

From Understanding Learning Difficulties Among Students To Providing High-Quality Automated Feedback

by

Huanyi Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Huanyi Chen 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Austin Cory Bart
Associate Professor
Dept. of Engineering Computer & Information Sciences
University of Delaware

Supervisor(s): Paul A.S. Ward
Associate Professor
Dept. of Electrical and Computer Engineering
University of Waterloo

Internal Member: Wojciech Golab
Associate Professor
Dept. of Electrical and Computer Engineering
University of Waterloo

Internal Member: Derek Rayside
Associate Professor
Dept. of Electrical and Computer Engineering
University of Waterloo

Internal-External Member: Peter Wood
Assistant Dean of Online instruction and Lifelong Learning
Dept. of Mathematics
University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Students face various difficulties during their learning journeys. However, providing timely feedback often poses a challenge for educators due to availability constraints. Fortunately, automated feedback systems have been introduced, offering invaluable assistance.

To equip instructors with a general understanding of students in their teaching activities in computing education, we conducted an analysis of students' learning analytics to gain insights. In this study, we applied clustering techniques to behavior data naturally collected within an automated feedback system. We discovered that although students spent a significant amount of time using the system, the learning outcomes were often limited. A predictive model was derived based on these observations.

To assist students in their learning, we explored whether offering trivial-penalty time extensions could be beneficial and why students use them. Implementing flexible late policies was straightforward and placed minimal burden on instructors. We analyzed a fourth-year course that utilized flexible late policies and found that time conflicts and underestimation of coursework were the top two reasons for utilizing time extensions. In addition, our findings revealed a correlation between students' abilities and their usage of time extensions. This latter result was re-examined in a replication study and a reproduction study. While the automated feedback system was not initially considered in the main study, in the reproduction study, we found that even with time extensions and automated feedback systems, low/middle-performing students still could not match the performance of high-performing students. This suggests a fundamental issue: feedback from automated feedback systems may not be as effective as anticipated, which plays an essential role in assisting students' learning at scale.

Consequently, the critical question arises: how to provide effective feedback from automated feedback systems. We identified two main issues in current automated feedback systems: incorrect components marked as correct and correct components marked as incorrect. To address these issues, we argue that the unit testing philosophy, widely adopted in the software industry, should not be naively applied to automated feedback systems in an educational context. We completely redesigned the procedure and proposed a novel guideline for composing automated assessments. Following this guideline, we developed an automated assessment for an entity-relationship question in a database course. Our evaluation showed that students had significantly improved their understanding of the topic.

Acknowledgements

First and foremost, I would like to express my sincere appreciation to my supervisor, Prof. Paul A.S. Ward, whose guidance, unwavering support, and patience have been instrumental throughout my PhD journey. I have had the good fortune of working with him in courses, meetings, and various activities. His insightful sayings, imparted during countless discussions, have not only steered my research but have also profoundly influenced my personal growth and perspectives in life.

Furthermore, I would also like to thank my committee members. Prof. Wojciech Golab allowed me to assist in his courses over several terms, where I gathered numerous research ideas. Prof. Derek Rayside reminded me the importance of practical action over mere data analysis, which has deeply inspired my work in the field of automated feedback. Prof. Peter Wood opened my eyes to new possibilities for applying my research across various educational fields such as math. Last but not least, I sincerely appreciate Prof. Austin Cory Bart for serving as my external examiner and offering insightful comments on my thesis.

In addition to my committee members, I would also like to thank Prof. Jeff Zarnett, Prof. Patrick Lam, Prof. Scott Chen, and Prof. Yiqing Huang for offering me the opportunity to teach. I learned so much from those experiences. They are invaluable.

I am grateful to the Shoshin group for being a source of endless discussion and inspiration. Special thanks go to Prof. David Taylor and Prof. Bernard Wong for their valuable feedback on my research ideas, and to Liuyang Ren and Sharon Choy for sharing observations that helped shape my work.

Finally, I must express my profound appreciation for my family. My wife, Hua, has always been cheerful and supportive, and my daughter, Zitian, inspires me daily to be kind and motivated. I would also like to thank my parents, Hang and Feng, for their never-ending love and support throughout my life.

Dedication

This thesis is dedicated to my wife, Hua, and our daughter, Zitian.

Table of Contents

Examining Committee	ii
Author's Declaration	iii
Abstract	iv
Acknowledgements	v
Dedication	vi
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Problem Statement	3
1.2 Approach	3
1.3 Contributions	4
1.4 Organization	5
2 Background	6
2.1 Learning Analytics	7
2.2 Automated Assessments	9

2.2.1	Evaluation Techniques	10
2.2.2	Terminologies	11
2.2.3	Tests Organization	15
2.3	Automated Feedback	17
2.3.1	Expert-Authored Feedback	20
2.3.2	Data-Driven Feedback	22
2.3.3	Knowledge About Meta-cognition Feedback	23
2.4	Anecdotal Experience Using Marmoset	24
2.5	Chapter Summary	28
3	Learning Analytics Analysis	29
3.1	Related Work	30
3.2	Course Background	31
3.3	Experiment and Results	32
3.3.1	Clustering results from different clustering algorithms	33
3.3.2	Characteristics of students	35
3.3.3	Consistency in student behaviours	37
3.3.4	Exam grades of students	40
3.4	Chapter Summary	43
4	The Value of Time Extensions	45
4.1	Related Work	46
4.2	Course Background	47
4.3	The Main Study	51
4.4	The Replication Study	56
4.5	The Reproduction Study	59
4.6	Chapter Summary	61

5	Guideline for High-Quality Automated Assessments	62
5.1	Related Work	63
5.2	Problems	65
5.2.1	Incorrect components marked as correct	65
5.2.2	Correct components marked as incorrect	66
5.2.3	Improving existing assessments	67
5.3	Methods	68
5.3.1	Eliminating correct components being marked as incorrect	69
5.3.2	Adding new code safely	71
5.3.3	Providing high-quality feedback	73
5.4	Chapter Summary	77
6	Evaluation on the Novel Guideline	78
6.1	The Entity Relationship Modelling Question	78
6.1.1	The homework ER modeling question	79
6.1.2	ER encoding	79
6.1.3	The automated assessment	80
6.1.4	Automated feedback	84
6.2	Error rate of automated feedback	87
6.3	Feedback transitions of consecutive submissions	89
6.4	Final exam performance	90
6.5	Chapter Summary	92
7	Discussions and Future Work	94
7.1	Mistakes and Misconceptions	98
7.2	Contexts of Students	98
7.3	Artificial Intelligence (AI)	99
7.4	Evaluation Design	99
7.5	Chapter Summary	100

8	Conclusions	101
	References	103
	APPENDICES	118
A	Clustering Algorithms	119
B	Customized Marmoset	121
C	An Actual Assessment	123
C.1	Folder Structure of the Assessment	126
C.2	Assessment Code	130
C.2.1	Tests	130
C.2.2	Feedback	149
C.3	Quality Assurance on Assessment Code	161

List of Figures

2.1	Students receive feedback to correct misconceptions	7
2.2	Illustration of Data Granularity [62]	8
2.3	Example test report from GitLab	10
2.4	Components in Automated Platforms	14
2.5	Course and assessment in Marmoset	24
2.6	Instructor Views	26
2.7	Student Views	27
3.1	Assignment Characteristics	36
3.2	Coding Lab Characteristics	36
3.3	Box-plots of coding examination grades	41
3.4	Midterm grades and final exam grades including bonus midterm questions	42
3.5	Midterm grades and final exam grades including bonus midterm questions	42
4.1	Box-plots of grace day usage and assignment performance	53
4.2	Box-plots of grace day usage and final exam performance	54
4.3	Box-plots of grace day usage and assignment performance	57
4.4	Box-plots of grace day usage and final exam performance	58
5.1	Automated Assessments, KM Feedback, and KH Feedback	65
5.2	Relationships between test and assessment categories	68
5.3	Handling unexpected failures	70

6.1	Sample ER model and its encoding	81
6.2	Example feedback screenshot (partial)	83
6.3	Two example exemplary designs and two example student designs	84
6.4	Feedback category and the number of submissions	86
6.5	Feedback category and the number of submissions in the sample set	88
6.6	Error rate of provided automated feedback	88
6.7	Feedback categories of current submission and subsequent submission	89
6.8	Grade percentage (%) on SQL, Q1, and Q2 in the final exam	92
B.1	Customized Marmoset with a Help Link	121
B.2	An Piazza Post Example	122
C.1	ER solution A	124
C.2	ER solution B	125
C.3	Folder Structure (Collapsed)	126
C.4	Folder Structure (Expanding <code>assessment_tests</code>)	127
C.5	Folder Structure (Expanding <code>maps</code>)	128
C.6	Folder Structure (Expanding <code>tests</code> , <i>i.e.</i> , regression tests)	129

List of Tables

2.1	Feedback Categories [64]	18
3.1	Total number of tests for different assignments and coding labs pre-midterm	33
3.2	The size of different clusters in different clustering algorithms	34
3.3	The adjusted Rand index results	34
3.4	The adjusted Rand index results (high to low)	35
3.5	Assignments and coding labs before and after the midterm	37
3.6	Total number of tests for different assignments and coding labs post-midterm	38
3.7	The p values of the Welch's t-test results of assignments	38
3.8	The p values of the Welch's t-test results of coding labs	39
4.1	Student categorizes and sizes in the main study	51
4.2	Student categorizes and sizes in the replication study	56
4.3	First-year student categorizes and sizes	59
4.4	First-year student categorizes and sizes of each coding lab	60
4.5	Average final grades of students and corresponding t-test results	60
4.6	Average number of all passed tests and corresponding t-test results	60
4.7	The "extended" students who did not pass all <i>public</i> tests	61
5.1	Test outcomes	67
5.2	Differences in sorting results between SQL's <code>ORDER BY</code> and Python's <code>sorted()</code>	73
5.3	Test outcomes of probing tests and diagnosing tests considering dependencies	74

6.1	Feedback categories for the ER question	85
6.2	Average performance on the final exam, and p-value	91

Chapter 1

Introduction

The world has been greatly reshaped by the power of computing technologies [1]. It is so powerful that it has been integrated into every place of the world, which proposes the rapid growing demand to have a large number of well-educated computing experts. The field where people publish their studies and share their practices in teaching computing subjects is called *Computing Education*. The initial tertiary learning and university study in the discipline focused on Computer Science; however, the focus has greatly broadened since computing now includes many other areas, such as software engineering, information technology, informatics, data science and cyber-security, not to mention the enormous importance of recent developments in Artificial Intelligence and Machine Learning [36].

Although the number of learners majoring in computing has increased rapidly over the last two decades, the limited availability of teaching resources remains a challenge [115]. Educators have sought to develop various techniques to leverage such limitations while providing learners with high-quality learning outcomes. One crucial component is the technique to provide high-quality automated feedback.

The concept of automated feedback systems in computing education appeared back in 1960s [61, 49], where the feedback was whether student programs are correct or incorrect under a rigorous set of conditions. The procedure is mostly the same as today's software testing procedure [58, 40, 70], where the latter typically requires software developers to compose test cases to check the software meets requirements and specifications and that it fulfills its intended purpose. A test case is a specification of the inputs, execution conditions, testing procedure, and expected results, defining an executable test to achieve a particular software testing objective [4]. In the educational context, the software being verified and validated, is the student's program.

Automated feedback systems in early years are often so highly integrated that they serve a specific purpose to evaluate student programs within a specific course or for a specific assignment [61, 49]. These systems were primarily used by instructors, who were usually the assessment creators. However, people started to realize that functionalities such as submission management and code to guarantee secure assessment execution could be shared, therefore, automated feedback platforms appeared [98, 75, 43, 100]. By using automated feedback platforms, assessment creators can put more focus on the essential assessment logic development, leveraging the difficulties in using automated assessments in their courses. More recent, tools to support assessment logic creations also came up [72, 52].

Automated feedback is typically based on the outcomes of underlying test cases, where each outcome is binary—either a pass or a fail. Summing up the total points from these outcomes can be thought of as a grade for a submission and is probably the simplest form of feedback. Some automated feedback systems also provide more detailed information about the tests to assist students improve their solutions [43, 98]. For example, in addition to the total points, test outcomes of individual tests can be revealed. Further, if a test failed due to a pre-coded reason, the reason can be revealed. In this case, students not only know the test is failed, but they also know why it fails.

Automated feedback systems bring numerous benefits to both instructors and students in computing education. They enable students to develop effective self-learning strategies. Additionally, these systems allow educators to automatically collect a tremendous amount of learning analytics, the importance of which will be soon described.

Learning analytics play a crucial role in computing education across many aspects. They not only provide educators with insights into their teaching processes but also offer essential metrics to understand learners. Treating all learners the same is never a good approach; different learners struggle with different issues and require tailored assistance based on their unique contexts. The interpretation of this data and its potential uses are important components in computing education studies and of great interest to many researchers.

Ihantola et al. [62] conducted a literature review so as to form an overview of the body of knowledge regarding the use of educational data mining and learning analytics focused on the teaching and learning of programming. However, they also pointed out that verifying the findings of existing studies is challenging and it is often failure prone. Ihantola et al. [62]’s finding was further confirmed by Hao et al. [56], that only 58% of the computing education studies can be replicated when all authors of a study were different from the authors of the study they were attempting to replicate.

With an investigation of learning analytics of students enrolled in a programming

course, we discovered that understanding students is not sufficient; knowing when and how to intervene in students' learning is also crucial. The order of the studies presented in this thesis generally represents the roadmap that we have taken to explore the field of computing education in the context of university learning in Canada.

1.1 Problem Statement

The primary problem we are trying to address in this thesis is: **How to improve computing education so that every student can achieve the best learning outcome?**

This central problem, while critically important, lacks clarity and does not readily suggest actionable solutions. To manage this complexity, we have decomposed it into three distinct but interconnected sub-problems, which we examine in a structured sequence.

The sub-problems explored in the thesis are summarized as follows:

1. How to understand students' learning based on the learning analytics data?
2. What can be a reliable indicator that pinpoint students in need of immediate academic support?
3. How can the design of automated assessments be enhanced to effectively improve students' learning experiences and outcomes?

More concretely, the first sub-problem aims to equip instructors with a general understanding of students in their teaching activities. The second sub-problem focused on identifying a reliable point in time at which instructors can intervene proactively, preventing potential academic failures. Lastly, the third sub-problem sought to develop concrete and effective strategies for such interventions automatically. Inherently, each problem progresses logically from the previous, adding generalizability to the solutions proposed.

1.2 Approach

For the first problem, we analyzed students' pre-midterm behaviour data with an automated feedback platform, Marmoset, in an introduction-level programming course in an R1 university in Canada. Because various studies has found a strong correlation between

the midterm exam grades and the final exam grades, which is also true for our data (Pearson correlation is 0.81), therefore, we deliberately picked pre-midterm data for analysis. We applied clustering techniques to the data to find out student clusters and understand the characteristics of students in different clusters. We also investigated how students' performances connect to the clustering results.

For the second problem, we aimed to collect data from three undergraduate programming courses where instructors granted students time extensions. One course was a first-year programming course, while the other two were final-year programming courses. In the main study, which involved a final-year distributed computing course, we provided students with penalty-free grace days. We then collected data on students' usage of these grace days and their reasons for using time extensions through emails. We conducted a thematic analysis to identify the common reasons why students utilized time extensions. Additionally, we investigated whether there was a correlation between the use of time extensions and students' performance. This analysis showed that using or not using time extensions is an effective indicator. This latter result was re-examined under different contexts in the other two courses.

For the third problem, we identified drawbacks in the current automated feedback provision process and developed a novel guideline that enables instructors to compose automated assessments capable of providing accurate feedback. A key consideration within this guideline is that whenever an automated assessment determines pre-coded automated feedback cannot be provided, it turns to human assistance. We evaluated this guideline using an Entity-Relationship (ER) question in a database course and evaluated the effectiveness of the corresponding automated feedback.

1.3 Contributions

The thesis makes following contributions:

- It reveals that students exhibit different learning behaviors when engaging with automated feedback systems. Some of the content was previously published in Chen and Ward [28].
- It presents that whether or not time extensions are used is an effective indicator for identifying students who need assistance, along with the most common reasons why students use time extensions. However, it also reveals that simply granting time

extensions may not be sufficient for students to achieve optimal learning outcomes. Some of the content was previously published in Chen and Ward [33].

- It demonstrates a novel and effective guideline which greatly diverges from the existing build-and-test philosophy for composing automated assessment. A concrete implementation is also presented. Some of the content was previously published in Chen and Ward [29, 32, 27].

1.4 Organization

The rest of this thesis is organized as follows: Chapter 2 provides the background that is essential for understanding the remainder of the thesis. Chapter 3 describes the study to reveal different categories of students using learning analytics collected from an automated feedback system. Chapter 4 describes a study to understand the value of time extensions, with partial results re-examined in a replication study and a reproduction study. Chapter 5 describes the novel and effective guideline for composing high-quality automated assessments, and Chapter 6 provides a concrete implementation following the guideline and its evaluation. Chapter 7 provides further discussion and outlines future research directions. Finally, Chapter 8 summarizes the thesis.

Chapter 2

Background

Learning is a complex process that occurs internally. Students achieve the learning objectives by attending lectures, watching pre-recorded videos, and/or reading lecture notes. When preparing these materials, instructors expect them to cover the essential learning objectives of the course.

Equally important, instructors need to track the learning progress of students. For in-person classes, the classroom serves as a valuable setting to gauge student confusion. Instructors can monitor learning progress through interactions with students during class. This method does not require additional effort from either instructors or students; however, it only tracks the progress of the class as a whole, not individual students. Office hours offer a way to monitor the learning progress of individual students through one-on-one interactions between instructors and students. However, this method requires both instructors and students to coordinate their schedules to find a mutually convenient time for meetings.

Assessments, on the other hand, do not require synchronization of schedules. A well-designed assignment can reveal students' misconceptions through their solutions, as shown in Figure 2.1. By analyzing students' answers, which reflect their understanding, instructors can determine if a student has learned effectively or needs additional help. However, this approach requires instructors to invest a significant amount of time in manually assessing students' work. Consequently, there is a desire to replace the role of instructors in the feedback loop with automated feedback systems, which can also automatically collect rich learning analytic data.

This chapter provides background on how researchers utilize learning analytics and the current procedures for composing automated feedback. It is based on existing literature

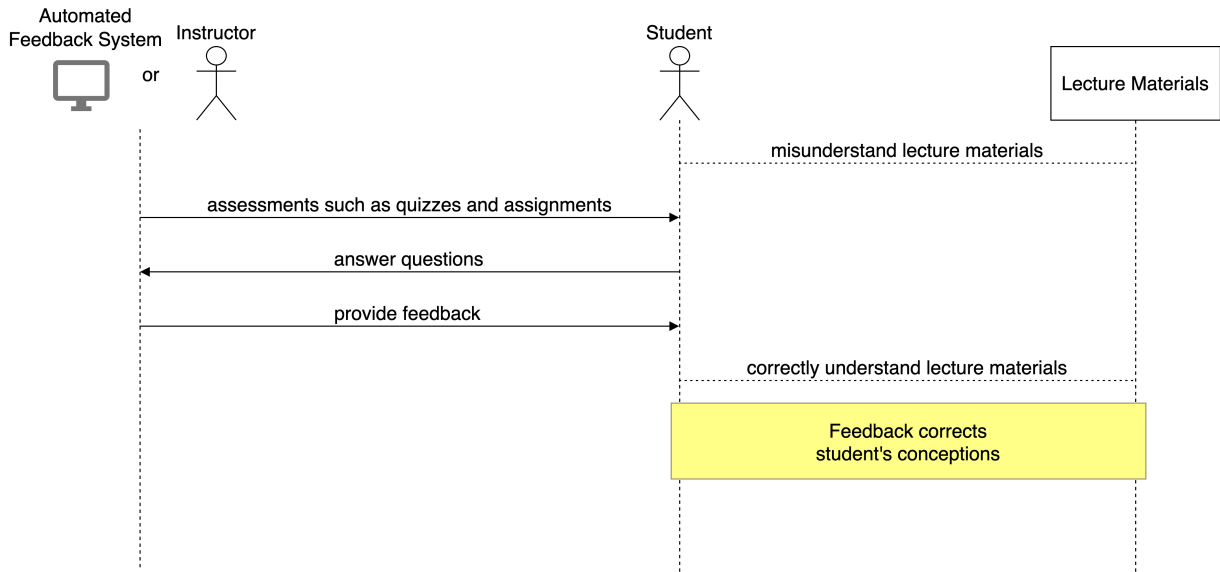


Figure 2.1: Students receive feedback to correct misconceptions

reviews and anecdotal observations.

2.1 Learning Analytics

Educational data mining and learning analytics offer insights into student behavior and knowledge, helping to uncover hidden factors that influence student actions. The information revealed can guide decisions in course design, tool selection, and teaching methods. Additionally, it can help engage students more effectively and support those who may be at risk of failing. Ihantola et al. [62] conducted a literature review to provide an overview of the body of knowledge regarding the use of educational data mining and learning analytics in the teaching and learning of programming.

Student interaction data are typically collected in the context where programming tasks are assessable and where automated feedback is provided. The granularity of these interactions within programming environments can be put into several categories: *key strokes*, *line-level edits*, *file saves*, *compilations*, *executions*, and *submissions*, as shown in Figure 2.2. The finest granularity involves individual key strokes, whereas the coarsest is complete assignment submissions.

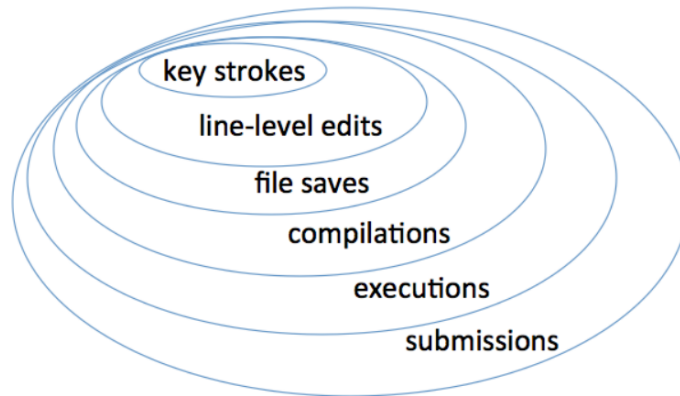


Figure 2.2: Illustration of Data Granularity [62]

The research objectives and motivations for analyzing learning analytics data can be broadly classified into three categories: *student*, *programming*, and *learning environment*. The *student* category is to predicting student performance, assessing student affect, and estimating student knowledge. The *programming* category focuses on identifying programming behaviors and strategies, as well as pinpointing errors within programs. The *learning environment* category aims to understanding students' usage and enhancement of tools and mechanisms associated with automated testing, grading, and feedback.

The methodological approaches in learning analytics span a spectrum from basic statistical methods, which primarily report counts and percentages, to more sophisticated statistical analyses, including inferential statistics, Bayesian methods, and t-tests. Additionally, complex exploratory statistical analyses, such as correlation, regression, and factor analysis, are employed. Interpretative classification and interpretive qualitative analysis have also been notable. With the emergence of Educational Data Mining (EDM) as a discipline [2], machine learning techniques such as clustering have increasingly become popular [39].

In addition to identifying various categories from the data, Ihantola et al. [62] also conducted several case studies to verify the findings of others. Their case studies were at three levels:

Re-analysis: In re-analysis, data from a previous experiment is used to verify the results, *i.e.*, there is no change in the procedure. Re-analysis can be used to confirm that no errors were made during the data analysis phase and that similar findings can be obtained using the same data as in a previous experiment.

Replication: In replication, the method from a previous experiment is followed to verify

the results. Replication can be used to confirm that the observed findings can be discovered more than once using the same method. For example, in software-related studies, the experimenter might vary the subjects (students) or the software artifacts (tools, systems).

Reproduction: In reproduction, the hypotheses from a previous experiment are tested to verify that the findings are independent of the experimental method used. This ensures that testing the hypothesis is not dependent on any particular procedure, materials, or instruments used in the experiment.

Replication, *i.e.*, changing subjects and/or used tools, and reproduction, *i.e.*, changing the data analysis approaches and methods but seeking the same phenomenon, as well as their combination, were the most common ways of verifying previously observed findings in the field of computing education.

Reproducing many well-known psychological studies in new contexts is difficult [34], as are learning analytic studies. Ihantola et al. [62] failed both replication and reproduction case studies, and their re-analysis results were only partially successful.

A literature review conducted by Hao et al. [56] noted that when a replication study was conducted by the same authors, with no replications reporting mixed results (*i.e.*, partial success), the success rate was 72%, and the failure rate was 22%. In contrast, the success rate dropped to 58% and the failure rate increased to 28% when all authors of a study differed from the authors of the study they were attempting to replicate.

Such failures in these replication case studies, where previously published analysis methods were applied to different datasets, suggest that context-specificity may lead to unfounded or erroneous conclusions. Overall, when replication studies cannot be easily conducted, more studies and observations of learning analytics remain necessary. Additionally, this suggests that researchers should conduct studies to better understand students' learning within their own contexts.

2.2 Automated Assessments

In software testing, test cases are used to verify that a software application operates according to its specified requirements. In the simplest form, a test case consists of a specific input paired with its anticipated output. Successful execution of a test case—where the software yields the expected output under a set of predetermined conditions and following designated testing rules—confirms that the software meets a given requirement. Figure 2.3 shows an example of a test report¹.

¹https://docs.gitlab.com/ee/ci/testing/unit_test_reports.html, accessed: 2024-Aug-14

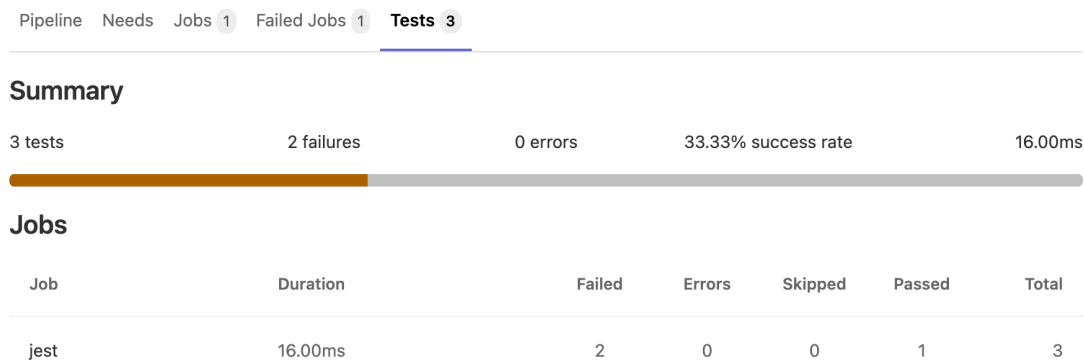


Figure 2.3: Example test report from GitLab

Because assessing students’ programs is similar to testing software, the most common strategy used in many existing automated assessments is to employ test cases. This approach allows any flaws or mistakes in the students’ programs to be detected through the failures of corresponding tests².

2.2.1 Evaluation Techniques

Broadly, evaluation techniques used in current automated assessments fall into four categories.

Output Comparison is an approach for testing whether a program meets specified requirements. It involves running the entire program as a black box with predefined input and comparing its output against expected results.

Unit Testing is extensively used in both industry and education to check the functionality of programs. It often relies on language-specific xUnit frameworks, such as JUnit for Java, CUnit for C, PyUnit for Python, and HUnit for Haskell. Unit testing offers a more detailed level of evaluation compared to the output comparison method, as it allows for the assessment of fine-grained components such as functions. A function is marked correct if its execution result matches the expected output for predefined input.

Output comparison and unit testing are often considered dynamic analysis, and programs are assessed in a black-box manner. However, purely relying on dynamic analysis

²A test is often a concrete implementation of a test case, but this thesis uses “test” and “test case” interchangeably for convenience.

is not ideal in many cases, especially when students' programs or functions are partially correct. Therefore, static analysis, which takes into account students' source code, has been gaining increasing attention in automated assessments.

One static analysis approach is *Structural Analysis*. It involves parsing a student's source code to create an abstract syntax tree, which is then converted into a graph representation. This graph is compared against a collection of model graph representations using a graph similarity measure. The model graph that most closely resembles the student's graph is identified, and any differences between them are calculated to assess the program.

Symbolic Execution is another static analysis approach that aims to identify *all* possible inputs that cause a test to fail [14]. In symbolic execution, program execution is carried out by a *symbolic execution engine*, which allows input values to be represented as symbols. Constraint solvers such as KLEE [25] are then used to construct actual instances to approximate the behavioral similarity between the reference program and student programs.

Recent research has investigated the effectiveness of Large Language Models (LLMs) in automated assessments. These studies reveal that even advanced models struggle with symbolic reasoning and program execution, which are essential for understanding the underlying bugs and possible student misconceptions, leading to the necessity for carefully designed prompts or prompting strategies [87]. Additionally, the inherent non-determinism of these models complicates the process of configuring them to reliably identify student misconceptions in a consistent manner.

The aforementioned techniques can be used independently or combined in actual automated assessments. For example, Vujošević-Janičić et al. [107] used output comparison, structural analysis, and symbolic execution in their grading framework. Ultimately, these assessment techniques are typically deployed in the form of various tests, and students are presented with the results of each test in hopes that they can identify their own misunderstandings.

2.2.2 Terminologies

Technically, instructors, teaching assistants, and assessment creators can be different individuals. However, it is common for people not to distinguish them, as is the case in this thesis.

Similarly, people often do not differentiate between *Tool*, *Platform*, and *System*. However, this thesis defines the following terminologies to clarify the implicit differences.

- *Tool*: A tool aids assessment creators in creating tests or provides pre-composed tests for automated assessments. It can be used by and is likely to be shared among assessment creators.
- *Platform*: A platform is a space that offers functionalities for assessment creators to host their automated assessments. It often includes features not directly related to automated assessments, such as submission management, course management, and student enrollment management. While it does not provide or only minimally assists in composing tests for automated assessments, it may offer functionalities for visualizing assessment metrics, such as code coverage statistics. It also defines rules, such as input and output formats, for automated assessments.
- *System*: A system is similar to a platform, except that automated assessments are not just hosted on it but are an integral part of it. As such, it is unlikely for a system to be used by other instructors if the automated assessment needs to be altered. Systems are often developed within a specific course context, which is only relevant to the system developers, and tend to lack robustness. By defining communication rules that decouple automated assessments from other parts of the system, a system can potentially be transformed into a platform.
- *Human Support Tool*: Not all tools, platforms, or systems developed are used for automated assessments. Some are designed to assist with manual assessments. If any such tool is mentioned in this thesis, it will be referred to as a human support tool.

Here are a few examples to demonstrate the differences.

Pedal³ [52] internally provides a collection of modules for assessment creators to evaluate students’ work. To use Pedal, assessment creators must compose *Instructor Control Scripts*—the logic of automated assessments—to orchestrate the execution of these modules. Feedback messages generated by each module are reported to a shared location. Finally, a resolver module selects feedback messages, considering their priorities, to be presented to students. The authors of Pedal refer to it differently in various contexts. For instance, they call it an “Infrastructure” or “Library” in the paper introducing it [52], but also a “Framework” in Pedal’s documentation⁴. Given that Pedal focuses on creating the logic for automated assessments, it is therefore categorized as a tool according to the thesis’s terminology.

³<https://github.com/pedal-edu/pedal>, accessed: 2024-Aug-14

⁴<https://pedal-edu.github.io/pedal/>, accessed: 2024-Aug-14

Several automated grading systems should be categorized as platforms using the thesis’s terminology. For example, Marmoset [98], developed around the year 2006 [97] but still used in several universities today, is often referred to as a *system* in various sources [97, 109, 82]. Upon receiving a new student submission, it runs a series of predefined tests designed by course instructors to check correctness. Instructors are required to provide a `test.properties`⁵ file, where they specify the test names, timeouts for each test, and so on. The format of this test properties file adheres to the `load()` and `store()` methods in `java.util.Properties`. Each test outcome is either *pass* or *fail*. Standard outputs and errors are gathered as feedback messages⁶. Since Marmoset does not offer any assistance for assessment creation beyond specifying rules for input (the `test.properties` file) and output (standard output and error) for the automated assessments, it should be categorized as a platform according to the thesis’s terminology. Another example is Web-CAT, initially referred to as a tool [41] but later as a system [18]. Web-CAT has many similar functionalities as Marmoset but emphasizes more on Test-Driven Development (TDD), requiring students to write their own tests to achieve high code coverage. Although it provides plugins and configuration options for processing Java or C++ assignments and handling student-written tests, it does not offer much assistance for assessment creators to compose actual tests. Therefore, Web-CAT should also be categorized as a platform according to the thesis’s terminology.

The predefined rules for automated assessments to retrieve student submissions and communicate the assessment results differ greatly from platform to platform. Unlike the aforementioned Marmoset [98] and Web-CAT [43], MarkUs [75] allows for various assessment setup mechanisms but requires that the outcomes be communicated in JSON format⁷. Submitty [5] utilizes a `config.json` file⁸, which contains both the setup for tests and the precise commands to be executed. However, there are seldom restrictions for assessment creators on what can or cannot be assessed by their automated assessments. Assessment creators have sufficient flexibility to compose tests in any manner they like, as long as the composed assessments can communicate with the automated platform following the predefined rules.

An automated platform is often composed of the following components related to automated assessments hosted on it: *student interface*, *instructor interface*, *automated assessments*, and *assessment executor*, as illustrated in Figure 2.4.

⁵<https://marmoset.cs.umd.edu/umd101/p0/>, accessed: 2024-Aug-14

⁶<https://cs.uwaterloo.ca/twiki/view/ISG/Marmoset>, accessed: 2024-Aug-14

⁷<https://github.com/MarkUsProject/markus-autotesting#the-custom-tester>, accessed: 2024-Aug-14

⁸<https://submitty.org/instructor/autograding/structure>, accessed: 2024-Aug-14

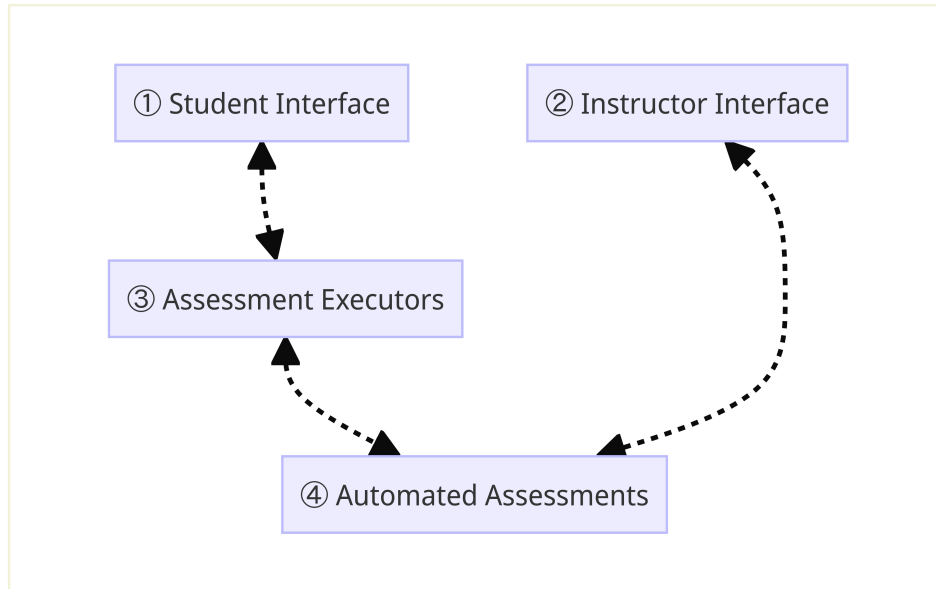


Figure 2.4: Components in Automated Platforms

The student interface can be understood as a web interface where students upload submissions and receive feedback. The instructor interface commonly includes features for instructors to set up and update automated assessments. The assessment executor is where automated feedback platforms execute automated assessments; however, instructors typically do not control this directly. Instead, automated platforms will read configurations from the automated assessments to manage them, such as the `test.properties` file in Marmoset.

