Online Programming Exams - An Experience Report

Seán Russell
University College Dublin
Dublin, Ireland
sean.russell@ucd.ie

Simon Caton
University College Dublin
Dublin, Ireland
simon.caton@ucd.ie

Brett A. Becker
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

ABSTRACT

When seeking to maximise the authenticity of assessment in programming courses it makes sense to provide students with practical programming problems to solve in an environment that is close to real software development practice, i.e., online, open book, and using their typical development environment. This creates an assessment environment that should afford students sufficient opportunities to evidence what they have learned, but also creates practical challenges in terms of academic integrity, flexibility in the automated grading process, and assumptions surrounding how the student may attempt to solve the problems both in terms of correct and incorrect solutions. In this experience report, we outline two independently observed cohorts of students sitting the same Java programming exam, with different weights, over three years. This is undertaken as a reflective exercise in order to derive a series of recommendations and retrospectively obvious pitfalls to act as guidance for educators considering online programming exams for large (i.e. $n > 150$) introductory programming courses. After discussing our assessment methodology, we provide 4 high-level observations and centre a set of recommendations around these to aid practitioners in their assessment design.

CCS CONCEPTS

• **Social and professional topics → Student assessment**; **Computing education**.

KEYWORDS

authentic assessment, plagiarism, programming, video

ACM Reference Format:
Seán Russell, Simon Caton, and Brett A. Becker. 2023. Online Programming Exams - An Experience Report. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2023), July 8–12, 2023, Turku, Finland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587102.3588829

1 INTRODUCTION

The task of accurately assessing the code writing abilities of students in programming classes has always been a challenge. Many university programming courses continue the use of paper exams requiring hand-written solutions, while others use controlled environments for programming and other computer-based exams [14]. The transition to online education precipitated by the COVID-19 pandemic was sudden and required the adaptation of many long-used teaching and assessment strategies [22].

This paper details the experiences of the authors in adapting the examination practices of two programming classes over three years for online delivery. In total 540 students (average of 180 per year) were observed. Prior to the pandemic these classes made use of computer-based testing, in the form of proctored programming examinations, as one of their principle summative assessments. The use of unique exam questions, coupled with the proctored environment provided little opportunity for cheating. This paper details the approach taken to migrate this assessment to an unproctored online environment while limiting the opportunity for cheating.

The risk of academic integrity violations prompted a consideration of alternatives that could be used to replace a timed programming assessment. The options considered included multiple choice questions (MCQ), individualised code tracing exercises [19], parsons problems and faded parsons problems [6, 29], and combinations of these. However, none of the options considered provided a similar measurement of the code writing ability of the students or afforded a particularly authentic environment.

Consideration then turned to what mechanisms could be put in place to restore confidence in the accuracy of the same assessment conducted online and assuage concerns about susceptibility for academic integrity violations. The following sections discuss related work and concepts, the context in which the strategies were applied, the detail of our approach to online programming examination, the lessons we learned from the process and improvements that we consider making in the future. We believe that this experience will prove useful to the many other educators who have, or still are, struggling with the difficulties of employing online assessments that provide genuine assessment (mitigating academic integrity violations) with a suitable degree of authenticity.

2 BACKGROUND

This section discusses research and areas of concern that have influenced the design of the assessment in this study. The primary concerns were to maximise the authenticity of the assessment while mitigating academic integrity violations. Finally, considerations around the utility of differing approaches to automatically grading code are considered.

2.1 Authenticity

One of the original aims behind the design of the assessment was a desire to make it as authentic as possible within the bounds of an examination. Programming experience gained prior to and during the completion of these classes will most likely have come while programming using an integrated development environment (IDE). When professionally applying the knowledge that they have learned, the students will most likely be programming using an

ITiCSE 2023, July 8–12, 2023, Turku, Finland
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0138-2/23/07.
https://doi.org/10.1145/3587102.3588829

IDE. Gulikers et al. [11] described the five-dimensional framework (5DF), by which the degree of authenticity of an assessment can be objectively viewed. The dimensions are the task itself, the physical and social context under which it takes place, the assessment result, and the assessment criteria.

