What Is a Good Test Case?

Cem Kaner, J.D., Ph.D.
Florida Institute of Technology
Department of Computer Sciences
`kaner@kaner.com`

STAR East, May 2003

This research was partially supported by NSF Grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers." Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Abstract

Designing good test cases is a complex art. The complexity comes from three sources:

- Test cases help us discover information. Different types of tests are more effective for different classes of information.
- Test cases can be "good" in a variety of ways. No test case will be good in all of them.
- People tend to create test cases according to certain testing styles, such as domain testing or risk-based testing. Good domain tests are different from good risk-based tests.

What's a Test Case?

Let's start with the basics. What's a test case?

IEEE Standard 610 (1990) defines *test case* as follows:

> "(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
>
> "(2) (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item."

According to Ron Patton (2001, p. 65),

> "Test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software."

Boris Beizer (1995, p. 3) defines a test as

> "A sequence of one or more subtests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next. The word 'test' is used to include subtests, tests proper, and test suites.

Bob Binder (1999, p. 47) defines test case:

> "A test case specifies the pretest state of the IUT and its environment, the test inputs or conditions, and the expected result. The expected result specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, returned values, and resultant state of the IUT and its environment. Test cases may also specify initial and resulting conditions for other objects that constitute the IUT and its environment."

In practice, many things are referred to as test cases even though they are far from being fully documented.

Brian Marick uses a related term to describe the lightly documented test case, the *test idea*:

> "A test idea is a brief statement of something that should be tested. For example, if you're testing a square root function, one idea for a test would be 'test a number less than zero'. The idea is to check if the code handles an error case."

In my view, a test case is a question that you ask of the program. The point of running the test is to gain information, for example whether the program will pass or fail the test.

It may or may not be specified in great procedural detail, as long as it is clear what is the idea of the test and how to apply that idea to some specific aspect (feature, for example) of the product. If the documentation is an essential aspect of a test case, in your vocabulary, please substitute the term "test idea" for "test case" in everything that follows.

An important implication of defining a test case as a question is that *a test case must be reasonably capable of revealing information*.

- Under this definition, the *scope* of test cases changes as the program gets more stable. Early in testing, when anything in the program can be broken, trying the largest "legal" value in a numeric input field is a sensible test. But weeks later, after the program has passed this test several times over several builds, *a standalone test of this one field is no longer a test case because there is only a miniscule probability of failure*. A more appropriate test case at this point might combine boundaries of ten different variables at the same time or place the boundary in the context of a long-sequence test or a scenario.
- Also, under this definition, the metrics that report the number of test cases are meaningless. What do you do with a set of 20 single-variable tests that were interesting a few weeks ago but now should be retired or merged into a combination? Suppose you create a combination test that includes the 20 tests. Should the metric report this one test, 20 tests, or 21? What about the tests that you run only once? What about the tests that you design and implement but never run because the program design changes in ways that make these tests uninteresting?

Another implication of the definition is that a test is not necessarily designed to expose a defect. The goal is information. Very often, the information sought involves defects, but not always. (I owe this insight to Marick, 1997.) To assess the value of a test, we should ask how well it provides the information we're looking for.

Information Objectives

So what are we trying to learn or achieve when we run tests? Here are some examples:

