Human vs. Automated Coding Style Grading in Computing Education

James Perretta, Westley Weimer, and Andrew DeOrio

1 Abstract

Computer programming courses often evaluate student coding style by hand. Static analysis tools provide an opportunity to automate this process. In this paper, we explore the effectiveness of human style graders and general-purpose static analysis tools for evaluating specific style-grading criteria.

We analyze data from a second-semester programming course at a large research institution with 943 students enrolled. Hired student graders evaluated student code with rubric criteria such as "Lines are not too long" or "Code is not too deeply nested." We also ran several static analysis tools on the same student code to evaluate the same criteria. We then analyzed the correlation between the number of static analysis warnings and human style grading score for each criterion.

Our initial investigation reveals that human graders do not reliably provide consistent style grading scores when compared against each other. We also see that human graders perform inconsistently on objective style grading criteria when compared against static analysis inspections used to evaluate those same criteria. While we note that existing, general-purpose static analysis tools are insufficient to evaluate all of the style criteria from the course, we identify several objective criteria for which static analysis tools perform more accurately than human graders. The static analysis inspections for these criteria can be easily implemented for common programming languages. In conjunction with an automated grading system, these tools can be used to reduce the effort spent by human graders and provide better, more immediate feedback to students on the quality of their code.

2 Introduction

Research and experience from industry have demonstrated that code review and following good coding practices are important parts of writing maintainable software[1]. Consequently, good coding style is an important learning goal of computer science courses. This is often achieved by evaluating student code by hand using a set of style criteria. This process is difficult to scale for large courses. In particular, having more submissions to grade increases the time it takes for students to receive feedback on their work. Static analysis tools offer a possible solution to this problem. Our goal is to determine which style grading criteria can be effectively automated by existing static analysis tools.

2.1 Code Style

In industry, software companies often have a formal set of coding standards dictating preferred practices for code style and quality[2,3]. They may include guidelines for code formatting and naming conventions in order to provide consistency across projects. They may also specify features of a particular language that are discouraged or forbidden from use. These conventions are usually enforced through a combination of static analysis tools and human code review[4,5]. In computer science courses, the coding standards used to evaluate student code are likely designed based on the experience level of students in the course. For example, an introductory programming course's coding standards might focus on things like properly indenting source code and using descriptive variable names.

2.2 Static Analysis Tools for Evaluating Code Style

A significant amount of effort has gone into writing tools that analyze coding style. As early as the 1990s, computer programming courses used static analysis tools to enforce some simple coding style rules for students' Pascal code[6,7]. Some modern tools are designed to enforce a specific set of coding standards. For example, cpplint[5] was written to enforce Google's coding standards, and pycodestyle[8] was written to enforce PEP 8, a section of the Python language standard that describes style conventions[9]. Similarly, checkstyle[10] can be used to enforce coding standards in Java code. The Clang compiler frontend has helped give rise to other configurable, extendable tools such as Clang-Tidy[11] and OCLint[12] that perform various inspections for C-family languages. Other tools, such as PMD[13] and cppcheck[14] detect potential bugs and undesirable coding practices in Java and C++ code, respectively.

2.3 Style Grading

In computer programming courses, the desired qualities for a style grading process differ from those for code review in industry. Unlike in industry where code review is often a prerequisite for merging a patch[15], students usually are not required to revisit their code from prior programming projects. This means that students have limited opportunities to respond to any feedback received after a project is due. Furthermore, the fact that style grading affects students' course grades requires additional transparency and precision in the process. Based on the published literature and our experience, we identify three desirable qualities in a style grading process.

Accuracy: A style grading process should be accurate and free from false positives, otherwise students will be assigned the wrong grade for an assignment. Researchers across educational fields have investigated methods of training teachers and graders[16,17] as well as protocols for assigning multiple graders to a single submission in order to improve consistency[18,19]. These practices, however, can be difficult to implement in courses where human resources are limited. With instructors and teaching assistants focused on leading lecture or lab sessions, there are often fewer hours left for grading than needed for double marking or other such approaches.

