What Are We Thinking When We Grade Programs?

Sue Fitzgerald  
Metropolitan State University  
sue.fitzgerald@metrostate.edu

Brian Hanks  
BFH Educational Consulting  
brianhanks@acm.org

Raymond Lister  
University of Technology, Sydney  
Raymond.Lister@uts.edu.au

Renée McCauley  
College of Charleston  
mccauleyr@cofc.edu

Laurie Murphy  
Pacific Lutheran University  
lmurphy@plu.edu

ABSTRACT
This paper reports on a mixed methods study which examines how four experienced instructors approached the grading of a programming problem. Two instructors used a detailed, analytic approach and two instructors employed a holistic approach. One instructor exhibited elements of a primary trait approach. Even though the four instructors used different grading scales and philosophies, their raw scores were highly correlated (Spearman’s rho of .81) supporting the conclusion that experienced instructors usually agree on whether a program is ‘very good’ or ‘very bad’. Clearly there is no single right way to grade programs. Further discourse should be encouraged for the benefit of both educators and students.

Categories and Subject Descriptors
K.3 [Computers & Education]: Computer & Information Science Education - Computer Science Education.

General Terms
Management, Measurement, Documentation.

Keywords
Grading, scoring, programming, rubrics.

1. INTRODUCTION
Grading is the Cinderella activity of teaching – often overlooked yet critically essential. Educators talk more about many other topics – attracting and retaining students, first languages, IDEs, nifty assignments and the latest technology – than we do about grading. But grades are crucial. Within the academic term, formative assessment provides the critical feedback that guides both students and teachers. At the end of the term, summative assessment – the final grade – has a profound influence on a student’s subsequent choice of major and eventual career. Class grade distributions also have an effect on driving pedagogic change by teachers.

Both students and teachers benefit from a methodical and consistent approach to scoring assignments and exam questions. “Learning increases, even in its serendipitous aspects, when learners have a sense of what they are setting out to learn, a statement of explicit standards they must meet, and a way of seeing what they have learned…Assessment requires [faculty] to articulate…explicit and public statements of criteria of performance. By doing so, faculty refine their own understanding of expected abilities, clarify for their colleagues the basis of their judgment, and enable students to understand what performance is required” [7 as quoted in 2].

This paper reports on a mixed qualitative and quantitative study which closely examines how four experienced instructors approached the grading of a programming problem. The differences in how the instructors approached the grading process and the rubrics they developed are revealing and may serve as a set of good alternatives for novice programming instructors.

Our thoughts may also serve as useful input for those who develop automatic graders. In the coming age of Massive Open Online Courses (MOOCs), automatic grading will be crucial. It is not clear to us that existing automatic graders accommodate the variety of approaches illustrated by the set of expert instructors as outlined in this paper. This is particularly true for those instructors who use the holistic approach.

This paper is organized as follows: Section 2 describes some of the theoretical background on developing rubrics and summarizes previous reports on using rubrics for grading programs. Section 3 describes how the student solutions were collected and Section 4 describes how four different instructors graded a programming problem. In Section 5 we analyze our approaches statistically. In Section 6 we discuss the implications of our work and we conclude in Section 7.

2. BACKGROUND
2.1 Manual Marking
Surprisingly little has been published on the grading of programming assignments.

Howatt wrote a practical paper which provides an example of a rubric, including a detailed explanation of how the authors arrived at it and how they used it [6]. The paper reflects on teaching (grading) practice and provides some insights for the novice teacher. However, it has no theoretical basis.

Miller and Peterson devised a problem-independent rubric for grading programs written by undergraduates [8]. One hundred points were available in their rubric, with 80 points for “minimum requirements” and the remaining points for “refinements above minimum”. There were 13 items in the rubric, with 10 items for minimum requirements. The smallest item was worth 2 points (for “author at the beginning”) and the largest was worth 20 points (“correct output”). Only one item was worth 20%, with the next
highest weighting being 10 points, for 5 items. Miller and Peterson evaluated their rubric by having three professors grade ten programs. Using a statistical test, they rejected the null hypothesis that their rubric was not reliable. However, the average scores across the ten programs for each of the three professors were 72%, 75% and 81%. In some institutions this would result in different letter grades for many students, depending upon which professor graded their program.

