# A Manager's Guide to Evaluating Test Suites

Brian Marick, Testing Foundations
James Bach, Satisfice Inc.
Cem Kaner, kaner.com

*Whatever else testers do, they run tests. They execute programs and judge whether the results produced are correct or not. Collections of tests –* test suites *– are expensive to create. It's fair to ask how good any given suite is. This paper describes ways we would go about answering that question.*

### Our audience

There are two broad traditions of testing. One holds that the purpose of testing is to justify confidence in the program. Reliability modeling [Lyu96] is in that tradition. The other tradition focuses on producing valid bug reports. We will not argue here which tradition is correct. We merely state that we belong to the second tradition. This paper is addressed to people like us.

## *Evaluating a test suite one time*

In this section, we'll assume that you have been presented with a test suite, the tester who created it, the bugs it found, and all relevant documentation. We'll present steps you could use to evaluate the suite. When detailed discussions of topics are readily available online, we only sketch them here.

### Step 1: What is the suite's purpose?

Suppose you have two test suites, Suite1 and Suite2. Is Suite1 better if it finds more bugs? Not necessarily. Let's explore the issue by way of an example.

Suppose the product has five subsystems. For whatever reason, the testing team devotes almost all its effort to subsystem 1 and files 100 bug reports. The development team works diligently and fixes them all. The testing team can't find any bugs in the fixed subsystem, so the product is released. After release, we discover that the testing team did a good job on subsystem 1: no bugs are reported in it. 12 bugs are reported in each of the undertested subsystems. Here's a table representing this scenario:

---

Contact information: marick@testing.com, www.testing.com; james@satisfice.com, www.satisfice.com; kaner@kaner.com, www.kaner.com

Copyright © 2000 Brian Marick, James Bach, and Cem Kaner. All Rights Reserved. Permission granted to copy for personal use.

Some images copyright www.arttoday.com.

| | *Bug reports before release* | *Bug reports after release* | |
|---|---|---|---|
| Subsystem 1 | 100 | 0 | |
| Subsystem 2 | 0 | 12 | |
| Subsystem 3 | 0 | 12 | |
| Subsystem 4 | 0 | 12 | |
| Subsystem 5 | 0 | 12 | |
| | 100 | 48 | 148 |

Here's another possible scenario. The testing team spreads its work more evenly among the subsystems. It finds only 50 of the bugs we know are in subsystem 1, but it also finds half of the bugs in the other subsystems. After release, all unfound bugs are reported by customers. We have the following situation.

| | *Bug reports before release* | *Bug reports after release* | |
|---|---|---|---|
| Subsystem 1 | 50 | 50 | |
| Subsystem 2 | 6 | 6 | |
| Subsystem 3 | 6 | 6 | |
| Subsystem 4 | 6 | 6 | |
| Subsystem 5 | 6 | 6 | |
| | 74 | 74 | 148 |

In which scenario did the testing team do the better job? In the first one, they find more bugs. But there's something disconcerting about the notion that they did a better job by mostly ignoring most of the system.

We can illustrate why they in fact did a worse job by considering what the test manager can tell the project manager *before release* in each of the two scenarios.

In the first scenario, the test manager can say, "We tested subsystem 1 into submission. We found a lot of bugs, and they got fixed. We've tried really hard to break it again, and we think it's now solid. Unfortunately, we don't really know anything about the stability of the other subsystems." The team is presenting the project manager with a gamble: just what's going to happen in subsystems 2 through 6 after release? Will there be a few bugs? Or will each subsystem have 100 bugs? Since we know the future of this project, we know

that the gamble to release would pay off – but the manager might not care to make that bet.

In the second scenario, the test manager can say, "We've done a pretty thorough pass over all the subsystems. We've found only a modest number of bugs in subsystems 2 through 5, so we predict few will be found there after release. But subsystem 1 is very shaky. We've found a lot of bugs, and we expect that there are still more to find. *What do you want to do?*" The team is presenting the project manager with information, not a gamble. It's unwelcome information – bad things will happen after release – but it's appropriate to the needs of the project manager.

