Measuring Software Engineering

Measuring the productivity of a software engineer is a deceptively difficult task. The complexity of modern software systems can lead to a corresponding complexity in the engineering of appropriate solutions to a given problem. As a result, a software engineer needs to have both a wide range of appropriate skills, and a deep understanding of the job they are working on. On top of this, they must also contribute to the actual implementation of a solution, and work closely with other team members to ensure the smooth and efficient development of the software systems. These kinds of requirements are by their nature difficult to accurately compare and measure.

In this essay, I will explore some of the possible solutions to this problem. I will look at four main topics, measurable data, computational platforms, algorithmic approaches, and ethics.

Measurable Data

The first issue which must be overcome when measuring a software engineer's productivity is what data we are using to study their work. In the past, somewhat simplistic measurements were considered. These included things such as measuring the lines of code written.

Using lines of code to judge a software engineer's contribution to a project is deeply flawed. The simplest case is where an engineer with more lines of code than their peers is considered the most productive member of the team. In this system, software engineers are placed in a situation where they are competing to add the most lines to a solution. This immediately runs into two major issues.

The first is that engineers could create code that is artificially longer to make themselves look better in comparison to others. This could be as simple as adding extra spacing in functions, to using longer versions of certain productions to inflate their line count. For example, using a full if statement for a conditional expression where the ternary operator '?' would suffice. In the most extreme examples of gaming the system, some engineers could decide to make lines of a program take up as much space as possible by adding new lines after each token in a statement, (assuming a language where whitespace and indentation doesn't matter to the compiler, e.g. Java). We now have a situation where attempts to measure productivity has led to a decrease in the quality of the code, and this leads into the next point.

Having more lines of code in a solution is often undesirable. Shorter solutions are often preferred as they are more concise and usually more efficient. In addition, a software engineer who fully solves a problem in ten lines shouldn't be considered less productive than one who solves it in fifty. As measuring the opposite, the less lines of code written the better, would be absurd, it is clear that we need a different approach.

Cyclomatic complexity is another data measurement that could be used to measure software engineering. Simply put, cyclomatic complexity is a measure of the complexity of some code, obtained by seeing how many different paths one can take through a given program. The idea of measuring cyclomatic complexity is to see if a program is overly complicated. If two software engineers successfully solve a problem, and one of their solutions is less complex than the other, it is taken as being the better solution. Proponents of using cyclomatic complexity to measure the quality of code say that more complex code leads to a higher probability of bugs and other undesirable errors. However, this has not been definitively proven, as cyclomatic complexity naturally increases with program size, as does the number of errors.

Code churn is a measurable piece of data that can be used to predict defects in software systems. Thus, it is useful in measuring the software engineering process. Code churn is a measure of the change to a particular section of code over time. High code churn occurs when a section of code is modified at a high frequency. This can be a useful measure even on its own to see whether useful additions are being made to the codebase.

Code coverage is a valuable form of data that can be used for measuring software engineering. Code coverage is a measure of how much of a program is run when testing is carried out. This is an essential measurement if your team is using Test Driven Development. One hundred percent code coverage is the ideal, this means that every line is executed at least once during a test. We can could use this metric to say with certainty that a particular software engineer has thoroughly investigated and tested their produced code. As unit testing is very important to ensuring the quality of the codebase as the project moves forwards, comparing code coverage for different engineers can give a valuable insight into their thoroughness. If one of them is not sufficiently testing their code, it can be highlighted as a weakness in their skills. One shortcoming using code coverage as a metric is that high code coverage doesn't translate directly into bug free code. If the tests themselves are insufficient, then even full code coverage won't mean good code. An engineer could write tests which pass implicitly, but also call all methods needed for high code coverage. These methods could be buggy, but the tests will still pass. This must be addressed if code coverage is to be taken as a metric.

