第 4 代白盒测试方法介绍—理论篇

2005-12-15

拟制：Wayne Chan	2005-12-15
审核：	2005-01-01
审核：	2005-01-01
批准：	2005-01-01

文档修改说明：

序号	修改描述	时间	责作人	版本
1	完成初稿	2005-12-19	wayne	1.0
2	将举例脚本适配至 VcTester V5.2 版本	2011-7-12	wayne	1.1

文档分发列表：

序号	角色	文档接收者	分发时间	说明

目 录

关键词： 白盒测试 第 4 代 测试方法 4GWM 在线测试 持续测试 灰盒 脚本驱动 脚本桩

摘 要： 本文是第 4 代白盒测试方法的理论介绍，描述 3 个关键领域内 9 项关键特征的概念与固有特征。同时介绍白盒测试发展历程，对比说明第 4 代白盒测试方法与以往测试方法的异同及优化要素。

缩略语：

4GWM：The 4th Generation White-box-testing Methodology，第 4 代白盒测试方法
XP：Extreme Programming，极限编程
TDD：Test Driven Development，测试驱动开发
IID：Incremental and Iterative Development，渐增迭代开发
CSE：Common Script Engine，通用脚本引擎（一种近似于 python 的脚本语言）
PCO：Points of Control and Observation，观察控制点
TDF：Test Design First，测试设计先行
MCDC：Modified Condition/Decision Coverage

1 背景

1.1 白盒测试的范围

白盒测试是软件测试体系中一个分支，测试关注对象是一行行可见代码，如果代码不可见就不是白盒，是黑盒测试了。白盒测试也通常被认为是单元测试与集成测试的统称，但这个概念是相对的，与当前项目遵循的研发流程有关，某些流程把白盒测试划分为单元测试与集成测试，而另一些流程，把白盒测试划分为模块单元测试、模块系统测试、多模块集成测试，还有一些流程把单元测试与集成测试混为一体，统称为持续集成测试。

随着测试技术的发展，白盒测试的概念也在发生变化，比如，本文提倡一种介于白盒与黑盒之间的灰盒操作模式，针对被测对象同样是可见源码，这时，白盒测试不只是白盒了。尽管如果此，我们仍遵循大家习惯的思维方式——把本文倡导的测试方法仍冠名为：第 4 代白盒测试方法（4GWM，The 4th Generation White-box-testing Methodology）。

本文讨论白盒测试方法，范围限定在功能测试之前，针对源码行的所有测试，即，被测对象是看得到的功能源码，每个测试者必须先获得源码才能实施测试。

1.2 第 1 代与第 2 代白盒测试

说到第 4 代白盒测试方法，就不能不回顾前几代方法。在测试发展初期，测试工具很不成熟，人们通常以单步调试代替测试，或采用 assert 断言、print 语句等简单方式的组织测试体系，即我们所谓的第 1 代白盒测试，这一时期的测试是半手工的，没实现自动化，测试效果也严重依赖测试者（或者调试者）的个人能力，缺少统一规范的评判标准。

当然，调试算不算测试在业界尚存争议，单论调试的目的（为了定位问题）与操作方式（过程不可重复），不应把调试看作测试，但调试确能发现软件 BUG，显然这也是一种测试手段。本文暂不评判调试用作测试手段是否合理，但有必要先确定调试是测试的某种形式，把它看作特定历史阶段或特定场景下的产物。特定历史阶段大家比较容易理解，调试伴随编程语言是天生的，测试工具却是后天形成，开发人员总喜欢认调试器当亲妈，测试工具则是爱管不管的后妈。特定场景是什么？比如，某种生僻的 RTOS 平台根本找不到对应测

试工具，怎么办？拿调试做测试是无奈之中的必然。这里，我们不否认调试也是一种测试，在此基础上再优化其操作过程，使调试能更好的服务于测试（下文介绍“灰盒调测”还有进一步论述）。

第 1 代白盒测试方法存在严重缺陷，主要有：测试过程难以重用，成功经验无法拷贝，测试结果也难以评估并用于改进，这些对于团队运作是非常致命的。

到第 2 代白盒测试，上述主要缺陷得到克服，将测试操作改用一种形式化语言（通常称为测试脚本）来表述，脚本可以组合成用例，用例可组合成测试集，用例与测试集再统一到测试工程中管理，把测试脚本保存到文件，重用问题解决了。另外，代码覆盖率功能使测试结果可以评估，能直观的看到哪些代码或分支未被覆盖，然后有针对性的增加测试设计。目前市面上有大量商用工具，如 RTRT、CodeTest、Visual Tester、C++ Tester 等都属于这第 2 代白盒测试工具。

1.3 第 3 代白盒测试方法

按理说，第 2 代白盒测试工具已经很完善了，那第 3 代又是什么？

软件测试是一门复杂科学，支持自动测试与覆盖率评估后不见得就能成功实施白盒测试，尤其重要的是，第 2 代白盒测试解决了重复测试问题，但没解决持续测试问题。简单来说，重复测试使测试操作能以规范格式记录，当被测对象没变化（或变化很少）时，测试用例是可重用的，但如果源码大幅调整（甚至重构），或者按迭代模式不停追加新功能时，如何维持用例同步增长，并与源码一起同步更新，已经不是简单的增强用例复用能力就能解决的。因为代码更新与用例更新交织进行，测试用例与被测源码一样对等的成为日常工作对象，必然促使原有工作模式与测试方法产生变革，概括而言，白盒测试过程要从一次测试模式过渡到持续测试模式。

