Measuring Productivity Using the Infamous Lines of Code Metric

Benedikt Mas y Parareda, Markus Pizka
itestra GmbH
Ludwigstrasse 35, 86916 Kaufering, Germany
mas@itestra.de

Abstract

Nowadays, software must be developed at an ever-increasing rate and, at the same time, a low defect count has to be accomplished. To improve in both aspects, an objective and fair benchmark for the productivity of software development projects is inevitably needed.

Lines of Code was one of the first widely used metrics for the size of software systems and the productivity of programmers. Due to inherent shortcomings, a naive measurement of Lines of Code does not yield satisfying results. However, by combining Lines of Code with knowledge about the redundancy contained in every software system and regarding total projects costs, the metric becomes viable and powerful.

The metric "Redundancy-free Source Lines of Code per Effort" is very hard to fake as well as objective and easy to measure. In combination with a second metric, the "Defects per Source Lines of Code", a fair benchmark for the productivity of software development teams is available.

1. The need to measure

The ability to produce innovative software at a high rate is of utmost importance for software development companies in order to persist in a competitive and fast moving market. At the same time, with increasing dependence of business processes on software, the ability to deliver high-quality software becomes crucial for economic success [9].

We define *productivity* as the ratio of the size of the output versus consumed input, i. e. the effort required to produce one unit of output. Applied on a coarse level to an entire software development project, this definition retrospectively describes the performance of the development effort.

Achieving high productivity is not easy and maintaining it requires constant attention. The economic incentive for improvement is enormous, as advancing the productivity not only increases the profit margin of individual projects, but also allows to implement more projects at the same time.

A prerequisite to manage and improve productivity is the ability to measure and compare it against industry standards and internal benchmarks. All processes that exert influence on productivity need to be appraised in order to identify potential for improvement that can lead to optimal performance.

Factors that might impair or advance development productivity range from external influences such as the temperature in office spaces over the motivation of developers to tricky technical challenges [11] [13]. However, the individual examination of all these factors is virtually impossible in commercial environments. Therefore, we are interested in a productivity metric that concludes the effects in a single assessment.

Counting *Lines of Code* is one of the oldest and most widely used software metrics [13] [7] to asses the size of a software system. It has been argued repeatedly that this metric does not adequately capture the complexity of software systems or the development process. Hence, it is considered wrong to rely on Lines of Code for appraising the productivity of developers or the complexity of a development [5] [11] project.

We will show that by excluding redundant parts of code from the Lines of Code count, combining the result with the total effort needed and the defect rate in the outcome, a highly objective and efficiently measurable productivity benchmark for software development projects is obtained.

2 Related work

We are well aware of the controversy that our suggestion to use some kind of Lines of Code as a foundation for measuring productivity might cause. Advantages and disadvantages of Lines of Code have been discussed in great detail by various authors [13] [7] [11]; a summary of this disussion can be found on the Internet [4]. Some authors even went as far as comparing Lines of Code to a weight-based metric that is based on the paper printout of a system [5].

Capers Jones explains in detail how the measurement of Lines of Code might result in apparently erratic productiv-

ity rates (see section 5.1.2). Therefore, instead of using Lines of Code, he suggests using function point based metrics [11].

To our knowledge, no attempt has been made yet to combine redundancy metrics [6] with Lines of Code to assemble a more suitable definition of the relevant size of a software product.

3 Basics of the productivity metric

A desirable property of a software productivity metric is the ability to compare a large variety of different systems. Furthermore, the computation of the metric must only cause low costs. This leads to four key requirements:

- The metric should be applicable to many different programming languages.
- The effort (cost and time) to perform a measurement should be low.
- The metric has to be objective and repeatable. This means that an assessment of the same system executed by different individuals has to produce identical results.
- Errors in measurement should be ruled out.

This indicates the use of metrics that can be assessed automatically by tools at least to a significant extent. We suggest to combine the following key performance indicators into a productivity metric:

Redundancy The redundancy of a software system describes the amount of code that is dispensable, i. e. the parts of code that are semantically duplicated or simply unused.

Source Lines of Code A count of relevant Source Lines of Code (SLOC) of a system, using a standard definition. Various extensive and precise definitions of Lines of Code are available [8] [12] [14].

RFSLOC By ignoring redundant parts of a system in the SLOC count, we obtain the number of Redundancy-free Source Lines of Code (RFSLOC).

Defect count Defects include software bugs as well as failures to implement required functionality.

Man-days The total count of the man-days (MD) a software development took or takes from setup until delivery.

To achieve fair results, these performance indicators have to be examined *after completion* of a software development project. They are combined into two metrics.

