RAIL SPIKES

search

Testing is overrated

Posted by **Luke Francl** on Friday, July 11

Next week at RubyFringe, I'll be taking on one of the programming world's favorite topics: **testing.**

Hear me out. Like everyone who's had their bacon saved by a unit test, I think testing is great. In a dynamic language like Ruby, tests are especially important to give us the confidence our code works. And once written, unit tests provide a regression framework that helps catch future errors.

However, testing is over-emphasized. If our goal is high-quality software, developer testing is not enough.

This is important because of what Steve McConnell calls **The General Principle of Software Quality.** Most development time is spent debugging. "Therefore, the most obvious method of shortening a development schedule is to improve the quality of the product." (Code Complete 2, p. 474.)

Problems with developer testing

Developer testing has some limitations. Here are a few that I've noticed.

Testing is *hard*...and most developers aren't very good at it!

Programmers tend write "clean" tests that verify the code works, not "dirty" tests that test error conditions. Steve McConnell reports, "Immature testing organizations tend to have about five clean tests for every dirty test. Mature testing organizations tend to have five dirty tests for every clean test. This ratio is not reversed by reducing the clean tests; it's done by creating 25 times as many dirty tests." (Code Complete 2, p. 504)

You can't test code that isn't there

Robert L. Glass discusses this several times in his book Facts and Fallacies of Software Engineering. Missing requirements are the hardest errors to correct, because often times only the customer can detect them. Unit tests with total code coverage (and even code inspections) can easily fail to detect *missing* code. Therefore, these errors can slip into production (or your iteration release).

Tests alone won't solve this problem, but I have found that writing tests is often a good way to suss out missing requirements.

Tests are just as likely to contain bugs

Numerous studies have found that test cases are as likely to have errors as the code they're testing (see Code Complete 2, p. 522).

So who tests the tests? Only review of the tests can find deficiencies in the tests themselves.

Developer testing isn't very effective at finding defects

To cap it all off, developer testing isn't all that effective at finding defects.

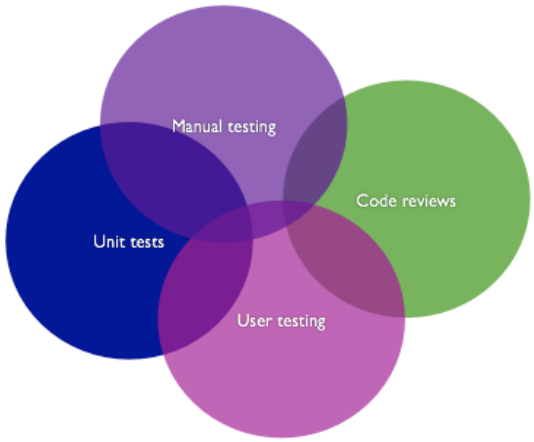
Defect-Detection Rates of Selected Techniques (Code Complete 2, p. 470)

Removal Step	Lowest Rate	Modal Rate	Highest Rate

Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Modeling or prototyping	35%	65%	80%
Formal code inspections	45%	60%	70%
Unit test	15%	30%	50%
System test	25%	40%	55%

Don't put all your eggs in one basket

The most interesting thing about these defect detection techniques is that they *tend to find different errors*. Unit testing finds certain errors; manual testing others; usability testing and code reviews still others.



Manual testing

As mentioned above, programmers tend to test the "clean" path through their code. A human tester can quickly make mincemeat of the developer's fairy world.

Good QA testers are worth their weight in gold. I once worked with a guy who was incredibly skilled at finding the most obscure bugs. He could describe *exactly* how to replicate the problem, and he would dig into the log files for a better error report, and to get an indication of the location of the defect.

Joel Spolsky wrote a great article on the Top Five (Wrong) Reasons You Don't Have Testers—and why you shouldn't put developers on this task. We're just not that good at it.

Code reviews

Code reviews and formal code inspections are incredibly effective at finding defects (studies show they are more effective at finding defects than developer testing, and cheaper too), and the peer pressure of knowing your code will be scrutinized helps ensure higher quality right off the bat.

I still remember my first code review. I was doing the ArsDigita Boot Camp which was a 2-week course on building web applications. At the end of the first week, we had to walk through our code in front of the group and face questions from the instructor. It was incredibly nerve-wracking! But I worked hard to make the code as good as I could.

This stresses the importance of what Robert L. Glass calls the "sociological aspects" of peer review. Reviewing code is a delicate activity. Remember to review the code…not the author.

Usability tests

Another huge problem with developer tests is that they won't tell you if your software *sucks*. You can have 1500% test coverage and no known defects and your software can still be an unusable mess.

Jeff Atwood calls this the ultimate unit test failure:

> I often get frustrated with the depth of our obsession over things like code coverage. Unit testing and code coverage are good things. But perfectly executed code coverage doesn't mean users will use your program. Or that it's even worth using in the first place. When users can't figure out how to use your app, when users pass over your app in favor of something easier or simpler to use, that's the ultimate unit test failure. That's the problem you should be trying to solve.

Fortunately, usability tests are easy and cheap to run. Don't Make Me Think is your Bible here (the chapters about usability testing are available online). For Tumblon, we've been conducting usability tests with screen recording software that costs $20. The problems we've found with usability tests have been amazing. It punctures your ego, while at the same time giving you the motivation to fix the problems.

Why testing works

Unit testing forces us to *think* about our code. Michael Feathers gets at this in his post The Flawed Theory Behind Unit Testing:

> One very common theory about unit testing is that quality comes from removing the errors that your tests catch. Superficially, this makes sense….It's a nice theory, but it's wrong….
>
> In the software industry, we've been chasing quality for years. The interesting thing is there are a number of things that work. Design by Contract works. Test Driven Development works. So do Clean Room, code inspections and the use of higher-level languages.
>
> All of these techniques have been shown to increase quality. And, if we look closely we can see why: all of them force us to reflect on our code.
>
> That's the magic, and it's why unit testing works also. When you write unit tests, TDD-style or after your development, you scrutinize, you think, and often you prevent problems without even encountering a test failure.

So: adapt practices that make you think about your code; and supplement them with other defect detection techniques.

Testing testing testing

Why do we developers read, hear, and write so much about (developer) testing?

I think it's because it's something that we can control. Most programmers can't hire a QA person or conduct even a $50 usability test. And perhaps most places don't have a culture of code reviews. But they can write tests. Unit tests! Specs! Mocks! Stubs! Integration tests! Fuzz tests!

But the truth is, **no single technique is effective at detecting all defects.** We need manual testing, peer reviews, usability testing *and* developer testing (and that's just the start) if we want to produce high-quality software.

Resources