Measurement of Software Engineering

Student Name: Ryan Dowling

Student Number: 17321943

Introduction:

Why measure software engineering? With the ever increasing number of software engineers in our societies, employers need a way to distinguish between engineers who are the most productive and those who are not. This goal of this report is to outline the ways in which we can measure software engineering, the platforms used to measure these engineers on, the algorithms used in measuring and will conclude by discussing the ethics concerns surrounding such measurements.

Ways in which software engineering can be measured:

There is a wide variety of ways in which the software engineering process can be measured. There are a multitude of different things that affect engineers and the way that they work, and as such there is no standardised process or procedure. However, there are many metrics and plenty of measurable data that when understood, can give a clearer insight into the productivity and efficiency of an engineer.

Lines of Code:

Lines of code written by a software engineer is often brought up as a way to measure performance, perhaps because it is one of the easiest ways to distinguish between two or more engineers. If one person has written more code than the other, then clearly he/she is better, right? From someone who knows nothing about programming, this might be true. If you've written more, your code should surely do more, which makes your code better. Of course, we know that this isn't the case at all. In fact, using this as a measurement for productivity could be very devastating.

Rewarding engineers solely based on the amount of code they write could lead to overly bloated code. Why bother writing 100 lines to add two numbers together when you could do it in 1? On the flip side of the coin, rewarding engineers for writing a tiny amount of code is detrimental too. A program that has very few lines of code is no good if it's illegible and unmaintainable by other engineers. It'd be much better to write more code for the sake of clarity.
In general, lines of code is a useless metric to base someone's performance on, and should be avoided if at all possible.

Completion Time:

Another metric often talked about to measure productivity is completion time. Surely an engineer that completes their given tasks faster than others must be more productive? Not necessarily. This metric does not take into consideration the complexity that different projects may have. One engineer might be designing a whole database, the other might only have to develop one function in a program. The two are completely different tasks, and as such completion time has no relevance in how productive the two are. This extends beyond individual work and is an issue for teams as well. Certain teams might be unwilling to work with others if it means completion time of their own work will be faster. This could lead to hostile work environments and would reduce work productivity as a result. Surprisingly however, in a poll made on Twitter asking for developers' opinions, 15.3% thought that speed was an important metric. This goes to show that despite the negatives associated with such a metric, people still regard it as an important one for measuring productivity.

Ticket Handling:

Ticket handling is another way in which software engineering can be measured, and in my opinion is one of the better ones, as you are basing productivity on how much an engineer helps to improve the project and business as a whole. There are a multitude of different ticket organisation methods, e.g. Kanban boards. The benefit of these is that it allows other departments to submit tasks that they want the technology department to handle. These submitted tasks can then be ranked or weighted as a way to compare the productivity of software engineers or teams.

For example, a team could be working on a simple task posted to a board, which could be ranked as "Easy". However, these measurements are not standardised across the whole organisation, so an "Easy" task for one engineering team could be designated as "Hard" for another team. This adds a scaling degree of difficulty to each task, which ensures that each team is working efficiently at their own maximum capacity.

In my opinion, this is a good way to measure software engineering, as you are both encouraging priority of tasks that the business needs completed, as well as giving engineers a sense of pride and accomplishment for completing more “difficult” tasks.

The only caveat I can see with such a system is that the business needs someone who is capable of organising and prioritising tickets, rather than having the engineers themselves gathering tasks from each individual department that needs a task completed.

Computational platforms used to measure data:

Personal Software Process:

Computational Platforms for measuring engineering began with Personal Software Process, or PSP for short. PSP is a metric gathering and analysing data from the software engineering process. By logging and managing data of the software engineer, it enables better analysis of the work, in turn helping them to improve and maintain good practices and products.

A big benefit of PSP is that it is low cost and both flexible and customisable. A simple spreadsheet can be created to suit the requirements of the project you are working on. The main issue however is that this approach is all manual. data must be logged yourself and calculations must be done by the developer. This can cause issues with regards to data quality, as human error is always a problem. This in turn makes the analytics fragile, and the conclusions drawn may sometimes be incorrect.

