Engineering Educators Bringing the World Together
2025 ASEE Annual Conference & Exposition
Palais des congrès de Montréal, Montréal, QC • June 22–25, 2025 ASEE
Paper ID #47142

Optimizing auto-graded programming activities: A data-driven approach for presenting assessments in a scaffolded format.

Jamie Emily Loeber, zyBooks, A Wiley Brand

Jamie Loeber is an Assessment Specialist at zyBooks, a Wiley Brand. She earned her B.S. in Computer Science at the University of California, Irvine. She has taught programing and machine learning to students across the globe. Jamie is passionate about improving computer science education and creating better learning experiences in STEM.

Ms. Efthymia Kazakou, zyBooks, A Wiley Brand

Efthymia Kazakou is Sr. Assessments manager at zyBooks, a startup spun-off from UC Riverside and acquired by Wiley. zyBooks develops interactive, web-native learning materials for STEM courses. Efthymia oversees the development and maintenance of all zyBo

Dr. Yamuna Rajasekhar, zyBooks, A Wiley Brand

Yamuna Rajasekhar is Director of Content, Authoring, and Research at zyBooks, a Wiley Brand. She leads content development for the Computer Science and IT disciplines at zyBooks. She leads the authoring and pedagogy team at zyBooks, developing innovative learning solutions that drive measurable student success. She is also an author and contributor to various zyBooks titles. She was formerly an assistant professor of Electrical and Computer Engineering at Miami University. She received her M.S. and Ph.D. in Electrical and Computer Engineering from UNC Charlotte.

Nicole Kehaulani Collins, zyBooks, A Wiley Brand

Nicole Collins is an Author Trainer and former Assessment Specialist at zyBooks, a Wiley Brand. She earned her B.S. in Computer Science and her M.Ed. in Learning, Design & Technology from UNC Charlotte. Her professional interests include computing education, online learning, educational technology, instructional design, curriculum development, and DEI in STEM. Nicole is passionate about creating engaging and effective learning experiences for students, leveraging her expertise in instructional design and technology to enhance educational outcomes for STEM disciplines.

Dr. Annie Hui, zyBooks, A Wiley Brand

Annie Hui is a zyBooks assessment specialist. She has 15 years of experience teaching computer science, information technology, and data science courses, in both in-person and online modes. She has taught in Northern Virginia Community College and George Mason University. She specializes on course design to maximize student engagement and success.

Optimizing auto-graded programming activities: A data-driven approach for presenting assessments in a scaffolded format

Abstract

Research has shown that in introductory programming courses, breaking complex concepts into smaller, manageable units is highly effective. Additionally, using scaffolding techniques helps learners progressively develop programming skills. However, determining the appropriate size of each conceptual unit depends on factors such as the learners' aptitude and experience.

In this paper, we present a data-driven approach to designing auto-graded activities in our online, interactive STEM textbooks, focusing on effectively breaking down complex concepts into smaller, more achievable steps for learners. We analyzed two types of activities: 1) activities on challenging topics as reflected by high struggle rates and 2) activities on introductory topics with lower struggle rates, but where students still needed assistance based on their feedback and incorrect submissions as they began learning programming. For both types of activities we examined multiple metrics such as students' average completion rates and common errors.

Based on these insights, we further refined the activities by dividing them into smaller components and measured the impact on student struggle rates. By comparing the metrics before and after these changes, we identify key best practices for designing and improving auto-graded programming problems, aimed at enhancing student learning outcomes in programming courses.

Introduction

Classroom-based learning can be distilled into a three-step process: 1) the instructor introduces the materials to be learned, 2) the learner completes assessments to demonstrate their understanding, and 3) the instructor provides feedback based on the learner's performance. When steps two and three are repeated in an iterative cycle, the learner’s success typically improves with each iteration. This cycle of assessment and feedback forms the basis of effective learning [1], [2].

However, in today’s classroom, a few common challenges exist that hinder the learner’s success. A common issue is the disconnect between the materials and the assessments, leading to struggles and the inability to complete the assessments. Another frequent problem is the lack of timely and suitable feedback on the learners' assessments which prevents learners from identifying and correcting their mistakes. These issues are particularly common in STEM education and can cause frustration, anxiety, and low performance that may eventually result in increasing class drop rates [3], [4]. Fortunately, these challenges can be mitigated by improving the design of the course material and with the assessments, and along with improving feedback provided to students to better meet their needs.

In the next two sections, we explore key factors from the literature that can enhance the design of course materials and their assessments. Additionally, we outline the motivation behind this study and introduce our approach to authoring and improving course content.

Scaffolding

An important characteristic of an effective learning activity includes an appropriate level of difficulty that builds upon prior knowledge through scaffolding. Scaffolding is an instructional approach that involves breaking down learning tasks into smaller, more manageable pieces and providing support at each step. In the context of introductory programming courses, scaffolding helps students build their skills incrementally by gradually increasing the complexity of programming tasks. Scaffolded activities present problems in a step-by-step manner, where each step builds upon the previous one. Research indicates that this method is highly effective in designing homework assignments, as it helps students retain concepts more effectively [5] - [8]. By significantly reducing the mental effort required to process information (known as "cognitive load") [9], scaffolding increases student interest and learning potential.

By using scaffolding, educators can reduce cognitive load and struggle, thereby increasing student interest and learning potential. This approach not only helps students understand complex programming concepts but also boosts their confidence and motivation to pursue further studies in computer science [10].

Feedback

In addition to scaffolding, an effective learning activity incorporates timely and constructive feedback that is both immediate and clear. Timely feedback allows students to address mistakes while the material is still fresh in their minds, maximizing the opportunity for learning. Constructive feedback provides step-by-step explanations of the expected answers, using clear language to guide students in identifying the specific areas of their mistakes. This approach encourages students to discover and correct errors on their own without revealing the solution outright. Furthermore, such feedback helps address student confusion, misconceptions, and recurring gaps by clarifying difficult concepts and providing targeted guidance where needed most [11], [12]. Together, these elements reinforce understanding, boost students' confidence, and ensure a deeper grasp of the material.

Motivation

This paper presents a case study on an assessment design aiming to optimize learners’ success through a continuous process of monitoring feedback, identifying common mistakes, pitfalls, and misconceptions and iteratively revising assessments. Assessment revision includes better scaffolding, more tailored feedback, and the use of code templates. In this paper, the effectiveness of the described design is evaluated by comparing the learners’ total failure rates and completion rates before and after multiple cycles of design improvement. This study shows that designing assessment as a continuous process that adapts to learners’ needs is vital to learners’ success.

