MEASURING SOFTWARE ENGINEERING

--Lydia Koilparambil--- #17324124--

November 2019

A REPORT INTO THE MEASUREMENT OF SOFTWARE ENGINEERING PROCESS

Objective:
This report aims to consider how the software engineering process can be measured and assessed in terms of measurable data. An overview of the computational platforms available to perform this work be given. Furthermore, the algorithmic approaches available will be analysed. Ethical concerns surrounding this kind of analytics will also be discussed.

INTRODUCTION

In the past hardware has seen several orders of magnitudes of improvements in technology while software has not. There are several reasons attributed to this. One important argument is that the “Essential problems”- the problems that are inherent to programming i.e. what to build, how to test it etc. aren’t being solved (Brooks, 1987). A lot of a developer’s time is spent on these essential difficulties. This is largely due to the Complexity, Conformity, Changeability and Invisibility problems that come with software engineering. It is suggested that we must simply find “Great designers” (Brooks, 1987). These developers work ten times better than the average programmer. However, how do we do know a program is of excellent quality? Or how can we identify a great programmer? Measurable data is required for this, and this report seeks to explore some of the metrics that are used to measure the software engineering process. By measuring empirical data, it is hoped that trends and patterns can be spotted, and breakthroughs may occur which will make better programs and better programmers.

Many argue that software engineering, can to a large extent, be considered a creative endeavour, with moments of inspiration and moments of low productivity, that makes it hard to quantify or measure. Such arguments may often cite the futility of measuring code lines or hours that a developer works on a project. However, it is advisable to use a useful metric to assess the software engineering process rather than completely ignore the patterns of the process. Arguments that proport the immeasurability of the software engineering process may emphasize measuring happiness, impediments or business value outcomes. Behind all these positions, seems to rest the proposition that software engineering is simply too complex an activity to measure. This report outlines the ways in which both software quality and engineer productivity could be assessed using measurable data. Furthermore, the computational platforms that aid the collection and measurement of data will be outlined along with an analysis of some algorithmic approaches that underpin the metrics discussed. Finally, the ethical concerns surrounding this kind of analytics will be discussed.

1) HOW IS THE SOFTWARE ENGINEERING PROCESS MEASURED IN TERMS OF MEASURABLE DATA?

The software engineering process is the set of activities carried out from the initial product conception to the release of the finished product. This usually involves techniques such as Analysis, Design, Coding, Testing and Maintenance. While there are several different methodologies that one can follow, such as the Waterfall method, Agile Methodology, Rational Unified Process Methodology, the assessment of these can generally be carried using similar metrics. There are several areas of this process that can be measured. Generally, one can measure the software quality and developer productivity.

I. SOFTWARE QUALITY

Software quality metrics generally focus on the quality aspects of the product, process, and project. They can be grouped into three categories following with the software life cycle: end-product quality metrics, in-process quality metrics, and maintenance quality metrics. (Kan, 2014). End - Product quality metrics, which can be assessed in two dimensions, will be outlined in this report.

a) The intrinsic quality of a product:

The intrinsic quality of a product is its most basic functionality separate from business or customer needs. It assesses software on how many "bugs" the software has i.e. functional defects and how long the software can run without it crashing. Formally this can be classified as the Defect Density and the Mean Time to Failure (MTTF). The MTTF metric is commonly used with safety-critical systems e.g. airline traffic control systems. The defect density metric, in contrast, is used in many commercial software systems.

Defect Density:

This measure defects relative to the software size. Finally, the defect rate metric (or the volume of defects) has another appeal to commercial software development organizations. The defect rate of a product or the expected number of defects over a certain period is important for cost and resource estimates of the maintenance phase of the software life cycle.

One can measure this by first calculating the number of defects and representing this as a percentage of the overall size of the program. As failures occur as a result of defects, the number of unique cases of observed failures is used to estimate the number of defects in the software. To obtain the size of the code, one calculates the Lines of Code (LOC).
There are two primary types of SLOC measures:

a) Physical SLOC (LOC) - It is a count of lines in the text of the program's source code which includes comment lines and blank lines unless the lines of code in a section consists of more than 25% blank lines.

b) Logical SLOC(LLOC) - This measures the number of "statements" however their specific definitions are related to specific computer languages i.e. a simple logical LLOC measure for C-like programming languages is the number of statement-terminating semicolons.

