**CS3012: Measuring Software Engineering**

**Name: Cian McGovern Leahy**

**Student No: 17329433**

**Introduction**

I plan on structuring this essay for measuring the software engineering process using four different headings: Measurable data, computational platforms available, algorithmic approach and ethical concerns.

Software is currently used in all major industries to improve productivity and innovate new solutions for businesses and governments and as a result, efficient, easy to understand and usable software is crucial for society. The measurement of the process itself however is not so simple. It is equal part art and science and cannot be easily quantified without risking oversimplification. Nevertheless it’s quantification should be attempted due to its importance in the world. There is a lot at stake for businesses building software due to fierce competition and a neverending technological arms race. Thus there is a huge commercial interest in ways to measure the software engineering process as well.

**Measurable Data**

**Lines Of Code**

A simple metric for how much work a software engineer has done is the number if lines of code they write. This can be taken on a per hour basis or over any arbitrary period of time. This has some serious flaws however as some languages may require significantly more lines of code than others for the same task. A software engineer that uses fewer lines of code to accomplish the same task may actually be better than one that uses more lines of code as they may come up with a more efficient solution which runs faster. A Software engineers may have different writing styles for there code. i.e. one person might use a lot more whitespace or split up code on different lines that could be done on just one line and therefore lines of code by itself is a somewhat flawed metric. As Bill Gates once said: “Measuring programming progress by lines of code is like measuring aircraft building progress by weight.”

**Numer of bugs per 1000 loc**

The industry average for number of bugs per 1000 lines of code is 15-20. The basic human error rate is 5%[50]. Debugging can take a substantial amount if time and is often what a software engineer spends the majority of their time on. It has been estimate that developers spend half of their time debugging software which costs an estimated $312 Billion per year[52]. Clearly having a low count of bugs in your code is extremely valuable and thus this may be a good metric for how good a software engineer is.

**Debugging**

Debugging is a lot more difficult to measure than loc of number of bugs per 1000 loc as bugs vary in complexity. Some may be solved almost immediately whereas others can persist in legacy code bases for weeks or even months. If a complex bug is contained within hundres of thousands of lines of codes it may be almost impossible to fix. Thus quantifying this is difficult to do.

**Number of commits**

Number of commits is a simple metric which is often used by big tech companies to gauge how productive a developer is. This is a generally good metric although it has some flaws as some developers may commit all new work that they have done almost immediately and this the indvidual value of each contribution is low wheresa others might work on a new commit for weeks before actually commiting the code.

**Test/Code Coverage**

Another easy to measure metric. Code coverage measures how many and what percentage of all the lines of code have actually been run during testing. It can be used often to detect bugs before they become lost in mountains of code.

**Code Reviews**

Code reviews is when another developer checks the code that a developer wrote. This is done to ensure that the code fulfills the functionality it was written for and to give feedback to the developer on the readability of their code. The reviewer can add comments to the code to give feedback or talk to the developer in person about things that they can improve on. The optimal metrics for code review are 300-500 loc per hour according to a study done by smartbear software[51].

**Computational Platforms available**

**GitPrime**

Used with Git to provide visualisation and reports on individual developers and entire projects. It provides engineering leaders with metrics in context to ask better questions and advocate for the team with substantive data:

According to the gitprime help page[53] GitPrime can be used to answer the following questions

* How much of your team's burn is going to refactoring old code?
* Did the change to our sprint cycle have a net-positive effect?
* How do Wednesday all-hands meetings affect productivity?
* How well do we share knowledge in code reviews?
* What did engineering accomplish last week?

This increases visibility about team contributions and gives developers visual feedback which may be more effective than just telling them what to do.

**Slack**

Slack is like a massive chatroom for a whole company. It allows workers to send files and images to eachother and supports private group chats between multiple employees. It can also be integrated withDropBox, Google Drive and GitHub.

**Trello**

A task management app which my team used in our software engineering project last year. There are many similar such apps. Trello gives a visual representation of the work that has been done. The work that is left to do and what work is currently being done. It follows the Canban system which allows developers to maintain high productivity while still maintaining flexibility.

