Software Engineering Measurement Report

Ryan
Barron
16329561

Introduction:

To continue on Fred Brooks paper, “No silver bullet”[1], I want to remind that software development is hard and complicated. In the paper, Brooks talks about what he called the essential difficulties of software development and its accidental difficulties. The latter are “artificial barriers” that can be completely removed given the right technologies, and in the past, burdens like low-level languages or making different program work together, were removed by higher level language and unified development environment. The former, the “essential difficulties”, they can not be removed, only minimized to some level, and it is those difficulties that Brooks was interested in. Indeed, nowadays the majority of development time is spent tackling those problems, so to improve software productivity we need to fix these essentials difficulties. One of Brooks solution to reduce the essential difficulties, is to “Identify and develop the great conceptual designers of the rising generation”, basically find good software engineers and use them to great effect. Again to quote his paper, “The differences between the great and the average [designer] approach an order of magnitude.”, i.e a great software engineer is 10x more productive than an average one. Which is a figure that has been talked a lot about with a lot of controversy for certain people, that claim would reach the status of “The mythical 10x programmer”. I believe in such mythical beast but as we will see later, identifying a 10x programmer isn’t easy.

Anyhow, it is for these two main reasons (software productivity can only be improved by reducing the essential difficulties and that a good programmer can yield an order of magnitude improvement) that much of the software world has pushed more and more metrics and measurement techniques to assess its main actors, the software engineers. The goal being, that with real data, we can make prediction about software projects to avoid problems with deadline and cost, and to improve the process to be faster and easier on the engineers. Finally, one of the other main goal is to achieve bug-free software, or at least get closer to it, with data, we can measure the quality of a program and can estimate how many bugs there should be and compare that to bugs found. Measuring software engineers is not something new, but new models and new metrics have seen a rise in the recent decades. And it is when combining all these methods that management teams can make real prediction and estimation on software projects and find out which developers are the most productive.

In the first part we will discuss the different metrics that have been used to assess software engineers, talk about their utilities and their limitations, and see how difficult it is to identify a good programmer. Secondly, we will the some of the modern tools that are used nowadays to gather these metrics and look in depth at its algorithmic and how meaning is extracted from the raw data, we will talk about particular platform and applications such as GitPrime or Hackystat that help management team. Finally, some discussion about the ethics surrounding these type of work will end the report. We will see how the way the data is collected can breach on people privacy and how a shift in the software industry such as constant monitoring of software engineer, can leave some of them out because of an old working habit that does not match well with what management is looking at.

I Measurements in software engineering

Over the years, many metrics have seen the day and there are too many to count[2]. But they are used to measure different aspect of software engineering or can be used and combined to measure different things. In the end, there are 3 main aspect that are measured. As Ecomputernotes[3] puts it (and other sources)[4], we can measure the process, the product and the project management. I will talk about each of those, what their goal are and what specific metric they use to achieve this. But we need to remember, that each individual metric, does not give us any meaningful information. It is by combining different type of metric that we can make real prediction and really assess what kind of process yields what benefits.

- Product measurement: The target here is the end product of a development phase. The software get analyzed in multiple ways. The goal is to check if the product meets the user requirements, the complexity of the software at a given stage of development, the load time and execution time, the size of the program, the quality of the program and how many defects are present, the code coverage of the program, how much documentation has been produced, how maintainable the code is etc.

LOC (Line of Code): Probably the oldest metric used in software engineering. It is used to measure many different things when combined with other measures, in this case it allows to measure the software complexity to some extent using the order of magnitude, the maintainability of it and the program size for example. It seems pretty self explanatory on how to measure it, just open a text editor and look how many lines of source code there, do that for each component, compute the sum and you’re done. Even such a simple metric has different definitions. But according to Wikipedia[5] there are two distinctions to be made, there are physical LOC and logical LOC, the first one are the lines that would be displayed in a text editor, that would count white spaces. The second count computable statements. Two examples of C code to compare the different measure:

```
1)for (i = 0; i < 100; i++) printf("hello"); /* How many lines of code is this?
*/

2)/* Now how many lines of code is this? */
for (i = 0; i < 100; i++)
{
    printf("hello");
}
```

The main difference here are the curly braces, but the point to take away is that logical LOC are less sensitive to formatting and editing conventions, while measuring more accurately the mental effort that goes into writing code.