The defect rate metrics measure code quality per unit. The aim then is to improve this ratio with each iteration of the product. However, a variation of this metric is to also count the number of defects and ensure that as the software gets more complex, the number of defects is not above a certain threshold. If a new release is larger than its predecessors, it means the defect rate goal for the new and changed code has to be significantly better than that of the previous release in order to reduce the total number of defects. For example, a product in its first iteration may have 20% defects and 1000 lines of code. In the next iteration, if the defect rate decreases to 10%, with the size of the code 3000 lines, this is not favourable to the customer because of the increase in actual defects. It is important to note in software quality engineering, however, consider the customer's perspective. From the customer's point of view, the defect rate is not as relevant as the total number of defects that might affect their business.

MTTF:

This measures the time between failures. However sometimes a fault causes more than one failure situation, and, on the other hand, some faults do not materialize until the software has been executed for a long time with some particular scenarios. Therefore, defect and failure do not have one-to-one correspondence. Gathering data about the time between failures is very expensive. It requires recording the occurrence time of each software failure. It is sometimes quite difficult to record the time for all the failures observed during testing or operation. To be useful, the time between failures data also requires a high degree of accuracy. This is perhaps the reason the MTTF metric is not widely used by commercial developers.

b) Non-Intrinsic Quality:

Customer-reported problems:

The problems metric is usually expressed in terms of problems per user month (PUM):

PUM = Total problems that customers reported (true defects and non-defect-oriented problems) for a time period
÷ Total number of license-months of the software during the period

where,

$$\text{Number of license-months} = \text{Number of install licenses of the software} \times \text{Number of months in the calculation period}$$

PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year. Approaches to achieving a low PUM include:

- Improving the development process and reducing product defects.
- Reducing the non-defect-oriented problems by improving all aspects of the products (such as usability, documentation), customer education, and support.
- Increasing the sale of the product.

This metric is advisable to follow as it allows the customers need to be paramount. Often software engineering products fail, as customers aren't satisfied. Furthermore, it may be more useful than the defects' ratio, as many defects do not affect customers regularly. This metric improves the process as the defects which affect customers most are fixed first.

Customer satisfaction:

Customer satisfaction is often measured by customer survey data via the five-point scale: Very satisfied, Satisfied, Neutral, Dissatisfied, Very dissatisfied. These points can be equated to numbers and various analyses can be made.

II. DEVELOPER PRODUCTIVITY

Developer productivity can be dependent on the quality of the code they write, which was discussed above. Furthermore, other useful metrics include Agile Process metrics such as Lead Time, Cycle Time, Velocity and Open/Close rates

1) Leadtime:
   This is how long it takes to go from an idea to a deliverable software. Lowering the lead time improves responsiveness to customers' needs.

2) Cycle time:
   How long it takes to change a software system and to implement that change into production.

3) Velocity:
   Measured by setting a sprint and for each sprint certain tasks are assigned. This set of tasks are to be completed before the next sprint meeting. This meeting usually takes place every two or three weeks. The number of tasks completed by a team is the velocity. Having a high velocity means that the team is performing well.

4) Open/Close Rates:
   The amount of production issues reported and closed within a specific set of time. The trend is analysed rather than specific numbers.

2) COMPUTATIONAL PLATFORMS AVAILABLE TO PERFORM METRIC ANALYSIS

Over the last decades, several computational platforms have arisen with improvements being made with each iteration. Personal Software Process will be discussed, followed by LEAP and Hackystat which were developed as a response to the shortcomings of PSP.

Personal Software Process

The Personal Software Process (PSP) gives software engineers an analytical personal framework for the tasks they work on. It is comprised of methods, scripts and forms that help software engineers in knowing how to measure and improve their performance. It is introduced with a textbook and a course which can be used in both the academic and industrial setting. The process is devised to be compatible with any programming language or design methodology. Furthermore, it lends itself to most aspects of software development including writing requirements, defining processes, running tests and repairing any defects. The recommended process goal, under the PSP system, is to produce zero-defect products on time and within budget. When used with the Team Software Process (TSP)SM, (Team Software Process and TSP are service marks of Carnegie Mellon University.) the PSP has proved successful in aiding software engineers to achieve their high-performance goals (Humphrey, 2005).

LEAP

Case study research illustrates that the PSP can give software engineers empirical support for improving estimation and quality assurance. However, there still exists a "PSP adoption problem", which may be due to two problems: the high overhead expenses of PSP-style metrics collection and analysis, and the requirement that PSP users "context switch" between product development and process recording. This paper overviews our initial PSP experiences, our first attempt to solve the PSP adoption problem with the LEAP system. A second generation approach uses Leap or another automated tool for PSP-style metrics such as the PSP Studio (Henry, 1997) and PSP Dashboard (Tuma 2000). The platforms use a similar approach to user interaction, in that, dialogue boxes are shown where the user fills in the effort, defect information and size. The platforms also give users access to different types of analyses if requested. These second-generation approaches decrease the overhead cost linked to metrics analysis and metrics collection. However, metrics changes require changes to the software and are hence more complicated than in the first generation approach.

