**MEASURING SOFTWARE ENGINEERING**

Muhammad Rvail Naveed

Student No: 1732983

**Introduction**

Software Engineering can be defined as the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software. Modern software is becoming increasingly more and more complex at an exponential rate. Since it’s inception in the late 50’s it has involved into a profession concerned with how best to maximize the quality of software and of how to create it. With technology advancing so fast, the need to understand how to make the engineering process faster and more efficient has advanced with it. Over the years there have been many discussions on how best to measure the effectiveness of the software engineering process and the productivity/competence of the developers responsible for constructing the products.

There are many metrics and methodologies that can be used to determine this. This report will discuss some of these widespread methods, their usefulness, benefits and possible ethical concerns that may arise from the collection of such data.

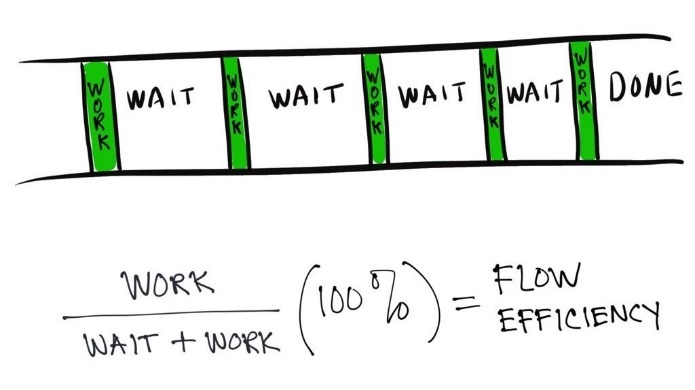
**Measuring and Gathering Metrics**

One of the most basic metrics that can be measured is Source lines of code (SLOC). It is used to measure the size of a program by counting the number of lines in the source. It is used to predict the effort it will take to produce a piece of software and can also be used to estimate programmer productivity once the software is complete. The basic approach is “the better the programmer, the more lines of code they will have written”. There are two main ways of measuring SLOC commonly referred to as physical SLOC and logical lines of code (LLOC). In physical SLOC the method is to count every line of code that the programmer has written, excluding comments whereas LLOC tries to determine how many statements are in the source code. Both ways can be achieved relatively easily by simple algorithms such as “wc” from GNU’s Coreutils for SLOC.

Some advantages of using SLOC as a metric for measuring the software engineering process is that it is easy to automate the process of counting the lines and many utilities to do this already exist. Another advantage is that it is an intuitive metric, it can be seen and the effect of it can be visualized. However, the bad greatly outweighs the good. As someone once said “using SLOC to measure the software progress is like using kg for measuring progress on aircraft manufacturing”. The SLOC metric encourages bad practises such as copy-paste syndrome and discouragement of refactoring to make things easier. Just because a developer is producing 1,000 lines of code a week does not mean that that code is optimised or even completely bug free. The lines of code would also depend on the experience of the developer, number of lines differs from person to person and an experienced developer may implement certain functionality in fewer lines of code than another developer of relatively less experience, even though they use the same language.

Hours spent programming is another very basic metric that by itself is not very useful. Although it can give a quick insight as to how long a specific problem is taking or how long it took to deploy a new feature. It is better integrated alongside Test Driven Development(TDD). Research has shown that doing TDD greatly reduces the amount of other non-coding work that developers must do, and this in turn motivates developers to get on board. A 2005 study, [*"On the Effectiveness of Test-first Approach to Programming"*](http://nparc.cisti-icist.nrc-cnrc.gc.ca/npsi/ctrl?action=shwart&index=an&req=5763742&lang=en), found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.

