Measuring Software Engineering

Measuring Software Engineering **1**
Abstract 1
Introduction 2
Measuring Engineering Activity 2
Platforms 4
Data and Analysis 7
The Ethical Conversation 12
Conclusion 13

Abstract

This essay will examine the process of measuring software engineering. The first section will examine the methods used to measure engineering activity. Then the platforms that can be used to gather and perform calculations on the data gathered will be discussed. The third section will explain some computational methods used to profile the performance of software engineers. Finally, I will give my opinion on whether this type of profiling is ethical.

Introduction

Measuring productivity is something that has been done by employers to get better results from employees. Making the software engineering process faster and more efficient is a major goal as technology advances so quickly. There have been many discussions over the years about the best way to measure software engineering. As data analytics improve there are many more ways to analyse lots of data on certain projects and it may be hard to know what is the best data to analyse.

Measuring Engineering Activity

A piece of software is at its essence, a computer program that consists of lines of code. However this is irrelevant when it comes to the end product. The software could also include documents of any requirements or specifications. Any consumer, however, will use software based on its functionality rather than any documentation provided or how the code was implemented (Sidler, 2002). Essentially to measure software engineering you will need a ratio of the software value / the labour and cost involved in creating it.

In the past Lines of Code (LOC) was the measure of a software engineer. This would be divided by Programmer Month (PM). This calculation gave an indicator of the programmer's productivity. There was also another measure used for program quality which was the number of defects per thousand LOC (KLOC). The LOC were used as a *size variable.* In the mid-1970s it became clear that this was a crude measure. There were many different types of programming languages and comparing the lines of code of a lower-level language vs a higher level language was not possible using the LOC measurement. Thus there was an increasing interest in measuring the complexity of code (Fenton & Neil, 1999).

Data from commits could also be used as a metric. The number of commits a user makes may not be a good metric in itself as there may only be a few line changes or new comments. Similarly, the size of a commit doesn't give too much information on the progress of a project. It is possible that an engineer may be solving a problem over several commits. Commit messages will tie all the data together and indicate how much the user has worked on the project and what has been achieved in the commit. The commits will also tell us how much progress a user considers to be reasonable enough to commit.

A software defect is essentially a flaw that impairs the software or the software process. These defects are caused by human error but not all mistakes cause a defect. These defects will be detected during formal reviews and testing activities as well as design and code inspections. Problems can also be detected during the development of code. It is important that problem reports are generated following these discoveries. Customers may also report problems in the user experience (Florac, 1992). Keeping tabs on defects and problems in code will give a good indication of how the process is working in a company. Ideally, most of the defects will be caught early. So it would be better for a company to catch defects during reviews, inspections and testing. If the customer is reporting more problems than your software engineers are reporting during the development of a piece of software then something is going wrong during the process. Another question can be asked: can we analyse the data from these problem reports? You could of course count the number of defects per project and call it a day. As we know, things are rarely this simple. Thus we may need to indicate how urgent a defect is or how critical the issue is, maybe even the type of problem (hardware, OS). By assigning traits to a problem we can then begin to analyse the data.

