



Revisiting Rainfall to Explore Exam Questions and Performance on CS1

Antti-Jussi Lakanen, Vesa Lappalainen, and Ville Isomöttönen
Department of Mathematical Information Technology
University of Jyväskylä, Finland
{antti-jussi.lakanen, vesal, ville.isomottonen}@jyu.fi

ABSTRACT

The Rainfall problem comprises small tasks that have been used to investigate student performance in introductory programming. We conducted several kinds of analyses to inform our understandings of student performance in CS1 relating to this problem. We analyzed implementation approaches and program errors, as in related studies, and also explored the role of test writing vis-à-vis the most common student error. Finally, using correlation analyses and manual inspection of the exam answers, we studied how well the Rainfall problem served as an exam question. The students' implementation choices reflected their familiarity with particular loop constructs, while the single most common error concerned division by zero (DivZ), as in many previous studies. Although many students wrote unit tests to guard against DivZ, they often failed to implement according to the tests, which could be partly attributed to the pen-and-paper exam format. From the correlation analyses, we concluded that some students had difficulties with knowledge transfer and that their conceptual understandings of program constructs were insufficient, regardless of them being able to produce working code. The results inform CS1 teachers in preparing materials and exams.

CCS Concepts

•Social and professional topics → Computer science education; CS1;

Keywords

CS1; rainfall; novice programmers

1. INTRODUCTION

One area of interest in novice programming concerns the elements of assignments that can predict overall learning outcomes. In this connection, the so-called *Soloway's Rainfall Problem* [12] has been utilized in several studies to assess

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 the author(s) 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.

Koli Calling 2015, November 19 - 22, 2015, Koli, Finland

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4020-5/15/11...\$15.00

DOI: <http://dx.doi.org/10.1145/2828959.2828970>

novice programmers' progress in learning to construct programs. The Rainfall problem, which may appear trivial at first, has been found to be challenging for many novices. The challenge arises from the several different tasks corresponding to acknowledged Computer Science 1 (CS1) learning goals, ones that need to be accomplished to solve the whole Rainfall assignment. Earlier studies on the Rainfall problem have reported errors in categorizations and students' implementation strategies [2, 3, 10, 13].

The goals of the present paper are two-fold. The first is to present analyses of students' program code for a Rainfall problem in a setting where paper-and-pen exam answers were used as the research data. The Rainfall problem has been studied by Simon [10] using a paper-and-pen exam, while we examined a slightly different version of it. We systematically identified the details of implementation strategies along with categorizing errors in the program code. We also specifically looked at the "division by zero" error that was present in our study as well as in related studies and analyzed whether writing unit tests helped students identify erroneous code. Finally, how the Rainfall problem qualified as an exam question was exploratively investigated through correlation analyses and manual inspection of the students' exam answers. This last step is similar to and was motivated by the study by Reges [8].

The second goal of the paper relates to our on-going experimental research in which we study the differences between pen-and-paper and computer exams. The questions we are interested in answering are as follows: Does the dissection of the problem using the paper-and-pen approach improve ensuing problem solving with the computer? Is one of the approaches superior to the other in terms of student performance? Based on these premises, we chose to use the well-known Rainfall problem and conducted a diagnostic and explorative Rainfall study, in which all the subjects took a paper-and-pen exam. Thus, the paper is centered around our first goal while being additionally motivated by the second one.

2. THE RAINFALL PROBLEM

The original Soloway's Rainfall problem [12] reads as follows.

Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.

Regardless of the short length of the problem, Soloway identified four tasks (or goals) in it: taking the input, sum-

ming the input, calculating the average, and outputting the average. Scholars have often slightly altered the original Rainfall problem to match it with their curriculum (e.g., [3, 10, 13]). The problem is revised to appear like the ones the students have been studying but still different enough to make it valid for measuring learning. Sometimes, it is also sensible to include some specific language constructs or characteristics of a programming paradigm; for instance, loops in procedural programming may be replaced with recursion and higher-order functions in functional programming (e.g., [3]).

The Rainfall version that we used in the final exam is shown in Listing 1. Similar to Simon [10] and Fisler [3], we omitted the use of input and output (I/O). Our rationale was that the approach of our course emphasize graphical interfaces (games) as compared to inputting text using a keyboard. We thereby utilized a ready-made dataset (array) that contained the recorded amounts of rainfall and the sentinel value. The use of an array containing the data was also adopted by Simon [10]. In short, because the use of I/O is omitted, students do not need to write a `while` loop within the outer loop, as the sole purpose of this `while` loop would be to keep determining if the value the user inputs is invalid (i.e., negative).

We included the original idea of ignoring the negative values, which is different from Simon, who registered the “invalid input” as a day with zero rainfall. This made our version of the problem a little more difficult than Simon’s. That is, students needed to keep track of the count of valid inputs while in Simon’s study, the index of sentinel value worked as the divisor.

