Informed Intervention Design, Deployment, And Analysis for the Computer Science Classroom

Jaxton J. Winder
Utah State University

INFORMED INTERVENTION DESIGN, DEPLOYMENT, AND ANALYSIS FOR THE COMPUTER SCIENCE CLASSROOM

by

Jaxton J. Winder

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

UTAH STATE UNIVERSITY
Logan, Utah

2024

ABSTRACT

Informed Intervention Design, Deployment, and Analysis For The Computer Science Classroom

by

Jaxton J. Winder, Master of Science
Utah State University, 2024

Major Professor: John Edwards, Ph.D.
Department: Computer Science

Improving the computer science classroom is the goal of many educators and researchers. Computing education researchers often do this through the introduction of targeted interventions to the classroom. These interventions are used in an effort to improve student learning and assist instructors in their teaching. Computing education researchers frequently aim to support intervention design with research. Understanding the effects of these interventions through continued research enables education researchers to improve and share best practices with the scientific and education community. In this thesis, I will be detailing novel research on computer science classroom interventions and related topics that can support the informed design of continued interventions. One chapter of this thesis details an intervention tackling student procrastination on software projects, with significant success. Another chapter studies a practice that many computing educators assume is good for students: incremental development. Through the practice of data mining using an open source dataset of USU CS1 student keystrokes, we find evidence supporting this practice correlating with positive student outcomes, supporting future computing education researchers to develop interventions to teach students the practice of incremental development. The last chapter details research and interventions aimed at teaching students the command shell. This chapter details a novel intelligent tutoring system for the UNIX command shell, the Shell Tutor, and includes an analysis of student perspectives on this tool, which are overwhelmingly positive.

All together, this thesis demonstrates the successful deployment of interventions in the computing education classroom. It also details research and the creation of tools that enable educators and researchers to continue work towards improving the computer science classroom.

(74 pages)

PUBLIC ABSTRACT

Informed Intervention Design, Deployment, and Analysis For The Computer Science Classroom

Jaxton J. Winder

Improving the teaching of computer science is a challenging task. Educators and computing education researchers devote large amounts of time, energy, and resources towards doing so effectively. One of the ways this is done is through research-informed design, deployment, and analysis of targeted interventions to the classroom. This thesis will detail research conducted at Utah State University targeting classroom interventions: centered around their design, deployment, and analysis.

One of these interventions aims to tackle student procrastination through the offering of "grace points"–forgiving a small amount of mistakes on a student's assignment–for analyzing a homework assignment early. Through studying this intervention, we found evidence supporting 1) the intervention caused students to start on their assignments an average of two days earlier over a twelve day assignment period and 2) that starting earlier correlated with a nearly 8-percentage point improvement on assignment scores.

The next section of this thesis details research that can support future intervention design. It is a commonly held belief that students coding a little bit and testing their work frequently–a process called incremental development–is a good practice. However, there was a lack of research on this topic. Utilizing a tool to analyze student development behaviors, we see that this practice of incremental development correlates with more efficient work on programming assignments. In our study, CS1 students were able to complete homework assignments in half the time and write half as much code, while still achieving the same scores on homework assignments. This research supports educators and researchers in their work towards designing classroom improvements and interventions which may help student complete assignments more efficiently.

Lastly, this thesis analyzes and details the impacts of a novel tool which helps students learn the command shell. The command shell is a tool many early computer science students struggle to learn. In the USU CS1440 course, we have introduced a new tool–the Shell Tutor–which aims to tutor students through their learning of the command shell. This thesis provides a detailed analysis of this tool and finds that students have overwhelmingly positive opinions about this tool. We also identify ways which students believe this tool may be better integrated into the classroom in the future.

All together, this thesis demonstrates the successful deployment of interventions in the computing education classroom. It also details research and the creation of tools that enable educators and researchers to continue work towards improving the computer science classroom.

This work is dedicated to all in my life who have helped shape me into the person I am today, giving me the many opportunities and support needed to accomplish all that I have accomplished. It is also dedicated to the curious student who is always striving to learn more.

ACKNOWLEDGMENTS

A special thanks to Erik Falor, who has been an amazing mentor of mine throughout my time in the Utah State University Computer Science Department. He assisted me greatly in learning the “command shell-fu” and professional software development skills necessary to accomplish this research and enabling me in providing valuable feedback and suggestions along the way. He also gave me many wonderful opportunities to learn and practice becoming an educator, allowing me to design, deploy, and analyze interventions and teaching tools in his courses.

Dr. John Edwards, my advisor and a man I admire greatly, deserves praise and gratitude for helping me find my path within the field of computing education research. He opened up a vast new world to a hopeful undergraduate who desired to assist in making aspects of education better and more accessible for all, kick-starting my formal journey into computing education research.

Additional thanks go to the educators who have shaped me into who I am today. Without the many teachers in my life each making their own unique and positive impact on me, there is no way I could be where I am today. As much as I would love to list out each one by name and note their contributions to me, it would not be appropriate to make these acknowledgements longer than the thesis it is contained in.

Last–and most definitely not least–a special thanks go out to my parents, Bryan and Julie Winder, who have supported me always. They have fostered my creativity and encouraged me to continually grow, providing for me when I have been unable to provide for myself. Without them, none of this would have ever been possible.

Jaxton J. Winder

CONTENTS

CHAPTER 1
INTRODUCTION

Educators frequently aim to improve their classroom through various means. One of these means is the introduction of targeted interventions in order to effect student behaviors, support healthy work habits, and make the learning of challenging topics more accessible. Interventions are often uniquely tailored to the fields of education they are deployed in, and computer science education is no different. In this thesis, I will be detailing 3 research projects centered around the informed design, deployment, and analysis of interventions for the computer science classroom.

Chapter 2 details the analysis and deployment of an intervention focused on student procrastination. Students submitting project milestones early correlates with better performance on project submissions as a whole. In our study, these milestones correlate with the creation of documentation for analyzing the project they are to work on. These milestones are submitted at least 5 days before a project is due during a nearly 2-week assignment period. There is statistical significance in positive correlation between assignment performance and early submissions of these project milestones for analyzing a project's requirements. Additionally, there is statistical significance that incentivizing students with "grace points" for doing this early analysis changes student behavior, leading to earlier work on projects.

Chapter 3 will outline the analysis and supporting research of a commonly accepted "good practice" in computing education that has had a dearth of research supporting the idea. This chapter reports an analysis of incremental development: the process in computer science education where students code a small amount and run their code regularly, making continual forward progress. We use a recently published measure of incremental development to evaluate reconstructed submissions collected from CS1 students using keystroke and IDE data. An element that was lacking in the original paper was an analysis of correlation of incremental development with academic outcomes, which our study does. We mark submissions as incrementally developed or not and we then analyze the keystroke behaviors between these submissions and final assignment performance. We also study if incremental development is correlated with grades, number of assignment interaction events, time spent on the assignment, number of coding sessions to complete the assignment, as well as other metrics. We present evidence that incremental development is correlated with time spent on the assignment, number of times the code was executed by the student, and similar measures. We do not find evidence that incremental development is correlated with assignment score or final grade in the course. Our analysis is consistent with the belief that incremental development will help students complete programming assignments more quickly but it does not support the idea that incremental development will result in better grades.

In Chapter 3, statistical significance is found correlating incremental development with the number of assignment interaction events, time spent on the assignment, number of times the code was executed by the student, and percentage of interactions done before testing the code in a coding session. No significance is found correlating assignment score, time between interactions, number of coding sessions, time between code executions, and number of incrementally developed projects with a student's final grade. This work supports the notion that incremental development is a good practice for students, but for different reasons than initially theorized by computing education researchers.

Chapter 4 discusses the design, deployment, and analysis of a novel intelligent tutoring system developed at Utah State University called the "Shell Tutor." The command shell and Git are important tools for computer scientists to learn and is taught in many computer science curricula. Many tools used by computer scientists are primarily interfaced with through the command shell, such as Git. However, there have been few studies and interventions designed to assist in understanding student behaviors in the command shell and better teaching of it. This chapter aims to provide an overview and reflection of this novel intelligent tutoring system we developed which assists in teaching students the command shell called the Shell Tutor. This chapter will also analyze features of this tool to better understand student behaviors within the command shell while using this intelligent tutoring system: a logging system which will enable researchers to better understand student behaviors in using the tool and the command shell in general. A study done with students who used this tool illuminates the perceived effects on student learning and their perspectives of the tool, which are overwhelmingly positive.

In its entirety, this thesis will showcase research with a central theme of improving the computer science classroom. It also demonstrates the successful deployment of interventions in the computing education classroom and the creation of tools and research that could enable future computing education researchers to continue work towards improving the computer science classroom.

CHAPTER 2
Early Submission of Project Analysis Milestones Correlates Positively With Student Project Performance; Incentives for This Early Project Analysis Positively Changes Student Behaviors

2.1 Introduction

Students procrastinating assignments is a tale as old as time. Well, at least as old as the first ever homework or publication deadline. Procrastination on assignments oftentimes leads to students missing key information about the assignment given in classes and tends to be correlated with poorer assignment performance [1, 2].

We hypothesize that encouraging students to start on assignments early–at least to the point of understanding what exactly the assignment entails and designing a brief plan for the assignment–will improve outcomes. By starting early on assignments, students will become more capable of connecting discussions in class with what they are asked to do on the assignment. Additionally, having an early plan for the assignment will allow a student additional time to adjust their plan for the assignment as they learn more. For software projects, we suggest that early planning will also lead to more robust programs, better documentation, and projects that meet more of the desired requirements.

In this paper, we report results of a study in which we incentivized the early completion of a project analysis milestone. Incentives were offered by allowing students to recover points missed on an assignment, dubbed "grace points." Forgiving small amounts of mistakes on a submission in exchange for students starting early may positively change student behaviors; a worthy trade if it encourages students to make less mistakes and perform better overall on the assignment.

This paper studies student submission behaviors in an early Computer Science course taught at Utah State University (USU), titled CS1440: Methods of Computer Science and the applied grace points intervention for early project analysis milestone submissions. We find that this intervention leads to considerable changes in student assignment behaviors both within the semester the intervention was deployed and when compared to other semesters the course was taught. Additionally, we found that students completing these project deadlines earlier does result in more robust programs, better documentation, and projects that meet more of the desired requirements. Our research questions are: **does achievement of early project analysis correlate with better assignment performance** and **does the proposed grace point intervention positively modify student behavior?**

This paper may impact the Computer Science education community by leading to more informed design of course interventions. If it is shown that early completion of a project requirements analysis milestone correlates positively with student performance, then interventions can be designed to encourage students to achieve this milestone earlier in the life-cycle of a software project. Additionally, if it is shown that the "grace points" intervention positively influences student assignment behaviors, more interventions have been identified such that student work ethics are positively influenced.

