<u>Measuring Software Engineering Report</u>

Cillian Fogarty - 19334609

1 Introduction

Software engineering has become a crucial part of everyday life with everything from coffee shops to sports clubs having some amount of software engineering incorporated into their operations. With how prevalent software engineering is in our society and the high demand for more software engineers, the question of can software engineering be measured is one that needs to be answered.

Measuring the productivity of anyone's work can be tricky, but particularly in software engineering where there are many variables beyond the code solution itself and all work can't be measured in the same way. To measure or analyse anything, many variables and how each of them relate to each other must be taken into account. The importance and value of some variables far outweigh others, this is what makes the idea of measuring software engineering such a complex challenge.

What exactly determines the value of someone's work? Is it possible to measure it? Can it be done to any degree of accuracy? Is it the amount of work done or is it the time taken or something completely different? These are the questions this report will explore and attempt to answer. In general a good worker is an efficient worker, this holds true for software engineering, meaning the real question to be addressed is, what metric can be collected and analysed to determine a developer's efficiency.

2 Measuring Software Engineering

In order to be able to measure software engineering, first an understanding of the various project structures and development dynamics such as individual or team projects must be established. Software engineering has developed over time to better suit client-developer communications. Now there are two main types of software development models, the older style of the Waterfall model[1] and the newest style of the Agile development model.[2]

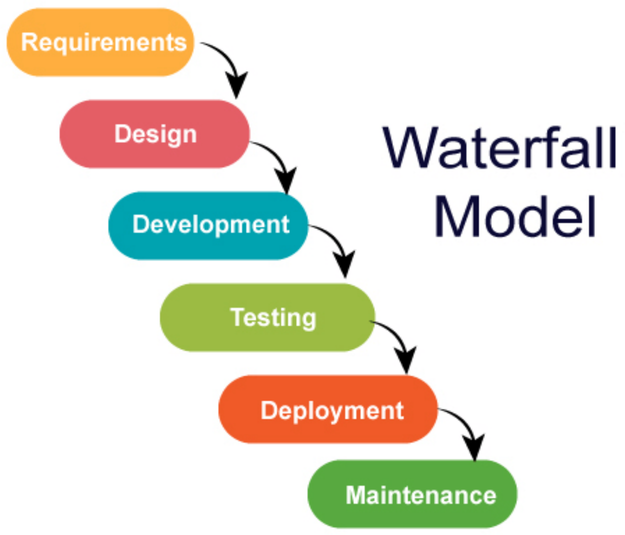

Figure 1: The Waterfall Model[1]

The Waterfall model was one of the first types of software development models.[1] The ideology of the model is one of a simplistic straightforward flow from start to finish. Each section is seen as a separate stage of the project, the project does not move to the next stage until the current stage is fully completed.[1] This is a very simple and functional approach, leading to straightforward testing as the requirements are set out at the beginning of the project and do not change. This also leads to simplicity in understanding which stage of development the project is in, as only one stage is occurring at a time.

The primary advantage of this model is that with the clear flow from one section to the next, software developers can focus on perfecting each step of the project before moving to the next step. This leads to no backtracking to a previous step occurring, as all the requirements for each step are set out in advance. This tends to create a smoother running project and ensures the client is delivered exactly what they asked for in the requirements step.

However, over time this linear approach to development became restrictive and unsuitable as the basis of client-developer communications changed. Clients began wanting to alter requirements as the project progressed, these requirement changes could lead to entire steps of the model having to be repeated. This need for flexibility meant the waterfall model quickly became outdated as demand for quick and easily changeable results grew. The Waterfall model was built on the concept that clients set out the requirements at the start and were then delivered a product at the end, with little communication in the intermediary stages.[1] The change to constant communication between the two parties and changeable requirements rendered this model unusable.

