The Future of Engineering Education
2024 Annual Conference & Exposition
Oregon Convention Center
Portland, OR . June 23 - 26, 2024
ASEE
Paper ID #43649

Providing High-Quality Formative Feedback for Database Assignments

Huanyi Chen, University of Waterloo
Prof. Paul Ward, University of Waterloo

Providing High-Quality Formative Feedback for Database Assignments

Abstract

Automated systems such as Marmoset, WebCAT, OK, MarkUs, and many others are widely used in assessing programming assignments. Although they enable instructors to assess students' solutions at scale, the core infrastructure of these systems is not much different from a standard build and test environment, which focuses on ensuring correct solutions. However, when it comes to learning, it would be more important to assist students in correcting their misconceptions when their solutions are incorrect, i.e., provide a feedback message accurately showing them what is wrong and what they can do. The latter, which requires high-quality assessment and considerable effort in composing feedback, however, is rarely discussed, not to mention that no tools or support have been developed in these systems to assist in writing them. In this paper, we aim to fill the gap by providing guidance for assessment writers to write effective assessments and feedback for students' solutions. We present an approach to properly organizing the test cases so that automated assessments can identify students' misconceptions accurately, enabling them to provide high-quality formative feedback to rectify students' misconceptions. Following the guidance outlined, we developed assessments for a database course. By comparing student performance with and without the high-quality formative feedback, we observed an overall improvement in RA of 21%, with a 73% improvement in query creation and an 11% improvement in ER, with a 32% improvement in composing new relationship sets and/or specializations.

1 Introduction

Test-based assessment is often incorporated into automated systems to provide formative feedback on students' work [1, 2, 3, 4]. The standard procedure of such assessments follows the build-and-test philosophy. When the system receives a new student submission, it runs all pre-coded tests on the submission, then it delivers the test outcomes to students, where each outcome is usually in the form of either pass or fail. Instructors expect students to understand what is correct and what is not based on these test outcomes.

Although test-based automated assessments enable instructors to deliver timely feedback at scale, they come with several limitations. Most notably, while they are designed at identifying correct solutions, they provide limited hints for students who produce incorrect solutions. Furthermore, they can lead to confusion when test failures stem from issues in the assessment code rather than the students' work.

While educators have tried to convert such test outcomes into more effective feedback, such as in the form of an actionable sentence, such systems rarely offer guidance on how to compose such feedback. They largely rely on instructors' expertise to provide meaningful feedback [5]. Acknowledging the importance of formative feedback [6, 7], researchers have expended effort to develop automated feedback tools that support instructors in generating effective feedback [8, 4, 9]. However, since many of these tools are tailored for courses tied to particular platforms or systems, their applicability in different contexts becomes challenging [10].

Moreover, most systems tend to de-emphasize the necessity of human intervention in automated assessments—a design decision we believe to be unwise. This implies that assessment writers often fail to distinguish between test failures due to bugs in the assessment code and those due to errors in students' work. If such test failures are revealed, they can mislead students into believing the fault lies within their work, leading them to spend time addressing non-existent issues. Thus, it is crucial to design automated systems in such a way that they prompt for human intervention upon encountering test failures due to unforeseen problems.

Therefore, in this paper, we propose guidance for writing assessments that greatly diverges from the existing build-and-test philosophy. We have also developed a tool designed to assist assessment writers in computing and engineering education in adopting our innovative approach. This tool will be made publicly accessible.

To test the effectiveness of this guidance, we developed assessments for database assignments based on it. The ultimate research question we aim to answer is:

- Do students learn better with our formative feedback?

We examined students' performance between two terms, with and without formative feedback, and found promising results.

2 Related Work

The automated grading and feedback system has a long history [11, 12]. These systems were developed to reduce the human effort required in assessing students' solutions to programming questions [11, 13]. Upon receiving a new submission, the system triggers one or more workers to run assessment tests, either sequentially or concurrently, against the student's work. After completion, the system gather the results and processes these outcomes by, for example, aggregating passes and failures, visualizing code coverage statistics, and then offering feedback to the students. This approach is rooted in standard build and test practices [14] and is embraced by numerous automated systems [15, 16, 17, 4, 18].

However, it is often the case that simple pass and fail test outcomes are not effective as feedback. Therefore, educators have sought to augment them by integrating additional information to make the feedback more effective. A prevalent modification involves categorizing tests. Based on these categories, the system can decide whether to display specific outcomes. This method helps shield students from an overload of test results [19, 17, 20]. For instance, Marmoset [17] prompts instructors to classify tests as build, public, release, or secret tests. Though all tests are run if the build test is successful, only the results of the public tests and few selected release tests are displayed. Lee [19] grouped tests into sample tests, unlockable tests, and hidden tests. When students' code surpasses a threshold of sample tests, their grading system discloses one of the failed unlockable tests.

