The gamification of Software Testing

20 January 2018 · critical thinking, exploratory testing, investigation, pre-scripted testing

A while back, I sat in on a planning meeting. Many planning meetings slide awkwardly into a sort of ad-hoc technical analysis discussion, and this was no exception. With a little prompting, the team started to draw up what they wanted to build on a whiteboard.

The picture spoke its thousand words, and I could feel that the team now understood what needed to be done. The right questions were being asked, and initial development guesstimates were approaching common sense levels.

The discussion came around to testing, skipping over how they might test the feature, the team focused immediately on how long testing would take.

When probed as to how the testing would be performed? How we might find out what the team did wrong? Confused faces stared back at me. During our ensuing chat, I realised that they had been using BDD scenarios [only] as a metric of what testing needs to be done and when they are ready to ship. (Now I knew why I was hired to help)

There is nothing wrong with checking that the critical behaviour works (Whatever that means for them) and doing that before continuing is generally a good idea. But, after talking through their recent successes and failures, it dawned on me. They are gaming, with the best of intentions, they want to get 100% green stars/ticks from their 5 latest Cucumber scenarios for level sprint 5.

Their game had a goal, a feedback/points scoring system and a simple set of rules. But most importantly, like all good games, it was voluntary.

The Goal? Get an all green set of ticks on their Continuous Integration display.

The Feedback? The list of Scenarios they had to ‘automate,’ as they change from red to green, provide a progress report to the player’s team.

Rules? The rules vary depending on the team, but I’ve known them to include:

- All ‘tests’ have to be written using formal sounding phrases. A sort of simplified legalese.
- A test isn’t valid if it can’t be ‘fully automated.’
- Coverage must adhere to the pyramid/triangle (name your polygon) school of test coverage distribution.
- All tests must use tool X.
- Each test must only test one ‘thing.’
- Each test must test as many ‘things’ as possible.
- Etc.

Philosopher Bernard Suits spoke of games when he said they were:

> the voluntary attempt to overcome unnecessary obstacles.

That describes much of the gaming approach to software testing. Eschewing the difficult task of testing for unknown issues in software, we seek a more relaxed game. We choose to do that instead and to replace the arduous task of discovering new truths about our software. We set our own more easily defined obstacles.

These obstacles are usually the ones we can defend most readily to those unfamiliar with software testing. They follow a simple process, whereby some simple criteria are translated into the rubric of Given, When & Then.

In another apt quote Bernard Suits states:

> To play a game is to attempt to achieve a specific state of affairs [prelusory goal], using only means permitted by rules [lusory means], where the rules prohibit use of more efficient in favour of less efficient means [constitutive rules], and where the rules are accepted just because they make possible such activity [lusory attitude].

The use of a more complex, less efficient way of working is essential to the game. We just wouldn't be playing correctly if we didn't follow the rules. A direct and efficient approach is avoided, the goal of finding out what the software does and whether that is good is displaced by a less suitable means to a different end. The new goal is to encode a simplification the system's behaviour while adhering to the rules.

The game provides cover, protection from the culture of blame common in software development. If I played by the rules, and have the evidence to show I played every card dealt - then I am not to blame. (right?)

The loser here is the quality of the application and important factors such as time to market and reputation. An unreliable product that is received poorly and criticised etc. Or an app that gets 'stuck' pre-release and slips deadlines.

Playing the game is a form of goal displacement. To remember what testing is and to do it well is difficult. To test a complicated application often takes code, tools, intelligence, experience and an inquisitive mind (to name but a few). But there isn't a simple game that can weave those skills together for into a simple script. Just as there isn't a simple script for writing the code in the first place, Fortunately, we exist, and it's our role to help our clients find the truth about what their products are actually doing.

I propose a new game. A game based on polite and friendly one-upmanship. Team members continually try to outdo one another. The game develops a virtuous circle of behaviour to everyone's benefit. Providing the team with challenging and exciting work, a quality product and a good reputation.

For example, let us say Alice is working as a lead developer of a new feature. She can win by avoiding the usual bugs, and also by pre-empting those the testers might find. Alice uses all the tools and skills at her disposal, in the limited time, she has to outfox the testers. The testers will not even be able to whinge (let’s be honest we do moan a bit) about testability. This is because Alice learned in a previous round about the testers ‘get out of jail’ card: poor testability. She has included a full array of test hooks, giving them no excuse!

Beverley, our tester has been busy preparing for Alice’s next iteration. She investigates and questions the requirements. She frequently consults with the Product Owner. Not afraid to think that she might not know it all, she looks at rival companies versions of the feature. From experience, she knows that there may be details of bugs that might also occur in Alice’s new code.

Knowing that Alice is pretty thorough, Beverley knows she can’t just run her test automation and walk away. Alice will have caught all the obvious issues with her comprehensive unit tests. As well as automating the relevant checks and updating her tools she spends time thinking laterally and consulting with Alice directly to see if any areas are already being missed. She spends as much time as she can investigating the new feature and the whole system - as soon as she can. She is consistently in search of an angle of attack to find a new layer of bugs. For example, She suspects Alice isn’t thinking about the performance implications of her new feature…

This ongoing Tester-Developer arms race leads to a skilled and well-tooled team, who progressively improve how and what they deliver. The team becomes both lean and flexible, without unnecessary or bloated processes. They have to focus on quality and how to adapt quickly to new and higher standards because that is how they win their game.