Measuring Software Engineering

Bibliography

Note these are a list of resources I studied for this article, however I won't be strictly referencing any of them as this is meant to be a personal response using these as inspiration to inform views.

https://www.youtube.com/watch?v=Dp5_1QPLps0

https://stackify.com/measuring-software-development-productivity/

https://dev.to/nickhodges/can-developer-productivity-be-measured-1npo

https://medium.com/the-liberators/agile-teams-dont-use-happiness-metrics-measure-team-morale-3050b339d8af

https://blog.usenotion.com/5-essential-software-development-metrics-to-measure-team-health-a08f089528d4

https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams

http://engineering.kapost.com/2015/08/you-can-and-should-measure-software-engineering-performance/

http://www.citeulike.org/group/3370/article/12458067

http://2016.msrconf.org/

http://www.nextlearning.nl/wp-content/uploads/sites/11/2015/02/McKinsey-on-Impact-social-technologies.pdf

http://s3.amazonaws.com/academia.edu.documents/35963610/2014_American_Behavioral_Scientist-2014-Chen-0002764214556808_networked_worker.pdf?AWSAccessKeyId=AKIAJ56TQJRTWSMTNPEA&Expires=1479122144&Signature=pO6ZkNbSZgbNTqHMd7xyIdMV5iE%3D&response-content-disposition=inline%3B%20filename%3D2014_Do_Networked_Workers_Have_More_Con.pdf

https://www.researchgate.net/profile/Jan_Sauermann2/publication/285356496_Network_Effects_on_Worker_Productivity/links/565d91c508ae4988a7bc7397.pdf

http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf

http://patentimages.storage.googleapis.com/pdfs/US20130275187.pdf

Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157.

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

Grambow, G., Oberhauser, R. and Reichert, M (2013) Automated Software Engineering Process Assessment: Supporting Diverse Models using an Ontology. Int'l Journal on Advances in Software, 6 (1 & 2). pp. 213-224.

P.M. Johnson et al., "Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined," Proc. 25th Int'l Conf. Software Eng. (ICSE 03), IEEE CS, 2003, pp. 641–646.

https://medium.com/static-object/3-key-metrics-to-measure-developer-productivity-c7cec44f0f67

https://www.7pace.com/blog/how-to-measure-developer-productivity

https://www.codesimplicity.com/post/measuring-developer-productivity/

https://www.quora.com/Whats-the-best-way-of-measuring-developer-productivity

https://www.devteam.space/blog/how-to-measure-developer-productivity/

https://dev9.com/blog-posts/2015/1/the-myth-of-developer-productivity

https://softwareengineering.stackexchange.com/questions/179616/a-good-programmer-can-be-as-10x-times-more-productive-than-a-mediocre-one

https://martinfowler.com/bliki/CannotMeasureProductivity.html

https://www.networkworld.com/article/2182958/software/how-ibm-started-grading-its-developers--productivity.html

Introduction

Before we look into how we can measure software engineering we need to look into why we would want to do that it in the first place. The obvious reason is to monitor developer productivity and try address the causes of any decreasing productivity and encourage factors that increase productivity. This concerns not only on the scale of individual developers but also the scale of projects or companies, as well as measuring factors such as progress and complexity of projects. This might be done by the developers themselves, the company hiring the developers, or by external bodies ie. researchers. Throughout this report we shall discuss factors such as measurable data, overview of tools to perform this measurement and also importantly the ethics of these actions.

Measuring software engineering is not a new idea or process, proxy measures such as lines of code or defect rate per 1000 lines of code in pre-production/production code have been used for a long time. Standardising and measuring software has been a way to try cope and manage with the increasing complexity and scale of software development. As monitoring tools become more pervasive, cheaper to use and deliver more accurate and useful information, it's likely such measurement becomes integrated into work environments and companies.

