ENGINEERING MEASUREMENT REPORT

BY MARTA LOBO DE PABLOS

Software Engineering is, without any doubt, one of the industries that have changed and evolved the most rapidly. Everyday new ideas and approaches appear, as technology is becoming such an important factor in our routines.

However, the software industry has a greater impact because it reaches to every single other industry, changing their paradigms. The way accounting is done today differs noticeably from those years when the *journal* and the *ledger* were physical books; most of the budget of every Hollywood big production goes to pay for the Special Effects.

Nowadays it is becoming harder and harder to think about a problem that does not already have a software solution – regardless of those that the software itself brings -, and this wide range of tools is helping change the way we do everything.

One of the areas where software helps the most is, indeed, developing new software. As one professor once said: "Software engineers are the only crafters creating their own tools". From IDEs or Test Suites, to Pipeline Integration Systems or SDKs, there are many software products that engineers use for getting the work done in the most efficient way.

This report is meant to analyze and set a debate start point in one specific segment of this engineering tools: those that are meant to measure working performance. Because, if you could develop a tool that would help develop more tools in a more efficient way, by spotting out the flaws and strong points in your development process / team, why wouldn't you?

Maybe there is a reason for it. But before the argument, an exposition of the facts: what is the data that can be measured and is related to the performance, what are the computational platforms currently available to do so, and which algorithmic approaches they could be using.

MEASURABLE DATA

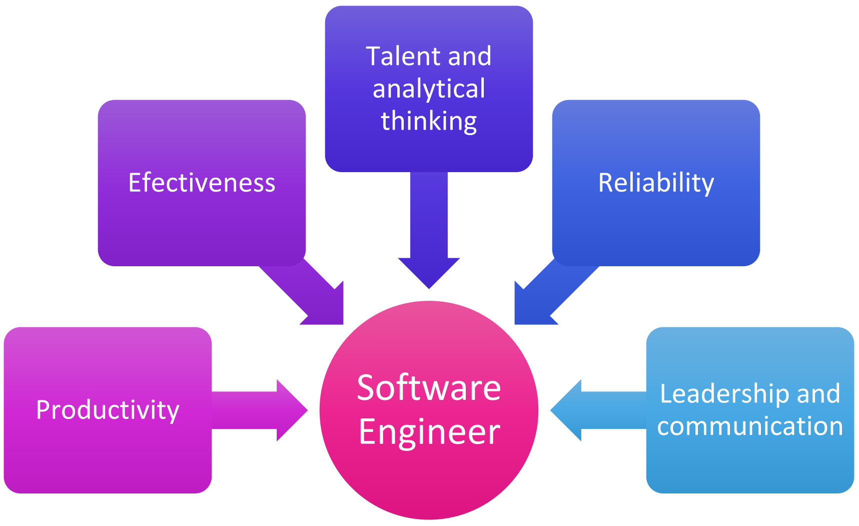
To measure performance and make decisions about it, there are two main issues to be discussed: what data should be gathered, and how can it be processed to get useful information out of it. Both dimensions can give a general perspective of a good or bad job.

As a piece of software in a competitive environment is hardly never developed by a single employee, it is harder to measure individual performance, although new tools are appearing to help with this.

Firstly, a further detailed analysis should be done, about these different *process* or *product* data types, to distinguish what is useful for making conclusions about a job well done.

The data that can be retrieved and is useful for analysis should be that which is related to all different dimensions of the software engineering job:

- Productivity: how to do more in less time and with less (waste of) resources. Procrastination can be measured in terms of the percentage of time a developer is idle within a given period.
- Effectiveness: how to do what is asked or needed to be done. Success does not always relate to quality, but when it comes to bug fixing or test coverage it is a good indicative of how valuable an engineer in is doing a good job.
- Talent and analytical thinking: how to do best, in terms of technical challenges or creativity: given a particular problem, assess how good a solution is in terms of complexity, lines of code, dependencies, etc.
- Reliability: how to do something trustworthy, such as meeting deadlines and requirements, without breaking something else.
- Leadership and communication skills: how to do a good job collaborating with more people. As software development may seem a lonely job – one person + one computer -, the value added to the team (pair programming, mentoring, managing…) is suitable for being measured as well, although it can be more subjective.