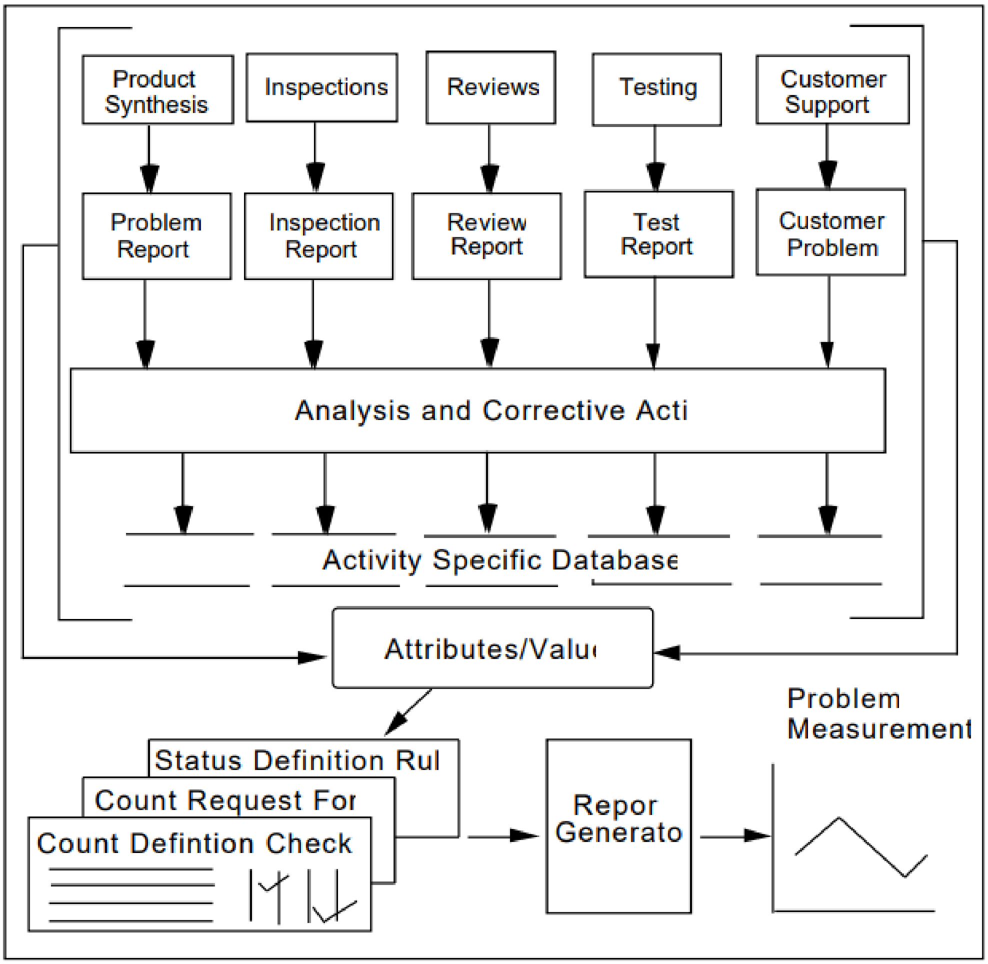


Figure 1: An example of methods for locating problems in software from (Florac, 1992, pg 10).

A pull request is a part of the code synchronisation required by version control servers. It is a request to change code in a central codebase. Pull requests are often used by managers as part of the engineering process. They require collaboration from a team and thus cannot be used to measure engineers on an individual level. When analysing the data from pull requests, comments can be analysed to see which comments engineers respond to more productively i.e. meaning issues will be fixed quickly (Ortu et al., 2017). Other metrics that can be pulled from pull requests include time to merge, pull request lead time, pull request comments etc. Pull request lead time is how long a pull request takes to be either merged or closed. This is an important measure of how long it takes before engineers are getting to solve pull requests after they are opened. Time to merge is how long it takes for the original branch to reach the master branch. Taking the difference between these two gives an indicator of how long an engineer works before opening a pull request. You can also pull the information from comments to see how much discussion there is before a pull request is closed. Measuring the size may also be a metric used (Guimarães, 2019). Of course, all this data is best used together to gather an accurate representation of how the team is merging and closing the pull requests. Ideally, you would want a team to be opening and closing pull requests at the same rate to keep the work flowing.

