Measuring Software Engineering

Sarah Phillips, 2018 - 16318649

0 - Introduction

Considering the fact that the work of a software engineer is exclusively technological, one would assume that its measurement would be achieved with ease. However, it can be difficult to compare the software engineering process to other forms of employment. A server could measure their productivity in terms of how many tips they received in a day, or a salesperson could check the total value of sales they made and either could conclude whether they achieved their target and worked in the most productive way possible. But the progress and efficiency of one programmer, or of many collaborating, doesn't have one concrete way of being quantified or defined. How does one define the productivity of a software developer? How many lines of code they've written, or how long they sit at their laptop for? It is easy to come up with one method of measurement, and subsequently realise what it doesn't take into account. Is it even possible to come up with the perfect measure of such a thing or is attempting to produce accurate predictions and evaluations a waste of effort? This report aims to explore the vague yet vast topic of measuring software engineering and to examine its various interpretations.

1 - Software Metrics

1.1 - Measuring Ambiguity

Since productivity can be interpreted in many different ways, it is hard to measure. Plenty of metrics have been introduced over time for measuring the software engineering process as well as the code itself, with flaws existing for almost all. These range from obvious metrics that use easily retrieved data such as Lines of Code, number of commits, keystrokes, code coverage from testing to more complex methods which will be covered later. Lines of Code perfectly demonstrates how these metrics can have inconsistencies and contribute little information of value to an analysis.

The Lines of Code method of measuring a software engineer's level of productivity was commonly used from the 60's, and this signified the beginning of the use of software metrics. This method originated from assembly languages, which are line-oriented, when punched cards were the primary form of data entry. A line of code was represented using one punched card, so this seemed like the obvious way to monitor how productive a programmer was, as it was tangible evidence and easy to collect this data. Program quality was also measured in defects per KLOC (thousands of lines of code). It is easy to recognise how this crude concept fails to act as an accurate form of analyzing productivity today. One might argue that isn't solving a problem in as little lines of code as possible not the main objective of the programmer? Furthermore, if the programmer knows their productivity is being assessed by program length, would this encourage them to write long, rambling algorithms and duplicate code? Bill Gates once pointed out the obvious weakness in the LOC metric, "Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

The data that is easiest to retrieve is often likely to be closer to redundant. However the LOC metric remains popular even today because it is trivial and requires low effort to retrieve.

Later on in the 70's came the introduction of high level programming languages such as C, Prolog and Pascal. With this came the desire for a metric which regarded program complexity as opposed to size. Maurice Howard Halstead pioneered the idea of Software Science, which introduced the concepts of program vocabulary, length, the minimum volume of an algorithm, level, difficulty, development effort and faults as software metrics. Each of these formulas manipulate syntactic parts of a program called tokens that are recognised by the compiler and are classified as either operands or operators. These metrics are known as Halstead's complexity measures.

$\mathbf{n_1}$ = number of distinct operators	Program Vocabulary = $\mathbf{n_1+n_2}$
$\mathbf{n_2}$ = number of distinct operands	Program length = $\mathbf{N_1+N_2}$
$\mathbf{N_1}$ = the total number of operators	Calculated Program Length = $\mathbf{n_1 \log_2 n_1 + n_2 \log_2 n_2}$
$\mathbf{N_2}$ = the total number of operands	Volume = $\mathbf{N * \log_2 N}$
	Difficulty= $\mathbf{(n_1/2)*(N_2/n_2)}$
	Effort= **Difficulty*Volume**
	Faults= $\mathbf{Effort^2/^3/3000}$

Figure 1.0 - Halstead complexity measure formulas

Software Science has been exposed to criticism, as many have debated over the equations usefulness and accuracy. The data needed for n_1 and n_2 requires the program to already be completed, since they refer to the number of operators and operands. Therefore these equations bear no use in terms of predictiveness.

Quantitative data can also be achieved through Github and other version control software. Monitoring the number of commits a programmer makes over a period of time and the size of each commit seems like an obvious indicator of consistent work. However simply looking at the numbers doesn't account for how useful each commit was. Did the programmer push useful, bug-free code in one commit or did they simply push whenever they wrote a new method without debugging it? On the surface level, seeing constant activity implies productivity, however this requires further investigation to actually deduce whether this is the case.

1.2 - Beyond the Numbers

Measuring code and measuring productivity are not mutually exclusive, but are also not completely correlated. If a writer was able to publish a thousand page book without ever proofreading, it is guaranteed that the pages would be littered with errors and inconsistencies. To conclude that the workflow of said writer was exceptionally productive just by looking at the rate of output would be erroneous, as now no-one will be able to read the book and all the time and effort used writing page after page would be in vain. One could apply this logic to assessing a programmer's progress by lines of code, number of commits, number of hours spent programming or number of keystrokes. What do all of these metrics have in common? Numbers.

