Ross McCrann 16326617

CS3012 Software Engineering

Measuring & Assessing Software Engineering Processes

Requirements Definitions

This report aims to analyse and consider the methods and approaches in which software engineering can be processed. In particular, this report will evaluate these mechanisms in terms of 1) measurable data, 2) computational platforms available to perform the work, 3) algorithmic approaches available and 4) ethical concerns surrounding the analytics.

In order to understand how and why we measure the software engineering process it is first necessary to define the term ‘software engineering’ as well as ‘measurement’ in order to clearly and effectively discern their true meaning within the context of this report.

Software Engineering is the systematic application of engineering approaches to the development of software. It methodically applies scientific and technological knowledge, methods and experience to the design, implementation, testing and documentation of software. Furthermore, software engineering has more frequently involved the process of analyzing existing codebases and libraries and updating and modifying it in order to meet the new requirements of the software. In this respect, there is a much greater impetus within the industry for better practices and software engineering processes to be applied to the development of software. The application of these practices and processes allows for codebases to be more efficiently updated and maintained which has become increasingly important as projects have become much larger and complex involving many different engineers at different times.

For the purpose of this report, a ‘metric’ or ‘measurement’ in terms of software engineering will be defined as an accurate and quantifiable demonstration of the size, quantity, amount or dimension of a software product or process. A software product is any item that has been produced or will be produced in the software development life cycle. Attributed to the product are the quantifiable characteristics of size, effort and cost as well as efficiency, testability and portability of code. Software processes are collections or instances of software related activities. The attributes that can be applied to this metric are of duration of activity, effort associated with the activity as well as cost, quality and stability.

1. Measurable Data

Software engineering is an inherently difficult field to apply quantifiable metrics to, however, managers and stakeholders need to be able to quantify productivity using data metrics in order to operate a software development cycle.

Lines Of Code

A logical however, admittedly arbitrary metric for quantifying software engineering productivity is Lines Of Code (LOC) aka Source Lines Of Code (SLOC). LOC or SLOC is a method of measuring the size of a software program by counting the number of lines in the text of the program's source code. This is then used to predict the software engineers level of productivity and estimate the amount of effort and resources that will need to be dedicated in order to complete the program. There are two main types of LOC: Physical LOC and Logical Loc. Physical LOC counts the lines of text in a program’s source code including blank lines. Logical LOC tries to measure the number of statements in a program’s source code. However, statement definitions are tied to specific programming languages (for example a Logical LOC for Java is the number of statement terminating semicolons ;).

Although large amounts of effort are exceedingly correlated with a high LOC, functionality of code is far less correlated with LOC. It is generally known within the industry that skilled software engineers are normally more able and well equipped to develop the same or improved functionality with far less code, so one program with less LOC may exhibit more functionality than another similar program. However, this is often due to a less skilled developer creating the same functionality with much more redundant code. While it may be very easy to implement LOC is a particularly poor measure of productivity amongst individuals, because a software engineer who only programs a few lines of code may still be more productive than an engineer who creates many more lines of code. Additionally, optimization of code to get rid of redundant code will mostly reduce the lines of code count.

Similar easily measured software metrics include

- Error Count
- Commit
- Keystrokes
- Active Days

These metrics can offer an important insight into the development process, they are individually very poor indicator of software quality or productivity and should only be considered in conjunction with other more accurate software engineering metrics.

Agile Metrics

Cycle Time

Cycle Time is the amount of time it takes from when a unit of work is actually started i.e in-progress State (e.g. A software engineer begins programming a task) until it is fulfilled i.e a completed state.

Lead Time

Lead time is the amount of time it takes for a unit of work to go from a request i.e. a queue state (eg. team lead has requests a task to be completed) to a fulfillment i.e. a completed state. The lead time encompasses the cycle time and is invariably longer in time. The Lead time describes two aspects of a system: the time it takes for work to be processed in that system (cycle time), and the time it takes for work to begin to be processed in that system, which is a function of the frequency that orders come into that system.