Platforms

Software companies use version control systems. These systems will gather information on each commit and this data can then be gathered and analysed. The repositories will have insight on pull requests, productivity, work style etc. This can give analysis into a team as to who is solving issues in the project, who is making the most significant changes in the code and so on. This data can be accessed using an *application programming interface* (API) such as:

- GitHub API
- BitBucket API
- GitLab API

These APIs are free and they offer raw data on repositories and users. Analysing the gathered data is another story. Most people in charge of software engineering teams will not have a deep understanding of statistics and wrangling meaning from data. So to properly understand the productivity of their team they will most likely need some software to perform analysis on this data. Luckily there are many platforms that do this kind of analysis and often will connect with GitHub or other such sites to perform analysis on data directly from the source. Some examples of this kind of software:

- **GitClear**

GitClear is a platform that amplifies how much is being done by software teams. It's a developer tool for GitHub and GitLab. Using git commits they will provide data about the code and development process. The analysis is not on simple data such as the number of commits or lines of code. Line impact analysis is one of the key features of the platform. It measures how much a commit evolves the codebase of a project (GitClear).

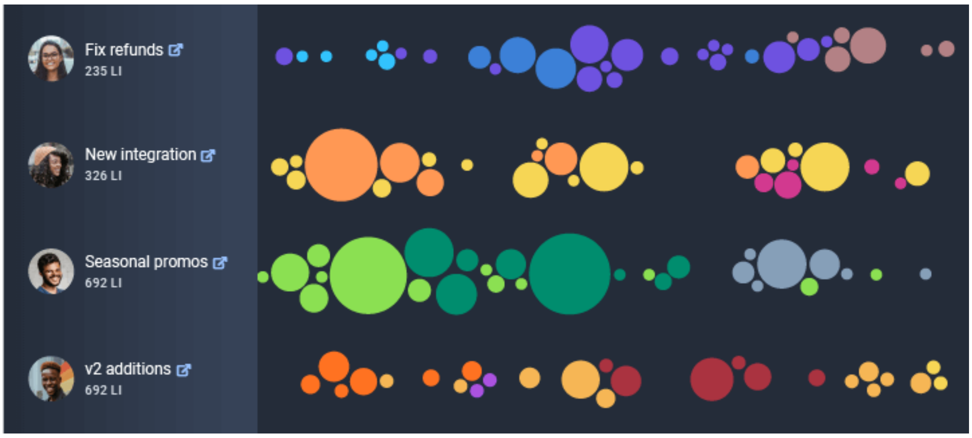
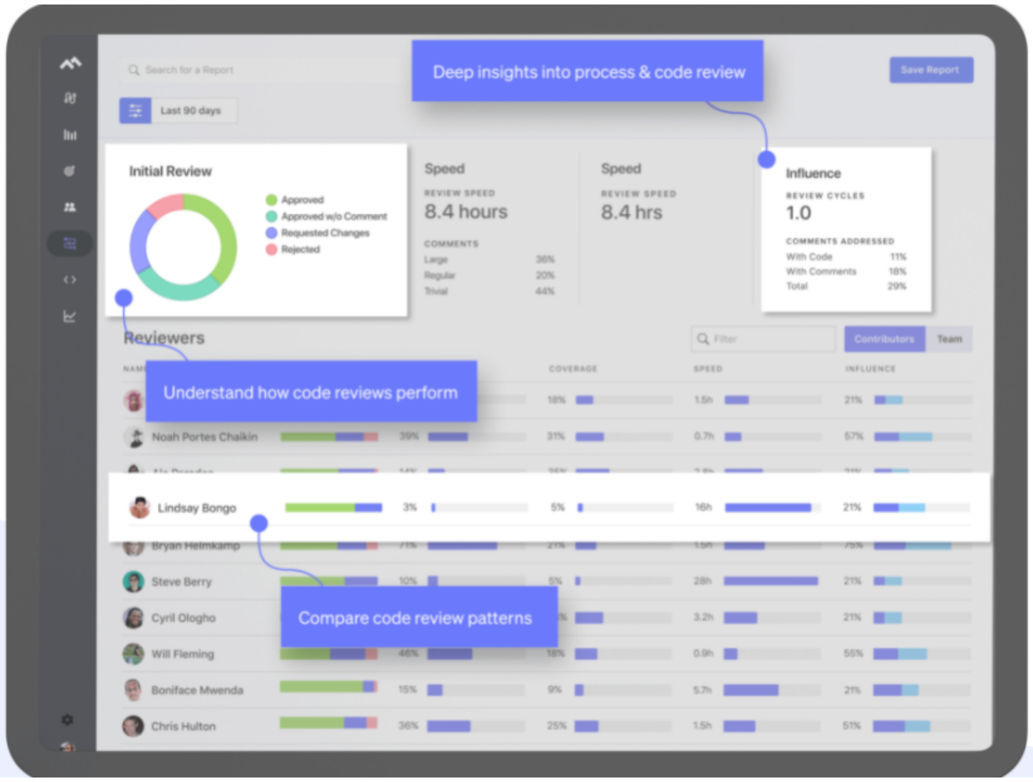


