Report on the Measurement of the Software Engineering Process

Introduction

The purpose of this report is to review and discuss the measurement of the software engineering process. I will begin with an analysis and discussion of the different means with which the software engineering process can be measured and assessed in terms of the measurable data. This will be followed by an investigation of the computational platforms available for such work. Finally, I will look at the measurement of software engineering from an ethical standpoint, especially in regards to the collection of personal data.
This report will show how one could use existing technology and methodology to monitor the productivity of software engineers, as well as determine the extent of the utility of this measurement. Different measurement approaches, both manual and automated, will be examined and compared. Ethical implications of this data measurement and interpretation will be considered, especially in the light of recent EU legislation which has put ethical data collection at the forefront of current discussions. There are many differing views when it comes to how best to measure software engineering, and whether it can be measured at all. These contrasting viewpoints will serve as a basis for this report.

Measurement and Assessment of the Software Engineering Process

Measurement

There is debate in the field of software engineering as to what measurements reflect the productivity of engineers. Historical measurements like lines of code (LOC) or person-months of work are still used by some, despite being generally accepted as inconclusive and archaic. In the mid-1970s there was a move away from these measurements towards what were hoped to be more illuminating ones. The need for this was precipitated by an increase in the diversity of programming languages, as for example LOC in assembly and high level languages were incomparable [1]. Manual data collection from software engineers became the norm.

Personal Software Processing (PSP) is a common type of manual data collection and analysis of the software engineering process [2]. From the measurement point of view, it involves the engineers recording data about themselves and their work processes. They would record their development plans, time estimations of work, bugs encountered and fixed and design checklists, among other things. PSP allows for a wide array of data to be collected for analysis. The manual nature of the data collection allows subjectivity to affect the data. It is the engineer’s choice what data they record, and whether or not it is a true reflection of the work done or just their interpretation is difficult to ascertain. The collection of data can also be tedious and error-prone, further negating its usefulness. In addition, recording data about their process interrupts an engineer’s rhythm, and can

prove detrimental to their productivity. It is a good example of how measurement can affect the outcome of the very thing being measured.

That is not to say that PSP is unhelpful, as manual data collection adds a degree of flexibility that may be lacking from automated systems. It is simpler to ask someone to record different data than it is to configure a system to change the data it collects.

A radically different approach to measuring software engineers involves collecting data not about their work, but by their well-being, or happiness. Productivity has been shown to be correlated to happiness [3]. This is a general approach to measuring the productivity of workers in knowledge based or service industries, but is applicable to software engineering. The collection of data in this instance is done via wearable technology. The sensors in the wearable device measure movement. As described by Yano et al. [3] physical activity is a good indicator of happiness, so it is the criterion by which happiness will be calculated. The wearable devices provide an alternative means of collecting data. While this type of automated measurement is used to collect different data to that used by PSP, it is also possible to automate the collection of more traditional data that relates to software engineering.

Collection of data similar to that used in PSP can be automated, as in the tool Hackystat [3]. Data collection software can be built into the tools used by software engineers to monitor them. This prevents engineers from having to interrupt their work to record data about their process. Automated collection of data is unobtrusive and doesn't directly affect engineer productivity, unlike manual data collection. It can also allow more data to be collected. However, there are ethical concerns about this mode of data collection. By removing engineers from the loop, the people being measured lose their say in what data about them is recorded. This raises the issue of whether or not all of the data being collected is invasive and even necessary to measure software engineers. Automated data collection will be discussed further under the heading of computational platforms for assessment, and ethical concerns and implications will also be discussed more generally later in this report.

Other types of data, not just that specific to PSP or well-being, can be measured both manually or with automation. For example, version control systems produce a lot of information about the development process that could be used for assessment. How often commits are made, by whom, and affecting what parts of the software under development are just some of the metrics that version control indirectly provides to assessors. Version control effectively records the entire code-writing process, and serves as a virtual cornucopia of data that could potentially be analysed.

