体项目的了解，为这些目标排队。对于所有适用的目标，找出可以用来评判的具体的成功指标。

需要考虑的任务要素

- ❑ 快速找出重要问题。
- ❑ 进行综合质量评估。
- ❑ 确认产品质量是否达到具体标准。
- ❑ 尽可能缩短测试时间或降低测试成本。
- ❑ 尽可能提高测试效率。
- ❑ 就提高质量或可测试性问题，向客户提出建议。
- ❑ 就如何测试向客户提出建议。
- ❑ 保证测试过程是可以充分说明的。
- ❑ 严格遵循特定的方法或指示。
- ❑ 使特定的项目相关人员感到满意。

可能的工作产品

- ❑ 说明测试任务的简短电子邮件。
- ❑ 一页纸篇幅的测试要求。

状态检查

- ❑ 是否知道谁是自己的客户？
- ❑ 关键人物是否赞同测试任务？
- ❑ 测试任务是否足够清晰，以作为制定计划的基础？

3. 分析产品

了解被测产品及其内部技术。了解如何使用被测产品。需要深入下去。随着对产品了解的深入，测试会变得越来越好，因为自己越来越接近成为产品专家

分析什么

- ❑ 用户（用户是谁，他们的职业是什么）。
- ❑ 结构（代码、文件等）。
- ❑ 功能（产品做什么）。
- ❑ 数据（输入、输出、状态等）。
- ❑ 平台（外部硬件和软件）。
- ❑ 运营（产品是用来完成什么任务的）。

分析方式

- ❑ 执行探索式测试。
- ❑ 评审产品和项目文档。
- ❑ 与设计人员和用户面谈。
- ❑ 与类似产品进行比较。

可能的工作产品

- ❑ 测试覆盖大纲。
- ❑ 带注释的规格说明。
- ❑ 产品问题清单。

状态检查

- ❑ 设计人员赞同产品覆盖大纲吗？
- ❑ 设计人员认为测试员理解了产品吗？
- ❑ 测试员能够可视化产品并预测产品行为吗？
- ❑ 测试员能够产生测试数据（输入和结果）吗？
- ❑ 测试员能够配置并操作被测产品吗？
- ❑ 测试员理解产品将被怎样使用吗？
- ❑ 测试员是否发现设计中的不一致问题？
- ❑ 测试员是否找出显式和隐式规格说明？

4. 分析产品风险

被测产品可能怎样以一种重要方式失效？开始测试员最多也只会有一个一般想法。随着测试员对产品了解的深入，测试策略和测试会变得越来越好，因为对被测产品的失效机理了解得越来越多。

分析对象

- ❑ 威胁（具有挑战性的条件和数据）。
- ❑ 脆弱性（在什么地方可能失效）。
- ❑ 失效模式（可能的问题种类）。
- ❑ 失效影响（问题的严重程度）。

分析方式

- ❑ 评审需求和规格说明。
- ❑ 评审实际失效。
- ❑ 与设计人员和用户面谈。
- ❑ 对照风险启发和质量评判大纲评审产品。
- ❑ 找出一般问题和失效模式。

可能的工作产品

- ❑ 组件/风险矩阵。
- ❑ 风险清单。

状态检查

- ❑ 设计人员和用户对风险分析认可吗？
- ❑ 测试员能够找出所有重要的问题种类吗？这些问题都应该在测试期间出现吗？
- ❑ 为了尽可能提高测试效果，测试员知道该把测试工作集中到哪些对象上吗？
- ❑ 设计人员是否采取措施使重要问题更容易被检测，或降低发生的可能性？
- ❑ 测试员如何发现自己的风险分析是否准确？

5. 设计测试策略

为了根据已有的产品最佳信息快速、有效地测试，测试员可以做什么？首先尽可能做出最好的决策，同时又要让测试策略能够在项目整个开发过程中改进。

考虑五方面的手段

- ❑ 以测试员为核心的手段。
- ❑ 以覆盖率为核心的手段（结构覆盖率和功能覆盖率）。
- ❑ 以问题为核心的手段。
- ❑ 以活动为核心的手段。
- ❑ 以评估为核心的手段。

计划方式

- ❑ 针对风险和产品域确定手段。
- ❑ 可视化具体和实用手段。
- ❑ 使测试策略多样化、尽可能减少遗漏重要问题的机会。
- ❑ 寻找通过自动化测试扩展测试策略的途径。
- ❑ 不要计划得过死，使测试员能够发挥自己的才智。

可能的工作产品

- ❑ 逐项列出的每条所选测试策略以及如何运用的说明。
- ❑ 风险/任务矩阵。
- ❑ 所选测试策略固有的问题或挑战清单。
- ❑ 针对没有充分覆盖的产品部分提出的建议。
- ❑ 测试用例（仅当需要时）。

状态检查

- ❑ 客户认同测试员制定的测试策略吗？
- ❑ 测试策略给出的所有内容都是必要的吗？
- ❑ 测试策略是否能够实际贯彻？
- ❑ 测试策略是否过于通用？可以容易地用于任何产品吗？
- ❑ 是否还有不准备测试的任何重要问题？
- ❑ 测试策略利用了可用的资源和帮助者吗？