Once all these competences are defined, it is possible to look at some types of *available* data and its relevance for measuring and evaluate them:

LOC:
Counting how many lines of code a developer has written may seem pretty straightforward, but it is exactly the opposite. Too many lines can be an indicative of poor performance, with lots of duplicates and useless syntax that, as programming languages evolve, they are getting rid of (callbacks, lambda functions…); however, too few is not synonym of either laziness or perfection for writing compact and fully functional code. More factors need to be considered.

CODE COVERAGE:
Good code needs to be tested, assuring its robustness when new requirements need to be met or succeeding code changes happen – even in bug fixing. The level of test coverage of a piece of code can indicate how much the developer cares about doing a good job.

COUPLING:
How classes and files in a project are related and depend one from the others. A higher level of coupling reflects a poorer architecture where little changes may affect several dependences, and therefore be susceptible for more bugs. The dependencies can be easily measured by the number of *imports*, for example.

DEVELOPMENT TIME:
With the rise of AGILE development techniques, it is getting easier to measure the time smaller chunks of code take to be finished. This may depend of all the previous factors described but, as a whole, can give a good perspective of a developers' productivity, since the estimations are usually based on the overall performance.

TECHNICAL DEBT:
There is some extra development work that comes from applying patches to bugs and then needing to refactor, to find less simple but more sophisticated and complete solutions. The bigger the percentage of technical debt in a software development project, the lower the quality of the implemented solution. This applies to the *short run vs long run* philosophies. In general terms, a percentage above 10% of technical debt shows a poor codebase that would need to be refactored.

These are just a few aspects that can be taken into account when it comes to try to define *a well-done job*. Once defined, a new problematic arises: how to measure them.

COMPUTATIONAL PLATFORMS

To achieve successful results in measuring engineering, which means to effectively improve the software engineering process, all this data must be gathered, classified and analyzed; this way, it is easier to extract conclusions and strategies out of it. This could be called *Software Intelligence*; the same way Business Intelligence "offers concepts and techniques to improve business decision making by using fact-based support systems, SI *would* offer software practitioners (not just developers) up-to-date and pertinent information to support their daily decision-making processes." [2]

Software Intelligence, as a stablished practice, is still *in diapers*. Most of the software development decision-making processes are usually guided by previous – subjective - experience, most popularly known as "gut feeling". Intuition and experience are the blinking flashlight that enlightens the software engineering job; and when in too much dark, senior developers' advice is also useful. These are useful tools that have turned out fine so far but, having the chance of being more meticulous and sophisticated, it is a road to take.

A new approach towards expanding the Software Intelligence discipline is by going through the Mining Software Repositories (MSR). This is a field which, given available code repositories, analyzes the data to uncover interesting and actionable information. This way, existing software systems and projects can be scrutinized, making use of existing sources:

HISTORICAL REPOSITORIES:

Source control, bugs records, archived communications record information about a project's evolution. Historical dependencies between functions, documentation and configuration files. This prevents to use code dependencies that may not be necessary or break if further changes are made afterwards.

RUN-TIME REPOSITORIES:

Deployment logs that display the usage and execution of a software system. They are useful to find anomalies in pinpoint executions, as they allow to identify dominant execution or usage patterns across deployments.

