ISTQB®软件测试人员认证

高级大纲
技术测试分析师

2012 版
（中文版 20131001）

国际软件测试认证委员会

中文版的翻译编辑和出版统一由 ISTQB®授权的 CSTQB®负责

英文版权声明

如果此文档的来源是公认的，则可以拷贝此完整的文档或部分。

中文版权声明

版本历史

Version	Date	Remarks
ISEB v1.1	2001 年 9 月 4 日	ISEB 从业人员大纲
ISTQB® 1.2E	2003 年 9 月	ISTQB® 高级大纲（EOQ-SG）
V2007	2007 年 10 月 12 日	认证高级测试工程师大纲 2007 版
D100626	2010 年 6 月 26 日	纳入 2009 年同意的修改，把每章的内容分至每个单独的模块。
D101227	2010 年 12 月 27 日	接受不影响语句含义的格式变化和修正。
Draft V1	2011 年 9 月 17 日	基于协定的范围文档而新拆分的 TTA 大纲第一版。ALWG 评审
Draft V2	2011 年 11 月 20 日	所有成员委员会评审后版本
Alpha 2012	2012 年 43 月 9 日	纳入了所有成员委员会对 10 月发布版本的意见。
Beta 2012	2012 年 4 月 7 日	Beta 版提交 GA
Beta 2012	2012 年 6 月 8 日	正文编辑版发布给各成员委员会。
Beta 2012	2012 年 6 月 27 日	加入 EWG 和术语意见。
RC 2012	2012 年 8 月 15 日	候选发布版，最终包括 NB 编辑
RC 2012	2012 年 9 月 2 日	纳入 BNLTB 和 Stuart Reid 的意见，Paul Jorgensen 交叉核对
GA 2012	2012 年 10 月 19 日	最终编辑和整理版提交成员大会（GA）
中文发布版	2013 年 10 月 1 日	最终编辑和整理提交 CSTQB®

目录

致谢

本文件由国际软件测试认证委员会负责技术测试分析师模块的高级子工作组的核心团队于 2010 至 2012 年间编制，他们包括： Graham Bath（主席）、Paul Jorgensen、Jamie Mitchell。

核心团队向评审团队和所有测试委员会成员提出的建议和帮助表示感谢。

直至高级课程大纲截稿时，其工作组成员如下（按字母排序）：

Graham Bath、Rex Black、Maria Clara Choucair、Debra Friedenberg、Bernard Homès（副主席）、Paul Jorgensen、Judy McKay、Jamie Mitchell、Thomas Mueller、Klaus Olsen、Kenji Onishi、Meile Posthuma、Eric Riou du Cosquer、Jan Sabak、Hans Schaefer、Mike Smith （主席）、Geoff Thompson、Erik van Veenendaal、Tsuyoshi Yumoto。

下列成员参与了评审、评论和大纲表决工作:

Dani Almog、Graham Bath、Franz Dijkman、Erwin Engelsma、Mats Grindal、Dr. Suhaimi Lbrahim、Skule Johansen、Paul Jorgensen、Kari Kakkonen、Eli Margolin、Rik Marselis、Judy McKay、Jamie Mitchell、Reto Mueller、Thomas Müller、Ingvar Nordstrom、Raluca Popescu、Meile Posthuma、Michael Stahl、Chris van Bael、Erik van Veenendaal、Rahul Verma、Paul Weymouth、Hans Weiberg、Wenqiang Zheng、Shaomin Zhu。

本文由 ISTQB® 大会于 2012 年 10 月 19 日正式发布。

参加本大纲翻译的 CSTQB® 专家有（按姓氏拼音排序）：陈耿、吴芳、徐文叶、赵钦佩（组长）。

负责本大纲评审的 CSTQB® 专家有（按姓氏拼音排序）：柴阿峰、崔启亮、周震漪

0. 课程大纲引言

0.1 本文档的编写目的

本大纲根据国际软件测试认证委员会对高级大纲（技术测试分析师）的要求进行编写，ISTQB®提供此大纲的主要目的：

1. 国家认证委员会需将大纲翻译成当地语言并分发给培训机构，并可根据特定语言对大纲进行适度润色、修改，以保证语句通顺可读。
2. 考试委员会可根据特定语言，按照高级大纲（技术测试分析师）的学习目标设计考题。
3. 培训机构需根据高级大纲（技术测试分析师）准备课程并选择最适宜的教学方法。
4. 需要认证的考生，根据高级大纲准备考试（作为培训课程的一部分或独立使用）。
5. 国际软件和系统工程领域，应以此大纲为基础，推进软件和系统测试蓬勃发展。并以此为基础著书和出文章。

国际软件测试认证委员会 ISTQB®允许其他组织、机构在获得书面授权后使用此大纲内容。

0.2 概述

高级大纲由以下 3 个单独的部分组成：

- 测试经理
- 测试分析师
- 技术测试分析师

高级大纲的概述文档 [ISTQB®_AL_OVIEW] 包括以下信息：

- 每个大纲的学习成果
- 每个大纲摘要
- 各大纲之间的关系
- 认知程度描述 （K 级别）
- 附录

0.3 预期学习目标

预期学习目标是学习成果的前提保证，同时也被作为高级技术测试分析师认证考试题目编写的基础。标记为 K1 级的知识点，需要考生牢记、回忆、识别和认知。K2，K3，K4 级的知识点会在之后相关章节出现。

0.4 期望

部分针对技术测试分析师的学习目标需要以下领域的基本经验：

- 综合的编程概念
- 综合的系统架构概念

1. 技术测试分析师在基于风险的测试中的任务 - 30 分钟.

关键词

产品风险（product risk）、风险分析（risk analysis）、风险评估（risk assessment）、风险识别（risk identification）、风险级别（risk level）、风险缓解（risk mitigation）、基于风险的测试（risk-based testing）

技术测试分析师在基于风险的测试中的任务相关的学习目标

1.3 风险评估

TTA-1.3.1 (K2) 总结技术测试分析师需要考虑的、典型的风险因素

通用的学习目标

以下学习目标涉及到本章的多个小节。

TTA-1.x.1 (K2) 总结在基于风险的测试方法中，技术测试分析师在测试计划和测试执行过程中的相关活动。

1.1 简介

测试经理具有建立和管理基于风险的测试策略的全面责任。测试经理往往要求技术测试分析师的介入以确保正确执行基于风险的方法。

由于其独有的技术特长，技术测试分析师与以下基于风险的测试任务密切相关：

- 风险识别
- 风险评估
- 风险缓解

为了处理出现的产品风险和变更优先级，以及定期地评估和沟通风险状态，以上任务会迭代地贯穿在整个项目中。

技术测试分析师在由测试经理为项目制定的、基于风险的测试框架中工作。他们贡献出与项目内在密切相关的技术风险知识，比如安全相关的风险、系统可靠性和性能相关风险。

1.2 风险识别

在风险识别过程中越广泛地接触各类项目干系人，能识别出最大数量的重大风险的可能性也就越大。由于技术测试分析师掌握独特的专门技能，他们特别适合进行专家访谈，与同事头脑风暴以及分析当前和以前的工作经验来确定可能存在产品风险的区域。特别地，技术测试分析师与其他技术同行（例如，开发人员、架构师、运维工程师）工作密切，有利于确定存在技术风险的区域。

识别的风险样本可能包括：

- 性能风险（例如，在高负载条件下无法达到响应时间的要求）
- 安全风险（例如，在安全攻击下泄露敏感数据）
- 可靠性方面风险（例如，应用程序无法满足服务等级协议中指定的可用性）

与特定的软件质量特性相关的风险区域将在本大纲的相关章节中介绍。

1.3 风险评估

风险识别致力于鉴别尽可能多的相关风险，而风险评估旨在研究这些识别的风险以便于更好地进行风险分类并确认风险的可能性和影响程度。

确认风险级别通常涉及为每个风险项评估发生的可能性及发生后带来的影响程度。风险发生的可能性一般解释为当系统在测试条件下，一些潜在的问题真实存在的可能性。

技术测试分析师致力于发现和理解每个风险项潜在的技术风险，而测试分析师致力于理解当问题发生时带来的潜在商业影响。

通常需要考虑的因素包括：

- 技术复杂度
- 代码结构的复杂度
- 项目干系人之间关于技术需求的冲突

- 由开发组织的地理分布产生的沟通问题
- 工具和技术
- 时间、资源和管理压力
- 缺少项目前期的质量保障
- 技术需求的高变化率
- 大量与技术质量特性相关的缺陷
- 技术接口和集成相关问题

在风险信息已知的情况下，技术测试分析师根据由测试经理建立的指导方针来建立风险级别。举例来说，测试经理可能决定将风险级别划分为 1 到 10 之间的值，其中值 1 为最高风险。

1.4 风险缓解

项目期间，技术测试分析师影响着测试如何响应所识别的风险。这个影响主要包括以下几个方面：

- 通过执行最重要的测试，实施测试策略和测试计划中所述的适当的缓解和应急活动来减少风险
- 在项目开展过程中，收集额外的信息，并在此基础上评估风险；而且利用该信息来实施缓解行动，旨在减少先前识别和分析的风险的可能性和影响

2. 基于结构的测试 - 225 分钟.

关键词

原子条件（atomic condition）、条件测试（condition testing)、控制流测试（control flow testing）、判定条件测试（decision condition testing）、复合条件测试（multiple condition testing）、路径测试（path testing）、短路/缩短（short-circuiting）、语句测试（statement testing）、基于结构的技术（structure-based technique）

基于结构的测试学习目标

2.2 条件测试

TTA-2.2.1 (K2) 理解如何实现条件覆盖以及为何判定覆盖比条件覆盖更严谨

2.3 判定条件测试

TTA-2.3.1 (K3) 应用判定条件测试的测试设计技术设计测试用例以达到规定的覆盖率

2.4 改进的条件/判定覆盖（MC/DC）测试

TTA-2.4.1 (K3) 应用改进的条件/判定覆盖（MC/DC）测试的测试设计技术设计测试用例以达到规定的覆盖率

2.5 复合条件测试

TTA-2.5.1 (K3) 应用复合条件测试的测试设计技术设计测试用例以达到规定的覆盖率

2.6 路径测试

TTA-2.6.1 (K3) 应用路径测试的测试设计技术来设计测试用例

2.7 API 测试

TTA-2.7.1 (K2) 理解 API 测试的适用性以及它所能发现的各种缺陷

2.8 选择一种基于结构的测试

TTA-2.8.1 (K4) 基于特定的项目状况选择一种合适的基于结构的测试技术

2.1 简介

本章主要描述基于结构的测试设计技术，也被称作白盒测试或者基于代码的测试技术。这种技术以代码、数据和架构以及/或者系统流程图作为测试设计的基础。通过每一个具体的技术能系统化地导出测试用例，同时每个技术关注的是所考虑的结构的特定方面。该技术提供覆盖准则，这些准则必须经过测量，而且必须与每个项目或组织所定义的目标相关联。达到全覆盖也并不意味着整个测试集是完整的，而是正在使用的技术对在考虑中的结构不再能提出任何有用的测试。

除了条件覆盖，本大纲中所考虑的基于结构的测试设计技术比初级大纲（ISTQB®_FL_SYL）中涉及的语句和判定覆盖技术更为严谨。

以下是本大纲中涉及的技术：

- 条件测试
- 判定条件测试
- 改进的条件/判定覆盖（MC/DC）测试
- 复合条件测试
- 路径测试
- API 测试

以上列出的前四个技术都是基于判定语句（真/假），同时能大范围地找到相同类型的缺陷。无论判定语句有多复杂，最终都能判定为真或假，从而可通过代码跟踪某一条路径。当由于一个复杂的判定语句没有得到预期的结果，导致一条预定的路径没有被运行到，就能检测出缺陷。

总体来说，前四个技术逐次递进地体现全面性。它们需要定义更多的测试以便达到预期的覆盖范围，同时能发现更多这种类型缺陷的例子。

参考[Bath08]，[Beizer90]，[Beizer95]，[Copeland03] 和 [Koomen06]。

2.2 条件测试

在判定（分支）测试中将判定看作一个整体，测试用例分别评估真和假的结果，而条件测试考虑的是在判定中的单个简单的“原子”条件。每个判定语句是由一个或多个简单的“原子”条件组成，而每个“原子”条件能计算出一个布尔值（真/假），这些值的逻辑组合便得出判定的最终结果。测试用例必须评估每个原子条件的两个方向（真和假），以达到覆盖率的要求。

