如何让编程产能翻二番？

eTDD：代码质量与编程产能双飞跃的奥秘

广州凯乐软件技术有限公司 王彤

wangtong@kailesoft.com

http://www.kailesoft.com

2014 年 05 月

目 录

0. 引子：开发过程改进的最大问题：轻视编码

开发过程改进的最大问题是轻视编码。

软件开发过程有一个很自然的比喻：建楼房，要画图纸、搭架构、砌砖头。这个比喻本身没有大问题，但是，我们常常把这个“砖头”看作普通的砖头，其结果就轻视编码。这个“砖头”绝不是普通砖头，具有以下特性：

一、每一块都由手工制作；

二、每一块的材质、形状、尺寸都不同；

三、任何一块有问题，在某种天气下，房子会漏水；如果 1%甚至 1‰的砖头有问题，房子会到处漏水；更严重的是，一块或多块砖头有问题，还可能造成房子在某些时间倒塌。

正是因为“砖头”的上述特性，所以存在两个 80%：软件的质量，80%依赖于代码；软件的开发成本，80%用于编写合格代码，请注意，是“合格代码”，不是“代码”，后期的测试和排错，也是编写“合格代码”的组成部分。即使两个 80%不准确，改为 70%，甚至 60%，决定软件质量和开发成本的大头，也仍然是编码。

我们在试图改进开发过程，提升管理时，是不是将代码看作普通的砖头？有没有将编码过程的改进，作为提升管理的重点？

管理，是有目标有对象的。作为开发团队，管理的目标，无非就是提升产品质量，降低开发成本。既然决定软件质量和开发成本的大头是编码，那么，不重视编码过程的改进，提升管理就失去了最主要的着力点，必然会感觉什么措施都效果不明显。

如果建楼房的砖头具有上述特性，相信任何一个建筑商，都会足够重视改进做砖头的工艺，一定会绞尽脑汁，找出一次性做出合格砖头的办法，而不是等到楼房建起来再查找和修补有问题的砖头。

编码过程的改进，并不复杂，只要做到一次性编写合格代码就行。如何做到一次性编写合格代码？首先是主观上的重视，特别是领导层的重视，态度决定一切；其次，找到可行的办法。eTDD(easy TDD，易行版测试驱动开发)，就是改进编码过程、一次性编写合格代码的可行办法。

编码，是决定软件质量和开发成本的大头，抓住了这一重点，就扭住了提升软件质量和降低开发成本的牛鼻子。其实，软件开发过程和建楼房有很大的不同，那就是软件的需求是不断变化的，而基于单元测试的 eTDD，另一个意义恰恰是使开发过程适应频繁变化的需求。

1. eTDD 是什么？

TDD（测试驱动开发）是敏捷开发的核心实践之一， Ruby On Rails 的创始人 David Heinemeier 曾说，当年它打开了一扇门，让自己看到了高质量代码的全新世界。然而，他近日发表了一篇文章：《TDD 已死，测试永生》，痛批 TDD 过于偏重单元测试，过于琐碎，会使系统同许多中间层、中间对象组成，带来复杂臃肿的架构。他明确声明自己将放弃 TDD。

TDD 拥有大量的狂热支持者，也不乏如 David Heinemeier 一样，曾经倾心支持，但经过长期实践后，最终放弃者。这说明两点：一是成本高，TDD 太麻烦、副作用大、成本高昂；二是效益高，成本如此高昂的情形下，仍然有那么多的支持者，说明 TDD 效益巨大，至少，支持者们相信，TDD 的综合效益高于它的综合成本。

假如对 TDD 进行改进，消灭它的主要成本，并放大它的效益，那么，就得到了一个大幅提升代码质量、大量降低开发成本的开发方式，这，就是 easy TDD，简称 eTDD。

eTDD 如何做到消灭 TDD 的主要成本、放大 TDD 的效益呢？答案是利用工具：由工具完成苦活、脏活、重活；由工具描述程序行实现可视编程。

苦活、脏活、重活包括：编写测试驱动、编写桩代码、编写 Mock、覆盖统计、找出遗漏用例、编写测试报告，并避免仅仅为了单元测试而做的重构，总之一句话：除了测试数据需要人工设定外，其他都由工具完成，这样，就消灭了 TDD 的主要成本。

可视编程则显著放大 TDD 的效益。可视编程就是编写代码时，可以随时察看程序行为。程序行为，就是在什么输入下，会执行哪些代码，会产生什么输出。单元测试的输出可以完整描述程序行为，使程序行为可视，这是极宝贵资源，TDD 忽略了这一点，甚为可惜。利用可视编程，程序员写几行代码，就可以看看程序会做什么，从而验证思路、发现错误、激发灵感。对于比较复杂的程序，用可视编程编写合格代码的效率，是传统方式的 4、5 倍，同时，还能降低编程的劳动强度，保护程序员的健康。

eTDD 就是：由工具完成大部分工作的、以可视编程为核心的测试驱动开发。

eTDD 强调利用工具，这是因为，人类生产力的提升，绝大多数依赖工具的进步，软件开发也不例外。不好好利用先进工具，而是热衷于那些天马行空难以落地的所谓新模式新思想，企图实现质量与产能的大提升，结果只能是竹篮打水。忽视人的成本，工具都能做的事，硬要由人工来做，那是极大的浪费。举个例子：一名月薪 1 万的程序员，公司要付出的实际成本为 1.5 万至 2 万，折算成小时成本，大概是 100 元，即手工编写测试代码，1 小时要付出 100 元，而这种工作完全可以由工具代劳。更大的损失是效益损失，请想一想，如果这 1 小时用来做开发，能产生多少效益？

如果不做单元测试，成本会更高：代码质量难以保证、要花更多的时间来排错，“欠下的债总是要还的”。有些公司为了降低成本，使用实习生或初级程序员来做单元测试，这是错误的，除非程序员为每个函数都编写非常详细的文档，否则测试只能跟着代码走，没有实际意义，更可惜的是，无法享受测试对开发的驱动效益。

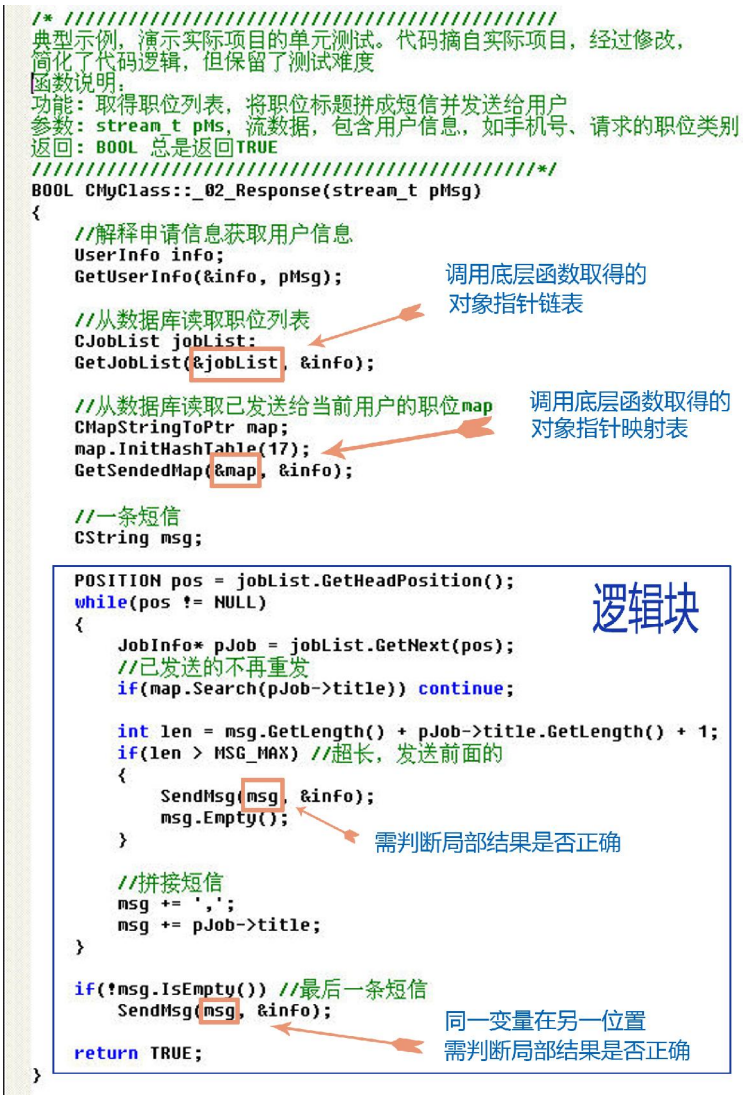
eTDD 将麻烦的事情交给工具，例如，不用考虑让人头疼的“可测性”问题，因为解决可测性问题，是 eTDD 工具的基本功能。eTDD 可以在项目周期的任意时段引入，还可以部分引入，例如，只对较底层或较重要的代码使用 eTDD。总之，只要使用合适的工具，eTDD 很容易实施，效果立竿见影。

2. eTDD 过程

eTDD 过程可以概括为：逻辑块可视编程；提交前完成覆盖；只进行粗略调试。可视编程是 eTDD 的核心，也是提升代码质量与编程产能的关键。

2.1 什么是逻辑块？

eTDD 也是单元测试驱动开发，但是，由测试驱动进行开发的，并不是全部代码，只是逻辑块而已。单元测试应该面向逻辑块。单元测试做什么？真正有过实践的工程师都知道，单元测试要做的、能做的，就是检测代码的功能逻辑，扯上其他东西没有任何意义。功能逻辑由什么实现？逻辑块！所以，单元测试应该面向逻辑块。



