Measuring Software Engineering

By Luan Williams

How can Software Engineering be Measured and Assessed?

In the Modern World we are a part of today, Software Engineers make up a large part of the workforce. As of 2016 it is estimated that there are 21 million professional software developers. As you can imagine, the heads of these Software Companies want to be on top of everything that is happening within the company. They also want to be able to watch their employees and to track their progress, the work they are doing, (or aren't doing compared to their colleagues) and how well they are actually doing it. In order to compare the works of each employee to one another, we need to decide what metrics we are measuring, and whether these are the correct metrics. Software is measured for various different reasons, such as to:

- Establish the quality of the current product or process.
- To predict future qualities of the product or process.
- To improve the quality of a product or process.
- To determine the state of the project in relation to budget and schedule.

A Measurement is an indication of the size quantity amount or dimension of a particular attribute of a product or process. For example the number of errors in a system is a measurement. On the other hand, a Metric is a measurement of the degree that any attribute belongs to a system, product, or process. For example the number of errors per person per hours would be a metric. Thus, software measurement gives rise to software metrics. Metrics are related to the four functions of management:

- Planning
- Organising
- Controlling
- Improving

[1](https://en.wikiversity.org/wiki/Software_metrics_and_measurement)

A good example of this decision going wrong is looking at the old way of measuring Software Engineering; by measuring and comparing lines of code. Another term for this type of measurement is Source Lines of Code, or SLOC. Using SLOC to measure software progress is like using kg for measuring progress on aircraft manufacturing. It is totally inappropriate, and encourages bad practices such as Copy-Paste-Syndrome, it discourages

refactoring to make things easier, and has software developers stuffing their codebases with meaningless comments just to add to their lines of code. The only real use of SLOC is that it can help estimate how much paper to put in the printer when a printout of the complete source tree is done. [2] (https://stackoverflow.com/questions/3769716/how-bad-is-sloc-source-lines-of-code-as-a-metric) SLOC is also ineffective at comparing programs written in different languages, as different languages balance clarity in different ways. For example, assembly languages would require hundreds of lines of code to perform the same task as a few characters in APL.

Choosing metrics for measuring Software Engineering requires considerable thought and care to support the specific questions a business really needs to answer. Measurements should only be designed as a way to answer business questions. [3] (https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams) Once a business can decide what metrics to actually measure, and what the correct ones to use are, some effective software measurement can be done, and the data collected can actually be put to good use.

From doing a deep dive of different ways of measuring Software Engineering, I found a number of different objective metrics that should be monitored continuously in order to make incremental improvements to processes and production environments. Improvements in these will not guarantee that customer satisfaction levels will rise, but at least they are the correct things to be measuring. There are as many ways to measure a project as there are to build it, but most of these metrics are useless, or "Vanity Metrics". [4] (Eric Ries https://www.youtube.com/watch?v=0dvsNmL__9U)

Below I will talk about a number of very beneficial Metrics that are being used on a daily basis in the Software Engineering Industry.[5](found at https://www.agileconnection.com/article/4-balanced-metrics-tracking-agile-teams) To name a few, there is:

Cycle Time

Cycle time is the direct connection to productivity. The shorter the cycle time, the more things are getting done in a given time frame. You measure cycle time from when the work starts to when it is complete. In software terms, it is thought of as "Hands on keyboard" time. Measuring cycle time is best done automatically via an agile lifecycle tool of choice, but measuring with a physical task board can prove just as useful.

Defect Rate

This approach was developed and refined continuously to make software defects more visible as well as to analyse the findings to show the difference testing makes. The measurement is the connection between customer satisfaction and the team. The lower the defect rate, the more satisfied the customer is likely to be with the product. With a high rate of escaped defects, even the best of products can be left with unsatisfied customers. To do this, the number of problems such as bugs or deceits etc must be measured, and counted in the product once it has been delivered to the user. Until a story is done, it is still a process, so focus on the execution of the story instead of tracking in-progress defects.

