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

Measuring Software Engineering

John Cosgrove

Student Number: 19336027

Section 1: Methods for measuring engineering activity

There are many ways to measure software engineering activity with a range of arguments for and against each. We can split these methods into two classes.

There are in-process metrics, which measures aspects of the development process. An example of this would be measuring the number of commits made by each developer on a project and number of lines added in each commit.

We also have code-orientated metrics. These analyse the code content, the most obvious example would be measuring the complexity of the code.

Many methods can be short-sighted, overlooking aspects of the software engineer's profession. An example of this would be measuring the number of lines added per individual. This misses developers who are going back to improve existing code and rewards those who are adding inefficient code. Another example would be counting the number of commits from each person, this would encourage developers to ignore larger development tasks and instead to focus on smaller ones, driving up the number of commits that they are making.

The requirements to make a metric useful are:

- Business value: Can we connect the metric to the business value it creates?
- Measurable: To measure software engineering activity, our method must produce a quantifiable value.
- Actionable: Can it regularly inform action which leads to a beneficial outcome?
- Convenient: Can a regular software engineering team implement the metric?
- Empirically proven: Is there enough evidence to prove that it works? This evidence could be documentation, testing and/or established usage.

A measure which fulfils these requirements could still be unsuitable if it can be easily gamed by developers but these requirements provide a framework to move us in the right direction.

Below I will be outlining some metrics which are used for measuring engineering activity.

Defect trends are a smart way to measure software quality by tracking the number of bugs escaping into production. You can then also track how long these bugs are taking to get fixed. This data becomes even more useful when combined with other sources such as the contents of commits being made and the developer making them. This could allow a team leader to see who is responsible for introducing buggy code and to see who is fixing it.

Deploy cycle time is a method used by teams which want to create an incentive to deploy their code frequently. It is essentially a measure of the time it takes between a commit to the main branch and deployment. This encourages teams to streamline their testing and deployment preparation processes.

Commit counts are another way to track engineering activity. It can be useful after a project to get an idea of how everyone performed. The results can be wildly inaccurate as a commit containing a one-line change and a huge code addition are treated the same. The data can also become extremely skewed if developers know they are being monitored this way, as they are inclined to make bogus commits. This is why it is only sensible to use this measurement in specific situations and with due caution.

Test coverage percentage is a commonly used method for measuring the health of the code. It is essentially the percentage of the code that is covered by unit testing. We can then track developer commits to ensure that they are adding adequate testing to their code. It is important to understand that this is not the be-all and end-all of testing and is only effective when used alongside integration and functional testing. It is also useful as it helps developers find dead code (code which is never run).

Code Churn is another measurement I feel it is relevant to mention. It tracks when the developer re-writes their own code in an interval after it has been checked in, typically three weeks. It typically measures lines of code that were modified, added, or deleted. Some platforms insist that it is useful for measuring engineering performance while others disapprove of it. I agree with the latter, its biggest issue is how difficult it is to understand why code churn could be spiking. It can be an indicator that a developer is experiencing difficulty solving a problem or has been given too vague of requirements.

In contrast, it can also point to a developer polishing their code or prototyping a feature.

Line counting is another poorly performing measurement which is occasionally used. It is regarded by most developers as one of the worst ways to measure effort in engineering activity. Its most simple issue is that the number of lines says very little about the code. Is it longer because it is well commented or it inefficient? Is it shorter because it is efficient or just contributes little? It is also easy to cheat, a developer can alter his code in a plethora of ways to make the line count in a commit shorter or longer. David Bayles said about this problem "You don't pay Michelangelo to make brush strokes – you pay him to be a genius".

An interesting alternative to line counting is line impact, this aims to achieve the same goals as line counting but while cutting out noise. I will be touching on this in section 3.

Section 2: Platforms

Organisations have many platforms available for processing the data from their repositories. Services such as AWS and Azure offer all of the computing power needed for large scale data analysis, at a reasonable price. Large companies like Google or Amazon would consider processing their own engineering data as they would have the resources to allocate a full team to it. Another organisation which would need to take this approach could be an organisation that processes health or government data and cannot give a cloud service access to their repository's.

