1 Introduction

Being able to quantify the software engineering process is a tough task to accomplish. With there being so many aspects to software engineering, it is very difficult to get an accurate measurement of a software engineer where every aspect of the job is taken into account. It is important to have a metric that measures accurately so someone non-technical within the company/organisation can judge the quality of work of any given software engineer. As this field is becoming more and more crucial to companies around the world, engineering efficiency is quickly becoming a big challenge to overcome and subsequently a lot of investment and effort has been put in to create tools and methodologies to provide this information.

The goal of a software metric is to provide someone outside the software engineering process, meaningful and understandable information on what progress is being made and the quality of that progress, where no technical knowledge is required to fully understand its meaning. This information can allow for better planning for future projects, team organisation and provide insight on software deadlines to improve efficiency.

2 Why use Metrics?

The benefit of metrics is to determine the quality of the current worked on product and the development process. It's an important aspect for any company for quality assurance of their product while also providing management benefits, increased productivity and estimating the cost of software, by giving a clear insight of the quality and pace of the software development process. The management can utilise this information to have a better overview on development teams. It allows managers to identify possible problem developers or project issues and to communicate these issues through a standard understandable measurement. Using these metrics they could possibly fix problems before they even arise. The metrics help management to be able to more effectively meet the objectives and project goals as well as provide them with a more accurate and informed time frame for completion.

While these metrics are extremely useful to management, they also can greatly benefit the engineers themselves, allowing them to more effectively measure their own productivity and quality giving them the ability to adjust to requirements. With teams being able to view their own metric, it provides them the opportunity and the knowledge of how they're performing, whether they need to improve their quality, or simply the knowledge that their team is performing well, its an extremely useful insight for the developers.

3 What Makes a Good Software Metric?

A software metric is a measure of software characteristics which are quantifiable or countable that allow software engineers to identify software performance, software quality, measuring productivity and provide insight to effectively plan work items.

A software metric must be easy to collect or it will not be done, these metrics should have several important characteristics to make them useful as metrics. An ideal metric would be easy to understand, not burdensome that it gets in the way of writing code and effectively quantify the quality of the product.

- **Simple and Computable:**
  Performance metrics must be understandable to both developers and non technical employees. Everyone must know what is being measured, how it is measured and what can be done to make their measurement better.

- **Consistent and Unambiguous**
  Consistency is key in any metric in order for it to be of any use as consistency allows for comparison, prevents confusion and adds clarity.

- **Independent of Programming Languages**
  The metric must be separate from the programming language and provide the same information regardless of the language used.

- **Easy to Calibrate and Adaptable**
  The metric must be measured and calibrated in a way that is understandable to those who are being measured and must be able to adapt to every project that is being developed.

- **Easy and Cost-Effective to Obtain**
  The benefits of the metrics must outway the cost to obtain them.

4 Types of Software Metrics

There are many different approaches one can take when trying to measure the software engineering process. There's measurable data which one can use to provide insight into the process, as well as working methodologies that are designed to provide a

structure to how teams work and give teams the information on their work rate so they can improve.

4.1 Measurable Data

There are ways that the software engineering process is measured using data. These metrics are production metrics and attempt to measure how much work is done and determine the efficiency of the development team.

- **Active Days**
  This metric is a measure how how often a developer is actively making changes/progress such as through commits. This is only measuring the time that actual code is being written and does not account for planning or other tasks. This is useful for measuring how often other tasks are interrupting the creation of new code. An example of this metric would be the GitHub active days metric.

- **Efficiency Metric**
  Efficiency metrics try to measure the efficiency of a developer. Code churn can be used to provide information on the efficiency, as a developer with low code churn could be highly effective.

- **Lines of Code**
  Lines of code is a very simple metric that can measure the size of a program. It is the most basic, oldest and most widely used of software metrics to measure the productivity of a team. It requires little to no effort to obtain this metric as version control applications such as GitHub already provide this functionality. Therefore it can be used as a quick estimate at how productive a development team is working. Lines of Code is believed to be not that useful as it is too restricted and does not account for anything else in the process and can cater to certain types of programming styles. For example, Developer A writes long, confusing code which accomplishes their given task, while Developer B writes the same code in a concise way and works perfectly. In the eyes of the Lines of Code metric, Developer A is the more efficient developer while Developer B's code is much more maintainable and arguably better. Language dependency can be an issue here as one line in one language may not be the same as a line in another. This creates an inaccuracy in comparison between them.

