Measuring Software Engineering

Michael O'Callaghan

Abstract

The purpose of this report is to review and discuss the different ways in which the software engineering process can be measured and assessed in terms of measurable data, the computational platforms available, the algorithmic approaches available and the ethics of such analytics.

1. Introduction

Software Engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance (Sommerville, 2015).

The term "software engineering" was first used at the NATO Software Engineering Conference in 1968 to address the "software crisis" that was unfolding at the time. During the 1960s and 1970s, developers did not have the expertise to build the necessary software, and if they did, the software was often unreliable and difficult to maintain. Furthermore, the previous individual approaches to program development could not be applied to large and complex systems. Independently, Margaret Hamilton, the renowned mathematician and computer science pioneer, first coined the term "software engineering" while working at NASA as the lead developer for Apollo flight software (Nasa.gov, 2003).

Since the software crisis, the discipline of software engineering has developed significantly and today "software is eating the world" as Marc Andreessen famously once said. Every industry is being transformed by software and even non-technology companies are being run on software and delivered as online services. Given the growing influence of software on society today and desire of managers to gain insights into the work of their software engineers, it is important to understand the process of how software engineering is measured and assessed.

This report addresses the different ways to measure this process of software engineering, the platforms that are available to analyse this information and the algorithmic approaches available to monitor this data. Finally, the ethical concerns that arise from this process are examined at the end of the report. There are many differing opinions regarding how best to measure software engineering, and whether it can be measured at all. These contrasting viewpoints will serve as a basis for this report.

The first step however is to understand what the software engineering process entails. The process of software engineering is the set of activities that are required to develop a software system (Sommerville, 2015). Although there are many different software processes, all involve the same four steps: 1) *Specification*; 2) *Design and Implementation*; 3) *Validation*; 4) *Evolution*. Firstly, one must define what it is the system should do. The next step involves defining the organisation of the system and then implementing it. Validation refers to checking the system executes the task correctly. Finally, one should continuously update and improve the system.

These steps are carried out to develop any software engineering project however there are several different software process models that can be used to implement such projects (Elgabry, 2017). These include:

- Waterfall Model – plan-driven process that has separate and distinct phases of development
- Incremental development – process that involves developing initial implementation, receiving user feedback and then updating implementation
- Integration and configuration – system is assembled from existing configurable components

It is important to note that, in practice, most large systems incorporate aspects from all of these models (Sommerville, 2015). The different phases between idea generation and idea system deployment is referred to as the Software Development Life Cycle (SDLC). The SDLC can be framed in many ways with the most common methodology today being *agile methods*. Agile methods refer to models that are based on the incremental and iterative approaches and the increments for agile methods are typically small with the implementations being made available to the user after each update.

2. Measurable Data

2.1 Overview

There is much debate in the discipline of software engineering as to what measurements accurately reflect the productivity of software engineers. Measuring the work of software engineers is more challenging and complex than the work of other professionals. Productivity or output cannot simply be measured by the number of units produced or hours worked. That said, it is vital to understand the quality of software that is being developed to maximise efficiency and minimise both cost and time. Historical measurements such as lines of code (LOC) are still used today by some, despite being generally accepted as inconclusive and archaic. In the latter half of the 1970s, there was a move away from these measurements towards what were hoped to be more accurate and comprehensive. The need for these new developments was precipitated by an increase in the diversity of programming languages, as for example LOC in assembly and high level languages were unparalleled. Manual data collection from software engineers became the norm.

2.2 Methods

2.2.1 Source Lines of Code (SLOC)

Source Lines of Code is a metric that uses the number of lines in a software's source code to measure the size and complexity of software project. As mentioned previously, this metric has long been the standard for measuring the software engineering process.

Although SLOC can be effective in measuring software engineers level of effort, Bhatt, Tarey and Patel outline some downfalls associated with using this metric in their paper

"Analysis of Source Lines of Code (SLOC) Metric". Some of these flaws include a lack of accountability, lack of cohesion with functionality and ambiguity surrounding whether comments should be included in the lines that are counted.