Recording the software engineer as they work is an alternative solution to the measurable data problem. This tactic can be used both in the academic sphere (when conducting studies into developer performance), and in industry. By recording the software engineer's every action, or more thoroughly their entire screen, a full review into their performance could be done by a superior. Unfortunately, this method carries with it several ethical questions, which I shall discuss in the "Ethics" section. On the other hand, having full access to an engineer's every action would allow a full evaluation of their abilities in context. This could be done with an examination of all other data measurements mentioned here simultaneously. The downside of this is that it would require a large time commitment, particularly in large organisations or teams. Some managers may consider this a worthwhile endeavour.

An alternative to the above approaches is to look away from data derived from the code itself, and to look at the software engineer instead. There is evidence that in general, a happy worker is a more productive worker. According to S. Achor's "Positive Intelligence" in Harvard Business Review, as cited by "Measuring Happiness Using Wearable Technology" in Hitachi Review, happy workers are 37% more productive than those who are unhappy. Could this measurement be applied to software engineers, and how could happiness be measured and quantified? One solution given by the above paper is to provide workers with some form of wearable sensor. This sensor measures physical activity taken by the wearer. Their argument is that physical activity during the workday corresponds to increased happiness, which in turn leads to increased productivity. If we follow this argument, we see that assigning breaks with physical activity, which could be measured by the device, would lead to more productive software engineers. If an individual engineer appears to be unhappy, they could be given access to some resources to help improve their mood and work ethic. While this doesn't give us precise measurements of the efficiency of a given engineer, it could be used to ensure the overall success and wellbeing of an entire development team.

Time taken to complete a task is a different potential data measurement. Under this scheme, the software engineer that gets their job done first would be considered the better employee. While it is true that meeting any assigned deadlines is critically important, it may not be the case that the quicker implementation is the best. A well thought out piece of code that takes some time to write is better than a quicker but worse solution. Encouraging engineers to rush through a problem could lead to even more issues down the line, when bugs that should have been sorted out earlier emerge elsewhere in the program. It could also lead to burnout amongst the team. Therefore, while the time taken to complete tasks should be considered an important measurement, particularly with time critical problems, it should not be considered in isolation.

As has been shown above, there are a large number of potential measurable sources of data when assessing the efficiency or quality of a software engineer. It is up to the mangers of a team or the heads of a company to decide which metric is most appropriate to their use case.

Computational Platforms

When measuring software engineering, there are a variety of platforms available for conducting the measurements. These range from opensource tools and resources, to offerings from the likes of Microsoft. In every case the basic goal is the same: to gather data and then represent it. Tools exist which cover most of the previously discussed data measurements.

One of these platforms is the opensource Hackystat reference framework. Hackystat describes itself as a facility for “software project telemetry”. In essence, Hackystat automatically collects a large series of data measurements, which can be accessed easily by the project team. The creators of Hackystat argue that it can be used to identify the source of problems added to a project. The example that they provide is that an increase in defect density is matched by an increase in complexity, then action should be taken to reduce complexity and see if this results in an improvement.

To achieve this so-called software telemetry, Hackystat requires access to software engineers' text editors or IDEs, and to any build tools that they use. It gathers the data as work is done, and stores it in a web server. Hackystat can use this data to generate graphs representing various metrics. These graphs can be compared to each other to try and find any noticeable patterns during the engineering process. Some of the metrics that can be viewed in this manner are code churn, lines of code committed, the active time on code before it was committed, the number of build attempts and failures, and information about the time of a commit. All of this data can be represented for both the software project as one project, and for the individual software engineers working on it.

Remote repository services such as GitHub, GitLab and Bitbucket can be used as platforms to gather useful metadata on software projects, in addition to their roles as spaces for collaboration and backing up of code. The basic use of these services is entirely free, usually only more advanced features require payment, such as unlimited private repositories, or advanced workflow features, or superior technical support. One massive advantage of using these services is that they all automatically track various metrics for each repository and each commit. They all also provide handy graphs which represent the gathered data in an easily visualised way. If you find that these graphs aren't sufficient, you can make your own. For example, you can query GitHub's API to gather various data points, and then create your own graphs, maps, and other graphical representations of the data. This automated data generation is very convenient for managing the project as a whole, not just tracking individual contributors.