  The lines of code metric is very simple, too simple in fact. However, this does not mean it's useless. By extending the Lines of Code to include other factors such as lines of modified or deleted code, as Code Churn, it can provide a much better and useful metric.

- **Code Churn**
  Code churn represents the number of lines of code that have been added, changed or deleted over a set period of time. An increase in code churn could indicate that the development project could need attention. Code churn has several uses.

  1. **Visualise the Development Process**
     By graphic the code churn, it allows you to see the frequency and consistency of commits. Spikes in this graph could indicate problems or stress in development teams.

  2. **Delivery Risks**
     If the code churn increases on the lead up to a release, this can indicate possible post-release defects and bugs. Code churn can be used here as a warning sign if when approaching a deadline and the code churn increases. It's a sign that your code becomes more and more volatile as you get closer to release and the increased workflow may cause bugs. This is un-ideal and you want the code churn to remain consistent the closer you get to release.

  3. **Identify Problems in the Process**
     Code churn can indicate to problems that might be present in the development process. Unclear requirements may increase code churn as developers try to meet the requirements without fully understanding them can lead to constant changes being required in the code. The difficulty of the problem can be seen in the code churn as more and more code may need to be written to solve a problem. Code churn could also indicate a lack of engagement from a developer; if their code churn is high but throughput is low.

- **Code Coverage**
  Code coverage is a measure used to describe the degree to which the source code of a program is executed what a particular test suite is run. A program with a high code coverage has more of its source code executed during testing.

  Test code is crucial to ensure the code is fully functioning, faulty code can be bad for the reputation of companies as well as being extremely risky when the code is used in safety-critical situations such as health-care. Code coverage can be used as a metric to check the completion and polish of a code base.
  There are different coverage criteria in Code Coverage;

  1. **Function Coverage**
     Function coverage is a measure of the number of functions or subroutines that have been called. It ensures that all functions are called in the test suite.

2. **Statement Coverage**
   Statement coverage is a measure of the number of statements or lines in the program that have been executed. It ensures that all the statements have been executed to limit bugs in the code.

3. **Branch Coverage**
   Branch coverage checks if each branch in an if else statement has been executed.

4. **Condition Coverage**
   Condition coverage checks if each boolean condition has been evaluated to both true and false.

- **Test Coverage**
  Test coverage is an indicator on how well a team has reached the functional requirements. These tests are written to meet the requirements and not the code. By testing the test coverage, the progress that has been made towards the functional requirements can be measured. It is also useful to the developer to see what other test cases must be met. An example of test coverage would be Unit Testing, where tests are written to check if a function performs as expected.

- **Technical Debt**
  Technical debt or code debt is a concept that reflects the implied cost of additional rework caused by choosing an easy but limited solution now instead of taking a better approach that would take longer. If technical debt is not “repaid” it can accumulate over time, making future changes more difficult and time consuming to implement. Technical debt is not inherently a bad thing and is sometimes required to move a project forward.

  Technical debt is expressed as a ratio. A ratio of the cost to fix a software system (Remediation Cost) to the cost of developing it (Development Cost), this is how we get the Technical Debt Ratio; TDR.

  Remediation Cost can be expressed in terms of time, if a team makes Remediation Cost the time it takes to fix issues within a code function, RC is then directly proportional to the cyclomatic complexity of that code function. So it becomes easy to determine how long it would take to fix entire files once we have their cyclomatic complexity.

5 GitHub

GitHub provides a few tools to allow for rudimentary software metrics. GitHub is a version control software, consisting of commits. Commits can be used as a software metric, the frequency of commits and the time of commits can be used to see efficient times for a development team. The software metric of Active Days as discussed in the previous section can be obtained through GitHub as well as lines of code and code churn can be calculated as well.

