目 录

重复测试的必要性......1
如何有效地与开发人员一起工作......7
通过 ODC 方法改善软件测试：3 个案例研究......32
自动化脚本编写方法......62
基于缺陷分布的质量目标分解和质量预测体系......68
UNIX 下自动化测试实践......72
在 QTP 中随机取下拉菜单的值......95
需求不明确的情况下如何做测试......99
基于 TMM 的软件测试过程评估......104
如何实现基于非标准控件开发的软件的自动化测试......113

重复测试的必要性

作者：James Bach　译者：肖艳霞

以发现错误为目的的测试与在雷区排查地雷类似，如果你只沿相同路径重复查找，你不会找到大量地雷，相反，这是一个躲避地雷的一个很好的方法。现代软件代表的空间比雷区复杂很多，所以设定一定数量的路径更是问题，例如，成百，成千，或成万的路径。当不断沿着这些路径查找时，会发现每一个重要的错误。一个测试团队在几周或几个月内尽他们最大努力执行的所有测试并不能涵盖可能发生在一个产品的所有情况。

雷区分析实际上以另一种方式说明测试是一个采样过程，并且我们需要的是一个比较大范围内的样本，而不是一个多次重复使用小样本，因此，雷区给我们的启发是：执行不同的测试，而不是重复相同的测试。

但是我所说的重复相同的测试指的是什么呢？我们知道，没有测试是可以精确的重复执行，就好像你不能精确的沿着你的足迹往回走，你可以很接近，但你总会有一点偏移。重复执行一个测试是否意味你第二次执行一个测试时，你要确保日光以同样的角度照射在你的鼠标上？可能，别以为这只是一个玩笑，我确实碰到过一个由日光照射在鼠标内的光学传感器触发的错误。所以你是不可能确保什么因素会影响一个测试，然而，当你测试时，你有一个确定的目标和对系统确定的预测，你就很可能可以针对目标和预测通过考虑以下方面：A)你知道的和 B)你关心的和 C)不是太昂贵的来重复测试。这并不是什么很难处理的问题。

因此，我所说的重复测试是指一个测试包含了在其它的测试中已经确定的元素。重复测试就是重复之前测试某些方面。以雷区分析的理论解释就是：最好尝试你没有做过的东西，然后重复你已经做过的东西。

如果你不同意这种想法，或同意，请做更深入的阅读。因为这种分析过于简单化！事实上，即使多元化的测试是很重要和强大的，即使是反对重复测试的理由一般情况下是有效的，我还是知道 10 种例外情况。以下是 10 个具体的理由，关于在某些特殊的情况下，重复测试并不是不合理的，相反，它可能相当重要。

出于技术原因，你可能需要理性的重复测试…

1. 再充电：当存在一个新的问题或一个旧问题的重复出现，可以通过现有的具体测试发现的明确可能性，或如果一个旧测试应用到一个新代码基础的时候。这就包括重新执行一个测试以确保问题已经修复，或者一个接一个地在较早期的版本上重复一个测试，因为你希望知道一个具体的问题或行为在什么时候出现的。这也包括对一个安装在新操作系统的，相同的软件上重新执行一个旧测试。换言之，在被测试的技术发生变化时，一个旧测试可以"再充电"。注意，再充电的作用并不一定意味着你应该执行相同的旧测试，无理性的这样做不必要的。

2. 间断：当可能因为一些你无法控制的相关重要变量，你怀疑正确的执行一个测试能否确保发现一个错误的时候。对于你来，执行一个测试，严格来说是和你以前已经执行过的，相同的一个测试，可能会找到之前已经存在，但直到无法控制的变量以某种方式起作用才可以发现的错误。这和一个赌徒在第一次失败后，还继续在老虎机上赌是一样的道理。

3. 重试：当你不确定一个测试在其它的时候是否被正确的执行时，这种情况的一种处理方法就是让几个测试员沿着同样的测试说明执行测试，检查他们是否得到相同的结果。

4. 改变：当你修改了一个测试中的重要部分，但同时保持其它部分不变时，即使这个测试的一部分元素是保持不变的，但对整个测试来说，它是新的，并且可能会引发新的行为。之所以会对一个测试作改变，那是因为虽然之前的测试涵盖了某些方面，但涵盖的范围还不足够。一个常见的修改测试形式就是以同样的方式操作产

品，但过程中采用不同的测试数据。区分改变，间断，和重试的关键就是在改变测试时，你可以直接控制改变，改变是有目的性的，间断是偶然事件的附带结果，并且因为偶然元素才会重试一个测试。

5. 基准：当重复的测试包含一个绩效标准时，这个标准的值是与相同测试之前的执行情况比较得出的。当历史测试数据被用作预期结果，那你得保证你执行的测试历史数据有可比性。掌握测试常量未必是使结果可比的唯一途径，但它可能最好的选择。

出于商业原因，你可能会理性地重复测试…

6. 便宜的：当重复测试有一定的价值，并且以一个新的，不同的测试相比，是十分便宜的时，但，这些测试可能不能确保产品的质量。

7. 重要性：当可以通过重复测试发生的问题可能比其它检测出来的问题更重要时，产品行为的重要性不一定是均衡分布的，有时一个特殊的问题只因影响重要用户一次就可能被认为难以接受的("决不允许再次发生"的情况)。这并不意味着你要执行完全相同的测试，只要重复的测试包含可检测出问题的足够相似的元素就可以了(查看改变部分)。注意，不要混淆问题的重要性和测试的重要性。一个测试，即使它检测出来的问题不是很重要的，但因为许多原因它也可能是重要的。同样，为了查找一个重要的错误而花费过多的精力在一个测试上，而忽略了其它同样可以，或更利于检测出那种问题的测试，这样的错误不要犯。

8. 充分性：当你重复的测试代表唯一值得重复的测试。这和病毒检测原理相似：可能对一个普通用户来说，一个重复的病毒扫描就足够了，持续变化的病毒测试则并不必要。然而，我们可能对重复的测试引入某些改变，因为我们不知道哪些测试真正值得重复的，或者说我们通过重复测试不能得到充分测试的效果。

9. 被授权的：当根据合同规定，管理法令或规则，你必须执行完全相同的测试，然而，即使在这些情况下，往往只执行授权的测试并不必要，你可以执行新的测试而不违法规则。

10.不重视/避免：当“测试”因一些原因而不是为了查找错误被执行时，例如，培训，演示(例如，你非常希望能通过的演示给用户看的验收测试)，或把系统置于某个阶段。如果你执行测试目的之一是避免缺陷，那么改变的主要论据就不成立了。

通过与测试课程的学生和同事近百小时的辩论，我收集了上述原因。我的很多同事都喜欢这些原因的其它表达方式和分类方法，我这样表达和分类并没特殊的原因(除了其它的一些分类方式会有很长的列表，并且列表中会出现非常相似的项)，重要的是，当我听到一个原因，它与我之前听到的那些不同。我就会添加这个原因到列表中，我在 1997 开始添加了头两个原因，第 10 个原因则是在 2004 年添加的。

雷区应用：一个例子

Ward Cunningham 写过：我相信 TDD[测试驱动设计](和修改)的自动化需求是不可比拟的，因为我们正在进行的研究是用于一个程序在测试之前的最好表达，而不是最好测试。

以下是我对上面理论的理解：

你的单元测试可能成功或失败，你编写这些单元测试，使它们在关注的预期结果被干扰的情况下不能通过，所以，你把它们叫做测试并且他们看起来也像是测试

在第一次执行成套单元测试中的任何指定测试时，我们引入雷区批判理论。你第一次执行单元测试时失败，对吧？当然，因为它不是 TDD，否则就会成功。下面的问题是从雷区启发性理论得出的：“使你的测试多样化，而不是单纯的重复执行”

问题：为什么要再次执行它？

解答：例外#1，“再充电”。你重新执行它是因为你已经添加了代码使这个测试能通过，因此重新执行测试并不是多余的，这个测试的价值已经被相关改变的代码重新激发。

问题：在开发的过程，但在第一次测试通过后，为什么不删除它？何必再执行一遍？

解答：原因有几个，再充电还有一点应用，因为你可能在开发过程中不小心破坏了产品。并且可以说大部分的单元测试大多时候不会失败，即使你改变很多代码，其中的一些单元测试很可能不会失败。这样你就有第二个原因了：例外#6，“便宜的”，创建这些测试，执行它们，和保持它们运行的成本非常低，同时这些测试又有一定的价值，即使价值不是很大。对于一些测试，你还可以有第三个原因：例外#7：“重要性”。对于很多的单元测试来说，失败可以预示一个非常严重的问题。如果你在测试的软件有一些非常复杂的代码，或包括了很多互相作用的子系统，这时因为例外#2，“间断”。你可能需要重复测试。可能在第 43 次执行后，因为测试中的概率因素，一些测试会失败，所以，最后一个原因，例外#3，“重试”。这个例外会提醒我们可能之前的测试没有被正确的执行。正如 Ward 你曾经说过，一些问题可能只会在你测试了成百次后才冒出一些苗头。换句话说，经过多次测试后，你可能会找到产品一直存在，但从没注意到的缺陷。

问题：如果我是一个非常好的程序员，并且虽然我写出很好的测试代码，因为我不会在我的代码引入错误，所以这些测试代码不会失败。我有一整套的测试，并且它们都不会失败，那么重复这些测试还有什么意义？

解答：两个潜在的原因，例外#10，“不重视/避免”，你可能文档的形式记载那些测试，以供以后的开发人员使用，并且为了把失败的机率减到最低，你希望他们完全遵循这些说明去执行测试(因此文档的作用会减少)。或者可能你希望你的软件给客户留下一个深刻的印象，但如果测试没有通过，客户会有不好的印象。第二个原因是例外#9，“被授权的”。你可能采用这种方式工作，因为你的组员或经理要求你这样做。这有一点像是避免，除非你确实希望通过授权的方式查找到错误。你在查找这些错误时，你需要使用某种技术去这样做。

因此，从在我提到的种种原因来看，我们可以看到 TDD 的相当

简单的，经常重复执行的单元测试实际与偏向于多样化测试的雷区分析理论不同，但TDD也没有偏离雷区的启示性分析。质疑重复测试的价值总是有理由的，并且雷区分析理论引导我们这样做。

(本文写作过程中得到了以下同事的协助：Doug Hoffman, Michael Bolton, Ken Pugh, Cem Kaner, Bret Pettichord, Jim Batterson, Geoff Sutton，并且还得到了参加"Minefield Debate"课程的学生的帮助，"Minefield Debate"是我测试课程的一部分，我提到的雷区分析理论由Brian Marick在典型测试错误的报告中提出。)

如何有效地与开发人员一起工作

作者：Brian Marick　译者：陈能技

开发人员与测试人员通常不能很有效地工作在一起。开发人员能改变，我们也能改变，大家都能改变。我未能幸运地直接改变其他人的行为。但是我能幸运地改变我自己的行为，从而间接地改进开发人员的行为。而他们的改变又给了我很好的反馈，从而建立起一个有效的循环关系。本文是关于我怎样有效地跟开发人员一起工作的方法介绍。

我希望我的读者，是那些花大部分的时间寻找 bug，跟开发人员讨论，以及写 bug 报告的测试人员。

本文介绍我这些年来学到的三个能把事情做得更好的方法。

定义自己的角色，让开发人员觉得他们需要我在旁边帮助他们。我帮助开发人员在 bug 还没出现之前就把它消除掉，从而减少项目总体成本。

给感到疑惑的开发人员解释我的工作。减少对我做出武断的评价和看法的机会，我会使用与众不同的方法来让我的开发人员相信：bug 报告不应该被看作是威胁。

尽量减少会产生误会和曲解的 bug 报告

这三部分很大程度上是独立的，方法和手段不互相依赖。

一、　选择一个有效的角色

在这一节，我首先描述一下我喜欢的角色和这个角色的日常工作。然后描述这个角色解决的问题，最重要的是，可能产生的新问题。

假设你被告知要测试某个开发人员的工作，也许是增加了一个新特性到产品中。你也许要同时测试多个开发人员的程序，但是我

会在后面的章节覆盖这些复杂的情况。我假设你会在编码阶段开始工作：在开发人员开始写第一行代码后，但在它被完成之前(除了修改 bug 的情况外)。如果你在更早的阶段介入，那会更好，但是我不假设那种情况。如果你在代码完成后才开始进入你的工作，那么这个章节的作用除了帮助说服你承认下次应该早点介入外，就作用不大了。它也不能应用在那些你不会去找一个开发人员的工作的 bug 的地方，例如配置测试、压力测试或其它类型的测试。

测试组是为谁服务的？一个普遍的回答是“顾客”。在这种模型下，测试人员站在顾客那边，为顾客服务，保护顾客的利益。顾客是一个无助的少女，被绑在铁轨上，一辆失控的火车(处于失控状态的产品开发)冲向她。只有测试人员才能救她，阻止火车(阻止产品发布 - 至少要到 bug 都修改完)。

这个模型有很多问题，不仅仅是这个模型几乎是不工作、不生效的。火车往往冲向少女，让她的救世主失落，士气低下、愤世嫉俗、工作效率低下[Marick97a]。然而对于这篇文章的目的而言，有一个问题是非常相关的：把人们想象成坏人的角色，对于想寻求他们帮助来说不是一个好的开端。不要误会：你其实是要花很多时间从开发人员那里寻求帮助的。你需要他们解释程序，这其实是在要求开发人员给你最宝贵的礼物：时间。你会登记 bug 并且希望开发人员聪明地处理它们，而不是寻找各种理由来宣布某个 bug 是一个特征而已，要求有更多的信息，或者干脆开始浪费你的最宝贵的资源：时间。

也许我有点夸张。也许没有开发人员会真正相信你在把他们想象成坏人。大概他们会相信某些更糟糕的事情：你把他们看成会频繁出错的人以致不可信赖。大部分开发人员会被认为是坏人而不是缺乏能力的人。但是不管反应的差别有多小，事实是这个模型给了开发人员和测试人员一个不一致的目标。开发人员希望早点发布；测试人员尝试阻止他们。开发人员的合作顶多是一种处于责任感的态度，甚至可能是不满的抵触态度：不可能达到让测试人员融合到

项目组中的境界。

所以，如果不是为顾客，那么测试组应该为谁服务？

我认为应该是为那位对项目的所有事情负责任，并决定产品是否能发布的人而服务。“所有事情”包括了：了解竞争者在做什么；知道程序员因为疲倦而存在的风险，不能进一步提供好的程序，而是提供很糟糕的代码；清楚承诺没有实现带来的代价，产品延期发布给公司和顾客带来的损失。(见[Bach97]，会有更详细的讨论。)我把那个人叫做“项目经理”。测试组的工作是为项目经理提供其中一项精确的评估：产品的缺陷率(这个产品的 bug 多不多)，主要是从顾客可能看到并关注的角度去看。(关于这个论点的细节，参见[Marick98a]。关于测试组如何报告缺陷数据和其它类型的需要跟踪的信息，参见[Marick97b]。)

让我强调一点，测试人员仍然做的是相同的核心工作产品：bug 报告。只是这些 bug 报告的使用方式不同。不是作为针对和指责产品的证据，而是作为提供给项目经理的信息，以便他做出更好的决定。

现在应该很清晰了，这个新的角色定位帮助你跟开发人员一起合作。从他们的角度看看：测试人员就在身边，不断尝试找出可能存在的问题，时刻准备着扑向 bug 并把它交给项目经理。当然没有人喜欢告密者或搬弄是非的人。

那么如何克服这些问题呢？一个解决办法是一个叫 egoless programming(无私编程)的项目组结构[Weinberg71]，在这种团队里程序员不认为代码是他们自己的扩展，他们不自私，所以他们自己的错误对于他们来说没有那么多的威胁。然而，我不赞成这种方法，因为这意味着开发人员需要改变他们的行为，而不是我要改变。跟我一起工作的开发人员很少有无私的，但是我没有其它的选择，只能跟他们一起工作，接受这个现实的局限性。

我的解决办法是转移到 private bugs(私有缺陷)和 public bugs(公有缺陷)的差异上。私有缺陷是程序员产生的 bug，但在把代码签入

到公共源代码库之前修正了，它可能会对他的程序员同伴们造成潜在的危害，或者对后面的顾客造成影响。但是一旦 bug 已经出现，为了保护同伴，要求它必须被公开。(注意这个重要的区别：它不是为了处罚某个制造这个 bug 的人而公布，而是为了保护其他同伴。)

我作为一名测试人员给这些开发人员带来的信息是：我是来帮助他们把 bug 阻止在 private 阶段，防止他们出现在 public 阶段。转眼间，我变成了他们的盟友，成为专注于帮助他们变得更好的人。我的工作就是像他们的一张完全网一样地服务，专注于那些他们不能避免或自己找出来的 bug。我是一个工具，作为开发人员的扩展(不要忘记，即使我们与开发人员紧密地工作在一起，他们其实通常都会自己找到更多的 bug，比我们找到的要多。)

让我澄清一些东西。假设我的开发人员准备好了一块新的代码并且修复了 3 个 bug。我会马上开始测试。唯一会使我推迟测试的是我已经在为另外一个开发人员在做相同的事情。

当我找到一个问题的时候，我会发 Email 给他。我通常不会把那些 bug 登记到公共的 bug 跟踪数据库中。我不会把它们跟其他测试员、其他开发人员、项目经理一起分享。它们是私有的。我用自己的 Email 目录记录并跟踪它们，除非开发人员要求我把它们公开地跟踪起来。

开发人员可能不能在把代码放到公共源代码库之前修改所有 bug。我一般不同意这种决定，但是我接受开发人员的决定。(这跟我的关于测试组与项目经理的关系的观点是一致的。)如果他不修改 bug，则我会把它从 Email 目录移到 bug 跟踪系统。因为现在这个 bug 是公开的，公众(项目经理、其它开发人员)需要知道它的存在。

我如果不是在找私有的 bug，就是在找公开的 bug。也就是说，我完成了测试的代码已经放到公共源代码库了。(也许它已经放到上面了，但是我没能及早分配到这个测试任务。或者是程序员需要让某些内容独立工作，出于演示 demo 的需要或出于让其它程序员的工作能正常进行的目的。)我在那里找到的 bug 是公开的并且会报告到

公共的 bug 跟踪系统中去。

这样说有点过于简单化了，目的是想说明我的观点：帮助程序员寻找私有的 bug。在现实中，事情会更加复杂点。假设刚刚完成的代码不是非常重要，但是代码中的 bug 已经是 public 的可能对其他程序员造成严重的阻碍。那么我可能会专注于 public 的 bug。我总能得到开发人员的支持，原因在下个大章节解释。有时候，在项目结束之前，我需要确保所有代码都得到了充分的测试。稳定的变更的测试也不能阻止我这样做。

现在什么问题变小了？

为什么我要这么麻烦呢？看起来我是想去巴结一些朋友。朋友是好的，但是公司不会为我的社交生活付钱。公司给我报酬是让我使用一部分权力来达到某些目的，一种减少问题的方法。什么问题？

一般而言，摩擦。

我遵照 John Daly 的原则(注：Noel Nyman 告诉我的。得到了 John Daly 本人的许可)，不断地问自己："我做测试不是找 bug 是做什么？"摩擦会减缓进度。开发人员和测试人员的一些典型摩擦浪费的时间其实可以更好地用在找 bug 上。

我的这种方法还帮助解决其它的问题。

找 Bug 的成本高、找得太迟。

如果一个 bug 能尽早发现，总是会比等到开发人员已经忘记了这个问题相关的内容时再修改的成本要低、修改速度要更快些。同时也可以避免正式的 bug 跟踪数据库的管理成本。

开发人员不测试。

对开发人员普遍的抱怨是他们根本不会自己尝试一下就"把最新的 build 版本扔过墙来"。一测试就停掉了，因此是不可测的，它们又被扔回去。没有意义的工作：浪费时间。

一个类似的问题是开发人员声称一个 bug 已经被修正了，但是最简单的测试就能揭示 bug 仍然存在。我曾经见过一个程序员把一个 6 行代码的函数重复修改了 4 次，每次都会引入一个明显的问题。

明显看到他根本没有对自己修改的程序进行任何的尝试。

看起来我把问题弄得更糟糕了，因为我让测试人员把这个程序员束缚住了，每次修改都及时进行测试。是否我没有给开发人员足够的时间测试？不是的，因为我改变了这里的经济学。在这之前，程序员如果节省了 10 分钟但是浪费了你两个小时，那不会对他造成任何损失。但是现在会了：那两个小时的浪费意味着更多的 private 的 bug 会变成 public 的 bug。他会有更多的积极性去跑那些你为他构建的自动验证测试(“冒烟测试”)，在把代码交给你之前他会尝试测试新加的功能和修改的 bug。他会跟你讨论怎样的分工是有效的。

“请为我 debug 一下这个问题”

开发人员通常要求测试人员为他们做很多 debug 的工作。他们会要求具体的测试用例即使是只有几个有限的步骤。他们会要求要对不同输入的测试的扩展文档即使是他们可以自己有效地尝试进行。

同样地，这个问题也得到了减轻，因为浪费你的时间也会使程序员付出一些东西。更进一步地，因为反馈的循环很紧凑 – 你只需要给开发人员展示错误是什么，而不需要填写详细的 bug 报告，而这些报告可能要等上 3 周的时间开发人员才会看到 – 你为开发人员 debug 的成本降低了。

(译者注：我认为这里作者的 debug 的意思不是指我们通常认为的程序的调试，而是测试人员为了重现问题而做的操作。)

不知道什么是最新的。

测试员通常会抱怨他们不知道什么是新的编译版本中的内容。所以他们不清楚应该测试什么东西。有时候我们没有考虑到这点：对于每个东西都及时更新并通知每个人实在是太麻烦了。有时候则是故意的：开发人员迫切希望增加这个新的特性，但是没能得到批准，然而他们听了 Grace Murray Hopper 的演讲和她的口号“得到宽恕比得到许可更容易”，所以增加了新特性。(注：海军少将 Grace Murray Hopper，已故，在计算机的早期阶段。她被认为是第一个发

现计算机 bug 的人，发现一个蛾虫躺在 Mark I 计算机里。(有些人对这个看法提出一些质疑。）)

对于一个与你一起工作的人很难会欠缺考虑的，尤其是他是帮助你把问题阻止在新特性处于 private 的状态时。同样你也很难对这样一个人隐瞒你的恶作剧。

“停止要求那些愚蠢的文档。”

测试人员在判断产品有没有做它要做的事情之前，需要知道产品应该做的是什么。程序员不会很乐意提供那些文档。

当工作在一起的时候，更多的信息能被非正式地流转。那也许不是很理想，因为比你后来的测试员还是什么都不知道，但是比什么都没有要好。(注：顾客也会什么也不知道，但是现在一般人认为他们通过尝试系统的功能特性来学习和了解，而不是通过阅读测试人员需要的详细的文档。)作为测试员，我也通过自己编写文档来贡献自己的价值(把非正式的信息转成正式的信息)。(这与 Coplien 的唯利是图分析者模式类似[Coplien95]。)

“请做这些不是你的工作的事情。”

测试组看起来在收集令人讨厌的、困难的、往往阻止它们做真正的测试的工作。尤其是测试组被冠以 QA 的头衔时，一个很模糊的术语，任何不想维护配置管理系统或创建系统构建版本的人都会同意 QA 应该做这些事情。(注：确实，这里我为了效果而夸大了。我作为“构建独裁者”超过了一年的时间，我没有发疯，最少在一开始的时候没有。)测试人员帮助开发人员可能导致方向的迷失，例如，编写一些辅助的代码。

你从我的技术作者的经验可能已经猜想到我不反对测试人员帮助项目组。但是，一旦 bug 修改成为开发人员的例行仪式，John Daly 的原则会更让人着迷。

不可测试的代码。

测试人员通常会碰到很多代码的问题是很难重现的，但是用户可能不小心就暴露了这个问题。或者他们不清楚自己做了什么操作。

或者不清楚代码究竟做了什么事情。当开发人员重视你的 bug 的时候，他们更倾向于添加增强可测试性的辅助代码。通常一点点就能帮了大忙([Marick95]，第 13 节）。