Planned-to-Done Ratio

Thus metric is a way to measure predictability. If a team commits to thirty products and only delivers nine, the product owner has about a 30% chance of getting what they want. If, on the other hand, a team commits to ten products and delivers nine, the product owner has a 90% chance of getting what they want. Measuring this is a simple exercise of documenting how much work the team commits to doing at the start of the sprint versus how much they have completed at the end.

Velocity

This is one of the first one everyone thinks of when it comes to Agile metrics. It is probable the most overused and overrated of them all as well. To calculate velocity, you add up the total of all the story points that were moved to “Done” in the sprint once it is finished. This is not a measure of effectiveness, efficiency, competency, or anything else. It is simply a measure of the rate at which a given amount of problem statements are turned into tested software.

Sprint Burndown

In this metric, teams organise development into time-boxed sprints. At the outset of the sprint, the team forecasts how much work they can complete during a spring. A sprint turndown report then tracks the completion of work throughout the spring. The x-axis will represent time, and the y-axis will represent the amount of work left to complete, measured in either story points or hours. The goal is to have all the forecasted work completed by the end of the sprint.

Happiness

This is the team “health” metric. It creates awareness that puts that puts other metrics into better context. If all other metrics are perfect and happiness is low, then the team is probably going to burn themselves out fast. This metric is very important to measure too, as this can give the employer insight into the teams mental health, and if anything negative is showing up the employer can keep checks over this team member. Obviously a member of a team is going to have a slightly worse work ethic if their happiness in the workplace isn’t there.

Even though all of these metrics mentioned above, and many more used in industry on a daily basis, “there is no match in content between the increased level if metrics activity in academia and industry”[5] (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.2683&rep=rep1&type=pdf). Like many other core subjects within the software engineering mainstream (such as formal development methods, object-oriented design, and even structured analysis), the industrial take-up of most academic software metrics work has been woeful. Much of the increased metrics activity in industry is based almost entirely on metrics that were around in the 1970’s. Much of the academic metrics research is inherently irrelevant to the needs of the industry. This irrelevance is at two levels:

1. Irrelevance in scope: Much academic work has focused on metrics which can only ever be applied/computed for small programs, whereas all the reasonable objectives for applying metrics are relevant primarily for large systems. Irrelevance in scope also applies to the many academic models which rely on parameters which
could never be measured in practice.
2. Irrelevance in content: Whereas the pressing industrial need is for metrics that are relevant for process improvement, much academic work has concentrated on detailed code metrics. In many cases these aim to measure properties that are of little practical interest. This kind of irrelevance prompted Glass to comment:

What theory is doing with respect to measurement of software work and what practice is doing are on two different planes, planes that are shifting in different directions (Glass, 1994). [6] (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.2683&rep=rep1&type=pdf).

Computational Platforms for Measuring Software Engineering

These metrics, along with many others being used in businesses to this day, all obviously have to be calculated by some means. The data collected must be taken in, reviewed, and somehow used in a meaningful way, otherwise the collection of the data in the first place would be rendered completely useless. There are hundreds of different ways in which the software engineering process can be measured and assessed, and there are just as many different computational platforms available to do this work.

Manual PSP(Personal Software Process)

In 1995, Watts Humphrey authored A Discipline for Software Engineering, a ground-ground-breaking text that adapted organisational-level software measurement and analysis techniques to the individual developer. These techniques are called the Personal Software Process (PSP). The main goals of the PSP are to improve project estimation and quality assurance. These goals are pursued by collecting size, time, and defects data on an initial set of software projects and performing various analyses on it. For example, given the estimated size of a new system, a PSP analysis called PROBE provides an estimate of the time required based upon the relationship between time and size on prior projects. [7] (https://www.researchgate.net/publication/4016775_Beyond_the_Personal_Software_Process_Metrics_collection_and_analysis_for_the_differently_disciplined). PSP can be used to support both project estimation and quality assurance. It is useful in many ways. For one, it collects useful data for the project heads to be able to look over, and see how their projects are coming along. On the other hand, it is also useful for the developer themselves, because all the data collected is manually entered by the Software Developer, so they can personally review the work they have been doing themselves, and consider whether they have been putting in sufficient work to the projects or not. This is very good for the area of self-reflection in Software Engineering, which can lead to the Software Developer pushing themselves harder to prove to themselves that they are hard workers.

