UC Merced
UC Merced Electronic Theses and Dissertations

Title
Using Apprenticeship and Product Based Learning to Improve Programming Outcomes in Introductory Computer Science Courses

Permalink
https://escholarship.org/uc/item/1x5732n5

Author
Gonzalez Araujo, Giovanni

Publication Date
2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, MERCED

Using Apprenticeship and Product Based Learning to Improve Programming Outcomes in Introductory Computer Science Courses

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Giovanni Gonzalez Araujo

Committee in charge:
Professor Angelo Kyrilov, Chair
Professor David Noelle
Professor Sarah Frey
Professor Shawn Newsam

2024

Copyright ©

Giovanni Gonzalez Araujo, 2024

All rights reserved

The dissertation of Giovanni Gonzalez Araujo is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

David Noelle

Sarah Frey

Shawn Newsam

Angelo Kyrilov, Chair

University of California, Merced
2024

Abstract

This dissertation presents the results and outcomes of an effort to design educational tools and curriculum to improve student learning in introductory programming courses. The work was conducted at the University of California, Merced (UC Merced), situated in the Central Valley of California, and home to a diverse student population. The findings of this research are applicable to courses in other universities, where instructors face similar challenges with high enrollments in courses, and students come from traditionally underserved communities, or are members of underrepresented minorities.

A significant portion of the work in this dissertation was the development of educational technology to support students and instructors in Computer Science courses, as well as researchers in the field. The tools contributed include a plagiarism detection tool based on fine-grained interaction data from students using an online Integrated Development Environment, allowing instructors to observe and detect behavioral patterns consistent with plagiarism.

This tool was used in a case study at UC Merced to discover unexpectedly high plagiarism rates in programming courses. In addition, we found students were spending less time than expected on their programming assignments, which is thought to limit their learning opportunities. Finally, the correlations between laboratory assignment and midterm examination grades were very weak. Many students were demonstrating high proficiency with programming during laboratory sessions, but in midterm examinations they exhibited a lack of understanding of programming concepts.

We attempted to create a curriculum and instructional methodology that would motivate students to work more on their programming assignments, providing them with more opportunities to practice and master the skill. We created a flavor of the popular Project Based Learning philosophy, that we call Product Based Learning, where students were asked to work on projects that resemble real-life software products, with Graphical User Interfaces (GUI), and a lot of room for creativity on their part. Unlike traditional lower-division programming assignments, the Product Based exercises are always graded by a human grader. We believe this additionally motivates students to put in more effort, as they know their products will be seen (and used) by other people.

To support the increased grading workload, we developed an online system to streamline the grading process for instructors, by automating as many of the mundane tasks as possible. It has the plagiarism detection tool we developed built in, allowing

instructors to not only see the final state of the product, but also replay the creative process the student employed while building it.

The teaching workload also increases, as there is additional material related to GUI development that needs to be covered by instructors. We adopted an Active Learning methodology inspired by the Apprenticeship Learning Model, where the instructor demonstrates how to build the software products. Apprenticeship learning is often more successful in lower apprentice to expert ratios. Too support the process at larger scales, we created an Interactive Code Rewind tool, where the instructor encodes some of their knowledge and thought process in a version controlled Git repository, that the students can then review efficiently, and get help directly from the tool when needed.

The curriculum and teaching methods described above were deployed in the same introductory programming course at UC Merced, where incoming students had the same experience as before. We repeated the experiments from the previous case study, and found that students were spending an average of 357 minutes a week on their programming assignments, up from 89 minutes a week. We found programming behaviors indicative of plagiarism for 6% of students submissions, down from 48%. Finally, the midterm exam grades went from an average of 68% to 75%, while average laboratory assignment grades dropped from 96% to 80%, which is a better correlation between practical and formal exam grades. The three observations taken together can be interpreted as improved learning rates for students in the course.

This work reported in this dissertation lays the foundations for promising research directions that can improve student learning of computer programming, especially for underserved populations with limited resources.

To my wife.

Acknowledgements

First and foremost, I am extremely grateful to my advisor, Angelo Kyrilov, for his expert guidance and continuous support throughout my Ph.D. journey. He has always been available to provide advice, feedback, and encouragement, no matter how busy his schedule was. Angelo's insights and expertise have been invaluable, and I am incredibly fortunate to have had the opportunity to learn from him. His dedication to my growth as a researcher and as an individual has profoundly shaped my academic and personal development. Angelo's enthusiasm and passion for teaching are an inspiration to me. His support has extended beyond the academic realm, and for that, I am truly grateful. I want to thank him for believing in me, for pushing me to reach my full potential, and for being an exceptional role model. It is with immense honor that I look forward to continuing to work with Angelo as colleagues. I am forever indebted to him for his guidance and support.

I would also like to thank the other members of my committee: David Noelle, Sarah Frey, and Shawn Newsam. They went above and beyond in reading my dissertation, even on short notice, providing incredibly detailed and insightful feedback that significantly strengthened my work. Their encouragement, understanding, and willingness to take time out of their busy schedule to help is truly appreciated. I am profoundly grateful of the time and effort they invested in ensuring the success of my dissertation.

I would like to thank my parents, Hugo and Carolina, for all the sacrifices they made to bring me to this country in hopes of a better life. Their tireless dedication, selfless love, and unwavering support have been truly invaluable and deeply appreciated. I want to thank them for always believing in me, I hope I have made them proud.

I also want to thank Michael Kyrilov for all the technical help in deploying the educational tools described in this dissertation.

Last but not least, I want to thank my wife Karely Gonzalez. Her patience, love, and constant support have been my anchor during this challenging journey. She stood by me during the late nights, the long weekends, and the times when I felt overwhelmed. I want to thank her for taking on more than her fair share of responsibilities at home, allowing me to focus on my research. Her sacrifices and understanding made it possible for me to dedicate the time and energy needed to complete this dissertation. This achievement is as much hers as it is mine, and I am deeply appreciative of everything she has done to help me reach this milestone.

Contents

1 Introduction **1**

2 Background and Related Work **7**

2.1 Plagiarism Detection and Prevention 7

2.1.1 Fraud Triangle 8

2.1.2 Similarity Based Detection 9

2.1.3 Detection Based on Programming Behavior 11

2.2 Automated Assessment in Programming 14

2.3 Innovative Instructional Methods 15

2.3.1 Live-Coding Demonstrations 15

2.3.2 Peer Instruction 16

2.3.3 Project Based Learning 17

2.3.4 Flipped Classroom 18

2.4 Apprenticeship Learning Model 19

3 Educational Software Tools **22**

3.1 Learning Management System 22

3.1.1 Lecture Materials Widget 23

3.1.2 Assignments Widget 24

3.1.3 Library Widget 26

3.1.4 Slides Widget 27

3.1.5 Fine-Grained System Logs 27

3.2 Plagiarism Detection Tool 29

3.2.1 Data Processing Module 29

3.2.2 File History Viewer 33

3.2.3 Cursor Position Plots 34

3.2.4 Event Timing Graphs 35

3.2.5 File Analytics . . . 36
3.3 Grading Platform . . . 37
3.4 Interactive Code Rewind Tool . . . 39

4 Case Study: An Introductory Programming Course 41
4.1 Introduction . . . 41
4.2 Student Population . . . 42
4.3 Course Description . . . 43
4.4 Average Formal Exam and Laboratory Scores . . . 45
4.5 Time Spent on Programming Assignments . . . 46
4.6 Plagiarism on Programming Assignments . . . 47
4.7 Conclusion . . . 48

5 Apprenticeship and Product Based Learning 49
5.1 Product Based Learning . . . 50
5.2 Apprenticeship Learning . . . 53
5.2.1 Lecture Delivery . . . 53
5.2.2 Interactive Lecture Code Rewind . . . 54
5.2.3 Self Guided Supplementary Exercises with Solutions . . . 57
5.2.4 Product Based Programming Assignments . . . 59
5.3 Evaluation . . . 59
5.3.1 Increased Time On Task . . . 61
5.3.2 Decreased Plagiarism in Programming Assignments . . . 62
5.3.3 Improved Learning Rates for Programming Assignments . . . 62

6 Conclusion and Future Work 65
6.1 Dissertation Summary and Discussion . . . 65
6.2 Future Work . . . 69
6.2.1 Automated Plagiarism Detection with Fine-Grained Data . . . 69
6.2.2 Elimination of Formal Exams . . . 70

Appendix A Data Sets 71
A.1 Programming Assignments . . . 71
A.1.1 CSE 24 - Spring 2022 . . . 71
A.1.2 CSE 24 - Spring 2024 . . . 73

Bibliography 74

List of Figures

3.1 The Learning Management System interface showing basic IDE features 23
3.2 The Lecture Materials Widget interface 24
3.3 The Assignments Widget interface showing exercise instruction 25
3.4 The Assignments Widget interface showing submission history for a student 25
3.5 The Library Widget interface 26
3.6 The Slides Widget interface 27
3.7 File History Viewer at first interaction (bootstrap event). 33
3.8 The File History Viewer interface. This example shows that interaction event 773 was the insertion of a semicolon at the end of line 21 in the file. 34
3.9 Cursor position plots for a student coding session. 35
3.10 Timing events for a student coding session. 36
3.11 File analytic events for a student coding session. 37
3.12 The Grading Platform interface showing the submission for student 107. 38
3.13 A flowchart of the Interactive Code Rewind tool. 39
3.14 The Interactive Code Rewind interface for `lecture_9` at commit 3, showing the notes and a diff editor with content that changed. 40

4.1 Student distribution according to race 42
4.2 National ACT college readiness benchmark attainment by annual family income 43
4.3 Average laboratory grades compared to midterm grades in Spring 2022. 45
4.4 Scatter plots of laboratory and midterm grades in Spring 2022. 46

5.1 Screenshot of GUI based “Hello World” program 52
5.2 Source code of GUI based “Hello World” program written in C++ . . 52

5.3 Interactive Code Rewind tool for `lecture_7` at commit 2. Includes instructor notes and diff editor with content that changed. 56
5.4 Interactive Code Rewind tool showing the output of running the program from `lecture_7` as it existed in commit 2. 56
5.5 The Interactive Code Rewind interface showing *solution* branch for self guided programming exercise. 58
5.6 Student solutions for Product Based Learning paint application assignment. 60
5.7 Average laboratory grades compared to midterm grade in Spring 2022 and Spring 2024 63
5.8 Scatter plots of laboratory and midterm grades for Spring 2022 and Spring 2024 64

List of Tables

3.1 Table representing the system logs stored in BigQuery. 28
3.2 Example File Changesets Table. 31

4.1 Time in minutes spent by students on programming assignments in CSE 24 for Spring 2022 47

5.1 Average time in minutes spent by students on programming assignments in Spring 2022 and Spring 2024 61
5.2 Summary of comparisons between 2022 and 2024 course offerings . . . 64

6.1 Machine learning model setup 69
6.2 Dynamic features extracted from every coding session 70

Chapter 1

Introduction

The work presented in this dissertation was a multi-year effort to design curriculum and instructional methods that improve student learning in introductory programming courses. The work was carried out at the University of California, Merced (UC Merced), situated in the heart of the San Joaquin Valley. UC Merced, a Hispanic Serving Institution (HSI), is home to a diverse student population, many of whom are members of underrepresented minorities, coming from underserved communities, oftentimes the first in their family to attend college. UC Merced is a suitable setting for efforts related to Broadening Participation in Computing (BPC), as the Computer Science student population faces similar challenges to Computer Science students from other underserved areas.

The first phase of the work was to evaluate the effectiveness of an introductory programming course at UC Merced: CSE 24 - Advanced Programming. This is the second course in the typical CS1 - CS2 sequence, focused on teaching students the process of writing programs to solve computational problems of increasing difficulty and complexity. In terms of curriculum, CSE 24 is similar to introductory Computer Science courses at other institutions, as it is articulated to courses offered at all other University of California campuses, all California State University campuses, and all Community Colleges in California, among other institutions. CSE 24 is delivered in a traditional format, with a large practical component, and formal examinations.

A pattern that emerged early on was that despite completing the programming courses, students were not developing adequate programming skills. This puts them at a disadvantage in their future Computer Science courses, and possibly in the job market, where a candidate must demonstrate strong programming and problem solving skills during the interview process. We compared the grades our students earned

in practical laboratory assignments to the grades they earned in midterm examinations and found low correlations. We included questions in the midterm examinations asking students to essentially reproduce parts of their programming assignment solutions. Many of the students who lost points on these questions showed no evidence of any programming ability, raising the question of how they were able to successfully complete their programming assignments.

Plagiarism is a well documented phenomenon in Computer Science Education, and has been a problem in programming courses for as long as they have existed [104, 127]. Unfortunately, in recent years it has become easier than ever for students to plagiarize solutions to programming assignments. Complete solutions to simple (traditional) programming exercises are available on websites, usually one Google search away. Some students make use of online forums, like Stack Overflow, to ask technical questions and receive expert assistance, while others make use of platforms like Chegg, which specifically allow students to upload homework questions and get complete solutions. This is counter-productive to learning as students are simply turning in the work of others as their own. With the latest advancements in Artificial Intelligence (AI), students now have access to Large Language Models (LLMs) that can successfully solve most (if not all) programming assignments from introductory programming courses [47]. These platforms and tools allow students to commit plagiarism without having to obtain the solution of a classmate, thereby evading traditional plagiarism detection tools that rely on a measure of software similarity between submissions of different students in a class.

In order to properly quantify the levels of academic dishonesty in programming assignments, it is necessary to be able to observe the entire process the student undertook to develop their solution, not only its final state at the time of submission. To accomplish this, we adopted an online Integrated Development Environment (IDE), that students can access through a web browser, and use to complete all their programming tasks. Since the system is online, it automatically saves all text entered into it at every keystroke, for each individual student. By examining system logs, we are able to reconstruct the entire programming process of every student as they were generating their solution to a programming exercise, including the source code they wrote, as well as the terminal commands they used to compile, run, and test their code. We built a tool that made it possible to observe dishonest patterns, such as pasting in complete solutions, copying code by manually typing it out, oftentimes without making any mistakes, editing existing lines of code, or compiling/running

the program. Our analysis of these fine-grained data revealed that 48% of students exhibited suspicious patterns consistent with plagiarism. We also submitted all solutions to MOSS, the most widely used plagiarism detection tool in Computer Science education, and it only flagged half the cases that we suspected, as the other half were not similar to one another.

Imposing stricter punishment for offenders has been shown to be an ineffective plagiarism deterrent. We suspect that at least in part, students commit plagiarism because the programming exercises they are asked to complete are not interesting or motivating enough. Project Based Learning has been reported to improve learning outcomes and increase motivation for students as they are asked to work on real-life projects they find interesting [74]. Project Based Learning strategies have been successfully implemented in Computer Science capstone courses, which lend themselves well to the methodology, but adoption in lower division courses is more difficult.

We adopted a flavor of Project Based Learning, where the projects that students work on always take the form of software products. The specifications for these products usually outline a set of required features, with a lot of room for student creativity. All products are required to have a Graphical User Interface (GUI), and should resemble real-life software products that are used by real users. Students develop a software product over the course of several laboratory sessions, building on their previous work, but receiving feedback at intermediate stages. We call this instructional methodology Product Based Learning.

With the introduction of Product Based Learning we anticipated in increase in student motivation and effort, which we expected to translate to improved learning, but we also recognized that there would be an increased labor cost for both course instructors and teaching assistants (TAs). Course instructors would have the additional burden of covering more extensive material related to GUI programs. As a simple example, a "hello world" program is 1-5 lines of code, depending on the programming language, whereas a GUI equivalent requires more lines of code and oftentimes employs more advances programming topics. In addition, GUI-based, open-ended programs are not as amenable to automated assessment as command line programs following a strict specification. This would lead to an increase in TA workload, or take time away from their other activities such as leading discussion, or office hours.

Since Product Based Learning requires instructors to cover additional material in the same amount of lectures as before, we sought to make better use of lecture time, through the adoption of Active Learning methods, including live coding demonstra-

tions, and Peer Instruction activities. Our initial attempts were not successful and students were failing the in-class exercises and assignments because they could not internalize the material after seeing it earlier in the lecture. To remedy this situation we started recording the live coding demonstrations, so students could revisit the explanations if they needed to at a later time. This led to the development of the Git-based "Interactive Code Rewind" tool, incorporated into our online IDE, which allows students to revert the source code of the lecture demonstration to an earlier point in time, and interact with the code as it existed at that point. The increased labor cost for the instructor amounts to invoking a "commit" command at various points during the course of the demonstration. Optionally, the instructor can provide a note for the commit, which can be edited at a later stage. This allows the instructor to encode their thought process and problem solving strategies into the demonstrations, and provide stopping points at natural milestones. This allows the students to experience the demonstration as many times as necessary, at their own pace. The design was inspired by the Apprenticeship Learning Model, which is widely used in vocational training, where an apprentice learns a skill by observing the expert, initially mimicking their actions, before starting to work more independently, and finally generalizing the skill. Since the Interactive Code Rewind tool is based on the Git version control system, it can also be used to share code with students during lecture in real time. This allows us to make lectures more interactive and engaging for students.

We also attempted to mitigate the increased grading workload for Product Based assignments by building an online grading system that automates the mundane tasks associated with grading programming assignments. Since all assignments have GUIs, our grading system is capable of compiling and running graphical applications inline, so there is no need for TAs to install any compilers, and GUI libraries in order to be able to compile student submissions. The fine-grained data collected by the online IDE is also useful in the grading process, so the entire grading system runs on top of our plagiarism detection tool, with the added capability of entering grades and feedback comments directly into the course grade book. The text entry visualization tool we used to detect plagiarism can also be used to follow along with the thought process of the student as they were building their software product, which is helpful in grading. Seeing how the code comes about is easier for the TA rather than having to read the entire source code in its final state, allowing the TA to get a better idea of the student's skill level more quickly and efficiently. If the TA suspects plagiarism,

they can flag the submission for further investigation by the instructor. The abilities described above, and the convenience features of the grading system have resulted in students receiving feedback on their programs in a timely manner, allowing them to continue work on their products in subsequent laboratory sessions.

We believe the introduction of Product Based Learning, in combination with aspects of Apprenticeship Learning has resulted in a measurable reduction in plagiarism. This is likely due to a combination of factors including the graphical user interfaces and open-ended nature of the assignments being more interesting and naturally more motivating for students. We also believe that students feel more ownership of the products, as they are longer term projects, developed over the course of several weeks, rather than simple throw-away scripts. It could also be that since the Product Based assignments are still new, with no existing solutions available from prior course offerings, plagiarism rates were naturally low. Further research would confirm this, but in the worst case scenario, the labor cost of coming up with completely new Product Based assignments is relatively low compared to traditional, automatically graded exercises. This is because the open-ended nature of the projects requires little explanation, and there is no need for generating test cases to be used in automated assessment. The visual nature of the programs also makes them easier to grade compared to traditional command line interfaces.

We have noticed a significant increase in the amount of time students are spending on their weekly programming exercises, and the amount of source code they generate, compared to prior course offerings that had no Product Based assignments. While this is not a reliable measure of effort and engagement, we expect that by spending more time on task, and writing more code, students are practicing their programming skills more, which could translate into improved learning rates. To confirm this, we repeated the experiment from prior course offerings where we included questions in the midterm that ask students to reproduce aspects of their practical laboratory exercises. In comparing the laboratory assignment grades to the midterm grades, we now see a correlation, meaning students who did well in their laboratory assignments are better able to demonstrate that knowledge under test conditions, which could be indicative of improved learning rates.

The rest of the dissertation is organized as follows. Chapter 2 presents relevant background information on plagiarism detection and prevention, automated assessment in programming assignments, and innovative instructional methods in Computer Science. Chapter 3 describes the educational software tools we developed to

help with course delivery. Chapter 4 presents a study on plagiarism in our practical programming component, as well as the engagement levels and effectiveness of our programming assignments. Chapter 5 describes our efforts in redesigning our course curriculum and programming assignments to follow the Apprenticeship Learning Model and Product Based Learning, as well as a study to determine the effectiveness of our new curriculum. Chapter 6 contains some concluding remarks and future work.

Chapter 2

Background and Related Work

In this chapter we review prior work in literature relating to common problems which plague undergraduate Computer Science Education, namely high rates of plagiarism, low pass rates, and the implications of automated assessment for programming assignments. We present information on innovative instructional methods to improve teaching and learning outcomes in Computer Science Education. We review popular plagiarism detection tools used by undergraduate Computer Science instructors. Lastly, we review the Apprenticeship Learning Model and outline the challenges of adoption in the Computer Science classroom.

2.1 Plagiarism Detection and Prevention

There is abundant research evidence summarizing that academic dishonesty is widespread among students [40, 70, 97, 98, 156]. Most relevant to this work, many studies suggest that Computer Science is the discipline with the highest cheating rates. High rates of plagiarism in Computer Science have been documented since the 1970's [104, 127].

In 1993, Lipson reports on a study performed at MIT regarding undergraduate academic dishonesty [91]. When participants were asked if cheating was more likely to occur in certain disciplines, 75% of students stated it was more likely to occur in subjects for which there are regularly organized and updated compilations of old homework assignments, quizzes and exams. Additionally, 50% of students believed that cheating was more likely to occur in computer programming subjects. Other references support this claim and discuss factors which contribute to higher academic dishonesty rates in Computer Science compared to other fields [119, 129, 132].

Roberts [119] found that Computer Science courses at Stanford University account

for the majority of the academic dishonesty cases, some 20 to 54 percent of the total cases, depending on the year. Stanford is not alone, at MIT in 1991, 73 students out of 239 were disciplined for academic dishonesty in an introductory Computer Science course [22]. There are various other news reports and studies [8, 15, 79, 101, 103, 110] that show the prevalence of plagiarism in Computer Science, and [24, 64, 88, 153, 159] confirm that this problem is both widespread and international.

2.1.1 Fraud Triangle

Albluwi [9] performs a systematic review of work in the computing education literature on plagiarism. They review and categorize papers according to the field of Fraud Deterrence named the Fraud Triangle [25, 41]. The Fraud Triangle is a framework which is used to explain the reason or reasons behind an individual's decision to commit fraud. According to the theory, fraudulent behavior is affected by three elements, or sides of a triangle: *opportunity*, *pressure*, and *rationalization*. In this case, we can define plagiarism as the fraudulent behavior and apply the Fraud Triangle framework as follows:

Opportunity: This is the first leg of the Fraud Triangle, and it refers to the circumstances that allow fraud to occur. Examples of plagiarism opportunities include readily available solutions to assignments, recycled assignments, weak consequences for offenders, and the lack of plagiarism detection tools [9]. There is a significant amount of research in attempting to reduce the *opportunity* to cheat. These efforts fall under three categories: designing assignments that are difficult to plagiarize [53, 134], using grading methods that make plagiarism more difficult [40, 45, 62, 72, 19] and using tools to detect plagiarism [6, 113, 56].

Pressure: This is the second leg of the Fraud Triangle, and it refers to the individual's incentive towards committing fraud. Sheard et al. [129, 130] found that time pressure, workload pressure, difficulty of the work, and fear of failure are among the top factors that contribute to student plagiarism. Additionally, Kyrilov and Noelle [86] studied the effects of binary automated feedback on cheating and found that students who received binary feedback were twice as likely to cheat as those who did not receive any feedback until after the assignment deadline. Their findings suggest that binary feedback can prompt undesirable behaviors and tempt struggling students to cheat.

