

Measuring Software Engineering Report

Adam Lysaght

18324893

Table of Contents

Introduction.....	2
How can Software Engineering be measured?.....	3
What platforms can be used to gather and process the data?.....	5
What algorithms can we use to analyse the data?.....	7
Ethical Discussion.....	10
Conclusion	11
References.....	11

Introduction

Software Engineering

In order to examine and assess how Software Engineering is measured, we must first define what we mean by Software Engineering. Today, SEVOCAB defines Software Engineering as the “application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software” (SEVOCAB, 2016). The discipline was defined during the 1960s through the work of pioneers such as Anthony Oettinger, Douglas T. Ross, and Margaret Hamilton. However, the industry struggled during the 1960s, 1970s and 1980s, with many projects running over budget and schedule. This is where the need and desire to accurately record and measure the software engineering process came from.

Software Measurement

Software measurement is the “continuous practice of defining, collecting, and analysing data on the software development process and its products in order to understand and control the process and its products” (van Solingen & Berghout, 1994). This practice stemmed soon after the birth of software engineering, during the time known as “the software crisis” and was routinely used during the 1960s. However, it was not officially defined in literature until 1971, when Akiyama made “the first attempt to use metrics for software quality prediction” (Fenton & Neil, 1999) using a regression model based on the number of defects per KLOC (thousands of lines of code). This was the starting point in the practice of Software Engineering measurement.

Since then, software measurement has evolved from size metrics alone and includes much more complex methods of measuring software, involving significant amounts of data and complex algorithms. I will be examining these methods in this report, with consideration of how the data is captured and processed, the algorithms used in these methods, and finally reflecting ethically on the measurement of software engineering in the age of big data.

How can Software Engineering be measured?

This has been an important question in the software industry for many decades. There are a variety of software metrics a manager may use when evaluating the quality of the software project. These can be broken down into two broad groups – product metrics and process metrics. I will be examining each group in turn, giving examples of, and evaluating the effectiveness of several methods. Finally, I will give my opinion on the software measurement process.

Product Metrics

Here we examine the deliverables of the project. These metrics look at issues such as software size, complexity, quality, and reliability. Examples include the following:

Lines of Code (LOC) / Thousand Lines of Code (KLOC)

This is a simple solution for measuring both programmer productivity and program defects. It was widely used during the earlier stages of software measurement, however today it is rarely used due to the number of issues when measuring by LOC. For example, a manager could assess who is the most productive programmer in the team by comparing the number of lines of code each has written. This sounds plausible in theory, but in practice it is not a viable measure. This is because more code does not mean better code. Code may include 'dead code' as well as comments, different coding conventions and programming style between two programmers. Furthermore, it cannot be compared between programming languages due to the different syntax in programming languages, such as Java and Haskell. As Bill Gates once said, "Measuring programming progress by lines of code is like measuring aircraft building progress by weight". Thus, it is not practical to measure programming performance with this metric, though I believe defects per KLOC could still be used as a broad measure of program quality.

Process Metrics

This category examines issues during the software development process such as fault detection, bug fixes and meeting deadlines. Examples include the following:

Throughput

This measure indicates the total value added work output by the team. The advantage of this metric is it can be measured throughout the development process to monitor performance. It is often measured using tickets completed. These tickets are smaller objectives that must be fulfilled in order to complete the wider project task, such as "Fix the Login page". Management can easily identify when the team is facing issues with the current tasks when current throughput has dropped against average throughput. This helps resolve issues before they become too big. I believe this is an important and useful measure to track progress in a software team. The team will feel a sense of accomplishment when they see tickets being completed and throughput increasing. However, it does require close and constant monitoring from management to ensure the team is working at its maximum capacity.

Code Churn

Code churn is when code is deleted and rewritten after it has been committed. This is a normal occurrence in software development, however unusually high or low levels of code churn can be a worrying sign. Code churn is usually measured to assess team or individual performance. A software engineer will regularly go back to previously committed code to refine and improve it, this is okay and a positive sign. However, if it happens too regularly, it is a worrying sign for management as it decreases the overall efficiency of the team. Normal code churn levels are generally between the 13-30% range (Pluralsight, 2019). The reason why code churn is a useful measure for management to measure performance is it can identify problems quickly. For example, if a team is constantly returning and changing code, it can mean that the requirements are unclear, or they are facing a difficult problem. Management can then intervene to solve these problem to help the team get back on track. On the flip side, if code churn is very low (almost 0%), it could mean employees are demotivated to improve the software or may be burnt out. Management can solve this problem by giving the team a break from that particular project. Therefore, it is evident code churn is an effective measure in monitoring the performance of a team. Though there is the potential of stifling creativity if projects are shut down when code churn is seen as too high.

