Measuring Software Engineering

Abstract

This report considers four interlinked aspects of measuring the process of software engineering in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethical concerns surrounding this type of analytics.

Introduction

In order to measure software engineering, we must first understand the fundamentals of what software engineering is and the process which it takes.

Software engineering is defined by the IEEE (Institute of Electric and Electronics Engineers) as the systematic application of engineering approaches to the development of software. The discipline has grown almost exponentially since the introduction of the first digital computers in the 1940s. Software engineering now reaches almost all aspects of modern life, with many depending on it daily. This meteoric rise means that it is more important than ever to measure, assess and analyse software engineers accurately, in order to ensure that they best possible software is being generated.

The process of developing software is commonly split into the 5 stages of its life cycle and involves a series of tasks that must be carried out regardless of the type of software being developed or the model being used (waterfall, agile, iterative etc.). Each stage can be a long and complex process and may have to be revisited multiple times, requiring engineers' time and effort. This costly process must be analysed and assessed in order to improve the process and efficiency of future development.

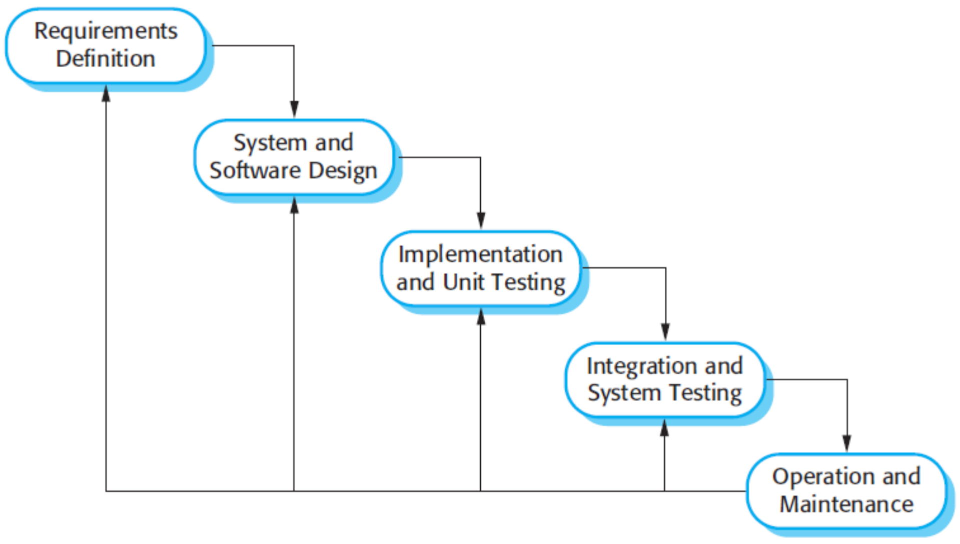

Requirements Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 1: The 5 Stages of the Software Development Life Cycle

Before we consider the ways in which software engineering is measured it is worth briefly pondering why we are interested in measuring the process so diligently in the first place. Software engineering, whether we are aware of it or not, touches almost every aspect of our lives and behind every program is often an army of software engineers. These software engineers represent a significant expense to firms who are keen to monitor performance with the aim of identifying the best engineers and improving the overall efficiency of their workforce.

Measurable Data

Source lines of code
Measuring source lines of code has long been a popular metric in estimating the cost of developing a system and the output of an individual engineer and was one of the original methods of evaluation. In its simplest form the principle of this measure is, the greater the number of lines of code, the greater the engineer.

This is obviously a highly flawed methodology as since the beginning of software engineering we have greatly moved forward to valuing simple and easily comprehendible code as opposed to long, drawn out code. By measuring engineers this way, they are incentivised to add unnecessary lines of code, or even make the code more complicated so as to generate more lines of code. In an academic paper produced by Bhatt, Tarey and Patel, 'Analysis of Source Lines of Code (SLOC) Metric', the authors identify nine key flaws in using this metric:

1. Lack of Accountability
Projects as a whole cannot be measured by this metric as the coding phase usually accounts for only 30 to 35% of the overall effort. This allows the other 2/3 of the project to go unmonitored and unmeasured, lacking accountability.

