A Validated Scoring Rubric for Explain-in-Plain-English Questions

Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, Craig Zilles
[chen386,sazad2,rhaldar2,mwest,zilles]@illinois.edu
University of Illinois at Urbana-Champaign, Urbana, Illinois, USA

Write a short, English language description of what the highlighted code does.

```
int f(int[] array) {
    if (array == null || array.length == 0) {
        return 0;
    }
    int x = array[0];
    int y = 0;
    for (int i = 1; i < array.length; i++) {
        if (array[i] < x) {
            x = array[i];
            y = i;
        }
    }
    return y;
}
```

Sample Student Responses	Score
It returns the index of the smallest value in the array.	6
The function loops through the array and stores the smallest integer value in the array and stores it in x. It stores the array location inside the y and starts off using the first value as the lowest value in the array and keeps looping through. It also has a safety if the array it is given is either null or empty.	3
Function f finds out the length of an array(y).	0

Figure 1: Example of a code reading question and scored student responses.

ABSTRACT

Previous research has identified the ability to read code and understand its high-level purpose as an important developmental skill that is harder to do (for a given piece of code) than executing code in one's head for a given input ("code tracing"), but easier to do than writing the code. Prior work involving code reading ("Explain in plain English") problems, have used a scoring rubric inspired by the SOLO taxonomy, but we found it difficult to employ because it didn't adequately handle the three dimensions of answer quality: correctness, level of abstraction, and ambiguity.

In this paper, we describe a 7-point rubric that we developed for scoring student responses to "Explain in plain English" questions, and we validate this rubric through four means. First, we find that the scale can be reliably applied with with a median Krippendorff's alpha (inter-rater reliability) of 0.775. Second, we report on an experiment to assess the validity of our scale. Third, we find that a survey consisting of 12 code reading questions had a high internal consistency (Cronbach's alpha = 0.954). Last, we find that our scores for code reading questions in a large enrollment (N = 452) data structures course are correlated (Pearson's R = 0.555) to code writing performance to a similar degree as found in previous work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE '20, March 11–14, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366879

KEYWORDS

code reading; CS1; experience report; reliability; validity

ACM Reference Format:
Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, Craig Zilles. 2020. A Validated Scoring Rubric for Explain-in-Plain-English Questions. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366879

1 INTRODUCTION

Learning to program remains a challenge for many students [25]. One explanation for students having difficulty with learning to write code is that courses often expect students to move quickly from learning syntax to writing code, which induces a high extraneous cognitive load that inhibits student learning [22]. Previous research (discussed in Section 2) has identified code tracing exercises, Parson's problems, and code reading exercises as lower cognitive load activities that are part of a loose hierarchy of skills leading to the ability to write code.

In this paper, we focus on *code reading*, also called "Explain in plain English" (EiPE) questions, like the one shown in Figure 1. In these questions, a small code fragment is shown to a student, and the student is asked to provide a high-level natural language (e.g., English) description of the code. Code reading questions are challenging because one needs to consider how the code behaves over all possible inputs and express that behavior at a high-level of abstraction. Code reading questions also exercise students' technical communication skills.

In contrast to code tracing and Parson's problems, which have been widely used in introductory programming instruction [7, 21], code reading exercises appear to have largely been used in the context of research studies. In the process of deploying code reading

Table 1: A SOLO taxonomy developed to score code reading questions by Whalley, et al. [27]

Category	Description
Relational	Provides a summary of what the code does in terms of the code's purpose.
Multistructural	A line by line description is provided of all the code. Summarization of individual statements may be included
Unistructural	Provides a description for one portion of the code (i.e. describes the if statement)
Prestructural	Substantially lacks knowledge of programming constructs or is unrelated to the question

questions in an assortment of introductory programming courses, we found that it was difficult to achieve high inter-rater reliability with previous rubrics.