Clarity: It should be easy for students and instructors to determine why a particular style grade was assigned. Instructors must have confidence that students were given the right grade, and students must be able to learn from the feedback and make improvements on future assignments. Not all tools support such clarity. For example, prior work by Buse et al. used machine learning

to quickly and accurately assign a "readability" score to a piece of code[20][21]. Although their automated model performs more accurately, on average, than humans, the tool provides only a numerical score. It is not a normative or explanatory model, so it is difficult for users of the tool to figure out what specific changes they should make to their code to improve it. Unlike rule-based static analyses, any approach involving machine learning is likely to face similar issues, as explainability is an ongoing research issue in that field[22][23].

Speed: Students benefit from receiving timely feedback on their work, especially if they are able to address that feedback and resubmit their work. This is an important part of the philosophy behind using automated grading systems in computer programming courses. Prior work suggests that giving students frequent, actionable feedback on their work can help them develop good habits and improve specific skills, such as writing high-quality software tests[24][25].

2.4 Contributions

In this paper, we explore the strengths and weaknesses of evaluating student code style manually using human style graders and automatically using static analysis tools. In particular, we examine three research questions:

1. Do human graders provide style grading scores consistent with each other?
2. Are human coding style evaluation scores consistent with static analysis tools?
3. Which style grading criteria are more effectively evaluated with existing static analysis tools and which are more effectively evaluated by human graders?

3 Methods

Our goal in this study is to identify code inspections offered by general-purpose static analysis tools that consistently provide high-quality feedback about style grading criteria. Hired student graders evaluated student code according to a predetermined rubric. We also ran static analysis tools on the same student code to evaluate the same criteria. We analyzed the distributions of style grading scores awarded by each human grader. We also examined the correlation between the number of static analysis warnings and the human style grading scores.

3.1 CS2 Course

Our study examines a second-semester computer programming course at a large research institution with a total of 943 students in one semester. The course contained five programming projects where students wrote C++ code according to a specification. A typical programming project consisted of implementing one or more abstract data types (ADTs) according to specification and writing a command-line program using those ADTs.

Students in the course attended three hours of lecture per week and a two hour lab session each week. Lab sessions consisted of a short exercise worksheet and a programming activity designed to supplement material from lecture. Lecture and lab sections in the course were synchronized to ensure that students learned the same material regardless of which section they attended.

We collected data from one project where the command-line program portion was longer and

more open-ended than on any other project in the course. The instructor solution for this project was 595 lines of code, while the average length of student solutions was 857 lines of code. Students were allowed to work alone or with a partner, yielding 621 distinct assignment submissions. Students submitted their code to an automated grading system, where they received automated feedback on the correctness of their code up to three times per day. After the project deadline, student code was evaluated manually by hired student graders. These human graders were given a predetermined list of criteria with which to evaluate student code. Each human grader was assigned 42 submissions to grade over a period of 2 weeks.

3.2 Human Style Grading Rubric

We trained our human graders by providing them with written instructions on how to apply style grading criteria. Our human style grading criteria are intended to address common style errors as well as programming concepts that appear in certain projects. Each rubric item is evaluated on a 3-value scale with 2 meaning "Always", 1 meaning "Usually", and 0 meaning "Almost never". These criteria represent common style evaluation guidelines in introductory programming courses.

The general programming practice criteria, common to all assignments, are as follows:

- Helper functions are used where appropriate: Human style graders were instructed to deduct points if students wrote excessively long functions rather than splitting them up into smaller ones. When a student's command line program consisted entirely of one long function, human style graders were instructed to give zero points for this criterion.
- Lines are not too long: Human style graders were instructed to use 80 characters as a soft limit and to not deduct points if a line of code exceeded this limit by only a few characters.
- Functions and variables have descriptive names.
- Effective, consistent, and readable line indentation is used.
- Code is not too deeply nested in loops and conditionals.

Some examples of project-specific criteria include:

- Explicit use of the `this` keyword is avoided.
- The Big 3 C++ special member functions (copy constructor, assignment operator overload, and destructor) are only implemented when needed.
- Interfaces are respected in C-style ADTs.

