CS3012-Measuring Software Engineering Analytics

Name: Alex Kennedy

Student No. : 17328638

In the software engineering industry today, there is an extremely rapid and constant cycle of developing, testing and production/deployment. Because of this cycle, organisations must find methods to assess and measure their developer's performance. These organisations need to be able to draw useful metrics from this style of work in order to make sure that productivity levels remain high and work is spread evenly across developers.

Traditional metrics of performance such as hours worked cannot be used when dealing with software engineering due to the problem-solving nature of the industry as the amount of time you put into working may not always yield a big output. So, the main questions of this report are: what metrics can be used to measure performance, what are the main platforms in which these metrics can be viewed, are there algorithmic approaches to gathering this data, and are there any ethical concerns surrounding this kind of analytics.

Measurable Data

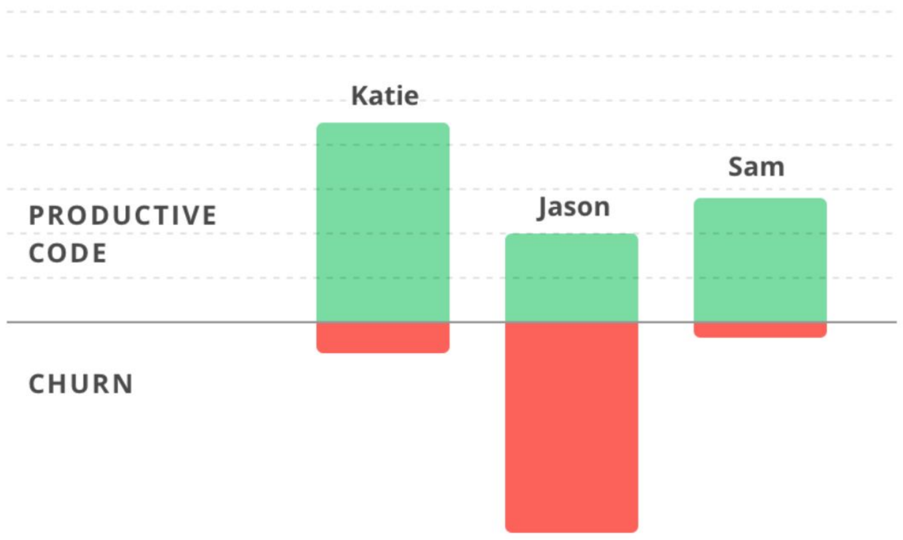
There are an almost endless amount of different metrics to gather from a software engineer. Starting at the most basic is the amount of lines of code they write in a certain time frame. Many factors are to be taken into consideration when talking about lines of code written as a performance metric. A developer may have written many lines of code in a day, but this code could be hard to read, unoptimized and filled with bugs.

Lines of Code

Another point to note is that if you were to look at lines of code as metrics, one would have to look at the impact of a developer's code made on a piece of software. A developer could have written 200+ lines of code could have been simple to implement, whereas another developer could change 2 lines of code to fix a huge bug that no one else could fix. While the first developer wrote 100 times more code, their work was of much less impact than the second developer. This simple example clearly shows the discrepancies with using lines of code as a performance metric of software development and shows that other metrics should be explored.

Code Churn

Code Churn is a metric which can show the rate at which your code changes and grows over time. This is a better metric to look at rather than looking at the number of lines of code. It can show how much duplicate code has written over a period of time. Below is a good figure demonstrating what code churn is.



Although Jason has technically written the most lines of code, the majority of it is duplicate or useless code and so Katie seems to have performed better than Jason in this instance. Although once again it is difficult to measure whether Katie had more of an impact on the project as her task could have been very trivial, while Jason could have undertaken a much more difficult and time consuming task which required much more iteration. For that reason, code churn, while a useful metric cannot be considered to solely measure an engineer's performance.