Rationalization: This is the third leg of the Fraud Triangle, and it refers to

the individual's justification for committing fraud. Students who plagiarize often resort to rationalizing their behavior so that it fits into their ethical standards, which in turn leads to increased chances of acting dishonestly in the future. Common rationalizations include taking inspiration from other's work [28], taking large chunks of programs and modifying small portions of it [30, 31], and suggesting that it fine to use other's code as long as you understand the code and learn from the assignment [30, 31, 128]. To combat this, educators have proposed educating students regarding the ethics and academic integrity [58, 118, 149], clearly communicate to students which acts are considered dishonest [54, 118, 131, 132, 141], and teaching students how to properly cite code [54, 131].

Fraudulent behavior can be avoided by removing one of the three sides (*opportunity*, *pressure*, or *rationalization*). A majority of the literature on plagiarism focuses on the *opportunity* side of the Fraud Triangle [9]. More specifically, the literature is skewed towards the use of strategies or tools to reduce plagiarism. There are several studies that provide empirical evidence for the effectiveness of using similarity based plagiarism detection tools [16, 20, 65, 139]. Among these tools, the most popular are: MOSS, JPlag, and SIM.

2.1.2 Similarity Based Detection

To combat high rates of plagiarism, educators have been relying on similarity based plagiarism detection software. These plagiarism detection tools work by measuring the similarity between student programs; they typically count and compare program attributes, such as the number of characters, lines, and keywords. More sophisticated tools also take the structure of the program into account. Below are descriptions of the most widely used plagiarism tools by educators.

MOSS (Measure of Software Similarity), is an automatic system for determining the similarity of programs [6]. MOSS was developed by Aiken et al. in 1994 at Stanford University. It supports 26 different programming languages and is provided as a web service that can be accessed via a script. Using MOSS is simple, you just need to supply a list of files to compare, and the analysis of the submitted codes is done remotely on a server at Stanford University. The MOSS server produces an HTML page listing the pairs of programs with high similarity, and it also highlights sections of code that are identical. Lastly, MOSS can ignore matches of code that one expects to be shared (libraries or instructor supplied code), thereby eliminating

false positives that arise from legitimate sharing of code. MOSS uses a document fingerprinting algorithm known as winnowing [125]. Document fingerprinting works by dividing a document into k contiguous substrings, called k-grams, and computing a hash value for each one. The document fingerprint is a subset of the k-gram hashes. The value of k is typically controlled by the user.

JPlag is a system that finds similarities among multiple sets of source code files [113]. JPlag was developed in 1996 by Guido Mahlpohl and others, at Karlsruhe Institute of Technology. It started out as a research project in 1996, but was later developed into an online system. It supports 12 programming languages, and it is provided as a web service [78]. JPlag presents its results in an HTML page, listing the most similar pairs of programs. Additionally, their clustering of pairs makes it easier to see whether certain submissions are similar to several other submissions. JPlag uses an optimized version of Michael Wise's *Greedy String Tiling* algorithm [160]. JPlag works by taking all the programs that need to be compared and converts them into token strings. Then, these token strings are compared, in pairs, for determining the similarity of each pair. During each comparison, JPlag attempts to cover a token string with a substring taken from another program, and the percentage of token strings that are covered determines the similarity score.

SIM is a software similarity tester developed by Dick Grune in 1989 at the VU University Amsterdam [56, 60]. It supports 8 programming languages, and is provided as a command line tool. SIM takes two source files and passes them through a lexical analyzer to produce a compact form of its structure in the form of a stream of integers, known as *tokens*. The tokens for the programming language keywords, special symbols and comments are predefined, while the tokens for identifiers are assigned dynamically from a shared symbol table. The whitespace is discarded. The purpose of tokenizing the source files is to remove any unwanted information, such as whitespace and comments, effectively producing a parse tree of the programs. After the tokenization of the source files, SIM takes the *tokens* of the second source file and groups them into sections, each representing a module of the program. Each module is then aligned with the token stream of the first source file separately, this allows SIM to detect similarity even when modules between the programs have permutations.

Ahadi et al. [5] performed a comparison on the three most popular similarity based plagiarism detection tools described above. Based on their findings, SIM seemed to have the highest precision, while JPlag was the most sensitive tool, but suffered from low precision. MOSS and SIM appeared to be the most promising tools for identi-

fying instances of plagiarism. However, SIM identified a larger number of potential plagiarism cases compared to MOSS. The authors suggest using SIM first to get a list of potential plagiarism cases and then running MOSS through this list to further reduce the cases and perform manual inspection.

Although the similarity based plagiarism detection tools mentioned above report good performance and good agreement with how educators judge similarity [52], these tools all share a common limitation. By understanding how these tools work and detect similarity, exploits can be performed to successfully bypass these tools and avoid detection. However, these similarity based plagiarism detection tools work on the premise that the cost of defeating the tool must be high. In other words, the difficulty of evading detection will take the same amount of effort, if not more, than completing the assignment honestly.

A recent study showed that bypassing similarity based plagiarism detection tools is not as difficult as conventional wisdom states Devore-McDonald et al. [38] present *Mossad*, an automated program transformation tool used to bypass plagiarism detection tools. The authors focus on MOSS, since it is the most popular plagiarism detection tool. They performed an in depth analysis on MOSS, and identified a key weakness to exploit, specifically, the hashing and winnowing approach it uses. Given an input file and a target similarity, *Mossad* uses genetic programming transformation techniques to introduce benign statements within a program to produce semantically equivalent variants that are less than or equal to the target similarity provided. The authors demonstrate that *Mossad* is fast and effective in defeating four plagiarism detection tools, including MOSS and JPlag, two of the most widely used tools by educators. In addition, they performed a study which revealed that the program variants produced by *Mossad* are just as readable and no more suspicious than legitimate programs produced by students.

2.1.3 Detection Based on Programming Behavior

Despite the popularity and high accuracy of similarity based plagiarism detection tools, they are unable to detect other common forms of plagiarism, such as applying transformations, as described above, or outsourcing, where students obtain an original solution generated by an expert, and present it as their own. To address these limitations, educators have proposed analyzing student's code at different stages of development, rather than just the final state of the submission.

Tahaei et al. [144] proposed a plagiarism detection method based on the sequence of submissions made by an individual student. The authors calculated submission difference using a common diff algorithm [71]. After preprocessing two consecutive submissions, the submission difference is simply the minimum number of line additions and deletions needed to transform one file into the other. The authors found that the number of submissions paired with the maximum difference between consecutive submissions resulted in the greatest accuracy for measuring the likelihood of plagiarism. Using these two features, they were able to get a probability score by using logistic regression. They applied this method to data from four exercises from an undergraduate programming course and their results strongly correlated with the assessments of plagiarism made by an instructor. Comparing their results to MOSS, they concluded that the patterns of resubmissions made by an individual student can be more predictive of plagiarism, compared to similarity measures performed on final submissions across students in a course. However, one major weakness of their method is students who make only a single submission.

Yan et al. [162] developed *TMOSS* (Temporal Measure of Software Similarity), a tool that analyzes the intermediate steps a student takes to complete a programming assignment. *TMOSS* extends the traditional similarity detection software, in this case MOSS, by analyzing student's code at intermediate snapshots, and not just the final code submission. In the study, the authors focus on the first large programming assignment of an undergraduate CS 1 course. The data set includes submissions of the assignment from 3 offerings, all taught by the same instructor: Fall 2012 (416 students), Fall 2013 (476 students), and Fall 2014 (528 students). The students used a modified Integrated Development Environment (IDE) which would take a snapshot of their code and upload to a Git repository every time the student compiled their assignment. After using *TMOSS* with human verification on this data set, the authors found 61 students plagiarized, compared to 35 by only using MOSS with human verification.

Fonseca et al. [55] developed *CodeInsights*, a monitoring tool that captures student performance information based on code snapshots produced by students while they solve their programming assignments. This information is available to the instructor in real time so that they can identify struggling students and intervene in a timely manner. In order to use *CodeInsights*, students only need to download a plugin for their IDE. This plugin will automatically take a snapshot of the source code every time the student runs their program. These snapshots will automatically be

uploaded to a server which compiles and tests the code, and generates information. This information is then used to provide visualizations and preprocessed data for the instructor on *CodeInsights*, which allows the instructor to easily see the performance of their students. Additionally, *CodeInsights* sends a notification to the instructor when it notices situations that can lead to problems, such as unusual number of lines of code, compilation errors, excessive number of attempts per assignment, assignments not attempted, or even unfinished assignments.

Fonseca et al. [48] present a new feature to *CodeInsights* which detects potential occurrences of plagiarism, in real time. The authors focus on the detection of low-level plagiarism, such as copy/paste or similar code with minimal modifications. Since *CodeInsights* already gets code snapshots every time the student compiles their code, the similarity between this new submission and the latest submissions from all other students are compared. A simple string comparison algorithm is used to compare the code snapshots and produce a similarity percentage. *CodeInsights* generates online and offline notifications to alert the instructor of potential plagiarism cases. The authors have seen some encouraging results when using this new plagiarism detection feature, as it allows instructors to identify students facing difficulties, and intervene in a timely manner.

Ljubovic et al. [94] propose a new method of detecting plagiarism that combines software repository mining with similarity based plagiarism detection tools. The authors use a cloud based IDE, to log all activities that students perform while working on their programming assignments, in the form of ultra-fine grained repositories. They then extract dynamic features from the repository that describe the student's behavior and habits while coding. These dynamic features reflect the process of coding, and not just the final result. The authors focus on the following dynamic features: lines of code added, lines of code deleted, lines of code modified, average paste length, max paste length, number of compilations, number of successful compilations, number of unit tests ran, average unit test score, characters per second, time spend coding, small breaks, and large breaks. The authors use a backpropogation artificial neural network that uses the dynamic features to create a binary classifier that outputs a probability in the range [0, 1] representing the probability of plagiarism. After this filtering step, another artificial neural network is trained in a similar way to improve the similarity ranking obtained from SIM, a substring matching plagiarism detection tool. The results show a significant improvement compared to other popular plagiarism detection tools such as MOSS [125], JPlag [113], SIM [61], SIM+NN, and fvpd [105].

2.2 Automated Assessment in Programming

Automated grading works on the premise of extracting a measurement value from a submitted program and comparing it to assignment requirements or a given solution [7]. There are numerous testing techniques at the core of automated grading. Functionality testing determines if a program functions according to the given requirements. This process, widely known as output comparison, involves running the submitted code through predefined test cases and comparing the generated output to the expected results [35, 150, 163, 92]. Another form of functionality testing involves the static analysis of the source code. This oftentimes involves parsing the program and constructing syntax trees [102, 157, 165] and pattern matching techniques which look for certain constructs in the code [66]. Automated grading can also involve the analysis of time complexity [14], and code quality [93, 11], which is designed to uncover programming flaws and bugs such as unused variables, empty catch blocks, and dead code. More modern automated grading tools also include plagiarism detection out-of-the-box [140, 26].

Automated assessment systems provide many benefits for educators and students alike. Educators will spend significantly less time grading programming assignments, allowing them to increase the number of weekly programming assignments, providing students with more practice to improve their programming skills [161]. Studies have shown that automated assessment systems are popular because of their convenience, efficiency, and objectivity [151]. In some cases automated assessment can provide students with feedback which can help students learn from their mistakes. However, most automated assessment systems only provide students with instant binary feedback.

Educators argue that instant binary feedback is not as useful as feedback generated by human instructors [13]. Not only is this limited to a predefined criteria, resulting in lack of personalized feedback [145], some studies have found that it can reduce student engagement and promote cheating [86]. Moreover, automated assessment is not possible when educators assign open-ended programming assignments which allow for student engagement and creativity.

2.3 Innovative Instructional Methods

The lecture based teaching approach is an instructor-centered teaching method in which students passively listen to information presented to them. Lectures are one of the oldest teaching methods, and they are still widely used in education, especially at the college and university level. Not too long ago, these lectures were typically delivered using a combination of a projector, for displaying visuals, and chalkboard or whiteboard for writing. Nowadays, this has largely been replaced by the use of presentation software such as Microsoft PowerPoint, Apple Keynote, or Google Slides, and digital whiteboards.

It is very common for educators to teach introductory Computer Science courses using this traditional lecture based teaching modality, through the use of lecture slide presentations with static code examples, which are snippets of code shown during lecture that students cannot directly access and test [122]. Moreover, these static code examples are generally correctly coded final solutions. Programming involves problem solving, similar to mathematics, where educators generally teach by explaining the theory first and then solving example problems from start to finish. By fully solving and walking through example problems during class, math educators are effectively showing their students the steps, logic, and techniques required to solve the problems. However, when educators use static code examples to teach Computer Science, students miss out on the programming process required to state of the final code.

Since the 1980s, researchers have been proposing Active Learning methods for Computer Science [18], with the promise of increased effectiveness. Active Learning is a student-centered teaching approach in which students actively engage in the learning process by reading, writing, discussing, and problem solving, rather than just sitting and passively listening [18]. There is an abundance of research outlining different active learning strategies which have been implemented in the Computer Science classroom, such as live-coding demonstrations, Peer Instruction, Flipped Classroom, and Project Based Learning.

2.3.1 Live-Coding Demonstrations

Since programming involves problem solving, it has been suggested that educators should focus on teaching the process of programming, and not just the theory, syntax, and knowledge required to program [120]. Rubin [122] proposes live-coding as the primary teaching method for introductory Computer Science. A live-coding demon-

stration is defined as "the process of designing and implementing a coding project in front of class during lecture period" [108]. This requires the instructor to start lecture with a blank text editor and teach by developing, compiling, and testing code. Rubin examined the effectiveness of live-coding demonstrations in introductory Computer Science courses and found that it led to significant increases in student performance on projects. Furthermore, he found that 90% of students in the study agreed that code examples were more educational than traditional lecture slides.

2.3.2 Peer Instruction

Peer Instruction (PI), coined by physics professor Eric Mazur in 1991, is a teaching method which modifies the traditional lecture format into a more evidence based, interactive teaching pedagogy [96, 33]. Peer instruction engages students during class through interactive activities that require students to discuss core concepts with their fellow students. In Peer Instruction, the traditional lecture is intermixed with conceptual questions called "Concept Test", which are used to probe the students' understanding on topics covered during lecture or on assigned pre-class readings. Following a mini-lecture, students are asked to answer a conceptual question individually via flash cards or "clickers". The instructor then asks the students to discuss their response with their neighbors and convince each other that they have the correct answer. Following the discussion, the instructor repeats the question and collects the new responses. Finally, the instructor explains the correct answer and continues with the lecture.

Peer Instruction has been extensively studied in physics. The result of a ten year study at Harvard University found that Peer Instruction dramatically improves student's conceptual and quantitative problem solving, compared to traditional lectures, doubling the normalized learning gains [138]. Peer Instruction has also been adopted in other natural sciences [23, 50, 82]. More recently, Peer Instruction has also been adopted in Computer Science [34, 106, 112, 111, 135, 136, 164]. Studies have shown that Peer Instruction has been valued by students as a positive learning tool [112, 135], valued by instructors as it improves student engagement [112], and results in increased individual learning [111, 136].

2.3.3 Project Based Learning

Project Based Learning is a student-centered pedagogy in which students learn by solving real world problems. This teaching idealogy dates back to the early 1900s, with John Dewey being regarded as the founder of Project Based Learning, as his work focused on the “learn by doing” approach, advocating that learning happens when students interact with real life tasks [39]. Education research has advanced the ideas of Project Based Learning into what it is today, and it has been explored in various contexts and levels of education, ranging from pre-school and primary school to higher education and pre-service teacher trainings [85].

In the last decade, Project Based Learning has become increasingly popular in Computer Science Education, as it promotes collaboration, problem solving, and active learning [117]. Project Based Learning has been widely adopted in capstone courses, where students collaborate with industry on real world projects, gaining practical experience and further developing their technical skills such as communication, teamwork, and self-management [76]. This helps students acquire the hard and soft skills required to become industry ready.

One of the fundamental advantages of Project Based Learning is the increased motivation levels because students are working on interesting, real world projects [74]. The hands-on approach of Project Based Learning also increases engagement, leading to improved learning outcomes, as well as the development of enhanced problem solving and critical thinking skills [32]. Studies have shown that Project Based Learning also encourages cooperation and communication skills [3], improved time management [63] and sense of ownership and autonomy, leading to students taking responsibility of their own learning [85], as well as decreasing the drop-out rate of students in programming courses [44].

Although Project Based Learning yields numerous positive outcomes, the adoption of this active learning methodology is not without difficulties. There are also many challenges of adopting Project Based Learning in Computer Science, such as faculty reluctance, inadequate resources, student evaluation challenges, and uneven workloads for students working in groups [117]. Many institutions do not have the funds and resources required to support Project Based Learning effectively [10]. Instructors are often required to adopt new teaching modalities, oftentimes requiring additional training, professional development, and updating of the curriculum [133]. Evaluating students’ learning outcomes is also more difficult because traditional assessment methods, such as formal exams, may be unable to capture the diverse abilities and

knowledge that students acquire from Project Based Learning [114, 121].

2.3.4 Flipped Classroom

The flipped classroom, also known as the inverted classroom, is an active learning methodology in which events that traditionally occur outside the classroom now occur inside the classroom, and vice-versa [87]. Although there are many different methods to flip the classroom, they all follow the same class model. The flipped classroom model consists of out of class content prepared by the professor. The out of class content includes assigned book readings, videos, scientific papers, podcasts, self assessments, etc. The in class content includes programming exercises, problem based learning, mini lectures, unit testing, etc. This allows the instructor to act as a learning mediator and gives students the ability to engage in active learning during class time. This changes the instruction method to a learner-centered model allowing students to review and learn the content before class, so that during class the topics can be covered more in depth, creating meaningful learning opportunities in the classroom. The use of learning technologies, especially multimedia, has provided students with new opportunities and methods to learn, greatly facilitating the institution of the flipped classroom.

Active learning methodologies are a core component of the flipped classroom. Based on the work of Vujovic, et.al [158], in-class activities for the flipped classroom can be classified into one of the six categories: Problem Based Learning, Project Based Learning, Team Based Learning, E-learning, Lifelong Learning, and Self-directed Learning. Problem Based Learning and Self-directed Learning are among the most popular for designing in class activities, which include examinations, quizzes, and classroom instructions such as presentations and discussions [49, 134], timed programming exercises [77], interactive group activities [59], case studies [4], pair activities such as pair programming [107], and interactive lectures [57].

Many researchers have used the flipped classroom approach with different pedagogical strategies and technologies [17]. Studies on flipping the classroom have yielded two impacts on student learning: the first is improved understanding of the subject studied, and the second is increased student performance [152]. Paschoal [107] observed that a cohort of students in a flipped classroom model gained more knowledge than another cohort of students in a traditional classroom. It was observed that students in the flipped classroom learned more, but they also spent a greater amount

of time to do so. Similarly, Elliott [42] reported notions of increased vigor and interaction in the classroom with the flipped classroom pedagogy. Davies [36] and Stone [142] also reported that flipping the classroom has resulted in large learning gains and positive student attitudes towards learning. Acharya [4] reported that students work in flipped classroom setting was superior compared to the traditional setting, and Day [37] reported significantly higher grades and strong positive attitudes between a cohort of students taught using web based lectures and active learning, compared to traditional lecture cohort.

Numerous studies have shown that flipping the classroom leads to improvement in student satisfaction, motivation, dedication and confidence [42, 46, 77, 81, 90, 95, 123, 137], improvement in learning [27, 59, 95, 107, 115, 124, 146], increased student engagement in the classroom [4, 43, 59, 68, 84, 109, 137, 143, 148], and optimization of class time [4, 43, 69, 95, 107, 143], and enhancement of instructor-student and student-to-student relationships [43, 137].

There are also challenges that arise when flipping the classroom, including significant initial overhead cost, as well as difficulties in class preparation and, in general, it is very time-consuming for instructors [27, 43, 67, 59, 68, 69, 81, 90, 95, 107, 115]. Other challenges include difficulty in keeping the students engaged and motivated [43, 81, 90, 95, 109, 124, 146], scalability for large courses [4, 43, 67, 90, 107, 137], and lack of feedback from students [27, 67, 84].

2.4 Apprenticeship Learning Model

Apprenticeship is the process in which an expert teaches an apprentice a skill or trade through observation and guided practice. It is one of the oldest forms of knowledge transmission and largely accepted as the natural way to learn. In apprenticeship, the learning process can be easily witnessed, as it generally involves learning something tangible which involves physical activity, such as learning to plant and harvest crops, sewing, tailoring cloths, or building furniture. This approach has proved to be highly effective in developing vocational skills and can be seen extensively used plumbing, carpentry, welding, and many more professions.

The Apprenticeship Learning Model has five phases [126]:

- *Modeling* where the teacher demonstrates and articulates all the steps involved in the process of completing a task.

- *Scaffolding* where the learner begins to mimic the actions of the teacher.
- *Fading* where the teacher slowly reduces scaffolds for learners to complete tasks more independently.
- *Self Directed Learning* where the learner is performing the actual task and only seeking assistance when needed from the expert.
- *Generalization* where the learner generalizes what they have learned and they conduct open-ended tasks with minimal to no involvement of the teacher.

The Cognitive Apprenticeship model was proposed by Collins et al. [29] as a method of incorporating traditional apprenticeship in the modern education system; it was first introduced in primary and secondary education to teach reading, writing, and mathematics, and now it has also been extended and applied to a range of disciplines in higher education. There are three important differences between traditional apprenticeship and Cognitive Apprenticeship, outlined by Collins.

- First, in traditional apprenticeship, the task being learned is easily observable. Building furniture, growing crops, or even learning to play a musical instrument, are all tasks which are tangible and the physical processes required to learn are easily visible. However, teaching subjects such as math, reading, writing, and problem solving are not always tangible and observable. In Cognitive Apprenticeship, the expert and apprentice need to deliberately express their thinking and problem solving.
- Second, in traditional apprenticeship, tasks arise naturally, making it easy for the apprentice to understand the subtasks involved in creating the finished product. For example, if an apprentice is learning to create furniture, it is natural for them to understand why the expert is showing them how to cut the wood, design different components, and assemble them into the finished product. However, in education, the curriculum is centered around reading, writing, math, science, and history. Students oftentimes do not understand how the concepts learned in these subjects translate to their everyday life. In Cognitive Apprenticeship, the challenge is to teach the students abstract tasks which are not easily perceivable. It is suggested that teachers express abstract tasks as the building blocks of authentic contexts, helping students understand the relevance of the work.

- Third, in traditional apprenticeship, the derived skills are relevant for the task learned, making unlikely that the apprentice will encounter situations that require transfer of skills. For example, an apprentice learning wood work would not need to know anything related to sewing. However, in education, students are oftentimes required to transfer knowledge across domains. In Cognitive Apprenticeship, the challenge is to provide students with a diverse background of knowledge and practice with a variety of tasks which require students to generalize certain skills across different contexts. It is suggested that teachers introduce a well rounded experience to students so they can better witness the transfer of skills.

Numerous studies have shown that Cognitive Apprenticeship improves student enthusiasm [75, 80, 155, 89], increases interests in programming [147], and improves course pass rates and student performance [12, 83, 100, 154]. Although there are many positive outcomes of Cognitive Apprenticeship in Computer Science, one of the key challenges is scalability. Numerous studies express the difficulty of scaling this approach in settings with high student to teacher ratios [154, 12, 83].

Chapter 3

Educational Software Tools

In this chapter, we present the educational software tools used in our introductory programming courses, as well as the additional software that was developed specifically to support this work. Section 3.1 describes the existing online Integrated Development Environment (IDE) deployed in our courses, with all its tools and features to support programming courses.

The tools that were developed specifically for this dissertation include, a plagiarism detection tool based on fine-grained data, presented in Section 3.2, a Git repository navigation tool, called "Interactive Code Rewind", outlined in Section 3.4, and a specialized grading platform for programming assignments, described in Section 3.3. All tools created for this dissertation were developed entirely by the author.

