Investigating Code Quality Tools in the Context of Software Engineering Education

DANILO SILVA,[1] INGRID NUNES,[2,3] RICARDO TERRA[4]

[1]*Universidade Federal de Minas Gerais, Belo Horizonte, Brazil*

[2]*Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil*

[3]*TU Dortmund, Dortmund, Germany*

[4]*Universidade Federal de Lavras, Lavras, Brazil*

Received 28 June 2016; accepted 21 December 2016

ABSTRACT: A key issue involved with software engineering education consists of how to guarantee that adequate software engineering principles are being followed at the code level, thus reinforcing that students produce high-quality code. Reviewing and grading student projects to verify whether they followed such principles is a time-consuming task, since this typically involves manual code inspection. In this paper, we exploit code quality tools and metrics to automatically assess student projects with respect to methods with many responsibilities (i.e., where the Extract Method refactoring should be applied), and evaluate their effectiveness. We conducted a study using two sets of student projects, developed in two academic semesters. Our results indicate that, to reduce the effort required to grade projects, two traditional code metrics, namely method lines of code and number of statements, perform best, and other metrics can be selected according to the system being implemented. © 2017 Wiley Periodicals, Inc. Comput Appl Eng Educ; View this article online at wileyonlinelibrary.com/journal/cae; DOI 10.1002/cae.21793

Keywords: software engineering education; object-oriented programming; code quality tools

INTRODUCTION

Teaching software engineering is a challenging task [1]. It is hard to motivate students to put techniques that are useful in medium and large-scale software into practice with small-scale examples, which is what we typically can handle in limited teaching time. In general, practicing techniques is an adequate approach for supporting the learning process. When they are adopted, it is usually in the form of projects to be developed by students. Such projects involve the elaboration of project instructions, and there are three main concerns associated with this task. First, we must identify adequate *problems and scenarios*, which demonstrate the need for software engineering approaches. Second, we must somehow guarantee that projects are *developed using the taught techniques*. Third, we must provide *adequate feedback for students*. As software engineering exercises often do not have a single correct answer, teachers typically perform manual inspection on produced models or source code to verify problems and give an appropriate grade to the work done.

To address these concerns, recently, software engineering courses tend to involve mainly practical activities [2], including games [3,4]. For instance, students may be required to develop software using up-to-date tools, such as Git for version control and automated unit tests using jUnit. Agile approaches, for example Scrum and Kanban, are also taught in this way [3]. Nevertheless, these approaches to teach software engineering tend to reinforce the need for *soft-skills*, for example, [5], and the dynamics in which software is developed, lacking the reinforcement of *code quality excellence*, with limited work on this topic [4]. Our focus in

Correspondence to I. Nunes (ingridnunes@inf.ufrgs.br).

this paper is this unexplored issue, targeting the problem of *grading* student projects and providing adequate feedback at the *code level*.

Over the past years, the software engineering research community has made significant effort to use software metrics in a systematic way or to develop approaches and tools to identify points in the source code where refactoring is potentially needed to improve code quality [6–9]. Examples are tools to analyse code quality, identify code smells [10,11] or check architectural conformance [12]. Not only such approaches and tools have been proposed, but have also been evaluated in the context of medium and large software projects. The targeted end-users of such tools are professional developers, so that they can be assisted in the task of design and coding to produce high quality software.

Software engineering students often produce design and code with compromised quality because of the lack of experience. They often learn abstract principles, for example, separation of concerns and modularity, and it is not trivial for them to put such principles into practice. It is thus a responsibility of teachers to analyse the design and code of students to point out problems in their design and code, and how to solve them. However, this is a challenge because checking many details of software projects of many students (or groups of students) is a time-consuming, repetitive task.

Therefore, given the code quality tools we have available, our goal is to investigate *whether they can help teachers identify design problems in code developed by software engineering students and support them in the grading process*. Although there is evidence that existing tools to support refactoring are effective, they were evaluated in restricted scenarios, and it is not guaranteed that they are effective in (very) small-scale software projects. Moreover, the coding style of an experienced developer is potentially different from that of a student that is learning software engineering principles for the first time. Consequently, this calls for studies that evaluate the effectiveness of code quality tools using software projects produced by students, so that they can be used by teachers to review and grade projects.

In this context, we present in this paper a study that evaluates the effectiveness of code quality tools to aid teachers of software engineering courses, in which principles such as modularity, information hiding, and polymorphism are taught. We focused specifically on the Java programming language, which is one of the programming languages to teach object-orientation [13] and is the number one programming language according to the TIOBE index.[1] Given that source code can present many problems, we investigate in particular the problem of a method having many responsibilities, and thus the *Extract Method* refactoring [14] should be applied. In Figure 1, we present a code sample in which *Extract Method* has been used. In this example, a new method named *checkInterfaceModifiers* was extracted to encapsulate part of the logic from the *visitToken* method, improving the separation of concerns and the readability of the original method. We selected this kind of code issue, because this is a recurrent problem we observe while teaching software engineering: students, who learn procedural programming before object-orientation, tend to create classes with methods that seem like procedures that implement required software functionality instead of splitting responsibility —actually this issue also occurs in the industry and thus it is important to ensure that students appropriately learn good programming practices.

Considering our target refactoring, we investigated two alternatives to identify it. First, we selected two tools that explicitly identify Extract Method opportunities, namely JDeodorant [11] and JExtract [10]. Second, we used an Eclipse plug-in to extract different code metrics. Although metrics do not explicitly identify our target refactoring, they are used to analyse code quality and thus can be used as evidence of problems in the code.

In short, our study consists of the following steps. First, an assignment was given to students, in which they had to design and implement a small software project according to a specification. Then, the teacher pointed out methods that have many responsibilities and should be split into two or more methods. We next collected data from projects with the selected tools and metrics, and matched them against the teacher output. Our results indicate that two traditional metrics perform best to support the grading process, and other metrics can be selected according to the system being implemented.

ANALYSED CODE QUALITY TOOLS

This section introduces the code quality tools we selected, which can potentially aid teachers to identify problems in the design and code of software engineering students. JDeodorant and JExtract sections present two tools that identify *Extract Method* refactoring opportunities, and Eclipse Metrics section presents a tool that calculates consolidated metrics, which we use for the same purpose. There are other tools that support refactoring, even IDEs such as Eclipse and IntelliJ. However, they were not selected because they are used to modify the code once refactoring opportunities are identified. Therefore, they do not support the identification process itself, which is the goal of our selected tools.

JDeodorant

JDeodorant suggests refactorings on Java systems, aiming to solve common code smells, such as Feature Envy, Long Method, and God Class [11,15,16]. Specifically, JDeodorant is able to suggest Extract Method refactoring opportunities in a target system and also apply them automatically. JDeodorant employs backward slicing to identify the slice of code that may affect a variable at a given point. This technique relies on the Program Dependence Graph (PDG)—nodes represent statements and edges represent dependencies—to represent the method under analysis. Therefore, backward slicing consists of selecting statements connected in the PDG, starting with a set of seeds. While traditional slicing algorithms consider the entire method body as a region where the slice may expand, JDeodorant adopts the concept of block-based slicing [17].

