Improving Measurement Certainty by Using Calibration to Find Systematic Measurement Error – A Case of Lines-of-Code Measure

Miroslaw Staron[1], Darko Durisic[2], and Rakesh Rana[1]

[1] Computer Science and Engineering, University of Gothenburg, Sweden
miroslaw.staron/rakesh.rana@gu.se,
[2] Volvo Car Group, Sweden
darko.durisic@volvocars.com

Abstract. Base measures such as the number of lines-of-code are often used to make predictions about such phenomena as project effort, product quality or maintenance effort. However, quite often we rely on the measurement instruments where the exact algorithm for calculating the value of the measure is not known. The objective of our research is to explore how we can increase the certainty of base measures in software engineering. We conduct a benchmarking study where we use four measurement instruments for lines-of-code measurement with unknown certainty to measure five code bases. Our results show that we can adjust the measurement values by as much as 20% knowing the systematic error of the tool. We conclude that calibrating the measurement instruments can significantly contribute to increased accuracy in measurement processes in software engineering. This will impact the accuracy of predictions (e.g. of effort in software projects) and therefore increase the cost-efficiency of software engineering processes.

1 Introduction

With the introduction of the measurement information model in the international ISO/IEC 15939 standard for measurement processes the discipline of software engineering evolved from discussing metrics in general to categorizing them into three categories – base measures, derived measures and indicators. The use of base measures is fundamental for the construction of derived measures and indicators. The base measures are also the types of measures which are collected directly and are a result of a measurement method. In many cases this measurement method is an automated algorithm (e.g. a script) which we can refer to as the *measurement instrument* which quantifies an attribute of interest into a number.

Since in software engineering we do not have reference measurement etalons as we do in other disciplines (e.g. kilogram or meter for physics), we often rely on arbitrary definitions of the base quantities. One of such quantities is the size of programs measured as the number of lines of code. Even though the number of lines of code of a given program is a deterministic and fully quantifiable

number the result of applying different measurement instruments to obtain the number might differ. The difference can be caused by a number of factors, such as: i) difference in implementation of the same measurement method, ii) difference in the defininition/design of the measurement method, or iii) faults in the measurement instruments. Since we do not know the true value without the measurement procedure we need to find what the accuracy (certainty) of the measured value is. When using measurement instruments it is often not possible to explore the measurement methods in details by analyzing the implementation of the measurement method (in practice the measurement instruments are provided as compiled code), which means that the measurement engineer needs to either accept the fact that the uncertainty is unknown or estimate the uncertainty. The latter leads to more benefits in terms of improved quality of the measurement results and therefore it is of interest for our work, which addresses the following research question:

How to reduce the uncertainty of measurement results obtained from measurement instruments with an unknown measurement method?

This research question is addressed by constructing a benchmarking study where we use four different measurement instruments which quantify the number of physical lines of code of a program. We use these measurement instruments on five different open-source code bases in order to explore the deviations between the results obtained from these tools. We also use a simple program with a known number of physical lines of code in order to estimate the systematic error of the tools and then we use that number (converted to a percentage) to reduce the error of the measurement obtained in the first step.

In the study we use the lines-of-code measure because of its practicality and availability of measurement instruments (both open source and proprietary) with unknown measurement method [1]. Another practical advantage is the fact that the lines-of-code measure is usually calculated using automated software tools which provide deterministic results and redefine the notion of the error for this measure. In other engineering disciplines the measurement tools are prone to systematic errors (related to calibration of the tools) and random errors (related to the measurement process). When using automated tools for measuring such quantities as LOC the measurement process always results in the same value when measuring the same entity – therefore seemingly without the random error. Calibrating the tools is sometimes difficult, which results in not being able to distinguish the systematic errors for measurement from the random ones. The measure has been both widely used in software engineering for calculating the size of programs [2], software complexity [3] or to predict software size [4]. The wide use of the measure has resulted in multiple variants of the definition – e.g. non-commented lines of code, total lines of code, or source statements. Although the different variants of the measure should not be mixed, it is often the case that these definitions are not recognizable in measurement tools (also known as measurement instruments) and thus result in measurement errors.

Our results show that using this simple approach we can reduce the uncertainty to 1.85% from 20% in some cases. We perceive this type of uncertainty reduction to be applicable to many other base measures (i.e. other than the lines-of-code measure used in this study) and have a potential to increase the validity of the prediction models using these base measures (e.g. COCOMO or defect density, [5]).

