# Relevance of the Cyclomatic Complexity Threshold for the Java Programming Language

Miguel Lopez, Naji Habra

## Abstract

*Measurement can help software engineers to make better decision during a development project. Indeed, software measures increase the understanding a software organization have concerning its projects. Measures can give answers to lots of questions, that is, how much are we spending on software development, what are the error (reliability) characteristics of software in our organization? Nevertheless, generating a set of measurement data is not enough to make good decisions. In fact, the interpretation of a measurement can be difficult, since the meaning of a numerical value that characterizes a given software attribute (reliability, coupling …) is still hard to understand without any strong reference.*

*One major problem met during software measurement process is related to the determination of measure thresholds. Thresholds are numerical bounds associated with a given measure, which allow identifying weak portions of the code. In other words, bounds help the measurer identifying attributes that are outside an acceptable range.*

*To illustrate our approach, we will work on the Mc Cabe's cyclomatic complexity number. The upper bound established by McCabe for its complexity measure in the context of procedural programming equals 10. Is this value absolute? How can we adapt this numerical value to other programming contexts? In other words, is the cyclomatic complexity threshold independent of the programming language or other context factors?In order to answer these questions, the paper proposes to check the relevance of cyclomatic complexity threshold for an object-oriented language, i.e. Java. A sample of 694 Java products has been measured. Some descriptive statistics have been computed in order to better understand the relevance of the cyclomatic complexity threshold. The first goal of this work was to build the frequency distribution of the cyclomatic complexity number. And, based on this distribution, the relevance of the threshold suggested by McCabe has been analyzed.*

*Experiment results are unexpected! Indeed, more than 90% of the measured methods have a complexity less than 5. Complexity threshold suggested by the state of the art seems irrelevant, since most methods are simple. In this case, using a threshold of 10 could lead the measurer to make wrong decisions.*

*This illustrates the fact that measure thresholds are context dependant; and that any metric-based decision model has a limited validity context which must be explicitly determined and stated.*

## 1. Introduction

Even though measuring correctly a given attribute of a given software product is difficult and complex, it is not enough for making good decisions. Other difficulties arise during the interpretation phase where those values that should be considered as acceptable have to be precisely defined.

In fact, maximum and minimum threshold values must be specified precisely for each measure, before measurement application, in order to help decision making during the result exploitation. However, determining such thresholds on a sound basis is far from being an obvious task. Going beyond an ad-hoc choice of thresholds based on the experience would only necessitate to manipulate significant data sample which is usually hard to collect and costly to deal with.

In addition, it seems obvious that these maximum and minimum threshold values depend on several factors that can modify them. Nevertheless, when the thresholds are provided with a given measure, the using conditions of these values are seldom specified.

The current paper attempts at clarifying the question of the relevance of thresholds on the case of a very famous software attribute, i.e., complexity measured by the McCabe cyclomatic number. This measurement method is proposed with a maximum threshold equal to 10 and no restricting conditions of such a limit are given. The paper is based on an empirical stydu involving 695741 measurement results; a huge amount of data which leads us to question seriously the soundness of McCabe threshold value of 10 or at least the conditions of its use.

According to [1], the cyclomatic complexity of a software module is calculated from a connected graph of the module or a method in the case of a Java program (that shows the topology of control flow within the program):

$$\text{Cyclomatic complexity (CC)} = E - N + p$$

where $E$ is the number of edges of the graph,

$N$ is the number of nodes of the graph, and

$p$ is the number of connected components.

The paper is organised as follows. Section 2 develops the problem with threshold in general and Section 3 presents some related works about threshold fixing in the case of McCabe number. The main part of the work is Section 4 which describes the empirical study. Section 5 discusses the result obtained by this study and Section 6 gives some concluding remarks.

## 2. Problem Statement

Measurement in software engineering remains an arduous activity. Different kinds of difficulties arise in the different phases of the measurement lifecycle, from the measurement design to the results exploitation. Several previous works highlight difficulties related to the design phases where it is not obvious to ensure that a measurement method measures what it purport to measure, that the scale is correct, etc [1],[4],[5]. But, even when all these difficulties are fixed, the exploitation of a given measurement result remains a problem rarely addressed in literature.