Listing 1: Our version of the Rainfall problem.

```
/* Implement the 'Average' function,
   which takes the amounts of rainfall
   as an array and returns the average
   of the array. Notice that if the
   value of an element is less than or
   equal to 0 ('lowerLimit'), it is
   discarded, and if it is greater than
   or equal to 999 ('sentinel'), stop
   iterating (the sentinel value is not
   counted in the average) and return
   the average of counted values. */
public class Rainfall {
    public static void Main() {
        double[] rainfalls = new double[] {
            12, 0, 42, 14, 999, 12, 55 };
        double avg = Average(rainfalls, 0,
            999);
        System.Console.WriteLine(avg);
    }

    public static double Average(
        double[] array, double lowerLimit,
        double sentinel) {
        // Write implementation here
    }
}
/* Bonus: Write unit tests. */
```

3. EARLIER FINDINGS

In addition to describing the problem itself, Soloway [12] introduced a plan-based approach for solving the Rainfall problem. For instance, students who attempted to realize reading the data, summing, and counting the values implemented the **sentinel-controlled running-total loop plan**. Soloway thereby suggested a “plan paradigm” for programming education and asserted that plans help students develop mechanisms and explanations for dealing with problem-solving situations. Further, he noted that choosing between different plans is important because realizing a particular plan requires certain implementation strategies. Yet another point he raised is that merging the different plans for completing the whole problem is hard for novices. Later, Ebrahimi [2] concluded that students have problems with both understanding plan composition (how related pieces of the program code represent a specific action) and the semantic interpretation of language constructs (assignment, `if`-statement, loops, etc.) For example, he found that students fail to perform error checking because “simply solving a problem is difficult enough.”

More recently, Venables et al. [13] investigated a Rainfall-like problem and found that several students did either very well or very poorly on the problem, while the distribution of scores was relatively uniform in between the extremes. Simon [10] studied the error categories and suggested that the Rainfall problem is still difficult today, and even harder than before, because programming is taught to a larger audience. For example, all Simon’s students failed to guard the division, thus exposing their solution to “division by zero” error. Furthermore, a great amount of students failed to ignore negative values or made errors with the `for` loop. It has to be noted that in Simon’s study, the assignment was part of a final pen-and-paper exam [11], while in Ebrahimi’s study students were asked to write code with computers.

Fisler [3] explored a functional-first CS1 and found that her students made fewer errors than in prior Rainfall studies and used a wide range of high-level composition structures. She posited that these results were obtained because of using lists, avoiding I/O, teaching data-oriented iterators, and emphasizing a certain design method (in this case, “How to design programs”, HTDP). Her students also had computers and development environments at their disposal, which may have affected the low-level error rates given the error listing features of modern environments. The usage of integrated development environments (IDE) might have affected the high-level details, such as composition structures. This, however, requires further studying.

Given the subtle differences between the earlier studies, we attempt to make informed comparisons between our results and the earlier findings. The differences in research settings motivate our on-going follow-up research in which we address the question of whether students perform better when completing an exam using computers or using the pen and paper method.

4. OUR CS1

Our CS1 course (6 ECTS, 160 hours) uses a procedural paradigm, utilizing ready-made objects from libraries. Our first programming language is *C#*. Course completion requires passing an exam, a sufficient number of completed

exercises during each week (minimum 40%), and a course assignment, which typically involves programming a game.

The learning objectives consist of variables and functions and other typical CS1 topics [1], such as selection, repetition, arrays, and information encapsulation. In the Rainfall assignment, knowledge of how to work with arrays is crucial; we introduce the concept of arrays in the first week of the course. Students start writing loops during the fifth week (out of 11). Students have an IDE (Visual Studio, Xamarin, MonoDevelop) at their disposal from the very beginning of the course. We also use a specific educational software tool that allows students to practice basic algorithms by dragging and dropping elements in an array while the software simultaneously generates the corresponding C# or Java code for the students to observe. The current version of the tool also supports variables and some basic arithmetic.

The game theme is given a lot of emphasis during the course to get into the “do mode” early, to motivate students [6], and to enable rapid visual feedback when making changes in the program code. Right from the first week, the students get to do graphical programs, draw shapes with different colors, and make them move with realistic physics, etc. A part of the exercises relate to games during each week. Towards the end of the course, each student works on the bigger course assignment (a working computer game, 30 working hours). We use the *Jypeli* programming library [4] as a technical framework for game development.

Writing unit tests with the *ComTest* tool [5] using the test-driven development (TDD) approach is also introduced. While writing tests first and using them to develop programs is highly recommended, it is not currently obligatory. However, students who write unit tests in the weekly assignments and in the final exam are credited with extra points. In light of the present study, we hypothesized that students writing tests would benefit from the convention of thinking over and writing down the examples of function inputs and outputs before writing the function implementation.

5. METHOD

A version of the Rainfall problem (see Section 2) was included in the final exam of a CS1 course in the fall of 2014 at the University of Jyväskylä. The subjects of the present study were the 139 students who took the exam. The exam was undertaken with a pencil and paper, and the data used was the students’ hand-written exam answers.

Our research questions were the following:

- RQ1: What implementation strategies and details are found and how they compare with previous studies?
- RQ2: What kind of errors did the students make?
- RQ3: Does writing unit tests help students to proceed with less errors?
- RQ4: How does the Rainfall problem qualify as an exam question?

The first two questions relate to previous research in light of our particular course setting, C# language, and a paper-and-pen exam format. The remaining two questions reflect an explorative research interest. The third question refines the second question from a test writing perspective and explores the usefulness of test writing in relation to the exam

setting used. The fourth question is motivated by Reges [8], who raised the topic of how particular exam questions might explain overall exam performance. Here, our initial working hypothesis was that the Rainfall problem, acknowledged to be challenging to the students, might explain the students’ exam success. Students’ Rainfall answers provided the entry point from where we proceeded with several kinds of correlation calculations.

With respect to RQ1 and RQ2, the first 20 solutions were read by the first two authors, who produced an initial, emergent coding scheme. The coding scheme and marking rubric were then communicated to an assistant who coded the rest of the answers. In the end, an agreement was reached through discussions among these three actors. Through shared discussions, we expanded the coding scheme to make it more fine-grained and thus bring out the important nuances in the dataset. This iterative process continued until the end of the process of writing up the results. Altogether, we identified 29 implementation codes (of which 8 related to unit tests) and 35 related to error codes. It must be noted that the emergent coding scheme and the marking rubric are two separate things. There were inaccuracies in the program code that were regarded as errors in our categorization but were not counted as errors in the marking rubric given the learning goals with the Rainfall problem.

