Teaching Software Quality via Source Code Inspection Tool

Pedro Henrique de Andrade Gomes, Rogério Eduardo Garcia, Gabriel Spadon,
Danilo Medeiros Eler, Celso Olivete Júnior and Ronaldo Celso Messias Correia

São Paulo State University (UNESP)
Faculty of Science and Technology – Presidente Prudente, São Paulo, Brazil,
Department of Mathematics and Computer Science – Presidente Prudente-SP, Brazil

pdrogomes@live.com, {rogerio, spadon, daniloeler, olivete, ronaldo}@fct.unesp.br

***Abstract*—Software Quality Assurance is a sub-process that ensures that developed software meets and complies with defined or standardized quality specifications. Focusing on source code, there are characteristics that can be used to evaluate the quality. Introductory courses must encourage freshmen students to improve internal quality of their source code, but only as sophomore they have contact with Software Engineering concepts, including Quality Assurance. In this paper we present a tool to source code quality evaluation aimed at supporting students to improve their source code and, consequently, their programming skills. The proposed tool uses quality reports (available to professional environment integrate with software repositories) to analyze students' source code and provide a feedback about the student coding. The proposed tool run locally, with few computational resources. In addition, we proposed the methodology to use the proposed tool: it consists of challenging students to perform a set of maintenance tasks in a controlled environment. We prepared a source code by introducing common defects, what decreases the quality of source code, and ask to students to perform maintenance tasks in order to both eliminate the introduced defects and introduce new features. After each modification, the students must evaluate their code using the proposed tool to obtain a feedback about quality of source code. To evaluate the approach and the tool, we created a survey and applied to students and the teacher. As a result, we show the benefits of using the proposed tool to both teachers and students perspectives. The results are positive to enhance the teaching-learning Software Quality Assurance to Software Engineering students.**

I. Introduction

Software Quality Assurance (SQA) is a sub-process focusing on developing software products that meets and complies with defined (or standardized) quality specifications. Researchers have dealt with SQA for a long time [1], [2], but usually focusing on developing method and tools. Similarly, there are several research on training students to apply adequately methods and tools. Despite the effort, *technical debt* concept was introduced by Ward Cunningham [3] representing issues related to architecture, structure, duplication, test coverage, comments and documentation, potential bugs, complexity, code smells, coding practices and style.

Focusing on source code, there are characteristics that can be used to evaluate the quality. SonarQube™ is an open-source code quality analysis platform to perform source code continuous inspection [4]. Its main goal is to analyze source code in order to identify violations and faults, caused by bad decisions (bad implementations) during the coding process [5]. The platform was developed as a set of open-source tools that have been used today in several projects. However, it is not suitable for educational purposes.

The increasing need for colleges and universities to produce graduates that are skilled in building good software products is a challenge. Introductory courses must encourage freshmen students to improve internal quality of their source code, but only as sophomore they have contact with Software Engineering concepts, including Quality Assurance.

In this paper we present a tool named SMaRT[1] aimed at source code quality evaluation, supporting students to improve their source code and, consequently, their programming skills. SMaRT uses Sonar Scanner as well as quality profile to analyze students' source code and provide a feedback about the student coding. The proposed tool run locally, with few computational resources. In addition, we proposed the methodology to use the proposed tool. To evaluate the approach and the tool, we created a survey and applied to students and the teacher. As a result, we show the benefits of using the proposed tool to both teachers and students perspectives. The results were positive to enhance the teaching-learning Software Quality Assurance to Software Engineering students.

To present SMaRT, its application using the methodology and results obtained, the remainder of this paper is organized as follows: SectionII presents the background about SonarQube technology as well as discuss its limitation for educational purposes; Section III presents the tools named SMaRT, followed by its application during a practical task and its evaluation (Section IV); Section V presents our final remarks and further works.

II. Background

A. SonarQube Platform

SonarQube™ is an open-source code quality analysis platform, developed by SonarSource S.A. to perform source code

[1]The tool is available at: http://www.fct.unesp.br/grupos/lapesa/smart/

978-1-5090-5920-1/17/$31.00 ©2017 IEEE

continuous inspection. Its main goal is to analyze source code in order to identify violations and faults, caused by bad decisions (bad implementations) during the coding process [6].

The platform was developed as a set of open-source tools that have been used in several projects, such as PMD (Program Mistake Detector), Findbugs, Checkstyle, and others [4].

SonarQube uses a set of rules to decide if a piece of code is (or is not) a bad implementation and, then, identify it as one. The analyzer, named *SonarQube Scanner*, look for snippets of source code that are not in agreement with the rules established for the project. The set of rules might be user defined, but there is a default set of rules (with more than 300 rules) [8].

A code snippet at odds with at least one rule is classified as an *issue*, which might compromise the quality of the software product in development. The same code snippet may violate multiple rules, in that case it will produce several issues. It's important to note that an *issue* is detected by violating rule and it is possible to choose which rules should (or not) be taken into account by the Scanner during the analysis process. The set of rules chosen to analyze a software project is named *Quality Profile*, and every project has only one *Quality Profile*. Those rules classify the violation into three different categories: Bugs, Vulnerabilities, and Code Smells [9].