The previous approach has become less popular now that there are many out of the box SaaS solutions for measuring software engineering activity. Most companies would view developing their own metrics as "reinventing the wheel".

There are platforms available for analysing software engineering activity and many contenders have entered this space in the last four years. The main ones are:

- GitClear
- Pluralsight Flow

- Waydev
- LinearB
- Pinpoint
- Code Climate

These platforms are built around three main data sources. These sources are lines of code, issues being linked to pull requests and commits. The data is pulled from the version control system being used such as GitHub, Bitbucket, SVN.

The selling point used by these platforms is that the average software development department of 50 people will cost well over a million per year to the company, why not spend a few thousand to ensure that the company is utilising them as best as they can?

A lot of these platforms are similar but they compete on price and metrics. Every platform has metrics not offered by their competitors and which they say are the best for analysing software development. Below I will cover some methods used by platforms to stand above the rest:

GitClear, which launched in 2019, boasts a new metric called Line Impact which the rest of their metrics are built off. It is marketed as a version of line counting that works. Most Git tools just show changes as green and red. Green meaning adding new code and red meaning that code that has been removed. The problem is that this disregards code been copied between files or swapped around. GitClear's parsing engine understands code changes as adding, deleting, updating, moving inside the file, and pasting from somewhere else. It awards points depending on the action. It awards significant points for deleting lines as this reduces tech debt. It doesn't give any points for moving lines as there is no work done here. This feature also has other behaviours making it quite complicated. This metric is designed to encourage developer to refactor legacy code and work efficient. I believe this is a cool idea and definitely one I'd consider if I was running a development team.

Pluralsight Flow is similar to GitClear but much more expensive. They only launched their software engineering analysis solution in September but have been much more successful as their online skills and training service was already well established. They are already used by 70% of Fortune 500 companies, but are not clear about what percentage are using their Flow service.

WayDev offers more resource planning and budgeting features than its competitors. You can input how much each engineer costs per hour etc and WayDev can help forecast the total cost of a product. It also helps estimate the cost and impact of bug fixes and issues on project delivery. This could be useful for a company trying to track its development spending.

Code Climate take an interesting approach also. They have split this problem into two separate solutions, one focusing on velocity and the other focusing on quality. The velocity solution is fairly similar to the offerings from Pluralsight and WayDev, it is priced similarly. The other solution is more interesting, it is cheaper and even free for Open-Source projects. It is designed to partly automate the code review stage, producing insights on test coverage, maintainability and can spot areas with high code churn. It seems to offer more code quality analysing features than any other platform. Because it is free for small or open-source projects, I am considering using it next semester for the software engineering project.

Section 3: Types of computation

In this section I will be covering various methods which can be used to analyse large sets of Software Engineering data. I will be starting with measures pioneered in the late 1970s and finishing with the techniques which haven't been implemented yet.

In 1977, Maurice Howard Halstead introduced Software Metrics called Halstead Complexity Measures. His goal was to make the measurement of software development an empirical science instead of a theoretical one. He wanted to make his metrics language agnostic and independent of platform. They are therefore computed statically from the source code.

It is built from four main data points:

