Measuring Software Engineering

Vsevolod Syrtsov - 18323202
syrtsovv@tcd.ie

Contents

1 The place occupied by measurement, and other fundamentals

The need to measure the output of software engineering is absolutely fundamental now. In the time that it has become accessible, the demand for being able to measure and record software development has followed. We've arrived at the point that most companies now provide the necessary services as part of the standards they would usually have. This availability has led to this analysis to be a tenet of professionalism in the software engineering industry. As a matter of fact, it has become such a massive part of the industry that companies may now choose to offload this to other companies. The services that these auxiliary companies package are, still, also subject to the same relatively scrutinous analysis. The amount of platforms and companies that are available for this matter are simply a testament to the transformative nature of the process.

Given two people who have spent time developing software and being exposed to different structures used by their respective groups, they will most likely hold different views of what can be defined as good metrics. Obviously, bickering over the effectiveness of one over the other is better left for corporate staff rooms, or stack overflow forums - the main focus of this report is examining the different characteristics of a metric. How much use will different configurations of metrics provide? A metric if being used, should be well researched before implementing it into a workflow. "Say what you mean, and mean what you say" should be the mantra of concise analysis.

Something that came to mind in the time spent working on this report was an anecdote - a professor in a lecture for AI is asked, "How do we know that an AI actually works, and does all of its tasks correctly?", he responds, "If it makes you money, then it works." Leaving aside the financial quip, neither party has an absolute, infallible idea for how to define the constraints of "what works" in the context of AI. Still, the same principle can be applied to using the metrics used in software engineering nowadays. Every situation is nuanced and project leads must understand their team, the context in which their product is being developed, requirements set out by the client - so on and so forth.

If one were to structure the process of measurement, the first thing to be done would be to understand what is being measured. Depending on what stage of development the product is at, and what type of product is being made, some assessments will be prioritised over others. These can range from budgeting costs, quality testing, security, bottlenecks in development, assessment of effort required to achieve some goal, automated tests, management

of computational power, all these being unique from others. The same way you wouldn't bring a ruler to bed to measure how long you slept, you must select your method of measurement carefully. Deadlines exist, going in and implementing any metric is expensive (albeit fashionable), the passing of time grows increasingly faster with every failed attempt, and false positives that feed useless information can be devastating if overlooked. In a professional setting, where the workflow is relatively watertight - especially in crunch time - failing to identify the correct variables can completely undermine an operation.

Assuming some information has been put together, based on the criteria set out, this set of data is never without a place. What must be done with this information? More importantly what kind of information is it? For example there exists a code base, in either early, middle or late stages of development. The complexity, size, usability, testability or reliability is needed to be evaluated. In this context, the code base will, ultimately, be the direct result of the process that is needed to be evaluated. Examining the code base as the output of this process, a set of resources would have surely been the subject upon which this craftful transmutation has acted upon. Finally, an empirical model can be derived, one where the data gathered has its place in one of three stages in this structure.

2 Observation of software engineering

Contemporary methods of software engineering are the result of decades of observing the process of creating software. What came first, the chicken or the egg? The perpetual state of change and layering of software on software, on hardware, layers upon layers of abstraction or the collective mentality that things must be faster in a way that isn't simply fundamentally a result of the base units being faster? The methods employed now are simply the pragmatic structuring of parts of engineering a product to its client's specifications. I think it's important to frame the most common methods of assessment alongside the most commonly employed software engineering practices.

The most common set of practices for analysis today is defined by the Agile approach. In this structure the requirements of the client and the development team develop gradually. Having a clearly defined approach is quite useful. It will most likely rid the engineer of the question of what they should measure, and instead let them focus on piecing metrics and measuring practices together in harmony. In this approach, the development and testing processes are aligned with the demands of the customer, with obligations given to a team of engineers being assessed and tuned to realistic goals. Project goals are assessed well in advance. Troves of previously established information, and the combined experience of long-serving software engineers aid in attentively selecting the points of attack on a project. With this new approach only information that is conducive to the development of a project is well considered, and then everything else is secondary.

The advantages of agile testing are shown in how it saves time and money, reduces documentation, is flexible and highly adaptable and provides adequate time for engineers to solve problems so that bugs do not end up everywhere.

To adopt conventional practices is easy; to be the first to accept them as conventional is hailed as revolutionary and often "a triumph" by the greatest contemporary observers. Modern practices in software engineering are often constructed from the creeping shift of regard to assessment of code. These new perspectives on how code should be written are influenced highly by quiet chatter throughout the industry.

