Measuring Software Engineering

Objective: To deliver a report that considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethics concerns surrounding this kind of analytics.

Introduction

The idea and practise of measuring the productivity of software engineers poses several issues on the accuracy and reliability of data, and the convenience and ethics surrounding their collection. A software engineer recording their own workflow, progress and accomplishments at a certain project will inevitably be skewed and biased towards seeming more productive. On the other hand, an automated software monitoring an engineer at work might miss some crucial stages of the development process such as planning, discussions, and problem-solving.

However despite all of these challenges, measuring software engineering is still necessary for us to be able to point out the activities that makes us most and least productive, and optimise the way we work based on these information. There have been many attempts regarding the measurement of the software engineering process. They have evolved from pure and simple manual collection of data to automated tools linked to editors, build and test tools, and build servers.

The software development process might seem immeasurable at first because of the uncertainty of the process itself. Even software engineers admit to be bad at predicting what a project will need and how much time is required to complete it. However, looking at the bigger picture, the software process is a repeatable timeline of actions, and a trend of workflow can be seen throughout various types of software development practises.

In this report, we will explore:

1) the different approaches to software development being used today,
2) its measurement and the ethics surrounding it,
3) different opinions on what aspects of it should be measured,
4) how it has been being measured in the past and the present,
5) how companies encourage engineers to improve productivity, and
6) how this can affect the future of computing.

The Software Engineering/Development Process

Software Development Process is dividing the production of a software into different stages in order to have a better understanding of the product and project management and ultimately to find ways to improve each stage. This may refer to developing a new product from scratch or taking an existing product and modifying it. The source of the project does not make a huge difference in terms of the stages that needs to be done to push a product into production.

No matter the outline of a software process may be, it will always involve the following key stages:

1) Software Design and Specification
2) Implementation
3) Testing and Validation
4) Evolution

Software Design and Specification involves the planning and designing of the main functionalities of a product. It is unlikely that code is written at this stage, however, design documents are written and proposed to the client or to a manager. These proposals undergo reviews and rewrites until it is pushed to the Implementation stage.

The Implementation stage is when the software is being programmed and functional parts are completed. The product at this stage usually contains a lot of bugs and flaw in the system. This is why the Testing and Validation stage is a must. The product is used the way it is intended to in order to ensure that it fulfils the functional requirements, and it is rigorously tested, reviewed and fixed.

No piece of complex software is ever truly finished and perfect, but there has to be a baseline for when a software is ready to push for production. When these requirements are met, the software is approved and is rolled out for consumers to use. However, in certain products, the development process does not stop there. It has to be monitored while it is being used by users who are external to the development team, and when bugs are reported, they have to be fixed. And over time, features need to be improved to suit the evolution of the consumers' needs.

These stages are approached by different engineers and product teams in different ways. The most common way is A*gile*. In Agile development, the product continually evolves based on the collaboration of the product teams and the consumers. It uses flexible planning, development, delivery and continual improvement. It is based on an incremental and iterative approach.

In contrast to Agile is the Waterfall model, which is plan-driven and all activities and tasks are planned first before execution. This usually works for products where evolution is not required and requirements do not change over time. However in practise, it is hard to treat software development as a linear process because the stages involved usually overlap and needs to be repeated at certain points more than once.

As a computer science undergrad, I have had to experience going through the software development process multiple times, whether on my own or in a team. It is no doubt easier planning and programming when you have complete control over every aspect of your software. However, in a bigger scale, what a team of five can do in twelve weeks might take a single programmer thirty weeks to complete. So realistically, we all need to work in teams in order to create a product as efficient as possible. This leads to one of the biggest questions in engineering: How do we measure the software development process in individuals and in teams?

Measurements and Its Ethics

Measuring the software engineering process can improve software development decision-making, provide a way of assessment, judge a program's quality and ensure a software's reliability. Diving into more details, software development involves a lot work and is more than just coding. In most cases, engineers and managers will initially need to meet with a client, write a design proposal, review and rewrite the proposal, get approval, write the software, test the software, fix bugs, and write the software again. And in between all of these are meetings, discussions, and learning a programming language, a tool or a new storage architecture.

Software development is a complex process that has no one-for-all easy solution. More importantly, there is a concept in measuring software process which says that the easier and less controversial the metric is collected, the less useful it actually is. So if we want to be more accurate and thorough in our assessment of the software process, we need to consider the useful yet more ethical way of collection.