Sometimes it is necessary to step away from the numbers and focus on qualitative data rather than quantitative. An employer could record how many hours an employee spent on their computer and use this data to deduce whether they were working productively. But this is far too vague to deduce any reliable statistics. Without knowing *what* they were doing and *how* they were doing it, the measurement is inaccurate on its own as a metric. This is one of the issues which makes the measurement of software engineering so complicated and ambiguous - context.

1.3 - Examining the Engineer

Manual data collecting methods such as the Personal Software Process (PSP) have been developed to give a more in-depth look into the software developing process and these require the engineer to collect data on themselves. Reflective software engineering allows the engineer to track their productivity and improve on weaknesses, however manually taking down information interrupts the workflow. As with a CPU, context-switching hurts performance - that is switching between two different activities. Personal data collection also relies on complete honestly on behalf of the engineer, which means data is easily contaminated by human nature. Although the PSP is flexible with its analytics, it is also fragile according to a study done by Johnson [3] . After conducting a study on 30,000 data values, it was concluded that the use of the PSP led to “incorrect process conclusions despite a low overall error rate”.

Systems such as HackyStat and PROM aimed to improve on this. These systems gather logs of commands run both inside and outside the IDE as well as clickstream data from the developer’s computer. They allow the developer’s work pattern to be monitored, allowing productive work to be recognized. HackyStat allows data to be collected automatically without interrupting the developer or depending on the developer themselves to record it, meaning it is less likely to be inconsistent due to human error or being tampered with. HackyStat doesn’t rely on network connectivity to collect data, storing data locally when the engineer is working offline and uploading it when a network connection is detected. The system can also track group-based development, recording interactions between team members, as well as gather second by second analysis of developer behaviour.

The hypothesis behind the use of these analytical systems is that developer’s work patterns can indicate whether they are employing effective working habits and techniques in order to complete an assignment. However there are ethical concerns regarding these types of systems, which will be touched on later.

1.4 - “The Customer is Always Right”

Kan wrote that “Customer satisfaction is the ultimate validation of quality” [2]. Product quality is often analysed in an effort to measure a worker’s productive capacity. It doesn’t matter how efficiently a software developer delivered a product for a client or customer - if that client is unhappy with the product, the engineer’s goal has not been achieved. This is not always at the fault of the developer or the development team, as it is possible for miscommunications and different interpretations of targets to arise, particularly if the client is not “tech-savvy”. However, in such situations it falls upon the programmer to inform the client to the best of their abilities of what are realistic expectations, as well as any technical difficulties which are likely arise during the course of the project.

Albrecht’s function point analysis is another form of function size measurement. It evaluates the functionality of the code that is delivered to the user based on the user’s functional requirements. It helps programmers to estimate the potential size of a software project and the amount of time required to finish the project. The functional user requirements of the software to be developed are categorized into one of five types - external inputs, external outputs, external inquiries, internal logical files and external interface files. FPA is independent of programming languages, which means it can be applied to more than one programming language. Although there are many advantages to using function points as opposed to LOC, such as being able to apply it early in the software development process and allowing more predictability, most effort and cost models are LOC based still as it is easier to apply and less time consuming. However, any metric that allows the developer to make good predictions for the client in regards to project milestones contributes to the client having expectations that are not too high.

2 - Algorithmic Approach

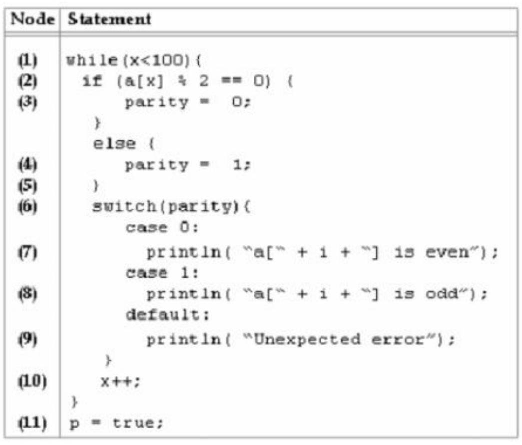
2.1 - The Perfect Measure

An algorithm is defined as a process or set of rules to be followed in calculations or other problem-solving operations. That being said, does there exist a list of instructions that calculates the characteristics of the perfect programmer which could be applied to every programmer of every team of every company? This is would be impossible. What works for one case brilliantly, may not work at all for another. For most problems there exists numerous algorithms that can solve them. However, not all algorithms are created equally. Some are faster, and hence more popular, others are made to focus on more detailed results as opposed to efficiency. Some algorithms don’t scale well, and so what works for assessing a small team of programmers may not work for a tech giant consisting of countless teams.