Previous work has used a *Structure of the Observed Learning Outcome* (SOLO) taxonomy [2] for evaluating code reading questions, as shown in Table 1. At first, we attempted to code our responses using a similar 4-point scale but found it to be too unreliable. We found that many student responses were difficult to reliably assign to a single category and, with so few categories, off-by-one errors led to low inter-rater reliabilities. In particular, for the question in Figure 1, we struggled with answers like "place of smallest" and "this function will *print* the location of the smallest array element"[1].

We found that the source of our scoring difficulty arose from the fact that student answers varied importantly in three dimensions: level of abstraction (as per the SOLO-inspired scale), correctness, and ambiguity. Furthermore, each of these dimensions is continuous; there are many degrees of incorrectness and ambiguity. We briefly considered coding the statements with three-tuples where each dimension was on a multi-point scale. We discarded this idea because not only would it be time consuming for graders, but a large fraction of the space isn't particularly important to distinguish. For example, if an answer has low correctness, we are less concerned whether it is also ambiguous or low level.

Our effort to produce a reliable scoring rubric led us to develop a 7-point scale (Section 4). While we make no claims that this scale is definitive, we demonstrate the construct's reliability and validity in four complementary ways in Section 5: (1) by measuring inter-rater reliability, (2) by assessing the degree to which it predicts other experts relative ranking of student responses, (3) through measuring internal consistency of a survey containing a collection of EiPE questions, and (4) through correlations between students scores on EiPE questions and the other questions on their exams.

2 BACKGROUND

Problem solving appears not to be the primary difficulty in learning to program [10, 13]. Studies show that many non-programmers can produce natural language algorithms [15, 20], but do so in a manner that precludes direct translation into a program for novices (e.g., using set operations instead of iteration with conditionals [15]).

Instead, researchers theorize that there is a loose hierarchy of programming skills where code writing is at the top of the hierarchy and many programming students struggle with tasks lower in the hierarchy [11, 29]. These skills span from understanding syntax (as the easiest), to code tracing (executing code in your head for one particular input), to code reading/explaining (abstracting the behavior of code across all inputs), to code writing (as the most complex) [11]. It has been shown that for a given piece of code, task difficulty generally increases as we move up the hierarchy (e.g., tracing a swap vs. reading/explaining a swap vs. writing a swap) [14, 27], and students' mastery of the lower-level skills is predictive of their code writing ability [6, 12, 14, 24]. In particular, Lopez et al. [14] find that students' performance on tracing and code reading questions account for 46% of the variance in their performance on code writing questions on a paper exam. Murphy et at. [16] found that this correlation replicates when the programming exam is performed on a computer. Lister et al. state that, while their data doesn't support the idea of a strict hierarchy, "We found that students who cannot trace code usually cannot explain code, and also that students who tend to perform reasonably well at code writing tasks have also usually acquired the ability to both trace code and explain code." [12]

Whalley argues that in order for a novice to write a particular piece of code, that they must be able to comprehend that same piece of code and the knowledge and strategies within it [27]. In particular, the important skill that programmers need to achieve is the ability to understand a piece of code at the *Relational* level, meaning that one can provide a summary of what the code does in terms of the code's purpose [27]. This is opposed to a simpler *Multistructural* level, where a student can only provide a line-by-line explanation of the code. Longitudinal studies have shown that students who are unable to explain code relationally early in the semester have difficulty writing code later in the semester [6].

It has been proposed that novice instruction should focus more on code tracing and reading [1, 4, 5, 12, 17, 18, 28]. Lister et al. state, "It is our view that novices only begin to improve their code writing ability via extensive practice in code writing when their tracing and explaining skills are strong enough to support a systematic approach to code writing [...] Until students have acquired minimal competence in tracing and explaining, it may be counter productive to have them write a great deal of code." [12] While all published works involving code reading questions have been on summative assessments, Corney et al. suggest that "the value of explain in plain English problems may therefore be more as formative assessment rather than summative assessment" [5].

The only recent study we identified on code reading questions studied giving 12 Explain in plain English questions on a paper final exam for 334 students [19]. The authors studied the word usage differences of students that reliably answered the questions compared to students that struggled with the questions, finding that the former are more meticulous in their word usage.