3.1 Learning Management System

Similarly to other introductory programming courses, we use a Learning Management System (LMS) for CSE 24. It offers a fully-fledged IDE accessible through a web browser, based on Theia, an open-source online platform, feature equivalent to the popular Visual Studio Code. A major benefit of using an online IDE is that it provides a consistent user experience for all students and instructors, and has all the compilers and other development tools preinstalled, eliminating the need for students to administer their own computers.

Figure 3.1 shows a view of the basic IDE features, including a file tree viewer on the left, a text editor with the contents of the selected file, and a terminal emulator, along the bottom of the interface. There are additional developer tools, such as a visual debugger interface, and standard version control tools.

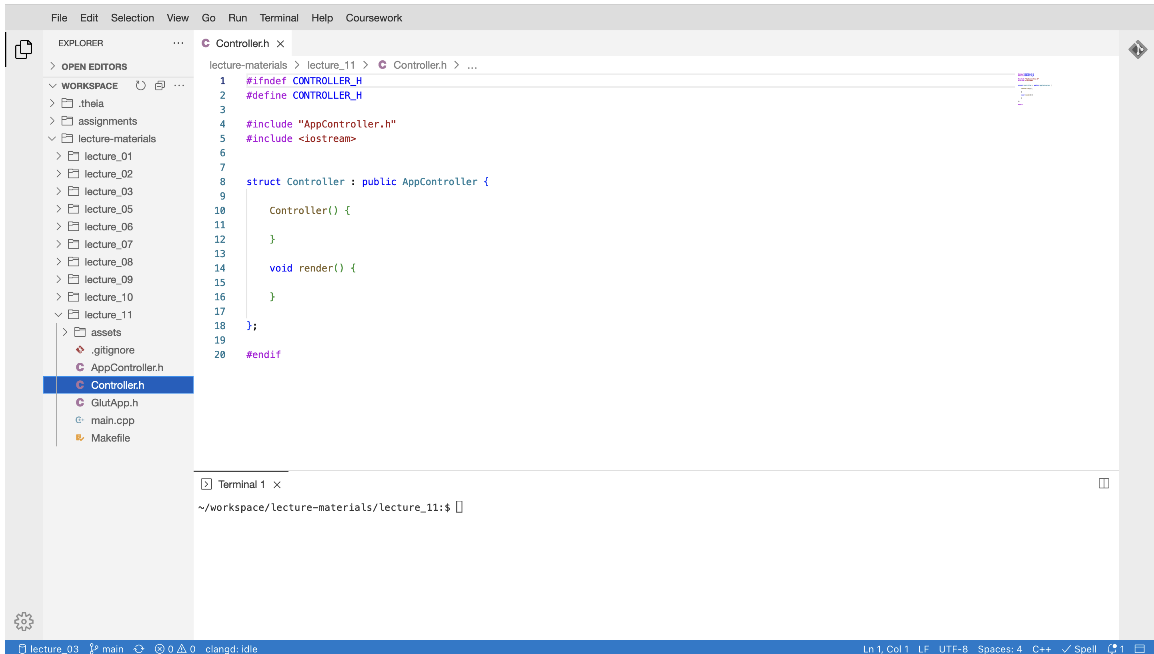


Figure 3.1: The Learning Management System interface showing basic IDE features

In addition to the standard IDE features described above, the system has Learning Management features to support course delivery. They take the form of widgets, accessible through the main interface. Each educational widget is described below.

3.1.1 Lecture Materials Widget

This interface allows the instructor to provide reading materials written in rich text format, as well as support files, which the students can copy to their workspace, with a button click. Support files typically include boilerplate code to be used as a foundation for a programming demonstration.

Figure 3.2, shows the interface of a student who has navigated to "Lecture 3: User Events in OpenGL Application" and has bootstrapped the lecture materials. The lecture readings and materials are displayed in the main portion of the window. In the file tree viewer, visible on the left sidebar of the interface, the system has created the appropriate `lecture_03` folder (highlighted in red) and copied the necessary support files into it.

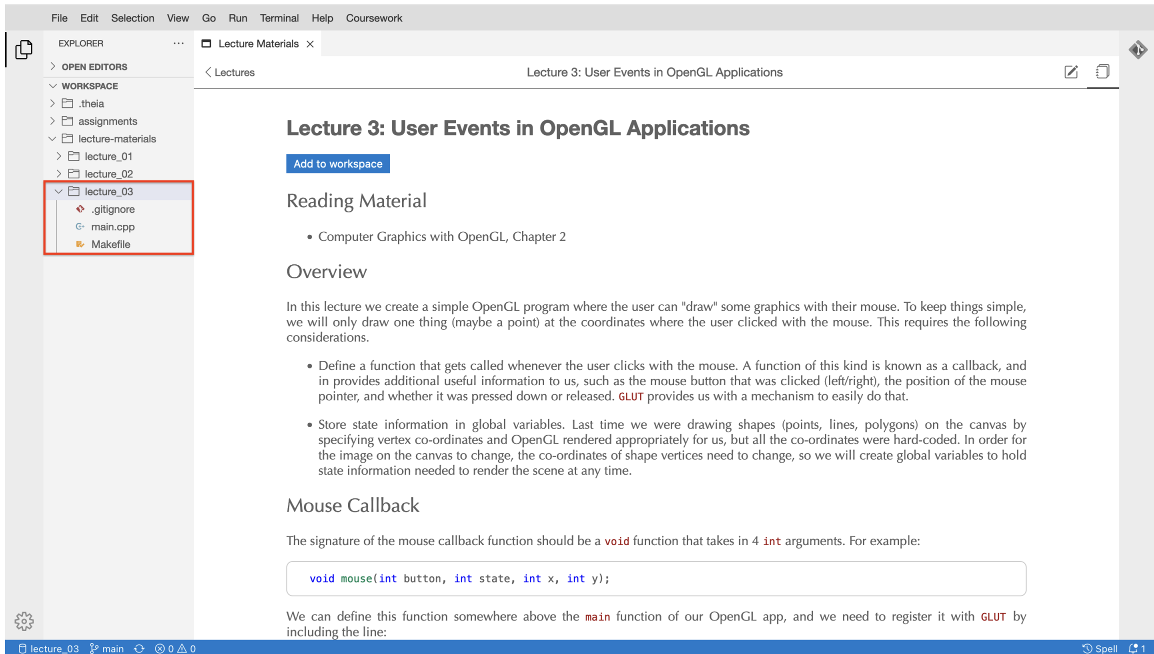


Figure 3.2: The Lecture Materials Widget interface

3.1.2 Assignments Widget

The system also supports programming exercises, with automated grading capabilities. The exercise creation process involves writing a problem statement, which can include sample input/output pairs, and suite of test cases, written in a unit-testing framework. The system automatically executes the test suite upon receiving a submission in order to determine its correctness.

Students are able to navigate the programming assignments and view the instructions for selected programming exercises. For convenience, the system has a "Bootstrap" feature which creates a project folder for the selected exercise and copies all support files needed.

Figure 3.3, shows a student who has navigated to "Lab 4" and has bootstrapped the "Gregorian Calendar Date" exercise. The instructions for the exercise are displayed in the main portion of the window, with the "Bootstrap Exercise" button on the top-right, and the "Submit" button on the bottom-left, as shown in Figure 3.4. In the file tree viewer, visible on the left sidebar of the interface, the system has created the appropriate `lab_04/exercise-2/` folder (highlighted in red) and copied all the necessary support files, which always include a `Makefile` for convenience. A student can begin generating their solution by simply clicking the `main.cpp` file from

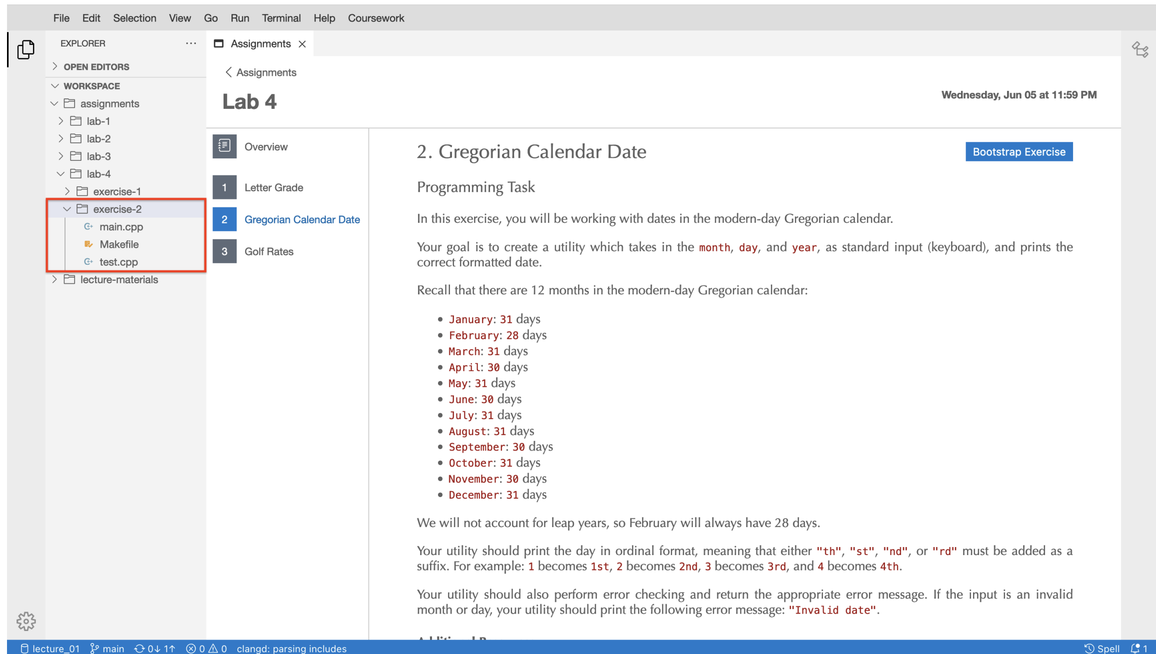


Figure 3.3: The Assignments Widget interface showing exercise instruction

the project folder and typing the code in the editor. The student can use the terminal emulator to compile and run their code.

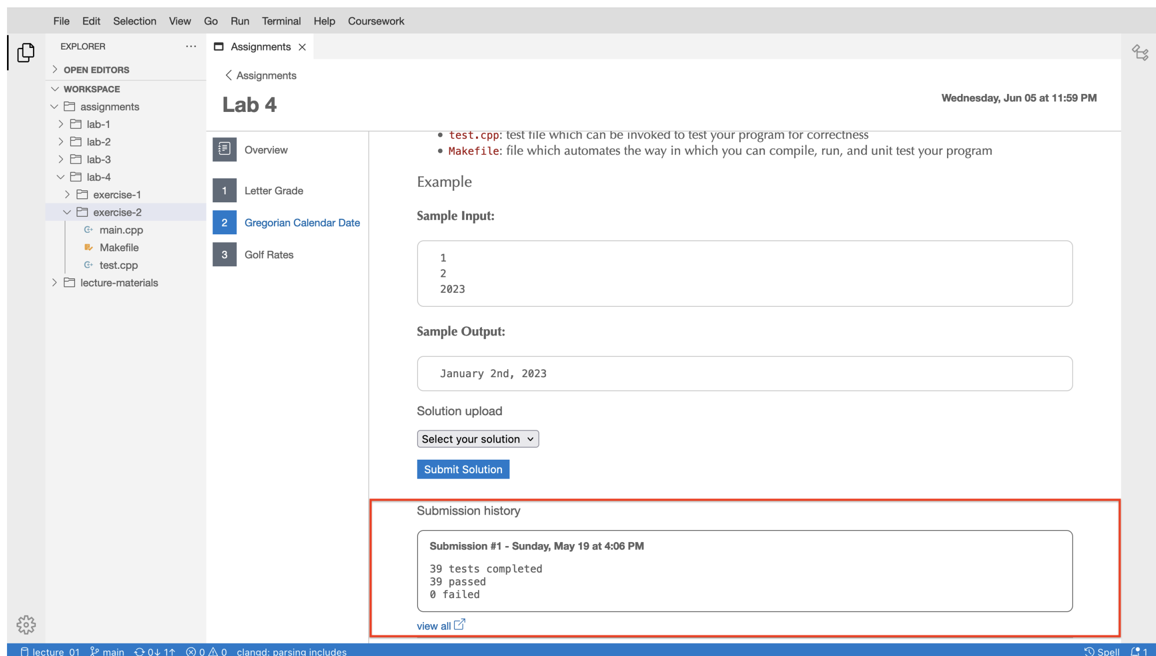


Figure 3.4: The Assignments Widget interface showing submission history for a student

The submission history, visible on the bottom portion of the interface (highlighted in red) in Figure 3.4, shows a student who has submitted their work for the "Gregorian Calendar Date" exercise of "Lab 4" and passed the suite of unit tests provided by the instructor.

3.1.3 Library Widget

The LMS also allows the instructor to create textbook reading materials, as well as end-of-chapter programming exercises, and distribute them to students. The reading material process can include code snippets, GIFs, images, videos, as well as links to other resources.

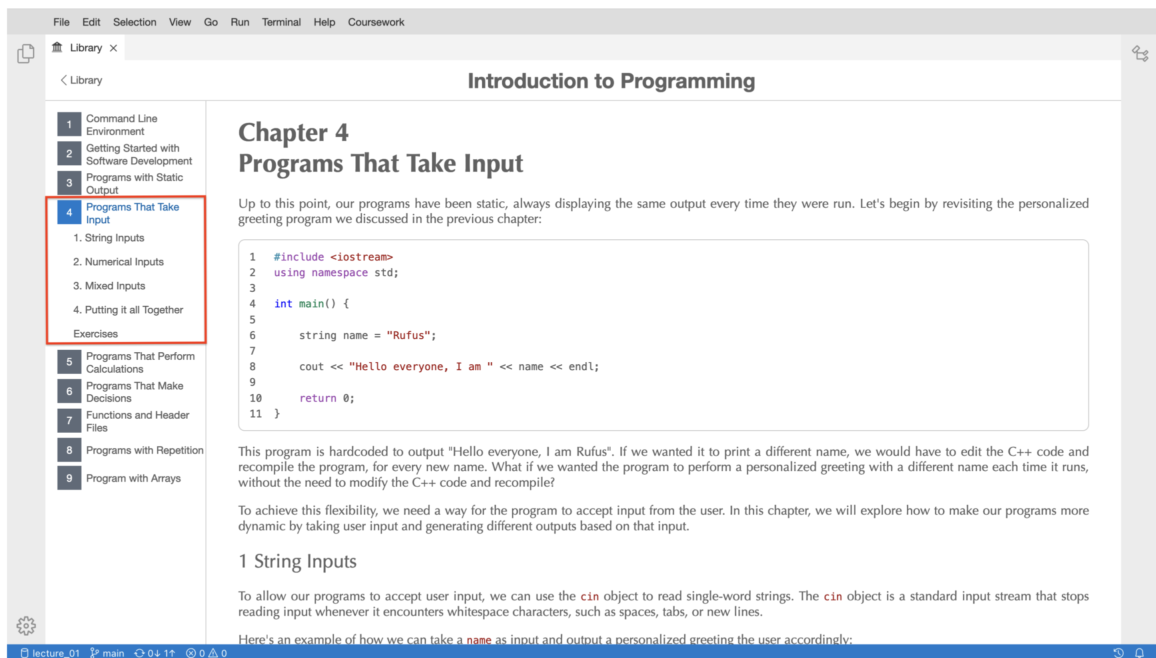


Figure 3.5: The Library Widget interface

Figure 3.5, is a screenshot of the interface of a student who has navigated to "Chapter 4: Programs that Take Input". The chapter content is displayed on in the main portion of the window, with the table of contents on the left sidebar of the interface (highlighted in red), showing that this chapter has four sections, followed by the end-of-chapter exercises.

3.1.4 Slides Widget

The LMS allows the instructor to import Google Slide presentations and distribute them to students. The process of importing slides only requires the url link of the presentation.

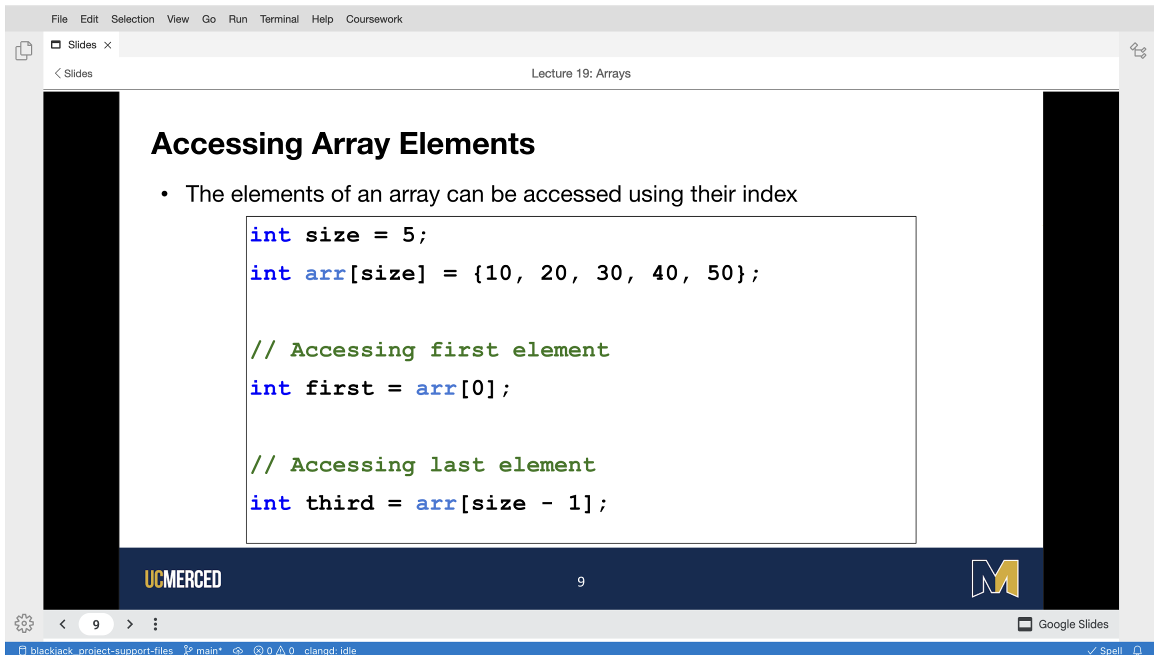


Figure 3.6: The Slides Widget interface

Figure 3.6 shows a student who has navigated to "Lecture 19: Arrays". The content of the slide is displayed on the main portion of the window, with the navigation buttons and slide information located on the bottom-left of the interface.

3.1.5 Fine-Grained System Logs

The online IDE produces fine-grained log data, by taking a snapshot of a student's file system at every interaction event, such as file change events, terminal command events, bootstrap events (recorded when a student begins working on their exercise), and submission events. These system logs are stored in BigQuery, a cloud-based database service that offers large storage capacity.

The system writes logs in response to the following 4 interaction events:

- **Bootstrap:** initialization of a new project folder for a selected programming exercise;

- **File Change:** any file on the file system is modified (entering text in an editor);
- **Terminal:** any command is invoked on the terminal emulator;
- **Submission:** submission of a programming exercise.

The data stored for each of the events above is described in Table 3.1.

Event Type	Field	Description
Bootstrap Event	`userId`	The ID of the user
	`timestamp`	The timestamp (in milliseconds) of the event
	`exercisePath`	The full path of the exercise
File Change Event	`userId`	The ID of the user
	`timestamp`	The timestamp (in milliseconds) of the event
	`filePath`	The full path of the file that changed
	`content`	The content of the file that changed
	`cursorPosition`	The cursor position (distance in characters)
Terminal Event	`userId`	The ID of the user
	`timestamp`	The timestamp (in milliseconds) of the event
	`workingDirectory`	The full path of current working directory
	`command`	terminal command
Submission Event	`userId`	The ID of the user
	`timestamp`	The timestamp (in milliseconds) of the event
	`exercisePath`	The full path of the exercise

Table 3.1: Table representing the system logs stored in BigQuery.

Processing log data from the system allows instructors to compute useful information, such as the exact amount of time a student spent editing a given file, or compute the sequence of transformations needed to recreate the process of entering text into an editor.

3.2 Plagiarism Detection Tool

We developed a plagiarism detection tool that enables the instructor to view all interaction events, between a student and the IDE, on a scrollable timeline. At each interaction event, the tool displays the state of the student's solution files. A diff editor can be used to highlight changes between consecutive interaction events. The tool can be used by instructors to examine the programming behavior of students, and look for potential signs of academic dishonesty. An obvious, and easy to detect pattern is pasting a complete solution from external sources, and either submitting it directly or performing basic text transformations, such as renaming variables. A more subtle form of plagiarism is the process of student manually typing in an external solution, instead of pasting it in. To find this type of plagiarism, an instructor could look for clues such as long typing sessions, with minimal to no intermediate testing, and the lack of any errors along the way. The tool presented in this section provides convenient interfaces that highlight the behaviors described above.

3.2.1 Data Processing Module

We developed a data processing module that is responsible for analyzing the raw system logs, given student and lab assignment, producing structured data capable of powering the plagiarism detection tool.

Listing 3.1: Data Processing Algorithm

```
1  pFiles = fetchProjectFiles(ASSIGNMENT_ID)
2  bootstraps = fetchBootstrapEvents(USER_ID, ASSIGNMENT_ID)
3  fileEvents = fetchFileEvents(USER_ID, ASSIGNMENT_ID)
4  terminalEvents = fetchTerminalEvents(USER_ID, ASSIGNMENT_ID)
5  submissions = fetchSubmissionEvents(USER_ID, ASSIGNMENT_ID)
6
7  events = MERGE(bootstraps, fileEvents, terminalEvents, submissions)
8  SORT(events)
9
10 allInteractions = []
11 files = {}
12
13 for i = 0 to events:
14   if events[i].type == BOOTSTRAP_EVENT:
15     for j in pFiles:
16       files[pFiles[j].filePath] = i
```

```
17          INSERT(events[i].userId, pFiles[j].filePath, pFiles[j].content, i)
18
19        allInteractions.push("Bootstrap", files, events[i].data)
20
21      else if events[i].type == FILE_EVENT:
22        files[events[i].filePath] = i
23        INSERT(events[i].userId, events[i].filePath, events[j].content, i)
24        allInteractions.push("File", files, events[i].data)
25
26      else if events[i].type == TERMINAL_EVENT:
27        allInteractions.push("Terminal", files, events[i].data)
28
29      else if events[i].type == SUBMISSION_EVENT:
30        allInteractions.push("Submission", files, events[i].data)
31
32  INSERT(allInteractions)
```

Listing 3.1 is the algorithm developed to process the raw system logs into structured data. Line 1 is responsible for fetching the support files for a given assignment. Lines 2-5 query the system logs (bootstrap events, file events, terminal events, and submission events) from BigQuery, and lines 7-8 merge and sort the data into a common collection consisting of all interaction events ordered by timestamp. Lines 10 defines the `allInteractions` data that we will be inserting into the `Interaction Events` table. Line 11 defines the `files` map which is used to store the `filePath-eventIndex` (key-value pairs) that can be used to query the file content at a given interaction event. This mapping is updated at every interaction event. Line 13 defines a loop that will iterate over all of the interaction events, with the main goal of tracking the file changes for all project files and inserting them into the `File Changesets` table. Lines 14-19, conditionally executed only for bootstrap events, inserts the content of the support files into the `File Changesets` table. Lines 21-24, conditionally executed only for file change events, inserts the content of the modified file into the `File Changesets` table. Lines 26-30, conditionally executed only for terminal and submission events, do not result in changes to the project files, so there is not need to update the `File Changesets` table. Instead, we simply push the interaction event data and `files` mapping to the `allInteractions` collection. Line 32 inserts the `allInteractions` collection into the `Interaction Events` table.