- ***Find defects.*** This is the classic objective of testing. A test is run in order to trigger failures that expose defects. Generally, we look for defects in all interesting parts of the product.
- ***Maximize bug count.*** The distinction between this and "find defects" is that total number of bugs is more important than coverage. We might focus narrowly, on only a few high-risk features, if this is the way to find the most bugs in the time available.
- ***Block premature product releases.*** This tester stops premature shipment by finding bugs so serious that no one would ship the product until they are fixed. For every release-decision meeting, the tester's goal is to have new showstopper bugs.
- ***Help managers make ship / no-ship decisions.*** Managers are typically concerned with risk in the field. They want to know about coverage (maybe not the simplistic code coverage statistics, but some indicators of how much of the product has been addressed and how much is left), and how important the known problems are. Problems that appear significant on paper but will not lead to customer dissatisfaction are probably not relevant to the ship decision.
- ***Minimize technical support costs.*** Working in conjunction with a technical support or help desk group, the test team identifies the issues that lead to calls for support. These are often peripherally related to the product under test--for example, getting the product to work with a specific printer or to import data successfully from a third party database might prevent more calls than a low-frequency, data-corrupting crash.
- ***Assess conformance to specification.*** Any claim made in the specification is checked. Program characteristics not addressed in the specification are not (as part of *this* objective) checked.
- ***Conform to regulations.*** If a regulation specifies a certain type of coverage (such as, at least one test for every claim made about the product), the test group creates the appropriate tests. If the regulation specifies a style for the specifications or other documentation, the test group probably checks the style. In general, the test group is focusing on anything covered by regulation and (in the context of *this objective*) nothing that is not covered by regulation.
- ***Minimize safety-related lawsuit risk.*** Any error that could lead to an accident or injury is of primary interest. Errors that lead to loss of time or data or corrupt data, but that don't carry a risk of injury or damage to physical things are out of scope.
- ***Find safe scenarios for use of the product (find ways to get it to work, in spite of the bugs).*** Sometimes, all that you're looking for is one way to do a task that will consistently work--one set of instructions that someone else can follow that will reliably deliver the benefit they are supposed to lead to. In this case, the tester is not looking for bugs. He is trying out, empirically refining and documenting, a way to do a task.
- ***Assess quality.*** This is a tricky objective because quality is multi-dimensional. The nature of quality depends on the nature of the product. For example, a computer game that is rock solid but not entertaining is a lousy game. To *assess* quality -- *to measure and report back* on the level of quality -- you probably need a clear definition of the most important quality criteria for this product, and then you need a theory that relates test results to the definition. For example, *reliability* is not just about the number of bugs in the product. It is (or is often defined as being) about the number of reliability-related failures that can be

expected in a period of time or a period of use. (*Reliability-related?* In measuring reliability, an organization might not care, for example, about misspellings in error messages.) To make this prediction, you need a mathematically and empirically sound model that links test results to reliability. Testing involves gathering the data needed by the model. This might involve extensive work in areas of the product believed to be stable as well as some work in weaker areas. Imagine a reliability model based on counting bugs found (perhaps weighted by some type of severity) per N lines of code or per K hours of testing. Finding the bugs is important. Eliminating duplicates is important. Troubleshooting to make the bug report easier to understand and more likely to fix is (*in the context of assessment)* out of scope.

- ***Verify correctness of the product.*** It is impossible to do this by testing. You can prove that the product is *not* correct or you can demonstrate that you didn't find any errors in a given period of time using a given testing strategy. However, you can't test exhaustively, and the product might fail under conditions that you did not test. The best you can do (if you have a solid, credible model) is assessment--test-based estimation of the probability of errors. (See the discussion of reliability, above).
- ***Assure quality.*** Despite the common title, *quality assurance,* you can't assure quality by testing. You can't assure quality by gathering metrics. You can't assure quality by setting standards. Quality assurance involves building a high quality product and for that, you need skilled people throughout development who have time and motivation and an appropriate balance of direction and creative freedom. This is out of scope for a test organization. It is within scope for the project manager and associated executives. The test organization can certainly help in this process by performing a wide range of technical investigations, but those investigations are not quality assurance.

Given a testing objective, the good test series provides information directly relevant to that objective.

Tests Intended to Expose Defects

Let's narrow our focus to the test group that has two primary objectives:

- Find bugs that the rest of the development group will consider relevant (worth reporting) and
- Get these bugs fixed.

Even within these objectives, tests can be *good* in many different ways. For example, we might say that one test is better than another if it is:

- ***More powerful.*** I define power in the usual statistical sense as more likely to expose a bug if it the bug is there. Note that Test 1 can be more powerful than Test 2 for one type of bug and less powerful than Test 2 for a different type of bug.
- ***More likely to yield significant (more motivating, more persuasive) results***. A problem is *significant* if a stakeholder with influence would protest if the problem is not fixed. (A stakeholder is a person who is affected by the product. A stakeholder with influence is someone whose preference or opinion might result in change to the product.)
- ***More credible***. A credible test is more likely to be taken as a realistic (or reasonable) set of operations by the programmer or another stakeholder with influence. "Corner case" is

an example of a phrase used by programmers to say that a test or bug is non-credible: “No one would do that.” A test case is credible if some (or all) stakeholders agree that it is realistic.

