Testing is overrated

Luke Francl <look@recursion.org>
RubyFringe, July 19, 2008

There's no question that programmatic testing is important, especially in a dynamic language like Ruby. I'll never forget the feeling I got the first time a unit test I wrote saved me from committing buggy code. And once written, tests provide a regression framework that helps catch future errors. In short, **testing is great.**

However, I believe we spend too much time talking about developer testing, and not enough about the other techniques necessary for making high quality software. We play to our strength – coding – and try to code our way out of buggy code. In its most extreme form, this leads to "mine's bigger than yours" code coverage contests and a belief that 100% test coverage will result in error-free software.

In fact, testing has a number of problems that preclude it from being the sole method of quality assurance.

> When I say "testing is overrated", I mean "testing" in the sense programmers do when they speak of "writing tests."

Testing is hard

...and most developers aren't very good at it. Most developers tend to write "clean" tests that verify the normal path of program execution, not "dirty" tests that verify error or boundary conditions. Immature testing organizations have 5 clean tests for every dirty test. Mature testing organizations have 5 dirty tests for every clean test. They accomplish this not by writing fewer clean tests, but by writing *25 times* more dirty tests! (McConnell 504)

Consider the following code, taken from Code Complete and translated into Ruby (McConnell 507-516). Note: this code may have bugs!

```
1.  total_withholdings = 0
2.
3.  employees.each do |employee|
4.
5.    if employee.government_retirement_withheld < MAX_GOVT_RETIREMENT
6.      government_retirement = compute_government_retirement(employee)
7.    end
8.
9.    company_retirement = 0
10.
11.   if employee.wants_retirement && eligible_for_retirement(employee)
12.     company_retirement = get_retirement(employee)
13.   end
14.
15.   gross_pay = compute_gross_pay(employee)
16.
17.   personal_retirement = 0
18.
19.   if eligible_for_personal_retirement(employee)
20.     personal_retirement =
21.       personal_retirement_contribution(employee, company_retirement, gross_pay)
22.   end
23.
24.   withholding = compute_withholding(employee)
25.   net_pay = gross_pay - withholding - company_retirement -
26.             government_retirement - personal_retirement
27.
28.   pay_employee(employee, net_pay)
29.
30.   total_withholdings = total_withholdings + withholding
31.   total_government_retirement = total_government_retirement + government_retirement
32.   total_retirement = total_retirement + company_retirement
33. end
34.
35. save_pay_records(total_withholdings, total_government_retirement, total_retirement)
```

> ***Structured Basis Testing***
> *To compute the minimum number of test cases:*
> 1. *count 1 for the method itself*
> 2. *add 1 for each* `if`*,* `while`*,* `and`*, and* `or`
> 3. *add 1 for each case in a* `case` *statement, plus 1 more if there is no default case*

Using structured basis testing, the minimum number of test cases to execute all the logic paths is 6. However, this is just the beginning, because it doesn't account for variations in the data. Steve McConnell eventually enumerates a full *17* test cases for just over 30 lines of code.

1. Nominal case: All boolean conditions `true`
2. The `employees` enumerable is empty
3. The first `if` is `false`
4. The second `if` is `false` because its first condition is `false`
5. The second `if` is `false` because its second condition is `false`
6. The third `if` is `false`
7. Define `company_retirement` on line 9, first use on line 21 (second `if` is `false`, third `if` is `true`)
8. Define `company_retirement` on line 9, first use on line 25 (second and third `if`s are `false`)
9. Define `company_retirement` on line 12, first use on line 25 (second `if` is `true`, third `if` is `false`)
10. Boundary condition test: `Employee#government_retirement_withheld == MAX_GOVT_RETIREMENT`
11. Compound boundary test: a large group of employees each with a large salary
12. Compound boundary test: a small group of employees, each with a salary of $0
13. Too much data: 100,000 employees ("too large" is system dependent)
14. Wrong kind of data: a negative salary
15. Wrong kind of data: `employees` is `nil`
16. Minimum normal configuration: 1 employee
17. Maximum normal configuration: a group of 500 employees (system dependent)

Structured basis tests (items 1–6)

That's a lot of tests! Note that creating just the first test case will score 100% code coverage with `rcov`, because every line in the program will be executed.

I think it makes sense to focus your testing effort on the parts of the program that are the most important, and most likely to contain errors. Errors tend to cluster, with about 80% of all errors found in just 20% of the code (Glass 136). It seems likely that complicated code is more likely to contain bugs. Either find a way to simplify the code, or make sure it's well covered.

You can't test code that isn't there

Missing requirements can cause cascading design failures, which could even impact the overall system architecture if the missing requirements are important enough. Even complete code coverage won't help you here. Worse, errors stemming from missing requirements are the most expensive type of error to correct if they slip into production. Fortunately, if caught in development, they are cheap to fix (McConnel 29-30, Glass 71-76).

The best ways to catch requirements errors are design reviews and prototyping/iterative delivery. I also find writing test cases useful for sussing out missing implications of a requirement, which can impact the requirement itself.

When doing iterative development, make sure your customer actually looks at the result. Otherwise it doesn t do you any good.

Tests are just as likely to contain bugs

All code has bugs. Tests are code. Therefore they contain bugs. Several studies have found that test cases are as likely to have errors as the code they're testing (McConnell 522). Who tests the tests? Only review of the tests can find deficiencies in the tests themselves.