Challenge Activities

Hands-on practice is essential for learning programming in CS1. Well-designed practice activities increase student engagement by fostering interaction between the student and the content [12], [13]. Common types of practice include multiple-choice questions, code ordering, code output prediction, and code writing. Among these, code writing provides the greatest

opportunity for students to recall concepts and apply their knowledge to solve programming problems. Code writing activities can vary in scale, with small-scale exercises requiring students to complete a program with only a few words or lines of code.

In our online interactive programming textbooks, Challenge Activities (CAs) are mastery-based assessments, consisting of a series of auto-graded, randomized questions, referred to as "levels". Each level is scaffolded to increase in difficulty, requiring students to complete code snippets correctly to progress. Randomization plays a key role in enhancing the learning experience, as each retry presents students with a unique problem. This approach prevents memorization of answers and fosters a deeper understanding of the material.

This paper focuses on improving CodeWriting CAs (Figure 1) which are designed to assess a student's mastery in programming by providing incomplete code that a student must complete given the activity's prompt. The prompt shows one or more examples of the expected output/result depending on the different cases examined so that the expected behavior and output of the code is clear to the student. An explanation is provided upon submission, regardless of correctness, to guide students in approaching the problem. CodeWriting CAs are highly randomized, meaning that the activities support three different forms of randomization: meaningful, cosmetic, and test cases [14].

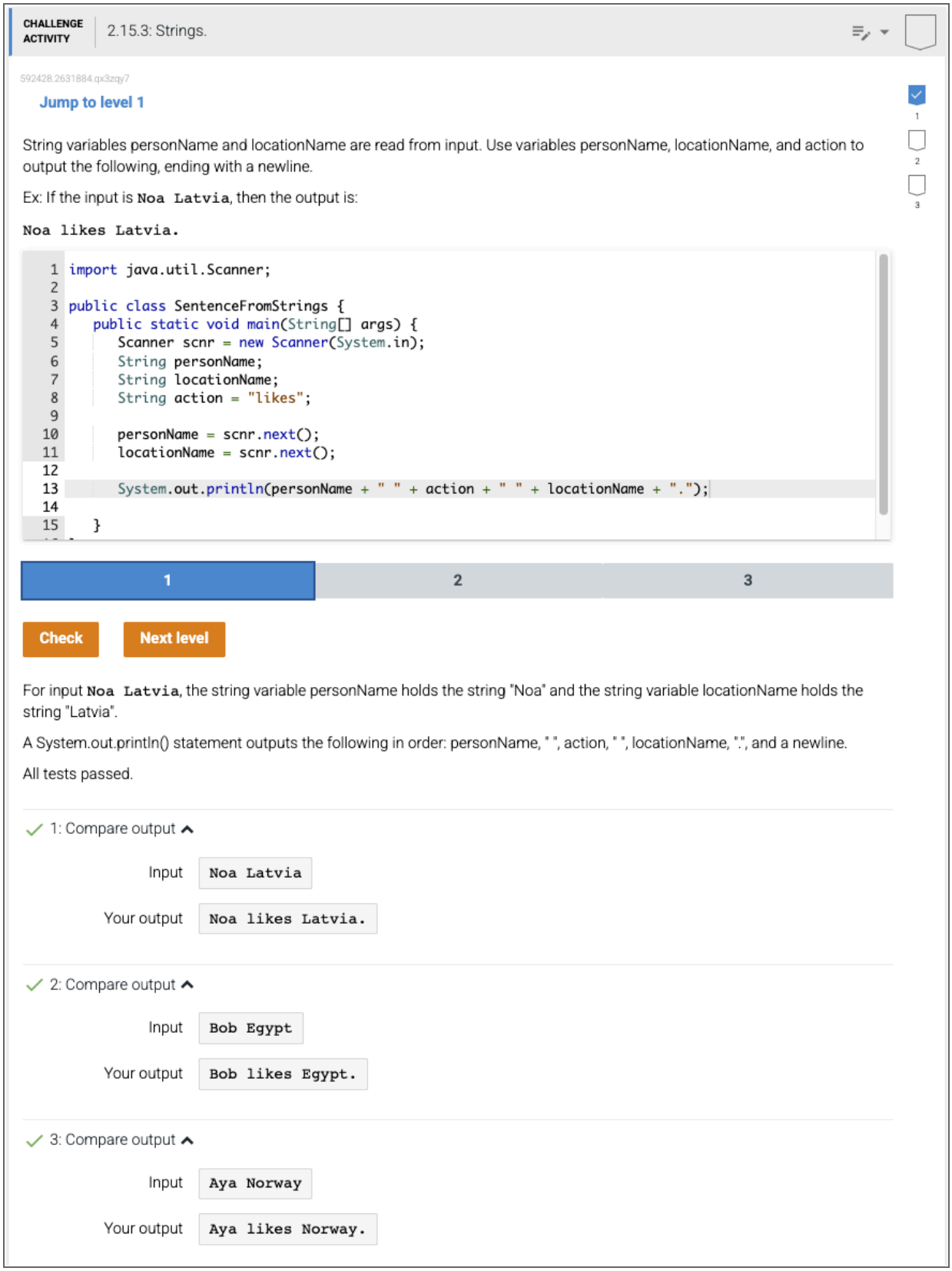

Figure 1. CodeWriting CA with 3 levels.

Methods and Metrics

In this section, we describe our data collection process, explain how we identified the CAs in need for revision, and provide the metrics that measure the effectiveness of the revisions.

Once published, a CA is monitored regularly for its usage and user feedback. Usage data include passing and failure rates as well as the answers submitted by users. User feedback includes bug reports and comments submitted by instructors, students and authors. A number of CAs were identified for revision in 2023. Data on students' performance and feedback from 2022 to 2024 were collected to study the impact of these revisions.

Revision Triggers

Revision priority is given to bug reports, users' comments, and failure rate statistics in this order. In the year 2023, the majority of the revisions were triggered by bug reports and reports on users' comments. Table 1 shows the various types of events that triggered updates.

Table 1. Events that triggered CA revisions.

Event categories	Number of CAs revisions triggered
Bug reports	14
Weekly reports of users' comments	9
Bi-annual failure rate statistics	3

To investigate the impact of improvements on auto-graded CAs, 26 revisions were conducted on 21 CAs in the year 2023. These activities were based on two criteria: 1) their difficulty, as determined by high failure rates and student feedback, and 2) their importance in teaching foundational programming concepts.

In 2024, we re-examined the struggle data to identify any improvements in learners' performance. For these CAs, we reviewed the comments and discussions on the platforms we use to view and address feedback (Trello, GitHub, and Wrike), and documented the changes made to address the struggles expressed by students in the bug reports and user feedback comments. After documenting the changes for each CA, the team noted common actions taken, which we categorized into six primary types of changes:

1. **Scaffolding**: Activities were divided into smaller, manageable components to reduce cognitive load and help students focus on one aspect of a problem at times.
2. **Simplified content**: Redundant of extraneous information was removed to minimize unnecessary cognitive load.
3. **Added hints:** Comments of syntax reminders were introduced to provide students with immediate guidance.

