~Measuring Software Engineering~

Declan J Roberts
18327644
Robertd4

~ Introduction ~

Measuring software engineering is akin to attempting to measure human productivity as a whole, a vastly researched topic that spans many decades of scientific literature and real world solutions designed to improve the quality and speed at which work is performed. This eponymous pandect will attempt to explore the act of measuring software engineering, the correlating factors altering it and possibilities of improving it, as well as some of the principles that govern it. Productivity as a metric is something that cannot be easily quantified, is it a quality that can be inculcated? Or possibly an inherent human quality? Exigent as it may be to evaluate such an incorporeal value there are many ways that we are attempting to do just that in many corporations today. This essay will attempt to explore the overwhelming array of information regarding human productivity, measuring it and then the specific evaluation of software engineering.

~ Measurement & Metrics ~

In the famous book by Paul Krugman 'The Age of Diminishing Expectations' He stated "Productivity isn't everything, but in the long run it is almost everything. A country's ability to improve its standard of living over time depends almost entirely on its ability to raise its output per worker". This epigram holds true for businesses as well as countries.

~ Labour Productivity ~

The simplest way to measure raw productivity is known as Labour productivity, expounded clearly by the output volume divided by the labour input use. This metric is used especially heavily in macroeconomics but is also extremely common in almost all sectors of business to a certain extent.

$$Labour\ Productivity = \frac{Output\ Volume}{Labour\ Input\ Use}$$

~ Multifactor Productivity ~

There is a metric however that supersedes Labour Productivity, Multifactor Productivity, it is a much more concise metric of actual human productivity than Labour productivity but requires a much larger pool of information to calculate. For example, if Labour productivity uses the SLOC (source lines of code) metric as the output volume, and hours as the Labour input use, then a Multi-Factor Productivity approach would take into consideration the complexity of the code written, the ugs in the code, the code coverage etc, this in turn creates a much more reliable systematic approach to productivity.

~ Productivity Index ~

Knowing the workplace productivity of your workers is one thing but there is no evaluation without context, the productivity index is the ratio of productivity during one period to another. If the average labour productivity of your workers was 2 during the first quarter of a year and you implement changes within the organisation to attempt to increase productivity during Q2, and the labour productivity increases to 2.2, then your productivity index will be 2.2/2 = 1.1, hence your productivity has increased. Tracking this metric over time will allow businesses to accurately assess if changes in the workplace affect the productivity of employees as a whole.

~ Realistic Metrics ~

This subsection is titled as such due to the fact there are many metrics that could be extremely useful to obtain but their collection would either be too complicated, too obtrusive, ethically ambiguous, or their use not appreciably noticeable. Some of the standard metrics obtained by many large corporations today are as follows:

- Number of Commits
- Frequency of Commits
- SLOC (source lines of code)
- Cycle time
- Open / close rates
- Sprint Burndown
- Bug Rates
- Code Coverage
- Lead Time
- Mean Time To Repair
- Mean Time Between Failures

All of the listed metrics are useful to obtain and do not influence the developers productivity negatively, hence that is why they are the most common ones in use today. Some, like Sprint burndown are more niche metrics that are less globally applicable and require certain predetermined circumstances to obtain. It is a product of a certain system intended to increase productivity as a whole, an Agile framework known as scrum and will be discussed later.

~ Issues and solutions to certain metrics ~

Number & Frequency of Commits - This metric can be useful in determining the consistency of a given developer but isn't wholly useful by itself as, for example a programmer could simply add a comment and commit it.

SLOC - When this metric was first introduced there were obvious problems with how it could be easily exploited, the easiest way to elucidate this is with an example such as he following:

Program (i):

```
for(int i = 0; i<100; i++){ System.out.println(i);}
```

Program (ii):

```
for(int i = 0; i<100; i++)
{
System.out.println(i);
}
```

These two extremely simple java loops do the exact same thing, yet one is 4 times the length of the other, hence this metric needs considerable refinement in its base form. LLOC, or Logical Lines Of Code is a metric that allows for much more accurate tracking of ‘the work done by the code’ as it were. Using the LLOC metric these code snippets would be given the same score.

~ Measuring Complexity ~

~ Why measure code complexity ~

The motivations for measuring the complexity of programmes become immediately apparent once you realise the inherent downfalls of the other metrics used to evaluate the productivity of a worker or attempt to measure the amount of appreciable work that has been done on a given task. Using the more accurate LLOC approach to the amount of code written we still do not know how difficult that piece of programming was to write, it may have been long and required many instructions, but it could have also been extremely simple, hence we should also measure the complexity of the code written.