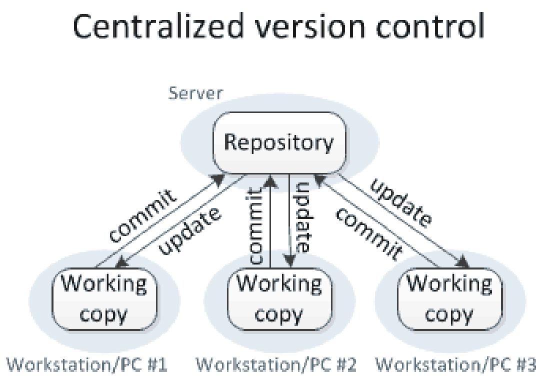
CODE REPOSITORIES:

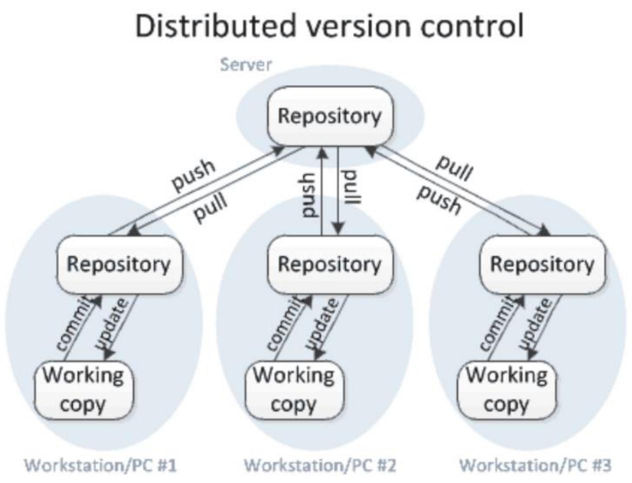
Source code from multiple projects can be used to find out about API usage and popularity, or what are the most representative frameworks for a specific solution. Usage patterns can also be discovered using this source.

Code repositories are more frequently used to keep track of previous bugs and implementation; their use for decision making, although undeniably relevant, is not that popular yet. However, it is gaining that popularity as the amount of available code repositories increase.

Source code gathering is supported by the use of *version control systems*, software pieces - crafters building their own tools, again – that track changes in source code, allowing to go back and forth to any version of the code that is being developed. This type of software appeared around late 1970's with Marc Rochkind's Source Code Control System.

Nowadays the most popular solutions are *distributed* version control systems, as development works are carried by more than one person or even team, and this way the whole codebase is available on every developer's machine. This way, working offline or *branching* are way easier.





Although the first solution of this kind was BitKeeper(2002), the most popular software, currently used worldwide, is Git.

GIT

Git is a multiplatform, low-level, operating-system-independent command-line tool that was developed by Linus Torvalds and his team in Linux and was launched in July 2005. It was conceived to meet new requirements that other tools such as BitKeeper or Monotone were not contemplating, and soon became a standardized solution. Nowadays it is integrated with almost every IDE, and multiple development tools have been born to facilitate its use:

GRAPHICAL USER INTERFACES

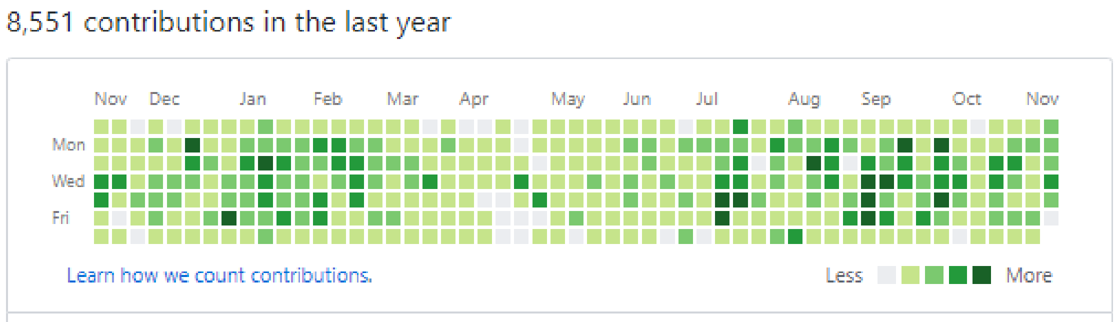
Although Git provides its own GUI (Graphical User Interface), it is rudimental as it does not align with its main goal. There are other solutions, such as SourceTree being the most popular, that offer a visual representation of the repositories, and allow to interact more directly with all Git commands (add, commit, push, pull…)

REPOSITORY PLATFORMS

Github, Gitlab and BitBucket are the most popular repository platforms, where the projects are stored and can be easily accessed. At the same time, they allow developers to communicate, collaborate and share knowledge.