2.2 Related Work

It has been found that as procrastination increases, poor academic performance also increases. This reference shows an adverse relationship between procrastination and academic performance [1, 2]. Conversely, this study shows that there is a positive correlation between higher grade point averages and low academic procrastination [3]. Decreasing student procrastination is a worthwhile target when designing course interventions.

Additionally, a model was defined that reliably predicts student success and failure in a course based on assignment submission behaviors. The model was able to accurately predict that students who spend a short time on the assignment and submit assignments later are more likely to perform poorly in a course when compared to those that submit assignments later but spend more time on the assignment. This seems to indicate that students starting early on assignments lead to better course performance, even if they submit the assignment later in the assignment period [4]. A key aspect of our proposed intervention is that it does not require early submission of the entire assignment, just the early completion of one phase of it.

Other studies aim to identify low-cost and low-effort interventions that reduce student procrastination, with some promising results [5]. An additional study found that incentives for better work habits encourage students to develop these better work habits. Felker and Chen state in their study on incentivizing work habits, "Our results suggest that assigning extra credit for completing parts of the assignment early can be an effective method to encourage better work distribution and longer study time for both high and low performing students" [6].

Case studies were conducted with undergraduate and masters level students creating software projects and employing different paradigms for software development where it was found that groups with plan-based or agile approaches to software development produced software that met more functional requirements and were of higher quality from an external user perspective [7]. Another study found that the academic software development education environment mirrors the software development industry and existing software development paradigms within education mirrors the benefits of these same paradigms in industry. [8]

A study was conducted which introduced students to the process of discovering software requirements and served as an introduction to requirements engineering and analysis. This was done in the context of the students' final year project, thus the problems analyzed by each students differed greatly. Through the process of creating a problem statement, identifying the stake holders in such a project, documenting required functionality, and writing a list of all functional requirements, notable benefits were found. This study found that their approach assisted students in turning ill-defined problems into well-defined ones. [9] The requirements analysis phase of the software development plan employed in the CS1440 course emulates this process at a smaller scale.

All of these studies support various aspects of the authors' hypothesis and show that there is clear value to be had in designing interventions that aim to change student assignment behaviors.

2.3 Background Of Course Studied: USU CS1440

The USU CS1440: Methods Of Computer Science course is a course that aims to assist students in developing the soft-skills needed to become a "good" software developer and computer scientist. This course is taught in Python, but the programming language is not the focus of this course. The course instead focuses on teaching principles and practices that help students manage program complexity, improve software quality, and leverage existing or new technologies. Students of this course are exposed to the Git version control system and learn the basics of the Unix command line interface. Student project submissions are done with Git by pushing projects to a self-hosted GitLab server with tracking capabilities.

The USU CS1440 course is taken after the CS1 course which is taught in Python. CS1 is a prerequisite for CS1440. The CS1440 course is typically taken in conjunction with a CS2 course taught in Java. Students are not required to take CS1440 in conjunction with CS2, and may take CS1440 any time after CS1.

The CS1440 course is required for computer science major and minor students at Utah State University. However, many students enrolled in the course are using it to meet general education requirements for other departments, and are not required to continue beyond this course. The students of this course have very diverse academic backgrounds. The enrollment in this course ranges between 100-200 students depending on the semester, oftentimes split between two sections.

Students in this course produce programming projects roughly once every 2 weeks. During a 15-week semester–accounting for holidays, the end of the semester, and an early assignment that only focuses on the shell and Git fundamentals–students produce 7 programming projects of increasing complexity and each with various focuses.

2.3.1 Submissions In CS1440

Student submissions in this course are required to use Git, and are pushed to a self-hosted GitLab server where the dates that certain commits are pushed are easily tracked. Usage of Git enables the capability to track student project development over time by analyzing different commits on the project and the files that change at each commit.

Students in the course are required to follow a course "software development plan" which has 5 key phases: an `analyzed` phase, a `designed` phase, an `implemented` phase, a `tested` phase, and a `deployed` phase. The `analyzed` phase consists of producing documentation that asks a student to analyze the problem at hand, restate the requirements in their own words, and note key attributes of correct and incorrect solutions. This phase is crucial for our study. The `designed` phase consists of producing documentation that asks a student to "plan their attack", producing pseudo-code and document design plans. The `implemented` phase is where the student actually *writes* the code and note key design changes needed. The `tested` phase is where the student produces documentation of their testing. Lastly, the `deployed` phase consists of the student ensuring their program is "customer ready" and polishes off documentation, ensuring the project is ready for grading.

Students gain familiarity with Git and the expected software development plan structure on the first two project assignments. After that they are expected to create Git tags which mark the commit that represents the end of each of these key phases. These key phases then become project milestones, marking the state of the project where certain "software development plan" phases are met. The students are given proper training of how to produce Git tags and verify they did them correctly.

2.3.2 Assessment Of Submissions In CS1440

Student projects are assessed according to a few metrics. Documentation of the phases mentioned above is a significant factor in project assessment. Additional developer and end user documentation of the program created may be a factor for many of the projects assigned. Project assessment is also based on the performance of the project created, behavior under certain test cases, and design patterns followed. The student's code is also directly analyzed for certain attributes that are not permitted, such as the usage of `eval/exec` functions, importing of unapproved modules, hard-coding of absolute file paths, and other code features that are deemed unacceptable. Students have access to a full list of these disallowed code features that result in submission penalties.

Given the size of the course, submissions are analyzed by teacher's assistants (TA's), with direction and guidance by the instructor and other TA's with seniority. Rubrics are provided that allocate point values to certain parts of a given project. Once the Git tagging of various project milestones is required to note the completion of these milestones, graders are instructed to analyze the existence of documentation at these milestones to ensure students are following the software development plan. Graders for this course utilize an interactive grading script tool that aims to assist them with their grading and ensure uniformity of assessment, which ensures documentation at each phase is assessed.

2.4 "Grace Points" For Early Milestone Submission Intervention

To encourage students to avoid the pitfalls of procrastination, we designed an intervention that encourages students to start analyzing and designing their project ahead of time. For the assignments that require students to make Git tags marking the ends of the various design phases, an incentive is given for students finishing the analyzed and/or designed tags by the end of the weekend before the assignment is due. During the semester studied, these early submission deadlines are usually 5 days before the assignment's due date, with slight variations due to the semester schedule. All assignments discussed in this paper have the early submission deadlines 5-7 days before the assignment's due date.

Incentives for these early submission deadlines are provided through the allocation of "grace points." Grace points allow a student to receive missed points back on their submissions, without exceeding the maximum point value. These grace points produce a small padding to students' grades if small mistakes are made without awarding them points beyond the scope of the assignment. They are only allocated on the assignment they are received on. Students are eligible for 5 grace points if they finish the `analyzed` phase and push the `analyzed` tag before this deadline. After completion of the analysis phase and pushing the `analyzed` tag, students are eligible for another 5 grace points if they finish the `designed` phase and push the `designed` tag before this deadline as well. Project point values vary, but students may earn back 8%-12% due to grace points.

2.5 Methods

To answer the research questions we break down the methodology into 3 key sections: grace points analysis, data collection, and data analysis. We aim to track if students who receive these grace points generally achieve better scores on the assignments when compared to their peers that do not achieve these grace points, once the score adjustment for the grace points is removed. Additionally, we aim to find if behaviors of students starting on assignments is changed by the introduction of the grace points intervention.

2.5.1 Grace Points Analysis

Students in the CS1440 course achieve grace points for completing either the `analyzed` or `designed` phases of their projects early. The completion of these phases can be found by identifying the creation of the Git commits that are tagged noting the end of each of these phases and the commit creation can be verified by analyzing the commit push logs of the GitLab server, to control for students manipulating the commit creation timestamps. Students Git repositories can be easily analyzed to ensure the correct work was completed in each of these phases.

Students achieving these grace points on submissions are tracked during assignment assessment. It is known exactly how many points are applied as grace points on each submission, and scores were adjusted to remove the score alteration by these grace points. This data is marked in the Canvas Learning Management System used by the course. Submissions that are marked as having received grace points are guaranteed to have completed the project analysis documentation early.

2.5.2 Data Collection

Data collection on the performance effects of early project analysis was done with existing grade data for 2 sections of the CS1440 course in the Fall 2022 semester. Three of the assignments were analyzed that were all eligible for grace points; Assignment 3: Big Data Processing, Assignment 4: Bingo! Card Generator, and Assignment 5.0: Fractal Visualizer - Refactoring. The assignments are described briefly in subsections below.

Data collection to understand the effects on students starting on submissions included data from all sections of the CS1440 course in the Fall 2021 and Spring 2022 semesters. Data was collected noting the time stamp of the first git commit made by a student, normalized over the assignment window. Data for the Fall 2021 and Spring 2022 semesters included the assignments noted earlier, as well as Assignment 6: Recursive Webcrawler, as this assignment was eligible for grace points in the Fall 2022 semester. For another analysis of just the Fall 2022 semester first commit timestamps, data for the assignments titled Assignment 1: The DuckieCrypt Decrypter and Assignment 2: Text Tools were included, as these assignments were before the grace points intervention was introduced to the students. These additional assignments are not discussed in detail due to the only metric measured is first commit times. All assignments were available over a roughly two week time-period. For the Spring 2022 semester, time-data for Assignment 4: Bingo! Card Generator was removed due to the time-period being 3 weeks due to spring break.

Additional Data Collection Notes For Performance Analysis

Data for each submissions performance was compiled and it was noted if students received any form of grace points on a given assignment. Scores were adjusted to remove the effect of receiving grace points on a submissions final results. Any other grade modifications not relative to assignment performance was also accounted for and scores were adjusted.

Assignment data for students who did not make any new commits–indicating no work was done–were removed. If a modification was made to the project and pushed to GitLab, the submission was part of the data collected. Students may have started working on the assignment at any time, but never end up creating the "final product," and their work was still graded and counted as part of the data collected. Both students receiving grace points and students not receiving grace points fell into this group of students who did not "finish" the product, and partial points may be awarded based on documentation created and other work done.

