Are Undergraduate Creative Coders Clean Coders? A Correlation Study

Peer-reviewed author version

GROENEVELD, Wouter; Martin, Dries; Poncelet, Tibo & AERTS, Kris (2022) Are Undergraduate Creative Coders Clean Coders? A Correlation Study. In: PROCEEDINGS OF THE 53RD ACM TECHNICAL SYMPOSIUM ON COMPUTER SCIENCE EDUCATION (SIGCSE 2022), VOL 1, ASSOC COMPUTING MACHINERY, p. 314 -320.

DOI: 10.1145/3478431.3499345
Handle: http://hdl.handle.net/1942/39056

Are Undergraduate Creative Coders Clean Coders? A Correlation Study

Wouter Groeneveld
KU Leuven
Leuven, Belgium
wouter.groeneveld@kuleuven.be

Dries Martin
Hasselt University/KU Leuven
Hasselt, Belgium
dries.martin@student.uhasselt.be

Tibo Poncelet
Hasselt University/KU Leuven
Hasselt, Belgium
tibo.poncelet@student.uhasselt.be

Kris Aerts
KU Leuven
Leuven, Belgium
kris.aerts@kuleuven.be

ABSTRACT

Research on global competencies of computing students suggests that next to technical programming knowledge, the teaching of non-technical skills such as creativity is becoming very relevant. Many CS1 courses introduce a layer of creative freedom by employing open project assignments. We are interested in the quality of the submitted projects in relation to the creativity that students show when tackling these open assignments. We have analyzed 110 projects from two academic years to investigate whether there is a relation between creativity and clean code in CS1 student projects. Seven judges were recruited that evaluated the creativity based on Amabile's Consensual Assessment Technique, while the PMD tool was used to explore code quality issues in the Java projects. Results indicate that the more projects are deemed as creative, the more likely code quality issues arise in these projects, and thus the less clean the code will be. We argue that next to promoting creativity in order to solve programming problems, the necessary attention should also be given to the clean code principles.

CCS CONCEPTS

• **Software and its engineering** → *Software development techniques*; • **Social and professional topics** → **Software engineering education**;

KEYWORDS

clean code, code quality, creativity, software engineering education

ACM Reference Format:
Wouter Groeneveld, Dries Martin, Tibo Poncelet, and Kris Aerts. 2018. Are Undergraduate Creative Coders Clean Coders? A Correlation Study. In *The 53rd ACM Technical Symposium on Computer Science Education (SIGCSE'22), March 2–5, 2022, Providence, Rhode Island, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

SIGCSE '22, March 2–5, 2022, Providence, Rhode Island, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Programming is only a small part of the complete software development life cycle, that also includes (re)design and maintenance [14]. In order for code to be easily maintained, and thus modified, code should be *clean*. In other words: the code quality should be high, which facilitates rapid feedback cycles in the development life cycle.

Furthermore, in order to succeed as a software developer in the industry, it is no longer enough to excel at programming or writing clean code. A recent Delphi study reveals that experts also rate *creativity* as an important skill [17]. Developers must use their creativity to solve a difficult programming problem - and perhaps also to write certain legacy code as clean as possible.

Students who are learning to program in CS1 courses are typically taught about syntax, control structures, compiling, testing, debugging, basic object-oriented design, and so forth. In a syllabi analysis report, Becker and Fitzpatrick summarized what exactly educators expect of introductory programming students, based on the learning outcomes of the CS1 courses [10]. Neither clean code nor creativity was explicitly mentioned, although "problem solving" as part of introductory computational thinking appeared in the list of most frequent syllabus concepts. The absence of teaching clean code practices is further confirmed by Keuning et al. [22].

At our local faculty, we give the necessary attention to the clean code principles in our CS1 course, such as stressing the importance of the readability of code. However, we feel that the teaching of these concepts can be further improved. Students are required to finish a programming project, as elaborated in Section 3.1. This project is an open assignment which facilitates the creativity of students, as also advocated by Brookes [12]. We suspected that the concepts of clean code and creativity could be linked. Specifically, we wondered the following:

- **RQ1**: *Is there a relationship between creativity and code quality in CS1 student projects?*
- **RQ2**: *Are certain code quality aspects more related to creativity than others?*

By answering our research questions, we can better direct our CS1 teaching focus, thereby potentially increasing both students' creativity and their awareness of clean code principles.

The remainder of this paper is structured as follows. Section 2 explores the concepts of clean code and creativity further, and touches on related work. In Section 3, we explain our employed

methodology. Next, Section 4 describes and discusses the results. The limitations of this study are described in Section 5. Finally, in Section 6, we conclude this work, and forecast on possible future work.

