Stream Your Exam to the Course Staff: Asynchronous Assessment via Student-Recorded Code Trace Videos

Rachel S. Lim
University of California San Diego
ral077@ucsd.edu

Joe Gibbs Politz
University of California San Diego
jpolitz@eng.ucsd.edu

Mia Minnes*
University of California San Diego
minnes@eng.ucsd.edu

ABSTRACT

Now that students in our introductory object-oriented programming course are more familiar with Zoom and screen sharing, we consider novel assessments that leverage these tools. We developed a style of assessment where students submit a recorded screencast of them tracing a submitted program. We describe the design of the assessment and tracing prompts, and we report on an analysis of 59 submitted student videos. Our findings include common mistakes and aspects of student understanding that were expressed in the video but not in the text and code submitted by students. For example, we observed different strategies that yielded the same correct trace of a loop, as well as incorrect traces of students' own correct recursive programs. These guide us towards ways to refine the assessment and prompts in future iterations.

KEYWORDS

exams, code tracing

ACM Reference Format:
Rachel S. Lim, Joe Gibbs Politz, and Mia Minnes. 2023. Stream Your Exam to the Course Staff: Asynchronous Assessment via Student-Recorded Code Trace Videos. In *Proceedings of the 54th ACM Technical Symposium on Computing Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569803

1 INTRODUCTION

When teaching introductory object-oriented programming, it is natural to assess how well students are able to generate code. However, some *conceptual* learning goals are hard to infer just from students' code. In this experience report, we present and analyze an assessment strategy we deployed to help uncover conceptual (mis)understandings of key topics like loops and recursion.

Recent emergency remote instruction resulted in all of our students being familiar with Zoom and screen sharing. Given institutional support for these tools, we used them to explore novel assessment methods that produce richer and more authentic artifacts: student traces and explanations of their own work.

Exam Task (Programming: Loops and Arrays)

Task: Implement the method `averageWithoutLowest` which takes an array of doubles and returns the average (mean) of them, leaving out the lowest number. So, for example, the average of 1, 2, and 3 according to this scheme is 2.5. If there are no elements to average, the method should return 0. If there is a tie for the lowest, leave out only a single one of the tied lowest numbers.

Exam Task (Video: Loops and Arrays)

Choose an example of calling `averageWithoutLowest` with an array of at length 4 where the lowest element is not the first or last element in the array. Write down a trace table corresponding to one loop you wrote in the body of the method. For reference, an example of a trace table for a loop is included below.

```
int z = 0;
for(int i = 1; i < 6; i += 2) { z += i; }
// i start  i end  z start  z end
// 1        3      0        1
// 3        5      1        4
// 5        7      4        9
```

Figure 1: The midterm tasks related to loops.

In this paper, we begin by sharing the context for our experience report: a large-enrollment first-year level class at a public research-intensive institution in the United States. We detail the instructional strategies and assignments that prepared students for the assessments and share our exam prompts. In a qualitative analysis of a sampling of student submissions, we find support for this asynchronous video exam assessment style: specific student errors that are visible only in the videos and not through (even careful) reading of submitted code. We discuss course design choices that can effectively support this assessment style, including competency-based grading (to offer multiple opportunities to work through the material and to mitigate the stress of oral exams), careful design of exam code snippets and question prompts, and grading rubrics that scale to large course teams with undergraduate graders.

2 CONTEXT

The context of this experience report is a large (enrollment around 550) introductory course on object-oriented programming at a public research-focused US institution. In addition to regular formative assessments and programming assignments, there were 3 exams during the term, followed by a final exam. Each exam had 5-6 programming problems that students completed individually in a

*The first author was a TA and the second was one of the instructors of record for the course described in this report; they contributed to assessment design and analysis. The third author managed data collection and anonymization, along with contributing to analysis.

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569803

Exam Task (Programming: Recursion)

Task: Add the method `lengthOfLongest` to the interface `Tweet`. It should be a method with no arguments that returns the length (number of characters) of the `Tweet` in the thread with the longest content among `Tweets` in that thread.

