28TH ANNUAL
PACIFIC NW
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010

ACHIEVING QUALITY IN A COMPLEX ENVIRONMENT

Conference Paper Excerpt from the

CONFERENCE PROCEEDINGS

Don't test too much! [or too little]
(Lessons learned the hard way)

Keith Stobie
Keith.Stobie@microsoft.com

Abstract

Over-testing can actually be as bad for you as under-testing. Whether it is testing to your own lofty expectations (versus those of the project), verifying too much in stress, or verifying too much in one test, you might end up with a less useful result than other approaches. You must also be careful of under-testing sequence and state or aspects of the software that are just hard to verify.

This experience report relates parables of mistakes I made and what I learned in the process, so you can avoid these mistakes. These lessons learned are also cross indexed to the book: *Lessons Learned in Software Testing* (Kaner, Bach, and Pettichord).

Biography

Keith Stobie is a Test Architect for Bing Infrastructure at Microsoft where he plans, designs, and reviews software architecture and tests. Previously, Keith worked in the Protocol Engineering Team on Protocol Quality Assurance Process including model based testing (MBT) to develop test framework, harnessing, and model patterns. With twenty-five years of distributed systems testing experience Keith's interests are in testing methodology, tools technology, and quality process. Keith has a BS in computer science from Cornell University.
ASQ Certified Software Quality Engineer, ASTQB Foundation Level
Member: ACM, IEEE, ASQ

1 Introduction

The many times I've taught various aspects of software testing, I've found many students learn best by hearing stories that demonstrate a principle in action. During my years of testing applications I have gathered a number of scars that I would prefer newcomers to the discipline avoid. Many of my lessons learned occurred before a book called *Lessons Learned in Software Testing* [Kaner 2001] was ever published. But in reading the book I could see how my experiences related to those in the book and correlate them as lesson #. The majority of the lessons are a direct result of my mistakes around over-testing!

2 Over- and Under-testing Experiences

What is over-testing? How can you test too much? These are typically quandaries from new testers who don't understand the business context in which most testing is done. Many books on testing state that testing is an infinite task. Business drivers dictate finite resources; the choice of what to test and how extensively is critical. Over-testing one area means you have used resources that may be better spent testing other areas.

I have learned many testing techniques over the years and have a catalog of over 50 different analysis methods. Why do we need so many techniques and analysis methods? Because software rarely shows its quality through only one aspect. The principle from Kaner's book that I most believe in is lesson #283, "Apply diverse half measures". This is also illustrated in the Manager's Guide to Evaluating Test Suites [Marick 2000], reporting on a very defect-prone portion while ignoring the rest of the product.

2.1 Verify Stress Tests Out of Band

A major difference between functional and stress tests is that stress tests typically run in a constrained resource environment. Stress wants as much (or even too much) work done as the system is capable of in the least amount of time, by consuming all of the CPU cycles, all of the I/O bandwidth, etc. Running functional tests in a loop has been shown not to be the most effective stress test.

Many systems must have much larger test driver systems than the actual system under test. For example, they might need 4 client machines to keep 1 server machine busy. Frequently the driver tests are not coded for optimal performance. Driver tests may cause unnecessary and useless work, for example driving a browser UI to generate HTTP traffic for a server. Driver tests may also make redundant or unnecessary verifications and thus they may check too much.

One of the earliest, most dramatic examples of this was when I tested a transaction system. Transaction systems let multiple concurrent users, applications, or threads, simultaneously access a resource, canonically a database system, with each getting their own consistent view of the resource. To do this, a Transaction system maintains Atomicity, Consistency, Isolation, and Durability, or ACID properties, of the results. I wrote a test system a model as it were - that mirrored the logical result of each operation, so that I could tell the transaction system did the same thing as my model. This worked great for functional testing. However, my model operated much slower than the real system, and thus concurrency was actually quite limited when used in a stress test.

Another engineer chose a much more clever approach. To measure Isolation, each database change was tagged with the test instance making the change. To measure Atomicity and Consistency the changes within a transaction were deliberately inconsistent with only the ending result restoring consistency. For example, in an ATM transaction when transferring money from Savings to Checking, you have two operations within a transaction. One operation debits savings and the other operation credits checking, but either both operations or neither (atomicity) must complete to keep consistency.