Bugs are violations that have high probability of affecting the correct execution of a program. Any bug found should be resolved immediately. Vulnerabilities are violations that might compromise the security – and consequently the reliability – of the software under development. SonarQube detects as vulnerabilities the incorrect use of encryption features, the use of plain text database passwords as well as *IP addresses*, for example. Code Smells are all violations that undermine program maintenance, such as duplication of code, lack of documentation, lack of standards and conventions, and very high complexity [10], [11].

To ensure its effectiveness and adequacy to the most variety of software development projects, such as project design patterns, programming languages, development environments, the SonarQube™ platform is composed by four main distinct elements: (1) **SonarQube Server** -– a WEB Server, a Search Server and a Compute Engine, responsible to allow users to browse and analyze snapshots of project quality; (2) **SonarQube Database** -– to store statistics provided by SonarQube Server; (3) **SonarQube Plugins** — as an open platform, almost all the features are available by plug-in and API; (4) **SonarQube Scanner** — to analyze the source code and submit the results to SonarQube Server [7]. The interaction between main elements is depicted in Figure 1.

B. SQALE Method

The Software Quality Assessment based on Life-cycle Expectations (SQALE) is a method for evaluating the quality of software. The use of SQALE is recommended by its creators to: (1) support in a more objective, accurate, reproducible and automated way the evaluation of source code; (2) provide an efficient method for managing technical debt [12].

The SQALE method divides violations found into five categories, according to severity, as *Blocker*, *Critical*, *Major*, *Minor* and *Info*. The severity is related to how each violation is able to directly impact the quality. For each type of issue, a non-compliance factor is assigned, what helps us to calculate how severe the problems related to the project are [13]. The description of each category of severity, followed by examples and the non-compliance factor are presented in Table I.

Table I. SQALE Method of classifying violations based on severity.

Severity		Description	Example	Factor
	BLOCKER	Bug with a high probability of impacting program behavior	Memory errors	5000
	CRITICAL	Bug with a low probability of impacting security issue	SQL Injection	250
	MAJOR	Quality failure that can have a major impact productivity	Code duplication	50
	MINOR	Quality failure that can have a significant impact productivity	Long lines	15
	INFO	Neither a bug nor a quality problem	Just a hint	2

C. Software Maintenance: skills and challenges

The Maintenance Process is a part of the Software Development Life-Cycle (SDLC) that aims to support a modification on a software product after deployment. The maintenance is categorized into four different classes: (1) **Adaptive** – refers to modifications as a result of external influences or strategic changes within the company; (2) **Preventive** – to prevent breakdowns; (3) **Corrective** – aims to correct errors on software; (4) **Perfective** – to improve software features [14]. Regardless of the type of maintenance, it is important to create source code with quality.

During software development, bad decisions made by development team, leads to problems. In a nutshell, the effort required to fix a problem refers to cost and, consequently, the problem is named *technical debt*. SonarQube™tool is capable of tracking and measure the technical debt of a project during the development process [5], [15], [16].

The increase of the technical debt in a software project is a problem, shown by several authors [17]–[22]. As long-term perspective, if developers do not care about code quality during implementation and maintenance, quality problems will arise along with technical debt. After successive maintenance, the debt increases and the necessary effort to correct problems in a software project become so high that they make the maintenance task unfeasible [23].

It becomes important then to show to the student that quality problems in the project must be corrected as soon as possible. Moreover, it is interesting to show the student that the use of standards and good programming practices avoid the growth

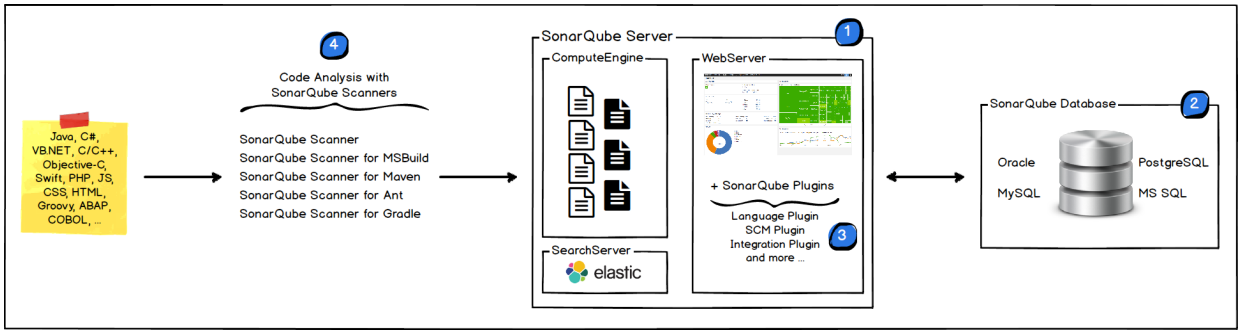

Fig. 1. A SonarQube platform and it's main elements. – SonarSource S.A, Switzerland. SonarQube™Docs. [7]

of technical debt in the project. When we started using the SonarQube™ platform as a teaching tool, we instructed our students to consider that if the code snippet had passed through *Quality Gates*, then maintenance is effective. The *Quality Gates* indicates a status of *approval* or *denial* obtained for the snippet of code submitted to the repository. The code analysis is performed using a set of established thresholds for the metrics present in the platform. Basically, if the code submitted reach thresholds, the interface indicates it is in compliance with minimal quality requirements, otherwise the submission fails considering the quality requirements [24].