Hamm et al. presented a scale for use in grading computer programs [5]. Their goal was to facilitate consistent grading among graders and across courses, and to speed up the grading process. The scale, called the Jacksonville University Scale, lists seven factors (execution of the program, correctness of the output, design of the output, design of the logic, design of test data, internal documentation and external documentation), along with a point system for distinguishing “poor”, “average” and “good” performance on programs. Their development of the scale was informed by similar work on grading English composition papers. They built a tool to generate grading forms based on the scale. The programs were graded manually, but the forms were used to guide the grading. They reported anecdotal support for the effectiveness of the grading scale within their own institution.

Olson [10] compared two approaches, analytic and holistic, for grading programs. “In analytic scoring, sub-scores on a series of specific criteria are totaled to give the program an overall score. In holistic scoring, raters read an entire program to obtain a general impression of its overall quality and assign the program a single score”. Each approach was used by two evaluators on a set of thirty end-of-term programming exams. Surprisingly, Olson found that “The inter-rater reliability for the student’s scores on holistically scored hands-on programming ability test was significant (r=.73) and higher than the inter-rater reliability for the analytically scored tests” (r=.56). The “low correlation between the scoring methods indicates that only 45% of the variance in students’ scores on the hands-on final from one scoring method could be explained by use of the other scoring method.”

Smith and Cordova [11] discussed a theoretical foundation for developing grading rubrics based on primary trait analysis (PTA), often used when evaluating writing and other topics in the liberal arts [11]. “PTA uses specific criteria for evaluating performance and defines the factors or ‘traits’ that are to be evaluated as part of the assignment evaluation. For each trait, a grading scale is used to determine the level to which a particular criterion is satisfied. The scale is defined by providing specific descriptions of the levels of achievement that correspond to each point in the scale.” Smith and Cordova provide a detailed rubric for grading programs based on PTA. They offer a six-part description of how to evaluate fourteen separate criteria. The criteria can be weighted. The identification of traits/criteria is analogous to the analytic approach but the level to which a criterion is satisfied more closely resembles the holistic approach as described by Olson [10]. Smith and Cordova developed a spreadsheet version of the tool and reported anecdotally that students liked the feedback it provides and that the tool increased consistency in grading.

2.2 Automatic Marking
A striking feature of the literature on manual grading is its age. In recent years, the bulk of the literature on grading has been in the context of automatic grading. Automatic grading tools have been used to grade student programs for decades, with many of those tools being developed and used within a single institution. According to a survey performed by Carter et al. approximately 70% of first-year programming courses use computer-aided assessment to some extent although not all of these computer-aided assessment techniques were tools for grading programs [3].

More automatic program grading tools have been developed than can be reviewed in this paper. We refer readers to reviews by Ala-Mutka [1] and Douce et al. [4]. Ala-Mutka identified several ways in which automatic graders assess programs, including: (1) functionality, which usually involves running programs against test data sets, (2) efficiency, which can be measured as run time or by number of statements executed, (3) coding style, which can measure one or more aspects of code, such as module length, number of comment lines and use of indentation, and (4) software metrics, such as cyclomatic complexity.

2.3 Types of Rubrics
Rubrics in general (i.e., not just rubrics for grading programs) are often classified into one of three types [12]:

- **Holistic**: Grading is done against a small number of criteria. For each criterion, a small point range is used, often 4 – 6 points (e.g., from 0 for “not attempted” to 4 for “exemplary”). There may also be some trade off between scores for each criterion. Thus the grader forms an overall impression of the piece of work. Speed is one advantage of this approach.
- **Analytic**: Grading is done against a larger and more detailed set of criteria than for holistic grading. Points are assigned to each criterion independently of other criteria and those points are summed to give the total score.
- **Primary trait**: Grading is focused on a very small number of aspects (perhaps one aspect) considered crucial. Like the holistic approach, primary trait grading is often quick, but the difference is that aspects considered secondary are ignored.

3. EXAM DATA
The data examined for this study were collected from a final exam given to 20 introductory students who learnt Java at a small liberal arts university in the United States. Students were given a two-part exam during a one hour and 45 minute period. The first part was taken on paper and comprised matching, expression evaluation, short essay, error detection, and code reading problems. The code reading questions included tracing and Explain in Plain English questions [9].

After turning in the first part, students logged onto classroom machines and completed a set of short Java programs by adding basic constructs and simple object oriented code to a series of templates. Students were given three reference sheets to use during the programming exam. They used temporary accounts to prevent access to previous homework solutions and examples. Classroom management software was used to monitor students as they worked. The programming exam counted for 45 of 100 exam points with an opportunity to earn 5 extra credit points.