Reflection

As seen above, there are a vast number of ways a manager can measure the performance of the software project. Many of these metrics will not suit the type of project that is being undertaken. Furthermore, the measurement may not give useful results which can be acted upon and improve the software. I believe the Goal-Question-Metric approach can be utilised to solve this problem. The method uses goal-oriented software engineering where a goal is defined, then questions are drawn from this goal, and finally metrics are defined to provide the information to answer these questions (van Solingen & Berghout, 1994). This ensures that metrics are useful and directly contribute to achieving the team's goals. If small steps such as this are taken in software teams, they can ensure the software project is measurable throughout the development cycle and the results are relevant, useful, and insightful.

What platforms can be used to gather and process the data?

An important step in the software measurement process is choosing a suitable method to collect and process the data. There are several frameworks that help us do this. I will be examining traditional methodologies such as Personal Software Process, and more automated solutions such as Hackystat and Timeular.

Personal Software Process

The Personal Software Process (PSP) was developed in 1993 by Watts S. Humphrey and acts as a management framework for the software development process. As the name suggests, it aids individual engineers in the areas of estimating and planning, meeting commitments, managing quality and reducing defects (Pomeroy-Huff, et al., 2009). One of the main principles is that the engineer is self-measurement; data should be collected by the person who are going to use it. Thus, after every session of coding the individual engineer will record what they worked on throughout the day. They will records metrics such as time spent, defects corrected, and products produced. There are three main stages of the Personal Software Process, PSP0, PSP1 and PSP2. Each stage involves a planning, development, and post-mortem phase. Each stage builds on the previous stage's data collected until they complete the final stage where the engineer will have a significantly greater planning capacity and insight into their previous work.

This platform is a simple yet effective solution to collection data for software measurement. However, the main disadvantage is the engineer will have to constantly update the data themselves. This is time consuming and may result in collection biases as the individual engineer may only select favourable data. Thus, more automated solutions have been developed which remove these disadvantages.

Hackystat

Hackystat is an open source framework for the automated collection and analysis of software engineering product and process data. It differs from the Personal Software Process as the data is automatically collected. It works through the developer attaching software plugins to their development tools. These sensors will automatically send data to the sensor base where daily reports are created. Furthermore, the user can see trends where daily data is analysed and a report is created, as seen in Fig. 1 below.

The main advantage of Hackystat over the PSP is it removes both data collection and analysis; it is done automatically. In a survey of 78 students from the University of Hawaii, after utilising the Personal Software Process, 72 of them abandoned PSP as they felt "it would impose an excessively strict process on them and that the extra work would not pay off" (Johnson, 2003). When compared to Hackystat, over 70% of respondents felt that Hackystat would be feasible in a software development setting (Johnson, 2004). This highlights the advantages of Hackystat; it takes the effort out of software measurement.

Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (4)	63.0	1.6	6.9	287.0	3497.0	3.2	9.0	25.0	25.0
duedates-shinshina (5)	61.0	1.5	7.9	1321.0	3252.0	25.2	59.0	194.0	274.0
duedates-ekala (8)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0

Fig. 1, (Hackystat, 2009).

Timeular

Timeular is an eight sided tracking dice that records the time spent on each activity throughout the day. The user can set each side with an activity they want to record, for example Emailing. Each time the user switches what they are working on, they flip the dice. The data is then stored and processed, and the user can then see a summary of the time spent on each activity on the app, as seen in Fig. 2 below.

Although this platform does not directly cater for software measurement, I believe it can be utilised as efficient solution for managing a software engineer's schedule throughout the day. A software engineer may have many different projects they are working on currently. In each of these projects, there will be a number of tasks to be completed. Timeular can help manage these tasks and prioritise time spent on the most important tasks and projects. It can also help focus the engineer on an individual task, as they will be aware of their current task and can block out other tasks and distractions. At the end of the week, Timeular provides a weekly recap. This can allow engineers to review where they spent their time and improve their workflow going forward.

