How Much of Debugging Software Is a Tester's Responsibility?

By **Michael Stahl** - June 4, 2018
https://www.stickyminds.com/article/how-much-debugging-software-tester-s-responsibility

Summary:
Everyone knows a tester's job is to help improve the quality of the software under test. But it gets a little murky when you try to define the boundary between testing and debugging. There's no clear delineation: Some testers would state how to reproduce the bug, write the report, and hand it off, while others learn the code, find the root cause, and even create builds to fix the bugs. How much is useful, and how much is too much?

Testing, a friend told me, is like being a grandparent. The grandchildren are so cute and sweet. It's fun to play with them, throw them in the air, and tickle them … until you smell something. You check to ensure you are not wrong, and when you discover you weren't, you hand the culprit to the parents to take care of business!

Remind you of something?

The question is, how much do you need to check and verify things before handing over the "grandchild" to the developers?

I have seen various approaches that cover a wide spectrum of attitudes. The first approach is to hand off the bug to the developers as soon as possible, clearly state how to reproduce the bug, write the report, and that's it—it's not your problem anymore.

A second approach claims that our added value as testers increases when we are able to help the developer find the root cause faster. Taking this to the extreme, I've also seen a third approach, when I worked with a test team that thought the ability to actually fix bugs was the apex of a tester's professionalism. The testers in that team were familiar with the code to the point that they could compile it and create builds. When they found a bug, they'd dive in, find the problematic line of code, and propose a fix as part of the bug report.

What approach is the right one?

Personally, I tend to advocate the second approach. I think the first approach is too us-versus-them, and it misses the opportunity to help the developers fix the bug in a short time. I also want to believe that such an approach is not a common one in test teams that work closely with development.

The second approach holds that the tester's role is not only to find bugs, but also to provide information that the developers need to effectively debug. Apart from reporting the bug symptoms, testers are expected to find the simplest way to reproduce the bug, as well as provide logs, screenshots, and any other data that is relevant. In some cases, I think we should go so far as attempting to reproduce the bug on a number of operating systems, browsers, and different hardware. Once this data is accumulated and provided to the developer, it's the developer's role to find and implement a fix.

Those who support the third approach justify it as a way of providing testers with a development path. It allows testers to work with the code, gain a deep understanding of the

architecture and design, and get more satisfaction from the job than they could from just finding bugs. Furthermore, the testers gain the developers' appreciation because they prove they can do the same work that the developers do. This also helps move away from the view that testing is just a destructive profession, as the testers have a hand in actually developing the product.

Weighty claims indeed, but I'm sticking to my position.

To start with, gaining deep knowledge of the code generates the phenomenon of tester's bias, which causes testers to refrain from designing certain tests because they know "the code does not work this way." I don't mean to say that testers should never see the code, but how much and when needs to be thought through.

For example, in his book ***The Craft of Software Testing***, Brian Marick suggests that for any code that is not a simple, small unit, testers should look at the code only after the test design is done using the external interface requirements.

But this is not the main reason for my position. What bothers me about the third approach are the following concerns.

First of all, the message of the third approach is that the "real thing" is development. Testers who are perceived as top professionals and role models in this approach are not those who have a knack for finding critical bugs, but those who can fix the code. This appreciation comes not only from the developers, but also from the test managers, who see this capability as a proof that "testing is just like development." They want testers to have something to contribute to the product, the hidden message being that finding bugs alone is not good enough.

My second concern is that the developers (who can sometimes be a bit prima donna-ish!) get used to having someone else solve their bugs. Once enough testers provide this service, the developers start thinking that this is actually the way things *should* be. They don't think they should have to fix bugs that are so simple "even a tester can fix them," and they start believing that a tester who does not fix bugs is not worth much. Developers that are exempted from fixing their bugs don't feel the pain these bugs cause; they don't experience the time loss associated, and they don't learn how to avoid such bugs in the future. We miss an important feedback loop that would improve the developers' skills.

Finally, I'm worried about opportunity cost. This is an economic term that, translated to our context, means while the tester is busy debugging the code, finding the problematic line and suggesting a fix, they could have run some other tests and found more bugs. What's the best use of the tester's time, fixing the bug or finding new ones? Even if the additional tests don't find any bugs, we gain higher confidence in the quality of the product—which is one of our goals.

Each of the above concerns justifies, in my opinion, avoiding debug work by the testers. What's also interesting is that my first and last concerns relate to other areas of test management.

The first concern is relevant to test automation development. There are many test teams that give automation development a higher status than testing. This does not happen on purpose but is a natural, evolutionary process: When a team decides to invest in test automation, it becomes the flagship project of the test managers. Usually, at the start of such a project, progress is fast and impressive, and those who are involved gain prestige and promotion. At a certain point, management realizes that they must allocate some testers solely to automation

work, and this work is a bit more interesting than running regression tests. Soon you have a situation where any tester who is looking for a promotion (and who isn't?) wants a way to transfer to the automation team. Because the skills needed for automation are geared toward development and not testing, you end up with a test team whose focus is on improving their development skills instead of improving their testing.

The third concern, opportunity cost, is relevant to much that we do. For example, I found a hard-to-reproduce bug. How much effort should I put into causing this bug to appear again? If this is a bug that I know will be given a low severity level and will probably not be fixed, it makes sense, if only due to the opportunity cost consideration, not to put any added effort into it—just write the bug report, make a note that it's irreproducible, and move on to other tests.

Big decisions, such as whether to buy or build an automation framework, should also give much weight to the opportunity cost consideration: Is it worth investing time and money in developing a framework that will do exactly what we need, or does it make more sense to buy something that will meet only 90 percent of our needs and instead use the resources to improve testing?

In truth, the opportunity cost question should be part of any priority call we make. Should I read email now, potentially learning some important information, or should I ignore my overflowing inbox and finish the test plan document? Should I invest time in figuring out why my setup hangs every now and then, or should I just resign myself to the need to reset it once every hour, but make progress on the test cycle?

Being constantly aware of tradeoffs you make can improve your personal efficiency, but to be honest, it can also drive you crazy. We don't have all the inputs, all the data, and all the probabilities to make an objective and totally correct decision ten times a day. If you keep thinking about it, you will always feel you are working on the wrong thing and wasting your time, which is very frustrating.

So, when should we think of opportunity cost? I suggest always being aware of it, but I would think it through thoroughly only when you are about to invest a significant bulk of time in something—for me, that's about half a day. Give a five-second thought about whether this is the right thing to do right now. This is probably similar to what you may have already heard in a time-management course.

You certainly should consider opportunity cost when you're about to undertake a large project or when defining processes for your team. It's important to dedicate a formal time in the discussion to understanding what we gain and what we potentially lose with each significant decision we make.

So … was it worth reading this, or could you have used the time better?