The Challenges of Measuring Productivity in Software Engineering

Leo McClean

Trinity College Dublin

# Introduction

The advent of software engineering came at the dawn of the 1960s, when most of the modern world realised its problems could be simplified by embedded systems that efficiently solved large calculations and complex operations involving machinery. For example, with space travel, complex calculations needed to be performed constantly by an on-board computer for a shuttle to correctly exit and re-enter the atmosphere. This ‘software engineering’, where the term was born in Cape Canaveral, involved first designing the routines by hand, then moulding them into punch cards to be tested. When the program was proved to be working, it was hard wired into a shuttle, quite literally, using wires threaded through and around magnetic cores (through signalling a binary 1, and around signalling a binary 0) (McMillan, 2015). It wasn’t until years after the mostly successful Apollo launches that software evolved from binary, into the first high-level languages such as COBOL and BASIC. In the thirty-year period from the 60s until the 90s, we leapt from punch-cards to a variety of languages, some imperative or procedural – object-oriented or functional. And within this rapid development of new technologies to better facilitate the creation and application of software, we ultimately failed to properly design any method to successfully determine the required input of labour for designing a system, the output we can achieve with a certain number of programmers, or even an effective measurement of productivity in software engineering. And although we have had another 30-year period of innovation and advancement since 1990, we still struggle to adequately measure the process of software engineering. In this essay we will explore what data we can measure as software engineers, why it is difficult to transform this data into a measure of productivity, how companies today are improving their development cycles, and how, as we look to the future, software engineering may become a fully measurable process eventually.

# The Data of Software Engineering

With a piece of functional, concise, yet fully fledged software visualised as the end-product of the work of a team of software engineers, what metrics could be used to measure progress towards this goal? What metrics are useful? Although having a completely digitised development means the measurement and collection of data is easier than ever, separating out the relevant metrics from the bulk is challenging. With so much data, what really tells the story of productivity?

When building a physical item productivity or efficiency can be measured as a function of output over input. We can also track waste with the creation of physical products. With a production speed of 50 units per hour of labour for say, and an average wastage of one unit, any manager can take a proposition such as: double efficiency for double wastage; and quickly decide whether 100 units per hour is worth the 4 unit wastage. Here all the numbers are relevant to the work, and finding inefficiencies is quite simple. In the manufacturing industry the easiest way to measure inefficiencies is to look at the physical and process flows, that is the movement of components around an assembly line, and the steps required to process base materials into the finished product. Rearranging an assembly line can yield massive increases in efficiencies for nearly no cost to a business (the relevant number being distance travelled in metres).

With development of software, there is one central location where all work is carried out. Although an increase in say, monitors, can yield improvements in productivity similar to physical flow mapping, these efficiencies are negligible as every component tool, or system at an engineer’s disposal is at their fingertips, merely milliseconds away. What then is measurable in our development assembly lines? Number of lines of code, time spent at workspace, number of commits, deployments, rollbacks, or bugs, these are all measurable quantities. Indeed, parallels can be drawn here to the world of physical design, such as product design or architecture. But the key differences between software and ‘hardware’, or physical products, is the end goal of a physical product has a fixed form, a shape, a measurable and visual entity that cannot exceed the constraints of a certain space. Software has no such final form, merely a set of requirements to fulfil. It is to the discretion of the programmer to find pathways to fulfil these requirements, and this can create huge differences between the visions or work of different programmers working to similar goals (Sillitti, Janes, Succi & Vernazza, 2003). A relevant example would be like an essay assignment for students: despite a fixed topic, or length, the content of the work and how it was reached would be entirely different processes, for some students taking significantly longer for comparably worse work.

The beauty of software isn’t in the bulk of its source code, but how less code can accomplish the same, or better, results. The removal of a fixed space which the program must occupy (mostly) allows software engineers to explore a variety of options to achieve the desired functionality. Indeed, in situations where a maximum byte-limit is given for a piece of software, a program holding less space may be more time-efficient than a larger alternative. This makes ‘lines written’, as a quantity, difficult to use to measure any amount of productivity. The worst engineer could write a hundred thousand lines to achieve a barely functioning product, but the best could write an accomplished final product in significantly less.

Time spent at workspace is also difficult to use as a valid comparison for work done. In one of the initial largest switches of a company to a 4-day work week, there were gains of productivity of around 20%, despite the decrease of 7 hours per week of work per employee (Booth, 2019). Although this is a more general approach to ‘time spent at workspace’, it is a valid critique of how time invested is not always proportional to input regarding productivity. Especially in design-oriented areas such as software engineering, problem-solving can account for large parts of the work, relying not just on time spent but the creative and inspired parts of an employee. While certain individuals could be stumped by a problem for days or weeks, others may naturally work around the problem with no difficulty. It could be said that the employee ‘spinning their wheels ‘ is inefficient, but in a field requiring not just knowledge but also creativity to deliver is it fair to measure a metric so dependant upon creativity, which itself is immeasurable (except anecdotally)?

