1单元测试

你的RP是由你的程序质量决定的。

——无名人士

软件开发讲究团队合作，那就会出现一个人写的模块被另一个人写的模块调用的情况。很多误解、疏忽都发生在两个模块之间。如何能让自己写的模块尽量无懈可击？单元测试就是一个很有效的解决方案。

1.1 用VSTS写单元测试

例子：我们写一个比较常用的类型，看看它的单元测试应该怎么写？比如在各种网站应用程序中都会用到的“用户”这一类型。看一下下面的代码：

创建了一个C#的类库（Class Library），并写了如代码清单1-1的代码：

代码清单1-1

namespace DemoUser

{

public class User

{

public User(string userEmail)

{

m\_email = userEmail;

}

private string m\_email; //user email as user id

}

}

好，现在右键选中User，就可以看到“Create Unit Tests”的菜单，这样就可以创建新的单元测试（如图1-2所示）。

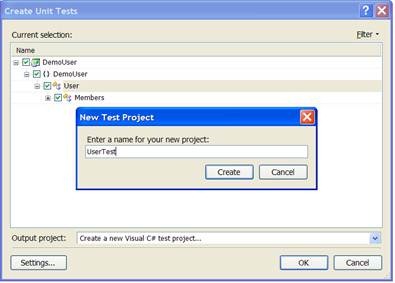
[](http://images.cnblogs.com/cnblogs_com/xinz/201111/201111201051279403.jpg)

图1-2 创建单元测试项目

创建单元测试后，注意到在Solution Explorer中出现了三个新的文件（如图1-3所示）。

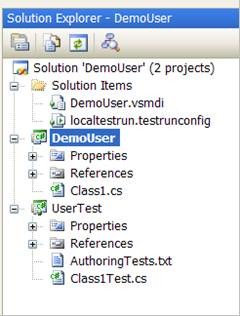
[](http://images.cnblogs.com/cnblogs_com/xinz/201111/201111201051294679.jpg)

图1-3 新的单元测试文件

Class1.cs是程序的文件，而Class1Test.cs是与之对应的单元测试文件。

DemoUser.vsmdi：测试管理文件。

Localtestrun.testrunconfig：本地测试运行设置文件。

如何管理设置文件呢？右键再选属性（Property）并不对。你得双击文件才能进入管理及设置界面。在设置界面中，你可以让单元测试产生“demouser.dll”的代码覆盖报告。

注意在单元测试中，VSTS自动为你生成了测试的骨架，但是你还是要自己做不少事情，最起码要把那些//TODO的事情给做了（如代码清单1-2所示）。在这个时候，单元测试还都是用的Assert. Inconclusive，表明这是一个未经验证的单元测试。

代码清单1-2

/// <summary>

///A test for User (string)

///</summary>

[TestMethod()]

public void ConstructorTest()

{

string userEmail = null; // TODO: Initialize to an appropriate

// value

User target = new User(userEmail);

// TODO: Implement code to verify target

Assert.Inconclusive("TODO: Implement code to verify target");

}

进行简单的修改后，我们得到了一个如代码清单1-3正式的单元测试：

代码清单1-3

[TestMethod()]

public void ConstructorTest()

{

string userEmail = "someone@somewhere.com";

User target = new User(userEmail);

Assert.IsTrue(target != null);

}

//我们还可以进一步测试E-mail是否的确是保存在User类型中。

解释单元测试的结构

从上面这个例子可以看到创建单元测试函数的主要步骤：

（1）设置数据（一个假想的正确的E-mail地址）；

（2）使用被测试类型的功能（用E-mail地址来创建一个User类的实体）；

（3）比较实际结果和预期的结果（Assert.IsTrue(target!= null);）。

现在可以运行单元测试了，同时可以看看代码覆盖报告“code coverage report”，代码百分之百地都被覆盖了。

当然这时候的代码还有很多情况没有处理：

处理空的字符串，长度为零的字符串，都是空格的串……

接下来用Copy/Paste又写了下面的三个测试，如代码清单1-4所示。

代码清单1-4

[TestMethod()]

[ExpectedException(typeof (ArgumentNullException))]

public void ConstructorTestNull()

{

User target = new User(null);

}

[TestMethod()]

[ExpectedException(typeof(ArgumentException))]

public void ConstructorTestEmpty()

{

User target = new User("");

}

[TestMethod()]

[ExpectedException(typeof(ArgumentNullException))]

public void ConstructorTestBlank()

{

User target = new User(" ");

}

如果不修改类库中的代码，单元测试会报告这三个新的测试都失败了。

再对代码做了相应的修改。结果出了这样的错误，见代码清单1-5：

代码清单1-5

Test method UserTest.UserTest.ConstructorTestBlank threw exception System.ArgumentException, but exception System. ArgumentNull- Exception was expected. Exception message: System.Argument- Exception: Value does not fall within the expected range.

