On the use of static analysis to engage students with software quality improvement: An experience with PMD

Eman Abdullah AlOmar*, Salma Abdullah AlOmar, Mohamed Wiem Mkaouer†
*Software Engineering Department, Stevens Institute of Technology, Hoboken, NJ, USA
†Software Engineering Department, Rochester Institute of Technology, Rochester, NY, USA
ealomar@stevens.edu, alomar.salma@gmail.com, mwmvse@rit.edu

***Abstract*—Static analysis tools are frequently used to scan the source code and detect deviations from the project coding guidelines. Given their importance, linters are often introduced to classrooms to educate students on how to detect and potentially avoid these code anti-patterns. However, little is known about their effectiveness in raising students' awareness, given that these linters tend to generate a large number of false positives. To increase the awareness of potential coding issues that violate coding standards, in this paper, we aim to reflect on our experience with teaching the use of static analysis for the purpose of evaluating its effectiveness in helping students with respect to improving software quality. This paper discusses the results of an experiment in the classroom, over a period of 3 academic semesters, involving 65 submissions that carried out code review activity of 690 rules using PMD. The results of the quantitative and qualitative analysis show that the presence of a set of PMD quality issues influences the acceptance or rejection of the issues, design, and best practices-related categories that take longer time to be resolved, and students acknowledge the potential of using static analysis tools during code review. Through this experiment, code review can turn into a vital part of the educational computing plan. We envision our findings enabling educators to support students with code review strategies in order to raise students' awareness about static analysis tools and scaffold their coding skills.**

***Index Terms*—static analysis tool, education, quality**

I. Introduction

Linting is the process of using static analysis tools to scan the source code and detect coding patterns that are considered bad programming practices. These patterns can be responsible for future bugs and stylistic anomalies beyond compiler errors. Given their importance, linters have been introduced in classrooms to educate students on detecting and potentially avoiding these code anti-patterns [1]. However, little is known about their effectiveness in raising students' awareness with respect to anti-patterns, given that these linters tend to generate a large number of false positives [2]–[5].

In this paper, we reflect on the experience of using linters to support students with their task of debugging and improving the quality of existing systems. In particular, we require students to use PMD [1], a state-of-the-art static analysis tool, to detect potential issues in a software system that they did not implement themselves, and then, for each reported issue, they reason whether it should be corrected and suggest corrective action, in the form of a code change, depending on the type of issue reported. The pedagogical goals of this assignment are multiple: 1) Develop the skill of enhancing source code quality through static analysis. Students will be exposed to various bad programming practices that they need to reason on how to address them and suggest corresponding fixes. 2) Train students to review existing code, using the linter, reason over its warnings, and only propose a solution if they are convinced. It trains them to contextualize the problem within the code scope and document the decision of whether it has to be fixed. 3) Initiate students with reading and comprehending code that is not theirs. It prepares them for a more realistic industrial setting, where they will eventually be reading and updating existing code bases.

This paper contributes to the broader adoption of static analysis warnings by (i) designing a practical assignment for improving the quality of software systems, and (ii) reporting experience of using the PMD tool in a software quality assurance course that has been taken by 65 graduate students. As part of this paper's contributions, we provide the assignment description and the tool documentation for educators to adopt and extend [2].

The remainder of this paper is organized as follows: Section II reviews the existing studies related to automated static analysis tools. Section III outlines our experimental setup in terms of data analysis and research questions. Section IV discusses our findings, while the reflection is discussed in Section V. Section VI captures any threats to the validity of our work, before concluding with Section VII.

II. Related Work

Research on automated static analysis tools (ASAT) has been important to practitioners, researchers, and educators. The research community has spent considerable effort studying static analysis tools from different domains. This literature has included the usage of ASATs in the context of bug fixes [6], [12], [15], quality [7], security [8], [16], code review [9], [11], [17], defect classification and predication [10], [14], and technical debt [13]. However, except for [1], [20]–[22], most of the above studies focus on studying and improving the effectiveness of using ASAT for open-source communities, as opposed to our work that focuses on educating students on locating and fixing software defects. In this section, we

[1]https://pmd.sourceforge.io

[2]https://smilevo.github.io/self-affirmed-refactoring/

TABLE I: Related work in automated static analysis tool (ASAT).

Study	**Year**	**Context**	**Tool**	**Purpose**
Kim & Ernst [6]	2007	Bug fix	PMD/FindBugs/JLint	Study warning prioritization
Plosch *et al.* [7]	2008	Quality	PMD/FindBugs	Study relation between EQA and ASAT
Di Penta *et al.* [8]	2009	Security	Splint/Rats/Pixy	Observe evolution and decay of vulnerabilities
Panichella *et al.* [9]	2015	Code review	CheckStyle/PMD	Study if ASAT helps with code review
Beller *et al.* [10]	2016	Defect classification	CheckStyle/PMD/FindBugs/JSl Eslint/Jscs/Jshint/Pylint/Rubocop	Analyse state of ASAT
Singh *et al.* [11]	2017	Code review	PMD	Study ASAT helps reducing review efforts
Liu *et al.* [12]	2018	Bug fix	FindBugs	Mine fix patterns for FindBugs violations
Digkas *et al.* [13]	2018	Technical debt	SonarQube	Fix issues & pay back technical debt
Querel & Rigby [14]	2018	Bug prediction	FindBugs/JLint	Integrate statistical bug models with ASAT
Marcilio *et al.* [15]	2019	Bug fix	SonarQube	Study how developers use SonarQube
Aloraini *et al.* [16]	2019	Security	Rats/Flawfinder/Cppcheck PVS-Studio/Parasoft/Clang	Study warnings generated by ASAT
Trautsch *et al.* [17]	2020	Code review	PMD	Study the effect of PMD on quality
Romano *et al.* [18]	2022	Test-driven development	SonarLint	Study if ASAT affects software quality
Licorish & Wagner [19]	2022	Bug fix	PMD	Detect performance faults
This work		**Education**	**PMD**	Support students in enhancing quality with PMD

are only interested in research related to using ASAT. We summarize these approaches in Table I.

Kim and Ernst [6] investigated the possibility of leveraging the removal times for ASAT warning prioritization by utilizing commit histories of ASAT warnings. Later, Plosch *et al.* [7] correlated software quality metrics and defects with warnings found by various ASATs. The authors utilized three releases of the eclipse ecosystem and demonstrated correlations for various aspects, including size, complexity, and object-oriented software metrics.

In a security-related context, Di Penta *et al.* [8] performed an empirical study to extract the history of three open-source projects and analyze security-related ASAT warnings using three static code analyzers. In a similar context, Aloraini *et al.* [16] analyzed security-related ASAT warnings using 116 open-source projects. Both of these studies concluded that the warning density of security-related ASAT remains constant throughout their selected time span.

On the other hand, Beller *et al.* [10] empirically investigated the usage of ASAT in open-source projects by focusing on the prevalence of ASAT and the evolution of the configurations for different programming languages. In another study, Querel and Rigby [14] utilized FindBugs and Jlint for bug prediction. Their main finding revealed the information provided by the ASAT warnings could improve statistical bug prediction models. Liu *et al.* [12] explored ASAT warning over time by performing a large-scale study using the tool FindBugs via SonarQube. Their approach identified fix patterns that are then applied to unfixed warnings. In another study, Digkas *et al.* [13] utilized SonarQube to detect ASAT warnings and their removal strategies. The authors focused on technical debt and the resolution time assigned by SonarQube to each detected ASAT warning. Marcilio *et al.* [15] concentrated on developer usage of ASAT through SonarQube. They focused on the active engagement of developers when fixing different types of issues reported by ASAT. In a similar context, Licorish & Wagner [19] combined GIN and PMD for code improvements by focusing on detecting performance faults from Stack Overflow code snippets. Their findings show that static analysis techniques could be combined with program improvement methods to enhance publicly available code.