Once the first-level coding was finished, we compared our implementation categories with ones that emerged in Fisler’s study [3]. We refer to these approaches as needed in the results section. Error categories in turn were associated with the acknowledged tasks of the Rainfall problem using the list adopted by Fisler:

- Sentinel: Ignore inputs after the sentinel
- Non-negative: Ignore invalid inputs
- Sum: Total the valid inputs
- Count: Count the valid inputs
- DivZero: Guard against division by zero
- Average: Average the valid inputs

We also report implementation and error categories that we could not find in the literature.

In addressing RQ3, we specifically observed whether test writing helped students to avoid the most common error in the data (DivZ). We additionally contacted the students who failed to avoid the error despite their writing unit tests and asked them about their difficulties. Finally, in our last research step (RQ4), correlations were calculated between individual exam questions and total exam score and between particular combinations of exam questions. Interesting cases (exam answers) were inspected manually.

5.1 Our model solution and marking rubric

Our model solution is shown in Listing 2.

Listing 2: Our model solution to Rainfall.

```
public static double Average(
    double[] array, double lowerLimit,
    double sentinel)
{
    double sum = 0;
    int count = 0;
    for (int i = 0; i < array.Length; i++)
    {
```

```

    double item = array[i];
    if (item <= lowerLimit) continue;
    if (item >= sentinel) break;
    sum += item;
    count++;
}

if (count == 0) return lowerLimit;
return sum / count;
}

```

The answers were graded using a scale from zero to six marks. An answer with full marks contained the following elements.

- Initialize `sum` and `count` variables
- Introduce loop correctly (index initialized, correct stopping condition, correct increment)
- Ignore negative and zero
- Stop summing and counting when sentinel encountered
- Count valid inputs
- Guard against division by zero
- Return correctly calculated average

Grading was based on the items above. For each list item, the reductions were done through case-by-case inspection and were based on the severity of the error, as it was impossible to define univocal reduction rules due to the wide range of student errors (see Appendix B). We could define only a few general reduction rules, such as “missing guarding against division by zero results in a 0.5 mark reduction” and “missing the loop results in the reduction of all marks”. The students could earn one bonus mark by writing unit tests in the Rainfall problem.

6. RESULTS

Although in this paper we are mostly studying student errors, we found that the Rainfall problem yielded the highest average score of the exam questions, which was 68.8%, while the overall average in the exam was 66.3%. Twenty-four students completed the Rainfall problem without committing any of the errors in our error categorization (Appendix B), and for 34 students the “division by zero” error was the only error they committed. With the extra marks received due to test writing, the Rainfall average was 71.6%.

6.1 Implementation categorization

Appendix A summarizes our coding scheme concerning the students’ implementation details. Of the 30 low-level categories, we identified larger groups of different granularity: the high-level composition structure and the more detailed choices that the students make with loops, flow control, and algorithmic design. In addition, we extracted categories related to unit testing. In the following, we raise the main points in the data according to these larger groups and discuss our data in relation to earlier findings.

6.1.1 High-level composition structure

In the present study, not many high-level structures to implement the Rainfall assignment appeared. In fact, a clear majority of students implemented it using the “Single Loop” structure (I1, Appendix A). The idea with this structure is

that incrementing the sum and counting the non-negative input, as well as checking if the sentinel value is reached, is all done inside a *one-loop structure*. After the loop, division by zero is checked and the average is counted and returned. A total of 129 students (93%) used “Single Loop”, whereas four students used “Clean First” (uses a separate data structure to add the “cleaned” data, I2), and one implemented “Clean Multiple” (traverses the input twice, once to count non-negative data, once to sum them, I4). In addition, three students used what we call “Clean Inside Single Loop” (I3), where the purpose was to clean the data by removing unwanted input from the original data structure. Unfortunately, removing items from a C# array is not possible which made this approach practically impossible to implement, and inevitably led to a lower grade. Finally, two students wrote answers that showed no clear structure (I5).

The Clean First approach, that is, copying or moving items from one data structure to another (with a possible filtering) was not practiced during the course. Rather, the weekly assignments concentrated on working with a single structure and getting “one thing” out of the data—averages, maximum and minimum values, or an index of the given value in an array. In “Soloway’s terms”, we could state that such tasks need a **walk-through-an-array plan**. Furthermore, C# functional programming features (LINQ) were only briefly mentioned in a few bonus tasks due to the restricted time frame. This explains the dominance of one structure and the small proportion of other high-level composition structures.

In comparison with our one popular composition structure, Fisler [3] found three popular high-level composition structures: Single Loop (18%), Clean First (42%), and Clean Multiple (21%). Further, she observed a diversity of other composition structures as well. She posited that knowing and using built-in and higher-order functions significantly influenced how students structured Rainfall solutions, which suggests that structuring the solution is, in addition to the course syllabus, explained by the choice of programming language.

6.1.2 Choice of loop structure

In this subsection, we discuss the choice of loop structure (categories I6–I9) and how this choice affected students’ utilization of other flow control structures (I10–I12), and how it affected their algorithmic design (I13–I18). We found these categories to be closely interconnected as the dominating high-level composition was Single Loop, as mentioned in the previous section.