第 3 代白盒测试工具以 xUnit 为代表，包括 JUnit、DUnit、CppUnit 等，当然，我们列举 xUnit 工具，并不说这些第 3 代工具就比第 2 代工具要好。事实上，目前 xUnit 工具在功能上普遍赶不上第 2 代商用工具，许多 xUnit 工具甚至连基本的覆盖率都支持不了，况且，xUnit 使用被测代码的编程语言写用例，普遍效率低下。这里，我们区别第 2 代方法与第 3 代方法，主要是测试理念上差别，而不以工具差别为基准，因为工具配套跟进还与诸多现实因素相关，是另一层面话题。

1.4 第 4 代白盒测试方法的产生背景

xUnit 是 XP 实践的重要支撑工具，XP 作为一种软件开发方法论，总体虽然敏捷，但很脆弱，它对程序员非常友好，但对组织不是。以 xUnit 为代表的 XP 测试实践同样表现出这一特质，据已有案例分析，XP 持续集成在 java 项目中成功的很多，C++有一些， C 语言项目就很少了，为什么编程语言对持续集成的影响如此深远？

第 4 代白盒测试尝试解决软件测试的深层次矛盾：测试的投入产出比问题。大家知道，研发资源总是有限的，你可以把测试人员与开发人员的比例配到 1:1，也可以配到 2:1，甚至 5:1，但你做不到 10:1、100:1，如果你有钱，也有人，完全可以按 100:1 或更高比例配置，这时所有测试瓶颈都没了，你可以让测试人员边喝咖啡边干活，因为每新写 1 行代码总有人编出 100 行脚本测试它，还怕产品不稳定吗？不过，疯子才会这么做，比尔盖兹有的是钱，一年捐款十多亿美金，但不见得微软旗下产品就经常让测试人员比开发人员多出一倍。我的意思是，测试资源必然是受限的，这个前提下我们才讨论第 1 代、第 2 代白盒测试向第 3 代、第 4 代演化的必要性。基于同样原理的 xUnit 工具，针对不同开发语言效果截然不同，这说明什么？说明这种实践的瓶颈仍在投入产出比上，也就是上面所说的 1:1 效果，

还是 2:1，抑或是 5:1 效果。

高效平台下的高效工具可以大幅提高测试效率，测试投入与开发投入之比小于 1:1 就能保证测试质量，项目就成功了，而低效平台下的低效工具，必然要投入更多测试资源（比方 5:1）才能保证效果，拐点就在这儿，哪个公司禁得起 5:1 的测试投入？！从这个意上说，推出第 4 代白盒测试方法意义重大，我们要尝试解决决定项目成败的拐点问题。

事先申明一下，下文涉及持续集成与测试先行（或称测试驱动开发，TDD）实践，虽然这两者都是 XP 的重要组成部分，但我们无意宣扬 XP，事实上，真正能适应 XP 的项目范围并不宽，跳过需求与预设计直接启动项目的做法，足以让客户敬而生畏，把文档丢给狮子，那是无政府主义散兵游勇行径。不过，XP 确有许多闪闪发光的实践，持续集成只要运用恰当还是不错的模式，测试先行的理念也不赖，只要不过度实施就好。

2 什么是第 4 代白盒测试方法

第 4 代白盒测试方法（4GWM）针对前几代测试方法不足提出，许多理念仍继承第 2 代与第 3 代测试方法。下表简要的列出第 1 代到第 4 代白盒方法的主要差别：

	是否评估测试效果	**是否自动测试**	**是否持续测试**	**是否调测一体**
第 1 代白盒测试方法	否	否	否	否
第 2 代白盒测试方法	是	是	否	否
第 3 代白盒测试方法	是	是	是	否
第 4 代白盒测试方法	是	是	是	是

上表中，“是否评估测试效果”指是否有覆盖率或其它评估测试效果的指标，“是否自动测试”指是否形式化描述测试操作并将它用于再次测试，“是否持续测试”指是否以按持续集成的模式开展测试，“是否调测一体”指是否将测试设计高效的融入产品编码与调试的日常实践之中。

第 2 代白盒测试与第 3 代的分水岭在于“是否持续集成”，或许您会说，我的项目也是经常出版本，反复追加测试用例的呀，请注意，这是两个概念，Joel 测试——改进代码的 12 个步骤中有一条：“编写新代码之前先修复故障吗？”，先修复故障是质量优先的项目，否则进度优先，这是两种完全不同的行事风格，前者时时测试，始终每写一两个函数就补全相关测试用例，测试实践是融入开发全过程的，而后者依时间表行事，测试仅是特定阶段里的任务。

对了，测试方法怎么跟软件开发方法扯上了？因为测试不是孤立的，测试是否有效强烈依赖于软件工程方法，就像早期的开发语言，只有 assert 语句与测试相关，发展到现有的 C#，单元测试框架也是该语言的固有组件了。测试脚本也是一种产品代码，测试方法实际与软件开发方法密不可分的，这在第 3 代与第 4 代白盒测试中体现得很充分。

第 4 代白盒测试方法相对第 3 代方法，增加了将测试过程（包括测试设计、执行与改进）高效的融入开发全过程，这里，“高效的”是关键词，那如何才算高效呢？我们先简单了解 4GWM 在 3 个关键领域的 9 项关键特征，如下：