Code Coverage

Another popular metric that is mentioned when it comes to a software team's performance is Code Coverage using Unit testing. This involves developing tests for your code that you have written to ensure all lines of source code has been executed and that your code functions correctly for all conditions. While I believe that Unit testing is an extremely important aspect of software development, in fact in most cases you are better off writing your test code before even writing actual source code, code coverage as a metric is very limited in showing a developer's performance. Developers can write very simple, unnecessary tests just to increase an arbitrary code coverage percentage. These badly designed tests do not contribute to the development of the software at all and in the end, are just a waste of time and resources.

Technical Debt

Technical debt is another metric that could potentially measure the performance of a software team. Technical debt is described as additional work needed to be done in the future if opting for a simpler solution now. In other words, if you want a speedy delivery of your software, then you will most likely run into problems in the future because of this rushed, unoptimised code. Technical debt should be tackled as it comes as the longer it is left, the more troubling the problem becomes. There are many forms of technical debt whether it be low test coverage, code complexity or a lack of documentation. Technical debt should not be measured manually as it would be extremely time consuming sifting through code. Also since the code is evolving, the technical debt would evolve also. The measure of technical debt is simply:

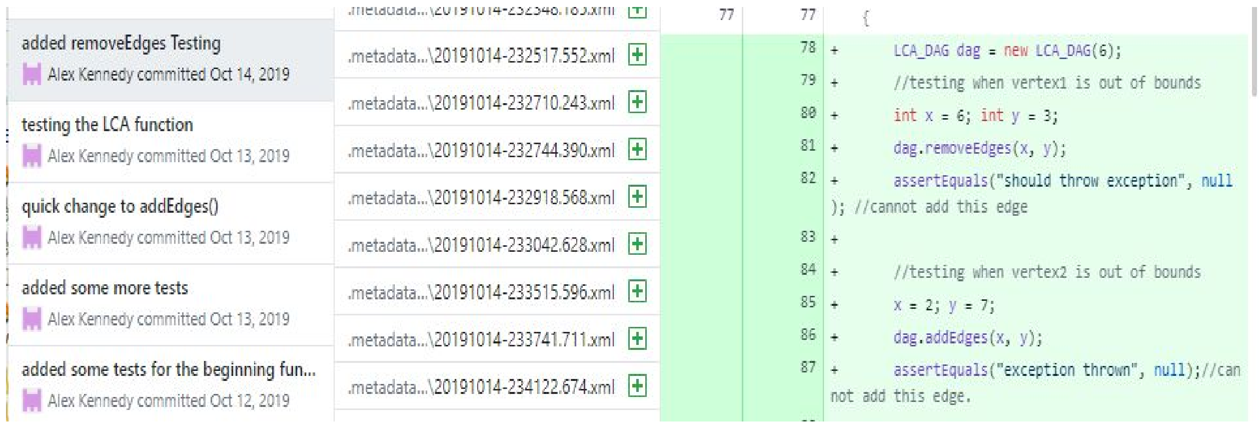
Technical Debt = Remediation Cost / Development Cost.

Remediation Cost is the cost to fix a problem already in the program. Technical debt could be used as potentially a decent metric for software team performance. Software with a low technical debt could suggest that a team is keeping on top of their work while also not taking any shortcuts when it comes to programming, thus showing they are performing well. Although technical debt is inevitable and because the amount of it changes with every new feature that is added to a piece of software, at some points the technical debt might be high and unavoidable so it is not a completely accurate measure of performance.

Computational Platforms

Github Commits

Many useful applications have been created to assist software teams in developing software. Many of these applications include a lot of useful analytics which could potentially be used as a basis for measuring the performance of a software engineering team. Github is the most popular repository hosting site in the world. Lots of interesting metrics can be drawn from commits made to Github. Commits provide information on the lines of code added or removed from a piece of software, along with a message on what the purpose of the commit was. Although the statistics of commits does not clearly show how impactful changes were, an interesting metric that can be drawn is the importance of certain developers in a project. You can see clearly how much they have contributed to the codebase. Because of this, it could be assumed that a developer that has worked on the project since the beginning would have a better time fixing bugs or solving problems rather than a new developer who just started due to just having more knowledge of the codebase. This does not necessarily measure performance however, like measuring lines of code, it measures how much a developer has contributed but not their performance on the project.



