Code Quality Issues in Student Programs

Hieke Keuning
Open University of the Netherlands
and Windesheim University of
Applied Sciences
hw.keuning@windesheim.nl

Bastiaan Heeren
Open University of the Netherlands
bastiaan.heeren@ou.nl

Johan Jeuring
Utrecht University and Open
University of the Netherlands
j.t.jeuring@uu.nl

ABSTRACT

Because low quality code can cause serious problems in software systems, students learning to program should pay attention to code quality early. Although many studies have investigated *mistakes* that students make during programming, we do not know much about the *quality* of their code. This study examines the presence of quality issues related to program flow, choice of programming constructs and functions, clarity of expressions, decomposition and modularization in a large set of student Java programs. We investigated which issues occur most frequently, if students are able to solve these issues over time and if the use of code analysis tools has an effect on issue occurrence. We found that students hardly fix issues, in particular issues related to modularization, and that the use of tooling does not have much effect on the occurrence of issues.

KEYWORDS

Code quality, programming education

ACM Reference format:
Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of ITiCSE'17, July 03-05, 2017, Bologna, Italy.*, , 6 pages.
DOI: http://dx.doi.org/10.1145/3059009.3059061

1 INTRODUCTION

Students who are learning to program often write programs that can be improved. They are usually satisfied once their program produces the right output, and do not consider the quality of the code itself. In fact, they might not even be aware of it. Code quality can be related to documentation, presentation, algorithms and structure [11]. Fowler [7] uses the term 'code smells' to describe issues related to algorithms and structure that jeopardise code quality. A typical example is duplicated code, which could have been put in a separate method. Another example is putting the same code in both the true-part and the false-part of an if-statement, even though that code could have been moved outside the if-statement. Low quality code can cause serious problems in the long term, which affect software quality attributes such as maintainability, performance and security of software systems. It is therefore imperative to make students and lecturers aware of its importance.

ITiCSE'17, July 03-05, 2017, Bologna, Italy.

DOI: http://dx.doi.org/10.1145/3059009.3059061

For a long time, researchers have been interested in how students solve programming problems and the mistakes that they make. Recently, large-scale data mining has made it possible to perform automated analysis of large numbers of student programs, leading to several interesting observations. For example, Altadmri and Brown [3] investigated over 37 million code snapshots and found that students seem to find it harder to fix semantic and type errors than syntax errors.

Although many studies have investigated the errors that students make, little attention has been paid to code quality issues in student programs. While Pettit et al. [10] looked at code quality aspects and found that several metrics related to code complexity increased with each submission, their study does not elaborate on the causes of these high metric scores. Aivaloglou and Hermans [1] analysed a large database of Scratch projects by measuring complexity and detecting different code smells. Although the complexity of most Scratch projects was not high, the researchers found many instances of these code smells.

In this study we analyse a wider range of code quality issues, and observe their appearance over time. Our data set, taken from the Blackbox database [6], contains over two million Java programs of novice programmers recorded in four weeks of one academic year. First, we investigate the type and frequency of code quality issues that occur in student programs. Next, we track the changes that students make to their programs to see if they are able to solve these issues. Finally, we check if students are better at solving code quality issues when they have code analysis tools installed.

The contributions of this paper are: (1) a selection of relevant code quality issues for novice programmers, (2) an analysis of the occurrence and fixing of these issues, and (3) insight into the influence of code analysis tools on issue occurrence.

The remainder of this paper is organised as follows: Section 2 elaborates on related studies on student programming behaviour. Section 3 describes the research questions, the data set we used, the code quality issues we have selected to investigate, and the automatic analysis. Section 4 shows the results for each research question, which are discussed in Section 5. Section 6 concludes and describes future work.

2 RELATED WORK

This section discusses previous research into student programming habits related to code quality. We also consider studies that have analysed student programming behaviour on a large scale.

Pettit et al. [10] have analysed over 45,000 student submissions to programming exercises. The authors monitored the progress that students made over the course of a session, in which students submit their solutions to an automated assessment tool that provides feedback based on test results. For each submission they computed

several metrics: lines of code, cyclomatic complexity, state space (number of unique variables) and the six Halstead complexity measures (calculations based on the number of operators and operands of a program). The authors also distinguish between sessions in which the number of attempts within a specific time frame is restricted. The main conclusion from the study is that although the metric scores increase with each submission attempt, restricting the number of attempts has a positive influence on the code quality of students. Second, the authors argue that instructors should also consider coding style and quality, because focusing solely on testing may result in inefficient programs. The study does not elaborate on the particular problems that cause high complexity scores.