Most automated tools or systems developed in the early stages [61, 49] should be categorized in the system category, as they primarily focused on grading specific assignments or assignments within a specific course. More recent, SQL Tester, which was referred to as an online SQL assessment tool in a paper [66], provides automated assessments for SQL queries by comparing query outputs against expected outputs. However, it also offers integrated functionalities such as student management, and displaying question content, table schema, and desired output alongside the output from the student’s last attempt. Since it focuses on assessing SQL questions and there is no clear way to alter its automated assessments, it should be categorized as a system according to the thesis’s terminology.

Head et al. [57] introduced a mixed-initiative approach allowing instructors to combine their deep domain knowledge with the results of data-driven program synthesis techniques,

and demonstrated and evaluated this approach in two novel systems, MISTAKEBROWSER and FIXPROPAGATOR. Both systems rely on a data-driven program synthesis technique that learns code transformations from examples of bug fixes. The transformations are used to: cluster incorrect submissions based on the transformation that corrects them; generate bug fixes for each incorrect submission so the fixed submission can pass all tests; and propagate instructor-written feedback to all incorrect submissions that are fixed by the same transformation. Given that MISTAKEBROWSER and FIXPROPAGATOR are so specialized and coupled with their user interfaces, they fall into the system category according to the thesis’s terminology.

Pensieve [114] serves as an example of human support tools. Rather than embracing the trend of automated assessments, it aims to “push back” against it in classrooms by thoughtfully integrating a human grader into the assignment feedback process. As a student works on an assignment, their process is recorded as a local repository, which is submitted along with their final answer. Pensieve provides a graphical user interface (GUI) for the human grader to interact with the four main components of the tool: a timeline of snapshots for the program file, a text view of the snapshot code, the visual output corresponding to running the snapshot code, and metrics such as lines of code and the number of runs over time. Although Pensieve may possess a complex architecture on its own, it is categorized as a human support tool since it has no relationship with automated assessments in any form.

2.2.3 Tests Organization

Tests are essential for automated assessments. The organization of tests is highly flexible in automated tools. For example, Pedal [52] does not assume dependencies among tests. Every test specified in the instructor control script is executed; however, the results of tests will be prioritized so that only their corresponding feedback messages are delivered. In contrast, PyTA⁹ [72] stops at the first violation¹⁰, indicating a dependency graph where a later test is executed only if the previous test passes.

The organization of tests commonly follows a convention in automated platforms, where there is a build stage that ensures a student’s code is compilable, and a test stage that executes the tests of automated assessments. The test stage depends on the build stage to succeed. If a student’s code fails to compile during the build stage, then no tests are run. Students are then presented with an error message detailing the compilation issues.

⁹<https://github.com/pyta-uoft/pyta>, accessed: 2024-Aug-14

¹⁰<https://www.cs.toronto.edu/~david/pyta/contracts/index.html>, accessed: 2024-Aug-14

The compilation process is often integrated into platforms so that assessment creators only have to focus on writing tests for the test stage. Once the build stage succeeds, tests in the test stage are executed as if no dependencies among them exist. Note that it is possible for assessment creators to create dependent tests by hacking into platforms. However, assuming tests without dependencies is the most common mindset adopted by assessment creators and platform developers.

Manual assessments are sometimes needed, such as checking the overall design of the solution and/or code readability. Some platforms incorporate mechanisms for both automated and manual evaluation of student work. For example, Gradescope [3] facilitates the integration of auto-grading and manual grading for programming assignments, as detailed in their documentation¹¹. In this context, a programming assignment is decomposed into segments that are either suitable for automated grading or necessitate manual assessment. The final grade is computed by aggregating the scores from these two segments.

The process of submitting code, executing tests, and then presenting the test outcomes is essentially akin to the conventional software quality assurance methodology employed in continuous integration systems. Software engineers who are developing the software, when seeing failed tests, can investigate these tests and thus determine which part of the software is flawed. However, despite the similarities, notable distinctions exist between the industrial and educational settings:

- *Complexity*: Software quality evaluation in the industry tends to be much more thorough and complete than assessments in educational settings. For example, *regression testing* is a common approach to finding defects after a major code change has occurred in software; however, it is rarely discussed on assessment code in education.
- *Visibility and Access*: Software developers usually have access to the internal details of tests; students, however, typically do not have such privileges due to the risk of exposing assessment details for repeated use.
- *Expertise Level*: Software developers typically possess a deep understanding of programming and debugging techniques, contrasting with students, who are often novice learners acquiring these skills.
- *Assessment and Grading*: For students, these tests often serve a dual purpose—not only do they provide feedback on code functionality but also contribute directly to course grades.

¹¹<https://gradescope-autograders.readthedocs.io/en/latest/>, accessed: 2024-Aug-14

The difference in complexity indicates that assessment creators tend to think of creating automated assessments as a one-time process and bug-free, hence no regression tests are needed. The difference in visibility and access leads to a consequence that some students may find it very difficult to interpret test results other than knowing whether a test is passed or failed. Even though they somehow understand their code was flawed, due to the difference in their expertise levels, students may find it hard to resolve such failures. Worse, students may become depressed and feel frustrated when they see an abundance of failed tests [23], especially when the test results are connected to their course grades, which is often the case.

In sum, this section describes the current state of the tests within automated assessments. The next section will discuss the feedback produced and delivered to students from these automated assessments.

2.3 Automated Feedback

The simplest form of feedback in automated assessments is binary feedback—messages that indicate either “correct” or “incorrect” for tests. However, it has been found that instant binary feedback can have harmful effects on student behavior [69]. Therefore, current automated feedback often has multiple forms. Narciss [79] conducted a literature review and provided the initial categories of automated feedback in year 2008. Ten years later, Keuning et al. [64] provided a literature review that extended the feedback categories from Narciss [79]’s work by adding representative subcategories. Specifically, they focused on formative feedback for a single exercise. A summary of the feedback categories from Keuning et al. [64] is shown in Table 2.1.

Table 2.1: Feedback Categories [64]

Category/Subcategory	Description
Knowledge About Task Constraints	
Hints on task requirements	Requirements such as using a specific language construct or avoiding certain library methods
Hints on task-processing rules	General hints on how to approach the task, independent of the student’s current work
Knowledge About Concepts	
Explanations on subject matter	Generated while a student is working on an exercise, such as relevant internet sources
Examples illustrating concepts	Examples that clarify concepts
Knowledge About Mistakes	
	Each of these subcategories of feedback has a level of detail. The level of detail can be <i>basic</i> , which can be a numerical value (total number of mistakes, grade, percentage), a location (line number, code fragment), or a short type identifier such as “compiler error”; or <i>detailed</i> , which is a description of the mistake, possibly combined with some basic elements
Test failures	Indicates that a program does not produce the expected output
Compiler errors	Syntactic or semantic errors detected by a compiler and are not specific for an exercise

Continuation of Table 2.1

Category/Subcategory	Description
Solution errors	Solution errors can be found in programs that do not show the behaviour that a particular exercise requires, and can be runtime errors (the program crashes because of an invalid operation) or logic errors (the program does not do what is required), or the program uses an alternative algorithm that is not accepted. A solution error feedback can be, the results of matching the student program with several model programs, comparing aspects such as size, structure, and statements
Style issues	Concerns formatting, documentation, or structural programming issues
Performance issues	Program takes too long to run or uses more resources than required
Knowledge About How to Proceed	Each of these subcategories of feedback has a level of detail: a <i>hint</i> that may be in the form of a suggestion, a question, or an example; a <i>solution</i> that directly shows what needs to be done to correct an error or to execute the next step; or both hints and solutions
Bug-related hints for error correction	Feedback clearly focuses on what the student should do to correct a mistake
Task-processing steps	Information about the next step a student has to take to come closer to a solution
Improvements	Hints on improving the solution, such as structure, style, or performance of solutions

Continuation of Table 2.1

Category/Subcategory	Description
Knowledge About Meta-cognition	Meta-cognition deals with a student knowing which strategy to use to solve a problem, if the student is aware of their progress on a task, and if the student knows how well the task was executed. According to Narciss [79], this type of feedback could contain “explanations on metacognitive strategies” or “metacognitive guiding questions.”

In their literature review, Keuning et al. [64] found that knowledge about task constraints and knowledge about concepts only exists to a small extent. Knowledge about mistakes was the largest type and it was found in almost all places. They also found knowledge about how to proceed appeared about half of the time, of which error correction was the largest subcategory.

2.3.1 Expert-Authored Feedback

Most of the time, automated feedback are pre-coded by experts, *e.g.*, assessment creators. This thesis refers such feedback as *expert-authored feedback*. Expert-authored feedback is from direct or customized test results of automated assessments.

An automated assessment tool can include pre-coded feedback by tool developers so that assessment creators who adopt the tool in their automated assessments can make use of those feedback as is. The following PyTA [72] example shown in Listing 1 shows that if the precondition $y \neq 0$ is not satisfied, a knowledge about mistakes feedback will be generated. However, some tools also allow instructors to customize the feedback to certain extent. For example, Pedal [52] allow customized feedback putting into its instructor control script as shown in Listing 2.

An automated platform often is designed to selectively present details from test results, configured by assessment creators. Even when information from all tests is available, the platform may choose to limit the details to only certain tests. By reducing the amount of information presented, students can more easily identify and concentrate on resolving the main issue. For example, Marmoset [98] displays not only the “passed” or “failed” test outcomes, it also displays contextual details of tests, which are often enriched by the

```

1 # demo.py
2 def divide(x: int, y: int) -> int:
3     """Return x // y.
4
5     Preconditions:
6     - y != 0
7     """
8     return x // y
9
10
11 if __name__ == '__main__':
12     from python_ta.contracts import check_all_contracts
13     check_all_contracts()
14
15     divide(10, 0) # Preconditions on y violated

```

```

1 Traceback (most recent call last):
2   File "./pyta/demo.py", line 15, in <module>
3     divide(10, 0) # Preconditions on y violated
4     ~~~~~
5   File "./pyta/.../_init_.py", line 96, in _enable_function_contracts
6     raise AssertionError(str(e)) from None
7 AssertionError: divide precondition "y != 0" was violated for arguments {x: 10, y: 0}

```

Listing 1: PyTA Demo

```

1 from pedal import *
2
3 # Common mistake is that students put a $ in their code
4 if "$" in get_program():
5     explain("You should not use the dollar sign ($) anywhere in your code!",
6            title="Do Not Use Dollar Sign")
7            priority='syntax', label="used_dollar_sign")

```

Listing 2: A customized feedback in the instructor control script of Pedal

assessment creators. In the event of an equality assertion failure, beyond merely showing the mismatched values on the left-hand side (LHS) and right-hand side (RHS), assessment creators can also choose to include the input value that triggered the test to fail in the output or even re-write the output so that it is easier to be interpreted by students. In addition, Marmoset [98] has three categories of tests with different disclosure policies. Students can see full details on one category; however, to receive feedback from the remaining categories, students must demonstrate sufficient progress measured by tests in the aforementioned category. Even then, only limited information is revealed such as only the names of tests are shown. This approach was also adopted by Lee [70] in his system.

For another example, Web-CAT combines duplicate feedback and sorts them so that precedence is given to feedback with more occurrences. Meanwhile, it limits the results of only top few tests to discourage students from using the assessment platform as their sole means of testing their own work [23, 40]. Submitty [5], in addition to build phase and test phase, adds a third phase called validation phase, which allow assessment creators to compose rules to determine what and how to disclose test results.

An automated system can function similarly to automated platforms regarding test results. For example, FIXPROPAGATOR [57] uses the number of passed tests to verify if an automated code fix that is going to be propagated leads to an improved solution. The test results are presented in a way similar to automated platforms, where it shows the actual result, the expected result, and console output for each test.

Expert-authored feedback assumes that the tests within automated assessments can accurately capture the underlying issues of students' programs. Once the underlying issue is identified, the feedback can be categorized differently. For example, in a question about counting the number of elements in the input array, a knowledge about mistakes feedback, such as "*expected: 5 numbers, actual: 6 numbers,*" can be provided when an off-by-one error is observed; alternatively, a knowledge about how to proceed feedback, such as "*It seems you have an off-by-one error, please check if you iterated one more time through the input array,*" can be provided.

There is no clear guidance on how to map test results to expert-authored feedback. In other words, expert-authored feedback is composed based on assessment creators' knowledge and experience. For example, a knowledge about concepts feedback may sometimes be helpful when paired with certain knowledge about mistakes feedback or knowledge about how to proceed feedback. The above feedback could be further enhanced as, "*It seems you have an off-by-one error, please check if you iterated one more time through the input array. You can find the Loop concept in slide 15.*"

2.3.2 Data-Driven Feedback

By leveraging existing students' solutions [37], *data-driven feedback* can be generated. This type of feedback typically falls into the knowledge about how to proceed category, focusing on providing hints that suggest one-step edits to improve the solution—often in terms of the number of passed tests [82, 89, 92]. The methods include clustering (grouping similar items), filtering (selecting relevant items), and pattern mining (discovering recurring patterns). Such approaches typically rely on a large volume of student submissions and aim to provide students with suggestions for repairing their code by measuring the distance

between exemplary solutions and individual students’ submissions. However, they tend to not identify the underlying misconceptions. For instance, ITAP [92] (Intelligent Teaching Assistant for Programming) is a data-driven tutor for programming in Python. It creates a solution space graph with (intermediate) program states as nodes, where directed edges represent next steps. ITAP matches a student’s solution to a state in the graph, constructs a path to a correct solution, and generates hints based on the steps of that path.

Data-driven feedback is particularly valuable in situations where assessing student work using predefined rules is not straightforward or when fine-grained evaluation is needed. For example, the smallest unit that can be dynamically tested in automated assessments is often a whole function. Data-driven feedback, however, can provide more detailed insights, such as specific lines of code or even individual values and/or literals within those lines.

2.3.3 Knowledge About Meta-cognition Feedback

As indicated in Table 2.1, metacognition involves a student’s knowledge of which strategy to use to solve a problem, their awareness of their progress on a task, and their understanding of how well the task was executed. Metacognitive skill is essential for self-regulated learners to effectively guide their own learning process [31]. Keuning et al. [64] identified only one example—HABIPRO—that provides knowledge about meta-cognition feedback. The HABIPRO automated system features a “simulated student” that responds to situations such as students repeatedly submitting the same incorrect solutions by explaining why the solution cannot work, or checking if a student truly understands why an answer is correct when a correct solution is submitted.

Providing knowledge about meta-cognition feedback often requires taking into account students’ historical submissions, which is not an easy task for automated assessments. Large Language Models (LLMs) may alleviate this difficulty to some extent; however, their effectiveness in this context is not yet well-established. Human support tools such as Pensieve [114] can be utilized effectively in this scenario. Pensieve was used in an assignment that required students to create a pyramid-like object using Java. With Pensieve, students can automatically save hundreds of their intermediate steps into a repository. This repository, along with the final submission, can be voluntarily submitted by students for a thorough review by teaching assistants. Upon reviewing the submitted materials, such as code changes and intermediate drawings, teaching assistants can offer in-person feedback not only on the final answer but also on problem-solving and learning strategies, thereby enhancing students’ metacognitive skills.

2.4 Anecdotal Experience Using Marmoset

In the studies conducted for this thesis, the primary underlying automated feedback platform utilized was Marmoset, developed by Spacco et al. [98]. Marmoset [98] is a platform designed to automatically evaluate and grade programming assignment submissions, allowing instructors to efficiently configure courses and their corresponding assessments.

Within Marmoset, an assessment is equivalent to a *project*. Instructors configuring assessments may allow multiple assessments to share the same deadline, especially when they represent distinct parts of a single assignment. Illustrations include the course listing interface, as depicted in Figure 2.5a, and an example of assessments shown in Figure 2.5b.

Courses

- [ECE 150:001 \(Fall 2017\): Submission Site for ECE 150, Section 001](#)
- [ECE 150 Practice \(Fall 2017\): Practice Questions for ECE 150](#)
- [ECE 150 \(Fall 2018\):](#)

(a) Courses in Marmoset

Projects

Project	Overview	testing setup	# to test	# retesting	Visible	Due	Title
P9-CartesianPoint	view	active	0	0	true	13 Dec, 01:00 PM	Point Class
P8-FileExt	view	active	0	0	true	13 Dec, 01:00 PM	Checking File Extensions
P7-Lights	view	active	0	0	true	13 Dec, 01:00 PM	Lights Control
P6-Sequence	view	active	0	0	true	13 Dec, 01:00 PM	Checking Arithmetic Sequences
P5-FindFirst	view	active	0	0	true	13 Dec, 01:00 PM	Find First Uppercase or Digit
P4-Digits	view	active	0	0	true	13 Dec, 01:00 PM	Rearranging Digits
P3-LCM	view	active	0	0	true	13 Dec, 01:00 PM	Lowest Common Multiple

(b) Automated assessments in Marmoset

Figure 2.5: Course and assessment in Marmoset

Each assessment within the platform requires a set of tests, which are categorized into three types: *public* tests, *release* tests, and *secret* tests. The key distinction among these tests lies in their visibility to students; students can view the outcomes of all public tests for any submission. However, the outcomes of release and secret tests are not accessible and come with restrictions.

Figure 2.6 and Figure 2.7 are based on the `test.properties` as shown in Listing 3.

From the perspective of an instructor, as illustrated in Figure 2.6a, green indicates that a submission has successfully passed a specific test, while red denotes failure. Grey

```
1 build.language=c
2 build.make.command=/usr/bin/make
3 build.make.file=Makefile
4 test.timeout.testProcess=15
5 test.output.maxBytes=1048576
6 test.class.public=correctAverageRandom.py correctMaxRandom.py correctMessage.py correctNumberArguments.py
  ↪ formatSpace.py formatStyle.py invalidCorrectAvg.py invalidReadings.py returnError.py
7 test.class.release=correctMinRandom.py formatCase.py formatStream.py notEnoughArgs.py noValidData01.py
8 test.class.secret=formatNewline.py invalidCorrectMax.py noValidData02.py returnInvalid.py
```

Listing 3: Example test.properties

indicates that the submission did not compile. It is important to note that the points assigned to tests can vary, with each test not necessarily equating to a single point. The results of tests are presented in two forms: a short result indicating the pass or fail status, and a long result detailing the standard output and error captured for each test. The long result can be understood as the feedback. Assessment creators often customize tests to ensure that the feedback is novice programmer-friendly.

The student’s view of the assessment summary in Marmoset, depicted in Figure 2.7a, primarily shows release tests as marked by a question mark for the majority of submissions. This indicates that students can view the outcomes of release tests only after passing all public tests. Furthermore, as shown in Figure 2.7b, the information provided to students for release tests is more limited than for public tests, typically restricted to the pass or fail status and the test names, without the inclusion of short or long results. To access the pass and fail outcomes of release tests, students must spend a release token, which regenerates every 12 or 24 hours based on the configuration. It encourages students to pass all public tests promptly, motivating them to start working early. Regarding secret tests, their outcomes remain concealed during the ongoing assessment period. Nonetheless, instructors have the option to configure the platform to disclose the results of secret tests to students after the assessment’s deadline.

Submissions

#	submitted	public tests score	release tests score	release tested	detailed test results	Download
18	Thu, 13 Sep 2018 at 10:16 AM	8 / 8	5 / 5	Thu, 13 Sep at 10:16 AM	view	download
17	Thu, 13 Sep 2018 at 09:50 AM	8 / 8	?		view	download
16	Thu, 13 Sep 2018 at 09:48 AM	8 / 8	4 / 5	Thu, 13 Sep at 09:48 AM	view	download
15	Thu, 13 Sep 2018 at 09:47 AM	2 / 8	?		view	download
14	Thu, 13 Sep 2018 at 09:43 AM	5 / 8	?		view	download
13	Thu, 13 Sep 2018 at 09:42 AM	4 / 8	?		view	download
12	Thu, 13 Sep 2018 at 09:37 AM	did not compile			view	download
11	Thu, 13 Sep 2018 at 09:31 AM	4 / 8	?		view	download
10	Thu, 13 Sep 2018 at 09:29 AM	4 / 8	?		view	download
9	Thu, 13 Sep 2018 at 09:28 AM	4 / 8	?		view	download
8	Thu, 13 Sep 2018 at 09:26 AM	4 / 8	?		view	download
7	Thu, 13 Sep 2018 at 09:25 AM	2 / 8	?		view	download
6	Thu, 13 Sep 2018 at 09:17 AM	4 / 8	?		view	download
5	Thu, 13 Sep 2018 at 09:12 AM	4 / 8	?		view	download
4	Thu, 13 Sep 2018 at 09:05 AM	4 / 8	?		view	download
3	Thu, 13 Sep 2018 at 09:01 AM	4 / 8	?		view	download
2	Thu, 13 Sep 2018 at 09:00 AM	4 / 8	?		view	download
1	Thu, 13 Sep 2018 at 08:57 AM	did not compile			view	download

(a) Student view of the assessment summary in Marmoset

Test Results

Note: For test outcomes, *failed* means *wrong* and *error* means *crashed*.

type	test #	outcome	points	name	short result	long result
public	0	passed	1	correctAverageRandom.py	Test correctAverageRandom.py passed	passed
public	1	passed	1	correctMaxRandom.py	Test correctMaxRandom.py passed	passed
public	2	passed	1	correctMessage.py	Test correctMessage.py passed	passed
public	3	passed	1	correctNumberArguments.py	Test correctNumberArguments.py passed	passed
public	4	passed	1	formatSpace.py	Test formatSpace.py passed	passed
public	5	passed	0	formatStyle.py	Test formatStyle.py passed	passed
public	6	passed	1	invalidCorrectAvg.py	Test invalidCorrectAvg.py passed	passed
public	7	passed	1	invalidReadings.py	Test invalidReadings.py passed	passed
public	8	passed	1	returnError.py	Test returnError.py passed	passed
release	4	failed	1	noValidData01.py		

(b) Student view of the test outcomes in Marmoset

Figure 2.7: Student Views

2.5 Chapter Summary

This chapter provides a background on learning analytics, automated assessments, and automated feedback in computing education, which are essential for understanding the remainder of this thesis. Automated assessments have greatly improved the efficiency of collecting learning analytics. However, many findings from mining learning analytics cannot be replicated. The low replication rate suggests that researchers should conduct studies to better understand students' learning within their specific contexts.

Additionally, this chapter outlines several critical aspects of automated assessments. Common evaluation techniques for students' programs include dynamic analysis and static analysis. Dynamic analysis typically involves output comparison and unit testing, while static analysis often includes structural analysis and symbolic execution. These techniques are ultimately integrated into various tests. Conventionally, an automated assessment consists of two stages: a build stage and a test stage. The build stage is primarily used to ensure the student's code is compilable; once successful, tests are executed during the test stage. It is generally assumed that tests do not have dependencies.

Based on the test results, automated feedback can be generated. Most commonly, this feedback is composed by assessment creators based on their domain knowledge and experience, referred to as expert-authored feedback. While data-driven feedback also exists, it relies on the underlying traditional tests to determine if a feedback can improve the solution, *i.e.*, passing more tests. Large Language Models (LLMs) represent another source of automated feedback; however, their effectiveness in computing education has yet to be well-established.

Chapter 3

Learning Analytics Analysis

Educational data mining and learning analytics, as discussed in Chapter 2, includes a wide array of topics but is often context-dependent. Numerous studies have focused on analyzing students' data to gain insights into their behaviors [51, 9, 94]. The clustering technique was one of the methodologies used [39, 63, 94, 78], as they do not require manual configurations for categorizing students into groups, which may potentially mitigate the subjectivity introduced by researchers [99].

Although clustering techniques can be a valuable tool for researchers to understand different student behaviors, it is common that researchers only apply a single type of clustering algorithm in their experiments [44, 111, 65, 11, 73]. However, clustering results can be affected by the wrong choice of clustering algorithm, which can threaten the validity of the results. To address this issue, researchers should experiment with multiple clustering algorithms to confirm that different clustering algorithms produce similar results that the essential characteristics of the resulting clusters are identical. Our experiment confirms that the essential characteristics of the clusters are the same across multiple clustering algorithms, forming a foundation for further discussion.

Clustering is also a common technique in predicting students performance [59, 101, 96]. Such studies typically include both pre-midterm and post-midterm data. Due to the strong correlation between midterm exam grades and final exam grades [30, 59, 56, 71], which was also true in our data, evidenced by a Pearson correlation of 0.81, students who fail the midterm are likely to fail the final exam. Therefore, it is critical to identify at-risk students before the midterm. Our study integrates multiple clustering algorithms into a predictive model using pre-midterm data to identify these students.

This study applied clustering techniques to students' behaviors in their pre-midterm

submissions to the auto-grading system Marmoset [98, 97], categorizing them into different clusters. We applied multiple clustering techniques and compared their results to mitigate any effects caused by the selection of a particular clustering algorithm. We then attempted to predict students’ performance using these techniques.

3.1 Related Work

Understanding student characteristics has always been a popular interest, and clustering techniques are commonly used for this purpose. As early as 2009, Perera et al. [85] collected data from students using TRAC¹—an open-source, professional software development tracking system—over a 12-week period. During this time, students worked in groups of five to seven to develop a software solution for a client. By applying the K-Means clustering algorithm, they identified four categories of students in group work: “Managers”, “TRAC-oriented developers”, “Loafers”, and “Others”. This classification helped the audience understand the essential differences among student groups.

Later, Wiggins et al. [111] investigated the help-seeking behaviors of novice programmers within an intelligent block-based coding environment. Their analysis identified five distinct clusters of hint-asking behavior, highlighting correlations between students’ pretest scores, their perceived computer skills, and the completeness of their code when they sought hints. Notably, students with higher pretest scores and those who perceived themselves as more skilled typically completed more of their code before requesting hints.

Lorås and Aalberg [73] conducted an exploratory cluster analysis to examine variables such as students’ organization, independent study habits, planning and prioritization, time engagement, and preferences for different study environments. Their findings underscore the close ties between educational design and students’ study habits, revealing patterns like studying predominantly in the evenings and preferences for studying at home or, to a lesser extent, in computing labs and study areas.

More recently, Rahman et al. [90] applied a modified K-means (MK-means) clustering algorithm [76], which is better at selecting the optimal center point and detecting and removing noise than K-means, to students’ submission logs, test scores, and code evaluation results. This data was collected from approximately 70,000 real-world problem-solving entries from 537 students in a programming course (Algorithms and Data Structures) using an online judge (OJ) system. Four clusters were identified, revealing some interesting observations: students in clusters 1 and 4 obtained the most AC (accepted) verdicts,

¹<https://trac.edgewall.org>, accessed: 2024-Aug-14

whereas students in cluster 3 received the least. Additionally, students in cluster 3 got the most error verdicts, and those in cluster 1 received the most TLEs (time limit exceeds) compared to other clusters.

Few studies have applied clustering techniques to predict student performance [96, 80]. For instance, Sorour et al. [96] proposed a new approach based on text mining techniques applied to free-style comments written by students after each lesson to predict student performance using latent semantic analysis (LSA) and K-means clustering methods. However, clustering techniques are generally underutilized in prediction tasks. According to a literature review by Hellas et al. [59] on predicting academic performance, partitioning-based clustering algorithms were used in only 3.06% of the reviewed papers. Similarly, hierarchical and density-based clustering algorithms were each used in just 0.81% of the papers reviewed.

Our study aimed to understand students through data that can be naturally collected from automated feedback platforms, thereby enhancing the generalizability of the findings. Additionally, we demonstrated an easy-to-implement predictive model using clustering techniques. This model could potentially be integrated into the platform to alert instructors to conduct timely interventions for at-risk students.

3.2 Course Background

The data we used in the study is collected from an introduction-level programming course in an R1 university in Canada. Students were supposed to learn how to carry out operational projects using the C and C++ languages, perform procedural and object-oriented programming, and other relevant programming knowledge. The preliminary experiments were based on the data of 130 students who attended both the midterm and final exams in the course. Prior knowledge on programming was not required.

In the course, an auto-grading system called Marmoset [98] was utilized, the details of which were described in Chapter 2. Output comparison was the primary technique used to assess students' solutions. Students were allowed to make multiple submissions.

In the course, there were different types of coding questions.

1. **Coding Labs:** coding labs took place in the lab room. There was exactly one assigned project for each coding lab, which was to be completed and submitted during the lab period (2 hours in the morning). Before the midterm exam, the coding lab was scheduled weekly. After the midterm exam, the coding lab was scheduled

biweekly. There was a corresponding extended deadline (the same date but in the evening). Some students may rely on that deadline rather than finish during the lab.

2. **Homework Assignments:** homework assignments were assigned for students to do at home. They were assigned during lab time. Before the midterm, homework problems were due the following week. After the midterm, homework problems were due approximately two weeks later. In every homework assignment, there were multiple projects configured in Marmoset, all of which had the same deadline.
3. **Coding Examination:** there was an in-lab coding examination during the course. It was similar to a coding lab. However, its grade comprised a portion of the midterm grade. An extended deadline was also allowed for the coding examination.

3.3 Experiment and Results

We consider the submission as the most generalizable granularity level for understanding students, since this is the level at which students are mostly likely to consider their solutions to be complete and correct. Any mistakes made at this level indicate a gap between the students' understanding and the knowledge they need to master. Therefore, we analyze the learning analytics of students' submissions. In our study, we extracted three features that we consider most accessible across automated feedback platforms and systems.

- **passrate:** for every Marmoset project, we calculated the best score (the best number of tests passed among the submissions) a student made before the deadline. Then we divide the total number of tests of that project to form the *passrate* feature. Because every assignment had multiple projects, so for assignments, we need to sum the projects' best scores to form the best score for an assignment, and we sum projects' total number of tests to form the total number of tests for an assignment. For example, assignment 1 had 4 projects and a total number of 54 tests. If a student's best submissions of that 4 projects passed 6, 6, 7, and 8 tests separately, then the passrate of that student for assignment 1 will be $(6 + 6 + 7 + 8)/54 = 0.5$. This process was not needed for coding labs since there was only one project in a coding lab. Table 3.1 shows the total number of tests for different assignments and coding labs.
- **lastsub:** we extracted the *lastsub* feature as how many minutes between the last submission a student made and the project deadline. If a student made any submission

Table 3.1: Total number of tests for different assignments and coding labs pre-midterm

assignment	# total tests	coding lab	# total tests
a1	54	11	8
a2	85	12	19
a3	73	13	19
a4	87	14	19

before the deadline, this feature would contain a non-zero value. Only if a student did not submit before the deadline can the value be zero. Note that this feature will be zero for students whose submissions met the extended deadline but did not meet the original deadline for coding labs.

- **nsub**: the *nsub* feature represents how many submissions a student made before the project deadline for a given project. For assignments, we summed the numbers of submissions of its projects to form the nsub of an assignment.

3.3.1 Clustering results from different clustering algorithms

Hellas et al. [59] mentioned three categories of clustering algorithms: partitioning-based such as K-means, hierarchical, and density-based. We explored all these types of clustering algorithms in the study. In addition, we also explored message-passing-based such as Affinity Propagation and graph-based such as Spectral Clustering algorithms. We used the scikit-learn Python package (version 1.0.2) [84, 24]. The scikit-learn Python package is a popular choice in machine learning, and exploring its clustering algorithms should be representative. In sum, the clustering algorithms explored in the study included K-Means (KM), Affinity Propagation (AP), Spectral Clustering (SC), Hierarchical Clustering (HC), and Density-based Spatial Clustering (DBSC). Explanations of these algorithms are in Appendix A.

We standardize every feature by removing the mean and scaling to unit variance before clustering. For a sample x , the standard score is calculated as:

$$z = \frac{x - u}{s}$$

where u is the mean of the samples and s is the standard deviation of the samples.

We tested different options for setting the number of clusters in different algorithms. We found that setting the number to 3 will give us good interpretive results. In this section, we will only compare the labelling. For the characteristics of different clusters, we will discuss them in Section 3.3.2.

Table 3.2 presents the size of different clusters using different clustering algorithms. To compare the cluster results across different clustering algorithms, we used *adjusted Rand index* for evaluation. Random labelling samples will make the adjusted Rand index close to 0.0, and the value will be exactly 1.0 when the clustering results are identical. Table 3.4 presents the results. Note that a cluster may be given a different label in different clustering algorithms. We re-labelled it according to its characteristics detailed in Section 3.3.2.

The adjusted Rand index computes a similarity measure between two clusters by considering all pairs of samples and counting those that are assigned to the same or different clusters. It ensures that its value will be close to 0.0 for random labeling, independent of the number of clusters and samples, and exactly 1.0 when the clusters are identical.

Table 3.2: The size of different clusters in different clustering algorithms

	KM	AP	HC	SC	DBSC
cluster 1	62	56	48	58	32
cluster 2	59	65	75	63	89
cluster 3	9	9	7	9	9

Table 3.3: The adjusted Rand index results

	KM	AP	SC	HC	DBSC
KM	-	0.78	0.68	0.55	0.32
AP	0.78	-	0.73	0.47	0.41
SC	0.68	0.73	-	0.51	0.38
HC	0.55	0.47	0.51	-	0.52
DBSC	0.32	0.41	0.38	0.52	-

From Tables 3.2 and 3.4, we can see that some clustering algorithms labeled students similarly, while others did so differently. For example, the clustering results from KM, AP, and SC are similar to each other, as they have similar cluster sizes and adjusted Rand index values around 0.7, which corresponds to only roughly 10% mismatching pairs. However, HC and DBSC seem to function differently, since adjusted Rand index between them and

Table 3.4: The adjusted Rand index results (high to low)

first algorithm	second algorithm	adjusted Rand index
KM	AP	0.78
AP	SC	0.73
KM	SC	0.68
KM	HC	0.55
HC	DBSC	0.52
HC	SC	0.51
AP	HC	0.47
AP	DBSC	0.41
SC	DBSC	0.38
KM	DBSC	0.32

other algorithms are low. For example, KM and HC has an adjusted Rand index as 0.55, which corresponds to roughly 20% mismatching pairs; and KM and DBSC has an adjusted Rand index as 0.32, which corresponds to roughly 35% mismatching pairs. To properly avoid potential bias, one should consider combining KM, HC, and DBSC.

3.3.2 Characteristics of students

We carefully examined students in different clusters. Interestingly, although the cluster results differed in the cluster size and the adjusted Rand index metric, we found that students in different clusters share similar characteristics across different clustering algorithms. We name the students in the three clusters as: *Potential-Top-Performance (PTP)* students, *Potential-Poor-Performance (PPP)* students, and *Mixed-Performance (MP)* students.