The programming examinations in these classes are designed with an aim to be authentic with respect to all dimensions of authenticity except the social context in which the students perform the assessment. Prior to the pandemic, these assessments were carried out within a proctored examination environment (in-person) where no internet access or communication were permitted. Students were allowed notes that were either printed or digital, so a suitably prepared student would have access to the Java API and example code to refer to. This is considered an important aspect of creating a more authentic environment, as students will not be expected to produce code only from memory in a professional capacity. As a result of this, the primary concern of exam invigilators was the prevention of communication between students and illicit use of devices or internet access.

It should be noted that in this paper we are applying a subjective and somewhat artificial view of authenticity [10] due to educational requirements. A truly authentic environment would have the students free to communicate and collaborate and would not have such a strict time limitation, but would be less useful at measuring individual student ability.

2.2 Plagiarism and Collusion

Software development often encourages code reuse and collaborative development practices, which makes the concept of academic integrity difficult to formalise in computing [26]. Previous research has found that as much as 90% of students will admit to committing academic integrity violations (including cheating) at least once [21]. The difference between assignments in the field of computing and others means that the boundaries of acceptable and unacceptable practices are much more difficult to define for computing assessments than for essays [24]. The nature of programming assessments makes them vulnerable to plagiarism as it can be often difficult to distinguish between two correct solutions, particularly when strict or descriptive code style rules are applied. Frequently all students will have a shared scaffold to start from and specification to meet [16].

The Fraud Diamond is a model used to explain when fraud is likely to occur and how to prevent it, it has previously been applied to the context of plagiarism [1]. It frames the likelihood of fraud (or plagiarism in our case) in terms of incentive (the student wants/needs to plagiarise), opportunity (the student can plagiarise), rationalisation (the student can convince themselves it is worth the risk), and capability (the student believes they can do it and not get caught). Conceptually, prevention of plagiarism can be achieved by eliminating any one of these dimensions, i.e. if there was no opportunity to plagiarise it can't happen, or if the student does not believe they will get away with it then they won't do it.

In the original in-person proctored exam setting, the principle concern was for the prevention of collusion. Exam questions were never repeated, so there was no concern that solutions existed on the internet for the students to plagiarise. Any discussion between students was prevented and any use of internet access resulted in the student receiving 0, thus the aim was to prevent any opportunity for students to collude and reduce their belief in the capability of doing so without get caught.

2.2.1 *Contract Cheating.*

Contract cheating is not a new development in higher education [5], though it is one that appears to be on the rise [8]. This form of plagiarism can often be difficult, if not impossible to detect. Sites like Chegg and Course Hero advertise the low response times for questions, which can be as little as 15 minutes. This makes it feasible for students to post questions and receive answers within the time window of the exam.

2.2.2 *AI.*

Within the last two years, a new threat to academic integrity emerged with the release of Generative AI tools that generate code from text prompts such as GitHub Copilot, and more recently ChatGPT [2]. Although only in their infancy, these tools have already proven capable of scoring in the upper quartile in a CS1 exam, where problem descriptions were given verbatim to the Codex API [9]. The demonstrated capability of these services requires the consideration of plagiarism as well as collusion when using assessments for the first time.

They also have an effect on the dynamics of the model of plagiarism more generally. Many universities apply the same punishments to students who share their work as they do to students who have copied [18]. Opportunity to collude with another student is contingent on the trust of that student that they will not get caught and jeopardise their own grade. These services pose no risk of rejection and as a result students will have an excellent opportunity to plagiarise as long as they have access to one of these services. It should be noted that the nascent tools designed to determine the likelihood that a given text was generated by AI are easy to fool, and are not useful for detecting if code was AI generated.

2.2.3 *Plagiarism Detectors.*

Traditional collusion and plagiarism has been the battleground of a quiet war between students and the developers of similarity (plagiarism) detectors. Plagiarism detectors, such as MOSS and JPlag [17, 20], analyse student assignments and identify where pairs or groups of submissions share similarities and highlight this for analysis by instructors. Different tools operate within different domains and offer different capabilities, for instance some will work on any text and others will operate only on a specific set of programming languages, some will compare submissions with content from the internet and other will only compare against other submission. These tools are useful in their principle role after plagiarism has occurred, but can also be discussed early in the class to reduce students belief that they can get away with it [25].

2.2.4 *Remote Proctoring.*

Remote proctoring became a fundamental component of the assessment strategy of many universities during online education [27]. Research has noted that remote proctoring had only minimal impact on student performance in examinations [12]. Other universities, in the face of concerns over privacy, security and accessibility, chose to oppose the use of remote proctoring [23].

