Measuring software engineering

Calum Comerford

Student No: 16318401

Cs3012 - Software Engineering

Software development metrics

When we look to measure software engineering there are many avenues we can look down, but none will give us a truly accurate picture of the whole situation. That is why it is important that we use as many metrics as is reasonable. They include the productivity of a software development team, the performance of the software being produced, the security of the software and the quality of the source code.

Measuring Team Productivity

When the leader of a software team needs to make decisions that will affect the future of the development it's important that that decision is based off data and not their personal feelings. Data should allow the leader to identify issues early and cut them down early before they become much larger problems. However, measuring key points of information can be difficult. There are many potential key performance indicators (KPI's) that one might consider when leading a development team, and it can be difficult to distinguish between relevant indicators and meaningless ones in your particular situation as the usefulness can vary hugely from project to project, leadership style and development style and more. The following metrics don't measure the quality of the software; however, they can be invaluable during the development process.

Velocity

Velocity is a metric for work done, which is often used in agile software development. Velocity always involves counting the number of units of work completed in a certain interval. However, units of work can be measured in many different units. The main idea behind velocity is to help teams estimate how much work they can complete in a given time based on how quickly similar work was previously completed. Velocity is relative measure. In other words, the raw numbers mean little; it is the trend that matters.

One problem with velocity is that it conflates work done with planning accuracy. In other words, a team can inflate velocity by estimating tasks more conservatively. A second problem with velocity is that it does not take quality, alignment with user goals or priority into account. Velocity can be increased by neglecting good design, refactoring, coding standards and technical debt. A third problem with velocity is that it is often misused as a measure of efficiency or team performance. Velocity is a metric of work done, not efficiency. In summary, velocity is a problematic metric because it is easy to manipulate and often misused as an indicator of efficiency. It's important that if you use velocity that you know how to use it and don't rely on it too heavily as it can mislead and be harmful to a project if miss used.

Cycle time

Cycle time, a metric borrowed from lean thinking and manufacturing disciplines, is simply the time it takes to bring a task or process from start to finish. Shorter cycle times often mean an optimized development process and a faster time to shipping the product to market, while longer usually means the process is inefficient. The term "cycle" is used because the metric measures how long it takes for the team to cycle a task from start to finish. Often these cycles are split into sub cycles to more accurately measure progress more precisely.

Like velocity once you have a baseline establish you can use variations in that baseline to gather information about the progress of the team and identify problems with the team. Of course, the issue could be anywhere. That's the biggest issue with cycle time, it doesn't give you any indication of what is going wrong just that something is.

Number of open pull requests

A pull request is made when a developer asks their team or manager to review changes they've made to a code repository. When the pull request is sent, other team members can review the changes, provide feedback, and make additional pushes. Once that's been done, the pull request is marked as closed.

Using the number of open pull request in a repository can be useful. Any version control tool your using allows you to get this number and is a good indictor on how the overall project is progressing. If the number of open pull requests keep increasing, then the team's progress might be slow. Tracking this metric can be useful in identifying bottlenecks in the process and may indicate if more time or resources should be dedicated to code review.

Code Churn

Code churn measures the changes made to a codebase over a period and is a way to quantify the extent of these changes. Code churn is important to a software development team as it can be used to predict defect density. A study by Nagappan et al. showed that certain code churn metrics correlate highly with flaws in the code. These metrics have a role in driving the development and test process, as highly churned code is more prone to defects and should be tested more carefully and more thoroughly.

Code churn is not necessarily bad of course. Testing and reworking code is a normal part of the development process, so some level of code churn is expected and necessary. It's when code churn deviates from the average that there may be a problem. For example, a significant increase in code churn levels among the

developers in the team may suggest that they were given unclear requirements, or the requirements they've been given aren't productive.

Lead time

Lead time is defined as the length of time it takes a team to go from an idea to delivered software. Lowering lead time is often a great way to improve the responsiveness to customers and clients of developers. This metric isn't the most useful during development but as a reflationary tool it can be a great way to improve in the future.

Measuring Software Performance

The following metrics are used to measure the quality of the software they will not tell you about specific features of the program or users affected by its failure, however these metrics are still very useful and aren't complicated to use to obtain data.

Mean time between failures (MTBF)

MTBF is the average time elapsed between failures of a piece of software during normal operation. In other words, it's the time it takes for a system failure to be resolved and another failure to be found. Therefore, this metric measure how long the system is running. This metric can be useful, as it allows us to determine how stable the system is and estimate roughly when then next failure will occur.

Mean time between recovery (MTTR)

Mean time to recover is defined as the average time required to troubleshoot and repair a software system and return it to it normal operations. This metric can help measure the maintainability of a system, as well as its reliability. To calculate MTTR we take the total maintenance time divided by the number of repairs over some specified time.