The paper is structured as follows. Section 2 outlines the most relevant work in the field. Section 3 presents the concept of the measurement error used in this paper. Section 4 presents the measure of lines of code. Section 5 presents the evaluation of the different definitions of errors on a set of open source programs. Section 6 provides recommendations on how to use the LOC measure in practice and section 7 presents the conclusions.

2 Related Work

We review work in three areas – standardization in the area of measurement in software engineering, measurement theory (in general and its applications in software engineering) and the overview of critical articles of measurement in software engineering.

Lincke et al. [6] studied the deviations of results from measurement tools for sizing object-oriented designs. Their results showed significant deviations between tools. Their work shows how important the notion of systematic and random measurement errors are and can to a large extent be explained by the work presented in our paper.

The recent advances in the field of metrology have also started to arrive in the field of software engineering. Abran [7] used a combination of the ISO VIM standard to discuss the validity of the most common metrics used in software engineering e.g. software complexity or the function points measures. Although the developments are recent, they are based on the previously defined needs for more precise terminology and validation of software metrics, e.g. [7]. One of the major current trends identified by Abran is the search for etalons – elementary units – in software engineering.

The basic concepts of measurement theory for software engineering have been redefined by Briand et al. [8] – for example the concepts of relational systems, mappings and scales. Similar to the definition of relational systems one can define properties of software measures, which are a foundation for defining a general measurement instrument model. Such properties are defined by Briand et al. [9] based on the general properties of measures defined by Weyuker [10], Zuse [11] and Tian and Zelkowitz [12]. Although the property-based definition of measures addresses the problem of correct definition of software metrics, it does not address the issues of uncertainty of the measurement process in practice (i.e. the instantiation of the mapping between the empirical world and the relational systems).

Measurement theory has been used as a basis for the main international standard in measurement on common vocabulary in metrology – VIM [13].

The standard defines such concepts as measurement uncertainty, measurand and quantification. These definitions capture the meaning of the concepts from the measurement theory in engineering.

VIM standardizes the most important concepts which influence measurement processes, for example:

- Measuring instrument: device used for making measurements, alone or in conjunction with supplementary device(s)
- Instrumental measurement uncertainty: component of measurement uncertainty arising from the measuring instrument or measuring system in use, and obtained by its calibration
- Measuring system: set of one or more measuring instruments and often other devices, including any reagent and supply, assembled and adapted to give measured quantity values within specified intervals for quantities of specified kinds

VIM does standardize the vocabulary, but the international standard ISO/IEC 15939:2007 [14] (Systems and Software Engineering - Measurement Processes) focuses on the processes of measuring – data collection, processing and analysis. The ISO/IEC 15939 standard has an impact on the definition of metrics and measurement guidelines for the ISO/IEC 25000 series of standards. However, none of these two standards addresses the actual problem of measurement errors.

3 The concept of measurement error

Measurement error is defined in measurement theory as the deviation between the real value of the measurand and the value obtained from the measurement process. It is derived from the concept of *measurement uncertainty* which is the dispersion of the values attributed to the measurand. The main definition is provided in ISO/IEC 17045 (General requirements for the competence of testing and calibration laboratories, [15]) and later on used in software engineering in ISO/IEC 25000 series of standards [16].

The measured quantity can be referred to as the *estimator* as the true value of the measurand always remains unknown because of the finite accuracy of the measurement instruments. In general the estimators (X) combine both the expected value of the measurement (M) plus the error (E) – see formula 1.

$$\hat{X} = \hat{M} + \hat{E} \tag{1}$$

The measurement error is combined of two types of errors – the systematic error (S) and the random error (R) – formula 2.

$$\hat{E} = \hat{S} + \hat{R} \tag{2}$$

The systematic error is usually caused by the miscalibration of the measurement instruments and is the same for all measurements taken by the instrument.

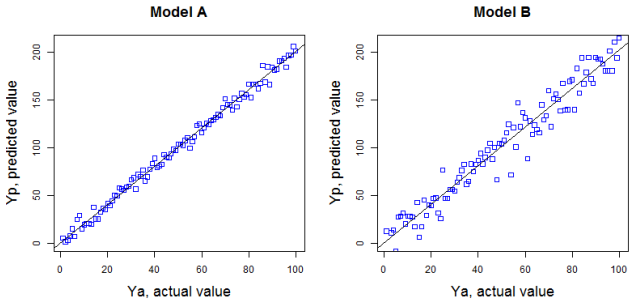

Fig. 1. Regression models with different accuracy of prediction

The mean of the systematic error is expected to be non-zero, and causes skewness of the measurement results. In order to minimize the systematic errors the measurement instruments are calibrated – adjusted to show the correct results when measuring entities of known properties.