开发人员缺乏对 bug 的反省。

当我与开发人员紧密地工作在一起的时候，他们学到了我检测 bug 的技巧。他们会更加有能力防止那些我发现的 bug 的再次出现。这种方式下，我可以“顺流而上”，帮助开发人员设计和计划而不是仅仅对于已成事实的软件进行测试。我不会把这些东西推给开发人员；我会等他们来问。

我可能会引发什么问题？

每个解决方案都可能会带来潜在的新问题。

你必须要早点开始

如果你等到代码都写好了并放到公共的地方了才开始你与开发人员的工作，则剩下的就只有 bug 的修改了，你不能得到很大的益处。

引发对测试人员和程序员的比例的关注

考虑我的这种方法与“扔过墙”的测试方法的区别。

更多的中断驱动。为了维持好的关系，当程序员需要你的帮助时你要快速地反应。在传统的测试中，你拥有更多的自由去安排你的测试任务。

更加依赖结构的知识。在两种类型的测试中，你都必须从用户的角度看一个功能特性。在传统测试中，对功能特性的结构或整个系统的架构的理解会对你的测试有所帮助；它加强了你的以用户为中心的测试，并且有时候允许你忽略某些不必要的测试。但是，当你与开发人员紧密地工作在一起的时候，这些东西的理解就会更加有用：它们是有效沟通的关键并且 – 也同样重要地 – 让开发人员感觉到有效的沟通。

这里我需要小心说明。我不认为你需要成为一名程序员才能了解关于结构的知识。我曾经看到过很多非程序员能把这些学得很好。

我曾经看到一个很棒的程序员高度赞扬一名非程序员因为解释结构给他们听需要他阐明他的思想。

这些区别意味着什么呢？

假设测试人员与开发人员的比例是 1 比 10，而你尝试跟每个开发人员都紧密地工作在一起。

一般请求帮助会以不可预知的间隔发生。你需要安排好帮助他们的顺序。但是那自然会降低你对某些人的价值，所以下次他们就不会那么愿意叫你帮忙了。

你不能跟上每个程序员的子系统的结构。你会需要程序员解释一些东西，而这些东西在他们看来你应该早就知道了。你可能需要他们再次解释某些东西，而这可能会激怒程序员。（在他看来，他刚刚给你解释完。从你这边看来，介于 7 个不同的任务之间，很容易把刚才解释过的东西忘记了。）或者，更糟糕的是，你可能记住的一些东西不再是有效的，导致错误的测试或误报的 bug。

在这种情况下，测试会退化成“扔过墙的测试”，因为在这种局限下“扔过墙的测试”会工作得更好些。为了避免不好的感觉，你也许倒不如就按这种方式测试更好些。

假设比例是 1 比 3 左右。则容易处理些，但是仍然可能会让开发人员失望，因为不能像他想的那样测试（意味着太多 private 的 bug 被忽略了）。你需要谨慎地处理这些问题。注意确保不要过分承诺，导致不能兑现期望。我曾经那样做过。很不好，比不紧密工作在一起还要糟糕。要尤其注意识别出开发人员代码的风险区域，或者高风险类型的 bug，明确说明那些是你将要重点关注的区域。

因为我知道开发人员很容易寄希望于测试人员，我更喜欢被指定与一名开发人员紧密工作在一起，而同时用传统的方式兼顾其他两名开发人员。

你不能从秘诀中学习

假设测试人员和开发人员干得很漂亮，很少 bug 出现在公共源代码。这样的话谁还能从这些少得可怜不值一提的公共 bug 报告中

学到东西呢？Noel Nyman 读了本文的初稿后写到：“当我到了一个新的测试组时，首要的信息源就是 bug 历史。它是洞悉产品的无价之宝，并且是制定回归测试内容的依据。没有谁告诉我关于产品的信息能超过我从旧 bug 中发现的信息的 10%。”其他评论者也指出，开发人员可能学着避免他们自己独特类型的 bug，但是他们不能从看其他开发人员制造的其他类型的 bug 中学到东西。过程改进组也会有类似的问题。

对于这个问题我没有很好的回答，除了作为一个我愿意付出的代价外别无它法。公共的 bug 还是会被发现并登记。在某种意义上，它们是最好的能学习的东西，因为它们逃脱了开发人员和测试人员的眼睛，但是了解私有的 bug 还是会更好些。成功实施过程改进的公司（我从未在这样的公司工作过）可能会更加无私一些，在那里开发人员会更加愿意把 private 的 bug 登记进去，为了公众着想。（有两个评论者 Johanna Rothman 和 Jeanine Brown 曾经看到开发人员是这样做的。）我在下一主要章节的论点会帮助我们朝那个方向前进。我没能有幸在一家公司是能很好地做评审的，但是这样的公司可能已经习惯于共享 private 的 bug，至少在开发人员之间。

控制 Build

我曾经把这样的方法应用到相对单片集成的和自我包含的产品中：我从开发人员手中获取可执行程序然后在我的机器上执行。如果产品被分解成很多块（共享库、配置文件等。）并且开发人员不负责构建和整体地提交，或者没有实用的方法可以单独测试开发人员的每个模块，知道它是真的孤立的，跟其它模块是隔离的（尤其是包含其他开发人员在同时开发的模块时） - 那么你就碰到了配置控制的难题了，它可能吞没紧密工作带来的好处。最好能定期获取完整的 build 版本。

设置了错误的期望值

开发人员可能会期望获得很好的益处但是不付出什么代价。应该小心设置他们的期望值。

你的工作是提供关于 bug 的早期信息。那就意味着所有的任务 – 项目经理认为有足够的质量的代码创建 – 要快速地进行[Maguire94]。然而，如果开发人员修正 bug 时你在找 bug，那么任务的第一部分 – 获得第一个可演示的代码版本并放到源代码控制中 – 可能会延迟。开发人员可能会认为那很恼人或不合适。

虽然你会小心的保证他的时间，但是你还是需要知道很多信息。你需要跟他讨论，或给他发送 Email。这些都会占据他的时间，不可避免地使他觉得受到打扰。承诺尽量减少中断他的工作。例如，为了充分利用他的非工作时间，我曾经在开发人员吃中午饭、早餐、晚饭的时候访问他，或者在送他到机场的时候或取修理店取他的车时。但是有些时候打扰还是避免不了的。

确保你强调了你们的关系会不可避免地引发一些摩擦。你会烦人地在他认为他已经完成后发现一些严重的 bug，让他感觉你为什么不能一开始就发现这些问题。他最好抱怨，即使是不恰当的抱怨，也好过“避而不理”（作为一种逃避不愉快的问题或人的方法而消失）。避而不理不是解决的办法，因为你会坚持。虽然是客气的坚持，但是还是坚持着。（我曾见这样说“我还是会坚持帮助你及时你再也不能忍受。”）

合作可能会失败

紧密的合作关系是对时间的投资。有时候投资免不了得不到回报：

你的程序员是如此的固执以致你尖叫起来 – 只可惜很可能你的尖叫声还没他尖叫着说你固执来得响亮。

程序员可能会看起来故意阻碍或令人误解。（他也许在尝试通过使用公正的手段或不正当的手段来指示你从而节省他的时间。但是有时候他就是不可避免地粗心大意，或尝试隐藏他的无能，或其他什么原因。）

你的期望值没有达到。程序员对你做的事情不高兴。

我个人倾向于向糟糕的投资倾注更多的精力，更多的时间。那

是错误的。有时候你必须承认计划失败并转向另外一个能工作得更好的计划。在这种情况下，后备计划是“扔过墙的测试”：撤退，保持一定的距离来工作，更正式的工作，专注于公共源代码的重大版本的测试。

因为我容易陷入一个角色，我发现清晰地评估合作关系的状态很重要，尤其是在早期。所以我记录下我花在跟开发人员沟通上的时间。有时候我会问自己是否值得投入这么多时间。沟通的人的性格如何？

是花在回答特定的问题上还是花在解释某个观点？或者是关于普遍性的问题？（当我与开发人员纠缠在一起的时候，我发现我会写很长的关于软件测试的基本原则的 Email 来解释我的决定。）

我是否发现自己很小心地写 Email 来避免任何可能的误解，或者在我跟开发人员讨论之前要很仔细地准备自己的言辞？那些是应付低效率的谈话的前期准备，但是它们本身会花费很多的时间。

我在重复自己的论点吗?开发人员呢？

有些低效率是一开始不可避免的。但是后面有没有减少？是否减少得足够快？

我也会问自己从这些得到了什么。

我学到的关于代码的东西能帮助我的测试吗？我忘记了什么好的注意没有？

有什么 bug 是比我预期的要早发现的？

我是否花费更少的时间在写东西、讨论、重新看 bug 报告上？

我改进效率的机会是什么？

当我受到挫折的时候，我会在我暂时离开工作一周后问自己这些问题。然后我平衡一下成本和效益。如果我怀疑我能达到我需要的结果，我会短时间尝试一下新的沟通方式，然后重新评估。或者我会简单地撤退。如果让我陷入的只是一个测试任务，我仅会从那里退出而已。

我不会因此而责怪谁。如果需要受到责备，我会乐于自己接受，

有两个原因。第一，已经有很长时间被浪费掉而没有发现 bug 了，为什么还要花费更多的时间？第二，我确信，我不是足够的好，我不能跟任何开发人员一起有效的工作。

虽然没有必要大事宣扬你的计划改变的原因，但是确保你清楚地跟开发人员沟通，你的管理者，还有开发人员的管理者。他们有权利知道，不告诉他们会把问题弄得更糟糕。

你是明星吗？

一个评论者这样写到："从我的角度看来，你的建议是让我找机会使得跟我一起工作的开发人员看起来很优秀而我看起来是什么都没做...我碰到的每个测试经理都高唱着 bug 数量不是有效衡量测试员的效率的标准的口号。但是真正我为他工作的每个人，在某种程度上，都在使用 bug 数量来挑选、奖励、提拔测试员。"

这个对于我来说不成问题。不管是为了什么原因，我从来没有意识到经理在使用 bug 数量（除非在某种含糊的意义上-像印象之类的）来评价我。如果你的经理这样做了，那我的方法就是一个问题。为了满足开发人员让你牺牲你的工作会有点过分。取样的方式可能会生效：假设你与 4 个开发人员一起工作，一个是比较紧密的，而另外 3 个是用传统方式的。你的经理可能会认为你在这 3 个开发人员身上发现的 bug 数量是你全部工作的代表。(这样的话就取决于你再他们的代码上测试花费的时间，但是这样简单的衡量就像"玩数字游戏的赌博"一样。）用开发人员方面获得证明，尤其是以请求你的服务的方式，会产生奇妙的效果。

个性和经验

有些测试员喜欢协作完成工作；有些则是不合群的。我的这种方法会减少你与其他测试员的交流。你的经理会看不大清楚你在做的事情，如果你没有意识到这一点则会很糟糕。这种方法给没有经验的测试员增加了风险。

有些测试员喜欢制定计划然后执行它。有些则比较能容忍过程的中断。我的这种方法倾向于后者。除了要能容忍，你还必须是能

自我组织良好地完成你的所有工作。

斯德哥尔摩综合症

“斯德哥尔摩综合症”指的是与俘虏联结在一起并且采纳他们的观点的倾向。例如，人质可能成为敌人去观察警察。（注：http://www.vswap.com/confuser/stockhol.htm 上有一个很好的简短的描述，但是当你去看的时候也许已经不见了。）

把测试员看成是程序员的俘虏有点过于戏剧化。但是长期与开发人员在一起工作，你可能采纳他们的观点而放弃了顾客的观点，导致你遗漏了 bug。尤其是你可能遗漏这样的 bug：产品决定这样做的并且是这样做的，但是却是顾客不需要的。可用性问题是这类问题的一个例子。一个科学计算器提供以 2 为基数的对数计算（主要是对程序员来说有用），而不提供大部分用户期望的自然对数的计算，或者一个在线银行系统给用户默认设置与用户名相同的密码。（一个随机的密码会更好，因为很多用户从来不修改他们的密码。）

所以我们要权衡好紧密工作的风险和价值。在大部分项目中，大部分时间里，我相信价值大于风险。“大部分”不是指全部。在安全至上的项目中，如果妨碍测试员吸取程序员关于潜在错误模式的假设，那么带来的附加的成本需要这些独立的测试组来自己承担。

尝试把顾客或其他用户的模型明确下来会减少这种感染的风险。下面这些书教会我如何在跟开发人员讨论时心里想着顾客：[Moore91]，一本关于市场的书，描述塑造目标顾客特性的过程；[Cusumano95]，描述基于行为的计划；[Gause89]，一个主流的描述如何挖掘需求的软件工程的书；[Hohmann97]，提倡用完美的将来时态描述用户行为。（除了[Gause89]，这本书写的很多是与测试不相关的其它东西。）

测试组的一部分应该专注于整个产品的测试，而不是测试孤立的功能特性。这样的测试通常是基于工作流来组织的（用例、场景、任务），最好由领域专家（例如，会计师来测试会计模块的内容）来测试。这些独立的测试人员来寻找那些因为受开发人员感染的测试

人员遗漏的 bug（还有功能模块交叉区域的 bug 和可用性问题）。

二、 解释你自己和你的工作

不管你的角色是什么，你都必须正确地展现自己，这样才能使开发人员对你的看法是你想得到的。

这个章节不假设你使用前一个章节的方法。它适用于所有你需要给开发人员解释测试的情况下。

良好的第一印象

测试人员通常被开发人员武断的评价所伤害。一个很普遍的情况是测试员想要成为程序员，但是开发能力没有达到。所以你的第一个任务是避免太快地被归类了。这里有 3 种方法。

永远不要过分夸大你的编程或技术能力

我曾经参加一个会议，会议上一个经理夸夸其谈“我们在 1971 年就开始了面向对象编程；我们只是没有这样叫它而已”。虽然他在某种意义上是正确的，但是他从那一刻开始失去了他的听众而且永远也找不回来了。

他试图发出的信息是“我是技术型的，就像你一样。技术上的指导可以听我的。”但是程序员接收到的信息是“我想我比你知道得更多。我不需要进一步的学习。我会在痛苦的培训中浪费你的时间，或者让你做愚蠢的事情。”

测试人员夸大他们的编程技能发送的也是同样的讯息。

忽略你自己的技术能力会好一点。很多程序员希望你是这样的。400 多年后，他们回应了 Castiglione 的关于完美文艺复兴追捧者的描述：“少讲点话，多做点事，对于值得赞扬的事不要赞扬自己，假装不知道...” [Castiglione28]。要警惕任何想吹牛一样的话。

在某些偏门的技术区域拥有专长会很有用

但是不吹嘘一下又不足以建立自己的威望。转移是一个好方法。假设你是恰好是公元前第五世纪的希腊有桨船的专家。程序员也会尊敬你，因为他们普遍对所有专家都尊重。在一些古怪的主题上成

为专家，与编程没有什么关系的方面，比在同行领域有专长要更有价值。对你的这些业余专长的尊敬会转移到对你的测试能力的尊敬。我不知道原因，但是很多很棒的程序员也拥有令人惊讶的、与本专业完全不相关的专长。（注：有一个技术专长是很容易用于转移的：测试人员如果非常了解硬件，能很快找出打印机的问题，或者为什么一个以太网卡不工作了，或者能凭记忆配置任何调制调解器。这些问题可能发生在开发人员的机器上。这些是与编程很接近的专长，但是提高了测试人员的形象。）

但是，还是要小心点。我拥有英语文学学位，给了我写文章时使其变得适当的有趣和旁征博引的能力背景。但是我宣称是这方面的专家，则会带来危险和错误。我拥有学士学位只是培训的一个证明而已。如果我尝试把它当成专长，则是冒险陷入某些人的视线中。

置身层次地位之外

我的英文学位不会让我成为专家，但是在其它方面是有用的。如果我是博士学位，但是计算机方面只有学士学位。那么如果在证书展示的场合，我的位置就很明显了 – 会比我想象的低。但是，如果我不提我的计算机学位，而是描述我的英文主修课程，那么突然我就被归类了。我还获得了作为交叉学科人员的威望，专家在这面前感到一点心虚是好事。（因为对英文主修存在普遍的偏见，所以这需要我展示一些技术性的能力后才会生效。）

测试员可以简单地避免被归类。如果你有绿色的头发，几乎每天晚上跟摇滚乐队一起玩音乐，则程序员很难把你归类为“想成为程序员的测试员”。

解释你的工作

然而，我不推荐你把头发染成绿色，那样不够直接。在某些时候，你必须直接让程序员信服你的价值。在这一节我会介绍一种基于程序员民族心理学的方法。民族心理学是一般人们解释为什么大家做某样特定的事情。它是基于人们有信仰和愿望，基于信仰而行动，从而实现他们的愿望。它不同于存在心理学，存在心理学的目

的是发现信仰和愿望是什么，他们产生的原因和导致行动的原理是什么。也不同于流行心理学，因为它不是用来销售杂志或杂志上的广告产品。

对于很多程序员来说，支配他们意愿的是他们的同伴程序员。他们相信要达到他们的愿望需要下面几点特征：

创造简明强大的抽象概念的能力。这与数学上或物理上的“优雅”的概念比较类似。环境因素可能阻止一个好的程序员创造优雅的代码，但是能力和愿望必须被认知道。

掌握很多相关技术的细节的能力。参见[Raymond96]的“A Portrait of J.Random Hacker”，在里面，Eric Raymond 描述为了写一本书如何快速理解一种固定类型语言的帮助手册。(这与前面的有些关联，让程序员看起来是个通才和专才的结合。)

很强的问题解决能力，包括把那些抽象概念的逻辑推理能力应用到技术细节中。

高效率，由代码生产数量衡量。不发布的东西不像发布的东西那样有效衡量（例如，设计文档）。

作为传奇人物的程序员会在面临很多一团糟的问题时，弄清楚问题需要掌握的细节，跟踪问题到核心代码中，排除不清晰的地方，然后想出简单的操作就能使复杂的问题得以隔离和解决，向人们展示所有这些细节问题都可以用恰当的操作的组合来处理，并在一周内产生可工作的系统。

测试人员则会对程序员的自我形象造成威胁，他们会打击程序员的那些特征。他们会展示给人们看到那些抽象概念没有用，细节没有被掌握好，或者是问题还没被解决。这一点也不奇怪，然后，程序员往往会把测试员的注意力从那些基础的概念转移出去，把它看成是对他们写的代码的系统的探索，寻找代码错误。代码错误不是什么大问题。一个对代码错误不重视的程序员可能会失去一些威望，但是仍然可能会被认为是优秀的。程序员可能从来不制造代码错误，但是创建笨拙的抽象概念。

现在，对于测试人员而言，寻找代码错误只是工作的一小部分。概念上的错误，例如不恰当的抽象或错误的假设，更加重要。它们会需要付出更多的代价来修正。

这种期望值的不一致会导致强烈的冲突。程序员会认为你在做错误的工作，使用错误的方法。那么现在是时候发现和解释正确的做法的时候了。程序员是无情的问题解决者。（我的妻子，她是个兽医，在见过我的程序员之前对程序员不了解，后来她对我说："你知道吗？你的朋友对任何事情都有强烈的主张和看法。"）

通过给程序员解释你的工作来避免这些问题。解释可以分成几部分。

解释你的工作与他的工作之间的区别。

表明你拥有程序员不大可能有的技能，完全可以从第一点的原则推理出来。

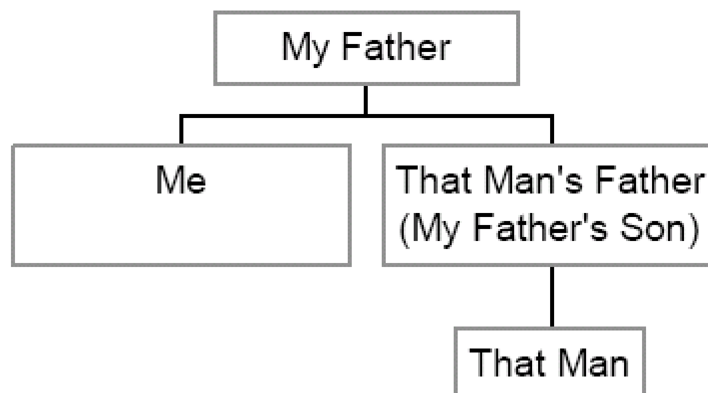
展示你锻炼出来的那些技能是如何帮助程序员实现他们的完整性的，而不要让他们质疑这些能力。你以不可或缺的方式弥补了控制和驾驭这个世界的能力。

让我以一个特定的例子来描述。这是我在写这篇文章的时候想到的，所以只是在一个开发人员身上试过，但是我想这是我发现的最好的描述我的工作的方法。这里是一个谜题：

那个人的父亲是我的父亲的儿子，

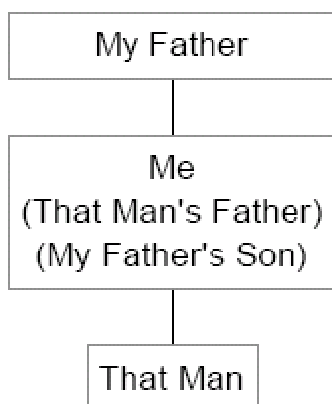
但是我没有任何兄弟姐妹。

这使人迷惑，因为第一行文字描述了下面的情形：



但是这张图与第二行文字描述的相矛盾，因为我的父亲没有其他儿子。

这个谜题很容易解决，如果你意识到文字暗示的“那个人的父亲”除了我之外别无他人。如果是同一个人，你就可以画出下面的图，这与第二行文字不矛盾：



接下来，我会期待任何程序员能快速地解决这个谜题 – 毕竟，他们是问题的解决者。但是问题的关键是某人必须在谜题被解决之前创造出谜题。而问题创建的技能是与问题解决的技能不一样的。在这个例子里，谜题的创建者需要把两个情况搅和在一起，知道父亲不涉及他们自己的儿子或者他们自己不一定是那个人的父亲。技巧在于知道如何误导谜题的解决者。

这个例子与软件测试关系挺大的，因为测试人员要做的其中一个事情就是去考虑是否两个不同的名字会跟同一样东西关联。(“如果输入数据的文件跟输出的 log 文件是一样的呢？代码是否会在重写文件之前读一下？”)

现在，程序员可能会跳出来说，我们也能考虑到这样的情况。他们是通过修改 bug 学到的。然而，测试员比开发人员更能找出问题来：

这是他们唯一要做的事情。他们看到过很多的 bug。他们关于 bug 的思考会更多。他们花更多的时间在形成问题上。

就像谜题的创建者很清楚谜题的解决者一样，测试人员研究程

序员目的是为了洞悉程序员可能忽略的问题的类型。程序员很难考虑到他们不深入思考的问题。（注：要想研究好程序员，你需要成为一名好的观察者。我推荐大家看[Weinberg85]和[Weinberg93]。学习其他人已经观察到的东西，可以参见[Gabriel96]，[Hohmann97]，[Lammers86]，[Levy84]，[Turkel84]，[Weinberg71]和[Weizenbaum76]。）

测试人员还会研究用户，特别在清楚用户知道什么、真实的用户会做什么可能的操作方面。程序员很难去做这些东西。他们可能没有这么多的时间。他们过多地陷入他们的解决方案里，而不会把自己置身于用户的角度。（注：在斯德哥尔摩综合症那一节，我提到一些书可能帮助你学习更多关于你的独特产品的用户：[Moore91]，[Cusumano95]，[Gause89]，[Hohmann97]。我没有读关于大众用户的很多书，但是我可以推荐[Cooper95]，[Norman90]，[Nielsen93]。）

因此，测试人员做的事情是通过呈现某些特定的细节（以测试用例的方式）帮助程序员掌握相关的技术细节，而这些细节本来很可能不会引起他们的注意。不幸的是，你通常太晚才呈现这些细节（在代码写完后），因此揭示的问题是抽象概念或它们的使用方面的问题。但是那是把测试人员过迟地放到项目中的副作用，也是认为测试人员只是执行测试而不是设计测试的不幸的观念的副作用。如果程序员能及早地了解到细节，问题就不会发生了。

