Measuring Software Engineering

Cathal Byrne: 18328661

Introduction

Managing a group of software engineers is an onerous task. Like any position where you oversee a task to completion, a high quality of work is desirable and the ability to measure your engineers' contributions to the project is paramount. In a discipline like software engineering, debate has ensued for decades on the efficacy of certain methods and millions of dollars spent on improving the way we measure the quality of code. A quick google search will provide all the conflicting answers needed to show that we are far from a consensus on how best to go about measuring modern day "knowledge workers".

There are several metrics used in the measurement of software engineering, but first off, we must define exactly what a software metric is. Norman Fenton defines software metrics as *"a collective term used to describe the very wide range of activities concerned with measurement in software engineering… that help predict software source requirement and software quality"* (Alexandre, 2002). This essentially tells us that a software metric is a piece of data taken during development than can be used to measure productivity.

In the following report I will attempt to outline why this task of measuring software engineering is a much more difficult task than you might initially think. I will also explore the methods that have been used historically and how they have improved to leave us with the modern techniques and technologies used in the field today.

Can We Even Measure Productivity?

"How can we measure the productivity of employees whose tasks are not fixed, have no production standard times, and whose tasks can be performed differently by among the various workers? … Currently there are no accepted methods to measure knowledge worker productivity, or even accepted categories" (Ramirez, Y.W. 2004). Ask yourself how you would measure the productivity for complex roles like that of an accountant or a surgeon. Less complex roles, like say a bricklayer have far clearer ways to measure their

productivity i.e. bricks laid compared to other bricklayers or an expected output. If you adopt this quantity-based approach to software engineering, you can come up with certain metrics that give some indication of performance. Some of these metrics include Lines of Code (Oliveira et al, 2017), as well as other one-dimensional metrics like tickets closed and commits.

Source Lines of Code:

Source Lines of Code (SLOC), alternatively referred to as Lines of Code (LOC) is about as simple and one-dimensional a yardstick for productivity as exists. As the name implies, SLOC counts each line of source code written by a software engineer. As anyone with a mild knowledge of software engineering will tell you, this metric is archaic and is easily gamed. If you asked 3 software engineers to produce a program that would print the numbers 1 through 10, you could conceivably get 3 very different programs:

```
public class report{
    public static void main(String[] args){
        //Program 1
        System.out.println("1");
        System.out.println("2");
        System.out.println("3");
        System.out.println("4");
        System.out.println("5");
        System.out.println("6");
        System.out.println("7");
        System.out.println("8");
        System.out.println("9");
        System.out.println("10");

        //Program 2
        int i;
        for(i=1;(i<11);i++){
            System.out.println(i);
        }

        //Program 3
        for(int j = 1;(j<11);j++)System.out.println(j);
    }
}
```

Using the SLOC approach to measure code, program 1 is 2.5x better than program 2, which is 4x better than program 3. This is the clear drawback of using SLOC as a metric. If a project manager were to employ software engineers based solely off this metric, you'd see code start to be written like

Program 1 industrially. A logical improvement to using SLOC is adopting the aptly named Logical Lines of Code (LLOC). Nguyen (2007) explains that logical source lines of code essentially counts each logical code statement or step instead of each individual line. In this instance programs 2 and 3 would be seen as superior and would both have a Logical Line of Code count of 2, despite program 2 containing 4 times as many lines. This is definitely an improvement on pure SLOC, but it is not the gold standard, as Nguyen states that whilst it bypasses artificially lengthening code *"its imprecise definition has been the source of contention among tool developers."* (Nguyen, 2007)

As programs and languages advance and new techniques become in vogue, and the rate of development of frameworks and packages increase (Kapanoglu, 2020) it becomes increasingly difficult to accurately gauge the quality of software engineer you have at your disposal.

Tickets Raised/Closed:

An alternative way to measure the work of a given software engineer is to look at the amount of tickets they've cleared over a project's development period. One ticket cleared = one positive task completed. Again, the problem with one dimensional metrics exist in that this can easily be cheated, either by creating smaller and more achievable tasks to dope your numbers or by actively prioritising clearing small tickets over the advancement of the project.