Additionally, certain attributes of submissions are strictly forbidden and clearly noted to the students. Submissions with these attributes have their scores reduced to 0. Students receiving grace points and students not receiving grace points accrued these penalties in the assessed submissions included in this study.

Assignment 3: Big Data Processing

Students are tasked with parsing a large `.csv` file provided by the United States Bureau of Labor Statistics (with small modifications by the instructor), and reporting on the data present in the file. Students are required to read external documentation to understand the format of this data, and are encouraged to make smaller data sets from this large data-set for validation, with guidance from the instructor. Students *initially* perceive this assignment to be very challenging, but frequently find that the implementation is fairly simplistic once they understand the structure of the assignment. This project is the project that receives the highest average scores of the semester.

Assignment 4: Bingo! Card Generator

Students are tasked with creating an Object Oriented program that allows a user to interactively generate Bingo cards of various sizes and interact with them. There is required documentation to be produced, such as a UML class diagram for developers and a user's manual for the end user of the product. Students are also tasked with crafting unit tests for this program, with some unit tests given.

While it is not generally considered to be the hardest assignment in the course, it can cause significant difficulties for students who are not comfortable with object oriented programming in Python. This assignment has a notable increase in difficulty from the previous assignment.

Assignment 5.0: Fractal Visualizer - Refactoring

Students are given a poorly crafted–but working–program that creates images for various fractals. They are tasked with cleaning up the code and maintaining the same functionality of the program. Additionally, students are required to produce a UML class diagram documentation for developers and a user's manual for end users.

This refactoring assignment is rated as one of the hardest assignments in the course, and the hardest of the three in the performance analysis.

2.5.3 Data Analysis

To analyze performance data, the adjusted scores were transformed to be values in the range of $[0.0, 1.0]$ as a percentage of points received on a given submission.

To analyze submission start times, first commits that were made within the assignment period were normalized on a scale of $[1.0, 0.0]$, with 1.0 being the time the submission was released and 0.0 being the moment the assignment was due. Submissions that were started after the assignment due date were removed from this data, which was less than 3% of submissions studied. All submissions analyzed were over a roughly two week period with slight variation due to holidays and the semester schedule. On the scale, 0.4 was roughly the start of the week an assignment was due, which correlates with the grace points deadline.

2.6 Results

2.6.1 Performance Analysis

All Assignments

For all assignments, we have $n = 133$ for submissions receiving grace points, and $n = 179$ for submissions not receiving grace points. The average score of assignments receiving grace points is 86.0%, and the average score for those not receiving grace points is 78.2%. For submissions receiving grace points, 63.2% scored a 90% or higher, with 38.3% scoring 100%. For submissions not receiving grace points, 43.6% scored a 90% or higher,

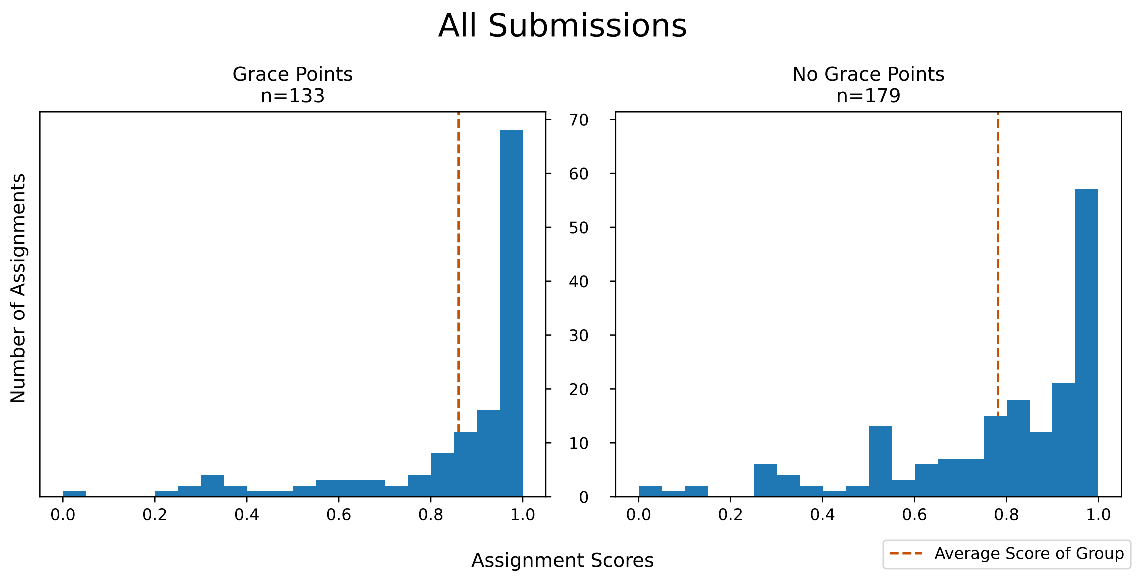

Fig. 2.1: Score distributions for all submissions, grouped by submissions that did or did not receive grace points. $n = 133$ for submissions receiving grace points, $n = 179$ for submissions that did not receive grace points. The mean score of submissions receiving grace points was 86.0%, and for submissions not receiving grace points, 78.2%.

with 19.0% scoring 100%. The distribution of the two groups **differed significantly**, (Mann-Whitney U=8785.5, $n1 = 133$, $n2 = 179$, $p = 0.0001$ two-tailed).

Assignment 3: Big Data Processing

For Assignment 3, we have $n = 47$ for submissions receiving grace points, and $n = 63$ for submissions not receiving grace points. We find that the average score of assignments receiving grace points is 84.7%, and the average score for those not receiving grace points is 82.5%. For submissions receiving grace points, 64.8% scored a 90% or higher, with 44.7% scoring 100%. For submissions not receiving grace points, 52.4% scored a 90% or higher, with 23.8% scoring 100%. The distribution of the two groups did not differ significantly, (Mann-Whitney U=1200, $n1 = 47$, $n2 = 63$, $p = 0.0849$ two-tailed).

Assignment 4: Bingo! Card Generator

For Assignment 4, we have $n = 42$ for submissions receiving grace points, and $n = 60$ for submissions not receiving grace points. We find that the average score of assignments

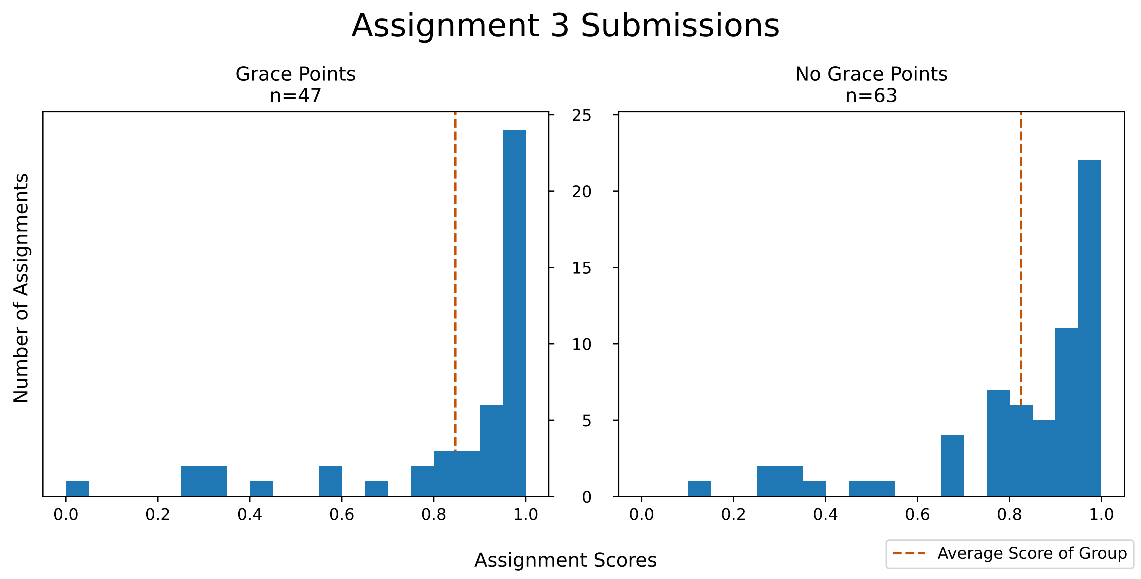

Fig. 2.2: Score distributions for Assignment 3, grouped by submissions that did or did not receive grace points. $n = 47$ for submissions receiving grace points, $n = 63$ for submissions that did not receive grace points. The mean score of submissions receiving grace points was 84.7%, and for submissions not receiving grace points, 82.5%.

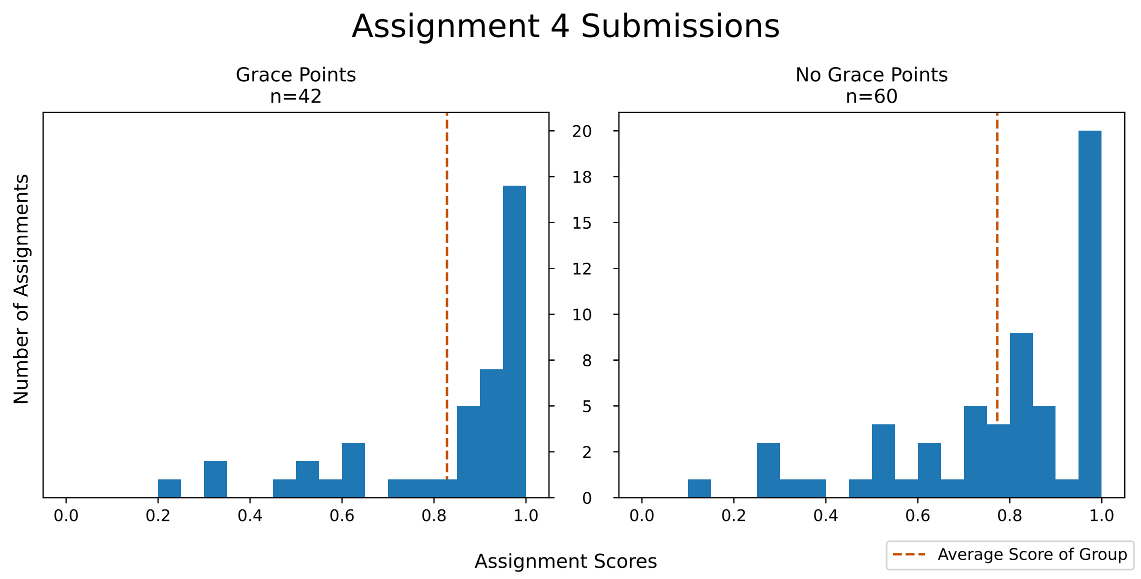

Fig. 2.3: Score distributions for Assignment 4, grouped by submissions that did or did not receive grace points. $n = 42$ for submissions receiving grace points, $n = 60$ for submissions that did not receive grace points. The mean score of submissions receiving grace points was 82.8%, and for submissions not receiving grace points, 77.3%.

receiving grace points is 82.8%, and the average score for those not receiving grace points is 77.3%. For submissions receiving grace points, 57.1% scored a 90% or higher, with 28.6% scoring 100%. For submissions not receiving grace points, 35.0% scored a 90% or higher, with 15.0% scoring 100%. The distribution of the two groups did not differ significantly, (Mann-Whitney U=1013, $n1 = 42$, $n2 = 60$, $p = 0.0922$ two-tailed).