It is evident Timeular is an innovative solution to time management. It removes the need for timesheets and pen and paper to track schedules. Although it does not contain metrics such as code churn or defects corrected like the previously discussed platforms, in my opinion it is the best measure at managing the software engineer's time.

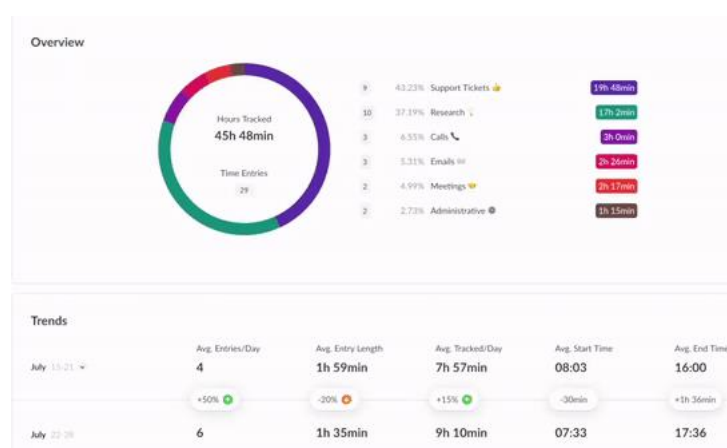


Fig. 2, (Timeular, 2020).

What algorithms can we use to analyse the data?

At this stage, the management team have selected which metrics they want to track. They have chosen a platform to gather and process the data of their engineers. They are then faced with the challenge of analysing the data collected to identifying patterns and to produce useful results. Many of the platforms I examined do this already for us. In this section I will be examining the algorithms that are used to accomplish this task, performing analytics to answer questions we may or may not have asked.

Computational Intelligence

The ability of a computer to learn from the data and make smarter decisions as a result is a useful algorithm when measuring software engineering. Once the software data has been gathered, management will be looking to ask questions regarding the efficiency, usability, correctness, reliability, and maintainability of the software system. This is the basis of McCall's software quality model, where these measures (and more) are used to assess whether the software meets a certain quality standard (McCall, et al., 1977).

Building upon this model, a Software Quality Decision System can be designed using computational intelligence. Pedrycz et al. proposed such a system (as seen in Fig. 3 on the right) utilising granulation from fuzzy sets and rule derivation from rough sets to assess software quality (Pedrycz, et al., 1998). As we can see, processed data is fed into the system. From there, a decision table is created with software deployment rules drawn up. A threshold is selected based on experience gained from previous projects, and this is used in combination with the decision table to assess the overall program quality. Several measures, such as efficiency, understandability, and human engineering of the software system are assessed. These measures can help decide if the software system is able to be deployed or not.

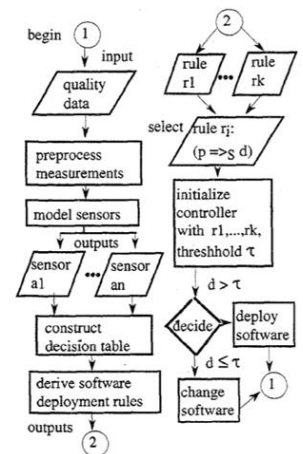


Fig. 3 Steps in Quality Decision System

Although this study is only a small example of utilising computational intelligence in the field of software measurement, it clearly shows the potential of the model. Using computational intelligence to answer important questions in the software development process such as “What is the quality of the system, and is the system ready for deployment?” is no doubt a powerful tool which will only improve in the future.

Gamification

One interesting proposal in the field of software measurement is the gamification of the software development process (Passos, et al., 2011). This proposal takes the concepts from game design, such as achievements and feedback and applies them to the field of software engineering. The main idea is achievements are set, just like in a game, for completing certain tasks. For example, the team will earn a gold medal (virtual or even physical) for 100% test code coverage, a silver for 75%, and a bronze for 50%, as seen in Fig. 4 below. This process of converting metrics to earnable achievements will undoubtedly team motivation and performance within the team, especially if there is competition between teams.

Furthermore, additional gaming concepts such as a profile creator can be brought to the software development process. Here, developers could model their own skills and actions, list their achievements in recent projects, and list their current quests (or tasks) they are working on. This would act as a model of the developer, and act to encourage continuous progress and improvement.