The purpose of testing is to give good information to help the release decision. It's to let project managers know what will happen if they let the product go [Marick98c].[1]

The same applies when a tester helps a developer who needs to know whether it's safe to add new or changed code to the master source base [Marick98a]. Because that case is less common than the previous, we will assume for concreteness that the project manager is the customer of the test suite.

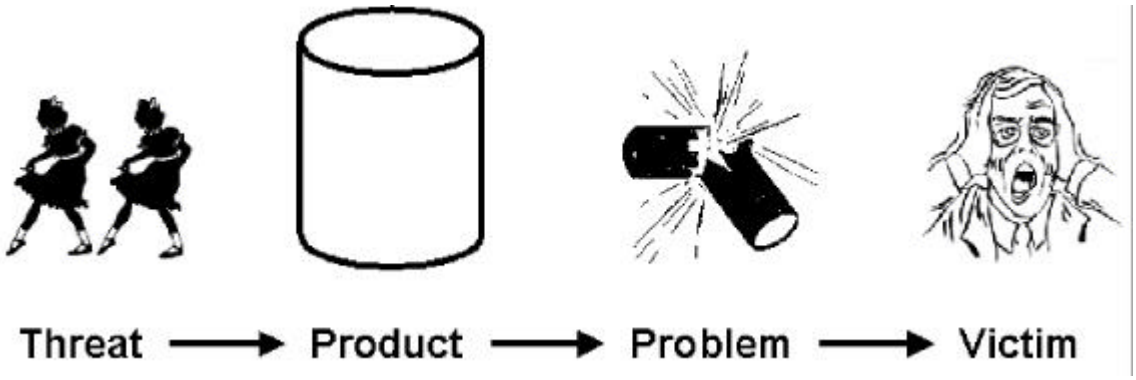
A test suite that provides information that the project manager already knows, and leaves unanswered questions she has, may be technically proficient, but it is not a good one.

If you are evaluating a test suite, you must therefore first evaluate the test suite creator's specific understanding of what the project manager needs to know. This is a matter of risk analysis. In this paper, we will analyze risk according to four factors: Product, Threat, Problem, and Victim. See also [Bach99a].

[1] Kaner is less sure of this assertion. Certainly, *sometimes* the purpose of testing is to help the project manager make a tough decision. But sometimes the project manager isn't looking to the testing group for the primary ship/no-ship release data. Instead, the company might just be hoping to find and fix the maximum number of defects in a fixed period of time. In that case, if the first group did enough testing of the five subsystems to realize that bugs were easy to find in subsystem 1 and hard to find in the others, then it followed the right strategy in focusing on subsystem 1. Alternatively, the company might have decided to ship the product at a certain date and be unwilling to change the decision *unless* the test group buries the company in open bug reports. In this case too, as long as the first subsystem is the richest source of bugs, the first group is acting appropriately.

Students of Kaner's have sometimes attacked this example as artificial. It is constructed to be simple in its presentation, and in that sense it is artificial. But every test group runs into the tradeoff that this example illustrates. Toward the end of the project, the test group is often under fundamentally conflicting pressures. They can operate in "assessment mode" and (like the second group) test broadly but find fewer bugs or they can operate in "bug-hunter mode" and (like the first group) maximize their bug finding rate at the expense of their assessment. Either role might be appropriate.

The following picture illustrates this style of risk analysis. What does it mean to say that a product has a bug? First, the product must be presented with some *Threat* – something in the input or environment that it cannot handle correctly. In this case, the threat is two cute but identical twins. (This corresponds to the common case of software being unable to handle duplicate inputs.) Next, the *Product* must perform its processing on that input or in the context of that environment. If it cannot handle the threat, it produces a *Problem* in the form of a bad result. But a problem means little or nothing unless there is some *Victim* who is damaged by it.




All four of these factors have to be considered in order to direct testing toward those bugs that are most likely to make it unwise to ship the product. As evaluators, we would ask questions probing whether the tester had considered them well.