3.3 Static Analysis Tools and Data Collection

We investigated several existing, open-source static analysis tools and selected ones that offered inspections that were closely related to one or more items in our human style grading rubric. We required that tools support C++ code, provide specific inspections with configurable thresholds, and be free to use. The inspections needed to be concerned with code style rather than program correctness. The output of the inspections also needed to be easy to parse to count the number of warnings. We chose CPD (part of the PMD tools) and three inspections offered by OCLint. We also considered, but did not select, Clang-Tidy[11], cppcheck[14], duplo[26], sonarsource[27], and cpplint[5]. Here we describe the inspections that we used to evaluate student code style.

OCLint LongLine: This inspection reports each line of code longer than a specified threshold. Based on the instructions in our rubric, we chose 90 characters as the threshold.

OCLint DeepNestedBlock: This inspection reports each block nested deeper than a specified threshold. We chose a threshold of 4 because the instructor solution only exceeded this level of nesting once.

OCLint HighNcssMethod: This inspection reports each function with more non-commenting source statements than a specified threshold. We chose a threshold of 30 statements and scaled the warning count so that warnings for longer functions had a higher weight.

CPD: Copy Paste Detector: This inspection reports exact sequences of duplicated code longer than a threshold of 100 tokens. We recorded the number of duplicated lines for each submission.

4 Results

We analyze data from 621 student code bases. We collected style grading scores assigned to each submission by a human grader and counted the number of warnings from automated static analysis tools. We compare these two sets of data.

4.1 Consistency Among Human Graders

We analyzed the distributions of scores assigned by 15 individual human graders to determine how consistently the graders perform when compared against each other. Each human grader was assigned a disjoint group of 42 submissions to grade. Using the Kruskal-Wallis test for one-way ANOVA, we see that 10 of the 15 human graders assigned scores consistently with each other, while the remaining 5 assigned scores inconsistently with those 10 graders.

Since each assignment was graded by only one human grader, we will first show that the populations of student submissions assigned to each grader are comparable. Our independent variable is the human grader who evaluated a group of submissions for style. Our dependent variable is the code correctness score (determined by automated test cases) of students' submissions. We used the Kruskal-Wallis test to compare the populations of student submissions. We fail to see a statistically significant difference between the median project correctness scores of the groups of submissions ($p=0.23$). Since all the groups of submissions had comparable code correctness scores, we expect these groups to also have comparable code style scores.

We now compare the style grading scores assigned by each human grader. Table 1 contains descriptive statistics of the scores assigned by each grader. Our independent variable is the human grader who evaluated a group of submissions for style. Our dependent variable is the style grading score assigned to the students' submissions. We used the Kruskal-Wallis test to compare the populations of student submissions assigned to each grader. We do not see a statistically significant difference between the median scores of graders 1-3 and 6-12 in Table 1 ($p=0.06$). In the remaining 5 groups of submissions (graders 4, 5, 13, 14, and 15 in Table 1), we see a statistically significant difference between the median style grading scores of each of those groups

	Human Grader #														
	1∗	2∗	3∗	4†	5†	6∗	7∗	8∗	9∗	10∗	11∗	12∗	13†	14†	15‡
Mean	20	20	20	20	19	20	20	21	20	20	20	21	20	20	17
Stdev	2.1	1.7	1.9	1.8	2.5	2.0	1.7	1.8	2.7	2.1	2.7	1.5	1.8	1.9	4.7
Median	21	20	20	20	20	21	21	21	22	21	21	21	20	19	18

Table 1: Descriptive statistics of scores awarded by human graders. Scores were out of 22 points. Graders marked with the same symbol (∗, †, or ‡) indicates that we do not see a statistically significant difference between their medians using the Kruskal-Wallis test with p-values greater than 0.05.

and the 10 aforementioned groups of submissions, with p-values less than 0.05. We discuss the implications of this human grader inconsistency in Section 5.1.