程序员帮助测试人员描述他们是怎样测试程序的。这有两个好处。第一，它能帮助测试人员理解程序员的盲点是什么。第二，它照亮了测试人员的盲点 – 当然测试人员也会忽略一些东西。

那是很理性的情况。测试人员通过提供详细的细节给程序员来掌握，从而帮助程序员实现他的真正能力。这是程序员容易接受的论点 – 至少暂时接受。但是在实践中会碰到障碍，它们很可能是跟bug报告相关的。在第三章将讨论这个主题。

你需要解释的关于你自己的其它怪癖

[Pettichord98]包含开发人员与测试人员之间个性的比较。我需

要强调其中的 3 点，因为我看到过它们导致的摩擦。

测试人员能容忍冗长乏味的工作任务；程序员则想办法自动化这些工作。

当开发人员转向测试的时候，往往只是专注于自动化测试。（有时候他们花费了很多的时间在自动化的支持上，而不去写任何实际的测试...）当程序员看到测试人员手工地运行测试时，尤其是重复地执行相同的测试，关于测试人员能力不强的观点又加强了一些。明确地反对手工测试，指出自动化测试的经济有效，但是他们大概不会去权衡一下。由于篇幅限制，我不能展开讨论；见[Marick98b]，让你知道不仅仅需要权衡自动化测试的时间和手工执行 N 遍测试的时间。

测试人员能快速学习；程序员倾向于全面的理解。

每个人都赞赏能快速学习的人，但是程序员可能不会意识到通常测试人员面临的是几乎没有时间去学习一个产品的情况。那诱发了人们“摘挂得比较低的水果”（选择轻而易举的目标去实现）的想法：找出他们需要知道的东西，找到重要的 bug，然后继续。因为程序员重视完整性，这样的测试人员看起来是肤浅的而不是有效率的，除非你给他们解释你的工作。

测试人员相信无知是重要的；程序员认为专业是重要的。

测试人员知道对于产品构成的无知和所谓“正确使用方法”的无知能让他们发现程序员忽略的东西和用户会看到的东西。比用户更专业其实是危险的事情。问天真的问题能产生令人惊讶的答案。需要让程序员知道“天真”是一种有预谋的策略，不是缺点。

三、 报告 Bug

在这一章节我描述怎样报告 bug，从而帮助建立有效的关系。它不要求你使用前两章介绍的方法和技巧。对于更广阔的讨论范围，我推荐[Kaner93]和[Kaner98]。

Bug 报告是展开你和开发人员之间的关系的一部分。你在报告 bug 时做两件事：

提供关于程序状态的信息。这是 bug 报告为人熟知的目的。

提供关于你的信息还有程序员能依靠你的程度的信息。

一开始，开发人员可能会怀疑。一些程序员有测试人员浪费时间在报告无用的 bug 的不好经验。有些程序员则听到很多关于测试人员的恐怖故事。你必须让程序员相信你是有用的人，是个值得在身边的人。有 3 部分内容你可以做。

不要浪费他的时间

Bug 报告不应该让他来猜测你提供的信息。

如果可能，把能重现 bug 的清晰的操作步骤包括进去。写完后，在提交 bug 报告之前尝试一下步骤序列。确保你有一个清楚明确描述的开始状态（例如，确保你退出程序并重新启动）。

把你期待要发生的、实际发生的描述清楚，还有为什么这样不对。

大部分程序员喜欢获得能在最短的步骤下能重现缺陷的 bug 报告。有些则不喜欢，因为那没有真正节省他们的时间。（例如，GNU C 编译器的缺陷报告方法说明书就清楚地告诉你不要因为需要举个小的例子而怕麻烦。）

其它也会导致错误的步骤序列能帮助程序员更快地发现错误的原因，但是不要琐碎地描述每个操作序列的差异。那同样会浪费程序员的时间来决定是否有新的东西。（可能会耗费你很多时间去看那些细节从而决定存在的变化是微不足道的。）

描述一些能成功的场景（令人惊讶地），也会有所帮助

如果问题在他的机器上不出现，要尽快发现他的机器与你的机器的差异在什么地方。检查 bug 报告使其反映对配置的依赖。学习产品是如何对配置敏感的。（当然，这对查找配置方面的 bug 也会有用。在测试的早期，对帮助开发人员扫除障碍很有用。）

保持自己置身事外

冷酷地把任何隐含个人批评的语句从 bug 报告中擦掉。你也许需要让一些 bug 报告“晾”上一个晚上，然后以一种新的眼光来读

它。你可能会发现问问题会比做出武断陈述要安全点。（不要忘记问一些诚恳的问题，你需要学习的还有很多。）问一下你自己当发现你认为是 bug 的问题原来是正确的行为时会有什么感觉。如果你会感觉像个傻瓜，那么同样地，很可能你在 bug 报告中的某些措辞也是有问题的。

不要试图解决你报告的问题。你也许没有足够的很好地解决问题的能力。尝试这样做可能会带来反效果。

用重要的 bug 展示你的价值

你和你的程序员之间可能会对什么是重要的持有不同的观点。你可能也像我一样认为可用性问题时很关键的问题。但是程序员可能不这样认为。你如何解决这个冲突？你可能把一些可用性的问题抛给他，希望说服他修改。或者，你首先抛给他那些他认为重要的 bug，bug 是如此重要以至他认为自己被抓住了。然后，慢慢地建立对某人的观点是有道理的这样的信任，你就可以扩展“重要的”范围，把可用性的 bug 也包括进来。

你或许在想我倾向于后面的方法。我曾经保留不是那么重要的 bug，等到我获得了程序员的尊重后才把它们报告出来。那样做不容易，尤其是那些 bug 是程序员也认可，只是认为优先级要低点。假设我被公共汽车撞了怎么办？如果这个程序员被调到其它任务，现在连最小的 bug 也不能修改怎么办？但是有时候我认为那是能做的最有效的事情。

这里是其它一些让他们关注重要的 bug 的方法。

解释为什么对于顾客来说这是个重要的 bug。如果你的场景被指责说不切实际，那么想出一个更真实的场景来展示这个 bug。如果你的程序员说没有哪个真正的用户会这样做，那么首先仔细地想想他说的有没有道理。他可能是对的。如果不是，收集维护你的观点的好的论据。客服人员是怎样说的？

很多 bug 跟踪系统都会有严重级别和优先级别两个字段。严重级别字段描述 bug 的后果。优先级别字段描述一个 bug 应该多快地

解决。通常，一个严重的 bug 也是一个高优先级别的 bug，但不总是这样的。可能有很好的理由不去修改 bug。作为测试员，你大概不会理会那些原因，你也不应该有权利计划一个程序员的时间。因此，当你报告 bug 时把优先级别字段留空。这样你可以避免把“重要的”定义成“对于我测试来讲是重要的”。（注：就像任何其他的同事一样，如果你强烈不同意推迟修改 bug，你应该考虑进行“斗争”。但是小心地挑选“战场”，在你提出自己的观点之前先让别人做出决定。）

如果 bug 跟踪系统只有严重级别字段，则把它明朗化 – 用语言或实际行动 – 你填入的值不代表优先级。并且尝试添加一个优先级字段。

当你找到一个 bug 的时候，寻找更多的逻辑结果。例如，我曾经测试一些这样的代码，改变名字用于向产品标识用户自己。为了美观的原因，不能在名字的前面和后面加空格。（因为空格是不可见的，会造成其他用户不知道真正的名字是什么。）我发现代码没有实现这个规则。假设我录入一个 bug 标题叫“能创建一些用户很难输入的名称”。那么这个 bug 会很容易地被置为低优先级的。然而，我最近发现用户名是用户本地磁盘的文件名的一部分。我也知道，从以前的 bug 看到并且从与开发人员的 Email 看到，产品在与含有空格的文件工作时会有很多问题。这触发了我去尝试使用以空格为结束符的用户名，使用在与磁盘文件交互的地方。也尝试了包含 DOS 目录分割符‘\’的名称。我会找到更多严重的 bug。具体的细节我不大记得了，但是 bug 报告的标题可以改成“所有过程数据丢失（包含反斜杠符或后面带空格的名称）”。那会更吸引人。

随着你跟开发人员和谐地合作，你会发现不需要花费那么多的时间在 private 的 bug 报告或 Email 上。上面的大部分的建议会使 bug 报告对其他可能的读者更有用：其他测试人员、项目经理、关心开发过程或度量的人。所以你应该继续保持小心地录入 bug 到公共的 bug 跟踪系统。

总结

那么，最后我们所站的位置是什么？

测试人员常常抱怨开发人员是君主。因为那样，测试人员没有得到注意，尊重，他们需要的资源。

但是如果你接受这种情况，然后抱着那些隐喻去做你的工作呢？

我的观点是开发人员通常是慈善和有用的君主，可以他有他的不可避免的用户知识上的不完整性，还有他自己的错误倾向性。有效的顾问尝试完善这位君主，让他更聪明地看待这个世界，不要让他成为他自己的君主。

当然，暴君是需要被打倒的。

感谢

我从 98 年的质量周上获得了这个主题的灵感。可惜我没能取到会议的签到表。所以我只能感谢那些给了我名片的人：Kevin Connery, Reneé Henderson, Nabil Hoyek, Susan Keim, Otto Vinter。我在早期一些人关于这个主题的评论和建议：James Bach, Jon Hagar, Brian Jackson,Mark Johnson, Mark Jones, Monica Lichtenstein, Connie Fleenor Lloyd, Pat McGee,Paula Parash, Pierre Porter, Marcus Porterfield, Linda Rising, Johanna Rothman, Joe Yakich。当我发出这篇文章的修订稿时，我从下面这些人那里得到了格外有用的帮助：James Bach, Jeanine Brown, Kevin Connery, Tim Dyes, W.W. Everett,Peggy Fouts, Nabil Hoyek, Susan Keim, Vince Mehringer, Noel Nyman, Adrienne O'Sullivan, Jeff Payne, Johanna Rothman, Andrew Tinkham, Otto Vinter, Joe Yakich, Ned Young。

通过 ODC 方法改善软件测试：3 个案例研究

作者：M. Butcher, H. Munro, and T. Kratschmer　译者：fennek

正交缺陷分类法(ODC)是一种用于分析软件缺陷的归类方法。它可以结合软件开发过程的一系列数据分析技术，为测试组织提供了一个强大的针对开发过程和软件产品的评估方法。在本篇文章中，会列举三个案例研究来说明如何使用 ODC 改善我们的软件测试。第一个案例描述一个团队如何开发一个高质量的、成熟的产品，并以此来达到他们在测试策略所制定的减少区域缺陷的目标。第二个案例是一个中间件软件项目，他们通过 ODC 确定需要加强的系统测试的范围。第三个案例则描述了一个小型团队，在测试策略不够充分的情况下，他们是如何认识到如果按计划的时间发布会产生的风险，以及通过压后发布时间来获得更为稳定的产品，还有他们添加的那些必需的但很是糟糕的测试场景。所有三个案例都强调了技术团队如何在他们的开发过程和产品评估中使用 ODC，并得到客观的反馈信息。而这些反馈信息会推动组织对（开发和测试等）活动进行确认，以此提高开发和测试的效率和有效性，并最终改善组织的资源管理水平和软件质量。

软件系统的复杂度和规模都在不断地增加着。而商业需求却在要求更短的开发周期，这就迫使软件开发组织去努力地寻找一个在功能、市场时间和质量之间的折衷方法。由于缺乏相关的技能，以及面对时间计划带来的压力和有限的资源，加之软件开发的高度手工化，最终导致无论是大型还是小型的团队组织，都会产生同样的问题。 这些问题包括不完整的设计，无效的测试，低劣的质量，高额的开发和维护费用以及可怜的用户满意度等。而作为一种防止缺陷逃逸的方法，很多公司都在软件测试上投入了更多的资源。此外，为了改善测试的其他方面（例如测试人员的技能水平，自动化测试，新工具的开发和测试过程等），需要有一种方法来评估当前测试过

程，以了解它的优势和缺点，并强调和揭示现存的风险。尽管大家都很清楚，在过程的早期发现缺陷，花费也会少的多，而一旦缺陷在外部发生，则势必需要用更多更昂贵的花费来修复它们。然而，测试人员通常并没有意识到他们所面对的是什么样的特殊风险，以及如何加强测试来达到他们的质量目标。

在本文中，我们会讨论使用正交缺陷分类法（ODC），即一种缺陷分析技术，来评估测试过程。同时为大家列举三个案例研究。前两个案例所讨论的软件产品是在 2000 年初开始使用 ODC 的，那时本文的三位作者正一起在 IBM Hursley 网站上从事 ODC 部署工作。而在第三个案例中出现的项目经理 T. Kratschmer，也是本文的作者之一。

第一个案例中的产品开发人员开始使用 ODC 来评估区域缺陷。尽管产品质量很高，很成熟，没有产生很多的缺陷，但缺陷还是在外部发生了，而且不得不用非常昂贵的花费来修复这些缺陷。这个使用了 ODC 的团队，他们的目标是如何改善测试过程，在内部发现更多的这类（field）缺陷，以此来减少逃逸到外部的缺陷，并降低售后质保的成本（如果用户在购买产品后一段时间内发现产品缺陷，生产商必须免费修理或更换产品）。第二个案例描述了一个大型的中间件软件项目，这个项目具有定义良好的开发过程。尽管有成熟的过程，这个团队还是遇到了几个问题，于是决定看看 ODC 能否给他们一些团队改进上的启发。这个团队对 ODC 关于过程所给出的任何重点启示和在产品发布前需要注意的地方都有着浓厚的兴趣。第三个案例展示了一个非常小的项目。而 ODC 被用来评估产品的稳定性，并指出在产品发布前，为达到一个可接受的测试标准需要加强的地方。由于对测试的关注不够充分和适当，拥有一个客观的衡量方法就显得非常重要了，这个方法同时也方便了他们的管理层与开发团队就未来的决定达成一致。

Overview of Orthogonal Defect Classification

正交缺陷分类法（ODC）介绍

有关 ODC 的介绍性文章一般都会描述 ODC 的概念，及 ODC 计划的定义，同时也会讨论 ODC 各个属性间的关系，并附上一些试验研究的结果。因此，我们在这篇文章中只对 ODC 的细节做一些简短的描述。ODC 方法提供了一个软件缺陷的分类方案和一系列概念，这些概念为分析这些分类聚合的缺陷数据提供了指导。“正交”指由缺陷属性和属性值捕捉到的信息具有非冗余的本质，而缺陷属性和属性的值其本身就是用来对缺陷进行分类的。与几何学中的笛卡尔坐标非常的类似，近十年的研究表明，对于大多数软件开发的问题，这些属性（和它们的值）充分地“填补”了缺陷信息空间中那些有趣的部分。与管理层使用的缺陷分析工具不同，ODC 以技术团队为目标—如开发人员，测试人员及服务人员等。因此，进行缺陷分类的技术团队，应在数据评估中发挥积极的作用，从而决定要实施的活动。本文最后的附件 A 和 B 列出了 ODC 的属性。附件 C 则列出了一个 ODC 缺陷评估中所包含的典型项。

ODC 部署过程　在过去的 10 年间，ODC 的部署过程一直在发展。下列的基本步骤是成功部署 ODC 的关键：

管理层必须承诺进行 ODC 的部署，并根据 ODC 评估所产生的结果来实施活动。

缺陷数据必须由技术团队分类，且应保存在一个易于存取的数据库中。

要定期（或经常）地对已分类的缺陷进行确认，以保证分类的一致性和正确性。

一旦开始确认，则必须定期地（或经常）对数据进行评估。通常，由一名熟悉此项目且有数据分析能力和兴趣的技术人员来执行评估。

向技术团队定期地反馈已确认和评估的结果是非常重要的。这样不仅可以改进分类的质量，也为团队提供了必要的信息，据此他们能够决定（或确定）那些需要改进的活动。对于从技术团队获得必要的承诺而言，反馈也是很重要的。一旦他们看到这些客观的、

量化的数据，以及合理的和切实可行的活动，通常团队对 ODC 过程的承诺和责任感也会随之增加。

一旦团队获得这些反馈，他们就能确定并优先考虑那些要实施的活动。

当这个 ODC 过程与组织的过程整合到一起时，就能实现全组织范围内的互利。对开发过程和其生成的产品进行持续地监控和改进，以使我们能够从开发的最初阶段起建立产品的质量。

缺陷分类和确认　缺陷的分类具有两个不同的时间点（提交和应答）。首次发现或提交一个缺陷时，根据 ODC 的提交者（submittor）属性，activity（活动）、trigger（触发器）以及 impact（影响）来分类。

Activity 指实际的过程步骤（如代码检查，功能测试等），它们表示发现缺陷时正在执行的活动。

Trigger 描述缺陷产生时的环境和条件。

Impact 指对用户造成的感觉或实际的影响。

修改或应答一个缺陷时，根据 OCD 的响应者（responder）属性，target（目标）、defect type（缺陷类型）、qualifier（限定）、source（来源）和 age（码龄）来分类。

Target 表示被修改的对象所属的高层方位（如设计、代码、文档等），即在哪个高层目标中修改缺陷。

Defect type 指缺陷的所固有的性质，即缺陷类型，如算法、初始化等。

Qualifier（适用于 Defect type，对缺陷类型的限定说明）说明所作的修改是丢失的、不正确的、还是无关的代码或信息所引起的。

Source 指缺陷的来源，是源自内部代码、对某个库的重用、平台之间的移植，还是源自第三方的外包代码。

Age 指缺陷对应的码龄（代码的龄期），说明缺陷是出现在新代码中、老代码（基础代码，还是重写的或重新修改的代码中。

有关属性的定义和更多的详细内容可在下列网址中找到：

http://www.research.ibm.com/softeng/.

通常，捕捉 ODC 属性的工具与用来收集其他缺陷信息的工具是相同的。这里，有两种确认数据的方法。其中之一是由具备适当技能的人对已逐个分类的缺陷进行复查（是否有分类错误）。这种方法也许需要等到团队成员对分类类别和用途非常适应的时候才能适用。其二是使用总体的数据分析来帮助确认。尽管这个确认方法更为快速，但是它要求的技能超出了分类法所要求的技能范围。为了执行使用了此方法的数据确认，确认者需要复查缺陷属性的分布状态。如果数据或过程内部包含了不一致的信息，则表明数据质量具有潜在问题，我们需要保持疑问的心态，并通过对一个缺陷子集进行更为详细的复查，从而找到这些问题的所在。虽然在某些情况下，个人对分类法的步骤存在着误解，但一般只限于一个或两个特定的方面，而它们也是容易被澄清的。一旦团队明白了基础概念和它们的用法，数据本身的质量就不再是个问题了。

数据评估　对数据进行确认后，开始准备评估这些数据。在进行评估时，关注点不能只放在单个的缺陷上，还要进行因果分析。准确地说就是对总体数据的趋势和模式进行研究。对 ODC 分类的数据进行数据评估基于 ODC 各个属性之间，以及与非 ODC 属性之间的相互关系，其中非 ODC 属性包括 component（模块），severity（严重性）和缺陷 open 日期。例如，要评估产品的稳定性，我们要考虑—defect type（缺陷类型）、qualifier（限定）、open 日期和 defect severity（缺陷严重性）等属性之间的关系。缺陷类型为“功能缺失”或严重等级高的缺陷，如果有一个增长的趋势，则表示产品的稳定性正在下降。

接下来的三个案例研究，都使用了 ODC 来度量测试的有效性。同时，每个案例的那些需要注意的测试过程，OCD 都给与了重点说明。而评估则为 ODC 过程的最后一个步骤—选择适当的改进活动，提供了必要的信息。案例研究包括了产品的背景信息，以及他们如何实现 ODC 过程和评估的具体细节，最后包括与之对应的改进活

动。

Case Study 1: Learning from the field defects in a mature product

案例 1：从一个成熟产品的区域缺陷中得到的启示。

第一个案例中的产品，拥有一个成熟的开发过程，而且被认为是一个高质量的产品，它为关键性任务应用程序提供了具有工业化实力的解决方案。但是，仍然有一些缺陷逃逸到外部（即区域缺陷），需要提供服务以修复缺陷。这些逃逸给客户的缺陷所产生的成本源自于缺陷本身的影响和产品的服务成本两个方面。在我们开发组织中，一个区域缺陷（逃逸到外部的缺陷）所产生的成本远远大于测试阶段发现此缺陷的成本，而与设计和编码阶段发现此缺陷的成本相比就更高了。即便不考虑实际的成本节约，也非常明显的是，最好不要在设计或编码的阶段引入缺陷，或者至少尽可能早的发现缺陷。

团队应首先以避免引入缺陷为目标，而许多已有的质量活动都能够减少引入到代码中的缺陷数量。开发过程中使用的度量方法包括缺陷数量和缺陷率（缺陷检测模型）。还有随着开发过程的改进，在每次发布结束时进行的“学习经验教训”的分析。

此产品的开发和测试过程包括以下几个主要阶段：

高层设计，设计文档和检查记录。

低层设计，设计文档和检查记录。

代码检查，或采用同伴检查方式。

功能验证测试（检查测试计划）

系统测试，充当产品的第一个用户（检查测试计划）

打包和发布测试（重复已有的测试用例）

解决方案测试（与其他产品的集成测试）

在前面的发布中，测试人员创建了大量的回归测试序列。这些序列被用于功能验证测试（FVT）和系统测试，以保证新版本的变化不会在现有的代码库中引发问题。此外，对每个逃逸的缺陷进行因果分析，是为了了解这些缺陷是如何逃逸的，以及如何防止未来

有类似的缺陷再次逃逸。尽管进行了这些大量的最好的实践，但还是需要更加地关注于改进。在基于客户使用的测试上，ODC 能够通过对最大几率区域的识别来提供这些改进。

部署　在做出对产品的最后版本实施 ODC 的决定之后，由分别来自于开发、功能确认测试、系统测试和解决方案测试的各个人员组成了一个七人团队，随后，他们学习了 ODC 的使用。这个团队每周都会聚在一起对客户发现的特殊缺陷进行分类。

在对区域缺陷进行分类时，首先基于实际的环境描述选择 trigger（触发器），揭示缺陷以及最有可能引发此缺陷的动作，看其是否能在内部发现。如果我们不知道用户是如何发现缺陷的，即用户的操作是不可知的，那么我们需要记录如何在内部重现此问题。如果在代码执行时出现了缺陷，则可以假定此缺陷是从测试阶段逃逸的。实际上，多数缺陷都是从测试阶段逃逸的。其中的一少部分同样也逃过了文档审核。

评估　在对 12 个月的 ODC 数据进行分类后，我们做了一次评估。由于这个产品的生命周期超过了 12 个月，因此我们需要谨慎对待主要的变更，尽管如此，这次评估还是突出了一些有趣的事实（细节）。

基础代码的重要性　复查的第一个 ODC 属性是 age（码龄）。用户发现的缺陷表明其出自于基代码、重写的、重新修改的代码，还是新代码中？图 1 为 age（码龄）vs.活动图。虽然开发团队的成员希望在新代码中发现大部分的缺陷，因为缺陷通常是由新代码或已更新的代码引入的，但令他们惊讶的是他们在基础和重写的代码中也发现了相当多的缺陷。有两个没有包含任何新功能的功能区域，由于存在着大量的重写（代码），因此其对应的缺陷被分类为“rewritten”（重写）。然而，基代码中的缺陷数量也看似很高的样子。下一步则确定这些基缺陷（基代码中的缺陷）是从哪个测试阶段逃逸的，以及包含了这些缺陷的功能模块是哪些。基缺陷对应的活动 vs.模块组合图（图 2）表明大部分逃逸到外部的缺陷来自于 FVT

