Measuring Engineering Report

1 Introduction

Software engineering has come a long way over the last number of decades. It can argued that they can be divided into a number of eras. Stephen H. Kan states in his paper "Metrics and Models in Software Quality Engineering" that "the 1960s and earlier could be viewed as the functional era, the 1970s the schedule era, the 1980s the cost era, and the 1990s and beyond the quality and efficiency era". As the current era is certainly focused in quality and efficiency, the need for measurement in software engineering is growing, as consumers demand high quality while suppliers prioritise low costs at max productivity. This report will be looking at the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available and the ethics concerns surrounding this kind of analytics.

2 Examples of Measurable Data

Ian Sommer states in his book that "Software measurement is concerned with deriving a numeric value for an attribute of a software product or process". He also describes a software metric as "any type of measurement which relates to software system, process or related documentation" such as lines of code in a program, number of person/days required to develop a component etc. We will be looking at some of these as well as the beneficial information that they give from their measurement.

Source lines of code:

Arguably the simplest and easiest to understand metric is source lines of code (SLOC). This is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. SLOC is considered quite controversial in terms of its use as a measure. In one sense, SLOC can be a measure of effort, as it has been shown that the more source lines of code, the more time it took for the program to be developed. On the other hand, SLOC is a poor measure of functionality, as highly skilled could potentially create a program with far less lines of code than an inexperienced developer can of a program with

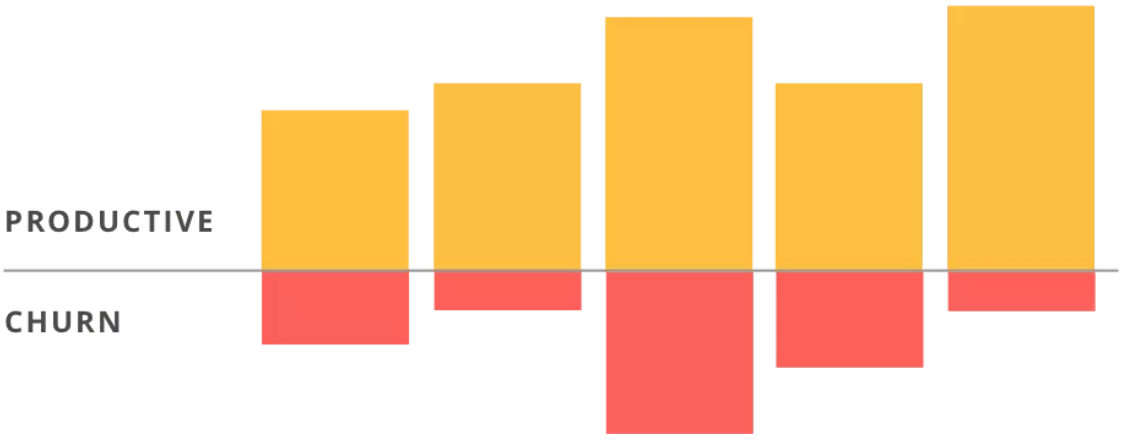
the same functionality. Thus, SLOC is a poor and unreliable measure of an individual's productivity. So despite SLOC being straightforward to measure with the use of automated tools, it is generally not used as a form of measure to the reasons stated above.

Code Coverage:

According to wikipedia, "**code coverage** is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs". A program with high code coverage means more of its lines of code were ran and tested compared to a program with less code coverage. This lowers the chance of undetected program bugs being present compared to a program with low code coverage. Code coverage is a good metric in how much testing one is doing in a software process.

Code Churn:

Code churn is defined as the lines of code added, deleted or modified by a developer over a short period of time such as a few weeks. Code churn is generally measured in lines of code (LOC). Code churn is useful for software managers as it allows them to control the software development process, particularly the quality and productivity. For example, if a developer is continuously altering code based on the same task over a repeated period of time, it will show up as code churn. This indicates that the developer is not progressing at a forward rate and thus slowing down productivity. Measuring and assessing code churn helps identify the problems that an individual developer or software engineer is facing. If the rate is high, then that developer is facing difficulties with their current task and is need of assistance or it may also mean that they are under-engaged. Thus, more time is allocated to resolving these issues which in turn increases overall productivity and improves the main software engineering process.



