Half-Century Old Techniques within DevOps

An exploration of the roles of code coverage and software metrics within a modern software engineering framework

Richard Uggelberg

1 Introduction

In this essay I will introduce two concepts of software engineering, code coverage and software measurement. More importantly, however, I will also reflect on their relation to each other and their combined importance and to the field of software engineering as a whole. The goal of which is to create better and more reliable software. To understand this essay the reader is assumed to have a good general understanding of the software development process but not necessarily any in-depth knowledge of specific aspects. If anything, an understanding of general practices of software testing and verification would be beneficial. That is because the topics of code coverage and software measurement follows from just that line of thinking.

Furthermore I will examine these concepts in the context of DevOps. An interesting exercise since they are products of very different eras in software engineering. While, in the olden days of computing, there was the notion software development as a science and with a firm basis in mathematics as a core. However, as with many things, theory and practice often diverge. As perfection is the enemy of the good, while trying to deploy new software features we also have to factor in the speed at which this can be done. What is meant to be highlighted here is the intersection of these two modes of thinking.

If code coverage and software measurement are ways of developing software that is correct, and the mixture of development and operations of software is a way to speed up delivery, then can these not be merged together successfully to ensure higher quality while maintaining deployment rate? Naturally, there are many ways of ensuring software quality. The ones brought up by this essay can be, in my experience, underutilized.

2 Background

In our quest to develop better software faster we have many tools and techniques at our disposal. One aspect of this that always seems to remain troublesome, though, is the question of if the code that is written is actually correct. Not only if it is syntactically correct but we also wonder if it does what we think it does. Usually this can be determined through testing.

We seek to understand what parts of our code is tested, or as we say, covered by testing. This is where we get the notion of code coverage. However, as testing becomes a more valued part of software development, however, we run into a new issue. As projects grow in size so does the amount of testing required. Not necessarily linearly, at that. This is how we get from testing to the idea of software measurement. Simply put, code coverage determines what is tested and with software measurement we can get another perspective. Namely, what should be tested.

2.1 Code Coverage

To start off, though, we need to define what we mean by code coverage, more precisely. This section is generally based on chapter 9 in the book Software Reliability Methods by Doron Peled[5]. It is important to state that code coverage only applies to certain types of testing. Quite obviously it is difficult to judge from the code alone whether or not acceptance tests set up by a customer will pass or not. Thus we are limited to mainly technical aspects. That being testing such as unit testing, integration testing and regression testing. Basically, as we've stated previously, whether or not the code behaves as expected.

Whatever the aspect that we would like to test, we need to understand the notion of a execution path. That is, what parts of a program are executed during a certain testing scenario. If we want to assure that all parts of the program runs correctly we want to involve all parts in at least one execution path. This is how we arrive at code coverage analysis.

Of course, it is not all too simple. As there are different types of testing, there is also different types of code coverage. Defining code coverage by the proportion of covered execution paths is simply infeasible since the amount of paths grows exponentially. This is called *path coverage*, by the way, but is wholly impractical as a measurement.

2.1.1 Statement Coverage

Starting at the other end, however, we have the notion of statement coverage. What is, then, a statement and what does it mean that a statement is "covered"? A statement is any executable statement in the code. Be it an assignment or conditional statement. Note that this requires that only one evaluation of a conditional statement is included.

2.1.2 Edge Coverage

Edge coverage expands upon statement coverage with the criterion that conditional statements need both evaluations covered by testing to be considered "covered". Note here that this does not include the possible complexities of compound conditional statements. Only the possible evaluations of them as a whole.

2.1.3 Condition Coverage

Condition coverage addresses just that. Here we're interested in every condition, even inside larger conditional statements. An important distinction here is that condition coverage does not include all evaluations of conditional statement and thus is not necessarily an expansion of edge coverage.

Edge- and condition coverage can, of course, be combined to form edge/condition coverage that, then, is an expansion on both of them. Furthermore if we want to consider all combinations of of conditions we have what's called *multiple condition coverage* but then we're starting to venture into impractical territories.

2.2 Software Measurement

As stated before, we touched upon the subject of possible impracticalities of trying to achieve 100% code coverage. No matter which type of coverage we're talking about. What can we do then? Well, I think it is quite possible that most of us have the notion that all code is not equally likely to produce errors. Naturally we would like to focus our testing efforts on the more error prone code. It is hard to know whether or not the important parts are covered by testing. Thereby, we arrive at *software measurement* or *software metrics*.

What to measure is then a concept central to this essay. Mainly, we associate error prone code with complex code. Of course, as long as software is still written by humans we still experience human error and humans are more likely to make mistakes when things get complicated.

Before we get to more advanced defintions we'll start with the metrics that most already know. Whether knowlingly or unknowingly there's possibly the common conception that simply sections of code that are longer are more likely to contain errors. This is, of course, the measure of *lines of code* or *LOC*. Depending on the context we might have more use for *non-comment lines of code* but I think the idea is intuitive enough. This is coupled with the idea of statement coverage, or maybe more commonly, line coverage. If we wish to know the percentage of lines executed at least once then we naturally need to know the number of lines that can be executed.