As an experienced gamer, I believe this proposal has great potential in the software development industry. An algorithm to convert metrics and other data in the software engineering field to a gaming setting would add a new dimension to the development process. I believe it could act as a motivator for improvements in efficiency, production, and quality, all through a competitive and enjoyable interface.
















Clockwork Developer		
Project	Developer	Medals
Project I	Developer A	
Project I	Developer B	-
Project I	Developer C	 
Project II	Developer D	
Project II	Developer E	-
Project II	Developer F	
Project III	Developer G	
Project III	Developer H	
Project III	Developer I	-
Project IV	Developer J	 
Project IV	Developer K	 
Project IV	Developer L	 
Project IV	Developer M	 

Fig. 4, (Passos, et al., 2011)

Bayesian Belief Networks (BBNs)

A BBN is a graphical network that represents probabilistic relationships among variables (Neil & Fenton, 1996). This can be applied to the software measurement space, to make use of the data collected and draw inferences from this data. As seen in Fig. 5 below, the nodes represent uncertain variables, and the arcs represent the relationship between the variables. Thus, we can map the measures collected in previous steps as a BBN. Then we can examine the relationships and raise interesting questions regarding changes to these metrics. For example, what is the impact in Design effort in response to an increase of New defects inserted?

I believe this algorithm can have an extremely positive impact, especially when communication with non-technical stakeholders. A shareholder may have no idea why code churn of 50% is too high. However, explaining it to him/her through a BBN - stating that a reduction in code churn of 20% will decrease complexity by 50%, is an effective way to communicate with non-technical shareholders. The main challenge is constructing the BBN in the first place and mapping the relationships between variables, however, once that step is achieved, the BBN will be very useful at analysis and exploring patterns in the data.

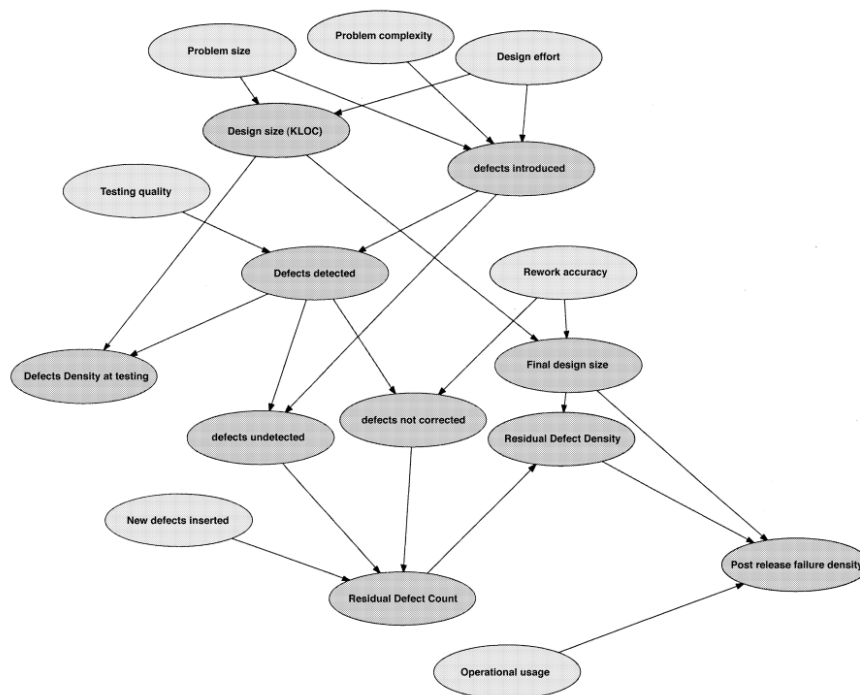


Fig. 5, (Fenton & Neil, 1999)

Ethical Discussion

In the age of big data and data analytics, one would have thought that there was a clear line drawn between what data could be measured and collected and what could not. However, the question of data collection remains a decisive topic, especially in the relatively new field of software engineering. Employers will maintain the position that is their right to collect data about their workers. But at what stage does it go too far? Is all the data being collected to improve performance or are the privacy of the workers being exploited? Where do we draw the line?

