Computer Science and Information Systems 17(1):181–203 https://doi.org/10.2298/CSIS181012035B

Comparison of systematically derived software metrics thresholds for object-oriented programming languages

Tina Beranič[1] and Marjan Heričko[1]

University of Maribor,
Faculty of Electrical Engineering and Computer Science,
Koroška cesta 46, 2000 Maribor, Slovenia
{tina.beranic, marjan.hericko}@um.si

Abstract. Without reliable software metrics threshold values, the efficient quality evaluation of software could not be done. In order to derive reliable thresholds, we have to address several challenges, which impact the final result. For instance, software metrics implementations vary in various software metrics tools, including varying threshold values that result from different threshold derivation approaches. In addition, the programming language is also another important aspect. In this paper, we present the results of an empirical study aimed at comparing systematically obtained threshold values for nine software metrics in four object-oriented programming languages (i.e., Java, C++, C#, and Python). We addressed challenges in the threshold derivation domain within introduced adjustments of the benchmark-based threshold derivation approach. The data set was selected in a uniform way, allowing derivation repeatability, while input values were collected using a single software metric tool, enabling the comparison of derived thresholds among the chosen object-oriented programming languages. Within the performed empirical study, the comparison reveals that threshold values differ between different programming languages.

Keywords: software metrics, threshold values, reference values, object-oriented, benchmark data, programming language interdependence, reliable derivation, repeatability, replication

1. Introduction

Measurement is a key component for good software engineering, important for understanding, control and improvement [9]. It is performed with software metrics that facilitate the monitoring of the achieved quality level [19]. As defined, the software metric is a quantitative measurement of the degree to which an evaluated entity possesses a specific attribute [17]. Many different object-oriented software metrics have been introduced [18,29,6,3,24]. However, their use in practice, especially within software quality evaluation, is limited, since reliable threshold values have not been proposed, [19,10,1].

Evaluating software quality with software metrics thresholds is a known approach [29]. When software metric values of the assessed software entity exceed the threshold values of the evaluated software metrics, this indicates potential problems in the form of code deficiencies or smells, non-optimal source code, or different structural problems. In the study by Beranič et al. [5], identification of deficient code was done using the combination of software metrics and corresponding threshold values. The performed expert

judgment confirmed the efficiency of identification based on derived thresholds, since, by using the highest threshold, the proposed evaluation resulted in the detection of truly deficient software classes.

Since efficient software quality evaluation can be done only with reliable threshold values, the process of threshold derivation is very important. Different approaches for deriving threshold values are proposed in the literature [4], including approaches based on benchmark data, like [19,10,1,30,13,21]. They use software metric values as an input, and provide concrete threshold values for selected software metrics. Resulting thresholds vary between studies, and besides, generally applicable and accepted threshold values cannot be found within related work. This may be due to the different challenges which exist in the domain of software metrics threshold derivation.

When deriving threshold values, it is crucial that the input data sets are transparent, and that they are gathered in a systematic and uniform way. With this, the reliability of the results is increased and repeatability of calculations is achieved. Different software metric tools are available that enable the collection of software metric values. However, the implementation of the same software metric often varies within different tools [31,10,13,44,22,14,33], resulting in different values for the same software metric using the same input data. In various software metric tools, a set of supported metrics differs and, additionally, new tool-specific software metrics can be detected. Available software metric tools rarely enable the collection of software metric values for more than one programming language, wherein the broadest support is available in the Java programming language [4]. The above-mentioned challenges affect the derivation of software metric threshold values directly, especially when using benchmark-based approaches, where the gathered metric values present an input data set into the threshold calculation step.

Also, different approaches for software metric threshold derivation based on benchmark data are available in existing literature. Each has its own characteristics, that are reflected in differing threshold values, though the input data set is the same. In our preliminary research, we compared threshold values derived by using approaches by Ferreira et al. [10], Oliveira et al. [30] and Alves et al. [1], and confirmed that the derived thresholds vary. This was also confirmed by Yamashita et al. [44], where they observed that the derived thresholds would differ if a different derivation approach were to be used. Hence, to provide comparable results, it is important that only a single derivation approach is used.

The prevalence of the Java programming language within software metric tools is also detectable between existing threshold derivation approaches, where most of the threshold values are derived for Java, while other object-oriented programming languages are not covered. The only exceptions, using the benchmark based threshold derivation approach, are the studies by Alves et al. [1], which derived threshold values for C#, and by Lanza and Marinescu [19], who derived threshold values for the C++ programming language. Though some individual examples of derived thresholds exist, the systematic analysis of threshold values between programming languages was not detected.

Based on the presented background, our research pursued the following research question: *Do software metric threshold values differ between different object-oriented programming languages?* We analyzed if the programming language has an influence on the derived threshold values. The presented research work describes a systematic threshold derivation, which enables a reliable analysis and comparison of software met-

ric threshold values. In the paper, software metric threshold values are derived for four object-oriented programming languages, namely Java, C++, C# and Python. Considering the above-mentioned challenges, the derivation was conducted using a single threshold derivation approach within all of the selected programming languages. Thresholds were derived for nine class level software metrics. Metric *CountLineCode*, that counts the number of lines of code in a class, *AvgLineCode*, expressing the average size of methods in a class, metric *SumCyclomatic*, presenting the sum of cyclomatic complexities of all the methods in a class, *AvgCyclomatic*, expressing the average value of cyclomatic complexity in the methods of a class, metric *MaxNesting*, measuring the maximal nesting level in a class, *CountClassCoupled*, counting the classes to which a class is coupled, software metric *PercentLackOfCohesion*, expressing the lack of cohesion in a class, metric *CountDeclMethodAll*, counting the number of methods in a class, and metric *MaxInheritanceTree*, expressing the maximum depth of a class in the inheritance tree. Benchmark data was collected systematically, and input values were collected using a single software metric tool. With this, the comparison of derived software metric threshold values between programming languages was enabled, and a replication of the performed derivation approach was provided.

The structure of the paper consists of the following parts. Chapter 2 presents related work, followed by Chapter 3, presenting a threshold derivation approach that is based on a statistical distribution of benchmark data. In Chapter 3, adjustments to the approach are proposed, and the tools and data sets used within the empirical study are presented. Chapter 3.3 describes the calculation of thresholds, covering the distribution analysis of input values, resulting in concrete threshold values. Later, an analysis and comparison of derived threshold values are presented in Chapter 4. Limitations and threats to validity are presented at the end.

2. Related work

Software quality evaluation with software metrics can be done only when reliable threshold values are defined. Different approaches for deriving threshold values are available in the literature [4]. Fontana et al. [13] categorizes derivation approaches into (1) approaches based on observations, (2) error-based approaches, (3) approaches using machine learning and, (4) approaches that derive thresholds based on a statistical analysis of benchmark data. In the presented research, we focus on the latter.

