# CS3012 Report: Measuring Software Engineering

It can be an intrinsically challenging undertaking to measure the work rate and efficiency of a software engineer. The intricate nature of new age software systems results in a similar intricacy in developing solutions for particular projects. Thus, it is now a requirement for software engineers to not only have a suitably broad programming skillset, but to also have an extensive knowledge of the project they're engaged with. They must put forth a combination of the skillset and knowledge in every step of the implementation of these projects while maintaining co-operative and productive relationships with team-mates to ensure no roadblocks are hit at critical stages of developing the software systems.

So this poses the question, how does one expect to compare and/or measure these somewhat intangible requirements? I will attempt to answer this question under the headings of four main talking points; Measurable Data, Computational Platforms, Algorithmic Approaches, and the Ethical concerns involved in the process.

## *Measurable Data*

There are various types of data that can be used to measure the productivity of a software engineer, each with its own pro's and/or con's.

Firstly, a more simplistic measurement, but heavily flawed, includes measuring the lines of code written. This methodology is very inaccurate as it measures the productivity of a software engineer by looking at the number of lines he/she has written, therefore someone who has written a lesser amount of lines is considered not as productive. I will provide examples as to why this is not the case and how inaccurate a measurement it is.

Measuring productivity in terms of an engineer's lines of code creates an environment where software engineers are trying to maximise the number of lines in their program to look better than their peers. However, a program with less lines and the same functionality will probably look better and in many cases, is also likely to be more efficient that the longer program. Developers in some instances may be recklessly adding unnecessary methods or variable declarations to their code which could impact on the overall functionality of the software. Another aspect that is flawed in using this methodology is the comparison between high and low level languages. Imagine this; a manager would like to measure the productivity of his/her software engineers, and is responsible for having the same program implemented in two different languages, one being a higher-level language, and the other being a low level. The lines of code written by the engineer programming in the low-level language is 5,000 more than that of the other developer coding in the high-level language. Instantly, the issue with using the lines-of-code methodology is apparent. It would be absolutely absurd to think that the programmer coding in the low-level language would be more productive, just because the language they're using lacks the functionality of a higher-level language, and thus requires thousands of lines more of code to reach the same software functionality that could be done in a fraction of the lines in the high-level language. Similarly, this argument can be made if the case was "the less lines of code, the more productive the engineer", which just means we must look to alternative methodologies to use to measure the productivity of software engineers (Files.ifi.uzh.ch, 2018).

Secondly, code churn is a methodology that some may consider useful in measuring the productivity of software engineers. Many times in developing software systems, developers may scrap a lot of their code and rewrite a significant part of their program. However, if a manager is logging the number of lines of code committed by each developer over certain time period, it may inaccurately depict that one software engineer is a lot more productive than another, basing it on the amount of lines of code they have committed. This can be resolved if the manager decides to monitor the code churn, which monitors how much of an engineer's code has been scrapped in the development process, and allows the manager to see how much of the code that the engineer delivered was actually productive and contributed to the final software system. This is a more effective way of measuring the productivity of a software engineer (GitPrime, 2018).

Another form of measurable data would be the code coverage of the unit tests written by software engineers. In some respects, this can prove a useful way to gauge the productivity of software engineers. How rigorous a developer was in testing their code's functionality for errors is shown in the code coverage of their unit tests, the optimum coverage being 100%. A 100% code coverage score means that every line of code and every condition in their conditional statements has been executed at least once. A manager can compare how meticulous each engineer in a team was by looking at the tests they have written and make judgements on their work rate and efficiency based on that. However, there are various downfalls in using code coverage as a sole way of gauging productivity. Although an engineer may receive a score of 100% code coverage with their tests, their tests may be faulty themselves. This could be the case if developers write tests that are made to pass with their code and not actually made with the intention of testing the desired functionality of the software they're developing. This must be kept in mind by managers when looking at code coverage as one of the ways to scale how productive a software engineer is (Confluence.atlassian.com, 2018).

Finally, the cyclomatic complexity of a program is a method that is sometimes used by managers to measure the productivity of a software engineer. But what exactly is cyclomatic complexity? Cyclomatic complexity is a term commonly thrown around in the development community, and in lay-man's terms, it looks at how unnecessarily complicated a developer's program might be. As we all know, there are many different ways to achieve different solutions to various problems, but when it comes to software development, it is generally accepted that if a program is over-complex it is more susceptible to bug-problems and errors. Measuring cyclomatic complexity revolves around two concepts that are familiar to most software engineers, nodes and edges, and it is generally accepted that the cyclomatic complexity measure of a program can be derived from the formula $CC = (E-N) +2$, where $E$ is the number of Edges, and $N$ is the number of Nodes. However, this measure is not perfect, as it is not proven that a higher cyclomatic complexity leads to an error prone program, so a bit of perspective is advised to managers who wish to use this as a metric (Softwaretestinghelp.com, 2018).

## *Computational Platforms*

Above, I spoke about the different types of data that can be used as a methodology for measuring software engineering, and now I will speak about the various computational platforms available that can be used to carry out the various measurements.

Firstly, a very common platform used by managers when overseeing software engineers is GitHub, or BitBucket. These platforms are free to use for the most part and present useful descriptive data about developers and their programs. These platforms enable managers to easily monitor productivity of software engineers as they keep track of commits made and provide backups of code written. They also support visualisation of different types of data regarding the developer through various useful charts and graphs, and through the GitHub API, managers can make data visualisations of their own if GitHub doesn't provide the specific information they're looking for as standard. As regards being a platform for measuring the previously discussed data, GitHub keeps track of lines of code that have been both added and deleted throughout various commits, and can therefore assist managers in measuring through the lines-of-code methodology and the code churn methodology. It also allows the manager to measure the overall performance of the software engineers (SearchITOperations, 2018).