JExtract

JExtract identifies and ranks Extract Method refactoring opportunities, which are directly automated by IDE-based refactoring tools, based on the notion of similarity of structural dependencies [10]. Similarly to JDeodorant, JExtract's output is also Extract Method refactoring recommendations, which may be useful for identifying methods with many responsibilities. Additionally, the refactoring opportunities are ranked by a relevance metric.

Basically, the approach underlying JExtract consists of two main phases: *candidate generation* and *ranking* phases. In the

[1] http://www.tiobe.com/tiobe-index/

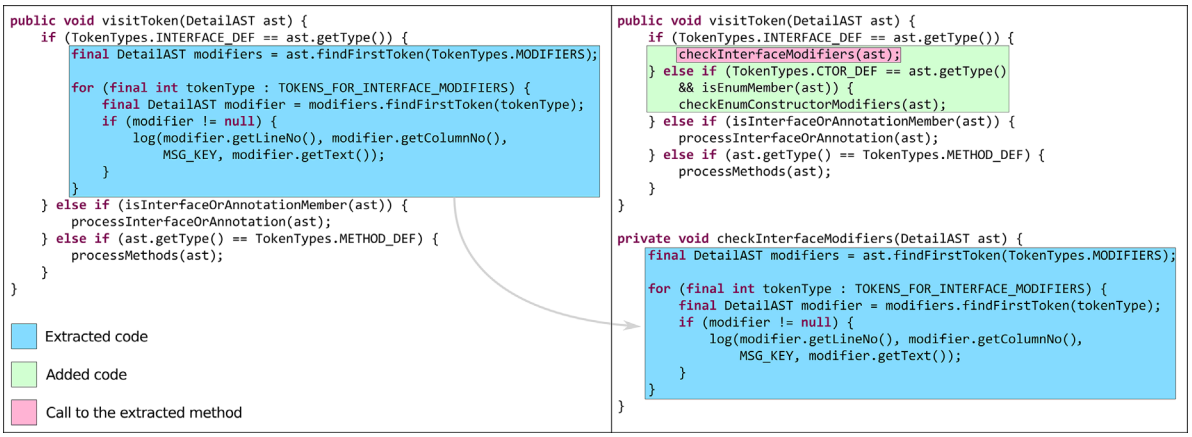


Figure 1 Extract Method Example.

candidates generation phase, it generates an exhaustive list of Extract Method candidates, that is, all possible series of sequential statements that follow a linear flow in the Control Flow Graph (CFG). In the ranking phase, JExtract shows only the most relevant candidates as Extract Method recommendations. By the exhaustive nature of the *candidate generation* algorithm, there are usually dozens of candidates for each method. Since users are usually interested in receiving just a few good recommendations, JExtract filters the list of candidates by setting the maximum recommendations per method and minimum score value thresholds.

Eclipse Metrics

Among the many existing metrics [18–21], this study contemplates the metrics provided by the Eclipse Metrics[2] tool. It is a plug-in for the Eclipse IDE that calculates several consolidated metrics for Java projects during build cycles and warns developers in case of range violations.

The plug-in was selected given that it is integrated to a development environment frequently adopted in academic environment and, as said, calculates many widely used metrics. Selected metrics can be directly associated with code problems, because they are related either to code complexity or size. For example, high number of statements in methods may indicate a long method that has many responsibilities (low cohesion), and high McCabe cyclomatic complexity may indicate that the code legibility is poor. In this study, we investigated all method-level metrics provided by the tool, which are detailed as follows.

- Number of locals (NumLocals): it refers to the total number of local variables declared in the scope of method. Formal parameters are not accounted for.
- MethodLines of Code (LOCm): it refers to the total number of lines of code of the method, including signature, empty, and comment lines.
- Feature Envy: it refers to the maximum difference of features the method uses from the given class by the ones the method uses from the current class. Stated differently, $FeatureEnvy = max_{c \neq cm}(|F_c| - |F_{cm}|)$, where m is the target method, F_c is the set of features m uses from a type c, and c_m is the class m is defined.
- Number of levels (NumLevels): it refers to the maximum number of levels of nesting in a method.
- Number of parameters (NumParams): it refers to the number of formal parameters of a method.
- Number of statements (NumStatements): it refers to the total number of statements of the method, which includes if, switch, for, do, while, explicit method/constructor calls, assignments, return, throw, try, catch, finally, break, and continue.
- McCabe cyclomatic complexity: it refers to the number of code segments with no branches in a method and hence can be used to determine the number of tests that are required to obtain complete coverage.

[2]http://sourceforge.net/pro jects/eclipse-metrics, v. 3.14.1

STUDY SETTINGS

In this section, we detail the study performed to evaluate the effectiveness of the previously described tools and metrics to identify problems in students' code, and thus be helpful to grade them. Although tools have been previously evaluated [10,11], projects used in experiments have not the size of student projects and were developed by experienced developers. Coding style evolves based on developer maturity and therefore previous results cannot be generalised to our context. We next present the goal of our study and the associated research question, and then describe the study procedure. Participants and developed projects are also presented.

Goal and Research Question

Inspecting software projects developed by students to assess whether they followed object-oriented principles after learning them is a time consuming task. In order to provide support to this task, our goal in this study is to evaluate how existing code quality tools that focus on supporting software development in the industry perform when target systems are very small and are developed by students that have just learned object-orientation. This goal is presented in Table 1, according to the GQM template [22]. Based on our goal, we derive a single research question: *what is the effectiveness of code quality tools to identify situations in which the Extract Method refactoring should be applied?*

Table 1 Goal Definition

Element	Our experiment goal
Motivation	To understand the benefits of using code quality tools in the context of Software Engineering Education,
Purpose	Evaluate
Object	Their effectiveness to identify programming issues in students' code
Perspective	From a perspective of the researcher
Scope	In the context of undergraduate student projects in a Computer Science course.

Procedure

Given that we explained our goal and research question, we now detail the steps we conduced, which comprise our study procedure.

Student Assignment. In an introductory software engineering course, in which students learn object-oriented programming, they were assigned to develop in groups a software system following an informal specification of the functionalities to be implemented. Students had to submit the assignment in a specified deadline, and received a grade for their work. This guarantees that they invested time and effort in this work.

Grading. An experienced teacher, responsible for the course, manually graded all student projects, that is, without the support of any tool. This involved a manual inspection of the code to point out problems, including methods that had more than one responsibility and part(s) of it that should be moved to other methods. Classes and methods were individually analysed.

Data Collection. After completing the grading step, we analysed the source code of all student projects with the three selected tools described in Jdeodorant section. As previously discussed, both JDeodorant and JExtract report Extract Method refactoring recommendations as their output, while Eclipse Metrics reports metric values computed for each method of the studied systems. Thus, to be able to compare them, we mapped the output of JDeodorant and JExtract to a metric that indicates how likely each method may have design problems.