Table 1 lists derivation approaches based on benchmark data. The approach proposed by Lanza and Marinescu [19] derives threshold values using the mean and standard deviation, but the distribution of input data sets is not considered. On the other hand, the majority of studies consider the assumption that software metrics values usually follow a power law distribution [10,30,38,1,7], since distribution has a significant impact on software metric interpretation [38]. The distribution is also considered by Lavazza and Morasca [20]. Although they use the mean and the standard deviation, an improvement is proposed that enables the use of data that does not follow a normal distribution. Approaches by Ferreira et al. [10], Oliveira et al. [30], Alves et al. [1], Lima et al. [21], Vale and Figueiredo [41] and Vale et al. [42] consider the fact that software metric values usually do not follow a normal distribution, leading to the inapplicability of methods connected to normal distribution [1,38]. Ferreira et al. [10] identify threshold values for six

Study	Programming language(s)
Alves et al. [1]	Java, C#
Ferreira et al. [10]	Java
Filo et al. [12]	Java
Fontana et al. [13]	Java
Lanza and Marinescu [19]	Java, C++
Lavazza and Morasca [20]	Java
Lima et al. [21]	Java
Oliveira et al. [30]	Java
Vale and Figueiredo [41]	SPL benchmark (FH-Java, AHEAD, FH-JML)
Vale et al. [42]	SPL benchmark (FH-Java, AHEAD, FH-JML), Java

Table 1. Literature proposing software metric threshold derivation approaches

object-oriented software metrics reflecting a common practice. Filo et al. [12] introduced two improvements to the approach presented by Ferreira et al. [10], the modification of the ranges names, and the use of two percentiles, dividing the values into three areas. Alves et al. [1] propose an approach using weighting according to the size of the entities, wherein the approach was an inspiration for the work by Fontana et al. [13]. Fontana et al. [13] define thresholds for the metrics used in the code smell detection rules. Lima et al. [21] addressed the area of threshold derivation for annotations in the Java programming language, and Vale and Figueiredo [41] present a threshold derivation approach in the software product lines context. In the study [42], Vale et al. generalize the approach by using a benchmark composed of 103 Java open source projects. The major difference according to other available studies is that Vale et al. [42] also extract the lower bond thresholds, namely the 3rd and 15th percentiles. Oliveira et al. [30] introduce the concept of the relative threshold, implemented in an RTTool [31]. Another threshold derivation tool, a TDTool, is presented by Veado et al. [43].

As seen in Table 1, the threshold values are mainly derived for the Java programming language. In rare cases, thresholds are derived for two or more languages, as in [1,19,42]. However, the empirical comparison of threshold values between programming languages was not found within the related work.

The studies were done in order to define the impact of different contextual factors on derived threshold values, or on the distribution of software metric values. Ferreira et al. [10], in an experiment, analyzed the impact of thematic domains in gathered benchmark data. As they observed, there is only a slight difference between the thresholds derived using a full data set and thresholds derived within each of the thematic domains [10]. Even more, the results show that software metric values follow the same probability distribution regardless of the application domain. Mori et al. [28] also analyzed the impact of domains on derived thresholds. In contrast to the study described above, they found evidence that software metrics thresholds are sensitive to the software domain, but we can still find domains that have similar thresholds for some of the analyzed software metrics [28].

On the other hand, Dósea et al. [8] conducted an empirical study regarding design decisions influencing the distribution of software metrics. As they conclude, the design roles affect the distribution of metric values, wherein design roles include architectural roles and classes with application-specific responsibility that are not connected to any specific reference architecture [8]. The impact of programming language on the distribution of software metrics values was included in the study by Zhang et al. [45] and a

study by Gil and Lalouche [15]. Zhang et al. [45] present a study about the impact of different contextual factors on maintainability metrics. They found out that application domain, programming language, and the number of changes, are the most influential factors regarding the distribution of software metric values [45]. They use the data from 320 software projects, selected randomly from SourceForge, but they were not represented equally for each of the used programming languages, which can have a great impact on the validity of results. A similar study was done by Gil and Lalouche [15], where they found out that every project is different, therefore, measurement in one project could not be used for making predictions in another software project.

Namely, the challenges connected to benchmark data present one of the biggest challenges when using benchmark based threshold derivation approaches. Considering the above-mentioned challenges, we collected the data in a systematic way, where each programming language is represented equally and selected software projects correspond to the whole population of open-source software available on the chosen online repository. Moreover, to avoid the impact of a specific software project on a final result, we gathered a large set of benchmark projects, since Lochmann [23] found out that, with large numbers of data, the diversity of areas and the variance of results decreases correspondingly.

Our research work aims at providing a comparison of the software metrics threshold values for the object-oriented programming languages, namely Java, C++, C# and Python, considering known challenges. The comparison is based on the performed analysis of systematically derived threshold values for nine software metrics. By adopting and adjusting the existing derivation approach based on benchmark data and considering the distribution of software metric values, thresholds are calculated based on systematically collected benchmark data and uniformly collected input values.

3. Threshold derivation approach

As presented in the related work, software metric values are used as an input in different threshold derivation approaches. Lanza and Marinescu [19] do not consider data distribution, and Lavazza and Morasca [20], despite the presented changes, do not consider fully the distribution of software metric values. Vale and Figueiredo [41] and Vale et al. [42] present a method that also derives the lower thresholds that are not a priority when identifying deficient source code, and Lima et al. [21] present an approach targeting annotation for Java programming languages. Therefore, only papers by Alves et al. [1], Oliveira et al. [30] and Ferreira et al. [10], with improvements presented by Filo et al. [12], that present derivation approaches and resulting in concrete threshold values used for the evaluation of software projects, were selected for a detailed analysis. Approaches are similar, wherein Alves et al. [1] weigh the program entities based on their size expressed with lines of code software metric, Ferreira et al. [10] consider the frequency of a specific software metric value, and Oliveira et al. [30] introduce the concept of a relative threshold. As shown by the performed comparison, weighing by size results in very high threshold values, and the approach presented by Oliveira et al. [30], with the exception of the newly presented concept, resulted in threshold values consistent with thresholds provided by the approach proposed by Ferreira et al. [10].

We decided to adopt the approach presented by Ferreira et al. [10]. The approach focuses on the statistical properties of analyzed data and object-oriented programming

languages. Thresholds are defined according to the frequency concept within the benchmark data. The proposed approach can be summarized with the following steps [10]:

1. The software metrics values of selected benchmark projects are gathered, forming an input data set.
2. The distribution of software metrics values is determined for each metric, using a visual analysis, and by using a distribution fitting tool.
3. The thresholds are derived based on the best-fitted distribution. If representative values exist for a best-fitted distribution, it is defined as the threshold. Otherwise, three areas are determined using a visual examination:
   - The *Good* area joins the values of software metrics with a high frequency of occurrence. Those values are used most commonly in practice.
   - The *Regular* area represents an intermediate zone, joining the values that are not commonly used, and, on the other hand, are also not very rare.
   - The *Bad* area joins values with a very low frequency of occurrence.

In the performed experiment, Ferreira et al. [10], with the use of derived thresholds, identify software classes with structural problems, wherein a bad value indicates the existence of design problems, and a good value indicates the absence of structural problems in a class.

The approach was repeated and upgraded by Filo et al.[12]. They introduced two main improvements. The first is connected to the identification of thresholds. Instead of using the visual identification, Filo et al. [12] introduced the use of two percentiles that divide the values into three areas. The percentiles are points dividing values into 100 equal parts [11]. The use of percentiles was adopted from Alves et al. [1]. The second improvement is connected to threshold naming. They complement existing names to achieve a better understanding of each defined threshold. The names of the ranges are as follows: *good/common*, *regular/casual* and *bad/uncommon*, but the use of derived software metrics' thresholds for identification of the anomalous values indicating a potential problem in source code, remains unchanged [12].

3.1. Adjustments of the adopted approach

Based on the analysis of the replicated approach presented by Ferreira et al. [10] and improvements introduced by Filo et al. [12], we propose some additional adjustments for the adopted threshold derivation approach. When software metric thresholds are fied using the presented steps, a few challenges arise. The first one is related to the fication of threshold values that limit the mentioned areas. A visual examination was already replaced by the use of two percentiles in the study presented by Filo et al. [12]. Adapted from Alves et al. [1], they use the 70th and 90th percentiles to form three risk areas, although the primary study by Alves et al. [1] proposed the use of three percentiles, i.e. 70th, 80th and 90th to form four risk areas.

We propose an adjustment of the replicated approach by using two or three percentiles, depending on the input data range and suitability of values. If the software metric values occupy a wide range of data, thresholds can be determined with three percentiles: 70th, 80th and 90th. If the values occupy a limited range of data, thresholds should be fied with two percentiles: 70th and 90th, and the very high risk area should not be included. Also, to

exclude any subjectivity and to gain accuracy, we propose that the fication of percentiles is done based on raw data sets instead of the visual examinations used by Ferreira et al. [10] and Filo et al. [12].The detailed steps of the used threshold derivation approach are presented within algorithm 1.