Assignment 5.0: Fractal Visualizer - Refactoring

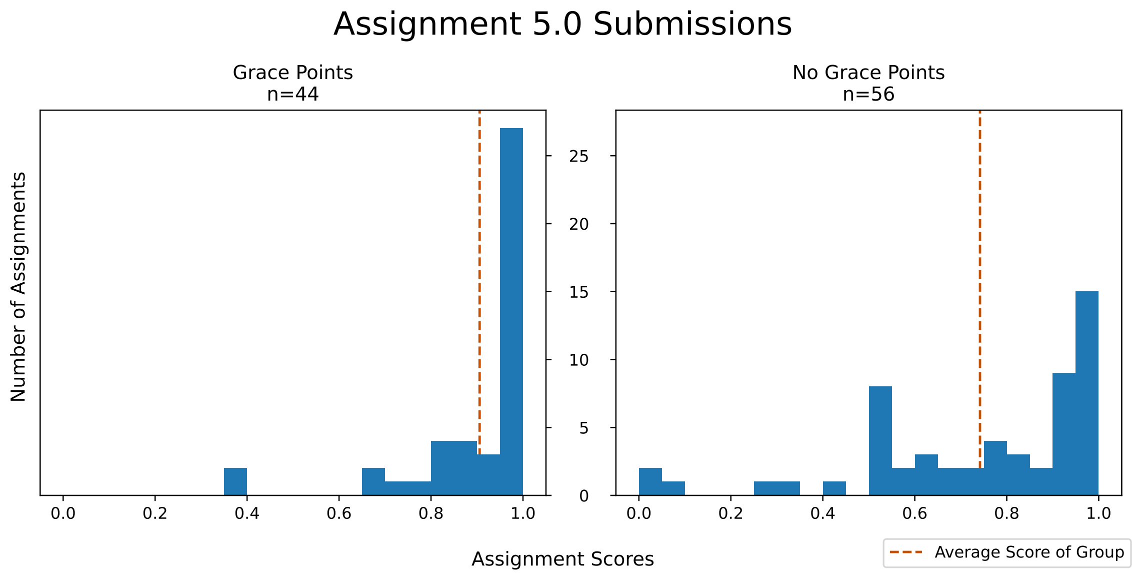

Fig. 2.4: Score distributions for Assignment 5.0, grouped by submissions that did or did not receive grace points. $n = 44$ for submissions receiving grace points, $n = 56$ for submissions that did not receive grace points. The mean score of submissions receiving grace points was 90.6%, and for submissions not receiving grace points, 74.2%.

For Assignment 5, we have $n = 44$ for submissions receiving grace points, and $n = 56$ for submissions not receiving grace points. We find that the average score of assignments receiving grace points is 90.6%, and the average score for those not receiving grace points is 74.2%. For submissions receiving grace points, 68.2% scored a 90% or higher, with 40.9% scoring 100%. For submissions not receiving grace points, 42.9% scored a 90% or higher, with 17.9% scoring 100%. The distribution of the two groups **differed significantly**, (Mann-Whitney U=714.5, $n1 = 44$, $n2 = 56$, $p = 0.0003$ two-tailed).

2.6.2 Time Analysis

Comparing Fall 2022 To Previous Semesters

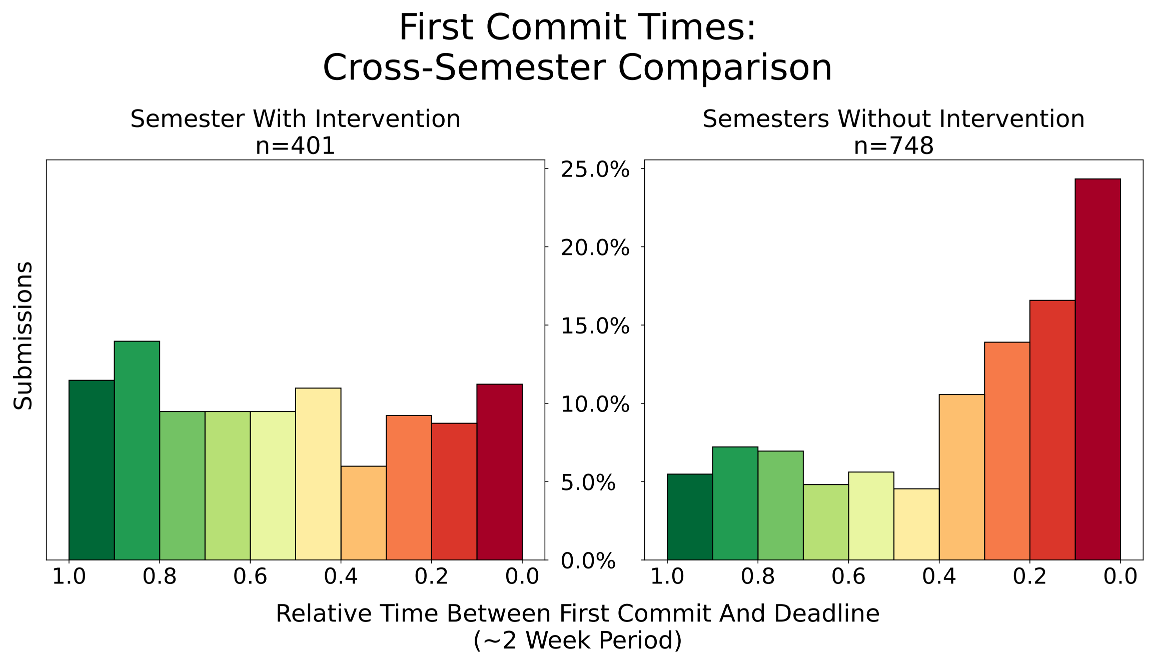

Fig. 2.5: The comparison of first commit times between Fall 2022–the semester that the intervention was introduced–and previous semesters on the same assignments. Submissions from Fall 2022 and Spring 2022/Fall 2021 are compared. $n = 401$ for submissions with the intervention, $n = 748$ for submissions before the intervention. On the relative scale, 1.0 is when the assignment was released, 0.0 is the moment the assignment was due.

For the students in Fall 2022 on the submissions eligible for grace points, the mean was 0.523, with 64.8% of students starting before 0.4 on the relative time-scale. For the submission behaviors of students on the same set of assignments for Spring 2022 and Fall 2021–before the grace points were introduced–the mean is 0.355 with 34.6% starting before 0.4 on the relative scale. The distributions of the two groups **differed significantly** (Mann-Whitney U=197130, $n1 = 401$, $n2 = 748$, $p = 1.43\text{e-}18$ two-tailed).

Comparing Assignments Before And After Intervention Within Fall 2022

We see that for the submissions on assignments that had the grace point intervention

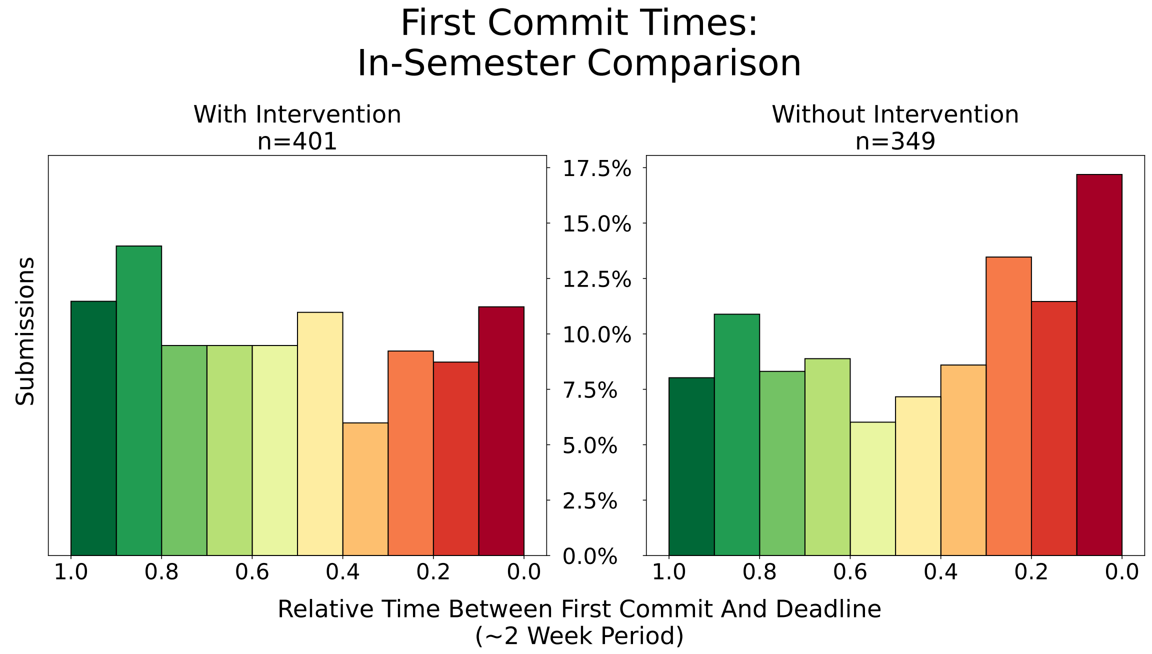

Fig. 2.6: The comparison of first commit times in Fall 2022, the semester that the intervention was introduced, before and after the introduction of the intervention. $n = 401$ for submissions with the intervention, $n = 349$ for submissions before the intervention. On the relative scale, 1.0 is when the assignment was released, 0.0 is the moment the assignment was due.

possible, the mean was 0.523, with 64.8% of students starting before 0.4 on the relative time-scale. We also see that for the submission behaviors of students on the assignments in the semester before the grace points were introduced, the mean is 0.441 with 49.3% starting before 0.4 on the relative scale. The distributions of the two groups **differed significantly** (Mann-Whitney U=80527, $n1 = 401$, $n2 = 349$, $p = 0.0004$ two-tailed).

