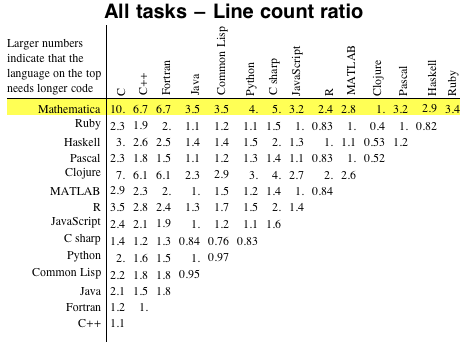
*Consider the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethics concerns surrounding this kind of analytics.*

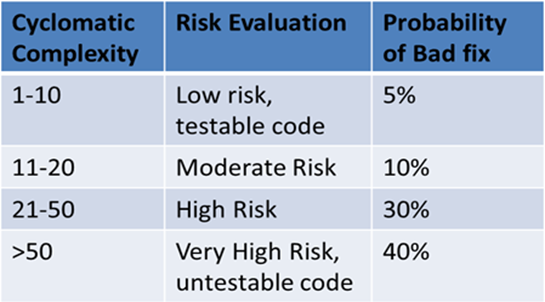
**Introduction:**   
 Measuring developer productivity has long been debated. A simple search on the matter will yield many results from developers who adamantly oppose any form of quantitative metric for assessing developer performance. Some developers are of the opinion that trying to apply any sort of metric to measure performance will ultimately only prove detrimental to the task at hand. While in some way, this view can be respected, it is unreasonable. There has to be some kind of measure for the process, the work that is put in, and the quality of performance.  
 In addition to being a relatively new discipline, software engineering is a profoundly complex area of study. Some people regard software as the most complex human-built artefacts that exist, and this is what makes well-established methods for measuring and assessing the engineering process so difficult to find and comprehend.

**Software Metrics:** IEEE defines a software metric as “a standard of measure of a degree to which a software system or process possesses some property”. Software metrics help project managers to gain an insight into the efficiency of the software process, project, and product. This is possible by collecting quality and productivity data and then analysing and comparing this data with past averages in order to know whether quality improvements have occurred. Also, when metrics are applied in a consistent manner, it helps in project planning and project management activity. Metrics can be described simply as quantitative measures that allow software engineers to identify the efficiency and improve the quality of software process, project, and product. However, in order to measure and assess the software engineering process one must be able to quantitively measure some kind of data within that process.  
 Norman E. Fenton and Martin Neil, in their paper ‘Software metrics: successes, failures and new directions’ describe software metrics as “the rather misleading collective term used to describe the wide range of activities concerned with measurement in software engineering”. Their paper tells us how basic software metrics have been around since the 1960’s. ‘Lines of code per a certain time period’ in this day and age seems like a primitive way of assessing any kind of software development, but before the introduction of high-level programming languages this was the norm (and is still in practice in some areas today). Lines of code (LOC) and thousands of lines of code (KLOC) were routinely used to measure engineer productivity and program quality (defects per KLOC). The obvious drawbacks of crude methods such as these were not recognised until the mid-1970’s accompanied by a surge in the number of programming languages available to developers.  
 The graph below (figure 1.) shows a comparison in lines of code between fourteen different programming languages. It was compiled by Jon McLoone, an author of Wolfram Blog. The illustration shows typically how many times longer a task written in the language at the top is compared to the language on the left. For example, on average, it would take ten times more lines of code to write a task in C than it would to write it using the Mathematica programming language.



(Figure 1.)

Despite the obvious flaws in the LOC and KLOC methods of measuring productivity and quality, up until the early 2000’s they were still widely used by companies and organisations as they are the most simplistic and easy to comprehend from a managerial perspective. The data is also very easy to collect. Fenton and Neil were not entirely against the use of simplistic approaches to measurement, as long as a less ‘isolationist’ approach was adopted. Fenton and Olsson, in 1999, used case study data from two releases of a major commercial system in order to test a number of common hypotheses about the basic metrics that were popularly used in practice. The system was split up into hundreds of different modules and the metrics data was analysed at the module level. The metrics used were: lines of code, defects found at four different testing phases (including in operation) and a number of complexity metrics including cyclomatic complexity. The outcome of the testing of these hypotheses was devastating for a lot of software metrics work that had been done. One outcome that stood out was the testing of the hypothesis that ‘modules with higher incidence of faults in early pre-release [are] likely to have higher incidence of faults in system testing’. Not only was this hypothesis strongly rejected, but there was evidence found to support a converse hypothesis.