Measurable data concerns

Measurable data has been a large sticking point for software engineering. For years inaccurate proxies such as loc (lines of code) and defect rates were used for a number of factors. One is that these tasks are relatively impersonal and correspond to the project itself, hence has less objection from the developers themselves. The ethical considerations of these are also thereby lessened. Another reason is the low overhead associated with these tasks. Measurement often has an overhead component, especially the manual variety and this overhead itself can affect productivity. Associated with this is the fact that these methods are easy to measure, non-interrupting, measurable for all software and automate measurement of. These are all factors that can be considered overhead. As automating tools and measurement methodology improved this overhead has been shrinking over time.

On the developer side the ethical and personal privacy concerns are noticeable. It's understandable why developers do not want their behaviour to be measured. From the privacy side factors detailing the behaviour of developers could potentially be misused and misunderstood by "simple" algorithms which lack the decision making capabilities of humans. It's argued that software development productivity is difficult to measure and account for, that an algorithm would in fact even have a detrimental effect by ignoring the intangibles. Add to that the fear that management would misconstrue the data and that having such a trail is in itself undesirable it's easy to see why developers would be worried. People also do not like to be compared negatively to their peers or be considered as inadequate, maybe not even as average. This adds another human element to the desire to prevent such measurement. We see that methodologies that self measure data and use it for personal diagnosis had as a result less concerns over privacy.

From the non-developer side there's the factor that measuring software development can lead to misinterpretation. The phrase "correlation not causation" comes to mind. The fear that poor inference of the data by humans or machines can cause unintended consequences is a relatively logical one. Furthermore, this can cause developers themselves to react to this measurement by trying to "game" the measurements to shine favourably on them but also as a result lead to poorer code.

Now that I've stated the misgivings about measuring software engineering, we can nicely lead to the next section on the approaches to try measure software development as well as the technical tools to do so. These are closely interlinked as by having more automated tools and effective algorithms we can reduce the overhead of measuring software engineering while boosting returns on the measurement. Another fact that might become obvious while reading ahead is that many of these are relatively impersonal and based more on detecting issues in projects rather than ascertain any facts about any individual developers for the reasons given above.

Metrics

From reading through quite a bit of the material above on software engineering it appears that software engineering measurement methodologies are varied, have mixed reception and adoption as well as have evolved over time. Comparatively software metrics seemed to have been more durable as a measure of something more concrete and common across programs. One has to be careful making inferences from any single one of these measures but at the same time these measures act as proxies to signal issues in the code base. Due to the reasons discussed above a lot of the measures seem to be more focussed on monitoring project health over developer productivity. So to begin with we will list some of the measurable metrics of software engineering. More vague metrics such as cohesion and coupling are not going to be included as they are harder to pin down observable metrics.

The obvious one is Source lines of code (SLOC) to estimate project size. This is a fairly simple metric used as a proxy to try estimate the complexity of the project. Size itself very rarely matters but more importantly this is used as a proxy for complexity and progress. The drawbacks for this is relatively obvious, more lines of code does not necessarily mean more progress and neither does it mean that the project itself is necessarily more complex. By encouraging this you may inadvertently cause developers to write worse code to inflate this number.

Another one is defect rate in public facing code. A high defect rate can be used as a proxy for quality of the code such as if there’s sufficient testing. Along with this defect rate in internal code and rate of dealing with the defects can be used to signal any worrying patterns. Drawbacks include the fact that the defect rate is based on the number of observed defects, not the actual defect rate hence a lower defect rate doesn’t necessarily mean a lower rate of actual defects in the program.

Milestones can be used to measure progress towards tasks as well as time taken per milestone to observe if the project is on track or not. While milestones can monitor progress it can’t itself figure out why the milestones might not be being met. Perhaps the milestones were poorly defined, perhaps the domain has inherently more complexity than expected. Include code velocity within this section.

A common favourite is instead of measuring the progress in the software itself you measure the value the software brings. This feels relatively useless to me in dealing with a project itself, since it only answers how effective the solution. I suppose this acts as a sort of proxy of how well you addressed the specification. Again this feels like it answers the what happened rather than the why it happened, and any number of reasons can result in a project not bringing sufficient value.

