A report on the essence of software measurement

Xander De Jaegere - 18341938

1. Abstract:

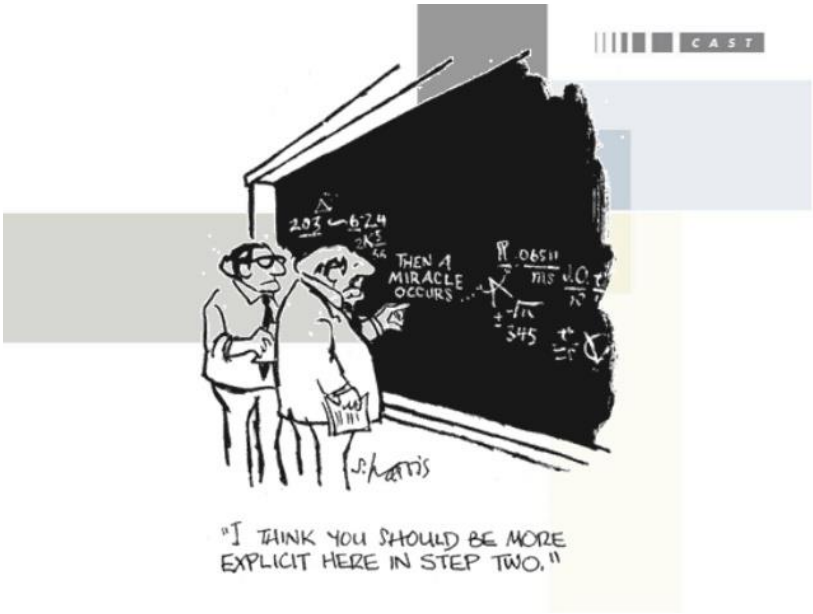
In this report I will talk about the reason for software measurement. The history, the present and the possible future of measuring and quantifying of software engineers and their projects. After this I'll top it off with an ethical standpoint of my own about the task of measuring and what is acceptable and what isn't.

2. Introduction:

To begin talking about software measurement we first have to understand what it is and why it is done. At first you would think this is an easy definition by simply stating software measurement is the act of quantifying software some programmers has written that allows us to objectively rank some code from other codes to see which one is better. While this is technically correct, once you start thinking about how you measure software against each other you'll see that it is no immediate easy feat and that there are multiple ways each with its own pros and cons.

Why we should measure software is also a debated topic. Software engineers don't like to be measured for their work as they fear this measurement might misrepresent their abilities or undermine their future at a certain company. This is why so many of the software engineers like to act in a certain mysticism about their job. Saying that it is impossible to give accurate measurements about their code as every piece of code is complex and different from the other lines of code previously written.

The cartoon below[1] describes perfectly what some software engineers would like you to believe. Of course this isn't true and measuring, while more difficult than for some other fields, is certainly achievable. The reason why measuring someone's work in the company is important should be obvious. Measuring how much work someone does allows the management to better supervise their employees and make decisions based on that work. This improves the total efficiency of the business and hence improves the worth of the company. But measuring someone is not all about that one specific person. It is to see how and why some people are more proficient or efficient than others and to learn the behaviors to other people working in the same functions or teams.



[1] Describing the process as magic
https://www.slideshare.net/it_cisq/software-assurance-forum-march-2010-cisq-software-quality-measurement-24015494

Having a quantitative measurement allows for an engineer to know if he's doing well or if he should step up his efforts. It is a tool that you have to learn from. While it's natural to be scared of falling behind or being shown that you're not doing the best is painful. It is a good way to learn from your mistakes. Failure isn't the worst thing to happen it is just the first step to success. Hence it is an important topic to delve into.

3. Data measurement

Now that we have talked about what software measurement is and why we should participate in it, it is time to discuss the specifics of how to actually measure software. There are many ways of measuring software and that's the reason it is such a debated topic. There is no standard platform that every company uses to evaluate their engineers. While a standard measurement system may be very useful to compare different projects and to estimate the success of future projects. Every project is different and the companies themselves may want different statistics about their employees work. Some companies may care for speed and speed only. So the faster software runs the better. Others may want the robustness of code and are willing to sacrifice the efficiency and simplicity of their programs, assuming that we're comparing projects which goals are fundamentally different. The first one may be a very specialized program while the other has to take many other variables into account. If you compare two programs which accomplish the same goal then the faster one is mostly better.

