Wish-List Ideas for Software Testing Research

By **Michael Stahl** - June 15, 2020
Share URL
Bookmark

Summary:
There are many established ideas for ways to test software, but the industry is changing every day, and there's plenty of room for growth of new ideas—or challenges to traditional ones. Here are three ideas for "wish-list" research to conduct in order to shake up some of the conventional notions you may have about software testing techniques.

Like many people, I keep a list of things I plan to do sometime, when I have the time. I have a list of books I plan to read, skills I plan to learn, DIY projects I plan to do, and classic movies I plan to watch. Among these lists I also keep one with ideas for research in software testing.

Since this list only grows longer and I've never gotten to execute even one of these ideas, I decided to publish some of them here, hoping someone will accept the challenge.

Bug clustering

Software testing theory says that bugs are social creatures that like to congregate. Therefore, when a few bugs are found in a specific feature or a module, one should assume there are more bugs in that code area, and it's worthwhile to invest more testing effort on this part of the product.

On the face of it, validating this principle is straightforward: Check the distribution of bugs in each of the product's modules and see if there are more bugs in certain modules. But does this truly prove anything?

If most of the bugs in a module were found during a single test cycle, after which this module was practically bug-free, then the bug-clustering claim is refuted, which means that just looking at the overall distribution is not enough. You need to look at the distribution over a number of test cycles and check if a module that had many bugs in an early cycle continues to show many bugs in consecutive test cycles.

But even this is not so simple. If we see some cycles that reveal many bugs in a module while other cycles don't, we need to make sure the test cycles are comparable. It is possible that before test cycle A there were a lot of changes and additions to a module, while before test cycle B there were only bug fixes. In this case, we expect that test cycle A will find more bugs than cycle B.

We also need to make sure that in all test cycles we executed more or less the same tests—that is, there are no test cycles where, for reasons of time pressure or resource scarcity, we only executed part of the test suite. It's also recommended to have the same testers run all test cycles, since some testers have a sixth sense for catching bugs.

The more you think about it, proving the principle is not so easy.

Another approach can be to analyze a large quantity of data over a long time, in order to identify repeating phenomena that may be explained by the bug-clustering principle. Some years ago, I worked on an embedded system where most of the code was in firmware. In this product line, across many years and many versions, the same three modules had the largest number of bug reports. Most bugs were found in the lab of course, but these modules also had a disproportionate level of customer-reported bugs.

Isn't this clear proof that bugs are clustering? Well, maybe not. The problematic modules were the installation program and two other modules related to users' interaction with the system. That means the high number of bugs could have been due to a different explanation than clustering. Interacting with an embedded system is not straightforward for users or testers, and it was easier to arrive at challenging corner cases in those few modules that provide easy interaction.

To run a controlled experiment to check the bug-clustering principle, we would need to create a rather abnormal situation: a project without pressures that will allow for planning all the test cycles to be very close to each other in terms of coverage, test time, and the involved testers. All the code would be written up front, and the only changes after the first release would be bug fixes, done by the same team of developers. In short: There is no such project.

Does anyone have a good way to prove the principle? Let me know in the comments!

Conventional test techniques

A few months ago, a new vulnerability was discovered in the sudo command in Linux that allows an unprivileged user to **run code with root-level permissions**. Apart from the perverse joy I get from a critical bug, I also enjoy how this bug proves the theory behind boundary-value testing. To exploit this vulnerability, one must run the privileged command with a user ID of -1 or 4294967295 (which is the highest value represented by a 32-bit variable). In short, the bug appears only in a boundary value case.

Software testing theory teaches us that we should run tests with input values at the boundary of input ranges, since there is a higher chance of finding bugs with these values. This is so because of the classic joke: The two main reasons for bugs in software are mistakes in requirements, complex design, and off-by-one errors.

But how correct is this boundary-value theory? Or, in more general terms, what percentage of found bugs are caught by the set of standard test techniques taught in software testing textbooks and courses?

In principle, this analysis should be simple. All you have to do is to extract all the bugs from the bug database, analyze each of them …

Wait, analyze each of them?! There are thousands of bugs in the database. This is going to take a very long time! If a bug were found by a written test script and the design of that script were done using one of the well-known test techniques, then it may be a positive example for the benefit of test techniques. I say "may be" since sometimes a test case reveals a bug it was never targeted to find.

For example, say a text field accepts a maximum 512 characters. Running the boundary-value test of entering 512 characters to the field resulted in a failure. However, a deeper analysis of the bug found that the failure was not due to the length of the string, but to the time it took the tester to key in this string, which triggered a timeout. Any test where the text entry took more than five seconds would fail.

As for bugs found in exploratory testing, or found while testing something else, we would need to do detailed analysis. We would need to imagine the design of a test case that would catch this bug. Only then we would be able to determine if we could classify this bug to one of the well-known test techniques.

So again, we would need to invest rather significant amounts of resources to answer this question—which is a pretty important one. All books and courses teach this theory. How close is it to reality? Which technique is the most effective? What technique is useful only for certain cases?

I know of one large-scale study done in this direction: James Whittaker's research summed up in his book How to Break Software. Most of the techniques taught in this book are less trivial than the standard, basic test techniques we all learned. Makes you wonder.

Bug profiles

Assume I can show that a certain development team has the following characteristic: The first time they hand over a piece of code to test (the first drop in a project), it works really well and hardly has any bugs. As the project continues, usually right after the alpha milestone, suddenly the number of bugs skyrockets and reaches a peak at beta time. After beta, the number of new bugs diminishes at a constant rate until the release date.

One can analyze why this happens and look for ways to improve the situation, but I am not talking about that right now. If the same characteristic appears one project after another, then, if we happen to get a new project from this team with a lot of bugs in the pre-alpha release but few after alpha, we would immediately be able to note that something changed. It may be a good change; it may be a bad one—but we would be able to note that a change has occurred and try to understand it before more surprises hit us.

Similarly, it may be possible to describe how projects of certain types behave in terms of number of bugs and their distribution over time and severity. We may find that web applications, in a rather consistent way, behave the same: Initially, most bugs are of medium severity and related to the user interface. Later in the project, we find performance bugs that are usually of high or critical severity. In contrast, the behavior of an embedded system may be different: First to be found are functionality bugs and later integration bugs. Given such profiles, if the bug profile looks significantly different from usual, we again would be able to get early warning that something is different in the project.

More than this, if such profiles become publicly known, companies would be able to benchmark their projects against this general data and learn from the differences.

What's on Your List?

These three ideas are only part of the list I have. I encourage you to think of more ideas and make them public, or work with these as a jumping-off point.

You could just test them and publish the results. Use them as engineering projects for interns or as a final project for engineering degree students. Or be like me and add them to the list of things you plan to do, when you finally find the time.