（功能验证测试），并且有两个模块包含的缺陷数量超过了总数的百分之四十。

Figure 1　Age versus activity in uncovering defects in code

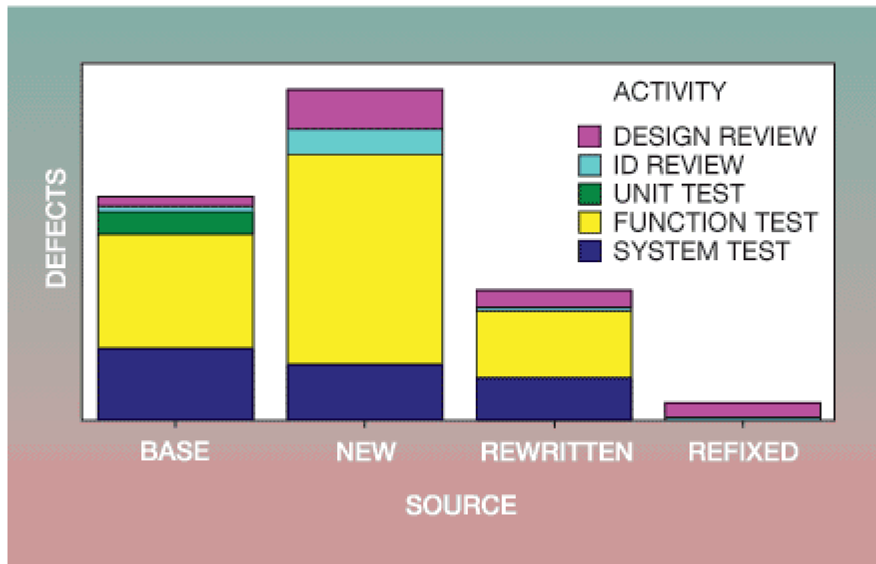
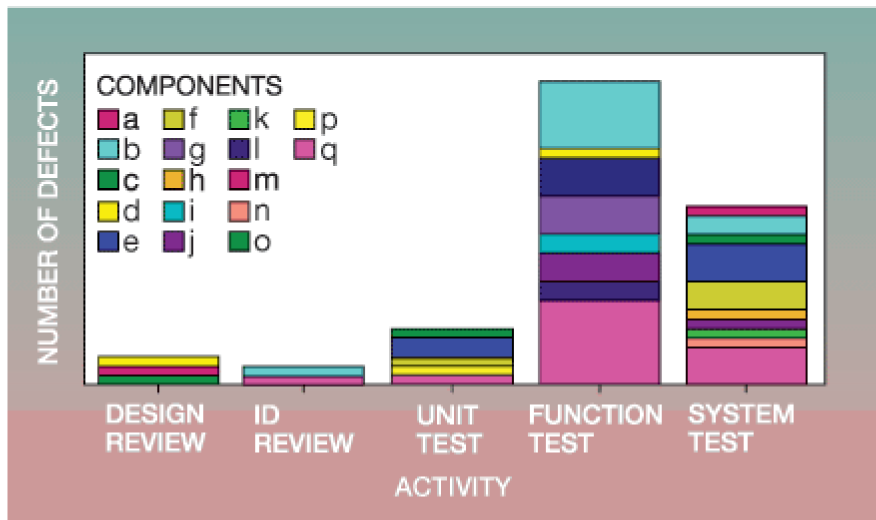


Figure 2　Frequency for component within activity (base defects only)



ODC 评估确定了需要进一步调查的区域。然后，利用因果分析技术对这些具体模块中的基缺陷做进一步的分析。这个评估显示：

q 模块需要更多的回归测试，特别是临近测试结束时。在下次发布期内执行此项测试。

b 模块中添加了大量的新代码，因此这些缺陷是在新代码运行某些方面的基代码时发现的，在此之前那些基代码并没有被使用。此区域中的缺陷还需要进一步研究，但是有一个明显的改进之处，对受新代码所影响的基础功能模块，要进行更多的功能测试。

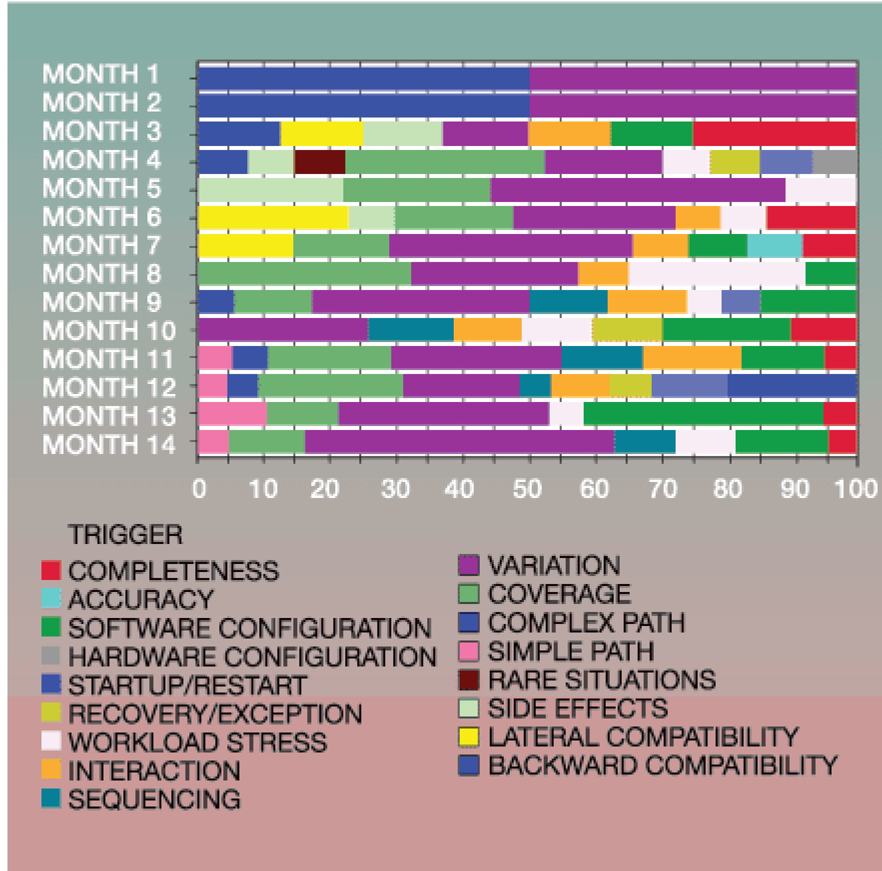
很明显，在全部的缺陷中，b 模块的缺陷数量最多。在项目期间这个功能区域经历了显著的变化。团队知道由于计划时间的约束，一些变更没有进行完全的文档说明，同时，他们也接受了缺乏文档所带来的风险。但是， ODC 评估表明了该决定所造成的影响。在未来的发布中，一定要实施文档的变更（管理）过程。

改进功能测试　理想的情况是，客户所发现的缺陷数量会随时间而减少。图 3，显示了随着时间的推移，外部缺陷的 trigger（触发器）分类的百分比视图，从中可以看到缺陷的数量实际上是在增加的。尽管我们可以预计到这些增加，是由于 licenses（用户的使用许可）的增加而引起的。但是比之观察缺陷总数趋势更为重要的是对用户使用方面的评估。这个评估会指出用户是如何发现这些缺陷的。

图 3 表明在最初的几个月中，主要的缺陷出现于执行某个单独的功能时，正如图中的 trigger（触发器）--coverage（覆盖）和 variation（变量）所指出的那样。随后，在 9 月里，通过更为复杂的测试场景所发现的缺陷在增多，图中的 trigger（触发器）--interaction（交互）、sequencing（连续）和 software configuration（软件配置）可以说明这一点。但是在大多数月份里 variation 是占主导地位的 trigger（触发器）。Variation 表示通过改变单个功能的输入参数所发现的缺陷。图 2--活动和模块图，表明很多缺陷是从 FVT（功能验证测试）中逃逸的，正如图中的 trigger-- coverage（覆盖）、variation（变量）、interaction（交互）和 sequencing（连续）所表现出来的那样。这些观测结果表明，我们必须对 FVT（功能验证测试）过程进行改进，以确保逃逸的缺陷少之又少。具体点说，测试计划中必须包括更多的变量（variation）测试。通过对测试人员的培训，以及检查测试计

划时对这一点的强调，我们可以实现上述的改进。

Figure 3 Trigger classification for field defects over time



小结　ODC 帮助我们在过程中找出下列不足和改进措施

多数缺陷是从 FVT（功能验证测试）逃逸的。Variation（变量）是最普遍的 trigger（触发器）。因此，团队需要对测试人员进行有关 FVT（功能验证测试）中各种测试类型的培训，确保 FVT（功能验证测试）的测试计划中有足够的变量（variation）测试用例。

q 模块中有太多的基缺陷。团队必须针对此模块区域增加回归测试的次数。

大多数缺陷都在 b 模块中。团队要保证所有的变更都具备清晰的文档说明，同时保证一旦有代码发生变化，受其影响的基代码应被作为测试中的重点对象来对待。

未来的计划　团队会继续对外部（发现的）缺陷进行分类，并利用此信息改进他们的开发和测试过程。ODC 外部缺陷为定义用户

惯用法的简档提供了基础，而简档可以帮助测试团队来复查缺陷。在新版本的开发周期中，这个团队也开始对“过程中”发现的缺陷进行分类，并与外部缺陷的分类进行了对比。这样的对比会为当前的测试过程提供更加准确的信息，并为将来的测试工作提供关注点。

Case Study 2: A large middleware project

案例 2：一个大型的中间件软件项目

这个案例中包含了一些隶属于 IBM 系列的中间件产品，这些产品可用于大部分的操作系统平台。它们都是由几百人的团队开发的成果，开发过程遵循典型的瀑布过程，通过各个阶段的判定检查点评估产品是否可以进入到下一个阶段。

产品团队中最先开始使用 ODC 的是测试组。而测试组正处于重组的过程中，以期通过此来解决一些问题。他们需要一套度量规则，使他们可以客观的评估他们的进度，并面对一些他们自己的问题，这些问题是：

单元测试阶段，功能验证（FV）阶段和系统测试阶段，三个阶段互相叠加进行。理论上讲，这些阶段都是假定为有序的阶段，但是它们会经常叠加，有时是完全地叠加进行。系统测试的入口检查点的目的是判断前一个测试阶段的进度，测试是否充分，是否可以进入到系统测试的阶段中。而他们需要的度量规则能够为这一判断提供可度量的、客观的数据，而不是个人的（主观）意见。

测试组开发了大量的测试序列，但是他们并不知道这些序列的有效性如何。他们只有两个度量方法：（1）缺陷--上升和关闭率 （2）测试用例--执行率和测试用例结果，如成功或失败。

这些度量方法度量了测试进度，却不能说明测试的有效性如何。而 ODC 可以提供客观的、可量化的数据，对组织寻找的判断依据而言是非常必要的。

ODC 过程　分别由两个项目组对 ODC 过程展开实验，并以一系列的培训讲座为开始。团队成员参加的培训内容包括 1 小时的概貌介绍，以及 2 小时的关于如何分类缺陷的详细培训。其中的一部

分测试人员和开发人员还要参加一个 2 小时的关于如何确认数据的课程。一个月后，很多人都接受了 2 小时的评估培训，学习如何复查数据，以及如何开始使用这些数据。

数据由确认小组每周进行一次确认，以保证分类的正确性，并纠正每个错误。随后，由评估小组对数据进行复审，并评估项目的状态，决定是否需要采取某些特定的行动。对于个别项目，可以选择合适的时机进行评估。

评估　在评估期间，他们决定是否可以开始系统测试，以及需要对系统测试和功能测试做哪些改进。

提早进入系统测试的一个案例.　在这个产品的 FV（功能验证）阶段进行到大约四分之三时，项目组开始使用 ODC。七周内，项目到达了系统测试入口的判断检查点处。虽然 ODC 数据在那个时候还没有被正式地纳入到决策的过程范围内，但项目组还是饶有兴趣地复查了这些数据。为项目组启动系统测试提供的支持，证明这些数据是非常有用的。

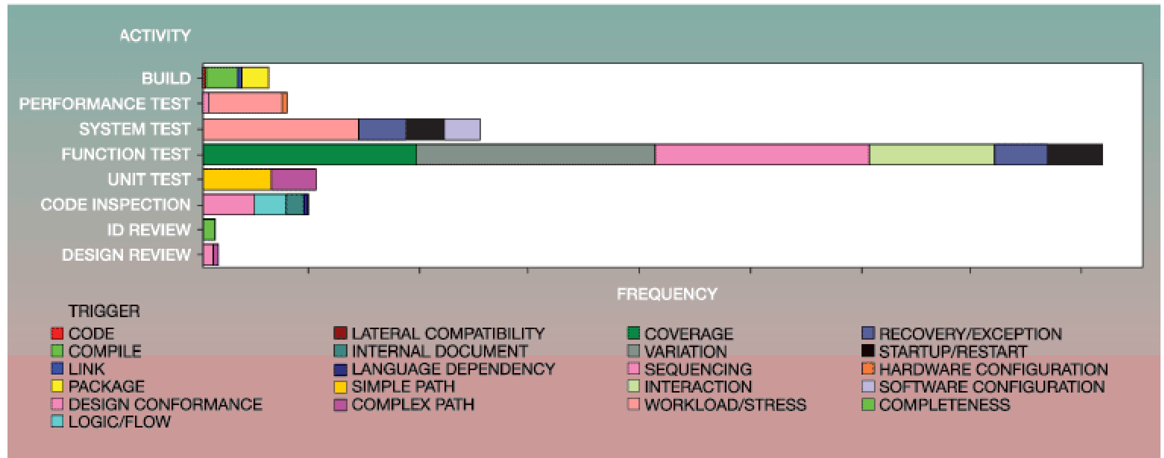
进入系统测试的（判断）标准之一是“FV（功能验证）的覆盖度超过百分之八十，而且完成了不少于百分之五十的功能验证”。在这个案例中，FV 并没有达到百分之八十的覆盖度目标，但是有超过百分之七十的功能验证都完成了–非常高的通过率。此外，FV 团队相信代码已经非常稳定了。图 4 则展现了 activities（活动）和 triggers（触发器），说明：

功能测试的范围已经很大了，达到了 ODC triggers（触发器）的范围。Coverage（覆盖）、variation（变量）、sequence（顺序）和 interaction（交互）测试包含了所有已暴露的缺陷。实际上，在使用更复杂的系统测试的 triggers（触发器）时，如 recovery/exception（恢复/异常）和 startup/restart（开始/重起），项目组已经让缺陷完全地暴露了。

功能测试进入到更为复杂的 triggers（触发器），意味着产品的基本功能已经稳固，准备进入系统测试阶段。这个暗示为 FV 团队的

“直觉”提供了支持。

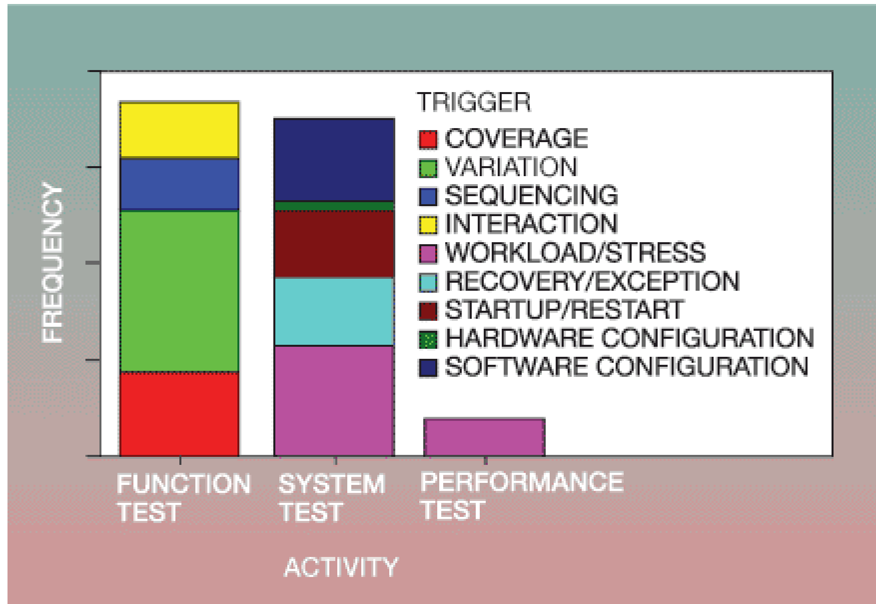
Figure 4 Each activity uses a broad range of triggers to expose defects



尽管没有达到 FV 的正式标准，但 ODC 提供的信息让团队相信项目已可以开始系统测试了。随后，系统测试团队取得了快速地进展。

分析外部缺陷，改进系统测试. 图 4 也说明系统测试的大部分关注点都在 workload/stress（工作量/压力）上。有超过百分之五十的缺陷是通过 workload/stress（工作量/压力）测试发现的。图 5 展现了对前一版本的外部缺陷进行后续分析所产生的结果。图中的 activities 和 triggers 清晰的显示了用户暴露的缺陷较之内部测试所发现的，其对应的系统 triggers（触发器）的范围更为广泛，特别是在 software configuration（软件配置）上表现的尤为明显。因此，系统测试团队会复查他们的测试，以扩充他们的测试场景，特别是 software configuration（软件配置）的区域。

Figure 5 Frequency of activities with triggers



通过对外部和内部缺陷的分析，改进功能测试. 随着 FVT（功能验证测试）的完成，测试团队复查了 ODC 数据以寻找潜在的改进区域。图 6、7 和 8 突出了在未检查的情况下可能出现的潜在问题。

Figure 6 Frequency of defect type with qualifiers in previous release

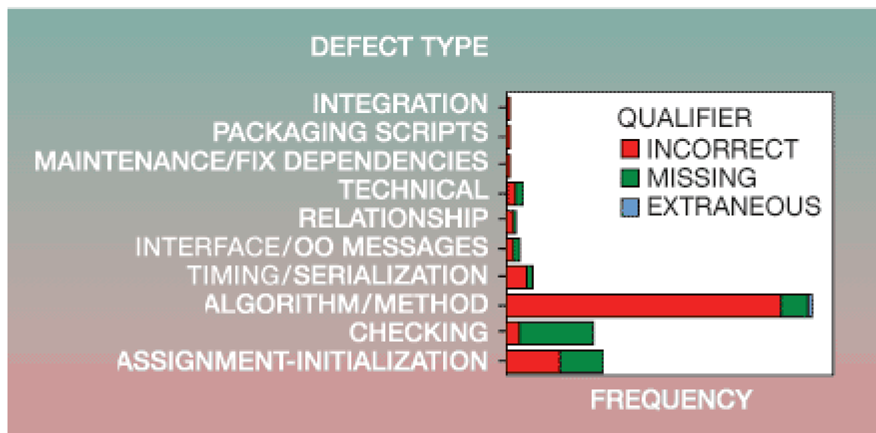


Figure 7 Frequency of defect type with qualifiers for current release

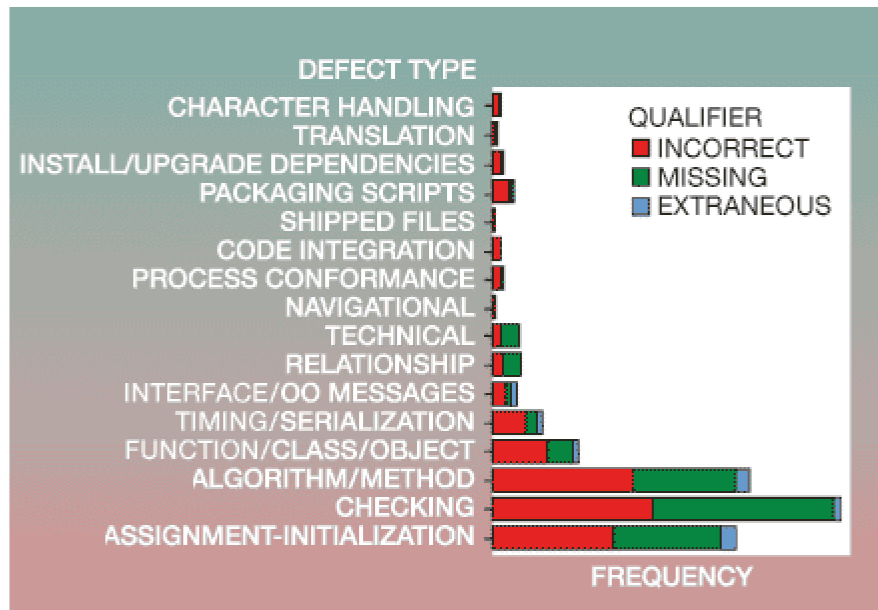


Figure 8 Defect frequency for trigger within defect type

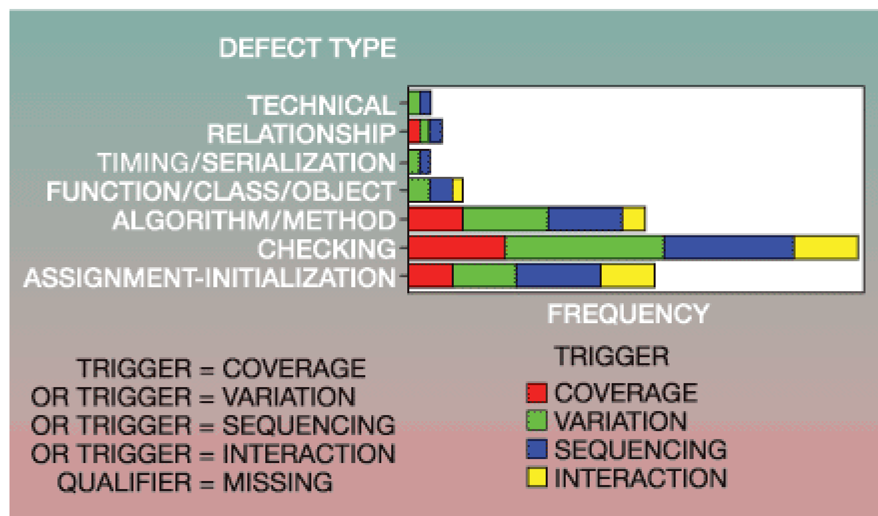


图 6 展现了产品前一个版本的外部缺陷对应的 defect type（缺陷类型）和 qualifier（限定）。虽然 checking 类的缺陷并不是最大的一组，但其中有相当大的一部分是由 missing（缺失）的代码引起的。图 7 展现了一个相同的表，但显示的是当前版本在内部发现的缺陷。这次，checking（检查）类的缺陷是最大的一组，而且其中的大部分同样是由于代码 missing（缺失）引起的。图 8 则展现了 qualifier（限

定）均为 missing 的内部缺陷，以及它们的 defect type（缺陷类型）和 triggers（触发器），同时表明大部分因 missing（缺失）检查造成的缺陷是由 variation（变量）和 sequencing（顺序）测试发现的。这说明低层设计的稳定性存在问题。

根据这次评估的结果，FV（功能验证）团队的成员们决定以提高发现“缺失检查”的能力为目标，重新复查他们的测试序列，特别是：

使用代码覆盖工具度量由功能测试序列获得的语句和分支覆盖率。测试人员在开发人员的帮助下利用输出的结果，来确定测试序列中的缺口和重复。

ODC 的 triggers（触发器）是良好覆盖率的保证，因此，可以根据它们对功能测试用例进行分类和分析。为了向 2000 年的悉尼奥运会提供最好的产品，IBM 团队使用了这种技术作为一种确定方式，事先确定测试序列能否使用大范围的 triggers（触发器）。获得此产品 FV（功能验证）的测试序列的有效性评估是至关重要的，特别是随着时间的发展，测试用例的数量会明显地增加。

由评估引出的改进. 几项致力于改进过程的措施都来自于 ODC 评估结果。在一个典型的发布周期内，仅仅在最后的几周或几个月里，才尝试对缺陷的修复进行优先级划分，往往会导致测试工作长期停滞不前，并超出实际需要的时间。图表 9 里面的数据指出，很多缺陷的严重等级都是 3 级，而通常会认为它们已足够严重，需要修复，但实际上它们还没有严重到让用户的系统崩溃的地步。尽管这些缺陷占了已提交缺陷总数的百分之八十，但是不可能把它们放在优先的位置。因此，我们在在缺陷描述中添加了一个新的属性“importance”（重要性）。基于对有多少工作会受此缺陷影响的考虑，提交人可以指定缺陷的重要性，以使这个缺陷能够得到快速的修复。“重要性”有：

