第 4 代白盒测试方法介绍—VcTester 实践篇

2006-6-15

拟制：Wayne Chan	2006-06-15
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	2006-06-15	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 代白盒方法测试实践介绍，结合使用 VcTester 实施一次具体测试过程，描述在线测试、灰盒调测、持续测试等特性的实践特征。

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
SAR：Select And Run，选中执行
PCO：Points of Control and Observation，观察控制点
TDF：Test Design First，测试设计先行
LICC：Location-Independent Call Coverage，位置无关调用覆盖
LDCC：Location-Dependent Call Coverage，位置相关调用覆盖
TCC：Test Case Coverage，用例覆盖度

1 从实践到理论，再到实践

1.1 先实践后总结

本文的上篇从理论角度描述了 4GWM 的九项关键特征，本文再从实践角度诠释这些特征。

4GWM 方法来源于长期工程实践，强调实践性是该方法论的显著特色，现实测试中某个问题难以解决，我们先尝试各种各样方法，最后形成最佳解决方案，归结升华就成为 4GWM 方法论体系。

比如，当年引入测试先行，并非要赶 XP 时髦，而是克服一个具体问题。在推行在线测试时，许多项目组反映测试工具好用但缺少意义，因为测试针对看得见的代码，比方计算“1+1”，设计用例测试它，其结果 1+1 肯定是等于 2 的。这个问题在一次测试模式下普遍存在，其根源在于测试设计方法，若只按代码（而非按规格）设计用例，肯定会漏掉不少设计问题。尽管如此，我们后来的实践表明，这个问题并不是向大家强调要改变测试习惯就能解决的，若从根本上去解决，还得引入测试设计先行的实践。

再如，引入测试设计程度评估，也是为了解决一个长期实践中难以回避的问题，即：覆盖指标可以评估大家有没做过测试，但无法推断是否用心的做测试了。统计问题缺陷密度不足以说明问题，因为发现问题数量直接与被测代码质量相关，测试发问题多了，一定程度上能说明问题，但发现问题少了就不好说，高手会宣称自己写的代码不必测，而假装高手的人会说，“嗯，我写的代码问题本来就少嘛！”，引入用例覆盖度尝试解决这个问题。

1.2 已有的 4GWM 实践

目前第 4 代白盒方法主要实践集中在 C 语言与 CSE 语言，包括在线测试、持续测试等特性，已在许多产品得到充分验证。当然，4GWM 体系经历长期发展才最后形成，业界已有一个测试工具支持它。VcTester 是其中一款，它支持 C/C++语言的 4GWM 方法测试，支持脚本驱动、脚本桩、在线测试等功能。

下文并不尝试把 4GWM 涉及的功能彻头彻尾介绍清楚，做到这一点实际很难，也不见得必要。我们仅列举该测试方法区别于常规方法的主要特征，更细节内容，请大家参考

VcTester 相关参考手册。4GWM 是通用方法论，对具体被测语言没有限定，既然 C 语言能够支持，相信多数编程语言都能支持（开发一个类似 VcTester 的工具即可），下面我们拿 VcTester 商用版为例展开介绍。

1.3 从一个例子开始

下面我们以开发“冒泡排序”模块为例展开叙述。我们的任务是：使用冒泡排序算法，开发一个原型为“`void BubbleSort(OBJ_DATA *ObjList, int iMax)`”的函数，该函数对传入参数 ObjList 数据排序，`BubbleSort` 中调用 ObjCompare(Obj1,Obj2)函数两两比较数据大小，比如 `ObjCompare(Obj1,Obj2)`返回小于 0 的值表示 Obj1 小于 Obj2，返回 0 表示两者相等，返回值大于 0 表示 Obj1 大于 Obj2。

源码请参见附 1：被测代码 BubbleSort

2 在线测试

2.1 在线驱动与在线桩

被测代码已定义待排序的数据结构：

```
typedef struct {
  int Level;
  double Data;
} OBJ_DATA;
```

该数据定义映射到 CSE 脚本就是 vt.struct.OBJ_DATA，比较两个 OBJ-DATA 数据，先比较 Level 子成员，Level 相同再比较 Data 成员，BubbleSort 按此规则将数组从小到大排序。开发功能代码之前，我们先编写如下调测脚本：

```
buff as vt.struct.OBJ_DATA[3]();
buff[0].Level = 0;
buff[0].Data = 4.0;
buff[1].Level = 0;
buff[1].Data = 3.0;
buff[2].Level = 0;
buff[2].Data = 1.0;

vd.SortAndPrint(buff,2);
assert(buff[0].Data() == 1.0);
assert(buff[1].Data() == 3.0);
assert(buff[2].Data() == 4.0);
```

接着设计功能代码：

```
int __stdcall ObjCompare(OBJ_DATA *Obj1, OBJ_DATA *Obj2)
{
    RaiseExpt("ENotImplement",NULL);
}

void BubbleSort(OBJ_DATA_PTR ObjList, int iMax)
{
  int i,j,exchanged;

  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
                ObjList[j+1] = ObjList[j];
                ObjList[j] = ObjList[j+1];
                exchanged = 1;
            }
        }
        if ( !exchanged ) return;
    }
}
```

上面代码编译通过后，我们启动测试系统，在 CseWin 界面选中上面 CSE 脚本，按快捷键 Ctrl+E，该段代码立即运行，这样的过程是选中执行（Select And Run，SAR），SAR 是 CSE 脚本调测的主要模式，因为实时交互、所见即所得，用起来很直观，很便捷。

由于 `ObjCompare` 函数尚未定义，运行报错如下：

ENotImplement:Undescribed error.