For JDeodorant, we collected the number of recommendations given for a method as an indicator of potential design problems. The assumption behind this decision is that a method likely needs decomposition when there are Extract Method recommendations for it. We could not apply the same reasoning to JExtract because its recommendation heuristic relies on a maximum number of recommendations per method as input. However, JExtract computes a score for each recommendation, indicating its relevance. Thus, we configured the tool to suggest at the maximum one recommendation per method and collected the score of the best recommendation for each method as an indicator of potential design problems. We assigned zero as the score for methods with no recommendations.

Lastly, we also analysed the code using Eclipse Metrics. We computed all method-level metrics available in this tool, for all methods in all projects. In this case, there is no need for translating the output as before, because it consists of the quality metrics discussed in Eclipse Metrics section. By the end of the data collection step, we compiled them into a table with a total of 2,950 methods and their corresponding metrics.

Result Comparison and Analysis. The data collected in the previous step do not explicitly indicate whether methods should be refactored, but instead express, as a set of metrics, to what extent each method has more than one responsibility and, therefore, must be refactored. Consequently, to identify which methods should be refactored, a threshold must be set, either using a default value or choose one based on experimentation. In our case, we explore results obtained with a wide range of thresholds. Methods associated with a metric that has a value above the threshold must be refactored. That is the reason why we also analysed other code metrics, such as method lines of code and McCabe cyclomatic complexity—similar reasoning can be applied to them. Perhaps, a typical simple code metric can be useful in our scenario.

By specifying a threshold, we split methods into two groups (those above and below the threshold), and one of them corresponds to the set of methods that should be refactored according to the different tools and associated metrics. We also have a ground truth that is the evaluation made by the teacher, so we are able to calculate two widely used metrics to evaluate the performance of the tools: *precision* and *recall*. This allows us to discuss how many methods can be ignored during the grading process, without compromising recall. In addition, we also analyse the distribution of the values obtained for each metric, understanding their behaviour for methods that should be refactored and those that should not.

This completes the description of our study procedure. We next describe the participants involved in the study and the software projects students had to develop.

Participants

The first two steps of the procedure described above, that is, student assignment and marking, were performed in two different academic semesters, involving two different classes of the same course, which is part of a Computer Science undergraduate program. In this course, students learn for the first time software engineering concepts, focusing on the code level. Topics taught in this course include modularity, reuse, code conventions, and unit tests. In addition, object-orientation is introduced in this course, using Java as programming language. Before this course, students learn procedural programming in C.

Two sets of projects developed in the two academic semesters were evaluated. In the first semester of 2013 (2013/1), projects were developed by 12 groups of students, each composed of 3 members, except one that had 2 members. In the second semester of 2014 (2014/2), students formed 7 groups of 4 students each. The same teacher was responsible for these two classes, and graded all projects. This teacher has industrial experience in software development, and had previously taught software engineering courses. Note that participants had no knowledge about this study while developing projects, to guarantee that there was no change in their usual programming style (permission to use their code was obtained afterwards).

Software Projects

Here we describe the two software systems that students were requested to develop in each academic semester to practice the learned object-oriented concepts.

Bank Management System. In 2013/1, students were requested to implement a software system typically used to teach

object-oriented concepts, namely the *Bank Management System*. Students were instructed to develop a desktop system with two different interfaces: one for bank employees in branches and one for customers in ATMs. The system should simulate a database, so the system should start up with provided initial data, and data do not need to be persisted—the goal was to simplify student's effort, as persistence was not a concern of the course. Both interfaces should provide access to typical bank transactions, such as printing account balance, and make deposits and withdrawals. The transactions should be exactly the same through both interfaces, except that bank employees must inform the bank account in the system, while, in ATMs, this information is obtained when customers identify themselves in the system with an account number and password.

The key idea underlying this assignment is to make students model classes with their attributes and behaviour, such as an account class, and make system functionalities extensible—transactions should be implemented once as business operations independent from the user interface that invoke them. In addition, the user interface (text-based or graphic) should not be a god class, for example, implement a loop with a switch-case statement, each case implementing the logics of a specific functionality. This is what they were used to do when programming in procedural languages.

In 2014/2, students received a different assignment because the teacher responsible for the course noticed that students were not able to successfully implement the Bank Management System following good object-oriented and software engineering principles. They often created data classes and classes that implement the transactions, which caused the system to resemble a procedural system. Even though they seemed to understand modularity principles in theory, in practice, they tend to implement systems in the way they were used to, that is, with a procedural paradigm. In the next edition, the teacher followed an alternative approach. In the first practical assignment, students received an implementation of the Bank Management System made by an experienced developer and they were requested to evolve it by adding and modifying functionalities. After this experience, they were requested to develop a conference support system, described next.

Simple Conference Support System. The Simple Conference Support System provides basic functionality to support the management of conferences. As before, it is assumed that the system has a runtime database with existing data, that is, students did not need to develop functionality to put these data in the system. Students were requested to develop three main functionalities, all performed by an administrator. First, the system should be able to automatically allocate papers to be reviewed by committee members based on topics and conflicts—a simple algorithm was described to students. Second, a score, associated with a particular reviewer, can be given to a paper. Third, based on provided scores, the system should inform accepted and rejected papers, considering a given threshold.

In this project, students again must be able to appropriately design and implement a modular user interface. However, given that they had previously evolved a system that served as a concrete example of how it can be implemented (the example used a command pattern structure), it was assumed that they would follow the example. The key task in this system was then the design and implementation of the allocation algorithm, which should not be implemented in a single method, but split into different responsibilities assigned to different classes. Nevertheless, many groups of students did not successfully complete the task, using their old programming habits.

RESULTS AND ANALYSIS

Now we detail the results obtained after executing the steps of our procedure. We first present, in Table 2, the overall statistics of each studied project grouped by academic semester, detailing their number of packages, number of classes, number of methods, and total lines of code (LOC). Additionally, the last column (*Issues*) reports the number of methods to which the Extract Method refactoring should be applied, as marked by the teacher while reviewing and grading the student projects.

We can observe that the average number of lines of code is higher in the 2013/1 group (1,624.75) than that in the 2014/2 group (1,395.14). One may argue that this is due to the system size, considering that the Bank Management System seems to have more functionalities than the Simple Conference Support System. However, the average number of classes is higher in the 2014/2 group, causing the average number of lines of code per class to be lower—in the 2013/1 group the average of each project number of lines of code per class is 84.6 versus 58.1 in the 2014/2 group. In order to verify that this difference is statistically significant, we applied the one-tailed variant of the Mann-Whitney U test with the following null hypothesis: "the average number of lines of code per class in the 2013/1 group is less than or equal to that in the 2014/2 group." The null hypothesis was rejected with significance at 95% confidence level (P-value < 0.05), and thus we can conclude that the 2013/1 group has higher number of lines of code per class than the 2014/2 group. This analysis is relevant to understand the differences among metric values between the two groups. We believe that this significant difference may be due to the extra effort spent in the 2014/2 semester to reinforce modularity principles with students prior to the project

