CSU33012 – Software Engineering

Measuring Software Engineering Report

Dominik Guzowski – 19334866

Introduction

In this report I will discuss the processes and methods of measuring, or attempting to measure, software engineering. I will cover some of the naïve approaches that are being used and discuss their effectiveness and what side-effects their implementations introduce. Measuring anything can be extremely useful to help us better understand the thing that is being measured, and in the context of software engineering, such measurements if done right, can provide insightful information to managers and the engineers themselves about the work being done and the effectiveness of that work. Managers may want to know if their employees are working to their fullest potential and engineers may want to track their own performance to better understand how they work, such as their speed, productivity and overall effectiveness.

Information required to measure something as intricate as software engineering is hard to obtain, however there are several services which attempt to collect this data and extrapolate useful information from it, allowing managers to monitor their engineers and their performance.

Measuring human performance in any field can be useful but it brings many ethical concerns along with it. Closely monitoring people is often a breach of privacy and some boundaries may need to be crossed to obtain the necessary information. Intensive monitoring of engineers can be taxing on their mental health and brings into question whether such monitoring is at all ethical.

Methods of Measuring & Measurable Data

There are many ways which attempt to measure software engineering using tangible information produced by the engineer, such as the amount of code they write, the amount of commits they make or their velocity. Most of the methods listed are poor methods of measuring software engineering as each end up creating unwanted behaviours in engineers and in their approaches to development.

One of the most common ways to measure an engineer’s productivity is through the lines of code they write. This method is so common due to the data being so easily available and it being simple to gather and come up with conclusions based on it. This method however is one of the worst possible methods to measure an engineer's productivity due to its flawed nature in that it incentivises writing long, obtuse code. This method rewards the volume of code that is written, most often by the number of lines of code, and it rewards code that is longer but not necessarily better. This method has no means of checking the performance of the code and therefore provides no meaningful information about the engineer apart from the fact that they are capable of writing a lot of code.

```java
boolean isPassingGrade(double grade) {
    return grade >= 40.0;
}
```

```java
boolean isPassingGrade(double grade)
{
     boolean pass;
     if(grade >= 40.0)
     {
          pass = true;
     }
     else
     {
          pass = false;
     }
     return pass;
}
```

Figure 1 showing how the same piece of code can be written in two different ways to artificially increase the number of lines when it is not necessary, a behaviour greatly rewarded by measuring software engineering by the number of lines of code.

This method is one of prime examples of measurement methods which incentivise 'gaming', that is, doing something in a way that will artificially inflate stats through exploiting the short-comings of a measurement method purely to benefit the player, in this instance the engineer, through methods which were not intended in the first place.

Figure 1 shows an example of how using lines of code as a metric to measure software engineering can be exploited by the engineer. The approach on the left is concise and more elegant than the approach on the left which is long and obtuse just to increase the number of lines of code written. Even though the code on the left is by considered by most to be better, the engineer would’ve been unjustly penalised by counting lines of code to determine their performance.

Another naïve method attempting to easily measure software engineering is through the number of commits. While this method is ever-so-slightly better than counting lines of code, it is also greatly flawed. This method incentivises committing changes quickly of potentially untested code and regardless of whether the change was a good one or not, or whether it was a worthwhile change, and if the change was not useful, or even damaging, another commit to fix it would be made, adding more and more commits which were in reality useless. This metric is not only easily exploitable by the engineer, but it also fails to consider many aspects of the commit itself, such as how much code was added or changed, and of course whether the commit itself was meaningful or not.

Both of these methods fail to hit the mark in being good ways to measure software engineering as they are way too easy to game and also aren’t inherently meaningful without further inspection into the code itself.

The third common way to measure software engineering productivity is through velocity, that is the speed at which the project requirements are completed in a single sprint. While this method suffers of gaming as well, it is by far more meaningful than the previous two as with velocity, the output must be truly ready and in a state that could be shipped, therefore things like lines of code or number of commits no longer matter. This method however is not perfect in representing the performance of individual engineers but rather a team, hence not ideal for the purpose of monitoring individuals. Velocity just like the other two metrics can be exploited. If the team is in charge of setting out the requirements themselves, they can split them up into much smaller pieces which technically aren’t full requirements themselves, but this allows them to increase the supposed number of requirements that they have completed and hence inflating the statistics about their productivity.

While velocity can still be gamed, it is still a better metric due to the fact that the output that is being measured is complete, any bugs are potentially fixed and the code is tested enough to be considered safe, therefore at the end of the day, the work that was done is in some capacity useful, which the other two methods do not guarantee.