Defects per KLOC (Kilo LOC): A metric use to measure software quality, it is derived from the LOC measure. The number of defects or bugs can be measured in different ways, but the most practical one is to define a test plan, right test unit, run them and count the number of failures or defects and compare that to the LOC of the tested modules.

Cyclomatic complexity: This is, as its name indicates, a measure of software complexity. Invented by McCabe in 1976. Informally, it is a measure of the number of IF statements in a program when translated in machine language. More formally it is “the number of linearly independent paths through a program’s source code.”[6] It is calculated using a control flow graph of the program. McCabe’s cyclomatic number has been used for software testing, it is thought that the number determines the number of test suits needed to be written. Also studies have shown that a complexity number equal or higher to 16 means the program needs a rewrite[6][7].

Halstead complexity measure: This approach of measuring software tries to identify the properties of a program just like you could measure the properties of a gas. It is language dependent as it measures the number of unique operators and operands found in a program and the total number of operators and operands in that same program. From this number, Halstead provides a number of equation from which new information can be gathered. Such as the difficulty to write and understand that code, the effort required to code, which can derive the actual time needed to program. Even the number of bugs expected on delivery.

Cohesion & Coupling: These two metric represent the opposite sides of a same coin. One is the inverse of the other. Indeed, high cohesion, which tend to display traits such as robustness, reliability and re-usability, is desirable just like low coupling. Basically it is the measure of how inter-dependent different pieces of code are. How much one module reuses code from another modules and how much data they share. A low cohesion / high coupling program will feature modules that share the same global variables and public function that are used from one module to another. Whereas in a high cohesion / low coupling, program, we would see different distinct data structures for different modules and only parameters would be shared across modules.

Function point: This is a measure of the number of functionalities requested by the user. That is, the functional size of the application is measured from the user point of view, of what he will interact with. It is considered a hard analysis to do, despite that function point analysis is a popular method and even has its own ISO standard.

All this measurement, and others such as program execution time or test coverage, are needed to predict and estimate the cost, time and man effort required to develop an application. On their own, they tell us a lot about the quality of an application at a given state.

- Process measurement: Here we measure the programmer themselves and the way they program. The goal is to measure the different aspect of their development process and compare them between one another and with product metric to find out which process is the most productive or output the best quality software etc. Some of the measures here are harder to measure than others, as some can only be measured in the physical world, it is harder to automate it. But some can still be easily measured and so, these are the kind of metrics that are recurring across the industry and more known.

LOC per programmer per month/week/day: Again a measure based on the LOC but this time, we don’t measure all the code in the source files. We measure the output of a specific programmer within a time period, to see how productive he/she is. This is done with a repository system such as Git or SVN. We can easily track each users commit and within these, how many LOC were added or changed. So the metric here is a bit different, we don’t measure the result but the effort.

Code Churn: This is build up of the last one. Code churn looks at the code that was push to the repository and looks at how much is brand new and how much of the LOC pushed are just modification made to old code. It is the difference between the total LOC committed and the final output, the final LOC count like seen in the previous part.

