Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Title: Measuring Software Engineering

Submitted By: Catalina Rete

Date: 14.11.2018

Submitted For:

Module: Software Engineering CS3012

Programme: TR033-Integrated Computer Science

Lecturer: Stephen Barrett

Contents

Objective

With the considerable increase in the number of software engineers needed in our current society, we start looking for methods of comparison between a productive engineer and someone who pretends to be productive. The aim of this report is to discuss the kind of data available for highlighting productivity, as well as the tools that are available to use in gathering this kind of data. Finally, we will conclude by weighting whether we can truly measure productivity in the field of computer science and if the data we get brings us to any valuable insight.

Measurable data

So how do we measure productivity? There seem to be a lot of things affecting software engineers and the way they work, and choosing one way of measuring over another could have unwanted side effects that could actually impact any company negatively. With a simple Google search, one can see that there are developers out there who believe that even trying to measure this data could have a bad impact on motivation and ultimately drive good people out of a company (vartec, n.d.). Even if there are good ways of measuring that could lead to useful data, there are quite a few particularly bad ones that are worth mentioning, as they come first into mind for anyone who thinks about coding.

Lines of code

When we were given this assignment, we were told to write 10 pages, perhaps if I write less I may still get some good grade for it, but by requiring everyone to write the same amount of pages instead of letting everyone decide how much they feel it is worth talking about the subject, it becomes an attempt at measuring how much effort each student puts in this assignment. Does writing more indicate that the report is better? Or are the rest of the words following on this page random and simply some thoughts based on no research at all? It's clear that even in writing a report, quantity doesn't necessarily mean quality.

In the context of software engineering, equating productivity with the lines of code written could have devastating effects. Rewarding developers for writing more code could lead to having broken and bloated code. Why would you reward someone for solving a problem in 3000 lines instead of 300?

This quantitative measure doesn't work the other way around either. If there's anything that I learned through my short period of internships, is that big companies would always rather have readable and easily maintainable code than having a really smart developer write a condensed and illegible line of code instead of 300. If no one can read your code, it is as good as useless in a team. To conclude that this metric is generally useless, there's a nice story about Apple Lisa team and how they found out that the lines of code meant nothing. [1]

[1] http://www.folklore.org/StoryView.py?story=Negative_2000_Lines_Of_Code.txt

Velocity

One may say that certainly each team has its own projects, and the team that completes them in the least amount of time must be doing something right. But this idea and technique for measuring performance ignores how different projects can be and the context of each team.[2] It can also encourage certain teams not to collaborate with others if it means completing their own tasks faster. Even on an individual level, surely it would be much faster to just do your work than stop and help others when they're stuck. This method could potentially increase the productivity and performance of certain teams, however it could also create an unhealthy workplace and hinder productivity as a whole. Not to mention that a faster developer could also produce unmaintainable code due to not having enough tests (as skipping tests could make you faster) or introducing bugs and not spending enough time on deciding the design of the project. Still, despite all these arguments, in a random poll on Twitter asking for developers' opinions, a surprisingly large percent (15.3%) thought the speed is an important metric. [3] But, what's even more surprising and important for the purpose of this report are how varied the answers were. This just shows that measuring productivity in computer science proves to be way more challenging than it seems at first sight.

Peer Reviews

Perhaps all the measures mentioned above are so bad because they can potentially create a workplace where nobody would want to work in, with selfish developers focusing on completing their tasks the fastest, with disregard for their colleagues. A lot of companies nowadays adopt agile practices, and the first value of the agile manifesto is: "**Individuals and interactions** over processes and tools"[4]. In other words, people in a team and company need to know how to work together effectively, and this is why a lot of tech companies have adopted Peer Reviews as a way to measure someone's performance. "The best way to be a 10x developer is to help 5 other developers be 2x developers" (Elliott, n.d.). It's often the case there are new people on the team, as software engineering is a quite fast-paced industry where people change jobs frequently. Most big companies also have their own internal tools that are confidential as well. A big part of someone's first few months in a company is learning as much as possible, but sometimes documentations and Stack Overflow are not enough. More experienced developers could use a lot of their time to mentor new employees or discuss bigger issues. This time could've been used to write more lines of code, close more bugs or finish more tasks. Instead, they are helping other developers become better at the job they're meant to do. This is a really important measure and it helps managers see how people interact with each other and whether they work effectively together or not.

