

CSU33012 – Software Engineering
Measuring Software Engineering Report

Cormac Madden

Student Number 18319983

Contents:

1. Introduction
2. Methods for Measuring Software Engineering
3. Metrics for Measurement
4. Overview of Computational Platforms Available
5. Algorithmic Approaches
6. Ethics and Concerns of the Analytics

Introduction

Measuring processes is a standard practice in Engineering operations and it is considered necessary for the coordination of an organization. However when it comes to the relatively new discipline of software engineering, measuring productivity, efficiency and accuracy, has proven to be a lot more complicated than in other engineering disciplines.

This report will discuss what sort of data one may be interested in when analysing the productivity of software engineers and what methods can be used to measure it. I will outline the various metrics, computational platforms, and algorithms used to capture, and present the data in a useful manner. Finally, I will describe some of the ethical and legal aspects that must be considered when doing analysis of software engineering.

Reasons for Measuring Software Engineering

When people talk about measuring software engineering, there are two contradicting adages: "What's measured improves". - Peter Drucker and, "When a measure becomes a target, it ceases to be a good measure." -Goodhart's Law

Software Engineering is a well paid profession, and because of this it's important for managers to maximize the return on their investment. The interview process is often a very inadequate way of gauging the potential of a software engineer. For example, tricky programming challenges in an interview, will not show you how well someone will work in a team. Companies need to effectively find and also try to keep a hold of their effective software engineers. In fact, the tech sector has the highest turnover rate at 13.2% out of every single business sector, according to a recent turnover report from LinkedIn.

Hiring, monitoring performance and retaining the best employees are all clearly important objectives for any organisation and generating more information for those decisions, creates a strong argument for why management should be measuring software engineers and the development process.

Methods for Measuring Software Engineering

Peer Review

Many companies nowadays (most notably Google), find peer review is the most useful method for analysing a software developers' ability. This involves getting other developers to write reviews about their colleagues. Peer review is Google's most important factor when considering whether an employee should get a promotion or not. The advantage of this method is that you are judged by your colleagues on your knowledge, the quality of your code, and how well you contribute to the team. The people judging you know what you're like to work with, the effort put in and the difficulty of the problem you're trying to solve. There are however some obvious disadvantages. Office politics get in the way of how well one engineer will judge another's work. When this becomes an issue it is very difficult to combat and overcome.

Metrics for Measurement

There is an overwhelming amount of choice for managers and teams when it comes to what metrics they should focus on and use to measure their progress. Choosing the right metrics and interpreting them correctly is vital in order for them to be effective.

Lines of Code

The most obvious, yet most obviously flawed metric, for measuring software engineering. Strictly measuring LOC (lines of code) is flawed when comparing languages that require varying amounts of code for the same function. Concise, clear and effective code is often more difficult to write than long code, and it is very easy for software developers to manipulate a simple LOC metric. The list of disadvantages for this metric go on.

However, it is the most popular metric, and is useful when used in more depth or combination with other metrics.

There are two types of LOC; physical and logical. Physical counts the total lines of code in a project. This includes comments and blank lines if the LOC consists of less than 25% blank lines. Logical LOC counts the number of executable statements. I unfortunately learnt this the hard way when doing a programming project in Year 2 of college. I took on the responsibility of writing a large proportion of the code for the project and so another team member offered to write up a section of comments for each of the functions. We were using

Tortoise SVN and of course I was not so happy when I discovered the analytics section and saw them tower over me with their commit % calculated by LOC.

LOC is also used relative to time:

TT100 Raw: This is the time it takes any engineer to produce 100 lines of code.

TT100 Productive: The time it takes for an engineer to produce 100 lines of code, after churn. These are metrics both available when using GitPrime and it is recommended to use them together.

Task Completion Rates / Burndown Charts

A burndown chart is a high-level measurement tool for visualising planning and monitoring the progress of a project.

