How to Misuse Code Coverage

Brian Marick
Testing Foundations[1]
marick@testing.com
www.testing.com

Code coverage tools measure how thoroughly tests exercise programs. I believe they are misused more often than they're used well. This paper describes common misuses in detail, then argues for a particular cautious approach to the use of coverage.

There are other ways to measure the thoroughness of tests. See [Kaner96] for a list. These ways all fit under the umbrella term “test coverage.” My claims in this paper are specific to code coverage.

To avoid tedious repetition of the phrase “code coverage,” I’ll follow common practice and use the shorthand term “coverage”. I’ll use the full phrase occasionally to remind you of the scope of the paper.

Like code coverage itself, this paper is most useful to programmers testing their own code or to testers who read the source code while testing. It also describes coverage's relevance to the independent product tester (someone who doesn't look at the code) and to managers of developers and testers.

What is code coverage?

There are many different coverage measures. A simple one is to record which lines of code were executed. If a line has never been executed, it's a safe bet you didn't catch any bugs lurking in it. This type of coverage is usually called “statement coverage”.

In this paper, I'll use a somewhat more powerful type. Many commercial tools measure some variant of this coverage; few measure anything more powerful. Different people use different names for this type of coverage: "segment coverage", "multiple condition coverage", "branch coverage" (arguably incorrectly), and so on. I'll use "multiple condition coverage". It measures whether each logical condition in the code has evaluated both true and false (in some execution of the program, not necessarily in the same one). Suppose your code looked like this:

[1] This version of this paper was written while I was an employee of Reliable Software Technologies (www.rstcorp.com). I have been granted permission to reproduce it.

```
if (x < 0)
   do something...

if (y < 3 && z <= x)
   do something else...
```

You would measure its coverage by first *instrumenting* it. The coverage tool would modify the code to record which expressions evaluate which way. That modification is done either to the source code or to the executable the compiler generates. If it's done to source code, most tools don't change the original source; instead, they modify a copy that's then passed to the compiler in a way that makes it look like the original. After instrumentation, you would run several tests, accumulating coverage results in some log. Looking at the log, you might see lines like this:

```
sourcefile.c, line 33: branch was never taken true.
sourcefile.c, line 36: the second expression (z<=x) has never evaluated false.
```

The first line tells you that no test caused the first `if` statement to be executed with x < 0. The second tells you that `z` has never been greater than `x` in the second `if` statement. With a typical coverage tool, the fact that the log says nothing about the first logical expression in the second `if` statement means y<3 has been true in at least one test and false in at least one.

The obvious way to use code coverage tools

I'll now give a short example. I'll pretend I'm testing part of the first program I wrote after graduating from college and show how a coverage tool can be used to improve the tests. That's a typical use of coverage: create your tests, apply the tool, react to what it tells you.

My program was a tool to count the number of lines of code and comment in C programs.[2] Since this was in the command-line era, you executed this program, `lc`, like this:

```
lc lc.c
```

lc produced output that looked something like this:

```
        Pure Code    Pure Comment          ...
lc.c        164             305           ...
```

(There were more columns of output, but they're not relevant to this example.) `lc`, being a good Unix program, could take as its input the output of another program, so you could use it like this:

[2] I think such tools are quite often useless or even counterproductive, but that's a different paper.

```
echo "main(){}" | lc
```

In that case, the output might look like this:

```
Pure Code    Pure Comment          ...
     1               0             ...
```

After the program was in use for a short while, a furor erupted. C uses curly braces to group statements. As was the case in many C programming shops, there was considerable controversy over where to put the curly braces. One group considered it *obvious* that the opening brace should be at the end of the line, as in:

```
if (x < 0) {
}
```

Another group was convinced that the first group was willfully and maliciously oblivious to the manifest superiority of putting the opening brace on the next line:

```
if (x < 0)
{
}
```

Neither side would give an inch, and the company had wisely decided not to have an official policy. When `lc` was put into service, though, the first group noticed that the second now got credit for an extra line of code. (In the above example, the second group has written three lines to the first group's two, a 50% increase in productivity.)