A. 第一关键域：在线测试
   1、 在线测试驱动
   2、 在线脚本桩

3、 在线测试用例设计、运行，及评估改进

B. 第二关键域：灰盒调测
   4、 基于调用接口
   5、 调试即测试
   6、 集编码、调试、测试于一体

C. 第三关键域：持续测试
   7、 测试设计先行
   8、 持续保障信心
   9、 重构测试设计

3 为什么持续集成

为什么要持续集成？这个问题太重要了，我们专门拎出来讲，请大家先不急于跳过本章去看 4GWM 的 9 个关键特征怎么定义的。

3.1 JOEL 测试

Joel 是个怪人，当然他不认识我，我拜读他的 Blog 才知道他的。这家伙总有许多稀奇古怪的思想在小脑瓜里蹦达，他是“经常放猫出来闲逛”的人。科学研究表明，人的大脑只占体重 2%，却消耗 20%的能量，当大脑思考问题时，释放出的能量等同于夜间放一只猫出来活动。他的“Joel 说软件”专栏（www.joelonsoftware.com）很火，有一些不乏真知灼见。比如，Joel 测试——改进代码的 12 个步骤：

1、 有版本控制机制吗？
2、 能一步完成编译链接吗？
3、 每天都做编译吗？
4、 使用缺陷跟踪库吗？
5、 编写新代码之前先修复缺陷吗？
6、 有最新的进度表吗？
7、 有规格说明书吗？
8、 程序员拥有安静的工作环境吗？
9、 你用到了你资金能力内可买到的最好工具吗？
10、 有测试人员吗？
11、 要求新聘人员在面试时编写代码吗？
12、 进行走廊可用性测试吗？

每个问题可以回答“是”或者“否”，答“是”则加 1 分，得 12 分是完美，11 分勉强接受，10 分以下问题就大了，大家有兴趣看看你所在的组织能打多少分。

有测试人员吗？干嘛这么问，没测试人员还叫软件公司吗？这个问题并不可笑，还真有不少公司从未配置过专职测试人员。某白炽灯生产商在使用说明中特意声称，灯泡不能往嘴里塞，否则会出严重医学事故，说明书中还郑重其事的介绍灯泡不慎入口后，如何求医，如何抹润滑剂，如何左转 90 度右转 90 度慢慢取出来。有人觉得滑稽，谁白痴有事没事拿灯泡往嘴里送？即使放嘴里了也不用这么麻烦吧？非得试试，结果如何？怎么也拿不出来了，只得嘴里叼个灯泡打的上医院，最后，医生按照说明书费老劲才将那玩意卸下。所以，不要轻易否定前人经验，早有人试过了。

看看上面 12 个步骤，前 5 步活脱脱在讲如何实施持续集成，若进一步了解其内容，大家

不妨浏览 Joel 的 Blog 原文。

3.2 持续集成不是 XP 专有实践

持续集成属于 IID（持续迭代开发）方法学，在测试上，就现实而论是以 xUnit 实践为代表，持续集成概念被 XP 刻上深深烙印，但它确非 XP 专有实践。

早在 20 世纪 60 年代 IBM 的 Federal Systems Division 就开始应用 IID 开发模式了，源于 IBM 的集成产品开发流程（IPD）相对 CMM，有个显著特征，它支持渐增迭代开发，虽然迭代频度比不上微软每日构造，但其理念仍是持续的迭代开发。有意思的是，IPD 流程在华为公司本土化后，发展出“版本火车”理论，有点类似于 Scrum 实践了，版本火车不仅让产品（通常是大产品）版本发布更加规范有序（因为火车总是定点出发的），也推动研发以更快频度推陈出新。

但目前持续集成仍在有限范围能成功应用，微软无疑是个样板，毕竟纯软件产品容易实施每日构造，还有不少实践 XP 的项目，持续集成也运用得很成功。所以，就整体而言，持续集成能否成功，已经不是方法论问题，更多是 IT 工具如何支撑的问题。

3.3 为什么持续集成

我们看一个实际案例，某通信产品在 V1 版本编码完成时，进行过规范的单元测试活动，之后 V2、V3 要不断增加功能、修改功能，就放弃单元测试了，当 V3 最后市场交付时统计发现，相对 V1 版本，代码修改量已达到 40%。QA 从其中两个模块随机抽取 100 个问题单做缺陷分析，结果发现：第一个模块有 50%的问题是在 V1 版本单元测试结束后引入的，而另一模块也有 30%问题是单元测试后引入的。

也就是说，在第一次完整单元测试之后，代码修改了 40%，也因此产生了 40%的问题，由于增量白盒测试难以实施，这些问题都被遗留到后期功能测试中才发现。单元测试没能持续开展，带来后果是：发现问题不彻底，付出代价也更高。

上述模式在业界还普遍存在，我们称为一次测试，与持续测试不同，一次测试的测试设计只做一次，用例仍可重复拿来跑，因为测试脚本与源码不同步，用例维护是间歇进行的，或者干脆不维护。注意，一次测试与持续测试的差别不在于用例是否可重用，而在于测试设计的持续性。

许多企业做不到持续测试，其主要原因不是不想做，第一次测试都认真做了，追加代码或修改代码当然也要做测试，做不了是因为操作上存在困难。持续测试是需要一开始就规划，测试工具要配套跟进才能顺利实施的，对于老产品，代码修修补补，无论一次测试还是持续测试都很难做得好。

引入持续测试，不仅以更低代价发现更多问题，更重的是，它体现了一个组织在测试理念上有质的飞跃。一次测试是一种被动测试，开发人员受制于组织纪律（或主管、QA 等压力）才去做，而持续测试是主动测试，大家在测试中尝到甜头，从原先不自觉状态，过渡到自发、自觉的时时做测试。这两种情形无疑有天壤之别，前面提到的 Joel 测试 12 步骤，实际上是微软实践，与持续集成相关的有 5 条，足见它的重要性，是否引入持续集成，以及实施的效果如何，实际反映了一流公司与二流公司的差距。

4 第 4 代白盒测试方法的关键特征

白盒测试是一项实践性很强的技术，我们讲第 4 代白盒测试方法，离不开相关测试实践，尤其是测试工具支撑。本文的上篇先从理论上介绍什么是 4GWM，下篇则结合具体测试工具介绍 4GWM 的典型实践。