Developer testing isn't very good at finding defects

An analysis of developer testing compared to other defect detection techniques shows that it is middling at best. Studies indicate unit tests remove between 15% and 50% of all known defects, compared to 20% to 70% for code reviews/inspections and 35% to 80% for prototyping (McConnel 470).

The most interesting thing about these defect detection techniques is that they *tend to find different errors* (McConnell 471). This shows two things. First, developer testing is not enough; and second, that additional types of defect detection methods are not a waste of time.

Therefore: **Don't put all your eggs in one basket.** Combine multiple defect-detection techniques to assure high quality software..

Defect-Detection Rates (McConnell 470)

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal Design Review	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	55%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low volume beta test (< 10 sites)	25%	35%	40%
High-volume beta test (> 1,000 sites)	60%	75%	85%

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.

Since all projects require some level of manual testing and developers are both bad at manual testing and resentful at being forced to do it, it makes sense to have a QA person – unless you want your customers do it for you. They may not be pleased with this arrangement. (Spolsky).

Code reviews

Code reviews and formal code inspections are incredibly effective at finding defects. Studies show they are more effective at finding defects than testing (developer or otherwise) and cheaper, too (McConnell 472, Glass 104-107). Plus, the peer pressure of knowing your code will be scrutinized helps ensure higher quality from the start.

Pair programming can provide a type of continuous peer review.

Code reviews can be very stressful. When done right, they require intense concentration by participants. Meanwhile, the reviewee's code (and ego) is on the line – and with more than one way to solve any typical problem, there's plenty of chances for conflict. Therefore, Robert L. Glass stresses the "sociological aspects" of peer review (113-115). Reviewing code is a delicate activity. Remember to review the code…not the author.

Can code reviews also help us grow our skill as programmers? The best way to learn programming is to read others' code (Glass 181-184), and constructive criticism during code reviews could help developers learn new skills. As a programmer who isn't young enough to know everything, I am hopeful.

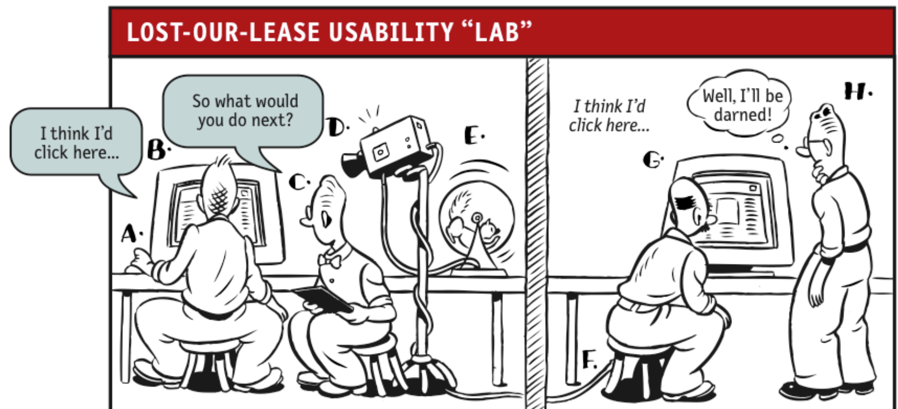
Usability testing

Another huge problem with programmatic testing is that they won't tell you if your software *sucks*. 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. (*emphasis in original)*

I have been amazed at the number and variety of errors ranging from simple to system-changing that we've found with usability testing. It is humbling to see your carefully crafted software proved so obtuse. But seeing these problems first-hand also gives me the necessary motivation to fix them.

Fortunately, usability testing is cheap and easy, and testing just a few people can find a majority of usability problems (Krug 142, Nielsen). Based on Steve Krug's "Lost our lease" usability testing guidelines, we perform usability tests with $20 screen recording software and a microphone. Participants are paid $50. Considering the cost-benefit ratio, there's no excuse not to do usability tests.



Test subject (A) sits in front of computer monitor (B), while facilitator (C) tells him what to do and asks questions. Camcorder (D) powered by squirrel (E) is pointed at the monitor to record what the subject sees.

Meanwhile, cable (F) carries signal from camcorder to TV (G) in a nearby room where interested team members (H) can observe.

From *Don t Make Me Think* by Steve Krug.

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! And of course, the positive benefits of testing get blown up when passed through our field's perpetual hype machine.

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. I hope this presentation will lead to more discussion of these other methods among developers.

Bibliography

Atwood, Jeff. "The Ultimate Unit Test Failure." *Coding Horror.* 14 Feb 2008. 12 Jul 2008. <http://www.codinghorror.com/blog/archives/001059.html>.

Glass, Robert L. *Facts and Fallacies of Software Engineering*. Boston, MA: Addison-Wesley, 2003.

Krug, Steve. *Don't Make Me Think: A Common-Sense Approach to Web Usability*. 2nd ed. Berkeley, CA: New Riders, 2005.

McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. 2nd ed. Redmond, WA: Microsoft Press, 2004.

Nielsen, Jakob. "Why You Only Need to Test with 5 Users." *useit.com.* 19 Mar 2000. 12 Jul 2008. <http://www.useit.com/alertbox/20000319.html>

Spolsky, Joel. "Top Five (Wrong) Reasons You Don't Have Testers." *Joel on Software.* 30 Apr 2000. 14 Jul 2008. <http://www.joelonsoftware.com/articles/fog0000000067.html>