Figure 3.1 and Figure 3.2 demonstrates the characteristics of different students. Each point in the figures represents a mean and its 95% confidence interval from one of the clustering algorithms. The lastsub feature of the first coding lab for PTP and MP students were not shown because the first coding lab was due on the second day rather than the 2-hour lab time. Therefore, they are out of the y-axis range in Figure 3.2. In general, we can observe that PTP students tend to finish their work early (lastsub) and achieve the highest passrates, while PPP students tend to finish their work late and achieve the lowest passrates. An interesting observation is that MP students tend to make the highest number of submissions for assignments; however, their passrate is still lower than that of PTP students. This may indicate the ineffectiveness of the automated feedback provided.

We also examined the students that were put into different clusters from different algorithms. In general, those students put into different clusters from different algorithms were those whose behaviour was in the middle of PTP students and MP students or the middle of PPP and MP students. However, we found there were no students put into the PTP cluster in one algorithm while being put into the PPP cluster in another algorithm. We can tell that students from these two clusters share no typical behaviour from any aspect.

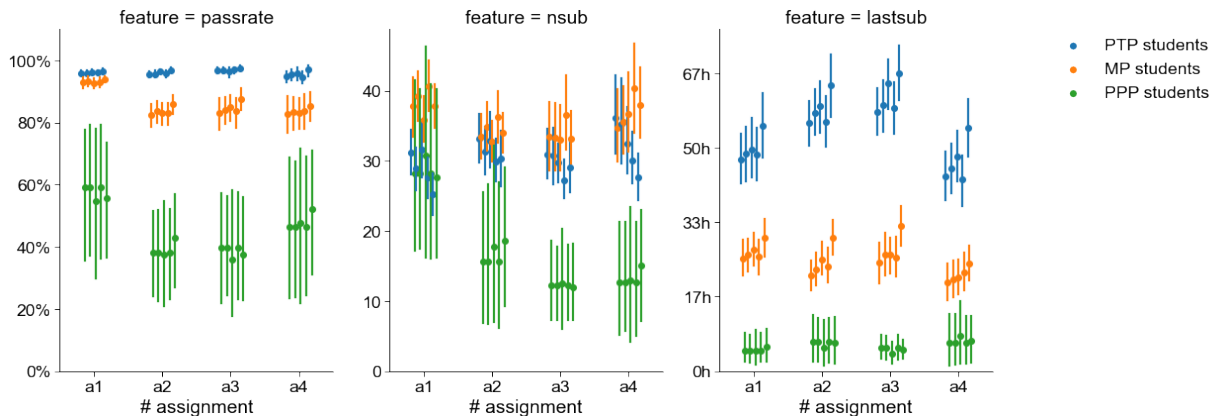


Figure 3.1: Assignment Characteristics

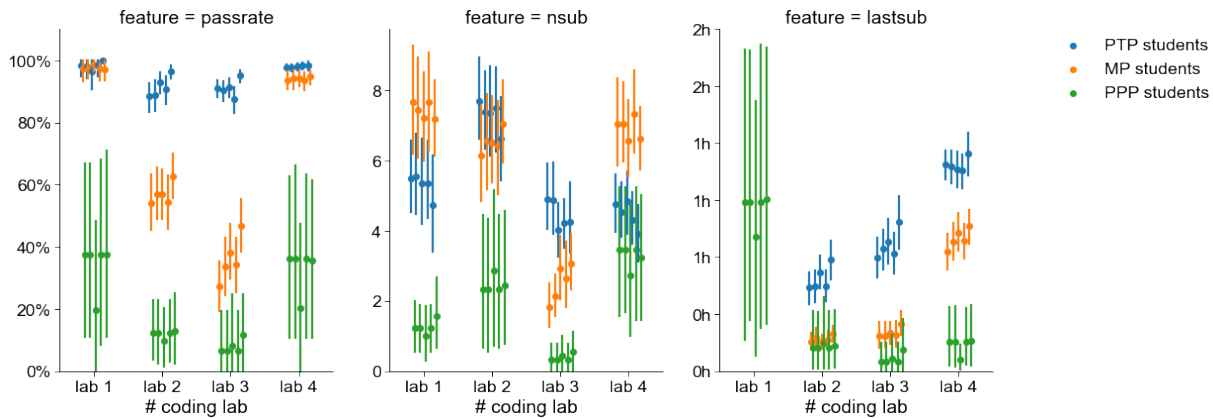


Figure 3.2: Coding Lab Characteristics

3.3.3 Consistency in student behaviours

Table 3.5 summarized how many assignments and coding labs used before the midterm and after the midterm. Table 3.6 summarized the number of tests in assignments and coding labs after the midterm. Note that lab 1 and lab 7 were excluded since they were due on a different date.

Table 3.5: Assignments and coding labs before and after the midterm

pre-midterm	post-midterm
a1, a2, a3, a4	a5, a6
l2, l3, l4	l5, l6

We applied the Welch's t-test against pre-midterm data and post-midterm data cluster by cluster and algorithm by algorithm (in total 5 algorithms, 3 clusters each). Welch's t-test is an adaptation of the standard Student's t-test and is particularly useful when comparing the means of two groups that may have unequal variances and possibly unequal sample sizes, as was the case in our study. Then we checked the p values (significant level as 0.05) to see if the data is consistent (equal means). A p value larger than or equal 0.05 indicates the data was consistent; while a p value that is less than 0.05 indicates the data was not consistent, they are marked by an asterisk. Table 3.7 shows us the Welch's t-test results of assignments and Table 3.8 shows that of coding labs.

Table 3.6: Total number of tests for different assignments and coding labs post-midterm

assignment	# total tests	coding lab	# total tests
a5	104	15	22
a6	133	16	31

Table 3.7: The p values of the Welch's t-test results of assignments

algorithm	feature	PTP	MP	PPP
AP	assignment passrate	$5.32e - 18^*$	$1.47e - 23^*$	0.11
	assignment lastsub	$2.32e - 15^*$	$1.15e - 05^*$	0.30
	assignment nsub	$4.97e - 10^*$	$7.48e - 03^*$	0.92
DBSC	assignment passrate	$9.86e - 12^*$	$5.43e - 28^*$	0.13
	assignment lastsub	$1.64e - 10^*$	$4.03e - 10^*$	0.04^*
	assignment nsub	$7.56e - 13^*$	$1.99e - 05^*$	0.92
HC	assignment passrate	$1.01e - 14^*$	$3.10e - 22^*$	0.37
	assignment lastsub	$4.31e - 14^*$	$4.07e - 07^*$	0.24
	assignment nsub	$2.50e - 09^*$	$7.85e - 05^*$	0.99
KM	assignment passrate	$4.25e - 19^*$	$2.42e - 20^*$	0.11
	assignment lastsub	$6.45e - 16^*$	$5.85e - 05^*$	0.30
	assignment nsub	$8.05e - 09^*$	$1.24e - 03^*$	0.92
SC	assignment passrate	$4.81e - 17^*$	$3.72e - 23^*$	0.11
	assignment lastsub	$1.82e - 14^*$	$3.38e - 06^*$	0.30
	assignment nsub	$9.11e - 12^*$	$1.51e - 03^*$	0.92

Table 3.8: The p values of the Welch's t-test results of coding labs

algorithm	feature	PTP	MP	PPP
AP	lab passrate	$3.13e - 02^*$	$4.38e - 01$	0.23
	lab lastsub	$2.16e - 08^*$	$1.75e - 06^*$	0.61
	lab nsub	$1.77e - 07^*$	$5.69e - 07^*$	0.02*
DBSC	lab passrate	$7.82e - 03^*$	$8.34e - 01$	0.14
	lab lastsub	$9.49e - 06^*$	$3.53e - 09^*$	0.42
	lab nsub	$1.36e - 05^*$	$7.72e - 10^*$	0.02*
HC	lab passrate	$1.68e - 02^*$	$5.18e - 01$	0.17
	lab lastsub	$2.14e - 07^*$	$1.07e - 07^*$	0.71
	lab nsub	$5.45e - 06^*$	$8.94e - 09^*$	0.07
KM	lab passrate	$1.76e - 02^*$	$3.12e - 01$	0.23
	lab lastsub	$1.04e - 09^*$	$4.86e - 05^*$	0.61
	lab nsub	$6.21e - 08^*$	$1.62e - 06^*$	0.02*
SC	lab passrate	$2.30e - 02^*$	$3.57e - 01$	0.23
	lab lastsub	$1.92e - 08^*$	$2.38e - 06^*$	0.61
	lab nsub	$1.37e - 07^*$	$4.49e - 07^*$	0.02*

From Table 3.7, we can see for PTP students and MP students, none of their behaviour was consistent for assignments. Since the assignments were assigned bi-weekly after midterm and the difficulties and the number of test cases of assignments increased, it is reasonable that PTP students' and MP students' behaviours on assignments became not consistent. By looking at the median and the mean values, we think the trend can be summarized as: *the last submission submitted by them was not as early as they did before the midterm, even though they had one more week to work on it; the number of submissions increased; their assignment performances decreased.*

From Table 3.8, we can see for coding labs, both PTP students and MP students tended to make more submissions for post-midterm coding labs, which causes the number of submissions inconsistent with the number before the midterm. Also, they both tended to submit not as early as they did before the midterm. However, for how much percentage of test cases was passed, it's interesting to see that MP students were having similar performances as they did before the midterm; while PTP students' performance decreased.

Again from Table 3.7 and Table 3.8, we can see for PPP students, their performances on both assignments and coding labs were consistent pre-midterm and post-midterm. They also submitted late consistently for both assignments and coding labs (Except for PPP students from the DBSC clustering algorithm). They made consistent number of submissions for assignments but they slightly ($p = 0.02$) tended to make more submissions for coding labs post-midterm than pre-midterm (Except for PPP students from the HC clustering algorithm). It seems students who were clustered into this cluster were not trying to change their behaviour much in terms of passrate and lastsub; or it might mean PPP students became anxious after the midterm so they made more submissions in coding labs, but they could not find an effective way to really improve their performance.

3.3.4 Exam grades of students

Because some students were put into different categories from different clustering algorithms, for this section, we consider students in the PTP cluster only if they were put into the PTP cluster by all clustering algorithms. Similarly, students in the PPP cluster were only those students who were put into the PPP cluster by all clustering algorithms. The remaining students will be MP students. This approach forms a simple but effective predictive model to make a binary prediction on if a student needs help in passing the midterm and the final exam.

In addition to the midterm grades and final exam grades, there was a coding examination grade, which will comprise a portion of the midterm grade. Figure 3.3 shows us the

relation between the coding examination grades and different clusters. Figure 3.4 shows us the relation between midterm grades and final grades of different clusters.

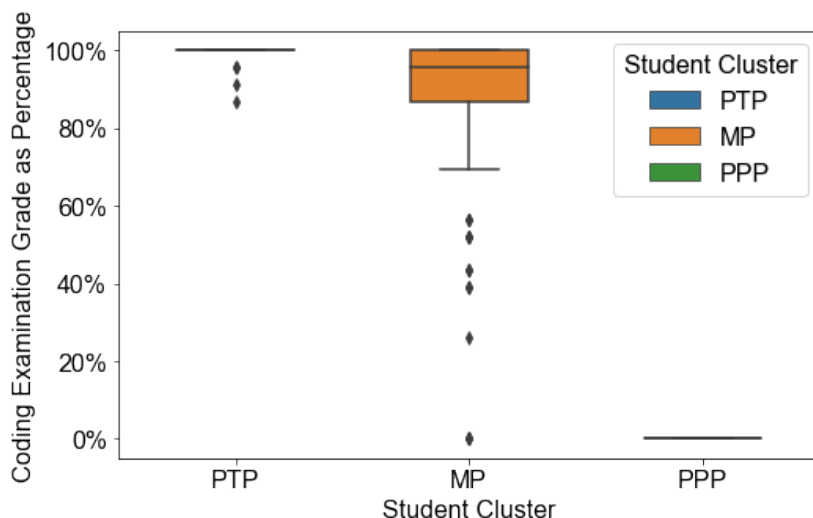


Figure 3.3: Box-plots of coding examination grades

From Figure 3.3, we can see PTP students achieved a very high score on the coding examination, while PPP students achieved 0% on the coding examination. The result is expected since PTP students mostly performed well on assignments and coding labs, which were programming questions. It is reasonable that they achieved a high score on the coding examination. In contrast, for PPP students, since they performed poorly on those questions, it is not surprising that they got a significantly low score.

From Figure 3.4, we can see the performances of MP students mixed up with other clusters of students. However, if we exclude them, as shown in Figure 3.5, we can see the performances of PTP students and PPP students were completely different. The reason we set a cut off point as 50% for the exam grades is that it is the required grades for passing the course.

In addition, we looked into the students who had midterm performance between 50% and 60% (labelled as 1, 2, and 3 in Figure 3.5). We found that Student 1 and Student 2 made early submissions for some assignments before the midterm. Although most of those submissions ended up with an unsatisfactory passrate, they might have alerted these students to prepare for the midterm early, which led them to achieve slightly better grades than most other PPP students. We did not observe anything special from Student 3 that

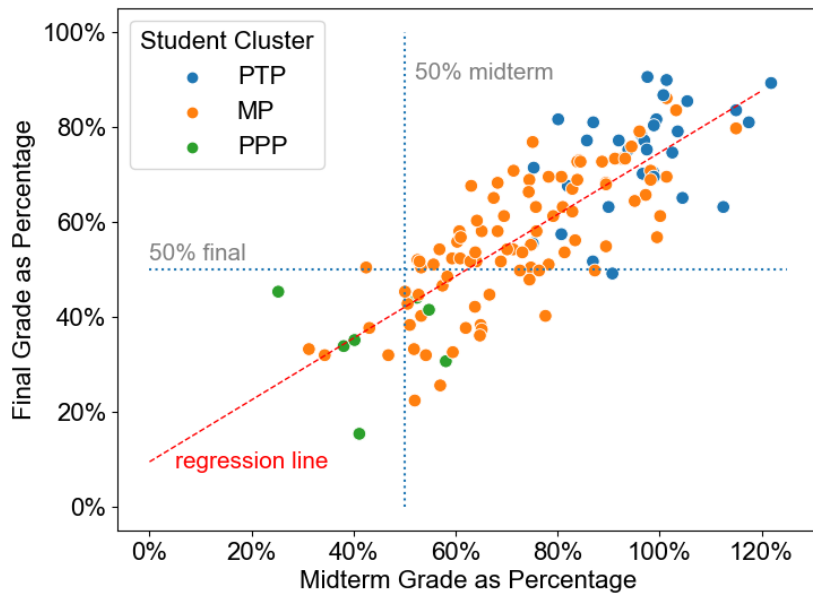


Figure 3.4: Midterm grades and final exam grades including bonus midterm questions

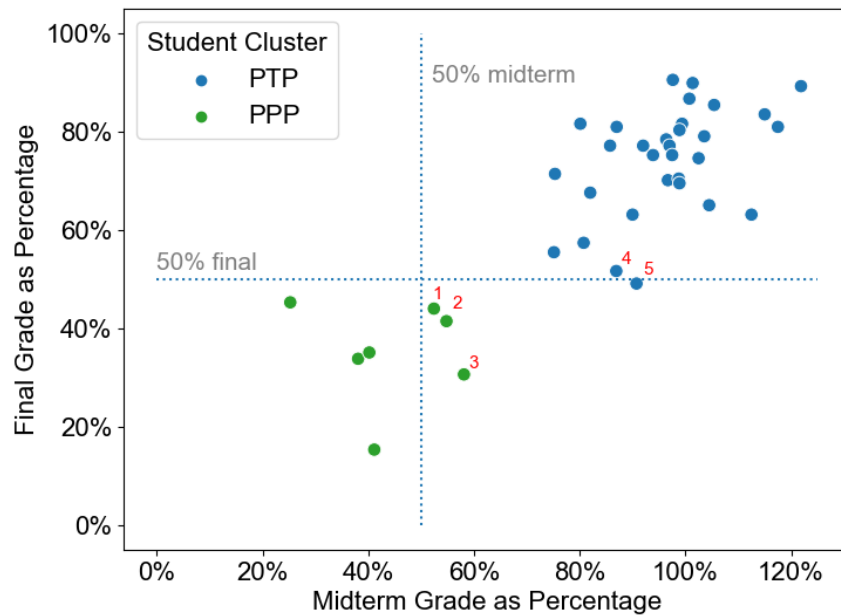


Figure 3.5: Midterm grades and final exam grades including bonus midterm questions

makes this student apart from other PPP students. The reason why this student achieved a slightly better midterm grade remains unclear.

We also looked into the two students who got roughly 85% on the midterm while later only got roughly 50% on the final (labelled as 4 and 5 in Figure 3.5). We found they had a significant performance drop on assignments and labs when they contained object-orient programming concepts. It might indicate that both Student 4 and Student 5 faced significant difficulty understanding those concepts, which in turn dragged down their final exam performance.

Due to limited teaching resources, it is important to allocate these resources effectively [30]. Therefore, the only metric we consider in evaluating our predictive model is precision, *i.e.*, how many students actually achieved more than 50% on the exam from the identified PTP students and how many achieved less than 50% from the identified PPP students. The higher the precision, the better. For our predictive model, the precision of the PPP students is $4/7 = 0.57$ (4 students out of the total 7 PPP students were below 50% in the midterm); and the precision of the final grades is 1.0, since no PPP students had a final grade above 50%. The precisions of the PTP students are 1.0 for both the midterm and the final exam. These results are promising, implying the effectiveness of the proposed predictive model.

3.4 Chapter Summary

In this study, we applied clustering techniques on pre-midterm students' behaviour data, namely how early students make their last submissions, how many submissions they make, and what the best score of those submissions is. We found although different clustering algorithms may produce different clusters in terms of the sizes and adjust Rand index metrics (in other words, they may label students differently and put them into different clusters), we found those clusters actually share the same characteristics. We can summarize those clusters as: potential-top-performance (PTP) cluster, potential-poor-performance (PTP) cluster, and mixed-performance (MP) cluster.

For pre-midterm, PTP students generally achieved very high scores on assignments and coding labs before the midterm, and they submitted their last submissions early. On the contrary, PPP students achieved low grades on assignments and coding labs before the midterm, and they tended to submit late. MP students shared some similarity of those two clusters of students, but their behaviour were typically in the middle of them. That's also the reason why some of them will be labelled differently from different clustering algorithms.

We also looked into whether students have consistent behaviour pre-midterm and post-midterm. We found PTP students and MP students mostly have inconsistent behaviour before and after the midterm, while PPP students have consistent behaviours. Combining the insights above, it suggests that if a student consistently achieves low grades and submits work late before the midterm, they are likely to be PPP students who need assistance.

Lastly, we proposed a binary predictive model to determine whether a student needs help in passing the midterm and the final exams. We consider precision to be the most essential metric for evaluating our model, which has achieved very promising results.

Chapter 4

The Value of Time Extensions

Numerous studies have reported that students tend to perform well if they start or finish work early [71, 6, 9, 30]. However, leveraging this observation effectively remains a challenge. Specifically, there is no straightforward mechanism to set up a particular time point for instructors to identify students who may require assistance. While instructors can implement complex adjustments to course configurations, this chapter presents an alternative strategy: offering time extensions can effectively address the issue.

Time management skills affect students academic success [112]. By using strict deadlines, instructors can assist students to organize and prioritize assignments. In addition, it also reduce chaos when instructors want to conduct actions that can only be conducted until all students submit their answers. For example, instructors need enough time between assignment deadlines to do the grading [7], or they want to provide sample solutions [48]. However, strict deadlines can also be problematic, as they may be unrealistic or overly demanding in some cases. For instance, students dealing with a heavy workload or personal challenges may struggle to meet strict deadlines, leading to increased stress [60] and decreased performance [77].

On the contrary, flexible deadlines offer a different approach to manage student work. By providing a range of dates—often in the form of time extensions—within which students can submit their work, instructors can allow some flexibility in the event of unexpected challenges or changes in circumstances [8, 93]. This can be particularly useful for students who are facing unforeseen personal or academic difficulties. Additionally, students learn autonomy and self-regulation under flexible deadlines, as they learn to be responsible for managing their own times. However, flexible deadlines increase the possibility that students procrastinate their work [26, 104] as they may perceive the extra time as an excuse to delay

starting their work, leading to decreased motivation and lower quality of work [30, 51].

Instructors often deploy flexible deadline policies in programming courses for the reason that these courses often involve heavy workload [38], frequent assignments, and complex concepts [113, 102]. Students are likely to meet unexpected situations, especially when they lack the skill to estimate the time needed to implement and debug their programs [16].

Previous studies have found strict deadlines had a negative direct effect on the assignment performance [77], while flexibility in deadlines could significantly improve overall grades on assignments [17, 83]. However, researchers have yet to determine if such flexibility can result in deeper learning which would be reflected in better exam results. Neither, they have not yet investigated the reasons why students use them.

In this chapter, we tried to answer two research questions.

- **RQ1:** Is there a correlation between the use of time extensions and students performance?
- **RQ2:** Why do students use time extensions?

We meant to conduct a single study investigating the research questions using data from a final-year programming course. However, it was possible to validate the RQ1 result using different data from other programming courses. Therefore, we conducted one *Replication* study and one *Reproduction* study. The *Main* study investigated both RQ1 and RQ2, and the other two studies validated the RQ1 result.

For RQ1, we found that students who used time extensions had a significantly lower assignment and exam grades than those who chose not to. For RQ2, we found that students considered the top two reasons for using grace days were conflicting obligations from other curricular requirements and under-estimation of the assignment requirements.

This paper reveals the predictive power and provides essential knowledge of time extensions to both educators and learners.

4.1 Related Work

Depending on how educators configure deadlines, one way to apply the flexible policy is to eliminate deadlines [20, 26], where students have the greatest control over what they want to prioritize and what to learn. The learning process thus becomes completely self-paced. This format, albeit common in Massive Open Online Courses (MOOC) [47], is

rarely adopted in traditional computing education. Campbell et al. [26] once designed a self-paced CS1 course. However, they found it needed significantly more resources and time for instructors than a regular course, students hesitated to treat mastery quizzes as formative, and too much flexibility provided few incentives to help students stay on track, leading to considerable procrastination. Therefore, completely eliminating deadlines (or late penalties) may not be as helpful as expected [19, 50].

Intermediate deadlines are helpful to keep students on track on large projects [45, 35] and may assist self-learning [15]. However, instructors usually apply penalties if students fail to meet such intermediate deadlines, making them strict deadlines rather than flexible deadlines. We have yet to find studies exploring the relationship between student performance and their lateness in this setting.

Other than the previous two options, one popular option to deploy a flexible policy is to give students a fixed number of grace days which can be seen as their time banks, for them to use at their discretion [12, 93, 13]. This approach takes a trivial amount of effort to apply. Also, because students need not explicitly ask for extensions, it helps reduce students' stress [60]. Students can focus on the actual learning rather than conducting unhelpful actions to ensure submitting everything on time. Surprisingly, students are good at tracking their remaining grace days [13], so instructors only need to put a little effort into tracking the grace day usage.

Many studies showed that flexible policies improve learning outcomes [17, 60, 93, 86]. For instance, Becker [17] tried different strategies in class across multiple terms. They found that flexibility in deadlines, combined with the option to re-submit assignments, could significantly improve overall grades on assignments. However, few studies looked into the relationship between students' behaviour under flexible policies and their comprehensive performance, *e.g.*, final exam performance. Even fewer studies looked into why students use time extensions—no need to mention the rareness of such studies in computing education.

4.2 Course Background

This section presents the background of the three involved courses.

Main Study: Final-Year Distributed Computing Course

The course introduces software techniques for distributed computing with a focus on data storage and analytics. There were five programming assignments, which were worth 60%

(10%, 15%, 15%, 15%, and 5%) and one closed-book on-campus final exam, which was worth 40%. The first assignment was done individually, and the other four assignments were done in groups of up to three students. Students could enrol in different groups for different group assignments. The data was collected from 146 students.

Grace day policy Every student was given a total of 5 grace days which can be considered as their time banks for all assignments. The lateness is calculated after deducting their grace days. Late assignment submissions were penalized 1% per hour or part thereof. The instructor did not restrict the maximum grace days students could use for each assignment. However, once they decided, they could not adjust it afterwards. Each grace day can only be used as a whole, even for a lateness of only one hour.

For group work, instead of averaging the available grace days of group members [13], the instructor let group members decide how many days they want to use. If the number exceeds the remaining grace day of any group member, then only that member will take the lateness penalty. This approach could potentially influence group formation, particularly when students have varying numbers of remaining grace days. However, it appears that students do not consider grace day usage as a crucial factor when joining groups.

The instructor announced the grace day policy ahead of the first assignment. Additionally, the instructor team sent a confirmation email asking how many grace days an individual or a group would like to use if they observed a late submission. A group representative was expected to reply if it was sent to a group. The email explained the grace day policy, asking about their proposed grace day usage, and an open question about why they chose to use it. Students may or may not reply to the email. If one did not reply to the email, a minimum number of grace days would be used, which would just cover the lateness hours. For example, if one submitted 12 hours late and did not reply to the email, one grace day will be used by default. Note that the lateness is calculated by summing the grace days needed for each assignment. For example, if a student is 2 days and 10 hours late for assignment 3 and 5 hours late for assignment 4, then the grace days needed will be 4 days (3 days + 1 day), not 3 days as calculated from the summed late hours (2 days 15 hours).

Replication Study: Final-Year Programming Course

The course discusses techniques and theories to assist programmer to improve program performance, such as queuing theory, bottlenecks recognition, profiling, and so on. Also, numerous techniques for programming multi-core processors are discussed, such as cache

consistency, vectorization, single-instruction multiple-data (SIMD), and so on. A programming language—the Rust programming language—which most students are not familiar with are required for assignments.

There were four programming assignments each worth 16% and one open-book on-campus final exam, which was worth 35%. The remaining 1% was assigned to the academic integrity exercise, which was to ensure students were well-aware of the academic integrity policy. All assignments were done individually. Students had to achieve at least 30% on the final exam to pass the course. The data was collected from 438 students.

Grace day policy Every student was given a total of 5 grace days for all four assignments. Grace days were counted in units of whole days. Students had to use Git [105] to upload their solutions. The last commit time on the Git server on the default branch was used to determine the submission time. If a student used too many grace days, the sixth day of lateness causes the lowest assignment mark to be halved. The seventh day causes the lowest two assignment marks to be halved. If there were more than seven days of lateness, assignment marks would be converted to zero and associated grace day usage would be dropped. There was no credit for unused late days. Grace days were posted soon after an assignment was due for transparency.

Reproduction Study: First-Year Introduction to Programming Course

This course is to provide students fundamental concepts of programming. Students were supposed to learn how to carry out operational tasks using the C and C++ languages, perform procedural and object-oriented programming, and other relevant programming knowledge.

Marmoset [98], detailed in Chapter 2 was also used in the course. The instructor team set up Marmoset tests following a principle, which is *the tests covered in release and secret tests should connect to public tests, so that students who pass all public tests should be able to pass all release and secret tests*. For example, if a public test is testing if the student program can calculate the median of 15 values fed into the program in a streaming manner, then a release test can be designed to test if the program can calculate the mode of the 15 values. Therefore, if a student managed to pass all public tests in the lab, it meant that the student has learned the knowledge.

There were different types of coding questions, namely homework assignments, coding labs, and the coding examination. All coding questions used Marmoset and were supposed to be done individually. In this study, we only looked at the coding labs since they were the

questions where instructors granted students time extensions. Coding labs took place in the lab room. There was exactly one programming question for each coding lab, designed to be completed during the lab period. The lab period was 2 hours but students should be able to finish them within an hour if they paid close attention to lectures, regardless of their prior programming experience. Instructors set an extended deadline on the same date in the evening. The penalty was trivial and close to zero since the intention was to motivate students to finish the lab and learn things rather than providing a summative judgment. Therefore, some students may rely on the extended deadline instead of trying hard to finish their work in the lab. Both strategies were legitimate. Before the midterm exam, the coding lab was scheduled weekly. Among all coding labs, the first and last labs were not included in the study since their deadline was extended, much longer than the 2-hour time limit. The earliest lab used in our study was the second lab, which was scheduled during the second week.

Similarities and Differences

Two aspects of the course settings need further elaboration, which are presented here.

Time extension formats All three courses offer students time extensions, but in different formats. Both the two final-year courses offered students a fixed number of grace days which they could use at their discretion. That indicates the deadlines were determined by students, implicitly implying a requirement of time management skills, which should not be an issue since they were all final-year students and should have developed the skill in previous years. The first-year course, however, was not that flexible in terms of time extensions. It only gave students two deadlines, one was early in the morning, and the other was in the evening, which may lessen the necessity of good time management skills.

Evaluation of student performance The three courses evaluated the overall performance using an on-campus final exam, and students needed to do it individually. However, courses involved in the main study and reproduction study were closed-book, while the course involved in the replication study was open-book. Additionally, there were some differences in non-exam evaluations. Specifically, the course involved in the main study allowed students to work on some assignments in groups, assigned the same group grade to all group members, and allowed students to form different groups for different assignments. The other courses, however, required students to work individually. The grading involved in the main study took around a week before the instructors released grades back

to students, it took longer—roughly two to three weeks—for the replication study; however, students involved in the reproduction study were able to see their public test grades immediately since they were auto-graded.

4.3 The Main Study

Method for RQ1 We categorized the students into different categories based on whether they used grace days. Additionally, we put final-year students whose lateness could not be covered by the five grace days into an extra category for the reason that there was a non-trivial number of such students. Table 4.1 summarizes the category information for these students.

Table 4.1: Student categorizes and sizes in the main study

category	size	description
0	25	On time
1-5	99	Used up to five grace days for the term
> 5	22	Five grace days were not enough

We will then analyze their assignment and final exam grades to understand the connection between grace day usage and performances. We applied Kruskal-Wallis H -test with Conover’s posthoc test in the analysis. We adjusted the p-value using Bonferroni correction.

The Kruskal-Wallis H -test is a non-parametric statistical method designed for comparing independent groups when the data do not conform to the normal distribution assumption or when the data distribution is unknown. By comparing the ranks of data points across groups, the Kruskal-Wallis H -test assesses whether data points from one group stochastically dominate those from any one of the other groups. The null hypothesis is that all students had no difference in the ranks of their grades.

When the Kruskal-Wallis test indicates significant differences, it doesn’t specify which specific groups differ from each other. Conover’s posthoc test is used for this purpose. It involves pairwise comparisons of the group ranks to identify where the differences lie.

The Bonferroni correction is a method used to adjust the significance levels when conducting multiple pairwise tests, such as dividing the original significance level 0.05 by the number of comparisons being made. For example, if 10 comparisons are being made, each test would need a significance level of 0.005 to be considered statistically significant.

Method for RQ2 Overall, we sent 129 emails seeking grace day usages and reasons. There were 113 replies (87.6%) in total, and 79 out of them (61.2%) gave reasons.

We conducted thematic analysis [21] on students’ email replies. After getting meaningful themes, we count the themes to get an initial quantitative insight into students’ reasons. Note that if a student or a group reported reasons that fall into multiple themes, each of those themes would be counted.

Results and Discussion for RQ1 For both experiments, the Kruskal-Wallis H -test results were statistically significant. In both Figure 4.1 and Figure 4.2, the green triangle represents the mean. Significant differences are marked by an asterisk.

Figure 4.1 shows that students who did not use any grace day performed significantly better than other students in the final exam. Also, students who did not need more than five grace days performed significantly better than those who needed more than five grace days. It indicates that students who needed more grace days tended to achieve lower assignment grades, which has a certain similarity with prior studies where students tend to perform poorly if they submit their work late [51, 6]. However, in the course, students could pick deadlines of their own will. Ideally, students could choose a time when they validated the correctness of their assignment solutions and thus achieve identical scores as top students. However, the difference in assignment performance indicated that students who used grace days also lacked the skill to validate their answers.

Figure 4.2 shows that the highest exam grade appeared in the “1-5” students. Since the very top students can learn quickly and manage their time well, it is not surprising that they earned the very top grades.

In addition, we can see that students who did not use any grace day performed significantly better than other students. However, no significant difference was found between students who used grace days.

From assignment to final exam performance Students who did not use grace days tended to perform better on both assignments and the final exam. One possible reason is that such students were self-disciplined. They knew how to schedule their time and how to tackle challenges. They were equipped with better self-learning skills so they performed well in the end.

For the two categories in which students used grace days, their performance showed a significant difference for assignments but not in the final exam. In addition, the lowest exam grade appeared in the “1-5” students. Since students worked in groups for some

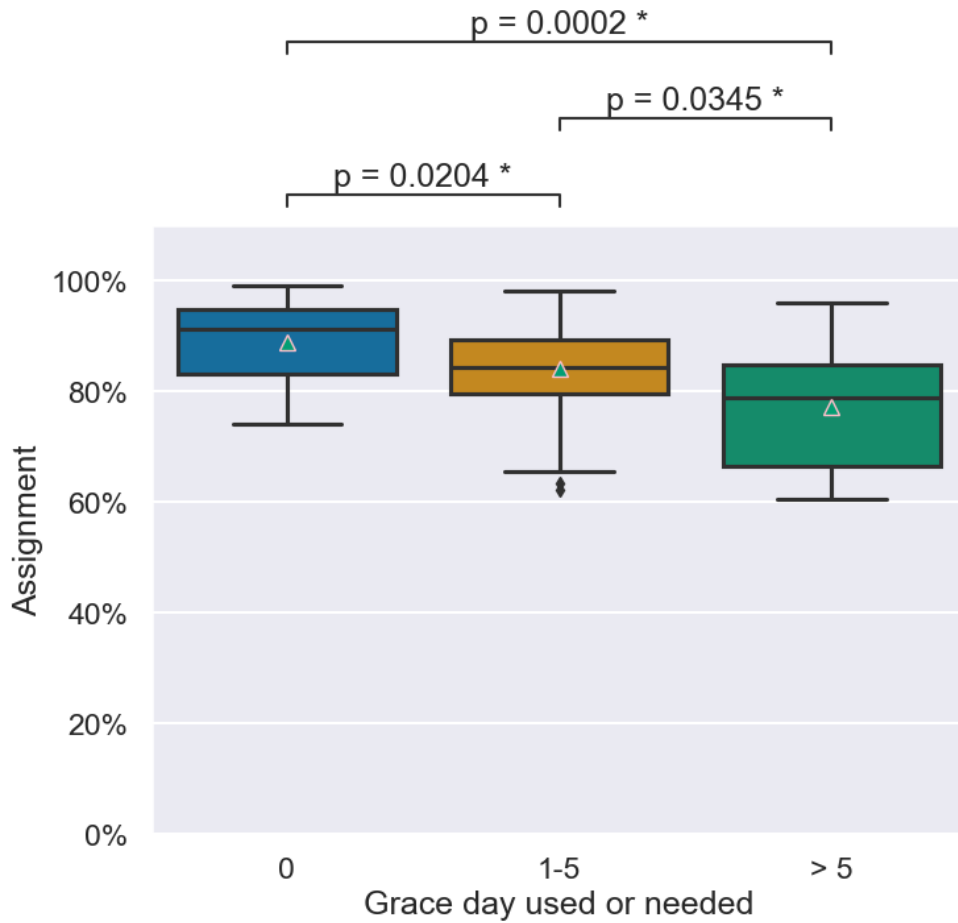


Figure 4.1: Box-plots of grace day usage and assignment performance

assignments, therefore, it is possible that some students were carried by their group mates. It is expected that such students could not perform well in the final exam.