Hackystat

Even after the second-generation improvement on the PSP platform, the adoption rate was low. The biggest barrier was that software engineers didn’t want to switch between doing work and “telling the tool” about what work was carried out. Even if it was only a button press, many users didn’t want to interrupt their flow of work by inputting metric into the tool.

Hackystat solves this by using client-side sensors that work as add-ons to development tools and inconspicuously gather effort, defect, size and other metrics concerning the user's development activities. The system currently includes sensors for the Emacs and JBuilder IDEs, the Ant build system, and the JUnit testing tool. These sensors gather activity data (e.g. which files are under active modification by the software engineer), size data (e.g. non-comment source lines of Java code), and defect data (pass/fail status of JUnit Tests).

The user has to install one or more sensors and then register with a Hackystat server. Once sensor installation and registration are complete, he/ she can return to her development activities. The collected data is unobtrusively sent to the server at regular intervals. A “Daily Diary” created by the tool, is useful for visualizing and outlining Hackystat's representation of developer behaviour. This serves as a foundation for formulating other analyses i.e. the change in the size of a program per week. Hackystat allows the user to define alerts, which are analyses that run periodically over developer data and that specify threshold value for the analysis. If the limit is exceeded, the server sends an email to the developer indicating that analysis has discovered data that may be of interest to the developer. A URL is also sent through which the user can know more about the data in question. Furthermore, the alert mechanism can make them aware of impending problems without them having to regularly “poll” their dataset looking for them (Johnson et al., 2003).

4) ALGORITHMIC APPROACHES TO PERFORM METRIC ANALYSIS

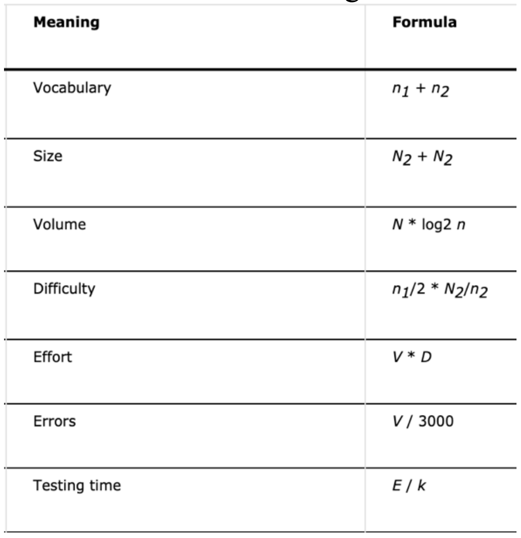
Several algorithms underpin metric analyses. This section will outline a few of these approaches, namely: Halstead's Metrics, Cyclomatic Complexity and Machine Learning solutions.

Halstead's Metrics

The goal of this algorithm is to identify measurable properties of software, and the relations between them (Halstead, 1977). Any programming task consists of selecting and arranging a finite number of program "tokens," which are basic units of code that are identified by a compiler. These tokens can be labelled as operand or operators. The basic measures of this algorithm are:

n_1	= Number of distinct operators in a program
n_2	= Number of distinct operands in a program
N_1	= Number of operator occurrences
N_2	= Number of operand occurrences

Based on these, one can calculate the overall program length, the potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), program difficulty, and other features such as development effort and the projected number of faults in the software. Halstead's major equations include the following:



Meaning	Formula
Vocabulary	$n_1 + n_2$
Size	$N_2 + N_2$
Volume	$N * \log2\ n$
Difficulty	$n_1/2 * N_2/n_2$
Effort	$V * D$
Errors	$V / 3000$
Testing time	E / k

Cyclomatic complexity (CYC)

This is a metric for software quality, developed by Thomas J. McCabe Sr. in 1976. In its simplest form, CYC is a count of the number of decisions in the source code. The higher the count, the more complex the code. It can be used to limit the complexity of code and ascertain the number of test cases required. The following formula to calculate cyclomatic complexity.

$$\text{(CYC): CYC} = E - N + 2P$$

In this equation:

- P = number of disconnected parts of the flow graph (e.g. a calling program and a subroutine)
- E = number of edges (transfers of control)
- N = number of nodes (sequential group of statements containing only one transfer of control)
-

This translates to the number of decisions + one. Binary decisions — such as “if” and “while” statements — add one to complexity. Boolean operators can add either one or nothing to complexity. For instance, one may be added if a Boolean operator is found within a conditional statement.