Some ASATs are used in the context of code review. Panichella *et al.* [9] studies if ASAT warnings are taken care of during the code review process. Their main finding indicated that the density of warnings slightly varies after each code review. Singh *et al.* [11] evaluated how ASAT can reduce code review effort. They investigated the overlap between reviewer comments on GitHub pull requests and warnings from the tool PMD. Their finding showed that PMD overlapped with around 16% of reviewer comments. Trautsch *et al.* [17] performed a longitudinal study of ASAT warning evolution and the effect of ASAT on quality. The authors analyzed the commit history of 54 projects, taking into account 193 PMD rules and 61 PMD releases. They found that significant global changes in ASAT warnings are mostly related to coding style changes. Another study relevant to our work is by Romano *et al.* [18]. The authors [18] studied the benefits of leveraging an ASAT on software quality in the context of test-driven development (TDD). Their study reveal that the use of a SonarLint helps the participants to improve software quality, although the participants found that TDD is more difficult to be performed.

To summarize, the study of static analysis tools has been extensively studied (*e.g.,* [18][12] [7]). Since we are focusing on Java, there are a few widely adopted Java-based open-source static analysis tools such as CheckStyle [3], FindBugs [4], JLint [5], PMD [6] and SonarQube [7]. The choice of PMD is motivated by different factors: its widespread use in the Java community and its maturity (*i.e.,* it is available since 2002 and therefore has been in use for a long time [17]), and works on the Java source code to find coding style problems, which is not the case by FindBugs and JLint that work on byte code and focus on finding programming errors and

[3]https://checkstyle.sourceforge.io
[4]http://findbugs.sourceforge.net
[5]http://jlint.sourceforge.net/
[6]https://pmd.sourceforge.io
[7]https://www.sonarqube.org/

neglecting programming style issues [7]. CheckStyle focuses more on readability problems compared to PMD, which tends to highlight suspicious situations in the source code [9]. Further, although there are recent studies that explored the use of static analysis tools on open-source communities focusing on how warnings evolve across the software evolution history [6], [8], [17], this is the first study investigating how PMD can support students in improving code quality, and reflecting on how students are using it in the classroom. To advance the understanding of the practice of learning how to find and fix bugs, in this paper, we performed a study in an educational setting using open-source projects having a large number of issues. This study complements the existing efforts that are done in open source systems [6]–[19] and in education [1], [20]–[22], by complementing the quantitative analysis with a qualitative one, providing evidence of several kinds of warnings students pay more attention to, during code analysis.

III. Study Design

A. Goal & Research Questions

We formulate the main goal of our study based on the *Goal Question Metric* template [23], as follows:

> ***Analyze*** *the use of an automated static analysis tool (ASAT)* ***for the purpose of*** *familiarizing students with improving source code, by developing the culture of reviewing unknown code and patching it with respect to software quality* ***from the point view*** *of educators* ***in the context of*** *Master's students in SE/CS who analyze Java-based software projects.*

According to our goal, we aim to answer the following research questions:

- **RQ1.** ***What problems are typically perceived by students as true positives versus false positives?***
  Motivation: This RQ aims at evaluating whether students can apply analytical skills to identify and fix issues in existing systems. The findings will shed some light on the feasibility of this learning activity to educate students to perform effective code reviews and quality control.
  Measurement: We examine (1) the types of issues that can be identified and (2) the PMD ruleset categories that are perceived by students as a true positive and false positive.
- **RQ2.** ***What category of problems typically takes longer to be fixed?***
  Motivation: This RQ investigates which PMD ruleset is taking longer time to be fixed, with respect to other rulesets. The finding raises educators' awareness of issues types that are hard for students to understand and address.
  Measurement: We examine the resolution time taken by students to fix each ruleset, clustered by category.
- **RQ3.** ***What is the perceived usefulness of PMD?***
  Motivation: This RQ explores the tool's feedback, and how students perceive PMD in general. The finding will inform educators on the usefulness, usability, and functionality of the tool, and allow them to make decisions about what ASATs can better support students with code improvement.
  Measurement: We examine students' feedback. We extract all their positive and negative comments as they describe their experience with using PMD.

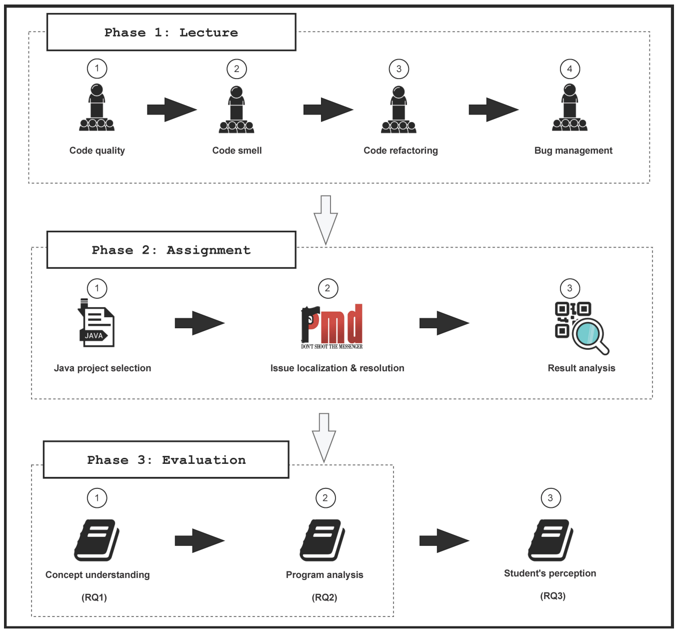

Fig. 1: Key phases of our study.

As part of this paper's contributions, we provide the assignment description, dataset, and tool documentation for educators to adopt and extend[8].

B. Course Overview

Software quality assurance is a graduate course, consisting of 2 lectures weekly, an hour and 15 minutes each. The course explores the foundations of software quality and software maintenance and introduces several challenges linked to various aspects of software evolution along with support tools to approach them. Also, the course covers various concepts related to software analysis and testing, along with practical tools, widely used as industry standards. Students were also given a number of hands-on assignments related to software quality metrics, code refactoring, bug reporting, unit and mutation testing, and technical debt management. The course deliverables consisted of five individual homework assignments, a research paper reading and presentation, and a long-term group project.

C. Teaching Context and Participants

The study involves one assignment in the software quality assurance course. The course was taught at Stevens Institute of Technology and Rochester Institute of Technology for a period of 3 semesters. Before conducting the assignment, students have already learned about several code, and design quality

[8]https://smilevo.github.io/self-affirmed-refactoring/

concerns: (1) code quality (teaching quality concepts and how to measure software quality), (2) code smells (teaching bad programming practices that violate design principles), (3) code refactoring (teaching refactoring recipes that help improving software quality), and (4) bug management (teaching software bugs and how to locate and fix them). The assignment constituted 7.5% of the final grade. It was due 14 days after the four corresponding sessions.

