Measuring Software Engineering
Eoin Leahy - 16320343

INTRODUCTION

Software, like any other product must be measurable, and as Software Engineers, we must take an interest in this. This is easier said than done, as the difficulty lies in how we can measure software. Take many other types of products, food and drink for example. These have tangible, pre-defined metrics that are tied into measurement systems that has been standardised and (almost) universally agreed upon. However, a big part of the process of measuring software can be left to the discrepancy of the person the taking the data. Was this product successful? Does it have enough features? A look into the production of that software, i.e. Software Engineering, may give us some valuable insights, but we must take that human factor into account. In this report I will try to explain how Software Engineering can be assessed in terms of measurable data, platforms that do this work, how we can interpret this data and the ethics surrounding this work.

MEASURABLE DATA

Code Quality

A big part of measuring Software Engineering can be rather subjective and as I said before, up to the discrepancy of the whoever is taking these measurements. The quality and success of the software must be considered. A product may have short running times and fit the exact requirements specification, doing everything it was designed to do perfectly, but it is not secure and can be subject to exploitation. Can we then objectively say that this product was successful and is of a high quality? We then must to ask ourselves why it wasn’t made secure in the first place, what tradeoffs did we make during development that resulted in an insecure product?

Lead Time

A look into the development process of software can often give valuable context to the product we ended up, and usually it has measurable data to complement this. One such metric is Lead Time, the amount of time between when a product was first envisioned, to when it was finally given to the customer. It is easy to say that the 'size' of the software has a positive correlation to the time allocated to it, but we need to look at the milestones during development, and check if the project met these milestones. This then raises the question of how much of the allocated development time was just 'crunch time'. A product could be objectively successful even it was poorly managed in development.

A software project could miss its initial deadline but be better off as a result of this. Huge architectural changes might be made later in development, but if making these changes led to a more efficient, reliable and secure product then it could be worthwhile. A product with a shorter lead time might not be 100% perfect, but it does meet the minimum of software quality factors and does the job required. The second case is more likely to have a happy customer who would later commission more software products. For this reason, Lead Time is not a perfect metric without proper context.

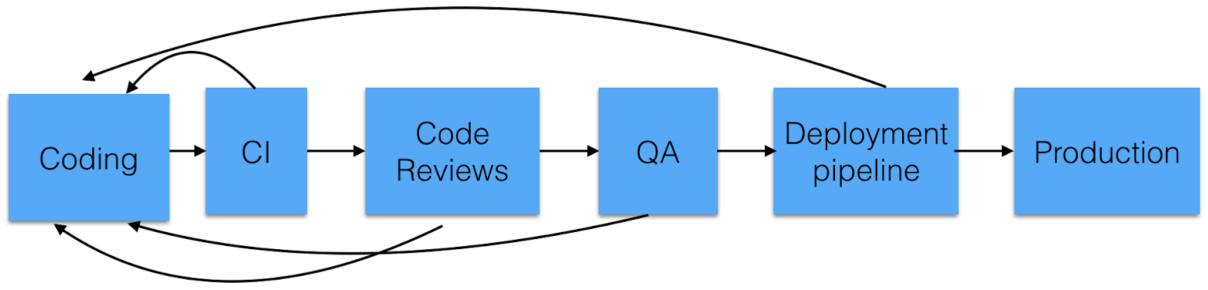
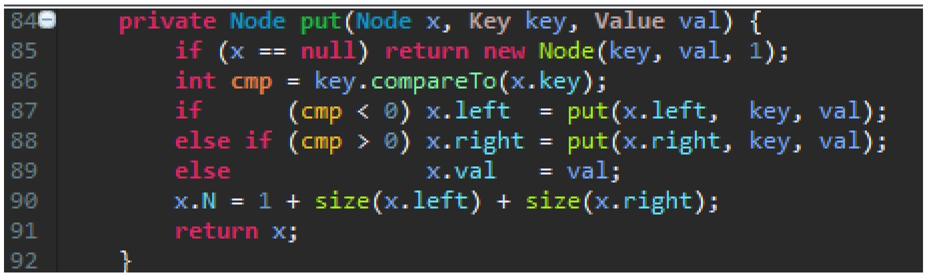


Figure 1: Lead Time including Cycle Time.