added removeEdges Testing
Alex Kennedy committed Oct 14, 2019

testing the LCA function
Alex Kennedy committed Oct 13, 2019

quick change to addEdges()
Alex Kennedy committed Oct 13, 2019

added some more tests
Alex Kennedy committed Oct 13, 2019

added some tests for the beginning fun...
Alex Kennedy committed Oct 12, 2019

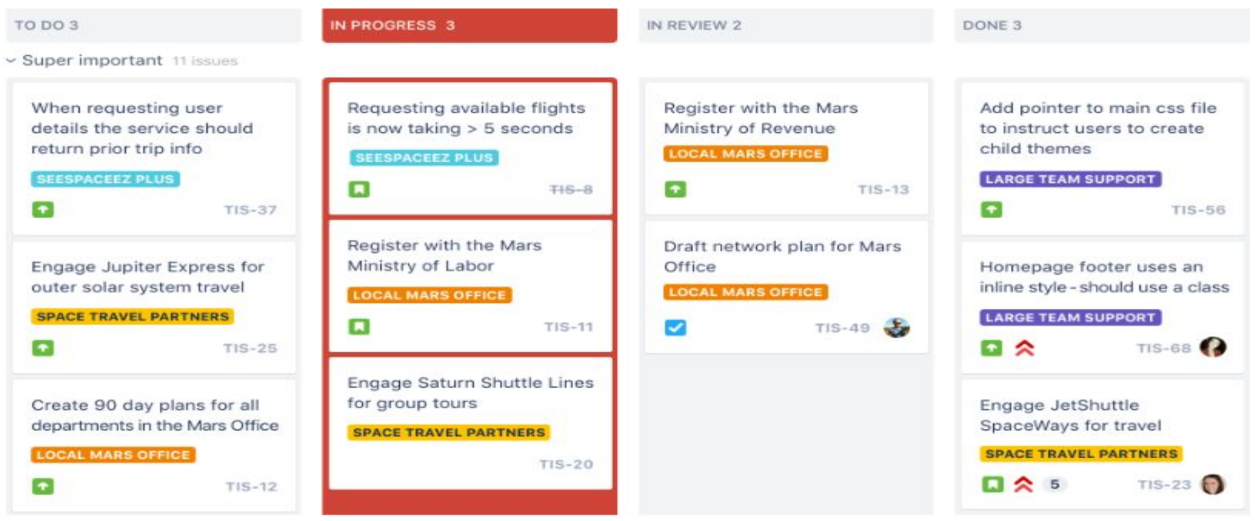
```
.metadata...\20191014-232517.552.xml
.metadata...\20191014-232710.243.xml
.metadata...\20191014-232744.390.xml
.metadata...\20191014-232918.568.xml
.metadata...\20191014-233042.628.xml
.metadata...\20191014-233515.596.xml
.metadata...\20191014-233741.711.xml
.metadata...\20191014-234122.674.xml
```

```
      {
+       LCA_DAG dag = new LCA_DAG(6);
+       //testing when vertex1 is out of bounds
+       int x = 6; int y = 3;
+       dag.removeEdges(x, y);
+       assertEquals("should throw exception", null
); //cannot add this edge
+
+       //testing when vertex2 is out of bounds
+       x = 2; y = 7;
+       dag.addEdges(x, y);
+       assertEquals("exception thrown", null);//can
not add this edge.
```

An example of how Github shows commits on a branch in Github Desktop, shows message and what code is added or removed.

Jira

Another tool is Jira, developed by Atlassian. It is a powerful work management tool originally built as a bug and issue tracker. It is very popular among software development teams who run under an agile software development methodology. Managers can create "Boards" and map tasks to software engineers. These tasks can be tracked in real time as developers work on them and create performance charts such as burn up charts, sprint reports and velocity charts. These statistics can show what is left to do and the likelihood of completing the tasks in the timeframe. Jira is a very good way of monitoring a team's productivity over time. Jira also has seamless integration with Github, allowing you to see branches, commits and much more within Jira, increasing productivity and efficiency when bringing code from development to deployment.