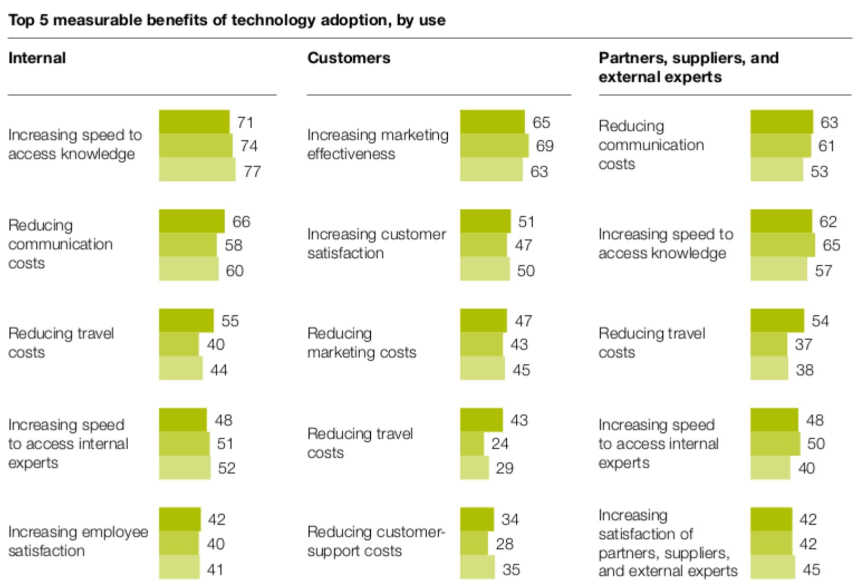
Code commit: This looks at the number of commit made by a developer. Indicator of the regularity and it is important that developer commit often and keep in touch with the codebase and the project as a whole.

Throughput: Measure of total amount of work done by a team or single programmer in a given time. This can be a measure of LOC, or a measure of bug fixes, functionalities added.

Cycle time: This is a build up on throughput, but looks at the average time needed to complete a task. A task being a bug fix, or adding a functionality, but more broadly any issue that is being brought up during the development process. To measure cycle time, the team needs a way to issue ticket that create those issues and tasks that need to be completed. Cycle time is just the measure of time from when the ticket was created to when it was closed.

Hours worked: Measure of how much time a developer is at his office working. This is more important than at first glance. More hour don't necessarily lead to more throughput. Indeed more hours can make developer less productive but it can make them produce less than if they had less hours. A week of 40hours can lead to lower throughput than a week of 30hours.

Team communication: Studies have found that peer communication leads to increased employee productivity and enhanced organizational processes[8] and certain benefits such as reduced communication costs, reduced traveling costs or increased access to knowledge[8][9] (Figure 1).
This is can be measured in a number of different ways. If a social technologies is used then that process that can be number driven and automated. Slack conversation can be analyzed, the number of emails sent for examples. But if we want to measure programmer going from their desks to one of their coworker, this is harder to measure and often, survey are used in companies to measure that kind of communication.



(Figure 1. Extract from " McKinsey Global Survey results" showing the benefits of social technologies)

This list is not perfect, and in actuality, the kind of metric measured vary a lot and not of whole lot of them are ubiquitous. Indeed, depending on the methodology, different metrics can be measured. For example, in agile development, notions like velocity, build, or test are very important and will be measured. Anyhow, the ones listed are already enough for basic assembly of data points. We can get some basic idea of the time needed for a developer to finish a feature, by comparing his LOC/week with the total LOC of that feature, which can be determined by a function point analysis. Is this a valid way of making conclusions? We will talk about that in section 2.

- Project measurement: These are metrics only useful to the management team and look at the development process as a whole. All the measurements done here are very generic and not specific to software development, I will just go over them quickly as they are well known metric.
Some of the main metric used to assess a project are: Time (how long it took to complete project and if team is on time compared to estimation), Cost (the monetary cost for the project), Effort (a measure of the human effort, how many people worked on the project), ROI (return on investment or the ratio between money produced and money expended by and for the project), Total amount of rework (how many bug tickets were sent out and many were fixed) …

A lot of these metrics are used for project prediction or just for comparison. It is when combining all of these measurements that useful information can come out such as which software development model is best for a given project? (Waterfall vs Agile)
How long will it take and how much money will it cost, to complete this project given the functionalities required and the team size?
To answer all these questions, you can use all the metrics above. Each of them help indicate answer to these questions.

Some people claim that you can not measure a software engineer productivity[10] and that his work is more akin to that of a doctor or lawyer. While I do agree that software developing is similar to the work of a doctor and lawyer, in that it is nuanced and the task that need to be done are not explicitly specified, that is, there is a design aspect, almost artistic, to software development. I still think that some useful information can be taken from these metrics. It has been clearly shown that more peer interaction and communication leads to better results, and probably more productivity in general.

