A Tool for Assisted Correction of Programming Exercises in Java Based in Computational Reflection

Original Title: Uma Ferramenta para Correção Assistida de Exercícios de Programação em Java Baseada em Reflexão Computacional

Francisco A. O. Santos[1], Plácido S. das C. Segundo[1], Mardoqueu S. Telvina[2]

[1] Instituto Federal de Educação do Maranhão – IFMA, MA – Brasil
[2] Faculdade de Educação São Francisco, MA – Brasil

ARTICLE INFO

Article history:
Received 18 March 2018
Accepted 25 October 2018
Available online July 1st 2018

Keywords:
Automatic Assessment,
Programming Exercises
Java

ISSN: 2595-9077

DOI:JCThink.2018.V2.N1.p.51

ABSTRACT

INTRODUCTION*: This work reports on the creation and use of a tool to verify compliance in java programming exercises. The solution is based on the hypothesis that computational reflection can provide a way to automatically assess the programing competences of students. The work reflects the concern to make students learning a programming language have practical activities in parallel to what they learn in theoretical classes.* ***OBJECTIVE****: Attesting the effectiveness of using computational reflection to automatically correct programming exercises. Provide the teacher with a tool to support the follow-up of practical activities. Provide students with immediate feedback on their learning, so as to encourage them to behave more autonomously.* ***METHOD****: A case study was carried out with two classes of a computer science course. They answered five practical programming exercises and their responses for each activity were collected in source code format, which were used as the basis of solutions, totaling 100 responses. A comparative analysis was made between the scores obtained through CodeTeacher and the scores assigned by a group of teachers.* ***RESULTS****: Comparing the evaluation in CodeTeacher and the scores assigned by teachers, the average between the pairs of evaluations was lower than the confidence level of significance established in three groups, which demonstrates that the automatic correction obtained an acceptable accuracy.* ***CONCLUSION****: The use of computational reflection techniques for assisted correction in programming classes can bring beneficial result. Teachers can optimize their work and have performance reports of their students. Students can also be benefited by having an immediate feedback, so they can perceive themselves capable of achieving the learning objectives defined by the teacher.*

1. Introduction

Practical exercises are essential to the development of computational thinking. In programming classes, learning to code requires related knowledge, such as: notions of logic, programming techniques, correct use of syntax and resources of a programming language, and application of best practices in software development, among others. The evaluation of these issues requires a thorough analysis by the teacher, since it is necessary to review the code produced by the students to assess the knowledge acquired, usually materialized in the form of practical works [Prior 2003]. Due to the amount of details to be observed, individual monitoring is necessary for a more accurate learning [Tobar et al., 2001].

However, evaluating the student's knowledge from code analysis constitutes a challenge for programming teaching, especially when the classes are extensive and the number of classes per teacher too - a common reality in programming courses [Oliveira et al. 2015]. According to França et al. (2011), technical programming disciplines are extensive in Computing and Engineering courses. This reality ends up hampering, if not hindering, the evaluation capacity of the teacher, given the large volume of work to be corrected and the time restrictions for correction and delivery of scores [Nunes 2004], besides being a repetitive, laborious activity that little adds to the teacher. Because it causes an overload of activities to the teacher, such factors tend to affect the quality of assessments.

According to França et al. (2011) this difficulty can lead to discouragement, sometimes impelling the class to dispersion. Given this difficulty, alternatives to optimize this process have arisen through the automation and execution of source code tests [Hollingsworth 1960, Ebrahimi 1994]. The attempt to replace the visual analysis and manual execution of programs brought to light tools of assisted correction of programming works. Peterson et al. (2015) consider the use of software metrics based on the student code a revolutionary strategy for teaching. However, although there has been a considerable evolution of these systems [Romli et al. 2010], according to Oliveira et al. (2015), there are still considerable shortcomings with regard to the effectiveness of these tools, especially in assessing whether the desired educational objectives have been achieved. For Ihantola et al. (2010), these systems need to evolve in the sense of providing greater direction from a pedagogical perspective.

The main objective of tis paper is attesting the effectiveness of using computational reflection to automatically correct programming exercises, to fulfill this purpose, this article presents a case study for an automatic source code analysis and evaluation tool, as an alternative for optimizing the teaching-learning of programming process. The tool chosen for this purpose is CodeTeacher [Santos et al. 2017], a software for automatic analysis and evaluation of source code. In order to assist the assessment of programming exercises made with the Java language, the tool enables automatic evaluation of Java classes, as well as support for the assignment of scores to the evaluated activities. In addition, it generates reports of the executed tests and assigns scores to the tested classes.

The main contribution of this work is to promote discussion in the educational computing community about teaching and learning programming and propose a solution to find new approaches for this challenge. In presenting evidence of using computational reflection as a key mechanism for evaluating novice students performance in programming, we believe to be encouraging the development of technologies that may be an auxiliary work tool to understand students' learning difficulties and guide decision making of teachers in the choice of teaching actions to improve learning, thus contributing to the progress of computational thinking.