很低层的函数，例如排序函数、字符串处理函数，通常只有一个逻辑块，且逻辑块的输入输出就是函数的输入输出，这是最简单的情形。更常见的情形是，一个函数中调用了其他

函数来取得数据，然后处理一个或多个功能逻辑，也就是包含一个或多个逻辑块。这种情形下，逻辑块的输入输出往往与函数的输入输出不一致。如上图所示，函数的功能是发短信，但真正的逻辑计算，是拼接短信，蓝框内的代码就是逻辑块，是单元测试的目标。一旦我们把目光转向逻辑块，测试思路就会变得简单，例如，上图中，逻辑块的输入是一个链表和一个映射表，输出是拼接后的字符串，测试时只要直接设定这三项数据就行，具体的测试方法跟工具有关，后文会进一步说明。

2.2 逻辑块可视编程

当编写没有逻辑计算的代码时，不用考虑测试，按原来的习惯编写；

当编写逻辑块时，设计先行，将逻辑块的功能点用输入输出的方式记录下来，这样，既完成了逻辑块的设计，明确和细化了功能点，同时也建立了测试用例；

编写逻辑块代码，随时察看程序行为，如果程序行为与预期不一致，则可能存在思维偏差或录入错误，对比输入输出及所执行的代码，一般可以找出问题原因。当思维滞塞时，观察现有代码的行为，通常也可以激发灵感；

代码编写完成后，察看是否存在失败的测试，并察看测试覆盖状况，补充遗漏用例。最后，看看代码是否有需要优化的地方，有的话完成优化并再次执行测试。

eTDD 并不强求设计先行。eTDD 认为，编码和设计是互相促进的。如果功能逻辑很简单，但数据很复杂，设计先行的意义就不大，可以先写代码，然后再添加用例察看程序行为。当然，也可以写代码和添加用例交替进行。eTDD 强调舒服，怎么舒服怎么来。

2.3 提交前完成覆盖

“提交前”的含义，是指当工作告一段落时，例如，一个小模块完成了。如果使用了版本控制工具，这时将要提交代码。提交前，对未测试的函数执行测试，完成覆盖。通常，没有逻辑计算的代码，用工具生成的空数据即可完成覆盖，即使要添加测试数据，往往也是很简单的一、两组数据。没有逻辑计算的代码，通常也没有测试意义，之所以要完成覆盖，主要考虑管理上的意义，管理层一般通过测试报告中的覆盖率来检查测试完整性。

2.4 只进行粗略调试

eTDD 强调可视编程，可以随时察看程序行为。程序行为就是在什么输入下，会执行哪些代码，会产生什么输出，这本身就是更便捷的调试，错误原因通常很容易找出。

在开发过程中，调试只用于跟踪大的流程。

后期测出错误后，可用调试来跟踪执行流程，粗略定位 bug，即定位到可能出错的函数。然后，不要对函数的功能逻辑进行调试，而是：

1）补充用例数据。一般来说，根据错误表现，可以知道错误是在什么数据下产生的，检查用例中是否存在此类数据，如果没有，添加数据，重新执行测试，如果测试失败，表示定位正确；否则，错误可能位于其他函数，重新定位。

2）用可视编程修改代码使测试通过。

传统开发方式下，调试时间往往是编码时间的 2 至 5 倍，eTDD 可以省略 90%的传统调试。

2.5 eTDD 过程总结

eTDD 不强调绝对的测试先行，而是视需要测试。对于没有逻辑计算的代码，通常只需要在提交前用空数据执行测试，而编码时不考虑测试，这是尊重程序员的工作特性，编程需要专注，测试不能破坏专注。

对于逻辑块，尤其是复杂的逻辑块，通常难以一下子想清楚具体要做什么，本身具有强烈的设计先行的需求，用输入输出的方法，记录下自己所想，明确逻辑块的功能点，这不但可以整理思路，也可以最大限度减少遗漏。这种记录，就是测试用例。对逻辑块的设计，是开发思维的记录和完善，且未跳出开发方向，因此不会破坏专注。

对于功能不复杂，而数据很复杂的逻辑块，设计先行的意义不大，可以先写代码再添加用例，也可以写代码和添加用例交替进行。

可以看出，eTDD 过程中，只有设计逻辑块的功能，算得上是测试工作。可视编程过程，是享受单元测试所带来的好处的过程。eTDD 对工具提出了很高的要求，工具必须支持直接设定逻辑块的输入输出，不用编写测试代码，且能适应各种各样的复杂需求，后文会进一步阐述。

3. 可视编程简单示例

可视编程是 eTDD 的核心。下面用一个简单示例，展示可视编程过程，所使用的工具是 C/C++单元测试工具 Visual Unit 4，简称 VU4。我们要编写一个函数，其功能是删除字符串左边空格。这个示例，函数只有一个逻辑块，且逻辑块的输入输出就是函数的输入输出，因此，逻辑块等同于函数。

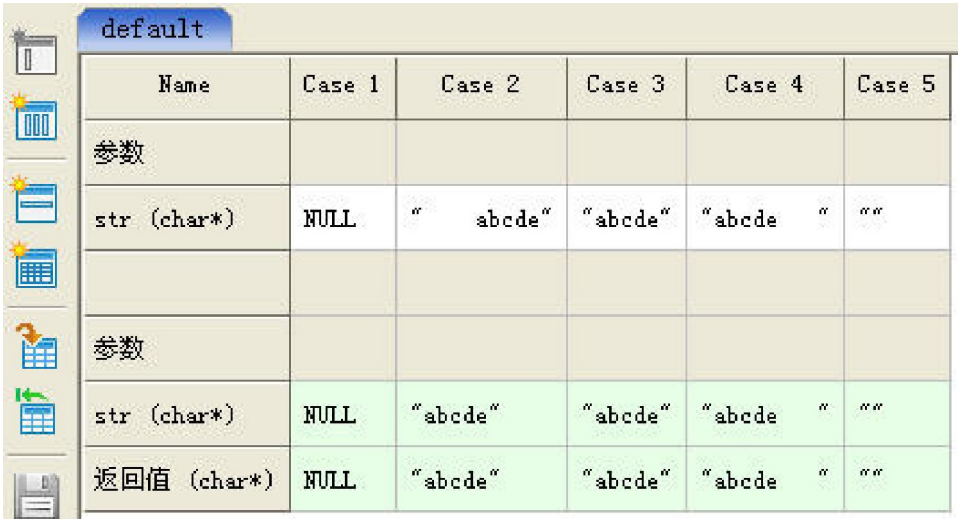
步骤一：编写函数框架

能通过编译就行：

```
char* strtrml(char *str)
{
    return str;
}
```

步骤二：设计代码功能

代码的功能，特别是逻辑块的功能，可以用“什么输入下应该产生什么输出”来表达。编码时，程序员一定需要想清楚代码要实现的功能，把“所想”用输入和输出的方式记录下来，也就完成了代码功能的设计，这就是测试用例。下图中，白色背景的单元格是输入，淡绿色背景的单元格则是输出。输出是指预期的正确计算结果，如果实际结是与预期不符，工具就会报告测试失败。



default

Name	Case 1	Case 2	Case 3	Case 4	Case 5
参数					
str (char*)	NULL	" abcde"	"abcde"	"abcde "	""
参数					
str (char*)	NULL	"abcde"	"abcde"	"abcde "	""
返回值 (char*)	NULL	"abcde"	"abcde"	"abcde "	""

步骤三，编写代码

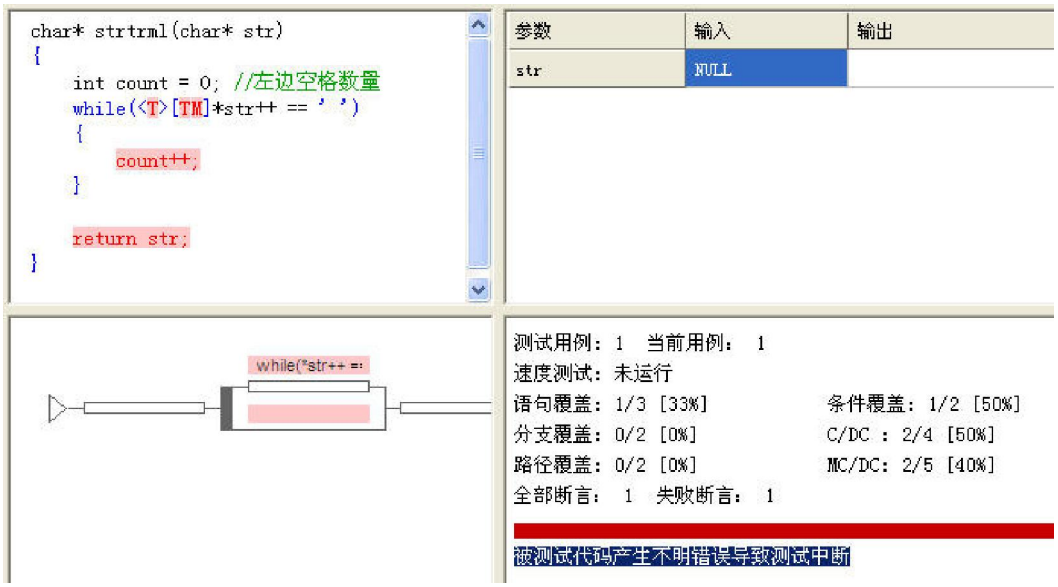
编写几行代码、就可以察看程序行为，然后修改错误、继续编写，直到测试全部通过。

假设编写strtrml()的思路是：首先计算左边空格的数量，然后再把字符串朝左边移动。先编写计算左边空格数量的代码(**粗体且带下划线的为新增代码**)：

```
char* strtrml(char *str)
{
```

```
    int count = 0; //左边空格数量
    while(*str++ == ' ')
        count++;

    return str;
}
```

