A Heuristic Tool for Measuring Software Quality Using Program Language Standards

Mohammad Abdallah
Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan, Jordan
m.abdallah@zuj.edu.jo

Mustafa Alrifaee
Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan, Jordan
m.rifaee@zuj.edu.jo

Abstract: *Quality is a critical aspect of any software system. Indeed, it is a key factor for the competitiveness, longevity, and effectiveness of software products. Code review facilitates the discovery of programming errors and defects, and using programming language standards is such a technique. In this study, we developed a code review technique for achieving maximum software quality by using programming language standards. A Java Code Quality Reviewer tool (JCQR) was proposed as a practical technique. It is an automated Java code reviewer that uses SUN and other customized Java standards. The JCQR tool produces new quality-measurement information that indicates applied, satisfied, and violated rules in a piece of code. It also suggests whether code quality should be improved. Accordingly, it can aid junior developers and students in establishing a successful programming attitude. JCQR uses customized SUN-based Java programming language standards. Therefore, it fails to cover certain features of Java.*

Keywords: *Java, code review, code inspection, quality.*

Received August 28, 2020; accepted July 12, 2021
https://doi.org/10.34028/iajit/19/3/4

1. Introduction

Quality ascertainment is essential in software development. Software quality is "the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations." [33] According to the IEEE definition, software quality is pertinent throughout the software life cycle: from the early stages to the operational phase and, of course, maintenance and subesequent evolution [52]. Thus, it is quite important to measure software quality. According to [27], software quality measurement is "the process by which numbers or symbols are assigned to attributes of entities in the real world is such a way as to describe them according to clearly defined rules." Therefore, software-quality measurement techniques depend on the stage of the life cycle. For example, code quality can be measured using code inspection tools such as Checkstyle [53] and PMD [47], whereas other aspects, such as requirements and usability, can be measured using different models and tools. Even though several models have been developed to measure software quality, only few use programming-language standards [1].

Programming-language standards represent the best practices for writing a program. Each programming language has its own standards. For example, for Java, there are the SUN standards and the Google Java style [1, 2]. The rules set forth in these standards should be followed to produce high-quality code with minimum ambiguity and misunderstanding [46].

Code-quality measurement and assessment is a crucial problem for software developers stakeholders. However, most software quality issues are discussed using quality-measurement techniques. Accordingly, there is a need to inspect code quality [35].

Quality measurement is predictive and does not focus on the complete system but on the development phase. In previous models, quality measurement was primarily conducted at two levels: management and quality assurance [41].

Static analysis can reduce manual effort by automatically checking for standard coding and style infringements so that code reviewers can focus on more critical tasks, such as identifying logical problems. Even when using static analysis, coders should examine static analytics to determine important issues and to provide feedback on matters that have never been identified by static analysis [51].

In this study, a model for measuring the quality of Java code is proposed. Specifically, it is used to determine the quality level of a piece of Java code according to SUN programming-language standards. A new feature in this model is that it not only measures code quality but also suggests the best practices for rewriting the code following the standards.

The peoposed Java Code Quality Reviewer (JCQR) is an inspection tool that reads a piece of Java code, checks every line using customized Java language standards, and then delivers a report indicating

satisfied and violated rules, with suggestions for improving code quality. These features represent the novelity of the proposed method. Moreover, JCQR is executed in less time; it is a standalone program, not a plugin. Therefore, it can be used more widely and does not require a Java compiler. Moreover, any piece of Java code, even if it is not a complete program, can be checked. Therefore, it can help in regiression testing or preserving the copyright on the program.

The remainder of the paper is organized as follows. In section 2, related work is reviewed, and current issues addressed by the proposed techniques are discussed. In section 3, the proposed technique and the JCQR tool are introduced, explained, and discussed. In section 4, JCQR is evaluated using a small case study and a comparison with similar tools. Section 5 concludes the paper.

2. Related Work

In previous studies, the software quality was considered whether (and how) it can be quantitatively measured [17]. The present study answers this question in the affirmative: There is a quantitative method that can be used to measure software quality. This method, which is proposed in this paper, uses programming-language conventions and standards and provides a numerical value (percentage) to describe the overall quality level.

The most widely accepted quality factors are maintainability, reliability, portability, flexibility, correctness, testability, reusability, efficiency, usability, integrity, readability, and interoperability [4, 14].

In [34], a quality metric was introduced to weight software-quality attributes based on a set of quality, specialized, and application-oriented characteristics. The Thongtanunam *et al*. [56] proposed an algorithm that uses file path similarity between code reviewers to prevent review redundancy.

Code smells [40], which indicate a deeper code problem, were inspected and automatically visualized in (Emden and Moonen [24]). jCOSMO is a framework used to detect and remove code smells in Java.