This was unacceptable, so I was instructed to add an option to `lc`. On Unix, command line options begin with "-" and follow the name of the command. `lc` would now accept these two options:

```
lc -{
lc -}
```

Either meant that a curly brace alone on a line would be treated as a blank line. (So both groups would now only get credit for one line, the line containing the `if`.)

My example will be part of the code that processes command-line arguments. It's a loop whose purpose is to handle the options. It looks at each argument. If it's an option, such as -}, it does what's appropriate. If it's not an option, it passes control to the rest of the program, which handles any remaining arguments. Since you may not know C, I'll write it in pseudo-code:[3]

[3] ...which also allows me to hide *my* preferred brace-handling style. Because if I revealed it, I know half of the C programmers reading this would immediately conclude that I was a fool.

```
100:    while there are arguments left AND the next one begins with '-'
101:            if the option is -} OR it's -{
102:                    make a note to treat brackets alone as white space
103:            else
104:                    handle the other options
```

Suppose that, after testing, I look at the coverage log. It contains two lines describing missed coverage; that is, logical expressions my tests failed to exercise completely. Call those *coverage conditions*. The first coverage condition is:

> lc.c, line 101: the first expression (option is -}) has never evaluated true.

What this means, in terms of the user interface, is that I've never tested the "-}" option. It seems like a good idea to test all options that are advertised to users. I might test it immediately, but let's suppose I first look at the rest of the missing coverage. The next (and final) coverage condition is:

> lc.c, line 100: the first expression (there are arguments left) has never evaluated false.

What does this mean? The loop exits if either clause is false, which means:

1. there are no arguments left (be they options or names of source files), or
2. the next argument isn't an option (that is, it's the name of a source file).

Since the first expression has never been false, `lc` has never run out of arguments before it found the name of a source file. In other words, I've never run `lc` when its input came from the output of another program.

This is a more serious revelation. The fact that "-}" is handled correctly would be pretty obvious by inspection (because it does nothing differently than "-{", which coverage tells me has been tested), but a lot more could go wrong when handling non-file input. For example, the program might incorrectly try to open some uninitialized file variable, leading to a crash. So this is certainly worth testing.

Given this information from the tool, what should I do? I should try the missed cases to satisfy the coverage conditions. Fortunately, that's easy to do. I can write a single simple test that handles both. Here it is:

```
echo "{" | lc -}
```

Input from another program contains a line with only a curly brace. Because of the "-}", the expected result is zero lines of code and zero lines of comment. Suppose I ran that case. (I would find no bugs.) The coverage log would now be empty - there would be no missed coverage conditions. I'd be done testing.

What's not obvious

Or would I? I'd be done only if "being done testing" is necessarily the same thing as "exercising all the coverage conditions"; that is, the same thing as "causing all logical expressions to evaluate both true and false".

But it's not. Consider this code:

```
23:     status = perform_operation();  /* do something and assign status code. */
24:     if (status == FATAL_ERROR)
25:             exit(3);                /* exit with error code 3 */
26:     else
27:             all is OK; continue on.
```

Suppose I ran some tests over this code and checked the coverage. Further suppose that the coverage tool told me I'd never taken the `if` on line 24 in the true direction. The natural thing to do would be to set up a test such that the operation fails with a FATAL_ERROR, run that test, and check that the system performs as it should (in this case, that exiting with error code 3 is both required by the specification and also sensible for the user). I've satisfied the coverage tool.

But I'm not done testing, because this code is wrong. The programmer either never knew or forgot that `perform_operation` can actually return three status codes: FATAL_ERROR, ALL_OK, and RECOVERABLE_ERROR. The code above doesn't handle the last case. To put it more precisely: it treats RECOVERABLE_ERROR like ALL_OK, which is surely incorrect.