Complexity:

Wikipedia refers to software complexity as a term that "encompasses numerous properties of a piece of software, all of which affect internal interactions". As the number of entities increase, the number of interactions between increases exponentially to the point where it is difficult or near impossible to know which interactions are taking place between which entities. This can lead to unintentionally interfering with these entities which in turn increases the chance of defects occurring while making these changes.

The measurement of complexity has proven to be quite useful in the software engineering process. Complexity measures has allowed software engineers to limit complexity of their software during development and to prioritise easy to understand systems which in turn leads to lower chances of accidental defects. The need for maintainable and clean code has become a staple throughout the history of the software engineering process. The two most commonly used complexity metrics are McCabe's cyclomatic complexity and the Halstead complexity measures, which will be discussed in greater detail later on.

Individual Efficiency:

The efficiency of a software engineer or developer is the percentage of the engineer's contributed code which is productive. The higher the efficiency of a software engineer is, the more valuable the engineer is to the overall software process. Efficiency correlates to code churn, as the higher code churn corresponds to a lower efficiency rate of an engineer.

The measurement of a software engineer's efficiency is important as it helps identifies difficulties the engineer. This allows for an investigation into the cause which can help allocating or assigning the engineer to a role which help boost the productivity and efficiency of the overall software process.

Defect removal efficiency

This is simply the percentage of defects removed prior to shipping or release of the software product. It is calculated as a ratio of defects found to number of defects resolved prior and at the moment of release. The measuring of this data has proven to be extremely useful in the software engineering process as they can be used to do root cause analysis such as examining why certain bugs were present at release and what changes can be made to help reduce these types of occurrences in the

future. Doing this can help drastically reduce the number of bugs shipped which in turn leads to greater defect removal efficiency.

3 Computational Platforms Available

This section will be explaining the computational platforms available to measure the data discussed in the last section.

Source lines of code:

Despite being mentioned earlier that SLOC is a inconsistent metric in practice, it remains popular among software teams and thus there are computational platforms available to measure them.

There are two major types of SLOC measures: physical SLOC(LOC) and logical SLOC (LLOC). Physical SLOC is a count of lines in the text of the program's source code excluding comment lines while Logical SLOC measures the number of “statements”. Therefore it is much easier to create tools that measure LOC compared to LLOC. However, physical SLOC is more sensitive to logically irrelevant formatting and styling conventions, while logical SLOC less so. This reinforces the point of source lines of code being a poor measure, as physical SLOC can often be significantly different to logical SLOC.

There are multiple platforms that can be used to measure source lines of code, with many tools being found in quick searches online. One well known platform that can undertake this task is Github, which displays the number of lines as well as the number of lines of code for each file in a repository.

While measuring SLOC is no challenge in itself, the main problem has been identifying and communicating the attributes that describe what this measure represents.

Code Coverage:

Code Coverage is a measurement of how many lines/blocks/arcs of your code are executed while the automated tests are running.
“Code coverage is collected by using a specialized tool to instrument the binaries to add tracing calls and run a full set of automated tests against the instrumented product. A good tool will give you not only the percentage of the code that is

executed, but also will allow you to drill into the data and see exactly which lines of code were executed during particular test."