Table 2 Project Characteristics

Year	Project	Packages	Classes	Methods	LOC	Issues
2013/1	1	5	27	134	1,432	4
	2	17	44	405	3,161	1
	3	2	26	158	1,658	5
	4	5	14	83	894	6
	5	2	16	121	1,636	7
	6	6	28	154	1,377	2
	7	1	11	94	1,081	11
	8	4	19	112	1,606	1
	9	4	16	181	1,636	1
	10	3	18	199	2,524	12
	11	2	14	105	1,014	3
	12	2	13	71	1,478	14
	Average	4.42	20.50	151.42	1,624.75	5.58
	St. Dev.	4.25	9.35	88.87	639.12	4.56
2014/2	1	6	20	94	1,237	4
	2	6	27	125	929	1
	3	9	40	214	1,743	6
	4	8	24	177	1,444	0
	5	10	31	173	1,650	4
	6	7	26	177	1,708	7
	7	1	12	86	1,055	5
	Average	6.71	25.71	149.43	1,395.14	3.86
	St. Dev.	2.93	8.73	48.20	327.67	2.54

development (see Simple Conference Support System section). This is an indication that giving an example of modularised code before letting students develop a system from scratch seems to be a good approach.

We next present further data about our study, split into two parts. First, we discuss precision and recall in Precision and Recall section, and then present and analyse the distributions of the metric values in Distribution of Metric Values section.

Precision and Recall

As said in Result Comparison and Analysis section, the selected tools and metrics solely provide values, and do not point out which methods should be indeed refactored. This can be made by setting a threshold in the metrics to indicate whether the method should be refactored. Using this reasoning, we present precision vs. recall curves for each computed metric in Figure 2a,c,e. These plots should be interpreted as follows. Let R be the set of methods that should be refactored (as marked by the teacher) and M be the set of methods with a metric that is above a certain threshold t. A point in the plot is a tuple (Recall, Precision), such that Precision $= |M \cap R|/|R|$, and Recall $= |M \cap R|/|M|$.

By moving the value of the threshold t, we plot a precision versus recall curve for a metric. Each line in the plot represents one of the studied metrics. Usually, precision versus recall curves start at high precision (y-axis) and falls down when we move right in the x-axis. Ideally, a perfect metric would yield a straight line at the 1.0 precision. In Figure 2a, we present the overall comparison between all studied metrics.

We can observe that the NumStatements and LOCm curves are very similar and both dominate all other curves. Therefore, simple size metrics showed to be better indicators of methods that need refactoring than all other metrics in this experiment setup. The behaviour of the other metrics varies depending on the desired recall level. For example, if we focus on the 0.4 recall mark, JDeodorant, and McCabe are the third and fourth most precise indicators respectively. On the other hand, if we focus on the 0.8 recall mark, McCabe, and JExtract assume the third and fourth places.

In Figure 2c,e we present the same comparison, but with different plots for the 2013/1 and 2014/2 groups, aiming to investigate whether the different characteristics between the groups of projects influence the precision of each approach. By comparing both plots, we observe that NumStatements and LOCm are still dominant. Specifically in 2014/2, they completely dominate all other curves. However, other curves are much closer to both of them in the 2013/1 group. In fact, JDeodorant surpasses them at 0.5 recall. Another interesting observation is that JDeodorant and NumLevels achieve significantly better results in 2013/1, while JExtract achieves better results in 2014/2. We believe that this may be due to the implemented system itself. In the Bank Management System, many students implemented the user interface as a god class, and different variables were used in different parts of a long method, while in the Simple Conference Management System the core issue was the algorithm to distribute the papers, which has separate steps. The latter is a case that the JExtract captures well, so in this case it tends to perform better. Therefore, NumStatements and LOCm can be used as a rule-of-thumb, and other metrics can be selected according to the selected teaching strategies. Yet, this requires the teacher to know in advance the best metric according to the characteristics of the system being implemented by students.

To complement the precision vs. recall analysis, we also present recall vs. rank position plots in Figure 2b,d,f. These plots present what proportion of a ranked list of all methods one should inspect to achieve a certain recall level, for all studied metrics. In Figure 2b we present the overall recall vs. rank position comparison. We can observe that, when ranking by NumStatements or LOCm, more than 80% of the methods with problems are in the top 10% of the ranked list. Moreover, both NumStatements and LOCm achieve 1.0 recall at the 40% mark.

In Figure 2d,f we also present the recall vs. rank position comparison, but with different plots for the 2013/1 and 2014/2 groups. The differences between these plots are consistent with those observed in the precision vs. recall analysis. NumStatements and LOCm are the best metrics, but this is accentuated in 2014/2. Moreover, JDeodorant and NumLevels achieve significantly better results in 2013/1, while JExtract achieves better results in 2014/2.

Distribution of Metric Values

After analysing precision and recall, we investigated the distribution of the values of each metric selected in our study in order to have a better understanding of the metrics and possible thresholds to be set. Specifically, we investigated whether the distribution of values is different when we divide our set in two partitions: (1) the group of methods with no issues pointed out by the teacher, which we denote by *ok*, and (2) the group of methods with issues marked by the teacher, which we denote by *refactor*. To compare the distribution of values of these two groups, Figure 3a–i show violin plots for each studied metric. Violin plots show the median and the distribution of data in quartiles, similarly to box plots, along with a probability density estimation of the data at different values. As a general sense, we expect that metrics such as lines of code, McCabe's cyclomatic complexity, and so on, tend to be higher in the methods from the refactor group, that is, a method that should be refactored is likely to be longer and more complex than one that should not. For example, in Figure 3b we can observe that the median (represented by the white dot) is close to 50 in the refactor group, which means that about half of the methods in this group have 50 lines of code or more. On the other hand, the median is about 4 lines of code in the ok group. We can visually confirm that the median of the refactor group is higher for all metrics in Figure 3a,b,d–f,h,i. The only exceptions are Figure 3c,g which correspond to NumLocals and NumParams. The distribution of values for these metrics is slightly different between groups, but the median is the same.

We also applied the one-tailed variant of the Mann-Whitney U test to find if the difference in the expected value of each metric is statistically significant between groups. In this case, we tested for the following null hypothesis: "the measures in the refactor group are less than or equal to the measures in the ok group." The null hypothesis was rejected with significance at 95% confidence level (P-value < 0.05) for all metrics, and thus we can conclude that the expected value for all metrics is really higher in the refactor group. This result indicates the using metrics, either traditional or derived from the JDeodorant or JExtract tools, makes sense, because otherwise, that is, if there was no difference between the metrics of these two groups, they could not be used to discriminate methods with issues.

