Theoretical Explanation of Recent Empirically Successful Code Quality Metrics

Vladik Kreinovich[1], Omar A. Masmali[1],
Hoang Phuong Nguyen[2], and Omar Badreddin[1]
[1]Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
vladik@utep.edu, oamasmali@miners.utep.edu
obbadreddin@utep.edu
[2]Division Informatics, Math-Informatics Faculty
Thang Long University
Nghiem Xuan Yem Road
Hoang Mai District
Hanoi, Vietnam, nhphuong2008@gmail.com

Abstract

Millions of lines of code are written every day, and it is not practically possible to perfectly thoroughly test all this code on all possible situations. In practice, we need to be able to separate codes which are more probable to contain bugs – and which thus need to be tested more thoroughly – from codes which are less probable to contain flaws. Several numerical characteristics – known as *code quality metrics* – have been proposed for this separation. Recently, a new efficient class of code quality metrics have been proposed, based on the idea to assign consequent integers to different levels of complexity and vulnerability: we assign 1 to the simplest level, 2 to the next simplest level, etc. The resulting numbers are then combined – if needed, with appropriate weights. In this paper, we provide a theoretical explanation for the above idea.

1 Formulation of the Problem

Need for code quality metrics. Computers are ubiquitous in our lives, computers are a vital part of many systems, including systems which are critically important – e.g., systems that control airplanes, systems that monitor patients in hospitals' emergency rooms, etc. In view of this importance, it is desirable to make sure that all software is as reliable as possible. For this purpose, soft-

ware engineering is designing procedures and techniques that would increase such a reliability, from recommendation on how to best design the program to recommendations on testing.

There is a need to make sure that each piece of code and each software system work perfectly on all possible situations. However, it is infeasible to test each piece of code in all possible situations: millions of lines of code are written every day, and many pieces of code are intended for use in multiple different situations. It is therefore desirable to allocate more efforts into testing software for which the probability of failure is higher – and correspondingly, somewhat less effort in testing software for which the probability of failure is lower.

Techniques that help decide which software packages have higher probability of failure and which has lower probability of failure are known as *code quality metrics*.

Where code quality metrics come from. On the qualitative level, we know – from common sense and from experience – what makes a method or a class potentially less reliable. For example:

- the longer a code, the more probable it is that it may contain a bug,
- the more methods a class contains, the more probable that this class may contain a problem,
- the more complex data types processed by the method, the more probable it is that some of the operations may be faulty, etc.

There are known qualitative classifications of some of these criteria. For example, with respect to processed data types:

- the simplest data types are integer, Boolean, and character;
- next simplest are real numbers, long-integer types, and strings;
- next in the hierarchy are arrays and tuples consisting of elements of previously mentioned types; and
- finally, the most complex are user-defined objects and arrays of complex types.

Similarly, the probability of a possible fault increases with what is called the visibility of the variable:

- the least vulnerability comes from variables marked as *private*; by definition of this marking, they can only be used by other methods from the same class;
- second in vulnerability are variables marked as *protected*, they can also be accessed by methods from several other classes; and
- the most vulnerable are variables marked as *public*, they can be, in principle, accessed by any method from any class.

There are many other such qualitative lists.

To come up with an appropriate numerical code quality metric, it is desirable to provide a numerical value to each item on each list, and then combine the resulting numerical values into a single numerical characteristic.

Currently used code quality metrics are not perfect. Several code quality metrics have been proposed; see, e.g., [3, 8] and references therein. To check how good is a code quality metric, software engineers use several software packages for which experts thoroughly analyzed all methods and all classes, and agreed on which methods and classes are better written and which are not so well written and are, thus, potentially more vulnerable.

When tested on these classes and methods, it turns out that many of the proposed code quality metrics work reasonably well. However, of course, these metrics are not perfect:

- sometimes, they mark a software as suspicious, while software engineering experts consider this software practically flawless; while
- sometimes, they mark a software as perfect, while software engineering experts see numerous faults.

Both problems are hindering our efforts:

- in the first case, if we follow the code quality metric's recommendation, we waste time and efforts on testing a perfectly good piece of code, while this time and effort could be better used to deal with really suspicious pieces of code;
- the second case is even more troublesome: if we follow the code quality metric's recommendation, we will not spend enough time and effort on testing a potentially vulnerable piece of code, and, as a result, we may miss an important mistake.

