Method for Determining the Number of Lines of Manually Written Source Code

Tetiana Hovorushchenko[a], Yurii Voichur[a]

[a] *Khmelnytskyi National University, Institutska str., 11, Khmelnytskyi, 29016, Ukraine*

Abstract

Improving the veracity of estimating the effort of software development is currently an urgent task. The purpose of this study is to develop a method for determining the number of lines of manually written source code to calculate the actual number of lines of code created by the programmer(s), which will allow a more accurate and reliable assessment of the effort of software development. The developed in this article method for determining the number of lines of manually written source code makes it possible to determine not only the LOC-estimation (number of code lines) of the source code written by a programmer, but also the LOC-estimation (number of code lines) of automatically generated source code, as well as the number of uses of each automatically generated construction of a particular programming language. The method is universal because it can be customized for any used programming language.

Keywords

Software development effort (labor intensity), LOC-estimation, source code, manually written source code, automatically generated source code.

1. Introduction

Analyzing the software development process and estimating the effort required to complete it is an important task. An early estimate of effort (labor intensity) is important for effective planning of resource utilization in a software project [1, 2].

In today's highly competitive market, it is very important for a software project development team to ensure timely delivery of their software product and stay within the planned budget, but in practice they are constantly faced with schedule delays and budget overruns [3].

Estimating the effort of a software project is an important process that involves predicting how much time and money it will take to complete a software development project. For both clients and developers in software development, effort estimation is vital. Underestimating the effort can lead to poorly designed processes, low quality, delayed schedules and budget overruns, inadequate project approval by management and customers, insufficient project team size, excessively tight development timelines, and, as a result, reputational damage and loss of trust in developers in the event of budget and schedule violations. Moreover, overestimating the complexity of software development may not be any better. If more resources are allocated to a project than are actually needed, the software project will be more expensive and time-consuming, and will result in a delay in the start of the next project or in its refusal to the software development. It is effort estimation that helps to exchange information necessary for the successful achievement of project results [4].

The success of a software project includes three main elements: time, budget, and functionality. When changes are made to one of the elements, the other elements will necessarily change as well, and the nature of the impact of such changes depends mainly on the specifics of the project and the

ITTAP'2023: 3rd International Workshop on Information Technologies: Theoretical and Applied Problems, November 22–24, 2023, Ternopil, Ukraine, Opole, Poland
EMAIL: tat_yana@ukr.net (T. Hovorushchenko); voichury@khmnu.edu.ua (Yu. Voichur)
ORCID: 0000-0002-7942-1857 (T. Hovorushchenko); 0000-0003-3085-7315 (Yu. Voichur)

circumstances. For example, a reduction in project execution time may in one case lead to a decrease in its budget due to a reduction in functionality, and in another case to an increase in its budget due to the involvement of more developers to maintain the planned functionality [5-9].

Estimation of effort is used to solve many problems, including the following [3, 10]: development of a budget and schedule of a software project; analysis of the degree of risk and selection of a compromise solution; planning and management of a software project; analysis of the costs of improving the quality of software.

Estimating effort project remains a difficult task for project managers. In the early stages of the project, a high level of uncertainty and lack of experience lead to an inaccurate estimate of effort [11]. The rapid growth of the software industry drives the need for new technologies to improve the accuracy of software effort estimation methodologies.

Thus, improving the veracity of software development effort estimation is *an urgent task*.

2. Case Study

One of the main methods for estimating the effort of a software project is algorithmic modeling [12-16], which is a method that determines the dependence of the project effort on some quantitative indicator of the software (usually the size of the code). This indicator is estimated for a given project, after which the model predicts future costs. Most models for determining the complexity of software development can be reduced to a function of five main parameters [17-20]:

1. the size of the final product – usually the number of lines of code or the number of function points required to implement a given functionality
2. features of the process used to obtain the final product, in particular, its ability to avoid unproductive activities (rework, interaction costs)
3. capabilities of the personnel involved in software development, especially their professional experience and knowledge of the project subject area
4. the environment, which consists of tools and methods used to effectively perform software development and automate the process
5. the required quality of the product, which includes its functionality, performance, reliability and adaptability.

