How Do You Measure Quality, Anyway?

By Matthew Heuser for Subject7

A few years ago I worked with a team implementing a continuous integration system. The system was pretty simple. It checked out the code and ran unit-tests, then waited an hour to run again. The manager counted not only the number of passing assertions per day, but also the growth rate.

Of course, some programmer changed the delay, so it was 50 minutes, then 45, then 40. The number of test runs per day went steadily up -- with zero impact on quality. Personally, I take a healthy skepticism when people count the number of "test cases executed." In some places it can be helpful, but I wouldn't call it a measure of quality.

Counting bugs that escape to production is a similar problem. How many people did those bugs impact, did it stop them from their work, was that work essential, did they abandon the platform over it, and how long did the bug "live on" in production -- all of these are questions that a simple count fails to answer.

According to H.L. Mencken, for any complex problem there is an answer that is simple, intuitive, and wrong. So how do you measure software quality, then?

I'm not completely sure, but I suspect it has more to do with the features customers care about running than with the new and shiny features they might use later.

Quality Through Inside-Out Testing

Picture a common piece of software you use every day. Perhaps it is a word processor, or a spreadsheet, a social media application like Facebook or an eCommerce site like Amazon. Facebook, for example, became immensely popular in 2008. If you look at the profile of people in their 30's and up, they joined the site around 2007-2010 and have been using it since. Those users represent a continuous advertising stream. All the company has to do is to get a few things right and the users will keep coming back for more -- that is the ***core*** functionality. For something even simpler, let's consider YouTube, a video-sharing and upload site. The vast majority of users come to YouTube to find and watch videos. The company offers reviews, comments, the ability to curate your own stream of favorites, the ability to upload your own videos. Still, I suspect that 99%, if not 99.9%, of the action on YouTube is discovering and watching videos.

If the users can't watch the videos, nothing else really matters, and in short order, those users will abandon the site due to poor quality. And if they can watch videos, very much else will likely matter, as YouTube is "good enough," enough of the time, for enough of the people. Phillip Crosby called quality "conformance to requirements;" Gerald Weinberg called it "value to some person." If videos work on YouTube, YouTube is fulfilling its brand promise. I can't tell you what quality is to the beholder, but if you can't find or watch videos today on YouTube, I expect you'd agree that YouTube has a quality problem.

Follow this logic through and you find an idea. That idea is: the focus of a testing program should be on making sure the core features continue to work, and not necessarily about the new and shiny features. Ken Schwaber, a co-creator of the Scrum Framework, calls this the "core systems problem" in his talk "Agile Quality: A Canary in the Coal Mine."

The Core Systems Problem

According to Schwaber, companies tend to develop an older, core system that would be expensive to rewrite, because it uses outmoded programming constructs that people don't want to work on anymore. This is the COBOL system at the bank, the C++ system that pulls data from CAD in manufacturing, the old claims processing system at the insurance company. Schwaber says he runs into this problem over and over again. Every project touches the core system, and needs people who can touch, modify, and understand it. The new programmers don't want to, so Schwaber has to build teams that include experts in the old system. Eventually, he runs out of experts, and the new teams are simply not able to make progress. Without this kind of knowledge and organization, the teams will have scattershot staffing and simply write a lot of bugs.

You might not literally have a core system, but the ideas that fall out might be valuable. For example, a bug that impacts a core workflow, that can impact 25% of the user base in a week and 95% within a month, is a much greater impact to quality than a bug in a new feature. As long as there is a chance that changes can have unintended consequences, it makes sense to test the core flows, test them well and test them often, because that is where the biggest perceived risk to value lies. Karen's Johnson's RCRCRC provides other places to look, such as places in *chronic failure*, *risky areas*, *configuration-sensitive* or *recently repaired*. Still, the core matters.

A test strategy that focuses on core functionality could be part of a prescription for quality. It certainly doesn't stand alone; you'll want to look at time to identify problems in production and time to resolve them.

The point today is to give you a different way to look at quality, and some implications on how to measure them. The next question is how to build a test discipline around this, which is where I'm going next.