Suppose that a student used the online IDE to complete a “Hello World” pro-

userId	eventIndex	filePath	fileContent
0000000	0	exercise-1/main.cpp	...
0000000	0	exercise-1/Makefile	...
0000000	0	exercise-1/test.cpp	...
0000000	1	exercise-1/main.cpp	... c
0000000	2	exercise-1/main.cpp	... co
0000000	3	exercise-1/main.cpp	... cou
0000000	4	exercise-1/main.cpp	... cout
...	...	...	...
0000000	30	exercise-1/main.cpp	... cout << “Hello World” << endl;

Table 3.2: Example File Changesets Table.

gramming exercise. The exercise has three support files: `main.cpp`, `Makefile`, and `test.cpp`. The student worked exclusively inside the `main.cpp` file and completed their exercise in 35 interaction events: 1 bootstrap event, 30 file change events, and 3 terminal events, and 1 submission event. More specifically, after bootstrapping the exercise, the student typed `cout << "Hello World" << endl;` one keystroke at a time, followed by three terminal commands and then a submission.

Table 3.2 shows the rows of data inserted into the `File Changeset` table by the algorithm described above. The first three rows of the table contain the file contents of the three files at `eventIndex` 0 (bootstrap event). The following 30 rows denote the file changes in `main.cpp` that resulted from the student typing their solution one character at a time (file change events). The last four interactions, whereby the student submitted their work after using the terminal to compile, run, and unit test their code, did not result in any files being changed, therefore, the `File Changeset` table does not contain any more rows of data.

Listing 3.2 is the `allInteractions` object that was produced by the data processing algorithm, and stored in the `Interaction Events` table. There are 35 objects in the collection, each representing one of the 35 interactions events that the student engaged in while completing their programming exercise. Using this object, we can power the scrollable timeline of events. Instead of directly passing the file content at every file change event, we can use the event's index to fetch the content of any file from the database. This optimization enables the plagiarism detection tool to handle large amounts of data.

Figure 3.7 illustrates how the object from Listing 3.2 is used to power the in-

Listing 3.2: Example of processed Interaction Events data

```
[
    {
        type: Bootstrap,
        files: {
            exercise-1/main.cpp: 0,
            exercise-1/Makefile: 0,
            exercise-1/test.cpp: 0
        },
        data: {...}
    },
    {
        type: File,
        files: {
            exercise-1/main.cpp: 1,
            exercise-1/Makefile: 0,
            exercise-1/test.cpp: 0
        },
        data: { ... cursorPosition: 62, fileSize: 78, ... }
    },
    ...
    {
        type: Submission,
        files: {
            exercise-1/main.cpp: 30,
            exercise-1/Makefile: 0,
            exercise-1/test.cpp: 0
        },
        data: { ... }
    }
]
```

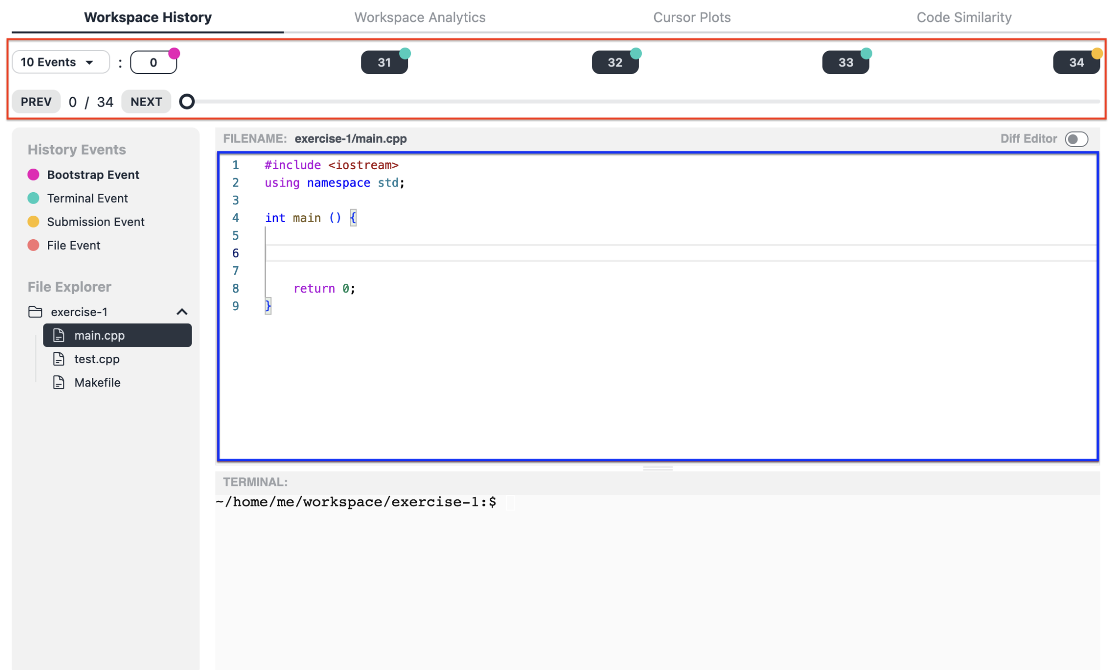


Figure 3.7: File History Viewer at first interaction (bootstrap event).

terface of the plagiarism detection tool. The draggable scrollbar, located at the top of the interface (highlighted in red), is populated using the processed interaction events collection. Five events of interest are shown: interaction 0 (bootstrap event), interactions 31 to 33 (terminal events), and interaction 34 (submission event). The instructor is currently viewing interaction 0, whereby the file content, displayed in the main portion of the window (highlighted in blue), is shown for the `main.cpp` file. The file content was fetched by querying the `File Changeset` table using the processed data from interaction 0, as seen in Listing 3.2: `userId = 0000000`, `eventIndex = 0`, and `filePath = exercise-1/main.cpp`. The `PREV` and `NEXT` buttons can be used to navigate to the previous and next interaction event, respectively. The terminal can be used to compile, run, and unit test the code.

3.2.2 File History Viewer

The File History Viewer puts all interaction events on a scrollable timeline and displays the state of the student solution file before and after the selected event, in a diff editor. A screenshot of this interface appears in Figure 3.8.

The instructor is able to replay the entire file creation process by dragging the timeline. In Figure 3.8, the particular student solution contained 2790 interaction

Figure 3.8: The File History Viewer interface. This example shows that interaction event 773 was the insertion of a semicolon at the end of line 21 in the file.

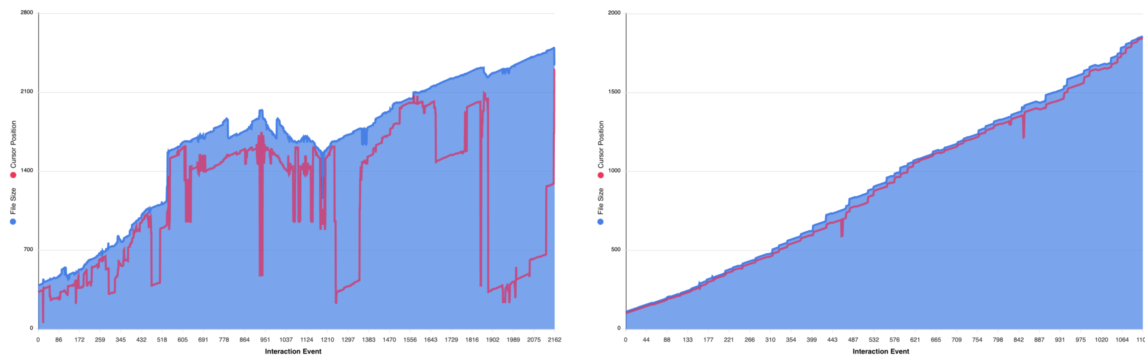
events. The interface also displays the points of interest, which are interaction events that resulted in a large amount of text being inserted, marked in green, or deleted, displayed in red.

Student solutions with few keystrokes, usually 2 or 3, where the entire file content was input in a single interaction event, are clear instances of copy-paste plagiarism. In some cases the solution being pasted in was "borrowed" from another student in the class, in which case, traditional plagiarism detection tools would have identified it, but in many cases, the solution pasted in was sufficiently different from all others, indicating that it was external.

3.2.3 Cursor Position Plots

The Cursor Position Plots display the cursor position, relative to the file size, at every file change event (keystroke). These plots are helpful in identifying students that copy complete solutions from an external source, typing it in one line at a time, from top to bottom, resulting in a cursor that is always at or near the end of the file.

The blue line in the plot (area under the line shaded for clarity) represents the file size at each interaction event, while the red line represents the position of the cursor at that event. It is clear that the student who generated the plot in Figure 3.9b typed



(a) The large RSS between the file size and the cursor position curves is indicative of honest programming behavior.

(b) The small RSS between the file size and the cursor position curves indicates manual entry of an external solution.

Figure 3.9: Cursor position plots for a student coding session.

their solution without ever removing text from the editor, always typing at or near the end. This is indicated by the small distance between the two lines. Formally, we compute Residual Sum of Squares (RSS) between the two curves by:

$$\text{RSS} = \sum_{i=1}^{n} (f_i - c_i)^2$$

Where f_i is the file size at event i, and c_i is the cursor position taken as the number of characters from the beginning of the file, to the cursor position (not line number, and column number). A small RSS value can be indicative of plagiarism. For comparison purposes, Figure 3.9a is a plot generated by a student whose text editing patterns are indicative of honest programming behavior.

The cursor plot in Figure 3.9a is representative of honest programming behavior mainly because the student is editing the file in different locations at various points in time, which is consistent with typical programming, for example defining a new variable several lines above the current cursor location, and then going back down to use the variable. The other characteristic to note is that the file size is not a strictly increasing sequence, indicating that the programmer deleted lines of code, which is also a common programming behavior.

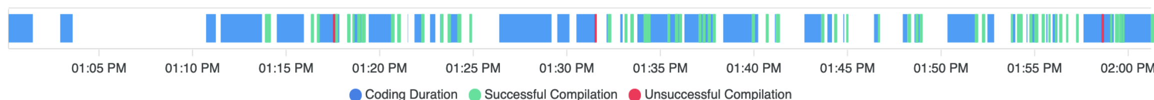
3.2.4 Event Timing Graphs

The Event Timing Graphs serve as a visual representation of the time a student spends programming, capturing the bursts of active coding (typing), idle time (breaks

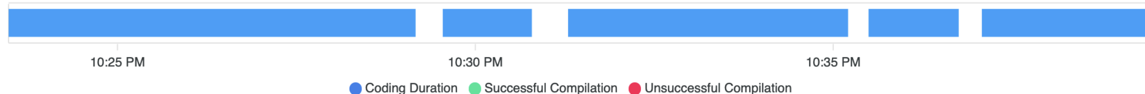
from typing), and the number of times students compile their program. The active coding and idle time are calculated by taking the difference between two consecutive keystroke events. If the difference in time is less than 10 seconds, it is considered active typing. However, if the difference in time is greater than 10 seconds, but less than 10 minutes, it is considered a break from typing.

The Keystrokes to Recover is computed by taking the average number of file change events it takes a student to go from an unsuccessful compilation to a successful compilation. The Time to Recover is computed similarly, except that it reports the average time it took a student to go from an unsuccessful to a successful compilation.

Figure 3.10 shows the Event Timing Graphs, whereby the blue time intervals indicate active typing, while green vertical lines are successful compilations, and red vertical lines indicate unsuccessful compilations. These graphs are helpful in identifying suspicious behavior that may be consistent with plagiarism.



(a) Timing events for a student who likely engaged in honest programming.



(b) Timing events for a student who likely plagiarized by typing in a complete solution obtained from an external source.

Figure 3.10: Timing events for a student coding session.

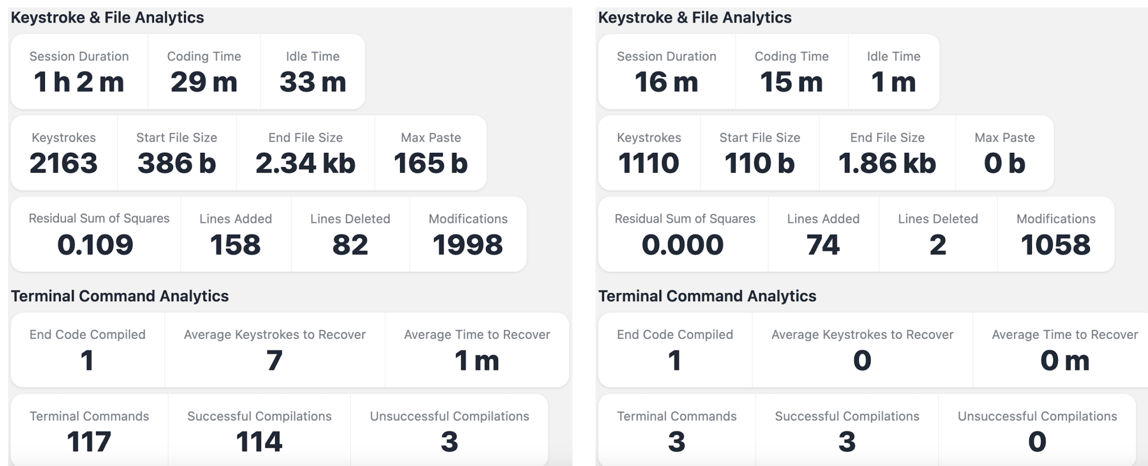
Figure 3.10b shows a student that likely engaged in copying a complete external solution because they completed their programming exercise in less than 15 minutes, without taking any breaks to read, compile, or test their code.

For comparison purposes, Figure 3.10a shows another student that likely engaged in honest programming. This student exhibited a large number of short typing intervals with a substantial amount of testing at intermediate stages.

3.2.5 File Analytics

The File Analytics, presented in Figure 3.11, serves as a bird's eye view of a student's programming behavior. The interface contains useful statistics, like total session duration, with typing time and total break time (idle time) in minutes. It also displays the number of keystroke events, starting file size in bytes (number of characters), the

file size at the end of the session, and the number of characters in the largest paste operation, if any. The Residual Sum of Squares is the value computed from the cursor position plot, described in the previous section. The interface also shows the number of successful and unsuccessful compilation commands.



(a) File analytics events for a student who likely engaged in honest programming.

(b) File analytics for a student who likely committed external plagiarism.

Figure 3.11: File analytic events for a student coding session.

Figure 3.11a shows the file analytics features for a student who likely engaged in honest programming. This student spent over 1 hour generating their solution, with the majority of the time spent not typing, which typically means that the student is reading their code and reasoning about their future work. It is also clear that this student did not have any major pastes, and they tested and compiled their code extensively. With 117 compilation commands, 3 of which were unsuccessful, but the student was able to quickly address the issues and get back to a working state.

Alternatively, Figure 3.11b shows the file analytics features for a student who likely copied a complete external solution because for 15 minutes out of the 16-minute session, the student was continuously typing, without stopping to possibly think about their code, without making a single mistake, and testing their code only 3 times, all of which succeeded.

3.3 Grading Platform

We developed a grading platform on top of the existing LMS and plagiarism detection systems in an effort to streamline the grading process for instructors. It is a web-

based solution that allows instructors to view and run students' code in the same environment in which it was developed, eliminating potential compatibility issues. Upon selecting a student's programming exercise, the submission files, as well as interaction events, are fetched from the database. This allows the human grader to verify the correctness of the solution, examine the thought process and problem solving strategies employed by the student, and flag any suspicious behaviors for further investigation.

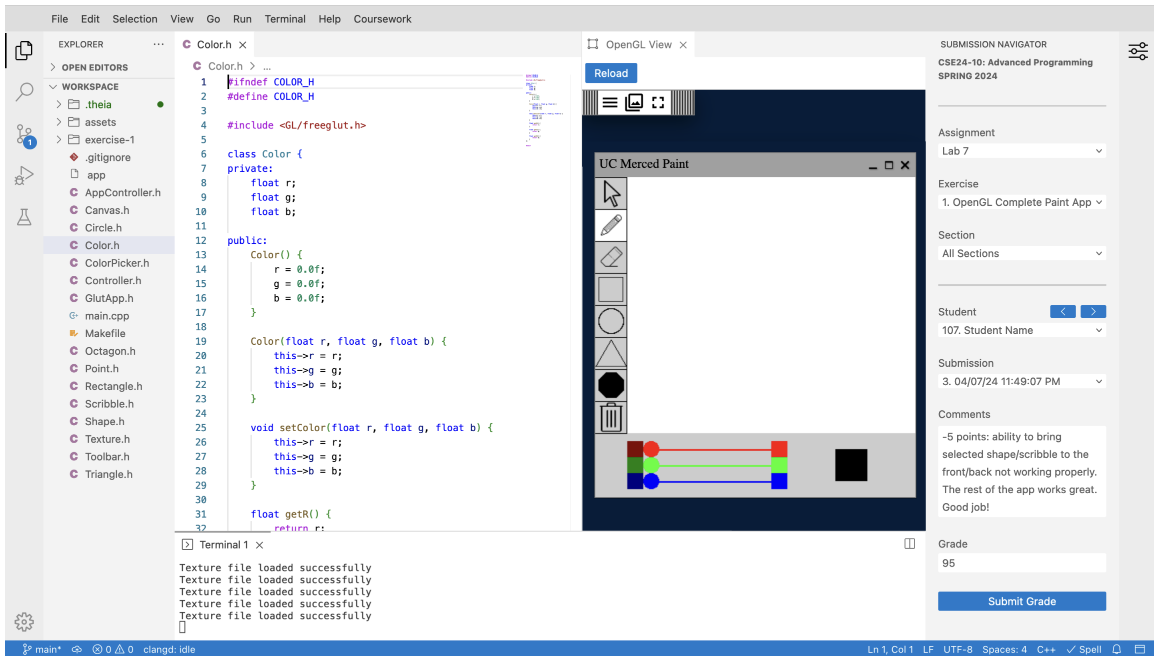


Figure 3.12: The Grading Platform interface showing the submission for student 107.

Figure 3.12 shows the Grading Platform. The right sidebar, labeled as the Submission Navigator, features several dropdown menus that enable the instructor to select a student's submission for a specific assignment exercise. The Section dropdown displays the various course sections, allowing Teaching Assistants to easily select their assigned section and filter the students accordingly. The two buttons adjacent to the Student dropdown facilitate navigation to the previous and next student, respectively. The Comments and Grade fields provide the teaching team with the means to offer feedback and assign grades.

Figure 3.12 shows the Grading Platform being used to assess submission 3, of exercise 1 (OpenGL Complete Paint Application) from Lab 7, for student 107 of CSE 24-10 (Advanced Programming). The submission files are accessible via the file tree viewer, located on the left sidebar. The text editor, displayed on the left portion of the

interface, shows the contents of the `Color.h` file, while the OpenGL View window on the right showcases the graphical window which is rendering the program submitted by the student, which constitutes a paint application.

3.4 Interactive Code Rewind Tool

We developed the Interactive Code Rewind tool that enables navigation and annotation of Git repositories. In the context of educational settings, Git repositories are useful for reviewing the thought process of the programmer at intermediate stages of development. Git stores snapshots of file contents at different milestones (commit points), and requires a message at each one, describing the changes introduced to the code since the previous commit point. Educators can use commit messages to describe their thought process and problem solving strategies while developing a coding demonstration, but commit messages are brief, contain only plain text, and are immutable after creation. To augment instructors' ability to describe the changes introduced at each commit, we built a database layer to store additional notes in rich text format, associated with a particular commit, which instructors have the ability to edit after the fact.

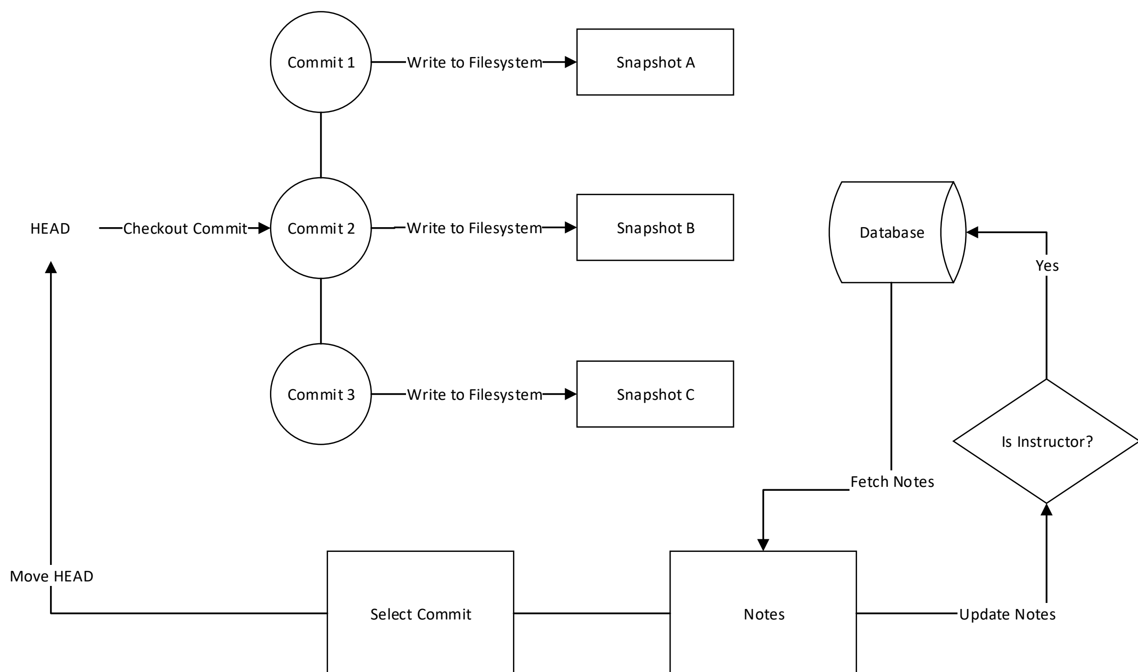


Figure 3.13: A flowchart of the Interactive Code Rewind tool.

Figure 3.13 illustrates the workings of Interactive Code Rewind system. The HEAD pointer, seen in the diagram above, represents the commit that is currently checked out, which is to say that the contents of the repository files are synchronized to the corresponding snapshot. The notes stored in the database are associated to a specific commit, which is uniquely identified by the Git system using a 40-character hash. Students typically have read-only access to the database, allowing them to view the notes written by the instructor, who has full database access.