TO DO 3

Super important 11 issues

- When requesting user details the service should return prior trip info — SEESPACEEZ PLUS — TIS-37
- Engage Jupiter Express for outer solar system travel — SPACE TRAVEL PARTNERS — TIS-25
- Create 90 day plans for all departments in the Mars Office — LOCAL MARS OFFICE — TIS-12

IN PROGRESS 3

- Requesting available flights is now taking > 5 seconds — SEESPACEEZ PLUS — ~~TIS-8~~
- Register with the Mars Ministry of Labor — LOCAL MARS OFFICE — TIS-11
- Engage Saturn Shuttle Lines for group tours — SPACE TRAVEL PARTNERS — TIS-20

IN REVIEW 2

- Register with the Mars Ministry of Revenue — LOCAL MARS OFFICE — TIS-13
- Draft network plan for Mars Office — LOCAL MARS OFFICE — TIS-49

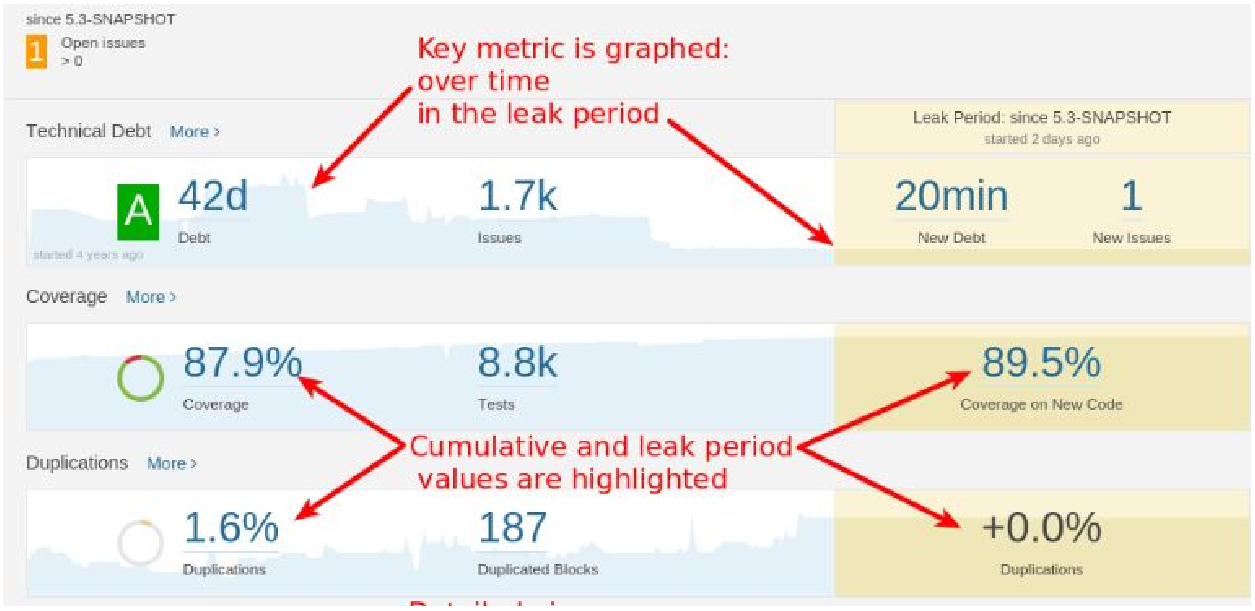
DONE 3

- Add pointer to main css file to instruct users to create child themes — LARGE TEAM SUPPORT — TIS-56
- Homepage footer uses an inline style - should use a class — LARGE TEAM SUPPORT — TIS-68
- Engage JetShuttle SpaceWays for travel — SPACE TRAVEL PARTNERS — 5 — TIS-23

Example of a Kanban Board in Jira, easy to see tasks finished and in progress.