We used one of the programming questions from the exam in this study. In that question, students wrote a program to prompt the user to enter a series of integers (using -99 to signal the end of input) and then display the largest and smallest numbers entered.

4. FOUR GRADERS IN ACTION
Four of the authors, all of them experienced university faculty members, graded the 20 student programs. Each instructor approached the problem independently without prior consultation although all the graders were aware that the original intention of the programming exam was to separate those students who should
continue to the next course from those for whom this was not advisable. The instructors did not discuss grading criteria or rubrics before grading. Analysis of the grading techniques showed that the graders approached the problem with very different goals.

### 4.1 Author Murphy

Author Murphy designed the test problem and was the instructor of the class that completed the test. She teaches at a small, undergraduate liberal arts institution in the northwestern part of the United States. Her class sizes are small and her students are primarily residential and traditionally aged. She has over 20 years of teaching experience. Murphy’s intended purpose for the machine-based portion of the final exam was to provide a “litmus test” to determine if students with borderline (C/C-) overall grades should advance to the next course. Her approach was simple and direct. Students’ source files were compiled and executed using a script containing typical input values. Programs were not tested for incorrect or illegal input values. If the program passed the test cases, the source code was manually scanned for general correctness. If everything appeared in order, the program was awarded full points. If any errors or oddities were noticed, they were examined more closely to determine how they should impact the score. Programs that did not compile or contained serious errors were given lower scores. Because the source files contained multiple problems, students were sometimes given full or partial credit for one or more problems that appeared to be implemented correctly, even if the code overall did not compile or contain a runtime error.

Murphy deducted points if the program:
- did not have a priming read
- did not display the smallest or largest value
- did not check the smallest or largest
- initialized the smallest or largest to an arbitrary number

Murphy gave no credit if the student did not attempt to solve the problem, and she gave full credit even if the code potentially returned the loop sentinel (-99) as the highest or lowest value. As Murphy noted, “I clearly wasn’t that picky and in some cases I gave pity points.” Her intention, again, was simply to separate those students who programmed well enough to continue to the next course, from those who did not.

### 4.2 Author Fitzgerald

Author Fitzgerald teaches at a comprehensive university in the US Midwest with approximately 10,000 undergraduate and master’s level students. Her students are non-residential adult learners and her class sizes are usually 25-35 students. She has over 25 years of teaching experience at the graduate and undergraduate level.

Fitzgerald graded the programs by reading the solutions and mentally tracing them, as if they were answers to a problem given on a written exam. She did not run the solutions. She began by developing a rubric based on her understanding of the problem. The rubric identified key program elements and assigned points values for each element. The first version of the rubric was created after looking at the skeleton code but before looking at any student answers.

Correct functionality was considered the most important part of solving the problem and the rubric assigned points accordingly, identifying two large pieces of functionality (find the smallest and find the largest) and assigning them approximately one-third of the points each. Correctly reading the data and stopping at the sentinel value was also considered important, but to a lesser degree.

Fitzgerald’s first rubric was quite simple, checking to see if the program displayed the largest and smallest values, prompted for and read the input values and ended with the sentinel value (-99). However, the rubric was quickly revised after she made a first attempt to use it. She added more details to the generic descriptions, ‘Displays largest’ and ‘Displays smallest’, having failed to anticipate that students might choose arbitrary initial values for ‘Largest’ and ‘Smallest’ which could exclude ranges of numbers. Fitzgerald also discerned that other key elements of the solution included: considering all of the elements entered, avoiding any possibility of choosing the sentinel as the highest or lowest value, and actually displaying the correct results.

After grading the first student’s program using the second version of the rubric, Fitzgerald asked herself if the total score reflected her gut feeling about the student’s understanding. The answer was, “probably not”. The first student’s program did not find the correct answer since it used incorrect initial values and included the sentinel value when comparing for largest and smallest, yet it earned 71% of the possible points (a C). The rubric was revised again by placing more weight on finding the largest/smallest number and by distinguishing between finding and displaying the correct result. The first student’s program was re-graded, and the resulting score of 53% of the possible points (an F) was a more accurate representation of this student’s understanding of the problem. One final adjustment was made to the rubric after grading the second student’s answer.Neither of the first two participants accounted for the case where the user entered no numbers to compare but instead entered the sentinel as the first input value. The fourth and final version of the rubric included one point for catching this error condition. Sadly, none of the students wrote code which caught this error. Fitzgerald concluded that rubrics cannot be finalized before they are tested against actual student work and may require adjustment even toward the middle or end of the grading process.

