

Can we rely on code coverage to keep the code  
safe and bug-free?

Arvidsson, Isac  
`isacarv@kth.se`

Henriksson, Andreas  
`anhenri@kth.se`

May 2021

# 1 Introduction

With its over 2,6 million hits on Google Scholar, code coverage is seen as a well-discussed metric. Also, there exists a large amount of tools to measure code coverage for different programming languages [1]. This could be seen as a sign of its importance in software development.

The code coverage purpose isn't defined or restricted in any way and could therefore have many purposes and meanings. One of them being a measurement on how well tested the source code is. With that said, it isn't clear how well code coverage accomplishes this. Hamedy has mentioned a few purposes for code coverage in a post on codeburst [2]. Since it could be beneficial in both creating and selling the software it is already convincing that code coverage matter and is of importance in software development. But, then the question of trust in the code coverage is raised. Could we rely on the metric? Are there other alternatives that are more reliable?

The essay will discuss the pros and cons with relying solely on code coverage as well as looking at other alternatives. More precise, it will look in to the answer to the question if we can rely on code coverage to keep the code safe and bug-free?

## 2 Arguments against relying on code coverage

The first question to ask should be, what is code coverage not telling us? Given a measured value on the code coverage, what do one miss out? Behind the scenes of code coverage there are actually dark areas where it is possible to manipulate the metric.

Let a measured value of code coverage be 80% as recommended in [3]. If we are talking about line coverage, this means that 80% of lines of the code are tested. This could be seen as a great measurement because purely probabilistic, the chance of a bug-free code is high since there are only 20% of the code that could contain safety risks. The dark area in this measurement is what part of the code that is tested. If 80% of the code base consist of easy logic that rarely creates problem and 20% of the code is the crucial part of the hole program, then the choice of what 20% to leave out of the testing makes a huge difference. In other words this could be interpreted as the following: if we just cover the best 20% of the code, then we could be more safe than the earlier coverage of 80%.

Another argument for not falling into the trap of relying on code coverage is the measurements drawbacks. The tool itself is very limited to unit testing and doesn't always provide a fair metric in either directions.

For example, sometimes more coverage doesn't even have to be better, like in cases where some parts of the code could be viewed as unnecessary to reach. Another example of flaws in the metric is that it can be hard to use if a project is using more than one language. If this is the case it would often lead to having to use different tools which provide different results for different parts of the code.

If there would be possible to obtain a satisfied level of code coverage for a project, which is subjective and not commented in this case, is it worth all the time and effort it takes to first get there and then to keep it at least as high over its development?

Software today does often contain millions of lines of code, and most of them keeps on growing [4]. So when the code base grows over time it's harder to maintain relevant and useful test cases. An example on this is in figure 1 which shows the open source project Teammates line coverage over time. It shows how the line coverage decreases over time when also the complexity increases. With this in mind, is it necessary to strive for such high code coverage that one should be able to trust it?

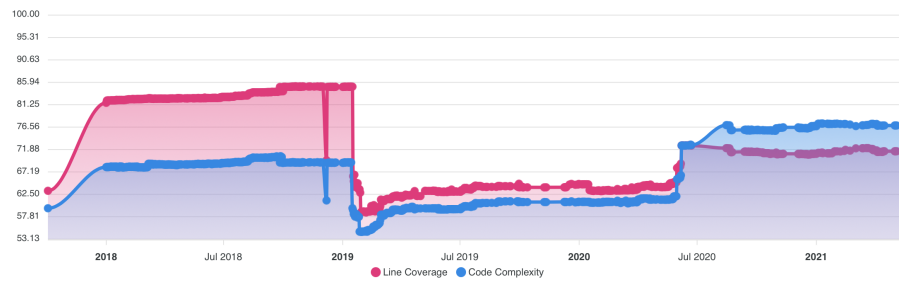


Figure 1: Illustrating the complexity and line coverage in percentage over time(See [\[5\]](#))

### 3 Arguments for relying on code coverage

Even if the metric, code coverage, seems to be easily manipulated, limited and costly to maintain there should be some arguments for its reliability, how could it else be so popular?