The text is organized as follows: Section 2 lists the objectives of this work. In Section 3, we present the theoretical background needed to understand the mecanism, bringing related works and similar solutions. In Section 4, we present the CodeTeacher and describe the types of analysis that compose it. In Section 5, we explain the operation of the tool. Section 6 provides a case study to ascertain the viability of the tool. And finally, in Section 7, we conclude with the final considerations and future work.

2. Theoretical Reference

This section introduces the basic concepts, methods, strategies, tools and techniques already used for assessment of student programming competences. It also discusses the related works and its results.

2.1. Automatic Evaluation Approaches

According to Ala-Mutka (2005), the most used techniques for automatic correction of programs are static and dynamic analysis. Static analysis is an approach to automatic assessment of programming learning based on source code analysis. Through static analysis, it is possible to analyze effort, complexity, efficiency and quality of programming [Curtis et al. 1979b, Berry and Meekings 1985, Rahman et al. 2008]. Dynamic evaluation is an evaluation based on the correct and efficient execution of

programs. Oliveira et al. (2015), introduces the dynamic-static analysis for automatic assessment of source code.

Naude et al. (2010) propose a method for evaluating programs using graph similarities. The programs developed by students are normalized in abstract syntatic trees and the scores are assigned by linear regression based in previous solutions. The results indicated a high similarity of scores assigned by teacher in comparison with those obtained automatically, but only when the student get higher scores.

Oliveira et al. (2016) report an approach that combines Principal Component Analysis (PCA) algorithyms and clustering techniques to recognize examples of solutions in responses developed by students. The experiments led to a rubric scheme that requires a little assessment effort by teachers.

Benford et al. (1995) and Vujosevic-Janicic et al. (2013), published strategies combining testing techniques, analysis and prediction. The results of both works are very promising.

Estey and Coady (2016) presented BitFit, a submission tool for activities from which a predictive model was built based on classification binary to identify the probability of a student being failed.

Otero et al. (2016) proposed a set of software metrics based on the static analysis of student codes. The study verified that the metrics have a correlation with students' grades.

2.2. Similar Solutions

Among the many tools to support the practice of programming with the purposes of submission, execution and evaluation of exercises, we highlight ProgTest [De Souza et al. 2011], PCodigo [Oliveira et al. 2015], BOCA [Campos and Ferreira 2004, França et al. 2011] and MOJO [Chaves 2013].

ProgTest is an automated support system for evaluating submissions of programs written in Java. Along with the programs are also submitted their respective test cases. ProgTest compiles the student programs and submits them to the tests.

PCodigo is a complement to the Virtual Learning Environment (VLE) Moodle [Moodle 2011] for mass execution and program analysis, developed in C language, but it is applicable to different programming languages. Integrated with Moodle, it receives solutions from programming activities submitted by students, executes them and issues evaluation reports for teachers.

BOCA Online Contest Administrator is an internet system for exercise submission and online code correction. It is the current platform used in programming competitions promoted by the Brazilian Computer Society (SBC).

MOJO is a tool that integrates the concept of Online Judges (OJ) [Kurnia et al. 2001] to Moodle. It consists of a module that aims to assist the teacher in the process of preparing, submitting and correcting programming questions.

The differential of CodeTeacher in relation to existing solutions, besides being focused on Java, is that it consists in giving more flexibility to the evaluation of the teacher, allowing a more holistic analysis. In addition to guiding evaluation through a pedagogical perspective [Ihantola et al. 2010]. It also has the characteristic of being extensible, that is, its modular architecture favors the inclusion of new features.

Consequently, with the possibility of providing immediate feedback to the student, the tool has the potential to get more involved in this, since the students' awareness of their situation tends to cause a reaction towards better results. In addition to enabling greater transparency, making the evaluation process visual, exposing the results and reporting to all those interested in learning.

For the teacher, there is also the possibility of perceiving the deficiencies of a class through the identification of recurrent errors in the same evaluation, as well as the possibility of a closer monitoring of the progress of the discipline, with greater clarity about the individual and collective income of the class.

3. CodeTeacher

CodeTeacher is a platform-independent desktop application based on static, dynamic and static dynamic analysis [Oliveira 2015]. It is a system for source code analysis by the configuration of pre-defined evaluation criteria. The application was made to accept projects in Java, regardless of the development IDE used, since it only needs that compiled classes (files with a .class extension) to be submitted to the tool.

CodeTeacher proposes to help teachers and students in the evaluation of programming exercises. To fulfill this purpose, some requirements were defined. These goals are listed as follows:

- Providing the teacher with a tool to support the follow-up of practical activities;
- Giving flexibility to the evaluation of the teacher, allowing a holistic analysis.
- Guiding evaluation through a pedagogical perspective.
- Being extensible.
- Providing programming students with immediate feedback about their learning, so as to encourage them to behave more autonomously.