It is worth noting that some of this change *could* be explained by students at the beginning of the course not having much familiarity with Git. However, there was an assignment before the first assignment studied which allowed students to practice with Git and encouraged good Git committing behaviors, reducing this effect. We do not believe the change would be so drastic if the effect was only due to student's bettering their Git habits. When paired with cross-semester data, our belief of this strengthens, and we are seeing the effects of the grace point intervention on student behaviors.

2.7 Conclusions

From the data, it is clear that there is a correlation between student assignment performance and early completion of the project analysis milestone. Data for all assignments aggregated together showed statistical significance, while one of the assignments had clear statistical significance in it's data alone. This data suggests that if students put in the work to understand the assignment given to them early, they are placed in a position for success on assignments. Additionally, it suggests that the effect increases as project difficulty increases. It is clear that this is the case in the Computer Science course studied. It is suspected that these results generalize beyond this course, and beyond the scope of the field of computer science; especially so for project based assignments within other fields if one is able to design an analog to the "requirements analysis" planning phase used in the CS1440 course.

Additionally, it was found that the introduction of the grace points intervention caused students to start working on their assignments earlier. With both of these hypotheses being supported by our study, it is found that interventions which aim to encourage students to analyze projects or assignments earlier can be done and provides a significant benefit to student project performance.

This study can inform education researchers in intervention design, as there is clear assignment performance improvements shown by students understanding the expectations and requirements of an assignment early on in the assignment's release period. It is clearly valuable for project-based work in the Computer Science course studied, and many courses within other disciplines share numerous traits with.

CHAPTER 3

Incremental Development and CS1 Student Outcomes and Behaviors

3.1 Introduction

Computer science educators frequently seek ways to encourage students to make continual forward progress on their assignments to increase success rates on an assignment and reduce student burden. One suggested way is to use incremental development. This is where students are encouraged to write small chunks of code and then test their new additions—making continual forward progress—as opposed to large additions of code that require lots of debugging time. Previous research has been done in developing a tool written in Python to measure this incremental development [10]. This tool—which analyzes and compares the source code of students' projects each time the project is run—seeks to analyze if students are making continual forward progression between runs or are stuck debugging the same blocks of code. The tool assigns projects a Measure of Incremental Development (MID) score, which can be used to classify if a submission is incrementally developed or not.

This tool was developed with the notion that incremental development is good for computer science students. At this time, there have been few measurements made to indicate if incremental project development positively correlates with student project outcomes, using a definition of incremental development as "working early and often, incrementally checking work via either interactive program launches or software test execution, and incrementally writing software tests," which differs slightly from the computable incremental development metric defined in Shah et al. which our paper studies [10, 11]. Additionally, the assignment creation behaviors of students–such as time spent on the assignment, how many coding sessions are required to complete the assignment, and how many times the code is tested–have not been studied to understand if there is correlation with incremental development, using the definition of incremental development provided by the MID score metric. When submissions are marked as “incrementally developed” by the MID score metric, are there different behavioral trends observed when compared to students who do not develop their projects incrementally?

Our study seeks to answer two key research questions related to this MID score metric using previously collected keystroke data from students in a CS1 course at Utah State University (USU).

RQ1: Does incremental development correlate with better assignment performance?

RQ2: What trends are found in student behaviors when they perform incremental development as measured by the MID scores?

With answers to these questions, more informed interventions may be designed and deployed to assist CS1 students in their success. By better understanding the various behaviors that correlate with incremental development, we will be able to analyze if various assumptions about the benefits of incremental development are true. This information can inform future development of the MID score tool in order to ensure that the MID score correlates with submission and performance behaviors desired by the authors, and potentially identify if any problematic behaviors are incentivized by the tool.

3.2 Background And MID Score

The majority of our work relates to the research paper *Understanding and Measuring Incremental Development in CS1* [10]. This paper defines a computable metric to measure incremental development. The paper defines this metric to classify projects as being incrementally developed.

The authors of the paper created a package published to pypi, titled measure-incremental-development [12]. This package is used to calculate the proposed incremental development score, referred to as the MID score metric. This package is fairly new, and has little public use beyond the scope of the initial research. The way this package computes MID scores is defined formally in Shah et al. and briefly detailed in the remainder of this section [10].

MID scores are calculated by categorizing code changes, observing struggles, and penalizing time spent making repeated debugging attempts after large changes. These code changes are observed through various snapshots of student's code throughout their development. Meaningful code changes are classified into three different categories: *forward progress steps*, *adjustment steps*, or *test steps*. Forward progress is adding new lines of code. Adjustment is when a student only makes changes to existing lines of code. A test is where a student adds print statements or otherwise does work to verify outcomes without significantly affecting the program. For the most part, test steps are ignored by the MID score.

The forward progress and adjustment step categories are used in a computation of struggle for a given code change. For each forward progress step s, the number of later adjustment steps affecting that forward progress step, a_s, affects the measure of struggle for that given forward progress step.

$$\text{struggle}(s) = \frac{(a_s)^2}{a_s + 1}$$

For a given forward progress step s with no associated adjustment steps, $a_s = 0$, the struggle calculation produces 0; no struggle was detected in the project to accomplish that forward progress step. If a forward progress step has a single adjustment step, $a_s = 1$, and we have a struggle computation of 0.5. Two adjustment steps ($a_s = 2$) produces a struggle computation of $1.\overline{3}$, and three adjustment steps ($a_s = 3$) produces a struggle computation of 2.25. As more adjustment steps are needed for a given forward progress step, the computed struggle metric increases exponentially.

To penalize for large additions of code that are untested and require lots of adjustment, the MID score factors in the size of a given forward progress step in its computation, defined by the number of tokens in the step. Each forward progress step s is assigned a size metric, proportional to the size of the whole project. Assuming `numTokens`(s) returns the number of tokens for a given forward progress step s, the size is computed as:

$$\text{size}(s) = \frac{\text{numTokens}(s)}{\sum_{k=1}^{n} \text{numTokens}(k)}$$

The MID score summarizes the entire development process of a student's project, collected in a sequence of snapshots, with each snapshot collected every time a student's code is compiled and executed. For n various forward progress steps in a given project, the MID score computation is defined as:

$$\text{MID} = \sum_{s=1}^{n} \text{size}(s) \cdot \text{struggle}(s)$$

The range of MID scores is $[0, \infty)$. If a project consists of no adjustment steps, the returned MID score will be 0. As more adjustment is needed in a project, and larger untested chunks of code are added at a time, the MID score increases. According to Shah et al., classifying projects with MID scores in the range of $[0, 2.5]$ as incremental and projects with MID scores greater than 2.5 as non-incremental. This classification cutoff resulted in 80% to 85% agreement with human evaluators, depending on the project studied. Many of the misclassifications were noted as being "close-calls," with reviewers also struggling in their classification of projects [10].

An important difference between this definition of incremental development and the definition of incremental development used in a prior student outcome analysis– which defined incremental development as "working early and often, incrementally checking work via either interactive program launches or software test execution, and incrementally writing software tests" [11]– is that this metric directly analyzes code *changes* instead of external behaviors to categorize incremental development.

3.3 Related Work

To study student development habits in an effort to see if students achieve continued forward progress and iterative improvements, others have analyzed student code version control repositories [13, 14]. However, only having student code repositories can lead to known issues with project analysis. Namely, student commit behaviors are different, with some committing to version control often and with every test, and others rarely committing changes to version control. Analysis of version control does not always fully illustrate the development and testing of student code [15]. Due to the fine grained data desired within the scope of the small problems CS1 students encounter, we make the choice to forego code repository analysis and instead seek out keystroke-level data, discussed later.

Prior work has been done to gather keystroke-level data and study student development behaviors [16]. The study by Kazerouni studies student procrastination and incremental development [11]. However, their study primarily focuses on incremental development and procrastination; not a direct study of incremental development with other metrics such as student performance or interaction events with an assignment. They utilize different measures of incremental development. Kazerouni et al. does note some ancillary findings from one of their metrics for incremental development—the mean time of test writing compared to the project deadline, which is part of an indicator for incremental development—correlates positively with time spent on an assignment. Earlier test writing leads to longer time spent on an assignment.

Incremental development in commercial software development has been adopted by many companies, and has become one of the most common deployment strategies of modern software. However, there has been an observed phenomenon called Incremental Development Productivity Decline that indicates incremental development may not be the silver bullet some desire it to be [17, 18]. Our research may determine if a similar phenomenon is seen in early learners of computer science, as we can study time spent on an assignment compared to educational outcomes as our measure of productivity within the framework of incremental development.

Within the task of information seeking and retrieval, the researchers McGregor et al. [19] define various measures of cost, effort, and load. Some of these measures that influence their determination of a tasks cost, effort, and load include system interaction events and time spent completing the task at hand. Reducing cost, effort, and load is beneficial to learners. Despite the fact that this study was done within the scope of information seeking and retrieval, we suspect that these measures of cost, effort, and load are analogous and generalize to computing education and programming tasks. We can study these metrics to determine if incremental development is suspected to modify the cost, effort, and load learners experience during a task.

3.4 Methods

3.4.1 Dataset

This analysis uses the keystroke data collected from CS1 students at Utah State University [20]. This data contains keystroke information, record of execution runs, student information, and due dates for the assignments. All data was de-identified and made publicly available with oversight from their ethics review board. It includes a collection of 41 students and 8 assignments, with all assignments being done in Python. Of those assignments and students, there are 209 assignment records that were able to be reconstructed correctly for the MID analysis. Each of these assignment records were used to determine run states that could then be fed to the MID score tool.

The data was used to understand the MID score relating to: the assignment grade, the student's final course grade, number of runs, number of coding sessions, assignment interaction events, the average time between events in a coding session, average time between runs in a session, and the percentage of interactions made before testing code in a coding session.