Starter Code:

```
interface Tweet{}
class TextTweet implements Tweet {
  String contents;
  /* other fields and constructor ... */
}
class ReplyTweet implements Tweet {
  String contents; Tweet replyTo;
  /* other fields and constructor ... */
}
```

Exam Task (Video: Recursion)

Choose a test for this method where the thread has at least 2 `ReplyTweets` and 1 `TextTweet`. Next to the example, describe the stack in a comment.
Here is an example of a recursive program and the format for a stack trace:

```
interface I { boolean m(); }
class C implements I {
  public boolean m() { return false; }
}
class D implements I {
  I i; //constructor elided
  public boolean m() { return !this.i.m(); }
}
class Examples {
  C c = new C();
  D d_a = new D(this.c);
  D d_b = new D(this.d_a);
  boolean ans = this.d_b.m();
  // class    method reference  return value
  //  C     m      :2         false
  //  D     m      :3         true
  //  D     m      :4         false
}
```

Output of code above (via the Tester Library [8]):

```
new Examples:1(
 this.c = new C:2()
 this.d_a = new D:3(this.i = C:2)
 this.d_b = new D:4(this.i = D:3)
 this.ans = false)
```

Figure 2: The midterm tasks related to recursion.

48-hour period. The programs were graded automatically for correctness against an instructor-written test suite. On each exam, 2 of the programming problems had associated video tasks, which were graded manually by the instructional team (the instructor along with graduate and undergraduate teaching assistants) using a shared rubric. The video component was designed to provide assurance that a student understood the code they had submitted, which served as both an assessment of learning outcomes related to tracing code and a check on academic integrity. This format of exams had been in previous offerings of the course, and the course team had experience designing and administering them.

Exams were graded on an effectively pass/fail basis with a high level of achievement required to pass: there were no fractional points, and failing a more than a few automated test cases on any programming problem would result in a no-pass (details in Figure 3). Then, to support competency-based grading (and to reduce the stakes associated with any one exam), the final exam was structured as 3 explicitly-separated sections, each corresponding to one of the earlier exams. Students could replace their grade on an earlier exam with a strong performance on the corresponding part of the final; similarly, if they earned a high score on an exam during the term, they could skip that part of the final. Since these assessments give multiple opportunities to demonstrate knowledge of each concept, this fits within the competency-based grading paradigm.

In this paper, we use the second exam (which we call the midterm) as a case study of our coding + screencast videos assessment strategy, its benefits, and how we plan to refine it in the future. Figures 1 and 2 show 2 programming problems with their associated video tasks from this exam. The exam covered arrays, structural recursion through interfaces, and command-line programs.

2.1 Course and Problem Context

There are a few relevant details about course design and context that motivate the problems shown in Figures 1 and 2.

For `averageWithoutLowest`, students had seen several similar array-focused problems in lecture and in homework. The problem has special cases for array lengths 0 and 1 and interesting behavior on duplicates, which makes it rich enough to have a variety of meaningful test cases. A sample solution might be:

```
double averageWithoutLowest(double[] nums) {
  if(nums.length <= 1) { return 0; }
  double total = 0.0, lowest = nums[0];
  for(double n: nums) {
    if(n < lowest) { lowest = n; }
    total = total + n;
  }
  return (total - lowest) / (nums.length - 1);
}
```

The Tweets data structure used in the tasks in Figure 2 is a recursive structure (Tweets and replies to them) implemented through the shared `Tweet` interface. We spent time in lecture and homework discussing Tweets and Twitter,[1] so students were familiar with the idea of content, replies, and so on; understanding Twitter was not a new task for them as part of the exam. Similarly, students

[1]We acknowledge to the Twitter-using reader that we made an arguably nonstandard choice to represent threads starting with the *last* reply.

Code Score	Video Score 1	2	3
1-4	UV	UC	UC
5	UV	UC	ICP
6	UV	IVP	ICP
7	IVP	IVP	NM

Midterm Result	Count	Passed Final	Sampled	Sampled, Passed Final
(NM) No mistakes, passed	222	✓	9	✓
(ICP) Imperfect code, passed	144	✓	6	✓
(IVP) Imperfect video, passed	56	✓	4	✓
(UC) Unsatisfactory code	33	26	10	8
(UV) Unsatisfactory video	57	43	10	9
No Video	16	9	0	-
Total	**528**		**39**	