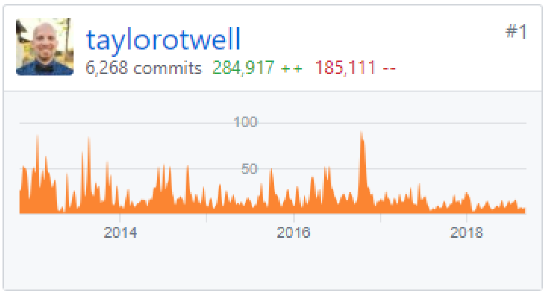
These tools focus their efforts in provide information and statistics about some engineering measurements, to complement the teams' jobs. Taking Github as an example, some of its more popular statistic visualizations are:

- Developer's Punch Card: a graphical perspective of a user's commit activity. Here is the example of Taylor Otwell's (PHP Laravel's creator) contributions in the last year:

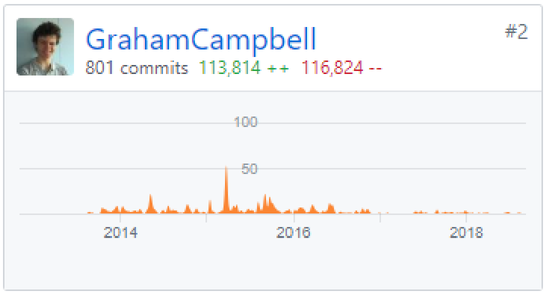


8,551 contributions in the last year

- Repositories insights: some statistics about a particular codebase, including how many people contribute to it and how frequently, dependencies with other repositories or tools, lines added or deleted, etc… Here is an example of the two most active contributors in Laravel's framework repository:



taylorotwell #1
6,268 commits 284,917 ++ 185,111 --



GrahamCampbell #2
801 commits 113,814 ++ 116,824 --

CAST

There are also software measurement solutions for businesses or companies that want to keep this data more private (less accessible to the big public). One of the most recent business solution, that exemplifies the close relation between BI and SI, is the one provided by Cast Software, market leader in this field. It is a company founded in Paris in 1990 but didn't include software quality measurement into its analysis products until 2001.

Its main product is CAST HIGHLIGHT, launched in 2015. It is a SaaS (software as a service) platform that allows to track several software metrics using predictive pattern analysis.

ALGORITHMIC APPROACHES

All these sources of information are not entirely useful until something is done with the data they contain. Going back to the example of the types of repositories (historical, run-time or code repositories), an overview of how these repositories can be *mined* to extract information is given in the following chart:

Software Engineering Data	Mining Algorithms	Software Engineering Tasks
Sequences: execution/static traces, co-changes, etc.	association rule mining, frequent itemset/subseq/ partial-order mining, seq matching/clustering/classifi-cation, etc.	programming, maintenance, bug detection, debugging, etc.
Graphs: dynamic/static call graphs, program dependence graphs, etc.	frequent subgraph mining, graph matching/clustering/ classification, etc.	bug detection, debugging, etc.
Text: bug reports, emails, code comments, documentations, etc.	text matching/clustering/ classification, etc.	maintenance, bug detection, debugging, etc.

The first column describes the data that can be analyzed, and the third one gathers the applications that analysis can have.

The second column contains different algorithmic approaches to manipulate this data, all related to the Data Mining field, which plays a fundamental role in this task. As new data appears and becomes available, new algorithmic needs appear.

The interest of processing data and extract useful information out of it started to blunt in the 1980's, although in the 1960's there were already some references to this activity; *data fishing or data dredging* were terms used pejoratively to describe analyzing data without an a-priori hypothesis.