4. **Detailed prompts and explanations:** Instructions and explanations were rewritten for clarity and depth to improve students' understanding of the task.
5. **Better scenarios:** Problem contexts were adjusted to be more realistic or relatable, helping students connect the problem to real-world applications.
6. **Other changes:** These included updates such as fixing typos, changing fonts for readability, and correcting provided code.

These six categories were used as a metric, as they summarize the changes to improve the CAs displaying high failure rates.

Revision Frequencies

The frequency of each type of revision is shown in Table 2.

Table 2. Frequency of each revision type.

Revision Category	**Frequency**
Scaffolding	5
Simplified	8
Added hints	4
Detailed prompts/Explanations	15
Better scenarios	4
Other	7

To evaluate the impact of these changes, we compared metrics from two periods, before and after the revisions. The metrics included are:

- **Number of attempts:** The total number of students attempting the CA.
- **Level failure rates:** The percentage of students failing at each level of the CA.
- **Total failure rates**: The percentage of students who attempted the CA and failed at Level 1, and thus could not proceed to higher levels of the CA.
- **Completion rates**: The percentage of students who attempted and passed all levels of the CA and thus completed the CA.
- **Improvements:** Reduction in failure rates and increases in completion rates across the periods.

In the next section, we consider three case studies of these revisions. These three cases are chosen to illustrate common challenges students encounter at the beginning, middle, and end of a CS1 course.

Case study #1: Random numbers are often covered at the beginning of a CS1 course as an application of one essential operator, namely, the modulo. The concepts of integer division and reminder are not novel to students entering CS1. However, the novel use of the remainder in integer division poses unique challenges to students when this technique is applied to random number generation. Case study #1 reveals the struggles encountered by such students and the remedy that helps them succeed.

Case study #2: While loops are generally introduced in the middle of a CS1 course and are often considered a major component of logical reasoning. The concept of the while loop is relatively simple compared to other programming constructs, even the for loop. However, the while loop's versatility often makes problem solving very challenging to students at this stage. Case study #2 explores common struggles that students encounter on the while loops and reveals that the remedy is found in guiding the students through their thinking process.

Case study #3: File input is an advanced topic usually covered at the last stage of CS1 because this topic requires the understanding of various flags and different orders of operation involved in handling files. The large amount of detailed information poses a daunting challenge to students. Case study #3 shows how this complex topic can be decomposed into smaller meaningful pieces to help students connect the details together progressively.

Case Studies and Results

Overall Results

Figure 2 illustrates each programming activity and the overall completion rates before and after revision. Figure 3 illustrates each programming activity and the failure rates for each activity. For each CA that went through the revision, the level failure rate is reduced while the corresponding completion rate increases.

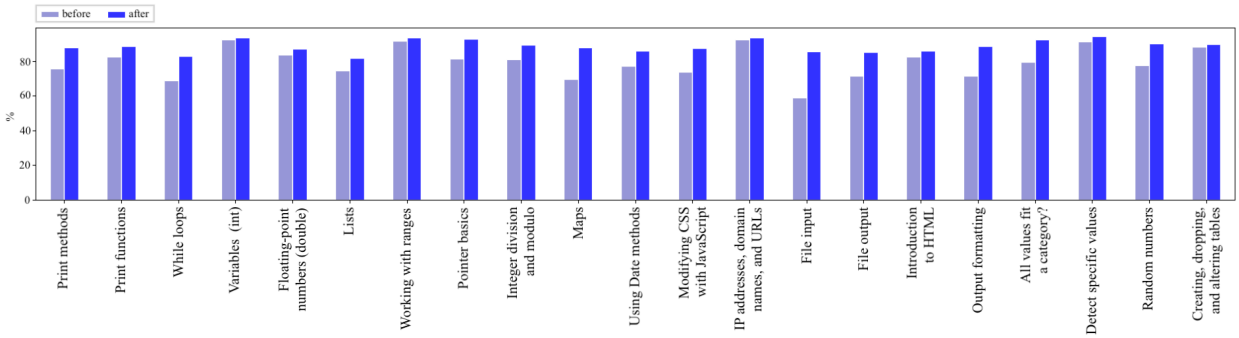

Figure 2. Overall completion rates of all updated CAs before and after revision.

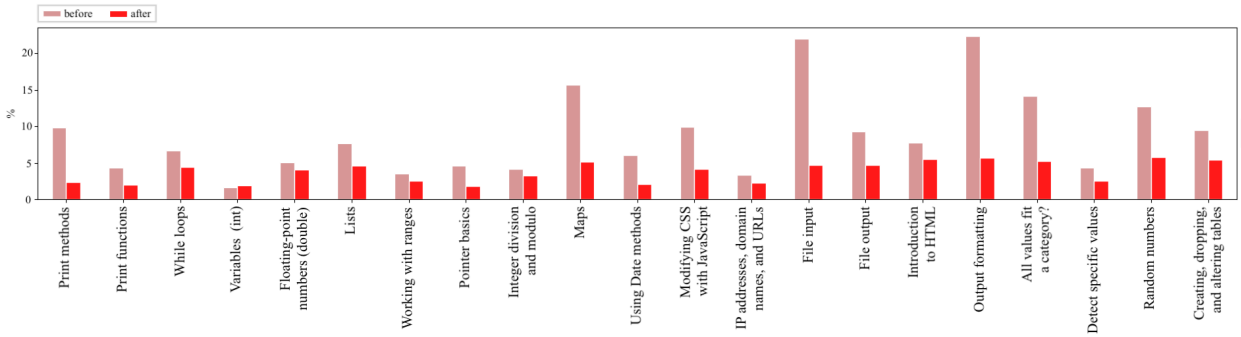

Figure 3. Total failure rates for each programming activity before and after the updates of students who did not complete Level 1.

In the following three sections, we will present three case studies of CAs that have been enhanced, highlighting the changes made and their impact on failure rates.

Case study #1: Random numbers

The Random Numbers activity was designed with three levels of increasing complexity with the following objectives:

- **Level 1:** Use rand() and modulo arithmetic to generate random numbers from 0 to a given bound.
- **Level 2:** Use rand() modulo arithmetic, and addition/subtraction to generate a random number in an adjusted range.
- **Level 3:** Use srand() to seed the pseudorandom number generator with a given input and generate random numbers in a given range.

Over three semesters of Winter 2022, Spring 2023, and Fall 2023, the statistics of students' performance on this activity is summarized in Table 3.

Table 3. Students' performance on the original Random Numbers activity.

