Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CS3012 Software Engineering

Measuring Software Engineering Report

Kazuhiro Tobita
18308725

November 27, 2020

Table of Contents

1. Introduction
2. Measurement and Assessment
    2.1 Why do we measure software development?
    2.2 What can we measure software development?
    2.3 How do we assess the data?
3. Analytical Practices
    3.1 PSP (Personal Software Process)
    3.2 Hackystat
4. Algorithmic Approaches
    4.1 McCabe Cyclomatic Complexity
    4.2 Halstead's Software Metrics
5. Ethics Consideration

1. Introduction

As business leaders keep to seek a new edge, software measurement plays an increasingly important role in understanding and controlling software development practices and products. Consequently, the measures we use must be valid. In other words, measures must represent those attributes accurately they purport to quantify. Therefore, validation is critical to the success of software measurement. Unfortunately, however, it is a fact that there is a large gap between software quality frameworks and standards such as ISO 9001/9000-3; the requirement that quality measurement should be carried out and the guidelines on how to carry out the measurements.

The objective of this paper is to deliver a report that considers how the software engineering process can be measured and assessed in terms of measurable data, an overview of some analytical platforms this work, the algorithmic approaches available, and the ethics surrounding this kind of analytics. The achievement of the measurements is to make incremental improvements to processes and production environments and to have genuinely data0driven software development.

2. Measurement and Assessment

2.1 Why do we measure software development?

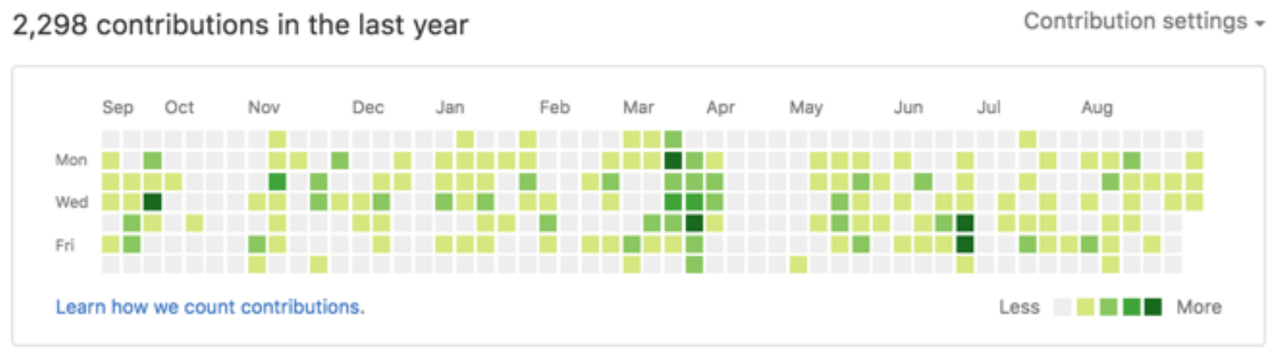
Measuring software engineering is a tremendously complex process, because there is not a conventional way of measurement; there are many factors to evaluate the validity of codes. A "good" programmer, however, should deliver high-quality software code with minimal bugs; any tiny bugs can be inevitable and what issues is how they control them. Furthermore, according to Kaner et al., the higher the stakes associated with a measurement, the more essential the validation (p.5). That is why it is crucial to measure its efficiency, reliability and the stability of the product for facilitating evaluating project status and enhancing individual/team performance, that is, measurement is empirical.

2.2 What can we measure software development?

Although some of the software metrics which are currently being measured in the industry, Kienitz lays out the validation criteria:

Code quality. It can be derived into quantitative and qualitative criteria. The former ones measure how big and complex the program is, such as lines of written source code and bug rates, which is the average number of bugs that come up as new features are being deployed.

Performance. It includes the velocity of programmer's coding. The amount and frequency of commits are also indicative of how active a developer is in a project. If people see Github, for example, one can grasp a graphical perspective of a user's commit activity.



Security. This metrics measure the inheritance safety and stability of software products; it ensures there are no unauthorised fixes in the code when it is handed over the customer.

Usability. From a design point of view, it is vital whether the service is practical and user-friendly, since all software products are built for an end-user.

