Strategy for Improving Source Code Compliance to a Style Guide

Pablo Becker, Luis Olsina, and María Fernanda Papa

GIDIS_Web, Facultad de Ingeniería, UNLPam, General Pico, LP, Argentina
[beckerp, olsinal, pmfer]@ing.unlpam.edu.ar

Abstract. This paper illustrates the evaluation and improvement of a Java source code considering the non-compliance with a selected set of items of the Google Java Style Guide. To do this, a strategy was used to understand and improve the Java source code. The strategy has activities that allow specifying non-functional requirements (characteristics and attributes) and designing and implementing measurement, evaluation, analysis, and change. The case was applied in the context of an advanced undergraduate course in System Engineering as a mandatory exam. The evaluation results of attributes' adherence to the aforementioned coding style guide and the improvement of non-compliances are discussed.

Keywords: Java Source Code, Google Java Style Guide, Compliance, Improvement, Evaluation Strategy.

1 Introduction

As stated by Elish *et al*. [6] "The use of agreed-upon coding practices is believed to enhance program comprehension, which directly affects reuse and maintainability". There are agreed coding conventions and style guides for different programming languages that try to improve the readability and maintainability of software source code. Recently, dos Santos *et al*. [4] conducted a field study with a set of 11 coding practices to find out the impact on code readability. Their findings were that 8 out of 11 coding practices had evidence of affecting readability.

Today, it is common for programmers and teams involved in industrial software projects to work with these coding practices. According to Broekhuis [3] "Teams adopt or adapt coding styles, and in some cases, they are mandatory. This means coding practices are an integral part of software development". In 2021, the author conducted a survey with 102 responses from professionals in the Netherlands, including 95 developers, 5 project managers, and 2 testers, and showed that more than 90% of the participants used coding styles in their software projects. As a synthesis of his study, he summarizes "It is, therefore, reasonable to conclude that they [coding conventions and style guides] have a critical role in industries. This could imply the necessity of teaching these coding styles to students".

The present work discusses the evaluation and improvement of a Java source code considering the non-compliance with a selected set of items of the Google Java Style

Guide [7]. This online document serves as the complete definition of Google's coding standards for source code in the Java programming language. As in any other existing coding style guide, in [7], there is a set of items or guidelines mainly in the categories 'Source file structure', 'Formatting', and 'Naming', among others, that the evaluated code must comply with.

The case that we show here was applied in the context of an advanced undergraduate course in System Engineering as a compulsory integrated exam, which regularly lasts around 35 days. The subject called Software Engineering II is taught in the first semester of the 5th year of the degree. The conceptual content of the subject deals with non-functional requirements, measurement, evaluation, and analysis of the quality of a software product or system. To apply these contents and promote the technical and transversal competencies of the students, each year, considering the problem to be solved, an evaluation strategy is selected, from a family of strategies [12].

In the current year (2022), we selected the strategy with the purpose of understanding and improving the quality of a candidate Java source code, considering the compliance of the code with a subset of items of [7]. Note that the code we provided to students was deliberately and slightly modified to partially comply with this coding guide.

The learning objectives were mainly twofold. First, as in any year, apply the concepts of characteristics, attributes, metrics, and indicators, as well as the concept of analysis of the situation for decision-making. These concepts and practices are embedded in the processes and methods of an evaluation strategy. Second, we consider it relevant that students as close future professionals learn software coding styles through practice, as suggested by Broekhuis. In summary, the main contribution of this work is to illustrate both aspects from a practical point of view. Since the study may be of interest to students of other similar degrees, the complete documentation of the case is linked to an additional resource.

The rest of the article is organized as follows. Section 2 overviews the improving strategy and the Google Java Style Guide. Section 3 describes a little more the context of the case study presented. Section 4 shows in detail the application of the aforementioned concepts and practices. Section 5 discusses related work and, finally, Section 6 summarizes conclusions.

