Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Measuring Software Engineering

Ammar Magomadov

18327041

Software Engineering (CSU33012)

Professor Stephen Barrett

23/11/2020

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Introduction

"Software engineering is the process of analysing user needs and designing, constructing, and testing end-user applications that will satisfy these needs through the use of software programming languages." [1] Nowadays software engineering is one of the fastest growing professions with almost 27 million developers in 2020. It is estimated the software engineers will number "45 million in 2030." [2] As the number of software engineers continues to rise rapidly along with the massive strides being made in technology, the need to make the software engineering process faster and more efficient is being made evident. This brings up the question, how *do* we measure software engineering?

Measuring software engineering is not a new idea, in fact it has been going on for quite some time now. There have been many methods used in the past to measure the software engineering process. These methods can be used to find the conditions in which software engineers are most productive or even to distinguish productive software engineers from those who are not. As more and more companies have started to adopt methods of measuring software engineering, more and more questions are being asked. How do we deduce the productivity of a software engineer? How can we judge the product of the software engineering process? Are these methods ethical? This report will discuss the measurable data, computational platforms, algorithmic approaches available and the ethical concerns.

Measurable Data

Source Lines of Code (SLOC)

Counting the number of lines of code written is the simplest metric that can be used to measure the software engineering process. In essence, the more lines the programmer writes, the more productive he is. This method is used by many due to the fact that it can be easily automated. There are many utilities out there that can count the SLOC in a program.

However, this approach to measuring productivity is flawed. When writing code, the goal is to write code that is efficient and easy to understand. Writing more code doesn't necessarily mean that one was more productive. In an article titled 'Analysis if Source Lines of Code (SLOC) Metric', written by Bhatt, Tarey and Patel, the flaws of SLOC were pointed out.

These flaws have been summarised and inserted below:

- ***Lack of Accountability:***
  - Only 30% to 35% of the overall effort is used to write code.
- ***Adverse Impact on Estimation:***
  - Due to the lack of accountability, estimates based on lines of code can adversely go wrong, in all possibility.
- ***Lack of Cohesion with Functionality and Developers Experience:***
  - An experienced developer may implement certain functionality in fewer lines of code than a developer with less experience would. The experienced developer would still be considered the more productive one.
- ***Difference in Languages:***
  - If one application is written in two languages, e.g. C++ and COBOL, the number of function points would be the same, but the lines of code needed to develop the application would certainly be different. Therefore, the amount of effort required to develop the application would also be different.
- ***Lack of Counting Standards:***
  - There is no standard definition of what counts as a line of code. Do comments count as lines of code? Do statements that are extended over several lines count? Organizations like SEI and IEEE have published guidelines in an attempt to standardise counting but they are difficult to put into practice.
- ***Psychology of a Programmer:***
  - If a programmer's productiveness is measured based on the lines of code they write, then programmers would be inclined to write unnecessarily verbose code. This then makes the code more complex and as a consequence, more time would be needed for debugging and the cost of maintenance would also increase.

Considering the above, there is too much uncertainty surrounding this form of measurement. While writing many lines of code could very well show that a developer is productive, it could also be an indicator for inefficient code. The same can be said for writing little code, it could either be very efficient or it could just be lacklustre code.

Number of Commits

Another metric often used for measuring the software engineering process is based on the number of commits one does. This process is also used by many since there are tools, such as GitClear, capable of analysing repositories to calculate productivity using the number of commits as data. However, counting the number of commits is not a reliable metric since it all comes down to personal preference. Some developers prefer to commit often, while others don't.

When working in a software engineering environment, a developer might have set tasks that he needs to complete. Some developers may choose to get the task done as quickly as possible and move on to the next while others may take their time and make sure that the code, they've written will not need reworking. If we were to measure these two developers based on the number of commits they've made, it wouldn't be an accurate measure of how productive they were. As we can see, this metric doesn't consider the difference between the tasks that each developer had to complete and doesn't give us any context as to why one developer is committing far more often than the other. The frequency of commits does not correlate with the work needed to complete the task.