To curate feedback tailored to students' misconceptions, tools enabling flexible feedback have been crafted. For example, Gusukuma et al. [8] introduced a feedback model wherein an interface facilitates the interaction of condition-response pairs with an underlying structure. An exemplary tool for this model is Pedal [8]. On the other hand, hint generation tools offer suggestions to assist students to transform their current program towards a solution state [21, 22].

While being less discussed, however, enhancing feedback quality is an iterative process. Rarely does feedback set up initially function optimally. Some tools permit human intervention to refine the feedback [23, 24, 25, 3]. For example, Head et al. [24] showcased a mixed-initiative method that combine instructors' profound domain knowledge with solution structures derived through program synthesis techniques. However, these tools typically cater to specific programming languages, such as Python, since they need to analyze the abstract syntax tree (AST). We aim to demonstrate that refining feedback from a more holistic perspective for other languages is feasible. For example, even if the assessment code is written in Python, it can be used to assess students' work done in other programming languages, such as the structured query language (SQL).

3 Problems

Different automated systems offer various features to support educators' needs. However, the essential component, automated assessments, is composed by assessment writers. The role of automated systems is no more than that of a standard build and testing environment. In this paper, we focus on problems in writing assessments, hoping to provide guidance to assessment writers to compose better automated assessments.

From our experience, there are primarily two categories that cause assessments to give incorrect feedback: 1) **false positives**, where incorrect components are marked as correct; 2) **false negatives**, where correct components are marked as incorrect. We will discuss why it is difficult to detect the former, and we will focus on dealing with the latter (false negatives) category in this paper. Although we will primarily focus on database courses, the proposals in this paper can be applied to other courses as well.

3.1 False positives: incorrect components marked as correct

When an automated assessment is deployed, the assessment writer expects it to function correctly. Therefore, if a solution is marked as correct, the assessment writer would consider that solution correct. There would not be motivation for the assessment writer to retrospectively examine the assessment code.

On the other hand, assessment writers become aware of defective code that causes false negatives, as students are usually motivated to complain when they have correct solutions marked as incorrect. However, students are less motivated to report false positives. It is akin to students requesting point deductions upon realizing they received a higher grade than deserved.

Therefore, detecting this category is very challenging. One potential solution could be to periodically sample student solutions for human inspection.

3.2 False negatives: correct components get marked as incorrect

In test-based assessments, a test can fail for many reasons, far beyond what the assessment writer might expect. For example, to verify the accuracy of a SQL query, a test might execute both the standard SQL query and the student's SQL query against the same database, then check if the results match. If the results do not align, the test fails. However, is a mismatch in results the sole cause of test failure? It is not. The test can also fail due to a connection failure, where the assessment code fails to connect to the database; or it might fail due to insufficient memory to store the returned result, among other potential causes. The key point is that it is often more challenging than anticipated to create assessment code that only fails for predefined reasons. Students are likely to be confused if the test indicates failure when they believe their work is error-free.

This is is referred to as the *Unexpected Failure* problem in the paper.

The situation worsens with multiple tests. Although many assessment writers adopt the Test-Driven Development (TDD) philosophy, where each test is considered isolated from others, it is common to find dependencies among tests. For example, assessments cannot proceed without a submitted file or if the file is incorrectly named. Additionally, the runtime results of C/C++ and/or Java programs can only be evaluated if the code is compilable.

Table 1 illustrates the outcomes a student would encounter if dependencies among test properties are not considered. In this scenario, suppose each property partially relies on the preceding property to pass (*i.e.*, property B depends on property A, property C depends on property B, and property D depends on property C). In essence, if the student's solution fails to meet property B, subsequent properties are bound to fail as well. However, without visibility into the underlying assessment code (which is common), students are unaware of these implicit dependencies. Consequently, they might mistakenly believe there are issues with properties other than property B, even though resolving the issue with property B would lead to all tests passing. This lack of transparency can mislead students into thinking there are multiple errors in their solution, diverting their focus from the actual root cause.

name	**outcome**
check_property_A	passed
check_property_B	failed
check_property_C	failed
check_property_D	failed

Table 1: Test outcomes

This is referred to as the *Missing Dependency* problem in the paper.

3.3 Improving existing assessments

Assuming that an assessment writer identifies a bug in a test or a dependency between tests, amending the existing assessment code to address these issues is not straightforward. Any fix carries the risk of introducing additional bugs, potentially complicating matters further. It is often suggested that any new code should undergo comprehensive testing before deployment. Yet, the concept of "testing thoroughly" lacks a clear definition in automated assessments. This ambiguity presents a challenge in ensuring that the new code does not adversely affect the assessment's integrity or introduce unforeseen complications. The need for a well-defined testing strategy is critical to mitigate such risks and ensure that modifications to the assessment code enhance its reliability and effectiveness.