原来是Copy/Paste用了原来的ArgumentNullExcep- tion，而不是ArgumentException。

如果有人加了下面的代码：

if (!m\_email.Contains("@"))

{

**throw new ArgumentException();**

}

这时，代码覆盖测试就会报告代码覆盖率是85%左右。那还得加上新的单元测试以保证所有的代码都得到了基本的测试。

二柱：现在我知道为什么有些软件写了好几年都没有发布了，敢情他们都忙着写单元测试了。

阿超：也许因为他们没有在一开始就写单元测试，所以后来有很多小强要处理。很多调查显示，在软件开发后期发现的Bug，修复起来要花更多的时间。

芸芸：这对我们设计人员有什么用呢？好像都是一些细节的东西。

阿超：在我们写规格说明书（specification）的时候，要越详细越好，最好你的各项要求都可以表达成单元测试的一个测试用例。

芸芸：如果不能表示为一个单元测试呢？

二柱：那就是你写得还不够细。

小飞：我大胆地说一句。如果是一个人写写程序玩玩，单元测试似乎不那么重要。

二柱：你可以大胆地对你的女朋友说：“我们只是玩一玩……”看看效果如何。

阿超：如果玩一玩，什么都不太重要。如果你写的模块会有不同的人，在不同的时间使用，那你最好把你这一“单元”要做的事，以及它不能做的事，用单元测试清晰地表达出来。

1.2 好的单元测试的标准

下面我们讲讲怎样才算一个好的单元测试。

单元测试应该准确、快速地保证程序基本模块的正确性。下面是验证单元测试好坏的一系列标准：

单元测试应该在最低的功能/参数上验证程序的正确性。

单元测试应该测试程序中最基本的单元——如在C++/C#/Java中的类，在此基础上，可以测试一些系统中最基本的功能点（这些功能点由几个基本类组成），从面向对象的设计原理出发，系统中最基本的功能点也应该由一个类及其方法来表现。单元测试要测试API中的每一个方法及每一个参数。

单元测试必须由最熟悉代码的人（程序的作者）来写。

代码的作者最了解代码的目的、特点和实现的局限性。所以，写单元测试没有比作者更适合的人选了。

问：如果我很忙，能不能让别人代劳做单元测试？

答：如果忙到连单元测试都没有时间做，那么你也没有时间写好这个功能。在一些极限编程的方法中，是可以考虑让别人来做单元测试的，但是，程序的作者还是要对单元测试负责。

最好是在设计的时候就写好单元测试，这样单元测试就能体现API的语义，如果没有单元测试，语义的准确性就不能得到保障，以后会产生歧义。

单元测试过后，机器状态保持不变。

这样就可以不断地运行单元测试，如果单元测试创建了临时的文件或目录，应该在Teardown阶段把这些临时的文件或目录删除。

如果单元测试在数据库中创建或修改了记录，那么也许要删除这些记录，或者每一个单元测试使用一个新的数据库，这样可以保证单元测试不受以前单元测试实例的干扰。

单元测试要快（一个测试运行时间是几秒钟，而不是几分钟）。

快，才能保证效率。因为一个软件中有几十个基本模块（类），每个模块又有几个方法，基本上我们要求一个类的测试要在几秒钟内完成。如果软件有相互独立的几个层次，那么在测试组中可以分类，如数据库层次、网络通信层次、客户逻辑层次和用户界面层次，可以分类运行测试，比如只修改了“用户界面”的代码，则只需运行“用户界面”的单元测试。

单元测试应该产生可重复、一致的结果。

如果单元测试的结果是错的，那一定是程序出了问题，而且这个错误一定是可以重复的。

问：如果用随机数以增加测试的真实性，好么？

答：一般情况下不好，如果某个随机数导致程序出错，但是下一次运行又不能重复这一错误，于事无补。要注意我们还是要用随机数等办法“增加测试的真实性”，但是不是在单元测试中。单元测试不能解决所有问题，所以也不必期望它会发现所有的缺陷。