For automatic detection of nonconformities in code from the definition of criteria available to the teacher, it is used the reflection programming or metaprogramming [Horstmann 2000]. This paradigm provides the ability of a computer system to access information about itself to examine its structure, state, and representation, and to be able to self-modify its behavior at runtime. This functionality is provided by some programming languages, including Java.

The strategy adopted to evaluate the classes without changing the code developed by the students was to intercept the creation of objects and the invocation of methods on the test target class, verifying the occurrence of nonconformities and checking the return after its execution.

The current focus of using CodeTeacher concerns initial and intermediate programming disciplines. It is also important to note that the tool only covers practical coding works in Java, not embracing textual analysis of responses to exercises and subjective questions.

Four types of evaluation are possible, named as follows: structural analysis, behavioral analysis, standard export analysis and conceptual analysis. The following are the types of CodeTeacher analysis.

3.1. Structural Analysis

In this type of analysis, the static elements of the code are verified, such as the correct declaration of the attributes and methods of a class, and its modifiers can also be checked, as well as whether a given member is a class or instance.

Another possibility is the analysis of the use of inheritance through navigation by the hierarchy of classes implemented by the student. Similarly, this analysis allows us to verify whether or not a particular class implements a given interface. It is also possible to make the analysis more flexible considering only some elements in the evaluation. For example, it is possible to configure a criterion that evaluates whether there is a method in the defined scope that returns data and / or receives parameters of certain types, and does not need to enter the name of the method to be searched for.

In order to make the student's implementation more flexible, without, however, harming the effectiveness of the evaluation, it is possible to use regular expressions, using wildcards in the specification of criteria. It is possible, for example, define that there must be a method whose name starts with a prefix such as "register ...". For this, the teacher would only need to enter the following expression in the method name: "register.*", where the asterisk (*) would mean any string. Similarly, it would be possible to verify the existence of a method with the suffix "?data", with the question mark corresponding to any character. Table 1 shows the predefined types of common errors in structural analysis.

Error	Description
CLASS_NOT_FOUND	A required class is not in the project
PARAM_CLASS_NOT_FOUND	Some class that is used as parameter is not in the project
SUPERCLASS_NOT_FOUND	A specified extended superclass is not found in the project
METHOD_NOT_FOUND	A method specified in the criteria is not declared in the class
FIELD_NOT_FOUND	A field specified in the criteria is not declared in the class
CONSTRUCTOR_NOT_FOUND	A specified constructor is not present in the class
INTERFACE_NOT_IMPLEMENTED	A class should implement some interface
BIN_NOT_FOUND	The source folder with the compiled classes was not found
FOLDER_NOT_FOUND	The project folder is not in the specified directory
METHOD_NOT_ABSTRACT	A method should be declared as abstract
METHOD_NOT_FINAL	A method should be declared as final
METHOD_NOT_FOUND	A specified method should be declared
METHOD_NOT_PRIVATE	A method should be declared with the private modifier
METHOD_NOT_PUBLIC	A method should be declared with the public modifier
METHOD_NOT_STATIC	A method should be declared with the static modifier
METHOD_NOT_PROTECTED	A method should be declared with the protected modifier
METHOD_MODIFIER_MISMATCH	The method modifiers declared mismatches the criteria
METHOD_NOT_RETURN	The method return is not the same specified in the criteria

Table 1. Types of structural error.

3.2. Behavioral Analysis

For Ala-Mutka (2005) and Rahman et al. (2008), fairness and functionality are important evaluation items. The behavioral analysis consists of attesting the correctness of the code from its functional testing in order to simulate the behavior of the program in a real environment or scenario. It is an analysis based on input / output that tests the services provided by an object, that is, verify the correct output of a program from inputs previously provided and compared with predicted results. The code is executed and the responses of the object to external stimuli are checked, so messages are passed to an object in order to find an expected response. The teacher models a set of cases to be evaluated and submits them to the evaluation of the tool and, through the combination of the provided inputs and the expected outputs, it is possible to infer the quality of the program with respect to its functionalities, to the requirements specified by the teacher.

An example of using this approach may be checking the values returned from invocation of methods by passing a list of pre-established parameters and defining the expected values that should be returned. It works like black-box tests, when values are entered and specific outcomes are expected, depending on the test case. No intervention in the implementation of the solution is made, the code is tested based only in its outcomes.

3.3. Standard Output Analysis

In this type of analysis it is verified if the executed code performs the printing of some text in the standard output of the system (console). It is common in early programming disciplines to create programs that write data or messages to standard output. They are usually data resulting from calculations made by the application or even informational messages. Typically this information is presented in a textual way in a command line interface. The tool captures the standard output, interrupting the print flow and diverting it to a proxy that stores the printed content, and then checks whether what is printed by the student program is equivalent to the text defined in the evaluation criteria set by the teacher. It must inform the content to be printed in order for the equality comparison to be made and, in this case, the exact match between the terms compared is considered success.