This is referred to as the *Assessment Iteration* problem in the paper.

4 Methods

Before we introduce our methodologies for solving the aforementioned problems, we want to clearly define the terminologies for two categories of assessments and three categories of tests.

Assessment Code This refers to the type of automated assessment conducted by current automated systems or tools. It consists of tests designed to assess the quality of a student's work. The outcomes of these tests inform students about their performance on the assignment.

Assessment on Assessment Code This type of assessment ensures that the introduction of new code does not bring bugs or inconsistencies. It addresses the "assessment iteration" problem. This is similar to software quality assurance in software development.

Probing Tests These tests assess the quality of a student's work. They are a part of the assessment code.

Diagnosing Tests This newly introduced category includes tests designed to diagnose the causes of probing test failures, in order to provide precise guidance for students to improve their work. These tests are also a part of the assessment code. Together, probing and diagnosing tests make up the assessment code.

Regression Tests These tests belong to the assessment on assessment code category and are its only test category. They can be considered standard tests, such as unit tests or integration tests, if developing assessment code is deemed the same as developing normal software.

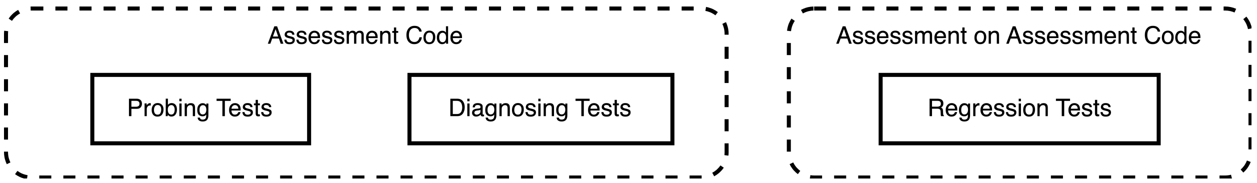


Figure 1: Relationships between test and assessment categories

Figure 1 shows the relationships among them. In summary, the assessment code contains probing and diagnosing tests, while assessment on assessment code contains regression tests. We will use these terminologies throughout the remainder of the paper.

4.1 Eliminating false negatives

To address false negatives, we must tackle the issues of "unexpected failure" and "missing dependency."

Consider a probing test that consists of only one line, which is an assertion. If this test fails, we can attribute the failure directly to that specific assertion, thereby understanding the reason behind the test's failure.

However, complications arise when a probing test includes multiple lines and has the potential to fail before reaching the assertion statement. One strategy could involve duplicating the probing test and inverting the assertion in the duplicate. Consequently, if the original test fails, the duplicate will pass, indicating that the failure in the original test occurred at the assertion statement.

```
# If one passed and the other failed
# we know they both reached the assertion line
def test_a():
    x = common_code()
    assert x == 0

def test_a_flipped():
    x = common_code()
    assert x != 0
```

Alternatively, if a probing test fails for reasons other than assertions—implying it encountered exceptions other than assertion errors—it should theoretically run to completion. To identify such scenarios, we can encapsulate the entire probing test in a comprehensive `try ... catch/except` block, ensuring it is marked as failed only if an assertion error is detected.

```
def test_a():
    try:
        x = cause_unexpected_failure()
        assert x == 0
    except AssertionError as e:  # known failure
        raise ExpectedError from e
    except Exception as e:  # capture general case
        raise UnexpectedError from e
```

This method presumes that each test comprises a single assertion or a contiguous block of assertions. However, dealing with tests that contain multiple, scattered assertions presents a challenge. A possible solution is to divide such a test into multiple smaller tests, each focusing on a single assertion, with dependencies among these segmented tests.

Then, how do we handle unexpected failures? The most appropriate approach is to inform a human to inspect. Once the cause is identified, huamn feedback can be provided for the solution. Possibly, new code may be introduced to detect the newly determined cause. The procedure for handling unexpected failures is depicted in Figure 2.

At this point, the "unexpected failure" problem is handled properly.