Figure 2: Line impact example from (GitClear)

- **Code Climate**

Code climate has different options for different needs such as Code Climate Velocity and Code Climate Quality. These are subscription services. They do more than just analyse code and the software will analyse each pull request before integration to catch any subtle issues. It allows you to test coverage and range of code with information on the capabilities of the code. It will analyse each team members skill in different languages. There is also other analyses available on pull requests etc (Code Climate).



Deep insights into process & code review

Understand how code reviews perform

Compare code review patterns

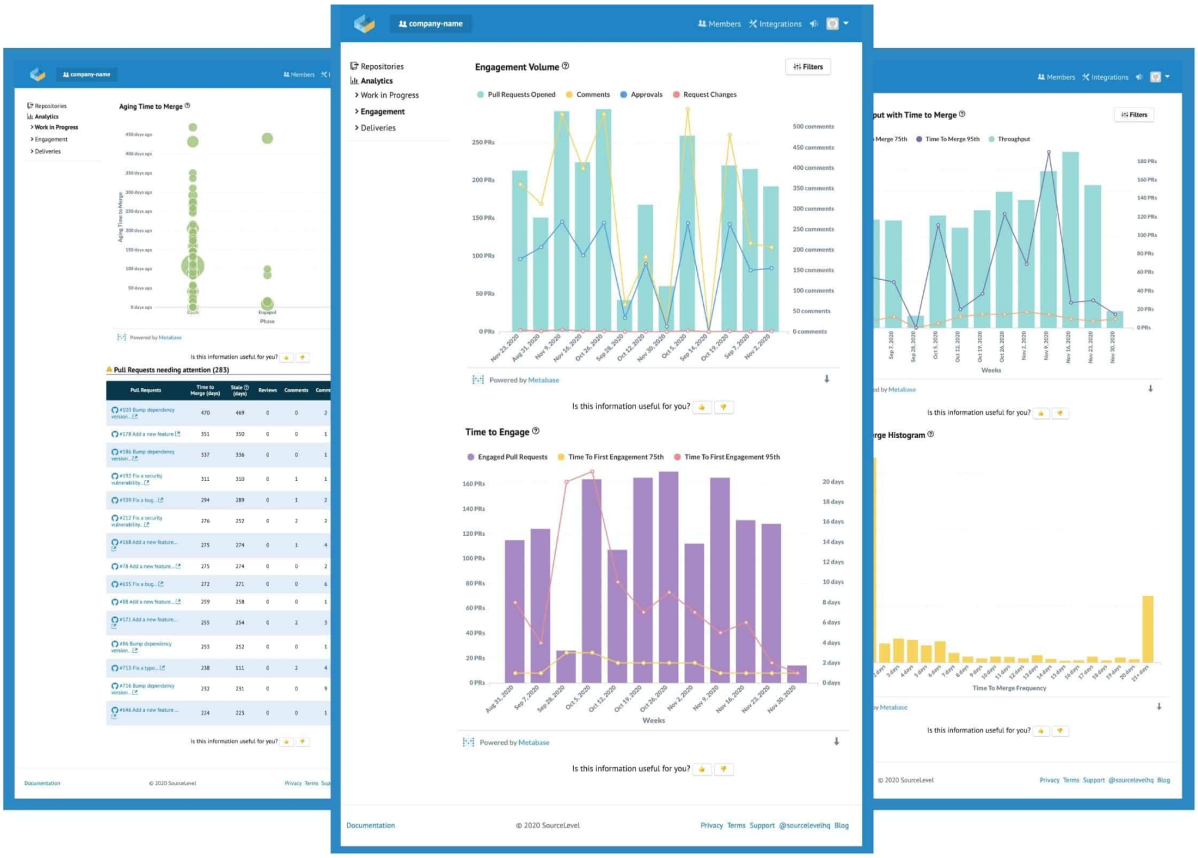
Figure 3: Code review statistics from (Code Climate).

