Inefficiency and Ineffectiveness of Software Testing: A Key Problem in Software Engineering

Cem Kaner, Ph.D., J.D.[1]

Most testing techniques in current use were developed before 1980. Back then, significant programs were less than 10,000 statements. In the commercial environments that much testing theory evolved, the programs were written in COBOL, a language designed to be understandable by nonprogrammers. Under these circumstances, a subject matter expert could serve as a tester, read the program, identify all the variables and their most interesting interactions, and trace the key paths through the program. Thoughtfully handcrafted tests, painstakingly documented, were the best practice of the day.

Today, it is common to find consumer products with a few million lines of code. Reuse of components (that may have never been tested with the new use in mind) makes it possible to assemble products orders of magnitude larger, in the same time. Even if the source code for the components were available (programmers often work with libraries that make visible only the application programming interface, not the underlying source), no individual could master the internal details of an application of this complexity.

Testing efficiency has improved somewhat in the past thirty years, but not nearly at the rate of programming efficiency. The most common automation technique, regression test automation, automates only test execution and simple result comparison—design, documentation and maintenance of these tests are time-costly human tasks. Test case management systems still call for step-by-step test documentation—some testers of life-critical applications report that they spend as much as 90% of their testing time on documentation-related tasks, only 10% on test execution and result analysis.

Simplistic coverage metrics are still widespread. They may be easy to compute, but they are poor indicators. Consider the problem of securing our borders—preventing the importation of bombs, dangerous chemical or biological agents, or narcotics. Even if we could secure the border such that every input (ships, autos, airplanes, pedestrians) came to a staffed checkpoint, how many could we search? Achieving complete statement coverage is like asking each entering vehicle or person for identification. You'll catch some problems that way, but it's not the same as searching the ship. We cannot search everything thoroughly, we cannot test everything thoroughly, but we must test some things thoroughly. Selecting the right ones requires human judgment and risk-assessment technology.

Recommendations

- Rely more heavily on high volume automation techniques.
- Retrain software testers to help them focus more effectively on risk.
- Rework test documentation practices to provide only those details needed to reasonably support auditing and test maintenance.

---

[1] Cem Kaner is Professor of Software Engineering at the Florida Institute of Technology. He is also the lead author of *Testing Computer Software* (with Hung Quoc Nguyen & Jack Falk), *Lessons Learned in Software Testing* (with James Bach & Bret Pettichord), and *Bad Software* (with David Pels).

Inefficiency and Ineffectiveness of Software Testing: A Key Problem in Software Engineering

Adopting high volume automation techniques

High volume techniques execute thousands (or billions) of tests without human intervention. Here are four examples of methods being used industrially:

- *In function equivalence testing,* we generate random input data (or sample the input space thoroughly and systematically) and use it compare the behavior of the function under test with a reference program. *This was the testing that first exposed the Pentium bug.*
- *In Random Regression testing,* we reuse tests that the program has already passed in this build. Run the tests in random order, checking whether the program continues to pass them over time. This style of testing highlights problems like gradual memory corruption and timing oddities that don't show up with an individual test that starts with the computer in a clean state and a test process that cleans up after each test. Traditional testing is like checking your brakes when you first start your car. RR is like testing them after you've been driving all day. *A leading maker of office automation products relies heavily on this in firmware testing. They regularly find significant problems that are hard to catch with traditional methods but that would cause serious intermittent failures in the field.*
- *State model based testing* can be adapted similarly to random regression. In this style of testing, you start from a known state, with a model that specifies what states you can reach, how you can tell which one you got to, and what states you can get to from each of those. Coverage-oriented state model testing ensures that every transition pair is covered at least once. Markovian testing relies on the key theorem for Markov chains, that if you transition randomly enough times, you'll reach every possible transition pair. Given that, the high volume version stops on failure or after a set (long) time rather than after a coverage criterion is reached. The super-long sequences can expose the same types of stack corruption, memory leaks, memory corruption, race conditions (etc.) that random regression exposes.
- *Hybrid performance and functional testing* runs the system under load and monitors system responsiveness (timing) as well as behavioral correctness. A common finding among performance testers is that functional failures rise dramatically at levels much lower than system saturation.

High volume automation sometimes simply covers a much higher proportion of tests in a well understood set (such as function equivalence testing, where you might cover N% of the possible input combinations). Other times, it reaches risks (memory corruption, timing errors, improperly modeled hardware state transitions) that don't fit within any coverage model because they are side effects of the tests rather than explicitly planned foci of the tests. These are harder to troubleshoot, especially if the failure happens in the field, because their replication conditions are complex. That doesn't make them less serious, just less easy to find with traditional, low-volume methods.

