Feedback in Context

Using a Code Review Tool for Program Grading

Mary Elaine Califf
School of Information Technology
Illinois State University
Normal, Illinois, United States
mecalif@ilstu.edu

Nick Dunne
School of Information Technology
Illinois State University
Normal, Illinois, United States
ndunne2018@gmail.com

ABSTRACT

The goal of evaluating student work should go beyond summative feedback and the assignment of a grade. However, providing meaningful formative feedback on a program requires connecting the feedback with relevant code segments. We report on program grading using an open-source extension, Code Review, for Visual Studio Code. The extension is designed for professional code review but can be adapted for grading with minimal setup. Once set up, we found the tool straightforward and easy to use. The report given to students provides the category and feedback along with the code itself for each code segment the grader comments on, sorted by file name and location within the file. The category for each comment was used to connect feedback to the rubric used to evaluate programs in the course. We surveyed students in our upper-level algorithms and data structures course in spring 2021. More than 80% of students responding reported that this approach to program grading was more helpful than feedback received in other programming courses the student had taken. Similar numbers of students reported that the feedback improved their learning in the course and helped them "become a better programmer."

CCS CONCEPTS

•Social and professional topics→Computing education→Student assessment

KEYWORDS

Program grading; Formal code review; Rubrics

ACM Reference format:

Mary Elaine Califf and Nick Dunne. 2022. Feedback in Context: Using a Code Review Tool for Program Grading. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V.1 (SIGCSE 2022), March 3-5, Providence, RI, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3478431.3499402

1 Introduction

One of the great challenges of teaching programming is providing reasonably efficient and helpful feedback. As demonstrated in a panel at SIGCSE 2020, there is not a lot of agreement about the best approaches to handling this challenge, but programming instructors generally agree that it is important [15].

Our philosophy of grading seeks to emphasize the combination of functionality, program design, and code clarity. Automated approaches to grading tend to do very well on the first of those but are less able to provide helpful feedback on the other aspects of student submissions. Our teaching environment provides well-trained student teaching assistants helping with program grading, but since we moved away from having students submit printouts of their programs a number of years ago, the TAs' comments have been provided in a textbox on the learning management system. That makes it more difficult for students to recognize the context of some comments. It's simply more difficult for them to understand for sure where the mistake is that caused the wrong result, what specifically in the code is being identified as poor program design, or where their program code is not as clear as it should be. Students also sometimes find it challenging to recognize the relative importance of different comments. This is particularly true in upper-level courses where programs may be quite large and complex.

In the 2020-2021 academic year, we began using a formal code review tool to provide feedback on programs in an upper-level algorithms and data structures course. The feedback was tied to a categorized rubric for program grades. We surveyed students in the course regarding their use of and reaction to the feedback in Spring 2021 and received very positive response to this approach from the survey respondents.

2 Related Work

Much of the work on program grading has focused on automated grading, [8, 10, 14], with some recent interest in applying machine learning to program grading [12]. While the desire to make grading easier and more manageable is praiseworthy, automated grading can provide only limited feedback on issues of design and code quality. While we do employ automated testing of program functionality, using bash scripting, we believe that it is important to provide a consistent and detailed feedback approach that balances a focus on functionality with other concerns. While we do have more grading resources than some teaching environments allow, we believe that all students benefit from receiving feedback on these other areas.

Teachers of programming have sought and debated effective criteria and tools for decades. An early example of such tools provided a tool specifying categories and points for each category, with a goal of promoting consistency in grading [5]. James Howatt discussed a rubric with criteria for each category of grading, but the suggested form for providing feedback does not connect specific pieces of feedback with the relevant program code, only giving an overall evaluation of the specific aspect of the program [6]. Thus, students may have difficulty determining what changes to make to improve their code.

Attempts have been made to provide feedback in the context of the program. One such approach is to grade programs in an interactive demonstration [13]. However, this approach may be better suited to very small classes, and it can focus more on functionality and user interface than on program design and code quality. It also does not leave students with an artifact for later reference unless the session is recorded or detailed notes are taken.