This method is not to be discarded however, as correct implementation of tickets is an excellent yardstick to determine progress. *"Measuring tickets is an excellent metric if the tasks are written well and assigned based on business priority. When more tickets get closed, more good things are happening with the project."* (Osbourn, 2019). This goes a long way towards giving a good indication of work being done but is still not bulletproof as every job brings its own complexity. *"Not all tickets are equal, which is why many teams opt to score tickets by effort."* This is clearly a much better way of doing things but now you leave the realm of one-dimensional metrics and grading "effort" is much more difficult.

Commits:

Committing code is how we as software engineers add work to a codebase. In theory it's easy to conceptualise why counting commits could be used to measure performance, programmers who commit more do more work right? Once again, we fall into the trap of having an easily cheated metric, the correlation between high level of commits and productivity is nowhere near 1.

An article by Job van der Voort outlines that commits are definitely useful, but cannot be used in a vacuum. "*The best way to get an idea of someone's progress is to look at both their communication and activity. An effective engineer will either be making commits to further their progress or communicate in some way about their work."* (Van der Voort, 2016). This is a much better use of commits as a measurement, as it looks at what's important (the improvement of the project) as opposed to an effectively arbitrary number.

Why Do We Need to Measure Productivity?

The next question that needs asking is "Why are we doing this?" The simple answer is to do so is beneficial for business, and as such we must pair the measurement of software engineering with the achieving of business goals. Throughout my research I kept coming across articles from Stephen Lowe, the Product Technology Manager at Google and he makes the point in a 2016 article that metrics *"Don't matter in software development (unless you pair them with business goals)"* (Lowe, 2016). He is of course correct, as I've outlined above, this data that can easily be gathered about specific engineers is useless until given context. One great thing about software engineering is that the methods used to deliver a project often provide measurable metrics themselves; an example I'll use is agile. Agile is by far the most common method for software engineering today and according to Lowe (2016), the prime metrics associated with agile are *"lead time, cycle time, team velocity and open/close rates"*

Lead Time:

Lowe defines lead time as *"How long it takes to go from idea to delivered software".* Having the ability to millions of lines of working code means nothing if there's no end product, and agile teams always strive to lower their lead time. Lowe recommends doing this by being more responsive to your customers, reducing wait time and simplifying decision-making processes.

Cycle Time:

Cycle time is a measurement of how long it takes your team to implement a change in your software system and deliver that change into production. In agile development where you can have multiple deployments and a minimum viable product is often sought after in the beginning, having a low cycle time is paramount. It's desirable for some teams to implement continuous delivery, a methodology of engineering whereby your cycle time is often measured in minutes or even seconds instead of months.

Team Velocity:

Velocity in agile development is one of if not the most important metrics used to measure performance of a team. It measures *"the amount of work a single team completes during a software development iteration or sprint."* (Lines, 2020). It can be expressed formulaically as:

Vi = Units of Effort Completed/Sprint Time

Team Velocity should mainly be used for projecting how long/many more sprints a project will take until completion.

Open/Close Rates:

Open/Close Rates is essentially a continuation of counting tickets. Keeping track of not only how many problems a team is solving but also of how many they are stumbling upon is a useful metric when determining the efficacy of a team of software engineers.

In short, software engineering metrics exist to help speed up the process of developing and delivering software that make a difference in their field and produce results.

Platforms Used

We'll now look at some of the platforms used to gather the data associated with measuring software engineering. Monitoring employee activity during a working period is not a new concept; as Rosenthal (2018) outlines in Accounting for Slavery that quantitative management practices were implemented on West Indian and Southern slave plantations. I've split this section into two main types of methods used: Computational platforms and algorithmic modelling.

Computational Platforms:

The first platform I'll look at regarding computational platforms for measuring software engineering is a platform called LEAP.

LEAP: Originally, the excessive overheads involved with collecting and analysing data was a serious issue. To address this, Jonson et al (2003) talk about *"A toolkit for PSP-style metrics collection and analysis called LEAP"* was created. PSP is an acronym for Personal Software Process, an ideology for monitoring individuals or an individual monitoring themselves throughout the