~ Cyclomatic Complexity ~

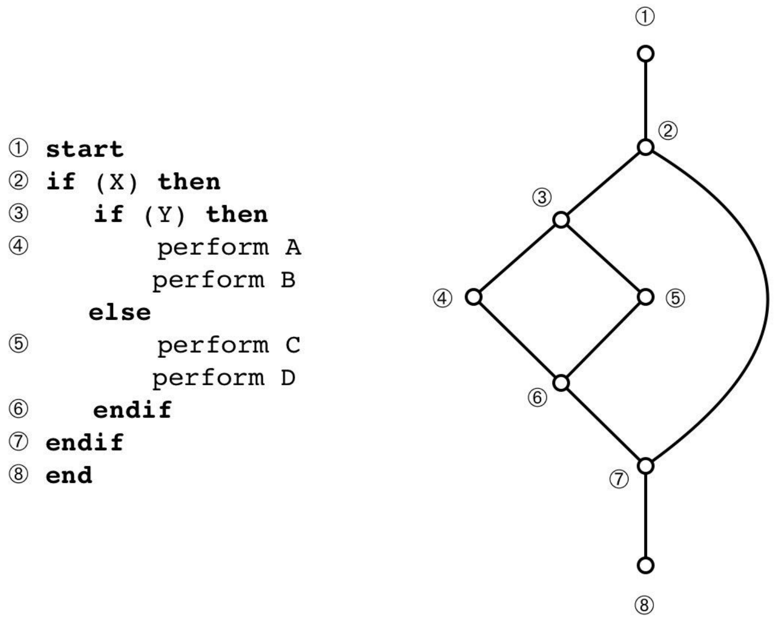
The primary method used to evaluate this metric is cyclomatic complexity; it is defined as a measure of the number of linearly independent paths in any given piece of software. It is often used to define the 'quality' of the code, but even this can be misleading as overly complex programmes won't necessarily be better than more simple ons, as a general standard however, it holds up well.

The method for deriving cyclomatic complexity is based upon the control flow graph of the programme and is calculated as follows :

$$M = E - N + 2P$$

Whereby
M = Cyclomatic Complexity
E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

In this simple diagram we can see how cyclomatic complexity would be assessed
Nodes = 8, edges = 9, 1 component. Therefore M = 9 - 8 + (2x1) = 3



```
① start
② if (X) then
③    if (Y) then
④         perform A
          perform B
      else
⑤         perform C
          perform D
⑥    endif
⑦ endif
⑧ end
```

~ Dynamic Metrics ~

A Large portion of Dynamic measurement of software engineering comes in the form of different aspects of code testing as this is generally the form of measurement that impacts the end user of the product the most.

~ Branch Coverage ~

Branch Coverage testing aims to ensure that each one of the possible branches from each decision point is executed at least once during the use of the software, thus attempting to ensure that all reachable code is executable.

Usually this is done by attempting to access all branches of a given piece of software and setting their access ability to true once it has been executed. This method of testing is generally useful to find unused or unnecessary code within a system and can help find bugs by displaying unavailable branches and elucidating why some problems may be occurring.

~ Statement Coverage ~

Statement coverage while being similar to branch coverage in its goal is an awful lot simpler to implement due to it being a much more surface level coverage metric. It is the primary method used to test software coverage today due to its simplicity to implement.

It falls short of branch coverage however when to comes to traceability of errors due to its simplicity, with branch coverage it is possibly to easily see where a section of code becomes unreachable from, with statement coverage this is not the case as it doesn't take into consideration the order of execution of the programme and thus the root cause of the issue is far less traceable than it would be with a branch coverage testing technique.

~ Measurement Software ~

~ HackyStat ~

While no longer undergoing development Hackystat is widely available to anyone who wishes to implement it on repositories such as github or bitbucket. It allows unobtrusive and efficient collection of software metrics from individual programmers. It is a framework of tools and makes use of google's chart services and also uses twitter in some aspects for data transfer.