- ***Representative of events more likely to be encountered by the customer***. A *population* of tests can be designed to be highly credible. Set up your population to reflect actual usage probabilities. The more frequent clusters of activities are more likely to be covered or covered more thoroughly. (I say *cluster of activities* to suggest that many features are used together and so we might track which combinations of features are used and in what order, and reflect this more specific information in our analysis.) For more details, read Musa's (1998) work on software reliability engineering.
- ***Easier to evaluate.*** The question is, did the program pass or fail the test? Ease of Evaluation. The tester should be able to determine, quickly and easily, whether the program passed or failed the test. It is not enough that it is *possible* to tell whether the program passed or failed. The harder evaluation is, or the longer it takes, the more likely it is that failures will slip through unnoticed. Faced with time-consuming evaluation, the tester will take shortcuts and find ways to less expensively guess whether the program is OK or not. These shortcuts will typically be imperfectly accurate (that is, they may miss obvious bugs or they may flag correct code as erroneous.)
- ***More useful for troubleshooting.*** For example, high volume automated tests will often crash the system under test without providing much information about the relevant test conditions needed to reproduce the problem. They are not useful for troubleshooting. Tests that are harder to repeat are less useful for troubleshooting. Tests that are harder to perform are less likely to be performed correctly the next time, when you are troubleshooting a failure that was exposed by this test.
- ***More informative.*** A test provides value to the extent that we learn from it. In most cases, you learn more from the test that the program passes than the one the program fails, but the informative test will teach you something (reduce your uncertainty) whether the program passes it or fails.
    - For example, if we have already run a test in several builds, and the program reliably passed it each time, we will expect the program to pass this test again. Another "pass" result from the reused test doesn't contribute anything to our mental model of the program.
    - The notion of equivalence classes provides another example of information value. Behind any test is a set of tests that are sufficiently similar to it that we think of the other tests as essentially redundant with this one. In traditional jargon, this is the "equivalence class" or the "equivalence partition." If the tests are sufficiently similar, there is little added information to be obtained by running the second one after running the first.
    - This criterion is closely related to Karl Popper’s theory of value of experiments (See Popper 1992). Good experiments involve risky predictions. The theory predicts something that many people would expect not to be true. Either your favorite theory is false or lots of people are surprised. Popper’s analysis of what makes for good experiments (good tests) is a core belief in a mainstream approach to the philosophy of science.

    - Perhaps the essential consideration here is that the expected value of what you will learn from this test has to be balanced against the opportunity cost of designing and running the test. The time you spend on this test is time you don't have available for some other test or other activity.

- ***Appropriately complex.*** A complex test involves many features, or variables, or other attributes of the software under test. Complexity is less desirable when the program has changed in many ways, or when you're testing many new features at once. If the program has many bugs, a complex test might fail so quickly that you don't get to run much of it. Test groups that rely primarily on complex tests complain of *blocking bugs*. A blocking bug causes many tests to fail, preventing the test group from learning the other things about the program that these tests are supposed to expose. Therefore, early in testing, simple tests are desirable. As the program gets more stable, or (as in eXtreme Programming or any evolutionary development lifecycle) as more stable features are incorporated into the program, greater complexity becomes more desirable.
- ***More likely to help the tester or the programmer develop insight into some aspect of the product, the customer, or the environment.*** Sometimes, we test to understand the product, to learn how it works or where its risks might be. Later, we might design tests to expose faults, but especially early in testing we are interested in learning *what it is* and *how to test it.* Many tests like this are never reused. However, in a *test-first design* environment, code changes are often made experimentally, with the expectation that the (typically, unit) test suite will alert the programmer to side effects. In such an environment, a test might be designed to flag a performance change, a difference in rounding error, or some other change that is not a defect. An unexpected change in program behavior might alert the programmer that her model of the code or of the impact of her code change is incomplete or wrong, leading her to additional testing and troubleshooting. (Thanks to Ward Cunningham and Brian Marick for suggesting this example.)

Test Styles/Types and Test Qualities

Within the field of black box testing, Kaner & Bach (see our course notes, Bach 2003b and Kaner, 2002, posted at www.testingeducation.org, and see Kaner, Bach & Pettichord, 2002) have described eleven dominant *styles* of black box testing:

- Function testing
- Domain testing
- Specification-based testing
- Risk-based testing
- Stress testing
- Regression testing
- User testing
- Scenario testing
- State-model based testing
- High volume automated testing

- Exploratory testing

Bach and I call these "paradigms" of testing because we have seen time and again that one or two of them dominate the thinking of a testing group or a talented tester. An analysis we find intriguing goes like this:

> If I was a "scenario tester" (a person who defines testing primarily in terms of application of scenario tests), how would I actually test the program? What makes one scenario test better than another? Why types of problems would I tend to miss, what would be difficult for me to find or interpret, and what would be particularly easy?