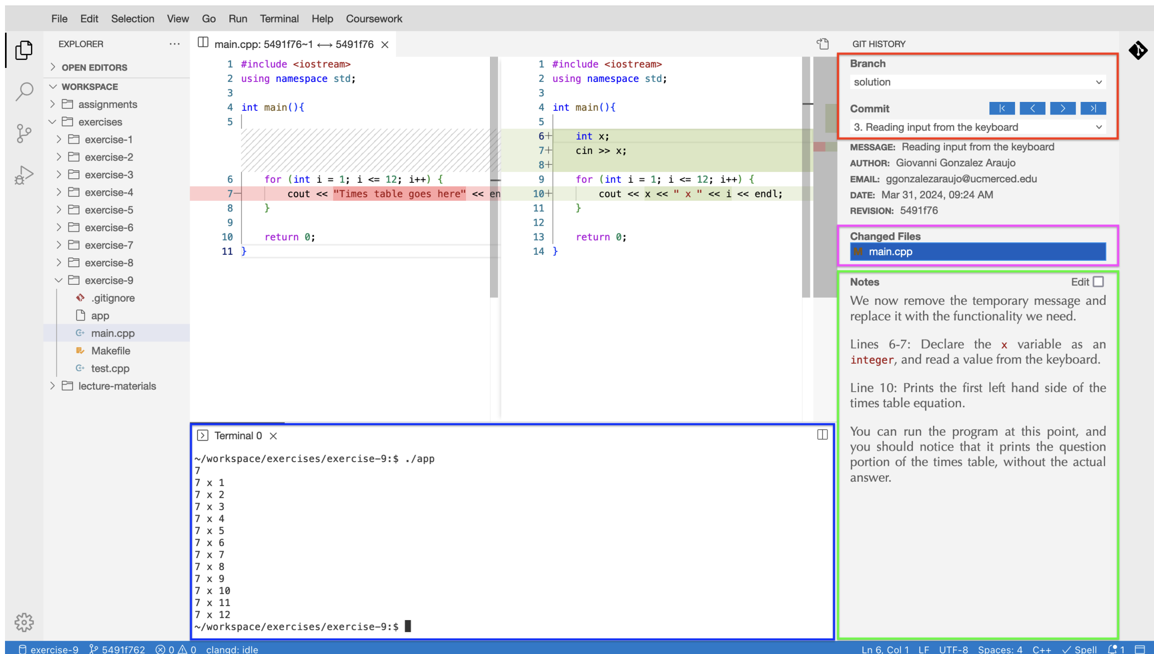


Figure 3.14: The Interactive Code Rewind interface for `lecture_9` at commit 3, showing the notes and a diff editor with content that changed.

The Interactive Code Rewind tool, presented by Figure 3.14, shows a student who has selected "Commit 3: Reading input from the keyboard" of the *solution* branch from the `exercise-9` repository (highlighted in red). The four buttons adjacent to the Commit heading allow users to navigate to the first commit, previous commit, next commit, and last commit, respectively. The Changed Files section (highlighted in magenta) lists the files that were modified during that commit, while the diff editor shows the lines of code that were added (shown in green), as well as the lines which were deleted (shown in red). The Notes section (highlighted in green) displays the rich text annotations that were written by the instructor. The terminal (highlighted in blue), shows the result of the student compiling and running the code, as it exists at the current commit.

Chapter 4

Case Study: An Introductory Programming Course

4.1 Introduction

The case study reported here was performed at the University of California, Merced (UC Merced), in an introductory programming course, during the Spring 2022 semester. UC Merced is a suitable setting for work of this kind, as it hosts a diverse student population, with a high percentage of first-generation students, many of whom are members of under-represented minority groups, coming from underserved areas, especially in terms of access to computing. Characteristics of the student population are explained in Section 4.2.

The study was performed in an introductory programming course, called CSE 24, described in Section 4.3, which is considered similar to other introductory programming courses in the state of California, as it articulates to every major educational institution in the state.

We observed that the average lab grades for students in the course were significantly higher than their midterm grades, with many students who appeared proficient in programming during their lab assignments failing to demonstrate any knowledge of programming in their midterm exams. The details of the study appear in Section 4.4. We also looked at the amount of time students are spending on programming, as an indication of the amount of effort and engagement, with details presented in Section 4.5.

The disconnect in performance between programming assignments and exams, as

well as the minimal amount of time students in the course spent programming, led us to suspect high levels of plagiarism as a possible explanation for these results. Section 4.6 presents the results of an investigation into plagiarism levels in the course.

4.2 Student Population

In Fall 2023, there were a total of 8,373 undergraduate students, of whom 65% are first-generation [2]. The distribution according to race and ethnicity can be seen in Figure 4.1. Over half the student population at UC Merced is Hispanic, therefore the university is classified as a Hispanic Serving Institution (HSI).

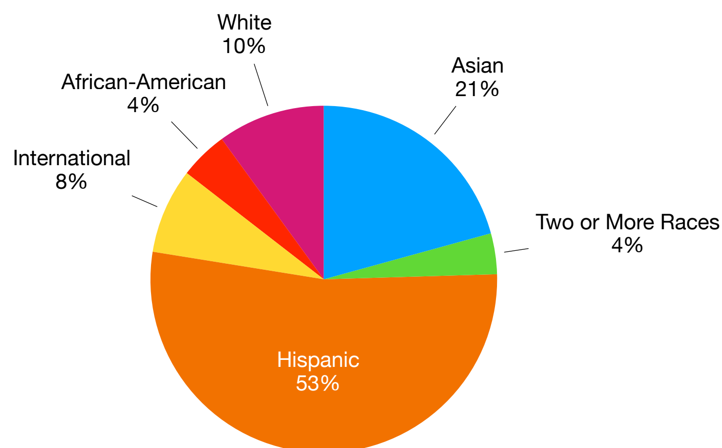


Figure 4.1: Student distribution according to race

A large percentage of the student population of UC Merced can be characterized as low-income. The university has the highest percentage of undergraduate students who receive need-based financial aid, among all public universities in the United States. In Fall 2023 61% of students were Pell grant eligible, while another 13.5% received other forms of federal assistance. In total 90% of UC Merced undergraduate students receive financial aid of some form.

Low income has been shown to negatively affect college readiness levels of students. The ACT defines college readiness benchmarks in four areas, including English, Mathematics, Reading, and Science. The chart in Figure 4.2 shows students' attainment of the four readiness benchmarks according to family income levels (dark blue), as well as their self-reported family income (light blue) [1].

For UC Merced students, the vast majority are eligible for financial aid, indicating that their annual family income is below $60K. This suggests that only 10-20% of them may meet all four college readiness benchmarks, based on the trends observed in the national study. However, it is important to note that these figures are inferred from the broader data and may not directly reflect the specific outcomes for UC Merced students. College readiness levels tend to increase sharply for students whose family income is on the higher end.

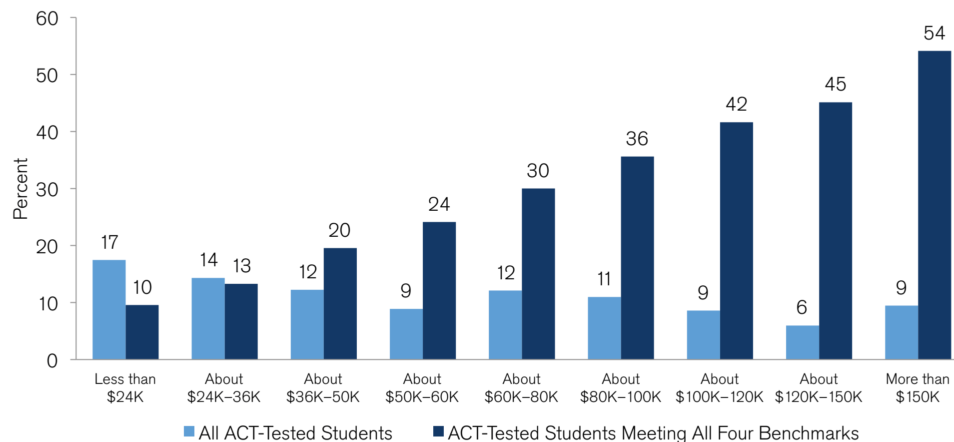


Figure 4.2: National ACT college readiness benchmark attainment by annual family income

Exposure to computing early in life has clear benefits for students pursuing a degree in Computer Science. Students from low-income backgrounds are at a disadvantage due to the lack of adequate access to such resources in their families and communities. Research has also shown that first generation students face challenges in their undergraduate studies, making them more likely to switch away from Computer Science, or to leave the university entirely [51]. The lack of role models has also been identified as a barrier to success for students from underrepresented minorities.

4.3 Course Description

CSE 24 is the second course in the introductory programming sequence, and is the equivalent of a CS2 course. The course introduces students to object-oriented programming, pointers and dynamic memory management, while providing additional practice for students to develop their programming and problem solving skills. By the end of this course students are expected to be competent programmers, in at least the

C++ language, setting them up to be successful in upper-division Computer Science courses.

The course earns 4 academic credits, equating to 12 hours of effort per week over a 15-week semester. There are two weekly lectures of 1 hour 15 minutes each, and a laboratory session of 2 hours 50 minutes. Lectures are conducted in large auditoriums, where the instructor uses a combination of lecture slides, live-coding demonstrations and notes written on a whiteboard, to deliver the material to the students. Due to the large number of students in the classroom, instructors typically spend class time lecturing to the students, with limited opportunities for individual attention. Students are expected to practice the material they learn in lectures by completing their weekly programming assignments, which they can work on at home, and/or during their dedicated laboratory sessions.

A weekly programming assignment is a collection of programming exercises, where the number of tasks is typically 5-6, but could vary depending on the scope and difficulty of each task. Students are presented with a prompt, explaining the programming task, as well as sample input and output pairs that they can use to test their solutions. Students are expected to produce fully working programs that can be compiled and executed. Solutions to programming exercises are graded automatically by a system, described in Section 3.1, that compiles and runs the code submitted by the students, and compares the outputs produced to the expected outputs provided by the instructor. Students receive immediate feedback on the correctness of their solution, and are allowed an unlimited number of resubmission attempts until the specified deadline.

Every student is assigned to a weekly laboratory session where they can work on their programming assignments. These sessions are led by Teaching Assistants (TAs) and are restricted to 30 students per session. This is to allow the TA to spend time with individual students who may require additional help with their programming exercises.

The course has a formal assessment component made up of a midterm, and a final examination. Due to the practical nature of the course, formal examinations typically ask students to demonstrate their understanding or mastery of programming concepts they had learned in lectures and practiced during lab sessions. Exam questions are structured similarly to programming interviews, where the candidate is asked to write working code on paper (without the aid of a compiler), trace or explain given code snippets, or answer conceptual questions related to programming.

A student's course grade is calculated as a combination of their scores from formal

assessments, such as midterm and final examinations, class participation grades, and their lab assignments grade. Although it varies from semester to semester, the laboratory grade makes up to 30% of a student's course grade, so it carries a significant amount of weight.

4.4 Average Formal Exam and Laboratory Scores

In the Spring 2022 semester there were 115 students in the class, the average programming assignment grade was 96%, while the average midterm grade was 68%. Figure 4.3 shows these averages, with the error bars representing Standard Deviation.

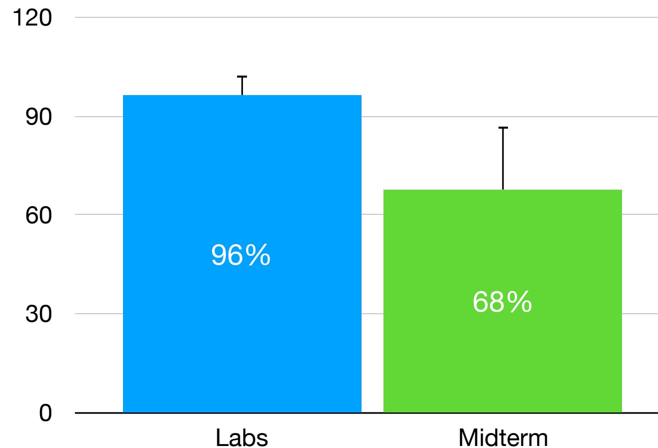


Figure 4.3: Average laboratory grades compared to midterm grades in Spring 2022.

In addition to the large difference in terms of mean, the correlation between laboratory grades and midterm grades is weak, with a Pearson Correlation Coefficient of $\rho = 0.28$. The scatter plot in Figure 4.4, with laboratory scores on x-axis and midterm scores on the y-axis, also illustrate the weak correlation. 70% or above is considered a passing grade, and the pass rate for laboratory assignments was 99.1%, while the pass rate for the midterm examination was 49.6%.

The weak correlations observed above are an indication that students are not learning from their practical assignments. A possible explanation for this was that students commit plagiarism. Practical programming exercises are assigned on a weekly basis, and students have ample opportunity to seek outside assistance, from platforms like Chegg and Stack Overflow, from fellow students in the course, or from generative

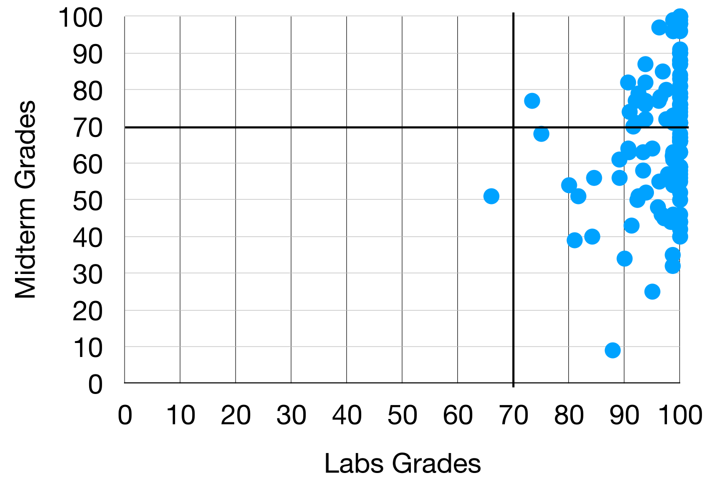


Figure 4.4: Scatter plots of laboratory and midterm grades in Spring 2022.

Artificial Intelligence tools. Furthermore, the automated nature of grading for these assignments contributes to plagiarism.

4.5 Time Spent on Programming Assignments

The fine-grained log data available from our IDE, described in Chapter 3, can be used to obtain an accurate measure of the amount of time students spend on programming assignments. In this course offering, we there were seven laboratory sessions, each containing between one and ten programming exercises. The first lab included ten exercises designed to review topics from previous courses, such as printing a personalized greeting, unit converters, and creating a dinner bill calculator that takes in as inputs the bill amount, tip percentage, and number of people, and computes the price each person must pay. The second lab had five exercises focusing on problem-solving, including finding the longest consecutive increasing sequence of numbers in an array, printing a parameterized ASCII triangle, and calculating the number of years, weeks, and days given a certain number of days (37 days = 0 years, 5 weeks, and 2 days). The third lab consisted of five exercises centered on strings, such as finding the longest hyphenated word in a text, reversing a string without using STL libraries, and performing shift cipher encryption. The fourth lab comprised five exercises on vectors, including removing or updating elements from a vector, interleaving two vectors, and visualizing a two-dimensional vector on the terminal. The fifth lab featured two exercises on pointers, such as calculating the distance in bytes between the memory

locations of two variables. The sixth lab had one exercise on pointer arithmetic and manipulation, where students were tasked with creating a cryptographic library. The seventh lab included one exercise on dynamic memory management, tasking students with creating a resizable container capable of storing elements of different data types (bool, int, float, char), similar to a Python list. A complete listing of assignment questions can be found in Appendix A.

Lab	Time on Task
1	151
2	77
3	76
4	67
5	56
6	72
7	62

Table 4.1: Time in minutes spent by students on programming assignments in CSE 24 for Spring 2022

Table 4.1 shows the average time students spent working on their programming assignments during the Spring 2022 offering of CSE 24. On average, students spent about 80 minutes a week working on their programming exercises.

4.6 Plagiarism on Programming Assignments

The observations presented in this chapter, namely poor correlation between laboratory and exam grades, paired with the short periods of time students spend on their programming assignments, roused our suspicion that plagiarism must be a significant factor in the course. We set out to determine an accurate plagiarism rate, so it can be used to draw conclusions about student learning in the course.

For this task, we selected two exercises from the course and used the tool described in Chapter 3 to analyze the programming process of each student who submitted a solution. This is a highly labor-intensive task, as it requires an instructor to replay process of text entry for each submission and look for suspicious patterns. Students who copy/pasted complete solutions were detected efficiently, as the instructor had to scroll through a relatively small number of interaction events. Students who manually typed in plagiarized solutions had on the order of 1500 keystrokes, so it was a time-consuming process to examine them all, whereas students who worked honestly had

several thousand keystrokes to examine. It is important for instructors to complete this review carefully as a student may appear to be working honestly on a solution, only to paste an external one late in the process.

There were a total of 210 unique students who made submissions, and we found suspicions of plagiarism in 101, or 48% of them. For comparison, the popular plagiarism detection tool MOSS was only able to flag 49 of them. Our intuition is that MOSS was only able to detect half of the suspected plagiarism cases because many students are making use of external resources, where they can obtain unique solutions, or are able to perform a sufficient number of code transformations on a borrowed solution so as to evade detection based on similarity.

4.7 Conclusion

The overall conclusion from the case study described here is that students are not acquiring as much programming skills as they should from the course. This is evident from the uncorrelated grades in laboratory assignments and formal exams, where students are apparently proficient in programming but are unable to demonstrate this during formal testing, which is designed to be similar to interviews for programming jobs. Since UC Merced students are behind their nationwide peers in terms of college readiness, they would benefit from spending more time and putting in more effort on their programming assignments. The high levels of suspected plagiarism also contribute to the perceived low learning rates. It is hard to imagine how so many students complete their programming assignments correctly in so little time, in the presence of so much suspicious behavior.

Chapter 5

Apprenticeship and Product Based Learning

As is common in many introductory programming courses, CSE 24 at UC Merced makes use of laboratory assignments to provide opportunities for students to internalize concepts covered in lectures through practical exercises. Despite the presence of a strict Academic Honesty Policy at the department, plagiarism rates have remained consistently high, as reported in Chapter 4. Plagiarism, and to some extent disengagement/non-attendance, are believed to be contributing factors for the low exam scores in CSE 24, reported in Chapter 4. We believe the low exam scores are indicative of low learning rates, since the exams are structured similarly to programming interviews, asking students to demonstrate their understanding of programming concepts. We hypothesize that if students worked more on their programming assignments, without committing plagiarism, then their exam scores will increase, which will be an indication of better learning rates.

An easy way to increase the amount of time students spend programming is to increase the size of their assignments, either by including more exercises or increasing the scope of each task. An equally simple method of reducing plagiarism is to increase detection efforts, possibly with the help of specialized software tools, and institute harsher punishments for offenders. Both of these strategies have been proven ineffective.

We believe that if programming assignments were more interesting and/or relatable to students, they would be naturally more motivated. This could lead to increased effort, as well as a reduction in plagiarism. Project Based Learning, described in Chapter 2, has seen much success in Computer Science, especially in cap-

stone courses, where students work in teams on real-world projects. Adoption in introductory classes is more challenging, with researchers and educators citing difficulties in evaluating student performance, as well as the necessity to adopt new teaching modalities and modify existing curriculum.

We introduced a flavor of Project Based Learning, less reliant on teamwork, making it more suitable for individual student assessment, as is common in introductory programming courses. Our approach, called *Product* Based Learning, is presented in Section 5.1. The additional course material needed to support Product Based Learning necessitated the adoption of new course delivery techniques, inspired by Apprenticeship Learning. Section 5.2 describes our applications of Apprenticeship Learning, including the lecture delivery methods and software tools needed to support them.

5.1 Product Based Learning

Reflecting on our traditional assignments, we identified significant issues related to their rigid structure, which not only hindered student creativity, but also made it easy for students to find ideal solutions online. Furthermore, the assignments had a definitive ending point; once students passed all the unit tests, it meant that they had completed the assignment, leaving no further incentive for continued programming.

Given students' inexperience with command line environments, we observed an apparent lack of interest and motivation on the part of students when working on traditional programming assignments. When tasked with developing command line programs that produce textual output, students struggled to perceive this as creating real software. Additionally, many students expressed interest in developing websites or mobile applications, likely influenced by their frequent interaction with these technologies in their daily lives.

We decided to adopt a Project Based Learning approach for programming assignments, where students would be asked to work on projects with the following requirements:

- completed individually;
- graded by a human;
- employ graphical user interfaces;
- have open-ended specifications;

- resembles a real-world product.

We place strong emphasis on the last point, which is that whatever programs students write, those programs should resemble real software products, hence we named our instructional methodology: Product Based Learning.

We believe that creating graphical user interfaces will motivate students, as it aligns more closely with their perception of what constitutes software, and the open-ended nature of the project specification will increase engagement as students will be free to use their creativity when implementing features in their product. Since students know that their projects will be graded by a human grader, we expect that some of them may put in additional effort in order to impress the grader, a phenomenon known as the Hawhtorne Effect, which is the tendency of people to change their behavior due to their knowledge that they are being observed. It is also expected to have a significant effect on reducing plagiarism.

It is a common practice to show students a "hello world" program sometime near the beginning of an introductory course. Depending on the programming language of choice, this program could be 1 line of code, as in Python or Ruby, or about 5 lines of code, as in Java or C++. The result of running the program is always a variation of the text `Hello World` appearing on the command line where the program was invoked.

In contrast, a Product Based version of the "hello world" program would involve a graphical window with the "Hello World" message on it. Figure 5.1 shows a screenshot of a sample GUI based "hello world" program, while Figure 5.2 lists the source code a student would need to produce.

While it can be argued that the number of lines of code for the GUI based "hello world" program is not significantly larger than the number of lines needed to produce a command line version of the program in C++, the GUI based version employs more advanced programming concepts such as variable declaration, object instantiation, and inheritance, even if one is to ignore the use of pointers and dynamic memory management in the snippet above. Furthermore, in a command line C++ program, reading input from the user would amount to writing a correct `cin` statement, whereas in the GUI version there would need to be a button object instantiated, and a callback defined for handling click events.

The added complexity would be worth the effort, if it resulted in a significant reduction in plagiarism, increased effort and engagement levels on the part of students, and higher learning rates. In order to increase the changes of success in Product Based

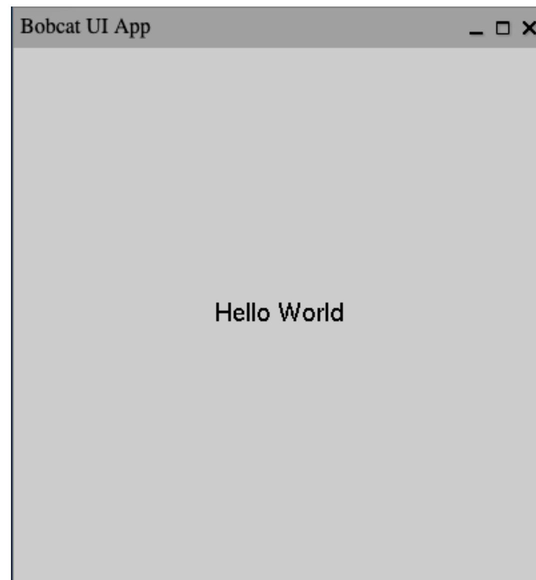


Figure 5.1: Screenshot of GUI based "Hello World" program

```
#ifndef WINDOW_H
#define WINDOW_H

#include "AppWindow.h"
#include "Label.h"

class Window : public AppWindow {
    Label* message;

public:
    Window(){
        message = new Label("Hello World");
        addControl(message);
    }

    ~Window(){
        delete message;
    }
};


#endif
```

Figure 5.2: Source code of GUI based "Hello World" program written in C++

Learning, we also adopt a new teaching philosophy based on the Apprenticeship Learning Model.

5.2 Apprenticeship Learning

The Apprenticeship Learning Model, described in Chapter 2, is an educational model in which an apprentice learns a skill or trade through observation and practical experience under guidance of an expert. Apprenticeship has proven to be highly effective in the development of skills which involve tangible processes that can be easily observed, such as learning to play an instrument or building furniture. Learning to program and building furniture share similarities in that they both require understanding the components involved, whether it's the syntax and logic of a programming language or the various materials and assembly techniques for furniture. Both activities also involve an incremental work process, and they demand problem solving skills to overcome challenges and errors encountered along the way, oftentimes requiring the use of specific tools and techniques to achieve the desired outcome. Furthermore, they generally follow an iterative process, where the initial version of the project is refined and improved over time.