D. Continuous Inspection

In a major software development project, analyzing the entire source code requires a high usage of computational resources, even if only a small change occurred. To avoid that, the SonarQube platform identifies changes in the source code and only analyzes the source that was locally modified. This feature is known as "*continuous inspection*" of source code.

Many studies [25]–[29] show that it is possible to perform a *Continuous Inspection* in the source code, allowing the continuous analysis of what is produced in order to increase the possibility of detecting possible problems during development.

Continuous Inspection aims to prevent the degradation of the source code after successive changes (Adaptive, Perfective, Corrective and Preventive maintenance) avoiding the introduction of new defects and allowing the development of software with quality.

E. Problem

As mentioned at Section II-A, *SonarQube Server* provides a set of tools that allow users to browse between *Quality Snapshots* of a project. An example of a quality snapshot that "passes" the SonarQube's defaults quality profiles can be seen in Figure 2. By comparing both snapshots (figures 2 and 3), as a teacher perspective, we realize that the quality of the student's projects varied considerably from one submission to the next.

Many factors contribute to compromise the quality of a project, such as tight deadline, fatigue, lack of clarity or understanding. After maintenance, by submitting the source code with more problems introduced than resolved, the student compromises the quality of the project and disrupts the evolution of the project.

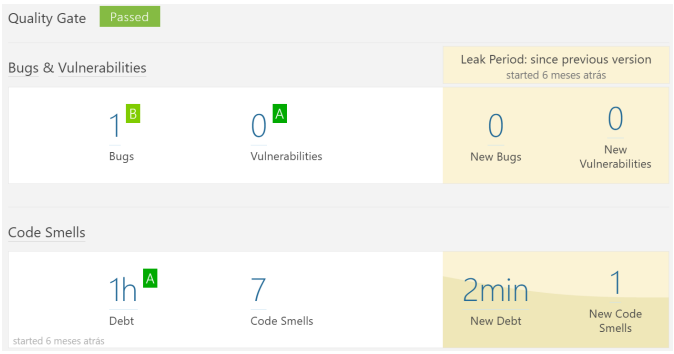

Fig. 2. Example of a SonarQube report for a **good** maintenance.

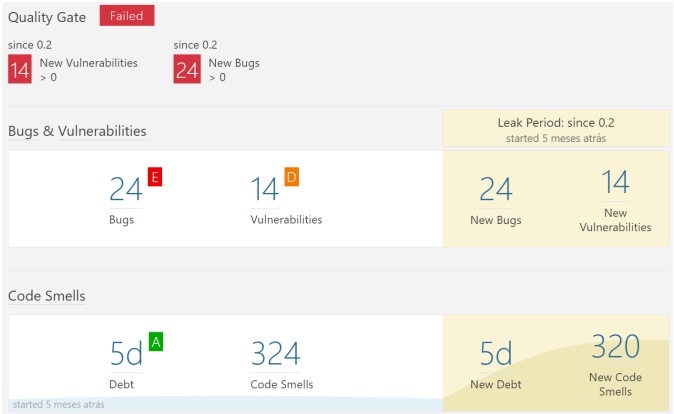

Fig. 3. Example of a SonarQube report of a **poor** maintenance. It is possible to see that the *Quality Gates* have changed their status from PASSED to FAILED.

By default, the *SonarQube Server* only shows the statistics after source code submission to repository. So, if the student does a bad maintenance, s/he will only realize that after submitting the code to the repository and impairing the quality of the project. It is possible to observe that in Figure 3.

As next maintenance, lets consider that the student perform a good job, solving many bugs and security issues, just adding

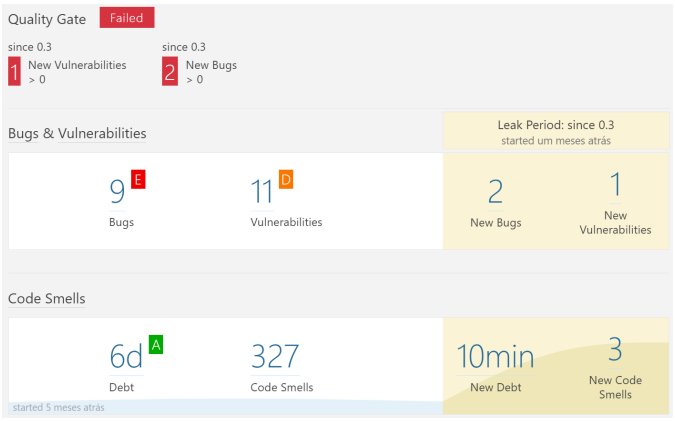

Fig. 4. *Quality Gates* indicates that maintenance failed, although there was a significant improvement in project quality. It is difficult to perceive the progress.

some minor issues quality problems. The problem in this case is, from student perspective, it is difficult to perceive the quality progress.