Code review and inspection is not a new research area. In [25, 26, 45] code inspections for quality measurement were introduced. Code defects and errors are organized, and understandability and maintainability are improved. Code review is a systematic examination that can detect code defects, which can be removed or reported and fixed later. A piece of code is usually reviewed by peers and technical experts, but not by the code developer or author. After the review process, a list of findings is prepared and is formally or informally reported to decision makers that take appropriate action.

Code review is used in various software development projects to improve software quality based on static code analysis. Peer code review is the most commonly used method. It has been demonstrated in several studies that, in most cases, peer review is a useful technique to achieve a satisfactory code quality level [43]. Peer code review is primarily conducted by humans and is affected by their experience and effort. Moreover, code review can foster knowledge sharing that benefits authors as well as reviewers, and improves team collaboration [21].

The Belli and Crisan [12] introduced a semi-automated approach, in which a checklist that facilitates the individual review process is generated. The principle of this approach is the use of a rule-based system by adapting concepts from compiler theory and knowledge engineering for acquisition and representation of knowledge about the program. An important feature of this method is that the checklist can be used to obtain fault classification rules. However, this approach can be applied only to C programs.

Code comments and their effect on the code review process were investigated in [15]. It was demonstrated that more code changes resulted in fewer comments. This has a negative effect on the code review process over time. According to IEEE Std 1028-2008 [32] inspection is "a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications."

Code inspection is the most formal type of code review. It is static testing and is usually based on rules and checklists to avoid defect multiplication at a later stage. Code inspection is aimed at detecting code defects and suggesting possible corrections; it produces reports that can be used to measure and subsequently improve code quality. The report in [13] indicates the successful application of continuous, asynchronous, and distributed code reviews in any context. Inspectors of a piece of code are usually trained moderators, who are not the authors of the code and are expert in the programming language in which the code was written a follow-up meeting, and reviews.

Code inspection is most easily performed through checklists [22], These lists can be designed for a particular development environment using historical error data, contain questions that target specific features, and encourage code understanding. Accordingly, they can aid code inspectors. They are aimed at improving fault-avoiding efficiency by highlighting public areas of previous failures. In practice, although the benefits of using checklists have been quantified in some studies, the statistical robustness of the conclusions has not been satisfactorily demonstrated [44]. However, according to [30] there is no evidence that using numbered checklists significantly improve inspections.

Infrastructure-as-Code (IaC) is a method of introducing continuous delivery by allowing management and supply of code infrastructure by specifying and automating machine-readable files instead of specific hardware configurations or virtual setup software. This methodology can reduce the number of faults and errors when a piece of code is reused. However, it is not clear how the code behind IaC can be preserved, rapidly established, and constantly improved in a measurable manner [19].

Orthogonal Defect Classification (ODC) [28] was introduced to categorize code defects and accordingly generate a checklist that can be used for further investigation. The ODC technique has been demonstrated to reduce resource consumption and improve code inspection performance.

Taba and Ow [55] proposed a scenario-based system that contrasted it to conventional methods. The proposed models are a systematic approach that fulfills the program verification process with several consistency attributes. This study does not include a comparative examination, as testing cannot typically be performed in large-scale enterprises. However, it has been concluded that there is no clear association between software fault severity and frequency [20, 62].

Code inspection is quite useful for improving developer knowledge and skills [38]. It facilitates the generation of code that is easier to understand and modify, as demonstrated in [42]. Moreover, it increases trust in safety-critical software.

However, manual code-inspection techniques have certain drawbacks: They are not attractive to reviewers, are usually not discussed, and do not allow prompt feedback [9, 57]. In [3], a group of software quality measurement models for open-source programs were studied. The programs were written by several unrelated developers. However, different types of models were sought, and the use of programming language standards was not considered. This would aid open-source developers in following common guidelines.

The Ala-Mutka [5] considered various tools that measure the quality of student programs in several different ways. One of them is code style, which can be added to the evaluation section. In [18], properties or factors that affect software quality were discussed from a Chief Information Officers (CIO) perspective. However, the code style was not clearly and directly mentioned. It was only implied from some of the factors addressed. It can be embedded in some of the factors addressed.

The relation between software quality, code-review coverage, and code-review participation was studied in [43]. It was demonstrated that both code-review coverage and assistance are intimately associated with software quality. Low code-review coverage and participation are estimated to produce components with up to two and five additional post-release defects, respectively. In another framework, quality-in-use is adopted. The proposed topic prediction is based on a semantic-similarity technique that determines the resemblance between related quality-review features [7].