每当想看看代码会做什么时，就可以编译当前源文件，通过编译后，测试自动执行，按Ctrl+Alt+空格，切换到VU4界面，显示测试结果，如下图：



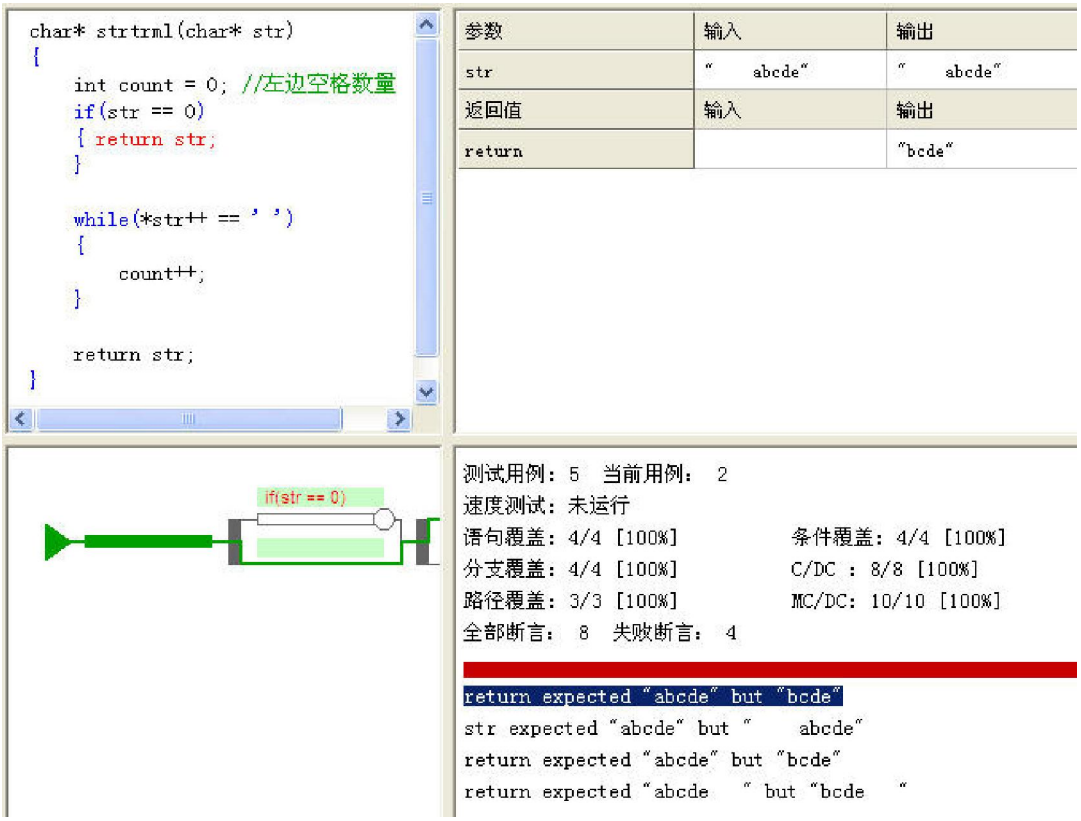
代码崩溃了！点击左下窗口红条下的错误信息，自动切换到对应的用例，看看为什么崩溃。右上窗口是输入和输出，输入为NULL。左上窗口显示当前输入下所执行的代码，其中，黑色的是当前输入下已执行的代码，红色则是未执行的代码，红色且带淡红色背景的是未覆盖代码（所有用例均未执行）。当输入为NULL时，*str++ == ''是最后的黑色代码，表示这里崩溃，显然，当输入为NULL时，*str会导致崩溃。

按ESC或Ctrl+Alt+空格，回到开发环境界面，继续编写代码，添加对NULL的判断：

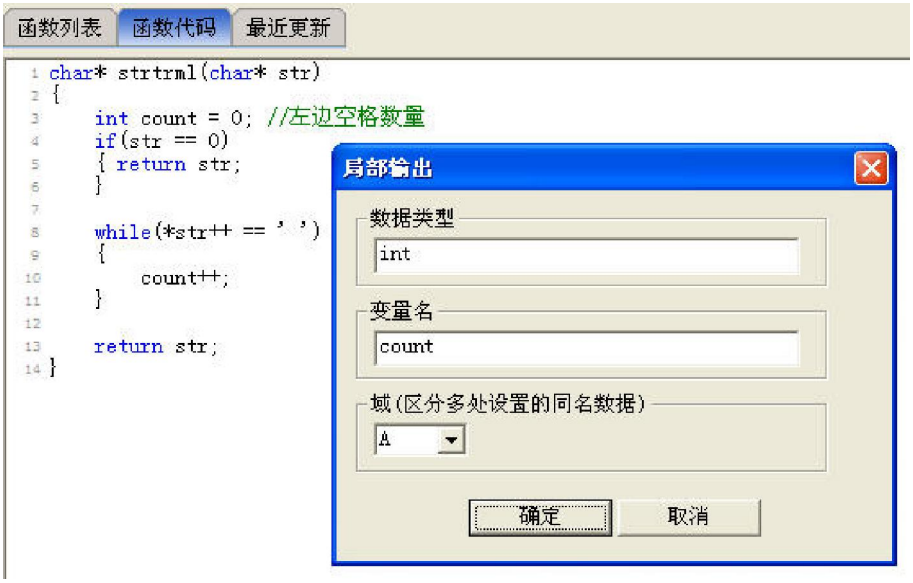
```
char* strtrml(char *str)
{
    int count = 0; //左边空格数量
    if(str == NULL) return str;

    while(*str++ == ' ')
        count++;

    return str;
}
```

再看程序行为，崩溃消失了。用右键菜单切换用例，显示用例2，即输入为左边有4个空

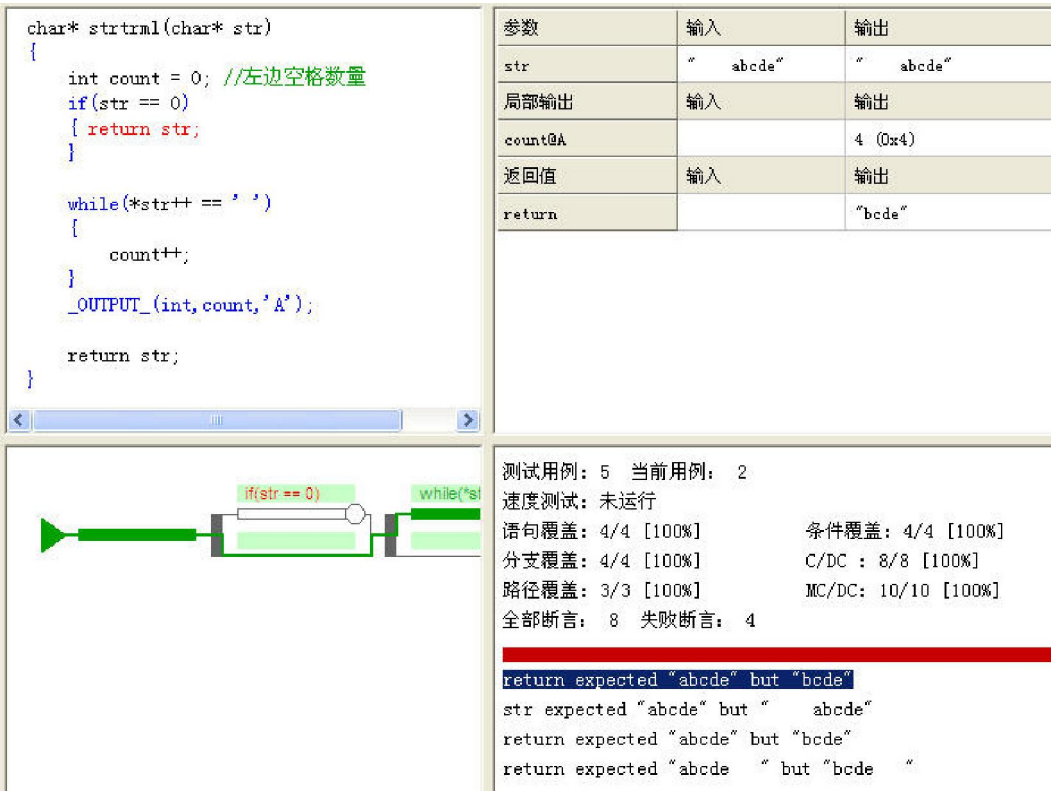
格的字符串，程序行为如下图：



如果想看看左边空格数计算结果对不对，即局部变量count的计算结果，可使用局部输出功能，方法是在VU4左边的函数代码窗口，单击输出位置，右键菜单选择“局部输出”，然后在函数代码中双击变量类型int，再双击变量名count，自动填写到局部输出界面中，如下图：



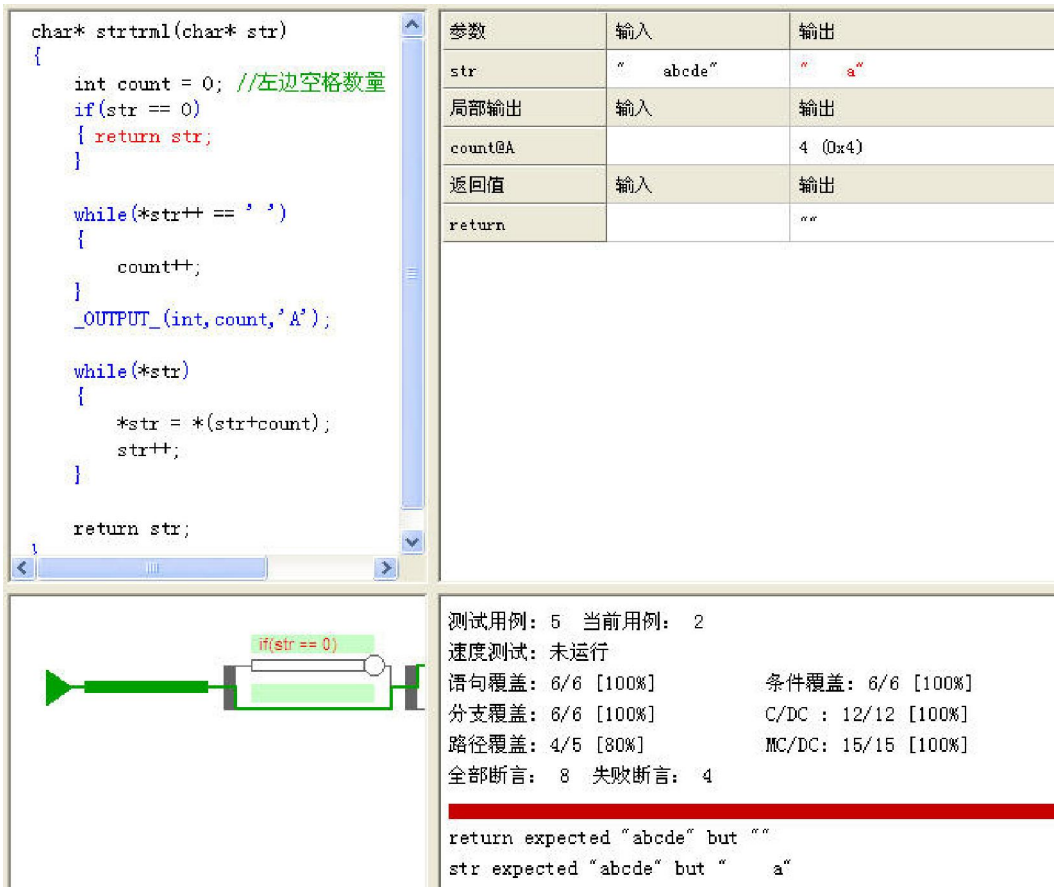
添加后，数据表格中会自动增加局部变量count，可以像返回值一样，判断它的结果是否正确。在这里，我们只是想看看它的值，不进行判断，因此，不在用例中设置它的输出值。程序行为如下图，可以看到，左边空格的计算结果是正确的。