Figure 1 above shows the typical development of a software product, with emphasis on the Cycle Time. Cycle time is the time it takes to make a change to a software system during development. It is an important part of the Lead Time.

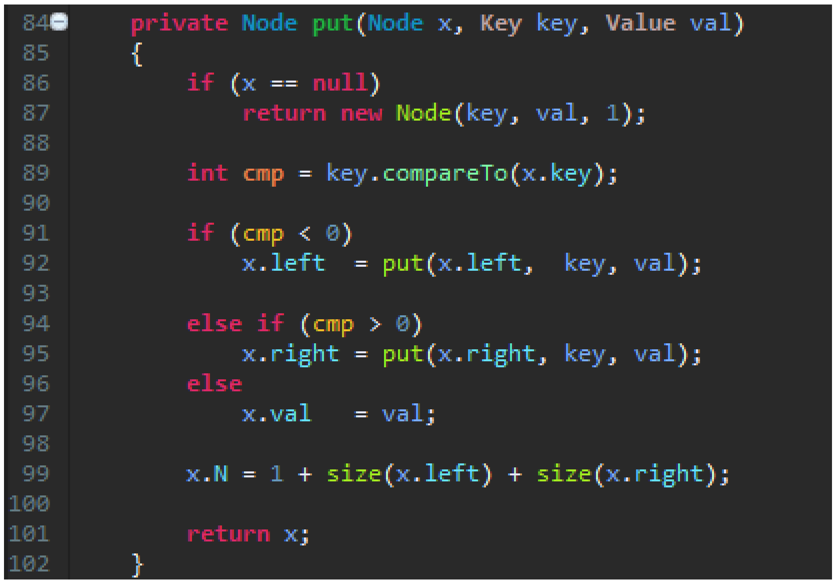
Lines of Code

By looking into the source code, we often get valuable metrics that are useful in measuring the Software Engineering process itself. The simplest of these metrics to measure is lines of code in the project. The advantage of measuring lines of code is the simplicity in gathering data the data itself, it can be done in the command line and requires no huge effort. The obvious disadvantages to this are that it encourages developers to pad out their code and this could in part make it unreadable, thus leading to a decrease in developer efficiency.



```java
    private Node put(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val, 1);
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x.left  = put(x.left,  key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else              x.val   = val;
        x.N = 1 + size(x.left) + size(x.right);
        return x;
    }
```

Figure 2: Unpadded code, slightly more readable



```java
    private Node put(Node x, Key key, Value val)
    {
        if (x == null)
            return new Node(key, val, 1);

        int cmp = key.compareTo(x.key);

        if (cmp < 0)
            x.left  = put(x.left,  key, val);

        else if (cmp > 0)
            x.right = put(x.right, key, val);
        else
            x.val   = val;

        x.N = 1 + size(x.left) + size(x.right);

        return x;
    }
```

Figure 3: Code that has been padded out to extend line count

If we compare **figures 2** and **3**, it becomes obvious how simple it is to add extra lines to a code base. Both snippets of code do the exact same thing, however the latter example is 10 lines longer and might look better to an employer who enforces line quotas. Aside from this, counting lines of code will also encourage developers to just repeat code instead of writing it in methods and to be more verbose in how they write code. Counting code lines doesn't consider any refactoring that took place, which could drastically reduce the line count, during the development process.

Number of Bugs

Another source code metric to consider is the number of bugs prevalent in the code base, and this can further extend to how many working hours must be invested to solve these bugs. Poorly written and maintained code is bound to be bug-ridden and will inevitably lead to some type of failure during development or deployment. Solving these failures could cost dozens to hundreds of man-hours, adding to an increased lead time, or could cost the customer a lot of money if failure happens during peak business hours.