The source audit tool [10] introduced a high-level software quality audit. In this technique, the technical quality level is calculated according to The International Organization for Standardization) and The International Electrotechnical Commission (ISO/IEC) 25010, and it is integrated with a derived cost model. The source audit tool is quite useful for managers, who can also interact with their team using the Eclipse plugin to comment on the code. In [39], the effect of source-code quality on business process quality is discussed. Another domain-specific tool is Metaprogramming Language (RASCAL) [37], which is a high-level tool designed to combine different quality factors to measure source-code quality and related technical issues that may affect program quality. The FindBugs tool [8] uses static code analysis to detect code defects. Other than code inspection and code review. Other techniques and tools have been introduced that focus on code reviewer teamwork. The Intelligent Code Inspection in a C Language Environment (ICICLE) tool [16] was developed to inspect code that was written, modified, and tested by a group or team of developers to ensure that all members have the same updated version of the code. Moreover, it records those who write or modify the code. In addition, a crowdsourcing tool [61] was developed to facilitate team code inspection. EduPCR [60] is a tool for code peer-reviewing. It is primarily aimed at students. JFreeChart [49] is an open-source project that uses static analysis of unit tests, which may also contain defects or errors. This tool detects problems in test code and improves its quality. Other code review and inspection tools are presented and analyzed in [58]. In addition, graphs have been used for code understanding and evaluation [50].

As an overall critic of the literature, the authors discovered that the program code analysis and measurement tools, mentioned earlier, does not declaim the junior programmer or the students. Moreover, they address the code style rules violations and mistakes with no guidelines or recommendations for their best practices.

3. Proposed Technique

It was demonstrated in [25, 26] that 54% of the reviewed changes introduced bugs in the code. Moreover, both code-reviewer personal metrics, such as workload and experience, and participation metrics, such as the number of involved developers, have a significant effect on the quality of the code review process. Another empirical Suma and Nair [54] demonstrated that more than 80% of defects can be

removed and programmer time can be saved by implementing software inspections. This facilitates the identification and analysis of factors that can complicate the reviewing process. Accordingly, it is essential to use standards to ensure that the best metric is selected for inspecting a piece of code [21, 23] and to support continuous integration, where further code reviews may be requested [48].

Therefore, it becomes essential to have a tool that review, check, and discover the bugs and code style violation not only in the added and fixed code made after code maintenance.

In this study, we developed the automatic code reviewer JCQR, which is a Java code review tool that reads a piece of Java code, tests it using specific Java language standards, and then submits a report indicating the rules that have been followed or violated, and providing recommendations to improve the quality of the code. Previous methods do not have these features. Furthermore, JCQR is run as a standalone program, rather than a plugin, and has short execution time. Consequently, it can be used more widely and does not require a Java compiler. Moreover, any piece of Java code, even if it is not a complete program or if it has not been compiled, can be checked.

The proposed tool is limited to Java code, since the Java is one of the most used programming language and Java style is supported by big companies like SUN and Google.

The proposed tool is primarily aimed at students and jounior developers. It can provide suggestions regarding the proper application of standards to a piece of code.

In JCQR, a Java program will be assessed using different groups of code standards. Each of these groups has multiple quality rules that the code syntax should satisfy. After JCQR reads the code, and makes an evaluation using the standards, it outputs a report indicating all applied regulations and, among them, those that are satisfied or violated. Moreover, it provides guidelines regarding possible corrections.

The flowchart in Figure 1 shows the process of the JCQR algorithm, which is summarized as follows:

1. Read the Java source code from the UTF-8 text file. Where the code can be written in any style.
2. Store each line in an array list to simplify the search process and enable comparison.
3. Search for style violations. The styles used in this study were selected, from SUN Java style, based on self-learning experience and feedback from four experienced courses Java programming courses offered at three accredited Jordanian universities. Furthermore, we used rules appearing in related literature on software engineering education [29].
4. Classify style violations into groups. The algorithm searches the code line-by-line for rule violations, and if a violation is detected, it is classified into one out of five predefined groups.
5. Display a warning message according to the violated rule. The Java source code is described as violation-free if and only if it adheres to the rules in the groups.

- Group 1 (header and comment): as commenting affects code quality [6], the "header and comment group" was formed. This group contains the rules for writing copyright notices along with programmer name (student name and enrollment number), package statement, and class declaration. Here, the copyright and name information should be surrounded with multiline comments, whereas package and class names should start with uppercase letters to indicate the contributions.
- Group 2 (indentation and white spaces): after the beginning of each block, the indentation should be increased by four spaces and return to the previous level after the block closes. In addition, each block end should be followed by only one white line. Finally, each variable, value, keyword, brace, operation, colon, or semicolon should be followed by one space for better reading (give examples).
- Group 3 (line length): the rules in this group limit the number of line length to 80 characters, and any line exceeding this limit should be wrapped. No wrapping for short routines is permitted. Furthermore, this group forces one statement per line followed by programmer comments even if the comment is empty (screenshot).
- Group 4 (class, method, and variable names): in this group, the names for variables, methods, classes, and predefined class objects are checked. The related constraints are applied to all user-defined data; names must be at least six characters in length. Moreover, method, variable, and object names should start with lowercase characters; uppercase characters are reserved for class names. Finally, this group forces the user to follow the usual naming standards for variable, and predefined class object names. For example, (i) for loop counter variables, (i,j,….) for nested loop counter variables, and (g,e) for graphics and exception class objects, respectively.
- Group 5 (control structures, arrays, and exceptions): the style rules in this group check the Java source for basic programming control (*if*, *else if*, *for*, *while*, … etc.) blocks, where each control statement must be followed by block-begin and block-end braces, even in a single-statement control. In addition, each brace should be followed by a whitespace line. Moreover, array brackets should be declared after the variable name with a single space between

brackets in two-dimensional arrays. Regarding exception handlers, each exception must consist of three blocks (try, catch, and finally) even if some blocks are empty.