4.1 在线测试

4GWM 第一个关键域是在线测试，包括 3 个关键特征：

- 在线测试驱动
- 在线脚本桩
- 在线测试用例设计、运行，及评估改进

一次白盒测试中（即一个用例中）我们关注被测单元功能是否实现，被测单元作为整体，在特定环境下运行（比如某些全局变量取特定值、某些依赖线程或任务已启动等），具有特定的输入输出，这几项都属于“测试驱动”。另外，被测单元若能正确运行，还依赖它调用的子函数是否提供正常功能，这些子函数我们称为“测试桩”。分层结构如下图：

在三层实体中，被测单元是测试关注对象，要求尽可能真实，我们设法维持其原状，测试驱动与测试桩可以模拟（或叫仿真），允许存在一定失真，但要求尽可能高效，否则测试产出的拐点问题解决不了。

4.1.1 脚本驱动与脚本桩

先回答一个基础问题，编写测试用例应优先采用脚本语言，而不与被测代码使用同一的语言，为什么？

还是应为软件测试的深层次问题——投入产出比，如果被测编程语言的抽象度较低、封装性差，用起来就很麻烦。比如拿 C 或 C++写测试用例，得处处小心内存操作，要正常申请释放、注意不越界，时常关心使用变量是否安全、是否已初始化等。也许有人说，不对，CppUnit 中拿 C++测 C++，我用得很爽呀？噢，没错，我得先恭喜这位老兄，安于现状不失为一种好品质。

我们设想一下，编写一万行 C++代码，你要写多行代码测试它，一千行？两千行？不对，是一万行，按业界普遍规律，测试代码行至少要与被测代码行数相当才见效果，测试代码要不要调试？当然要调，天哪，算出来的了，测试投入至少是开发投入的三、四倍才做得下来（后期还有功能测试、性能测试、兼容性测试等等，还要占用大量精力），这样的项目是不是处在能否成功的拐点上？所以，如果您还在用 C、C++等过程语言写用例，请尽快换到脚本语言，如 python、ruby、CSE 等，用脚本语言能让你编写用例的效率提高 3 到 5 倍。

用脚本编写用例，意味着测试驱动与测试桩仿真也用脚本语言。我们看一下 VcTester 工具使用的测试脚本，假定被测对象是 C 代码的冒泡排序算法：

```
void BubbleSort(OBJ_DATA_PTR *ObjList, int iMax)
{
  int i,j,exchanged;
  OBJ_DATA *tmp;

  for (i = 0; i < iMax; i++) // maximum loop iMax times
  {
      exchanged = 0;
      for (j = iMax-1; j >= i; j--)
      {
          if ( ObjCompare(ObjList[j+1],ObjList[j]) < 0 )
          {   // exchange the record
              tmp = ObjList[j+1];
              ObjList[j+1] = ObjList[j];
              ObjList[j] = tmp;
              exchanged = 1;
          }
      }
      if ( !exchanged ) return;
  }
}
```

排序函数（BubbleSort）中调用了对象比较函数（ObjCompare(Obj1,Obj2)），假定当前测试对象是 BubbleSort 函数，我们编写测试用例如下：

```
func StubFunc():
  if Obj1->Data < Obj2->Data:
    return -1;
  end else return 1;
end;

vd.ObjCompare.stub(StubFunc);  # 打脚本桩
vd.BubbleSort(vd.gList,6);     # 发起测试
assert(vd.gList[0]->Data <= vd.gList[1]->Data); # 检查测试结果
vd.ObjCompare.stub(dummy);     # 清除脚本桩
```

脚本驱动是指将被测系统的全局变量与全局函数映射到脚本系统的 vd 模块，然后使用脚本读写 C 语言变量，调用 C 语言函数。在 VcTester 中，C 语句的全局变量与函数映射到脚本的 vd 集合下，如上面脚本使用“`vd.gList`”读取 C 变量，使用“`vd.BubbleSort()`”调用 C 函数。

脚本桩是指定义一个脚本函数，然后让这个脚本函数代替某个 C 函数，打脚本桩是为了让一段脚本化测试逻辑，在动态执行中，代替被测系统中的桩函数。因为测试中我们经常要让某些子函数返回特定值，使被测函数的特定路径能被覆盖。上面例子定义了一个脚本桩函数 StubFunc，拿这个脚本函数模拟对象比较功能，通过打桩替换 C 函数 ObjCompare。

4.1.2 在线测试逻辑更新

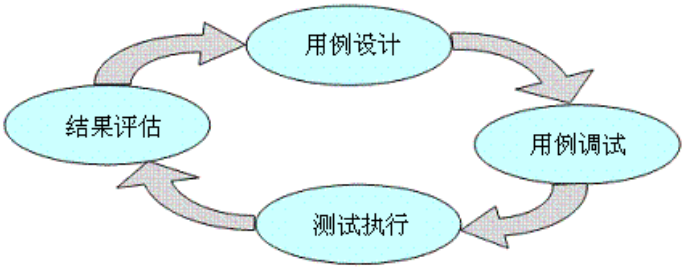
4GWM 引入脚本驱动与脚本桩，不只是提高测试设计效率，还以此保障在线测试。所谓在线测试，是指被测程序启动后，用例在线设计、调试、运行，运行结果在线查看的测试方法。因为所有测试操作都在线进行，测试用例不必编译链接，被测程序也不用复位重起，被测环境（被测系统的变量、函数等属性）在线可查看，所以该测试模式非常高效，另外，各测试步骤所见即所得，人性化的操作过程很容易被广大开发人员接受。

