CS3012 Software Engineering Report

Measuring Software Engineering.

Author: Vahe Sasunts

Student ID: 16316570.

Introduction.

This report aims to provide an insight as to how the software engineering process can be measured, what kind of data we will consider analysing and what are the best tools out there to collect and analyse this data. I will also talk about the reasoning behind measuring the software engineering process and the ethics involved. To date there has been many attempts at measuring software engineering, however there is no formal universal approach to it.

The measurable data can be many things and even the relation between the different data collected. This may include the number of lines coded, the number of commits to a repo or even the time between each commit.

The tools used to measure this data can come from third party vendors or can be a programme developed by yourself. It's all based upon what you want to measure and why you consider measuring this specific data.

As always when we start collecting data there will always be an issue of ethics. I will also talk about this in the report. With recent laws passed in the EU (notably GDPR[i]) collecting data has become much more difficult and requires consent from those you collect and store data from. This can also land you in trouble if you use data without permission or secure it properly.

Why measure software engineering?

There are many reasons to measure software engineering. If you are an employer, it would be in your best interest to know that the person you have hire is up to speed with your company and is able to deliver on the project. Companies need to be able to monitor the performance of their employees to ensure they meet deadlines, this key to reducing costs for the company. It is a great way to diagnose a problem and fix it sooner rather than later and may even lead to innovation as the software engineer may be inspired to produce better results since they are being assessed.

Table of Contents

What is software engineering?

Definitions

I guess it would be wise to start defining what software engineering is and what a software engineer does throughout his/her day. If we look at the IEEE (the Institute of Electrical and Electronics Engineers) and ask them to define software engineering we get this: "the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software"[ii]. This may be a bit difficult to follow but it simply means that we apply the same methodology in engineering and apply it to software-based solutions. We analyse the users' needs and design, construct and test possible solutions that may satisfy those pain points that the end user may have. These solutions come from developing complex software and not by means of simple programs.

A software engineer on the other hand is a person, usually part of a team who collaborates to build the required software for the end user. This means that these software engineer will be the main persons involved in creating the product, testing it and designing the underlying architecture. Software engineers usually come from a computer science background, but are not strictly limited to do so, there has been many cases where software engineers have self-thought themselves the required skills to become one.

What makes computer science different to software engineering?

While computer science is the name given to anything related in the field of computer theory, software engineering looks at the techniques used to create functional and quality code that can be used in industry or for end users. Computer science is something that is taught in schools and universities, and software engineering is really the application of the theory that you learn. Knowing the technology is one thing but being able to apply it in the real world is another. That's where the two distinguish themselves.

How do we know that they are good?

Now that we understand what software engineering is and what a software engineer is, there is one question left; "how do we know if the software engineer is good?". In this report I will talk about how we can measure the level of competency and provide a few possible ways to do so.

Data

Data types

A software engineer must produce data. To provide solutions to a customer they type code, sketch prototypes, run tests etc. This is where most of the data can be collected from. When a software engineer types code, we can get data like the number of lines they typed or the number of characters they typed etc. This code then must be stored somewhere. In a software engineering environment this is usually done on either a local or online git repository. Here we can see the revisions made to the code, the difference in time since the code was last changed, how many times there were changes made and how severe were the changes.

For this report we will look at a few metrics to analyse and see how they can be good or limiting:

- Number of lines in the code
- Number of commits to the repo
- Test coverage
- Time stamps of commits
- Number of contributors to project

Number of lines in the code

This type of measurable data is slightly misleading as measuring the number of lines of code is not a good way to access someone's ability to code. "Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs."- Bill Gates[iii], this quote by Bill Gates really puts it into perspective how awful it is to measure software engineering based on lines of code but in some cases, this may be applicable. It is usually the case that fewer lines of code means that the code is cleaner and better but so long as it achieves the same function as a longer code segment does. 1 line of code is by no means better than 50 lines of code but if you could do the function of the 50 lines of code in 30 or even 20 then yes measuring lines of code can show great competency in the software engineer. If you measure that a software engineer has only coded 30 lines compared to another who has coded 70 this does not mean that the first engineer is coding more efficiently or that the second engineer is coding on something more complex. This is quite a limiting way of measuring productivity so as a rule of thumb it is wise to not invest in a lot of time measuring lines of code.