接下来继续编写代码，把字符串朝左边移动：

```
char* strtrml(char *str)
{
    int count = 0; //左边空格数量
    if(str == NULL) return str;

    while(*str++ == ' ')
        count++;

    while(*str)
    {
        *str = *(str+count);
        str++;
    }

    return str;
}
```

现在已经完成移动了，结果对不对呢？看看程序行为：

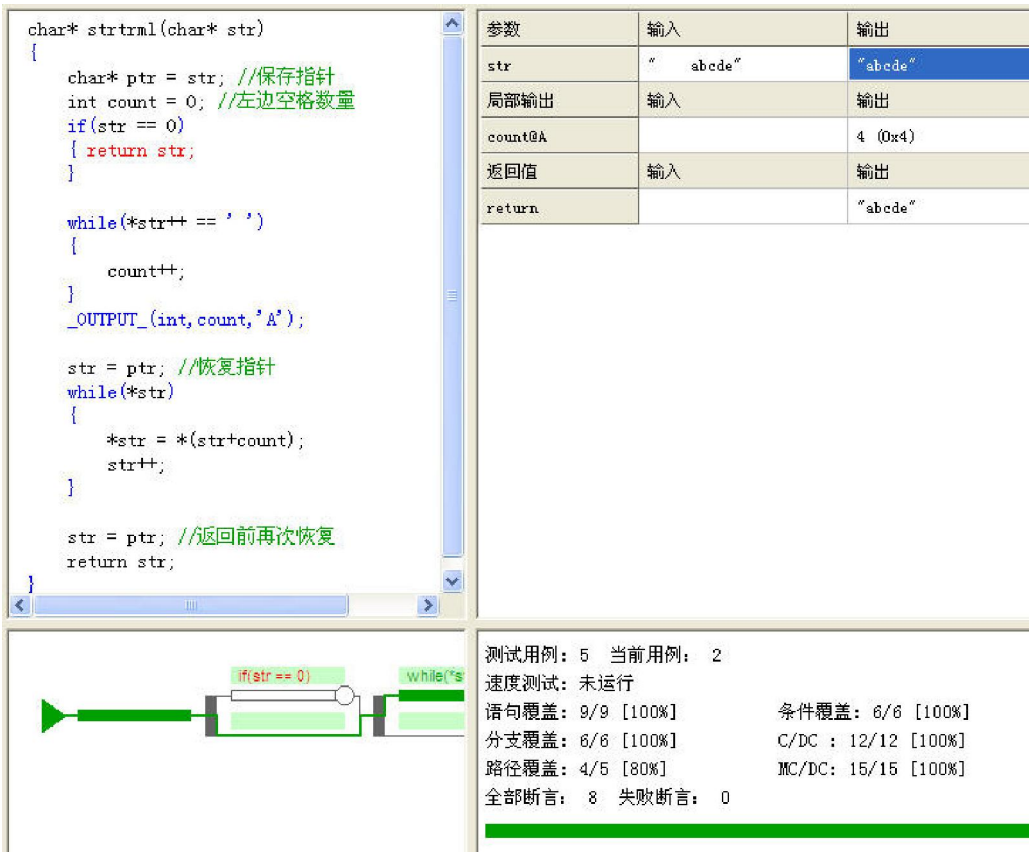


首先看数据的输出，结果完全不对，参数str和返回值都指向了一个预料之外的位置。看看代码，正常执行到最后。察看代码可以发现，在计算左边空格后，指针已经偏移，移动字符串操作的是已偏移后的指针，且再次造成了指针的偏移，结果当然不对了。所以要先把指针保存，每次偏移后恢复：

```
char* strtrml(char *str)
{
    char* ptr = str; //保存指针
    int count = 0; //左边空格数量
    if(str == NULL) return str;

    while(*str++ == ' ')
        count++;

    str = ptr; //恢复指针
    while(*str)
    {
        *str = *(str+count);
        str++;
    }

    str = ptr; //返回前再次恢复
```

```
    return str;
}
```

程序行为如下图，输出已经是我们想要的结果，并且显示绿条，表示测试全部通过，至此，编码和单元测试同步完成。



步骤四：优化代码

最好重读一下代码，看看有没有需要改进的地方。有了单元测试，可以放心大胆地修改代码。

可视编程总结

在可视编程过程中，付出了什么？获得了什么？

所付出的是设计程序功能的步骤二，即用输入输出的方式，将“所想”记录下来，这不需要多少时间，因为在编写代码前，程序员本来就需要想清楚这些，只不过用这种简单的方式记录下来而已。

所获得的主要有三点：

1）在编写代码过程中，程序行为一目了然。写代码很难一气呵成，要频繁回顾上下文，但代码与文字不同，它的真正意义在于执行起来会做什么，这是眼睛看不出来的，需要在头脑中推理。推理是隐性的但高强度的脑力劳动，可视编程省略了这种推理，减轻了头脑负担，能够比较轻松地整理思路，思路有偏差的时候，能够及时做出调整，这可以大幅提升编码效

率，并降低劳动强度。

2) 基本上免除调试。调试最花费时间。eTDD与TDD不同，首先关注的是程序行为而不是测试是否通过。很多函数，一个功能点的实现就需要大量代码，甚至代码基本写完时，第一个用例才能通过，在这个过程中，关注测试是否通过没有意义。eTDD自动执行测试，可以随时察看程序行为，在最短时间内发现错误并改正，基本上不再需要调试。

3) 测试与开发同步完成。传统开发方式下，很多程序员都将单元测试视为包袱，而eTDD使单元测试变成开着走的“车子”，而不是背着走的“包袱”。

4. 可视编程复杂示例

前面的示例，是比较低层的函数，函数本身就是一个逻辑块。对于较高层函数，如前文发短信的函数，逻辑块只是函数的一部分，可视编程和 eTDD 要面向逻辑块。

对于没有逻辑计算的代码，直接编写，例如，先编写以下代码：

```
BOOL CMyClass::_02_Response(stream_t pMsg)
{
    //解释申请信息获取用户信息
    UserInfo info;
    GetUserInfo(&info, pMsg);

    //从数据库读取职位列表
    CJobList jobList;
    GetJobList(&jobList, &info);

    //从数据库读取已发送给当前用户的职位 map
    CMapStringToPtr map;
    map.InitHashTable(17);
    GetSendedMap(&map, &info);

    //一条短信
    CString msg;

    //下面开始编写拼接短信的逻辑块

    return TRUE;
}
```

如果逻辑块的功能比较复杂，不能一下子想清楚它的功能，则先设计逻辑块的功能（设计用例）。但如果功能逻辑很简单，而数据很复杂，则可以先写代码，然后再添加用例并察看程序行为。当然，写代码和添加用例也可以交替进行。

实际上，这个例子涉及到局部输出，逻辑块的计算结果（输出），即局部变量 msg，保存拼接后的短信，在代码中分两处调用其他函数发出去，没有通过返回值之类的方式输出，对于这种数据的判断，因为涉及到判断位置，只有在相关代码存在时，才能加入用例，因此，这个例子更适合编码基本完成时再建立用例。

那么，如何设计用例呢？特别重要的是，要有面向逻辑块的思想。用例的输入是逻辑块的输入，用例的输出是逻辑块的预期计算结果。

下面的说明比较复杂，可能不容易理解。您可以简单浏览一下，只需要了解一点就行：eTDD 可以适应各种复杂情形，构建用例的过程，只是在表格中把逻辑块的输入和预期的正确输出记录下来就行，不用编写测试代码。

逻辑块要做的，是从 jobList 读取一系列数据并拼接字符串，拼接前要检索映射表 map，

如果数据已存在则跳过。jobList 的类型是 CJobList，一个结构指针链表，定义如下：

```
typedef struct tagJobInfo
{
    CString sort;
    CString title;
    CString content;
}JobInfo;
typedef CList<JobInfo*, JobInfo*> CJobList;
```

map 保存的是以 title 为 key 的 JobInfo 的指针。

具体来说，逻辑块的输入，是保存在 jobList 中的一系列 JobInfo 指针，以及保存在 map 中的 JobInfo 指针，输出是由一系列 JobInfo 指针中的 title 拼接成的字符串。看起来非常复杂，实际上，拼接字符串，用到的是 JobInfo 中的 title，map 中也是以 JobInfo 中的 title 为 key，所以，用例的输入只是一些字符串，输出 msg 也是一些字符串。其他不涉及逻辑计算的数据，是不必考虑的。