[2] https://www.infoq.com/articles/measuring-tech-performance-wrong
[3] https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0
[4] http://agilemanifesto.org/

Tools Available

But how much time does a developer spend on fixing bugs? What about closing tickets, reading documentation? How much time does it take a developer from the start of a new project to when it's fully committed and in production? How many steps and what languages do they use? If we want to increase performance, first we need to check what things are affecting it in the first place and how an engineer can improve by assessing their behavior and interaction with tools and people. Below there's an overview of some of the computational platforms available.

PSP (Personal Software Process)

"Text-and-fix" is a strategy well-known in the industry of computing. In some cases, developers are even encouraged to write tests first, and then go on and write code, compile and run until it passes all the test cases. Some may argue, however, that there's too much time spent on testing the code and most of the time, there are still bugs and defects left even when the code goes to production. This is why Watts Humphrey came up with a new way of providing software quality and increase the productivity of the engineers at the same time. This new process is called "Personal Software Process" and it's basically a framework and a guide on how developers can understand their own performance and how they can improve it. It's based on the idea that developers should aim for quality code from the very beginning, instead of relying heavily on tests.

PSP puts forward a few principles[5]:

- All developers are different and in order to increase performance, they need to assess their own personalized way of working. A general framework would fail as it's not tailored for the working habits of different people. Engineers need to use personal, well-defined processes and plan their work before starting it.
- "The right way is always the fastest and cheapest to do the job", as it's more efficient to prevent defects than find them later with tests.
- The earlier you find defects, the cheaper it is to fix them.
- In order to produce quality products, engineers need to feel responsible for the quality of their work.
- Engineers must analyze the results of each job to find ways to improve.

The PSP process has multiple phases. First, developers must plan the execution of a project, then they must write a Design document. Before moving forward this design needs to be reviewed. Finally, we get to the coding part. The code needs to be reviewed as well, and only after all this we get to testing. The last step is to write a "Postmortem" describing the flow of the project, what went well, what went wrong and general logs of every step. So the developer needs to keep track of their own performance, note when they didn't meet a deadline or when there were defects in the code, they also need to record the size of the project, the timeline and the lines of

[5] https://resources.sei.cmu.edu/asset_files/TechnicalReport/2000_005_001_13751.pdf

code. This process has been taught in 6 different universities, and the results obtained confirm that it can indeed boost productivity. [6]

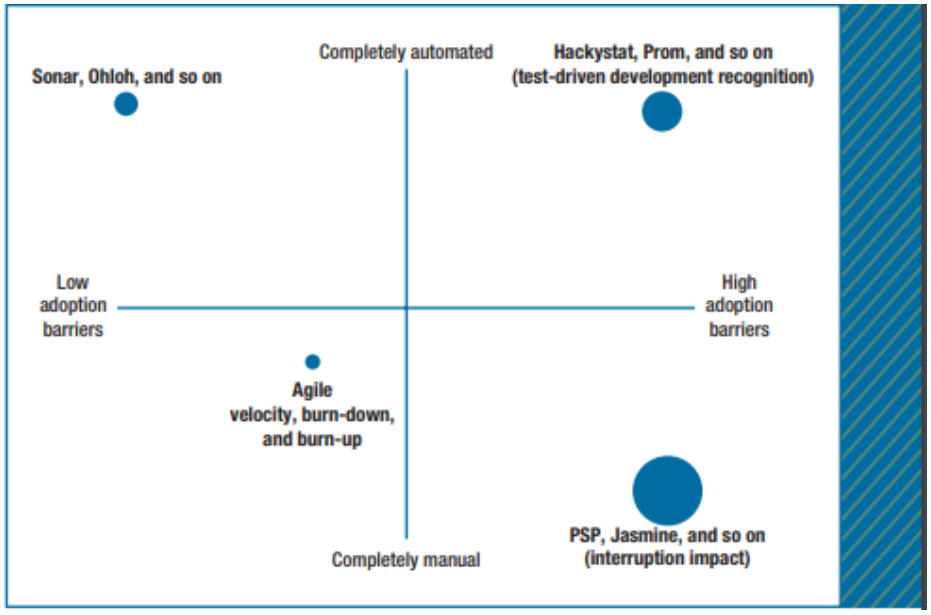
Hackystat

PSP sounds good, but it also sounds like a lot of manual work, thus it's no surprise that there's a tool out there that collects data for you (developers are fond of automating stuff, after all). Hackystat is "A framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data." [7]

This tool can be used for educators, practitioners (so engineers) and researchers. So how does it work? Hackystat uses software 'sensors' that are attached to development tools. These sensors collect (unobtrusively) and send 'raw' data to a web based storage called the Hackystat SensorBase. The SensorBase can be queried with other tools to generate visualizations of the data. The objective of Hackystat is to facilitate "collective intelligence" in software development and it tries to give insight and knowledge about the raw data it collects.

