Name: Leong Kai Ler
Student Number: 15334636
Course: CS3012 Software Engineering
Lecturer: Professor Stephen Barrett
Date: 7th November 2018

Task: A Report on Measuring Software Engineering

Description: To deliver a report that considers 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. Reading list available here. 10 pages.

Abstract: In this report, we will first discuss introduce software engineering process measurement and why is it so important to be heavily practiced in various industries. Then, we will talked about the various methods to measure and assess it along with how we quantify its measuring data. Subsequently, we will scrutinise the computational and algorithmic solutions available to perform such task and explain why some have gone obsolete while the others thrived. Lastly, this report will also list out the ethics concern surrounding this kind of analytics.

Introduction:

In software engineering, developer contributes their time to produce useful software applications. So, it is plausible to state that the productivity of a software engineering process in any particular industry is defined by the ratio of its output to its input, with the input being the time spent by developers and the output being the software applications produced. The output of a software engineering process can be measured from two requisite perspectives: quantity and quality. By denotation, quantity is the amount of labor hours that a coder consumes to make any code changes, whether if it is insertion of new codes of deletion of old ones. On the other hand, quality is quantified by predicting and identifying the different kinds of problems with analysis tools using static codes.

There is a significant importance to measure the productivity of software engineering processes due to the urge to identify and analyse solutions to be propagated, in order to increase development productivity and reduce cost along with wastage of resources available simultaneously. Moreover, by continuously monitoring these measurements, incremental improvements can be implemented to the production processes and its environment.

Methods for measuring and assessing software engineering process:

The key challenge in performing such tasks is that there is no single metric that can fully capture or cover every aspect of the software production line. As mentioned, there are two factors that determine the output of production: quantity and quality. So, in the context of software engineering, we can inferred that a highly productive coder is one that can stage in **more** code changes that are also **efficient** during deployment. This can be achieved by collecting rich data sets from developers' version control repositories and measuring their coding productivity based on their code changes. We can break down the measurement of coding productivity into two aspects: (i) the code quantity, which is the amount of code changes produced and (ii) the code quality, which is the efficiency embodying the source code. In order to quantify these two aspects, we introduced metrics system as a management tool to facilitate better decision-makings during software development and testings.

Before the 1970s, some traditional metrics such as Lines of Code(LOC) and cyclomatic complexity were heavily endorsed as the most accurate metrics to measure software engineering development. Even though some of them are still being practiced today, these metrics are surrogate measures that are too fixated on the program size, rendering them futile in efforts to analyse the functionality and complexity of high level programming languages.

Code quantity aspect is more fixated on the time factor, mainly on how much code changes can a standard developer contribute in a given amount of time. From the perspective of Agile Process, the basic metrics for such measurement of the development can be divided into four main categories: Lead Time, Cycle Time, Team velocity and Open/Close Rates. Lead Time represents the time taken from planning a software to building it, while Cycle Time is the time required to delivering the process forward to production. The delay in wait time for these both metrics can be reduced by simplifying and improving decision making. Team Velocity measures how many software a team of developers can complete in one iteration, while Open/Close Rates count the amount of issues raised and solved during the process. We further scrutinise into the Open/Close Rates metrics by dividing it into MTBF (Mean Time between Failures) and MTTR(Mean Time to Repair).

As for code quality, (also known as code coupling) according to Semmle's report, it can be viewed as an ensemble of non-linear quantile regression models due to its nature of having many facets. For instance, a run-time bug would imply that the source code quality is bad, but on the other hand, if the source code of the program is error free, it can still contain high technical debt, thus making the maintenance process difficult and costly. Intuitively, it is reasonable to say that the more errors detected in a source code, the worse its code quality. However, there are other situations that should be considered too. One simple example is when a code with less LOCs but one serious error alert is to have worse quality than one with more LOCs but less serious errors. This means that the properties of the code that triggers the alerts and their types has to be taken into account.