We could include a second set of operations within the same transaction, e.g. transferring money from Checking to Savings. Now, if we transferred the same amount both ways, we could not tell if all the

operations or none of the operations succeeded. However, by checking a consistency property of the result, we can. Our test decides it will maintain all account balances as evenly divisible by 1,000. Suppose Checking and Savings both start with 10,000 as their balance. For the first transfer, we can randomly choose any amount to transfer, e.g. 1,234.57. However this would leave our balances not evenly divisible by 1,000. We create a consistency-preserving second transfer. It could transfer more money, e.g.

First Savings -> Checking 1,234.57
Second Savings -> Checking 765.43

The completion of this transaction leaves savings 8,000 and checking at 12,000 which are both evenly divisible by 1,000, but if any 1, 2 or 3 operations within the transaction were not complete, then consistency was lost. Instead of a second addition, we could have done a subtraction, e.g.

First Savings -> Checking 1,234.57
Second Checking -> Savings 234.57

The complete of this transaction leaves savings at 9,000 and checking at 11,000, both evenly divisible by 1000. Again, if any 1, 2 or 3 operations within the transaction were not complete, then consistency was lost.

Now independent operations could randomly build sets of operations that each measured consistency individually. The tests could run as many operations as fast as the system could handle for as long as you wanted to test. At the end of the test, a simple check would tell you if consistency (and atomicity and Isolation) had been maintained. Diagnosing the failure was far more difficult. If consistency was lost, you would have to use the data logged by the tests into the test data for clues. You could also use the logs kept by the transaction system and its notion of roll-back to isolate when the failure occurred.

The inability to pinpoint the time and location allowed for much greater concurrency with limited resources. The first several times this approach was run numerous race conditions never exposed by the earlier method, which checked results in real time, were exposed.

Lesson – my original functional transaction validator validated too well. It assumed too much knowledge which limited its speed while consuming great resources. A simpler, out-of-band verification worked much better.

2.2 Test Oracle Complexity <= System Under Test Complexity

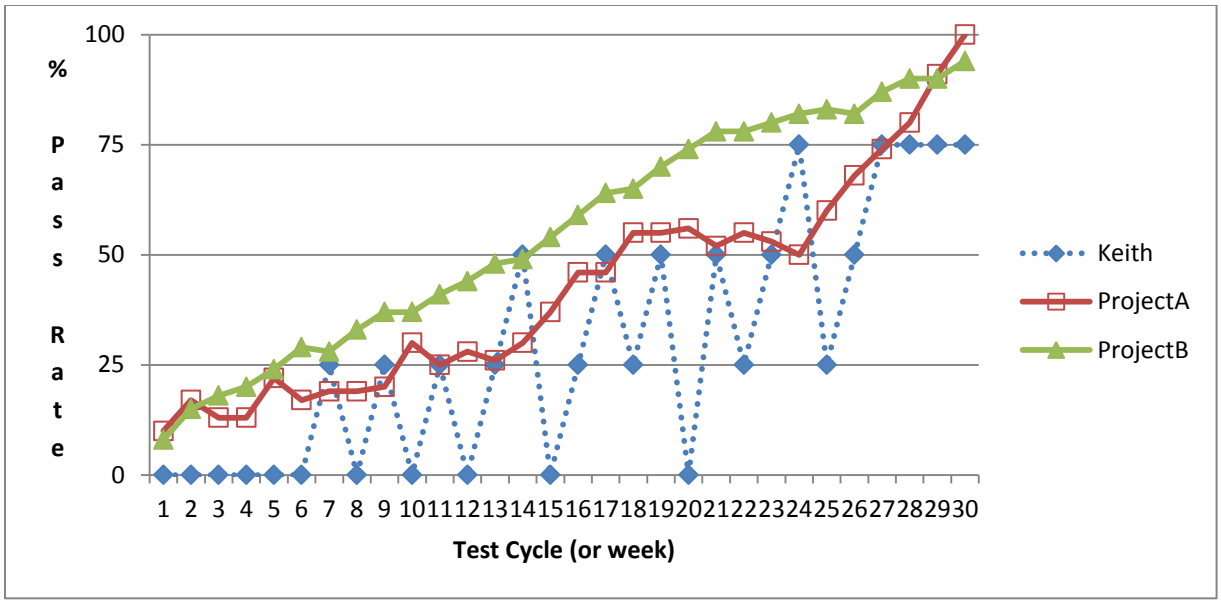
Another rule I've learned, which this example illustrates, is your test oracle for determining correct results should never be more complicated than the system you are testing. Test oracles are just as error-prone (if not more error-prone) as a system under test. The worst case test oracle should be doing what is called n-version programming. Create two (or more) versions of a system and compare the results. There are numerous papers on this approach and its limitations (e.g. both implementations sharing bugs from a buggy specification document). My simplest example of this is testing the Square Root function. Test shouldn't create another square root function to compare. Test should use a simpler oracle, which is just to square the result and compare with the original!