3 DATA SET

With the assistance of colleagues, we posed a total of 52 code reading questions to students at a large, public US university. The bulk of these questions were included on online homework [26] and/or computer-based exams [30, 31] in three courses: 1) 8 homework

[1]These answers are scored as 3 and 2 on the 7-point scale introduced in this paper.

Table 2: Sources of student data

Context	No. of students	No. of questions	Avg. responses per question
CS1 for engineers	631	8	583
CS1 for CS majors	590	5	282
Data structures	553	26	105
Survey	93	13	92

problems in a 'CS1 for engineers' course, 2) 5 homework and exam problems in a 'CS1 for CS majors' course, and 3) 26 exam problems, of which each student was assigned one problem on each of 5 exams, in a data structures course. Students were provided course credit for completing these questions. These questions predominantly involved arrays and loops, except in the data structures course, which also included questions involving lists and trees.

In addition, 13 questions were posed as part of a paid survey that was offered to sophomore-level CS students. The survey was entirely optional, and students were given a $5 gift card for completing the survey. Counts of the questions posed and total number of responses received and coded can be found in Table 2. Altogether, we collected and scored 10,024 student responses.

We designed our questions as much as possible to be straightforward implementations of common, level-appropriate tasks or using common patterns that can be described using a short (12 or less words) sentence. In particular, our code reading questions are not intended to be puzzles; they avoid using tricks and uncommon syntax. After our initial trials, students were provided with samples of desirable and undesirable answers to an example question. We found that students were otherwise prone to give long, low-level, and over-complete answers, which are both more time consuming to score and often don't demonstrate whether the student is capable of a brief, high-level description.

4 OUR 7-POINT SCALE

After reviewing a number of responses that raters had scored differently, we ultimately settled on the 7-point scale (0–6) shown in Table 3. While our results in Section 5 suggest that there is some utility to this scale, we are by no means suggesting that this is the only useful scale, and we recognize some of its shortcomings. Nevertheless, we find that the additional categories help distinguish responses that were unnecessarily ambiguous in the smaller scale, and the ambiguity that remains seems unlikely to be significantly improved by further discretization, as it largely results from interpreting the responses rather than from mapping them to scores. Our finding here is in line with previous research which finds that 7-point scales result in stronger correlations with t-test results [9] and that inter-rater reliability generally increases steadily up to 7 scale points, beyond which no substantial increases occur [3].

In mapping the three-dimensional space of (level of abstraction, correctness, ambiguity) to a linear scale we (in hindsight) most valued answers that were correct and at a high level of abstraction, but we were more tolerant of ambiguity. If aspects of an answer potentially fit multiple categories, we assigned the answer to the lowest category.

Table 3 provides example responses in each category for the code fragment in Figure 2. A few additional examples drawn from

```
void f(vector<double> &x, double y) {
    for (int i = 0; i < x.size(); i++) {
        x[i] -= y;
    }
}
```

Figure 2: Code for reducing each vector element by a specified value.

the code in Figure 1 are potentially useful for illustrating how we distinguish different answers at a low-level of abstraction. The response "Stores the smallest value of an int array in x and stores the index of the said smallest value in y." we score as 4 because it references local variables without indicating that one of them is a return value. Since programmers often implicitly refer to a value being returned (e.g., "find out the index of the minimum integer in the array." which is scored as 6), we don't require answers to explicitly specify this. However, we find that assuming that a value is returned is much less obvious when the response indicates that the value is written to a (local) variable. The following is representative of answers that are so low-level that we score them as 1 because they show no ability to raise the level of abstraction.

> "The function f has a return type of int and takes in a first dimension int array. It declares an int x and sets it equal to the first value of the passed array. Then, it declares int y and sets it 0. Then, it runs a for loop that compares x to the other values in the array. If x is greater than the next value of the array, it will set that value to x and the indicie of that value to y. At the end, function f will return the last indicie of the array value which was greater than x."

