**Measuring Software Engineering**

**Report**

**Anton Tiscovschi 19336441**

**Introduction**

Software engineering is becoming one of the most important disciplines of our time. Industries such as hospitality, communication, entertainment, and transportation are becoming more and more reliant on computing. As software engineering is increasing in importance, a question has been raised: can we measure it?

Can we measure the performance of a team or calculate the efficiency of an employee. Modern business depends heavily on the usage of useful and actionable metrics. As software engineering quickly develops and becomes such an important discipline, the same questions are inevitably asked.

Even software engineers themselves are usually interested in their performance.

However, is this possible? Should we ask these questions? Can they even be answered to a degree where they are useful? Are there ethical consequences of measuring software engineers as individuals and teams? What conclusions can we draw from these measurements and how should we act on them? In this report, we will explore these questions and try to discuss the most important topics of interest.

**Section One: Measure Engineering Activity**

So how does one measure the activity of a software engineer and assess their strengths and weaknesses?

In this section, we will have a look at the most common ways this can be done and then have a look at some larger questions.

**Individual Work**

The first idea that comes to a lot of people's minds when discussing the performance of a software engineer is their productivity.

In a corporate environment or in any environment where a solution to a project is sought to be achieved, one needs results and if the engineer in question is not delivering and working consistently to find a solution there can be problems.

So, we need to figure out how consistently the engineer is providing solutions through their code.

In today's era, this is easily trackable by looking at an engineer's commit history.

We can effectively track how often an engineer produces solutions by just measuring his commits and commit frequency.

We can have a look at how many times they commit weekly, monthly, or even yearly.

Using this, we can take a look at their average weekly commits, measure their frequency and thus measure their productivity.

We now have measured the software engineer and we can differentiate which engineer is working hard and which has been slacking.

Unfortunately, it is not so simple and this is not the case.

This solution has many holes in its logic.

For example, an engineer could be producing more code, more often it does not mean that that code is any good.

They could just be piling onto a mountain of badly spaghetti code and incrementally making the overall solution worse.

The manager who was expecting an expeditious software solution to the current project instead receives incomprehensible

code that barely works and will take another engineer more time to fix than to just start from scratch.

This programmer could have been committing at an incredible rate and appear productive however as we see this is not the case.

So the engineer was productive but they were not performing well and delivered a bad solution.

Knowing this, is there any way we could take into account a software engineer's productivity but also measure if the engineer is good at designing and implementing their solutions?

This is a more difficult question to answer and often does not get the weight it deserves.

To try and answer this question we could begin by diving deeper into the actual commits.

For instance, we can have a look at the number of lines of code that an engineer adds in each commit versus the amounts of lines they delete each commit.

This is useful as it gives us a better look into whether the engineer is trying to arrive at an efficient solution.

If the engineer continuously strives for an efficient solution, they will be deleting a lot of lines as they improve the solution to meet the desired result.

This relationship between lines added and removed is a perfect method of measuring quality but it does give an insight into how the engineer comes to their final result.

**Working in a Team**

Nowadays, a software engineer is much more likely going to work in a team, whether he is working for a corporation or a start-up project. This means it is important for us to measure a software engineer's ability to work in a team.

If we were measuring an engineer solely on their performance against themselves, we can only learn so much, luckily we do have the capability to compare and contrast different engineers and their work.

This becomes especially useful when looking at engineers working on the same project allowing us to make more accurate analyses.

For example, if we were using the method previously discussed, looking at lines added against lines removed per commit and looking at two engineers working on the same part of the project.

Assuming that the functionality remained the same.

If the first engineer creates a solution and then the second engineer reduces the line amount by a significant amount.

You argue that the second engineer came to a more efficient and therefore has improved the solution.

Another way we could measure a software engineer working in a team is by looking at the number of tickets they resolve.

Tickets in this case meaning problems within the larger project

assigned to be resolved by different engineers in a team. Many tickets make up the entire project.

By measuring the number of tickets fixed by an engineer we get a feel of their productivity, similar to checking code frequency.

In this case, we could also rank tickets based on their difficulty and what kind of tickets the engineer solves, and how tough are those tickets.

This is very useful information for a manager because if they see an engineer is easily brushing past trivial tickets, it might be time to give that engineer harder tickets.