The code should look like this:

```
23:     status = perform_operation(); /* do something and get status code. */
24:     if (status == FATAL_ERROR)
25:             exit(3);                /* exit with error code 3 */
26:     else if (status == RECOVERABLE_ERROR)
27:             recover from error
28:     else
29:             all is OK; continue on.
```

The test suite should check that the code handles recoverable errors correctly. But no test does that (else the bug would have been discovered). And the coverage tool didn't tell me to write the needed tests. It can't. It can only tell me how the code that exists has been exercised. It can't tell me how code that ought to exist would have been exercised.[4]

[4] A reviewer commented that I might also be measuring the coverage of `perform_operation`. The coverage tool could then tell me that RECOVERABLE_ERROR had never been returned. By writing a test that caused that to happen, I would reveal the missing check for that return status. The problem is that `perform_operation` might be called from sixteen different places. If RECOVERABLE_ERROR were returned to one of them, coverage wouldn't help find a missing check in another. Most subroutines are called from more than one place.

This is a fundamental problem for all code coverage tools, not just ones that measure multicondition coverage. They all have to work on the code they're given.

These types of bugs - those that are fixed by adding new code - are sometimes called *faults of omission*. Unfortunately, they are quite common in real life, as numerous studies have shown. My favorite one is [Glass81], in which he describes faults of omission as "code not complex enough for the problem". Faults of omission aren't just unchecked status returns: they include missing boolean conditions in `if` statements, missing "catch" statements for exceptions, and other, more complex cases.

Using code coverage well

So what to do? It would be a mistake to abandon coverage tools. By forcing you to exercise code you missed, they certainly force you to find bugs you wouldn't otherwise. But is there a better way to use them than I did in the examples above?

Yes. Let me first discuss what may seem an unrealistic case, then cover a more common situation later in the paper.

When I have the luxury of having enough time to thoroughly test those subsystems I've written, using test design techniques that are not directly intended to achieve high code coverage, I expect to achieve better than 90% (I usually get around 95%) feasible[5] coverage from my initial test suite, using types of coverage somewhat stronger than the ones described in this paper. I then spend less than 4% of my total testing and debugging time to reach 100% feasible coverage using the following approach. I consider that time "in the noise" and believe that the bugs - few but occasionally serious - discovered during that testing make the effort of using coverage worthwhile. (I'm discounting the cost of acquiring the tool and learning how to use it. That cost is amortized across many people and projects.)

My approach is based on this key observation:

> *If a part of your test suite is weak in a way that coverage can detect, it's likely also weak in a way coverage <u>can't</u> detect.*

How likely is it that your tests are good enough to catch all faults of omission, but just happen to miss exercising a few coverage conditions?

I do the following with each missed coverage condition:

1. I think of what feature in the interface (user interface or other) that condition corresponds to. ("Feature," here, usually means a small amount of code, not something

[5] "Feasible" coverage takes into account the fact that some logical expressions can never be evaluated true. The most common example is self-testing code of the sort that contains assertions like "if some impossible thing happens then fail, noting the program error." If the impossible thing really is impossible, the `if` can never be taken true. When testing, of course, you want to think hard about how to make "impossible" things happen. But if you fail, it doesn't make sense to count that as missed coverage.

like "this product features the ability to display HTML".) I know that feature has been undertested in at least one way.

2. I rethink how I should have tested that feature. I try not to worry about how to satisfy the coverage condition, or think too much about other missed coverage conditions in the same feature. In practice, you can't completely ignore coverage conditions - that would be like deliberately trying not to think of the word "elephant" - but you should make a concerted effort to think only about what kind of test design mistake they represent, not about how to satisfy them.
3. I run my new tests and recheck coverage. I expect to have satisfied the missed coverage conditions in the feature, even though I didn't specifically target them. If not, I erred again in test design.
4. I repeat for other features. Because the original suite was thorough, I don't have to rethink my entire test design.[6]

While using coverage is not a lot of work, let me say that it's nonetheless ***hard***. I've been working with coverage since 1981. I've written four coverage tools (one with Thomas Hoch and K. Bertram Schaefer). I *still* find myself looking at a coverage condition, saying "I know how to satisfy that," and getting an almost physical urge to write a quick-and-dirty test that would make the coverage tool happy. It's only the certain knowledge that customers don't care if the coverage tool is happy that restrains me.