2.2.2 Number of Commits

This metric is based on the number of times a software engineer contributes to the code. The number of commits performed by an engineer can highlight the level of their activity however it does not provide any indication of the quality of the contributions.

2.2.3 Technical Debt

Technical Debt is a concept in software development that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the optimal solution which would take longer. A certain level of technical debt is inevitable however management should aim to minimise this metric.

2.2.4 Lead Time

Lead time is defined as "the time elapsed between the identification of a requirement and its fulfilment". Put simply, it measures the total time it takes for a project to move through the development process, from the original idea to the completed implementation. An advantage associated with using this metric is that it can be used to estimate the amount of work an employee is capable of completing in a particular timeframe and assess developers' adherence to deadlines.

2.2.5 Cycle Time

This metric measures the length of time it takes to actually implement a new change into a software system. Cycle time only accounts for the time once the developer starts working on the changes whereas Lead Time counts the time from when the initial request is made.

Cycle Time can be used in conjunction with Lead Time to assess the efficiency of a software engineering process and the capability of an individual developer as well as to estimate the length of time a project might take to complete by evaluating data from previous projects.

2.2.6 Code Coverage

This metric measures the proportion of the software engineer's code that is covered by the test cases. Code coverage be used as a quality assurance tool as the higher the code coverage, the lower the probability of having undetected software bugs. Therefore, this metric can be used by the management of an organisation to determine the attention to detail of a particular software developer. However, a drawback associated with this metric

is that the tests devised must be sufficient in covering all eventualities, otherwise optimal code coverage does not translate into a bug free software application.

2.2.7 Code Churning

This metric measures the rate at which an engineer's code changes over the development life cycle. Code churn is typically measured in LOC that were modified, added and deleted over a short period of time. This metric can be useful as it can provide an indication into an engineer's efficiency given that it describes how much time a developer spends on a feature while making limited progress. If a churn rate is particularly high or low, it is important that management analyse why this would be the case. Inferences can be drawn from the churn rate. For example, a high churn rates could indicate indecisiveness and uncertainty as well as the fact that the project could be at an early stage as many changes are being implemented.

2.2.8 Bug Fixing

When using this metric, management should pay attention to the severity of bugs and how they are overcome as opposed to the number of bugs in a developer's code. The time spent fixing bugs could be used as a metric, but the time spent in relation to bug severity/quantity will vary from engineer to engineer.

2.3 Limitations

Caution must be taken when using these metrics in isolation to draw conclusions. Although these metrics are useful, other aspects of engineers work should also be analysed to gain a more accurate, holistic perspective of the process, system and employee. Some other metrics that should be included in assessing the process are a developer's ability to collaborate in a team environment and positively impact the business in addition to their software contributions. Furthermore, the idea of measurement dysfunction is another flaw that should be noted. This concept suggests that by placing too much pressure on developers to reach certain targets for these metrics, companies can unintentionally negatively impact their performance. The following quote by Bill Gates illustrates this idea that one cannot simply rely solely on these metrics: "Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

In the article "Why metrics don't matter in software development", Lowe considers the relevancy of metrics in software engineering. He suggests that the quality and success of a software engineering system should be determined by the customer happiness it creates and the business value it delivers instead of the current scientific method of measurement. This represents a shift from traditional, academic and scientific metric use, to more subjective and difficult to measure ones.

3. Computational Platforms Available

3.1 Overview

Software engineering is one of the few professions today where the professionals develop their own tools. Unsurprisingly, there are many computational platforms available for the analysis of the software engineering process. The importance of efficient data collection systems cannot be understated as it is through data collection that management can accurately assess its systems and make informed decisions. Some of the tools examined in this report use manually collected data, but don't necessarily automate the analysis. Early research on this topic focused on these manual processes, which may appear to be inefficient compared to today's where more modern, big data driven approaches that have been created. On the other hand, other platforms automate the entire process, from collection to analysis to results presentation.