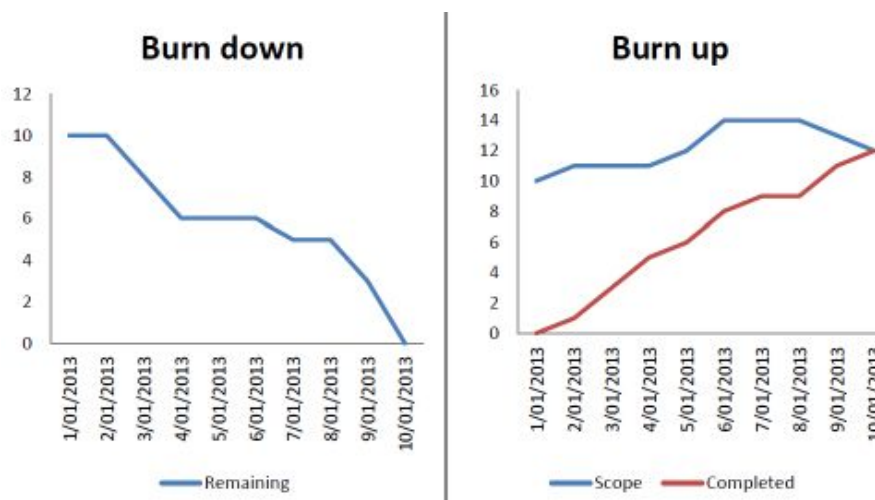
It is a graph of the project's tasks against time. Both the estimated task completion date and realised completion date are plotted in order to show the discrepancy between them.

This information helps teams with making decisions, regarding adding or dropping features, based on whether the project is ahead or behind schedule.

Discrepancies can either be the result of poor time estimates or unproductive workers.

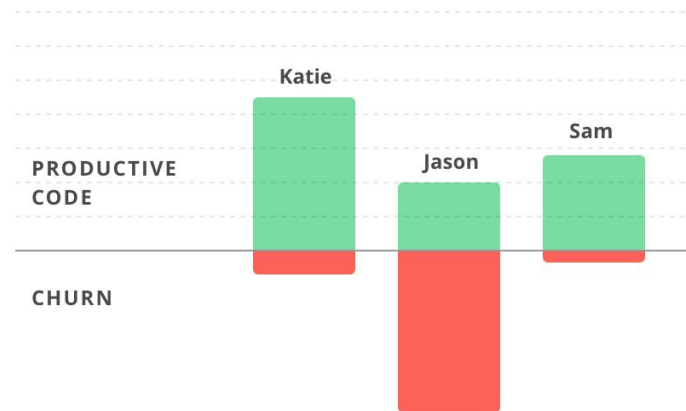
However, more information is required for a manager to determine the cause of a problem and make constructive changes. The burndown chart however, provides a good starting point for conversation.

An alternative to the burndown chart is a burn up chart. A burn up chart is designed so that new tasks can be added as the project developed. It focuses more on incremental progress and tasks done as opposed to tasks remaining.



Code Churn

Different systems implement and define code churn slightly differently but it is essentially the metric used to measure the volume of “changed code” over a defined period of time. It tries to ignore new features being added and tries to target adjustments to recently added code.



Measuring code churn gives managers an understanding of the evolution of their software development process. A high code churn rate can be caused by: quickly changing requirements; a difficult problem; rapid prototyping; incapable engineers or polishing. Each of these scenarios requires changes to the code and more commits, causing a growth in the code churn rate.

A high churn rate can either represent wasted effort reworking existing code, or making useful changes to add new requirements. This will depend on the development framework being used by the team. It is important that the churn metric is interpreted correctly. An agile framework will have a lot more changed code when compared to a more waterfall approach which should have more added or new code.

Lead time / Cycle time, Open/Close Rates

These two metrics are very alike but they measure different sections of the development process. **Lead time** is the time between the definition of a new feature and its availability to the user. **Cycle time** is the total time between the moment when the work starts until the project is completed and ready for delivery. These metrics help to estimate when a project will be completed and how quickly new features can be developed and delivered. They help engineers understand how long work takes to flow through their value streams. Tracking lead time helps determine the impact of changes that have been made.

One can understand a developer's speed per task by tracking their cycle time. This is done by breaking down the total throughput into median time by status or issue type. This helps to pinpoint bottlenecks and set accurate expectations.

These metrics however don't account for varying difficulty and time required for tasks.

Software “Availability” (MTBF, MTTF, MTTR)

In reliability engineering, the term availability means, the probability that an item will operate satisfactorily at a given point in time when used under stated conditions in an ideal support environment. Normally high availability systems have an availability above 99.98%.

Software availability is measured in terms of mean time between failures (MTBF).