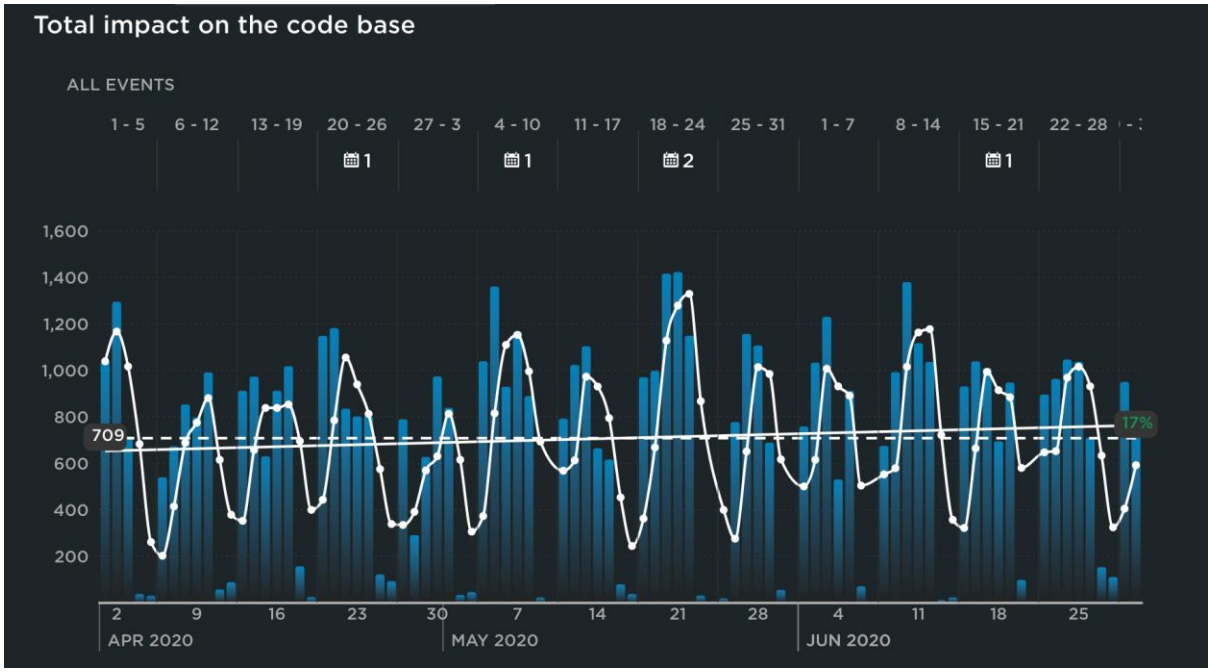
development process. This toolkit suffered low adoption as engineers struggled to change over from whatever software they had been using to this new toolkit. What LEAP did though was to pave the way for future platforms of analysing software engineering.

Hackystat: Hackystat is a framework made to allow engineers collect and analyse PSP data automatically. This cuts out a lot of the changeover issues associated with LEAP. It operates using sensors attached to development tools communicating with a centralised server using the SOAP protocol. Hackystat's structure emphasises privacy and its main function is to gather individual data and its uses. One major benefit is how lightweight it is, as Hackystat stores data in XML files as opposed to a backend database.

PROM: PROM (Pro Metrics) draws similarities to Hackystat but is far more comprehensive, as well as having a larger focus on individual's privacy. It covers data at three different levels, which Siletti et al (2003) define as personal, workgroup and enterprise. It protects and individual's privacy by using individual data in a group model, as such only giving managers an aggregated dataset. It integrates widely used accounting methods such as Activity Based Costing by automatically tracking data for both software engineers as well as managers.

Git and Pluralsight Flow: One of the single most powerful tools for measuring software engineering is source control. Tools like Git allow for the collection of data throughout the team's development period without the need for additional tools or services. Git makes it incredibly easy to access SLOC and commit metrics for example. There are also several platforms that take the power of Git and visualise it with additional tools, these are incredibly common in the workplace if not the industry standard. The most popular one is Pluralsight Flow, formerly Gitprime. These platforms allow for more complex computations and visually display results giving some of the most comprehensive measurements of a software engineer's performance that we can access today.

Pluralsight Flow Example:



Algorithmic Modelling:

This describes the use of relatively simple algorithms to predict a whole host of variables including cost and time. There are several algorithmic models available to calculate cost but the most popular one is COCOMO. The Constructive Cost Model (COCOMO) is a procedural software cost estimation model. The formula makes use of regression and information based off of Lines of Code found in previous projects. The model, developed by Barry Boehm in the 1980s, is based off of two key parameters: Effort and schedule. Effort defines the “amount of labour required to complete a task” and is measured in human months. Schedule is measured as time to completion in

the relevant unit (days, weeks etc).