D. Assignment Content and Format

Initially, students are asked to analyze one version of a JAVA software of their choice approved by the instructor to ensure its eligibility based on popularity, besides making sure it correctly compiles, since PMD requires it. The rationale behind giving students the choice of project is to let them choose one that they are comfortable with and that fits into their interests. For students who do not want to search for a project, they have given a list that the instructor has already curated (see Table II). We selected these projects as we already know they contain a variety of software defects. Then, students are requested to set up and run PMD to analyze the chosen project production code. Students are also given the choice of running either the stand-alone version of the tool, or one of its plugins associated with popular IDEs (Eclipse, IntelliJ), as we want the students to be familiar with the coding environment, and reduce the setup overhead. Upon running PMD, students are required to choose a minimum of 10 warnings, and at least one from each category, if applicable. We enforce the diversification of warnings, to ensure a wider exposure to different types of potential issues, varying from design to multithreading, and documentation. It also increases students' learning curve as they cannot reuse their fix to address multiple instances of the same exact warning. Yet, we allow students to choose the instances they want to address. It implicitly makes students read many warnings, from all categories, which increases the likelihood of incidental learning. Furthermore, letting students choose the code fragments to review, increases their confidence in the decision they will make with respect to the given warning, *i.e.,* either consider it as a true positive, and provide a code fix, or consider it as a false positive, and provide a justification. In a nutshell, students followed these steps:

1) Install the PMD.
2) Run PMD on a project of students' choice and select 10 issues of different types.
3) Report the findings for each issue: (1) the type of issue, (2) whether it is a true or a false positive, (3) if it is a true positive, what are the necessary steps to fix it, (4) how long it took to check it / fix it, and (5) the code snippet.
4) Add to the report a concise comment about the experience with PMD (optional).

Submissions artifacts' evaluation was based on 1) assessment of students' ability to understand the type of issues (concept understanding (RQ1)); 2) assessment of whether students have provided acceptable fixes, or proper justification in both cases or accepting or rejecting PMD's recommendation (program analysis and evolution (RQ2)). Students' perception of the code was excluded from the evaluation process, as it can bias the experiment, as students would be filling out the survey arbitrarily, under the pressure of being graded. Also, providing feedback was anonymous and not mandatory, to increase the magnitude of PMD usage experience (RQ3). Although feedback was optional, many students have completed it (92.3%). Figure 1 depicts an overview of our experiment setup and execution.

E. Assignment Execution

The assignment was performed over three consecutive semesters. 65 students, primarily from computer science (CS) and software engineering (SE) majors, were enrolled during these semesters and completed the assignment.

F. Data Analysis

We analyzed the responses to the open-ended question to create a comprehensive high-level list of themes by adopting a thematic analysis approach based on guidelines provided by Cruzes *et al.* [24]. Thematic analysis is one of the most used methods in Software Engineering literature [25]–[27]. This is a technique for identifying and recording patterns (or "themes") within a collection of descriptive labels, which we call "codes". For each response, we proceeded with the analysis using the following steps: *i*) Initial reading of the survey responses; *ii*) Generating initial codes (*i.e.,* labels) for each response; *iii*) Translating codes into themes, sub-themes, and higher-order themes; *iv*) Reviewing the themes to find opportunities for merging; *v*) Defining and naming the final themes, and creating a model of higher-order themes and their underlying evidence. The above-mentioned steps were performed independently by two authors. One author performed the labeling of students' comments independently from the other author who was responsible for reviewing the currently drafted themes. By the end of each iteration, the authors met and refined the themes to reach a consensus. It is important to note that the approach is not a single-step process. As the codes were analyzed, some of the first cycle codes were subsumed by other codes, relabeled, or dropped altogether. As the two authors progressed in the translation to themes, there was some rearrangement, refinement, and reclassification of data into different or new codes. We used the thematic analysis technique to address RQ3.

IV. Results

A. What problems are typically perceived by students as true positives versus false positives?

In Tables III and IV, we illustrate PMD rules that are perceived by students as True Positive (TP) and False Positive (FP). It is worth noting the diversity of these warnings/violations, *i.e.,* they spread from warnings regarding style, code practice, and documentation, to warnings dealing with design and performance. Thus, upon analyzing students' assignment solutions, we cluster the issues according to the PMD

TABLE II: The list of open-source projects used in the assignment.

Project	# commits	# contributors	Domain
Ant	14,887	64	Java builder
GanttProject	4,361	38	Project management
Hutool	4,074	191	Code design
JCommander	1,009	64	Command line parsing
JFreeChart	4,218	24	Data visualization
JHotDraw	804	3	Data visualization
Log4J	12,211	137	Logging
Nutch	3,293	46	Web crawler
Rihno	4,119	80	Script builder
RxJava	6,004	289	Java VM
Xerces	6,463	5	XML parser

ruleset categories listed in the PMD official documentation [9], namely, 'Best Practices', 'Code Style', 'Design', 'Documentation', 'Error Prone', 'Multithreading', 'Performance', and 'Security'. These categories were captured at different levels of granularity (*e.g.,* package, class, method, and attributes). In the rest of this subsection, we provide a more in-depth analysis of these categories and the associated PMD rulesets.

Category #1: Best Practices. This category refers to the rules which enforce generally accepted best practices that are vital in an overall assessment of software quality. This category assists students in uncovering code that might violate main design and coding strategies, or indicate areas that might consider unnecessarily inefficient or difficult to maintain. Examples of the rules perceived by students as true positives include `LooseCoupling`, `SwitchStmtsShouldHaveDefault`, and `JUnitAssertionsShouldIncludeMessage`.

Category #2: Code Style. This category refers to the rules which enforce a specific coding style. Most prominently, brace and naming rules, consist of good coding practices regarding code blocks and naming conventions. This can be illustrated briefly by rules `IfStmtsMustUseBraces` and `ClassNamingConventions`, which shows that it should be followed by braces even if it is followed only by a single instruction and class names should be in camel case naming conventions to improve naming quality in the code and reflect the actual purpose of the parameters and variables.

Category #3: Design. This category refers to the rules which help discover design issues. Students captured design rules that contain best practices concerning overall code structure. By way of illustration, `AvoidDeeplyNestedIfStmts` rule indicates avoiding deeply nested if statements and `GodClass` shows the violation of single responsibility principles that increases the complexity of the code (*e.g.,* `CyclomaticComplexity`).

Category #4: Documentation. This category refers to the rules which are related to code documentation. Documentation is the description, in natural language, of the code-level changes and such description is crucial as it reveals the developer's rationale behind their coding decisions. `CommentRequired` rule is a good illustration of this group and shows that students seem to pay attention to the quality of the code comments.

Category #5: Error Prone. This category refers to the rules which detect constructs that are either broken, extremely confusing, or prone to run time errors. One of the key aspects to avoid these issues is readability. If the students refactor the code to be easily read and understood, there is less chance for misunderstandings and coding mistakes, and so students spend less time comprehending code. This is exemplified by error-prone-related rules such as `AvoidDuplicateLiterals` and `MissingStaticMethodInNonInstantiatableClass`.

Category #6: Multithreading. This category refers to the rules that flag issues when dealing with multiple threads of execution. For example, PMD recommends `AvoidUsingVolatile` to be avoided as the keyword 'volatile' is used to fine-tune a Java application and requires the expertise of the Java Memory Model.

Category #7: Performance. This category refers to the rules that flag sub-optimal code. Since performance plays a vital role, students are encouraged to follow best coding standard practices such as `AvoidArrayLoops` and `AvoidInstantiatingObjectsInLoops` rules to avoid degrading the performance of the code.

For our study, we analyzed a total of 690 rules of students' selected issues violating 155 distinct PMD rules. We found that 89 (57.41%) issues had 100% acceptance rate, while 9 (5.80%) issues had 100% rejection rate. The remaining had a mix of acceptance and rejection (36.79%). As can be seen from Tables III and IV, the majority of the violated rules are accepted and perceived by students as true positives, a few of these violated rules are rejected and perceived by students as false positives, and the remaining ones are distributed among both groups. While our results are not intended to be generalized, as it requires further experiments with larger sample sizes, our experience shows the success of PMD in triggering students' critical thinking about the problems outlined in the issues. Since students are given a choice to either accept or reject to address the issue, they can take the easy route of rejecting the majority of recommendations, as it is easier than accepting them and performing the necessary fixes. Fortunately, PMD has successfully attracted them to explore the various types of problems and provided sufficient documentation for them not only to comprehend it but also to code the needed fix for the problem. Moreover, it achieves another goal of our maintenance class, as it trains students to comprehend and act on code that they do not own.