6. 条件计划

测试经理将如何实现测试策略？测试策略会受到条件约束或指示的很大影响。努力争取所需的资源，并尽量利用可用的所有资源。

保障条件方面的问题

- ❑ 测试工作量估计和进度安排。
- ❑ 可测试性宣传。
- ❑ 测试团队力量（合适技能）。
- ❑ 测试员培训与管理。
- ❑ 测试员任务分配。
- ❑ 产品信息收集与管理。
- ❑ 项目团队会议、沟通和协同。
- ❑ 与项目团队所有其他小组，包括开发小组的关系。
- ❑ 测试平台的获得和配置。

- ❑ 约定和协议。
- ❑ 测试工具和自动化测试。
- ❑ 插桩和模拟需要。
- ❑ 测试包的管理和维护。
- ❑ 构建和传送协议。
- ❑ 测试周期管理。
- ❑ 错误报告系统和协议。
- ❑ 测试状态报告协议。
- ❑ 代码冻结与增量测试。
- ❑ 项目最后的压力管理。
- ❑ 测试停止协议。
- ❑ 测试效果的评估。

可能的工作产品

- ❑ 问题清单。
- ❑ 产品风险分析。
- ❑ 责任矩阵。
- ❑ 测试进度计划。

状态检查

- ❑ 项目团队的保障条件是否支持已制定的测试策略?
- ❑ 是否存在阻碍测试的问题?
- ❑ 测试条件和策略是否能够修改，以适应可以预见的问题?
- ❑ 现在是否可以开始测试，以后再解决其余问题?

7. 共享测试计划

测试员并不孤独。测试过程必须服务于项目团队。因此，要吸收项目团队成员参加测试计划制定。不必夸大这个问题，至少要与团队的关键成员讨论，从而得到他们的理解和隐含的支持，以争取实现测试计划。

共享方式

- ❑ 吸收设计人员和项目相关人员参加测试计划制定过程。
- ❑ 积极征求有关测试计划的意见。
- ❑ 尽自己所能帮助开发人员获得成功。
- ❑ 帮助开发人员理解他们的行为会对测试产生的影响。
- ❑ 与技术文档编写员和技术支持人员就分享质量信息进行沟通。

- ❑ 请设计人员和开发人员评审和批准参考材料。
- ❑ 记录和跟踪约定。
- ❑ 请别人分段评审测试计划。
- ❑ 通过减少测试计划文档中不必要的文字，来改进文档的可评审性。

目标

- ❑ 对测试过程的一致理解。
- ❑ 对测试过程的一致承诺。
- ❑ 测试过程的合理参与。
- ❑ 管理层对测试过程的合理预期。

状态检查

- ❑ 项目团队是否关注测试计划？
- ❑ 项目团队，特别是一线管理人员是否理解测试小组的角色？
- ❑ 项目团队是否感觉到测试小组关心项目团队的最佳利益？
- ❑ 测试小组和项目团队其他小组之间是否有对立或积极的关系？
- ❑ 是否有人认为测试员没有将注意力集中到重要的测试上？

这个测试计划有多好？

“这个测试计划有多好？”这个问题的答案只能参考测试计划应该是什么样而得出。虽然有一些描述测试计划文档格式的公开标准，但是这些标准并没有提供多少区分好计划和坏计划的准则。本模型确定基本概念、测试计划所提供的功能、测试计划应该满足的准则，以及辅助确定功能是否满足了准则的一些直观推断。

术语与概念

- ❑ 测试计划。测试计划是指导或表示测试过程的一组想法。这些想法常常只部分地形成文档，散布在多个文档中，并且随着项目开发的展开而变化。
- ❑ 测试计划文档。测试计划文档是用来描述测试计划想法的任何文档。但是，测试计划文档并不是有关测试计划的惟一信息源。测试计划信息还包含在项目团队的口头讨论惯例和公司文化中。
- ❑ 测试策略。测试策略描述测试项目和测试任务之间的关系。测试策略说明要测试什么，怎样测试。测试策略与实现该策略的保障条件不同。
- ❑ 测试保障条件。实现测试策略并提交测试结果的手段。测试保障条件包括诸如谁、在哪里、什么时候完成测试这样的详细信息，以及将要使用

的支持材料。

- 测试过程。过程有很多含义。在本文档中，我们使用“测试过程”表示测试是如何实际展开的（而不是预期怎样展开，或文档说该怎样展开）。

测试计划的功能

测试计划的功能，是测试计划预期能够帮助测试员完成的工作。以下列出的是理想测试计划应该提供的功能。但是，测试计划文档可能只描述这些功能的一个子集，其他功能在其他文档中描述，或直接由测试经理或测试员个人掌握，而不依靠任何文档的支持。因此，我们应该结合测试计划所提供的功能，或通过其他方式没有充分提供的功能来对它进行判断。

- 支持产品开发的质量评估，从而提供明智、及时的产品决策。
- 结合产品需求和产品风险描述并论述测试策略，增进人们对测试策略优点和局限性的认识。
- 描述并论述为了推进测试项目必须满足的所有特殊需求或进入条件，以及退出条件或确定什么时候停止测试的过程。
- 支持测试项目的发起和组织，包括准备、人员召集、责任委派、设备获取、任务计划和进度安排。
- 支持测试项目和测试策略的日常管理和评估。
- 支持测试小组成员之间，以及测试小组与项目团队其他小组之间的有效协同、协作和其他关系。
- 找出并管理可能影响项目的任何风险或问题。
- 描述测试项目的可交付产品和交付过程。
- 记录历史信息以支持过程审计、过程改进和未来的测试项目。