	Level 1	**Level 2**	**Level 3**
Students attempted	8061	6604	6006
Students failed	1343	451	383
Students failed (%)	16.660	6.829	6.377

Further investigation showed that many students were having trouble understanding what the problem was asking. To combat this issue, the problem was simplified and reworded to reduce the cognitive load. The changes introduced a restructured problem with reduced cognitive load, enabling students to better comprehend the task and arrive at a solution.

However, the primary cause of struggle did not appear to solely be the prompt. The most difficult part of Level 1 that students seemed to struggle with was using the modulus operator (%) with a variable such as largestVal as seen in Figure 3.

Given integer variables seedVal and largestVal, generate four random numbers that are less than largestVal and greater than or equal to 0. Each number generated is output. Lastly, the average of the four numbers is output.

Ex: If largestVal is 14, then a possible output is:

```
6
8
9
10
Average: 8.3
```

Note: The input seeds are large integers that approximately equal the time in seconds since January 1, 1970.

```
12    int val4;
13
14    cin >> seedVal;
15    cin >> largestVal;
16
17    srand(seedVal);
18
19    /* Your code goes here */
20
21    cout << val1 << endl;
22    cout << val2 << endl;
23    cout << val3 << endl;
24    cout << val4 << endl;
25    cout << "Average: " << fixed << setprecision(1) << ((val1 + val2 + val3 + val4) / 4.0) << endl;
26
27    return 0;
28 }
```

Solution:

val1 = rand() % largestVal;
val2 = rand() % largestVal;
val3 = rand() % largestVal;
val4 = rand() % largestVal;

Figure 3. A sample question and solution of Level 1 of the Random Numbers activity.

To address this struggle, a new Level 1 was created to improve the scaffolding of the CA, with the goal of better preparing students for the transition to using the modulus operator with a variable. The new Level 1 is identical to the original Level 1, but uses a constant instead of a variable as the upper limit. The question added as the new Level 1 is shown in Figure 4.

Integer seedVal is read from input. srand() is called with seedVal as the seed. Use rand() to assign variables val1, val2, and val3 each with a random number generated between 0 and 23, both inclusive.

▼ Click here for example

Ex: One possible output is:

```
14
13
11
Sum: 38
```

```
   int seedVal;
   int val1;
   int val2;
   int val3;
   int sum;

   cin >> seedVal;

   srand(seedVal);

   /* Your code goes here */

   sum = val1 + val2 + val3;

   cout << val1 << endl;
```

Solution:

val1 = rand() % 24;
val2 = rand() % 24;
val3 = rand() % 24;

Figure 4. A sample question and solution of the new Level 1 of the Random Numbers activity.

Since a new Level 1 was added to this CA, the remaining levels shifted up. The original Level 1 became Level 2, the original Level 2 became Level 3, and the original Level 3 was ultimately dropped (discussed elsewhere).

The level now known as Level 2 was revised with prompt updates for clarity of the program's expected behavior. Level 2 of the revised activity is shown in Figure 5.

Integers seedVal and boundValue are read from input. srand() is called with seedVal as the seed. Use rand() to assign variables dataPoint1, dataPoint2, and dataPoint3 each with a random number generated between 0 and boundValue - 1, both inclusive.

▼ Click here for example

Ex: If boundValue is `18`, then one possible output is:

```
15
0
14
Sum: 29
```

```
11   int sum;
12
13   cin >> seedVal;
14   cin >> boundValue;
15
16   srand(seedVal);
17
18   /* Your code goes here */
19
20   sum = dataPoint1 + dataPoint2 + dataPoint3;
21
22   cout << dataPoint1 << endl;
23   cout << dataPoint2 << endl;
24   cout << dataPoint3 << endl;
25   cout << "Sum: " << sum << endl;
26
```

Solution:

dataPoint1 = rand() % boundValue;
dataPoint2 = rand() % boundValue;
dataPoint3 = rand() % boundValue;

Figure 5. A sample question and solution of the new Level 2 of the Random Numbers activity.

Level 2 of the original activity (Level 3 in the revised activity) had similar issues as the previous level. The problem focused on the given inputs but introduced a complex question requiring implicit knowledge, such as calculating a random number between two values. Level 2 of the original activity is shown in Figure 6.

Given integer variables seedVal, smallestInput, and greatestInput, output a winning lottery ticket consisting of four numbers in the range smallestInput to greatestInput inclusive. End each output with a newline.

Ex: If smallestInput is 30 and greatestInput is 85, then a possible output is:

```
41
63
51
35
```

```
3  using namespace std;
4
5  int main() {
6     int seedVal;
7     int smallestInput;
8     int greatestInput;
9
10    cin >> seedVal;
11    cin >> smallestInput;
12    cin >> greatestInput;
13
14    srand(seedVal);
15
16    /* Your code goes here */
17
18    return 0;
19 }
```

Solution:

```
cout << (rand() % (greatestInput - smallestInput + 1)) + smallestInput << endl;
cout << (rand() % (greatestInput - smallestInput + 1)) + smallestInput << endl;
cout << (rand() % (greatestInput - smallestInput + 1)) + smallestInput << endl;
cout << (rand() % (greatestInput - smallestInput + 1)) + smallestInput << endl;
```

Figure 6. A sample question and solution of the old Level 2 of the Random Numbers activity.

This is a significant leap in difficulty from the original Level 1, which overwhelmed students who were still new to the material. To address the students' difficulty with the original Level 2, the level (now Level 3) had two changes: move outputting the result to the given code so the student can focus on the level's task, and add in a note in the prompt to assist with assumed prerequisite knowledge. Level 3 of the revised activity is shown in Figure 7.

Integers seedVal, lowBound and upBound are read from input. srand() is called with seedVal as the seed. lowBound and upBound represent the range of numbers that can be picked on a lottery ticket. Assign variables attempt1 and attempt2 each with a random number between lowBound and upBound, both inclusive.

▼ Click here for example

Ex: If lowBound is `5` and upBound is `65`, then one possible output is:

```
49
35
```

Note: `(rand() % N) + K` yields a random number ranged from K to K + N - 1, both inclusive. If the lower limit is lowBound and the upper limit is upBound, then N is upBound - lowBound + 1 and K is lowBound.

```
7   int lowBound;
8   int upBound;
9   int attempt1;
10  int attempt2;
11
12  cin >> seedVal;
13  cin >> lowBound;
14  cin >> upBound;
15
16  srand(seedVal);
17
18  /* Your code goes here */
19
20  cout << attempt1 << endl;
21  cout << attempt2 << endl;
22
```

Solution:

attempt1 = (rand() % (maxValue - minValue + 1)) + minValue;
attempt2 = (rand() % (maxValue - minValue + 1)) + minValue;
attempt3 = (rand() % (maxValue - minValue + 1)) + minValue;

Figure 7. A sample question and solution of the new Level 3 of the Random Numbers activity.