Assessment

Clearly, it is possible to quantify the software engineering process. The question is not whether or not data can be collected, but whether meaningful insights can be derived from it. As already discussed, archaic means of measurement like LOC and person-months offered little insight into the software engineering process. They were very subjective means of assessing engineers and could only be useful in a limited set of scenarios. Broader schemes of measurements, collected either manually or via automation, offer better hope for useful data.

PSP has been shown to convey useful information from the data it collects [4]. With proper application, one can become more effective at estimating how long a piece of software will take to write, or reduce the amount of defects in the delivered software. PSP analysis presents clear insights

into the engineering process, and makes clear what factors contribute to the process. However, given the significant overhead associated with the collection of the data PSP isn't often incorporated into development for a sustained period of time.

Another approach for analysing the data collected given by [1] is Bayesian Belief Nets (BBN). BBNs are graphical networks with associated probability tables. The nodes represent the uncertain variables in the model and the arcs represent the causal and relevance relationships between the variables. The probability tables for each node give the probability of the variable of the node being in one of its possible states. For nodes without parents these are the marginal probabilities, while for nodes with parents these are the conditional probabilities for each combination of parent state values. BBNs allow the analysis to capture the uncertainty around the variables. The evidence gleaned from data collection is often subjective and laden with complex dependencies that could give it a different meaning based on context. With BBNs, this volatility and uncertainty can be captured and incorporated into the analysis of the data, thereby giving a more accurate interpretation of the data and a more useful analysis. This allows the software engineering process to be assessed in terms of not only the raw data collected, but the influences the different categories of data have on each other.

We see that it is possible to use the data collected to assess the software engineering process. Another key aim of this analysis is measuring individual software engineers. Like in any industry, project managers and employers desire means with which to compare their employees. They need to know which workers deserve greater remuneration, are candidates for promotions and which require extra training or potentially to be let go. The measurements used for the process as a whole can themselves be used to compare software engineers.

PSP allows project managers to review how well their engineers stick to deadlines, follow plans and complete their allotted work. They can also see which engineers are likely to introduce fatal defects into code, and which ones write safer code. From this information they can measure the performance of their engineers. This analysis gives managers a basis with which to compare individual engineers.

The data obtained from version control and issue tracking software presents a rich trove of information that can be used to compare engineers. Project managers can see which engineers make the most commits, and the amount of code they add when they do so, and infer their productivity from this information. For instance, when comparing two engineers a manager might notice that one engineer commits at a rate of one commit a day with each commit only adding a small amount of new functionality, while the second engineer may commit only once every two days but each of their commits adds a substantial amount of new functionality to the software. By using a combination of the metrics measured by version control project managers can get a strong insight into how well their engineers are accomplishing their tasks, as well as being better prepared to plan for future work.

This type of individual analysis by supervisors could be done manually, but rarely is due to the need for real-time information processing. This is a key area in which automated analysis and assessment is heavily favoured. We can see, therefore, that while measuring the software engineering process can be done manually, regardless of the overall aims of the assessment an automated approach is almost always preferable. Project managers and engineers can rest easy, as automation has proliferated in the software engineering metrics landscape.

Computational Platforms for the Analysis of the Software Engineering Process

Software engineers are one of the few professionals that make their own tools. Unsurprisingly, there are many computational platforms available for the analysis of the software engineering process. Some of these tools use manually collected data, but necessarily automate the analysis. Others automate the entire process, from collection to analysis to results presentation.

The Leap toolkit analyses the data collected by PSP [2]. Leap still requires the data to be collected manually, but it facilitates easier processing of the data. Leap irons out issues associated with erroneous or missing data, which is common when data is obtained manually. It also speeds up the analysis of the data, which makes PSP far more usable in many scenarios.

Hackystat is a substantial upgrade from the typical PSP or Leap toolkit. It is a platform for automating the collection and analysis of PSP data, well as extending the type of data it can collect and run analysis on [2]. Unlike generic PSP and Leap, Hackystat seeks to automate the entire process leaving engineers with an unobtrusive way of being measured and analysed. This can improve the degree to which analysis occurs, as engineers don't feel slowed down by the continuous data collection process. Software sensors that can interact with an engineer's existing toolchain, from text editor to version control, are a key part of the client side of Hackystat. These sensors quietly collect data while the engineer gets on with their work. This data is sent to a server which runs quantitative and qualitative analysis. Hackystat is also capable of combining data from multiple engineers. For instance, if two engineers are working on the same file together, Hackystat can interpret their data in the context of their collaboration and measure its effects on their performance and productivity.