The composed *Redundancy-free Source Lines of Code per Man-day* metric,

`RFSLOC/MD`

represents a simple yet viable measure for the productivity of a software project. The second metric, the *Defects per Redundancy-free Source Lines of Code*

`defects/RFSLOC`

is an important indicator for the quality of the outcome of the project. Obviously, when judging productivity using RFSLOC/MD, the defect rate has to be considered, too.

4 Performance indicators

In order to ensure fairness and comparability, the four base indicators used in the composed productivity and quality metrics have to explained in greater depth.

4.1 Redundancy-free Source Lines of Code

The precise definition of Source Lines of Code is largely irrelevant to the productivity metric, as long as the same definition is reused for projects that are compared. However, the Source Lines of code count should *not* include the following classes of code:

- Re-used code (such as external libraries)
- Test code; test code is not part of the final product
- Redundant code (see 4.1.1)
- Dead code, i. e. unreachable code
- Generated code (see below)

A standard code formatting has to be applied to the source code before the counting takes place.

As the intricate differences between programming languages make a uniform formatting for all languages impossible, the definition has to be extended by language-specific code formatting rules. If need be, normative factors can be defined to diminish differences in formatting between programming languages.

Particular attention has to be paid to generated code. Generating code is usually much faster than writing code. Hence, if this type of code would be included in the metric, the productivity of projects using code generation would be elevated artificially. To prevent this, we omit generated code. Instead, we include the size of the configuration and the input to the code generator, thereby regarding all human achievements as part of productivity.

One advantage of regarding a variation of Lines of Code is that the assessment of the metric can be performed at very little cost. For almost virtually any programming language counting Lines of Code is easy. David A. Wheeler's script *SLOCCount* [3] alone is capable of counting source lines of code for approximately 25 different programming languages.

4.1.1 Redundancy

By copying code instead of re-using it, the volume of a system increases significantly while the functionality gets hardly extended. Besides, the changeability of a software system will even suffer.

In order to take this effect that could have a strong impact on the productivity metric into account, we always rely on Redundancy-free Source Lines of Code – lines that do not contribute to the functionality of the system are excluded. As a consequence, it becomes very difficult to fake the productivity metric. The most effective way to fake productivity measures by copying code lines is ruled out.

Redundancy is defined on two layers:

- Two code snippets are *syntactically redundant* if they are syntactically similar with respect to some definition of edit distance. Such textual copies are most frequently produced if developers fail to recognize a possibility for re-use or are unable to implement the necessary abstractions and instead use copy and paste.
- The definition of redundancy can be extended to *semantically redundant*. Two pieces of code that implement a similar behavior with respect to a given narrow defintion of similarity are considered semantically redundant.

The vast amount of redundancy contained in a software system can be detected or at least estimated with specialized clone analysis tools such as ConQAT [2]. To ensure fairness, all assessments have to be carried out using a standard tool configuration.

4.2 Defect count

It is generally accepted that producing error-free code is considerably more expensive than implementing "quick-and-dirty" solutions that possibly contain a substantial amount of bugs. To ensure fairness of the metric, only projects that display a similar defect rate can be compared with each other.

4.3 Man-days

The count of man-days includes all effort put into completion of the project. This includes activities such as administrative tasks, requirements analysis, implementation or testing. Overtime and unpaid work should also be considered.

The inclusion of activies that are not directly related to the implementation means that the efficiency of e.g. project management will be reflected in the result. This behavior of the proposed productiviy metric is absolutely desirable as all tasks performed within the development project contribute to its (economic) success or failure.

5 Justification

> Measuring programming progress by Lines of Code is like measuring aircraft building progress by weight. *Bill Gates* [1]

For various reasons, Lines of Code are regarded as delivering wrong and misleading results for measuring the productivity of developers. If applied to the comparison of systems written in different programming languages, it is even considered to be "professional malpractice" [11].

We will now justify how the proposed metric compensates many shortcomings of Lines of Code. While certain limitations to the applicability will remain, we believe that these are of limited importance for practical application in commercial environments.

5.1 Compensating LOC disadvantages

The following sections are a summary of the most commonly cited disadvantages of Lines of Code-based metrics from various sources [4] [11] [5]. The first paragraph of each section describes the shortcoming, the paragraphs below explain our counter-arguments.

5.1.1 Lack of accountability

> The implementation phase of a software project makes up for only one third of the overall effort of a software development project. Besides, the implementation is only one of many results. Hence, measuring the productivity of a project by Lines of Code means ignoring the bigger part of the effort.