The last level of the original CA, Level 3, assessed students' mastery of how to use srand() to generate a random seed, generate multiple random numbers within certain bounds, and use the generated results to calculate some basic arithmetic. While this level was intended to assess overall mastery of random number generation, this activity was recognized to be difficult to students, due to the increase of student feedback on this activity. The initial structure of the level is shown in Figure 8.

Given integer variables seedVal, lowestDiningTable, and highestDiningTable, seed the pseudorandom number generator with seedVal. Then, output two random table sizes in the range lowestDiningTable to highestDiningTable inclusive. Lastly, given a total of 104 people, output the number of people that are *not* at a table. End each output with a newline.

Ex: If lowestDiningTable is 10 and highestDiningTable is 19, then a possible output is:

```
15
17
72
```

Note: The number of people that are *not* at a table is 104 - table1 - table2.

```
4
5  int main() {
6     int seedVal;
7     int lowestDiningTable;
8     int highestDiningTable;
9     int table1;
10    int table2;
11    int table3;
12
13    cin >> seedVal;
14    cin >> lowestDiningTable;
15    cin >> highestDiningTable;
16
17    /* Your code goes here */
18
19    return 0;
20 }
```

Solution:

```
srand(seedVal);

table1 = (rand() % (highestDiningTable - lowestDiningTable + 1)) + lowestDiningTable;
table2 = (rand() % (highestDiningTable - lowestDiningTable + 1)) + lowestDiningTable;
table3 = 104 - table1 - table2;

cout << table1 << endl;
cout << table2 << endl;
cout << table3 << endl;
```

Figure 8. A sample question and solution of the old Level 3 of the Random Numbers activity.

The decision was made to drop this level from the revised CA for two reasons:

1. The level was deemed too difficult at the time of evaluating struggle
2. The trigger for this CA was treated as a bugfix. Therefore, to ensure students who completed all levels of the activity prior to the bugfix received proper credit (i.e. 3 out of 3 possible points), the CA needed to maintain three levels.

After the revisions, the final objectives for the revised Random Numbers activity are as follows:

- **Level 1:** Use rand() and modulo arithmetic to generate random numbers given a constant.
- **Level 2:** Use rand() and modulo arithmetic to generate random numbers from 0 to a given bound.

- **Level 3:** Use rand() with modulo arithmetic, and addition/subtraction to generate a random number in an adjusted range.

The data for the revised Random Numbers activity is shown in Table 4, which shows decreased failure rates.

Table 4. Improved students' performance on the revised Random Numbers activity.

	Level 1 (New)	**Level 2 (Original Level 1)**	**Level 3 (Original Level 2)**
Students attempted	8128	7333	7165
Students failed	691	86	275
Students failed (%)	8.501	1.173	3.838

Across all levels, the problem and solution were restructured to clearly define expectations, making it easier for students to understand both the task and the desired outcome. The comparison of student failure per level before and after the revision is shown Figure 9.

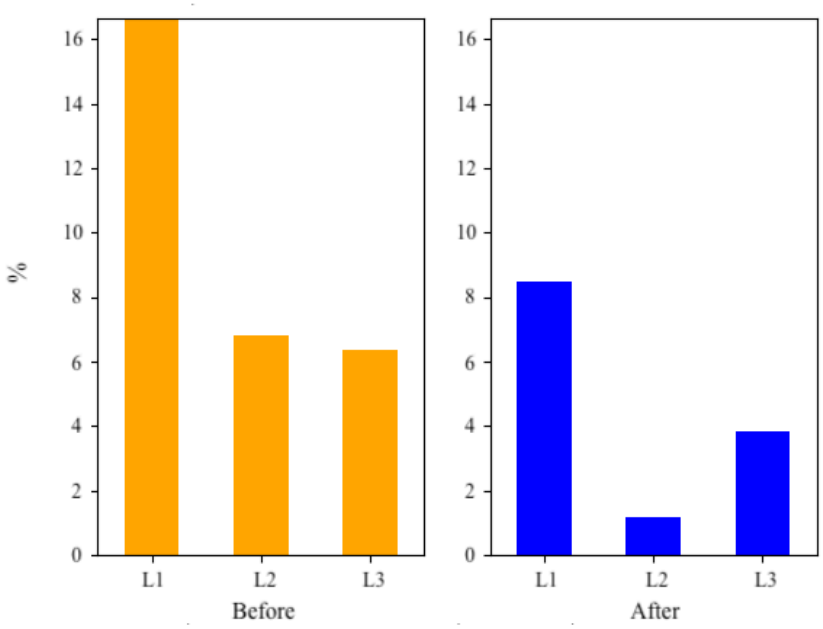

Figure 9. Student failure rate per level of the Random Numbers activity. Left: before revision. Right: after revision.

Case study #2: While loops

The While Loops CA was identified as a challenging exercise for students. The activity, designed for a Java programming course, includes three levels of increasing complexity.

- **Level 1:** Complete a partially written while loop with missing syntax.

- **Level 2:** Write a while loop to calculate basic arithmetic.
- **Level 3:** Write a while loop combined with another programming structure, such as an if-statement, to emulate a real-world coding problem.

Students must complete each level sequentially, meaning failure at Level 1 prevents progression to subsequent levels.

Over two semesters of Winter 2023 and Spring 2023, the statistics of students' performance on this activity is summarized on Table 5.

Table 5. Students' performance on the original While Loops activity.

	Level 1	**Level 2**	**Level 3**
Students attempted	875	713	564
Students failed	128	124	72
Students failed (%)	14.629	17.391	12.744

Level 2 exhibited the highest failure rate, followed by Level 1. This suggests that students struggle significantly when transitioning from completing pre-written code to writing independent loops.

Level 1 failures were commonly observed to be from syntax errors and an incomplete understanding of loop mechanics, while Level 2 failures are primarily related to difficulties in constructing functional loops from scratch. Further analysis into both levels is demonstrated. Figure 8 shows a sample question followed by a sample solution for a question in Level 1.

A while loop reads characters from input. Write an expression that executes the while loop until the character 'e' is read.

▼ **Click here for example**

Ex: If the input is `w d e`, then the output is:

```
User entered w
User entered d
Loop terminated
```

```
1  import java.util.Scanner;
2
3  public class CharacterReader {
4     public static void main(String[] args) {
5        Scanner scnr = new Scanner(System.in);
6        char value;
7
8        value = scnr.next().charAt(0);
9
10       while (/* Your code goes here */) {
11          System.out.println("User entered " + value);
12          value = scnr.next().charAt(0);
13       }
14
15       System.out.println("Loop terminated");
16    }
17 }
```

Solution:

value != 'e'

Figure 8. A sample question and solution for Level 1 of the While Loops activity.