6. In the last step, all warning messages are stored in a database and displayed in table format.

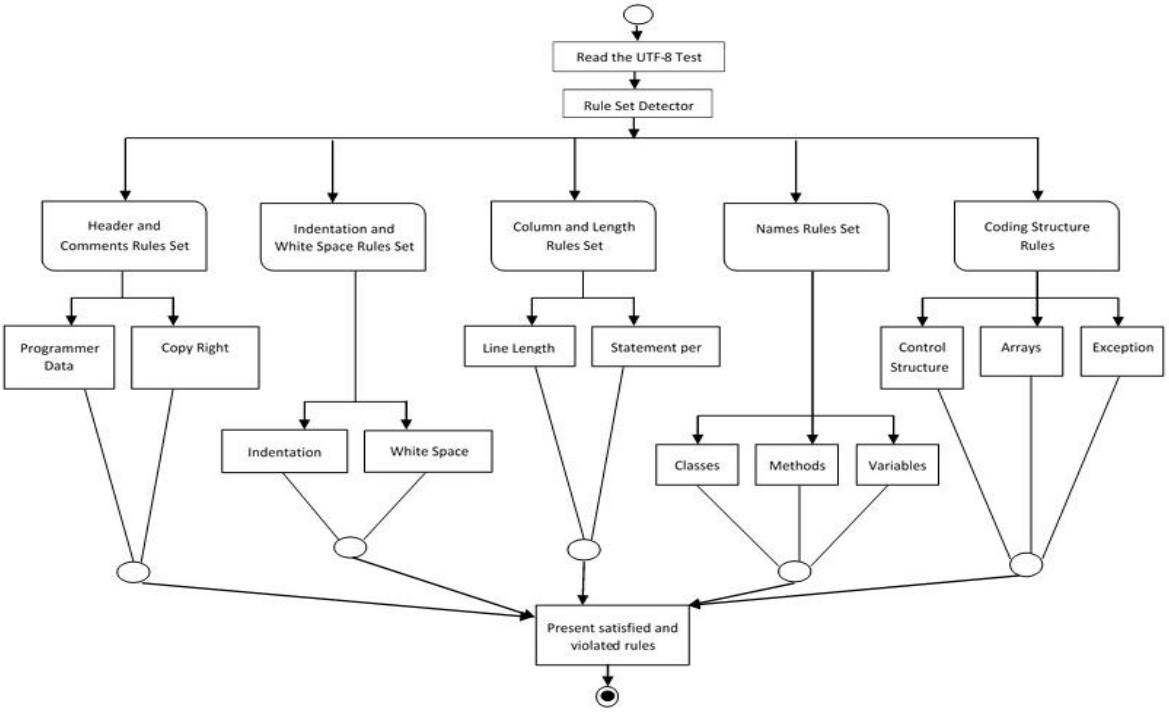


Figure 1. Process of JCQR algorithm.

4. Evaluations

Herein, the JCQR tool was evaluated and compared with other tools using a Java test program [36] stored as a text file in UTF-8 encoding. The program has 3103 lines of code. It was checked using our tool by considering its functionality, classes, packages, and whether it was correctly compiled and run on Java platforms. This piece of code was input to JCQR, which read the code, performed an evaluation using the predefined quality standards, and detected any defects. Moreover, the tool provided advice regarding possible corrections. Additionally, the Java code was tested by other code reviewer tools, namely, PMD and Bug Finder, for comparison. Once the target program file has been imported to JCQR, the code quality reviewer can select any rule groups (discussed earlier) to examine the code. In this study, JCQR was run with all the rule categories. The output summary is shown in Table 1.

Table 1. JCQR test program output summary.

Error Name	Incidence
No rule violation	19
Class names must start with a capital letter.	1
If, for, while, switch must be followed by a "{"in the next line.	852
Line length must be less than 80 characters, the line should be wrapped	40
The second for loop counter must be int j	1
The third for loop counter must be int k	1

As seen in the table below (Table 1), there are various error types. However, some errors appear quite often. All applicable rules, whether satisfied or violated, are shown with the corresponding line number. If a rule is satisfied, this is indicated by the message "No violation detected" or "No 'rule category name' violations", whereas if a rule is violated, the corresponding line number is output along with two messages. The first describes the violation, and the second message suggests a correction. Figure 2 shows an output sample.