Although the distribution of values is different, we can observe that there is always an overlap between the distributions of ok and refactor groups. For example, if we inspect Figure 3d

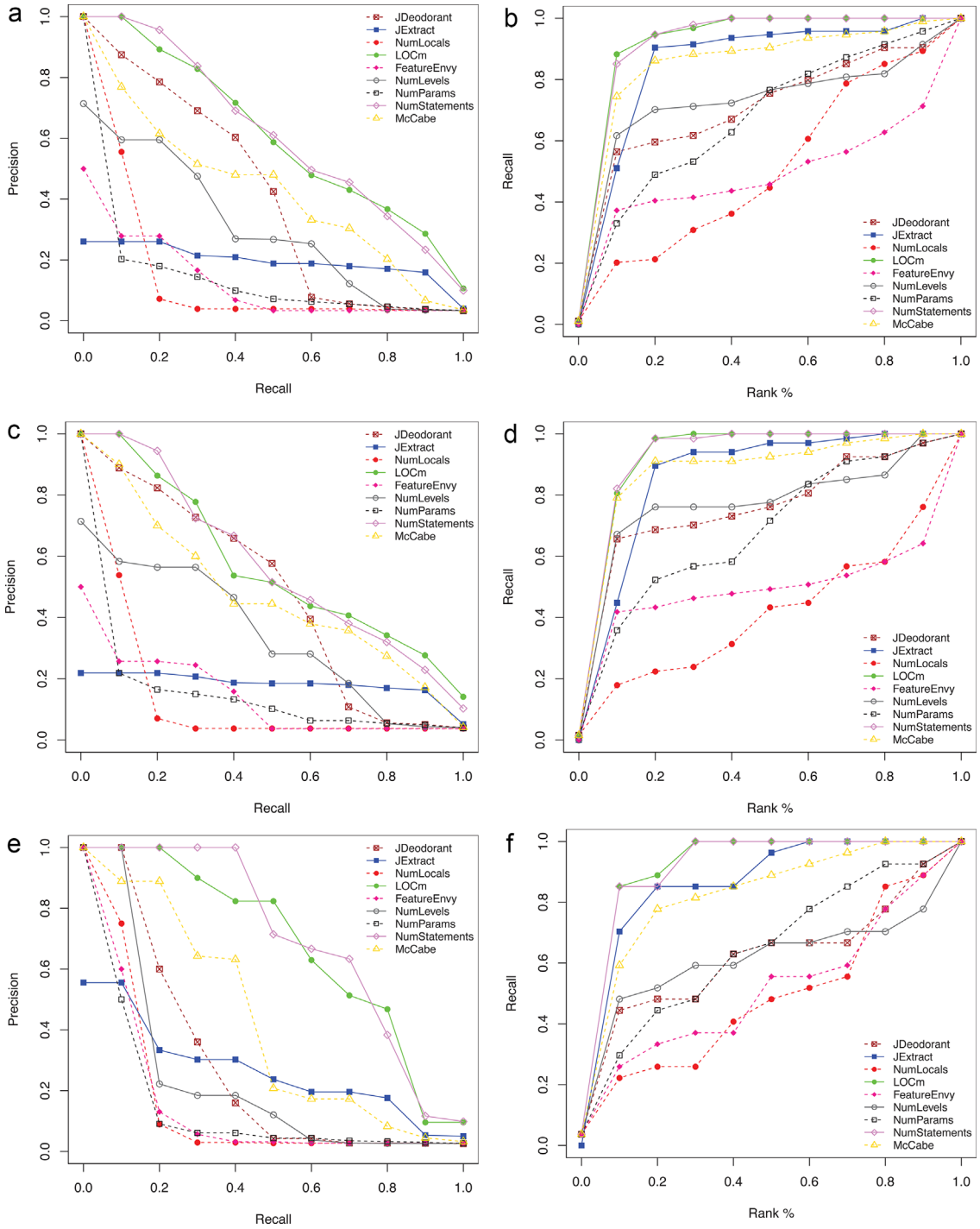


Figure 2 Precision and Recall Analysis: (a) Precision vs. Recall, (b) Recall vs. Rank Position, (c) Precision vs. Recall (2013/1), (d) Recall vs. Rank Position (2013/1), (e) Precision vs. Recall (2014/2), (f) Recall vs. Rank Position (2014/2).

we can note that it is impossible to set a threshold for the LOCm metric such that all methods from the refactor group are concentrated above it and all methods from the ok group are concentrated below it. This is unsurprising, as we should not expect a single metric to perfectly capture the judgment criteria of a human expert.

DISCUSSION

Based on the results obtained in our study, we now proceed to a discussion regarding the use of code quality tools in software engineering education. First, we discuss the effectiveness of the tools and metrics investigated in this study (Effectiveness of