2 Overview of the Evaluation Strategy and Coding Style Guide

In [12], a family of evaluation strategies guided by measurement and evaluation activities is presented. Those strategies allow achieving different purposes such as to understand, improve, monitor, compare and adopt, among others. In this work, the strategy called *Goal-Oriented Context-Aware Measurement, Evaluation and Change* (GOCAMEC) is used, which allows us to understand and improve the current state of an entity that in the present case is a Java source code. Fig. 1 shows the GOCAMEC process using the UML activity diagram and the SPEM notation.

As depicted in Fig. 1, the process begins by performing the *Define Non-Functional Requirements* (NFRs) activity (A1), which aims to define the quality attributes and characteristics to be evaluated. A1 has as input a quality model (e.g. those prescribed in [8] and [9]) and produces a "NFRs Specification", which includes "NFRs Tree".

In the *Design Measurement and Evaluation* activity (A2), metrics and indicators are defined or selected from a repository. Then, the *Implement Measurement and Evaluation* activity (A3) implies obtaining the measures and indicator values.

In A4.1 the analysis is designed, which includes, among other aspects, establishing the criteria for the analysis of the results. As seen in Fig. 1, A4.1 can be performed in parallel with A3. Next, in the *Analyze Results* activity (A4.2), the measures, the indicator values, and the "Analysis Specification" are used as input, to produce the "Conclusion/Recommendation Report". The objective of this activity is to detect weaknesses in the evaluated entity and recommend changes.

If there are no recommendations for changes, for example, because the level of satisfaction achieved is optimal, the process ends. But, if there are recommendations for change due to detected weaknesses, *Design Changes* (A5) is carried out, generating an "Improvement Plan" in which the specific changes to be made are indicated. Then, the plan serves as input to *Implement Changes* (A6). The result is a new version of the entity under study.

As shown in Fig. 1, once A6 activity is finished, A3 must be executed again in order to carry out the measurement and evaluation of the new entity. Based on these new results, A4.2 analyzes whether the changes have increased the level of satisfaction achieved by the NFRs. If the improvement is not enough to reach the main business goal, new cycles of change, re-evaluation, and analysis can be carried out until the goal established by the organization is reached.

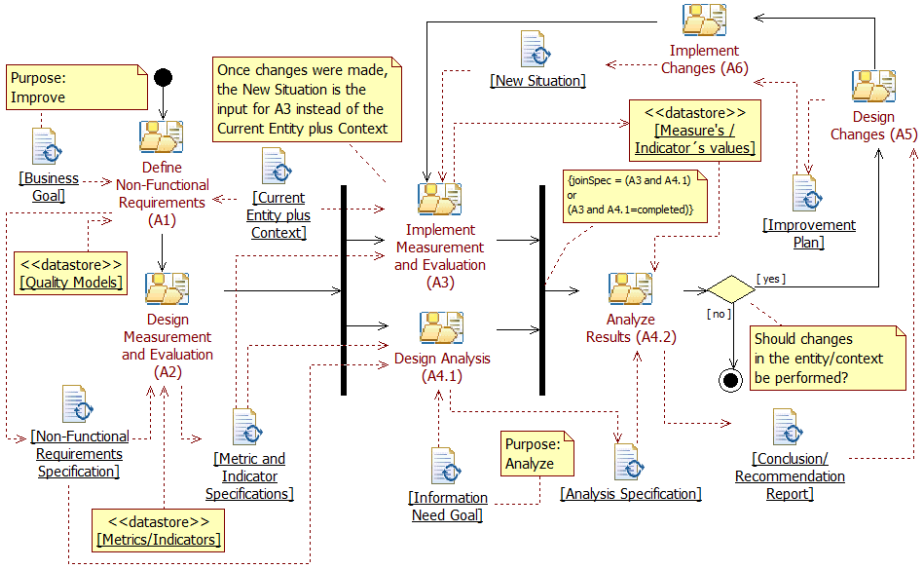

Fig. 1. Generic process specification for the GOCAMEC strategy.

On the other hand, regarding the coding style guide, we use the Google Java Style Guide. This guide includes sections related to: *Source file basics* (that deals with filenames, file encoding, whitespace characters, and special characters), *Source file structure* (that deals with license information, package and import statements, and class member ordering), *Formatting* (that deals with braces, indents, line wrapping,

whitespace, parentheses, enums, arrays, switch statements, annotations, comments, and modifiers), *Naming* (that deals with identifiers such as package, class, method, constant, field, local variable, type variable), *Programming Practices* (that deals with @Override, exceptions, static members and finalizers), and *Javadoc* (pointing out how to format Javadoc and where it is required).