3.4. Conceptual analysis

In this type of analysis, standards and metrics can be defined to be contemplated. This mechanism allows the evaluation of the application of Object Oriented Programming (OOP) concepts such as inheritance, polymorphism, degree of encapsulation, among others [Horstmann 2000]. In this way, it is possible to find out if there are abstract classes that are not extended or interfaces not implemented, for example. The use of polymorphism can be identified and evaluated considering, for example, the presence of overloaded and / or overwritten methods. Such factors can be framed in a gradation scale defined by the teacher. As an example, the level of encapsulation can be measured by configuring a minimum percentage of encapsulated members to be achieved.

Although not yet available in CodeTeacher, other concepts and metrics can be used in this type of analysis, such as degree of cohesion and coupling, as new metrics and concepts from Software Engineering can be added to the tool in addition to functionality, with the inclusion of plug-ins.

4. Evaluation Steps

All the details involving the total evaluation process are detailed in the following topics. The flow of activities is shown in Figure 1.

Figure 1. Evaluation process

Figure 2 shows the first screen of the application, with all the options available. The sessions, menus and buttons are presented bellow.

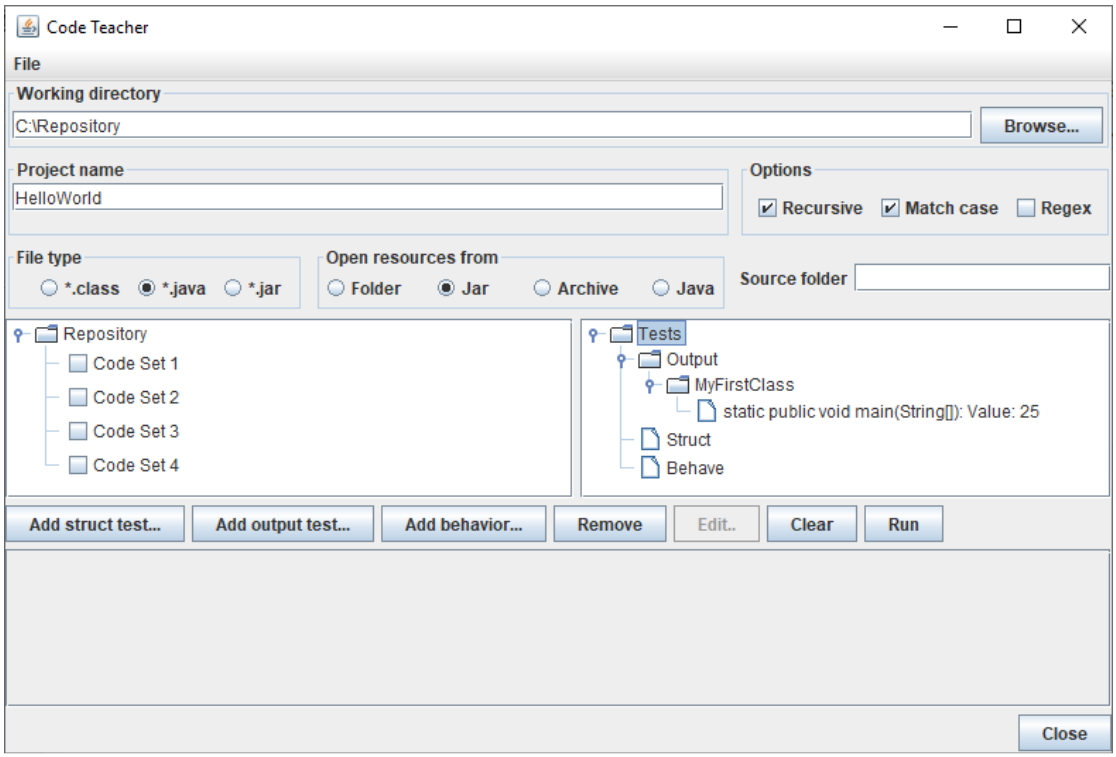

Figure 2. Application home

4.1. Selecting artifacts

This step indicates which elements will be chosen for evaluation. Assets are explicitly selected by the user, firstly by indicating which directories should be accessed to look for the code artifacts to be analyzed, being it possible to be, projects, classes, or class packages. This indication determines the scope of the assessment. The convention used by CodeTeacher to associate a particular set of code artifacts with a particular student is that all files contained in a folder belong to the same student. Therefore, it is recommended that there be a folder with the name of each student, because the name of the folder will serve as reference to identify the student.

4.2. Setting Criteria

Once the software elements have been selected, the items that will compose the evaluation are then created. Thus, a set of criteria must be informed by the teacher to be applied during the analysis. The criteria are defined using the graphical interface, where the evaluative items considered are indicated. The format of the criteria varies according to the type of analysis, but each criterion must have a value to compose the student's grade. The assessment of the criteria is established as weights are assigned according to the degree of importance considered by the evaluator. It is at this moment that the teacher feeds the system with his previous judgment of the expected competences of the student with the accomplishment of the practical activity in question, in the sense of reaching the established learning objectives.