In fact, the exploitation of a specific measurement result leads the measurer to analyze the concerned value and to make decision on basis of his analysis; this determinant activity is still quite difficult. As a simple example, how can the measurer conclude that a given O.O. software having, say 100 classes, is a *big* software or a *median* one? More precisely, the question is to know on which basis he should make such a judgement. What is the relevance of an estimation based only on his/her own experience?

So, one important issue of the measurement exploitation phase, which is seldom investigated in scientific literature, is the threshold fixing. And, different questions are related to this issue :

- ♦ How to fix thresholds delimiting acceptable values for a given software measurement method ?
- ♦ Are those thresholds context-dependent?

- If we admit they are context-dependent, how to define the parameters impacting the context in order to determine whether using given thresholds are still meaningful or not?

These questions are both *crucial* and *hard* to answer.

The questions are crucial. In fact, answering these questions determines the reliability of the decisions being made and thereby the soundness of the whole measurement process. In other words, lack of confidence about the rationale of a given threshold can prevent the use the measurement result. Practically, in actual software projects, an unfounded threshold can lead the measurer to consider the measurement result as useless and to look for a "better measure", even though the measurement design and the measurement application fulfil all the required validity conditions.

The above questions are also hard to answer. In fact, the most usual way to determine thresholds is to choose them simply on basis of experience. But, is that possible to imagine another way of identifying such values? Another approach might be, to compare individual values against an empirical frequency distribution of a given measure and to pay attention to those cases which differ dramatically from the mean. This approach is time and effort consuming, since the production of a representative sample remains a difficult task.

In addition, checking the relevance of thresholds within a data set presents two practical difficulties according to [6]. On the one hand, software engineers do not measure their projects; software measurement is still a rare activity within software projects. On the other hand, even when those data exist, they are often considered as confidential. And, so building a representative sample of real software systems remains an arduous task.

Despite the above difficulties, our belief is that suggesting a repeatable and reproducible method to identify thresholds could facilitate the exploitation of measure, improve their validity and thereby increase their potential use.

## 3. Related Works

Two main methods for identifying the cyclomatic complexity number thresholds are described in the literature.

1. According to the first method, the threshold of the complexity attribute is determined in relation with two other attributes. Indeed, the cyclomatic complexity number is studied in terms of the attribute *testing effort* and the attribute *bug density* or problem amount [15][16][17][18][19][20]. This method is three-fold.

   Firstly, a set of programs is measured in terms of cyclomatic complexity. Secondly, the testing effort or the problem amount met during testing is collected. Finally, the correlation is computed between these measures. When the testing effort or the problem amount is considered too heavy by the decision makers, the corresponding cyclomatic complexity value is labelled as too complex. In that case, a rework of the code is recommended [1],[3].

   The main problem with such kind of studies remains in the conditions under which the experiments have been run, i.e. the context. Indeed, factors like programming languages and programming paradigms, developer skill level, tester skill level or development model (collaborative development, iterative, …) are not taken into account. Therefore, the relevance of the complexity number is seriously questionable in that paper. Is this meaningful to specify thresholds without specifying external factors that can influence their values? In other words, it can be assumed that complexity of object-oriented programs is not captured by such a measurement method. In that sense, it could be interesting to specify those influent factors and their impact on the value of the threshold.

2. According to the second method [7], an empirical study shows the frequency distribution of the cyclomatic complexity number computed within a sample of 16 products; and threshold is determined on basis of the distribution. Three languages are represented in this sample. Nevertheless, only one type of programs is considered in the study. In other words, the factor ‘programming language’ is not considered as influent. This work indicates that 50% of the functions have a complexity less than 10 and 90% less than 80. Based on these data, three categories of complexity are defined (< 10, >= 10, and > 80). Moreover, this study briefly explains investigations on correlations with the number of problem reports and the maintenance effort. For these investigations, the experimental conditions are not specified.

Nevertheless, studying the correlations between cyclomatic complexity number and other measures is a quite important task that must be realized. Moreover, such information can help making decisions with measures.