3.2 Description of Platforms

Personal Software Processing (PSP) is a common type of manual data collection and analysis of the software engineering process, which was developed by Watts Humphrey (Johnson, 2013). The platform allows software engineers to improve their performance by tracking their predicted and actual development of code. It involves the developers recording data about themselves and their work processes. They would record their development plans, time estimations of work, bugs encountered and fixed and design checklists, among other things. With proper application, one can become more effective at estimating how long a piece of software will take to write, or reduce the amount of defects in the delivered software. PSP analysis presents clear insights into the engineering process, and makes clear what factors contribute to the process. Key metrics monitored by PSP include: productivity, reuse percentage, defect information and earned value metrics (Humphrey, 1995). PSP allows project managers to review how well their engineers adhere to deadlines, follow plans and complete the work that has been allocated to them. They can also see which engineers are likely to introduce fatal defects into code, and which ones write safer code. From this information they can measure the performance of their engineers. This analysis gives managers a basis with which to compare individual engineers.

However, the manual nature of the data collection allows subjectivity to affect the data. Humphreys alludes to this flaw by saying "it would be nice to have a tool to automatically gather the PSP data. Because judgment is involved in most personal process data, however, no such tool exists or is likely in the near future." Extensive research has been conducted on PSP and the findings have referred to similar flaws. The Collaborative Software Development Laboratory at the University of Hawaii found that the manual nature of PSP meant that there is potential for human error, which could impact the quality of data. As PSP cannot be fully automated, many believe it is not an adequate platform to use in isolation. To address this problem, the University of Hawaii developed the *Leap Toolkit*.

The Leap toolkit advances on the PSP process through automation (Johnson, 2013). Put simply, it analyses the data collected by PSP. It is lightweight, portable and provides reasonably in-depth conclusions. Although Leap still requires the data to be collected manually, it facilitates easier processing of the data. Leap addresses problems associated with erroneous or missing data, which is common when data is collected manually.

The next innovation developed was *Hackystat*, an open-source framework that implements a 'service oriented' architecture. Hackystat is a significant upgrade from PSP and Leap toolkit. It is a platform for automating the collection and analysis of PSP data, well as extending the type of data it can collect and run analysis on (Johnson, 2013). Unlike PSP and Leap, Hackystat automates the entire process leaving engineers with an unobtrusive way of being measured and analysed. The automation of the entire process improves the degree in which analysis can be conducted as engineers aren't required to continuously collect data. Software sensors attached to development tools gather process and product data at regular intervals without interrupting the engineer, send it to a server which conducts quantitative and qualitative analysis. A useful feature of Hackystat is that it can interpret collaboration between engineers.

Moreover, another useful platform for measuring the software engineering process is *GitPrime*. GitPrime uses data from version control repositories to assess the productivity of software engineers, and provide visualisations to managers to better understand the data. The overall objective of the platform is to maximise the productivity of engineers. The platform enables managers to see and track the work of its software engineers by gaining insight into data such as who is the most active in terms of making the most commits, who is making the most errors or spending the most time fixing errors. Other practical features include being able to understand if projects are on track to meet deadlines. Perhaps the most important feature of GitPrime however is its ability to create visualisations for managers that capture all of the data, identify trends and simplify the process of sharing information. A similar software development tool to GitPrime that also presents data in a visual format is *Jira*. Jira is used today to track the progress made by software engineers in developing and implementing features. Managers can follow which engineers complete what tasks which enables tasks in the future to be distributed among engineers according to the each person's strengths. This platform can be integrated with existing development tools, making it very convenient for engineers in practice as well as minimises its obtrusiveness.

Code Climate is a software product that offers analysis tools to aid software engineers in the improvement of code quality. Its focus in on helping software developers to improve the quality of their source code. Code Climate performs automated analysis of software code to identify and flag potential code errors, security vulnerabilities and instances of flawed coding methodology. It can collect and present quantitative and qualitative metrics describing and summarizing various aspects of the code. The platform can be hosted on-site in a company data centre or accessed as a cloud-based service. Today market leading organisations such as PayPal, Adobe and Salesforce use the platform to measure the performance of its software engineers. The platform enables automated analysis of complex codebases written in Javascript, Ruby, Python, PHP and CSS, among others. Code Climate's software can be

integrated with Github and provide test coverage information, technical debt and activity reports.