我们得给 ObjCompare 定义脚本桩，让当前 BubbleSort 排序函数能正常调起来，如下：

```
func stub_func():
  if Obj1->Data > Obj2->Data:
    return 1;
  end else if Obj1->Data == Obj2->Data:
    return 0;
  end else return -1;
end;

vd.ObjCompare.stub(stub_func);
```

然后再运行测试脚本，发现测试通不过，我们选中上面脚本中局部代码“`buff[0].Data`”，按快捷键 Ctrl+E，可查看该变量值为 4.0，再分别选中“`buff[1].Data`”与“`buff[2].Data`”查看变量值，都是 4.0。到这里，我们立刻猜出，肯定是冒泡排序的两两数据交换出问题了，对照一下原码，果不出所料，相关代码修改如下：

```
tmp = ObjList[j+1];
ObjList[j+1] = ObjList[j];
ObjList[j] = tmp;
exchanged = 1;
```

顺便提一句，越是高手就越应该通过看代码找问题，而不是架调试环境费时费力的去定位。之后我们重起修改后的被测程序，重新执行前面调试脚本，可发现脚本跑通过了。

调试通过的脚本，我们可随时转换成正式测试用例，如下：

```
CurrClass = class TTest_BubbleSort1(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;

  func stub_func():
    if Obj1->Data > Obj2->Data:
      return 1;
    end else if Obj1->Data == Obj2->Data:
      return 0;
    end else return -1;
  end;

  func run(me):
    buff as vt.struct.OBJ_DATA[3]()
    buff[0].Level = 0;
    buff[0].Data = 4.0;
```



```
        buff[1].Level = 0;
        buff[1].Data = 3.0;
        buff[2].Level = 0;
        buff[2].Data = 1.0;

        vd.ObjCompare.stub(global.stub_func);
        vd.SortAndPrint(buff,2);
        vd.ObjCompare.stub(dummy);

        assert(buff[0].Data() == 1.0);
        assert(buff[1].Data() == 3.0);
        assert(buff[2].Data() == 4.0);
      end;
    end;
    CurrCase = CurrClass(CurrSuite);
```

当然，我们也可以先创建测试用例框架，在框架内编写测试脚本，通过 SAR 选中执行方式调试脚本，必要时配合在线单步跟踪进行问题定位。

从上面操作步骤可以看出，VcTester 提供了脚本测试驱动与脚本桩，这两者都是在线定义、再线运行的，使用起来非常方便。另外脚本桩可替代眼前尚未实现函数（如上面举例的 ObjCompare），能有效隔离函数之间的耦合关系，使“写一点、测一点”的操作模式能顺利展开。

2.2 在线测试改进

一次测试完成后，我们想看看当前 BubbleSort 函数还有哪些语句还没跑到。打开 Inspector 导航页，将编辑焦点移到 bubbleSort 函数，导航页即显示当前函数的抽象语法树。

如图 1 所示，抽象语法树下红色图标节点表示该节点尚未运行，灰色节点表示已运行。由图 1 我们立即看出，判断数据两两是否要交换的条件，其不成立分支（即不必交换的分支）还没跑到，因为前面设计用例是对 4.0、3.0、1.0 排序，每次都要交换，现在我们增加一个用例，排序 4.0、1.0、3.0，就会有一次（即 1.0 与 3.0 比较）不交换的分支能跑到。

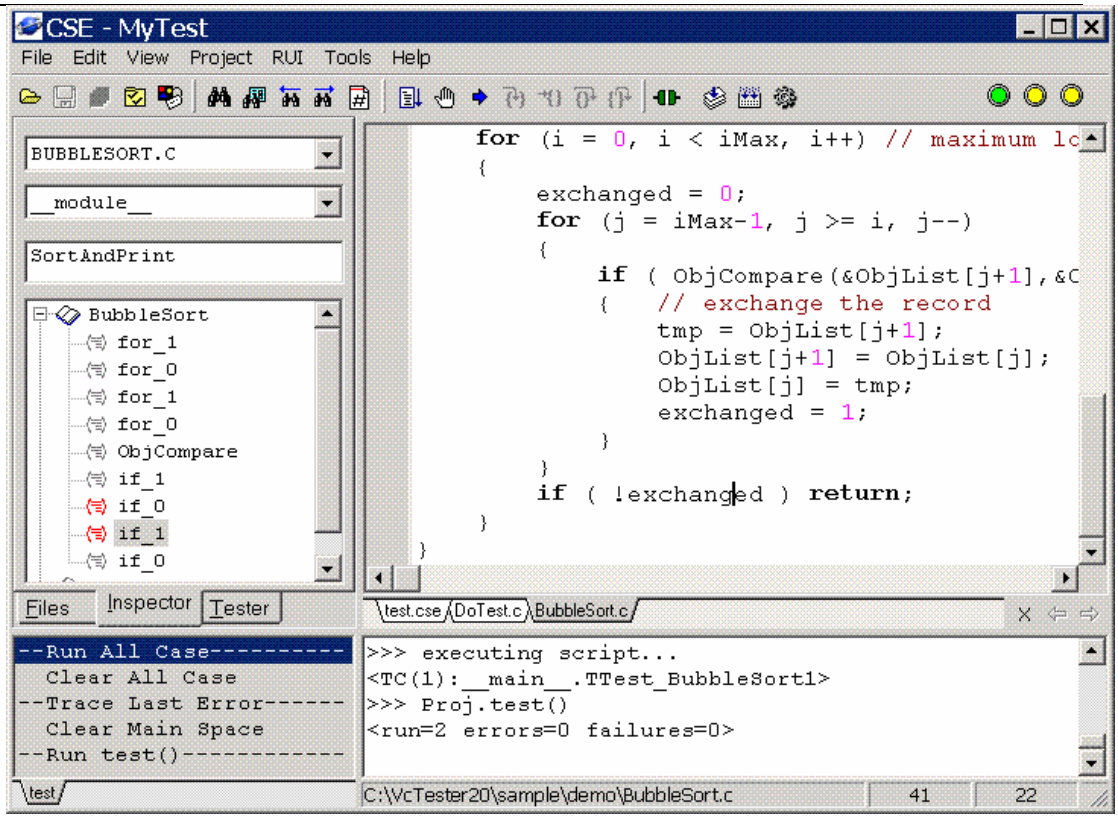

图 1：快速查看未被覆盖语句

从前面用例 TTest_BubbleSort1 拷贝代码生成新用例，TTest_BubbleSort2，调换 buff[1]与 buff[2]的顺序，即生成一个新用例。许多同志不愿写测试用例，觉得麻烦，像前面第一个用例，除掉自动生成的框架代码，仍有 20 多行，以前如此投入不值。其实不然，VcTester 提供的操作模式使用例设计尽可能重用，这里简单拷贝修改，立即设计出第 2 个用例，后面还有第 3 个、第 4 个用例都是这样的拷贝修改，所以整体效率不低。另外，VcTester 测试脚本还被调试重用，边写代码边写用例，有个很大的好处就是让尚在编码中的系统随时可调测。