Current work suggests considering the cyclomatic complexity number without any supposed related measures. Indeed, the main goal of this paper is to evaluate the relevance of the cyclomatic complexity threshold suggested in [1] by verifying the percentage of Java methods that have a complexity number greater or equal than 10 within a representative sample.

## 4. Empirical Study Description

### 4.1. Sample Description

In this work, we build a sample of 694 software Java projects. Most of the products (691) are coming from the Open Source community, *i.e.* Sourceforge [8]; and a very low number of products (3) are coming from the industry. So, the projects that populate this sample are real software systems and in that sense, it can be argued that we have a representative sample.

But, which population does this sample represent exactly? Before answering this question, it is important to solve the three following problems.

Firstly, which is the entity related to cyclomatic complexity? According to [1],[4], the McCabe cyclomatic complexity related entity (i.e. the entity being measured) for Java programs is the “method within a class”. Concretely (at the measurement tool level), the Cyclomatic Complexity number is related to a method. In that sense, the population studied here is the population of *Java methods*.

Secondly, most of the products of the sample are labelled in the Sourceforge website as *production software*. That is, these products are mature enough to be deployed in a production environment. Actually, this quality of the software products is not considered as relevant, since the own members of the project freely assign the production label to their product. In that context, the verification of such quality remains difficult.

Thirdly, the open source characteristic can be considered as a weakness of the sample. Indeed, it is often accepted that open source and closed source (industrial) software are two different types [12][13] of software. Both projects are so different in terms of process that the resulting products cannot be regarded as similar software. This implies that mixing both “types” can lead us to draw incorrect conclusions. Basically, in the scope of current paper, it is assumed that this distinction (close source *vs* open source) is not relevant. In other words, close and open source programs belong to the same category in terms of cyclomatic complexity. It is accepted that the independent variables *close source* and *open source* do not have any impact on the dependent variable cyclomatic complexity number.

So, our sample represents a population of Java methods. The "maturity level" of the product and the "open source label" are not seen as qualities of the entity *method* that can have a serious influence on the cyclomatic complexity number. Both working assumptions are set up in order to facilitate the current work.

Table 1 shows a summary of the Java methods sample. The methods considered in this sample are the method with a non-empty body. A method with a non-empty body is a method whose Halstead program length is greater than 0. The Halstead program length takes into account the number of operators and operands [9].

Only concrete methods with a non-empty body have been considered in this study. Indeed, empty methods would seriously enhance the proportion of the method with a cyclomatic complexity number of 1, whereas the real cyclomatic number value of such methods would have to be non-applicable.

| | |
|---|---|
| Number of Products | 694 |
| Number of Classes | 80136 |
| Number of Non-Empty Methods | 695741 |
| Proportion of Non-Empty Methods | 100% |

**Table 1**

## 4.2. Empirical Study Procedure

The current empirical study is organized in 5 steps:

- Automatic download of the 694 software *products* from SourceForge done by a python script: each project is copied into a directory whose name is the name of the product.
- Computing of the cyclomatic complexity number: a python script launches the tool such that, for each product, the cyclomatic complexity number is computed. The measurement tool JStyle [10] can be launched with a command line interface.
- JStyle generates a text file, which contains the measurement result. So, each file corresponds to one software product.
- Another python script merges all the text files into one single file.
- The merged file is loaded in R software wherein the frequency distribution of the cyclomatic complexity number is computed.

## 4.3. Results

Current section gives the results of the computing in statistical software R. Table 2 shows the results of the frequency distribution for the cyclomatic complexity number. The value zero does not exist because the counting of the decision nodes always starts at 1 [1][2].

The first column shows the value of the Cyclomatic Complexity Number (CCN). The second column shows the percentage of methods with a CCN less than the value of the first column. The frequency column is the percentage of methods with a CCN that equals the value of the first cell in column CCN. The last column shows the percentage of methods with a CCN strictly greater than the corresponding CCN value.

