Measuring Software Engineering

Ben Folan

18322674

How can SE be measured

Software Engineering is a complicated, and sometimes unpredictable process that can be difficult to measure. Typically, the simplest methods used to measure it are also the most flawed, rewarding consistency and volume over efficiency and clever problem solving.

The most primitive method for measuring progress in software engineering is also the most primitive method for measuring progress in any task. That is to simply watch whoever it is doing the work to ensure that they do what they're meant to be doing, as opposed to getting distracted or just generally slacking. This is achievable by simply having the supervisor and the manager work in the same room. Most companies would rather place a camera in each office to monitor workers, but this is illegal in Ireland so compromises would need to be made. However, this practice is perfectly legal in the US and is employed in almost every office space.

The most obvious downside to this in respect to software engineering is that it doesn't monitor the work that is being done. Instead it just ensures that the person sat the computer is working (or pretending to work). It's much more useful to monitor the work that is being done by employees, rather than the employees themselves. There are a plethora of methods that have been devised to fulfil this purpose, with some that are clearly superior to others.

The simplest of these methods would be to count the number of lines an engineer has written. At a first glance this seems to make sense. The more code produced, the closer your product is to being completed. The biggest problem with this method is that it neglects to factor in the efficiency of the code or if it even works. This method also ignores the time spent debugging software, where lines of code may have to be altered or removed entirely, leaving you with a more functional, but shorter piece of work.

Something this method also fails to notice is the readability of the code, or how easily expandable the code may be. Most of the time spent writing code is maintaining and expanding already existing code, as opposed to writing code from scratch. Long, inefficient code can be difficult to understand, even if it has been well commented, as the method being used to achieve said goal most likely isn't very intuitive. This means future programmers will be spending more time trying to understand the code they're working with than they'll spend working on the code. This method of measurement encourages programmers to write code that in the long run, slows down progress, which is the exact opposite of what anyone would be trying to accomplish.

Another method that can be employed is to count the number of commits made to a repository. This focuses more on how many changes a programmer is making to a code base, which is a step in the right direction, but I believe is

also quite flawed. A focus on commits accounts for the number of changes being made to the code, as opposed to how many lines those changes are adding to what's already there. The obvious benefit of measuring development this way is that there isn't an incentive to write long and awful code. However, it doesn't account for how useful the code really is.

Any useful method for measuring software engineering is going to be rather complicated, and it will require multiple measurements to be combined in order to give a bigger picture of what's actually happening within a team.

What platforms can be used to gather and process data?

Pluralsight Flow (previously known as GitPrime) is a tool used for collecting metadata from Git repositories and presenting them in easy to read reports to enable teams to better organise themselves.

Setting up the program is quite simple. Once you've set up your Flow account, you create an integration with whichever Git tools you may be using, such as Github or BitBucket. You can also import any repository that's accessible over HTTPS or SSH from any server. Repositories can't be assigned to specific applications, but they can be sorted with tags. Contributors can be organized in a similar fashion, by being hidden from reports, assigned teams, or even merging said teams to create new ones.

Flow originally only imported data from Git, which means all they could look at was source-code, and it lacked any ability to view data on a team's collaborative efforts. This enabled them to integrate with a larger variety of tool kits but made practices such as code reviews invisible to any measurements in place to track progress. Packages have been released to track such tasks but they're yet to be fully implemented.

Flow has its own set of statistics that it generates using its own proprietary algorithms

- **Impact**
  This is a metric that measures the difficulty of a change made to the codebase. This is calculated based insertion points, the lines of code altered, as well as the number of files that were altered.

- **Churn**
  This measures how much code has had to be re-worked. This measurement is useful in determining how efficient a team is and how much of the effort spent on writing code has been wasted by writing bad code.

- **TT100 Raw**
  This measures how long it takes an individual to produce 100 lines of code, regardless of the quality. This measurement on its own isn't of much value, as the information displayed doesn't take the usability of the code into account.

- **TT100 Productive**
  This is a similar measurement to TT100 Raw, except it only looks at the usable code, and ignores any code that ends up getting scrapped. It's recommended that managers look at both TT100 Taw and TT100 Productive to get a more accurate view at how fast their team is working.

  Flow has 4 base metrics that it uses to track the individual performance of contributors. These metrics are active days, commits per day, impact, and efficiency (the latter two being correlated to the previously mentioned impact and churn).

  Another tool that fills a similar role is Velocity, which also tracks data from Git repositories.

  Setting up the application is as simple as logging in with either your Github or Bitbucket account and adding your repositories. Velocity is more focused on pull requests, therefore it has built in support for both code-level data as well as team efforts, such as code review. This makes both high level and low-level analyses easily accessible.

  Some of the metrics used to make said analyses are similar to those found in Flow so I won't bother to mention them again. However, they track some data that Flow doesn't. This includes:

- **Pull Request Activity Level**
  This tracks how much of a team's effort is being spent on any given pull request. This is done by tracking the number of comments, review cycles, etc. that a pull request has accrued. This helps managers figure out what aspects of a project are on schedule and what is likely to be the cause of a delay if left unchecked.

- **Review Cycles**
  This is a count of how many times a certain pull request has gone back and forth between the author and a reviewer. Typically, a pull request with a high Review Cycles count is causing issues and may need to be reconsidered if it's proving to be more complicated than originally anticipated.

- **Time to Open**
  This is a measurement of how long it takes for a contributor to write code, push it to a branch, and then open a pull request. This is a useful metric for managers to have as it gives an insight into how fast their team can take a problem and then implement it to a level where they feel comfortable for it to be reviewed and possibly merged.