LEAP (Or other automated tools for PSP-Style metrics)

These tools all have the same basic approach as manual PSP in terms of user interaction: they display dialog boxes where the user records effort, size, and defect information. The tools also display various analyses when requested by the user. These approaches do an excellent job of lowering the overhead associated with metrics analysis, and substantially reduce the overhead of metrics collection. Developers can utilise their LEAP Historical data to substantially improve their project planning and quality assurance activities.[8](https://www.researchgate.net/publication/4016775_Beyond_the_Personal_Software_Process_Metrics_collection_and_analysis_for_the_differently_disciplined). LEAP tools are very effective in the sense that they still use all the same principles and ideologies that Manual PSP had originally mustered up, but the actual process of collecting and using this data is much more convenient, and done in a more practical and efficient way. I believe that because of the reduced effort in using it, it was much more effective than Manual-PSP, as the developers spend much less time doing arbitrary manual tasks, and much more time using their collected data effectively and efficiently in order to improve their own personal Software Engineering processes.

Hackystat

Hackystat is not a conventional monolithic system, but it is a collection of small services that work together in order to provide you with information about your software engineering activities. The main examples of this are the use of Google Charts within Hackystat in order to give useful graphical information of the software engineering work that has been done. Another one is the use of Twitter in order to provide certain communication facilities. In a nutshell, Hackystat is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data. The Hackystat framework supports three software development communities:

- Researchers: Hackystat can be used to support empirical software engineering experimentation, metrics validation, and more long range research initiatives such as collective intelligence.
- Practitioners: Hackystat can be used as infrastructure to support professional development, either proprietary or open source, by facilitating the collection and analysis of information useful for quality assurance, project planning, and resource management
- Educators: Hackystat is actively used in software engineering courses at the undergraduate and graduate levels to introduce students to software measurement and empirically guided software project management.

Hackystat users typically attach software ‘sensors’ to their development tools which unobtrusively collect and send “raw” data about development to a web service called the Hackystat SensorBase for storage. The SensorBase

repository can be queried by other web services to form higher level abstractions of this raw data, and/or integrate it with other internet-based communication or coordination mechanisms, and/or generate visualisations of the raw data, abstractions, or annotations.

A long range goal of Hackystat is to facilitate “Collective Intelligence” in software development, by enabling collection, annotation, and diffusion of information and its subsequent analysis and abstraction into useful insight and knowledge. Hackystat services are designed to co-exist and complement other components in the “cloud” of internet information systems and services available for modern software development. [9](**https://hackystat.github.io**) The current version of Hackystat is compatible with IDE’s such as Emacs, JBuilder, the Ant build system, and theJUnit testing tools. In these environments, Hackystat will collect activity data(which file, if any, is being actively modified by a developer), size data(non-comment source lines of code, or SLOC), and defect data(unit tests pass/fail status). The collection of the activity mentioned above can be incredibly useful for Project Managers, and also for the Software Developers themselves. They don’t have to worry about manually entering the work that they have been doing, as you used to have to do when using Manual PSP, as all the data is collected by the IDE you are using, and automatically logged. This cuts out the developer time needed to manually do this, and also removes any possibilities of discrepancies between actual work done and work “said to be done.” Hackystat is incredibly powerful, as it gives both the Software Engineer and the Project Managers a much more in-depth insight into the work that is being done on particular projects, which makes the comparison between team members, and between projects much simpler, and much more useful.

Github

A very useful, and probably the most widely known platforms for assessing the Software Engineering process is Github. Github is a repository hosting service. Git, like other version control systems, manages and stores revisions of projects. Although its mostly used for code, it could be used to manage any type of file, such as Word documents or Final Cut projects. It is basically a filing system for every draft of a document. Github can be very effective in the measurement of Software Engineering, as it gives a very readable and easy to understand view of all the work that particular developers have done on particular projects, and who has done what at every draft and stage of that process. This means that it is easy to compare who has done the most work, or who has done the most useful work on a project, to those who haven’t even bothered to look at it yet. It provides extremely easy to use version control, so if any problems or bugs occur within a particular version of a project, the entire project can be restored back to the version before this

bug was introduced. This makes it easy to track any mistakes which may be made in the Software Engineering process.
Github is one of the most important tools in Software Development in this day and age mainly because it is free, super functional, and extremely easy to use. Nearly everyone from First Year Computer Scientists to the Head Developers at Google has had to push some project to a Github Repository. It is the universal language for version control of inside of the Software Engineering world, and I believe that everyone who considers themselves a Software Engineer needs to have a Github account, in order to be able to show off the actual development work that has been carried out in their career as a Software Developer.