In an interesting shift from measurement of progress to measurement of detrimental factors, number of bugs is a relatively easy metric to base decisions on provided that all bugs can be observed. This is unique to the other two data points examined thus far, as bugs are not always innately visible to a programmer, tester, or quality assurance worker. They can rely on incredibly specific use cases to be brought forth, or unique hardware, or even certain future conditions that will not be met for a long period of time, such as number of users, date, or other incremental variables. Sometimes bugs can rely on failure of several individuals works to accommodate all use cases and situations, and in these cases it is the managers discretion to decide which individual failed to make the necessary unit tests or precautions to prevent such bugs. In most cases though, it is easy to pinpoint the specific work which caused a bug, and which software engineer wrote such code. Measurement of creation of bugs as a metric can be useful when observed aside other metrics, such as lines written, so that a more focused figure such as bugs per line coded can be reached. This will highlight underperforming software engineers who need to work on their contributions.

And so we arrive at system specific metrics, such as number of commits, number of deployments, and number of rollbacks. In most every software company there is a different pipeline, or process for the development of software. With in-house repositories, development, staging, and production servers, as well as a myriad of other tools to assist in the engineering process, we find ourselves a myriad of metrics we can observe and use to measure productivity. However just as ‘lines written’ does not always accurately reflect productivity, these metrics may not either. Commits, in a generalised sense of code ‘committed’ to a development source as opposed to the git ‘commit’, can be as large or little as the user determines. In a differently defined environment, certain companies may have specific requirements for commit minimum size or define a commit by a finished module or file, whereas the difficulty or challenge in writing different modules could vary wildly and swing from huge times to write to mere minutes filling a module with boilerplate code and a few unique lines. Deployments, when the development source is ‘deployed’ to the next stage of the development environment, i.e. the standard path of development, testing, QA, staging, and production, can track when a series of commits is deemed to have successfully passed all stages and reached production. This measure allows for poorly written code to have more cycles between development, testing, and QA, which can prevent rushed code from having a ‘productive’ mask. Deployment however is again environment defined – certain companies may not care what amount of code is moved through each stage of the process, or it may be irrelevant as large progress may be represented by less lines of code, as already stated. While metrics such as deployment can be extremely valuable in a firm-specific setting given proper management controls to define modules to be of similar time-cost, these system specific metrics are not an end all.

Frequently in measuring software engineering it can be observed that metrics are devalued by the core concept of what ‘good’ code is, how it can be measured or tracked without manual review and comparison, and if a general comparison can be made between varying portions of code despite each instance varying wildly with complexity, difficulty, etc. To make sensible comparisons between individuals, projects, or companies there must be a standard in defining complexity, difficulty, and content-amount. This is an almost unreasonably difficult paradigm, so what can be done with only our metrics at hand?

# Comparing Software Metrics

Similar to how Big O notation works, we can too boil down our algorithmic approach to measuring productivity by discarding the lowest common factors in each situation and comparing only the highest order properties. By exploring the common generalised variables in an approach to measuring software engineering, our metrics, we can work towards boiling down the relationship between these factors into a simple equation. Although this is an overly simplistic approach, like how space and time complexity is measured, disregarding the finite details will allow valid comparisons to be made despite the inaccuracy of the data being compared. Although this section will not focus on the formulation of such an equation, we can explore the principles, or what such a formula may look like.

At the most basic level we have several factors to be considered when comparing the work of two different software engineers on two different projects. The knowledge and ability of each programmer, the complexity of the work, the time taken (time spent at workspace), lines written, and bugs created. I deliberately ignore the system specific variables mentioned in the previous section as the standards governing the frequency of these can vary wildly from system to system (company to company). While they are worthwhile variables to consider within an organization, they have no place in a theoretical generalized formula. Examining the variables remaining (listed above), I believe we can exclude ‘programmer knowledge/ability’ given the context that our productivity formula will itself reflect programmer ability, albeit for specific instances. The next to be considered for axing would be ‘bugs created’. While a fine metric in principle, there are a few assumptions carried with this data that can interfere with our equation. Examples of such assumptions include: that every bug has been discovered, all bugs carry the same weight, or bugs are innately the fault of the developer and should subtract from their productivity score as a result. As each bug is case specific, generalising them has a distinct negative impact on the integrity of a formula, so we will also abandon this variable, leaving us with work complexity, time taken, and lines written. A relatively simple formula involving these three variables would be the easiest way to model software engineering productivity at a basic, but useful level. How these variables may be transformed within an equation however, is a gargantuan task that will not be considered within the content of this report.

# Ethical Considerations with Measuring Productivity

Outside of the above considered metrics – simple trackable metrics that can be considered within a workplace environment without concern – there are a number of more ambitious approaches to examining employee behaviour in the workplace to better measure productivity. Inter-employee communication channels, application use on the desktop, subordinate-superior queries, and other more private metrics are frequently observed and monitored by employers in an attempt to measure every aspect of the working environment. If a state of constant scrutiny a large amount of information can be gathered concerning employee habits and practices, and all of this can feed into how productive an individual may be, but is it ethical to be so invasive?