**Algorithmic Approach**

**Halstead’s software Metrics**

In 1977 Halstead established an empirical science of software development. This was quite useful at the time as similar work had not been done very effectively. It is not so useful today as we use many different libraries for writing code but it still has some use. He declared that metrics of software should:

* Reflect the implementation or expression of algorithms in different languages
* Be independent of their execution on a specific platform
* Metrics should come from the code itself only

His method is as follows:

1. Count tokens and determine which are operators and operands.
   1. n1: Number of distinct operators
   2. n2: No of distinct operands
   3. N1: Total no of occurences of operators
   4. N2: Total number of occurences of operands
   5. n1\*: Number of potential operators
   6. n2\*: Number of potential operators

n1\* and n2\* are the minimum possible number of operators and operands for a module and a programme respectively.

1. Using these numbers we calculate metrics
2. Programme vocabulary: n= n1+n2 = total number of unique operators and unique operands occuring
3. Programme Length: N= N1 + N2 = the overall length of the programme
4. Programme Volume : V = Size \* (logbase2 (vocabulary)) = N\*logbase2(n) = the space necessary for storing the programme
5. Difficulty: D = (n1 / 2) \* (N2/n2) = Measuring how difficult the programme is to handle
6. Effort: E = D \* V = Level of mental activity needed to translate the existing algorithm into implementation on the specified programming language
7. Time Required: T= E/18

Some extra things to note

* Comments are not considered
* Functions calls are considered as operators
* All brackets, commas and terminators are operators
* All variables and constants are operands

**Cycle Time**

The total time that elapses from when work is started on a project to when it is completed. A simple metric and easy to understand. Often there is conflict between developers and non technical workers as people who are not familiar with software design may massively underestimate the time it takes to complete certain tasks. This is a common source of tension between managers with no programming background and swegs. This may also be difficult to estimate before and is often underestimated.

**Waterfall**

A simple 6 step process. Not as common as agile methods theses days for software engineering but still used.

1. Requirements: Defines what the application should do and needs to do. Requirements are addressed first before preferences are addressed
2. Analysis: System is analysed to generate the models and business logic that will be used in the application
3. Design: Technical design requirements. Choose language and technologies used in the project. Design specification report to explain how the logic will be implemented.
4. Coding: Write the actual code using the language and logic specified in step 3
5. Testing: Other swegs and beta testers will try to find bugs in the code by using the software and testing edge cases.
6. Operations: Software is released into the live environment. May be updated and maintained by swegs.

**Agile**

Various approaches to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customer/end user. [54]

**Scrum**

Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value. It is lightweight and simple to understand. It is essentially the opposite of interwoven mandatory components. Based on the scientific method of empiricism: see what works and do more of it and vice versa. In scrum you have daily meetings describing what you have done so far for the sprint(collecton of tasks to be completed) and talk about what you plan on doing for the day while adapting to the changing needs of the business or organistaion that you are developing for.

Flexibility and willingness to adapt are the key components of a successful scrum team. Productivity can be measured by the number of sprints completed over a given period of time.

**Ethical Concerns**

There are often conflicts between software engineers and those that do not have experience with software engineering because of the title of this essay. Measuring software engineering is inherently difficult. Many people with no experience of programming may be tempted to use a simple metric such as lines of code and force all developers to write as much as possible which can have a major impact on the psychological health of the developers and would surely create a buggy mess of a trash heap codebase that nobody would be interested in touching.

Data privacy is also a concern when it comes to the measurement of the scientific process. Should companies and managers simply be allowed to view all of a developers data from github. It seems likely that a company could try to pressure a developer into making all of his/her repositories and contributions public under fear of being let go which raises serious concerns about individual data privacy.

Big tech companies are known for working their employees hard. They create big campuses to ensure that the devs have all of their needs taken care of by giving them fresh food and then squeezing them like a fresh sponge. Burnout is a very common problem for many of the big tech companies and I think regulators should start looking at this more seriously as it doesn’t receive much attention in the media.

**Conclusion**

Measuring software engineering is difficult if not nearly impossible. There are many metrics and perhaps in the future we will have machine learning algorithms that can take all the data of a developer and spit out exactly how productive they will be expected to be. In the age of big data this could be a very real possiblity with some serious ramifications. I don’t plan on working in software engineering but the quantification of the process made for interesting reading for me.

**Bibliography**

1:

2:

3:

**References**

50: <https://www.quora.com/What-is-the-average-ratio-of-bugs-to-a-line-of-code>

51: <https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/index.html>

52: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.370.9611&rep=rep1&type=pdf>

53: <https://help.gitprime.com/general/what-is-gitprime-exactly>

54: https://en.wikipedia.org/wiki/Agile\_software\_development