2 BACKGROUND

In a 2018 paper on the evaluation of computational creativity systems, Jordanous hints at a relation between quality and creativity:

> *These two concepts are highly interrelated, to the point that it is difficult (and perhaps inappropriate) to define creativity without incorporating quality judgements into that definition [of creativity] [19].*

The author further mentions that many creative projects aim to produce high quality results, as can perhaps be expected. Bardzell et al. related crafting quality in design to integrity, creativity, and public sensibility by conducting interviews with elite craft practitioners [7]. Unfortunately, none of the craftsmen were software craftsmen, and creativity itself is only mentioned in passing. Within the field of computing education, we have not found any published work that investigates the potential correlation between the two concepts in context of student projects. We did find reports on the code quality of students [11, 22]. However, creativity was not mentioned. The same is true for reports on the creativity of computing students that do not mention code quality [12, 28].

In order to get a better understanding of both concepts that might be related, we first provide an overview of what exactly clean code and creativity is, according to the current literature.

2.1 What is Clean Code?

Software, and therefore its source code, is in a constant state of change. New requirements result in pieces of code that are changed, refactored, and redesigned. Learning to program is one thing, but learning how to write code that is easily maintainable is something else. Robert C. Martin offers in the first chapter of his book 'Clean Code' several definitions of clean code from different well-known software experts [23]. Many definitions are overlapping, of which the following concepts are the common divisor: (1) code that is easy to understand, (2) code that is easy to modify, and (3) code that is easy to test. Kent Beck proposed a fourth concept in The Agile Manifesto [9]: Code that works correctly.

These concepts are even more important when working in teams on a shared code base. Since the clean code principles are prevalent in the software engineering industry, we argue that when introducing students to programming in CS1 courses, the necessary attention should be given to these simple but important rules. This can be done by gradually instructing students to write code in such a way that it is easy to understand and modify, as Vihavainen et al. did in an apprenticeship-based CS1 course [33].

Keuning et al. say that so far, little attention has been paid to clean code issues in student programs. They found that novice programmers develop programs with a substantial amount of code quality issues, especially issues related to modularization. Keuning et al. use the term *"code quality"* to indicate the degree of cleanliness in students' source code. They cite Fowler's Refactoring book [15] to provide an example of low quality in code, such as the code duplication "code smell". In our view, clean code, code quality, and code smells are all closely related. In this paper, we consider a *clean coder* to be someone who is able to keep the code quality high by adhering to Martin's clean code principles - and thus also implicitly lowering the amount of Fowler's code smells. After all, code with a lot of duplication and other types of smell is not easy to understand, modify, or test.

2.2 What is Creativity?

Even though cognitive creativity and creative behavior are well-researched in the field of psychology [24, 30], it has proven difficult to create a uniform definition of creativity that can be applied to the field of computing education. Creativity is often defined as creating something that is both *novel* (to the creator), of *high quality*, and *usable* (appropriate to the task at hand) [21]. This definition already hints at the quality of the creative output, even though it cannot be simply ported to computing, since there is still an on-going debate whether or not creativity is domain-general or domain-specific [5]. Furthermore, in context of software development, does "high quality" equal a high build quality? And if it does, is it the same as code quality? The quality of software can be judged both from the outside, by looking at the product as a whole, as well as from the inside, by looking at the code.

Many published papers on measuring creativity in the SIGCSE community are limited to the constructs of divergent thinking [16, 28]. To make matters worse, creativity seems to exist in different contexts, where different kinds of creativity seem to manifest, according to Veale et al. [32]: creativity from within yourself, creativity in teams, and creativity on socio-organizational levels. Piffer argues that the notion of creativity can only be clarified after deciding how to measure it [25]. Therefore, we consider a *creative coder* to be someone who is able to program a creative piece of software, judged by evaluating the end product, which in this research is the finished CS1 student project. We elaborate on how the creativity of a project was measured in Section 3.3.

3 METHODOLOGY

In this section, we elaborate our applied method used to assess both the creativity and the code quality of student projects. First, in order to help the reader interpret our results, we overview our CS1 course where the study took place.

3.1 The CS1 Setting

The projects that were analyzed are part of a large-class CS1 course offered at our university (about 200 students yearly). The course has been designed for first-year engineering students. It is important to note that this course is part of a general engineering curriculum. Therefore, students who do not wish to specialize in computing, but like to pursue a chemical engineering degree instead, still have to pass the course. At our university, all courses of first-year engineering students are shared. This has two big ramifications. First, it implies that the course does not require prior knowledge of programming. Second, it means that the student group is very diverse, from students who would like to specialize in computing to students who prefer other engineering fields, such as chemistry.