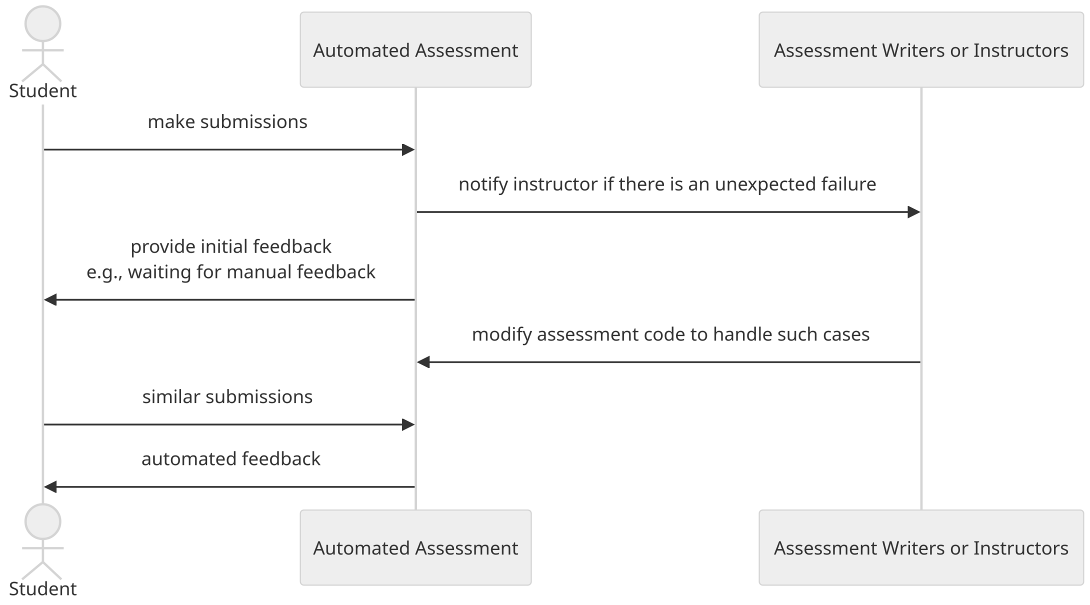

Student
Automated Assessment
Assessment Writers or Instructors
make submissions
notify instructor if there is an unexpected failure
provide initial feedback
e.g., waiting for manual feedback
modify assessment code to handle such cases
similar submissions
automated feedback
Student
Automated Assessment
Assessment Writers or Instructors

Figure 2: Handling unexpected failures

Regarding the "missing dependency" issue, upon closer examination, it is not truly an implementation problem. Instead, it is a problem that assessment writers often fail to acknowledge. The reason is that many assessment writers and developers of automated systems adhere to the TDD philosophy, which advocates for isolated tests when designing probing tests. However, isolation may not be appropriate in assessments within an educational context. This issue can be addressed if assessment writers recognize it and consider it when composing probing tests.

4.2 Adding new code safely

Standard TDD requires developers to write test cases before the software is fully developed, allowing new changes to be verified and validated. A test case specifies the inputs, execution conditions, testing procedure, and expected results, defining an executable test to achieve a particular software testing objective. In the context of assessment code, these are referred to as regression tests.

For example, if the solution is expected to satisfy property A but not B, C, and D, then both before and after changes to the assessment code (such as fixing bugs or code refactoring), the test outcomes (happen to be the same as in Table 1) for the solution should remain unchanged. In other words, if the input is the solution, then the output should be the same test outcomes.

Regression tests might not be needed if the assessment code is relatively simple. However, current automated assessments have been growing in complexity. Furthermore, the purpose of any given assessment is rarely spelled out explicitly. The purpose of the assessment should always be clearly defined; once it is, testing whether the assessment code fulfills that purpose becomes relevant.

4.2.1 An applicable example

We assume the database used for the assignment is the **world** database[1], and we have loaded it using MySQL. The assessment aims to verify whether the student's query returns correct results for the question: *What are the unique city names in the world database?*

One possible assessment code consists of two steps:

1. Execute both the student's query and the instructor's query to retrieve the table rows.
2. Sort the rows and then compare them.

Below is the function used in the probing test:

```
def check_correctness(stu_sql):
    inst_sql = "SELECT DISTINCT `name` FROM `city`;"
    inst_rows = get_rows(inst_sql)
    inst_rows_sorted = sorted(inst_rows)
    # Similarly, for the student's query
    stu_rows = get_rows(stu_sql)
    stu_rows_sorted = sorted(stu_rows)
    # Perform the comparison
    return student_rows_sorted == inst_rows_sorted
```

In order to ensure we remember to include `DISTINCT` in the instructor query when we change the `check_correctness` function, we created the following regression test.

```
def test_ensure_distinct():
    sql_no_distinct = "SELECT `name` FROM `city`;"
    sql_has_distinct = "SELECT DISTINCT `name` FROM `city`;"
    assert check_correctness(sql_no_distinct) == False \
            and check_correctness(sql_has_distinct) == True
```

However, later we find it might be clearer to use `ORDER BY` within `inst_sql`. Therefore, we updated the `check_correctness` code as follows:

```
def check_correctness(stu_sql):
    # Use ORDER BY for instructor's query
    inst_sql = "SELECT DISTINCT `name` FROM `city` ORDER BY `name`;"
    inst_rows_sorted = get_rows(inst_sql)
    # No change for student's query
    stu_rows = get_rows(stu_sql)
    stu_rows_sorted = sorted(stu_rows)
    # Perform the comparison
    return student_rows_sorted == inst_rows_sorted
```

This change would cause `test_ensure_distinct` to fail because `sorted()` in Python sorts rows differently from `ORDER BY` in SQL by default, as illustrated by Table 2. Despite not being its original intention, the regression test detected a potential discrepancy in our assessment code.