Figure 3: The midterm score was the sum of the code component (7 points max) and the video component (3 points max). The scores on the two components gave categories of achievement levels: NM, ICP, IVP, UC, UV, No video. The boxed categories all counted as effectively full credit for the midterm (and an automatic pass on the final portion corresponding to this midterm). Counts of students at each achievement level are tabulated on the right. A subset of student submissions from each achievement level was sampled for analysis. Samples from students that passed the midterm just included the midterm submissions; samples from students that did not pass the midterm included both midterm and final exam submissions.

had significant experience with the pedagogic testing library for Java [8] that gives the reference numbers for Examples; the `C:2` in `this.i = C:2` is explicitly referring to the `C` object printed on the line before it. Students had seen and worked with many examples like this in class and homework prior to the midterm. A sample solution to the programming task might be:

```
int lengthOfLongest() { // In ReplyTweet
  return Math.max(this.contents.length(),
    this.replyTo.lengthOfLongest());
}
int lengthOfLongest() { // In TextTweet
  return this.contents.length();
}
```

In both programming tasks, students were asked to write loop or stack traces in comments in plain text in their submitted code files. In lecture, the instructors drew such traces on virtual whiteboards with the same kinds of information, but not in a textual format. The submitted traces formed the basis for some of the student presentations in video and were part of the assessment.

3 RELATED WORK

The design of these exams is situated in existing research in several areas that motivate and justify our exam design decisions.

3.1 Assessment and Tracing Code

Perkins et al. notes the importance of students tracing their code and performing walkthroughs, as it helps with debugging and forces students to read their code more objectively instead of projecting their own presumptions onto their code [14]. Perkins et al. and others found that students may succeed with programming but struggle with program traces [10, 11, 14], suggesting that tracing their own programs is an opportunity for both instruction and assessment. Work on reverse tracing [5] suggests that having students come up with example inputs is also valuable. In our exam prompts, we did this by specifying constraints on the input that students should use for their traces without providing the actual input. There are many strategies and scaffolding for tracing [1, 9, 17]. We constrained ourselves to textual formats for simplicity and consistency for the exam, but investigating structured drawings and other tools may be incorporated in future versions.

3.2 Oral Exams in Computing Courses

Synchronous oral exams have a rich history in computer science and other disciplines. Their goals overlap with ours to some extent, and it is useful to consider the tradeoffs with our asynchronous recorded approach. We focus on a few studies in computing and adjacent fields that emphasize the typical benefits of oral exams.

Schurgers et al. [12, 16] handled oral exams at a similar scale to our video assessments. Their exams were graded by IAs, who could pass the student or indicate that the student had to take a followup exam with the instructors. We miss out on connecting students with instructional staff due to grading asynchronously. Ohmann [13] discusses oral exams specifically in introductory computer science, emphasizing followup questions and hints. We are not confident in training our course staff on this free-form, conversational exam, and instructors cannot hold over 500 oral exams themselves. Gharibyan [3] emphasizes the academic integrity (along with follow-up questioning); we get similar but weaker academic integrity benefits (authenticating the student's appearance).

3.3 Competency-based Learning

Our two-try exam style is an instance of competency-based grading (also called mastery learning, mastery grading, or specifications-based grading) which is an assessment technique used broadly in and out of computing [2, 7]. We have adapted this to serve as the major assessments for the course, replacing traditional on-paper exams. Our analysis in the paper does not rely heavily on this feature of our assessment, except that it gave us our stringent grading standards that allowed for an easy classification of pass/fail for sampling student submissions.

4 VIDEO ANALYSIS

We focus our analysis on the second exam and its corresponding part of the final exam. To select a sample of videos, we classified non-perfect exam submissions based on whether score deductions came from the code or the video. Figure 3 shows the breakdown of student performance. We over-sampled from midterm submissions that did not pass (UC and UV in Figure 3) with the goal of finding

misconceptions and comparing between passing submissions and non-passing ones. For the UC and UV midterm submissions, we compared their midterm submission and their submission from the corresponding part of the final exam. This resulted in a total of 59 videos from 39 students. Videos had a limit of 8 minutes and were submitted via Gradescope as a file alongside students' code.

