The arrogance of regression testing

24 December 2010 · bias, random, regression, testing

Lets assume we know that our software is not perfect. How can it be? Its complex, mortals created it and we don't have enough time to test every execution path & environment – so we could never be sure anyway. This is Ok - this is normal, testers deal with this situation every day.

This tends to be a typical scenario… Our team has been working on some new features. They're looking good, initial teething issues have been fixed and the new features are considered worthwhile enough and bug-sparse enough to be released into the wild.

This is where things can get a little awkward. The team member's opinions are often split across a wide spectrum. The relatively minor perceived impact of the work leads some to conclude that the work is ready for release as is.

Other team members, who are possibly twice shy from previous 'minor change' induced problems, argue for a comprehensive 'regression test' of the software. There is usually a range of views in between suggesting for example only 'regression testing' directly affected systems or those associated with the changes etc.

The oft-stated concern is: we may have broken something. Our new code might be good, the existing code might be even better, but what about emergent issues caused by the new 'system' we've created?

A common compromise proposed by teams and customers is generally translated as 'test that it [the core features etc] still works'. This may sound reasonable, intelligent and practical. But it just doesn't sit well with me. My unease stems from more than the assumption that testers can prove it works…again. The concept of regression testing seems arrogant and it even, worse it seems wasteful.

Arrogant? Regression testing assumes that we tested the application so well before that we found all the important areas of ambiguity and bugs. It's like saying that when Microsoft released a patch to Vista that they only needed to ensure it was up to the high standard of the original Vista release.

The root of the problem here is a bias. We start out with the perception 'Our system is great', and then lets check (sic) it still is. The Congruence bias is a powerful motivator to not upset the perceived status quo. We stop looking for problems that we don't think are caused by the new changes. Whereas we might investigate these 'issues' in other circumstances, we don't even think to look deeper unless they are practically labeled "BROKEN BY THE NEW CODE". How many issues are overlooked?

Wasteful? By misdirecting ourselves from the system as a whole – we are missing an opportunity to test again. A second chance to find those bugs we didn't find last time. We could be capitalizing on another chance to learn new and old areas of the application.

When I've been in this situation I've also fallen foul of the biases at play, I'm human. But as a tester, I've had to come up with a few strategies to help remedy the problem.

- Firstly, just be aware of the bias - don't let it lead you blindly.
- If you're lucky enough to have another tester in your team, split the problem with them. One of you can test the system as a whole, the other the 'affected areas'.
- Deliberate opposition. A technique I use when I've got a definite checklist of the affected areas. Deliberately pick things not on the list, or that are the direct opposite of what is on the list. E.g.: If the change affects Logging in as user X, What does logging out as user Y do? Or can you avoid being logged in all together?
- Randomness. Choose a path or some data at random. For good randomness www.random.org (http://www.random.org) is a useful source.

I find the above a useful means of breaking out of the tunnel vision of regression testing. They give you new paths to follow, that if explored can often yield new and old bugs.