2.2.5 *Video Explanations.*

The use of video in assessment would have been considered somewhat unconventional before the pandemic. Zarb and BirtlesKelman [30] trialled the use of video to replace a more traditional presentation based assessment. This was beneficial to students, who appreciated the opportunity to rehearse, and to staff, who were capable of giving better feedback.

VanDeGrift [28] augmented traditional written exams with short video explanations of their solutions. This was primarily designed as a measure to prevent academic integrity violations, though it was more effective as a reflective exercise for the students. Students were required to complete and submit a video explaining their solution to one of the questions in the exam by the end of the day. These exams took the form of Moodle quizzes where the questions could be multiple choice, short answer, true/false, short code fragments or free-response. Students had the opportunity to correct mistakes that they uncovered after completing the exam and consequently could improve their scores.

2.3 Autograding

One of the benefits of a computer-based programming exam is the possibility for automated analysis to be used to partially or fully complete grading. There are many tools which can be used for this functionality, but they are often limited to use within a particular system or virtual learning environment [3]. Automated assessment can dramatically reduce the assessment workload of teaching staff, but present different limitations depending on the technique employed [15].

The most common techniques for automatic grading are based on output comparison, unit test or code quality. Output comparison relies on execution of the program with specified input such that the output can be matched with expected output. Unit test based grading permits the inspection of components of a student's solution in isolation. Typically, comparisons are made between expected and returned values of functions or methods provided known parameters. Code quality is more difficult to assess automatically, though linting and source code analysis tools, such as PMD can help in this process.

These approaches can have drawbacks from the perspectives of both instructors and students. Output comparison is typically unforgiving, that is a single incorrectly placed character means a failed test, and as such considered by some to be inadequate for use in a learning context [15]. The introduction of testing code (like unit tests or similar) can complicate the compilation and/or execution process and produce error messages that are more difficult for students to solve [4]. Unit tests require a certain knowledge of the expected design of the components in the system in order to test them. This means rather than specifying the overall goal and leaving design to the students questions must specify the design more closely (potentially providing interfaces for students to base their work on). Static analysis tools are generally more suited to finding problematic code fragments than consistently measuring the quality of code that they analyse.

3 CONTEXT

University College Dublin is a European doctoral granting institution with approximately 40,000 students and the equivalent of an "R1" (very high research activity) in the US Carnegie Classification[1]. Approximately 1,000 undergraduates and 600 postgraduates are computer science students (major and minor). There are over 60 faculty members who teach both graduate and undergraduate courses resulting in courses which typically range in size from 60 to 200 students.

We focus on three years of two approximately isomorphic object-oriented programming (OOP) classes within different contexts. Class A are second year undergraduate students learning OOP in Java. These students have completed two semesters of procedural programming in C. Class B are completing the subject as part of a MSc in computer science. These students typically do not have prior OOP or Java experience from their undergraduate degrees as those who already have experience with these topics are prevented from enrolling.

The final summative programming exam was identical and synchronously completed by both classes. Table 1 shows the number of students of the last three years for both classes. The upward trend in these numbers is such that the use of in-person proctored exams would have become increasingly infeasible due to a lack of sufficiently large spaces to accommodate all students. As pandemic restrictions eased and returning to in-person teaching, the exam remained online.

Table 1: Student numbers for classes A & B 2020-2022

Class	2020	2021	2022	Total	Average
A	129	143	141	413	138
B	22	48	57	127	42
Total	151	191	198	540	180

The classes are equivalent in terms of the lecture material and practical work, but differ slightly in their assignments. Class A has a unit testing assignment (to improve familiarity with unit testing based evaluation) while class B (a post graduate course) includes project work. Both classes share the same weekly homework exercises, and practical exam. Since returning to in-person teaching, class A has live in-person lectures and labs, but class B has remained mostly online, as this was the existing pre-pandemic delivery modality. The programming exam constitutes a significant proportion of the final grade of both classes (70% for class A, 40% for class B).

Both classes make use of Coderunner [13], a plugin for the Moodle Virtual Learning Environment (VLE) that enables the automatic grading of computer programs in a number of languages. While Coderunner is principally designed to assess based on outcome testing, test cases can be designed to more closely simulate unit testing or though the use or the Java reflection API can measure some elements of code quality. As such students are sufficiently experienced with these testing methods that their use in an exam situation would not be a cause for concern.