One of Hackystat uses is to test whether the developers are doing Test Driven Design. Test-Driven Development relies on short work cycles, writing tests and then restructuring the code to pass all the tests, followed by the same phases again until the code passes all test cases. Hackystat could track and notice these cycles.

Below, there's an overview of a few metrics collection tools and the corresponding level of difficulty when it comes to adopting them at industrial-scale, as well as how much automation is involved.[8]



[6] ftp://ftp.sei.cmu.edu/pub/documents/articles/pdf/psp.over.prac.res.pdf
[7] https://hackystat.github.io/
[8] http://www.citeulike.org/group/3370/article/12458067

Zorro

Zorro is a system built on top of Hackystat that automatically determines whether a developer is complying with the principles of Test-Driven Development (TDD). Hackystat provides a framework for analyzing the data received, this framework is called Software Development Stream Analysis (SDSA).

A lot of people claim that TDD can generate 100% code-coverage[9] and that it can improve code quality and reduce bugs. However, this is not always the case. Although when introducing it in Microsoft and IBM it did improve code quality and decreased the amount of defects, other companies found that it produced less reliable software. For that reason, Zorro is a useful system for diving deep down into TDD and checking whether it is indeed beneficial, or if other factors had a larger influence over the results of the companies that noticed a change.

So far, Zorro has been validated through the use in classrooms and out of the 92 episodes under study, 82 were validated as correctly classified, for an accuracy rate of 89%. Zorro proves that with the right tools, it is possible to analyze "micro-processes" in software development.

Algorithmic approaches

In the previous chapter, there was an overview of tools available for gathering data and metrics. However, 'raw' data is of no use. We have the lines of code, so what? We've seen before that simply the number does not indicate anything valuable. We have the time spent, the bugs, but how do we get to any useful conclusions using these metrics? Below, there's a few algorithmic approaches for analyzing software metrics.

Halstead's method for analyzing static and dynamic complexity of software metrics.

Metrics can be classified into static and dynamic metrics. While static metrics give information at the code level, dynamic ones provide information at runtime. "Software complexity is an estimate of the amount of effort needed to develop, understand or maintain the code."[10]

Halstead's metric takes into account the length and volume of a program based on the number of operators and operands. He uses the following measurable data:

- n1 = the number of distinct operators
- n2 = the number of distinct operands
- N1 = the total number of operators
- N2 = the total number of operands

Using these notations, he defined the following terms:

- Vocabulary(n) = n1+n2

---

9 http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=956E1624304502A89128F90EAF854128?doi=10.1.1.69.4525&rep=rep1&type=pdf

10 https://waset.org/publications/2684/static-and-dynamic-complexity-analysis-of-software-metrics

- Length(n) = N1 + N2
- Volume(V) = Nlog2N
- Length(N) = N1+N2
- Potential Volume (V*) as V* = (2 + n2*) log2 (2 + n2*) (the smallest possible implementation of an algorithm).
- Program level (L) as L = V* /V (The closer L is to 1, the tighter the implementation.)

With these concepts in mind, he concluded that code complexity increases as volume increases and as program level decreases. However, his studies have met criticism for several reasons including methodology and assumptions and the difficulty for computing the individual metrics, like counting the operands, every time the program gets changed.