3 More Details of the Context Used for the Practical Case

As commented in the Introduction Section, the case was applied within the framework of the Software Engineering II subject in the first semester of the 5th year of the Systems Engineering degree. The study is illustrated in detail in Section 4 and here we describe a little more about the context.

Firstly, we selected the Java code since the students dedicate about 90 hours to the previous subject called Object Oriented Programming, in the first semester of the 3rd year, using this language to program a video game. The given program for this case named "GUICalculator.java" has 116 lines and is available in Annex IV at http://bit.ly/CACIC_Annexes. We deliberately modified this source code a bit to introduce some violations of the [7] coding conventions. Code lines with at least one evaluated incident are shaded orange in this Annex.

Secondly, the Software Engineering II course assesses students' technical skills on non-functional requirements (using characteristics and attributes to specify non-functional requirements), measurement (using metrics), and evaluation (using indicators) of entities. Note that "establishing software metrics and quality standards" is a specific competency in the new curricular standard in Argentina for Information Systems careers. Thus, the specification of quality requirements and the design and implementation of metrics and indicators are mandatory concepts and practices to pass this course. Consequently, for the presented problem to students, we established that one attribute (as an elementary quality requirement) must be mapped to a single item of the Google Java Style Guide. The maximum number of attributes was 8 mapped to 8 items of the guide. In the present work, we expand the scope of the assignment given to students, by including 11 attributes and by specifying 3 attributes for 1 item of the guide. Specifically, to the "*3.3.3 Ordering and spacing*" item in [7], we evaluate the adherence of the GUICalculator.java code to it by using the following attributes: *1.1.1. Compliance with the ordering of types of imports*; *1.1.2. Spacing compliance between static and non-static import blocks*; and, *1.1.3. Spacing compliance between import sentences*. In total, the present work evaluates 11 attributes mapped to 9 items of the guide. This will be illustrated in detail in the next section.

By allocating this problem to students that represents an integrated exam, we promote group work. In the current year (2022), there were 10 students (one international, by institutional exchange). The sizes of the groups were 1 with three members, 3 with two members, and 1 student who decided to work alone. There were slightly different restrictions regarding the size of the group, such as the number of attributes/characteristics and the size of the monograph as the final report to be examined. For instance, the group of 3 students must have requirements specified by two sub-characteristics of

Compliance [8] and 8 attributes. Additionally, they had to inspect the code beforehand to ensure that the requirements tree included at least 3 attributes that would have to be changed later in the code to fully comply with the guide. The resulting monographs ranged from 33 to 68 pages, including appendices.

Lastly, we give an account of some transversal competencies of the students, encouraging group work, providing all the material in English, and promoting oral and written communication skills in the native language.

4 Application of the Evaluation Strategy to Improve Code Compliance to Google Java Style Guide Items

This section illustrates the application of the GOCAMEC strategy to improve the compliance of the "GUICalculator.java" source code to the Google Java Style Guide. To achieve this goal, the strategy used allows: i) understand the degree of compliance of the source code to the style guide; ii) based on non-compliances apply changes to the current version of the source code (v. 1.0) to improve compliance with the style guide; and iii) understand the degree of source code compliance after the changes (that is, to the version 1.1 of the code). The activities carried out are illustrated below.

Table 1. Google Java Style Guide items mapped to characteristics and attributes related to "*Maintainability*" and "*Compliance*" of a Java source code and its evaluation results (in [%]). Note: The symbol ● means "Satisfactory"; ◆ "Marginal", and ■ "Unsatisfactory". Additionally, op stands for "operator", EI for "Elementary Indicator" and DI for "Derived Indicator".