Once the criteria have been created, it is possible to save them in a file for later use or editing, thus preventing the work of setting up all the criteria from being repeated in case of a later evaluation. Figure 3 shows the criteria configuration, note that the field name as “value” is the number of scores of the criteria.

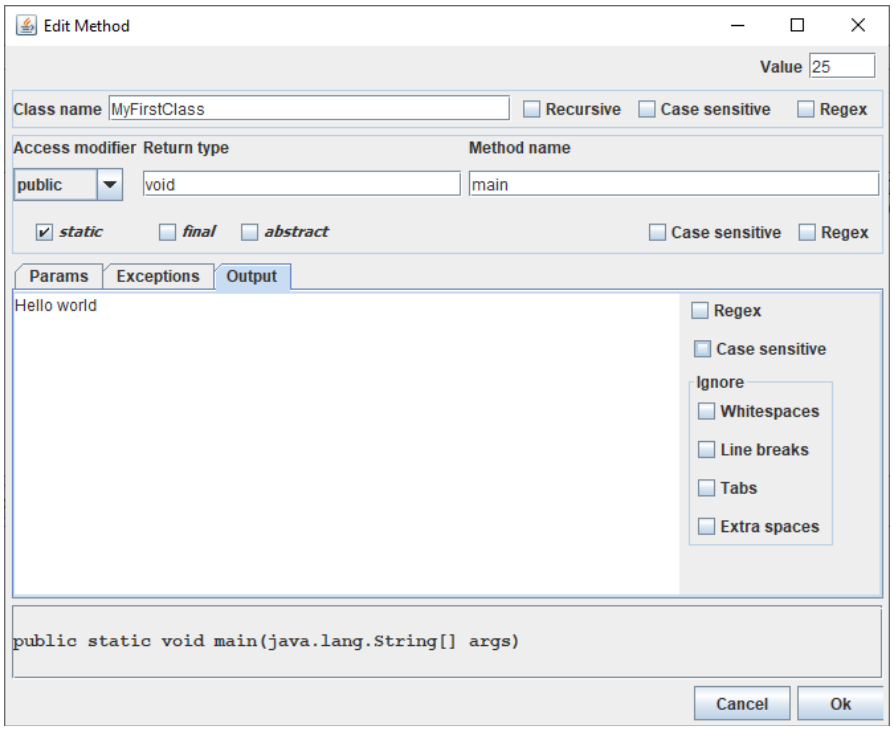

Figure 3. Criteria configuration

4.3. Running tests

The implementation strategy is defined by the teacher, in order to make the evaluation process flexible. The performance of the tests can be in mass or class to class, and can be changed later, so that it is possible to apply differentiated evaluations and focus on certain aspects, at the discretion of the teacher. Students' grades are calculated according to the total number of points obtained through the sum of all the criteria. It is considered that, before the start of the execution, each student has all points, but as irregularities are found in his code, the points corresponding to the criteria not met are debited from the total points of the student. Finally, the final grade is counted based on the student's percentage of correct answers. Figure 4 shows how the results are reported.