A typical solution for a question in Level 1 involves a simple boolean operation to check if a variable is equal or not equal to a value. Further examinations showed that many students who failed the level were having issues with the syntax of the placeholder comment. To address this issue, Level 1 was edited to include a note, as shown in Figure 9.

A while loop reads characters from input into variable inVal. Write an expression that executes the while loop while character 'q' is not read into inVal.

▼ **Click here for example**

Ex: If the input is `W v Y z q`, then the output is:

```
Input is W
Input is v
Input is Y
Input is z
Exit
```

Note: First, delete the comment `/* Your code goes here */`. Then, insert your expression for the while loop condition.

```
1  import java.util.Scanner;
2
3  public class ValueReader {
4     public static void main(String[] args) {
5        Scanner scnr = new Scanner(System.in);
6        char inVal;
7
8        inVal = scnr.next().charAt(0);
9        while (/* Your code goes here */) {
10          System.out.println("Input is " + inVal);
11          inVal = scnr.next().charAt(0);
12       }
13
14       System.out.println("Exit");
15    }
16 }
```

Solution:

inVal != 'q'

Figure 9. A sample question and solution for Level 1 of the revised While Loops activity.

Compared to Level 1, Level 2 jumps from requiring students to complete partially written code, to understanding how to utilize the functionality of a while loop to calculate basic arithmetic. The activity is illustrated in Figure 10, showing a sample question and a corresponding sample solution.

Integer userInput is read from input. Write a while loop that iterates until userInput is greater than or equal to 24. In each iteration:

- Update userInput with the sum of userInput and 6.
- Output the updated userInput, followed by a newline.

▼ Click here for example

Ex: If the input is `16`, then the output is:

```
22
28
```

```
import java.util.Scanner;

public class SumCalculator {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int userInput;

      userInput = scnr.nextInt();

      /* Your code goes here */

   }
}
```

Solution:

```
while (userInput < 24) {
  userInput = userInput + 6;
  System.out.println(userInput);
}
```

Figure 10. A sample question and solution for Level 2 of the While Loops activity.

A typical solution requires a while loop structure to be used to calculate some basic arithmetic. Further examination into common student failures revealed that students did not understand the logic behind the arithmetic. A solution to this problem was to include a note on how the arithmetic could be calculated, as shown in the revised version in Figure 11.

Integer userInput is read from input. Write a while loop that reads integers from input while a negative integer is read. In each iteration:

- Update result with the sum of result and userInput.
- Then, read the next integer from input into variable userInput.

The positive integer should *not* be included in the sum.

▼ **Click here for example**

Ex: If the input is `-20 -23 -47 -36 28`, then the output is:

```
-126
```

```
1  import java.util.Scanner;
2
3  public class SumCalculator {
4     public static void main(String[] args) {
5        Scanner scnr = new Scanner(System.in);
6        int userInput;
7        int result;
8
9        result = 0;
10       userInput = scnr.nextInt();
11
12       /* Your code goes here */
13
14       System.out.println(result);
15    }
16 }
```

Solution:

```
while (userInput < 0) {
  result = result + userInput;
  userInput = scnr.nextInt();
}
```

Figure 11. A sample question and solution of Level 2 of the revised While Loops activity.

Analysis into student answers showed that arithmetic was the main factor contributing to student failures. By providing more guidance on how to implement the arithmetic, students were able to focus on the logic of the while loop over the logic of the arithmetic asked of them. The data for the revised While Loops activity is shown in Table 6.

Table 6. Students' performance on the revised While Loops activity

	Level 1	**Level 2**	**Level 3**
Students attempted	25239	23573	22107
Students failed	1248	1128	932
Students failed (%)	4.945	4.785	4.216

The significant reduction in failure rates across all levels can be attributed to addressing fundamental areas of student confusion in Levels 1 and 2. The difference in failure rates is illustrated in Figure 12.

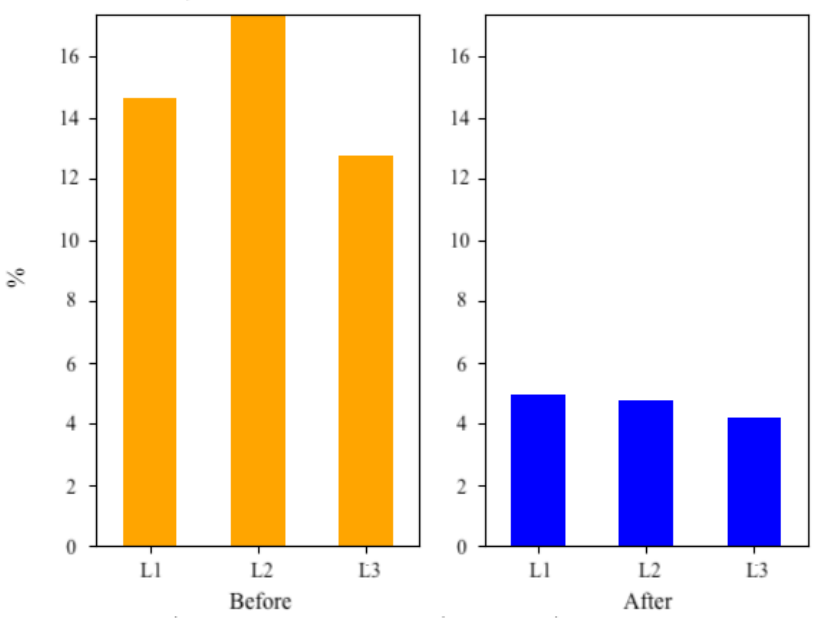

Figure 12. Student failure rate per level of the While Loops activity. Left: before revision. Right: after revision.

Case study #3: File Input

Topics on files are considered challenging to students because reading a file involves various status checks, error handling, and file operations to be executed in a certain order. The programming CA on reading a file in C++ covers the following objectives in increasing order of complexity:

- **Level 1:** Use file.open() and file.is_open() to open a file, check for status and read one data entry from the file.
- **Level 2:** Use file.fail() to detect any error encountered in file input, and handle read errors.
- **Level 3**: Use a while loop with file.fail() and file.eof() to read all the data entries from the file and output all the entries read.
- **Level 4:** Use a while loop to read all the entries from a file in which each entry has two data types.

Over the three semesters of Spring 2022, Fall 2022, and Spring 2023, the statistics of students' performance on this activity is summarized in Table 7.

Table 7. Students' performance on the File Input activity.

	Level 1	Level 2	Level 3	Level 4
Students attempted	7881	5657	5272	4680

Students failed	1763	236	475	56
Students failed (%)	22.37	4.17	9.01	1.20

Data shows that Level 1 has the highest failure rates, followed by Level 3. Furthermore, the percentage of failure at Level 1 is significant. In each CA, a student is required to pass a level before proceeding to the next level. Failing Level 1 means that the student is unable to proceed to higher levels. The cause of student failures is examined. Figure 13 shows a sample question followed by a sample solution for that question in Level 1.