4.2 Static Analysis Output vs. Human Style Grading Scores

We now examine correlations between the number of static analysis warnings and the human style grading score for each rubric item. Since a high warning count indicates more style criterion violations, we expect to see a negative correlation between warnings and scores. In Table 2 we see a weak negative correlation between the number of warnings emitted by OCLint's LongLine inspection and the scores for the "Line length" rubric item. We also see a weak negative correlation between the number of warnings emitted by OCLint's DeepNestedBlock inspection and the scores for the "Nesting" rubric item. When comparing the "Helper functions" score against the number of warnings emitted by OCLint's HighNcssMethod inspection or the number of lines of duplicated code reported by CPD, however, we see almost no correlation.

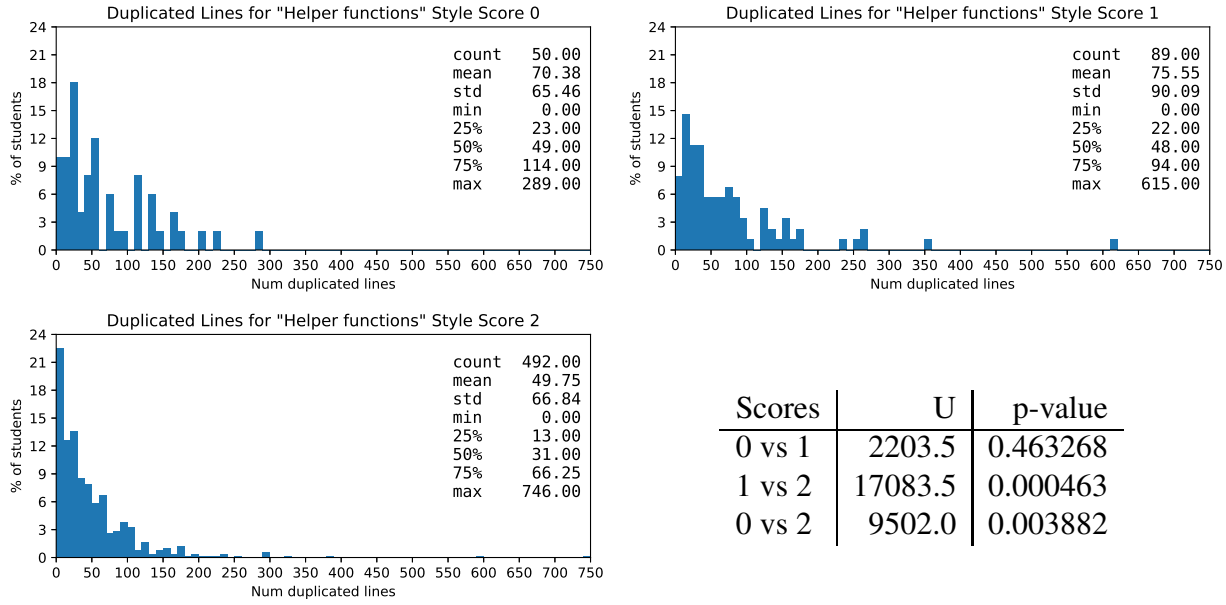
Style Rubric	Static Analysis	Pearson r
Line Length	OCLint LongLine	-0.22
Nesting	OCLint DeepNestedBlock	-0.21
Helper Functions	OCLint HighNcssMethod	-0.07
Helper Functions	Copy/Paste Detector	-0.12

Table 2: Pearson correlations between human style grading scores and number of static analysis warnings.

4.2.1 Distributions of Static Analysis Warnings

Next, we examine the distributions of the number of static analysis warnings emitted for each inspection, stratified by the human style grading score awarded. We expect a statistically significant difference between each of these strata, e.g. the number of warnings for submissions that received a 0 or a 1 should be different from that of submissions that received a 1 or a 2.