There's a hardness when deciding if code coverage can imply safe code in the general case. In the general case, code coverage that unsophisticated counts the number of lines in the code that's tested doesn't say much. However, since code coverage contains several different methods for measuring coverage, it might be appropriate to use different methods for their different strengths and thus perhaps be able to rely on the measured value[6]. For an example see figure 2 where line-, function- and class-coverage are presented.

Legend: Low: 0% to 35% Medium: 35% to 70% High: 70% to 100%










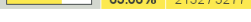
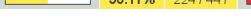

	Lines		Coverage		Classes	
			Functions / Methods			
Total		65.20% 2346 / 3598		54.14% 281 / 519		18.06% 13 / 72
Diff		98.06% 202 / 206		93.33% 56 / 60		75.00% 6 / 8
StickToThatLanguage		10.43% 12 / 115		8.33% 1 / 12		0.00% 0 / 2
Wikibase		65.06% 2132 / 3277		50.11% 224 / 447		11.29% 7 / 62

Figure 2: Illustrating different metrics of coverage(Taken from [7])

Code coverage comes with line coverage [6]. Line coverage can be used to ensuring that the code runs without termination. This means that the part of the code that's covered is guaranteed to not cause any unexpected termination for the specific case. So in some sense if one could use this right, one could also guarantee or rely on the code coverage to keep the code clean from unexpected terminations. Of course this is a huge claim since the guarantee is only valid in the covered code for the specific case that was tested, but the more code that's covered the more we could rely on it to not terminating unexpectedly for simple cases that is tested.

Code coverage can help with keeping the code healthy overtime when a project grows and becomes bigger. It's helpful to find dead parts of the code that are non-reachable which leads to less unnecessary code as well as heightens the readability of the code. Keeping a high code coverage also pushes developers towards keeping good habits by always writing good unit tests while also updating old tests as the source code changes. With that said, overseeing code coverage often help with the cyclomatic complexity [8]. The higher cyclomatic complexity there is, the harder it's to push the code coverage. With keeping a high code coverage we can ensure that unnecessary code is found and the complexity may be reduced.

## 4 Complement to code coverage

An addition to only using code coverage is to use test coverage as a metric [9,10]. One important thing to note is that one doesn't exclude the other. Test coverage is a black-box approach to make tests for a project. The focus is laying on the requirements of the system and looking more at what a certain system/function should produce. One could argue that test coverage is more focused on the quality of the test rather than the sheer quantity of what parts are tested. With good test coverage developers can be sure that there code produces the correct output with assertion. This should be seen as a complimentary to code coverage. Using only test coverage comes with the problem that it's hard to calculate and doesn't provide a comparable result.

It is possible to enforce tests such as they are written with purpose. In this article [11], this type of code coverage is called "clean code coverage". It might seem obvious that it's needed to put thought into the unit tests that are created. However, the idea is to only focus on tests that cover more lines. It is easy to write lazy tests that introduce false negatives and or positives[12]. Clean code coverage is in this explanation the number of tested lines of code that test a set of requirements. The problem with only considering requirements is that it is easy to miss parts of the code that aren't considered as requirements. In other words, it's good to keep in mind the line coverage to make sure that some parts of the code aren't forgotten. If no line coverage is considered the requirements have to be extremely well defined.

## 5 Discussion

Combining different methods for testing could turn out to be the best approach. Code coverage definitely has some useful aspects. Having a proactive approach to creating tests for a project, could be a team that decides what requirements that are to be tested. From these requirements form use cases and make tests from these cases. Continuing to write unit tests when adding new functionality is probably still a good idea but striving for 100 percent code coverage might just be a waste of time. Taking false negatives and positives into account it seems bad to blindly focus on having a high code coverage.