适用性

由于下述（局限性/难点）的难点，条件测试可能只限于在理论上的研究。然而，理解条件测试对于建立在它基础上，达到更高级别的覆盖非常有必要。

局限性/难点

当一个判定中存在两个或两个以上的原子条件，在测试设计过程中，如果没有很好地选择测试数据，会导致只达到条件覆盖而没有达到判定覆盖。举例来说，假定一个判定谓词，“A and B”。

	A	B	A and B
测试 1	假	真	假
测试 2	真	假	假

为了达到 100%的条件覆盖，运行以上表格中的两个测试。当这两个测试能达到 100%的条件覆盖的时候，它们并不能达到判定覆盖，因为在两种情况下，语句都是假。

当一个判定只有单个的原子条件，条件测试等同于判定测试。

2.3 判定条件测试

判定条件测试明确要求测试必须达到条件覆盖（见上），同时，也要求满足判定覆盖（见初级大纲[ISTQB®_FL_SYL]）。在不需要增加额外的测试用例以达到条件覆盖的情况下，对原子条件的测试数据值做出的慎重的选择可以达到这一覆盖要求。

下面的例子测试了与前面相同的判定语句，“A and B”。通过选择不同的测试值，运用相同数目的测试可以获得判定条件覆盖。

	A	B	A and B
测试 1	真	真	真
测试 2	假	假	假

因此，这种技术可能会有效率方面的优势。

适用性
这种覆盖级别应该适用于测试比较重要的，但还不是至关重要/关键的代码。

局限性/难点
此技术由于测试过程中需要比判定覆盖更多的测试用例，当时间很紧迫时，就可能存在困难。

2.4 改进的条件/判定覆盖 （MC/DC）测试

这种技术提供了一个更强的控制流覆盖级别。假设在 N 个单独的原子条件下，MC/DC 通常可以得到 N+1 个单独的测试用例。MC/DC 实现了判定条件覆盖，但是需要满足以下条件：

1. 至少在一个测试中，判定结果会随着原子条件 X 是否为真而改变
2. 至少在一个测试中，判定结果会随着原子条件 X 是否为假而改变
3. 每个不同的原子条件有满足条件 1 和 2 的测试

	A	B	C	(A or B) and C
测试 1	真	假	真	真
测试 2	假	真	真	真
测试 3	假	假	真	假
测试 4	真	假	假	假

上述例子中不仅达到了判定覆盖（判定语句的结果不仅有为真，也有为假），同时也达到了条件覆盖（A，B 和 C 可以计算得到真和假）。

在测试 1 中，A 是真，输出是真。如果将 A 改成假（如测试 3 中所示，保持其它值不变），结果会变成假。
在测试 2 中，B 是真，输出是真。如果将 B 改成假（如测试 3 中所示，保持其它值不变），结果会变成假。

在测试 1 中，C 是真，输出是真。如果将 C 改成假（如测试 4 中所示，保持其它值不变），结果会变成假。

适用性
这种技术广泛应用于航天和航空软件产业和其它安全关键系统中。它通常用于测试安全关键的软件，因为在这类软件中，任何一个失效会带来巨大灾难。

局限性/难点
当一个表达式中某个特定的项多次出现时，要达到 MC/DC 覆盖变得较为复杂，而在这种情况中一个在表达式中多次出现的项称为“耦合”项。根据代码中的判定语句，可能无法仅通过改变耦合项的值去改变判定的输出结果。解决此问题的方法之一是只针对非耦合的原子条件进行 MC/DC 级别测试，另外一种方法是基于对每个含有耦合项的判定根据实际情况进行分析。

某些编程语言和/或解释器被设计成当它们在评估代码中一个复杂的判定语句时，它们会采用缩短的决策行为。也就是说，如果最终的评估结果可以由部分表达式的评估结果来决定，则执行代码无需评估整个表达式。例如，如果要计算判定结果“A and B”，假如已知 A 是假，那么就没有必要去计算 B。不管 B 是什么值都不能改变最终值，所以代码可以通过不计算 B 来减少执行时间。这种缩短的决策行为可能会影响 MC/DC 覆盖的能力，因为一些为了满足覆盖要求的测试可能无法执行。

2.5 复合条件测试

在极少数情况下，可能需要测试所有可能的判定内所包含的真 / 假数值组合。这种穷尽级别的测试被称作复合条件覆盖。所需的测试数目依赖于判定语句中的原子条件个数，同时该测试数目可以由计算 2 的 n 次方来确定，n 是未耦合的原子条件个数。运用之前提及的相同例子，需要以下测试来实现复合条件覆盖：

	A	B	C	(A or B) and C
测试 1	真	真	真	真
测试 2	真	真	假	假
测试 3	真	假	真	真
测试 4	真	假	假	假
测试 5	假	真	真	真
测试 6	假	真	假	假
测试 7	假	假	真	假
测试 8	假	假	假	假

如果语言中使用了缩短的决策方法，实际的测试用例数目经常因作用于原子条件上的逻辑运算符的顺序和分组而减少。

适用性
这种测试技术历来被用于测试嵌入式软件，因为这种软件需要确保其运行在很长一段时间内的稳定可靠和不崩溃（例如：电话交换机期望能持续 30 年）。在大部分关键应用程序中，这种类型的测试很可能会被 MC/DC 测试所取代。

局限性/难点
因为测试用例的数目可以直接通过包含了所有原子条件的真值表计算得出，所以能容易地确定这个级别的覆盖。然而，这种方法所需的测试用例的绝对数量非常庞大，使得 MC/DC 覆盖更适用于大多数情况。

2.6 路径测试

路径测试包括识别贯穿于代码中的路径，然后创造相关测试来覆盖这些路径。原则上，测试每一条贯穿于系统的独特路径都是非常有用的。然而，在任何重要系统中，由于代码中存在循环而使得测试用例的数目会变得相当庞大。

尽管如此，如果将无限循环的问题放置一边，实行路径测试还是现实的。为了应用这种技术，[Beizer90]建议沿着软件模块的入口到出口的尽可能多的路径去创建测试用例。为了简化这个可能复杂的任务，他建议可以通过使用以下流程来系统化地实现路径测试：

1. 先挑选从入口到出口最简单的、有意义的功能的路径。
2. 挑选每条额外的路径作为之前路径的微小变异。对于每个后续的测试，每个路径中尽可能只改变一个分支。尽可能优先选择短的路径而不是长的路径。尽可能优先选择那些更有意义的功能路径，而不是无意义的功能路径。
3. 仅仅当要求覆盖率的时候，挑选那些无意义的功能路径。Beizer 在这条规则中加注，这样的路径可能是不相关的且应该受到质疑的。
4. 运用直觉来选择路径（也就是说，哪条路径是最可能被执行到的）。

注意，使用这种策略，可能会重复执行某些路径段。这个策略的关键点是代码中每个可能的分支至少测试过一次，也可能多次。

适用性
局部路径测试，如上述定义，往往在安全关键软件中实行。这对本章中介绍的其它方法是一个很好的补充，因为它着眼于贯穿软件的路径而不仅仅是单个的判定。

局限性/难点
虽然有可能使用控制流程图来确定路径，在现实中还是需要利用工具对复杂模块的路径进行计算。

覆盖率
创建足够的测试来覆盖所有路径（不考虑循环），这也已经保证达到了语句和分支覆盖。与分支测试相比较，路径测试仅增加相对少的测试数目，却提供了更彻底的测试。[NIST96]

2.7 API 测试

应用程序接口（API）是使不同进程、程序和/或系统之间的通信成为可能的代码。API 经常应用于客户/服务器端中，在这种关系中，一个进程给其它进程提供某种功能。

从某些方面来看，API 测试与测试图形用户界面（GUI）相当类似。重点是评估输入值和返回数据。

逆向测试在与 API 打交道时往往至关重要。程序员在使用 API 访问外部服务时，可能会使用不同于设计者预期的方式来调用 API 接口。这也意味着，强大的错误处理能力在避免不正确操作方面是必不可少的。因为 API 经常和其它 API 结合使用，同时也因为单个接口可能包含多个参数，而这些参数值可能以不同方式组合，所以多种不同接口的组合测试可能是必须的。

由于 API 之间经常松散耦合，会导致实际的事务丢失或时序问题，这使恢复和重试机制的全面测试成为必要。提供 API 接口的组织必须确保所有的服务具有非常高的可用性。这往往需要由 API 发布者和基础构架支持者提供严格的可靠性测试。

适用性
因为越来越多的系统变成分布式的或使用远程处理来分摊一部分工作给其它处理器，API 测试也变得越来越重要。例子包括操作系统调用、面向服务架构（SOA-Service-Oriented Architecture）、远程过程调用（RPC-Remote Procedure Calls）、Web 服务以及几乎所有其它的分布式应用程序。API 测试尤其适用于综合系统（system of systems）的测试。

局限性/难点
直接测试 API 往往要求技术测试分析师使用专门的工具。通常情况下 API 不提供直接的图形界面，可能需要工具来设置初始环境、数据编组处理、执行 API 的调用和确定结果。

覆盖率
API 测试是一种测试类型的描述。它不代表任何特定的覆盖级别。在 API 测试中不仅至少应该执行所有 API 调用，而且还应该使用所有的有效值和所有合理的无效值。

缺陷类型
通过测试 API 往往能发现各种不同类型的缺陷，经常涉及到的有接口问题、数据处理问题或时序问题以及事务丢失或交易重复。

2.8 基于结构技术的选择

被测试系统的实际情况决定应达到的基于结构测试的覆盖级别。越重要的系统需要越高级别的覆盖。一般情况下，所需的覆盖级别越高，也需要更多的时间和资源来达到该覆盖级别。

有时候，所需的覆盖级别可能是从应用于软件系统的适用标准衍生而来。例如，假如一款软件是应用于航空电子设备中，它可能被要求符合标准 DO-178B（或欧洲标准 ED-12B）。此标准包含了以下五种失效情况：

A. 灾难的：失效可能会导致飞机安全飞行或降落所需的关键功能的缺失
B. 危险的：失效可能会对安全或性能有巨大的负面影响
C. 严重的：失效（导致的影响）是严重的，但是严重程度低于 A 或 B
D. 轻微的：失效（导致的影响）是可察觉的，但是比 C 影响小
E. 无影响：失效不影响安全

如果软件系统属于 A 级，它必须达到 MC/DC 覆盖。如果是 B 级，必须达到判定覆盖级别，但 MC/DC 是可选的。对于 C 级要求至少达到语句覆盖。

同样，IEC-61508 是针对可编程的、电子的和安全相关的系统的功能安全而制定的国际标准。这个标准已经适用于许多不同的领域，包括汽车、铁路、制造工艺、核电站和机械制造。危险程度的定义使用一种分级的安全完整性等级（SIL-Safety Integrity Level）连续值（1 是最不关键，4 是最关键）。推荐的覆盖如下：

1. 建议语句和分支覆盖
2. 强烈建议语句覆盖，建议分支覆盖
3. 强烈建议语句和分支覆盖
4. 强烈建议 MC/DC

在现代的系统中，极少出现所有处理都由一个单一的系统完成。只要当一部分的任务是在另一个系统上远程处理时，就应该进行 API 测试。系统的关键性决定了需要投入到 API 测试中的开销。

通常，技术测试分析师应该根据被测软件系统的实际情况来选择基于结构的测试方法。

3. 分析技术 - 255 分钟.