2. Lack of Cohesion with Functionality
Although effort is highly correlated with SLOC, functionality is not. The most skilled developers can often develop the same functionality with far less code and hence someone who develops only a few lines may still be more productive that a developer creating more.

3. Adverse Impact on Estimation
Cost and time estimates for projects, when based off lines of code, can be severely awry because, as under point one, the coding phase usually only accounts for about a third of the overall effort of a project.

4. Developer's Experience
Implementation of a specific logic differs based on the developer's experience, allowing more experienced developers to produce relatively fewer lines of code with the same functionality than the less experienced engineer.

5. Difference in Languages

When comparing two applications that provide the same functionality but are written in two different languages, the lines of code needed to develop each application would vary significantly.

6. Advent of GUI Tools

GUI-based programming languages have resulted in programmers being able to write relatively little code and achieve high levels of functionality, because of the in-built functionality that these tools provide. Code that is automatically generated by a GUI tool is not taken into account when using SLOC methods of measurement.

7. Problems with Multiple Languages

Software is now often developed in more than one language and the languages employed usually depend on the complexity and requirements of the software system at hand. The tracking and reporting of productivity and defect rates cannot be attributed to a particular language subsequent to integration of the system.

8. Lack of Counting Standards

There is no standard definition of what a line of code is. Do comments count? Are data declarations included? What happens if a statement extends over several lines? There are some organizations such as the SEI who have published guidelines in an attempt to standardize counting, however with the frequent introduction of new languages it is difficult for governing bodies to keep up.

9. Psychology of a Programmer

A programmer whose productivity is lines of code will have an incentive to write unnecessarily long and complex code. This is undesirable since increased complexity can often lead to an increased cost of maintenance and increased effort required for bug fixing.

Code churn

Code churn is a measure that tells you the rate at which your code evolves, it measures the number of source lines of code that were modified, edited or deleted over the development life cycle. It has two key usages:

1. Visualizing the Development Process

If the code churn spikes, this would indicate that there may be an issue in the process and that there is a mini waterfall going on in your daily work.

2. Reason about Delivery Risks

Code churn is a good predictor of post-release defects and can help not only measure the process up until the release of the software but can also predict the number of hours of maintenance and bug fixing required after release.

Number of commits

Measuring the performance of a project by counting the number of commits made by developers is unsurprisingly a flawed, and relatively redundant, approach. While commits can be used to measure consistency of development, commits have no bearing on the quality or performance of the code being committed and hence is a pointless metric.

Agile metrics (lead time/cycle time)

Agile metrics are standards that help a team monitor how productive they are across different phases of the SDLC. Unlike measuring SLOC, these metrics cover every stage of the project and can be split into two key metrics of lead time and cycle time.

Lead time is defined as the time elapsed between the identification of a requirement and its fulfilment. It measures the total time it takes for the development process to be completed, from stage one of the SDLC (Requirements Definition), all the way through to delivery, however, does not count the final stage of the SDLC (Operations and Maintenance) in this metric. It is a useful metric in order to estimate completion time for future projects and can also be useful in terms of evaluating the capability of an individual developer and their ability to deliver on deadlines.

Similar to lead time, cycle time is the time elapsed between the commencement and fulfilment of a project, the key difference being that the timer starts when the software engineer commences work as opposed to when the request by the client is made. Both lead time and cycle time are relatively easy to measure and usually requires a single team member to record when the project is received, commenced and delivered.

Code coverage

Code coverage is a quality assurance tool that measures code with test code. The higher the percentage of code coverage the less likely there is to be a bugs and other issues. The effectiveness of code coverage as a metric is limited by the quality of testing that is developed as even if code coverage is 100%, there may still be a variety of possible scenarios that have not been tested for and may lead to bugs.

Technical Debt

Technical debt is the cost incurred when poor design or implementation decisions are taken for the sake of moving fast in the short-term, instead of a better approach that would take longer but preserve the efficiency and maintainability of the codebase. As with monetary debt, if technical debt is not repaid it can accumulate ‘interest’ making it harder to implement changes. Hence, a good engineer will pay back technical debt promptly with a rewrite. Technical debt is usually expressed as a ratio. The cost to fix a software system (remediation cost) to the cost of developing it (development cost).