脚本语言具有在线更新功能，比如定义一个脚本函数，调用一次后，发现某个地方处理不对，于是重写这个函数，然后在线的更新这个函数定义。编译语言做不到这一点，修改代

码后必须重新编译链接，程序要复位重起，脚本语言省去了这些繁琐过程。比如，在 GUI 界面编写测试用例，定义测试桩函数，然后选择待执行的脚本区块，按一个快捷键，指定范围的脚本就执行，相关脚本函数定义立即被更新，脚本执行后的测试结果也立即打印输出。

4.1.3 拉通测试小循环

测试用例设计、调试、执行，及评估改进是一个闭环迭代，如下图：



测试结果评估主要是覆盖率指标，包括：语句覆盖、分支覆盖、组合条件覆盖等，结果评估也是在线进行的，用例执行后，随即在线查阅覆盖率情况，针对未覆盖部分再增加用例。

当上图 4 个步骤都能在线操作后，测试小循环就拉通了，4GWM 的第一个关键域（在线测试）的目的就在这儿，拉通测试小循环，是大幅度提高测试工作效率的第一环节。接下来通过灰盒调测，拉通开发大循环是提高效率的第二环节。

4.2 灰盒调测

4GWM 第二个关键域是灰盒调测，包括 3 个关键特征：

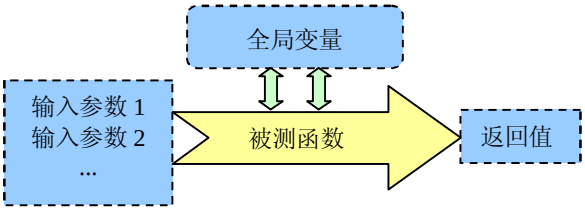
- 基于调用接口
- 调试即测试
- 集编码、调试、测试于一体

4.2.1 白盒测试的粒度

白盒测试关注被测函数的功能表现，要关注到什么程度，在不同的测试实践与测试工具中要求各不同。我们可以简单的分为 3 个级别，一是源码行级别，二是函数调用级别，三是组件接口级别。

源码行级别具有调试特征，可以关注到函数内局部变量，当测试停留于该级别会显得过于细碎，因为结构化程序开发总是以函数为单位逐级划分功能的，函数内的代码稳定性差，变量定义经常变化，过程处理也经常调整。组件接口级别的测试对象仅关注到组件接口，如 Corba 接口、控件调用接口、消息队列接口等，这一级别的白盒测试无疑偏于粗放。

4GWM 规定的白盒测试关注粒度是函数调用接口，即，测试设计只关心函数的输入、输出，及该函数运行中对全局变量的影响，遵循如下原型：



设计测试用例，先通过脚本构造被测函数的输入参数，修改特定全局变量，使被测函数处于某特定运行环境下，这两步属于测试驱动。然后调用被测函数，最后判断测试结果，因为运行被测函数可能影响输入参数、全局变量与返回值，所以判断用例是否运行通过，观察对象也是这三者。在用例设计过程中，我们并不关心函数内局部变量如何声明，也不关心函数内逻辑过程如何处理，只关心被测对象的输入与输出，这是一种典型的黑盒思维模式。

准确来说，4GWM 是一种灰盒测试方法，尽管操作方式是黑盒的，但测试设计是白盒的，因为看得见源码，测试设计可以有针对性的进行，测试过程评估也是白盒的，运行一遍用例后，查看哪些代码行有没跑到，再有针对性补充用例。所以，我们从整体来看，4GWM 是介于黑盒与白盒之间的灰盒测试。

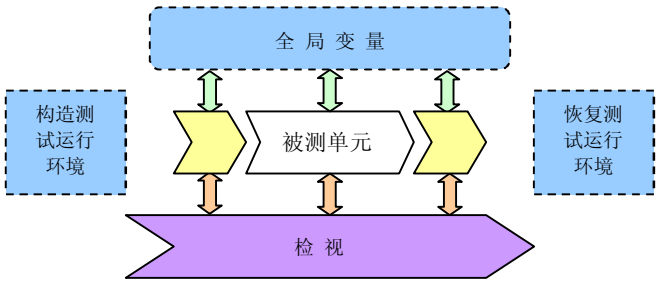
根据已有实践推断，上述灰盒模式关注的测试粒度是恰如其分的，既避开了调试操作的随意性，也使测试用例建立在较稳健的基础之上，只要函数调用接口没变，局部变量改了或逻辑过程调整了，就不会影响已有用例。同时，黑盒操作方式附带白盒分析模式，保障了 4GWM 具有高效、便捷的特性。

4.2.2 检视器

检视器（Inspector）是 4GWM 推荐的测试辅助工具，它介于测试器（Tester）与调试器（Debugger）之间，是一种能够提供脚本化控制的粗粒度的调试器。使用检视器有助于把无规则的调试过程转化为规范的测试过程。

检视器有两种运行模式：断点调试模式与测试模式。前者在断点条件满足时进入单步跟踪状态，后者在断点上附加特定脚本语句（比如修改变量、检查变量值等），当断点条件满足附加语句即自动执行，此时断点仅作为一个观察控制点（Points of Control and Observation，PCO）存在，不用作交互调试目的。

一次典型的检视过程如下图所示：



首先在被测函数上设置断点，接着用脚本构造调试环境，包括修改变量、设置脚本桩等，然后发起测试，在断点触发后的单步跟踪状态，观察各个变量值是否预期，还可以修改变量使被测函数中特定分支能够执行。最后在调试完成时，可以将当前调试操作，包括设置断点、检查变量值是否预期、修改变量等，自动转化为测试脚本。