For concreteness, let's suppose the tester was assigned to "black box" testing of the Tables feature in the Open Source word processor AbiWord (http://www.abiword.com). What might this tester's answers be?

**Victim** – who are the important classes of user?

The AbiWord developers consistently refer to their preferred user as "the church secretary" – someone less than a power user who has a strong preference for basic features that "just work". Therefore, a Tables test suite should concentrate more heavily on basic features at the expense of obscure ones. Everything on the Tables toolbar had better work in lots of circumstances. The subfeatures available through three levels of menu can fail much more often.

The AbiWord team's preferred user reminds us of the "target customer" in the marketing book *Crossing the Chasm* [Moore91] (chapter 4). There, Moore recommends creating 8 to 10 descriptions of users, giving them names, job descriptions, goals, and values. His reason?

> *Neither the names nor the descriptions of markets evoke any memorable images – they do not elicit the cooperation of one's intuitive faculties. We need to work with something that gives more clues about how to proceed. We need something that feels a lot more like real people. However, since we do not have real live customers as yet, we are just going to have to make them up. Then, once we have their images in mind, we can let them guide us to developing a truly responsive approach to their needs. (pp. 94-95)*

The same effect – engagement of the imagination – applies to testers as well as to marketers. We recommend Moore's approach.

**Problem** – what types of failure matter most?

Crashes would be bad for a church secretary, especially if they left documents unreadable by AbiWord. A failure to import Word tables is likely to be harmful – church secretaries exchange documents by email, so our AbiWord user will be often be using it to read Word attachments. When secretaries exchange documents, they probably cut and paste pieces from one document to another, so integration of tables with cut-and-paste code should work well.

Church secretaries are likely to be working with old equipment, so any performance problems will be amplified.

Has the tester looked at many church documents? Is there commonality across different types of churches, synagogues, mosques, and temples? If so, failures to handle commonalities – such as a particular preferred typeface – would be important problems to this class of victims.

Undo should work well with tables, because a non-expert user is likely to become confused if the actions that Undo undoes don't correspond well to the actions she thought she performed (as is likely if Tables are implemented as an afterthought, using underlying paragraph mechanisms[2]).

Finally, the testers should pay attention to the usability of the software. There is no usability team on this project, and the developers are power users, entirely unrepresentative of the canonical users. In order to properly prioritize their work, testers must develop an understanding of how the program is used. They should apply that knowledge to an evaluation of the program's design.

As these examples suggest, it can be hard to be sure you've found the important classes of ways in which the product might fail. Catalogs of failures, such as Appendix A in [Kaner93], can help.

[2] This is not the case with AbiWord, which doesn't even have tables as we write.

**Threat** – what kinds of failure-producing situations is the product likely to encounter?

One threat that an e-commerce application **must** contend with is activity from multiple simultaneous users. But testers of AbiWord needn't worry about that. However, AbiWord is highly cross-platform. It runs on Windows, BeOS, and Linux – indeed, this is one of its claims to fame. So a good test suite had better handle configuration issues well.

**Product** – what areas in the product are the shakiest?

Every product has certain things that are hard to get – and keep – right. As of late 1999, it seemed that printing might be one of those things for AbiWord (though it was really too early to say). More generally, it seemed difficult to get all layouts of the document looking right in all circumstances. In contrast, early users didn't seem to complain about undo bugs.

**Step 2: What is the suite's approach?**

A fine risk analysis doesn't assure a good test suite. Is the test suite's approach appropriate to the risks?

As an evaluator, we'd look for a written or oral description that went something like this:

> "Because the church secretary is so important, it seems we should favor a use-case (task-based) style of tests over strict functional tests.[3]
>
> "Further, the risk analysis affects our automation strategy. We've identified that usability is important. Moreover, some of the most persistent (but perhaps lower-impact) bugs are printer-layout or screen-layout bugs. Those argue for less automation than we might otherwise use. We need real humans executing real church secretary tasks and watching for usability problems. They should also watch for the kinds of layout problems that are notoriously difficult to detect with automation tools.
>
> "However, we have the problem of configuration tests, which are hard to do manually. It's bad enough to run a configuration test once, much less over and over again on different platforms. We have identified a subset of the test suite that can be automated and will be particularly sensitive to configuration dependencies. We believe it's sensitive because..."