Looking at the PMD rules that could play a role in acceptance and rejection, Figure 2 depicts the percentages of issues perceived by students as both TP and FP. As can be seen, the most common PMD ruleset category perceived as TP and FP concerns 'Code Style', representing 43.2% of the issues. This observation is in line with the findings of previous studies describing that most code reviewers look for style conformance when evaluating the quality of code [11], [28]. The next most common categories are 'Documentation', 'Best

[9]https://pmd.sourceforge.io

TABLE III: PMD rules that were perceived by students as true positive and false positive, broken down by category, as well as the percentages of responses for which students accepted/rejected the issues. Cells with 100% acceptance are highlighted in blue, while cells with 100% rejection are highlighted in red.

Category	Rule	Metric	Ratio
Best Practices	`AbstractClassWithoutAbstractMethod`	True Positive	100%
		False Positive	0%
	`ArrayIsStoredDirectly`	True Positive	80%
		False Positive	20%
	`AvoidGlobalModifier`	True Positive	100%
		False Positive	0
	`AvoidMessageDigestField`	True Positive	100%
		False Positive	0
	`AvoidPrintStackTrace`	True Positive	80%
		False Positive	20%
	`AvoidReassigningParameters`	True Positive	46.15%
		False Positive	53.84%
	`AvoidUsingHardCodedIP`	True Positive	100%
		False Positive	0
	`ForLoopCanBeForEach`	True Positive	100%
		False Positive	0
	`JUnit4TestShouldUseBeforeAnnotation`	True Positive	0
		False Positive	100%
	`JUnitAssertionsShouldIncludeMessage`	True Positive	100%
		False Positive	0
	`JUnitTestContainsTooManyAsserts`	True Positive	100%
		False Positive	0
	`JUnitTestsShouldIncludeAssert`	True Positive	0
		False Positive	100%
	`LiteralsFirstInComparisons`	True Positive	100%
		False Positive	0
	`LooseCoupling`	True Positive	100%
		False Positive	0
	`MethodReturnsInternalArray`	True Positive	50%
		False Positive	50%
	`MissingOverride`	True Positive	0
		False Positive	100%
	`OneDeclarationPerLine`	True Positive	60%
		False Positive	40%
	`PreserveStackTrace`	True Positive	80%
		False Positive	20%
	`ReplaceEnumerationWithIterator`	True Positive	100%
		False Positive	0
	`ReplaceHashtableWithMap`	True Positive	100%
		False Positive	0
	`ReplaceVectorWithList`	True Positive	50%
		False Positive	50%
	`SwitchStmtsShouldHaveDefault`	True Positive	75%
		False Positive	25%
	`SystemPrintln`	True Positive	50%
		False Positive	50%
	`UnusedAssignment`	True Positive	71.42%
		False Positive	28.57
	`UnusedFormalParameter`	True Positive	100%
		False Positive	0
	`UnusedImports`	True Positive	58.33%
		False Positive	41.66%
	`UnusedLocalVariable`	True Positive	90%
		False Positive	10%
	`UnusedPrivateField`	True Positive	100%
		False Positive	0
	`UnusedPrivateMethod`	True Positive	80%
		False Positive	20%
	`UseAssertNullInsteadOfAssertTrue`	True Positive	100%
		False Positive	0
	`UseTryWithResources`	True Positive	100%
		False Positive	0
Design	`AvoidCatchingGenericException`	True Positive	100%
		False Positive	0%
	`AvoidDeeplyNestedIfStmts`	True Positive	100%
		False Positive	0%
	`AvoidRethrowingException`	True Positive	100%
		False Positive	0%
	`AvoidThrowingNullPointerException`	True Positive	100%
		False Positive	0%
	`AvoidThrowingRawExceptionTypes`	True Positive	83.33%
		False Positive	16.66%
	`ClassWithOnlyPrivateConstructorsShouldBeFinal`	True Positive	50%
		False Positive	50%
	`CognitiveComplexity`	True Positive	100%
		False Positive	0%
	`CollapsibleIfStatements`	True Positive	100%
		False Positive	0%
	`CyclomaticComplexity`	True Positive	100%
		False Positive	0%
	`DataClass`	True Positive	100%
		False Positive	0%
	`DoNotExtendJavaLangError`	True Positive	100%
		False Positive	0%
	`ExcessiveImports`	True Positive	33.33%
		False Positive	66.66%
	`ExcessiveMethodLength`	True Positive	100%
		False Positive	0%
	`ExcessivePublicCount`	True Positive	100%
		False Positive	0%
	`FinalFieldCouldBeStatic`	True Positive	80%
		False Positive	20%
	`GodClass`	True Positive	100%
		False Positive	0%
	`ImmutableField`	True Positive	83.33%
		False Positive	16.66%
	`LawOfDemeter`	True Positive	83.33%
		False Positive	16.66%
	`NPathComplexity`	True Positive	100%
		False Positive	0%
	`SignatureDeclareThrowsException`	True Positive	100%
		False Positive	0%
	`SimplifyBooleanExpressions`	True Positive	100%
		False Positive	0%
	`SimplifyBooleanReturns`	True Positive	100%
		False Positive	0%
	`SingularField`	True Positive	100%
		False Positive	0%
	`TooManyFields`	True Positive	66.66%
		False Positive	33.33%
	`TooManyMethods`	True Positive	75%
		False Positive	25%
	`UselessOverridingMethod`	True Positive	80%
		False Positive	20%
	`UseObjectForClearerAPI`	True Positive	100%
		False Positive	0%
	`UseUtilityClass`	True Positive	100%
		False Positive	0%
Code Style	`AbstractNaming`	True Positive	100%
		False Positive	0%
	`AtLeastOneConstructor`	True Positive	70%
		False Positive	30%
	`AvoidFinalLocalVariable`	True Positive	100%
		False Positive	0%
	`AvoidPrefixingMethodParameters`	True Positive	0%
		False Positive	100%
	`BooleanGetMethodName`	True Positive	100%
		False Positive	0%
	`CallSuperInConstructor`	True Positive	100%
		False Positive	0%
	`ClassNamingConventions`	True Positive	66.66%
		False Positive	33.33%
	`CommentDefaultAccessModifier`	True Positive	100%
		False Positive	0%
	`ControlStatementBraces`	True Positive	100%
		False Positive	0%
	`DefaultPackage`	True Positive	50%
		False Positive	50%
	`DontImportJavaLang`	True Positive	100%
		False Positive	0%
	`EmptyMethodInAbstractClassShouldBeAbstract`	True Positive	60%
		False Positive	40%
	`FieldNamingConvention`	True Positive	0%
		False Positive	100%
	`FinalParameterInAbstractMethod`	True Positive	50%
		False Positive	50%
	`ForLoopsMustUseBraces`	True Positive	87.5%
		False Positive	12.5%
	`GenericsNaming`	True Positive	100%
		False Positive	0%
	`IfElseStmtsMustUseBraces`	True Positive	90%
		False Positive	10%
	`IfStmtsMustUseBraces`	True Positive	100%
		False Positive	0%
	`LocalVariableCouldBeFinal`	True Positive	66.66%
		False Positive	33.33%
	`LocalVariableNamingConventions`	True Positive	100%
		False Positive	0%
	`LongVariable`	True Positive	33.33%
		False Positive	66.66%
Error Prone	`AssignmentInOperand`	True Positive	50%
		False Positive	50%
	`AvoidFieldNameMatchingMethodName`	True Positive	66.66%
		False Positive	33.33%
	`AvoidLiteralsInIfCondition`	True Positive	80%
		False Positive	20%
	`CompareObjectsWithEquals`	True Positive	90.90%
		False Positive	9.09%
	`ConstructorCallsOverridableMethod`	True Positive	50%
		False Positive	50%
	`CloseResource`	True Positive	83.33%
		False Positive	16.66
	`EmptyWhileStmt`	True Positive	75%
		False Positive	25%
	`LoggerIsNotStaticFinal`	True Positive	50%
		False Positive	50%
	`SuspiciousEqualsMethodName`	True Positive	50%
		False Positive	50%
	`UnnecessaryCaseChange`	True Positive	75%
		False Positive	25%
	`EmptyCatchBlock`	True Positive	71.42%
		False Positive	28.57%
	`AvoidCatchingThrowable`	True Positive	100%
		False Positive	0%
	`AvoidInstanceofChecksInCatchClause`	True Positive	100%
		False Positive	0%
	`CloneThrowsCloneNotSupportedException`	True Positive	100%
		False Positive	0%
	`EqualsNull`	True Positive	100%
		False Positive	0%
	`EmptyFinallyBlock`	True Positive	100%
		False Positive	0%
	`EmptyIfStmt`	True Positive	100%
		False Positive	0%
	`MissingBreakInSwitch`	True Positive	100%
		False Positive	0%
	`MissingStaticMethodInNonInstantiatableClass`	True Positive	100%
		False Positive	0%
	`NonStaticInitializer`	True Positive	100%
		False Positive	0%
	`NullAssignment`	True Positive	100%
		False Positive	0%