4.1 Training procedure

The two members of our research team that developed the 7-point scale trained five additional annotators to distribute the scoring load. The training consisted of three steps. First, the new annotators were provided with textual descriptions of the scores as in Table 3 along with two to three examples of student responses at each level of the scale and any questions they had were answered. Second, new annotators were asked to score a dozen previously-scored student responses chosen to represent a variety of different kinds of student answers and feedback was provided. The third step was a repeat of the second step with an additional set of response. At this point, scores had sufficiently converged to consider initial training complete

4.2 Scoring procedure

The following procedure was performed by the trained annotators to score all of the student responses. First, two annotators independently scored the student answers. For any responses where these two scores matched, the score was considered final. The remaining responses were sent to a third annotator, who independently scored them. This third annotator would then consider all three scores and assign a final score for any responses they felt comfortable doing so. Final scores for the remaining responses were established by a process of discussion and reconciliation between all three raters until consensus was reached.

Table 3: 7-point scale for scoring code reading questions based on level of abstraction, correctness, and ambiguity.

Score	Description	Example (See: Figure 2)
6	Clearly and correctly articulates all of the key ideas at a suitably high level of abstraction, with nothing confusing or wrong. May also include lower-level descriptions, as long as they are consistent with the code. Spelling mistakes are allowed, but the English must be generally correct.	"Decreases every value in x by y." — *How we might have said it*
5	Articulates key ideas at a high level of abstraction, but contains improper or incorrect English usage or syntax. No obviously incorrect programming language interpretations.	"Decrease every x in the array by y." — *x was the name of the array variable*
4	Articulates key ideas, but an incorrect interpretation is possible due to imprecise use of language or an insufficiently high level of abstraction. It is likely that the student did understand the purpose of the code, and the errors are in the writing. A notable example in this category is explicitly saying that the code modifies a local variable without indicating that the local variable is returned.	"Every elements in x[] minus y and the result is stored in previous position." — *"previous" could mean that element i is being moved to index $i-1$, but I think they mean stored where it came from*
3	Does not contain enough information to be clearly categorized as absolutely right or wrong. For example, some of the key ideas are not articulated, but nothing clearly wrong is stated.	"This function decreases scale of vector x by y." — *Changing scale usually means multiplication/division, but could mean subtracting*
2	Articulates most of the correct ideas at a high level of abstraction, but at least one thing is clearly wrong.	"Sets all the elements in the vector x equal to -y." — *Recognizes the same thing is done to every element but thinks it is assignment instead of subtraction*
1	Demonstrates some understanding of the code beyond the types of statements, but is far from constructing a high-level description. Either uses a very low level of abstraction or contains multiple errors.	"Subtracts a vector into another vector." — *Recognizes that subtraction is being performed somehow on a vector*
0	Shows no understanding of the code beyond that it is a collection of statements in a programming language. May include true statements about the code (e.g., English transliterations of statements in the code), but shows almost no ability to correctly interpret code behavior.	"Create a vector named x and each element of x is y-1." — *Doesn't create a vector, no elements are $y-1$*

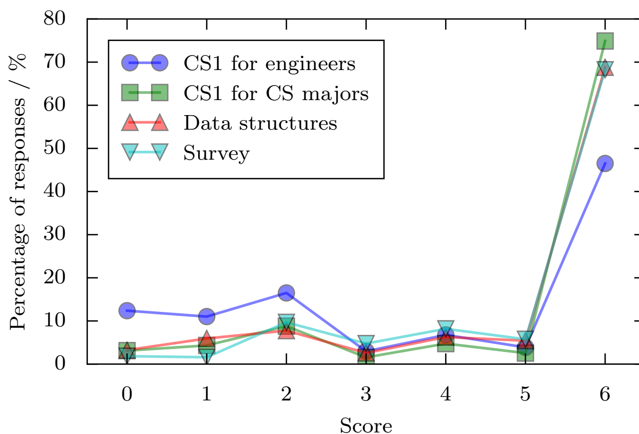

Figure 3: Distribution of scores for each course.