运行 TTest_BubbleSort2 后再看代码覆盖情况，发现最后一个 if 分支仍未覆盖，即“if (!exchanged) return;”的 TRUE 分支没跑到，据此我们再增加用例 TTest_BubbleSort3，仍从前面用例拷贝代码再作修改，用例参见附 2：本文用到的测试用例，本处不赘述。

在 CSE 集成环境中编辑源码、调试与测试、评估测试结果，到再次完善源码或增加用例，所有操作都是在线进行的，每一步都实时交互进行。这里，我们演示了一种快速改进设计的方法，后面章节还将介绍测试覆盖率与用例覆盖度评估，评估结果也用于改进测试设计。

3 测试设计先行

3.1 测试设计先行

前面举例已经按测试先行的模式进行操作了，实施测试先行的基本步骤是：

1. 搭起新增功能的架子，即定义相关数据结构及申明待实现代码的原型

2. 编写调测脚本
3. 实现新功能
4. 发起测试，由测试问题驱动促进产品功能完善

测试设计先行的开发模式具备三个基本特征，一是系统始终处于可运行状态，不可运行是临时状态，如每次功能迭代的编码过程，一次功能迭代一般只新增几个函数，所以系统不可运行的状态非常短暂。

二是调试与测试混为一体，这是 4GWM 具备的特征，我们编写脚本驱动与脚本桩，首先要让系统运转起来，然后在调试或测试中发现问题、解决问题。当编写的脚本较为随意，它是调试用脚本，稍经整理，规范一点就是测试用例了。

三是产品趋于完善是测试所驱动的，我们经常先不具体实现某项新功能，而只用语句“RaiseExpt("ENotImplement",NULL)”让它报错，调测通不过了就要解决问题，通过问题依次解决逐步推进产品功能实现，及优化、稳定。这种测试先行的思路有别于常规软件过程，它使开发过程变为“不断解决问题”的过程，测试用例不停增加，当前用例多少直接反映待开发功能完成多少。

上面 BubbleSort 功能实现是一次迭代，我们接着下一次迭代，完成 ObjCompare 定义，首先设计调测脚本，如下：

```
CurrClass = class TTest_ObjCompare1(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;

  func run(me):
    Obj1 as vt.struct.OBJ_DATA();
    Obj2 as vt.struct.OBJ_DATA();

    Obj1.Level = 0;
    Obj1.Data = 5.0;
    Obj2.Level = 0;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert(result == 1 );

    Obj1.Level = 0;
    Obj1.Data = 4.0;
    Obj2.Level = 0;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert(result == 0 );

    Obj1.Level = 1;
    Obj1.Data = 4.0;
    Obj2.Level = 2;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert(result == -1 );

    Obj1.Level = 1;
    Obj1.Data = 5.0;
    Obj2.Level = 2;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert( result == -1 );
  end;
end;
```

然后实现 ObjCompare 函数定义，进行测试，发现尚有分支未被覆盖，增加用例使之覆盖

完全。完整的用例及源码参见附 1 与附 2。

3.2 增量开发

测试先行必然伴随增量开发，我们遵循的思路是“写一点测一点，再写一点，再测一点”，这个模式被 XP 实践称为持续集成。但请注意，4GWM 提倡的测试先行是测试设计先行，而非测试实施先行，本方法论对测试实施的要求相对宽松，只要不过于滞后就行。所以，4GWM 并不强求用例一定要先于功能代码写出来，如果测试设计针对规格，是有必要先行的，但测试操作只针对现成代码，只要不影响调测连续性，滞后是允许的。

比如前面举例的几个步骤，先关注冒泡排序的逻辑，ObjCompare 如何实现先放一边，即使是打脚本桩需要，也只比较其 Data 成员，而不综合 Level 与 Data 一起比较。等调通冒泡逻辑后，再专项关注 ObjCompare 实现。这样的迭代过程中测试与功能开发交替推进的，但我们不强求一次功能实现就彻底测试它，许多情况下，先设计用例把基本功能调通，就可以继续往下编码，等功能累积到一定程度，才细致的去分析代码覆盖情况，一次性增加用例使测试质量评估达标。

这里，我们不难看出，4GWM 奉行一种实用至上的现实主义理念，它并不为了增量开发而开发，也不为了测试先行而先行。尤其是多个功能之间耦合很紧时，每写一点代码就彻底测试可能会很低效，因为前后功能相互影响，在后面功能没还调通前，前面的功能代码不够稳定，会经常变化。

4 检视器

VcTester 的检视器一种提供脚本化控制的调试器，它在被测单元的上下层之间插入观察控制点（Points of Control and Observation，PCO），PCO 依附于检视断点，VcTester 规定一个被测函数只能设置一个检视断点，当断点条件满足，定义于 PCO 的 PreCheck 脚本与 PostCheck 脚本即自动执行。如图 2 所示：

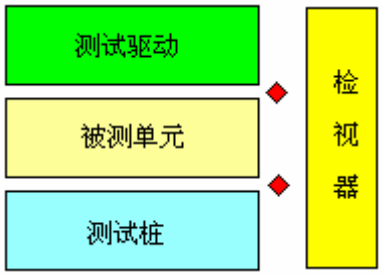

图 2：检视器结构

PreCheck 脚本在被检视函数刚进入时运行，PostCheck 在函数退出前运行，被测函数中的 PreCheck 脚本常用于修改传入参数，或修改全局变量构造出特定的测试环境，而 PostCheck 脚本常用来检查测试结果是否预期。此外，测试桩函数的 PostCheck 还常用于修改调用返回值，实现的功能相当于前面提到的脚本桩，让桩函数返回特定值便于被测函数中特定代码分支能被覆盖。

