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

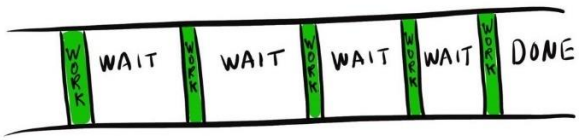
One of the most basic metrics that can be measured is Source lines of code (SLOC). It is used to measure the size of a program by counting the number of lines in the source. It is used to predict the effort it will take to produce a piece of software and can also be used to estimate programmer productivity 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"*, 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:

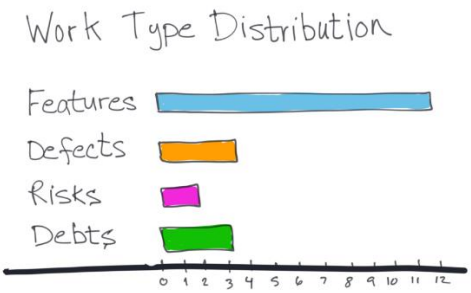


$$\frac{\text{WORK}}{\text{WAIT} + \text{WORK}} (100\%) = \text{FLOW EFFICIENCY}$$

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:

$$Productivity = ESLOC \div PM$$

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):

$$AAF = 0.4F_{des} + 0.3F_{imp} + 0.3F_{test}$$

where:

- F_{des} = Percentage of the reused software requiring redesign and reverse engineering
- F_{imp} = Percentage of the reused software that must be physically
- F_{test} = Percentage of the reused software requiring regression testing

ESLOC then becomes:

$$ESLOC = S_{new} + S_{mod} + S_{reused} * AAF$$

where:

- S_{new} = New lines of code
- S_{mod} = Modified lines of code
- S_{reused} = Reused lines of code

This metric proves to be very useful in the long run when used continuously over a long period of time. 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.

Software Entropy, like the chief characteristic of entropy in the real world, software entropy is a *measure of chaos that either stays the same or increases over time*. In software terms that roughly translates to it being a measure of the inherent instability built into a software system with respect to altering it. Unfortunately, software entropy is rarely accorded the importance it deserves. It is never taken into consideration when someone from a development team leaves or if a development cycle is started prematurely, both situations where it is most likely to grow. The proposed definition of software entropy is as follows:

$$E = I'C \div S$$

where:

- I' = No. of unexpected problems that arose during the last development iteration.
- C = Perceived probability that changes to the system will result in a new $I' > 0$.
- S = Scope of the next development iteration.

An E value of < 0.1 is seen as good, 0.5 being high and $E > 0.5$ is regarded as being overwhelming. The goals that during the software iteration are called its *scope*. In software development, such cycles involve modifying or expanding the software's underlying code. In practical terms, then, we can define software entropy as follows:

Any given system has a finite set of known, open issues I_0. At the end of the next development iteration, there will be a finite set of known, open issues I_1. *The system's inherent entropy specifies how much our expectation of I_1 is likely to differ from its actual value and how much the difference is likely to grow in subsequent iterations.* (Adam Wasserman, Toptal).

Software entropy **cannot** be reduced, it either increases or stays the same. *Risk tolerance* is a measure of how much ad what type of unexpected behaviour can comfortably be permitted from one development iteration to the next. Risk tolerance and software entropy are related in

that software entropy must be minimal to be certain that it will stay within a system's risk tolerance during the next development iteration.

Entropy can arise from factors such as a lack of knowledge, the logic being well known but it diverges from its original characteristics and the logic being not well known. Though the most important factor for an increase in entropy is that there are multiple, incomplete interpretations of the logic the system is intended to express. A lack of communication between team members can cause for each person to interpret the goal of the development iteration differently thus leasing to insufficient, lacklustre and skewed results.

Calculating software entropy proves intuitive as I' , the number of work units abandoned or left partially complete the last development iteration + the new issues that have arisen this current iteration are tracked by software such as JIRA, previously discussed *C* and *S* are more conceptual and organization specific. Although software entropy cannot be reduced, efforts can be made to restrain its growth. Adapting the pair-programming practises of Agile development and adopting methodologies such as Waterfall will likely see a settling of entropy. Entropy should be considered by organizations big and small, although negligible at the start of a project, if left unchecked, it keeps growing every iteration eventually bringing development to halt prematurely.