Related ideas for partial test oracles can be found under topics like metamorphic testing [Chen 2004, Hu 2006], sampling oracles [Hoffman 1998], or consistency oracles [Hoffman 2006].

2.3 Test in the Presence of Failures

I have never shipped a major piece of software defect free. Rather than explaining why it is OK to ship with bugs, I will concentrate on the testing implications of the fact.

I have one development manager who will never forgive me for the grief I caused their project. I wrote only four rather large "test cases". These four test cases covered all of the functionality of the project. However, the company expected teams to report percentage of test cases passing (as though that is a useful number). While most teams showed pretty wavy lines typically growing towards 90% or better passing rate, my development manager had a nasty step function, for example the dashed line below:



Part of the problem is using the amorphous, ill-defined, "test case" as a unit of measure (lesson #280 "How to lie with test cases"). I've seen test cases vary from 5 lines of text or code to hundreds of pages of text and thousands of lines of code. A better measure relates to number of requirements or features shown to be working (lesson #34 "It Works" really means it appears to meet some requirement to some degree.)

The other problem is each of my test cases was potentially testing too much. When I came to Microsoft and started using formal Model Based Testing tools this became most starkly evident. Research liked the idea of generating a single test case that covered everything in a model. That is, they would generate only one test case to verify everything! On the surface this seemed great to them; If this test case passes, you know your product has no detected bugs and if it fails, you still have bugs to remove. You can get this behavior today using the "long test" strategy with Spec Explorer for Visual Studio which attempts to generate a single test to cover all transitions in the model.

The danger of my four test cases or auto-generating a single all-encompassing test case is they test too much. The simplest way to design test cases, like reliable software, is to fail fast. As soon as an error is detected, note the error and stop any further processing. Most unit testing frameworks like xUnit (http://en.wikipedia.org/wiki/XUnit) or MSTest (http://en.wikipedia.org/wiki/MSTest) incorporate this notion into their Assert statements. When an Assert fails, an exception is raised and reported. The exception stops further progress. Combining fail fast testing with only a few large test cases is a risky combination. Once a test case stops due to failure, you lose any information about other work the test case might have performed. For example, suppose a test case has 100 asserts. If the tenth assert fails, you do not know the result of the remaining 90 asserts. The failure blocks your ability to learn more information. You can work around this by changing the test case (expecting the current behavior, commenting out the offending assert, etc.), but now you have two test cases: the original and the modified.

Thus, one way to find out if you are testing too much in test cases is if you have too many blocking failures. There are numerous other reasons for not testing too much in a single test case. Test Prioritization tools work best in minimizing testing amongst many test cases that don't overlap too much. Automatic Failure Analysis tools work better when test cases are likely to find different failures instead of the same failure multiple ways. With multiple test cases (instead of one), you can run more tests in parallel using more hardware and potentially reduce your overall testing time.

Note, that I am also not a proponent of very small test cases (other than for unit testing). I like Marick's advice:

> "If you blend ideas into a more complex test, you stand a chance of inadvertently implementing a test idea that finds a bug." [Marick 2000]

and Kaner's "Appropriately complex" [Kaner 2003] concept. Choosing a reasonable test case size and complexity is important and difficult. Unit tests should be very small. Once the unit tests and perhaps basic functional tests have shown the software sufficiently stable, moderately complex tests should be used. Very complex tests, while sometimes useful, are usually difficult to maintain and to diagnose software failures from.