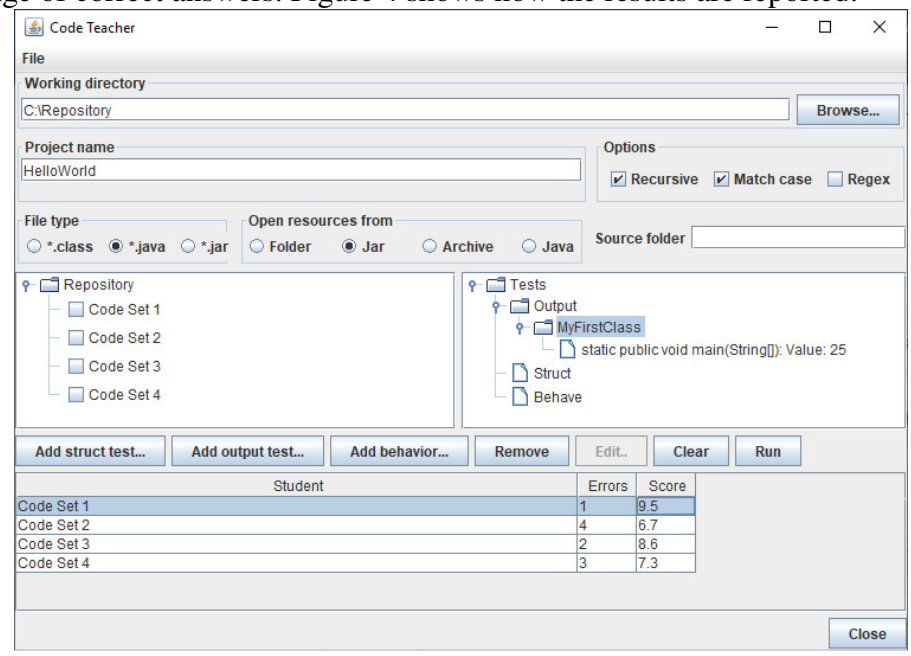

Figure 4. Execution of tests

4.4. Performance report

After the completion of all the tests, comes the scores prediction phase. The results are tabulated and a performance report is generated showing the grades of the students in the class. Initially the report is presented in its summary form, containing a summary of the scores, with statistical information such as class average and standard deviation. However, it is possible to analyze individual performance by obtaining a detailed result of each student. Other visualization options are also available, such as an extract of the most committed types of errors, percentage of correctness, rate of recovery, among others. With this complementary information it is possible to identify generalized difficulties in the class and to plan strategies to remedy such deficiencies.

At this stage, there is also an option to export the resulting data to a spreadsheet, if it is necessary to manipulate such data for graphing, for example, or even to report to students about their performance. The generation of the file also serves as documentary evidence to indicate the evolution of the class in the course of the discipline taught. Figure 5 shows an individual performance summary.

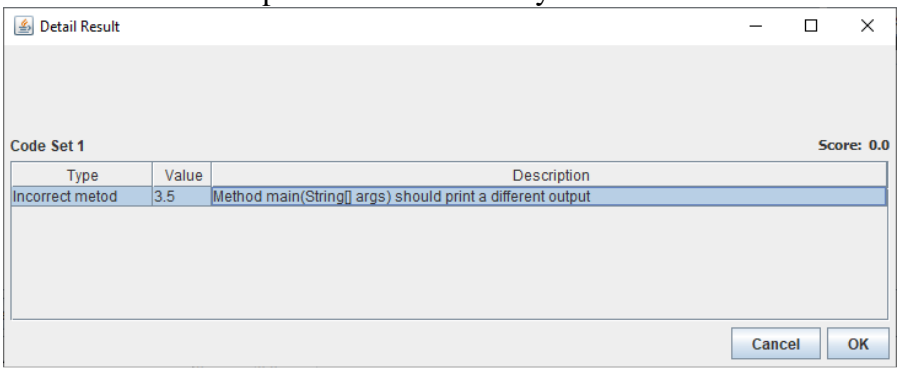

Figure 5. Individual performance summary

5. Case Study

A case study was performed to prove the effectiveness of CodeTeacher as a learning object. A comparative analysis was made between the grades obtained by CodeTeacher and the grades assigned by teachers from an actual school. A sample of responses to programming exercises questions from a regular group of computer science course students was selected to compose the test set. Four practical programming activities (A, B, C and D) were selected to compose the corpus of the experiment. For each activity, responses were collected in source code format, which were used as the basis of solutions, totaling 100 responses.

The responses were submitted to structural, behavioral, output and conceptual analysis, respectively. The final grade was the arithmetic mean among these scores. For each analysis, evaluative criteria were elaborated as model responses, which took into account aspects considered relevant in a student beginning programming, involving the basic use of the main Java elements, such as language syntax, procedural aspects and basic guiding resources to objects. The values of the criteria were assigned individually and arbitrarily by tree teachers. If there were divergences in the value attributions of each teacher, they were confronted and underwent a review process where the two discussed until reaching a consensus.

Subsequently, the answers were evaluated manually by the same tree teachers, who assigned grades on a scale 0-10. Each teacher received a checklist with the evaluative items identical to the criteria defined in the application. The teachers deliberately indicated each item as "Complied" or "Not complied" and the calculation of the grade was obtained from the percentage of complied items. This set of exercises

already evaluated composes the basis of comparison of the experiment. In this experiment we have two factors, the grade and the evaluators, and we hope that there is no significant effect of evaluators.

In order to investigate the influence of the evaluation method on the variation of the grades assigned, the One-way Analysis of Variance (ANOVA) [Rumsey, 2016] was adopted. The ANOVA is a statistical technique for analyzing data by comparing the means of subsets of the data, the data is sub-divided into groups based on a single classification factor. The dependent categorical variable considered was the evaluation author (the human teachers evaluation - named by their initials as EP, GA and PS - and the CodeTeacher evaluation - named as CT). The four evaluations were compared, and the goal was to verify the difference of grades with the control, CodeTeacher being the reference.

In order to identify the groups in which the difference was significant, the Tukey's test [Rumsey, 2016] was performed. Tukey's test is a statistical method that compares all possible pairs of means to find means that are significantly different from each other. The results were compared according to their categories, that is, the evaluator responsible for each evaluation.