It is therefore desirable to come up with new code quality metrics, metrics that would have fewer mismatches with expert estimates.

Recent empirically successful code quality metrics and formulation of the problem. Recently, a new class of code quality metrics have been proposed; see, e.g., [9].

One of the main ideas behind these metrics is to use sequential integers to describe the above-mentioned qualitative characteristics. For example, with respect to processed data types:

- the simplest data types such as integer, Boolean, and character, are assigned complexity 1;
- next simplest data types, such as real numbers, long-integer types, and strings, are assigned complexity 2;
- arrays and tuples consisting of elements of previously mentioned types are assigned complexity 3; and

- the most complex data types – user-defined objects and arrays of complex types – are assigned complexity 4.

Similarly, depending on the variable's vulnerability, we assign different numerical values:

- to variables marked as *private* we assign complexity 1;
- to variables marked as *protected* we assign complexity 2; and
- to variables marked as *public*, we assign complexity 3.

The resulting complexities are then either simply added – or combined with appropriate weights.

An empirical analysis shows that the resulting metrics are indeed in better accordance with the metric estimates. An important question is why.

- If this empirical success is largely accidental, so that there is no good theoretical explanation for this success, then we should not expect that this metric works well in other cases.
- On the other hand, if there a good theoretical explanation for the empirical success, then we are much more confident that this metric will work in other situations as well.

What we do in this paper. In this paper, we show that the main idea behind the new code quality metrics – of assigning consequent integers to different situations – has a reasonable theoretical explanation.

2 Main Idea Behind the New Empirically Successful Code Quality Metrics: A Theoretical Explanation

Let us reformulate the problem in general terms. In both above situations, we have several groups of alternatives sorted in the increase order of their complexity:

- in the first case, we have 4 levels of complexity describing different data types;
- in the second case, we have 3 levels of vulnerability describing different options of variable's visibility.

Let us denote the number of such groups (levels) by n. Then:

- for data types, $n = 4$, and
- for visibility options, we have $n = 3$.

We want to assign, to each level i, where i goes from 1 to n, a number c_i, so that higher levels will be described by larger numbers:

$$c_1 < c_2 < \ldots < c_n. \tag{1}$$

Without losing generality, we can restrict ourselves to numbers from the interval $[0,1]$. In principle, we can use large numbers or small numbers c_i. However, since we will multiply these numbers by some weight anyway, it does not matter how big or how small are the original numbers. What is important is their relation to each other, e.g., their ratios, since these ratios do not change if we multiply all the values c_i by the same weight, i.e., go from the original values c_i to the new values $w \cdot c_i$ for some weight w.

From this viewpoint, we can always apply an appropriate weight and make sure that the resulting values are within the interval $[0,1]$. So, without losing generality, we can safely assume that all the values c_i are within the interval $[0,1]$, i.e., that we have

$$0 \le c_1 < c_2 < \ldots < c_n \le 1. \tag{2}$$

Which values c_i should we choose? In principle, we can have all possible tuples $c = (c_1, \ldots, c_n)$, as long as the corresponding tuple satisfies the condition (2).

We have no reason to believe that some of these tuples are more probable than others. So, it is reasonable to assume that all such tuples are equally probable, i.e., in probabilistic terms, that we have a *uniform* distribution on the set of such tuples. It should be mentioned that this argument – known as Laplace Indeterminacy Principle – is widely used in statistics and in data processing in general; see, e.g., [5].

Which of the possible tuples should we use? In general, in statistics, a natural idea is to use the estimate for which the mean square deviation from the actual (unknown) value is the smallest possible; this is the main idea behind the usual least squares approach; see, e.g., [10].

From this viewpoint, a natural measure of the difference between the two tuples $c = (c_1, \ldots, c_n)$ and $c' = (c'_1, \ldots, c'_n)$ is the sum of the squares of differences in coordinates

$$(c - c')^2 \stackrel{\text{def}}{=} (c_1 - c'_1)^2 + \ldots + (c_n - c'_n)^2,$$

i.e., in effect, the square of the usual Euclidean distance between the corresponding n-dimensional vectors c and c'. Thus, we should select the vector $\overline{c} = (\overline{c}_1, \ldots, \overline{c}_n)$ for which the following expected value is the smallest possible:

$$\int (c - \overline{c})^2 \, d\mu = \int ((c_1 - \overline{c}_1)^2 + \ldots + (c_n - \overline{c}_n)^2) \, d\mu, \tag{3}$$

where $d\mu$ means integration over the probability measure corresponding to the uniform distribution – i.e., in effect, over the n-dimensional volume (since a

uniform distribution in n-dimensional space means that the probability is proportional to volume, just like in a 1-D uniform distribution, probability is proportional to the length).