Both on Figure 3 and Figure 4, the submissions were disapproved by the quality analysis. On Figure 4, it is possible to see the new issue found by the analyzer, but nothing is demonstrated about what were solved. There were no changes in any of the category indicators, that is *Bugs* and *Vulnerabilities* just continues to indicate category E and D, respectively. Moreover, the *Quality Gate* indicator did not approved both submissions, but by comparing the snapshots, it is clear that the student did a really good job on the last submission, solving 17 bugs and 4 security issues.

Unfortunately, during the hard job, the student did some faults that introduces two bugs and a new vulnerability. That is why the *Quality Gates* rejects the second submission. In fact, a new bug, or a new security issue, is a serious problem, but we are convinced that it is important to consider not only wrong decision to analyze the project. Specially because we wonder to avoid the failure feeling – that what the student did was not good or was not enough.

Obviously, *SonarQube* was not proposed for educational purposes. So, we proposed a tool based on *SonarQube* to support teaching-learning process about of Software Quality focusing on maintenance tasks.

III. The Tool SMaRT

The tool SMaRT (Software Maintenance, Report and Tracker) was developed as an Eclipse plug-in aimed to check if modifications on source code made by students improve (or decrease) its quality. For that, SMaRT uses quality reports from *SonarQube Scanner*, in order to obtain measures about source code and, then, present them to students. We choose to develop SMaRT based on SonarQube and Eclipse IDE because both are open platforms that can be used both on professional and educational environment.

SMaRT tool verify if a given modification in the source code, that had been made to correct defects, was effective and how effective it was. There are three main goals: the first one is to show to students the importance of improving source code internal quality and how their decisions on coding may impact the source code quality; second, we intend to show how to avoid source code degradation, even with successive changes (corrective, adaptive and perfective maintenance); third, we intend to contribute to train students think about and act to software quality, especially by being able to avoid new defects (technical debts).

A. Architecture and Integration

We have developed a tool able to integrate and take advantage of the features provided by both the SonarQube platform and the Eclipse IDE. For that, we had to understand, follow and respect the specificity of each technologies involved in order to allow the coexistence of both with our tool.

In order to provide features within the development environment, we use Eclipse Plug-in Development Environment to create SMaRT as Eclipse IDE plug-in, which is widely used by students. By integrating *SonarQube Scanner*, using a set of configuration parameters that are usual part of SonarQube platform, we allowed the *SonarQube Scanner* communicate with the *SonarQube Server*. So, the analysis uses the *Quality Profiles* stipulated by the teacher. We allowed the teacher to establish rules (more or less rigid), according to the student's skills, for example.

As mentioned, it is only possible to obtain quality information from the project after it submission to the *SonarQube Server*. In order to analyze source code at local repository (avoiding submitting to the software project repository), SMaRT computes statistics similarly to *SonarQube Scanner*, but assessing local source code. After that, we compare the data locally generated by SMaRT and data obtained from *SonarQube Server* in order to compare source code analysis before and after modification.

By comparing the results, it is possible to to analyze the quality of maintenance in a local environment, knowing with precision each issue solved and each issue introduced.

If the student evaluates their maintenance as good enough to submit to the project repository, SMaRT allows to submit the statistics locally obtained to the *SonarQube Server*. Figure 5 shows how the architecture of our tool is and how it is integrated with Eclipse and SonarQube.

B. Metrics

Similar to SonarQube [30], our metrics and indicators are based on SQALE. In order to support the student in the evaluation of the quality of his/her maintenance aiming to improve software quality, we create three metrics capable of assisting the maintenance task. The first one, PG Index, is obtained from the ratio of the number of new issues by the number of resolved issues[2] during maintenance.

$$PG = \frac{issuesAdded}{issuesResolved}; \quad (1)$$

[2]If the developer has not resolved any issues, the number of new issues will be returned.