A clear majority of the students (117, 84%) used `for` loop to iterate through the input array (I6), nine (6%) used `foreach` (I7), nine (6%) used `while` (I8), and one used `do-while` (I9). The remaining few students were unable to produce pertinent code. Even though `while` is often a more natural way to iterate through an unknown number of elements (after all, sentinel value may or may not exist in the data), `for` loop was presented as the primary structure for traversing an array of fixed size during the course.

The choice of `for` caused about half the students (48%, I10) to use `break`-clause within the loop after encountering the sentinel (`if (array[i] >= sentinel) break;`); `break` was presented in the lectures and the course material. Others used either `return`-clause immediately (23%, I12) or more complex `if / else-if` structures, while a handful of

students altered the index variable's value when the sentinel was found to satisfy the loop condition in the next iteration (I15). Surprisingly, only one student used a logical **AND** expression for loop condition evaluation (I18), which replaced the corresponding **if** inside the loop, as shown in Listing 3. On the other hand, a typical student answer to the traditional *Leap Year* assignment—also used in our course—consists of two or even more conditions combined with logical **AND**s. However, combined conditions were not stressed during the course because in our observation they are prone to errors. That is, students writing combined logical expressions struggle much more than those writing just simple **if** clauses with one condition.

While 33% of the students used **continue** (I11), it was more common to use some other way to handle the values equal to or less than the `lowerLimit`. Even though in our model solution we presented **continue** as the preferable way to skip an unwanted input and move on to the next index, in our course material we in fact discouraged the use of **continue** and **break** in normal situations where these clauses can be easily avoided. One common solution among the students was to wrap the “base case” (valid input) with an **if** structure (I13). Some used combined or nested **ifs** to handle both the sentinel and the `lowerLimit` (thus avoiding both **break** and **continue**), but, as discussed above, combined conditions or nested **ifs** often produced errors in DivZ or Count tasks; the errors are discussed in Section 6.2. All these solutions were “equally good” in terms of marking if they produced the correct result.

In conclusion, the dominance of the **for** loop is probably due to the mental model developed during the course of how to traverse an array (the *walk-through-an-array plan*). The students thus selected the exact form of the loop structure they were most familiar with and adapted the rest of the code based on this selection. In one sense, students meet the expectations of teaching just by following the models presented to them. This familiarity with a particular loop structure is also referenced by Simon [10]. In his study, about 60% of the students who were able to write a reasonable solution used **for** loop and about 25% used **while** loop.

6.1.3 Categories related to unit testing

Students earned one bonus mark by writing unit tests for the Rainfall problem. Sixty-nine students at least tried to write unit tests (I22). We discovered seven categories of tests (I24–I29), and the most common was to test the function with an array that included a sentinel somewhere in the array (I24) or with an array that did not include a sentinel (I25).

While conducting the analysis, we noticed that our categories could have had even more granularity. For instance, based on our emergent coding scheme, we cannot say with certainty if the students who tested with a sentinel (that is, I24) also included valid input, non-valid input, or both in their array, which would have created some new vantage points from the learning objectives' perspective. We will focus on further elaborating the coding scheme in our future studies.

6.2 Error categorization

Appendix B summarizes our coding scheme concerning the students' errors. In the appendix, individual counts in-

dicate the number of students committing the particular error. However, it has to be noted that some error categories are interrelated in a way that affected the counts. For instance, if a student answer indicated a “Count missing” error (E3), he then also committed a “Count not initialized” error (E19), while only the count of the (hierarchically) first category (E3) was increased. Many of the error categories reflect the acknowledged tasks in the Rainfall problem, while we also used the category “Other”. We discuss our data according to these acknowledged tasks below. Several tasks with a low error count (Sum, Neg) are dismissed due to the space limitations.

6.2.1 DivZ

The most common error in composing the plans (different tasks) was guarding against the division by zero (DivZ, E1, and E2 in Appendix B), which accords with the findings in earlier studies. In our case, this error was committed by 60% of all students. Within the DivZ category, the errors fell into two clear groups: ignoring guarding either partially or completely. It was much more common to ignore the guarding completely than partially. A typical way of partial guarding was to check at the beginning if the array was zero-length. Complete guarding is discussed more in Section 6.3.

6.2.2 Count

The second most common error was related to counting valid inputs (Count). Typically, students used `array.Length` as a divisor and ignored the counting completely or used the loop iterator `i` as a divisor after encountering the sentinel or after traversing the whole array. This suggests that some students had problems merging the different plans into one, as proposed by Soloway.

6.2.3 Sentinel

Most of the errors regarding the sentinel task related to the misuse of **continue** and **break** clauses. The most common error (15 students, 11%) was to **continue** iterating when the sentinel was found instead of exiting the loop with a **break**.

6.2.4 Average

The categories E13–E14 comprise mostly errors made in program flow. For instance, one student had placed the counting of the average inside an **else** block, which is never executed. There were only few mistakes related to arithmetics as opposed to Ebrahimi [2] and Simon [10]. One explanation for this is that the data in our question was inside a **double** array, which in our study eliminated the errors related to **int** division.

6.2.5 Other

There were a number of different other error categories (I16–I35) that were not connected to any specific task. The most common was to ignore using parameter values when checking `lowerLimit` and `sentinel` and to use literal values (0 and 999) instead. We assume that this is due to how the assignment text was written: students clung to the thought of literal values that were given *first* and only secondarily to the parameter names that were mentioned in the parentheses. The rest of the categories (with examples) are presented in Appendix B

6.3 Did writing unit tests help students?

Half the students (69 out of 139) wrote at least some Rainfall unit tests (“testers”), and this half scored slightly better on the assignment (72% “pure” score, 78% with the bonus score from test writing) than the half that did *not* write any tests (65% score, “non-testers”). The difference between the testers and non-testers is, however, not statistically significant (t -test, 1-tailed distribution, $p = 0.10$). When looking at the most frequently occurring error, the DivZ error, 40 (58%) out of the 69 testers made the error, while 43 (61%) out of the 70 non-testers made this error, meaning that both groups committed the error almost equally frequently.