Algorithm 1 Threshold derivation process

1: *collect* and *prepare* input data set for each software metrics $SM_{1...i}$
2: **for each** software metric $SM_{1...i}$ **do**
3: *collect* descriptive statistics
4: *obtain* kurtosis and skewness
5: *analyze* collected data set
6: *verify* normal distribution
7: **end for**
8: **for each** software metric $SM_{1...i}$ **do**
9: *find* best fitted distribution
10: *verify* heavy tail distribution
11: **end for**
12: **for each** software metric $SM_{1...i}$ **do**
13: *derive* threshold value $T_{1...i}$
14: **if** distribution *equals* power law **then**
15: *determine* thresholds value $T_{1...i}$ regarding the distribution
16: using 70th percentile *determine* low risk area
17: using 70th and 80th percentile *determine* moderate risk area
18: using 80th and 90th percentile *determine* high risk area
19: using 90th percentile *determine* very high risk area
20: **else**
21: *determine* thresholds value $T_{1...i}$ regarding the distribution
22: using threshold values *determine* risk area
23: **end if**
24: **end for**

Filo et al. [12] complemented the naming of areas proposed by Ferreira et al. [10] to increase the understanding of the derived thresholds. The proposed names reveal the frequency of use of values within each area, whereas we propose renaming the areas to express the risk each area represents within the context of quality evaluation. The naming was suggested by Alves et al. [1] and is based on the risk perspective. We propose the following naming of the fied areas:

- *low* risk, $\leq$70th,
- *moderate* risk, $>$70th and $\leq$80th,
- *high* risk, $>$80th and $\leq$90th and
- *very high* risk, $>$90th.

The formed areas express the risk that an evaluated program entity includes irregularities in the context of different smells, specific structural problems, or potential deficiencies. The low risk area is determined with the 70th percentile, therefore, coinciding with the good/common area, as proposed by Filo et al. [12], indicating the absence of structural problems in a class. On the other hand, if a very high risk area coincide with a bad/uncommon area, this indicates the existence of design problems within the chosen software entity. The described risk areas were used in the study by Beranič et al. [5] for the detection of deficient source code. The study detects deficient software entities based on the combination of quality aspects, increasing the reliability of the identification. Since the use of only one software metric covers a single quality dimension, the use of a combination is crucial for reliable results. The expert judgment performed within the evaluation

of the proposed approach confirmed that software entities that have the majority of metric values in the area of very high risk, are evaluated accurately as deficient.

3.2. Selection of tools and data set for the empirical study

Within threshold derivation, one of the well-known challenges is how to provide comparable results between selected programming languages, and this constitutes precisely the key driver of our research. We aimed at providing threshold values for the same software metrics in four different programming languages. To overcome the distinctive definitions of software metrics within various software metric tools, the input data set into the derivation process should be gathered using a single software metric tool. Based on the performed analysis, we chose an Understand tool [35], that supports the collection of software metric values for multiple programming languages [36]. With this, the risk was addressed and eliminated of providing varying, tool dependent values for the same metric. Besides, collecting values with a single software metric tool enables a more objective comparison among different programming languages.

In the threshold derivation approach, the statistical properties of the input data set were assessed with an SPSS tool [16], and by using R [32]. The best fitted distribution according to the input data set was found with the EasyFit tool [26], and the fication of thresholds was obtained from software metric values using the SPSS tool [16].

The second major challenge that has to be addressed when calculating thresholds using a benchmark data approach is the input data set. To provide reliable thresholds, the input data has to be diverse, extensive and transparent. As Lochmann [23] found out, when the input data set is larger, the diversity of areas and the variance of results decreases correspondingly. Therefore, a large benchmark base reduces the impact of randomly selected software products [34,23]. To determine the optimal size of input data set, we reviewed studies deriving threshold values from benchmark data. The number of software projects used by each study is presented in Table 2.

Study	Number of software projects
Alves et al. [1]	100
Arar and Ayan [2]	10
Ferreira et al. [10]	40
Filo et al. [12]	111 (from Qualitas Corpus [40])
Fontana et al. [13]	74 (from Qualitas Corpus [40])
Mori et al. [28]	3,107
Oliveira et al. [30]	106 (from Qualitas Corpus [40])
Yamashita et al. [44]	4,780
Vale et al. [42]	103 (from Qualitas Corpus [40])

Table 2. Number of projects used as benchmark data within the threshold derivation process

Oliveira et al. [30], Fontana et al. [13], Filo et al. [12] and Vale et al. [42] used projects from Qualitas Corpus [40], whereas Alves et al. [1] used 100 software products, including open source and proprietary solutions. Yamashita et al. [44] also used the combination of open source and industrial software solutions, wherein 205 projects were proprietary

and 4,757 projects were open source projects. Mori et al. [28] included 3,107 software systems divided into 15 domains, since their focus was to analyze the impact of domains on derived threshold values. On the other hand, Ferreira et al. [10] and Arar and Ayan [2] used a smaller benchmark, including 40 and 10 projects, respectively.

Summarizing the collected numbers, we decided to use 100 software projects and use the obtained values of software metrics as an input data set for threshold derivation. Since we derived thresholds for the programming languages Java, C++, C# and Python, the use of Qualitas Corpus collection [40] was not possible. The collection combines software developed in the Java programming language, and, not knowing the conditions by which the software projects were chosen, a comparable suite for the other three programming languages could not be gathered.

We formed a reusable suite of software products that enables repeatability, and contributes to the objectivity of the presented empirical research. The suite includes 400 software projects, 100 in each of the selected programming languages. The list of used software products is available at: https://bit.ly/2RIQhle. Since software projects were chosen and gathered systematically, it allows a reliable comparison of derived thresholds.

In the implemented study, only open source software was used. We collected the input software solution from SourceForge [39], that allows categorization of software projects using different criteria, e.g., programming language, operating system, license, user interface and others. Also, software projects are categorized into different thematic domains, where each domain includes a different number of projects. Therefore, the ratio within every programming language was transferred to the selected sample of 100 projects to keep a ratio of the population. The impact of application domains on benchmark data set values was studied by Ferreira et al. [10]. Thresholds were derived using the benchmark data from 11 application domains. The results show that software metric values follow the same probability distribution, regardless of the application domain, and that there is only a slight difference between the thresholds derived for the used domains and the thresholds derived using a full data set [10]. As they conclude, regardless of the observed minor differences, the general results can be used for all application domains [10].

Within the scope of our experimental study, the selection of software projects was performed in several steps. In the first step, we applied the programming language filter, therefore, four lists were formed, a list of Java, C++, C# and Python projects. In the second step, we applied different filters to the project lists to fy only those projects that are regularly maintained and stable, which suggests that they follow best software development and maintenance practices. We considered criteria related to status and freshness, which were chosen by using the following filters: (1) status - production/stable, (2) freshness - recently updated. The third step sorted the filtered software projects by their popularity in descending order, forming another filter for fication of stable projects. In the fourth step, ordered and filtered lists were divided into different thematic domains using a category filter, and producing a final input list into the software selection step. In the fifth step, the corresponding number of software projects were chosen from each category, considering the ratio accessible in the population. The data about each project were documented, and the actual version of source code was downloaded. Each project got a unique fication key that allows for traceability across a derivation approach.

The descriptive statistic of the established reusable suite is presented in Table 3. It presents the statistics for selected software projects for each of four programming lan-

	# files	# classes	# lines	# code lines
		Java		
Min	12	12	1,511	957
Max	18,469	22,837	4,897,144	2,823,916
Avg	1,436.1	2,266.3	314,915.9	184,994.5
		C++		
Min	6	1	1,236	687
Max	26,517	34,718	10,032,886	5,996,402
Avg	1,644.8	987.4	542,526.0	312,326.8
		C#		
Min	5	4	570	378
Max	7,656	10,422	1,311,745	857,729
Avg	529.3	686.4	124,241.2	80,875.4
		Python		
Min	6	1	2,139	1,114
Max	4,167	10,085	915,078	583,278
Avg	287.8	587.9	82,126.5	54,830.1

Table 3. Descriptive statistics of selected software projects in a reusable suite

guages, presenting the minimum, maximum and average values of the number of files, classes, the total number of lines, and number of lines of code.

Since the methodology of the selection of software products is presented and documented systematically, it can be repeated, and, with this, the formed suite of software projects can be extended to other programming languages.

3.3. Calculation of threshold values

The focus of our research was on deriving threshold values for class level software metrics. As presented in Chapter 3.2, a reusable suite of software products was established, including 400 software products, 100 for each of the selected programming languages. For every software project in the suite, software metrics were collected with the Understand tool [35]. Input files were prepared according to guidelines presented by the replicated approach [10], wherein the input file for the Java programming language included 206,730 records, the file for C++ 98,762 records, the file for C# had 81,293 records and the input file for Python included 60,462 records.

The Understand tool [35] allows for the collection of 102 software metrics, evaluating different levels. Our study is limited to Java, C++, C# and Python. Since all software metrics are not supported in all programming languages, meaning that the support for Python is limited, we decided to calculate thresholds for nine software metrics:

– *CountLineCode* - number of lines of code in a class,
– *AvgLineCode* - average size of methods in a class in lines of code,
– *SumCyclomatic* - the sum of cyclomatic complexities for all the methods in a class,
– *AvgCyclomatic* - the average value of cyclomatic complexity in the methods in a class,
– *MaxNesting* - the maximal nesting level in a class,
– *CountClassCoupled* - number of classes to which a class is coupled,
– *PercentLackOfCohesion* - the lack of cohesion in a class,
– *CountDeclMethodAll* - number of methods in a class, including inherited ones, and
– *MaxInheritanceTree* - the maximum depth of a class in the inheritance hierarchy.

Metrics evaluating size and complexity are probably the most widely used software measurements [19]. Size-related software metrics are aimed to quantify the size of a software [37], for example, the metric *CountLineCode* expresses the number of lines of source code in a chosen software class, excluding blank lines and comment lines, and *AvgLineCode* expresses the average method size in a class, expressed with the number of lines of code. Related to the latter, is also the metric *CountDeclMethodAll*, counting all the methods within a class, taking into account inherited methods [36]. McCabe [27], in 1976, introduced a complexity measure known as Cyclomatic Complexity. *SumCyclomatic* and *AvgCyclomatic* are measuring complexity, wherein *SumCyclomatic* expresses the sum of cyclomatic complexities of all the methods in a class, and metric *AvgCyclomatic* gives an overview, expressing the average value of cyclomatic complexity of all the methods in a class. Another aspect affecting the complexity of a program entity is covered by metric *MaxNesting* [46], expressing the maximal nesting level in a class.

In addition to the above-mentioned software metrics, thresholds were also derived for different object-oriented software metrics. Chidamber and Kemerer [6] proposed a metrics suite aimed at measuring specific object-oriented properties. Among others, they define a metric measuring coupling between object classes, a metric expressing lack of cohesion in methods, and a metric expressing depth of the inheritance tree. The latter are implemented in Understand tool [35] as *CountClassCoupled*, *PercentLackOfCohesion* and *MaxInheritanceTree*, respectively. The software metric *CountClassCoupled* measures the coupling of a class to other classes. Two classes are coupled if one class uses the methods and variables defined in another class [6]. High coupling is not desirable, since the reuse is difficult because of decreased modularity [37]. On the other hand, the software metric *PercentLackOfCohesion* expresses the lack of cohesion in a class. High cohesion means that methods and attributes cooperate with each other and form a logical whole [25]. The lack of cohesion may suggest that a class should be divided [37]. One of the advantages of object-oriented design is the reuse of program entities. We can form classes that inherit functionalities from their parent class [19]. Software metric *MaxInheritanceTree* expresses the maximum depth of a class in an inheritance hierarchy. The *MaxInheritanceTree* of the root node is 0 [36]. In a case of multiple inheritances, the metric expresses only the maximum length from the class node to the root of the inheritance tree [6]. The deeper the class is in a hierarchy, the more methods could be inherited, which, consequently, increases the complexity in the design [37].

For Java, C++ and C#, thresholds were derived for nine different software metrics, and for Python, thresholds were derived for seven software metrics, since the metrics *CountClassCoupled* and *PercentLackOfCohesion* are not supported by the used software metric collection tool. Following the approach presented by Ferreira et al. [10], the threshold derivation approach starts by checking the distribution of the input data set, which is to say, by finding the best-fitted distribution. First, it was checked to see if data are distributed normally with the use of descriptive statistics. The latter was used to confirm the power law distribution in the data set by Shatnawi and Althebyan [38]. Within normal distributions, the values are centralized strongly around the arithmetic mean, meaning that the latter presents a representative value that a random variable can occupy [38]. For each software metric in each of the selected programming languages, we gathered values for the arithmetic mean, median, standard deviation and maximal value. Also, the values of kurtosis and skewness were obtained, that enable an insight into data distribution and in-

	CountLineCode				*AvgLineCode*			
	Java	C++	C#	Python	Java	C++	C#	Python
kurtosis	2,439.1	5,911.9	15,921.9	834.9	17,284.6	310.7	6,056.9	5,966.7
skewness	32.1	60.2	113.7	19.6	91.2	10.7	64.7	59.3
arithmetic mean	88.3	114.9	136.9	58.6	9.4	7.8	9.5	7.6
median	27	29	41	17	6	4	5	4
standard deviation	283.4	568.8	995.8	166.6	18.9	12.3	28.9	19.0
maximum	36,273	74,278	156,163	11,798	4,312	666	3,414	2,159
kurtosis	leptokurtic				leptokurtic			
skewness	positive				positive			

	SumCyclomatic				*AvgCyclomatic*			
	Java	C++	C#	Python	Java	C++	C#	Python
kurtosis	4,182.6	11,670.1	24,038.2	677.3	1,297.9	783.2	7,018.4	348.8
skewness	45.7	84.3	148.8	17.8	21.8	18.5	68.3	12.9
arithmetic mean	14.8	22.5	18.2	13.7	1.7	1.9	1.6	1.7
median	4	6	5	4	1	1	1	1
standard deviation	52.1	125.9	208.5	40.1	2.2	3.5	4.3	2.7
maximum	7,026	21,581	34,702	2,430	206	228	524	150
kurtosis	leptokurtic				leptokurtic			
skewness	positive				positive			

	MaxNesting				*CountClassCoupled*			
	Java	C++	C#	Python	Java	C++	C#	Python
kurtosis	5.3	3.2	3.5	3.3	522.4	140.1	58.3	n/a
skewness	1.9	1.6	1.7	1.6	12.0	10.1	4.9	n/a
arithmetic mean	1.1	1.2	1.3	1.2	4.8	6.6	9.9	n/a
median	1	1	1	1	2	3	6	n/a
standard deviation	1.4	1.6	1.6	1.5	8.4	14.2	12.7	n/a
maximum	21	16	18	16	704	328	403	n/a
kurtosis	leptokurtic				leptokurtic			
skewness	positive				positive			

	PercentLackOfCohesion				*CountDeclMethodAll*			
	Java	C++	C#	Python	Java	C++	C#	Python
kurtosis	-1.4	-1.6	-1.6	n/a	2,699.3	248.7	11,728.9	7.8
skewness	0.5	-0.1	0.2	n/a	22.8	13.3	82.7	2.6
arithmetic mean	32.8	48.9	37.2	n/a	20.21	43.9	30.8	27.24
median	0	52	33	n/a	5	16	16	11
standard deviation	38.1	40.8	37.2	n/a	46.8	128.1	90.5	41.3
maximum	100	100	100	n/a	7,113	3,776	14,246	464
kurtosis	mesokurtic				leptokurtic			
skewness	symmetric				positive			

	MaxInheritanceTree			
	Java	C++	C#	Python
kurtosis	655.5	2.7	422.1	1.1
skewness	8.4	1.4	14.5	1.0
arithmetic mean	1.7	1.2	0.9	1.8
median	1	1	1	1
standard deviation	1.1	1.2	1.5	1.8
maximum	118	11	58	10
	leptokurtic			
	positive			

Table 4. Descriptive statistics of analyzed software metrics values

dicate a deviation from the normal distribution. Kurtosis expresses the size of peaks and skewness measures the symmetry of the used data set. Data that are normally distributed have a value of kurtosis and skewness of approximately zero [11]. When the values move away from zero it proves that they are not following a normal distribution and values are gathered on one end of the scale, and values are distributed in a peak or are flattened. Data that follow heavy tailed distributions have a positive skew. For the positive skew, also apply [38]:

$$\begin{gathered} \textit{standard deviation} \gg \textit{arithmetic mean} \\ \textit{standard deviation} \gg \textit{median} \end{gathered} \tag{1}$$

$$\textit{maximum} \gg \textit{arithmetic mean} \tag{2}$$

Besides the mentioned, the positive skew is indicated by:

$$\textit{arithmetic mean} \geq \textit{median} \geq \textit{mode} \tag{3}$$

Descriptive statistics for the evaluated metrics are presented in Table 4. Software metric values are limited to the left, with a value 0, and unlimited to the right, since the maximum value is usually not defined [13]. Among the gathered values, data describing the metric *PercentLackOfCohesion* that expresses a lack of cohesion in a class, stand out. The metric can occupy a value from 0 to 100, since the result is expressed in percentages. Based on the numbers, it is the only metric for which descriptive statistics do not discard normal distribution. Other software metrics, without a doubt, do not follow a normal distribution, as reflected by their positive skew and leptokurtic distribution. Namely, when the values of skewness are more than 0, a positive skew is present, which is reflected with values gathered on the left, and individual values on the right that form a tail [11,38]. On the other hand, the positive value of kurtosis is shown in a bigger peak of distribution, and indicates that the values are forming a heavy tail [11].

After the descriptive statistics were analyzed, the best fitted distribution for data was determined using an EasyFit tool [26]. More than 55 distributions are available, and the tool checks how well a chosen distribution fits an input data set, and arranges them according to performance. Table 5 presents the best-fitted distributions for selected software metrics values in four programming languages.

The threshold values were determined after the data distribution was determined for each software metric in all of the four programming languages. All software metrics, except *PercentLackOfCohesion*, correspond to a heavy tail distribution. Because of this, the derivation could be done as proposed by Ferreira et al. [10], by using percentiles, and considering the proposed adjustments related to risk areas. As suggested in 3.1, thresholds were determined using two or three percentiles using an SPSS tool [16].

The values of software metrics *AvgCyclomatic*, *MaxNesting* and *MaxInheritanceTree* are presented within a small range of data. For example, the metric *MaxNesting* has the same value for the 70th and 80th percentile for C++ and C#. Because forming the area with such small differences between the borders is not feasible, the 80th percentiles was excluded and only the 70th and 90th percentiles were used for forming the threshold risk areas. The metric *PercentLackOfCohesion* follows a Uniform distribution, and thresholds cannot be determined using percentiles. For this purpose, the threshold value was determined using the arithmetic mean and standard deviation.

Software metric	Programming language	Distribution
CountLineCode	Java	Inverse Gaussian
	C++	Dagum
	C#	Pareto 2
	Python	Wakeby
AvgLineCode	Java	Generalized Pareto
	C++	Generalized Pareto
	C#	Generalized Logistic
	Python	Generalized Logistic
SumCyclomatic	Java	Phased Bi-Weibull
	C++	Generalized Pareto
	C#	Generalized Pareto
	Python	Generalized Pareto
AvgCyclomatic	Java	Generalized Logistic
	C++	Wakeby
	C#	Phased Bi-Exponential
	Python	Phased Bi-Exponential
MaxNesting	Java	Gumber Max
	C++	Gumber Max
	C#	Gumber Max
	Python	Gumber Max
CountClassCoupled	Java	Phased Bi-Weibull
	C++	Generalized Logistic
	C#	Wakeby
	Python	n/a
PercentLackOfCohesion	Java	Uniform
	C++	Uniform
	C#	Uniform
	Python	n/a
CountDeclMethodAll	Java	Wakeby
	C++	Wakeby
	C#	Generealized Pareto
	Python	Johnson SB
MaxInheritanceTree	Java	Gamma
	C++	Gumber Max
	C#	Logistic
	Python	Johnson SB

Table 5. Best fitted distributions

4. Empirical analysis of derived threshold values

Based on derived threshold values, calculated points were used to set three or four risk areas. Thresholds are presented in the form of areas.

Areas, as determined in Chapter 3.1, are formed according to the risk that an evaluated program entity includes irregularities. For example, if a class has 300 lines of code, a very high risk (VHR) exists that something within the entity is not optimal. This does not mean that defects are present, but that there may be some irregularities in the context of different smells or specific technical debts. However, we have to be aware, that combining different software metrics when evaluating software quality could improve the reliability of provided results significantly. Values lower than the 70th percentile belong to a low risk (LR) area, values between the 70th and 80th percentiles form a moderate risk (MR) area, and values bigger than the 80th percentile and smaller, or equal to the 90th percentile

		Java	C++	C#	Python
CountLineCode	LR	x⩽61	x⩽66	x⩽90	x⩽43
	MR	61<x⩽100	66<x⩽112	90<x⩽144	43<x⩽71
	HR	100<x⩽197	112<x⩽235	144<x⩽278	71<x⩽135
	VHR	x>197	x>235	x>278	x>135
AvgLineCode	LR	x⩽9	x⩽8	x⩽10	x⩽8
	MR	9<x⩽13	8<x⩽11	10<x⩽14	8<x⩽11
	HR	13<x⩽19	11<x⩽18	14<x⩽20	11<x⩽17
	VHR	x>19	x>18	x>20	x>17
SumCyclomatic	LR	x⩽10	x⩽13	x⩽11	x⩽9
	MR	10<x⩽17	13<x⩽22	11<x⩽18	9<x⩽16
	HR	17<x⩽33	22<x⩽45	18<x⩽36	16<x⩽33
	VHR	x>33	x>45	x>36	x>33
AvgCyclomatic	LR	x⩽2	x⩽2	x⩽1	x⩽2
	MR	2<x⩽3	2<x⩽4	1<x⩽3	2<x⩽4
	HR	x>3	x>4	x>3	x>4
MaxNesting	LR	x⩽1	x⩽2	x⩽2	x⩽2
	MR	1<x⩽3	2<x⩽3	2<x⩽4	2<x⩽3
	HR	x>3	x>3	x>4	x>3
CountClassCoupled	LR	x⩽5	x⩽6	x⩽11	n/a
	MR	5<x⩽7	6<x⩽9	11<x⩽15	n/a
	HR	7<x⩽11	9<x⩽14	15<x⩽23	n/a
	VHR	x>11	x>14	x>23	n/a
PercentLackOfCohesion	LR	x⩽71	x⩽90	x⩽74	n/a
	HR	x>71	x>90	x>74	n/a
CountDeclMethodAll	LR	x⩽14	x⩽42	x⩽26	x⩽24
	MR	14<x⩽24	42<x⩽49	26<x⩽34	24<x⩽51
	HR	24<x⩽51	49<x⩽90	34<x⩽60	51<x⩽70
	VHR	x>51	x>90	x>60	x>70
MaxInheritanceTree	LR	x⩽2	x⩽2	x⩽1	x⩽2
	MR	2<x⩽3	2<x⩽3	1<x⩽2	2<x⩽4
	HR	x>3	x>3	x>2	x>4

Table 6. Risk areas (low risk (LH), moderate risk (MR), high risk (HR) and very high risk (VHR)) based on threshold values

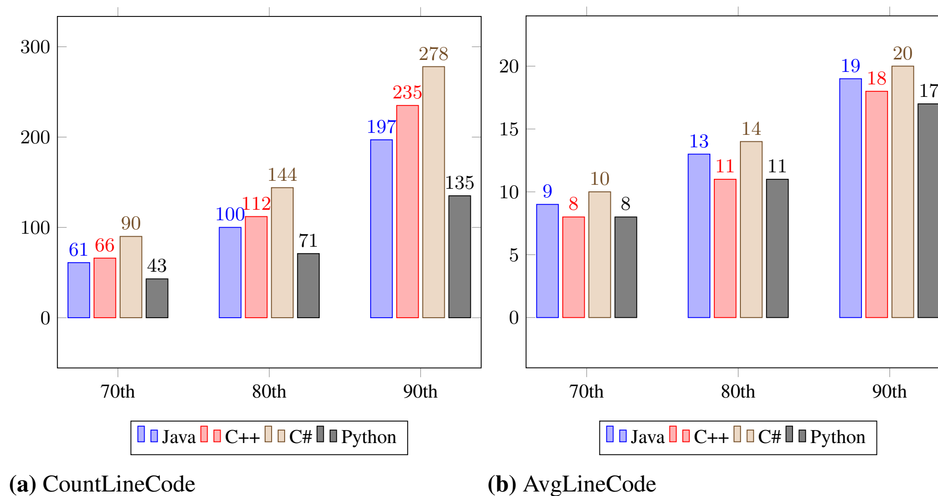
constitute high risk (HR) area, and values that are bigger than determined with the 90th percentile are considered to be in the area of very high risk (VHR). Table 6 presents the defined risk areas and corresponding threshold values. The values are shown for nine software metrics in four programming languages. Where areas are determined with only two percentiles, i.e. in the case of *AvgCyclomatic*, *MaxNesting* and *MaxInheritanceTree*, only three areas are given. Values lower than those determined with the 70th percentile are in the area of low risk (LR), between the 70th and 90th percentile there is a moderate risk (MR) area, and values in a high risk (HR) area are values bigger than determined with the 90th percentile. A special case is the metric *PercentLackOfCohesion*, where only one area is defined, based on the calculated threshold value. Values that are bigger than the threshold are in the area of high risk.

As presented within the related work in section 2, different threshold derivation approaches exist. To allow the comparison, the threshold values have to be derived using the same benchmark data, the software metric tools with coincidental definitions of implemented software metrics, and, finally, using the same threshold derivation approach. Therefore, comparison of our results with threshold values provided by Ferreira et al. [10]

or Filo et al. [12] in a meaningful way is not possible, due primarily to use of different software metrics' definitions, followed by varying input data.

4.1. Comparison of derived threshold values

Figures 1, 2, 3 and 4 present threshold values for the 70th, 80th and 90th percentiles of the same software metrics for different programming languages. A visual comparison of threshold values for Java, C++, C# and Python is enabled with this. Furthermore, since the approach for the threshold derivation is based on the frequency of values within the software, the results also indicate the structure of software written in the four selected programming languages.



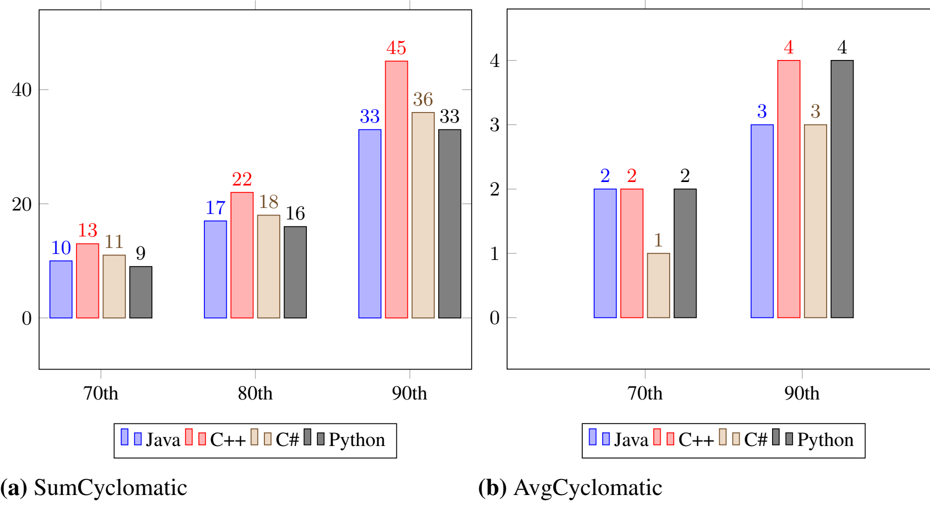
(a) CountLineCode **(b)** AvgLineCode

Fig. 1. Threshold values of the software metrics *CountLineCode* and *AvgLineCode*

Figure 1 illustrates risk areas and threshold values of the 70th, 80th and 90th percentiles for software metrics *CountLineCode* and *AvgLineCode* measuring lines of code in a software class. Axis *x* shows programming languages, and axis *y* threshold values. In every figure, there are three lines: green, representing the 70th percentile, orange, representing the 80th percentile, and red, representing the 90th percentile. Three lines form four risk areas, while connecting values of the same percentile value among programming languages. The green color presents a low risk area (LR), orange presents a moderate risk area (MR), light red color stands for a high risk area (HR), and red presents a very high risk area (VHR).

As indicated in Figure 1a, the threshold determining very high risk is the highest in the C# programming language, whereas the smallest is within Python. The same ratio is also between thresholds formed using the 80th and 70th percentiles. Figure 1b plots the threshold values for the average size of methods in a class. The values are closer in comparison to the metric measuring lines of code in a class, but still, values vary. The 90th percentile is again the highest for C# and the lowest for Python, whereas the values for

Java and C++ are in between. The derived values show that the most extensive software classes can be found in the C# programming language, followed by C++, Java and Python classes. Given the small difference in the average size of methods within a class, we can conclude that software classes written in C# possess more methods than classes developed in the Python programming language.



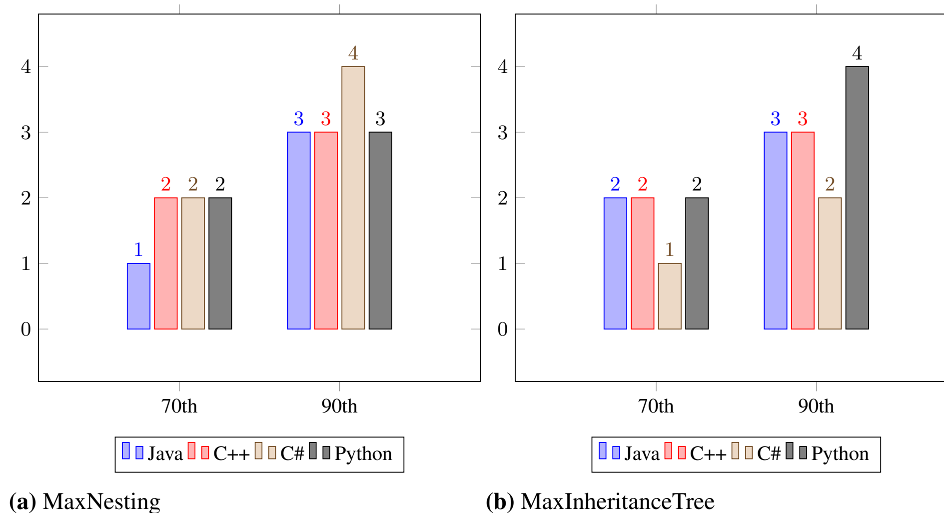
(a) SumCyclomatic **(b)** AvgCyclomatic

Fig. 2. Threshold values of the software metrics *SumCyclomatic* and *AvgCyclomatic*

Figure 2 presents the threshold values of the software metrics *SumCyclomatic* and *AvgCyclomatic*. Figure 2a visualizes the thresholds for the sum of cyclomatic complexities for all the methods in a class. The highest threshold value is derived for the programming language C++, followed by C#, Java and Python. A noticeable leap can be detected in C++, whereas in other programming languages, the threshold values are rising more gradually. If we consider a number of methods in classes expressed with software metrics *CountLineCode* and *AvgLineCode*, we can conclude that methods written in Python have the highest cyclomatic complexity, whereas the smallest complexity is present in methods developed in C#. This is also confirmed with threshold values for the metric *AvgCyclomatic* in Figure 2b, presenting the average value of the methods in a class. The *AvgCyclomatic* is also one of the metrics where derived threshold values are very close. Because of this, the 80th percentile was excluded, and only three risk areas were formed.

Figure 3 presents threshold values for the software metrics *MaxNesting* and *MaxInheritanceTree*. Figure 3a illustrate the thresholds of a software metric measuring the maximum nesting level in a class which affects class complexity. The 70th and 90th percentiles were included, forming three risk areas. As can be seen, the 90th percentile is the highest in C# and the lowest, but coinciding, for Java, C++ and Python. Another software metric that has a thresholds value that is only defined for three areas is *MaxInheritanceTree*, presented in Figure 3b. Based on statistical properties, it is very similar to the metric

MaxNesting. The 90th percentile is the highest within Python and the lowest in C#, meaning that inheritance hierarchy is the deepest in software projects developed in Python.

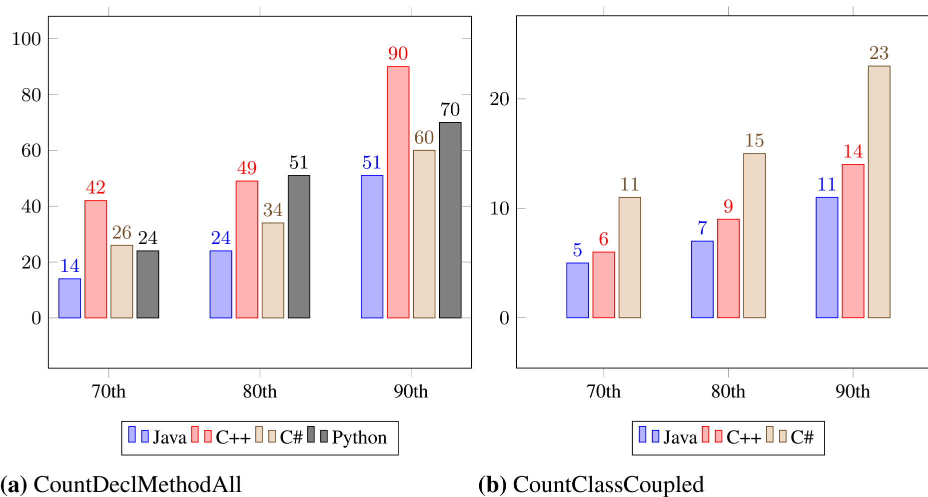


(a) MaxNesting **(b)** MaxInheritanceTree

Fig. 3. Threshold values of the software metrics *MaxNesting* and *MaxInheritanceTree*

Closely connected to the mentioned metric is also the metric *CountDeclMethodAll*, counting methods in the classes, including inherited ones. The threshold values are presented in Figure 4a. With a 90th percentile value, the C++ threshold set is the biggest, followed by values derived for Python, C# and the Java programming language. If we connect the findings to the determined number of methods based on software metrics *CountLineCode* and *AvgLineCode*, we can see that the ranking by values is different, since the metric *CountDeclMethodAll* also considers inherited methods. As presented in Figure 4a, the highest number of methods can be detected in C++, which is due to a bigger inheritance hierarchy, as presented in Figure 3b. A high number of methods, according to the metric *CountDeclMethodAll*, can also be found with Python classes, though they have the smallest metric value, based on lines of code. Again, this is due to a deeper inheritance hierarchy. On the other hand, Java classes have fewer methods than classes developed in Python, according to *CountDeclMethodAll*, but based on the number of lines of code, the case is different. Since the inheritance hierarchy for Java is smaller, this is the logical conclusion.

The threshold values for the software metric *CountClassCoupled* are presented in Figure 4. The mentioned metrics were not calculated for Python, since the Understand tool [35] does not support that calculation. Therefore, the results are only presented for Java, C++ and C#. Figure 4b presents threshold values for metric measuring coupling with other classes. The threshold defining the area of the very high risk is the biggest for C# classes. The lowest is the threshold value for Java, whereas the programming language C++ is in between. Thresholds indicate that C# classes are the most coupled with others, which can be related to a large number of lines of code, as seen in Figure 1. On the other



(a) CountDeclMethodAll **(b)** CountClassCoupled

Fig. 4. Threshold values of the software metrics *CountClassCoupled* and *PercentLackOfCohesion*

hand, the coupling is lowest for Java classes. Another aspect that can impact the use of coupling is also the age of the projects. Since the C# programming language is much younger that C++, the age could influence the threshold values.

The *PercentLackOfCohesion* metric measures the lack of cohesion in classes, and is presented with percentages. The threshold is presented only with one value, that divides the area into low and high risk, since the threshold was not derived based on percentiles. The values can be seen in Table 6. The defined threshold values again vary between programming languages, and are the highest for C++, where classes that exceed the value 90 present the high risk of containing irregularities. Within C#, the threshold value is the highest, while the values indicate that the C++ classes are least cohesive, which could be connected to the high cyclomatic complexity of classes and methods with a large number of lines. On the other hand, classes in C# are also large, but with a lower cyclomatic complexity, which is reflected in class cohesion. The connection to cyclomatic complexity can also be confirmed for the Java programming language, where cohesion is better and complexity lower.

As can be concluded based on the presented analysis, the threshold values of software metrics vary between different programming languages. Therefore, they have to be calculated for each programming language separately.

5. Limitations

In this research, some limitations and potential threats to validity arise. They are presented here.

We limited ourselves to object-oriented programming languages and software metrics supported by the used tool. The results may be affected by the tool used for collecting metric values and the corresponding implementation of software metrics. To reduce the threat,

a single tool was used throughout the entire research, and for all four of the programming languages. In this way, software metrics were calculated in the same way, regardless of the programming language.

The results can also be affected by the approach used for deriving the threshold values of software metrics. With the use of only a single approach for all the metrics and all the programming languages, the risk of providing inconsistent results was limited. Another threat surrounds the benchmark data used for derivations. To limit the impact, the data set was collected in a transparent and systematic way, covering a broad scope of different properties. Also, the size of the benchmark data was determined based on related work and good practices.

The definitions of software metrics present a limitation within the research. The validation of software metrics was not a part of the presented study.

6. Conclusion

Quantification with software metrics is important, especially when we make decisions related to software quality [1,9], thereby knowing that the reliable thresholds are crucial. Within the presented empirical study, threshold values were derived for nine software metrics for four object-oriented programming languages, namely Java, C++, C# and Python. Using the replicated threshold derivation approach and proposed adjustments, threshold values were derived considering challenges arising in the software metrics domain. Since the approach uses benchmark data, the latter were collected systematically and transparently, allowing repeatability and supplementation. For each programming language, a suite of 100 software projects was selected, which is, according to related work, an optimal number. Input values were gathered using a single software metric tool, and threshold values were provided using a single threshold derivation approach by following well-defined steps.

The main research question driving the presented study was *if software metric threshold values vary between different object-oriented programming languages*. By this, we could provide information about whether thresholds have to be derived for each programming language separately, or if a single threshold can be applied to all programming languages. Thresholds derived for a particular software metric were analyzed and compared to provide the answer. Based on the findings, we can conclude that threshold values for the same software metric vary among different programming languages. This can be attributed to different structural properties for programming languages, and established practices used in a specific community. Therefore, the derivation for each programming language has to be done separately.

In future work, we plan to use the threshold derivation process to provide threshold values for other programming languages, and expect to derive threshold values for software metrics on different levels, i.e. the method and file levels. Also, we will analyze the rules and properties of the different programming languages, in order to explain the reasons for the differences. In addition, we plan to study different factors impacting derived thresholds within each programming language.

Acknowledgments. This work was supported by the Slovenian Research Agency (SRA) under The Young Researchers Program (SICRIS/SRA code 35512, RO 0796, Program P2-0057).

References

1. Alves, T.L., Ypma, C., Visser, J.: Deriving metric thresholds from benchmark data. In: 2010 IEEE International Conference on Software Maintenance (2010)
2. Arar, Ö.F., Ayan, K.: Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies. Expert Systems with Applications 61, 106 – 121 (2016)
3. Benlarbi, S., Emam, K.E., Goel, N., Rai, S.: Thresholds for object-oriented measures. In: Proceedings 11th International Symposium on Software Reliability Engineering (ISSRE 2000) (2000)
4. Beranič, T., Heričko, M.: Approaches for software metrics threshold derivation: A preliminary review. In: Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications SQAMIA 2017, Proceedings (2017)
5. Beranič, T., Podgorelec, V., Heričko, M.: Towards a reliable identification of deficient code with a combination of software metrics. Applied Sciences 8(10) (2018)
6. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (Jun 1994)
7. Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-laws in a large object-oriented software system. IEEE Transactions on Software Engineering 33(10), 687–708 (Oct 2007)
8. Dósea, M., Sant'Anna, C., da Silva, B.C.: How do design decisions affect the distribution of software metrics? In: Proceedings of the 26th Conference on Program Comprehension (2018)
9. Fenton, N.E., Neil, M.: Software metrics: Roadmap. In: Proceedings of the Conference on The Future of Software Engineering (2000)
10. Ferreira, K.A., Bigonha, M.A., Bigonha, R.S., Mendes, L.F., Almeida, H.C.: Identifying thresholds for object-oriented software metrics. Journal of Systems and Software 85(2), 244 – 257 (2012)
11. Field, A.: Discovering Statistics Using IBM SPSS Statistics. Sage Publications Ltd., 4th edn. (2013)
12. Filó, T.G.S., da Silva Bigonha, M.A., Ferreira, K.A.M.: A catalogue of thresholds for object-oriented software metrics. In: First International Conference on Advances and Trends in Software Engineering (2015)
13. Fontana, F.A., Ferme, V., Zanoni, M., Yamashita, A.: Automatic metric thresholds derivation for code smell detection. In: IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics (2015)
14. Gerlec, C., Rakić, G., Budimac, Z., Heričko, M.: A programming language independent framework for metrics-based software evolution and analysis. Computer Science and Information Systems 9(3), 1155–1186 (2012)
15. Gil, J.Y., Lalouche, G.: When do software complexity metrics mean nothing? – when examined out of context. Journal of Object Technology 15(1), 2:1–25 (Feb 2016)
16. IBM: Spss. https://www.ibm.com/analytics/data-science/predictive-analytics/spss-statistical-software (2018), accessed: 2. 2. 2018
17. ISO/IEC/IEEE 24765:2017: ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary pp. 1–541 (2017)
18. Kitchenham, B.: What's up with software metrics? – a preliminary mapping study. Journal of Systems and Software 83(1), 37 – 51 (2010)
19. Lanza, M., Marinescu, R.: Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer-Verlag Berlin Heidelberg (2006)
20. Lavazza, L., Morasca, S.: An empirical evaluation of distribution-based thresholds for internal software measures. In: Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering (2016)

21. Lima, P., Guerra, E., Meirelles, P., Kanashiro, L., Silva, H., Silveira, F.F.: A metrics suite for code annotation assessment. Journal of Systems and Software 137, 163 – 183 (2018)
22. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis (2008)
23. Lochmann, K.: A benchmarking-inspired approach to determine threshold values for metrics. SIGSOFT Softw. Eng. Notes 37(6), 1–8 (Nov 2012)
24. Lorenz, M., Kidd, J.: Object-oriented Software Metrics: A Practical Guide. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
25. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edn. (2009)
26. MathWave Technologies: Easyfit. http://www.mathwave.com (2004–2018), accessed: 20. 12. 2017
27. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering SE-2(4), 308–320 (Dec 1976)
28. Mori, A., Vale, G., Viggiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., Jamshidi, P., Kastner, C.: Evaluating domain-specific metric thresholds: An empirical study. In: Proceedings of the 2018 International Conference on Technical Debt (2018)
29. Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E., Soubervielle-Montalvo, C.: Source code metrics: A systematic mapping study. Journal of Systems and Software 128, 164 – 197 (2017)
30. Oliveira, P., Valente, M.T., Lima, F.P.: Extracting relative thresholds for source code metrics. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE) (2014)
31. Oliveira, P., Lima, F.P., Valente, M.T., Serebrenik, A.: RTTool: A tool for extracting relative thresholds for source code metrics. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (2014)
32. R Core Team: R: A language and environment for statistical computing. https://www.R-project.org (2017), accessed: 13. 12. 2017
33. Rakić, G., Budimac, Z., Savić, M., Ivanović, M.: Towards the formalization of software measurement by involving network theory. In: SQAMIA (2015)
34. Remencius, T., Sillitti, A., Succi, G.: Assessment of software developed by a third-party: A case study and comparison. Information Sciences 328, 237–249 (2016)
35. Scientific Toolworks Inc.: Understand^{TM}. https://scitools.com (1996–2018), accessed: 18. 8. 2018
36. Scientific Toolworks, Inc.: Understand, User Guide and Refrence Manual (2017)
37. Sharma, A., Dubey, S.K.: Comparison of software quality metrics for object-oriented system. International Journal of Computer Science & Management Studies (IJCSMS) 12, 12–24 (2012)
38. Shatnawi, R., Althebyan, Q.: An empirical study of the effect of power law distribution on the interpretation of oo metrics. ISRN Software Engineering 2013 (March 2013)
39. SourceForge: https://sourceforge.net (2017), accessed: 16. 8. 2017
40. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC2010). pp. 336–345 (Dec 2010)
41. Vale, G.A.D., Figueiredo, E.M.L.: A method to derive metric thresholds for software product lines. In: 2015 29th Brazilian Symposium on Software Engineering (2015)
42. Vale, G., Fernandes, E., Figueiredo, E.: On the proposal and evaluation of a benchmark-based threshold derivation method. Software Quality Journal pp. 1–32 (May 2018)
43. Veado, L., Vale, G., Fernandes, E., Figueiredo, E.: TDTool: Threshold derivation tool. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (2016)

44. Yamashita, K., Huang, C., Nagappan, M., Kamei, Y., Mockus, A., Hassan, A.E., Ubayashi, N.: Thresholds for size and complexity metrics: A case study from the perspective of defect density. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS) (2016)
45. Zhang, F., Mockus, A., Zou, Y., Khomh, F., Hassan, A.E.: How does context affect the distribution of software maintainability metrics? In: 2013 IEEE International Conference on Software Maintenance. pp. 350–359 (Sept 2013)
46. Zou, Y., Kontogiannis, K.: Migration to object oriented platforms: a state transformation approach. In: International Conference on Software Maintenance, 2002. Proceedings. (2002)

Tina Beranič received a PhD degree in computer science and informatics from the University of Maribor in 2018. She is a Teaching Assistant and a Researcher at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Her research interests are in the area of software quality, especially the domain of software metrics and software metrics thresholds. She is working on their involvement in the quality assessment process.

Marjan Heričko received a PhD degree in computer science and informatics from the University of Maribor in 1998. He is currently a Full Professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor, where he is also the Head of the Institute of Informatics. His main research interests include all aspects of IS development with an emphasis on software engineering, software process improvement, software metrics, and process modeling.

Received: October 12, 2018; Accepted: November 15, 2019.