During the pandemic, university leadership adopted the position that remote proctoring services would be institutionally disallowed.

[1]carnegieclassifications.acenet.edu/carnegie-classification/classification-methodology/basic-classification/

This decision was based on concerns over student privacy and security, as well as concerns of the rights afforded to students under the EU General Data Protection Regulation (GDPR)[2].

4 METHOD

Given the requirement of accurately assessing the ability of students to write code, it was decided to transition the programming exam from a Bring-Your-Own-Device (BYOD), proctored, in-person setting to an online exam without proctoring. To address concerns of a potential increase in academic integrity violations, an additional must-pass video component was added to the exam. This section describes the approach taken to question design as well as to the overall timing and grading of an online programming exam with video explanations.

4.1 Instructor vs. Machine

Following the release of AI coding assistants like GitHub Copilot, the questions in previous exams were assessed against these new systems. Questions that were poorly answered by these systems were analysed and a rudimentary determination was made as to the attributes that made them difficult for the AI to solve them. The key attribute was questions that would be difficult to solve using only the problem statement, but with example output and sufficient unit tests become much more solvable by students.

The issue faced in designing questions to beat AI code generation is that obfuscation in the wording or design of the problem statements would have a negative effect on the ability of students to understand them. While this has not been exhaustively investigated, it seems that these systems struggle with determining information from more abstract output examples. This approach to question design is not compatible with all of the learning objectives in course, and as such were only used in at most one question.

4.2 Exam Details and Timing

The exam contained five questions in total. The students were required to complete two: one from part A (Q1 and Q2; fundamentals questions, e.g. inheritance and code design) and one from part B (Q3, Q4, and Q5; more problem solving questions, e.g. reading files, manipulating arrays etc.). Both questions carry equal weight. The timing of the exam is as follows:

-3 weeks: Formal exam instructions are released
-15 min: Scaffolded code is released
-5 min: Exam questions released (so they could be printed)
0 min Exam commences
+180 min: Exam questions must be submitted to VLE
+180 min: Video questions released
+190 min: Grace period (code) ends, late penalty applies
+240 min: Video narrated demonstrations submitted
+260 min: Grace period (video) ends, no further submissions allowed

Students are given five minutes in which they can prepare their system for the exam by downloading and importing code and test files into their IDE. Students are not aware of the video questions that they will be asked to complete until +180 minutes. Students should have submitted their code before the questions are available, this is to prevent students from altering their submission at the last minute to selectively cut parts that they cannot explain (and presumably have gained through plagiarism or collusion).

[2]gdpr.eu/

At +180 minutes, the second question paper is released containing the video questions. For each of the questions (Q1 - Q5) there are four video questions. So if the student answered Q2 and Q5, then they must choose two questions from both Q2 and Q5 and address them in a video response, with code demonstration / highlighting to support their answer. We allow the student to choose 2 from 4 video questions (for example in Q2) because questions can relate to specific parts of the task, and this choice allows for situations where the student may not have attempted the entire question. It also allows for a larger distribution of answers (there are more questions). The following are examples of a typical question (with the objective to make the student reflect on their submission and coding thought processes) which includes some metacognitive reflection [7]:

- Discuss how you handled String parsing in this question. How did you break up the String and why did you do it this way? Were there alternatives, and why did you not use these?
- Pick the aspect of the question you found the most challenging, but believe you have answered correctly. Why specifically was it challenging and how did you go about developing your solution for this aspect of the question?
- Which of the unit test(s) (that your code passes) was/were the most helpful in developing your solution? Discuss how it/they helped you answer the question(s).

Students are expected to record a video that does not (significantly) exceed five minutes in duration which addresses four of the video questions. Students are explicitly not required to appear on camera, though they are free to if they wish, only to show the relevant code as they are explaining it. This is in consideration of privacy concerns and is in line with university policies regarding forced use of cameras by students.

4.2.1 *Late Submissions.*

Late submissions are allowed, but are penalised at the rate of 2.5 points per two minute block. To allow for disruption due to technical issues, a ten minute grace period is used. Immediately at the end of this period the full late penalty is enforced, so a student submitting at ten minutes and 30 seconds late would be penalised 15 points.