TABLE IV: PMD rules that were perceived by students as true positive and false positive, broken down by category, as well as the percentages of responses for which students accepted/rejected the issues. Cells with 100% acceptance are highlighted in blue, while cells with 100% rejection are highlighted in red (Cont'd).

Category	Rule	Metric	Ratio
Code Style (Cont'd)	`MethodArgumentCouldBeFinal`	True Positive	81.81%
		False Positive	18.18%
	`MethodNamingConventions`	True Positive	60%
		False Positive	40%
	`OnlyOneReturn`	True Positive	60%
		False Positive	40%
	`ShortClassName`	True Positive	0%
		False Positive	100%
	`ShortMethodName`	True Positive	0%
		False Positive	100%
	`ShortVariable`	True Positive	80.95%
		False Positive	19.04%
	`SuspiciousConstantFieldName`	True Positive	100%
		False Positive	0%
	`TooManyStaticImport`	True Positive	100%
		False Positive	0%
	`UnnecessaryConstructor`	True Positive	100%
		False Positive	0%
	`UnnecessaryFullyQualifiedName`	True Positive	100%
		False Positive	0%
	`UnnecessaryImport`	True Positive	100%
		False Positive	0%
	`UnnecessaryLocalBeforeReturn`	True Positive	100%
		False Positive	0%
	`UnnecessaryModifier`	True Positive	80%
		False Positive	20%
	`UnnecessaryReturn`	True Positive	100%
		False Positive	0%
	`UseDiamondOperator`	True Positive	100%
		False Positive	0%
	`UselessParentheses`	True Positive	70%
		False Positive	30%
	`UseShortArrayInitializer`	True Positive	0%
		False Positive	100%
	`VariableNamingConventions`	True Positive	75%
		False Positive	25%
Documentation	`CommentRequired`	True Positive	90%
		False Positive	10%
	`CommentSize`	True Positive	27.77%
		False Positive	72.22%
	`UncommentedEmptyConstructor`	True Positive	100%
		False Positive	0%
	`UncommentedEmptyMethodBody`	True Positive	100%
		False Positive	0%
Multithreading	`AvoidUsingVolatile`	True Positive	50%
		False Positive	50%
Error Prone (Cont'd)	`ProperLogger`	True Positive	100%
		False Positive	0%
	`OverrideBothEqualsAndHashcode`	True Positive	100%
		False Positive	0%
	`ReturnEmptyArrayRatherThanNull`	True Positive	100%
		False Positive	0%
	`ReturnEmptyCollectionRatherThanNull`	True Positive	100%
		False Positive	0%
	`UnconditionalIfStatement`	True Positive	100%
		False Positive	0%
	`UseEqualsToCompareStrings`	True Positive	100%
		False Positive	0%
	`AvoidMultipleUnaryOperators`	True Positive	50%
		False Positive	50%
	`DataflowAnomalyAnalysis`	True Positive	0%
		False Positive	100%
Performance	`AddEmptyString`	True Positive	75%
		False Positive	25%
	`AvoidArrayLoops`	True Positive	75%
		False Positive	25%
	`AvoidFileStream`	True Positive	100%
		False Positive	0%
	`AvoidInstantiatingObjectsInLoops`	True Positive	83.33
		False Positive	16.66
	`AvoidUsingShortType`	True Positive	100%
		False Positive	0%
	`BooleanInstantiation`	True Positive	100%
		False Positive	0%
	`ConsecutiveAppendsShouldReuse`	True Positive	100%
		False Positive	0%
	`InefficientEmptyStringCheck`	True Positive	100%
		False Positive	0%
	`InefficientStringBuffering`	True Positive	100%
		False Positive	0%
	`InsufficientStringBufferDeclaration`	True Positive	100%
		False Positive	0%
	`IntegerInstantiation`	True Positive	100%
		False Positive	0%
	`LongInstantiation`	True Positive	100%
		False Positive	0%
	`OptimizableToArrayCall`	True Positive	100%
		False Positive	0%
	`RedundantFieldInitializer`	True Positive	100%
		False Positive	0%
	`SimplifyStartsWith`	True Positive	100%
		False Positive	0%
	`StringInstantiation`	True Positive	100%
		False Positive	0%
	`TooFewBranchesForASwitchStatement`	True Positive	100%
		False Positive	0%
	`UnnecessaryWrapperObjectCreation`	True Positive	100%
		False Positive	0%
	`UseArrayListInsteadOfVector`	True Positive	75%
		False Positive	25%
	`UseIndexofChar`	True Positive	100%
		False Positive	0%
	`UselessStringValueOf`	True Positive	100%
		False Positive	0%
	`UseStringBufferForStringAppends`	True Positive	100%
		False Positive	0%

Practices', and 'Error Prone', representing 15.8%, 14.2%, and 13.1% of the issues, respectively. This might indicate that students have different perspectives on whether developers follow the best practices, write descriptive enough code comments, or make code less susceptible to errors. The category 'Design', 'Performance', and 'Multithreading' had the least number of issues perceived as TP and FP, which had a ratio of 9.9%, 2.8%, and 1.1%, respectively.

Among the 690 analyzed issues, representing 155 distinct PMD rules. Nearly 75% of the rules had 100% acceptance rate, and only 6% had 100% rejection rate.

