Failures in Reliably Assessing Program Code Readability

Neil C. C. Brown
neil.c.c.brown@kcl.ac.uk
King's College London
London, UK

Marcus Messer
marcus.messer@kcl.ac.uk
King's College London
London, UK

Jennifer Ikin
jennifer.ikin@kcl.ac.uk
King's College London
London, UK

Abstract

Writing correct code is an important part of learning to program, but other attributes such as readability also matter for good code, and thus it is desirable to assess them while teaching programming. In this paper we contrast two ways of grading readability: traditional grading (taken from datasets in three different previous studies), and the use of Comparative Judgement (in a new study with 20 graders on 80 projects). We find that both grading approaches are very unreliable, calling into question whether readability of program code can feasibly be graded at all.

CCS Concepts

• **Social and professional topics** → **Computer science education**; • **General and reference** → **Empirical studies**.

Keywords

Comparative Judgement, Grading, Programming Education, Readability

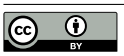
ACM Reference Format:
Neil C. C. Brown, Marcus Messer, and Jennifer Ikin. 2025. Failures in Reliably Assessing Program Code Readability. In *25th Koli Calling International Conference on Computing Education Research (Koli Calling '25), November 11–16, 2025, Koli, Finland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3769994.3770017

1 Introduction

Assessment is an important part of formal education. What teachers assess can determine what students learn, known as "washback" [52], and constructive alignment proposes to deliberately design assessment to align with learning objectives [3]. If we want to teach something, it is important to be able to assess it.

When teaching students to program, the most obvious criteria is that the code should be correct: it should do what is required. But this is not the only criteria by which program code is judged (either in education or in industry): it is also important that the program code should be readable to other people, to aid collaboration, code review and future maintenance [29]. Code readability includes factors such as meaningful names, clear and consistent styling, and good logical flow of code [35].

Taken together, this implies that we should grade the readability of code. Such grading could be done automatically by a computer, or manually by a human. We will consider each possibility in turn.



Koli Calling '25, Koli, Finland

ACM ISBN 979-8-4007-1599-0/25/11
https://doi.org/10.1145/3769994.3770017

Automated assessment systems in programming have primarily focused on whether the behaviour of the program is correct, but have ignored other issues such as code quality, readability, documentation and so on – confirmed by multiple reviews in the area [34, 37, 51, 54]. Artificial Intelligence (AI) is one possible solution to this problem. It appears that AI may be very good at giving formative feedback [36], but there remain questions around using it for summative aspects (i.e., grades) [18]. There are also issues to consider of fairness, equity and accountability. For example: is it acceptable to grade students' university assignments without a human in the loop? The European Union have recently passed an AI act [38], which classifies AI use for grading exams at university as a high risk activity and therefore subject to stringent regulation. It therefore seems wise to continue to consider human grading.

Human grading has historically been used to grade readability, but larger programming projects are time-consuming to grade and there can be issues with calibration and consistency [32, 35]. So an alternative approach to traditional manual grading would be welcome, if it offered improved consistency.

This paper investigates the use of a grading technique known as Comparative Judgement for grading the readability of university-level programming assignments; to our knowledge this is the first time Comparative Judgement has been investigated for use with program code. Comparative Judgement is like a sorting operation with humans as the comparison function, using repeated pairwise "which is better?" comparisons to produce ranks and then grades.

Our research questions are as follows:

- **RQ1:** Does Comparative Judgement of readability of university programming assignments show higher internal consistency than traditional grading?
- **RQ2:** Do the results of Comparative Judgement of readability of university programming assignments agree with those from traditional grading?

We begin by reviewing prior work grading programming assignments, especially readability and code quality. We then explain Comparative Judgement in detail, and describe how we used it on programming assignments. We report on the results of an experiment where we asked human graders to use Comparative Judgement to grade readability of programming assignment submissions, and compare this to traditional grading of the same submissions. We find that there are issues with all methods of grading readability, and we reflect on the implications for programming education.

2 Prior work on grading program readability

2.1 Code quality in education

Kirk et al. [24] recently published a validated code style manual for educational assessment of code style. Their definition of style – "aspects of maintainability, related to the ease of understanding

and changing code, that can be determined by reading the source code" – is broader than readability, but their principles are:

- The rationale, intent and meaning of code is explicit.
- Different elements are easy to distinguish and the relationships between them are apparent.
- Coding constructs are selected to minimise complexity for the intended reader.
- Elements that are similar in nature are presented and used in a similar way.
- All elements that are introduced are meaningfully used.
- Implementation choices are consistent with the problem.
- Code duplication is avoided.
- Related code is grouped together and dependencies between groups are minimised.

The same survey fed in to a working group by Izu et al. [16] on a similar topic. Börstler et al. [5] previously examined code quality perceptions among students, educators and professionals but found no common ground between or within these groups on what makes code readable. Keuning et al. [21] defined code quality as an aspect that appears after writing the initial program and includes but is not limited to documentation, layout, and naming – which was based on the Stegeman et al. [53] rubric and is in line with the Kirk et al. [23] literature-informed model for code style principles.