GitPrime is a platform aimed at maximising productivity. It uses data from version control repositories to assess the productivity of software engineers, and provide visualisations to managers to better understand the data. GitPrime allows team leaders to see how engineers affect the codebase. They can review which engineers are making the largest contributions, which engineers spend a lot of time fixing their own mistakes instead of adding new features, how external factors like adding new team members or restructuring affects productivity and if projects are on track to meet deadlines, amongst other things. It is a holistic tool in regards to software repositories. Not only does GitPrime analyse a multitude of different data points, it presents its findings in easy to understand visuals that managers can quickly use to identify trends and that can be presented in a non-esoteric fashion. One of the biggest boons of computational analysis platforms is their ability to generate visualisations that simplify the process of sharing information, and GitPrime is a great example of that.

Jira, a software development tool produced by Atlassian, can also be used to track developer productivity. Centred around creating software for the user, it can be used to follow the progress made in developing and implementing features, and all of the subtasks that come with that process. Like GitPrime Jira presents data in a visual manner that allows the insights gleaned from it to be put to use immediately. Jira can track which team members do what, from fixing bugs to refactoring code, so that tasks in the future can be distributed according to the engineers' strengths. It is designed to be integrated with existing development tools, reducing its overhead and minimising its obtrusiveness.

A framework developed in the course of academic research for measuring the software engineering process is the Context-aware Software Engineering Environment Event-driven frameworK, or CoSEEEK [5]. Developed by academics in Germany, this framework automatically gathers information from its users and their environment and combines it with information from a knowledge base. This is then used to assess the engineering process with parameters inspired by common assessment approaches in software development, namely CMMI, SPICE and ISO 9001.

Static Object is a platform that allows project managers to measure their engineers according to very project specific criteria. The USP of Static Object is the Line Impact. Line Impact is a measure of the value of a commit to the codebase's repository. Static Object analyses the changes that a commit made to the codebase, be they additions, updates and/or removals. Their impact is assessed in order to quantify the meaningfulness of the commit. Line Impact assesses the performance of both individual developers and the development team(s) as a whole. Line Impact scores can be tailored by managers to better reflect the specific needs of the project. In this way Static Object provides a strongly customisable tool for assessing software engineers.

Managing time is an important part of every project, and Toggl is a software tool designed to do just that. Toggl allows engineers to track how they spend their time. This can give them insights into where their time goes, so that they can better prioritise tasks and understand what causes delays in their development lifecycle. Managers can similarly use this data to plan out the workday of their subordinates and ensure that deadlines are met and clients are satisfied. Since time is the ultimate measure of efficiency, engineers can be readily compared by how effectively they use their time.

WakaTime is a platform that aims to "quantify your programming". WakaTime works as plugins that can be integrated into IDEs and text editors to monitor metrics about programming. These metric are obtained at the level of projects, files, branches, commits, and features. WakaTime can track the time spent on specific programming languages, and the statistics can be combined with commits so that how long each commit takes can be measured. This tool also provides a leaderboard feature that allows project managers to directly compare their software engineers in terms of time spent and coding efficiency.

A Java-centric tool called Analyst4j provides an environment to analyse Java code. This software also provides visualisations in the form of graphs and charts to explain the code quality of the code analysed. Analyst4j provides a wealth of features. One can do cause/Pareto analysis to determine which part of the code is doing the heavy lifting, and therefore if the programmer's code is unnecessarily verbose. It can detect antipatterns such as Spaghetti code. Analyst4j is also customisable, and allows one to fine-tune the analysis it runs based on automatic software metrics.