In real world sophisticated software programming, code complexity is a factor that should be emphasised in code quality. The measure of code complexity defines the overall sophistication of a program. The simpler the given axiom of a software, the easier it is to comprehend and the better it is engineered and developed. The crux is that in a sophisticated program, if the code is difficult to understand, it will be even troublesome to deal with the potential defects and bugs lying within it, which further delays any solutions to be devised and deployed.

Another parameter that is constantly being overlooked during a software development phase and should be taken into account while measuring the software quality is its security. The field of security extends to how susceptible is the software to virus infections and how much stressed test can it withstand or passed. MTTR (Mean Time to Repair), which is defined by the time between a security breach being discovered and a solution to be deployed, is crucial in the production and should be kept tracked over time to learn more about security issues and study ways to cope with them in automation.

If we further scrutinise into this matter, in a research report published by UFF Media Lab, software engineering metrics, shouldn't solely be applied to the software itself or the capabilities of coders. In fact, research has shown that even developer behavioural performance and diversity of the software team can, to an extent affect the progress or achievements of a software engineering project. The performance of a standard developer is correlated by his actions and habits during his regular working routines. For instance, a standard developer who does not implement test methods during software engineering development process, will inevitably affects the quality of the output, in this case the software end product.

Embracing diversity can be defined as acknowledging, understanding, accepting and valuing people regardless of their differences. These differences range from age, class, race, ethnicity, gender, disabilities, etc. International corporations have been attempting to go full throttle on embracing diversity by promoting inclusive working environment as an effort to yield higher productivity and competitive advantages in their respective sectors. Although there are numerous diversities in workplace, the diversity factors that can substantially impact the key performance of a software engineering team are age, gender, experience and education. These four factors are generally referred to as Key Performance Indexes (KPI).

Computational platforms to measure and assess software engineering process:

The simplest way to measure software engineering process is to gather huge data sets and analyse them and the aim is to do it with the least development overhead possible. Hackystat is a high-level data collection and analysis software that is suitable for developer to handle this job. Hackystat comprises of a service-orientated architecture that has a sensor attached to the development tools used by coders to gather information and send them to the server for intensive querying. The design features of this device encompass four distinct main areas.

1. Client- and Server-side Collection: Provide side tools such as editors, build and test tools for clients and configuration management repositories and build servers for servers.
2. Unobtrusive data collection: Collects data from developers without interrupting their work flow and transmit them back to Hackystat data repository when there's network availability.
3. Fine-grained data collection: Hackystat utilises Buffer Transition to collect data every time developers make changes in active buffer from one file to another, effectively tracking of his/her work and development.
4. Personal and Group Development: Hackystat can define projects and shared artefacts in group work, highlighting each developers' contributions in every file.

The mentioned compelling features of Hackystat is the raison d'être of its wide applications in various technological innovations. Some examples of these innovations include software project defining and visualising telemetry toolkit, high-performance computing software support tool, software ICU(intensive care unit) that examines projects health and its relations and etc.

In real world, decision makings in software engineering are mostly based on gut and feeling. These practices are actually haphazard and should be avoided as much as possible, as they can normally lead to unwanted resources wastage and substantially increase the cost of building and maintaining large, complex software systems. In order to avoid such repercussions, software practitioners should make development decisions based on facts derived from pertinent and up-to-date information. This lead to a need to equip softwares with systems that can support its decision making process throughout its lifetime rather than just during its development phase, ie a Software Intelligence (SI). The purpose of creating a SI is to provide the decision makers with a better comprehension of the true potentials and actual limitation of the software, so they can plan, devise and enact accurate short and long term stratagems. To simplify, we want to reduce the dependency onto gut and feeling and focus more on logical, fact-based decision makings.

