HOPE

HOPE COMPUTER COMPANY LTD.

计算机软件测试技术与实例

魏方　编译

本书介绍的软件测试技术新颖且实用，并附有很有参考价值的例子。

北京希望电脑公司

计算机软件测试技术与实例

魏　方　编译

北京希望电脑公司

一九九一年十二月

■ 北京市新闻出版局
准印证号：3065—90065
■ 订购单位：北京8721信箱资料部
■ 邮　　码：100080
■ 电　　话：2562329
■ 传　　真：01—2561057
■ 乘　　车：320、332、302路
车至海淀黄庄下车
■ 办公地点：希望公司大楼一楼
往里走101房间

前　言

本书主要讨论可应用于各种程序的软件测试技术．我们知道，大多数软件产品都是众多的软件设计人员和程序员多年努力的合作成果．因此，最终的产品可能不会达到每个人都能理解的程序．当然，我们可以通过有效的管理和控制方法来保证软件的质量．但是，无论测试系统的方法多么完备、文档多么齐全、结构多么好以及开发计划多么周详，如果单元级的例程没有进行适当地测试，整个系统仍将会失败．可见软件测试技术的重要性．

本书从程序设计的各个方面讨论了软件测试的重要技术并给出了许多很有应用价值的例子．对于从事软件设计和软件质量管理的人来说，本书是一本难得的参考书．此外，书中所阐述的众多技术是极其新颖和实用的．仔细领会，必将获益匪浅．

本书是根据国外有关资料编译而成，由于时间仓促，再加之编译者水平有限，错误之处在所难免，敬请指正．

最后，感谢北京希望电脑公司对本书的出版所给予的大力支持．在此表示衷心的谢意．

目　录

第一章 引言

§1.1 测试的目的

§1.1.1 我们所做的工作

测试至少占据了制做一个工作程序的工作量的一半。程序员们很少喜欢测试，更不喜欢测试设计，如果测试设计和测试的工作量比程序设计和编程的工作量大，则更少有程序员喜欢测试了。这种态度是可以理解的。软件的寿命是短暂的：它是一种无形的东西。从深处讲，大多数人都不相信软件——至少不象信任硬件那样相信它。既然软件是非物质的，那么软件测试似乎是更加看得见摸不着了。当我们完成测试设计时有些测试甚至不需要调试程序。如果测试没有发现错误，那么测试软件所付出的努力似乎是白费了。

更深一步地讲，另一个与测试相关的问题是我们使用测试的原因。测试确实可以捕捉错误。有一种很荒诞的想法认为，如果程序员确实擅长编程，则不应该有错误需要检查。如果我们确实全神贯注地编程，如果每位程序员都使用结构式程序设计、自顶向下的设计方法和判定表，如果程序是按 SQUISH 形式编写的，如果我们有正确的编程方法，则程序就不应该有错误。那么以上所说的想法就可以成立。如果程序有错，那是因为程序员不善于编程，并且如果确实做得不好，则应该感到内疚。因此，测试和测试设计相当于承认程序的编程失败，每测试一次就等于加深一次内疚感。单调的测试正好是对我们错误的惩罚。惩罚什么？惩罚人？内疚什么？是由于没有达到非人的完美而内疚？是由于与别的程序员所想的和所说的无区别而内疚？还是由于没有进行心灵感应或者没有解决哲学家和神学家讨论了近 4000 年的人类通讯问题而内疚？

统计表明，即使很好的编程，每百个表达式中仍然有 1～3 个错误（AKIY71、ALBE76、BOEH75B、ENDR75、RADA81、SHOO75、THAY76、WEIS85B）。当然，如果错误比率为 10%，那么或是需要进一步的编程学习，或是应该受到惩罚和感到内疚。有些人声称他们可以编制无错程序，但这是不现实的。因此测试总是伴随着编程和错误：我的程序有错，你的程序有错，我们大家的都有错，关键是如何防止出错和尽可能早地发现错误，而不是对错误感到内疚。程序员们！舍去你的内疚感！请在测试和调试上花费一半的工作时间！仔细的、有方法和目的地检查错误。建立查错陷阱。尝试无内疚编程的快乐。