2.2 - Assessing the Code

Productivity can be defined as ‘the effectiveness of productive effort, especially in industry, as measured in terms of the rate of output per unit of input’. Hence, it makes sense to look at the output of a programmer in order to assess the development process. Rather than analyzing the code in terms of its size, it can be more useful to look into its complexity.

In order to measure a program’s maintainability and to encourage quality control, the cyclomatic complexity metric was designed by McCabe in 1976. Simply put, a program is mapped out like a graph and the number of independent connections or ‘paths’ in a program examined. This allows one to see how much effort is required to test the program. Using the formula V(G) = e - n + 2p, where e is the number of edges, n is the number of nodes and p is the number of unconnected parts of the graph, we can calculate the cyclomatic complexity number of a program. 10 is the recommended maximum number for a program to have good maintainability and testability.



```
Node  Statement

(1)   while(x<100){
(2)     if (a[x] % 2 == 0) {
(3)          parity =  0;
         }
         else {
(4)          parity =  1;
(5)      }
(6)      switch(parity){
             case 0:
(7)            println( "a[" + i + "] is even");
             case 1:
(8)            println( "a[" + i + "] is odd");
             default:
(9)            println( "Unexpected error");
          }
(10)      x++;
       }
(11)  p = true;
```

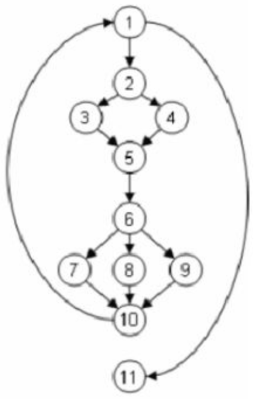


Figure 2.0 - Example of McCabe’s cyclomatic number

Kan concluded that this metric is most helpful in identifying overly complex portions of code as well as non-complicated portions likely to have a low number of defects. This aids the programmer in deciding what parts of the code they should be spending the most time inspecting and improving. Cyclomatic complexity also gives an estimate of the effort required to program, debug and test.

Code coverage refers to what percentage of the code is covered by the tests a programmer has written. McCabe’s cyclomatic number gives an indicator of the amount of tests that will be needed to fully test code, and the readability of the code. If a programmer can condense a 50 line algorithm into 10 lines, it ends up being a waste of effort if nobody else can understand it. If code is hard to maintain, it creates more work in the future for new developers trying to build on it. This metric has been widely used in order to enforce quality control in code, for example, Hewlett Packard decided that any module written with a complexity over 16 must be improved[1].

Kan also conducted a study on a particular component of the AS/400 software system, assessing how various module design metrics, including McCabe's cyclomatic number, relate to defect level and can help improve software development processes. As a result of the study, the development team came up with a software improvement plan that included the following steps,

- Examine modules that have a moderate complexity but high defect level in terms of design and implementation.
- Identify high-complexity modules and carry out debugging and restructuring until the complexity is reduced to 35 or lower.
- Ensure no compilation warnings arise for any modules
- Improve test effectiveness and code coverage for all modules, especially complex ones.
- Improve documentation on all components.

This study shows how algorithms measuring exclusively the work produced by software engineers can lead a team to develop plans for improving their software. Kan commented after his study on how the team working on the component have been making 'significant improvement in the component's quality' since following their quality plan. However, he also recommends the use of complexity metrics for small teams of programmers rather than large organizations, due to the analysis being carried out at the module level of the program.

2.3 - Test Driven Development

Every good programmer knows - or should know - that testing is a necessary part of developing any kind of software. Engineers are especially encouraged to make use of the Test-Driven-Development practice. This involves starting with writing a deliberately failing test case in order to define a new function of the code, then producing the minimum amount of code to make the test pass. The code is then tweaked and redesigned until it is up to standard. This method is sometimes known as the "Red-Green-Yellow" approach [5]. It is seen as a less time consuming approach to coding when compared with the test-last method and is said to generate 100% code coverage with ease.

Zorro is an automated system that was developed to test whether engineers are making use of Test Driven Development, making use of Hackystat to collect developer behaviour data. But how does software identify when a specific method of developing code is actually being carried out? Data that is collected by Hackystat is then organized by an application called Software Development Stream Analysis, which sorts the data into a sequence ordered by timestamp. The sequence is then split into tokens that define specific events taking place such as compiling or running a unit test. SDSA then classifies each event in the sequence and evaluates it according to the method of software development that is under analysis.

Although this method of identifying software engineering practices can be useful for those wanting to improve the way they or their employees develop code, there exists issues which can cause confusion and inconsistencies amongst programmers. Since it is a method of programming, not everyone is going to do it in the exact same way. Some programmers will apply the general convention of Test-Driven-Development more loosely than others. Where does the line between TDD and not TDD exist? In addition, some question whether it has actually been proven that Test-Driven Development is a more efficient approach than others. A few studies suggest that this method bears no advantages over test-last or refactoring. Johnson cites studies carried out by Erdogmus, who found that software development by means of TDD was of no higher quality than the control.