- **Hackystat**

Hackystat is for visualisation, inrtpretation and analysis of the software development process and any product data. It is no longer under active development. It uses REST architectural principles. The service is open source which is a major benefit. It also strives to be neutral in terms of language, process and environment. The architecture is also service-oriented which is one of the only one of its kind (Johnson et al., 2009).

- **SourceLevel**

SourceLevel provides insights and merics using data from GitHub and GitLab. This platform does its analysis on pull requests. They do not use averages and instead use percentiles as averages can hide too much information when certain metrics have a lot of variation. They compare the 75th percentile and the 95th percentile. There is a lot of analysis done on timing on this platform from time to discuss to time to first review etc (SourceLevel). As most of the analysis is done on pull requests this will give teams an insight into how their projects move along a pipeline as they are progressed. This will show any weaknesses in the pipeline and ny places where improvement is needed. This would be useful in processes like agile development.



Pull Requests needing attention (283)

Pull Requests	Time to Merge (days)	Stale (days)	Reviews	Comments	Comm
#103 Bump dependency version...	470	469	0	0	2
#178 Add a new feature	351	350	0	0	1
#186 Bump dependency version...	337	336	0	0	1
#192 Fix a security vulnerability...	311	310	0	1	1
#339 Fix a bug...	294	289	0	1	2
#212 Fix a security vulnerability...	276	252	0	2	2
#168 Add a new feature...	275	274	0	1	4
#78 Add a new feature...	275	274	0	0	2
#635 Fix a bug...	272	271	0	0	6
#88 Add a new feature...	259	258	0	0	1
#171 Add a new feature...	255	254	0	2	3
#86 Bump dependency version...	253	252	0	0	1
#713 Fix a typo...	238	111	0	2	4
#716 Bump dependency version...	232	231	0	0	9
#646 Add a new feature ...	224	223	0	0	1

Figure 4: Some of the metrics from (SourceLevel)

Most of the above platforms do include data processing before presenting the results. When analysing data the presentation of results is important to give an indication of the effects of code and changes. These are just a few examples and while most are paid platforms, some have free versions that can be used as well. Not all platforms will work for the needs of every team or project so it is important to find one that will work for your specific requirements. Something worth noting is that some of the paid platforms are vague about the exact metrics that they use and thus more information must be acquired by inquiring directly. The open-source platforms are more transparent and generally are more flexible in terms of platforms, languages and environments used.

Data and Analysis

There are many ways to analyse the code in a piece of software. The first three sections of this will be focused on some computations that can be performed on gathered data and then I will discuss how machine learning can be used on this data to give more insight on the metrics being measured.

- **Function Point Analysis (FPA)**

Function Point Analysis was developed in 1979 by Allan Albrecht (IBM). There were problems with other system size measures such as LOC as discussed previously. A system was needed to allow different systems/languages to be compared regardless of the technology used. It measures the size of the system in two areas: *specific user functionality* and *system characteristics*. Specific user functionality is essentially a measurement of 5 types of functions that the application delivers for user requests. These consist of external input, external output, external enquiries, internal logical files and external interface files. Each function will be placed in one of these categories and classified from low, average or high, after this it will be given a weight. The weights will tell us the size of any information processing and the sum is referred to as the Unadjusted Function Points.