A case study of more than six hundred software professionals has shown that only 27 percent viewed metrics as 'extremely' or 'very' important part of software project decision-making. It is also not surprising that these software metrics are often only used for cost and time estimations, areas that can be easily measured by statistics.

Engineers have different opinions on how software development should be measured. Some believes that it should be measured by the efficiency of their team meetings on top of the time spend writing code. There should only be a fixed about of reasonable time spent on discussing the project. If more than needed is spent, that means the team meeting is not well-planned or that engineers are having conversations that should be taking place outside the meeting.

Most engineers also believe that the time spent to code completion should be the focus of the development measurement. This completion time encompasses the initial draft of the code, the review phase, and the fixing and rewriting phase. An engineer's ability to produce production-grade functional software is believed to be the most crucial part of the process.

Another aspect of software development that is often unnoticed is the nature of the subtasks divided among a team of engineers. It may not seem like it but these subtasks play is huge role in decreasing or increasing the time spent in development. Essentially, the bigger and more complex the subtask is that a single engineer is working on, the possibility of this task blocking the other tasks also increases. Ideally, these subtasks should only take around six hours to complete. More than that will result to other engineers not having productive work to do.

Meetings, time spent to code completion and length of subtasks are just few of the broad ways of measuring software engineering. Additionally, the measurement of the process can be divided into two aspects such as qualitative and quantitative metrics. Qualitative measure the reliability, effectiveness and usability of the software in an abstract way, while quantitative uses numerical values to calculate what metrics say about productivity.

Another quantitative metric that is familiar to all engineers are counting *Lines of Code or Thousand Lines of Code* written for a given software. Although this certainly says how much of the program has already been written, it is not a dependable way of measuring the quality. For example, a program written in assembly will almost certainly be longer than the same one written in a high-level language such as C or Java. In this case, LOC is an insufficient basis and should not be used in isolation.

Furthermore, LOC cannot measure the complexity of the code and the effort of the engineer responsible. It is like pitting a person who has written a two-page essay about quantum computing against another who has a five-page report about their love for cupcakes. Both are completely different things and requires different amount of research and preparation. This is why LOC is rarely used on its own but combined with other forms of assessments.

An engineer's Lines of Code is straightforward and easy to measure. It is non-invasive and it doesn't require manual work at the age of powerful editors and IDEs. There is a concept in measuring software process which says that the easier and less controversial the metric is collected, the less useful it actually is. So if we want to be more accurate and thorough in our assessment of the software process, we need to consider the useful yet more ethical way of collection.

Counting the amount of bugs in a software is also another way to predict the productivity of a software engineer. It is assumed that the more focused one is at a given task, the less mistakes are made. But similarly to Lines of Code, bugs can be subjective. Small bugs like a failure to account a minor test case do not have the same level of magnitude as failure to return the correct result from a feature.

The cost of production also needs to be taken into account when measuring an engineer's productivity. Someone with a better workstation that is set up accessibly is likely to work faster than someone who works in an outdated and slow machine. For example, a team in a big company like Google will have more resources than a small team composed of college students with little resources.

Examples of qualitative metrics are customer satisfaction and how useful a software is compared to similar others in the market. If a customer leaves a satisfied review, it says a lot about the product and ultimately how the product was made. Also, if the product does much better than another product with the same specifications, it is credited to the productivity of the team.

Many research projects and companies have attempted to solve the issues revolving the measurement of the software development process. Some have created manual processes while others have combined manual and automation. At the end of the day, both processes will still have to deal with the ethics involved the collection of data and monitoring of an employee.

Algorithms and Platforms

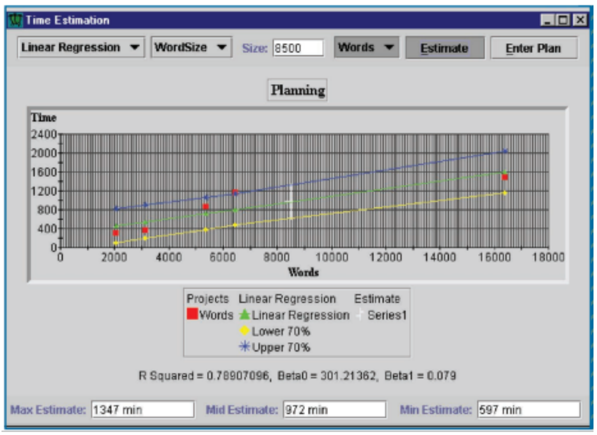
Software

Different processes have been developed to measure software engineering, and various platforms have adapted these processes and have invented their own to be used by big and small companies. An example of one that is non-invasive and accessible is by looking at user Git repositories. It provides a timeline of the code development and reveals how much work an engineer can produce at a given amount of time. Platforms like Github and Bitbucket can be used to find out an engineers skill level, work process and programming skill. However, it is not a complete picture of the development process of a given project, especially when we have to consider the non-technical aspects as well.