Overall, these analyses helped inform whether or not there is a correlation between student grades and coding behaviors with a measure of incremental project development.

3.4.2 Submission Reconstructions

Within the keystrokes dataset, we reconstructed student assignment submissions using the sequences of file edit events. Information of the cursor location and file name in the project was known and a simple algorithm to add and remove text from a file as it is reconstructed was employed. This was based on a sample algorithm that was provided inside the dataset [20]. We extended this algorithm to take “snapshots” of the project state any time a student executed a run event so we can see the executable and tested states of a student’s project over time. These snapshots were provided as the code state snapshots for the MID score calculation.

In the data set, not all students had provided data for all assignments. Additionally, some data corruption in the keystroke data set prevented some submissions from being reconstructed correctly, and these submissions were removed from our analyses.

3.4.3 Dataset Filtering

As noted earlier, some submissions could not be reconstructed correctly due to dataset corruption. Other submissions contained edits mislabeled as the wrong file. Because of these issues, projects were checked after reconstruction to ensure they reconstructed correctly before being used in our analysis. Additionally, any files that were automatically generated by the student’s IDE were removed, as well as any files that were not intended to be source code files (such as `.txt` files or files with names similar to `plan` the course’s convention for documentation files).

To check if a reconstruction was successful, the final reconstructions of source code files were executed with Python. If the source code executed successfully, it was marked as being a successful reconstruction. If the source code had an error, it was inspected further. Of the files that were deemed to be bad reconstructions, all had syntax errors. However, not all submissions with syntax errors were due to bad reconstructions. Students who did not complete assignments or made last minute changes that were unchecked before submitting had syntax errors in their project. These reconstructions remained in our analysis. If any files in a given project were reconstructed incorrectly, that submission was entirely removed from our analysis.

MID Score Package Bug

The MID score package is fairly new and documentation is limited, resulting in the discovery of bugs within the package. When passing in the various project submission states, newline characters were added to the end of each state to resolve a bug. This bug was found as some reconstructions that did not end with newlines and had only one character change would crash the package's MID score computation function. It was verified that this addition of a new line character did not affect the actual MID score computation, both through experimentation and analysis of the source code. This slight modification to the run states allowed all submissions with successful project reconstructions to be processed and have a MID score assigned to it.

3.4.4 Metrics

Various metrics were defined in order to measure student behavior on assignments within the keystroke data set. Many of these metrics defined were informed by previous work on this same keystroke dataset [21].

MID Score

The definition of a MID score is briefly discussed in the section 3.2. It is defined in Shah et al. [10] and is provided by the measure-incremental-development package [12].

Input to the measure-incremental-development software is a series of project snapshots at each code execution event. Using the keystroke dataset, we reconstructed these code snapshots at each code execution event. Each file in the student's project was concatenated together to represent the whole project at the given state before being given to the MID score metric. Thus, the MID scores represent a rating of the whole project rather than individual files or tasks in the project.

Final Score

The final score is the final class score including assignments, grace points, and exams. Students were given a small amount of "grace" for missing points on assignments, and this was applied uniformly and to all students over the semester. The final score also includes assignments from earlier in the semester than the keystroke data set has information for.

Assignment Interaction Events

Assignment interaction events are defined as IDE actions performed by a student during development. Some of these actions include edit keystrokes, executing the assignment, and switching between tabs of source code files. The keystroke data was filtered to remove any events that are not directly interactive, such as a separate event representing the insert of a character resulting from an already logged keystroke and any auxiliary events produced automatically by the student's IDE. Additionally, events where students close the IDE were also removed from this measure.

Coding Sessions

A coding session is defined as continuous working time on an assignment. A new coding session begins after a 5 minute period of no activity on the assignment. The decision of a 5 minutes of no activity being the threshold for a coding session was informed by a study measuring the time on task while programming, which found that 5 minutes of no activity indicates a greater than 50% chance that the student has become disengaged with the task [22]. Activity is deemed as any interaction event, such as a test run or edit.

Length of time on Assignment

The length of time the student spends on an assignment is the sum of the time spent during all coding sessions as suggested in [22]. Consideration for the amount of time to complete an assignment and incremental development was a hypothesis from Shah et al [10]. This metric was especially important in studying this hypothesis.

Time Between Interactions

The time between interaction events was measured for all interaction events in the same coding session. The time between interaction events that end a coding session and start a new one were not measured, as that effectively would be a measure of break time with our definition of a coding session.

Runs Of Code

While the data set has run start events, and run end events, our study only utilized the run start events. A run is counted as any attempt to execute their code, even if the Python interpreter finds a syntax error and does not execute their code. The current state of the student's project was saved every time a run start event happens.

Time Between Runs Of Code

Much like the time between interactions metric, the time between runs of code was measured for all runs in the same coding session. Two runs in different coding sessions did not have their time between them measured, as that becomes significantly effected by the time of students breaks between coding sessions.

Percentage Of Interactions Before Running Code In A Session

When a new coding session starts, all interactions that happen *before* an execution event happens are tracked and compared to the total number of interactions on an assignment. If a student does not execute their code during that coding session, all interactions in that session are marked as interactions happening before a run. A low percentage of interactions before running code indicates that students would start testing their code early on in their coding sessions. A high percentage of interactions before running code in a session indicates a student was likely not spending that coding session focused on debugging tasks.

3.4.5 Analysis Of Metrics

To analyze the defined metrics, we found common statistical information such as the mean, median, and standard deviation, separated by a submission's "incremental" or "non-incremental" status using the MID score. To see if any differences in the mean, median, and standard deviation found are statistically significant, Mann–Whitney U tests are applied.

To gain understanding into incremental development, multiple analyses were completed. The analyses compared MID to the following: final score, number of runs from assignment start to submission, and number of keystrokes from assignment start to submission. Other analyses included, the average time between runs in coding sessions and the average time between interaction events.

Our work can be seen on GitHub.

3.5 Results And Discussion

Table 3.1: Reported results, grouped by measure, for the metrics studied. Details and Mann-Whitney U results are reported in the relevant subsection. In the *Group* column, **I** represents the group of assignments marked as incrementally developed and **N** stands for the group of assignments marked as non-incrementally developed.

Measure	Group	Median	Mean	STD	p
Assn. Score (3.5.1)	**I**	93.125	90.24	10.95	.97
	N	91.05	91.18	6.88	
# Interaction Events (3.5.2)	**I**	3861.0	4289.3	3452.0	$1.4e^{-5}$
	N	6573.5	7228.3	4487.2	
Event Time Δ (3.5.3)	**I**	6.15	11.16	11.48	.14
	N	4.52	7.26	7.26	
Avg. Assn. Time (3.5.4)	**I**	2.25	2.52	1.59	$4.6e^{-6}$
	N	3.94	4.02	2.00	
# of Coding Sessions (3.5.5)	**I**	12.0	13.92	8.88	.13
	N	13.5	16.57	10.50	
# of Assn. Runs (3.5.6)	**I**	59.0	75.76	72.0	$1.1e^{-6}$
	N	113.5	147.83	111.61	
Time Between Runs (3.5.7)	**I**	106.25	123.15	91.09	.08
	N	86.41	99.14	47.74	
Event % Before Run (3.5.8)	**I**	0.42	0.47	0.29	.03
	N	0.31	0.37	0.22	

3.5.1 Assignment Score

This difference in assignment scores between incrementally and non-incrementally developed projects was not statistically significant (Mann-Whitney U=4116, $n1 = 154$, $n2 = 53$, $p = 0.926$, two tailed). This can probably be explained by the fact that most assignments have scores near 100. A dataset with more discriminating scores may yield more interactions between assignment score and MID score.

3.5.2 Number of Assignment Interaction Events

The difference in the number of assignment interaction events between incrementally and non-incrementally developed projects was statistically significant (Mann-Whitney U=2520, $n1 = 155$, $n2 = 54$, $p = 1.37\text{e-}05$, two tailed). Incrementally developed projects required fewer keystrokes to complete.

There are two possible explanations for this. The first, that incremental development results in shorter development times, which is consistent with conventional wisdom among computer science educators. However, the effect may not be causitive—better students may simply be more likely to use incremental development practices. A within-subjects study in which incremental development behavior is modified over time could help determine causation.

3.5.3 Average Time (seconds) Between In-Session Interaction Events

This difference in the average time between in-session interactions between incremental and non-incrementally developed projects was not statistically significant (Mann-Whitney U=4757, $n1 = 155$, $n2 = 54$, $p = 0.135$, two tailed). In other words, we have no evidence that the frequency with which a student interacts with their code is correlated with incremental development, indicating there is no found correlation between in-session pauses and incremental development.

3.5.4 Assignment Time (hours)

This difference in time spent on assignments between incrementally and non-incrementally developed projects was statistically significant (Mann-Whitney U=2430, $n1 = 155$, $n2 = 54$, $p = 4.555\text{e-}06$, two tailed). This is a very similar finding to the number of assignment interaction events measure, discussed in section 3.5.2, especially when one pairs it with the findings of no statistically significant difference in time between in-session interaction events 3.5.3.

3.5.5 Number of Coding Sessions

The difference in the number of coding sessions between incrementally and non-incrementally developed projects was not statistically significant (Mann-Whitney U=3604, $n1 = 155$, $n2 = 54$, $p = 0.129$, two tailed). Despite the fact that incremental developers are using fewer events to complete their assignments, there is no evidence that they are doing so in fewer coding sessions.

3.5.6 Number of Runs

This difference in the number of runs between incrementally and non-incrementally developed projects was statistically significant (Mann-Whitney U=2322, $n1 = 155$, $n2 = 54$, $p = 1.14\text{e-}06$, two tailed). We observe that students that developed incrementally had on average **half** the number of runs. At first glance, it may be surprising that incremental developers are using fewer run events than non-incremental devs. However, it makes sense when we consider that incremental developers are using fewer total interaction events and incremental development is hypothesized to require less time spent debugging.

3.5.7 Difference in Average Time Between Runs

The difference in time between runs on a submission between incrementally and non-incrementally developed projects was not statistically significant (Mann-Whitney U=4500, $n1 = 144$, $n2 = 54$, $p = 0.08$, two tailed). We hypothesized that there would be a statistically significant difference. After all, colloquially we say incremental developers run their code "more often." However, our hypothesis was that incremental developers would have the shorter time-frame between runs and our data suggests the opposite. We suggest one or both of the following explanations.