Computational Platforms

Once the desired data has been collected, managers need computational platforms to automatically assess and analyse the data, as manual analysis is both resource intensive and often severely flawed. These platforms interpret the data at a basic level and the best platforms for individual managers are often highly linked to the type of data they have collected and the type of analysis they are looking to get out of the computational platform.

Personal software process (PSP)

The Personal Software Process is a structured software development process, designed by Watts Humphrey, that is designed to help software engineers better understand and improve their performance by bringing more formal discipline to the way they develop software and track their predicted and actual development of code.

The Personal Software Process has four levels:

1. *PSP 0*

The first level includes personal measurement, basic size measures and coding standards.

2. *PSP 1*

This level includes the planning of time and scheduling.

3. *PSP 2*

This level introduces the personal quality management, design and code reviews.

4. *PSP 3*

The last level of the PSP is for the personal process evolutions.

In the PSP, the level of success is dependent on the self-motivation of the individual engineer to continually monitor their own work and make note of their statistics. The PSP theory is that it is a more economical and effective method of removing defects as they are detected by those who wrote the defective code, hence software engineers are encouraged to conduct reviews, both design and code, for each phase of development and before their work undergoes a peer or team review. This makes PSP one of the cheapest computational platforms.

The PSP has however also been critiqued due to its manual nature, introducing large amounts of overhead for the developer. The need for the developer to manually collect, analyse and collate data for post-mortem reports is very time consuming and is often not where the software developers' skills are really being put to full use. Furthermore, the entire PSP process is only focused on source code and hence if the developer does not produce source code, they cannot use PSP.

Leap Toolkit

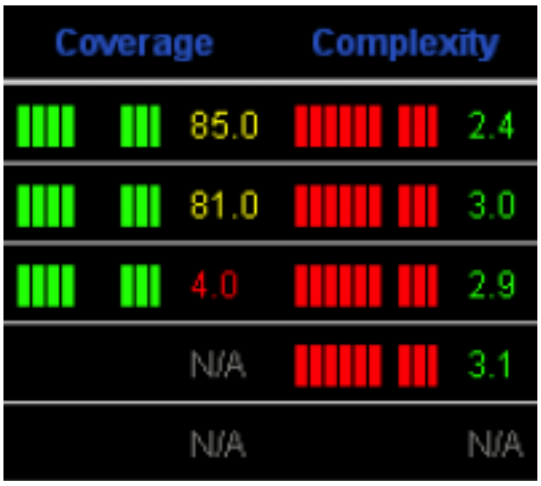
The LEAP (Lightweight, Empirical, Anti-measurement and Portable software process measurement) toolkit is an automated tool, developed at the University of Hawaii, that supports personal developer improvement. It incorporates ideas from the PSP and group review while reducing the process overhead and relaxing some of the constraints of the PSP.

In this toolkit the software engineer must still manually input the data, but unlike the PSP, the subsequent analysis is generated automatically. It creates a repository of individual process data that can be used from project to project.

The Leap toolkit also allows the developer to define their own processes, unlike PSP where they must use the prescribed flow. The developer can also define processes for non-software development activities including planning, writing and editing.

Hackystat

Hackystat is an open source framework for the collection, analysis and interpretation of software development process and product data. It uses software sensors which typically attach to development tools and unobtrusively collect and send raw data about development to the Hackystat SensorBase for storage. These databases can then be queried to generate visualisations and insights, which makes it particularly easy to integrate with repositories such as GitHub. The long-term goal of Hackystat as a whole is to facilitate "collective intelligence" in software development by enabling the collection and subsequent analysis of data to generate useful insight for the wider software engineering community.



Coverage	Complexity
85.0	2.4
81.0	3.0
4.0	2.9
N/A	3.1
N/A	N/A

Fig 2: Example of Hackystat displaying code coverage and complexity for a sample input

One critique that is levied at Hackystat is that the data collected from both the client and the server can be seen as extremely intrusive as users are often unaware that data is being collected.

Code Climate

Code Climate is one of a number of systems built on the foundations of Hackystat to provide an automated data collection system. Complex analytical techniques are applied to create digestible insights for the user, usually in the form of a graph. The main function of Code Climate is to review code and ensure that it is easily understood, it doesn't repeat itself and can be modified and maintained with ease. It allows organisations to properly analyse the quality of their code through full testing and reducing code complexity.