2.2 Human grading of program readability

In a mapping study conducted on computing education research between 1975 and 2022, Keuning et al. [21] found 195 papers that discussed aspects of code quality education, with how code quality is assessed being a major theme. They found current research focuses on quantitative approaches, including tools to identify code smells and calculate quality metrics automatically – which we will return to in the next subsection. Only a few papers have investigated human roles in assessment.

Ichinco et al. [14] investigated whether expert-provided code changes could be adapted to generalisable rules, and Andrade and Brunet [2] conducted a study to evaluate how students' peer feedback can be used to provide meaningful suggestions to improve the quality of source code. Keuning et al. [20] examined the formative feedback that teachers give on code quality, but not grading/summative feedback which interests us in this study.

Perretta et al. [40] recruited 15 graders and found that the average grade was not always consistent – but this assessed individual grader bias rather than agreement, since each grader had a totally separate set of submissions to grade compared to all other graders.

There has also been work on human evaluation of program readability outside of educational assessment that overlaps with our interest. As part of work to construct a metric for readability, Buse and Weimer [6] asked 120 students to grade readability of code on a 1–5 scale and found a moderate to strong association (Pearson correlation of 0.56). Sergeyuk et al. [48] asked 390 developers to evaluate readability and found a very low level of agreement (Krippendorff's $\alpha = 0.14$). Wiese et al. [56] found that 231 students had varying levels of agreement, from 24% to 90%. Scalabrino et al. [46] asked 30 students to evaluate readability of code snippets and achieved a Cronbach's $\alpha = 0.98$ (very high agreement). Thus the results on human grading of readability show a wide variation in their reliability, from very strong to very weak.

2.3 Automated grading of program readability

Most published work on grading – or more generally evaluating, including formatively – program readability has not used humans, but rather automated assessment. Historically, this generally involved static analysis tools like PMD, Checkstyle or similar [4, 15, 19, 44]. More recent work has focused on using large language models (LLMs) to improve the quality of automated feedback [25] and support automated assessment [1, 11, 30]. We are not aware of any work on validating such automated tools against any other measure of readability, such as human evaluation.

3 Comparative Judgement

In this study we use Comparative Judgement, a novel way to grade students' work. Its core operation will be familiar to computer scientists as a sorting algorithm. Comparative Judgement aims to sort a set of students' submissions into an order, by repeatedly presenting two items to a human and asking which is the better item. This is used to gradually sort the list of student submissions by order of quality. Standard Comparative Judgement uses all possible comparisons (somewhat like a bubble sort), while Adaptive Comparative Judgement [42] aims to reduce the number of needed comparisons by avoiding comparisons to which the answer is already likely to be known, somewhat like a Shell sort [49].

The output of Comparative Judgement is a sorted list, and it is a common mistake to assume that this means its output can only be a relative ranking. The relative ranking can be changed to a set of absolute grades via the insertion of anchor items. The idea is this: create an example assessment answer (either manually created, or from a previous year) that is the exact minimum required to get an A (Anchor-AB) or a B (Anchor-BC). Then you perform Comparative Judgement on your submissions (S1 to S5) and these anchors. Imagine that you get the following ranking:

- S4 (best)
- S3
- Anchor-AB
- S1
- Anchor-BC
- S5
- S2 (worst)

S4 and S3 are grade A because they ranked above Anchor-AB; S1 is grade B because it was above Anchor-BC but below Anchor-AB; S5 and S2 are grade C because they fell below Anchor-BC[1].

Comparative Judgement has been used to assess domains such as primary school writing ability [7] and also across a wide variety of primary, secondary and tertiary education domains (see San Verhavert and Maeyer [45] for a review). To our knowledge it has never been used on computer program code for grading. A similar approach was tried by Luxton-Reilly et al. [27]; they asked students to peer review program code by presenting two side-by-side solutions to the same problem, with the aim of eliciting better *formative* feedback on the code – there was no summative grading.

[1]The reader may want to know what happens if the anchors appear out of order; this indicates that the grading is too inconsistent to be reliable.

4 Domain of interest and dataset

In this paper, we are interested in grading university-level programming projects. For a specific focus we have chosen to investigate grading Java projects, as that is the language primarily used at our institution, and it is also the most popular language in use at universities in general [31, 50]. We are interested in grading larger programming projects, as we feel this is the most challenging grading task in terms of time (and thus workload), and complexity.

To be able to compare Comparative Judgement with traditional grading, we needed a dataset of large Java programming projects. There are several datasets of smaller programming tasks [8, 41], and Java datasets without an associated specific task [55], but we could not find many with larger Java projects. We decided to use our Menagerie dataset [33, 35], which contains several hundred larger Java student submissions for a specific assignment, which have already been graded traditionally on four dimensions (one of which is readability) by multiple graders. This meant we could collect data for Comparative Judgement grading of the dataset and then compare Comparative Judgement to traditional grades from the same set of student submissions in order to answer RQ2. The Menagerie dataset "consists of a second semester CS1 assignment that ran over four academic years... the assignment was a small-group, open-ended paired programming assignment to utilise object-oriented programming concepts to develop a predator/prey simulator." [33]

5 Adapting Comparative Judgement to large programming projects

Comparative Judgement has so far been most frequently used to evaluate short pieces of work [45]. The main reason is quite prosaic: for Comparative Judgement to be efficient, the comparisons need to be fast and done almost at a glance – the items to compare need to fit side-by-side on a screen together. Few university-level assignments (our domain of interest) are of the $\leq$ 30 lines of code length that would allow two pieces of work to fit on one screen side-by-side. Therefore to allow Comparative Judgement to be feasibly used on large programming projects, we need to somehow condense them into a smaller form for grading. We constructed an algorithm to pick a representative method from a larger Java project. (Method is Java's term for what is also known as a class member function.)