  Velocity also has an easy to use reports builder that lets you pick whatever data set you want and how you want it to be displayed. As well as this, you can pick how you'd like said data to be summarised (average, mean, sum, etc.). These graphs can then be dragged and dropped in order to display them in any way you please, helping you to clearly visualise whatever you're trying to display.

What algorithms can we use?

Given the complicated nature of measuring software engineering, the use of any one algorithm will prove futile in presenting a result that truly represents the progress that has been achieved by a team. Multiple algorithms would need to be employed to generate individual figures, but those numbers on their own still won't be enough to draw any kind of reasonable conclusion. These results would need to be pieced together somehow, but the method for doing so is another challenge. It's quite commonplace for neural networks and machine learning to be employed when problems that are this complicated and unclear need to be solved.

These kinds of algorithms (or rather, collections of algorithms), consist of multiple interconnected nodes that simulate neurons in the brain, and aim to solve problems in a manner that is similar to how the human brain would. These kinds of solutions are quite powerful as they are easily scalable and can process large swathes of data with relative ease.

Machine learning, as a concept, terrifies me and I don't think I'll ever be truly comfortable with it, but I'd be lying if I said it wasn't useful in the creation of these tools. This is a branch of neural networks that learns to do its job better the more it's used. This is useful in software engineering as the exact metric for "success" isn't exactly obvious so telling a computer what to look out for

isn’t very straight forward. Instead, this method allows the algorithm to learn from experience by spotting patterns that lead to either success or failure.

If set up correctly, this could help in tweaking the tools in order to produce the most accurate data possible.

Is this ethical?

Whether or not it’s ethical to monitor programmers (or anyone) to this degree is a highly debated topic that could be argued very convincingly from either perspective. I’m personally somewhere in the middle but if pressed I’d say I’m opposed to the idea of being monitored. There are many arguments both for and against these practices and I’ll discuss both, starting with the arguments for it.

The most immediate benefit of these measurements is that managers can very easily identify who on the team is pulling their weight and who could possibly need to be replaced. Given the competitive nature of the industry, employers will want to have the best talent possible working for them, allowing them a faster turnover, which will in turn, make them more money. Being able to identify who on the team is the most competent will also allow the project manager to more effectively assign tasks, ensuring that whatever work needs to get done, gets done as quickly and as efficiently as possible. This done by assigning the tasks which the manager suspects will require the most work to their team members that they believe to be the most capable of finishing that amount of work, whilst off loading as much of that work as they can to the rest of the team, ensuring no one has been assigned more than they can manage. Some managers may have unreasonable expectations of their team, but that’s a different discussion altogether.

Aside from determining who the outliers are from the team, these kinds of measurements can also determine who in the team is better suited for which tasks. Different developers are all going to be more comfortable with different things, be it languages, algorithms, packages, or a combination of things. If a manager knows that person A is a faster worker but person B has a better understanding of the language that needs to be used (let’s say Python), then they’ll be able to take that information and make a much better informed decision on who to assign the task. Also, they now know who they can assign the task to if their initial pick doesn’t work out in the end.

On another note, these methods of measurement could also be used to determine how effective the members of your team communicate with one another. This can be seen quite easily by comparing what your team members have said they want to be made/fixed and see if the code that the rest of team

have been writing matches that description. If a product isn't built to the correct specification, then it may as well not have been built. This is why communication is key when building software, and being able to recognize when it's effective and when it's lacking are vitally important.

The opposing arguments to any of these points always lead back to the lack of privacy for those being monitored. In recent years, what is deemed to be private information and what isn't has changed quite significantly, and most corporations would have databases full of their clients private info, whether that be their name, address, phone number, PPS, or anything else they wouldn't be comfortable handing out to a complete stranger. As well as this, patterns in the way people interact with companies and their purchase history are also used to target ads at consumers. A notable example of this is in 2012 when Target would target ads for baby products to people who were pregnant but were yet to even realise it themselves. This kind of data collection and processing is an incredibly powerful tool, one that's powerful enough to make incredibly accurate predictions of what's happening in someone's life, as well as what might be going through their head, whether that be consciously or subconsciously.

This level of surveillance has been in the world of software engineering before and is quite prevalent still. IBM have claimed that they use A.I. to determine which of their employees are searching for a new job, with claims of it being up to 95% accurate. The scary part of this is that it could leave IBM employees stuck if they don't manage to find another job. If your employer sees you as a flight risk, they might see little value in presenting you with new opportunities, such as roles on upcoming projects. From their perspective, they see the possibility of having to replace a member of the team half-way through development and would rather assign the role to someone who will stick it out.

Whether you're for or against the use of these methods of monitoring and measuring software engineering is simply a matter of your own priorities. People who value the accomplishments of a group of people over the freedoms of the individual will typically be in favour of the use of such technologies, as they ensure both quality and consistency from teams that are developing software. On the contrary, those who value their individual freedoms over anything else will view these tools as an invasion of a person's privacy. They'll point to the past when people's lives weren't documented on the internet and speak of how it was deemed unacceptable to open another person's mail, listen in on their phone calls, etc. All that's changed between now and then is that said exchanges happen over the internet instead of in

physical form, and I'd be inclined to agree with the people arguing for the world to return to such views.

These tools can be used to aid in many tasks, a lot of which have no moral ambiguities. The problems start to arise when it's made apparent what these technologies are capable of achieving, and I'd sooner live in a world without the good they can offer, than to have their benefits but also be weary of the fact that almost everything I do can be very easily monitored and processed, whether that be in or outside of work.