The Personal Software Process or PSP is development processes that helps engineers understand and improve their performance by tracking their development of code, both predicted projection and the actual results. PSP aims to better an engineer's ability to estimate and plan their projects, make action plans than they are able to execute, and lower the amount of bugs in their code and generally, write better code. In Watts Humphrey's book *A Discipline for Software Engineering*, it implements PSP using spreadsheets, manual data collection, and manual analysis. Developers need to fill out forms like project plan summary, time-recording log, a code checklist and more.

Due to the manual collection of PSP, the data collected seems fragile like 'lighting a candle'. It is vulnerable, exhaustive and can be controlled. A software that can collect PSP data automatically might be more reliable than the manual method. Given this, the manual nature of PSP has its perks. It encourages users to navigate their analytics and choose ones that best suits them. However, the conclusion of PSP can still be incorrect even with a low error rate, and to solve this problem, Leap (lightweight, empirical, anti-measurement dysfunction, and portable software process measurement) toolkit was developed.

The Leap toolkit was designed to address the data quality problems from PSP by automating and normalising data analysis. It does not prescribe a sequence of analysis and it allows the developers to control their own data files. It is also portable because it can be moved from one project or organisation to another. It also provides time estimation based on historical data as shown on figure 1.



Size: 8500

R Squared = 0.78907096, Beta0 = 301.21362, Beta1 = 0.079

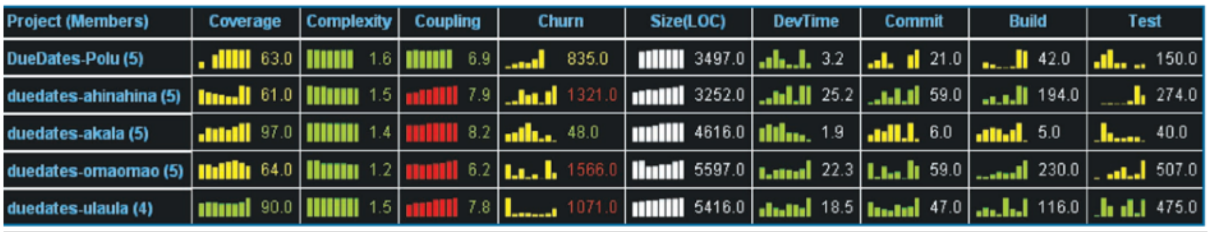
Max Estimate: 1347 min Mid Estimate: 972 min Min Estimate: 597 min

Figure 1. The time estimation component in the Leap.

However even with Leap, it is concluded that PSP can never be fully automated and would require significant manual data.

When developing these type of measurement platforms, a common strategy is to figure out the data collection and analysis, and how to achieve them first. But a tool called Hackystat was developed the opposite way in which it focused first on collection of software processes and data with little or no overhead for developers. It implements a service-oriented architecture by gathering data through the collectors attached to development tools such as editors, build tools, test tools and build server, and then data is sent to a server, where it is analysed. It is also unobtrusive which means users wouldn't notice if they are being monitored. It also supports a measurement called buffer transition which detects when a user switches from one buffer to another.

Hackystat can be used for both personal and team development. It is also high-performing, test-driven, provides *project health* status and visualises software project telemetry. It measures the DevTime, time spent on development, number and size of commits, numbers of builds, and how often it is tested. A disadvantage of Hackystat is that some developers consider it as a bug, it is invasive and it causes interruptions at work.



Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (5)	63.0	1.6	6.9	835.0	3497.0	3.2	21.0	42.0	150.0
duedates-ahinahina (5)	61.0	1.5	7.9	1321.0	3252.0	25.2	59.0	194.0	274.0
duedates-akala (5)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0
duedates-omaomao (5)	64.0	1.2	6.2	1566.0	5597.0	22.3	59.0	230.0	507.0
duedates-ulaula (4)	90.0	1.5	7.8	1071.0	5416.0	18.5	47.0	116.0	475.0

Figure 2. Hackystat Software ICU.