独立性，单元测试的运行/通过/失败不依赖于别的测试，可以人为构造数据，以保持单元测试的独立性。

程序中的各个模块都是互相依赖的，否则它们就不会出现在一个程序中。一般情况下，单元测试中的模块可以直接引用其他的模块，并期待其他的模块能返回正确的结果。

如果其他的模块很不稳定，或者其他模块运行比较费时（如进行网络操作），而且对于本模块的正确性并不起关键的作用，这时可以人为地构造数据以保证这个单元测试的独立性。

单元测试应该覆盖所有代码路径，包括错误处理路径，为了保证单元测试的代码覆盖率，单元测试必须测试公开的和私有的函数/方法。

单元测试必须覆盖所测单元的所有代码路径。

问：啊！这样岂不是要写很多啰里啰唆的测试方法？

答：对，因为程序中很多缺陷都是从这些啰里啰唆的错误处理中产生的。如果你的模块中某个错误处理路径很难到达，那你也许要想想是否可以把这个错误处理拿掉。

大栓：这对于那些爱写复杂代码的人是一个很好的惩罚，不对，是一个很好的锻炼。

阿超：对，把单元测试的责任和代码作者绑定在一起后，代码作者就能更真切地体会到复杂代码的副作用，因为验证复杂代码的正确性要困难得多。要注意的一点是：100%的代码覆盖率并不等同于100%的正确性。在下面的情况下，100% 的覆盖率和100% 的正确性不是同一回事:

     a) 代码中并没有处理错误情况。 例如代码打开了文件，但是并没有处理一些异常情况，例如文件不存在，权限有问题，等等

     b) 代码中有效能问题，虽然代码执行了，并且也正确地返回了。但是代码执行得也许非常慢。

     c) 多线程环境中的同步问题, 这个问题和本地代码执行与否关系不大。

     d) 其它和外部条件相关的问题 (例如和设备相关，和网络相关的问题)

单元测试应该集成到自动测试的框架中。

另一个重要的措施是要把单元测试自动化，这样每个人都能很容易地运行它，并且可以使单元测试每天都运行。每个人都可以随时在自己的机器上运行。团队一般是在每日构建中运行单元测试的，这样每个单元测试的错误就能及时被发现并得到修改。

单元测试必须和产品代码一起保存和维护。

单元测试必须和代码一起进行版本维护。如果不是这样，过了一阵，代码和单元测试就会出现不一致，而且所有代码的作者要花时间来确认哪些是程序出现的错误，哪些是由于单元测试更新滞后造成的错误。这样就失去了单元测试的意义，同时又给大家增加了负担。如此折腾多次以后，大家就会觉得维护单元测试是一件很费时费力的事。

1.3 回归测试

在单元测试的基础上, 我们就能够建立关于这一模块的回归测试 (Regression Test).

Regress 的英语定义是： return to a worse or less developed state。是倒退、退化、退步的意思。

在软件项目中，如果一个模块或功能以前是正常工作的，但是在一个新的构建中出了问题，那这个模块就出现了一个“退步”（Regression），从正常工作的稳定状态退化到不正常工作的不稳定状态。

在一个模块的功能逐步完成的同时，与此功能有关的测试用例也同样在完善中。一旦有关的测试用例通过，我们就得到了此模块的功能基准 (Baseline) , 一个模块的所有单元测试就是这个模块最初的Baseline。

假如，在3.1.5版本，模块A的测试用例125是通过的，但是测试人员发现在新的版本3.1.6，这个测试用例却失败了，这就是一个“倒退”。在新版本上运行所有已通过的测试用例以验证有没有“退化”情况发生，这个过程就是一个“Regression Test”。如果这样的“倒退”是由于模块的功能发生了正常变化（由于设计变更的原因）引起的，那么测试用例的基准就要修改，以便和新的功能保持一致。

针对一个Bug Fix, 我们也要作Regression Test。

（1）验证新的代码的确把缺陷改正了。

（2）同时要验证新的代码没有把模块的现有功能破坏，没有Regression。

所以对于“回归测试”中的“回归”，我们可以理解为“回归到以前不正常的状态”。

回归测试最好要自动化，因为这样就可以对于每一个构建快速运行所有回归测试，以保证尽早发现问题。单元测试是回归测试的基础.

在专注于模块基本功能的单元测试之外, 还有功能测试 – 从用户的角度检查功能完成得怎么样。  在微软的实践中，在一个项目的最后稳定阶段，所有人都要参加全面的测试工作，把所有以前发现并修复的bug 找出来, 一个一个验证, 以保证所有已经修复过的Bug的确得到了修复，并且没有在最后一个版本中“复发”, 这是一个大规模的、全面的“回归测试”。