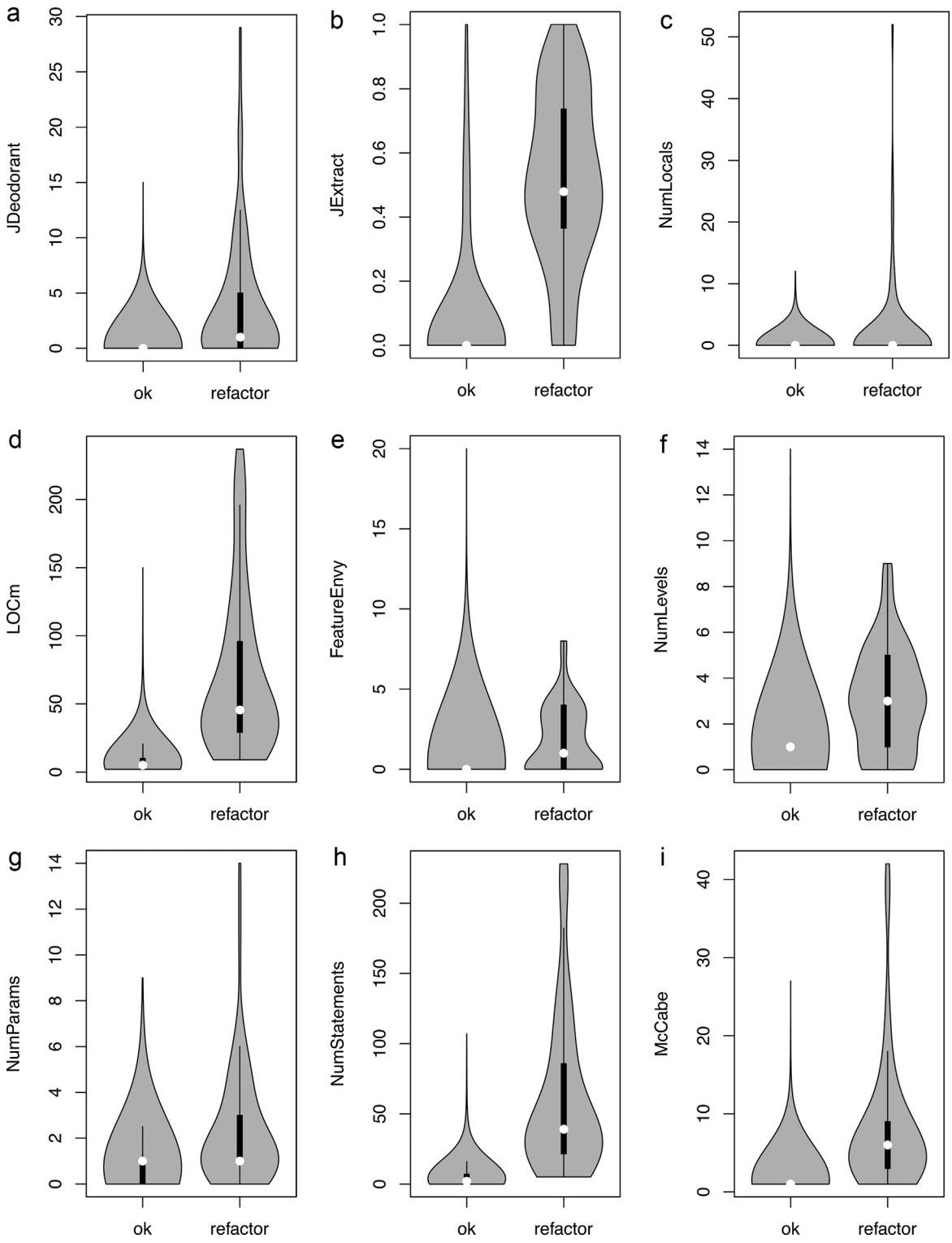


Figure 3 Violin plots comparing the distribution of values between ok and refactor groups: (a) JDeodorant, (b) JExtract, (c) NumLocal, (d) LOCm, (e) FeatureEnvy, (f) NumLevels, (g) NumParams, (h) NumStatements, (i) McCabe.

Tools in Small-Scale Student Projects section) and, second, analyse their effectiveness with respect to different thresholds (Threshold Identification for Tools section). Third, despite our study focuses on the use of code quality tools to assist the grading process, we also discuss how students can use such tools (Providing Refactoring Tools for Students section). Finally, we point out threats to the validity of our study (Threats to Validity section).

Effectiveness of Tools in Small-Scale Student Projects

Our study evaluated the effectiveness of existing tools and metrics to identify methods that should be refactored in student projects. This means that we are evaluating them not only in the context of small-scale projects, but also with source code written by inexperienced programmers. Moreover, such programmers may have strong procedural programming habits, because they learned this paradigm before learning object-orientation. This kind of evaluation is typically not performed in the research community, as results would not be generalizable to real-world applications, which is of interest of researchers.

As introduced, JDeodorant provides recommendations for different kinds of refactorings. In previous work, Tsantalis and Chatzigeorgiou [11] evaluated its effectiveness to identify Extract Method opportunities with existing software projects—actually parts of systems, because of their size. This evaluation showed that JDeodorant "demonstrated a precision of 51% and a recall of 69% on average." In contrast, our results indicate worse performance considering our investigated scenario, ranging from 25% of precision and 56% of recall to 100% of precision and 3% of recall, as discussed in Precision and Recall section—and this is valid for both groups of projects (2013/1 and 2014/2). Nevertheless, the kind of systems we are investigating is different; consequently, results are expected to be different.

JExtract was previously evaluated with open-source and commercial systems [10]. In this previous evaluation, a recommendation was considered adequate not only if experts certified that the method with a recommendation needs refactoring, but also the recommended statements are those that should be extracted. Therefore, it is not possible to directly compare our results to previous evaluations.

We believe that the difference between results obtained with large- and small-scale projects is due mainly to the programming style of the students that participated of the study, which are inexperienced. One of their programming habits, which in fact they were requested to abandon, is to declare local variables in the beginning of the method. This is something that experienced programmers typically do not do and hence is a scenario unaddressed by the rationale underlying code quality tools, or the adopted heuristics. Consequently, precision and recall are expected to be different.

Threshold Identification for Tools

Most code quality tools that identify potential problems in the code rely on a metric, which indicates the degree of a particular property of the code, for example, cohesion or coupling. Therefore, in order to give a binary answer—whether there is a problem in the code or not—thresholds must be specified. This is the case for the metrics adopted by these tools, or traditional metrics, such as those investigated in this paper such as lines of code or number of statements. As a consequence, selecting an appropriate threshold is crucial, and in general this means specifying the desired compromise between precision and recall. This issue is clearly seen in Figure 2a,c,e which show the higher the recall, the lower the precision.

Based on our study, we do not recommend any specific threshold for any of the investigated tools and metrics because this depends on the goal of the lecturer that is using them to evaluate student projects. On the one hand, if the goal is to point out *representative problems*, but not all of them, a *high* threshold may be selected. In this case, it is very likely that the recommended methods indeed must be refactored, and the teacher needs to evaluate a very limited number of methods. However, not all problems in the code will be reported. On the other hand, if the goal is to indicate *all problems* in the code with respect to the Extract Method refactoring, a *low* threshold must be set. In this case, many of the recommended methods will be false positives; however, none or few methods that must be refactored will remain unreported. Even though with a low threshold many methods must be inspected, the adopted tools and metrics can still reduce the number of methods to be analysed, thus reducing the effort to evaluate student projects.

Note that it is very unlikely, or even impossible, that we are able to achieve 100% of both precision and recall solely with metrics. As can be seen in the violin plots in Figure 2, there is no threshold that could separate the ok and refactor groups, for all metrics. Perhaps, heuristics for our specific scenario can be proposed, as suggested above, or different metrics can be considered to make a recommendation.

Providing Refactoring Tools for Students

The motivation of our work is to reduce the effort of teachers while marking and grading student projects. This task is time-consuming since providing adequate feedback for students typically requires manual inspection of the code. As discussed earlier, metrics and tools can be indeed leveraged to support this. Nevertheless, not only such tools can be used by teachers, but can also be exploited in the teaching process and be adopted by the students themselves.

The idea is that students may use code quality tools to inspect their code and evaluate themselves their quality, understanding scenarios in which they did not properly follow object-oriented principles. A simple approach is to ask them to collect metrics from their code and analyse them. However, students that just learned object-orientation are not experienced enough to do so. Therefore, pointing out where the problems are is important.

As discussed above, the goal of using a tool or a metric to identify issues in the code influences the choice for a threshold. In the case of inexperienced students, they may not know, based on the identification of a problem in their code, whether it is a true positive or a false positive. Therefore, in this case, it is *better to prioritise precision*, instead of recall, by setting a high threshold.

The use of code quality tools can be used in a 2-step project to be assigned to students, as illustrated in Figure 4. The first step consists of the development of a system that implements a given specification and, as result, students produce code. This step is similar to what was performed in the projects used in our study. The student projects will be put into a repository, which will be used by code quality tools. Then, in the second step, students receive feedback from these code quality tools, and should refactor their projects and also write a report of the changes made, and why they were needed. A variation of this step is to exchange projects among students, so that students that did not implement it assess the quality of a project. The advantage of building a repository is that all projects in it implement the same specification, so the metrics of all projects should have a similar behaviour.

This idea of introducing the use of code quality tools to help students learn object-oriented principles emerged from our study and we plan to adopt them in future classes.