The most influential factor in effort estimating in these models is the size of the software [21-24]. The main units of measurement of software size are: the number of lines of code (LOC) and function points (FP). The number of lines of code (LOC-estimation) is the most famous, widespread, and most used unit of measurement [4, 11, 25, 26].

The advantages of using LOC-estimations as units of measurement [4, 11, 25]: widespread and easy adaptability; the ability to compare methods of measuring size and performance in different groups of developers; direct connection with the final product; easy evaluation before the end of the project; estimation of software size based on the developer's point of view - a physical assessment of the created product.

Along with the advantages, the use of LOC has a number of problems [4, 11, 25]: LOC-estimation is difficult to use when estimating the size of software at the early stages of development; lines of source code may differ depending on the types of programming languages, design methods, style and abilities of the programmer; LOC-indicators cannot be used for normalization if the development platforms or languages used are different; the use of estimation methods by counting the number of lines of code is not regulated by industry standards; software development can be associated with high costs that do not directly depend on the size of the source code – preparation of requirements specifications and user documents that are not included in the direct costs of coding; programmers may be undeservedly rewarded for achieving high LOC if management mistakenly considers it a sign of high productivity, but there is no carefully designed project (source code is not an end in itself when creating a product – functional properties and performance indicators play a major role); code generators often produce an excessive amount of code, which distorts LOC-estimations; when counting the number of lines of code, should distinguish between automatically and manually generated code.

Most development environments include sets of standard elements. Whenever a new project is created, the respective development environment automatically creates the "skeleton" of the future application, and this code can be immediately compiled and run without errors. In this case, the software project contains automatically generated source code, which is the basis of the future program. More complex controls have so-called "wizards" that help you customize the behavior of controls by automatically generating code depending on the selected options. Automatic generation of source code saves developers' time, eliminates the need to re-create a typical source code every time and, of course, reduces the effort of software development. It is for the purpose of correctly determining the LOC-estimation of the source code and further reliable estimation of the effort of software development that it becomes necessary to distinguish between automatically generated source code and manually written (by a programmer(s)) source code.

Therefore, *the purpose of our study* is to develop a method for determining the number of lines of manually written source code to calculate the actual number of lines of code created by the programmer, which will allow a more accurate and reliable assessment of the effort of software development.

3. Method for Determining the Number of Lines of Manually Written Source Code

Since a software project contains automatically generated source code, which is the basis ("skeleton") of the future program, and source code that is added manually by the programmer(s), any source code can be represented as a union of sets of automatically generated source code and source code written manually by the programmer(s):

$$C = AGC \cup MWC, \tag{1}$$

where AGC is the automatically generated source code, MWC is the manually written source code.

Therefore, the LOC-estimation of the source code can be represented in the form:

$$LOC = LOC_{AG} + LOC_{MW}, \tag{2}$$

where LOC_{AG} is the LOC-estimation of the automatically generated source code, LOC_{MW} is the LOC-estimation of the manually written source code.

The overall LOC-estimation LOC can be obtained using a variety of static analyzers and other tools that provide metric analysis of the source code.

For determining the number of lines of manually written source code, the LOC-estimation of automatically generated source code is determined. Lines of automatically generated code consist of certain constructions that form the following set (this set will have different content for different languages and programming environments):

$$AG = \{constr_1, \dots, constr_m\}, \tag{3}$$

where $constr_j$ is *j*-th automatically generated source code's construction for a particular programming language under consideration (the main restriction on such a construction is that it must be a single line of code), *m* is the number of constructions that can be automatically generated (this number is different for different environments and programming languages).

The entire source code can also be represented as a set of its lines:

$$C = \{line_1, \dots, line_{LOC}\}, \tag{4}$$

where $line_i$ is *i*-th line of source code, moreover $line_i$ is, in turn, also a set consisting of the constructions of a particular programming language, letters, numbers and symbols allowed by the alphabet of a particular programming language. The main requirement for a source code is mandatory compliance with the Code Style rules, in particular, in terms of code formatting (a new construction is written on a new line, etc.).