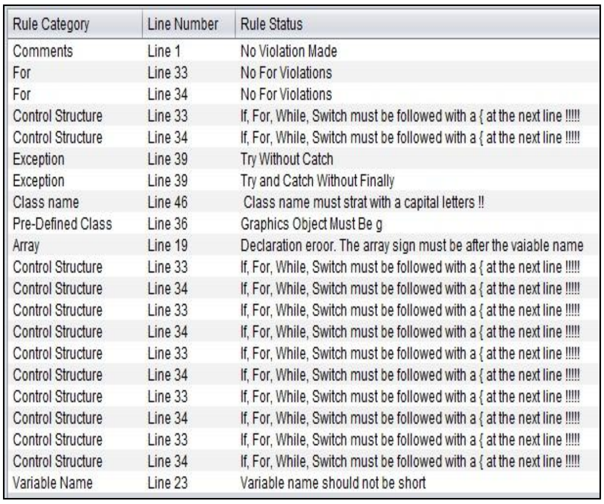
High-incidence errors may indicate programmer weaknesses or pieces of code that require practice. This type of code inspection is quite useful for junior developers and programming students. It can increase their awareness of coding standards and can assist in establishing a successful programing culture. Code inspection tools such as JCQR and PMD can aid developers in reducing the defects in their code, as well as the bugs detected by testers and the quality assurance department. The proposed tool provides students with the opportunity to practice quality inspection of Java code at the early stages of programming learning, as it offers advice and suggestions regarding code defects, thereby assisting in incorporating coding standards. This has also been discussed and proposed as a Pedagogical Code Reviews (PCR) tool [31].

In addition, JCQR facilitates code acceptance in the industry, where change-based code review is gaining

support, demonstrating that the present tool is well aligned with current code reviews [11].

Writing code without following standards is correlated with the number of defects discovered in the reviewing process. Therefore, JCQR, which is based on code standards, may be beneficial in this regard.

However, JCQR has certain limitations with medium and large programs, and it is more useful for simple programs that are used to train and teach students and beginner developers, as it can detect fewer errors than PMD.



Rule Category	Line Number	Rule Status
Comments	Line 1	No Violation Made
For	Line 33	No For Violations
For	Line 34	No For Violations
Control Structure	Line 33	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 34	If, For, While, Switch must be followed with a { at the next line !!!!!
Exception	Line 39	Try Without Catch
Exception	Line 39	Try and Catch Without Finally
Class name	Line 46	Class name must strat with a capital letters !!
Pre-Defined Class	Line 36	Graphics Object Must Be g
Array	Line 19	Declaration eroor. The array sign must be after the vaiable name
Control Structure	Line 33	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 34	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 33	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 34	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 33	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 34	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 33	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 34	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 33	If, For, While, Switch must be followed with a { at the next line !!!!!
Control Structure	Line 34	If, For, While, Switch must be followed with a { at the next line !!!!!
Variable Name	Line 23	Variable name should not be short

Figure 2. JCQR output sample.

Unlike PMD and Bug Finder, JCQR is a standalone, platform-independent tool, and therefore it can be used in different computers with different platforms. Table 2 shows a comparison between JCQR, PMD, and Bug Finder.

Table 2. Comparison between JCQR, PMD, and bug finder.

Example Statistics			
	JCQR	**PMD**	**Bug Finder**
Rule applications	914	Not available	Not available
Rule-satisfaction incidents	19	Not available	Not available
Rule-violation incidents	895	2۲	۱۲

It can be seen that only JCQR indicates the rules that were applied and, among them, those that were satisfied. This facilitates the identification of code strength areas.

It should be noted that the number of violations is significantly higher in PMD and Bug Finder than in JCOR because JCQR uses only a few code writing standards. Therefore, when these standards are not applicable, a rule-violation incident occurs. However, this can be an advantage of JCQR because it focuses on students and junior developers, and it can assist this group in effectively learning programming standards.

In Table 3, the three tools are compared in terms general specifications, providing a general overview. Thus, developers and inspectors may select the best tool. Moreover, the differences of JCQR are pointed out.

Table 3. Inspection tools comparison using general specifications.