Machine Learning

Machine learning aims to enable the computer program to learn by itself rather than from pre-programmed instructions from a developer.

Supervised machine learning:

This requires the machine to learn a function that maps an input to an output based on example input-output pairs. The algorithm iteratively predicts the training data, with a human correcting any errors until the algorithm achieves a certain level of performance. Some of the algorithms include Linear regression and Neural networks

Unsupervised Machine learning:

The algorithm used to draw inferences from datasets consisting of input data without labelled responses. There are only input data and no corresponding output variables. The aim is to model the data to learn more about the data.

Reinforcement machine learning:

This allows the machine to learn its behaviour based on feedback from the environment. Some of the algorithms include Q-Learning and Deep adversarial Networks.

3) ETHICAL IMPLICATIONS OF MEASURING AND ASSESSING DATA

It is becoming more and more obvious that Artificial Intelligence paired with Big Data is getting increasingly more powerful. Traditional managerial roles, such as monitoring software engineers and assessing them are today, being aided by computational platforms, as discussed (Susskind and Susskind, 2015). It is argued that these methods will only improve and take over from traditional methods in the future, perhaps leaving little room for human action. This, of course, has both positive and negative effects. However, we must keep ethical concerns in mind when we use methods of measuring and assessing the software engineering process. The key ethical principles when it comes to general data storage can be related in the software realm as follows.

1)Data Collection and Purpose

If employee data is collected to test productivity, then they need to be made aware that data is collected, and explicitly told what it is collected for. For data collected from deployed software e.g. usage statistics, crash reports, the user needs to be made aware, and permission needs to be received.

2)Accuracy and Retention

The data recorded must be regularly tested for accuracy. Measured need to be in place to ensure information is up to date and accurate. Employees and users should be able to view all records that are held on them and be able to make amends. Furthermore, it should not be kept for longer than required.

3) Use Principle

The data needs to be used for the specified purpose and nothing else.

4) Privacy and Security

Data needs to be secure from all external parties and not given to future employers or unauthorized personnel.

5) Human Decision Making

While artificial intelligence and software are increasingly able to do human tasks, decisions regarding promotions, employee turnover, bonuses etc. should be assessed by managers also, and not only subject to computational platforms. This is particularly important as measurement platforms may have errors and thus important decisions cannot be solely based on the results of these computations. A thorough investigation must be done by the manager before decisions are made.

CONCLUSION

This report highlights that while measuring the software engineering process may not be always straightforward, there are still valid and useful ways we can ascertain how good a software is, or how great a programmer is. One can measure software quality as well as developer productivity. The methods are underpinned by rigorous algorithms such as Halstead's Metrics. Furthermore, this area is continuously improving, in particular, due to the advances of machine learning in this field. Based on the algorithms, many computational platforms aid the collection and measurement of data, such as LEAN and Hackystat. Ultimately unless software engineering processes are measured, it will be very difficult to identify how and where to improve. By improving our measurement techniques, we may be able to improve the software engineering process significantly, and perhaps find the "Silver Bullet".

BIBLIOGRAPHY

- Anaxi. (2019). Software Engineering Metrics: An Advanced Guide. [online] Available at: https://anaxi.com/software-engineering-metrics-an-advanced-guide/ [Accessed 5 Nov. 2019].
- Brooks (1987). No Silver Bullet Essence and Accidents of Software Engineering. Computer, 20(4), pp.10-19.
- D. Tuma. PSP dashboard. http://processdash.sourceforge.net/, 2000.
- Fenton, N. and Neil, M. (1999). Software metrics: successes, failures and new directions. Journal of Systems and Software, 47(2-3), pp.149-157.
- Halstead, M.H. (1977) Elements of Software Science
- Humphrey, Watts. (2002). Personal Software Process (PSP)
- J. Henry. Personal Software Process studio. http://www-cs.etsu.edu/soften/psp/, 1997.
- Javdani, T. , Zulzalil H., Ghani A., Bakar A.,Sultan A.(2013). On the Current Measurement Practices in Agile Software Development
- Kan, S. (2014). Metrics and models in software quality engineering. [Place of publication not identified]: Addison-Wesley.
- P.M. Johnson et al., "Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined," Proc. 25th Int'l Conf. Software Eng. (ICSE 03), IEEE CS, 2003, pp. 641–646.
- Susskind, R. and Susskind, D. (2015). The future of the professions.