A variety of specialized tools have been created for providing program feedback that attempt to accomplish some or all of our goals in providing grading feedback. The ELP environment allowed for program annotations within the system the students used for programming and interaction with TAs [2]. This system gave students clear connection of comments to the code; however, it is not publicly available for use. The ALOHA environment focused on providing consistent, rubric-based feedback, though comments were not specifically tied to sections of the code [1]. Again, it is not publicly available. Penmarked was an attempt to provide a code grading environment that allowed for written comments using a tablet, but it required a specialized environment that lacked tools such as syntax highlighting, a serious limitation [11]. In addition, it, too, is not currently available for use. Another approach used a plug-in for the Eclipse IDE to provide Web 2.0 style tag-based feedback [3]. This tied feedback to the code, but some students found the feedback difficult to understand, probably because of the very short comments in the tag format. All of these tools seek to support effective feedback, but none are readily available for use.

We are certainly not the first to use some form of code review in providing program feedback. A number of programming instructors have used some form of it, but code review had been primarily used by peers rather than by the instructor or TA [7]. Code review by the instructor has been used to teach operating systems using a specially developed code review system called Gradeboard, which is publicly available but has not been updated since 2014 [4].

Our goal is to provide consistent rubric-based feedback with the specific feedback explicitly tied to the relevant code. We have sought to avoid the issues of custom built tools by using an open source extension to Visual Studio Code. This provides us with a tool that is specifically geared toward working with program code, that is freely available to use with any programming language, and that can be used without committing to a particular learning environment.

3 Teaching Environment

We are reporting on our experience using a code review tool for program grading in an upper-level algorithms and data structures course in a mid-sized public university. Along with covering the theoretical content, the course has a heavy emphasis on implementation, for which students use C++. Students include sophomores, junior, and seniors, and class size is typically between 20 and 25.

Programs in the course are evaluated using a holistic rubric which covers a range of implementation concerns. Functionality and correct data structure and algorithm implementation are the primary factors in evaluation, but we also consider program design, code quality including commenting, correct dynamic memory management, and efficiency.

Our grading process has consisted of having a teaching assistant test programs for functionality and review the code according to the instructor's guidelines, making comments in the textbox for the assignment in the learning management system. The instructor then reviews the comments, editing and checking students' submissions as needed, and then assigns the grade based on a rubric.

Students in the course are provided an opportunity to redo programming assignments (at a 10% penalty on grade) as long as they provided a substantial initial submission. The purpose of this is to encourage students to apply the feedback they have received and grow as programmers. In order to encourage this learning and to make reviewing resubmissions feasible, students must submit a document detailing how they addressed the feedback they received on the program they are redoing. Feedback on the resubmissions is minimal.

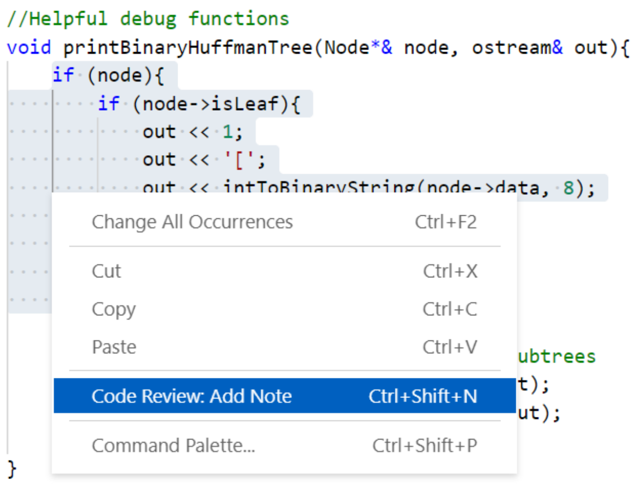
Two clear issues have arisen with this grading process. One is the lack of context for program comments. Even the instructor may not always be clear on what the teaching assistant is referencing in the code. The second is that students are not always clear about the category and severity of the issues each comment addresses. This is more of an issue because of the holistic approach, since a points per error approach would indicate the severity. Therefore, we sought to improve the transparency of the grading process. We wanted to connect the grading comments more clearly to both the rubric category and the code that they referenced.

