A Description of Software Metrics and Their Seeming Uselessness

Richard Uggelberg

April 2020

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 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.

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.

3 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[4]. 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.

3.1 Types of code coverage

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.

3.1.1 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.

3.1.2 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.

3.1.3 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.

4 Software Measurement

As stated before, in the background section, we touched upon the subject of possible impracticalities of trying to achieve 100% code coverage. 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*.

4.1 Types of software measurement

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 possibly the common conception that simple section 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.

4.1.1 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.

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.

4.1.2 Halstead's metrics

Around the same time as McCabe there was a bit of different approach made by Halstead[1]. 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 *operanads*, 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.

5 Conclusion

So what is the usefulness of all these measurements? Software metrics and code coverage are not the largest of topics and thus their use, especially together, is limited. It is quite telling that a quite comprehensive book on software reliability such as Peled's[4] does not mention software metrics, as far as I could see. Therefore, also, the research regarding code coverage and software metrics in conjunction is rare. This despite them being, in my mind, closely correlated. It is hard, at least relatively speaking, to find research from the last 10 years on these topics. Much was done in the early days of computing. A big reason that McCabe and Halstead were brought up here is because they were not only influential in their findings, but also for their vision of computing as a harder science than it was or has become.

Naturally, I have not been able to scour the entire internet for all the available information. However, to me there seems to be a somewhat of a consensus among researchers and laymen alike. That is plainly this: That code coverage is beneficial [5], at least to some degree, and that software metrics, like those mentioned here, have fallen out of favor [2]. The sources presented are mere examples of what I believe is a wider sentiment. Not only through my experience but also time spent researching these exact topics.

It is quite telling how little research into software metrics have been done. But also, how little of it is actually used. Most who speak or write of it are academics and what happens in academia does not necessarily reflect what software developers will actually use. Something that also demonstrates this disparity is the different affinity for functional programming which i, relatively, popular in academia because its' relation to mathematical notation and lambda calculus.

How does this relate to the modern DevOps format then? Well, my understanding from this research is that software metrics, as they are described here, don't have much to add. That code coverage would be more useful seems likely, but also dependent on context. As with most things, there is not clear answer. If there's room in the development cycle for measurements here and there, what harm is there? If, though, there is more to do in interpreting these measurements than there is in conducting them, then maybe time and effort is better spent elsewhere.

However, this does not have to be the last we hear of these measurements. There are many measurements that seem to be gaining popularity. They are not the same as those mentioned in this text, though. They relate more to other areas of software development such as organization and planning. Although, if the way we develop software can influence how programming languages and other tools are formed, then cannot also change in development styles change what measurements we use? Only time will tell.

References

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

[2] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 131–142, New York, NY, USA, 2008. Association for Computing Machinery.

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

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

[5] W. E. Wong, Y. Qi, L. Zhao, and K. Cai. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 449–456, 2007.