Altogether, 48 students (70% of testers, 35% of the exam population) explicitly tested against DivZ error for the cases where the data contains no valid inputs at all (and no sentinel) or where none of the inputs before the sentinel are valid. In fact, the latter also covers the cases where the array is empty. Model solution test cases for these two different situations are given below:

```
Rainfall.Average(new double[] {-1, -2, 0},
0, 999) ~~~ 0;
Rainfall.Average(new double[] {-1, -2, 999,
1}, 0, 999) ~~~ 0;.
```

However, 24 of these 48 students partly or completely failed to implement a solution (guarding) satisfying the tests. This conflicts with the obvious expectation that guarding should have been an easy task after explicitly framing test cases for it. We contacted all 24 students who wrote tests to guard against zero but failed to implement the guarding in their program code. Sixteen of them replied. One problem was that the task of writing unit tests was given as a bonus task and was therefore done *after* the actual solution, which differs from test-first practice. For some students, the exercise layout had thus affected the order in which they wrote their answers. Moreover, regardless of whether the tests were written early or late, the students rushed to proceed with other exam questions and thereby did not remember to make proper modifications that would make the program code (solution) pass the tests. A few students commented that they did not dare to touch the program code afterwards because they thought that they would break something else.

Taken together, we expected to observe some benefits from test writing but rather observed limitations arising from the paper-and-pen examination and how we posed the task of test writing in the exam situation (“something that is considered a bonus and is thus done afterwards”).

6.4 Rainfall as an exam question

In this section, we investigate how the Rainfall problem qualified as an exam question. With respect to the variety of tasks needed to complete the Rainfall problem, one might think that a Rainfall-like question alone would indicate student performance in regard to CS1 learning objectives.

To give context to evaluating the Rainfall question as part of the whole exam, we briefly describe the other exam questions below. All the students were required to answer the Rainfall question (Q1) and to additionally choose three of the six remaining questions. Thus, the Rainfall problem made up to 25% of the total exam score.

- Q1: Rainfall (N=139). Writing tests gave a maximum of one extra mark (N=69).

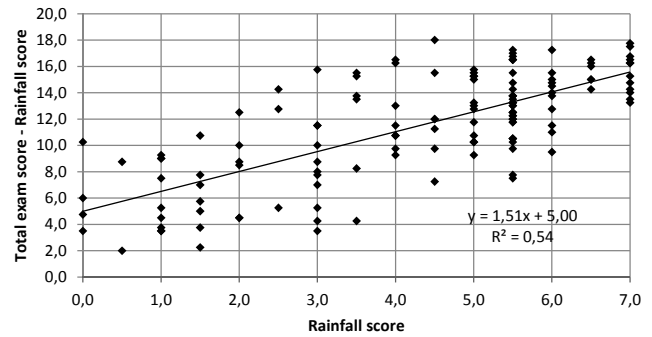


Figure 1: Total exam score (with the Rainfall score subtracted) as a function of Rainfall score.

- Q2: Code trace and theory. E.g., recognize variable and parameter types and names (N=102).
- Q3: Code write, trace, and types. Writing `Main` method using ready-made functions. Writing function signatures with minimal function implementations (N=120).
- Q4: Code trace, explanation, theory. Six small questions about code trace, differences of `i++`, `++i`, hex and binary calculations, recognizing program flow structures, differences of `String` and `StringBuilder`. (N=112)
- Q5: Code modify. Removing repetition from a given code sample and writing documentation (N=48).
- Q6: Code write, program design. Splitting a string to an array and then return the array sorted and writing documentation (N=22). Tests: 1 extra mark.
- Q7: Recursion (N=12). In addition, writing also a non-recursive version gave one extra mark.

The four questions with the highest response frequency can be roughly divided into programming questions (Q1 and Q3) and conceptual questions (Q2 and Q4).

We first investigated if there was a higher correlation between the Rainfall score and the total score as compared to correlations between other questions and the total score. An analysis was thus made between each individual exam question and all the remaining questions at a time. For questions Q1–Q5, the resultant correlations varied from $R^2 = 0.10$ (Q5) to $R^2 = 0.60$ (Q3), with the Rainfall question (Q1) being $R^2 = 0.54$ (see Figure 1). None of the questions alone was particularly better than the others in this analysis¹, which indicates that even if the student performed well in the Rainfall question, he or she did not necessarily achieve good results for the other questions. Through a brief manual inspection of the exam answers, we observed that a few students who passed the exam would have (relatively speaking) failed if the exam scores would have been based only on the Rainfall problem, whereas a few students would have got “too good” of a grade (high marks on Rainfall, low marks on the total exam score).

We continued by combining the scores of two questions and calculating correlations between these pairs and the total exam score (see the second column in Table 1). We were interested in the four questions with the highest response

¹Q5 stood out with a notably lower correlation though.

frequency (see the first column for question pairs in Table 1), while the total score used was the student total regardless of the questions the student had chosen to answer. Q1 combined with Q2 gave the highest correlation with the total exam score, $R^2 = 0.91$. However, almost the same correlation was reached when combining the other two exam questions, with the exception of Q2 combined with Q4, which showed the lowest correlation with the total exam score.