The actual approach should be compared to a checklist of common practices, such as the following:

- "Stroke tests" are tests in which a tester checks every claim of fact and example in the manual.

[3] By use-case tests, we mean tests that behave like a user. For example, one such test might build a table, add one shading feature to it, print the page, use undo to remove the shading, add text to the table, print the page again, etc. In contrast, strict functional tests concentrate on exploring the possibilities of one feature (like shading) at the expense of creating the kinds of tables that real users create, in the way that real users create them.

- "Staked application testing" [Kaner/Marick00] is an attempt to use the application under test to duplicate the results of a competing application. To do this for AbiWord, the tester should find some actual church secretaries, get some real church bulletins built with Word, and attempt to recreate them exactly with AbiWord. (This is a test of more than tables.)

At the moment, we know of no thorough checklist. One is slowly being built at www.testingcraft.com, partially based on Appendix A from *Testing Computer Software* [Kaner93]. It would be a good idea to build an explicit checklist for your company.

In evaluating any approach, it's important to look at two things:

- are important things being done?
- are unimportant things **not** being done?

You need to check that the planned tasks are proportionate to both the risks and the project resources [Kaner96a].

### Step 3: How's the maintainability?

For those tests that are automated, do they avoid the automation trap: tests that are intended to be rerun, but are so sensitive to product change that the testers will spend all their time trying to keep old tests working, none of their time creating new tests? This is a difficult matter of test suite design, and it's easy to go astray [Kaner97]. (See also the papers surveyed in [Marick99a].)

Don't just ask whether the suite was designed to be maintainable. Ask what's actually happened as the product evolved. How much time is spent upgrading the suite?

Do manual tests use elaborate scripts? These cost a fortune to develop, and they are expensive to maintain. After their initial use, they are unlikely to find additional defects. If scripts are used, they should straddle the boundary between the repeatability of scripting and the maintainability and benevolent side effects of less detailed test documentation [Marick97].

Has the tester chosen a good balance between manual and automated tests? [Marick98b]

### Step 4: What do the details tell you?

Spend a little time looking at the details of the tests. Can you find gaps in the overall approach, in the tester's understanding of the project, or in her raw testing skills? There are two areas to look at:

**Threats.** How well do the specific tests implement the test approach? What test inputs were chosen? Do they cover all things that could affect the processing the program does: user inputs, machine configurations, contents of databases, network timeouts, and the like? Does the tester seem weak on particular test design techniques?

**Result checking.** A threat that provokes a problem does no one any good if the problem is overlooked. Here are some examples of things to check:

- Suppose the tester is doing purely exploratory testing ([Bach99c], [Pettichord99]). Does she only look for obvious problems ("the program crashed"), or does she notice small oddities like oddly slow operations? Does she then attempt to escalate oddities into major failures?
- Suppose the product provides a GUI front end to a bank. Does a test that removes €100 from an account with €100 in it only check that the ending balance on the screen is €0? Or does it also check whether the backend database has been updated? More generally, does the test check whether everything that should be affected has been?
- Suppose a test uninstalls a product. Does it check that the action leaves alone those things that should be unaffected? For example, does it check that components shared with other products are not removed? Does it check that all user data is left on disk?
- As noted above, automated tests need to be insulated from irrelevant changes to the product's interface. This is done by selectively filtering out information. Sometimes that leads to an inadvertent side effect: the tests become insulated from bugs as well. [Bach96] notes an extreme example: 3000 tests that always returned success, no matter what the product under test did! When reviewing automated tests, ask what kinds of bugs they might miss.

**Step 5: What does the suite not do?**

We'll first discuss code coverage. Code coverage tools add instrumentation to code. When the test suite executes, the instrumentation records how it exercises the program.

Our experience with such tools has been mixed. Some coverage tools are of poor quality. The most common problems in the weaker tools are that they are difficult to integrate into the build process, they fail to instrument some valid code that the compiler accepts (breaking the build), and they slow down execution to a ridiculous extent. Less common – but worse – is instrumentation that introduces a bug into the program. Their management of data is often crude. They typically assume a fully-automated test suite that is rerun against every build of the product, so are not good at merging data from several builds. The better tools are less prone to these problems.

All that having been said, we believe that code coverage can be a useful, but usually minor, part of test suite evaluation. We would use code coverage in two ways. The first is a quick look for major omissions; the other is a detailed look at critical code.

First, let's describe the major types of coverage that are measured by these tools.

Line coverage
: This type measures which code statements were executed.

Branch coverage
: Branch coverage measures which branches in the code were executed in which directions. It's possible to execute all statements without executing all branches, so branch coverage is stricter than line coverage.

Multicondition coverage

This type measures whether each Boolean condition in a branch has been evaluated to both true and false. It's a stricter criterion than branch coverage.

Multicondition is the best of these types, but line coverage is sufficient for our first purpose.

We would have a coverage tool summarize coverage at a fairly high level of granularity. Since AbiWord is C++ code, we would probably use the class level, which roughly corresponds to file-by-file summaries. We would look at those classes which had low or no coverage, and we would ask "why not?". Was it an intentional omission in the test (probably due to lack of time), or a mistake?

Next, if the importance of the code under test warranted, we would look at a sample of missed lines, branches, or conditions. (We would also like to use coverage that checks whether a loop has been executed zero, one, and many times, and whether boundaries have been checked. But the only coverage tool that we know measures those [Marick92] doesn't work on C++.)

Detailed code coverage is valuable when omitted coverage can be related back to the test design process to yield useful insights [Marick99b]. For example, any branch that has never been taken corresponds to some expected condition in the product's input or environment that has never occurred. Why not? Is it a natural consequence of the test approach? (Perhaps it corresponds to a test that couldn't be run because of lack of time.) Is it a mistake in test implementation? Is it a test design oversight? Is it symptomatic of pervasive oversights – for example, has error handling been systematically undertested?

Note that code coverage is less useful for some kinds of testing than others. For example, knowing that a line of code has been executed does not tell you whether it's been executed in a realistic user task. So coverage sheds less light on use-case style testing than on functional testing.

Code coverage is not the only type of coverage. Any systematic and tractable method of enumerating the ways a test suite exercises the code can be used in the same way. For example, Marick learned a style of documentation coverage from Keith Stobie. You begin with a highlighter and user documentation. You highlight every sentence in the documentation that has a test. At the end, there should be a justification for each un-highlighted sentence. ("Other things were more important" can be a good justification.)

[Kaner96b] gives a list of many other types of coverage.

### Step 6: What has the suite missed?

Any test suite will miss some bugs. They might be discovered by other project members, by beta testers, or (after release) by end users. We think it is a good practice to examine those bug reports and ask questions. What do they tell you about how well the test approach has been implemented? Do they suggest flaws in the approach?

It's best to look at bugs that have been fixed. You can understand not only the observed failure, but also something of the underlying cause.

In a conversation on this topic on the SWTEST-DISCUSS mailing list (swtest-discuss-request@rstcorp.com), Jon Hagar warned that you should be careful not to overreact. Some bugs just get missed. Frantically changing plans in response to every serious missed bugs will lead to unproductive thrashing. We echo his comments.

**Summary**

This evaluation approach is highly subjective. In particular, it depends on the skill and experience of the evaluator. But an evaluator who is less good than the creator of the test suite can still produce value **if** she has good skills at human communication and cooperation, and good overall judgment. The worst thing is an evaluator who - like many drawn to evaluation - combines a lack of skill and experience with a dogmatic lack of knowledge of her lack of knowledge.

Although subjective, we think this method is preferable to an overly narrow objective one. Like it or not, objective rules are easy to misuse except in extremely well understood situations, such as counter work at a fast food restaurant. Ironically, we would only trust objective testing criteria in the hands of someone we had subjectively decided knows better than we do.