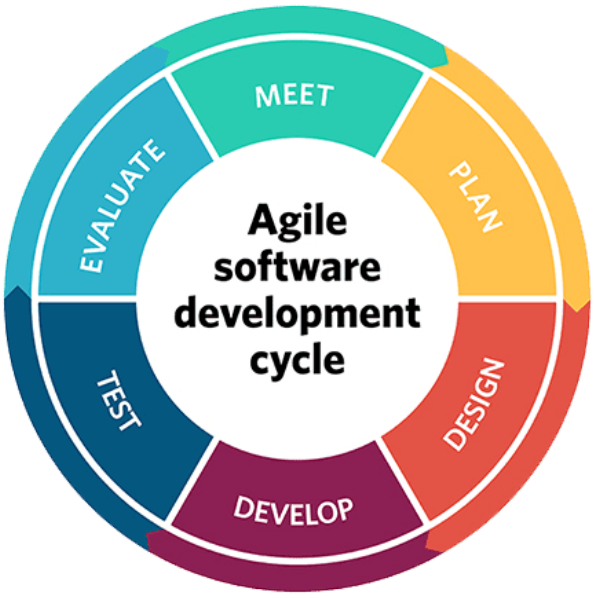

Figure 2: The Agile Development Model[2]

The Agile development model was created in response to these developing needs and has now become the most common development model used in software engineering. This model allows for constant change as a cyclical development approach is used, where small sections of the project are worked on at a time, with an evaluation process at the end of each cycle to determine if changes on what has already been completed needs to occur.[2] This means constant communication between the client and developers can occur and both parties are kept up to date with the progress of the project in real-time, allowing for easy flexibility of the design. Agile development allows for the creation and response to change to occur, meaning the client gets exactly what they want and developers waste less time on developing code that inevitably gets removed from the project, as progression is done in small increments. This flexibility comes with many great advantages but also a few disadvantages, since it leads to uncertainty and unpredictability of how the project will progress. A small project can quickly become a much larger project than anticipated as the clients requirements can be easily changed.

2.1 How Can It Be Measured

Following on from the brief outline of the most popular software development models, it is clear that measuring the efficacy of the work being completed is more difficult than just a simple look at the work done vs the time taken. This is due to the fact that the project can be quite flexible leading to continual revisions to what has already been done, this wouldn’t be seen by an observer only seeing the current stage of the project. A view of how the project developed must also be taken into account. This often leads to a heavy reliance on the number lines of code written or number of commits to a repository by a developer in a certain period of time being evaluated. This is a deeply flawed approach as only the quantity of code is taken into account and the quality of the code is completely overlooked. These measures are quite easy to manipulate[17], as a developer could choose to use a much longer and most likely less efficient solution to ensure they get a higher line count. A programmer solving a problem in 20 lines of code would be seen as better than a programmer solving the same problem in only 5 lines, although in reality the programmer using only 5 lines is more than likely the better programmer. Similarly a programmer could choose to use a series of micro-commits to ensure a higher commit count.

These metrics, while useful to get an overview of the individual or team, should not be used as the sole metric for measuring software engineering. These metrics can however be used to analyse a developer's refactoring of a codebase[3] and code churn,[4] which would prove much more useful especially when compared to the average of the team. This form of measurement was the basis for my design for the Github-API and Visualisation assignment.

The amount of time taken to solve a problem could be a useful metric, but then the size, complexity and importance of that problem must also be taken into consideration to ensure a fair evaluation of various developers.[5] While these metrics are still and always will be used, they are best used in compliment to other metrics which provide a more accurate representation of a developers efficacy.

A further difficulty is that no two projects are the same, meaning comparison between development teams at multiple companies or even at a single company is quite challenging. Further adding to this is the varying skill sets of each individual developer, each of which is valuable in their own right and adds to the overall team dynamic, but means comparing them against another is difficult, when metrics such as skillsets outside of physically completing a programming task cannot be taken into account. The greatest challenge in measuring software engineering is the diversity between each and every developer. A complete evaluation of a developer's efficacy may never be achievable, however there are methods to get a good general idea of it, which will be further outlined in the next section.

2.2 Measurable Data

In order for the measured data to be beneficial to a company or development team, the importance of easy collection of that data cannot be overstated. Ensuring that the process is as simple as possible will ensure it does not burden the team, leading to continuous measurement rather than it not being done in favour of completing actual work on the project. This is where automation shines, both ensuring no extra work to the developers and ensuring accurate, untampered data is collected. The measuring and analysing process should not be burdensome or cause any pauses in development while the data is being processed.

Several important characteristics of measurable software engineering metrics are that they are computable, consistent, adaptably, an accurate representation of the development process and most importantly easy to understand once processed and compared to other data sources. The simplest questions to ask when determining a useful metric to analyse is, how could this data be used to accurately determine the productivity of a developer? No two developers are the same and similarly no two projects are the same, so will the measurement of this data be a fair and universal metric? If the answer is no to either of the questions previously posed, then the metric is unlikely to be useful in measuring software engineering.

