Programmers & Testers, two roles divided by a common skill-set.

10 October 2016 · programmer, psychology

When we switch people from programming to testing and vice versa may reduce the quality of our software.

I'll get some quick objections out of the way:

- But, A person can be a great tester and programmer. - Yes I agree.
- But, Programmers do a lot of good testing. - Yes I agree.

None of the above are in conflict with my conjecture.

Programming or writing software automates things that would be expensive or hard to do otherwise. Our software might also be faster or less error prone at doing whatever it replaces. It may achieve or enable something that couldn't be done before the software was employed.

Writing software involves overcoming and working around constraints to achieve improvement. That is, looking for ways to bypass limitations imposed by external factors or limitations in the tools we use. For example, coping with a high latency internet connection, legacy code or poor quality inputs. A programmer might say they were taking advantage of the technologies' features to create a faster/more-stable system. A skilled and experienced programmer has learnt to deal with complexity and produce reliable and effective software. That's why we hire them.

A good example might be WhatsApp. Similar messaging systems existed before WhatsApp. But WhatsApp brought together the platform (mobile iOS & Android), cost (free at point of use), security (e2e encryption) and ease of use that people wanted. These features were tied together and the complexities and niggles were smoothed over. For example, skilled programmers automated address book integration and secure messaging instead of a user having to try and use multiple apps to achieve the same.

But the complexities or constraints are often leads to bugs. Leads that are easy to not fully appreciate. The builder's approach: It does that so I need to do this - can override the more investigative approach of - Puzzling over what does a systems apparent behaviour means about its underlying algorithm or supporting libraries? A good tester hypothesizes about what behaviours might be possible from the software. For example: Could we get the app to do an unwanted thing or the right thing at wrong time?

Alternatively a tester may observe the software in action, but bear in mind that certain symptoms may be caused by the constraints within the software or its construction. The psycholo-

gical ‘priming’ can make them more likely to spot such issues during the course of their examination of the app.

A common response at this point in the debate is, “The person writing the app/automated tests/etc would be able to read the code and see directly what the algorithm is!” But, that argument is flawed for 2 main reasons:

1. Many testers can read and write good code. This sort of investigation is and always has been an option - whether we are currently implementing the app or an ‘automated test’ or neither. The argument is often a straw man suggesting that all testers can not write (and therefore can’t read) code.
2. In a system of any reasonable complexity, there are situations where it’s easier to ascertain a system’s actual behaviour empirically. An attempt to judge its behaviour by purely examining the code is likely to miss a myriad of bugs caused by 3rd party libraries, environmental issues and a mish-mash of different programmers work.

For example…

Programmers:

Programmers can and do - test their own and colleagues code. They do so as programmers, focused on those very difficult constraints. One of the key constraints is time. Not just, can the code handle time zones etc. But, how long do I have to implement this? What can I achieve in that time? Can I refactor that dodgy library code? Or do I just ‘treat it as good’? Time and the other constraints guide the programmer down a different testing road. Their rationed time leads them to focus on what can be delivered. Their focus is on whether their code met the criteria in the ticket given the complexities that they had to overcome.

A classic symptom of this is the congruence bias, programmers implement code and tests that ascertain whether the system is functioning as they expect. That’s good. That can tell us the app can achieve what was intended. A good example might be random number generation. A team might be assigned to produce an API that provides a randomiser for other parts of the app. The team had been told the output needed to be securely random. That is very random.

The team, Knowing about such things use their operating system’s built in features to generate the numbers. (For example on Linux that might be /dev/random). Being belt and braces kind of people, they would implement some unit tests that would perform a statistical analysis of their function. This would likely pass with every build once they had fixed the usual minor bugs and all would be good.

Testers:

Luckily for the above team, they had a tester. That tester loved a challenge. Of course she checked the randomness of the system, and yes that looked OK. She also checked the code in conjunction with other systems, and again the system worked OK. The tester also checked if the code fast enough, and once again the system was fine. The tester then set up a test system with a high load. Boom. The log was full of timeout errors, performance was now atrocious and she knew she had struck gold. A little investigation would show that some operating system random number generators are ‘blocking’.

A blocking algorithm will cause subsequent requests to be queued (‘blocked’) until its predecessor has finished. Even if the algorithm is fast, there will be a tipping point when suddenly more requests are coming in than can be serviced. At that point the number of successful requests (per second, for example) will cease to keep up with demand. Typically we might expect a graph of the requests being effectively handled by our system to show a plateau at this point.

Our tester, had double checked the code could do the job. But also questioned the code in ways the team had not thought to look for. Given that there are tools and techniques to aid the measurement of randomness, this confirmatory step would likely be relatively short. A greater period of time would likely have been spent investigating [areas that at the time were] unknowns. Therefore, The question is less can the tester read the code or validated that it performs how we predicted. The question is more can they see what the software might have been? How might it fall short? What could or should we have built?

Our tester had a different mindset, she stepped beyond what was specified. We can all do this, but we get better the more we do it. We get better if we train at it. Being a good systems programmer, and training at that - comes at the cost of training in the tester mindset. Furthermore, the two mindsets are poles, each at opposite end of our cognitive abilities. Striving at one skillset might not help with the other. A tester that writes code has great advantages. They can and do create test tools - to tease out the actual behaviour of the system. These programming skills have a investigative focus. They may even have a exploratory or exploitative (think security testing) focus, but not a construction focus.

For those that are screaming BUT THEY COULD HAVE HAD A BDD SCENARIO FOR HIGH LOAD or similar, I’ll remind you of the Hindsight bias (https://en.wikipedia.org/wiki/Hindsight_bias) (tl;dr: “the inclination, after an event has occurred, to see the event as having been predictable”)

While the programmer and the tester often share the same headline skills, e.g. they can programin language X, understand and utilise patterns Y & Z appropriately. They apply these skills differently, to different ends.

The change from tester to programmer is more than a context switch. It’s a change in your whole approach. People can do this, but it has a cost. That cost might be paid in slower delivery, bugs missed, or features not implemented.