This policy is strictly enforced, but exceptions are made for students with official accommodations and students who face unexpected technical problems (in this event students must make contact with the instructor and provide documentation – claims after the time are not considered).

4.3 Grading Workflow

The grading commitment required for this approach to programming assessment is quite heavy and averages at about 15 minutes per student; it can be less for very high/low scoring submissions. The inspection and building of feedback consumes the bulk of this time. Automatic grading could reduce this time to the duration of the video explanations (or less if played at an increased speed).

The grading workflow is made up of the following steps; 1) a script processes student submissions, assembles them into prepared structure, compiles their submission and also the (question appropriate) unit tests against the student's submission, if needed), 2) manual inspection of code and fixing (small) errors, 3) unit tests are executed, 4) final round of small fixes (and tests run again); 5) manual and automated feedback are merged, and unit test score (computed by the test scripts) captured. To standardise the grading workflow, all information is entered into a GDPR compliant online form (which has additional data entry validation checks) and facilitates the capture of some descriptive statistics as well as additional logic and prompts (for code quality scoring) for grade computation; some VLEs offer the same functionality.

4.4 Grading

The grade for each question consists of three components, an automatically graded score based on unit tests, a score for code quality based on manual code inspection (workflow step two), and a score based on the explanation of the question in the video. In the questions in part A, this breakdown was 18 points determined by unit tests, 7 for code quality, and 10 points for the video explanations. In part B, unit tests determined 20 points of the score, while code quality contributed 5, and the video explanation contributed 10.

4.4.1 The Video Questions. The score that a student receives for the video component is based on a binary determination that the student understands and explains sufficiently the code that they have submitted. In the event that a student is determined to have provided a sufficiently clear response to the video question, then they are awarded a score based on the number of points they have earned in the other components of the question. For example, if a student scores 20 out of 25 for the other components of the question, then their score in the video will be 8 out of 10 (for a total of 28/35). Alternatively, if the video explanation is not satisfactory, the student will receive the score 0 out of 35 for that question.

This choice was made after consideration of two undesirable scenarios. In the first scenario, a student submits code that is severely flawed and scores five points from a possible 25, however the video description explains the code clearly. An award of 10 points would be significant and not in line with the quality of code actually submitted. In the second scenario, a student submits perfect code and gets 25 points out of the available 25, however the student is unable to explain the code or respond to the questions. An award of just 0 for the video and the original 25 points for the other part of the question would result in a final score of 70%.

Presumably, this student colluded or plagiarised in some way. However, lacking proof, the result would stand. The scoring of these questions is designed to remove the question of collusion or plagiarism and focus on the ability to show understanding. A score of zero in the video component is not considered an academic integrity violation, though it can provide excellent evidence in the event of referral to disciplinary procedures. In the analogy of the Fraud Diamond, we are attempting to undermine both rationalisation and capability dimensions in order to prevent plagiarism.

Naturally, a student who believes that they are going to fail the exam could rationalise that they there is no cost to attempting plagiarism. An emphasis on the fact that the disciplinary procedures of the university are more serious than a failing grade and strictly applied in cases of plagiarism may provide some mitigation.

4.4.2 Automatic Grading. A number of unit tests are supplied to the students before the beginning of the exam along with the scaffold code. These are only a subset of the tests used in the automatic grading part of the exam grade. The goal behind this decision was to encourage/require students to consider relevant edge cases and to test for them in their own solution. The subset of the tests is chosen such that if all of the supplied tests are passed, then the student will have at least a passing grade (presuming they pass the video assessment). This gives students a reasonable goal to strive for and pushes high achieving students to excel.

4.4.3 Code Style. Although a large component of the grading is automated, this desire to assess students on aspects of code style that are poorly measured by computers means that manual inspection of the code is required. An example here is inheritance: a student could pass some of the unit tests without exhibiting best practice; coding style captures this difference in the exam grade. While this inspection is taking place, errors that would prevent the compilation and execution of the unit test are fixed and detailed feedback is recorded. These fixes are minor in nature (e.g. removing additional spaces in generated Strings, typos in method signatures, etc.) and specifically do not seek to effect the result(s) of the unit tests. This review is typically completed while the student's video is playing, though it may be consulted later if the understanding of the student is in doubt. During the inspection process, thorough feedback is prepared for each student. This details the deficiencies identified in the code style portion, fundamental errors in the question, comments on approach to the solutions as well as comments on the effectiveness of the video explanations.