McCabe's metric

Another way of analyzing software complexity is through looking at the Cyclomatic complexity. Instead of looking at operands and operators, Thomas J. McCabe considered that the conditions and control statements are the ones that add complexity to a program. It measures the number of linearly independent paths in a program's source code. There's two equivalent ways that can be used to characterize the metric:

1. The number of decision statements in a program + 1
2. For a graph G with n vertices, e edges, and p connected components, v (G) = e - n + 2p. [11]

Function Point Analysis

Function points are a unit of measure just like we use hours to measure time. This metric is applicable for a broad range of software technologies. It can be really helpful in calculating the size of computer software, estimating costs and project management, measuring quality and setting functional requirements. The analysis is conducted from a user's view and it consists of breaking the systems into smaller components[12]:

1. External inputs
2. External outputs
3. External Inquiries
4. Internal and external files

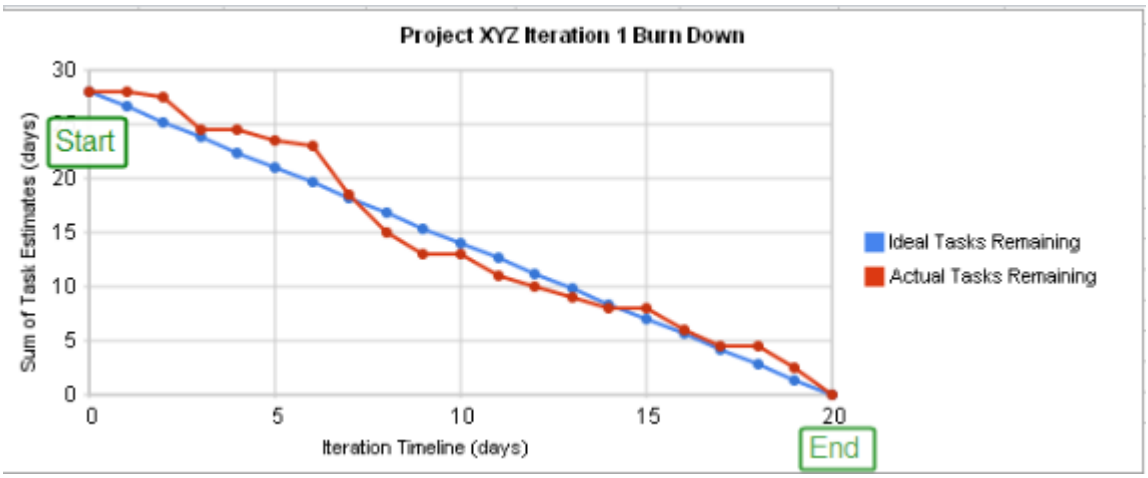
Once these function points have been identified, a complexity rating of the functional value for the end-user is determined. The factors in the analysis include whether the application will use data communication, whether there will be online data entries or online updates, etc.[13]. After

[11] https://waset.org/publications/2684/static-and-dynamic-complexity-analysis-of-software-metrics
[12] https://www.softwaremetrics.com/files/15minutes.pdf
[13] https://www.umsl.edu/~sauterv/analysis/function_point/FPARP488.html

adding all the points, they are considered “unadjusted”, until an analyst will assign a weighed value from 0 to 5, considering their influence on the development of a project.

Burn down chart

This method is not exactly algorithmic, but it’s an important way for keeping track of the progress of a team and also for estimating the time it takes to complete a project[14].



There are multiple ways to interpret a burndown chart. For the particular one above, at the beginning, the ideal and actual tasks start from the same point, but then they get recalculated as time progresses. For more accurate results, the charts usually take into account an efficiency factor that’s used when developers consistently underestimate/overestimate the deadlines.

The chart could also be used to measure performance by analyzing the ideal and actual tasks remaining. For other burn down charts, the workload is divided into story points[15], how small these stories should be is a debatable issue. One important principle is that a story point is not considered complete until the code has been written and tested, maybe even accepted by the manager/customer.

Others

There are a lot of other metrics that can be obtained from observing the workflow of a software engineer. These include:

- RTF Curve (Running Tested Features)
- Cumulative flow diagrams (CFD)
- Lead Time/Time in Stats (TIS)
- Fault Slippage

[14] https://en.wikipedia.org/wiki/Burn_down_chart
[15] http://idiacomputing.com/pub/BetterSoftware-BurnCharts.pdf

- Maintenance inflow and so on.