MTBF consists of mean time to failure (MTTF) and mean time to repair (MTTR). MTTF is the difference of time between two consecutive failures and MTTR is the time taken to fix the failure.

$$MTBF = MTTF + MTTR$$

Steady state availability (A) represents the percentage the software is operational.

$$A = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

For example, if MTTF = 1000 hours for a software, then the software should work for 1000 hours of continuous operations. If for the same software the MTTR = 2 hours, then the.

$$MTBF = 1000 + 2 = 1002$$

And the availability (A):

$$A = 1000/1002 \approx 0.998$$

This is just one of many metrics used to measure the reliability of software. Due to the many applications of software in safety and security critical systems, software reliability is now a very important area of research.

Code coverage (Unit Testing)

Code coverage is another metric that is very important for testing the reliability of software. It indicates the percentage of the code that is being covered when running test cases. This metric is useful when applied to newly written code but should not be taken too seriously when applied to large existing codebases.











For example, suppose you find a method in your codebase with a ridiculous metric, such as an NPATH complexity of 52 million. That means that it could take up to 52 million test cases to fully exercise every path through the code. One could try to put the code into a simpler structure, but before you do that, consider what the business impact would be. Chances are, the old, ugly code works well enough, and was tested when it was created (though its test coverage may be inadequate). It's best if the team agrees to the level of compliance and rules to which their code is subjected, but is also aware that examining outliers and getting concerned about trend blips can waste a lot of time.

Overview of Computational Platforms Available

In order to properly assess the software engineering process, it is not enough to merely collect data. The data must be collected, organized, analyzed and compared. There are many computational platforms to choose from, all with different benefits and areas of expertise.

Generic Employee Monitoring Software

In many firms the software engineering team is not particularly large and so firms stick to more generic monitoring tools and don't monitor software engineers differently to other employees. The following chart by PCMag is an overview of the popular employee monitoring platforms. They determined Teramind to be the best for close monitoring with screen recordings, live views of employee PCs, tracking emails, and keystrokes all the way to Zoom sessions. VeriClock is also recommended as the best time tracking solution for remote workers. It doesn't offer any live video features but comes at an affordable pricing.

Our Pick		User Privacy Settings	Stealth Monitoring	Screenshots	Screen Recording	Schedule Ahead	Remote Desktop Control	Policy Customization	Physical Agent Install
 Teramind	>	✓	✓	✓			✓	✓	✓
 Time Doctor	>	✓	—	✓	✓	✓	—	✓	✓
 VeriClock	>	—	—	—	—	—	—	—	—
 Veriato Cerebral	>	—	✓	✓			✓	✓	✓
 ActivTrak	>	✓	✓	✓			—	—	✓
 InterGuard	>	✓	✓	✓			✓	✓	✓
 Controllo	>	—	✓	✓			—	✓	✓
 Hubstaff	>	✓	—	—	✓	—	—	✓	✓
 StaffCop Enterprise	>	✓	✓	✓			✓	✓	✓
 Veriato 360	>	✓	✓	✓			—	✓	✓

The following platforms are some of the most popular for measuring some of the more software development specific metrics outlined in the previous section.

CodeScene



CodeScene is a code analysing and visualization tool that predicts and uncovers hidden risks and patterns in code. CodeScene analyses trends over time to uncover possible productivity bottlenecks, parts of the code that may be difficult to maintain, technical risks, areas for improvement regarding productivity and quality gain, and information about the knowledge distribution between team members. This platform enables managers and developers to visualize their development process. It is also very easy to set up by simply logging in with a GitHub account.

CodeScene measures code churn, and tracks tasks in relation to estimated deadlines. The platform generates a commit activity chart which shows the number and size of commits, which is linked to their authors. CodeScene calculates an active contributors' trend which enables teams to evaluate the impact of adding more contributors to a project.

Hackystat

Hackystat is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development processes and product data. This framework allows individuals to collect and analyze PSP (Personal Software Process) data automatically. It was developed at the University of Hawaii and is their third generation of PSP analytics systems. Hackystat sensors can be installed into various tools such as editing environments, build systems, and configuration management systems. These sensors collect data which is then sent to and stored on either a Hackystat SensorBase web service or locally. This ensures a high level of privacy, which is important in business. The SensorBase repository can be queried by other web services to utilize the data, integrate data into a project, or generate various visualizations of the data. Engineers can also set alerts to be notified when specific conditions occur.

Algorithmic Approaches

Halstead's complexity measures

Halstead complexity measures were developed to measure a program's complexity directly from source code, with emphasis on computational complexity. The measures were developed by Maurice Halstead for IBM as a means of determining a quantitative measure of complexity directly from the operators and operands.

