Measurement Report

Matthew Ng - 16323205
ngm1@tcd.ie

Introduction

The attempted measurement of software has been an ongoing process in Computer Science. Quantitative measures are necessary in all fields of science, but software has remained elusive. Each of these metrics have sought to either improve the debugging process, optimise software, estimate costs, plan schedules or ensure quality. Various computational platforms exist with the intention of measuring the work of software engineers, and many algorithms designed to measure the amount of work done. However, the deeper we dig into the metrics, the more it starts to move away from the software itself and into the software engineers themselves. Privacy concerns begin to bring new issues, and hesitancy into delving further into analysing it rises. This report provides an overview of some of the methods of measurement that have appeared over the years, the current computational platforms which offer these measurement services, and some of the algorithms designed to guide the assessment of software. In addition, I found it useful to talk about what businesses are currently doing. It also takes a look at the ethical concerns that quickly make themselves known as this data is collected.

Approaches to Measuring and Assessing Software Engineering

For many years, research has gone into measuring the development of software to improve development processes and thus, products. It remains obvious that this issue is far from trivial and the "Streetlight Effect" is often cited to highlight observational bias in these attempts. In the context of software, it refers to how we have attempted to measure development based on easily recovered metrics with little impact. For example, one could simply collect lines of code per developer from a repository. Nobody poses any concerns towards the collection of this data, and it's simple to collect. However, one must ask is this really accurate? At this stage, it seems to be taken for granted that linking lines of code with the work of a software developer is inaccurate. As Bill Gates supposedly stated:

> *"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."*

Some even attempt to use this to suggest that software simply can't be measured. However, such ideas have not prevented the continued research into software metrics. A large area where lines of code seemed to display some usefulness was the prediction of faults in code. It was taught that larger volumes of code would produce more errors. Unfortunately, given the ease of collecting this metric, it is conveyed through case studies that smaller bodies of code do not display any sort of correlation with lower volumes of error, not even a weak support. Lines of code does however, continue to be collected as a statistic as it's still recognised as holding a degree of significance. One can easily see the difference between somebody who has contributed 15,000 lines of code versus 10. The ambiguity arises where one developer has supposedly contributed an overwhelming amount of code, but that this functionality was simple and did not require much problem solving ability. The developer with a lower contribution of code has perhaps added better or more complicated code with which the entire project relies on. Perhaps there also existed a massive hurdle within this particular code that the developer had to overcome. These scenarios are easily imagined, and leads us on to more in-depth metrics.

Cyclomatic Complexity

Cyclomatic complexity was developed in 1976 by Thomas J. McCabe, Sr. It involves measuring the amount of linear paths that a program has. Therefore, a completely linear program, involving no control flow at all (loops, conditionals, etc.) would constitute the simplest program, and naturally, this corresponds to the simplest possible model a software engineer could envision, with little room for error. What is interesting to note, however, is that case studies have shown that the level of cyclomatic complexity in a program does not appear a better predictor of faults and failure-prone modules than simple size metrics. Is complexity in a program even a sign of a better one? A massively complex program may be monstrously inefficient and would be harder to debug. The software engineer should not be awarded for producing such bodies of code if a simpler solution exists. Of course, it could easily point to more work being done on behalf of the developer, but that is not necessarily a good thing. Any business wants time to be spent efficiently, and wasting time on a needlessly complex solution harms the development process.

Code Churn

Code churn describes how much code is being modified over a short period of time. The idea behind measuring this is that a higher code churn is not ideal, as it means that either the requirements were fuzzy, that the problem is difficult, or in some cases, it is interpreted that the software engineer is burned out, unmotivated or under-engaged. It is understandable that any project would want to minimise code churn, despite it perhaps providing additional optimisation, surely this optimisation could have been done at the start. Inefficient code was initially produced, and if good code was inserted from the project's conception, this would save an immense amount of time.

The simplest measure of code churn is used as a more accurate metric of "lines of code" written. It decides that editing the same line of code only counts for one line of code actually added, and that your previous contributions to this line of code were not valid and that you were not productive during this time.

A more complex measurement of code churn involves looking at modified complexity. We begin by determining a code "delta", which is the difference in relative complexity between builds. The idea is that, if software modules are replaced between builds by modules of equal complexity, it's delta will be zero. The absolute value of the code delta is then taken to be the code churn. Essentially, the change in complexity is measured as code churn, rather than change in simple size.

Measuring Engineer Activity - Personal Software Process

Attempting to predict error cases was not the only motivation behind the study of software metrics. The engineers themselves would become the source of much analysis. One method which strove to provide in-depth analytics into the development of software was the Personal Software Process, or PSP. This was a manual process in which the software engineers themselves would record their requirements, planning, development, estimations, time taken, number of defects and size of the project. Even compiler errors were recorded. Afterwards, there was a design and code review. The objective was that these analytics would help them better schedule their work and to improve the quality of their code. TSP also existed as an alternative, where teams would collectively record the data as opposed to the individual.