4.4.4 Feedback. The output of the unit tests is formatted and combined with a detailed explanation of what each of the tests was assessing. Combined with the feedback prepared in the inspection, this is delivered to each student individually. More general feedback is made available to the whole class describing common mistakes for each question, the maximum and average grades and the distribution of which students attempted which questions.

4.4.5 Plagiarism Detectors. The scaffolded nature of the assignments and the influence of the specifications and unit tests present a particular challenge to the use of plagiarism detectors. These tools are still used to indicate the pairs of submissions that are most probably to be a result of collusion, but still requires thorough manual inspection carried out after grading has been completed.

5 OBSERVABLE OUTCOMES

Perhaps the most poignant observation is that this assessment strategy acts as a wake up call for weaker students; the video in particular. There are several instances of students that do not submit the video if they perceive there to be no point; e.g. they will fail anyway (because of the provided unit tests). There are occasionally cases where the student submits "a video" because it is required, which constitutes a form of self-efficacy and self-reflection. Here common examples include videos where students discuss how or why they could not (successfully) undertake the coding problem(s).

Aligned to this is the general (reported via course evaluation) student view that the process is not fair, too hard, or lacking sufficient completion time. Anecdotally it seems that this view changes as students progress through the programme and begin to value the experience. This may become apparent after their first exposure to the work place either through a whiteboard interview or coding challenge as part of a recruitment process, and possibly during internships. This observation would, however, need to be formalised to solidify findings and pedagogical reasoning.

Students have very little room for "grade negotiation": the unit tests illustrate problems with code, video reflection causes self-evaluation of performance, students have a subset of these tests prior to submission so they (should) know if they did well or not even prior to receiving feedback, and the personalised feedback from the hidden tests comes on top of this. Over 3 years, only a handful of requests for additional feedback ($n < 10$) have occurred. Even students that fail the video question and subsequently got 0 in the assessment ask for feedback rather than question the outcome.

The videos can also act as an additional mechanism of code quality review: students may focus their discussion on something the assessor missed, or which wasn't (initially) in the grading criteria. Similarly, students can focus their discussion on parts of the question they did not complete, and receive partial credit for this. Closely related to this, weaker students often struggle to answer the video questions, and attempts to bluff tend to be quite obvious. This stems from the nature of the questions: reflective video questions where the student has to comment on why certain activities were undertaken or aspects of the problem that were most challenging tend to reveal areas of misunderstanding, misconceptions of the course and other fundamental issues with their submission(s).

Finally, for cases of suspected academic misconduct, the video can act as substantial evidence of misconduct. Here generally, the student couldn't explain the submission purported as their own, or has major misconceptions of their submission and/or its quality. An example here is when the student claims their code to pass all unit tests, but their video illustrates otherwise.

6 LESSONS, CONCLUSIONS AND RECOMMENDATIONS

The final design of the programming assessment has evolved over the last four years, with the last three being influenced by the pandemic. This section details some of the lessons that were learned during that evolution, some of the changes still being considered and overall conclusions about this approach to assessment.

The testing strategy employed in an exam heavily affects the skills being assessed. In other words, a completely output-based test can give freedom of design to students, however these are often marked in a pass/fail manner. Thus some level of manual inspection is required to assess the quality of the design (and give partial credit in the result that one or more tests does not pass). Unit tests can be configured to reward students for partially correct solutions, by having a more fine grained testing strategy, but require that the specification(s) of the question are much more specific. This, however, can create situations where the assessment of students' design skills is relatively shallow. Other assessments within the course can counter this. Choosing between these assessment and grading design options (output vs. unit tests, coarse vs. fine grained testing, automated vs. style and best practice assessment) should consider the skill(s) and course learning objective(s) are being evaluated.

The time commitment for this assessment strategy is significant, but scalable in the number of assessors (assuming they have appropriate programming competence). This is currently at the limit of what we consider appropriate for small-scale ($n < 3$) assessment teams, as class sizes increase, there may be a need to reduce or reprioritise the time spent manually inspecting code. This experience report reflects on introductory level programming courses. As such, were problem complexity to increase this would have implications for how specific parts of the grading workflow are instrumented.