Number of commits to the repo

Git commits are a good way of tracking progress. You can clearly see the state of a project from the beginning to its current position and the number of commits can reflect the time spent on the project. One thing to look out for is the relevancy of the commit and what does it add to the project. Someone with fewer commits may be working on a more complex part of a solution while someone else with more commits could only be working on easier parts of the project. Gitlab has stated before that number of commits does not show the engineers

competency[iv]. Then why is it relevant? Well for many reasons, here we can see the number of times the engineer likes to spend time changing, mending and fixing their code. It shows a desire to progress. An engineer with 1 commit and a fully finished project while it does sound good may not be ideal as this means they like to work alone and we can't gauge their progress, however an engineer with 100 commits and a finished project does allow us to see where they started and how far they have come. It shows that they are not embarrassed by their code and like to share it with the world or their team to get feedback and fix accordingly. This is a considered to be a strength of a software engineer and for this point number of commits to a repo does allow us to access this trait.

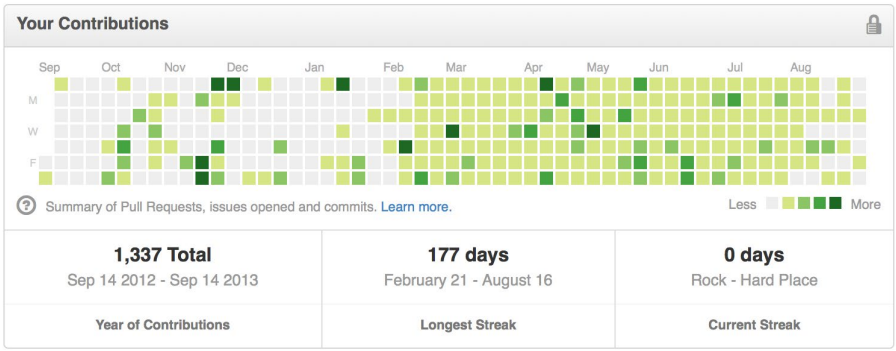

Figure 1: Github commit history

Figure 1 above shows us how Github is able to collect the number of commits a user makes and display it visually, where the dark green represents more commits and the light green represents fewer commits on that particular day.

Test Coverage

A good software engineer should always test their code to ensure that what they write; works, is efficient and has no redundant code. A test coverage should be able to have many test cases where each line of code written has been run at least once. Figure 2 shows us clearly that only the code highlighted in green is run and those segments in red have not been touched by our test cases. It may even be the case that it will never be run and here we can see a software engineer who wrote lines of code without thinking the implications of the code and why it would be run or not run. According to Sealight [v] testing code coverage has many benefits and these include:

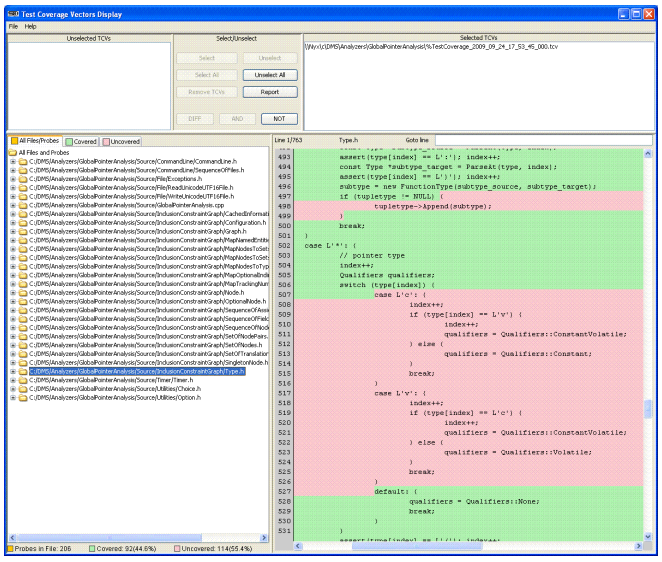

Figure 2: Code coverage

- Software with high results mean that there are fewer bugs in the code.
- High percentage may imply that the code is more maintainable and readable.
- It provides a measurable value to stakeholders on software quality.
- Levels between 70 and 90 percent suggest reliable software, according to a review of academic studies[vi] that examined the correlation between software quality and code coverage.