4 The Tool

The tool we are using is an open source extension for Visual Studio Code called Code Review. It was originally developed by Danny Koppenhagen to support professional code review. Because of the professional code review purpose, it has a variety of features that do not apply to our purpose of grading feedback, but it is quite easy to use for our purposes.

4.1 Using the Code Review Tool

Once the extension is installed, to make a comment on a section of code, we simply select the code and right-click, bringing up the context menu. Then we select the Code Review: Add Note option, as shown in Figure 1.



```
//Helpful debug functions
void printBinaryHuffmanTree(Node*& node, ostream& out){
    if (node){
        if (node->isLeaf){
            out << 1;
            out << '[';
            out << intToBinaryString(node->data, 8);
```

Change All Occurrences Ctrl+F2
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Code Review: Add Note Ctrl+Shift+N
Command Palette... Ctrl+Shift+P

Figure 1: Initiating a code comment

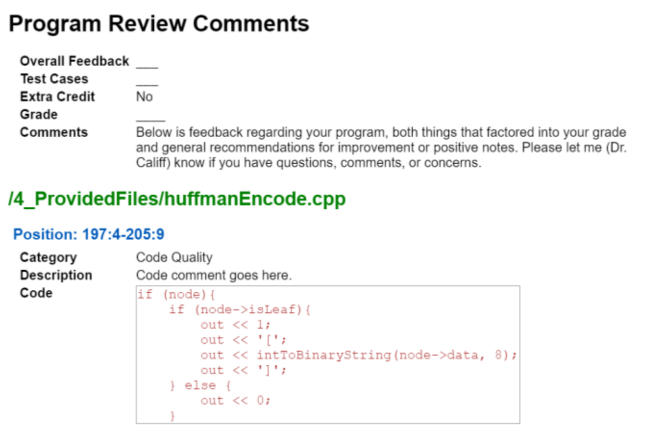
A dialog box then appears, and we enter the feedback for this code segment. Each code comment, or note, has 5 components: a title, a category, a priority, a description, and a place for additional notes. We use only the category and the description, since we believe that those are sufficient to provide good feedback to the students. The category is chosen from a list dropdown, so only the description is typed or pasted in. When grading, we maintain a list of comments for common issues in order to save time and improve consistency of feedback.

Once all comments are made for a program, we export a report in HTML format. This is one of several options the extension provides for saving or sharing the review. The report is created using a template file that we provide. The format of the report is shown in Figure 2.

The upper part of the report is provided for summary information beginning with a summary evaluation in "Overall Feedback." We then paste the results of automated testing of the program into the "Test Cases" section. Then the third section contains the grade for the assignment.

4.2 Setting Up the Code Review Tool

The Code Review extension is usable as installed, but some initial set up is useful to make it more effective for grading purposes. The most important thing is to turn on the option to include the code snippet for each comment, since that is turned off by default (with only the file name and line numbers provided).



Program Review Comments

Overall Feedback	___
Test Cases	___
Extra Credit	No
Grade	___
Comments	Below is feedback regarding your program, both things that factored into your grade and general recommendations for improvement or positive notes. Please let me (Dr. Califf) know if you have questions, comments, or concerns.

/4_ProvidedFiles/huffmanEncode.cpp

Position: 197:4-205:9

Category	Code Quality
Description	Code comment goes here.

