Using Code Coverage Analysis to Focus Testing Strategies

A Paper by Technology Strategy Research
A Practical Guide to Analyzing the Value of Developer Unit Tests

Many consider unit testing essential as a first step in identifying and fixing software bugs early in the process. Unit tests exercise specific parts of code to ensure that the functionality of that code is what the developer expects.

However, unit tests alone can't assure defect-free code. Without knowledge of how much code, and which code, is being exercised by unit tests, developers have no objective measure of the quality of the code. They may be able to say that their code passed its unit tests but can't make any statement about how much of the code has been exercised.

This is where code coverage analysis comes in. By determining what lines of code were executed during unit tests, developers and development teams can ensure that their tests have exercised enough code to gain a level of confidence that defects have been minimized. To be truly effective, unit testing and code coverage must be used together.

Developers can devise effective unit testing strategies using code coverage and cyclomatic complexity analysis to address application quality earlier in the development lifecycle. Employing complexity analysis to focus unit testing, and then using code coverage to ensure unit testing provides the necessary coverage of code enables developers to identify and correct defects long before they reach end users. Further, it enables both developers and testers to determine whether or not existing unit tests exercise sufficient code to provide a level of confidence that defects have been minimized.

Even unit tests with code coverage are insufficient at testing code at an early stage to verify functionality. It is difficult, if not impossible, to unit test all of the lines of code. For example, it is difficult to unit test error-handling code or code that requires a difficult-to-replicate set of conditions to execute. It is common for most application development efforts that only 60 to 80% of code can be easily unit tested.

Therefore, developers should focus unit tests where the value to the application is the greatest—the most complex code. Complexity analysis, such as McCabe’s Cyclomatic Complexity, is a way of measuring the complexity of an application. It directly measures the number of independent paths through a program's source code. The more paths there are through a unit, the more complex that unit is, and the more unit tests it will likely require. This is also likely the most fragile area of the application, so any changes here have the largest likelihood of failure.

Developers can measure cyclomatic complexity with the appropriate tool without actually running the evaluated code. This enables developers to understand where their complex code exists, so that they can concentrate on developing unit tests to exercise that code. It's typically a simple computation that with automated tools takes only a few minutes on most applications.

Developers can use code coverage to ensure their unit tests exercise sufficient code. Code coverage can provide a snapshot of the results of unit or function testing at many stages in the development lifecycle.
tests and functional test cases are adequate. Many teams set code coverage standards that must be met prior to software deployment, so that there is a high level of confidence that the vast majority of code has been tested.

Developers can use code coverage analysis in a number of ways to improve code quality. It can help developers identify dead code; that is, code that never executes. By running both unit and functional tests, developers can see what code is associated with what capabilities, and determine what code never executes. It's likely that some of that code is for error handling or edge cases, but it represents a good first approximation of code that may never execute.

Code coverage can also occur in conjunction with functional testing although its purpose in this role is somewhat different. Quality professionals typically perform functional testing and ensure that the application meets user requirements. Using code coverage with functional tests enables the development team to ensure that functional tests are comprehensive.

Code coverage can be built into any automated build process to serve yet another purpose. If a development team is performing nightly or continuous builds, running code coverage in conjunction with smoke tests enables the development team to determine how effectively its smoke tests are evaluating critical execution paths. This is especially effective with debug builds, which enable developers to better analyze and diagnose the location of any defects found during unit testing.

**Getting Ready for Code Coverage**
Performing effective code coverage analysis is more difficult than it looks. It requires building good unit tests, focusing those tests on the most important code and code paths, and setting up those tests to run on a regular basis without significant effort by the developers.

Building unit tests is one of the more difficult but necessary aspects of software engineering. The purpose of a unit test is to execute all functional paths through a particular code unit, to verify a basic level of correctness. To do so, developers need to have a high level of understanding of the unit under test, which is why most developers write their own unit tests.

The developer creates an input file with a range of inputs, and executes the unit with those inputs. The outputs are collected during execution and compared with expected outputs. Any deviation indicates a defect or other unknown event in the unit that must be investigated.

Because the developer doesn't know what code in the unit generated the defect, the unit test result itself provides little clue as to where to begin the investigation. If the developer executed code coverage in conjunction with the unit test, it would be possible to know what lines of code were executed, and how many times those lines were executed, providing a starting point for analysis and diagnosis.

**Running Code Coverage with Unit Tests**
Developers can run static code reviews with cyclomatic complexity analysis as they develop code and assign it to functional units. This will pinpoint the units that have many pathways and are potentially more complex than average. Developers can write unit tests as units mature, based on the results of cyclomatic complexity analysis.

Unit tests typically run in standalone mode with code units that enable discrete inputs and outputs. They typically execute within a unit test harness such as NUnit, enabling them to automate the process while collecting and recording data on the results of the test.

As soon as each individual unit is buildable, developers can finalize unit tests and add them to the test harness. Once there exists a body of code units and unit tests, individual developers can begin executing their own unit tests locally.

**Figure 2.** Code coverage can help developers get down to the individual line of code to determine what code paths were taken, and how many times they were taken, during unit testing.
Developers shouldn’t be surprised if their initial unit tests exercise less than 50% of their units. It’s difficult to write tests that execute every possible code path, and every line in every path. They can use the initial code coverage results and cyclomatic complexity analysis to determine where best to write additional unit tests.

Once an application has been built successfully and is undergoing nightly or continuous builds, code coverage can be configured to execute in batch mode, in conjunction with smoke tests. While smoke test primarily ensure a successful build and basic functionality, code coverage can help ensure that enough of the application is being tested to ensure that the build is useful.

At the point where the application is buildable, developers can also start combining their unit tests into a single test harness for more comprehensive coverage. While unit tests by design are intended for individual units, combining them once the application is building provides more comprehensive information for the team and management on testing status.

At this point, code coverage can move into the functional test realm. Testers employ code coverage tools while running functional test cases and can examine the results to ensure that their tests are exercising enough of the code to be comprehensive. In addition, they can compare their code coverage results with those of developer unit tests to determine what parts of the code base they are not exercising.

**Steps for Using Unit Tests Effectively**

Most development teams have their own unique methods for building, unit testing and functional testing. The following guide can be adapted to fit into just about any process, as long as the team is committed to unit testing and regular builds.

1. For every unit of developed code, evaluate its complexity by running McCabe’s cyclomatic complexity analysis or similar complexity measure. Rank units from most to least complex.
2. Using the cyclomatic complexity measure, determine the most critical and complex units in the application.
3. Write initial unit tests with the cyclomatic complexity measure in mind. Focus on the most complex units for the most unit tests.
4. Run code coverage in conjunction with unit tests. Examine code coverage results and write additional unit tests to ensure a more comprehensive level of coverage for complex units.
5. As the application becomes more complete and buildable, combine the unit tests into a single test harness to execute across the code base.
6. For nightly or continuous builds, run code coverage in conjunction with the smoke tests after the build is complete.
7. Once the application is available for functional testing, employ code coverage with functional tests to ensure broad coverage. Automate functional tests and code coverage where feasible.
8. Set code coverage percentages as a goal for defining when the application is complete and ready for deployment.

Using code coverage across the application development lifecycle, development teams can have greater confidence in their testing regimen; identify, analyze and fix defects earlier in the lifecycle; and better understand what tests need to be written and executed. Complexity analysis provides a means for developers to focus their testing to the most complex units of code. Together, these simple practices provide development teams with a roadmap to higher application quality and a better understanding of their testing practices.