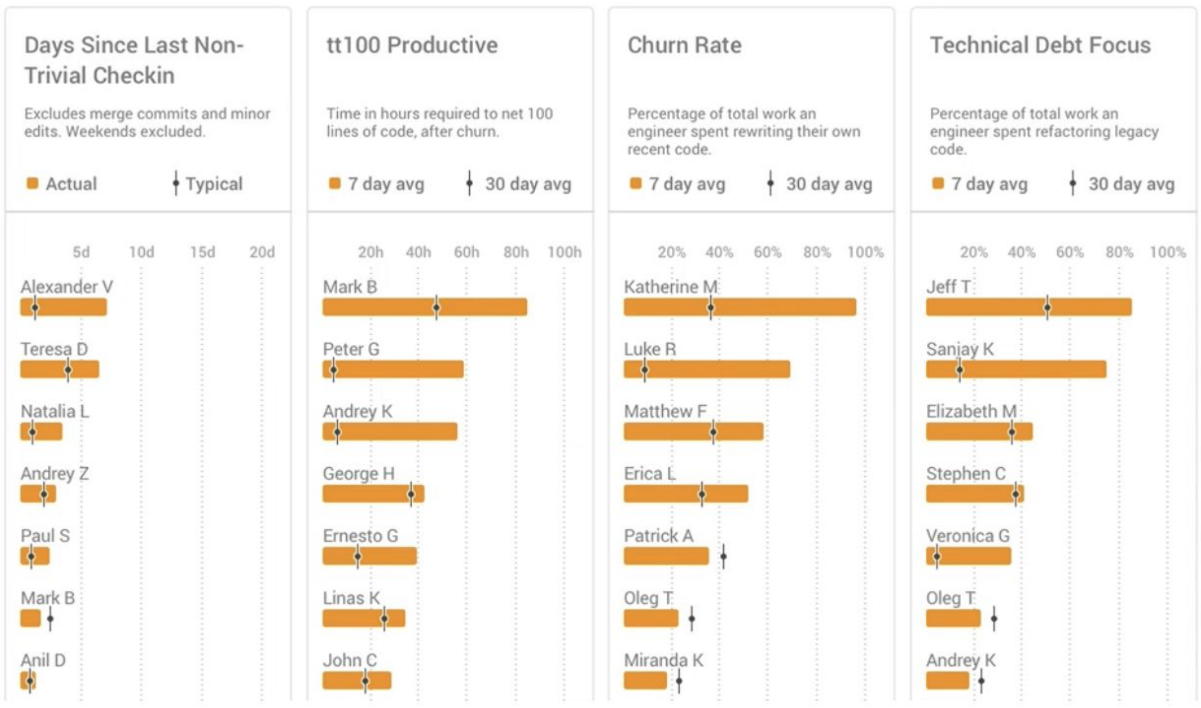
Hackystat isn’t tied to any particular development environment making it extremely malleable for many different data collection applications. It can collect server and client side data and analyze it without the need for additional work, this makes it very appealing as a framework as there is no cost to use hackystat and little to no labour required to process the data once it is collected.

Far from being a platform solely for singular data collection it can also collect metrics from a team of programmers working on the sme project to accurately assess the productivity of the team as a whole and not just on an individual basis.

~ GitPrime ~

GitPrime is a software measurement tool that heavily relies on version control for its assessment metrics, hence the name. It is specifically targeted for use around teams and projects as it allows you to aggregate and view contribution and reviewing statistics extremely easily and concisely.

It's main draw is perhaps the polish it presents all of this data with, the user interface and ease of operation is a major selling point for GitPrime. The statistics are well laid out and very comprehensive, offering a multitude of metrics for developers who are committing code and a different set for those reviewing it.



The metrics that the contributors are assessed on with GitPrime include Average Time to Resolve, Recent Ticket Activity Level, Average Number of Follow-on Commits, Coding Days, Commits Per Day, Impact and Efficiency. The standards used to assess reviewers are Responsiveness, Comments Addressed, and Receptiveness among others.

~ Beyond Measurement ~

Collecting mountains of data and using it to accurately measure software engineering is intrinsically pointless if you have no way of putting that information to use, in this section I will elucidate the ways in which companies are using this gathered data to make a marked impact on different aspects of the industry.

~ Prediction of future projects ~

The International Software Benchmarking Standards Group (ISBSG) has garnered a repository of various metrics from a total of over 9000 projects that have been submitted to them over the years. They use this data for many things but one of the more unique uses of this data is the estimations of different projects. They use previously gathered metrics from the set of developers that are working on the proposed projects, such as years programming, specialities, previous productivity metrics as well, along with project metrics such as the estimated complexity and scale. This data is then compared to their repository to estimate the total project effort that is going to be required.

This system allows for far more accurate time and cost predictions to be given to clients of software development companies, or just for in house development so that they do not over or underestimate the total scale of the project. In the case of over-estimation, the problem of under utilized resources and hence an unnecessary increase in cost and project duration are the issues. If the project is under-estimated however there will be 'unexpected delays', unrealistic deadlines and quality issues with the final product.

~ Improvement of productivity ~

The single most implemented approach to improving productivity for software development has been the adoption of an agile framework of development. This is a far departure from the more traditional waterfall approach that has been in use since the very inception of software engineering as a discipline. It relies upon taking inputs from both the team of developers and the customers to dynamically alter the goals.