There are cases where code coverage is a really good bonus if the tests are auto-generated. With auto-generated tests, too many tests or unnecessary tests aren't the same problems as when they are manually created. No time and effort has been put into creating these tests so they might as well be there. With this, we can guarantee that all parts of the program run without unexpected termination. However, they are still lengthening the number of lines of tests and making the tests harder to maintain in the cases where developers need to alter them. Even when the tests are auto-created, a 100 percent code coverage still seems unnecessary. At least until auto-generation of tests are so reliable that manual checks no longer are needed.

With all positives and negatives for code coverage mentioned, it's clearly a useful tool. Especially when considering how it often produces clear measurements that easily can be interpreted. It shows dead parts of the code as well as it is easy to maintain. At the same time it doesn't guarantee anything about the tests assertions. By using test coverage in parallel with code coverage we know that the requirements are met with good assertions as well as keeping the easy measurement that code coverage provides. Code is most often not bug-free and the bigger a project gets the harder it is to keep bugs away. So saying that code coverage is enough to counteract this completely would be an overstatement. However, assuming the tests written contain good assertions code coverage will help with finding potential bugs.

## 6 Conclusion

The answer to if we can rely on code coverage alone to keep the code safe and bug-free when considering false negatives and positives is a no in the general case. Although, it is in many cases a useful tool, the essay suggests pure code coverage shouldn't be the only metric to consider. Tests should be written with the intention to check requirements and validate reasonable outputs. Code coverage is a good metric for measuring how well tested a project is, just not when it's the only thing considered in a general case.



## 7 References

- [1] “Code Coverage Tools: 25 Tools for Testing in C, C++, Java,” Stackify, May 30, 2017.  
<https://stackify.com/code-coverage-tools/>  
(accessed Apr. 20, 2021).
- [2] R. Hamedy, “10 Reasons Why Code Coverage Matters,” Medium, Jul. 09, 2020.  
<https://codeburst.io/10-reasons-why-code-coverage-matters-9a6272f224ae>  
(accessed Apr. 20, 2021).
- [3] ‘Minimum Acceptable Code Coverage’.  
<https://www.bullseye.com/minimum.html>  
(accessed Apr. 20, 2021).
- [4] I. is Beautiful, “Million Lines of Code,” Information is Beautiful.  
<https://informationisbeautiful.net/visualizations/million-lines-of-code/>  
(accessed Apr. 20, 2021).
- [5] Coverage chart.  
<https://app.codecov.io/gh/TEAMMATES/teammates>  
(accessed Apr. 20, 2021).
- [6] Atlassian, ‘Introduction to Code Coverage’, Atlassian.  
<https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>  
(accessed Apr. 20, 2021).
- [7] ‘File:Coverage.png’.  
<https://commons.wikimedia.org/wiki/File:Coverage.png>  
(accessed May. 18, 2021).
- [8] ‘Cyclomatic Complexity - GeeksforGeeks’.  
<https://www.geeksforgeeks.org/cyclomatic-complexity/>  
(accessed Apr. 20, 2021).
- [9] “Test Coverage in Software Testing.”  
<https://www.guru99.com/test-coverage-in-software-testing.html>  
(accessed Apr. 20, 2021).
- [10] ‘Code Coverage vs Test Coverage: A Detailed Guide’, BrowserStack.  
<https://www.browserstack.com/guide/code-coverage-vs-test-coverage>  
(accessed Apr. 20, 2021).

- [11] ‘Why You Should Enforce 100% Code Coverage\*’, reflectoring.io, Sep. 01, 2018.  
<https://reflectoring.io/100-percent-test-coverage/>  
(accessed Apr. 20, 2021).
- [12] P. J. Sparrow, “Avoiding False Negatives: When Your Tests Pass But Production Is Broken,” Medium, Jun. 21, 2019.  
<https://medium.com/broadlume-product/>  
(accessed Apr. 20, 2021).