A table with a lot of other metrics is available in the following study:
https://pdfs.semanticscholar.org/9108/3db3d57fea06261d819e2a13ffc59adab0ee.pdf

Ethics

You could have developers simply put forward the data they would like to be analyzed, as you would in the PSP framework. This process however takes a really long time, developers need to complete around 12 forms and there are a lot of details that they need to manually calculate in order to fill in these forms. The creator of PSP embraced the manual nature of PSP: "It would be nice to have a tool to automatically gather the PSP data. Because judgement is involved in most personal process data, no such tool exists or is likely in the near future." (Humphrey, 1995). It's true that the manual nature of this tool makes the data very flexible, however it also makes it fragile and prone to error. Researchers at the Collaborative Software Development Laboratory (CSDL) at the University of Hawaii at Manoa have been studying Software Engineering metrics for more than 15 years. They also used the PSP framework in teaching for more than two years and their findings show that actually PSP has an important data quality problem. They analyzed more than 30.000 data values generated by the use of PSP in class[16]. Because of its manual nature, PSP sometimes led to incorrect conclusions, although the error rate was quite small (less than 5%). In order to solve this issue they looked into automated tools, including Hackystat.

The large issue with getting this data automatically is that if you want to get rich data that can provide insight into the way developers perform, you need to deal with privacy and overhead concerns. Hackystat is supposed to be able to gather data unobtrusively. This can be seen as a feature, however some developers didn't like the idea of adding sensors to their projects that could collect data they didn't know about.

Hackystat provides client-side, fine-grained data collection. These analytics are valuable, yet a software engineer called this product 'hacky-stalk', because it allows everyone to see the working behavior of their colleagues. In addition, this data is available to management and because it is so detailed, developers are uncomfortable with this kind of transparency, even though managers promise to use it appropriately. It's clear that highly automated tools that gather data from a developer environment are more useful the more information they can scan and analyze, but this also makes them more invasive in terms of privacy, making software engineers uncomfortable with the process. On top of this, the study was performed before GDPR came into effect, so my guess is that now people would care even more about management and coworkers being able to look at all their working habits.

Another practical issue with automated tools is that they need to adapt to a wide variety of IDEs and programming languages available and they constantly need to be updated to collect relevant

[16] P.M. Johnson and A.M. Disney, "The Personal Software Process: A Cautionary Case Study," IEEE Software, vol. 15, no. 6, 1998, pp. 85–88

information. They also don't offer as much flexibility as the PSP where everything is personalized and well-defined. If a developer felt that a factor hindering him from being more productive is constant interruption, it would be very difficult for a program to conclude this just by analyzing his coding behavior[17].

Increasing performance

While the main focus of this report is how we can measure productivity, I feel like there's a need to talk about what employers can do to increase productivity of their employees as well. Measuring could certainly lead to understanding where developers get stuck or what stage before production takes the longest time, so that managers know which areas they need to address. However, there are some aspects of working beyond coding that need to be taken into account as well.

Quantifying happiness

It's quite obvious that this aspect is important by now. Most tech firms try to have as many benefits as possible, nice-looking offices, colorful and stylish restaurants and the list goes on. This is all an attempt to increase the happiness of the employees. And they seem to be right as well. There are multiple studies out there that show that there's a correlation between happiness and productivity. In fact, compared to people who are unhappy, it has been found that people who are happy have 37% higher work productivity and 300% higher creativity. [18]

Conclusion

When I first heard about measuring software engineer work, I thought: "How hard can it be?" After looking more into all the possibilities and metrics available, as well as the ethical issues that rise up when collecting data, it's clear that the answer is: "Very challenging, in fact". There's a lot of factors to consider, and there isn't a general solution that can fit every company. Some people even argue that it's impossible to measure productivity in this field. I tend to disagree, as there are a lot of quantitative metrics out there, so it's just a matter of choosing a combination of the ones that have the most impact on the kind of work required. One thing is clear, technology is all around us now and it's continually growing, thus all the studies into metrics and performance are necessary for the industry.

[17] http://www.citeulike.org/group/3370/article/12458067

[18] S. Achor, "Positive Intelligence," Harvard Business Review (Jan./Feb. 2012)

References

Elliott, E. (n.d.). *Assessing Employee Performance*. Retrieved from Medium.com: https://medium.com/javascript-scene/assessing-employee-performance-1a8bdee45c1a

Humphrey, W. (1995). *A Discipline for Software Engineering, Addison-Wesley.*

vartec. (n.d.). *StackExchange*. Retrieved from https://softwareengineering.stackexchange.com/questions/62817/how-should-developer-performance-be-measured/62833#62833