The Halstead complexity measures are more software metrics first proposed but Maurice Howard Halstead in 1977. For a given problem, let:

- n_1 = No. of distinct operators
- n_2 = No. of distinct operands
- N_1 = Total number of operators
- N_2 = Total number of operands

Using these variables, several other metrics can be deduced such as:

- **Program Vocabulary**: $\mathrm{n} = \mathrm{n}_1 + \mathrm{n}_2$
- **Program Length**: $N = N_1 + N_2$
- **Calculated Estimated Program Length**: $\widehat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- **Volume**: $V = N * \log_2 n$
- **Difficulty:** $D = \frac{n_1}{2} * \frac{N_2}{n_2}$
- **Effort:** $E = D * V$

Halstead's volume describes the size of the implementation of an algorithm. The computation of V is based on the number of operations performed and operands handled in the algorithm. Therefore, V is less sensitive to code layout than the SLOC metrics. The volume of a function should be at least 20 and at most 1000. A volume greater than 1000 tells that the function probably does too many things.

The difficulty metric is related to the difficulty of the program to write or understand. This would be most prominent when doing a code review. The effort measure can be transformed into actual coding time using:

$$T = \frac{E}{18}$$

Halstead also had a metric for the amount of errors in a software implementation, known as the delivered bugs metric:

$$B = \frac{E^{\frac{2}{3}}}{3000}$$

Delivered bugs in a file should be less than 2. Research has shown that, when programming with C or C++, a source file almost always contains more errors than B suggests. The number of defects tends to grow more rapidly than B. When dynamic testing is concerned, the most important Halstead metric is the number of delivered bugs. The number of delivered bugs approximates the number of errors in a module. As a goal, at least that many errors should be found from the module in its testing.

During the time of their proposal in the late 20th Century, the Halstead metrics were instrumental in calculating software complexity. They are more often used nowadays as a maintenance metric, however there is some evident that the measures may also prove fruitful during development, to assess code quality in computationally dense programs. Halstead's metrics have been used and experimented with extensively since their inception. They are one of the oldest measures of program complexity, however nowadays they are coupled with other algorithmic approaches such as Cyclomatic Complexity calculations for a stronger understanding of a software's complexity.

SEER for Software (SEER-SEM) is an algorithmic project management software designed to estimate, plan and monitor the effort and resources required for any type of software development. SEER-SEM uses a combination of algorithms to accomplish this goal.

Software size is calculated by SEER-SEM as such:

$$S_e = NewSize + ExistingSize * (0.4 * Redesign + 0.25 * Reimpl + 0.35 * Retest)$$

S_e increases as the amount of new code being developed increases. The extent of the increase depends on the amount of redesign that must be done in order to reuse the code.

Ina project, Effort and Duration are related and as such so are their calculations. From this effort can be calculated like so:

$$K = D^{0.4} * (\frac{S_e}{C_{te}})^E$$

where:

- S_e = Effective Size
- C_{te} = Effective technology, captures factors relating to the efficiency or productivity with which development can be carried out. A higher rating means that development will be more productive.
- D = Staffing Complexity
- E = Entropy. Fixed nowadays at 1.0 – 1.2 depending on project attributes.

Once the effort metric is calculated, duration can then be found:

$$t_d = D^{-0.2} * (\frac{S_e}{C_{te}})^{0.4}$$

The 0.4 exponent indicates that as a project's size increases, duration also increases, though less than proportionally. This size-duration relationship is also used in component-level scheduling algorithms with task overlaps computed to fall within total estimated project duration.

SEER-SEM algorithms are very useful for teams to gain an idea of the effort required to produce an end product, reduce uncertainty with accurate forecasts and create detailed project plans.

Artificial Intelligence is a rapidly growing industry with applications of the concept involved in almost all aspects of software. Based on predictions the industry's revenue will reach $1.2 trillion by the end of 2019, up 70% from last year. Over the years there has been a phenomenal increase in the amount of compute power and data available to experiment on. Given the abundance of resources, AI has also recently been assisting in the software development process, aiming to improve deploy times and automate mundane tasks that so often cause a stark decrease in developer productivity.

There are levels of sophistication that AI assistance can be classified into. The first regards automation of manual tasks such as testing that can increase reliability and efficiency of software creation. The second level concerns AI that can make decisions based on fixed rules defined by a human. A good example of this can be seen in auto-scaling infrastructure. The AI determines the compute power necessary to service loads being handled by an application and humans configure the bounds that the compute power can scale.

AI can also assist in the testing process during software development. For example, writing tests for legacy systems can be hard and tedious. Developers can leverage the power of AI to map out an application's functionality, using usage and code analytics, allowing teams to quickly build a safety net around such legacy systems. This allows for developers to make changes without breaking existing functionality. According to Manish Mathuria, the CTO of Infostretch, intelligent testing has seen gains in agility by 25-30% and quality by 35 to 40%.