Hackystat was a product of the movement that aimed to produce a telemetry-based project management software. Software telemetry is a style of software metrics definition, collection and analysis, that contrasts Personal Software Process (PSP). In software telemetry, data is automatically and regularly measured with timestamps accompanying every entry. Instead of solely relying on the data value itself for predictions and monitoring, it focuses on the changes in the metrics. Software developers and managers can continuously access the data relating to their projects, which encourages decentralised project management in the team. This style aims to predicts defects and detects early warnings of anomalies considering the complexity of the program.

Software telemetry has paved the way for more automatic and service-based architecture software measurement platforms in the public market. The development of software for the analysis and measurement of software engineering has gained popularity over the years, delivering products from Atlassian, CAST, Parasoft, McCabe, Coverity, Sonar, and others. Most of these current products are configurable and has a defect-tracking system and a visualised User Interface. They also provide easy installation and integration. All these tools have accepted some tradeoffs in their systems including expensiveness, simplicity and social acceptability.

Atlassian is one of the most popular choices among companies that provide software development and collaboration tools. It is best known for Jira, an issues detecting system, and Confluence, a team collaboration and wiki product. It has also acquired Trello, which is a web-based project management application.

Unlike Atlassian, SonarQube (formerly Sonar) is an open source platform that automatically inspects and reviews code quality to detect bugs, code smells and security vulnerabilities. It works on over twenty programming languages and reports code duplication, coding standards, unit tests, code coverage and complexity, and system vulnerabilities. It is technical-based and does not provide any support for team collaboration or project management. However for an engineer who solely want to focus on their code quality, this free tool should be enough.

Some companies like Parasoft also provides automatic software testing. They claim to combine end-to-end software test automation with analytics. Automating testing aims to lighten the load for developers and testers when testing their software and allow them to focus on more complex things that currently cannot be automated. Similar to Parasoft, a tool from McCabe also does software testing and security analysis. But both companies do not provide tools for non-technical aspects on team and project management, which are also vital part in the software engineering process.

Hardware and Wearables

In addition to the manual and automatic process of measuring software engineering, there are multiple ways of tracking an engineer's workflow and routine while they are in the office. One can argue that the more time an engineer spends on his desk, the more productive they become, because their desk is where most of the software development happens. However, in a way, tracking the amount of time people spend at a certain place can raise privacy issues, but this doesn't stop employers from doing so.

The Badge Location System is one way of tracking where employers are at certain times of the day. Badges are the most popular way of controlling access to an office building. Most companies ensure security with badge readers on every entrance, exit, office floors and restricted rooms. There is usually a policy in every workplace that demands an employee to always have a badge with them at all times whenever they are in the workplace. This may seem like a harmless request at first, but there could be cases where these badges have tracking hardware that send signals to sensors in order for security to keep a record of where you currently are and where you go when you are at work.

Tracking, as a supplement to the telemetry tools monitoring an employee's workstation, might seem like a good source of data when it comes to keeping track of productivity. Having said that, this may not be the case for all types of employees. While software developers are expected to work with their machines most of the time, managerial roles are expected to be a different case. As managers, they have to attend multiple meetings a day, and go in and out of the office to meet clients.

Based on my personal experience, where a person is at a given amount of time cannot tell how productive they are being. When I was an intern at an internet company with hundreds of engineers based in Dublin, I noticed that the most productive people did not spend their entire day on their desks. There were meetings and interesting talks to attend do and casual conversations about issues different teams have faced or are facing. Although, while I was working, I learned a lot programming on my desk, I still value the conversations I had about privacy laws, compression and data management with experienced and intelligent engineers while I was at the micro-kitchen making some tea. Therefore, I believe the most productive moments of an engineer does not necessarily happen at their workstations.

There are also personal time tracking products commercially sold like Timeular. It's an octahedron-shaped object with each side corresponding to a task. The idea is that whenever a user switches task, they need to put the side mapped to that task facing up, so that the device would synced into their smartphones, log the time and starts a timer. With enough data, the app can give the user reports and statistics on tasks they have worked on and which ones they spent time on the most. Furthermore, it is a personal device that does not track any real-time location, which erases any concern for privacy.