[1]`https://dev.mysql.com/doc/world-setup/en/world-setup-installation.html`, accessed: 2024-03-29

SQL	Python
`'[San Cristóbal de] la Laguna'`	`'A Coruña (La Coruña)'`
`''s-Hertogenbosch'`	`'Aachen'`
`'A Coruña (La Coruña)'`	`'Aalborg'`
`'Aachen'`	`'Aba'`
`'Aalborg'`	`'Abadan'`
...	...

Table 2: Differences in sorting results between SQL's `ORDER BY` and Python's `sorted()`

4.3 Providing high-quality feedback

Understanding why a probing test fails is important. However, even more crucial is informing students how they can improve. For example, if a student's SQL query does not yield the expected results, could it be due to the absence of `DISTINCT`? Or is it because they used the wrong `JOIN` (i.e., a `LEFT JOIN` was expected but the student used an `INNER JOIN`)? Or is it simply because the student returned incorrect attributes? Anecdotally, we have observed many students making countless submissions to pass a single probing test, mainly because they have no clue about their mistakes or how to fix them.

Diagnosing tests are employed for this purpose. These tests will not run if probing tests are passed; however, if a probing test fails, diagnosing tests that depend on its failure will be executed. For example, considering the previous example of mismatched returned results, one could code three diagnosing tests such as "missing_distinct", "check_join_type", and "check_attributes". The outcomes of these tests can guide the student on what steps to take next.

probing tests	**outcome**
check_property_A	passed
check_property_B	failed
check_property_C	skipped
check_property_D	skipped

diagnosing tests	**outcome**
incorrect_component_X	passed
incorrect_component_Y	failed

Table 3: Test outcomes of probing tests and diagnosing tests considering dependencies

Table 3 presents the enhanced test outcomes of the assessment code, clearly indicating that a solution fails to meet property B, with component X being at least partially responsible. However, in the absence of passing diagnostic tests, students may need to identify the underlying cause by themselves.

The transition from Table 1 to Table 3 marks a notably improvement in the precision of automated feedback. Nonetheless, we argue it is necessary to transform raw test outcomes into actionable feedback. For example, the feedback derived from Table 3 could be framed as, *"Your program fails to meet property B. Perhaps double check component X"*.

There are at least two advantages for doing that:

- It reduces the cognitive burden on students in interpreting the feedback; and
- It introduces an additional layer of abstraction, enabling assessment authors to conceal the specifics of the assessment code, thus preventing students from knowing the exact tests and what they are assessing.

This transformation can be achieved by establishing a mapping between the set of test outcomes and the corresponding actionable feedback.

5 Actual Assessments

The actual assessments were implemented using the popular unit test framework **pytest** [26]. Although it does not natively support test dependencies, this functionality can be achieved with the **pytest-dependency** plugin[2]. Despite the assessment code being written in Python, there is no inherent connection or limitation to what it can assess.

For SQL questions, the assessment code essentially verifies whether students' queries yield the same results as the canonical solutions.

For relational algebra (RA) questions, students were required to use a novel QWERTY-Compatible Syntax (QCS) for coding their queries. For example, to identify the names of employees who earn the highest salary, the RA query can be

```
p[name]s[salary in (G[max(salary)]Employee)] Employee;
```

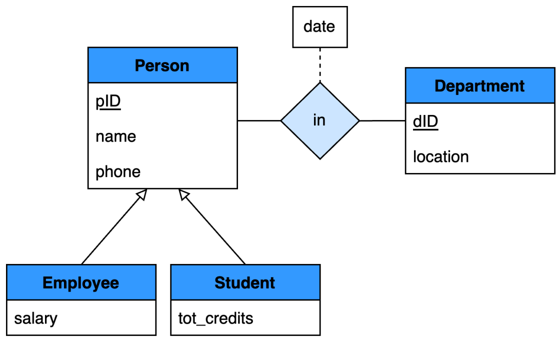
where `p` is for projection, `s` is for selection, and `G` is for aggregation. Additionally, the join operation (`JOIN`) is supported; for example, A natural join B is represented as `A |><| B`. The assessment code focuses on syntax errors, using a novel RA parser to detect such errors. It then uses regular expressions to identify the specific part causing the error, transforming the parser's message into actionable feedback.

For example, if one forgets the `Employee` in `G[max(salary)Employee]`, the syntax error, "syntax error at line 1 column 35," would be translated into feedback like, "At line 1, you forgot to name the table on which you intended to apply the aggregation." Incorporating semantic error checking is planned for future work.