B. What category of problems typically takes longer to be fixed?

By looking at Figure 3, we found issues belonging to the 'Design' and 'Best Practices' categories (μ = 7.11 and μ = 3.20, respectively) take more time to be fixed, compared to the other categories. We speculate that these two categories take time to fix as students need to apply techniques to resolve quality flaws or design issues, including code smells (*e.g.,* `GodClass`), and quality attributes (*e.g.,* `CyclomaticComplexity`). This is exemplified in the assignments undertaken by students to resolve the issues. Students refactor the code and optimize the design by performing repackaging, *i.e.,* extracting packages and moving the classes between these packages, and merging packages that have classes strongly related to each other. The present observations are significant in at least two major respects: (1) improving the quality of packages structure when optimizing cohesion, coupling, and complexity and (2) avoiding increasing the size of the large packages and/or merging packages into larger ones which might have an impact on the following design rules, namely, `CognitiveComplexity`, `CyclomaticComplexity`, and `NPathComplexity`. Further, students might find fixing PMD issues violating best coding practices to be a challenge when trying to balance the trade-offs when maintaining efficient code with minimal errors. The other categories take less time to fix when comparing the mean ('Code Style' = 2.0, 'Documentation' = 1.63, 'Error Prone' = 2.79, 'Performance' = 2.90), which might possibly hint at an easy resolution of the reported issues or warnings.

The fact that design issues take significantly longer to be resolved, reflects a deeper challenge for the students. These antipatterns are symptoms of poor design or architectural decisions, requiring students to comprehend the anticipated design first, in order to scope the symptoms of the antipattern. Unlike other issues, this requires going beyond one or few instructions, into reasoning over methods and classes, and how they are architected. While there is empirical evidence of how these antipatterns significantly increase the code's proneness to errors [29], and hinder program comprehension [30], there is no consensus on how to detect [31] and correct them [32]. Yet, students are not trained to handle the subjective nature of the problem, and therefore, it can potentially cause longer reflection before reaching a refactoring decision.

Similarly to design patterns, being widely adopted in modeling classes [33]–[35], students should also be exposed to antipatterns. When design patterns are being taught, students are engaged in identifying key aspects of common design structure that make it reusable, while adhering to Object-Oriented design principles. Yet, existing large and complex systems are known to exhibit the existence of antipatterns [36]. Therefore, students shall be able to identify symptoms of bad design and programming practices, *i.e.,* **problem-based learning**. This learning paradigm leverages complex real-world problems (*e.g.,* antipatterns) to vehicle the learning of concepts (*e.g.,* design principles) [37]. One success criterion of this paradigm is the ability of problems in motivating students to propose multiple solutions for their resolution. Antipatterns challenge students' design thinking as they reason over their correction. Since there are multiple refactoring opportunities associated with each antipattern, students would need to justify their choices. This can be achieved by measuring design

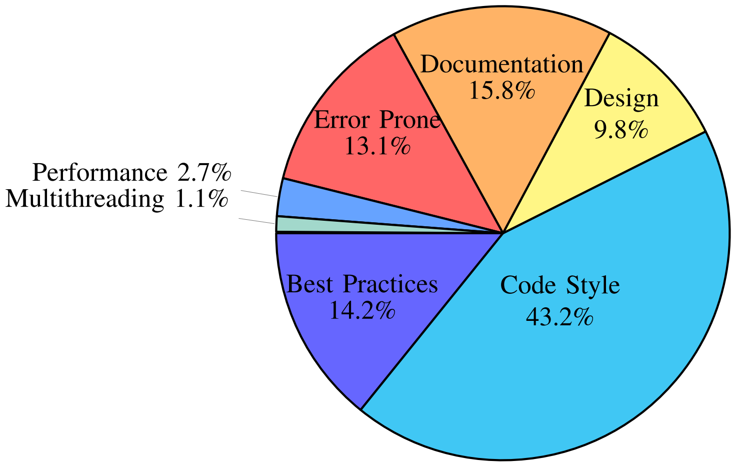

Fig. 2: Percentage of issues perceived as true positives and false positives, clustered by PMD categories.

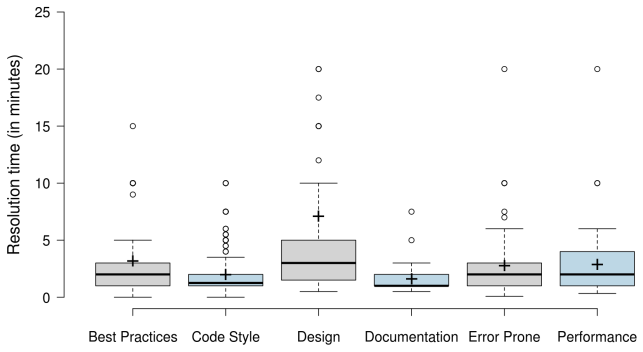

Fig. 3: Boxplots of time taken to fix issues, clustered by PMD ruleset categories.

quality, in terms of structural metrics, before and after the refactoring is implemented. Design evaluation is another aspect that students need to develop. Unlike error fixes, where students can systematically test their code for correctness, there is no trivial approach to validate the adequacy of a design change, without proactively measuring its impact on quality. The existence of multiple potential solutions, to address a given antipattern, can be leveraged by educators to establish a **cooperative learning** assignment. This paradigm allows students to compare their solutions through the assessment of refactorings' impact on antipattern resolution, along with its impact on software quality.

Design' and 'Best Practices' PMD ruleset categories take longer time to be resolved.

C. What is the perceived usefulness of PMD?

In Table V, we report the main thoughts, comments, and suggestions about the overall impression of the usefulness, usability, functionality, and recommendation of the tool, in ac-

TABLE V: Student's insight about the usefulness, usability and functionality of the tool.

Category	Sub-category	Example (Excerpts from a related student's comment)
Usefulness	Automation	"*The PMD code plugin allows us to automatically run the PMD code analysis tool on our project's source code and generate a site report with its results.*"
	Awareness	"*It generates several defects with Id, and description which will be helpful for developers to detect any bugs in source code.*"
	Debugging	"*The software is really powerful since it can find and categorize errors throughout the project very comprehensively and systematically in categories, making the debugging process efficient and effective.*"
	Efficiency	"*Bug detection and source code analysis are completed in a matter of seconds.*"
	Quality	"*it is great tool for keeping the code clean and maintainable.*"
Usability	Documentation	"*PMD plugin provides a very detailed explanation about each and every bug listed. It describes how it can be fixed along with an explanation why it was classified as an issue.*"
	Ease of use	"*The PMD is easy to use and covers a wider range code defects as compared to eclipse IDE. I have found it easy to use and the issue type are easy to understand.*"
	Configurability	"*I appreciated the custom violations overview window - it clearly gives the PMD plugin moer flexibility over what information is shown (and how the hierarchy is organized).*"
	Visualization	"*I liked the fact that it had its own view within Eclipse to make it really easy what I'm looking at and to organize all the different issues that were found. The labels and colors also made it simple to determine the severity of anything that was found by the program so I knew what I could look at and handle for this assignment reasonably.*"
Functionality	Design	"*PMD provide a better introduction to the common programming flaws. It also provides possibilities for recognize the complex problems which insist many flaws.*"
	Rules	"*I also appreciated the violations having very specific, detailed names presented in a clearly standardized fashion. I do like the violation categories (eg., Urgent, Important, Critical, Blockers, and Warnings), but the color scheme and order of priority significance is not immediately apparent in the current design.*"
	Warning severities	"*A good feature is that they have filter for severity filter for these violations which help to focus on the critical ones.*"
Recommendation	Automation	"*It is very simple and convenient to use. But there are still some shortcomings need manual analysis, but in general very good.*"
	Correctness	"*the reported bugs have two problems: First, the bugs sometimes repeat in reports,[...] maybe they are different at some detail in rules, but the performed result are identical. Another Problem is too much bugs about conventions in reported bugs. It's quite excellent to make standardized name for variables, but in actual developments, especially informatization project, the name of variables should conform to the meaning, not to the conventions. Both of these two problems could be fixed with rules option, but for real use PMD needs more improvement.*"
	Customization	"*PMD provides a clear list of issues per file, although there is no straightforward method to group the issues by categories across the entire project. It is relatively sound and works well for what it is.*"
	Resolution	"*Need to provide more fixes as to how developers can proceed in fixing the bug. Found that to be lacking using this tool.*"

cordance with the conducted labeling. The table also presents samples of the students' comments to illustrate their impressions of each theme.