上述检视操作向自动脚本转换还解决测试数据构造问题，尤其在复杂系统中，构造测试数

据比较麻烦，比如通信协议的消息包数据，创建消息后要填写数十，甚至数百个字段的值。检视操作可以在函数调用链中插入一段脚本代码，比如被测代码先调用一个初始化协议消息的函数，得到正确消息包后传递给被测函数，我们通过插入脚本，在被测函数运行之前修改传入消息包的特定字段，从而实现特定路径的覆盖测试。采用该方法设计用例是非常廉价的，直接重用被测系统的局部功能，免去了繁重的测试驱动构造工作。

检视过程类似于调试，主要差别如下：

1. 检视器断点只在函数入口设置，调试器可以在任意语句设断点。
2. 检视既可以在 IDE 界面手工操作，也可以通过写脚本控制，调试器一般只支持手工操作。
3. 检视器在断点状态下可以运行任意合法的测试脚本，调试器无此功能。

由于检视器与编程语言自带的调试器实现原理不同，一般情况下两者可以同时使用，可同时设置检视断点与调试断点。

4.2.3 调试就是测试

调试为了定位问题，测试是为了发现问题，两者虽不能互相替换，但当测试手段趋于丰富，测试工具也能越来多的承担调试职责。让测试工具承担部分调试功能，可在如下方面获益：

1. 调试与测试共享运行环境

   被测代码片断是在特定环境下运行的，无论调试还是测试，都得先构造运行环境，比如准备特定的数据、修改状态变量、启动特定线程或任务。借助测试工具在线构造测试驱动与测试桩，调试环境能便利的搭建起来，而且，构造运行环境的脚本能直接在相关测试用例中重用。

2. 将不可重复的调试转化为可重复的测试

   调试过程具有随意性与不可重复性，在哪儿设断点、如何看变量、如何单步跟踪都因人而异。调试的操作过程难被重用，不像测试用例，以形式化脚本记录操作过程，想怎么重复就怎么重复，上节介绍的检视器就是一种可重复的调试器。

   操作自动重复是提高工作效率的基本途径，不必强求全过程重复，片断可重复就能大幅提高效率了。

3. 测试设计可以很好的重用被测系统中局部功能

   如上一节举例，直接调用被测系统的消息构造函数，能避开繁重的协议消息仿真工作。

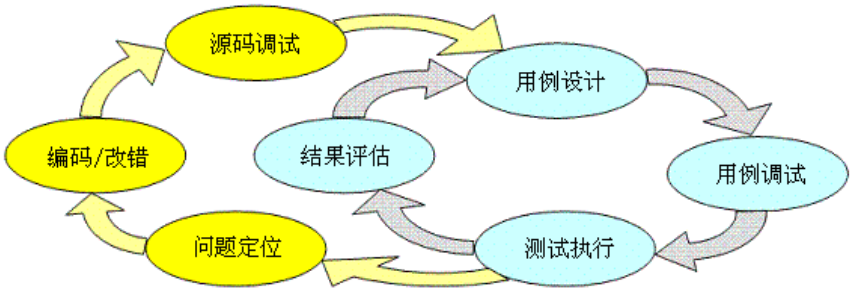
4. 解决脚本调试与源码调试的交叉影响问题

   实践证明，白盒测试的大部分时间消耗在脚本编写与调试中，调试好的用例，执行几乎不要时间（即使要时间，挪到晚上让它自己自动跑好了）。测试脚本调试与源码调试是交叉进行的，单元测试中的源码与测试脚本都不稳定，通常我们让脚本发起测试，须同时跟踪脚本与源码，查看执行结果正不正确。如果这两者调试过程是分离的，调源码时不能看脚本，或调脚本时不能看被测变量，其操作过程必然非常痛苦。

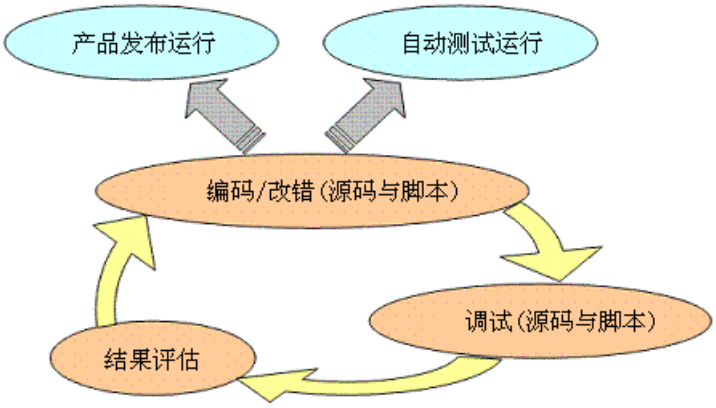
   当测试承担起调试职责，两者合二为一，交叉影响的问题即自动解决。实事上，大家把测试当测试、调试当调试，很大程度上是因为没把测试脚本也看作产品代码，不把它当成产品固有部件，如果观念转变过来了，测试脚本也是代码，调试脚本就是调试代码，两者本应合二为一的。当然，还存在工具的问题，缺少好工

具，将两者强扭一起最终仍会不欢而散。

4GWM 尝试让测试工具承担起 90%的调试工作，完全替换并非必要。如果测试工具能承担大部分调试，开发大循环就能拉通了。下图是开发与测试尚未拉通，是孤立两个过程的情况：



拉通开发大循环后，测试不再是独立的闭环过程，如下图：



测试设计（即写脚本）与产品设计（即编码）融为一体，调试脚本与源码成为开发人员主要日常工作。上图的结果评估，对于测试脚本是覆盖率，对于产品源码是其运行表现（其结果可能预期，也可能出差错了），评估这两者，再补充用例及完善源码，之后进入下一轮迭代循环。