假如我们先编写逻辑块的代码，代码如下：

```
BOOL CMyClass::_02_Response(stream_t pMsg)
{
    //解释申请信息获取用户信息
    UserInfo info;
    GetUserInfo(&info, pMsg);

    //从数据库读取职位列表
    CJobList jobList;
    GetJobList(&jobList, &info);

    //从数据库读取已发送给当前用户的职位 map
    CMapStringToPtr map;
    map.InitHashTable(17);
    GetSendedMap(&map, &info);

    //一条短信
    CString msg;

    //下面开始编写拼接短信的逻辑块
    POSITION pos = jobList.GetHeadPosition();
    while(pos != NULL)
    {
        JobInfo* pJob = jobList.GetNext(pos);
        //已发送的不再重发
        if(map.Search(pJob->title)) continue;

        int len = msg.GetLength() + pJob->title.GetLength() + 1;
```

```
            if(len > MSG_MAX) //超长，发送前面的
            {
                SendMsg(msg, &info);
                msg.Empty();
            }

            //拼接短信
            msg += ',';
            msg += pJob->title;
        }

        if(!msg.IsEmpty()) //最后一条短信
            SendMsg(msg, &info);


        return TRUE;
    }
```

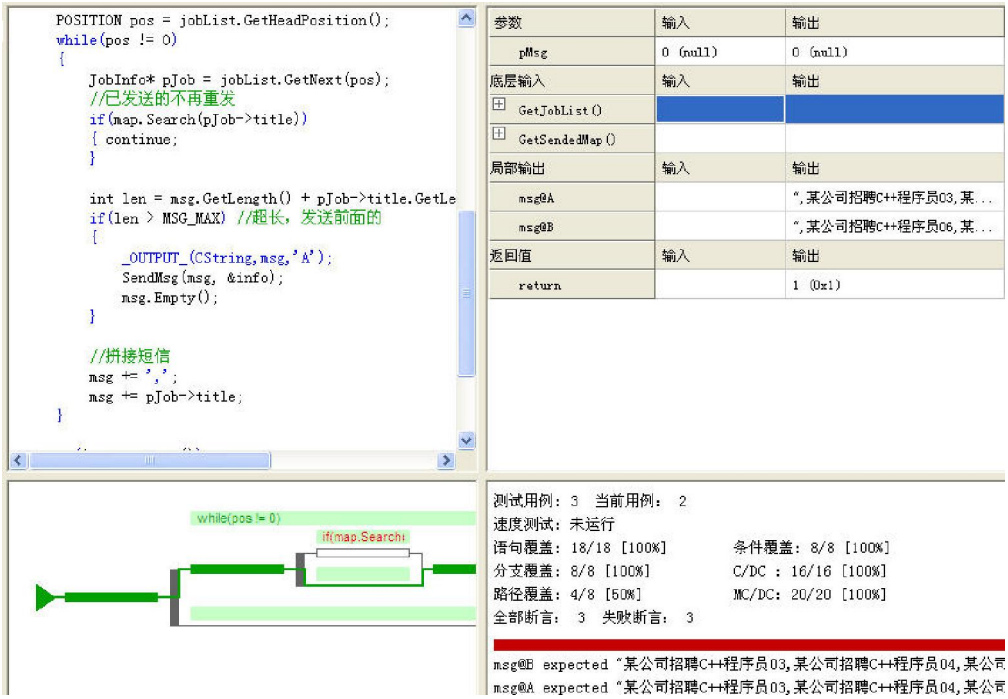
然后建立用例，如下图：

default

Name	Case 1	Case 2	Case 3
底层输入			
GetJobList() jobList[0]->title...	"某公司招聘C++程序员01","某公司...	"某公司招聘C++程序员01",...	
GetSendedMap() map[0]->key (LP...	"某公司招聘C++程序员01","某公司...	"某公司招聘C++程序员01",...	
GetSendedMap() map[0]->ptr->ti...	...[key]		
局部输出			
msg@A (LPCTSTR : CString)	""	"某公司招聘C++程序员03,...	
msg@B (LPCTSTR : CString)	"某公司招聘C++程序员03,某公司招...	"某公司招聘C++程序员06,...	

数据都是一些字符串，其实并不复杂，但可能难以理解：这些字符串如何能构造出 jobList 中的是一系列 JobInfo 指针以及 map 中的 JobInfo 指针呢？这里涉及到底层输入、回调赋值、序列赋值、局部输出等技术，这些技术是 eTDD 工具的基本技术，在后文再进一步介绍。

下图是测试结果，可以看出，字符串开头多了一个逗号，再看代码，拼接字符串时未判断 msg 是否为空，直接就添加了逗号，这就是错误所在，把 msg += ',';改为 if(!msg.IsEmpty()) msg += ',';，测试通过。



```
POSITION pos = jobList.GetHeadPosition();
while(pos != 0)
{
    JobInfo* pJob = jobList.GetNext(pos);
    //已发送的不再重发
    if(map.Search(pJob->title))
    { continue;
    }

    int len = msg.GetLength() + pJob->title.GetLe
    if(len > MSG_MAX) //超长，发送前面的
    {
        _OUTPUT_(CString,msg,'A');
        SendMsg(msg, &info);
        msg.Empty();
    }

    //拼接短信
    msg += ',';
    msg += pJob->title;
}
```

参数	输入	输出
pMsg	0 (null)	0 (null)
底层输入	输入	输出
⊞ GetJobList()		
⊞ GetSendedMap()		
局部输出	输入	输出
msg@A		",某公司招聘C++程序员03,某...
msg@B		",某公司招聘C++程序员06,某...
返回值	输入	输出
return		1 (0x1)

测试用例：3　当前用例：　2
速度测试：未运行
语句覆盖：18/18 [100%]　　条件覆盖：8/8 [100%]
分支覆盖：8/8 [100%]　　C/DC ：16/16 [100%]
路径覆盖：4/8 [50%]　　MC/DC：20/20 [100%]
全部断言：　3　失败断言：　3

msg@B expected "某公司招聘C++程序员03,某公司招聘C++程序员04,某公司
msg@A expected "某公司招聘C++程序员03,某公司招聘C++程序员04,某公司

5. eTDD 之关键技术

eTDD 由工具完成苦活、脏活、重活，用户要做的，只是用记录输入输出的方式，设计逻辑块的功能。eTDD 的关键技术，也是 eTDD 工具应该具备的功能。工具至少要具备以下功能，才能顺利实施 eTDD：表格驱动、直接设定底层输入/局部输入/局部输出、覆盖统计与标示、找出遗漏用例、描述程序行为、统计最近更新的函数、生成测试报告。

5.1 表格驱动

应用 eTDD，用户要做的，只是用记录输入输出的方式，设计逻辑块的功能。如何记录才最简单？当然是填表格。表格驱动是 eTDD 工具的最基本要求。

表格的两种模式

表格要能适应复合数据类型、指针、数组，又要便以构建大量用例，因此，要有树表模式，用于展开复合类型，还要有表格模式，便于构建大量用例。下图是 C/C++单元测试工具 Visual Unit 4 的数据表格界面，其中，表格模式只显示填写了数据的行，忽略对测试无意义的行。

default

Name	Case 1	Case 2	Case 3
case desc	用例数据可用//或/**/添...		
参数			
p_str (const char*)		"abc@kailesoft.com"	"ab.c@kailesoft.com"
size (size_t)		MAX_MAIL_LEN/*支持宏...	.../用...表示使...
底层输入			
malloc() return (void*)	NULL//用NULL表示空指针...		
返回值 (Mail*)			
return->mail (char*)		"abc@kailesoft.com"	"ab.c@kailesoft.com"
return->name (char*)		"abc"	"ab.c"
return->server (char*)		"kailesoft.com"	"kailesoft.com"

表格模式

处理难以表格驱动的数据

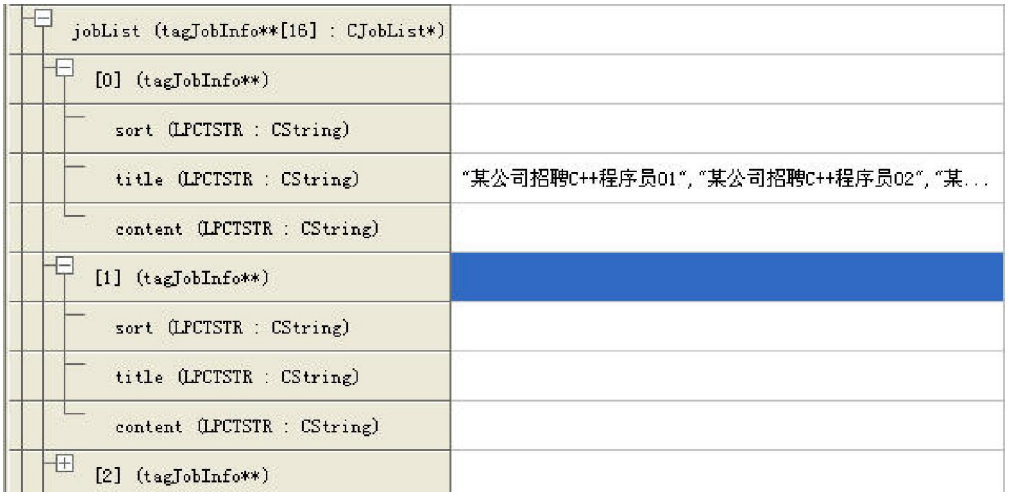
有一些数据不适合表格驱动，例如，前文发短信的函数，输入是保存在链表和映射表的结构对象指针系列，如何做到表格驱动？为了解决这个问题，工具还必须提供接口，将不便于表格驱动的数据，转换为易于表格驱动的数据，例如，链表不便于表格驱动，但数组则便于表格驱动，通过转换接口，在表格中填写数组来构造或判断链表。C/C++单元测试工具 Visual Unit 4 就提供了这个功能，称为回调赋值。