Here are thumbnail sketches of the styles, with some thoughts on how test cases are "good" within them.

Function Testing

Test each function / feature / variable in isolation.

Most test groups start with fairly simple function testing but then switch to a different style, often involving the interaction of several functions, once the program passes the mainstream function tests.

Within this approach, a good test focuses on a single function and tests it with middle-of-the-road values. We don't expect the program to fail a test like this, but it will if the algorithm is fundamentally wrong, the build is broken, or a change to some other part of the program has fowled this code.

These tests are ***highly credible*** and ***easy to evaluate*** but not particularly powerful.

Some test groups spend most of their effort on function tests. For them, testing is complete when every item has been thoroughly tested on its own. In my experience, the tougher function tests look like domain tests and have their strengths.

Domain Testing

The essence of this type of testing is sampling. We reduce a massive set of possible tests to a small group by dividing (partitioning) the set into subsets (subdomains) and picking one or two representatives from each subset.

In domain testing, we focus on variables, initially one variable at time. To test a given variable, the set includes all the values (including invalid values) that you can imagine being assigned to the variable. Partition the set into subdomains and test at least one representative from each subdomain. Typically, you test with a "best representative", that is, with a value that is at least as likely to expose an error as any other member of the class. If the variable can be mapped to the number line, the best representatives are typically boundary values.

Most discussions of domain testing are about input variables whose values can be mapped to the number line. The best representatives of partitions in these cases are typically boundary cases.

A good set of domain tests for a numeric variable hits every boundary value, including the minimum, the maximum, a value barely below the minimum, and a value barely above the maximum.

- Whittaker (2003) provides an extensive discussion of the many different types of variables we can analyze in software, including input variables, output variables, results of intermediate calculations, values stored in a file system, and data sent to devices or other programs.

- Kaner, Falk & Nguyen (1993) provided a detailed analysis of testing with a variable (printer type, in configuration testing) that can't be mapped to a number line.

These tests are ***higher power*** than tests that don't use "best representatives" or that skip some of the subdomains (e.g. people often skip cases that are expected to lead to error messages).

The first time these tests are run, or after significant relevant changes, these tests carry a lot of ***information value*** because boundary / extreme-value errors are common.

Bugs found with these tests are sometimes dismissed, especially when you test extreme values of several variables at the same time. (These tests are called *corner cases.*) They are not necessarily credible, they don't necessarily represent what customers will do, and thus they are not necessarily very motivating to stakeholders.

Specification-Based Testing

Check the program against every claim made in a reference document, such as a design specification, a requirements list, a user interface description, a published model, or a user manual.

These tests are ***highly significant*** (motivating) in companies that take their specifications seriously. For example, if the specification is part of a contract, conformance to the spec is very important. Similarly products must conform to their advertisements, and life-critical products must conform to any safety-related specification.

Specification-driven tests are often weak, not particularly powerful representatives of the class of tests that could test a given specification item.

Some groups that do specification-based testing focus narrowly on what is written in the document. To them, a good set of tests includes an unambiguous and relevant test for each claim made in the spec.

Other groups look further, for problems in the specification. They find that the most informative tests in a well-specified product are often the ones that explore ambiguities in the spec or examine aspects of the product that were not well-specified.

Risk-Based Testing

Imagine a way the program could fail and then design one or more tests to check whether the program will actually fail that in way.

A "complete" set of risk-based tests would be based on an exhaustive risk list, a list of every way the program could fail.

A good risk-based test is a ***powerful representative*** of the class of tests that address a given risk.

To the extent that the tests tie back to significant failures in the field or well known failures in a competitor's product, a risk-based failure will be ***highly credible*** and ***highly motivating***. However, many risk-based tests are dismissed as academic (unlikely to occur in real use). Being able to tie the "risk" (potential failure) you test for to a real failure in the field is very valuable, and makes tests more credible.

Risk-based tests tend to carry ***high information value*** because you are testing for a problem that you have some reason to believe might actually exist in the product. We learn a lot whether the program passes the test or fails it.

Stress Testing

There are a few different definition of stress tests.