Bitbucket

There is another very useful and very widely used version control platform called Bitbucket. Bitbucket is a little less widely used, as it has around 3 million registered users, whilst Github has a massive 14 million registered users. Bitbucket is not nearly as open as Github when it comes to source codes. In fact, Bitbucket appeals more to enterprises who are developing proprietary source codes and keeping most of their trade secrets. Bitbucket's advantage over Github is that it supports the Mercurial VCS in addition to Git. But it also doesn't support SVN, yet. Bitbucket is written in Python and uses the Django web framework. Bitbucket is just as useful for version control as Github, the choice of which one to use is down to the developers wishes and needs for their Software Development Platform. It is a powerful tool for keeping different versions of a project or program in the same place, so can easily track the contributors and contributions on a particular project, meaning the comparison of work done between Software Engineers is simple to do.

Timeflip.io

A more abstract platform in which the computation and collection of Software Engineering measurements can be done is known as Timeflip. This is one of the only computational platforms available for the measurement of the Software Engineering Process that is a physical entity, rather than just a program or a website. Timeflip is an interesting piece of technology; it is a 12-sided plastic 'dice' with a motion sensor built inside, and there are 12 different 'tasks' on each of the sides. To use it, you simply start tracking when you start your work, and as your day goers on, you flip the Timeflip facing up to match whatever task you are carrying out. For example, if you are browsing the web, you set the web browsing side up, and leave it sitting like that for as long as you are browsing. When you move on to something else, such as a business call or developing software, you flip it to match that task. This means it tracks the time you spend doing everything throughout the business day. If used correctly, you will get an accurate measure of how well you spend your time during the day. This can be extremely useful as a

computational platform for working out how much time per day a person spends being productive, or carrying out certain tasks. This makes the gathering of measurable data extremely easy, as all you have to do is flip a dice. With enough Software Engineers in a firm or on a team using these, the comparison between different team members and how efficiently they spend their time becomes very obvious. In my eyes, this is one of the best computational platforms available if you want to track an individuals productivity levels.

The Mylar Monitor

Mylar is an open source technology project hosted at www.eclipse.org/mylar. The Mylar Monitor is a standalone framework that collects and reports on trace information about a user's activity in Eclipse. The Mylar Monitor captures events such as preference changes, perspective changes, window vents, selections, periods of inactivity, commands invoked through menus or key bindings, and IRL's viewed through the embedded Eclipse Browser. These events are logged to a local file and uploaded by the developer to a HTTP server. The uploading mechanism manages user IDs, pro-vides anonymity if requested, and obfuscates the handles of targets of selections and other such user data that might be collected. This ensures that sensitive information about the source code and user isn't transmitted. [10] (https://www.researchgate.net/publication/220093228_How_Are_Java_Software_Developers_Using_the_Eclipse_IDE) As we can see above, the Mylar Monitor is a great example of how these comparable metrics can be gathered. Eclipse has built in functionality for tracking the work done by Software Engineers. This makes the comparison of the work done by different Software Engineers using Java much simpler, as the data is collected by their IDE as they are working on various projects, and sent away to HTTP servers for the heads of these projects to look over, giving them a great overview of the work being done on their projects.

As you can see above, there are many different computational platforms available to perform the measurement and assessment of Software Engineering. There are many more being used in the industry which weren't mentioned above, as nearly every Software Engineering firm will be using some form of computational platform to be able to track, monitor, and compare the developmental work being done by individuals, and by the team involved. Choosing which one to use all depends on what works for the business, or the individual. Some platforms will be perfect for one team of Software Engineers, but wouldn't even be considered by the next. It's all down to what you want to actually measure, and how you would like to do so.