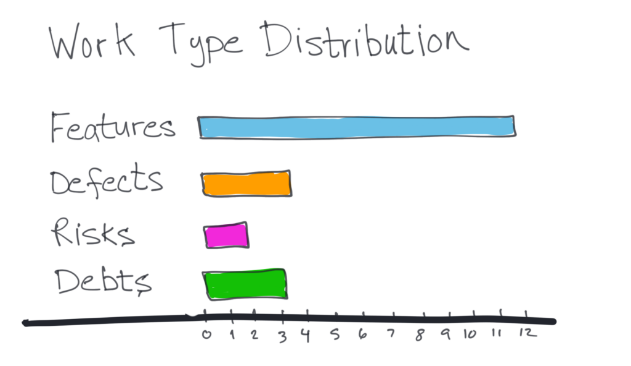
Flow metrics are a collection of metrics to help identify how long a piece of software is taking to complete and efficiency of development. “Flow” can be thought of as value that can be pulled through a system smoothly and predictably. Flow time is the amount of time a flow unit spends in the development pipeline from conceptualisation to deployment. This can be useful to help the team understand which items are taking longer to complete so more resources can be dedicated to that task. Flow efficiency examines the two basic components that make up flow time: working time and waiting time. Waiting time can be encountered for many reasons such as: dependencies, priority changes, too much work in progress etc. Simply put, work in progress is not always *actually* in progress. Flow efficiency shows how often that is true. Flow efficiency can be calculated like so:



Teams that are not aware of this metric generally have a flow efficiency of 15%, compared to the acceptable 40+% described by David J. Anderson in ”Low Flow Efficiency: Resist temptation to design out waste”. Once you dive into the details of the work, you can begin to discover the causes of the unnecessary wait times and the impact each one of those causes has. At that point, you can start to design experiments to try to reduce the wait and increase your flow efficiency.

Analysing a Work in Progress (WIP) report can help teams to see the relationship between WIP and speed of delivery. Too much WIP opens the door for more dependencies, more conflicting priorities and more unplanned work to creep in, which causes delay. Capturing WIP trends and comparing them to Flow Time can help teams see the relationship between WIP and speed in the project.

Flow Distribution categorizes work into different work types which supports changing work priorities and report data filtering. It helps to show targeted (and historical) proportion of work item types, brining visibility to planned work allocation.



Commits are also an excellent way of gauging developer productivity as well as the impact of their code. For example, a commit with 100 lines of additions, deletions or changes has little impact to the overall software and probably did not take much time to implement. However, a commit showing multiple changes spanning across multiple files indicates complexity as some planning and thought would likely have to have been done by the developer.

**Computational Platforms Available**

Almost every software-oriented organisation uses GitHub to host their code base whether that’s private repositories for internal use or public open source projects. The platform offers some useful tools to give insights about the development lifecycle of a project and some useful analytics of the team behind it. Commits can help draw conclusions about a developer’s behaviour, work style, competence, productivity and more. They show how many lines of code were added/deleted and what changes were made. Valuable team members can also be identified by analysing who is making the most effective pull requests, which team members are doing code reviews to accept those pull requests and which team members are solving open issues in the project.

Gitcolony is an excellent tool that can integrate with GitHub and Bitbucket with support for Gitlab coming soon and provides powerful features to further improve developer productivity. Gitcolony claims to help teams save 360 hours a month in a 10-developer team working 180 hours a month, which would be equivalent to hiring two more developers. It offers teams the ability to perform “progressive checks to avoid titanic reviews before deploying” via Partial Reviews. Partial reviews allow for code to be checked as it’s being written, making the review process more actionable and meaningful. Unlike traditional code reviews, Gitcolony also saves your location if leave a review for later, saving a monumental amount of time that would have been spent reviewing it twice. One can also create virtual pull requests which are not populated on GitHub so people can vote to have it merged.

Along with the virtual pull request, merges are also made simpler in Gitcolony. Before merging, the pull request can be analysed, discussed, mergers can be individually assigned, and merges can be voted on. Coupled with Gitcolony’s intuitive merge interface this makes the merge a much more meaningful and less error prone.

Gitcolony also allows teams to enforce the informal code review policies they follow internally. The Early Warning System(EWS) works in tandem with these rules and ensures code quality and can detect issues before they happen. An incident is created for every broken rule which allows for quick fixes and avoids such things as a bad merge before it is deployed. Gitcolony can even integrate with other tools that the team is probably also using such as Jenkins, Slack and Jira.