关键词

控制流分析（control flow analysis）、圈复杂度（cyclomatic complexity）、数据流分析（data flow analysis）、定义-使用对（definition-use pairs）、动态分析（dynamic analysis）、内存泄漏（memory leak）、成对集成测试（pairwise integration testing）、邻域集成测试（neighborhood integration testing）、静态分析（static analysis）、野指针（wild pointer）

分析技术学习目标

3.2 静态分析

TTA-3.2.1 (K3) 运用控制流分析来检测代码是否存在控制流异常
TTA-3.2.2 (K3) 运用数据流分析来检测代码是否存在数据流异常
TTA-3.2.3 (K3) 提出运用静态分析的方法来提高代码的维护性
TTA-3.2.4 (K2) 解释调用图在建立集成测试策略中的作用

3.3 动态分析

TTA-3.3.1 (K3) 列举运用动态分析所能达到的目标

3.1 简介

有两种分析手段：静态分析和动态分析。

静态分析（3.2 节）包括可以无需执行软件而进行的分析测试。由于没有执行软件，只能通过工具或者人工来检查软件在执行时是否能正确工作。软件的静态检查无需为场景的执行创建数据和前置条件就能进行详细的分析。

需要注意的是，第五章包含与技术测试分析师相关的不同形式的评审。

动态分析（3.3 节）需要实际地执行代码。动态分析经常用来寻找在代码被执行的时候更容易检测到的代码缺陷（例如，内存泄漏）。与静态分析一样，动态分析也可以依靠工具或人工来监视系统执行，观察各项指标，例如内存的快速增长。

3.2 静态分析

静态分析的目标是检测代码和系统架构中实际的或潜在的缺陷及提高其维护性。静态分析通常有工具支持。

3.2.1 控制流分析

控制流分析是一种静态分析技术，借助于控制流图或使用工具来分析程序的控制流。运用这种技术可以发现许多系统中的异常，包括设计不当的循环（例如，有多个入口点）、在某些语言（比如，Scheme）中模棱两可的函数调用的目标、不正确的操作序列等。

控制流分析最常见的用途之一是确定圈复杂度。圈复杂度值是一个正整数，这个正整数代表在强连通图中的独立路径数。在连通图中，循环和迭代在被遍历一次后就被忽略。每条独立的从入口到出口的路径代表了一条贯穿模块的独特路径。每条独特的路径都应该进行测试。

圈复杂度通常是用来理解一个模块整体的复杂程度。根据 Thomas McCabe 的理论 [McCabe 76]：系统越复杂，就越难维护，且会包含更多的缺陷。多年来许多研究指出复杂度和包含缺陷数目的相关性。NIST（美国国家标准和技术研究所）建议以 10 作为最大复杂度值。任何以更高的复杂度来度量的模块可能需要被划分成多个模块。

3.2.2 数据流分析

数据流分析包含了多种用以收集关于系统中变量使用信息相关的技术。这里，对变量的生命周期（即，何处变量被声明了、被定义了、被读取了、被评估和被撤销了）进行详细的研究，因为异常可能发生于变量生命周期中的任何操作。

一种常见的技术叫做定义-使用符号，在这种技术中，每个变量的生命周期被分成三种不同的原子操作：

- d（defined）：当变量被声明、定义或初始化
- u（used）　：当变量在计算或判定语句中被使用或读取
- k（killed）　：当变量被终止、撤销或者超出范围

这三个原子操作组合成对（定义-使用对）来说明数据流。例如，一条“du-路径”代表数据变量被定义随后使用的代码段。

可能的数据异常包括在错误的时间对变量进行正确的操作，或对变量中的数据进行不正确的操作。这些异常包括：

- 给一个变量赋一个无效值
- 在使用一个变量之前没有对其进行赋值
- 由于控制语句中不正确的值而采用了不正确的路径
- 尝试使用已经撤销的变量
- 引用不在作用域内的变量
- 说明和撤销一个没有使用过的变量
- 重复定义一个已经在使用的变量
- 没有终止一个动态分配的变量（可能导致内存泄漏）
- 由于改变一个变量导致不可预期的副作用（如在改变一个全局变量的时候没有考虑该变量的所有用途而引起的连锁反应）

正在使用的开发语言可以指导在数据流分析中所使用的规则。编程语言可能允许程序员执行某些非法的变量操作，但这可能在某些情况下会引起系统的表现与程序员的预期有所不同。例如，一个变量可能定义了两次，然而当跟随某条路径时该变量却没有实际使用。数据流分析往往会标明这些使用“可疑”。虽然这可能是变量赋值能力的合法使用，但它会导致代码在将来的维护性问题。

数据流测试“使用控制流图去探索那些可能发生在数据上的不合理的事情”[Beizer90]，因此数据流测试能发现与控制流测试不同的缺陷。技术测试分析师应该在计划测试时涵盖这种技术，因为大部分这些缺陷会引起动态测试时很难发现的间歇性缺陷。

然而数据流分析是一种静态技术。它可能错过在系统运行时发生在数据上的一些问题。例如，静态数据变量可能包含一个指向动态生成的数组的指针，该指针可能在运行之前都不存在。多处理器的使用和抢占式的多任务处理可能会产生在数据流或控制流分析中无法找到的竞争状态条件（实时情况）。

3.2.3 运用静态分析来改善维护性

静态分析可以应用在很多方面以提高代码、架构和网站的维护性。

低质量、未加注释及没有结构的代码往往很难维护。开发人员需要更多的精力来定位和分析代码中的缺陷。而且，修改代码以更正缺陷或添加新功能可能进一步引入缺陷。

静态分析在支持工具的帮助下，可以通过验证代码是否遵守代码标准和指导方针来提高代码的维护性。这些标准和指导方针描述了所需的编码实践，比如，命名规范、注释、缩进和代码模块化。注意，静态分析工具通常会标记出警告而不是错误，即使代码从语法上可能是正确的。

模块化设计通常会使代码更可维护。静态分析工具通过以下几种方式来支持模块化代码的开发：

- 它们寻找重复的代码。这部分代码可能是重构到模块的候选（虽然利用模块调用所带来的运行时开销可能会成为实时系统的一个问题）。
- 它们生成能测量代码模块化的价值指数的标准。这些包括了耦合和内聚的测量标准。一个有着良好维护性的系统更可能有低耦合值（在执行过程中，模块之间的依赖程度）及高内聚值（一个模块独立和关注单一任务的程度）。
- 它们会指出，在面向对象的代码中，哪些派生对象可能有太多或太少的父类可视性。
- 它们将代码或构架中具有高度结构复杂度的区域高亮显示，高结构复杂度经常被看作是维护性差和高潜在失效的标志。圈复杂度（参见 3.2.1 节）的可接受级别可以在指导方针中加以具体说

明，从而确保在注意维护性和缺陷预防前提下，用模块化的方式进行代码开发。高圈复杂度的代码可能是模块化的候选。

网站的维护也可以得到静态分析工具的支持。这里的目标是检查网站的类树形（导航）结构是否很好地平衡，或存在不平衡会导致以下问题：

- 更难的测试任务
- 增加的维护工作
- 用户浏览困难

3.2.4 调用图

调用图是通信复杂度的静态表述。它们都是有向图，其节点代表了程序的单元，而边代表了单元之间的通讯。

调用图可能运用在组件测试中，当不同的函数或方法互相调用时；或运用在集成和系统测试中，当独立的模块互相调用时；或在系统集成测试中，当独立的系统互相调用时。

调用图可以作为以下用途：

- 设计调用特定模块或系统的测试
- 在软件中确定调用模块和系统位置的数量
- 评估代码和系统架构的结构
- 对集成的顺序提供建议（成对集成和邻域集成，在下面的章节中将介绍更多这些方面内容）

在初级大纲 [ISTQB®_FL_SYL] 中，讨论了两种类型的集成测试：增量的（从上到下，至底而上等）和非递增的（大爆炸）。递增的方法更受偏爱因为它们使代码递增地介入，由于所涉及的代码量有限，可以更容易地隔离缺陷。

在高级大纲中，介绍了三种非增量的、利用调用图的方法。它们可能比增量的方法更好，因为增量的方法可能需要额外的构建来完成测试，以及写入不可交付的代码来支持测试。这三种方法是：

- 成对集成测试（不要与黑盒测试技术的成对测试混淆），以成对的组件作为目标进行集成，这些协同工作的组件如调用图中所示。虽然这种方法只是少量地减少了构建的数目，但是它减少了测试用具所需的代码量。
- 邻域集成测试所有的与给定节点相连接的节点，以此作为集成测试的基础。在调用图中，一个特定节点的所有前置和后置节点是测试的基础。
- McCabe 的设计方法是使用针对模块调用图的圈复杂度理论。这种方法需要创建调用图，在图内包含了模块间各种相互调用的可能性。这些可能的调用包括：
  - 无条件调用：一个模块总是调用另外一个模块
  - 有条件调用：一个模块有时调用另外一个模块
  - 互斥的条件调用：一个模块仅调用许多不同模块中的一个
  - 迭代调用：一个模块调用另外一个模块至少一次，但也可能多次
  - 迭代的条件调用：一个模块可以调用另外一个模块零到多次

  在创建了调用图后，可以计算集成复杂度，创建测试来覆盖调用图。

更多关于调用图和成对集成测试，请参考 [Jorgensen07]。

3.3 动态分析

3.3.1 综述

有些失效症状可能不是立即可见的，动态分析可用来检测这类失效。例如，内存泄漏的可能性可以通过静态分析来检测（发现代码中分配了内存，但却从未释放内存），但是内存泄漏在动态分析中是一目了然的。

不可立即重现的失效对测试的工作量以及发布的能力或高效使用软件的能力可能有重大的影响。这样的失效可能是由内存泄漏、不正确使用指针和其它损坏（例如，系统栈）导致的 [Kaner02]。由于这些失效的性质，包括系统性能的逐渐恶化，或者甚至是系统崩溃，测试策略必须考虑与这种失效相关联的风险，同时在适当的情况下，执行动态分析来减少失效（通常是通过使用工具）。由于这些失效的发现和纠正通常都是最昂贵的，建议在项目早期进行动态分析。

动态分析可以用来完成以下工作：

- 通过检测野指针和系统内存的泄漏来防止失效的发生
- 分析不能轻易再现的系统失效
- 评估网络行为
- 通过提供运行时系统行为的信息来提高系统性能

动态分析可以在任何测试级别进行，而且需要技术和系统技能来做到下面内容：

- 指定动态分析的测试目标
- 确定恰当的时间来启动和停止分析
- 分析测试结果

在系统测试过程中，即使技术测试分析师只有基本的技术技能，依然可以使用动态分析工具。所利用的工具通常创建全面的日志信息，这些日志能被那些有相应技术能力的人做进一步分析。

3.3.2 检测内存泄漏

内存泄漏发生在当程序可用的内存区域（RAM）被该程序所分配，但是不再使用后没有得到释放。这块内存区域被分配了但是不再能重用而遗留下来。当这种情况发生频繁或发生在低内存情况下，程序可能会耗尽可用的内存。从历史上看，内存的操作是程序员的责任。任何动态分配的内存区域必须由负责分配的程序在正确的范围内进行释放，以避免内存泄漏。许多现代的编程环境包含自动或半自动的“垃圾回收”机制，在这种机制中，所分配的内存可以在不需要程序员直接干涉的情况下得到释放。当现有分配的内存由自动垃圾回收进行释放时，隔离内存泄漏可能变得非常困难。

内存泄漏造成的问题可能是逐渐生成的而不总是立即显现的。例如，如果软件是最近安装的或者系统被重启，那么内存泄漏的情况可能不能立即显现的，而这种情况（指使用新安装的软件或在重新启动后的测试）在测试中经常发生。由于这些原因，往往当程序到了线上的时候，内存泄漏的负面影响才可能被注意到。

内存泄漏的症状是系统响应时间的不断恶化，最终可能导致系统失效。虽然这种失效可以通过重启系统来解决，但是重启并不总是有用的，甚至有时是不可能的。

许多动态分析工具找出会发生内存泄漏的代码，使它们能得到纠正。简单的内存监视器也可以对可用内存是否随时间推移而下降获得一个总体印象，虽然仍需一个后续分析来确定下降的确切原因。

其它来源的泄漏也应该考虑，例子包括文件句柄、访问许可证（信号）和资源的连接池。

3.3.3 检测野指针

程序中的“野”指针是那些不能使用的指针。例如，野指针可能已经“丢失”了它所指向的对象或函数，或它没有指向想要指向的内存区域（例如，它指向超出了某个数组分配边界的区域）。当程序使用了野指针，可能发生如下后果：

- 在某些情况下程序可能会达到预期效果。可能的情况是，野指针访问的内存是目前未被程序所使用的，名义上是“自由”的，和/或包含了合理的值。
- 程序可能会崩溃。在这种情况下，野指针可能造成部分内存被错误地使用，而这些内存在程序运行时是非常关键的（例如，操作系统）。
- 程序运行不正确，因为程序所需的对象不能访问。在这种情况下，该程序可能继续运行，虽然可能会给出一个出错信息。
- 数据在内存中的位置可能会被指针和随后使用的不正确的值破坏。

注意：对程序的内存使用情况所做的任何更改（例如，一个软件变更后的新构建）可能会诱发上面列出的四种后果。最初程序按预期执行，尽管程序使用了野指针，然后随着软件变更之后，程序意外崩溃（甚至可能发生在产品阶段），这种情况显得尤为危急。值得注意的是，这种失效往往是存在一个潜在的缺陷（即，野指针）的征兆（参见 [Kaner02]，第 74 课）。当程序使用了工具后，工具可以帮助识别野指针，不论它们对程序执行的影响。有些操作系统已经内置函数来检查在运行时内存访问异常。例如，当一个应用程序试图访问该应用程序所允许的内存区域之外的内存位置时，操作系统可能会抛出一个异常。

3.3.4 性能分析

动态分析不仅仅对检测失效有用。在对程序性能进行动态分析后，工具可以帮助确定性能瓶颈，且产生适用范围广的性能指标；开发人员可以利用此性能指标来优化系统性能。例如，提供关于模块在执行过程中被调用的次数的信息。那些被频繁调用的模块将可能是提升性能的候选模块。

结合软件的动态行为信息和静态分析中获得的调用图信息（参见 3.2.4 节），测试人员也可以识别那些适合做详细和广泛的测试的候选模块（例如，频繁被调用和有诸多接口的模块）。

程序性能的动态分析通常在系统测试阶段才进行，但也可以在早期测试阶段（系统测试之前）进行，例如，当借助于测试框架（测试用具）对一个子系统进行测试时。

4. 技术测试的质量特性 - 405 分钟.