Algorithmic Approaches available for measuring Software Engineering

Depending on what kind of data is required from the use of these algorithmic approaches for measuring Software Engineering, there is a wide variety of choice as to which kind of algorithms to use. There are algorithms which can predict how long it should take you to develop a particular program, ones which can give you an estimate of the efficiency of your team, and there are Algorithmic Methods for Cost Estimation of Software Development. They are all useful in their own ways to different people, it all depends on what kind of information is needed at the end of the calculations.

Algorithmic Methods for Cost Estimation

Cost estimation is a key part of project management. An accurate cost estimation plays a huge role in completing a project within a particular time and budget. Inaccurate cost estimation can lead to project failure, big overruns, and performance compromises. These mathematical equations to perform software estimation are based on research and historical data and use inputs such as SLOC, number of functions to perform, and other cost drivers such as language, design, methodology, skill-levels, risk assessments etc. The algorithmic methods have been largely studied and there are lots of models that have been developed, such as COCOMO models, Putnam model, and Function points based models.

The advantages of the algorithmic methods are:
- It is able to generate repeatable estimations.
- It is easy to modify input data, refine, and customise formulas.
- It is efficient and able to support a family of estimations or a sensitivity analysis.
- It is objectively calibrated to previous experience.

The disadvantages of algorithmic methods are:
- It is unable to deal with exceptional conditions, such as exceptional personnel in any software cost estimating exercises, exceptional teamwork, and an exceptional match between skill-levels and tasks.
- Poor sizing inputs and inaccurate cost driver rating will result in inaccurate estimation.
- Some experience and factors cannot be easily quantified.

[11](https://pdfs.semanticscholar.org/4935/410e22756a262e41614747e0d94291cf860b.pdf)

COCOMO Models

The Constructive Cost Model (COCOMO) is an easy-to-understand model that predicts the effort and duration of a project, based on inputs relating to the size of the resulting systems and a number of cost drivers that affect productivity. It is an algorithmic approach to estimate the cost of a software project. By using COCOMO, it is possible to calculate the amount of effort and the time schedule for projects. From these calculations it is possible to then find out how many team members are required to complete a project on time. The main metric used for calculating these values is lines of code(KLOC for COCOMO II, or KDSI for COCOMO 81 and measured in thousands), function points (FP), or object points(OP). The basic COCOMO model has a very simple form:
MAN-MONTHS = K1* (Thousands of Delivered Source Instructions) K2
Where, K1 and K2 are two parameters dependent on the application and development environment.

COCOMO is very useful, as it is an industry standard, the information gathered is very profound, and very easily available. It is not reasonable to do these calculations for small projects though.

Putnam Model

Another popular software cost model is the Putnam model. The form of this model is:

Technical constant C= size * B^1/3 * T^4/3

Total Person Months B=1/T^4 *(size/C)^3

T= Required Development Time in years

Size is estimated in LOC

Where, C is a parameter dependent on the development environment and it is determined on the basis of historical data of the past projects.

Rating: C=2,000 (poor), C=8000 (good) C=12,000 (excellent)

The Putnam model is very sensitive to the development time, decreasing the development time can greatly increase the person-months needed for development.

An estimated software size at project completion and organisational process productivity is used. Plotting effort as a function of time yields the Time-Effort Curve. The points along the curve represent the estimated total effort to complete the project at some time. One of the distinguishing features of

the Putnam model is that total effort decreases as the time to complete the project is extended. This is normally represented in other parametric models with a schedule relaxation parameter.

This estimating method is fairly sensitive to uncertainty in both size and process productivity.

One of the key advantages to this model is the simplicity with which it is calibrated. Most software organisations, regardless of maturity level can easily collect size, effort and duration (time) for past projects. Process Productivity, being exponential in nature is typically converted to a linear productivity index an organisation can use to track their own changes in productivity and apply in future effort estimates.