The solutions proposed above bring up another common aspect of successful engineers and that is structure.

This is everything from the design and plan of a project to the maintainability and testing.

In this case, we are not directly monitoring the code.

If we could somehow get an idea of how an engineer structures his solutions, we can gain an understanding of their competence.

One thing we can have a look at are the test coverage.

This allows us to see if the project is adequately tested.

Can we then look at the engineer's thought process by finding out when the test was written compared to the actual code?

Were they an afterthought or were they essential to the process of creating the project?

We can also take a look at how the project was built, by seeing if any tickets were written or assigned.

Where and how was the plan of action written, was there even one? This is particularly useful when a team of developers is working on a project, if this is missing it would suggest the structure of the project was weak.

Lastly, the goal is when working on a team is that everyone improves each other.

An engineer must have the ability to improve everyone in his team.

Thankfully we can measure that.

We can have a look at how many comments and deliberations an engineer performs for his group.

We can also take a look at how many comments the engineer obtains.

**Measuring Health**

A factor that is commonly overlooked when talking about software engineering is the health of engineers and the team.

This is a more qualitative metric than the ones we previously discussed as they mostly depend on feedback given by team members and management.

The data is often collected in the form of surveys given at regular intervals throughout the year.

The questions might vary across organizations, but they should likely include questions targetting workload, motivation, levels of stress, quality of communication, and conflicts with management.

In software-centric companies, the software engineering teams and most importantly the software engineers they are made of are one the most important assets. Therefore, it is massively important to make every effort possible to evaluate their working health as much as possible. This makes sure they are working at their full potential and checks for possible points of improvement. Not doing this will often lead to toxic working environments and a decrease in productivity. Leading to the outflow of talent elsewhere.

**Data Computation**

**Introduction**

This section is about various kinds of computation that could be done over software engineering data, to profile the performance of software engineers.

**Counting lines of codes**

It's simple, counting lines of codes that a software the engineer has written to determine their productivity.

This is an old and clear technique, but notorious at the same time.

It has had a long journey through software development history and its advantages when used in the right context and better understood. Measuring the Lines of Code per day metric can give you a view of the size of a software program, how the code base is changing, and potentially the complexity of that codebase.

However, the metric can give incorrect behaviors, expectations, and results, when used in the wrong context.

One benefit of this method is it tells you how large your system is which is great for evaluating complexity and resources which helps you prepare the next developer for working on a codebase.

It can help you predict defect density. LoC can be useful as a value to derive other metrics, such as predicting defect density in your software.

It's appropriate for automation of counting since Lines of Code is a physical entity.

The counting effort can be removed by automating the process.

It is an instinctive metric, LoC is great for measuring the size of software because it can be seen and the effect of it can be visualized.

It's the most used metric in cost estimation, analyses do show a rough correlation between LOC and the overall cost and length of development. You are usually better off with a lower LOC.

It is cheap, LoC is one of the easiest software metrics and is cheap to collect.

However, there are some negatives too counting lines is viewed as a vanity metric since extra lines of code are not a sign of progress.

Vanity metrics are antithetic to correct software engineering, which revolves around reducing complexity and reducing lines of code.

It's an easy metric to the game as the number of Lines of Code per day does not tell you the quality of the product being delivered.

Similarly the number of commits and pull request count is very easy to game.

Just create more lines of code.

Negative behaviors are encouraged even if it's not gamed, a rise in LoC doesn't indicate more productivity, output, or value delivered.

Instead, this causes bloat in the code review process and creates excessive overhead across the team.

It promotes less efficient code since some think it isn't useful to measure the productivity of a project using only results from the coding phase, which usually accounts for only twenty to forty percent of the overall effort.

Code complexity is increased, useful software developers know that good code or refactors are trying to produce less code complexity, not more.

There is no balance between cohesion and complexity. Experiments have shown that effort is highly correlated with LoC, but functionality is much less related.

This means that skilled developers may be able to develop the same functionality with a lot less code. One program with less LoC may exhibit more functionality than another similar program.

It is a poor productivity measure of individuals.

A developer who writes only a few lines may still be more productive than someone writing one hundred lines of code.

It doesn't take developer experience into account.