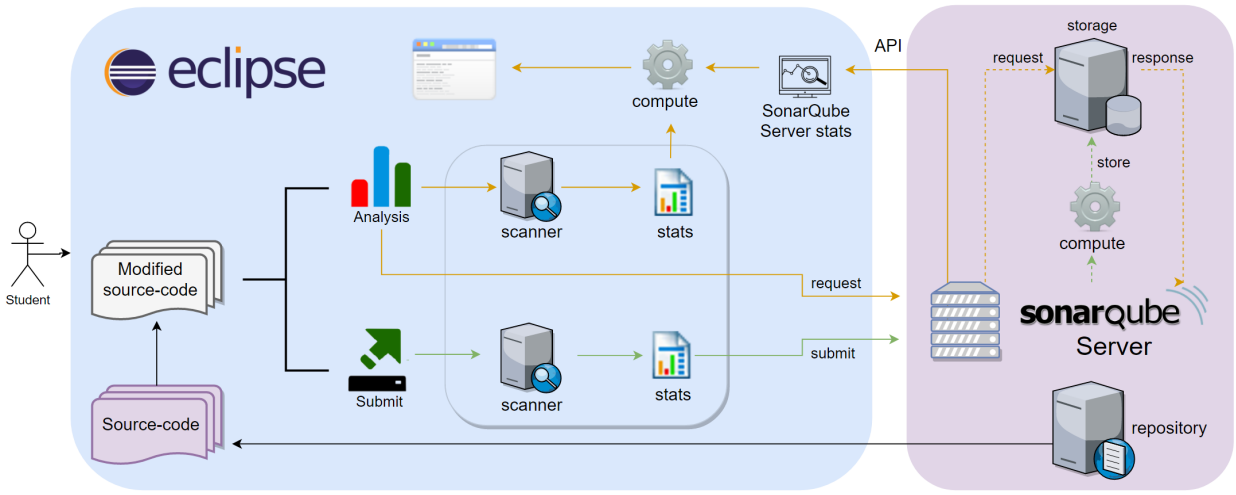

Fig. 5. A representation of the architecture of our tool and how it communicates with SonarQube™ platform and Ecplise IDE

Similarly, the Relative-Programmer Guidance (rPG) index is obtained from the ratio of the number of issues at local repository by the number of issues presented at SonarQube server[3]. This indicator helps to understand the evolution of the project after maintenance.

$$rPG = \frac{localIssues}{serverIssues}; \tag{2}$$

Also, we noticed that the two indicators presented were able to help the student to perceive the quantity of problems solved/introduced. Both indicators are efficient, but we realized that we should take into account what was actually corrected and what was introduced. We realized that it was important to analyze each of the violations committed and corrected by the developer. We also observed that it is necessary to take into account the impact of each violation on the project.

So, we create a third indicator: the Programmer Guidance-2 (PG-2) index is obtained from the ratio of the sum of the SQALE's Impact Factor[4] of new issues by the sum of the same factor on issues resolved[5].

$$PG2 = \frac{\sum_{i=1}^{A} w(A(i))}{\sum_{j=1}^{R} w(R(j))}; \text{ if } w = \begin{cases} BLOCKER & = 5000 \\ CRITICAL & = 250 \\ MAJOR & = 50 \\ MINOR & = 15 \\ INFO & = 2 \end{cases} \tag{3}$$

C. Rating

The calculated metrics are provided to students through an Eclipse IDE interface. In order to help the student to evaluate and to understand how the quality of his/her code is, we use the values obtained from the indicators to classify the source code quality using five categories, where *level A* represents a very good maintenance.

On the other hand, *level E* indicates many problems in the code after maintenance. In this case, the student should consider revising his/her source code before submitting it to the project repository (the goal is to ensure that bad coding does not affect software quality).

It is possible to observe in Table II the categorization of results obtained by the Programmer's Guidance indexes calculation. PG and PG2 indexes are based on SQALE indicators, but rPG index considers that, if the student solved 10% of problems, it's a very good maintain. On the other hand, if student introduces 10% more problems, the maintenance will be considered as very poor quality.

Table II. Categorization of the results obtained into 5 different classes, where A is a very good maintenance and E a very bad maintenance.

Class	PG	rPG	PG2
A	< 0.2	< 0.9	< 0.2
B	< 0.4	< 1.0	< 0.4
C	< 0.6	< 1.05	< 0.6
D	< 0.8	< 1.1	< 0.8
E	≥ 0.8	≥ 1.1	≥ 0.8

To exemplify the values provided by these category indicators, we will consider two situations: (1) during maintenance process, the student was capable to solve 6 issues, but introduces one new issue; (2) during another maintenance, the student solves two more issues, resulting in 8 solved issues, but with more faults too.

On the second case (see Table IV), the student perform more corrections, resolving 8 issues during their maintenance. Although the student solved more problems, he also committed 5 violations, against only one of the other example.

[3]If the **SonarQube server** has no reported issue (new project, for example), the number of local issues will be returned.

[4]The Impact Factor is defined on **SQALE** method

[5]If the developer has not resolved any issues, the number of new issues will be returned.