Nonetheless it is hard to quantify a programmer productivity and maybe it is not the best measurement goal. Indeed as John D. Cook talks about in his articles[11], it is hard to identify a 10x programmer because that programmer is not necessarily more productive. It's been shown that a good and bad programmer are about as productive, and generate about the same LOC a day, although the bad programmer will create more churn. But most important a 10x programmer is really valuable because he comes with better ideas on how to implement large applications. He knows what the easiest path to solve the problem. A 10x programmer is not some kind of super fast genius who types 200 words per minute and delivers bug-free software on his first attempt without any need to rewrite code. No, a 10x programmer is more so someone who takes the time to think, talk to his peers and remember something he might have learned in the past on how to deal with the problem. And that is where the problem is, a 10x programmer is hard to measure, even if you could measure his productivity, you would not find any sign of the help he provided. It is only measurable when looking a the whole project, at which point it can hard to track back down to the individual who saved 3 months of development with a single idea. Indeed if you just measure your software engineers you can end up with wrong prediction and wrong results, your metrics and what you did with them become a negative. It is not our current measure of productivity that will help us find a 10x programmer, and I do not know if better process and higher productivity can yield an order of magnitude improvement on development time the same way as a 10x programmer can.

There is another problem, according to Fenton and Neil[7] a lot of industrial software measurement is poorly executed and only uses a few product metric from the 1970s in poor manners. They claim that companies who just want to measure productivity to find out which development method is better or wants to make accurate prediction don't get any benefits of using software metrics and can even have worst outcome because of misuse. For example, they talk about how certain companies don't make a distinction between defect encountered during development and during operation. The LOC is still routinely used and it is nowadays a poor measurement especially when comparing between different programming languages.

It is for these reasons that some people are skeptical of metric being used in software engineering, they claim that an engineer's work can not be measured and it creates more harm than good. While I do agree that measuring the whole contribution of a software engineer to a large project is hard, and almost impossible, I do think that more brute productivity can be measured and metrics can lead to positive results and faster / better development. Although we need to be careful of what the metrics we collect and what we claim they measure. There is this notion of "construct validity"[12] which basically asks "is what we are measuring really a measure of what we want to measure? ", that is discussed by Kaner and Bond. What they've found out, after formally defining construct validity, is that, again, a lot of metrics are used in poor fashion, the data they have does not tell the whole story, and yet companies use these to recreate a story of the development process and assess their engineers. Before ending this, the last problem with measurement plan are often started because of some external pressure, and often cost about 4-8% of total budget and are an overhead on the project[7]. During crunch periods, measurement plan are often the first thing to be stop completely.

Well, there are quite a few negatives about metric in software engineering. It is a heated debate and everyone has its own opinion on it even to this day. Nevertheless solutions have been found to some of these problems and should convince people that metrics can be useful. Namely, the problem of wrongly using data can be fixed using Bayesian Beliefs Nets, and platform such as GitPrime or Hackystat remove a lot of the overhead and automates a lot of the data collection. We will discuss in the next part, about some of the algorithmic approaches that prevent wrong use of the data as well as the platforms and applications that automate a lot of the measurement work and allow management team to make useful decisions.