Code Climate can also be integrated with GitHub and BitBucket repositories to provide further insight into their code coverage, technical debt and progress reports and can provide analysis on code written in a variety of different languages including Python, PHP and JavaScript.

Algorithmic Approaches

So far in this paper we have analysed how and why to collect the data however we now need to leverage this data to generate actual insights. Algorithmic models are needed to analyse the data at a higher level and there are a variety of different models that we use to do this. These algorithms can identify patterns and trends which are useful for companies in analysing employees and predicting future output. Currently some of this is done by human analysis however, machine learning and AI are making large strides in improving this analysis which I will cover in this section.

Computational Intelligence

Computational intelligence (CI) refers to the ability of a computer to learn a specific task from data or experimental observation and then apply it to future tasks/problems. The approaches to CI are generally inspired by natural processes in order to solve real-world issues in which traditional mathematic reasoning cannot be applied, usually because the process contains too many uncertainties, or the process is stochastic by nature.

As identified by Siddique and Adeli (2013), there are five main approaches to computational intelligence:

1. *Fuzzy Logic*

This is the measurement and process modelling that can accommodate uncertainties such as incompleteness, randomness and ignorance of data which makes it particularly useful in the real world where this is often the case. This method usually deals with reasoning and inference on a higher level, such as semantic or linguistic.

2. *Neural Networks*

These are not algorithms as such but act as a framework for machine learning algorithms to work together and process data inputs and usually demonstrate good performance in processing numerical data. The learning takes place in different forms in neural networks, such as supervised, unsupervised, competitive and reinforcement learning.

3. *Evolutionary Computation*

This group of algorithms is based on the theory of natural selection and survival of the fittest first introduced by Charles Darwin. Applications of this kind are found when the problems are too complex for traditional mathematical techniques to be used so computational intelligence must be used to compute a new solution.

4. *Learning Theory*

These algorithms attempt to replicate human learning which is comprised of the processing of cognitive, emotional and environmental experiences to acquire knowledge and skills (Omrod, 1995). It is one of the main approaches to CI and helps us to make predictions based on previous experience.

5. *Probabilistic Methods*

One of the main elements, probabilistic methods were first introduced by Erdos and Spences (1974) in order to evaluate the outcomes of a CI system, mostly defined by randomness. Probabilistic methods therefore present possible solutions based on prior knowledge.

Multivariate data analysis
Multivariate analysis is based on the principles of multivariate statistics which involves the observation and analysis of more than one statistical outcome variable at a time.
The process is generally split into two main, distinct methods: supervised and unsupervised learning (Brownlee, 2016).

Supervised learning is where the algorithm generates a function that maps inputs to desired output. It is considered "supervised" as we teach the algorithm what conclusions should be drawn from a given set of data by using training datasets to teach it. The algorithm uses trial and error, where it iteratively makes predictions based on the data set at hand and then corrects any outlying predictions, repeating the process until the desired accuracy is achieved. The two-main classifications of supervised learning are K-nearest neighbours, a non-parametric method of classification, and linear discriminant analysis, which allows us to place multivariate data into classification groups.

Unsupervised learning models a set of input features and maps them to similar patterns, like clustering and outputs an unknown. It is generally referred to in terms of machine learning as the algorithm learns to identify complex processes and patterns from inputted data and then maps them to similar patterns to produce an unknown output. Unsupervised learning algorithms are generally defined as either clustering algorithms or as principal component analysis algorithms. Clustering algorithms divide a set of clusters according to their shared characteristics and therefore identifying a group structure in the dataset. In principal component analysis, a dataset with a significant number of different aspects is distilled into something that captures the essence of the original data, identifying which aspects cause the most variation in the data. This method helps explain relationships between variables.

Ethical Concerns

Those that measure the work of software engineers have a duty to carry out their work ethically. While many companies define “ethically” differently depending on their business priorities, the three key areas in which most focus is placed is on data collection, regulation and sovereignty.

Data collection

Data collection is often the most widely talked about ethical concern surrounding software engineering. Hyped on by various documentaries, such as The Social Dilemma which highlights the vast quantity of data being collected through social networking sites and other sites, data collection has reached the public consciousness. Software engineers are likely to feel more in control if they know what data is being collected on them, for what purpose and how it is being used. Without an understanding of why they are being so closely monitored, software engineers might begin to being like they are just a statistic to the company and become unmotivated, or even worse become resentful if they feel the company is monitoring them so closely because they don’t think the job is being done well. By giving engineers some ownership over their own data and by encouraging them to use their own data for their own benefit and professional development, engineers might begin to feel more like the data is working with them than against them.