First, incremental developers are fitting in less interactions in the time between run-events. This seems unlikely, considering the fact that we are not able to show any difference in temporal density of interactions (Section 3.5.3). The other explanation is that the difference is actually present despite the p-value exceeding our threshold of 0.05, and that the shorter time span between runs seen by non-incremental developers is because they are spending more time in bug fixing stages, running their code consecutively.

3.5.8 Percentage of Events Before A Run In Coding Session

The percentage of events before a run in a coding session between incrementally and non-incrementally developed projects was statistically significant (Mann-Whitney U=5030, $n1 = 155$, $n2 = 54$, $p = 0.027$, two tailed). This may indicate that coding sessions are filled with more forward progress and less debugging.

3.5.9 Incremental Development and Final Grade

The percentage of assignments completed that were marked as using incremental development according to the MID score tool was computed for students with more than three reconstructed assignment submissions. There was no statistically detectable correlation between the final score and the percentage of incremental development use as seen in Figure 3.1. This, together with the lack of finding that assignment score is affected by differences in incremental development (Section 3.5.1), indicates that academic outcomes aren't necessarily affected by incremental development. If this is true, then educators should not make statements such as "incremental developers will get better grades" within the context of CS1 courses. A better claim would be, "incremental developers are more productive and take less time on their assignments."

3.5.10 Threats To Validity

The data set used contains only data from a small set of students enrolled in a CS1 course from Utah State University. This data set is limited in scope and the behaviors of these students are not guaranteed to generalize towards students with different backgrounds than those at Utah State University. In addition, the dataset does not include information on whether a student had prior programming experience or not, which could have an effect on their level of incremental development.

The MID score metric has a few limitations noted by Shah et al. [10]. An additional possible shortcoming uncovered by our work is how instructor-provided starter code is handled. It is unclear how this should influence the calculation of the MID score and its classification. The effects of starter code are likely compounded when starter code gets

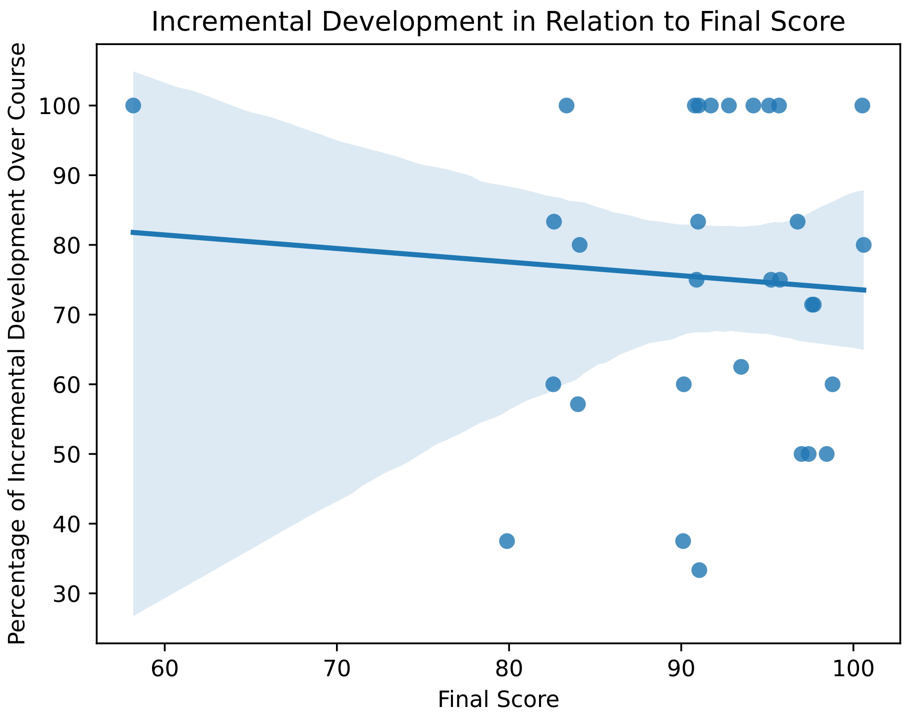

Fig. 3.1: The figure shows the percentage of a student's incremental development over the course in relation to the student's final grade. The percentage is the number of assignments developed incrementally compared to the number of assignments a student completed over the course. There is no discernible correlation.

added into the project part way through the project's life-cycle as a large single addition, with some of this starter code intentionally being syntactically incorrect. Some starter code expects students to edit existing lines of code, which the utilized MID score metric would penalize as being an adjustment instead of forward progress. We recommend that future versions of this tool include a way to provide starter code that a student may utilize such that the MID score can be calculated only from the student's changes.

3.6 Conclusions

For **RQ1** we failed to find a measurable difference between scores for students who developed using incremental development versus those who did not. Assignment scores of projects marked as being incrementally developed did not have any statistically detectable differences from the scores of assignments that were not marked as being incrementally developed. Additionally, the percentage of a student's projects that were incrementally developed had no notable correlation with a student's final score in the course.

For **RQ2**, this research has provided insight into how incremental development is of use to students. There were strong measurable relationships between incremental development and many of the metrics studied.

The number of interaction events generally correlates to more work being required to complete a project. We can see that incremental development correlates with less interaction events, and thus implies that incremental development correlates with less work required to complete a project as well.

The amount of time spent on an assignment has negative correlation with incremental development. This also supports the earlier notion that incremental development of projects correlates with less work required to complete a project. This shows that the industry phenomenon of incremental development productivity decline does not apply to early CS students, as students are performing work faster and receiving the same results when developing incrementally [17, 18]. With both the number of interaction events and time spent on an assignment being reduced when a learner uses incremental development strategies, and with no change in outcomes, we speculate that learning requires less cost, effort, and load [19].

The number of program runs negatively correlates with incremental development. This may suggest that projects which are marked as being incrementally developed require less bug-fixing and adjustment,, which was something the authors of the MID score metric desired to codify with the metric.

Lastly, the percentage of events in a coding session that happen before a run event negatively correlates with classification of a project as being incrementally developed. This suggests that projects which are classified as incrementally developed require less time spent in bug-fixing and adjustment during coding sessions.

Our findings are consistent with the notion that instructors should be encouraging students to use incremental development. We have found evidence to indicate that students will spend less time on their assignments and require less testing of their code to achieve the same results. We find no evidence that recommending to students that they incrementally develop their projects changes their assignment creation behavior, nor that interventions should not be designed to instill incremental development practices in early CS students. Indeed, should future within-subjects studies show causality, these practices would be shown to be highly beneficial.

3.6.1 Classroom Implications And Future Work

Some computing educators may wish to find ways to encourage incremental development in the classroom. According to Ditton et al. [23], students who view visual "playbacks" of their development have positive code review experiences and desire to make behavioral changes when developing software again. The findings of Kazerouni et al. [11] show that with keystroke level data, it is possible to produce visualization summaries that show development progress over an assignment period as well as various measures of incremental development on the project. They also theorize that these visualizations could be used to both reduce student procrastination and encourage better incremental development practices. Both live code playback and of the various metrics determined over the lifetime of a project could be used to help students be more aware of their incremental development practices.

Future work includes studying whether incremental development causes improvements in time taken on assignments, correlation between perceived difficulty of assignments and incremental development practices, direct measures of cognitive load when students develop incrementally versus those that do not, differences in types of errors encountered by students who develop incrementally, and if prior programming experiences correlates with incremental development in early CS courses.

CHAPTER 4

The Shell Tutor: An Intelligent Tutoring System For The UNIX Command Shell And Git

4.1 Introduction

Despite the overwhelming abundance of graphical user interfaces (GUIs) in modern computing, the command shell[1] is still frequently used in modern development. According to the 2022 Stack Overflow developer survey, 29% of developers noted using Bash / Shell scripting frequently during their development. In addition, 84% of developers who use version control systems (VCS) note that they utilize the command shell in some way to interact with their VCS [24]. The command shell clearly still has modern use among software developers.

Git is another tool frequently utilized by modern developers. Git is a VCS that allows developers to manage their code, track changes to the code base, and facilitate team development [25]. Version control is considered to be a necessary tool for many modern developers with high-adoption rates. According to the 2022 Stack Overflow developer survey, 96% of developers utilize some form of version control, and 94% of developers utilize Git [24].

Because of this, many computing and software engineering educators teach both the command shell and Git VCS in their courses, oftentimes early on in the curriculum. Teaching of these tools can be challenging to do in a classroom setting, given the hands-on nature of these tools. Computer science faculty at our institution have noted this difficulty.

We became acutely aware of these struggles in our Fundamentals of Software Engineering course: a required course for all computer science students taught immediately after CS1. Starting the Summer of 2020, we devised and developed an intelligent constraint-based tutoring system designed to assist in teaching students the UNIX-like command shell

[1]For this paper, we use the term "command shell" to refer to the usage of the command line to control a system shell.

and Git VCS [26]. This tool was first deployed in the Fall 2020 semester and was initially very successful. We continued development of the Shell Tutor over the years, and have noted success in deploying the tool in the course. We have even provided the Shell Tutor to students in other courses we have taught for select students in need. Many students at our institution have noted sharing the tool with friends outside of the program to assist them in their learning of the command shell and Git, and other instructors have informally shared the tool with the students of theirs who do not know Git and the command shell. As of Spring 2024, this tool has been used by over 1100 students.

This paper aims to provide a review and reflection of this tool. Information is shared regarding the tool, its technical compatibility and support, development features, features available to lesson authors, and its logging features. Additionally, a study was conducted on current students of the course and students who took the course previously after the tools deployment. This study assists in gaining a better understanding of the perceived effects that the Shell Tutor has on student learning, difficulty of the tool, student opinions on the logging feature of the tool, and additional context around the tool, such as identifying what issues students may have with the tool. We include questions focused on student opinions about shell logging as it may be a controversial feature, but has the potential to enable this tool to be used by future CS education researchers.

This survey focuses on the following questions. Do students believe the tool to assist in their learning of the command shell and Git? What is the perceived difficulty of using the Shell Tutor compared to the tools the Shell Tutor aims to teach? How do student feelings on the Shell Tutor compare to their feelings on the command shell and Git? What negative perceptions exist about the tool? How do students feel about the existence of a shell logging feature?

This study was conducted to gain a better understanding of the effects of this tool on student's education and enable us to provide a reflection of the Shell Tutor. This reflection will also assist in the continued development of the tool and guide researchers in future studies of the tool.