SeaLights is a platform for measuring code quality and activity, specifically how it relates to testing. SeaLights analyses code changes and detects the risks posed by them. Using this, it can measure quality trends over time and determine if a piece of code is dropping in quality. SeaLights also analyses tests, and alerts managers if extraneous or useless tests exist within the project. Furthermore, it can tell managers if there is code that is executed when the project is in production, but not tested by regression tests. This helps project managers to ensure that they are releasing a reliable product and it gives them insight into the performance of their testing teams. As well as being able to monitor the code produced by their software engineers they can determine if their test engineers are up to the job.

Many software engineering analysis tools exist, and the metrics they measure performance by can differ significantly. These computational platforms give managers and engineers themselves a vast range of choice when it comes to monitoring productivity and efficiency. They also ease some of the pain of manual data collection and analysis, which makes them suited to modern software engineering processes.

Algorithmic Approaches for Software Engineering Process Analysis

Given the enormous amounts of data that can be produced by the software engineering process, it follows that there have evolved systematic algorithmic approaches for analysing this data. Indeed there is a vast academic body of work in this area. This is to be expected, given the algorithmic nature of software engineering itself. We will examine some of the approaches that have emerged from this field of study.

Mentioned earlier in this report, BBNs [1] are one systematic approach to analysing software engineering. These graphical networks and associated probability tables use Bayesian probability to model the uncertainty of the subjective variables in measured data, and provide a means with which to interpret data in light of its context. The nodes represent the uncertain variables, and the arcs in the graph model the dependencies between nodes, as the value of a variable may depend on the value of others. These values may be discrete or continuous, as necessary for the specific process being modelled.

There can be a myriad of ways to calculate the values for the associated probability tables, or node probability tables (NPTs). One useful feature of BBNs is that it accommodates both subjective and objective probabilities [6]. The subjective probabilities can come from domain experts, and may be values derived from academic research or small sample size analysis that hasn’t been proven to be generally applicable but is suspected to be so. The objective probabilities can come from empirical research. By combining expert ideas with experimental results, BBNs allow modelling to be tailored to domain-specific needs without losing objectivity.

BBNs are also useful even when there is missing data. This is an important feature as it is frequently possible that data could be missing or erroneously collected and therefore unusable. Many modelling techniques lose their utility when the dataset is incomplete, but BBNs are more robust in that regard. They are an appropriate means of analysing data, and can provide forecasts of future engineering performance.

Line Impact, an algorithmic way of measuring the value of a commit used by the platform Static Object, is another way to assess the software engineering process. Line Impact aims to capture the degree of cognitive load required to create a line of code [7]. This is done by reviewing the action done to a line of code during a commit, and then factors regarding the context in which this action occurred are interpreted into the score. The main actions that can be done are additions, deletions, updates, moves, find-and-replaces and no-ops. The main factors are line-based, file-based, commit-based and branch-based [8].

Actions are assessed according to the impact they have on the codebase. For example, an addition has a high Line Impact score as it involves supplying new code to the codebase. En-masse find-and-replaces have a low impact, as they don’t represent extensive new code. Line Impact scores can also

be updated retrospectively. For instance, if a newly added line of code is updated shortly after first having been added, the impact score for the initial addition is reduced as it clearly had a low impact on the codebase if it required revision almost immediately. This allows changes like additions, deletions, moves and find-and-replaces to have weighted values according to their perceived importance.

Factors also affect the score that a code change might incur. Line-based factors operate on a line by line basis. This means that they are centred around changes like additions and deletions, but they are more nuanced than simply noticing that a line has been changed. If, for example, the only change was to whitespace, indentation or comments the Line Impact score would be unaffected as none of these changes alters the meaning of the code. File-based factors include observing the language that the code is written in. Changes to a C++ file might get a higher score than changes to a Python file because of the relative difficulty of writing code in C++ to writing code in Python. File-based factors also let you ignore specific files or file types, such as those from a third party library that would otherwise skew the analysis. Files can also be assigned different weights for scoring based on their function. It might be useful to give test files a higher weight than code files so that engineers who write more exhaustive tests are rewarded.