§1.1.2 软件的效率和质量

首先考虑成批生产的小型器具的制造。无论设计成本是多少，分摊到总的生产成本中它仅占很小的一部分。一旦投入生产，为了保证出厂以前最后检验的合格，每个生产工艺

都要接受质量控制和元件检验．如果发现瑕疵，则小型器具或其元件或是报废或是重新返工．材料成本、返工成本、报废元件的成本、质量保证和测试成本的总和决定了装配线的生产效率．质量保证成本和制造成本之间是交替影响的．如果花在质量保证上的努力不够，则不合格率将很高，同时成本也将很高．反过来讲，如果检测的效率很高，所有的瑕疵在其发生时即可以查到，则检测的成本将占主要地位，总的成本仍然不可接受．生产过程的设计者试图建立一个测试和质量保证水平，它对应于给定的质量目标而使总的成本降至最低．制造业的测试和质量保证成本对于消费品而言占其成本的 2%，而对于航天飞机、核反应堆和飞机而言则要占其成本的 80%，这些产品的瑕疵要危及生命．

软件的效率和质量之间的关系与制造业的产品大不相同．软件拷贝的"制造"成本是微不足道的：磁带或磁盘和几分钟计算机的机时．此外，软件"制造"的质量保证由检查和及以其它错误检查方法的使用而自动完成．因此，开发占据了软件成本的主要地位．软件的维护与硬件不同，它确实不存在维护的问题，但为增强其功能，软件需要进一步开发、安装和改正其不足．软件成本的最大部分是错误成本：检测错误的成本、修改错误的成本，设计发现错误的测试程序的成本和运行测试程序的成本．小型器具和软件的生产效率之间的主要差别在于，对于硬件质量而言，效率仅是几个决定生产效率因素中的一个，而对于软件而言，质量和效率是几乎等价的．

§1.1.3 测试的目标

做为质量保证的一部分，测试和测试设计也应该把焦点集中在错误预防上．对于不能防止错误的测试和测试设计而言，它们应该能发现由错误产生的症状．最后，测试应该提供消除错误的方法以便错误可以很容易地改正．错误的预防是测试的第一个目的．预防错误比检测错误和改正错误更好，因为如果预防出错，则就不必改动程序．因此，不再需要重新测试以确定修正程序是否有效，也不再有麻烦，不再占用内存，而且预防错误可以不破坏日程表．必须生成一个有用的测试程序的想法可以在错误被编程以前发现它们和消除它们——确实，测试设计的想法可以在软件生成的每个阶段发现和消除错误，从概念到详述、设计、编程、最后到完成．由于这个原因，Dave Gelperlin 和 Bill Hetzel (GELP87) 提倡"测试，然后编程"．理想的测试工作应该在错误预防上获得成功以致于不必进行实际的测试，这是因为在测试设计期间所有的错误已被发现和改正．

不幸的是，我们不能达到这个理想的境界．尽管我们努力，但由于我们是人，所以程序仍然有错误．由于测试不能达到它的第一个目标，错误预防，则它必须达到它的第二个目标，错误的发现，错误不总是十分明显的．错误以偏离期望特性的形式表示出来．测试设计必须为期望目标提供文件，必须详细地编制测试程序，实际测试的结果——所有结果都以错误形式表示．但是知道一个程序不正确并不意味着知道其错误．不同的错误可以有同样的表示，一个错误可以有多种症状．只有使用小的，详细的测试程序才可以分清错误的形式及其原因．

§1.1.4 测试者思维

§1.1.4.1 为什么测试?

测试的目的是什么?下列五个阶段可以刻划出认识方法上的进步:

阶段0——测试和调试没有什么区别,与调试的支持不同,测试无目的.
阶段1——测试的目的是显示软件工作.
阶段2——测试的目的是显示软件不工作.
阶段3——测试的目的是不证明任何事情,仅是为了将已察觉的不工作的风险程度减小到一个可接受的值.
阶段4——测试不是一种行为,它是需要更多的测试工作即可产生低风险的软件的一种认识上的训练.

§1.1.4.2 阶段0的想法

我们把不能区分测试和调试的认识阶段称做"阶段0",这是由于该想法否定测试,这也是为什么不使用一个数字表示该想法的原因.测试和调试之间的差异请参阅本章的第二节第一段.如果阶段0的想法在编程中占主导位置,则那里不可能有有效的测试、质量保证和质量.阶段0的想法在早期的开发中做为编程标准,直到70年代初期仍占支配地位.此时测试做为一种训练形式出现.

对于由昂贵的和不足的计算财力、低成本(相对于硬件)、孤独的程序员、小的计划和免费提供的软件构成的环境而言,阶段0的想法很恰当.今天,这种想法已经是良好的测试和质量软件的最大障碍.由于在阶段0的想法做为规范时许多软件的管理者学习和练习编程,所以它对于今天的测试者和开发人员而言是一个问题——改变人的想法是多么不容易.

§1.1.4.3 阶段1的想法——软件工作

由于阶段1的想法承认测试和调试音的差别,所以它代表着进步.在70年代末期该想法的谬误发现以前,这种想法一直在测试中占统治地位.这个谬误是Myers(MYER79)找到的,它观察到这种想法自我矛盾.它仅使用一次失败的测试显示软件不能工作,则无数次成功的测试都不能证明该软件能够工作.所以阶段1想法的目标是不能达到的.由于显示软件工作的可能性随测试的增加而减少,也就是说,用户测试的次数越多则发现错误的可能性也就越大.因此,阶段1想法是自我矛盾的.因而,如果用户的目标是证实软件工作的高可靠性,则该目标的最好实现途径是测试!尽管这个结论对于神志清醒,有理性的头脑而言是愚蠢的,但它是无意识头脑的人喜欢使用的一种诡辨.

§1.1.4.4 阶段2的想法——软件不工作

当做为测试者，我们的想法进步到阶段2时，我们将与设计者分道扬镳，不是赞同它，而是反对它们。阶段1和阶段2之间想法上的差异可以通过分析簿记员和审计员之间的差别加以说明。簿记员的目标是显示账簿上的平衡，而审计员的目标是显示尽管账面上是平衡的，但簿记员仍然贪污。阶段2想法导致强有力的、揭露式的测试。

当一次失败的测试满足了阶段2的目标时，阶段2的想法也显示出其局限性。测试找出一个错误，程序员改正它，测试设计者设计和执行另一个测试以寻找另一个错误。阶段2的想法导致了无穷无尽，恶魔似的测试。测试走向极端，则永远不会结束，并且其结果是永远不能销售的可信赖的软件。阶段2想法引出的麻烦是我们不知道测试何时停止。

§1.1.4.5 阶段3的想法——减小风险的测试

阶段3想法与承认统计质量控制原理相比很简单。这里说"承认"而不是"完成"是因为在软件上统计质量控制用得不多。对于捕捉和改正错误而言，测试的确改善了产品的质量。如果测试通过，则产品的质量就不会改变，我们的质量感知也不会改变。测试，无论其通过还是失败，都将减少我们关于软件产品的风险感知。测验得越多，就越苛刻，则我们对产品质量的信心就越强。当对质量有足够的信心后，我们就会推出这个软件产品。

§1.1.4.6 阶段4的想法——认识水平

有阶段的想法的程序员其关于什么测试可以和什么测试不可以的知识，再加上软件可以做什么测试的知识将导致软件不需要太多的测试就可以达到低阶段的目标。可测试性做为目标有两个原因。第一个且十分明显的原因是我们想要减少测试工作。第二个可更为重要的原因是可测试的编程玳码比难于测试的程序的错误要少。这两个因素结合在一起产生的效率上的影响是倍增的。如何使程序可测试？学习测试技术的一个主要的原因是回答这个问题。

§1.1.4.7 渐增的目标

以上的目标是渐增的。调试是依靠做为探索错误症状假定原因的工具测试。有许多方法可以分断软件，它与软件的功能要求毫无关系：单独的阶段2想法的测试可能永远不能显示软件应该做什么。分断软件直到可工作性的简单证明出现在面前。这是不现实的，在可接受的风险下使用统计方法做为测试设计的总和，达到一个良好测试的手段是一种好的工作方法。它应该在大的、带少量错误的、健全的产品上使用。最后，只有认识水平是不够的：大多数可测试的软件必须调试，必须工作，必须难以中断。

§1.1.5 测试设计

尽管程序员、测试者、编程管理人员知道程序必须设计和测试，但是，许多人似乎是想不到测试本身必须设计和测试——这个过程与程序设计同样严格和同样控制．经常出现不进行预先的程序要求或结构分析就试图设计测试的情况．这样的测试设计仅是在测试执行前或执行后不要记录的一个偶然的情况．由于它们没有正式设计，所以它们不能准确地重复执行并且没有人相信是否有无错误．在错误改正后，没有人确信重新的测试与原发现错误的测试等同．在调试期间，这种测试是有用的，调试的主要目的是帮助确定错误的位置，但是，这种调试式的测试，无论其如何使用也不能代替经过设计的测试．