We consider each of the four analyses in turn, starting with the CPD tool and the "Helper functions" rubric. In Figure 1, we see a statistically significant difference between the mean number of duplicated lines of code reported by CPD for students who received a "Helper functions" style score of 1 or 2. Students who received a 1 had on average about 29 more lines of duplicated code than students who received a 2. We fail to see a statistically significant difference for students who received a 0 or a 1. We see that only about 10% of students who received a 0 and about 8% of students who received a 1 had no detected duplicated lines of code. About 13% of students who received a 2 had at least 100 duplicated lines of code. We also note the presence of several outliers who received a 2 despite having 385, 590, and 746 lines of duplicated code, respectively.



Scores	U	p-value
0 vs 1	2203.5	0.463268
1 vs 2	17083.5	0.000463
0 vs 2	9502.0	0.003882

Figure 1: Distributions and ANOVA for number of duplicated lines of code reported by CPD, stratified by "Helper functions" style grading score. We see a statistically significant difference in the number of lines of duplicated code between students who received a 1 or a 2, but not between students who received a 0 or a 1.

Similarly, we see a statistically significant difference between the mean number of OCLint LongLine warnings for students who received a "Line length" style score of 1 or 2 (p=9.7e-10, U=20245). Students who received a 1 had on average about 7 more warnings than students who received a 2. We do not see a statistically significant difference between the mean number of warnings for students who received a score of 0 or 1 (p=0.34, U=2565). We also see that about 15% of students who received a 1 and about 23% of students who received a 0 had no LongLine warnings. Additionally, about 27% of students who received a 2 had 10 or more warnings.

We also see a statistically significant difference between the mean number of OCLint DeepNestedBlock warnings for students who received a "Nesting" style score of 1 or 2 (p=0.008, U=14377). Students who received a 1 had on average one more warning than students who received a 2. We do not see a statistically significant difference between the mean number of warnings for students with a "Nesting" score of 0 or 1 (p=0.4, U=232). We also fail to see a statistically significant difference between the mean number of warnings for students who received a "Nesting" style score of 0 or 2 (p=0.19, U=1907). We also note that 56% of students who received a 1 and about 63% of students who received a 0 had no warnings.

Finally, we also see a statistically significant difference between the mean number of OCLint HighNcssMethod warnings for students who received a "Helper functions" score of 1 or 2 (p=0.04, U=19334). Students who received a 1 had on average about one more warning than students who received a 2. We fail to see a statistically significant difference between the mean number of warnings for students who received a score of 0 or 1 (p=0.2, U=2038). We also do not see a significant difference between the mean number of warnings for students who received a 0 or a 2 (p=0.42, U=12055). We see that 36% of students who received a 0 and 27% of students who received a 1 had no warnings in this category, while about 7% of students who received a 2 had 10 or more warnings.

5 Discussion

In our initial investigation, we intended to use the scores assigned by human graders as a ground truth for evaluating the performance of static analysis tools. After analyzing the scores assigned by those graders, we realized that our human graders did not perform consistently enough with each other to function as ground truth. Furthermore, we see trends when comparing human style grading scores to static analysis output indicating that our human graders failed to assign the correct score in many clear-cut cases. This is particularly troubling because resource constraints preclude multiple graders from evaluating each human submission.

5.1 Human Graders Compared Against Each Other

Our analysis of variance revealed that while the median scores assigned by 10 of the graders were consistent, the other 5 graders had a statistically significant difference between their median scores and that of other 10. We also see that the standard deviations of the scores assigned by individual graders range from 1.5 to 4.7. Grader 15 appears to have deducted more points overall than the other graders, with a mean score of 17/22 and median score of 18/22 points. The 10 graders that performed consistently with each other tended to not deduct many points. Among these graders, the lowest median score we see is 20/22 points. We looked for evidence of fatigue or familiarity effects in our graders, but we did not see a statistically significant relationship between the order in which a submission was graded and the score assigned to that submission. Inconsistency in human annotators for programming criteria has been reported in the literature (e.g., Figure 1 of the Buse et al. study[20]); the variance in these graders is the rule rather than the exception[16,18].