The development of any non-trivial software system requires certainly a complicated process that creates several additional results such as a user manual and design documentation. While these artefacts are valuable in their own right, the main outcome of the project is still executable code,; i. e. the implementation. Additionally, the investment in e. g. requirements analysis and system design is usually performed because these activities are necessary for a successful completion of the project in the first place and they increase overall productivity and software quality:

- An increased effort in system design will reduce the effort required for implementation.
- A well-designed and engineered system will be implemented with less defects per RFSLOC. As the fixing of defects is costly, producing less defects initially will increase the overall productivity rate of a software development project [11].

Therefore, assessing the productivity of a project by the resulting system itself already includes intermediate results apart from the implementation. In order to incorporate the complexity of all contributing activities, it is even essential to include all efforts of all activities and not only the effort put into the implementation phase.

The volume of supplemental results that were possibly required by the customer of the project is not included in the assessment, as our definition of productivity is focused on the development of software without possibly additionally requested byproducts.

However, e. g. the requirement to produce extensive documentation justifies some reduction in productivity.

5.1.2 Lack of correlation with functionality

> Different programming languages are different in their verbosity, the statements offered by the language and, most importantly, the functionality displayed by a fixed amount of lines.
>
> If two systems written in different programming languages exhibit identical functionality, they will still be expressed in a different number of Lines of Code. Due to the diseconomy of scales, this might lead to confusing results. Consider the following example [11]:
>
> A program written in Assembler requires 1,000,000 SLOC to implement and implementation takes 10,000 days. Implementing the same program in C takes 6,250 days and requires 500,000 SLOC. The Assembler version of the program is obiously the economically worst option, as the total cost are higher. However, if the metric RFSLOC/MD is calculated, the Assembler version will yield a ratio of 100 while the C version will only yield 80 SLOC per man-day.

In contrast to Jones, we argue that the example supports the correctness of the proposed metric and that the result is exactly what is expected from a productivity metric.

It is generally accepted that project size and effort do not scale equally; big projects require a disproportional effort due to increased complexity and are less likely to conclude successfully than smaller projects [10]. Hence, being double in size, the Assembler project can be expected to require a proportionally stronger effort than the C project. Consequently, if the bigger project is able to produce more redundancy-free source lines of code per man-day than the smaller project and achieves a similar defect count (a prerequisite for the application of the productivity metric), the productivity of this project has to be rated higher. This holds true even if one considers that the total cost of that project is higher.

5.1.3 Lack of correlation with effort

> For various implementation tasks, effort and lines of code do not correlate well. Activities such as bugfixing usually require great effort, but do not add a significant number of lines. Hence, while these tasks are necessary for the completion of functionality, their valuable contribution is not reflected in the metric. This shows that the metric occasionally delivers wrongful results.

The reason for the failure of the metric is the restriction of the assessment to a particular activity or time frame within a project. The assessment will only be fair if performed on the total cost of a completed project.

Depending on the development process, different activities are required in different phases of the project. As each activity influences the lines of code count differently, it is impossible to assess productivity at an arbitrary point in time or to compare the productivity of individual activities. On completion of the project, the individual tasks are not carrying weight any more, as they are subsumed in the total effort of the project, and the metric will be correct, showing a correlation with effort.

5.1.4 Code verbosity

> Skilled development teams are able to develop the same functionality with less code than less skilled development teams. In a metric based on lines of code, the more skilled development teams will come off worse than less skilled development teams.

While we agree that a theoretical possibility for such misleading results exists, we believe that they are very unlikely to occur in real world development projects, because:

- Verbose code becomes redundant quickly. Hence, large parts of the overly verbose code do usually not affect the RFSLOC count.
- Skilled development teams will produce code that contains fewer bugs than unskilled development teams and will spend less time on redoing and undoing work.

- Skilled development teams will more likely fulfill the requirements, resulting in a lower defect count and receive less productivity penalties on defects.
- Concise code is easier to understand than verbose code. Therefore, expensive activities such as fixing (inevitable) bugs will be cheaper in the less verbose code base. This will affect productivity during development, making it unlikely that less skilled teams achieve the same productivity rate if it is measured on the total cost of a project.

5.1.5 Generated code

> As generated code is excluded from the Lines of Code count, the metric might display undesirable behaviour: if project A uses code generation for developing a certain functionality and project B is developing the same functionality without using code generation, it is likely that project B will be more expensive in total and take longer. But as project A produces fewer lines of code, the project might achieve an almost identical rate of RFSLOC/MD.

Similar to the code verbosity problem, we consider this scenario to be very unlikely in reality. If significant parts of a system are generated, an alternative manual implementation will also be of significant size. Hence, project B would be significantly larger than project A. As explained in section 5.1.2, due to the diseconomy of scales this makes it very unlikely that project B will achieve a similar or even higher level of productivity than project A.