SonarQube (Technical Debt)

Developers can make use of automated software tools in order to reduce some of the workload and potentially increase productivity. SonarQube is a tool used to help measure technical debt within a piece of software, potentially improving code quality. While it has many features, its main draw is its use in reducing technical debt. Its “Water Leak Paradigm” is the way in which it manages code quality. It can detect “leaks” which are essentially any potential bugs or vulnerabilities within the software. Once these leaks are fixed, SonarQube can then tell us if the code quality got better or worse during the leak period. This tool can give some good metrics on technical debt within the software and which code is causing this technical debt. How these metrics are useful was already mentioned above in the previous section about technical debt.



Algorithmic Approach

Halstead Complexity Measures

One of the more well known algorithms for measuring software engineering metrics is the Halstead complexity measures developed by Maurice Howard Halstead in 1977. Typically, these measures are used as maintenance metrics because they are directly applied to code. This method doesn’t draw metrics on an engineer's performance. Instead, it uses the software's codebase and tries to produce some metrics on the quality of the code, and how long a task may take. Because maintainability should be a main concern during code development, Halstead's metrics could be used to follow complexity trends.

Using just 4 pieces of data, a lot of metrics can be calculated:

n1 = number of distinct operators, n2 = distinct number of operands,

N1 = total number of operators, N2 = total number of operands.

The different metrics that can be calculated are found in the figure below:

Program vocabulary: $\eta = \eta_1 + \eta_2$

Program length: $N = N_1 + N_2$

Calculated estimated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

Volume: $V = N \times \log_2 \eta$

Difficulty : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$

Effort: $E = D \times V$

The Difficulty metric attempts to demonstrate how difficult to understand or write. The effort metric eventually translates to an estimation on how much time it took to write the software using:

T = E/18 seconds.

The number of bugs delivered can also be calculated using the following equation :

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

The number of bugs(B) should be less than 2. Note that time and delivered bugs are estimation only and don't calculate the exact time and bugs in every program created. In general, the actual number of errors in a program is usually greater than 2. Some of the metrics have been introduced into current computational platforms, so even though this method was created in 1977, it is still used quite widely in the software engineering industry.

<u>**Productivity**</u>

Traditionally, a workers productivity can be calculated by the following:

Productivity = Output / Input.

However, due to the copious amount of factors affecting software engineering, this formula is redundant and so another formula has been historically used regarding software engineering:

Productivity = ESLOC / Person Month

Where ESLOC is Effective Lines of Code and ESLOC has to be greater or equal to the number of source lines of code created or changed. Calculating the ESLOC when all the code is brand new is a rudimentary task. But the majority of the time, there are modifications to the code and so, a number of other factors must be considered. A modification to the code requires an understanding of the code that needs to be changed, as well as knowledge of the underlying system and architecture. Modifications must also not break any existing parts of the software. Complexity of software increases if the modification has to be done on a new language, compiled on a new compiler or linked with newer libraries.

Because of these factors, an adaptation adjustment factor(AAF) is applied to the formula if there is modified code:

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

Where:

- F_{des} is the fraction of the reused software requiring redesign and reverse engineering.
- F_{imp} is the fraction of the reused software that has to be physically modified.
- F_{test} is the fraction of the reused software requiring regression testing.

And so there is a new formula for ESLOC

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

Where:

- S_{new} are the new lines of code.
- S_{mod} are the modified lines of code.
- S_{reused} are the re-used of code.

It is very difficult to measure ESLOC within an organisation due to multiple developers, working on different tasks. The use of external tools can help to overcome this problem and if used correctly over multiple projects within an organisation, useful metrics can potentially be drawn between the productivity of software engineers.

Ethics Regarding Software Analytics

While collecting some of these kinds of metrics mentioned above can be extremely important in improving the production quality of your product and an orgainsation's bottom line. Some ethical concerns do arise.

If software engineers are constantly being watched over while working, this could cause for some serious invasion of privacy. While having tools installed that can track problems with your code and generate useful software engineering metrics, organisations do have to be careful not to collect too much personal data on their employees regarding their daily and personal life. Too much collection of unnecessary data such as tracking how long an employee's break or bathroom break is could create tension between managers and developers. This could create quite a hostile working environment for the developers and so employee performance and productivity could be severely hindered.