Code Churn

Code churn is easily measurable in Software Engineering. It refers to the percentage of code that is changed when a developer makes an edit to a project. This could be the removal, addition or modification of code during a given time period in development. Code churn does give us an insight into the management of a software product, if it is high or spikes frequently it provides an indication that something has gone wrong during the process. Ideally, code churn would decrease and stabilise as you near deployment, this implies a safer and more structurally intact code base.

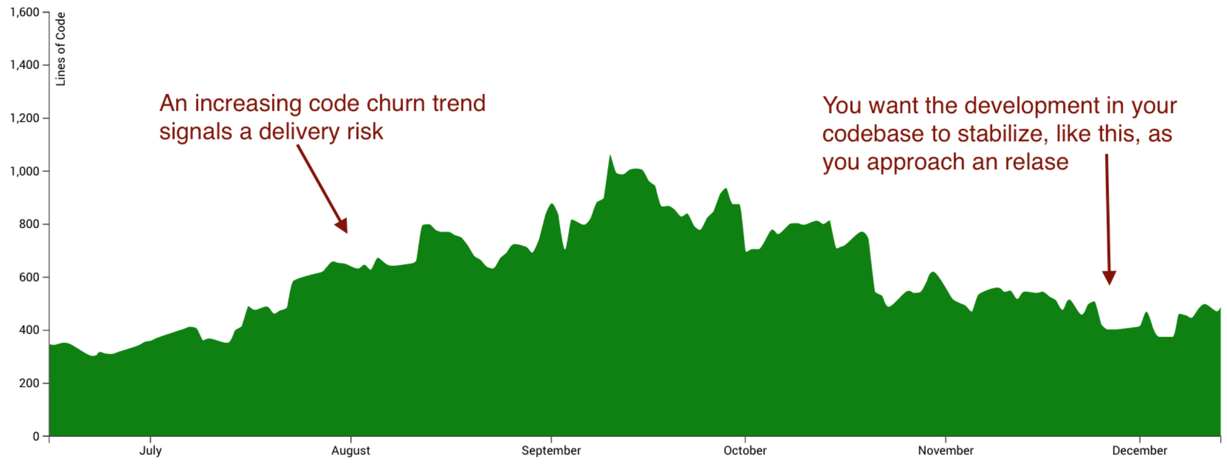
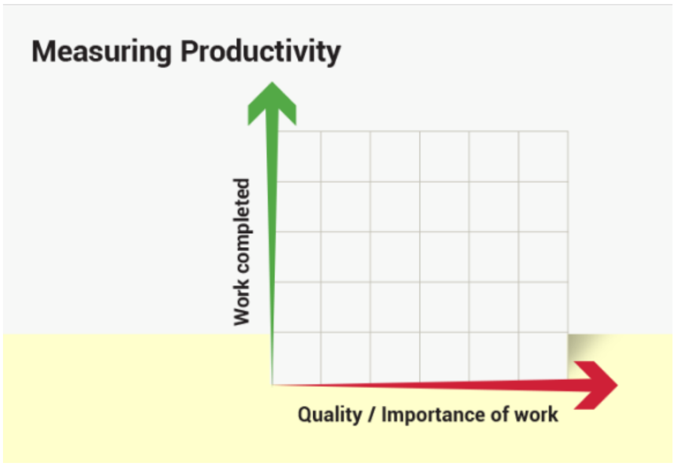


Figure 4: Code churn for a project, note that it stabilises towards the end

If we look at the code churn metrics for the individual Software Engineer, we can see how efficient they are by themselves. High rates of code churn aren't inherently bad, as the Engineer could have a perfectly valid explanation, but it is worth paying attention to. During a project there could be a major miscommunication from management, the specifications could've been changed by the customer, or maybe the engineer was just adding polish to finished code or overcame a huge problem in the implementation.

Productivity, Quantity & Conclusion



Not all these metrics can be considered perfect and, in my opinion, they are useless when used individually, without proper context. As such, the human factor still has an important stake in how we measure productivity of a Software Engineer. An Engineer could have solved a dozen different problems in one day, but there is always the chance that only half of these added any value. We could investigate the more discrete metrics such as hours spent on a project and lines of code, but these bring their own series of problems to light. There is no definitive metric type for measuring Software Engineering, and we should investigate how the quantity of meaningful work completed correlates with the quality and value of work completed. The two of these combined should provide a meaningful insight into the Software Engineering process of a product.