Hackystat:

Hackystat was a third generation PSP data collection tool, that aimed to overcome the issues of manually logging and analysing data. Originally created by the University of Hawaii, they focused on developing ways to collect data with a minimal amount of overhead. This was successfully achieved with data collection through sensors attached to development tools. This data collection occurred automatically, making it a lot easier to gather the data, as well as being less intrusive. It utilises some of the previously mentioned metrics, such as number of lines of code. This gives a broad overview of the project, and detailed insight into progress.

Hackystat is individual in nature, as it analyses the performance of an individual software engineer, rather than team. This may be beneficial when comparing two engineers against each other, but less useful if you wish to compare two teams.

PROM:

PRO Metrics (PROM), is similar to Hackystat in the fact that it is an automated tool for gathering and analysing software metrics and data. Like PSP, it encourages software engineers to refine and improve their work with individual analysis. It also adds on tools for business management. A problem that some users had while using Hackystat was that you could visualise each developer's DevTime trend, despite some wanting to keep this private. In PROM, individuals have the ability to view their own data but also grant access to another user for collaboration and assistance purposes. There is no access to an individual's data from a manager's perspective, however there is a whole project summary available to them. This allows managers to still acquire useful information without compromising the privacy of the engineers.

These means of monitoring progress and productivity are further enhanced by the adoption of game theory. Gamification is a powerful tool that has been proven to promote motivation within business, and in doing so directly increasing productivity. More motivated employees generally produce better quality work. The gamification of project based goals ties into creating motivation, where the engineer *wants* to complete a task. This form of motivation makes the engineer more likely to persist and work through difficult obstacles or problems encountered in the project, because he/she is treating it like a game. This is achieved through an engagement loop or cycle. The computational platforms fit perfectly into the cycle, delivering feedback to the engineer which drives motivation.

Algorithms used to measure data:

In the previous section, I gave you some examples of tools that can be used to collect measurable data in regards to software engineering productivity. However, that's all it is, data. What good is lines of code written if you've nothing to do with it? In this section I will discuss some algorithms that can be used with the data collected, and how these can be used to measure productivity.

Function Point Analysis:

Function points are units of measurement that are used to express the amount of business functionality an information system provides to a user. These can be really helpful in calculating the size of computer software, estimating costs, setting functional requirements and of course, measuring quality. Functional Point Analysis is done from a user's point of view and it consists of breaking the systems into five characteristics:

1. External inputs
2. External outputs
3. Logical Internal Files
4. External Interface Files
5. External Inquiries

When all five of these characteristics have been identified, a complexity rating of the functional value for the end-user is determined. Some of the factors considered are:

- Will the application use data communication?
- Are data or functions distributed?
- Will there be on-line data entry?

The function points are then added and considered "unadjusted". An analyst will then assign weighted values from 0 to 5, considering the influence the function points have on the development of a project.

Halstead's Method:

Metrics can be split into two different categories, static metrics and dynamic metrics. Static metrics give information based on the code written, while dynamic metrics give information at runtime.

Halstead's method takes into account the length and volume of a program based on the number of operators and operands. He uses the following measurable quantities:

- n1 = the number of distinct operators
- n2 = the number of distinct operands
- N1 = the total number of operators
- N2 = the total number of operands
-

He also uses the following terms using the previous notations:

- Vocabulary(n) = n1+n2
- Length(n) = N1 + N2
- Volume(V) = N*log2N
- Length(N) = N1+N2
- Potential Volume (V*) as V* = (2 + n2*) log2 (2 + n2*) (the smallest possible implementation of an algorithm).
- Program level (L) as L = V* /V (The closer L is to 1, the tighter the implementation.)

Using these terms and notations, Halstead concluded that as code complexity increases, volume increases while program level decreases. However, his studies have met criticism for a multitude of reasons such as methodology and assumptions, as well as the difficulty for computing the individual metrics, e.g. counting the operands every time the program is changed.