Feature deployment speed can be used to measure how long it takes for a feature to be created and then successfully be deployed after passing through the pipeline of QA etc. A long time can indicate poor infrastructure or testing etc. Within this I include the time taken between the formation of an idea and actually making it (lead time), time taken to open and fix an opened issue (fix rate) and cycle time.

Code churn, the percentage of a developer's code representing an edit to their recent work over a certain period of time. High software churn is used to indicate poor quality code since it needs to be repeatedly modified.

Customer satisfaction which falls under the bringing value banner in my opinion. Feature adoption I think is a good proxy here personally since it's relatively objective and measurable. We also see an increase in telemetry etc. to measure factors such as adoption since these metrics give insight into the functionality and purpose part of software.

As you can see many of these metrics come with caveats on their efficiency and are often used as canary warning systems to try and indicate the presence of issues rather than accurately pinpointing the root cause. The idea of looking where it's brightest is emphasized with relevance to this as the more subtle causes are also harder to measure or track. There are other metrics that are used but many of them seem to fall generally under one of these banners. However, many of these metrics are frequently used in some way, shape or form across the software industry which indicates that they are probably somewhat efficient in diagnosing issues in software. These metrics allow for engineers to maintain control over large scale complex projects by ensuring production code meets a minimum level. Furthermore, some of these metrics such as code churn and defect rate can be targeted against individual developers themselves to identify their proficiency at the task.

Important Addendum:

I added this section a bit later and I feel this topic is where this best fits. I decided to add this after thinking about metrics that I think measure software engineering and realised that there were some metrics that weren't included but are tonally quite different on what they measure. In a way this section could go under methodologies or ethics, but I feel these are in every way valid metrics to attempt to measure.

The type of metrics I'm talking about are metrics that measure developer satisfaction, culture and communication within the company. The people working at these companies are humans and therefore human factors matter just like technical factors. A poorly cohesive team I think could very well be the source of many of these technical issues and it's better to fix the root cause rather than simply the symptom.

Polling is the most obvious form of measuring factors such as communication, satisfaction and cohesiveness. The quality of the polling questions and the frequency has a big impact on how effective these questions are at discovering issues. Rather than simply trying to make some token questions, spending time making less subjective more focussed questions would yield better measurements.

The number of open developer positions was suggested as one, since the more the number of open developer positions the greater the pressure on the team to work while understaffed. Similarly the average time to fill can be used to predict how big an impact attrition has on the team.

Methodologies/Algorithms

Methodologies and toolsets for identifying and measuring software engineering have been a constant theme and are as crucial as metrics to try diagnose issues in software. It's not simply about the metrics you use to collect info but how you use these metrics in practise and how you structure the environment to try diagnose and resolve issues, which is where software engineering methodologies come in handy.

One of the obvious methodologies for measuring software engineering quality is testing. Code coverage, regression testing, fuzzy testing etc. This is one of the most blatant examples of a software engineering measuring methodology. By having an effective testing infrastructure the overall quality of engineering improves as a result. Metrics such as code coverage and quality of unit tests are used to measure software engineering/code quality. An extension of this is for example the test driven development philosophy with varying degrees of strictness. This is also a relatively clear cut example of a software methodology and software measurement in general providing clear benefits in managing complexity etc. by ensuring that software meets certain quality guidelines. The purpose of most software measurement methodologies in general to try detect these issues more accurately and deal with them.

Another quite common methodology is peer review of code. Ideas such as pair programming also fall under this umbrella. Whereas unit tests are more automated this is a more human based system. When code successfully passes the unit tests, a peer review of code can ensure that the code is sufficiently readable in terms of style and understandable in terms of general logic quality. Another way to think about this is that unit tests try to capture functionality, where as peer review tries to establish style. This allows code to be more easily managed in the future which is invaluable for large code bases. I think it might also have the side effect of spotting error cases by having other people try to trace through the logic.

A less frequently used methodology is monitoring version commit history and finding code hotspots etc. where changes are routinely taking place to try discover which section of code is being changed most frequently. This is almost more like a metric but considering it's more of a process to try discover where the issues are rather than any measured number I put it under methodology. Really it can go under either, but the key concept is the idea of tracking where changes are being made.