Students in other categories could also be carried in group assignments. However, a student who ended up using zero grace day would still be somewhat competent, since the first assignment was an individual assignment and it was not easy. On the other hand, a student who ended up using over five grace days indicated that the groups the student joined were not very highly competitive. Otherwise, the student would not be that much late. For both cases, we consider the group carrying effect was weaker than that among the “1-5” students. It may also explain why the “1-5” students had the largest difference between the maximum and minimum in the final exam grades.

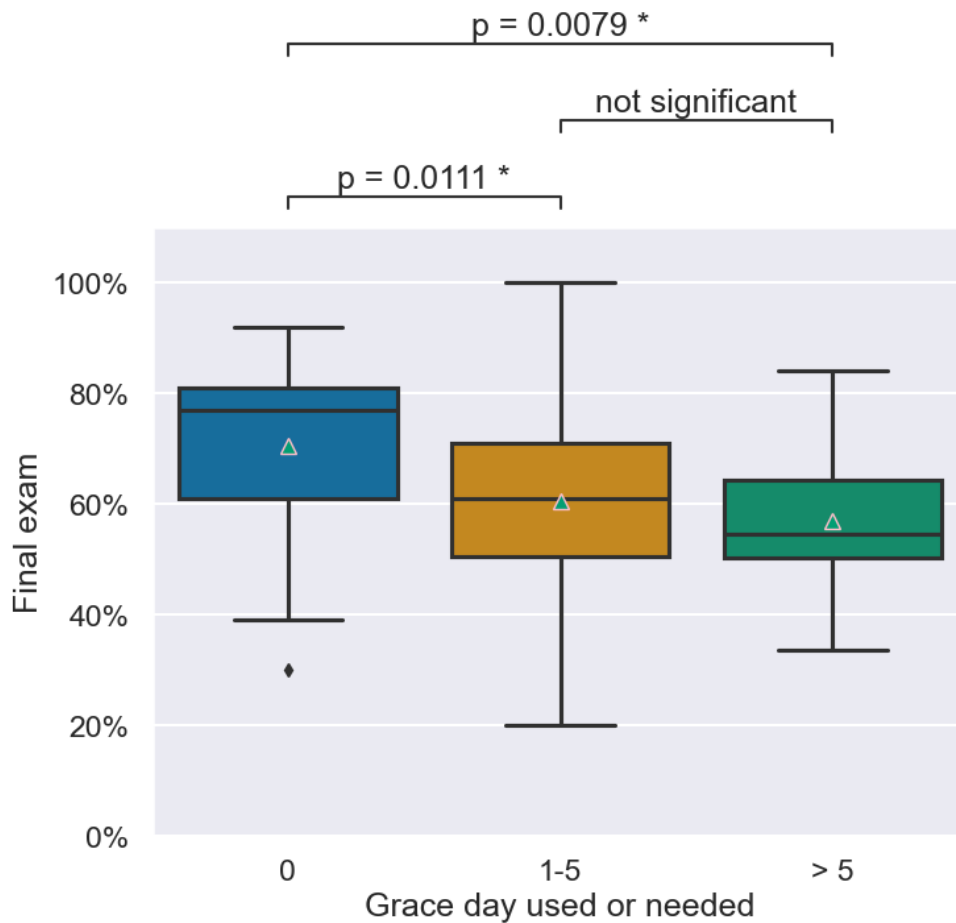


Figure 4.2: Box-plots of grace day usage and final exam performance

Results and Discussion for RQ2 Overall, we categorized the reasons into two major themes.

- **Conflicting obligations from other curricular requirements:** *e.g.*, an assignment due in another course, the final year design project (FYDP), which was required for final-year students, or finding co-op jobs (required in some Canadian universities).
- **Under-estimation of the assignment requirements:** *e.g.*, starting late, debugging took longer than anticipated, or needing additional time to ensure that the implementation was bug-free and could execute efficiently

The majority of students (in thirty-eight replies, 29.5%) talked about conflicting obligations from other curricular requirements. For example, one reply stated, “*We had to use grace day for the assignment submission due to the bulk of assignment deadlines on the week of [assignment]. We had other assignment deadlines from other courses that week and weren’t able to finish [assignment] by the original deadline*”. Also, “*(We) had some tight FYDP deadlines early this week that we needed to unfortunately prioritize*”. Lastly, “*This assignment was due in the week of [job searching time period] and all of our group members were actively applying jobs . . . That time period should be the busiest for all students*”.

Many students (in twenty-nine replies, 22.5%) talked about underestimating the assignment requirements. Some replies stated they underestimated the assignment, “*The reason we submitted late is because we underestimated the time it would take to implement [functionality] . . .*”. The majority of replies talked about last-minute modifications. For example, “*We used a grace day because we had issues with the implementation for a certain edge case . . .*”, “*We used the grace day to do some more testing for our assignment solution*”, and, “*We submitted it late because we wanted to ensure our code was error-free. We wanted to spend one grace day for debugging*”. Some replies talked about starting late, “*We had to use it because we started a little late if I remember correctly and would prefer to use the late day than stress out the night of*”.

Other reasons (in seventeen replies, 13.2%) include:

- Conflicting obligations from non-university life, “*I submitted a paper to [a conference] and needed the time last week to finish the submission*”, and, “*I decided to take grace days because I was busy organizing club events during these days, and did not have enough time to finish my assignment on time*”. These reasons only showed up in the first assignment, when the workload was light.
- Careless, “*I used the grace day because I realized I submitted an earlier revision of my code after the deadline had passed*”
- Assignments were too difficult, “*The reason we decided to use a grace day is because the assignment was very difficult and we were not able to finish it in the given time*”.
- Use them since it was the last assignment, “*we decided to use the grace days because we hadn’t used them yet from the other assignments*”, and, “*I decided to use them all up since this is the last assignment and saving late days doesn’t matter beyond this point*”.
- Illness, “*The main reason why we used these days was because two of us had COVID-19 two weeks ago, which resulted in us losing time for working on this assignment.*”

Since each above reason was in a trivial number of replies and there are no explicit connections between them, therefore, we did not consider them as themes.

The results were somewhat expected. Students tended to report time conflicts and under-estimations as excuses for using grace days. It may suggest showing students the history data of the typical completion time and when time conflicts frequently occur can be helpful.

It is worth noting that some students used grace days simply because it was the last assignment. Although it is a valid time management strategy, it may suggest there exists a certain amount of procrastination. It can harm a student's self-discipline since the statement means no other than "I decided to be late because I am allowed to be late".

Summary In sum, the main study reveals that students tended to perform better if they did not use grace days even when they were available and legitimate to use. On the contrary, using grace days reflected that students might need help in learning, potentially from two aspects, how to solve a question and how to validate their solutions. However, because students were allowed to form different groups for different assignments, it blurred the relation between using time extensions and the performance of individual students. The other studies will provide more insights into this aspect. In terms of why students used grace days, students tended to report time conflicts and under-estimations, which were somewhat expected.

4.4 The Replication Study

Method The same methods used in the main study were used here, including the approach to categorize students and statistical tests. Table 4.2 summarizes the category information for these students.

Table 4.2: Student categorizes and sizes in the replication study

category	size	description
0	87	On time
1-5	338	Used up to five grace days for the term
> 5	13	Five grace days were not enough

Results and Discussion For both experiments, the Kruskal-Wallis H -test results were statistically significant. In both Figure 4.3 and Figure 4.4, the green triangle represents the mean.

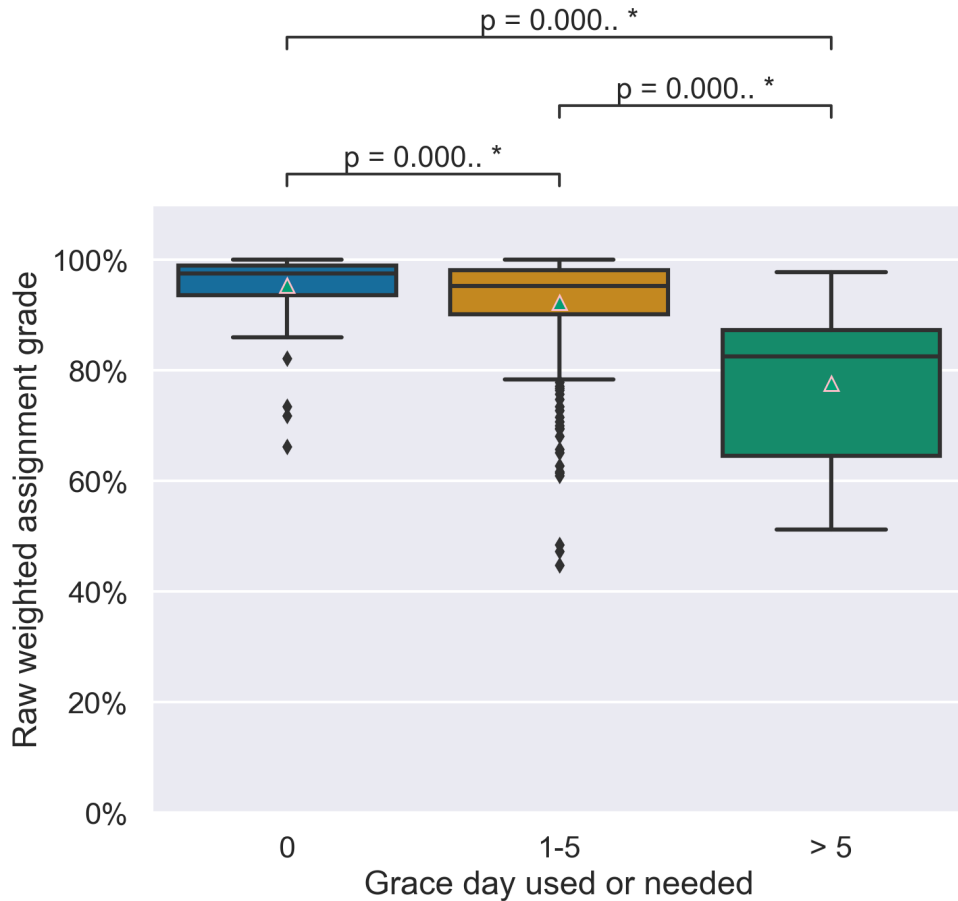


Figure 4.3: Box-plots of grace day usage and assignment performance

Figure 4.3 and Figure 4.4 from the replication study present almost the same information as Figure 4.1 and Figure 4.2 from the main study, successfully replicating the findings.

The most notable difference is that in the replication study, the difference in final exam grades between “1-5” students and “> 5” students was significant, whereas it was not significant in the main study. This is interesting since, in both the main and replication studies, “1-5” and “> 5” students showed a significant difference in their assignment grades. This may suggest that treating students who used grace days as a single group, regardless of

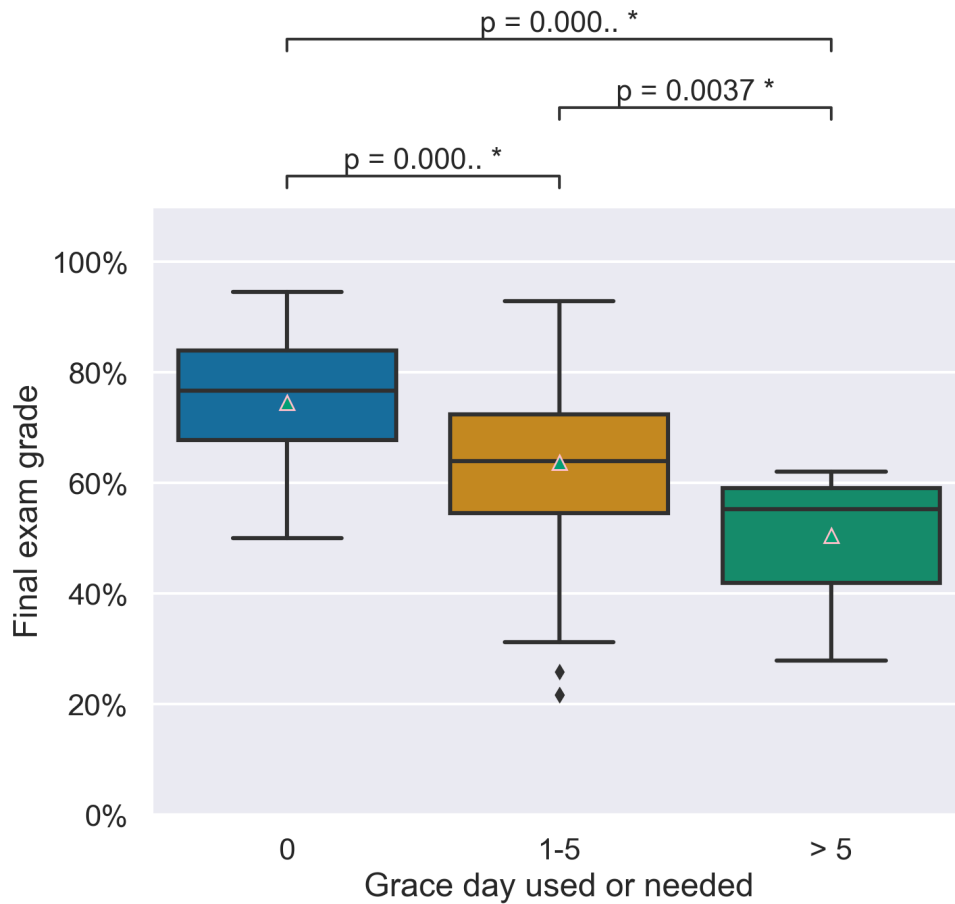


Figure 4.4: Box-plots of grace day usage and final exam performance

the amount used, might be more ideal for analyzing final exam performance in programming courses.

Another interesting difference is that, in the main study, the best final exam grade appeared in the “1-5” category, while in the replication study, it appeared in the “0” category. The phenomenon that the best final exam grade could appear in different categories suggests that for very-top-performing students, the use of grace days will not greatly affect their performance; rather, grace days provide them with the opportunity to manage their time more effectively.

4.5 The Reproduction Study

Method We split the first-year students based on whether they successfully passed all public tests during the 2-hour lab time. As mentioned in Section 4.2, if they succeed, it indicates they learned all the necessary knowledge to solve the lab question. Note that because there was only a trivial number of students (mostly one student) who did not make at least one submission before the extended evening deadline, therefore, we did not treat them separately as we did in previous studies (they were categorized as the “> 5” category in previous studies). Table 4.3 provides a summary of the categories. The total number of first-year students was 130.

Table 4.3: First-year student categorizes and sizes

category	size	description
ontime	22	Always pass all public tests during the lab time
extended	108	Passing public tests outside the lab time for at least one lab

We then analyzed students coding lab performance and final exam performance using Welch’s t-tests. The coding lab performance was defined as the total number of passed tests, and the final exam performance was the raw final exam grades in the study. Raw final exam grades were out of a maximum of 157 points.

Because intro-level programming courses often structure questions by topics, therefore, to understand whether individual lab question would impose a difference in student performance, we repeated the same analysis for each coding lab. Table 4.4 presents the number of students in each category for each coding lab. However, this time the “ontime” means a student passed all public tests in the lab time in a single lab, not all labs, and “extended” means otherwise.

In the main study, one potential reason that students did not achieve a similar-level assignment performance as top students did because they did not know how to evaluate their solutions by themselves. Therefore, in this study, we also investigated if all students could pass all *public* tests when they were granted time extensions. In other words, we wanted to understand if students could learn well without needing to evaluate their solutions by themselves.

Results and Discussion For lab performances, we found the “ontime” students passed a mean of 106.96 total tests, while “extended” students only passed 98.22 total tests on

Table 4.4: First-year student categorizes and sizes of each coding lab

	lab 2	lab 3	lab 4	lab 5	lab 6
extended	63	76	11	37	91
ontime	67	54	119	93	39

average. There was a significant difference ($p = 0.007 < 0.05$) in their lab performance between the two student categories.

Students passing all public tests would make a difference in their final exam performance. The “ontime” students tended to perform better in the final exam with an average of 117.67 points, while “extended” students tended to get a lower score with a mean of an average of 87.45 points. The difference in the final exam grades between these students was statistically significant ($p = 0.000.. < 0.05$).

Table 4.5: Average final grades of students and corresponding t-test results

lab	extended	ontime	stat	p-value
lab 2	80.10	104.73	6.28	0.000.. *
lab 3	81.85	108.20	6.75	0.000.. *
lab 4	54.09	96.37	5.93	0.000.. *
lab 5	74.39	100.12	5.83	0.000.. *
lab 6	85.18	110.58	5.85	0.000.. *

Table 4.5 shows that “ontime” and “extended” students had a significant difference in the final exam performance. In addition, such differences existed for every coding lab.

Table 4.6: Average number of all passed tests and corresponding t-test results

lab	extended	ontime	stat	p-value
lab 2	15.56	18.33	4.72	0.000.. *
lab 3	14.83	18.09	5.24	0.000.. *
lab 4	13.33	18.45	6.63	0.000.. *
lab 5	19.69	21.70	3.71	0.0003 *
lab 6	28.61	30.18	2.39	0.0185 *

Table 4.7: The “extended” students who did not pass all *public* tests

	lab 2	lab 3	lab 4	lab 5	lab 6
# students	22	25	3	9	10

Table 4.6 shows that “ontime” students consistently achieved better average coding lab performance than “extended” students. The differences were all significant ($p < 0.05$ for every lab).

Table 4.7 shows that even with time extensions, some “extended” students could not solve all public tests. It indicates that simply granting such students time extensions would not help their learning.

The results indicate that once instructors identify a student who needs a time extension, they can offer help immediately without considering the student’s behavior or performance in other labs, which greatly simplifies the step of identifying students who need assistance.

Integrating public tests in the Marmoset platform allowed students to improve their solutions along with their submissions because they could make multiple submissions and gain detailed feedback from public tests immediately. However, not all students could improve their answers even though they could see the feedback. It again supports the statement that students needed proper guidance even when evaluation metrics were provided.

Summary Both replication studies replicated the RQ1 results of the main study. Students who needed time extensions tended to perform poorly on both the coding lab and the final exam.

4.6 Chapter Summary

This chapter discusses the value of time extensions in identifying students’ abilities and provides essential knowledge of time extensions to both educators and learners. Students who used time extensions tended to perform significantly worse than students who did not use them. Time conflicts and underestimation of the coursework were the top two reasons for using time extensions.

Having a reliable indicator of when instructors should take actions to assist students is crucial; however, *how* to help them remains a challenge. We will present our ideas on addressing this problem, particularly in an automated manner, in the following chapter.

Chapter 5

A Guideline for Composing High-Quality Automated Assessments

Chapter 2 discussed the benefits of automated assessments and how they are incorporated into courses to provide automated feedback. This is an essential part of programming courses [67, 42, 55] in many places nowadays. Instructors expect students to understand what is correct and what is not based on the test outcomes. Automated assessments play a role that takes the student’s work as input and produces automated feedback as output. The automated feedback is mostly categorized as knowledge about mistakes feedback and knowledge about how to proceed feedback. As mentioned in Chapter 2, expert-authored feedback can fall into any of these categories, while data-driven feedback is mostly categorized as knowledge about how to proceed feedback.

Automated assessments have greatly enhanced teaching in programming courses, allowing instructors to deliver timely feedback at scale. However, they come with several limitations. Previous chapters have highlighted a common but crucial observation with current automated assessments: students who made the most submissions were typically not those who achieved the best assignment performance (Chapter 3 Figure 3.1 and Chapter 4 Table 4.6). Moreover, allowing students more time to interact with such assessments did not appear to improve outcomes—there were still students who could not pass all public tests with additional time, despite receiving immediate automated feedback (Chapter 4 Table 4.7).

One potential reason is that automated assessments are designed not to identify miscon-

ceptions but to validate correct solutions. If a solution passes all tests, it is deemed correct; however, a solution that fails does not necessarily indicate incorrectness. This misunderstanding leads assessment creators to develop expert-authored feedback that they believe is effective, but which can actually be misleading. Additionally, if pre-coded canonical solutions do not cover all possible correct solutions, students may struggle with their correct solutions being marked as incorrect simply because these solutions do not match the pre-coded canonical solutions.

Another issue is that the creation of automated assessments is often seen as a one-time process, after which the assessments are expected to function fully autonomously without human intervention. This mindset is evident from the fact that developers often neglect to write regression tests—or “tests for tests”—for their automated assessments. However, creating flawless automated assessments is rare, given the complexity of what they need to assess. Automated assessments are essentially programs; creating bug-free programs is inherently challenging. If test failures occur due to bugs in the assessment code rather than errors in students’ work, they can mislead students into believing the fault lies within their work, leading them to waste time addressing non-existent issues.

There are more problems that need to be addressed than those mentioned above. In this chapter, we define the terms to refer to these problems and propose a novel guidance that greatly diverges from the existing build-and-test strategy for creating automated assessments. We have also developed a tool designed to assist assessment creators in following our novel guidance. This tool is publicly accessible¹.

5.1 Related Work

Given the tremendous benefits of automated assessments and automated feedback, various automated tools, platforms, and systems have been developed. Blanchard et al. [18] published a literature review in 2022, noting that instructors and researchers often resort to developing in-house solutions, essentially reinventing the wheel generation after generation. Therefore, a literature review was deemed highly necessary. Although they did not distinguish between automated tools, platforms, and systems as we do in this thesis, they provided a comprehensive summary. Based on that, we examined several tools they summarized and found it rare for people to mention how they addressed incorrectness in the associated automated assessments or automated feedback.

¹<https://github.com/h365chen/socassess>, accessed: 2024-Sep-15

To our knowledge, only CSF² [54, 55] explicitly outlines how the quality of automated feedback can be enhanced. The steps in the CSF² framework generally form an iterative process to create high-quality hints connected to concepts and skills. A crucial component of the CSF² framework is grouping submissions into “buckets” based on their test outcome signatures. A signature is the concatenation of the binary pass or fail outputs by all the test cases in the test suite. New test cases are then added to distinguish between subsets of submissions within the same bucket but with varying logical errors upon manual inspection. This process concludes when all submissions in a bucket exhibit identical logical errors. Misconceptions identified using the data from a prior semester lead to enhanced automated feedback for the following semester.

Data-driven feedback may take into account feedback enhancement; however, it is often misleading. For example, the automated system MISTAKEBROWSER [57] allows instructors to annotate hints found via typical hint generation techniques [92, 91, 89], where hints are in the form of code transformations such as code edits. The mechanism for a hint to be a valid hint is whether the number of passed tests can increase after applying those code edits. However, it does not address incorrectness in those referenced tests.

In fact, data-driven feedback is sometimes more difficult to control than expert-authored feedback since it heavily relies on the underlying data. The same statement applies to feedback from Large Language Models (LLMs) as well, since their feedback is also data-driven and the mechanism is non-deterministic.

Of course, human support tools do not have such problems since they themselves do not provide any feedback, but require humans to compose the feedback manually on demand [114]. However, this chapter will not consider human support tools since they cannot possibly eliminate human efforts completely as they do not provide any support for composing automated assessments. We target fully automated assessments despite recognizing several problems in the current mechanisms for composing them.

Gusukuma et al. [53] proposed an approach to provide misconception-driven feedback. Although they did not mention how their feedback gets improved, their terminologies are useful and will be used in this thesis. In their approach, *mistakes* are the observed performance of the student and a mistake is associated with a vector of *misconceptions* and is the underlying incorrect belief or understanding that leads to those mistakes. Misconceptions can be isolated by cross-referencing multiple mistakes.

5.2 Problems

The problems outlined here are considered most essential based on our experience.

Existing literature often fails to distinguish between automated assessments and automated feedback. However, it is crucial to recognize that automated feedback typically derives from the outcomes of automated assessments, *i.e.*, test results. Additionally, there is a lack of differentiation between knowledge about how to proceed (KH) feedback and knowledge about mistakes (KM) feedback, with the latter often being a direct translation of the test outcomes from automated assessments. From the perspective of an assessment creator, a closely related question arises: how can one effectively provide automated KH feedback if accurate automated KM feedback cannot be reliably generated? This dilemma underscores a significant flaw in the current framework for constructing automated assessments. We propose that a valid approach is to acknowledge that automated feedback inherently *depends* on the precision of the outcomes of automated assessments. An automated assessment must first guarantee the accuracy of KM feedback before attempting to offer other types of feedback.



Figure 5.1: Automated Assessments, KM Feedback, and KH Feedback

Two primary types of errors commonly occur when generating KM feedback: 1) incorrect components are erroneously marked as correct; 2) correct components are marked as incorrect. This chapter will discuss why detecting the former is challenging, and will primarily focus on addressing the issues related to the latter. While our examples predominantly come from database courses, the strategies discussed here are applicable to other courses as well.

5.2.1 Incorrect components marked as correct

When an automated assessment is deployed, the assessment creator expects it to function correctly. Consequently, if a solution is marked as correct, the creator generally accepts this

outcome without further inspection, as there is little incentive to retrospectively examine the assessment code. Students are neither likely to report incorrect components being marked as correct, which can be compared to a scenario where students would need to request point deductions after realizing they received a higher grade than deserved.

Conversely, assessment creators become more aware of potential defects in the code when it leads to correct components being marked as incorrect, as students typically express their dissatisfaction when correct solutions are erroneously marked as incorrect.

Detecting incorrect components marked as correct is challenging. One potential strategy to address this issue is to periodically sample a subset of student solutions for human inspection, ensuring a higher level of accuracy and reliability in the automated assessment. However, we believe addressing the next two problems can effectively reduce the likelihood of this problem occurring.

5.2.2 Correct components marked as incorrect

A test within an automated assessment may fail for numerous reasons that extend well beyond the expectations of the assessment creator. For instance, to assess 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 and then compare the results. If the results do not match, the test is considered to have failed. However, a mismatch in results is not the only reason for a test failure. The test could also fail due to a connection issue, where the assessment code is unable to connect to the database, or it might fail due to insufficient memory to handle the returned result, among other potential issues. The critical point here is that constructing assessment code that fails only for predefined reasons is often more challenging than anticipated. Students are likely to be confused and frustrated if the test indicates failure when they believe their submission is free of errors.

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

The situation becomes more complex with multiple tests. While many assessment creators adopt the unit testing philosophy, where each test is considered isolated from others, dependencies among tests are common. For instance, 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 successfully compiles. Furthermore, if a student's program relies on external libraries or other modules, the corresponding tests are subject to the presence and proper configuration of these libraries. Current build-and-test strategy only considers the dependency between compilation and

execution, but it overlooks a variety of other potential dependencies, thereby limiting their effectiveness in accurately evaluating student submissions.

Table 5.1 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 (*e.g.*, 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 5.1: Test outcomes

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

5.2.3 Improving existing assessments

Assuming that an assessment creator 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. However, 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 monotonically.

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

5.3 Methods

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

Assessment Code This refers to the essential component of an automated assessment. 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.

Quality Assurance (QA) on Assessment Code This ensures that the introduction of new code does not bring bugs or inconsistencies. It addresses the Assessment Iteration problem.

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 QA 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.

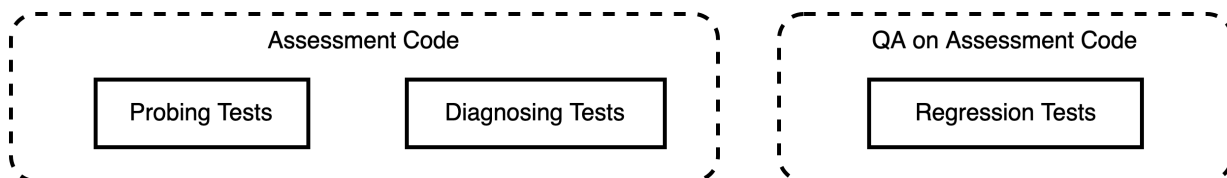


Figure 5.2: Relationships between test and assessment categories

Figure 5.2 shows the relationships among them. In summary, the assessment code contains probing and diagnosing tests, while QA on assessment code contains regression tests. We will use these terminologies throughout the remainder of the thesis.

5.3.1 Eliminating correct components being marked as incorrect

To address correct components being marked as incorrect, we must tackle Unexpected Failure and Missing Dependency problems.

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 (Listing 4).

```
1     # If one passed and the other failed
2     # we know both assertion lines were reached
3     def test_a():
4         x = common_code()
5         assert x == 0
6
7     def test_a_flipped():
8         x = common_code()
9         assert x != 0
```

Listing 4: Ensure failure is caused by an assertion statement

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 (Listing 5).

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. Nonetheless, there is a trade-off between the granularity of assertions and the amount of uncertainties.

Then, how do we handle unexpected failures? The most appropriate approach is to inform a human to inspect. Once the cause is identified, human feedback can be provided

```

1  def test_a():
2      try:
3          x = may_cause_unexpected_failure()
4          assert x == 0
5      except AssertionError as e: # known failure
6          raise ExpectedError from e
7      except Exception as e: # capture general case
8          raise UnexpectedError from e

```

Listing 5: Ensure failure is caused by an assertion statement (alternative)

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 5.3.



Figure 5.3: Handling unexpected failures

At this point, the Unexpected Failure problem is handled properly.

Regarding the Missing Dependency issue, upon closer examination, it is not truly an implementation problem. Instead, it is a problem that assessment creators often fail to acknowledge the existence of dependencies. The reason is that many assessment creators and developers of automated systems adhere to the unit testing philosophy, which advocates for isolated tests when designing probing tests. However, isolation may not be appropriate

in automated assessments within the educational context. This issue can be addressed if assessment creators recognize and consider dependencies when composing probing tests.

5.3.2 Adding new code safely

Standard test-driven development (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 5.1) for the solution should remain unchanged. In other words, if the input is the same 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.

For example, we assume the database used for an assignment is the **world** database², and we have loaded it into a MySQL database. 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 three steps:

1. Execute both the student's and the instructor's query to retrieve table rows;
2. Sort the rows;
3. Compare the sorted rows.

Below is a sample function which could be used in the probing test (Listing 6):

²<https://dev.mysql.com/doc/world-setup/en/world-setup-installation.html>, accessed: 2024-Aug-14

```

1  def check_correctness(stu_sql):
2      inst_sql = "SELECT DISTINCT `name` FROM `city`;"
3      inst_rows = get_rows(inst_sql)
4      inst_rows_sorted = sorted(inst_rows)
5      # Similarly, for the student's query
6      stu_rows = get_rows(stu_sql)
7      stu_rows_sorted = sorted(stu_rows)
8      # Perform the comparison
9      return student_rows_sorted == inst_rows_sorted

```

Listing 6: Example probing test

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 (Listing 7).

```

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

```

Listing 7: Example regression test

However, later we find it might be clearer to use `ORDER BY` within `inst_sql`. Therefore, we updated the `check_correctness` code as follows (Listing 8):

```

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

```

Listing 8: Updated probing test

This change would flip the outcome of `test_ensure_distinct` to fail because `sorted()` in Python sorts rows differently from `ORDER BY` in SQL by default, as illustrated by Table 5.2. Despite not being its original intention, the regression test detected a potential discrepancy in our assessment code.

Table 5.2: Differences in sorting results between SQL’s `ORDER BY` and Python’s `sorted()`

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

5.3.3 Providing high-quality feedback

Understanding why a probing test fails is important. However, even more crucial is informing students how they can improve [53]. 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 composed 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.

Table 5.3 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 5.1 to Table 5.3 marks a notably improvement in the precision of automated feedback. Nonetheless, we argue it is necessary to transform raw test

Table 5.3: Test outcomes of probing tests and diagnosing tests considering dependencies

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

outcomes into actionable feedback. For example, the feedback derived from Table 5.3 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.

The actual implementation was making use of a popular unit test framework **pytest** [68]. Although it does not natively support test dependencies, this functionality can be achieved with the **pytest-dependency** plugin³. Despite the assessment code being written in Python, there is no inherent connection or limitation to what it can assess. Listing 9 and Listing 10 shows an example automated assessment. Note that in the implementation, we allow a high level feedback to hide low level feedback, although we are not sure yet whether it should be part of the aforementioned guideline. In this case, if `test_query_missing_distinct` passes, then only the feedback “*Oops! Your query returns incorrect results. Maybe double check if you have used DISTINCT?*” will be shown. A more complicated assessment can be found in Appendix C.

³<https://pypi.org/project/pytest-dependency/>, accessed: 2024-Aug-14

```
1 def test_exist(stu_answer):
2     assert stu_answer.exists()
3
4 @pytest.mark.dependency(depends=['test_exist'])
5 def test_content(stu_answer_content):
6     """Test content only if the solution file exists."""
7     assert len(stu_answer_content) > 0
8
9 def test_query():
10    ...
11
12 def test_query_flipped():
13    ...
14
15 @pytest.mark.dependency(depends=['test_query_flipped'])
16 def test_query_missing_distinct():
17    ...
```

Listing 9: Example assessment code

```

1   # Ensure the student's solution is not empty
2   frozenset([
3       'test_it::test_exist::failed',
4   ]): {
5       'feedback': 'Oops! Have you named the file correctly?',
6   },
7   frozenset([
8       'test_it::test_content::passed',
9   ]): {
10      'feedback': 'Nice! You have put some content in the solution!',
11  },
12  frozenset([
13      'test_it::test_content::failed',
14  ]): {
15      'feedback': 'Oops! Your solution seems to be empty.',
16  },
17  # Evaluate the student's SQL query
18  frozenset([
19      'test_it::test_query::passed',
20  ]): {
21      'feedback': 'Nice! Your query returns correct results.'
22  },
23  frozenset([
24      'test_it::test_query::failed',
25      'test_it::test_query_flipped::passed',
26  ]): {
27      'feedback': 'Oops! Your query returns incorrect results. Remember you can
28      ↪ always contact the instructor team if you have spent too much time figuring
29      ↪ it out on your own.'
30  },
31  frozenset([
32      'test_it::test_query::failed',
33      'test_it::test_query_flipped::passed',
34      'test_it::test_query_missing_distinct::passed',
35  ]): {
36      'feedback': 'Oops! Your query returns incorrect results. Maybe double check if
37      ↪ you have used DISTINCT?',
38      'level': FeedbackLevel.HIGH
39  },

```

Listing 10: Example automated feedback

5.4 Chapter Summary

This chapter addresses three critical problems that are often overlooked by developers of automated assessments: how to prevent incorrect components from being marked as correct, how to ensure that correct components are not marked as incorrect, and how to enhance existing assessments without introducing bugs or feedback inconsistencies.

To tackle these problems, we have defined terminologies for two categories of code: Assessment Code and Quality Assurance (QA) on Assessment Code; as well as three categories of tests: Probing Tests, Diagnosing Tests, and Regression Tests. The Assessment Code encompasses both Probing and Diagnosing Tests, while QA on Assessment Code involves Regression Tests. We also provide a guideline to organize them so that the second and third problems can be effectively addressed, thereby reducing the likelihood of the first problem occurring. As a result, accurate and effective high-quality automated feedback can be provided.

Chapter 6

Evaluation on the Novel Guideline

To understand the effectiveness of the automated assessments composed following aforementioned novel guideline, we conducted an evaluation of an entity relationship (ER) modeling question. We investigated the error rate of automated feedback on students' submissions, analyzed the students' learning analytics related to the question, and examined their performance on the final exam.

