Wasting your time with Test Automation

30 December 2010

Software Testing is essentially an infinitely time consuming task being attempted in a finite time. The ‘test space’ is almost always vast and near infinite. Your time to test is usually counted in hours. There’s an obvious mismatch there. We’re testers, we are hired to to help marry the two. We need to find as many issues, and important issues, in that vast test-space in less than a few hours. Thats fine, thats testing, thats what testers work (live?) for.

Roll on Test Automation, our saviour, it can check vast areas of the test space rapidly and efficiently. It can use its ‘Data’ (http://en.wikipedia.org/wiki/Data_%28Star_Trek%29) like abilities to test the application tirelessly. No? Well Ok, It could ‘check’ the application tirelessly, for a set of expected results. This itself is potentially valuable, and could examine a range of combinations or test data that we could not be reach alone.

Why do many of the test automation efforts I’ve witnessed on customer sites, not deliver this? Many customers seem to put in the hard work, they’ve made a serious time and effort contribution to the problem. So why do they seem to be spending more and more time fixing the tests? slowly lowering their expected ‘pass rate’? or re-running the tests until they ‘give the right result’.

One major issue is reliability, the ‘tests’ are just not reliable enough, for what they were intended to do. The large array of checks should be slowly helping the testers more and more by helping them reach those hard to reach parts of the application. But the unreliable nature of the tests mean that each newly implemented check is actually just adding to the signals noise and the maintenance burden.

Lets look at an example, Say we have 300 ‘automated tests’.

Lets assume these tests are 95% accurate, and only give a false positive 5% of the time.
(This 5% could be down to a plethora of causes such as flakiness in the test-tool itself, problems with the wider system/support systems, network issues, out-of-date ‘expected results’ - or even poorly written test code.)

Also, lets say the tests are applied in the correct areas and would highlight 15 bugs in the system.
I’ll be even more generous and say that if the tests find a bug when testing a buggy area- then there definitely is a bug.
(no false negatives)

That means that the checks will correctly flag the 15 real bugs.
They will also false-flag (300-15) x 5% = 14 fake-bugs

When the checks finish: 14 of 29 or 48% of the results will be false positives or incorrect indications of a bug.

So in summary, even if the tests are 95% reliable (thats high from my experience) then approximately half of the testers work, reviewing the results, will potentially be a waste of time. Time that could be spent looking at the system under test is instead spent looking at flaky test results and bad test code. Those precious few hours of testing are misspent.

The solutions are not as simple as just ‘making the tests more reliable’. While having well written code, and good infrastructure can help immensely, the problems tend to be more fundamental. Some of the problems and solutions I’ve seen are:

1. The checks are asking binary questions [of complicated systems]. Try giving reports back instead, rather than hard pass/fail results. For example: is a HTTP 302 response a FAIL when you expected a HTTP 200? It might just be that the application has changed. A report covering the actual findings and all the other information that you get for free might be more useful. For example: How long did that response take? what was the size of that response? You could view those results directly or even analyse / graph them as you see fit - looking for patterns/issues. PASS/FAIL checks often seem to be ‘change detection’ systems rather than ‘bug detection’ systems.
2. Keep the checks simple, really simple. Its difficult to write the complicated code needed to handle the various inputs, outputs and state changes a real system undergoes. It’s the very reason we find work as testers, we are not immune to the problem of complex code.
3. Be aware that these are just ‘checks’ and all they can do is report. They can never find the ‘human stuff’. They can’t question or investigate the system. Leave time and resource for using your own testing skill to tackle the system. For example, trying to get your checks to do a visual ‘layout’ check for example can lead to time-consuming problems. See (4) You, yourself, could perform such a test in seconds, and probably provide better feedback.
4. Beware of using test automation with GUI’s. GUI’s are uniquely designed for human use, they:

   - Update in human-time frames, not at machine speeds - So you may ‘check’ at the wrong point in time.
   - Report information visually, and so use visual effects that can make test automation messier.
   - Are often out-of-sync with back-end server systems (Their code runs in a browser or in a separate thread etc)
   - Require the test tool to emulate user behaviour when ‘automating’ a check, programming the keyboard events, mouse clicks/events etc making the test-code more complicated. Look into accessing the System through other more programatic interfaces, for example use the XML, JSON or RMI API etc ‘behind’ the GUI if its available. You maybe

able to check much of the systems logic through this route. And if you can’t - You might have found an issue - i.e.: Lots of the application logic is in the GUI, when it might be better off on the server.

In summary, use the test-code for what its good at, and don’t be afraid to report information back to a human who can look for issues. The computer can do the heavy lifting and grunt work, and we can do the smart work of interpreting the feedback.