In a world of increased scrutiny, it is more the norm than ever before that there is a large amount of monitoring on the part of employers. According to Hartman, 1998, work ‘surveillance’ can have a varied and largely detrimental effect on an individual’s wellbeing, and not only their mental state but also their physical health. In software engineering there is a temptation to reduce employees to their software presence; their messages on Slack or other employee communication services, or their Stack Overflow history, but any kind of invasive measurement will only serve to harm productivity, and a measure of productivity that itself harms productivity is not worth considering – and this perspective completely ignores the real ethical concerns around privacy in the workplace.

Indeed, in the software engineering world data is the greatest asset. It accounts for significant revenues for many large technology companies, and with these companies working on improving their systems by using their user’s information, the temptation is all too clear to improve workplace productivity using their employee’s information. Such data can be anything from chat logs, emails, statistics about breaks, time at desk, time in different applications, time spent working on different projects, non-work-related browsing or interactions within the workplace and even more. In a world of significant surveillance nearly everything is measurable. It is up to the software companies themselves to recognize that while measuring progress on projects and employee engagement at the workplace is entirely acceptable, comparing employee’s habits and personal quirks is not.

# Computational Platforms for Measuring Productivity

There are a variety of different services available to track productivity in a software engineering environment, with companies offering solutions with customized metrics, equations, or new and more invasive tools to track every aspect of an employee’s desktop environment. Additionally, tools are available to assist in the development process; reminder, note-taking, scheduling applications to augment a developer process and streamline the process. Many of these applications tread a fine line between ethical assistance and over-reaching managerial-oppression tools, which as previously mentioned, has been found in studies to negatively impact worker productivity (Hartman, 1988).

Products available to workplaces such as Time Doctor are terribly invasive, working as spyware masquerading as “time management software” (Agu, 2017). The feature-set available – regular screenshots of user’s desktops, tracking of all internet history and application use time, while no doubt useful for a Stalin-like manager to rule over his subordinates, has no real measure of worker productivity in the workplace. Recent studies on the principle of the four-day workweek, such as the one by Microsoft Japan, show gains of 40% in productivity with a time reduction of 20% in the week (Gilchrist, 2019). This further discredits the typical McGregor, 1960, Theory X management approach to employees – that employees are inherently unmotivated, inefficient, and need to be monitored and corrected to improve a business. It would be assumed that although these monitoring tools can provide a more comprehensive data set to measure and compare workplace productivity, the motivation-cost isn’t worth the benefit.

Lastly, we are left with the more standard platforms for measuring employee participation: services such as Git and SVN allow basic employee monitoring in the form of output, bug, and revision tracking, with the focus on measuring the production process, not the employee. These services are typically known as version control systems, and their contribution to software metrics is quite significant – offering feature-complete tracking of what files are being worked on, when a file was last worked on, who wrote what and how long ago, questions that provide data to be aggregated and analysed by a company to understand what each employee is doing, and how a project is progressing. Almost all software engineering firms use some form of version control and it is because of the clear benefits not only to employers, but also employees.

# Conclusion

The world of software engineering is a bloated landscape, filled with options, alternatives, solutions for programs that don’t yet exist (or that will never exist). The data we can gather from the software engineering process is much the same, although much of it can be incredibly useful, most of it cannot stand on its own, and we need new and inventive ways to compare and manipulate different factors to create an accurate portrayal of productivity. It is easy to draw incorrect conclusions from looking at any dataset in the wrong way, never mind the nuance or care needed to fully understand the software engineering process. It is for this reason that productivity in software engineering is so hard to recognize, or quantify, and why solutions to this problem are so sought after.

# Bibliography

Agu, K. (2019). *6 Software Tools for Monitoring Employee Productivity*. Retrieved 1 November 2019, from https://www.huffpost.com/entry/post\_11966\_b\_10099296

Booth, R. (2019). *Is this the age of the four-day week?* Retrieved 28 October 2019, from https://www.theguardian.com/world/2019/mar/13/age-of-four-day-week-workers-productivity

Gilchrist, K. (2019). *Microsoft Japan's 4-day workweek experiment sees productivity jump 40%*. Retrieved 1 November 2019, from https://www.cnbc.com/2019/11/04/microsoft-japan-4-day-work-week-experiment-sees-productivity-jump-40percent.html

McGregor, D. (1960). Theory X and theory Y. *Organization theory*, *358*, 374.

McMillan, R. (2015). *Her Code Got Humans on the Moon—And Invented Software Itself*. Retrieved 28 October 2019, from https://www.wired.com/2015/10/margaret-hamilton-nasa-apollo/

Pincus Hartman, L. (1998). Ethics*. Journal of Business Strategy*, 19(3), 16-19. doi: 10.1108/eb039931

Sillitti, A., Janes, A., Succi, G., & Vernazza, T. (2003). Collecting, integrating and analyzing software metrics and personal software process data. *Proceedings of The 20Th IEEE Instrumentation Technology Conference (Cat No 03CH37412) EURMIC-03*, 336-342. doi: 10.1109/eurmic.2003.1231611