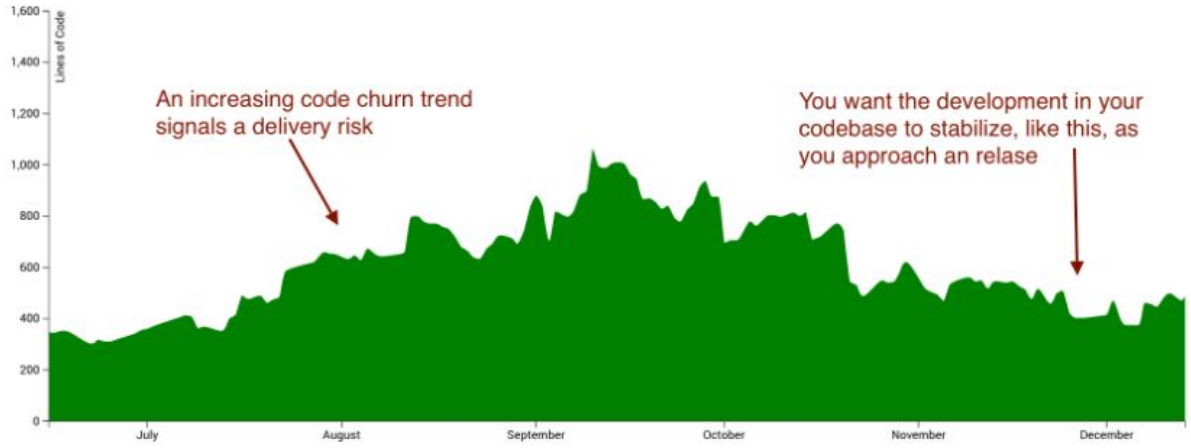
- Stackoverflow link in Sources

Code coverage is similar to that of source lines of code in the fact that there are multiple different tools/platforms available in order to perform this measurement. Different code coverage tools are generally specific to one language. Popular code coverage tools include 'Emma', one of the oldest code coverage tools, designed specifically for Java. Others include CoberTura, CodeCover etc.

Code Churn:

Code churn is returned by displays lines that have been added, deleted or modified in a file over a short period of time. The use of such data can be used to measure the performance of an individual software engineer or developer and the difficulties he/she may be facing. Code churn can also be used to estimate the efficiency of a software engineer as high code churn tends to correlate to low efficiency.

Unlike the previous two metrics, the computational platforms that can be used to calculate code churn is scarce in comparison despite its usefulness in relation to the software engineering process. However there are still a small number of tools available. One example is by using visual studio. There is code churn perspective available on the SQL Server Analysis Services cube for Visual studio team foundation service. This perspective allows measures, dimensions, and attributes that are associated with the changes in lines of codes to be viewed by the user. The perspective can also allow reports to be written on questions such as "how many files with a specific filename extension changed in a particular build" or "what changesets have been submitted and what are the details of each change". CodeScene also provides a platform to measure code churn. CodeScene provides two different metrics: the number of lines added and the number of lines deleted. It constructs a graph showing the rolling average of code churn using these two metrics. This graph can help identify trends within the software engineering process and possibly prevent any larger problems from occurring by studying these trends.



Defect removal efficiency

Defect removal efficiency is usually calculated using defect tracking system. This system should track:

- affected version
- version of software in which the defect was found
- release date

The DRE is then calculated using the following equation:
Number of defects resolved by the development team / total number of defects at the moment of measurement.

There are number of defect tracking systems available for use. Common systems such Jira, RollBar and FogzBugz are used by software teams all over the globe.

4 Algorithmic Approaches Available

As well as computational platforms, there is also a number of algorithmic approaches available which have been to used to measure data in the software engineering process. Among the metrics discussed earlier, complexity is one which relies on different algorithms to measure this data, namely cyclomatic complexity and Halstead's complexity measure.

Cyclomatic Complexity

Cyclomatic complexity is used to indicate the complexity of a program. It does this by quantitatively measuring the number of linear independent paths through a

program's source code. Comparing two programs of the same size, the one with more decision making statements will be considered more complex as more jumps occur in this program.

Cyclomatic complexity is calculated using a control flow graph of the program. Each graph shows several nodes and nodes and edges. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program. Nodes represent processing tasks while edges represent control flow between the nodes. Cyclomatic complexity is then computed mathematically using the following formula:

(G) = E - N + 2
Where,
E - Number of edges
N - Number of Nodes
V (G) = P + 1
Where P = Number of predicate nodes (node that contains condition).

Using tools such as OCLint and devMetrics to carry out this work, cyclomatic complexity can prove to be very useful by helping software engineers limit complexity during development and in software testing.

Halstead Complexity Measures

In 1977, Maurice Howard Halstead introduced metrics to measure software complexity. Halstead observed that metrics of a software should reflect the implementation or expression of algorithms in different languages but be independent of their execution on a specific platform. These metrics are computed directly from the code in a static manner.