The goal of the CS1 course is twofold. First, to introduce all engineering students to the basic concepts of object-oriented programming. Second, to lay the foundations for subsequent deeper learning of software engineering related courses offered to second-year students specializing in computing.

The course employs Java to explain the basics of program structures. During the first part of the course, BlueJ is used to introduce objects and relationships between them [8]. Typical topics of similar CS1 courses are covered, such as loops, variables, functions, iteration, but also classes and inheritance, as well as some less common ones such as exception handling, the very basics of threading, and the Model-View-Controller (MVC) pattern. During the second part, the NetBeans IDE is introduced, together with basic GUI programming using JavaFX and SceneBuilder. We find that by introducing something tangible such as JavaFX, students who do not enjoy programming or have no wish to further pursue computing still enjoy creating visible components. This allows the students to create little games as part of their project.

Four ECTS credits [13] of the course are granted if students pass both their theoretical exam and their project. The project, which is an open programming assignment, accommodates for 33% of the total grade and thus is major part of the course. Students are free to form groups of at most three people. We encourage them to work together, but it is not mandatory to do so. The assignment itself is completely open. Open assignments are known to spark the creativity of students [3] and to increase their motivation [2]. Students propose a subject themselves, which is then accepted or revised based on the complexity of their ideas.

This paper analyzes the projects of students from academic years 2019-2020 (217 students) and 2020-2021 (197 students). In both years, the course was taught by the first and last author. In total, 134 projects were processed, of which 110 were included in the study. The remaining 24 projects did not compile or bootstrap correctly, which made it impossible to judge the creativity, and therefore excluded from the study.

3.2 Evaluating Clean Code

To evaluate the code quality of the projects, we adapted the methodology from Keuning et al. [22], who used PMD [4], a well-known static analysis tool that is able to detect a large set of bad coding practices in Java programs. PMD allows you to define *rules*, which are bad coding practices, that are detected and reported in a `CSV` format. In this paper, we define an *issue* as a problem detected by PMD in relation to a specific rule. Keuning and colleagues categorized their utilized PMD rules according to Stegeman et al.'s developed rubric for assessing code quality [31]. We used the same five categories to subdivide our own rules:

(1) **Flow**; e.g. issues with nesting, paths and unreachable code.
(2) **Idiom**; e.g. unsuitable choice of control structures, no reuse.
(3) **Expressions**; e.g. overly complex expressions.
(4) **Decomposition**; e.g. too large methods, duplicate code.
(5) **Modularization**; e.g. too many methods, tight coupling.

PMD comes equipped with more than a hundred different rules. Therefore, we needed to select rules that were deemed relevant to both our study and our CS1 course. To aid with the decision, we randomly ran all 110 anonymized projects against both Keuning's PMD ruleset and the built-in 'quickstart' PMD ruleset, which contains common best practice rules. First, all 58 rules were applied to the projects. Next, for each rule, we calculated the percentage of the total occurrence and the percentage of presence in the projects.

We discarded rules using the following criteria:

- A rule found in fewer than 5% of the projects (8).
- A rule found in fewer than 0.10% of the total errors (4).
- A rule related to documentation, presentation, imports or name conventions (e.g. `LocalVarNamingConventions`) (12).
- A rule marked as deprecated or controversial in the PMD documentation (e.g. `UnnecessaryConstructor`) (3).
- A rule deemed not relevant for our CS1 course (e.g. `Unnecessary Modifier` or `AvoidInstantiatingObjectsInLoops`) (5).

The final ruleset, containing 26 rules, categorized according to Stegeman's rubric, is visible in Table 1. All other rule settings have been left on their default settings, as per recommendation in [22].

Table 1: Selected PMD rules per category. Rules marked in italics also occur in the top 10 list of most frequently found issues, in Table 3.