编程的测试设计阶段应该可以清楚地识别．代替"设计、编程、桌面检查、测试和调试"，编程过程应该描写成"设计，测试设计、编程、测试编程、程序检查、测试程序检查、测试调试、测试执行、程序调试、测试."在工作量计划表上给测试设计一个明确的位置对于"测试和调试"栏下不定形的工作量而言提供了更多的可见性．这使得当预算少、工作日程紧和预期的目标模糊时不可能对测试设计漠不关心．一旦是这样，则系统将不带错误．

§1.1.6 测试不是万能的

这是一本关于测试技术的书，它仅是我们对付错误的武器．探索和实践（BASI87、FAGA76、MYER78、WEIN65、WHIT87）揭示了其它生成好的软件的途径是可能的和必要的．本书认为，测试仍然是我们最有力的武器，但是很明显其它方法可能也很有效：但是由于测试和检查捕捉和预防不同的错误，所心我们不能用检查代替测试．今天，如果我们想要预防我们可以预防的所有错误并且捕捉到所有不能预防的错误，我们必须回顾、检查、读，做初调，然后测试．我们不知道可以在任何环境下使用的测试方法．经验表明"最佳测试方法"的获得要依据开发环境、应用、计划的大小、语言、规律和修养．其它主要的方法按其效果按照递减次序在下面给出：

检查方法——在这方面中包括初排、桌面检查、正式检查（FAGA76）和程序读．这些方法同测试一样有效，但错误捕捉不完全。

设计风格——这个术语意味着程序员使用风格上的准则定义一个"好"程序的风格。停止使用过时的设计风格，例如为了性能指标而损害质量的"紧密的"程序或"最优化"。反过来，采纳诸如测试能力、公开度和透明程度这样的风格上的目标可以预防错误。

静态分析方法——方法这些方法包括在编辑期间或结合编程时原程序正式分析所可以完成的工作。在早期编辑器中的句法检查是基本的并且做为程序员"测试"的组成部分。编辑器采用覆盖方法。强输入和类型检查消除了一种类型的所有错误。静态分析可以发现不少错误。它是一个广阔的探索和开发区域。例如，异常的数据流通检测（看节五章和第八章）将完全收入编辑器的静态分析中。

语言——原程序语言有助于减少某种错误．语言进一步发展和预防错误已成为语言的

发展的驱动源泉．可是难以理解，由于程序员在新语言中发现新的类型的错误，因此错误率似乎与使用的语言无关。

设计方法学和开发环境——设计方法学（也就是说，使用的开发过程和方法学所处的环境）可以预防多种错误．例如，变化信息的组态控制自动分配可以预防错误，这些错误的来源是由于程序员没有意识到信息的变化。

§1.1.7 杀虫剂的反论和复杂性障碍

假如你是一位在亚拉巴马州种植棉花的农民并且棉铃虫毁坏了你的庄稼．你抵押了农场购买 DDT，撒播在棉田里，杀死了 98%的害虫，救活了棉花。第二年，虽预先撒播了 DDT，但棉铃虫仍然吃掉了庄稼，这是因为去年没有杀死的 2%害虫今年对 DDT 有了抵抗力．现在，则必须再抵押农场去购买 DDT 和 Malathion；则下年棉铃虫对这两种农药都有抵抗力，因此，就不得不再抵押农场，这就是棉铃虫的农药反论，而软件测试也是如此．

第一条法则：杀虫剂反论——用来预防和寻找错误的每种方法都留下少部分相对于这些方法是无效的错误．

这不太坏，因为至少软件逐渐在完善．

第二条法则：复杂性障碍——软件复杂性产生对处理该复杂性能力的限制。

通过消除（预先）简单的错误，允许特性和复杂性的另一种逐步升级，但这次将面临更少的错误，仅保证以前所拥有的可靠性．由于我们都想要额外的铃声、哨子声和性能的相互影响，所以，大家似乎不愿意限制复杂性．因而，我们的用户总是迫使我们遇到复杂性障碍，这种障碍在很大程度上由我们处理更复杂和更少的错误的技术水平决定。

§1.2 一些两分法

1.2.1 测试与调试

测试和调试在同样的标题下经常混为一谈，并且无容置疑的是它们的作用也经常混淆：某些情况，词组"测试和调试"看做一个单字．测试的目的是显示一个程序有错误．调试的目的是寻找导致程序失败的错误并且设计和完成程序的修改以改正错误．通常调试跟在测试的后面，但它们在目的、方法上不同，而更重要的是在物理上的不同：