4.3 Distribution as a function of population

Figure 3 shows the distribution of final scores from the four sources. For all sources, the mode was 6, suggesting that each population received questions that were at a reasonable level of difficulty for them. Three of the populations (CS1 for majors, data structures, and survey) had nearly identical distributions, with roughly 70% 6s, 10% 2s, 5% 4s and a smattering of the other scores. The CS1 for engineers course had fewer strong answers (just under 50% 6s) and over 10% in each of 0s, 1s, and 2s. We believe that two factors could have affected the distribution: 1) these students had weaker programming skills at the time the questions were deployed, and 2) only this class used a dynamically-typed language (Python) and our questions didn't provide type information for the function arguments, which (in hindsight) clearly makes the questions harder because they require the reader to perform type inference. At present, we don't know the relative contribution of these two effects.

5 RELIABILITY AND VALIDITY

The proposed 7-point scale only meaningfully measures a construct if it can be reliably measured and others agree that the construct has meaning. We performed four analyses: inter-rater reliability, consistency with expert orderings of student responses, internal consistency, and correlation to other programming related skills.

5.1 Inter-rater reliability

Inter-rater reliability indicates how reliably two annotators can assign the same or nearby scores, which is indicative of the existence of a construct and the ability to teach someone to be able to measure it. In this work, we computed the inter-rater reliability between the two initial annotators for each question. Because our data is interval in nature, we used Krippendorff's alpha with interval metric [8] for our evaluation. A Krippendorff's alpha of 1 indicates perfect agreement, 0 means that the agreement is no better than chance, and -1 indicates perfect disagreement. The usual cutoff for reliable

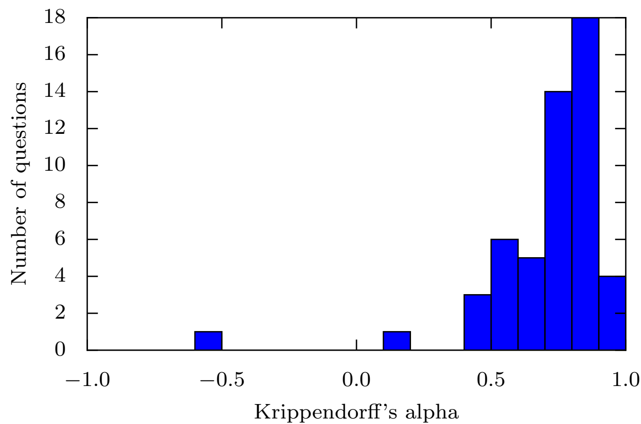

Figure 4: The distribution of Krippendorff's alphas for each annotated question between the two initial annotators.

annotations is 0.8 [8]. We calculated Krippendorff's alpha between the two initial annotators for each annotated question. The distribution of the alphas is plotted in Figure 4. Of the 52 questions, 22 of them (42.3%) had $\alpha > 0.8$, and 15 of them (28.8%) had α between 0.667 and 0.8. The median Krippendorff's alpha was 0.775.

We examined the data for the two outlier cases with low alphas. The question with the negative alpha is explained by the fact that our annotators don't communicate extensively before coding each question. In this particular question, the two annotators had different standards for what was needed for a correct answer. The code we posed to the students was to print integers in a given array, each on their own line. One annotator consistently scored answers without the notion of "each on their own line" as 3 while the other annotator consistently scored similar answers as 6. As such, it is not surprising that they achieved an alpha worse than chance. We view the relative infrequency of this occurring in spite of our lack of communication as a testament to the independence of the scale to the specifics of a given prompt. The explanation for the other outlier is more subtle; the frequency and magnitude of the disagreements for that question were similar to other questions, but because the question had the highest concentration of answers in three adjacent categories (97% of responses were scored as 4, 5, or 6) the denominator of the alpha computation was lower, so our errors represented a larger fraction of the mis-labelling that would happen by chance for such a distribution of answers.

While these inter-rater reliabilities are encouraging, it does show that humans remain somewhat error prone in performing this task. In addition, it motivates the continued use of multiple raters and a reconciliation process (Section 4.2) to achieve reliable scores.

5.2 Consistency with expert rankings

Even if a construct exists and is teachable, it is only meaningful if others find meaning in it. We conducted a study to evaluate the validity of the 7-point scale by asking instructors of introductory programming courses with no knowledge of our scale to order collections of four student answers from best to worst.

