



"turning data into dollars"

Tom's Ten Data Tips – September 2009

Software Testing

Testing is an information gathering activity, aimed to support decision making. Since testing is never "free", the information gathered from it must have sufficient value for decision making to offset the costs. Costs can be investments in testing, as well as economic costs associated with a delay in release dates. Unless the outcome of testing actually influences decision making, you might as well forego it. Testing has value insofar it contributes to better (management) decisions about quality improvement, or going ahead to release a product after sufficient data points have been gathered to mitigate risks.

In many organizations testers are looked down upon, earn less than "real" software developers, and the job rarely provides a fast track to career success. Yet the information from testing is crucial for quality management. If testing is done poorly, it results in a decision process based on invalid information. And the consequences there can range from dear to fatal – imagine the casualties that have resulted from faulty programming of medical equipment! If testing is done too late, problem reports will be deferred because it is too late and risky to (safely) make any changes.

1. Testing Is Error Finding

One of the most dangerous misconceptions about testing is that it should prove the quality of software. This provides completely the wrong frame of mind. Instead, the objective of testing is to find as many errors as possible, in the least possible time. The sooner you can 'break' the software the better! Testing is about gathering information on which decisions will be based. If information resulting from a test isn't being used, you can argue there was never any test, either! The purpose of finding errors, however, is improvement of product quality.

When people see 'successful' testing as not having found any errors, they are actually at great risk of not finding (enough) errors. Instead, the more errors are found (before the product launch!), the better the test has been. Unfortunately, experience shows that when many errors

were found in a (section of a) program, there are likely many more still present.

2. "Human" Testing Is Very Effective

So called "human" testing methods like program inspections and walkthroughs (desk checking to a lesser extent) often appear "soft" and subjective, but in practice have definitely proven their worth.

The sheer fact that someone 'external' (*not* the developer) is reading a program does interesting things. It requires some degree of discipline to make a program (at least minimally) readable and explainable. Sometimes the "simple" act of explaining out loud what a program aims to "accomplish" can bring a lot of insight for the developer. This undoubtedly is *one of* the reasons why some agile "schools" advocate pair programming.

Given that about 2/3 of program cost over its lifetime goes to maintenance (see e.g. Martin & McClure, 1982), and is often not performed by the initial developer, a premium should go to making code readable and explainable anyway.

3. Testing Requires Sampling

Because exhaustive testing is physically impossible, any real life test involves taking a "sample." The fact that exhaustive testing is *never* possible unfortunately isn't quite obvious to everyone. If it isn't, point this out as soon as possible in your organization. Even the simplest of programs has an (almost) infinite number of tests that could be run.

Given the cost (or risk) versus information trade off in *any* test, a "reasonable" sample is required. It should be large enough to provide the information that is sought with sufficient reliability. At the same time it needs to be small and/or short enough so as to minimize costs and/or delays. Testers absolutely need to be aware of the (business) context so that they can add value in discussing these trade-offs.

4. The Test System Needs Testing, Too!

Although this may sound obvious as a ham sandwich, managing the testing project is often done under considerable time pressure. As you lay out the steps that need to be accomplished to fulfill testing all the high-risk components, your test system can become a bottleneck. For

complex systems or automated testing you will need to rely on a (automated) test system.

The test system itself requires documentation in order to make transparent what it does. Does it function according to specifications? Is it capturing all retrieved bugs for future investigation? One of the biggest problems is false positives during late integration testing. The answers to these questions clearly lie on the critical path of test plan development.

5. Fixing Errors Introduces *More Errors*

In consecutive development cycles where errors are (presumably!) fixed, new ones are invariably introduced. This can occur in one of three ways. The fix itself contains possible errors, the changed code will cause the rest of the program to 'behave' differently (with undesired "side-effects"), and, the fix may surface previously existing errors that were not yet apparent because of the (now resolved) error. This dynamic is one of the (very good) reasons why developers are embracing test driven development (TDD), where writing (automated) tests is an integrated part of development.