Additionally, it can be assumed that fixing bugs in the generated code requires less effort than in the manually written code. First, there will hardly be any bugs in the generated code with a reasonable generator. Second, reconfiguration of the code generator can be expected to be less extensive than changing the manual implementation. The time needed to detect and fix bugs in the manual implementation decreases its productivity.

5.1.6 Project complexity and comparability

> In general, different projects can not be compared to each other due to various external influences and characteristics that severly affect productivity. These include:
>
> - The particular customer caused high management overhead.
> - The implementation technology is new.
> - The project domain is particularly complex.
> - The project is using a certain development process.
> - Volatile requirements cause frequent change.

We consider these characteristics to be typical and common challenges for the management of software development projects that have to be dealt with as part of the development process. If these factors affect productivity by increasing the total cost of a project, this is also considered part of the outcome of a project and should therefore be reflected in the metric.

5.2 Limitations

The proposed metric compensates many of the commonly cited shortcomings of Lines of Code-based metrics. However, limitations to the comparability between projects remain.

5.2.1 Non-functional requirements

Development efforts often have to deal with non-funtional requirements such as:

- Performance requirements
- Security requirements
- Availability and reliability requirements

Non-functional requirements have a considerable impact on the complexity of a development project. The development of a critical system with high performance requirements is not comparable to the development of a less demanding system, as increasing the performance of a system is expensive and laborous.

This is not reflected adequately in the RFSLOC count and less demanding projects will generally perform better. Therefore, only the comparison of systems with similar non-functional requirements is valid.

6 Conclusion

The proposed productivity metric has several advantages over alternative assessment methods:

- The metric can be measured using tools.
- The metric is very hard to fake.
- RFSLOC are universally applicable: any programming language as of today is based upon code. Even visual tools result in code that requires compiling.
- The metric is fair, if a few limitations are observed.
- Effects of the environment, such as e. g. ineffective tools and management, are reflected in the metric.

We were able to employ the proposed metric in the assessment of the application portfolio of a large industrial partner, examining systems comprising more than 20 million lines of code in total. The results of the experience showed that productivity can be measured effectively using Lines of Code, redundancy and the defect count. Additionally, we could ascertain that possible failures of the metric (such as described in 5.1.2, 5.1.4, 5.1.5) are of a low probability for all practical purposes.

7 Future work

The productivity metric has to be refined to become universally applicable. A major gap is the comparison of projects of different size. To allow the comparison of projects of random size, the productivtity rate should be normalized according to the size of each project. However, at the moment it is unclear how this normalization could take place.

7.1 Maintenance projects

More promptly, the metric will be extended to cover the efficiency of software maintenance services. Maintenance can be conducted successfully without increasing the lines of code count but greatly advancing the functionality of a system. Obviously, the absolute count of lines of code can not be used any more.

Key to evaluate the efficiency of maintenance could be a count of the lines that were added, removed or changed. These lines can be counted easily through clone detection: all lines that were *not* changed will be detected as clones in a clone assessment that compares the resulting system with itself before the maintenance activities.

As maintenance is an entirely different task from the green-field development of a system, this metric can not be compared to the results found for development projects. New experimental applications to maintenance projects are needed and the metric will probably need to be refined to address the particularities of maintenance activities.

References

[1] Best programming quotations. World Wide Web, Aug. 2007. http://www.linfo.org/q_programming.html.

[2] Conqat - continuous quality assessment toolkit. World Wide Web, Aug. 2007. http://conqat.cs.tum.edu/.

[3] Sloccount. World Wide Web, Aug. 2007. http://www.dwheeler.com/sloccount/.

[4] Wikipedia: Source lines of code. World Wide Web, Aug. 2007. http://en.wikipedia.org/wiki/Source_lines_of_code.

[5] P. G. Armour. Beware of counting loc. *Communications of the ACM*, 47(3):21–24, 2004.

[6] B. S. Baker. On finding duplication and near-duplication in large software systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[7] B. W. Boehm. *Software Engineering Economics*. Advances in Computing Science & Technology. Prentice-Hall, Englewood Cliffs , NJ , USA, Dec. 1981.

[8] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[9] W. E. Deming. *Out of the Crisis*. The MIT Press, 1986.

[10] T. S. G. I. Inc. Chaos: A recipe for success, 1999.

[11] C. Jones. *Software Assessments,Benchmarks and Best Practises*. Information Technology Series. Addison Wesley, 2000.

[12] R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Carnegie Mellon University, Sept. 1992.

[13] C. E. Walston and C. P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, 1977.

[14] D. A. Wheeler. More than a gigabuck: Estimating gnu/linux's size. World Wide Web, July 2002.