Velocity

Velocity Is the amount of functional software delivered in a single iteration. Velocity tracking over different ranges of time can be extremely useful in estimating project and task completion time in the short, medium and long term

Production Metrics

Code Churn

Code Churn is a metric for measuring the degree to which a codebase or system changes over a period of time. It can be measured in terms of code churn size, type, breadth, and depth. It can occur when a software engineer writes and rewrites their own code within a set period of time. For example, an engineer may write 100 lines of code in a day however only 30 lines of the code were able to pass tests and be shipped. So while a software engineer may have written a lot of code, the net change to the code base was minor and represents a low yield of productive code. Code churn based analysis and testing can be very effective in uncovering issues (especially at later stages) in the software development life cycle and be instrumental in protecting software quality. Code churn indicates the percentage of code that sticks around to deliver business value and by noticing sudden peaks in code churn, software development teams can resolve problems early before they have a chance to develop into greater and more complex issues later in the development cycle. High churn rates are more common at the beginning of the development cycle, while low churn rates later in the cycle generally indicate an engineer or team reaching their deliverables .

Commit Frequency & Active Days

Commit frequency and active days are two metrics that measure software engineering processes which essentially serve the same purposes. An active day is said to be a day in which a software engineer contributes code to the project, which includes specific tasks such as writing and reviewing Code. The need to invest time and effort into non-coding tasks such as planning, meetings, and evaluating software specifications are inevitable. It is common and accepted within the industry that individual software developers and team will not be able to commit code everyday and will often often lose time and days to activities that are necessary and required to supplement the creation of code. Monitoring the commit frequency enables you to analyse which activities have an impact on a development team's ability to commit code. These metrics are important for managers and developers to monitor as committing productive, functional code is the primary method creates value in the software development cycle. These metrics allow team managers to ensure that software production overheads do not become a burden to the creation of code. A common mantra that encapsulates this methodology is "Commit Often, Perfect Later, Publish Once".

Operational Metrics

Operational Metrics indicate how software is running in production and how effective the operations engineers are at maintaining it. This includes Software reliability testing, is a field which tests a software's ability to function, given different environmental conditions, for a particular amount of time. Software reliability testing can help to discover many problems in a software's design and functionality.

Mean Time Between Failures (MTBF)

This prediction uses previous observations and data gathered from various stages of development, including the design and operating stages to determine the average time between failures. MTBF predictions are often used to designate overall failure rates, for software products. The formula for MTBF is:
Mean Time Between Failure(MTBF) = Mean Time To Failure(MTTF) + Mean Time To Repair(MTTR).
The probability of failure is calculated by the Number of failing cases / Total number of cases under consideration.

Mean Time to Failure (MTTF)

Similar to MTBF, the mean time to failure (MTTF) is used to predict a software's failure rate. The key difference is that MTTFs are used only for replaceable or non-repairable products that a software relies on, which can include various networking components such as: servers, firewalls, modems etc.

Mean Time to Recover (MTTR) `

While the MTBF and MTTF measure time in relation to failure, the mean time to repair (MTTR) measures how long it will take to get failed software up and running again after a break or malfunction. MTTR includes the time it takes to for a software team to find out about the failure, diagnose the problem and then repair it. It is a basic measure of how maintainable software is and, ultimately, is a reflection of how efficiently a software problem can be fixed.

These metrics can be very useful in measuring a software product's reliability however, these metrics are only as good as the data they rely on. By monitoring these metrics, software engineers can put in place measures for maintenance and repairs before a software failure.

Test Metrics

Test metrics measure how comprehensively a system is tested, which is generally correlated with software quality. Test metrics are an increasingly important part of the software development process and is often a key factor in the success of a software product.

Code Coverage

Code coverage is a metric that indicates the lines of code that are executed when one of automated tests are run on the codebase, expressed as a percentage of the entire codebase. For example, 90% code coverage would mean that the tests execute 90% of the code and that 10% of the overall code available in the code base is never executed when the tests are run. Whether or not code coverage is an important metric to consider or not is hotly debated as the level of coverage achieved does not necessarily translate to the value of the code that is being tested. The consensus for generally accepted code coverage within industry hovers around 80% of the code base.