测试计划质量准则

测试计划是否很好地支持其各种功能？以下列出的是有助于读者思考的准则：

- 有用性。测试计划会有效地支持其所提供的功能？
- 准确性。测试计划文档是否准确地与事实描述保持一致？
- 高效性。测试计划是否能够高效地利用已有资源？
- 可适配性。测试计划是否能够适应项目中合理的变更和不可预测性？
- 清晰性。测试计划是否自我一致并且足够明确？
- 可使用性。测试计划文档是否简练、可维护并有很好的结构？
- 兼容性。测试计划是否满足外部提出的需求？
- 依据。测试计划是否是有效测试计划过程的产品？
- 可行性。测试计划是否没有超出必须使用该计划的机构能力？

测试计划提示

为了评估测试计划，可考虑测试计划如何实现其功能以及如何满足质量准则。为此，我们建议读者利用以下提示。一条提示就是一条经验法则或有指导的猜测。我们给出的提示并不是在每种环境下都是同等重要的，有些甚至完全不适用某些特定环境。每种提示都用一种通用规则以及该规则的简要依据来描述。这些依据有助于读者确定什么时候、在什么情况下使用某种提示。

提　示	依　据
1. 快速找出重要问题。应该通过优化测试快速找出重要问题，而不是试图以同样的重视程度找出所有问题	错误修改可能是困难的，并且很费时间，可能会引入新问题。因此，尽可能快地找出问题会使开发小组有更好的机会安全地解决这些问题
2 关注风险。测试策略应该把关注的重点放在有潜在技术风险的内容上，同时也要投入一部分精力在低风险的内容，以免分析有误	完备测试是不可能的，我们永远也不会知道自己对技术风险的感觉是否完全准确
3. 尽可能提高多样性。测试策略在测试手段和观察方法上具有多样性。测试覆盖率的评估方法应该考虑多种覆盖要素，包括结构、功能、数据、平台、操作和需求	没有单一的测试手段能够线性地发现所有重要问题。我们永远也不能确保找出所有重要问题。多样性可以最大限度地降低测试策略被局限于特定类型问题上的风险
4 避免编写过死的脚本。避免预先详细确定测试用例，除非有具体、重要的理由需要这样做。测试策略应该包含合理的变化，并充分利用测试员的能力根据具体情况进行分析，将关注点集中到重要但是没有预料到的问题上	过死的测试策略能够提高发现特定问题子集的机会，但是对于复杂系统，它会降低发现所有重要问题的可能性。测试中的合理可变性，例如交互测试、探索式测试，都会意外地提高测试覆盖率，同时又不会显著牺牲基本覆盖率
5. 根据需求测试。测试隐含的需求是很重要的，即要测试需求含义的全部外延，而不只是字面上的需求。为什么要提出每个需求？要找出原因，根据需求的内涵精神测试，而不仅是字面的意思	只测试明确形成书面文档的需求不会发现所有重要问题，因为已定义的需求一般来说都是不完备的，并且自然语言有其固有的模糊性。很多需求也许并没有形成书面文档
6. 我们并不孤独。测试计划必须促进与项目团队其他小组的协作，尤其是编程、技术支持和技术文档编写小组。只要有可能，测试员还应该与实际客户和用户协作，以便更好地理解他们的需求	其他小组和项目相关人员常常掌握有关产品问题或潜在问题的信息，这些信息可能对测试小组很有用。他们的观察可以帮助测试员更好地分析风险。测试员还可能掌握对其他小组很有用的信息
7. 促进可测试性。与程序员联系，帮助他们构建更具可测试性的产品	测试策略达到目标的可能性，会受到产品可测试性的很大影响
8 测试计划不要太通用。测试计划应该突出测试策略和测试项目非例行的、与具体项目有关的方面	值得开发的每个软件项目都会面临特殊的技术挑战，好的测试工作必须解决这些问题。通用测试计划反映出的是很弱的测试计划过程
9. 点明即可。测试计划文档应该避免不必要的文字。不要陈述很明显的事实。要使每句话都有意义。如果要面对不同读者，可考虑为每类读者准备不同的计划版本	看起来没有必要或很明显的文字，都会降低别人阅读整篇文档的可能性。读者会认为他们已经理解了文档内容

（续）

提　示	依　据
10. 不要限制人员。测试小组应该把人员投入到适合人工完成的工作中，使用自动化测试工具用来完成适合自动化测试工具完成的工作。手工测试应该允许即兴发挥和现场思考，而自动化测试适用于需要高度可重复性、高速、不需要人工判断的测试	很多测试小组都误认为，测试员在确切地使用已描述测试脚本上发挥作用，或自动化测试工具可提高人员在测试执行过程中的价值。手工和自动化测试并不是同一种测试的两种形式，而是两类完全不同的测试手段
11. 受测试进度制约。提出和说明测试进度时，应突出与开发进展、产品可测试性、报告程序错误所需时间和项目团队对风险的评估的相关性	测试计划中的单调测试进度计划，常常说明把测试看作是一项独立的活动。测试进度只有当产品具有高度可测试性、开发是完备的并且测试过程不被经常所需的问题报告打断时，才是独立的
12. 解决瓶颈问题。测试过程应该尽可能避免占据关键路径。通过与开发并行进行测试，以比程序员修改缺陷更快的速度快速找出值得修改的问题，可以做到这一点	为了缓解截断测试过程的压力，这一点很重要
13. 快速反馈。测试员和程序员之间的反馈环应该尽可能紧凑。所设计的测试周期，应该向程序员迅速反馈有关在着手进行完整回归测试之前的最新补充和变更的信息。只要有可能，测试员和程序员都应该在比较靠近的场地工作	为了尽可能提高质量改进效率和速度，这一点很重要。这也有助于使测试离开关键路径
14. 测试员不仅仅是测试员。测试团队应该利用除正式测试之外的有关质量的信息渠道，以便帮助评估和调整测试项目。这些渠道的例子包括检查、现场测试或测试小组之外的人完成的非正式测试	通过研究由测试小组之外的各种方式收集到的产品质量信息，可以发现正式测试策略中的盲点
15. 评审文档。所有归档测试文档应该由作者之外的人评审。所使用的评审过程应该与文档的关键性相称	测试最大的职业问题是视野狭窄。评审不仅有助于发现测试设计中的盲点，而且还有助于推动关于测试实践的对话和同级教育

