MEASURING ENGINEERING: A REPORT

ZACHARY MEADE

17340169

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

TABLE OF CONTENTS

TO DELIVER A REPORT THAT CONSIDERS THE WAYS IN WHICH THE SOFTWARE ENGINEERING PROCESS CAN BE MEASURED AND ASSESSED IN TERMS OF MEASURABLE DATA, AN OVERVIEW OF THE COMPUTATIONAL PLATFORMS AVAILABLE TO PERFORM WORK, THE ALGORITHMIC APPROACHES AVAILABLE, AND THE ETHICS CONCERNS SURROUNDING THIS KIND OF ANALYTICS.

INTRODUCTION

In the field of software engineering, measuring productivity is crucial in creating a reliable software product. However, measuring their productivity is no easy task. Due to the growing complexity in software engineering, measuring the productivity of engineers has become equally complex. In order to effectively collect data from engineers inside a company the manager must ask themselves the following questions. What metrics can we use to measure our engineers? Is there a platform out there that can gather and display this data for us? Is it ethical to collect this data?

MEASURABLE DATA:

In order to measure an engineer's productivity we need to have a source of data. For the measurements to be usable we must decide what data we are going to measure and how we are going to gather the data. Not only must the data be gatherable, it must be processable into a displayable form. We must consider what data we want to study their work.

Historically, using source lines of code(SLOC) has been a common measurement metric. This metric is used to measure the size of a computer program by counting the number of lines in the program's source code. This metric has been phased out recently as it is easy to game. As competition in the workplace brings out the worst in people, many people in the past have tried to game the SLOC metric in order to make their performance seem better than it actually is. This can occur in a number of ways. Some engineers create code that is artificially longer compared to their team members. For example, using a full if statement where the ternary operator '?' would suffice. Although this code inflation is small, it can create a huge increase in SLOC in larger systems. This flaw can also be applied across different languages. **While a piece of code in Python may only be a couple of lines, the same piece of code may be significantly longer in other languages.** If a system is using many different coding languages unjust comparisons can be made between different teams.

Furthermore, some engineers may seek out longer solutions to simple problems in order to appear better to their managers. For these reasons this metric is deeply flawed. The attempts to measure the productivity of the engineers has led to a decrease in the quality of code and could lead to conflict between employees due to unhealthy competition.

Example Of

Java Ternary Operator	Java Full If Statement	Assembly Language

minVal = (a < b) ? a : b;	If(a<b){
	minVal = a;
	}
	Else{
	minVal = b;
	}
		MOV R1, #5
		MOV R2, #1
		CMP R1, R2
		BGT bLessB
		MOV R0, R2
		B end
		bLessB
		LDR R0, R1
		end

In a similar fashion to SLOC the number of commits each engineer has can also be used as a productivity metric. However, this metric runs into many of the same problems as SLOC. Engineers may attempt to game the system in order to appear as a harder working employee than they actually are. The same can be said for measuring the features delivered by an engineer. By only looking at what features an engineer delivers productivity measures may be skewed towards employees who did less work but delivered more small features. Although these metrics are simple to measure and track , using systems like GitHub, I cannot consider them to be accurate measures of a software engineer's productivity.

The fundamental problem with these metrics is that engineers tend to be very good at solving problems and getting the best output of systems. Unfortunately, this also includes systems for measuring them.

Code coverage is a valuable form of data that can be used for measuring software engineering. Code coverage is a measure of what percentage of an engineer's code is being executed when tests are run. One hundred percent code coverage is the ideal. If at 100% every line of the software is being at least once. This is a good indicator of how thorough an engineer is in their work. Additionally, low code coverage may highlight a weakness in their skills. However, high code coverage does not mean bug free code. An engineer could write tests which pass implicitly, but also call all methods needed for high code coverage. These methods may contain bugs that are problematic to the system. This can be seen as a flaw in this measurement.