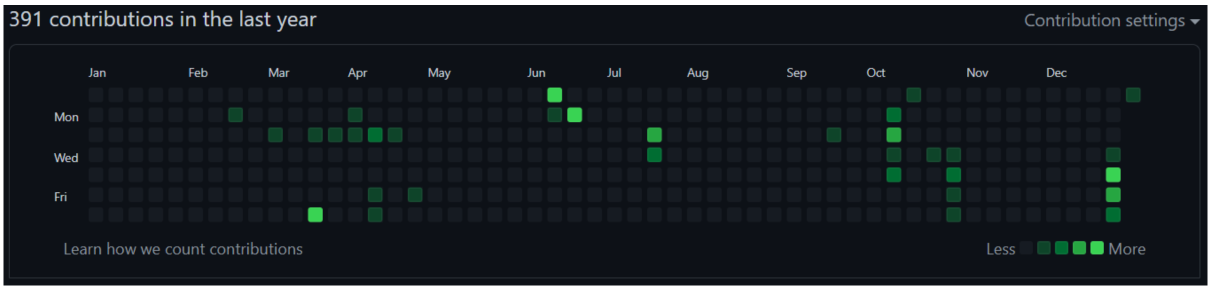
391 contributions in the last year

Figure 3: An example of a Github contribution history[6]

Take for example the commit history of a user shown above. This data shows the contributions of the user across an entire year, but is it a useful or fair measure of the user's productivity? This data is quite useful when analysing the activity of a developer, but as outlined earlier this data is quite easy to manipulate with a series of micro-commits when one commit would have done.[17] Looking at just the number of commits isn't very useful unless being compared against the other team members. The quality of the code is far more important, this is where refactoring[3] and code churn[4] measurement comes in. By comparing the line additions and deletions in each commit against the team's average data, a much more representative metric of a developer's productivity is found. As when compared to the team average many line additions or deletions can be a sign of a sloppy developer.

Two types of metrics often used when measuring software engineering are 'Size-Related',[7] meaning the size of the outcomes from an activity and 'Function-Related',[7] meaning the amount of useful functionally added. This is useful when deeper analysis of code complexity and code functionality is desired.

Many tools exist to analyse and compare code data, but few take into account the additional elements involved in developing the code in that data. When measuring software engineering, analysing the code itself should not be the only metric, factors such as planning, organising, designing and meetings are crucial to delivering a successful project. These simple factors can save hours or even days of work, especially further down the line. The importance of these factors cannot be ignored, but unfortunately as they are not easily measurable or comparable metrics they are often left out of the equation when measuring software engineering.

3 Existing Platforms / Resources

Further to the exploration of the various ways in which software engineering can be measured, attention should now be drawn to the different ways in which companies measure software engineering and the technologies that make it possible. Services such as Github[8] or Gitlab[9] which incorporate the Git[10] system, offer version control and are a treasure trove of data on how the project was developed just waiting to be extracted and analysed. Approaching from a different angle, technologies such as Fitbit[11] allow health and wellbeing monitoring of a developer. While the health of a developer may not be often considered as a measure of software engineering productivity, it is one that should be investigated. Each of these technologies measure different aspects of a programmer's productivity, while Git provides data on the work created by the developer, Fitbit offers data on the developer themselves.

Git is by far the most common form of measuring software engineering, primarily used as a version control and repository system, which allows multiple developers to contribute and work on the same project together. Git not only provides that functionality but also data on what a developer changed in each commit, how many lines of code were added, how many lines were deleted, comment tracking, languages used and a whole lot more.[10] This allows for the frequency of commits and the quantity of code added and deleted in each commit, to be easily tracked. These metrics on their own are a good measure of the activity of a developer, without being compared against other developers, it is of little use as the quality of the code cannot be established. When compared to others the refactoring of the code base and code churn of the developer can be measured and compared to others to establish if it is significantly different.

