

Measuring Engineering – A Report

Anna-Zorina Honer

15324090

Trinity College Dublin



**Trinity College Dublin**  
The University of Dublin

## Abstract

The purpose of this paper is to examine and compare various methods of measuring progress, productivity, and the process of coding. Methods suggested including a combination of examining commit history against lines of code

*Keywords:* engineering process, measurable data, algorithmic approaches, ethics.

## Contents

Abstract .....	2
Measuring the Software Engineering Process. ....	3
Algorithmic Approaches .....	5
Alternative Programming Environments .....	6
References .....	7
Figures.....	8

### **Measuring the Software Engineering Process.**

“Moving from measurements to metrics is like moving from observation to understanding”<sup>1</sup>.

The purpose of this paper is to reflect on the various methods of measuring software engineering and the applications this data can have on the engineering process. The topic of how to measure one’s progress through coding is highly debated and many agree that a combination of many methods may be needed to reach the most accurate conclusion. We will hit on various methods which have been suggested and list the advantages and otherwise of each method, how they may be implemented, and the ethics behind measuring this data.

Before we begin discussing the various methods of measuring the engineering process, we must ask why we are measuring this information – a new programmer may wish to track their progress through a new language, or a seasoned employee may need to report on their work to a higher up. Each situation requires a different approach and must be considered before heading into the measurement.

For someone who is new to programming, it may be tempting to use Source Lines of Code (SLOC) as a measurement of progress, it is a clear and understandable approach that does not require programming experience to record. However, it is easy to exploit and may encourage bad programming habits. If higher SLOC was the ideal situation, why would a programmer use more concise code, when the higher SLOC would be encouraged. Despite this, SLOC is used often to as a basis for cost estimation as seen in Boehm’s survey on cost estimations approaches<sup>2</sup>.

That being said, SLOC is not inherently a bad metric for progress through coding. SLOC is often considered when calculating the complexity of code which is believed to be a good measure of one's progress when learning to code. For a student, having highly complex code may be an achievement however higher code complexity means it is "harder to test and maintain"<sup>3</sup>. To calculate the complexity of a piece of code, you may refer to its Cyclomatic Complexity Number (CCN) which is the number of possible execution paths through -typically- a function. The CCN of a program as a metric is useless on its own, however it can be used as a guideline for a programmer when beginning their unit testing phase. An ideal unit test will have as many tests as the CCN of the code – one test for every possible branch.

A programmer's commit history may be seen as a suitable metric of progress through an assignment or program, however like SLOC, it is particularly easy to exploit. Smaller but more frequent commits give the impression of progressing through a program but can be as frequent as the user desires and may not give a relevant commit comment to see if the code submitted is worthwhile. The Linux Kernel Documentation describes the average commit comment as 50 characters long however we can see in Figure 2 that the comments tend to be shorter than 50 characters. Combining the ambiguous nature of commit comments, and the irregular length of git commits, it is impossible to find the quality of the work being committed. An important metric that can be obtained from git commits, is the activity of the programmer over a period. An active programmer will commit to a project over many days before completion, as opposed to a single and final commit. Each of these commits can be measured in length to find the average SLOC

written in a day. While not a perfect measure of the quality of code, it is a good metric for the progress and productivity of a programmer as it shows continuous and constant work.

### Algorithmic Approaches

$n_1 =$ number of distinct operators,	$n_2 =$ number of distinct operands
$N_1 =$ total number of operators,	$N_2 =$ total number of operands
Program Vocabulary: $n = n_1 + n_2$	Program Length: $N = N_1 + N_2$
Volume: $V = N * \log_2 N$	Difficulty: $D = n_1/n_2 * N_1/N_2$
Effort: $E = D * V$	Time to Program: $T = E/18(\text{sec})$

The issue of readability is not unique to commit comments. Halstead Complexity Measures are an attempt to quantify the readability of code, using various calculations. These calculations can give a look in the readability of code – fewer unique operators and operands means fewer elements to understand. While documentation does aid a programmer reading code that is not clear, the time

taken to read and then apply that understanding would not have been necessary had the code been clear initially. Beyond making the code easier to understand for the programmer, the readability of code plays into the technical debt of a program, the implied cost of adding or changing a program due to poor decisions made in the initial coding phase. Technical debt occurs in many situations in which poor technical decisions are made, of which includes poor readability and documentation. Insufficient planning and last-minute changes play a huge role in slowing the development of a

program when readability is poor. Programmers lack the fluidity to be capable of making quick changes due to lack of understanding.