The implementation of a specific solution can be different based on the experience level of the developer.

Experienced developers may implement certain functionality in fewer lines of code than another developer with less experience.

Doesn't take into account the difference in languages.

It only accounts for the volume of code, you can only use LoC to compare projects that use the same language.

The same code could be written in many different ways, each doing the same thing while taking up more or fewer lines of code.

Doesn't take into account code that is automatically generated by a GUI tool.

With the advent of GUI-based programming languages, programmers can write relatively little code and achieve high levels of functionality.

There is a lack of LoC standards meaning there is no standard definition of what a line of code is.

Questions continue to arise such as "Should we count comments?", "Are data declarations included?", "What happens if a statement extends over several lines?".

It can affect the programmer's mental health.

A programmer whose productivity is being measured in lines of code will have an incentive to expand their code with unneeded complexity.

**Ethical Issues**

In the previous section, we touched on several methods to calculate and analyze software engineering metrics.

In this final section, I would like to start a discussion on the ethical and moral significance of this analysis, in particular the decisions made from these metrics.

The ethical issues arising from software engineering measuring are made even more complex and nuanced given the fact that the measuring usually takes place in a corporate setting, and the people do not have much choice as to what data can be collected and how it will be used.

The first ethical concern is the issue of consent.

Software engineering measurement and research usually happen in a corporate environment, not an academic one.

Employers do not necessarily offer a choice to employees on how their work effectiveness will be measured.

Their work is the company´s property, and therefore it is difficult to limit the scope of what they can collect and analyze.

In Europe, there is the GDPR legislation that may put limits on personal data, but this is not always relevant to the metrics described above.

If it is not possible to get consent from everyone, they should at least be informed of what data is being collected and what is being done with it.

The company should be extra careful if external companies are involved.

If this is the case efforts should be made to anonymize and protect employee data, as a data breach of any sort could cause irremediable damage to the software engineers in question.

Of course, there is an obvious issue when software engineers are made aware of how they are being measured practically every metric can be gamed.

If there are motivations to perform in a specific metric, software engineers will very probably find a way to adjust their work to exceed that metric.

The problem is that this does not always translate to higher-quality work.

If the metric is line count, engineers will write more verbose work, if the metric is a number of commits per issue, engineers will make lots of small commits, if the metric is a number of messages per feature request, engineers will extend their conversations at length on every little issue.

This poses an ethical issue to those being measured do they keep working honestly in the way they understand will produce higher quality output, or do they instead work in a way to optimize the way they are measured.

In the best case, these two aims should merge, but generally, they never do.

Of all the issues present when measuring someone, maybe the most important ones are the issues dealing with how the insights are used.

What actions can derive from that measurement? Is the metric accurate enough to make that decision?

We live in a world of data-driven decision-making.

Using data to make decisions has the aim of being impartial and unbiased.

The promise is that if we base hiring, promotion, or firing decisions on data alone then we do not have to worry about personal biases. However, if decisions are being made on the results of a variety of algorithms then management needs to be very clear on what do results truly mean, and how the algorithms used have processed the data. Not only, but the employees being measured should also be made aware of and understand the reasoning behind decisions being made for them meaning the standards must be public. As mentioned, this can lead to the "gaming" of the metrics.

The clearness of how data is being processed and conclusions being made is not always easy or even possible.

In conclusion, while software engineering measurement can provide very useful and actionable insights, care should always be taken to make sure decision-making bodies understand how these metrics are collected and interpreted.

Software engineers and the team should be made aware of the data that is being collected, how it is being processed, and what decisions are being made using it.

If possible, consent should be requested to provide a trusting and open culture in the workplace.

**Conclusion**

In this report, we discussed four main areas within the idea of measuring software engineering.

We have looked at the ideal and possible implementations of a software engineering measuring tool.

We have looked at platforms that implement such tools and we have looked at how those tools could be implemented.

Finally, we had a look at the ethical and legal issues that arise when discussing such a topic.

Bibliography

A. Sillitti, A. Janes, G. Succi, and T. Vernazza, "Collecting, integrating and analyzing software

metrics and personal software process data," in Proceedings of the 29th Euromicro Conference.

IEEE, 2003, pp. 336– 342