The manual aspect of PSP meant that a lot of issues could easily arise with the analytics, despite it being very flexible due to it being controlled by the engineers themselves. It also absorbed a lot of time, since no tool existed that could manually collect the data. One version of PSP even required 12 forms to be filled out.
One automated attempt to gather this data was created, known as the LEAP toolkit *(Lightweight, Empirical, Anti-measurement dysfunction, and Portable software process measurement)*. It still required the developer to manually enter most data, but subsequent analysis did not require this. It was hoped that this would address data quality issues and also provide some extra information such as forms of regression. However, it's increased automation did mean certain flexibility was lost despite being considerably more lightweight. After a few years of use, it was decided that PSP could never be fully automated; manual input would always be required in the majority of cases. The development overhead had also started to be seen as having too much cost for it's return.

Hackystat

Hackystat was another approach to measure software development but went in a different direction than that already considered by the PSP and LEAP. Hackystat sought to not involve developers in the data gathering, in fact, the idea was that they should not be aware of it at all while working. Data was collected and stored until a network connection was available, and it would then send it to a server. It's objective was to measure software in an incredibly fine-grained way. Sensors would be attached to development tools, gathering raw data. Other services would then query this data and build high-level analysis. The client-side instruments could therefore collect data on a minute-by-minute or even a second-by-second basis. It's tracking includes how the developer edits methods, constructs test cases and runs tests. Hackystat was also able to track how developers worked together when they edited the same file.

However, many concerns arose with Hackystat. Firstly, the unobtrusive data collection was actually seen as a problem. Developers did not like to install software that would track them without telling them much about it. It was also referred to as “hacky-stalk”, due to the transparency available regarding other team member’s working style. Software engineers were not comfortable with this detailed data about them being available to management.

Process Assessment

Apart from assessing the precise second-by-second activities of a software engineer, the assessment of the development process itself could bear some results by pointing out the possible weak points of development, where errors in design or implementation could have arisen. This process could even be standardised to accelerate good practice. It has been considered that a metric of the software itself is not useful until late in the development process, and therefore process quality can lead to better results. A typical process assessment method involves a diagnostic tool to enable the “maturity” of software to be measured, and this is then compared to a generic model. Best practices are then analysed. A questionnaire is often then used as an aid to data gathering. Essentially, rather than providing a numeric representation, the software process is divided into various fields of complacency with standards and the level of the software process maturity is determined. The process can be defined in terms of “initial”, where there is an almost chaotic, undocumented environment that the software is being developed in, to “optimising”, where the process is efficient, and all the focus is on improving performance. The “optimising” stage is the ideal.

The main thing that this approach is defining is the measurement of the project as a whole rather than individual developer contributions. This is a defining part of an agile development process, where the perspective is less on definitive metrics of code, not to say that these aren’t also collected. However these are only used as a general guideline and it’s up to the development team to interpret this.

Real World Examples

To take the example of Google, a large amount of focus is on maintaining a standardised practice and having extensive review tools. All code added also must be reviewed by another engineer. What should be noted here is that rather than trying to collect exact metrics, this attempts to address the essence of the issue which software metrics has attempted to improve. That would be to enhance the output and quality of software engineers and thus software. It's more of an assessment of the process. It relies on the engineers themselves to recognise the work of another and means there should not be much worry about misinterpreted statistics such as tracking the code itself. Google does track code errors too through it's extensive automated testing process, and tracks each build. Finding and avoiding problems is the main task of much of this. However, there is also an experimental section of repositories where standard review processes are not enforced.

IBM's practice includes measuring the following: Cost-Performance Index, Schedule-Performance Index, planned tasks versus tasks completed and "scrap" (lines of code deleted). Basically, IBM is focused on whether the team is meeting deadlines, how much the level of progress is costing, and how much code being written is even used. Hence, if you end up deleting a lot of code, you shouldn't have written it in the first place, the same thought process with the concept of measuring code churn. This hopefully encourages the software engineer to write good code from the beginning, so that the end product may be better. After all, you can't expect all code to be refactored, even if it really should be.

Current Computational Platforms

What should be noted here is that the there is no shortage of platforms dedicated to project management as a whole, rather than individual analysis of software engineers. This short list seeks to simply gain a grasp at the variety of approaches used by different platforms rather than to be representative of the proportion of those that are widely used.

Process Assessment - CoSEEEK

CoSEEEK (Context-aware Software Engineering Environment Event-driven Framework) is a framework that assesses software engineering on the basis of process. To achieve this it uses an algorithm[4] that applies ratings to maturity and capability levels, compared to a base practice. Sensor support for each is necessary to work with CoSEEEK. Eclipse and Subversion are two well known tools for instance which support it. As you can see, this is inspired by the earlier mentioned "Hackystat", and actually uses Hackystat framework to implement "Event Extraction", which, in summary, receives certain events from it's sensors, such as test cases being run, to determine the stage of the process.