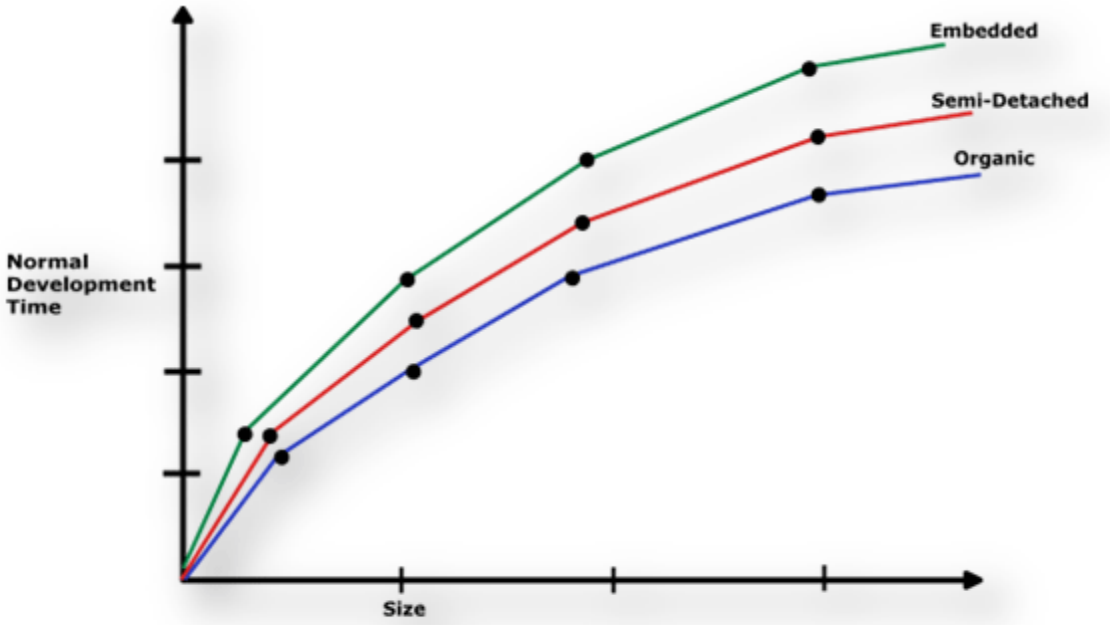


Fig. Development time Verses size

The COCOMO has three levels in increasing accuracy: Basic, Intermediate and Detailed. COCOMO also defines 3 types of systems: Organic, Semi-Detached and Embedded. These systems represent the different level of complexity present in each project. Different formulae exist for each, making the COCOMO and incredibly versatile and useful model for measuring software engineers. It has also been adapted and developed over the years to such a degree making it one of the most used and useful models available today.

Basic COCOMO Model: Formula

The basic COCOMO equation

- $E = a_b (KLOC \text{ or } KDSI)^{b_b}$
- $D = c_b (E)^{d_b}$
- $P = E/D$ where
  - **E** is the effort applied in person-months,
  - **D** is the development time in months,
  - **KLOC** / **KDSI** is the estimated number of delivered lines of code for the project (expressed in thousands)
  - **P** is the number of people required and
  - a_b, b_b, c_b and d_b are coefficients given in next slide.

Ethics

The ethics surrounding the mass collection of data is a hotly debated topic and for good reason, as the 2018 Cambridge Analytica scandal displayed just how commodified data has become and the dangers surrounding this. At the 2019 Web Summit technology conference in Lisbon, Edward Snowden stated that *"The problem isn't data protection; it's data collection"* (Snowden, 2019). Recently the collection of personal data for analysis in marketing has become the norm, and with that comes its own problems. The main issues arising from collection of personal data are the potential loss of personal privacy, as well as the fear of corporations/governments being able to manipulate and influence people with this data. An example of this that went viral is Target correctly predicting that a teenage girl was pregnant in 2012 by analysing her purchases (Business Insider, 2012). Target identified 25 products that when purchased together strongly indicate someone is pregnant. The business benefit of this was that Target could send the girl coupons at an expensive and habit-forming period of her life. The main reason it went viral was because the father saw the coupons being sent and took issue with Target, accusing them of promoting teenage pregnancy, when they in fact knew more about his daughter through data analysis than himself.