- Under one common definition, you hit the program with a peak burst of activity and see it fail.
- IEEE Standard 610.12-1990 defines it as "Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements with the goal of causing the system to fail."
- A third approach involves driving the program to failure in order to watch *how* the program fails. For example, if the test involves excessive input, you don't just test near the specified limits. You keep increasing the size or rate of input until either the program finally fails or you become convinced that further increases won't yield a failure. The fact that the program eventually fails might not be particularly surprising or motivating. The interesting thinking happens when you see the failure and ask what vulnerabilities have been exposed and which of them might be triggered under less extreme circumstances. Jorgensen (2003) provides a fascinating example of this style of work.

I work from this third definition.

These tests have ***high power***.

Some people dismiss stress test results as not representative of customer use, and therefore not credible and not motivating. Another problem with stress testing is that a failure may not be useful unless the test provides good troubleshooting information, or the lead tester is extremely familiar with the application.

A good stress test pushes the limit you want to push, and includes enough diagnostic support to make it reasonably easy for you to investigate a failure once you see it.

Some testers, such as Alberto Savoia (2000), use stress-like tests to expose failures that are hard to see if the system is not running several tasks concurrently. These failures often show up well within the theoretical limits of the system and so they are more ***credible*** and more ***motivating***. They are not necessarily easy to troubleshoot.

Regression Testing

Design, develop and save tests with the intent of regularly reusing them, Repeat the tests after making changes to the program.

This is a good point (consideration of regression testing) to note that this is not an orthogonal list of test types. You can put domain tests or specification-based tests or any other kinds of tests into your set of regression tests.

So what's the difference between these and the others? I'll answer this by example:

Suppose a tester creates a suite of domain tests and saves them for reuse. Is this domain testing or regression testing?

- I think of it as primarily domain testing if the tester is primarily thinking about partitioning variables and finding good representatives when she creates the tests.
- I think of it as primarily regression testing if the tester is primarily thinking about building a set of reusable tests.

Regression tests may have been powerful, credible, and so on, when they were first designed. However, after a test has been run and passed many times, it's not likely that the program will fail it the next time, unless there have been major changes or changes in part of the code directly involved with this test. Thus, most of the time, regression tests carry little information value.

A good regression test is designed for reuse. It is adequately documented and maintainable. (For suggestions that improve maintainability of GUI-level tests, see Graham & Fewster, 1999; Kaner, 1998; Pettichord, 2002, and the papers at www.pettichord.com in general). A good regression test is designed to be likely to fail if changes induce errors in the function(s) or area(s) of the program addressed by the regression test.

User Testing

User testing is done by users. Not by testers pretending to be users. Not by secretaries or executives pretending to be testers pretending to be users. By users. People who will make use of the finished product.

User tests might be designed by the users or by testers or by other people (sometimes even by lawyers, who included them as acceptance tests in a contract for custom software). The set of user tests might include boundary tests, stress tests, or any other type of test.

Some user tests are designed in such detail that the user merely executes them and reports whether the program passed or failed them. This is a good way to design tests if your goal is to provide a carefully scripted demonstration of the system, without much opportunity for wrong things to show up as wrong.

If your goal is to discover what problems a user will encounter in real use of the system, your task is much more difficult. Beta tests are often described as cheap, effective user tests but in practice they can be quite expensive to administer and they may not yield much information. For some suggestions on beta tests, see Kaner, Falk & Nguyen (1993).

A good user test must allow enough room for cognitive activity by the user while providing enough structure for the user to report the results effectively (in a way that helps readers understand and troubleshoot the problem).

Failures found in user testing are typically ***credible*** and ***motivating.*** Few users run particularly powerful tests. However, some users run complex scenarios that put the program through its paces.

Scenario Testing

A scenario is a story that describes a hypothetical situation. In testing, you check how the program copes with this hypothetical situation.

The ideal scenario test is ***credible***, ***motivating***, ***easy to evaluate***, and ***complex***.

In practice, many scenarios will be weak in at least one of these attributes, but people will still call them scenarios. The key message of this pattern is that you should keep these four attributes in mind when you design a scenario test and try hard to achieve them.

An important variation of the scenario test involves a harsher test. The story will often involve a sequence, or data values, that would rarely be used by typical users. They might arise, however, out of user error or in the course of an unusual but plausible situation, or in the behavior of a hostile user. Hans Buwalda (2000a, 2000b) calls these "killer soaps" to distinguish them from normal scenarios, which he calls "soap operas." Such scenarios are common in security testing or other forms of stress testing.