This approach is also expensive, if done in full. But it needn't be. The effort spent in evaluation should be tailored to the risk posed by a weak test suite. If the risk is low, a quick spot-check would suffice: an hour or so of interviewing about the test suite's purpose and approach, a brief check of maintainability, a small sample of bug reports and coverage.

## *What's missing?*

This paper is incomplete in an important way. It assumes that evaluation is a "point activity". You as evaluator swoop in, take a snapshot of the knowledge of the tester and test suite, make your judgment, then ride off into the sunset.

Real projects don't work that way. At least, they shouldn't. A project begins with radically incomplete and incorrect answers to all of the questions we asked in this paper. Over time, the answers get better. For example, a project's idea of the canonical user will change.

If you evaluate once, when should that be? At the end of the project, when you can make the most accurate judgment? – but when it's too late to affect anything? Or early, when you can make a difference, but perhaps a difference based on bad data?

The answer, of course, is that you shouldn't evaluate once. The evaluation process should be continuous throughout the project. These questions should be constantly asked, to constantly refine your understanding of what a good (appropriate) test suite for this project would be.

Moreover, any project is in effect a conversation among product members (and, indeed, among all stakeholders[4]). By asking these questions, and others, you not only learn the answers, you also change them. That's an important service. For more, see [Bach97] and [Bach99b].

## *Appendix A: Approaches we reject*

Error seeding and mutation testing are sometimes advocated as ways of evaluating a test suite. Here we explain why we reject them.

### Background

A bug report describes what the tester believes to be a *failure* in the program. A failure is a difference between what a program did in some execution and what it should have done. The question of whether a particular result is really a failure is sometimes cause for lively debate.

A failure is caused by an underlying *fault*. In a program, a fault can be defined as the code that's changed to resolve the bug report. A single fault may cause multiple failures, and it may not be possible to tell that two failures are caused by the same fault until the fault is discovered.

Since there may be several reasonable fixes for a bug, a single failure might be said to be caused by multiple faults. What the fault *really* is can also be cause for great debate.

Specifications and requirements documents may also contain faults. If so, they describe or require results that are wrong from the perspective of a reasonable user. It is quite possible for the code to be right and the requirements document to be wrong. The fault in such an "upstream" document can be defined as the text that's changed to resolve a bug report. In the common case where the documents don't exist or are not updated, the fault is whatever you'd write down if you wrote down those kinds of things.

### Error seeding

Let's define a *perfectly effective test suite* as one that reveals at least one failure for every fault in the program.

Given the background, this definition has some immediate problems:

1. What constitutes a failure is debatable.
2. The set of faults is debatable.
3. Some faults are not in the program.

[4] See *The Cluetrain Manifesto* [Locke00]: "A powerful global conversation has begun. Through the Internet, people are discovering and inventing new ways to share relevant knowledge with blinding speed. As a direct result, markets are getting smarter—and getting smarter faster than most companies… These markets are conversations…"

However, these are secondary issues. The fundamental question is how you would measure how close a test suite comes to perfection. You cannot simply say:

*Effectiveness = number of faults found by suite / number of faults present*

You don't know – and cannot know – the number of faults present in the code.

You could estimate *number of faults present* by adding the number of faults found in, say, the first six months after release to *number of faults found by suite*. This estimate of effectiveness comes a bit late. We'd like an estimate that would let us do something useful before release.

One alternative estimate is *error seeding* [Mills70]. In our terminology, it would be called "fault seeding". The process is this:

1. Seed, or inject, *N* representative faults into the program. For example, one seeded fault might be the removal of code that checks whether the network has gone down.
2. Test. You'll find some fraction, *some-of-N*, of the *N* seeded faults.
3. You probably found roughly the same proportion of the unseeded "native" faults. The estimated effectiveness is *(some-of-N / N)*.

The problem is that it is surprisingly difficult to inject representative faults. We lack a useable taxonomy of the different kinds of faults, knowledge of their relative chances of occurring, and an understanding of how the placement of faults in a program affects their visibility as failures.[5] If we seed the wrong proportions of the wrong kinds of faults in the wrong places in the program, our estimate may be wildly off.