Students should be given sufficient opportunity to experience the conditions and expectations of the exam. This can be for low or no stakes in the course, but should give them the experience of all of the principle components of the exam. For example, download of scaffolding, completion of questions, the specific requirements of the tests and how it will score their code. For class A, they do a small(er)-scale assignment on a previous exam question to familiarise them with unit testing based assessment of code quality. Students have the opportunity to (for extra credit) demo lab work in practical classes in a form similar to the video questions. Students that avail of this score significantly higher in the practical exam.

Students must be clearly informed about how the assessment is graded and of the importance of the video explanations. Additionally, the detection tools and consequences of academic integrity violations should be emphasised. This should be done as early as possible before a student could potentially fall too far behind in their study of the class materials, and also to give the student enough time to process the implications and manner of the exam and its execution. This can (and perhaps should) include providing past exams as a point of reference (with the associated unit tests).

It should be noted that a small portion of the class was observed completely ignoring the supplied unit tests, preferring instead to less formally (if at all) evaluate their solutions. Some simply do not use them, others reconstruct tests based on example output in the exam paper using additional main methods with consideration of the console output. Essentially this behaviour is a form of "false friend" to the student in that it can give the impression that the code is "good enough" whereas in reality it would fail multiple unit tests, have method signature (or other code) errors and general oversights that the unit tests would otherwise draw attention to.

There are still a number of aspects of the assessment process that can be improved and which we are currently considering. First, is to require that the task in the video is to show the execution and results of the unit tests. This is to try and combat the population of the course that do not use the tests, or who do not understand their importance. Second, to integrate the testing more explicitly into the submission process. For example, integration with the CI/CD workflows of GitHub (as the mechanism to run the unit tests) as this would better align with modern day software development practices. This would also have the added benefit of generating a trace of how the student attempts the question(s) giving more feedback and self-reflection opportunities. Finally, the introduction of a low stakes practice assessment (in the same vein) would help make students aware of the complexity of the task awaiting them and tangibly incentivise their preparation for it.

REFERENCES

[1] Ibrahim Albluwi. 2019. Plagiarism in Programming Assessments: A Systematic Review. *ACM Trans. Comput. Educ.* 20, 1, Article 6 (dec 2019), 28 pages. https://doi.org/10.1145/3371156

[2] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 500–506. https://doi.org/10.1145/3545945.3569759

[3] Jeremiah Blanchard, John R. Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. 2022. Stop Reinventing the Wheel! Promoting Community Software in Computing Education. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (Dublin, Ireland) *(ITiCSE-WGR '22)*. Association for Computing Machinery, New York, NY, USA, 261–292. https://doi.org/10.1145/3571785.3574129

[4] Simon Caton, Seán Russell, and Brett A. Becker. 2022. What Fails Once, Fails Again: Common Repeated Errors in Introductory Programming Automated Assessments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 955–961. https://doi.org/10.1145/3478431.3499419

[5] Robert Clarke and Thomas Lancaster. 2013. Commercial Aspects of Contract Cheating. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (Canterbury, England, UK) *(ITiCSE '13)*. Association for Computing Machinery, New York, NY, USA, 219–224. https://doi.org/10.1145/2462476.2462497

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

[7] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '19)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3364510.3366170

[8] David J. Emerson and Kenneth J. Smith. 2022. Student Use of Homework Assistance Websites. *Accounting Education* 31, 3 (2022), 273–293. https://doi.org/10.1080/09639284.2021.1971095 arXiv:https://doi.org/10.1080/09639284.2021.1971095

[9] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[10] Judith Gulikers, Theo Bastiaens, and Paul Kirschner. 2006. Authentic Assessment, Student and Teacher Perceptions: The Practical Value of the Five-dimensional Framework. *Journal of Vocational Education & Training* 58, 3 (2006), 337–357. https://doi.org/10.1080/13636820600955443

[11] Judith T. M. Gulikers, Theo J. Bastiaens, and Paul A. Kirschner. 2004. A Five-Dimensional Framework for Authentic Assessment. *Educational Technology Research and Development* 52, 3 (2004), 67–86. https://doi.org/10.1007/BF02504676

[12] Elizabeth A. Hall, Christina Spivey, Hailey Kendrex, and Dawn E. Havrda. 2021. Effects of Remote Proctoring on Composite Examination Performance Among Doctor of Pharmacy Students. *American Journal of Pharmaceutical Education* 85, 8 (2021). https://doi.org/10.5688/ajpe8410 arXiv:https://www.ajpe.org/content/85/8/8410.full.pdf