附　录

软件测试的语境驱动方法

我们属于有时叫作软件测试语境驱动学派的一帮人。经过多年（断断续续），我们最终开发出一种原则描述，我们相信这种描述反映了这些松散地聚合在一起充当这一派思想领导的人们的共同观点。

本书给出了语境驱动思维的大量例子，并解释了我们在软件开发中的体会。随着本书的出版，我们也创建了一个网站，即context-driven-testing.com，以进一步发展这个学派。

如果读者阅读了以下给出的原则和说明，决定也亲自加入这个学派，可访问context-driven-testing.com并加入这个集体。

语境驱动学派的七个基本原则

1. 任何实践的价值都取决于其语境。
2. 在特定语境下有好的实践，但是没有最佳实践。
3. 在一起工作的人，是所有项目语境的最重要的组成部分。
4. 项目以常常不能预测的方式逐渐展开。
5. 产品是一种解决方案。如果产品不是解决方案，它就不能发挥作用。
6. 好的软件测试是一种具有挑战性的智力过程。
7. 只有通过判断和技能，在整个项目团队中始终进行协作，才能在合适的时间做合适的事，以有效地测试自己的产品。

贯彻基本原则的说明

- 成立测试小组是为了提供与测试有关的服务。测试小组并不开发项目，而是为项目提供服务。
- 在为开发、质量管理、调试、调查或产品销售提供服务时，测试是代表项

日相关人员实施的。完全不同的测试策略对于这些不同的目标是合适的。

- 不同的测试小组有不同的任务是很正常的。服务一种任务的核心实践可能与另一种核心实践无关或生产率相反。
- 采用无效的指标是危险的。
- 任何测试用例的基本价值，在于它提供信息的能力（即减少不确定性的能力）。
- 所有征兆都会有欺骗性。即使产品看起来通过了测试，但是也很有可能以测试员（或自动化测试程序）没有监视到的方式失效。
- 自动化测试并不是自动化的手工测试：把自动化测试作为自动化的人工测试来讨论是没有意义的。
- 不同类型的测试会暴露不同类型的缺陷——随着程序变得更稳定，测试应该更具进取性，或应该关注不同的风险。
- 测试工作产品应该达到满足项目相关人员有关需求的程度。

举例

请考虑两个项目团队。一个团队开发用于飞机的控制软件。“正确行为”具有高度技术和数学含义，必须遵循FAA制定的规范。团队成员所做的一切或没有做的一切，都是从现在起20年时间内的法律诉讼证据。开发团队成员都遵循崇尚小心、准确、可重复性和反复检查每个人的工作的工程文化。

另一个团队开发的是在Web上使用的字处理程序。“正确行为”就是把大量Microsoft Word用户吸引到自己的软件上。没有必须遵循的规范要求（只有控制公共栈的要求）。投放市场的时间很重要——从现在起20个月，到时候无论好坏必须拿出产品。开发团队成员肯定没有很好的工程文化，如果读者试图以第一种文化的普通方式讨论问题，就会使他们认为那是“应该避免出现的损失。”

适合第一个项目团队的测试实践会在第二个项目团队中失败。适合第二个项目团队的实践对于第一个项目团队就是会受到起诉的失职。

语境驱动学派的成员

如果读者赞同这些原则，并想加入这个学派，可通过context@satisfice.com与我们联系。

以下一些专著的作者，都赞同这些原则：

Cem Kaner

James Bach

Bret Pettichord

Anna S. W. Allison

Ståle Amland

Bernie Berger

Jaya R. Carl

Ross Collard

Christopher Denardis

Marge Farrell

Erick Griffin

Sam Guckenheimer

Elisabeth Hendrickson

Kathy Iberle

Bob Johnson

Karen Johnson

Mark Johnson

Alan A. Jorgensen博士

Brian Marick

Patricia A. McQuaid博士

Alan Myrvold

Noel Nyman

Pat McGee

Johanna Rothman

Jane Stepak

Paul Szymkowiak

Andy Tinkham

Steve Tolman

参考文献

-A-

Agre, Phil. 2001. Networking on the Network. Available at dlis.gseis.ucla.edu/people/pagre/network.html.

Amland, Ståle. 1999. Risk Based Testing and Metrics. Available at www.amland.no/Word%20Documents/EuroSTAR%20'99%20Paper.doc.

Asböck, Stefan. 2000. *load testing for eConfidence*. Lexington, MA: Segue Software, Inc. Available via www.segue.com.