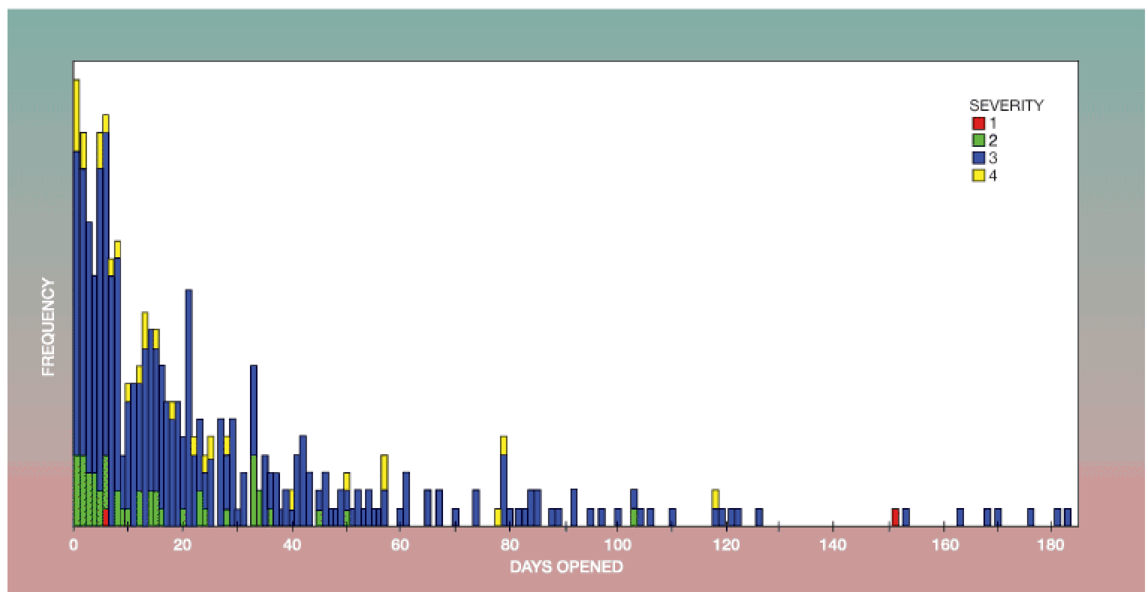
1. 高重要性：阻碍了大量的测试工作，需要立即修复的—紧急。
2. 中等重要：阻碍了一部分测试工作，但同时还可以进行其他

的工作，需要尽可能快的修复。

3. 低重要性：最多阻碍了一到两个测试工作，可以继续进行工作，有时间的时候修改。

这个属性字段能让团队把重点放在对整个进程影响最大的缺陷上。

Figure 9 Severity of defects



另一处由 ODC 评估结果引出的改进是让缺陷包含更多的信息，以便于对 ODC 分类的缺陷进行确认。这处改进对分类缺陷的团队也同样有用。提供更清晰的解释和说明，能够有效地对缺陷进行处理。

小结　这个项目使用 ODC 的时间刚好超过了一年。事实证明它是有价值的，当未达到传统的标准时，它能够帮助我们评估通过一个判断检查点时所产生的风险。

这个团队能够客观地查看他们的数据，而且很快就能确定出针对他们的过程和最终产品的改进措施。这些改进是合理的，不需要投入大量的时间、金钱或人力。通过对 ODC 的使用，测试团队达到了他们提高测试效率的目标。这项技术的价值在这个团队得到了证明，因此， ODC 的使用范围也扩大到了其他的项目上。

Case Study 3: A small team project

案例 3：一个小型团队项目

这个案例属于一个小型项目，由几个开发人员开发一个基于Java**的软件工具，其用途是让用户把与软件开发相关的数据进行可视化。这个工具支持 ODC 分析。这个产品的基本用户是整个 IBM 软件开发试验室中的测试人员、开发人员和服务人员。这个开发团队面临的挑战是：

缺少开发资源且经验有限 -- 这个团队缺乏面向对象编程和 Java 语言的开发经验，也没有工具开发方面的经验。此外，他们只有一点测试经验。团队成员们依靠一个功能检查表和他们自己的感觉来制定测试的内容。

时间进度上的压力 -- 他们经常要修复 bug 和增加功能，导致进度表越来越紧张。

客户需求 -- 用户要求这个工具必须能够大规模且成功地部署 ODC。因此，为了最大限度地成功实现这项功能，它必须是一个高质量且稳定的产品。

ODC 是有效管理资源、度量产品稳定性，以及指导团队进行高效测试的关键。

ODC 过程　自产品的第一个版本发布以来，ODC 已经得到了实践，但本文的关注点是产品在 1.3 版本中所做出的改进。到此版本可用时，团队已经有了 ODC 分类的经验。由于项目经理曾在过去的五年里做过 ODC 的顾问，在执行确认和评估方面有着丰富的经验，因此在执行评估的时候没有遇到“学习曲线”的问题。

评估　最初，他们计划在 2000 年的八月至九月间发布 1.3 版本。此版本的发布主要是解决一个软件 bug，并没有增加新的功能。在计划发布的一个月前，团队成员们表达了他们的信心，产品已经为发布做好了准备。他们相信这次只是因为要修改 bug，基本的功能都不会发生变化。他们已经花了几个星期的时间测试了新的代码，所以他们觉得时间进度不会在发生意外的变故了，产品能够按照原定的计划发布。然而，在他们进行了 ODC 评估后，却证明产品并没有做好发布的准备。

图 10 展现了相关的 activity（活动）和 triggers（触发器）。需要注意的第一点是只有 15 个缺陷。即使对于这个小型项目而言，功能和系统测试仍然会发现更多的缺陷。第二点，在功能测试中发现的缺陷数量和在用户界面检查中发现的缺陷数量一样多。但在产品的历史记录中，功能测试中发现的缺陷数量明显比 GUI 检查中的多很多。第三点，由 coverage（覆盖）和 variation（变量）可以证明，在功能测试中暴露的缺陷只是通过执行单个的功能发现的。通过点击按钮或执行简单的命令就可以很容易的发现这些缺陷。然而，在通常情况下，当每个活动上的缺陷都是通过大范围的 triggers（触发器）发现的时，产品才是经过良好测试的产品。虽然在界面检查时发现的缺陷是通过各种不同的 triggers（触发器）发现的，但功能和系统测试则没有。Coverage（覆盖）和 variation（变量）只代表了此活动上的 4 个可用 triggers（触发器）中的 2 个而已。在产品发布之前，还需要通过更复杂的 triggers（触发器）进行测试。特别是，通过功能测试中的 sequencing 测试和系统测试中的 workload/stress、software configuration 测试发现更多的缺陷。

Figure 10 Frequency versus activity with triggers

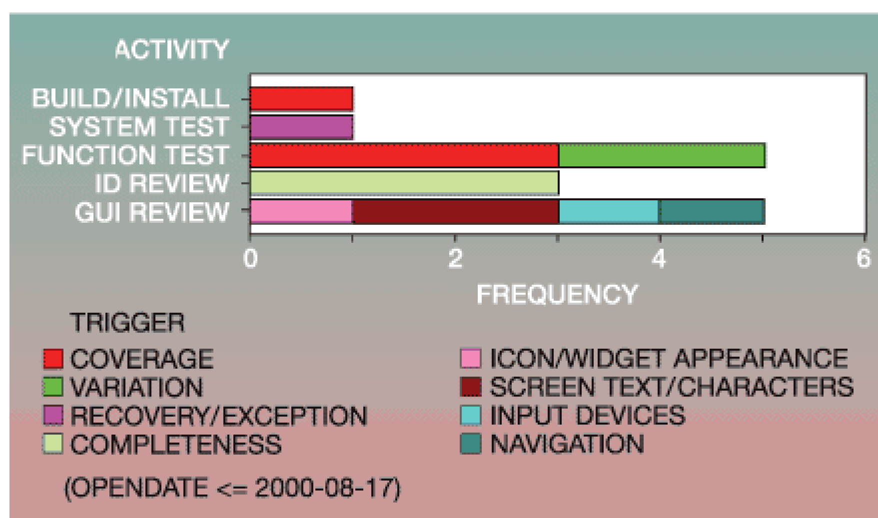
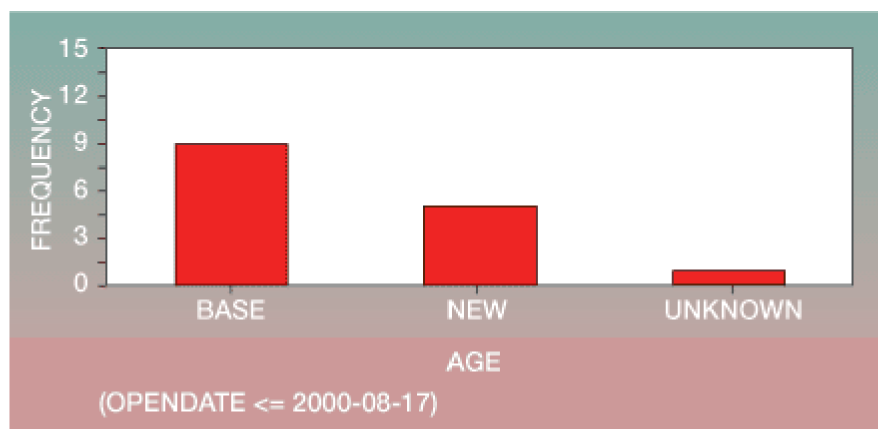


图 11 展现的是 ODC 属性为 age（码龄）的缺陷数量，进一步支持了产品还没有准备好发布的结论。其中有 9 个缺陷（占总数的 60%）

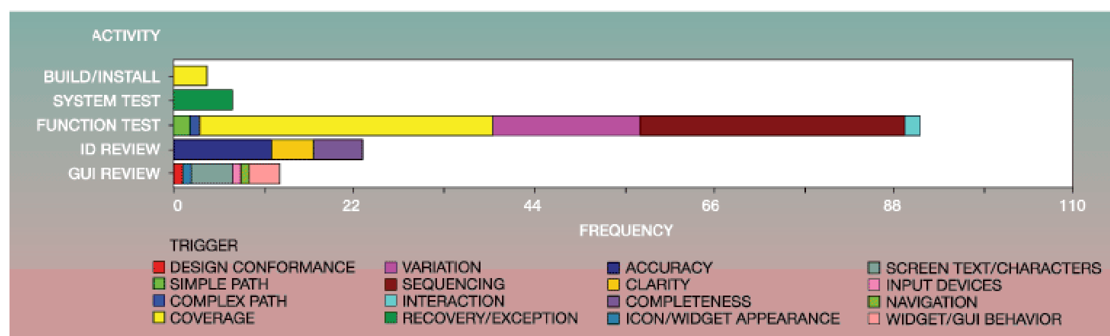
是在基础代码中发现的 – 缺陷在之前的版本中就已经存在了。这些是被隐藏的缺陷。本应该在之前的版本中发现却没有发现。因此，不仅通过执行单个的功能发现了缺陷，而且还有很大一部分的缺陷是在老代码（基代码）中发现的，并非在新代码中。此信息更进一步地支持了团队关于对代码没有进行充分测试的怀疑。所以依照情况，团队采取了唯一可能接受的措施。增加测试工作，推迟发布日期。

Figure 11 Number of defects by age



为了提高测试效率，在 variation、sequencing、interaction、software configuration 和 workload/stress 区域里，添加了几个测试用例。图 12 展现了新增测试用例的测试结果，以及对应的 ODC 活动。

Figure 12 ODC activity by triggers



到 12 月时，除了之前的 15 个缺陷，他们又发现了 138 个缺陷。新增的测试工作得到了回报。接下来，对 triggers（触发器）图表进行复查。在此之前，单个的功能 triggers（触发器）构成了整个测试的全部（百分之百）。虽然计划的时间被延长了，但由多功能 triggers

（触发器）-- sequencing 和 interaction 暴露的缺陷增长到了总数的37%。此外，我们通过系统测试的 triggers（触发器）—recovery/exception（恢复/异常），发现了 7 个缺陷。测试场景由简单场景发展为更复杂的场景，从而发现了更多的缺陷。

在 workload/stress 和 software configuration 上还需要说明两点。由于测试执行方面的限制，没有发现新的缺陷。同时，由于缺少资源，这个结果被认为是可以接受的。

小结　在已延长的测试周期临近结束时，产品比 8 月份表现出了更强的稳定性。大部分的缺陷都已被发现，当然可以预计得到还会有很少量的缺陷逃逸到外部。事实上，它们还是被这个团队发现了。1 月份报告了 4 个缺陷，其中的一个是 build（构建）问题。然而，如果在 8 月份他们并没有做 ODC 评估，产品照原计划发布，那么九月和十月的测试所发现的缺陷很显然会对用户产生影响。用户满意度势必会降低，对团队开发高质量产品的能力失去信心。此外，开发团队还会变成“救火员”的角色。相反，这个团队利用了 ODC 的分类缺陷数据，评估了测试的有效性。在随后创建的测试计划中明确了目标，在之前缺少的 sequencing 测试中使用多功能场景。这些步骤都可能提高测试效率，同时降低弱点，让开发团队向用户交付一个稳定的、高质量的产品。

正如前面所提到的一个重要的方面，ODC 提供的是基于数据的判断的支持。这个评估花去了一个小时的时间，随后决定扩大测试的范围，而这个决定是在客观的数据基础上做出的。一旦团队复查了图表中列出的 ODC 属性，有关产品是否准备就绪的争议就不会再有了。团队有了下一步的决定，有了要采取的应对措施，经理就不会再被两个对立的观点搅得痛苦不堪了。这些图表清晰地表现出了软件的状态，以及采取的措施，对团队来说，它们都是显而易见的。

这次的经历使开发团队相信，在测试期间不用花太多的时间和精力去精确地度量过程和强调风险。尽管是个小型团队，但对分类缺陷的数据评估可以很快就完成了，并采取相应的措施以避免潜在

的灾难性的情况发生。最重要的是，从用户那里收集到的反馈表明这个版本是最稳定的版本！

Conclusion

总结

我们描述了三个不同产品的项目团队使用 ODC 的经历。通过使用 ODC，每个团队都达到了提高测试效率的目标，同时对组织资源造成的影响也是非常小的。在所有三个项目中，他们在采用 ODC 之前已经收集并使用了缺陷相关的信息，而 ODC 能够让这些团队利用丰富的缺陷信息集，以一种客观的方式，来改进各自的产品质量。

第一个案例是一个高质量的产品，其目的是减少客户报告的缺陷，以此提高客户的满意度，同时降低质保的费用。通过对这些引起客户报告的潜在问题的确定，测试的有效性得到了提高。来自于逃逸缺陷的分类 triggers（触发器），为测试和开发的某个具体区域的改进提供了必要的依据。这个团队所采取的措施其重点是加强测试计划，改进 FVT 培训，增加对已确定模块的回归测试，以及提高文档的质量。

第二个案例使用了在功能和系统测试数据中发现的缺陷来提高测试的有效性。这个研究列举了一个减轻进度压力的例子，通过 ODC 的分类缺陷，表明产品的进度充分，足以进入到下一步测试中。团队能够早于计划的时间开始系统测试，得益于 ODC 评估提供的信息，它表明团队已经达到了早期活动的退出标准。与第一个案例相似，这个团队也确定了一些薄弱的测试区域。随后，他们针对这些区域采取了具体的措施。他们实施的措施包括使用代码覆盖工具和功能测试用例的 ODC 分类，来确保他们所需的测试场景有足够的覆盖范围。

第三个案例描述了一个拥有很少资源的小团队，他们对测试存在的具体不足进行确定，并从中受益。这个团队使用的缺陷信息是在项目发布前不久在内部发现的。产品发布前，他们针对需要改进的功能测试，使用了 ODC 来确定多功能的测试场景。这样，团队能

够快速地评估测试的效果并实施正确的措施来加强他们的目标 triggers（触发器）。

ODC 数据收集可以在软件开发过程中的任意点开始。所有的三个案例都提供了这样的例子，他们分别在软件生命周期中不同的时间点上对 ODC 分类的缺陷进行分析，并从中获益。而他们实施的改进措施都是合理且可行的，也不需要为实现提高测试有效性的目标投入大量的资源。

除了此处所表现的 ODC 的积极影响外，还有很多不容易度量的结果，但尽管如此，我们还是从中得到了很多的收益。我们的测试人员和开发人员身上产生了微妙的变化，比如他们的技能都提高了，因为现在他们能看到自己的工作是以一种新的客观的方式进行的。测试人员学会了用复杂的 triggers（触发器）来考虑测试用例。他们会根据预定计划的阶段查看当前进度的趋势，也更加警觉于这些趋势里出现的任何偏差。他们变成了缺陷 triggers（触发器）的比较能手，而这些 triggers 分别暴露了用户发现的和他们自己的测试所发现的缺陷。测试人员增加了测试有效性方面的知识水平，从而带来了更好的产品质量和更强壮的过程。最终，这些增长的知识生产出了质量更高的且开发费用更少的软件 -- 一个所有的软件组织想努力达到的目标。

Appendix A: Scheme version 5.11 for design and code

附录 A：针对设计和编码活动的分类表　版本 5.11

更多的信息和例子请参阅 http://www.research.ibm.com/softeng。

提交缺陷时的分类属性：

Activity（活动）—指实际（软件开发过程）的活动。例如，在系统测试阶段，当你点击一个“选择打印机”的按钮时，出现了一个缺陷。虽然处于系统测试阶段，但对应的活动应为功能测试，这是由于此缺陷是通过执行一个“功能测试类型”的活动发现的。

Triggers（触发器）—触发缺陷出现的环境或条件。它的定义更类似于一种“测试类型”。

Impact（影响）—逃逸到外部的缺陷对用户产生的影响或开发过程中未发现此缺陷的影响。

表 1 为活动到触发器的映射表。

Table 1　Attributes identified when a defect is uncovered

Activity and Triggers				Customer Impact
Inspection	Unit Test	Function Test	System Test	
Design conformance	Simple path	Coverage	Workload/stress	Installability
Logic/data flow	Complex path	Variation	Startup/restart	Serviceability
Lateral compatibility		Sequencing	Recovery/exception	Standards
Backward compatibility		Interaction	Hardware configuration	Integrity/security
Language dependency			Software configuration	Migration
Concurrency			Blocked test (formerly Normal Mode)	Reliability
Internal document				Performance

Side effects	Documentation
Rare situations	Requirements
	Maintenance
	Usability
	Accessibility
	Capability

修改缺陷时可用的属性：

Target（目标）—在哪里修改缺陷，如设计、代码、文档等。

Defect type（缺陷类型）—实际的修改类型。

Defect qualifier（缺陷限定：适用于缺陷类型）—捕捉的不存在、错误或无关的实施元素。

Source（来源）—缺陷的起源，一般如内部开发的代码、重用的代码、外包的代码或端口等。

Age（码龄）—缺陷对应的代码历史。

The attributes identified when a design or code defect is fixed are shown in Table 2.

Table 2 Attributes identified when a design or code defect is fixed

Target	Defect Type	Qualifier	Source	Age

Design /code	Assignment/initialization	Missing	Developed in house	Base
	Checking	Incorrect	Reused from library	New
	Algorithm/method	Extraneous	Outsourced	Rewritten
	Function/class/object		Ported	Refixed
	Timing/serialization			
	Interface/OO messages			
	Relationship			

In this paper, we have focused on defects found in design and code and so have not included the target of information development, build/packaging, National Language Support, and their values.

Appendix B: Triggers for graphical user interface review activity

附录 B：

Table 3 shows the list of triggers that can expose defects while reviewing a graphical user interface consisting of graphical elements such as buttons, labels, and scrollbars. Please see http://www.research.ibm.com/softeng/ for more information.

Table 3 List of triggers in GUI review

Triggers

1. Design conformance
2. Icon/widget appearance
3. Screen text/characters
4. Input devices
6. Navigation
7. Widget/GUI behavior

Appendix C: Some typical assessment topics and associated defect attributes

Table 4 is a list of typical topics of interest to a software development organization and the defect attributes that are useful in addressing them. An assessment would usually consist of looking at the defect distribution across these attributes as well as evaluating the relationships of the attributes to each other.

Table 4 Typical topics and defect attributes

Topic	ODC Attributes	Non-ODC Attributes
Measuring test effectiveness	Activity, trigger, qualifier	Open date, severity, component, phase
Evaluating product stability	Defect type, impact, qualifier, source, age	Open date, severity, close date, component

Identifying strengths and weaknesses in design and code	Type, qualifier	Component
Evaluating customer usage	Impact, trigger, defect type, qualifier	Open date, severity, component
Measuring progress	Activity, trigger	Severity, component, phase

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

Accepted for publication August 30, 2001.

Glossary

Verification & Validation

表 1：两者的区别

Validation	Verification
Am I building the right product	Am I building the product right
Determining if the system complies with the requirements and performs functions for which it is intended and meets the organization's goals and user needs. It is traditional and is performed at the end of the project.	The review of interim work steps and interim deliverables during a project to ensure they are acceptable. To determine if the system is consistent, adheres to standards, uses reliable techniques and prudent practices, and performs the selected functions in the correct

	manner.
Am I accessing the right data (in terms of the data required to satisfy the requirement)	Am I accessing the data right (in the right place; in the right way).
High level activity	Low level activity
Performed after a work product is produced against established criteria ensuring that the product integrates correctly into the environment	Performed during development on key artifacts, like walkthroughs, reviews and inspections, mentor feedback, training, checklists and standards
Determination of correctness of the final software product by a development project with respect to the user needs and requirements	Demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle.