Aivaloglou and Hermans [1] analysed a database of over 230,000 Scratch projects. Scratch is a block-based programming language that is often used to teach children how to program. Besides investigating general characteristics of Scratch programs, the authors also looked at code smells, such as cyclomatic complexity, duplicate code, dead code, large scripts and large sprites (image objects that can be controlled by scripts). Translating to the object-oriented domain, a large script is comparable to a large class and a large sprite to a large method. In 78% of over 4 million scripts the cyclomatic complexity is one. Only 4% of the scripts has a complexity over four. In 26% of the projects the researchers identified code clones (12% for exact clones), consisting of at least five blocks. It should be noted that Scratch only supports procedure calls within sprites, leaving copy-pasting code as the only option. Dead code occurs in 28% of the projects. Large scripts (with at least 18 blocks) are present in 30% of the projects and large sprites (with at least 59 blocks) in 14% of the projects.

Breuker et al. [5] investigated the differences in code quality between first and second year students in approximately 8,400 Java programs in 207 projects, using a set of 22 code quality properties. They found that for half of the properties there were no major differences. For the remaining properties, some differences could be attributed to increased project size and complexity for second year students. Finally, second year students performed better because their code had smaller methods, fewer short identifiers, fewer static methods and fewer assignments in while and if-statements.

Much more research into code smells exists for professional code. For example, Tufano et al. [12] investigated the repositories of 200 software projects, answering the question when and why smells are introduced. They calculated five metrics related to the size and complexity of classes and methods, and proper use of object-orientation. They found that most smells first occur when a file is created and that, surprisingly, refactorings may introduce smells.

Altadmri and Brown [3] used data from one academic year of the Blackbox database to investigate what common student mistakes are, how long it takes to fix them, and how these findings change during an academic year. Although there are various other studies that look at these aspects, it had not been done on such a large scale before. Individual source files were tracked over time by checking them for 18 mistakes, and calculating how much time had passed before the mistake disappeared from the source file. One important observation from the study is that students seem to find it harder to fix semantic and type errors than syntax errors.

3 METHOD

This study addresses the following research questions:

RQ1 Which code quality issues occur in student code?
RQ2 How often do students fix code quality issues?
RQ3 What are the differences in the occurrence of code quality issues between students who use code analysis extensions compared to students who do not?

3.1 Blackbox database

Our data set is extracted from the Blackbox database [6], which collects data from students working in the widely used BlueJ IDE[1] for novice Java programmers. BlueJ, used mostly in first year programming courses, has a simplified user interface and offers several educational features, such as interacting with objects while running a program.

The Blackbox database stores information about events in BlueJ triggered by students, such as compiling, testing and creating objects. Blackbox stores data on sessions, users, projects, code files and tests, which are linked to these events. A *source file* is a file of which there may be multiple versions called *snapshots*, which are unique instances of the source file at a certain *event*.

The database has been receiving data constantly since June 2013, and contains millions of student programs to date. BlueJ users have to give prior consent (opt-in) to data collection, and all collected data is anonymous. Permission is required to access the database. In this study we have investigated programs submitted in four weeks of the academic year 2014–2015 (the second week of September, December, March and June). From the Blackbox database we extracted data on source files, snapshots, compile events, extensions and startup events, which we stored in a local database. We only extracted data on programs that are compilable.

3.2 Data analysis

We performed an automatic analysis of all programs in our data set that compiled successfully. To enable replication and checks, we have published the code online.[2] We counted the source lines of code (SLOC) for each file using the cloc tool.[3] Although this metric is sensitive to style and formatting and therefore not very accurate, it provided us with an indication of the size of a program.

3.2.1 Issues (RQ1). Stegeman et al. [11] have developed a rubric for assessing code quality, based on their research into professional code quality standards from the software engineering literature and interviews with instructors. The rubric is based on a model with ten categories for code quality. We omit the categories that deal with documentation (the names, headers and comments categories) and presentation (the layout and formatting categories). Our study focuses on the remaining five categories that deal with algorithms and structure, because they are the most challenging for students:

Flow Problems with nesting and paths, code duplication and unreachable code.
Idiom Unsuitable choice of control structures and no reuse of library functions.