As the past measurement metric have been easy to game but also easy to track. In order to get around the gaming more advanced ways to track an engineer's productivity have been introduced. One such metric is code churn. In simple terms code churn is a measure of the change to a particular section of code over time. If a section of code has high code churn it means the section of code is modified at a high frequency. If a section of code is being modified frequently the additions may not be useful. These additions can indicate that engineers are not working productively.

Another possible metric that is hard to game is getting another engineer to review your code. Each engineer would note how many bugs they find in their team members code. This could not only lead to better overall code quality but would help remove the issue of people creating bugs for themselves to find as it would it would not help their metrics.

A possible way to get comprehensive data on an employee's productivity could be gained through monitoring their whole screen. Having this kind of access to an engineer's whole day would allow for a full evaluation of their abilities in context. However, access to this kind of data also brings with it countless ethical issues which will be discussed later in the report.

Recording engineers away from the keyboard may also be an effective way of measuring an engineer's productivity. By recording a software engineer's every action, or possibly there entire screen. Even though this method would be very effective for analysing there are countless ethical issues that would prohibit this kind of method. Ignoring the countless ethical issues there are a few downsides to this method. Surveying engineers with this method would require a large time commitment, especially if there organisation is large.

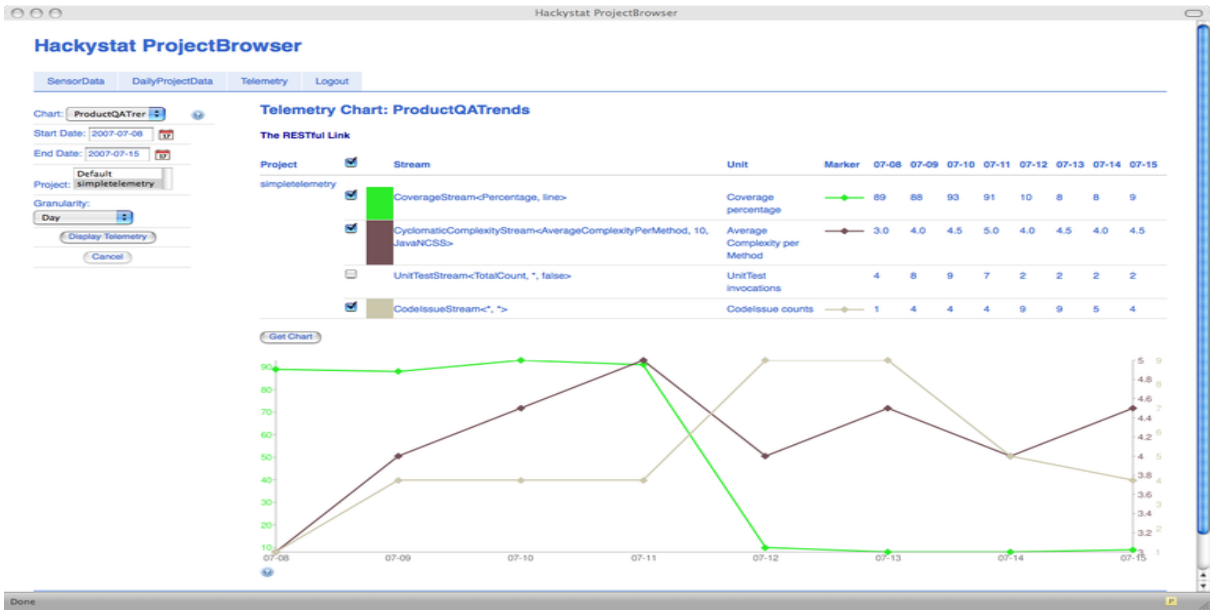
COMPUTATIONAL PLATFORMS

above must be gathered, classified and analysed. By doing this we are able to read and draw conclusions from the data. To gather and process the data in this way many companies big and small use computational platforms. Using these platforms facilitates rapid analysis of data and helps keep the company on the competitive edge of the market.