2.3 How do we assess the data?

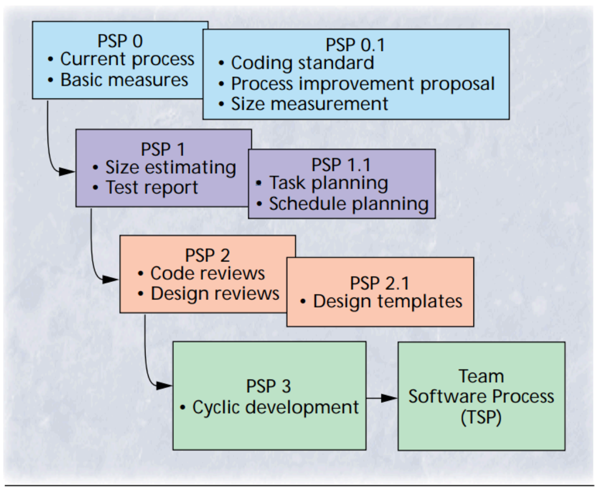


Figure 1. PSP process evolution.

There seems a great deal of factors affecting software engineers and the way they work, and selecting one way of measuring over another could have unexpected side effects that could actually impact on any company negatively. With a simple Web search, developers can compare the costs and benefits of certain practices to determine which is worth the cost. In another way, they can benchmark two different practices to select the better approach. There are many platforms and algorithmic approaches to analysing this data, more on those in the following sections of the report.

3. Analytical Practices

3.1 PSP (Personal Software Process)

In 1995, Watts Humphrey created a platform to adapted organisational level software measurement and analysis techniques to the single developer along with a one-semester curriculum in his book, *A Discipline for Software Engineering*. These techniques are named the Personal Software Process (PSP); It is designed to help developers better understand and enhance their performance, and these archives are pursued by collecting size, developing time, and defect data on an initial set of software projects and performing various analyses on it.

According to *The Personal Software ProcessSM (PSPSM) Body of Knowledge, Version 2.0*, the basic PSP process has three phases; planning, development and postmortem. Firstly, developers must plan the execution of a project, and they must write a design document. Before moving forward, this design needs to be reviewed. Secondly, we get to the coding part; The code needs to be reviewed as well, and only after all this we get to testing. The last step is to write postmortem describing the flow of the project, what went well or wrong, and general logs of every step. Therefore, a software engineer needs to keep track of their own performance, note when they did not meet a deadline, or there were defects in the code, they also need to record the size of the project, the timeline and the lines of code.

Ferguson et al. shows that PSP follows an evolutionary improvement approach; an engineer learning to integrate the PSP from the lowest level PSP0 and more complicated material until the final level PSP3. Each level introduces new elements and has detailed scripts checklists and templates to guide the engineer. Humphrey encourages developers to personalise tases scripts as they get a better understanding of their individual strengths and weaknesses.

- PSP0, PSP0.1

PSP0 - Developers essentially follow their current practices, learning some basic PSP techniques. This level introduces how to log each compiler and test defect and how to record development time. These measures is utilised as a benchmark for assessing improvement.
PSP0.1 - Adds the process by improving a size measurement and the development of a personal process improvement plan (PIP). In the PIP, the engineer writes ideas down for improving his own process.

- PSP1, PSP1.1

PSP1 - Based on the historical baseline data collected in PSP0 and PSP0.1, the engineer estimates how extensive a new program will be and determines the accuracy of the estimate.
PSP1.1 - This level adds resource and schedule estimating and earned-value tracking. Accumulated data from previous projects is used to judge their progress as they finish some tasks early and other late.

- PSP2, PSP2.1

PSP2 - Adds two new phases: design review and code review. Defect prevention and removal of them are the focus at the PSP2. Developers also construct and use checklists for design and code reviews.
PSP2.1 - In this level, engineers learn how to design a specification and to prevent defects.

- PSP3

The highest step level, software engineers become fully conversant in PSP. This level covers design verification techniques and methods for adapting PSP to developer’s working environment. By applying the techniques and procedures learned during the course, engineers see how to apply PSP to large-scale software development. Figure 2 shows the spiral-like strategy of PSP level 3 for developing program modules of up to several thousand lines of code. This cyclic process builds on several well-known software engineering principles; First, by using abstraction and modular design concepts, engineers are better able to produce clean designs and to capitalise on reusable parts. Second, this cyclic development strategy follows the common practice of building large programs through a family of progressively enhanced versions. Finally, the process incorporates the divide-and-conquer strategy of Barry Boehm's spiral model for minimising risks by attacking complex problems a step at a time.