[1] http://www.bluej.org
[2] https://github.com/hiekekeuning/student-code-quality
[3] https://github.com/AlDanial/cloc

Expressions Expressions that are too complex and use of unsuitable data types.

Decomposition Methods that are too long and excessive sharing of variables.

Modularization Classes with an unclear purpose (low cohesion) and too many methods and attributes, and tight coupling between classes.

For each category, we selected a number of issues to investigate by applying the PMD tool to a limited set of student programs to identify the issues that occur most frequently. PMD[4] is a well-known static analysis tool that is able to detect a large set of bad coding practices in Java programs. We also used the Copy/Paste Detector tool (CPD)[5] included with PMD for duplicate detection. In PMD a *rule* defines a bad coding practice, and running PMD results into a report of rule violations. In this paper we use the term *issue* to refer to a rule in PMD. The PMD version we used offers 26 sets consisting of issues that all deal with a particular aspect.

We discarded sets of issues using the following criteria:

- An issue is too specific for Java, such as issues that apply to Android, JUnit and Java library classes.
- An issue is too advanced, strict or specific for novice programmers, such as exceptions, threads, intermediate-level OO concepts (abstract classes, interfaces) and very specific language constructs (e.g. the final keyword).
- An issue falls under the documentation or presentation categories.
- An issue points at an actual error.

Our first selection consisted of 170 issues in 12 sets. We used the default value for issues with a minimal reporting threshold, such as the value 3 for reporting an if-statement that is nested too deeply. Additionally, we added 'code duplication' as three issues that fire for duplicates of 50, 75 and 100 tokens. Our initial analysis was applied to a smaller set of programs from four different days throughout the academic year 2014–2015. For each unique source file we recorded for each issue if it occurred in some snapshot of that file.

For a more detailed analysis we made a selection of the 170 issues. For each issue we decided whether it should be included or not, based on the criteria mentioned above. We also discarded all issues in the 'controversial' set, 'import statements' set and the 'unused code' set, and issues that occurred in fewer than 1% of the unique files. Table 1 shows our final set of issues, now grouped according to the categories of Stegeman et al.

We ran PMD for these 24 issues on all compilable programs in the final data set of four weeks and stored the results in our local database. We cleaned the database by removing all data of the files that could not be processed and files with 0 LOC. For each of these 24 issues, we counted in how many unique source files it occurred at least once, and how often. We also calculated the differences in occurrence over time.

3.2.2 Fixing (RQ2). For RQ2 we examine the changes in a source file over time. For each issue we calculated the number of *fixes* and the number of *appearances*. As an example, let us assume that source file X has 6 snapshots in which the occurrences of issue Y are `2 1 3 0 4 2`. The number of fixes is 6: the total number of issues that were solved in a subsequent snapshot (1 + 0 + 3 + 0 + 2). The number of appearances is 8: the total number of issues that were introduced in a subsequent snapshot (2 + 0 + 2 + 0 + 4 + 0). These metrics are simplified measures to investigate fixing: we cannot be sure the student really fixed the problem, or simply removed the problematic code.

Table 1: Selected issues (report level) per category

Flow	
CyclomaticComplexity (10)	Strict version that counts boolean operators as decision points.
ModifiedCyclomaticComplexity (10)	Counts switch statements as a single decision point.
NPathComplexity (200)	
EmptyIfStmt	
PrematureDeclaration	
Idiom	
SwitchStmtsShouldHaveDefault	
MissingBreakInSwitch	
AvoidInstantiatingObjectsInLoops	
Expressions	
AvoidReassigningParameters	
ConfusingTernary	
CollapsibleIfStmts	
PositionLiteralsFirstInComparisons	
SimplifyBooleanExpressions	
SimplifyBooleanReturns	
Decomposition	
NCSSMethodCount (50)	Counts Non-Commenting Source Statements, report level in statements.
NCSSMethodCount (100)	
SingularField	The scope of a field is limited to one method.
CodeDuplication (50)	Only identified in single files, not over projects.
CodeDuplication (100)	
Modularization	
TooManyMethods (10)	Excludes getters and setters.
TooManyFields (15)	
GodClass	
LawOfDemeter	Call methods from another class directly.
LooseCoupling	Use interfaces instead of implementation types.