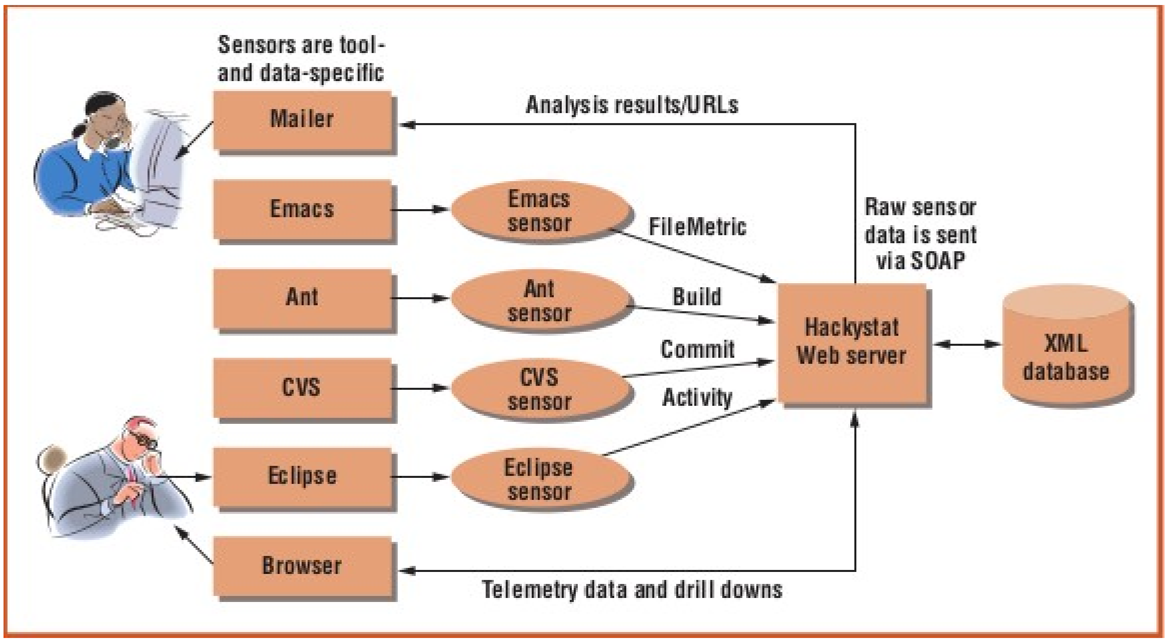
II Algorithms and applications: Methods and tools to help measurement plans

In the last two decades, work has been done to make software measurement more practical and more effective. Methodologies were developed, applications were made. Here I want to discuss some of the solutions that have been established, talk about their strengths and shortcomings and some of the tools used in industry today.

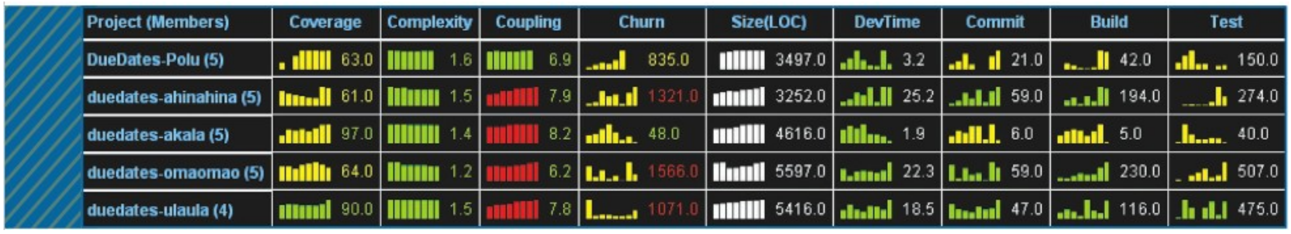
I want to first talk about the PSP, or personal software process, which is a methodology to measure the software process of an individual developer, and help them become more productive. The PSP was created in 1996, and a lot of other tools took inspiration from it, that is why I will first talk about it.