常用数据集合的回调赋值，如链表、映射表，一般由工具自带，可以直接使用。

用户可以添加自定义的回调赋值，简化数据的构造过程，例如，CList<CToken*, CToken*>是一个由 C 语言代码字符串解释而来的 token 系列，如果直接设置很多个 token，比较麻烦，可以利用回调赋值，调用项目中的现有代码，将字符串解析为 token 系列，或将 token 系列拼接成字符串，这样，在表格中直接填写字符串就可以构造或判断一个 token 系列。

简化数据集合的填写

另一个功能对于简化表格数据很重要，称为序列赋值，其功能是，在数组的第一项中填写一系列数据，由工具自动分配到数组的各项，这样，只需要一两行数据，就可以构造或判断整个数组，如下图，title 填写了用逗号分隔的一系列字符串，相当于在数组的第一项填写”某公司招聘 C++程序员 01”，第 2 项的同样位置填写”某公司招聘 C++程序员 02”，其余类推。



表格驱动的综合应用

前文介绍的发短信函数，其测试数据使用了回调赋值、序列赋值，因此，只需要一行数据就构造出链表 objList 中的一系列对象，用户不需要编写一行代码。关于回调赋值和序列赋值的更详细信息，请查看 Visual Unit 4 的帮助文档。

表驱动是 eTDD 工具的基本要求，有了表格驱动功能，用户就不需要编写和维护测试驱动代码，这样，单元测试的一大部分成本的就被消灭了。

5.2 底层输入

什么是底层输入？

底层输入，就是调用底层函数获得的输入。例如：

```
/*解析一个Email
参数: p_str，一个Email 的完整字符串
      size 缓冲区大小
返回: 解析得到的Email 对象指针*/
Mail* _01_read_mail_one(const char* p_str, size_t size)
{
    //其他代码省略
    Mail* p_mail = (Mail*)malloc(sizeof(Mail));
    if(p_mail == NULL)
        return NULL;

    //其他代码省略
    return p_mail;
}
```

p_mail 是调用底层函数 malloc()获得的一个数据，数据的值为 NULL 或一个有效内存地址，如果改一下代码，p_mail 改为由参数传递，如下：

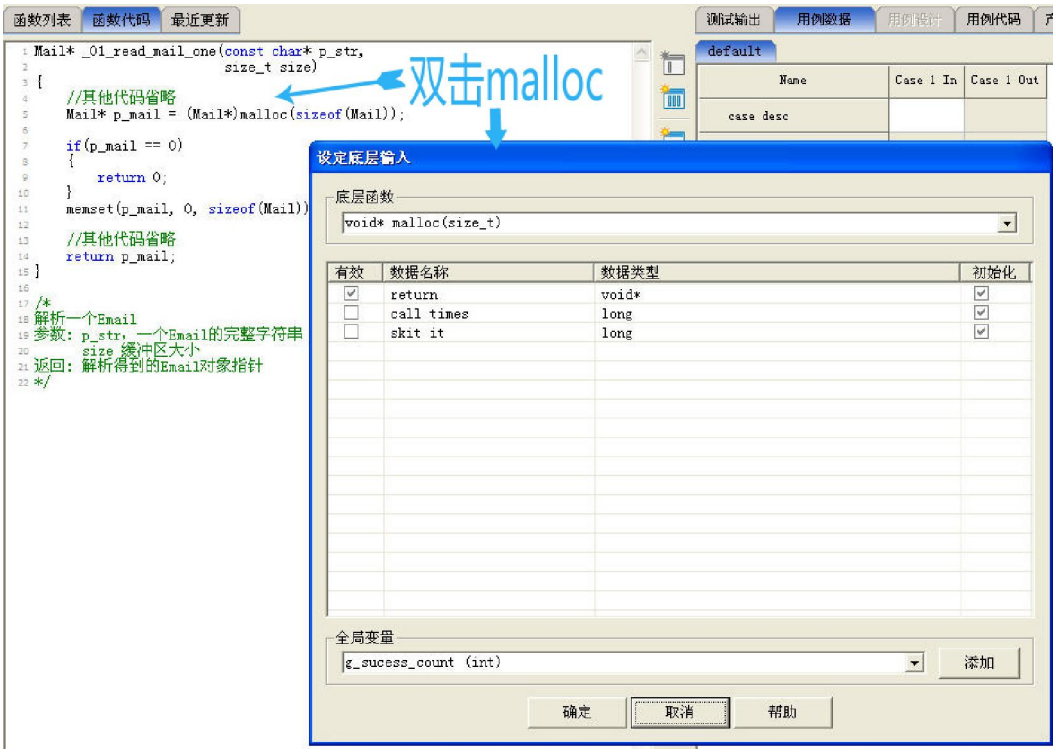
```
Mail* _02_read_mail_one(const char* p_str, size_t size,  Mail* p_mail)
{
    //其他代码省略
    if(p_mail == NULL)
        return NULL;
    //其他代码省略
    return p_mail;
}
```

两个函数的功能逻辑完全一样的，由此可以看出，调用底层函数获得的数据，与参数传递的数据并无本质区别，也是一种输入，称为底层输入。

底层输入概念是单元测试技术的重要进步，从前，代码之间的耦合关系是单元测试很困难的主要原因，一般通过编写桩代码或使用模拟对象来解决，大幅增加工作量，也造成了测试思维上的混乱。桩函数还造成底层函数产生的数据与其他数据不在一处管理，增加了维护成本；而模拟对象通常需要在产品代码中增加中间层，带来复杂臃肿的架构。

底层输入设置示例

eTDD 工具必须支持直接设置底层输入。下图是 C/C++单元测试工具 Visual Unit 4 的底层输入设置过程：



点击确定，设为有效的变量就会加入表格，如下图：

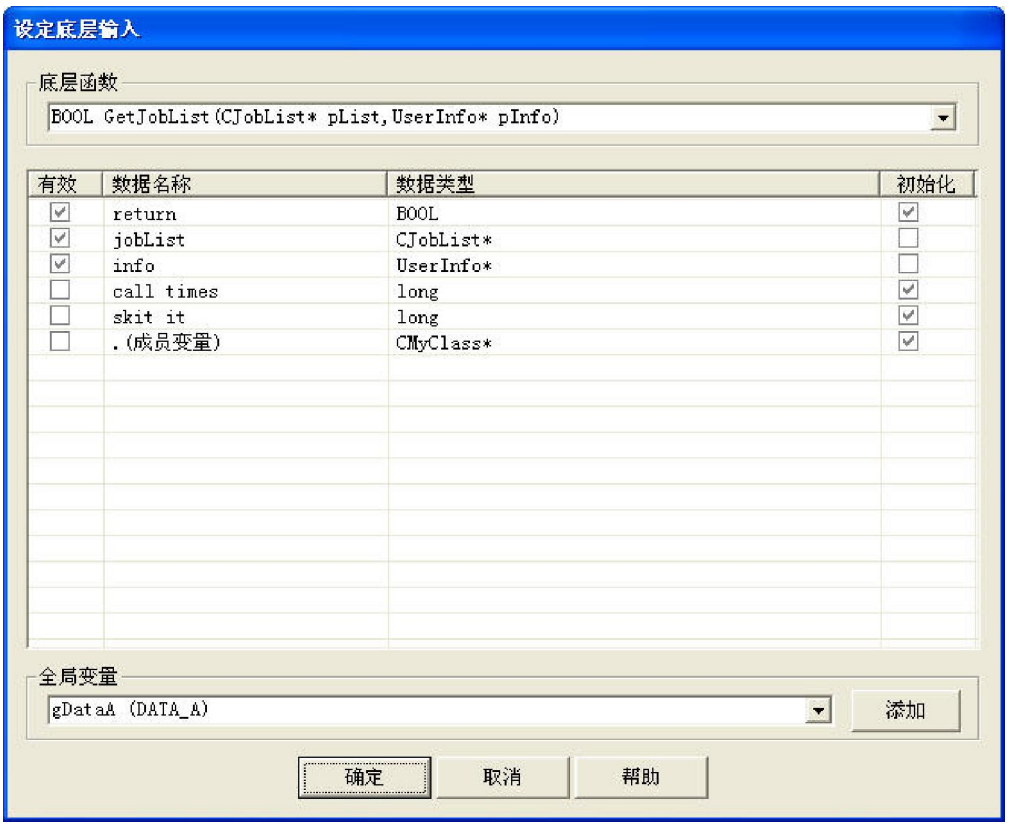
default

Name	Case 1 In	Case 1 Out	Case 2 In	Case 2 Out
case desc				
参数				
p_str (const char*)				
size (size_t)				
底层输入				
malloc()				
return (void*)	NULL			

在第一个用例，malloc()的 return 设为 NULL，第二个用例不填，这样，malloc()在第一个用例返回 NULL，第二个用例则正常申请内存。

底层输入不仅仅是底层函数返回值

底层函数产生的数据，除了返回值外，还可能保存到出参、成员变量、全局变量，可以一次性加入表格，如下图，设为有效的变量都会加入表格。call times 和 skit it 是两个特殊变量，前者用于判断底层函数的调用次数，后者用于直接跳过底层函数，可以视需要选择使用。底层输入变量加入表格后，就与其他变量完全一样处理了。