Code

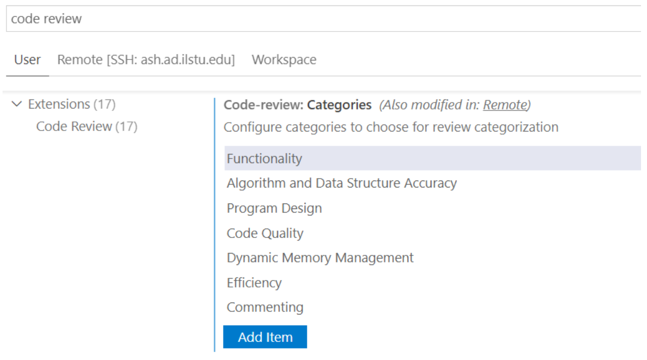
```
if (node){
    if (node->isLeaf){
        out << 1;
        out << '[';
        out << intToBinaryString(node->data, 8);
        out << ']';
    } else {
        out << 0;
    }
```

Figure 2: Code Review Report Format

Another aspect of the tool that should be specialized is the list of categories. By default, this is a fairly typical list of categories for professional code review as shown in Figure 3.

Depending on specifics of a course and how programming standards are communicated to students, this list may work well. In our first semester of using the tool, we simply made use of the default list. However, this list did not use language that closely matched the rubric students has for program grading. It also doesn't address certain educational concerns. For example, in this algorithm and data structures course, we require students to implement specific algorithms, not just solve the problem in any way that could work reasonably. This is not a concern for someone doing professional code review, so there were not categories to address such issues.

In the spring, we modified the category list to help students connect comments on their code to specific parts of the program grading rubric we provided to them. This was trivial to do, as categories can be edited, deleted and added as needed in the settings for the extension.



Code-review: Categories (Also modified in: Remote)
Configure categories to choose for review categorization

- Functionality
- Algorithm and Data Structure Accuracy
- Program Design
- Code Quality
- Dynamic Memory Management
- Efficiency
- Commenting

Figure 3: Code Review Categories in Extension Settings

The second aspect of the extension that benefits from modification to better support educational feedback is the report format. The extension does come with a default template that is appropriate for its purpose but is a bit more complex than is ideal for the educational setting. It includes all of the fields that can be populated for each comment: priority, title, category, description, additional info, and a Git commit hash if available. As code reviewers in a classroom setting, we did not need the majority of this information. Priority of a comment should be determined by the student correlating the category of the comment with the matching name from the program grading rubric. The title was not necessary because there is already a long form description box which has sufficient room for detail. The additional info and commit hash are also not needed since we can fit everything needed in the description, and we are not using Git. Therefore, we modified the template so that our report includes only the category and the description for each comment.

In addition to simplifying the content for each comment on the program, we added sections at the top of the report template to hold summary comments on the program, results of functional testing, and the program grade as was shown above in Figure 2. The report format can also easily be tweaked to add sections for things like extra credit.

The report template combines HTML, CSS, and Handlebars [9]. Handlebars is a templating language that produces HTML. The report is static HTML that can be pasted directly into the learning management system as the instructor comment on the assignment. We created our report template by modifying the existing template to add the sections at the beginning, remove unwanted fields, and slightly modify the format of the resulting report. Our version of the report template can be found at http://www.itk.ilstu.edu/faculty/mecalif/GradingTemplate.zip.

We did find one challenge in creating and using the report template, which was that initially the template include some JavaScript processing in the final HTML report. Since our learning management system does not allow script tags in the feedback boxes for obvious reasons, we need the final report to be static HTML. An advantage of working with an open source tool was that one of our authors was able to modify the extension so that all active processing is done in the report generation process, so we end up with the static HTML that can be used in our environment as well as a wide variety of other environments.

5 Our Experience

We began using the tool in Fall 2020 in our upper level algorithms and data structures course. In that initial semester, we found the tool helpful, but, as noted above, we made some adjustments in the subsequent semester.

The key adjustment involved both the rubric we use for program grading and the code review tool. Because the rubric is used holistically, prior to Spring 2021 students were provided a rubric in the form of a description of A, B, C, D, and F quality work without explicit categories of issues. We found that students had difficulty relating the categories and sometime the content of comments on their programs to the holistic rubric. They also sometimes found it challenging to relate the comments on their code to the descriptions in the holistic rubric. Therefore, we modified the rubric to identify different aspects of program quality explicitly, and we modified the categories used in the code review tool to match the categorization provided by the modified rubric.