In the Rational Unified Process, scenarios come from use cases. (Jacobson, Booch, & Rumbaugh, 1999). Scenarios specify actors, roles, business processes, the goal(s) of the actor(s), and events that can occur in the course of attempting to achieve the goal. A scenario is an instantiation of a use case. A simple scenario traces through a single use case, specifying the data values and thus the path taken through the case. A more complex use case involves concatenation of several use cases, to track through a given task, end to end. (See also Bittner & Spence, 2003; Cockburn, 2000; Collard, 1999; Constantine & Lockwood, 1999; Wiegers, 1999.) For a cautionary note, see Berger (2001).

However they are derived, good scenario tests have ***high power*** the first time they're run.

Groups vary in how often they run a given scenario test.

- Some groups create a pool of scenario tests as regression tests.
- Others (like me) run a scenario once or a small number of times and then design another scenario rather than sticking with the ones they've used before.

Testers often develop scenarios to ***develop insight*** into the product. This is especially true early in testing and again late in testing (when the product has stabilized and the tester is trying to understand advanced uses of the product.)

State-Model-Based Testing

In state-model-based testing, you model the visible behavior of the program as a state machine and drive the program through the state transitions, checking for conformance to predictions from the model. This approach to testing is discussed extensively at www.model-based-testing.org.

In general, comparisons of software behavior to the model are done using automated tests and so the failures that are found are found easily ***(easy to evaluate)***.

In general, state-model-based tests are ***credible, motivating*** and ***easy to troubleshoot***. However, state-based testing often involves simplifications, looking at transitions between operational modes rather than states, because there are too many states (El-Far 1995). Some abstractions to operational modes are obvious and credible, but others can seem overbroad or otherwise odd to some stakeholders, thereby reducing the value of the tests. Additionally, if the model is oversimplified, failures exposed by the model can be difficult to troubleshoot (Houghtaling, 2001).

Talking about his experiences in creating state models of software, Harry Robinson (2001) reported that much of the bug-finding happens while doing the modeling, well before the automated tests are coded. Elisabeth Hendrickson (2002) trains testers to work with state models as an exploratory testing tool--her models might never result in automated tests, their value is that they guide the analysis by the tester.

El-Far, Thompson & Mottay (2001) and El-Far (2001) discuss some of the considerations in building a good suite of model-based tests. There are important tradeoffs, involving, for example, the level of detail (more detailed models find more bugs but can be much harder to read and maintain). For much more, see the papers at www.model-based-testing.org.

High-Volume Automated Testing

High-volume automated testing involves massive numbers of tests, comparing the results against one or more partial oracles.

- The simplest partial oracle is running versus crashing. If the program crashes, there must be a bug. See Nyman (1998, 2002) for details and experience reports.
- State-model-based testing can be high volume if the stopping rule is based on the results of the tests rather than on a coverage criterion. For the general notion of stochastic state-based testing, see Whittaker (1997). For discussion of state-model-based testing ended by a coverage stopping rule, see Al-Ghafees & Whittaker (2002).
- Jorgensen (2002) provides another example of high-volume testing. He starts with a file that is valid for the application under test. Then he corrupts it in many ways, in many places, feeding the corrupted files to the application. The application rejects most of the bad files and crashes on some. Sometimes, some applications lose control when handling these files. Buffer overruns or other failures allow the tester to take over the application or the machine running the application. Any program that will read any type of data stream can be subject to this type of attack if the tester can modify the data stream before it reaches the program.
- Kaner (2000) describes several other examples of high-volume automated testing approaches. One classic approach repeatedly feeds random data to the application under test and to another application that serves as a reference for comparison, an oracle. Another approach runs an arbitrarily long random sequence of regression tests, tests that the program has shown it can pass one by one. Memory leaks, stack corruption, wild pointers or other garbage that cumulates over time finally causes failures in these long sequences. Yet another approach attacks the program with long sequences of activity and uses probes (tests built into the program that log warning or failure messages in response to unexpected conditions) to expose problems.

High-volume testing is a diverse grouping. The essence of it is that the *structure* of this type of testing is designed by a person, but the individual test cases are developed, executed, and interpreted by the computer, which flags suspected failures for human review. The almost-complete automation is what makes it possible to run so many tests.