To determine whether any of the differences between the means are statistically significant, a significance level (denoted as α or alpha) of 0.05 was adopted. That indicates a 5% risk of concluding that a difference exists when there is no actual difference. To assess the hypothesis that the population means are all equal, we compare the p-value to our significance level. The p-value is the smallest familywise significance level at which a particular comparison will be declared statistically significant.

- P-value ≤ α: The differences between some of the means are statistically significant
- P-value > α: The differences between the means are not statistically significant

Group	Diff	Lower	Upper	P-Value
EP-CT	1,53	1,68	9,38	0,67
GA-CT	1,57	1,71	9,42	0,92
PS-CT	1,02	1,17	8,73	0,09
GA-EP	3,58	1,84	9,12	0,22
PS-EP	5,11	3,63	6,60	0,00
PS-GA	5,47	3,98	6,96	0,00

Table 2. Exercise A.

In Table 2, the p-values for the comparisons with CT are greater than the significance level, wich confirms the hypothesis and conclude that all of population means are equal. Considering only the p-value of the pairs of means wich evaluations were made by human teachers, it is possible to see that all the means are less than the confidence level adopted (p-value < 0,05), except in the GA-EP means comparison. So, we can say that the means of the sample group (GA - EP) are sigificantly equivalent.

Group	Diff	Lower	Upper	P-Value
EP-CT	3,17	4,68	8,91	0,01
GA-CT	2,79	3,39	8,62	0,03
PS-CT	2,44	4,41	9,34	0,08
GA-EP	3,62	3,58	4,81	0,04
PS-EP	2,41	3,38	7,16	0,07
PS-GA	1,04	5,08	9,85	0,81

Table 3. Exercise B.

According to Table 3, it is possible to say that there is at least two evaluators with evaluations significantly diferente from CT, for the p-value is lower than the

confidence level. One can see that only the confidence interval for GA-EP is greater than 0,05. Thus, it appears that EP and GA do not differ among themselves, but are different from CT.

Group	Diff	Lower	Upper	P-Value
EP-CT	2,44	3,58	8,49	0,01
GA-CT	2,44	3,58	8,49	0,01
PS-CT	2,53	1,04	9,31	0,03
GA-EP	4,93	3,58	8,74	0,01
PS-EP	2,54	1,04	9,24	0,02
PS-GA	2,54	1,04	9,23	0,02

Table 4. Exercise C.

Analyzing the results of Table 4, it was possible to see that the p-value is lower than 0,05, which means that the ANOVA p-value for each evaluation is highly significant, indicating the difference between them. From this, we concluded that the average performance of the students are significantly different. Also in the table we see that the differences between the means of the teachers are statistically significant too.

Group	Diff	Lower	Upper	P-Value
EP-CT	0,86	0,98	9,89	0,08
GA-CT	0,92	6,05	10,00	0,09
PS-CT	0,00	6,76	9,67	0,33
GA-EP	0,85	4,05	9,74	0,34
PS-EP	0,86	4,98	9,83	0,33
PS-GA	0,09	4,72	10,00	0,09

Table 5. Exercise D.

In Table 5, comparing the evaluation scores, the average between the pairs of evaluations is greater than the confidence level of significance established in three groups. This leads us to conclude that the average performance of students (GA-CT), (PS-CT) and (PS-GA) are significantly equivalent.

Thus, it is concluded that the average performance of the students was significantly equivalent in tree of the analyzed evaluation, since the ANOVA was significant and the Tukey test demonstrated that all evaluations do not differ, with the exception of the comparison of exercise C. It demonstrates the feasibility of using the tool with a considerable degree of assertiveness. Some aspects were also observed on the results, such as: the evaluations were more homogeneous in exercise D. This similarity also increased when the grades were closer to 10,0.

6. Conclusions

CodeTeacher, a proposal of automated aid to the teaching of programming focused on the increase of teacher productivity was presented. In order to give a greater dynamism to the educational process through the systematic use of the tool, it is expected to reduce the time to correct the work, thus leaving the teacher free to dedicate himself to other teaching practices, such as elaboration of activities, planning of classes and preparation of didactic material, besides accompaniment of students.

A study case was conducted to evaluate the performance among students and teachers. The evaluation was done through the use of the tool to obtain the data related to its use. In order to identify if the evaluation method (automated or manual) factor exerts some influence on the students performance evaluation, a difference test among population means was applied for paired data from the same population. The results showed that, when compared to human evaluation, CodeTeacher achieved similar outcomes, since the difference was considered insignificant in most of the tests.

As future work, we intend to develop a web interface to deploy the application as an online service available on the web, as well as its integration with mobile platforms, and the development of complementary functionalities that can bring value to the tool, such as the ability to detect plagiarisms. There is also the possibility of attaching the tool to an VLE (Virtual Learning Environment) such as Moodle.

References

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102.

Benford, S. D., Burke, E. K., Foxley, E., and Higgins, C. A. (1995). The Ceilidh system for the automatic grading of students on programming courses. *In Proceedings of the 33rd annual on Southeast regional conference*, ACM-SE 33, pages 176–182, New York, NY, USA. ACM.