Association for Computing Machinery. 2000. A Summary of the ACM Position on Software Engineering as a Licensed Engineering Profession. Available at www.acm.org/serving/se_policy/selep_main.html.

Austin, Robert. 1996. *Measuring and Managing Performance in Organizations*. New York: Dorset House Publishing.

-B-

Bach, James, 1999a. Reframing Requirements Analysis. *IEEE Computer* 32:6. 113–114.

Bach, James, 1999b. A Low Tech Testing Dashboard. Available at /www.satisfice.com/presentations/dashboard.pdf.

Bach, James, 1999c. James Bach on Risk-Based Testing. STQE Magazine. vol. 1 #6.

Baron, Jonathan. 1994. *Thinking and Deciding*. Cambridge: Cambridge University Press.

Beck, Kent, 1999. *Extreme Programming Explained*. Reading, Massachusetts: Addison-Wesley.

Beck, Kent, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick. R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas. 2001. *Manifesto for Agile Software Development* [online]. Available at www.agilealliance.org/.

Beizer, Boris. 1990. *Software Testing Techniques, 2nd edition*. Boston: International Thompson Computer Press.

Bender, Richard A. 1991. Requirements-based Testing, *Quality Assurance Institute Journal*, 27–32.

Black, Rex. 1999. *Managing the Testing Process*. Redmond, Washington: Microsoft Press.

Booth, Wayne C., Gregory G. Colomb, and Joseph M. Williams. 1995. *The Craft of Research*. Chicago: University of Chicago Press.

Brooks, Frederick P. 1995. *The Mythical Man-Month: Anniversary Edition with Four New Chapters*. Reading Massachusetts: Addison-Wesley.

Brown, John Seely and Paul Duguid. 2000. *The Social Life of Information*. Boston: Harvard Business School Press.

Buwalda, Hans and Maartje Kasdorp. 1999. Getting Automated Testing Under Control, *Software Testing & Quality Engineering*, November 1999.

-C-

Chapman, Jack. 1996. *Negotiating Your Salary: How to Make $1000 a Minute, 3rd edition*. Berkeley: Ten Speed Press.

Cohen, Daniel M., Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. 1996. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, Volume 13#5, September. Available at www.argreenhouse.com/papers/gcp/AETGissre96.shtml.

Cohen, D. M., S. R. Dalal, M. L. Fredman, and G. C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Transactions on Software Engineering. Vol 23#7, July. Available at www.argreenhouse.com/papers/gcp/AETGieee97.shtml.

Cohen, Noam. 2000. Building a Testpoint Framework. *Dr. Dobbs Journal*. March 2000.

Collard, Ross. 1999. Deriving Test Cases from Use Cases. *Software Testing & Quality Engineering*, July–August.

Collard, Ross. Forthcoming. *Software Testing & QA Techniques* (a multivolume series). (Some of this material is available in the course notes he hands out to his students.)

Constantine, Larry L. 1995. *Constantine on Peopleware*. Yourdon Press.

Construx 2001. Software Engineering Professionalism Web site. Available at www.construx.com/profession/home.htm.

-D-

Daconta, Michael C., Eric Monk, J. Paul Keller, and Keith Bohnenberger. 2000. *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs*. New York: John Wiley & Sons.

de Bono, Edward. 1970. *Lateral Thinking: Creativity Step by Step*. New York: Harper and Row.

DeMarco, Tom. 1997. *The Deadline*. New York: Dorset House Publishing.

DeMarco, Tom and Timothy Lister. 1999. *Peopleware: Productive Projects and*

Teams, 2nd *edition*. New York: Dorset House Publishing.
Deming, W. Edwards. 1986. *Out of the Crisis*. Cambridge, Massachusetts: MIT Press.
DeNardis, Chris, 2000. Perspectives of a Test Manager. *STQE Magazine* 2: 5. Available at www.stickyminds.com/sitewide.asp?sid=194646&sqry=%2AJ%28MIXED%29%2AR%28relevance%29%2AK%28simplesite%29%2AF%28Perspectives+from+a+Test+Manager%29%2A&sidx=0&sopp=10&ObjectId=1976&Function=DETAILBROWSE&ObjectType=ART.
DiMaggio, Len. 2000. Looking Under The Hood. *STQE Magazine*. January.
Dörner, Dietrich. 1996. *The Logic of Failure*. Trans. Rita Kimber and Robert Kimber. New York: Metropolitan Books. Originally published in 1989.
Drucker, Peter. 1985. *The Effective Executive*. Harper Colophon.
Dustin, Elfriede, Jeff Rashka, and John Paul. 1999. *Automated Software Testing*. Reading, Massachusetts: Addison-Wesley.
Dwyer, Graham and Graham Freeburn. Business Object Scenarios: a fifth-generation approach to automated testing. In Fewster and Graham (1999).

-E-

Elmendorf, William R., 1973. *Cause-Effect Graphs in Functional Testing* Technical Report TR-00.2487, IBM Systems Development Division, Poughkeepsie, N.Y.

-F-

Fewster, Mark and Dorothy Graham. 1999. *Software Test Automation: Effective Use of Text Execution Tools*. Reading, Massachusetts: Addison-Wesley.
Feynman, Richard. 1989. *What Do You Care What Other People Think: Further Adventures of a Curious Character*. New York: Bantam Books.
Fisher, R. and D. Ertel. 1995. *Getting Ready to Negotiate: The Getting to Yes Workbook*.
Fisher, Roger, William Ury and Bruce Patton. 1991. *Getting to Yes*. Boston: Houghton Mifflin Co.
Freund, James C. 1992. *Smart Negotiating: How to Make Good Deals in the Real World*. Simon and Schuster.