The first way to measure code is to only take into account the data you can gather from the source code only. These methods include the LOC method (lines of code), cyclomatic

complexity as first described by Thomas J. McCabe, code coverage, and coupling and cohesion.

The second type of data to measure doesn't look at only the source code. It also looks at the frequency of changes, the number of commits a developer makes, the number of bugs a commit has fixed and the amount of time the software engineer has spent on the commit.

The last type of data takes into account the work method of developers, it doesn't look at what they develop but how they develop it. This data can be very helpful to discover why some engineers are 10 times more productive than others. While this data may be considered essential to some companies or management. It is worth the time considering the ethics of this kind of supervising and will be discussed at the end of the essay.

3.1 LOC

The most obvious way to measure software from only the source code is simply counting the lines of code. This way of analyzing software is the most useless measurement you could use as It provides to nothing. Comparing a codebase of 500 lines of code to one of 1000 that arrives at the same goal doesn't mean the one of a 1000 lines is better at all. In contrary most of the time code that is written with less lines of code is more concise and less complex therefore less likely to introduce bugs in the software. Also it is extremely easy to game the system as a software engineer. Instead of using ternary operators one can use if and else statements.

Even worse, you can intentionally not use built in functions and design bad code. Just to get more lines of code like I did in the example[2] below. Assuming you have an array of numbers where all the numbers need to be incremented, if you know the size of the array will always stay the same you can game the system by not using a for loop this will give you 8 lines of code more than if you use the built in for method. Not only is this unnecessary this will also make your code more rigid and since this obligates you to always have that array as size 10.

```
# list that holds data
list_of_10_items = [x + 1 for x in range(10)]


def incrementer(integer):
    return integer + 1


list_of_10_items[0] = incrementer(list_of_10_items[0])
list_of_10_items[1] = incrementer(list_of_10_items[1])
list_of_10_items[2] = incrementer(list_of_10_items[2])
list_of_10_items[3] = incrementer(list_of_10_items[3])
list_of_10_items[4] = incrementer(list_of_10_items[4])
list_of_10_items[5] = incrementer(list_of_10_items[5])
list_of_10_items[6] = incrementer(list_of_10_items[6])
list_of_10_items[7] = incrementer(list_of_10_items[7])
list_of_10_items[8] = incrementer(list_of_10_items[8])
list_of_10_items[9] = incrementer(list_of_10_items[9])
```

```
#list that holds data
list_of_10_items = [x +1 for x in range(10)]

#loop to increment all data
for x in range(list_of_10_items):
    list_of_10_items[x] +=1
```

[2] Inefficient code to game the LOC system. Is standard code

3.2 Cyclomatic complexity

As previously mentioned measuring software by checking it's cyclomatic complexity was first suggested by Thomas J. McCabe in his paper "*A complexity measure 1976".* It is the idea that we can define the complexity of a method or function by checking the amount of branching paths from that method. Thomas described the complexity of modules but we see that as methods now. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command.

The complexity is measured by the following formula $V(G) = e - n + 2p$.

$$e = the\ number\ of\ edges\ in\ the\ graph$$

$$n = the\ number\ of\ nodes\ in\ the\ graph$$

$$p = \ the\ number\ of\ components\ in\ the\ graph$$

The difficulty with this method is that there also is no standard way of implementing it. Which means that by using 2 different platforms, that supposedly give you an accurate representation of this measurement based on your code, you can get 2 different numerical values. This is because Thomas wasn't extremely clear on how to count if statements . Some platforms consider an if statement with multiple conditions as one branch while others count every condition as a separate branch.

Finally the value tries to give an objective result to a subjective question. Complexity in itself is not just the amount of branching paths, it's also what those branches entail. If I write a method that returns the string of a month depending on the number that you put in, this might have a complexity of 12 while the code itself is very understandable and changing it will never be complex. Opposed to if you write a method with convoluted conditions, this might give you a lower complexity number but the method can break more easily by changing something in the code itself.

3.3 Code coverage

This measurement represents the amount of lines that are tested by the developer. It is a very useful way to check if you haven't missed out on testing some code and if your test suite is complete enough. This measurement is used for test oriented development. It can show you if there is a developer that systematically doesn't test his code enough. Having 100% coverage means that every line of code in your program is run at least once in the test suite.

But using this measurement you inherently assume that the test suite written provides coverage that is useful and can determine if there are problems with the code itself.

Developers can game the system by writing tests that just trivially go through the entire code without actually testing anything.