At its core, the algorithm takes the set of all methods in a student's submission and attempts to pick the most suitable method for comparison. This is done by first conjoining two filter criteria:

Similarity to baseline The programming assignments included in the Menagerie dataset have a starting project that the students progressively modify. If this is not accounted for, it is possible that we pick part of the baseline code to grade (or a very lightly modified part of the baseline code), which would be unrepresentative of how students actually code. So we used an existing similarity metric [43] to filter out methods which were too similar to the baseline project.

Length The methods needed to fit in one half of a program screen to allow fast comparison. We narrowed down the methods to just those of a suitable length by taking all methods above 10 lines (which we posit is a critical minimum for being able to evaluate code readability), then narrowed down to methods between the median and 95th percentile length (of methods over 10 lines).

We found that a very small number of submissions did not have any methods at all which satisfied both criteria. On closer inspection this was because the students had submitted the baseline code without any modification. In this case it is clearly impossible to grade the readability of the student's own code.

In cases where the two criteria are satisfied by exactly one method, that method is picked. Where there are multiple methods available to pick from, we pick the method which has the median cyclomatic complexity. Cyclomatic complexity is a code complexity metric used in software engineering [9]; we pick the median method as being the one most representative of the complexity of code that the student has added to the baseline project.

6 Experimental design and method

We conducted an experiment to use Comparative Judgement to grade the dataset. We selected 80 graded projects from the Menagerie dataset using a stratified sample to try to get as good a spread of readability grades as possible from the dataset. Adaptive Comparative Judgement [42] suggests that 10 comparisons per item are needed for grading [22], meaning 800 comparisons overall; we chose to require 40 comparisons from each participant, meaning that 20 participants were needed[2]. Participants were recruited from the typical demographic for Teaching Assistants (TAs) at our institution: 3rd year (or later) undergraduates, Masters students, PhD students and postdoctoral researchers. This is the same process used for the graders of the Menagerie dataset (who were the same demographic, also from King's College London a few years before). As with the Menagerie graders we deliberately did not train them on the specific rubric, as we believe this reflects the typical grader recruitment process at most institutions, including our own.

Participants were asked to perform the Comparative Judgement task and then complete a short survey reflecting on their experience, as well as providing relevant demographics (e.g. their experience with Java). They were incentivised with the equivalent of 12 EUR for their participation, which was expected to take around 30 minutes. Our experiment was approved according to the ethical approval procedure of King's College London (ref: MRA-23/24-45348).

6.1 Readability rubric

The readability rubric used in the Menagerie traditional grading uses a representative description for each letter grade. For example, the description for the highest (A++) grade is as follows [33]:

> The application is exceptionally well-organised and very easy to understand; the student used indentation appropriately and consistently to delineate code blocks; each function performs a single well defined operation; the student used meaningful identifier names, i.e., good function and variables names; the student used white space between logical code blocks; the student uses consistent spacing around operators and variables; classes are self-contained with private data hidden and methods are public only when necessary.

[2]For real grading we would anticipate that we would use less participants with more comparisons each, but we wanted to be able to evaluate inter-grader consistency and that would be less valid with fewer participants.

The later descriptions for lower grades are much more concise. We wanted an equivalent rubric to let us compare Comparative Judgement to the existing traditional grades which used this rubric, but we could not provide this rubric as-is to our participants because they were not assigning grades directly but rather comparing items. Instead we used the following instruction which captures the attributes from the original Menagerie rubric:

> We will show you two pieces of code at a time, and we want you to judge which code is more readable. Examples of good readability are:
> - The variables and methods are named meaningfully.
> - Consistent styling (e.g. placement of curly braces).
> - Clear formatting (e.g. good indentation and appropriate whitespace between blocks).
> - Logical flow of code.

7 Experimental results

20 participants completed the experimental task and ensuing survey. According to the survey, 11 had graded before and 8 had not (1 did not answer) – but this is not an unusual response for TAs at our institution, who often only grade for one year. They reported between 3–22 years of experience with programming, with a median of 7 years. 8 of them reported being 3rd year bachelor's students, 10 reported being PhD students, and 2 were postdoctoral researchers.

7.1 Validity checks

We made several checks that participants took the Comparative Judgement seriously. First, we checked if participants repeatedly clicked the left or right choice, which would indicate boredom or disengagement. Using a normal approximation to a binomial distribution, we checked if the percentage of left clicks could come from a distribution with a mean of 50%. The p-value of 0.480 suggests that the users were picking left and right with equal probability, and thus did not show signs of repeatedly clicking the same button. Second, we checked if participants showed a noticeable change in speed, which would again indicate boredom or disengagement. Participants showed a gradual speeding up as they become familiar and experienced with the task, but there was not an indication of boredom such as response time dropping to only a few seconds.