3 Useful (and very situational) metrics, their character and their application

As mentioned previously, the selection of which metrics used must come under heavy scrutiny. Failure to do so will surely result in financial blunders, so for the sake of pride at the very least, selection must be suited to their application. The requirements of the project should be considered in this selection process, as otherwise the usage of the metrics will end up simply as a waste of time and money. The information gathered will render the hard drive on which it is stored effectively a very expensive brick with absolutely no uses. On the other hand, successfully selecting the correct metrics will greatly contribute to the success of a project, and may even result in less time spent on future projects that use that metric (limitations apply of course). The metrics a project lead chooses to implement should be consistent, and most importantly usable.

Size oriented metrics are quite basic, but can be used as a building block. These measurements are taken by checking the size of the software produced in some way, this includes how long the project has taken, how many people are working on it, the amount of money spent, the number of errors or defects in the project, the lines of codes used and more. When considering a project, for a company it would be important to know how much money it is costing, as well as how many people are working on the project. Lines of Code can be chosen as a normalisation value, this means that the other measurements are tested based on how many lines of code were produced. For example, the errors per KLOC (thousand lines of code), the cost per KLOC etc. This can be valuable as a quick shorthand for how much work is being done, and it's fairly easy to count, however there is no guarantee if all the lines of code are necessary, or even if they are of a decent quality. This is very basic and should be approached with an understanding of other methods that could be employed. This approach can be viewed as measurement of a scalar object, whereas functional oriented metrics use the functionality of the product as their normalisation value.

The idea is that this functional analysis can give an idea of how much a system will cost, how much time it will take etc. The difficulty lies in how much of the data can be measured correctly. Computation values are subjective rather than objective and when the result of the functional analysis is found, it means very little on its own. Its value is being compared to other projects' previously analysed value, estimating cost, time and other factors. Time is quite expensive, days turn into weeks, and weeks turn into months. Programs and code bases should be updated, maintenance is expensive in

terms of time spent on it that could be spent on new lucrative projects. More on this will be covered later in the report. Software can be assessed in not only its size or functionality, but its change over time with respect to its movement towards the team's goal.

Automated testing may measure how concisely a software system is observed. The data returned from this type of testing is simple, given that the tests are properly written. These types of tests require another team, as time spent on figuring out kinks on software can run into dozens of hours, as we all well know. The existence of a quality assurance team will need to also come into consideration. However, in an agile environment where the code is being expanded upon frequently, in my experience, testers can and generally do save a lot of grief down the road.

Metrics can serve to be very useful alongside consideration of the humanity of a developer. Recognising vulnerabilities and limitations in a team or an individual can benefit management/the client greatly. Correlating changes in data on the measured productivity of a team/single engineer can be used to spot fall offs, and used to adjust goals accordingly. Simple metrics, which don't necessarily even need to be related to the actual software that is written by a developer, can be looked at. This can yield useful nuanced information which can amend a hidden problem that one may not have known was manifesting. Active days, number of assignments allocated to a developer, how much they interact with other people on the team, time spent on tasks are all very powerful metrics when applied correctly.

4 People and machines playing nicely

Developers can employ an algorithmic approach to assessing data, or utilise the services of an AI to further help with this. Assessing data collected from software engineers is, according to some, a waste. Some have suggested that there doesn't exist an objective metric of developer productivity, and the problem of it will remain unsolved. While it may be difficult to find a truly objective metric of productivity, the use of algorithms to assess it may still be useful, especially as comparison points. As mentioned previously, data in isolation is relatively useless but when found in groupings of other information to control against it can contribute to the success of a project. All sorts of measurement data can be fed into an AI machine to significantly reduce the amount of time engineers spend on debugging, while also speeding up the process of rolling out new software.

As mentioned previously, algorithms relating to measurement of productivity, or code quality are constrained by a set of variables. These should

generally give a perspective on how well some code base is getting along. I feel there is no use in naming a list of metrics which can be found with a simple internet search, instead examining the use of algorithmic metrics will suffice. Obviously, once again, the variables that are subject to the algorithmic measurement should be chosen with respect to the type of data that a project needs to assess. The function point algorithm (Allen J. Albrecht) is a general description of what variables should be considered when creating, or choosing an algorithm to use. A measure of how many inputs, outputs, modularity, software accesses (through an API for example), use of external APIs, can all be measured.