The random error, on the other hand, is different for each measurement taken. The mean value of the random measurement error is expected to be zero and it causes the distribution of the measurement results to be wider.

In general it seems easy to distinguish between these two types of errors, but in practice it might be very difficult. In section 5 we show examples of this problem.

3.1 Impact of measurement error on predictions – Standard Error of the Estimate

Software measures are usually used within various models designed for monitoring and forecasting [17, 18]. When using these measures within prediction models, the other major source of errors come from estimation error. Every estimation method involves an estimation error, which comes from the simple fact that the real quantity generally differs from its estimated quantity.

Taking an example of simple prediction model with single predictor (x) and predicted variable (y) have a linear relationship between them ($y \sim m * x$, where m=2), two scenarios of prediction model can be shown as in Figure 1.

As evident from Figure 1, the prediction model A seems more accurate then prediction model B, or in other terms the estimation error is expected to be lower for model A compared to mode B.

The standard error of the estimate (or SEE) is the measure of accuracy of predictions. For estimations using linear regression minimizes the sum of squared deviations of prediction (also referred to as Sum of Squared Errors, or SSE). Thus the standard error of estimate for method of least squares (linear or non-linear models) is equal to SSE, which can be calculated as:

$$\sigma_{est} = \sqrt{\frac{\Sigma(Y_a - Y_p)^2}{N}} \tag{3}$$

Where σ_{est} is the standard error of estimate, Y_a is the actual value, Y_p is the predicted value, and N is the number of observations.

4 Lines of code (LOC)

In this paper we use the measure of Lines of Code (LOC) as an example to illustrate the concept of measurement error. The measure has been used in practice since 1950s and there is substantial body of research on it, [19]. It is also used as a input variable to many prediciction models – e.g. the Constructive Cost Model (COCOMO) and its newer versions [4].

LOC measure is often also called SLOC (Source Lines of Code) as an acronym and has multiple variations, for example:

1. Physical (Source) Lines of Code – measure of all lines, including comments, but excluding blanks
2. Effective Lines of Code – measure of all lines, excluding: comments, blanks, standalone braces, parentheses.
3. Logical Lines of Code – measure of those lines which form code statements

The variations of the measure are used for specific purposes and can be regarded as measures of the same entity, but with different measurement methods according to the definitions included in ISO/IEC 15939 [14]. We can also observe that these measures are liable to systematic errors in different ways – for example the number of physical lines of code will include comments which do not add to the complexity of the algorithm, but may impact the effort needed to develop the program; the logical lines of code will naturally not be sensitive to the same type of error (i.e. comments).

5 Empirical evaluation

In order to assess what the effects of different ways of calculating measurement error have on the results, we designed a benchmarking study of calculating LOC on a number of open source projects. For the calculations we chose a set of tools which calculate the effective lines of code and should be comparable. The goal of the evaluation was to evaluate the impact of the systematic and random measurement errors on the result of the measurement process from the perspective of a quality manager.

5.1 Design

For the purpose of the evaluation we randomly chose four measurement instruments:

– Unified CodeCount (UCC), Release 2011.10,
`http://sunset.usc.edu/research/CODECOUNT/`

Table 1. Results from measuring five source code repositories with four different tools

Source code	UCC	Understand	Code Analyzer	Universal CLC	Median
Linux Kernel	11 631 288	9 466 175	11 650 363	11 207 899	11 419 931
Mozilla Firefox	5 066 234	4 282 587	5 457 003	4 966 983	5 016 609
Open Office	4 855 090	4 431 717	5 231 869	4 742 033	4 798 562
Android	878 157	878 041	876 795	871 710	877 418
Chrome	5 502 768	4 460 016	6 263 157	5 453 027	5 419 742

– Understand, `http://www.scitools.com/download/`
– Code Analyzer, Version 0.7.0, `http://sourceforge.net/projects/codeanalyze-gpl/`
– SLOCCount, Version 2.26, `http://www.dwheeler.com/sloccount/`

We also chose five different source code packages in order to capture variability in the programming styles:

– Linux Kernel, Release 3.13.6
– Mozilla Firefox, 27.0.1
– Open Office, V4.0.1 R1524958
– Android, V4.0.3 R1
– Chrome, V18.0.1025123

The size of the measured programs was more than a few hundred lines of code, which was important for the estimations – the size was too large for a person to count the number of lines of code manually and the estimates were needed.