关键词

适应性（adaptability）、易分析性（analyzability）、易改变性（changeability）、共存性（co-existence）、效率（efficiency）、易安装性（installability）、维护性测试（maintainability testing）、成熟性（maturity）、运行验收测试（operational acceptance test）、运行配置（operational profile）、性能测试（performance testing）、可移植性测试（portability testing）、易恢复性测试（recoverability testing）、可靠性增长模型（reliability growth model）、可靠性测试（reliability testing）、易替换性（replaceability）、资源利用性测试（resource utilization testing）、健壮性（robustness）、安全性测试（security testing）、稳定性（stability）、易测试性（testability）

技术测试的质量特性的学习目标

4.2 总体规划问题

TTA-4.2.1 (K4) 针对特定的项目和被测系统，对非功能需求进行分析，并编写测试计划的相关部分

4.3 安全性测试

TTA-4.3.1 (K3) 为安全性测试定义方法和设计概要测试用例

4.4 可靠性测试

TTA-4.4.1 (K3) 为可靠性质量特性及其相应的 ISO 9126 子特性定义方法和设计概要测试用例

4.5 性能测试

TTA-4.5.1 (K3) 为性能测试定义方法和设计概要测试用例

通用的学习目标

以下的学习目标与本章中多个部分所涵盖的内容相关。

TTA-4.x.1 (K2) 理解和解释在测试策略和/或测试方法中包括维护性测试、可移植性测试和资源利用性测试的原因
TTA-4.x.2 (K3) 给定一个特定的产品风险，定义最合适的非功能性的测试类型
TTA-4.x.3 (K2) 理解和解释在应用程序的生命周期中，需要实施非功能测试的各个阶段
TTA-4.x.4 (K3) 对于一个给定的场景，定义通过运用非功能测试类型能找到的缺陷类型

4.1 简介

通常情况下，技术测试分析师侧重在测试产品工作得“如何”，而不是它做了“什么”的功能方面。这些测试可以在任何测试级别进行。例如，实时性系统和嵌入式系统的组件测试过程中，进行性能基准测试和资源使用的测试都很重要。在系统测试和运行验收测试（OAT-Operational Acceptance Test）过程中适合进行对可靠性方面的测试，例如，易恢复性。因为在这个级别的测试都建立在一个由软件和硬件组合的特定系统，该被测的特定系统可以包括各种服务器、客户端、数据库、网络和其它资源。无论哪个测试级别，应该根据已确定的风险优先级和可利用的资源进行测试。

ISO9126 中提供的产品质量特性的描述是用来描述这些特性的指南之一。也可以使用其它标准，诸如 ISO25000 系列（已经取代了 ISO9126）。ISO9126 质量特性被划分成包含子特性的特性。这些都列举在下面的表格中，同时描述了哪些特性/子特性是包含在测试分析师和技术测试分析师大纲中的。

特性	子特性	测试分析师	技术测试分析师
功能性	准确性，适合性，互操作性，依从性	X	
	安全性		X
可靠性	成熟性（健壮性），容错性，易恢复性，依从性		X
易用性	易理解性，易学性，易操作性，吸引性，依从性	X	
效率	性能（时间上的行为），资源利用性，依从性		X
维护性	易分析性，易改变性，稳定性，易测试性，依从性		X
可移植性	适应性，易安装性，共存性，易替换性，依从性		X

虽然这种分配在不同的组织的工作中可能会不同，这里所提及的是在 ISTQB® 大纲中所遵循的分配。

依从性的子特性在每个质量特性中都有提到，至于在某些安全关键或受控环境下，每个质量特性可能需要遵守特定的标准和法规。由于标准在不同行业是有很大的变化，所以在此不做深入讨论。如果技术测试分析师在一个受依从性需求影响的环境中工作时，理解这些需求以及确保测试和测试文档满足依从性需求是很重要的。

对所有在本节中所讨论的质量特性和子特性，必须识别典型风险，以便形成合适的测试策略并将之文档化。质量特性测试需要特别关注生命周期时间点、所需工具、软件和文档的可用性和技术专长。如果没有对每个特性和其独特的测试需求策略进行规划，那么测试人员在制定时间计划表时可能无法正确地规划出测试计划、测试准备和测试执行的时间 [Bath08]。某些测试，例如性能测试，需要大规模的计划、需要专用的设备、特定的工具、专业的测试技能以及（通常情况下还需要）大量的时间。质量特性和子特性的测试必须通过为相应的工作量分配足够的资源来集成到整体测试时间表中。这些领域中的每项都有特定的需求，面向特定的问题，而且它们可能出现在软件生命周期的不同时段，如下节所讨论。

当测试经理关注在编制和报告汇总有关质量特性和子特性的度量信息时，测试分析师或技术测试分析师则（根据上面的表格）负责收集每个度量的信息。

技术测试分析师在软件进入生产阶段前的测试中收集的质量特性的度量构成软件系统的供应商和项目干系人（例如，客户，运营商）之间的服务等级协议（SLA-Service Level Agreement）的基础。在某些

情况下，测试可能在软件进入生产阶段后继续执行，通常是由一个独立的或在相同领域的团队或组织负责。这通常会在效率和可靠性测试中见到，在这些测试中，在实际的生产环境中与在测试环境中的测试结果往往会不同。

4.2 总体规划问题

如果在计划中忽略了非功能测试，要确保应用程序的成功就会非常困难。测试经理可能要求技术测试分析师识别相关质量特性的主要风险（参见 4.1 节中的表格），并解决任何与所提议的测试相关的规划问题。这些结果可能应用于创建主测试计划。在执行这些任务时，会考虑以下这些综合因素：

- 项目干系人的需求
- 所需工具的获得和人员培训
- 测试环境的要求
- 组织方面的考虑
- 数据安全方面的考虑

4.2.1 项目干系人的需求

非功能性需求定义通常都很粗略，有时甚至根本不存在。在计划阶段，技术测试分析师必须能够从受影响的项目干系人那里获得与技术质量特性对应的期望级别，且评估这些级别所代表的风险。

常见的方法是假设客户对当前的系统版本满意，只要系统版本能够保持所达到的质量级别，那么他们对新版本也会满意。这使得系统的当前版本（的质量特性指标）可以当做基准。当对一些非功能质量特性（比如性能），项目干系人很难详细说明他们的需求的时候，这也是特别有用的方法。

当捕获非功能需求的时候，获得多方面的观点是非常明智的。这些观点必须从诸如客户、用户、操作人员和维护人员等项目干系人那里获得，否则一些需求很有可能会被忽略。

4.2.2 所需工具的获得和人员培训

商业工具或模拟器特别适用于性能测试和特定的安全测试。技术测试分析师应该评估涉及到采购、学习和实施工具所需的成本和时间。在使用专门的工具时，在规划时还应该考虑到使用新工具的学习曲线和/或聘请外部工具专家的成本。

一个复杂的模拟器的开发可能本身就代表了一个开发项目，也应如此规划。尤其，在时间计划和资源计划中必须考虑所开发工具的测试和文档。如果当被模拟的产品发生变化时，模拟器必须更新和重新测试，这一部分的时间和预算也必须充分体现在计划中。当模拟器用于安全关键应用时，在计划中还必须考虑相应的验收测试以及通过一个独立机构对模拟器进行认证。

4.2.3 测试环境的要求

许多技术测试（例如，安全性测试，性能测试）要求有一个与生产环境类似的测试环境，以便进行实际的测量。被测系统大小和复杂度的不同可能对测试的计划和资金预算产生很大影响。由于构建此类环境的成本较高，可以考虑以下选择：

- 使用实际的生产环境
- 使用缩减版的系统。必须注意的是，要确保所得到的测试结果足以代表整个生产系统。

必须仔细规划这类测试的执行时间。这类测试极有可能只有在特定的时间段（例如，在较少使用系统时间内）执行。

4.2.4 组织方面的考虑

技术测试可能包括测量完整系统中的几个组件的行为（例如，服务器、数据库、网络）。若这些组件分布在多个不同的场所和组织，则可能要花很大的精力进行测试的计划和协调。例如，某些软件组件可能只在每天或每年的特定时间段才能用于系统测试，或组织只能提供有限的天数用于支持测试。如果其它组织的系统组件或人员（例如，“外借”的专家）不能确定是否能参与测试，将可能严重影响预定的测试。

4.2.5 数据安全方面的考虑

在测试计划阶段就应考虑到用于确保系统安全的手段，以确保所有的测试活动可行。例如，使用数据加密技术可能会增加创建测试数据和验证结果的难度。

数据保护政策和法令可能会禁止在实际生产数据的基础上生成任何所需的测试数据。数据匿名化是一件重要的任务，必须将其作为测试实现的一部分来计划。

4.3 安全性测试

4.3.1 简介

安全性测试在两个重要方面不同于其它形式的功能性测试：

1. 选择测试输入数据的标准技术可能会遗漏重要的安全问题
2. 安全性缺陷与其它功能测试所发现的缺陷的症状有较大不同

安全性测试以破坏系统的安全策略为目标进行攻击，从而检查系统的漏洞或薄弱环节。以下列出了需要在安全性测试中寻找的潜在威胁：

- 未经授权的应用或数据拷贝
- 未经授权的访问控制（例如，用户能够执行其没有权限执行的任务）。用户权限、访问权限和特权是这个测试的重点。系统的规格说明应该提供此信息。
- 当执行其预期功能的时候，软件显示出意想不到的副作用。例如，一个媒体播放器虽然能正确地播放音频，但是它是通过将文件写入到未加密的临时存储空间中的方法实现的，软件盗版者就能利用这些副作用。
- 插入网页的代码，可能被后续用户（跨站点脚本 Cross-Site-Scripting 或 XSS）所使用。这种代码可能是恶意的。
- 缓冲区溢出（缓冲区超出限度），这可能由在用户界面的输入区域输入了能比正确处理的长度更长的字符串所导致的。一个缓冲区溢出漏洞就给了运行恶意代码指令一个机会。
- 服务拒绝，阻止用户与应用程序的交互。（例如，通过发送“骚扰”请求使网络服务器超载）
- 第三方的秘密窃听、复制和/或改变然后转发通信（例如，信用卡交易），而用户根本没有意识到第三方的存在（“Man in the Middle 中间人”攻击方式）。
- 破解用来保护敏感数据的加密代码。
- 逻辑炸弹（有时称为“复活节彩蛋”）,可能被蓄意的埋进代码中，在特定条件被激活（例如，在某个特定日期）。当逻辑炸弹激活时，它们可能执行恶意行为，如文件删除或格式化磁盘。

4.3.2 安全性测试计划

一般而言，以下几个方面在规划安全性测试时非常重要：

- 因为在系统架构、设计和实施过程中会引入安全问题，安全性测试可能安排在单元、集成和系统测试级别。由于安全威胁不断变化的性质，安全性测试也可能定期地安排在系统投入生产后。
- 技术测试分析师所提出的测试策略应该包括代码评审和利用安全性工具的静态分析。这些在发现架构、设计文档和代码的安全性问题时是有效的，这些安全性问题在动态测试的时候往往容易忽略。
- 可能要求技术测试分析师来设计和执行某些安全性“攻击”（见下文），这些“攻击”需要认真地规划、与项目干系人协调。其它安全性测试可能在与开发人员或测试分析师（例如，测试用户权限、访问控制和特权）的合作下执行。对此，安全性测试计划内必须包括这些组织方面问题的考虑。
- 安全性测试计划的一个重要方面是获得批准。对技术测试分析师而言，这意味着从测试经理处获得明确的许可来执行计划好的安全性测试。执行任何额外的，没有计划的测试显得与实际的攻击一样，而进行那些测试的人可能会处在法律诉讼的风险中。在没有书面显示意图和授权情况下，“我们在执行安全性测试”这种借口很难是令人信服的解释。
- 应当指出，为系统安全所做的改进可能会影响其性能。在改进安全性方面后，考虑是否有必要进行性能测试是明智的（见下文第 4.5 节）。