It is clear that code coverage is important, there are many frameworks used to test this and ultimately any software engineer who looks to improve their code should always test for coverage to ensure that what they are writing is at least being executed by the program.

Time stamps of commits

As I had mentioned before, commits allow us to see the progression of a project. But what makes commits even better is that it allows us to see the exact time when each commit has been made to a repo. This means that we can see the behaviour of the software engineer. One engineer may have a block of commits all 10 minutes apart for the space of 2 hours and another might have a block of commits all an hour apart in the space of 5 hours. This is a good way to understand the nature of the software engineer, whether they like to work fast or if they like to take breaks and write more lines of code before they commit to a repo. One advantage of committing so often is that it's easy to understand where the code is going wrong and revert to a point where nothing is breaking. However, the issue arises when there are too many commits and it's hard to navigate through the history of the code. A reason why committing less but with more code is good is that it's easier to navigate back through the history of the code but again more difficult to understand what is going wrong in the code if there are bugs.

Number of contributors to the project

The last data type that I would like to consider in this report is the number of contributors in the project. To measure software engineering success, it is accepted that success comes in a way of meeting the deadline and satisfying the project brief. Some projects might not meet their desired deadlines while others reach it sooner than expected. Here is where the number of contributors can really make it clear as to why this may be the case. When a project meets its deadline or even finishes sooner than expected it would be wise to see the number of contributors, a well-managed high-volume team, could be capable of delivering results fast, while a small team might still be able to achieve the same results in a longer period of time. Measuring the competency of software engineers this way may be misleading as hitting targets might not always mean that it's the best software engineers and vice a versa. This is again a topic that can be used to aid our verdicts rather than be a definitive answer.

Data understanding

Now that we have talked about the different data types. It would be wise to briefly talk about how we should interpret these data types. By no means can we measure the success or failure of software engineering by any one particular data type. Every project will have its own level of complexity and deadlines which can be reasons to make software engineers in some cases work harder or other cases fail. A good way to go about understanding this data is to use it all to come to a verdict that reflects the true state of software engineering level and competency. Everything we look at has to be checked within context and cannot be the same for every other project. This way we can compare the difference between project and find similar projects to compare with and understand what is regarded as better or worse.

Measuring the data

Platforms

There are many platforms available to measure the data we have talked about above. Some of these are free and some of these come at a premium usually in the form of a subscription service. Also, some are built into the tools developers use while they work on a project such as the Github commit history visual diagram.

Free tools include:

- Github API / Gitlab API / Bitbucket API etc.
- Code beat
- Unit testing frameworks
- Code climate quality
- github.com/topics/data-visualization (list of project to visualise data)

Premium tools include

- Code climate velocity
- Codacy
- Hubstaff

As you can see there are many platforms available to measure the data with some of the premium ones like Hubstaff allowing to measure productivity of the software engineers.

Approach

With the mentioned platforms above there are a few different approaches to measure the data and access the competency of software engineering. The premium route is a costly yet proven method, this means that the software used will be accessed and the data will be recorded automatically. An algorithm would filter the data and provide you with the results you are looking for and then you can make a final assessment yourself. This is the easy route.

With the free route there are to sub routes you can take. Some of the tools such as code beat or codacy offer free tools to start-ups or open source projects. If this fits your description, then perfect you can go ahead and use those tools for free and get the required data.

The other route would be to build the assessment tool yourself and tailor it to your exact needs.

The Git API's are great to pull relevant information straight from the repo. Once you have pulled the required data then you can start creating the algorithm which will take this data store it in a data structure and display it in an easy to read manner.