The most common coverage mistake is giving into that urge. I warn against it by saying that coverage tools don't give commands ("make that evaluate true"), they give clues ("you made some mistakes somewhere around there"). If you treat their clues as commands, you'll end up in the fable of the Sorcerer's Apprentice: causing a disaster because your tools do something very precisely, very enthusiastically, and with inhuman efficiency - but that something is only what you *thought* you wanted.

Designing your initial test suite to achieve 100% coverage is an even worse idea. It's a sure way to create a test suite weak at finding those all-important faults of omission. Further, the coverage tool can no longer tell you where your test suite is weak - because it's uniformly weak in precisely the way that coverage can't directly detect.

Don't use code coverage in test design. The return on your testing dollar (in terms of bugs found) is too low.

Managers make mistakes, too

I expect a high level of coverage. Sometimes managers require one. There's a subtle difference.

[6] I left one step out of this algorithm. Sometimes - not often - I look at a coverage condition and say, "Wait a minute! I *know* I've got a test for that." When I look at that test (if automated) or retry it (if manual), I discover I made a mistake - I didn't implement the test I intended.

Suppose a manager requires some level of coverage, perhaps 85%, as a "shipping gate". The product is not done - and you can't ship - until you have 85% coverage.[7]

The problem with this approach is that people optimize their performance according to how they're measured. You can get 85% coverage by looking at the coverage conditions, picking the ones that seem easiest to satisfy, writing quick tests for them, and iterating until done. That's faster than thinking of coverage conditions as clues pointing to weaknesses in the test design. It's especially faster because thinking about test design might lead to "redundant" tests that don't increase coverage at all. They only find bugs.

So which are people going to do? Spend a larger amount of time writing better tests that rank the same as tests tailored for high coverage, or write the quicker tests?

Perhaps this might hint at the answer: when I talk about coverage to organizations that use 85%, say, as a shipping gate, I sometimes ask how many people have gotten substantially higher, perhaps 90%. There's usually a few who have, but everyone else is clustered right around 85%. Are we to believe that those other people just happened to hit 85% and were unable to find any other tests worth writing? Or did they write a first set of tests, take their coverage results, bang away at the program until they got just over 85%, and then heave a sigh of relief at having finished a not-very-fun job?

I'm not picking on these developers and testers. I've said before that it's **hard**, even for me, to use coverage well. It's easy for people of the best will to slip into using it as a crutch, a mechanical aid that replaces thought instead of encouraging it.

I'm picking on managers. It's their job to provide appropriate incentives - motivations to work in the interests of the company - or at least to avoid disincentives. Coverage numbers (like many numbers) are dangerous because they're objective but incomplete. They too often distort sensible action. Using them in isolation is as foolish as hiring based only on GPA.

Using coverage sensibly also avoids another testing mistake: not using it at all. Too often, I've seen inflated hopes get attached to coverage: at last, here is a quantitative way to get a handle on the mess that is testing! But a year passes and the coverage tool does not bring Utopia. Therefore, it must be useless. And the testing organization lurches wildly in yet another direction.

[7] 85% is a common number. People seem to pick it because that's the number other respectable companies use. I once asked someone from one of those other respectable companies why they used 85%. He said, "When our division started using coverage, we needed a number. Division X has a good reputation, so we thought we'd use the number they use." I didn't follow the trail back through Division X. I have the horrible feeling that, if I traced it all the way back to the Dawn of Time, I'd find someone who pulled 85% out of a hat. I don't know of any particular reasons to prefer one high number over another. Some claim that 85% is the point at which achieving higher coverage gets too hard. Based on my experience, I'd say that whether and where a "knee" in the effort graph appears is highly dependent on the particular application and the approach to test implementation.

What's all this about "having enough time"?

I said that I expect better than 90% coverage from my initial test suite, given sufficient time. For me, that means that a thoroughly tested program will take roughly twice as long to produce as one that's not tested at all.[8]

What if I don't have twice as long? What if the cost of bugs is too small to justify the time? What if deadlines simply don't permit that much testing?