4.3.3 安全性测试规格说明

特定的安全性测试根据安全风险的来源可归纳为 [Whittaker04]：

- 用户界面相关的--未经授权的访问和恶意输入
- 文件系统相关的--访问文件或资料库中存储的敏感数据
- 操作系统相关的--敏感信息的存储，如密码，在内存中以非加密的形式；当系统由恶意输入导致崩溃时，这些信息可能会被暴露。
- 外部软件相关--系统利用的外部组件之间发生的相互作用。这可能是在网络层面（例如，不正确的数据包或消息传递）或软件组件层面（例如，该软件所依赖的软件组件的失效）

以下方法 [Whittaker04] 可以用来开发安全性测试：

- 收集在指定的测试中可能有用的信息，如员工姓名、物理地址、内部网络相关细节、IP 地址、所用软件或硬件的标识和操作系统版本。
- 用各种有用的工具进行漏洞扫描。这些工具不是直接用来危害系统，而是识别那些可能是或导致违背安全政策的漏洞。也可以使用检查表来识别特定的安全漏洞。（美国）国家标准和技术研究院（NIST- National Institute of Standards and Technology）提供 [Web-2] 类似的检查表。
- 通过使用获得的信息，研制“攻击方案”（也就是企图破坏某一系统安全性策略的测试行为的方案）。需要在攻击方案中制定针对各种接口（用户界面，文件系统）的输入，以检测出最严重的安全性缺陷。在 [Whittaker04] 中描述的各种“攻击”，是一种有价值的技术来源，尤其是为安全性测试开发的技术。

安全性问题也能通过评审（见第 5 章）和/或使用静态分析工具（见第 3.2 节）发现。静态分析工具包含了一套大量的规则。这些规则针对安全威胁，依靠这些规则可以对代码进行检查。例如，缓冲区溢出问题，是由数据分配前未能检查缓冲区大小而造成的，这些都可以通过工具来发现。

静态分析工具可用于检查网页代码，以检查其可能暴露的安全漏洞，如代码注入、cookie 安全性、跨站脚本攻击、资源篡改和 SQL 代码注入。

4.4 可靠性测试

产品质量特性的 ISO 9126 分类标准定义了以下可靠性的子特性：

- 成熟性
- 容错性
- 易恢复性

4.4.1 测量软件成熟性

可靠性测试的目标之一是在一段时间内，观察软件成熟性的统计测量数据，并将此与期望的可靠性目标做比较。该可靠性目标可能用服务等级协议（SLA）表达。这些测量数据可能采取平均故障间隔时间（MTBF- Mean Time Between Failures）、平均修复时间（MTTR- Mean Time To Repair ）或任何其它故障强度度量（例如，每周发生的、具有特定严重程度的故障数目）的形式。这些度量可能用来作为（例如，产品发布的）出口准则。

4.4.2 容错性测试

在功能测试中，通过输入意外输（例如，无效数据），对软件处理意外输入值（所谓逆向测试）的容限进行评估，除了功能测试以外，需要额外的测试来评估系统的容错性，这种故障往往在被测应用程序之外发生。这些故障通常由操作系统报告（例如，磁盘已满、进程或服务不可用、未找到文件、内存不可用）。系统级的容错性测试可由特定的工具支持。

注意：术语“健壮性”和“容错”也常常在讨论容错性的时候用到（详见 [ISTQB®_GLOSSARY]）。

4.4.3 易恢复性测试

可靠性测试的进一步表现是对软件系统按照规定的方式从软硬件故障中恢复至正常状态的能力进行评估。易恢复性测试分为失效切换（Failover）和备份（Backup）/恢复（Restore）测试。

某些软件失效可引发严重后果，因而会采取专门的硬件和/或软件措施以保障系统即使在失效事件中仍能运行。针对这种情况要进行失效切换测试。如，失效切换测试适用于金融损失风险极高或存在严重安全隐患的情况。这种对于灾难性事件引发失效的易恢复性测试也被称为“灾难恢复（disaster recovery）”测试。

典型的硬件方面的措施包括平衡各个处理器负载，集群化服务器、处理器或硬盘，以便在一个硬件失效的时候，另一个马上接管（例如，冗余系统）。典型的软件方面的措施则可在一个所谓的非相似冗余系统（Redundant dissimilar system）中实现多个独立软件系统（例如，航空飞行控制系统）。冗余系统一般结合了上述的软件和硬件措施，根据独立实例的个数（2、3 或 4 个）可以分别称之为双重、三重或四重系统。令两个（或多个）互不相关的开发团队按相同的软件需求，开发不同的软件来提供相同的服务，从而达到实现软件的非相似性的目的。因而使非相似冗余系统在相似的缺陷输入情况下不至于导致相同的后果。这些增强系统的可恢复性所采取的措施会直接影响其可靠性，可在可靠性测试中予以考虑。

失效切换测试的目的是，通过模拟失效模式或在受控的环境中实际地制造失效来明确地测试系统。针对失效切换机制进行测试，可以保证数据不至丢失或缺损且系统保持原有服务级别（例如，功能可用性和响应时间）。关于失效切换测试的更多信息，请参阅 [Web-1]。

备份/恢复测试关注的是将故障影响最小化的过程性措施。这类测试对进行不同形式的备份并将其在数据丢失或缺损时进行恢复的（通常写入手册的）过程进行评估。测试用例的设计要保证能够覆盖该过程的关键路经。可通过技术评审来预演这些场景并针对实际的安装过程验证各类手册。在运行验收测试（OAT）时，这些场景可以在实际的产品环境或类似的环境中实施，以确认它们的实际效用。

备份/恢复测试的指标可能包括以下几部分：

- 完成各类备份（例如全备份或增量备份）所需时间
- 恢复数据所需时间
- 确保数据备份的级别（例如，恢复不超过 24 小时内的所有数据、恢复不超过 1 小时内的具体交易数据）

4.4.4 可靠性测试的计划

总体上来说，以下方面在规划可靠性测试时非常重要：

- 在软件进入生产阶段后，可以继续监视其可靠性。当以测试规划为目的，收集可靠性需求时，必须咨询负责软件操作的机构及其人员。
- 技术测试分析师可以选择一个可靠性增长模型（reliability growth model），该模型显示了在一段时间内的可靠性预期级别。可以通过可靠性增长模型对预期和实际达到的可靠性级别之间进行比较，给测试经理提供有用的信息。
- 可靠性测试应在类似生产环境中进行。所使用的环境应保持尽可能的平稳，使得能在一个时间段内对可靠性的趋势进行监测。
- 由于可靠性测试往往需要使用整个系统，所以可靠性测试最常见的是作为系统测试的一部分。然而，可靠性测试可以在单独的组件上，也可以在集成成套的组件上。详细的架构、设计和代码评审也可以用于去除一部分发生在实施系统中的可靠性问题的风险。
- 为了产生统计学上有效的测试结果，可靠性测试通常需要很长的执行时间。这可能使其很难安排在与其它测试一起执行。

4.4.5 可靠性测试的规格说明

可靠性测试可能采取一种创建重复的、预定的测试用例集合并执行的形式。这些可能是在一个测试用例池中随机选择的测试，或者运用统计模型并借助于随机或伪随机方法生成测试用例（测试数据）。可靠性测试也可以基于被称作为“运行概况（Operational Profiles）”的使用模式（参见第 4.5.4 节）。

某些可靠性测试可能指定重复地执行内存密集型的行动以便检测出可能存在的内存泄漏问题。

4.5 性能测试

4.5.1 简介

产品质量特性的 ISO 9126 分类标准将性能（时间行为）计入效率的子特性。性能测试侧重在组件或系统在特定的时间和特殊的条件下，响应用户或系统输入的能力。

性能的测量随着测试目标的变化而变化。针对单独的软件组件，性能可以根据 CPU 周期来测量；而针对基于客户的系统，性能可以根据其响应特定用户的请求所需的时间来测量。对那些架构是由多个组件构成的系统（例如，客户端、服务器、数据库），性能的测量是针对单独的组件之间的事务，从而可以识别出性能的“瓶颈”。

4.5.2 性能测试类型

4.5.2.1 负载测试

负载测试的关注点在于系统在处理预期实际负载级别增长方面的能力，实际负载级别增长是由大量用户或者进程并发使用所产生的事务请求导致的。可以测量和分析各种典型的使用情况（运行概况）下系统的平均响应时间。请参见 [Splaine01]。

4.5.2.2 压力测试

压力测试关注的是系统或组件在达到或超过其预期或指定的工作负载的界限，或在可用的计算容量和可用带宽等资源可用性减少的情况下处理负荷的能力。在压力级别逐渐增加时，系统性能应该按照预期缓慢下降，而不致失效。特别应该在峰值负载下对系统的完整功能进行测试，以发现在功能处理时的故障或确定数据的不一致。

压力测试的另一目标是确定系统崩溃的临界点，从而发现系统中的最薄弱环节。在进行压力测试时，允许向系统中逐渐地添加额外的容量（如内存，CPU 处理能力，数据库存储量）。

4.5.2.3 可扩展性测试

可扩展性测试，关注于系统满足未来效率要求的能力（这可能超过目前要求的）。测试的目的是确定在不超出目前规定的性能要求或不失效的情况下系统扩展的能力（例如，容纳更多的用户，存储更多的数据）。了解这些限度后，可以设定一些阈值以便在运行过程中对出现的问题进行监控和预警。此外，可以用适量的硬件调整生产环境。

4.5.3 性能测试的计划

除了 4.2 节中描述的总体计划外，以下因素可以影响性能测试的计划：

- 根据所使用的测试环境和所测试的软件，（参见 4.2.3 节）性能测试可能要求在做有效的测试之前，已经实现了整个系统。因此，通常是在系统测试中安排性能测试。其它在组件测试级别所进行的有效的性能测试，同理也应该在组件测试级别中安排。
- 一般来说，即使还不存在类似实际生产的环境，最初的性能测试也是越早进行越好。这些早期测试可能会发现性能问题（如瓶颈），且通过避免在此后的软件开发阶段或投入生产后再去从事耗时的错误修复来减少项目风险。
- 代码评审，尤其是专注于数据库的交互、组件交互和错误处理方面的评审，能识别性能问题（特别是关于“等待并重试（wait and retry）”的逻辑和低效的查询），这类代码评审应安排在软件生命周期的早期进行。
- 运行性能测试所需的硬件、软件和网络带宽应在计划内得到保障并编入预算。需求主要取决于要生成的负载，该负载是基于模拟的虚拟用户的数量以及它们可能产生的网络流量。如果在计划中没有考虑这点，可能导致采用不具代表性的性能测量。例如，验证大访问量的互联网网站的可扩展性需求，可能需要模拟数十万的虚拟用户。
- 生成性能测试所需的负载可能对硬件及工具采购费用具有显著影响。这必须在计划性能测试的时候考虑在内，来确保有足够的资金可用。
- 可以通过租用所需的测试基础设施，使性能测试生成负载的成本最小化。这可能涉及到，例如，租用顶级性能工具或使用一个第三方的服务提供商以满足硬件需求（例如，云服务）。如果采取此方法，用于性能测试的时间受到限制，因此必须精心规划。
- 在计划阶段应小心谨慎，以确保所使用的性能工具提供了与被测系统所使用的通信协议的兼容性。
- 性能相关的缺陷往往对被测系统有重大影响。当系统的性能需求是非常重要的时候，有用的做法是在关键组件（通过驱动和桩）上进行性能测试，而不是等待系统级别的测试。

4.5.4 性能测试的规格说明

不同性能测试（例如，负载和压力测试）的测试规格说明是基于所定义的运行概况，而运行概况包含了与应用程序交互的用户行为的不同形式。对于一个给定的应用程序可以有多个运行概况。

使用监测工具（如果实际或类似的应用系统已经可用）或通过预测可以确定每个运行概况的用户数目，这些预测值可以是基于算法或者由业务部门提供，这些对于确定可扩展性测试的运行概况非常重要。

在性能测试过程中，运行概况是所使用的测试用例的数目和类型的基础，通常使用测试工具为被测运行概况生成模拟用户数或“虚拟”用户数（参见 6.3.2）。

4.6 资源利用

产品质量特性的 ISO 9126 分类标准将资源利用性计入效率的子特性。与资源利用性有关的测试是根据定义的基准数据评估系统资源的使用情况（例如，内存的使用率、硬盘容量、网络带宽和连接）。这些系统资源的利用不仅在正常系统负载下，而且在压力条件下，如较高的交易率或大数据量进行比较，从而确定是否有不寻常的使用增长。