The universal *rules for estimating the number of lines of automatically generated source code* were developed:

1. if $constr_1 \in line_1$, then $LOC_{AG1} = LOC_{AG1} + 1$
2. …
3. if $constr_j \in line_1$ $(j = 1..m)$, then $LOC_{AGj} = LOC_{AGj} + 1$
4. …
5. if $constr_m \in line_1$, then $LOC_{AGm} = LOC_{AGm} + 1$
6. …
7. if $constr_1 \in line_i$, then $LOC_{AG1} = LOC_{AG1} + 1$
8. …

9. if $constr_j \in line_i$ ($j = 1..m,\ i = 1..LOC$), then $LOC_{AGj} = LOC_{AGj} + 1$
10. …
11. if $constr_m \in line_i$, then $LOC_{AGm} = LOC_{AGm} + 1$
12. …
13. if $constr_1 \in line_{LOC}$, then $LOC_{AG1} = LOC_{AG1} + 1$
14. …
15. if $constr_j \in line_{LOC}$ ($j = 1..m$), then $LOC_{AGj} = LOC_{AGj} + 1$
16. …
17. if $constr_m \in line_{LOC}$, then $LOC_{AGm} = LOC_{AGm} + 1$

The developed rules are universal for each language and programming environment, but the content of the set $AG = \{constr_1, \ldots, constr_m\}$ will be different (individual) for each specific language and programming environment.

Method for determining the number of lines of manually written source code consists of the following steps:

1. to form a set of automatically generated constructions $AG_{lang} = \{constr_{1\_lang}, \ldots, constr_{m\_lang}\}$ for the programming language *lang*
2. using the method of searching in width in the forward direction in the set of rules for estimating the number of lines of automatically generated source code, to search the rules for each of the elements of the set $AG_{lang} = \{constr_{1\_lang}, \ldots, constr_{m_{lang}}\}$ in the set $C = \{line_1, \ldots, line_{LOC}\}$, according to which the counters $LOC_{AG1}, \ldots, LOC_{AGm}$ of the number of lines (used in the source code) of each automatically generated construction of the programming language *lang*
3. to determine the LOC-estimation of the automatically generated source code by the formula:
$$LOC_{AG} = \sum_{j=1}^{m} LOC_{AGj}, \tag{5}$$
where LOC_{AGj} is the number of uses in the source code of each automatically generated construction of a particular programming language
4. to determine the LOC-estimation of manually written source code using the formula derived from formula (2):
$$LOC_{MW} = LOC - LOC_{AG}, \tag{6}$$
where LOC is the total number of lines of source code (as mentioned earlier, there are many tools that provide LOC-estimation of source code)

The developed method for determining the number of lines of manually written source code is scientifically new and makes it possible to determine not only the LOC-estimation of the source code written by a programmer(s), but also the LOC-estimation of automatically generated source code, as well as the number of uses of each automatically generated construction of a particular programming language. The method is universal because it can be customized for any used programming language.

4. Results & Discussion

A real case of using the developed method for determining the number of lines of manually written source code using the Visual C++ programming language as an example was considered. The analyzed Visual C++ source code, which consists of 225 lines, is represented as a set $C = \{line_1, \ldots, line_{225}\}$.

First, a subset of the automatically generated constructions was formed $AG_{VisualC++} = \{\#include, \#define, hinstance, wchar, atom, bool, lresult, int\_ptr, LoadStringW, unreferenced\_parameter, MyRegisterClass,\}$ for the Visual C++ programming language (this subset for the example under consideration does not include all automatically generated Visual C++ constructs, but only 11 such constructs, which is sufficient to demonstrate the operation of the proposed method).

Using the method of searching in width in the forward direction in the set of rules for estimating the number of lines of automatically generated source code, we searched the rules for each of the elements of the set $AG_{VisualC++}$ in the set $C = \{line_1, \ldots, line_{225}\}$, according to which the counters of the number of lines (used in the source code) of each of the 11 automatically generated constructions of the Visual C++ programming language were counted: $LOC_{AG1} = 4$, $LOC_{AG2} = 5$, $LOC_{AG3} = 1$, $LOC_{AG4} = 10$, $LOC_{AG5} = 6$, $LOC_{AG7} = 10$, $LOC_{AG8} = 7$, $LOC_{AG9} = 13$, $LOC_{AG10} = 20$, $LOC_{AG11} = 8$.