2.2.1 McCabe's Cyclomatic Complexity

As we associate errors with complexity we also need some measure of complexity. Maybe most notably we have the attempt at such a measure by McCabe[3] all the way back in 1976. This is however, quite commonly used even today and thus requires a bit of an introduction.

The main goal of the measurement was to keep code maintainable and testable and to measure this by quantiative means. McCabe, being a mathematician, based the notion on the control flow graph of a program. This is, simply, a graph representation of a program.

Consider a graph G with n vertices, e edges and p connected components. The cyclomatic complexity number $V(G)$ for this graph is then defined using the following equation.

$$v(G) = e - n + 2p \tag{1}$$

What actually is measured is the amount of linearly independent execution paths. If this, conceptually, seems a bit too disconnected from the rest of the discussion, the key points here is that the measure has a mathematical basis and that the measure produces a number corresponding to complexity.

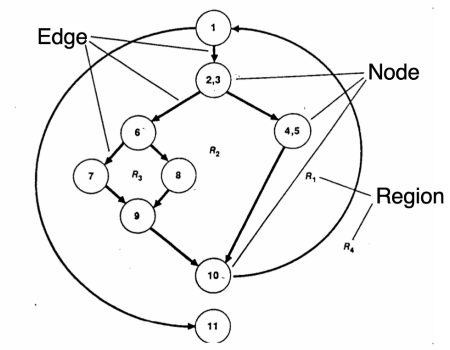


Figure 1: An example of a control flow graph with elements of edges, nodes and regions highlighted. The notion of edges (or sometimes, branches) here is the same as in code coverage. Hence the linkage between the two concepts. [6]

Realistically though, using a graph representation is not an all too practical way to measure software in practice. How this measure is produced then is through simply counting the branching condition of code. This has some limitations though since certain rules of properly structured programming need to be followed. However, the numbers should be similar either way.

This idea is closely coupled with the idea of edge coverage or branch coverage in testing. As our program might be divided into different modules we could easily get notions of where additional testing might be beneficial.

2.3 Halstead's Metrics

Around the same time as McCabe there was a bit of different approach made by Halstead[2]. Halstead's metrics are calculated directly from the code itself by simple categorization of the elements into to distinct categories. These are the *operations*, what actions that can be performed, and the *operands*, those on which actions can be performed.

From these two classes of elements we can construct a set of four numbers. These are:

- n_1 The number of unique or distinct operators in the program
- N_1 The total number of operators in the program
- n_2 The number of unique or distinct operands in the program
- N_2 The total number of operands in the program.

Naturally, what is and operand or operation depend on a lot of factors. Maybe most notably the language on which the measure is intended to be used. Either way, when that is done we can continue the calculations to get some useful numbers.

- *Length* is $N_1 + N_2$. The total number of operators and operands.
- *Vocabulary* is $n_1 + n_2$. The number of unique operators and operands.
- *Difficulty* is $(n_1/2)*(N_2/n_2)$. The number of unique operators, divided by two, multiplied by the total number of operands in relation to the number of unique operands.

From all this, three more metrics are defined.

- *Volume* is *Length + log_2(Vocabulary).* This can be seen as the amount of code that has to be understood by the user.
- *Effort* is *Difficulty * Volume.* This is the estimated amount of effort it takes to recreate software. Given the combination of difficulty and and volume, this makes sense.
- *Bugs* is *Volume/3000* or $Effort^{2/3}$*/3000.* This estimates the how many bugs there are in a system. The second equation is Halstead's original equation but the first has been seen as more appropriate in recent years.

All these metrics are, naturally, supposed to represent a quantitative measure of what their names suggest. Whether or not that is actually true is subject to further discussion. They do, of course, represent *something.*

Something important to state at the end of this section here is that there are tons of ways to measure software. These ways are generally divided into *static analysis* and *dynamic analysis*. Both cyclomatic complexity and Halstead's metrics are both clear examples of static analysis. That is, measurements that can be taken without running the code.

3 Discussion

What is then the usefulness of measurements such as complexity or code coverage? Well, no matter how we produce software we want to make sure that what we do works. Especially in the context of DevOps, we also want to put in as little effort as possible when testing and making sure that everything works the way we think it does. If we also want to easily add features to software then making sure our code is easily maintained is of chief importance. Having an idea of where there is complexity within a system and then reducing it might help with maintaining it.

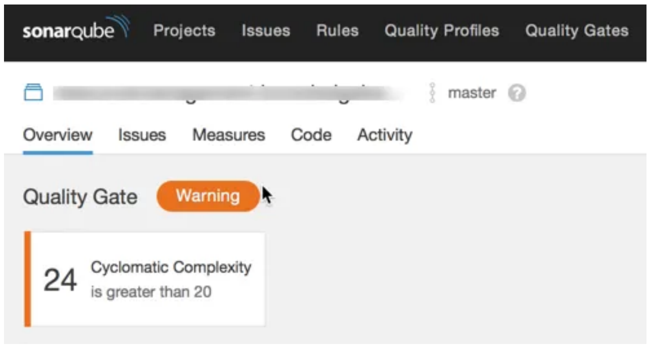
The main benefit of software measurement might be in the lack of effort and time category. This is because, especially static analysis, like the ones described earlier, require almost no effort at all from the side of either development or operations to conduct. Simply running a script, or more likely, as part of a monitoring tool. When writing code and its tests we can see where tests might do the most good. When monitoring existing and running software we can be wary of where bugs and issues might likely have originated in the event that something goes wrong.