Unit Testing

Unit tests are a level of software testing where individual units/ components of a software are tested. The purpose of unit testing is to validate that each individual unit of the code performs as it was intended to. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In object-oriented programming, the smallest unit is a method.

Customer Satisfaction Metrics

Customer Satisfaction Metrics indicate the overall perception of a software by its end users. Metrics such as the Net Promoter Score (NPS), Customer Effort Score (CES) and Customer Satisfaction Score (CSAT) can reveal to developers how well they are meeting the needs of clients and how they can improve the software to better meet requirements. Ultimately, the greatest measure of software's success can be determined by the end users response to their interaction with the software and how it performs once it is deployed.

2. Computational Platforms

Personal Software Process

The Personal Software Process (PSP) is a framework which provides a rigorous software methodology allowing the continuous monitoring of software development throughout the stages of it's development cycle. PSP allows software engineers to monitor an analyse their personal performances and compare them between their targets and projected schedule in order to evaluate any difficulties that may be affecting their progress. By identifying any inefficiencies in their workflow a developer can self evaluate and improve their own productivity. PSP is a low cost method which is both flexible and customisable to an individual software engineer's needs. However, PSP requires the continuous and meticulous manual input of detailed metrics and data throughout all stages of the development cycle so that they can be analysed statistically. This approach can invariably lead to data quality problems as the probability of human error increases dramatically. Additionally, the constant input of metrics and data can be both stifling and frustrating to an engineer software development workflow.

PRO Metrics

Pro Metrics (PROM) is a tool designed to aid software developers and managers to keep software projects on track and under control. PROM automates the data acquisition and analysis of code and development metrics. The system can provide support to its users by compiling reports tailored to their individual role within a project. A software developer can exclusively access their own PSP data and statistics as well as a detailed analysis of their work, highlighting inefficiencies so that they can improve their personal software development workflow. PROM addresses two of the major issues arising from the PSP; human error and time spent inputting data metrics. PROM's data collection procedures are fully automated and standardised across workflows allowing engineers to focus on developing productive software uninterrupted. While managers cannot access an individual software engineers data however they can access a report that aggregates important data metrics on a project in its entirety.

Hackystat

Similarly to PROM, Hackystat is a software development framework that allows individuals to collect and analyze PSP data automatically. Hackystat is open source and has tools for the collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data. The framework focuses on individual data collection and the individuals exclusive use of that data. The tool's structure allows a very high level of privacy, which is crucial in software and IT industries however limiting in potential upside for the for the project team as a whole and team managers. Hackystat users typically attach software 'sensors' to their development tools, which are used to unobtrusively collect and transmit "raw" data about development to their own proprietary web service repository where it is stored. This repository can then be queried by development teams to form higher level abstractions of the collected raw data and integrate it with other software metrics and processes to provide further analytics and information. This infrastructure facilitates software development by expediting that can be used for quality assurance, project planning, and resource management.

3. Algorithmic Approaches

Machine Learning Algorithms

The improvement of the currently used processes and quality assurance mechanisms is an important part of software engineering. By applying different machine learning techniques to software development data metrics

Machine learning has the inherent advantage of being unbiased, whereas managers, experts and consultants instinctively use their experiences, intuition and expertise to assess ways in which software engineering can be processed, which are vulnerable to the influence of bias. The utilisation of machine learning algorithms can also eliminate noise from data metric sets. Machine learning algorithms implicitly solve the problem of imperfect sets of data that may otherwise influence people analysing software production metrics. Machine learning algorithms can be subdivided into three categories: supervised, unsupervised and reinforced learning.

Supervised ML Algorithms

Supervised learning as the name suggests, indicates the presence of a supervisor as a teacher. Supervised learning is essentially learning in which the algorithm is trained using data that has already been tagged with the correct output. It is then provided with a new set of (data) so that the supervised learning algorithm can analyse the training data and produce a correct outcome from the verified output.