-G-

Gause, Donald C. and Gerald M. Weinberg. 1989. *Exploring Requirements: Quality Before Design*. New York: Dorset House Publishing.
Gilb, Tom. 1997. *Evo: The Evolutionary Project Managers Handbook*. Available at /www.result-planning.com/, click on Download Center. If this document is not available, check for related material at www.stsc.hill.af.mil/

swtesting/gilb.asp.
Glaser, Barney G. and Anselm L. Strauss. 1999. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* New York: Aldine de Gruyter.

-H-

Heidtman Steel Products, Inc. v. Compuware Corp. 1999 U.S. Dist. Lexis 21700, U.S. District Court, N.D. Ohio.
Hendrickson, Elisabeth. 1999. Making the Right Choice: The Features You Need in a GUI Test Automation Tool. *STQE Magazine*, May. Available at www.qualitytree.com/feature/mtrc.pdf.
Hendrickson, Elisabeth. 2001a. Better Testing, Worse Quality? Available at www.qualitytree.com/feature/btwq.pdf.
Hendrickson, Elisabeth. 2001b. Bug Hunting: Going on a Software Safari. *Proceedings of the Software Testing Analysis & Review Conference* (STAR East). Orlando, Florida: May.
Hendrickson, Elisabeth. Undated. Architecture Reverse Engineering. Available at www.testing.com/test-patterns/patterns/Architecture-Reverse-Engineering.pdf.
Hendrickson, Elisabeth. Forthcoming. *Bug Hunting.* New York: Dorset House Publishing. (Some of this material is available in the course notes she hands to students in her course on *Bug Hunting.)*
Hoffman, Douglas. 1999a. Cost Benefits Analysis of Test Automation, *Proceedings of the Software Testing Analysis & Review Conference* (STAR East). Orlando, Florida: May.
Hoffman, Douglas. 1999b. Test Automation Architectures: Planning for Test Automation. *Proceedings of the International Software Quality Week.* San Francisco, May.
Hoffman, Douglas. 2000. The Darker Side of Metrics. Proceedings of the *Pacific Northwest Software Quality Conference,* October 17–18. Portland, Oregon.
Houlihan, Paul. 2001. Targeted Software Fault Insertion, *Proceedings of the Software Testing Analysis & Review Conference* (STAR East) Orlando, Florida: May. Available at www.mango.com/technology.
Humphrey, Watts S. 1990. *Managing the Software Process.* Reading, Massachusetts: Addison-Wesley.
Humphrey, Watts S. 1997. *Managing Technical People.* Reading, Massachusetts: Addison-Wesley.
Hutchins, Edwin. 1995. *Cognition in the Wild.* Cambridge, Massachusetts: MIT Press.

-I-

IEEE Computer Society. 2001. *Software Engineering Body of Knowledge, trial*

version 0.95. Available at www.swebok.org/documents/stoneman095/Trial_Version_0_95.pdf.

InstallShield Corporation. 1999. *Creating a Project with the NetInstall Spy*. Available at http://support.installshield.com/reference/netinstall/UG/ugchapter3.pdf.

-J-

Jacobson, Ivar. 1992. *Object-Oriented Software Engineering*. Wokingham: Addison-Wesley.

Jeffries, Ron, Ann Anderson, and Chet Hendrickson. 2000. *Extreme Programming Installed*. Reading, Massachusetts: Addison-Wesley.

Jensen, Arthur Robert. 1980. *Bias in Mental Testing*. Free Press.

Johnson, Karen. 2001. Mining Gold from Server Logs. *STQE Magazine*. January.

Jorgensen, Paul C. 1995. Software Testing: A Craftsman's Approach. Boca Raton, Florida: CRC Press.

-K-

Kaner, Cem. 1995a. Software Negligence and Testing Coverage. *Software QA Quarterly* volume 2, number 2: 18. Available at http://kaner.com/coverage.htm.

Kaner, Cem. 1995b. Liability for Defective Documentation. *Software QA Quarterly*, volume 2, number 3: 8. Available at www.kaner.com/baddocs.htm.

Kaner, Cem. 1996a. Computer Malpractice, *Software QA*, volume 3, number 4: 23. Available at www.badsoftware.com/malprac.htm.

Kaner, Cem. 1996b. *Negotiating Testing Resources*. Available at www.kaner.com/negotiate.htm.

Kaner, Cem. 1998a. *Avoiding Shelfware: A Manager's View of Automated GUI Testing*. Available at www.kaner.com/pdfs/shelfwar.pdf.

Kaner, Cem. 1998b. Liability for Product Incompatibility. *Software QA Magazine*, September.

Kaner, Cem. 2000a. Measurement of the Extent of Testing. Available at www.kaner.com/pnsqc.html.

Kaner, Cem. 2000b. Architectures of Test Automation. Available at http://kaner.com/testarch.html.

Kaner, Cem, James Bach, Hung Quoc. Nguyen, Jack Falk, and Bob Johnson. 2002. *Testing Computer Software, 3rd edition, Volume 1*. Forthcoming.

Kaner, Cem, Jack Falk, and Hong Nguyen. 1993. *Testing Computer Software, 2nd edition, 1999 reprint*. New York: John Wiley & Sons.