3.2.3 Extensions (RQ3). BlueJ users may install various extensions to support their programming, such as UML tools, submission tools and style checkers. We generated a list of all extensions used in the selected four weeks of the year 2014–2015. We selected extensions related to code quality from the 29 that were active in at least 0.05% of all BlueJ-startups in those weeks:

- Checkstyle[6] (9,626 start-ups), a static analysis tool for checking code conventions.
- PMD (3,751 start-ups), the tool used for our analysis.
- PatternCoder[7] (507 start-ups), which helps students to implement design patterns.

Findbugs[8] translates Java code into bytecode, and then performs static analysis to identify potential bugs. It is a relevant tool, but with 242 start-ups not used often enough. We also excluded a small number of extensions that we could not find information about.

For RQ3, we calculated the occurrence of issues for each of the extensions, and for source files for which no extensions were used.

[4] http://pmd.github.io/pmd-5.5.2/
[5] http://pmd.github.io/pmd-5.4.1/usage/cpd-usage.html
[6] http://checkstyle.sourceforge.net
[7] http://www.patterncoder.org
[8] http://findbugs.sourceforge.net

4 RESULTS

Table 2 shows some general information on the data sets taken from the academic year 2014–2015.

Table 2: Data set summary

Initial data set (4 days)	Unique source files	90,066
	Snapshots	439,066
Final data set (4 weeks)	Unique source files	453,526
	Snapshots	2,661,528
	Avg events per source file	5.87
	Median events per source file	2
	Max events per source file	700
	Average LOC per source file	52.75
	Median LOC per source file	27

4.1 All issues (RQ1)

Table 3 shows the summary of checking the initial data set of four days for 170 issues. For each unique source file we recorded for each issue if it occurred in some snapshot of that file. In total we found 574,694 occurrences of 162 different issues (8 issues did not occur in any file). The top 10 issues is shown in Table 4.

In the controversial set, seven issues were found in at least 5% of the unique source files. DataFlowAnomalyAnalysis is at the top of the list with 38.6%. This issue deals with redefining variables, undefinitions (variables leaving scope) and references to undefined variables, which may not always be a serious problem. AvoidLiteralsInIfCondition is second with 14.0%. For other issues such as AtLeastOneConstructor and OnlyOneReturn it is also questionable whether they are problematic in novice programmer code, therefore we decided to further omit all issues in this set.

The top 10 also includes issues that we omit in the remainder of this study. The two issues that occur in the most files, 84.2% for MethodArgumentCouldBeFinal and 61.3% for LocalVariableCouldBeFinal, are both in the optimization set and point at the possibility to use the final keyword to indicate that a variable will not be reassigned. A reason for these high percentages may be that this language construct is not being taught to novice programmers. UseVarargs deals with the 'varargs' option introduced in Java 5, allowing parameters to be passed as an array or as a list of arguments. UseUtilityClass points at the option to make a class with only static methods a utility class with a private constructor. ImmutableField detects private fields that could be made final.

4.2 Selected issues (RQ1)

We now focus on the selection of 24 issues in five categories (Flow, Idiom, Expressions, Decomposition, Modularization), which we applied to our final data set of four weeks. In total we found over 24 million instances of these issues. Table 5 shows in how many unique source files an issue occurs at least once, and the average number of occurrences per KLOC. To calculate this last value, we first calculated the average for each source file, and then the overall average, so the number of snapshots of a source file does not influence the total.

LawOfDemeter stands out as an issue with a very high number of occurrences. Upon closer inspection, it was not always clear why this issue was reported, and it has been suggested online that there might be false positives. We therefore decided to omit this issue in the remainder of this study.

Table 3: Summary of running PMD on the initial data set, showing per PMD set the number of issues that were seen, the percentage of unique files in which at least one issue from that set occurred, the median of the occurrences in % and the maximum.