例如，对嵌入式实时系统来说，内存使用情况（有时也称为“内存占用”）在性能测试中扮演了非常最重要的角色。如果内存占用超过所允许的程度，系统可能因为没有足够内存而不能在特定时间内执行要求的任务。可能会拖慢系统，甚至导致系统崩溃。

此外，动态分析也可应用于检查资源利用性（参见 3.3.4 节）和识别性能瓶颈的任务中。

4.7 维护性测试

在软件生命周期中，软件的维护阶段占据了大部分时间，与此相比，软件开发阶段只占有较少的时间。维护测试是用来测试正在运行的系统的变化或环境的变化对正在运行系统的影响。为了确保对一个系统高效维护，在维护性测试中探索如何对程序代码更容易地进行分析、更改和测试。

受影响的项目干系人（例如，软件所有者或操作者）的典型的维护性目标包括：

- 拥有或运行软件的成本降至最低
- 软件维护所需的停机时间最小

以下一个或多个因素发生时，测试策略和/或测试方法中应包括维护性测试：

- 软件的变化可能发生在软件上线后（例如，纠正缺陷或引入计划中的更新）
- 受影响的项目干系人认为在软件生命周期中实现维护性目标（见上文）的收益远大于执行维护性测试的成本以及做出任何必要的变更所需的成本
- 软件维护性差的风险（例如，对用户和/或客户报告的缺陷的响应时间过长）证明进行维护性测试是正确的

针对维护性测试的有效方法，如3.2节和5.2节中所讨论的，包括静态分析和评审。维护性测试应该在设计文档就绪后就开始介入，并在整个实现过程中不断持续。由于维护性嵌入到每个组件的代码和相应的文档，因此，维护性可以在生命周期的早期进行评估，无需等待一个完整的已在运行的系统。

动态维护性测试的关注点在于文档化规程，开发这些规程是为了维护一个特定应用程序（如进行软件升级）。测试时可以选取维护场景作为测试用例，确保使用文档化的规程就既可达到所要求的服务级别。此测试方法尤其适用于下列情况：基础设施相对复杂，并需要多个部门协作支持。这些测试同时可以作为运行验收测试（OAT）的一个部分。[Web-1]

4.7.1 易分析性、易改变性、稳定性和易测试性

系统的维护性可以根据三个方面测量：诊断系统内已经识别的问题所需的开销（易分析性），实现代码的更改所需的开销（易改变性）和测试更改系统所需的开销（易测试性）。稳定性则涉及到系统对变化的响应。具有低稳定性的系统当发生变更的时候会展现大量下游问题（downstream）（也被称为“连锁反应”）[ISO9126][Web-1]。

执行维护任务所需的工作量由各种因素决定，如软件设计方法（例如，面向对象的）和使用的编码标准。

注意本文的，“稳定性”不应该与 4.4.2 节涉及的“健壮性”和“容错性”混淆。

4.8 可移植性测试

可移植性测试通常和软件移植到某个特定的运行环境中的难易程度相关，包括第一次的安装或从现有环境上移植。可移植性测试包括易安装性测试、共存性/兼容性测试、适应性测试和易替换性测试。可移植性测试可以从单独的组件开始（例如，一个特定的组件的易替换性，比如从一个数据库管理系统的组件换到另一个），然后当更多代码可用时再扩大范围。易安装性在产品的所有组件都能正常工作之后才能测试。由于可移植性必须在系统设计时已经考虑，并且内置在产品中，因此这个质量特征在系统设计和系统架构的早期阶段就非常重要。架构和设计的评审对识别潜在的可移植性需求和问题（例如，对特定操作系统的依赖性）富有成效。

4.8.1 易安装性测试

易安装性测试是检查软件的安装以及为软件安装到目标环境的文档化的安装手册。它可以包括安装操作系统的软件或在客户端 PC 电脑上安装软件产品的安装软件（向导 Wizard）。

典型的易安装性目标包括：

- 验证软件能根据安装手册（包括任何安装脚本的执行）上的指示，或通过使用安装向导顺利地进行安装。其中包括针对不同的软硬件配置，选择相应的选项进行安装，以及进行不同级别的安装（如初始安装或系统更新）。
- 测试安装软件是否能够正确处理安装过程中所出现的失效（例如无法安装某些 DLL）现象，而不至于使系统处于某个不确定的状态（例如，软件只安装了一部分或造成错误的系统配置）
- 测试部分（不完全的）的安装 / 卸载能否完成
- 测试安装向导是否能成功识别无效的硬件平台或操作系统配置
- 测量是否能在指定的时间段内或少于指定的步骤数内完成整个安装过程
- 验证软件是否可以成功降级（安装一个早期的版本）或卸载

在易安装性测试之后通常还要进行功能性测试，以检测任何在安装过程中可能引入的故障（例如，不正确的配置，不可用的功能）。在易安装性测试的同时一般还会进行易用性测试（例如，验证用户在安装过程中是否获得易懂的指示和反馈 / 出错信息）。

4.8.2 共存性/兼容性测试

互不相关的计算机系统当它们能在相同的环境中运行（例如，相同的硬件）而不影响彼此的行为（例如，资源冲突）时，可以看作是兼容的。当需要在一个新的已经安装了应用程序的环境中安装一个新的软件或进行软件的升级，这时需要进行兼容性测试。

在没有安装其它应用程序的环境中，可能检测不出软件的兼容性问题，但如果将其配置到另一个安装了其它应用程序的环境（如产品环境）时，则可能会发生兼容性的问题。

典型的兼容性测试目标包括：

- 评估在同一个运行环境中加载其它应用程序所导致的功能性的负面影响 （例如，当一个服务器运行多个应用程序时可能产生资源冲突）
- 评估由于操作系统的修补和升级造成对每一个应用程序的影响

在规划预定的目标环境时，应该分析兼容性问题；通常是在系统测试和用户验收测试已顺利完成后才执行兼容性测试。

4.8.3 适应性测试

适应性测试检查一个给定的应用程序是否能在所有预期的目标环境中（硬件、软件、中间件、操作系统等）正常工作。因此，自适应系统是一个开放的系统，它的行为能适应环境的变化或适应自身部分系统（子系统）的变化。在规范适用性测试时必须对预定目标环境的组合进行识别、配置并提供给测试团队，选择一组功能性的测试用例在这些环境中测试，而这些测试用例能在环境中检查应用程序的各个组成部分。

适应性还涉及到通过完成一个预定过程将软件移植到各种特定运行环境的能力。测试可以对该过程进行评估。

适应性测试还可以与可安装性测试一起进行，通常情况下，随后辅以功能测试，以检验软件在适应其它运行环境时是否会出现问题。

4.8.4 易替换性测试

易替换性测试侧重于系统中的软件组件替换成其它的组件的能力。这对那些在特定的系统组件中使用商业现货软件（COTS）的系统来说尤为重要。

如果在完整系统的集成过程中存在可选的替代组件，则易替换性测试可以和功能集成测试平行进行。可以通过对系统架构或系统设计的技术评审或审查来对易替换性进行评估，重点就是为可能替换的组件定义明确的接口。

5. 评审 - 165 分钟.

关键词
反面模式（anti-pattern）

评审的学习目标

5.1 简介
TTA 5.1.1 (K2) 解释为何评审准备工作对技术测试分析师很重要

5.2 在评审中使用检查表
TTA 5.2.1 (K4) 根据大纲提供的检查表来分析架构设计及识别问题
TTA 5.2.2 (K4) 根据大纲提供的检查表来分析一部分代码或伪代码及识别问题

5.1 简介

技术测试分析师必须积极参与到评审过程中，提供其独特的看法。技术测试分析师应该有正式的评审培训以便更好地理解在任何技术评审过程中技术测试分析师各自所扮演的角色。所有评审参与者必须致力于从严格实施的技术评审中受益。一份完整的技术评审描述，应包括众多评审检查表，参见[Wiegers02]。技术测试分析师通常参与技术评审和审查，并带来可能被开发人员忽略的有关系统行为方面的观点。此外，技术测试分析师在评审检查表和缺陷严重信息的定义、应用和维护方面发挥重要的作用。

不管进行哪种评审，都必须给予技术测试分析师充分的准备时间。包括评审工作产品的时间、检查交叉引用文档来验证一致性的时间，以及确定工作产品完整性的时间。如果没有足够的准备时间，评审可能成为编辑练习而不是一个真正的评审。一次好的评审包括理解所写，确定所失，验证所述产品与其它已经开发出来或还在开发中的产品保持一致性。例如，评审一份集成级别测试计划，技术测试分析师必须将正在集成的项也考虑进去。它们是否已经能够集成？是否有必须记录的依赖性？是否有数据来测试集成点？评审并不孤立地仅限于所评审的工作产品。评审还必须考虑被评审项与系统中其它项的交互。

让正在被评审的产品作者感觉到他们受到批判并不罕见。技术测试分析师应确保任何评审意见都是为了与作者一起尽可能创造最好的工作产品。通过这种方法，所提意见应具有积极性的措辞，且针对工作产品而不是作者。例如，假如一份声明是模糊不清的，最好这样说“我不明白如何进行测试才能验证这条需求是否被正确实现了，你能帮助我去理解吗？”，而不是“这条需求不明确，没有人能弄明白”。

技术测试分析师在评审中的工作是保证工作产品中提供的信息足够用以支持测试工作。如果信息不存在或不明确，那么这可能是需要作者纠正的一个缺陷。通过保持一个积极的态度而不是批判的态度，能更好地接受意见，而且会议会更富成效。

5.2 在评审中使用检查表

检查表是一组在评审过程中用于提醒参与者验证的特定要点。检查表也可有助于避免“个性化/针对性的评审”，例如，“我们每次评审都用相同的检查表，我们不是只针对你的工作产品”。检查表可以是通用的且用于所有评审，或者只关注特定的质量特性或区域。例如，一份通用的检查表可能会验证“shall”和“should”的恰当用法，验证恰当的格式和相似的不符合项。有针对性的检查表应集中在安全性问题或性能问题方面。

最实用的检查表是由单独的组织逐步发展起来的，因为它们反映了：

- 产品的本质
- 本地开发环境
  - 员工
  - 工具
  - 优先级
- 以往的成功和缺陷历史
- 特殊项（例如，性能，安全性）

应该为组织和可能的特定项目定制检查表。本章中所提供的检查表只是作为例子。

一些组织扩展了软件检查表通常的概念使其包括“反面模式”，是指常见的错误，不好的技术，及其它无效的做法。“反面模式”一词来源于“设计模式”这一流行的概念，是常见问题可重复使用的解决方

案，这些方案在实际情况中证明是有效的 [Gamma94]。此外，一个反面模式是一个常犯的错误，经常作为有用的捷径来实现。

需要记住的是，如果需求是不可测的，这意味着它没有被定义成技术测试分析师能决定如何测试的方式，那么它就是一个缺陷。例如，一条声明为“该软件应该很快”的需求是不能被测试的。技术测试分析师如何能确定这软件是快的？相反，如果这条需求这样说“该软件在特定的负载条件下，必须提供最大三秒的响应时间”，如果我们定义了该“特定的负载条件”（例如，并发用户的数目，用户执行的活动），那么这条需求的易测试性将大大改善。这也是一个总体要求，因为这一条需求可以在一个具有一定规模（中大型）应用程序中轻易地产生大量独立的测试用例。从这条需求到测试用例的可追溯性也是至关重要的，因为如果该需求有所变化，所有的测试用例都需要进行评审，并进行所需的更新。

5.2.1 架构评审

软件架构涉及到系统的基本组织结构、系统的组件、组件之间的相互关系、系统环境以及在系统设计和开发过程中所遵循的原则（标准和规范）。[ANSI/IEEE Std 1471-2000] [Bass03]

用于架构评审的检查表，例如，包括查证以下项是否恰当地实现（引自[Web-3]）：

- 连接池 – 通过建立一个共享的连接池来减少与建立数据库连接相关的执行时间的开销
- 负载平衡 – 将负载均匀地分散在一组资源之中
- 分布式处理
- 缓存 – 用本地数据拷贝来减少访问时间
- 惰性实例化（Lazy instantiation）
- 事务并发
- 在线事务处理（OLTP – Online Transactional Processing）和在线分析处理（OLAP - Online Analytical Processing）之间的进程隔离
- 数据的复制”

更多详情（与认证考试无关），可以从 [Web-4] 中找到。[Web-4] 是一篇文章，其调研了 24 个来源的 117 个检查表。讨论了不同类别的检查表项目，且提供了好的检查表以及应该避免的检查表的例子。