Now the LOC-estimation of the automatically generated source code in the analyzed source code is determined:

$$LOC_{AG} = 4 + 5 + 1 + 10 + 6 + 10 + 7 + 13 + 20 + 8 = 84.$$

After that, the LOC-estimation of the manually written source code in the analyzed source code is determined:

$$LOC_{MW} = 225 - 84 = 141.$$

So, the analyzed Visual C++ source code, which consists of 225 lines, contains 84 lines of automatically generated code and 141 lines of manually written code.

5. Conclusions

Improving the veracity of estimating the effort of software development is currently an urgent task. The purpose of this study is to develop a method for determining the number of lines of manually written source code to calculate the actual number of lines of code created by the programmer(s), which will allow a more accurate and reliable assessment of the effort of software development.

The developed in this article method for determining the number of lines of manually written source code makes it possible to determine not only the LOC-estimation (number of code lines) of the source code written by a programmer, but also the LOC-estimation (number of code lines) of automatically generated source code, as well as the number of uses of each automatically generated construction of a particular programming language. The method is universal because it can be customized for any used programming language.

The directions of the authors' future research are: formation of complete sets of automatically generated constructions for different actual programming languages; automation of the analysis of source code in different languages to search for automatically generated constructions in it; development of a tool for determining the number of lines of manually written source code, which will work on the basis of the rules and method developed in this article.

6. References

[1] O. Pomorova, T. Hovorushchenko. The Way to Detection of Software Emergent Properties, in: Proceedings of the 2015 IEEE 8-th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications IDAACS-2015, Warsaw, 2015, vol. 2, pp. 779-784. doi: 10.1109/IDAACS.2015.7341409.

[2] O. Pomorova, T. Hovorushchenko. Research of Artificial Neural Network's Component of Software Quality Evaluation and Prediction Method, in: Proceedings of the 2011 IEEE 6-th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS-2011, Prague, 2011, vol.2, pp. 959-962. doi: 10.1109/IDAACS.2011.6072916.

[3] R. Arora, R. Mittal, A. G. Aggarwal. Investigating the impact of effort slippages in software development project. International Journal of System Assurance Engineering and Management 14 (2023) 878-893. doi: 10.1007/s13198-023-01887-3.

[4] N. Sreekanth, J. Rama Devi, K. A. Shukla, D. K. Mohanty, A. Srinivas, G. N. Rao, A. Alam, A. Gupta Evaluation of estimation in software development using deep learning-modified neural network. Applied Nanoscience 13 (2023) 2405-2417. doi: 10.1007/s13204-021-02204-9.

[5] T. Hai, J. Zhou, N. Li, S. K. Jain, S. Agrawal, I. B. Dhaou. Cloud-based bug tracking software defects analysis using deep learning. Journal of Cloud Computing 11 1 (2022), 32. doi: 10.1186/s13677-022-00311-8.

[6] L. Aversano, M. L. Bernardi, M. Cimitile, M. Iammarino, D. Montano. Forecasting technical debt evolution in software systems: an empirical study. Frontiers of Computer Science 17 3 (2023), 173210. doi: 10.1007/s11704-022-1541-7.

[7] T. Hovorushchenko, O. Pomorova. Methodology of Evaluating the Sufficiency of Information on Quality in the Software Requirements Specifications, in: Proceedings of 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies DeSSerT-2018, Kyiv, 2018, pp. 385-389. doi: 10.1109/DESSERT.2018.8409161.

[8] T. Hovorushchenko, O. Pomorova. Information Technology of Evaluating the Sufficiency of Information on Quality in the Software Requirements Specifications. CEUR-WS 2104 (2018) 555-570.

[9] T. Hovorushchenko T. Methodology of Evaluating the Sufficiency of Information for Software Quality Assessment According to ISO 25010. Journal of Information and Organizational Sciences 42 1 (2018) 63-85. doi: 10.31341/jios.42.1.4.

[10] P. G. F. Matsubara, I. Steinmacher, B. Gadelha, T. Conte. Much more than a prediction: Expert-based software effort estimation as a behavioral act. Empirical Software Engineering 28 4 (2023) 98. doi: 10.1007/s10664-023-10332-9.