Set	Issues seen	% of files with issues from set	Median %	Max %
Type resolution	4/4	26.04	3.96	20.1
Optimization	12/12	91.75	2.71	84.2
Unused code	5/5	26.86	2.50	16.2
Code duplication	3/3	4.99	2.28	5.0
Code size	13/13	13.69	1.40	8.2
Controversial	21/22	65.10	1.37	38.6
Import statements	6/6	10.61	1.02	8.5
Design	54/57	81.73	0.32	38.0
Unnecessary	8/8	10.25	0.11	9.6
Empty code	10/11	5.18	0.08	2.2
Coupling	3/5	41.98	0.04	39.7
Basic	23/24	2.52	0.02	1.3

Table 4: Top 10 issues

Set	Issue	In % of files
Optimization	MethodArgumentCouldBeFinal	84.2
Optimization	LocalVariableCouldBeFinal	61.3
Coupling	LawOfDemeter	39.7
Controversial	DataflowAnomalyAnalysis	38.6
Design	UseVarargs	38.0
Design	UseUtilityClass	36.2
Design	ImmutableField	27.8
Type Res.	UnusedImports	20.1
Unused Code	UnusedLocalVariable	16.2
Controversial	AvoidLiteralsInIfCondition	14.0

It is expected that SingularField occurs quite often with 8.2%, because most of the snapshots in our data set are unfinished programs. CyclomaticComplexity and the more lenient ModifiedCyclomaticComplexity version occur quite often with 7.7% and 5.2% respectively, which could point to serious problems, but that depends on the type of code. LooseCoupling occurs in 6.7% of the files implying that students do not always have knowledge of the use of interfaces. Duplicate50 occurs much more often than Duplicate100 with 4.7% against 1.3%. We argue that the lower threshold of 50 tokens is more suitable for novice programmers, whose programs are relatively short, so duplicates can be spotted more easily.

Figure 1 shows the occurrence of issues by the month in which they appeared, grouped by category. In the week of September the number of issues is quite low, probably because most courses had just started and only a limited set of topics would have been introduced. For the other three months we cannot see major differences, other than an increase in decomposition issues. In March we see a slight decrease in issues mainly in the flow and expressions category, but towards the end of the academic year the values slightly increase.

Table 5: Per issue, column I shows the percentage (%) of unique files in which the issue occurs, column II shows the average number of occurrences per KLOC

Cat	Issue	I	II
M	LawOfDemeter	38.7	42.6
D	SingularField	8.2	3.8
F	CyclomaticComplexity	7.7	1.5
M	LooseCoupling	6.7	2.1
I	AvoidInstantiatingObjectsInLoops	6.3	1.6
E	AvoidReassigningParameters	5.7	1.7
F	ModifiedCyclomaticComplexity	5.2	0.8
M	TooManyMethods	5.0	0.3
D	Duplicate50	4.7	0.7
E	ConfusingTernary	4.4	0.7
D	NcssMethodCount50	3.9	0.3
E	PositionLiteralsFirstInComparisons	3.5	1.6
F	NPathComplexity	3.3	0.3
E	SimplifyBooleanExpressions	3.1	0.8
F	PrematureDeclaration	2.6	0.4
M	GodClass	2.1	0.1
F	EmptyIfStmt	2.0	0.3
E	SimplifyBooleanReturns	1.9	0.4
I	SwitchStmtsShouldHaveDefault	1.7	0.3
I	MissingBreakInSwitch	1.4	0.2
D	Duplicate100	1.3	0.1
E	CollapsibleIfStatements	1.3	0.2
M	TooManyFields	1.2	0.1
D	NcssMethodCount100	1.0	0.0

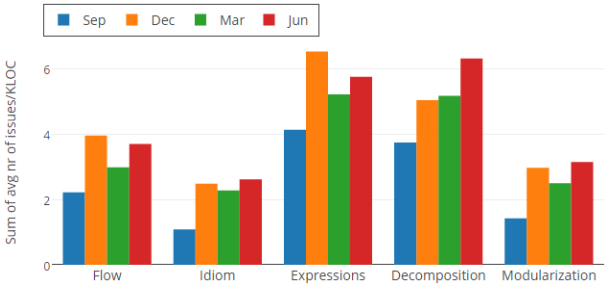

Figure 1: Issues over time

4.3 Fixing (RQ2)

Table 6 shows our fix metrics for each issue. EmptyIfStmt is solved in almost half of the cases, which can be expected because an if-statement with no code in it is probably not finished. The same can be said for SingularField: a student might start with defining the field of a class that is needed for methods that will be added later. On the bottom of the list we find four issues from the modularization category (GodClass, LooseCoupling, TooManyFields, TooManyMethods) that are fixed in fewer than 5% of the appearances.