6.1 The Entity Relationship Modelling Question

This was conducted in an introductory course on database systems. The course has been offered for many years. For this evaluation, we examine two offerings, one with and one without automated assessment.

The same instructor taught both offerings. There was no midterm exam, but there was a final exam, conducted in-person on campus. There were several homework assignments. The number of assignments was four and eight separately. Although the number of assignments varied, the individual questions in the assignments were identical. For example, the third assignment in the offering without automated assessment contained three questions: correcting an incorrect ER model; translating from the ER model to relational schema; and a normalization question regarding functional dependencies. Each question became a separate assignment in the offering with automated assessment, resulting in three assignments instead of one. The automated feedback was set up for correcting the incorrect ER model.

There was no difference between the offerings for the ER modeling question. Students in both offerings received the exact same ER modeling question in their homework assignment.

The ER modeling question in the final exams, which was identical for both offerings, was based on a different domain from that assessed in the homework assignment. The first sub-question Q1 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 Q2 asked students to compose new relationship sets and/or specializations to enhance the existing ER design.

6.1.1 The homework ER modeling question

The ER question was the same for both offerings. The question provided a detailed background of a draft ER model for a given database, which was pre-loaded with a real-world dataset (over 1.5GB in size) from Kaggle¹. It then explained to students why the draft ER model was flawed, *i.e.*, failing to capture several real relationships among entities. For example, the draft ER model contained essentially one giant relationship set involving all entity sets. However, it turned out that some entity sets should only have relationships with another under certain conditions, indicating specializations.

To briefly illustrate the complexity of the question, a well designed solution should include at least 10 entity sets (of which 1 should be weak), 3 relationship sets, 4 roles, 5 specializations, and 20 attributes. Given this complexity, there were only two valid designs, each with trade-offs compared to the other.

6.1.2 ER encoding

The notations taught in the course were those of extended ER notations (EER) [95]. However, instead of allowing students to freely draw diagrams, we required them to use a specific encoding method by inserting values into some pre-defined relational tables. This approach was adopted even before the introduction of the automated assessment.

The primary reason for this method is that, anecdotally, we found reading hand-drawn diagrams to be painful and time-consuming. While some students used UML tools to make their diagrams clearer, the notations in these tools differed from those taught in the course. This discrepancy created a cognitive burden for both graders and students, since they had to familiarize themselves with these notations before grading or drawing the diagrams.

¹<https://www.kaggle.com>, accessed: 2024-Aug-14

However, we did not want to use UML notations directly, either, since they include more features than necessary for ER models alone [106].

In our ER encoding, students needed to write a small number of insertion statements, and their diagrams were represented by the resulting MySQL tables. Since students had already learned SQL by that time, this method did not impose much additional cognitive load. An example of this ER encoding is provided in Figure 6.1.

The encoding currently supports all EER notations, although the encoding of certain notations, such as weak entity sets and composite attributes, is not demonstrated. We did not observe a large number of questions about how to use the encoding, suggesting that it is relatively easy to learn. Students were not required to use the encoding in the final exam; they could hand-draw it instead.

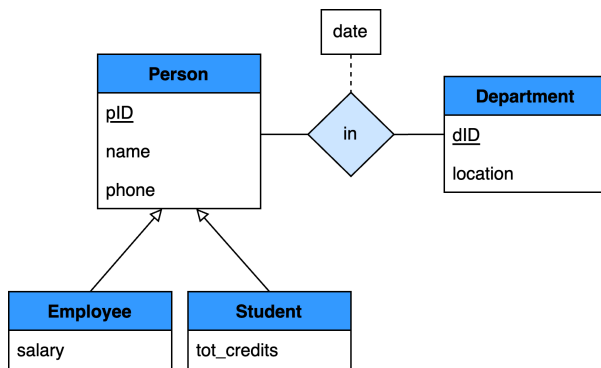
6.1.3 The automated assessment

Students submitted their solutions in the form of an executable SQL file, containing the insertion statements. Then, some preparations were necessary for the automated assessment:

1. A set of valid solutions, ideally encompassing all possible valid solutions.
2. The essential entity sets that are presented in all solutions in the solution set need to be identified. These entity sets are so called **anchor entity sets**. The overall layout of an ER model is greatly influenced by its relationship sets between anchor entity sets.
3. Attributes are known and involving entity sets are known for most of the attributes.

The automated assessment was hosted on Marmoset. The entire automated assessment was configured as a single test on Marmoset (for more details, see Appendix B). The phases of the automated assessment were as follows:

1. Executing the submitted SQL file, ensuring the SQL file is executable. If it was not executable, the feedback message would show student the execution error.
2. Searching for anchor entity sets. If any anchor entity sets were missing, students were advised to re-think their design. We also reminded students of the names of anchor



(a) ER model

EntitySets

esName	primaryKey
Person	pID
Department	dID

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

RelationshipAttributes

rsAttrName	rsName
date	in

IsA

specializedES	generalizedES	disjoint	total
Employee	Person	0	0
Student	Person	0	0

(b) Encoding

Figure 6.1: Sample ER model and its encoding

entity sets in case it was just a typo issue. This information would not leak much hint to students since the ER modeling question essentially was asking students how to connect provided entity sets, most of which were already known from the assignment description.

3. If all anchor entity sets were present, the design would then be checked against pre-defined rules. At this phase, we examine if students had pieces in their design that are incorrect.
4. The closest exemplary design was also needed. Each exemplary design had its own set of rules. Given a student's design, if all rules of an exemplary design passed, the layout of the design was determined. If no such exemplary design could be found, students were advised to have a conversation with the instructor team. Although it was likely the student's design to be a flawed one, it was possible that the instructor team failed to consider all possible valid designs in advance. In fact, the automated assessment was capable of sending an email to the instructor team whenever a design failed to match any of the exemplary designs. However, this approach resulted in an overwhelming number of emails, so it was only used for the very first few submissions. Instead, the automated feedback would encourage students to post their questions on Piazza, an online learning and discussion forum, to seek help.
5. Once the overall design was determined, other parts were examined by comparing the student's design against the specific exemplary solution, such as checking attributes.

Although students were required to use pre-defined names for entity sets, they were not required to do so for relationship sets, since relationship sets could be identified by the involved entity sets, even if their names differed. For example, two relationship sets from two designs could have different names, but if the same entity sets were involved, then they were considered the same relationship set.

From our perspective, it may not be necessary to enforce strict dependencies among different phases of the assessment. For example, in our assessment, we checked for specific attributes within a few entity sets before determining the overall design—*i.e.*, relationship sets and specializations—because we were certain of the possible correct ways to model them. Nonetheless, we encouraged students to contact the instructor team at any time if they encountered difficulties or had confusions at any phase. We customized Marmoset to display a single feedback message and Figure 6.2 shows part of the feedback with the entity set names modified.

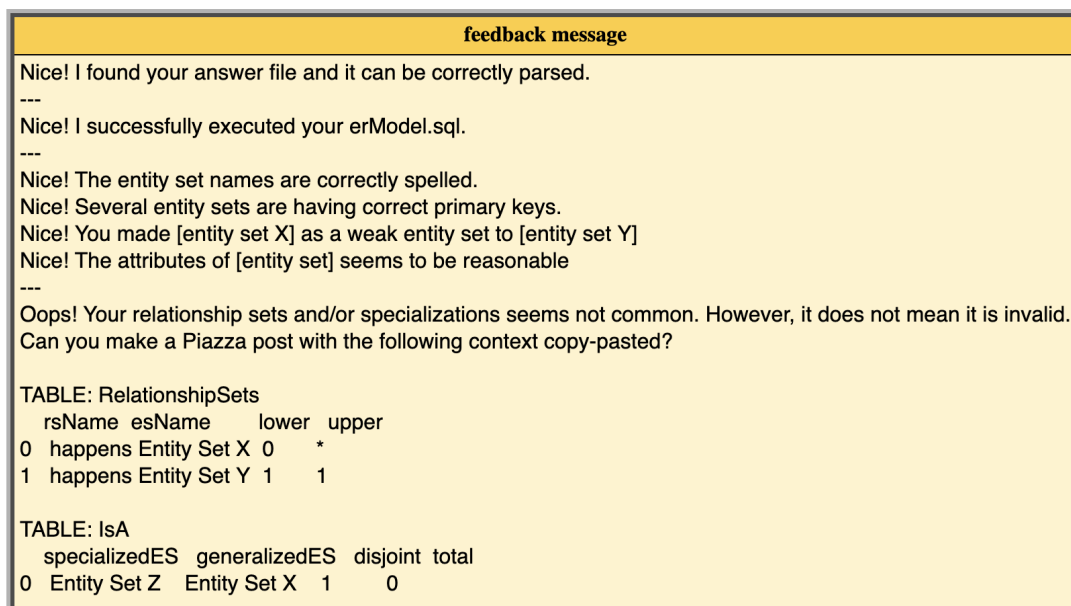


Figure 6.2: Example feedback screenshot (partial)

In addition to positive exemplary designs, negative exemplary designs could also be coded. Then a pre-coded actionable feedback could be provided.

Students received feedback messages from the automated assessment, usually in the form of questions encouraging them to rethink parts of their design. However, if a design is mostly perfect, the feedback message will be “*Nice! Your design seems to be reasonable on [certain parts]*” instead of a score. It was because we did not want students to have the notion that getting perfect feedback messages means getting a full mark on the question. We also would like to have them further re-fine their designs even if the assessment passed, meanwhile due to limited time in assessment development, we had to left certain parts unchecked such as the cardinality constraints.

Figure 6.3 shows an example of the evaluation process. The anchor entity sets are **Instructor**, **Student**, and **Project**, and the matching rules are: for the design of Figure 6.3a, there should be three relationship sets in total, and each involves two anchor entity sets; for the design of Figure 6.3b, there should be one relationship set in total, and it involves all the three anchor entity sets. Cardinality constraints are not presented. Based on the rules, the student design of Figure 6.3c will match the design of Figure 6.3b even though the relationship set is named differently, and the student design of Figure 6.3d will not match either of the exemplary designs because the total number of relationship

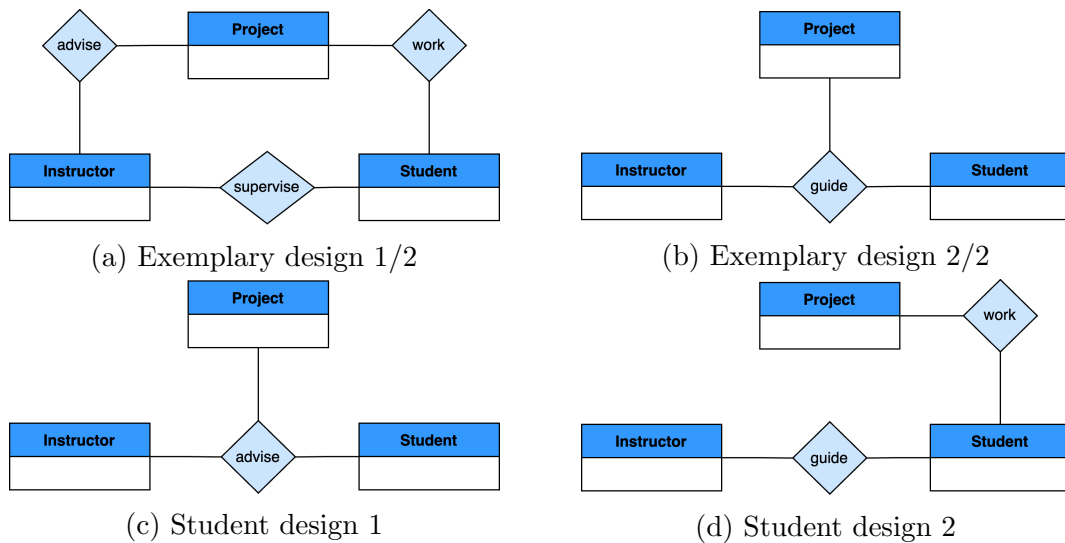


Figure 6.3: Two example exemplary designs and two example student designs

sets is different.

6.1.4 Automated feedback

We present the data collected from the semester during which the automated assessment was deployed. As discussed in Section 6.1.3, to receive automated feedback on ER designs, students needed to ensure their SQL file was executable. An example of feedback provided for a non-executable SQL file is shown in Listing 11. Essentially, it contains the raw error message from the MySQL client.

```
I ran into an error executing your erModel.sql.
...

You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to use near ...
...
```

Listing 11: Example feedback for non-executable SQL file

Table 6.1: Feedback categories for the ER question

Category	Phase	Students	Example															
-	1	147	I ran into an error executing your SQL file: [SQL error message]															
A	2	49	I expect entity set names from the list: [list of entity set names]															
B	2	136	Nice! The entity set names are correctly spelled															
C	3	59	It seems the relationship set around [entity set X] and [entity set Y] are not correct. I would expect [entity set X] to be a weak entity set. Also, [entity set Y] should be involved in at most one relationship set.															
D	3	14	<p>It seems some of the important entity sets I am looking at are not correct.</p> <p>Expected important entity sets:</p> <table style="margin-left: 40px;"> <tr> <td></td> <td>esName</td> <td>primaryKey</td> </tr> <tr> <td>0</td> <td>entity set X</td> <td>pkX</td> </tr> <tr> <td>1</td> <td>entity set Y</td> <td>pkY</td> </tr> </table> <p>But I got them as:</p> <table style="margin-left: 40px;"> <tr> <td></td> <td>esName</td> <td>primaryKey</td> </tr> <tr> <td>0</td> <td>entity set X</td> <td>pkX</td> </tr> </table>		esName	primaryKey	0	entity set X	pkX	1	entity set Y	pkY		esName	primaryKey	0	entity set X	pkX
	esName	primaryKey																
0	entity set X	pkX																
1	entity set Y	pkY																
	esName	primaryKey																
0	entity set X	pkX																
E	3	124	Nice! You made [entity set X] as a weak entity set to [entity set Y]															
F	3	138	Nice! Several of them are having correct primary keys															
G	4	28	For specializations of [a specific entity set], it seems you have [entity set X] as a specialized entity set of [entity set Y], and [entity set Z] as a specialized entity set of [entity set X]. However, you don't have any relationship set around the [entity set X]. Do you really need the [entity set X]?															
H	4	91	<p>Your relationship sets and/or specializations seems not common. However, it does not mean it is invalid. Can you make a Piazza post with the following context copy-pasted?</p> <p>[The encoding of student's design]</p>															
I	4	58	Nice! It seems your relationship sets and specializations are reasonable between [some entity sets]															

Continuation of Table 6.1			
Category	Phase	Students	Example
J	5	89	You might want to move [attribute] elsewhere outside [entity set].
K	5	79	The attributes in [entity set] do not match my expectation.
L	5	115	Nice! The attributes of [entity set] seems to be reasonable

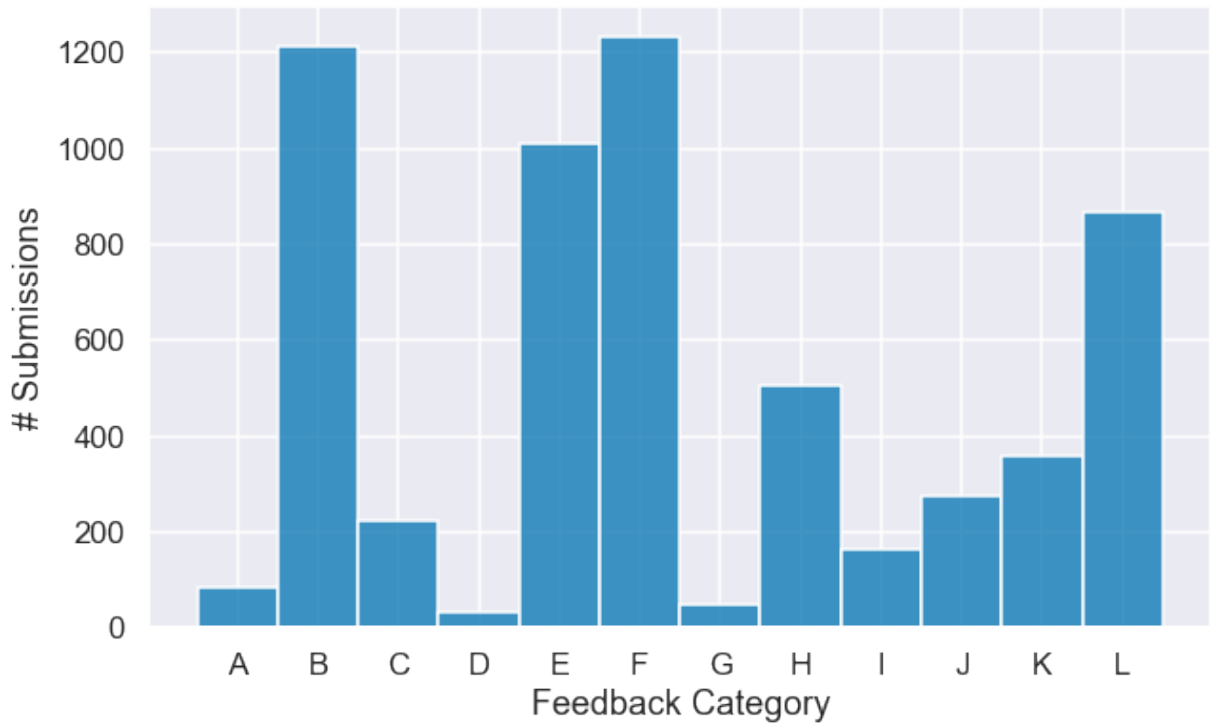


Figure 6.4: Feedback category and the number of submissions

Table 6.1 shows the automated feedback categories and their examples. The total number of involved submissions was 1615 but the same submission could receive feedback from multiple feedback categories. Figure 6.4 shows the histogram of the number of submissions receiving feedback.

6.2 Error rate of automated feedback

In order to thoroughly examine the error rate of each feedback category, we decided to conduct a manual examination over students' submissions. This is a labour intensive task.

Since the total number of cases needing manual examination is too large (the number of feedback categories multiplies the number of submissions, which are $11 \times 1615 = 17765$ cases), therefore, we decided to use disproportionate stratified sampling to reduce the workload. It was disproportionate because we did not consider every submission contributes equally to the error rate; instead, we consider every student contributes equally. As a result, the sample set was created by randomly selecting 2 submissions from every student. There were in total 296 submissions in the sample set.

Figure 6.5 presents a histogram of the number of submissions that received feedback for the sample set. This histogram closely resembles that in Figure 6.4, thereby confirming the correctness of the sampling process. Figure 6.6 displays the error rates for the sample set.

We can observe that, the top three *negative* feedback categories which provided feedback messages informing students of incorrectness in their solutions, *H*, *J*, and *K*, had no errors. While it appeared that there were some inaccuracies for feedback categories *C*; however, we found that the only inaccurate case of *C* was because of a small mismatching between the expert-authored feedback and the assessment code. The more accurate feedback message should be: *Both [entity set X] and [entity set Y] should be involved in at most one relationship set.* There were few error cases for *D*. In the end, it turned out those errors were because of a logic mistake in the assertion statement. As mentioned in Chapter 5.3.1, there is a trade-off between the granularity of assertions and the amount of uncertainties. There were also few error cases for *G*. Those errors were because of an oversight in specifying the design comparison rules in the diagnosing test.

For *positive* feedback which inform students of their correctness, categories *B*, *E*, *F*, and *L*—despite large number of submissions—had no errors, indicating the associated assessment code were well written. Similar to *G*, the error of *I* was also due to an oversight in specifying the design comparison rules. It appears there was a code reuse in the assessment code for *G* and *I*, thereby they were both lack of the same design comparison rule. We did not realize it until we conducted the manual examination.

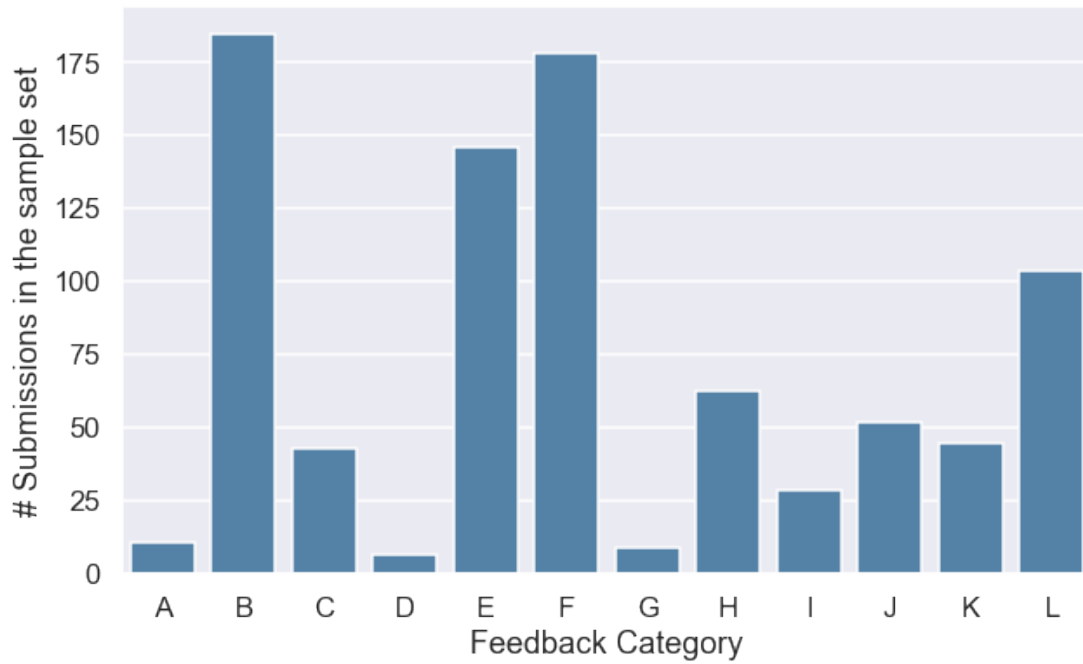


Figure 6.5: Feedback category and the number of submissions in the sample set

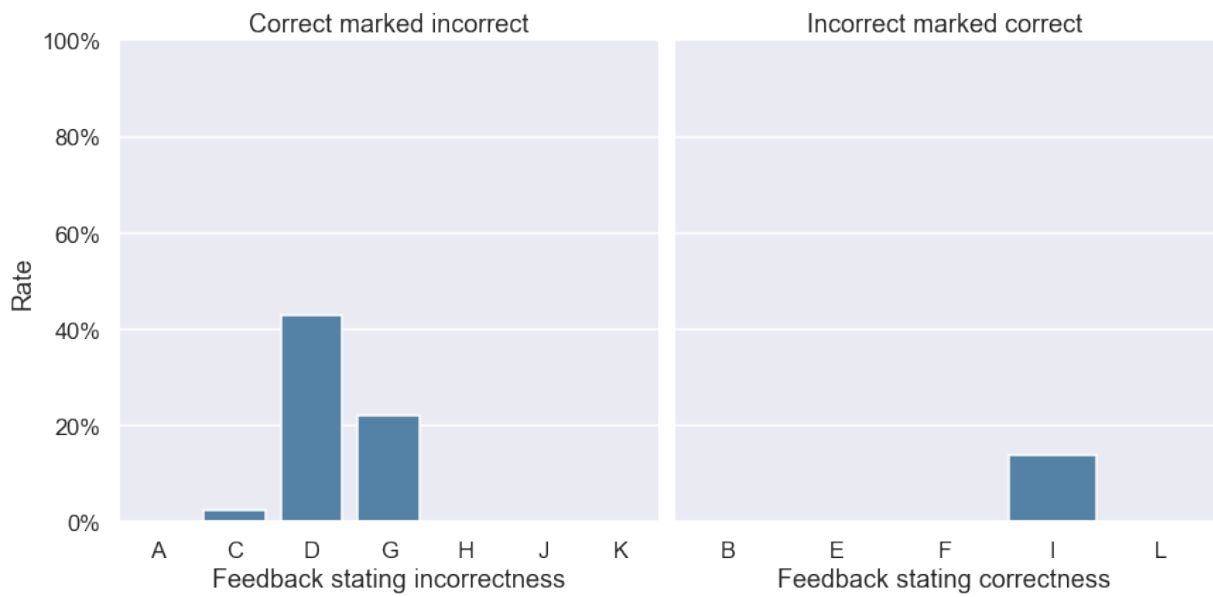


Figure 6.6: Error rate of provided automated feedback

The most common transition is from vertex $BEFHL$ to itself, with 23.4% as the weight. Given B , E , F , and L are positive feedback, it is expected since the only negative feedback H informs students that although the automated assessment checked most parts of their design and they seem satisfactory, certain aspects, particularly the relationship sets and specializations in their design, may require human inspection. Although we recommend students to contact the instructor, students chose to make several more attempts while waiting for human feedback. Those submissions typically ends up with the same feedback.

The second most common transition is from $BEFK$ to itself, with 10.8% as the weight. The third most common transition is from J to itself, with 9.2% as the weight. If we only consider negative feedback, *i.e.*, excluding positive feedback from B , E , and F , it suggests negative feedback in K (10.8%) and J (9.2%) were not easily solvable. Since they are all knowledge about mistakes feedback, it suggest that knowledge about mistakes feedback is sometimes very difficult to resolve; students need more detailed explanation or accompanied knowledge about how to proceed feedback.

Additional to aforementioned, transitions between $BEFIL$ to itself is also common, with a weight as 5.7%. Although $BEFIL$ suggests the design is reasonably good (all B , E , F , I , and L are positive feedback), since students had the notion that the assessment code only assessed their designs partially, thus, some of them would made additional submissions to reassure modifications they made afterwards did not introduce unexpected issues invalidating the $BEFIL$ feedback.

Last, there is an interesting observation: there is a node labelled BEF . Since B , E and F are all positive feedback, it indicates the associated feedback did not point out any issues within students' submissions. However, there must be something wrong since otherwise the feedback should fall into $BEFIL$. It suggests there were some unexpected failures. After retrospective investigation, we confirmed it was an unexpected failure, otherwise it should have received feedback in $BEFK$. Although an unexpected failure occurred, the students were not misled about the nature of the issue. Instead, they received a message indicating that feedback for some of their components was unavailable, which aligned precisely with our expectations under the new guideline.

6.4 Final exam performance

Since summative assessments are better at capturing the comprehensive understanding of students, therefore, we investigate students' final exam performance on the ER question between the two offerings without and with the automated assessment to understand if the automated feedback has improved students' learning.

To conduct a fair comparison, we will use the grades from unrelated questions in the final exams to assess potential shifts in students’ abilities. Since the SQL questions in both exams were mostly the same with minor variance, therefore, the grades from those questions will be used to examine students’ overall abilities.

Students’ sums of the SQL questions are converted to grade percentages between 0% and 100%. For example, if a student achieved 10 out of 20 points, the corresponding grade percentage will be $\frac{10}{20} \times 100\% = 50\%$.

The comparison is conducted by performing a Kruskal–Wallis H -test between students’ grade percentages from both offerings. The null hypothesis is that students from both offerings had no difference in the ranks of their SQL performances. The threshold is 0.05, meaning that if the p-value is smaller than 0.05, then we will reject the null hypothesis and accept the alternative hypothesis that students from one offering had better ability than those in the other offering. If the grade percentages from the two offerings do not have a significant difference, then we consider no adjustment is needed. Otherwise, if the percentages from one offering significantly differ from the other, then we will adjust for such difference before comparing their performances on the ER modeling question.

Following the above methodology, we found the p-value of the Kruskal–Wallis H -test between students’ grade percentages is 0.792, which is far more larger than 0.05. Therefore, we do not reject the null hypothesis. This suggests that the abilities of students from both offerings were comparable. Consequently, no adjustment was necessary before comparing their performances on the ER modeling question.

Having adjusted the students’ abilities, the comparison on the ER modeling question will also be done using the Kruskal–Wallis H -test, between students’ grade percentages.

Table 6.2: Average performance on the final exam, and p-value

Question	without feedback	with feedback	p-value
SQL	72%	72%	0.792
ER-Q1 Identify PKs, weak entity sets, constraints, etc.	63%	60%	0.496
ER-Q2 Create relationship sets and specializations	39%	52%	0.016

For Q1, the p-value was 0.496, significantly above 0.05, indicating no substantial difference in student performance. Conversely, for Q2, the p-value was 0.016, well below 0.05, suggesting a significant difference in performance. The average scores for Q1 were 60%

with automated assessments and 63% without them. For Q2, the averages were 52% with automated assessments and 39% without, representing a 33% improvement on Q2.

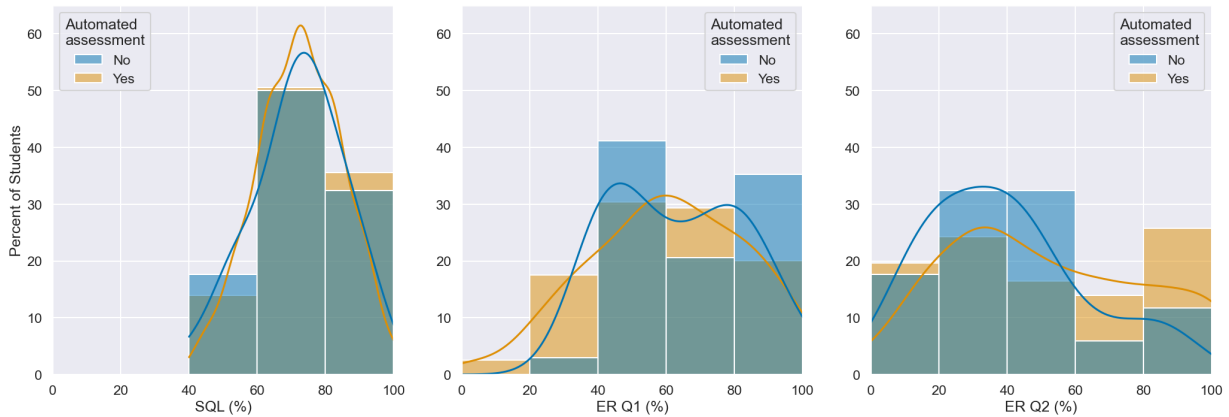


Figure 6.8: Grade percentage (%) on SQL, Q1, and Q2 in the final exam

Figure 6.8 shows the distributions of students' grade percentages. We can see that the distribution for the SQL question were very similar, which indicates that students had similar abilities. However, for Q2, more percent students achieved a performance greater than 60% with the automated assessment than without it, leading to an improvement.

The results were expected. As Table 6.1 shows, the automated assessment primarily focused on giving feedback regarding entity sets, weak entity sets, attributes, relationship sets, and specializations. In the final exam, Q1 mainly focused on primary keys, weak entity sets, and participation constraints. We assume that students did not show performance improvement because the automated feedback provided for those components was mostly knowledge about mistake feedback, which may not be as effective as we anticipated. On the other hand, since Q2 primarily focused on relationship sets and specializations, where the automated feedback was mostly knowledge about how to proceed feedback, an improvement was anticipated.

6.5 Chapter Summary

This chapter provides an evaluation on the guideline we proposed in Chapter 5 on an entity relationship (ER) modeling question. We investigated the error rate of automated feedback on students' submissions, analyzed the students' learning analytics related to the question, and examined their performance on the final exam.

We found that the automated assessment almost always provided highly accurate feedback. However, we noticed that feedback about mistakes was sometimes very difficult to resolve; students required more detailed explanations or additional knowledge on how to proceed. Lastly, we examined the students' performance on the final exam across two terms with and without the automated feedback. A significant performance improvement was observed when automated feedback was used.

Chapter 7

Discussions and Future Work

Recognizing the characteristics of students in different categories requires instructors to acknowledge that students may need varied assistance (Chapter 3). Having an effective indicator that informs instructors to take action prevents situations from becoming irrecoverable (Chapter 4). Nonetheless, in both cases, the provider of assistance is human. Given the increasing number of learners majoring in computing, the question of scalability arises. Therefore, we introduce guidelines in Chapter 5 for instructors to compose effective automated assessments that provide high-quality feedback, aiming for every student to achieve the best learning outcome eventually.

Chapter 3 introduced our study, which aimed to understand students' learning processes by analyzing their interactions with an automated feedback platform and produced highly valuable findings. In our study, we utilized three features that we consider the most accessible, and we used them to understand students and identify at-risk students; however, replication or reproduction of our study could still be challenging. Therefore, we sought a more generalizable approach, as presented in Chapter 4.

Chapter 4 presented our studies to explore the value of time extensions. Given that the top two reasons students used time extensions were time conflicts and underestimation of the course work, it would be beneficial to provide historical statistics—potentially including data across multiple courses—to encourage students to plan more effectively at the beginning of the semester. Additionally, an effective notification mechanism could be set up to assist students. This system could alert students about average completion times and common scheduling conflicts in specific courses. Implementing a notification platform that integrates these statistics and alerts could lead to fewer time extension requests and more successful time management among students.

It may be more difficult than people expected to make a submission early since that requires students to have a clear understanding of the quality of their work and the metrics used in evaluations. In addition, they also need to have good time management skills. Not many students are capable of meeting all the pre-mentioned requirements. Therefore, students typically stick to the deadline set by instructors to submit their work.

While using grace days and submitting assignments late—after the lab deadline but before the extended deadline—were both legitimate and harmless to students’ performance on assignments, students do not receive extensions on their final exams. Students who used time extensions on their assignments tend to be under-prepared for the final exam. Time extensions, despite their many benefits, can become counterproductive incentives, hindering students from developing effective time management skills.

However, from the instructor’s perspective, students who tend to be under-prepared for the final exam now explicitly self-report their readiness. These time points provide effective indicators for instructors to intervene. While many prior studies found students tended to have low performance if they finished work late, they did not point educators to a time point they could use to identify at-risk students. Our study, instead, shows that instructors can use the original deadline under flexible deadline policies as an indicator to identify such students. Even better, it can be an early indicator, as students showed a significant performance difference in the reproduction study as early as in the second lab, which was scheduled in the second week of the term.

It is a widely accepted assumption that students may learn better with time extensions than without them, which was also one of the motivations for granting such extensions. However, although our studies did not provide evidence about how individual students benefit from this policy, a notable observation was that low-performing students were likely to have trouble learning from feedback. The reproduction study showed that even when detailed feedback was provided in a timely manner, not all students could pass the public test before the extended deadline. This indicates that such students may not be able to interpret feedback properly, meaning they did not know where the answer was wrong or they did not know what to do with the feedback. The real reason may be a mix of the two. This suggests that simply granting students time extensions may not be helpful to these students. It also points out that properly designing feedback is critical to guaranteeing an ideal learning environment for every student [108, 91]. This insight significantly influenced the development of Chapter 5.

Chapter 5 proposes a key shift from the current conception in developing automated assessments—treating the development of an automated assessment similarly to software development, rather than as a one-time process. Thus, regression testing is an essential part

to ensure the assessment code assesses what it is supposed to. However, regression testing is only one part of it. In practice, there are at least two more aspects to consider. First, assessment creators should be able to select some, but not all, probing and diagnosing tests for execution for agile development. Second, re-organization of probing and diagnosing tests should be convenient. For example, if an assignment contains multiple questions, and the initial assessment code was developed to provide feedback on each individual question, this assessment code should be convenient enough to be combined for the whole assignment. A concrete scenario can be an assignment for SQL queries, typically containing multiple questions where students need to write a SQL query for each. Should there be multiple automated assessments, or one automated assessment for all questions?