5.3 局部输入与局部输出

什么是局部输入？

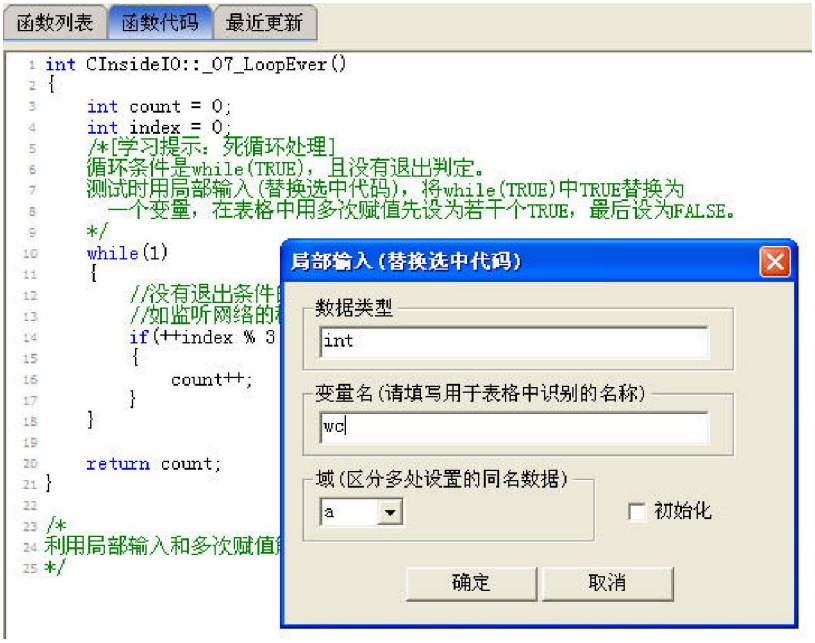
局部输入，是在被测试代码执行过程中，设置变量、表达式的值，用于解决一些特别的测试问题。例如，下面的代码会造成死循环，如何测试？

```
/*
利用局部输入和多次赋值解决死循环
*/
int CInsideIO::_07_LoopEver()
{
    int count = 0;
    int index = 0;
    while(TRUE)
    {
        /*没有退出条件的死循环通常用于永续运行的代码，
          如监听网络的程序。这里只用简单代码代替*/
        if(++index % 3 == 0)
            count++;
    }

    return count;
}
```

类似的情形还有很多，例如：局部静态变量、调用界面函数读数据、嵌入式测试中模拟中断对全局变量的修改、缓冲区常量太大希望换成一个可在用例中设置的变量，等等。

局部输入设置示例

下面是 C/C++单元测试工具 Visual Unit 4 设置局部输入的过程。



如上图，将 while 条件设为一个变量，因为 while 条件本身不是变量，因此，需设一个变量名，用于表格中的识别，如果设置的是已经存在的变量，则变量名应该是原有的变量名，这种情形下，可以通过双击代码中的变量类型和变量名，自动填写。点击确定后，变量加入表格。

前述设定内部输入使用的命令是“局部输入(替换选中代码)”，将 while 条件替换为一个变量。还可以使用替换”=”右边、替换赋值语句、插入赋值语句等方式。对于模拟中断对全局变量的修改，则使用“模拟中断”命令。

多次赋值

现在还有一个问题，变量 wc 应该设为什么值呢？设为 0 则不进入循环，设为 1 则仍然是死循环。这里，要用到另一种赋值方式：多次赋值，即在一个用例中，代码会被多次执行，每次执行设置不同值。Visual Unit 4 采用分号分隔多个值来实现多次赋值，如下图。用例 1，循环执行三次后退出，用例 2，循环执行 6 次后退出。多次赋值除了应用于局部输入，还应用于底层输入和局部输出。

Name	Case 1 In	Case 1 Out	Case 2 In	Case 2 Out
case desc				
⊞ 成员变量 (CInsideIO*)				
局部输入				
wc@a (int)	1;1;1;0		1;1;1;1;1;1;0	
返回值 (int)		1		2

局部输出

局部输出是指在被测试代码执行过程中，判断变量或表达式的实时值，其设置方法与局部输入类似，在可视编程示例部分已有说明，这里不作重复。设置局部输入与局部输出时，

不会修改产品代码。

内部数据与面向逻辑块

底层输入、局部输入、局部输出，统称为内部数据。逻辑块的输入输出，除了参数、返回值、成员变量、全局变量外，还经常用到内部数据。能够直接设置或判断内部数据，面向逻辑块才成为可能。支持直接设定内部数据，也是 eTDD 工具的基本要求。只要内部数据可以直接设定，“可测性”问题就不再存在。

面向逻辑块的测试思维，使测试变得很简单：一是思路变得很简单，无论多复杂的函数，只要分析它有哪些逻辑块，逻辑块的输入输出是什么，就可以轻松设计用例。如果没有逻辑块，则可用空数据直接完成覆盖。如果有多个逻辑块，则分别测试，Visual Unit 4 支持一个函数多个数据表格，可以每个逻辑块对应一个表格；二是数据简单，功能逻辑计算所涉及的数据，通常是简单的数据。三是数据直接设置，不需要绕很多弯子。

5.4 覆盖统计与标示

覆盖指标

覆盖统计用于衡量测试的完整性，以及找出遗漏用例。

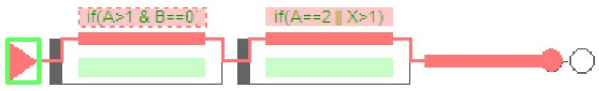
常用的覆盖统计有：语句、条件、判定、MC/DC、分支、路径。

其中，MC/DC 是欧美航空标准，要求每个条件独立影响判定的结果，实际上已包含语句、条件、判定覆盖。分支覆盖如果不考虑空分支，则相当于判定覆盖，而空分支的覆盖意义不大。路径覆盖的意义依赖于代码结构，如果一个函数中包含不相干的两个或多个逻辑块，则会组合出无意义的海量路径。综上所述，单元测试覆盖的推荐指标是 MC/DC，应该完成 100%的 MC/DC 覆盖。

覆盖标示

工具应该自动统计覆盖状况，并清晰标示。下图是 Visual Unit 4 的覆盖统计标示，其中，<T>/<F>表示未覆盖的判定的真/假值，[T]/[F]表示未覆盖的条件的真/假值，[M]表示未覆盖的 MC/DC。红色且带淡红色背景的代码是未覆盖语句；条件带淡红色背景的分支是未覆盖分支；用红色画出的路径是未覆盖路径。

```
int CWhiteBox::_01_WhiteBox(int A,int B,int X)
{
    if(<T>[TM]A>1 && [TFM]B==0)
    {
        X = X/A;
    }
    if(<T>[TM]A==2 || [TM]X>1)
    {
        X = X+1;
    }

    return X;
}
```

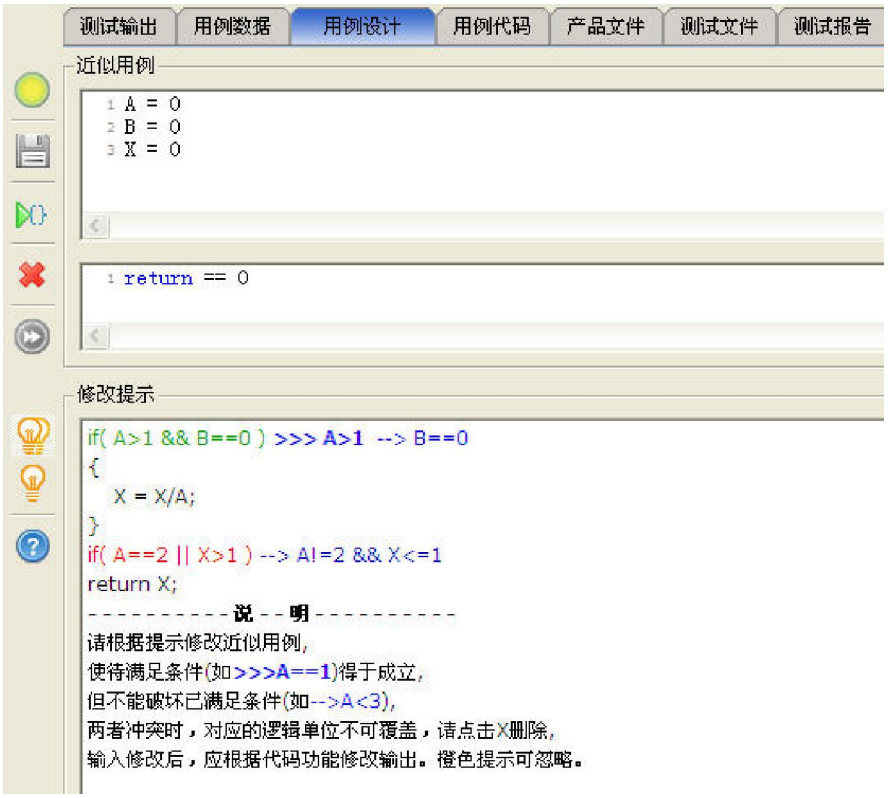



```
测试用例：1  当前用例：  1
速度测试：未运行
语句覆盖：1/3 [33%]          条件覆盖：3/8 [37%]
分支覆盖：2/4 [50%]          C/DC ：5/12 [41%]
路径覆盖：1/4 [25%]          MC/DC：5/16 [31%]
全部断言：  1  失败断言：  0
```

5.5 找出遗漏用例

完成覆盖很难是单元测试成本高的原因之一，所以也是 eTDD 要解决的问题。eTDD 工具应该具有找出遗漏用例的功能，帮助用户快速找出遗漏用例实现高标准覆盖，一般的要求是完成如 100%的 MC/DC。

一种找出遗漏用例的思路是，针对一个未覆盖的逻辑单位（如某条件的真值），由工具从现有用例中计算出近似用例，并提供修改提示，用户按提示修改输入，并依功能需求修改输出，就可以完成覆盖。C/C++单元测试工具 Visual Unit 4 实现了这个功能。如下图，根据修改提示，只要将 A 改为大于 1 的数，并修改输出，就可以完成覆盖。找出遗漏用例的功能，使完成高标准的覆盖很简单，这是 eTDD 要求完成 100%的 MC/DC 的原因。



5.6 其他技术

描述程序行为

描述程序行为是支持可视编程的关键。单元测试的输出应该自动显示什么输入下，执行了哪些代码，产生了什么输出，前文已有展示，这里不重复。

统计最近更新的函数

统计最近更新的函数是一个小功能，但对 eTDD 来说很重要。eTDD 过程是：逻辑块可视编程；提交前完成覆盖；只进行粗略调试。提交前，要查看最近编写或修改过的函数的测试状态，以便完成覆盖。下图是 C/C++单元测试工具 Visual Unit 4 的最近更新列表，黄灯表示覆盖不完整，红灯表示有错误，红黄灯表示有错误且覆盖不完整，绿灯表示无错误且覆盖完整，没有灯的表示未测试。