Two key “methodologies” are software development philosophies such as Waterfall and Agile. I see these as methodologies trying to solve another aspect of measuring software development- specifications and goal estimation. Rather than trying to measure the quality of the code, these measures attempt to provide a framework to measure the time to provide functionality and measuring the value provided. Obviously, these frameworks also attempt to maximise productivity in these factors rather than just measuring.

Agile appears to focus on providing functionality and supporting rapid iteration and specification development in parallel with design to ensure that the final product provides the most value, hence meets the specifications. Waterfall on the other hand focuses on firmly establishing the specification to estimate the time taken to create the functionality. They both have their merits in my opinion and both try to tackle the issue of measuring software functionality in terms of value provided to the end consumer and the time/cost taken.

Some more personal methodologies to measure software engineering also exist. This is the concept of tracking factors relating to each individual developer. Whereas the above metrics and methodologies tend to measure more or less in aggregate to improve overall project quality, individual engineer skill can also be tracked and there exist tools and methodologies to track those too. While many of the metrics above can be applied in small scale to each individual developer the issue is that it encourages them to try game the metric, hence why most of the focus historically has been on measuring the progress of the entire project. In the section on tools/platforms I will discuss platforms that try to measure developer performance by measuring a wide range of statistics rather than simply one. Hence, why there was no such section in the metrics part since it requires the low overhead of automation to be viable.

One example is the PSP (Personal Software Process) which allows developers to specify and detect their own productivity and factors such as error rate etc. This is a very manual method which has a high overhead in terms of personal effort and interruptions in implementing this method. In return however, the developer can specify their own criteria to see correspondence to productivity and monitor their own progress. Many of the metrics are more subjective and relies on the developer themselves wishing to measure themselves.

Similar to measuring projects the same metrics can also be applied to individual developers to a degree, the main risk from this is encouraging gamification from developers. If developers have more of an incentive to write worse code then it seems obvious that they would try do so.

Measuring time taken for individual developers to complete tasks, measuring quantity of work done (through loc or more advanced proxies) and measuring quality of work done(defects in code etc.) are all ways to try estimate the work done by an individual developer. Measuring time spent actually working on the task while on the job is another one that applies uniquely to individuals rather than projects(usually measured by tools).

Tools and platforms

In this section I'll some tools and platforms that allow measurement of software development metrics in projects and teams. Many of these platforms also provide other features to developers but I'll only be talking about the metrics they provide to users.

Jira is a platform for agile development that provides in depth analytic info to measure progress during sprints. It provides info such as Velocity charts, version reports, sprint reports etc. It's integrated with software planning tools for sprints and agile development. In a way this goes to show that these metrics are simply considered part of planning and project management. This type of data is easy to collect for software that is integrated with the planning tools and is requested enough by it's users that it's included by default, which may give an indication of the future of the industry and metrics. Many of these tools like Jira provide more open planning metrics like Kanban boards as they recognize that this automated measurement can be improved by allowing more flexibility on what's being monitored rather than reduce it down to a simple one size fits all metric. This matches with the idea that these tools are trying to capture the many nuanced metrics of software engineering that may ultimately impact productivity.

Toggl is a time tracking tool to manage and measure where time is being spent that is available for multiple platforms. This platform supports many platforms so that time tracking can be implemented by anyone easily. The premise is that you type what you are working on, hit start and it works like a stopwatch monitoring what you spend your time on. The site also gives a number of links on how to motivate your employees to time track and training or integration with an experienced Toggl user. I think in many ways this falls under one of the more invasive metrics of trying to measure productivity and I personally find it uncomfortable to fell like I'm being monitored on how I spent my time. This falls back to what the company is paying you for, some people believe that companies pay you solely for you time and basically you are just an easily replaceable cog in the machine (true to an extent in my opinion). The other view is that the company pays you for your skills and the results you can deliver the company. In my opinion both of these views have merit and it really depends on getting the balance right. Personally I feel this metric is also one that encourages gamification in at least it's current format and as thus provides some security to developers as not that many valid inferences can be necessarily made from the data.