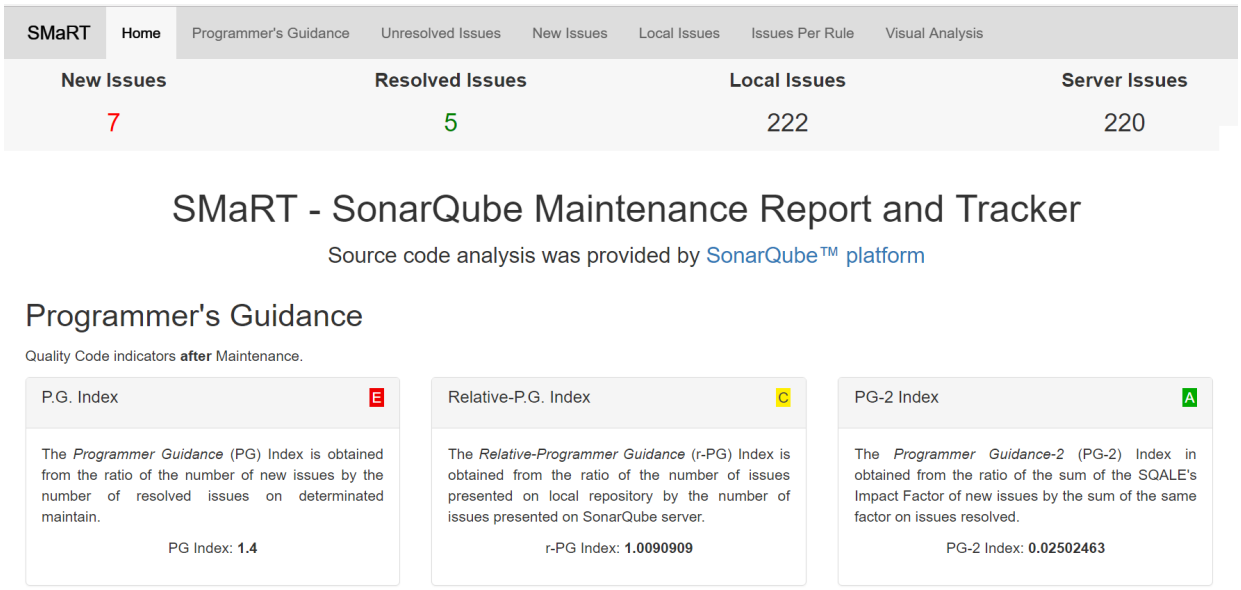

Fig. 6. A snapshot of main view of SMaRT tool report

Table III. Example I - 6 resolved issues and 1 added.

Indicator	Value	Class
PG-Index	0.1667	A
rPG-Index	0.9515	B
PG2-Index	0.3125	B

Table IV. Example II - 8 resolved issues and 5 added.

Indicator	Value	Class
PG-Index	0.625	D
rPG-Index	0.9709	B
PG2-Index	0.5907	C

By comparing Table III and Table IV it is possible to notice that PG-Index ranged from A to D, PG2-Index ranged from A to C. That is, quantitatively the number of issues solved may not indicate an improvement in the quality of the project, if the committed violations have a greater severity, even in a smaller quantity.

IV. Methodology and Evaluation

As mentioned previously, SonarQube platform is very useful and has been used in professional environments. But SonarQube platform is not suitable to educational purposes. So, it is difficult to implement a methodology capable of helping the students.

We aimed to ensure that the student would be able to perceive, in fact, how his/her decisions can affect the project quality, without necessarily to submit source code to repository. We performed a qualitative analysis using questionnaires to verify if the students could clearly perceive the quality of their maintenance.

In order to carry out the proposed tasks, each student should use a computer connected to the Internet, able to access our previously configured SonarQube server. Each computer had an Eclipse IDE installation as well as SMaRT plug-in and an already configured project.

After completing the task, we asked all volunteers to send us the modified source codes – we would be able to compare and analyze if the student's perception of the quality of his/her maintenance was according to what s/he indicated in the questionnaires.

During the analysis of data provided by the students, we did not take into account, for example, if there was an evolution in the quality of the maintenance. We were interested to ensure that our methodology was able to allow a better understanding by students.

A. The Pilot Study

We conducted a pilot study of a controlled experiment, which a group of students was randomly divided into two small groups. Due to the large number of concepts that involve the Software Engineering discipline, we chose to exclude students who had not been approved on that course.

Prior to the activities, a "Free, Prior and Informed Consent" (FPIC) form was provided to all participants. The volunteers were instructed to read the document and, in the case of non-acceptance, they would be excluded from the experiment.

Before carrying out the proposed activities, we gave a brief presentation of the tools and concepts involved with our project. SonarQube™ platform is integrated and presented throughout the "Software Development Life-Cycle" (SDLC). We demonstrated the main SonarSource tools – SonarQube Server, SonarQube Scanner and SonarLint. Finally, we demonstrated SMaRT tool and how it integrates into the SonarQube™ platform during the SDLC.

In order to ensure volunteers could understand the explanation and if they would be able to perform tasks using the presented tools, we proposed a training section task, which allowed the participants to test their skill by interacting with the development environment.

We prepared source codes by introducing common defects, normally caused by developer's mistakes. Both training and main activity source codes were commons "Object-Oriented Programming" (OOP) educational learning samples, like Banking and Library applications. At the main task, we provided the same source code to both groups. First group was able to use SonarLint plug-in, a SonarSource application to give some reports inside developer's "Integrated Development Environment" (IDE). On the other hand, second group was asked to use SMaRT tool.

Volunteers from each groups was challenged to individually perform maintenance task prior to solve problems like bugs, security fails and code snippets non-compliant, with coding good manners and language standardizations.

To evaluate the approach and the proposed tool, after the main task, we applied a survey to students. Our goal is to identify if the students were able to use SMaRT as well as they had a better understanding of how their maintenance impacted source code quality.