Various very public data leaks, such as Ashely Maddison and British Airways, has also given rise to the concern about the strength of data security. Even once consent has been given to collect their data, very few people know how it is then being stored and secured, even software engineers. Wider awareness about how, where and for how long data is being stored may put some people at ease.

Data regulation

In May 2018, the EU implemented the General Data Protection Regulation (GDPR) in order to regulate the transfer of personal data outside the EU and give control to individuals over their personal data. This law has greatly strengthened the protections surrounding data collection and requires companies to be much more transparent on this matter, requiring user consent to hold, store and utilize their data. Those which do not comply with the new rules face high fines, such as Google Inc. who were fined €50 million for unlawful use of personal data. Not only has GDPR strengthened regulation about data privacy on the internet, but it has also educated the wider population on their rights and raised awareness about the quantity of data being collected on you.

Data sovereignty

Data sovereignty is the notion that data is subject to the laws within the nation it is collected and was encapsulated within the GDPR laws when they were created in 2016. This generally prevents foreign governments from accessing data originating in another country although some governments, frequently the US, have tried to bypass these laws unsuccessfully in the name of national security.

Conclusion

This report has considered the four interlinked aspects of measuring the process of software engineering in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethical concerns surrounding this type of analytics. While these aspects of measurement give us a good understanding of the fundamental metric of software engineering, there is still a long way to go in term of being able to quantify the intricacies of software engineering. Key aspects that can have a large impact on the process such as creativity, innovation, ability to work collaboratively are often overlooked in measurement despite the importance of these "soft skills". Many developers see the process as an art not a science, and therefore believe it should be measured in a similar way, where the nuances of the process are taken into account.

While measuring software engineering is often expensive, time-consuming and difficult, it is a necessary evil in order to ensure that the software development process works for everyone, the managers, the developers and ultimately those that use the software.

Bibliography

Bhatt, Tarey and Patel, 2012. ***Analysis Of Source Lines Of Code (SLOC) Metric***. [online]

Siddique and Adeli, 2016. ***Brief History Of Natural Sciences For Nature-Inspired Computing In Engineering***. [online] Taylor & Francis.

Brownlee, J., 2016. ***Supervised And Unsupervised Machine Learning Algorithms.*** [online] Machine Learning Mastery.

Ormrod, J., 1995. ***Human Learning***.

Erdös, P. and Spencer, J., 1974. ***Probabilistic Methods In Combinatorics***. Budapest: Akadémiai Kindó.

Ieeexplore.ieee.org. 2020. *610.12-1990 - 610.12-1990 - **IEEE Standard Glossary Of Software Engineering Terminology** - IEEE Standard*. [online] Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=159342> [Accessed 21 November 2020].

GeeksforGeeks. 2020. ***Personal Software Process (PSP)** - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/personal-software-process-psp/> [Accessed 21 November 2020].

Moore, C., 1999. ***Teaching Software Engineering Skills With The LEAP Toolkit***.

Siddique and Adeli, 2013. **Synergies of Fuzzy Logic, Neural Networks and Evolutionary Computing.**

Fox, C., 2019. ***Google Hit With £44M GDPR Fine Over Ads***. [online] BBC News. Available at: <https://www.bbc.com/news/technology-46944696> [Accessed 21 November 2020].

Johnson, P., 2007. **Requirement and Design Trade-offs in Hackystat:** An In-Process Software Engineering Measurement and Analysis System. In: *Empirical Software Engineering and Measurement Symposium*.

Figures
Figure 1: https://medium.com/@a01633605/the-ultimate-software-engineering-guide-part-1-34e691f843e9

Figure 2:
https://www.google.com/url?sa=i&url=https%3A%2F%2Fgithub.com%2Fhackystat&psig=AOvVaw1C9nQRuEzKU-feJN8iem6A&ust=1606066643049000&source=images&cd=vfe&ved=0CA0QjhxqFwoTCPCK3tCWlO0CFQAAAAAdAAAAABAD