It's still my responsibility to do as good a job as I can in the time allowed. There's always *some* time allowed. All but a few programmers do some testing, be it only unorganized tinkering that's usually called debugging.

Here's how I approach my job, and how coverage helps.

1. I divide the code under test into three categories.

   - **High risk** code could cause severe damage (wipe out data, injure someone, give non-obviously wrong answers that might cost a user a lot of money), or has many users (so the cost of even minor bugs is multiplied), or seems likely to have many mistakes whose costs will add up (it was a tricky algorithm, or it talks to an ill-defined and poorly-understood interface, or I've already found an unusual number of problems).
   - **Low risk** code is unlikely to have bugs important enough to stop or delay a shipment, even when all the bugs are summed together. They would be annoyance bugs in inessential features, ones with simple and obvious workarounds.
   - **Medium risk** code is somewhere in between. Bugs here would not be individually critical, but having too many of them would cause a schedule slip. There's good reason to find and fix them as soon - and as cheaply - as possible. But there are diminishing returns here - time spent doing a more thorough job might better be spent on other tasks.

   Clearly, these are not hard-and-fast categories. I have no algorithm that takes in code and spits out "high", "medium", or "low". The categories blend together, of course, and where debatable code lands probably doesn't matter much.

   I also oversimplify by treating risk as being monolithic. In reality, some medium risk code might be high risk with respect to certain types of failures, and I would tailor the type of testing (and the amount and rigor of code inspections) to the blend of risks. Discussing that would be too lengthy a digression from the topic of this paper.

2. I test the high risk code thoroughly. I use up most of the remaining time testing the medium risk code. I don't intentionally test the low risk code. I might hope that it gets

[8] Much depends on how easy the code is to test. A subsystem that's buried deep inside an operating system is inherently harder to test than one inside a compiler. More precisely, the tests are harder to execute, but not harder to design. If tests need to be automated, building the necessary scaffolding can range from trivial to as hard - harder! - than writing the code itself.

exercised incidentally by tests that target higher-risk code, but I will not make more than a trivial, offhand effort to cause that to happen.

3. When nearing the end of a testing effort (or some milestone within it), I'll check coverage.
   - Since high risk code is tested thoroughly, I expect good coverage and I handle missed coverage as described earlier.
   - I expect lower coverage for medium risk code. I will scan the detailed coverage log relatively quickly, checking it to see whether I overlooked something - whether the missed coverage suggests cases that I'd really rather a customer weren't the first person to try. I won't spend any more time handling coverage results than I would for thorough testing (even though there's more missed coverage to handle).
   - The coverage for low risk code is pretty uninteresting. My curiosity might be piqued if a particular routine, say, was never entered. I might consider whether there's an easy way to quickly try it out, but I won't do more.

So, again, coverage serves its purpose: I spend a little time using it to find omissions in my test design.

Looking at aggregate coverage is a matter of idle curiosity. It tells me more about how much of the code was low risk than it tells anything about testing.

How product testers can misuse code coverage

Product testers (independent testers, black box testers) differ from developer testers. Instead of focusing on a particular body of code, they are responsible for the whole product. Instead of looking closely at the code while testing, they may not look at the code at all. While they may know how to program, they may use few of those skills while testing.

How do these people use coverage? Here's a typical conversation, where Tom has gotten an evaluation copy of a coverage tool, instrumented the product, run the regression test suite, and looked at the aggregate results.

Tom:	Guess how much coverage we got!
Tester Little:	Sixty percent?
Tom:	Thirty.
Little:	Thirty! Oh, no! We're doomed! We'll be fired! The company will fail! *Runs in a circle, fluttering wings.*

There's another common response:

Tom:	We only get thirty percent coverage from our suite!

> Tester Eeyore: I knew it all along. I said we didn't have enough testing. But no one listens to me.

It's rarely as bad as all that. Much testing is not being measured: developer testing, testing through use, alpha and beta testing - in short, all the testing that is not part of the official regression test suite. For the aggregate coverage number to have any meaning at all, it must capture all sources of testing.