6 Agile Process Metrics and Jira

The agile methodology, is not a metric on the quality of the software/product itself and rather is used to improve the development process. These metrics focus on how development teams plan their work and make decisions. Jira Software is a management tool for agile projects and supports any agile methodology, one such methodology is scrum. Scrum is an agile methodology where products are built in a series of fixed length iterations called Sprints. There are three artifacts involved in Scrum;

- **Product Backlog**
  This is the entire list of work to be done which is maintained by the product manager. It's a dynamic list that is constantly being added to and changed.

- **Sprint Backlog**
  This is the list of items from the product backlog that are selected for the current sprint cycle. Before each sprint, in the sprint planning meeting, the team chooses which items will be in the sprint from the product backlog.

- **Increment**
  This is the sprint goal and is the usable end product from the sprint.

There are 5 main stages that a scrum agile development team may undertake, this is simply a guideline that a team may follow and is not a requirement.

1. **Organize the Backlog**
   This is the job of the product manager, and the backlog list needs to maintain the goal of the product vision and must dynamically change to the need of the users. Therefore this list is maintained using feedback from the users of the product.

2. **Sprint Planning**
   This is organising the scope (work to be done) for the current sprint, this is done during the sprint planning meeting by the entire development team. This meeting is led by the scrum master where the team decides on the sprint goal. Stories or sometimes called issues, are added to the sprint if they align with the sprint goal and are deemed to be feasible to implement.

3. **Sprint**
   The sprint is the actual time period where the work is being done. Two weeks is the typical length of time for a sprint to produce a valuable increment. The benefit of a short sprint is the reduction in unknown unknowns in the current sprint stories as the total workload and complexity of a short sprint is smaller than that of a longer sprint.

4. **Daily Scrum**
   This is a daily meeting, usually at the start of the day that is short and concise, where the goal is to get everyone on the same page, with the sprint goal clear in their minds. These are short, generally 15 min, where team members state what they did yesterday, what they plan to do today and any problems that have or may arise. This is the time to voice any concerns you have with meeting the sprint goal or any blockers to it.

5. **Sprint Review**
   This is at the end of a sprint, where the development team gets together to review the increment. Here the backlog issues are now “Done”, and the increment is released. Here the product manager updates and changes the product backlog to reflect the sprint which can then feed into the next sprint.

Jira provides many tools for agile development. Jira’s tools for sprint planning include; easy backlog grooming to be able to easily re-prioritize user stories by simply dragging and re-organising, story points are easily tracked on Jira. The scrum board is used to visualise all the work that is given in a current sprint. These can be easily customised on Jira for better workflow. Jira also provides scrum reports to visualise the increment in various ways and gives development teams the ability to see what went wrong and where they can improve for future sprints.

7 Algorithmic Approaches

- **Halstead Complexity**
  Halstead complexity measures was developed to measure a program module’s complexity directly from source code. It was developed as a means of determining a quantitative measure of complexity directly from the module. It is based on

interpreting the source code as a sequence of tokens and classifying each token to be an operator or operand. This is then counted;

- Number of unique operators = n1
- Number of unique operands = n2
- Total number of operators = N1
- Total number of operators = N2

The program length can be calculated using;

$$N = N1 + N2$$

The size of the vocabulary is the sum of the number of unique operators and the number of unique operands;

$$n = n1 + n2$$

The program volume is the information contents of the program, which is measured in mathematical bits and is calculated as the program length times the 2-base log of the vocabulary size;

$$V = N * \log_2(n)$$

The volume V describes the size of the implementation of an algorithm. As V is calculated based on the number of operations performed and the operands, it is less sensitive to code layout than lines of code metrics.

Difficulty level is proportional to the number of unique operators and describes the error proneness of a program.

$$D = (n1 / 2) * (N2 / n2)$$

Effort to implement E is proportional to the volume and the difficulty level of a program

$$E = V * D$$

Number of Delivered bugs B correlates to the overall complexity of the software.

$$B = (E^{2/3})/3000$$

From Halstead's estimates, delivered bugs in a file should be less than 2, however this is not the case and most of the time there are more bugs than B suggests. This is an example of how this is not a great metric.

- **Cyclomatic Complexity**
  Cyclomatic complexity is a metric used to indicate the complexity of a program. It's a quantitative measure of the number of independent paths through a software's source code. This is computed using a control flow graph, nodes represent groups of commands in a program and the directed edge connects two nodes if one node could be executed directly after another node. This method can be applied to individual methods or classes. An application of cyclomatic complexity is to determine the number of test cases that are necessary to achieve thorough test coverage.