3.4 Coupling and cohesion

Coupling is the indication of how much individual units of code are dependent on other units. Two units with low coupling means that they don't interact with each other except for maybe passing information to each other with a parameter list. While high coupling may mean that one unit has to wait on the other unit for control statements and can't keep running if the other unit doesn't give a response.

From a maintenance perspective, high coupling means more time spent on figuring out all the modules to understand one single module. It is not possible to understand one module by simply looking at the code of that unit only. For example if you look at the python library pyGithub which acts as a wrapper for communication with the Github API v3 you can't understand the main class Github by just looking at the code of that class. You have to look at the many classes it uses to stay working. Which means that it's highly coupled with other classes in the library

Measuring how well the different modules work together can be represented as the cohesion of the software. Software where each unit has its own function and there are no redundant methods and units will have a high cohesion. Having high cohesion make software maintenance and upgradeability easier in the future than software with low cohesion.

3.5 Code churn

Code churn is the amount of times a certain file or lines of code has been altered. This data is gathered by looking at all the commits and looking at the code altered. High code churn is a sign of code with either a high degree of bugs or code that isn't easily understandable.

The code churn of a specific developer is interesting to investigate because it shows the development process of that software engineer. If his/her code churn tends to spike more and more nearing a deadline that means their code grows more volatile near the end. This is the opposite of what you want. You want code to stabilize near the release date. High code churn can be seen as an omen for many code defects in the lifecycle of the code.

3.6 Happy-productive worker thesis

One of the hypothesis surrounding workers productivity is their general happiness in life and satisfaction with their jobs. So a company has a benefit assuring people are happy working there. In a study[1] it was found that there is a high correlation between the productivity and the subjective happiness level.

Since this study was meant as a general hypothesis and not solemnly for one field of work we can also use this as a way of measuring software engineers. When it is noticed by management that someone's productivity is declining. They can consider this as a possible lack of happiness or satisfaction with their work and try to figure out what is causing this discontent by talking with the developer in question. Dependent on the information gathered by this talk, action can be taken to resolve the issues.

4. Computational platforms available

Having discussed different ways to measure a software engineer I'll take a look at the different platforms available now for companies to gather these measurements. Since having people work on implementing those measures in a company can be more expensive than it is worth or they don't have the resources available to tackle such a project.

One of the first structured platform used was the Personal Software Process (PSP). It was created by Watts Humphrey to apply the underlying principles of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to the software development practices of a single developer[2].

PSP was developed as a disciplined data driven procedure that would help engineers to improve the quality of their code, improve estimating and planning skills, help keep their commitments and reduce the total of defects in their projects.

The way this process works is that a software engineer would fill in their own form of what they did during a session, what they worked on and how long it took. This would then be gathered and be used as a way to give feedback of things that could be improved. But the system itself was considered to be too slow. It negatively impacted the productivity of engineers and the data gathered was considered to be too subjective. Since you have to fill in the data of yourself by yourself.

Other platforms available are Hackystat, Github, the LEAP toolkit, Codescene , Semmle, code climate and many others. These platforms allow companies to see multiple metrics mostly based on commit history and source code.

4.1 Hackystat

Hackystat is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data. The Hackystat Framework supports three software development communities:

Hackystat users typically attach software 'sensors' to their development tools, which unobtrusively collect and send "raw" data about development to a web service called the Hackystat SensorBase for storage.

The SensorBase repository can be queried by other web services to form higher level abstractions of this raw data, and/or integrate it with other internet-based communication or coordination mechanisms, and/or generate visualizations of the raw data, abstractions, or annotations.

Hackystat services are designed to co-exist and complement other components in the "cloud" of internet information systems and services available for modern software development.

One of the biggest advantages of Hackystat is it automatic data gathering after you have set up all the sensors. Software engineers don't have to fill in any information manually anymore. Which allows them to work in one continuous state of productivity. Without having to worry about the interruptions of noting what they did during the last interval of work.

In a study done by game developer magazine where they recorded 10000 programming sessions they noted that it takes 25-30 minutes to restart editing and making code after an interruption. This shows the importance of not having to worry about manually entering data required by the supervisors

4.2 code climate

Code climate is an open platform created by Bryan Helmkamp. It is integrated with Github and is able to analyze many different languages like python, java , ruby. It is one of the most used platforms as it deals with 2 billion lines of code daily (self-proclaimed) and works with over 100000 projects. They deliver metrics such as code churn, maintainability of code, coverage statistics and much more. This allows for programmers to work in a streamlined manner where they receive instant feedback about their programs.