2.4 You Must Understand the Goals of Your Release

One of my areas of expertise is distributed transaction systems. I first did testing for Tandem's distributed transaction manager. The test suite became one of my examples of "test driven" development. When Tandem decided to rewrite the transaction system when moving from 16-bit to a 32-bit system, the developers found the documentation lacking and the code too confusing. They used the test suite as their oracle of what to build. They assumed that if they built a 32-bit version of the system with the same test results as the 16-bit system, then customers should see no difference when moving their code to the new system – and they didn't!.

I then worked on a distributed database with transactions at Informix. Finally, I was hired by BEA Systems which focuses on transactions for databases. BEA had purchased the industry standard Tuxedo transaction system [Andrade 1996] used in many vendors benchmarks. BEA was creating a Java Transactions Application Programming Interfaces (APIs) which became part of Java Second Enterprise Edition (J2EE). In designing the testing of the system, we naturally wanted to assure all of the ACID properties. With only a couple weeks before our first Beta release to customers, I and the Development Manager were shocked and discouraged to learn that the Java transaction system did not provide isolation or consistency when more than one thread was in use! We thought we had a disaster on our hands. As lesson #12 says, "Never be the gate keeper". We presented our findings to the project manager and asked if we needed to slip the release. While the project manager was discouraged by the findings, he had a better grasps of the goal of the release. We already knew the underlying Tuxedo system was robust and reliable. We were also confident that the coding flaws were not deep design flaws in our new Java interface. The purpose of the Beta was not to go into production, but to gather feedback from real users about the usability of the new Java Transactions API (JTA). Concurrent usage was not necessary to evaluate the usability of the APIs for programmers (i.e. programmers just code and execute one thread. They couldn't run multiple copies of the thread or different threads). By simply noting in the release notes that the Beta did not support concurrent users, the Beta could still achieve its primary purpose of gathering feedback on the APIs. At the same time, while Beta customers were getting used to a new API they had never seen before (this was the early days of Java), we could make the code fixes and verify concurrent usage.

The release of a product is a business decision. A test group provides information to make that decision (see lesson #205 "Don't sign-off to approve the release of a product"). Test shouldn't say "don't ship because we can't handle concurrent users". Test should state the risks as they see them; "Shipping now would result in customers being unable to do concurrent operations". The business leaders based on information from many sources choose if the benefits of shipping outweigh the risks. Note: professional ethics encourages you to make sure customers understand what they are (or are not) getting [Hall 2009 and Berenbach 2009].

2.5 Under-testing Sequence and State

Many new testers tend to treat what they are testing in a very isolated way. While this may make it easier to think about, it frequently obscures many possible failures. Lesson #25, "All testing is based on models" is a key idea for testers. Most well-engineered systems use information hiding (http://en.wikipedia.org/wiki/Information_hiding) to encapsulate state. Testers need to have an intuition (or perhaps direct knowledge from reading the code) of the hidden state. This will reveal many more tests and possible product failures.

The incident that brought this home to me was testing the new Enterprise Java Beans (EJB, http://java.sun.com/products/ejb/docs.html). My testers had already done a good job of verifying correct sequences worked and that errors were correctly reported. But that is not enough. In most object oriented system with Exception mechanisms, an exception can ripple through the code ("writing the code that lies in between the" throw and the catch – Griffiths 2000) causing havoc ("no well-defined techniques exist for building robust exception handling into a system." – Shelton 1999). We got a bug report back indicating exceptions didn't work. It was due to the fact that a previous exception had corrupted state.

The issue with our EJB testing was we hadn't tested for sequences with errors. Does the system continue to work ***after*** returning an error? For details, see Stobie 2000, but the answer is that a short sequence can cover many requirements. The 6 requirements considered are:

1. first call - normal return
2. first call - exception raised
3. previous call raised exception - current call raises exception
4. previous call raised exception --current call returns normally
5. previous call returns normally--current call raises exception
6. previous call returns normally--current call returns normally

The first call is considered special as it goes from a never-called state to an initial state. The sequence below covers 5 of the requirements with just 7 calls.