This leads one to question whether employers should require their engineers to develop code in a set way that they deem to be most efficient. Surely if a specific work routine allows the programmer to deliver output effectively, then they should be able to continue with the method that suits them best. Similar to how students have their own ways of studying for an exam, programmers may have their own convention for developing software. However, in terms of group software engineering, it can be understandable that programming according to an agreed protocol could contribute to easier code-merging and interactions between team members.

3 - Ethics and Concerns

3.1 - Big Brother

Nowadays, the primary concern with regards to data collection is privacy and ‘being watched’. Any form of monitoring behaviour over a period of time has the potential to evoke concern in the general public and media, as it can be seen as essentially spying. This is one of the adoption barriers for the HackyStat system. Although the unobtrusive method of data collection is seen as an advantage, others see it as an issue. Many programmers understandably feel uncomfortable at the thought of the activities at their workstation being monitored at any time without it being obvious to them. This is especially due to the fact that the technology allows for such fine-grained data collection and engineers are concerned with regards to management’s access to such detailed statistics on individuals. However it can also be argued that there aren’t many alternatives when it comes to evaluating employees performance. It is unfortunate but without such invasiveness, the accuracy and depth of the results is significantly limited.

HackyStat has taken some action to reassure its users that their privacy is still intact. Access to the data recorded requires a password that is only known to the developer who owns the data. Users are also able to store all the data offsite and out of reach of management.

One point to consider is that once someone knows they're being watched, they can become more efficient workers. Labour is one of the primary expenses of a company, so it makes sense for management to be able to monitor who is worth what they're being paid. As a result, the top performers can be recognized and rewarded accordingly. However, does obtaining effective results this make the action acceptable?

3.2 - Innocent Data Collection

Services for software product analytics have grown in popularity, one of which is Sonar. This service performs assessments of data that is already available in a repository, rather than gathering the data itself, reducing overhead expenses. This is an uncontroversial method of measuring productivity, as the focus of the analytics is on the product and not the developer's behaviour. However, without the focus on the developer's work process, the results are quite limited and don't account for what methods of development are most productive. This is the tradeoff between lack of controversy and useful, in-depth results.

4 - Conclusion

There is no cookie cutter for the ideal software engineer, just as there is no flawless unit of measurement for such, despite software metrics having been in use for decades. One method may reap great results for a specific situations, such as Kan's study regarding McCabe's Complexity Number. But what works for one case may not work for another. The topic of measuring software engineering itself is subjective. Some interpret this as measuring productivity of workers, and this itself opens up a subtree of ways to do so, such as correcting software engineer's methods of developing code, examining work and behaviour patterns and what takes place during the development process. On the flip side, analyzing the code of the finished product can also stand as a means of measuring software engineering - assessing it's quality and how the code as well as the development process can be improved in order to reach a higher level of achievement for both the individual engineer and the business.

As a result of this topic's ambiguity, it is complicated for employers to choose the metrics that can point to the steps needed to be taken to improve the output and efficiency of their employees. An interesting observation is that the metrics that existed in the 70's are still being used today, especially in the case of LOC and defect counts. Although these are seen as incredibly crude units of measurement, companies insist on using them due to their simplicity and the reduced time, costs and controversy associated with collecting such data. This is the tradeoff between those metrics and the more the complex and potentially invasive yet sophisticated methods that were outlined earlier on in this report. Time is arguably the leading factor in the majority of processes which take place within a company, since time is money. Therefore, those that may help in the long run can be lower on the list of priorities in comparison to what is happening in the now. This isn't to say that software metrics are not worthwhile to any company or individual, it just remains to be seen that software engineering could not progress without them.

5 - Bibliography

1 - Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157.

2 - Stephen H. Kan. 2002. Metrics and Models in Software Quality Engineering (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

3 - Philip M.Johnson, University of Hawaii at Manoa (2013) “Searching under the Streetlight for Useful Software Analytics”

4 - Sillitti, A. Janes, G. Succi, and T. Vernazza, “Collecting, integrating and analyzing software metrics and personal software process data,” in Proceedings of the 29th Euromicro Conference. IEEE, 2003, pp. 336– 342.

5 - Johnson, Philip M., and Hongbing Kou. "Automated recognition of test-driven development with Zorro." Agile Conference (AGILE), 2007. IEEE, 2007.

6 - W. Snipes, V. Augustine, A. R. Nair and E. Murphy-Hill, "Towards recognizing and rewarding efficient developer work patterns," 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, 2013, pp. 1277-1280.

7 - Edwin C.Lim, Edith Cowan University, 1994. “Software metrics for monitoring software engineering projects”