- The individual tests are often weak. They make up for low power with massive numbers.
- Because the tests are not handcrafted, some tests that expose failures may not be particularly credible or motivating. A skilled tester often works with a failure to imagine a broader or more significant range of circumstances under which the failure might arise, and then craft a test to prove it.
- Some high-volume test approaches yield failures that are very hard to troubleshoot. It is easy to see that the failure occurred in a given test, but one of the necessary conditions that led to the failure might have been set up thousands of tests before the one that

actually failed. Building troubleshooting support into these tests is a design challenge that some test groups have tackled more effectively than others.

Exploratory Testing

Exploratory testing is "any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests" (Bach 2003a).

Bach points out that tests span a continuum between purely scripted (the tester does precisely what the script specifies and nothing else) to purely exploratory (none of the tester's activities are pre-specified and the tester is not required to generate any test documentation beyond bug reports). Any given testing effort falls somewhere on this continuum. Even predominantly pre-scripted testing can be exploratory when performed by a skilled tester.

> "In the prototypic case (what Bach calls "freestyle exploratory testing"), exploratory testers continually learn about the software they're testing, the market for the product, the various ways in which the product could fail, the weaknesses of the product (including where problems have been found in the application historically and which developers tend to make which kinds of errors), and the best ways to test the software. At the same time that they're doing all this learning, exploratory testers also test the software, report the problems they find, advocate for the problems they found to be fixed, and develop new tests based on the information they've obtained so far in their learning." (Tinkham & Kaner, 2003)

An exploratory tester might use any type of test--domain, specification-based, stress, risk-based, any of them. The underlying issue is not what style of testing is best but what is most likely to reveal the information the tester is looking for at the moment.

Exploratory testing is not purely spontaneous. The tester might do extensive research, such as studying competitive products, failure histories of this and analogous products, interviewing programmers and users, reading specifications, and working with the product.

What distinguishes skilled exploratory testing from other approaches and from unskilled exploration, is that in the moments of doing the testing, the person who is doing exploratory testing well is fully engaged in the work, learning and planning as well as running the tests. Test cases are good to the extent that they advance the tester's knowledge in the direction of his information-seeking goal. Exploratory testing is highly goal-driven, but the goal may change quickly as the tester gains new knowledge.

Concluding Notes

There's no simple formula or prescription for generating "good" test cases. The space of interesting tests is too complex for this.

There are tests that are *good for your purposes*, for bringing forth the type of information that you're seeking.

Many test groups, most of the ones that I've seen, stick with a few types of tests. They are primarily scenario testers or primarily domain testers, etc. As they get very good at their preferred style(s) of testing, their tests become, in some ways, excellent. Unfortunately, no style yields tests that are excellent in all of the ways we wish for tests. To achieve the broad range of value from our tests, we have to use a broad range of techniques.

Acknowledgements

This research was partially support by grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers."

Thanks to Andy Tinkham, Al Jorgenson, and Pat McGee for comments on previous drafts.

References

Al-Ghafees, Mohammed; & Whittaker, James (2002) “Markov Chain-based Test Data Adequacy Criteria: A Complete Family”, *Informing Science & IT Education Conference*, Cork, Ireland, http://ecommerce.lebow.drexel.edu/eli/2002Proceedings/papers/AlGha180Marko.pdf (accessed March 30, 2003)

James Bach (2003a), “Exploratory Testing Explained”, www.satisfice.com/articles/et-article.pdf (accessed March 30, 2003).

Bach, James (2003b) Rapid Software Testing (course notes). www.testingeducation.org/coursenotes/bach_james/cm_200204_rapidtesting/ (accessed March 20, 2003).

Berger, Bernie (2001) "The dangers of use cases employed as test cases," *STAR West* conference, San Jose, CA. www.testassured.com/docs/Dangers.htm. accessed March 30, 2003.

Bittner, Kurt & Ian Spence (2003) *Use Case Modeling*, Addison-Wesley

Buwalda, Hans (2000a) "The three holy grails of test development," presented at *EuroSTAR* conference.

Buwalda, Hans (2000b) "Soap Opera Testing," presented at *International Software Quality Week Europe* conference, Brussels.

Cockburn, Alistair (2000) *Writing Effective Use Cases*, Addison-Wesley.

Collard, R. (1999). "Developing test cases from use cases," *Software Testing & Quality Engineering*, July/August, p. 31.

Constantine, Larry, L & Lucy A.D. Lockwood (1999) *Software for Use,* ACM Press.