A typical solution for a question in Level 1 involves a sequence of commands that include a file opening operation, a file status check, and two alternative branches of follow-up actions that depend on the file status. It is observed that the tasks expected in Level 1 may be too many and too complex for students. Therefore, the activity was revised to enhance scaffolding, simplify tasks, provide syntax hints, and reduce the cognitive load of the question.

String dataFileName is read from input. Open dataFileName as an input file containing the number of desks sold by a furniture store. If the file is successfully opened, read the first integer from the file into deskQuantity and output deskQuantity minus 8. Otherwise, output dataFileName followed by ": failed to open file". End each output with a newline.

Ex: If the input is `data3.txt` and:

Contents of file data3.txt
22 87 85

then the output is:

```
14
```

Ex: If the input is `data10.txt`, then the output is:

```
data10.txt: failed to open file
```

main.cpp | data1.txt | data2.txt | data3.txt

```
#include <fstream>
using namespace std;

int main() {
   ifstream deskFS;
   string dataFileName;
   int deskQuantity;

   cin >> dataFileName;

   /* Your code goes here */

   deskFS.close();

   return 0;
}
```

Solution:

```
deskFS.open(dataFileName);

if (!deskFS.is_open()) {
   cout << dataFileName << ": failed to open file" << endl;
   return 1;
}

deskFS >> deskQuantity;

cout << deskQuantity - 8 << endl;
```

Figure 13: A sample question for Level 1 of the File Input activity.

In the updated version, Level 1 focuses solely on the file-opening operation, while Level 2 addresses the required status check along with one branch of follow-up actions. The activity's learning objectives are revised to:

- **Level 1**: Use file.open() and file.is_open() to open a file.
- **Level 2:** Use file.is_open() check for file status. Terminate the program if the file is not open.
- **Levels 3 and 4:** No change.

Samples of the revision are shown Figure 14 and Figure 15.

String fileName is a file's name read from input. Open fileName as an input file associated with ifstream dataFS.

▶ **Click here for example**

main.cpp | data1.txt | data2.txt | data3.txt

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  int main() {
6     ifstream dataFS;
7     string fileName;
8     int futonCount;
9
10    cin >> fileName;
11
12    /* Your code goes here */
13
14    if (!dataFS.is_open()) {
15       cout << fileName << ": file could not be opened" << endl;
```

Solution:

```
dataFS.open(fileName);
```

Figure 14. A sample question and solution for Level 1 of the revised File Input activity.

String dataFileName is a file's name read from input, and is opened as an input file. If `bedFS.is_open()` returns false, then:

- Output dataFileName followed by ": failed to open file"
- Terminate the program with the return value 1, so that the program does *not* proceed to read the file.

▶ **Click here for examples**

main.cpp | data1.txt | data2.txt | data3.txt

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
   ifstream bedFS;
   string dataFileName;
   int bedCount;

   cin >> dataFileName;

   bedFS.open(dataFileName);

   /* Your code goes here */

```

Solution:

```
if (!bedFS.is_open()) {
  cout << dataFileName << ": failed to open file" << endl;
  return 1;
}
```

Figure 15. A sample question and solution for Level 2 of the revised File Input activity.

After the revision, students' performance is monitored over the next two semesters, from Fall 2023 to Spring 2024. Before revision, the completion rate and total failure rates are at 58.67% and 22.37%, respectively. After the revision, the complete rate increases to 85.65% and the total failure rate is reduced to 4.73%. Table 8 summarizes student performance on the revised File Input activity. The difference in failure rates before and after the revision is illustrated in Figure 16.

Table 8. Students' performance on the revised File Input activity.

	Level 1	**Level 2**	**Level 3**	**Level 4**
Students attempted	6069	5682	5515	5269
Students failed	287	68	190	71
Students failed (%)	4.73	1.20	3.45	1.35

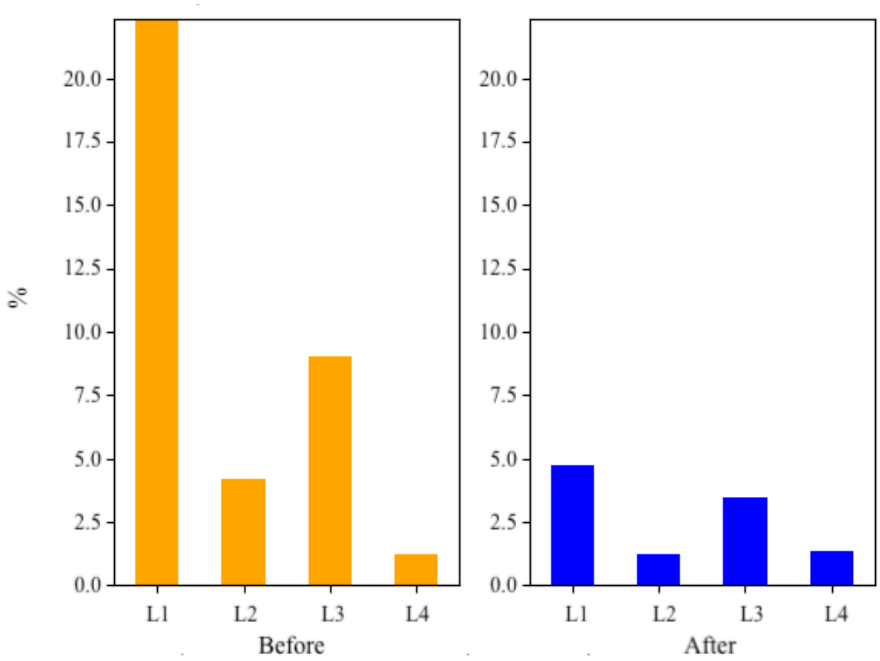

Figure 16. Student failure rate per level of the File Input activity. Left: before revision. Right: after revision.

Discussion

The results of our study underscore the effectiveness of a data-driven approach in designing and refining auto-graded programming activities, aligning with existing research on scaffolding and feedback in learning environments. Prior studies emphasize scaffolding's role in reducing cognitive load and improving comprehension in programming education [5] - [8]. Our findings confirm that structuring activities into progressively challenging components enhances student performance.

By analyzing metrics such as completion rates and common student errors, we identified key areas where learners struggled and addressed them by scaffolding the activities into smaller components. This approach, shown to enhance knowledge retention and self-efficacy [9], [10], proved especially effective for challenging topics with high struggle rates as well as for introductory topics where students needed extra guidance. The observed reduction in average failure rates from 12.90% to 4.35% (an 8.55 percentage point decrease) demonstrates the value of our method in promoting mastery and reducing student frustration, aligning with studies that advocate iterative assessment designs for better learning outcomes [11].