一、测试以已知条件开始，使用预先定义的程序并且有可预知的结果：不可预见的仅仅是程序是否通过测试．调试从可能地不可知的内部条件开始，结果不可预见，统计性调试除外．

二、测试可以和应该能够计划、设计和定工作日程表．调试的程序和持续时间不受约束．

三、测试是错误或明显的改正的示范．调试是下推理过程。

四、测试证明了一个程序员的失败．调试是程序员证明其正确。

五、当执行时 测试应该力求可预见性、单调、被约束、严格和非人的．调试要求直觉的飞跃、推测、经验和自由．

六、大多数测试在无设计知识的条件下完成．而调试若无详细的设计知识是不可能完成的．

七、测试经常由局外人完成．调试必须由编程者完成．

八、测试有一套健全的理论，它建立了测试是否可以进行的一套理论上的限制．调试仅在最近才受到理论学家的重视——并且至仿为止仅有初步的结果．

九、大多数测试执行和设计可以自动完成．而自动的调试仍然是个梦．

§1.2.2 功能与结构

测试可以按功能或结构观点进行设计，在功能测试中，程序或系统按黑盒子处理．它接受输入并且它的输出按照具体的行为而变化．软件用户应该仅关心其功能和特性，并且程序的完成细节应没问题．功能上的测试占据了用户的观点．

结构上的测试着眼于完成细节．诸如编程风格、控制方法、源语言、数据库设计和编程细节这些事情在结构测试中占主要地位．但结构测试与功能测试的分界是模糊的．好的系统由层构成——从外部到内部．用户仅能看见最外一层，纯功能层．每个内部层很少与系统功能相联系而更多由它的结构约束：某层的结构是其下一层的功能．例如，在线系统的用户不知道系统具有内存分配固化程序．对于用户来讲，这样的程序是结构细节．内存管理程序的设计者依据用于该程序的规范工作．在该层上这个规范是"功能"的定义．内存管理程序使用一个连接——块子程序．内存管理程序的设计者写一个用于连接——块子程序"功能上的"规范，从而定义深一层的结构细节和功能．从更深水平上讲，程序员把操作系统看成一个结构细节，而操作系统的设计者把计算机硬件逻辑当成结构细节处理．

本书的大部分内容讲的是程序模型和可以使用这些模型设计的测试．一个给定的模型和其配属的测试可以首先包括在结构内容中且在以后再用于功能内容中．如何描述模型的最初决择是依据该模型的最有可能用于测试设计的性质完成的．正如不能清楚地区分功能和结构一样，你也不可能固定一个模型的实用性是相对于结构测试还是功能测试的．如果它能帮助你设计有效的测试，则使用模型的可以用于测试工作的内容．

在结构与功能测试的使用上没有争议．两者都有用．两者都有限制，两者各以不同的错误为其目标．在原理上，功能测试可以发现所有错误，但这样做要花费无穷的时间．结构测试时间上有限但不能测到所有错误．从局部上讲，测试技术就是如何在结构和功能测试之间选择．

§1.2.3 设计者与测试者

如果测试完全依赖功能上的规范和独立的完成细节，那么设计者和测试者可以完全分开．反过来，设计一个仅依据程序的结构细节而定的测试计划将需要软件设计者的知识，因此，仅她能设计测试．对设计了解得越多，则越可能消除无用的测试，而它，不管功能上的差异如何，都由同样路径上的同样程序处理；但对设计了解得越多，则越可能产生与

设计者同样的误解。忽略结构是单独的测试者最好的朋友和最坏的敌人。因此，朴实的测试者没有关于什么是或者可能是和将要是程序的设计者永远不认为是的设计测试——以及永远不该认为是的测试。知识是设计者的力量，它带来测试的效率但也盲目地失去功能和特殊情况。由软件设计者设计和执行的测试自然偏向于结构考虑，因此必须接受结构测试的限制。由一个独立的测试者设计和执行的测试无偏向且不能完成。测试的技术是平衡知识和它的相对忽视的偏向性和无效性之间的关系。

在本书中讨论的"测试者"、"测试组成员"、或"测试设计者"都相对于"程序员"和"程序设计者而言，就好象他们是截然不同的人。正如从单元测试到单元一体化，到成分测试和一体化，到系统测试，最后到正常的系统特性测试一样，如果"测试者"和"程序员"是不同的人，则测试的效率更高。在本书中出现的技术可以用于所有测试——从单元到系统。当技术用在系统测试，设计者和测试者或许是不同的人，但当技术用在单元测试，则测试者和程序员合并成一个人，它有时是程序员，有时是测试者。