In our survey, we asked the participants if they felt they were consistent in their assessments. 12 answered positively (they felt they were consistent) and 8 answered negatively. We asked if they found it easy or hard; 3 reported it was hard, 7 reported it was easy, and 10 said it was both, depending on the specific comparison.

Finally, we asked participants what they felt was most important to their judgement. The responses are shown in Table 1, and when compared to the rubric in subsection 6.1 we can see a good alignment. The main difference is that participants mentioned commenting which was not part of the rubric (because it was part of the documentation dimension in the original Menagerie dataset).

7.2 Internal consistency (RQ1)

To measure internal consistency of the Comparative Judgement, we used Split-Half Reliability with Spearman-Brown correction [22] – henceforth referred to as SHR. For this measure, 0.7 is considered acceptable, and beneath 0.6 is considered poor [26, 47].

Table 1: The criteria that were mentioned by at least a quarter of the participants when asked (in a free-text response) which criteria they focused on to judge readability during the Comparative Judgement task, with the amount of people (out of 20) that mentioned them.

Criteria	Participant count
Consistent indentation/whitespace	13
Good explanatory comments	11
Good naming of variables and methods	11
Avoiding large amounts of nested code	9
Avoiding long code (in number of lines)	5
General understandability	5

Table 2: The Split-Half Reliability (SHR) with Spearman-Brown correction, plus the 95% Confidence Interval (CI) calculated by bootstrapping (500 repetitions). This is defined directly for Comparative Judgement (top row) and calculated for the rest by simulating a Comparative Judgement task using the absolute grades for readability.

Dataset	SHR	95% CI
Our Comparative Judgement	0.495	(0.316, 0.645)
Messer et al. [35]	-0.858	(-2.794, 0.266)
Buse and Weimer [6]	0.443	(0.195, 0.633)
Scalabrino et al. [46]	0.431	(0.186, 0.609)

To enable comparison with traditional grading, we can calculate this same measure for other datasets using a simulation approach to simulate a Comparative Judgement task as follows. We take two items at random (but preferring closer items, as Adaptive Comparative Judgement does) and use the traditional absolute grades to decide the outcome of that comparison: the higher grade wins the comparison, tied scores are decided by a coin flip. We do this for 10 comparisons per item, and thus we simulate a Comparative Judgement task based on the absolute grades. We simulate the whole judgement process 500 times, which gives us a better estimate and a 95% confidence interval (CI). We did this for several historical readability grading datasets which had available public data: Messer et al. [35], Buse and Weimer [6] and Scalabrino et al. [46].

The results are shown in Table 2. Note that 0.45 (around our best values) is worse than 95% of the Comparative Judgement studies surveyed by Kinnear et al. [22], all results are easily in the poor category of SHR < 0.6 and no confidence intervals contain the acceptable SHR score of 0.7. The negative outcome for Messer et al. [35] indicates that the data is below chance agreement; note that their dataset is for readability of an entire project, whereas the other three are all for snippets of code around 10–50 lines.

7.3 Comparison to traditional grading (RQ2)

Since the snippets in our Comparative Judgement are derived from Messer et al. [35], we can compare whether our Comparative Judgement rankings are consistent with the traditional marking performed in that study. For our analysis we leave aside the issue of mapping the traditional grades and Comparative Judgement outcome to a single scale. If Comparative Judgement is consistent with

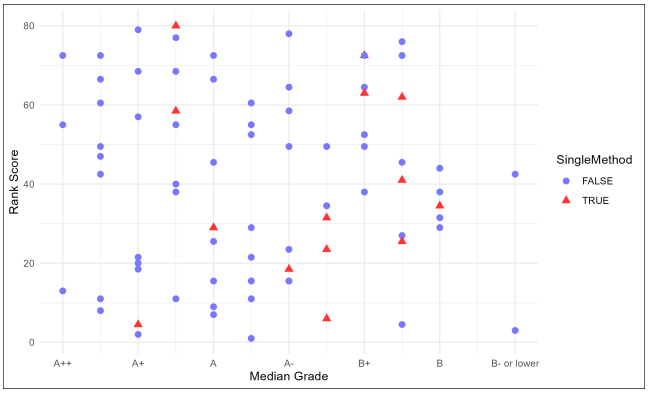


Figure 1: The association between the Comparative Judgement score (Y axis) and the traditional grade (X axis). Red triangles indicate the project in question had a single method of 10+ lines added to the project, blue circles indicate there was more than one such method.

traditional grading, then we would expect to see an ordinal relationship between the two grades even if they are on different scales. In Figure 1, we provide a plot of traditional grades vs the Comparative Judgement ranking of all of the 80 student submissions.

This figure is almost a textbook example of no association. We calculated Kendall's τ, where +1 indicates perfect agreement, -1 perfect disagreement, and 0 no association: the result was -0.01.