At the same university where the student data was collected, faculty involved in introductory programming were contacted to

```
int f(int[] array) {
    int s = 0;
    for (int i = 0; i < array.length; i++) {
        if (array[i] % 2 == 0) {
            s += array[i];
        }
    }
    return s;
}
```

```
there is a function called f which
has an int array called "array" and
```

```
The function searches through the
array "f" and returns the sum of
every even number
```

```
Prints the sum of even numbers from
the array
```

```
The function f returns the sum of
all the even integers in array.
```

Figure 5: The main interface for the validity questionnaire. Participants were asked to rank the responses below in terms of their descriptions of the code above.

identify a few of the most experienced and conscientious teaching assistants. Eleven faculty and 16 teaching assistants (TAs) were then invited to the study by email with a link to an online questionnaire. The questionnaire contained items that display a snippet of code and four anonymous students' responses, as shown in Figure 5. These responses were randomly selected so as to have different scores on our scale by first randomly selecting four different scores and then randomly selecting responses with the corresponding scores. A total of 11 questions with a wide range of scores from our annotated data were used in the questionnaire. Each of the questions was repeated three times with randomly selected responses, which resulted in 33 items in the questionnaire. The participants were asked to drag and drop the responses to order them from the best (at the top) to the worst (at the bottom) in terms of describing the code snippet. Participants were never informed about the existence and details of the 7-point scale. Instead, they were asked to consider the following when ranking responses:

- Which student statements give you the most confidence that the student understands the purpose of the code (i.e., what the code accomplishes).
- Ideally, their statements should be unambiguous, brief, at a high-level of abstraction, written in English, and completely correct.
- We aren't concerned with incorrect spelling/grammar that doesn't interfere with you interpreting their descriptions.

Participants were compensated with a $30 gift card for completing the questionnaire. Six faculty and nine TAs completed the entire questionnaire for a 56% response rate.

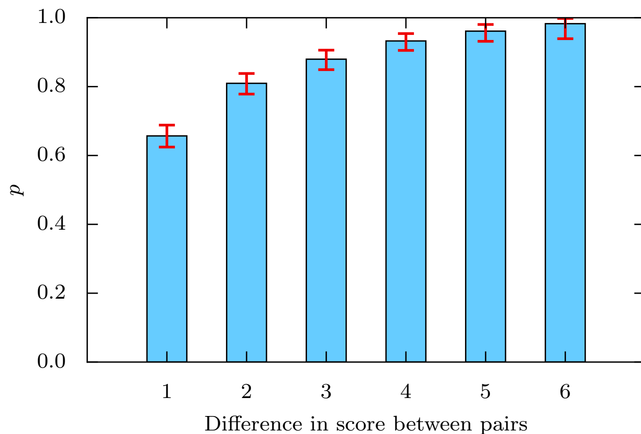

Figure 6: The proportion of agreements p between participants in the validity study and our annotations as a function of the difference in score between each pair of responses.

We computed the number of agreements between the participant rankings and our 7-point-scale scores as follows. Each survey question involved ordering four statements, yielding six pair-wise comparisons. For each pair, if the participant ranked higher the higher-scoring response, we count it as one agreement. In Figure 6, we plot the proportion of agreements p as a function of the distance between the scores of the two responses, along with their 95% confidence intervals. As can be seen, when we've scored responses in adjacent categories on the 7-point scale, participants agreed with our annotations 65% of time. As the score distance increases to 2, the percentage of agreements increases to around 80% and continues to increase from there. These results suggest that the 7-point scale is broadly in agreement with the intuitive responses from experienced instructors. Furthermore, these results match our experience from scoring responses, in that it can sometimes be difficult to choose between an adjacent pair of categories when scoring a response, but rarely is it hard to narrow a response down to a pair of categories. Finally, we found that there wasn't any particular pair of scores that were disproportionately mis-ordered.

