Measuring Software Engineering

A Report by James Tait

The accurate measurement of a Software Engineer's productivity is important for many reasons. Employers want to know if they're getting enough bang for their buck, and to be able to give consistent and reliable time estimations to their customers. Of course, as with most things in this field, it's easier said than done. None of the techniques developed so far have been proven to be very *accurate*, which is usually accredited to how complicated the process of Software Engineering is. Most, if not all, these techniques have actually been seen to be *ineffective*.
So some people have gone so far as to take the stance that Software Engineering is 'simply too complicated': it **can't** be measured properly, and we should leave our software devs alone.

Personally, I think these people are giving themselves too much credit, and just because we haven't found the 'Holy Grail' of productivity measuring *yet*, that does not mean there isn't one. Furthermore, if in fact we can't ever find some perfect system, it is still worth exploring and refining the systems we have, and just taking their output with a pinch of salt.

Measuring the Process

So what are the metrics that we can measure as we strive for the perfect model of developer productivity? What kind of data can we collect to help us rank employees in a team or even teams in a company? There are some obvious and easily monitored metrics, but they often have obvious and easily exploited downfalls.

You can measure a Software Engineer's productivity by the number of lines of code they produce. Although you shouldn't, because this rewards inefficient and inelegant code, exactly the opposite of what you want. A developer can pad out their code with very little effort, and appear to be more 'productive' without actually doing any extra work. This also encourages repeating sections of code rather than putting the section into a function, increasing the risk of bugs and reducing the maintainability of the project.

Maybe instead of lines of code, you opt to measure the number of commits your developer makes. At first glance this seems more reasonable; each commit can have a variable amount of code in it, but they represent stable stages in development when *something* is added successfully. The problem is that different developers commit at different rates. Plus, this analytic is also easily skewed by committing more often over the course of the same amount of work.

Another metric is the number of bugs a developer solves. It almost makes sense to measure productivity like this, except that not all bugs are fixed with the same level of ease, or with the same amount of work. On top of this, a particularly conniving contributor could purposefully write bugs in to the project for themselves to solve later, with the intent of passing it off as a boost in productivity deserving of a raise. *Clearly*, this metric won't do either.

Code churn is when a developer rewrites their own code within a short period of time, i.e. they are producing a lot in terms of raw amount of code, but the value they are adding in comparison is very little. They are contributing the same code, just written differently. This can be used to measure an engineer's productivity: the total code contributed minus the churn is productive code. You can also assess how efficient the engineer is being by the ratio of productive code to churn. This is a good metric to keep track of, but certainly not the answer. "Productive code" can still be padded out by the developer to make it seem like they're doing more than they actually are.

A slightly different approach is to abstract away from the software itself and to ask for time estimates at the start of development. The developer should give their best guess as to how long the task will take, and then once the task is done, you can see how accurate their estimate was. What ends up happening though is when the task isn't completed within the estimated time, the employee is berated, or conversely when the task is completed ahead of schedule, the employee is praised. This leads to everyone over-estimating time scales, making it look like they're getting things done quickly, even when that isn't genuinely the case.

Theoretically, there are lots of metrics to be measured, but as you can see it's not so easy to extract reliable information from them. So one can understand where people are coming from when they say you **can't** measure software engineering, it is 'simply too *complicated*'.
What do you do instead? Well they suggest you measure things like impediments, or delays outside the control of the developer, or customer satisfaction, or even employee satisfaction. An employee will be more productive if they're happier doing the work, and if the customer is happy, it follows that the work the employee is doing is good.

This isn't a totally satisfying solution though; It doesn't provide fine tuned analytics we can run software over that spits out candidates for raises and promotions, or alternatively for the chopping block. It seems to me that the 'measure happiness' approach more accurately answers the question "Are my software engineers being productive?", and not "**How** productive are my software engineers being?". Software engineering is undeniably a complicated process: no single metric is enough to measure a developers productivity, rather a certain combination of them all, each weighted accordingly, is needed.

Computational Platforms

In pursuit of a system that accurately measures the productivity of software developers, a number of tools have been developed. They range from being totally manual, to totally automatic, with everything in between. When choosing a tool to use for your team or company, it's important to consider what metrics it measures, what metrics are relevant in the context of your projects, and how much overhead you are willing to put up with for the data the tool provides.

The PSP (or Personal Software Process) as detailed in the book *A Discipline for Software Engineering* by Watts Humphrey uses manual data collection and manual analysis. This takes substantial effort to maintain: 12 forms to be filled out, with 500 distinct values in total. After using this for two years, the Collaborative Software Development Laboratory (CSDL) at the University of Hawaii set out to design their own piece of software, the LEAP toolkit.

The goal of the LEAP (Lightweight, Empirical, Anti-measurement dysfunction, and Portable) toolkit was to address data quality problems with the PSP arising from human error during manual data collection and analysis. Users still entered much of the data manually, but the subsequent analyses were done by the software. CSDL found that by introducing this automation, some analytics were easily obtained, but it actually made certain other analytics more difficult to collect. And so after several years of using this tool, CSDL came to the conclusion that the development overhead generated by their toolkit often wasn't quite justified by the insights it gave in return.

CSDL went in a different direction for their next project: Hackystat. They wanted to make the tool as unobtrusive as possible, by implementing automatic client-side data collection systems a developer could attach to their editor, build tools, and test tools. This got rid of the interruptions to productive work that come about when manually collecting data. Another big feature of Hackystat is how fine-grained the data collection is. It can track the software engineer editing a method, constructing a test case for that method, and running that test. This means it can shed light on the extend to which the developer is adhering to Test Driven Development (TDD) practices.