The question is more complicated when the assessment code needs to be reused. Anecdotally, we observe instructors tend to write their own version of automated assessments. The reason varies. For example, if the original assessment code was written in the Bash programming language, since the original assessment creator was more familiar with Bash, then this assessment code is unlikely to be reused by a person who is more familiar with Python, especially when there are bugs (very likely) in the assessment code about which students may ask.

Above are some of the reasons why we chose `pytest`, as it allows developers to select and organize tests flexibly, and it standardizes the structure of assessment code to some extent.

However, composing ideal automated assessments is more complex than one might anticipate. Like any software development, developing assessment code faces similar issues. For instance, consider a scenario where multiple assignments require automated assessments for a course. If a portion of their assessment code, such as checking for the existence of a student's solution file, is largely the same, should the instructor copy and paste this assessment code multiple times? If so, what happens if there is a bug? Alternatively, should the instructor create a shared library? However, what if the remaining parts of different assessment codes rely on different versions of this shared library?

Furthermore, maintaining assessment code is not trivial, and developing effective probing and diagnosing tests also requires specialized skills. Although automated assessments can access students' source code and, technically, "see" everything, analyzing source code automatically is challenging.

One possible solution for the above problems could be to mimic the idea of package management in software development. In this model, a package is maintained by a small group of highly skilled programmers, while most people focus solely on how to use the package. Here, the assessment code would be divided into discrete chunks, each versioned

with specific tests and associated feedback. Each chunk could then be installed as needed, similar to a “package”. For instance, there could be a chunk dedicated to evaluating whether a student’s solution file is correctly named. Another chunk might assess if the student’s code adheres to style guidelines. Additionally, there could be a more complex chunk designed to provide novice-friendly feedback on the student’s source code through static analysis.

Another common issue is that the runtime environment for developing automated assessments can differ from the environment in which they are executed. This is similar to software development, where there are distinct Development (Dev) and Production (Prod) environments. Because these environments are separate, automated assessments may function well in one but encounter errors in the other. Synchronizing these runtime environments is crucial, but a clear solution has yet to be established.

Instructors developing automated assessments must make decisions regarding certain issues. One of them is if the assessment code is found to be defective and requires a fix, *when* should this fix be applied? 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 automated feedback, 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 while the assessment is in use.

Another scenario requiring decisions involves deciding when and whether to seek human feedback after a probing test fails for *expected* reasons. This is crucial when the probing test evaluates solutions that can have multiple correct invariants. For example, a probing test might compare students’ ER designs against standard examples; therefore, a design that is not within the standard example is expected to fail the probing test. However, this 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 assessment code development. In such cases, human evaluation is advisable. In practice, our implementation involves an automatic notification to the instructor of any probing test failure, regardless of whether the failure was anticipated. This approach allows us to quickly gain insights into students’ solutions. Subsequently, diagnosing tests were introduced after deployment. These tests were not pre-coded. However, this approach was only used for initial submissions, since otherwise, there would be overwhelming notifications. Later submissions received feedback suggesting students contact the instructor directly on the Piazza discussion forum. We customized Marmoset so that students could make such posts trivially with a link click. Once the link is clicked, it will create a Piazza post with relevant information automatically (see Appendix B).

The aforementioned problems are *implementation* problems of automated assessments, focusing on current submissions without considering other factors such as submission histories. Although it is the foundation for any further extensions, to have automated assessments—*eventually*—replace manual assessments, more aspects need to be considered.

7.1 Mistakes and Misconceptions

Following Gusukuma et al. [53]’s terminologies, mistakes are observable manifestations of a student’s understanding or skill in a specific task or problem, whereas misconceptions are the underlying incorrect beliefs or understandings that lead to those mistakes. They proposed using mistake patterns to automatically detect misconceptions; however, we consider the problem to be much more complex. In particular, to identify the underlying misconceptions automatically, careful construction of assessment code and possibly significant human effort are required. For example, consider an aggregation question such as identifying the names of employees who earn the highest salary in a MySQL database. The assessment code can check if an SQL query returns the correct result. If the result is incorrect, it can then check for an unexpected `LIMIT` keyword. If present, a potential misconception could be that the student thinks “there is always only one employee who has the highest salary”, thus allowing for feedback that addresses the misconception to be provided. Despite this appears to be a straightforward case, it still requires careful consideration of the implicit dependencies. Real scenarios tend to be much more complicated. It again emphasizes the need for highly skilled assessment creators for automated assessments.

7.2 Contexts of Students

A fundamental difference between current automated assessments and manual assessments is that human evaluators can take into account contexts beyond the current submission. The submission history [114] is one possible context. Whether students use time extensions or not can be another relevant context. All these contexts can be integrated into automated assessments. For example, the assessment creator can code two versions of automated feedback. One is coarse-grained, providing students with a general direction for solving a mistake, thereby motivating critical thinking; the other is fine-grained, revealing more detail on the direct fix. The version of feedback provided to a student should depend on whether the student made too many failed attempts, or whether the student is one of

those who need time extensions. Also, certain feedback, such as feedback on file names, may have lower priority as students make more attempts. It might be even better for the assessment code to apply a fix for a trivial mistake internally rather than asking the student to fix it. For example, if an assignment requires students to submit one file named `main.cpp`, but a student submits `main.c`, then rather than asking the student to fix it, it may be more beneficial to have the assessment code rename it directly. The feedback can include a line indicating that the file was corrected during the assessment, but it seems not worthwhile to reject it and send it back to the student to submit again.

7.3 Artificial Intelligence (AI)

Considering the capabilities of emerging Artificial Intelligence (AI) techniques such as GPT-4 [22, 81], it is worth exploring their integration into automated assessments. One approach is to use AI to directly assess students' work. However, its non-deterministic nature makes this method unreliable. It is doubtful to rely on an assessment where a subsequent result might differ from a previous one for the same submission. Therefore, a more viable approach might be using AI to generate tests, reducing the time assessment creators need to spend composing the assessment code. This approach also converts non-deterministic assessments into deterministic ones. However, Poldrack et al. [88] attempted to use GPT-4 to automatically generate code and tests. They found that although the generated tests could achieve good coverage of the generated code, these tests often failed. Additional debugging was frequently necessary to determine the root cause of the test failures.

Additionally, AI is proficient in many programming tasks but has limitations in design questions, based on our experience. We have tried using GPT-4, one of the top-tier AI models, which has shown no proficiency in assessing ER modeling problems. This could be due to a lack of an effective prompting strategy; however, it may also highlight another limitation of AI if its capability is highly sensitive to prompts.

Anyhow, further exploration is definitely needed to integrate AI into practical use.

7.4 Evaluation Design

Evaluating the effectiveness of automated assessments on student learning improvement is challenging due to the large number of potential factors and the difficulty in controlling

those that contribute to experimental results. In industry, A/B testing, a simple randomized controlled experiment, is often used to test the effect of a single variable among a number of samples between A and B . We tried this approach in a fourth-year course to compare the effectiveness of automated assessments developed using the unit test philosophy against our new guidelines. We set up automated assessments on a practice question in the course, one version revealed raw test outcomes without considering dependencies, and the other provided feedback messages based on the new guideline. However, unexpectedly, we observed that students somehow posted their solutions publicly, thereby compromising the results. Additionally, although we observed that students struggled less with the feedback following the new guideline compared to those who received raw test outcomes, the data collected were insufficient to determine real significance. We believe graded questions are more ideal than practice questions for such evaluations, and a more robust evaluation design is necessary for future work.

7.5 Chapter Summary

This chapter provides a comprehensive discussion of the studies presented in this thesis and outlines future directions. It interprets the findings from Chapter 3 and Chapter 4, providing insights into the motivations behind the implementation details following the guidelines proposed in Chapter 5. Additionally, it discusses potential enhancements to these implementations.

Chapter 8

Conclusions

This thesis aims to address the ultimate question: *How can computing education be improved so that every student achieves the best possible learning outcome?*

It presents studies aiming to equip instructors with sufficient knowledge to achieve this goal from three perspectives, namely:

1. Correctly understanding differences among students;
2. Effectively identifying help-seeking students; and
3. Conducting scalable actions by providing high-quality automated feedback.

We applied clustering techniques to pre-midterm student behavior data, which included metrics such as how early students make their last submissions, the number of submissions they make, and the best score of those submissions. Although different clustering algorithms produced different clusters, we found that these clusters share common characteristics. We can summarize these clusters as: potential-top-performance (PTP), potential-poor-performance (PPP), and mixed-performance (MP) clusters.

PTP students generally achieved very high scores on assignments and coding labs before the midterm, and they submitted their last submissions early. Conversely, PPP students achieved lower grades on assignments and coding labs before the midterm and tended to submit late. MP students exhibited behaviors that were typically in the middle of these two extremes.

We also examined whether students exhibit consistent behavior pre-midterm and post-midterm. We found that PTP and MP students mostly show inconsistent behavior before

and after the midterm, possibly because the difficulty of assignments increased, while PPP students exhibit consistent behaviors. Combining the insights above, it suggests that if a student consistently achieves low grades and submits work late before the midterm, they are likely to be PPP students who need assistance.

Furthermore, we proposed an easy-to-implement prediction model by excluding MP students identified by any clustering algorithm. We considered students as PTP only if they were placed in the PTP cluster by all clustering algorithms, and as PPP only if they were placed in the PPP cluster by all clustering algorithms. This model achieved high precision in making binary predictions about whether a student needs help to pass the midterm and final exams.

Despite the predictive model, it may be too late due to the high correlation between midterm and final exam performances; there might not be enough time for effective intervention. To address this, we explored the value of time extensions in identifying students' abilities and provided essential knowledge about time extensions to both educators and learners. Students who used time extensions tended to perform significantly worse than those who did not. Time conflicts and underestimation of the coursework were the top two reasons for using time extensions.

Since the primary reasons for using time extensions were time conflicts and underestimation of coursework, it would be helpful to provide historical data to encourage students to plan more effectively at the start of the semester.

Intuitively, a student should learn more effectively with time extensions than without them. However, this does not guarantee that they will achieve the same level of learning outcomes as top-performing students. From the instructor's perspective, if assistance is to be offered, it is more beneficial to help students who used time extensions, as they likely have more room for improvement than those who did not.

Having a reliable indicator of when instructors should take action to assist students is crucial; however, *how* to help them remains a challenge. We examined scalable automated feedback to achieve this. We identified drawbacks in the current automated feedback provision process and developed novel guidelines that enable instructors to compose automated assessments capable of providing accurate feedback. A key consideration within this guideline is that whenever it determines that pre-coded automated feedback cannot be provided, it turns to human assistance. This human assistance can then be integrated as new tests of the automated assessments, gradually enhancing their quality. We evaluated this guideline using an Entity-Relationship (ER) question in a database course. We found the automated feedback to be both precise and helpful, and students' performance improved in the final exam.

References

- [1] *The Cambridge Handbook of Computing Education Research*. Cambridge Handbooks in Psychology. Cambridge University Press.
- [2] International Educational Data Mining Society. URL <https://educationaldatamining.org>. Accessed on: 2024-Jun-10.
- [3] GradeScope. URL <https://www.gradescope.com>. Accessed on: 2024-Apr-15.
- [4] ISO/IEC/IEEE international standard - systems and software engineering – vocabulary. pages 1–418. doi: 10.1109/IEEESTD.2010.5733835.
- [5] Submittity. URL <https://submittity.org/index/overview>. Accessed on: 2024-Apr-15.
- [6] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. WatDFS: A project for understanding distributed systems in the undergraduate curriculum. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19*, pages 920–926. Association for Computing Machinery. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287473. URL <https://doi.org/10.1145/3287324.3287473>.
- [7] Howard E. Aldrich and Joseph Lowman. Assignments: Better Late Than Never? 24 (6):10–11. ISSN 10572880. doi: 10.1002/ntlf.30045. URL <https://onlinelibrary.wiley.com/doi/10.1002/ntlf.30045>.
- [8] William Allen, Shelly Belsky, Ben Kelly, Jenay Barela, Matthew Peveler, and Barbara Cutler. Metrics for student classroom engagement and correlation to software assignment plagiarism. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2, SIGCSE 2022*, page 1141. Association for Computing Machinery. ISBN 978-1-4503-9071-2. doi: 10.1145/3478432.3499133. URL <https://doi.org/10.1145/3478432.3499133>.

- [9] Mingxiao An, Hongyi Zhang, Jaromir Savelka, Shijie Zhu, Chris Bogart, and Majd Sakr. Are working habits different between well-performing and at-Risk students in online project-based courses? In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 324–330, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8214-4. doi: 10.1145/3430665.3456320.
- [10] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: Ordering points to identify the clustering structure. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 49–60. ACM Press, 1999. doi: 10.1145/304182.304187.
- [11] Raheela Asif, Agathe Merceron, and Mahmood Khan Pathan. Investigating performance of students: A longitudinal study. In *Proceedings of the Fifth International Conference on Learning Analytics and Knowledge*, LAK '15, pages 108–112, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3417-4. doi: 10.1145/2723576.2723579.
- [12] Owen Astrachan, Nick Parlante, Daniel D. Garcia, and Stuart Reges. Teaching tips we wish they'd told us before we started. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 2–3, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 1-59593-361-1. doi: 10.1145/1227310.1227314.
- [13] John Aycock and Jim Uhl. Choice in the classroom. *ACM SIGCSE Bulletin*, 37(4): 84–88, December 2005. ISSN 0097-8418. doi: 10.1145/1113847.1113883.
- [14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *Acm Computing Surveys*, 51(3), May 2018. ISSN 0360-0300. doi: 10.1145/3182657.
- [15] Doug Baldwin. Discovery learning in computer science. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '96, pages 222–226, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0-89791-757-X. doi: 10.1145/236452.236544.
- [16] Theresa Beaubouef and John Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *Sigcse Bulletin : A Quarterly*

Publication of The Special Interest Group On Computer Science Education, 37(2): 103–106, June 2005. ISSN 0097-8418. doi: 10.1145/1083431.1083474.

- [17] K. Becker. How much choice is too much? *Sigcse Bulletin : A Quarterly Publication of The Special Interest Group On Computer Science Education*, 38(4):78–82, June 2006. ISSN 0097-8418. doi: 10.1145/1189136.1189176.
- [18] Jeremiah Blanchard, John R. Hott, Vincent Berry, Rebecca Carroll, Bob Edmison, Richard Glassey, Oscar Karnalim, Brian Plancher, and Seán Russell. 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*, ITiCSE-WGR '22, pages 261–292, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9798400700101. doi: 10.1145/3571785.3574129.
- [19] Jonas Boustedt, Robert McCartney, Josh Tenenberg, Scott D. Anderson, Caroline M. Eastman, Daniel D. Garcia, Paul V. Gestwicki, and Margaret S. Menzin. It seemed like a good idea at the time. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 528–529, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 978-1-59593-799-5. doi: 10.1145/1352135.1352311.
- [20] Michael K. Bradshaw. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 488–493, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-2966-8. doi: 10.1145/2676723.2677247.
- [21] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi: 10.1191/1478088706qp063oa.
- [22] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of Artificial General Intelligence: Early experiments with GPT-4, April 2023.
- [23] Kevin Buffardi and Stephen H. Edwards. Responses to adaptive feedback for software testing. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education - ITiCSE '14*, pages 165–170, Uppsala, Sweden, 2014. ACM Press. ISBN 978-1-4503-2833-3. doi: 10.1145/2591708.2591756.

- [24] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: Experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.
- [25] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08*, pages 209–224. USENIX Association.
- [26] Jennifer Campbell, Andrew Petersen, and Jacqueline Smith. Self-paced Mastery Learning CS1. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 955–961, Minneapolis MN USA, February 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287481.
- [27] Huanyi Chen and Paul Ward. Connecting actionable feedback with course concepts using an auto-feedback framework - a case study in a database course. . URL <https://ojs.library.queensu.ca/index.php/PCEEA/article/view/17139>.
- [28] Huanyi Chen and Paul Ward. Clustering students using pre-midterm behaviour data and predict their exam performance. In *Proceedings of the 15th International Conference on Educational Data Mining*, pages 689–694. International Educational Data Mining Society, July 2022. doi: 10.5281/zenodo.6853143.
- [29] Huanyi Chen and Paul Ward. Providing High-Quality Formative Feedback for Database Assignments. In *2024 ASEE Annual Conference and Exposition*, pages 1–17, Portland, Oregon, June 2024. ASEE Conferences. doi: 10.18260/1-2--47903.
- [30] Huanyi Chen and Paul A. S. Ward. Predicting student performance using data from an auto-grading system. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON ’19*, pages 234–243. IBM Corp., .
- [31] Huanyi Chen and Paul A S Ward. Metacognitive Accuracy in Homework Assignments, Time-Limited Quizzes, and Learning Objectives. *Proceedings of the 30th International Conference on Computers in Education*, 1:1–6, 2022.
- [32] Huanyi Chen and Paul A.S. Ward. Enhancing automated feedback in on-going assignments. In *Proceedings of the 55th ACM Technical Symposium on Computer*

- Science Education V. 2*, Sigcse 2024, pages 1596–1597. Association for Computing Machinery, . ISBN 9798400704246. doi: 10.1145/3626253.3635571. URL <https://doi-org.proxy.lib.uwaterloo.ca/10.1145/3626253.3635571>.
- [33] Huanyi Chen and Paul A.S. Ward. The value of time extensions in identifying students abilities. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, pages 512–518, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701382. doi: 10.1145/3587102.3588847.
- [34] Open Science Collaboration. Estimating the reproducibility of psychological science. 349(6251):aac4716. doi: 10.1126/science.aac4716. URL <https://www.science.org/doi/abs/10.1126/science.aac4716>.
- [35] David Coppit and Jennifer M. Haddox-Schatz. Large team projects in software engineering courses. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 137–141. Association for Computing Machinery. ISBN 1-58113-997-7. doi: 10.1145/1047344.1047400. URL <https://doi.org/10.1145/1047344.1047400>.
- [36] Mats Daniels, Lauri Malmi, Arnold Pears, and Simon. What is computing education research (CER)? In Mikko Apiola, Sonsoles López-Pernas, and Mohammed Saqr, editors, *Past, Present and Future of Computing Education Research : A Global Perspective*, pages 9–31. Springer International Publishing. ISBN 978-3-031-25336-2. doi: 10.1007/978-3-031-25336-2_2. URL https://doi.org/10.1007/978-3-031-25336-2_2.
- [37] Galina Deeva, Daria Bogdanova, Estefanía Serral, Monique Snoeck, and Jochen De Weerd. A review of automated feedback systems for learners: Classification framework, challenges and opportunities. *Computers & Education*, 162:104094, March 2021. ISSN 03601315. doi: 10.1016/j.compedu.2020.104094.
- [38] Martin Dick, Margot Postema, and Jan Miller. Teaching tools for software engineering education. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '00, pages 49–52. Association for Computing Machinery. ISBN 1-58113-207-7. doi: 10.1145/343048.343072. URL <https://doi.org/10.1145/343048.343072>.
- [39] Ashish Dutt, Maizatul Akmar Ismail, and Tutut Herawan. A systematic review on educational data mining. *IEEE access : practical innovations, open solutions*, 5: 15991–16005, 2017. doi: 10.1109/ACCESS.2017.2654247.

- [40] Stephen Edwards and Zhiyi Li. Towards progress indicators for measuring student programming effort during solution development. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, pages 31–40, Koli Finland, November 2016. ACM. ISBN 978-1-4503-4770-9. doi: 10.1145/2999541.2999561.
- [41] Stephen H Edwards. Using test-driven development in the classroom: Providing students with concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications*. EISTA’03, 2003.
- [42] 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.
- [43] 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.
- [44] Andrew Emerson, Andy Smith, Fernando J. Rodriguez, Eric N. Wiebe, Bradford W. Mott, Kristy Elizabeth Boyer, and James C. Lester. Cluster-based analysis of novice coding misconceptions in block-based programming. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE ’20, pages 825–831, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-6793-6. doi: 10.1145/3328778.3366924.
- [45] Bradley Erickson, Sarah Heckman, and Collin F. Lynch. Characterizing Student Development Progress: Validating Student Adherence to Project Milestones. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, pages 15–21, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9070-5. doi: 10.1145/3478431.3499373.
- [46] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD’96, pages 226–231, Portland, Oregon, 1996. AAAI Press.

- [47] Katrina Falkner, Rebecca Vivian, Nickolas Falkner, and Sally-Ann Williams. Reflecting on three offerings of a community-centric MOOC for K-6 computer science teachers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 195–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4698-6. doi: 10.1145/3017680.3017712.
- [48] J. Michael Fitzpatrick, Ákos Lédeczi, Gayathri Narasimham, Lee Lafferty, Réal Labrie, Paul T. Mielke, Aatish Kumar, and Katherine A. Brady. Lessons learned in the design and delivery of an introductory programming MOOC. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 219–224, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4698-6. doi: 10.1145/3017680.3017730.
- [49] 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.
- [50] Dan Garcia, Jim Huggins, Christine Alvarado, Paul Gestwicki, Andy Gunawardena, Victoria Hong, and Ellen Spertus. It Seemed Like a Good Idea at the Time (COVID-19 edition). In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2*, pages 1021–1022, Providence RI USA, March 2022. ACM. ISBN 978-1-4503-9071-2. doi: 10.1145/3478432.3499250.
- [51] Seth Copen Goldstein, Hongyi Zhang, Majd Sakr, Haokang An, and Cameron Dashti. Understanding how work habits influence student performance. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, pages 154–160, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6895-7. doi: 10.1145/3304221.3319757.
- [52] 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. Association for Computing Machinery, . ISBN 978-1-4503-6793-6. doi: 10.1145/3328778.3366913. URL <https://doi.org/10.1145/3328778.3366913>.
- [53] Luke Gusukuma, Austin Cory Bart, Dennis Kafura, and Jeremy Ernst. Misconception-Driven Feedback: Results from an Experimental Study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 160–168. ACM, . ISBN 978-1-4503-5628-2. doi: 10.1145/3230977.3231002.

- [54] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, Sigcse '18, pages 278–283. Association for Computing Machinery. ISBN 978-1-4503-5103-4. doi: 10.1145/3159450.3159502. URL <https://doi-org.proxy.lib.uwaterloo.ca/10.1145/3159450.3159502>.
- [55] 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.
- [56] Qiang Hao, David H. Smith IV, Naitra Iriumi, Michail Tsikerdekis, and Amy J. Ko. A systematic investigation of replications in computing education research. *ACM Transactions on Computing Education*, 19(4), August 2019. doi: 10.1145/3345328.
- [57] 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. ACM. ISBN 978-1-4503-4450-0. doi: 10.1145/3051457.3051467.
- [58] 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.
- [59] Arto Hellas, Petri Ihantola, Andrew Petersen, Vangel V. Ajanovski, Mirela Gutica, Timo Hynninen, Antti Knutas, Juho Leinonen, Chris Messom, and Soohyun Nam Liao. Predicting academic performance: 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 175–199. Association for Computing Machinery. ISBN 978-1-4503-6223-8. doi: 10.1145/3293881.3295783. URL <https://doi-org.proxy.lib.uwaterloo.ca/10.1145/3293881.3295783>.
- [60] Melissa Hills and Kim Peacock. Replacing Power with Flexible Structure: Implementing Flexible Deadlines to Improve Student Learning Experiences. *Teaching and Learning Inquiry*, 10:25, July 2022. doi: 10.20343/teachlearning.10.26.

- [61] 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.
- [62] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, Iticse-Wgr '15, pages 41–63, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-4146-2. doi: 10.1145/2858796.2858798.
- [63] David Joyner, Ryan Arrison, Mehnaz Ruksana, Evi Salguero, Zida Wang, Ben Wellington, and Kevin Yin. From clusters to content: Using code clustering for course improvement. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 780–786, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287459.
- [64] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education*, 19(1), September 2018. doi: 10.1145/3231711.
- [65] Hassan Khosravi and Kendra M.L. Cooper. Using learning analytics to investigate patterns of performance and engagement in large classes. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 309–314, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4698-6. doi: 10.1145/3017680.3017711.
- [66] Anthony Kleerekoper and Andrew Schofield. SQL tester: An online SQL assessment tool and its impact. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pages 87–92. ACM. ISBN 978-1-4503-5707-4. doi: 10.1145/3197091.3197124. URL <https://dl.acm.org/doi/10.1145/3197091.3197124>.
- [67] 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.

- [68] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laughner, and Florian Bruhin. Pytest 7.4, 2004.
- [69] Angelo Kyrilov and David C. Noelle. Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 122–126, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-4020-5. doi: 10.1145/2828959.2828968.
- [70] 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.
- [71] Soohyun Nam Liao, Daniel Zingaro, Kevin Thai, Christine Alvarado, William G. Griswold, and Leo Porter. A robust machine learning technique to predict low-performing students. *ACM Transactions on Computing Education*, 19(3), January 2019. doi: 10.1145/3277569.
- [72] David Liu and Andrew Petersen. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 666–671, Minneapolis MN USA, February 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287503.
- [73] Madeleine Lorås and Trond Aalberg. Characteristics of the student-driven learning environment in computing education: A case study on the interaction between educational design and study behavior. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 11–17, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8214-4. doi: 10.1145/3430665.3456310.
- [74] U. V. Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17: 395–416, 2007.
- [75] 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.

- [76] M. Md Rahman and Y. Watanobe. An efficient approach for selecting initial centroid and outlier detection of data clustering. In *Proc. 18th Int. Conf. Intell. Softw. Methodol., Tools Techn. (SoMeT19)*, pages 616–628, Kuching, Malaysia, 2019.
- [77] Laurie A. Miller, Carlos J. Asarta, and James R. Schmidt. Completion deadlines, adaptive learning assignments, and student performance. *Journal of Education for Business*, 94(3):185–194, April 2019. ISSN 0883-2323, 1940-3356. doi: 10.1080/08832323.2018.1507988.
- [78] Shirin Mojarad, Alfred Essa, Shahin Mojarad, and Ryan S. Baker. Data-driven learner profiling based on clustering student behaviors: Learning consistency, pace and effort. In Roger Nkambou, Roger Azevedo, and Julita Vassileva, editors, *Intelligent Tutoring Systems*, pages 130–139, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91464-0.
- [79] Susanne Narciss. Feedback strategies for interactive learning tasks. In *Handbook of Research on Educational Communications and Technology*, pages 125–144. Routledge, 2008.
- [80] Shinichi Oeda and Genki Hashimoto. Log-data clustering analysis for dropout prediction in beginner programming classes. *Procedia Computer Science*, 112:614–621, 2017. ISSN 1877-0509. doi: 10.1016/j.procs.2017.08.088.
- [81] OpenAI. GPT-4 technical report, 2023.
- [82] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. 22(3). doi: 10.1145/3513140. URL <https://doi-org.proxy.lib.uwaterloo.ca/10.1145/3513140>.
- [83] Mark A Patton. The importance of being flexible with assignment deadlines. *Higher education in Europe. Enseignement supérieur en Europe. Vysshee obrazovanie v Evrope*, 25(3):417–423, 2000. ISSN 0379-7724. doi: 10.1080/713669270.
- [84] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

- [85] Dilhan Perera, Judy Kay, Irena Koprinska, Kalina Yacef, and Osmar R. Zaïane. Clustering and sequential pattern mining of online collaborative learning data. *IEEE Transactions on Knowledge and Data Engineering*, 21(6):759–772, 2009. doi: 10.1109/TKDE.2008.138.
- [86] Joel Peterson and Matthew Digman. A Comparison of Learning Outcomes and Learner Satisfaction in a CADD Course with Flexible and Rigid Deadlines. In *2018 ASEE Annual Conference & Exposition Proceedings*, page 29664, Salt Lake City, Utah, June 2018. ASEE Conferences. doi: 10.18260/1-2--29664.
- [87] Tung Phung, Victor-Alexandru Pădurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation. In *Proceedings of the 14th Learning Analytics and Knowledge Conference*, pages 12–23, Kyoto Japan, March 2024. ACM. ISBN 9798400716188. doi: 10.1145/3636555.3636846.
- [88] Russell A. Poldrack, Thomas Lu, and Gašper Beguš. AI-assisted coding: Experiments with GPT-4, April 2023.
- [89] Thomas W Price, Yihuan Dong, and Tiffany Barnes. Generating Data-driven Hints for Open-ended Programming. page 8, 2016.
- [90] Md. Mostafizer Rahman, Yutaka Watanobe, Taku Matsumoto, Rage Uday Kiran, and Keita Nakamura. Educational Data Mining to Support Programming Learning Using Problem-Solving Data. *IEEE Access*, 10:26186–26202, 2022. ISSN 2169-3536. doi: 10.1109/ACCESS.2022.3157288.
- [91] Kelly Rivers. Automated data-driven hint generation for learning programming. doi: 10.1184/R1/6714911.v1. URL https://kilthub.cmu.edu/articles/thesis/Automated_Data-Driven_Hint_Generation_for_Learning_Programming/6714911.
- [92] 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.
- [93] Meadow Schroeder, Erica Makarenko, and Karly Warren. Introducing a Late Bank in Online Graduate Courses: The Response of Students. *The Canadian journal*

for the scholarship of teaching and learning, 10(2), 2019. ISSN 1918-2902. doi: 10.5206/cjsotl-rcacea.2019.2.8200.

- [94] Varshita Sher, Marek Hatala, and Dragan Gašević. Analyzing the consistency in within-activity learning patterns in blended learning. In *Proceedings of the Tenth International Conference on Learning Analytics & Knowledge, LAK '20*, pages 1–10, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7712-6. doi: 10.1145/3375462.3375470.
- [95] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. ISBN 978-0-07-802215-9.
- [96] Shaymaa E. Sorour, Tsunenori Mine, Kazumasa Goda, and Sachio Hirokawa. A Predictive Model to Evaluate Student Performance. *Journal of Information Processing*, 23(2):192–201, 2015. ISSN 1882-6652. doi: 10.2197/ipsjjip.23.192.
- [97] 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.
- [98] 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.
- [99] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. Analyzing student work patterns using programming exercise data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Sigcse '15*, pages 18–23, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-2966-8. doi: 10.1145/2676723.2677297.
- [100] Sumukh Sridhara, Brian Hou, Jeffrey Lu, and John DeNero. Fuzz Testing Projects in Massive Courses. In *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*, pages 361–367, Edinburgh Scotland UK, April 2016. ACM. ISBN 978-1-4503-3726-7. doi: 10.1145/2876034.2876050.

- [101] D. Steinley. Properties of the hubert-arabie adjusted rand index. *Psychological methods*, 9 3:386–96, 2004.
- [102] Daniel E. Stevenson and Paul J. Wagner. Developing real-world programming assignments for CS1. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 158–162, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1-59593-055-8. doi: 10.1145/1140124.1140167.
- [103] G. Strang. Introduction to linear algebra. chapter 6, pages 288–296. Wellesley-Cambridge Press, MIT, 2016. ISBN 978-0-9802327-7-6.
- [104] T. Lowe. Rethinking assessment in early computing courses. In *2022 IEEE Frontiers in Education Conference (FIE)*, pages 1–7, 8. ISBN 2377-634X. doi: 10.1109/FIE56618.2022.9962573.
- [105] Linus Torvalds. Git. Software tool, 2005.
- [106] Scott A. Turner, Manuel A. Pérez-Quñones, and Stephen H. Edwards. minimUML: A minimalist approach to UML diagramming for early computer science education. *Journal on Educational Resources in Computing*, 5(4):1, December 2005. ISSN 1531-4278, 1531-4278. doi: 10.1145/1186639.1186640.
- [107] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology*, 55(6):1004–1016, June 2013. ISSN 0950-5849. doi: 10.1016/j.infsof.2012.12.005.
- [108] Wengran Wang, Yudong Rao, Rui Zhi, Samiha Marwan, Ge Gao, and Thomas W. Price. Step tutor: Supporting students through step-by-step example-based feedback. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, pages 391–397, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-6874-2. doi: 10.1145/3341525.3387411.
- [109] Wengran Wang, Chenhao Zhang, Andreas Stahlbauer, Gordon Fraser, and Thomas Price. SnapCheck: Automated testing for snap! Programs. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 227–233, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8214-4. doi: 10.1145/3430665.3456367.

- [110] J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58:236–244, 1963.
- [111] Joseph B. Wiggins, Fahmid M. Fahid, Andrew Emerson, Madeline Hinckle, Andy Smith, Kristy Elizabeth Boyer, Bradford Mott, Eric Wiebe, and James Lester. Exploring novice programmers’ hint requests in an intelligent block-based coding environment. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 52–58, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8062-1.
- [112] Christopher A. Wolters and Anna C. Brady. College Students’ Time Management: A Self-Regulated Learning Perspective. 33(4):1319–1351. ISSN 1573-336X. doi: 10.1007/s10648-020-09519-z. URL <https://doi.org/10.1007/s10648-020-09519-z>.
- [113] Ursula Wolz, Henry H. Leitner, David J. Malan, and John Maloney. Starting with scratch in CS 1. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE ’09, pages 2–3, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 978-1-60558-183-5. doi: 10.1145/1508865.1508869.
- [114] Lisa Yan, Annie Hu, and Chris Piech. Pensieve: Feedback on Coding Process for Novices. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 253–259. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287483. URL <https://dl.acm.org/doi/10.1145/3287324.3287483>.
- [115] Lisa Yan, Nick McKeown, Chris Piech, Balaji Prabhakar, Mehran Sahami, and Stanford University Department of Electrical Engineering. *Tools to Understand How Students Learn*. PhD thesis, [Stanford University], [Stanford, California], 2019.

APPENDICES

Appendix A

Clustering Algorithms Used in Chapter 3

K-Means The K-Means algorithm will try to separate samples in groups of equal variance, minimizing a criterion known as the *inertia* or *within-cluster sum-of-squares*.

Affinity Propagation Affinity propagation algorithm will create clusters by sending messages between pairs of samples until convergence. The advantage of using it is that it does not require the number-of-cluster parameter be set in advance. It will group points to a same cluster if all of them having the same exemplar point. Although there isn't explicit hyper-parameter for setting the number of clusters, there's a "preference" hyper-parameter affecting the number of clusters. We test it with different settings and found when the grouped number of clusters is 3, the result can be well interpreted.

Spectral Clustering Spectral clustering algorithm [74] performs a low-dimension embedding of the affinity matrix between samples, followed by clustering of the components of the eigenvectors [103] in the low dimensional space.

Hierarchical Clustering Hierarchical clustering represents a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample. Here we used agglomerative clustering algorithm [110], which will recursively merges the pair of clusters that minimally increases a given linkage distance.

Density-Based Spatial Clustering The density-based spatial clustering algorithm [46] views clusters as areas of high density separated by areas of low density. The implementation we used in our experiment was *OPTICS* [10].

Appendix B

Customized Marmoset Mentioned in Chapter 6 and Chapter 7

We customized Marmoset so that its webpage only shows a single feedback message to students. The approach was to make any automated assessment appear to Marmoset as if it contains a single test, for example, a single test inside the `test.properties` configuration file. In addition, we added a link below the feedback message, so that once clicked, a Piazza post will be created summarizing the most recent feedback on the student's submissions.