1. Normal return from Call (Requirement 1)
2. Exception from Call (Requirement 5)
3. Normal return from Call (Requirement 4)
4. Exception from Call (Requirement 5 again)
5. Exception from Call (Requirement 3)
6. Normal return from Call (Requirement 4 again)
7. Normal return from Call (Requirement 6)

Requirement 2 and Requirement 1, by definition, can't be covered in the same test case. Requirement 6 is actually covered in most other test cases and thus not really needed here.

2.6 Skipping Coverage of the Hard Stuff

In a system logging project I worked on, I was able to cover every line of code for the logging system I was testing except under the unusual condition when the log was on a remote system using a deprecated file system. I convinced my management that this unlikely occurrence was not worth the set up costs to increase the coverage. They agreed. The only bug reported? Using a remote deprecated file system didn't work!

I misjudged the risk and convinced my management of little risk. What I would do differently in the future is consider alternative means of verification. If I can't economically set up and execute the code, then the code deserves a more careful and detailed code review and inspection (which would have caught this particular bug). Alternatively, today you should consider making sure you can mock the system being called so the code is at least thoroughly unit tested.

2.7 Over and under-testing combinations

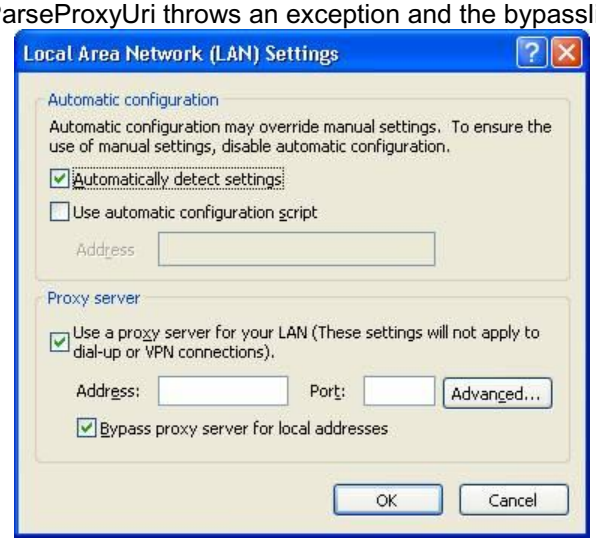
Many testers waver between testing too little by only verifying each value independently and perhaps over-testing by verifying all combinations of values even with little risk of interactions or failures. If testing all combinations is inexpensive (for example looping through a single API call that takes 1 microsecond to execute with various values) then testing all combinations is probably cheaper than even doing the analysis for less testing. However, when the combinations represent the setup or configuration of a system which may involve long periods of time to install the correct software and get it to the correct state, then combinatorics comes into play.

Empirical studies (along with a lot of war stories) show that assuming no interaction misses a number of defects. However, it also appears that given the number of additional tests needed to cover all combinations, the consequent reduction in risk isn't typically worth the cost. That is, you would be better

spending your test budget doing other things than verifying all combinations. If you know of interactions, you should test for them (see 2.5 Under-testing sequence and state above). It is only when testing for unknown interactions that combinatorics most applies. Doing pairwise testing (http://www.pairwise.org/), the all pairs approach, or using a 2-covering array (all synonyms) can help. It appears to be the best compromise between number of test cases and likelihood of revealing unexpected failure causing interactions. You can use 3-covering or high order arrays, but the number of tests grows very quickly while the likelihood of finding new failures appears to also drop quickly.

In teaching pairwise testing, I've helped other testers quickly determine over and under-testing. In one case a complicated distributed transaction system (COM+) had many configurations depending on the versions of the components. They had 80 configurations they tested and were still getting bug reports. After learning pairwise testing and applying to their array of configurations, they discovered only 40 configurations were called for by pairwise testing and the recently reported bug would have been uncovered if they had used this reduced set of 40 configurations.

Another case was the testing of a simple API that also had a user interface component. After learning pairwise testing and applying to their API for configuring a LAN, they found a bug that had existed for the past three shipped releases. Each input could take multiple values (checked/unchecked, strings, and Uris). One object, under a particular combination, did not reflect the fact that its check box was checked. If the proxy Uri is null, the ParseProxyUri throws an exception and the bypasslist is not processed.



