Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33012 Software Engineering

Measuring Software Engineering Report

Matteo McGuinness, Std# 18324545

December 31, 2021

Contents

In this report we will look at how one could measure software engineering. Initially we will take a deep-dive into what are the qualities of a software engineer and how we could measure their activity so that we could get a grasp as to their strengths and weakness. We will then take a look at platforms on which one could gather and perform calculations on the data provided by an engineer's working history. After that we will take a look at the various kinds of computations that can be done over such data. Finally at the end we will take a step back and look at the ethical, moral and legal issues that surround the processing of this kind of personal data relating to a software engineer.

Section One: Measure Engineering Activity

So how does one measure the activity of a software engineer and assess their strengths and weaknesses. In this section we will have a look at the most common ways this can be done and then have a look at some larger, more complex questions and try and find an answer to those.

Individual Work

The first idea that comes to a lot of peoples minds when discussing the performance of a software engineer is their productivity. In a corporate environment or in any environment where a solution to a project is sought to be achieved, one needs results and if the engineer in question is not delivering and working consistently to find a solution there can be problems.

So, we need to figure out how consistently the engineer is providing solutions through their code. In today's era, this is easily trackable by looking at a engineers commit history. We can easily track how often an engineer produces solutions by just measuring his commits and commit frequency. We can have a look at how many times they commit weekly, monthly or even yearly. Therefore, we can take a look at their average weekly commits, measure their frequency and thus measure their productivity. We now have measured the software engineer and we can differentiate one that is worth his salt and one that is not.

It would be nice if it were so simple, but unfortunately that is not the case. The aforementioned solution has many holes in its logic. For instance, if an engineer is producing more code, more often it does not necessarily mean that that solution is any good. They could just be piling on to a mountain of badly written code and incrementally making the overall solution worse and worse. Then the manager who was expecting a quick and expeditious software solution to the project, because he sees how often his engineer is committing his work, instead receives a badly structured solution, that barely works and would take another engineer more time to fix that solution then to just start from scratch.

Now in this case, the engineer was productive but they were not performing well and delivered a poor solution. On account of this, is there any way we could take into account a software engineers' productivity but also measure if the engineer is competent in designing and implementing their solutions. This is probably a more difficult question to answer and often times does not get the weight it deserves. To try and answer this question we could begin by delving deeper into the commits of an engineer.

For instance, we can have a look at the number of lines of code that an engineer adds in each commit versus the amounts of lines they delete each commit. This is useful as it gives us a glimpse into whether the engineer is seeking to arrive to an efficient solution. If the engineer continuously strives for an efficient solution, they will be deleting a lot of lines as they improve and refactor a solution to meet more and more criteria. This relationship between lines added and removed is not be all and end all of measuring techniques but it does give an insight into how the engineer comes to their solution.

Working in a Team

Nowadays, a software engineer is most likely going to work in a team, whether he is working for a corporation or even working for an open-source project. Therefore, it is important that we can measure a software engineer's ability to work in a team.

If we were measuring an engineer solely on their performance against themselves, we can only gleam so much, but thankfully we do have the capability to compare and contrast different engineers and their work. This becomes especially useful when looking at engineers working on the same project and we can make more accurate analyses. For example, let's say we were using the method discusses previously and looking at the lines added vs lines removed per commit and we were looking at two different engineers working on the same part of the project. Assuming that the functionality stayed the same. If the first engineer writes up a solution and then the second engineer comes along and reduces the line amount by a significant amount. One could argue that the second engineer came to a more efficient and thus better solution.

Another way we could measure a software engineer working in a team is by looking at the number of tickets they resolve. Tickets in this case meaning problems within the larger project assigned to be resolved by different engineers in a team. Many tickets make up the entire project. By measuring the number of tickets resolved by an engineer we get a sense of their productivity, similar to looking at code frequency. However, in this case we could also rank tickets based on their complexity and what kind of tickets the engineer solves and how complex are those tickets. This is very useful information for a manager as if they see an engineer is easily brushing past simple tickets, it might be time to give incrementally harder tickets.

The solutions proposed above highlight another common aspect of successful engineers and that is structure. This everything from the design and plan of action of a project to the maintainability and testing. While in this case, we are not directly monitoring the code. If we could somehow grasp how an engineer structures his solutions, we can gain an insight into their competency. Some things we can have a look at are the test coverage. Is this project sufficiently tested? Can we then look at the engineer's process by finding out when the test was written in relation to the actual code? Are they an after thought or were they integral to process of building the project?

We can also take a look at how the project was built, by seeing if any tickets were written or assigned. Where was the plan of action written, if there even was one? This especially useful when a team of developers is working a project, so if this is missing questions could be asked about the structure of the project.

Finally, the hope is when working a team, each member improves the other member. So, it is important that we have a look at an engineer's skill to improve the circle of people around him in his team. Thankfully we can measure that. We can have a look at how may comments and reviews an engineer performs for other team members. Vice versa, we can also take a look at how many comments the engineer receives.