Here are the screenshots of the customized Marmoset mentioned in Chapters 6 and 7, along with an example of a Piazza post created by clicking the `Ask an instructor` link.

name	feedback
feedback	You used <code>`inner join`</code> in your code, which only considers the intersection between two tables. However, in this case, we want to retain every record from one of the two tables in the joined table, so using an <code>`inner join`</code> is not appropriate. Can you consider other possible options?

[Ask an instructor on Piazza?](#)

Figure B.1: Customized Marmoset with a Help Link

[Student Name] - A1T2 - Marmoset

Question: A1T2

Context

Most recent submission information and feedback

submission submission_timestamp feedback

- 4 2023-10-24 17:02:19 You used `inner join` in your code, which only considers the intersection between two tables. However, in this case, we want to retain every record from one of the two tables in the joined table, so using an `inner join` is not appropriate. Can you consider other possible options?
- 3 2023-10-24 16:15:02 Remember to keep every row of table `Teams`
- 2 2023-10-24 15:49:30 Remember to keep every row of table `Teams`

Question

(Please post your real question using followup discussions)

socrates/marmoset

This private post is only visible to

Edit good note 0

Updated 59 seconds ago by

followup discussions for lingering questions and comments

Resolved Unresolved @94_f1

Actions

Student 53 seconds ago
I was wondering how I can proceed? Thanks!

good comment 0

Reply to this followup discussion

Figure B.2: An Piazza Post Example

Appendix C

An Actual Assessment Composed using socassess¹

This is an actual assessment that will be used in the future for the ER question. For this question, we aim to check if a student's design matches Design A (Figure C.1) or Design B (Figure C.2). We will describe the following aspects of the assessment: 1) folder structure, *i.e.*, how we organize various components of the assessment; 2) its assessment code, *i.e.*, probing tests, diagnosing tests, and feedback messages; and 3) its quality assurance code, *i.e.*, regression tests.

¹<https://github.com/h365chen/socassess>, accessed: 2024-Sep-15

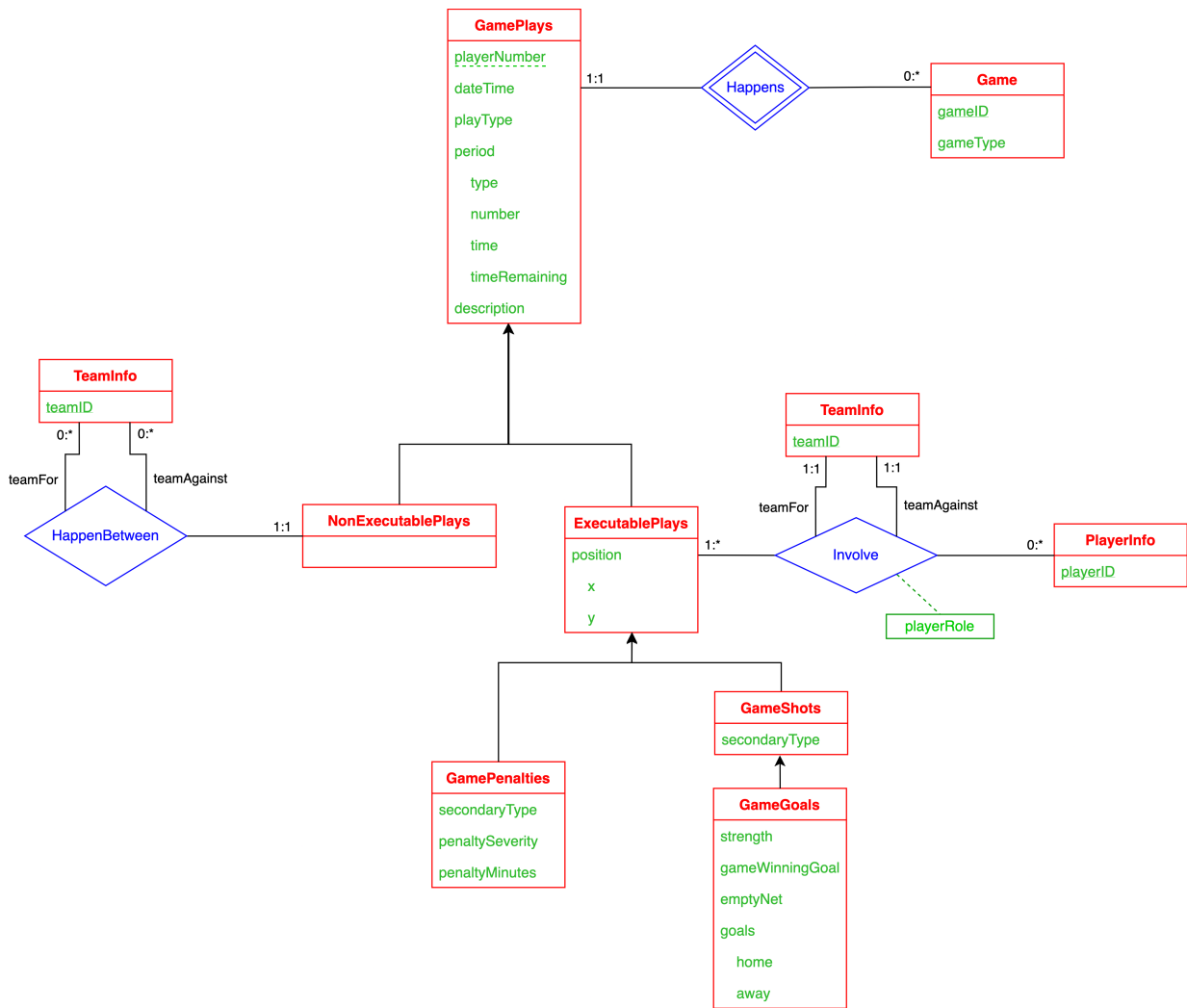


Figure C.1: ER solution A

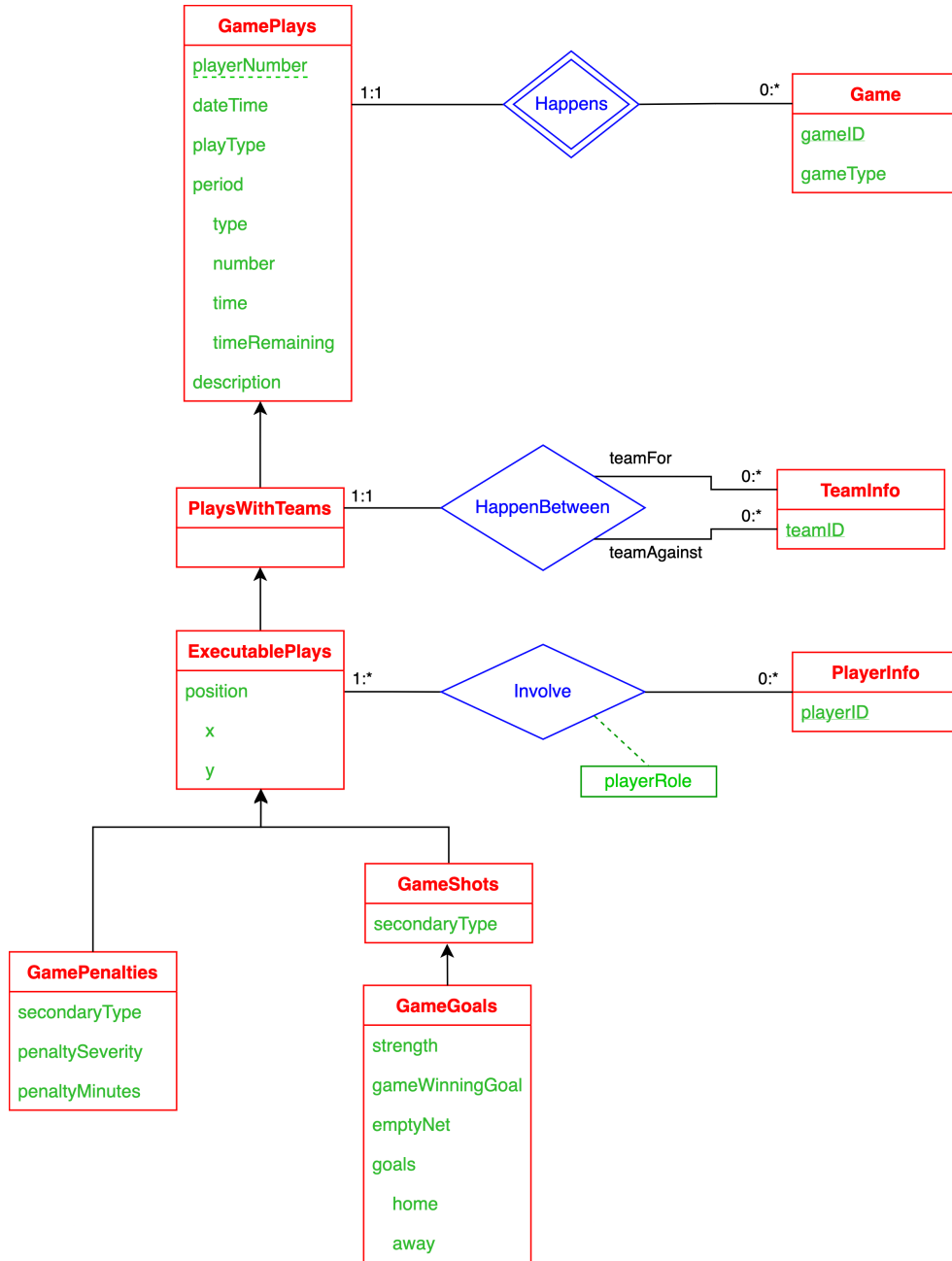


Figure C.2: ER solution B

C.1 Folder Structure of the Assessment

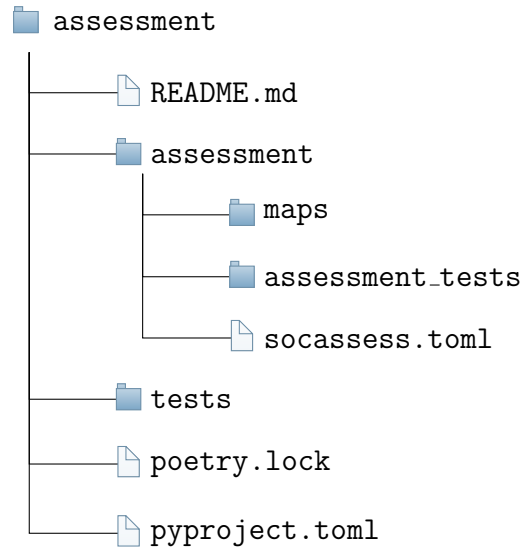


Figure C.3: Folder Structure (Collapsed)

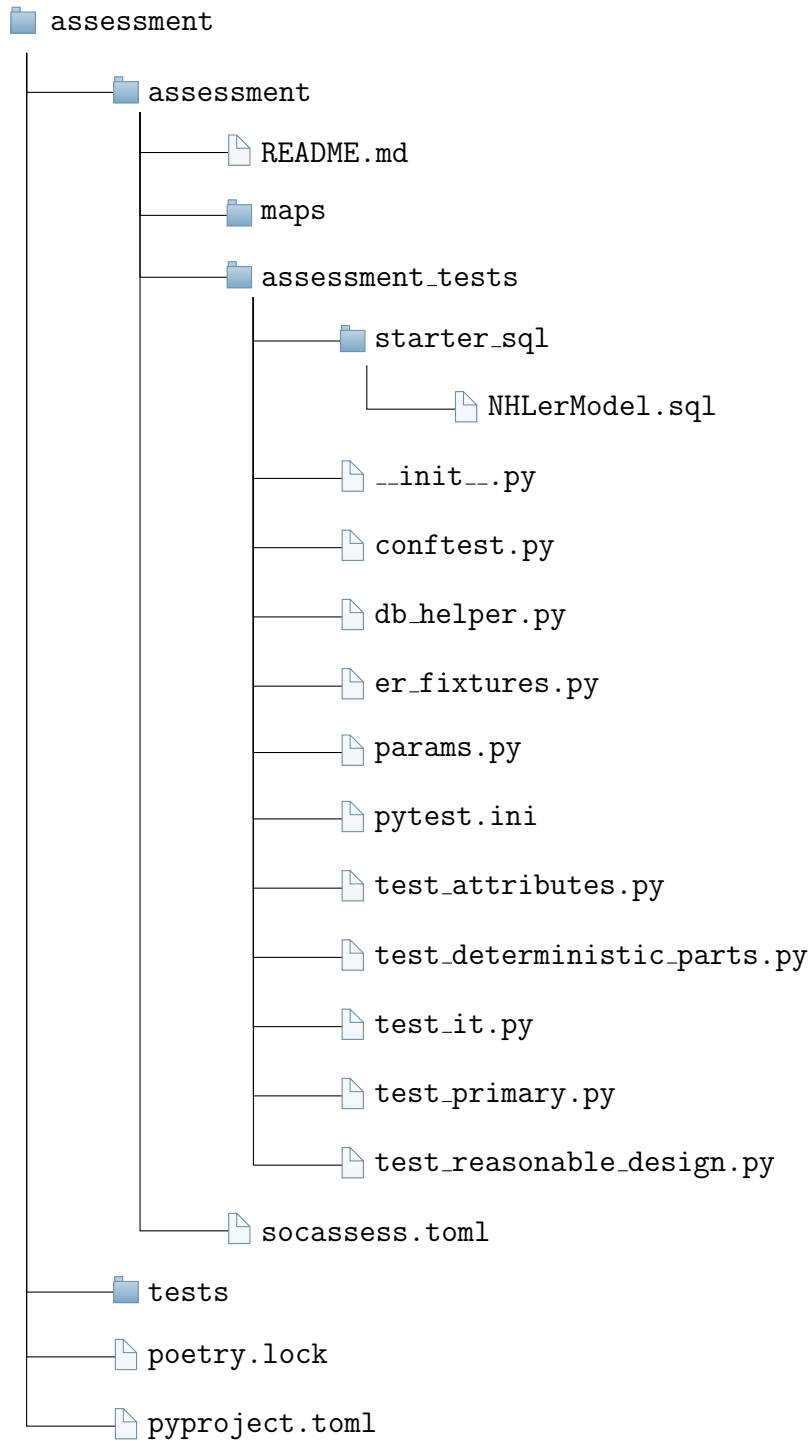


Figure C.4: Folder Structure (Expanding `assessment_tests`)

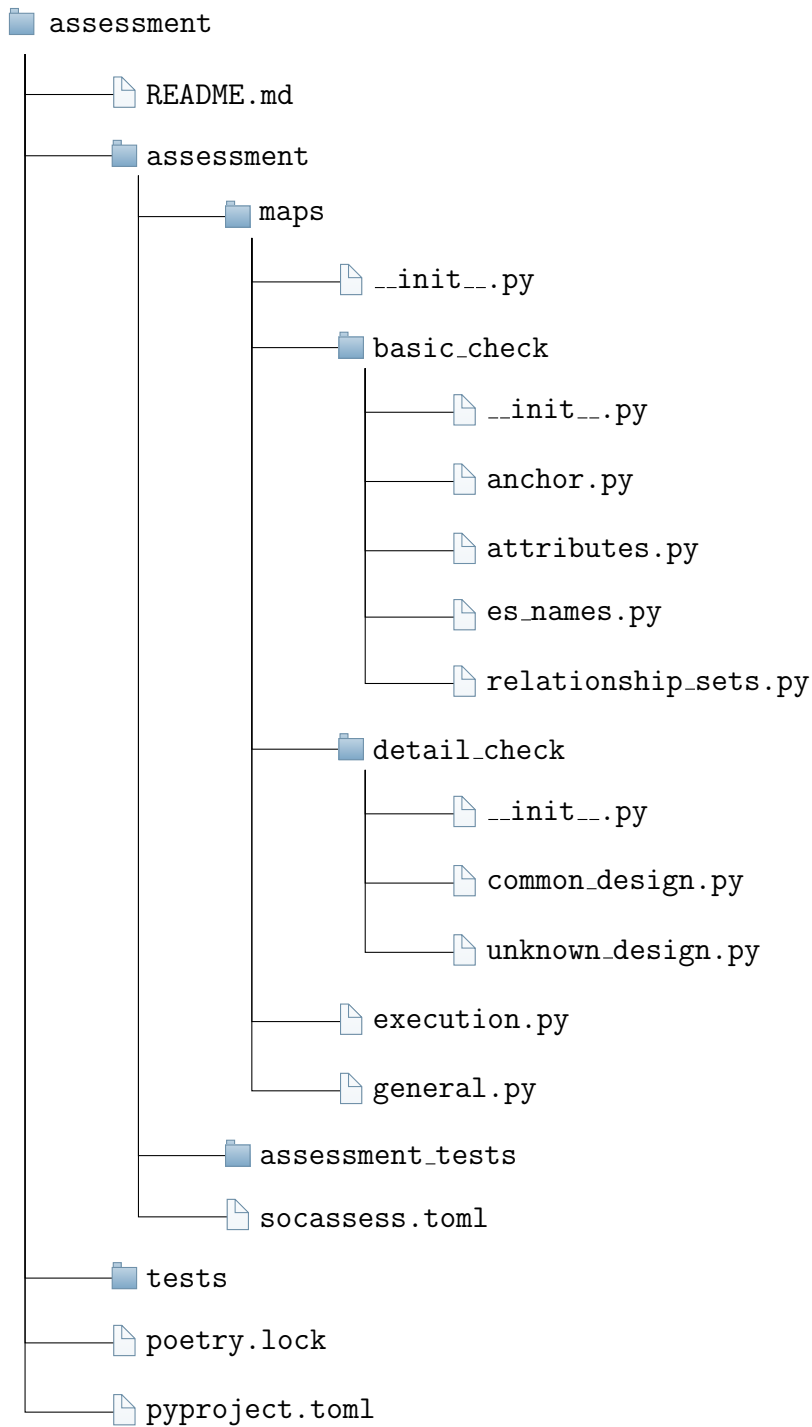


Figure C.5: Folder Structure (Expanding maps)

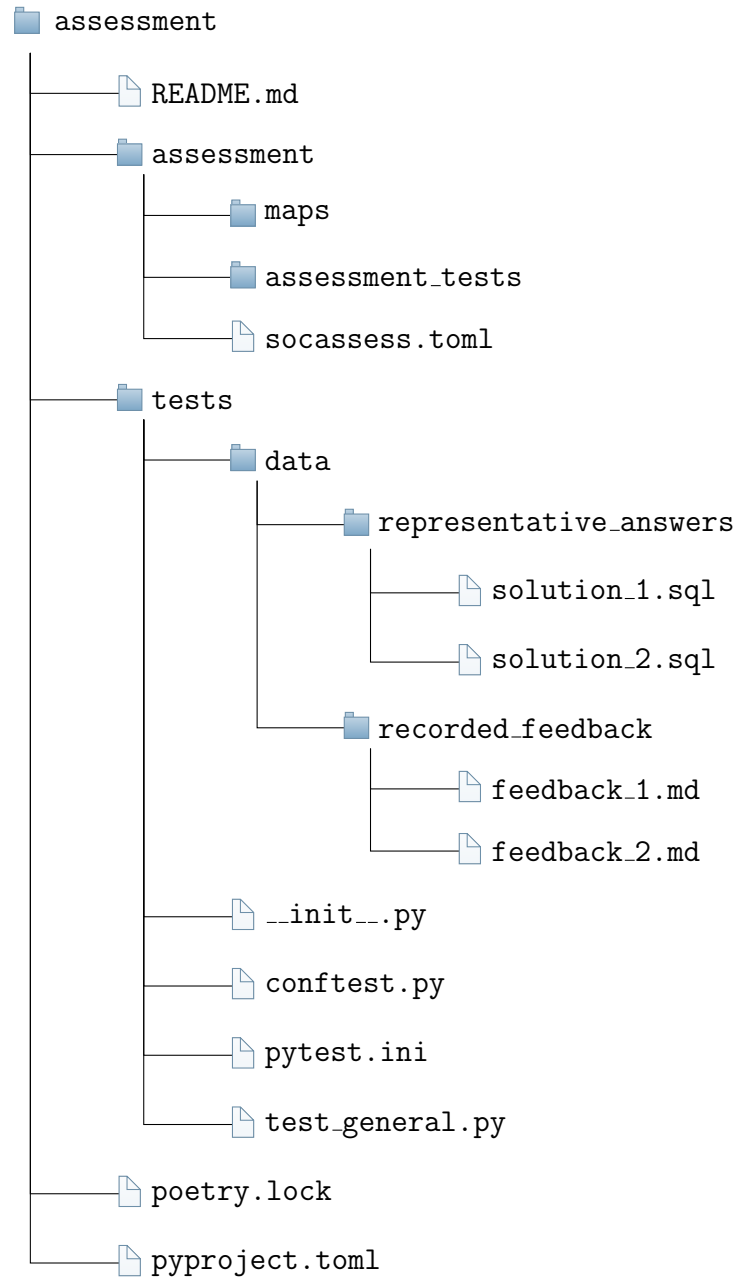


Figure C.6: Folder Structure (Expanding tests, *i.e.*, regression tests)

C.2 Assessment Code

We will share our tests, *i.e.*, probing tests and diagnosing tests, and how test outcomes of these tests get mapped into feedback messages.

C.2.1 Tests

```
1 import re
2 from pathlib import Path
3
4 import pytest
5
6 from .db_helper import db_cursor, db_engine, db_name
7 from .er_fixtures import (attributes, entity_sets, isa,
8                           relationship_sets, weak_entity_sets)
9
10
11 @pytest.fixture(scope="session")
12 def stu_answer_text(artifacts: Path, stu_answer: Path) -> str:
13     """Read the student solution."""
14     # if this fixture throws an exception, the calling test will fail, marked
15     # as "ERROR"
16     ans: str = stu_answer.read_text()
17     # remove tee
18     ans = re.sub(r'^tee .*$', '', ans, flags=re.I | re.M)
19     ans = re.sub(r'^warnings.*$', '', ans, flags=re.I | re.M)
20     ans = re.sub(r'^nowarning.*$', '', ans, flags=re.I | re.M)
21     ans = re.sub(r'^notee.*$', '', ans, flags=re.I | re.M)
22     # the first always fails since we always have a new database
23     ans = re.sub(
24         '^.*alter table EntitySets drop foreign key EntitySets_ibfk_1.*$',
25         '', ans, flags=re.I | re.M
26     )
27     # remove delimiter, which causes problems when it is executed using
28     # mysql-connector. see https://github.com/mysql-net/MySqlConnector/issues/645
29     ans = re.sub(r'^delimiter (@|;).*$', '', ans, flags=re.I | re.M)
30     ans = re.sub(r'^end; @@.*$', 'end;', ans, flags=re.I | re.M)
31     (artifacts / "erModel.sql").write_text(ans, "utf-8")
32     with (artifacts / '_attachments.txt').open('a') as f:
33         f.write("erModel.sql\n")
34     return ans
35
36
37 def pytest_addoption(parser):
38     """Add configurations that are passed by the calling module."""
39     parser.addoption(
40         "--artifacts", action="store", default="artifacts"
41     )
42     parser.addoption(
43         "--ansdir", action="store", default="stu"
44     )
45
```

```

46
47 @pytest.fixture(scope="session")
48 def artifacts(request) -> Path:
49     """Contains the path to the expected student solution file."""
50     opt = request.config.getoption("--artifacts")
51     return Path(opt)
52
53
54 @pytest.fixture(scope="session")
55 def stu_answer(request) -> Path:
56     """Contains the path to the expected student solution file."""
57     opt = request.config.getoption("--ansdir")
58     return Path(opt) / 'erModel.sql'
59
60
61 @pytest.fixture(scope="session")
62 def shared_vars():
63     """Share information between tests."""
64     return {}
65
66
67 def testdb_config():
68     """Get test MySQL server info."""
69     host = '...' # omitted
70     user = '...' # omitted
71     password = '...' # omitted
72     database = '' # we create temporary database later
73     return host, database, user, password

```

Listing 12: assessment_tests/conftest.py

```

1 import random
2 import string
3
4 import pandas as pd
5 import pytest
6 from mysql.connector.cursor import MySQLCursor
7 from sqlalchemy import create_engine
8
9 from . import conftest
10
11
12 def random_string(length=8):
13     """Create a random string which is used as the database name."""
14     characters = string.ascii_letters + string.digits
15     result_str = ''.join(random.choice(characters) for i in range(length))
16     return result_str
17
18
19 def execute_multi_return_last(cursor: MySQLCursor, sql) -> pd.DataFrame:
20     """Execute multiple statements and return the last result."""
21     result: MySQLCursor
22     for result in cursor.execute(sql, multi=True):
23         if result.with_rows:
24             data = pd.DataFrame(cursor.fetchall())
25             if not data.empty:
26                 data.columns = cursor.column_names
27     return data
28
29
30 def execute_multi(cursor: MySQLCursor, sql):
31     """Execute multiple statements and consume their results."""
32     result: MySQLCursor
33     for result in cursor.execute(sql, multi=True):
34         if result.with_rows:
35             result.fetchall()
36     return
37
38
39 @pytest.fixture(scope="session")
40 def db_engine():
41     """Prepare the MySQL engine."""
42     host, database, user, password = conftest.testdb_config()
43     engine = create_engine(
44         f'mysql+mysqlconnector://{user}:{password}@{host}/{database}',
45         # echo_pool="debug",
46         pool_pre_ping=True,
47         pool_reset_on_return=None,
48         # connect_args={
49         #     "consume_results": True
50         # }
51     )
52     return engine
53
54
55 @pytest.fixture(scope="session")
56 def db_name():

```

```

57     """Generate a name for the temporary database."""
58     db_name = "ER_" + random_string(length=8)
59     return db_name
60
61
62 @pytest.fixture(scope="session")
63 def db_cursor(request, db_name, db_engine):
64     """Contains code to setup and teardown a new connection."""
65     # read base sql statements to set up the starter database
66     with db_engine.connect() as con:
67         # setup
68         # CMysqlCursor by default
69         cur: MySQLCursor = con.connection.cursor()
70         sql = f"""
71         create database {db_name};
72         use {db_name};
73         """
74         print(sql)
75         execute_multi(cur, sql)
76
77     yield cur
78
79     # clean up
80     sql = f"""
81     drop database {db_name};
82     """
83     print(sql)
84     cur.execute(sql)
85     cur.close()

```

Listing 13: assessment_tests/db_helper.py

```

1 import pytest
2 from mysql.connector.cursor import MySQLCursor
3
4 from . import db_helper
5
6
7 @pytest.fixture(scope="session")
8 def entity_sets(db_cursor: MySQLCursor):
9     """Read table `RelationshipSets`."""
10    sql = "select esName, primaryKey from EntitySets;"
11    df = db_helper.execute_multi_return_last(
12        db_cursor, sql
13    )
14    print("EntitySets:")
15    print(df)
16    return df
17
18
19 @pytest.fixture(scope="session")
20 def weak_entity_sets(db_cursor: MySQLCursor):
21    """Read table `RelationshipSets`."""
22    sql = "select weakES, identifyingES, discriminator from WeakEntitySets;"
23    df = db_helper.execute_multi_return_last(
24        db_cursor, sql
25    )
26    print("WeakEntitySets:")
27    print(df)
28    return df
29
30
31 @pytest.fixture(scope="session")
32 def relationship_sets(db_cursor: MySQLCursor):
33    """Read table `RelationshipSets`."""
34    sql = "select rsName, esName from RelationshipSets;"
35    df = db_helper.execute_multi_return_last(
36        db_cursor, sql
37    )
38    print("RelationshipSets:")
39    print(df)
40    return df
41
42
43 @pytest.fixture(scope="session")
44 def isa(db_cursor: MySQLCursor):
45    """Read table `IsA`."""
46    sql = "select specializedES, generalizedES from IsA;"
47    df = db_helper.execute_multi_return_last(
48        db_cursor, sql
49    )
50    print("IsA:")
51    print(df)
52    return df
53
54
55 @pytest.fixture(scope="session")
56 def attributes(db_cursor: MySQLCursor):

```

```
57     """Read table `RelationshipSets`."""
58     sql = "select attrName, esName from Attributes;"
59     df = db_helper.execute_multi_return_last(
60         db_cursor, sql
61     )
62     print("Attributes:")
63     print(df)
64     return df
```

Listing 14: assessment_tests/er_fixtures.py

```

1 import pytest
2 from mysql.connector.cursor import MySQLCursor
3
4 from . import db_helper
5
6
7 @pytest.mark.dependency(name="exist")
8 def test_submission_existence(stu_answer):
9     """Ensure the file exists."""
10    assert stu_answer.exists()
11
12
13 @pytest.mark.dependency(name="stu_answer", depends=["exist"])
14 def test_stu_answer(stu_answer_text: str):
15     """Ensure the `stu_answer` fixture passes."""
16    pass
17
18
19 @pytest.mark.dependency(name="execution", depends=["stu_answer"])
20 def test_execution(
21     artifacts,
22     shared_vars,
23     stu_answer_text: str,
24     db_cursor: MySQLCursor):
25     try:
26         db_helper.execute_multi(db_cursor, stu_answer_text)
27     except Exception as e:
28         shared_vars |= {
29             'execution_error': e
30         }
31         (artifacts / 'execution_error.txt').write_text(str(e))
32         raise e
33
34
35 @pytest.mark.dependency(name="execution_fail", depends=["stu_answer"])
36 def test_execution_fail(shared_vars):
37     assert 'execution_error' in shared_vars, 'no execution error found'

```

Listing 15: assessment_tests/test_it.py

```

1  """Contain primary tests."""
2
3  import pandas as pd
4  import pytest
5  from mysql.connector.cursor import MySQLCursor
6  from numpy.testing import assert_equal
7
8  from . import db_helper
9
10
11 @pytest.mark.parametrize("table_name, order_by", [
12     # most relevant
13     ["RelationshipSets", "rsName"],
14     ["IsA", "generalizedES"],
15     # then other useful tables
16     ["EntitySets", "esName"],
17     ["WeakEntitySets", "identifyingES"],
18     ["AggregateEntitySets", "rsName"],
19     ["Attributes", "esName"],
20     ["ComponentAttributes", "compositeName"],
21     ["RelationshipAttributes", "rsName"],
22     ["Role", "rsName"],
23     # -- the following have not been used yet
24     # "AttributeRange",
25     # "RelationshipSetsBounds",
26 ])
27 def test_all_table_contents(table_name, order_by,
28                             artifacts, db_cursor: MySQLCursor):
29     """Print out table contents for human inspection.
30
31     This test should always pass
32
33     """
34     sql = f"SELECT * FROM {table_name} ORDER BY {order_by}"
35     actdf = db_helper.execute_multi_return_last(
36         db_cursor, sql
37     )
38     print(table_name)
39     print(actdf)
40     if not actdf.empty:
41         with (artifacts / "design.txt").open('a') as f:
42             f.write(f'TABLE: {table_name}')
43             f.write('\n')
44             f.write(actdf.to_string())
45             f.write('\n')
46             f.write('\n')
47
48
49 class TestEntitySetsSpellings:
50     key = 'entity_sets_spelling_error'
51
52     @pytest.mark.dependency(name="entity_sets_spelling",
53                             depends=["execution"],
54                             scope="session")
55     def test_entity_sets_spelling(self, shared_vars, db_cursor: MySQLCursor):
56         """Check entity sets' names."""

```

```

57     # prepare esnames
58     expected_esnames = [
59         "Game",
60         "GameChallenges",
61         "GamePlays",
62         "NonExecutablePlays",
63         "ExecutablePlays",
64         "PlayerInfo",
65         "TeamInfo",
66         "GamePenalties",
67         "GameGoals",
68         "GameShots",
69         "PlaysWithTeams", # some students name this as `NonExecutablePlays`
70     ]
71     # get actual df
72     sql = """
73 SELECT DISTINCT esName FROM
74 (
75     SELECT DISTINCT esName FROM EntitySets
76     UNION
77     SELECT DISTINCT specializedES as esName FROM IsA
78 ) t;
79 """
80     actdf = db_helper.execute_multi_return_last(
81         db_cursor, sql
82     )
83     print("actdf:")
84     print(actdf)
85     try:
86         for es in actdf['esName']:
87             assert es in expected_esnames
88     except AssertionError as e:
89         shared_vars |= {self.key: (es, expected_esnames)}
90         raise e
91
92     @pytest.mark.dependency(depends=["execution"],
93                             scope="session")
94     def test_entity_sets_spelling_fail(self, artifacts, shared_vars):
95         assert self.key in shared_vars, 'no incorrectness found around in anchor entities'
96         # otherwise report it
97         es, exp_esnames = shared_vars[self.key]
98         exp_esnames_str = '\n'.join(exp_esnames)
99         (artifacts / f'{self.key}.txt').write_text(f"""
100 I expect entity set names from the list:
101 {exp_esnames_str}
102
103 However, I got `{es}`
104 """
105         .strip())
106
107     class TestAnchorEntitySets:
108         key = 'anchor_entity_sets_error'
109
110         @pytest.mark.dependency(name="anchor_entity_sets",
111                                 depends=["entity_sets_spelling"],
112                                 scope="session")
113         def test_anchor_entity_sets(self, shared_vars, entity_sets):

```

```

114     """Check if there are anchor entities, i.e., Game, GamePlays, TeamInfo,
115     PlayerInfo."""
116     # prepare expected df
117     expdata = [
118         ("Game", "gameID"),
119         ("GamePlays", "gameID"),
120         ("PlayerInfo", "playerID"),
121         ("GamePlays", "playNumber"),
122         ("TeamInfo", "teamID"),
123     ]
124     expdf = pd.DataFrame(
125         expdata, columns=['esName', 'primaryKey']
126     ).sort_values(['esName', 'primaryKey']).reset_index(drop=True)
127     actdf = entity_sets.query("""
128     esName == 'Game'
129     or esName == 'GamePlays'
130     or esName == 'TeamInfo'
131     or esName == 'PlayerInfo'
132     """.replace('\n', ' ').strip()).sort_values(
133         ['esName', 'primaryKey']
134     ).reset_index(drop=True)
135     print("expdf:")
136     print(expdf)
137     print() # newline
138     print("actdf:")
139     print(actdf)
140     try:
141         assert_equal(expdf.values, actdf.values)
142     except AssertionError as e:
143         shared_vars |= {self.key: (expdf, actdf)}
144         raise e
145
146     @pytest.mark.dependency(depends=["entity_sets_spelling"],
147                             scope="session")
148     def test_anchor_entity_sets_fail(self, artifacts, shared_vars):
149         assert self.key in shared_vars, 'no incorrectness found around in anchor entities'
150         # otherwise report it
151         expdf, actdf = shared_vars[self.key]
152         (artifacts / f'{self.key}.txt').write_text(f"""
153     Expected important entity sets:
154     {expdf}
155
156     But I got them as:
157     {actdf}
158     """)

```

Listing 16: assessment_tests/test_primary.py

```

1  """Contain tests to check entity attributes."""
2
3  import pandas as pd
4  import pytest
5  from numpy.testing import assert_equal
6
7
8  class TestGamePlays:
9      key = 'gameplays_attrs_error'
10
11     @pytest.mark.dependency(name="attrs_in_gameplays",
12                             depends=["anchor_entity_sets"],
13                             scope="session")
14     def test_attrs_in_gameplays(self, shared_vars, attributes):
15         """Check attributes in GamePlays."""
16         # prepare expected df
17         expdata = [
18             ('playNumber', 'GamePlays'),
19             ('dateTime', 'GamePlays'),
20             ('playType', 'GamePlays'),
21             ('period', 'GamePlays'),
22             ('type', 'GamePlays'),
23             ('number', 'GamePlays'),
24             ('time', 'GamePlays'),
25             ('timeRemaining', 'GamePlays'),
26             ('description', 'GamePlays'),
27         ]
28         expdf = pd.DataFrame(expdata, columns=[
29             'attrName', 'esName'
30         ]).sort_values(by='attrName')
31         actdf = attributes.query('esName == "GamePlays"') \
32             .sort_values(by='attrName')
33         print("expdf:")
34         print(expdf)
35         print() # newline
36         print("actdf:")
37         print(actdf)
38         try:
39             assert_equal(expdf.values, actdf.values)
40         except AssertionError as e:
41             shared_vars |= {self.key: (e, actdf)}
42             raise e
43
44     @pytest.mark.dependency(name="attrs_in_gameplays_fail",
45                             depends=["anchor_entity_sets"],
46                             scope="session")
47     def test_attrs_in_gameplays_fail(self, artifacts, shared_vars):
48         assert self.key in shared_vars, 'no incorrect attributes found in the GamePlays attributes'
49         # otherwise report it
50         e, _ = shared_vars[self.key]
51         (artifacts / f'{self.key}.txt').write_text(str(e))
52
53     @pytest.mark.dependency(depends=["attrs_in_gameplays_fail"],
54                             scope="class")
55     def test_position_in_gameplays(self, artifacts, shared_vars):
56         """Check if `position` related attributes are still in GamePlays.