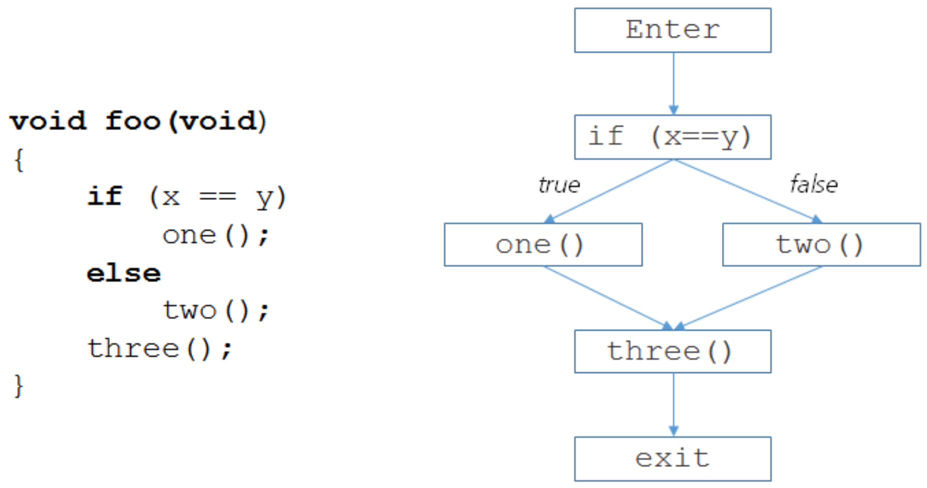
- n_1 = number of distinct operators.
- n_2 = number of distinct operands.
- N_1 = The total number of operators.
- N_2 = The total number of operands.

From these measures we can calculate program vocabulary, program length, calculated estimated program length, volume, difficulty, and effort.

It is difficult to find usage of these metrics today as they correlate strongly with SLOC(Source Lines Of Code), making them redundant.

A year before Halstead released his metrics, Thomas McCabe created the Cyclomatic complexity metric. It still can be useful today. It is a quantitative measurement of the complexity of a program, it does this by counting the number of linearly independent paths through a program’s source code.

This is done by modelling groups of commands in the program as nodes. A directed edge connects nodes which are executed one after another. We can apply this to entire programs or individual functions, modules, methods or classes.



```
void foo(void)
{
      if (x == y)
            one();
      else
            two();
      three();
}
```

Nowadays this metric has two main usages:

- It is commonly used during the testing phase as the Cyclomatic complexity is equal to the minimum number of test cases needed to achieve complete code coverage. This is useful for estimating the total number of test cases needed and for indicating if a section of code could be more difficult to test fully.
- It is used to indicate if code could be too complicated. Overcomplexity can impact readability, maintainability, and reliability, so programmers are keen to keep the cyclomatic complexity low. This can be done by reducing duplicate code, using smaller functions etc.

Function Point analysis is a metric used to express the amount of functionality of a given software. The functional user requirements are split into five types:

- Outputs
- Inquiries
- Inputs
- Internal Files
- External Interfaces

Once a function is given a type, it is assessed for complexity and assigned a number of function points. This metric is not very empirical, although there have been efforts to improve this by modifying the method to account for algorithmic complexity.

Function Point analysis is still in use but criticised by many. Like the Halstead Complexity Measures, it has a strong correlation with SLOC, impacting its perceived usefulness.

The three methods covered above are quite limited, Cyclomatic Complexity being by far the most useful. This is why platforms such as GitClear use a wide range of different metrics instead of attempting to use a single one, using proprietary algorithms to combine these measures together.

Computational intelligence(CI) is also a direction this science can take in the future, although there is little evidence of usage at the moment. CI is an offshoot of artificial intelligence, consisting of three core technologies. These are, Fuzzy Logic, Artificial Neural Networks and Evolutionary computing. The one most relevant to this paper is Artificial Neural Networks. This technology involves manipulating hard datasets and adjusting your neural network to learn the right way to approach a problem. I believe through this it would be possible to teach a neural network to recognise good code, a good developer, or an underperforming project. There are two huge obstacles in our way here:

- We would need a vast collection of data to train a neural network like this. There is a range of ways any one programming problem can be approached, and neural training would need to reflect this.

- Because we want our network to perform better than the metrics we have already, it will need to be trained by humans. This is an issue as it will take thousands of hours from competent developers to train it to a worthwhile standard .

In conclusion, although we could have better options in future, our best option currently is to pick a range of metrics to solve our problem. Then we can use a measurement platform to combine these using an algorithm to produce useful data for us to work from.