前面我们以自顶向下方式进行功能开发，下面我们换一个方式，采用自底向上方式来集成，先调测 ObjCompare 函数，然后调测 BubbleSort 函数，自底向上方式下，代码与测试脚本都容易重用。

4.1 选择检视变量

编写如下脚本测试 ObjCompare：

```
Obj1 as vt.struct.OBJ_DATA();
Obj2 as vt.struct.OBJ_DATA();
Obj1.Level = 0;
Obj1.Data = 5.0;
Obj2.Level = 0;
Obj2.Data = 4.0;

vd.ObjCompare(&Obj1,&Obj2);
```

这几行脚本用于发起测试，修改 Obj1 与 Obj2 的值可形成各式各样的用例，接下来我们演示怎么样把它快速的转化为正式用例。

先在 ObjCompare 函数设置检视断点，运行上面脚本发起测试，当程序在断点函数的头部停住时，我们在 watch 列表添加 Obj1 与 Obj2 这两个变量，再按住 Ctrl 键，用鼠标双击“Obj1->Data”与“Obj2->Data”这两个节点使之选中，如图 3，标记选中的变量是为了将相关检查语句添加到自动生成列表。

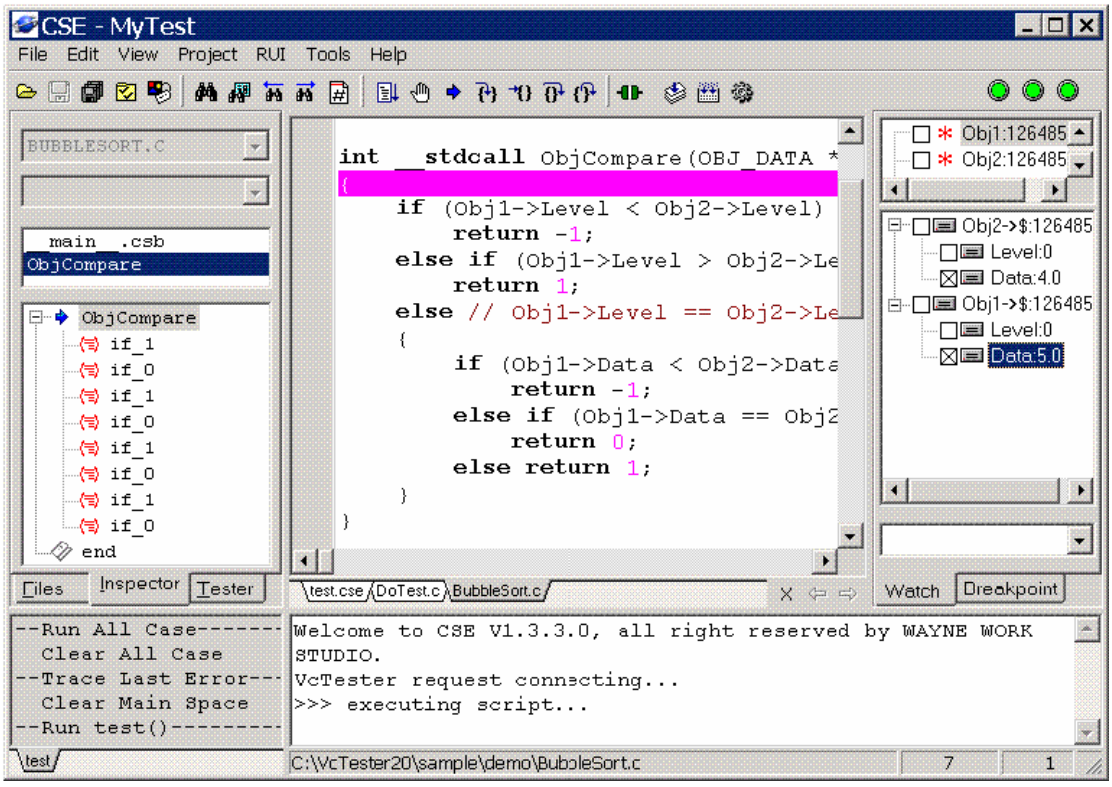

图 3：在函数首部添加检视变量

然后点击 step out 按钮（或按快捷键 F6），当前程序随即运行到断点函数的尾部停住，如图 4 所示，同样，我们按住 Ctrl 键双击鼠标，选中 ret_，这个变量表示当前函数调用的返回值。