```

```

57
58     This is a diagnosing test.
59
60     """
61     _, actdf = shared_vars[self.key]
62     assert "position" in actdf.index \
63         or "x" in actdf.index \
64         or "y" in actdf.index
65
66     @pytest.mark.dependency(depends=["attrs_in_gameplays_fail"],
67                             scope="class")
68     def test_secondaryType_in_gameplays(self, artifacts, shared_vars):
69         """Check if `secondaryType` related attributes are still in GamePlays.
70
71         This is a diagnosing test.
72
73         """
74         _, actdf = shared_vars[self.key]
75         assert "secondary" in actdf.index \
76             or "secondaryType" in actdf.index
77
78     @pytest.mark.dependency(depends=["attrs_in_gameplays_fail"],
79                             scope="class")
80     def test_goals_in_gameplays(self, artifacts, shared_vars):
81         """Check if `goals` related attributes are still in GamePlays.
82
83         This is a diagnosing test.
84
85         """
86         _, actdf = shared_vars[self.key]
87         assert "goals" in actdf.index or "goal" in actdf.index
88
89
90     class TestGameGoals:
91         key = 'gamegoals_attrs_error'
92
93         @pytest.mark.dependency(name="attrs_in_gamegoals",
94                                 depends=["execution"],
95                                 scope="session")
96         def test_attrs_in_gamegoals(self, shared_vars, attributes):
97             """Check attributes in GameGoals."""
98             # prepare expected df
99             expdata = [
100                 ("strength", "GameGoals"),
101                 ("gameWinningGoal", "GameGoals"),
102                 ("emptyNet", "GameGoals"),
103                 ("goal", "GameGoals"),
104                 ("home", "GameGoals"),
105                 ("away", "GameGoals"),
106             ]
107             expdf = pd.DataFrame(expdata, columns=[
108                 'attrName', 'esName'
109             ]).sort_values('attrName')
110             actdf = attributes.query('esName == "GameGoals"') \
111                 .sort_values(by='attrName')
112             print("expdf:")
113             print(expdf)

```

```

114     print() # newline
115     print("actdf:")
116     print(actdf)
117     try:
118         assert_equal(expdf.values, actdf.values)
119     except AssertionError:
120         try:
121             expdf = pd.DataFrame(
122                 expdata + [{"secondaryType", "GameGoals"}],
123                 columns=[
124                     'attrName', 'esName'
125                 ]).sort_values('attrName')
126             assert_equal(expdf.values, actdf.values)
127         except AssertionError:
128             try:
129                 expdf = pd.DataFrame(
130                     expdata + [{"secondary", "GameGoals"}],
131                     columns=[
132                         'attrName', 'esName'
133                     ]).sort_values('attrName')
134                 assert_equal(expdf.values, actdf.values)
135             except AssertionError as e:
136                 shared_vars |= {self.key: e}
137                 raise e
138
139     @pytest.mark.dependency(depends=["execution"],
140                             scope="session")
141     def test_attrs_in_gamegoals_fail(self, artifacts, shared_vars):
142         assert self.key in shared_vars, 'no incorrectness found around the GameGoals attributes'
143         # otherwise report it
144         e = shared_vars[self.key]
145         (artifacts / f'{self.key}.txt').write_text(str(e))
146
147
148     class TestGamePenalties:
149         key = 'game_penalties_attrs_error'
150
151         @pytest.mark.dependency(name="attrs_in_game_penalties",
152                                 depends=["execution"],
153                                 scope="session")
154         def test_attrs_in_game_penalties(self, shared_vars, attributes):
155             """Check attributes in GamePenalties."""
156             # prepare expected df
157             expdata = [
158                 ('penaltySeverity', 'GamePenalties'),
159                 ('penaltyMinutes', 'GamePenalties'),
160             ]
161             expdf = pd.DataFrame(
162                 expdata + [('secondaryType', 'GamePenalties')],
163                 columns=[
164                     'attrName', 'esName'
165                 ]).sort_values('attrName')
166             actdf = attributes.query('esName == "GamePenalties"') \
167                 .sort_values(by='attrName')
168             print("expdf:")
169             print(expdf)
170             print() # newline

```

```

171     print("actdf:")
172     print(actdf)
173     try:
174         assert_equal(expdf.values, actdf.values)
175     except AssertionError:
176         try:
177             expdf = pd.DataFrame(
178                 expdata + [('secondary', 'GamePenalties')],
179                 columns=[
180                     'attrName', 'esName'
181                 ]).sort_values('attrName')
182             assert_equal(expdf.values, actdf.values)
183         except AssertionError as e:
184             shared_vars |= {self.key: e}
185             raise e
186
187     @pytest.mark.dependency(depends=["execution"],
188                             scope="session")
189     def test_attrs_in_game_penalties_fail(self, artifacts, shared_vars):
190         assert self.key in shared_vars, 'no incorrectness found around the GamePenalties attributes'
191         # otherwise report it
192         e = shared_vars[self.key]
193         (artifacts / f'{self.key}.txt').write_text(str(e))

```

Listing 17: assessment_tests/test_attributes.py

```

1  """Contain tests to check deterministic parts of the design."""
2
3  import pandas as pd
4  import pytest
5  from numpy.testing import assert_equal
6
7
8  class TestRsBetweenGamePlaysAndGame:
9      key = 'rs_gameplays_game_error'
10
11     @pytest.mark.dependency(name="rs_gameplays_game",
12                             depends=["anchor_entity_sets"],
13                             scope="session")
14     def test_rs_gameplays_game(self, shared_vars,
15                               relationship_sets, weak_entity_sets):
16         """Check relationship set between gameplays and game.
17
18         GamePlays should be a weak entity set of Game.
19
20         Also, there should be only one relationship set involving Game and
21         GamePlays, and it should involve only Game and GamePlays entity sets.
22
23         GamePlays -- weak_rs -- Game
24
25         """
26         rsdf = relationship_sets
27         expdata = [
28             ("GamePlays", "Game", "playNumber"),
29         ]
30         expdf = pd.DataFrame(expdata, columns=[
31             'weakES', 'identifyingES', 'discriminator'
32         ])
33         try:
34             wsdf = weak_entity_sets.query('weakES == "GamePlays"')
35             assert len(rsdf.query('esName == "Game"')) == 1
36             assert len(rsdf.query('esName == "GamePlays"')) == 1
37             assert_equal(
38                 rsdf.query('esName == "Game"')['rsName'].values,
39                 rsdf.query('esName == "GamePlays"')['rsName'].values,
40             )
41             # GamePlays is a WeakEntity of Game
42             assert_equal(expdf.values, wsdf.values)
43         except AssertionError as e:
44             shared_vars |= {self.key: e}
45             raise e
46
47     @pytest.mark.dependency(depends=["anchor_entity_sets"],
48                             scope="session")
49     def test_rs_gameplays_game_fail(self, artifacts, shared_vars):
50         assert self.key in shared_vars, 'no incorrectness found between GamePlays and Game'
51         # otherwise report it
52         e = shared_vars[self.key]
53         (artifacts / f'{self.key}.txt').write_text(str(e))

```

Listing 18: assessment_tests/test_deterministic_parts.py

```

1  """Contain tests to check reasonable designs."""
2
3  import pytest
4
5
6  class TestDesign1:
7      key = 'design1_error'
8
9      @pytest.mark.dependency(name="design1",
10                             depends=["anchor_entity_sets"],
11                             scope="session")
12  def test_design1(self, shared_vars, relationship_sets, isa):
13      """Check if the student design matches design1.
14
15      The idea is that we have a relationship set connecting ExecutablePlays,
16      TeamInfo, and PlayerInfo, where ExecutablePlays is a specializedES of
17      GamePlays.
18
19      Separately, we also have GameChallenges to connect TeamInfo.
20
21      In this design, we do not have NonExecutablePlays since it does not
22      have any associated relationship sets thus it is redundant. (See the
23      diagnosing test below)
24
25      """
26      rsdf = relationship_sets
27      isadf = isa
28      rsdf_teaminfo = rsdf.query('esName == "TeamInfo"')
29      isadf_gameplays = isadf.query('generalizedES == "GamePlays"')
30      try:
31          assert len(rsdf_teaminfo) == 2
32          rs1, rs2 = rsdf_teaminfo['rsName'].values
33          rs1es = rsdf.query(f'rsName == "{rs1}"')['esName'].to_list()
34          rs2es = rsdf.query(f'rsName == "{rs2}"')['esName'].to_list()
35          rs1es, rs2es = (rs1es, rs2es) if len(rs1es) < len(rs2es) else (rs2es, rs1es)
36          # it has to be GameChallenges, TeamInfo in rs1es
37          assert 'GameChallenges' in rs1es and 'TeamInfo' in rs1es
38          # it has to be PlayerInfo, ExecutablePlays, TeamInfo in rs2es
39          assert 'PlayerInfo' in rs2es and 'ExecutablePlays' in rs2es and 'TeamInfo' in rs2es
40
41          assert len(isadf_gameplays) == 2
42          assert 'ExecutablePlays' in isadf_gameplays['specializedES'].to_list()
43          # `GamePenalties` have to be a specializedES of ExecutablePlays
44          # `GameChallenges` have to be a specializedES of GamePlays
45          isadf_executable_plays = isadf.query('generalizedES == "ExecutablePlays"')
46          assert 'GamePenalties' in isadf_executable_plays['specializedES'].to_list()
47          assert 'GameChallenges' in isadf_gameplays['specializedES'].to_list()
48      except AssertionError as e:
49          shared_vars |= {self.key: e}
50          raise e
51
52      @pytest.mark.dependency(name="design1_fail",
53                             depends=["anchor_entity_sets"],
54                             scope="session")
55  def test_design1_fail(self, artifacts, shared_vars):
56      assert self.key in shared_vars, 'no mismatching found from design1'

```

```

57     # otherwise report it
58     e = shared_vars[self.key]
59     (artifacts / f'{self.key}.txt').write_text(str(e))
60
61 @pytest.mark.dependency(depends=["design1_fail"],
62                          scope="session")
63 def test_design1_extra_NonExecutablePlays_1(self,
64                                             relationship_sets, isa):
65     """Check if there is the extra `NonExecutablePlays`.
66
67     The assertions are basically the same as above, with the exception that
68     we assert `GameChallenges` is a specializedES of `NonExecutablePlays`
69     and `NonExecutablePlays` is a specializedES of `GamePlays`.
70
71     `NonExecutablePlays` is not involved in any relationship sets.
72
73     This is a diagnosing test.
74
75     """
76     rsdf = relationship_sets
77     isadf = isa
78     rsdf_teaminfo = rsdf.query('esName == "TeamInfo"')
79     isadf_gameplays = isadf.query('generalizedES == "GamePlays"')
80     assert len(rsdf_teaminfo) == 2
81     rs1, rs2 = rsdf_teaminfo['rsName'].values
82     rs1es = rsdf.query(f'rsName == "{rs1}"')['esName'].to_list()
83     rs2es = rsdf.query(f'rsName == "{rs2}"')['esName'].to_list()
84     rs1es, rs2es = (rs1es, rs2es) if len(rs1es) < len(rs2es) else (rs2es, rs1es)
85     assert 'GameChallenges' in rs1es and 'TeamInfo' in rs1es
86     # it has to be TeamInfo, PlayerInfo, ExecutablePlays in rs2es
87     assert 'PlayerInfo' in rs2es and 'ExecutablePlays' in rs2es and 'TeamInfo' in rs2es
88
89     assert len(isadf_gameplays) == 2
90     assert 'ExecutablePlays' in isadf_gameplays['specializedES'].to_list()
91     isadf_executable_plays = isadf.query('generalizedES == "ExecutablePlays"')
92     isadf_non_executable_plays = isadf.query('generalizedES == "NonExecutablePlays"') # diff
93     assert 'GamePenalties' in isadf_executable_plays['specializedES'].to_list()
94     # assert 'GameChallenges' in isadf_gameplays['specializedES'].to_list()
95     assert 'GameChallenges' in isadf_non_executable_plays['specializedES'].to_list() # diff
96
97 @pytest.mark.dependency(depends=["design1_fail"],
98                          scope="session")
99 def test_design1_extra_NonExecutablePlays_2(self,
100                                             relationship_sets, isa):
101     """Check if there is the extra `NonExecutablePlays`.
102
103     The assertions are basically the same as above, with the exception that
104     we assert `GameChallenges` is a specializedES of `NonExecutablePlays`
105     and `NonExecutablePlays` is a specializedES of `GamePlays`.
106
107     `GameChallenges` is not involved in any relationship sets.
108
109     This is a diagnosing test.
110
111     """
112     rsdf = relationship_sets
113     isadf = isa

```

```

114     rsdf_teaminfo = rsdf.query('esName == "TeamInfo"')
115     isadf_gameplays = isadf.query('generalizedES == "GamePlays"')
116     assert len(rsdf_teaminfo) == 2
117     rs1, rs2 = rsdf_teaminfo['rsName'].values
118     rs1es = rsdf.query(f'rsName == "{rs1}"')['esName'].to_list()
119     rs2es = rsdf.query(f'rsName == "{rs2}"')['esName'].to_list()
120     rs1es, rs2es = (rs1es, rs2es) if len(rs1es) < len(rs2es) else (rs2es, rs1es)
121     assert 'NonExecutablePlays' in rs1es and 'TeamInfo' in rs1es # diff
122     # it has to be TeamInfo, PlayerInfo, ExecutablePlays in rs2es
123     assert 'PlayerInfo' in rs2es and 'ExecutablePlays' in rs2es and 'TeamInfo' in rs2es
124
125     assert len(isadf_gameplays) == 2
126     assert 'ExecutablePlays' in isadf_gameplays['specializedES'].to_list()
127     isadf_executable_plays = isadf.query('generalizedES == "ExecutablePlays"')
128     isadf_non_executable_plays = isadf.query('generalizedES == "NonExecutablePlays"') # diff
129     assert 'GamePenalties' in isadf_executable_plays['specializedES'].to_list()
130     # assert 'GameChallenges' in isadf_gameplays['specializedES'].to_list()
131     assert 'GameChallenges' in isadf_non_executable_plays['specializedES'].to_list() # diff
132
133
134 class TestDesign2:
135     key = 'design2_error'
136
137     @pytest.mark.dependency(name="design2",
138                             depends=["anchor_entity_sets"],
139                             scope="session")
140     def test_design2(self, shared_vars,
141                     relationship_sets, isa):
142         """Check if the student design matches design2.
143
144         The idea is that we have a relationship set connecting `TeamInfo` to a
145         new Entity, say, `PlaysWithTeams`. Then we make `ExecutablePlays` a
146         specializedES of the `PlaysWithTeams`, with a second relationship set
147         connecting `PlayerInfo`.
148
149         """
150         rsdf = relationship_sets
151         isadf = isa
152         rsdf_teaminfo = rsdf.query('esName == "TeamInfo"')
153         rsdf_playerinfo = rsdf.query('esName == "PlayerInfo"')
154         isadf_gameplays = isadf.query('generalizedES == "GamePlays"')
155         try:
156             assert len(rsdf_teaminfo) == 1
157             rs = rsdf_teaminfo['rsName'].values[0]
158             rses_teaminfo = rsdf.query(f'rsName == "{rs}"')['esName'].to_list()
159             # rses_teaminfo should not be involved in any other relationship
160             # sets other than with `TeamInfo`
161             rses_teaminfo.remove('TeamInfo')
162             assert len(rses_teaminfo) == 1
163
164             assert len(rsdf_playerinfo) == 1
165             rs = rsdf_playerinfo['rsName'].values[0]
166             rses_playerinfo = rsdf.query(f'rsName == "{rs}"')['esName'].to_list()
167             # rses_playerinfo should not be involved in any other relationship
168             # sets other than with `PlayerInfo`
169             rses_playerinfo.remove('PlayerInfo')
170             assert len(rses_playerinfo) == 1

```

```

171     # and it has to be `ExecutablePlays`
172     assert rses_playerinfo[0] == 'ExecutablePlays'
173
174     assert len(isadf_gameplays) == 1
175     assert rses_teaminfo[0] in isadf_gameplays['specializedES'].to_list()
176     isadf_plays_with_teams = isadf.query(f'generalizedES == "{rses_teaminfo[0]}"')
177     assert 'ExecutablePlays' in isadf_plays_with_teams['specializedES'].to_list()
178     isadf_executable_plays = isadf.query('generalizedES == "ExecutablePlays"')
179     assert 'GamePenalties' in isadf_executable_plays['specializedES'].to_list()
180 except AssertionError as e:
181     shared_vars |= {self.key: e}
182     raise e
183
184 @pytest.mark.dependency(name="design2_fail",
185                         depends=["anchor_entity_sets"],
186                         scope="session")
187 def test_design2_fail(self, artifacts, shared_vars):
188     assert self.key in shared_vars, 'no mismatching found from design2'
189     # otherwise report it
190     e = shared_vars[self.key]
191     (artifacts / f'{self.key}.txt').write_text(str(e))

```

Listing 19: assessment_tests/test_reasonable_design.py

C.2.2 Feedback

```
1  """Contain necessary information for socassess to provide feedback."""
2
3  from . import basic_check, detail_check, execution, general
4
5  __all__ = [
6      # required
7      "questions",
8      "selected",
9  ]
10
11
12  # =====
13  # Required
14  # =====
15
16  selected = {
17      "general": general.mappings,
18      "execution": execution.mappings,
19      "basic check": basic_check.mappings,
20      "detail check": detail_check.mappings,
21  }
22
23
24  questions = {
25      "general": "...", # omitted
26      "execution": "...", # omitted
27      "basic check": "...", # omitted
28      "detail check": "...", # omitted
29  }
```

Listing 20: maps/___init___.py

```
1 from socassess import FeedbackLevel
2
3 mappings = {
4     frozenset([
5         'test_it::test_submission_existence::failed',
6     ]): {
7         'feedback': ""
8     }
9     Can you double check your file name? It should be `erModel.sql`. Also, you
10    probably don't need to zip it.
11
12     """.strip(),
13     'level': FeedbackLevel.SINGLE,
14 },
15 frozenset([
16     'test_it::test_submission_existence::passed',
17     'test_it::test_stu_answer::passed',
18 ]): {
19     'feedback': ""
20 }
21 Nice! I found your answer file and it can be correctly parsed.
22
23     """.strip(),
24 },
25 }
```

Listing 21: maps/general.py

```

1  """Provide mappings between test outcomes and feedback messages."""
2
3  from socassess import FeedbackLevel, userargs
4
5
6  def fill_execution_error():
7      """Fill {content} with the execution error when it fails."""
8      # The file name is determined in `assessment_tests`
9      return (userargs.artifacts / 'execution_error.txt').read_text()
10
11
12  mappings = {
13      frozenset([
14          'test_it::test_execution::failed',
15          'test_it::test_execution_fail::passed',
16      ]): {
17          'feedback': """
18
19  I ran into an error executing your erModel.sql.
20
21  ...
22  {content}
23  ...
24          """.strip(),
25          'function': fill_execution_error,
26          'level': FeedbackLevel.SINGLE
27      },
28      frozenset([
29          'test_it::test_execution::passed',
30      ]): {
31          'feedback': """
32
33  Nice! I successfully executed your erModel.sql.
34
35          """.strip(),
36      },
37  }

```

Listing 22: maps/execution.py

```
1 """Provide mappings between test outcomes and feedback messages."""
2
3 from . import anchor, attributes, es_names, relationship_sets
4
5 mappings = \
6     es_names.mappings \
7     | anchor.mappings \
8     | attributes.mappings \
9     | relationship_sets.mappings
```

Listing 23: maps/basic_check/__init__.py

```

1  """Provide mappings between test outcomes and feedback messages."""
2
3  from socassess import userargs
4
5
6  def fill_entity_sets_spelling_error():
7      """Fill {content}.
8
9      I expect entity set names from the list:
10     {exp_esnames_str}
11
12     However, I got `{es}`
13
14     """
15     # The file name is determined in `assessment_tests`
16     return (userargs.artifacts / 'entity_sets_spelling_error.txt').read_text()
17
18
19  mappings = {
20      # anchor entity sets
21      frozenset([
22          'test_primary.TestEntitySetsSpellings::test_entity_sets_spelling::passed',
23      ]): {
24          'feedback': ""
25      }
26      Nice! The entity set names are correctly spelled.
27
28          """.strip(),
29      },
30      frozenset([
31          'test_primary.TestEntitySetsSpellings::test_entity_sets_spelling::failed',
32          'test_primary.TestEntitySetsSpellings::test_entity_sets_spelling_fail::passed',
33      ]): {
34          'feedback': ""
35      }
36      {content}
37          """.strip(),
38          'function': fill_entity_sets_spelling_error,
39      },
40  }

```

Listing 24: maps/basic_check/es_names.py

```

1 from socassess import userargs
2
3
4 def fill_anchor_entity_sets_error():
5     """Fill {content}.
6
7     Expected important entity sets:
8     {expdf}
9
10    But I got them as:
11    {actdf}
12
13    """
14    # The file name is determined in `assessment_tests`
15    return (userargs.artifacts / 'anchor_entity_sets_error.txt').read_text()
16
17
18 mappings = {
19     # anchor entity sets
20     frozenset([
21         'test_primary.TestAnchorEntitySets::test_anchor_entity_sets::passed',
22     ]): {
23         'feedback': ""
24     }
25     Nice! Several entity sets are having correct primary keys.
26
27     """.strip(),
28     },
29     frozenset([
30         'test_primary.TestAnchorEntitySets::test_anchor_entity_sets::failed',
31         'test_primary.TestAnchorEntitySets::test_anchor_entity_sets_fail::passed',
32     ]): {
33         'feedback': ""
34     }
35     It seems some of the important entity sets I am looking at are not correct.
36
37     {content}
38     """.strip(),
39     'function': fill_anchor_entity_sets_error,
40     },
41 }

```

Listing 25: maps/basic_check/anchor.py

```

1 mappings = {
2     # GamePlays and Game
3     frozenset([
4         'test_deterministic_parts.TestRsBetweenGamePlaysAndGame::test_rs_gameplays_game::passed',
5     ]): {
6         'feedback': ""
7     }
8     Nice! You made GamePlays as a weak entity set to Game.
9
10    """.strip(),
11 },
12 frozenset([
13     'test_deterministic_parts.TestRsBetweenGamePlaysAndGame::test_rs_gameplays_game::failed',
14     'test_deterministic_parts.TestRsBetweenGamePlaysAndGame::test_rs_gameplays_game_fail::passed',
15 ]): {
16     'feedback': ""
17 }
18 It seems the relationship sets around GamePlays and Game are not correct. I
19 would expect GamePlays to be a weak entity set to Game. You may also consider
20 how many relationship sets can be there between them.
21
22 For example, the original `happens`/`GamePlaysPlayers` relationship set
23 connects four entity sets. However, connecting PlayerInfo to Game seems
24 redundant, since players are actually involved due to GamePlays.
25
26     """.strip(),
27 },
28 }

```

Listing 26: maps/basic_check/relationship_sets.py

```

1 from socassess import FeedbackLevel
2
3 mappings = {
4     # GamePlays
5     frozenset([
6         'test_attributes.TestGamePlays::test_attrs_in_gameplays::passed',
7     ]): {
8         'feedback': ""
9     }
10    Nice! The GamePlays attributes seems to be reasonable.
11
12    """.strip(),
13    },
14    frozenset([
15        'test_attributes.TestGamePlays::test_attrs_in_gameplays::failed',
16        'test_attributes.TestGamePlays::test_attrs_in_gameplays_fail::passed',
17        'test_attributes.TestGamePlays::test_position_in_gameplays::passed',
18    ]): {
19        'feedback': ""
20    }
21    Oops! You might want to move `position` elsewhere outside GamePlays.
22
23    """.strip(),
24    'level': FeedbackLevel.MEDIUM
25    },
26    frozenset([
27        'test_attributes.TestGamePlays::test_attrs_in_gameplays::failed',
28        'test_attributes.TestGamePlays::test_attrs_in_gameplays_fail::passed',
29        'test_attributes.TestGamePlays::test_secondaryType_in_gameplays::passed',
30    ]): {
31        'feedback': ""
32    }
33    Oops! You might want to move `secondary`/`secondaryType` elsewhere outside GamePlays.
34
35    """.strip(),
36    'level': FeedbackLevel.MEDIUM
37    },
38    frozenset([
39        'test_attributes.TestGamePlays::test_attrs_in_gameplays::failed',
40        'test_attributes.TestGamePlays::test_attrs_in_gameplays_fail::passed',
41        'test_attributes.TestGamePlays::test_goals_in_gameplays::passed',
42    ]): {
43        'feedback': ""
44    }
45    Oops! You might want to move `goals` elsewhere outside GamePlays.
46
47    """.strip(),
48    'level': FeedbackLevel.MEDIUM
49    },
50    frozenset([
51        'test_attributes.TestGamePlays::test_attrs_in_gameplays::failed',
52        'test_attributes.TestGamePlays::test_attrs_in_gameplays_fail::passed',
53    ]): {
54        'feedback': ""
55    }
56    The attributes in GamePlays do not match my expectation. It could a spelling

```

```
57 issue (please check the assignment pdf) or an issue with composite attributes
58 (please make sure that all composite attributes contain more than one
59 attributes).
60
61     """.strip(),
62 },
63 }
```

Listing 27: maps/basic_check/attributes.py

```
1 """Provide mappings between test outcomes and feedback messages."""
2
3 from . import common_design, unknown_design
4
5 mappings = \
6     common_design.mappings \
7     | unknown_design.mappings
```

Listing 28: maps/detail_check/__init__.py

```

1 mappings = {
2     frozenset([
3         'test_reasonable_design.TestDesign1::test_design1::passed',
4     ]): {
5         'feedback': ""
6
7     Nice! It seems your relationship sets and specializations are reasonable
8     between TeamInfo, PlayInfo, and ExecutablePlays.
9
10        """.strip(),
11    },
12    frozenset([
13        'test_reasonable_design.TestDesign2::test_design2::passed',
14    ]): {
15        'feedback': ""
16
17    Nice! It seems your relationship sets and specializations are reasonable
18    between TeamInfo, PlayInfo, and ExecutablePlays.
19
20        """.strip(),
21    },
22    frozenset([
23        'test_reasonable_design.TestDesign1::test_design1::failed',
24        'test_reasonable_design.TestDesign1::test_design1_fail::passed',
25        'test_reasonable_design.TestDesign1::test_design1_extra_NonExecutablePlays_1::passed',
26    ]): {
27        'feedback': ""
28
29    For specializations of GamePlays, it seems you have NonExecutablePlays as a
30    specialized entity set of GamePlays, and GameChallenges as a specialized
31    entity set of NonExecutablePlays. However, you don't have any relationship set
32    around the NonExecutablePlays. Do you really need the NonExecutablePlays?
33
34        """.strip(),
35    },
36    frozenset([
37        'test_reasonable_design.TestDesign1::test_design1::failed',
38        'test_reasonable_design.TestDesign1::test_design1_fail::passed',
39        'test_reasonable_design.TestDesign1::test_design1_extra_NonExecutablePlays_2::passed',
40    ]): {
41        'feedback': ""
42
43    For specializations of GamePlays, it seems you have NonExecutablePlays as a
44    specialized entity set of GamePlays, and GameChallenges as a specialized entity
45    set of NonExecutablePlays. However, you don't have any relationship set around
46    the GameChallenges. Given that GameChallenges is the one addressing one of the
47    problems and NonExecutablePlays is not, maybe think about if you really need
48    the NonExecutablePlays?
49
50        """.strip(),
51    },
52 }

```

Listing 29: maps/detail_check/common_design.py

```

1 from socassess import FeedbackLevel, userargs
2
3
4 def fill_all_tables_contents():
5     """Fill {content}.
6
7     content contains the encoding of student's design
8
9     """
10    # The file name is determined in `assessment_tests`
11    return (userargs.artifacts / 'design.txt').read_text()
12
13
14 mappings = {
15     frozenset([
16         'test_reasonable_design.TestDesign1::test_design1::failed',
17         'test_reasonable_design.TestDesign2::test_design2::failed',
18         'test_reasonable_design.TestDesign1::test_design1_fail::passed',
19         'test_reasonable_design.TestDesign2::test_design2_fail::passed',
20     ]): {
21         'feedback!': ""
22     }
23
24     Your relationship sets and/or specializations seems not common. However, it
25     does not mean it is invalid. Can you make a Piazza post with the following
26     context copy-pasted?
27
28     {content}
29     """
30     .strip(),
31     'function': fill_all_tables_contents,
32     'level': FeedbackLevel.LOWEST - 1
33 }

```

Listing 30: maps/detail_check/unknown_design.py

C.3 Quality Assurance on Assessment Code

```
1 from pathlib import Path
2
3 import pytest
4
5
6 @pytest.fixture(scope="session")
7 def expected_filename() -> str:
8     """Return the expected file name of the solution file."""
9     return 'erModel.sql'
10
11
12 @pytest.fixture(scope="session")
13 def assessment_dir(pytestconfig) -> Path:
14     """Contains the path to the expected assessment folder.
15
16     `pytestconfig.rootdir` is the folder containing `pytest.ini`.
17
18     """
19     return pytestconfig.rootdir / '..' / 'assessment'
20
21
22 @pytest.fixture(scope="session")
23 def feedback_dir(pytestconfig) -> Path:
24     """Contains the folder path to recorded feedback."""
25     return pytestconfig.rootdir / 'data' / 'recorded_feedback'
26
27
28 def pytest_generate_tests(metafunc):
29     """Parametrize `one_answer` for `cproc`.
30
31     Each `one_answer` is a file inside the `tests/data/representative_answers`
32     folder. We iterate files in that folder and create a test for each file in
33     it. The parametrization is dynamically done.
34
35     """
36     if "one_answer" in metafunc.fixturenames:
37         rootdir = metafunc.config.rootdir
38         answerdir = Path(rootdir / 'data' / 'representative_answers')
39         files = list(answerdir.iterdir())
40         # one_answer is a full path, so we use `one_answer.name` to get only
41         # the file name, e.g., 1.sql, 2.sql, etc.
42         metafunc.parametrize(
43             "one_answer",
44             [
45                 pytest.param(one_answer, id=one_answer.name)
46                 for one_answer in files
47             ]
48         )
```

Listing 31: tests/conftest.py

```

1  """Test automated feedback generation.
2
3  This module can only be invoked in the folder which is the common parent of the
4  folder containing assessment code and the folder containing regression tests.
5  In this case, this module needs to be invoked at:
6
7  assessment/ # <= `pytest tests` in this folder
8      README.md
9      assessment
10     poetry.lock
11     pyproject.toml
12     tests
13
14  """
15
16  import shlex
17  import shutil
18  import subprocess
19  from pathlib import Path
20
21  import pytest
22
23
24  @pytest.fixture
25  def cproc(one_answer: Path,
26           expected_filename,
27           assessment_dir,
28           tmp_path) -> tuple[str, str, str]:
29      """Invoke CLI without typing it manually.
30
31      During execution, this fixture creates a temporary folder to store
32      artifacts and a copy of the student answer (`one_answer`). The student
33      answer is copied to avoid unexpected modification to the original file.
34      Then it will invoke `socassess` CLI to assess `one_answer`. Since
35      `one_answer` is parametrized, so this fixture will be invoked multiple
36      times.
37
38      The scope of this fixture should be 'function' since we deal with one
39      answer at a time. `one_answer` is parametrized dynamically, see
40      `confstest.py`.
41
42      """
43      cwd = Path.cwd() # get current dir
44      # copy the file to avoid modifying the original file by accident
45      print('processing: ', one_answer.relative_to(cwd))
46      (tmp_path / 'stu').mkdir()
47      shutil.copyfile(one_answer, tmp_path / 'stu' / expected_filename)
48      # similarly, use a temp folder for artifacts since we do not care about
49      # them in these tests
50      artifacts = tmp_path / 'artifacts'
51      # refer to that file for assessment
52      stu = tmp_path / 'stu'
53      assessment_code_tests = Path(assessment_dir / 'assessment_tests').relative_to(cwd)
54      feedback = Path(assessment_dir / 'maps').relative_to(cwd)
55      toml = Path(assessment_dir / 'socassess.toml').relative_to(cwd)
56      cmd = f"""

```

```

57 socassess feedback
58     --artifacts={artifacts}
59     --ansdir={stu}
60     --config={toml}
61     --probing={assessment_code_tests}
62     --feedback={feedback}
63     """.strip()
64     print(cmd)
65     complete_proc = subprocess.run(shlex.split(cmd),
66                                   capture_output=True,
67                                   text=True)
68     return one_answer.name, complete_proc.stdout, complete_proc.stderr
69
70
71 def test_match(cproc: tuple[str, str, str],
72               feedback_dir: Path,
73               pytestconfig,
74               datarecorder):
75     """Ensure feedback for existing answers remain unchanged.
76
77     We use a pytest plugin `pytest-datareorder` for this test. It records the
78     feedback into a record file if the file has been created; otherwise, it
79     compares the content of the file with the feedback being created. If there
80     is a mismatch, the test fails.
81
82     """
83     fname, feedback, error_msg = cproc
84     # record it as a markdown file
85     record_file = feedback_dir / (fname + '.md')
86     datarecorder.record_data(
87         recording_type='txt',
88         recording_file=record_file,
89         data=f"""
90
91     # Representative answer
92
93     `{fname}`
94
95     ---FEEDBACK START
96     {feedback}
97     ---FEEDBACK END
98
99     # Error message if any
100
101     {error_msg}
102     """.strip(),
103     )

```

Listing 32: tests/test_general.py