Section 4: Ethics

The ethics of measuring software engineering are complex. I am going to focus on the guidelines and ethics in the EU as the rules and ethics elsewhere are inconsistent. Adhering to GDPR guidelines does not guarantee an ethical approach to measuring software engineering but does ensure that it is legal. This is where we will start. The GDPR rules for monitoring employee activity (such as engineering activity) rely little on consent. They are focused more on whether the processing is necessary for a legitimate interest of the employer and whether it violates the rights of the employee. This is because in the eyes of the EU, the employee cannot fully consent to any agreement because they could face retaliation from the employer if they don't.

Covert monitoring is only allowed in exceptional cases such as when investigating employee theft or fraud, it is not something an employer can do when measuring the software engineering activity of an employee. Employees therefore must be notified of how they are being monitored and why.

A data protection impact assessment must also be completed before monitoring is undertaken. Its purpose is to:

- Identify the purpose behind the monitoring and the benefits likely to be delivered.
- Identify any likely adverse impact of the arrangement.
- Consider alternatives to monitoring.
- Judge whether the monitoring is justified.
- Identify the lawful basis of processing the data.

So in the context of collecting the software engineering activity data of an engineer, the most important legal point is that they are told how and why their activity is being measured.

Legal data collection can still be an ethical overstep in the eyes of employees. According to a 2019 workplace technology survey by the Chartered Institute of Personnel and Development, approximately one in 10 workers think workplace monitoring will have more benefits than downsides.

I feel that there is little debate to be had around the collection of data as most of it is already in the repository and is needed for it to function. For example, their needs to be a record of who makes each commit, when they make it and what lines were changed. The main discussion is centred around how the data is actually processed.

Most organisations collecting this data will use platforms as such as GitClear and CodeClimate to process it. An important ethical consideration here is the security of the data. It would be a catastrophe if an employer gave private employee data to a 3rd party only for them to be breached. This problem can be mostly solved if users are only identified by username and personal information is not sent to the platform.

The biggest ethical dilemma to be had is how the analytics are actually used. Most developers would be ok with these metrics being used to track the progress of an agile development team but would be vehemently opposed to the idea of some Github analytics being used to decide which employees are kept during a period of lay-offs. This could result in unfair dismissals and the inevitable legal fallout which follows. There are many that believe we can use analytics to automate engineer management but there are many who disagree.

My personal opinion is that there is plenty of room to measure software engineering activity in industry, provided that it is implemented alongside a heavy dose of common sense. Metrics and charts can certainly be useful for management gaining an understanding of what is happening in a project, but humans are complex and business decisions should never be based solely on computational analysis of engineering activity. For example, a software engineer could be the best in their team but could have low metrics as they are spending considerable team helping weaker members. If a manager wasn’t aware of this and only looked at the metrics, they could get the completely wrong picture.

I believe that analysis of engineering activity should be used as an aid for informed managers, not a replacement.

Sources

https://ecommons.luc.edu/cgi/viewcontent.cgi?article=1205&context=cs_facpubs

https://www.apptio.com/blog/software-development-metrics/

https://www.gitclear.com/popular_software_engineering_metrics_and_how_they_are_gamed

https://www.pluralsight.com/blog/teams/5-developer-metrics-every-software-manager-should-care-about

https://www.pluralsight.com/blog/tutorials/code-churn

https://en.wikipedia.org/wiki/Cyclomatic_complexity

https://en.wikipedia.org/wiki/Halstead_complexity_measures

https://www.peoplemanagement.co.uk/experts/legal/gdpr-implications-monitoring-your-workforce

https://www.computerweekly.com/feature/Deploying-productivity-monitoring-software-ethically

http://2016.msrconf.org/

https://www.researchgate.net/profile/Jan_Sauermann2/publication/285356496_Network_Effects_on_Worker_Productivity/links/565d91c508ae4988a7bc7397.pdf

https://www.researchgate.net/publication/233893645_Ethical_Issues_In_Monitoring_And_Based_Tracking_Systems/fulltext/57a9a44308ae659d1823a450/Ethical-Issues-In-Monitoring-And-Based-Tracking-Systems.pdf

https://www.gitclear.com/

https://codeclimate.com/

https://aws.amazon.com/

https://en.wikipedia.org/wiki/Software_measurement