We don't deny that fault seeding can provide some insight into a test suite. But does it provide more than coverage does? Statement coverage tells you that any fault seeded into an uncovered line will not be caught. Missed multicondition coverage tells you that a test suite could not have caught certain Boolean operator faults. Generally, each type of coverage can be viewed as targeting a particular class of seeded fault [Marick95] without actually doing the seeding. To accept fault seeding, we require experimental evidence that seeding additional classes of faults provides additional value.

### Mutation testing

A variant of fault seeding is called *mutation testing* [DeMillo78]. It attempts to avoid the problems just cited through brute force and one key assumption. Mutation testing makes use of *one token faults*. A one token fault is, for example, changing a "+" to a "-", a "<" to a "<=", or the variable "i" to the variable "j". These faults are injected everywhere they can be, each one in a new copy of the program. This avoids the issue of deciding where to place faults and whether bad placement affects visibility.

[5] See, however, Voas's testability models for work on understanding the placement of faults. [Voas92] is an introduction. See also papers at the Reliable Software Technologies web site (www.rstcorp.com).

It's still the case that not all faults in programs are one-token faults. For example, the fault of omitting a network error check is a multiple-token fault. Mutation testing's *coupling assumption* asserts that a test suite adequate to find all one-token faults is also adequate to find all (or almost all) faults of all kinds. That given, it would perhaps be reasonable to say that a test suite adequate to finding 50% of one-token faults will probably also find 50% of all faults.

Mutation testing has problems that we believe makes it unsuitable for commercial use.

The coupling assumption is one that must be tested by experiment. To our knowledge, that has not been done, except for the simple case of pairs of one token faults [Offutt92]. We suspect that the coupling assumption is not true. A key problem is *faults of omission*, which are faults fixed by adding code. For example, a program that fails to check for network outages is fixed by adding code to do the check. Mutation testing – which concentrates on checking for variant implementations of code that's present – would seem to have little leverage for code that's not present. Since faults of omission represent a large fraction of faults that escape test suites (see [Marick00] for some numbers), that lack of leverage is worrisome.

Still, our approach in this paper was to recommend a collection of approaches, each making up for some of the weaknesses of the rest. Why shouldn't mutation testing be included in that collection?

Mutation testing requires tools to inject the faults. As far as we know, no commercially relevant mutation testing tools are available. One of us wrote a tool that provides a variant of mutation testing for C programs [Marick92], but it is available only for C programs and works only on old versions of Unix. Moreover, his experience with that tool [Marick91] left him unconvinced that mutation testing provides value over code coverage. Examining undiscovered mutation faults did not lead him to discover omissions in his test suite. Experiments or case studies applying mutation testing to large-scale software over an extended period of time might change his mind.

## Acknowledgements

Boris Beizer prompted this paper by asking this question in an exchange with Marick on the SWTEST-DISCUSS mailing list (swtest-discuss-request@rstcorp.com):

> ASSERT: A good tester is one that produces a good test suite -- a suite that has a high probability of finding bugs if there are any to find. [...]
>
> QUESTION: What objective method do you propose to use to measure the effectiveness of a test suite?

As you've seen, we have not provided an objective method (that is, one that would yield the same results no matter who applied it). We do not believe that a useful objective method is possible. Still, it was a good and fair question.

## *References*

[Bach96]
James Bach, "Test Automation Snake Oil,", *Windows Tech Journal*, October 1996. http://www.satisfice.com/articles/test_automation_snake_oil.pdf.

[Bach97]
James Bach, "Good Enough Quality: Beyond the Buzzword" (Software Realities column), *IEEE Computer*, August 1997. http://www.satisfice.com/articles/good_enough_quality.pdf

[Bach99a]
James Bach, "Risk-based Testing", *Software Testing and Quality Engineering Magazine*, Vol. 1, No. 6, November/December 1999. http://www.stqemagazine.com/featured.asp?stamp=1129125440

[Bach99b]
James Bach, " What Software Reality is Really About" (Software Realities column), *IEEE Computer*, December 1999. http://www.satisfice.com/articles/software_reality.pdf

[Bach99c]
James Bach, "General Functionality and Stability Test Procedure for *Certified for Microsoft Windows* Logo". (A description of an exploratory testing procedure.) http://www.satisfice.com/tools/procedure.pdf

[DeMillo78]
R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer*, Vol 11, no. 4, April 1978.