5.2.1 Lecture Delivery

A live coding demonstration involves the design and implementation of code in front of an audience. Many studies have shown positive outcomes of live coding demonstrations in Computer Science, such as improved programming and debugging understanding [21, 108, 116, 73]. We adopted live coding demonstrations as the primary teaching method for delivering our lectures. Using the classroom projector, we share our screen with students as we explain and develop code in real time. We typically begin with an empty text editor and develop all portions of the code during lecture. This approach gives students the opportunity to observe our cognitive processes, problem solving strategies, and debugging techniques firsthand. The live coding demonstrations offer a comprehensive insight into the entire programming process, extending beyond mere code composition within a text editor. Through these demonstrations, we showcase other important skills which are essential in programming, such as navigating the file system, using the command line, as well as compiling, running, and testing programs.

Although live coding demonstrations effectively fulfill the *Modeling* phase of apprenticeship learning, by providing instructors with a platform to demonstrate and articulate the programming process to students, they present scalability challenges in large courses. In traditional apprenticeship settings, the learner-to-expert ratio is typically small, often comprising only one or a few apprentices per expert. This enables the expert to allocate one-on-one time to each apprentice, allowing for individualized attention. In typical introductory programming classes, the number of students is usually in the hundreds, and there is a high degree of variance in preparation and programming skill levels among students. This makes it difficult, if not impossible, for instructors to execute the `Scaffolding` phase during lecture.

5.2.2 Interactive Lecture Code Rewind

In apprenticeship learning, scaffolding is when students begin to mimic the actions of the instructor, after having witnessed the demonstration. To facilitate this, we started recording the live coding demonstrations, including both the computer screen and external audio, allowing students to view the live coding demonstration and listen to the instructor's explanations. The source code associated with the programming demonstration was made available along with each recording.

Despite these efforts, the lecture recordings proved ineffective for several reasons. Students expressed that re-watching the recordings was time-consuming and mentally exhausting. To fully replicate the live coding experience, students would need to simultaneously play the lecture recording while writing the code themselves in a text editor to run and experiment with the code, similar to the instructor's actions during the live lecture.

Recognizing the limitations of the existing method, it became apparent that we needed to provide students with a more efficient means to revisit the live programming demonstration, as close as possible to how they experienced it during the live lecture. Instead of recording lectures as video and audio, we started each programming demo as an empty Git repository. Git is a popular version control system, allowing developers to track changes to their codebase, among other features. During live coding demonstrations, the instructor would develop their code as usual, with the addition of pausing at important developmental milestones, and performing a commit operation, which is the process of storing a snapshot of the current state of the codebase in the version control database, and allowing users to revert to that state in the future.

It is possible to share entire Git repositories through online platforms, such as GitHub or GitLab, both of which offer this service free of charge. After every commit point in a lecture demonstration, we share the code with the students in the class, through an online Git platform. This gives our students access to the code that we have written in real time, without the need for them to frantically keep up with the instructors typing, allowing them to focus on the instructors' explanations during lectures.

This mechanism allows the instructor to provide code to students with a high degree of control in terms of timing, as well as partitioning. Students can use the code they receive for in-class exercises, and other forms of active learning. Due to its dynamic nature, the instructor can easily share materials with the class, even code they came up with on the fly, lending itself well to the live nature of the lecture.

The main reason for structuring lecture demonstrations as Git repositories is to allow students to use the Interactive Code Rewind tool, described in Chapter 3 to re-create the lecture experience after the fact. Figure 5.3 shows a student is viewing commit 2 of `lecture_7`. To provide context, the goal of this lecture was to introduce the basics of drawing simple objects (points, lines, and polygons) with OpenGL. The diff editor makes it clear that lines 24 to 33 were added to the file `Controller.h`, and the notes explain that the newly added lines of code simply draw two points on the canvas, a red point at coordinates `(0, 0)`, and a blue point at coordinates `(0.5, 0.5)`. Figure 5.4 shows the output of the student running the program, in its current state (commit 2).

There are many advantages of using the Interactive Code Rewind tool compared to watching a video recording of the lecture. When students revert the lecture demonstration to a prior state, the tool displays the notes relevant for that state, and all the lines of code that were introduced since the last checkpoint, allowing efficient navigation between all the important milestones in the development. In a video recording, there is a significant amount of time between the important milestones of the program, because it takes time for the instructor to type the code, and explain their thought process. The video does not highlight the important changes in a diff editor, the way the Interactive Code Rewind tool does, as seen in Figure 5.3. Having a bird's eye view of the changes introduced in a given milestone is more efficient and less boring than watching the code appear character-by-character.

The second important advantage of the Interactive Code Rewind tool is that navigating to a prior state of the codebase affects the user's file system, which means

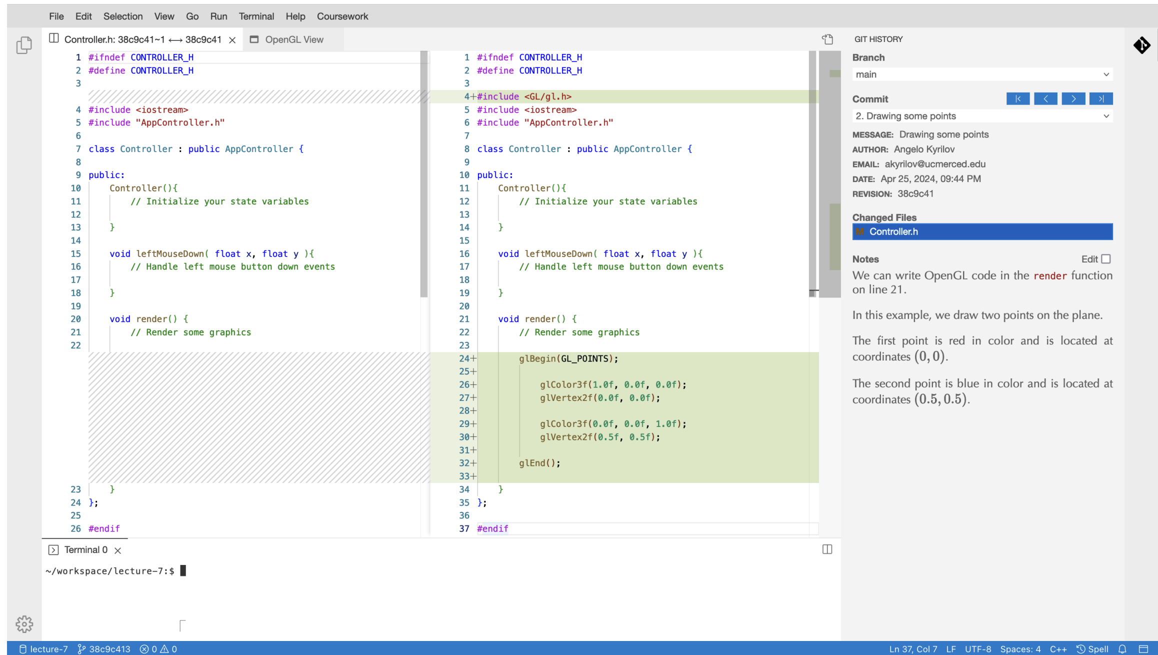


Figure 5.3: Interactive Code Rewind tool for `lecture_7` at commit 2. Includes instructor notes and diff editor with content that changed.

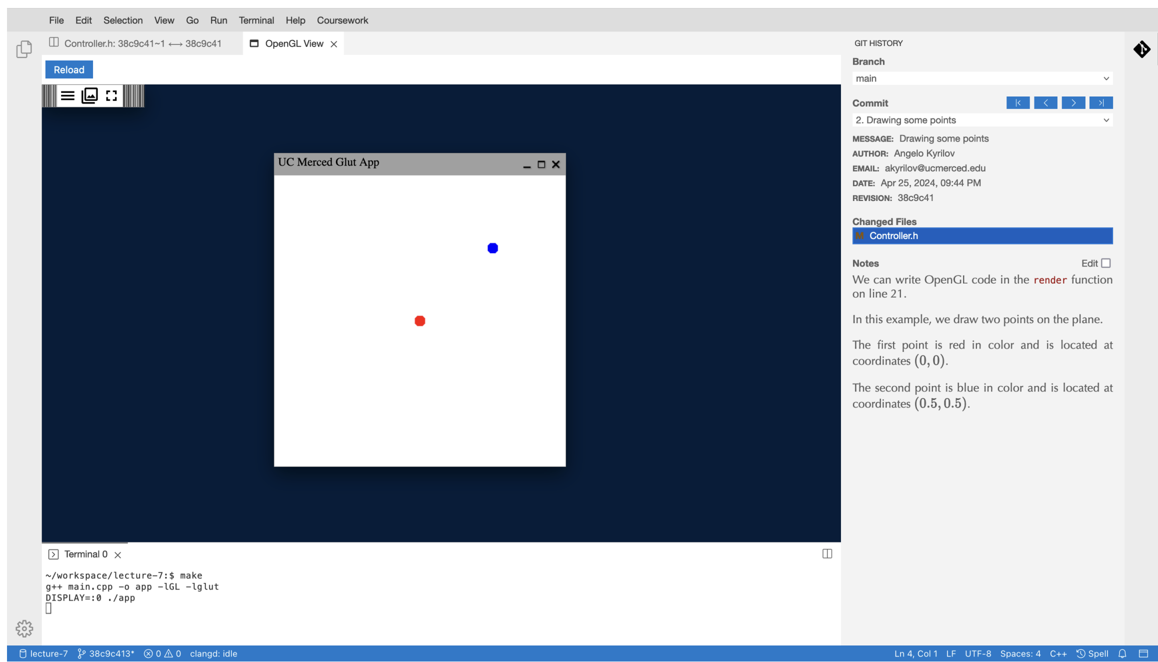


Figure 5.4: Interactive Code Rewind tool showing the output of running the program from `lecture_7` as it existed in commit 2.

that the source code of the project as it existed in a prior state is made available on the user's local file system, allowing them view it in a text editor, and scroll to sections that may not have been visible in a video recording. Since the code is available in the file system, the student can also compile and run it, as seen in Figure 5.4. To be able to do the same from a video recording would have required the student to manually type the code as it appeared in the video, which is a very mundane and error-prone process.

5.2.3 Self Guided Supplementary Exercises with Solutions

In addition to lecture demonstrations, students can also use the Interactive Code Rewind tool, described in Chapter 3, to assimilate course material, especially in the form of programming examples and optional exercises. Traditional programming textbooks and course materials often provide code listings for entire programs. For anything but the most trivial programs, the source code includes nested structures, function definitions, and possibly multiple modules, defined in separate files. This makes listing a complete example impractical and difficult to read, follow and understand. Furthermore, compiling, running, and experimenting with only part of the source code is challenging.

If the programming examples are created as Git repositories, with commits at important milestones, instructors can provide notes for each commit, thereby encoding their thought process and problem solving strategy into the repository. Optionally, the solution can be stored on a separate *branch* of the repository, that students can switch to and from as needed. The advantage of presenting programming examples with this tool is that the code appears for students in small, manageable chunks, clearly highlighted in the diff editor, and explained in the notes section.

Figure 5.5 is a screenshot of a student viewing the solution branch of a programming example, currently at commit 2. The instructor has decided to provide a solution that begins with reading two integers from Standard Input and printing their values to the console. The text editor contains only the code relevant for this step, which is highlighted in green in on the right-hand side of the diff editor, and it is described in the notes. The student has also decided to compile and run the code in its current state and experiment with different input values.

Had the solution been provided in the traditional way, which would have been the entire solution to the problem, with explanations referring to lines in the source code,

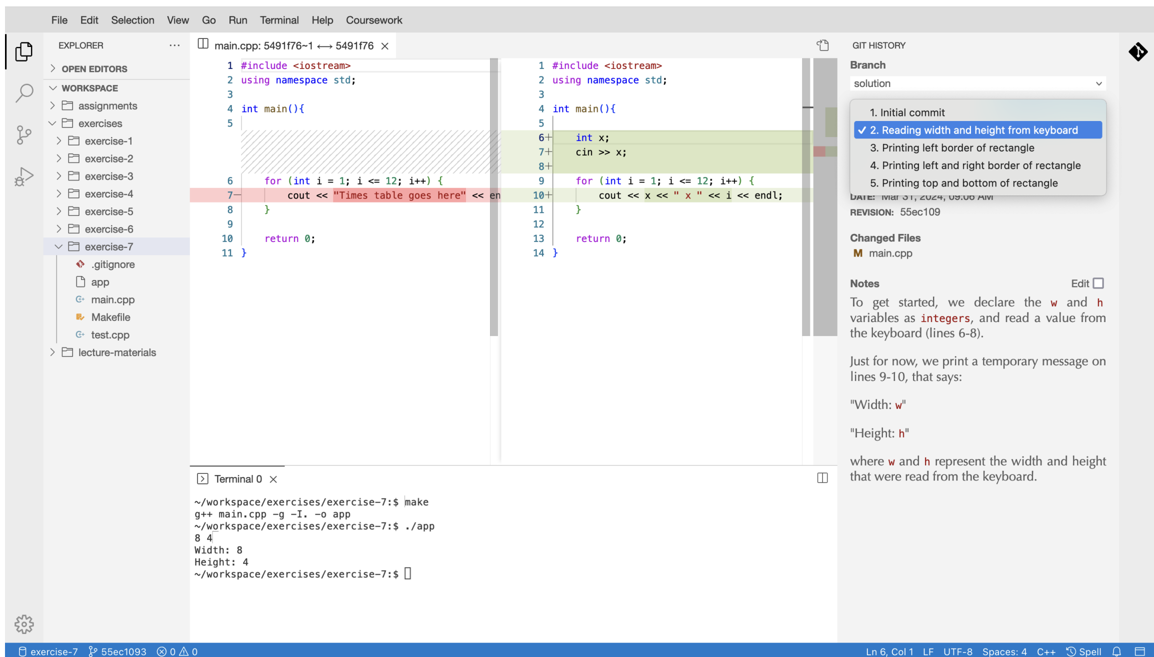


Figure 5.5: The Interactive Code Rewind interface showing *solution* branch for self guided programming exercise.

the student would have had to read the entire source code, and pick out the relevant lines for the current step that they are on, which may not be in order from top to bottom of the text file. The ability to compile and run the program at earlier stages of its development would have also been removed.

Finally, the exercise depicted in Figure 5.5 includes a suite of test cases that students can execute to determine the correctness of the code written. This allows the student to navigate to a prior state of the source code, including to the first commit, and attempt to solve the exercise from that point. They can use the test suite to verify the correctness of their solution. If they are unsuccessful, or do not know how to get started, they need to only navigate to the next commit, where the interface will provide explanatory notes, written by the instructor, as well as relevant code to get started on a solution. In the example presented here, the problem being solved is to produce an ASCII rectangle, given its dimensions. Commit 2 of the example is for students who may not know how to properly get started, so it is suggesting that reading in the dimensions and storing them as integer variables is a good start.

Since examples and exercises of this nature are not assigned for course credit, but merely for additional practice and academic enrichment, there is no incentive for students to plagiarize solutions for them. The provided step-by-step solutions are

only meant to serve as guidelines for students wishing to receive additional practice, with expert guidance available upon request.

The self-guided supplementary exercises with solutions, described above are useful tools for the *Fading* and *Self-directed learning* phases of the Apprenticeship Learning Model.

5.2.4 Product Based Programming Assignments

To fulfill the *Generalization* phase of the Apprenticeship Learning Model, we introduced open-ended laboratory assignments, according to the Product Based Learning philosophy, described is Section 5.1. In the first course offering where this was adopted, students were asked to develop a working clone of the popular Paint application, with only a high level description of the set of features the product needed to have. Each week, students were asked to add features to their existing application, building up a product that was interesting and motivating enough to keep students engaged, with a lot of room for their own creative input to the process.

All students were given boilerplate code that generates a blank graphical window, with sample functions to handle certain events such as mouse clicks, mouse motion, and keyboard input. Students could modify the code in these event handlers in order to implement the desired functionality. At first students were asked to make their application paint and erase anywhere on a given canvas. This was followed by the ability to select various shape tools, as well as a color for the brush tool from a palette of choices. The third iteration asked students to implement a color tool that allows the user to create any color in the RGB spectrum, with the freedom to pick any design they wish, as long as the user is able to perform the task.

Figure 5.6 presents various student implementations for the generic color tool. It is worth noting that despite receiving identical instructions, each student produced a distinct and unique product.

5.3 Evaluation

In this section, we report and discuss the results of adopting of Apprenticeship and Product Based Learning. The original goals of this work were to find instructional and assessment methods that would motivate students to spend more time on their programming, without committing plagiarism. We believe in the “learn by doing”

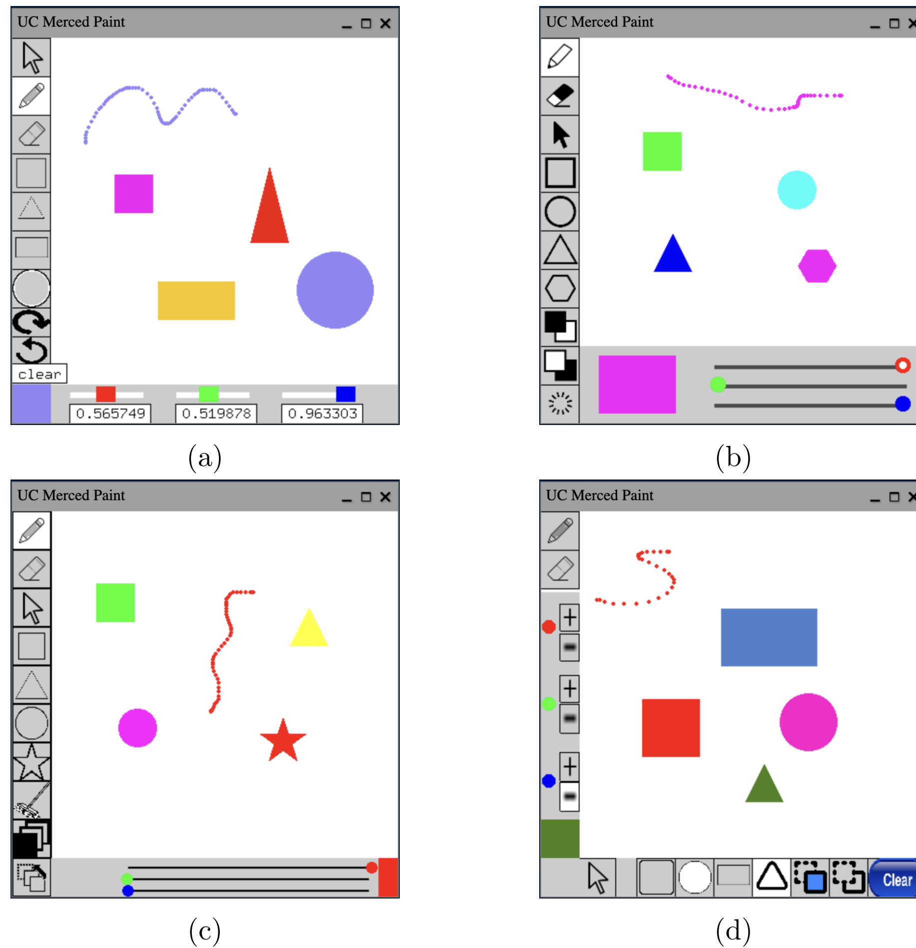


Figure 5.6: Student solutions for Product Based Learning paint application assignment.

approach to computer programming, so it is imperative for students to do more programming. It is equally important for students to work earnestly and follow academic honesty practices, otherwise no learning is taking place. If successful, the teaching methodologies introduced here should also result in higher exam scores, indicative of improved student learning.

5.3.1 Increased Time On Task

In the Spring 2024 offering of CSE 24, we introduced the graphical-based Product Based assignments. Table 5.1 presents the average time students spent working on these assignments. We observed a significant increase in the amount of time students devoted to coding. Students were spending nearly 6 hours per week on their programming assignments, marking a threefold increase compared to the Spring of 2022.

Lab	Spring 2022	Spring 2024
1	151	102
2	77	305
3	76	607
4	67	274
5	56	469
6	72	388
7	62	-

Table 5.1: Average time in minutes spent by students on programming assignments in Spring 2022 and Spring 2024

A possible reason for the additional time and apparent effort students put into their Product Based assignments is the Hawthorne Effect [99], which arises due to the fact that student assignments were assessed by a human grader. Since students knew that their work will be scrutinized by a person, some of them are likely to have tried to impress the grader, who is their instructor or teaching assistant.

Another reason is that programming assignments were long-lived, meaning their products evolved over the course of weeks, and were resembling real software that the students are likely to know or have used. Creating their own versions of such products may invoke a sense of ownership and pride in one's work.

From informal interactions with students, we heard that comments like: "It was fun creating entire programs and applications in OpenGL". Another student commented "What I liked the most about the course was getting to learn how to make

code that would allow me to render graphics in OpenGL. It was fun trying to figure out how to create different colors and a color picker for the paint app".

5.3.2 Decreased Plagiarism in Programming Assignments

Graphical-based Product Based Learning assignments have also mitigated issues related to plagiarism. As discussed in Chapter 4, traditional programming assignments we used previously were plagued with plagiarism. We found that 48% of students were committing plagiarism on the weekly programming assignments they were given. In addition to being an academically dishonest practice, it is also

During the grading process of graphical-based programming assignments, we observed that almost no two students had the exact same visual interface for any programming assignment. We ran MOSS and used our visualization tool and found an average of 8 out of 118 students committed plagiarism on their Product Based Learning assignments. This is only 6.7%, compared to the 48% in prior course offerings.

We believe this was in part due to the open-ended nature of our assignment instructions, which encouraged student creativity. For instance, when tasked with developing a simple paint application, we deliberately refrained from prescribing specific requirements regarding the visual interface's appearance or the implementation details of individual features.

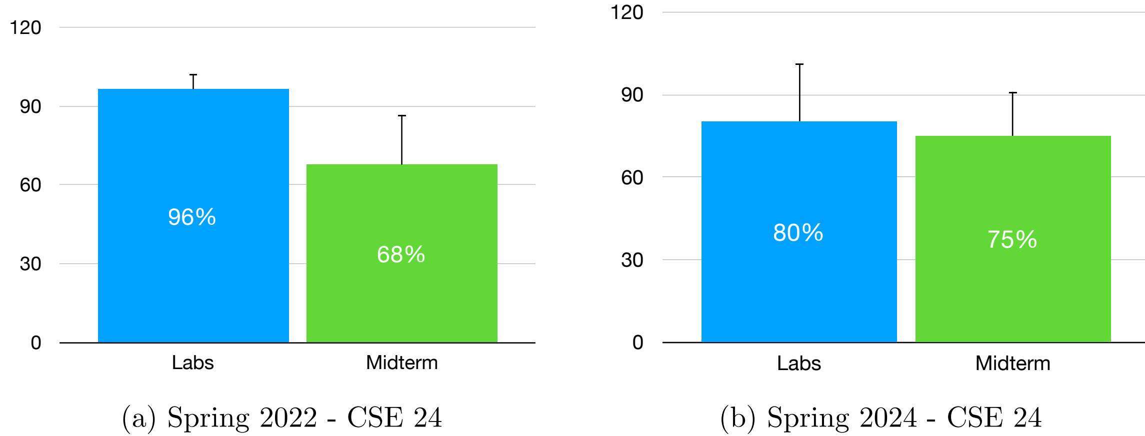
Instead, we provided a set of broad guidelines, outlining features such as a drawing tool and an eraser tool, various shape tools, and color selector. By giving students the freedom to make independent choices in designing their products, we fostered a sense of ownership, thereby increasing their engagement and motivation levels. Consequently, students were motivated to invest longer periods of time to programming, driven by the opportunity to express their creativity and tailor their solutions to their unique preferences.