Halstead's metrics can be used on C/C++/Java source code.

These metrics only make sense at the source file level, and vary with the following parameters:

n_1 = number of unique operators

n_2 = number of unique operands

N_1 = number of total occurrence of operators

N_2 = number of total occurrence of operands

These parameters are used to calculate the following metrics:

Vocabulary: $n = n_1 + n_2$

Program Length $N = N_1 + N_2$

Calculated Estimated Program Length: $N^{\wedge} = n_1 \log_2 n_1 + n_2 \log_2 n_2$

Volume: $V = N * \log_2 n$

Difficulty: $D = (n_1 / 2) * (N_2 / n_2)$

Effort: $E = V * D$

Errors: $B = V / 3000$

Time Required To Program: $T = E/18$

```
void sort ( int *a, int n ) {  
    int i, j, t;  
  
    if ( n < 2 ) return;  
    for ( i=0 ; i < n-1; i++ ) {  
        for ( j=i+1 ; j < n ; j++ ) {  
            if ( a[i] > a[j] ) {  
                t = a[i];  
                a[i] = a[j];  
                a[j] = t;  
            }  
        }  
    }  
}
```

$V = 80 \log_2(24) \approx 392$

- Ignore the function definition
- Count operators and operands

3	<	3	{
5	=	3	}
1	>	1	+
1	-	2	++
2	,	2	for
9	;	2	if
4	(1	int
4)	1	return
6	[]		

1	0
2	1
1	2
6	a
8	i
7	j
3	n
3	t

	Total	Unique
Operators	N1 = 50	n1 = 17
Operands	N2 = 30	n2 = 7

Maccabe's Cyclomatic Complexity

McCabe's cyclomatic number (McCabe, 1976) is a very popular metric. It is easily computed, and is often used for quality control (Fenton, N. E., Martin, N., 1999). It measures the complexity of a program by counting the number of linearly independent paths through the code.

A higher cyclomatic number indicates a more complex code. This means the code will be more difficult to understand and have a higher probability of containing errors.

The number also indicates the amount of test cases that need to be written for all paths to be covered.

Cyclomatic complexity is calculated as follows:

$$\text{Cyclomatic complexity (CC)} = E - N + 2P$$

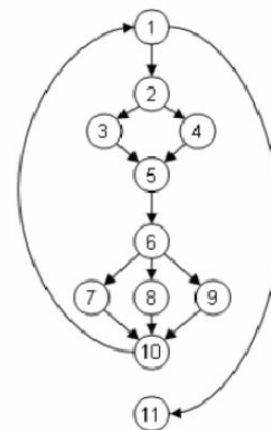
Where:

P = number of disconnected parts of the flow graph

E = number of edges

N = number of nodes

Node	Statement
(1)	while (x<100) {
(2)	if (a[x] % 2 == 0) {
(3)	parity = 0;
	}
(4)	else {
	parity = 1;
(5)	}
(6)	switch(parity){
	case 0:
(7)	println("a[" + i + "] is even");
	case 1:
(8)	println("a[" + i + "] is odd");
	default:
(9)	println("Unexpected error");
	}
(10)	x++;
(11)	} p = true;



If a high cyclomatic complexity number is returned, developers should either ensure its well tested or simplify the code where possible.

There are however some disadvantages to the cyclomatic complexity metric. It is the measure of the program's control complexity and not the data complexity. In case of simple comparisons and decision structures, it may give a misleadingly-high figure.

COCOMO Model

The COCOMO Model (Constructive Cost Model) is a regression model based on LOC. It is a cost estimate model for software projects and often used as a process of reliably predicting the various aspects of developing a software project such as effort, cost, time and quality. It was proposed by Barry Boehm in 1970 and is based on the study of 63 software development projects.

Different models of Cocomo have been proposed to predict the cost estimation at different levels. Whether your project is described as Organic, Semi-Detached or Embedded will determine the constants you use in the cocomo model.

Boehm's definition of organic, semi detached, and embedded systems are as follows:

1. **Organic** – small team size, the problem is well understood and has been solved in the past, the team members have nominal experience regarding the problem.
2. **Semi-detached** – team-size, experience, knowledge of the various programming objectives lie in between that of organic and Embedded. less familiar and are difficult to develop compared to the organic ones and require more experience and better guidance and creativity.
3. **Embedded** – Highest level of complexity, creativity, and experience large team size
The developers need to be sufficiently experienced and creative to develop such complex models.