We believe this change helped students understand the comments on their code and the relationship of the issues identified to the program grade. We also found that it improved the grading experience. It was much easier for the teaching assistant to assign an appropriate category to a given issue with the program. It also made it easier for the course instructor to be sure of the intent of the comments made, and the impact they should have on the assigned grade.

From the perspective of the evaluators of the code, we see two major benefits of this tool. First, it provides a systematic mechanism for creating code comments that is fairly easy to work with. Especially with a carefully designed rubric, the tool simplifies the process of going through the code and making comments. This helps encourage consistency and helps eliminate distractions.

Perhaps the more important benefit is the provision of context so that students can understand the program design or code clarity issues because the feedback is provided with the code it is about. From the perspective of the teaching assistant, this saves time. There's no need to try to explain where in the code the issue is or copy and paste pieces of code into a separate document. All that is required is selecting the appropriate code segment, picking the category, and typing (or pasting) in the appropriate note.

From the point of view of a course instructor, the context is valuable as a time saver as well. Since the grader's comment is with the code it references, the course instructor can easily see what the issue is. Comments that might have been cryptic out of context are often understandable in context, so we find that we need to reference the program files less frequently. In addition, the availability of the code as context for comments can increase confidence in the teaching assistant's work, as well as making it easier to catch errors and provide correction where necessary.

The primary purpose of the tool was, of course, to improve student learning and create better programmers. The following section reports on students' reaction to the tool. Our observation of the students' work and interaction with us indicates that they did understand the feedback better. A higher percentage of them took advantage of the program redo opportunity than had previously been the case. We also saw clear improvement over the course of the semester in the quality of initial program submissions in areas of design, clarity, and efficiency.

In the spring, we also extended use of the code review tool to a second course, a CS2 class. The teaching assistant for that course was able to use the tool easily and effectively after very little training. Therefore, we are confident that it will be possible for others with similar grading approaches to adopt it.

6 The Student Experience

The most important factor that determines the value of a grading approach is its impact on students. To examine this factor, we surveyed students in the upper-level algorithms and data structures in Spring 2021 and received 22 responses.

We wanted to know whether the code review experience was new to the students. We were surprised to learn that 18% of the students who responded had participated in at least one formal code review experience outside the course. We did not see any correlation between experience with code review and belief that this approach to grading was useful.

We also wanted to know whether students had used the feedback. All respondents indicated that they had read the summary comments and looked at the comments on specific code segments at least once in the semester, with over 90% indicating that they had done so on 4 or more of the 7 assignments on which they received feedback. All respondents also indicated that they had used the comments on specific code segments to improve or fix their program in the course at least once, with the majority indicating that they had done so at least 4 times. We found this very encouraging, since one of our primary goals in using the code review process was to provide meaningful formative feedback on programming assignments.

Using a Likert scale, we asked students to agree or disagree with several statement about the utility of the approach. 95% of students agreed at least slightly with statements that the feedback on specific code segments helped them understand the grade on the program and the summary feedback. We were pleased that students could see the connection between the specific feedback comments and the holistic grade and summary.

Two of our statements focused on program improvement and the program redo process in the course. 90% agreed that the feedback made them more likely to redo a program for an improved grade, and 90% also agreed that the feedback helped them make corrections and/or other improvements to their programs. It is worth noting here that the redo process required students to go through the feedback and provide a table that indicated how they addressed each issue with their program as part of the submission, along with the improved code. Thus, we had made that connection between the feedback and the redo explicit throughout the semester.

We also included some summary statements. 85% agreed that the feedback "made me a better programmer." 80% agreed that the feedback "improved my learning in the course." We also asked to students to compare the feedback to other program grading feedback they had received. 90% of respondents agreed that it was "more helpful than feedback I have received in other programming courses I have taken."

6.1 Student Comments

Our survey provided two open-ended questions for students to address. This is a selection of responses to those questions. Almost all were positive in nature.

The first question was: "Please describe how you believe the use of the code review to provide additional feedback on specific code segments impacted your learning positively or negatively this semester."