Cycle Time

"Cycle time, a metric borrowed from lean thinking and manufacturing disciplines, is simply the time it takes to bring a task or process from start to finish." [3] This doesn't necessarily need to be the time it takes to start and finish a whole project. Within the project you could possibly have multiple tasks that have their own cycle time.

Measuring cycle time is quite easy, it only requires one person to take note of when the task was started and finished. Since a software engineering team tends to have so many tasks going on at once, the measuring of cycle time will need to be automated. It would require too much effort to do it manually at that scale. If you don't manage to automate this system for big projects, extracting enough useful data will be difficult.

When cycle time is recorded in frequent intervals, it becomes much easier to identify where and when problems occur. New tasks can be compared against previous similar tasks to provide an estimate for the time required for completion. It could also be used to measure a developer's capability, maybe his development practices or the software tools he's using are reducing his productivity.

Code Churn

Testing, rewriting and trying multiple solutions for a program is a part of a software engineers' job. This software engineering metric assesses the number of edits made in a particular area of code. This method doesn't measure the quality of the code, but it can be used to signal if something is wrong with the code. If the churn spikes, it indicates that something may be off since it had to be edited repeatedly.

Code churn is not the same for everyone though. Different teams will have different levels of 'normal' code churn. The manager has to figure out what 'normal' is for each team so he can identify when it starts to spike. This will allow him to identify the problem and find a solution quickly. "Watching trends in code churn can help managers notice when a deadline is at risk, when an engineer is stuck or struggling, or when there are issues concerning external stakeholders." [4]

Computational Platform

Now that metric tracking has become so widespread, various tools and platforms have been developed to automate the measuring of the software engineering process. These tools are used to collect various types data that can be used to analyse the teams and their productivity.

Github

Github is a web-based version-control and collaboration platform used by almost all software-oriented organisations as well as individual developers of varying expertise. Github measure various types of data including the number of commits made on a project, number of changes made to a code base, number of developers contributing to a project etc. The REST API can be downloaded and used to obtain all this information. This information can then be used by managers to identify valuable team members. Platforms such as GitClear and GitColony can be integrated with Github to create visual dashboards and automated reports using the metrics obtained by Github.

- **GitClear**
  "GitClear amplifies how much software teams get done. We analyse git commits to simplify code review, identify tech debt, and promote subject matter expertise. GitClear separates from the competition through its detailed analysis of data quality factors, some of which can lead to data pollution." [5]
  GitClear can give in depth insights into the performance of software engineers. It analyses git repositories, collects all the data and combines it to calculate team output in the form of a new metric called 'line impact'. GitClear can also review your code leaving you with more time to code.

- **GitColony**
  According to GitColony themselves, using GitColony can save 360 hours a month in a 10 full-time devs team working 180 hours a month. GitColony gives teams the ability to perform small reviews frequently to avoid huge reviews before deploying. This is down to the fact that GitColony allows for the code to be reviewed as its being written. It also saves the location of where you stopped reviewing allowing developers to pick up where they left off at a later stage.
  GitColony also allows you to make virtual pull requests which only exist inside of GitColony. The team can then review your code and if they feel that the code is ready, they can vote to merge. At this point the virtual pull request is transformed into a standard one and can be merged like you normally would.

Personal Software Process

The Personal Software process was created by Watts Humphrey in 1995. Humphrey explained what PSP is in his report:

"The Personal Software Process (PSP) provides engineers with a disciplined personal framework for doing software work. The PSP process consists of a set of methods, forms, and scripts that show software engineers how to plan, measure, and manage their work. It is introduced with a textbook and a course that are designed for both industrial and academic use. The PSP is designed for use with any programming language or design methodology and it can be used for most aspects of software work, including writing requirements, running tests, defining processes, and repairing defects. When engineers use the PSP, the recommended process goal is to produce zero-defect products on schedule and within planned costs. When

used with the Team Software Process (TSP), the PSP has been effective in helping engineers achieve these objectives." [6]

Humphrey published his technique in an attempt to help software engineers understand and improve their code by tracking their predicted and actual development of code. Humphrey didn't like the idea of writing code, testing it and fixing the code until all the tests passed. He felt that too much time was wasted trying to fix the code. He believed that developers should be writing quality code right from the start instead of relying on test code.