Zorro is a software based on Hackystat, that focuses specifically on TDD. Also developed in CSDL, it actually automatically detects the extent to which a developer is complying with TDD practices. It measures data on a second by second basis, allowing you to extract exactly how long your developer is spending on each task, and how effectively they're completing them.
However some developers are uncomfortable with how detailed and fine-grained the data collection is; especially when it is so automated, and the software isn't explicit when doing all this data collection.

Moving away from TDD, Jira is a platform developed by Atlassian for managing projects in Agile development teams. Purportedly used by big companies such as Ebay and Spotify, it allows you to define your workflow, plan out projects (with things like roadmaps with time estimates), and then track the work being done. The data collection is mostly manual, and it seems much less finely-grained, but the platform still provides an insight into the progress being made with a project, as well as who's making said progress. Due to this slightly more superficial perspective over projects, developers are often much happier to work with this than with tools like Zorro.

GitPrime is a computational platform built specifically to extract developer activity data from Git-based repositories. In contrast to Jira, GitPrime automatically generates reports detailing things like commits and code churn, allowing for a deeper insight into what a developer is accomplishing and how they compare to the rest of their team. The automatic nature of it allows for easy integration into a dev team's workflow, even for already existing projects. Of course it is *somewhat* limited in that it only works for Git based projects, but at least in my mind that's a very small caveat.

There is a plethora of computational platforms out there, each with their own level of automation and scope of metrics. Obviously they also all have their own advantages and disadvantages.

Generally speaking, the more manual in nature the tool is, the less return on investment you get due to the subsequent overhead. On the other hand, the more automated the tool's data collection is, the less willing developers may be to use it (due to the potential lack of transparency).

Algorithmic Approaches

Over the past number of years, there have been many different algorithms designed for the software development process. Despite each algorithm being deployed in many distinct teams around the world, the effectiveness of them all seems to still be up in the air. Nevertheless, lots of these teams absolutely swear by their respective development algorithm and are abhorred by teams who code without some similar sort of system.

Agile development is an approach to software engineering where requirements and solutions evolve over time based on continuous revision of goals, taking input from both the project team and the customer(s). The work is broken up into short time frames called iterations or sprints, and after each one the product is demonstrated to stakeholders. The iterative approach, as opposed to a more classic waterfall approach, allows for progress to be measured regularly. This in turn yields incrementally more accurate time intervals, as well as minimizing any product straying too far from the given spec (which is also iteratively revised).

Scrum builds on the Agile philosophy with a more rigid layout for how to operate. The work is split into tasks that can be completed within one iteration (which is usually defined to be two weeks), and fifteen minute *daily scrums* are held each morning to track progress and re-plan. At the end of each iteration there is a sprint review: an event lasting approximately two hours where the team reviews both the work completed and not completed during the last sprint, demos any functioning work done to the stakeholders, and then collaborates with the stakeholders to decide what work needs to be done next. *Story points* are the most often used unit in scrum as an estimate of the time and effort a particular piece of work will take. Theoretically, the more story points a software engineer can tick off, the more productive they've been.

Studies have shown that taking on too much work can make you less productive. Kanban is an approach to software engineering whereby the individual work tasks are allocated as capacity permits, rather than being allocated upon creation, essentially aiming to reduce bottlenecks for the developer in hopes of increasing overall productivity. This system is actually sometimes used in conjunction with scrum, and there is significant overlap between *kanban boards* and *scrum boards* which visualize tasks moving (from left to right across the board) through different development stages, such as backlog, in progress, peer review, in test, or completed.

Test Driven Development (TDD) is wholly different algorithm. Put simply, you start development by making tests (that fail), you then 'hack' together some code in any which way that you can such that the tests pass, and *then* you refactor your code to be more efficient/elegant/structured, all the while making sure the tests still pass. It has been found to increase code stability, but claims have also been made that it doesn't help at all. I think its effectiveness depends greatly on the context of the project it is being applied to. The Zorro paper suggests that the simple description of the algorithm (stated above) doesn't describe it fully, and it needs to be adapted to the context in order to increase stability and productivity as much as people claim it can.

Unfortunately, properly ascertaining how much value you gain from practicing these algorithms is very difficult; there are lots of variables that can skew your observations in many directions. It can also take a long time before the effects of such systems show; even if a perfect software engineering algorithm has already been laid out, years could pass before we actually realise it.

Ethics Concerns

Data collection always raises ethics concerns; the majority of people find it invasive, and that it breaches their privacy. In philosophy, the Hawthorne Effect states that a subject's behaviour can change due to nothing more than the fact that some aspect of the subject's behaviour is being measured. This raises a few questions; is the data more authentic when you don't disclose that you're collecting it?, and is it ethical not to disclose that you're collecting it? So long as all subjects of the analysis are made aware that they are being analysed, and they subsequently consent, I believe that analysis is ethical. The 'quality' of the data collected may or may not suffer for it, but in order to remain ethical that bullet must be bitten.

Sources

- Searching under the Streetlight for Useful Software Analytics
- Automated Recognition of Test-Driven Development with Zorro
- The Myth of Developer Productivity
- Jira Official Website
- GitPrime Official Website
- Agile Development