Once that's done, the aggregate number *still* doesn't mean much, for all the reasons discussed earlier. But coverage is still useful, because of clusters. If you have 70% coverage, it will not be uniformly distributed over the product. You are likely to have some "black holes" in the code: big chunks of code, perhaps entire features, that have not been tested at all.[9] If that was unintentional, you now know where to direct some testing effort.

You will also have some code that has been only lightly tested. Suppose you see that the contents of the directory *\src\product\net\server* are only 40% covered. If that's a surprise, you should do something about it:

1. The first step is to examine the tests and compare them to the external description of what that part of the product is to do (to the help pages, the user's manual, the specification, company folklore, or whatever). Did you overlook something?
2. Are the tests really testing what you thought they were testing? Did you make some grievous implementation mistake?
3. Finally, talk briefly to the developer to get an idea of what you missed. Sit by as the programmer scans the uncovered code and tells you what it does (from the user's perspective). You're looking for large patterns of missing tests. For example, if branch after branch of missed coverage in *net\server* are error checks, you know you undertested error handling. Thank the programmer, then go off and design error tests. Don't look in detail at the code:
   - It's not cost effective. You don't have time to learn the code.
   - You should be as worried about discovering errors the code overlooks as about exercising the errors it purports to handle.

All that's useful, but is coverage as much of a help to product testers as to developer testers? No, because product testers have more tasks than developer testers.

1. They either supplement or replace developer testing by testing particular features in isolation. ("In the drawing window, can you apply every transformation to every type of circle?")
2. They also test user scenarios that span features. ("Can you create, from scratch, a graphics-heavy document, print it, resize it for A4 paper, and print it again?") This type of testing is poorly measured by coverage. The problem is that coverage is a point

[9] I got the term "black hole" from Rich Conley.

measurement - has a particular branch on a particular line been taken? - but finding the targeted bugs depends on interactions between subsystems. Coverage will tell you that subsystem A has been exercised, but not that it's never been exercised after subsystem B, which (unknown to subsystem A's implementor) transforms the data that subsystem A uses.[10]

3. Product testers also test the product against reasonable user expectations, looking for requirements previously overlooked ("this had better work with 15 inch monitors even though everyone in this company uses 17 inch monitors") and usability problems, especially those that would grate on expert users ("it takes me five steps to do this operation I have to do twenty times a day").

A succinct way of putting this is that product testers are even more concerned with omissions than developers. Developers care more about whether their code does what was intended; product testers care more about whether what was intended is right. Because product testers are looking for requirements-level or specification-level omissions, coverage provides fewer and weaker clues than it does to developers, who are looking for design-level and code-level omissions.

Summary

I wouldn't have written four coverage tools if I didn't think they're helpful. But they're only helpful if they're used to *enhance* thought, not *replace* it.

Acknowledgements

I thank James Bach, Danny Faught, Erik Petersen, Mike Powers, and Roger Turner for comments on a draft of this paper.

Bibliography

[Glass81]
Robert L. Glass, "Persistent Software Errors," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 2, March, 1981.

[Herman76]
P.M. Herman, "A Data Flow Analysis Approach to Program Testing," *Australian Computer Journal*, Vol. 8, No. 3, November, 1976.

[10] I oversimplify. Dataflow coverage (as described in [Herman76] or [Rapps85]) measures whether some bit of code is exercised after some previous bit. I am unaware of its successful use in whole-product testing of products with perhaps thousands of routines. There needs to be some way of winnowing down the possibilities to those really worthy of testing, and of presenting coverage information in a way that can be understood without much knowledge of internals. You could also measure which routines in subsystem A have been exercised after each routine in subsystem B. Again, I don't know whether that's been done in practice. I've done something similar (but simpler to implement) for the specialized domain of operating system stress testing that targets locking problems.

[Kaner96]
C. Kaner, “Software Negligence and Testing Coverage,” Proceedings of STAR 96 (Fifth International Conference on Software Testing, Analysis, and Review), Software Quality Engineering, Orlando FL, 1996. (http://www.kaner.com/coverage.htm)

[Rapps85]
S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, 1985.