First of all, PSP is just a methodology on how to develop software. It promotes developers to take notes about their overall process and the PSP is there to give them a plan about what to measure and how to make prediction and estimate of their productivity. As Watts Humphrey, the creator of the PSP, puts it, improving process is a better way to assure quality than testing. The PSP starts by asking developers to measure their activity and record the data, then they are ask to make estimate and make goals they can keep up with. Finally the last goal of the PSP is to review their design and code quality so that engineer can improve on those. The PSP is a manual process, developers are asked to fill in spreadsheets during development. This is one the downsides of the PSP, it interrupts development[13]. On top of that, the human oriented approach can lead to incorrect process conclusions. The PSP has a large overhead cost but nonetheless it one of the best tools for analytic and has been shown to provide high-impact analysis. And one of the most important aspect of PSP compared to some of the other tools that I will discuss later, is that PSP is flexible. If a developer has the impression that he gets interrupted too often during work and that has negative effect on his process, than the PSP encourages to write this down and keep track of this data. Finally, Humphrey always thought of the PSP as a manual task, and doesn't believe he can be automated. Indeed Leap is an application that was supposed to solve the problems of the PSP, but created new problems and overall ended up being a failed attempt. Leap introduced automatic analysis of the data, to prevent measurement dysfunction and even provide extra analysis compared to the PSP, but that automation made the overhead cost even bigger. Some type of data became really hard to integrate into the process, for example the number of interruption a developer has. This brings us to Hackystat, the next evolution in term of PSP analytic and improvement on what Leap was supposed to accomplish.

Hackystat is a piece of software supposed to automate some part of the PSP. It was a research project by Philip M. Johnson and his team at the University of Hawaii. Its main goal was to reduce the overhead of the PSP and automate the collection of data. Hackystat is built around services that it delivers, most of which are built inside the applications, but other tools come from Twitter or Google. Some of the services are used to collect data about the engineers, others are used to assemble that data into larger, more coherent pieces of data and a few others are here to take those large pieces of data and bring a high-level analysis(Figure 3.) that can be used my management teams. This is done via client side tools and server side tools. Indeed Hackystat monitor an individual engineer software process via tools that are implemented into his IDE, or into the repository system (Figure 2).



(Figure 2.[14] Showing the functioning on Hackystat and how the different sensor track the users and send the data to a central server for analysis.)