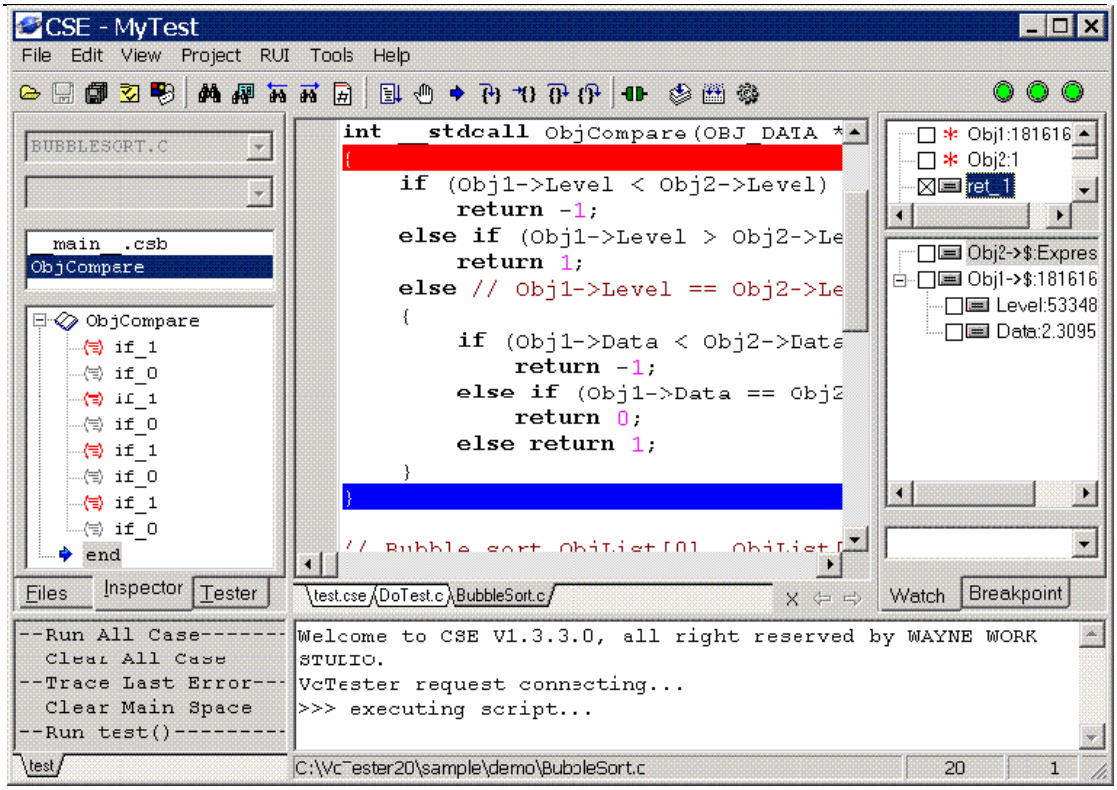

图 4：在函数尾部添加检视变量

由图 4 可见当前 ObjCompare 返回值是 1，正果是正确的。然后我们点 continue 按钮（或按 F9 快捷键）消调试断点，让程序一直跑结束。

最后，我们希望把刚才的调试操作转化为测试脚本，用 Ctrl+P 热捷粘贴自动生成的代码，这个代码加上刚才的测试驱动就形成一个完整用例。如下（方框内代码是自动生成的）：

```
Obj1 as vt.struct.OBJ_DATA();
Obj2 as vt.struct.OBJ_DATA();
Obj1.Level = 0;
Obj1.Data = 5.0;
Obj2.Level = 0;
Obj2.Data = 4.0;
```

```
Executor.setBp(vd.ObjCompare,'',0,
  # pre-check
  "assert(Obj1->Data == 5.0);
  assert(Obj2->Data == 4.0);",
  # post-check
  "assert(ret_ == 1);",
);
```

```
vd.ObjCompare(&Obj1,&Obj2);
```

自动生成的脚本用于设置检视断点，setBp 的原型为：

```
func setBp(vFunc,sCond='',ignoreNum=0,sPreCheck='',sPostCheck='')
```

其中，vFunc 用来指定断点函数，sCond 是断点条件，取空字串值表示它是无条件断点，ignoreNum 是断点的忽略次数（取值 N 表次 N 次条件满足后才在断点停下来），sPreCheck 与 sPostCheck 分别是断点函数运行前的检查语句与运行后的检查语句。用户可以手工修改自动生成代码，比方修改断点条件，设置断点忽略次数使断点函数在调用 N 次后才触发 PreCheck 等。

4.2 添加检视操作

上面我们测试 OBJ_DATA.Data 值为 5.0 与 4.0 的两个数据比较，接下来，我们重用这个用例设计，修改 OBJ_DATA.Data 为其它值再做测试。

在同样函数设断点，同样调用“vd.ObjCompare(&Obj1,&Obj2)”发起调试。当执行在函数头部停住时，我们把 Obj1->Data 从“5.0”改成“3.0”，选中相应节点，击回车在弹出对话框中输入新值即可。

选中待修改的节点，然后击回车在弹出的对话框中输入新值完成修改，系统自动将修改操作添加到检查列表，接着如前面用例，在函数末尾时再标记 ret_变量。结束调试后，按 Ctrl+P 热键粘贴自动脚本，整理用例如下：

```
Executor.setBp(compFunc,'',0,
  # pre-check
  "Obj1->Data=3.0;",
  # post-check
  "assert(ret_() == -1);",
);

vd.ObjCompare(&Obj1,&Obj2);
```

其中方框内代码是自动生成的，这个例子中，我们在检视函数的 PreCheck 修改传入参数，而在 PostCheck 检查测试结果。

5 测试设计模式

5.1 三种测试设计模式

在白盒测试中，被测单元（Unit Under Test，UUT）是我们的观察对象，UUT 上层是测试驱动，发起测试的脚本处于这一层，UUT 的下层是支撑被测单元提供正常功能的桩函数。为保证被测单元中各个分支都被覆盖到，我们通常要仿真桩函数，同样要使被测单元各项功能都遍历到，测试驱动层也通常是模拟的。

根据测试驱动与测试桩的构造方式不同，形成三种主要测试模式。其一，仿真模式，测试驱动与测试桩都在测试用例中模拟。其二，点控制模式，测试驱动与测试桩都使用真实的产品代码，但在被测单元的上下层之间插入观察控制点（Points of Control and Observation，PCO）。这一模式下，检视器控制 PCO 的行为，比如在被测单元调用前，修改某些变量的值，包括修改传入参数的值，另外也可以在桩函数调用结束时，篡改返回值，PCO 的行为可以模拟测试驱动与测试桩。其三，混合模式，仿真与点控制同时存在，测试驱动与测试桩其中一个是模拟的，另一个通过点控制实现。

前面列举了几个例子，大家回头看看，它们都属于哪种设计模式？不妨自行分析一下它们的应用特点。

本文一开始的例子中，因这 ObjCompare 函数尚未定义，我们通过打脚本桩让 BubbleSort 函数测起来的，现在我们把桩改成检视断点，如下：

```
Executor.setBp(vd.ObjCompare,'',0,
  # pre-check
  "if Obj1->Data > Obj2->Data:
     return 1;
```



```
    end else if Obj1->Data == Obj2->Data:
       return 0;
    end else return -1;",
    # post-check
    "",
  );

  buff as vt.struct.OBJ_DATA[3]();
  buff[0].Level = 0;
  buff[0].Data = 4.0;
  buff[1].Level = 0;
  buff[1].Data = 3.0;
  buff[2].Level = 0;
  buff[2].Data = 1.0;

  vd.SortAndPrint(buff,2);
  assert(buff[0].Data == 1.0);
  assert(buff[1].Data == 3.0);
  assert(buff[2].Data == 4.0);
```