Unsupervised ML Algorithms

Unsupervised learning involves the training of an algorithm using information that is unclassified and unverified allowing the algorithm to act independently on the data set without any guidance. The algorithm groups unsorted information according to different patterns, similarities and differences it identifies without any prior training of data.

Reinforcement ML Algorithms

Reinforcement learning is about taking suitable action to maximize reward in a particular situation. It is employed by various software algorithms to find the best possible behavior or path it should take in a specific situation in order to maximise the achievable reward. Reinforcement learning differs from the supervised learning in that the training data does not contain the correct output value with it so that the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent makes a decision what it believes it should do to perform the allocated tasks. In the absence of a training set, a reinforcement algorithm must learn from each experience/iteration it has in order to correct its decision making and maximise reward in a situation.

K Means Clustering Algorithm

Artificial Neural Networks

Artificial Neural Networks (ANN) apply concepts adapted from biological neural networks in the brain , artificial intelligence and machine learning. ANN are based on a collection of connected nodes called neurons and have an exceptional capability to extract meaning from complicated and imprecise data. They can be used to derive patterns and detect trends that are too complex or indistinguishable to be noticed by either humans or even other software engineering techniques. ANN can use machine learning and pattern recognition methods to find accurate estimates for a team's software development 'effort' within software engineering processes.

The Algorithm for software development effort estimation using ANN is:
1. Data Collection: Collect data and characteristics from previously developed projects like methods and tests used.
2. Division of data set: Divide the data set into a training set and a validation set.
3. ANN Design: Construct a neural network with the same number of neurons in it's input layers as the number of characteristics in the given project.

4. Training: Grain the training set first to train the neural network.
5. Validation: When training is complete the ANN is validated with the data from the validation set.
6. Testing: Lastly test the created ANN by feeding test dataset.
7. Error calculation: The performance of the ANN is analysed. If it is deemed satisfactory then the system is stopped, otherwise it loops back to ANN Design step (3), makes alterations to the network parameters used and then proceeds.

Artificial Neural Networks are a great approach in estimating overall development effort. They have a greater and more precise estimation capability than other models and hence can be used to calculate software effort estimation for a variety of project types.

4. Ethical Concerns Surrounding Analytics

The utilisation of empirical methods and quantitative metrics by business to measure software engineering processes and products is on the rise and increasing rapidly. Tests, metrics, case studies and frameworks are examples of empirical methods being leveraged by businesses to investigate both software engineering processes and products. The increased application of these methods has also brought about an increase in discussions about how these methods can be altered to account for different idiosyncrasies present in the field of software engineering. In stark contrast, the ethical issues raised by a reliance on empirical methods have received little, if any, attention within an industry increasingly driven by productivity, code commits and profit margins. It is crucial that companies consider the ethical and moralistic consequences of the software engineering processes and frameworks that they implement.

Privacy

Should employers have access to and be able to utilise the information that informs the metrics that measure software engineering processes and products if said data is obtained from the employee. The law acts as the defining ethical and moral framework that companies that businesses and software development processes generally adhere to. In Europe, the EU General Data Protection Regulation (GDPR) came into effect on 25 May 2018. This regulation significantly increases employers' obligations and responsibilities in relation to how they collect, use and protect personal data including that of employees. GDPR does recognises that the processing of specific data is necessary for organisations to perform their functions for example processing employee payment. For cases such as these, the GDPR specifies the lawful grounds on which organisations can process personal or sensitive data in Europe. So employers can access the data of an employee such as information regarding their workflow through the aforementioned computational platforms and apply them to the above measurable data metrics so long as that any data that they use or hold is void of any information that could be used to distinguish or indicate an individual's identity, such as their name, sex, employee number etc. In this respect, an employee who is not performing well as indicated by an arbitrary metric does not take into account any of the external factors affecting employee performance and thus any measures taken against an employee should not be predicated on any individual metrics. However, Different countries have different laws and the enforcing of regulations can be difficult given that data can be transferred from country to country where different sets of laws, regulations and standards apply. To combat this and try to provide an ethical framework to software engineering the IEEE Computer Society have created a universal framework for ethical software engineering practices that contain eight principles related to the behavior of and decisions made by software engineers. While such a framework cannot be universally enforced it should be promoted for the betterment of mankind as a whole given the profound presence software has in the world today globally.