The process of our benchmarking study was as follows. First we calculated the LOC for all software packages and calculated the median for each package. The median was chosen as it is the value that is in-between the middle data points (or the mid data point in case of odd number of data points) and therefore is can be the value that has been measured by one tool. Once we calculated the median we calculated the absolute error which is the difference between each measurement and the median. Then we calculated the relative error as a ratio between the absolute error and the median, which we use later on for comparison. Then we calculated the systematic error by using a pre-defined program as a measured entity for each measurement instrument. Then we used the measured systematic error to adjust the values of the LOC measurements from the five software packages and repeated the procedure with finding the median and calculating the absolute and the relative errors. We then compared the differences between these two calculations and discussed them.

5.2 Results

The results are presented in Table 1 and are predictably correct – all measurement tools provide different measurement results. We do not know the systematic and the random measurement error for these programs. Therefore it is not possible to distinguish the differences between these two types of errors.

Table 2. Absolute measurement error when using median as the estimator of the true value

Source code	UCC	Understand	Code Analyzer	Universal CLC
Linux Kernel	211 695	- 1 953 419	230 770	- 211 695
Mozilla Firefox	49 626	- 734 022	440 395	-49 626
Open Office	56 529	-366 845	433 308	- 56 529
Android	739	623	- 623	- 5 708
Chrome	24 871	- 1 017 882	785 260	-24 871

Table 2 shows the results from calculating the absolute relative error using the median as the substitute for the true value of the measurement. We chose median instead of arithmetic mean as it has the property of being the middle value in the data set (if the number of elements is odd). This means that it is a value which exists in the data set whereas the arithmetic mean has the property of being in the middle, but most often it does not exist in reality.

The error values shown in Table 2 indicate that there is a large discrepancy in which tools present the lowest and the highest results – visible for the measurement results for the Android source code.

Table 3 presents the percentage – the relative measurement error for the same measurement results.

Table 3. Relative measurement error when using median as the estimator of the true value

Source code	UCC	Understand	Code Analyzer	Universal CLC
Linux Kernel	1.85%	17.11%	2.02%	1.85%
Mozilla Firefox	0.99%	14.63%	8.78%	0.99%
Open Office	1.18%	7.64%	9.03%	1.18%
Android	0.08%	0	0.07%	0.65%
Chrome	0.45%	18.58%	14.34%	0.45%

The results presented in Table 3 shows that the relative error could be as large as 18.58% for the example data set (Android code measured using Understand). It is natural that this kind of error is unacceptable for any evaluation and that this kind of magnitude of the error could be caused by a systematic error combined with the total size of the software (i.e. low numbers combined with large systematic error will result in large percentage of relative error).

Table 4 shows the results from calibrating the measurement tool on a small C++ program with 21 lines of code as shown in Figure 2. The program has 10 lines of executable code and the code is written with extra white spaces (e.g. after { after the main procedure). Each line which is counted as a LOC has a number in bracket next to it (e.g. "#include<stdio.h>(1)").

The results shown in Table 4 show that several of the programs provide the results which count the lines of executable code as we intended – e.g. UCC and Code Analyzer. The tool Understand does not even count the curly brackets and

```
/* Hello World program */

#include<stdio.h> (1)
#include<stdio.h> (2)

main() (3)
{(4)
    /* multi-line
       comment
    */

    int x = 0;(5)
    printf("Hello World1");(6)

    printf("Hello World2"); // comment
                            (7)
    printf("Hello (8)
            World3"); (9)

    // single-line comment
}(10)
```

Fig. 2. Example program used for calibration of the tools

therefore can be seen as the one with a negative systematic error – the number of LOC is 8 for that program.

Table 4. Results from measuring 5 source code repositories with 5 different tools

Measurement instrument	**Measured LOC**	**Systematic Error**	**Systematic Error (relative)**
UCC	10	0	0%
Understand	8	2	20%
Code Analyzer	10	0	0%
Universal CLC	9	1	10%

One observation which we can make by comparing the relative error and the systematic error is the difference between these two. For example let's consider the tool Understand, which has a difference of 2 LOC in table 4 with the calibration results. 2 LOC is 20 % of 10 LOC of the program used for the calibration. In table 3 that tool shows a relative error in the range of $0.07\% - 18.58\%$. This means that knowing an estimate of a systematic error allows to reduce the random error from as much as 18.58% to 1.42% (calculated as $(20\% - 18.58\%)$. If we apply the same approach to all measures in table 3, we get the results as presented in table 5.