In the next step, we focused purely on the first four questions Q1–Q4, which had the highest response frequency². We ran a correlation analysis between question pairs across the questions Q1–Q4, the results of which are displayed in the last column of Table 1. The Rainfall question (Q1) combined with a question that consisted of code tracing and code explaining (Q2) correlated best with the score for the remaining questions (Q3 and Q4), giving $R^2 = 0.58$. The correlation between the combination of Q1 and Q4 and the combination of Q2 and Q3 was only marginally smaller, at $R^2 = 0.57$. Finally, the pair of Q1 and Q3 compared with the pair of Q2 and Q4 gave $R^2 = 0.39$. Given the nature of our exam questions (Q2 and Q4 were conceptual questions, while Q1 and Q3 emphasized code writing), this rather small correlation supports the idea of having at least two kinds of exam questions: a code writing question, for example, one similar to Rainfall, and a conceptual question, for example, one requiring code tracing and code explaining.

Table 1: The correlation analysis for question pairs.

Q pair	Pair vs. total	Pair vs. Reference Qs
Q1 + Q2	0.91	0.58 (Q3+Q4)
Q1 + Q4	0.90	0.57 (Q2+Q3)
Q3 + Q4	0.88	0.58 (Q1+Q2)
Q2 + Q3	0.87	0.57 (Q1+Q4)
Q1 + Q3	0.87	0.39 (Q2+Q4)
Q2 + Q4	0.80	0.39 (Q1+Q3)

We finally continued by manually investigating all the exam answers of the students who received at least 5 (out of 6) Rainfall points but less than 10 total points for the remaining questions. This sampling yielded 20 exam question answers from 5 students (see the samples in the lower right area in Figure 1). Interestingly, we observed that students showing no difficulties with particular aspects of the Rainfall problem could show considerable difficulties with the same aspects in other kinds of exam questions. We summarize our observations with two examples, as follows.

When looking at how students dealt with parameter passing, we noticed that a student struggled with how to assign argument values to corresponding parameter variables in the function definition. More precisely, the student had written the exact same literal values in the function definition where they should have written variable names. Interestingly, there were plenty of examples in the exam questions themselves as to how to do the parameter passing correctly.

²We hypothesize that this was due to the remaining questions being more demanding and being placed in the last part of the exam.

Here, we interpret the difficulties as being related to knowledge transfer.

Another illustrative example is that the student who committed no errors with variable usage and scoping in the Rainfall problem (E20, in Appendix B) demonstrated difficulties in identifying arrays generally as variables and even in identifying iterators as variables in a code tracing task. Thus, it seemed that the student was able to use “various constructs” but did not conceptually know these constructs were being called variables. Here, we interpret that the student can possess a “working” model that makes it possible to produce working code (cf. success in the Rainfall problem), although the student’s conceptual understandings are still insufficient or developing with respect to many aspects present in the written code. We do not simply refer to “blind coding” learned by mimicking lecture examples. We rather refer to a phenomenon in novice programming where students first learn to *do* programming, while perhaps having difficulties in conceptually naming the programming constructs, and only gradually understanding about *what* they are doing. This observation encourages us to increasingly imbricate the programming and the conceptual tasks in our CS1 learning materials.

Taken together, while we started to explore how the Rainfall problem qualifies as the exam question, we discovered knowledge transfer and knowledge development issues among the novices, ones that merit a separate study with extensive manual analysis of the exam questions. In our interpretation, these issues might explain the fact that we could not see clear differences in how individual questions “qualified” through correlation analyses (cf. knowledge transfer challenge across different kinds of questions) and that there were rather small correlations between conceptual questions and programming questions (cf. a working model for programming can be underpinned by in-progress/insufficient conceptual understandings).

We also speculate that, for various reasons such as a learning style or insufficient skills, the student *preferring* conceptual questions over programming questions may have focused on the conceptual ones—and vice versa.

7. CONCLUSIONS

We investigated CS1 students’ exam performance on a Rainfall problem—a programming assignment originally highlighted by Soloway. Students’ solutions indicated their familiarity with particular loop constructs, while, in line with previous studies, the single most common error concerned division by zero (DivZ). Regardless of the large set of program errors that we identified, the average Rainfall score was 68.8%, which is more than the corresponding score for the total exam. The students had difficulties with guarding against DivZ, even though they wrote tests for it. As reported by the students, the challenge of not being able to benefit from test writing originated from the limitations of the pen-and-paper examination and the exam situation itself. Finally, when we explored how the Rainfall problem qualified as an exam question, we found that some novice students had difficulties with knowledge transfer across the exam questions and that their conceptual understandings of program constructs were insufficient or in-progress, regardless of them being able to write correct code.

Almost all students used a “Single Loop” implementation approach, which we assume was due to our teaching prefer-

ences and the mental model that was constructed during the course for processing data that is contained in an array. In short, students met the expectations of teaching by following what was presented to them. We think that different implementation strategies could be discussed during the course, keeping in mind the possible “natural” approaches to the programming language and the paradigm used. In that way, we could reinforce students’ ability to choose their approach in an informed manner and with rationale. We are referring to small steps in students’ personal epistemology (cf. [7]) during CS1. In so saying, we acknowledge the high cognitive load typically imposed by CS1 on students and think conceptual discussion and increased awareness can decrease confusion and uncertainty to a certain extent and can thus counteract the heavy cognitive load.