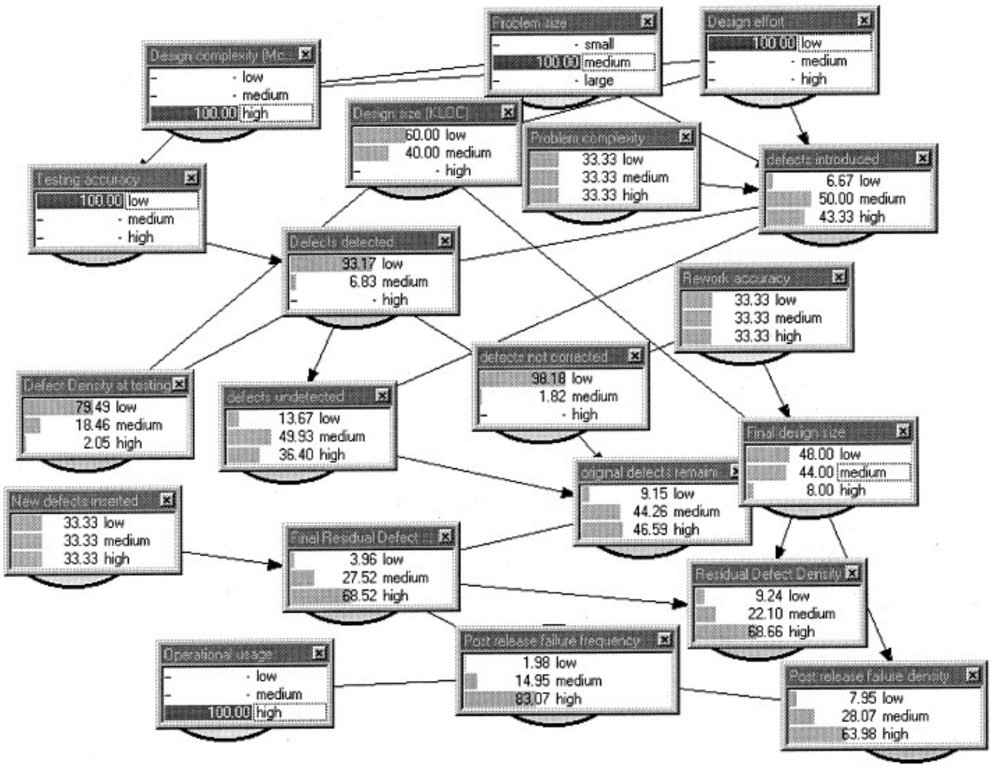


Project (Members)	Coverage	Complexity	Coupling	Churn	Size(LOC)	DevTime	Commit	Build	Test
DueDates-Polu (5)	63.0	1.6	6.9	835.0	3497.0	3.2	21.0	42.0	150.0
duedates-ahinahina (5)	61.0	1.5	7.9	1321.0	3252.0	25.2	59.0	194.0	274.0
duedates-akala (5)	97.0	1.4	8.2	48.0	4616.0	1.9	6.0	5.0	40.0
duedates-omaomao (5)	64.0	1.2	6.2	1566.0	5597.0	22.3	59.0	230.0	507.0
duedates-ulaula (4)	90.0	1.5	7.8	1071.0	5416.0	18.5	47.0	116.0	475.0

(Figure 3. Example of Hackystat's intermediate process, where it assembles data and trends into more meaningful information.)

However Hackystat is not the PSP and it has problems of its own, although these are mainly ethical and I will talk about those in part 3. Indeed Hackystat automatic nature makes it hard to collect new data that might have an impact on a developer. Again, if a developer feels interrupted often by someone knocking on his doors, to make the data collection automatic, hardware and software would need to be developed to add that information into Hackystat.[13]

While the PSP and Hackystat are more geared towards measuring the behaviour of developers, BBN (Bayesian Belief Nets) are more about making useful analysis of product metrics. Indeed the challenge of behavior metric is to collect that data automatically, but the data is very useful and can easily be assembled into higher level data without measurement dysfunction. What Fenton and Neil proposes with BBN, is a way to use product metric appropriately. They claim that one of the major reason for wrong use of data, is a too simplistic approach to analysis that does not understand the notion of cause and effect. Correlation between size and defect rate is translated into a cause and effect relation although they point that no studies have proved that relation. Their solution, the BBN, is a graphical network of variables with associated probability table that tell us the chance of it being in a certain state. Each connection between two variable represents the causal relationships between two nodes, the parent being the cause. The original probability tables are set up with empirical data and subjective judgments and then they get updated based on the values of their parent nodes. BBNs allow to models unknown variables that we are trying to predict and display the uncertainty surrounding them. And each time evidence for the state of a variable is entered, the change is spread through the whole network and new probabilities are calculated for the rest of the variables which depend on the new one.



(Figure 4. Example of a BBN, each variable has three possible state: low, medium or high. And each of those has an associated probability. Dark gray bars are user entered probabilities, the others are estimation done by the program based on the relationships.)

So, the main advantages from BBNs are that they make the relationships between variables explicit and so it makes it visible for everyone in the company to audit it, and it’s statistical model based on Bayesian theory combined with notions of uncertainty makes the system really robust for estimations.

Before ending this section, I want to talk about GitPrime. GitPrime is a recent example of a measurement tool to help management teams. The service was created in 2015 and is used by many companies today. For the most part, it is similar to Hackystat, as it is based on product and process metric automatically collected from any kind of git-based repository system. My point is that, while tools like Hackystat are old, and only used in academia, its ideas are very common place nowadays and it important to understand them, what they can and can not tell us. Measurement dysfunction can still happen and as I will talk about in the next part, there are a few important ethical questions to ask. Like what if someone is wrongly considered to be a “Net Negative Production Programmer” or NNPP, and gets dismissed? Maybe if he was not a NNPP, and by finding a replacement, it actually hinders the team. This scenario can happen, and so it is up to management team to ask themselves if this is really worth it. Do they see productivity improvement since they’ve introduced a measurement plan? Do they see a drop in employee happiness since the introduction of the measurement plan? What about the automatic telemetry of employees ? Did you ask for their consent? And what if they disagree?