你肯定是一位推定的精神病患者。十分清楚作为程序员和测试者角色上的差异。你的测试者角色必须是多疑的、坚定的、怀有敌意的、着迷于毁坏、彻底毁坏程序员的软件。你的程序员角色必须试图在时间和预算允许的性况下用可能的最简单的和最清楚的方法完成工作。有时，通过洞察编程问题达到这个目标，它减少复杂性和劳动并且几乎是完全正确的。记住，当本书把"测试设计者"和"程序员"看做单独的人时，它们分开的程度依据测试水平和技术应用的内容。

§1.2.4 模块化与效率

测试和系统都可以是模块化的。模块是系统的一个分离的、良好设计的、小的组成部分。组成越小，它就越容易理解；但是，每个部分与其它部分有界面联接，所有界面都是混乱的根源。组成部分越小，则界面错误出现的可能性越大。大的组成部分减少了内部的界面但具有难于了解或不可能了解的、复杂的内部逻辑。软件设计技术部分是设定组成部分的尺寸和平衡相对于界面复杂性的内部复杂性以达到全部复杂性最低点的边界。

测试可以且也应该构成模块式的组成。小的、独立的测试情况有可重复性的优点。如果测试发现一个错误，则只有小的测试，而不必由几百个相互独立的测试构成的大的组成，需要重复测试以确认测试设计错误。同样，如果测试有一个错误，仅该测试需要改变而不必改变整个测试计划。但是，微小的测试需要单独的设定且每个这样的设定（数据，输入）都可能有错。正如使用系统设计一样，测试技术通过设定大量的小测试的测试组以便在不降低效率的情况下使测试设计、测试调试和测试执行的工作量最小。

§1.2.5 小与大

我经常写几百行小的分解程序，它一旦用过就丢弃不用。我确实使用正式的测试技术、质量保证和其它我如此激烈拥护的技术吗？当然不，我不是伪君子。在同样情况下，我的所做所为与每个人一样：我设计、编程、我测试少数几种情况、调试、重新设计、重新编程等等。我在30年前就已经这样工作了，我可以摆脱这样的（未开发的）活动，这

是由于我正在编程是为了小的、智能的、可忘记的、用户全体——我自己的程序。它是最小的程序且依据直观的手段和非正规性，它的效率最高。

让我们上升到一个大软件包的水平。我仍然是唯一的程序员和用户，但现在，软件包有30个组成，每个平均有750个语句，它开发5年了，现在我必须生成和维持一个数据库且测试全部单元。但我将使用自己的词表示它并且不操心所有测试或练习正式的组态控制。

测试者可以从这里推知或者利用已有的经验。大规模的编程（DERE76）意味着由不同人写成的许多小成分构成的结构程序。小规模编程是我们为自己工作所编写程序或在大学编程课上完成的家庭作业。程序大小带来了非线性规模影响。它在今天并没完全了解。质量的变化由大小和测试方法及质量准则决定。最初的例子是范围概念——测试完整程度的测量。不必担心这些术语准确表达的意思。对于单元测试而言，其100%完整度是必须的，但是对于大的软件包而言我们不做这个要求。大多数系统为75%-85%。千万行程序的大系统可能低至50%。

§1.2.6 编程者与客户

大多数软件用同样的结构写成和使用。不幸的是，这种情景的责任不清而不诚实。今天，许多组成结构承认独立软件开发和操作的长处，这是由于它将导致更好的软件、更好的社会性、以及更好的测试。独立的软件开发并不意味着所有的软件都应该从软件公司购买，但软件开发实体与购买软件的实体应该分开以确保各方的责任清晰。我听到过这种情况，软件开发组和操作组在同一公司内，而谈判和签署正式合同同另一方———同出席的律师签订。如果编程者同买者没有分开，则就没有责任。如果无责任，那么软件质量动力消失且不做恰当的调试。

正如程序员和测试者可以合并为一，编程者和购买者也可以是同一实体，同上所述，在软件的开发性格中有几种不同的人可以分开或合并：

一、编程者，它为购买者设计和负责软件开发。
二、购买者，它购买系统以从提供给使用者的服务中得到希望的效益。
三、使用者，系统的长期的受益者和受害者。使用者的权利由测试者保证。
四、测试者，它致力于对编程者的破坏和操作。
五、操作者，它必须与编程者的错误、购买者难懂的规范、测试者的遗漏和使用者的抱怨一起工作。

§1.3 用于测试的模型

§1.3.1 设计

测试既可以用于子程序，也可以用在由上百万个语句构成的系统中。原始模型的系统允许探索测试的各个方面，而没有与测试无关但影响某些大的设计的复杂性。它是中等规