Jira which was developed by Atlassian is the #1 tool used by agile teams. It allows teams to plan out their “sprints” which are periods of time, usually about 2 weeks, specified by Agile teams to complete a task. Jira offers the ability to create Scrum boards, giving the ability to track progress of sprints and interact with tasks and team members. This allows Agile teams to stay focused on delivering iterative and incremental value, as fast as possible. Time tracking capabilities and real-time performance reports such as burn down charts are provided out of the box enabling teams to closely monitor their productivity over time. Issue tracking also comes standard, allowing teams to bring information from their preferred version control or feature flagging tool into Jira and obtain instant visibility into their development pipeline.

Testrail is an automated test case management tool that can integrate with many testing frameworks and IDE’s. It provides a web-based interface to efficiently manage, track and organize software testing efforts. Unlike Jira, TestRail focuses on letting QA members and software developers the ability to create and manage test cases, monitor test results and code coverage. The platform provides flexible project organization that lets teams manage all their test projects and have access to relevant project details instantly. TestRail also keeps a record of all test case history to track changes and ensure transparency and baselines for multiple branches and versions. QA time can be greatly reduced by having the majority of it automated, which allows for quickly checking if code is covering all cases, a desired code coverage has been reached and any bugs that needs to be fixed can be in a fast and orderly fashion.

**Algorithmic Approaches**

Productivity can be formulated as output/input. However, in software development it is commonly calculated as:

where:

* ESLOC *= Effective or equivalent source lines of code*
* PM = *Person Month*

ESLOC must be greater than or equal to the number of source lines created or changed. The productivity metric is intuitive and straightforward when all the code is new. Once modifications begin to occur in the code other factors become prevalent:

* Modification must be preceded by a thorough understanding of the code to be modified.
* If the existing documentation for the software is insufficient then some reverse engineering of the code will have to be done.
* Modifications can’t break any existing interfaces, putting a burden on the QA team as well as developers.
* If the new modification is in a different language or linked with new libraries etc the complexity increases.

Taking these factors into consideration and applying an adaptation adjustment factor (AAF):

where:

* = Percentage of the reused software requiring redesign and reverse engineering
* = Percentage of the reused software that must be physically
* = Percentage of the reused software requiring regression testing

ESLOC then becomes:

where:

* = New lines of code
* = Modified lines of code
* = Reused lines of code

This metric proves to be very useful in the long run when used continuously in a team. Over time ESLOC can provide useful statistics about the productivity of a software development team, allowing realisation and the ability to be make improvements to work style to result in a better ESLOC value.

* Halstead metrics
* AI
* Entropy
* SEAR-SEM equation

**Physical Approaches**

* Team ethics, work style
* Agile
* Scrum

**Ethical Concerns of gathering such data**

**Conclusion**

# Bibliography

* https://www.scss.tcd.ie/Stephen.Barrett/teaching/CS3012/reading\_workmeasurement.html
* https://nortal.com/blog/the-myth-of-developer-productivity/
* https://redfin.engineering/measure-job-satisfaction-instead-of-software-engineering-productivity-418779ce3451
* https://stackify.com/measuring-software-development-productivity/
* https://go.tasktop.com/rs/246-WDG-185/images/Understanding%20Software%20Development%20Productivity%20from%20the%20Ground%20Up.pdf
* https://www.tasktop.com/blog/5-best-metrics-youve-never-met/
* https://www.tasktop.com/blog/flow-time-vs-lead-time/
* http://se.inf.ethz.ch/old/teaching/2008-S/se-0204/exercises/07-Algorithmic-Estimation-Techniques-ho.pdf
* http://www.cs.umd.edu/~mvz/cmsc435-s09/pdf/slides16.pdf
* https://semmle.com/assets/papers/measuring-software-development.pdf
* http://sunnyday.mit.edu/16.355/metrics.pdf
* https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/
* https://qa-platforms.com/software-metrics/
* https://en.wikipedia.org/wiki/Halstead\_complexity\_measures
* https://en.wikipedia.org/wiki/Test-driven\_development#cite\_ref-12
* https://dzone.com/articles/measure-time-spent-coding
* https://www.tasktop.com/flow-metrics
* https://www.gitcolony.com/features
* https://www.atlassian.com/software/jira
* https://www.thoughtworks.com/insights/blog/augmenting-software-development-artificial-intelligence
* https://www.theserverside.com/feature/AI-development-tools-make-software-development-easier
* https://www.gurock.com/testrail/tour/modern-test-management