Code completion has seen significant strides in the past decade with tools such as Microsoft's IntelliCode and Kite boosting developer productivity by using AI to predict keystrokes. Kite, in particular, claims to cut down developer keystrokes by nearly 50% saving monumental amounts of time. Facebook has also recently been experimenting with SapFix, an in-house software created to assist in the automatic detection and repair of bugs in software. SapFix is an end-to-end code repair tool that can automate test case design, bug detection and code repair. The company has used it to fix six production systems to date, which consist of tens of millions of lines of code. The system automatically generates fixes which are then approved by a human.

Ethical Concerns of Gathering Developer Data

As demonstrated in this report, the collection of metrics about the software engineering process is fundamental and necessary for innovation and increasing productivity within a business. These metrics and tools provide useful insights into how a team or organization is running and can be instrumental in recognising downfalls which may be halting progress.

It can be easy to fall into the trap of thinking that metrics are the only definitive answer to boosting developer productivity, however, it is important to understand that metrics are not everything. Multiple studies and reports have proven that employee happiness is directly related to productivity. In 2012, the University of Warwick found that happy workers saw an increase of 12% in their productivity while unhappy workers actually declined in productivity by 10%. Shawn Anchor, the author of "*Happiness Advantage"* also proposed that the brain works better when it is happy. This is why companies like Google and Facebook have started investing resources to ensure employee happiness, with Google reporting a happiness total of 37%. They have realised that financial incentives are no longer the only thing that can boost employee productivity.

The use of wearable technology has also become prominent in workplaces such as JP Morgan Chase. The devices can capture data about their wearer's location, environment, activity and biometrics, as well as transmitting visuals and sounds. This can very easily lead to abuse of the huge amount of data being collected and could even be viewed as a privacy concern. Treating employees this way may lead them to feel highly alienated and constantly monitored, decreasing their drive to succeed and dedication as well as respect for the organisation they are working for. Some of the data may be information that the employer does not want or need or even look at. It may also be data that employees or their unions did not consent to have monitored. It just comes with the wearables.

This is partly why tech giants such as Google have changed their approach to focus more on making employees comfortable instead of gathering more data about them. Undoubtedly, keeping your employees satisfied will increase their drive to succeed and dedication to their

work, allowing the company to reap the benefits without infringing on possible privacy and ethical concerns.

Conclusion

In conclusion, the tools and metrics used to measure the software engineering process are abundant and sometimes quite complicated. However, with careful use and analysis, the processes can prove fruitful to developers, cutting down on their workloads dramatically and assisting with mundane tasks. A combination of metrics, the use of tools such as JIRA and adoption of methodologies such as Agile and Waterfall can lead to a more productive and faster team.

References

- Measuring Software Devlopment Productivity:
  - **https://stackify.com/measuring-software-development-productivity/**
- Interesting Software Development Metrics:
  - **https://www.tasktop.com/blog/5-best-metrics-youve-never-met/**
- Halstead's Software Metrics:
  - **https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/**
- Software Metrics, Benefits and Use:
  - **https://qa-platforms.com/software-metrics/**
- Halstead's Complexity Measures:
  - **https://en.wikipedia.org/wiki/Halstead_complexity_measures**
- Test Driven Development (TDD):
  - **https://en.wikipedia.org/wiki/Test-driven_development#cite_ref-12**
- Flow Metrics:
  - **https://www.tasktop.com/flow-metrics**
- Gitcolony Features and Uses:
  - **https://www.gitcolony.com/features**
- JIRA Features:
  - **https://www.atlassian.com/software/jira**
- Augmenting Software Development with AI:
  - **https://www.thoughtworks.com/insights/blog/augmenting-software-development-artificial-intelligence**
- AI Tools to Assist in Software Development
  - **https://www.theserverside.com/feature/AI-development-tools-make-software-development-easier**
- TestRail:
  - **https://www.gurock.com/testrail/tour/modern-test-management**
- Software Entropy:
  - **https://www.toptal.com/software/software-entropy-explained**
- Measuring Productivity of Software Teams:
  - **http://scet.berkeley.edu/wp-content/uploads/Report-Measuring-SW-team-productivity.pdf**
- SEER-SEM:
  - **https://en.wikipedia.org/wiki/SEER-SEM**
- Productivity and Happiness:
  - **https://www.knowmail.me/blog/productivity-affect-happiness/**