Flow (6)	
CyclomaticComplexity	The number of decision points.
NPathComplexity	The number of acyclic exec paths.
PrematureDeclaration	
EmptyIfStmt	
EmptyCatchBlock	
ForLoopCanBeForeach	Prefer Java's foreach syntax.
Idiom (8)	
UnusedPrivateField	
SwitchStmtsShouldHaveDefault	
UnusedLocalVariable	
UselessParentheses	
CloseResource	Ensure closure of connections.
ControlStatementBraces	Enforce braces for control stmts.
UnnecessaryLocalBeforeReturn	Creation of unneeded variables.
UnusedFormalParameter	
Expressions (5)	
SimplifyBooleanExpressions	
CollapsibleIfStatements	Consolidate if statements.
SimplifyBooleanReturns	
ConfusingTernary	Negation problems in if/else clauses.
AvoidReassigningParameters	
Decomposition (3)	
SingularField	Fields with limited scopes.
NcssCount (Lines of Code)	Methods (60) and Classes (1500)
CPD-50	Copy-Paste Detector for 50 tokens
Modularization (4)	
LawOfDemeter	Low coupling: "Only talk to friends".
TooManyFields	
TooManyMethods	
GodClass	A class that does too many things.

For each project, PMD collected all issues regarding our selected 26 rules. Then, also for each project, we calculated the total amount of issues and the total amount of unique issues reported by PMD. For instance, if a project contains three unused private fields and two God classes, the total amount of issues is 5, while the amount of

unique issues is 2 (`UnusedPrivateField` and `GodClass`). Irrelevant information, such as the the specific location of each issue, was discarded.

To cross-validate whether the selected PMD rules effectively measure the code quality, half of the projects from academic year 2019-2020 were randomly assessed manually by inspecting the anonymized source code. A score bewteen 1 and 10 was assigned by the first author that indicates the *code quality rate* of these projects. The first author has more than a decade of experience in the software engineering industry and has guided many software development teams on how to write clean code. We found a strong negative correlation between the manual assessment and the amount of total PMD issues for each project ($r = -0.70$, $p = 0.00$) and a moderate negative correlation for the total number of unique issues ($r = -0.54$, $p = 0.00$). This signifies that the 26 PMD metrics reported in this paper are solid ways to evaluate the global code quality of all student projects.

3.3 Evaluating creativity

As mentioned in Section 2.2, there is no uniform definition of creativity. Therefore, evaluating the creativity of student projects can be problematic. Rhodes' 4P Creativity Model [26] defines four different dimensions that influence creativity: *Person*, *Press* (or Place), *Product*, and *Process*. Each dimension can be evaluated in its own way. For instance, to assess the creativity of a *person* (the student), one can rely on different personality tests. For this study, we are interested in the assessment of students' CS1 programming projects, which are tangible outcomes of their creative process, and thus are part of the *Product* dimension.

In the field of cognitive psychology, Amabile's Consensual Assessment Technique [1] (CAT) is a common evaluation technique to assess the creativity of a product [6]. This method relies on the opinions of expert judges that give a score between 1 and 10 for each product in a pool. Unlike many other techniques for creativity assessment, the CAT technique is not tied to any particular theory of creativity. Since the CAT scoring process is highly subjective, to reconcile differences, it is recommended to form a panel of expert judges, and to calculate the average of the given scores [6]. Even though assembling a group of expert judges is very resource-insensitive, replacing all experts with novice judges, such as students, is generally not recommended [20].

For this study, we recruited both novice and expert judges. Two members of the CS1 teaching staff acted as expert judges, while five second-year students acted as junior judges. After explaining the design of this research and the CAT concept, each judge blindly evaluated the creativity of every anonymized project. After spending exactly one minute for each project, a score was administered individually, without looking at the source code. We refrained from providing a definition of creativity, as per recommendation in [6]. The novice judges highly enjoyed evaluating projects of their fellow students. More than a few times, they made the remark *"I wish I spent more time working on my own project, now that I see all the others"*. Engaging students in grading is known to increase their motivation [18].

The standard deviation (*SD*) of the scores between the judges, averaging on 1.06, was low enough for us to conclude the same as Baer and McKool: judges score surprisingly similar [6]. Therefore, an inter-rater reliability is not relevant when using the CAT technique. However, as an extra verification step, when the judges reached a significant disagreement ($SD > 1.50$, 10 out of 110 projects), the project concerned was discussed in group, after which a new global score was assigned. The projects with the largest *SD* were suspected of plagiarism by a few judges, which resulted in a low score.

3.4 Finding relationships

According to a very recent systematic literature review on inferential statistics in computing education research, one of the most frequently used tests for inferring a relationship between ordinal and interval level data is the Pearson product-moment correlation [29]. Since the creativity scores are ordinal (arranged on a scale form 1 to 10) and the PMD metrics can be considered as interval level data (a continuous scale), we chose to answer our research questions by using this recommended correlation technique.

4 RESULTS AND DISCUSSION

We will first discuss creativity and code quality separately, before investigating potential relationships. The full data set of the 110 processed projects in both academic years can be consulted at https://people.cs.kuleuven.be/~wouter.groeneveld/correl/.