One challenge for investigating these results is that the traditional grading was performed on a whole project, but the Comparative Judgement was performed on a representative method; perhaps our system of picking a representative method was flawed and to blame. However, we can perform an analysis to investigate this. Some of the projects only had a single 10+ line method in them, over and above what was carried forward unmodified from the baseline. In this case, we can be confident the method is representative of the code readability, because any other code is very short methods or classes. The colours and shape in Figure 1 provide this information. The red triangles show the 14 (out of 80) projects where there was only a single 10+ line method. Kendall's τ for this subset was 0.17, indicating a mild improvement where there was only one possible method available, but suggests the representative-method algorithm was not the main cause of the lack of agreement.

8 Discussion

We discuss the results of each of our research questions in turn, but first we discuss some limitations.

8.1 Limitations of our methods

The transformation to Comparative Judgement via simulation that enabled our comparison to historic datasets carries both advantages and disadvantages. One advantage is penalising limited grade distributions. In the original Scalabrino et al. [46] dataset, over a third of the snippets were graded 4 on a 5-point scale (where 5 is best). This makes it easy to achieve a high agreement score among graders with metrics like Cronbach's α: if everyone agrees on a 4, the measure produces good agreement. However this is a poor discriminant for grading: one use of grading for both educators and students is to distinguish students' performance [28] and this is not possible if almost everyone gets the same grade. Transforming to Comparative Judgement implicitly penalises the use of similar grades, because we transform ties into coin flips, so if everyone has the same coarse grade, the Comparative Judgement ranking essentially becomes noise and scores poorly on reliability.

The disadvantage of simulation is that it depends on the parameters of the simulation, such as the number of comparisons. We chose 10 per item to be in line with our study and established guidelines [22, 42], but a higher value would increase the SHR of our simulations. Doubling to 20 comparisons would raise reliability to approximately 0.55 for the last two rows of Table 2, although this would still be in the poor category.

8.2 Internal consistency (RQ1)

In this study we investigated the use of Comparative Judgement for grading readability. We found that our participants took the Comparative Judgement task seriously and did not show signs of boredom. We used the SHR metric for calculating consistency, and were able to use simulation to calculate the same metric (thus enabling comparison) on three historic datasets. The technical answer for RQ1 is actually that Comparative Judgement shows consistency at least as good as traditional grading, but this disguises that they are all similarly *very poor* in their consistency.

Consistency is one of the fundamental pre-requisites for grading. If grading the same submission by different markers or even the same marker (as in Messer et al. [35]) produces a very different result then it is unfair to use it to grade students work: the grade is random noise and is not informative for either the students themselves or for universities to distinguish student performance. If no methods can produce a consistent grade for readability then the idea of grading readability at all would be invalid.

There are two possible ways forward for future work. One is to try to increase the consistency of readability grading. Perhaps our rubrics are too minimal and should be made more detailed – although prior work [17, 39] suggests making the rubric more detailed does not necessarily increase consistency. Perhaps our graders need more training on how to grade readability rather than just being given a rubric. Based on prior findings [12] of inconsistent grading in their own courses, Hicks and Douglas [13] suggest that graders need feedback on their feedback to improve, but point out that training can be tedious, and graders (paid per item graded) have little incentive to improve their accuracy.

The other way forward is to stop summatively grading program readability. It is possible that it is too subjective to be able to grade consistently at all, and instead graders should restrict themselves to formative grading only: providing notes on limited and specific places where readability could be improved rather than worrying about assigning an overall grade for the whole assignment.

Note that this result is completely orthogonal to how we derived the representative methods for our dataset (see next subsection); the reliability is about the grading task regardless of where the code itself came from. This pattern was found across three different historic datasets and our new study, which gives us high confidence in our result that grading readability is hard or impossible.

8.3 Comparison to traditional grading (RQ2)

We found that there was no relation between our Comparative Judgement of representative methods, and the grading of the full projects from the same dataset. How can traditional grading and Comparative Judgement be perfectly unrelated? One possible explanation was that picking a representative method is a flawed concept – but even when there was only one method to pick from, agreement was not substantially increased. Another possible explanation relates to the different rubric that was necessitated for Comparative Judgement compared to traditional grading, but the differences were relatively slight.

Instead, the most obvious explanation relates to our RQ1 result. If traditional grading of whole-project readability is close to chance then it makes sense that another measure such as Comparative Judgement, whether chance or valid, would not have a reliable association. So again, the wider implication is that human grading by any method is too self-inconsistent to be able to ask sensible questions about consistency between any of the grading methods.

9 Conclusions

The answers to our research questions were:

- **RQ1:** Comparative Judgement showed poor reliability, but so did several previous studies that graded readability traditionally.
- **RQ2:** The results of Comparative Judgement of representative methods and traditional grading of the whole project are completely unrelated, although this is likely because of the RQ1 result.

Although we have answered our research questions, there is clearly a broader issue. Multiple datasets from different studies all seem to show poor consistency results for grading readability. There are potential limitations: we are assessing consistency through simulation in some cases, and the datasets are assessing different levels of granularity. But it suggests there might be a fundamental problem with the basic idea of grading readability of program code. It may simply be too subjective to be reliably graded by humans.