In the field of coding softwares, Mining Software Repositories(MSR) is a noteworthy example that resembles the functionality of a typical SI. MSR is capable of analysing and cross-linking data stored in software repositories to derive useful and actionable information about the designs of the software projects and their system architecture. In essence, the routines of MSR entail heavy mining source codes and bug related repositories by running an extensive mining algorithm on historical, run-time and code repositories and process them into warnings for the developers to propagate complex changes and fixes to the errors, bugs and potential shortcomings. Furthermore, enabling automation in MSR data analysis and processing facilitates its performance in pinpointing execution anomaly in Run-Time repositories by identifying dominant execution or usage patterns across deployments, and flagging deviations from these patterns. It can also assists in identifying and verifying dominant and correct framework or library API usage pattern by mining their usage in projects. The efficiency of MSR mainly stems from its ability to transform even static record-keeping repositories into active mined data for practitioners to enact changes based on potential risks issued. MSR significant advantage in predicting bugs and curating repositories to ensure the high quality data stored in it has propelled its popularity to significant heights as it has been widely integrated into large-scale projects such as Coverity and Project Insight.

One existing models that can be implemented to predict the coding time required to produce any code change is the deep MDN (Mixture Density Network). Since every commit is time-stamped in the version control repositories, it is also possible to predict the coding time profile of a coder as well. In this model, insertions and deletions from commits in multiple files are both regarded as code changes and are concatenated to construct a composite change string. The MDN model then captures the average coding time to produce the code changes and utilise the time interval to predict the standard coder's labor time. Moreover, MDN model is also not susceptible to irreducible noise caused by the generative process due to the presence of interruption in work nature and the skilled difference of coders. This allows MDN to be able to constraint the boundary of uncertainty in the coding time predictions. Thus, MDN is able to capture the behavioural pattern of a coder more precisely than other existing models, aside from accurately calculating the amount of output(code change) of a standard coder based on the given input(labor hours).

The relationship between code change and coding time is irreducibly noisy due to fundamental properties of the generative process (e.g., due to skill differences between developers and random noise due to work interruptions). In consequence, the regression model must be robust to noise. Plus, we want to bound the uncertainty of our coding time predictions. A MDN fulfils both criteria. Deep MDNs are multi-layered neural networks that predict the parameters of a mixture distribution (i.e., predict a random, rather than deterministic, target), where the parameters are a nonlinear function of input features. We train with a likelihood loss. At prediction time we may compute the mixture mean, or use the full distribution.

To further examine the behaviour and performance of a standard developer, we can use gamification, a novel approach proposed by Erick B. Passos and his team, that incorporates game mechanics into software development process, running it like a live game. This statement is justified with the resemblance of the process bear to a live game and its key concepts such as goals, rules, challenge and interaction that can be map onto the iterative process of a software development cycle. Therefore, it is possible to organised a software development as a set of hierarchal and partially ordered challenges that must be overcome by developers individually and as a team. The resemblance can even be further enhanced by incorporating immediate feedback and rewarding system into the process itself. So, in essence, developers who completed more challenges would have accumulated more ‘skill points’ and be entitled to better ‘rewards’, whereas the ideas they are regarded as immediate feedbacks to the projects.

A Hackystat-based Zorro System is commonly used to ensure developers adhere to their responsibilities of executing relevant tests in Agile manner during their work time. As this Zorro System is based on one of Hackystat’s prime features: unobtrusive data collection, it can implicitly gather information regarding the workflow and then transmit them to the Hackystat server for further analysis. As the process is automated in real time, this ensures constant flow of information to the data server, which enables it to provide an illumination and insight to the workflow of the developers without causing any interruptions, inconvenience and disturbance.

This monitoring concept is then further developed by researchers at MIT and applied by Humanyze at workplace in the form of a wearable, sociometric badge. These devices are equipped with sensors for recording movements, postures and tone of voice. Moreover, these badges can also detect the presence of other employees who are closing by through infrared rays, providing insights of social interaction of the wearer. These IoT(Internet of Things) based, data driven gadgets are not made solely to monitor workers’ performance. In contrast, they assist companies in delivering solutions to optimise the ambience of working environment. In essence, by acquiring and analysing these data, better management decision makings can be facilitated to maximise productivity, save cost and streamline the physical working space.