Overall the rate of fixing issues is low. Either students do not recognise these issues in their code, or do not care to fix them. It should be noted that our data set was not cleaned of source files that continued to be fixed beyond the weeks (Monday to Sunday) we investigated, missing some possible fixes.

Table 6: Issue fixes

Cat	Issue	Appeared	Fixed	%
F	EmptyIfStmt	18,460	9,064	49.1
D	SingularField	103,004	30,152	29.3
F	PrematureDeclaration	21,008	5,891	28.0
D	Duplicate100	35,033	7,388	21.1
E	CollapsibleIfStatements	15,087	2,579	17.1
D	Duplicate50	91,951	15,520	16.9
E	AvoidReassigningParameters	76,359	10,023	13.1
I	MissingBreakInSwitch	9,594	1,033	10.8
F	NPathComplexity	20,549	2,129	10.4
E	ConfusingTernary	36,391	3,558	9.8
E	SimplifyBooleanReturns	12,612	1,162	9.2
E	SimplifyBooleanExpressions	48,965	4,347	8.9
F	ModifiedCyclomaticComplexity	56,822	4,475	7.9
I	AvoidInstantiatingObjectsInLoops	78,588	6,167	7.8
I	SwitchStmtsShouldHaveDefault	12,507	961	7.7
D	NcssMethodCount50	23,569	1,790	7.6
F	CyclomaticComplexity	85,426	6,240	7.3
D	NcssMethodCount100	6,178	410	6.6
E	PositionLiteralsFirstInComparisons	86,536	4,833	5.6
M	GodClass	9,575	437	4.6
M	LooseCoupling	57,039	2,056	3.6
M	TooManyFields	5,539	175	3.2
M	TooManyMethods	23,003	515	2.2

4.4 Extensions (RQ3)

Table 7 shows general information on the use of extensions. Figure 2 shows the differences in occurrence of issues between source files for which extensions were and were not active. The figure shows that there is only a small difference between the use of a tool compared to using no tool. Students using no tool even have a slightly smaller number of issues with 18.2 issues on average per KLOC versus 19.7 for students that use some tool.

Table 7: Extension use

Name	Snapshots	KLOCs	Unique source files
Checkstyle	73,553	7,756	10,833
PMD	26,126	1,840	4,299
PatternCoder	2,433	113	609

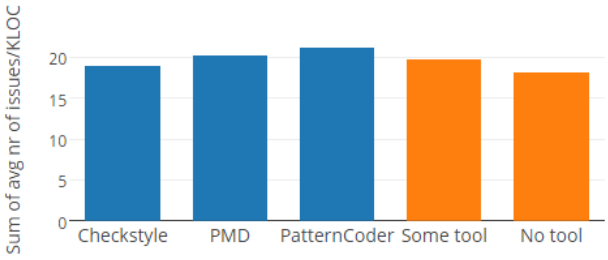

Figure 2: Issues and extension use

5 DISCUSSION

One of our main findings is that most issues are rarely fixed, especially when they are related to modularization. Another finding is that the use of tools has little effect on issue occurrence. Compared

to the study of Scratch projects by Aivaloglou and Hermans [1], we found lower percentages of files that contain duplicates, large classes and large methods. Some reasons might be that block-based code cannot be directly compared to statement-based code and that block-based programming is targeted at a younger audience. Another reason is that we investigated single source files instead of projects. Our study supports the work of Pettit et al. [10] by observing that quality issues are not often solved, although we cannot confirm the positive effect of restricting submission attempts, because our data set does not contain information on submissions.

From working with PMD as a source code analyser we have noticed some problems with regard to suitability for students. PMD integrates with many IDEs and also provides an extension for users of BlueJ. We found that many of the checks PMD can perform are not suitable for novice programmers, and may cause confusion with students that might result in neglecting the tool. We advise educators to customize the tool by selecting a small set of relevant checks and adjusting threshold values. Other recommendations for using PMD for assessing programming exercises have been proposed by Nutbrown and Higgins [9].