4.1 Projects and creativity

On average, the judges rated the creativity of the projects at 5.92 out of 10, with a global *SD* of 1.46. The normal distribution of the CAT scores is visible in Figure 1. The average is 13 out of 20, with a global *SD* of 3.37. There is a moderate correlation between the creativity score and the total grade of the project ($r = 0.46$, $p = 0.00$). This is to be expected, as creativity is one of the criteria in the evaluation rubric of the CS1 projects, next to others such as complexity and JavaFX UI design.

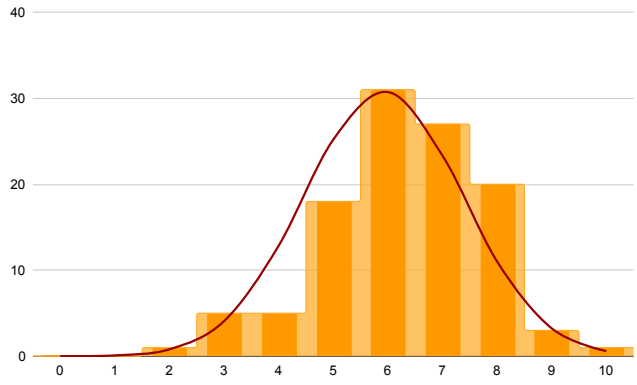


Figure 1: The normal distribution of the creativity CAT scores (x-axis) for all projects (y-axis).

We categorized each submitted project by topic. Since the assignment itself was open, it is interesting to see that many, but not all, students resorted to implementing popular small games. The topics, along their respective average scores, are visible in Table 2. Projects

that did not fit in any other category were placed in the "original" category, which has the second highest average CAT rating. Examples of simple boardgames we found were the Game of the Goose and Monopoly. Examples of simple dicegames we encountered were Yahtzee and higher/lower. We considered some projects to be more original than others, such as the Pokémon-inspired random battle games. However, since they were based on an existing game, and occurred more than once, we decided against categorizing them as "original". Furthermore, it is also interesting to note that original ideas do not necessarily mean high grades. For instance, students who played it safe and implemented a solid version of chess or checkers scored in total on average 15 out of 20, while completely original projects scored on average 12.18. This is because next to creativity, complexity is also a component of the project grade.

Table 2: Identified project topics and their average CAT scores and total grades.

Topic	#	%	CAT	Grade
original	**20**	**18.18%**	**7.09**	**12.18**
doodle jump	16	14.55%	4.79	11.75
space shooter	15	13.64%	5.40	11.87
race/frogger	8	7.27%	6.51	13.94
arkanoid	7	6.36%	5.87	14.93
tic-tac-toe/4-in-a-row	7	6.36%	4.58	10.00
chess/checkers	6	5.45%	5.70	15.00
snake	6	5.45%	4.96	12.67
simple boardgame	6	5.45%	5.61	12.25
pacman	3	2.73%	5.46	13.67
bubble shooter	3	2.73%	6.57	16.67
simple dicegame	3	2.73%	4.36	9.67
poké-battle	2	1.82%	6.95	10.00
2048 game	2	1.82%	7.65	15.00
mastermind	2	1.82%	6.47	12.00
hang man	2	1.82%	6.14	9.50
tetris	2	1.82%	6.35	13.50
minesweeper	1	0.91%	4.67	14.00
Total	110			

4.2 Projects and their code quality issues

Table 3 contains a list of the top 10 most frequently found code quality issues by PMD. These issues are marked in italics in Table 1. We also counted the number of issues that frequently occurred in the same projects. First, the quotient of total issues in relation to unique issues found in each project was calculated. Next, for each project with a quotient higher than 10, issues that occur 15 times or more are counted. We found 11 projects with such a high quotient. In total, 7363 issues have been found in 104,456 lines of code. The average amount of issues per project is 64, and each project contains on average 836 lines of code. We counted 10 unique issues on average per project.

When inspecting the top 10 PMD issues of the millions of student projects analyzed by Keuning et al., we see that `LawOfDemeter` also occurs often: it is the third most common issue [22]. It is interesting to note that coupling-related issues are commonplace. In addition, `LawOfDemeter` is also the issue that is the most recurring within a single project.

Table 3: The most frequently found code quality issues, including project counts with a high number of recurring issues, and respective correlations with the average CAT creativity score.