So what are the resulting values c_i. To find the minimum of the expression (3), we can differentiate this expression with respect to the unknown $\overline{c}_i$ and equate the derivative to 0. As a result, we get the equation

$$2 \int (\overline{c}_i - c_i)\, d\mu = 0.$$

To solve this equation, we divide both sides by 2, and use the facts that the integral of the difference is equal to the difference of the integrals, and that a constant factor (in this case, $\overline{c}_i$) can be taken out of the integral sign. As a result, we get the formula

$$\overline{c}_i \cdot \int d\mu - \int c_i\, d\mu = 0.$$

Here, $\int d\mu$ is the overall probability, i.e., 1, thus,

$$\overline{c}_i = \int c_i\, d\mu.$$

In other words, $\overline{c}_i$ is equal to the mean value of c_i with respect to a uniform distribution on the set of all the tuples $c = (c_1, \ldots, c_n)$ that satisfy the property (2). This mean value is known (see, e.g., [1, 2, 4, 6, 7]), and it is equal to

$$\overline{c}_i = \frac{i}{n+1}. \tag{4}$$

So, we arrive at the following conclusion.

Conclusion. The best way to assign a numerical value to each level i is to use the value

$$c_i = \frac{i}{n+1}.$$

This is exactly what we wanted to explain. As we have mentioned earlier, we are considering the values c_i modulo multiplication by a weight. In particular, if we take the weight $w = n + 1$, we end up with the new values

$$w \cdot c_i = i,$$

i.e., exactly with the values that lead to the new empirically successful code quality metric.

Acknowledgments

This work was supported in part by the US National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science) and HRD-1242122 (Cyber-ShARE Center of Excellence).

References

[1] M. Ahsanullah, V. B. Nevzorov, and M. Shakil, *An Introduction to Order Statistics*, Atlantis Press, Paris, 2013.

[2] B. C. Arnold, N. Balakrishnan, and H. N. Nagaraja, *A First Course in Order Statistics*, Society of Industrial and Applied Mathematics (SIAM), Philadelphia, Pennsylvania, 2008.

[3] O. Badreddin, R. Khandoker, A. Forward, O. Masmali, and T. C. Lethbridge, "A decade of software design and modeling: A survey to uncover trends of the practice", *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MODELS'18*, Copenhagen, Denmark, October 14–19, 2018, pp. 245–255.

[4] H. A. David and H. N. Nagaraja, *Order Statistics*, Wiley, New York, 2003.

[5] E. T. Jaynes and G. L. Bretthorst, *Probability Theory: The Logic of Science*, Cambridge University Press, Cambridge, UK, 2003.

[6] O. Kosheleva, V. Kreinovich, J. Lorkowski, and M. Osegueda, "How to transform partial order between degrees into numerical values", *Proceedings of International IEEE Conference on Systems, Man, and Cybernetics SMC'2016*, Budapest, Hungary, October 9–12, 2016.

[7] O. Kosheleva, V. Kreinovich, M. Osegueda Escobar, and K. Kato, "Towards the most robust way of assigning numerical degrees to ordered labels, with possible applications to dark matter and dark energy", *Proceedings of the 2016 Annual Conference of the North American Fuzzy Information Processing Society NAFIPS'2016*, El Paso, Texas, October 31 – November 4, 2016.

[8] O. Masmali and O. Badreddin, "Model driven security: a systematic mapping study", *Journal of Software Engineering*, 2019, Vol. 7, No. 2. pp. 30–38.

[9] O. Masmali and O. Badreddin, "Towards a model-based fuzzy software quality metrics", *Proceedings of the International Conference on Model-Driven Engineering and Software Development MODELSWARD'2020*, Valletta, Malta, February 25–27, 2020, Vol. 1, pp. 139—148.

[10] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman and Hall/CRC, Boca Raton, Florida, 2011.