B. The Questionnaires

Before starting the proposed tasks, we applied a survey to evaluate the profile of the volunteers, in order to identify if the volunteer fit the requirements that we expected to participate in the research. The questionnaire had just simple questions to ensure that the student completed courses that provide knowledge needed to understand the experiment. The questionnaires used to analyze our results are presented below.

1) Activity Questionnaire: Questions 1 through 6 helped us to identify if the student was able to properly understand and operate SMaRT. The following questions focused our project, aiming to verify if the student was able to perceive and evaluate his/her maintenance results. The questions are presented on Table V.

Table V. Activity Questionnaire

Question
1) In how many classes did you make any kind of modification?
2) What was the HIGHEST degree of severity among the violations resolved?
3) What was the LOWEST degree of severity among the violations resolved?
4) How many violations do you believe to have resolved?
5) How many violations do you believe to have introduced?
6) Explain how you did to know the number of violations committed and corrected;
7) Soon after you started using the tool, could you assess the quality of the code?
8) Explain how you would assess the quality of the code at the start of maintenance;
9) On a scale of 1 to 5, how would you rate the quality of maintenance you performed?
10) Justify the note given regarding maintenance performed.

After all the activities, we applied a third and last survey which the students could evaluate our tool with respect to the ease of use, the reliability and we ask they to collaborate with ideas for future projects.

2) Final Questionnaire: All questions were open-ended questions. We ask that all participants in the research be as clear and direct as possible and explain to them the importance of sincere and accurate answers to our evaluation (Table VI).

Table VI. Final Questionnaire

Question
1) Was the training sufficient to explore and clarify the operation of the tools?
2) With what tool was it possible to identify the problems present in the code in a more simple way (SonarLint or SMaRT)?
3) What were the biggest challenges encountered when using the SMaRT tool?
4) Would you use the SMaRT tool in your projects?
5) What do you like about the SMaRT tool?
6) What do you dislike about the SMaRT tool?
7) What changes do you suggest to the SMaRT tool?

C. Results

As described on Section IV, we performed qualitative analysis of each participant answers. The motivation for conducting a qualitative analysis is strongly related to our goal: to support student to evaluate the quality of their maintenance.

Regarding the training performed, all the students reported that the training was enough to carry out the proposed activity. It's evidenced on students feedback, same as the following: "*the training was objective in what we should do and what tools we should use*". It was also evident that SMaRT presented a simple and user-friendly interface to the students, since all the volunteers informed found no usability difficulties.

All the students indicated that the task of identifying how many violations were committed/corrected after maintenance was simpler when using the SMaRT tool. We identified the facility from the answers given to Question 6 of the Activity Questionnaire. Answers like: "*I could observe the quality of my maintenance through the Quality Code indicator in the tool*" or "*I was able to analyze from the summary of the new issues*" showed that the students were able to understand the idea of the tool and mainly to correctly evaluate the quality of their code.

The biggest challenge pointed out by majority (75%) of students is related with the time of analysis. In fact, for an academic-level project, with something between 5 to 10 thousand lines of code (KLOCs), the analysis time ranged up to 40 seconds. The high time spent is due to the need of analyzing the source code, compute the statistics and still communicate with the *SonarQube Server*, depending on external factors such as the bandwidth of Internet connection.

The time spent is not detrimental to the progress of the project, at real use. But the time spent makes it unfeasible to use SMaRT as a substitute for the SonarLint tool (we do not intend that, and SMaRT was developed for educational

purpose): as answered, some students suggested a new version for professional purpose.

We were interested in whether the problems encountered would be enough to discourage students on using SMaRT. No students responded that they would not use our tool, but a small number of students answered they would use SMaRT only on academical projects.

Finally, according to the analysis of questions 9 and 10 of the Activity Questionnaire, we had evidence that the students' evaluation regarding the quality of their maintenance was similar to the values indicated by SMaRT, using the metrics proposed. Only one student could not rely on the results provided by SMaRT to evaluate the quality of his code.

V. Final Remarks

The present study makes several noteworthy contributions to demonstrate to the student the importance of Software Quality Assurance (SQA), a field of Software Engineering responsible for ensuring that software products are developed with quality. We developed a tool, named SMaRT, based on a well-known continuous inspection tool, aiming at supporting students to maintenance tasks – the analysis of source code allows the students to evaluate how the quality of the code present in their local repositories is, without compromises the project repository.

In addition, pilot study of a controlled experiment was conducted. We asked students to perform maintenance tasks on a source code prepared with addition of some common defects. From the applied questionnaires, we were able to have evidence that our proposed tool helps the students to understand what a source code with quality is.

A further study will assess the long-term effects of using the methodology focused on supporting the learner to evaluate the quality of her/his code and understand what actions s/he should, and should not, take to improve the project quality.

Acknowledgment

We are grateful to FAPESP (São Paulo Research Foundation) by the support. Our most sincere thanks to the students from Laboratory of Software Engineering and Applications (LaPESA) and the Department of Mathematics and Computer Science (DMC) at Faculty of Science and Technology (FCT) by the resources granted to this research.