Case Study #1: The revised version of the activity led to a significant reduction in student complaints and a notable improvement in overall success rates, indicating that adding a scaffolded level to prepare students for the remainder of the activity was beneficial and reduced struggle. These results are consistent with previous research that suggests effective scaffolding enhances student engagement and problem-solving skills [6], [7]. Additionally, the rewording of the prompts that prioritized clarity appears to be a beneficial change, supporting prior findings that clear and structured instructions help reduce cognitive overload and improve comprehension

[12]. Simplifying the desired solution enables students to concentrate on demonstrating their understanding of the concept without being hindered by excess numerical or printing errors.

The failure rates in Level 1 and Level 2 both decreased significantly, suggesting that the revised wording and scaffolding effectively reduced student confusion. Furthermore, the decrease in the failure rate for Level 3 suggests that the improvements in earlier levels as well as restructuring of the level positively influenced students' understanding and performance at the advanced stage.

Case Study #2: Clarifying the questions presented and reducing ambiguity in the problem, helped students better engage and complete Levels 1 and 2. This improvement facilitated greater success in Level 3, as students were more prepared after progressing through the earlier levels. Research indicates that reducing ambiguity in problem statements minimizes student frustration and leads to better problem-solving performance [14].

Case Study #3: The improvements not only helped students master the basic tasks of file input, but further prepared the students to handle the more advanced tasks of Levels 3 and 4. These findings align with prior work emphasizing the role of structured and scaffolded practice in developing programming competency [5].

The data from all revised activities exhibit trends consistent with those observed in the case studies. This suggests that the implemented improvement strategies are effectively mitigating student struggles and providing an appropriate level of guidance to support students in strengthening their understanding in the concepts tested. Our results support existing literature on assessment design in computer science education, highlighting the significance of scaffolded and feedback-driven learning environments in fostering student success.

Future work

While our study has provided several valuable insights into the suggested design for auto-graded programing activities, there are several avenues for future research and work. One important direction is to continue refining and improving existing activities that demonstrate high struggle and failure rates. By analyzing student feedback and performance data, we can pinpoint more specific areas where learners face difficulties, and identify what topics scaffolding appears to have a higher impact on performance, if any. However, our data is limited to what our platform collects and is fully anonymized. In the future, we may have the opportunity to gain additional insights from surveys on summative assessment results and students' comfort level when transitioning from CS1 to CS2 and other advanced CS courses, though this is not guaranteed.

Additionally, we can extend our best practices beyond programming books and investigate if scaffolding activities for other disciplines have a similar effect. It would be beneficial to apply the same data-driven approach to ensure the effectiveness of CAs, thereby benefiting a broader range and larger number of students.

Conclusion

This study demonstrates the effectiveness of a data-driven approach to designing auto-graded activities in online, interactive STEM textbooks. By breaking down complex programming concepts into smaller, manageable units and incorporating scaffolding techniques, we have validated that student struggle rates can be significantly reduced achieving an average decrease of 8.55% and demonstrating proficiency in programming activities. Our analysis of various metrics such as completion rates and common errors, has provided valuable insights into the areas where students face the most difficulties and how to address the difficulties effectively.

References

[1] A. Steen-Utheim and A. Wittek, "Dialogic feedback and potentialities for student learning," *Learning, Culture and Social Interaction*, vol. 15, pp. 18-30, December 2017.

[2] A. Smith, S. McCarthey, and A. Magnifico, "Recursive Feedback: Evaluative Dimensions of e-Learning," in *E-Learning Ecologies: Principles for New Learning and Assessment*, Taylor & Francis. 2017, pp. 118-142.

[3] M. D. Gurer, I. Cetin, and E. Top, "Factors affecting students' attitudes toward computer programming," in *Informatics in Education*, vol. 18, no. 2, pp. 281–296, 2019

[4] L. B. Nilson, "Where STEM education falls short" in *Teaching at Its Best: A Research-Based Resource for College Instructors*, 4th Ed. San Francisco, CA, USA: Wiley, 2016, pp 273.

[5] R. E. Mayer and R. Clark,"Applying the segmenting and pretraining principles" in *e-Learning and the Science of Instruction: Proven Guidelines for Consumers and Designers of Multimedia Learning*, 4th Ed. Hoboken, NJ, USA: Wiley, 2016, pp. 203–208.

[6] B. R. Belland, A. E. Walker, N. J. Kim, and M. Lefler, "Synthesizing results from empirical research on computer-based scaffolding in STEM education: A meta-analysis." *Review of Educational Research,* vol. 87, no. 2, pp. 309–344, 2017.

[7] B. R. Belland. *Instructional scaffolding in STEM education: Strategies and efficacy evidence*. Springer Open, 2017.

[8] M. C. Lovett, M. W. Bridges, M. DiPietro, S. A. Ambrose, and M. K. Norman, "How do students become self-directed learners?" in *How learning works: Eight research-based principles for smart teaching*, 2nd Ed. San Francisco, CA, USA: Jossey-Bass, 2023, pp. 210-211.

[9] I. Horvath, "Reducing cognitive load through scaffolding," *eLearning Industry*, December 18, 2023. [Online]. Available: https://elearningindustry.com/reducing-cognitive-load-through-scaffolding [Accessed March 17, 2025].

[10] Grand Canyon University, "What Is Scaffolding in Education and How Is It Applied?," *GCU Blog: Teaching and School Administration*, September 19, 2023. [Online]. Available: https://www.gcu.edu/blog/teaching-school-administration/what-scaffolding-in-education-how-applied. [Accessed March 17, 2025].

[11] M. C. Lovett, M. W. Bridges, M. DiPietro, S. A. Ambrose, and M. K. Norman, "What Kinds of Practice and Feedback Enhance Learning?" in *How learning works: Eight research-based principles for smart teaching*, 2nd Ed. San Francisco, CA, USA: Jossey-Bass, 2023, pp. 130-161.

[12] R. E. Mayer and R. Clark “Does practice make perfect?” in *e-Learning and the Science of Instruction: Proven Guidelines for Consumers and Designers of Multimedia Learning*, 4th Ed. Hoboken, NJ, USA: Wiley, 2016, pp. 275-278.

[13] R. E. Mayer and R. Clark “Engagement in e-learning” in *e-Learning and the Science of Instruction: Proven Guidelines for Consumers and Designers of Multimedia Learning*, 4th Ed. Hoboken, NJ, USA: Wiley, 2016, pp. 221-223.

[14] E. Kazakou, A. D. Edgcomb, Y. Rajasekhar, R. Lysecky, and F. Vahid, "Randomized, structured, auto-graded homework: Design philosophy and engineering examples," in *ASEE Annual Conference and Exposition, Conference Proceedings*, *2021 ASEE Virtual Annual Conference*, *July 26-29, 2021*. [Online].