Accountability & Testing

Due to their control in developing software systems and the inherent scalability of software, software engineers have an unprecedented capacity to affect the lives of others and potentially cause harm to different entities for the better or worse. It is paramount to ensure, to the utmost amount that is possible, that software developers are committed to software processes and products being created and used for the benefit of others. In accordance with that commitment, professional software engineers should adhere to a universally agreed and mandated Code of Ethics and Professional Practice such as those laid out by the IEEE Computer Society. In reality this can be hard to achieve as different businesses have different ambitions and objectives with the benefit of some stakeholders often conflicting with the desires of others. However, an area software engineering processes that should be agreed upon is testing. Software Engineering methods are applied to some of the most important and critical systems in the world. The proper amount of testing should take place to ensure that lives are not put at risk through a lack of proper care. For that matter, professional software engineers should be culpable for a certain amount of professional accountability and malpractice similar to other professions. But to what end should software be tested? Is an increasingly pertinent question. As technology increases in its level of capability and sophistication so does too out reliance on it in our lives. And with further complexity of code comes a greater amount of edge case testing with further degree of unpredictability. For example a software flaw in the Boeing 737 MAX system created two fatal crashes leaving engineers scrambling for a fix. In cases such as these where an unforeseen edge test case or bug occurs should individuals be held responsible for malpractice? Or is it a problem more systemic in software engineering as a whole as the complexity of the software engineering processes increase year by year.

References

1. https://www.geeksforgeeks.org/software-measurement-and-metrics/
2. https://www.tutorialspoint.com/software_quality_management/software_quality_management_measurement.htm
3. http://www.projectcodemeter.com/cost_estimation/help/GL_sloc.htm
4. https://www.sealights.io/software-development-metrics/top-5-software-metrics-to-manage-development-projects-effectively/
5. https://www.cui.com/blog/mtbf-reliability-and-life-expectancy
6. https://www.bmc.com/blogs/mtbf-vs-mtff-vs-mttr-whats-difference/
7. https://searchstorage.techtarget.com/definition/mean-time-to-repair-MTTR
8. https://dzone.com/articles/code-churn-a-magical-metric-for-software-quality
9. https://www.caroli.org/en/lead-time-vs-cycle-time-from-manufacturing-to-sw-development/
10. https://kanbanize.com/kanban-resources/kanban-software/kanban-lead-cycle-time/
11. https://dzone.com/articles/code-churn-a-magical-metric-for-software-quality
12. https://blog.gitprime.com/why-code-churn-matters/
13. https://www.bmc.com/blogs/mtbf-vs-mtff-vs-mttr-whats-difference/
14. https://searchstorage.techtarget.com/definition/mean-time-to-repair-MTTR
15. https://ieeexplore.ieee.org/abstract/document/345828
16. http://sunnyday.mit.edu/16.355/metrics.pdf
17. Machine Learning based on Software Metrics for Process Assessment @ https://pdfs.semanticscholar.org/042e/dca6afa191013529c61cff6d2f6f5d8695fa.pdf
18. A Comparison of Techniques for Software Development Effort Estimating http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.572.3879&rep=rep1&type=pdf
19. https://en.wikipedia.org/wiki/Artificial_neural_network
20. https://dzone.com/articles/7-useless-test-metrics
21. https://en.wikipedia.org/wiki/Reinforcement_learning
22. http://softwaretestingfundamentals.com/unit-testing/
23. https://www.itgovernance.eu/blog/en/expert-gdpr-qa-the-material-scope-of-personal-data-and-legal-implications
24. https://www.citizensinformation.ie/en/employment/employment_rights_and_conditions/data_protection_at_work/data_protection_in_the_workplace.html
25. https://www.computer.org/education/code-of-ethics
26. https://www.bloomberg.com/news/articles/2019-07-27/latest-737-max-fault-that-alarmed-test-pilots-rooted-in-software