The promising potentials of this product have encouraged more renowned companies like Bank of America, BP, Target, Tesco, etc. to take the initiative of adopting it in working environment to get a full picture of an individual developer’s daily routine, from biometrics and stress levels to productivity and locations throughout the office. This allows companies to provide actionable insights and employ appropriate coaching to improve employees’ performance. Exemplary companies in this context are UPS, which utilises wearable tracking device to study and improve how workers stack parcels and boxes and Bank of America and Cubist Pharmaceuticals, which revamped their working environment and space after analysing the data collected to stimulate productivity of workers.

Algorithmic approaches to measure and assess software engineering process:

Some of the traditional algorithmic method to measure the software development are the cyclomatic algorithm and Halstead's metrics. These algorithms are designated to measure the code overall complexity of a software program.

Cyclomatic algorithm measures the complexity of a software by calculating the number of linear independent paths through source code of a program. The calculation is done by utilising a control flow graph in an external program, where indivisible or atomic statements in the subject program are represented by nodes in a graph. These nodes can be connected with a directed edge if one of the connecting nodes can be executed immediately after the connecting ones before it. Computation of cyclomatic complexity can be implemented by subtracting the number of edges with the number of edges present in the graph and then summing it the result up with twice the number of connected components. The equation is given as below:

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

where M is the cyclomatic complexity,
E is the number of edges,
N is the number of nodes,
C is the number of connected components

As for Halsteads' Metrics, it is commonly used by researchers in evaluating programs and query languages, measuring real-time switching system software, functional programs and open source software. It is built on the basis that computer programs are implementation of algorithms that can be considered as collection of tokens are either operators or operands. In other words, we can think of a program as a sequence of operators along with their associated operands. Halsteads' metrics calculates the code complexity by determining the operators and operands based on a counting strategy and then counting the distinct tokens among them. This calculation has a significant flaw, that is his metrics can only applied to calculations based on intuition since they do cannot explicitly define the generic measurable concepts of operators and operands in any program context. This is heavily not encouraged in practice as intuition is not sufficient to obtain accurate results that are repeatable and reproducible. Despite its incomplete and unclear counting strategy and faulty calculations, Halstead's metrics is still widely endorsed and used by research scientist by applying some adoptions into its calculations.

To assess a huge code data set in repositories, we can use Semmle's LGTM, an innovation that utilises QL to query complex, potentially recursive data structures encoded in a relational data model by treating code as data. QL is a declarative, object-orientated logic programming language that is well suited to analyse static code and query them because of its ability to identify potential problems lying between source codes. Using QL advantages, LGTM can continuously execute queries on source code snapshots, detecting simple mistakes such as initialised variables and complex errors like taint analysis that stems from data sources to data sinks. Every query generated is a quality probe that is associated with alerts that varies according to types of errors. Using domain knowledge developed by programming experts on code quality, the quality of code in each data set

can be assessed according to the error alerts triggered during the analysis by LGTM and given a 'quality score' depending on the error distributions in the project. The score ranges from 0.0 (worst) to 1.0(best). The 'quality score' for each category of errors is given by the following equation:

$$s_i = 1 - F_l, i(n_i)$$

where l is the project's LOC,

n_1 is the actual number of error alerts in i.

To calculate the final code quality score, the weighted score for each cumulative category is summed up, where the weights are defined by simple heuristics based on the statistical distribution of alert categories in LGTM data.

In the context of security and penetration, Agile practices should be prioritised over waterfall in augmenting security settings of software. In other words, it is highly recommended for developers to integrate security into each step of the delivering process. A controversial method that can be implemented is Automated security testing. The easiest and most direct way to improve the security features of a software is by finding its vulnerabilities of its system. A security tester can detect flaws in the system by writing automated tests and avoid testing vulnerabilities that have been discovered before. Tools like OWASP Zed Attack Proxy have been corroborated to be competent and practical for writing automated security tests. These test should be conducted in conjunction to functional testing, that is products should be continuously tested and fixed until there are no remaining vulnerabilities and loss of functionality, in order to progressed into the accepted state. Security tests and unit testings are vital as they allow developers to amend mistakes or address shortcomings to mitigate catastrophes during the deployment of their software.