打桩与设置检视断点效果等同，实际上这两者是按同一机制实现的，打脚本桩等同于在检视断点的 PreCheck 中直接返回结果值。检视断点比脚本桩提供更强大的调测能力，可以设置断点条件、忽略次数等。

5.2 如何选择测试设计模式

上述三种测试设计模式各有优劣，仿真模式要手工编写脚本驱动与脚本桩，工作效率低下，但测试代码与被测代码有隔离，用例可维护性较好。点控制模式能自动生成脚本，开发用例的效率较高，但测试脚本与被测代码耦合紧，对用例可维护性有影响。

应该根据实际情况选择哪种模式，如果被测系统的功能比较内聚，少量用例即驱动大量测试时，可以多用点控制模式。如果被测系统功能分散，或者函数接口简单容易模拟的，不妨多采用仿真模式。还有，越是低层代码或是库函数代码，较少依赖其它函数，就能越多采用仿真模式，反之，越是高层应用，应更多采用点控制模式。

6 测试效果评估

6.1 基于函数调用的评估体系

VcTester 的测试质量评估体系由两部分组成，一是测试覆盖程度评估，二是测试设计程度评估。前者主要针对用例有效性，后者主要针对用例完整性，都是基于函数调用分析来构造的。

支持两类测试覆盖率评估，一是位置无关调用覆盖率（Location-independent call coverage，LICC），二是位置相关调用覆盖率（Location-dependent call coverage，LDCC），定义如下：

LICC = (已覆盖的不重复的函数调用个数 / 全部不重复的函数调用个数) * 100%
LDCC = (已覆盖的函数调用个数 / 全部函数调用个数) * 100%

比如某函数中调用了 3 个子函数，其中第 1 个子函数在该函数定义的两个地方出现，其余 2 个子函数都只在一处调用。如果这个 3 个子函数都被调用过，其中第 1 个子函数只在一个位置调用了，另一个位置尚未调用到。这时，我们计算 LICC 是“3/3 = 100%”，而 LDCC 是“3/4 = 75%”。

LICC 指标主要用于粗测，确保某模块具备一定的初始稳定度，适合于与其它模块集成，LICC 指标尝试说明新定义的函数是否跑到过，而不关心是否每个地方都跑到。LDCC 则在正式测试中使用，比较完整的评估测试程度。

测试设计程度也称用例覆盖度（Test Case Coverage，TCC），以被测函数在正式测试中使用频度与其函数定义中分支总数之间关系作为依据，定义如下：

TCC = 用例中调用被测函数的总次数 / 函数定义的分支总数

其中，某函数的分支总数的定义如下：

函数分支总数 = 1 + if 语句总数 * 2 + while 语句总数 * 2 + for 语句总数 * 2

if 语句有 TRUE 与 FALS 两个分支，原理上说应该设计两次测试才是完整的，while 与 for 也类似，循环至少进入一次应设计一个用例，一次都不进入也应设计一个用例，所以计算出 if、while 与 for 语句总数后要乘以 2，但如果一个函数没使用这 3 个控制语句，也应至少设计一个用例测试它，这是上面公式额外加上 1 的含义。

用例覆盖度通常不作为测试结果的硬性评价指标，而只设置一个下限值，未达标时测试结果应该是一票否决的。设置这个指标可监控不经意测试，即，非测试目调用被测函数而导致代码覆盖率上升，使得覆盖指标虚高。

6.2 测试效果评估

最后我们进入正式测试，先检查一遍前面调通的用例，为让它们能连续执行，前后用例不应产生藕合，比方前面用例修改全局变量了，而后面用例是假定全局变量未被修改，冲突就产生了。前面举例的用例经整理，放在附 2：本文用到的测试用例。