Commit-based factors affect the Line Impact score too. For example if a project involves subtrees or subrepos then committing code that is already committed elsewhere won't obtain a score. Commits involving large amounts of lines in a group earn lower scores than a single line change. The reason for this is that new features tend to be when large chunks of code are added, but targeted bug fixes, which warrant a higher score, usually only involve changing a single line or two of code. Line Impact also accounts for merges, and ensures that code is not double counted when it is merged into a branch. Branch-based factors typically relate to ignoring selected branches. Often a branch is created for the sake of experimentation and is not a critical part of the development life cycle. In situations like this you would not want changes to the branch to affect Line Impact scores, so they are ignored. The effects of all of these actions and factors can be tailored by managers to better suit their specific project. In this way Line Impact is quite a customisable yet systematic way of assessing software engineers.

Optimised Set Reduction (OSR) is an approach that allows for the building of models for quality assessment [9]. OSR is based on pattern recognition techniques. A learning sample consisting of N vectors with n independent variables and one dependent variable is supplied to the algorithm to train it. A measurement vector with values for each of the independent variables for a particular object whose dependent variable is to be calculated is what the prediction is run on. Both the independent and dependent variables are divided into classes to facilitate prediction-making. The patterns that will be recognised take the form of non-uniform distributions of probabilities on the dependent variable's classes.

OSR is used in quality assessment by comparing the data of the current engineering process to some baseline. The independent variables represent some project features and the dependent variable can be productivity or defect rate or some similar aspect of the process that needs to be quantified. By comparing the value of the dependent variable to its expected value, the quality metric can be assessed. It is also possible to use this method to determine which of the independent variables it is that affects the dependent variable the most, so as to ascertain which aspect of the engineering process it is that needs the most attention.

Ethical Concerns over Data Collection in order to Analyse the Software Engineering Process

Due to advancements in computer technology, the acquisition of data has been occurring at unprecedented levels in recent years. Companies are able to gather vast amounts of data about individuals and use this data to infer incredibly accurate information about them. While it is clear that there are benefits to this, such as being shown relevant items when shopping online, there are growing concerns over how companies manage this data. One need only look to the Cambridge Analytica scandal involving Facebook to ascertain the dangers of mass data collection and the public's attitude towards its misuse, especially as laws in many jurisdictions were not designed to regulate it. When laws are inadequate to deal with circumstances surrounding data collection, companies and individuals must turn to ethics to determine what is and is not acceptable. This applies to data collection in any scenario, including for the purposes of analysing the software engineering process.

Analysing the software engineering process necessitates the analysis of software engineers. Therefore it requires the collection of personal data about the engineers, such as how long they work for, what they do when they are at work, how much code they write, etc. Naturally some engineers have privacy concerns about their data being collected, particularly when it is done by automated software. When data is collected, the person who the data is about loses control over the information. The most basic reservation about this loss of control is security. If the data is not stored securely, one runs the risk of having information about themselves fall into the hands of those with potentially nefarious motives. Data collectors, in this case software engineering companies, have the responsibility to ensure that no unauthorised personnel gain access to their gathered data. Improper security measures could lead to conflict between a company and its employees as the employees may feel that their employers are belittling them by not taking their concerns seriously. Most nations have basic laws regarding the minimum amount of security needed to legally collect data, but often this isn't enough. It is a judgement call by the company as to what constitutes adequate security, but the cost of implementing the security measures could cloud the decision making process. Ultimately it comes down to a question of the subjective culpability that the company would feel if the data was stolen, which often may not align with the engineers' views.

Another ethical concern is what the data itself is used for. As already mentioned companies are able to learn a lot from the data they collect thanks to algorithms and tools like those discussed previously. They have the ability to build a very accurate and perhaps invasive profile of their engineers. These profiles can be used to decide a range of things, from who gets a promotion to which tasks are assigned to whom at the beginning of a new project. Engineers may feel aggrieved at such a process, as often automated assessment fails to accurately capture context. For instance, if the number of commits made by an engineer was significantly lower than that of his peers, any analysis would rank him as a less competent engineer without necessarily giving the reason why this is the case. If that engineer had spent a considerable amount of his time helping a new colleague who was unfamiliar with the project or language used, the context shows that the engineer is in fact a valuable resource even though the analysis shows otherwise. When evaluation is trusted to an algorithm , nuance in interpreting the results is often lost, potentially to the detriment of those being evaluated. A company must balance the ease of use of automated tools with the subjectivity

natural to human analysis, or risk the ethical conundrum caused by allowing machines alone to inform decisions.

