Measuring Software Engineering

Name: Sean Candon

Student No.: 16321521

An Overview of the metrics that can be used, the software and algorithmic approaches available, and the ethical concerns involved.

This report is designed to provide an overview of everything involved in measuring the process of software engineering. This includes the various ways that it can be measured and the metrics that can be used to assess engineering in terms of measurable data. It also includes the many software platforms that are available to allow one to do this as well as the algorithmic approaches that can be used. In addition, this report seeks to explore the potential ethical concerns that are involved with measuring this process.

1. Software Development Metrics

When discussing the metrics used to measure the software engineering process, a distinction should be made between the various different things that can be measured. They include the productivity of a software development team, the performance of the software produced by a team, the security of the software, and the quality of the source code itself. What follows is an overview of each.

Measuring Team Productivity

For a leader of a development team, the right piece of data at the right time can be crucial in nipping a problem in the bud before it becomes a much larger issue. However, measuring software development is complicated. There are a lot of potential key performance indicators (KPIs) that one might consider when leading a development team, and it can be hard to distinguish between those that are relevant and those that aren't. Of course, these vary massively from project to project, and depend on the development style, the product being worked on, leadership style, and more.

Now we will discuss a variety of different metrics for measuring the software engineering process. The following metrics are designed to give teams an idea about how much progress they're making, and will ideally inform decisions about how to improve the process and make it more efficient. The following metrics don't measure the quality of the software (or, for that

matter, the source code), however they can be invaluable during the development process. We will begin by discussing team velocity.

-Velocity

Velocity is a measurement for work done by a team, and is typically used in measuring productivity in agile software development. Essentially, team velocity measures how many software units a team finishes in an interval. To better understand it, it's worth clarifying what is meant by this terminology.

The value of a unit varies from team to team, and is chosen by the team itself. This unit of work can be anything, from something easily measurable like hours worked or something a bit more abstract. Because what a unit is is defined by the team, velocity is best used as an internal metric, and isn't very insightful when comparing and contrasting different teams.

By interval we mean the length of time of each iteration over which the velocity is measured. Again, like a unit, the value of an interval is decided by the team itself, and can be a day, a week, a month, or anything else.

Because this measurement is so relative, the hard numbers that result from it don't matter too much. Like most software development metrics, it's not so much the numbers that count, but the trend, and that is very much the truth in this case. It can be a useful metric for tracking the progress of a team and establish a baseline for the pace at which it is working. This would of course be helpful in setting delivery expectations, and whether or not the pace is slowing down or needs to be sped up.

-Cycle Time

Cycle time is another software development metric, and can be a very useful one. Shorter cycle times often mean an optimized development process and a faster time to shipping the product to market, while longer (or - perhaps more importantly – lengthening cycles) usually means the process is inefficient.

The definition for cycle time is rather simple. It is merely a measurement of the amount of time a team spends working on an individual task or issue. The word "cycle" is used because the metric measures how long it takes for a team to cycle a task from a start state to a finish state. Of course, within tasks there are often smaller tasks, and those smaller tasks have their own cycle time. This means that if you choose to you can measure the cycle time for your entire development process, or you can choose to focus on an isolated section of that process and measure its cycle time.

Like velocity – as discussed above – once you establish a baseline when measuring Cycle Time, variations in that baseline can be very informative about the progress of the team. If, for example, a bug fix's mean cycle time is about three days, but then suddenly that number doubles, you know that there must be a problem.

Of course, the problem could be anywhere in the process, which is where measuring the cycle time for each key step in the process, large or small, becomes so useful, as it allows you to pinpoint where the problem is originating and come up with an effective solution.

-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 requests in a repository can therefore be useful. Any version control tool you are using, from Github to Bitbucket, allows you to retrieve this number, and it can give a good indication of the overall pace the team is working it. If the number of open pull requests increases from week to week, the team's throughput might be slowing down. Tracking this particular metric can be useful in spotting bottlenecks in the process, and may indicate if more time or resources should be dedicated to code review.

-Code Churn

Code Churn is a representation of the amount of code that was modified, added, or deleted in some designated period of time. In other words, code churn is when an engineer rewrites their code in some period of time, typically a short one. This metric can be useful because the number of lines of code written can be a deceptive measure of progress. If a lot of code is being written and pushed, but that code is just a repetition of previous code, no progress is being made. More important is the amount of productive code that is being committed, and code churn can help us distinguish between that and useless code.

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.

Code churn doesn't necessarily have to be consistent across the entire development process either. In fact, you can expect the code churn to be far higher in the beginning of the project than at the end, as developers try out different approaches to the overall task. As the process

continues, developers choose a path and box themselves in, leaving them with fewer options, leading to lower churn rates.

-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.