Some of the particular metrics that GitHub can gather and visualise are the number of lines of code (both added and removed) by each developer, when a contributor committed their changes to the codebase, the exact lines edited by the contributor, and the frequency of commits to the repository by every single software engineer working on the team. These metrics, and many other possible metrics, are all extremely valuable to any project manager or team coordinator. They provide an insight into the efficiency and quality of a software engineering team. They can enable a decent overview of the work contributed by any individual engineer, as discussed in more detail in the "Measurable Data" section of this essay above.

There exists a range of software products which review code quality. These can detect a number of "code smells". One of the things measured by the Sonarqube code quality checker is the before mentioned cyclomatic complexity. A software engineering process can be enforced to follow Sonar's conventions, which will lead to the complexity of the code being measured, in addition to the other features of this tool.

Various integrated development environments (IDEs) contain their own testing suites. Microsoft's Visual Studio is one of them. As part of the test functionality, Visual Studio contains a tool which can be used to calculate code coverage for each tested method. This can provide an engineering team and their managers with direct feedback as they write and design their tests. In addition to Visual Studio, other IDEs have features that enable the calculation of code coverage in a series of tests. For example, a code coverage plugin for the Eclipse IDE called EclEmma exists. Both of these tools could be used within software teams to ensure that the software engineering process is being supported by proper unit testing. Such tools exist for most popular programming languages. In each case, the general concept for representing code coverage is the same.

As I have shown, the range of computational platforms for gathering data that relates to the measurement of software engineering is quite large and diverse. Anyone seeking to measure the quality of it has a large assortment of options to choose from.

Algorithmic Approaches

Algorithmic approaches to this topic exist in a variety of forms. These include algorithms for calculating the data being measured, to algorithms which change the engineering process to allow measurement.

One metric whose calculation algorithm is quite simple is code churn, provided that the code being analysed is being put in a repository such as GitHub. Calculating code churn is as simple as adding up all added lines, modified lines and deleted lines for a certain section, or for a certain commit. These values can be easily obtained from the repository and used for the calculation.

However, we can go further. In the visualstudio.com/learn/using-simple-code-churn-metric-find-software-bugs/ link, the author defines code churn in a more comprehensive way. They use the idea of an active file, which is a file which is being frequently modified. The argument is that files that are active are experiencing high code churn, which is likely due to a poor design. This can be used to algorithmically determine weaknesses in the design. This is done by first defining what an active file is, (a subset of all the files, which contains those being frequently changed) then defining recurrently active files. They define these as those active files that have changed during every window of time, in a certain defined time period. The conclusion of the article is that recurrently active files are more likely to be less understood, hence the high code churn, and thus lead to more bugs. This algorithmic approach can be used to see which files are currently problematic.

An algorithm to calculate the rough cyclomatic complexity of a program is quite simple for programs which do not use goto, jump, branch or any variation of that statement. All that you have to do is start by setting the cyclomatic complexity to one. Then you add one to this value for each loop, and each condition in an if statement. The actual mathematical definition given by Thomas McCabe (who came up with cyclomatic complexity) in his 1976 paper is: "cyclomatic complexity V(G) of a graph G with n vertices, e edges, and p connected components is $v(G) = e - n + p$". Calculating this value requires constructing the control flow graph. While possible for individual methods, small sections of code or small programs, this will get too complicated to do by hand at larger scales. It should then be calculated using some tool.

One very interesting, but different approach is to "gamify" the entire software engineering process. This idea basically means to add common elements of games to the software engineering process. The goal of doing so is to make the process of software engineering more enjoyable while simultaneously encouraging best practise. For example, in L. Singer and K. Schneider's paper on the subject, computer science students were encouraged to commit frequently to source control. They did this by turning the process of committing into a game, in which people gained achievements for committing a certain number of times (which they called milestones). They sent weekly team updates to the students, in which team members commits were compared in a kind of leader board. All of these are features common to many video games. While some of the students viewed this change negatively, it did lead to some positive outcomes, such as an increase in commits when they were ready rather than as one big update.