生成测试报告

eTDD 工具当然应该支持自动生成测试报告。生成测试报告不需要什么高技术，报告的内容无非就是一些测试统计数据，因此，这里不作详细阐述。

6. eTDD 首要效益：保证代码质量

与 TDD 一样，eTDD 的首要效益是保证代码质量。保证代码质量具有多方面的含义，很多 TDD 相关文档均有阐述，这里只作简单归纳。

改进代码的整体结构

程序员边开发边测试，自然会将代码写得易以测试，自动改进代码整体结构，例如：函数规模趋于合理、函数功能趋于单纯。eTDD 不要求代码的可测性，不强制重构，但基于趋易避难的本能，eTDD 仍然会推动程序员编写更易以测试的代码，自然地改进代码整体结构。eTDD 不需要 mock，不需要仅仅为了单元测试而增加中间层，因此，只有正面的改进，没有负面的影响。

保证功逻辑的正确性

只有单元测试，能够做到完整地测试代码的功能逻辑。

避免开发周期失控

开发周期失控，很大程度上源于后期的调试排错不可控。只有一次性编写合格代码，才能避免开发周期失控。华为有句名言：“我们没有时间把事情一次做好，却总有时间把事情一做再做”，对于编程来说，边开发边测试是把事情一次做好的唯一途径。

提升代码的可维护性

一般企业，都不会去编写函数级的详细文档，而单元测试数据，就是最好的函数级文档，即使人员发生了很大变化，新加入的工程师仍然可以便利地维护老代码。

回归测试使开发过程适应频繁变化的需求

回归是指代码修改后，原有的功能不致被破坏，有了单元测试，代码修改后只需重新执行测试，就可以验证原有功能是否被破坏，这使开发过程可以适应频繁变化的需求。

小结

TDD 将会让代码质量上一个档次，而 eTDD 则会比 TDD 再上一个档次，原因是 eTDD 工具解决了可测性问题，测试不留死角，且 eTDD 工具使高标准的覆盖很容易做到，测试完整性远高于 TDD。

7. eTDD 附加效益：让编程产能翻二番

7.1 小实验：编程产能翻二番

我们的小实验数据

我们进行了一些 eTDD 与传统方式对比小实验，得到的数据是：eTDD 的编程产能大约是传统方式的 4 倍。

实验过程：由两个程序员 X 和 Y，交叉编写两个函数 A 和 B，即程序员 X 用传统开发方式写函数 A，用 eTDD 写函数 B，程序员 Y 则相反。交叉编写的目的是尽量减少程序员编程能力、代码复杂度等差异造成的误差。下表是数据记录模板：

程序员	编程方式	实验函数	编码时间	即时调试	后期调试	总时间
X	传统方式	A				
	eTDD	B				
Y	传统方式	B				
	eTDD	A				

对于 eTDD 方式，只需要记录总时间。对于传统方式，则记录三个时间：

编码时间：编写代码，直到自己认为代码功能已全部实现，且编译通过所用的时间；

即时调试：编码后调试程序的时间；如果编码和调试交叉进行，则只记录编码时间。调试环境在编码前预先准备。

后期调试：对代码进行测试，如果发现了错误，进行调试排错使测试通过所用的时间。后期调试时间不包括测试时间，测试由他人完成，例如，程序员 X 用 eTDD 方式编写函数 B（这个过程已经设计了测试用例），程序员 Y 用传统方式编写函数 B，程序员 Y 写完代码并认为调试通过后，将代码拷贝给程序员 X 进行测试，如果测试通过，则后期调试时间为 0，否则，告知程序员 Y 在什么数据下测试失败及错误信息，程序员 Y 进行调试排错，并交程序员 X 再次测试，重复此过程直到测试通过，仅记录程序员 Y 的调试排错时间，不包括沟通、传递代码及测试时间。

上面是一对函数的实验过程。为了取得比较准确的数据，一组实验要针对 3-5 对函数，然后计算总时间。

下面是我们进行的一组实验结果数据。实验对象为 Visual Unit 4 示例代码中，文件 _2Y_SendMail.c 的 10 个函数，第一位程序员用 eTDD 编写函数 1、3、5、7、9，用传统方式编写函数 2、4、6、8、10，第二位程序员用 eTDD 编写函数 2、4、6、8、10，用传统方式编写函数 1、3、5、7、9。时间单位为分钟。

编程方式	编码时间	即时调试	后期调试	总时间	比值
传统方式	326	503	347	1176	4.32
eTDD	272			272	

分析：

eTDD 总时间少于传统方式的单纯编码时间，似乎很不合理，因为 eTDD 过程，包括了设计用例、排错的过程，但实际上是合理的，因为：设计用例实际上是设计代码的功能，设计可以促进编程，费时不多而效果显著；更重要的是，程序行为可视，就像黑暗中有了一盏明灯，减少了思维的滞塞，可以比较流畅地编写代码。这是对有一定复杂度的代码而言，如果逻辑简单，编码过程只是敲键盘，那 eTDD 的优势就发挥不出来。

传统方式的即时调试和后期调试，时间均多于编码时间，两者的和是编码时间的 2.6 倍。这个倍数其实比实际项目要小，因为实际项目的调试，涉及的范围要大得多。

邀请您也做做小实验

您大概不会相信上述实验结果，但是不要紧，这些实验谁都可以做。方法是：

1) 下载安装 Visual Unit 4，打开示例工程，如下文件中的代码均未编写，可以作为实验标的：_1Y_Begin.c、_2Y_SendMail.c、_3Y_MyClass.cpp、_8Y_String.c。所有函数前面均有函数功能说明。

2）选定 3-5 对函数，由两位程序员，用前面介绍的交叉方式编写函数代码，并记录时间。

3）用传统开发方式时，可以使用 eTDD 工具以外的任何工具，包括不支持 eTDD 的商业单元测试工具或开源单元测试框架。

4）实验前要先掌握相关工具的基本使用，学习时间不应计入编程时间。

5）实验主要目的是取得传统开发方式与 eTDD 所用的时间比值，此比值受很多因素影响，其中最主要的有：

代码的复杂度：复杂度越高，比值越大，即 eTDD 的优势越大；

对 eTDD 的熟悉程度：如对 eTDD 工具和过程还不熟悉，会使比值偏小；

bug 的偶然性：在传统方式下，一个 bug，就可能造成长时间的调试，而多数 bug 的产生具有偶然性，因此，如果实验代码较少，比值有可能严重偏大或偏小。

7.2 实际开发实验：编写合格代码的产能达 400 行/天

开发任务

C/C++单元测试工具 Visual Unit 的插装模块的重新开发。

任务简述

原插装模块是 Visual Unit 1.0 时代开发的，已有 10 年历史，由于 VU1 缺少代码解析功能（即缺少预处理、词法分析、语法分析），插装模块基于对被测试代码的字符串分析，VU2 增加了对被测试代码的解析功能，但插装模块并未改写。随着 VU4 应用范围的扩大，原插装模块对一些特殊代码的处理有缺陷，必须改进。原插装模块只有一个类，逻辑极复杂，难以修改，所以，需编写新的插装模块，将插装过程改为基于词法分析和语法分析。

eTDD 实验的目标是取得单纯编码的效率方面的数据，实验任务最好不涉及难以度量的工作，例如新技术的研究，需求的变更。本任务需求明确，不涉及新技术的研究，但代码需要全部重新编写（实现方式与原版完全不同，原版的代码基本不能使用），因此，适合作为 eTDD 实验任务。

实验结果

代码行：新开发的插装模块，共 18 个类，约 2500 行，不算头文件；相关模块新编写的代码约 600 行，总行数约 3100 行。

用时：53 小时（包括集成后排除缺陷），按每天 7 小时计算，约 7.5 天。

产能：约 400 行/天。

结果分析

众所周知，以代码行计算产能并不科学。上述 400 行/天的产能，是有条件的：需求明确，没有很难的技术难点，只是单纯的编写代码。遗憾的是，缺少准确的对比数据，我们估计，如果用传统方式开发，考虑后期排除缺陷的时间，编写合格代码的产能不会超过 100 行/天。这个任务，技术上难度不大，但代码的功能逻辑比较复杂，传统开发方式下，调试时间会是编码时间的几倍，而 eTDD 省略了大部分调试。

您也可以用实际开发任务做实验

一、向凯乐软件申请 Visual Unit 4 的免费试用 License。

二、选择需求明确、不涉及新技术研究或者已完成前期研究的模块做实验。

7.3 eTDD 能缩短多少开发周期？

由于程序员的产能难以精确度量，且各个企业、各个项目有很大的差异性，上述实验数据并不具通用性，不能作为定量依据，但完全可以作为定性参考。

无论是小实验还是实际任务实验，得到的结果都是：eTDD 可以让编程产能翻二番。请注意，这是指单纯编写合格代码的产能。程序员真正用于编码和调试的时间往往只占工作时间的三分之一，更多时间用于对技术难点的研究、查阅文档、沟通、开会、写文档、发呆等等，这些时间，TDD 或 eTDD 不能产生作用，另外，eTDD 对没有逻辑计算的代码的编写，也没有驱动效果。所以，不能认为，原来四个月开发的项目，用 eTDD 一个月就行了，那

是绝对不可能的。

一般来说，如果没有大规模的需求变更，eTDD 缩短开发周期 30%是可以预期的。这里说的开发周期并不是“计划”周期，而是“实际”周期，例如，计划三个月完成的项目，实际周期往往是六个月，如果用 eTDD，则可以四个月完成。缩短开发周期只是 eTDD 的次要效益，eTDD 的首要效益，是保证代码质量，以及提升开发过程管理水平。