The Business of Software Testing: Failing to Succeed

Optimizing what we learn from testing.

There are two situations in software testing that scare testers: when they see “too many” defects and when they do not see “enough.” Too many defects may indicate the system contains a lot of problems even though it has moved into the testing phase. Too few discovered defects might imply the product is of high quality but it usually means the testing process itself is not working.

This is symptomatic of the paradox of testing: We want to find defects in the system under test, but we do not really want to find them. While the business of software has evolved from actually punishing people for finding defects, we can still be pretty ambivalent about them. Experienced testers have a good feel for the balance between finding too many defects and not finding enough, but most organizations and most customers would still prefer to hear “we haven’t found many defects” than “we’ve found huge numbers of defects.”

Testing as Knowledge Acquisition

If we view testing as a knowledge acquisition activity rather than simply a post hoc quality assurance process we get a different view. We also get a different view when we line testing up against the Five Orders of Ignorance. Zero Order Ignorance (0OI) is lack of ignorance; it is proven knowledge, knowledge that we have acquired and against which some trial has made that certifies it as “correct.” Second Order Ignorance (2OI) is lack of awareness of ignorance; it occurs when we don’t know something and we are unaware that we don’t know it. That is, we have ignorance about our lack of knowledge. We test systems primarily for these two of the Five Orders of Ignorance and their focus is quite different.

Testing for 0OI

We test to ensure the system performs as it was specified and as it was designed to perform. This is known as clean testing and it is ensuring the “proven” part of the definition of 0OI—we are showing that what we think we know is, in fact, correct. This kind of testing is relatively straightforward to set up, run, and prove. While the 0OI test set might be large, it is bounded—there are a finite number of things we want any system to do.

Testing for 2OI

The other kind of testing we do is to try and flush out any 2OI—we run dynamic tests on a system to see if there is anything we don’t know we don’t know about how the system actually runs. We can’t be specific about the problems we are trying to detect (in advance of actually running the test) or presumably we would have already fixed them. However, it is true that designing a good test may structure our thinking so we see the problem before we actually run the test. In this case the test we do run will be focused on 0OI, proving that our preemptive knowledge insertion really works. Testing for 2OI is an unbounded test set—the number of things a system might do but should not is essentially infinite. This is also a difficult test set to create since we have to devise tests for something we are not specifically looking for. The best we can do is to apply testing heuristics: test to the boundaries of ranges, test all classes on inputs and outputs, test all combinations of multiple logic conditions, and so forth. To expose our ignorance about system usability, we might put the system in front of a naïve user and see what hap-

Figure 1. Information content equation.

\[ \text{Information Content of a Test} = \text{Pf} \times \log \left( \frac{1}{\text{Pf}} \right) + \text{Ps} \times \log \left( \frac{1}{\text{Ps}} \right) \]

Figure 2. Binary view of information content of a test.
pens. We might simply “monkey test” by firing all manner of random data at the system. In all these cases we do not know a priori what will happen. We are looking for something, but we do not quite know what it is.

**Not Testing for 1OI**  
First Order Ignorance (1OI) occurs when we do not know something but we are fully aware of what we do not know. We should never test for 1OI. If we truly knew in advance what we did not know and where the limitations of our system’s knowledge lie, we would resolve our ignorance first, incorporate what we have learned into the system, and then conduct a 0OI clean test to prove it.

**A Successful Test**  
Here we see the dichotomy of testing: for 0OI a “successful test” does not expose any new knowledge, it simply proves our existing knowledge is correct. On the other hand, a 2OI test does expose new knowledge, but usually only if it “fails.” The yardsticks for success in testing: passing and failing tests are focused on these two different targets. This is partly where the tension between exposing and not exposing defects comes from. While having defects in our system is clearly a bad thing, finding them in testing (versus not finding them) is equally clearly a good thing. As long as there aren’t too many.