Function Point = (User Functionality) * (System Characteristics)

In terms of System Characteristics, there are fourteen that are identified to evaluate the functionality. They each have a Degree of Influence (DI). These will range from having no influence to strong influence (0-5). The sum of these will then tell us the Value Adjustment Factor for each project (The Government of the Hong Kong Special Administrative Region of the People's Republic of China, 2009).

The product of each of these gives us the Adjusted Function Point:

Adjusted Function Point = (Unadjusted Function Point) * (Value Adjustment Factor)

Function point analysis provides a good analysis of the relationship to effort. It also doesn't rely on the programming language. Using function point analysis is more beneficial than using LOC and is a better goal to have. There are some drawbacks though, as many models are based on LOC as it was used more readily throughout history so there is some conversion needed. It also requires some subjective evaluations and so it may have low accuracy. In terms of time, it is certainly more time consuming than evaluating LOC.

- **Constructive Cost Modeling (COCOMO)**

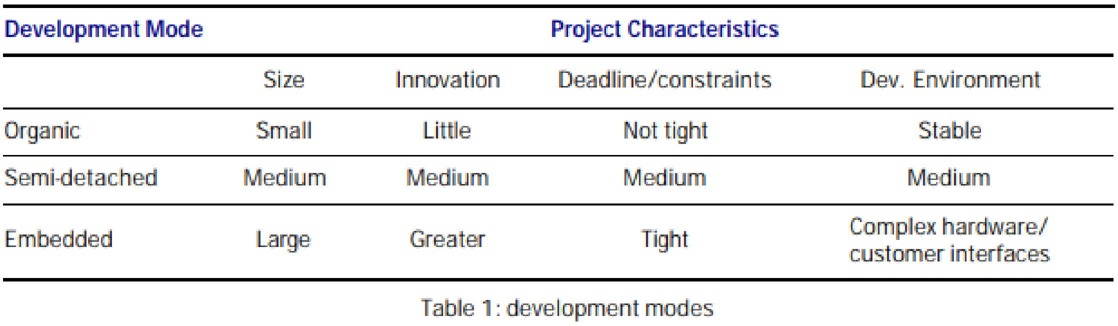
This model was first introduced by Barry W. Boehm in 1981. The software cycle underneath must be a waterfall. The original version of this model is single-valued and static. It computes effort as a function of the program size - computed using thousands of delivered source instructions (KDSI). The intermediate version uses a set of fifteen “cost drivers” as well as the program size. Finally, the advanced model uses all the characteristics of the intermediate version with the impact of the cost drivers on each step of the process. This has changed over the years as it has been developed but first I will focus on the two main equations it requires.

Development Effort: $MM = a * KDSI^{b}$

MM is a “man-month” i.e. a month of effort by a single person. In the original version, there are 152 hours in one of these months. Of course, these values will be different in every company.

Effort & Development Time (TDEV): $TDEV = 2.5 * MM^{c}$

The coefficients depend on the development mode of the project.



Development Mode	Project Characteristics			
	Size	Innovation	Deadline/constraints	Dev. Environment
Organic	Small	Little	Not tight	Stable
Semi-detached	Medium	Medium	Medium	Medium
Embedded	Large	Greater	Tight	Complex hardware/ customer interfaces

Table 1: development modes

Figure 5: Development modes for COCOMO from (Merlo - Schett et al., 2003).

When doing the basic version of this model there is no consideration as to the characteristics of the project. The intermediate uses fifteen cost drivers as described earlier and they are rated on a scale of “very low” to “very high”. The adjustment factor for something that is considered “normal” is 1. Using this system an EAF (Effort Adjustment Factor) must be calculated. A new man-month must be calculated:

$MM_{correction} = EAF * MM_{nominal}$

The advanced model requires calculations at each step of the process: requirements planning and product design (RPD), detailed design (DD), code and unit testing (CUT) and integration testing (IT). Each of the cost drivers is broken down by these phases (Merlo - Schett et al., 2003).