定义所有用例后，我们双击快捷面板“Run All Case”运行测试工程，由打印信息可知，当前测试通过了，再查看主界面右上角三个提示灯，如图 6 所示：

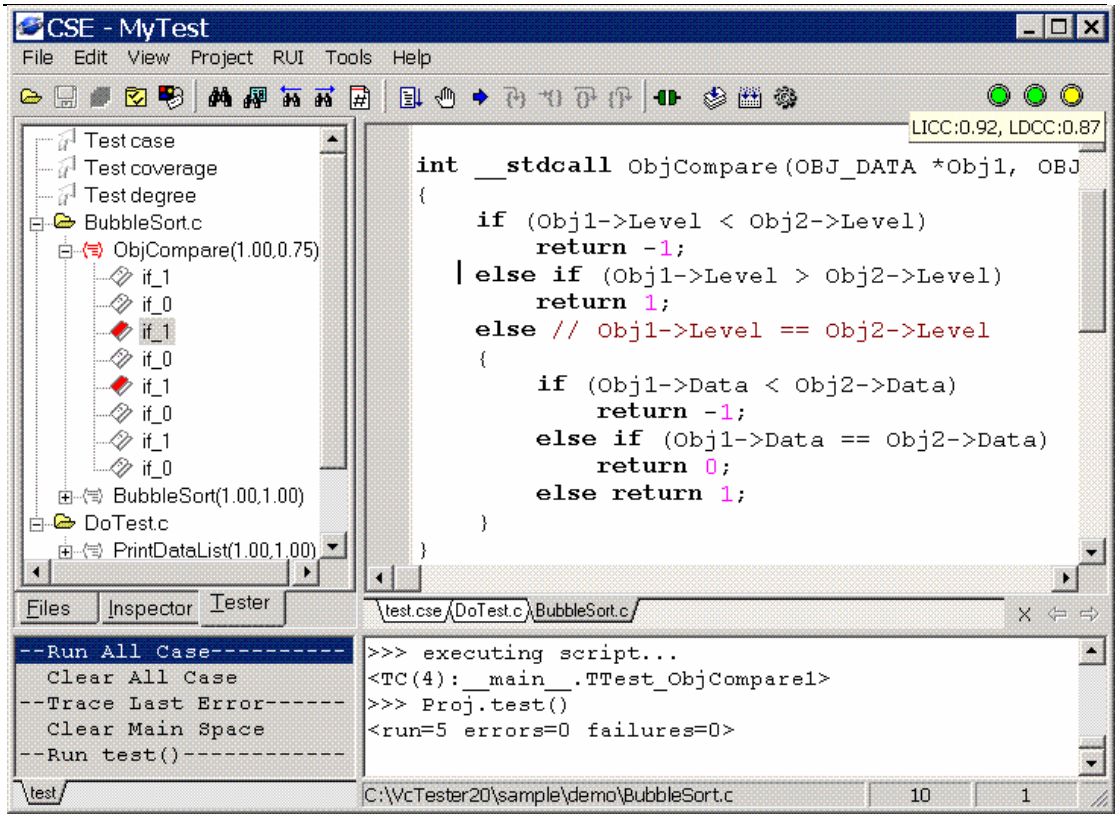

图 6：测试结果评估

左边的指示灯表示测试是否通过，中间指示灯表示覆盖率是否达标，右边的指示灯表示测试设计是否达标。把鼠标移到中间指示灯上，系统将显示详细的覆盖率信息，如上图提示信息表明当前测试已通过，LICC 覆盖率为 92%，LDCC 覆盖率为 87%，用鼠标双击覆盖率指示灯可在 Coverage 导航树查看详细信息。如上图，覆盖率导航树下灰色图标节点表示已覆盖，红色图标是目前尚未调用的函数，用鼠标双击某节点，编辑光标将跳转到该语句行，可以看出 ObjCompare 的第二个 if 语句条件为 TRUE 的分支尚未被覆盖。

我们当前设定的测试通过标准是：LICC 值为 0.8，LDCC 为 0.8，用例覆盖度为 0.5，所以本次测试，覆盖率已满足要求，右边指示灯亮黄灯，表示测试设计程度未达标（当前用例覆盖度才 0.43），同样双击这个指示灯，可在导航树查看各个函数的用例覆盖度的值。

6.3 红灯停，绿灯行

如果当前有用例没跑通过，左边指示灯将亮红灯，如果覆盖率未达标，中间指示灯将亮黄灯，如果用例覆盖度未达标，右边指示灯将亮黄灯，只有当前测试通过、代码覆盖率与用例覆盖度都达标 3 个指示灯才都亮绿灯，使用 VcTester 的开发过程，就是让这 3 个绿灯常常亮起的编码与测试过程。

红绿灯机制规定了简单的研发准则，亮红灯必须先解决问题，亮黄灯要改善测试设计，只有 3 个指示灯都是绿灯，才能进入下一步开发。如果所有团队成员都遵守“交通规则”，形成职业习惯，项目整体研发就能高质量、有节奏的平稳推进。

有了红绿灯机制，我们还得提供配套措施，保证它规范的执行起来。首先，要解决测试标准的定制问题。不同产品、不同模块有不同的质量要求，我们要求 LICC、LDCC 及 TCC 的达标值可定制，另外，当前评估范围可调整，某些关注范围之外的函数定义不必纳入统计，再有，函数内特定语句可定义到例外列表，不纳入覆盖率统计，比如错误处理，或者

某些当前不必关心的语句，如处理跨平台的语句等。

其次，还得有系列配套措施，使研发过程的代码质量、用例质量能集中监控，能自动生成报告、配合 QA 审计等。限于篇幅我们不对上述配套功能展开论述，感兴趣者请参阅 VcTester 使用手册。

基于调用分析的评估体系，是我们经过多年实践，最后沉淀下来的操作模式，优于其它常见评估手段。比如，如何选择覆盖标准存在一个误区，大家通常认为覆盖率评估得越细越好，标准越高越好，其实未必，作为一项可持续的软件过程，首先要求适度，保障操作可持续性应该摆第一位，其次才是拔高要求，能够锦上添花是最好。

另外，维持测试评估体系的一致性与稳定性也很重要，商用工具经常采用的分支覆盖、MCDC 覆盖等评估标准，在适应不同场合或不同风格的代码时，都存在一定“噪声”。比方分支覆盖中，条件判断的 AND 与 OR 具有短路功能，只取条件结果分析哪条路径覆盖或未覆盖是有失偏颇的，尽管不必考虑条件组合情况，但该对等看待的分支确实漏统计了。再如 C 语言中的 switch 语句，统计各 case 分支实际只分析可见代码，漏写一处 case 是统计不到的，但在 if 与 else 分支统计时，漏写 else 却可以统计，显然不怎么协调。

再有，常见商用工具将特殊语句（如错误处理、不可达或难到达代码），不作区分的纳入评价标准，这些都会给项目持续平稳的运作带来伤害。

7 总结

4GWM 集中了我们认为比较优秀的白盒测试实践，是经提炼的通用方法论，VcTester 工具作为该方法论的载体，又是通用方法的具体语言下具体实践。

工具与方法论是互为依赖并共同促进的，这好比是茶道与茶具，茶道总结了良好的喝茶体验，而茶具固化并传承这种体验，比如，茶壶要多大，多少茶只能加多少水，要泡多长时间等都有讲究，茶杯必小，且分三口徐徐下咽（三口才成品，否则是牛饮），这种具体化、规格化的器具，加上形式化的操作规则，必能有效传承经验。

VcTester 就是这种规格化的器具，承载了良好的测试习惯，我们在详细介绍第 4 代白盒方法同时，也希望广大同道中人积极参与，尽快将 4GWM 扩展到其它语言，如 Java、C#等。