Some (ignorant) project managers would like to schedule *two* testing cycles: one to find the bugs, and the second to verify the fixes. Other than arriving at "code complete" as early as possible (what good is *that* if it's full of bugs??), we can think of no other reason to follow this "big bang" route. It most likely *extends* the development cycle and leads to inferior products.

6. "Bugs" Were Once Actually *Bugs*

The genealogy of the term "bug" stems from insect species. History has it that on 9 September 1947, when Grace Hopper was working on the Mark II (an early, and rather primitive relay computer) the machine was experiencing problems. There turned out to be a moth trapped in a relay, causing it to short-circuit.

Their repair log noted they had found a bug, and the engineers actually taped the moth to their rapport. In earlier years the term had been used in several (industrial and engineering) contexts, but since 1947 the term "debugging a computer program" has become a universal expression.

7. Don't "Abuse" Your Bug Tracking System

Bug tracking databases (testing systems) should be used for *just* that. As soon as you start to use (abuse) your testing system, for instance, to monitor individual tester performance, it will lose credibility and effectiveness. That includes progress-, or management statistics, etc. Testers and others involved learn this very quickly and in the end you get "what you pay for." Testers will cut their time short to properly investigate and describe problems. They will avoid difficult and time consuming issues. Instead they'll start producing large numbers of small yet unimportant bugs, etc.

Refrain from using your system in this way at all costs, not even under pressure (or bribe) from senior management. It's the beginning of the end. By the time the testing system has lost its effectiveness, there is no way back. And probably a way *out* for those responsible, out the back door...

8. Configuration Testing Requires Extensive Hardware Knowledge

It's been said before: exhaustive testing is impossible. The reasons for this become even clearer in configuration testing. When you're testing a program in conjunction with every possible modem, printer (&drivers!), disk, controller, mouse, card, monitor, etc., the combinatorial explosion quickly leads to prohibitive numbers. To narrow these down "intelligently" you divide items into groups. So EGA, VGA, SVGA, XGA monitors, etc. Testing is organized by cascading through devices and (sub)groups. You might cascade through device-independent errors, printer class-specific errors, driver-specific errors, and then printer-specific errors. Test "a" printer, then one printer from each group, end then possibly multiple within a group, etc.

Organizing the product space requires marketing knowledge to arrive at "guesstimates" of what configurations are most likely to be used by consumers. When you're developing a test plan you need extensive hardware knowledge to group products and drivers sensibly, for instance. And without this grouping there is simply no way to test efficiently! Since you cannot test every combination, you want to be reasonably assured that you're picking up "high probability" issues which impact the end-user experience the most.

9. Testers Do *Not* Determine “Sufficient” Quality

Testers’ job is to find as many important bugs (or design issues) as quick and economic as possible. However, they never ever get to decide when a product is “good enough” or ready to ship. That is, and forever must remain the responsibility of the project (or maybe product) manager.

When testers (unduly) feel the need to influence these decisions, relations with team members are bound to go awry. Testers must stick with their roles of *informing* these decisions. The responsibility to release the product (and live with the consequences) resides with the “owner” of the product.

10. Gather Test Information That You Use (Duh?)

Towards the end of the product development cycle there is often some kind of “rush”: things always seem to take longer than anticipated (isn’t that a surprise?? See also tip# 5), and all too often the testing phase gets squeezed like a harmonica. You can always continue testing, but may not be able to use that information to improve the (current) release. Maybe this data is used for a next release, or maybe you can provide it to customer service so they can use it. But why waste time and money on testing if you can’t realistically use the results?

Bear in mind that the fixes themselves also need to be tested! Expect in the order of magnitude of 25-50% new bugs for all fixes implemented. That time needs to be factored in. Due to these dynamics, a slight increase in new bugs (which empirically is more prevalent while under time pressure...) will cause a significant increase in time needed to fix. If that time is no longer available, then release as is, and don’t bother testing.