Aside from using machine learning in and automation in security testing, machine learning algorithms are also efficacious tools in quality assurance in overall software engineering management. One notable example in this context is the Bayesian networks. Bayesian Networks is an abstract technique for modelling causal relationship based on Bayesian interference. The functionalities of a Bayesian Network is mainly focused on identifying crucial variables to be modelled, then forming topologies and construct node probability table (NPT) consisting of the hypothetical variables identified, eg. Code Complexity, Testing Effort and Number of Field Failures. By conducting these operations, Bayesian Network can systematically provide assessment and prediction models derived for further corrections, performance improvements or environment adaptations. Furthermore, its operation system is supported by the Activity-Based Quality Model (ABQM), that is designated to generate guidelines and checklist corresponded to the structure quality of its target. This enables Bayesian Networks to enhance its capabilities in assessment and prediction. Moreover, Bayesian Network has been verified to be superior in predicting software quality associated with diverse factors over neural network approaches and Least Square regressions, that do not depend on structured quality models. Due to its promising results in quality predictions, Bayesian Network has been implemented in the bug pattern measurements of the Tomcat servlet container by NASA.

As mentioned before, undeniably, there is a correlation between diversity of a software development team and its efficiency. A software engineering team with high efficiency implied their

good performance in work. Ergo, a software team with high diversity index would be more efficient than one with a lower diversity index. To calculate diversity index, we can subtract 1 from the Simpson Diversity Index derived by Edward H. Simpson:

$$SimpsonDiversityIndex = \frac{\sum n(n-1)}{N(N-1}$$

$$DiversityIndex = 1 - \frac{\sum n(n-1)}{N(N-1}$$

where
n is the total number of members of a particular species,
N is the the total number of members of all species.

In the diversity context, we can replace species by categories in diversity. For instance, we can divide gender diversity into men, women and other and age diversity into young, middle and old group. This equation is applicable to all Key Performance Indexes (KPIs). To prove that productivity and efficiency increases along with the diversity index of a software engineering teams, we can use the Data Envelopment Analysis (DEA) , which can compute relative efficiency measures from multiple inputs and outputs. Productivity of a software development can be difficult to calculate as there are many input and output factors that needs to be taken into account. Examples of inputs include labour hours, fund invested, etc, whereas outputs of a project can be LOCs, function and feature points, profits, use cases, objects, etc. So, in most cases, productivity measurement can be simplified into the following equation:

$$Productivity = \frac{ProjectEffort}{TeamSize}$$

Then we can utilise the advantage of the DEA model to refactor all the inputs ie all the KPIs and outputs into the same units of measurement to enable proper calculation. DEA make use of linear programming calculation to adjust the relative performance and outputs of all the organisational units. This allows us to then calculate the efficiency of a software development team in a simple expression, as all the inputs and outputs are refactor to the same units of measurement.

$$Efficiency = \frac{Output}{Input} * 100\,\%$$

Ethical concerns:

One of main ethical concerns that could be raised around software engineering measurements is transparency issues. The extensive and suggested use of platforms such as MSR and Hackystat does have its trade offs in terms of data access. To increase the efficiencies of these platforms in data collection and analysis, rich data sets in repositories specifically in the historical, run-time and code repositories of the project have to be extensively mined. The more data content and its complexities gathered, the better the results of the analysis is presumed to be in providing guidelines for software practitioners to carry out precautionary measures and immediate fixes.

When all of these processes are being conducted, information encompassing an engineer's working hours, working behavioural style and pattern, average performance, contributions to the project, overall quality of work, etc will become visible to every personals that access to it. This kind of fine-grained data collection could sometimes create discords in a development group due to the amount of transparency manifested from the measurements and analysis. This could also cause unwanted tension and nervousness among coworkers, as some developers don't feel comfortable being monitor every single second and handing over private datas. In addition to the mentioned syndromes, despite the total control gained from the constant automated monitoring, companies could lose the trust and loyalty of their employees as they might feel exploited. In general, the efficiency of using computational platforms to deliver results of data management comes with a extortionate cost. It all boils down to where we draw the line during the data collection process, as we are compromising privacies for platform performance and accuracy.

