PROFESSIONAL TESTER

Essential for software testers

February 2012 | £4 / €5 | v2.0 | number 13

development testing

Including articles by:

Chris Adlard
Coverity

Les Hatton
Kingston University and
Oakwood Computing
Associates

**Harry M. Sneed and
Manfred Baumgartner**
ANECON

Boguslaw Czwartkowski
Parasoft

Geoff Quentin

The covers are off

by Edward Bishop

Coverage is not a number

PT editor **Edward Bishop** proposes a test design improvement method using Ranorex

The full test suite is run and statement coverage is measured. It's 90% and the exit criteria for the phase calls for only 80%. Great!

But 95% of the tests are covering exactly the same statements. 50% of the statements covered are executed by only a handful of very similar tests. Not great.

10% of statements are not executed by the test suite, yet static analysis indicates that there is no unreachable code. This is because the statements can be reached, but only if the last transaction on the currently-logged-in account happened at 0000 hours UTC. Also not great.

Simple and more complex coverage

Programmers who practice test-driven development correctly measure coverage of their code continually. At component level its meaning is clear, especially using one of the many powerful coverage measurement tools available. These integrate closely with development environments and show in visual reports how many times each line of code has been covered and, in some cases, which tests covered a particular line. This is all the information a developer needs to know whether (i) the unit tests are working as expected and (ii) the code needs to be refactored: if code is written, as it should be, only to pass unit tests the coverage should always be close to 100% and if it falls below that when all the unit tests needed are run something is wrong.

To help assure the effectiveness of testing at higher levels, more and different coverage information is needed. But after integration, when tests are run in a test rather than a development environment, getting it becomes much more complicated because simple measurements may or may not be meaningful and that is hard to establish. Most coverage measurement tools can highlight code that is "insufficiently covered" but what appears to be "sufficiently covered" may only have been covered repeatedly by too few tests. Dynamic analysers that generate diagrams visualizing control and data flow can also help to find rarely-occurring paths. But testers need a way to find out more: which statements or decisions are covered as the result of which test events and data. Having that information would create a lot of potential: not just to detect more defects with existing tests, but to improve the tests to provide more assurance and detect yet more defects if they exist.

Getting it requires a way to link each coverage event to the test that caused it. It has been suggested, including by Harry M. Sneed, a contributor to this issue, that this can be done using time. Here is a theoretical method:

1. The probes inserted when the code is instrumented are designed to record not only that they have been executed, but when, according to the system clock
2. Each test script, or the tool executing it, also records when it starts and terminates and, in data-driven testing, a means of identifying the data it uses (eg the number of a line read from a CSV file)
3. After execution, the two output files are correlated and analysed.

Doing this could reveal which tests are very similar and, more interestingly, which are very different, to others in terms of the code they cause to be executed. Introducing more variations of these tests, then repeating steps 2 and 3 to show that more code is executed by more different tests, would increase the defect-finding potential of the suite.

A coverage comparator tool

It may be possible to get the time-based method to work, but there are obvious difficulties including the familiar problem of time dependencies. There will be a discrepancy between the two recorded times and for many system architectures it will vary significantly within and between runs. Reliable correlation may be a significant challenge. It seems likely but not certain that technical solutions could be found.

A more direct method is easier: execute tests or groups of tests individually and discover the detailed coverage each achieves separately. That removes the need for the probes to do anything other than identify themselves which is how nearly all coverage measurement tools work. One of these could be used, or a new program written, to instrument the code: figure 1 shows how it can be done in and for VB.NET for simple line coverage. The number of probes needed could be reduced and other coverage types achieved by slightly more sophisticated parsing. The analysis program is nearly as easy to write: it reads all the coverage data generated by the test runs, searches for long blocks of text duplicated between them, then sorts the lines in each of them and compares the sorted lists. Figure 2 shows an example of some of the output that can be generated.

The hardest part is running the individual tests or groups of tests and organizing their output. Starting them manually would take a lot of effort, and between runs each coverage file created would have to be renamed or moved, then the information about the name or location of the files passed to the analysis program. It would be better to collect all the coverage information (the lines of text written by the probes) in one large file, but the blocks created by each run need to be separated. That could be achieved by a tiny program that appends the identity of the run to the coverage file, but that would have to be executed (and fed the run ID) between runs. Conducting any of these tasks manually would be onerous and prone to human error. It might be possible to automate them with a complex configuration of the test execution tool.

I have noted in previous articles (see the July 2010 and April 2011 issues of PT) that the functional test automation tool Ranorex provides exceptional flexibility that promotes close collaboration between development and testing. An example of this is the key to a simple way to build the coverage comparator tool this article describes. A test suite created in Ranorex is a standard .NET project saved as a .EXE file executable from the command line requiring only standard Windows runtime components and accepting arguments. One of them allows an individual test in the suite to be executed, thus:

```
project.exe /testcase|tc:<name
of test case>
```

So individual tests and the “run boundary marker” program can be run from a batch file. To define a group of tests as a single test for coverage measurement purposes they are run consecutively without running that program between them ■

```
Sub Main()
Dim CommandLineArgs As
System.Collections.ObjectModel.ReadOnlyCollection(Of String) =
My.Application.CommandLineArgs
Dim themodule, theinstrumentedmodule, thecoveragefile, thisline,
probeline As String, probenum As Integer
themodule = CommandLineArgs(0)
theinstrumentedmodule = "instumented-" & themodule
thecoveragefile = "coverage.txt"
If System.IO.File.Exists(theinstrumentedmodule) Then
My.Computer.FileSystem.DeleteFile(theinstrumentedmodule)
End If
FileOpen(1, themodule, 1)
FileOpen(2, theinstrumentedmodule, 8)
thisline = ""
While Not Left(LTrim(thisline), 3) = "Sub"
thisline = LineInput(1)
PrintLine(2, thisline)
End While
probenum = 0
PrintLine(2, "FileOpen(3, """ & thecoveragefile & """, 8)")
While Not EOF(1) And Not Left(LTrim(thisline), 7) = "End Sub"
thisline = LineInput(1)
If thisline <> "" Then
probenum = probenum + 1
probeline = "PrintLine(3, """ & themodule & "--PROBE--" & probenum
& """)"
PrintLine(2, probeline)
End If
PrintLine(2, thisline)
End While
While Not EOF(1)
thisline = LineInput(1)
PrintLine(2, thisline)
End While
FileClose()
```

Figure 1: Instrumentation program

```
Identical statement coverage (same path of control): Tests: 2, 2a,
2b, 2c, 2d, 2e, 3, 5a
Equivalent statement coverage (same statements executed in
different order): 2, 2f, 2g, 5, 5c
Unique statement coverage (cover statements no other test does): 3,
3c, 6b
```

Figure 2: Output of the analysis program

A free trial of Ranorex is available from http://ranorex.com