[Kaner93]
Cem Kaner, Jack Falk, and Hung Quoc Nguyen, *Testing Computer Software*, 2nd edition, 1993.

[Kaner96a]
Cem Kaner, "Negotiating Testing Resources: A Collaborative Approach," a position paper for the panel session on "How to Save Time and Money in Testing", in *Proceedings of the Ninth International Quality Week* (Software Research, San Francisco, CA), 1996. (http://www.kaner.com/negotiate.htm)

[Kaner96b]
Cem Kaner, "Software Negligence and Testing Coverage," *STAR 96 Proceedings*, May 1996. (http://www.kaner.com/coverage.htm)

[Kaner97]
Cem Kaner, "Improving the Maintainability of Automated Test Suites," in *Proceedings of the Tenth International Quality Week*, May 1997. http://www.kaner.com/lawst1.htm

[Kaner/Marick00]
Brian Marick wrote a description of Kaner's Staked Application Testing at this URL: http://www.testingcraft.com/staked-application-testing.html.

[Locke00]
Christopher Locke, Rick Levine, Doc Searls, and David Weinberger, *The*

*Cluetrain Manifesto: The End of Business as Usual*, Perseus, 2000. Much material at http://www.cluetrain.com.

[Lyu96]
Michael Lyu, *The Handbook of Software Reliability Engineering*, IEEE Computer Press, 1996.

[Marick91]
Brian Marick, "Experience with the Cost of Different Coverage Goals for Testing", *Pacific Northwest Software Quality Conference*, October 1991. http://www.testing.com/writings/experience.pdf.

[Marick92]
Brian Marick, The Generic Coverage Tool (GCT). Source and user documentation available at ftp://cs.uiuc.edu/pub/misc/testing/gct.1.4/. See also ftp://cs.uiuc.edu/pub/misc/testing/gct2.0Beta/. Brian Marick **does not support** this software any more.

[Marick95]
Brian Marick, *The Craft of Software Testing*, 1995.

[Marick97]
Brian Marick, "Classic Testing Mistakes", *STAR Conference*, May 1997. http://www.testing.com/writings/classic/mistakes.html.

[Marick98a]
Brian Marick, "Working Effectively With Developers", *STAR West Conference*, October, 1998. http://www.testing.com/writings/effective.pdf.

[Marick98b]
Brian Marick, "When Should a Test be Automated?" *Proceedings of International Quality Week*, May, 1998. http://www.testing.com/writings/automate.pdf.

[Marick98c]
Brian Marick, "The Testing Team's Motto". http://www.testing.com/writings/purpose-of-testing.htm.

[Marick99a]
Brian Marick, "Web Watch: Automating Testing", *Software Testing and Quality Engineering Magazine*, Vol. 1, Num. 5, Sep/Oct 1999. http://www.stqemagazine.com/webinfo_detail.asp?id=102

[Marick99b]
Brian Marick, "How to Misuse Code Coverage", *International Conference and Exposition on Testing Computer Software*, June 1999. http://www.testing.com/writings/coverage.pdf.

[Mills70]
Harlan Mills, "On the Statistical Validation of Computer Programs". Reprinted in Harlan Mills, *Software Productivity*, 1983.

[Moore91]
Geoffrey A. Moore, *Crossing the Chasm*, 1991.

[Offutt92]
A. Jefferson Offutt, "Investigations of the Software Testing Coupling Effect," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, 1992.

[Pettichord99]
Bret Pettichord, " An Exploratory Testing Workshop Report," July 1999. http://www.testingcraft.com/exploratory-pettichord.html

[Voas92]
J. Voas, "PIE: A Dynamic Failure-based Technique," *IEEE Transactions on Software Engineering*, August 1992.