Google Java Style Guide Items	Characteristics and Attributes (*in italic*)	op	v1.0 EI/DI	v1.1 EI/DI
	1 Maintainability		73.74 ◆	100 ●
	1.1 Compliance	C-+	73.74 ◆	100 ●
3. Source file structure	1.1.1 Source file structure compliance	A	53.33 ■	100 ●
3.3.3 Ordering and spacing	*1.1.1.1 Compliance with the ordering of types of imports*		100 ●	100 ●
	1.1.1.2 Spacing compliance between static and non-static import blocks		100 ●	100 ●
	1.1.1.3 Spacing compliance between import sentences		0 ■	100 ●
3.4.1 Exactly one top-level class declaration	*1.1.1.4 Compliance with the number of top-level class declarations per source file*		33.33 ■	100 ●
4. Formatting	1.1.2 Formatting compliance	*A*	72.10 ◆	100 ●
4.1.1 Use of optional braces	*1.1.2.1 Compliance with the use of optional braces*		9.09 ■	100 ●
4.8.2.1 One variable per declaration	*1.1.2.2 Compliance with the number of variables per declaration*		82.61 ◆	100 ●
4.3 One statement per line	*1.1.2.3 Compliance with the number of statements per line*		88.24 ◆	100 ●
4.4 Column limit: 100	*1.1.2.4 Compliance with the maximum line size*		95.15 ◆	100 ●
5. Naming	1.1.3 Naming compliance	*C--*	82.96 ◆	100 ●
5.2.2 Class names	*1.1.3.1 Class naming compliance*		60.00 ■	100 ●
5.2.3 Method names	*1.1.3.2 Method naming compliance*		100 ●	100 ●
5.2.6 Parameter names	*1.1.3.3 Parameter naming compliance*		100 ●	100 ●

(A1) Define Non-Functional Requirements: For this activity, we consider the quality models prescribed in [8] and [9]. Since adherence to a coding style guide favors the source code maintainability, we use the model for external and internal quality proposed in [8]. This quality model includes the "*Maintainability*" characteristic, which in turn explicitly includes the "*Compliance*" sub-characteristic that was removed from [9]. The "*Compliance*" characteristic is defined as "*The degree to which the software product* (e.g. the source code) *adheres to standards or conventions relating to maintainability*". Then, from the Google Java Style Guide we select a set of items to evaluate (see Table 1, 1st column) and for each item, we define one or more attributes. E.g., for item *3.3.3 Ordering and spacing* we define 3 attributes while for item *3.4.1 Exactly one top-level class declaration* we define a single attribute. The 2nd column of Table 1 shows the identified attributes and their mapping to the guide items. All the characteristic and attribute definitions can be seen in Annex I at http://bit.ly/CACIC_Annexes.

(A2) Design Measurement and Evaluation: In this activity, a set of metrics were defined to quantify all the attributes. E.g., to quantify the attribute "*Compliance with the number of top-level class declarations per source file"* (coded 1.1.1.4 in Table 1) the indirect metric "*Percentage of top-level class declarations per source file*" (%TLC) was defined. Table 2 shows the specification of this indirect metric. Additionally, for each indirect metric, one or more direct metrics were specified. E.g., for the indirect metric %TLC, the direct metric "*Availability of valid top-level class*" (AVTLC) was defined. The measurement procedure for this metric was defined as follows: "*AVTLC = 0; if (class declaration defines a top-level class) and (class name is equal to the source file name) then AVTLC = 1;*"

It is important to say that to clarify the measurement procedures, sometimes some notes were included. E.g., for the previous measurement procedure we include the following notes: "*1. A top-level class is any class that is not a nested class. A nested class is any class whose declaration occurs within the body of another class or interface. 2. The keyword "class" is the tag for any class declaration in Java. 3. Class name and source file name are case-sensitive*".

The rest of the indirect and direct metrics defined for this work can be found in Annex II of the document available at http://bit.ly/CACIC_Annexes.

Table 2. Indirect metric specification to quantify the "*Compliance with the number of top-level class declarations per source file*" attribute coded 1.1.1.4 in Table 1.