Good metrics that are used to measure software engineering productivity and efficiency are lead time and mean time to recovery (MTTR). These two metrics show the performance of a software engineering team on a live product and how efficiently they can add new features which are being requested by the customer and how quickly they can fix any bugs or outages of the product.

Lead time is the time taken to deliver a feature requested by the client. This is a good metric since it shows how well the team is performing. MTTR is similar to lead time in that sense, except it deals with the team's speed to fix any problems that could have arisen in the live product, like fixing a critical server fault. A small MTTR indicates a team's capability to quickly resolve such problems. Both lead time and MTTR are better metrics than most of the aforementioned measurements as they measure meaningful data which deals with real-world time, rather than meaningless lines of code.

Platforms Facilitating the Measurement of Software Engineering

There are many platforms that provide services which allow the tracking of developer activities. Some platforms provide automated measurements based on those activities while others allow some to manually inspect the progress and come to their own conclusions based on the data. Some of these platforms include Pluralsight, Swarmia and Jira.

Pluralsight is a platform which provides a service called Flow. Flow extrapolates git data to come up with meaningful and useful conclusions based on a wide range of metrics tracked by git. It allows businesses to see how the development process is coming along by inspecting commit data, pull requests and more. It gives insight into the team workflow patterns to help understand the strengths and weaknesses of a team.

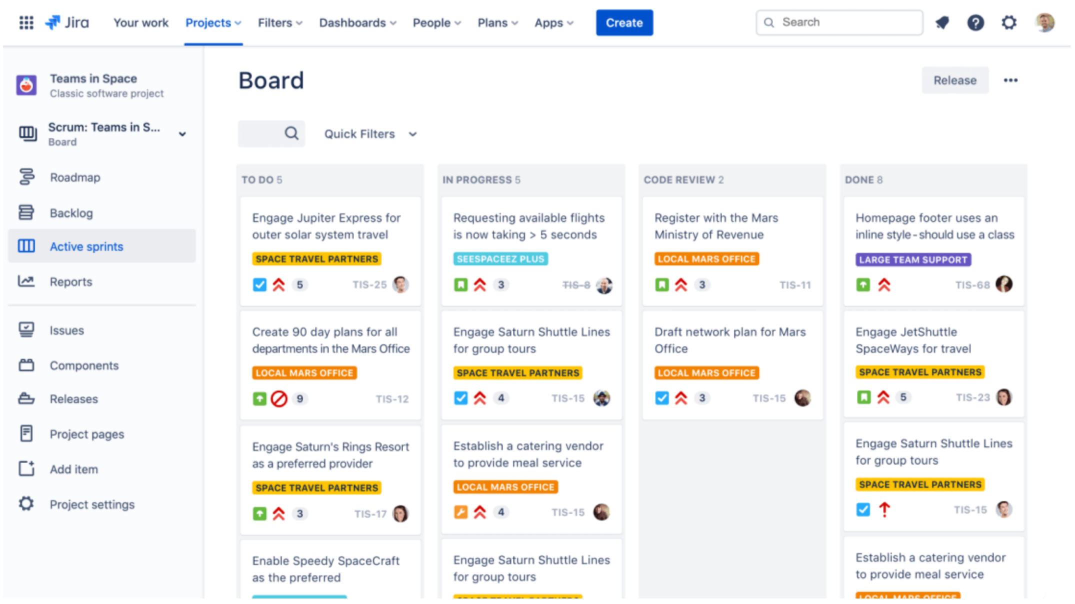
Flow also keeps track of collaborations between team members, showing the team dynamics and who is involved in certain activities.

By tracking certain information about each developer, Flow is able to recognise the developer's skills, their strengths and weaknesses. By collecting this data, Flow is able to give recommendations to teams in terms of their workflow and skill development, allowing senior developers to better mentor junior developers.

Swarmia is platform that allows developers to get a clear insight into their workflow. Swarmia allows developers to see their progress, as well as focus on certain aspects of the work they are doing to get more context about the work. It tracks things like time spent writing code, allowing for accurate representation of developer's workflow and eliminating any guessing, as knowing exact time can help to come up with more meaningful conclusions about the developer's productivity. Swarmia also provides a visual representation of the work that was done by the team in a work log, which allows developer's and managers to fully keep track of what is being done and when.