Issue	Category	#	Recur?	Corr.
LawOfDemeter	Modularization	4113	11	0.19
SimplifyBooleanExpressions	Expressions	646	4	0.26
UnusedPrivateField	Idiom	491	0	0.12
CPD-50	Decomposition	448	2	0.27
UselessParentheses	Idiom	411	0	0.09
SingularField	Decomposition	359	0	0.13
ControlStatementBraces	Idiom	327	2	-0.11
CyclomaticComplexity	Flow	242	1	0.36
SwitchStmtsShldHvDefault	Idiom	170	0	0.28
CollapsibleIfStatements	Expressions	156	2	0.02
Total		7363		

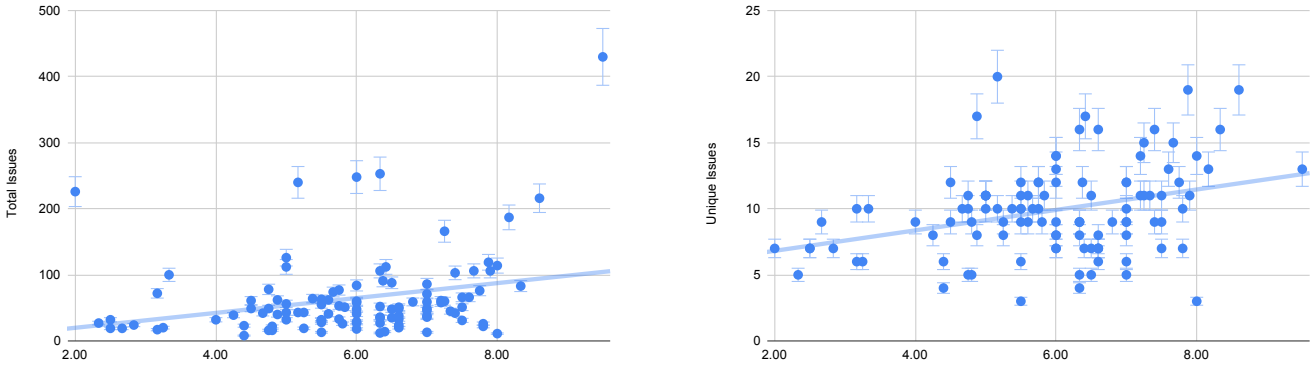
4.3 RQ1: Relating creativity to code quality

We found almost moderate positive correlations between lines of code and evaluated creativity ($r = 0.38$, $p = 0.00$), between unique code quality issues and evaluated creativity ($r = 0.33$, $p = 0.00$), and between total code quality issues and evaluated creativity ($r = 0.27$, $p = 0.00$). Although the correlations are not strong, they do seem to suggest that more creative projects contain more code quality issues. Figure 2 displays scatter plots of the CAT scores projected on the unique issues and total issues, including a linear trend line. If we remove the edge cases with high error bars from Figure 2a, where the total issues of the projects exceeds 200, the correlation between the total code quality issues and evaluated creativity increases further from 0.27 to 0.34. The results of this paper were informally shared among second-year students (who completed the CS1 project in 2019-2020) as part of an introduction into academic research. When asked for a possible explanation for the correlations, the students suggested two distinct possibilities.

First, it could be that students who decide to take on more original projects also experiment more. A consequence of experimentation during the development is encountering unforeseen bugs which students were not able to quickly resolve due to lack of experience. As a result, several code quality issues may have been overlooked while working as fast as possible to make up for the time lost looking for the bugs. This is in line with existing research that shows that first-year students are not yet competent enough to accurately plan and estimate software projects [34]. A lack of decent planning could be more damaging for creative projects, as our results indicate that the higher the creativity, the higher the complexity. This is further elaborated in Section 4.4.

Second, during the academic year, students have to complete multiple projects of multiple courses, not limited to computing. This rapid context switching potentially causes a heavy fragmentation of the students' time. Spending little fragments of time on a software project could be harmful to the code quality, compared to working on a piece of software without interruptions or distractions [27].

The above reasons have merit but are only informal remarks of the students themselves, although supported by literature. We plan to continue the investigations of the rationale in future work.



(a) Compared to total issues found

(b) Compared to unique issues found

Figure 2: Scatter plots of evaluated creativity CAT scores compared to code quality issues.

4.4 RQ2: Are certain issues more relevant?

The top 10 found PMD issues listed in Table 3 reveals that some issues are more correlated to the CAT scores than others. Yet the possible interpretations of these numbers are dubious. For instance, why would `SwitchStmtsShouldHaveDefault` be more related to creativity than `LawOfDemeter`? Not every issue appears as often as others: students seem to make more mistakes against good coupling practices than to forget the `default` statement in a switch block, although we did not count the total amount of switch blocks in a project. We argue that some of the categorizes defined by Stegeman et al., as explained in Section 3.2, can have more severe consequences for Robert C. Martin's clean code concepts than others. Missing a `default` case would be less devastating for long-term code maintainability than violating the Law of Demeter.