Overview of Platforms to do this Work

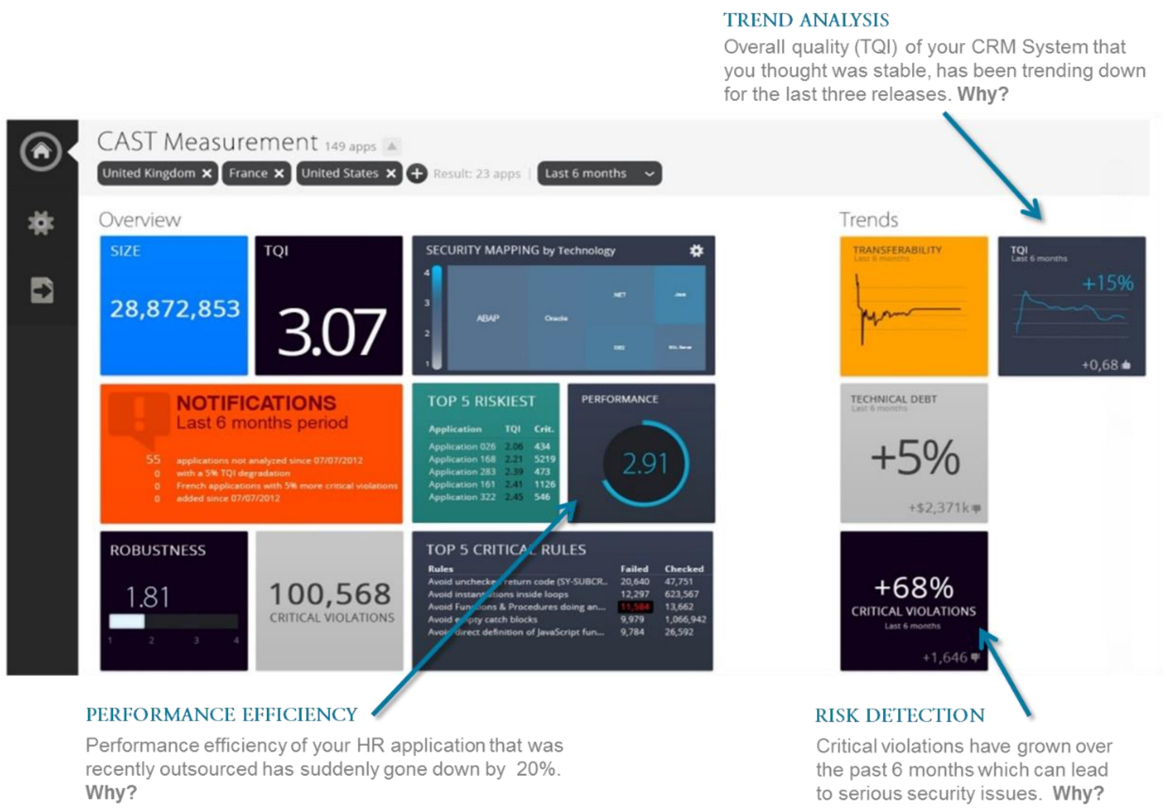
We have looked at the types of data that can be measured and I will now explain the different tools and platforms which can be used to find his data. It is important for a company to provide their employees with these tools, as it can say a lot about the efficiency of the development teams, this can further keep the major stakeholders happy.

Software Intelligence Systems - CAST

Software intelligence systems provides us ways of find this information. CAST is an example of a company that provides products for this task. From CAST's own description of Software Intelligence;

"Software Intelligence is the insight into a complex application's composition and integrity based on the analysis of the database structure, application frameworks, project files, stored procedures, and source code."

Their products, specifically the Application Intelligence Platform (AIP) do that job. The overall architecture of the system is analysed for weaknesses and findings are compared against industry standards. **Figure 5** below shows a snippet of the information found upon execution.