- "I greatly appreciated receiving specific feedback on my assignments. I was able to see every detail that I may have gotten wrong in my code so that I may improve upon it in the future."
- "The feedback was helpful when I was doing later programs because I was able to see exactly where I went wrong, rather than having to guess where the professor was referencing in their feedback."
- "It was a huge help to get the specific feedback on code. It helps narrow down mistakes being made and helps the students correct them."
- "I think it had a good impact. I was able to see exactly what issues my program had in a more structured and organized way."
- "The feedback helped me see exact problems with my code, and made it easier to avoid in future programs."

The second question was: "Please explain how you believe this approach to feedback was more helpful or less helpful than feedback you have received in other programming courses."

- "It was more specific than the feedback I normally get, so I feel like I understand my grade better than I do in other courses."
- "This method is more helpful in pointing out the exact problems with my code and did not leave me with much to question, unlike other grading methods."
- "We actually got good focused feedback instead of an overarching, 'the code as a whole is x'"
- "More helpful because at least you know what you did wrong and why it was wrong instead of receiving a bad grade and being told you did something wrong somewhere in the assignment."
- "Significantly more helpful since it is so much more detailed and organized "

Even students who were unhappy with the grading process overall seemed to like the format. One expressed concerns about the holistic grading approach and the fact that there were not points associated with each issue identified in the code, but the response begins, "It was helpful to see exactly what portions of the code were not correct and feedback on how to fix them." None of the response to the open-ended questions expressed concerns with the feedback format or seeing the feedback in conjunction with specific code segments.

7 Discussion

Overall, the use of this code review tool was beneficial to student learning and useful from our perspectives as well. It improved the consistency and clarity of the feedback provided to students on their programs. In addition, it was fairly easy to train a teaching assistant for another course in the use of the tool.

There were several factors that contributed to the successful use of the tool. Designing the report effectively was helpful. Effective feedback required a careful process of program review. In order to help ensure consistency, we maintained a bank of comments that could be used or adapted for common problems. We also automated the functionality testing of the programs, which allowed more of the grading time to be devoted to providing formative feedback on other aspects of the program.

Over the two semesters, we also worked to improve the testing to help pinpoint specific issues, limiting the time spent understanding the source of functional issues in the code.

Another key factor we improved was having a clearly defined grading rubric and ensuring that the categories identified in the feedback match the categories in the grading rubric. While our student in first semester benefited the greater clarity, consistency, and organization of the feedback, anecdotal evidence suggests that they were less able to relate the comments on their programs to the program grading rubric that they were given. This was probably due both to the nature of the rubric, which did not break down categories of the issues as clearly, and to the mismatch between the categories used and the implicit categories of the rubric.

We do believe that there is room for improvement, still. One of the issues that we want to address in the future is helping students distinguish between serious issues, minor issues, and suggestions for best practice that aren't actually impacting the grade. We've found that while the categories and comment content are sufficient for some students to distinguish among these, others have difficulty.

In the coming semester, we want to experiment with some possible ways to improve clarity on these issues. One would be to experiment with using the priority field built-in to the Code Review extension and including priority in the report provided to students. This could help students focus first on the most important issues.

We are also considering experimenting with the order of comments in the report. We are currently listing issues by file in alphabetical order and then by line number. However, the tool does allow for two other orders: by category and by priority. We see potential benefits in each of those. If we order by category, then students will be able to focus on one aspect of their programming at a time as they review the comments. This will be particularly valuable if we determine the order of the categories. If we order by priority, then we can draw students' attention to the most serious issues first, leaving suggestions for best practice that are not serious to impact the evaluation of the program to last.

8 Conclusion

We have described our use of the open-source Code Review extension to Visual Studio Code for program grading in an upper-level algorithms and data structures class. The tool helped to provide context for comments on code and facilitated an organized, consistent, and detailed approach to evaluating the programs. When paired with a detailed grading rubric that used the same categories as the code review comments, students found it more helpful than program feedback they had received in previous courses and believed that the feedback approach had improved their learning in the course and helped them to become better programmers.

