How much code coverage is enough?

Arvidsson, Isac
`isacarv@kth.se`

Henriksson, Andreas
`anhenri@kth.se`

April 2021

1 Introduction

1.1 What is code coverage?

Software today does not too rarely consist of several millions of lines of code. In 2017, Visual Capitalist made a list of 50 software programs, all of which consisted of over 1 million lines of code [1,2]. Even software with only around a thousand lines of code can be a struggle for a human brain when it comes to keeping track of the executed path during runtime. It's therefore important for the tests to discover many different paths through the code so various paths in the code have been explored and executed during the test phase. But, how to know then if a particular line, statement, branch, or decision is covered by the tests or which one isn't covered in any test?

Code coverage is a metric that is used for describing the level of coverage by the tests. There exist multiple different software that helps to keep track of which paths of the source code that have been executed. Some examples are mentioned in the list published on Stackify's web page [3]. The software are often more advanced than to only be able to present the level of coverage. By analyzing the code, a code coverage tool can for instance also produce a detailed report about the whole project when it comes to all the different coverage strategies(i.e line, branch, statement, and decision coverage). It can reference each bottleneck of the code and even calculate the cyclomatic complexity [4] in the code.

1.2 What does a high code coverage mean?

With its over 2,6 million hits on Google Scholar, code coverage is seen as a well-discussed metric. Also, the number of code coverage tools is a sign of its use within almost every programming language. What is the reason?

The code coverage purpose isn't defined or restricted in any way and could therefore have many purposes and meanings. Hamedy has mentioned a few purposes for code coverage in a post on codeburst [5]. Since it could be beneficial in both creating and selling the software it's already convincing that high code coverage is the goal. But, then the question about 100% code coverage is raised. How does it come that 100% code coverage is never the goal in projects? Are the benefits reduced above a certain percentage of code coverage? Is it possible to distinguish the perfect limit of sufficient code coverage?

In the following section code coverage with its advantage and counter-arguments, for why 100% isn't the goal, is discussed. The essay ends with a conclusion section where takeaways from the findings in the discussion are presented.

2 Discussion

2.1 Arguments for high code coverage

High coverage is subjective and highly connected with unit tests [6]. To have a 100% code coverage means that the unit tests are accessing all parts of the source code. Code coverage is a quantitative metric that helps developers by checking that all parts of the code are reached and work properly. It's also a useful tool for a developer to guarantee that a certain unit test reaches the intended scopes.

This is a very useful metric to have to keep 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.

Even though 100% code coverage doesn't mean that there are no bugs in the code, it is possible to enforce tests such as they are written with purpose. In this article [7], this type of code coverage is called "clean code coverage". It might seem obvious that it is needed to put thought into the unit tests that are created. However, the idea is to focus only on tests that cover more lines. It is easy to write lazy tests that introduce false negatives and or positives. 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 is 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.

2.2 Counter arguments for high code coverage

A high code coverage comes with a lot of good things as ensuring that the code runs without termination [8]. The question to ask is if it is worth all the time and effort it takes to keep a high code coverage. An alternative to focusing on having a high code coverage is to use test coverage as a metric [7]. 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.

Code coverage is very limited to unit testing and doesn't always provide a fair metric when sometimes more coverage does not have to be better. Some parts of the code could be viewed as unnecessary to reach. Another drawback with code coverage is that it is 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. However, as discussed before code coverage does output a clear and often comparable measurement that test coverage does not.

Only striving for a high code coverage does not necessarily mean that the code is safe and well tested [7]. It only says that a high number of lines are covered by accumulating all unit tests. It does not say anything about whether the output is correct or enough illegal inputs are being tested. However, having a high code coverage is still a good thing as long as it is not the only metric that is considered.

2.3 Code coverage in auto-generated tests

In automated test generation, code coverage comes in very handy in many ways. An unsophisticated software that auto-generates tests for a specific source code could in fact be built with almost only code coverage in mind. "Make sure to optimize the code coverage in any way", could be the instruction to the software. The software that auto generates test cases could then easily produce a silly amount of test cases without a single assertion and then covers 100% of the code. Note that it is not always promised to get 100% because some part of the code could be implemented in an unreachable way for the computer. But even if the program succeeds in producing tests that cover every single line of the source code, what does it say to us? Since there are no assertions, it tells us that the program runs without unwanted termination or errors. That the program runs as intended, produces the correct actions, and outputs the correct value are completely ignored with tests generated in this way.

2.4 How much code coverage?

With all cons and counterarguments for code coverage mentioned, it's still 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. Having a good amount of code coverage is a good thing and the question to be answered is, how much is needed? Even though there is no definite answer to this question there exists a consensus that 100% code coverage is unnecessary. Some sources like [10] have specific numbers or ranges on what amount of code coverage is reasonable to aim for. As previously discussed, there are many flaws with relying on code coverage as a metric. With a certain amount of code coverage, it should be expected to have covered the majority of the serious bugs that could come up. But, that is the indirect result of high code coverage and not the high code coverage itself.

For instance, let say that the aim is around 80% as in [10]. Which 80% of the code that is chosen to be cover is drastically affecting the reliability of the tests. Therefore a recommendation of a specific number or range of code coverage is rarely stated because of the hardness of augmenting for that specific value. Instead, using the code coverage metric as a probabilistic measurement is an alternative way of usage. With high code coverage, the probability of covering bugs is increasing but not ensuring anything.

3 Conclusion

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.

Answering exactly how much code coverage one should aim for is hard. However, as some of the sources mention a good middle ground is around 80%. To conclude, 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 is the only thing considered.

4 References

[1] I. is Beautiful, "Million Lines of Code," Information is Beautiful. `https://informationisbeautiful.net/visualizations/million-lines-of-code/` (accessed Apr. 20, 2021).

[2] Atlassian, 'Introduction to Code Coverage', Atlassian. `https://www.atlassian.com/continuous-delivery/software-testing/code-coverage` (accessed Apr. 20, 2021).

[3] "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).

[4] 'Cyclomatic Complexity - GeeksforGeeks'. `https://www.geeksforgeeks.org/cyclomatic-complexity/(accessedApr. 20,2021).`

[5] 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).

[6] 'Code Coverage vs Test Coverage: A Detailed Guide', BrowserStack. `https://www.browserstack.com/guide/code-coverage-vs-test-coverage` (accessed Apr. 20, 2021).

[7] "Test Coverage in Software Testing." `https://www.guru99.com/test-coverage-in-software-testing.html` (accessed Apr. 20, 2021).

[8] 'Why You Should Enforce 100% Code Coverage*', reflectoring.io, Sep. 01, 2018. `https://reflectoring.io/100-percent-test-coverage/` (accessed Apr. 20, 2021).

[9] P. J. Sparrow, "Avoiding False Negatives: When Your Tests Pass But Production Is Broken," Medium, Jun. 21, 2019. `https://medium.com/broadlume-product/avoiding-false-negatives-when-your-tests-pass-but` (accessed Apr. 20, 2021).

[10] 'Minimum Acceptable Code Coverage'. `https://www.bullseye.com/minimum.html` (accessed Apr. 20, 2021).