Using PSP allows developers to evaluate themselves and figure out where they need to improve. The PSP improvement process is outlined in eight steps:

- Define the quality goal
- Measure product quality
- Understand the process
- Adjust the process
- Use the adjusted process
- Measure the result
- Compare the results with the goal
- Recycle and continue improving [7]

Although PSP is a very useful tools to have, most of the information needed has to be put in manually. PSP uses forms to collect information such as project plans, design checklist, defect recording log etc.

Hackystat

Hackystat is an automated alternative to Personal Software Process. It collects much of the same information as PSP, such as number of commits, time spent on the project, code complexity, code coverage etc. "Hackystat is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data."[8] Hackystat automatically collects a large amount of developer information at frequent intervals. Sensors are attached to the tools of developers which then proceeds to collect raw data, every 30 seconds, about the development process. Hackystat can also be integrated with Github to collect a more in-depth insight into the developer. This data is then sent to the Hackystat SensorBase to be stored. The stored data can be used to create a visual representation of the development process.

Timeular

"The Timeular Tracker is an 8-sided dice that sits on your desk. Assign an activity to each side and flip to start tracking your time." [9]

Timeular provides ones of the easiest ways to manage your time. Timeular records the time you spent on a certain task which you can then view on their mobile app. The data can then be exported as per your needs to make accurate reports or timesheets. Timeular also allows supports team time tracking. Every member of the team uses the tracker and all the data is recorded in one place, ready to inspect.

"**Track common activities and keep data consistent**: Assign common activities and tags to your Shared Spaces and track uniform data together. Filter, analyse and action without moving or changing data." [10]

"**Powerful analytics for team-tracked time**: Spot team time trends, identify bottlenecks, discover clients costing you money and more." [10]

Testrail

"A complete web-based test case management solution to efficiently manage, track, and organize your software testing efforts." [11] Testrail can easily integrate with many frameworks, including Github and Bitbucket among others, to automatically manage your test cases. It gives developers the ability to create and manage test cases and monitor the test results. Testrail will keep a record of all your test case history to ensure that there are baselines for multiple branches. Using this platform will save quality assurance members a great deal of time by having most of it automated. This will allow developers to quickly check for the desired amount of code coverage and to fix any bugs that may be found.

Algorithmic Approaches

Halstead Complexity

In 1977, Maurice Howard Halstead introduced the Halstead complexity measure which are software metrics. Halstead's metric is a static analysis of code and is used in many tools/platforms that count lines of code. Halstead's metric measures the program volume, program length, program effort and estimated program level based on the number of operators and operands.

For a given program, let:

n = no. of distinct operators in program

n2 = no. of distinct operands in program

N1 = total number of operator occurrences

N2 = total number of operand occurrences [12]

Using these variables, we can get the following definitions:

Program vocabulary: n = n1 + n2

Program length: N = N1 + N2

Program volume: V = N * $\log_2 n$

Program difficulty: D = (n1 / 2) * (N2 / n2)

Program effort: E = D * V

Specification abstraction level: L = (2 * n2) / (n1 * N2)

Time required to program: T = E / 18 seconds

Number of delivered bugs: B = ($E^{2/3}$) / 3000

Program volume describes the size of an implementation of an algorithm. The result should be over 20 but not more than 1000, anything over 1000 means that the functions is trying to do too many things. The number of delivered bugs in any file should be less than 2.

Halstead metrics were heavily relied on at the time of its inception in calculating software complexity. These days, Halsteads metrics are used as a maintenance metric but when used alongside algorithmic approaches like Cyclomatic Complexity, they can be used to understand the complexity of programs in much greater detail.

Cyclomatic Complexity

Cyclomatic complexity, a metric used to measure the complexity of a program, was developed by Thomas J. McCabe in 1976.

"Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors. It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module." [13]

Cyclomatic complexity can be obtained from flow graphs by using the formula:

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

Where:

E = the number of edges

N = the number of nodes

P = the number of nodes with an exit point

When measuring cyclomatic complexity, it is better to have a low McCabe number. Having a high McCabe number indicates that the code is difficult to understand and has a higher chance of containing bugs. "The cyclomatic complexity number also indicates the number of test cases that would have to be written to execute all paths in a program." [14]

Ethics

When dealing with data collection, ethics is constantly involved. What type of data is ok to collect and what is it? Can we trust management to deal with this information responsibly? Many people, including developers, are not comfortable having their data collected without their consent and rightfully so. The data being collected may be sensitive. If it were to be placed into the wrong hands it could easily be used against them.

Collecting data is not always a bad thing though, it just depends on the type of data being collected. There are some excellent tools out there that can be used to help you improve your decision making or to aid in identifying problem areas that need to be worked on. The problem arises when the people are constantly being monitored on everything they do. What they are doing, how they are doing it, where they are doing it and how they are spending their time. If management decides to implement platforms to measure the software engineering process in this way, it is likely that developers will feel like they are under constant scrutiny. Hence the need for transparency in regard to the data being collected.

The other issue with data collection is that the data needs to be handled and utilized in a responsible way. It needs to be used to improve the software engineering process in a moral and non-discriminative manner. There are companies out there that track your health as well as your work. This type of data could easily be used against the employee. For example, if the data shows that an employee's health is starting to decline, they may decide to remove him from the health benefits. If the data is not utilized appropriately, it could impact the work and overall happiness of developers. According to the many studies that have been done, happiness is directly correlated with productivity. Happy workers are more motivated and are, on average, 35% more productive.

The debate on whether or not data collection is ethical will likely continue for the foreseeable future. One the one hand, firms need to collect a vast amount of data in order to improve productivity within the company to gain better results. On the other hand, the data being collected my breach developer privacy and confidentiality. Developers that cannot trust their management, lose their motivation which decreases their overall productivity and happiness. There will be risks involved whichever route the firm decides to go down.

There are efforts being made to improve the issue with data collection. In 2018, the EU replaced the Data Protection Act with the General Data Protection Regulation (GDPR). This new regulations states that organisations need to have the employees' consent if they want to collect their data. These organisations may be fined heavily if they are found to have breached the elements of GDPR

Conclusion

In this report, I have discussed four aspects of measuring the software engineering process – the measurable data, the platforms and algorithmic approaches available and also the ethical issues surrounding data analysis. There are many different variables that go into the software engineering process. This could be the reason as to why many people seem to think that measuring software engineering is impossible. I have to disagree with this to a certain extent. Measuring the engineering process is possible. There are many tools, platforms and algorithmic approaches available some of which were mentioned in this report. This is all the evidence we need to prove that it is possible. However, I fell that we may be nearing the limit of measuring software engineering. What more is there to track? Sure, we could try to make the current tools more accurate, but there is only diminishing

marginal benefits from trying to improve it any further. We may be at a point where we need to find a new way to make the software engineering process faster. One way could be to have developers specialise into different niches so that the data that is collected will be more accurate and comparable to other developers in the same niche.

References

[1] Techopedia: https://www.techopedia.com/definition/13296/software-engineering

[2] DAXX: https://www.daxx.com/blog/development-trends/number-software-developers-world

[3] Klipfolio: https://www.klipfolio.com/blog/cycle-time-software-development

[4] Pluralsight: https://www.pluralsight.com/blog/tutorials/code-churn

[5] Gitclear: https://www.gitclear.com/

[6] PSP: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5283

[7] PSP: https://www.win.tue.nl/~wstomv/quotes/humphrey-psp.html

[8] Hackystat: https://hackystat.github.io/#:~:text=Hackystat%20is%20an%20open%20source,Researchers.

[9] Timeular: https://timeular.com/

[10] Timelular: https://timeular.com/timeular-for-teams/

[11] Testrail: https://www.gurock.com/testrail/

[12] Halstead: http://sunnyday.mit.edu/16.355/metrics.pdf

[13]Cyclomatic Complexity: https://www.tutorialspoint.com/software_testing_dictionary/cyclomatic_complexity.htm

[14] Cyclomatic Complexity: https://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php