在软件工程中，验证(Verification)和确认(Validation)的区别：

“确认”是要证明所提供的（或将要提供的）产品适合其预计的用途，而“验证”则是要查明工作产品是否恰当地反映了规定的要求。换句话说，验证要保证“做得正确”，而确认则要保证“做的东西正确”。

验证注重“过程”，确认注重“结果”

简单一点说：

验证我们是不是正确的做了软件（实现和需求规格一致）

确认我们是不是做了正确的软件（实现和用户需求一致）

Review & Inspection

复查（审核）&检查

复查（审核）—指到达软件生命周期的各个阶段，所进行的确认活动，主要以会议的形式对各个阶段的产物进行评审。一般会组成评审委员会，组织对各种产物的评审（复查）活动。

检查—指对产品或产物进行一步一步的详细检查。目的是发现被检查物中的错误。实施检查的组织由开发、测试和质量保证等同级人员组成。

自动化脚本编写方法

作者：Ranjit Shewale　译者：陈能技

摘要

这篇文章详细描述几种自动化脚本编写方法，各自的优、缺点，同时在脚本编写的成本、编程技巧和脚本可维护性方面作出相应的评价。

声明

作者在对这几种自动化脚本编写方法作出关于成本的评价时，没有参考任何自动化测试项目的成本分析文档或成本效益分析结果。建议读者基于自己的理解和考虑风险来消化利用这些信息。

文章的编排

这篇文章主要分析自动化的成本，然后在描述每一种脚本编写方法时指出它的优点和缺点。

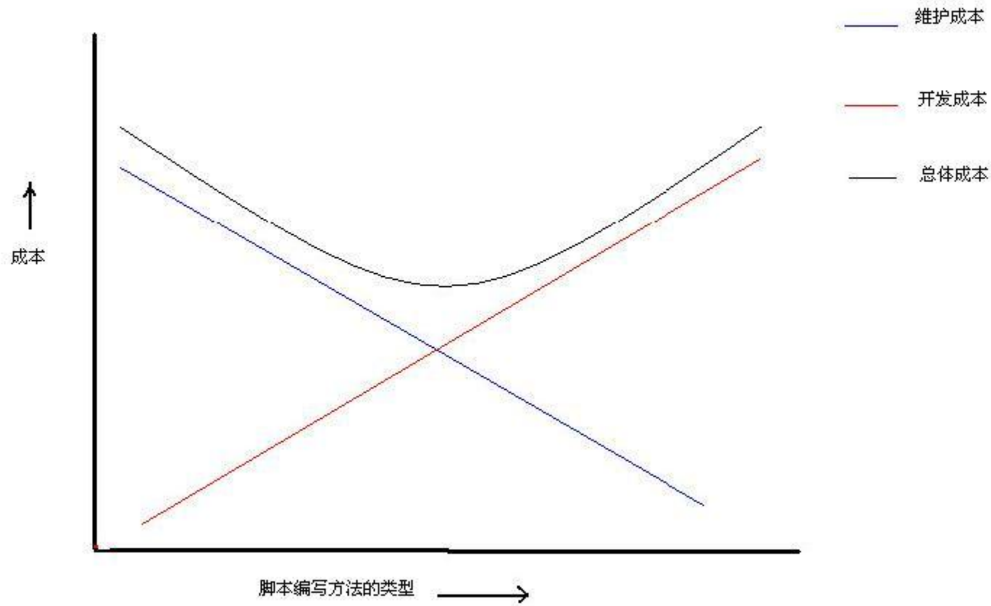
关于优缺点，从以下方面进行评价：

- 脚本的编写是否是结构化的
- 测试用例在什么地方定义
- 开发脚本的成本
- 需要怎样的编程技能
- 如何计划、设计和管理脚本
- 测试数据放在什么地方
- 脚本如何维护，维护成本
- 其它方面

成本

测试脚本相关的成本主要由开发成本和维护成本组成。在自动

化测试过程中使用不同的脚本编写方法会对成本有不同程度的影响。



“录制回放”的方法是简单的，也是脆弱的，但是它的开发成本很低，然而维护成本很高，因此总体成本也会很高。使用先进的关键字驱动测试的方法，则维护成本会很低，但是开发成本会很高，因此总体成本也会很高。测试经理需要在这些方法中作出明智的选择，以便把总体成本尽量降低。

编写脚本的方法

不同的自动化测试脚本编写方法主要有：

1. 线性的
2. 结构化的
3. 共享的
4. 数据驱动的
5. 关键字驱动的

线性脚本编写方法

线性脚本编写方法是使用简单的录制回放的方法，测试工程师使用这种方法来自动化地测试系统的流程或某些系统测试用例。它可能包含某些多余的、有时候并不需要的函数脚本。

优缺点：

1. 是一种非结构化的编程方式
2. 测试用例由脚本定义
3. 非常低的开发成本
4. 测试人员所需要的编程方面的技巧几乎可以忽略
5. 不需要计划、设计
6. 测试数据在脚本中是硬编码的
7. 脚本会很脆弱，因此维护成本会很高
8. 没有公用的脚本，因此可能造成重复劳动

结构化脚本编写方法

结构化脚本编写方法在脚本中使用结构控制。结构控制让测试员可以控制测试脚本或测试用例的流程。在脚本中，典型的结构控制是使用“if-else”，“switch”，“for”，“while”等条件状态语句来帮助实现判定、实现某些循环任务、调用其它覆盖普遍功能的函数

优缺点：

1. 是结构化的脚本编写方法
2. 测试用例在脚本中定义
3. 编程的成本要比线性脚本编写方法略为高一点
4. 需要测试员的调整编码技巧
5. 需要某种程度上的计划、设计
6. 测试数据也是在脚本中被硬编码
7. 因为相对稳定一点，所以需要相对少的脚本维护，维护成本

比线性脚本编写方法的要相对低

8. 除了编程知识外，还需要一些脚本语言的知识

共享脚本编写方法

共享脚本编写方法是把代表应用程序行为的脚本在其它脚本之间共享。意味着把被测应用程序的公共的、普遍的功能的测试脚本独立出来，其它脚本对其进行调用。这使得某些脚本按照普遍功能划分来标准化、组件化。这种脚本甚至也可以使用在被测系统之外的其它软件应用系统。

优缺点：

1. 脚本是结构化的
2. 测试用例在脚本中定义
3. 开发成本相对于结构化脚本编写方法来说要降低一些，因为减少了很多复制的劳动
4. 需要测试员的调整代码的编程技巧
5. 由于脚本需要模块化，所以需要更多的计划和设计
6. 测试数据也是硬编码的
7. 脚本维护和维护成本要比线性脚本编写方法的相对低

数据驱动脚本编写方法

这种方法把数据从脚本分离出去，存储在外部的文件中。这样脚本就只是包含编程代码了。这在测试运行时要改变数据的情况下是需要的。这样脚本在测试数据改变时也不需要修改代码。有时候，测试的期待结果值也可以跟测试输入数据一起存储在数据文件中。

优缺点：

1. 脚本是以结构化的方式编程的
2. 测试用例由测试数据或脚本定义

3. 由于脚本参数化和编程成本，这种方法的开发成本跟共享脚本编写方法比较要相对高
4. 需要测试员较高的代码调整方面的编程技巧
5. 需要更多的计划和设计
6. 数据独立存储在数据表或外部文件
7. 脚本维护成本较低
8. 推荐在需要测试正反数据的时候使用

关键字驱动脚本编写方法

这种方法把检查点和执行操作的控制都维护在外部数据文件。因此测试数据和测试的操作序列控制都是在外部文件中设计好的，除了常规的脚本外，还需要额外的库来翻译数据。是数据驱动测试方法的扩展。

优缺点：

1. 综合了数据驱动脚本编写方法、共享脚本编写方法、结构化脚本编写方法
2. 测试用例由数据定义
3. 开发成本高，因为需要更多的测试计划和设计、开发方面的投入
4. 要求测试人员有很强的编程能力
5. 最初的计划和设计、管理成本会比较高
6. 数据在外部文件存储
7. 维护成本比较低
8. 需要额外的框架或库，因此测试员需要更多的编程技巧

评价

- 关于开发的成本

随着脚本编写方法从线性到关键字驱动的改变，开发的成本不

断地增加。

- 关于维护的成本

随着脚本编写方法从线性到关键字驱动的改变，维护的成本在降低。

- 关于编程技能要求

随着脚本编写方法从线性到关键字驱动的改变，对一个测试员的编程熟练程度的要求在增加。

- 关于设计和管理的需要

随着脚本编写方法从线性到关键字驱动的改变，设计和管理自动化测试项目的要求在增加。

参考

Software Test Automation：Effective use of test execution tools （作者：Dorothy Graham、Mark Fewster）

基于缺陷分布的质量目标分解和质量预测体系

作者：罗耀秋

摘要：质量管理如何去量化要求？怎么设定一个合理的质量目标？质量目标如何被分解？质量目标如何随着项目实施过程推进而被修正？这些都是困扰项目质量管理的老大难问题。本文尝试从缺陷分布模型的角度去提供一种思考维度。

关键词：质量管理，量化，质量目标，质量预测。

项目量化管理与量化质量目标

CMU SEI 的 CMMI 模型中，把一个软件企业的软件能力成熟度分成五个等级。分别为初始级，可重复级，已定义级，已管理级和优化级。各成熟度等级的特征如下：

初始级—软件过程是无序，无章可循的，软件项目的成功依赖项目组中的关键成员的个人能力，项目的成功是偶然和不可预见的。

可重复级—项目已经定义了最基本的项目管理过程，对项目的进度，成本和质量在一定程度上起到控制作用，对同类型的项目，一些成功的经验是可以被复用和优化到新的项目过程中去的。

已定义级—软件项目管理过程已经上升为组织级别的标准过程规范。组织的项目过程采用或裁剪自组织标准过程。

已管理级—软件过程表现逐步稳定，软件过程和产品质量都能有量化的衡量准则，可以量化控制和预测过程和产品质量。

优化级—在已管理级的基础上，通过对过程革新，来不断的优化过程，从而达到持续改进。

那么，对于达到或者将要达到已管理级的这些公司，如何利用这些过程数据来控制和预测产品质量呢？基于规模数据和缺陷数据

是很多公司比较早收集的，而且相关过程也是比较早达到稳定的，我们可以先从缺陷数据开始入手，建立初步的质量目标分解和质量预测体系。

构建缺陷分布模型和项目缺陷密度性能基线

对于达到成熟度等级四级的公司来说，基于这个成熟度等级的项目一般来说项目管理、评审、测试等子过程的过程性能比较稳定，因此也有条件生成该过程的过程性能基线。通俗的讲，我们可以把一个公司某个过程的过程性能基线看作是该公司该过程的基准值。如，测试缺陷密度性能基线为[25±2] 个/千行，那么类似的项目的缺陷表现就可以参照这个数据了。

对于构建本模型，我们需要一个缺陷分布模型和一个缺陷密度性能基线数据。假设 A 公司 X 类项目的缺陷分布模型如下：

缺陷发现阶段	缺陷比例
1-User Requirement / System Requirement Review	3%
2-High Level Design Review	4%
3-Low Level Design Review	4%
4-Unit Test	14%
5-Code Review	15%
6-System Integration Test	51%
7-External Defects	9%

假设 A 公司 X 类项目的缺陷密度性能基线为 M1 = [25±2] 个/千行。

设定质量目标

按照项目给定的范围，进行项目规模估算。假设估算的项目规模是 Size = 20，000 行，那么，根据缺陷密度性能基线数据，可以

推算，该项目预计的缺陷总数为 M2 = M1*Size/1000，得出 M2 = [500±40] 个。然后按照缺陷分布模型的百分比，可以把缺陷发现指标分解到各个阶段中，如下表：

项目实施阶段	缺陷发现质量目标 （个）
1-User Requirement / System Requirement Review	14±1
2-High Level Design Review	19±2
3-Low Level Design Review	19±2
4-Unit Test	71±6
5-Code Review	75±6
6-System Integration Test	255±20
7-External Defects	47±4

这样我们就得到了每个阶段缺陷发现的目标数和控制上下限，并作为质量目标固定下来。

运用缺陷分布模型进行质量目标跟踪与质量预测

大家也许会问，那么如果某个阶段没有达到质量目标或者超额完成质量目标怎么调整下一阶段的质量目标情况？

在回答这个问题之前我们先基于这样一个假设：即同样技术水平等级的人，遵循同样的过程，其产品的总体质量表现是一致的。那么我们由此假设可以推断，对于一个给定项目规模来说，其总缺陷数是不变的。

还以上面的数据为例，假设到代码基线化时，Unit Test 和 Code Review 一共比计划少发现 30 个缺陷。那么，基于缺陷总量不变的假设，这 30 个缺陷势必要被泄漏到后续的系统测试和交付后的环节中。

按照缺陷分布的比例，系统测试缺陷将需要多发现 M3 = 30 *

51%/(51%+9%) = 26±2 个， 这样，调整后的系统测试缺陷发现质量目标将为：[381 ± 22] 个。预计泄漏到客户现场的缺陷将为：[51 ± 4] 个。

假设在后期维护期间，原先预计的修复外部缺陷的人天数为 47 人天，即一个缺陷 1 人天，按照调整后的质量预测结果估计，修复缺陷的维护工作量将增加 4 人天，变为 51 人天。

结束语

本文只是从一个维度基于若干假设条件下，对质量目标分解和质量预测作了一个实例介绍。大家在具体运用时候，也可以基于本公司的实际情况，选择适合本公司的建模方法去实际运用。

Unix 下自动化测试实践

作者：方耀

【前言】

网上看到很多关于用 qtp、winrunner 等工具来进行页面自动化测试的应用，但随着软件产业的发展，越来越多更大型的系统得到应用，随之变化的就是后台的变换，比如 unix、linux、solaris、aix 等。早期的很多系统都是基于 windows server 的应用和部署，qtp 等工具很好的解决了自动化回归测试的问题，但对于 unix 平台下的测试，它就有点无能为力。而截止目前，还没有出现 unix 下的自动化回归测试工具，所以，我们工作中的自动化测试工具或脚本都是自己分析、设计、编码和测试，最终达到我们自己测试的需求，目的就是要提高在后台测试工作中的效率，让计算机帮我们完成大部分的手工劳动。笔者根据工作的过程和体会，把工作中 unix 平台下自动化测试应用的经验跟大家分享，希望能够起到抛砖引玉的效果。闲话少说，我们“挨踢”人喜欢直接，还是用例子来说明一下。

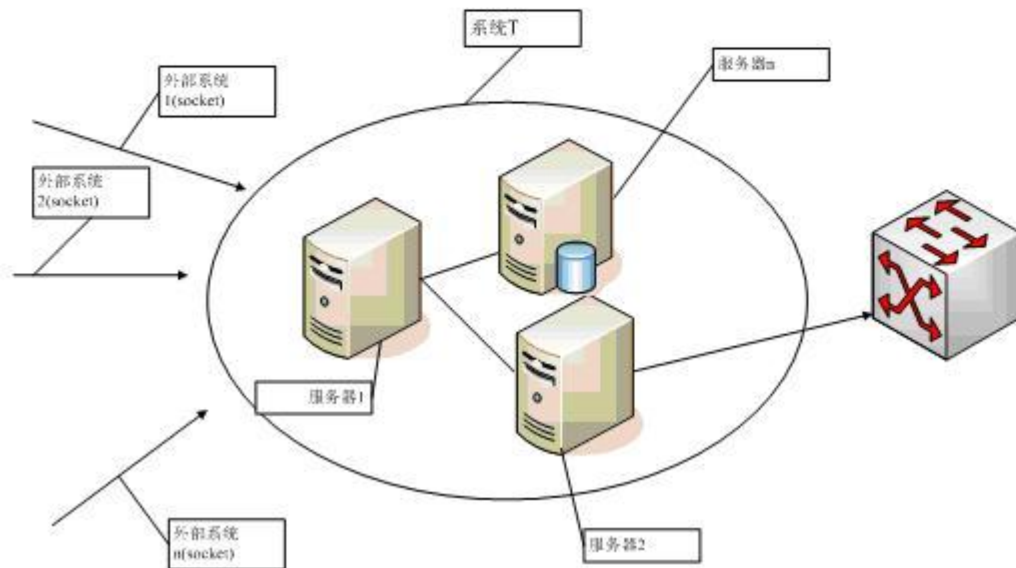
【举例 1】

背景：

某应用系统 T 作为一个信息处理平台，要接收处理各种外部系统的电子单，而这种电子单是根据一定协议进行发送和接收、并处理返回的，但这种协议不是基于通用的 TCP、UDP 等协议，是各系统之间统一定制的特殊行业协议。而在这种电子单中，包含了各种各样的业务信息，系统会根据这些不同的信息进行不同的处理，得到的处理结果自然也是不同的。

应用系统 T 的前身是应用系统 N，系统 N 所管理的外部系统比

较分散，所用的协议与系统 T 也有差别，但有一个共同点就是，它们对于同样的业务信息所处理的结果应是一样，而且都会持久化保存数据库。下面的图是系统 T 的大概业务逻辑图：



工作平台：

OS：HP_UNIX

DB：Sybase

WebService：Tomcat

Develop Language：Java

测试工作场景：

1、 按照不同业务制造大量电子单，然后发送到系统 T 的指定端口；

2、 发送电子单后，将系统 T 执行的结果与系统 N 的结果进行对比,检查系统 T 的结果是否执行正确；

测试困难分析：

1、 我们没有外部系统，如何模拟外部系统发送电子单？

2、 系统 T 管理了 7 中业务类型，而每种业务类型大概有 100 种业务，如果全部覆盖，大概需要 700 多条测试数据，手工制作不太现实；

3、 大量的电子单的处理结果，难道都用肉眼比对？

需求分析：

1、 编写模拟外部系统的发单客户端（基于 socket 协议）；

2、 使用系统 N 的海量电子单，然后按系统 T 的协议进行转换，用第一步的客户端发送（能保证覆盖所有业务）；

3、 编写程序，对系统 N 的处理结果和系统 T 的处理结果进行比对，然后打印出详细的对比结果，以供开发员定位错误问题所在；

工具实现步骤：

1、 由于开发平台都是 java，为了跟系统 T 衔接的更好，我们选用 java 编写自动化测试工具。当然，测试工具编写人要对 Java 比较熟悉；

2、 对于需求 1，我们需要知道如何进行 socket 数据的收发（网上资源大把，可以参考）；

3、 对于需求 2，我们需要知道如何进行数据库的操作（JDBC 资源网上也是大把）。数据库操作的目的，就是要从系统 N 的数据库中把旧的电子单取出来；

4、 对于需求 3，在基于上面的第 2 和第 3 点上，当电子单发送后，系统 T 中会记录处理结果，然后就是要编写算法，对新旧结果进行比对；

5、 对于测试工具还应该实现可配置化，比如服务器 IP、服务端口，旧数据库的 IP、用户名、密码，新数据库的 IP、用户名、密码，业务类型等，这就要求引入第三方的 jar 包，对 xml 文件进行解析；

部分代码：

下面部分代码说明：

此代码是 java 语言编写，IDE 是 Eclipse3.1；

下面是主体类（对比指令类）的一段主要代码，主要实现系统T和系统N的指令对比，并打印详细的对比日志。这其中可以看到有一个rx的变量，这是一个读取外部配置文件的对象，系统T和系统N的数据库连接信息、电子工单类型、对比条件等都是通过配置文件进行自定义的；

下面的代码用到的第三方jar包有jconn3.jar（sybase数据库连接）、dom4j-1.4.jar（读取配置文件）等；

```
try{
    FileWriter fw = new FileWriter("Instruct_err_order.txt");
    FileWriter fw_log = new FileWriter("Contrast_log.txt");
    FileWriter fw_true = new FileWriter("Contrast_true_order.txt");
    PrintWriter out = new PrintWriter(fw);
    PrintWriter out_log = new PrintWriter(fw_log);
    PrintWriter out_true = new PrintWriter(fw_true);

    out_log.println("Contrasting start time is:"+new Date());

out_log.println("-----------------------------");

    //int
grp_id=Integer.valueOf(rx.getItem("/project/dbinfo","old_db_order_grp"+table_id));
```

```
                    String
start_time=rx.getItem("/project/config","contrast_start_time");
                    String
end_time=rx.getItem("/project/config","contrast_end_time");

                  //2006.04.29 修改，加入*****，然后把新加
入的字段插入到临时表中，为
                  //后面的统计做准备
                    String
order_type=rx.getItem("/project/config","order_type");
                    //指令对比工单类型的判断
                    if(order_type.equals("all")){
                     str_Tips=" and 1=1 ";
                     out_log.println("Contrast workorder type
is:all");
                    }else{
                     String[] tmp_order;
                     tmp_order=order_type.split(",");
                     str_Tips="                                and
(workorder_type_id='"+tmp_order[0]+"'";
                     out_log.print("Contrast workorder type
is:"+tmp_order[0]);
                     for(int i=1;i<tmp_order.length;i++){
                        str_Tips=str_Tips+"                     or
workorder_type_id='"+tmp_order[i]+"'";
                        out_log.print(","+tmp_order[i]);
                     }
                     str_Tips=str_Tips+")";
                     out_log.println();
```

```
                    //System.out.println(str_Tips);
                    }

out_log.println("----------------------------------------------------------")
;
                    //这里复用一下 order_type 变量，为了增
加对比指令时的条件，因为
    *******等工单的类型
                    //都同**相同，可能还要增加其他的条件
才能过滤

order_type=rx.getItem("/project/config","buz_condition");
                    sql_temp="select
    workorder_no,office_id,officegrp_id,product_id,business_id,recei
ve_time
    from "+new_ordername+" where (office_id is not null) and "+
                    "receive_time>='"+start_time+"' and
    receive_time<='"+end_time+"'"+str_Tips+" "+order_type+" order
by
    workorder_no";
                    //System.out.println(sql_temp);

rs_temp=stmt_temp.executeQuery(sql_temp);
                    rs_temp.next();
                    int i=0;
                    String TempOrder="";
                    String oldOrder="";
                    while(!rs_temp.isAfterLast()){
                    //2006.04.29 增加，为生产环境对比而增
```

加

```
    product_id_tmp=Integer.valueOf(rs_temp.getString(4).trim());

business_id_tmp=Integer.valueOf(rs_temp.getString(5).trim());

office_id_tmp=Integer.valueOf(rs_temp.getString(2).trim());
      officegrp_id_tmp=Integer.valueOf(rs_temp.getString(3).trim());

date_tmp=rs_temp.getString(6).trim();

    old_ordername=rx.getItem("/project/dbinfo","old_db_ordername");

    old_instrctname=rx.getItem("/project/dbinfo","old_db_instrctname
");
                                TempOrder=rs_temp.getString(1).trim();
                                sql_old="select      numListNo,szRecord
from  "+old_ordername+"  where  szRecordID='"+TempOrder+"'  and
iLogID=-1 order by numListNo";
                                rs_old=stmt_old.executeQuery(sql_old);
                                rs_old.next();
                                if(rs_old.getRow()<=0){
                                    rs_temp.next();
                                    continue;
                                }
                               i=Integer.valueOf(rs_old.getString(1));
                               oldOrder=rs_old.getString(2);
                               sql_old="select
szCmd,iCmdID,iChildCmdID    from    "+old_instrctname+"    where
```

```
numListNo="+i+" order by iCmdID";
                        rs_old=stmt_old.executeQuery(sql_old);
                        rs_old.next();
                        int old_rs_count=rs_old.getRow();
                        if (old_rs_count<=0){
                            out_log.println("Old workorder unique
No is:"+i);
                            out_log.println("Workorder No
is:"+TempOrder);
                         out_log.println("Old workorder's instruct
is empty,it can not contrast instructs!");

   out_log.println("----------------------------");
                         rs_temp.next();
                         CountOrder1=CountOrder1+1;
                         continue;
                        }
    //                  else{
    //                   sql_temp="select count(*) from
"+new_ordername+" a where
a.workorder_no='"+rs_old.getString("szRecordID").toString().trim()+
"'";
    //
   rs_temp=stmt_new.executeQuery(sql_temp);
    //                   rs_temp.next();
    //                   if(rs_temp.getRow()<=0){
    //                       continue;
    //                    }
    //                  }
```

```
out_log.println("Old workorder unique No is:"+i);
out_log.println("Workorder No is:"+TempOrder);

//构建新工单查询 sql 语句，按照工单号查找
sql_new="select instruct_id,childinstruct_id,instruct_text from "+new_instrctname+" where workorder_no='"+TempOrder+"' order by instruct_id";
rs_new=stmt_new.executeQuery(sql_new);
rs_new.next();
int new_rs_count=rs_new.getRow();
if(new_rs_count<=0){
    int insold_id=1;
    while(!rs_old.isAfterLast()){
        out_log.println("The "+insold_id+"'s instruct 'iCmdID' is(old):"+rs_old.getString("iCmdID").trim());
        out_log.println("The "+insold_id+"'s instruct 'iChildCmdID' is(old):"+rs_old.getString("iChildCmdID").trim());
        out_log.println("The "+insold_id+"'s instruct is(old):"+rs_old.getString("szCmd").trim());
        rs_old.next();
        insold_id=insold_id+1;
    }
    out_log.println(Contrast_result1);

out_log.println("-----------------------------");
```

```
                    out.println("Old workorder unique No is:"+i);
                    out.println("Workorder No is:"+TempOrder);
                out.println("New workorder's content is:"+woc.converse(oldOrder,"1"));
                    out.println("Old workorder's content is:"+oldOrder);
                out.println(Contrast_result1);

out.println("-----------------------------");
            }
            else{
                boolean isTrue=true;
                int ins_id=1;
                while(!rs_old.isAfterLast()){
                    out_log.println("The "+ins_id+"'s instruct 'iCmdID' is(old):"+rs_old.getString("iCmdID").trim());
                    out_log.println("The "+ins_id+"'s instruct 'iChildCmdID' is(old):"+rs_old.getString("iChildCmdID").trim());
                    out_log.println("The "+ins_id+"'s instruct is(old):"+rs_old.getString("szCmd").trim());
                    if(rs_new.isAfterLast()){
                        out_log.println("The "+ins_id+"'s instruct is empty(new)!");
                        isTrue=false;
                        //继续输出旧指令
                        rs_old.next();
```

```
ins_id=ins_id+1;
while(!rs_old.isAfterLast()){
    out_log.println("The "+ins_id+"'s instruct 'iCmdID' is(old):"+rs_old.getString("iCmdID").trim());
    out_log.println("The "+ins_id+"'s instruct 'iChildCmdID' is(old):"+rs_old.getString("iChildCmdID").trim());
    out_log.println("The "+ins_id+"'s instruct is(old):"+rs_old.getString("szCmd").trim());
    rs_old.next();
    ins_id=ins_id+1;
}
break;
}
else{
    out_log.println("The "+ins_id+"'s instruct 'instruct_id' is(new):"+rs_new.getString("instruct_id").trim());
    out_log.println("The "+ins_id+"'s instruct 'childinstruct_id' is(new):"+rs_new.getString("childinstruct_id").trim());
    out_log.println("The "+ins_id+"'s instruct is(new):"+rs_new.getString("instruct_text").trim());
    if(contrastType.equals("id")){
        //指令对比方式
        if
(!rs_old.getString("iCmdID").trim().equals(rs_new.getString("instruct_id").trim()) ||
```

```
(!rs_old.getString("iChildCmdID").trim().equals(rs_new.getString("childinstruct_id").trim()))){
                        //if
(!rs_old.getString("szCmd").trim().equals(rs_new.getString("instruct_text").trim())){
                            isTrue=false;
                            //break;
                        }
                    }else{
                        //内容对比方式
                        if
(!rs_old.getString("szCmd").trim().equals(rs_new.getString("instruct_text").trim())){
                            isTrue=false;
                        }
                    }
                    ins_id=ins_id+1;
                    rs_old.next();
                    rs_new.next();
                }
            }
            out_log.println("Instuct's contrasting result is:"+isTrue);

    out_log.println("-----------------------------");
            if(isTrue){
                CouTrue=CouTrue+1;

    out.println("-----------------------------");
```

```
out_true.println(woc.converse(oldOrder,"1"));
                    //2006.04.29 增加，为生产环境对比
而增加
                    str_Tips="insert                    into
t_Contrast_Instuct values"+
    "('"+TempOrder+"',"+office_id_tmp+","+officegrp_id_tmp+","
+product_id_tmp+","+business_id_tmp+",1,convert(datetime,'"+date_t
mp+"'))";
                    //System.out.println(str_Tips);
                    stmt_Tips.executeUpdate(str_Tips);
                }
                else{
                    out.println("Old workorder unique
No is:"+i);
                    out.println("Workorder              No
is:"+TempOrder);
                    out.println("New workorder's content
is:"+woc.converse(oldOrder,"1"));
                    out.println("Old workorder's content
is:"+oldOrder);
                    out.println(Contrast_result2);

out.println("------------------------------");
                    //2006.04.29 增加，为生产环境对比
而增加
                    str_Tips="insert                    into
t_Contrast_Instuct values"+
    "('"+TempOrder+"',"+office_id_tmp+","+officegrp_id_tmp+","
```

```
+product_id_tmp+","+business_id_tmp+

    ",0,convert(datetime,'"+date_tmp+"'))";
                                stmt_Tips.executeUpdate(str_Tips);
                             }
                          }
                          rs_temp.next();
                          CountOrder2=CountOrder2+1;
                        }
                          //------------------------
                          DecimalFormat df = new DecimalFormat();
                          df.applyPattern("#.0000%");
                          //------------------------
                      out_log.println("The number which instruct's
contrasting is ture is:"+CouTrue);
                        EndTime=new Date();
                        out_log.println("Contrasting end time
is:"+EndTime);
                        out_log.println("It has "+CountOrder1+" old
workorder instructs is empty,so it can not contrast the instructs!");
                        out_log.println("Have been contrasted
"+CountOrder2+" workorders!");
     out_log.println("-----------------------------------------------------
---------");
                        out_log.println("The right percent
is :"+df.format((double)CouTrue /(double)CountOrder2));
                        out_log.println("---------------------");
                        System.out.println("The number which
instruct's contrasting is ture is:"+CouTrue);
```

```
                    System.out.println("Contrasting end time is:"+EndTime);
                    System.out.println("It has "+CountOrder1+" old workorder instructs is empty,so it can not contrast the instructs!");
                    System.out.println("Have been contrasted "+CountOrder2+" workorders!");
                    System.out.println("The right percent is :"+df.format((double)CouTrue /(double)CountOrder2));
        }catch(java.io.IOException ex){
                System.out.println("File not found!");
            }
```