Berry, R. E. and Meekings, B. A. (1985). A style analysis of c programs. *Communications of the ACM*, 28(1):80–88.

Chaves, José Osvaldo M., Castro, Angélica F., Lima, Rommel W., Lima, Marcos Vinicius A., Ferreira , Karl H. A. (2013). MOJO: Uma Ferramenta de Auxílio à Elaboração, Submissão e Correção de Atividades em Disciplinas de Programação. In *XXI Workshop de Educação em Computação (WEI) - SBC 2013*, Maceió, AL.

Campos, C. and Ferreira, C. (2004). Boca: um sistema de apoio para competições de programação. In *XII Workshop de Educação em Computação (WEI) - SBC 2004*, Salvador, BA.

Curtis, B., Sheppard, S. B., and Milliman, P. (1979a). Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE ’79, pages 356–360, Piscataway, NJ, USA. IEEE Press.

Oliveira, M. G., Oliveira, E., Abordagens, Práticas e Desafios da Avaliação Automática de Exercícios de Programação. In: *4o. DesafIE - Workshop de Desafios da Computação Aplicada à Educação*, 2015, Recife, PE. Anais do 4o. DesafIE, 2015. p. 1-10.

Oliveira, M. G., Ciarelli, P. M., and Oliveira, E., (2013). Recommendation of programming activities by multi-label classification for a formative assessment of students. *Expert Systems with Applications*, 40(16):6641–6651.

Oliveira, M., Nogueira, Araújo, M., and Oliveira, E. (2015). Sistema de apoio à prática assistida de programação por execução em massa e análise de programas. In *CSBC 015-Workshop de Educação em Informática (WEI)*, Recife-PE.

De Souza, D., Maldonado, J., and Barbosa, E. (2011). Progtest: An environment for the submission and evaluation of programming assignments based on testing activities. In *Software Engineering Education and Training (CSEE T), 2011 24th IEEE-CS Conference on*, pages 1 –10.

Estey, A. and Coady, Y. (2016). Can interaction patterns with supplemental study tools predict outcomes in cs1? Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE ’16, pages 236–241.

Ebrahimi, A. (1994). Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies*, 41(4):457 – 480.

França, A., Soares, J., Gomes, D., and G.C.Barroso (2011). Um sistema orientado a serviços para suporte a atividades de laboratório em disciplinas de técnicas de programação com integração ao ambiente Moodle. *RENOTE - Revista Novas Tecnologias na Educação*, 9(1).

Hollingsworth, J. (1960). Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529.

Horstmann, Cay S; Cornell, Gary. Core Java 2. Vol.1: Fundamentos. Makron Books, 2000.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA. ACM.

Moodle – "A Free, Open Source Course Management System for Online Learning."(2011). Disponível em http://moodle.org/. Acesso em 26 ago. 2018.

Naude, K. A., Greyling, J. H., and Vogts, D. (2010). Marking student programs using graph similarity. Computers & Education, 54(2):545 – 561.

Neves, A., Oliveira, M., França, H., Lopes, M., Reblin, L., and Oliveira, E. (2017a). Pcodigo ii: O sistema de diagnóstico da aprendizagem de programação por métricas de software. In *Anais dos Workshops do Congresso Brasileiro de Informática na Educação*, volume 6, page 339.

Neves, A., Reblin, L., França, H., Lopes, M., Oliveira, M., and Oliveira, E. (2017b). Mapeamento automático de perfis de estudantes em métricas de software para análise de aprendizagem de programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 28, page 1337.

Otero, J., Junco, L., Suarez, R., Palacios, A., Couso, I., and Sanchez, L. (2016). Finding informative code metrics under uncertainty for predicting the pass rate of online courses. 373:42–56.

Peterson, A., Spacco, J., and Vihavainen, A. (2015). An exploration of error quotient in multiple contexts. Proceedings of the 15th Koli Calling Conference on Computing Education Research, pages 77–86.

Prior, J. C. "Online assessment of SQL query formulation skills". In Proceedings of the Fifth Australasian Conference on Computing Education. Adelaide, Australia. 2003.

Rahman, K. A., Ahmad, S., Nordin, M. J., and Maklumat, F. T. D. S. (2008). The Design of an Automated C Programming Assessment Using Pseudo-code Comparison Technique.

Romli, R., Sulaiman, S., and Zamli, K. (2010). Automatic programming assessment and test data generation a review on its approaches. In *Information Technology (ITSim), 2010 International Symposium in*, volume 3, pages 1186 –1192.

Rumsey, Deborah J. Statistics for Dummies. 2 ed. Chichester: John Wiley and Sons Ltd, 2016.

Tobar, C. M. et al. "Uma Arquitetura de Ambiente Colaborativo para o Aprendizado de Programação". XII Simpósio Brasileiro de Informática na Educação, Vitória, ES. 2001.