Even though DivZ (the most common student error) can be understood as one error among others, we think it should also be seen more broadly from the perspective of the function of being able to handle input. For instance, a large proportion of information security issues originate in the attacker exploiting the shortcomings of the code that has not been designed to handle a certain kind of input and at the worst gives the attacker full control of the software. In the long run, instead of memorizing a list of special cases, such as DivZ, it is more important for the students to learn to apply the principle that not all input is “neat” and that their program must be able to handle the situation even if it is fed with erroneous input. Besides DivZ, this phenomenon was visible with the Count task, as students often failed to skip the unwanted input.

We have attempted to acquaint the students with the connection between function input space and expected function behavior using a test-driven development approach. However, it seems that even with writing the tests, students are still making errors in the DivZ and Count tasks. On the other hand, students hesitated rewriting code according to the tests in the paper-and-pen exam situation. We hypothesize that if the students *did* have the opportunity to write the code using the computer, they would be less prone to certain errors. At the least, those who explicitly write (“thought out and written down”) unit tests related to DivZ would be able to identify erroneous code after running the tests without the fear of breaking the part that they consider completed. These conclusions have motivated and informed our ongoing experimental research on the exam format.

Our interpretations of novice students’ insufficient conceptual understandings about programming constructs reflect our course setting with a strong doing mode and informs our teaching. Regardless of the doing mode and the presence of attractive game programming, we must remember to include a sufficient number of conceptual tasks (tracing and explaining program code) *during* and *throughout* the course. Generally, our conclusion (see Section 6.4) that the exam should include at least one versatile programming question and one versatile conceptual question harmonizes with the folklore of CS1 exam designs; see, for example, the study by Sheard et al. [9].

Should the CS1 educators use long-standing exam questions, it is recommended that they review them in detail in their particular local contexts. We hope that our particular exploration informs CS teachers regarding their material and exam design.

8. REFERENCES

- [1] N. Dale. Content and emphasis in CS1. *SIGCSE Bull.*, 37(4):69–73, 2005.
- [2] A. Ebrahimi. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies*, 41(4):457–480, Oct. 1994.
- [3] K. Fisler. The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research, ICER ’14*, pages 35–42, New York, NY, USA, 2014. ACM.
- [4] V. Isomöttönen, A.-J. Lakanen, and V. Lappalainen. K-12 game programming course concept using textual programming. In *Proceedings of the 42nd ACM technical symposium on Computer science education, SIGCSE ’11*, pages 459–464, New York, NY, USA, 2011. ACM.
- [5] V. Isomöttönen and V. Lappalainen. CSI with games and an emphasis on TDD and unit testing: piling a trend upon a trend. *ACM Inroads*, 3(3):62–68, Sept. 2012.
- [6] A.-J. Lakanen and V. Lappalainen. What Students Think About Game-Themed CS1. In M. Koskela and K. Heikkinen, editors, *Federated Computer Science Event 2014*, volume 24, pages 19–22, Lappeenranta, Finland, 2014. Lappeenranta teknillinen yliopisto, LUT Scientific and Expertise publications.
- [7] R. McDermott, I. Pirie, A. Cajander, M. Daniels, and C. Laxer. Investigation into the personal epistemology of computer science students. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE ’13*, pages 231–236, New York, NY, USA, 2013. ACM.
- [8] S. Reges. The mystery of “b := (b = false)”. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, SIGCSE ’08*, pages 21–25, New York, NY, USA, 2008. ACM.
- [9] J. Sheard, Simon, A. Carbone, D. Chinn, M.-J. Laakso, T. Clear, M. de Raadt, D. D’Souza, J. Harland, R. Lister, A. Philpott, and G. Warburton. Exploring programming assessment instruments: A classification scheme for examination questions. In *Proceedings of the Seventh International Workshop on Computing Education Research, ICER ’11*, pages 33–38, New York, NY, 2011. ACM.
- [10] Simon. Soloway’s rainfall problem has become harder. In *Learning and Teaching in Computing and Engineering (LaTiCE), 2013*, pages 130–135. IEEE, Mar. 2013.
- [11] Simon. Personal communication, Dec. 2014.
- [12] E. Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.
- [13] A. Venables, G. Tan, and R. Lister. 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 Workshop, ICER ’09*, pages 117–128, New York, NY, USA, 2009. ACM.