【举例 2】

背景：

某系统 M 对外开放了 3 个基于 SOAP 协议的接口，而每个接口都提供了多达数十种的业务操作，而业务操作可能有先后依赖关系。

工作平台：

OS：Solaris 9.0

DB：Oracle

WebService：Weblogic 8.1

Develop Language：Java

测试工作场景：

构造不同业务的 xml 文件,比如开帐号（addUser.xml）,修改帐号信息(mdifyUser.xml)等，用 xmlspy 发送，然后验证返回信息是否正确；

测试困难分析：

1、 然后用 xmlspy 工具一次发一条的方式来进行回归，需要

大量的重复劳动；每次回归测试接口，都要制作达 30 个以上的 xml 文件，由于系统 M 的特殊性，工作环境不能直连，只能通过一台 Linux 服务器或者一台 windows 2003server 服务器中转，此 windows 服务器上安装有 xmlspy，但 windows 服务器的远程连接 license 只有 2 个，经常抢不到连接，后果就是延误测试时间或者加班测试；

需求分析：

由于 Linux 服务器不受连接限制，我们可以通过 Linux 服务器来发送

xml 文件给接口，不受 windows 的 license 限制；

2、 需要编写工具，能批量发送 xml 文件给接口服务；

3、 需要该工具能按定义顺序发送 xml 文件并检查返回结果是否正确，并打印出返回结果的关键字信息，供判断出错问题所在；

4、 对于该测试工具，更主要的是要能实现可配置，比如接口个数、文件发送顺序、参数化字段、主关键字值等；

工具实现步骤：

对于需求 1，我们需要知道如何发送 xml 文件到接口服务端口（笔者开始在此事想用基于 soap 协议的 jar 来构建 xml 文件，后来由于需要回归的 xml 文件太多，全部参数化不太现实，就改用文件发送方式。此处可以共享一下笔者借鉴的东西，大家可以到网上搜索 SOAPClient4XG 关键字）；

对于需求 2，其实很简单，知道 Java 取文件夹和文件操作、循环操作即可；

对于需求 3，也很简单，知道 Java 的文件流操作和 BufferedReader 类操作即可；

对于需求 4，在上面例子 1 中的测试工具编写要求中的第 5 点已说明；

部分代码：

```
try{
    // 建立同接口之间的连接，以备发送接口文件。
        URL url = new URL(SOAPUrl);
        URLConnection connection = url.openConnection();
        HttpURLConnection httpConn = (HttpURLConnection) connection;

        // 打开输入的接口 xml 文件
        FileInputStream fin = new FileInputStream(xmlFile2Send);
        // 初始一个空的字节输出流
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        copy(fin,bout);
        fin.close();

        byte[] b =null;
        // 进行模板的参数替换
        b=getReplace(bout,interface_index,xml_file);

        // 设置 HTTP 参数
httpConn.setRequestProperty("Content-Length",String.valueOf(b.length));

httpConn.setRequestProperty("Content-Type","text/xml; charset=utf-8");

    httpConn.setRequestProperty("SOAPAction",SOAPAction);
        httpConn.setRequestMethod( "POST" );
```

```
httpConn.setDoOutput(true);
httpConn.setDoInput(true);

// 发送接口文件，发送完毕后关闭输出流
OutputStream out = httpConn.getOutputStream();
out.write(b);
out.close();

// 从对话通道中读取接口返回信息。
//System.out.println(xml_file);
outlog.println("------"+xml_path+"------");
InputStreamReader isr = new
InputStreamReader(httpConn.getInputStream());
BufferedReader in = new BufferedReader(isr);
String inputLine;
int int_start;
int int_end;
String str_inputLine="";

// 不断循环从通道中读取返回信息，直至读取为
空
while ((inputLine = in.readLine()) != null){
    str_inputLine=str_inputLine+inputLine;
}

// 打印 resultNo 节点的信息
int_start=str_inputLine.indexOf("ext:resultNo");
int_end=str_inputLine.indexOf("/ext:resultNo");
if(str_inputLine.indexOf("ext:resultNo")>=0){
```

```
                outlog.println(xml_file+"                                   :
"+str_inputLine.substring(int_start-1,int_end+14));
            }

            // 当 resultNo 的值不为 0 时，还需打印 resultInfo 信息，
以供查问题
            int_start=str_inputLine.indexOf("ext:resultInfo");
            int_end=str_inputLine.indexOf("/ext:resultInfo");
            if(str_inputLine.indexOf("ext:resultInfo")>=0){
                if(str_inputLine.indexOf(">0<")<0){
                    outlog.println(xml_file+"                               :
"+str_inputLine.substring(int_start-1,int_end+16));
                }
            }
            outlog.println();

              }catch(IOException ex){
                ex.printStackTrace();
                outlog.println(xml_file + " : this file is failed!");
                outlog.println();
              }
        }

        // 同步拷贝文件流到字节输出流，并保证在拷贝过程中，
这两个流不受其他线程影响
        public void copy(InputStream in, OutputStream out) throws
IOException {

           synchronized (in) {
```

```
    synchronized (out) {
        byte[] buffer = new byte[256];
        while (true) {
            int bytesRead = in.read(buffer);
            if (bytesRead == -1) break;
            out.write(buffer, 0, bytesRead);
        }
    }
  }
}
```

【举例 3】

背景：

系统 M 有一后台服务，是该系统的最前端模块，它负责接收并处理基于国际通用的 X 协议包。而该系统将处理多达 200 多种的业务组合，接收不同的业务包，处理返回的结果就不同。但有一点需要注意，对于同一个包，在基础数据不变的情况下，系统对它的处理返回结果也是不变的。

工作平台：

OS：Solaris 9.0

DB：Oracle

Develop Language：C++

测试工作场景：

构建不同业务包，用开源的客户端发送，然有用肉眼校验返回的结果是否正确；

测试困难分析：

每次系统回归，都要对 200 多种业务进行发包回归测试，需要进行大量的手工劳动；

现有的业务还在增加，也就是说，手工回归的工作量会继续慢慢增大；

需求分析：

1、 批量发送所有的业务包数据，并保存系统处理返回信息；

2、 可以用一次系统回归测试的正确结果做一个校验基准文件，以后每次回归，可以把当次测试的系统返回信息同基准文件进行比对；

工具实现步骤：

1、 对于需求 1，由于这是后台发包，而且有开源的客户端，我们在 Solaris 9.0 服务器上编译一个客户端，然后用 shell 编写脚本，循环发包即可，但得事先把测试数据准备齐全和正确；

2、 对于需求 2，就是对 shell 脚本的要求了，主要用到了 head、perl、diff 等命令；

需要注意的是，这个脚本完成后，只需要对不断增加的业务测试数据进行维护即可；

部分代码：

```
for i in $FILE_LIST
do
    while read LINE
    do
        ACCOUNT=`echo $LINE | awk -F@ '{print $1}'`
        FILE_NAME="$ACCOUNT"_"$i"
        ./radtest $LINE > $TEMP_DIR/$FILE_NAME
        ID=`head -1 $TEMP_DIR/$FILE_NAME | awk '{printf $5}'`
        IPPORT=`head -1
```

```
$TEMP_DIR/$FILE_NAME | awk '{printf $7}'`
              #sed 's/'"$ID"'/999/g' $TEMP_DIR/$FILE_NAME > $TEMP_DIR/$FILE_NAME.bak
              #rm $TEMP_DIR/$FILE_NAME
              #mv $TEMP_DIR/$FILE_NAME.bak $TEMP_DIR/$FILE_NAME
              perl -pi -e 's/id '"$ID"'/id 999/g' $TEMP_DIR/$FILE_NAME
              perl -pi -e 's/id='"$ID"'/id=999/g' $TEMP_DIR/$FILE_NAME
              perl -pi -e 's/'"$IPPORT"'/'"$IP":"$PORT"'/g' $TEMP_DIR/$FILE_NAME
              DIFF=`diff $TEMP_DIR/$FILE_NAME $TRUE_RESULT_DIR/$FILE_NAME | wc -w`
              if [ $DIFF -gt 0 ]
              then
                   printf "This account is error : $FILE_NAME\n" >> $RESULT
                   COUNT=$((COUNT+1))
              fi
         done < $DATA_DIR/$i
         LOOP=$((LOOP+1))
         printf "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\bProcess: $LOOP / $SUM"
    done
```

【总结】

当这些测试工具完成后，不言而喻，测试的工作就轻松、简单多了！而且，测试工具提供的详细的日志，给开发员的纠错提供了

很好的帮助，大大的节省了开发和测试成本。

上面的 3 个例子，都是笔者在工作中实践总结出来的，这里主要讲的是测试工具或脚本的产生过程和思路，设计到的技术也无非是大家耳熟能详的 Java、JDBC、Shell 等，由于商业原因，笔者不方便把工具全部共享出来。下面我们来分析一下这些测试工具共同点和不同点。

共同点：

测试工作很难开展（外部条件未达到）或者是测试工作量巨大（而且是大量的纯手工有规律的劳动）；

可以借助编程或者写 shell 脚本来实现自动化；

选择开发工具时，尽量跟后台开发语言一致；

不同点：

当涉及到数据库、协议等时，得用编程来实现自动化。而对于后台已经有客户端、操作命令等时，可以用 shell 来实现自动化。当然，涉及的方面较多时，可能既得编程，又得写 shell 脚本；

不同的数据库、协议等，所使用的第三方 jar 是不同的。对于 shell 脚本也是一样，不同的平台，可能一些基本命令也不同，比如 awk 和 nawk，等等；

当然，这里的例子在众多的测试情况中，是微不足道的，但是只要我们养成一种良好的思考习惯，注意积累，我想我们的测试工作可能会越来越轻松，越来越快乐！

如果有对其中部分知识点感兴趣的话，可以加我一起讨论（qq：16029687，msn：owen_fy@hotmail.com）。

在 QTP 中随机取下拉菜单的值

作者：风过无息

摘要：产生期望的随机数来选取到动态的下拉菜单的值。

关键字：随机数；GetROProperty

有网友在论坛上提出问题，在使用 QTP 中如何随机选取动态的下拉菜单。在此笔者总结了一些测试经验，利用 51testing 的登录界面（http://bbs.51testing.com/logging.php?action=login）作为测试页面进行讲解。

首先我们拿登录页面中的　安全提问　这个下拉菜单作为测试对象。

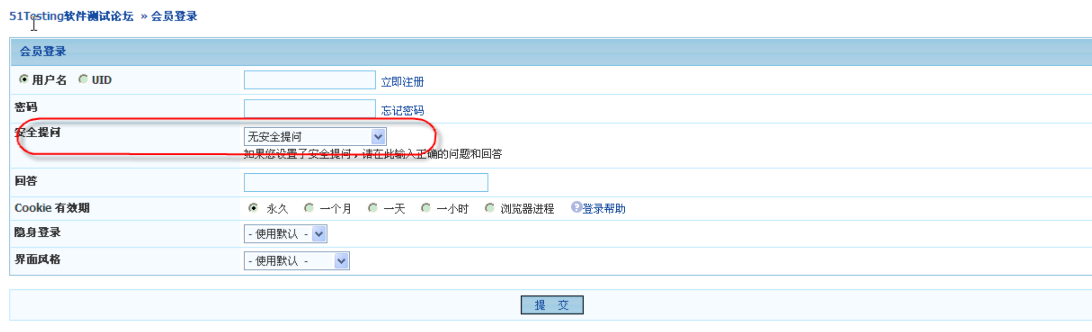


图 1

我们可以先录制一段选取下拉菜单的脚本。

Browser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试").WebList("questionid").Select "您个人计算机的型号"。

备注：无

这边有个技术处理就是使用#加数字来选择我们的下拉菜单。

Browser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试

").WebList("questionid").Select "#2"

备注：这个方法在论坛上也出现过，比较适合这个例子。

下面我们要取得下拉菜单中选项的个数。

Browser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试").WebList("questionid").GetROProperty ("items count")

备注：这边使用 GetROProperty，应该算比较常见的，所以不多做解释。

接下来是要个随机函数，参考帮助。

```
Randomize
x=RandomNumber (0,2)
```

备注：这边是随机生成 0-2 之间的三个数字中的一个。

我们可以把随机函数写成 function，方便以后使用。

```
Function Get_Ran(i)
      Randomize
   Get_Ran=RandomNumber (0,i)
End Function
```

备注：这边需要注意的就是使用了函数返回值

最后我们把脚本整合起来

```
Function Get_Ran(i)
      Randomize
   Get_Ran=RandomNumber (0,i)
End Function
```

Get_Count=Browser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试").WebList("questionid").GetROProperty ("items count")

Ran_Number=Get_Ran(Get_Count-1)

Browser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试").WebList("questionid").Select "#"&Ran_NumberBrowser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试").WebList("questionid").GetROProperty ("items count")

Browser("51Testing 软件测试论坛 测试 | 软件测试").Page("51Testing 软件测试论坛 测试 | 软件测试").WebList("questionid").GetROProperty ("items count")

备注:需要注意的就是在下拉菜单选择的时候从#0 开始计算的，所以随机数字从 0 开始，传入的值也需要减去 1。

题外话：这边主要是使用 QTP 自带的随机数字函数这个方法来实现随机选择下拉菜单的内容，其实这个随机数字可以开展到随机字符串。因为我们经常会有一些输入域的测试，有的就需 255 个字节，多的就更可怕，使用随机函数能大大减少我们的工作量。而下面这个例子是实现在 abc 这三个字母中随机取出来拼成字符串。

```
Function makestring(inputlength)
If IsNumeric(inputlength) Then
For I = 1 To inputlength
'you may add a random function here
A = Array("a","b","c")
Randomize
x=RandomNumber (0,2)
B = A(x)
```

```
makestring =makestring +B
Next
msgbox ("output the string:"&makestring )
else
msgbox ("error format:"&inputlength)
End If
End Function
Call makestring("8")
```

需求不明确的情况下如何做测试

作者：张文

摘要：本文针对需求不明确的情况下如何做测试，列举了 3 个步骤。这些步骤，都是实际经验的总结。利用这些步骤，可以在需求不明确，或是没有需求的情况下，进行必要的测试工作。但是，这些都是不规范的方法。需求不明确，或是根本没有需求，这本身就是一件不规范的事情，无论是开发人员，还是测试人员都无法在不规范的环境中，做规范的事情。但是工作要继续，不能因为某些障碍而停止。这时，请参考每一个步骤，尽可能地完成测试工作。

关键字：需求；需求规格；测试需求；文档；猜测；沟通

软件生命周期中，需求是整个周期的源头。良好的开端，是成功的一半。需求的重要性自然不言而喻。但是，在很多企业中，并没有对需求引起足够的重视。原因并不是 PM 们不知道需求的重要性，而是商业竞争中不得不裁剪某些看似不能获得很大利益的步骤。

什么是需求？很多 PM 和开发人员都未必真正考虑过这个问题。IEEE 对需求有以下两种定义的方式。

1. 解决用户问题或达到用户目标需要具备的条件或能力
2. 遵守合同、协议、规范或其他要求

然后用规范的文档描述出来，就成了我们熟悉的 SRS。

我们常说的需求，其实并不是我们认为的 SRS。SRS 应该叫做需求规格说明书。那需求是什么呢？与需求规格有什么区别？

需求：对要实现的功能的粗略描述

需求规格：对需求的精确定义

我们知道，在软件开发过程中，只有得知了需求的精确定义，

才能开展工作。比如功能方面，编辑框能支持多少位字符。性能方面，时间和容量规定等。当然还包含其他非功能，性能方面的定义。

除了以上所说的需求，对于测试人员，还必须有测试需求。这个环节，很少有企业会重视。测试需求分为 2 方面：

需要测试哪些方面

软件是否可测，需要增加哪些开发需求

其中第一条，很多企业都列到了测试计划中，这也可以，没有规定一定要放到哪个文档里。但是对于第二条，可以说几乎没有多少企业去做。

接下来，在没有明确需求，需求规格，测试需求的情况下，我们怎么去做测试呢？现在很多企业，其实就是在这种情况下做项目的。

当测试人员接手一个项目后，第一件事情一定是想了解这个系统的功能，背景，架构。于是，马上就会想得到需求文档。但结果往往是失望的，根本没有文档，或者文档根本不具备参考价值。此时不必太失望，因为这种情况实在是太常见啦。这时，请试着从以下几个步骤着手。

查阅文档：文档是最具权威的，也是记忆最长久的。有时，我们的项目可能是在原有产品的基础上，进行版本升级。这时，先去找找，有没有原有版本留下的需求，或者是用户手册等文档。从这些文档中，了解项目的背景，系统的基本功能。这对了解新项目是有很大好处的。并且，在产品升级的项目中，验证老版本的功能在新版本中是否正常，也是一个必要的工作。可以先参考老版本的相关文档，设计新版本中的用例。

也有时，我们的项目是一个行业项目，比如金融项目。我们可以参考一些行业知识的书籍，文档。这对理解系统也有很大的好处。

实在没有文档，那只好暂时跳过这一步骤了。

在进入下一步骤之前，你可能得到了一些相关文档，也可能什么也没得到。无论如何，你可能对系统已经有了一些了解。这时，

请记录下来，写成文档。无论是对自己，还是对别人，在以后都可能极有参考价值。试想一下，如果前人已经给你留下了这些文档，你是否可以轻松很多？还要注意及时更新你的文档。因为你对系统的理解，随时都在变化着，一定要保证你的文档和当前你对系统的理解是一致的。

试着使用系统，根据经验和常识猜测：既然没有需求，那可以推测，该项目的管理一定是很糟糕的，对测试也不会投入很大的成本。因此，测试人员一般都是在编码完成后才进入项目。这时，应该已经可以看到成型的系统了。在没有需求的情况下，试着先“玩”一下系统吧。在这过程中，你应该对系统有可更深入的认识，在上一阶段中，你可能留下很多疑惑或是猜测，这时应该能排除一部分了。

使用系统的同时，你应该具备行业知识。系统可能是针对某个专业领域设计的。例如一个期货交易系统。你没有基本的期货知识，比如什么是持仓，什么是平仓。那么你如何能真正理解这个系统呢？当你有了业务知识以后，你会进行更深入的思考，来全面测试系统。

你还需要具备良好的软件知识。比如某些控件的特性。单选框只能单选，不能多选。日历控件是否可以手工输入非法格式等。这些都是应具备的意识。

最后加上你的主观判断，你对系统的整体感觉怎么样？是否越用越厌烦，为什么厌烦。系统的反应速度是否可以容忍，细节处理是否圆滑，等等。

在你认识系统的时候，可以使用一些方法，来帮助你更有效率地学习。比如可以画一些流程图。一图胜万语。同时，你也留下宝贵的文档。当然，这个步骤中，你也要随时注意保留和更新文档，以备后用。

沟通：需求规格不一定非要以文档的形式表现出来。软件既然能做出来，那肯定是有需求的。而最清除需求的，一定是软件的直接制造者，开发人员。开发人员自己知道需求，但一般不会主动和

测试人员沟通。因此，测试一定要主动和开发人员沟通。可以安排会议，让开发人员给测试人员介绍系统，并演示系统。让测试人员对系统有一个整体了解。然后测试人员能进行更细致的测试。在进行细致测试的时候，一定会有更多不明确的地方。这时就需要利用自己的行业知识，计算机知识等，猜测一部分。不需要每个细节都去询问开发人员。因为开发人员也有自己的工作，他们不希望花太多时间来给你解释。

有些项目中，客户会直接参与到项目组来。这时，测试人员在权限允许的情况下，可以和客户进行沟通。客户那得来的需求，是最原始的需求。但是，客户未必有良好的表达能力来描述希望的功能，也未必有计算机知识，因此不能描述出一些隐式的需求。在被允许的情况下，测试人员可以和客户进行交流，不仅可以帮助客户正确描述出真实需求，测试人员也能详细了解需求。但是项目是要考虑成本的，客户的期望是无限制的。在客户提出需求以后，测试人员要先和 PM 或其他相关负责人协商后，才能将与客户交流得来的需求，作为测试的依据。同事，第一时间告知相关开发人员最新的信息，也记录成文档。这时，你就将非文档形式的需求，转换为文档形式了。至于文档的格式，不一定要按照标准 SRS 的格式。因为它本身就不是个规范的 SRS。以任何容易理解的方式，组织你的文档。