For entity relationship (ER) questions, we provide a special database schema involving tables such as "Entities", "RelationshipSets", "Attributes", "IsA" (i.e., specialization), etc., enabling students to construct their ER designs by writing insertion statements in SQL. Figure 3 shows an example.

The steps for assessing ER designs can be summarized as follows: 1) Executing the student's SQL file; 2) Checking whether all essential entity sets, as presented in all valid designs in the canonical solution set, were present in the resulting relational tables; 3) Searching for elements that must not exist, such as extra specializations; 4) Matching the student's design to an exemplary design using a set of rules, such as checking the involved entity sets in each relationship set between

[2]`https://pypi.org/project/pytest-dependency/`



```
EntitySets
+------------+------------+
| esName     | primaryKey |
+------------+------------+
| Person     | pID        |
| Department | dID        |
+------------+------------+
```

```
IsA
+---------------+---------------+----------+-------+
| specializedES | generalizedES | disjoint | total |
+---------------+---------------+----------+-------+
| Employee      | Person        | 0        | 0     |
| Student       | Person        | 0        | 0     |
+---------------+---------------+----------+-------+
```

```
RelationshipAttributes
+------------+--------+
| rsAttrName | rsName |
+------------+--------+
| date       | in     |
+------------+--------+
```

```
RelationshipSets
+--------+------------+-------+-------+
| rsName | esName     | lower | upper |
+--------+------------+-------+-------+
| in     | Person     | 0     | *     |
| in     | Department | 0     | *     |
+--------+------------+-------+-------+
```

```
Attributes
+-------------+------------+-------------+
| attrName    | esName     | multivalued |
+-------------+------------+-------------+
| pID         | Person     | 0           |
| name        | Person     | 0           |
| phone       | Person     | 0           |
| dID         | Department | 0           |
| location    | Department | 0           |
| salary      | Employee   | 0           |
| tot_credits | Student    | 0           |
+-------------+------------+-------------+
```

Figure 3: Sample ER design and its encoding

the student's design and every design in the canonical solution set; 5) Evaluating the remaining components, such as attributes and constraints. Note that students were encouraged to reach out to instructors if their design did not match any of the pre-coded designs.

For all assessments, dependencies were carefully configured, such that the first probing test is to check if the submitted file is named correctly. Emails were used to inform instructors and assessment writers when human inspection was needed.

6 Evaluation

This is the first time we introduced automated assessment. Therefore, we chose to examine the final exam performance in SQL, RA, and ER design questions across two terms—with and without formative feedback—to determine if students achieved better learning outcomes. The same instructor taught the course in both terms, and there were no substantial changes in homework assignments. Additionally, the questions on both final exams covered the same topics and were very similar, with only small variations. The ER question was the same, with two sub-questions. The first sub-question asked students to identify primary keys or discriminators, weak-entity sets, or participation constraints that were intentionally removed from the ER model. The second sub-question asked students to compose new relationship sets and/or specializations to enhance the existing ER design. Both final exams were in-person and closed book, and were marked using the same rubric by the same person.

We applied the two-sided Welch's t-test to compare students' grade percentages between the two terms for each type of question. The two-sided Welch's t-test is a statistical method used to test the hypothesis that two populations have equal means, where the populations do not necessarily have the same variance (heteroscedasticity) and may have different sample sizes.

Table 4 presents the results of the statistical tests. It reveals that there was no significant difference in students' performance on SQL comprehension and creation questions, nor was there a significant difference in performance on RA comprehension questions. However, there was a significant difference in performance on RA creation questions. The aggregate performance on RA question was also significant. The aggregate performance on ER question was close to significant and the performance on ER Q2 was significant.

	SQL Comp	SQL Cr	RA Comp	RA Cr	RA Total	ER Total	ER Q1	ER Q2
mean (without)	81.76	62.35	72.35	38.24	60.98	51.27	63.14	39.41
mean (with)	77.38	59.45	78.09	66.35	73.77	57.73	60.20	51.98
p-value	0.128	0.469	0.181	1.2E-5	6.17E-4	0.058	0.428	0.006

Table 4: Grade percentage statistics with and without feedback ("Comp" stands for "comprehension" and "Cr" stands for "creation")

This indicates an overall improvement in RA of 21%, with a 73% improvement in query creation and an 11% improvement in ER, with a 32% improvement on ER Q2. These results are promising.

7 Discussion and Future Work

We would like to discuss our further considerations in providing high-quality feedback in this section.

7.1 Probing test vs. Diagnosing test

It may not be necessary to distinctly separate probing tests from diagnostic tests. For example, diagnostic tests can exist for other diagnostic tests. The latter can, to some extent, be considered probing tests. This can be considered to creating a chain of tests, similar to how humans investigate problems through a series of checks.