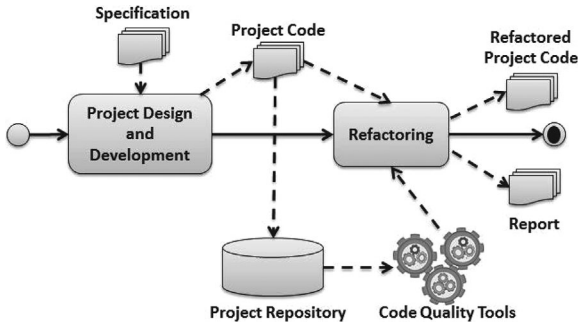


Figure 4 Tool-supported Project Development.

Note that we do not suggest the use of open-source datasets to collect metrics, or identify thresholds as discussed above, because student projects are small-scale projects and may not follow industrial patterns. Because of the small size of such projects, metrics are potentially sensitive to small variations. Consequently, using open-source repositories may be inadequate in our scenario.

Threats to Validity

A threat to the validity of our study is that a single individual provided our ground truth. And this is a potential threat because humans may not identify all refactoring opportunities when analysing the code. This has been mentioned by Tsantalis and Chatzigeorgiou [11], based on their empirical evaluation of JDeodorant. We emphasise, however, that our target systems are small and implement simple specifications. Therefore, it is trivial to analyse such systems in comparison with real-world systems. In addition, the teacher involved in this study, as said before, has both academic and industrial experience, and carefully analysed all projects. In fact, this teacher is the second author of this paper, but she evaluated the projects without knowledge about this study; as a consequence, there was no bias towards any of the tools or metrics. The second author of this paper learned about the other authors' work after 2014/2 and, based on a discussion among them, the idea of this study emerged. Therefore, the data analysed in this work was built before our study was conceived, and appropriate actions were taken to conduct it.

An external threat to the validity of this study is the number of projects. Note that our analysis was always within a group of projects—developed by a single class of a semester—therefore, regardless of how many classes we assess, the number of projects investigated together would be always small, considering the typical size of classes in the investigated university. However, analysing more classes to which the same project was assigned would provide further evidence to support our conclusions. The two classes that we analysed not only implemented different systems, but also received different prior assignments to be done. Despite they had these differences, we managed to identify similarity in the results: NumStatements and LOCm were the best metrics in the two groups, and had similar behaviour. Therefore, although we did not have two instances of similar classes, different classes allowed us to investigate similarity in their results. In addition, changes from one semester to another is always expected, to prevent students of copying projects made by students from previous semesters.

RELATED WORK

To the best of our knowledge, this is the first work that empirically investigates code quality tools in the context of software engineering education. However, other researchers have proposed tool-supported teaching and development processes, and investigated metric thresholds that are related to our study.

Software Engineering Education

Some studies focus on the automatic assessment of programming assignments with respect to the expected result (behaviour), not on the code quality excellence as this paper is centred on. Sánchez et al. [23] report an experience in automatically evaluating practices using surveys in Moodle for self-learning in engineering. The automatic evaluation tool, which provides an immediate feedback to the student, improved the practice exam marks and decreased the workload for the teachers. Similarly, Pape et al. [24] propose an approach to automatically evaluate and grade software engineering exercises also in Moodle. The tool, named STAGE, measures the test coverage based on test cases defined by the students; essentially, the higher the test coverage, the higher the grade. Empirical evidences show that (1) most of 250 students gave a positive or neutral feedback and (2) the automation of the assessment frees up teaching resources to improve the teaching in other ways. In another research line, Cervantes et al. [4] describe their experiences with the development of a game that aids in teaching architecture design. In contrast to traditional (and usually time-consuming) exercises, the consequences of design decisions within the game are immediately tangible, that is, the participants get rapid feedback in terms of a score. Since this is not usual in design assignments, the authors claim a great pedagogical aid. Last but not least, Ihantola et al. [25] conducted a systematic literature review of the automatic assessment tools for programming exercises. The authors discuss the major features the tools support and the different approaches they are using both from the pedagogical and the technical point of view. The authors conclude that new proprietary systems are ceaselessly developed and hence claim that open sourcing the existing tools could make them much more willingly adopted.

Tool-Supported Teaching Processes

Evans [26] proposed an approach to teach software engineering that exploits lightweight analysis tools. Similarly to our study, the author focuses on tools that offer clear and immediate benefits with minimal costs. On the other hand, his approach provides pedagogical benefit on information hiding, invariants, memory management, and security. More specifically, Evans' approach relies on LCLint to exploit source code annotations, ESC/Java to incorporate theorem provers, and Daikon to determine likely program invariants.

Tool-Supported Development Processes

There are approaches that promote the use of specific tools in the development processes. For instance, Fox and Patterson propose [2] an approach that establishes Ruby On Rails for software architecture, RSPec for test-first development and unit tests, Cucumber for behaviour-driven design and integration tests, Pivotal Tracker for agile iteration-based project management, etc. In contrast, these approaches focus on supporting the development process, not on increasing the code and design quality.

Identification of Thresholds

Although the large number of software metrics, the effective use of software metrics is hindered by the lack of meaningful thresholds. In this context, there are studies that derive thresholds from programming experience [27], from metric analysis [19], by stating methodologies for characterizing metric distributions [18,28], or by proposing methods to derive thresholds [29]. In contrast, all aforementioned studies have investigated systems developed by experienced programmers, which is potentially different from systems developed by students that are learning software engineering principles for the first time.

CONCLUSION

There is a gap between the theoretical understanding and the practical application of several software engineering principles, such as separation of concerns and modularity. This study was centred on the problem of a method having many responsibilities, which is a recurrent problem we observed while teaching software engineering due to students' procedural programming background. Our goal, therefore, consisted of verifying *whether code quality tools can help teachers identify problems in the design and code of software engineering students to support them in the grading process*.

To address this shortcoming, we exploited two refactoring recommendation systems, namely JDeodorant and JExtract, and seven metrics, namely NumLocals, LOCm, FeatureEnvy, NumLevels, NumParams, NumStatements, and McCabe, to provide some automation to the assessment of student projects with respect to methods with many responsibilities and evaluated their effectiveness. We conducted a study using two sets of student projects, developed in two academic semesters, focusing on the Java programming language.

First, we investigated the precision vs. recall curves. In an overall comparison, simple size metrics, namely NumStatements and LOCm, showed to be better indicators of methods that need refactoring than all other metrics, in which more than 80% of the methods with problems are in the top 10% of the ranked list. Nonetheless, the characteristics of the system impact on some metrics. For instance, JDeodorant achieves significantly better results in the Bank Management System (2013/1), whereas JExtract in the Simple Conference Support System (2014/2). Thus, we concluded that NumStatements and LOCm are effective and can be used as a rule-of-thumb, and other metrics can be selected according to the system being implemented. However, there must be a choice for high precision, for example, 100% of precision and 17% of recall with NumStatements, or for high recall, for example, 33% of precision and 81% of recall with LOCm. Therefore, if a teacher wants to provide examples of code problems, inspecting only a small portion of the code, high precision should be prioritised. However, if (most of) all problems must be identified, high recall is more important—but the tool can still help reduce the amount of code analysed.