Application Crash Rate (ACR)

This metric is related to the last two and measures the rate a system fails. It can be calculated by deciding the number of times an application crashes by the number of time that application has been used. This metric isn't as helpful at predicting failures or reliability, but it does provide a look at how a system is preforming.

Measuring software security

This section is highly connected to the previous one as the security of the software is a crucial part of the quality of the software, but it is often a component that is overlooked as you can see by the various "mass hacking" incidents in the news every

other week that leak millions of people's data. Security metrics be tracked over time to show if and how software development teams are developing security responses.

EndPoint incidents

This is probably the best metric to use due to its simplicity, but overall information provided. This just measures number of endpoints - endpoints being things like mobile devices, laptops, workstations and so on - that have experienced security issues (virus infection, etc) over a certain defined period, as well as how often an endpoint suffers from such an issue.

Mean Time to Repair (MTTR)

This is the exact same metric as I previously discussed. In security terms MTTR specifically refers to the average time elapsed between a security problem being discovered and the eventual resolution of the problem. As previously stated this metric should be tracked over a specified period and should be compared to historic data when available. If MTTR is decreasing over time, the software development team is becoming more effective at handling the security issues. If its increasing the team is somehow losing effectiveness and this should be addressed as early as possible.

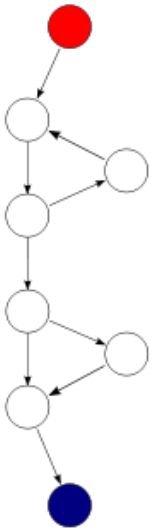
Measuring source code

One additional way that software engineering can be measured is by analysing the source code itself. This can be done by measuring the number of lines the developers write to something as complex as cyclomatic complexity.

Lines of code

Lines of code as stated above is the simple metric of counting the number of lines of code the development team write in a given period. You can simple compare this to historical data from the same team to see if progress is increasing or decreasing. This is obviously a deeply flawed metric that if you but more weight in it than a simple warning system for huge issues when code lines increase or decreases significantly in a short period. If the productivity of software development team members is measured by the number of lines of code they've written, this will lead to several problems, the most obvious being that engineers would then be incentivized to write unnecessarily long, redundant code so that they appear to be more productive than they are. In end writing inefficient code that looks good in this metric could be harmful in the long term. (En.wikipedia.org, 2018)

Cyclomatic complexity



Cyclomatic complexity is a software metric, used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. One testing strategy, called basis path testing by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program.

Computational platforms available

Developers has several tools at their disposal to measure the quality of the code they write. One of the more popular approaches is to use one of the available open source automated code review tools such as Codacy or Scrutinizer. These tools each have their own benefits and drawbacks, it’s up to the developer to

Code review tools

Codacy is an automated code analysis/quality control tool that is integrated into your online version control tool. Among the metrics it provides are cyclomatic complexity, code duplication and code coverage changes from unit testing with each commit.

Companies use tools like Codacy to enforce their respective coding standards. It also comes with a built in docker analysis and extensive security checks. Codacy also has a tool (in Beta as of 2017) that allows users to define their own patterns which will then be checked automatically. There are several tools like Codacy, but it seems to be the gold standard analysis tool among developers.

Hackystat

Hackystat is another open source framework for analysis off the software development process, though it differs from Codacy and other similar systems is how it collects data. Instead of collecting and analysing code upon each commit to a repository it instead requires its users to integrate it with their local development tools. Hackystat will then “unobtrusively” collect and send development data to a web

server, after which it can use it generate graphs for a variety of metrics. The metrics that Hackystat measures are the same as most other services of its kind: code churn, cyclomatic complexity, lines of code, etc. Hackystat's different approach to the collection of data can be useful, as it makes it easier for differentiating data for the entire project, and data for a single developer in the team. (Hackystat, 2018)

Version control tools

Version control tools such as Git and Bitbucket can be very useful in measuring software development, as they track various metrics about every repository, and every commit in a repository. For example, you can use these tools to measure something like commit frequency. You can also combine them with other tools (e.g. Codacy) to measure something more complicated, perhaps related to the quality of the code itself.

Github and other tools like it do measure some code-related metrics. For example, one metric they measure is lines of code either added or removed in a commit, as well as the exact lines edited by a single developer. These tools also can visualize this data in a variety of graphical representations. Of course, if that's insufficient, these tools also provide APIs for you to interrogate, allowing you to collect the specific data you want and represent it however you wish.

Algorithmic Approaches

This section will be tackling the few algorithmic approaches that are available to us when measuring software engineering.

Lines of code

This report has previously discussed the positives and negatives associated with this metric, but how exactly is LOC calculated? There are three main ways it is done. The first is the most obvious: simply count each line of code in whatever source files being analysed. This method doesn't consider blank lines or comments. A raw count that does take those into account, like the line count a text editor provides, can be a useful quick indication of the scale of a project, but it's obviously better to filter those out. Of course, this count still comes with plenty of uncertainty.