El-Far, Ibrahim K. (1995) *Automated Construction of Software Behavior Models*, M.Sc. Thesis, Florida Tech, www.geocities.com/model_based_testing/elfar_thesis.pdf (accessed March 28, 2003).

El-Far, Ibrahim K. (2001) “Enjoying the Perks of Model-Based Testing”, *STAR West* conference, San Jose, CA. (Available at www.geocities.com/model_based_testing/perks_paper.pdf, accessed March 25, 2003).

El-Far, Ibrahim, K; Thompson, Herbert; & Mottay, Florence (2001) “Experiences in Testing Pocket PC Applications,” *International Software Quality Week Europe*, www.geocities.com/model_based_testing/pocketpc_paper.pdf (accessed March 30, 2003).

Hendrickson, Elisabeth (2002) Bug Hunting (course, notes unpublished)

Graham, Dorothy; & Fewster, Mark (1999) *Software Test Automation: Effective Use of Test Execution* Tools, ACM Press / Addison-Wesley.

Houghtaling, Mike (2001) Presentation at the *Workshop on Model-Based Testing*, Melbourne, Florida, February 2001.

Institute of Electrical & Electronics Engineers, Standard 610 (1990), reprinted in *IEEE Standards Collection: Software Engineering 1994 Edition.*

Jacobson, Ivar, Grady Booch & James Rumbaugh (1999) *The Unified Software Development Process*, Addison-Wesley.

Jorgensen, Alan (2003) "Testing With Hostile Data Streams," http://streamer.it.fit.edu:8081/ramgen/cs/Spring03/CSE5500-Sp03-05/trainer.smi

Kaner, Cem (1998), "Avoiding Shelfware: A Manager's View of Automated GUI Testing," Keynote address, *STAR '98 Conference*, Orlando, FL., www.kaner.com/pdfs/shelfwar.pdf (accessed March 28, 2003).

Kaner, Cem (2000), "Architectures of Test Automation," *STAR West* conference, San Jose, CA, www.kaner.com/testarch.html (accessed March 30, 2003).

Kaner, Cem (2002) *A Course in Black Box Software Testing (Professional Version),* www.testingeducation.org/coursenotes/kaner_cem/cm_200204_blackboxtesting/ (accessed March 20, 2003)

Kaner, Cem; Bach, James; & Pettichord, Bret (2002) *Lessons Learned in Software Testing*, Wiley.

Kaner, Cem; Falk, Jack; & Nguyen, Hung Quoc (1993) *Testing Computer Software,* Wiley.

Marick, Brian (1997) "Classic Testing Mistakes", *STAR East* conference, www.testing.com/writings/classic/mistakes.html (accessed March 17, 2003).

Marick, Brian (undated), documentation for the Multi test tool, www.testing.com/tools/multi/README.html (accessed March 17, 2003).

Musa, John (1998) *Software Reliability Engineering*, McGraw-Hill.

Nyman, N. (1998), "Application Testing with Dumb Monkeys", *STAR West* conference, San Jose, CA.

Nyman, N. (2002), "In Defense of Monkey Testing", www.softtest.org/sigs/material/nnyman2.htm (accessed March 30, 2003).

Patton, Ron (2001) *Software Testing*, SAMS.

Pettichord, Bret (2002) "Design for Testability" *Pacific Northwest Software Quality Conference*, October 2002, www.io.com/~wazmo/papers/design_for_testability_PNSQC.pdf (accessed March 28, 2003).

Popper, Karl (1992) *Conjectures and Refutations: The Growth of Scientific Knowledge.* 5th Edittion. Routledge.

Robinson, Harry (2001) Presentation at the *Workshop on Model-Based Testing*, Melbourne, Florida, February 2001; For more on Robinson's experiences with state-model based testing, see www.geocities.com/model_based_testing/ (accessed March 20, 2003.)

Savoia, Alberto (2000) The Art and Science of Load Testing Internet Applications, *STAR West* conference, available at www.stickyminds.com.

Tinkham, Andy; & Kaner, Cem (2003) "Exploring Exploratory Testing", STAR East conference, www.testingeducation.org/articles/exploring_exploratory_testing_star_east_2003_paper.pdf (accessed March 30, 2003).

Wiegers, Karl E. (1999) *Software Requirements*, Microsoft Press.

Whittaker, James (1997) "Stochastic Software Testing," *Annals of Software Engineering* Vol. 4, 115-131.

Whittaker, James (2002) *Why Software Fails*, Addison-Wesley.