manuscript No.
(will be inserted by the editor)

A Complexity and Entropy-based Metric to Calculate Source Code Quality

João Pedro Schmitt · Fabiano Baldo

Received: date / Accepted: date

Abstract Developer teams are continuously required to be more productive and efficient during software development to attend market demands. Additionally, despite the software growing complexity pushed by development tasks, these teams are asked to produce code with better quality in order to increase the code readability for future refactoring. This scenario brings the opportunity of proposing new metrics able to qualify a given source code. This kind of metric could help developers to improve their source code and, en hence, mitigate the software quality degradation along the time caused by new requirements and market pressures. Due to this fact, this work proposes a complexity and entropy-based metric used to calculate the quality of a given source code. Besides that, it is presented a plug-in for a Java code editor that implements this metric and compares the quality of existing source codes of the same software project. To validate the metric, it has been selected a development team based on Java stack responsible by one single software product. In this team, a case study was conducted to analyze if the metric is capable to measure the source code quality as well as stimulate the software quality improvement. The obtained results have shown that the proposed metric is suitable to help improving the software quality by its ability of measuring the source code shortcomings.

Keywords Software Quality · Entropy · Cyclomatic Complexity · Source Code Quality

João Pedro Schmitt
Santa Catarina State University - UDESC, Paulo Malschitzki, 200 - Zona Industrial Norte - Joiville, Santa Catarina, Brazil
E-mail: schmittjoaopedro@gmail.com

Fabiano Baldo
Santa Catarina State University - UDESC, Paulo Malschitzki, 200 - Zona Industrial Norte - Joiville, Santa Catarina, Brazil
E-mail: fabiano.baldo@udesc.br

1 Introduction

Despite the growing complexity of source codes along the time caused by new software requirements, development teams are continuously asked to be more productive and efficient to attend the new market demands. This scenario makes the source code gradually loses quality, impacting directly on the developer team productivity and motivation [1]. Therefore, one of the major concerns regarding this scenario is the challenge of keeping source code quality with the successive code updates. This is a common situation due to the intrinsic nature of software development caused by constant software modification, that tends deviate from the original project and therefore turning the source code disorganized [2]. In this context, one important source code quality aspect is the maintainability, that helps the software maintenance along the time by making sure that it will meet the business needs [3].

Source code quality is also related to the software developers' personal behavior. Characteristics like few technical skills, high workload and lack of motivation, influence directly in the source code quality because they reduce the developer' willingness of applying the coding best practices [4]. Today, there are several tools capable of analyzing software source code and extract metrics associated with characteristcs like reusability, modularity, complexity, coupling, cohesion, maintainability and documentation [5]. These metrics are mainly used to support developers during development tasks.

However, the use of isolated tools to extract source code metrics and present these to developers with the objective to motivating them to fix the problems is not a trivial task, and in some cases, the use of separate metrics is not sufficient to guarantee the desired quality. Therefore, one possible way to improve this scenario is by unifying a set of different metrics in a majority function. By this way, we can increase the reliability of the analysis because we are considering more aspects concerning software development [7].

Besides that, we need to think about how to present these metrics results in a friendly way for the developers. A possible approach is unify the results and apply a ranking system, thus we can inform the developers about how they are positioned based on the current team ranking [8]. By doing this in the right way, we can motivate developers to improve the source code by concerning the right attending of the metrics established. Another similar approach is by applying this analysis in an online fashion, where during the development phase, the developers have information about how good that source code is comparing with the overall source codes from the software. This last is similar to the concept applied in Grammarly App [9] used to improve English text producing.

In this context, this work has the following research question: how to measure the quality of source code combining several quality metrics in a quantitative metric in order to compare the developer's coding quality? To answer this question, this work proposes a complexity and entropy-based metric used to calculate the quality of a given source. To validate the metric, a case study was applied in a private company. To perform the assessment, the proposed metric was added into the Web-based Integrated Development Environment (IDE) used by the company development team. Basically, the IDE interface was adapted to inform online how statistically good the source code in development is compared with the other source codes. Additionally, the IDE also shows the coding mistakes and bad practices with hints to help the developers in solving the problems.

The next sections are organized as follows. Section 2 presents some relevant related works. Section 3 presents the complexity and entropy-based metric proposed in this work. Section 4 present a implementation of the metric for Java programming language. Section 5 presents the case study and the results. Finally, section 6 presents the conclusion and future works.

2 Related Works

Software quality evaluation in the early phases of software development is a essential task to improve the software longevity [11]. Therefore, to support good software quality, the use of source code metrics plays an essential role during the development phase and can guide developers to improve the software along the time [12]. The metrics help developers to keep source code quality by supporting several characteristics like: maintainability, traceability, availability, reliability, reusability, testability, and readability [3]. However, the best set of metrics definition to use in a specific environment is not a trivial task.

To help in the metric selection process, Timoteo et al. [5], Briand et al. [6] and Santos et al. [13] presents guidelines to identify what metrics must be selected. Basically, Timoteo et al. [5] states that a good metric should quantify software attributes like reusability, modularity, complexity, coupling, cohesion, maintainability and documentation. Besides that, Briand et al [6] give directions about the main aspects to select a valid metric, which are 1) measurement goal, used to validate if a metric is well-defined in the applied context, 2) experimental hypothesis, used to understand what is expected to be learned with that metric, 3) environmental context, used to validate if a given metric is applicable in a software context, 4) theoretical validation, check if the attributes being measured are well defined and 5) empirical validation, how to validate the fitness of the metric to the software related. While in Santos et al. [13] is given a direction about the metrics selection process for object-oriented (OO) software. The authors presents a systematic mapping about the most used methods and statistical techniques used in internal quality evaluation of source code, they enumerate the top 15 metrics for Object-Oriented software.

Besides the metrics selection, the tools are another important aspect used to automatize the quality evaluation process. Today, there are many tools that support different kinds of metrics analysis. For example, Ludwig et al. [1] conducted a software source code quality analysis in open-source projects from GitHub using a metric extractor plugin. The authors applied the plugin with the best set of metrics suitable for source code unitary analysis.

Concerning software quality evaluation there are various studies available in literature. Alsmadi and Alazzam [14] conducted a study evaluating a group of projects using different quality metrics. They found a strong positive correlation between cyclomatic complexity and popularity from open-source projects, this positive correlation explains the fact that projects with more functionalities tends to have a higher cyclomatic complexity and higher popularity. We known from literature that source code with high cyclomatic complexity is harder to be maintained [15]. In Campos et al. [16] the cyclomatic complexity is used to detect unnecessary source code based on the Control Flow Graph (CFG). The authors validated the proposed method applying this one in an educational center to help students during the classes of software development. The results indicates that the use of tools to suggest source code optimization improved code quality from around 95% of developers. Based on the study of Campos, we can presume that applying valid tools to quantify valid metrics, presents to be a viable choice to improve the source code.

Besides Campos et al. [16], there are studies considering more complex scenarios with development teams and projects, like Melo et al. [8] and Barbosa et al. [10]. Melo et al. [8] presents a gamification method used to improve the developer's engagement in the software project. The method was applied in a source code repository, where for each commit made by the developers, is computed a single ranking metric based on the number of modified lines and in the cyclomatic complexity of the modification. Essentially, this approach gives more reward to developers that reduce the complexity, and less reward for developers that increase the complexity. Besides the proposed metric calculation, the results indicate usefulness in counterbalance the Pareto's principle [17], where developers that worked in 20% of the code that represents 80% of its total complexity should receive more points than the others. Barbosa et al. [10] presents another example of metric calculation applied for teams. In their approach, the authors propose a single metric called *Tolerance to Complexity (τ)* used to compare different technical skills from different teams. The calculation is based on issues about system bugs and in the cyclomatic complexity of the system before and after the issues being fixed. The metric was used to classify teams in different tolerance to complexity categories and is useful to find teams with low tolerance to complexity that could be trained in software development.

Based on works from [8] and [10], we can presume that cyclomatic complexity is useful to indicate teams or regions of the software to be improved. Thought the cyclomatic complexity presents to be a suitable metric as observed in [8], [10], and [16], in many cases we need to consider more aspects from software development like cohesion, coupling, structural and logical patterns to improve the quality evaluation. In this context, various studies are applying the use of entropy to consolidate different aspects from software in a single metric to measure the overall software disorganization degree. The entropy is a interesting metric, because as found by Ray et al. [18], source code with bugs tends to be have higher entropy, and higher number of bugs indicates poor software quality.

From literature the first example that uses entropy to measure software quality is presented by Turno et al. [2]. The authors investigate the second law of software evolution, which states that software entropy increases with the time while new requirements, improvements and bugs fixes are implemented, and to keep a small entropy in the software, is necessary to take constantly efforts to improve source code quality. To support the second law analysis, the results indicate that for sufficient releases number, the number of bugs and the entropy calculation value have a strong correlation. In Zhang et al. [19] is proposed a method combining cross-entropy and neural networks to predict software defects. The idea is to complement the set of metrics with a new metric that measures code naturalness. The results indicated effectiveness in predict software defects. In Parveen [20] is proposed a multilinear regression model using entropy based on code changes to predict bugs and release time in software releases. The scenarios considered releases from software's in GitHub and bugs registered in BugZilla.

Considering refactoring purposes, Nyamawe et al. [21] proposes an entropy metric to indicate the best refactoring solutions from a pool of possible solutions given a piece of source code. This metric fixes the problem of the cohesion and coupling in support the traceability between source code and requirements. In the study, the Java language was used to validate the metric over a dataset with source codes composed by refactoring tasks, and the results presented that in 71% of cases the developers had preferred the suggested solutions. Additionally to refactoring purposes, entropy has been used to prevent the software quality degradation trending. Gupta [22] proposes a mathematical method to suggest bad-smells prediction based on entropy and regression models. The study aims to predict possible bad-smells before the development step. To support the study, the software Apache Abdera was applied to validate if the prediction results were consistent with the metric calculations. The scenarios had found that the squared error from the regression line prediction is around 53% with outliers, and leaving out the outliers, the squared error changes do 93%.

Another entropy application is their usage to calculate source code development complexity across different programming languages. Cholewa [23] applies this concept, using the Shannon entropy, to compare the complexity of several programming languages in coding algorithms. The study assumes that languages with less entropy are easier to code than others with higher entropy. This hypothesis is based on the assumption that in languages with less entropy the developer needs to absorb less knowledge about the language structure to produce a functional algorithm. The qualitative results had presented a very suitable classification related to programming languages, where the SmallTalk language was classified as optimal.

About the robustness of entropy measure, Akundi et al. [24] conducted a study applying an entropy-based method to calculate the software complexity degradation along the time and test its stability against the eight Weyuker axioms [25]. The main objective is to measure different logical and structural communications flows between software components to define the software complexity. To test the new metric formulation, a comparison was made considering other metrics, as cyclomatic complexity, against the axioms. The final results, presented that the entropy measure is sufficient robust as a software metric.

Analyzing the related works, firstly, it is possible to identify some contributions concerning this work. This work proposes an approach to calculate an entropy-based quality value used to rank a source code according to the cyclomatic complexity class. The cyclomatic complexity calculation is inspired by [8], [10], [16] and [15] due to the interesting results using this one. Besides that, cyclomatic complexity in terms of measurements goals [6], presents to be a valid metric to counter balance the Pareto's principle [17]. While entropy utilization is inspired by [23], [2], [21], [22], [24], [19], and [20] due to the indicative suitability in different contexts considering disorder and robustness. Secondly, every source code being produced is compared with the overall already produced source codes from the same software product. This comparison aims to inform the developer about how good a source code is in terms of complexity and quality. This information is useful to guide the developer to improve the source code based on the knowledge of the overall system. To support the calculations, the use of quality and analysis development tools are applied to extract valid software metrics, as performed in [1]. Finally, a Web-based IDE is integrated with the proposed metric to calculate and show to the developers the current quality of the source code being produced with hints about the violations found.

3 Complexity and entropy-based metric

The reasoning behind the proposed metric is to execute a source code ranking to help developers to understand how good their source code are compared with the overall source codes already produced, and besides that, suggests improvements to better rank their source codes. This metric is calculated based on the source code complexity and quality violations found during the development process using source code static analysis. Static analysis can be seen as machine-assisted code review that helps in find developers mistakes with a low cost when compared with human peer-review [4].

The proposed metric has two main components, the cyclomatic complexity class and the entropy-based quality metric. Both components will be better explained in the next sub-sections, but for sake of simplicity, the cyclomatic complexity class aims to calculate the cyclomatic complexity value and use this value to classify one source code in one complexity class (e.g. LOW, MEDIUM, HIGH and VERY HIGH), while the entropy-based quality value, measures the code quality applying the Shannon Entropy [28] calculation based in code violations problems extracted by analysis tools. In this metric, both components are used to provide feedback to the developer about the source code quality using a ranking system. To execute this task, we create a database that stores the calculated source codes statistics, and then a IDE can query this database to rank the code that is been editing. The metric can be presented in an online fashion for the developers to turn them informed about the source code quality during the development process. Besides the source code ranking metric, the extracted quality violations used in the calculation can also be presented to developers, hence supporting them about how to improve that code. Utilizing the proposed method the developers will have the necessary support to fix the violations problems reported and improve their source code ranking position.

To better explain the metric calculation process, imagine a developer editing a source code in the IDE, then we 1) first calculate the cyclomatic complexity class of that source code, 2) then we calculate its entropy-based quality metric, 3) and finally, we rank this source code using the entropy metric and cyclomatic complexity class comparing with the already analyzed source codes from the same software product stored in the database. We conjecture that this ranking process is fair due to the comparison of each source code, in terms of quality, with other source codes that belong to the same complexity class. Briand et al. [6] suggests that the metric must be consistent, and due to the fact that complex source codes generally executes calculations while simple source codes just execute services call or configurations, is presumed to be fairer to compare source codes from the same complexity classes. To better support this rule, we reinforce the statement based on the Pareto's principle used in Melo et al. [17] to explain that 80% of complexity is explained by 20% of source codes. In the next subsections, the components of the proposed metric will be presented in details.

3.1 Cyclomatic complexity class measure

The cyclomatic complexity class is the first component of the statistical raking calculation, various studies had presented interesting results using it ([8], [10], [16] and [15]). Basically, the cyclomatic complexity indicates how many different deviations an algorithm can have during its execution. The most well-known cyclomatic complexity calculation was proposed by McCabe [15], and this one uses graph theory to represent a program flow. The cyclomatic complexity is based on the program control flow (PCF) modeled as a direct graph. In PCF, each vertex represents a linear program block or a deviation structure of the program flow (e.g. if, $elseif$, for, $while$), and the edges are used as links between these blocks. The cyclomatic complexity is defined in the domain of PCF as a function of $v(G)$ using the equation (1) given by McCabe [15].

$$v(G) = e - n + 2p \tag{1}$$

Where e is the number of edges, n is the number of vertices and p is the number of different connected components. The role of p is to calculate the relationship between different connected components. As in our study we are evaluating unitary algorithms, that have only one input vertex, one output vertex and all intermediate vertices allow input vertex reaches the output vertex, the value of p is defined as 1. Therefore, the simplified version of (1) applied for unitary algorithms is $v(G) = e - n + 2$.

The cyclomatic complexity was chosen in this work due to its representation of the different complexity degree that can be considered when comparing source code quality. It supports the assumption that a simple source code requires different levels of rigor about quality than a complex source code, because algorithms with high cyclomatic complexity are harder to be maintained. As we are considering cyclomatic complexity classes (labels) we need to define threshold values to apply the classification. Based on empirical studies, McCabe [15] suggested that algorithms with cyclomatic complexity lower than 10 are simpler. However, his investigation was carried in the Fortran structured language and this threshold is not suitable for different programming language paradigms, like for example, Java that is an Object-Oriented (OO) language. In the case of Java, the complexity class should be adjusted because the language offers many resources to improve code segmentation and re-usability. An alternative, in this case, is the Java source code analyzer tool, PMD [26], that suggests a more suitable set of cyclomatic complexity categories. For example, the proposed values ranges are LOW $[1 - 4]$, MEDIUM $[5 - 7]$, HIGH $[8 - 10]$, and VERY HIGH > 10. As previously stated in the related work, the cyclomatic complexity classification aims to counter-balance the Pareto's principle, stating that 80% of the system complexity is explained by 20% of the source codes [17]. Therefore, the tools, thresholds and number of classes must to be adapted to better help developers to understand their source code by distributing better the complexities.

3.2 Entropy-based quality measure

The entropy-based quality measure is the second component of the proposed metric, various studies had presented interesting results using entropy to support software quality (e.g. [23], [2], [21], [22], [24], [19], and [20]). The base concept of the entropy calculation proposed by Shannon measures how much information is produced by a system or how much uncertainty exists in a system related to the overall chance of the events occur [28]. For example, suppose a binary system given by a set of n different events, where each event can occur with probability $P(x_1), P(x_2), P(x_3), \dots, P(x_n)$, the entropy calculation is given by (2).

$$H = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{2}$$

To exemplify the entropy calculation, considers the information carried by a tossed coin system, where each face has $\frac{1}{2}$ of probability to be selected, its entropy calculated by (2) is $H = -1 \times (2 \times \frac{1}{2} \log_2 \frac{1}{2}) = 1$. This value means that it is sufficient to make only one question for the system to know the result. In contrast, the information carried by a rolled-die system, where each side has $\frac{1}{6}$

of probability to be selected, has an entropy of $H = -1 \times (6 \times \frac{1}{6} \log_2 \frac{1}{6}) \approx 2.58$. Therefore, it is necessary to make 2.58 questions to the system to known the result (in average). Based on the entropy is possible to see that for small entropy values there is less uncertainty about the result, however, for high entropy values, there is more uncertainty about the results.

In this work, as presented in [2], the Shannon Entropy is used to calculate the uncertainty quality about one source code. We can state this quality uncertainty as code disorder. In our study, the calculation depends of the following information from a source code: types of violations, number of violations for each type of violation, and the correction priority for each type of violation. About the information retrieval, these ones can be extracted using different analysis tools, and as suggested by [1], is necessary to analyze the best tools for each specific programming language or paradigm.

To better understand the entropy calculation, first is necessary to detail how the code violation statistics are organized in order to provide the necessary input to the entropy calculation. To exemplify the calculation, suppose a set of quality tools capable to find source code problems, like for example: *UnusedLocalVariable*, *VariableNamingConventions*, *IfElseStmtsMustUseBraces*, and others. These tools can extract the number of occurrences found for each type with their respective priorities, in our example we are assuming that our analysis tools classifies the priorities with values like HIGH = 5, MEDIUM HIGH = 4, MEDIUM = 3, MEDIUM LOW = 2, and LOW = 1. In our calculation proposal the priority factor is used to increase the importance of critical violations problem. Table 1 illustrates an example of violations extracted from a supposed source code. In the proposed metric, there is no fixed rule to define the violation priorities, therefore it can be adapted accordingly for each scenario.

To better explain why entropy depends of violation types, number of occurrences and priorities, this work proposes the following assumptions: 1) The number of violation types found represents the necessary knowledge to fix a given source code, for example, Table 1 depicts 6 violation types, it indicates that it is necessary the knowledge about six different problems to fix them. 2) The total number of violation occurrences represent the necessary effort to fix a given source, Table 1 depicts 42 violations, thus it is necessary to fix those 42 problems to improve the total source code quality. And finally, 3) the priority represents the impact of each violation type, table 1 exemplify that *VariableNamingConventions* should be fixed first, because as more violations with high priority appear, more urgency to solve these are required.

The proposed entropy-based calculation intends to unify all the above-mentioned assumptions in a single quality value. This value aims to measure the uncertainty about the source code violations combining the number of violation types, their priority, and the number of occurrences. To apply these values to the entropy formula, first we need to consolidate the priority and the number of occurrences in a single value. This value is represented as the priority-effort value. Therefore, the proposed entropy calculation is made as following: define X as the set of violation problems extracted from a given

Table 1 Violations extracted by analysis tools.

Violation type	Occurrences	Priority
UnusedLocalVariable	10	3
VariableNamingConventions	5	5
IfElseStmtsMustUseBraces	1	3
ConfusingTernay	1	3
DataFlowAnomalyAnalysis	20	1
StyleNonConformity	5	1

source, where i indicate the i^{th} violation type, x_i is the number of violations found in type i and δ_i is the priority related to the type i. The priority-effort relationship is calculated by (3), where x'_i is the priority-effort value for each violation type.

$$\forall x_i \in X,\ x'_i = x_i \times \delta_i \tag{3}$$

The objective of the priority-effort value is to unify these variables in a single component necessary to represent the disorder degree of a source code. The next step, is to measure the probability $P(x'_i)$ for each priority-effort value before to apply the entropy formula. This probability will define the uncertainty about that problem in the source code. But to apply this value in the entropy formula, first a normalization step is applied to reduce the influence of violations with many occurrences in the code. This normalization is necessary because the violations with high priority-effort values will gain more relevance than the others and the entropy will be wrongly stuck in very recurrent problems. For example, suppose a given source code where was extracted one hundred of *DataFlowAnomalyAnalysis* violations with priority one and ten *StyleNonConformity* violations with priority one. In this scenario, the entropy will be small because there is a high chance to found the *DataFlowAnomalyAnalysis* with a probability of 90%. To fix this saturation, the normalized priority-effort probability is calculated by (4). This equation uses the natural logarithm to remove the saturation and maintain an adjusted priority-effort scale for each violation type found in the source code.

$$P(x'_i) = \frac{\ln x'_i}{\sum_{x'_i \in X} \ln x'_i} \tag{4}$$

Finally, the entropy-based quality metric $E(X)$ is calculated by (5). It measures the uncertainty of finding quality problems in one source code. So, the higher the entropy value is, the lower the quality of the source code is because there is more chance to find different code violations. In contrast, the lower the entropy is, the higher the quality of the source code is because there is less possibility of finding code violations.

$$E(X) = -\sum_{x'_i \in X} P(x'_i) \log_2 P(x'_i) \tag{5}$$

3.3 Algorithms

To integrate the proposed complexity and entropy-based metric with an IDE in an online fashion, a quality source code analyzer is proposed to execute the steps of the proposed metric, that is rank a currently editing source code and compare it with the overall source codes with same complexity from the same software product. To perform the raking, first, a database is generated using the Algorithm 1.

Algorithm 1 Create the database for all Java source codes

```
Require: $S$
  $Q \leftarrow CreateDatabase()$
  for $k \leftarrow 0$ to $length(n)$ do
    $c_k \leftarrow GetCyclomaticComplexity(s_k)$
    $X_k \leftarrow GetQualityProblems(s_k)$
    $e_k \leftarrow GetEntropy(X_k)$
    $r_k \leftarrow c_k e_k$
    $Persist(Q, s_k, c_k, X_k, e_k, r_k)$
  end for
  return $Q$
```

Algorithm 1 creates a database Q for a given set of source codes, defined as S. In this algorithm, for each source code $s_k \in S$, first is calculated the cyclomatic complexity class, c_k, and it is stored using integer representation, for example: Low = 1, Medium = 2, High = 3 and Very High = 4. After that, the violation problems, X_k, are extracted using quality analysis tools, and then the entropy quality measure, e_k, is calculated. The next step is to index a single overall score for the source code r_k combining the cyclomatic complexity class c_k and the entropy quality measure e_k. Finally, all the information is persisted in the database Q. This statistical database could be updated every night by executing the Algorithm 1 in order to have a fresh database to execute the source code comparisons for the next day.

With the created the database, the Algorithm 2 is used to calculate to the users their source code quality during development. Basically, it uses the database instance Q to define the ranking index ($r_{editing}$) for an editing source code ($s_{editing}$). The algorithm starts calculating the cyclomatic complexity class ($c_{editing}$), the violation problems ($X_{editing}$), the entropy ($e_{editing}$) and the source code index ($r_{editing}$). After that, the $r_{editing}$ is used to filter over the database Q and find all source codes classified in the same complexity class ($c_{editing}$), after that we rank the editing source code based on the results. In this algorithm, each source code is always compared in their same complexity class.

Algorithm 2 Rank a Java source code using the statistical database

```
Require: $s_{editing}, Q$
  $c_{editing} \leftarrow GetCyclomaticComplexity(s_{editing})$
  $X_{editing} \leftarrow GetQualityProblems(s_{editing})$
  $e_{editing} \leftarrow GetEntropy(X_{editing})$
  $r_{editing} \leftarrow c_{editing} e_{editing}$
  $idx \leftarrow RankBasedOnComplexityClassAndQuality(Q, r_{editing})$
  return $idx$
```

4 Implementation

To presents a practical application of the proposed metric, this section describes a metric extractor tool implementation for Java language. This tool was implemented to support a private company environment where users develop source code in a WEB IDE interface. The implementation considers a source code repository to execute the calculations, and to analyze the source code violation problems, the PMD [26] and CheckStyle [27] analysis tools were applied. Both tools offers support to Java language.

The PMD was chosen due to the efficacy in find code violations. In literature, there are studies using it to reduce smart-phone battery consumption from apps based in violations correction (Nguyen [33]). Besides that, PMD embraces 7 out of 15 of the most suitable quality metrics applied for object-oriented software (Santos et al. [13]). In the documentation site there are registered support to around 300 violation types classified in knowledge areas of programming skills like: best practices, code style, design, documentation, error-prone, multithreading, performance, and security.

The CheckStyle was chosen because it supports source code formatting based in the Java convention guidelines (e.g. space between if statement and parenthesis, and blank line at end of file), like the one define by Sun [32]. In the documentation site, there is registered around 170 code formatting problems that CheckStyle can handle. The CheckStyle was choose because it better support Java language formatting than PMD.

In the conducted case study, the problems reported by PMD are grouped by their violation types while CheckStyle problems are grouped in a single type, called *CheckStyle*, and all violations found in each type is summed. This strategy was applied, because in the company context, the PMD finds more critical quality problems while CheckStyle only carries about code formatting. Therefore, PMD is used as the main problem analyzer to guarantee a better usage of the Java programming language and its patterns, while CheckStyle is used to assist the PMD with the Java source code formatting analysis.

Related to the metric thresholds, in this implementation, the cyclomatic complexity classes applied are the same from PMD, where the ranges are: LOW $[1-4]$, MEDIUM $[5-7]$, HIGH $[8-10]$, and VERY HIGH > 10, and related to the priority factors the values are the same of PMD: HIGH $= 5$, MEDIUM HIGH $= 4$, MEDIUM $= 3$, MEDIUM LOW $= 2$, and LOW $= 1$. Tough CheckStyle presents different prioritization among the formatting styles, in this work, its priority factor is defined as LOW = 1, because the formatting problems are considered less critical than the violations extracted by PMD.

To define the cyclomatic complexity class for a given Java source code, the cyclomatic complexity value is measured for each method and the method with the highest value defines the source code cyclomatic complexity class. The Java method with the highest value is selected because it represents an absolute upper bound. Using some other alternative as summation from all methods or the average could result in some side effects. For example, using the summation, a class with several methods with low cyclomatic complexity values could result in a summation value greater than a class with only one method with one higher cyclomatic complexity value. While using an average value, one source code with a single method with high cyclomatic complexity could be smoothed by several other methods with low cyclomatic complexity. Therefore, the use of the high cyclomatic complexity method seems to be more fairly in terms of comparison. Though this implementation uses Java, for different programming languages different analysis tools can be applied with different thresholds values.

In this implementation, the metric calculation and the database generation have been implemented in Java language [29] using the Spring Framework [30], these technologies were chosen based on the company environment. To persist the statistical data about the analyzed source codes calculated by Algorithm 1, the MongoDB database [31] has been selected. A micro-service has been developed to implement the software back-end using the current company architecture and it has been deployed in the company private cloud infrastructure within a Docker container. The service is available through a REST endpoint interface that is called by the Web IDE. The REST interface receives the source code and returns the statistical score information with the source code ranking. Besides that, the proposed tool also presents the violation problems and hints to solving them. Figure 1 presents the defined architecture.

The next section will discuss about application of the proposed implementation in a real scenario as a case study.

5 Assessments and Results

To validate the proposed metric and implementation tool, theoretical analysis and a case study were assessed. The case study was applied in a private company of the energy sector that uses a project management system developed in Java and is maintained by the internal development team. In the development team, around 5 employees have Computer related under graduation with high Java stack development skills which are responsible to maintain the core of the project management system, while around 45 employees have Engineering under graduation and are responsible to just maintain and create new business rules in Java language as well. The engineers use a private Web IDE platform to code, compile and store the Java source codes in an online fashion. Cur-

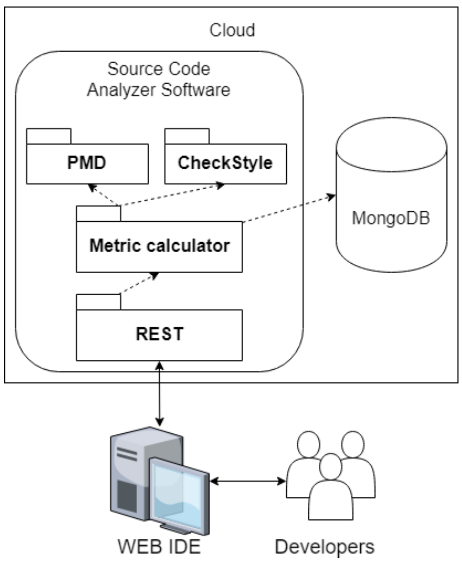


Fig. 1 Software architecture for the proposed metric.

rently, the problem lays on how to reduce the source code complexity and help the developers with limited programming skills to improve their source codes quality. To execute the tests, the existing Web IDE was adapted to show the ranking score and the violation problems extracted from the current source code being developed using algorithms 1 and 2. Therefore, the aim of this assessment is 1) validate the consistency and applicability of the proposed metric and 2) check if the source code quality had been improved after applying the metric in the company.

5.1 Pareto principle analysis

Concerning the first validation objective, which is to check the consistency and applicability of the proposed metric, our first analysis is related to the Pareto principle [17]. This analysis was made using the code base of the company, where the proportion of source codes between different complexity classes was compared. The results presented in Table 2 shows that 87.3% of the source codes are Low complex, while 12.7% of the source codes are Medium to Very High complex. Though the values does not fit in 80% and 20% perfectly as stated in the Pareto principle [17], the principle is still assumed valid to explain the different magnitudes between the complexity classes.

Table 2 Pareto principle considering number of source codes by cyclomatic complexity class.

Complexity class	Num. source codes	Proportion (%)
Low	162448	87.3%
Medium	7812	4.2%
High	4124	2.2%
Very High	11604	6.6 %

5.2 User interaction with the metric analysis

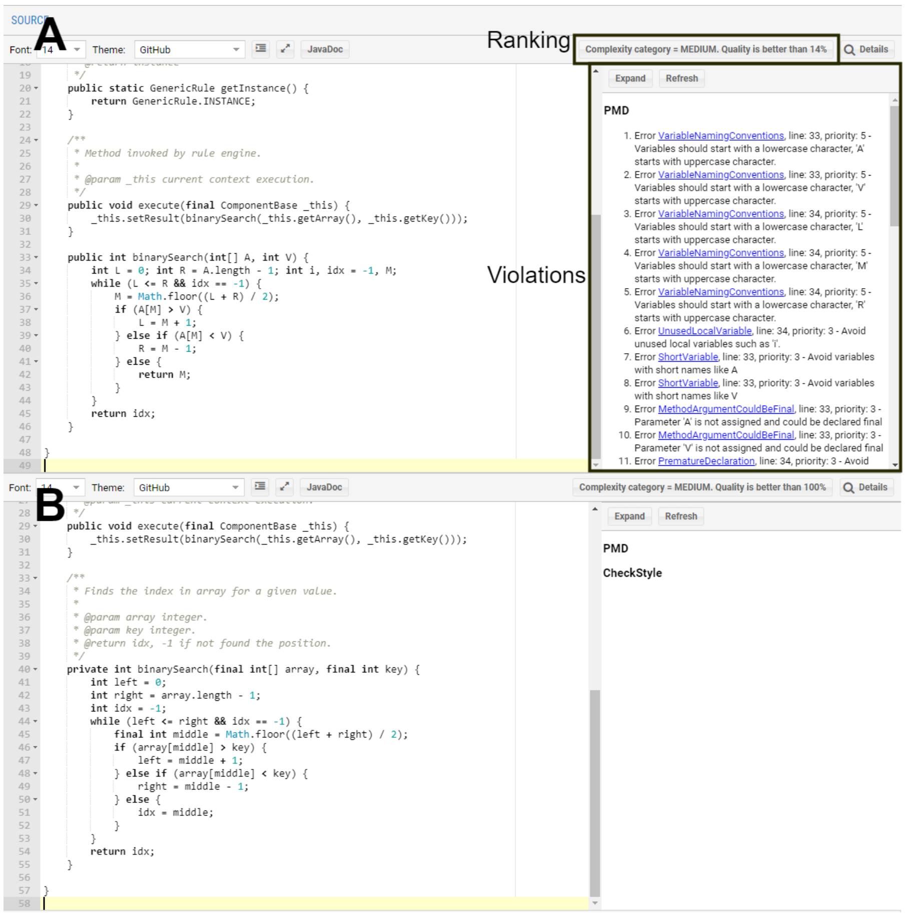
To validate the user interaction with the metric, Figure 2 demonstrates the source code analyzer tool developed. Basically, the Figure 2.A presents a given source code before the correction of the violation problems found (enumerated in the *Violations* rectangle) and the Figure 2.B presents the same source code with the violations fixed. The proposed source code ranking is available in the rectangle labeled as *Ranking* in the editor, which presents to the developer the cyclomatic complexity class and the source code ranking in that complexity class. The ranking value is basically the entropy value normalized in percentage between the code with highest and smallest entropy. In Figure 2.A, the ranking means that the current source code is better than 14% of the overall source code that belongs to the complexity class *MEDIUM*. The enumerated violations found in the *Violations* rectangle are presented for the developer with a link to the problem, the line where it was found, its priority and a brief description about what it means (e.g. *VariableNamingConvention*, line 33, priority 5 - Variables should start...). Besides that, the violations are ordered by descendent priority to facilitate the identification of the most important ones.

Figure 2 presents a binary search algorithm written with some coding mistakes in order to validate the quality metric implementation. Some examples of the violations found in the source code of Figure 2.A are:

- *VariableNamingConvetions* and *ShortVariable* indicate that a variable is not well named;
- *UnusedLocalVariable* indicates that a variable is declared but not used;
- *MethodArgumentCouldBeFinal* indicates that method parameters should be declared as final to prevent variable reassignments;
- *PrematureDeclaration* indicates that a variable has chance to not be used due to the while condition.

Figure 2.B presents the same source code after the violations are fixed. Now, the statistical ranking states that the current code is better than 100% of those source code that belongs to the MEDIUM cyclomatic complexity.

After performing this experiment, it is possible to point out that the source code analyzer tool is suitable to find and show code violations as well as to suggest solution hints. Besides that, the proposed statistical ranking can be useful to demonstrate for the developer the source code quality progress along the time, given a guide to fix the violations found during development and



Complexity category = MEDIUM. Quality is better than 14%

```
     */
    public static GenericRule getInstance() {
        return GenericRule.INSTANCE;
    }

    /**
     * Method invoked by rule engine.
     *
     * @param _this current context execution.
     */
    public void execute(final ComponentBase _this) {
        _this.setResult(binarySearch(_this.getArray(), _this.getKey()));
    }

    public int binarySearch(int[] A, int V) {
        int L = 0; int R = A.length - 1; int i, idx = -1, M;
        while (L <= R && idx == -1) {
            M = Math.floor((L + R) / 2);
            if (A[M] > V) {
                L = M + 1;
            } else if (A[M] < V) {
                R = M - 1;
            } else {
                return M;
            }
        }
        return idx;
    }

}
```

PMD

1. Error VariableNamingConventions, line: 33, priority: 5 - Variables should start with a lowercase character, 'A' starts with uppercase character.
2. Error VariableNamingConventions, line: 33, priority: 5 - Variables should start with a lowercase character, 'V' starts with uppercase character.
3. Error VariableNamingConventions, line: 34, priority: 5 - Variables should start with a lowercase character, 'L' starts with uppercase character.
4. Error VariableNamingConventions, line: 34, priority: 5 - Variables should start with a lowercase character, 'M' starts with uppercase character.
5. Error VariableNamingConventions, line: 34, priority: 5 - Variables should start with a lowercase character, 'R' starts with uppercase character.
6. Error UnusedLocalVariable, line: 34, priority: 3 - Avoid unused local variables such as 'i'.
7. Error ShortVariable, line: 33, priority: 3 - Avoid variables with short names like A
8. Error ShortVariable, line: 33, priority: 3 - Avoid variables with short names like V
9. Error MethodArgumentCouldBeFinal, line: 33, priority: 3 - Parameter 'A' is not assigned and could be declared final
10. Error MethodArgumentCouldBeFinal, line: 33, priority: 3 - Parameter 'V' is not assigned and could be declared final
11. Error PrematureDeclaration, line: 34, priority: 3 - Avoid

Complexity category = MEDIUM. Quality is better than 100%

```
     */
    public void execute(final ComponentBase _this) {
        _this.setResult(binarySearch(_this.getArray(), _this.getKey()));
    }

    /**
     * Finds the index in array for a given value.
     *
     * @param array integer.
     * @param key integer.
     * @return idx, -1 if not found the position.
     */
    private int binarySearch(final int[] array, final int key) {
        int left = 0;
        int right = array.length - 1;
        int idx = -1;
        while (left <= right && idx == -1) {
            final int middle = Math.floor((left + right) / 2);
            if (array[middle] > key) {
                left = middle + 1;
            } else if (array[middle] < key) {
                right = middle - 1;
            } else {
                idx = middle;
            }
        }
        return idx;
    }

}
```

PMD

CheckStyle

Fig. 2 Example of same source code improved by the quality metrics suggested.

refactoring tasks. Therefore, the ranking is considered suitable to help the developers to improve their source code qualities and achieve best-ranking positions due to the online metric feedback.

5.3 Entropy analysis

To analyze the contribution of the entropy measure in the source code violation analysis, Figure 3 presents how entropy contributes to better distribute the total priority-effort necessary to fix the code violations found. In Figure 3, each point represents a source code statistic from the database created by Algorithm 1. The violations type axis presents the number of violations types found, the total priority-effort axis presents the result calculated by (3) and, finally, Entropy axis presents the result calculated by (5). The total priority-effort and entropy values are compared with the violations type of the same

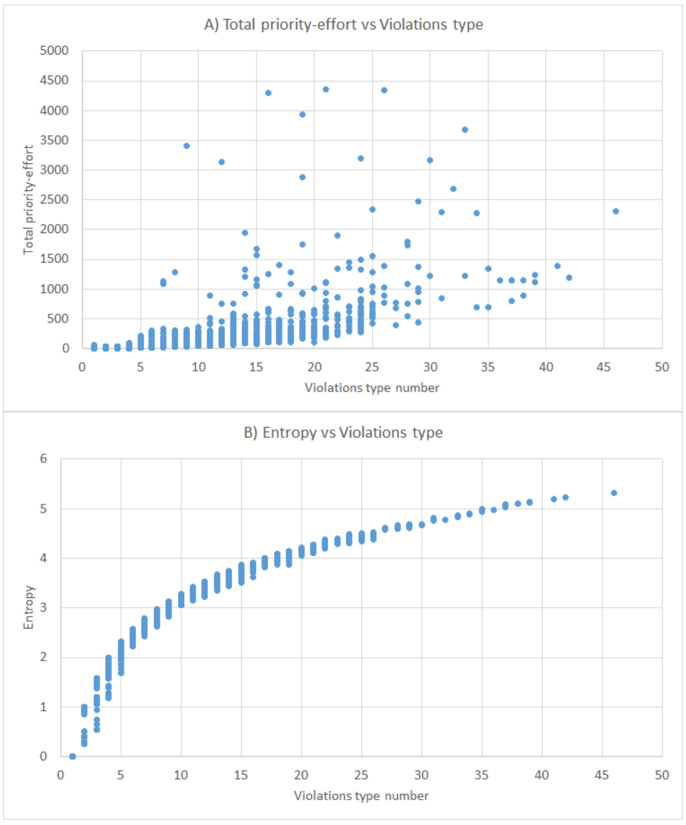


Fig. 3 Entropy and priority-effort relationship with knowledge.

set of source codes to analyze the correlation with the knowledge (number of violation types). The graph A presents the correlation between total priority-effort and violations type, while graph B presents the correlation between entropy and violations type. The analysis of the graphs indicates that entropy can better quantify the source code quality because it can better distribute the priority-effort to fix the violations found by means of uncertainty in finding them. This is due to the normalization step defined in (4) that reduces the saturation caused by violations type with a massive number of violations, and the entropy calculation value.

5.4 Metric theoretical validation

In terms of theoretical validation is important to execute the fundamental analysis of the metric. Briand et al [6] presented a guideline that must be followed to validate the importance and reliability of a given metric. Based on the criteria pointed out, we can execute the following analysis:

- Measurement goal: the metric establishes a ranking system between codes in the same software. Its support our metric goal to expect that when all source codes achieve the level of 100% the software will be totally in accordance with the quality tools.
- Experimental hypothesis: based on the analysis of Figure 3, it is possible to infer that the metric is capable to quantify the source code quality using a formulation that combines entropy and violation uncertainty. The model, formulated in a logarithmic function, states that entropy increases at the same time which that the uncertainty about find violations in source code increases.
- Environmental context: due to the Pareto principle, the environmental context is better distributed to compare source codes with different levels of rigor.
- Theoretical validation: the different classes of complexity support our analysis that different levels of rigor are given by the source code complexity. While the entropy value, that is calculated using the combination of violation types, number of occurrences, and priority, indicates the level of uncertainty as code quality.
- Empirical validation, the empirical validation presented in Figure 2 that shows a source code being improved while the developer is receiving feedback about the problems.

5.5 Company results analysis

Related to the second validation aim, an analysis of the entropy variation has been performed over the repository of source codes to figure out whether the code violations have been reduced since the quality analysis tool started to be used by the development team. Figure 4 presents the entropy for every source code since the beginning of 2018. As the tool was deployed in May 1st, 2018, every statistics before this date do not have any support from the proposed metric. Therefore, it is supposed that the source code quality before May 1st, 2018, is totally associated with technical skills and engagement of developers themselves. However, code quality improvements after this date possibly are partially related to the ranking and fixing suggestions presented by the tool.

In Figure 4, it can be observed that after the code quality analyzer deployment in May 1st, 2018, there is noted a reduction in the entropy measured in several source codes. However, this reduction has been partially obtained by fixing the code violations of the Web IDE default Java class template that is generated by the software automatically. This class contains a Java singleton signature (design pattern) defined by the company and is used by engineers as the main structure to insert the codes related to the business rules. The template correction was necessary to achieve the measurement goal of software total accordance by automatic source code generation. After fixing the template, it can be observed that some new source codes have sensibly improved their overall quality by keeping the entropy between 0 and 2. However,

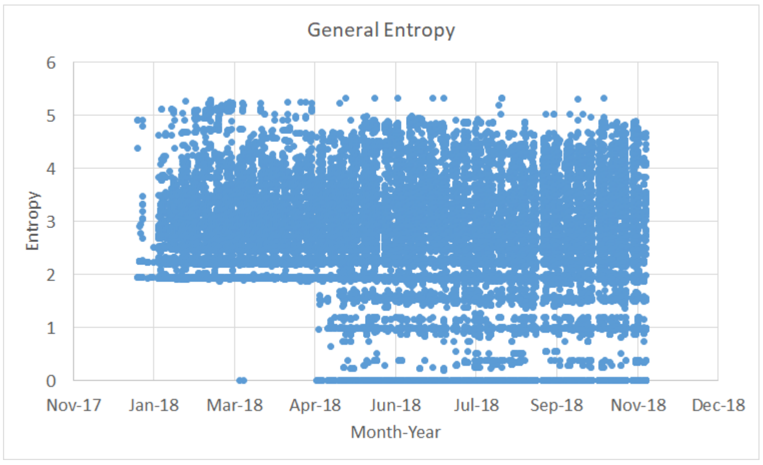


Fig. 4 General entropy of the system. Data from the first semester of 2018.

old codes that were created before May 1st, 2018, do not have the benefit brought by the template corrections, even if updated after this date. So, to improve their quality it is necessary to make the correction in the default template manually. This explains why many codes after this date are still having entropy ≥ 2.

In order to evaluate the company results of the proposed metric for each cyclomatic complexity class, Figure 5 presents the source code accumulative entropy and number of source code modified. The source code modification number represents the number of commits made in the system repository. We use accumulative values because as developers needs to constantly check previous version of source code during development, the complexity keeps alive in the sense of maintenance process. About the Figure 5 organization, the horizontal axis presents the time segmented by month, the left axis presents the entropy accumulated and the right axis presents the number of source codes accumulated. Besides that, the analyzer deployment date is highlighted by the vertical line. The results time span consider the 2018 year, which is the year of the tool release.

Analyzing the Figure 5 results, it is possible to see that codes with LOW, MEDIUM, and HIGH cyclomatic complexity presents improvements in source code quality, due to the entropy reduction. While VERY HIGH cyclomatic complexity codes had not presented an entropy reduction, which means that the quality was not improved. A hypothesis about the quality improvements related with source codes with LOW, MEDIUM and HIGH complexity is partially because they are less complex than the others by nature and are easy to fix the quality problems. Besides that, the number of analyzed low complexity codes is greater than the high complexity ones. This means that it is necessary to take more time to observe if the high complexity source codes will improve. Is important to point out that users were not trained about the

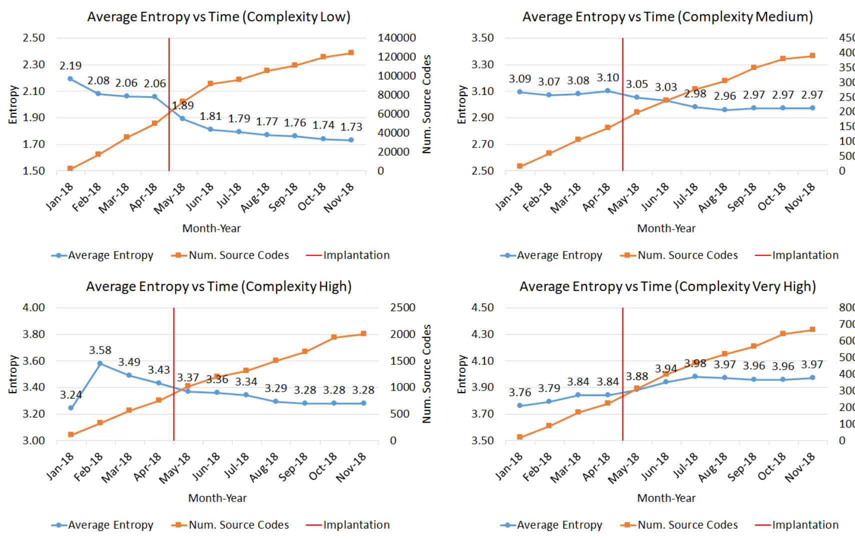


Fig. 5 Accumulative entropy trending of the system. The vertical line presents the proposed metric and tool deployment date, the left axis represents the entropy and the right axis represents the number of source codes.

tool usage and neither stimulated to follow the tool suggestions, therefore is expected that with some incentive by part of the managers, the source code quality improvements could be more effective.

As observed in the assessment presented in the Figure 5, was no perceived a significant difference in the quality improvement by parts of the overall developers themselves. To extend this analysis, we have selected four programmers of the top source code modifiers to evaluate their performance isolated. The results about their accumulative performance after the software deployment date is presented in the Figure 6. In these results, the horizontal axis represents the time interval segmented by months, the left axis presents the accumulative entropy and the right axis presents the accumulative number of source codes. Besides that, the tool deployment date is highlighted by the vertical line. The results presents that, in general, these programmers follows a entropy reduction trend after May 1st, 2018. Specifically, Programmer 1 and Programmer 2 which had presented a continuous improvement, while Programmer 3 and Programmer 4 have some peeks of high complexity but in most months the complexity was small than in the months before the deployment date. As these developers are the top source code modifiers, they have high amount of source code modified (right axis), it can indicate new source codes or recurrent maintenance's. Besides that, we can observe that Programmer 2 have presented entropy reduction even with the increasing number of source codes along the time.

Based on the results, we seen that the proposed source code analyzer presented an entropy reduction after the deployment date for some complexity

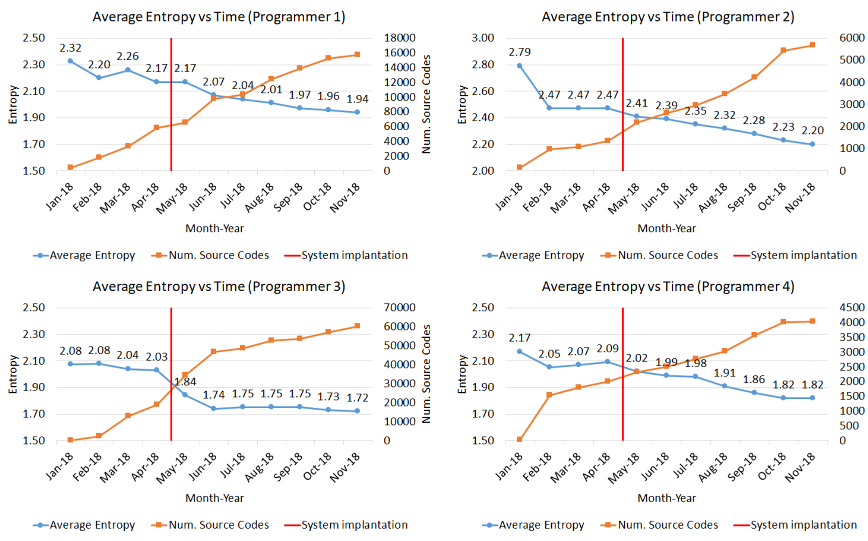


Fig. 6 The top four source code modifiers. This results are related with the accumulative entropy trending after the source code analyzer deployment date (1st May. of 2018). Left axis presents the entropy and the right axis presents the number of source codes committed.

classes and developers, it is partially caused by default template correction and the usage of the proposed metric. Related to the overall scenario, we suppose that some kind of reward given by part of company to developers could be a good motivator to improve the software quality. As presented in [8], a reward system where the premiums could be a rest day, some kind of extra money or gifts, could motivate the developers to achieve best source code qualities. Therefore, the proposed quality metric is a good mediator to support the decision process to define the best reward developers at the same time which the tool is the focal point to learn how to fix the code problems. Besides that, as proposed in [10], is possible to use the tool to find developers that need to be trained in the quality policies and find software regions that should receive investments to improve the quality.

6 Conclusions and future works

This work proposed a complexity and entropy-based metric to help software developers to produce code with better quality reducing violation problems. The proposed metric combines entropy calculation to measure the uncertainty of finding quality problems in a source code with a cyclomatic complexity classification used to compare codes with similar complexity.

Besides that, an online source code analyzer has been developed. The source code analyzer aimed to assist users from a private company to improve their source code during the development process. The tool compares the current editing source code against the overall codes with same complexity class produced in the same software product. The comparison ranks the source code in a range from $[0-100]\%$ using the entropy quality metric. Therefore, the developer can see how far his/her code is from the better ones. Besides this comparison, it is presented to the developers hints about how to improve their source codes in an online fashion.

To validate the proposed metric, some statistics and analyzes have been made. The analyzes indicated that the proposed metric with the source code analyzer is suitable to help users to improve their source code quality. The results indicating that after the analysis tool deployment date the cyclomatic complexity from some top source code modifiers had being reduced. However there is a lack for experimental tests, where different software products, teams and technologies can be applied to validate the metric.

As future work, it is suggested to improve the theoretical analysis about the entropy measure calculation and also test the tool in different programming languages, like PHP and Python.

References

1. Ludwig, J., Xu, S., Webber, F.: Compiling static software metrics for reliability and maintainability from GitHub repositories. 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC). (2017).
2. TURNU, I., CONCAS, G., MARCHESI, M., TONELLI, R.: ENTROPY OF SOME CK METRICS TO ASSESS OBJECT-ORIENTED SOFTWARE QUALITY. International Journal of Software Engineering and Knowledge Engineering. 23, 173-188 (2013).
3. Singh, B., Gautam, S.: The Impact of Software Development Process on Software Quality: A Review. 2016 8th International Conference on Computational Intelligence and Communication Networks (CICN). (2016).
4. Alfayez, R., Behnamghader, P., Srisopha, K., Boehm, B.: An exploratory study on the influence of developers in technical debt. Proceedings of the 2018 International Conference on Technical Debt - TechDebt '18. (2018).
5. Timóteo, A., Alvaro, A., Santana De Almeida, E., Meira, S.: Software Metrics: A Survey. Research Gate. (2014).
6. Briand, L., Morasca, S., Basili, V.: An operational process for goal-driven definition of measures. IEEE Transactions on Software Engineering. 28, 1106-1125 (2002).
7. Beranič, T., Podgorelec, V., Heričko, M.: Towards a Reliable Identification of Deficient Code with a Combination of Software Metrics. Applied Sciences. 8, 1902 (2018).
8. de Melo, A., Hinz, M., Scheibel, G., Diacui Medeiros Berkenbrock, C., Gasparini, I., Baldo, F.: Version Control System Gamification: A Proposal to Encourage the Engagement of Developers to Collaborate in Software Projects. Social Computing and Social Media. 550-558 (2014).
9. Write your best with Grammarly., https://grammarly.com/.
10. Barbosa, M., de Lima Neto, F., Marwala, T.: Tolerance to complexity: Measuring capacity of development teams to handle source code complexity. 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC). (2016).
11. Eeles, P., Bahsoon, R., Mistrik, I., Roshandel, R., Stal, M.: Relating System Quality and Software Architecture. Relating System Quality and Software Architecture. 1-20 (2014).
12. Binanto, I., Warnars, H., Gaol, F., Abdurachman, E., Soewito, B.: Measuring the quality of various version an object-oriented software utilizing CK metrics. 2018 International Conference on Information and Communications Technology (ICOIACT). (2018).
13. Santos, M., Afonso, P., Bermejo, P., Costa, H.: Metrics and statistical techniques used to evaluate internal quality of object-oriented software: A systematic mapping. 2016 35th International Conference of the Chilean Computer Science Society (SCCC). (2016).

14. Alsmadi, I., Alazzam, I.: Software attributes that impact popularity. 2017 8th International Conference on Information Technology (ICIT). (2017).
15. McCabe, T.: A Complexity Measure. IEEE Transactions on Software Engineering. SE-2, 308-320 (1976).
16. Campos Junior, H., Martins Filho, L., Araujo, M.: An Approach for Detecting Unnecessary Cyclomatic Complexity on Source Code. IEEE Latin America Transactions. 14, 3777-3783 (2016).
17. Dunford, R., Su, Q., Tamang, E. The Pareto Principle. The Plymouth Student Scientist, 7, 140-148 (2014).
18. Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P.: On the "naturalness" of buggy code. Proceedings of the 38th International Conference on Software Engineering - ICSE '16. (2016).
19. Zhang, X., Ben, K., Zeng, J.: Cross-Entropy: A New Metric for Software Defect Prediction. 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). (2018).
20. Parveen, T., Arora, H.D.: Estimating release time and predicting bugs with Shannon entropy measure and their impact on software quality. Thai Journal of Mathematics, 15, 91-105 (2017).
21. Nyamawe, A., Liu, H., Niu, Z., Wang, W., Niu, N.: Recommending Refactoring Solutions based on Traceability and Code Metrics. IEEE Access. 1-1 (2018).
22. Gupta, A., Suri, B., Kumar, V., Misra, S., Blažauskas, T., Damaševičius, R.: Software Code Smell Prediction Model Using Shannon, Rényi and Tsallis Entropies. Entropy. 20, 372 (2018).
23. Cholewa, M.: Shannon information entropy as complexity metric of source code. 2017 MIXDES - 24th International Conference Mixed Design of Integrated Circuits and Systems. (2017).
24. Akundi, A., Smith, E., Tseng, T.: Information entropy applied to software based control flow graphs. International Journal of System Assurance Engineering and Management. 9, 1080-1091 (2018).
25. Cherniavsky, J., Smith, C.: On Weyuker's axioms for software complexity measures. IEEE Transactions on Software Engineering. 17, 636-638 (1991).
26. PMD, https://pmd.github.io/.
27. checkstyle – Checkstyle 8.12, http://checkstyle.sourceforge.net/.
28. Shannon, C., Weaver, W.: The mathematical theory of communication, by Claude E. Shannon and Warren Weaver. University of Illinois Press, Urbana (1971).
29. Java – Java You, Download Today!, https://java.com/.
30. Spring – spring.io, https://spring.io/.
31. MongoDB – MongoDB for GIANT Ideas, https://www.mongodb.com/.
32. Code Conventions for the Java Programming Language: Contents, https://www.oracle.com/technetwork/java/codeconvtoc-136057.html.
33. Nguyen, M., Huynh, T., Nguyen, T.: Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint. Lecture Notes in Computer Science. 343-356 (2016).