Moreover, if these datas could also be abused by software industries to undermine workers. Software developers' overall performance data is crucial in determining their continuance to be retained in the team. Employers could easily take advantage of the weak performance statistics to force negotiation for a decrease in wages or salary by threatening with termination letters. Moreover, data collection and analysis can reflect the poor quality of work of the engineers and ended being used as a bargaining chip again. As a result, some employees have are deprived of choices and forced to implore their employers by taking up their nasty offers. Although this may seem a bit far-fetched, as it is more reasonable and rational to remove incompetent coders from the team, employers could seize the chance to shift these coders to other easier jobs, thus saving management costs.

Additionally, by utilising an automated synergetic feedback loop between data generation and mining over time, it is possible to rank, a software practitioner long term performance based on his work quality and quantity. However, there are possibilities where these calculations could sometimes be inaccurate or more precisely unfair. This is because it is difficult to integrate every existing software engineering process measurement factors into computations and then quantify them with equations. Therefore, problems such as inequality in individual achievement assessment could arise due to different factors being prioritised in the software engineering team or during the development process. For instance, bias factors such as age, race, gender, religion, etc can sometimes determine the performance assessment system in some companies.

The same issue can happen when integrating gamification approach into software development process. As mentioned, the fundamental concept for this approach is to map the requirements and tasks in the software project to a challenge map. A challenge map consists of project requirements that have to be met output a production-grade software and each requirement

fulfil is represented as an achievement. The same problem appeared as the challenges assigned to every developer or team are not exactly tailored to them, since they do not have the same skill set and their respective level and style. This

Last but not least, the use of wearable tracing devices such as sociometric badges created by Humanyze to track workers' behavioural pattern and habits can be seen as violation of their data privacy. Despite the remarkable functionalities of this gadget such as

data protection privacy right
privacy issues
carelessness in management
security breach causing data leak

Reference List:

- https://www.businessinsider.com/tracking-employees-with-productivity-sensors-2013-3?IR=T
- https://semmle.com/assets/papers/measuring-software-development.pdf
- http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf
- Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157.
- http://www.citeulike.org/group/3370/article/12458067
- https://www.thoughtworks.com/insights/blog/application-security-agile-projects
- https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams
- http://www.agilemodeling.com/artifacts/securityThreatModel.htm
- Hassan, A.E. and T. Xie, Software intelligence: the future of mining software engineering data, in Proceedings of the FSE/SDP workshop on Future of software engineering research2010, ACM: Santa Fe, New Mexico, USA. p. 161-166.
- https://klevas.mif.vu.lt/%7Esigitas/Kokybe/Straipsniai/1-s2.0-S0950584910001175-main.pdf
- http://uk.businessinsider.com/summer-vacation-synchronized-schedule-2018-2?r=US&IR=T
- https://www.washingtonpost.com/news/business/wp/2016/09/07/this-employee-badge-knows-not-only-where-you-are-but-whether-you-are-talking-to-your-co-workers/?utm_term=.9e93746f78a2
- Passos, E. B., Medeiros, D. B., Neto, P. A. S., & Clua, E. W. G. (2011). Turning Real-World Software Development into a Game. *2011 Brazilian Symposium on Games & Digital Entertainment*, 260. Retrieved from http://elib.tcd.ie/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=86478234
- Altiner, S., & Ayhan, M. B. 1. batuhan. ayhan@marmara. edu. t. (2018). An approach for the determination and correlation of diversity and efficiency of software development teams. *South African Journal of Science*, *114*(3/4), 69–77. https://doi-org.elib.tcd.ie/10.17159/sajs.2018/20170331
- Nelson, R. (2018). Design through test technologies boost real-world intelligence. *EE: Evaluation Engineering*, *57*(4), 12–16. Retrieved from http://elib.tcd.ie/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=ofm&AN=128759530
- http://profs.etsmtl.ca/aabran/Accueil/AlQutaish-Abran%20IWSM2005.pdf
- https://www.ptc.com/en/product-lifecycle-report/iot-at-work-is-your-office-wired-for-success