Industry Practises

In an industry setting where there are thousands of engineers working on the same codebase, tools are in place in order to ensure the code quality of every commit. These tools are also used to track an employee's contribution to the company. Most companies use a Git-like system where an engineer can pull a repository or a part of it and make changes on their local workstation. Whenever an engineer commits, a log is written into the commit which specifies the time and date when the committed changes were completed. The more useful commits the engineer has at a set amount of time, the more productive they have been.

Commits can also have messages where engineers can write a description of the changes or added features they have made. Writing descriptive commit messages is a widely known good practise. How an engineer is at describe their work is also a testament to how good of an engineer they are. A commit message must be understandable, even to the ones who have no deep understanding of the technologies and libraries are being used in the program.

Good commit messages also makes Code Reviews an ever better experience for the engineer making the commit and the others who are doing the reviews. Code Reviews are also widely used in industry to minimise the amount of common bugs in programs. An automatic testing tool usually checks for syntax error, code coverage and compliance to coding standards during Code Review, and other engineers in the company spend time reading through the code and pointing out vulnerabilities and mistakes in the program. The amount of time an engineer's code is going through review is an evidence of the quality of code they write and how good they are at detecting bugs actively while writing their code. This is another metric that can be considered for measuring a software engineer's productivity.

Improving the Software Engineering Process

Gamification

They say that if you love your work, it wouldn't feel like work at all. Companies are taking this idea to another level by bringing turning work into play. They are incorporating video game features into simple tasks to make them seem exciting and instantly rewarding. These features include giving employees badges and points for completing jobs, which are then presented to the company's internal community in a form of a leaderboard. This tactic encourages friendly competitions and initiates conversations in the workplace.

Gartner, a technology-centred research company, estimates that seventy percent will use the gamification technique in at least of their processes. A popular process in which companies integrate gaming into is their training platforms. Completing training courses gives the employee a points as a reward, which then unlocks more training courses that are more complex. Training courses, usually about company policy, privacy, proper conduct and the like, can usually become a little boring over time, and so bringing the reward system into it changes its whole mood.

Another process where a game-like structure is being adapted into is the version control. When computer science students are given a web-based tool where version control is turned into a game with a point system and leaderboards, they are more inclined to write code and make more commits to surpass their peers. Given that humans have a competitive nature, this isn't surprising especially when young students enjoy seeming impressive.

Although gamification of mundane tasks in the office increases engagement, it doesn't solve problems in bad management. In some cases, even gamification is not enough to promote better quality of work, or in the case of software, better code.

Future of Computing

Software Engineering and Artificial Intelligence

It is important to ask ourselves what the effects of the new technologies being developed to the measure of software engineering and to software engineering itself. These days, people are talking about artificial intelligence. Today, we have Machine Learning as an application of the study of artificial intelligence in our current technology. As computers get better at learning and studying behaviour, they might be able to conclude a more accurate pattern of what leads to the production of good software. Even though this may sound all good at first, we also need to consider how much these machines can conclude about us, the users.

So far, there has not been a lot of software engineering tools that makes use of Machine Learning as a core technology for the metric analysation process. And there is also no reason why, aside from privacy concerns, researchers cannot explore this area. Maybe if we can get artificial intelligence to review our code, they will be able to do it faster and it will make our code output higher.

Conclusion

‘If you can’t measure it, you can’t manage it’ is a dogma that much of the measurement of software engineering is motivated by. Companies’ need to continuously produce revenue-generating products fuel the research for better solutions to predicting when a software can fail, when a team does not work well together or when a project’s result is not what is expected. There are certainly aspects of software engineering that can be measured by numerical metrics and can be analysed through statistics. But there are also the immeasurable cases that involve authentic human connections and soft skills. Ultimately, measuring the software engineering process is still beneficial to certain parts of development, and the better we become at doing it, the better quality of software we produce.

References

Websites

- https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc
- https://en.wikipedia.org/wiki/Software_development_process
- http://www.citeulike.org/group/3370/article/12458067
- https://timeular.com/?v=d2cb7bbc0d23

Video

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

Research Papers

- Johnson, Philip M., et al. "Improving software development management through software project telemetry." IEEE software 22.4 (2005): 76-85.
- Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157.
- R. Want, A. Hopper, a. Veronica Falc and J. Gibbons. The active badge location system. ACM Trans. Inf. Syst., 10(1):91–102, 1992.
- 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.

Print

- Silverman, Rachel (Nov 2, 2011). "Latest Game Theory: Mixing Work and Play — Companies Adopt gaming Techniques to Motivate Employees". Wallstreet Journal.