So how can gamification be used to also measure the process of software engineering, and individual engineers? We can turn the process into a kind of RPG as shown in "Turning Real-World Software Development into a Game". This is the idea where completing challenges assigned to you allows you to "level up". Then we can use the idea of competition and achievements to encourage best practise. While the engineer is completing their assigned tasks, they will be providing valuable information which can then be used to assess the overall state of the project. For example, engineers could be challenged to have as little complexity as possible, or complete tasks the fastest, or to have the lowest code churn, or who generate the most testing code with complete code coverage. Then the people who best complete these challenges could be placed on an internal leader board. All the while the data from each challenge is being compiled and analysed for recognisable patterns. As shown in the study, when interesting or unexpected results occur, such as one project succeeding more than others, or certain tasks being completed more than tasks of a different variety, then the company can investigate the issue. This leads to a better understanding of the software development and maintenance process.

Other algorithmic approaches include things such as mining data from source code, bug reports and code comments, among other sources, to build up a detailed view of a piece of software. No matter which one we are referring to, these algorithmic approaches provide an insight into the software engineering process in a deep way.

Ethics

Of course, there are a number of ethical concerns which must be considered when measuring software engineering. These can range from measurements which infringe on the privacy of the software engineers being assessed, to concerns over too much personal information being stored. No matter the size of the issue, nor the size of the team concerned, these ethical questions should be raised whenever measurement of a team is being considered. Critically, they should then be sufficiently addressed before any actual monitoring or examining of employees is started. This is essential to protect the rights of all of the employees or team members working on a software project. If they are not addressed, then the happiness and wellbeing of the workers could be negatively affected. In an extreme instance, the company itself could face difficulties retaining talented staff, or even end up in an unwanted court case. Obviously, it would be best for both employers and employees to avoid such drastic problems.

One of the methods used for measuring the performance of software engineers was to record a video of each engineer's screen while solving a task. This was the methodology used in "How Effective Developers Investigate Source Code: An Exploratory Study", a study that was carried out to compare what successful software engineers did when modifying a software project, to unsuccessful engineers. If this method was to be extended from the academic environment to the workplace, it would carry with it numerous ethical concerns. Primarily, these concerns would be privacy issues. Recording everything that an employee does could be compared to having a manager standing over their shoulder as they worked. It is quite possible that this could negatively impact the engineer's mental state, which would ironically lead to a decrease in productivity. This would also make it completely unworkable for an employee to work from home. Such a workplace would result in a negative environment for staff. It could lead to negative publicity for the employer, and could discourage skilled engineers from applying for jobs there. Not to mention the potential comparisons to George Orwell's "1984" that may result from this decision. If this idea was to be properly implemented, it should be reserved for instances where it is absolutely necessary. Additionally, all software engineers employed by a company with this practise, both current and potential, should be made aware of this monitoring being in place.

A less extreme version of the above subject is less extreme monitoring of software engineers work. This would be a system such as Hackystat which monitors data from the software engineer's tools and text editors to form its database of statistics. In this case, the information being gathered pertains almost exclusively to the task of software engineering. Assuming that everyone using the system is informed of the data gathering, and that the data is stored securely, this method is far less ethically concerning.

Choosing a poor or inaccurate metric for measuring software engineering may raise unintended ethical issues as a side effect. If an insufficient methodology is being used to assess the contribution of team members to a software project, it may well lead to those who do pull their weight being singled out as poor workers. If this were the case, those employees could end up being fired, or being passed over for a pay rise or promotion, despite actually contributing a great deal to the team. This is clearly an undesirable outcome for everyone involved.

Related to the previous issue, another ethical concern is that focussing too much on metrics can lead to an increase in stress in the engineering team, as they are forced to meet certain number of requirements per week. In the worst case, the whole team could end up focussing more on meeting requirements, rather than actually solving the task at hand. This might be detrimental both to the team's efficiency, but more importantly, to the mental health of the group of software engineers. It is important to check that best practise is being followed, and to identify the source of any issues that arise during production, but it should not come at the cost of increased stress to the workers on the team.