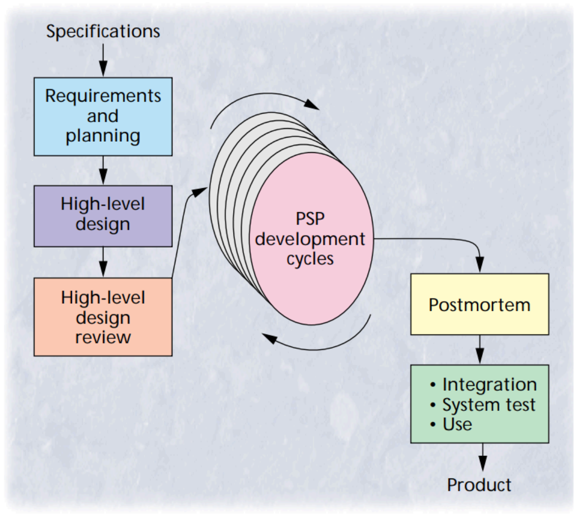


Figure 2. PSP 3 process.

3.2 Hackystat

Hackystat is an open-source framework for the automated collection, visualisation, analysis, annotation, implementation, and dissemination of software development processes and product data developed by Philip Johnson. It is no longer inactive, but it is open-source, with it is code being widely available on GitHub. The objective of the platform is to provide an extendable mechanism which can reduce the overhead associated with collection of data about software engineering.

Kou et al. (2002) describes Hackystat implements a service-oriented architecture in which sensors attached to development tools gather process and product data and send it to a server every 30 seconds in Hackystat, which other services can query to build higher-level analysis. That is, the sensors record the developer's activities even if there is any tiny change; For instance, an action such as renaming the variable from A to B, opening a file, executing a program is recorded with timestamp, runtime, the name of tools and resource in XML format.

Hackystat does both client and server-side data collection. Another feature is that the data collection is unobtrusive. It does not make assumptions about connectivity and Hackystat client-side instrumentation locally caches any data collected while a developer works offline. Hackystat also supports group measurements and can track when multiple engineers are working on the same project.

4. Algorithmic Approaches

4.1 McCabe Cyclomatic Complexity

Thomas J. McCabe describes that the considerations and control statements are the ones that add complexity to a programme in his paper, "A complexity Measure" in 1976; this is one of the most famous, authoritative theses about software metrics. The objective of this mythology is to calculate a cyclomatic complexity, which is a software metric used to indicate the complexity of a program. It is computed using the control flow graph of the program: the nodes of the graph corresponding to invisible groups of commands of a programme and a directed edge connects two nodes. Kaur et al. proposed the formula of the cyclomatic complexity of a function is:

$$M = E - N + 2P$$

Where,

M = McCabe's cyclomatic complexity

E = the number of edges in the control flow graph

N = the number of nodes in the control flow graph

P = the number of connected parts of the flow graph

Function points are a unit of measure as if people use hours to measure time; if the programs contain a high McCabe number, for example less than 10, it would be difficult to understand, and it has a higher probability of containing errors. The formula itself is simple, and this metric is applicable for calculating the size of computer software, estimating costs and project management, measuring quality and setting functional requirements. However, there has been much debate over the pros and cons, because the paper in IEEE TSE does not mention the definition of the complexity and what is a cyclomatic complexity.

4.2 Halstead's Software Metrics

After the argument, in the next year, Maurice H. Halstead issued "Elements of Software Science". This metrics count tokens and determines which are operators and operands; He suggests that complexity is found by measuring the number of linearly independent paths throughout the program, with his words, "Software complexity is an estimate of the amount of effort needed to develop, understand or maintain the code." Namely, the higher the number, more complex the code would be. The following base measures are used:

n1 = Number of distinct operators.

n2 = Number of distinct operands.

N1 = Total number of occurrences of operators.

N2 = Total number of occurrences of operands

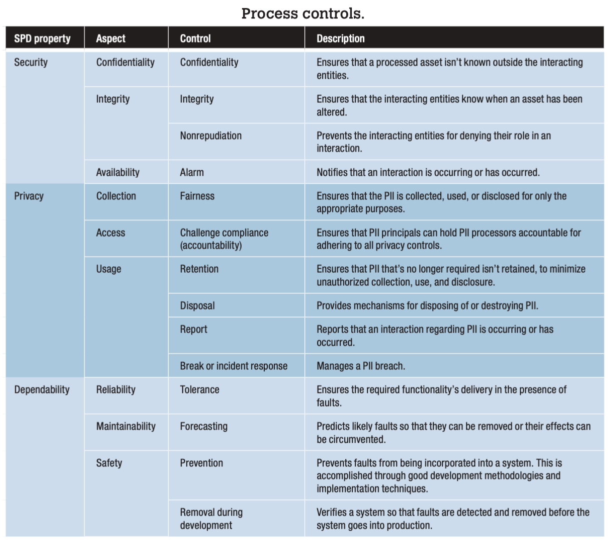
Using these notations, Kan defined the following terms:

- *Vocabulary* (n) = n1 + n2 : The sum of the number of operands and operators.
- *Length* (N) = N1 + N2 : The sum of the total number of operators and operands in the program.
- *Volume* (V) = Nlog2(n) : The information contents of the program measured in bits. The computation of v is based on the number of operations and operands in an algorithm.
- *Level* (L) = V* / V :The inverse of the error proneness of the program, that is, a low-level program is more likely to have more errors than a high-level program.
- *Difficulty Level* (D) = V / V* : This is also known as error proneness and the program is proportional to the number of unique operators in the program.
- *Effort* (E) = V / L : It is proportional to volume and the difficulty level.
- *Number of delivered Bugs* (B) = V / S* : It correlates to the overall complexity of the software.

Where V * is the minimum volume represented by a built-in function performing the task of the entire program, and S * is the mean number of mental discriminations (decisions) between errors (S * is 3,000 according to Halstead).

Halstead's work has had a significant impact in software measurement. He concluded that the code complexity of a program soars once the level decreases and volume increases. However, his studies have received much criticism, including methodology, derivations of equations, human memory models.

5. Ethical Consideration

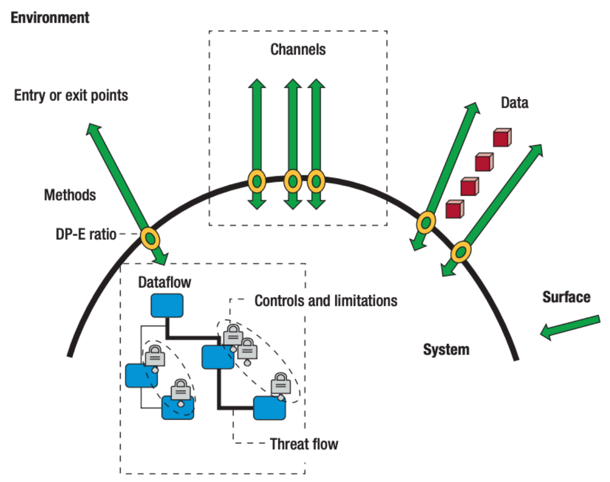


Process controls.

SPD property	Aspect	Control	Description
Security	Confidentiality	Confidentiality	Ensures that a processed asset isn't known outside the interacting entities.
	Integrity	Integrity	Ensures that the interacting entities know when an asset has been altered.
		Nonrepudiation	Prevents the interacting entities for denying their role in an interaction.
	Availability	Alarm	Notifies that an interaction is occurring or has occurred.
Privacy	Collection	Fairness	Ensures that the PII is collected, used, or disclosed for only the appropriate purposes.
	Access	Challenge compliance (accountability)	Ensures that PII principals can hold PII processors accountable for adhering to all privacy controls.
	Usage	Retention	Ensures that PII that's no longer required isn't retained, to minimize unauthorized collection, use, and disclosure.
		Disposal	Provides mechanisms for disposing of or destroying PII.
		Report	Reports that an interaction regarding PII is occurring or has occurred.
		Break or incident response	Manages a PII breach.
Dependability	Reliability	Tolerance	Ensures the required functionality's delivery in the presence of faults.
	Maintainability	Forecasting	Predicts likely faults so that they can be removed or their effects can be circumvented.
	Safety	Prevention	Prevents faults from being incorporated into a system. This is accomplished through good development methodologies and implementation techniques.
		Removal during development	Verifies a system so that faults are detected and removed before the system goes into production.

There are many ethical concerns around measuring software engineering mostly because of the massive amount of data that is being gathered. That is to say, a practical methodology of software measurement is so vital in risk management during development, and developers must intensely focus on the security and privacy of a software. Indeed, Prasad et al. refers to Ford, for instance, set a research methodology for modelling automotive mobile security, privacy, usability and reliability (SPUR). This methodology analyses the software functionality on the standard of these properties and assigns qualitative values to each one; Ford applied software for their antilock braking system and a valet key.

Hatzivasilis et al. propose a multi-metric methodology to evaluate and quantify software system security, privacy, and dependability (SPD). Although their SPD evaluation is hinged on the Attack Surface Metric (ASM), they has enhanced the security specification and the basic surface calculation following the European Data Protection Directive 95/46/EC and ISO/IEC standards 29100 and 27018.



This figure illustrates the SPD methodology. The interaction points are located at the boundaries of the system's surface, and it separates the system from the environment.
Also, they predict the damage potential-ratio, namely DP-E, which stands the effort of performing an attack and exploiting a dataflow in tandem, since these two factors are mostly related in a real attack scenario. A hacker, for example, might be willing to spend effort in obtaining a

right of access, and if they success that, it would cause more critical damage.