One significant problem with the PUTNAM model is that it is based on knowing, or being able to estimate accurately, the size (in lines of code) of the software to be developed. There is often great uncertainty in the software size. It may result in the inaccuracy of cost estimation. Time is very dominating factor in Putnam model. Putnam model is basically based on only two variables which are time and size. It is not considering all other aspects of software development life cycle. Whereas in COCOMO II we are getting more nearer results because it is considering almost all aspects of SDLC.
[12](https://pdfs.semanticscholar.org/4935/410e22756a262e41614747e0d94291cf860b.pdf)

Function Point Analysis Based Methods

The Function Point Analysis is another method of quantifying the size and complexity of a software system in terms of the functions that the system delivers to the user. A number of proprietary models for cost estimation have adopted a function point type of approach, such as ESTIMACS and SPQR/20. There are two steps in counting function points:

- Counting the user functions. The raw function counts are arrived at by considering a linear combination of five basic software components: external inputs, external outputs, external inquiries, logic internal files, and external interfaces, each at one of three complexity levels: simple, average or complex.. .The sum of these numbers, weighted according to the complexity level, is the number of function counts (FC).
- Adjusting for environmental processing complexity. The final function points is arrived at by multiplying FC by an adjustment factor that is determined by considering 14 aspects of processing complexity. This adjustment factor allows the FC to be modified by at most 35% or - 35%.

The collection of function point data has two primary motivations. One is the desire by managers to monitor levels of productivity. Another use of it is in the estimation of software development cost.

There are some cost estimation methods which are based on a function point type of measurement, such as ESTIMACS and SPQR/20. SPQR/20 is based on a modified function point method. Whereas traditional function point analysis is based on evaluating 14 factors, SPQR/20 separates complexity into three categories: complexity of algorithms, complexity of code, and complexity of data structures. ESTIMACS is a propriety system designed to give development cost estimate at the conception stage of a project and it contains a module which estimates function point as a primary input for estimating cost. [13] (https://pdfs.semanticscholar.org/4935/410e22756a262e41614747e0d94291cf860b.pdf).

The accurate prediction of software development costs is imperative for making good management decisions and accurately working out how much effort and time a project needs in terms or project managers, as well as system analysts and developers. There are many different algorithmic approaches available for cost estimation, and no one method is particularly better or worse than the other.

Cyclomatic Complexity

Cyclomatic complexity is a software metric used to measure the complexity of a program. These metrics measure independent paths though program source code. An Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. Cyclomatic Complicity can be calculated with respect to functions, modules, methods, or classes within a program. The complexity of a program can be defined as:

$V(G) = E - N + 2$

where,
E = Number of Edges
N = Number of Nodes
$V(G) = P + 1$
Where P = Number of predicate nodes (Nodes that contain conditions).

Cyclomatic Complexity can help developers and testers determine independent path executions. It means Developers can assure that all paths have been tested at least once, which can improve code coverage. It can also evaluate the risk associated with the application or program being developed. [14] (https://www.guru99.com/cyclomatic-complexity.html)

Halstead's Complexity Measures

Halstead's metrics depends upon the actual implementation of a program and its measures, which are computed directly from the operators and operands from source codes in static manner. It allows to evaluate testing time, vocabulary size, difficulty, errors, and efforts for C/C++/Java Source Code. According to Halstead, "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands". Halstead metrics think a program as sequence of operators and their associated operands.

He defines various indicators to check complexity of module.

Parameter	Meaning
n1	Number of unique operators
n2	Number of unique operands
N1	Number of total occurrence of operators
N2	Number of total occurrence of operands

When we select source file to view its complexity details in Metric Viewer, the following result is seen in Metric Report:

Metric	Meaning	Mathematical Representation
n	Vocabulary	n1 + n2
N	Size	N1 + N2
V	Volume	Length * Log2 Vocabulary
D	Difficulty	(n1/2) * (N1/n2)
E	Efforts	Difficulty * Volume
B	Errors	Volume / 3000
T	Testing time	Time = Efforts / S, where S=18 seconds.

Depending on the metric you want to work out, there are many different Algorithms for the calculations. These algorithms can be vital in the planning and development of software, as a lot of different things about the program can be determined before the program is even written. This can make planning extremely easy, and will help with the development of programs.