Another ethical issue we should consider is who has access to the details of an individual employee. With the potential for so much information about a software engineer's working life to be gathered and stored for later evaluation, companies could build up enough data on individuals to paint a picture of their overall lifestyle.

This could have massive ramifications were there to be any sort of leak or data breach of these stored records. This is especially chilling if we considered the fact that wearable technology, such as that promoted in the Hitachi source, takes the gathering of data to a more personal and intimate level. One possible negative outcome of a data breach in such a case would be an employee with a less active lifestyle suddenly facing higher insurance premiums, if their insurance company was to get a hold on the activity data. Marketing companies could end up circulating the employee's personal information, as they seek more data for targeted advertisements. The level of infringement of the software engineer's privacy of such an event would be catastrophically high. This level of a crisis could also be in breach of data protection law, as much of the data being stored could be considered "sensitive personal data". Even if we disregard the legal aspect of a data breach scenario, the affect it would have on a software engineer, (or indeed any employee), would be extremely negative.

The ethical matters discussed in this section are not insurmountable, but they should all be seriously discussed before any plan for measuring software engineering gets off the ground. If they are not sorted out early on, then it could lead to a failure in protecting the rights and health of the software engineers in question.

Conclusions

In conclusion, there are a wide variety of methodologies for measuring the software engineering process available today. However, no one approach stands out as universal best practise, nor is there any de facto industry standard.

Personally, I believe that a combination of gamification (to improve the engineering process and help gather data), and a variety of measurable data, (collected by a system similar to the Hackystat telemetry facility), would be best for measuring the overall software engineering process. All of the data gathered should be carefully selected, and stored securely to lower the risk of unethical violations of the team's individual privacy.

Bibliography

- Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. 2004. How Effective Developers Investigate Source Code: An Exploratory Study. IEEE Trans. Softw. Eng. 30, 12 (December 2004), 889-903
- http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf
- N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. Software Engineering, 2005. ICSE 2005.
- http://blog.ploeh.dk/2015/11/16/code-coverage-is-a-useless-target-measure/
- http://www.literateprogramming.com/mccabe.pdf
- http://www.eecs.qmul.ac.uk/~norman/papers/defects_prediction_preprint105579.pdf
- E. B. Passos, D. B. Medeiros, P. A. S. Neto and E. W. G. Clua, (2011) "Turning Real-World Software Development into a Game," Games and Digital Entertainment (SBGAMES), 2011 Brazilian Symposium on, Salvador, 2011, pp. 260-269
- L. Singer and K. Schneider, "It was a bit of a race: Gamification of version control," Games and Software Engineering (GAS), 2012 2nd International Workshop on, Zurich, 2012, pp. 5-8.
- https://martinfowler.com/articles/useOfMetrics.html
- https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0
- Johnson, Philip M., et al. "Improving software development management through software project telemetry." IEEE software 22.4 (2005): 76-85
- https://www.visualstudio.com/learn/using-simple-code-churn-metric-find-software-bugs/
- https://docs.sonarqube.org/display/SONAR/Metric+Definitions
- https://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity
- https://msdn.microsoft.com/en-us/library/jj159340.aspx#sec12
- Hassan, A.E. and T. Xie, Software intelligence: the future of mining software engineering data, in Proceedings of the FSE/SDP workshop on Future of software engineering research2010, ACM: Santa Fe, New Mexico, USA. p. 161-166
- http://www.eclipse.org/community/eclipse_newsletter/2015/august/article1.php
- http://www.hrmagazine.co.uk/article-details/the-ethics-of-gathering-employee-data
- https://dataprotection.ie/viewdoc.asp?DocID=1467&ad=1
- https://www.dataprotection.ie/docs/A-guide-to-your-rights-Plain-English-Version/r/858.htm