Kaner, Cem, Elisabeth Hendrickson and Jennifer Smith-Brock. 2000.

Managing the Proportion of Testers to (Other) Developers. *Proceedings of the International Software Quality Week.* San Francisco.

Kaner, Cem and David Pels. 1997. *Article 2B and Software Customer Dissatisfaction.* Available at www.badsoftware.com/stats.htm.

Kaner, Cem and David Pels. 1998. *Bad Software.* New York: John Wiley & Sons.

Kaner, Cem and John R. Vokey. 1984. A Better Random Number Generator for Apple's Floating Point BASIC. *Micro,* June, 26–35. Available at www.kaner.com/random.html.

Kaplan, Robert S. and David P. Norton. 1996. *The Balanced Scorecard: Translating Strategy into Action.* Cambridge, Massachusetts: Harvard Business School Press.

Koslowski, Barbara. 1996. *Theory and Evidence: The Development of Scientific Reasoning.* Cambridge, Massachusetts: MIT Press.

Kruchten, Philippe. 2000. *The Rational Unified Process, an Introduction, 2nd edition.* Reading, Massachusetts: Addison-Wesley.

-L-

Lakatos, Imre. 1976. *Proofs and Refutations: The Logic of Mathematical Discovery.* Cambridge, Massachusetts: Cambridge University Press.

Lawrence, Brian and Bob Johnson. 1998. *A Product Life Cycle (PLC) Model.* Available at www.coyotevalley.com/plc/builder.htm.

Lebow, Rob. 1990. *A Journey Into the Heroic Environment.* Rocklin, California: Prima Publishing.

Levy, David A. 1997. *Tools of Critical Thinking: Metathoughts for Psychology.* Boston: Allyn and Bacon.

Linz, Tilo and Matthias Daigl. 1998a. GUI Testing Made Painless: Implementation and Results of the ESSI PIE 24306. Available at www.imbus.de/forschung/pie24306/gui_test_made_painless.html.

Linz, Tilo and Matthias Daigl. 1998b. How to Automate Testing of Graphical User Interfaces. Available at www.imbus.de/forschung/pie24306/gui/aquis-full_paper-1.3.html.

-M-

Marick, Brian. 1995. *The Craft of Software Testing.* Upper Saddle River, New Jersey: Prentice Hall.

Marick, Brian. 1998. When Should a Test be Automated? Available at www.testing.com/writings/automate.pdf.

Marick, Brian. 1999. How to Misuse Code Coverage. Available at www.testing.com/writings/coverage.pdf.

Marick, Brian. 2000. Using Ring Buffer Logging to Help Find Bugs. Available at http://visibleworkings.com/trace/Documentation/ring-buffer.pdf.

Marick, Brian. Undated. How Many Bugs Do Regression Tests Find? Available at www.testingcraft.com/regression-test-bugs.html.
Mead, Nancy. 2001. Issues in Licensing and Certification of Software Engineers. July 23. Available at www.sei.cmu.edu/staff/nrm/license.html.
Michalko, Michael. 1991. *Thinkertoys: A Handbook of Business Creativity.* Berkeley, California: Ten Speed Press.
Miller, L. J. 1998. Get More Money on Your Next Job: 25 Proven Strategies for Getting More Money, Better Benefits, & Greater Job Security. McGraw-Hill Professional Publishing.

-N-

Nguyen, Hung Quoc. 2000. *Testing Applications on the Web.* New York: John Wiley & Sons.
Norman, Donald A. 1993. *Things that make us smart: Defending human attributes in the age of the machine.* Reading, Massachusetts: Addison-Wesley.
Notkin, David A., Michael Gorlick, and Mary Shaw. 2000. An Assessment of Software Engineering Body of Knowledge Efforts: A Report to the ACM Council. Available at www.acm.org/serving/se_policy/bok_assessment.pdf.
Nyman, Noel. 2000. Using Monkey Test Tools, *Software Testing & Quality Engineering*, January.

-O-

Olve, Nils-Goran, Jan Roy, and Magnus Wetter. 1999. *Performance Drivers: A Practical Guide to Using the Balanced Scorecard.* New York: John Wiley & Sons.
O'Malley, Michael. 1998. *Are You Paid What You're Worth?* Broadway Books.
Ostrand, Thomas J. and Marc J. Balcer. 1988. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM.* Volume 31 #6: 676–686. June.

-P-

Park, S. K. and K. W. Miller. 1988. Random Number Generators: Good ones are hard to find. *Communications of the ACM*, October, volume 31, issue 10, 1192–1201.
Pettichord, Bret. 1996. Success with Test Automation. *Proceedings of the International Software Quality Week.* San Francisco, California: May 1996. Available at www.io.com/~wazmo/succpap.htm.
Pettichord, Bret. 1999. Seven Steps to Test Automation Success. *Proceedings of the Software Testing Analysis & Review Conference* (STAR West). San Jose, California. November. Available at www.io.com/~wazmo/papers/