Metric Name: Percentage of top-level class declarations per source file (**%TLC**)
Objective: Determine the percentage of valid top-level classes with respect to the total of top-level classes in the source code to be measured. **Author:** Pablo Becker and Luis Olsina **Version:** 1.0
Calculation Procedure Formula: $\%TLC = \left(\frac{\sum_{i=1}^{\#JF}\sum_{j=1}^{\#TLC} AVTLC_{ij}}{\sum_{i=1}^{\#JF} \#TLC_i}\right) * 100$
Scale: Numeric Scale Type name: Ratio Value Type: Real Representation: Continuous
Unit: Name: Percentage Acronym: %
Related Direct Metrics: AVTLC: Availability of valid top-level class; #TLC: Number of top-level classes; #JF: Number of Java files

Since the measured values do not represent the level of satisfaction of an elementary requirement (attribute), a transformation must be performed that converts the measured

value into a new value that can be interpreted. Therefore, for each attribute, an elementary indicator was specified. For this work, the elementary indicator for the attribute 1.1.1.4 is specified in Table 3. The rest of the elementary indicators can be found in Annex III at http://bit.ly/CACIC_Annexes.

Derived indicators were also defined to interpret the requirements with a higher level of abstraction, that is, the characteristics and sub-characteristics documented in Table 1. For all these indicators, an aggregation function named Logic Scoring of Preference (LSP) [5] was used whose function is:

$$DI(r) = (w_1 * I_1^r + w_2 * I_2^r + ... + w_m * I_m^r)^{1/r}$$

where DI represents the derived indicator to be calculated and I_i are the values of the indicators of the immediate lower level, or grouping in the tree, in a range $0<=I_i<=100$; w_i represents the weights that establish the relative importance of the elements within a grouping and must comply with $w_1 + w_2 + ... + w_m = 1$, and $w_i > 0$ for $i = 1 ... m$; and r is a coefficient for LSP operators. These operators model different relationships among the inputs to produce an output. There are operators (op) of simultaneity or conjunction (operators C), replaceability or disjunction (operators D), and independence (operator A). LSP operators for this work are shown in the 3rd column of Table 1.

As shown in Table 3, the elementary and derived indicators have the same three acceptability levels. We decided to use the traffic light metaphor to facilitate the visualization of the levels of satisfaction achieved: ■ red / Unsatisfactory (values less than or equal to 60%), ◆ yellow / Marginal (values greater than 60% and less than 100%) and ● green / Satisfactory (values equals to 100%).

Table 3. Elementary indicator specification for the attribute "*Compliance with the number of top-level class declarations per source file*" (coded 1.1.1.4 in Table 1). Note: %TLC stands for the "*Percentage of top-level class declarations per source file*" metric.

Name: Performance Level of the Compliance with the number of top-level class declarations per source file (PL_TLC)	
Author: Santos L.	**Version:** 1.1
Elementary model: **Specification:** the mapping is PL_TLC = %TLC **Decision criterion (3 acceptability levels):** **Name 1:** ■ Unsatisfactory; **Range:** [0 ; 60] **Description:** Indicates that corrective actions must be performed with high priority. **Name 2:** ◆ Marginal; **Range:** (60 ; 100] **Description:** Indicates that corrective actions should be performed. **Name 3:** ● Satisfactory; **Range:** [100 ; 100] **Description:** Indicates that corrective actions are not necessary since the attribute meets the required quality satisfaction level.	
Numerical Scale: **Value Type:** Real **Scale Type:** Ratio **Unit:** **Name:** Percentage **Acronym:** %	

(A3) Implement Measurement and Evaluation: This activity produces the measures and indicators' values. E.g., for attribute 1.1.1.4 the value was 33.33%. This derived measure is produced by applying the calculation procedure specified in Table 2. All the base measures (used to calculate this and other derived measures) can be seen in Annex V of the document at http://bit.ly/CACIC_Annexes.

Then, the derived measures were used to calculate the values of elementary indicators and the latter to calculate the derived indicators during the evaluation. Indicators' values both for elementary and derived indicators are shown in Table 1, 4th column.