We believe that use of this tool or something similar can help programming instructors provide more effective formative feedback on aspects of programming beyond functionality. While other work has had similar goals, this particular tool is desirable because it is freely available and actively maintained.

ACKNOWLEDGMENTS

Thanks to Traci Carte and Cristan Embree for their support and comments on early drafts of this work.

REFERENCES

[1] Tuukka Ahoniemi and Ville Karavirta. 2009. Analyzing the use of a rubric-based grading tool. In Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education (ITiCSE '09). Association for Computing Machinery, New York, NY, USA, 333–337. https://doi.org/10.1145/1562877.1562977

[2] Peter Bancroft and Paul Roe. 2006. Program annotations: feedback for students learning to program. In Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06). Australian Computer Society, Inc., AUS, 19–23.

[3] Stephen Cummins, Liz Burd, and Andrew Hatch. 2010. Tag based feedback for programming courses. SIGCSE Bull. 41, 4 (December 2009), 62–65. https://doi.org/10.1145/1709424.1709447

[4] Christoffer Dall and Jason Nieh. 2014. Teaching operating systems using code review. In Proceedings of the 45th ACM technical symposium on Computer science education (SIGCSE '14). Association for Computing Machinery, New York, NY, USA, 549–554. https://doi.org/10.1145/2538862.2538894

[5] R. Wayne Hamm, Kenneth D. Henderson, Marilyn L. Repsher, and Kathleen M. Timmer. 1983. A tool for program grading: The Jacksonville university scale. In Proceedings of the fourteenth SIGCSE technical symposium on Computer science education (SIGCSE '83). Association for Computing Machinery, New York, NY, USA, 248–252. https://doi.org/10.1145/800038.801059

[6] James W. Howatt. 1994. On criteria for grading student programs. SIGCSE Bull. 26, 3 (Sept. 1994), 3–7. https://doi.org/10.1145/187387.187389

[7] Theresia Devi Indriasari, Andrew Luxton-Reilly, and Paul Denny. 2020. A Review of Peer Code Review in Higher Education. ACM Trans. Comput. Educ. 20, 3, Article 22 (September 2020), 25 pages. https://doi.org/10.1145/3403935

[8] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education (SIGCSE '97). Association for Computing Machinery, New York, NY, USA, 335–339. https://doi.org/10.1145/268084.268210

[9] Yehuda Katz. 2021. Handlebars. Computer Software. https://github.com/handlebars-lang/handlebars.js

[10] David G. Kay, Terry Scott, Peter Isaacson, and Kenneth A. Reek. 1994. Automated grading assistance for student programs. In Proceedings of the twenty-fifth SIGCSE symposium on Computer science education (SIGCSE '94). Association for Computing Machinery, New York, NY, USA, 381–382. https://doi.org/10.1145/191029.191184

[11] Beryl Plimmer. 2010. A comparative evaluation of annotation software for grading programming assignments. In Proceedings of the Eleventh Australasian Conference on User Interface - Volume 106 (AUIC '10). Australian Computer Society, Inc., AUS, 14–22.

[12] Yuze Qin, Guangzhong Sun, Jianfeng Li, Tianyu Hu, and Yu He. 2021. SCG_FBS: A Code Grading Model for Students' Program in Programming Education. In 2021 13th International Conference on Machine Learning and Computing (ICMLC 2021). Association for Computing Machinery, New York, NY, USA, 210–216. https://doi.org/10.1145/3457682.3457714

[13] Fritz Ruehr and Genevieve Orr. 2002. Interactive program demonstration as a form of student program assessment. J. Comput. Sci. Coll. 18, 2 (December 2002), 65–78.

[14] Chris Wilcox. 2016. Testing Strategies for the Automated Grading of Student Programs. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). Association for Computing Machinery, New York, NY, USA, 437–442. https://doi.org/10.1145/2839509.2844616

[15] Ursula Wolz, Gail Carmichael, Dan Garcia, Bonnie MacKellar, and Nanette Veilleux. 2020. To Grade or Not To Grade. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20). Association for Computing Machinery, New York, NY, USA, 479–480. https://doi.org/10.1145/3328778.3366978