5.3 Internal consistency

Internal consistency is the degree to which several items (on a given assessment) that propose to measure the same general construct produce similar scores. In this respect, we would expect a degree of internal consistency in any situation where we ask students a collection of code reading questions of similar difficulty all at the same time, as they are all assessing a students' ability to generate high-level descriptions of code. The first survey that we conducted (described in Section 3) fits this description, as all 92 survey respondents responded to the same 12 code reading questions at one point in time. We computed a Cronbach's alpha of 0.954 for this survey, which indicates high internal consistency [23].

In all of our other data sources, each student responded to fewer questions and the responses were collected over a longer period of time (a few weeks to a few months) where learning could occur between questions. In addition, in two of the courses, students were given a random subset of the code reading questions, and Cronbach's alpha is not robust to missing data.

5.4 Correlation to other skills

Since previous work [14, 16] has demonstrated that code reading skill is associated with code writing skills, we investigated how our students' exam performance on code reading questions correlated to their other exam questions. The student responses collected from the data structures class was used for this analysis since it was the only class that used code reading questions extensively in exams. In the data structures class, exam questions can be separated into three types: code reading questions, code writing questions, and multiple choice questions that mainly focus on understanding concepts such as time complexity. Of the 553 students that completed at least the pre-requisite exam (offered in the second week of classes), we excluded 101 students from the analysis because they did not attempt a large portion of either the code writing questions or multiple choice questions. This resulted in 452 students for the analysis.

Because students received different subsets of the code reading questions, we first applied z-score standardization on a per question basis to account for the variations in question difficulty. After standardization, we computed the average z-score for each student to account for the fact that some students answered fewer code reading questions. For aggregated measures of performance on code writing questions and multiple choices questions, we simply summed each student's score on the two types of questions separately and then applied z-score standardization to the sums. We then computed Pearson's correlation on code reading questions against the other two types of questions.

For code writing questions, the correlation is 0.555, which is significantly positive ($p < 0.0001$). This result is surprisingly close to the correlation coefficient reported by Lopez et al. [14], which is 0.559. Also, it should be noted that we don't expect perfect correlation between reading and writing code, as they are significantly different activities, so this correlation is in a range that makes sense. For the multiple choices questions, the correlation with code reading questions is 0.468 (significantly positive, $p < 0.0001$). In hindsight, it makes sense that code reading is more correlated with code writing than the knowledge related to properties of data structures tested by these multiple choice questions.

6 CONCLUSION

In this paper, we have attempted to share our experience with scoring 'Explain in plain English' (EiPE) questions deployed into a number of contexts. Our primary finding is that student answer quality varies in three dimensions: correctness, level of abstraction, and ambiguity. We demonstrated, at least at one institution, that a scoring rubric developed using these three dimensions can be reliable, can be reliably taught, lines up with the intuition of experienced instructors, can yield internally consistent instruments, and is appropriately correlated to code writing ability. This work has only strengthened our belief that EiPE questions have significant potential in programming instruction and are under-utilized. Future work should explore reducing the effort of grading these questions so that they will be used more broadly.

ACKNOWLEDGMENTS

The authors would like to acknowledge Sayantani Basu, Siwei Shen, Yuren (Sean) Xie, and Chinny Emeka for their assistance in coding the responses and Julia Hockenmaier for discussions relating to this work.

REFERENCES

[1] Owen Astrachan and David Reed. 1995. AAA and CS 1: The Applied Apprenticeship Approach to CS 1. In *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '95)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/199688.199694

[2] John B Biggs and Kevin F Collis. 2014. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

[3] Carolyn C Preston and Andrew Colman. 2000. Optimal Number of Response Categories in Rating Scales: Reliability, Validity, Discriminating Power, and Respondent Preferences. *Acta psychologica* 104 (04 2000), 1–15. https://doi.org/10.1016/S0001-6918(99)00050-5

[4] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and Pedagogy. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education (SIGCSE '99)*. ACM, New York, NY, USA, 37–42. https://doi.org/10.1145/299649.299673

[5] Malcolm Corney, Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. 2014. 'Explain in Plain English' Questions Revisited: Data Structures Problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 591–596. https://doi.org/10.1145/2538862.2538911

[6] Malcolm Corney, Raymond Lister, and Donna Teague. 2011. Early Relational Reasoning and the Novice Programmer: Swapping As the "Hello World" of Relational Reasoning. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 95–104. http://dl.acm.org/citation.cfm?id=2459936.2459948

[7] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[8] Klaus Krippendorff. 2004. *Content analysis: An introduction to its methodology*. Sage publications.