She noted, “I commonly create the rubric at the same time I design the test problem. I usually solve programming problems using a computer before administering the test to discover unanticipated quirks in the solution and to focus my attention on what it took to solve the problem. But I’m really not very good at anticipating the sorts of mistakes the students will make.”

### 4.3 Author McCauley

Author McCauley is an experienced educator with more than 25 years experience teaching at the undergraduate level, including introductory programming. Her institution is a state-supported liberal arts and sciences institution with a focus on undergraduate education. Class sizes in the introductory courses are about 25 students. McCauley graded by inspecting hardcopies of the code and graded them as though they were part of a written, rather than computer-based, programming test. She started by reading the problem and listing what she considered the characteristics of a correct solution were:

1. Should work for any set of integers, including the empty set.
   a. If the data set is empty (-99 is the first value entered), an error message indicating that there is no largest or smallest value should be displayed.
   b. If the data set is non-empty, a message displaying the correct values is printed.
2. Should include a proper sentinel-controlled loop, with a priming read before the loop, a next read at the end of loop, and no checks for the sentinel value within the body of the loop.
3. Should not contain any superfluous assignments, control structures or other statements even if the program generated correct results (see point 1 above).
4. Should be logically correct, even if not syntactically correct or beautiful.
   a. The programs were not executed, since they were received in PDF format.
   b. The grader ignored issues related to indentation, documentation and meaningful identifier names.

After listing the characteristics of a correct solution, McCauley glanced at several participant submissions, to inform herself of the types of solutions to expect. This convinced her that seeing a correct solution, before proceeding further, would be helpful. She produced a solution that satisfied 1-4 above; this reminded her of how difficult the problem actually was.

With a correct solution in hand, McCauley read through all 20 solutions, building a spreadsheet of notes, with column headings such as “bad initialization”, “good loop form”, and “produces correct results for non-empty data”. She noted these characteristics for each program while reading through it. Since new column headings were generated as the McCauley read through all of the programs on the first pass, she made a second pass through the programs and updated the notes for each. A 10-point grading scale was established:

- Any program that satisfied 1-4 above received a grade of 10.
- Any program that satisfied 1-4 above, but only for non-empty data sets received a grade of 8.
- If a program returned a correct result for non-empty data, but had one imperfection (such as an extra if statement, bad initial values or worked only for the given sample data), it received a score of 7.
- For other programs close to returning a correct result for non-empty data except for one or two minor mistakes, a grade of 6 was assigned.

Submissions that did not receive 6 points or more as described above were placed into buckets and awarded scores based on the buckets into which they fell:

- No work (no loop for input or any logic to find the largest or smallest value): score of 0
- Correct initialization of variables holding min and max
- Correct sentinel controlled loop
- Correct logical structure for updating min and max variables
- Students that did not handle the case where -99 is the first number input. However, none of the students’ solutions handled that case, so he decided to ignore that problem, as did Murphy.

Lister considered the quality of the code if and only if a student scored the two full points for functionality. The only measure of code quality that was considered was indentation. Lister commented, “I don’t expect end-of-first-semester students to understand good design beyond choice of variable names, writing some comments, and indentation. In the context of this machine-based exam, with a tiny amount of code to write (including only two new variables), plus a little bit of time pressure, I didn’t want to be harsh about variable names and comments, but I did feel that even under these circumstances a programmer should still use indenting to keep their own thoughts clear, and to some degree just out of habit. But I only wanted to reward indentation if the code met my minimum criteria for functionality.” After inspecting the first few student solutions, Lister decided to award one point if a student had indented the code within the body of the loop, and another point if the student had indented “if” statements appropriately. These two indenting points were awarded independently of each other.

Thus, in Lister’s grading scheme, the highest possible score was 4. At Lister’s institution, letter grades are awarded on the basis of a total set of points out of 100, and 50 points is considered the minimum passing score that allows a student to proceed to the next programming course. Therefore, after completing the initial grading, Lister then chose to change the total number of points to 2 for students who scored only one functionality point (i.e., a passing score). However, those students who had their score lifted remained ineligible for points from indentation.