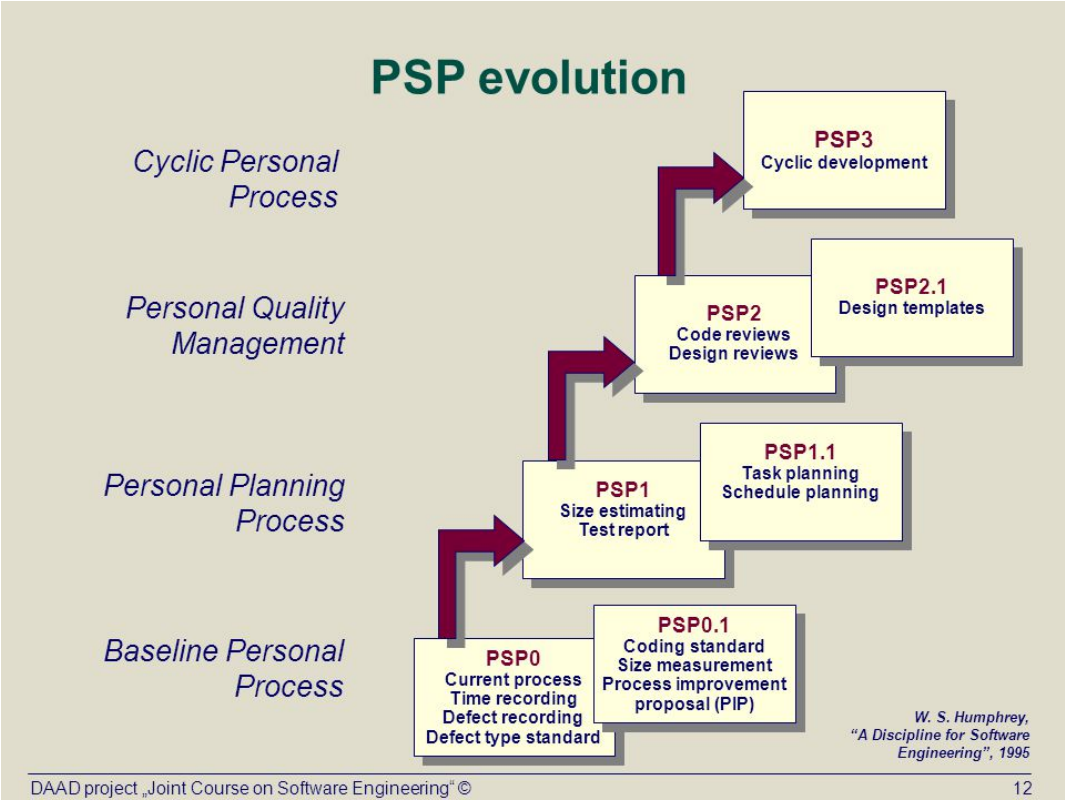
One of the most popular online version control systems is Github. This massive platform is used by many for collaboration or for personal projects. Not only is GitHub an online version control system. It also supplies users with analytical tools to help managers monitor their engineers work. The available tools can help you to track number of commits and lines of code. In addition to the graphs and maps provided by GitHub, you can also

query the GitHub API to gather more data points which you can create into graphs yourself. Unfortunately, these metrics are not commonly used in large organisations due the flaws mentioned above. However, for smaller scale projects GitHub is the perfect tool. In smaller groups it's easier to make sure no one is trying to game the system and make themselves look better.