So, reading this table teaches us that the value 10 has a frequency of 0,39%. In other words, 0.39% of the methods has a cyclomatic complexity number that equals 10. While only 2,01% have complexity greater than 10.

| CCN | Cumulative Frequency <= | Frequency | Cumulative Frequency > |
|---|---|---|---|
| 1 | 66,73 | 66,73 | 33,27 |
| 2 | 80,73 | 14,00 | 19,27 |
| 3 | 87,60 | 6,87 | 12,40 |
| 4 | 91,45 | 3,85 | 8,55 |
| 5 | 93,85 | 2,40 | 6,15 |
| 6 | 95,31 | 1,46 | 4,69 |
| 7 | 96,35 | 1,04 | 3,65 |
| 8 | 97,07 | 0,72 | 2,93 |
| 9 | 97,60 | 0,53 | 2,40 |
| 10 | 97,99 | 0,39 | 2,01 |
| 20 | 99,45 | 1,46 | 0,55 |
| 30 | 99,75 | 0,30 | 0,25 |
| 40 | 99,86 | 0,11 | 0,14 |
| 50 | 99,91 | 0,05 | 0,09 |
| 60 | 99,93 | 0,02 | 0,07 |

**Table 2**

Moreover, 94% of the methods have a complexity number between 1 and 5. And only 33,27% have complexity number greater than 1.

## 5. Results Discussion

The results present in Table 2 are quite surprising, since most of the methods (95%) have a complexity between 1 and 6. And, only 2% of the methods have a complexity greater than 10. If the rule of thumb suggested by McCabe [1] is applied, only these 2% of the methods must be reworked in order to reduce complexity. Moreover, 94 % of the methods can be considered as simple, that is, reworking is not necessary. And, 66% of the methods have a complexity number that equals 1.

The observation made upon this sample is that methods are mainly simple in terms of complexity ($\leq$ 5). And, a minority of these methods are complex (2%). In other words, if a method is randomly taken out of the sample, the probability that this method is complex equals 2%. So, this probability is very small.

It seems that in this population, methods are simple and developers take into account the complexity problem in terms of cyclomatic complexity. The question is now: how can we explain such a low level of cyclomatic complexity? Three assumptions can be set in order to answer this question.

1. Firstly, the label *Production* of all the open source software of the sample can eventually considered as reliable. Indeed, the empirical study shows that the maturity level of these products is quite high in terms of cyclomatic complexity. Therefore, we observe a majority of simple methods (cyclomatic complexity number $\leq$ 5). As mentioned in Section 4.1, the team members freely give the "production" label to software, and, for that reason, we do not take into account this label. However, the results could lead us to infirm the assumption of non-relevance of the *Production* label.
2. Secondly, the open source development is currently investigated in order to find out its specificities [11][12][13][14]. A main characteristic of such a development model is the distributed environment features. In open source projects, the team often stands in several

physical sites and therefore must share source code which is supposed to be analysable, modifiable, testable, and finally maintainable. So, since the state of the art teaches us that the cyclomatic complexity is related to maintainability and testability [3][15][16][17][18][19][20], it could be possible that the distributed context of open source software actually represents a constraint, which forces the software, *i.e.* the methods, to be as simple as possible.

3. Thirdly, the Object-Oriented paradigm can be a factor that influences the threshold. Indeed, the cyclomatic complexity number was designed for Fortran programs, and the programming language of the current sample is Java. Maybe, due to some Object-Oriented features, the complexity of such source code is not captured enough.

## 6. Conclusion

The third assumption above leads us conclude with the following question: "is the threshold of 10 for McCabe number significant for O.O programs?".

In fact, as the cyclomatic complexity number was designed for imperative programming paradigm where the concept of "decision point" corresponds to a "node" in the flow graph, one can argue that in O.O. "decision points" are not really captured by the graph. In that sense, polymorphism can be one of the rationales for the low frequency of complex methods (66% of the methods with a complexity number that equals 1). In fact, overriding methods allow the developer to embed decision points in different implementations of the same method signature. It is possible that this kind of idiom is often used in Java programming, and therefore can explain the low amount of complex methods.