SOFTWARE PROJECTS	A	B	C	D
Organic	2.4	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

These constants are used in the following formulae.

$$\text{Effort: } E = a(KLOC)^b$$

$$\text{Time} = c(\text{Effort})^d$$

$$\text{Personrequired} = \text{Effort}/\text{Time}$$

The constant can be adjusted to give a result more closely tailored to the project in question. There are more complex versions of this model that take in more parameters and give more results in the output.

Ethics and Concerns of Analytics

A right to privacy?

These new tools that are continuously getting better at monitoring employees and software developers, can raise some ethical concerns. At what point, if any do these tools conflict with our right to privacy in the workplace? At the moment it seems that software engineers somewhat accept the monitoring of their code and performance when working for a company, but when other information such as video, screenshots, and statistics such as number of breaks, employee attention span and coworker engagement begin to be recorded, it raises the question. Where should the line between privacy and company productivity be drawn?

Data protection has become a huge talking point globally especially after the passing of the EU's GDPR regulation. This has had huge impacts on companies that deal with the selling of user information. The question then arises of whether companies can sell the data they've collected on their employees. Workplace analytics is expected to be a \$1.87 billion industry by 2025. I believe that we are entitled to certain levels of privacy even in the workplace. The recent passing of the European Union's General Data Protection Regulations peoples' data is now treated as their own personal property. One important right outlined in the regulation is known as 'the right to be forgotten'. Individuals are now allowed to request for data collected on them to be erased.

Algorithms that humans don't understand.

Another concern I believe should be raised is the impact machine learning and AI could have on the analysis of software engineers. Will it get to a stage where the algorithm making decisions on the engineers is not a simple formula that anyone can break down. Is there a risk that managers could blindly follow the AI output, because it believes in the "Neural Network Human Resource Manager's" judgement, over their own? It seems like an unlikely

possibility. Even Google today, turn to peer reviews as the most important source of information when promoting an employee.

Employee Wellbeing

The feeling of being constantly watched can also significantly affect the mental health of engineers and I believe could cause them to be afraid, and write code as fast as they can, potentially sacrificing well optimized and efficient code. If software engineers are only graded based on features implemented or amount of code written, it encourages bad habits such as staying up late working, ignoring documentation, and skipping out on important research before starting a project. These habits have obvious negative impacts on both the employee and the firm.

The mental stress that 'over the top' monitoring software can have on employees is simply immoral. Employees will be afraid to have a bad day at work. Days when they just can't manage to solve difficult problems will have a significant effect on their mental health. I believe the effects of having poor performance scores at work will also bleed over to the overall wellbeing of employees outside of the workplace. Feelings of worthlessness may arise when employees see that their score is dropping or is below that of their coworkers.

The software mentioned above already have features that track a lot more than your average employee is comfortable sharing. The trick seems to be, how well a company can balance their performance monitoring and trusting their employees. A sense of trust and mutual respect between the software engineers and their managers can go a long way, not only for employee wellbeing but also employee retention, which as we discussed earlier is a serious concern in the software development industry.

In many workplaces, rudimental versions of these systems are put in place and rarely or 'lightly' used to actually monitor the employees. This however is absolutely not the case in

less developed parts of the world. In social media content censoring, or phone call scam centers very strict employee monitoring software is used to intimidate and force their workers to meet their high objectives. They may not be monitoring software development but the software used to track their employees is often the same, just used to its full capabilities and heavily enforced.. It's worth bearing in mind the almost sinister side of these monitoring systems before we blindly subscribe to them, hoping to boost productivity.

Bibliography

1. <https://www.businessinsider.com/how-google-performance-reviews-work-2015-6?r=US&IR=T>
2. <https://www.forbes.com/sites/forbesbusinessdevelopmentcouncil/2018/06/29/the-real-problem-with-tech-professionals-high-turnover/?sh=522dabcd4201>
3. <https://uk.pcmag.com/cloud-services/90313/the-best-time-tracking-software-for-2020>
4. <https://codescene.io/>
5. <https://code.google.com/archive/p/hackystat/>
6. <https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679#d1e2001-1-1>
7. www.w3schools.com
8. <https://www.bbc.com/worklife/article/20170524-can-a-fear-of-getting-fired-make-you-work-harder>