55 applications not analyzed since 07/07/2012
0 with a 5% TQI degradation
0 French applications with 5% more critical violations
0 added since 07/07/2012

Application	TQI	Crit.
Application 026	2.06	434
Application 168	2.21	5219
Application 283	2.39	473
Application 161	2.41	1126
Application 322	2.45	546

Rules	Failed	Checked
Avoid unchecke return code (SY-SUBCR..	20,640	47,751
Avoid instan tions inside loops	12,297	623,567
Avoid Fun ions & Procedures doing an...	11,584	13,662
Avoid e pty catch blocks	9,979	1,066,942
Avoi direct definition of JavaScript fun...	9,784	26,592

TREND ANALYSIS
Overall quality (TQI) of your CRM System that you thought was stable, has been trending down for the last three releases. **Why?**

PERFORMANCE EFFICIENCY
Performance efficiency of your HR application that was recently outsourced has suddenly gone down by 20%. **Why?**

RISK DETECTION
Critical violations have grown over the past 6 months which can lead to serious security issues. **Why?**

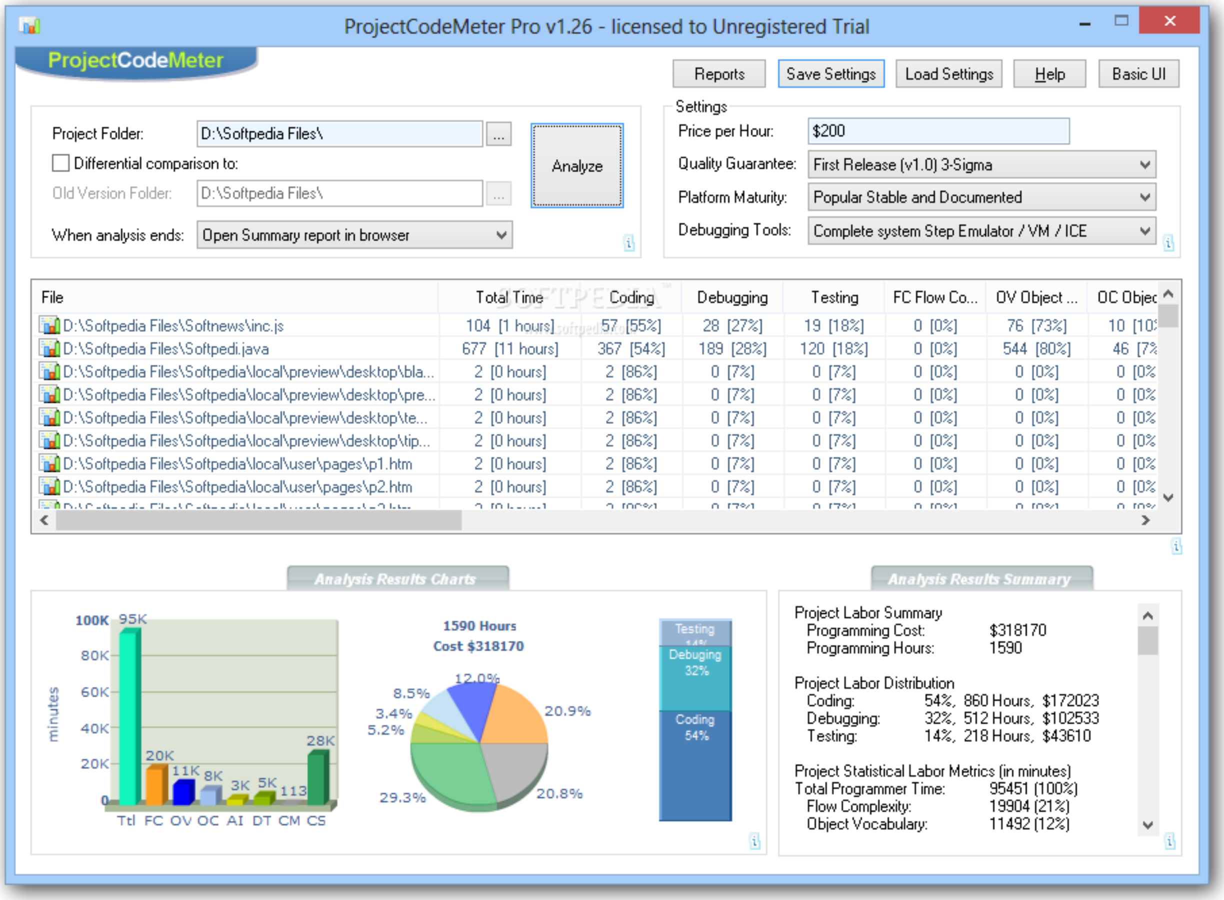
Figure 5: CAST's software measurement

ProjectCodeMeter

ProjectCodeMeter provides a wide array of the analytics of a software development project. These include but are not limited to:

- Work hours
- Value of source code in time and money
- Predicted development budget
- Team Productivity

It also provides basic source code metrics and the quality of the source code. **Figure 6** shows an example of these measurements in a development project. Whether or not these are useful is up to debate. It seems like it would be used heavily by the management side of the project.


ProjectCodeMeter Pro v1.26 - licensed to Unregistered Trial

Project Folder: D:\Softpedia Files\
☐ Differential comparison to:
Old Version Folder: D:\Softpedia Files\
When analysis ends: Open Summary report in browser

Settings
Price per Hour: $200
Quality Guarantee: First Release (v1.0) 3-Sigma
Platform Maturity: Popular Stable and Documented
Debugging Tools: Complete system Step Emulator / VM / ICE

File	Total Time	Coding	Debugging	Testing	FC Flow Co...	OV Object ...	OC Objec