Swarmia also provides metrics called Insights, which provide crucial information to teams. Swarmia is able to keep track of the scope of activities and help prevent scope creep from happening, which is when the team attempts to tackle a project which at the given moment is too large and must be broken down into much smaller pieces. Swarmia can investigate individual projects and based on those can provide meaningful information to help with project planning in the future. It is also capable of determining bottlenecks and patterns which may be slowing down the team performance.

Jira is a platform that is used mainly in agile development. It is a Kanban style system which allows to split tasks into different stages of development. When a task is finished in one stage it gets moved to the next. Jira collects data based on the usage of the system allowing developers or managers to view this information and come to conclusions.



Board

TO DO 5
- Engage Jupiter Express for outer solar system travel — SPACE TRAVEL PARTNERS — 5 — TIS-25
- Create 90 day plans for all departments in the Mars Office — LOCAL MARS OFFICE — 9 — TIS-12
- Engage Saturn's Rings Resort as a preferred provider — SPACE TRAVEL PARTNERS — 3 — TIS-17
- Enable Speedy SpaceCraft as the preferred

IN PROGRESS 5
- Requesting available flights is now taking > 5 seconds — SEESPACEEZ PLUS — 3 — ~~TIS-8~~
- Engage Saturn Shuttle Lines for group tours — SPACE TRAVEL PARTNERS — 4 — TIS-15
- Establish a catering vendor to provide meal service — LOCAL MARS OFFICE — 4 — TIS-15
- Engage Saturn Shuttle Lines for group tours

CODE REVIEW 2
- Register with the Mars Ministry of Revenue — LOCAL MARS OFFICE — 3 — TIS-11
- Draft network plan for Mars Office — LOCAL MARS OFFICE — 3 — TIS-15

DONE 8
- Homepage footer uses an inline style - should use a class — LARGE TEAM SUPPORT — TIS-68
- Engage JetShuttle SpaceWays for travel — SPACE TRAVEL PARTNERS — 5 — TIS-23
- Engage Saturn Shuttle Lines for group tours — SPACE TRAVEL PARTNERS — TIS-15
- Establish a catering vendor to provide meal service

Figure 2: Jira Kanban-style board with 4 stages of development. Credit: atlassian.com/software/jira

Tasks can be split into sections such as 'To do', 'In progress', 'In testing', 'Complete', providing a structured roadmap for the development of the product. Jira can keep data based on each stage of development and combined with other metrics such as lead time or velocity, the information it provides can show how the team is performing in great detail at every stage of development. Since Jira does not inspect code like Swarmia or Pluralsight, it is less intrusive and hence may be better for the mental health of engineers, knowing that their every action isn't measured. However, the manual aspect of Jira might be unappealing as it will take time to analyse the data that Jira collected, which businesses may not be willing to do as readily, especially if there exist systems which can automate this process, such as the aforementioned 2 platforms.

Algorithmic Approaches

There are many algorithmic approaches that have been developed over the years to attempt measuring software engineering activity accurately. These approaches range from simple solutions such as simply looking at the pure volume of data, in a 'the more the better' sense, such as counting lines of code, commits and number of pull requests, or more manual approaches which have people determine the complexity of a given task and how effectively it was completed. Then there are algorithms which take a more in-depth approach and perform some sort of computation given certain sets of data in attempt to extrapolate some useful information. Some approaches use statistical analysis in attempt to understand the data, and the impact an engineer has in their team. Finally, there are approaches which utilise artificial intelligence and machine learning. These methods attempt to understand the engineer and how they work, their habits, skills, strengths and weaknesses, as well as trying to determine the complexity of code written and how well it was written.

The most common ‘algorithms’ that are used are essentially brute-force counting of things. These are extremely cheap and require no work but also produce data which is not super useful and can easily be gamed. These algorithms tend to reward volume of work, rather than its quality, which may give a false sense of productivity for a given engineer. Oftentimes inexperienced programmers write obtuse, inefficient code, and counting algorithms would see that as better than elegant, short and efficient code that is written by an experience engineer. Due to the fact that these algorithms are contextless, they often miss the mark completely in trying to measure software engineering productivity.

Manual approaches often have people decide what is considered productive. Tasks that are deemed to be difficult are awarded more in terms of team productivity, and it is often expected that difficult tasks take longer to complete. When such a task is completed in a shorter time than initial expected it can lead to false outcomes as sometimes tasks are perceived to be more difficult than they truly are, which would overly credit tasks which only seemed to be difficult. This method is quite inaccurate on the surface and care must taken in order for this approach to have any meaningful value, such as looking retrospectively at the task that was done and only then deciding whether it was done efficiently or not. However, this approach also has its weaknesses due to human bias, where if someone working on the team was the person evaluating this performance, they might subconsciously award their team more credit than the task was truly worth. Due to these shortcomings, I believe that manual evaluation of perceived efficiency and productivity are poor approaches to measure engineering activity, as they can lead to inaccurate results, and standards can differ greatly from team to team.