Codescene measures VCS history to find the areas of code with the most technical debt by finding coding hotspots and knowledge distribution among the team. This is a relatively impersonal metric in some sense but can also be used to identify how skills are diversified across your team. Is one developer writing spaghetti code only they can maintain? It's very easy to see the appeal of this type of metric measurement as it has a direct correspondence to code. Again however, I feel it might be easy to misinterpret this data as many people are unqualified to make inferences from data no matter how “nicely” presented it is. People make poor inferences from data all the time and if this type of data is to be understood properly than you need people with the appropriate background in statistics. I feel that making such easy to digest data encourages people to try make inferences from this data without realising the logical fallacies and mistakes they engage in because they have no background in dealing with statistics.

GitHub commit history and quantity of code submitted etc. are very simple and basic metric measurement. This is probably used by many teams and individual developers of varying sizes and expertise. In many ways this represents that metric tracking has become ubiquitous and invaluable to developer not just at the scale of large corporations but also for small teams. It really goes to show how this type of data is used not just as interesting information but mainly to try understand the structure and progress of the actual project, ie. managing complexity.

Trello is a taskboard system that allows developers to measure what tasks have been completed and plan for future tasks. Basically one of the simplest and easiest to use measures of what tasks are being done and what still has to be done. The main idea yet again is to track the complexity of the project and the constituent parts of the project.

Semantic Design offers analysis of structure of source code and provides certain metrics on actionable issues. I’m not too familiar on this and it seems to be a complex measurement method that’s basically just meant to provide a better loc type metric. But the point is that tools such as these show that more complex metrics can be measured now as these tools increase in complexity and nuance in how they track metrics. Logically this should reduce the amount of false positives and false negatives.

Hackystat is an automated measurement of a large number of developer statistics. This case is interesting because despite the productivity improvement many of the developers and users objected to the very invasive and pervasive monitoring of this tool. This may in turn even affect the behaviour of these developers (I think there’s a thing where people behave differently when they know they are being monitored). The author of this tool believed that by putting the proper barriers in place developers would use the tool as they can know that only they would have access to this info. Personally I believe this is naive because what would probably happen instead in my opinion is that management would force their team to use this tool and read all the data.

Cast is a commercial software analysis tool which provides metrics around the structural integrity of code. Used by IBM to grade developers. Good example of high profile companies using these types of tools as they only want the highest grade employees. More than that they see someone who contributes bad code to be a liability rather than an asset.

Personal Comments

I feel this is a good time to add some personal comments before I talk about the ethics of this data collection. I believe one reason why measuring Software engineering is hard is the diversity of the field, the transferability of skills and the relative lack of software engineers relative to demand, at least compared to other industries. This means that software engineers are relatively less specialized and thus harder to measure. Over time we may see fragmentation of software companies to fill specific niches and a consequent specialisation of skills in turn. If developers perform more similar tasks it would be easier to develop comparable metrics and the benefit of being able to define set deadlines and better price signalling might favour the modularisation of jobs over the more custom work developers do right now. One of the key topics emphasized by the computer science course for example is flexibility and the ability to find solutions through searching. It's often more important to know how to use new tools quickly rather than be able to use one tool very effectively. This is probably due to the large topic area that computer scientists have to work in compared to more specialized fields.

Another thing I want to mention is that from the perspective of a company I'm uncertain how important these types of productivity gains really are. It feels to me that rather than try create even more sophisticated metrics and metric collection, for a lot of these companies it may in fact be more efficient to try spend their money and effort in a different way to increase overall company profitability. This may explain also why to me the metrics so far feel rudimentary and software engineering is considered 'impossible' to measure. There is after all diminishing marginal benefit from trying to improve anything and most developers etc. will fluctuate around the median.