Cyclomatic Complexity:

Cyclomatic complexity is another way of analysing software complexity. Thomas J. McCabe considered that conditions and control statements are the ones that add complexity to a program, rather than operands and operators that Halstead looked at. This method measures the number of linearly independent paths in a program's source code. There are two ways to characterise this metric, both of which are equivalent:

1. The number of decision statements in a program + 1
2. For a graph G with n vertices, e edges, and p connected components, $v(G) = e - n + 2p$.

Code Coverage and Test Coverage:

Code Coverage and Test Coverage are another way to measure the quality of software engineering.

Using Test Coverage in terms of an algorithmic measure, we can compare when an engineer submits code that does or doesn't pass continuous integration testing. This makes it a lot easier to see if a software engineer is producing worthwhile code or if he/she is simply pushing forward technical debt, hoping that it will pass unit tests. Test coverage is an incredibly important metric which allows us to see if an engineer is continuously producing buggy code and trying to make pull requests to the staging/master branch with it. Test coverage as a metric has been getting increasingly popular with the increase in popularity of test-driven development.

Ethics Concerns:

With the vast amount of data being collected by companies with regards to software engineering performance, there are of course a lot of ethical concerns that need to be considered.

Human Bias:

As a lot of the data being compared is done by humans, it's important that everyone being compared is treated equally and that the same measurements are being used to compare everyone. If different criteria are used for different employees, the accuracy of your results will be reduced, on top of it being unfair and unethical.

In a similar vein, the person doing the comparisons must be completely unbiased. If, for example, a project manager harbours any feelings, positive or negative, towards someone they are reviewing, they must base their decisions purely on the work of the software engineer, not the feelings they have towards them. This is an advantage automatic reviewing has over manual, as there is no bias involved, it simply compares people on raw data. However, this is also a disadvantage, as someone who produces create work may also create a hostile work environment for others, meaning that raw data isn't the be all end all, and that some form of human interaction is required. A balance needs to be found between the two, both from an ethical and an accuracy standpoint.

Privacy:

When collecting any kind of data on people, software engineers or otherwise, privacy is always a huge ethical concern. Some data being collected will most likely not bother most engineers. I doubt anyone really cares if a company is monitoring the amount of lines of code you have written. Other data however can be a lot more sensitive, and some engineers might have problems with what's being collected. Take for example the Employee Badge produced by Humanyze. This badge contains a microphone and Bluetooth LE beacon, which allows the person collecting the data to listen in to what you're saying and where you're located. While the aim of this device is to see which employees interact with others and how they speak to one another, you can see the serious privacy issues that could be associated with such a device. Some, if not most, people would take up issue with someone recording every interaction you have with other people in the workplace, especially if you're having private conversations that have no real impact on your performance.

Serious ethical consideration needs to be taken when collecting data from a privacy standpoint, as mishandled data could have serious implications.

Conclusion:

To summarise, there are a vast amount of ways in which one can measure software engineering, some useful, others not so much. There are a multitude of computational platforms as well as algorithm to collect and categorize this data, but ethical concerns should always be considered when doing so.

Bibliography:

1. https://www.infoq.com/articles/measuring-tech-performance-wrong/
2. https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0
3. **Humphrey Watts, The Personal Software Process (PSP), CMU/SEI-2000-TR-022, Software Engineering Institute, Carnegie Mellon University, November, 2000.**
4. **Johnson PM, Searching under the streetlight for useful software analytics, IEEE Software, July, 2013.**
5. https://en.wikipedia.org/wiki/Function_point
6. https://www.softwaremetrics.com/files/15minutes.pdf
7. https://www.umsl.edu/~sauterv/analysis/function_point/FPARP488.html
8. https://publications.waset.org/2684/pdf
9. https://en.wikipedia.org/wiki/Cyclomatic_complexity
10. https://www.humanyze.com/