Bug Tracking - Jira and Buganizer

Jira is an issue tracking product which later developed into covering project management. Buganizer is a similar bug tracking system used by Google. It also tracks clean-up processes and categorises bugs and takes in customer issues. The idea behind these metrics is that tracking the amount of bugs and issues will improve the quality assurance of your program. In many agile development teams, the source of bugs is not often focused on, and the team attempts to work collectively to avoid future problems. Of course, in reality the team members will not ignore if a disproportionate amount of bugs are arising with one team member, but surely the nature of how teams are generally organised and run should hope to solve this issue themselves. The measurement of bugs, of course, should always be used alongside other measurements, as alone it does not count for much.

Time Tracking - Toggl

Toggl is a simple system that measures the amount of time various tasks are being completed at. It's basically a form of process assessment and seeks to analyses how long certain processes are taken to better schedule future tasks and also gain an idea of the speed of engineers. This may seem incredibly primitive compared to some of the more algorithmic approaches, but it is a useful statistic in assessing the development process.

Behavioural code analysis - CodeScene

CodeScene is a "social code analysis service". CodeScene seeks to identify code "hotspots" by selecting which code has the most development activity. It also measures code complexity change, code churn and refactoring efforts. The idea is that: increased complexity is bad, and hotspots are more likely to contain bugs as they're constantly being changed.

CodeScene also measure the connectivity of teams in a social graph. The idea is that your software architecture is going to correlate with the connectivity of teams to facilitate communication. This concept is known as Conway's law. This approach also tries to determine the value of certain engineers in the team, as there may be a large reliance on this member to hold the social structure of the project together.

Ethical Concerns

Various approaches to the measurement and assessment of software engineering have been discussed, and some ethical concerns have been mentioned. This section seeks to cement any previously mentioned issues and to also describe much more of them.

Firstly, we can easily find metrics to which people find little issue with. Namely, the measurement of lines of code added, code churn and complexity. Naturally, there are issues related to engineers wanting to be measured at all, but well shall ignore this. The data gathered related to these is completely transparent to the engineer and it is known to them when they are influencing them. Therefore, they feel in control of this aspect and it does not make them uncomfortable.

On the topic of control, whilst the Personal Software Process is incredibly in-depth, the engineer is fully in control of all information. While incredibly tedious, the engineer freely interprets and records this information himself.

Hackystat was already mentioned as having immediate concerns over the gathering of data. Engineers displayed discomfort over their every action being personally measured as they had no control over the interpretation of this data. It is only natural for them to take on the opinion that it shouldn't matter how productive they were over time or how well they were designing and writing software as long as the end product was fine.

The further we delve into what data we gather does the opposition to it increase. For example, we have seen how CodeScene attempts to piece together social connectivity within teams, but to what extent would an engineer be comfortable with their social interactions being monitored? Outside of what are you working together with people on, the increased monitoring of the relationships within a team in order to determine more and more analytics makes engineers uncomfortable.

Perhaps this concept is why agile teams tend to focus on team performance rather than individual, as in general business practice, the needs of employees and their morale is considered important in productivity. How comfortable the employee feels in the workplace may be more important than pure productivity.

Conclusion

As we can see, there is no consensus on how software should be measured. Various methods of measuring software engineering exist, and are used in many platforms available to software engineering teams today. We have explored how in two cases how large companies have sought to measure their engineers, with some aspects of the methods we already discussed as well as additional ideas. Examining the platforms also showed that various methods we did not discuss initially existed, but generally were related to them. It was also conveyed how ethical concerns arise through the privacy and loss of control is mainly the cause of discomfort with the gathering of metrics. The measurement of teams as a whole seems to be a generally applied model to mitigate this, with the individual metrics being simple and easily accepted due to their relative transparency. Software metrics will no doubt continue to develop as it becomes known that some methods are not accurate.

Bibliography

1. “Searching under the streetlight for useful software analytics” *IEEE Software* (July 2013) by Philip M. Johnson

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

3. “Network Analysis of Software Repositories: Identifying Subject Matter Experts” by Andrew Dittrich, Mehmet Hadi Gunes and Sergiu Dascalu

4. “Automated Software Engineering Process Assessment: Supporting Diverse Models using an Ontology” by Gregor Grambow, Roy Oberhauser and Manfred Reichert

5. “Process Assessment and Process Improvement - the need to Standardise?” by Edwin M. Gray and Robin B. Hunter

6. “Software Engineering at Google” (2017) by Fergus Henderson

7. “Effective management through measurement” IBM (2004) by Doug Ishigaki

8. “Code Churn: A Measure for Estimating the Impact of Code Change” by Sebastian G. Elbaum and John C. Munson

9. “6 causes of code churn and what you should do about them” GitPrime (2016) by Ben Thompson

10. https://codescene.io/

11. “30+ Metrics for Agile Software Development Teams” (2016) by Andy Cleff