Without using unit testing frameworks, one can implement these tests from scratch. However, this means there might be several features overlapping with unit testing frameworks, such as ensuring a fresh start for every test and generating a report to collect test outcomes. Therefore, in our current implementation, we have tried to leverage unit testing frameworks—with one limitation addressed, where unit testing frameworks do not support conditional test branching, as shown in the pseudo code below.

```
if test_a() passed:
    test_b()
else:
    test_c()
```

In the case of conditional branching, we duplicated the test to be conditionally checked. Then, we made subsequent tests depend on each of these duplicates. For example, in the following code, `test_b` will execute if `test_a` passes; otherwise, `test_c` will execute.

```
def test_a():
    assert x == 0
def test_a_flipped():
    assert x != 0

@pytest.mark.dependency(depends=["test_a"])
def test_b():
    assert y == 0

@pytest.mark.dependency(depends=["test_a_flipped"])
def test_c():
    assert z == 0
```

7.2 Efficiency of Assessments

Introducing dependencies may reduce the efficiency of assessments, but it is meaningless if correctness cannot be guaranteed. Nonetheless, parallelism can still be applied if some tests have no dependencies.

For example, if we have two sets of tests, where `test_a` → `test_b` means `test_b` gets executed only if `test_a` passes.

1. `test_a` → `test_b` → `test_c`
2. `test_e` → `test_f`

Apparently, the first chain (`test_a/b/c`) can be executed in parallel with the second chain (`test_e/f`).

7.3 Policy Decisions

Assessment writers or educators using automated systems must make policy decisions for certain issues. One issue arises when the assessment code is found to be defective and requires a fix. The question then becomes when to apply this fix.

A straightforward decision is to implement the fix immediately. However, this becomes complicated if some students have already submitted their solutions. If they resubmit the same solution and observe a change in the assessment outcome, they might be confused by the inconsistency. The decision may also depend on what is being assessed. If the component being assessed is minor, ignoring the issue might be considered acceptable when the assessment is being in use.

Another scenario requiring policy decisions involves deciding whether to seek human feedback after a probing test fails for expected reasons. This is crucial when the probing test evaluates solutions that are not uniquely correct. For example, probing tests might compare students' ER designs against standard examples. A failed probing test does not necessarily indicate that the student's design is incorrect; it might be a valid design that was not anticipated during the test's development. In such cases, human evaluation is advisable.

In practice, our implementation involves informing a human of any test failure (*i.e.*, through email) during the initial submissions, regardless of whether the failure was anticipated. This approach allows us to quickly gain insights into students' solutions. Subsequently, many diagnostic tests were introduced after deployment. These tests were not pre-coded. They were coded after the assignment has been released.

However, one of the associated issue was that the number of emails could be overwhelming. There could be multiple solutions triggering multiple notifications although they all failed for the same misconception. Ideally, the automated assessment should aggregate similar solutions within a single request for human inspection.

7.4 The Tool

We extract common components that can possibly be reused from one assessment to another into a tool. We hope this facilitates more people adopting our approach to develop their automated assessments. It is publicly accessible[3].

7.5 Artificial Intelligence (AI)

Future work could consider incorporating AI into the automated assessment. For example, if there is a SQL syntax error, instead of displaying the message "ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds..." to students, a more user-friendly response could be generated using GPT-4 [27]. We have observed that GPT-4 provides relatively accurate responses to SQL syntax errors. However, proper validation is still necessary to ensure its performance meets expectations, especially for complex queries such as RA or ER questions.

8 Summary

In this paper, we introduced guidance that diverges significantly from the build-and-test approach currently adhered to by existing automated assessments. Following this guidance, we developed our automated assessments for a database course. By comparing student performance before and after the deployment of the automated assessments, we observed an overall improvement in RA of 21%, with a 73% improvement in query creation and an 11% improvement in ER, with a 32% improvement in composing new relationship sets and/or specializations.

References

[1] Carsten Kleiner, Christopher Tebbe, and Felix Heine. Automated grading and tutoring of SQL statements to improve student learning. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research - Koli Calling '13*, pages 161–168, Koli, Finland, 2013. ACM Press. ISBN 978-1-4503-2482-3. doi: 10.1145/2526968.2526986.

[2] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 26–30, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1-58113-798-2. doi: 10.1145/971300.971312.

[3] Georgiana Haldeman, Monica Babeş-Vroman, Andrew Tjang, and Thu D. Nguyen. CSF: Formative feedback in autograding. *ACM Transactions on Computing Education*, 21(3), May 2021. doi: 10.1145/3445983.

[3] `https://github.com/h365chen/socassess`

[4] Jianxiong Gao, Bei Pang, and Steven S. Lumetta. Automated Feedback Framework for Introductory Programming Courses. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 53–58, Arequipa Peru, July 2016. ACM. ISBN 978-1-4503-4231-5. doi: 10.1145/2899415.2899440.

[5] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 13–17, New York, NY, USA, June 2006. Association for Computing Machinery. ISBN 1-59593-055-8. doi: 10.1145/1140124.1140131.