To analyze the videos, two researchers independently watched a small subsample of the videos and developed an initial set of codes capturing salient features based on their observations. Then they watched all the videos together and iteratively developed a codebook organizing these features. Some of these features appeared frequently enough to suggest themes that give evidence for the usefulness of this assessment approach (demonstrated learning in paired videos, loop trace features, recursion stack trace features, bad but working code, bugs not caught by tests) and others pointed towards revisions we plan to implement in future offerings of this class (differences from prompt, inaccurate grading feedback). The remainder of our analysis uses specific examples from the sampled exam submissions to draw lessons about this assessment approach.

4.1 Disparate Loop Trace Strategies

Figure 4 illustrates how students filled in loop traces in their videos for the task in Figure 1. Students chose different *orderings*, all of which (for their program) could result in correct final results. The right-hand side of the figure represents the order in which the table given on the left would be filled in. We did not observe any students with traces that *did not* start at the top left.

We take trace order Ⓐ for detailed example. The first value the student fills in is the top-left – the `0` representing the initial value of `i` in the commented trace table. Then they next fill in the updated value of `i`, `10`, which is its value at the end of the first iteration. Next, they fill in the value `40` for the initial value of `small`, then the value of `small` at the end of the iteration, which is still `40`. Then they proceed with the same ordering in the next iteration: `i` followed by its updated value, then `small` followed by its updated value. We call this trace **by iteration** and **not in evaluation order**, because it does not follow the order in which Java would evaluate the variable updates. In particular, `small` gets its updated value of `10` in the second iteration while the value of `i` is still `1`, not when `i` is `2`.

In contrast, in trace order Ⓑ, the student fills in the values of variables **in evaluation order** of the program. They first consider the initial value of `i`, then the initial value of `small`, then the updated value of `small`, then the updated value of `i`.

Finally, a completely different strategy was Ⓒ, where students filled in the table one **variable at a time**. That is, they filled in *all iterations'* values of `i`, first initial, then updated, then *all iterations'* initial and updated values of `small`.

Each strategy Ⓐ, Ⓑ, Ⓒ was observed at least once in the sampled videos we watched. Interestingly, for this kind of program, we did not observe students making mistakes because of choosing one order over another, even though they needed to be careful to use the *starting* value of `i`, not the *ending* value of `i` when considering the array lookup `arr[i]`. However, other programs with loops (that we did not assign for tracing) are less amenable to strategies Ⓐ and Ⓒ. For example, consider strategy Ⓒ and the core of a merge algorithm on arrays:

```
// left, right, merged are arrays
int iL = 0, iR = 0, i = 0;
while (iL < left.length && iR < right.length) {
  if (left[iL] <= right[iR]) { merged[i] = left[iL]; iL++; }
  else { merged[iMerged] = right[iR]; iR++; }
  i++;
}
// left = { 1, 3, 5 } right = { 2, 4 }
// iL  iL'  iR  iR'  i   i'
// 0   1
// 1   ???
```

In this example, the updated values of `iL` depend on the values in the array and the value of `iR`, so the `iL` column cannot be filled without at least implictly tracing through some of the results for other columns. Here, a direct application of strategy Ⓒ does not make sense for tracing this program.

Sometimes variables' updated values depend on a mix of the original and updated values of other variables. Consider this implementation of Russian Peasant Multiplication:

```
// first and second are integers
int result = second % 2 == 1 ? first : 0;
while(second > 1) {
  first = first * 2; second = second / 2;
  if(second % 2 == 1) { result += first; }
}
// first  first' second second'  result result'
// 3      6      10     5        0      6
```

The tracer has to be careful to select the *updated* versions of `first` and `second` to update `result`. In contrast, in the loops assigned in the exam, the *initial* value of the variable was used, even if the updated value was already filled in.

In the exam, the question prompt ordered columns in the trace template in an order that suggested strategy Ⓐ. Some students still followed evaluation order, and some even reordered the columns in their own trace to work left-to-right in evaluation order.