I will, in contrast to what I said earlier, briefly mention the Halstead Metric, as an example of how these constraints can be used to loosely approximate volume and program effort. For a given program, the number of distinct operators, distinct operands that appear in a program, total number of operator and operand occurrences, several measures can be calculated. Vocabulary, which is the variance of operands and operators, estimated program length (sum of log functions outside of the scope of this report), volume, difficulty and effort. The number of bugs that a program could generate, and time to run can be extrapolated further from these measurements. This very general set of equations have been subject to controversy, but when applied practically have shown to be loosely correct, but I digress. If the effort is done to include all of the elements that affect the output or characterise the program, the algorithm can become so complicated that it's useless. Being pragmatic is integral to creating usable software, and a clinically comprehensive algorithm should be considered with that axiom in mind. It means picking the constraints that are the most important to the team, developer, client or management.

An example of AI being used to measure software can be found in Facebook's Getafix AI, which implements another AI called SapFix, to detect bugs and software changes. Getafix automatically finds fixes for bugs and offers them to engineers to approve. This allows engineers to work more effectively, and it promotes better overall code quality. The goal of Getafix is to let computers take care of the routine work, albeit under the watchful eye of a human, who must decide when a bug requires a complex, nonroutine remediation.

Finally, computational platforms which allow engineers to work together and have a domain where all of these services are provided in one spot. Modularity upon modularity effectively removes the need to constantly be putting together the most common services by the engineer. Commits, time spent on tasks, time spent on research can all be charted on these platforms. These platforms can provide general analytics to its users on things relating to the

development of software and team health alike.Communication is essential to any fruitful relationship, and this can be also quantified (however evil it may sound) using these platforms.

5 What about the engineer?

Computers have generally gained their popularity from doing things faster, more efficiently than regular old human beings but most importantly they provide a tool which aids us in our own experiences. The furious stride of progressive and organisational change is that which very few can stay on the border of. If the health of resources of an organisation is of any concern then forcing needless, or otherwise detrimental practices on a team should be heavily reconsidered. Every engineer that comes into work every day is a human being, and having machines breathing down their neck watching their every move without their consent (or even full consent) may be incredibly uncomfortable.

What goes around comes around, and useless metrics will ultimately foster useless work being done. If an employer doesn't trust their recruitment officer to correctly select people to employ, then I envision a far more bleak environment than there should be. The recruitment process should be there to lend engineers the trust in their work, to choose engineers which take pride in their work as much as management requires the work to be finished. It is the developers responsibility to be a good worker, as far as they can manage, not the employer's to enforce this ethic.

6 Closing words

Finally, from the perspective of the uninitiated, the practice of pedantic observation, management and measurement of a software engineer at work can well be a deferrant from any involvement in software engineering whatsoever. Personally I have no grudge held against the assessment of some creation or another. Why should I? It is fundamental, especially in an industry where products are to be delivered under the obligation of a standard. As with anything, if a step back isn't taken to observe the progress, that lack of assessment will surely lend itself to all those involved being incredibly lost at some stage.

The progress, however, that is being interrogated must absolutely not be mistaken for the actual act of observing that progress. The goal must always be towards that which is at the horizon. Effective measurement of

software engineering will take into account all factors delegated to the process and tweak them to create a highway towards that goal. Any amount of intricate footwork while taking that aforementioned simple step back will only serve to confuse and alienate those who are at the control panel making the software. The information being gathered must at all costs be used to encourage development and improve the derivative product of the concoction that is the software engineering workflow.

All things must be considered, on a micro and macrocosmic level, when assessing these measurements and metrics and models and so on. The team lead or management is as much responsible for being conscious of their choices as much as giant corporations, and the software engineering industry as a whole, in employing these practices. How far does this rabbit hole of information go? Is it all just a pyramid scheme of information where every tier is only important once the information derived from that tier is proved to be important? Measuring software engineering is a double edged sword which must be respected and treated tenderly.

7 References

https://ai.facebook.com/blog/getafix-how-facebook-tools-learn-to-fix-bugs-automa

https://www.researchgate.net/publication/220344622_Measurement_in_software_engineering_From_the_roadmap_to_the_crossroads

https://www.infoq.com/news/2008/01/crunch-mode/

https://www.researchgate.net/publication/236845436_Metrics_models_and_measurements_in_software_reliability

https://en.wikipedia.org/wiki/Halstead_complexity_measures

https://engineering.fb.com/2018/11/06/developer-tools/getafix-how-facebook-tools

A. J. Albrecht, "Measuring Application Development Productivity," Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14–17, IBM Corporation (1979), pp. 83–92.