**Complex Metrics + Algorithmic Approaches:**  
 Moving on from LOC methods of measuring and assessing the software process, what other types of data could we collect in order to measure performance and productivity? Some people would suggest measuring the quantity and frequency of commits pushed by an engineer throughout the development process. However, I would have to disagree with the idea that this is a useful metric to use in isolation. Commits are arbitrary changes captured in a single moment in time and can only be seen as a measure of activity and not one of productivity. Job van der Voort, vice president of product with GitLab, says that one should never measure their productivity with commits as commits can be meaningless when there is no additional communication involved. Commits show no reference to the quality of code submitted or the relevance to the task at hand and therefore as a standalone quantitative measure they do not hold their own.   
 Another commonly used metric is the measure of technical debt. Technical debt can be defined as an ‘incomplete, immature, or inadequate artefact in the software development lifecycle’ (Cunningham, 1992) or more simply it is aspects of the development that have been done incorrectly but have not yet been corrected and pose a risk of causing problems in the future. Technical debt can arise from inadequate test cases and low code coverage when performing unit testing as well as from prioritising some aspects of a code base and leaving the fine tuning of other features to a later date. In order to quantitively measure technical debt one must first be equipped to measure all of the contributing factors: code duplication, code complexity, test coverage, dependency cycles and coupling, lack of documentation, and programming rules violations.  
 Patroklos Papapetrou argues that all of these metrics are easily measured and can be expressed without having to resort to technical jargon that would be difficult for a project manager or employer to get their head around. For example, duplicate code can be algorithmically measured and is expressed as the ratio of duplicated code to the total lines of code within a range. A popular way to measure this is to examine the same series of identical tokens with some occasional variation. The duplication detection mechanism splits the lines of code into tokens, sets the minimum number of identical tokens that can be considered as duplicate code and the searches the code for the same series of tokens.  
 Code complexity can be measured using the metric ‘cyclomatic complexity’. Cyclomatic complexity of a code structure (method, function, class, etc.) can be measured by adding one unit per branch in the code (If, Case, Switch statements) and adding one unit per exit path or exception in the code. A complexity below or around ten is considered normal in most programming languages while a complexity around or above twenty-five is considered high risk. A program with cyclomatic complexity of over fifty is said to be unstable or unmaintainable. These are limits stated by McCabe and Watson in their 1994 and 1996 publications on software complexity. A program with a very high cyclomatic complexity will incur some technical debt as a result. 

Carolyn Seaman, professor of Information Systems at the University of Maryland Baltimore, teaches that it is impractical to completely automate the monitoring of technical debt as some kinds of technical debt can only be detected and interpreted by humans, no matter how effective and time-efficient the algorithmic approaches may be.  
 On their own, none of these metrics are adequate for quality and productivity prediction, but using them as a group of metrics you are presented with a lot of data that can be examined and interpreted in order to make an informed decision regarding the quality and performance of the software development process.