We could see that the tools which have the largest systematic error still have the largest error in total (e.g. Understand). Now, that we know the systematic error and its direction (underestimation) we can take that into the account and change the values of the LOC measurement from Table 1 and redo the calcula-

Table 5. Relative measurement error when using median as the estimator of the true value reduced by the systematic error from table 4

Source code	UCC	Understand	Code Analyzer	Universal CLC
Linux Kernel	1.85%	2.89%	2.02%	8.15%
Mozilla Firefox	0.99%	5.37%	8.78%	9.01%
Open Office	1.18%	12.36%	9.03%	8.82%
Android	0.08%	19.93%	0.07%	9.35%
Chrome	0.45%	1.42%	14.34%	9.55%

tions which results in the following number of the relative error as presented in Table 6.

Table 6. Relative measurement error when using median as the estimator of the true value reduced by the systematic error from table 4

Source code	UCC	Understand	Code Analyzer	Universal CLC
Linux Kernel	0.08%	2.42%	0.08%	5.91%
Mozilla Firefox	4.38%	3.00%	3.00%	3.13%
Open Office	7.06%	1.80%	0.15%	0.15%
Android	4.39%	14.71%	4.54%	4.39%
Chrome	4.31%	6.93%	8.91%	4.31%

The results should be compared with Table 3 as they are recalculated in the same way, given the new median (as the LOC measurement for Understand and Universal CLC are increased by 20% and 10% respectively). What the results show is that we can decrease the uncertainty (compared to Table 5) as we only have the relative error component in the measurement.

Naturally more studies are needed to use different programming styles when defining the programs used to calibrate the tools, but even this small example shows that the calibration has a significant impact on the measurement error and thus on the estimation formulas where the measure value is used (e.g. in the COCOMO estimation model).

5.3 Threats to validity

The empirical validation of the results have a number of validity threats as any other empirical study. We use the framework of Wohlin et al. [20].

The main threat to the *external validity* is the fact that we only used open source code in the evaluation. Although this threat can indicate a potential bias, we understand that the coding style differs enough between the open source projects (e.g. code written by different consortia) in order to make claim which can be generalized to other types of programs. The study by Lincke et al. [6] showed the same types of deviations as we observed in our work (but using higher-level, object-oriented metric suite), which was an independent study. This increases the validity of our conclusions as it is in fact a triangulation of the approach. However, we believe that more evaluations on more base measures to

explore if our conclusions are valid in more contexts. We also plan to expand this study in the future to allow to draw the calibration code sample from the code base and therefore have consistency between the calibration code and the code base.

The main threat to the *conclusion validity* is the size of the sample – i.e. the number of measurement instruments (code counting tools) and the number of open source programs. We used a simple calibration program in order to establish the baseline for the systematic error in the measurement. Since our main goal was to understand the scale of the measurement error in practice, the small sample size does not lower the validity of our conclusions.

The main threat to the *construct validity* stems from the assumption that there is a difference between systematic and random error in software engineering. Both concepts are established in the measurement theory and they are based on the assumption that there is some degree of non-determinism in the measurement. The non-determinism assumption can be debatable, though, in software engineering (due to automated measurement instruments). Given the fact that the tools used in our study produced a result we treated them as correct (i.e. free from bugs), but this does not have to hold for all measurement instruments. Having bugs in the measurement instruments contributes to the presence of random error and therefore we plan to investigate it more in the next steps. The goal of this study was to investigate whether this assumption is correct and the results of our study show that these concepts are present.

The main threat to the *internal validity* is the choice of measurement tools and programs. They were chosen randomly based on internet search given the criteria that they can calculate the same type of LOC measure. Since there were differences between the measurement tools we believe that the internal validity is not jeopardized; had this not been the case, i.e. there were no differences we would require to expand the study to more tools.