There has been work in the past to try to build models of readability, for example by Buse and Weimer [6] and Scalabrino et al. [46]. They use the human evaluation as the source material, so such models are again dependent on human grading of code readability. These models are built by grading code on a five point scale, which is already coarse (and the data is often skewed/condensed) followed by dividing code into readable/non-readable which is a very coarse categorisation that can inflate model performance. Fakhoury et al. [10] cast doubt on the value of such models. Sergeyuk et al. [48] found that such models did not match with human evaluation of readability, and – as in this study – found that humans did not really agree with each other on readability evaluations. Börstler et al. [5] found the same result about lack of agreement among humans about readability. So the readability evaluation literature, coupled with the results of this paper, seem to point in the same direction: it is not possible to get a collective understanding of readability of program code, nor reliable human agreement when grading it.

An obvious alternative is to turn to automated grading. Systems such as static checkers are deterministic, so they are completely consistent in their grading; problem solved? Surely: problem elided. If no humans can agree on what makes readable code, adding an automated rule means either the humans should have been using this rule, or there are non-automatable dimensions that humans believe should be used in grading readability. Of course, humans could manually perform the equivalent to automated grading if given a very strict rubric, but the fact that they are inconsistent when given a looser rubric suggests that such a strict rubric might again be avoiding the problem rather than solving it.

An alternative possibility is that humans need better training on how to evaluate readability. The datasets included in this study provided a rubric to evaluators, but no training or certification that graders could evaluate readability accurately according to the rubric. This would again require an establishment of a reliable ground truth for readability of code. Overall, grading readability of program code seems too inconsistent to be useable in education, and much existing work [5, 10, 48] suggests that it is unlikely to be possible to find common ground on readability. Therefore our recommendation is to stop summatively grading program readability unless further research can solve these problems and find anagreed-upon model of readability that can be consistently graded against.

In the spirit of open science, all of our materials, study data, simulation code and analysis code are available in an OSF repository: https://osf.io/cyudn/

Acknowledgments

We are very grateful to our participants, and to Steffen Zschaler for his help in recruitment. We would also like to thank our reviewers for their extensive engagement with the paper, as well as Michael Kölling, Joshua Lock and other members of the King's College London Computing Education Research Centre for their feedback.

References

[1] Umar Alkafaween, Ibrahim Albluwi, and Paul Denny. 2025. Automating Autograding: Large Language Models as Test Suite Generators for Introductory Programming. *Journal of Computer Assisted Learning* 41, 1 (2025), e13100.

[2] Raul Andrade and João Brunet. 2018. Can students help themselves? An investigation of students' feedback on the quality of the source code. In *2018 IEEE Frontiers in Education Conference (FIE)*. 1–8. doi:10.1109/FIE.2018.8658503

[3] John Biggs. 1996. Enhancing teaching through constructive alignment. *Higher education* 32, 3 (1996), 347–364.

[4] Anastasiia Birillo, Ilya Vlasov, Artyom Burylov, Vitalii Selishchev, Artyom Goncharov, Elena Tikhomirova, Nikolay Vyahhi, and Timofey Bryksin. 2022. Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments. In *SIGCSE 2022*. ACM, 307–313. doi:10.1145/3478431.3499294

[5] 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 *ITiCSE-WGR 2017*. ACM, 70–85. doi:10.1145/3174781.3174785

[6] Raymond P.L. Buse and Westley R. Weimer. 2008. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, 121–130. doi:10.1145/1390630.1390647

[7] Daisy Christodoulou Christopher Wheadon, Patrick Barmby and Brian Henderson. 2020. A comparative judgement approach to the large-scale assessment of primary writing in England. *Assessment in Education: Principles, Policy & Practice* 27, 1 (2020), 46–64. doi:10.1080/0969594X.2019.1700212

[8] Liliya A. Demidova, Elena G. Andrianova, Peter N. Sovietov, and Artyom V. Gorchakov. 2023. Dataset of Program Source Codes Solving Unique Programming Exercises Generated by Digital Teaching Assistant. *Data* 8, 6 (2023). doi:10.3390/data8060109

[9] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. 2016. Cyclomatic Complexity. *IEEE Software* 33, 6 (2016), 27–29. doi:10.1109/MS.2016.147

[10] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving Source Code Readability: Theory and Practice. In *ICPC 2019*. 2–12. doi:10.1109/ICPC.2019.00014

[11] Skyler Grandel, Douglas C. Schmidt, and Kevin Leach. 2024. Applying Large Language Models to Enhance the Assessment of Parallel Functional Programming

Assignments. In *Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code '24)*. ACM, 102–110. doi:10.1145/3643795.3648375

[12] Nathan M. Hicks and Heidi A. Diefes-Dux. 2017. Grader Consistency in using Standards-based Rubrics. In *2017 ASEE Annual Conference & Exposition*. ASEE Conferences, Columbus, Ohio. https://peer.asee.org/28416.

[13] Nathan M. Hicks and Kerrie A. Douglas. 2018. Efforts to Improve Undergraduate Grader Consistency: A Qualitative Analysis. In *2018 ASEE Annual Conference & Exposition*. ASEE Conferences, Salt Lake City, Utah. https://peer.asee.org/30366.