**PSP and Available Computational Platforms:**  
 It can be very difficult to collect useful measures because a lot of developers do not consider it an important activity, compared to the code that they are writing. Moreover, data that has been collected manually is often affected by errors, making it unusable for the most part. Engineering measures can be broken into two categories: product metrics and process metrics. The former describes product’s qualities such as dimensions, physical data, etc; while the latter describes the process’ qualities such as effort required and production time. In software, the former includes code length, complexity, reusability, maintainability, and the latter includes editing time, number and type of changes in a file. Tools for retrieving such data in software do exist but require some human effort and may collect only one of the sets previously described. Moreover, these tools allow for data acquisition but they do not provide any advanced or integrated analysis. In addition to this most manual data collection techniques are extremely unreliable. Too many human errors or missing data badly affects the analysis process. These errors appear mostly in critical periods, when the data correctness should help to understand better the situation, such as during high stress working periods. Another issue is that collecting metrics is a time expensive task. This is a problem because software projects are notorious for being late and there is no time to spend in activities that do not produce immediate benefits. A completely automated data acquisition tool should be able to both improve data quality and reduce to zero the acquisition effort done by programmers. Software development is an activity mainly based on human effort. For this reason, costs highly depend on the time spent doing each activity.   
 The Personal Software Process (PSP) provides a rigorous methodology to monitor software development from the early stages of a project to the finished product. This helps software engineers to keep track of their work and allows it to be easily analysed at any stage of the development. This perpetual monitoring helps the developer to trace their own performances, compare them to the planned schedule, and find out whether specific elements (i.e. environment, behaviours, interactions, etc.) affect them and the quality of their work. As stated in lean production environments; identification of inefficiencies is the first step to improve productivity. PSP requires the collection of detailed metrics of the development time, defects discovered and corrected at all development stages, as well as software size. Then all collected data is analysed statistically. These results provide software engineers with sets of data mainly used to make reliable estimates on variables such as time schedule and quality of an ongoing project, and to highlight problems in order to find out how to improve the development process. The process is transparent for users that do not require interrupting their main activity to collect data, solving the issue of unnecessary time being spent on the collection of that that would inevitably be riddled with human error and have to be disregarded.   
 In their 2003 paper, *Collecting, Integrating and Analysing Software Metrics and Personal Software Process Data,* Sillitti, Janes, Succi and Vernazza present PROM (PRO metrics), an automated tool for collecting and analysing software metrics and PSP data. PROM collects both code and process measures. The tool focuses on comprehensive data acquisition and analysis to provide elements to improve products. The data that is collected encapsulates a wide range of variables including all PSP metrics, procedural and object-oriented metrics, as well as ad-hoc developed metrics to trace activities rather than coding, such as writing documents with a word processor. The tool collects and analyses data at different levels of granularity: personal, workgroup and enterprise. This differentiation takes a picture of the whole software company and preserves developers’ privacy by providing to managers only aggregated data.  
 There are several commercial tools for metrics collection and analysis such as MetricCenter, an add-on application produced by Distributive Software that collects metric data without the need for any integration effort, and ProjectConsole, an add-on for Rational Suite produced by IBM Rational that collects both standard and custom metrics and presents the results graphically so that one can easily assess project progress and quality. However these data collection add-ons are domain dependant, use proprietary solutions, and often still require some human effort for data collection.