Open source software such as SonarQube[12] can be used to analyse and determine the quality of the code written. This software offers the ability to check code to ensure it is bug free and healthy. Continuous analysis of the code base means these errors are flagged immediately and can be rectified, thus saving time finding/correcting bugs and ensuring an efficient solution is developed. SonarQube allows connecting directly to a git repository,[13] meaning no additional work is required to be done by the developers to get their code analysed. Auto issuing of assignment of tasks can help streamline the process of resolving issues as each developer is given a set of tasks to complete in order to fix bugs found or improve code health. This integration directly with Git means incorporating SonarQube into existing development processes is straightforward and easy, adding huge benefits as the code is automatically analysed. This measurement of software engineering ensures that all developers are writing quality code rather than just a large quantity of code, reducing the ability to manipulate the system[17] and ensuring the final solution is as efficient as possible.

A seemingly obscure but very useful platform in measuring software engineering is health trackers such as Fitbit. These have been growing increasingly more popular both in personal use and in company wide use in recent years. They allow the measurement of a person's health, a particularly important metric in the era of the Covid-19 pandemic as health has become a common topic both in society and in the workplace. The move to remote work has meant many do not leave their home at all during the day, and with the reduced amount of meeting with friends, there has been a great increase in sedentary lifestyles since the beginning of the pandemic. Companies are now beginning to incentivise workers with free or discounted Fitbits[14] to help both keep them active and to allow the company to analyse the health and fitness data of it's workers. It is well known that being fit and healthy is a key factor in productivity as it is greatly linked with mood and focus, both of which are immensely impacted by the health and fitness of the person.[15] Managers or a processing system would be able to analyse a worker's Fitbit data such as their daily steps, average heart rate, water intake and sleep quality to help recognise problems before they are too great and could be used to help boost the wellbeing of the workers by identifying stress heavy areas and reducing the exposure to those areas or type of work. This would allow the company to be proactive in the wellbeing of the developers and other workers, by improving the facilities of the workplace whether it be a gym, massage room, sleep pods or therapist, or by subsidising access to these services. This tracking of such personal data of a developer has many ethical concerns which will be addressed in a later section, but this data could prove far more beneficial than detrimental. The effectiveness of analysing this kind of data is still being researched, but the health and wellbeing of workers is a major area of concern in the current unprecedented time of the Covid-19 pandemic.

4 Available Algorithms

Several metrics of software engineering and some available resources to aid with the collection and analysis of the data have been discussed. Later the ethical issues and considerations of that data collection and analysis will be discussed, but firstly the various methods and algorithms that can be used to calculate these metrics and produce useful interpretations of the data will be addressed.

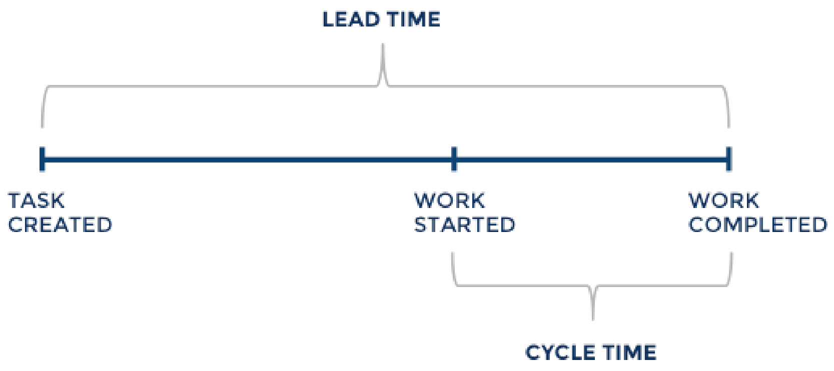

Figure 4: Cycle Time Metric[16]

Software engineering metrics focus on four main categories; Specification, Design, Coding and Testing.[18] Analysing these metrics allows a manager to achieve a very good insight into tagged productivity of the team of a particular software engineer. One of the most popular algorithms for analysing these metrics is Cycle Time.[16] This allows the amount of time spent working on a particular issue to be analysed. These issues can be further broken down into types, such as bugs and feature requests. This algorithm allows for analysis on the whole process rather than just the code written, making it a good measure of efficiency from start to finish, with specification, design, coding and testing included. Including these aspects is crucial, as with any issue there are steps that must be taken such as planning, organisation and initial design before the coding can begin. The Cycle Time is often a true reflection of the time taken to complete a solution, the whole way from idea to release. This true reflection of the time taken means it can be easily compared to similar issues or projects. When this process is repeated over many issues it allows the development team to gather a much more realistic estimate of how fast new features or bug fixes can be delivered. Bottlenecks in the process that could be affecting the team's performance can more easily be found, since the entire time from start to finish is analysed instead of just the coding stages.