Codacy is another automated code review tool and is considered to be Code Climate's biggest competitor. According to the founder, Codacy "helps developers optimise around 30% of their code review time" (O'Hear, 2017). One of Codacy's main advantages is its simple and visually appealing interface which allows managers to easily access and understand the most important information. The platform aims to centralise the main problems, alerts and metrics by integrating them into developer's workflow. The platform enables managers to track the work of its engineers and measure their performance by monitoring statistics such as churn, complexity, duplication, number of lines of code.

In any industry or profession, a project will always be measured by time. *Toggl* is a platform that allows software engineers to monitor how they spend their time with the objective of managing time in the most efficient manner. This tool is particularly useful as engineers can learn and prioritise urgent tasks when they gain insights into how they manage their time. Similarly, management can use the data collected to plan out the workday of their engineers and ensure that deadlines are met. Since time is the ultimate measure of efficiency, engineers can be readily compared by how effectively they use their time.

SeaLights is a tool used in software engineering to measure the quality of code and the activity of developers with a particular focus on testing. SeaLights analyses changes that developers make to their code and calculates the risks posed by these changes. This feature allows management to measure the quality of developer's code and provides insights into the level of unnecessary tests that exist within code. In addition, the tool illustrates if there is code that is not being tested by regression tests when it is being executed while the project is in production. Therefore managers can gain an insight into the performance of their testing teams.

It is clear that many software engineering analysis platforms exist today, and the metrics they measure performance by can differ significantly. These computational platforms provide managers and software engineers themselves with valuable insights that help to monitor productivity and efficiency. They also ease some of the pain of manual data collection and analysis, which makes them suited to modern software engineering processes.

4. Algorithmic Approaches

Given the vast amounts of data that is generated in the software engineering process, it is unsurprising that systematic algorithmic approaches have been developed to analyse this data. This area of software engineering is frequently the subject of academic studies. This section of the report will examine some of the approaches that have emerged from this field of study.

4.1 Bayesian Belief Network

Bayesian Belief Nets are one systematic approach to analysing software engineering. Fenton and Neil discuss the idea that software metrics are misleading indicators in describing all of

the different activities undertaken during the process of software engineering in their paper "Software metrics: successes, failures and new directions". They expand on their point to state that metrics fail to act as reliable indicators of future code dependability. In doing so, they highlight the fact that utilizing the Bayesian Belief Network model as a substitute for the issue of imperfect metrics leads to better results.

The evidence obtained from data collection is often subjective and laden with complex dependencies that could give it a different meaning based on context. The Bayesian Belief Network model applies Bayesian statistics to software engineering and in doing so, this volatility and uncertainty can be accounted for and incorporated into the analysis of the data, thereby giving a more accurate interpretation of the data. This allows the software engineering process to be assessed in terms of not only the raw data collected, but the influences the different categories of data have on each other.

Bayesian Belief Nets are graphical networks with associated probability tables. A probabilistic value, computed through empirical analysis, is assigned to each variable in the model. The model collects data such as the defect counts at different stages of testing, the size of complexity metrics at each development phase, and the approximate development and testing effort.

4.2 Multivariate Analysis

Multivariate Analysis is an approach to statistically analysing data that has more than one dependent variable. This algorithmic approach consists of two main subgroups, supervised learning and unsupervised learning.

Supervised Learning is an approach that involves the algorithm generating a function which maps inputs to desired outputs. The term 'supervised' is used as the data scientist guides the algorithm in deciding what conclusions should be drawn from the data collected. The objective is that the algorithm will be able to predict the output for new inputs following this teaching phase. Some supervised methods that are used today to measure the software engineering process are linear discriminant analysis, k-nearest neighbours and logistic regression. They fall into two main categories, classification and regression.