Algorithmic approach

Today we hear a lot of buzz words such as artificial intelligence, computational Intelligence, machine learning, deep learning etc. These words are applied to almost everything now and at its core is the ability to learn the data in its input and understand what success is and what failure is. These algorithms can be trained and fined tuned to understand the data and given enough examples of good software engineering and bad software engineering it can be made to automatically detect faults, or clean precise code.

Neural networks

The human brain is used as a source of inspiration behind the underlying concepts of neural networks. They are massively parallel distributed networks that have the ability to generalize and learn from examples[vii]. If this is implemented correctly then of course you can save time effort and money when it comes to analysing and reviewing data.

This is one of the many reasons why there is so much money invested in these fields of AI data analysis.

Fuzzy Systems

Once again, we look to humans as a source of inspiration, this time its human language that is the underlying concepts behind fuzzy systems. This involves clustering data based on approximation. The reasoning behind this is that some data may be incomplete and therefore needs to be extrapolated to provide us with a valid measurement.

Data visualisation

Perhaps this is the most important part of the process. There are numerous ways of visualising data, it can be in the form of graphs, charts, network maps etc. It all comes down to how you want to view the data and label the findings. I will now talk about the different types and their benefits.

Bar Charts

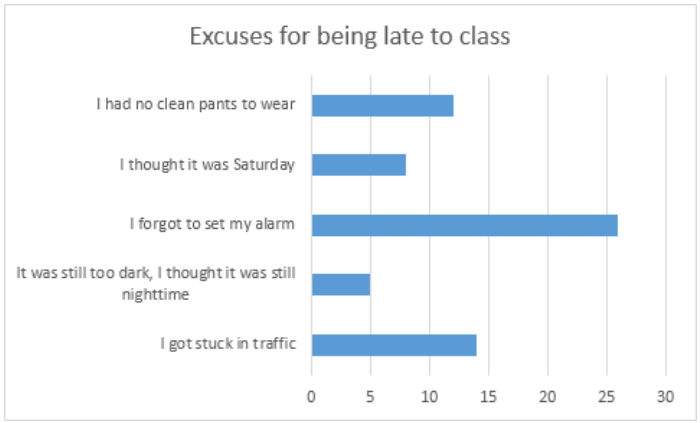

Figure 3: Bar chart for excuses of being late into class

Figure 3 shows us how a bar chart is constructed. This one in particular is a quantitative way of measuring who came up with what excuses for being late to class. The humour aside this method of collecting data is quite popular and tells us a few things. In the context of software development, it could be the number of commits each contributor on a project commits for each day of the week. This means that we can find specific days where the team really like to perform (such as the start of the week) and days where there is little activity (end of the week).

Scatter graphs

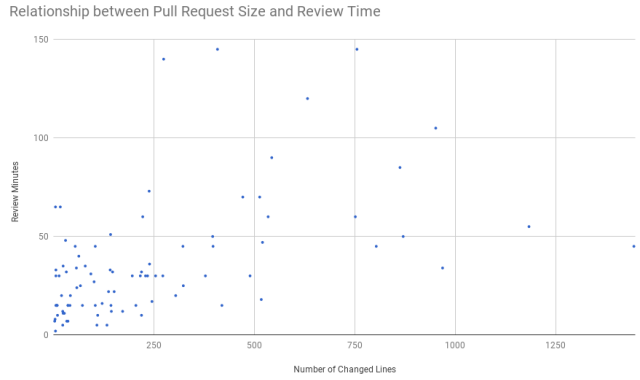

Figure 4: Scatter graph on the relationship between pull request size and review time.

Figure 4 is an example of a scatter graph. In this case we see the relationship between the review time of a pull request and the size of the changes made to the source code. From the Graph we can clearly see that the less changes made the code the quicker it is to review it and merge with the source code. This isn't always the case but as per usual there are anomalies. Also, from the graph we can see that the larger the changes, the longer it takes to review the code and merge with the source code. This is a perfect example to measure software engineering as you could conclude that missing deadlines might come from reviewing large commit changes therefore you could tell the team to commit more frequently with fewer changes to allow the review team to access the code and merge with the source.