5.2 Human Graders Compared Against Static Analysis

The static analysis inspections we chose all have clear relationships with our style grading criteria. For example, students with fewer lines of code exceeding the length threshold should receive more points on the "Line length" criterion than students with more lines of code exceeding the threshold. Similarly, the presence of duplicated code or functions that are too long indicates that the code should be refactored to use helper functions that reduce duplication or divide long tasks into smaller sub-tasks. Our qualitative and quantitative analyses suggest that static analyses provide a much more accurate and consistent evaluation of these criteria than human graders.

In general, we see a weak, if any, correlation between the number of static analysis warnings and the score assigned by human graders for the corresponding style criterion. We also tend to see a significant difference between the mean number of static analysis warnings for students who received a 1 or a 2 for a particular criterion, but not between students who received a 0 or a 1. This suggests that although they may be able to distinguish between students who made no mistakes and some mistakes, *human graders do not make a consistent distinction between students who made some mistakes and those who made many mistakes*.

In other cases, we see no statistically significant difference between the mean number of static analysis warnings for students who received a 0 or a 2 for the corresponding style criterion. That is, *many submissions were either unfairly penalized by humans or should have been penalized by humans but were not*. In some cases, we see that more than 50% of submissions (about 40 students in this case) that had no static analysis warnings were still given a score of 0 by humans. Similarly we see cases where as many as 13% (about 64 students) or 27% (about 123 students) of submissions that received a score of 2 had a non-negligible number of warnings.

Furthermore, we see some cases where submissions with an egregious number of static analysis warnings still received full credit for the corresponding style criteria. For example, we see one student with 746 lines of duplicated code who received full points for the "Helper functions" criterion. Simply seeing that one of the students' source files was over 1200 lines long should have been an immediate sign that the student did not effectively use helper functions.

Overall, it appears that *static analysis tools perform more consistently and accurately than humans when a given style criterion can be evaluated with simple rules*. Since the threshold for these inspections is configurable, there is a very low risk of false positives. Unlike a human grader who could accidentally miss violations of a style guideline, these static analysis inspections consistently analyze the entire source code.

Additionally, we note that while human graders had 2 weeks to render their feedback, the static analysis tools we investigated provide feedback in seconds or less.

5.3 Static Analysis Limitations

Some of the criteria in our style grading rubric are not amenable to static analysis by the tools we investigated. Many of the project-specific criteria are too specific to justify including in general-purpose static analysis tools. While inspections for some of these criteria can be

implemented using abstract syntax tree analysis, instructors would have to decide whether to dedicate human resources towards doing so.

Other criteria are too complicated to be evaluated with simple metrics like those provided by the tools we investigated. For example, OCLint provides an inspection that emits a warning when it sees a variable name shorter than a given threshold (3 characters by default). Although this inspection could correctly detect some poorly named variables, there are notable cases where the tool would produce false positives or false negatives. Some single-letter variable names are established by convention and considered acceptable, such as `i` for loops and `e` for caught exceptions. On the other hand, being longer does not mean that a variable is named well, as in this student code with poorly-named variables that all exceed 3 characters:

```
void set_players(string arg1, string arg2, string arg3,
                 string arg4, string arg5, string arg6,
                 string arg7, string arg8) {...}
```

These names fail to convey meaningful information about values they store. While machine learning may be able to provide more sophisticated analyses, the explainability issues that arise from such approaches make them difficult to use in a grading setting.

5.4 Limitations of the Study

Although our dataset involved a large number of hired graders, they were undergraduate students who received a limited amount of training in how to apply our rubric. Since we analyzed historical data, we did not select or train the graders ourselves. Student submissions contained identifiers, and thus human grader scores may have been affected by grader bias. Each submission was also only evaluated by one human, which limits the conclusions we can draw about inter-rater reliability. However, the scale and approach are indicative of large introductory courses.

6 Conclusions and Discussion

We conclude with some recommendations for how to more effectively use human style graders and static analysis tools to evaluate student code style.

First, style criteria that can be evaluated using simple abstract syntax tree rules should preferably be evaluated using a static analysis tool. In many cases, a preexisting tool can be found to perform common inspections. In other cases (for example, we did not find a static analysis tool to evaluate block indentation in C++, only automatic code formatting tools), such a tool can likely be written using an existing static analysis framework.