Inefficiency and Ineffectiveness of Software Testing: A Key Problem in Software Engineering

Retraining Software Testers

Despite the enormous proportion of test-related work (and staff) in software engineering projects, few universities offer more than a one-semester introduction to software testing (many offer none). Most training in testing will continue to be on-the-job.

Software testing is best understood as an empirical investigation of the product under test, done to provide quality-related information to stakeholders.

It takes a challenging combination of testing skill and subject matter expertise to pick areas of emphasis for testing and tests that would be maximally effective for exposing critical information in a credible way. There are many different subject expertises. One, for example, is the knowledge of platform, language, implementation style and details (and the risks associated with them) that risk-aware programmers on the project might have. Another might be a deep understanding of the target configurations and how variations in the execution environment might affect software behavior. Several more include the different perspectives of different subpopulations of users. The ideal test team is highly diversified, able to attack the product from several different perspectives, applying several different areas of subject matter expertise. Few or none of these people will have received a strong testing education while they were developing their subject matter expertise. They start learning about testing when they start testing.

Commercial instruction in software testing does not necessarily provide strong value. The challenge is that in a 3-day class, there is almost no time for practice, application to the tester's own environment, personalized coaching, or development of skill. The short courses convey an overview of an area, perhaps instill enthusiasm in the students, and may provide some good examples, but the real incorporation of the material into the tester's knowledgebase and skillbase will happen after the class is done—if it happens at all.

University training spreads courses over a much longer period, with homework and assessments to guide the student and give feedback on development of knowledge and skill. This is effective, but impractical for commercial courses. Few instructors would fly into town twice a week for a one or two hour presentation.

Online and hybrid (online-plus-live) approaches are making a new approach to testing education viable. The NSF-funded materials at www.testingeducation.org/BBST illustrate a course developed academically that is being used, royalty free, at several companies who spread their classes out over significant periods, use in-house staff (such as a test manager) as the course facilitator, and have students try to apply each new technique or key lesson to their current project.

Such materials are available to the public (including DoD) for free under a Creative Commons license. As with open source software, which provides software for free but support for a fee, DoD might sometimes look to training vendors for support in training test managers to lead the course, in developing customized assessments, or in developing new materials (video and print) for specialized versions of the course. The cost of such support is probably much less than the cost of sending staff to external, commercial training.

Inefficiency and Ineffectiveness of Software Testing: A Key Problem in Software Engineering

Reworking Test Documentation Practices

It is common to write immense documents that describe all of the details of the testing effort. In terms of industry standards and alleged best practices, this is not changing. The latest draft revisions for IEEE 829 present a long list of highly detailed documents.

This has some benefits, but carries several risks:

- The budget for test-related work is finite, but the task is infinite. Money spent on test documentation is not available for tool development, test creation, execution or interpretation. Documentation has value, but it competes with many other test-related activities of value. There has to be a sensible balancing of investment.
- The more details of the implementation we store in our test case description, the more things we have to change whenever the software changes. A painstakingly detailed test library becomes a source of project inertia, of maintenance costs that discourage improvement of the product.
- Early in development, a sensible test might check one value for one field as a complete test. For example, this is how we do traditional boundary value testing. If the software is unstable, narrowly-scoped tests provide stronger troubleshooting support. As the program stabilizes, single tests of single fields can be usefully combined into tests that supply values to many fields at once. The combinatorial explosion of possible tests for multiple variables tested together means that we probably can't find the time to run all the possible tests. However, we can achieve more in the time available if we can automate the process of generating test case inputs and of comparing the result to the behavior of a reference system, product or model. Documenting each of these tests creates a cost explosion.
- Providing detailed descriptions of a block of tests that provide at least one test per requirement doesn't achieve much of value. There are thousands of possible tests for most requirements. Wiser documentation might show the logic of the possible approaches to testing of that requirement providing a context for the test(s) actually run.

We need a new approach that is designed with the explicit intent of providing an appropriate return on investment.

- Certainly, documentation should support audits, provide records for litigation and regulatory investigation—to the extent that these are relevant to a particular project.
- The issue of maintenance support (support for maintenance of tests and for ongoing evolution of the product under test) must be separated from contract-related or liability-related recordkeeping. For maintenance support, design strategy might be more valuable than implementation details. Test idea matrices might be more value than step by step description of tests drawn from the (undocumented) matrix. Support for ad hoc recombination of tests to increase test power as the program gets more stable might be more valuable than details of any particular test.