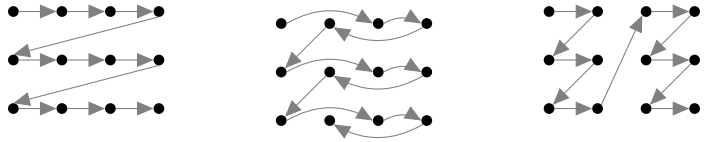
We take three lessons from this:

(1) In choosing whether to provide a sample table layout (and, if so, which) we need to balance assessing different skills (e.g. by leaving students to make choice of ordering columns, or choosing layouts that follow the order of evaluation) with the need for enough structure to grade an assessment.
(2) We can design problems and programs that would distinguish if students traced using strategy Ⓐ, Ⓑ, or Ⓒ as a more refined assessment of evaluation order understanding.
(3) We should consider explicit coaching or practice with setting up loop trace tables from scratch to make them as useful as possible for students' program comprehension. We do not want students to think that they have to overfit to particular schemas drawn from just one type of example.

4.2 Mistakes in Tracing Recursion

Students mostly generated correct loop traces; most deductions on the loop problem were due to code mistakes. In contrast, when tracing recursion, students with correct recursive code had mistakes in their trace. Setting aside students with wrong code, significantly

```
// double[] arr = {40, 10, 20, 30};
for(int i = 0; i < arr.length; i++) {
  if(arr[i] < small) { small = arr[i]; }
}
// i start  i end   small start   small end
// 0        1       40            40
// 1        2       40            10
// 2        3       10            10
// 3        4       10            10
```



A

By iteration, not in evaluation order

-

19/39 total students
8 of 19 passed exam

B

By iteration, in evaluation order

-

10/39 total students
7 of 10 passed exam

C

By variable

-

5/39 total students
3 of 5 passed exam

Figure 4: Different loop trace strategies seen in student videos. The remaining 5/39 students not counted above used a strategy other than one of these three, or switched between them in different parts of a trace. 1 of those 5 passed.

```
int lengthOfLongest() { // In ReplyTweet
  int len = this.replyTo.lengthOfLongest();
  if(len > this.contents.length()) { return len; }
  else { return this.contents.length(); }
}
int lengthOfLongest() { // In TextTweet
  return this.contents.length();
}

Tweet t1 = new TextTweet("Hi!");
Tweet t2 = new ReplyTweet("Hello there!", this.t1);
Tweet t3 = new ReplyTweet("Sup!!!!", this.t2);
Tweet t4 = new ReplyTweet("Sup!", this.t3);
assertEquals(this.t4.lengthOfLongest(), 12);

/* trace for this.t4.lengthOfLongest()
                               (wrong) (correct)
class      method          ref  return  return
TextTweet  lengthOfLongest :8   3       3
ReplyTweet lengthOfLongest :9   12      12
ReplyTweet lengthOfLongest :10  7       12
ReplyTweet lengthOfLongest :11  4       12      */
```

Figure 5: A pattern of incorrect stack traces made by 4 students. We show 4 Tweets here in order to illustrate several related scenarios.

incomplete traces, or one-off issues like missing base cases or wrong ordering, we observed:

(1) A pattern in **4** students' incorrect stack traces **when they had written correct code**
(2) A pattern in **the majority of submissions** that raises some questions about more complex traces

Intermediate Return Values. We saw a mistake in incorrectly-traced return values for intermediate (and final) stack frames, shown in Figure 5. Students wrote the result of `this.contents.length()` as the return value at each level. However, the correct return value is either that length **or** the result of the recursive call, depending on the outcome of the comparison. As a result, sometimes the final return value described in the trace was not what would actually be returned by the method (as in the example in Figure 5). We observed this even in some cases where students had a correct test case for the return value (shown as `assertEquals` in the example).

This has aspects of other incorrect models of recursion, especially the *Return Value Model* of Götschi et al. [4], described as:

> ... the misconception that at each instantiation a value is evaluated, before the next instantiation is complete. These values are stored and all combined into a solution.

That is, students associate relevant values with each recursive call but do not correctly describe when or how they are combined into a final answer. Indeed, given that students' traces would sometimes indicate a different answer than what their own code had produced in a test case, they may have been applying incorrect evaluation semantics by rote and not making connections to their program. This general mistake is also related to misconceptions that have been repeatedly observed about recursive calls whose results are used in subsequent computation [6, 15].

We take a few lessons from this. First and most importantly, this assessment is giving information about student understanding beyond what we can get from automated test cases. In these examples, the students' code typically produced the right answer, and the video provided useful triangulating information about their understanding of recursion.