References

[1] K. F. Fischer, "Software quality assurance tools: Recent experience and future requirements," *SIGSOFT Softw. Eng. Notes*, vol. 3, no. 5, pp. 116–121, Jan. 1978. [Online]. Available: http://doi.acm.org/10.1145/953579.811110

[2] B. A. Kitchenham, "Software quality assurance," *Microprocessors and Microsystems*, vol. 13, no. 6, pp. 373 – 381, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0141933189900458

[3] W. Cunningham, "he wycash portfolio management syste," 1992, oOPSLA '92 – Experience Report.

[4] SonarSource S.A., "SonarQube Java Analyzer: The Only Rule Engine You Need," 2016. [Online]. Available: http://www.sonarqube.org/sonarqube-java-analyzer-the-only-rule-engine-you-need/

[5] ——, "Technical Debt - SonarQube-5.2 - SonarQube," 2016. [Online]. Available: http://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt

[6] G. A. Campbell and P. Papapetrou, *SonarQube In action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2013.

[7] SonarSourse S.A., "Architecture and Integration - SonarQube Documentation - SonarQube," 2016. [Online]. Available: http://docs.sonarqube.org/display/SONAR/Architecture+and+Integration

[8] SonarSource S.A., "SonarAnalyzer for Java," 2016. [Online]. Available: http://dist.sonarsource.com/reports/coverage/rules_in_squid.html

[9] SonarSource S.A, "Metric Definitions - SonarQube Documentation - SonarQube." [Online]. Available: http://docs.sonarqube.org/display/SONAR/Metric+Definitions

[10] SonarSource S.A., "Bugs and Potential Bugs - Home - SonarQube," 2016. [Online]. Available: http://docs.sonarqube.org/display/HOME/Bugs+and+Potential+Bugs

[11] ——, "SonarQube – Bugs and Vulnerabilities are 1st Class Citizens in SonarQube Quality Model along with Code Smells," 2016. [Online]. Available: http://www.sonarqube.org/bugs-and-vulnerabilities-are-1st-class-citizens-in-sonarqube/

[12] J.-L. Letouzey, "The sqale method for evaluating technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*, ser. MTD '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 31–36. [Online]. Available: http://dl.acm.org/citation.cfm?id=2666036.2666042

[13] B. Hervé, "Software Qualimetry at Schneider Electric: a field background," in *Proc. of Embedded Real Time Software and Systems*, 2011.

[14] B. P. Lientz and E. B. Swanson, *Software maintenance management*. Reading, MA: Addison-Wesley, 1980. [Online]. Available: http://cds.cern.ch/record/101847

[15] SQuORE, "Techinal Debt - SQUORE," 2016.

[16] BlackBox, "Black Box Code - Technical Debt," 2016. [Online]. Available: http://www.blackboxcode.com/technical_debt.html

[17] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012*, pp. 91–100, 2012.

[18] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical Debt : From Metaphor to Theory and Pratice," *IEEE Software*, vol. 29, no. 6, pp. 18–22, 2012.

[19] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, no. 25-46, p. 44, 2011.

[20] E. Tom, A. Aurum, and R. Vidgen, "The Journal of Systems and Software An exploration of technical debt," *The Journal of Systems & Software*, vol. 86, pp. 1498–1516, 2013.

[21] A. Nugroho, J. Visser, and T. Kuipers, "An Empirical Model of Technical Debt and Interest," in *2nd International Workshop on Managing Technical Debt - Hawaii*, 2011.

[22] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," *Proceeding of the 2nd working on Managing technical debt - MTD '11*, p. 31, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1985362.1985370

[23] L. B. Foganholi, R. E. Garcia, D. M. Eler, R. C. M. Correia, and C. O. Junior, "Supporting technical debt cataloging with td-tracker tool," *Adv. Soft. Eng.*, vol. 2015, Jan. 2015.

[24] SonarSource S.A., "Quality Gates - SonarQube Documentation - SonarQube," 2016. [Online]. Available: http://docs.sonarqube.org/display/SONAR/Quality+Gates

[25] C. R. Prause and S. Augustin, "An Approach for Continuous Inspection of Source Code," in *WoSQ'08*. New York, New York, USA: ACM Press, 2008, pp. 17–22.

[26] M. Huo, J. Verner, L. Zhu, and M. A. Babar, "Software Quality and Agile Methods," *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, pp. 520–525, 2004.

[27] W. E. Lewis, D. Dobbs, and G. Veerapillai, *Software testing and continuous quality improvement*. CRC Press, 2009.

[28] P. M. Duvall, *Continuous Integration*. Pearson Education India, 2007.

[29] A. Miller, "A hundred days of continuous integration," *Proceedings - Agile 2008 Conference*, pp. 289–293, 2008.

[30] SonarSource S.A., "SonarQube – SQALE, the ultimate Quality Model to assess Technical Debt," 2016. [Online]. Available: http://www.sonarqube.org/sqale-the-ultimate-quality-model-to-assess-technical-debt/