5.2.2 代码评审

代码评审的检查表必须非常详细，只有当此检查表是特定为某种开发语言、某个项目和某个公司量身定制时，此检查表才是最有用的，这一点与架构评审的检查表完全一样。在代码级别引入反面模式非常有用，特别对那些缺少经验的软件开发人员更有帮助。

用于代码评审的检查表可以包括以下六个方面：（基于 [Web-5]）

1. 结构
   - 代码是否完整地、正确地实现了设计？
   - 代码是否符合任何相关的编码标准？
   - 代码是否结构合理、风格统一、格式一致？
   - 是否有任何未调用或不需要的程序或不能到达的代码？
   - 代码中是否有任何残留的桩或测试程序？
   - 是否有能被替换成通过调用外部可重用组件或库函数的任何代码？
   - 是否有任何重复的代码块，可以浓缩成单个程序？
   - 存储使用是否有效？
   - 是否使用有意义的符号而不是“幻数”常量或字符串常量？
   - 是否有任何模块过于复杂，且应重组或分割成多个模块？

2. 文档
   - 代码是否以易于维护的注释风格，清楚且充分地文档化？
   - 是否所有注释与代码是一致的？
   - 文档是否符合适用标准？

3. 变量
   - 是否所有变量都以有意义的、一致的、清晰的命名来恰当地定义？
   - 是否有任何多余或未使用的变量？

4. 数值运算
   - 代码是否避免了比较浮点数的异同？
   - 代码是否系统化地防止舍入误差？
   - 代码是否避免了对有巨大数量级差别的数字进行加法和减法？
   - 除数是否测试了零或噪声？

5. 循环和分支
   - 所有循环、分支和逻辑结构是否完整、正确和恰当地嵌套？
   - 是否先测试 IF-ELSEIF 链中最常见的情况？
   - 是否涵盖了 IF-ELSEIF 或 CASE 块中所有的情况，包括 ELSE 或 DEFAULT 语句？
   - 是否所有 case 语句都有默认项？
   - 循环终止条件是否明显且总是能达到？
   - 索引或下标是否在循环之前恰当地初始化了？
   - 循环中的语句是否可以置于循环体外？
   - 循环中的代码是否避免了索引变量在退出循环后继续操作或使用？

6. 防错性程序设计
   - 所有索引、指针和下标是否参照数组、记录或文件范围进行了测试？
   - 是否所有导入的数据和输入的变量都测试过有效性和完整性？
   - 是否所有输出变量赋值了？
   - 每个语句中是否都在对正确的数据元素进行操作？
   - 每个内存分配是否得到释放了？
   - 外部设备访问是否用了访问超时或错误捕捉？
   - 在试图访问文件时是否检查文件是否存在？
   - 所有文件和设备在程序终止后是否保留在正确的状态？

不同测试级别代码评审检查表的更多例子可参见 [Web-6]。

6. 测试工具及自动化 - 195 分钟.

关键词

数据驱动测试（data-driven testing），调试工具（debugging tool），缺陷散播工具（fault seeding tool），超链接测试工具（hyperlink test tool），关键字驱动测试（keyword-driven testing），性能测试工具（performance testing tool），录制/回放工具（record/playback tool），静态分析工具（static analyzer），测试执行工具（test execution tool），测试管理工具（test management tool）

测试工具及自动化的学习目标

6.1 工具之间的集成和信息互换

TTA-6.1.1 (K2) 描述多个工具同时使用时技术层面需要考虑的内容

6.2 定义测试自动化项目

TTA-6.2.1 (K2) 总结建立一个测试自动化项目时，技术测试分析师需要执行的任务
TTA-6.2.2 (K2) 总结数据驱动自动化和关键字驱动自动化的区别
TTA-6.2.3 (K2) 总结引起自动化项目不能达到预期投资回报的普遍技术问题
TTA-6.2.4 (K3) 根据给出的业务流程创建关键字列表

6.3 特定测试工具

TTA-6.3.1 (K2) 总结缺陷散播和缺陷注入工具的使用目的
TTA-6.3.2 (K2) 总结性能测试和监控工具的主要特性和实施方面的问题
TTA-6.3.3 (K2) 解释基于网页测试的工具的一般目的
TTA-6.3.4 (K2) 解释工具如何支持基于模型测试的概念
TTA-6.3.5 (K2) 概述支持组件测试和构建（build）流程的工具的目的

6.1 工具之间的集成和信息互换

虽然测试经理承担选择和集成工具的责任，但是为了确保从各种不同领域的测试（例如静态测试，自动化测试，配置管理）中得到准确的数据，可能要求技术测试分析师评审工具或工具间的集成。此外，根据技术测试分析师的编程技能，也有可能将参与创建代码工作以便在一定范围内集成工具。

理想的工具集应该能够消除跨工具间的重复信息。如果测试执行脚本同时保存在测试管理数据库内和配置管理系统内，这不仅需要花费更多的开销，而且还容易导致错误。最好是在一个测试管理系统中有配置管理组件，或者能够集成组织中已有的配置管理工具。整合良好的缺陷跟踪系统和测试管理工具使得测试人员可以在测试用例执行过程中得到缺陷报告，而不用脱离测试管理工具。整合良好的静态分析工具应该能够直接向缺陷管理系统报告任何发现的事件和警告（这个应该是可配置的，否则可能会生成过多的警告）。

从单一供应商购买测试工具套件并不自动地意味着这些工具能恰当地在一起工作。考虑如何集成这些工具时，应当以数据为中心进行集成。数据在交换时必须保证及时性、高精度以及故障恢复，而且无需人工干预。尽管提供一致的用户体验和服务是非常重要的，但是在工具的集成中更应该优先考虑数据的采集、存储、保护和呈现。

组织应该评估采用自动化信息交换所需的成本，并且与可能带来数据丢失或者由于必要的人工干预造成数据不同步的风险进行比较。因为集成可能是昂贵或困难的，这应该成为整体工具策略中重点考虑的方面。

一些集成开发环境（IDE）可能会简化在该环境中运行的工具间的集成。它帮助统一了工具的界面风格并且使人感觉这些工具共享同一个框架。然而，一个相似的用户界面并不能保证组件之间信息交换的顺畅。而完成集成则可能需要编写代码。

6.2 定义测试自动化项目

为了保证成本的有效性，测试工具特别是自动化测试工具，必须仔细地设计和构建。如果没有可靠的系统架构，实施测试自动化策略常会导致这个工具集既需要很高的成本来维护，还不能达到既定目标来实现投入产出比。

一个测试自动化项目应该作为一个软件开发项目。它应该包括架构文档，详细设计文档，设计和代码的评审，组件和组件的集成测试，以及最后的系统测试。如果使用了不稳定或不准确的测试自动化编码，测试可能会被不必要地拖延或是变得复杂。关于测试自动化，技术测试分析师需要做一系列的工作，包括：

- 定义谁对测试执行负责
- 根据组织、时间表、团队技能、维护要求等挑选最合适的工具（注意：可能需要决定自主开发某个工具而不是购买某个工具）
- 定义自动化工具和其它工具间的接口需求，例如测试管理工具和缺陷管理工具
- 选择自动化的方法，例如，关键字驱动或数据驱动（参见 6.2.1）
- 与测试经理一起估算实施成本，包括员工培训
- 计划自动化项目的时间表并且为维护工作分配时间
- 培训测试分析师和业务分析师如何使用自动化工具和为自动化工具提供数据
- 确定如何执行自动化测试
- 确定如何将自动化测试结果和手工测试结果结合在一起

这些工作和决策将会影响自动化解决方案的可扩展性和维护性。必须保证有足够的时间来进行研究，包括调查可用的工具和技术，理解未来组织的计划。特别在决策过程中，一些工作可能需要更多的考虑。这些会在以下章节中详细讨论。

6.2.1 选择自动化方法

测试自动化不仅仅局限于基于 GUI 的测试。也有一些工具在 API 层面通过一个命令行接口（CLI）和被测软件的其它接口来实现自动化测试。而技术测试分析师必须做的第一个决策就是为测试自动化确定最有效的访问接口。

通过 GUI 测试最困难的一点是随着软件的开发进程，GUI 也在不断地变化。由于设计测试自动化代码的方式不同，可能会导致重大的维护负担。比如，如果 GUI 发生了变化，使用测试自动化工具的录制/回放功能可能会导致这些被自动化运行的测试用例（也常被称为测试脚本）不能如期运行。这是因为当测试人员手动运行软件时，被录制的脚本是通过图形对象来抓取交互的。如果被访问的对象发生了变化，录制的脚本就需要更新以反映这种变化。

通过抓取和回放工具可以很方便地创建自动化脚本。测试人员会记录一个测试会话，然后被记录下的脚本则会进一步被修改以提高其维护性（比如，把被记录的脚本中原有功能替换为可重用功能）。

虽然执行的测试步骤几乎相同，但是由于被测试的软件不同，用于每个测试的数据可能也会有所不同，（例如，通过输入多个无效值，并检查每个返回错误来测试一个输入字段的错误处理）。为每个将被测试的值开发和维护自动化测试脚本是非常低效的。通常解决这一问题的技术方案是将数据从脚本中放置到外部存储中，如电子表格或数据库。在每次运行测试脚本时，都会写一些功能以获得特定的数据，使得在提供一组测试数据输入值和其预期的结果值的情况下（例如在文本域的一个显示的值或一个错误消息），单个脚本也能工作。这种方法就被称为数据驱动。当使用这种方法时，测试脚本将被开发来处理所提供的数据，用具和所需要的基础架构，以支持一个或一组脚本的执行。同时由熟悉软件业务功能的测试分析师创建在电子表格或数据库中放置的实际数据。这种分工使得负责开发测试脚本的人（例如技术测试分析师）能够专注于智能自动化脚本的实现，而测试分析师负责维护实际的测试。在多数情况下，一旦自动化已经被实现并开始测试了，测试分析师将负责执行测试脚本。

另一种方法被称为关键字或动词驱动，它更进一步将对提供的数据进行操作的行为从测试脚本进行分离[Buwalda01]。为了完成操作的行为进一步从测试脚本分离，业务领域的专家（例如，测试分析师）会创建一个抽象的元语言，这种语言只是对行为进行描述而不可直接执行。抽象元语言的每个语句描述了一个（或部分）可被测试的业务流程。例如，业务流程的关键字可以是 “Login”、“CreateUser”或者“DeleteUser”。一个关键字描述了一个在应用领域可以被执行的抽象行为。而具体的行为描述了与软件接口本身的交互，如：“ClickButton”、“SelectFromList”或“TraverseTree”。这些具体的行为也可以用于测试图形用户界面（GUI）的功能，但它们并不完全适合现有的业务流程关键字。

一旦测试脚本的关键字和数据已经可用，测试自动化人员（例如，技术测试分析师）将把与业务流程相关的关键字和所属的操作行为转换成可执行的测试自动化代码。这些关键字和操作行为，以及要使用的测试数据，可存储在电子表格或通过支持关键字驱动的自动化测试的特定工具来导入。测试自动化的框架将这些关键字实现为一组由一个或多个可执行的函数或脚本。工具读取由关键字写成的测试用例，并且调用相应的测试功能或测试脚本。可执行脚本以一种高度模块化的方式实现，以便能够方便地映射到特定的关键字。为了实现这些模块化的脚本编程，需要一定的编程技巧。

这种将业务逻辑知识与测试脚本实现的实际编程明确分工的方法能确保测试资源的最有效利用。技术测试分析师可以作为测试自动化专家的角色有效地应用其编程的能力，而不需要具备很多业务领域的专业知识。

通过从不断变化的数据中分离代码可以避免测试自动化的不断修改，提高整体代码的维护性并提高自动化的投资回报率。

在设计任何测试自动化中，重要的是要预测并处理软件失效。如果发生失效，自动化工程师必须确定软件应该如何处理。是记录这个失效接着继续进行测试？还是终止测试？这个失效能不能通过特殊的操作（例如在对话框中点击一个按钮）或是在测试中增加一个延迟来处理呢？当失效发生时，未处理的软件失效可能会破坏后续的测试结果，并导致正在执行的测试出现问题。

同样重要的是需要考虑系统在测试初始和测试结束时的状态。需要确保在测试执行完成后系统返回到一个预先定义的状态。这将允许可以反复执行一个自动化测试套件而无需人工复位系统。要做到这一点，测试自动化必须，例如，删除所生成的数据，或者改变在数据库中记录的状态。自动化框架应该确保测试结束时实现一个适当的终止（例如在测试完成后退出）。