Most issues show no moderate correlation with creativity, although `CyclomaticComplexity` ($r = 0.36$) nears 0.40. A cyclomatic complex project is a project where the code contains too many decision points in certain methods to be easily readable. In practice, this is usually visible by nesting multiple `if`, `for`, `while`, and `case` statements. There is an internal strong positive correlation between cyclomatic complexity and the issues `CPD-50` (Decomposition, $r = 0.67$), `LawOfDemeter` (Modularization, $r = 0.57$), and `NPathComplexity` (Flow, $r = 0.51$). Since the CS1 project assignments were open, we argue that some students challenged themselves by picking a complex problem (e.g. a 2D platformer with gravity and enemies) instead of a simple one (e.g. a tic-tac-toe game). Their risk got rewarded by receiving higher creative scores from the judges. However, since these first-year students are still novice coders, a complex problem introduces many messy complexities in their code, hence the higher amount of aforementioned PMD issues, more duplication, and thus generally more lines of code.

5 LIMITATIONS

This study analyzes CS1 projects of two academic years. Having access to more data would potentially alter the normal distribution of Figure 1 or the correlations of Figure 2. Nonetheless, we think that the results of the 110 projects are significant enough to warrant the results described in Section 4. Another possible threat to validity could be the interpretation of the judges during the CAT phase. Our aim was to employ CAT to evaluate creativity and to employ PMD to evaluate quality. Jordanous warned about the potential confusion between creativity evaluation and quality judgments, leading to less grounded evaluative results [19]. We believe that by cross-validating and by using multiple judges, as explained in Section 3.3, the effects of potential confusions have been mitigated.

An additional shortcoming could be focusing on the wrong PMD issues, thereby correlating creativity with irrelevant code quality issues. However, as we based our method on Keuning's work, and took great care in selecting the relevant PMD issues for our CS1 setting and based on the cross-validation, as explained in Section 3.2, we are fairly certain that this eliminated the possible inclusion of irrelevant PMD issues. Another risk could be that certain excluded rules might be more related to creativity. Rejected rules, according to the criteria from Section 3.2, do not accurately represent the students' code quality, and therefore, are not deemed relevant.

6 CONCLUSION AND FUTURE WORK

We have analyzed 110 projects from two academic years to investigate whether there is a relation between creativity and clean code in CS1 student projects. Our results show preliminary evidence that the more creative projects are, the more code quality issues arise in these projects and the less clean the submitted code turns out to be. This potentially creates a new problem: the more we as educators encourage creativity, the more messy the produced code can be. Thus, it is vital that the necessary attention to the clean code principles is given. We argue that this should become more important as we let students experiment and be creative. This is in line with Keuning et al's conclusion [22].

If we want to increase the creativity of students' work, we need to pay attention to the quality as well. Future work might shed more light on the teaching of clean code in relation to creativity. For example, we do not yet know what the effects of explicitly teaching creativity are on the code quality. Most studies, like this paper, limit the implementation of creativity in a CS1 course to the use of open assignments. Also, we wonder whether the observed effect would be reversed for last-year computing students, as they should be more disciplined in keeping their code quality high.

ACKNOWLEDGMENTS

We would like to thank both junior and senior creativity judges for their participation and interest in our mini research lab that evolved into this paper.

REFERENCES

[1] Teresa M Amabile. 1982. Social psychology of creativity: A consensual assessment technique. *Journal of personality and social psychology* 43, 5 (1982), 997.

[2] Mikko Apiola, Matti Lattu, and Tomi A. Pasanen. 2010. Creativity and Intrinsic Motivation in Computer Science Education: Experimenting with Robots. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (Bilkent, Ankara, Turkey) *(ITiCSE '10)*. Association for Computing Machinery, New York, NY, USA, 199–203. https://doi.org/10.1145/1822090.1822147

[3] Mikko Apiola, Matti Lattu, and Tomi A Pasanen. 2012. Creativity-Supporting Learning Environment—CSLE. *ACM Transactions on Computing Education (TOCE)* 12, 3 (2012), 1–25.

[4] PMD Authors. 2021. PMD. https://pmd.github.io/.

[5] John Baer. 2010. Is creativity domain specific. *The Cambridge handbook of creativity* 321 (2010).