调试通过的脚本打包到测试工程，就是能够支持每日构建的用例库；测试通过的源码经 release 发布，就是在市场上能提供预期功能的正式产品。

4.2.4 编码、调试、测试集成平台

4GWM 在方法论上要求大家把测试脚本也看成产品代码，以黑盒调测代替大部分单步调试，但方法论能否顺利被实践支持，还严重依赖于测试工具的品质。为此，4GWM 要限定测试工具必须将编码、调试、测试集成到一个平台。

该要求实际限定测试脚本要拥有与源码一样的权益，由于历史原因，各主流语言的集成开发环境总是让代码能在同一平台下编辑、调试的，现在既然把脚本也看成一种代码，就应该赋予它同等权益。拿通俗的话来讲，我们要构造一种集成平台，集编码、调试、测试于一身，是为了让“测试”这个后妈晋升级为亲妈，原先“调试”是亲妈，占尽天时地利，不妨从 IDE 让出一些位置。

把调测一体化平台作为 4GWM 特征之一明确下来，可以防止 4GWM 在不同编程语言及不同测试工具下实施走样。请注意，集成平台的规定不是 4GWM 本质方法论，但 4GWM 对工具化支持有比较高要求，配套工具要有足够的功能，能让广大开发人员随心所欲的使用

测试手段替代调试。

4.3 持续测试

4GWM 第三个关键域是持续测试，包括 3 个关键特征：

- 测试设计先行
- 持续保障信心
- 重构测试设计

4.3.1 测试设计先行

测试先行是 XP 典型实践，XP 中的测试先行是 Test Driven Development（TDD），4GWM 规定的测试先行是 Test Design First（TDF），两者主体内容应该一致，细节要求稍有差异。

为方便大家理解，我们还是从 XP 的 TDD 基础上介绍 4GWM 的 TDF。TDD 是测试驱动开发，测试代码在产品代码之前编写，要求产品先能测试，然后在解决问题过程中补充设计或完善设计。一个简单的 TDD 例子，比如我们要编写一个函数 GetHash 计算某对象的 hash 值，定义 GetHash 函数的原型后，即开始设计用例，如：

```
// 确定函数原型
int GetHash(void *obj)
{
    assert(0,"Not define yet.");
}

// 设计用例
assert( GetHash(newObject(12)) == 12 );
assert( GetHash(newObject("AName")) == 63632 );
```

上述测试肯定通不过，所以要解决问题，先是整形对象的 hash 值算不对，我们在 GetHash 函数中添加处理分支：

```
int GetHash(void *obj)
{
    if ( ObjType(obj) == dtInt )
    {
        ...
        return iHash;
    }
    assert(0,"Not define yet.");
}
```

然后，再次运行用例发现字串对象的 hash 值也不对，再添加相应处理代码。

TDF 也按上述模式操作，但相比 TDD 稍有差异，主要表现在：

1. TDD 强调测试驱动开发，即：测试先做，然后在测试主导下完善被测系统。而 TDF 只是要求测试设计先做，并不强制测试代码总比被测功能先跑起来。

   TDD 要求一开始就写规范的用例，而 TDF 更多的是让调试环境先跑起来，调测代码既可以是规范的用例，也可以是待整理的脚本，即草稿状态的用例。

2. TDD 更倾向于自顶向下的开发模式，TDF 则较少受此限制，实际操作时，使用最多的是混合模式。即：如果自顶向下比较容易操作，就自顶向下先设计用例，如果自顶向下不好操作，先自底向上先写底层代码也无妨。

   TDF 通常采用三文治操作模式，即：先设计少量用例，让调测环境顺利跑起来，接

着补充功能代码，最后再增加用例使新写的代码能完整测试。因为功能编码夹在中间，成为三文治的馅，过程的两端都是用例设计。由于结构化设计的缘故，TDF 三文治模式也是层层嵌套、依次深入的，先写高层次测试脚本，接着高层次编码，然后补充高层次测试设计，之后进入下一层结构化设计，同样先设计下层测试脚本，接着下层功能编码，再补充下层测试设计。

3. TDF 要求尽可能高效的编写用例，调试操作可以转化成用例，已测试通过的功能也可以在用例中重用，TDD 对此没有特别要求。

TDD 与 TDF 都强调尽可能在编码之前设计用例，看得到代码后编写用例容易坠入惯性思维陷阱，比如，某个被测函数少了一个分支处理，看自己写的代码做测试，也同样容易忽略这个分支。所以，先写脚本后写代码可以检验设计是否合理，这时测试设计依据的是规格。

测试先行经 XP 实践论证，整体是可行的， Boby George 与 Laurie Williams 的统计数据表明（参见《An Initial Investigation of Test Driven Development in Industry》），实施 TDD，有 87.5%的开发者认为能更好理解需求，有 95.8%认为 TDD 有助于减少 bug，78%的人认为 TDD 提高了生产率，另外还有 92%的人认为 TDD 能促进代码质量，79%的人认为 TDD 有助于简化设计。同时，这份统计还表明，有 40%开发者表示采用 TDD 比较困难，困难主要原因在于看不到代码情况下先做测试设计，容易让人无所适从。

TDF 在一定程度上克服 TDD 应用困难的弊端，它并不过于强调测试设计一定先于编码，但要求先行编写的测试脚本与代码能尽早展现功能，或尽早的验证规格，脚本与代码一起对等的被设计者用来实施他的意图——当然，遵循结构化设计原则，越高层越抽象的逻辑应先验证，越重要的功能也应先验证。尽早展现功能，也意味着：写一点测一点、测一点写一点，一有可展现或可调试的小功能，测试设计总与功能编码同步跟进的。