Unresolved Methods

As mentioned earlier, at the end of this section, we will take a look at more complex problems and ideas of how one could measure a software engineer and their ability.

A common attribute typically associated with a well-rounded software engineer is their ability to adapt to new situations and new project. It is easy to just provide the same solution to any problem you encounter. But that means that as a software engineer you have reached a ceiling and you are not really testing yourself. The true value of an engineer comes when he solves a problem no one has ever solved before. This means being put in uncomfortable situations and finding an answer. Therefore, I was curious if there was any way one could measure such a quality in an engineer. I am going to pose some questions and hopefully at the end of my research in the next three sections I can provide some answers. We could for instance, look at how many new languages and frameworks the engineer learns. But then the engineer could be falling into the trap of a jack of all trades, master of none. Just because you learn more languages does not mean you are any good at them. Is it not better to learning a few very specific languages master those and use those to solve any problem you may encounter?

As you can see there can be debate over what makes a good software engineer that is why, it is important to keep in mind that all the solutions mentioned above are all parts of complex set of measurements that one can apply to an engineer. Note that if you focus only one parameter, you might be missing the larger picture. For instance, one could look at the number of forks, an engineer receives on their project as proof for good code. But it could just as easily be a great marketing campaign on behalf of the engineer.

Secondly, one can only really fully assess engineers within their organization with work they have access to. We will have a look a little later as to the ethical and legal ramification of analyzing code that an engineer has asked not be made privy to other viewers. However, it does pose a problem to measuring a software engineer if we do not have access to all their work. Often-times engineers only publicize their best works on software development platforms such as GitHub and this may not give the full picture.

Section Two: Platforms

In this section we will take a look at the platforms on which one can gather and perform calculations over the growing data, provided by working software engineers. The emergence of utility computing which supports the gathering of large volumes of data and processing of this data with algorithms. We will take a look at what is possible using these platforms and also the specialist infrastructure that has emerged to perform various kinds of data analysis.

Following on from research performed and own personal experience platforms gathering and analyzing data based on a software engineer's or group of software engineers' activity fall into some common pitfall due to the unorthodox nature of developing a software product. There is no one shoe fits all. Let's first take a look at the data gathering part of the process.

Data Gathering

Intuitively, one would think that having access to someone's online software development platform such as GitHub or Bitbucket, would be enough to gather the data needed to assess the performance of an engineer. Let's go a little more in depth into these platforms and see why there are certain things that cannot be achieved by only gathering data from these platforms.

Using GitHub as a case study we have an online software development platform, where an engineer can create a project(repository) and store all materials and code related to that project in the repository. One can also join a project with other collaborators and work together on a larger scale project. GitHub allows a user to create issues (similar to tickets mentioned previously) and assign them to different users. Thus, allowing to have in one place all that's required to have a structured software project. There are many other features GitHub allows but the previously mentioned functions are all in the scope of this report.

As mentioned earlier, GitHub is an online platform. So, a developer can pick and chose when to upload his code, by committing and pushing it to the repository. However, all the information how the code was developed while the computer was offline is unavailable. If were to do a thorough analysis of the software engineer, we would fall short on that critical data.

One must also be aware of the typical problem that can occur with the development of software products where the idea of process analysis, that may help in future projects is left as an afterthought as there are real hard deadlines that are at the forefront of the engineers and manager's mind. This is also the reason why a software engineers manual self-analysis can often be really unreliable as during the software development process they don't stop and think about the implications of shortcuts taken and solely focus on getting a product delivered as quickly as possible.

If only there was a way to make the gathering of data easier so that the engineer did not have to tax their precious time with such concerns. Well, after reading a couple of research papers specifically “Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data” it seems that developers are finding ingenious ways to do so.

The paper mentioned above details how they are in the process of creating a system that would monitor an engineer’s activity offline as a plugin and would take care of the data gathering itself. They have come up with a very thorough monitoring method called Personal Software Process (PSP). It helps software engineers easily keep track of their work that than can be easily analyzed at any stage of the development process. This continuous monitoring allows helps a developer find out whether specific elements affect their work, such as environment, interactions and behaviors. It also allows them to compare their planned scheduled with the actual delivery of the product. By being able to identify these inefficiencies an engineer can improve their self.

PSP requires the collection of quite detailed metrics of the development times, bugs discovered for all development stages. This is quite powerful because by having these metrics it allows the engineers to have historical data that will be used to:

- Make reliable estimates on variables such as time, schedule and quality
- Figure out to improve the development process
- Highlight any problems

The framework used to collect the data in this particular case is called Hackystat. Developed by the University of Hawaii, it allows users to collect process and analyze data. It collects this data using so called sensors attached to development tools.

Data Analysis

We have looked at how data can be gathered, what do we do once it is? Ideally, we would want to analyze it and come to conclusions based on the results. There exists a myriad of projects online that attempt to just that. A lot of them interface with the GitHub API and receive the data of the user that way. However, having a third party analyze the data using public API’s, may be a security risk some companies just are not willing to take.