[13] Richard Lobb and Jenny Harlow. 2016. Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (feb 2016), 47–51. https://doi.org/10.1145/2810041

[14] Terence Nip, Elsa L. Gunter, Geoffrey L. Herman, Jason W. Morphew, and Matthew West. 2018. Using a Computer-Based Testing Facility to Improve Student Learning in a Programming Languages and Compilers Course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 568–573. https://doi.org/10.1145/3159450.3159500

[15] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (jun 2022), 40 pages. https://doi.org/10.1145/3513140

[16] Jonathan Pierce and Craig Zilles. 2017. Investigating Student Plagiarism Patterns and Correlations to Grades. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 471–476. https://doi.org/10.1145/3017680.3017797

[17] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding Plagiarisms Among a Set of Programs With JPlag. *J. Univers. Comput. Sci.* 8, 11 (2002), 1016.

[18] Charles P. Riedesel, Alison L. Clear, Gerry W. Cross, Janet M. Hughes, Simon, and Henry M. Walker. 2012. Academic Integrity Policies in a Computing Education Context. In *Proceedings of the Final Reports on Innovation and Technology in Computer Science Education 2012 Working Groups* (Haifa, Israel) *(ITiCSE-WGR '12)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2426636.2426638

[19] Seán Russell. 2022. Automated Code Tracing Exercises for CS1. In *Proceedings of 6th Conference on Computing Education Practice* (Durham, United Kingdom) *(CEP '22)*. Association for Computing Machinery, New York, NY, USA, 13–16. https://doi.org/10.1145/3498343.3498347

[20] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[21] Judy Sheard, Martin Dick, Selby Markham, Ian Macdonald, and Meaghan Walsh. 2002. Cheating and Plagiarism: Perceptions and Practices of First Year IT Students. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (Aarhus, Denmark) *(ITiCSE '02)*. Association for Computing Machinery, New York, NY, USA, 183–187. https://doi.org/10.1145/544414.544468

[22] Angela A. Siegel, Mark Zarb, Bedour Alshaigy, Jeremiah Blanchard, Tom Crick, Richard Glassey, John R. Hott, Celine Latulipe, Charles Riedesel, Mali Senapathi, Simon, and David Williams. 2022. Teaching through a Global Pandemic: Educational Landscapes Before, During and After COVID-19. In *Proceedings of the 2021 Working Group Reports on Innovation and Technology in Computer Science Education* (Virtual Event, Germany) *(ITiCSE-WGR '21)*. Association for Computing Machinery, New York, NY, USA, 1–25. https://doi.org/10.1145/3502870.3506565

[23] Sarah Silverman, Autumm Caines, Christopher Casey, Belen Garcia de Hurtado, Jessica Riviere, Alfonso Sintjago, and Carla Vecchiola. 2021. What Happens When You Close the Door on Remote Proctoring? Moving Toward Authentic Assessments with a People-Centered Approach. *To Improve the Academy* 39, 3 (mar 2021). https://doi.org/10.3998/tia.17063888.0039.308

[24] Simon, Beth Cook, Judy Sheard, Angela Carbone, and Chris Johnson. 2013. Academic Integrity: Differences between Computing Assessments and Essays. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '13)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2526968.2526971

[25] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, and Jane Sinclair. 2018. Informing Students about Academic Integrity in Programming. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 113–122. https://doi.org/10.1145/3160489.3160502

[26] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. 2016. Negotiating the Maze of Academic Integrity in Computing Education. In *Proceedings of the 2016 ITiCSE Working Group Reports* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, NY NY, USA, 57–80. https://doi.org/10.1145/3024906.3024910

[27] Patriel Stapleton and Jeremiah Blanchard. 2021. Remote Proctoring: Expanding Reliability and Trust. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1243. https://doi.org/10.1145/3408877.3439671

[28] Tammy VanDeGrift. 2022. Post-Exam Videos for Assessment in Computing Courses: See and Hear Students' Thinking. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 230–236. https://doi.org/10.1145/3478431.3499273

[29] Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 53, 4 pages. https://doi.org/10.1145/3411764.3445228

[30] Mark Zarb and Jen BirtlesKelman. 2020. Through the Lens: Enhancing Assessment with Video-Based Presentation. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 187–192. https://doi.org/10.1145/3341525.3387376