On the software engineering side of things, there does exist a Software Engineering Code of Ethics and Professional Practice, formed by Don Gotterbarn, Keith Miller and Simon Rogerson in 1997. This contains 8 principles outlining software engineer's *"commitment to the health, safety and welfare of the public"* (Gotterbarn et al, 1997). These are as follows:

1. PUBLIC – Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.

8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

It is difficult to strike a good balance between gathering information that is beneficial to you as a project manager and to your team, and protecting their integrity and the integrity of the public at large through needless and often predatory data collection.

Conclusion

To conclude, the measurement of software engineering through collection and analysis of data is both a difficult process to get right yet incredibly beneficial when done so. We are now at a stage where data on almost any relevant field to software engineering measurement is readily available. We also have some of the most advanced and efficient platforms and models available to carry out this measurement. Doing so effectively using the correct metrics has been

shown to reduce the product development cycle as well as increase the impact software has in its intended field.

We must also take into account the ethical responsibility that goes measuring software engineering. Once you begin to record information regarding each individual engineer, you must accept to only use it in a context beneficial to both the engineer and the business; the “what” you record must always have a “why”. Not every field has the capability to measure performance to such an extensive degree, and as such this creates incredibly exciting prospects for the advancement of software engineering, but also poses incredible risks to those who are subjected to measurement.

Bibliography:

Alexandre, S., 2002. *Software Metrics An Overview*. [online] Cetic.be. Available at: <https://www.cetic.be/IMG/pdf/Software_Metrics_Overview.pdf> [Accessed 24 November 2020].

Ramírez, Yuri & Nembhard, David. (2004). *Measuring Knowledge Worker Productivity: A Taxonomy.* Journal of Intellectual Capital [Accessed 24 November 2020]

Oliveira, E., Viana, D., Cristo, M. and Conte, T., 2017. *How Have Software Engineering Researchers Been Measuring Software Productivity? A Systematic Mapping Study*. [online] Scitepress.org. Available at: <https://www.scitepress.org/papers/2017/63144/63144.pdf> [Accessed 24 November 2020].

Nguyen, V., Deeds-Ruben, S., Tan, T. and Boehm, B., 2007. *A SLOC Counting Standard*. [online] Citeseerx.ist.psu.edu. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.196&rep=rep1&type=pdf> [Accessed 25 November 2020].

Kapanoglu, S., 2020. *How is computer programming different today than 20 years ago?* [online] <https://medium.com/swlh/how-is-computer-programming-different-today-than-20-years-ago-9d0154d1b6ce> [Accessed 25 November 2020]

van der Voort, Job., 2016. *Commits Do Not Equal Productivity* [online] https://about.gitlab.com/2016/03/08/commits-do-not-equal-productivity/ [Accessed 25 November 2020]

Lowe, Steven A,. 2016 *9 Metrics That Can Make a Difference in Today’s Software Development Teams* [online] https://techbeacon.com/app-dev-

testing/9-metrics-can-make-difference-todays-software-development-teams [Accessed 25 November 2020]

Lines, Dan,. 2020 *Why Agile Velocity Is The Most Dangerous Metric For Software Dev Teams* [online] https://linearb.io/blog/why-agile-velocity-is-the-most-dangerous-metric-for-software-development-teams [accessed 25 November 2020]

Johnson, P.M. & Kou, Hongbing & Agustin, J. & Chan, C. & Moore, Carleton & Miglani, J. & Zhen, Shenyan & Doane, W.E.J.. (2003). *Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined.*

Rosenthal, C., 2018. *Accounting For Slavery*. 1st ed.

Sillitti, Alberto & Janes, Andrea & Succi, Giancarlo & Vernazza, Tullio. (2003). *Collecting, integrating and analyzing software metrics and personal software process data.*

Snowden, Edward., (2019) Web Summit Technology Conference Keynote Speaker.

Lubin, Gus., (2012) *The Incredible Story of How Target Exposed A Teen Girl's Pregnancy* https://www.businessinsider.com/the-incredible-story-of-how-target-exposed-a-teen-girls-pregnancy-2012-2?r=US&IR=T [accessed 27 November 2020]

Gotterbarn, Don. & Miller, Keith & Rogerson, Simon., (1997), *The Software Engineering Code of Ethics and Professional Practice* ACM/IEEE.