The final measurable metric one might use to find a program's overall

### **Alternative Development Environments**

Test Driven Development (TDD) is an alternative method of designing and writing code. TDD is a test focused approach which relies on a short development cycle – a initial test is written that defines the desired outcome of the program, the test is run and likely fails, the code updated, repeated until test passes. It was created in response to ad hoc testing which “usually leads to last minute or even no testing at all”<sup>4</sup>. A study carried out by Erdogmus, Morisio, and Torchiano found TDD allowed programmers “better task understanding” and “better task focus” but tends to “increase the variation in productivity”<sup>5</sup>. These papers support the idea that TDD increases the morale of the programmers but doesn't reflect in the final quality of the code, see Figure 1. This may lead us to believe that tests are not a viable way to track one's progress through code., however it is a good indicator of the quality of code, more passing test means more functional code.

### **Ethics of Measuring.**

### **Conclusion.**

## References

1. Goodman, P. (2004). *Software Metrics: Best Practices for Successful IT Management*. Rothstein Associates Inc.
2. Boehm, B., & Abts, C. (2000). Software Development Cost Estimation Approaches – A Survey. *IBM Research*, 44.
3. Binstock, A. (2012). Measuring Complexity Correctly. *Architecture and Design*
4. Maximilien, E., & Williams, L. (2003). Assessing Test-Driver Development at IBM. *IBM Research*, 6.
5. Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 12.
6. Linux. (n.d.). *Linux Kernel Documentation*. Retrieved from Kernel.org:  
<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/process/submitting-patches.rst?id=bc7938deaca7f474918c41a0372a410049bd4e13#n664>
- 7.

Figures

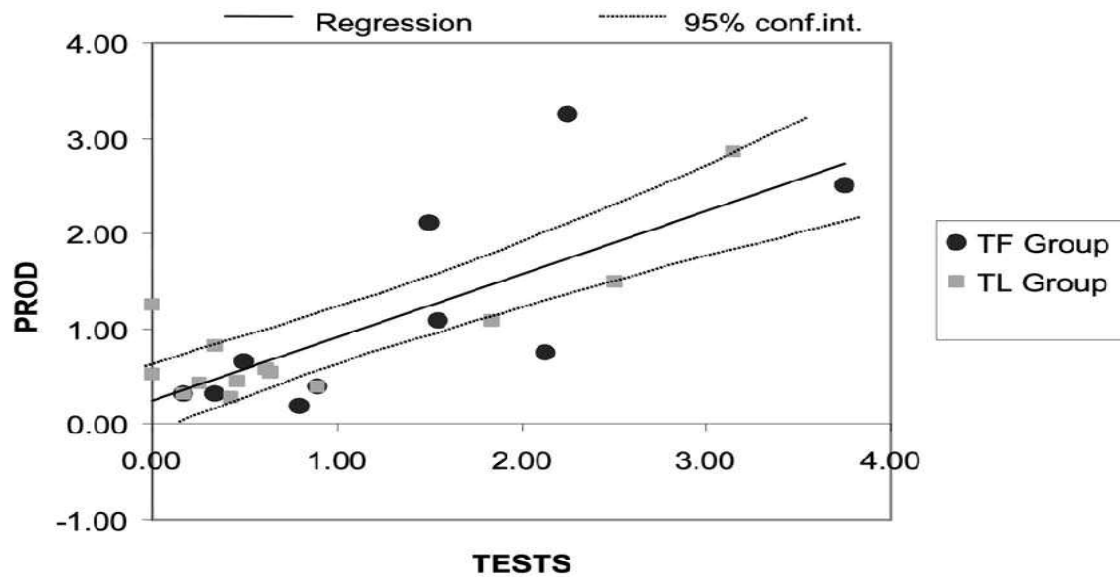


Figure 1 Productivity Against Tests, shows the change in productivity of programmers in a test first, vs test later environment.

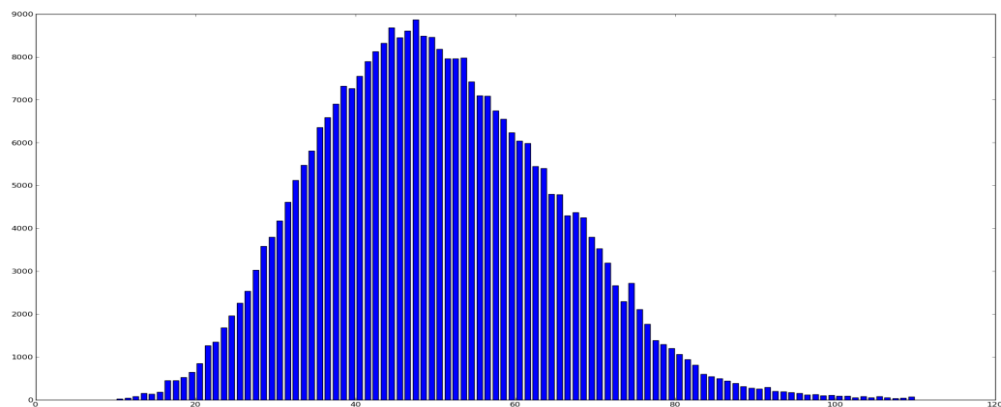


Figure 2 Length of Commit Comments<sup>7</sup>