6.2.2 自动化的业务流程建模

为了实现关键字驱动的测试自动化方法，必须以基于抽象的关键字驱动语言对被测试的业务流程建立模型。重要的是要让用户（在项目中很可能是测试分析师）能直观的进行测试。

关键字通常用于映射抽象的业务与系统的交互。例如，“Cancel_Order”可能需要检查任务（Order）是否存在、验证要求注销（Cancel）的人是否有访问权限、显示应该注销的任务、并要求确认注销。测试分析师使用关键字的序列（例如，“Login”，“Select_Order”，“Cancel_Order”）以及相关的测试数据来定义测试用例。下面的例子描述了一个简单的关键字驱动输入表，用来测试软件处理用户帐户的能力（例如，添加、重置和删除用户帐户）：

关键字	用户	密码	结果
Add_User	User1	Pass1	User added message/用户已添加信息
Add_User	@Rec34	@Rec35	User added message/用户已添加信息
Reset_Password	User1	Welcome	Password reset confirmation message/密码已重置的确认信息
Delete_User	User1		Invalid username/password message/无效用户名/密码信息
Add_User	User3	Pass3	User added message/用户已添加信息
Delete_User	User2		User not found message/用户未找到的信息

使用该表的自动化脚本会寻找该自动化脚本所使用的输入值。例如，当它到达关键字“DELETE_USER”所在的行时只需要有用户名的值。而在添加一个新用户时就必须要有用户名和密码。也可以引用一个数据存储作为输入值，就像第二行“ADD_USER”的情况，这里关联的一个数据的引用而不是实际数据本身。增加了访问数据的灵活性，在测试执行时也能改变数据。这就是数据驱动技术与关键字驱动技术结合的方案。

要考虑的问题如下：

- 关键字的颗粒度越高（越精细），越能覆盖具体和特定的场景，但是通过抽象语言进行的维护工作却变得更加复杂了。
- 如果测试分析师也定义具体的操作行为（“ClickButton”，“SelectFromList”等），则关键字驱动测试就能更好地应对不同的情况。然而，因为这些行为都直接与用户界面（GUI）连接，也会由于变更而导致测试的高昂的维护费用。
- 使用聚合概念作为关键字可以简化开发，但会使得维护变得复杂，例如，可能有六种不同的关键字共同创建一个数据记录。是否应该创建一个连续调用了六个关键字的单个关键字来简化操作呢？

- 无论对关键字语言做了多少分析，还是会需要新的或不同的关键字。对一个关键字来说有二个不同的方面（例如，所描述的背后的商业逻辑和所执行的自动化功能）。因此必须创建一个能同时处理这两个方面的过程。

基于关键字驱动的测试自动化可以显著降低测试自动化的维护成本，但它的开发更昂贵，也更困难。此外，为了实现测试自动化，并能够获得预期的投资回报，这就需要花费更多的时间来正确设计。

6.3 特定的测试工具

本节包括的工具方面的信息，大部分是针对技术测试分析师，并且可能超越了在高级测试分析师教学大纲 [ISTQB®_ALTA_SYL] 和基础级教学大纲 [ISTQB®_FL_SYL] 中有关工具的讨论内容。

6.3.1 缺陷撒播/缺陷注入工具

缺陷撒播工具主要应用在代码级，系统化生成单一或某些类型的代码错误。这些工具故意在测试对象内插入缺陷，为的是能对测试套件的质量进行评估（例如它们发现错误的能力）。

缺陷注入工具主要考虑的是在非正常条件下测试对象的故障处理机制。缺陷注入工具故意提供不正确的输入到软件中，以确保该软件可以处理故障。

缺陷散播工具和缺陷注入工具主要都是技术测试分析师在使用，但在测试新开发的代码时，开发人员也有可能使用这些工具。

6.3.2 性能测试工具

性能测试工具有两个主要的功能:

- 生成负载
- 给定负载下系统响应的测量与分析

生成负载是通过预先定义的运行概况作为脚本来实现（参见 4.5.4 节）。脚本最初可能为单个用户而捕获（可能用到录制/回放工具），然后使用性能测试工具实现特定的运行概况。实现时必须考虑每个事务（或一系列事务）处理数据的变化（数据波动）。

性能测试工具通过模拟大量并发用户（即“虚拟”用户）生成负载，按照其指定的运行概况来生成特定的输入数据。与个别测试执行自动化脚本相比，许多性能测试脚本是在通讯协议层再现用户与系统交互，而非通过图形用户界面来模拟与系统的交互。通常在一定数量上减少了单独的测试“会话”。某些负载生成工具也可以通过用户接口来控制应用程序，从而更加准确地测量系统在所生成的负载下的响应时间。

性能测试工具在测试执行期间或测试执行后为分析采用各种测量数据。这些测试工具所采用的典型的度量与所提供的报告需要关注以下方面：

- 整个测试中虚拟用户的数量
- 由虚拟用户产生的事务数量和种类以及事务的输入率
- 对特定的、由用户请求的事务的响应时间
- 系统负载对应响应时间的报告和图表
- 对资源使用情况的报告（例如，随着时间推移的使用情况，以最小值和最大值来表现）

性能测试工具实施时主要考虑因素：

- 生成负载所需的硬件和网络带宽

- 被测试系统所使用的通讯协议和测试工具的兼容性
- 工具的灵活性以保证不同的运行概况易于执行
- 监视、分析和报告所需的功能

由于开发需要很大的投入，性能测试工具通常都是采购的，而不是内部开发的。然而有时候由于技术的限制，现有产品不能满足需求，或者需要的负载概况和实施都相对简单，也可以自行开发特定的性能测试工具。

6.3.3 基于网页的测试工具

各种开源和商业专用的工具，可用于网页测试。下面的列表显示了基于网页测试工具的常见用途：

- 超链接测试工具用来扫描和检查在网站上是否有损坏或丢失的超链接
- HTML 和 XML 检测工具是用来检查网站创建的页面是否符合 HTML 和 XML 的标准
- 负载模拟器是用来测试当大量用户连接时服务器将如何应对
- 轻量化的自动化执行工具和不同浏览器一起工作时的表现
- 工具通过扫描服务器来检查孤儿（未链接）文件
- HTML 特定的拼写检查
- 风格样式表（CSS- Cascading Style Sheet）检查工具
- 检查是否违反了标准的工具，比如，美国的第 508 条无障碍标准或欧洲的 M/376
- 发现各种安全问题的工具

[Web-7] 是个不错的开源网页测试工具。这个网站的组织建立了互联网的标准，同时也提供了多种工具来检查违背标准的错误。

一些包含网络蜘蛛引擎（web spider engine）的工具也能够提供有关信息，页面的大小，下载它们所需要的时间，页面是否存在（例如，HTTP error 404）。为开发人员，网站管理员和测试人员提供了有用的信息。

测试分析师和技术测试分析师主要是在系统测试阶段使用这些工具。

6.3.4 基于模型测试的工具支持

基于模型的测试（MBT）是依靠正式模型的一种技术，如有限状态机是用来描述由软件控制的系统的预期执行行为。商业 MBT 工具（参见[Utting07]）通常会提供一个引擎，它允许用户“执行”模型。有趣的执行线程可以被保存并作为测试用例。其他可执行模型如 Petri 网和 Statecharts 也支持 MBT。MBT 模型（和工具）可以用于生成大量不同的执行路径。

在一个模型中可能产生大量的执行路径，而 MBT 工具可以帮助减少其数量。

使用这些工具进行测试可以为被测软件提供不同的思路，从而发现一些可能被功能测试遗漏的缺陷。

6.3.5 组件测试工具和构建工具

虽然组件测试自动化工具和构建（build）自动化工具在大多数情况下是开发人员的工具，但在很多情况下由技术测试分析师维护，特别是在敏捷开发背景下。

组件测试工具往往是特定的一种语言，该语言用于模块编程。例如，如果编程语言是 Java，JUnit 可能会用来做自动化单元测试。许多其他语言有自己特定的测试工具；这些统称为 xUnit 框架。这样一个框架为测试对象生成每一个类，从而简化了程序员需要在组件自动化测试时需要做的工作。

在较低的层面上，调试工具可以协助手动组件测试，允许开发人员和技术测试分析师在执行过程中改变变量的值，并且在测试过程中逐行地扫描代码。当一个失效被测试团队报告出来，调试工具也能够帮助开发人员分离并判断代码中的问题。

构建自动化工具通常可以使得在任何时候，一旦有组件被改变后，就会有新的内部版本自动被触发。当内部版本完成后，其他工具就会自动去执行组件测试。围绕内部版本流程的自动化通常出现在持续集成的环境中。

这套工具在设置正确的情况下能对即将测试的内部版本的质量有积极的影响。如果由于程序员做的改变导致了一个缺陷，它通常会引起自动化测试的失败，在内部版本发送到测试环境前，触发机制将立即调查失败的原因。

7. 参考资料

7.1 标准

以下标准皆列于这些相应的章节

- ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems /软件密集型系统的构架描述的推荐做法
  第 5 章
- IEC-61508
  第 2 章
- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE) /软件工程 – 软件产品质量要求和评估（SQuaRE）
  第 4 章
- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality/软件工程 – 软件产品质量
  第 4 章
- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992./ 在机载系统和设备认证的软件的注意事项
  第 2 章

7.2 ISTQB ®文档

- [ISTQB®_AL_OVIEW] ISTQB ®Advanced Level Overview, Version 2012 / ISTQB® 高级大纲 - 概述 2012 版
- [ISTQB®_ALTA_SYL] ISTQB® Advanced Level Test Analyst Syllabus, Version 2012 / ISTQB 高级大纲 – 测试分析师 2012 版
- [ISTQB®_FL_SYL] ISTQB® Foundation Level Syllabus, Version 2011 / ISTQB® 基础级大纲 2011 版
- [ISTQB®_GLOSSARY] ISTQB® Glossary of Terms used in Software Testing, Version 2.2, 2012 / ISTQB® 在软件测试中使用的术语词汇

7.3 书籍

[Bass03] Len Bass, Paul Clements, Rick Kazman “Software Architecture in Practice (2nd edition)”, Addison-Wesley 2003] ISBN 0-321-15495-9
[Bath08] Graham Bath, Judy McKay, “The Software Test Engineer’s Handbook”, Rocky Nook, 2008, ISBN 978-1-933952-24-6
[Beizer90] Boris Beizer, "Software Testing Techniques Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3

[Beizer95] Boris Beizer, “Black-box Testing”, John Wiley & Sons, 1995, ISBN 0-471-12094-4
[Buwalda01]: Hans Buwalda, “Integrated Test Design and Automation” Addison-Wesley Longman, 2001, ISBN 0-201-73725-6
[Copeland03]: Lee Copeland, “A Practitioner's Guide to Software Test Design”, Artech House, 2003, ISBN 1-58053-791-X
[Gamma94] Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 0-201-63361-2
[Jorgensen07]: Paul C. Jorgensen, “Software Testing, a Craftsman’s Approach third edition”, CRC press, 2007, ISBN-13:978-0-8493-7475-3
[Kaner02]: Cem Kaner, James Bach, Bret Pettichord; “Lessons Learned in Software Testing”; Wiley, 2002, ISBN: 0-471-08112-4
[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon, "TMap Next for result-driven testing"; UTN Publishers, 2006, ISBN: 90-72194-79-9
[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976. PP 308-320
[NIST96] Arthur H. Watson and Thomas J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, Prepared under NIST Contract 43NANB517266, September 1996.
[Splaine01]: Steven Splaine, Stefan P. Jaskiel, “The Web-Testing Handbook”, STQE Publishing, 2001, ISBN 0-970-43630-0
[Utting 07] Mark Utting, Bruno Legeard, “Practical Model-Based Testing: A Tools Approach”, Morgan-Kaufmann, 2007, ISBN: 978-0-12-372501-1
[Whittaker04]: James Whittaker and Herbert Thompson, “How to Break Software Security”, Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0
[Wiegers02] Karl Wiegers, "Peer Reviews in Software: A Practical Guide", Addison-Wesley, 2002, ISBN 0-201-73485-0

7.4 其它引用

下列参考地址指向因特网上可用的信息。当本高级大纲出版时已检查过这些引用，尽管如此，如果这些参考地址不存在的话，ISTQB®不承担任何责任。

[Web-1] www.testingstandards.co.uk
[Web-2] http://www.nist.gov NIST National Institute of Standards and Technology,
[Web-3] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-4] http://portal.acm.org/citation.cfm?id=308798
[Web-5] http://www.processimpact.com/pr_goodies.shtml
[Web-6]http://www.ifsq.org
[Web-7] http://www.W3C.org

Chapter 4: [Web-1], [Web-2]
Chapter 5: [Web-3], [Web-4], [Web-5], [Web-6]
Chapter 6: [Web-7]

8. 索引