Statistical methods are also used to determine the impact of engineers on their teams, and also to attempt to predict productivity in the future based on past projects. Analysing things like programming languages and can be used to determine what should be used for a given project to optimise performance based on runtime. Things like Bayes' Theorem can be used to predict things like delivery time and delays based on past events, taking into accounts things like days off due to illness or other events which may contribute to the project not being delivered on time. This probability can be used then to better estimate how long things like sprints will take, allowing for more accurate estimates and less delays.

Machine learning has been growing as a tool to measure software engineering. Machine learning is based on complex algorithms which learn and adapt based on what the machine learns from the given data, which makes them ideal for measure the engineering process as it allows the machine to find patterns in behaviour and habits. The use of machine learning in measuring software engineering is in my opinion ideal for the fact that there is so much data produced by developers all over the world, which is readily available through systems like GitHub. This data can be used to train the machine learning model to improve its ability to analyse and predict software engineer activity, productivity and more.

Ethics of Measuring Software Engineering

Collecting any data from people comes with ethical concerns and software engineering is no exception. The information required to monitor and measure the performance of a software engineer can be incredibly specific and can oftentimes raise personal privacy concerns.

Certain companies have begun to take monitoring their engineers to the extreme, by measuring things such as their posture, heart rate and even mood. This extent of monitoring is far from ethical as it doesn't respect the privacy of the engineers. On top of that, these engineers often do not have a say in whether they agree to this monitoring or not.

On top of monitoring their physical and psychological behaviours, software engineers also being monitored through screen invigilation, such as constant screen sharing or through screenshots which occur at certain intervals. In my opinion such monitoring is not only unethical, but it also shows lack of trust towards the developers, as well as the fact the some of this information seems excessive and should not be measured in the first place.

In my opinion, such constant monitoring of developers must have adverse mental health effects due to the constant feeling of your every move being watched by someone, potentially leading to feelings of paranoia. Another effect of constantly being monitored is the Hawthorne effect, which is the way in which people adjust their behaviour due to the fact that they are being observed, which makes the information acquired from the monitoring not fully authentic, and in turn being rather counterproductive.

All in all, I believe that the monitoring and measuring of software engineers is unethical and shouldn't be done to an extent that potentially makes the developers feel uneasy or such that they feel a lack of privacy at work. However, that does not mean I think all measurements and monitoring is bad. Monitoring things like posture and heart rate can be more positive depending on the context, in that is someone has metrics indicating something negative, this information should be used to help the person in question to help them improve, rather than using this information against them and potentially firing them for lack of productivity. Also monitoring actual software engineering, is good in that it helps individual developers know how they are getting on and in which areas they could potentially improve, however this information should also not only be used as a basis to dismiss an engineer, but first of all should be an indicator that someone requires guidance.

Conclusion

Measuring something as intricate as software engineering is not an easy task and there is no perfect solution. Measuring software engineering can potentially give useful insights into the productivity of engineers but there are limits to everything. Some approaches are easy and cheap but not very useful, while other approaches are expensive but do provide some insightful information, however they often cross the line in terms of privacy. Measuring engineers' performance can be used to help the said engineers improve and become better at their work, and this should be the main cause for monitoring their work. In the end it is the developers who write the code and they often know what and how things should be done. Measuring software engineering productivity and performance should not be a game in which the main goal is to have the engineer ultimately lose for small mistakes.

References

- Wikipedia Contributors (2019). *Hawthorne effect*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Hawthorne_effect.

- Atlassian (2019). *Jira Cloud*. [online] Atlassian. Available at: https://www.atlassian.com/software/jira.

- Wikipedia Contributors (2019). *Machine learning*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Machine_learning.

- www.pluralsight.com. (n.d.). *Pluralsight Flow | Overview*. [online] Available at: https://www.pluralsight.com/product/flow [Accessed 28 Dec. 2021].

- Wikipedia Contributors (2019). *Bayes' theorem*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Bayes%27theorem [Accessed 29 Dec. 2021].

- www.swarmia.com. (n.d.). *Gain visibility. Remove blockers. Ship 10x faster. | Swarmia*. [online] Available at: https://www.swarmia.com/ [Accessed 28 Jan. 2021].