General Tool Specifications			
	JCQR	**PMD**	**Bug Finder**
Platform independent/plugin	Platform independent	Plugin	Plugin
IDE	Independent	Dependent	Dependent
Number of available rules	56	332 (PMD Doc)	424 (Doc V 3.0.1)
Shows applied rules (both satisfied and violated)	Yes	No	No
Shows code line of applied rules	Yes	Not Available	Not Available
Input type	Any text file with UTF-8 encoding	Only .java file	Only .java file
User friendly	Suitable for junior developers and students	Not suitable for students.	Not suitable for students
Standards used	Sun rules and customized rules	Sun rules and customized rules	Sun rules and customized rules
Rules Covers	Five groups	All Java features	All Java feature
Code is tested against all rules	Yes	Selected Roles	Selected Roles
Number of supported languages	All Java-like syntax languages	Six languages	Only Java
Version	All	Specific	Specific
Target	Academic (students)	General	General
Naming rules	Multilevel naming rules (I, J, K)	Single	Single
Operating system	Windows/Mac	Windows/Mac	Windows/Mac

5. Conclusions and Future Work

In this paper, the Java code reviewer tool JCQR was proposed. It was developed as a free IDE, offering more dynamic options to developers. It is based on Java code standards and can check any piece of Java code saved as a text file.

It uses five categories of code rules. Once the Java code is entered, it is checked using all the rules of each category. Subsequently, a table indicating all applied rules and, among them, those satisfied or violated is output. In addition, for violated rules, JCQR suggests a correction.

This tool can help students, junior developers, and perhaps senior developers to produce high-quality code and learn how to apply Java rules in their code.

In addition, the JCQR can help researchers who are interested in program analysis, code quality, and regression testing to discover the errors and code style mistakes introduced after maintenance. Moreover, it

can help in defined how different developers who work in the same program are using different code styles.

In future work, JCQR will be updated with more rules that cover the remaining Java features. In addition, we intend to apply it extensively in IT colleges and companies to determine its effectiveness.

Moreover, we may introduce machine-learning algorithms and techniques, as suggested in [59], and use programming language standards as training data to detect code defects and regenerate the code so that it fulfills the standards.

References

[1] Abdallah M. and Al-Rifaee M., "Java Standards: A Comparative Study," *International Journal of Computer Science and Software Engineering*, vol. 6, no. 6, pp. 146-151, 2017.

[2] Abdallah M. and Al-Rifaee M., "Towards A New Framework of Program Quality Measurement Based on Programming Language Standards," *International Journal of Engineering and Technology*, vol. 7, no. 2-3, pp. 1-3, 2018.

[3] Adewumi A., Misra S., and Omoregbe N., "A Review of Models for Evaluating Quality in Open Source Software," *IERI Procedia*, vol. 4, pp. 88-92, 2013.

[4] Ahmed B., Gargantini A., Zamli K., Yilmaz C., Bures M., and Miroslav S., "Code-Aware Combinatorial Interaction Testing," *IET Software*, vol. 13, no. 6, pp. 600-609, 2019.

[5] Ala-Mutka K., "A Survey of Automated Assessment Approaches for Programming Assignments,," *Computer Science Education*, vol. 15, no. 2, pp. 83-102, 2005.

[6] Arafati O. and Riehle D., "The Comment Density of Open Source Software Code," *in Proceedings of 31st International Conference on Software Engineering-Companion Volume*, Vancouver, pp. 195-198, 2009.

[7] Atoum I., "A Novel Framework for Measuring Software Quality-In-Use Based on Semantic Similarity and Sentiment Analysis of Software Reviews," *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 1, pp. 113-125, 2020.

[8] Ayewah N., Pugh W., Hovemeyer D., Morgenthaler J., and Penix J., "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22-29, 2008.

[9] Bader R., Alokush B., Abdallah M., Awad K., and Ngah A., "A Proposed Java Forward Slicing Approach," *Telkomnika*, vol. 18, no. 1, pp. 311-316, 2020.

[10] Bakota T., Hegedüs P., Siket I., Ladányi G., and Ferenc R., "Qualitygate Sourceaudit: A Tool for Assessing the Technical Quality of Software," *in Proceedings of Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse* Engineering, Antwerp, pp. 440-445, 2014.

[11] BaumT., Leßmann H., and Schneider K., "The Choice of Code Review Process: A Survey on the State of the Practice," *in Proceedings of International Conference on Product-Focused Software Process*, Improvement, pp. 111-127 2017.

[12] Belli F. and Crisan R., "Towards Automation of Checklist-Based Code-Reviews," *in Proceedings of ISSRE '96: 7th International Symposium on Software Reliability Engineering*, White Plains, 1996.

[13] Bernhart M. and Grechenig T., "on The Understanding of Programs With Continuous Code Reviews," *in Proceedings of 21st International Conference on Program Comprehension*, San Francisco, pp. 192-198, 2013.

[14] Boehm B., *Characteristics of Software Quality*, North-Holland, 1978.

[15] Bosu A., Greiler M., and Bird C., "Characteristics of Useful Code Reviews: An Empirical Study At Microsoft," *in Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, pp. 146-156, 2015.

[16] Brothers L., Sembugamoorthy V., and Muller M., "ICICLE: Groupware for Code Inspection," *in Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, Los Angeles, pp. 169-181, 1990.

[17] Cavano J. and McCall J., "A Framework for The Measurement of Software Quality," *SIGSOFT ACM SIGSOFT Software Engineering Notes*, vol. 3, no. 5, pp. 133-139, 1978.