APPENDIX

A. IMPLEMENTATION CATEGORIES

Table 2: Implementation categorization

Id	Category description and possible example(s)	Approach	Count
I1	Single Loop—Iteratively consumes inputs, increments sum and count on each non-negative input, then checks for division by zero and computes the average upon reaching the sentinel.	High-Level	129
I2	Clean First—Iteratively adds non-negative inputs to a data structure until the sentinel is reached. It then counts and sums the cleaned data, checks for division by zero, and computes the average.		4
I3	Clean Inside Single Loop—Iteratively consumes inputs, and removes unwanted input from the original data structure		3
I4	Clean Multiple—Traverses the input twice, once to count non-negative inputs and once to sum them. Each traversal terminates at the sentinel. It then checks for division by zero and computes the average.		1
I5	Unclear—Generally mangled code with no clear structure.		2
I6	Used <code>for</code> loop.	Loop	118
I7	Used <code>foreach</code> loop.		9
I8	Used <code>while</code> loop.		9
I9	Used <code>do-while</code> loop.		1
I10	Used <code>break</code> in loop.	Flow control	67
I11	Used <code>continue</code> in loop.		46
I12	Used <code>return</code> when hit sentinel. E.g. , <code>if (t[i] >= sentinel) return average;</code>		32
I13	Sum is counted inside <code>if</code> block. E.g. , <code>if (t[i] >= lowerLimit) { sum += t[i]; count++;}</code>	Algorithm	58
I14	Average counted inside the loop average counted every time sum is counted (does not need guard for division by zero).		24
I15	Change loop variable value when hit sentinel E.g. , <code>if (t[i] >= sentinel) i = t.Length;</code>		5
I16	Corrected count when hit lowerLimit E.g. , <code>if (t[i] <= lowerLimit) limitsFound++ ... ka = ka / (i + 1 - limitsFound).</code>		3
I17	Sum is corrected after hitting lowerLimit or sentinel.		3
I18	Used logical AND in <code>for</code> loop. E.g. , <code>for (...; i < t.Length && t[i] != sentinel; ...)</code>		1
I19	Used an extra array variable E.g. , <code>int[] rainfalls = array;</code>	Other	3
I20	Used <code>DivideByZeroException</code> .		3
I21	Wrote down the algorithm without implementing it in the code.		2
I22	Test cases (one or many) written, may contain syntax errors.	Tests	69
I23	Test cases syntactically correct.		30
I24	Tested with sentinel E.g. , <code>{1,-2,3,999,4,5}</code> . E.g. , <code>{999}</code>		61
I25	Tested without sentinel E.g. , <code>{1,2,3}</code> . E.g. , <code>{1,-2,3}</code>		60
I26	Tested against division by zero. When sentinel found not a single value counted in the average. E.g. , <code>{-1, -1, 999, 2, 3, 4}</code> .		40
I27	Tested against division by zero with an empty array.		32
I28	Tested with negative values E.g. , <code>{1,-2,3,999}</code>		25
I29	Tested against division by zero. Ehen hit the end of the array not a single value counted in the average E.g. , <code>{-1, -1, -2}</code>		20

B. ERROR CATEGORIES

Table 3: Error categorization. The same student answer may have indicated errors in many categories

Id	Category description and possible example(s)	Task	Count
E1	Checking against DivZ totally missing.	DivZ	66
E2	Partly missing (many <code>returns</code>). E.g. , Handled after the loop, but not when hit sentinel		17
E3	Count missing. E.g. , Used <code>t.Length</code> or in-scope loop counter variable as the divisor	Count	42
E4	Erroneous count. E.g. , Adding <code>count</code> variable in wrong place. Modified loop iterator variable to keep track of count.		17
E5	Division by loop counter variable that is not in scope.		10
E6	Sentinel is “observed” but loop does not terminate. E.g. , <code>if (t[i] >= upperLimit) continue;</code>	Sentl	15
E7	Breaks out of the loop when encountered the <code>lowerLimit</code> .		2
E8	Ends iterating with <code>t[i] == upperLimit</code> instead of <code>t[i] >= upperLimit</code>		2
E9	Sentinel is not observed in any way		1
E10	Sentinel counted in sum. E.g. , Conditional for checking <code>upperLimit</code> is too late.	Sum	7
E11	Sum counted wrong or not counted at all. E.g. , <code>ka = t[i] + t[i++]</code>		5
E12	<code>lowerLimit</code> included in sum (but not in count)		1
E13	Return average that is not counted or is counted incorrectly. E.g. , Division is made in <code>else</code> block which is not executed necessarily.	Avg	8
E14	Division missing. E.g. , A variable for average may be declared but only sum is counted.		3
E15	Negative values are counted in sum. E.g. , <code>lowerLimit</code> is observed in <code>if</code> but flow in loop body is incorrect. Missing, e.g., <code>else</code> block or <code>continue</code> .	Neg	9
E16	Used constant value for sentinel (19 students) or <code>lowerLimit</code> (16 students) instead of parameter.	Other	20
E17	<code>return</code> wrong or missing. E.g. , Various mistakes when returning the average, for example, extra <code>return</code> .		16
E18	Index out of bounds. E.g. , <code>while (t[i] < upperLimit)</code>		12
E19	Count or sum variables not initialized		11
E20	Used out-of-scope variable. E.g. , <code>rainfalls</code> instead of <code>array</code> , <code>rainfalls</code> variable appears in <code>Main</code> .		10
E21	Wrong syntax in loop declaration.		9
E22	Other syntactic errors. E.g. , Invalid <code>if</code> comparisons. (4) E.g. , Confusion with logical <code>AND</code> / <code>OR</code> (4). E.g. , Used non-array variables as arrays. (1)		9
E23	Wrong type for sum (should be <code>double</code>)		8
E24	Incrementing the looping variable incorrectly. E.g. , <code>if (t[i] <= lowerLimit) i++;</code> inside <code>for</code> .		8
E26	Variable type re-declared.		7
E27	Extracted an item from array incorrectly. E.g. , <code>i</code> or <code>[i]</code> instead of <code>t[i]</code> .		6
E28	Wrong type used when extracting an item from the array. E.g. , item is <code>int</code> , should be <code>double</code> because array is <code>double[]</code>		6
E29	No pertinent code. May be syntactically correct, but algorithmically makes no sense.		6
E30	Average/count declared or initialized incorrectly. E.g. , <code>avg = ...</code> instead of <code>double avg = ...</code> E.g. , <code>avg = t[0]</code> (first item could be sentinel).		5
E31	Endless loop. E.g. , <code>while</code> inside <code>for</code> loop.		5
E32	Parameters re-initialized		5
E33	Confusion with using loop counter variable with the array. E.g. , <code>t[]</code> or <code>[]t</code> instead of <code>t[i]</code> .		4
E34	Does not work if no sentinel, loop ends <i>only</i> when sentinel observed		3
E35	Loop counter variable(s) not initialized or used counter that does not exist.		3