3.1 Modern Role

To put this into perspective a bit more clearly we can mention how this is, and maybe more interestingly, *could* be used in practice as part of development cycles.

Unless you're already developing software strictly using Test-Driven Development then you need to spend time writing tests for already existing code. What then if your code base is huge and code coverage not satisfactory? Do you randomly choose what to test? Naturally, in order to spend time and effort where it is most needed we need some sort of system of rules. Or, at least, guidelines. This is a role that static analysis software measurements such as cyclomatic complexity or Halstead's metrics might serve. Provided we don't have parts of the code that are more critical than others we want to focus our attention on the parts that are most likely to produce errors of any kind.

How then are these used, if at all? For example, cyclomatic complexity is used as a metric in the Visual Studio IDE to along with other static analysis metrics [4]. Similarly there are more dedicated static analysis tools such as SonarQube [7]. The latter also claims to seamlessly integrate into a CI/CD pipeline as part of a quality check step before deployment. It supports integration with many other DevOps tools and should thus validate the viability of static analysis in a DevOps context. [1] Of course, it is a much more sophisticated tool than to just use the software measurements detailed in this essay. However, I believe that having a deep understand of how static analysis tools for code quality work is beneficial when using tools like these.



24 Cyclomatic Complexity is greater than 20

Figure 2: A very simple showcase of how cyclomatic complexity may be used in the real world. In this case it is set to warn the developer about pieces of code the exceed a complexity of 20. I.e. having more than 20 linearly independent executions paths.

3.2 Limitations

Speaking of importance of deep understanding, there is also the aspect of limitations when it comes to software measurement and code coverage. The most obvious one, that also applies to both code coverage and software metrics, is the issue of simplification. Source code is complex and thus difficult to reduce to a quantitative value. This may make them somewhat illusive. Similar percentages of code coverage, especially simple line coverage, might indicate that two vastly different sets of source code are equally reliable.

Similarly, while the notion of cyclomatic complexity is meant to conform to our intuitive notion of cognitive complexity that is not always the case. This is easily imagined with switch-case statements where there is a lot of branching but, for many programmers, not much cognitive complexity. The same can be said about Halstead's metrics such as "difficulty". There must certainly be cases where this notion of difficulty does not at all conform to our intuitive ideas.

Furthermore, there is the operations side of DevOps. These measures are mostly beneficial to the development side as they are applied to the source code. Sure they might provide some indication of the origin of defects but that's about it as far as I see it.

3.3 Bottom Line

As with anything, software measurement and code coverage are not without their benefits and drawbacks. However, as testing remains and important part of software development there is always the issue of what to test. With code coverage we can an idea of what is tested. Depending on what type of coverage is used the information provided might be different in order to give us more to go on. However, in terms of software metrics, we can get a different idea of what *should* be tested (or simply changed, for that matter) which is a related but not synonymous idea. The benefits of these things should not be overstated. They have not and, most likely, will not revolutionize the way we write code much like CI/CD pipelines and agile development. However, I believe there is place for these relics of the early days of software engineering in our modern ways of creating and maintaining software.

4 Conclusion

DevOps is a way to streamline the development and maintenance of software. Not only to ensure quick turnaround times for features and bug fixes. But also to ensure that what is set into production is sound and reliable code. After all, unreliable code requires more maintenance, putting stress on the operations team.

As one of the most fundamental ways of ensuring reliable code is testing, we seek to ensure that as much as possible of our code is tested in order to be sure that it will survive updating and reworking. This provides us with information on what can be done to increase the reliability. There are, however, more ways to measure exactly that. That is, static analysis software metrics. These provide indications of where code might be too complex and thus error prone or where an errors might've originated when it's already appeared.

Either way, these things will not likely make or break an entire production process. However, static measurements can easily be run as part of any development cycle to ensure some level of code quality and reliability. Thus there is little reason not to use them.

References

[1] P. Gowtham. *SonarQube — DevOps (Static Code Analysis)*, (accessed May 29, 2020). Available at `https://medium.com/@potureddigowtham/sonarqube-devops-static-code-analysis-8b32fe8a362d`.

[2] Maurice Howard Halstead et al. *Elements of software science*, volume 7. Elsevier New York, 1977.

[3] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[4] Microsoft. *Calculate code metrics - Visual Studio*, (accessed May 29, 2020). Available at `https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019`.

[5] Doron A Peled. *Software Reliability Methods*, chapter 9. Springer-Verlag, New York, 2001.

[6] Daniel Solano. *White Box Test: Cyclomatic Complexity*, (accessed May 29, 2020). Available at `https://medium.com/akurey/white-box-test-cyclomatic-complexity-276fb5803a04`.

[7] SonarQube. *SonarQube*, (accessed May 29, 2020). Available at `https://www.sonarqube.org/`.