[6] John Hattie and Helen Timperley. The Power of Feedback. *Review of Educational Research*, 77(1):81–112, March 2007. ISSN 0034-6543, 1935-1046. doi: 10.3102/003465430298487.

[7] Stephen H. Edwards. Automated Feedback, the Next Generation: Designing Learning Experiences. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 610–611, Virtual Event USA, March 2021. ACM. ISBN 978-1-4503-8062-1. doi: 10.1145/3408877.3437225.

[8] Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. Pedal: An Infrastructure for Automated Feedback Systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, pages 1061–1067, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-6793-6. doi: 10.1145/3328778.3366913.

[9] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462195.

[10] Johan Jeuring, Hieke Keuning, Samiha Marwan, Dennis Bouvier, Cruz Izu, Natalie Kiesler, Teemu Lehtinen, Dominic Lohr, Andrew Peterson, and Sami Sarsa. Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education*, pages 95–115, Dublin Ireland, December 2022. ACM. ISBN 9798400700101. doi: 10.1145/3571785.3574124.

[11] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, October 1960. ISSN 0001-0782, 1557-7317. doi: 10.1145/367415.367422.

[12] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Communications of The Acm*, 8(5):275–278, May 1965. ISSN 0001-0782. doi: 10.1145/364914.364937.

[13] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2018 Companion, pages 55–106, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-6223-8. doi: 10.1145/3293881.3295779.

[14] D. Saff and M.D. Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 281–292, Denver, Colorado, USA, 2003. IEEE. ISBN 978-0-7695-2007-0. doi: 10.1109/ISSRE.2003.1251050.

[15] Sarah Heckman and Jason King. Developing Software Engineering Skills using Real Tools for Automated Grading. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 794–799, Baltimore Maryland USA, February 2018. ACM. ISBN 978-1-4503-5103-4. doi: 10.1145/3159450.3159595.

[16] Morgan Magnin, Guillaume Moreau, Nelle Varoquaux, Benjamin Vialle, Karen Reid, Mike Conley, and Severin Gehwolf. MarkUs: An Open-Source Web Application to Annotate Student Papers On-Line. In *Volume 4: Advanced Manufacturing Processes; Biomedical Engineering; Multiscale Mechanics of Biological Tissues;*

Sciences, Engineering and Education; Multiphysics; Emerging Technologies for Inspection and Reverse Engineering; Advanced Materials and Tribology, pages 301–307, Nantes, France, July 2012. American Society of Mechanical Engineers. ISBN 978-0-7918-4487-8. doi: 10.1115/ESDA2012-82141.

[17] Jaime Spacco, William Pugh, Nat Ayewah, and David Hovemeyer. The Marmoset project: An automated snapshot, submission, and testing system. In *Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications - OOPSLA '06*, page 669, Portland, Oregon, USA, 2006. ACM Press. ISBN 978-1-59593-491-8. doi: 10.1145/1176617.1176665.

[18] Stephen H. Edwards and Manuel A. Perez-Quinones. Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, page 328, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 978-1-60558-078-4. doi: 10.1145/1384271.1384371.

[19] Haden Hooyeon Lee. Effectiveness of Real-time Feedback and Instructive Hints in Graduate CS Courses via Automated Grading System. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 101–107, Virtual Event USA, March 2021. ACM. ISBN 978-1-4503-8062-1. doi: 10.1145/3408877.3432463.

[20] Kevin Buffardi and Stephen H. Edwards. Reconsidering Automated Feedback: A Test-Driven Approach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 416–420, Kansas City Missouri USA, February 2015. ACM. ISBN 978-1-4503-2966-8. doi: 10.1145/2676723.2677313.

[21] Thomas W Price, Yihuan Dong, and Tiffany Barnes. Generating Data-driven Hints for Open-ended Programming. page 8, 2016.

[22] Kelly Rivers and Kenneth R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, March 2017. ISSN 1560-4292, 1560-4306. doi: 10.1007/s40593-015-0070-z.

[23] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Bjorn Hartmann. Learning Syntactic Program Transformations from Examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415, Buenos Aires, May 2017. IEEE. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.44.

[24] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, pages 89–98, Cambridge Massachusetts USA, April 2017. ACM. ISBN 978-1-4503-4450-0. doi: 10.1145/3051457.3051467.

[25] Chinmay E. Kulkarni, Michael S. Bernstein, and Scott R. Klemmer. PeerStudio: Rapid Peer Feedback Emphasizes Revision and Improves Performance. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, pages 75–84, Vancouver BC Canada, March 2015. ACM. ISBN 978-1-4503-3411-2. doi: 10.1145/2724660.2724670.

[26] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. Pytest 7.2, 2004.

[27] OpenAI. GPT-4 Technical Report, 2023.