4.3.2 如何持续保障信心

4GWM 非常强调维持良好的客户体验，在线测试保证白盒测试所见即所得，人性化操作催生快感，拉通测试小循环与开发大循环，使工作效率大幅提高，强化了这种快感，现在再加一条：测试过程可度量，让开发者至始至终都对自己的代码充满信心，巩固快感使个体愉悦延伸到团队愉悦。

白盒测试最重要的度量指标是覆盖率，包括语句覆盖、分支覆盖、条件覆盖、组合条件覆盖、路径覆盖、数据流覆盖等。设计测试度量标准，不是种类越多就越好，也是越高标准（如路径覆盖、MCDC 覆盖）就越好，最重要的是，要恰如其分，另外还得考虑现实因素：测试工具能不能支持。尤其在持续测试模式下，恰当的选择覆盖指标尤显重要，要求过高使测试成为累赘，必然让持续测试做不下去。与一次测试不同，不恰当覆盖指标带来的负面影响，在持续迭代中放大了，稍过复杂就带来很大伤害。

实践经验表明，常规的白盒测试拿语句覆盖与分支覆盖度量已经足够，对于局部逻辑复杂的代码，再增设 MCDC 覆盖就够用了。4GWM 推荐把调用覆盖（近似于语句覆盖）当作主要测试指标，调用覆盖是观察函数调用与被调用关系的一种覆盖指标，因为 4GWM 以函数为单位关注测试过程，函数是识别不同测试及同一测试中不同分层的依据，以调用关系度量测试程度，是这种基于调用接口、灰盒模式的测试方法论自然延伸。

除了覆盖率指标，我们还得区别经意测试与不经意测试。比方测试某特定分支设计一个用例，除了你期望的分支跑到外，同一函数中其它部分的某些分支也能跑到，这是不经意产生的覆盖率贡献。不经意测试使结果评估产生偏差，也给想偷懒的员工带来便利，比方，测试某通信产品，设计用例打一个电话，就可能贡献 20%的覆盖率。

为避免上述情况，4GWM 设计出另一指标：测试设计程度（或称用例覆盖度），该指标分

析测试工程中，被测函数调用次数与该函数分支总数的关系。一个函数分支越多，就应设计更多的用例来测试它。用例覆盖度是作为基础条件参与测试评估体系的，设置门槛阀值，过了门槛条件，即使多设计用例也不给测试效果加分，但没过门槛，结果评估则是一票否决的。

4GWM 要求测试工具以直观、简洁的方式随时统计测试程度。因为是增量式设计，被测代码与测试脚本都按对等速度递增的，测试评估先要求定义测试观察范围，选中当前关注的被测源文件与脚本文件，成为测试工程，然后，工具始终以工程为单位进行评估，在主操作界面显示一个标志灯，亮红灯表示当前测试未通过，有 bug 等待先解决，亮黄灯表示测试通过了但覆盖率指标不符合要求，亮绿灯表示满足覆盖指标并且测试通过。

遵循 4GWM 的软件开发过程，就是时时刻刻要让界面绿灯亮起的持续开发过程，这好比开车，功能编码是踩油门，测试编码是踩刹车，界面红绿灯是执法标准，只亮绿灯才能往前走。规则已经很清晰了，时时刻刻遵守交规就是持续信心的保障。

4.3.3 重构测试设计

做好人不难，难就难在一辈子都做好人（做坏人更难？没见过一辈子只做坏事的人）。我们照章开车，没人给你开罚单，但不意味着项目就没问题了，方向走反了是南辕北辙，方向偏了可别指望歪打正着。同样，要让白盒测试能持续的跟进，很重要一点，测试设计要能快速重构。软件设计总是难免出错，事实上，多数产品开发都会经历几次局部重构，当被测代码大幅调整，规模与之对等的测试代码如何快速修正成了迫切待解决的难题。

重构测试设计要依据被测代码，测试工具应保存最近绿灯状态时的源码信息，比如，系统中都有哪些全局符号（变量、函数），符号是什么类型，被测函数都调用哪些子函数、都使用哪些全局变量等。重构测试设计时，依据历史被测代码与重构后代码的差异，自动分析当前哪些用例会受影响，如何影响，再具体指出哪些脚本行应作调整。这好比开车走错路，要回头想想在哪个十字路口开始错的，错在哪个方向。当上述过程有工具帮我们分析，维护用例的效率就高多了。

5 结论

目前，4GWM 已有实践主要集中在 C 语言测试，在线测试、持续测试诸多实践很早就有测试工具支持，已有数年应用积累。本文归纳的 4GWM 九大特征，都来源于白盒测试长期实践，先实践后总结，先有具体应用，然后归纳出通用方法。

这里再总结一下，上文介绍的 3 个关键域中，在线测试是基础，是维持良好客户体验的第一步，在线测试不仅拉通测试小循环，初步解放生产力，而且，在线特性让灰盒调测成为可能。灰盒调测拉通开发大循环，再次大幅度解放生产力。当测试效率两度提升后，持续集成就不再困难了。

参考资料

1. E. Michael Maximilien, "Assessing Test-Driven Development at IBM"

2. Joel Spolsky, "Joel On Software"

3. Elfriede, D. "Effective Software Testing: 50 Specific Ways to Improve your Testing"

4. George, B. and Williams, L., "An Initial Investigation of Test-Driven Development in Industry"

5. ezTester inc., "VcTester User Manual"

6. Philip M. Johnson, and Joy M. Agustin, "Keeping the coverage green: Investigating the cost and quality of testing in agile development"

7. IPL Information Processing Ltd, "Why Bother to Unit Test?"