有时候，会根本找不到可以沟通的人。不要奇怪，确实就是有这种时候。比如：

1. 测试一个开源软件

2. 接到一个测试外包，但又没有得到相关文档，为了追求利益，还是接下了

3. 软件项目组的部分人员已经联系不上等等

这时候，一方面需要 PM 协调获取相关资料，联络相关人员。另一方面，测试人员也可组织头脑风暴，利用集体的智慧，共同探讨和猜测软件中的各个环节。也可以安排 Bug Bash，让尽可能多的

人员参与随机测试。一定会有人提出具有创造性的意见的。

在进行以上步骤的时候，利用良好的工具，能让你事半功倍。我经常在使用的一个工具，就是 Mindjet MindManager。这是一个很好的，帮助扩展思维的工具。它以分支的形式，来表现你的思维层次。你可以先列出个最基本的系统整体结构，然后逐步细化，增加分支。不要急于一次就将真个系统分析透彻，这是不可能的。你在进行以上步骤的时候，随时会细化这个结构。当项目结束后，看看这个结构图，简直可以当作 SRS 来参考了！

基于 TMM 的软件测试过程评估

作者：薛丽

摘要：为了提高软件测试的效率和软件测试的质量，软件人员不仅要研究软件测试的各种技术、方法和工具，还必须注重软件测试过程的管理和改进。在这种需求的牵引下，Illene Burnstein、C. Robert Carlson 和 Taratip Suwannasart 在 1996 年质量周会议上提出用以评估测试过程的软件测试成熟度模型（TMM）[1]。本文在介绍 TMM 五级结构的基础上，讨论在各成熟度等级上的可用于评估软件测试过程的对象和方法。

关键字：软件测试成熟度模型，测试过程 ，评估 ，度量

一、 TMM 简介

软件测试成熟度模型(TMM)[1]是当前影响力最大的软件测试过程模型，具有如下优点[2]：

等级水平结构、关键活动和角色的定义最为精细

测试相关因素覆盖最全面

支持测试过程成熟度增长

有定义良好的评估模型的支持

作为一类等级递增模型，TMM 体现了 20 世纪 50 年代到 20 世纪末的测试阶段划分和测试目标定义的发展历程。它来源于当前的工程实践总结，并充分利用了 Beizer 的测试人员思考模式的进化模型。它能够用于分析软件测试机构运作过程中最优秀或最混乱的区域，并辅助软件测试机构进行测试过程的评估与改进。

TMM 制定了五个成熟度等级：初始级，阶段定义级，集成级，管理和度量级，优化、缺陷预防和质量控制级。各级成熟度水平包含了一组成熟度目标和子目标，以及支持它们的任务、职责和活动。

1) 第一级 初始级

TMM 初始级软件测试过程的特点是测试过程无序，有时甚至是混乱的，几乎没有妥善的定义。初始级中软件的测试与调试常常被混为一谈，软件开发过程中缺乏测试资源，工具以及训练有素的测试人员。初始级的软件测试过程没有定义成熟度目标。

2) 第二级　阶段定义级

TMM 的阶段定义级中，测试活动是按照计划进行的，测试已具备基本的测试技术和方法（如白盒测试和黑盒测试），软件的测试与调试已经明确地被区分开。这时，测试被定义为软件生命周期中的一个阶段，它紧随在编码阶段之后。但在定义级中，测试计划往往在编码之后才得以制订，这显然有背于软件工程的要求。

TMM 的阶段定义级中需实现 3 个成熟度目标：制订测试与调试的目标和策略，启动测试计划过程，制度化基本的测试技术和方法。

3) 第三级　集成级

在集成级，测试不仅仅是跟随在编码阶段之后的一个阶段，它已被扩展成与软件生命周期融为一体的、一组已定义的活动。测试活动遵循软件生命周期的 V 字模型。测试人员在需求分析阶段便开始着手制订测试计划，并根据用户或客户需求建立测试目标，同时设计测试用例并制订测试通过准则。在集成级上，应成立软件测试机构，提供测试技术培训，关键的测试活动应有相应的测试工具予以支持。在该测试成熟度等级上，没有正式的评审程序，没有建立质量过程和产品属性的测试度量。集成级要实现 4 个成熟度目标，它们分别是：建立软件测试机构，制订技术培训计划，软件全寿命周期测试，控制和监督测试过程。

在 TMM 的定义级，测试过程中引入计划能力。在 TMM 的集成级，测试过程引入控制和监督活动。两者均为测试过程提供了可见性，为测试过程持续进行提供保证。

4) 第四级　管理和度量级

在管理和度量级，测试活动除测试被测程序外，还包括软件生命周期中各个阶段的评审，审查和追查，使测试活动涵盖了软件验

证和软件确认活动。根据管理和度量级的要求，软件工作产品以及与测试相关的工作产品，如测试计划，测试设计和测试步骤都要经过评审。因为测试是一个可以量化并度量的过程。为了度量测试过程，测试人员应建立测试数据库，收集和记录各软件工程项目中使用的测试用例，记录缺陷并按缺陷的严重程度划分等级。此外，所建立的测试规程应能够支持软件组织最终对测试过程的控制和度量。管理和度量级有 3 个要实现的成熟度目标：建立组织范围内的评审程序，建立测试过程的度量程序和软件质量评价。

5) 第五级　优化、预防缺陷和质量控制级

由于本级的测试过程是可重复的、已定义的、已管理的和已度量的，因此软件组织能够优化调整和持续改进测试过程。测试过程的管理为持续改进产品质量和过程质量提供指导，并提供必要的基础设施。优化、预防缺陷和质量控制级有 3 个要实现的成熟度目标：应用过程数据预防缺陷，质量控制，测试过程优化。

二、　基于 TMM 的测试过程评估

虽然 TMM 在管理和度量级才明确提出了正式的度量以及评估计划，但定义度量对象与度量方法的工作则可以在四级以下进行，因为度量本身就是支持更高成熟度目标的实现以及当前能进行的最佳测试活动的实施[3]。软件测试机构可从低级、粗略的度量做起，随着成熟度等级的提高,度量逐步深入和细化。与成熟度的等级关系相似，高等级的度量包含了低等级的度量。

下面讨论在各成熟度等级上，可应用于软件测试机构的度量对象和度量方法。

1) 初始级度量

初始级没有成熟度目标，但从这一级开始收集测试度量信息仍然是有必要的，如此测试机构才能建立相应的项目信息数据库，推动测试机构实现更高等级的测试过程成熟度。初始级实施度量的主要目的是建立基线过程，并为测试机构实现第二级的成熟度目标做

好准备。

处于 TMM 初始级的软件测试机构可度量以下信息：

a) 规模度量

规模对于测试项目，测试计划，成本 / 工作量预测，风险评估，效率度量而言，是一项非常重要的度量因素，它包括：

软件产品规模

测试对象数量

测试用例编写数量

测试用例执行数量

b) 缺陷度量

软件异常数量

缺陷率

测试机构在此级开始收集缺陷信息的目的是：在实际测试中对发现的各类问题有着文档化的记录。缺陷信息有助于评价软件质量，支持软件开发和测试过程的改进，同时对软件缺陷数据库的建立提供了基础。

c) 成本 / 工作量

测试项目总体成本

测试执行小时数

测试工作量成本

成本度量则有助于测试计划（TMM 2 级）中的测试成本预测。通过收集成本 / 工作量度量信息，软件测试机构就可以开始着手建立成本数据库，为将来的测试项目提供成本预测方面的支持。

2) 阶段定义级度量

此级中，软件测试机构应当选择支持测试过程基线工作和实现第二级成熟度目标的度量项。除了等级 1 中的成本 / 工作量，还应当收集各测试类型的时间 / 工作量数据，以支持和评估测试活动的效率。度量对象如下：

代码走查、系统测试的计划时间

代码走查、系统测试的执行时间

代码走查和系统测试是软件测试机构被授权的测试类型。如果有其它测试类型，则同样要搜集其设计和执行时间。

有助于制定，评估和改进测试计划的度量包括：

计划生成的测试用例数量

实际生成的测试用例数量

计划的测试覆盖率

实际的测试覆盖率

成本性能指数（CPI，指计划的测试时间与实际测试时间之间的比例[4]，这是评价测试计划的质量的最简单的方法）

为了实现 TMM 等级 2 中的成熟度目标，还应当收集缺陷相关度量数据：

测试各阶段（代码个人走查，代码会议走查，系统测试，系统回归测试等）发现的缺陷数量

各类型缺陷的数量

以上描述的度量能用于评估各类型测试的效率，同时也有助于测试机构评估测试过程变更的影响。收集的以上度量数据为缺陷预防，设置软件质量目标以及软件风险管理提供了有力支持。

3) 集成级度量

TMM 等级 3 建立了专门的测试机构，测试机构可以收集，存储，分析和应用度量信息，并进行相关培训，以便实施过程改进。培训计划部分也应当包括评估和使用测试与度量的工具。TMM 等级 3 中应当有工具支持数据收集（例如缺陷跟踪工具和覆盖率分析软件）。此级中，软件测试机构应当在成本允许范围内，开发实用的模版和工具，以支持收集和分析测试数据。

TMM 等级 3 中的一个重要成熟度目标就是控制和监督测试过程，因此评估测试过程效率就成了实现该目标的一个重要手段，同

时，测试过程效率也是决定测试停止的重要参考因素。测试过程效率的度量如下：

测试功能覆盖率

执行通过的测试用例数目

各测试人员每周生成的测试用例数目

缺陷修正率(DRL)

单位时间发现的不同严重度的软件缺陷发现数目

TMM 等级 3 中，软件机构的一个重要任务便是制定技术培训计划。以下度量项将有助于评估和改进技术培训计划：

培训人员人数/测试人员总数

培训成本(总计的和单个培训阶段的)

各培训阶段持续时间

给每个测试人员分配的培训时间

为了评估使用测试工具的效益，软件测试机构还应当度量：

评估测试工具的成本

测试工具使用培训的成本

购买或开发测试工具的成本

测试工具升级和维护的成本

测试机构可以通过一些成本度量方法对培训和工具的影响进行评估，例如：

培训之前和之后的测试成本或测试工作量

使用工具之前和之后的测试成本或测试工作量

同时，测试机构还应当在引进技术培训和测试工具之前和之后，对发现缺陷的数量进行监控。

软件测试组织在工作过程中要实时与开发方和客户方沟通与协商，这不仅对测试工作的顺利进行起着举足轻重的作用，而且是评估测试机构协同工作和效率的一个指标。三方沟通的度量如下：

测试执行过程中与开发方和客户方的沟通次数和时间长短

需要沟通的问题数量和实际沟通的问题数量

通过沟通解决问题的比例

沟通消耗的成本（时间、人力等）

TMM 等级 3 中，测试机构除了完成被授权的各类型测试外，还要进行相应的回归测试，因此测试用例的重用性也值得度量，以便对回归测试过程和效率进行监测。回归度量包括：

测试用例重用数目和比例

新增测试用例数目和比例

集成级中，因此测试活动与开发方行为密切相关，测试机构应当确保测试计划在测试实施之前完成，测试风险得到预测，测试用例应当基于软件需求，说明和设计文档。评估测试活动的度量如下：

确定测试需求花费小时数

制定测试计划花费小时数

测试设计花费小时数

从需求文档中生成的测试用例数目

从设计文档中生成的测试用例数目

制定需求跟踪矩阵花费小时数

评估需求覆盖率花费小时数

4) 管理和度量级度量

在 TMM 等级 4 中，测试机构正式运行了测试度量计划，还建立了软件质量评价和评审过程。此级中，测试机构的度量的范围很广，对质量成本，客户方/开发方的参与程度等因素也具备了度量能力。等级 4 的度量重点在软件产品度量，还有对评审的度量，以便对其进行评估和改进。最后此级还正式建立了完整的软件缺陷库。

对评审的度量如下：

评审项的规模

评审会议的次数和持续时间

评审平均每小时发现问题的数量

评审平均每小时进行的文档页数

评审解决问题的力度

既然 TMM 等级 4 中的测试度量是正式的，那么测试机构还应当度量测试度量活动本身相关信息，以评估度量活动的强弱区间，如：

度量培训的成本

度量工具的成本

维护度量数据库（如缺陷数据库）的成本

TMM 等级 4 中强调了软件质量评价的重要性，因此，达到此级目标的软件测试机构应当度量一系列的软件质量属性，包括为每一个测试项目选择合适的度量对象，并体现在测试计划中。以下列举了部分重要的质量属性和相关度量对象：

正确性：软件完成其规定功能的能力。通常使用的度量项是缺陷密度（每千行代码软件缺陷数量）

有效性：软件系统在规定时间和环境框架下表现期望性能的能力。通常使用的度量项是反应时间——系统相应用户请求的时间

可测试性：量化的可测试性表现为充分测试一个软件系统所需测试用例数量或一个模块的圈复杂度

可维护性：通常使用平均修复时间（MTTR），它反映了分析需求变更，重新设计模块，编码，测试和发布的时间。

可移植性：反映了将软件系统从一个软硬件环境转移到另一个软硬件环境中所需工作量。

可重用性：这一属性反映了新开发的程序代码在将来得到重用的可能性，反映这一属性的度量项是进入重用库的代码行数。

典型的软件缺陷库包含内容如下：

缺陷 ID

缺陷发现时间

软件 ID

缺陷类型

缺陷发现位置（模块或功能点）

缺陷来源

缺陷产生阶段

缺陷发现阶段

缺陷表征（体现在软件行为）

缺陷修复时间/成本

缺陷发现者

缺陷确认者

由上可知，每个测试过程都具有与其软件测试成熟度等级相对应的度量关注点和提供度量信息的能力。决策者可以根据其测试机构所处的测试成熟度等级制订出合理的度量和评估计划、采集对应的过程数据，进而通过对数据的分析结果制订改进测试过程的有效方案。

参考文献

[1] Burnstein I., A Testing Maturity Model for Software Test Process Assessment and Improvement. Software Quality Professional, September 1999, Volume 1(4)

[2] Weatherill, T., In the Testing Maturity Model Maze. Journal of Software Testing Professionals, Mar.2001: 8-13

[3] Burnstein Ilene. Practical Software Testing: A Process-oriented Approach. New York: Springer-Verlag, Inc., 2002

[4] Humphrey W.A Discipline for software Engineering, Addison-Wesley, Reading, MA, 1995

[5] Gelperin, D., A. Hayashi. "How to Support Better Software Testing." Application Development Trends May 1996: 42-48

[6] Rodger Drabick, Susan Burgess，Testing Capability Maturity Model. Testing Computer Software Professional, 1996

如何实现基于非标准控件开发的软件的自动化测试

作者：王菊玲

摘要：针对目前市场上软件自动化测试的主流工具，不能实现基于非标准 Windows 控件开发的软件的自动化测试，本文提出了使用 Rational Robot 外挂辅助自动化测试 DLL，通过 Windows 消息机制，实现对被测试应用程序的控制，从而实现自动化测试的一套可行的实施方案，为广大自动化测试爱好者，提供参考思路。

关键字：外挂 DLL；Windows 消息机制；非 Windows 标准控件；自动化测试

一、 前言

随着计算机软件产业的飞速发展，软件产商采用了各种开发模式来提高开发速度，比如说叠代开发，代码重用等技术，使得软件开发的速度大大地提高。同时，给软件质量保证人员带来了非常巨大的挑战。

在一些发达国家，软件测试人员与开发人员的比例会达到 1：1，或者 1：2，而在我们国内，虽然很多人已经认识到软件质量的重要性，有些大企业成立了专门的测试部门，但是传统的重视开发而轻视测试的思想依然占据主导地位。软件测试人员与开发人员的比例依然非常低。我就职的软件企业是一家著名的跨国软件企业，公司上层已经非常重视软件质量了，可是这个比例也只能达到 1：5，甚至只有 1：7。对于其他国内一般的软件企业来说，测试人员与开发人员比例之低就不言而喻了。

随之而来有个问题我想问，这么繁重的测试任务单纯地依靠手工测试能行吗？回答是否定的，我们需要采用自动化方式替代一部分的手工测试。

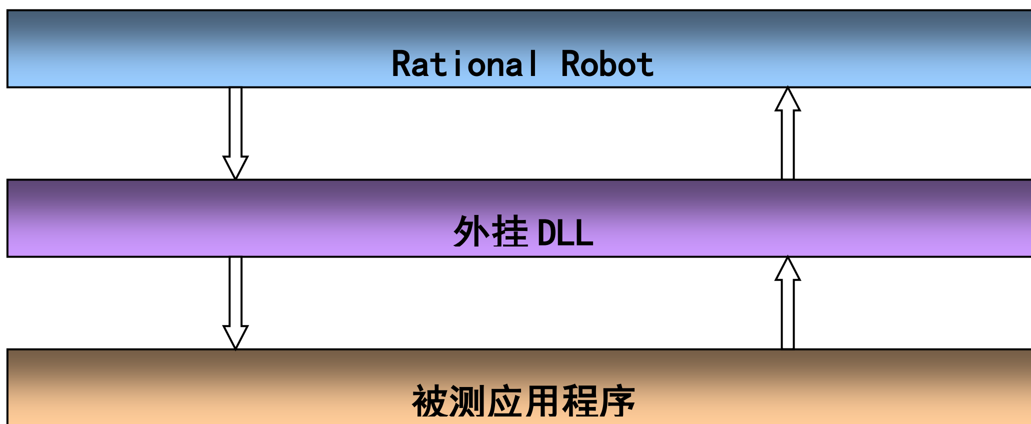
在实现自动化测试的过程中，我们曾遇到过有些控件根本无法

被自动化测试工具识别，当然对它的操作更加是无从谈起。那我们公司所开发的软件来说吧，为了使软件界面更加美观，风格更加统一，公司专门成立了一个部门，开发了一种软件界面的设计工具，对 Windows 标准控件进行二次开发并封装。对于这样的控件,Rational Robot 是没有办法控制的。

因为公司测试的需要，我对基于非 Windows 标准控件开发的软件实现自动化测试进行了比较深入的探索。琢磨出了一套方法想跟大家一起分享。那就是使用 Rational Robot 工具外挂 Dll,结合 Windows 操作系统中的消息机制的方式，实现对非 Windows 标准控件的识别和控制。

二、 自动化测试工程的整体架构

这套实现对非 Windows 标准控件开发的软件自动化测试工程，采用三层架构方式，Rational Robot 层，外挂 DLL 层，被测应用程序层。为了能够清晰地说明它们之间的关系，我想使用一个图来说明。如下图：



Rational Robot 层的主要工作是根据测试的需求，控制测试的过程，管理测试的检查点，编写测试代码，执行测试脚本，最后生成测试结果报告。

外挂 DLL 层，负责 Rational Robot 与被测试应用程序之间的通信，接收 Rational Robot 发送的指令信息，再将这些指令信息发送给

被测试应用程序端接收该消息的 DLL。应用程序端的 DLL，根据接收到的消息，完成对被测试应用程序的操作后，将消息发送给外挂在 Rational Robot 端的 DLL。这样一个来回完成 Rational Robot 与被测试应用程序之间的通信。

被测试应用程序层接收到辅助自动化测试的 DLL 发送过来的消息，完成测试人员预先设定好的操作步骤，完成对被测试应用程序的测试任务。

三、 系统的通信方式

使用 Rational Robot 自动化测试工具,把被测试的应用程序运行起来,我们可以通过进程监控工具查看,它们是属于父子进程关系。父子进程，也是两个独立的进程。那么运行在 Rational Robot 进程中的 DLL，是不能够直接操作被测试应用程序的进程的。我们怎么办呢？

如果对 Windows 操作系统比较深入了解的人会知道，Windows 操作系统提供一种消息机制，它可以实现不同进程空间的信息共享。如果能够把这种技术应用于自动化测试过程中，是否能行得通？

基于这样的思路，我们在 Rational Robot 端放置一个 DLL，同时在被测试的应用程序端也放置一个辅助自动化测试的 DLL。放置在被测试应用程序端的 DLL 要它不会对被测试应用程序本身造成任何影响，被测试应用程序不会因为辅助自动化测试的 DLL，而改变其控制的流程以及需要实现的具体功能。因此，我们采用当被测试应用程序在启动的时候，就把辅助自动化测试的 DLL 导入进来。就在这个地方增加几行代码，其他方面，被测试应用程序不再做任何的改动。

Rational Robot 所提供的编程语言为 VB Script,它是一种弱语言。像 For...Each 等基本的枚举语句不支持，指针当然也不支持，直接在 Rational Robot 里边发送消息和接收消息是无发实现的。为了解决此问题，我们采用在 DLL 中消息重构的方式，在外挂 DLL 层直接完成不同进程空间的消息传递，而 Rational Robot 层只需控制测

试流程和管理测试检查点。这样一来，系统层次比较分明，实现也变得简单。

这两个 DLL 实现了 Rational Robot 与被测应用程序之间的通信，完成了非 Windows 标准控件的识别和控制。

四、 自动化测试在 Rational Robot 中的实现

对于大多数使用 Rational Robot 进行自动化测试不深的人来说，也许根本就没有听说过能够在 Robot 中调用 DLL。在这里，我想告诉大家这样一个事实，Robot 中 DLL 的调用，让不可能能变可能。现在，告诉大家如何调用 DLL。

第一，DLL 放在什么地方

使用 Rational Administrator 新创建一个自动化测试工程，浏览目录到 TestDatastore\DefaultTestScriptDatastore\TMS_Scripts，我们会发现有一个目录，名称就叫 DLL。

这是非常有用的一个目录，如果把 DLL 放置在这个目录下面，在自动化脚本运行时，它直接就会去导入与 DLL 名称对应的 DLL，不需要在测试脚本中使用 DLL 所在目录的全路径了，DLL 放置在这个目录下面的效果跟放在 Windows System32 目录下面是一样的。

我们知道，自动化测试工程开发出来之后，不单是自动化测试工程师使用，更多的是给递交给测试某个产品的工程师使用的。一个产品同时会有多个测试工程师，测试环境也不同，我们总不能要求他们拷贝了这个自动化测试工程之后，再拷贝 DLL 到 Windows System32 下面吧，这样他们会嫌麻烦。特别是将测试操作系统重新 Ghost 的时候，很容易忘记拷贝 C:\下面的东西。因此，我建议把 DLL 放置 Rational Robot 本身提供的 DLL 目录下面，既不用担心 DLL 会因为某些原因而忘记，更不需要测试人员自己去放置。

第二，DLL 中的函数哪里声明

说完了 DLL 的存放之后，我们来看看，在 Rational Robot 中如何声明 DLL 中提供的函数。

从 Rational Robot 的 File 菜单中，先创建一个项目头文件，在这个头文件中输入这样的语句：Declare Function ClickBtn_BTN Lib "TestAssistDll" (byval BtnID As String) As Long。ClickBtn_BTN 就是 TestAssistDll 中的一个函数。TestAssistDll 就是放置在 TestDatastore\DefaultTestScriptDatastore\TMS_Scripts\DLL 里边的一个动态连接库文件。因为 DLL 不是 Rational Robot SQABASE32 下面的库文件，请大家注意使用 Lib 代替 BasicLib.

第三，DLL 中的函数在 Rational Robot 测试脚本中的使用

DLL 中的函数在 Rational Robot 中声明之后，Rational Robot 的测试脚本首先使用 include 方法把声明 DLL 函数的项目头文件包含进行。比如，刚才我声明 DLL 函数的项目头文件名称为“AutoTestMagAPI.sbh”，我可以使用这样的语句’$include “AutoTestMagAPI.sbh”.

在此之后，我可以直接使用里边的函数了，比如上面提到的点击按钮的函数。在 Rational Robot 使用 Call ClickBtn_BTN(“btn_ok”),btn_ok 为某个按钮的 ID，它在开发人员创建软件界面的时候就确定下来，我们可以直接从开发人员那里拿到的。

五、 总结

软件测试自动化是软件测试发展的必然趋势，基于非 Windows 标准控件开发的软件可以采用 Rational Robot 外挂 DLL 方式实现自动化测试，在我们的单位已经取得成功。也许只是其中一种解决方案，我只是抛砖引玉，欢迎更多的人加入到这个领域的研究。