Usefulness. Generally, the respondents found the tool to be useful in regard to five main aspects: automation, awareness, debugging, efficiency, and quality. A group of students commented that the tool's ability to automatically perform code analysis on a project's source code and generate a report containing information about bugs is useful. Nearly 75% of the students commented that PMD is intuitive to use and was efficient to locate software defects, and convenient for developers to use. Further, 6.42% of students commented that the tool's ability to find and categorize errors throughout the project makes the debugging process efficient and effective. A few students (12.84%) revealed that the PMD was fast in terms of analyzing huge and complex software, and identifying `GodClass`, and `DataClass` was one of the perks. 5.5% of students communicated that detecting the issues assists in increasing its *readability*, which helps improve overall code quality.

Usability. Based on the feedback provided by the students, the key areas in usability related to documentation, ease of use, configurability, and visualization. 70% of the students pointed out the tool is user-friendly and provides meaningful documentation with examples. Other comments also stated the tool's compatibility with various Java build tools, such as Maven, Ant, and Gradle, etc.

Functionality. According to the students' feedback about the tool's functional features, 60% of the students' comments appreciate the idea of the approach and are satisfied with various aspects of the tool's operation, and how this feature helps in better understanding of bad programming practices in real-world scenarios. Additionally, the students commented on their ability to practice a variety of strategies to remove issues. 20.18% of students mentioned that they liked the rule violation categories, indicated by the tool, and the detailed descriptions of these violations as they helped them decide whether or not the code should be fixed or if the detected issues were acceptable.

Recommendation. From the students' feedback, we have also extracted suggestions to improve the tool. 28.3% of the students' comments show a couple of suggested changes as a recommendation to be made to the tool's operation. We found out the students pointed out some of the recommendations related to automation, correctness, customization, and resolution. Students recommend the static analysis tool to be integrated with refactoring IDE instead of manually making those changes, as this feature will provide will allows developers to leverage built-in, safe refactoring features, instead of performing them manually. Other students felt that some issues are duplicates (*e.g.,* a student reported that `LiteralsFirstInComparisons` and `PositionLiteralsFirstInComparisons` are almost the same bug issues), and can be clustered, to reduce the number of suggested fixes. Students also recommended grouping the issues by categories across the entire project when outputting the analysis results, and generating informative messages and fixes (*e.g.,* enable access to refactoring tools to resolve static analysis warnings) since some of the flaws were observed with inexplicable fixes or suggestions. This observation is in line with previous studies [2], [4] finding that bad warning messages and no suggested fixes are one of the pain points reported by industrial software developers when using program analyzers.

Overall, the students were satisfied with the PMD and rated its various aspects positively.

V. Reflection

This section provides the lessons learned from both students' and educators' perspectives. Below, we first discuss our thoughts as to what went well, and what our plans are for Spring 2023. Then, we share the main reactions of the students.

A. Student Perspective

Lesson #1: ***Learning the best coding practices for software quality improvement.*** Running a static analysis tool as an assignment allows students to learn best coding practices to improve code quality and make the code less vulnerable to errors. For example, upon analyzing students' comments about the tool, students learn and use refactoring strategies for code smell resolution and quality improvement. Therefore, adopting the best practices and sharpening students' coding skills further improve development skills as a professional programmers, and support the characteristics of good code base health (*e.g.,* maintainability, readability, and understandability).

Lesson #2: ***Using the static analysis tool as a quick reference during the code review process.*** It is fundamentally vital to expose students to work on open-source projects to provide them with training to use the code that they did not write since upon their graduation, they are most likely to work on existing projects. That is why, this assignment challenges and trains them to perform code review, which is becoming a standard practice in the industry [5], [38]–[41]. For instance, Google [5] and Facebook [41] found that providing developers with an extensive list of warnings rarely motivates them to fix them, but reporting warnings during code review improves the adoption and removal of static analysis warnings. More specifically, students practice the review of different kinds of code changes (*i.e.,* functional and non-functional) as it helps a student to grow as a quality-aware developer. From a pedagogical perspective, code review is considered an active learning technique in SE education as it is a team-based activity and requires technical knowledge to review and analyze code [42], [43]. Our first and third research questions show how students are being challenged in code comprehending and fixing the issues. To cope with this challenge and develop students' critical and analytical skills, educators can consider applying **active learning** approach by following activities: (1) teach best practices for quality improvement, using metrics, and performing refactorings, (2) provide students with complementary tools for issues detection and correction, (3) instruct how to provide constructive criticism to others during code review, and (4) spread instructions of how to leave useful descriptive comments. Furthermore, it is necessary to teach the next generation of software engineering students the best practices for reviewing code that can result in higher quality code since, so far, these skills are generally acquired through experience or training.

B. Educator Perspective

Lesson #1: ***Creating custom PMD rules to enforce software engineering principles and good development practices.*** Using PMD was beneficial to students as it offers insights into various optimization possibilities and possible flaws in the code. According to the student's comments about the tool, students are interested in defining their own specific rules that would benefit their organization or future long-term class project. Thus, in the next course iteration, we plan to add scenarios where students are requested to design their own ruleset and use them to identify what they consider to be bad coding practices. Also, teachers can support students with crowd-sourced further PMD rules by mining repositories and detecting a range of faults in code provided on question-answering sites like Stack Overflow [19]. Further, since research in code smells mentioned that existing approaches can be subjectively perceived by developers [44]–[48], it is essential to translate that to students early enough so they learn how to customize static analysis tools, and know how to make their decision about their correction measure.

Lesson #2: ***Developing complementary assignments.*** Finding that reviewing design-related code changes takes longer than other changes reaffirms the necessity of integrating existing tools and techniques that can assist students in the code review process. For this to be successful and not troublesome to the students, the static analysis assignments can be augmented with refactoring recommendations (*e.g.,* JDEODORANT [10]) and software metrics (*e.g.,* UNDERSTAND [11]) to help students with creating a pipeline of detecting issues, correcting them, and measuring the impact of their change in code quality. Since one of our primary goals is to enhance students' problem-solving abilities, we rely on ASAT as a medium for **interactive learning**. When students analyze code, ASAT provides potential coding issues that violate coding standards. Therefore, students are being exposed to violations through examples, which facilitate their understanding. As students attempt their fix, they will interactively run the tool to verify the impact of their changes and close the feedback loop. Moreover, we noticed that poorly naming the code elements is one of the main bad naming conventions practices, typically caught by students when reviewing code changes. Integrating chapters about naming convention, in the class, would support students with refactoring bad names.

Lesson #3: ***Training students for real-world setting.*** Students are typically given assignments where specific guidelines are given about how their work should address the outlined problems. Our assignment spins off by providing students with an open-ended problem, where they are given the freedom to select issues, and the responsibility to properly address them. It trains students to approach existing systems, and

[10]https://github.com/tsantalis/JDeodorant

[11]https://scitools.com/

carefully choose their changes. Also, students learn how to justify their choices, either when accepting or denying a given issue. Furthermore, we observe from Figure 3 that 'Design' and 'Best Practices' take longer to be resolved due to the fact that resolving quality issues might require effective correction measures. To facilitate the resolution time, increase students' engagement, and improve the team's code review culture, we recommend educators implement **cooperative learning** strategy in the classroom in which students work in small groups to assist one another in learning the content. This can be achieved by applying the following tasks: (1) advocate for students to contribute to an open source project to fix issues as it has been shown that this helps with improving students' technical skills and self-confidence [49], (2) experience students with coming to a consensus during code review in cases opinions differ, and (3) engage students in early computing courses in the peer code review process.