Ethics Concerns surrounding Measuring Software Engineering

Whilst all of these methods for Measuring Software Engineering can be extremely beneficial to both Software Firm heads, and the Software Developers themselves, there is always the question as to how much of these measures are Ethically Correct. Do the project managers really have the right to be tracking your every move? Should they be allowed to collect data on the work you are doing? How much monitoring is really warranted? In every case, the argument is filled with assurances that all the methods on gathering data on a programmer will only be used for good, and useful things, like watching productivity levels. But what if this data is used against the developer? What if the wrong person gets a hold of the data gathered about a particular person.

The use of students in empirical studies has also raised some ethics concerns in the past. Several empirical studies have been carried out with college students as subjects in the last few years. Researchers usually use these studies to pilot experiments before they are carried out in industrial environments. But is it ethically correct to “use” students for the benefit of researchers? Is it useful or suitable in the context of a college course? Surely there would be more productive ways for students to spend their time rather than devoting it to participating in an empirical study. This also poses the question as to whether or not it is ethically correct to encourage students to participate in an empirical study.

Aside from students, we can also question whether or not a Software Developer wants to be “watched” or “tracked” by their bosses. What if they are having a particularly bad day at the office due to unforeseen family circumstances, or some sort or out-of-work personal issues. Is it still ethically right to collect data on them, and compare them to other developers who may be having the most productive day of their life? These are just some of the ethical concerns raised by Measuring Software Engineering. Do these software companies really and truly have the right to measure each and every move of their Software Developers. Whilst all this data collection may prove very beneficial to the heads of the companies, but what benefits do the actual Software Developers themselves see?

Conclusion

From all the work that I have done to deliver this report, I have discovered many different things about about the ways in which the software engineering process can be measured. I discovered a large amount of different metrics that are used in the industry on a daily basis, such as Cycle Time, Defect Rate, Planned-To-done Ratio, Velocity, Sprint Burndown, Happiness, and multiple others that I didn't have a chance to mention. They are all widely used, and extremely important for the measuring and assessment of the Software Engineering process. Without these metrics, it would be extremely difficult to measure and compare the work that is being done on particular Software Engineering projects. These metrics make it possible to compare different Engineers work, and see who is doing beneficial work, and who is just filling up their time writing useless code. This makes it very easy to reward those doing important and useful work, and to discipline those who aren't contributing to the team effort.

I also explored many different Computational Platforms available to perform this work, which can all collect data on the Software Engineers using them, and can then measure and assess the Software Engineering process. The main computational platforms that are used today that I found during my research are Manual PSP(Personal Software Process), LEAP(Or other automatic PSP-Style metrics), Hackystat, Github, Bitbucket, timeflip.io, and The Mylar Monitor. These are all different services available to allow for the measurement and assessment of the Software Engineering Process. They are all heavily used in industry, and there are many more services available that I didn't get a chance to mention. These computational platforms are extremely important in industry, as they can do all the measurement work in the background, or very easily with a version control website. The work that these platforms do can prove very beneficial for Software Teams, as they can help conclude who is doing what work, and how important the work being done is.

There are also many different algorithmic approaches available for measuring and assessing the Software Engineering process, such as the COCOMO Models(Constructive Cost Model), the Putnam Model, Function Point Analysis Based Methods, Cyclomatic Complexity, and Halstead's Complexity measures. These are all Algorithms that take in particular variables taken from Programs, or the Software Engineering Process, and can calculate different things such as how much a project is going to cost, how long it should take to complete a project, or even the complexity of an actual program. These can all be very useful for measuring the Software Engineering Process, as they can predict the entire time needed for the process, or how much money or man hours will be needed. These can be

very important to companies as it can give them an estimation of the workload they are taking on before any code is written.

All in all, there are thousands of different ways in which the Software Engineering process can be measured and assessed, and choosing which ways to use all depends on the type of project, the type of business, or the type of Software Engineers involved in a project. These measures and assessments of the process can either compliment or conflict with one another depending on how they are used. These measures and assessments should all be considered when choosing how you want to assess the process, as they could all prove to be beneficial in their own ways. At the same time, some of them should be completely disregarded for certain projects, as they would just waste time trying to even start using them.