**How Much of a Good Thing?**  
For our 0OI testing 100% passing is the goal. Any test that “fails” indicates the presence of 2OI in the system (or the test setup or possibly sloppiness in testing, which is a different, Third Order Ignorance, process kind of failure). For 0OI testing, the ideal situation is that every bit of knowledge we baked into our system is correct and the successful tests simply prove it.

But what about the 2OI tests? Logic would suggest that a set of 2OI tests that exposed no defects at all would not be a good test run since nothing new would be learned. It is possible that a test run that exposed no defects at all shows the system is very, very good and there are no lapses in the knowledge that we built into the system. But this is unusual and most testers would be very suspicious if they saw no defects at all, especially early in a testing cycle. Logic aside, emotion would suggest that a set of 2OI tests that exposed 100% errors would also not be a “good” test run. While finding such a huge number of errors might be better than not finding them, it indicates either a criminally poor system or a criminally poor test setup. In the second case the knowledge we acquire by testing relates to how we conduct tests that might be easily learned and fixed. In the poor system case our ignorance is in the system being tested and it may indicate an awful lot of rework in the requirements, design, and coding of the system. This is not an effective use of testing. Indeed, it might be that we are actually using testing as a (very inefficient) way to gather requirements, design, and code the system since the original processes clearly did not work.

So if 0% defect detection is too low, and 100% defect detection is too high, what is the right number? Well, it would be somewhere between 0% and 100%, right? To find where this sweet spot of defect detection is, we need to look back to 1928.

**Transmission of Information**  
In 1928, Ralph Hartley, a telephone engineer at Bell Labs, identified a mechanism to calculate the theoretical information content of a signal. If we think of the results of a test run as signals containing information about the system that are transmitted from the system under test to the tester, at what point is this information maximized? Hartley showed the information content of a signal is proportional to the logarithm of the probability that the event occurs. Viewing a test result as a simple binary—a test throws a defect (“failure”) or a test does not throw a defect (“success”)—the information content returned is given by the equation shown in Figure 1, where $P_f$ = probability of failure (error is detected); $P_s$ = probability of success (error is not detected). The graph of this function is shown in Figure 2. In the simple binary view the maximum amount of information is returned when tests have a 50% probability of success (no error thrown). At that point, of course, they also have a 50% probability of failure.

**Managing Test Complexity**  
This gives testers a metric by which to design test suites. If a set of tests does not return enough defects we should increase the test complexity until we see about half the tests throw errors. We would generally do this by increasing the number of variables tested between tests and across the test set. Contrariwise, if we see that more than 50% of the test cases expose defects, we should back off the complexity until the failure rate drops to the optimal level.

This optimization is ideal for knowledge-acquiring (2OI) tests. For knowledge-proving (0OI) tests, the ideal is 100% pass rate. The problem is, we do not know in advance that (what we think is) a 0OI test won’t expose something we were not expecting. And sometimes what is exposed in a 0OI test is really important, especially since we weren’t expecting it. Still, as we migrate testing from discovery to proof we should expect that the failure rate will switch from 50% to 100%. How this should happen is a story for another day.

**I Knew That**  
I showed this concept to a tester friend of mine who has spent decades testing systems. His response: “I knew that.” He said. “I mean, if no errors is bad and all errors is bad, of course a good answer is some errors in the middle and 50% is in the middle, right? I don’t need an 80-year-old logarithmic formula derived from telegraphy information theory to tell me that.”

Hm, it seems the unconscious art of software testing is alive and well.

---

**References**


Philip G. Armour (armour@corvusintl.com) is a senior consultant at Corvus International Inc., Deer Park, IL, and a consultant at QSM Inc., McLean, VA.

Copyright held by author.

---

a Information theory does assert that the knowledge content of a system is increased by a 2OI test that “passes”—specifically it assures that the system will not throw an error under the specific conditions the test employed and provides some assurance for certain related tests. However, since the possible 2OI test set is functionally infinite, this assurance is not strong.