Many of the platforms available give similar types of feedback and metrics. So when deciding which one to use the reasons may tend to which platform is more responsive and flexible than the rest. But the similarity of these platforms doesn't mean that there is a standard way of measuring software engineers.

5. Algorithmic approaches

5.1 Maintainability index

This algorithm computes a number that represents how easy it is to maintain the code you have written. The number goes from 0 to 100 where higher means better. It takes into account the cyclomatic complexity as described in paragraph 3.2, the SLOC (source lines of code), comment lines, and the halstead volume. This algorithm is used by multiple environments like virtual studios

The formula states : $MI = 171 - 5.2 * log2(V) - 0.23 * G - 16.2 * log2(SLOC) + 50 * sin(sqrt(2.4 * CM))$

Where V is the halstead volume. G is the cyclomatic complexity, and CM the percent of comment lines. Since this algorithm is based on multiple metrics which are mostly based on lines of code. The use is limited. It just isn't possible to make on all encompassing metric that tells you everything you need to know about the program.

6. ETHICS

Of course there are ethical considerations to be considered when measuring software engineers. Depending on the type of measurement you use in a company it can infringe upon the privacy of the developer. Typical metrics measured by software platforms like Hackystat aren't invasive and give valuable data to the company so that it can better supervise you and potentially help the developer realize where he is making common mistakes.

But systems where you record the screen of a developer 100% of the time can be seen as a huge breach of privacy. The definition of privacy is a state where you aren't being observed or disturbed by other people (oxford dictionary en). So the act of recording ones screen can immediately be seen as a having no privacy. Even if the company can gather valuable information like good programming habits. Eye trackers to see what percentage of the time you're focused on the screen versus looking away also falls in this category.

Another example of a practice I would consider unethical would be having an active badge which has a GPS and cameras with face recognition in it to record where you are and who you talk with during breaks. This information might be useful to learn further habits during the work day, the fact that you're conversations are being monitored by these systems is an infraction on your privacy.

Having some of these implementations in place can negatively impact the productivity on the workplace of the code of conduct. When employees are being supervised 100% of the time they can feel more stress on their shoulders and therefore start being less productive or they can burn themselves out.

As an example when I worked at a summer job in a magazine. The company started to focus more on the daily quota every employee had to reach. This led to people taking dangerous shortcuts to work more efficiently.

This means that companies have to consider carefully which measurements they want to take and how they are going to store this data. They also have to communicate openly with their employees how they are going to supervise them.

7. Conclusion:

Measuring the work and productivity of a software engineer is an inherently complex issue and has yet to be standardized. There are multiple metrics and platforms available and there is no one best option. The current platforms are working towards a more unobtrusive way of checking data and are trying to get the most useful metrics out of your software.

Bibliography and references

1. Zelenski, John & A. Murphy, Steven & A. Jenkins, David. (2008). The Happy-Productive Worker Thesis Revisited. Journal of Happiness Studies. 9. 521-537. 10.1007/s10902-008-9087-4.
2. "Using a defined and measured Personal Software Process" by Watts S. Humphrey, published in IEEE Software, May 1996.
3. J. Gibson, 2018, “Personal Software Process” [online] Available at: http://www-public.imtbs-tsp.eu/~gibson/Teaching/CSC5524/CSC5524-PSP.pdf
4. Cyclomatic complexity [online] Available at: https://en.wikipedia.org/wiki/Cyclomatic_complexity
5. Hackystat [online] Available at: https://hackystat.github.io/
6. T. Kuipers, J. Visser, 2007, ”Maintainability index revisited” [online] Available at: https://www.cs.vu.nl/csmr2007/workshops/SQM07_paper3.pdf
7. Lee, Ming-Chang. (2013). Software measurement and software metrics in software quality. International Journal of software engineering and its application. 7. 15-33.
8. S. Lowe, “9 metrics that can make a difference to today’s software development teams” [online] Available at: https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams
9. P. Johnson, 2013, “Searching under the streetlight for useful software analytic” [online] available at: https://ieeexplore.ieee.org/document/6509376
10. A. Carlson, “Coupling and Cohesion” [online] available at : https://courses.cs.washington.edu/courses/cse403/96sp/coupling-cohesion.html
11. H. Kou, X. Xu, “ Most Active File Measurement Validation in Hackystat “ [online] Available at: https://pdfs.semanticscholar.org/d33c/76e89c8db632be691ea8d655c8f3afb5882d.pdf
12. https://en.oxforddictionaries.com/definition/privacy