Another algorithm that can prove very useful in measuring software engineering is Throughput.[19] This indicates the total value-added work done by counting tasks, chores and bugs within a released feature. Similar to Cycle Time, Throughput gives a realistic view of the entire amount of work completed within a feature from start to release, this can then be used to estimate the time for future features to be completed. Since no two features are exactly the same, instead of measuring the entire workload of a feature, measuring the different tasks within each feature will allow for a more accurate representation of the productivity of the developer to be established and a better estimating of the time required for future features to be established.

In the wise words of Bill Gates "Measuring programming progress by lines of code is like measuring aircraft building by weight",[20] any metric that uses lines of code as the primary data source must be approached with caution. This is due to the fact that lengthy and not concise code is incentivised, leading to bad practices where code is bulked out with repetitive or meaningless lines of code and comments. A far better metric is Logical Lines of Code over time,[19] which gives a meaningful measure of the coding efficiency of the developer since only logical lines are included. Using a measure of the logical lines prevents the incentivisation of unnecessarily long code as the additional lines do not benefit the measure, in fact they are more likely to bring the measure down as more time is spent writing those extra lines. This approach avoids bad habits and encourages cleaner, more concise code giving a truer measure of the efficiency of a developer.

Generally the same base theory of inputs in the form of code written or time spent divided by outputs in the form of features created, is shared between all these algorithms. Unfortunately having this as the basic cost analysis form used for the algorithms, means that often analysis can only be fully done after all of the work has been done. This is because all the data must be known to achieve the true result, as such approximations must be used when determining how long a task or feature will take to be fully completed. In essence the problem is how accurate we can get these estimations, and the more data previously analysed that can be compared against, the more accurate it will be.

5 Ethical Issues

Many ethical concerns are raised when the topic of measuring software engineering is addressed. In the previous sections an outline of how to collect and analyse data in order to measure software engineering was addressed, but now the most important topic needs to be addressed. Should this data be collected and analysed?

Data is the most important aspect in accurately determining the productivity of a software engineer, but the interpretation of that data is almost equally as important. At what point does the collection of data and continuous scrutiny of a developers work become damaging to the developers wellbeing.[21] The collection of any data can be invasive not only to the developer themselves but also to the project as a whole, the more strenuous this data collection and analysis is the more invasive it will be. The fear of being constantly watched and having every aspect of the work done being analysed can have profound mental health effects, leading to increased stress levels, anxiety and fear of failure.[21] This can greatly affect the workflow of the developer but more importantly can damage the bonds between the developer and other team members and managers. This can greatly affect the quality of the work done, leading to further scrutiny and increased stress levels. It really is a rabbit hole of darkness that can be all too easy to fall into.

The ethics of every piece of data proposed to be collected and analysed must undergo rigorous investigation prior to collection beginning, to ensure the wellbeing of the developers are not compromised and must always stay as the priority. Ensuring the analysis is only to improve productivity of the developers must be done and clear communication with the developers must be established to prevent the feeling of always being watched and analysed.

Adding Fitbit data into the mix muddies the water even further, as now the fear of the developers health and fitness being scrutinized can further lead to mental turmoil. Employers knowing so much about a worker's health, can lead to the fear of being fired if the company decides they are not healthy enough and do not want to take a risk on them continuing working there. How much employee data is too much data? What are the metrics used for? How lenient are they? Is the data being collected for the right reasons?

The primary concern is how invasive and critical the analysis will be and how it'll be interpreted. Data collections such as these can easily lead to a damaged work environment if not done carefully, leading to rifts between teammates with everyone working as an individual trying to outcompete their colleagues in a fight to keep their job. This would not be the first time concerns of a toxic work environment in tech giants would be raised, with the surge in these companies trying to orient their employee's lives around the workplace with gyms, restaurants, beds, cinemas, coffee shops and much more now being incorporated into the workplace it sounds like a dream come true to some, but to others could lead to even more concern as more and more aspects of their lives are now being watched and put under scrutiny.

Clearly there are a large number of unanswered ethical questions on the measuring of software engineering that need to be addressed. Personally I think the concerns over what could be done with the data far outweigh the benefits it could bring, especially if brought to this extent outlined above. These tech giants already know so much about our lives, adding more data to that and increased scrutiny over developer productivity can be quite damaging to the mental health of the developer, especially in the era of the Covid-19 pandemic where stress levels are far higher than usual, with the uncertainty in the world and mental health problems are increasing by the day. Giving this much extra data may not seem like much today, but how will things look in five years or in ten years if each year we allow more and more data to be collected.