On the other hand, there is no teaching involved in unsupervised learning methods. This approach is based on the idea that algorithms learn to identify complex processes and patterns without human guidance. Examples of unsupervised learning algorithms which are widely used today include Principal Component Analysis, which represents complex data sets in a reduced number of dimensions so that the structure of the data can be more easily visualized; and K-Means Clustering, which divides the data into k distinct groups based on similarities and differences.

4.3 Computational Intelligence

Computational Intelligence (CI) is branch of artificial intelligence which is used for advanced information processing. It refers to the ability of a computer to learn a specific task from data or experimental observations. It is increasingly being used to gain insights into the data collected in the software engineering process. Computational Intelligence is particularly useful in scenarios where mathematical approaches are not adequate to solve a complex problem. Traditionally the three main pillars of CI have been *Neural Networks*, *Fuzzy Systems* and *Evolutionary Computation*. Neural networks provide a framework for machine learning algorithms to work together and process data inputs. They are interconnected groups of nodes that use examples to generate identifying characteristics from the learning material that they process. Fuzzy Systems model uncertain problems that are based on a generalisation of traditional logic. Finally, Evolutionary Computation involves solving optimization problems by evaluating a population of possible solutions that have been generated.

5. Ethical Concerns

The innovation and advancements that have taken place in technology in the modern world has seen the acquisition of data occur at unprecedented levels. Today, organisations gather vast amounts of personal data on individuals, which they use to infer incredibly accurate information about them. Although there are undoubtedly many benefits associated with these advancements for the average person today, such as tailored advertisements online that show relevant items, concerns are growing with regards to how these large multinational companies manage this data. Regulators and data protection authorities have publicly announced in recent years that they are investigating companies such as Google, Amazon and Apple to assess how they manage their data. The danger of mass data collection was evident with the Cambridge Analytica scandal involving Facebook. When regulations are inadequate to control the collection of data, companies and individuals must turn to ethics to determine what is and is not acceptable. This applies to data collection in any scenario, including for the purposes of analysing the software engineering process.

The European Union introduced new data protection regulations known as General Data Protection Regulation (GDPR) to give EU citizens greater control over their personal data (Gordon and Ram, 2018). GDPR, which applies to any organisation that operates in the EU as well as organisations that have customers in the EU, has greatly reformed the area of data privacy by placing legal obligations on organisations to maintain records of personal data and how it is processed. Harsh fines are given to organisations found to have breached elements of GDPR, and so it is essential that management abide by the regulations when measuring the software engineering process.

As a software engineer, it is clear that the job demands wider responsibilities than the application of technical skills. Professional developers must conduct themselves in an ethically and morally responsible way. Principles like confidentiality, honesty and integrity are key to the successful performance of a software engineer (Sommerville, 2015). In spite of this, in order to be able to

analyse the software engineering process, the analysis of software engineers is required and the data that is collected on engineers is equally as important as that collected on citizens. Understandably, some developers have concerns about their personal data being collected with their main reservation being the security of their data, particularly when it is conducted by automated software. If the data is not stored securely, one runs the risk of having information about themselves fall into evil hands. Data collectors, in this case software engineering companies, have the responsibility to ensure that no unauthorised personnel gain access to their data. Companies must decide on an adequate data security system however the cost of implementing these security measures could negatively impact the decision-making process. Ultimately it comes down to a question of the subjective culpability that the company would feel if there was a data breach, which often may not align with the engineers' views.

The way in which data is used is another ethical concern today. One need only look to the European Commission's recent announcement about Amazon's breach of antitrust rules to understand how companies are misusing data for their own gains. The European Commission has accused Amazon of using data on the activity of third-party sellers to benefit its own business, (Espinoz, 2020). This report outlines the ability of companies today to analyse the data collected in the software engineering process thanks to algorithms and tools like those discussed previously. They have the ability to build a very accurate and perhaps invasive profile of their both their customers and engineers. These profiles can be used to decide a range of things, from who gets a promotion to which tasks are assigned to whom at the beginning of a new project. A company must balance the use of automated tools with the subjective element of human analysis to make informed and effective decisions. It is important that management ensure that the analysis of the data is fair and impartial, and that the data is used in an ethical and non-discriminatory manner. It is imperative that data is utilized in such a manner that improves the software engineering process and does not simply allow an organisation to track it more accurately.