Measuring Software Performance

The following metrics related to measuring the performance of software will tell you nothing about specific features of the program or users affected by its failure. However, despite this these metrics can still be useful in measuring software quality, and aren't complicated to use in order to obtain useful data.

-Mean Time Between Failures (MTBF)

MTBF is the predicted elapsed time between inherent failures of a piece of software during normal system operation. In other words, it's a metric that measures the time interval between when a system failure is fixed and when a new failure is detected. Therefore, this is a metric that measures the time interval when the system is actually running. This metric can be useful, as it allows us to determine how stable the system is and estimate roughly when the next failure will occur. The latter means that maintenance teams can be prepared for the next system failure to occur.

-Mean Time to Recovery/Repair (MTTR)

Mean time to recovery/repair is defined as the average time required to troubleshoot and repair a software system and return it to its normal operations. It is a helpful metric for measuring the maintainability of a system, as well as its reliability. In order to calculate MTTR, it can be expressed mathematically as total maintenance time divided by the number of repairs over some specified time period.

-Application Crash Rate (ACR)

This metric is obviously related to the previous two, and measures the rate at which a system fails. It can be calculated simply by dividing the number of times an application crashes by the number of times that application has been used. It is a simple metric that isn't as helpful as the above two in predicting failures or measuring reliability, but it does provide a superficial look at how a system is performing.

Measuring Software Security

This section is strongly related to the previous one on measuring software quality, as software security is a crucial component of that. Unfortunately, it's also a component that is often overlooked. Security metrics must, like the one's above, be tracked over time to show if and how software development teams are developing security responses.

-Endpoint Incidents

This is an obvious metric to use, and while it is very simple, it's also probably the best in measuring how secure a piece of software is. 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 time period, as well as how often an endpoint suffers from such an issue.

-Mean Time to Repair (MTTR)

This is the same metric that was discussed in the previous section. In security terms, MTTR specifically refers to the average time elapsed between a security problem being discovered and the eventual resolution of the problem with a working fix. Of course, this metric should be tracked over specific time intervals, and should be looked at in comparison to previous values. If MTTR is decreasing over time, the software development team is becoming more effective in adapting to security threats and coming up with working solutions to them. On the other hand, if MTTR is increasing over time that means that the team is becoming less effective at handling security threats, and it may indicate that the team is overwhelmed.

Measuring Source Code

One additional way that software engineering can be measured is by analysing the source code itself. There are many metrics that exist to do this, and they range from the very simple (something like lines of code, for example) to the more complicated, such as cyclomatic complexity.

-Lines of Code

Lines of code is an incredibly simple metric, and is obviously deeply flawed. 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 actually are. In fact, writing more code than necessary might in the end be harmful to the quality of the software, as in the most extreme cases it can lead to an increase in the program's complexity.

-Cyclomatic Complexity

Cyclomatic complexity is a valuable software metric that indicates the complexity of a program. Simply put, it's a measure of the amount of linear independent paths through a program's source code. Typically, cyclomatic complexity is used as a software metric by getting the average cyclomatic complexity per method in the code.

Cyclomatic complexity per method in a program is a very useful and insightful metric. This is because the more possible paths there is through a piece of code, the greater the number of tests required to properly verify that the code is working properly. Greater complexity makes it harder for developers to test for edge cases, because the number of edge cases will increase with greater complexity. The less complex a piece of code is, the easier it is to test, and the likelihood that it will fail once the software is out in the world is lessened.

2. Computational Platforms Available

There are a number of platforms available for developers to properly measure the quality of their code. One approach is to use one of the many available open source automated code review tools such as Codacy and Scrutinizer. Below is an overview of several of such platforms, and the various benefits and drawbacks of using each.

Code Review Tools

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

Codacy is used by some major companies, including the likes of Paypal and Adobe, to enforce their respective coding standards. It also comes with a built-in docker analysis, as well as 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. The programming languages supported by Codacy are as follows: Scala, Java, Ruby, JavaScript, PHP, Python, CoffeeScript, and CSS.

There are several other tools like Codacy that are available, though Codacy seems to be the gold standard code analysis tool among developers. There's Codeclimate, which is a pretty good alternative, but is limited in many ways that Codacy isn't. For example, Codeclimate's API is still in Beta, and is unpredictable. It is also one of the most expensive tools on the market, and it doesn't provide helpful or detailed descriptions about problems with the code.

Another option is Codebeat, though that platform is particularly weak when it comes to security checks. In fact, it does not perform any security checks whatsoever. There's also Scrutinizer, which is a tool that lacks predictability, and while its API has been praised, it is only available to those who are subscribed to the most expensive package. Also, with Scrutinizer, you pay per container, with each container being a single task that can be run at once.