While testing another application (Attrib.exe), another Windows team found that 13 test cases generated via pairwise analysis achieved block coverage identical to 370 exhaustive test cases.

2.8 Summary

Lessons Learned in Software Testing [Kaner 2001] presents a *Five-fold Testing System* Classification system for testing techniques with overlapping technique categories:

Who: Developers, testers, internal users, beta users, etc.
What: Coverage – Requirements, scenarios, functions, code, errors, etc.
Why: Potential problems – Risks you are testing for.
How: Activities – Regression, exploration, installation, etc.
Evaluation: How to tell whether the test passed or failed.

The key to verifying stress tests out of band lies in carefully choosing your evaluation technique, or oracle. Testing in the presence of failures covers not only evaluation techniques, but also how much coverage for each test case. Understanding the goals of your release covers all of the classifications – who will use your release (beta customers), what coverage do they need (good, clean APIs), what potential problems must be uncovered (API failures preventing usage of other APIs), and how you

evaluate if your product is ready for release. Under-testing sequence and state relates to what (coverage of the sequence and state) and why (potential problems in state corruption). Understand your coverage (what) of interactions (why) and how to most effectively test them. Finally, it is mostly about how you achieve coverage – code review, unit testing, system testing, customer testing, etc.

3 Conclusion

You can test too much in one test case causing unnecessary blocking due to bugs. You can test too much for a release by expecting more than its requirements. You can verify too much during stress testing resulting in less stress testing or overly expensive stress testing.

Testing is a business activity with a cost. Using your resources wisely to most effectively cover the risks requires you to:

- understand the product and release goals,
- deal with common issues like known errors,
- anticipate likely errors
- choose the right evaluation methods

References

Andrade, J., et al 1996 *The TUXEDO System: Software for Constructing and Managing Distributed Business Applications*, Addison-Wesley Professional, 1996

Chen, T. et al, 2004, Case *Studies on the Selection of Useful Relations in Metamorphic Testing*, HKU CS Tech Report TR-2004-13, http://www.csis.hku.hk/research/techreps/document/TR-2004-13.pdf

Berenbach, B. and M. Broy 2009, *Professional and Ethical Dilemmas in Software Engineering*, IEEE Computer, January 2009

Griffiths, A. 2000, *Here be Dragons*, http://www.octopull.demon.co.uk/c++/dragons/

Hall, D., 2009, *The Ethical Software Engineer*, IEEE Software July 2009

Hoffman, D., 1998, *A Taxonomy for Test Oracles*, Quality Week 1998
http://www.softwarequalitymethods.com/Slides/Orcl_Tax_Slides.PDF
http://www.softwarequalitymethods.com/Papers/OracleTax.pdf

Hoffman, D. 2006, *Using Oracles in Testing and Test Automation*, LogiGear newsletter.
http://www.logigear.com/newsletter/using_oracles_in_testing_and_test_automation_part-1.asp

Hu, P. et al 2006, *An empirical comparison between direct and indirect test result checking approaches*, Proceedings of the 3rd international workshop on Software quality assurance Pages: 6 – 13, ISBN:1-59593-584-3

Kaner, C., J. Bach and B. Pettichord. 2001, *Lessons Learned in Software Testing*, Wiley

Kaner, C. 2003, *What Is a Good Test Case?*, STAR East, May 2003
http://www.kaner.com/pdfs/GoodTest.pdf

Marick, B, J. Bach, and C. Kaner 2000, *Manager's Guide to Evaluating Test Suites,* Quality Week 2000
http://www.exampler.com/testing-com/writings/evaluating-test-suites-paper.pdf

Marick, B. 2000, *Testing For Programmers*, (pages 68-72)
http://www.exampler.com/testing-com/writings/half-day-programmer.pdf

Shelton, C. 1999, Exception Handling, 18-849b Dependable Embedded Systems, Spring 1999
http://www.ece.cmu.edu/~koopman/des_s99/exceptions/

Stobie, K. 2000, *Testing for Exceptions*, Software Testing & Quality Engineering July 2000.
http://www.testingcraft.com/stobie-exceptions.pdf