Whatever is the explanation, our empirical study showed that the thresholds associated to McCabe cyclomatic number is not discriminating enough to make good decisions. By analogy, 2% of the human beings have a height greater than 2,10m. If we decide that 2,10m is a good threshold to identify a given person as big, then a woman with 2m is not big. This last assertion is a non-sense, because a woman whose height is 2m cannot be considered as not big. In this analogy, it is obvious that the threshold depends in particular on the gender. So, the height threshold is context-dependent.

Finally, it seems important, for any thresholds proposed with a given measurement, to determine the contextual parameters that influences those thresholds, and thereby to delimit the context of their usability. According to the result of the frequency distribution of the cyclomatic number in our empirical study, one parameter that seems to have an influence of the threshold value has been identified, *i.e.* programming paradigm. It could be interesting to verify these assumptions, and to find out the other parameters.

## 7. Acknowledgement

This research project is supported by the European Union (ERFD) and the Walloon Region (DGTRE) under the terms defined in the Convention n EP1A12030000072130008.

## 8. References

[1] McCabe, T.J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, October 1976

[2] Abran, A., Lopez, M.,. and Habra, N., “An Analysis of the McCabe Cyclomatic Complexity Number”, in IWSM Proceedings, Berlin, 2004.

[3] Watson, A. H. , and McCabe, T. J., “Structured testing: A testing methodology using the cyclomatic complexity metric”. *NIST Special Publication*, 1996

[4] Habra, N., Abran A., Lopez, M. & Paulus, V., “Towards a framework for Measurement Lifecycle”, University of Namur, Technical Report, TR37/04, 2004 *(to be published).*

[5] Lopez, M., Paulus, V., and Habra; N., “Integrated Validation Process of Software Measure.”, In *Proceedings of the International Workshop on Software Measurement* (IWSM 2003), 2003.

[6] Britoe e Abreu, P., Poels, G., Sahraoui H.A., and Zuse, H., “Quantitative Approaches in Object-Oriented Software Engineering”, Kogan Page Science, 2004

[7] Stark G., Durst, R.C., “Using Metrics in Management Decision Making”, Computer (IEEE), 1994

[8] http://www.sourceforge.net

[9] Halstead, M. H. “Elements of Software Science, Operating, and Programming Systems”, Series Volume 7. New York, NY: Elsevier, 1977

[10] http://www.mmsindia.com/

[11] Capiluppi, A. and Lago, P., “Characterizing the OSS process”. In Proceeding of 2nd Workshop on Open Source Software engineering, (2002), Florida.

[12] Feller, J., Fitzgerald, B., “Open Source Software Development”, Addison-Wesley, 2002.

[13] Feller, J., Fitzgerald, B., “A framework analysis of the open source software development paradigm.”, In Proceedings of ICIS 2000, (2000), 58-69.

[14] Scacchi, W., “Software Development Practices in Open Software Development Communities.”, In Proceedings of 1st Workshop on Open Source Software Engineering, Toronto, Ontario, (2001).

[15] V.R. Basili, L.C. Briand, and W.L. Melo, “A Validation of ObjectOriented Design Metrics as Quality Indicators,” IEEE Trans. Software Eng., vol. 22, no. 10, pp. 751-761, Oct. 1996.

[16] Rombach, H.D. ,"Design Metrics for Maintenance", Proc. 9th Annual Software Engineering Workshop, NASA, Goddard Space Flight Centre, Greenbelt, Maryland, Nov. 1984, pp. 100-121.

[17] Blaine, J.D., Kemmerer, R.A. "Complexity Measures for Assembly Language Programs", JSS, 5, 1985,

[18] Gill, G., and Kemerer, C., “Cyclomatic Complexity Density and Software Maintenance Productivity,” IEEE Transactions on Software Engineering, December 1991.

[19] Heimann, D., “Complexity and Defects in Software—A CASE Study, ”Proceedings of the 1994 McCabe Users Group Conference, May 1994.

[20] Kafura, D., and Reddy, G., “The Use of Software Complexity Metrics in Software Maintenance,” IEEE Transactions on Software Engineering, March 1987.