COCOMO is a good model to use as it is very transparent and the mechanism is evident in comparison to other models. Using the cost drivers will give a good indicator as to what factors are directly affecting the cost of projects. The result of the model will entirely depend on an accurate classification of the development mode of the project. It also requires an understanding of the needs of the project using data that may not be available. KDSI is not a size measurement as it is a length measurement and it cannot be used at the start of the project as there may not be many instructions written. It is also fair to say that KDSI is not a good measurement of effort as discussed earlier in this essay.

- **Cyclomatic Complexity**

Cyclomatic Complexity is a concept developed by Thomas McCabe. The general concept of cyclomatic complexity is concerned with conditions and control statements. It is a quantitative measure of the complexity of a program. With more complexity comes more risk for defects. When calculating cyclomatic complexity we use graph theory and these formulae:

1. The number of regions in a flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G

   $V (G) = E-N +2$

   Where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity, V(G), for a flow graph, G

   $V (G) = P + 1$

   Where P is the number of predicate nodes

The flow graph must be drawn such as in figure 3. The code must be broken into blocks that are split by control statements such as while, for, if, goto etc. These will be the nodes of the graph. If a node *f* can branch to node *g* then they must be connected in the graph. The control flow of a program can be created mechanically using either procedural or object-oriented languages. Ideally, the cyclomatic number of a graph for a simple program with no conditions will be non-zero (Ikerionwu, 2010).