Halstead's metrics are as follows:

Parameters:

- n1 = number of distinct operators
- n2 = number of distinct operands
- N1 = total number of operators
- N2 = total number of operands

Several measures can be calculated using these figures:

Program vocabulary: n = n1 + n2

Program length N = N1 + N2
Volume V = N * log2n
Difficulty D = n1/2 * N2/n2
Effort E = V * D
Errors B = V/3000
Testing Time T = E/18

5 Ethics Concerns

Despite the growing trend of software engineering measurement in recent times, questions are still being asked over its ethical implications, which will be looked at in this section.

Over reliance of measurable data

The overuse and reliance of measurable data is worrying to some. While measurable data should be used in conjunction with other entities to measure the software engineering process, its use as the sole factor in judging for example a sole engineer's performance in a software development process.

Poor usage of measurable data

Even though some data measured may seem useful for analysis of the current software engineering process, its misuse can be a cause for concern. An example of this is the source lines of code metric. As mentioned earlier, source lines of code are generally used to estimate the performance of an individual software engineer or developer. While it is not disputed that effort is highly correlated with SLOC, it can become a problem when using the metric to value the individual developer as a whole. An example of this is determining how productive a software developer is based on how many lines of code they have produced. This can be an issue as source lines of code does not directly correlate with the productivity value of a developer. A skilled developer has the ability to create a program with less lines of code than an inexperienced developer can with the same program functionality. This could lead to ethical issues such unfair dismissal if the project lead or manager feels an engineer is no longer due to their supposed low productivity when this may not be the case.

Privacy

In recent times, employers have been experimenting in different ways to measure their employees productivity and performance. In addition to the methods explained throughout this report, wearable technology has risen in popularity over the last few years. A notable player in this is a start-up based in Boston, Massachusetts who go by the name 'Humanyze'. Humanyze supplies companies with employee ID badges loaded with inbuilt biometric measuring capabilities. The badges are able to track everything from movement within the office and interactions in the office to simple voice conversations, measuring conversation length and level of voice tone. Humanyze have stated these analytics can be used to help give employers better information in regards to evaluating performance.

However, with the increased methods for companies to track employees at their disposal, more concerns have been raised about privacy protection. Some say with the increased introduction of such methods, more employees will experience increased vulnerability. In response, companies have assured the sole purpose of such devices is to improve efficiency, protection and productivity. Despite this, questions will still be asked about the usage of such methods and measurements

6 Conclusion

In conclusion, it is important to realise that in this day and age that the use of measurements in the software engineering process has expanded. With increased emphasis on software quality and productivity in recent times, the need and analysis of these measurements has proven vital in the software industry. As explained in this report there are many different types of data that can be collected using many different platforms and algorithms. With good analysis of this data, the software engineering field will continue to improve and grow in the coming years.

Sources

Stephen H. Kan. 2002. Metrics and Models in Software Quality Engineering (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

https://www.slideshare.net/software-engineering-book/ch24-quality-management.7
http://ecomputernotes.com/software-engineering/software-measurement
https://en.wikiversity.org/wiki/Software_metrics_and_measurement
https://dl.acm.org/citation.cfm?id=559784
https://en.wikipedia.org/wiki/Source_lines_of_code
https://stackify.com/track-software-metrics/
https://blog.gitprime.com/5-developer-metrics-every-software-manager-should-care-about/
https://www.tutorialspoint.com/software_engineering/software_design_complexity.htm
https://en.wikipedia.org/wiki/Programming_complexity
http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf
https://stackoverflow.com/questions/195008/what-is-code-coverage-and-how-do-you-measure-it
https://en.wikipedia.org/wiki/Cyclomatic_complexity
https://www.embedded.com/electronics-blogs/break-points/4419386/Metrics-We-Need
https://codescene.io/docs/guides/technical/code-churn.html
https://docs.microsoft.com/en-us/vsts/report/sql-reports/perspective-code-analyze-report-code-churn-coverage
https://swtestingconcepts.wordpress.com/test-metrics/defect-removal-efficiency/
https://www.guru99.com/cyclomatic-complexity.html
https://en.wikipedia.org/wiki/Halstead_complexity_measures
http://www.bbc.com/capital/story/20170613-the-tech-that-tracks-your-movements-at-work