The main focus of the field of automated feedback and assessment of programming exercises has been on functional correctness of programs, although some tools incorporate feedback on quality aspects as well [8]. This is often done by integrating a lint-like tool or calculating metrics such as cyclomatic complexity and LOC (e.g. [2, 4]). Many professional IDEs detect code quality issues and offer refactorings, but these are often too advanced for novices and not intended to support learning. We argue that these tools need to be better suited to novices, and should be used at various moments during learning and not only for assessment.

5.1 Threats to validity

The designers of the Blackbox project mention some restrictions of their data set that also affect this study [6]. First, BlueJ is often used in courses that use an 'objects-first' approach. Second, it is unknown on what task the student is working, and what the requirements of this task are, such as using a particular language construct. Third, we know nothing about the users of BlueJ. We expect them to be novices, but some programs have probably been written by instructors or more experienced programmers.

We have a limited data set of four weeks in one year. We also cannot be sure that we have all snapshots, events might be missed because something went wrong (e.g. no internet connection) or a user continued to edit the code in another program. Because we store weeks, we miss some snapshots that were compiled just before or after the week. However, because of its size we believe our data set has enough information to answer our research questions. Only tracking single files and not complete BlueJ projects gives an incomplete view of the presence of duplicates.

Vihavainen et al. [13] have investigated the effect of storing student data of different granularity: submission-level, snapshot-level (e.g. compiling, saving), and keystroke-level (e.g. editing text), and found that data might be lost if only snapshot events are studied. Although the Blackbox data set also stores keystroke events, we believe that researching compile events provides us with sufficient information. For a more detailed analysis, investigating keystrokes could provide more insight into how students fix quality issues.

Although this study focuses on Java programs, we believe that the findings may apply to other languages too. The issues we investigated are not Java specific and can also be seen in other modern object-oriented languages. For functional and logic languages some issues are not applicable or should be adjusted for the paradigm.

6 CONCLUSION AND FUTURE WORK

In this study we have explored quality issues in 2.6 million code snapshots written by novice programmers using the BlueJ IDE. We have composed a list of issues that are relevant for novices. We found that novice programmers develop programs with a substantial amount of code quality issues, and they do not seem to fix them, especially when they are related to modularization. The use of tools has little effect on the occurrence of issues. Educators should pay attention to code quality in their courses, and automated tools should be improved to better support students in understanding and solving code quality issues. Further research is required to better understand how students deal with quality issues, for example by investigating the changes made in snapshots. Also, it is of importance to examine the reasons why students produce low-quality code: they may be unaware of it, or they simply do not know how to fix their code. Paying attention to code quality in education is vital if we want to keep improving our software systems.

ACKNOWLEDGMENTS

This research is supported by the Netherlands Organisation for Scientific Research (NWO), grant number 023.005.063.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. 2016. How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. In *Proc. of ICER*. 53–61.
[2] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. *J. of Inf. Technol. Educ.* 3, 1 (2004), 245–262.
[3] Amjad Altadmri and Neil C. C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proc. of SIGCSE*. 522–527.
[4] Eliane Araujo, Dalton Serey, and Jorge Figueiredo. 2016. Qualitative aspects of students' programs: Can we make them measurable?. In *Proc. of FIE*. 1–8.
[5] Dennis Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring Static Quality of Student Code. In *Proceedings of ITiCSE*. 13–17.
[6] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proc. of SIGCSE*. 223–228.
[7] Martin Fowler. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.
[8] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of ITiCSE*. 41–46.
[9] Stephen Nutbrown and Colin Higgins. 2016. Static analysis of programming exercises: Fairness, usefulness and a method for application. *Computer Science Education* 26, 2-3 (2016), 104–128.
[10] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An Empirical Study of Iterative Improvement in Programming Assignments. In *Proc. of SIGCSE*. 410–415.
[11] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a Rubric for Feedback on Code Quality in Programming Courses. In *Proc. of Koli Calling*. 160–164.
[12] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *Proc. of ICSE*. 403–414.
[13] Arto Vihavainen, Matti Luukkainen, and Petri Ihantola. 2014. Analysis of source code snapshot granularity levels. In *Proc. of SIGITE*. 21–26.