[18] Curcio K., Malucelli A., Reinehr S., and Paludo M., "An Analysis of The Factors Determining Software Product Quality: A Comparative Study," C*omputer Standards and Interfaces*, vol. 48, pp. 10-18, 2016.

[19] Dalla-Palma S., Di-Nucci D., Palomba F., and AndrewTamburri D., "Toward A Catalog of Software Quality Metrics for Infrastructure Code," *Journal of Systems and Software*, vol. 170, pp. 110726, 2020.

[20] Dey T. and Mockus A., "Deriving A Usage-Independent Software Quality Metric," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1596-1641, 2020.

[21] Dos-Santos E. and Nunes I., "Investigating The Effectiveness of Peer Code Review in Distributed Software Development Based on Objective and Subjective Data," *Journal of Software Engineering Research and Development*, vol. 6, no. 1, pp. 14, 2018.

[22] Dunsmore A., Roper M., and Wood M., "The Development and Evaluation of Three Diverse

Techniques for Object-Oriented Code Inspection," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 677-686, 2003.

[23] Ebert F., Castor F., Novielli N., and Serebrenik A., "Confusion Detection in Code Reviews," *in Proceedings of IEEE International Conference on Software Maintenance and Evolution*, Shanghai, pp. 549-553, 2017.

[24] Emden E. and Moonen L., "Java Quality Assurance By Detecting Code Smells," *in Proceedings of 9th Working Conference on Reverse Engineering Proceedings*, Richmond, pp. 97-106, 2002.

[25] Fagan M., "Design and Code Inspections To Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

[26] Fagan M., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, pp. 744-751, 1986.

[27] Fenton N. and Bieman J., *Software Metrics: A Rigorous and Practical Approach*, CRC Press, 2014.

[28] Fisher M. and Cukic B., "Automating Techniques for Inspecting High Assurance Systems," *in Proceedings of 6th IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*, Boco Raton, pp. 117-126, 2001.

[29] Hanna S., Jaber H., Abu-Jaber F., Al Shalaby T., and Almasalmeh A., "Enhancing The Software Engineering Curriculums: A Case Study of The Jordanian Universities," *in Proceedings of IEEE 27th Conference on Software Engineering Education and Training*, Klagenfurt, pp. 84-93, 2014.

[30] Hatton L., "Testing the Value of Checklists in Code Inspections," *IEEE Software*, vol. 25, no. 4, pp. 82-88, 2008.

[31] Hundhausen C., Agrawal A., and Agarwal P., "Talking About Code: Integrating Pedagogical Code Reviews into Early Computing Courses," *ACM Transactions on Computing Education*, vol. 13, no. 3, pp. 1-28, 2013.

[32] IEEE: IEEE Standard for Software Reviews and Audits. IEEE Std 1028-2008, 2008.

[33] IEEE, 730-2014, IEEE Standard for Software Quality Assurance Processes, 2014.

[34] Ivan I., Zamfiroiu A., Doineaa M., and Despa M., "Assigning Weights for Quality Software Metrics Aggregation," *Procedia Computer Science*, vol. 55, pp. 586-592, 2015.

[35] Jubilson E. and Sangam R., "Software Metrics for Computing Quality of Software Agents," *Journal of Computational and Theoretical Nanoscience*, vol. 17, no. 5, pp. 2035-2038, 2020.

[36] Katleman., Added tag jdk7u6-b30 for changeset 4bd052837497". Version 1. http://hg.openjdk.java.net/jdk7u/jdk7u6/jdk/file/8c2c5d63a17e/src/share/classes/java/lang/String.java, Last Visited, 2021.

[37] Klint P., Storm T., and Vinju J., "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," *in Proceedings of 9th IEEE International Working Conference on Source Code Analysis and Manipulation*, Edmonton, pp. 168-177, 2009.

[38] Kononenko O., Baysal O., and Godfrey M., "Code Review Quality: How Developers See It," *in Proceedings of IEEE/ACM 38th International Conference on Software Engineering*, Austin, pp. 1028-1038, 2016.

[39] Lad´anyi G., "Business Process Quality Measurement using Advances in Static Code Analysis," *Acta Cybernetica*, vol. 22, pp. 135-150, 2015.

[40] Lafi M., Botros J., Kafaween H., Al-Dasoqi A., and Al-Tamimi A., "Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability," *in Proceedings of IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology*, Amman, pp. 663-666, 2019.

[41] Li Z., Jing X., and Zhu X., "Progress on Approaches to Software Defect Prediction," *IET Software*, vol. 12, no. 3, pp. 161-175, 2018.

[42] Mántylá, M., "Empirical Software Evolvability-Code Smells and Human Evaluations," *in Proceedings of IEEE International Conference on Software Maintenance*, Timisoara, pp. 1-6, 2010.