附 1：被测代码 BubbleSort

```
//=========================================================
// BubbleSort.h

#ifndef __BubbleSort__
#define __BubbleSort__

typedef struct {
  int Level;
  double Data;
} OBJ_DATA;

typedef OBJ_DATA *OBJ_DATA_PTR;

extern int __stdcall ObjCompare(OBJ_DATA *Obj1, OBJ_DATA *Obj2);
extern void BubbleSort(OBJ_DATA_PTR ObjList, int iMax);

#endif


//=========================================================
// BubbleSort.c

#include <stdio.h>
#include "__symbols.h"
#include "BubbleSort.h"

int __stdcall ObjCompare(OBJ_DATA *Obj1, OBJ_DATA *Obj2)
{
  if (Obj1->Level < Obj2->Level)
     return -1;
  else if (Obj1->Level > Obj2->Level)
     return 1;
  else // Obj1->Level == Obj2->Level
  {
     if (Obj1->Data < Obj2->Data)
        return -1;
     else if (Obj1->Data == Obj2->Data)
        return 0;
     else return 1;
  }
}

// Bubble sort ObjList[0]..ObjList[iMax]
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;
  }
}
```


附 2：本文用到的测试用例

```
uses unittest;
uses CppSys*;

CurrClass = class TTest_BubbleSort1(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;

  func stub_func():
    if Obj1->Data > Obj2->Data:
      return 1;
    end else if Obj1->Data == Obj2->Data:
      return 0;
    end else return -1;
  end;

  func run(me):
    buff as vt.struct.OBJ_DATA[3]();
    buff[0].Level = 0;
    buff[0].Data = 4.0;
    buff[1].Level = 0;
    buff[1].Data = 3.0;
    buff[2].Level = 0;
    buff[2].Data = 1.0;

    vd.ObjCompare.stub(global.stub_func);
    vd.SortAndPrint(buff,2);
    vd.ObjCompare.stub(dummy);

    assert(buff[0].Data == 1.0);
    assert(buff[1].Data == 3.0);
    assert(buff[2].Data == 4.0);
  end;
end;
CurrCase = CurrClass(CurrSuite);

CurrClass = class TTest_BubbleSort2(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;

  func stub_func():
    if Obj1->Data > Obj2->Data:
      return 1;
    end else if Obj1->Data == Obj2->Data:
      return 0;
    end else return -1;
  end;

  func run(me):
    buff as vt.struct.OBJ_DATA[3]();
    buff[0].Level = 0;
```



```
    buff[0].Data = 4.0;
    buff[1].Level = 0;
    buff[1].Data = 1.0;
    buff[2].Level = 0;
    buff[2].Data = 3.0;

    vd.ObjCompare.stub(global.stub_func);
    vd.SortAndPrint(buff,2);
    vd.ObjCompare.stub(dummy);

    assert(buff[0].Data == 1.0);
    assert(buff[1].Data == 3.0);
    assert(buff[2].Data == 4.0);
  end;
end;
CurrCase = CurrClass(CurrSuite);

CurrClass = class TTest_BubbleSort3(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;

  func stub_func():
    if Obj1->Data > Obj2->Data:
      return 1;
    end else if Obj1->Data == Obj2->Data:
      return 0;
    end else return -1;
  end;

  func run(me):
    buff as vt.struct.OBJ_DATA[4]();
    buff[0].Level = 0;
    buff[0].Data = 4.0;
    buff[1].Level = 0;
    buff[1].Data = 5.0;
    buff[2].Level = 0;
    buff[2].Data = 3.0;
    buff[3].Level = 0;
    buff[3].Data = 1.0;

    vd.ObjCompare.stub(global.stub_func);
    vd.SortAndPrint(vc.buff,3);
    vd.ObjCompare.stub(dummy);

    assert(buff[0].Data == 1.0);
    assert(buff[1].Data == 3.0);
    assert(buff[2].Data == 4.0);
    assert(buff[3].Data == 5.0);
  end;
end;
CurrCase = CurrClass(CurrSuite);

CurrClass = class TTest_ObjCompare1(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;
```



```
  func run(me):
    Obj1 as vt.struct.OBJ_DATA();
    Obj2 as vt.struct.OBJ_DATA();

    Obj1.Level = 0;
    Obj1.Data = 5.0;
    Obj2.Level = 0;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert(result == 1 );

    Obj1.Level = 0;
    Obj1.Data = 4.0;
    Obj2.Level = 0;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert( &result == 0 );

    Obj1.Level = 1;
    Obj1.Data = 4.0;
    Obj2.Level = 2;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert( result == -1 );

    Obj1.Level = 1;
    Obj1.Data = 5.0;
    Obj2.Level = 2;
    Obj2.Data = 4.0;
    result as vd.ObjCompare(&Obj1,&Obj2);
    assert( result == -1 );
  end;
end;
CurrCase = CurrClass(CurrSuite);

CurrClass = class TTest_ObjCompare2(TCase):
  func __init__(me,Owner):
    TCase.__init__(me,Owner);
  end;

  func run(me):
    Obj1 as vt.struct.OBJ_DATA();
    Obj2 as vt.struct.OBJ_DATA();
    Obj1.Level = 0;
    Obj1.Data = 5.0;
    Obj2.Level = 0;
    Obj2.Data = 4.0;
    vd.ObjCompare(&Obj1,&Obj2);

    Executor.setBp(vd.ObjCompare,'',0,
      # pre-check
      "assert(Obj1->Data == 5.0);
      assert(Obj2->Data == 4.0);",
      # post-check
      "assert(ret_ == 1);",
    );
    vd.ObjCompare(&Obj1,&Obj2);
```



```
    Executor.setBp(vd.ObjCompare,'',0,
      # pre-check
      " Obj1->Data = 3.0;",
      # post-check
      "assert(ret_ == -1);",
    );
    vd.ObjCompare(&Obj1,&Obj2);

    Executor.setBp(vd.ObjCompare,'',0,
      # pre-check
      "Obj1->Data = 4.0;",
      # post-check
      "assert(ret_ == 0);",
    );
    vd.ObjCompare(&Obj1,&Obj2);
  end;
end;
CurrCase = CurrClass(CurrSuite);
```