Hackystat

Aside from the above options, Hackystat is another open source framework for analysis of the software development process, though it differs from those above in some key ways, which are detailed below.

Where Hackystat differs from Codacy and the like is how it collects its data. Instead of collecting and analysing code upon each commit to a repo (meaning that it would be integrated with some version control tool), it instead requires its users to integrate it with their local development tools. Hackystat will then "unobtrusively" (according to their website) 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.

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 in particular 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 particular 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.

3. Algorithmic Approaches

This section of the report will spend some time going into greater detail of a few of the algorithmic approaches available to us when measuring software engineering. Each of the below algorithmic approaches have been mentioned and discussed briefly in this report.

Lines of Code

This report has previously discussed the various 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 take into account 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 fairly reliable. You no longer have to 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.

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 particular 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.

4. Ethical Concerns

Of course, when in the process of measuring the software engineering process, there are many ethical concerns to be taken into consideration. Measuring software engineering is incredibly useful for everyone involved in a project: the developers, the team leader, the client, and of course ultimately the consumers of the product being developed. The first section of this report highlighted many ways in which measuring the process can be so beneficial. But performing such measurements should never come at the cost of anyone's privacy or well-being.

The various metrics that have been detailed in this report have been rather non-invasive. There aren't any ethical concerns to monitoring the commits a developer makes to a repository and using that data to calculate metrics that either evaluate the quality of the code being committed or give an indication as to the progress the team as a whole is making.

One thing listed above that could theoretically run into some ethical issues is the software analytics tool Hackstat, which requires developers to integrate it with their IDE in order to gather data. In this way Hackystat differs from most other analytics tools, which collect data upon the commitment of code to a repository. However, Hackystat only monitors the activity in the IDE it has been integrated with and nothing else on the computer, and as long as the developer is aware beforehand that their work will be monitored in such a way, there shouldn't be too much of a problem with this particular way of measuring.

Ethical concerns can have as much to do with employee well-being as it does with their privacy. To look at an extreme example of software development measurement, in "How Effective Developers Investigate Source Code: An Exploratory Study", the screens of engineers are recorded as they're asked to perform some task. Obviously, this methodology brings with it several ethical concerns.

Firstly, most would argue that this method is an egregious breach of the developers' privacy. Not only would the work that the employee is doing in their development environment be monitored, so would the employee's activities outside that environment, say in a web browser. While measuring the amount of time a developer spends away from his/her environment might be a useful metric (though that is questionable), this method would go one step further, as not only would it see when and for how long a developer is at their workstation not working, it would also see what they're doing.

Of course, the other major concern with this method is the potentially harmful psychological affect on the developers. This method would be equivalent to having their manager stand behind them looking over their shoulder all day, every day, watching every little thing they do. The vast majority of people would not be able to work under such conditions, given the amount of unnecessary pressure they would be under at all times, and even if they were somehow capable of working, their work would very likely be subpar.

But even in less extreme cases than the one above, ethical issues have to be considered. For example, one might consider the potential effects of using an inaccurate metric to measure a team's progress or an individual developer's contributions. If a developer is making a significant contribution and the metric doesn't reflect that, that will affect his/her morale, and eventually perhaps have a negative impact on their mental health.

Consideration must also be given to what kind of data is being saved about developers and/or consumers, and how secure that data is. In the case of developers, employers are potentially storing very sensitive information about them, and the if there were to be a data breach the ramifications would be significant.

One other thing to consider would be that focusing too much on metrics and measuring development might lead to increased stress levels among the team members, as each becomes more concerned with meeting certain measurement requirements rather than completing the task at hand. Obviously, this would have a negative impact on the team's true progress and the quality of the software.

5. Conclusion

There are many different ways to measure the software engineering process, and many different angles from which to approach it. 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, and from project to project.

There are many platforms available to track these various metrics, especially those related to code quality, though most of these are imperfect, so it's perhaps better to use the metadata provided by IDEs or version control tools to calculate the measurements you feel you need. Some of these metrics and platforms – as well as the possibility of focusing too much on measuring the software engineering process – come with ethical concerns that should be addressed. These metrics can prove to be extremely useful in increasing team productivity and the eventual quality of the software, but they shouldn't be prioritised over the privacy or well-being of developers, or anyone else.

Sources

https://blog.usenotion.com/8-essential-software-development-metrics-for-team-productivity-273737868960

https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams

https://blog.ndepend.com/how-measure-lines-code-lets-count-ways/

https://www.guru99.com/cyclomatic-complexity.html

https://www.netguru.co/blog/comparison-automated-code-review-tools-codebeat-codacy-codeclimate-scrutinizer

https://hackystat.github.io/

http://olgabaysal.com/teaching/fall16/comp5900/slides/Paper1_Prasanthi.pdf