Abstract visualisation

The above methods of data visualisation are quite standard and have been used for centuries. With today's technology we can showcase data in so many ways.

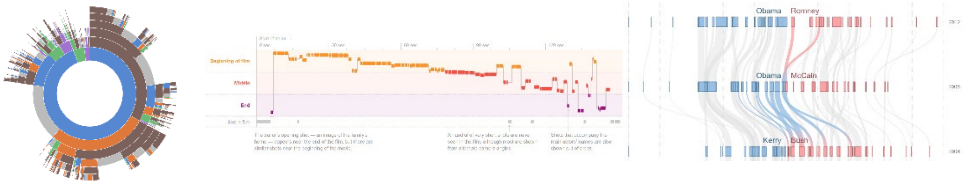

Figure 5: Examples of abstract visualisation by data-driven documents

Figure 5 shows us examples of different types of abstract visualisation [viii] which upon examining a specific point in our data we are given even more in detail description. This is possible thanks to digital technology and the ability to have a screen which can adjust what it displays on the fly unlike paper. This method of data visualisation has become quite popular as it allows users and data analysers to pack much more detail in a smaller footprint.

Ethics

Moral ethics

When it comes to collecting data from people, one issue that always arises is the ethics behind the collection, storing and distribution of data. The issue arises when people whose data is being collected don't even know that this is the case and often times this data is then sold to third parties who may use it to advance their business such as direct advertisement to individuals.

This is something that may not be considered morally ethical as selling other people's information can be considered illegal[ix] depending on the contexts. This does not mean that it is outright illegal to sell this data but there are strict laws governing the selling of data. In majority cases this is done through consent and also in recent times with GDPR in place this consent has a timeframe where after a period of time the consent has to be reviewed and if the person says no then the data has to be destroyed.

Privacy protection

Data collected can be quite sensitive especially those collected by hospitals or banks. This is why we have privacy protection laws where this information that is collected has to be stored in a secure manner, ensuring that access to it is restricted. People have the right for their data to be private and therefore those organisations who collect the data have a responsibility to ensure maximum security. This is done by means of password protection, encryption etc.

Ethics applied to software engineer data analysis

Employers who hire software engineers do have the right to ensure those who they pay are working to the best of their ability. The data collected by them must have the consent of the software engineer and should any of the results suggest poor performance it would be wise to ensure that the employer warns the employee of their finding and provides support to get the employee up to scratch.

Should an employer avoid providing support and fire an engineer based on their finding it may be seen as defamation of the employee. Since in many software engineering roles there are pre-screenings and coding challenges involved before they are hired, an impulse termination of the contract may not be beneficial to the company. After all this required a lot of HR hours and investment.

To be fair to your employees you must define key performance indicators (KPIs), let them know that you will be monitoring their performance and would access their work routinely. The transparency between what you as an employer do towards this data has to be clear since any miscommunication can result to lawsuits which in turn results to loss of profits.

Conclusion

Verdict

The final point I would like to make in this report is that there is no perfect way of collecting and analysing data. With this in mind there is no perfect way of visualising data. This will always be subjective based on the nature and culture of the organisation. Analysing and measuring software engineering must be mutually beneficial between the organisation and its employees.

Should an organisation feel oppressive it may result in poorer performance by their employees but also the lack of monitoring may also lead to this.

One approach in my opinion is one that empathises with both parties and provides enough information to the organisation without being obtrusive to the software engineer.

For the organisation this may lead to more efficient and clean solutions and for software engineers it means personal or team growth and development.

Best case is a third-party arbitrator who may solve any potential conflicts between the organisation and its employees.

References

[i] Official Journal Of The European Union, L119 Volume 59
[ii] Systems and software engineering - Vocabulary, ISO/IEC/IEEEstd 24765:2010(E), 2010.
[iii] Lines Of Code
[iv] Gitlab article
[v] Benefits of measuring code coverage by Sealight.
[vi] Exploring Code Coverage in Software Testing and its Correlation with Software Quality.
[vii] What is computational intelligence
[viii] https://d3js.org/
[ix] Warning over illegal trade in people's information.