Hackystat is another open source framework that automatically collects and analyses it data collected from the product of software engineers. The creators of hackystat claim that the platform can also be used to identify the source of problems added to a project. For example they believe that an increase in defect density is proportional to an increase in complexity, thus action should be taken to reduce complexity and see if defect density also decreases. This platform also gives a divide between an individual engineer and the whole projects giving you the options to see both separately. Hackystat's creators also describe the platform as a facility for software telemetry. In simple terms this means the platform is automatically recording and transmitting data from the software engineers to a remote source. For this platform to work, access is needed to the engineer's text editors/IDEs and to any build tools used. This access allows hackystat to transmit and analyse each engineer's data. The data gained from this process can then be used to try and find any patterns in the development process. This platform is very useful for metrics like code churn, lines of code and commits.



The Personal Software Process is a platform that helps engineers better understand and improve their performance by keeping track of their predicted and actual development of their source code. The platform was originally created by Watts Humphrey, to apply the underlying principles of the Software Engineering Institute's Capability Maturity Model to software development. In many implementations of the platform the engineer is required to manually input their own data. Although this does give the engineer a lot of freedom with what data they actually track it is also takes up a lot of their time. If each engineer has to take at least one hour out of their day to input raw data into a spreadsheet then a lot of productivity is being wasted. The manual input can also lead to inaccuracies in the data. No engineer is perfect so they are bound to make some mistakes which can result in misleading data in the spreadsheets. The leap toolkit tried to build on the PSP but also came up with many of the same issues.



Many IDEs also have built in testing suites that give useful functionality for data analysis. For example Visual Studio and Intellij both contain tools that allow engineers and managers to calculate the code coverage for any tested program. This gives them immediate feedback on the thoroughness of their testing. The feedback will highlight the areas of source code that may not be tested at all and may cause issues for the program if released.

There are also computational platforms outside of the development process for the computation of data. Humanyze is one of these platforms. Humanyze aims to help companies measure communication data to uncover patterns and analyze productivity. This platform requires each employee to wear an electronic badge, each of which contains a microphone, Bluetooth and an infrared sensor. These sensors are all used to track a number of different possible data sources. For example, the mic is used to record an employee's speech, Bluetooth can be used to track an employee's location and the infrared can be used to track who an employee is speaking to. In order to avoid many ethical issues Humanyze has a very strict data protection policy to keep all data anonymous and to ensure the data collected is only used for the analysis of productivity in the workplace.

ALGORITHMIC APPROACH

All these sources of information are not very useful until the source data is turned into something measurable. The process of turning the data into something that is both measurable and comparable is usually facilitated by some sort of algorithm. Not only do these algorithms calculate the data being measured they can also change the engineering process to allow measurement.

One of the most popular algorithms are the Halstead complexity measures. Introduced by Maurice Howard Halstead in 1977. He made the observation that measurement metrics should reflect the implementation of algorithms in different languages but also be independent of their execution on a platform. Therefore, these metrics should be computed statically from the code. For calculation Halstead made the following equivalences:

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

Numerous metrics can then be calculated with these values using the following equations:

- Program Vocabulary, unique operator and operand occurrences: n = n1 + n2
- Program Length, total number of operands and operators: N = N1 + N2

- Estimated Program length : N* = n1 log2(n1) + n2 log2 (n2)
- Volume, : V = N x log2(n)
- Difficulty: D = (n1/2) x (N2/n2)
- Effort: E = D x V

As these metrics are generalised across different languages, they can be used to compare engineers from not only the same team but also from different teams across the company.

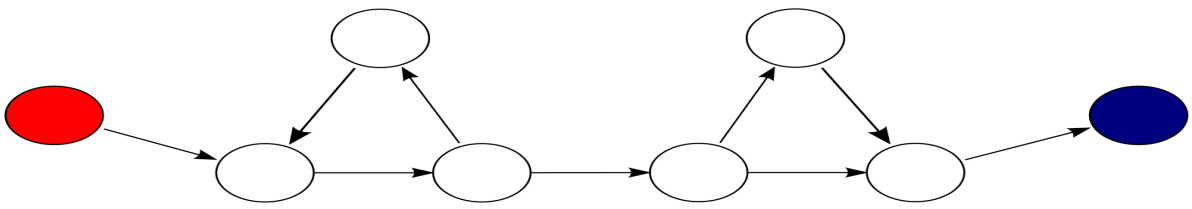
Cyclomatic Complexity

In simple terms cyclomatic complexity is a measure of the complexity of someone's code. This can be obtained by seeing how many different paths can be taken through a given program. For example, if a program contained no control flow statements the complexity would be 1 as there is only one possible path. The complexity of a program is calculated mathematically using the control flow graph of the program. The control-flow graph(CFG) is a representation of all the paths that might be traversed through a program during its execution.