The SPD values are composed of a triple vector representing the degree of asset protection under the controls or the reduction of the threats' impact. Although Each value ranges from 0 to 100, a maximum SPD value, which means <100, 100, 100>, does not always guarantee perfect protection. That is, it only shows that the potential safety based on existing set such as CVE (Common Vulnerabilities and Exposures) bulletins and US-CERT (US Computer Emergency Readiness Team) advisories.

In order to analyse the contribution of vulnerabilities, disclosure, or exposure, they carry out a calculation based on the malicious attack potential following ISO/IEC 15408, and set 5 factors as follow:

- *Elapsed time*. The time required to identify and exploit the limitation, measured for a certain period.
- *Specialist expertise*. It means whether an attacker has professional knowledge or not.
- *Knowledge of the target's design and operation.* It expresses, for example; their motivation is for obtaining restricted, sensitive, or critical information.
- *Window of opportunity*. An attacker might require considerable access route to exploit the threat without any detection successfully.
- *Resources*. It is whether they need a highly specialised IT hard/software or other equipment to hack.

In order to guarantee software privacy and protect software itself from cyber attack, one of the proposed solution will be offered as a service for embedded system development and system control. An application, such as blockchain, e-health and smart grid can leverage the protection level of existing services. Furthermore, Internet-of-Things architectures and cloud services will be available for software measurement and new practices for interaction. The SPD methodology would help developers gauge interconnected embedded software that communicate information to the cloud.

References

Ferguson, P. (1997), *Results of Applying the Personal Software Process*, available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=589907 (accessed 20 November 2020).

Hatzivasilis, G., Papaefstathiou, I. and Manifavas, C. (2016), *Software Security, Privacy, and Dependability: Metrics and Measurement*, IEEE Software, pp. 46-55, available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7436657 (accessed 20 Nov 2020).

Jorgensen, M. (1999), *Software quality measurement*, Department of Informatics, University of Oslo, Oslo, Norway, available at: https://www.sciencedirect.com/science/article/pii/S0965997899000150 (accessed 20 November 2020).

Kan, H., S. (2001), Metrics and Models in Software Quality Engineering, pp. 176, available at: https://flylib.com/books/en/1.428.1/halsteads_software_science.html#fastmenu_1 (accessed 23 November 2020).

Kaner, C. (2004), Software Engineering Metrics: What Do They Measure and How Do We Know?, 10TH INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, METRICS 2004, available at: https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5B32D690C921ECD8C2517E832F404E09?doi=10.1.1.1.2542&rep=rep1&type=pdf (accessed 21 November 2020).

Kaur, K., Minhas, K., Mehan, N. and Kakker, N. (2009), *Static and Dynamic Complexity Analysis of Software Metrics*, World Academy of Science, Engineering and Technology International Journal of Computer and Systems Engineering Vol:3, No:8, pp. 1936-1938, available at: https://publications.waset.org/2684/pdf (accessed 23 November 2020).

Kienitz, P. (2019), *How me measure Software Quality*, dcdl software, available at: https://www.dcslsoftware.com/how-we-measure-software-quality/#:~:text=That%20is%20why%20developers%20measure,analysers%2C%20and%20refractor%20legacy%20code. (accessed 21 November 2020).

Kou, H., Xu, X. (2002), Most Active File Measurement Validation in Hackystat, Collaborative Software Development Laboratory Department of Information and Computer Sciences University of Hawai'i, available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.8.3180&rep=rep1&type=pdf (accessed 20 November 2020).

Parasad, K. V., Giuli, T. J., and Watson, D. (2008), The Case for Modeling Security, Privacy, Usability and Reliability (SPUR) in Automotive Software, Model-Driven Development of Reliable Automotive Services, LNCS 4922, Springer, 2008, pp. 1-14, available at: https://www.researchgate.net/publication/220487421_The_Case_for_Modeling_Security_Privacy_Usability_and_Reliability_SPUR_in_Automotive_Software (accessed at 23 November 2020).

Pomeroy-Huff, M., Cannon, R., Chick, A. T., Mullaney, J. and Nicholas, W. (2009), *The Personal Software ProcessSM (PSPSM) Body of Knowledge, Version 2.0*, Software Engineering Process Management Program, available at: https://resources.sei.cmu.edu/asset_files/SpecialReport/2009_003_001_15029.pdf (accessed 20 November 2020).

The Common Criteria for Information Technology Security Evaluation, ISO/IEC 15408, 1996–2015, available at: www.commoncriteriaportal.org. (accessed 23 November 2020).