Second, we investigated the distribution of the metrics by comparing the values of the methods with no issues (*ok*) and of those with issues (*refactor*). We obtained statistical evidence that the expected value for all metrics is really higher in the refactor group. However, it is impossible to set a threshold for any metric such that all methods from the refactor group are above it and all methods from the ok group are below it. This is unsurprising, as we should not expect a single metric to perfectly capture the judgment criteria of a human expert.

Last, we state the contributions of our study: (1) the evaluation of the effectiveness of tools to identify Extract Method refactoring opportunities in small-scale student projects to support teachers in evaluating student projects; and (2) additional empirical evidences of the difficulty in setting thresholds for tools and metrics. In addition, our study allowed us to derive the proposal of a tool-supported teaching process where all students develop a system that implements a given specification and push it into a central repository. Since metrics of all projects should have a similar behaviour, students receive solid feedback from the code quality tools, and can refactor their projects properly.

Ideas for future work include (1) evaluating the educational gains with our proposed tool-supported project development and (2) investigating other code quality tools, such as those that identify bad smells or other refactorings.

ACKNOWLEDGMENTS

Our research is supported by CAPES, FAPEMIG, and CNPq. Ingrid Nunes thanks for research grants CNPq ref. 303232/2015-3, CAPES ref. 7619-15-4, and Alexander von Humboldt, ref. BRA 1184533 HFSTCAPES-P.

REFERENCES

[1] N. R. Mead, Software engineering education: How far we've come and how far we have to go, J Syst Softw 82 (2009), 571–575.

[2] A. Fox and D. Patterson, Engineering Software as a Service: An Agile Approach Using Cloud Computing. Strawberry Canyon LLC, San Francisco, USA, 2013.

[3] V. T. Heikkilä, M. Paasivaara, and C. Lassenius. Teaching university students Kanban with a collaborative board game, in: 38th International Conference on Software Engineering (ICSE), 2016, pp 471–480.

[4] H. Cervantes, S. Haziyev, O. Hrytsay, and R. Kazman, Smart decisions: an architectural design game, in: 38th International Conference on Software Engineering (ICSE), 2016, pp 327–335.

[5] C.-Y. Chen and P. P. Chong, Software engineering education: A study on conducting collaborative senior project development, J Syst Softw 84 (2011), 479–491.

[6] I. Kádár, P. Hegedüs, R. Ferenc, and T. Gyimóthy, Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods, Computational Science and Its Applications—ICCSA 2016, Volume 9789 of the series Lecture Notes in Computer Science, 2016, pp 610–624.

[7] I. Kádár, P. Hegedus, R. Ferenc, and T. Gyimóthy, A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability, in: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp 599–603.

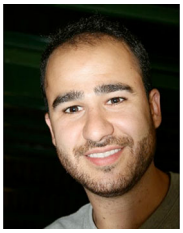
[8] F. Arcelli Fontana, M. Zanoni, and F. Zanoni. A duplicated code refactoring advisor, in: 16th International Conference on Agile Software Development (XP), 2015, pp 3–14.

[9] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier and M. Monperrus, B-Refactoring: Automatic test code refactoring to improve dynamic analysis, Inf Softw Technol 76 (2016), 65–80.
[10] D. Silva, R. Terra, and M. T. Valente, Recommending automated extract method refactorings, in: 22nd International Conference on Program Comprehension (ICPC), 2014, pp 146–156.
[11] N. Tsantalis and A. Chatzigeorgiou, Identification of extract method refactoring opportunities for the decomposition of methods, J Syst Softw 84 (2011), 1757–1782.
[12] J. Van Eyck, N. Boucḱe, A. Helleboogh, and T. Holvoet, Using code analysis tools for architectural conformance checking, in: 6th International Workshop on Sharing and Reusing Architectural Knowledge (SHARK), ACM, New York, USA, 2011, pp 53–54.
[13] C. Thomas Wu, An Introduction to Object-Oriented Programming with Java, 5th ed, McGraw-Hill Education, New York, USA, 2009.
[14] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[15] N. Tsantalis, and A. Chatzigeorgiou, Identification of Move Method refactoring opportunities, IEEE Transactions on Software Engineering 35 2009 347–367.
[16] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, JDeodorant: Identification and removal of feature envy bad smells, in: 23rd International Conference on Software Maintenance (ICSM), 2007, pp 519–520.
[17] K. Maruyama, Automated method-extraction refactoring by using block-based slicing. In: Software reusability: putting software reuse in context, Prooceedings of the 2001 symposium, (SSR '01), ACM, New York, NY, USA, 2001, pp 31–40. DOI=https://doi.org/10.1145/375212.375233
[18] S. R. Chidamber, and C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 1994 476–493.
[19] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, 2010.
[20] B. Henderson-Sellers, Object-oriented metrics: measures of complexity. Prentice-Hall, 1996.
[21] R. Martin, OO design quality metrics: an analysis of dependencies, in: Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, New York, NY, USA, 1994, pp 1–8.
[22] V. Basili, R. Selby, and D. Hutchens, Experimentation in software engineering, IEEE Transactions on Software Engineering 12 1986 733–743.
[23] C. Sánchez, O. Ramos, P. Márquez, E. Martí, J. Rocarias, and D. Gil. Automatic evaluation of practices in Moodle for Self Learning in Engineering. J Technol Sci Educ. [Online] (2015) 5:2.
[24] S. Pape, J. Flake, A. Beckmann, and J. Jürjens, STAGE: a software tool for automatic grading of testing exercises: case study paper, in: 38th International Conference on Software Engineering Companion (ICSE), 2016, pp 491–500.
[25] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments, in: 10th Koli Calling International Conference on Computing Education Research, 2010, pp 86–93.
[26] D. Evans, Teaching software engineering using lightweight analysis, Technical report, University of Virginia 2001.
[27] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (4) (1976) 308–320.
[28] P. Oliveira, M. T. Valente, and F. Lima, Extracting relative thresholds for source code metrics, in: IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014, pp 254–263.
[29] T. Alves, C. Ypma, and J. Visser, Deriving metric thresholds from benchmark data, in: International Conference on Software Maintenance (ICSM), 2010, pp 1–10.

BIOGRAPHIES

Danilo Silva holds a master's degree in Computer Science from Federal University of Minas Gerais, Brazil, and he is currently a PhD student in Computer Science at the same university. His research interests include software evolution, object-oriented design, and code refactoring. Contact him at danilofs@dcc.ufmg.br.

Ingrid Nunes is a senior lecturer at the Institute of Informatics, Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, currently in a sabbatical year at TU Dortmund in Germany. She obtained her PhD in informatics at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. Her PhD was in cooperation with King's College London (UK) and University of Waterloo (Canada). She is the head of the Prosoft research group, and her main research areas are software maintenance and evolution and agent-oriented software engineering.



Ricardo Terra received his PhD degree in Computer Science from Federal University of Minas Gerais, Brazil (2013) with a 1-year internship at the University of Waterloo, Canada. Since 2014, he is an assistant professor in the Department of Computer Science at Federal University of Lavras, Brazil. His research interests include software architecture maintainability and evolvability. Contact him at terra@dcc.ufla.br, or visit www.dcc.ufla.br/~terra.