Automated tools integrated into developer's work is definitely an important aspect of creating software and collecting data. However, the over collection of metrics and over emphasis on this data can make developers feel under pressure to fulfill these goals and deadlines and in turn, will not produce a positive work environment. The product itself might even suffer if there are certain quotas developers much reach such as lines of code written as developers may put all of their focus on reaching this quota rather than producing a good, optimised product. For this reason, it is up to the organisation and project managers to choose suitable metrics that will help the productivity of its employees.

Conclusion

In conclusion, one can see that attempting to collect effective and reliable data on the software engineering process is definitely not a simple task. There are so many factors to take into account. The whole industry is constantly changing and so basing performance of developers on source code alone, whether it be on lines of code written, commits, technical debt, etc really does not seem to be all that useful. Although Halstead's complexity metrics still do have some relevance today in terms of the maintenance of source code, it is not a very good indication of productivity.

Agile development techniques have started to become popular and seem to be a great way to gather metrics and track productivity. Although working in an agile environment using computational platforms such as Jira is very fast paced, the metrics they collect are quite useful and do not scrutinise workers, allowing for more productivity. Agile development techniques seem to be the way forward when it comes to metric gathering to allow organisations to maximise their profits.

Also in recent years, it has been shown that creating a positive and open work environment for your employees much like Google or Facebook has done does increase worker satisfaction and productivity. Which indicates that while analysing these metrics in software engineering can be essential to the success of a piece of software, there are other factors which should be considered in order to increase company profits.

Bibliography

1. Sealights. (n.d.). *The Real Problem with Code Coverage Metrics in 2020 | SeaLights*. [online] Available at: https://www.sealights.io/code-quality/code-coverage-metrics/ [Accessed 2 Nov. 2019].
2. Lafleur, J. (2018). *Do NOT Measure Developers – Measure Projects*. [online] Anaxi. Available at: https://anaxi.com/blog/2018/11/25/do-not-measure-developers-measure-projects/ [Accessed 2 Nov. 2019].
3. Atlassian. (n.d.). *What is Jira used for?*. [online] Available at: https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for#jira-for-agile-teams [Accessed 3 Nov. 2019].
4. Verifysoft.com. (2017). Verifysoft → Halstead Metrics. [online] Available at: https://www.verifysoft.com/en_halstead_metrics.html [Accessed 4 Nov. 2019].
5. Daniel Okwufulueze. (2019). What Technical Debt Is And How It's Measured. [online] Available at: https://medium.com/the-andela-way/what-technical-debt-is-and-how-its-measured-ff41603005e3 [Accessed 4 Nov. 2019].
6. Osetskyi, V. (2018). What Technical Debt Is and How to Calculate It. [online] DZone. Available at: https://dzone.com/articles/what-technical-debt-it-and-how-to-calculate-i t [Accessed 4 Nov. 2019].
7. Krapivin, P. (2018). *How Google's Strategy For Happy Employees Boosts Its Bottom Line*. [online] Forbes.com. Available at: https://www.forbes.com/sites/pavelkrapivin/2018/09/17/how-googles-strategy-for-happy-employees-boosts-its-bottom-line/#4340e6f922fc [Accessed 4 Nov. 2019].
8. Thompson, B. (n.d.). *Why Code Churn Matters | GitPrime Blog*. [online] GitPrime. Available at: https://blog.gitprime.com/why-code-churn-matters/ [Accessed 5 Nov. 2019].
9. SonarQube (2016). Project Space. [image] Available at: https://docs.sonarqube.org/display/SONARQUBE53/Project+Space [Accessed 5 Nov. 2019].
10. Chatterjee, D. (n.d.). *Measuring Software Team Productivity*. [online] Scet.berkeley.edu. Available at: http://scet.berkeley.edu/wp-content/uploads/Report-Measuring-SW-team-productivity.pdf [Accessed 7 Nov. 2019].