[14] Michelle Ichinco, Aaron Zemach, and Caitlin Kelleher. 2013. Towards generalizing expert programmers' suggestions for novice programmers. In *VLHCC 2013*. 143–150. doi:10.1109/VLHCC.2013.6645259

[15] Callum Iddon, Nasser Giacaman, and Valerio Terragni. 2023. GRADESTYLE: GitHub-Integrated and Automated Assessment of Java Code Style. In *ICSE-SEET 2023*. 192–197. doi:10.1109/ICSE-SEET58685.2023.00024

[16] Cruz Izu, Claudio Mirolo, Jürgen Börstler, Harold Connamacher, Ryan Crosby, Richard Glassey, Georgiana Haldeman, Olli Kiljunen, Amruth N. Kumar, David Liu, Andrew Luxton-Reilly, Stephanos Matsumoto, Eduardo Carneiro de Oliveira, SeÁn Russell, and Anshul Shah. 2025. Introducing Code Quality at CS1 Level: Examples and Activities. In *ITiCSE-WGR 2024*. ACM, 339–377. doi:10.1145/3689187.3709615

[17] Anders Jönsson and Gunilla Svingby. 2007. The use of scoring rubrics: Reliability, validity and educational consequences. *Educational Research Review* 2, 2 (2007), 130–144. doi:10.1016/j.edurev.2007.05.002

[18] Marcin Jukiewicz. 2024. The future of grading programming assignments in education: The role of ChatGPT in automating the assessment and feedback process. *Thinking Skills and Creativity* 52 (2024), 101522. doi:10.1016/j.tsc.2024.101522

[19] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *ITiCSE 2017*. ACM, 110–115. doi:10.1145/3059009.3059061

[20] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How Teachers Would Help Students to Improve Their Code. In *ITiCSE 2019*. ACM, 119–125. doi:10.1145/3304221.3319780

[21] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2023. A Systematic Mapping Study of Code Quality in Education. In *ITiCSE 2023*. ACM, 5–11. doi:10.1145/3587102.3588777

[22] George Kinnear, Ian Jones, and Ben Davies. 2025. Comparative judgement as a research tool: a meta-analysis of application and reliability. doi:10.31219/osf.io/c9q3b_v1

[23] Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. 2024. A Literature-Informed Model for Code Style Principles to Support Teachers of Text-Based Programming. In *ACE 2024*. ACM, 134–143. doi:10.1145/3636243.3636258

[24] Diana Kirk, Andrew Luxton-Reilly, and Ewan Tempero. 2025. CSM: A Code Style Model for Computing Educators. *ACM Trans. Comput. Educ.* 25, 1, Article 6 (April 2025). doi:10.1145/3716861

[25] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *SIGCSE 2023*. ACM, 563–569. doi:10.1145/3545945.3569770

[26] Katherine R Luking, Brady D Nelson, Zachary P Infantolino, Colin L Sauder, and Greg Hajcak. 2017. Internal consistency of functional magnetic resonance imaging and electroencephalography measures of reward in late childhood and early adolescence. *Biological Psychiatry: Cognitive Neuroscience and Neuroimaging* 2, 3 (2017), 289–297.

[27] Andrew Luxton-Reilly, Arthur Lewis, and Beryl Plimmer. 2018. Comparing sequential and parallel code review techniques for formative feedback. In *ACE 2018*. ACM, 45–52. doi:10.1145/3160489.3160498

[28] Anders Lysne. 1984. Grading of Student's Attainment: Purposes and Functions. *Scandinavian Journal of Educational Research* 28, 3 (1984), 149–165. doi:10.1080/0031383840280303

[29] Umme Ayda Mannan, Iftekhar Ahmed, and Anita Sarma. 2018. Towards understanding code readability and its impact on design quality. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering (NL4SE 2018)*. ACM, 18–21. doi:10.1145/3283812.3283820

[30] Lauren E. Margulieux, James Prather, Brent N. Reeves, Brett A. Becker, Gozde Cetin Uzun, Dastyni Loksa, Juho Leinonen, and Paul Denny. 2024. Self-Regulation, Self-Efficacy, and Fear of Failure Interactions with How Novices Use LLMs to Solve Programming Problems. In *ITiCSE 2024*. ACM, 276–282. doi:10.1145/3649217.3653621

[31] Raina Mason, Simon, Brett A Becker, Tom Crick, and James H Davenport. 2024. A Global Survey of Introductory Programming Courses. In *SIGCSE 2024*. 799–805.