[9] James Lewis. 1993. Multipoint Scales: Mean and Median Differences and Observed Significance Levels. *Int. J. Hum. Comput. Interaction* 5 (10 1993), 383–392. https://doi.org/10.1080/10447319309526075

[10] Marcia Linn and John Dalbey. 1985. Cognitive Consequences of Programming Instruction: Instruction, Access, and Ability. *Educational Psychologist - EDUC PSYCHOL* 20 (09 1985), 191–206. https://doi.org/10.1207/s15326985ep2004_4

[11] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.

[12] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '09)*. ACM, New York, NY, USA, 161–165. https://doi.org/10.1145/1562877.1562930

[13] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, NY, USA, 118–122. https://doi.org/10.1145/1140124.1140157

[14] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research*. ACM, 101–112.

[15] L. A.. Miller. 1981. Natural Language Programming: Styles, Strategies, and Contrasts. *IBM Syst. J.* 20, 2 (June 1981), 184–215. https://doi.org/10.1147/sj.202.0184

[16] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. 2012. Ability to 'Explain in Plain English' Linked to Proficiency in Computer-based Programming. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. ACM, New York, NY, USA, 111–118. https://doi.org/10.1145/2361276.2361299

[17] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in Plain English' Questions: Implications for Teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, USA, 385–390. https://doi.org/10.1145/2157136.2157249

[18] Greg L Nelson, Benjamin Xie, and Andrew J Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 2–11.

[19] Thomas Pelchen and Raymond Lister. 2019. On the Frequency of Words Used in Answers to Explain in Plain English Questions by Novice Programmers. In *Proceedings of the Twenty-First Australasian Computing Education Conference (ACE '19)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/3286960.3286962

[20] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. 2006. Commonsense Computing: What Students Know Before We Teach (Episode 1: Sorting). In *Proceedings of the Second International Workshop on Computing Education Research (ICER '06)*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/1151588.1151594

[21] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Trans. Comput. Educ.* 13, 4, Article 15 (Nov. 2013), 64 pages. https://doi.org/10.1145/2490822

[22] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.

[23] Mohsen Tavakol and Reg Dennick. 2011. Making sense of Cronbach's alpha. *International journal of medical education* 2 (2011), 53.

[24] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International workshop on Computing Education Research*. ACM, 117–128.

[25] Christopher Watson and Frederick W.B. Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*. ACM, New York, NY, USA, 39–44. https://doi.org/10.1145/2591708.2591749

[26] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. ASEE Conferences, Seattle, Washington.

[27] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P K Ajith Kumar, and Christine Prasad. 2006. An Australasian study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Eighth Australasian Computing Education Conference (ACE2006)* (01 2006).

[28] Susan Wiedenbeck. 1985. Novice/expert differences in programming skills. *International Journal of Man-Machine Studies* 23, 4 (1985), 383 – 390. https://doi.org/10.1016/S0020-7373(85)80041-9

[29] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A Theory of Instruction for Introductory Programming Skills. *Computer Science Education* 29, 2-3 (2019), 205–253. https://doi.org/10.1080/08993408.2019.1565235

[30] Craig Zilles, Matthew West, Geoffrey Herman, and Timothy Bretl. 2019. Every university should have a computer-based testing facility. In *Proceedings of the 11th International Conference on Computer Supported Education (CSEDU)*.

[31] Craig Zilles, Matthew West, David Mussulman, and Timothy Bretl. 2018. Making testing less trying: Lessons learned from operating a Computer-Based Testing Facility. In *2018 IEEE Frontiers in Education (FIE) Conference*. San Jose, California.