6 Recommendations

Measurement errors are present in almost every measurement taken, including measurements in software engineering. The notion of the errors in software engineering is similar to the same notions in other disciplines and we provide the following recommendations:

- **Calibrate the instrument using small programs**: Calibration is the most important part of measurement and provides the possibility to calculate the errors with the "true" value of the estimator known (which is not the case of measurement of larger entities). Small program allow to manually calculate the values which can be used for calibration.
- **For the calibration, use the same programming style as the code base**: Drawing the sample of the calibration code from the measured code base allows for consistency between the programming styles between the measured code and the calibration code – the same programmers usually write the code in the same way [21].

7 Conclusions

The goal of this paper was to study how the notion of measurement error can be defined for metrics in software engineering to enable correct measurement. In order to study this concept we chose one of the most used and the simplest metrics – Lines of Code. As the metric is designed to measure source code we could link the results of measuring a program to a number. This metric allowed us also to discuss the measurement error based on a small calibration program.

The results showed that both the systematic and random errors are present and that the key to distinguish between these is to use calibration of measurement tools. These results mean that the measurement processes can be significantly improved when using calibration. Instead of using the concepts of standard deviations and average to estimate the LOC in the programs, which are based on assumptions of normal distribution of the metric, calibration provides a better estimate of measurement error and thus a better estimate of the true value of LOC.

In the future we plan to expand our approach by taking into considerations the influence of the assumptions made in this work – one of these assumptions being the consistency in programming styles between the code base and the calibration code. To account for that we plan to draw the sample code for calibration from the measured code base and calculate the influence of certain programming constructs on the measurement. For example we can count the number of lines containing only the curly brackets,or the number of lines containing only white spaces to capture different programming styles.

Acknowledgment

The authors would like to thank the doctoral students in the "Measurement in Software Engineering" course at the University of Gothenburg for the discussions on the topic of measurement error and measurement theory.

References

1. A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *Software Engineering, IEEE Transactions on*, no. 6, pp. 639–648, 1983.
2. T. M. Khoshgoftaar and J. C. Munson, "The lines of code metric as a predictor of program faults: A critical analysis," in *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International.* IEEE, 1990, pp. 408–413.
3. J. S. Davis and R. J. LeBlanc, "A study of the applicability of complexity measures," *Software Engineering, IEEE Transactions on*, vol. 14, no. 9, pp. 1366–1372, 1988.

4. B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: Cocomo 2.0," *Annals of software engineering*, vol. 1, no. 1, pp. 57–94, 1995.
5. N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 284–292.
6. R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 131–142.
7. A. Abran, *Software metrics and software metrology*. John Wiley & Sons, 2010.
8. L. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," *Empirical Software Engineering*, vol. 1, no. 1, pp. 61–88, 1996.
9. L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *Software Engineering, IEEE Transactions on*, vol. 22, no. 1, pp. 68–86, 1996.
10. E. J. Weyuker, "Evaluating software complexity measures," *Software Engineering, IEEE Transactions on*, vol. 14, no. 9, pp. 1357–1365, 1988.
11. H. Zuse, *A framework of software measurement*. Walter de Gruyter, 1998.
12. J. Tian and M. V. Zelkowitz, "A formal program complexity model and its application," *Journal of Systems and Software*, vol. 17, no. 3, pp. 253–266, 1992.
13. I. B. of Weights and Measures, *International vocabulary of basic and general terms in metrology*, 2nd ed. Genve, Switzerland: International Organization for Standardization, 1993.
14. I. S. Organization and I. E. Commission, "Software and systems engineering, software measurement process," ISO/IEC, Tech. Rep., 2007.
15. I. B. of Weights and Measures, *General requirements for the competence of testing and calibration laboratories*, 1st ed. Genve, Switzerland: International Organization for Standardization, 2005.
16. ——, *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*, 2nd ed. Genve, Switzerland: International Organization for Standardization, 2014.
17. M. Staron, "Critical role of measures in decision processes: managerial and technical measures in the context of large software development organizations," *Information and Software Technology*, vol. 54, no. 8, pp. 887–899, 2012.
18. R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Torner, "Evaluating long-term predictive power of standard reliability growth models on automotive systems," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 228–237.
19. B. Boehm, "Managing software productivity and reuse," *Computer*, vol. 32, no. 9, pp. 111–113, 1999.
20. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer, 2012.
21. L. Kuzniarz and M. Staron, "Inconsistencies in student designs," in *the Proceedings of The 2nd Workshop on Consistency Problems in UML-based Software Development, San Francisco, CA*. Citeseer, 2003, pp. 9–18.