5.3.3 Improved Learning Rates for Programming Assignments

Given the reduced rates of suspected plagiarism, and the increased time spent on task for programming assignments indicates that students are writing more code without cheating. It is therefore reasonable to expect formal exam scores to increase, as that would be evidence that students have learned. An increased correlation between programming assignment and exam scores would also be indicative of better learning.

In the Spring 2024 semester there were 118 students in the class, the average

programming assignment grade was 80%, while the average midterm grade was 75%. Figure 5.7 shows these averages, with the error bars representing Standard Deviation. Compared to the Spring 2022 semester where the average programming assignment grade was 96% and the average midterm grade was 68%.



(a) Spring 2022 - CSE 24

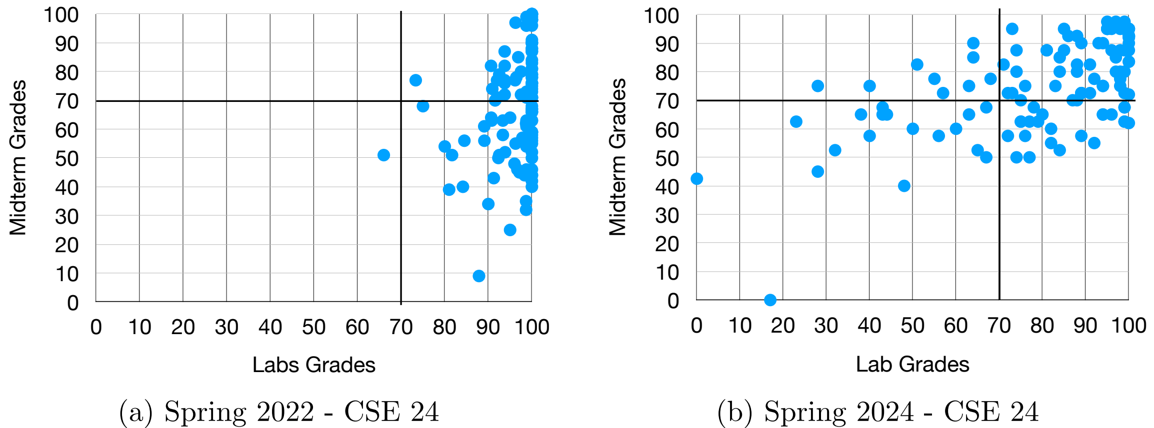
(b) Spring 2024 - CSE 24

Figure 5.7: Average laboratory grades compared to midterm grade in Spring 2022 and Spring 2024

The increase in average midterm grades, from 68% to 75% is an indication that students have a better understanding of the material, since in both courses, midterm exams tested students on concepts seen in laboratory assignments, asking students to demonstrate their understanding. Laboratory grades drop significantly, from 96% in the previous course offering, to 80%. The reduction is not necessarily a negative aspect because grades in Spring 2022 were artificially inflates by plagiarism, which did not happen as much in 2024.

The scatter plot for Spring 2024 in Figure 5.8, with laboratory scores on x-axis and midterm scores on the y-axis, also illustrates a stronger correlation between the grade students receive on their programming assignments compared to their midterm exams. The plots also show a significant reduction in the number of students who score close to perfect on their laboratory assignment, yet fail their exam. We recognize the additional stress associated with taking formal exams, compared to laboratory assignments, so it is reasonable to expect students to perform worse, but the difference should not as drastic as what is observed in Spring 2022.

Table 5.2 shows some statistical factors between the Spring 2022 and Spring 2024 offerings of CSE 24. The Pearson Correlation Coefficient has increased from $\rho = 0.28$ to $\rho = 0.59$, the pass rate on midterm exams has increased from 49.6% to 65.3%,



(a) Spring 2022 - CSE 24

(b) Spring 2024 - CSE 24

Figure 5.8: Scatter plots of laboratory and midterm grades for Spring 2022 and Spring 2024

and the percentage of students who pass their laboratory assignments but fail the midterm exam has decreased from 49.5% to 19.4%.

Feature	**Spring 2022**	**Spring 2024**
Pearson Correlation Coefficient	$\rho = 0.28$	$\rho = 0.59$
Midterm Exam Pass Rate	49.6%	65.3%
Pass Laboratory Assignment Fail Midterm Exam	49.5%	19.4%
Average Time Spent Programming Per Week	89 min	357 min
Percentage of Students Committing Plagiarism	48%	6.7%

Table 5.2: Summary of comparisons between 2022 and 2024 course offerings

The stronger correlations observed above are an indication that students are learning more from their practical assignments. A possible explanation for this is the decreased plagiarism rates, which dropped from 48% to 6.5%. Additionally, students are spending more time working on their GUI based programming assignments. The average time that students spent programming per week was increased from 89 minutes to 357 minutes. We interpret these findings as improved learning rates.

Chapter 6

Conclusion and Future Work

6.1 Dissertation Summary and Discussion

The work presented in this dissertation is motivated by Broadening Participation in Computing (BCP) efforts. It aims to design, evaluate, and deploy curriculum and teaching methodologies that improve student learning of computer programming. The work began with a case study of an introductory programming course at the University of California, Merced. It is believed that challenges faced by UC Merced students are similar to challenges faced by students elsewhere, especially underserved regions where prior access to computing resources has been limited.

The course that served as the vehicle for the initial case study is a typical introductory programming course, with a large practical component, for students to practice their skills, as well as formal exams for students to demonstrate understanding and mastery of the concepts.

The first finding of the study was that students were earning nearly full credit on their programming assignments, while getting significantly lower scores in their midterm exams. This was seen as unusual and suspicious because midterm exams were testing the students on the same concepts as the laboratory exercises. The two forms of assessment highlighted some contradictory cases where students would produce a perfect solution to a programming exercise, demonstrating mastery of the concepts being tested, while exhibiting a complete lack of programming knowledge in the midterm exam only a short time later.

In addition to the discrepancy between laboratory and midterm grades, we also found students were spending short periods of time on their weekly programming assignments. This is interpreted as a missed opportunity by students to practice and

improve their programming skills.

We also suspected high levels of plagiarism in the courses. Traditional plagiarism detection tools reported that about 20-25% of students engage in academically dishonest practices. To investigate the issue further, we built an interface to visualize fine-grained log data available from online Integrated Development Environments (IDEs), allowing us to observe the entire process each student followed while completing their programming assignments. The interface also highlights commonly occurring behavior patterns associated with plagiarism, such as copy/pasting entire solutions to assignments, with or without transformations, or manually entering an externally obtained solution line-by-line, completing an entire assignment in one continuous typing session without making mistakes, or stopping to test the code at intermediate stages, among others. Using the tool we built, we were able to identify twice as many cases of suspected plagiarism as we had from the traditional, similarity-based tools.

To address the issues described above, we created a curriculum, along with supporting teaching methods, and deployed it to the same course as we had originally studied. The curriculum was centered around the open-ended, graphical programming projects that resemble real-world products, thus we named our approach Product Based Learning. Another important feature of Product Based Learning is that projects are graded by human graders. We believe this is a crucial component of motivating students to work on their assignments, since knowing that they will be observed by another person, may motivate them to try and impress that person, a phenomenon known as the Hawthorne Effect.

There is additional labor costs associated with grading graphical products, especially when compared to automatically graded exercises, which are commonly found in Computer Science courses. We argue that the extra effort is worth it because of the motivation it could provide to students, not only through the Hawthorne Effect, but also because they are asked to work on products that resemble real-life software, and because they are allowed to be creative and have input into the process, including user experience (UX) design, problem solving strategies, and others. To streamline the process as much as possible, we built an online grading system, allowing the instructor to examine the source code, not only in its final state, but throughout all its stages of development. It also highlights cases of suspected plagiarism, and provides easy navigation between student submissions, and convenient grade and comment entry, all in one interface.

We also borrow ideas from Apprenticeship Learning for course material deliv-

ery. Many instructors employ live-coding demonstrations in their lectures, which is the first part of apprenticeship learning, having an expert model the process for an apprentice. In lower-division Computer Science courses however, the apprentice to expert ratio could be hundreds to one, so the rest of the Apprenticeship Learning principles of scaffolding, fading, and generalization, are not easily accomplished in a lecture setting.

For a large percentage of students who are still learning the basics, a good scaffolding exercise would be for them to do exactly as the instructor did during the lecture. This has been attempted with video recording of the instructor, or providing the source code of the lecture demonstration, but in our case study, none of these methods were effective.

To address this issue, we started saving the lecture demonstrations as Git repositories, where the instructor can create checkpoints at important milestones in the development of the codebase. Git is widely used in the industry, so even simply using Git in front of the students provides useful exposure for them. In addition to the standard commits, we built an external annotation feature that instructors can use to provide additional notes at each commit. This means that students can follow along the development of the code and get additional explanations at important milestones. The final component of the system was a visual interface, called the Interactive Code Rewind tool, for convenient navigation between the commit points of a demonstration.

Navigating a lecture demonstration with the Interactive Code Rewind tool is like watching the demonstration numerous times, but much more efficiently than watching a recorded video, as the user does not have to wait for the code to be typed in real time. The interface snaps to the next milestone, but it highlights in green the code that has been added since, or in red for code that has been removed. The most valuable aspect of navigating to a milestone in the Interactive Code Rewind tool is that it allows the user to immediately compile and run the code as it existed at that point in time. It is akin to scrubbing to a specific moment in a video of the lecture demonstration to look at the instructor's screen at that time, and have the ability to compile and run the code in the video in the state that it is currently in. The inability to do that is one of the reasons students were not watching the lecture recordings in video format.

The introduction of these modifications in Spring 2024 resulted in various positive outcomes in the course. First, plagiarism was down significantly. Using the same tools

as we did for the 2022 version of the course, we found suspicious behaviors indicative of plagiarism for 6% of the students in the class, whereas it was 48% before. We recognize that since the Product Based assignments are newly introduced, there were no solutions available from previous course offerings, and in future courses there will be, so plagiarism rates may increase.

If the increase is significant, it may be necessary to create completely new Product Based assignments, which is easier than regular, automatically graded exercises, because the instructor needs to only provide high level guidelines of how the product should work. For example, if we find that a solution for the current tic-tac-toe game is available for students, we can add a requirement to the specification that says "Modify your game to include power-ups", without specifying how the power-ups should work, or where they should be integrated. Each student should come up with a different idea for this, and similarities between different students should be easy to spot in the grading platform.

Another potential plagiarism deterrent that can easily be introduced in Product Based Assignments is assigning credit for how impressive the product is, thereby amplifying the Hawthorne Effect because students not only know that they will be observed, but the observer will expect to be impressed.

With the reduction of plagiarism in the 2024 version of the course came an increase in the time students spend on their programming assignments. This is likely due to increased motivation, fewer opportunities to cheat, and the increased size of the assignments. It can be argued that implementing graphical products simply requires more lines of code than implementing a command line program that computes the average of a given list of numbers. We believe that all three reasons are at play, but we argue that no matter the reason for writing more code, a novice programmer benefits from that. Even if the increased time spent on programming assignments was not due to pure interest and motivation, the fact remains that the student wrote more code, spent more time doing it, and did it without cheating, which only increases their learning opportunities.

It also appears that students in the course were learning more effectively, as evidenced by the improved exam grades. The change from 68% to 75% in average midterm grades is a modest increase, but it is in the right direction. An alternative metric is to consider the improved midterm pass rate, which went from 49.6% in 2022, to 65.3% in 2024, which is about a 30% improvement.

Finally, we recognize that drawing definitive conclusions from only one semester

of data is difficult, and further research is needed. We will continue to apply, evaluate and improve the methods presented in this work, in the hopes that it can help more students learn programming more effectively and efficiently, broadening participation in the field from as many people as possible.

6.2 Future Work

6.2.1 Automated Plagiarism Detection with Fine-Grained Data

The plagiarism detection tool described in Chapter 3, allow the instructor to recreate the entire programming process the student followed, by leveraging fine-grained log data collected by our online IDE.

These data are suitable for training Machine Learning models to automatically recognize plagiarism in student submissions, as well as other signs of academic struggle. As a proof-of-concept, we built a binary classification model for plagiarism detection, using TensorFlow and Keras. Table 6.1 shows our machine learning model setup, and Table 6.2 shows the dynamic features used to train our model.

Hyperparameter	Description
Input Layer	11 features
Hidden Layer 1	8 neurons - sigmoid activation
Hidden Layer 2	8 neurons - sigmoid activation
Output Layer	1 neuron - sigmoid activation
Optimizer	Stochastic Gradient Descent (Adam)
Loss Function	Binary Cross Entropy
Training Time	2000 epochs

Table 6.1: Machine learning model setup

We established ground truth values for the model from the process of grading student submissions for two programming exercises, manually classifying each submission as plagiarized or honest.

We trained our model with data from one course section and tested it with data from a different one. The training set consisted of 198 submissions, 56 of which were plagiarized, while the test set had 224 submissions, 78 of which were labeled as plagiarized.

The model achieved an accuracy of 84.85%. This is a promising result, given the limited amounts of data used to train it. It is significantly better than traditional

Feature	Description
Session Duration	Time elapsed during a coding session
Coding Time	Time spent coding (typing)
Idle Time	Time spent idling (not typing)
Keystrokes	Total number of keystrokes
Start File Size	File size at beginning of session
End File Size	File size at end of session
Max Paste	Largest paste length in bytes
Sum Squared Residual	Value used to determine top to bottom programming
Terminal Commands	Total number of terminal commands
Successful Compilations	Total number of successful compilations
Unsuccessful Compilations	Total number of unsuccessful compilations

Table 6.2: Dynamic features extracted from every coding session

plagiarism detection tools, relying on code similarity between different submissions. We expect accuracy to improve further as more data is added to the training set over time.

6.2.2 Elimination of Formal Exams

We also plan to explore the feasibility of eliminating formal examinations and basing the course grade entirely on the practical programming component. Historically, prior to the introduction of Product Based programming assignments, students excelled in the practical programming component but performed poorly on formal exams. This discrepancy was partly attributed to the nature of our textbook-style programming assignments and the high occurrences of plagiarism, which made the practical programming grades ineffective for accurately assessing our students' programming knowledge and proficiency. Consequently, we primarily relied on formal exams for this purpose. Since the implementation of Product Based Learning, we have observed that not all students are achieving perfect scores on their programming assignments. Preliminary data indicates that grades on practical programming assignments are fairly correlated with formal exam grades. This correlation suggests that, with further revisions and improvements to our curriculum and programming assignments, we may be able to predict formal exam scores based solely on the practical programming component. If this proves to be the case, it could allow us to eliminate the need for formal exams entirely.

Appendix A

Data Sets

A.1 Programming Assignments

A.1.1 CSE 24 - Spring 2022

Lab	**Exercise**	**Instructions**
Lab 1	Exercise 1	Create a program that prints "I love CSE! C++ is fun.".
Lab 1	Exercise 2	Create a program that prints an ASCII rectangle.
Lab 1	Exercise 3	Create a program that prints an ASCII triangle.
Lab 1	Exercise 4	Create a program that prints a shipping label for UC Merced.
Lab 1	Exercise 5	Create a program that prints a personalized greeting given a person's name and their favorite hobby.
Lab 1	Exercise 6	Create a program that prints a generic shipping label for a given address.
Lab 1	Exercise 7	Create a program that converts from miles to kilometers.
Lab 1	Exercise 8	Create a program that performs generic arithmetic operations on two integers (Addition, subtraction, multiplication, division, and modulo).
Lab 1	Exercise 9	Create a program that splits the dinner bill between the number of attendees.
Lab 1	Exercise 10	Create a program that splits the dinner bill, including tip percentage, between the number of attendees.

Lab	**Exercise**	**Instructions**
Lab 2	Exercise 1	Create a program that converts any number of days into the equivalent number of years, weeks and days (37 days = 0 years, 5 weeks, and 2 days).
Lab 2	Exercise 2	Create a program that takes in integers, as command line arguments, and determines the ratio of even to odd numbers (More even numbers, more odd numbers, same number of even and odd numbers).
Lab 2	Exercise 3	Create a program that determines the longest consecutive increasing sequence given a list of integers.
Lab 2	Exercise 4	Create a program that prints a parametrized ASCII triangle of a given base and height.
Lab 2	Exercise 5	Create a program that determines the largest 3 digit number that can be created given a sequence of numbers between 0 and 9.
Lab 3	Exercise 1	Create a program that reverses a string (without using any Standard Template Libraries).
Lab 3	Exercise 2	Create a program that finds the longest word in a text (assume words can only be delimited by spaces, commas, or periods).
Lab 3	Exercise 3	Create a program that determines the number of times a substring appears in a string.
Lab 3	Exercise 4	Create a program that uses a shift cipher method to encrypt a string.
Lab 3	Exercise 5	Create a program that determines the number of occurrences of a word in a text.
Lab 4	Exercise 1	Create a function that takes in a reference to a vector and converts all negative integers to their positive counterpart.
Lab 4	Exercise 2	Create a function that returns a vector of the first n prime numbers.
Lab 4	Exercise 3	Create a function that interleaves two vectors.
Lab 4	Exercise 4	Create a function that returns the indices of every pair of consecutive integers in a vector that sum up to a target value.
Lab 4	Exercise 5	Create a function that visualizes a 2D vector.
Lab 5	Exercise 1	Create a program that finds the distance (in bytes) between two variables (in memory).
Lab 5	Exercise 2	Create a program that implements a decode function, given the source code of the encode function and an encoded message.

Lab	Exercise	Instructions
Lab 6	Exercise 1	Create a cryptographic library that takes in an 8-letter word and converts it into a big number.
Lab 7	Exercise 1	Create a resizable data structure which is capable of storing elements of different data types (bool, int, float, char).

A.1.2 CSE 24 - Spring 2024

Lab	Exercise	Instructions
Lab 1	Exercise 1	Create an OpenGL application which draws your initials.
Lab 2	Exercise 1	Create an OpenGL paint application which allows the user to draw, erase, clear the screen, select a stroke thickness, and select a color.
Lab 3	Exercise 1	Update your existing OpenGL paint application and add the ability to draw three different shapes and a selector tool which can be used to move a selected shape.
Lab 4	Exercise 1	Update your existing OpenGL paint application and add an RGB color selector.
Lab 5	Exercise 1	Update your existing OpenGL paint application and add the ability to draw points as a scribble, make all shapes and scribbles moveable by dragging, and have the ability to bring/send selected shapes/scribbles to the front/back.
Lab 6	Exercise 1	Create an OpenGL tic-tac-toe game.
Lab 7	Exercise 1	Update your existing OpenGL tic-tac-toe game and add the ability to play on different board sizes, as well as the ability to play against an AI.

Bibliography

[1] Act college readiness benchmark attainment by annual family income 2015. https://www.act.org/content/dam/act/unsecured/documents/5076-Data-Byte-2015-13-ACT-College-Readiness-Benchmark-Attainment-by-Annual-Family-Income-2015.pdf. Accessed: 2024-04-12.

[2] Center of institutional effectiveness. https://cie.ucmerced.edu/overall-enrollment-totals. Accessed: 2024-04-12.

[3] Atef Mohammad Abuhmaid. The efficiency of online learning environment for implementing project-based learning: Students' perceptions. *International Journal of Higher Education*, 9(5):76–83, 2020.

[4] Sushil Acharya, Priya Manohar, Peter Y Wu, and Bruce R Maxim. Strategies for delivering active learning tools in software verification & validation education. In *2017 ASEE Annual Conference & Exposition*, 2017.

[5] Alireza Ahadi and Luke Mathieson. A comparison of three popular source code similarity tools for detecting student plagiarism. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, ACE '19, pages 112–117, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Alex Aiken. A system for detecting software similarity, 2019.

[7] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[8] Lubna S Alam. Is plagiarism more prevalent in some forms of assessment than others. In *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference*, pages 48–57. Citeseer, 2004.

[9] Ibrahim Albluwi. Plagiarism in programming assessments: A systematic review. *ACM Trans. Comput. Educ.*, 20(1), dec 2019.

[10] Shaban Aldabbus. Project-based learning: Implementation & challenges. *International Journal of Education, Learning and Development*, 6(3):71–79, 2018.

[11] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[12] Ray Bareiss and Martin Radley. Coaching via cognitive apprenticeship. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 162–166, New York, NY, USA, 2010. Association for Computing Machinery.

[13] Theresa Beaubouef and John Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106, jun 2005.

[14] Clara Benac Earle, Lars-Åke Fredlund, and John Hughes. Automatic grading of programming exercises using property-based testing. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 47–52, 2016.

[15] Jess Bidgood and JEREMY B Merrill. As computer coding classes swell, so does cheating. *New York Times*, 6, 2017.

[16] Lidija Bilic-Zulle, Josip Azman, Vedran Frkovic, and Mladen Petrovecki. Is there an effective approach to deterring students from plagiarizing? *Science and engineering ethics*, 14(1):139–147, 2008.

[17] Jacob Bishop and Matthew A Verleger. The flipped classroom: A survey of the research. In *2013 ASEE Annual Conference & Exposition*, pages 23–1200, 2013.

[18] Charles C Bonwell and James A Eison. *Active learning: Creating excitement in the classroom. 1991 ASHE-ERIC higher education reports.* ERIC, 1991.

[19] Steven Bradley. Managing plagiarism in programming assignments with blended assessment and randomisation. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 21–30, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Bear F Braumoeller and Brian J Gaines. Actions do speak louder than words: Deterring plagiarism with the use of plagiarism-detection software. *PS: Political Science & Politics*, 34(4):835–839, 2001.

[21] Russel E. Bruhn and Philip J. Burton. An approach to teaching java using computers. *SIGCSE Bull.*, 35(4):94–99, dec 2003.

[22] Fox Butterfield. Scandal over cheating at mit stirs debate on limits of teamwork. 1991.

[23] Jane E Caldwell. Clickers in the large classroom: Current research and best-practice tips. *CBE—Life Sciences Education*, 6(1):9–20, 2007.

[24] Janet Carter. Collaboration or plagiarism: What happens when students work together. *SIGCSE Bull.*, 31(3):52–55, jun 1999.

[25] Harry Cendrowski and James Martin. The fraud triangle. *The Handbook of Fraud Deterrence*, pages 41–46, 2012.

[26] Hsi-Min Chen, Wei-Han Chen, Nien-Lin Hsueh, Chi-Chen Lee, and Chia-Hsiu Li. Progedu-an automatic assessment platform for programming courses. In *2017 International Conference on Applied System Innovation (ICASI)*, pages 173–176. IEEE, 2017.

[27] Malolan Chetlur, Ashay Tamhane, Vinay Kumar Reddy, Bikram Sengupta, Mohit Jain, Pongsakorn Sukjunnimit, and Ramrao Wagh. Edupal: Enabling blended learning in resource constrained environments. In *Proceedings of the Fifth ACM Symposium on Computing for Development*, pages 73–82, 2014.

[28] Daniela Chuda, Pavol Navrat, Bianka Kovacova, and Pavel Humay. The issue of (software) plagiarism: A student view. *IEEE Trans. on Educ.*, 55(1):22–28, feb 2012.

[29] Allan Collins, John Seely Brown, and Ann Holum. Cognitive apprenticeship: Making thinking visible. *American Educator: The Professional Journal of the American Federation of Teachers*, 15, 1991.

[30] Beth Cook, Judy Sheard, Angela Carbone, and Chris Johnson. Academic integrity: differences between computing assessments and essays. In *Proceedings of the 13th Koli Calling international conference on computing education research*, pages 23–32, 2013.

[31] Beth Cook, Judy Sheard, Angela Carbone, and Chris Johnson. Student perceptions of the acceptability of various code-writing practices. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 105–110, 2014.