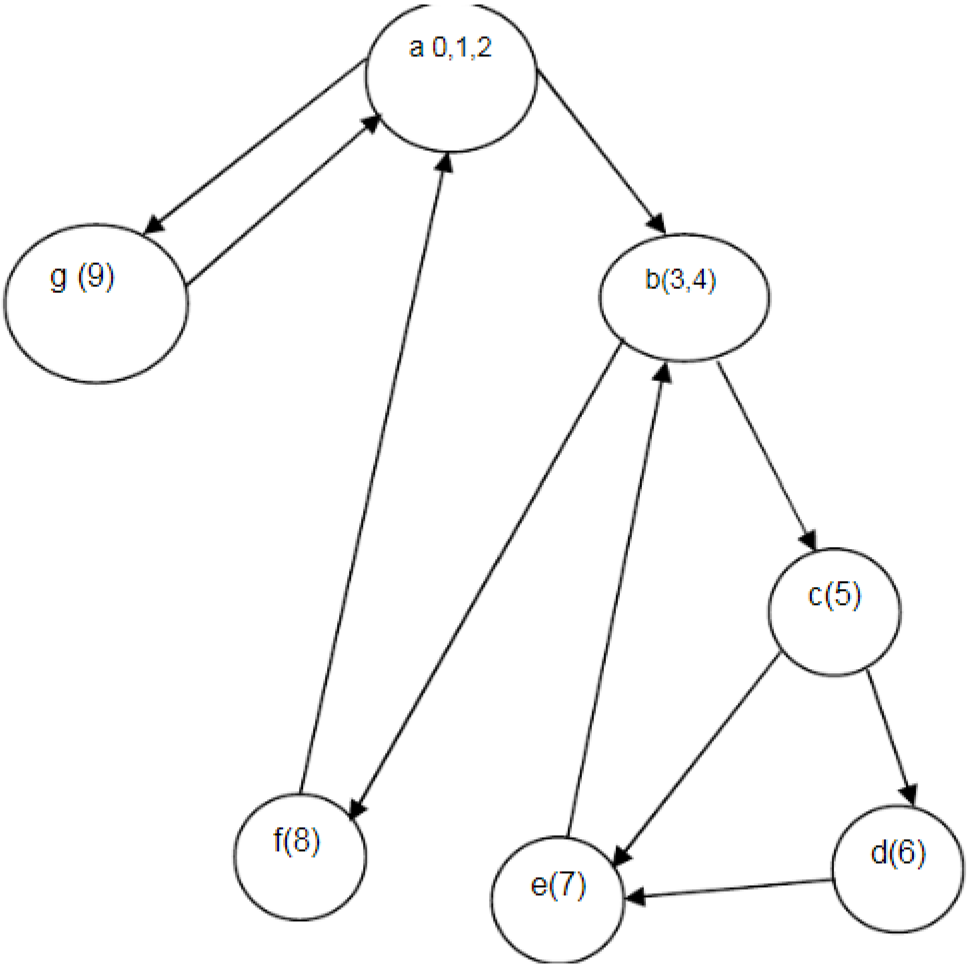


Figure 6: Flow graph of a program module from (Ikerionwu, 2010).

The advantage of cyclomatic complexity is that it gives a good indication of when a program module is too complex. Ideally, the cyclomatic complexity of a module should not exceed 10. The code is easier to test as complexity decreases. However, sometimes more branching will give more readable code (Karanth, 2016).

- **Machine learning**

Machine learning can be used for data analysis and it has been improving in the past few years. As machine learning algorithms require historical data then this can be used after gather various data on the process being used by a team. The data must be acquired using sensors in the environment the users are coding in. Then this data is fed into a machine learning model. Neural networks is one of the main ones used. The model will then be applied and maintained in order to keep it updated. This basic system will require different categories of metrics to measure (Beal et al., 2017). Artificial intelligence (AI) uses machine learning and this is improving with time and can do data analysis as well so it goes without saying that this most likely will become the standard in the industry.

The Ethical Conversation

Ethics must be considered when doing any kind of analysis of personal data. This is especially clear to people after the introduction of the GDPR (General Data Protection Regulation) in 2018. People are a lot more aware of how their data is being collected and processed. In light of this, the question must be asked. Does each engineer consent to their data being processed and analysed in order to assess their productivity. I can imagine that not everyone would be happy with this.

There have been plenty of dystopian films and media regarding the faults of measuring the productivity of workers. Most likely the measuring of software engineering would not extend to the measuring of heart rate and eye contact and the likes as is done in these films. That is not to say that it won't happen in the future and in my view that is certainly unethical. The introduction of GDPR was intended to slow down the exploitation of personal data but companies will find a way around this without a doubt.

I think it is important that people do have control over their own personal data. While it may be beneficial for management to have access to data from workers in order to promote productivity, I don't think that from an ethical point of view that they should. What's important to remember is that each engineer in a company is also a person. In general, each engineer specifically in such a specialist job like software engineering will be a fairly motivated person. Giving each engineer access to their own data may be a good solution to this problem. Each worker can then assess their own performance and work on improving what they regard as important in their work. Something that is worth remembering though, is that if workers are rewarded for achieving certain things in their work then it is certain that they will focus on these things. So is it worth doing this kind of analysis on your workers to encourage them to work harder or would it be easier for management to change the work environment to keep the workers happy?

Doing data analysis on worker productivity is not the only way to improve worker productivity either. That is why companies do not focus solely on these metrics. It has been seen in some of the bigger tech companies in the past few years that they are focusing more on keeping employees comfortable to motivate them. Employee satisfaction at their job will drive them to work harder. This method also doesn’t invade worker privacy and thus wouldn’t lead to ethical concerns.

Conclusion

In this essay, I discussed the measuring of software engineering. These measurements can be useful for employers to see where things are going right and where things are going wrong in the software process. It can analyse the work of a team and even individuals. There are clearly many platforms out there that provide services to do these analyses. I gave some examples of computations that may be done on the code and the process by software engineers. While this may be beneficial to employers, from an ethical point of view it may be questionable. In the world of Big Data where everything from our purchases to our browsing habits is analysed, is work just another place where each engineer is a number? From a capitalistic perspective absolutely but I think this can be shifted to focus more on creating productive work environments.