The equation for Cyclomatic Complexity uses the following variables:

- E = Number of edges in the CFG
- N = Number of nodes in the CFG
- P = Number of connected components in the CFG

Equation



M = E - N + 2P

While Cyclomatic Complexity is possible to calculate for individual methods and small programs, the calculation and CFG can very complicated when working with larger scale programs.

ETHICS

After thoroughly investigating the collection and analysis of data produced by both individual and teams of software engineers we are left with one unanswered question. Is all of this ethical? With the collection of data there is always concerns raised. The concerns can be anything from the data being to personal and infringing on the engineers privacy, to concerns on how much data is being stored about the engineer. When measuring a team no ethical issue is too small. Each issue should be treated with the same amount of respect and should be addressed before any actual data is collected. If not addressed properly ethical issues can be critical to the company but can also have a huge impact on the employees. In the most extreme cases, companies can have difficulty retaining or gaining talented staff in the future or even worse may end up in a serious court case.

One of the most important issues when collecting personal data is transparency. For an engineer or any person for that matter, to feel comfortable in their workplace while being monitored is very difficult to begin with. This becomes an even bigger challenge when an employee doesn't even know what data is being collected about them. This can lead to huge drops in employee productivity which ironically, will be seen in the tools that have caused this drop such as hackystat or PSP.

In some of the more extreme monitoring platforms such as Humanyze or the method of recording an entire engineer's screen, it can almost feel like you have a manger looking over your shoulder at all times. This can very easily overwhelm an engineer making them feel like nothing is private. Continuation of this kind of surveillance over a long period of time may lead to deterioration of an engineer's mental condition and thus reduction in their productivity. Not only is this kind of monitoring very unethical it is almost cruel to the employee as they may never know what data their employer has on them.

Furthermore, the way in which a company measures productivity may also bring about ethical concerns. Solely focusing on the metrics of productivity can be seen as unethical. Reducing everything an employee does down to a few simple figures can belittle the employee making them feel their efforts have been pointless. In order to truly measure an engineer's worth and productivity everything they do must be taken into account. These can be as simple as how they affect others in the workplace.

CONCLUSION

It is clear that there are many different platforms, algorithms and metrics that can be used to measure a software engineers productivity. However, from my research I have found that none of them are perfect. Many of the metrics mentioned above run into the same issue, that they're easy to game and manipulate into an engineer's favour making them look better to management. The metrics that were harder to game ran into their own issues. The issues ranged from serious ethical concerns to needing huge time investments from both the company and the engineer.

When choosing a measurement metric for a team's productivity, one of the most important factors to consider what data is truly needed. An employer must know when they have enough and cannot keep pushing their engineers for more and more data.

BIBLIOGRAPHY

Robillard, M., Coelho, W. and Murphy, G. (2004). How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12), pp.889-903.

Robillard, M., Coelho, W. and Murphy, G. (2004). How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering*, 30(12), pp.889-903.

https://blog.ploeh.dk/2015/11/16/code-coverage-is-a-useless-target-measure/

https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0

Johnson, P., Hongbing Kou, Paulding, M., Qin Zhang, Kagawa, A. and Yamashita, T. (2005). Improving Software Development Management through Software Project Telemetry. *IEEE Software*, 22(4), pp.76-85.

https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/

https://www.perforce.com/blog/qac/what-cyclomatic-complexity