It is challenging enough to determine whether or not making decisions based on automated analysis is ethical, but when the validity of the measurement being used is called into question – as it often is when it comes to software engineering – it becomes an even more complex question. This report has argued that the software engineering process can be measured and assessed, but to what degree the measurements are true reflections of the data they interpret is still not unequivocally known or accepted. Therefore , the question remains: is it right to judge someone on data that may or may not be valid? If one takes the view that software engineering is too complex to be fully or accurately captured by the metrics established in the literature then the argument naturally follows that these metrics can't be trusted to accurately describe software engineers. This leads to the issue of whether companies have the right to make decisions about their engineers based on this flimsy data. The validity of the whole process can be called into question. This presents an ethical dilemma, as companies will be judging engineers on data that can be argued does not reflect them.

Clearly there are many ethical concerns when it comes to measuring software engineers. While there have been some advancements in recent legislation, namely the European Union's introduction of GDPR, many of the decisions taken regarding the use of data comes down to the ethical views of those in charge. If these views do not align with those of the engineers, there is the potential for conflict.

Conclusion

There is a strong desire amongst project managers and software development companies to measure and analyse the productivity and efficiency of software engineers, and the software engineering process as a whole. There are myriad means of procuring the data, both manual and automated, to be analysed. Automated data collection is generally preferred for its unobtrusiveness and lower likelihood of producing missing or erroneous data. When it comes to analysis, not only is an automated approach far superior, but there exist many computational platforms with which to conduct this analysis. These platforms offer a wide range of metric choices, and allow the analysis of everything from the number of commits to a breakdown of the time spent coding by programming language. While some of these platforms simply collate and display the data, others take an algorithmic approach to its analysis. Approaches like BBNs or Line Impact have different advantages, but they all allow for a richer analytical experience. Regardless of the approach taken, ethical concerns must be considered before, during and after the analysis. Whenever data about an individual is collected it must be ensured that the data is not misused, although what that means is subjective.

This report has considered the ways that the software engineering process can be measured and assessed, the computational platforms for doing so, the algorithmic approaches available and the ethical concerns of this activity.

References

[1] Norman E. Fenton, Martin Neil, 1999. Software metrics: successes, failures and new directions. Centre for Software

Reliability, City University, Northampton Square, London EC1V OHB, UK.

[2] Philip M. Johnson, 2013. Searching under the streetlight for useful software analytics. University of Hawaii at Manoa.

[3] Kazuo Yano, Dr. Eng., Tomoaki Akitomi, Koji Ara, Dr. Eng., Junichiro Watanabe, Dr. Eng., Satomi Tsuji, Nobuo Sato, Ph.D., Miki Hayakawa, Norihiko Moriwaki, Dr. Eng., 2015. Measuring Happiness Using Wearable Technology.

[4] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, William E.J. Doane, 2003. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. Collaborative Software Development Laboratory, Department of Information and Computer Sciences, University of Hawai'i.

[5] Gregor Grambow, Roy Oberhauser, Manfred Reichert. Automated Software Engineering Process Assessment: Supporting Diverse Models using an Ontology. Aalen University (Grambow, Oberhauser), Ulm University (Reichert).

[6] Norman E. Fenton, Martin Neil, 1999. A Critique of Software Defect Prediction Models.

[7] Kevin Andrews, 2018. What is Line Impact, and how does it work?https://support.staticobject.com/hc/en-us/articles/115001729493-What-is-Line-Impact-and-how-does-it-work-

[8] Line Impact: What Powers Static Object. https://www.staticobject.com/line_impact_factors

[9] L.C. Briand, V.R. Basili, W.M. Thomas, 1991. Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland.