RETICULA: Real-Time Code Quality Assessment

Luigi Frunzio, Bin Lin, Michele Lanza and Gabriele Bavota
Faculty of Informatics — Università della Svizzera italiana (USI), Switzerland

***Abstract*—Code metrics can be used to assess the internal quality of software systems, and in particular their adherence to good design principles. While providing hints about code quality, metrics are difficult to interpret. Indeed, they take a code component as input and assess a quality attribute (*e.g.,* code readability) by providing a number as output. However, it might be unclear for developers whether that value should be considered good or bad for the specific code at hand.**

We present RETICULA **(REal TIme Code qUaLity Assessment), a plugin for the IntelliJ IDE to assist developers in perceiving code quality during software development.** RETICULA **compares the quality metrics for a project (or a single class) under development in the IDE with those of *similar* open source systems (classes) previously analyzed. With the visualized results, developers can gain insights about the quality of their code.**

A video illustrating the features of RETICULA **can be found at: https://reticulaplugin.github.io/.**

I. Introduction

Code quality metrics aim at assessing the internal quality of software systems. For example, in the context of Object-Oriented Programming (OOP), the Chidamber & Kemerer (CK) metrics suite [1] provides a set of metrics to assess the adherence of code to OOP principles (*e.g.,* high class cohesion).

While quality metrics can provide precious hints about code quality, they rarely lead to improvement actions when used in isolation. This is mainly due to the fact that quality metrics are difficult to interpret. Indeed, they take a code component as input and assess a quality attribute by providing a number as output. For example, the Weighted Methods per Class (WMC) metric [1] aims at assessing the complexity of a given class and it is computed as the sum of the McCabe's cyclomatic complexity [2] of its methods. The WMC provides an unbounded positive integer as output: The higher the WMC, the higher the class complexity. However, it is difficult for a developer to interpret the output of this metric:

Is a WMC of 20 for a class worrying?

To answer this question many authors tried to define thresholds for code quality metrics aimed at identifying code components in need of refactoring [3], [4], [5], [6], [7], [8], [9], [10]. However, it is difficult to identify for a given metric a single threshold acting as a silver bullet. For example, an average WMC of 20 can be perfectly fine for classes of a system implementing a Java parser, while being worrying for a system implementing UML drawing tools.

Lanza and Marinescu [3] suggested to compare the quality metrics measured on a project to those of systems written in the same language. This is one of the ideas behind their overview pyramid, a simple visualization in which metrics are normalized over the project's size, thus allowing a meaningful comparison among systems of different size.

We build on top of this idea, and present RETICULA (**RE**al **TI**me **C**ode q**UA**l**I**ty **A**ssessment), an IntelliJ[1] plugin able to (i) compute on-the-fly code quality metrics for the project under development and for the specific class opened in the IDE, and (ii) compare the computed values to those of *similar* projects mined from GitHub. With *similar* we mean projects written in the same programming language, having similar size and age, developed by teams of comparable size, and related to a similar application domain.

RETICULA exploits a knowledge base of ∼300k Java projects mined from GitHub. For each mined project RETICULA stores key information to filter projects similar to the one under development in the IDE (*e.g.,* the project size, its age, its textual description, etc.). Then, it measures a set of quality metrics for all Java classes in the project. RETICULA's knowledge base is in continuous expansion, and grows at the rate of ∼6k new projects per day. The RETICULA plugin exploits this knowledge base and implements effective visualizations to provide developers with a real-time feedback about the quality of the code they are writing as compared to that of similar software projects.

II. RETICULA in a Nutshell

Fig. 1 depicts the RETICULA architecture and the interaction of its components. The dashed arrows represent dependencies (*e.g.,* ①), while the solid ones indicate flows of information pushed from one component to another (*e.g.,* ②).

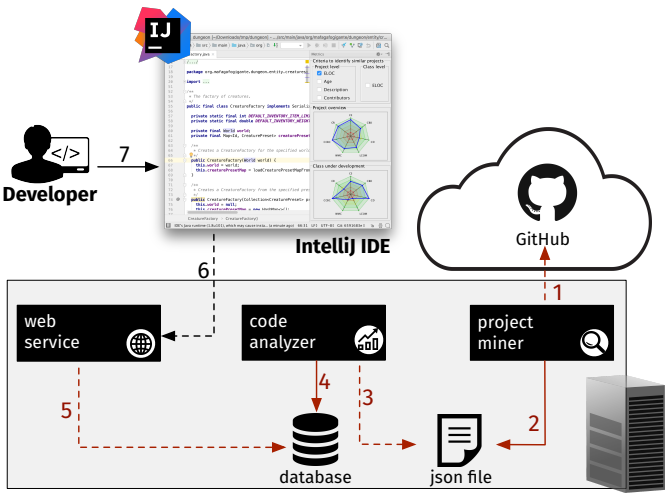


Fig. 1. RETICULA architecture

[1]https://www.jetbrains.com/idea/

The *projects miner* finds open source Java projects in GitHub (① in Fig. 1) and saves the URL of their repository in JSON format ②. The *code analyzer* clones the project's repositories and analyzes them with code quality metrics ③.

The results of this analysis are stored in a database ④, representing the RETICULA knowledge base from which identifying projects similar to the one under development in the IDE.

When the developer activates the RETICULA plugin in the IntelliJ IDE ⑦, a request is sent to the RETICULA web service ⑥, which is in charge of comparing the quality metrics of the project and of the Java class currently opened in the IDE (if any) to those of similar projects/classes already analyzed and stored in the database ⑤. The results of this comparison are presented to the developer in the IDE through visualizations and will be updated on the fly while the developer writes code.

A. RETICULA Server Side

1) ***Projects miner****:* The *projects miner* is developed with Node.js. It exploits the GitHub APIs to mine Java projects. The miner stores the URL of the repository of each identified project into a JSON list. Overall, we collected the URLs of over two million Java projects. Such a list can be updated at regular intervals by re-running the projects miner.

2) ***Code analyzer****:* The *code analyzer* is developed in Scala. It starts by cloning on the server the repository of each project in the JSON list created by the *projects miner*. For each project it extracts the information summarized in Table I and stores it in the database. At a higher level, this information can be classified into two sets.

The first set contains meta information useful to characterize the project and that is used in a second step to select from the RETICULA database the projects similar to the one under development in the IDE. This includes: (i) its textual description, extracted from the `README.md` file used in GitHub repository to describe the repository[2]; (ii) the number of developers involved in the project, as provided by the GitHub APIs; and (iii) the date at which the repository was created, needed to compute the age of the project.

The second set of information includes code quality metrics measured at class-level for the specific repository under analysis. The quality metrics are measured on the latest project's snapshot available at the date of the cloning. The set of metrics considered in RETICULA is listed in Table I. This includes: (i) the ELOC—Effective LOC (*i.e.,* lines of code excluding blank lines and comments)—as size metric; (ii) a subset of the Chidamber-Kemerer (CK) metrics' suite [1], measured by using the CK tool [11]; (iii) two conceptual (textual) metrics aimed at assessing the Conceptual Cohesion of Classes (C3) [12] and the Conceptual Coupling Between Classes (CCBC) [13]; (iv) the code readability metric defined by Buse & Weimer [14], for which we used the implementation made publicly available by the authors[3]; and (v) the comment density, computed as CLOC/ELOC, where CLOC represents the commented lines.

[2]See *e.g.,* https://github.com/apache/spark/blob/master/README.md

[3]http://www.arrestedcomputing.com/readability

TABLE I
INFORMATION STORED IN THE DATABASE BY THE CODE ANALYZER

Column	Description
Description	Textual description of the project retrieved from README.md file
DevNumber	Number of contributors
CreationDate	Tha date in which the GitHub repository has been created
ELOC	Effective Lines Of Code: counts the effective lines of code in a class excluding blank lines and comments
LCOM	Lack of COhesion of Methods: a class cohesion metric based on the sharing of local instance variables by the methods of the class. Lower values are preferred (*i.e.,* higher cohesion).
CBO	Coupling Between Object classes: measures the dependencies a class has (a proxy for how hard it is to reuse the class). Lower values are preferred (*i.e.,* lower coupling).
WMC	Weighted Methods per Class: measures the complexity of a class as the sum of the cyclomatic complexity [2] of its methods. Lower values are preferred (*i.e.,* lower complexity).
C3	Conceptual Cohesion of Classes: measures how well the methods in a class are conceptually related. It is computed as the average textual similarity between the pairs of methods in a class. Higher values are preferred (*i.e.,* higher cohesion).
CCBC	Conceptual Coupling Between Classes: measures the coupling between the classes of the system by exploiting their textual content. It is computed as the average textual similarity between a class and all the other classes in the system. Lower values are preferred (*i.e.,* lower coupling).
CR	Code Readability: measures how readable a piece of code is by exploiting a set of features such as the code nesting level, the average length of statements, etc. Higher values are preferred (*i.e.,* higher readability).
CD	Comment Density: measures the density of comments in source code.

Due to space limitations, we provide a short description of the used metrics in Table I. Details about their computation are available in the previously referenced papers.

The computed information in stored in the RETICULA database. The quality metrics are stored for each class in the analyzed project. At the date of writing, the RETICULA database includes information about ∼300k projects for a total of ∼15M classes. The metric values stored are expected to update from time to time.

3) ***Web service****:* When the developer activates the RETICULA plugin, a request is sent to the *web service*, a Java HTTP server acting as proxy between the plugin and the database. The *web service* receives from the plugin a request including:

1) Characteristics of the project under development, including its ELOC, a textual description, its age, and the number of developers working on it.
2) The value of the considered quality metrics (see Table I) measured for the whole project under development in the IDE. Each metric is aggregated at project level by using the average of its values measured across all classes of the project. Future work will be devoted to implement other forms of aggregation (*e.g.,* median).

3) The value of the considered metrics for the specific class under development in the IDE (*i.e.,* the one having the focus in the IDE at the time of the request).

Using this information, the *web service* retrieves from the database projects *similar* to the one under development. What *similar* means here can be customized by the developer using the plugin (Section II-B). In particular, the developer can specify zero or more of the following criteria to be used in the identification of projects similar to the one under development in the IDE (from now on P)[4]:

The project's size. Given S ELOC as P's size, a project in the database is considered similar to P if its size is $S \pm (S \times \delta\%)$, where the parameter δ can be set in the IDE by the developer (Section II-B). The default value for δ is 30%, meaning that a project in the database is considered similar to P if its size does not differ of more than 30% from P.

The project's age. Given A months as the P's age, a project in the database is considered similar to it if its age is $A \pm (A \times \delta\%)$.

The size of the team developing the project. Given T the number of P's contributors, a project in the database is considered similar to P if it is developed by $T \pm (T \times \delta\%)$ contributors.

The project's description. We compute the textual similarity between the description of each project in the database as mined from the `README.md` file and that of the project description provided for P by the developer (see Section II-B). The idea is that projects using the same terms in their description are more likely to implement similar features and/or be related to the same context. The textual similarity is computed by using Information Retrieval (IR) techniques, and in particular the Lucene[5] implementation of the Vector Space Model (VSM) [15]. We apply pre-processing to the text representing the project descriptions to remove English stop words and to stem words to their root form. As it often happens when using IR techniques, it is difficult to establish a similarity threshold to discriminate between projects in the database that should be considered similar to P and those that, instead, should be discarded when this similarity criterion is activated by the developer. Given the high number of projects in our database, we decided to extract from the database only the top 1% of the projects having the highest description similarity when this similarity criterion is selected[6]

Once the *web service* has retrieved the list of similar projects, it collects information about their quality from the database. Quality metrics at project-level—again, computed by aggregating through average the quality metrics for each class of the project—are compared to the P's metrics provided in the request, returning to the plugin for each metric the percentage of similar projects in the database having worse value for that metric (*e.g.,* lower class cohesion) as compared to P.

[4]If no criterion is selected, all projects in the database are considered as similar to the one under development.

[5]https://lucene.apache.org/core/

[6]This means that with ∼300k projects in our database, the top ∼3k will be used as term of comparison for the quality of the project under development.

Quality metrics at class-level for the extracted similar projects are used as term of comparison for the class C under development.

The *web service* provides to the RETICULA plugin, for each metric, the percentage of classes in the similar projects having worse value (*e.g.,* higher complexity) for that metric. Note that, besides the selection criteria applied to identify similar projects, the developer can also set an additional filter to only consider in the class-level comparison classes from the similar projects having a size (ELOC) similar to C. In this case, given S ELOC as the C's size, only classes having a size of $S \pm (S \times \delta\%)$ are considered in the comparison.

B. RETICULA Client Side

To use RETICULA in IntelliJ the developer only needs to install the plugin and restart the IDE. The client side of RETICULA consists of three main components. First is the metrics calculator settings panel shown in Fig. 2 through which the developer can customize RETICULA's behavior.

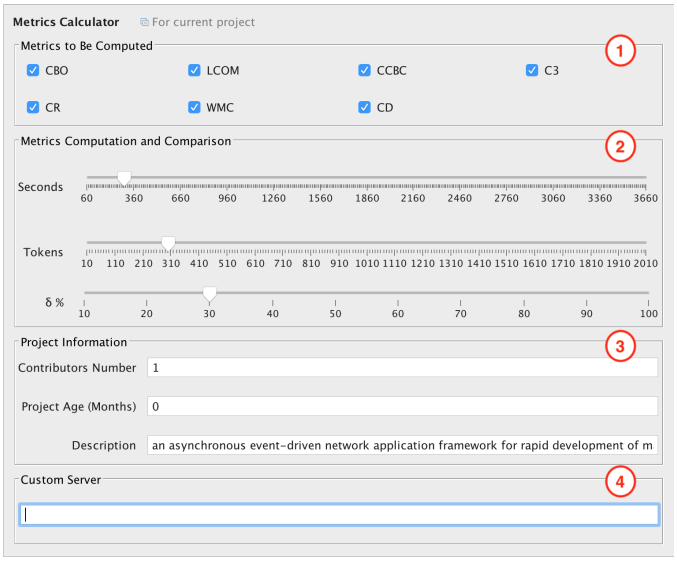


Fig. 2. Metrics calculator settings panel

In the top part of the panel ① the developer can select thorough checkboxes which of the seven metrics supported by our plugin she would like to compute. By hovering over a metric's acronym (*e.g.,* CBO) its full name is shown (*e.g.,* Coupling Between Object classes).

In the second ② part of the panel, the developer can customize the behaviour of the plugin for what concerns (i) the frequency with which RETICULA will update the value of the quality metrics for both the project and the class under development, and (ii) the value for the δ parameter previously described (see Section II-A3). As for the frequency, the metrics are recomputed every X seconds or every Y code tokens added/modified/deleted by the developer. It is sufficient that one of the two conditions is met to trigger the re-computation of the quality metrics.

The "Project Information" part ③ allows the developer to provide information about the project she is working on in the IDE. As previously explained, this information is used by RETICULA to identify similar projects in the database.

Finally, the developer can specify the address of the RETICULA web service ④, currently hosted on a server at our University (anonymized).

The second component allows the user to select the criteria to apply for the identification of similar projects/classes from the RETICULA database (Fig. 3). As previously explained, the criteria are related to the project's size, age, description, and number of contributors and to the class's size.

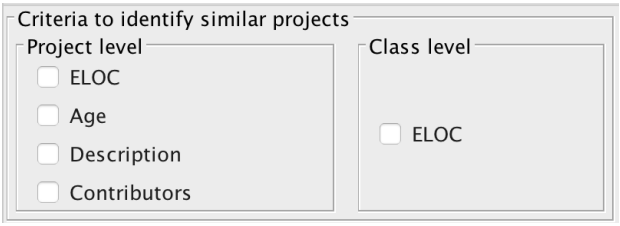


Fig. 3. Criteria used to find similar software projects

Finally, the results of the code quality metrics comparison are shown in two spider plots (Fig. 4) depicting the quality of the project in the IDE (top part) and of the class under development (bottom part) as compared to similar projects/classes identified by using the criteria selected in Fig. 3.

The value on each metric axis indicates the percentage of projects (classes) having worse value for that metric as compared to the project (class) under development. The closer the point to the external edge of the plot, the better the value for that metric. In the example shown in Fig. 4 we can see that the project exhibits good values for the conceptual cohesion of classes (C3), while the coupling (captured by CBO and CCBC) should be improved. The spider plots are continuously updated to reflect changes in code quality while the developer is working on code.

III. RETICULA IN ACTION

In this section we describe a scenario on a real open source project named *Dungeon*[7] to illustrate how RETICULA works. *Dungeon* is a text-based open-world role playing game. The scenario we describe in the following is inspired by the commit `5c16253` performed by the *Dungeon*'s developers.

A developer working on *Dungeon* wants to investigate how good the quality of `CreatureFactory`, one of the *Dungeon*'s classes, is. The developer installs RETICULA and opens the `CreatureFactory` class. Since *Dungeon* is not a very large project, the developer only wants to compare its quality with that of projects having similar size. Therefore, she checks the "Size" criterion in the RETICULA view. In a few seconds, RETICULA presents the results as depicted in Fig. 5(a).

[7]https://github.com/mafagafogigante/dungeon

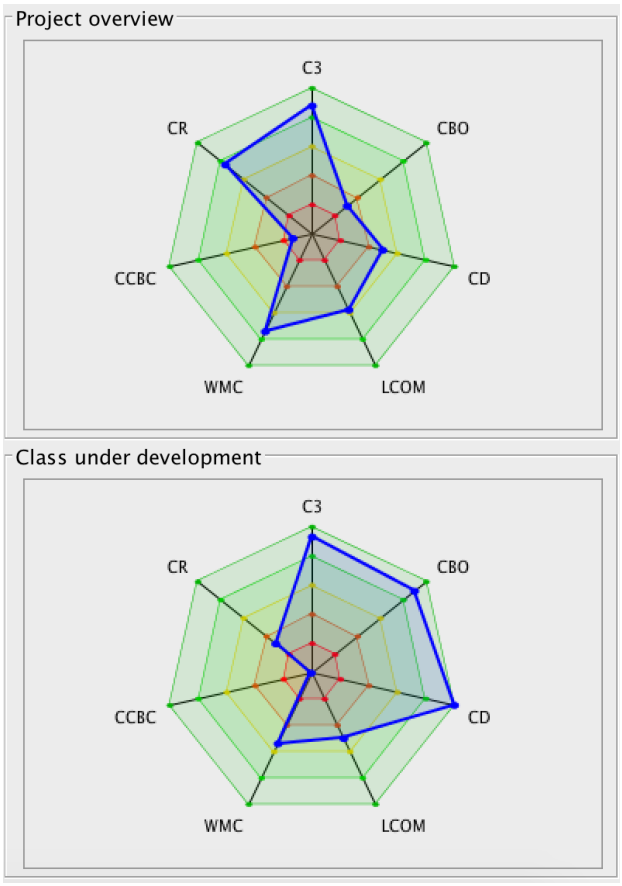
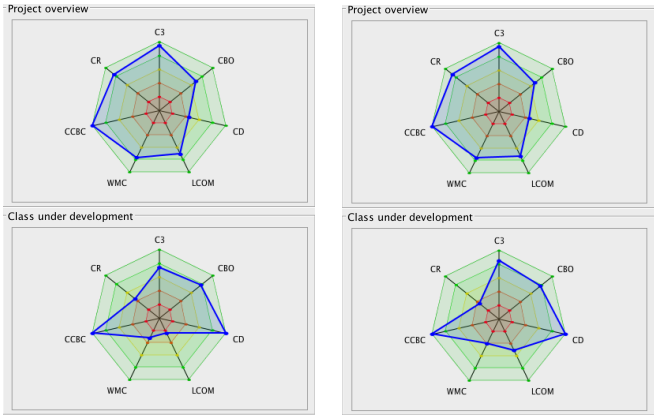


Fig. 4. Visualization of code quality metrics



(a) Before the refactoring (b) After the refactoring

Fig. 5. Code quality view before and after refactoring

The results show the good overall code quality of *Dungeon* as compared to projects of similar size, even if with a limited comment density (CD).

Concerning the `CreatureFactory` class, its quality seems to be suboptimal in terms of cohesion (LCOM) and complexity (WMC). The LCOM is better than only ∼20% of classes having a comparable size in similar projects.

This suggests that the methods in this class are not strongly related. Thus, the developer decides to extract some methods in a newly created class named `JsonCreaturePresetFactory` (*i.e.,* she applies extract class refactoring). RETICULA updates on the fly the quality metrics values for `CreatureFactory` as depicted in Fig. 5(b), showing a strong improvement in the LCOM metric, that is now better than ∼50% of similar classes. Clearly, the changes implemented to the single class only had a minor impact in the project-level plot.

IV. Related Work

Lanza and Marinescu [3] present techniques to grasp a basic understanding of the code quality of a given system using software metrics. Among the proposed techniques, the overview pyramid is the most relevant for our work, which compares the quality of systems written in the same language. RETICULA builds on top of that idea, taking advantage of the millions of open source projects available nowadays and by providing customizable criteria to select the set of projects with which to compare the quality of the code under development.

Many researchers have investigated how to define alarming thresholds for quality metrics. Yoon *et al.* [4] propose to use k-means to automatically detect outlier metric values. Shatnawi *et al.* [5] studied three releases of Eclipse and identified thresholds for several object-oriented metrics by exploiting receiver operating characteristic curves. Alves *et al.* [6], [7] firstly empirically determine metric thresholds from measurement data for different systems. Then, they aggregate individual metrics into a n-point rating scale, providing to developers an intuitive result about how good the code quality is. Ferreira *et al.* [9] study a large collection of open source Java programs, and identify thresholds for six object-oriented metrics. Herbold *et al.* [8] propose a data-drive approach to fix existing software metric values. Instead of traditional thresholds, Oliveira *et al.* [10] propose the concept of "relative thresholds" assuming that a system is in a good state as long as most code entities meet certain requirements, in spite of a few outliers.

The goal of RETICULA is similar to the one of the approaches discussed above: simplify the interpretation of quality metrics. However, we adopt a different approach that does not require the definition of any threshold, but is based on the comparison of similar projects.

Several tools based on code metrics have also been released to assess code quality. This includes SonarJava[8], Coverity Scan[9], and PMD[10]. Unlike RETICULA, these tools do not provide direct and intuitive visualization on code metrics, and they do not compare the quality of the project under development with that of similar projects.

V. Conclusion and Future Work

RETICULA aims to provide real-time feedback on code quality to developers with the support of visualization, seamlessly integrated in IntelliJ IDE.

It provides customizable metrics calculator and projects' filters, which enables developers to compare the quality of their code with that of similar open source projects. RETICULA can serve as a guidance for developers to continuously monitor the quality of their code.

As part of our future work, we would like to exploit the information in the version control system and provide developers with a time-line based code quality view. In that way, developers can have a clear idea about how the code quality is evolving over time. Also, we plan to experiment whether developers using RETICULA tend to write higher-quality code thanks to the real-time feedback it provides.

[8]https://docs.sonarqube.org/display/PLUG/SonarJava

[9]https://scan.coverity.com/

[10]https://pmd.github.io/

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and JITRA (SNF Project No. 172479).

References

[1] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[2] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[4] K. Yoon, O. Kwon, and D. Bae, "An approach to outlier detection of software measurement data using the k-means clustering method," in *Proceedings of ESEM 2007 (1st International Symposium on Empirical Software Engineering and Measurement)*, 2007, pp. 443–445.

[5] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves," *Journal of Software Maintenance*, vol. 22, no. 1, pp. 1–16, 2010.

[6] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proceedings of ICSM 2010 (26th IEEE International Conference on Software Maintenance)*, 2010, pp. 1–10.

[7] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings," in *2011 Joint Conf of 21st Workshop on Software Measurement and the 6th Conference on Software Process and Product Measurement, IWSM/Mensura*, 2011, pp. 20–29.

[8] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Empirical Software Engineering*, vol. 16, no. 6, pp. 812–841, 2011.

[9] K. A. M. Ferreira, M. A. da Silva Bigonha, R. da Silva Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[10] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE*, 2014, pp. 254–263.

[11] M. Aniche, "CK metrics tool," 2016. [Online]. Available: https://github.com/mauricioaniche/ck

[12] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Software Eng.*, pp. 287–300, 2008.

[13] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, 2006, pp. 469 – 478.

[14] R. P. L. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[15] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.