Second, there are a few ways to improve the assessment:

(1) If a student chooses an example that happens to have content lengths in increasing order, they could make this mistake and still produce a correct stack trace. Any time the last `ReplyTweet` has the longest contents, a student could make this mistake and get the final return value correct. We can improve similar future prompts to incorporate ordering.
(2) We made less use of the temporality of video when assessing this recursion trace than with loop traces; we only need to observe the final resulting stack trace to see the mistake and guess at misconceptions. However, we could also ask students to fill in the calls in the order that they happen and then the return values in the order that they happen. This would force them to use what Lewis calls the *Simulating Execution* strategy of tracing recursion [9] and may reveal more misconceptions about evaluation and return order.

(3) We could have a requirement that students include test cases for all intermediate results. The prompt only required showing the test for calling `lengthOfLongest` on the last `ReplyTweet`. Requiring the demonstration of intermediate results nudges students towards observing the discrepancy between their trace and the behavior of their program.

(Not) Tracing Multiple Recursive Calls from the Same Frame. We observed that 22 of the 32 students with a correct solution for this problem submitted code like this:

```
int lengthOfLongest() {
  if(contents.length() > replyTo.lengthOfLongest()) {
    return this.contents.length();
  }
  return this.replyTo.lengthOfLongest();
}
```

This solution has correct functionality, but it takes exponential time to run in the worst case. Runtime analysis is not an explicit learning outcome in this course. However, students who wrote this code did not incorporate the exponential tree-shaped recursion pattern into their traces. Indeed, we did not even give them a good structure to write this. However, a correct trace for a 3-Tweet thread as suggested in the prompt would show (up to) **7** calls: 1 for the initial call, which calls `ReplyTweet.lengthOfLongest` twice, each of which call `TextTweet.lengthOfLongest` twice.

Of the 19 students whose submissions we reviewed with a passing score on their first exam attempt, 11 wrote the (worse) exponential code and 8 did not. In contrast, of the 20 students who had to retake the exam, 11 wrote exponential code, 2 wrote non-exponential code, and 7 wrote incorrect code, suggesting that writing the "better" version of this method (storing the result of the recursive call in a variable) is associated with overall competency.

Ultimately, we did not deduct points for students who wrote the linear-style trace for these branching-recursive implementations. We did not give any explicit structure for students to do so, and their trace was accurate for the last snapshot of the stack at its deepest point. In the future, we want to find ways to trace more kinds of recursion or to refine the prompt to specify just a snapshot.

4.3 Grading Considerations

Videos were graded by course staff with a rubric provided by the instructors, and deductions included mistakes not mentioned above, for example completely omitting a column in the stack or loop trace. In terms of effort, we budgeted about 10 minutes to grade each 8-minute video. Grading of roughly 550 videos was split between 10 course staff, so there was around 8 hours of grading work per staff member. We did not measure how long was actually spent.

We likely need more structure than this: through our analysis, we reviewed the grading of all 59 videos. We agreed with the grading of 46 of them. In 12 cases, we found a mistake in the video that the grader missed, and 1 case where the grader gave too little credit. We are concerned about missing 12 cases where a deduction should have been applied. 3 of these were for students who passed the midterm – these students should have had to re-test.

In the future, we plan to implement more cross-checks and review of grading videos, which is challenging for our staff. We believe part of the issue is that since so many of the videos *are* fully correct (Figure 3), it's easy to start expecting that result and miss key details. Pair-grading is promising—that's effectively what we did for this report when reviewing the submissions in pairs or a group of three to find these discrepancies—though doubling grading work is difficult to fit into our course allocations.

5 WILL WE TRY THIS AGAIN?

Our analysis demonstrates that the tracing and video components of this assessment reveal aspects of students' (mis)understandings that are not evident just from their code submissions. This alone gives us confidence to continue to use it, even as we may be able to complement it with other assessment mechanisms (e.g. synchronous in-person exams) that are more feasible now than they were when we first started using this strategy for remote instruction.