4.4 Author Lister
Author Lister teaches at a large research institution in Australia. He has large class sizes, typically 200-400 students. He has 20 years of teaching experience. Lister reasoned that the students completed this task on a computer, which provided them with the opportunity to iteratively write-test-debug. Furthermore, the exam question provided an example set of inputs: 9, 11, 4, 6 and -99, along with the expected output, Largest: 11 Smallest: 4. Therefore, Lister decided that a solution would receive zero points if it did not output either the correct largest value or the correct smallest value for the sample input. His thinking was that the most important factor is the student’s capacity to reason about their own code – so, if a student cannot reason well enough to get at least one of the minimum or maximum correct, then that student is not ready to proceed to the next programming course. Moreover, to let them move on the next course would almost certainly condemn them to failure. Lister notes, “Had this been a pen-and-paper test I would have awarded non-zero points to solutions that scored zero in this test, since a pen-and-paper test does not provide the opportunity to iteratively write-test-debug.”

Lister started with a test of functionality. Most students declared two variables which they used to remember the smallest and largest input values inputted thus far. However, many initialized those two variables to a constant value, often zero. Thus, if the input data consisted entirely of positive numbers (as the sample test data does) then an incorrect smallest value would be output, and if the test data consisted entirely of negative numbers then an incorrect largest value would be output. One point was awarded for solutions that exhibited this problem, and two points to solutions that did not exhibit this problem (even when the chosen initial values only worked for the single given test case). Lister had originally intended to subtract functionality points for solutions that did not handle the case where -99 is the first number input. However, none of the students’ solutions handled that case, so he decided to ignore that problem, as did Murphy.

Table 1 offers a summary of the grading approaches of these four instructors. Fitzgerald and McCauley both used a detailed, analytic approach. Murphy and Lister used simpler approaches, because of their focus on the test as a “gate keeping” exercise, to stop weaker students from progressing to the next course. Murphy tended
toward a holistic approach while Lister tended toward a primary trait approach, given his emphasis on minimum functionality.

Table 1. What instructors were thinking

<table>
<thead>
<tr>
<th>Instructor</th>
<th>Objectives</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Murphy</td>
<td>Is the student sufficiently competent at programming to move to the next class?</td>
<td>Holistic (simplified) Focus on pass/fail</td>
</tr>
<tr>
<td>Fitzgerald</td>
<td>How did the student perform on this problem, based on a set of criteria?</td>
<td>Analytic</td>
</tr>
<tr>
<td>McCauley</td>
<td>How did the student perform on this problem, based on a set of criteria?</td>
<td>Analytic</td>
</tr>
<tr>
<td>Lister</td>
<td>Is the student sufficiently competent at programming to move to the next class?</td>
<td>Primary Trait. Focus on pass/fail</td>
</tr>
</tbody>
</table>

5. ANALYSIS

5.1 Statistical Analysis

As discussed above, the four instructors used different approaches and rubrics to grade the programs. As a result, the largest score possible under each rubric differed for each instructor, ranging from 4 to 18. The differences in scale granularity allowed instructors with larger scales to be more discriminating in their grading. Fitzgerald and McCauley, who had the most granular scales, did not give full credit to any students, while Murphy and Lister gave full credit to 12 and 10 students respectively.

Even though the four instructors used different grading scales and philosophies, the raw scores they gave to the programs were highly correlated, with an average Spearman’s rho of .81. The individual correlations, as shown by the values of Spearman’s rho are shown in Table 2. All correlations were significant at the p=.05 level (adjusted using Bonferroni’s correction technique for multiple comparisons).

Table 2: Correlation of assigned scores

<table>
<thead>
<tr>
<th>Instructor</th>
<th>Fitzgerald</th>
<th>McCauley</th>
<th>Lister</th>
</tr>
</thead>
<tbody>
<tr>
<td>Murphy</td>
<td>0.89</td>
<td>0.79</td>
<td>0.84</td>
</tr>
<tr>
<td>Fitzgerald</td>
<td>0.92</td>
<td>0.81</td>
<td></td>
</tr>
<tr>
<td>McCauley</td>
<td>0.60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although this study considered only one exam question, we were curious to see how the different scoring rubrics would have been 70% of the possible points in their grading scale, while Lister determined the number of students who passed or failed assuming curious to see how the different scoring rubrics would have