The term *data mining* appeared around 1990 linked to the use of databases, along with others such as *data archaeology, information harvesting, information discovery, knowledge extraction.* Different ways of exemplifying what software developers want to achieve with measuring their performance: discover useful information that helps them work more effectively and efficiently.

One of the most popular techniques of data science is ARM: association rule mining. It gathers multiple algorithmic approaches that can be very helpful in the engineering measurement duty.

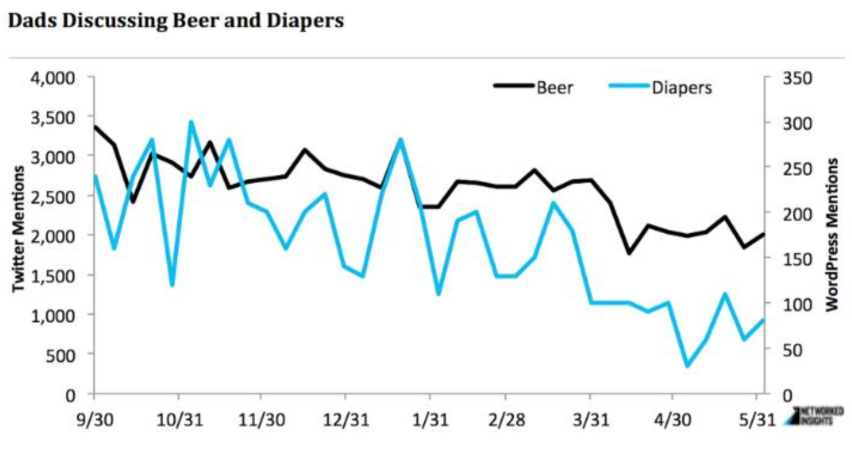
Association rule mining

This methodology is used for discovering interesting relations between variables in large databases. One of its most popular applications is for stage-effort estimation, a fundamental part of the Project Manager's job that allows her to re-allocate correct number of resources and control project progress to finish on time and within budget.

The Association Rule Mining approach, in this case, is combined with concepts of Fuzzy Logics to utilize prior effort records to predict stage effort.

Fuzzy Logic is a concept related to Artificial Intelligence, that tries to mathematically model imprecise, uncertain concepts, that are normally referred to with linguistic terms. This way it is possible to extract rules and instructions from vague states: something being *more* or *less* easy, with no need of formally quantify *how much* easy it is.

Association Rule Mining is one of the most popular techniques in data mining, as it aims to discover the associations and frequent patterns amongst items in – sometimes vast – databases. It is already used very ofter in many Business Intelligence processes: one of the most illustrating examples is the Supermarket offer. In Market Basket Analysis, the ARM is way more extended as it exemplifies how unexpected association rules might be found from everyday data: the following chart shows how "on Friday afternoons, young American males who buy diapers also have a predisposition to buy beer."



No love lost.

The problem of ARM applied to Software Engineering data is that ARM algorithms are meant to deal with nominal data (categories), and engineering measurements are numerical. This is where the Fuzzy Logic helps combine both disciplines: the numeric values are distributed in intervals, and then each interval is represented with a fuzzy tag (nominal data).

There are multiple algorithms to process the data and extract conclusions out of it. The chosen approach varies depending on the availability of the data and how it correlates within; and when no current solutions are suitable for the problem facing, new algorithms appear. To enumerate some of them: AIS, SETM, Apriori (and derivates), frequent-pattern growth. Further explanation can be found in *An Overview of Association Rule Mining Algorithms*, by Trupti A. Kumbhare and Prof. Santosh V. Chobe.

ETHICS OF THIS ANALYSIS

As the problematic of how to measure software engineering processes may seem to be solved, the next question should be if the idea itself is *ethical*. In the end, what all this solutions and algorithms gather is personal data, as there are people behind.

When developing software itself, there is no thing such as a Computer Science Hippocratic Oath. In many countries the profession is not regulated properly, the same way other guilds (doctors, architects, or even other engineers) are. They require specific qualifications and certifications to practice, because the impact they have on people is tremendous.