Agile is built on its development structure, this involves splitting the work into short timeframes, usually known as sprints, after each one the project is reevaluated by both the team and the stakeholders. This allows for much more regular updates and for a more cohesive and strict approach to development. The strict schedule also allows for the project not to stray from the requirements as much as would occur with a traditional approach.

An even Stricter application of the agile philosophy isa design structure known as scrum. The work is divided into segments that can be completed within one interval (Sprint), this interval is almost always 2 weeks in length but there is mild variation depending on the project. Short scrums are held every day to constantly assess progress and reassess the build specifications. At the conclusion of each sprint a review is held to fully evaluate the progress made by the project team, usually this also entails demonstrating the advancements made to the stakeholders. The meeting concludes once the goals for the next sprint have been reached.

The most commonly used metric to assess the productivity of workers within these sprints are often titled story points, these 'story points' are meant to represent a standard unit of work, so essentially the more story points that a developer can accomplish the more productive they have been in that last sprint.

~ Ethical Considerations ~

~ Irresponsible Data collection ~

Over seventy percent of employees say that they would only be willing to share personal data with their employer under the circumstance that the firm be transparent about exactly how and why this data is being collected, and only if that data will make an ascertainable improvement to the workplace. Unfortunately in the United States in particular there are no federal laws governing the collecting of employee data, with the only states having workplace data collection laws being Delaware and connecticut.

This insight becomes even more damning when you take into consideration the fact that when surveyed 56 percent of business leaders said that their companies don't ask for consent to collect worker data. Data collection is already posing as, in my estimation, a concern for a large overstepping of bounds between employer/employee confidentiality without a large majority of people even knowing that they are being monitored.

~ Data security ~

Database breach and leaks occur every day, so an obvious ethical concern is : is the data being collected going to get leaked? While we can never know if a database will be b reached until it has been, it is imperative that the least corporations can do is attempt to make the data they are collecting on their employees as secure as possible.

Even if the data collected seems benign and of no use to anyone outside of the company it may be found out that it is possible to use the procured information to exploit the people the data is about. Companies must encrypt their databases and make a conscious effort to not collect data on their employees that could ever be used maliciously, we really don't know if we can trust them to do that, especially at the moment.

The cambridge analytica scandal has roused a rightful suspicion of big data in the minds of many, and i would consider software engineers some of the most skeptical people of big data due to their knowledge of exactly how easy it is to collect swaths of information. Even if the suspicion surrounding the data being used ethically is an ungrounded fallacy it will make workers more distrusting and make them feel less safe in the corporation they work for, ultimately leading to a loss in productivity and possibly more issues.

~ Possibility of toxic Workplaces ~

This possibility lies upon the predicate that, for some reason, these figures be made available to the entirety of the workforce as a measuring tool, often in companies there is an “employee of the month”, for example a salesperson may sell more than everyone else and he is afforded that title. I can envision a misplaced attempt at improving worker productivity by venturing to implement a similar system in software engineering. Unlike a sales position, software engineering is predominantly a collaborative profession, so this may result in an extremely toxic work environment, with people exploiting the metrics, and sabotaging coworkers code.

~ Conclusion ~

Despite the ethical concerns of collecting data on software engineers I believe that collection can be an extremely useful tool, along with many more in the arsenal to allow for a much more productive and statistically accurate environment in which developers can practice.

In the field of general worker productivity when it comes to software engineering it seems there is quite a lot of ways to measure it and they don't quite cut it when it comes to accurately examining the value that someones code has, even when taking all of the metrics into consideration, there's nothing that can tell us the exact value that a programmer has produced. That's what it's about really, a software engineer may be extremely productive and write lots of complex code and be seeming to do a great job, but maybe a better one could have written a lot less simple code to accomplish the same goal? Productivity and skill are completely different things, and until we can write code that accurately measures both, we won't be any closer than we are today to measuring the value of someone's contributions, which really is the holy grail of measuring software engineers.

Software measurement as a discipline is one that will be constantly evolving as the technological world progresses, in ten years even, there will be so many new developments in the field that we will need to find new ways to test them, and thus new tools and techniques will be developed for that very purpose. The good thing is that code is an awful lot simpler than humans to test, so we have the ability to assess code that we write now, with tests that we write now, and I can assume it will be the exact same way as time progresses.

References

https://www.castsoftware.com/discover-cast/
https://sdtimes.com/metrics/integration-watch-using-metrics-effectively/
https://medium.com/data-science-group-iitr/association-rule-mining-decipheredd81
8f1215b06/
www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php/
https://www.semanticscholar.org/paper/Automated-Software-Engineering-Process-
Assessment%3A-Grambow-Oberhauser/
https://www.gitprime.com/platform/code/
https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/
https://digitalguardian.com/blog/history-data-breaches/