(A4.1) Design Analysis: Concurrently to A3, the A4.1 activity was carried out. In our case, it was decided to classify the attributes following the decision criteria defined for the indicators (see Table 3). Those attributes that fall into the Unsatisfactory range (■) would be the first to receive attention, and then those that fall into the Marginal range (◆). It is important to say that a guide item reaches the Satisfactory level only if all the mapped attributes fall into the Satisfactory (●) level.

(A4.2) Analyze Results: Following the guidelines of the "Analysis Specification", the values of the 4th column of Table 1 were analyzed and improvements were recommended for the attributes with a low level of performance (values marked with ■ and ◆). E.g., under the "*1.1.1 Source file structure compliance*" characteristic there are 2 attributes with a low level of performance. So, for the "*Spacing compliance between import sentences attribute"* (coded 1.1.1.3) which reached 0%■ the recommendation was "*Blank lines between import sentences must be eliminated*", and for the attribute coded 1.1.1.4 which reached 33.33%■, the recommendation was "*Each top-level class must be defined in a file named as the class considering that names are case-sensitive*".

Considering the "*1.1.3 Naming compliance*" characteristic, the recommendation for the "*Class naming compliance*" (1.1.3.1) –which reached 60%■- was: "*All class names must be in upper camel case*". Similarly, recommendations for each attribute under the "*1.1.2 Formatting compliance*" characteristic were made.

(A5) Design Changes: Using the "Recommendation Report" generated in A4.2, the changes to be made were designed. E.g., to improve the level of satisfaction achieved by the attribute coded 1.1.3.1, it was proposed that the classes named "calculatorFrame" and "calculatorpanel" were renamed as "CalculatorFrame" and "CalculatorPanel", respectively. Additionally, to improve the attribute coded 1.1.1.4, it was proposed that the classes named "CalculatorFrame" and "CalculatorPanel" (which are top-level classes) be defined in other source files, which should be called "CalculatorFrame.java" and "CalculatorPanel.java", respectively. All proposed changes were recorded in the "Improvement Plan" document.

(A6) Implement Changes: In this activity, the changes proposed in the "Improvement Plan" were made. The reader can find the new version (v1.1) of the source code in Annex VIII at http://bit.ly/CACIC_Annexes.

As prescribed by the GOCAMEC process (recall Fig. 1), once A5 and A6 activities were completed, a re-evaluation must be performed. Therefore, A3 and A4.2 activities were enacted again to determine the level of satisfaction achieved by the new version of the source code after the changes.

(A3) Implement Measurement and Evaluation: In this second execution of A3, the same metrics and indicators were used on the new source code (v 1.1). The results obtained are shown in the 5th column of Table 1.

(A4.2) Analyze Results: As can be seen in column 5th of Table 1, all the attributes reached 100% (●), that is, the new version of the source code satisfies all the Google Java Style Guide items considered for this work. Since new cycles of change, re-evaluation, and analysis are not required because the goal was successfully achieved, the process is finished.

5 Related Work and Discussion

Coding conventions for programmers to follow have been proposed since the mid-1970s. One of the most cited examples is Kernighan *et al*. [10], which gave many hints on how to write readable code in C language using real software cases.

Particularly for the Java language, the best-known coding style guidelines and conventions emerged from the work of Sun Microsystems in 1997 [15], and of Reddy in 2000 [14], who was also a member of this company. After these proposals, the Google Java Style Guide [7] appeared. We selected this guide for the current case, as it has the main categories and items to make code readable, as well as easy formatting and online access. Another recent reference for Java coding conventions and practices is Bogdanovych *et al*. [2], to name just a few.

Many studies and experiments of different coding styles that affect code readability have emerged, such as those by Lee *et al*. [11], dos Santos *et al*. [4], to mention just a couple of those works carried out so far. As commented in the Introduction Section, according to the survey conducted by Broekhuis [3], out of 102 responses from industry professionals, only 3% did not use a coding style in their software projects. This can emphasize the role that the learning process in academia should continue to play in these beneficial concepts and practices. For the case shown, this was one of the learning objectives established in Software Engineering II.