Ethics of data collection

I suppose this is probably the most personal of all the sections as a section on topics obviously in many ways comes down to personal opinion. This is a real concern against this data monitoring and collection, as for example in the Hackystat case where concerns against the breach of privacy by developers was noted as a reason for aversion from the software. It's easy to see why software that monitors every aspect of a person's behaviour and applies judgement on it, even implicitly raises privacy concerns from people. Not only that such software which ranks people causes distaste from developers as people don't like to be compared to others, particularly negatively. This can also lead to lowered morale from the team as a result, so these data accumulation tactics can even lead to worse performance if used poorly. There's also a question of how important people value such personal metrics about themselves to be known as the desire for privacy among developers causes obvious opposition against these measures. Particularly as the company or management with access to this data may not be trusted by the developers to handle it with sufficient security and integrity. Even if it secured by the mere act of collecting this data obviously they have created a vector for this data to leak to malicious third parties.

The other concern is that the data will be misinterpreted or is of poor quality leading to wrong conclusions or only marginally true conclusions that miss the bigger picture and lead to negative consequences for the developers or even on the other hand positive consequences for the wrong developers, thus providing the wrong incentives. The mantra “correlation not causation” is an important one that is frequently forgotten by people as it’s easy to draw false conclusions from statistics particularly if presented convincingly. This is a very legitimate concern for example encouraging lines of code as a measure of productivity leads to larger, less concise and pointless code. The factors that affect productivity are multi-faceted and hard to capture in it’s full essence with simple measurements. By prioritizing the wrong factors you incentivise poorer code and lose out on productivity gained from actions unrecognized by the measured factors. Whether this data is judged by humans or algorithms this same mistake can happen due to errors. Once again there’s also a ethical question for example, even if an algorithm or human measuring is usually accurate is it fair to allow these to make decisions on this data if there’s a chance of error? Particularly as these decisions can cause cascading negative influence for the future livelihood for the developer as it marks their work history.

Even among the data collected there’s a question of what point the data collection becomes invasive and what data it is moral to try make inferences from. I won’t beleaguer the point but it’s fair to say that making inferences on some data is immoral and, in many cases, even illegal. Even out of the data it is legal to collect I personally don’t think collection of data that can’t be shown to actually successfully measure indicators of productivity or quality should be allowed. This is a fair agreement for both parties I think as it prevents wasteful measurement and unnecessary breaches of privacy. The hard question comes from what data is it legal to collect, has a direct impact on quality of code and isn’t directly immoral to collect but still raises privacy concerns. I think this is an important topic to consider. One important step is to make sure people have useful claritive information on what data is being collected so individual developers who dislike the practise are aware and can campaign against it and/or leave the company.

My personal opinion on this is that first of all data that can’t be proved to actually relate to productivity are irrelevant to be measured and should definitely not be. The second factor is that definitely anything that’s illegal to discriminate based on also shouldn’t measured for that purpose. Other factors are generally fair game in my view. Sure the potential for invasive measurement exists but it also exists in the consumer space and I believe that the danger for privacy invasions exists more deeply in the consumer space as more parties are involved in more private information. There’s also the counter - acting force that developers themselves are relatively tech savvy and have more leverage than most normal consumers due to the relative demand for their services and are more capable of understanding the risks to their privacy and fighting back.

There should definitely be a concern for security of this personal data associated with individual people but against the many opposing concerns that people have to devote a limited quantity of resources to, I don’t see a big reason why the privacy of developer information matters so much beyond that fact that obviously I’d be personally biased to protecting it. As a person who prefers to give out the least amount of information possible in almost every scenario I’d be very happy to reduce the amount of information collected but I recognize that it’s an incredibly biased opinion as I don’t feel like I gain enough value in exchange for the information, whereas other might. I also don’t have sufficient information to evaluate how much value is actually being generated for me in exchange for my personal info and whether I’d actually be willing to make the trade-off. As such, the primary factor I find most important to support is the increase in clarity of what information is being collected and what it’s used for more than anything else. The security of this information is also another concern, but not one I feel needs to be communicated to me, as much as being mandated are enforced to be stored properly.