[43] McIntosh S., Kamei Y., Adams B., and Hassan A., "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of The Qt, VTK, and ITK Projects," *in Proceedings of Proceedings of the 11th Working Conference on Mining Software Repositories*, India, pp. 192-201, 2014.

[44] McMeekin D., Von-Konsky B., Chang E., and Cooper D., "Checklist Inspections and Modifications: Applying Bloom's Taxonomy to Categorise Developer Comprehension," *in Proceedings of 16th IEEE International Conference on Program Comprehension*, Amsterdam, pp. 224-229, 2008.

[45] Parnas D. and Weiss D., "Active Design Reviews: Principles And Practices," *Journal of Systems and Software*, vol. 7, no. 4, pp. 259-265, 1987.

[46] Pecka P., Nowak M., Rataj A., and Nowak S., "Solving Large Markov Models Described with Standard Programming Language," *in Proceedings of International Symposium on Computer and Information Sciences*, pp. 57-67, 2018.

[47] PMD Software 2020; Available from: https://pmd.github.io/, Last Visited, 2021.
[48] Rahman M. and Roy C., "Impact of Continuous Integration on Code Reviews," *in Proceedings of IEEE/ACM 14th International Conference on Mining Software Repositories*, Buenos Aires, pp. 499-502, 2017.
[49] Ramler R., Moser M., and Pichler J., "Automated Static Analysis of Unit Test Code," *in Proceedings of IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Osaka, 2016.
[50] Rodriguez-Prieto O., Mycroft A., and Ortin F., "An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations," *IEEE Access*, vol. 8, pp. 72239-72260, 2020.
[51] Schnoor H. and Hasselbring W., "Comparing Static and Dynamic Weighted Software Coupling Metrics," *Computers*, vol. 9, no. 2, pp. 24, 2020.
[52] Singh D., Sekar V., Stolee K., and Johnson B., "Evaluating How Static Analysis Tools Can Reduce Code Review Effort," *in Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, Raleigh, pp. 101-105, 2017.
[53] Style C., Software, Available from: http://checkstyle.sourceforge.net/, Last Visited, 2020.
[54] Suma V. and Nair T., "Defect Management Strategies In Software Development," *arXiv preprint arXiv*, 2012.
[55] Taba N. and Ow S., "A Scenario-Based Model to Improve the Quality of Software Inspection Process," *in Proceedings of 4th International Conference on Computational Intelligence, Modelling and Simulation*, Kuantan, pp. 194-198, 2012.
[56] Thongtanunam P., Kula R., Cruz A., Yoshida N., and Iida H., "Improving Code Review Effectiveness Through Reviewer Recommendations," *in Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, Hyderabad, pp. 119-122, 2014.
[57] Thongtanunam P., McIntosh S., Hassan A., and Iida H., "Review Participation in Modern Code Review," *Empirical Software Engineering*, vol. 22, no. 2, pp. 768-817, 2017.
[58] Tomas P., Escalona M., and Mejias M., "Open Source Tools for Measuring the Internal Quality of Java Software Products. A Survey," *Computer Standards and Interfaces*, vol. 36, no. 1, pp. 244-255, 2013.
[59] Tsuda N., Washizaki H., Fukazawa Y., YasudaY., and Sugimura S., "Machine Learning to Evaluate Evolvability Defects: Code Metrics Thresholds for a Given Context," *in Proceedings of IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Lisbon, pp. 83-94, 2018.
[60] Wang Y., Li H., Feng Y., Jiang Y., and Liu Y., "Assessment Of Programming Language Learning Based On Peer Code Review Model: Implementation and Experience Report," *Computers and Education*, vol. 59, no. 2, pp. 412-422, 2012.
[61] Winkler D., Sabou M., Petrovic S., Carneiro G., Kalinowski M., and Biffl S., "Improving Model Inspection with Crowdsourcing," *in Proceedings of the 4th International Workshop on CrowdSourcing in Software Engineering*, Buenos Aires, pp. 30-34, 2017.
[62] Ziade H., Ayoubi R., Velazco R., "A Survey on Fault Injection Techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171-186, 2004.

Mohammad Abdallah Received the Ph.D. degree in Software Engineering from Durham University, UK in 2012. The M.Sc. degree in Software Engineering from Bradford University, UK in 2008. BSc in Computer Science from Al-Zaytoonah University of Jordan in 2007. Currently, he is the Director of Technology Transfer Office and an Assistant Professor of Software Engineering Department in Al-Zaytoonah University. His research interests in Quality Engineering.

Mustafa Alrifaee Received the B.Sc. and M.Sc. degrees in information technology from the University of Sindh, Pakistan, in 2001. He received the Ph.D. degree in Computer Science\Multimedia from University of De Montfort, UK, in 2015. Currently he is a member in the Computer Science Department in Al-Zaytoonah.