[32] IC McManus, M. Thompson, and J. Mollon. 2006. Assessment of examiner leniency and stringency ('hawk-dove effect') in the MRCP(UK) clinical examination (PACES) using multi-facet Rasch modelling. *BMC Medical Education* 6, 1 (2006), 42. doi:10.1186/1472-6920-6-42

[33] Marcus Messer, Neil Brown, Michael Kölling, and Miaojing Shi. 2024. Menagerie: A Dataset of Graded Programming Assignments. https://osf.io/q8jbt/

[34] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* 24, 1, Article 10 (Feb. 2024). doi:10.1145/3636515

[35] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2025. How Consistent Are Humans When Grading Programming Assignments? *ACM Trans. Comput. Educ.* 25, 4, Article 49 (Sept. 2025). doi:10.1145/3759256

[36] Ha Nguyen and Vicki Allan. 2024. Using GPT-4 to Provide Tiered, Formative Code Feedback. In *SIGCSE 2024*. ACM, 958–964. doi:10.1145/3626252.3630960

[37] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (June 2022). doi:10.1145/3513140

[38] European Parliament. 2024. EU AI Act: first regulation on artificial intelligence. https://commission.europa.eu/news/ai-act-enters-force-2024-08-01_en.

[39] Rebecca J. Passonneau, Kathleen Koenig, Zhaohui Li, and Josephine Soddano. 2023. The Ideal versus the Real Deal in Assessment of Physics Lab Report Writing. *European Journal of Applied Sciences* 11, 2 (Apr. 2023), 626–644. doi:10.14738/aivp.112.14406

[40] James Perretta, Westley Weimer, and Andrew DeOrio. 2019. Human vs. automated coding style grading in computing education. In *2019 ASEE Annual Conference & Exposition*.

[41] Fynn Petersen-Frey, Marcus Soll, Louis Kobras, Melf Johannsen, Peter Kling, and Chris Biemann. 2022. Dataset of Student Solutions to Algorithm and Data Structure Programming Assignments. In *Proceedings of the 13th Language Resources and Evaluation Conference*. 956–962. https://aclanthology.org/2022.lrec-1.101/

[42] Alastair Pollitt. 2012. The method of adaptive comparative judgement. *Assessment in Education: principles, policy & practice* 19, 3 (2012), 281–300.

[43] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.* 8, 11 (2002), 1016.

[44] Liam Saliba, Elisa Shioji, Eduardo Oliveira, Shaanan Cohney, and Jianzhong Qi. 2024. Learning with Style: Improving Student Code-Style Through Better Automated Feedback. In *SIGCSE 2024*. ACM, 1175–1181. doi:10.1145/3626252.3630889

[45] Vincent Donche San Verhavert, Renske Bouwer and Sven De Maeyer. 2019. A meta-analysis on the reliability of comparative judgement. *Assessment in Education: Principles, Policy & Practice* 26, 5 (2019), 541–562. doi:10.1080/0969594X.2019.1602027

[46] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 6 (2018), e1958. doi:10.1002/smr.1958 e1958 smr.1958.

[47] Stefanie Schuch, Andrea M Philipp, Luisa Maulitz, and Iring Koch. 2022. On the reliability of behavioral measures of cognitive control: retest reliability of task-inhibition effect, task-preparation effect, Stroop-like interference, and conflict adaptation effect. *Psychological research* 86, 7 (2022), 2158–2184.

[48] Agnia Sergeyuk, Olga Lvova, Sergey Titov, Anastasiia Serova, Farid Bagirov, Evgeniia Kirillova, and Timofey Bryksin. 2024. Reassessing Java Code Readability Models with a Human-Centered Approach. In *ICPC 2024*. ACM, 225–235. doi:10.1145/3643916.3644435

[49] D. L. Shell. 1959. A high-speed sorting procedure. *Commun. ACM* 2, 7 (July 1959), 30–32. doi:10.1145/368370.368387

[50] Robert M Siegfried, Katherine G Herbert-Berger, Kees Leune, and Jason P Siegfried. 2021. Trends of commonly used programming languages in CS1 and CS2 learning. In *ICCSE 2021*. IEEE, 407–412.

[51] Draylson M. Souza, Katia R. Felizardo, and Ellen F. Barbosa. 2016. A Systematic Literature Review of Assessment Tools for Programming Assignments. In *CSEET 2016*. 147–156. doi:10.1109/CSEET.2016.48

[52] Mary Spratt. 2005. Washback and the classroom: the implications for teaching and learning of studies of washback from exams. *Language Teaching Research* 9, 1 (2005), 5–29. doi:10.1191/1362168805lr152oa

[53] Martijn Stegeman, Erik Barendsen, and Sjaak Smetsers. 2016. Designing a rubric for feedback on code quality in programming courses. In *Koli Calling 2016*. ACM, 160–164. doi:10.1145/2999541.2999555

[54] Zahid Ullah, Adidah Lajis, Mona Jamjoom, Abdulrahman Altalhi, Abdullah Al-Ghamdi, and Farrukh Saleem. 2018. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. *Computer Applications in Engineering Education* 26, 6 (2018), 2328–2341. doi:10.1002/cae.21974

[55] Ian Utting, Neil Brown, Michael Kölling, Davin McCall, and Philip Stevens. 2012. Web-scale data gathering with BlueJ. In *ICER 2012*. ACM, 1–4. doi:10.1145/2361276.2361278

[56] Eliane S. Wiese, Anna N. Rafferty, and Armando Fox. 2019. Linking code readability, structure, and comprehension among novices: it's complicated. In *ICSE-SEET 2019*. IEEE Press, 84–94. doi:10.1109/ICSE-SEET.2019.00017