Second, unless hired graders can be thoroughly trained, prefer evaluating style criteria on a binary scale rather than a larger multi-value scale. Since we observed that our human graders had trouble distinguishing between code with some errors and code with many errors, using a binary scale may help improve consistency. Additionally, a binary scale is more amenable to grading using a static analysis tool. Especially if students are able to run the static analysis tools on their own code before submitting it, style points can be fairly deducted if any warnings are present.

Third, if static analysis tools are used to evaluate simple aspects of student code style, a few

well-trained human graders can focus instead on evaluating other aspects, such as variable and function names, that static analysis tools may not yet evaluate clearly and accurately.

Fourth, any static analysis tools used to evaluate student code style should be provided to students (or included in feedback from an automated grading system) so that they can run the tools themselves and address the warnings emitted. Prior work suggests that this approach may encourage students to fix their style mistakes earlier in the development process and help them develop better coding style habits.

Ultimately, we compared scores assigned by human style graders to warnings produced by static analysis tools. We found that several of our style grading criteria can be evaluated more quickly and consistently with static analyses than by human graders. Static analysis inspections using simple, abstract syntax tree-based rules can be accurate, clear, and fast for style grading.

References

[1] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201, 2014.

[2] Google. Google C++ style guide, . URL `https://google.github.io/styleguide/cppguide.html`.

[3] Mozilla. Coding style. URL `https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style`.

[4] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721, 2013.

[5] Google. cpplint, . URL `https://github.com/cpplint/cpplint#cpplint---static-code-checker-for-c`.

[6] Tom Schorch. CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *SIGCSE Bulletin*, 27:169–172, 1995.

[7] Al Lake and Curtis Cook. Style: An automated program style analyzer for pascal. *SIGCSE Bulletin*, 22:29–33, 1990.

[8] Johann C. Rocholl. pycodestyle. URL `https://pypi.org/project/pycodestyle/`.

[9] Guido van Rossum, Barry Warsay, and Nick Coghlan. PEP 8: Style guide for Python code, 2001. URL `https://www.python.org/dev/peps/pep-0008/`.

[10] Roman Ivanov. checkstyle. URL `http://checkstyle.sourceforge.net/`.

[11] Clang Team. Extra Clang tools 6 documentation: Clang-Tidy, 2017. URL `http://clang.llvm.org/extra/clang-tidy/`.

[12] OCLint. Oclint, 2017. URL `http://oclint.org/`.

[13] pmd. URL `https://pmd.github.io/`.

[14] cppcheck. URL `http://cppcheck.sourceforge.net/`.

[15] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 181–190, 2018.

[16] Stephen D. Luft. How reliable is daily grading? the inter-rater reliability of daily grades assigned by trained teachers. *Japanese Language and Literature*, 51:1–29, 2017.

[17] Victoria Crisp. Towards a model of the judgement processes involved in examination marking. *Oxford Review of Education*, 36:1–21, 2010.

[18] Henry I. Braun. Understanding scoring reliability: Experiments in calibrating essay readers. *Journal of Educational Statistics*, 36:1–18, 1988.

[19] Val Brooks. Double marking revisited. *British Journal of Educational Studies*, 52:29–46, 2004.

[20] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. *ISSTA '08 Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, 2008.

[21] Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36:546 – 558, 2010.

[22] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-agnostic interpretability of machine learning. *ICML Workshop on Human Interpretability in Machine Learning*, 2016.

[23] A. Meka, M. Maximov, M. Zollhofer, A. Chatterjee, H. Seidel, C. Richardt, and C. Theobalt. Lime: Live intrinsic material estimation. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6315–6324, 2018.

[24] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3), 2003.

[25] James Perretta and Andrew DeOrio. Teaching software testing with automated feedback. In *2018 ASEE Annual Conference & Exposition*, 2018.

[26] duplo — C/C++/Java duplicate source code block finder. URL `http://duplo.sourceforge.net/`.

[27] sonarsource. URL `https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp.html`.