III Ethics

These kind of ethical questions are nothing new. They mostly revolve around privacy in one case, and the other being the usage and misuse of new technologies. The former is very common with information technology, I mean the name says it all, it’s about information and people’s private information. Indeed, in general, the more private an information is, the more valuable it is. So it is almost that these telemetry technologies and IT in general wants to spy on people to find the most valuable data. Johnson talks about this problem in his article about Hackystat[13], about how developers did not want automatic telemetry tools to be installed without their agreement, and to make data about their process be public within the work place. Johnson talks about a solution in the form of a “cloud-base, independent, privacy-oriented analytic repository” where developers can own their data and can choose to give access to it to the management team. I don’t know if management teams have the power to force employees to reveal that kind of data or not, and all the legal issues surrounding data. If they can do that, well there is still a problem where employees must made it public or else they get dismissed. If not, the solution works well and gives power to those holding the data.

Also we need to ask the problem of management teams misusing the data whether it is on purpose or not. What if a 10x programmer is fired because he spends much less time on his computer? We have seen that this kind of programmer are productive because of the new ideas he brings. Again if that is the case, there is a need to check if measurement plans are worth it or not. And what if a manager want to dismiss a developer for personal reasons? Can he just fire him by justifying his decision with some metric that is unrelated to whether or not he is doing a good job? There will always be misuse of new technologies and new techniques. The goal is to improve on those to prevent all the negatives, rather than throwing them away, saying they are not good. I think an easy fix to measure employees happiness on the side of the overall productivity increase. The goal being, to weigh out the pros and cons.

Finally, measurement plans are relatively new, and I believe it could create a difference between older and newer generations of developers. Older ones would be against measurements plan as they did not evolve with them and think poorly of them. Making it harder for them to find new companies to work with if they have to switch. For newer generations, they will have to deal with all sorts of different measurement plans and won’t have a say in it. And that can create some distance between these two groups as they will be jealous of the other group privileges.

Conclusion:

Finally I want to conclude this report on the fact, properly measuring a team of engineers is hard, it has its pros and cons and it is still early to say whether or not it is worth it. Lots of paper talks about how we should measure engineers, and what method / practice is best at measurement and we can extract meaningful information to improve the process. My hope is that more paper will come, looking at how much improvement can be made that way and look at the negative that it brings, which one we can solve and if it is worth it. I already proposed a temporarily solution to this by measuring engineers happiness. I still think we should measure engineers engineers at least to see if we can get an order of magnitude improvement by improving team productivity as Brooks would hope.

Sources & References:
[1] Frederick P. Brooks, Jr., “No Silver Bullet —Essence and Accident in Software Engineering” in Computer Journal Volume 20 Issue 4, April 1987 Pages 10-19.
[2]https://en.wikipedia.org/wiki/Software_metric
[3] http://ecomputernotes.com/software-engineering/classification-of-software-metrics
[4] https://fr.slideshare.net/swatisinghal/software-metrics-5079475
[5] https://en.wikipedia.org/wiki/Source_lines_of_code
[6] https://en.wikipedia.org/wiki/Cyclomatic_complexity
[7] Norman E. Fenton * , Martin Neil, “Software metrics: successes, failures and new directions” in The Journal of Systems and Software 47 (1999) 149±157.
[8] Jacques Bughin and Michael Chui, “Evolution of the networked enterprise: McKinsey Global Survey results”, http://www.nextlearning.nl/wp-content/uploads/sites/11/2015/02/McKinsey-on-Impact-social-technologies.pdf
[9] Emerson Murphy-Hill and Gail C. Murphy, “Peer Interaction Effectively, yet Infrequently, Enables Programmers to Discover New Tools”, in Proceedings of the ACM 2011 Conference on Computer supported cooperative work, CSCW ’11. ACM, 2011, pp. 405–414.
[10] https://dev9.com/blog-posts/2015/1/the-myth-of-developer-productivity
[11] https://www.johndcook.com/blog/
[12] Cem Kaner and Walter P. Bond, “Software Engineering Metrics: What Do They Measure and How Do We Know?” in metrics 2004, IEEE CS.
[13] Philip M. Johnson, “Searching under the streetlight for useful software analytics”, IEEE software July 2013, Pages 57-63.
[14] Philip M. Johnson and al., “Improving Software Development Management through Software Project Telemetry”,