By using the ideas outlined in the paper referenced above. They developed a system called PROM. It is designed to help both managers and developers keep projects on track. This system provides two different types of reports one for the manager and one for the software engineer.

Developers can only access their own data, including software metrics, PSP data and analysis results. By viewing this data, a developer can quickly identify their inefficiencies and improve their engineering methodologies.

On the other hand, the manager receives data analysis based on the overall project, instead of individual developers. Even if they wanted to, they could not access the individual developer’s personal data, as it would breach a privacy measure embedded in the system. This turns out to be a good thing anyway as it prevents the manager from getting bogged down in the details instead of focusing on the bigger picture.

Section Three: Computations

In the third section of this report, we will be discussing various kinds of computations that can be done over software engineers' data, in order to profile their performance. The techniques we will include defeasible argumentation which is used in expert systems, simple counting. We will also take a look at how large data sets can be sorted, grouped and fitted, so that we can analyze the software engineer.

As discussed in the first section, we could construct the ideal of a software engineer that fulfills the positive qualities we would like to see in a reasoned, efficient and productive engineer. We could set criteria similar to those laid out in section one and compare our model to the engineers in question. We could use techniques as alluded before such as defeasible reasoning where we defined a proposition to be warranted if it would be believed by an ideal reasoner. Without getting in the specifics of defeasible reasoning, it would basically allow us to come conclusions of a good software engineer without having reached that point through deductive reasoning, which is the reasoning from one statement to another to get to a logical conclusion.

Another main problem we will be looking at in this section is the assessment of large quantities of data. So far, we have only touched on the surface looking at analyzing data where only a few actors were involved, building small projects. However, how would the situation look like if we were to try and analyze the data of a large enterprise with hundreds if not thousands of software engineers working together on the same code base or even different code bases how could we analyze such large quantities of data.

To solve this issue of large data sets we could of course look at improving the technology at hand. However, that can only do so much. We need to take a look at multivariate algorithms that allow us to group and categorize data so that we can analyze the data. Some commonly used algorithms include Hierarchical clustering, K-nearest Neighbors, K-means Clustering. These methods are divided into supervised and unsupervised methods. Supervised Methods assign labels to observations using some kind of structure withing the data, while unsupervised methods using only the internal structure of the data explain the data in a reduced number of dimensions and assign labels to observations. It is important that we are aware of these algorithms when analyzing large sets of data.

Hierarchical clustering analyses the data sets and gives us a sense of the number of different groups in a data set. Let's say we wanted to see if within a set of engineers there was a division between backend and frontend engineers, we could use hierarchical clustering to see the different groupings within the data set based on their similarity.

If we were analyzing an engineer and wanted to see if they were more backend or frontend focused, we could use K-Nearest Neighbors and place them in the correct group according. This is known as Classification.

K-means Clustering on the other hand is similar to hierarchical clustering as in we can create groups within the data set based on relating attributes

Section Four: Ethics and Legal Issues

Hopefully now we have a sense of the different algorithms that can be used to analyze data and come to conclusions regarding the software engineer. There is still a critical element that we have glossed over and that is the ethical, moral and legal issues facing the software engineering measurement tactics discussed above and the processing of this kind of personal data. In this section, we will seek to explore such intricacies and hopefully come to some conclusions as to how best approach those topics.

Ethics

Strictly speaking the definition of ethics is summarized as the following. It is a branch in philosophy that involves systematizing, defending, and recommending concepts of right and wrong behavior.

In a system where every input on the computer, every movement of the mouse, every noise made and every pause taken is tracked, measured and quantized. One could argue that would breach privacy norms and would give an immense amount of power to the controller of such a system. It may seem a little extreme, however, such systems are believed to be already in place. The capability to do so for sure is. It is quite Orwellian, the concept of privacy at work would cease to exist.

However, are you forgoing your privacy for a wage? In a capitalistic system, what would happen is that if you were not willing to hand over your privacy, then the job would go to next highest bidder and you would have to find your merry way. Is this right? Is it fair?

In the paper mentioned in section two, certain precautions were mentioned that would prevent an employer from viewing a specific engineers' details and only look at the broader picture. In this case the manager would be respecting the individual's privacy and would also benefit from this new found information.

Legality

This is where the legal system would step in and ensure that any activity between manager and engineer would be done respecting the engineer's right to privacy.

This is a tricky legal situation to implement however, as we have seen all too many times, that when a corporation is given an inch it will take a mile. Therefore, it is critical that the government would be very strict with such measures.

Conclusion

In this report we have covered four main areas within the idea of measuring software engineering. We have looked at the ideal and possible implementations of a software engineering measuring tool. We have looked at platforms that implement such tools and we have look at how those tools could be implemented. Finally, we had a look at the ethical and legal issues that arise when discussing such a topic.

Bibliography

A. Sillitti, A. Janes, G. Succi, and T. Vernazza, "Collecting, integrating and analyzing software metrics and personal software process data," in Proceedings of the 29th Euromicro Conference. IEEE, 2003, pp. 336– 342.