Software Engineering – as other branches of Computer Engineering – is not regulated yet, maybe because of how fast it has become such an impactful activity. Technology is overtaking society development, and the longer it takes to catch up, the harder it will become to align regulations with the social situation.

Some computer associations, such as the ACM and the IEEE, have an Ethical Code with guidelines for developers to pursue. This is a first step towards proper legislation, but they currently do not have legal implications. Having a look at the ACM Code of Ethics, there is a distinction between General Ethical Principles, which form the bases for all the rest, and Professional Responsibilities, more specific; however, all of them are meant to be followed by all computing professionals. The following chart presents them:

1. GENERAL ETHICAL PRINCIPLES	2. PROFESSIONAL RESPONSIBILITIES
1.1 Contribute to society and to human well-being, acknowledging that all people are stakeholders in computing.	2.1 Strive to achieve high quality in both the processes and products of professional work.
1.2 Avoid harm	2.2 Maintain high standards of professional competence, conduct, and ethical practice.
1.3 Be honest and trustworthy	2.3 Know and respect existing rules pertaining to professional work.
1.4 Be fair and take action not to discriminate	2.4 Accept and provide appropriate professional review.
1.5 Respect the work required to produce new ideas, inventions, creative works, and computing artifacts.	2.5 Give comprehensive and thorough evaluations of computer systems and their impacts, including analysis of possible risks.
1.6 Respect privacy.	2.6 Perform work only in areas of competence.
1.7 Honor confidentiality.	2.7 Foster public awareness and understanding of computing, related technologies, and their consequences.
	2.8 Access computing and communication resources only when authorized or when compelled by the public good.
	2.9 Design and implement systems that are robustly and usably secure.

It is hard to determine if *measuring engineering processes* is ethical or not, as there are several principles involved. On one hand, it is undeniable that it is a way to achieve high quality in both the processes and products of professional work (2.1). The main goal is to improve performance, maintaining high standards (2.2).

However, the data gathered in the measuring may violate developers' privacy (1.6) and confidentially (1.7). Anyone is comfortable knowing their work is being monitored all the time, and that also has repercussions in the quality of the work. There are studies proving that happy software developers solve problems better and present better performance. Therefore, here is an ethical reflection to be made: what is more important to achieve? As soon as the privacy and confidentiality are respected and preserved, the conversation is over; it is worth trying to find data measurements that are equally useful and respectful.

REFERENCES

L. Singer and K. Schneider, "It was a bit of a race: Gamification of version control," Games and Software Engineering (GAS), 2012 2nd International Workshop on, Zurich, 2012, pp. 5-8.

Hassan, A.E. and T. Xie, Software intelligence: the future of mining software engineering data, in Proceedings of the FSE/SDP workshop on Future of software engineering research2010, ACM: Santa Fe, New Mexico, USA. p. 161-166.

S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), pages 283–294, November 2009

Fayyad, Usama; Piatetsky-Shapiro, Gregory; Smyth, Padhraic (1996). "From Data Mining to Knowledge Discovery in Databases" (PDF). Retrieved 17 December 2008.

Azzeh M, Cowling PI, Neagu D (2010) Software stage-effort estimation based on association rule mining and fuzzy set theory. Proc 10th Int Conf Comput Inform Technol, Bradford, UK: 249–256

T.A. Kumbhare, An overview of association rule mining algorithms, Int. J. Comput. Sci. Inform. Technol. 5 (2014) 927–930.

D. Graziotin, X. Wang, and P. Abrahamsson, "Happy software developers solve problems better: psychological measurements in empirical software engineering," PeerJ, vol. 2, no. 1, p. e289, Mar. 2014.

https://es.atlassian.com/git/tutorials/what-is-git

https://git-scm.com/

https://www.castsoftware.com/discover-cast

https://sdtimes.com/metrics/integration-watch-using-metrics-effectively/

https://en.wikiversity.org/wiki/Software_metrics_and_measurement

https://medium.com/data-science-group-iitr/association-rule-mining-deciphered-d818f1215b06