8 Ethical Concerns of Software Metrics

Using metrics and measuring developers will undoubtedly have an effect on the behaviour of those being measured. The measurement of something gives it an importance, this may make people work better and more effective by itself in an attempt to increase their rating, however if the metrics themselves are not utilised correctly this can lead to developers working in a way that will improve their rating but not necessarily improve the quality of their code. Therefore it may be more beneficial to measure the performance of a team rather than the individual so developers are focussed on the improvement of the team rather than focussing on themselves to the detriment of the team.

The more and more metrics that are being used are a cause for concern. Some developers see this as an infringement of their privacy as there is nothing stopping the recording of metrics other than strictly software metrics, such as time spent during lunch, as and indication of the efficiency of a developer. This is obviously worrying as it creates a stressful environment that is solely focussed on productivity and efficiency which is likely to increase the chance of developer burnout.

The use of metrics will have an impact on the psychology of a software engineer, while managers hope that it will be a positive affect where developers can see their progress and provide them with structure, it might also have the opposite effect. These harmful impacts can be due to the effect of increased stress and focus on performance in the workplace. This is a negative impact as developers may become overwhelmed and unhappy in their job reducing the overall effectiveness and efficiency of the individual but also the team.

Another concern is how companies use the software metric data. If companies begin to solely rely on the metrics to determine the worth of an engineer(s). This is a flawed approach as an engineer in a team can be negatively affected by the work of other engineers in the team.

A concern that stems from this is the gaming of a software metric system, with all the added stress that metrics can introduce, some individuals may resort to trying to game the

system. For example if an agile team or individual is working as hard as they can but are asked to increase velocity, the story points estimate will increase to give the illusion of increased velocity when in reality the same amount of work is being done. The same can be done if the team or individual is asked to complete more stories, then they will simply create smaller metrics. These examples of gaming the system are good for no one as it erodes the trust between developers and management creating a negative environment for actual work to get done due to the pressures that software metrics can introduce.

However, there can equally be an argument that metrics can help software engineers understand their importance to the company and that they are valued by management. That the metrics give the engineers a boost and desire to work better to increase their rating. I believe that this can definitely be true if the correct combination of software metrics are used, that encompass as much of the engineering process as possible. The ethical concern is choosing those metrics so engineers are correctly and fairly measured.

9 Conclusion

The gathering of software metrics is a difficult task in order to get accurate and meaningful information, a Holy Grail of software metrics would perfectly take into account all aspects of the software engineering process, the insights that are gained by achieving this would be very profitable to a company in order to be as efficient as possible. Companies will continue to invest in ways to achieve this.

While measuring the efficiency and quality of an individual developer is useful, I believe that it is the development methodologies that will provide the solution. Agile is widely used as a way to structure the software engineering process, while also providing metrics in the velocity of teams and what they can accomplish. I believe that agile development will continue to be adopted as the most beneficial approach to measuring the process of a software engineer.

References

https://tdwi.org/Blogs/TDWI-Blog/2010/04/Effective-Metrics.aspx##targetText=A%20good%20performance%20metric%20embodies,story%20of%20the%20organization's%20strategy.

https://stackify.com/track-software-metrics/

https://codescene.io/docs/guides/technical/code-churn.html

https://dzone.com/articles/code-churn-a-magical-metric-for-software-quality

https://en.wikipedia.org/wiki/Code_coverage

https://medium.com/the-andela-way/what-technical-debt-is-and-how-its-measured-ff41603005e3##targetText=Technical%20debt%20accumulates%20interests%20over,Technical%20Debt%20Ratio%20%5BTDR%5D.

https://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.2/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm

https://www.verifysoft.com/en_halstead_metrics.html

https://en.wikipedia.org/wiki/Cyclomatic_complexity

https://www.atlassian.com/agile/scrum

https://www.atlassian.com/software/jira/agile

https://anthonysciamanna.com/2016/06/26/the-problem-with-software-metrics.html

https://www.tutorialspoint.com/software_engineering/software_design_complexity.htm