The second method is to count the number of logical lines code, which means that you count the number of statements in a piece of code, rather than the number of lines. A line of code may not be a statement, or it might be several. In C/C++, Java, and C#, one quick approach to this might be to count the number of semicolons, though this isn't perfect and wouldn't work very well with loops and other things.

The third method is to count the number of lines the code compiles too. In other words, you count the number of executable statements the code compiles to in its

runtime environment. This method is reliable. You no longer must worry about formatting styles or dealing with different types of loops and comparisons. However, this method isn't perfect. One drawback is that this executable code doesn't include things like abstract methods or interface definitions, which all add complexity to the code. (En.wikipedia.org, 2018)

Cyclomatic Complexity

In the first section of this report cyclomatic complexity was explained as a software metric that determines the complexity of a piece of code by allotting the number of linear independent paths through that code.

Cyclomatic complexity can be computed by using the code being analysed to build a control flow graph, made up of nodes and directed edges. To translate code to a graph, you have the nodes correspond to a command or and indivisible group of commands in the code. You then connect two nodes with a directed edge if the second command (or group of commands) would be executed directly after the first one in that path through the program. The developer of cyclomatic complexity Thomas McCabe, Sr defined it as $V(G) = E – N + 2$,

where E is the number of edges and N is the number of nodes. It can also be defined using $V(G) = P + 1$, where P is the number of nodes that contain a condition, known as predicate nodes.

Ethical concerns

Measuring software engineering is a must when talking about larger projects, but collecting this data is a bit of an ethical grey area. Preforming these measurements should not come at the cost of anyone's privacy. The metrics mentioned in this report have been relatively nom-invasive. There aren't any major ethical concerns with monitoring the code submitted to a repository, but one thing listed above could be a cause of ethical concerns is Hackystat. Hackystat must be instigated with the developers IDE, if the developer is not told beforehand that this has been done it could be unethical and an invasion of privacy, but if the developer is told beforehand there shouldn't be any ethical issue with its use. (Hackystat, 2018)

Ethical concerns spike when managers start wanting to monitor developer's screens and record them for future analyse, this should throw up major red flags for any ethical developer. Firstly, most will agree that this is a breach of any user's privacy. This will not only track what the user is doing in his ide but also what he does in a web browser. Measuring the amount of time, the developer spends not working on the project could be a useful metric, this method of doing so goes too far. (Anon, 2018)

The next issue with this method is the mental strain this would cause an individual. This would be comparable to having a manager staring at your screen form over your shoulder always. This would cause a drop-in productivity even in the most hard working induvial. The final point I wish to get across is that measuring these metrics too closely can have negative effects on your workers and might lead to increased stress levels among its members.

Conclusion

There are many ways to measure the software engineering process, you can analyse the process in terms of a team's progress or the quality of the software. Some of the metrics aren't perfect – particularly those related to measuring source code quality, and which are useful will vary from team to team.

Every project manager should take a step back from his team to decide what metrics to use. Some are more relevant than others, some will be harmful to your project if not used correctly. Ethical concerns must be addressed first and foremost. These metrics can be extremely powerful in increasing a team efficiency, but they shouldn't be relied on too heavily, or they could be detrimental in the long term.

Bibliography

Hackystat. (2018). *Hackystat*. [online] Available at: https://hackystat.github.io/ [Accessed 16 Nov. 2018].

En.wikipedia.org. (2018). *Source lines of code*. [online] Available at: https://en.wikipedia.org/wiki/Source_lines_of_code [Accessed 13 Nov. 2018].

Citeulike.org. (2018). *Searching under the streetlight for useful software analytics*. [online] Available at: http://www.citeulike.org/group/3370/article/12458067 [Accessed 15 Nov. 2018].

Nextlearning.nl. (2018). [online] Available at: http://www.nextlearning.nl/wp-content/uploads/sites/11/2015/02/McKinsey-on-Impact-social-technologies.pdf [Accessed 15 Nov. 2018].

Anon, (2018). [online] Available at: http://s3.amazonaws.com/academia.edu.documents/35963610/2014_American_Behavioral_Scientist-2014-Chen-0002764214556808_networked_worker.pdf?AWSAccessKeyId=AKIAJ56TQJRTWSMTNPEA&Expires=1479122144&Signature=pO6ZkNbSZgbNTqHMd7xyIdMV5iE%3D&response-content-disposition=inline%3B%20filename%3D2014_Do_Networked_Workers_Have_More_Con.pdf [Accessed 15 Nov. 2018].

Guru99.com. (2018). [online] Available at: https://www.guru99.com/cyclomatic-complexity.html [Accessed 17 Nov. 2018].