seven_steps.html.
Pettichord, Bret. 2000a. Beyond the Bug Battle, *Proceedings of the Software Testing Analysis & Review Conference* (STAR East). Orlando, Florida: May.
Pettichord, Bret. 2000b. Testers and Developers Think Differently. *Software Testing & Quality Engineering*. January. Available at www.io.com/~wazmo/papers/testers_and_developers.pdf.
Pettichord, Bret. 2001a. Hey Vendors, Give Us Real Scripting Languages. *Stickyminds.com*. Available at http://stickyminds.com/sitewide.asp?sid=409206&sqry=%2AJ%28MIXED%29%2AR%28createdate%29%2AK%28simplesite%29%2AF%28scripting+languages%29%2A&sidx=0&sopp=10&ObjectId=2326&Function=DETAILBROWSE&ObjectType=COL.
Pettichord, Bret. 2001b. What you don't know may help you. *Stickyminds.com*. July. Available at www.stickyminds.com/sitewide.asp?ObjectId=2629&ObjectType=COL&Function=edetail.
Pettichord, Bret. 2001c. Let observation be your crystal ball. *Stickyminds.com*, May. Available at www.stickyminds.com/sitewide.asp?ObjectId=2498&ObjectType=COL&Function=edetail.
Polya, George. 1957. *How to Solve It*. Princeton: Princeton University Press.
Popper, Karl. 1989. *Conjectures and Refutations: The Growth of Scientific Knowledge*. London: Routledge.
PowerQuest Corporation. 2001. *Drive Image*. Available at www.powerquest.com/driveimage/.

-R-

Rational Software Corporation. 2001. *Rational Purify for Unix*. Available at www.rational.com/products/purify_unix/index.jsp.
Rational Software Corporation. 2001. *Rational Test Foundation for Windows 2000*. Available at www.rational.com/products/testfoundation/w2k_ds.jsp.
Robinson, Harry. 1999. Finite State Model-Based Testing on a Shoestring. Star West 1999. Available at www.geocities.com/model_based_testing/shoestring.htm.

-S-

Schneier, Bruce. 2000a. Computer Security: Will We Ever Learn? *Crypto-Gram*, May 15. Available at www.counterpane.com/crypto-gram-0005.html.
Schneier, Bruce. 2000b. *Secrets & Lies*. New York: John Wiley & Sons.
Simmonds, Erik. 2000. When Will We Be Done Testing? Software Defect Arrival. Proceedings of the Pacific Northwest Software Quality Conference, October 17–18. Portland, Oregon.
Sims, Henry P. Jr. and Charles C. Manz. 1996. *Company of Heroes*. New York:

John Wiley & Sons.
Solow, Daniel. 1990. *How to Read and Do Proofs*. New York: John Wiley & Sons.
Strauss, Anselm and Juliet Corbin, eds. 1997. *Grounded Theory in Practice*. Thousand Oaks: SAGE Publications.
Strauss, Anselm and Juliet Corbin. 1998. *Basics of Qualitative Research, 2nd edition*. Thousand Oaks: SAGE Publications.
Sweeney, Mary R. 2001. *Automation Testing Using Visual Basic*. Berkeley, CA: Apress.

-T-

Tarrant, John J. 1997. *Perks & Parachutes: Negotiating Your Best Possible Employment Deal, from Salary and Bonus to Benefits and Protection*. Times Books.
Telles, Matt and Yuan Hsieh. 2001. *The Science of Debugging*. Scottsdale, Arizona: Coriolis.
Tukey, John W. 1977. *Exploratory Data Analysis*. Reading, Massachusetts: Addison-Wesley.

-W-

Webster, Bruce. 1995. *Pitfalls of Object-Oriented Development*. M&T Books.
Weick, Karl E. 1995. *Sensemaking in Organizations*. Thousand Oaks: SAGE Publications.
Weinberg, Gerald M. 1992. *Quality Software Management, Volume 1: Systems Thinking*. New York: Dorset House Publishing.
Weinberg, Gerald M. 1997a. *Quality Software Management, Volume 2: First-Order Measurement*. New York: Dorset House Publishing.
Weinberg, Gerald M. 1997b. *Quality Software Management, Volume 3: Congruent Action*. New York: Dorset House Publishing.
Weinberg, Gerald M. 1997c. *Quality Software Management, Volume 4: Anticipating Change*. New York: Dorset House Publishing.
Weinberg, Gerald M. 1998. *The Psychology of Computer Programming, Silver Anniversary Edition*. New York: Dorset House Publishing.
Weinberg, Gerald M. 2001. *An Introduction to General Systems Thinking: Silver Anniversary Edition*. New York: Dorset House Publishing.
Whittaker, James and Alan Jorgensen. 1999. Why Software Fails. ACM Software Engineering Notes, July. Available at http://se.fit.edu/papers/SwFails.pdf.
Whittaker, James and Alan Jorgensen. 2000. How to Break Software. Proceedings of the Software Testing Analysis & Review Conference, May. Orlando, Florida.
Whittaker, James. 2002. *How to Break Software*. Reading, Massachusetts: Addison-Wesley.

Wiegers, Karl E. 1996. *Creating a Software Engineering Culture*. New York: Dorset House Publishing.

Worrall, John and Gregory Currie, eds. 1978. *The methodology of scientific research programmes*. Cambridge, Massachusetts: Cambridge University Press.

Wurman, Richard Saul. 1991. *Follow the Yellow Brick Road: Learning to Give, Take, and Use Instructions*. New York: Bantam Books.

-Y-

Yourdon, Edward. 1997. *Death March: The Complete Software Developer's Guide to Surviving 'Mission Impossible' Projects*. Indianapolis, Indiana: Prentice Hall Computer Books.

-Z-

Zuse, Horst. 1997. *A Framework of Software Measurement*. Walter de Gruyter, Inc.

[General Information]
书名=软件测试经验与教训
作者=
页数=234
SS号=11133456
出版日期=