The next computational platform that assists in measuring software engineering that I will be discussing is Hackystat. Hackystat is an open source platform which allows for the analysis, interpretation, and visualisation of data with regard to the software engineering process. For managers, it is an extremely useful framework as it can be utilised as a tool that supports the collection and analysis of metadata and information that ensures good project management and high productivity for software engineers. When using Hackystat, it monitors the environment on which the code is written and preserves collected data on its server. With this data, it then generates various visualisations such as charts and graphs for different aspects of the software engineering process including the number of commits, details about each commit, lines of code written, as well as code churn. Again, this information can be used as mentioned in the previous section to help gauge the work rate and efficiency of software engineers. Hackystat also provides for viewing these stats for individual engineers, or for the team of developers as a whole (YouTube, 2018).

There exists numerous plugins available for various IDEs that can measure code coverage for unit testing. In Eclipse, the most used is a plugin extension called EclEmma that allows for the fluid display of code coverage. It displays the percentage you have covered, as well as highlighting the different lines in green or red to show which lines have or have not been executed by the unit testing. This helps developers to write better tests that reach higher code coverage, and also allows managers to measure the progress of software engineers and how reliable their software can be considered depending on how well it has been tested. Similar tools exist in alternative environments, such as Visual Studio from Microsoft, which has a code coverage tool built-in to its functionality.

Finally, Metriculator is a tool from the Eclipse marketplace that creates static software metrics based on source code, for example, cyclomatic complexity. This is extremely helpful as it also makes assessments on the standard of the code and can be useful in fixing bugs and creating better software.

## *Algorithmic Approaches*

There are many ways one can look at measuring software engineering when it comes to algorithmic approaches. Some of the algorithms used to calculate the metrics I've mentioned range from simple, to more complex.

In the 'Measurable Data' section of this report, I discussed the approach of cyclomatic complexity, which allows one to measure the unnecessary level of complicated code in a software engineers program. The algorithm for this was developed by software engineer Thomas J. McCabe in 1976 in order to address questions that were common when developing software; Is the program testable, understood by everyone, and reliable enough? The algorithm initially set out by McCabe is made around the concept of a directed graph, which contains Nodes (N), Edges (E), and number of connected components (P). He defined the cyclomatic complexity (M) as

M=E-N+2P. The actual algorithm behind this formula varies slightly for different programs, but generally includes starting with an initial M of 1, and then for each loop, or condition in an 'if' statement, you add 1 to M. Bear in mind this is a very simplistic approach, and is not the case for larger and more in depth programs where tools such as Metriculator is necessary (Softwaretestinghelp.com, 2018).

Secondly, the algorithm behind code churning is much simpler. In the majority of cases, software engineers will be utilising platforms such as GitHub when working on projects. This makes the algorithm for calculating he code churn a simple addition algorithmic equation. You simply add up [Lines Added] + [Lines Deleted] + [Lines Modified] to derive the code churn measure for a particular commit. All of these statistics involved in the equation are readily available and clear in the repository (Docs.microsoft.com, 2018).

## *Ethical Concerns*

As in many aspects of working environments, ethical concerns are always present, and must be addressed if conflict or turmoil is to be avoided. In measuring software engineering, the selection of an unsuitable methodology such as the lines-of-code method that I previously outlined can cause a particular software engineer distress if they are being unjustly reprimanded by management for lack of productivity, when it is the measure that is at fault. Also, software engineers must be given their own degree of freedom when working and not feel like their personal space is being invaded by superiors constantly monitoring their work. It must be done in a way that is mutually agreed and doesn't impede on anyone's right to privacy. If managers wish to use or store any information about the software engineer that they feel would help them measure or understand their productivity, they must make sure that they have permission from the engineer and have precautions taken to prevent data leaks. Finally, managers must ensure that the focus on the use of metrics for their software engineers isn't so severe that they are caused mental stress and leads to counter-intuitive behaviour. Such behaviour may include the engineers focusing so much on metrics, that they lose focus on solving the actual task at hand, resulting in lower quality software systems, which is the opposite of what managers are attempting to achieve in the first place.

# Bibliography

Files.ifi.uzh.ch. (2018). [online] Available at: https://files.ifi.uzh.ch/rerg/amadeus/teaching/seminars/seminar_ws0203/Seminar_3.pdf [Accessed 21 Nov. 2018].

GitPrime. (2018). Why Code Churn Matters | GitPrime Blog. [online] Available at: https://blog.gitprime.com/why-code-churn-matters/ [Accessed 21 Nov. 2018].

Confluence.atlassian.com. (2018). About Code Coverage - Atlassian Documentation. [online] Available at: https://confluence.atlassian.com/clover/about-code-coverage-71599496.html [Accessed 21 Nov. 2018].

Softwaretestinghelp.com. (2018). What is Cyclomatic Complexity - Learn with an Example. [online] Available at: https://www.softwaretestinghelp.com/cyclomatic-complexity/ [Accessed 21 Nov. 2018].

SearchITOperations. (2018). What is GitHub? - Definition from WhatIs.com. [online] Available at: https://searchitoperations.techtarget.com/definition/GitHub [Accessed 21 Nov. 2018].

YouTube. (2018). Hackystat In A Nutshell. [online] Available at: https://www.youtube.com/watch?v=NrLFIpm0wps [Accessed 21 Nov. 2018].

Docs.microsoft.com. (2018). *Analyze and report on code churn and code coverage - TFS*. [online] Available at: https://docs.microsoft.com/enus/azure/devops/report/sql-reports/perspective-code-analyze-report-code-churn-coverage?view=tfs-2018 [Accessed 21 Nov. 2018].