To the best of our knowledge, what is not present in related works is the mapping of non-functional requirements in the form of characteristics and attributes with categories and items from the coding style guides, as illustrated in Section 4. This mapping enables systematic understanding and improvement of source code compliance by using metrics, indicators, and refactoring as methods for performing the measurement, evaluation, and change activities. In turn, these methods and activities are well established and specified in GOCAMEC. The employment of these concepts and practices was another of the learning objectives established in Software Engineering II. For this learning objective, the use of tools and analyzers was not promoted as is done in other works, but the design of metrics and indicators, and the elaboration of code changes manually from the data recorded from the implementation. However, the implementation of all these methods and activities for code evaluation and refactoring could be automated.

6 Final Remarks

In this paper, we have discussed the quality evaluation and improvement of a typical Java source code by considering the compliance with internal quality attributes properly mapped to a set of Google Java Style Guide elements. To carry out this study, the GOCAMEC strategy was used, which allows us not only to understand the current state of the entity but also to design and implement changes that positively affect the quality of the new version of the entity.

For the problem posed to the students in the context of an undergraduate course, the resulting Java source code was improved by justifying the different steps and results.

Additionally, the learning objectives of the subject and the skills and capabilities expected after passing were commented as well.

To conclude, we would like to highlight that what is not present in the literature regarding related works is the correspondence of quality (compliance) requirements in the form of characteristics and attributes with categories and items of the coding style guides and conventions, as illustrated in the previous sections. In future work, we are planning the automation of the presented approach, which can be an assignment for a student thesis in System Engineering.

Acknowledgment. This line of research is supported partially by the Engineering School at UNLPam, Argentina, in the project coded 09/F079.

References

1. Becker, P., Tebes, G., Peppino, D., Olsina, L.: Applying an Improving Strategy that embeds Functional and Non-Functional Requirements Concepts, Journal of Computer Science and Technology, 19:(2), pp. 153–175, doi: 10.24215/16666038.19.e15, (2019).
2. Bogdanovych, A., Trescak, T.: Coding Style and Decomposition. In: Learning Java Programming in Clara's World. Springer Nature Switzerland, Chap. 4, pp. 83-100, https://doi.org/10.1007/978-3-030-75542-3_4, (2021).
3. Broekhuis, S.: The Importance of Coding Styles within Industries, 35th Twente Student Conference on IT (TScIT 35), pp. 1-8, (2021).
4. dos Santos, R. M., Gerosa, M. A.: Impacts of coding practices on readability. In: International Conference on Software Engineering, pp. 277-285, (2018).
5. Dujmovic, J.: Continuous Preference Logic for System Evaluation, IEEE Transactions on Fuzzy Systems, (15): 6, pp. 1082-1099, (2007).
6. Elish, M., Offutt J.: The adherence of open source java programmers to standard coding practices. In: 6th IASTED International Conference on Software Engineering and Applications, pp. 1-6, (2002).
7. Google Java Style Guide. Available at https://google.github.io/styleguide/javaguide.html, and Last Accessed June (2022).
8. ISO/IEC 9126-1: Software Engineering – Software Product Quality – Part 1: Quality Model, International Organization for Standardization, Geneva, (2001).
9. ISO/IEC 25010: Systems and Software Engineering – Systems and software product Quality Requirements and Evaluation (SQuaRE) – System and software quality models, (2011).
10. Kernighan, B. W., Plauger, P. J.: The elements of programming style. McGraw-Hill, New York, 1st Ed., (1974).
11. Lee, T., Lee, J. B., In, H. P.: A study of different coding styles affecting code readability. Int'l Journal of Software Engineering and Its Applications, 7:(5), pp. 413-422, (2013).
12. Olsina, L., Becker, P.: Family of Strategies for Different Evaluation Purposes. In XX CIbSE'17, Published by Curran Associates, pp. 221–234, (2017).
13. Oman, P. W., Cook, C. R.: A paradigm for programming style research. ACM SIGPLAN Notices 23:(12), pp 69-78, https://doi.org/10.1145/57669.57675, (1998).
14. Reddy, A.: Java™ coding style guide. Sun MicroSystems, (2000).
15. Sun Microsystems: Java code conventions, Available at https://www.oracle.com/technetwork/java/codeconventions-150003.pdf, (1997).