D:\Softpedia Files\Softnews\inc.js	104 [1 hours]	57 [55%]	28 [27%]	19 [18%]	0 [0%]	76 [73%]	10 [10
D:\Softpedia Files\Softpedi.java	677 [11 hours]	367 [54%]	189 [28%]	120 [18%]	0 [0%]	544 [80%]	46 [7%
D:\Softpedia Files\Softpedia\local\preview\desktop\bla...	2 [0 hours]	2 [86%]	0 [7%]	0 [7%]	0 [0%]	0 [0%]	0 [0%
D:\Softpedia Files\Softpedia\local\preview\desktop\pre...	2 [0 hours]	2 [86%]	0 [7%]	0 [7%]	0 [0%]	0 [0%]	0 [0%
D:\Softpedia Files\Softpedia\local\preview\desktop\te...	2 [0 hours]	2 [86%]	0 [7%]	0 [7%]	0 [0%]	0 [0%]	0 [0%
D:\Softpedia Files\Softpedia\local\preview\desktop\tip...	2 [0 hours]	2 [86%]	0 [7%]	0 [7%]	0 [0%]	0 [0%]	0 [0%
D:\Softpedia Files\Softpedia\local\user\pages\p1.htm	2 [0 hours]	2 [86%]	0 [7%]	0 [7%]	0 [0%]	0 [0%]	0 [0%
D:\Softpedia Files\Softpedia\local\user\pages\p2.htm	2 [0 hours]	2 [86%]	0 [7%]	0 [7%]	0 [0%]	0 [0%]	0 [0%

```
Project Labor Summary
   Programming Cost:          $318170
   Programming Hours:         1590

Project Labor Distribution
   Coding:              54%,  860 Hours,  $172023
   Debugging:           32%,  512 Hours,  $102533
   Testing:             14%,  218 Hours,  $43610

Project Statistical Labor Metrics (in minutes)
Total Programmer Time:        95451 (100%)
   Flow Complexity:           19904 (21%)
   Object Vocabulary:         11492 (12%)
```

Figure 6: ProjectCodeMeter

Version Control Software - Git

On a smaller scale, the version control software itself can be used as a platform for taking measurements. Git is a common example of this and I am very familiar with it. Each time a commit is made, the developer can view the changes made to code. The number of lines changed is presented to the user in a clear fashion. This history is then stored, graphed and can then be viewed by the contributors at any time. Another convenient feature is the frequency graph of a given user that shows how often a user commits (although this data could be considered meaningless as a commit could be one line). Scripts can be added to calculate the code churn of a user within a repo.