<u>**6 Conclusion**</u>

There are many metrics and algorithms which can be used to measure software engineering, but each comes with its own set of problems too. Care must be taken to ensure the methods used to measure the productivity of a developer accurately and fairly represent their work, there is no one solution to this problem and each development team must choose the methods which suit them best.

As I outlined in the final part of the ethics piece, while I feel the risks of data being analysed to the extent discussed in this report would be far too invasive and detrimental to the developers wellbeing, if the correct balance between data collection and privacy can be found and strictly followed, where the goal is on helping the developer improve rather than just pointing out the flaws in their approach, then and only then do I believe the benefits could be reaped leading to more efficient software engineers.

7 References

1. Jayathilaka C. Waterfall vs. Agile Methodology [Internet]. Medium. 2020. Available from: https://medium.com/@chathmini96/waterfall-vs-agile-methodology-28001a9ca487

2. Denman J. Agile FAQ: Get started with these Agile basics [Internet]. SearchSoftwareQuality. 2020. Available from: https://searchsoftwarequality.techtarget.com/feature/Agile-basics-FAQ-Getting-started-with-Agile

3. Wikipedia. Code Refactoring [Internet]. Wikipedia. 2021. Available from: https://en.wikipedia.org/wiki/Code_refactoring

4. Azure DevOps. Analyze and report on code churn and code coverage - Azure DevOps Server [Internet]. Microsoft. 2021. Available from: https://docs.microsoft.com/en-us/azure/devops/report/sql-reports/perspective-code-analyze-report-code-churn-coverage?view=azure-devops-2020

5. Hilska O. Measuring Software Development Productivity [Internet]. Swarmia. 2021. Available from: https://www.swarmia.com/blog/measuring-software-development-productivity/

6. Github. Account Overview [Internet]. Github. 2021. Available from: https://github.com/cillfog1

7. Foster EC, Towle BA. Software Engineering : A Methodical Approach. Boca Raton: Auerbach; 2021.

8. GitHub. About GitHub [Internet]. GitHub. 2021. Available from: https://github.com/about

9. GitLab. About GitLab [Internet]. GitLab. 2021. Available from: https://about.gitlab.com/

10. Git. About Git [Internet]. Git-scm.com. 2021. Available from: https://git-scm.com/about

11. Fitbit. About Fitbit [Internet]. Fitbit. 2021. Available from: https://www.fitbit.com/global/us/about-us

12. SonarQube. Continuous Inspection [Internet]. Sonarqube. 2021. Available from: https://www.sonarqube.org/

13. SonarQube. GitHub Code & Quality Analysis | GitHub Integration [Internet]. sonarqube. 2021. Available from: https://www.sonarqube.org/github-integration/

14. Fitbit. Fitbit for Corporate Wellness [Internet]. Fitbit. 2015. Available from: https://www.fitbit.com/content/assets/group-health/FitbitWellness_InfoSheet.pdf

15. Sharma A, Madaan V, Petty FD. Exercise for mental health. Primary care companion to the Journal of clinical psychiatry [Internet]. 2006;8(2):106. Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1470658/

16. Linnanvuo S. Agile & Lean Metrics: Cycle Time [Internet]. Screenful. 2015. Available from: https://screenful.com/blog/software-development-metrics-cycle-time

17. Harding B. 17 popular software engineering metrics, and how to game them [Internet]. GitClear. 2021. Available from: https://www.gitclear.com/popular_software_engineering_metrics_and_how_they_are_gamed

18. Lafleur J. How to Use Software Productivity Metrics The Right Way [Internet]. DZone. 2019. Available from: https://dzone.com/articles/how-to-use-software-productivity-metrics-the-right

19. Goebelbecker E. How to Measure Lines of Code? Let's Count the Ways [Internet]. NDepend. 2018. Available from: https://blog.ndepend.com/how-measure-lines-code-lets-count-ways/

20. Gates B. Measuring Programming. Microsoft; 2009.

21. Sandoiu A. The effects of perfectionism on mental and physical health [Internet]. Medical News Today. 2018. Available from: https://www.medicalnewstoday.com/articles/323323