**Ethical Concerns:**  
 In their 2002 paper, Ethical Issues in Empirical Studies of Software Engineering, Janice Singer and Norman Vinson discuss the ethics and legal issues surrounding the field of metrics and software engineering. They discuss, using case studies, the correct and incorrect procedures to use when collecting data from software engineers, students, employees, etc. Consent is obviously a large factor in this argument. Ethicists do not fully agree on the necessary components of informed consent, but it is clear that it must contain at least some of the following elements: disclosure, comprehension and competence, voluntariness, the actual consent or decision, and the right to withdraw from the experiment. They debate this using the case of a software engineering professor, Dr. Gauthier, who used her students as test subjects to show how different views of source code influence program understanding. She did not request permission or ask consent from her students beforehand and as their professor and a figure of authority it was seen that she had coerced them into partaking in this research experiment. As their professor, the students may have taken the view that refusing to partake may affect their grade or their relationship with the school body. The research may have been completely innocent with no possibility of negative repercussions for any student but as Singer and Vinson state “The ethical difficulty arises not from the professor’s intent but from her power”.   
 Similar ethics apply when collecting data from software engineers to measure their performance and to provide productivity feedback to employers or managers. Consent is required when identifiable information is being collected. The information that is to be supplied to ‘subjects’ includes but is not limited to: the purpose of the data collection, the data collection procedure, any potential ‘risks’ to the subjects, the anticipated benefits for the subjects and the world at large, and a statement offering to answer the subjects’ questions on the process. Only when an individual cannot be identified from the data collected is it possible to dispense with obtaining the consent of individuals.  
 In metrics research, source code and class design documents are examined and flaws are then reported. From an ethical point of view this can be seen as harmful as flaws described in such documents can be damaging to a company. Singer and Vinson use a case study regarding a metrics researcher as an example. The researcher examines the relationships between metrics and defect rates in order to determine which metrics relate to software quality. In conducting an examination of the source code of some company’s systems he discovered that programmer identity accounted for a very large amount of variance in the defect rate of the code. Some developers work had far more defects than others. The researcher mentioned the relevance of developer identity in his report, but did not mention the particular developers or the defect data for any individual that would have contributed to such a variance. This kind of information, while although in this case not specifically naming anyone in the report, can raise issues of confidentiality, consent and beneficence in regard to those individuals within the company.  
 Another study used by Singer and Vinson was that of a Dr. Jonas who is a professor at a prestigious university in the newly created software engineering program. He recently embarked on a research project to determine how collaboration styles influence software quality. His hypothesis is that software engineers who work well together produce better software. Dr. Jonas collects data by observing software development teams at local companies. Next he categorizes the teams according to their success at collaboration and also collects metrics for software components previously developed by the same teams. Dr. Jonas plans to correlate the collaboration quality measures with the metrics to determine whether teams that collaborate better produce higher quality code. A couple of weeks into the research program, a manager asks to see Dr. Jonas’ field notes and wishes to know how his company compares to the other companies regarding the metrics assessment. In this situation Jonas is faced with the choice of angering a host organisation or breaching the confidentiality of his subjects. Both choices put him in a tough situation as choosing either option would probably lead him to lose access to his subject population. In order to have avoided this predicament Jonas should have considered the ethical implications of his research at an earlier planning stage of the process and ensured that all of his subjects and the hosting organisations involved understood their rights and responsibilities before he gave them to option to decide to participate in the study.  
 A final aspect of informed consent relates to how the data is used. Article 1.7 of the ACM (Association for Computing Machinery) states that all information gathered for a particular purpose should not be utilised for any other purposes without additional informed consent. For example, data collected to describe a software process should not be used to evaluate group members without the subjects explicitly consenting to this additional use of the data. In general, it is better to specify in the informed consent document all anticipated future uses of the data. If a researcher wishes to use the data for an additional purpose that was not pre-specified in the informed consent document, they must generally consult a review board to request how they should attempt to gain consent for the intended use of the data. Unfortunately for ESSE (Expert System for Software Evaluation) researchers, two of the largest subject bodies, students and company employees, are considered vulnerable to being coerced into testing a metric analysis as they may feel pressured to do so by their professor and employer.

**In Conclusion:**  
 Upon viewing and discussing these issues in relation to the measurement and assessment of the software process we can see that it is a complex and controversial field that is still developing as a new, emerging area of study. There are many different ways in which people within the field of software engineering feel their work can or should be measured, be it with physical quantitative measures such as lines of code or number of defects, or with more abstract methods such as the measurement of technical debt or cyclomatic complexity. Data can be collected and interpreted by metric analysis software and software add-ons such as PROM, MetricCenter and ProjectConsole, as discussed above. Software measurement and metrics help us a lot, in evaluating software processes as well as the finished software product. The set of measures, some identified in this paper, provide organisations and management teams with better insight into the validation activity, improving the software process towards the goal of building a successful software while having an efficient management process and being aware of ethical concerns.

**Bibliography:**

<https://www.accessengineeringlibrary.com/browse/applied-software-measurement/p2001b4bb9970071001>

<https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0>

<https://stackify.com/application-performance-metrics/>

<https://stackify.com/track-software-metrics/>

<https://dl.acm.org/citation.cfm?id=559784>

<http://ecomputernotes.com/software-engineering/software-metrics>

<http://www.academia.edu/9574064/Software_metrics_successes_failures_and_new_directions>

<http://blog.wolfram.com/2012/11/14/code-length-measured-in-14-languages/>

<https://about.gitlab.com/2016/03/08/commits-do-not-equal-productivity/>

<http://ccsl.org.br/files/TD%20talk%20USP.pdf>

<http://thinkapps.com/blog/development/technical-debt-calculation/>

McCabe, T. and A. Watson (1994), *Software Complexity* (CrossTalk: The Journal of Defense Software Engineering).

Watson, A. H., & McCabe, T. J. (1996). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric* (NIST Special Publication 500-235)

<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1158289>  
A. Sillitti, A. Janes, G. Succi, and T. Vernazza, “Collecting, Integrating and Analysing Software Metrics and Personal Software Process Data,” in Proceedings of the 29th Euromicro Conference. IEEE, 2003.