Measuring within the Codebase itself.

A lot of the time these platforms are not available, and it is more effective to measure the efficiency of a program within the codebase itself. Code can be added to find simple metrics such as execution time and how many threads are created. This could also be used to compare multiple implementations of the same method. Measuring within the codebase itself can be useful for students who don't have access to high-end platforms and need to find out information the programs they have written. This should rarely be the first option for a professional with better tools.

ALGORITHMIC APPROACHES

Halstead's Complexity Measures

Several metrics can be taken from the source code of a project to calculate the complexity of implementing a program:

- n1 = no. of distinct operators in program
- n2 = no. of distinct operands in program
- N1 = total number of operator occurrences
- N2 = total number of operand occurrences
- Program Length: N = N1 + N2
- Program volume: V = N log 2 (n1 + n2)

Specification abstraction level: L = (2 * n2) / (n1 * N2)

Program Effort: E = (n1 + N2 * (N1 + N2) * log2 (n1 + n2)) / (2 * n2)

Effort represents the mental strain taken when creating an software product.

Maintainability Index Algorithm

Using some source code metrics, we can calculate how maintainable a code base is on a scale from 0-100 (typically 0-25=poor, 25-75=maintainable, 75-100 = highly maintainable).

- V = Halstead Volume
- G = Cyclomatic Complexity
- LOC = Lines of Code

MI = MAX(0,(171 - 5.2 * ln(V) - 0.23 * (G) - 16.2 * ln(Lines))*100 / 171)

This algorithm is used in calculating code maintainability in some IDEs such as Visual Studio.

Other

Other calculations can be made such as how many days/weeks it took X number of engineers to implement a feature or program. This has its many obvious flaws.

Ethical Concerns

Interpretation of Data

A big ethical concern with the measurement of Software Engineering is how the data gathered is so open to interpretation. When an individual's credibility as a programmer is at stake, it is especially important to consider all the aspects around the gathered metrics. Further investigation should be taken when an Engineer doesn't meet some sort of quota or standard. A well-managed development team won't be subject to poor policies and mis-communication won't be as prevalent.

Privacy Concerns

Gathering data on employees can often be invasive. Measuring how often an Engineer gets up from their desk or how long they spend actually programming can be unattractive to potential employees and even demoralizing for current employees. Despite it being the right of an employer, gathering this data then manipulating it to create visualisations of what the employee does during the workday is dystopian at best.

Security

Data stored on employees is personal, and every individual should have a reasonable expectation to privacy. Storing data on an insecure server could lead to huge leaks of employee information. This could be detrimental to the well-being of the workforce.

Conclusion

Software Engineering overall is a difficult subject to measure. Data is so open to interpretation that it is not easy to come to a reasonable conclusion without further investigation. Many methodologies have been provided to us to measure the process, what's important is the information we gather and how we value different metrics. Lines of code written says nothing about the actual quality of the product, for instance. A concern lies in how we treat the data we have. One small metric can tell many stories, so it is mandatory that we approach the topic with care. Customer satisfaction is one of the most important metrics and is hard to quantify, but a returning customer is usually a good sign.

Bibliography / References

https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams

https://blog.gitprime.com/5-developer-metrics-every-software-manager-should-care-about/

https://www.castsoftware.com/glossary/software-measurement

https://www.tutorialspoint.com/software_quality_management/software_quality_management_measurement.htm

https://www.sciencedirect.com/science/article/pii/S0164121217300663

https://ardalis.com/static-code-analysis-and-quality-metrics

https://codescene.io/docs/guides/technical/code-churn.html

https://www.7pace.com/blog/how-to-measure-developer-productivity

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

http://www.projectcodemeter.com/cost_estimation/help/GL_maintainability.htm

http://sunnyday.mit.edu/16.355/metrics.pdf

Horst Z. (1997). *A Framework of Software Measurement.* Hawthorne, NJ, USA.

Walter de Gruyter & Co.