[11] B. Şengüneş, N. Öztürk. An Artificial Neural Network Model for Project Effort Estimation. Systems 11 2 (2023) 91. doi: 10.3390/systems11020091.

[12] M. Padmaja, D. Haritha. Software Effort Estimation Using Grey Relational Analysis. International Journal of Information Technology and Computer Science 9 5 (2017) 52–60. doi: 10.5815/ijitcs.2017.05.07.

[13] S. Basri, N. Kama, H. Md Sarkan, S. Adli, F. Haneem. An Algorithmic-Based Change Effort Estimation Model for Software Development, in: Proceedings of 2016 23rd Asia-Pacific Software Engineering Conference APSEC-2016, Hamilton, 2016, 177-184. doi: 10.1109/apsec.2016.034.

[14] P. Singal, A. C. Kumari, P. Sharma. Estimation of Software Development Effort: A Differential Evolution Approach. Procedia Computer Science 167 (2020) 2643–2652. doi: 10.1016/j.procs.2020.03.343.

[15] R. Silhavy, Z. Prokopova, P. Silhavy. Algorithmic optimization method for effort estimation. Programming and Computer Software 42 3 (2016) 161–166. doi: 10.1134/s0361768816030087.

[16] Y. Mahmood, N. Kama, A. Azmi, A. S. Khan, M. Ali. Software effort estimation accuracy prediction of machine learning techniques: A systematic performance evaluation. Software: Practice and Experience 52 (2022) 39-65. doi: 10.1002/spe.3009.

[17] K. H. Kumar, K. Srinivas. An accurate analogy based software effort estimation using hybrid optimization and machine learning techniques. Multimedia Tools and Applications 82 (2023) 30463-30490. doi: 10.1007/s11042-023-14522-x.

[18] R. Lalitha, P. Sreelekha. A methodology to analyse and estimate software development process using machine learning techniques. International Journal of Software Engineering and Knowledge Engineering 33 6 (2023) 815-835. doi: 10.1142/s021819402350016x.

[19] H. Sone, Y. Tamura, S. Yamada. Study of Effort Calculation and Estimation in Open Source Projects. International Journal of Reliability, Quality and Safety Engineering 30 3 (2023) 2350011. doi: 10.1142/s0218539323500110.

[20] R. K. Gora, R. R. Sinha. A Study of Evaluation Measures for Software Effort Estimation Using Machine Learning. International Journal of Intelligent Systems and Applications in Engineering 11 6s (2023) 267–275.

[21] V. Yadav, R. Singh, V. Yadav. Estimation Model for Enhanced Predictive Object Point Metric in OO Software Size Estimation Using Deep Learning. The International Arab Journal of Information Technology 20 3 (2023) 293-302. doi: 10.34028/iajit/20/3/1.

[22] M. Jørgensen. Improved measurement of software development effort estimation bias. Information and Software Technology 157 (2023) 107157. doi: 10.1016/j.infsof.2023.107157.

[23] E. Rodríguez Sánchez, E. F. Vázquez Santacruz, H. Cervantes Maceda. Effort and Cost Estimation Using Decision Tree Techniques and Story Points in Agile Software Development. Mathematics 11 6 (2023) 1477. doi: 10.3390/math11061477.

[24] I. Abnane, A. Idri, I. Chlioui, A. Abran. Evaluating ensemble imputation in software effort estimation. Empirical Software Engineering 28 2 (2023) 56. doi: 10.1007/s10664-022-10260-0.

[25] S. Singh, K. Kumar Software Cost Estimation: A Literature Review and Current Trends, in: Proceedings of 2023 Third International Conference on Secure Cyber Computing and Communication ICSCCC-2023, Jalandhar, 2023, pp. 469-474. doi: 10.1109/icsccc58608.2023.10176495.

[26] X. Yuan, J. Su, C. Yu, S. Ye. Power Grid Software Cost Estimation Based on Improved COCOMO Model, in: Proceedings of 2023 IEEE 3rd International Conference on Electronic Technology, Communication and Information ICETCI-2023, Changchun, 2023, pp. 1265-1269. doi: 10.1109/icetci57876.2023.10176686.