[32] Catalina Cortázar, Miguel Nussbaum, Jorge Harcha, Danilo Alvares, Felipe López, Julián Goñi, and Verónica Cabezas. Promoting critical thinking in an online, project-based course. *Computers in Human Behavior*, 119:106705, 2021.

[33] Catherine H Crouch. Peer instruction: an interactive approach for large lecture classes. *Optics and Photonics News*, 9(9):37–41, 1998.

[34] Quintin Cutts, Sarah Esper, and Beth Simon. Computing as the 4th" r" a general education approach to computing education. In *Proceedings of the seventh international workshop on Computing education research*, pages 133–138, 2011.

[35] Karol Danutama and Inggriani Liem. Scalable autograder and lms integration. *Procedia Technology*, 11:388–395, 2013.

[36] Randall S Davies, Douglas L Dean, and Nick Ball. Flipping the classroom and instructional technology integration in a college-level information systems spreadsheet course. *Educational Technology Research and Development*, 61(4):563–580, 2013.

[37] Jason A. Day and James D. Foley. Evaluating a web lecture intervention in a human-computer interaction course. *IEEE Transactions on Education*, 49(4):420–431, 2006.

[38] Breanna Devore-McDonald and Emery D. Berger. Mossad: Defeating software plagiarism detection. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[39] John Dewey. My pedagogic creed. *The School Journal*, LIV(3):77–80, Jan 1897.

[40] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, Trevor Harding, and Cary Laxer. Addressing student cheating: definitions and solutions. *ACM SigCSE Bulletin*, 35(2):172–184, 2002.

[41] Jack Dorminey, A Scott Fleming, Mary-Jo Kranacher, and Richard A Riley Jr. The evolution of fraud theory. *Issues in accounting education*, 27(2):555–579, 2012.

[42] Rob Elliott. Do students like the flipped classroom? an investigation of student reaction to a flipped undergraduate it course. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–7, 2014.

[43] Hakan Erdogmus and Cécile Péraire. Flipping a graduate-level software engineering foundations course. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, pages 23–32. IEEE, 2017.

[44] Seppo Törmä Esko Nuutila and Lauri Malmi. Pbl and computer programming — the seven steps method with adaptations. *Computer Science Education*, 15(2):123–142, 2005.

[45] Ernest Ferguson. Conference grading of computer programs. *ACM SIGCSE Bulletin*, 19(1):361–365, 1987.

[46] Majlinda Fetaji, Bekim Fetaji, and Mirlinda Ebibi. Analyses of possibilities of flipped classroom in teaching computer science courses. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 747–752. IEEE, 2019.

[47] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, pages 10–19, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Nuno Gil Fonseca, Luís Macedo, and António José Mendes. Using early plagiarism detection in programming classes to address the student's difficulties. In *2018 International Symposium on Computers in Education (SIIE)*, pages 1–6, 2018.

[49] Jeff Fortuna, Michael D Justason, and Ishwar Singh. Conversion of a software engineering technology program to an online format: A work in progress and

lessons learned. In *Online Engineering & Internet of Things*, pages 851–858. Springer, 2018.

[50] Scott Freeman, Eileen O'Connor, John W Parks, Matthew Cunningham, David Hurley, David Haak, Clarissa Dirks, and Mary Pat Wenderoth. Prescribed active learning increases performance in introductory biology. *CBE—Life Sciences Education*, 6(2):132–139, 2007.

[51] George M Froggé and Kathryn H Woods. Characteristics and tendencies of first and second-generation university students. *College quarterly*, 21(2):n2, 2018.

[52] Matheus Gaudencio, Ayla Dantas, and Dalton D.S. Guerrero. Can computers compare student code solutions as well as teachers? In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 21–26, New York, NY, USA, 2014. Association for Computing Machinery.

[53] Edward F Gehringer. Reuse of homework and test questions: When, why, and how to maintain security? In *34th Annual Frontiers in Education, 2004. FIE 2004.*, pages S1F–24. IEEE, 2004.

[54] J. Paul Gibson. Software reuse and plagiarism: A code of practice. *SIGCSE Bull.*, 41(3):55–59, jul 2009.

[55] Nuno Gil Fonseca, Luís Macedo, and António José Mendes. Supporting differentiated instruction in programming courses through permanent progress monitoring. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 209–214, New York, NY, USA, 2018. Association for Computing Machinery.

[56] David Gitchell and Nicholas Tran. Sim: A utility for detecting similarity in computer programs. In *The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, pages 266–270, New York, NY, USA, 1999. Association for Computing Machinery.

[57] Olena Glazunova, Tetiana Voloshyna, Valentyna Korolchuk, and Oleksandra Parhomenko. Cloud-oriented environment for flipped learning of the future it specialists. In *E3S Web of Conferences*, volume 166, page 10014. EDP Sciences, 2020.

[58] Tony Greening, Judy Kay, and Bob Kummerfeld. Integrating ethical content into computing curricula. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 91–99. Citeseer, 2004.

[59] Lucas Gren. A flipped classroom approach to teaching empirical software engineering. *IEEE Transactions on Education*, 63(3):155–163, 2020.

[60] Dick Grune. The software and text similarity tester sim.

[61] Dick Grune and Matty Huntjens. Detecting copied submissions in computer science workshops. *Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit*, 9, 1989.

[62] Dirk Grunwald, Elizabeth Boese, Rhonda Hoenigman, Andy Sayler, and Judith Stafford. Personalized attention @ scale: Talk isn't cheap, but it's effective. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 610–615, New York, NY, USA, 2015. Association for Computing Machinery.

[63] Pengyue Guo, Nadira Saab, Lysanne S Post, and Wilfried Admiraal. A review of project-based learning in higher education: Student outcomes and measures. *International journal of educational research*, 102:101586, 2020.

[64] James K. Harris. Plagiarism in computer science courses. In *Proceedings of the Conference on Ethics in the Computer Age*, ECA '94, pages 133–135, New York, NY, USA, 1994. Association for Computing Machinery.

[65] Nina C Heckler, Margaret Rice, and C Hobson Bryan. Turnitin systems: A deterrent to plagiarism in college classrooms. *Journal of Research on Technology in Education*, 45(3):229–248, 2013.

[66] Emlyn Hegarty-Kelly and Dr Aidan Mooney. Analysis of an automatic grading system within first year computer science programming modules. In *Proceedings of 5th Conference on Computing Education Practice*, CEP '21, pages 17–20, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Michael Helmick, Gerald Gannod, and Janet Burge. Using the inverted classroom to teach software engineering. 2007.

[68] Michael Herold, Joe Bolinger, Rajiv Ramnath, Thomas Bihari, and Jay Ramanathan. Providing end-to-end perspectives in software engineering. In *2011 Frontiers in Education Conference (FIE)*, pages S4B–1. IEEE, 2011.

[69] Michael J Herold, Thomas D Lynch, Rajiv Ramnath, and Jayashree Ramanathan. Student and instructor experiences in the inverted classroom. In *2012 frontiers in education conference proceedings*, pages 1–6. IEEE, 2012.

[70] Richard C Hollinger and Lonn Lanza-Kaduce. Academic dishonesty and the perceived effectiveness of countermeasures: An empirical survey of cheating at a major public university. *Journal of Student Affairs Research and Practice*, 46(4):1137–1152, 2009.

[71] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[72] C Jinshong Hwang and Darryl E Gibson. Using an effective grading method for preventing plagiarism of programming assignments. *ACM SIGCSE Bulletin*, 14(1):50–59, 1982.

[73] Derek Hwang, Vardhan Agarwal, Yuzi Lyu, Divyam Rana, Satya Ganesh Susarla, and Adalbert Gerald Soosai Raj. A qualitative analysis of lecture videos and student feedback on static code examples and live coding: A case study. In *Australasian Computing Education Conference*, ACE '21, pages 147–157, New York, NY, USA, 2021. Association for Computing Machinery.

[74] İlhan Ilter. A study on the efficacy of project-based learning approach on social studies education: Conceptual achievement and academic motivation. *Educational Research and Reviews*, 9(15):487, 2014.

[75] Wei Jin and Albert Corbett. Effectiveness of cognitive apprenticeship learning (cal) and cognitive tutors (ct) for problem solving using fundamental programming concepts. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 305–310, New York, NY, USA, 2011. Association for Computing Machinery.

[76] Lynette Johns-Boast and Shayne Flint. Simulating industry: An innovative software engineering capstone design course. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 1782–1788, 2013.

[77] Philip Johnson, Dan Port, and Emily Hill. An athletic approach to software engineering education. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 8–17. IEEE, 2016.

[78] Jplag. Jplag/jplag: Detecting software plagiarism and collusion since 1996.

[79] Kylie Jue. Stanford cs department battles honor code violations, 2014.

[80] Hansi Keijonen, Jaakko Kurhila, and Arto Vihavainen. Carry-on effect in extreme apprenticeship. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 1150–1155. IEEE, 2013.

[81] Pang Nai Kiat and Yap Tat Kwong. The flipped classroom experience. In *2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE&T)*, pages 39–43. IEEE, 2014.

[82] Jennifer K Knight and William B Wood. Teaching more by lecturing less. *Cell biology education*, 4(4):298–310, 2005.

[83] Maria Knobelsdorf, Christoph Kreitz, and Sebastian Böhne. Teaching theoretical computer science using a cognitive apprenticeship approach. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 67–72, New York, NY, USA, 2014. Association for Computing Machinery.

[84] Jeevamol Joy Kochumarangolil and VG Renumol. Activity oriented teaching strategy for software engineering course: an experience report. *Journal of Information Technology Education. Innovations in Practice*, 17:181, 2018.

[85] Dimitra Kokotsaki, Victoria Menzies, and Andy Wiggins. Project-based learning: A review of the literature. *Improving schools*, 19(3):267–277, 2016.

[86] 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*, pages 122–126, 2015.

[87] Maureen J. Lage, Glenn J. Platt, and Michael Treglia. Inverting the classroom: A gateway to creating an inclusive learning environment. *The Journal of Economic Education*, 31(1):30–43, 2000.

[88] Thomas Lancaster and Fintan Culwin. Towards an error free plagarism detection process. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 57–60, New York, NY, USA, 2001. Association for Computing Machinery.

[89] D. Brian Larkins, J. Christopher Moore, Louis J. Rubbo, and Laura R. Covington. Application of the cognitive apprenticeship framework to a middle school robotics camp. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 89–94, New York, NY, USA, 2013. Association for Computing Machinery.

[90] Jaejoon Lee, Gerald Kotonya, Jon Whittle, and Christopher Bull. Software design studio: a practical example. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 389–397. IEEE, 2015.

[91] Alberta Lipson and Norma McGavern. Undergraduate academic dishonesty at mit. results of a study of attitudes and behavior of undergraduates, faculty, and graduate teaching assistants. 1993.

[92] Xiao Liu, Yeoneo Kim, Junseok Cheon, and Gyun Woo. A partial grading method using pattern matching for programming assignments. In *2019 8th International Conference on Innovation, Communication and Engineering (ICICE)*, pages 157–160. IEEE, 2019.

[93] Xiao Liu and Gyun Woo. Applying code quality detection in online programming judge. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology*, pages 56–60, 2020.

[94] Vedran Ljubovic and Enil Pajic. Plagiarism detection in computer programming using feature extraction from ultra-fine-grained repositories. *IEEE Access*, 8:96505–96514, 2020.

[95] Santiago Matalonga, Gastón Mousqués, and Alejandro Bia. Deploying team-based learning at undergraduate software engineering courses. In *2017 IEEE/ACM 1st International Workshop on Software Engineering Curricula for Millennials (SECM)*, pages 9–15. IEEE, 2017.

[96] Eric Mazur and Robert C Hilborn. Peer instruction: A user's manual. *Physics Today*, 50(4):68, 1997.

[97] Donald L McCabe. Cheating among college and university students: A north american perspective. *International Journal for Educational Integrity*, 1(1), 2005.

[98] Donald L McCabe, Linda Klebe Treviño, and Kenneth D Butterfield. Cheating in academic institutions: A decade of research. *Ethics &Behavior*, 11(3):219–232, 2001.

[99] Jim McCambridge, John Witton, and Diana R Elbourne. Systematic review of the hawthorne effect: new concepts are needed to study research participation effects. *Journal of clinical epidemiology*, 67(3):267–277, 2014.

[100] Sarah Murray, James Ryan, and Claus Pahl. A tool-mediated cognitive apprenticeship approach for a computer engineering course. In *Proceedings 3rd IEEE International Conference on Advanced Technologies*, pages 2–6. IEEE, 2003.

[101] Hannah Natanson. More than 60 fall cs50 enrollees faced academic dishonesty charges. *The Harvard Crimson*, 3, 2017.

[102] Kevin A Naudé, Jean H Greyling, and Dieter Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545–561, 2010.

[103] Clifford Nowell and Doug Laufer. Undergraduate student cheating in the fields of business and economics. *The Journal of Economic Education*, 28(1):3–12, 1997.

[104] K. J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.*, 8(4):30–41, dec 1976.

[105] Enil Pajić and Vedran Ljubović. Improving plagiarism detection using genetic algorithm. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 571–576, 2019.

[106] R Pargas and D Shah. Things are clicking in cs4. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*, volume 10. Citeseer, 2006.

[107] Leo Natan Paschoal, Brauner R. N. Oliveira, Elisa Yumi Nakagawa, and Simone R. S. Souza. Can we use the flipped classroom model to teach black-box

testing to computer students? In *Proceedings of the XVIII Brazilian Symposium on Software Quality*, SBQS'19, pages 158–167, New York, NY, USA, 2019. Association for Computing Machinery.

[108] John Paxton. Live programming as a lecture technique. *J. Comput. Sci. Coll.*, 18(2):51–56, dec 2002.

[109] Cécile Péraire. Dual-track agile in software engineering education. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 38–49. IEEE, 2019.

[110] Paul Phillips and Luc Cohen. Convictions of plagiarism in computer science courses on the rise. *The Daily Princetonian, March*, 4:2014, 2014.

[111] L Porter and C Simon Bailey-Lee. B., zingaro, d. peer instruction: do students really learn from peer discussion. In *the 7th Annual International Computing Education Research Workshop*, volume 10, pages 2016911–2016923, 2011.

[112] Leo Porter, Cynthia Bailey Lee, Beth Simon, Quintin Cutts, and Daniel Zingaro. Experience report: a multi-classroom report on the value of peer instruction. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 138–142, 2011.

[113] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci.*, 8(11):1016, 2002.

[114] Michael J Prince and Richard M Felder. Inductive teaching and learning methods: Definitions, comparisons, and research bases. *Journal of engineering education*, 95(2):123–138, 2006.

[115] Pakawan Pugsee. Effects of using flipped classroom learning in object-oriented analysis and design course. In *2017 10th International Conference on Ubi-media Computing and Workshops (Ubi-Media)*, pages 1–6. IEEE, 2017.

[116] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. Role of live-coding in learning introductory programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, Koli Calling '18, New York, NY, USA, 2018. Association for Computing Machinery.

[117] Sadaqat Ur Rehman. Trends and challenges of project-based learning in computer science and engineering education. In *Proceedings of the 15th International Conference on Education Technology and Computers*, ICETC '23, pages 397–403, New York, NY, USA, 2024. Association for Computing Machinery.

[118] Charles P Riedesel, Alison L Clear, Gerry W Cross, Janet M Hughes, and Henry M Walker. Academic integrity policies in a computing education context. In *Proceedings of the final reports on Innovation and technology in computer science education 2012 working groups*, pages 1–15. 2012.

[119] Eric Roberts. Strategies for promoting academic integrity in cs courses. In *32nd Annual Frontiers in Education*, volume 2, pages F3G–F3G. IEEE, 2002.

[120] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer science education*, 13(2):137–172, 2003.

[121] Jacobo Rodríguez, Ana Laveron-Simavilla, Juan M del Cura, José M Ezquerro, Victoria Lapuerta, and Marta Cordero-Gracia. Project based learning experiences in the space engineering education at technical university of madrid. *Advances in Space Research*, 56(7):1319–1330, 2015.

[122] Marc J. Rubin. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 651–656, New York, NY, USA, 2013. Association for Computing Machinery.

[123] Sigrid Schefer-Wenzl and Igor Miladinovic. Game changing mobile learning based method mix for teaching software development. In *Proceedings of the 16th World Conference on Mobile and Contextual Learning*, pages 1–7, 2017.

[124] Sigrid Schefer-Wenzl and Igor Miladinovic. Leveraging collaborative mobile learning for sustained software development skills. In *International Conference on Interactive Collaborative Learning*, pages 157–166. Springer, 2018.

[125] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. Association for Computing Machinery.

[126] Anshul Shah and Adalbert Gerald Soosai Raj. A review of cognitive apprenticeship methods in computing education research. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, pages 1202–1208, New York, NY, USA, 2024. Association for Computing Machinery.

[127] Mary Shaw, Anita Jones, Paul Knueven, John McDermott, Philip Miller, and David Notkin. Cheating policy in a computer science department. *ACM SIGCSE Bulletin*, 12(2):72–76, 1980.

[128] Judy Sheard. In their own words: students and academics write about academic integrity. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 97–106, 2015.

[129] Judy Sheard, Angela Carbone, and Martin Dick. Determination of factors which impact on it students' propensity to cheat. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 119–126. Citeseer, 2003.

[130] Judy Sheard and Martin Dick. Directions and dimensions in managing cheating and plagiarism of it students. In *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*, pages 177–186, 2012.

[131] Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, and Jane Sinclair. Informing students about academic integrity in programming. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 113–122, 2018.

[132] Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. Negotiating the maze of academic integrity in computing education. In *Proceedings of the 2016 ITiCSE working group reports*, pages 57–80. 2016.

[133] David J Shernoff, Suparna Sinha, Denise M Bressler, and Lynda Ginsburg. Assessing teacher education and professional development needs for the implementation of integrated approaches to stem education. *International journal of STEM education*, 4:1–16, 2017.

[134] Simon. Designing programming assignments to reduce the likelihood of cheating. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, ACE '17, pages 42–47, New York, NY, USA, 2017. Association for Computing Machinery.

[135] Beth Simon, Michael Kohanfars, Jeff Lee, Karen Tamayo, and Quintin Cutts. Experience report: peer instruction in introductory computing. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 341–345, 2010.

[136] Beth Simon, Julian Parris, and Jaime Spacco. How we teach impacts student learning: Peer instruction vs. lecture in cs0. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 41–46, 2013.

[137] Robbie Simpson and Tim Storer. Experimenting with realism in software engineering team projects: an experience report. In *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, pages 87–96. IEEE, 2017.

[138] Marilyne Stains, Trisha Vickrey, Kaitlyn Rosploch, Reihaneh Rahmanian, and Matthew Pilarz. Research-based implementation of peer instruction: A literature review. *CBE life sciences education*, 14, 02 2015.

[139] Paul Stapleton. Gauging the effectiveness of anti-plagiarism software: An empirical study of second language graduate writers. *Journal of English for Academic Purposes*, 11(2):125–133, 2012.

[140] Thomas Staubitz, Hauke Klement, Jan Renz, Ralf Teusner, and Christoph Meinel. Towards practical programming exercises and automated assessment in massive open online courses. In *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 23–30, 2015.

[141] Michael Stepp and Beth Simon. Introductory computing students' conceptions of illegal student-student collaboration. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 295–299, New York, NY, USA, 2010. Association for Computing Machinery.

[142] Bethany B Stone. Flip your classroom to increase active learning and student engagement. In *Proceedings from 28th Annual Conference on Distance Teaching & Learning, Madison, Wisconsin, USA*, 2012.

[143] Tonghua Su, Shengchun Deng, Xiaofei Xu, Dong Li, and Zhiying Tu. Principled flipped learning paradigm for laboratory courses in software engineering. In *Software Engineering Education Going Agile*, pages 123–128. Springer, 2016.

[144] Narjes Tahaei and David C. Noelle. Automated plagiarism detection for computer programming exercises based on patterns of resubmission. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 178–186, New York, NY, USA, 2018. Association for Computing Machinery.

[145] Mohamed Tarek, Ahmed Ashraf, Mahmoud Heidar, and Essam Eliwa. Review of programming assignments automated assessment systems. In *2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, pages 230–237. IEEE, 2022.

[146] Egon Teiniker and Gerhard Seuchter. Improving the flipped classroom model by the use of inductive learning. In *2020 IEEE Global Engineering Education Conference (EDUCON)*, pages 512–520. IEEE, 2020.

[147] Neena Thota, Gerald Estadieu, Antonio Ferrao, and Wong Kai Meng. Engaging school students with tangible devices: Pilot project with .net gadgeteer. In *Proceedings of the 2015 International Conference on Learning and Teaching in Computing and Engineering*, LATICE '15, pages 112–119, USA, 2015. IEEE Computer Society.

[148] Dave Towey. Lessons from a failed flipped classroom: The hacked computer science teacher. In *2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 11–15. IEEE, 2015.

[149] Herbert H. Tsang, Alice Schmidt Hanbidge, and Tony Tin. Experiential learning through inter-university collaboration research project in academic integrity. In *Proceedings of the 23rd Western Canadian Conference on Computing Education*, WCCCE '18, New York, NY, USA, 2018. Association for Computing Machinery.

[150] Brad Vander Zanden and Michael W Berry. Improving automatic code assessment. *Journal of Computing Sciences in Colleges*, 29(2):162–168, 2013.

[151] Fatima Vapiwala and Deepika Pandita. Strategies for effective use of gamification technology in e-learning and e-assessment. In *2022 7th International Conference on Business and Industrial Research (ICBIR)*, pages 596–601. IEEE, 2022.

[152] Nécio L. Veras, Lincoln S. Rocha, and Windson Viana. Flipped classroom in software engineering: A systematic mapping study. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, pages 720–729, New York, NY, USA, 2020. Association for Computing Machinery.

[153] Kristina L Verco and Michael J Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *ACM International Conference Proceeding Series*, volume 1, pages 81–88, 1996.

[154] Arto Vihavainen and Matti Luukkainen. Results from a three-year transition to the extreme apprenticeship method. In *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 336–340, USA, 2013. IEEE Computer Society.

[155] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. Massive increase in eager tas: experiences from extreme apprenticeship-based cs1. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 123–128, New York, NY, USA, 2013. Association for Computing Machinery.

[156] Dieter Vogts. Plagiarising of source code by novice programmers a "cry for help"? In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, pages 141–149, New York, NY, USA, 2009. Association for Computing Machinery.

[157] 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, 2013.

[158] Vladimir Vujović, Mirjana Maksimović, and Branko Perišić. The different active learning strategies in software engineering and their effectiveness. *Proc. 7th ICERI*, pages 3183–3193, 2014.

[159] Neal R Wagner. Plagiarism by student programmers. *The University of Texas at San Antonio Division Computer Science San Antonio, TX*, 78249, 2000.

[160] Michael J Wise. Yap3: Improved detection of similarities in computer program and other texts. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 130–134, 1996.

[161] Denise Woit and David Mason. Effectiveness of online assessment. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 137–141, New York, NY, USA, 2003. Association for Computing Machinery.

[162] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. Tmoss: Using intermediate assignment work to understand excessive collaboration in large classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 110–115, New York, NY, USA, 2018. Association for Computing Machinery.

[163] YT Yu, Chung Man Tang, and Chung K Poon. Enhancing an automated system for assessment of student programs using the token pattern approach. In *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 406–413. IEEE, 2017.

[164] Daniel Zingaro. Experience report: Peer instruction in remedial computer science. In *EdMedia+ Innovate Learning*, pages 5030–5035. Association for the Advancement of Computing in Education (AACE), 2010.

[165] Soundous Zougari, Mariam Tanana, and Abdelouahid Lyhyaoui. Hybrid assessment method for programming assignments. In *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*, pages 564–569. IEEE, 2016.