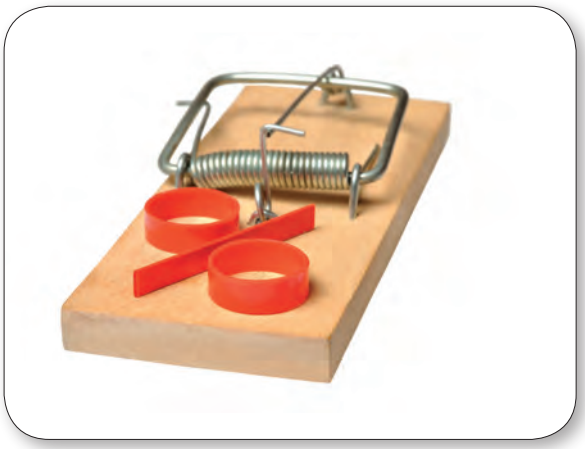
The two big traps of code coverage

By Arthur Hicken, Parasoft

Code coverage is important, and improving coverage is a worthy goal. But simply chasing the percentage is not nearly so valuable as writing stable, maintainable, meaningful tests.



Measurement of code coverage is one of those things that always catches my attention. On the one hand, I often find that organizations do not necessarily know how much code they are covering during testing – which is really surprising! At the other end of the coverage spectrum, there are organizations for whom the number is so important that the quality and efficacy of the tests has become mostly irrelevant. They mindlessly chase the 100% dragon and believe that if they have that number the software is good, or even the best that it can be. This is at least as dangerous as not knowing what you have tested, in fact perhaps more so since it can give you a false sense of security.

Code coverage can be a good and interesting number to assess your software quality, but it is important to remember that it is a means, rather than an end. We do not want coverage for coverage's sake, we want coverage because it is supposed to indicate that we have done a good job testing the software. If the tests themselves are not meaningful, then having more of them certainly does not indicate better software. The important goal is to make sure every piece of code is tested, not just executed. Failing enough time and money to fully test everything, at least make sure that everything important is tested. What this means is that while low coverage means we are probably not testing enough, high coverage by itself does not necessarily correlate to high quality – the picture is more complicated than that.

Obviously, having a happy medium where you have "enough" coverage to be comfortable about releasing the software with a good, stable, maintainable test suite that has "just enough tests" would be perfect. But still, these coverage traps are common.

The first trap is the "we don't know our coverage" trap. This seems unreasonable to me – coverage tools are cheap and plentiful. A friend of mine suggests that organizations know their coverage number is not good, so developers and testers are loath to expose the poor coverage to management. I would hope this is not the usual case.

One real issue that teams encounter when trying to measure coverage is that the system is too complicated. When you build an application out of pieces on top of pieces on top of pieces, just knowing where to put the coverage counters can be a daunting task. I would suggest that if it is actually difficult to measure the coverage in your application, you should think twice about the architecture.

A second way to fall into this trap happens with organizations that may have a lot of testing, but no real coverage number because they have not found a proper way to aggregate the numbers from different test runs. If you are doing manual testing, functional testing, unit testing, and end-to-end testing, you cannot simply add the numbers up. Even if they are each achieving 25% coverage it is unlikely that it is 100% when combined. In fact, it is more likely to be closer to the 25% than to the 100% when you look into it.

As it turns out, there is in fact a way to measure and add coverage together in a meaningful fashion. At Parasoft, we leverage the vast amount of granular data captured by our reports and analytics tool, Parasoft DTP, which we can use in this context to provide a comprehensive, aggregated view of code coverage. Our application monitors are able to gather coverage data from the application directly while it is being tested and then send that information to Parasoft DTP, which aggregates coverage data across all testing practices, as well as across test teams and test runs. If that sounds like a pretty significant amount of information, you are right! DTP provides an interactive dashboard to help you navigate this data and make decisions about where to focus testing efforts.

If multiple tests have covered the same code, it will not be overcounted, while untested parts of the code are quick and easy to see. This shows you which parts of the application have been well tested and which ones have not. So, no more excuses for not measuring coverage. This leads us to the next trap – the "coverage is everything" perspective. Once teams are able to measure coverage, it is not uncommon for managers to say "let's increase that number." Eventually the number itself becomes more important than the testing. Therein lies the

Parasoft DTP example dashboard

problem – mindless coverage is the same as mindless music. The coverage needs to reflect real, meaningful use of the code, otherwise it is just noise. And speaking of noise… the cost of coverage goes up as coverage increases. Remember that you not only need to create tests, but you have to maintain them going forward. If you are not planning on reusing and maintaining a test, you should probably not waste time creating it in the first place. As the test suite gets larger, the amount of noise increases in unexpected ways. Twice as many tests may mean two or even three times as much noise. The meaningless tests end up creating more noise than good tests because they have no real context, but have to be dealt with each time the tests are executed. Talk about technical debt! Useless tests are a real danger.

Now, in certain industries, safety-critical industries for example, the 100% coverage metric is a requirement. But even in that case, it is all too easy to treat any execution of a line of code as a meaningful test, which is simply not true. I have two basic questions I ask to determine if a test is a good test. What does it mean when the test fails? What does it mean when the test passes?

Ideally, when a test fails, we know something about what went wrong, and if the test is really good, it will point us in the right direction to fix it. All too often when a test fails, no one knows why, no one can reproduce it, and the test is ignored. Conversely, when a test passes we should be able to know what was tested – it should mean that a particular feature or piece of functionality is working properly.

If you cannot answer one of those questions, you probably have a problem with your test. If you cannot answer either of them, the test is probably more trouble than it is worth.

The way out of this trap is first to understand that the coverage percentage itself is not the goal. The real goal is to create useful meaningful tests. This of course takes time. In simple code, writing unit tests is simple, but in complex real-world applications it means writing stubs and mocks and using frameworks. This can take quite a bit of time and if you are not doing it all the time, it is easy to forget the nuances of the APIs involved. Even if you are serious about testing, the time it takes to create a really good test can be more than you expect.

At Parasoft we have an answer for this: the Unit Test Assistant inside the Java development testing tool (Parasoft Jtest) takes on the tedious tasks of getting mocks and stubs right. It can also help expand existing tests in a useful way to increase coverage – helping you create good unit tests as well as make recommendations to improve test coverage and test quality as well. ■