VI. Threats to Validity

In this section, we describe potential threats to the validity of our research method, and the actions we took to mitigate them.

External Validity. Concerning the generalizability of our results, our study is limited to 65 submissions. Although we obtained valuable information and performed accurate analysis, the results may not represent the larger population of students that use static analysis tools. However, our participant pool is of a similar size (56) to the study that analyzed how industrial and open-source developers engaged with static analysis tools [3]. Further, our analysis was performed on mature open-source Java projects that varied in size, contributors, and number of commits. However, we cannot claim the generality of our observations to projects written in other programming languages or belonging to different ecosystems. Further investigation of even more projects is needed to mitigate this threat.

Internal and Construct Validity. As for the completeness and correctness of our interpretation of the open-ended comments about the tool, we did not extensively discuss all comments because some of them are open to various interpretations, and we need further follow-up interviews to clarify them. Additionally, to avoid personal bias during the manual analysis, each step in the manual analysis was conducted by two authors until reaching a consensus. The choice of PMD, as a static analysis tool, may introduce some bias to the way these issues are detected, especially since the detection of bad programming practices and code smells is known to be subjective [44]–[48]. Also, students may have had a different experience, if another tool was selected in this assignment. We chose PMD as it is one of the popular state-of-the-art tools, but in future work, we plan on trying other static analysis tools, to see if they can also reach this level of satisfaction.

Since students are choosing what to fix, they may skip fixing relevant warnings for other non-technical reasons (*e.g.,* late assignment submission). However, since the rejection ratio is low, we believe that students did their best to take the issues seriously.

VII. Conclusion

Understanding the practice of reviewing code to improve the quality is of paramount importance to education. Although modern code review is widely adopted in open-source and industrial projects, the relationship between the usage of ASATs such as PMD and how students perceive it during code analysis remains unexplored. In this study, we performed a quantitative and qualitative study to explore the effectiveness of PMD in familiarizing students with improving source code, by i) detecting code issues and antipatterns, and ii) implementing fixes for their correction. The paper develops the culture of reviewing and patching unknown code.

Our results reveal that several kinds of ASAT warnings that students pay more attention to during code review, reviewing design and best practices related changes take longer to be completed compared to other changes, and students rated various aspects of the tool positively, while also providing valuable ideas for future development. For future work, we plan on using other ASATs which will complement and validate our current study to provide the software engineering community with a more comprehensive view of the use of ASATs in order to engage students with software quality improvement from educator and student perspectives. Moreover, we plan to investigate students' understanding of code review practice using various real-world applications in a semester-long course project.

References

[1] S. A. Mengel and V. Yerramilli, "A case study of the static analysis of the quality of novice student programs," in *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pp. 78–82, 1999.

[2] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, IEEE, 2013.

[3] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020.

[4] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pp. 332–343, 2016.

[5] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[6] S. Kim and M. D. Ernst, "Which warnings should i fix first?," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 45–54, 2007.

[7] R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, and S. Schiffer, "On the relation between external software quality and static code analysis," in *2008 32nd Annual IEEE Software Engineering Workshop*, pp. 169–174, IEEE, 2008.

[8] M. Di Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Information and Software Technology*, vol. 51, no. 10, pp. 1469–1484, 2009.

[9] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 161–170, IEEE, 2015.

[10] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 470–481, IEEE, 2016.

[11] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 101–105, IEEE, 2017.

[12] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.

[13] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the apache ecosystem?," in *2018 IEEE 25th International Conference on software analysis, evolution and reengineering (SANER)*, pp. 153–163, IEEE, 2018.

[14] L.-P. Querel and P. C. Rigby, "Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 892–895, 2018.

[15] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 209–219, IEEE, 2019.

[16] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *Journal of Systems and Software*, vol. 158, p. 110427, 2019.

[17] A. Trautsch, S. Herbold, and J. Grabowski, "A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects," *Empirical Software Engineering*, vol. 25, no. 6, pp. 5137–5192, 2020.

[18] S. Romano, F. Zampetti, M. T. Baldassarre, M. Di Penta, and G. Scanniello, "Do static analysis tools affect software quality when using test-driven development?," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 80–91, 2022.

[19] S. A. Licorish and M. Wagner, "Combining gin and pmd for code improvements," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 790–793, 2022.

[20] S. Edwards, J. Spacco, and D. Hovemeyer, "Can industrial-strength static analysis be used to help students who are struggling to complete programming activities?," in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[21] R. Luukkainen, J. Kasurinen, U. Nikula, and V. Lenarduzzi, "Aspa: A static analyser to support learning and continuous feedback on programming courses. an empirical validation," 2022.

[22] A. Senger, S. H. Edwards, and M. Ellis, "Helping student programmers through industrial-strength static analysis: A replication study," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pp. 8–14, 2022.

[23] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal question metric (gqm) approach," *Encyclopedia of software engineering*, 2002.

[24] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*, pp. 275–284, IEEE, 2011.

[25] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 858–870, ACM, 2016.

[26] E. A. AlOmar, M. Chouchen, M. W. Mkaouer, and A. Ouni, "Code review practices for refactoring changes: An empirical study on openstack," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 1–13, 2022.

[27] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 348–357, IEEE, 2021.

[28] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 358–368, IEEE, 2015.

[29] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *2009 16th Working Conference on Reverse Engineering*, pp. 75–84, IEEE, 2009.

[30] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[31] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, pp. 1–28, 2017.

[32] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[33] O. Astrachan, G. Mitchener, G. Berry, and L. Cox, "Design patterns: An essential component of cs curricula," *SIGCSE Bull.*, vol. 30, p. 153–160, mar 1998.

[34] H. B. Christensen, "Frameworks: Putting design patterns into perspective," in *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pp. 142–145, 2004.

[35] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson, "An introduction to program comprehension for computer science educators," *Proceedings of the 2010 ITiCSE working group reports*, pp. 65–86, 2010.

[36] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 482–482, 2018.

[37] B. J. Duch, S. E. Groh, and D. E. Allen, *The power of problem-based learning: a practical" how to" for teaching undergraduate courses in any discipline*. Stylus Publishing, LLC., 2001.

[38] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International conference on software engineering*, pp. 712–721, 2013.

[39] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 181–190, 2018.

[40] C. Raibulet, F. A. Fontana, and I. Pigazzini, "Teaching software engineering tools to undergraduate students," in *Proceedings of the 2019 11th International Conference on Education Technology and Computers*, pp. 262–267, 2019.

[41] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Communications of the ACM*, vol. 62, no. 8, pp. 62–70, 2019.

[42] T. B. Hilburn, M. Towhidnejad, and S. Salamah, "Read before you write," in *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pp. 371–380, IEEE, 2011.

[43] C. Y. Chong, P. Thongtanunam, and C. Tantithamthavorn, "Assessing the students' understanding and their mistakes in code review checklists: an experience report of 1,791 code review checklist questions from 394 students," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 20–29, IEEE, 2021.

[44] S. Bryton, F. B. e Abreu, and M. Monteiro, "Reducing subjectivity in code smells detection: Experimenting with the long method," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pp. 337–342, IEEE, 2010.

[45] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, 2006.

[46] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?," in *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pp. 612–621, IEEE, 2018.

[47] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[48] F. A. Fontana, P. Braione, and M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment.,” *J. Object Technol.*, vol. 11, no. 2, pp. 5–1, 2012.

[49] G. Pinto, C. Ferreira, C. Souza, I. Steinmacher, and P. Meirelles, “Training software engineers using open-source software: the students’ perspective,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 147–157, IEEE, 2019.