We have learned several ways we can improve the assessment:

- Design prompts for programs that will have different behavior based on students' choice of loop tracing strategy
- Design prompts for recursive programs that lead students to explore return values from intermediate stack frames
- Consider richer models of stack traces for richer recursive patterns and how students can author them
- Improve the grading process for instructional staff with more cross-checking and refined rubric design

We have also reflected on aspects of the course and assessment that do not show up directly as a measured outcome in our analysis:

Re-testing Recursion. We did not take into account the fact that more students struggled with recursion than with loops when designing the final exam. As a result, there was no tracing of recursion (just programming) on the final, which likely missed an opportunity for important re-testing. This was an assessment oversight; in future quarters we will be more deliberate about using the results of the in-quarter exams to choose the most important content to re-test at the end of the quarter.

Grading for Competency. We reconsidered the grades for students with satisfactory code but with video deductions who passed the midterm according to our rubric. We think that these students would have benefited from the practice and re-test of the final – scoring 1/3 on the video portion of the midterm should have **not** been sufficient to pass. The senior instructional staff debated this while grading, and given the newness of the policy, erred on the side of passing students. Also, some students (and staff!) had a misconception that points out of 10 ought to represent a percentage, and it was unfair that an "80%" would not pass. We will seek clearer ways to report our competency-based grading metrics in the future.

Ultimately, we have found an additional useful tool for our instructional toolkit. We expect to use it again and improve on it to learn about our students' understanding at scale.

Acknowledgements

We are grateful to Greg Miranda for co-teaching the course and for helping to design and administer the assessments. He and Gerald Soosai Raj gave valuable feedback on rubrics for videos. None of this work would be possible without the course staff who did the grading and helped to run the course.

REFERENCES

[1] John Clements and Shriram Krishnamurthi. Towards a notional machine for runtime stacks and scope: When stacks don't stack up. In *Conference on International Computing Education Research*, 2022.

[2] James Garner, Paul Denny, and Andrew Luxton-Reilly. Mastery learning in computer science education. In *Australasian Computing Education Conference*, 2019.

[3] Hasmik Gharibyan. Assessing students' knowledge: Oral exams vs. written tests. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2005.

[4] Tina Götschi, Ian Sanders, and Vashti Galpin. Mental models of recursion. In *SIGCSE Technical Symposium on Computer Science Education*, 2003.

[5] Mohammed Hassan and Craig Zilles. Exploring 'reverse-tracing' questions as a means of assessing the tracing skill on computer-based cs 1 exams. In *Conference on International Computing Education Research*, 2021.

[6] Hank Kahney. What do novice programmers know about recursion. In *SIGCHI Conference on Human Factors in Computing Systems*, 1983.

[7] Chen-Lin C. Kulik, James A. Kulik, and Robert L. Bangert-Drowns. Effectiveness of mastery learning programs: A meta-analysis. *Review of Educational Research*, 60, 1990.

[8] Benjamin S. Lerner, Sachin Venugopalan, Viera K Proulx, and Matthias Felleisen. The tester library, 2022. URL https://course.ccs.neu.edu/cs2510a/tester-doc.html.

[9] Colleen M. Lewis. Exploring variation in students' correct traces of linear recursion. In *Conference on International Computing Education Research*, 2014.

[10] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, 2004.

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

[12] Marko Lubarda, Nathan Delson, Curt Schurgers, Maziar Ghazinejad, Saharnaz Baghdadchi, Alex Phan, Mia Minnes, Josephine Relaford-Doyle, Leah Klement, Carolyn Sandoval, and Huihui Qi. Oral exams for large-enrollment engineering courses to promote academic integrity and student engagement during remote instruction. In *2021 IEEE Frontiers in Education Conference (FIE)*, 2021. doi: 10.1109/FIE49875.2021.9637124.

[13] Peter Ohmann. An assessment of oral exams in introductory cs. In *SIGCSE*, 2019.

[14] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

[15] Tamarisk Lurlyn Scholtz and Ian Sanders. Mental models of recursion: Investigating students' understanding of recursion. In *Conference on Innovation and Technology in Computer Science Education*, 2010.

[16] Curt Schurgers, Saharnaz Baghdadchi, Marko Lubarda, Maziar Ghazinejad, Alex Phan, and Huihui Qi. Evaluating oral exams in large undergraduate engineering courses. In *2021 IEEE International Symposium on Antennas and Propagation and USNC-URSI Radio Science Meeting (APS/URSI)*, 2021.

[17] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018.