This report has argued that it is possible to measure and assess the software engineering process. Nevertheless, the question about the degree to which these measurements are true reflections of the data they interpret still remains. Therefore, we must ask the question: should this data be used? If one takes the view that software engineering is too complex to be accurately captured by the metrics mentioned in this report, then the argument naturally follows that these metrics cannot be trusted to accurately represent the work of software engineers today. The validity of the process can be called into question, which raises an ethical dilemma, as companies will be judging engineers on data that can be argued does not reflect them.

Clearly, there are many ethical concerns when it comes to the collection and analysis of data as well as measurement of software engineers. While there have been some advancements in recent legislation such as GDPR previously mentioned, many of the decisions taken regarding the use of data comes down to the ethical point of view of those in positions of authority.

6. Conclusion

The management teams of companies in the industry today aspire to measure and analyse the productivity and efficiency of its software engineers as well as the software engineering process as a whole. Automated data collection methods are preferred to manual methods due to its unobtrusiveness nature and lower likelihood of producing erroneous data. With regards to the analysis of the data collected, many computational platforms exist which conduct wide ranging analysis. Regardless of the approach taken, ethical concerns will be raised and should be considered at all stages of the software engineering process.

This report has considered the methods that the software engineering process can be measured and assessed, the computational platforms and algorithmic approaches that are available to do so as well as the ethical concerns associated with this activity.

Bibliography

Bhatt, K., Tarey, V. and Patel, P., 2012. Analysis Of Source Lines Of Code(SLOC) Metric. *International Journal of Emerging Technology and Advanced Engineering*,.

Elgabry, O., 2017. *Software Engineering — Software Process And Software Process Models (Part 2)*. [online] Medium. Available at: <https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc> [Accessed 20 November 2020].

Espinoz, J., 2020. *EU Accuses Amazon Of Breaching Antitrust Rules*. [online] Financial Times. Available at: <https://www.ft.com/content/4908995d-5ba4-4e14-a863-bcb8858e8bd2> [Accessed 22 November 2020].

European Commission. 2020. *Antitrust: Commission Sends Statement Of Objections To Amazon For The Use Of Non-Public Independent Seller Data And Opens Second Investigation Into Its E-Commerce Business Practices*. [online] Available at: <https://ec.europa.eu/commission/presscorner/detail/en/ip_20_2077> [Accessed 20 November 2020].

Fenton, N. and Neil, M., 1999. *Software Metrics: Successes, Failures And New Directions*.

Gordon, S. and Ram, A., 2018. *Information Wars: How Europe Became The World'S Data Police*. [online] Financial Times. Available at: <https://www.ft.com/content/1aa9b0fa-5786-11e8-bdb7-f6677d2e1ce8> [Accessed 24 November 2020].

Humphrey, W., 1995. *A Discipline For Software Engineering*. Addison-Wesley.

Johnson, P. M., 2013. Searching under the streetlight for Useful Software Analytics. *IEEE Sofware,* 30(4).

Lowe, S., n.d. *Why Metrics Don't Matter In Software Development*. [online] TechBeacon. Available at: <https://techbeacon.com/app-dev-testing/why-metrics-dont-matter-software-development-unless-you-pair-them-business-goals> [Accessed 19 November 2020].

Nasa.gov. 2003. *2003 Annual Report Of The NASA Inventions & Contributions Board*. [online] Available at: <https://www.nasa.gov/pdf/251093main_The_NASA_Heritage_Of_Creativity.pdf> [Accessed 17 November 2020].

O'Hear, S., 2017. *Codacy, A Platform That Helps Developers Check The Quality Of Their Code, Raises $5.1M*. [online] Techcrunch.com. Available at: <https://techcrunch.com/2017/08/17/codacy/> [Accessed 23 November 2020].

Sommerville, I., 2015. *Software Engineering*. 10th ed. Pearson Education.