[6] John Baer and Sharon S McKool. 2009. Assessing creativity using the consensual assessment technique. In *Handbook of research on assessment technologies, methods, and applications in higher education*. IGI Global, 65–77.

[7] Shaowen Bardzell, Daniela K Rosner, and Jeffrey Bardzell. 2012. Crafting quality in design: integrity, creativity, and public sensibility. In *Proceedings of the designing interactive systems conference*. 11–20.

[8] David John Barnes, Michael Kölling, and James Gosling. 2006. *Objects First with Java: A practical introduction using BlueJ*. Pearson Prentice Hall London.

[9] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development. (2001).

[10] Brett A. Becker and Thomas Fitzpatrick. 2019. What Do CS1 Syllabi Reveal About Our Expectations of Introductory Programming Students?. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1011–1017. https://doi.org/10.1145/3287324.3287485

[11] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. " I know it when I see it" Perceptions of Code Quality: ITiCSE'17 Working Group Report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*. 70–85.

[12] Wayne Brookes. 2018. On creativity and innovation in the computing curriculum. In *Proceedings of the 20th Australasian Computing Education Conference*. 17–24.

[13] European Commission. 2015. European Credit Transfer andAccumulation System (ECTS). http://ec.europa.eu/education/ects/ectsen.htm.

[14] Kagan Erdil, Emily Finn, Kevin Keating, Jay Meattle, Sunyoung Park, and Deborah Yoon. 2003. Software maintenance as part of the software life cycle. *Comp180: Software Engineering Project* (2003), 1–49.

[15] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[16] David Ginat. 2008. Learning from Wrong and Creative Algorithm Design. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 26–30. https://doi.org/10.1145/1352135.1352148

[17] Wouter Groeneveld, Hans Jacobs, Joost Vennekens, and Kris Aerts. 2020. Non-cognitive abilities of exceptional software engineers: a Delphi study. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 1096–1102.

[18] Wouter Groeneveld, Joost Vennekens, and Kris Aerts. 2020. Engaging Software Engineering Students in Grading: The effects of peer assessment on self-evaluation, motivation, and study time. *arXiv preprint arXiv:2012.03521* (2020).

[19] Anna Jordanous. 2018. Creativity vs quality: why the distinction matters when evaluating computational creativity systems. In *Proceedings of The 5th Computational Creativity Symposium at the AISB Convention*. AISB.

[20] James C Kaufman, Claudia A Gentile, and John Baer. 2005. Do gifted student writers and creative writing experts rate creativity the same way? *Gifted Child Quarterly* 49, 3 (2005), 260–265.

[21] James C Kaufman and Robert J Sternberg. 2007. Creativity. *Change: The Magazine of Higher Learning* 39, 4 (2007), 55–60.

[22] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 110–115.

[23] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[24] ED Petkus Jr. 1996. The creative identity: Creative behavior from the symbolic interactionist perspective. *The Journal of Creative Behavior* 30, 3 (1996), 188–196.

[25] Davide Piffer. 2012. Can creativity be measured? An attempt to clarify the notion of creativity and general directions for future research. *Thinking Skills and Creativity* 7, 3 (2012), 258–264.

[26] Mel Rhodes. 1961. An analysis of creativity. *The Phi Delta Kappan* 42, 7 (1961), 305–310.

[27] Pilar Rodríguez, Jari Partanen, Pasi Kuvaja, and Markku Oivo. 2014. Combining lean thinking and agile methods for software development: A case study of a finnish provider of wireless embedded systems detailed. In *2014 47th Hawaii International Conference on System Sciences*. IEEE, 4770–4779.

[28] Andrea Salgian, Teresa M Nakra, Christopher Ault, and Yunfeng Wang. 2013. Teaching creativity in computer science. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 123–128.

[29] Kate Sanders, Judy Sheard, Brett A Becker, Anna Eckerdal, and Sally Hamouda. 2019. Inferential Statistics in Computing Education Research: A Methodological Review. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 177–185.

[30] Dean Keith Simonton. 2000. Creativity: Cognitive, personal, developmental, and social aspects. *American psychologist* 55, 1 (2000), 151.

[31] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. 160–164.

[32] Tony Veale, Pablo Gervás, and Alison Pease. 2006. Understanding creativity: A computational perspective. *New Generation Computing* 24, 3 (2006), 203–207.

[33] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. 2013. Massive increase in eager TAs: Experiences from extreme apprenticeship-based CS1. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 123–128.

[34] Timothy T Yuen. 2007. Novices' knowledge construction of difficult concepts in CS1. *ACM SIGCSE Bulletin* 39, 4 (2007), 49–53.