5.2 Dealing with Exceptions to the Norm

Overall scoring was highly correlated despite the different approaches chosen by the instructors. However, there were notable exceptions. One student wrote the following rather unusual solution:

```java
int tempNum = 0, lowNum, highNum;
print("Enter a number (-99 to stop): ");
tempNum = Integer.parseInt(kbd.nextLine());
lowNum = tempNum; highNum = tempNum;
while (0 == 0) {
    print("Enter a number (-99 to stop): ");
tempNum=Integer.parseInt(kbd.nextLine());
    if (tempNum == -99)
        break;
    else if (tempNum < lowNum)
        lowNum = tempNum;
    else if (tempNum > highNum)
        highNum = tempNum;
}
```

Murphy gave this student full marks, although it clearly it does not work correctly if the sentinel is the only input value. She attributed the rather odd use of Integer.parseInt for reading, the expression (0 == 0) for the loop condition, and the break statement to terminate the loop to the student’s previous programming experience in C and generally eccentric nature, which led him to enjoy writing unusual code. Fitzgerald and McCauley both deducted some points, assigning the solution 14/18 and 7/10 points, respectively, but gave it an overall passing grade. Because the code threw a runtime exception when tested, Lister gave the solution 0 points, reasoning that since students had the opportunity to compile, run, and test the code on a machine, there was no excuse for compilation or runtime errors.

The variation in scores for this solution highlights potential differences in grading manually vs. machine or automatic grading. It is also an example of the role that personal knowledge of a given student may play in the grading process, which is unlikely to be a factor with automatic grading or in a larger class setting.

6. DISCUSSION

While the graders were consistent in a statistical sense, the combination of the sample size and the traditional 1–in–20 criteria of statistics might provide a false sense of security. Consider McCauley and Lister, who had the poorest correlation in the scores among the four instructors, but still achieved a surprisingly high correlation of 0.60. Table 4 shows how the scores of those two instructors compared on each student with McCauley’s scoring range of 0 to 8 shown on the x axis and Lister’s scores of 0 to 4 shown on the y axis.

The top right cell of the table indicates that four students received the highest possible score from both instructors. A total of eight students are clustered at or near the top score for both instructors. Also, two students scored zero from both instructors. Despite the large differences in how these two instructors graded, very good solutions and very poor solutions tended to receive similar scores from both instructors – perhaps indicating why the graders were consistent in a statistical sense. The heavy lines in Table 4 indicate the pass/fail threshold for these two instructors. The eight students represented by the shaded cells (i.e., 40% of the sample) were passed by one instructor, but failed by the other. Furthermore, other students may have received different letter grades from these
two instructors. There is more to a fair and transparent grading scheme than statistical tests may indicate.

Table 4: Comparison of scores by McCauley and Lister.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

All four of the instructors have many years experience teaching introductory programming. Despite that experience, it is interesting that all four instructors revised their initial approach to grading. One reason for a revision was a mismatch between the score from an analytic rubric and the rubric creator’s “gut feel”. Another reason for revision was the initial expectation that students would successfully treat the case where the sentinel was the first input element, which proved unrealistic as none of the students successfully treated that case.

7. CONCLUSION

In this paper we described how four experienced instructors approach the grading of programs, placing those approaches in the context of the literature on holistic grading, analytic grading and primary trait analysis. The four instructors usually agreed on whether a program is ‘very good’ or ‘very bad’. However, there is no single right way to grade, and the very different motivations of the graders sometimes led to very different grading of programs that lay between the two extremes.

As a learning and teaching community, computer science educators rarely discuss how and what they grade; instead educators just assume that they hold similar values and grade in similar ways. We advocate further reflection and discussion of approaches to grading programs. As a community we should be able to articulate the reasons for our choices for the benefit of our students and to clarify our own thinking. Furthermore, we should discuss those reasons with our colleagues, not competitively, but to learn from each other.

A striking feature of the literature on grading programs is the age of the papers describing manual grading. Most of the recent literature occurs in the context of automatic grading. Even if more and more grading is done automatically in the future, we need to continue discussing how we think grading should be done. Otherwise, automatic grading systems will implement solutions which may not cover the full range of useful approaches to grading. This is especially apparent when holistic and revisionary approaches are considered.

Perhaps even more important than talking to our colleagues, we need to communicate with our students about how we intend to grade their programs. Given the variation in approaches to grading (and resulting scores) presented in this paper, while every educator may think their own approach to grading is obvious, it clearly is not obvious – not even to their colleagues, let alone their students.

8. REFERENCES