INCOMPLETE TESTS ARE WORSE THAN NONE AT ALL!

SUMMARY: Testing of software, known to be the primary way organizations detect (and defeat) defects, is only effective if it is comprehensive. This article explains how testing works and why less-than-complete testing may be a "false friend" when trying to achieve real product quality. The moral - if you don't use test coverage you're testing in the dark.

BACKGROUND

The nature of software development has some unusual aspects, but, as in most human activities, most software people building something usually *try* do it right the first time.

But software development is complex, intricate, and relies on detailed knowledge of properties of abstract languages like C and C++; it takes a lot of time to get it right. Good programmers build code, test it at least minimally, and then integrate it into other software products.

And, inevitably, mistakes are made. Even the best programmers make errors. The empirical evidence, sparse though it may be, suggests that for every 1000 lines of code written, there are around 50 defects more or less, some more critical than others. This is an error rate of around 5%, about the same as in many text-entry activities.

As part of the usual edit-compile-test-debug-edit cycle, good software developers normally discover a high percentage of these defects. As many as 25%-50% of these latent defects are easy to discover and are revealed during development and are quickly removed. They are easy to discover because, unfixed, they prevent even basic functionality in the product. The developer is hard put to miss these!

The other 50%-75% or more are more subtle and most of the time they are never found by the developer or even the system integrator. They are usually found the "hard way": from field generated problem reports. Some, in fact, are never found at all! (A defect that never reveals itself in a complicated software product is a risk, but not a risk that can be addressed directly by testing.)

IS THIS REALLY SERIOUS?

If you look at the big software disasters of the last decade you find that most of the biggest failures are from subtle errors, ones never likely to have been discovered in routine testing. Think of the Intuit tax calculation problem (1994-95), the Denver Airport baggage handling issue (1994), the Intel Pentium chip programming mistake (1994), or the AT&T long distance phone network failure (1990). All of these problems arose from defects that were found the "hard way".

SO, WHY TEST AT ALL?

Clearly, you have to test. You have to test to (i) build confidence and (ii) learn what really IS right/wrong about your product. Sometimes

having confidence is more important than having the product "right" for at least you know what's up.

So long as you are going to test, you really ought to do as good a job of it as you can. This means that the main goal in testing is not only to make sure you test efficiently and realistically, but to make sure you test as completely as you can.

Remember, a defect detected and removed now saves you between 10 and 1000 times the cost of repairing it after your product is in the field (depending on its level of criticality).

TEST AUTOMATION AS THE HERO

So obviously you have to test better ... more realistically ... more efficiently ... more thoroughly.

To make testing more realistic and more efficient there is a lot of technology to help you: GUI-based testing, test generators, regression testing controllers. These technologies are well developed and quite powerful.

But to know if you're really done the job well, you really need to have a "coverage analyzer," a simple tool that tells you whether you have really done the job fully or not.

And, don't even THINK about doing this task without test automation to help you.

LEVELS OF TESTING COMPLETENESS

The only way you can tell if you have tested enough is to measure the tests you do have to make SURE they are complete. This is called "test coverage" and any software test process that doesn't include some kind of software test coverage is not really going to provide any protection against defects.

You CAN measure test coverage by checking each test against the code and using that information to assess how well you have actually tested. There are several test coverage measures for this:

- C0, statement coverage, tells a developer if all the lines have been tried.
- C1, branch coverage, tells if all of the logic has been tried.
- S0, module coverage, tells whether all the functions have been tried.
- S1, call-pair coverage, tells whether all the connections between calling function and called function have been tried.

There are many tools that measure these, more for C0 and S0 than for C1 and S1. ANY kind of test coverage is better than NO test coverage, but there are limitations.

REALISTIC ASSESSMENTS

But C0 is a false friend, sorry to say: C0 is a real liar. You can have 100% statement coverage but have exercised only 30%-50% of the logic. You think you're done, but key parts of the programs have never been tried. You get a very false sense of security!

SO is not quite as bad, but it is a bit of a liar too. Just having tried a function, from any one of potentially hundreds of call sites, is good news because you know it doesn't fail immediately. But MOST of the errors in calling functions lie in their parameter chains, so you really have to exercise them well.

Only C1 and S1 even begin to tell the truth, and they have their problems, too. For one, even 100% C1 is no guarantee of defect-free code, even though studies suggest that 100% C1 is about as useful as 80% of a formal proof of correctness.

And, 100% S1 says only that you have no explicit caller-callee problems, but it says nothing about errors that occur when three or more functions cooperate.

THE MORAL OF THE STORY

The moral of this story is, if you don't use test coverage you're testing totally in the dark. And, if you do use test coverage you need at *LEAST* C1 and S1 (not C0 or S0) to shed really serious light on product defects.

-Edward Miller