Let us examine the previously mentioned data collection methods, PSP and Hackystat. They appear to be at opposite ends of the ethics debate. In PSP, the developer directly controls which data to track and collect. While with automated solutions such Hackystat, will collect significantly more data. There are two main ethical problems with these methods.

Personal Biases

Under the PSP, as hinted by the name, the individual developer plays a decisive part in the selection, collection, and analysis of the data. They are in control of what data they want to track and share. From a data ethics perspective, this sounds ideal as they are not exposing too much of their data. However, it presents other ethical issues such as the reliability and transparency of the metrics. The engineer can easily present themselves in a misleading fashion. For example, not tracking defects per KLOC and instead tracking defects solved. The code may have an abundance of existing defects, but it appears to others that many defects have been solved and the code is functioning well. This may present ethical issues when presenting measured data to the client during handover. It may appear all is working well but there could be underlying issues.

Volume of Data

Automated platforms such as Hackystat remove the problem of personal biases from the picture. Unfortunately, they bring a new issue – the abundance of data collected. For arguments sake, we will take the extreme case where everything is monitored on the worker's computer. Does the engineer's manager need to know at what time he takes his coffee break at? Is it necessary to record how many times he visits Stack Overflow? The employer will likely maintain the belief that these measures can help increase efficiency. However, it is unlikely that all the data collected will be utilised for its intended purpose – to improve productivity, and instead be used for authoritarian tracking. Measurement tools previously mentioned such as Timeular pose obvious problems relating to this issue. In essence, it is 24/7 monitoring and management may ask why the dice was left on the Email side for several hours. This will no doubt be a cause of conflict.

As aforementioned in this report, I believe all software measurements collected must have a specific purpose and answer a particular question. The Goal-Question-Metric model may be the best option we currently have in the field of software measurement until legislation catches up. This will ensure all measurements collected are utilised for the correct purpose,

and both employees and clients will have peace of mind of the ethical concerns regarding software measurement.

Conclusion

Software Measurement has been of major importance in the field of software engineering since its birth. There have been significant developments in the platforms we use to collect data and analyse the data over the past several decades. These developments have helped validate the field on software engineering and elevate it to its renowned and distinguished position it is in today. Perhaps progress has moved too fast, and the legislation needs to catch up to address the ethical issues that the collection of large amounts of personal data poses. We know the benefits of software measurement, but can firms deal with the ethical problems it causes? This will be an important question in the field in the coming years.

References

Fenton, N. E. & Neil, M., 1999. Software metrics: successes, failures and new directions. *The Journal of Systems and Software*, 47(2), pp. 149-157.

Hackystat, 2009. *Hackystat in a Nutshell*. YouTube.

Johnson, P. M., 2003. *Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined*, Hawaii.

Johnson, P. M., 2004. *Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH*, Hawaii: s.n.

McCall, J. A., Richards, P. K. & Walters, G. F., 1977. *Factors in Software Quality*, New York: Rome Air Development Center.

Neil, M. & Fenton, N. E., 1996. *Predicting software quality using Bayesian belief networks*. London, Center for Software Reliability.

Passos, E. B., Medeiros, D. B., Neto, P. A. S. & Clua, E. W. G., 2011. Turning Real-World Software Development into a Game. *Games and Digital Entertainment (SBGAMES)*, Volume 2011, pp. 260-269.

Pedrycz, W., Peters, J. F. & Ramanna, S., 1998. *Design of a software quality decision system: a computational intelligence approach*. Ontario, IEEE.

Pluralsight, 2019. *INTRODUCTION TO CODE CHURN*. [Online]
Available at: <https://www.pluralsight.com/blog/tutorials/code-churn>
[Accessed 20 November 2020].

Pomeroy-Huff, M. et al., 2009. *The Personal Software Process (PSP) Body of Knowledge, Version 2.0*, Carnegie Mellon.

SEVOCAB, 2016. *Software Engineering*. [Online]
Available at: https://pascal.computer.org/sev_display/index.action
[Accessed 18 November 2020].

Timeular, 2020. *The time tracking app you'll love*. [Online]
Available at: <https://timeular.com/product/pro-subscription/>
[Accessed 24 November 2020].

van Solingen, R. & Berghout, E., 1994. The Goal/Question/Metric Method: a practical guide for quality improvement of software development. In: London: McGraw-Hill Publishing Company, p. 19.