4.2 Background

4.2.1 Related Work

When it comes to teaching students the command shell and Linux, there are a few similar tools that have been created and detailed in publications. The first being a system called *ShellOnYou* which allows students to attempt exercises in the command shell and gain feedback [27]. Another tool exists called *TermAdventure*, which aims to emulate text based adventure games in a command line environment with the goal of teaching the command shell [28]. A tool that is more similar to the Shell Tutor is called *uAssign* which creates a web-accessible virtual environment in which students can access and learn the command shell [29]. The *uAssign* tool was developed in conjunction with *TuxLab*, which serves a similar goal [30]. Another similar system that has stopped being supported is called *LinuxGym*, which creates an actual virtual environment (and associated exercises) that give students access to a Linux system and command shell [31]. The Shell Tutor shares many similarities with these tools, but is notably different in that it is *entirely* powered by the student's command shell on their computer. The Shell Tutor does not provide a simulated environment for students to learn in; students will be learning–and modifying–on the same system that they use every day. Beyond the tool's educational purposes, it is well suited for system setup and configuration tasks.

Very few studies have been done to investigate student behaviors while using the command shell. Those that have been done have only gathered limited information in specialized contexts. To our knowledge, none of the teaching systems noted above have a similar level of logging capabilities to the Shell Tutor system. One of the closest works we have found details using a student's shell `.history` file (a simple file detailing only the commands ran) while conducting hands-on exercises [32]. A similar study was conducted using students' shell `.history` files to study their behaviors on a programming exam [33]. Lastly, we've identified a paper which creates a logging system to analyze student behaviors while learning some cyber-security tools [34].

4.2.2 Teaching Command Shell And Git In Class: Then Vs. Now

Before the introduction of the Shell Tutor in our course, the command shell and Git were taught through in-person lectures with accompanying notes. To ensure students knew basics of the command shell and Git, the first assignment in the course required students to submit a sample Git repository, demonstrating that they knew how to do some of the necessary beginner Git tasks required for success in the course. As the course progressed, new commands and Git concepts would be introduced with accompanying materials. Some of these Git concepts were required to be demonstrated on later assignments, like Git tagging.

After the introduction of the Shell Tutor, teaching of the command shell and Git changed. Students are still provided with lecture notes detailing important commands for their success in the course. However, less class time was spent teaching these concepts compared to previous semesters. This class time was significantly reduced initially, but over time was increased to the point that nearly a whole lecture is spent introducing the command shell and Git as the initial Shell Tutor assignment is introduced. As part of the initial Shell Tutor assignment, students still create and submit a sample Git repository demonstrating their completion of the Shell Tutor and associated completion files, with the direct guidance of the Shell Tutor. In later semesters students were provided with multiple Shell Tutor assignments after the initial assignment, teaching more advanced command shell and Git concepts. All assignments in the course, before and after the introduction of the Shell Tutor, are submitted through Git to an online repository that the instructor manages.

4.3 The Shell Tutor

4.3.1 Description Of Tool

The Shell Tutor is a constraint-based intelligent tutoring system which provides a student with lessons [26]. Lessons are a sequence of steps, which are individual tasks for a student to complete. When a student starts a lesson, the Shell Tutor “attaches itself” to a students command shell, executing tasks in both the pre-command execution and

```
Tutor: After your SSH key is created, you'll be shown its fingerprint and
Tutor: randomart image.  They look weird, and are mostly harmless, so
Tutor: you can ignore them.
Tutor:
Tutor: Run ssh-keygen -t rsa -b 2048 to create your SSH key.

5-ssh-key.sh - Step 0 of 6 [------]
[student@Computer ~/shell-tutor]
$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:

[ Text Removed For Brevity ...]

Tutor: Let's navigate to the directory that contains your SSH key.
Tutor:
Tutor: Change directories into ~/.ssh.

5-ssh-key.sh - Step 1 of 6 [+-----]
[student@Computer ~/shell-tutor]
$
```

Fig. 4.1: A demonstration of the Shell Tutor evaluating a students command, determining it is a success, and moving to the next step. In this example, the student creates a new SSH key.

```
Tutor: Begin by saying "Hello World" the command-line way.
Tutor: In Python you would write:
Tutor:   print("Hello, World")
Tutor:
Tutor: But here in the shell it looks like this:
Tutor:   echo Hello, World
Tutor:
Tutor: Notice that the arguments Hello, and World are not quoted.
Tutor:
Tutor: Run this command now.

0-basics.sh - Step 2 of 17 [++---------------]
[student@Computer ~/shell-tutor/lesson0]
$ echo hello
hello

Tutor: You gave echo too few arguments
Tutor:
Tutor: Run echo Hello, World
0-basics.sh - Step 2 of 17 [++---------------]
[student@Computer ~/shell-tutor/lesson0]
$
```

Fig. 4.2: A demonstration of the Shell Tutor evaluating a students command, determining it was incorrect, and providing targeted feedback based on the student's action.

post-command execution phases, allowing the student full control of the shell in between. Students submit “solutions” to a step by inputting a command to the command shell, attempting to accomplish the task provided by the Shell Tutor. In the pre-command execution phase, the command a student executes is captured, and additional context about the state of the shell is gathered for logging. The command the student enters is then executed by the system shell, rather than simulated as other systems do. In the post-command execution phase, before the student receives the next prompt, the inputted command and its side effects are evaluated. The student will be provided with feedback on their action and may proceed to the next step if successful. Depending on the action a student takes, the system may conduct tutor-performed actions to correct a mistake made by the student, such as the deletion of a necessary file, or provide the student with hints and more instruction. The tutor may also conduct setup actions for the next step during post-command execution phase. Examples of a successful action can be seen in Figure 4.1 and a failed action can be seen in Figure 4.2. This system adheres to the VanLehn two-loop model for intelligent tutoring systems, where the outer loop corresponds to the tasks involved in a lesson, and the inner loop corresponds to the evaluation of the commands ran to complete a task [35]. No machine learning is used in this tool.

Upon completion of a lesson, log files and a file indicating lesson completion are saved. After completing all lessons in an assignment, the final Shell Tutor lesson validates the students completion of other lessons and assists them in submitting a Git repository with a certificate and log files included, indicating their completion of the assignment. Students are assigned a grade of full credit if completion of the lessons is done, and no credit if the lessons were not completed. The instructor and TAs would assist students who experienced significant difficulties or bugs with the system in this process.

The Shell Tutor has been developed to work with both the Bash and Zsh shells [36,37]. As of Fall 2023, it supports Bash versions >= 4.2 and Zsh versions >= 5.2. Most modern Linux and Mac computers primarily utilize these versions of Bash or Zsh natively out of the box, and Windows users can easily install tools to make these shells accessible, such as Git For Windows (which comes bundled with Git Bash), Windows Subsystem for Linux (WSL), or Cygwin.

The Shell Tutor library and current Shell Tutor lessons can be found on GitHub at the iticse2024-shell-tutor repository[2]. After cloning the repository, a lesson may be started by executing the relevant `tutorial.sh` or lesson file.

4.3.2 Development Of Lessons

Development of the Shell Tutor library is in native Bash and Zsh scripting, with some dependencies on Git and its packaged tools. This decision was made to keep installation work required by the user as minimal as possible, increasing accessibility of this tool. Because of this, the technical features of this tool are relatively limited to features of Bash and Zsh scripting and Git features. Much can be done with this limited tool set. The initial course, for which the tool was developed, primarily contains Python projects. Due to the expectation students would have Python installed, some Shell Tutor lessons utilize short

[2]https://github.com/jaxtonw/iticse2024-shell-tutor

Python scripts to increase compatibility or to assist in illustrating select shell concepts (such as teaching a student about various output streams a program may print to). Python is not required for the Shell Tutor standard library, but Shell Tutor functionality can be extended by requiring other installed tools for a lesson, as we have done occasionally using Python.

The Shell Tutor requires code to be executed in both the pre-command and post-command execution phases. This is a feature that is native to Zsh with the `preexec` and `precmd` function hooks. The developers had to add the Bash-Preexec project to the Shell Tutor to allow users of the Bash shell access to the `preexec` function hook. With the existence of these features, the developers of the tool were able to create a "standard library" that can be installed and used to create lessons in the Shell Tutor environment. Lesson files add the library files to the system `$PATH` while the lesson is in progress, and then select library features can be `source`d to add their features to the tutor. A lesson can then be defined and started by sourcing a specific file and running a library command which declares the lesson order after functions for each lesson task are defined.

To define a single task in a lesson, authors are only required to write a `prologue` function and a `test` function to print directions and define the completion of a task. Additional functions can be defined to add more functionality to a task. For example, lesson authors can add a `hint` function to provide intelligent hinting based on student failures. A `pre` and `post` function can conduct setup and cleanup tasks for a task. An `epilogue` function may provide intelligent post-task completion feedback. Additional functions may be defined to generate more in-depth logs for specific tasks. Authors may use parts of the Shell Tutor "standard library" to assist in defining any of these functions. More documentation of this can be found on our GitHub.

4.3.3 Logging Features

The Shell Tutor has optional features to enable logging of user commands. Each command a user runs has context such as the timestamp, current working directory, command, exit code of the command, execution duration of the command, and Shell Tutor step logged. Additionally, Shell Tutor lessons log the status of tutor tests that are run to determine if a student succeeds at a step. Authors have the option to define a function which can be executed to gather additional “state log” information: such as defining a function which queries the state of a Git repository, or inspecting the existence and contents of select files.

Each time a new Shell Tutor lesson starts it generates a unique ID for the log session and the logger collects information on the students home directory, the directory the lesson started in, the time the lesson started, the system `$PATH`, operating system and shell information, which lesson is being run, and which version of the Shell Tutor is being used at the time.

All this information has enabled the ability to further study student behaviors in the Shell Tutor. Preliminary analysis of this data has allowed researchers to begin studying student struggle with select lesson steps, active-time spent on the lessons, time it takes students to determine their first action on a step, different ways to complete select steps, identify the “gaming” of the Shell Tutor hint system on certain lessons, which steps disengage students more, and the identification of behaviors that indicate later poor-performance in the associated course. Further research is being done on this that will be shared in future publications. The logging tool that is used in the Shell Tutor is also being developed as a standalone tool to enable further research independent of the Shell Tutor. Because of the extensive nature of this logging, we worry there may be negative perceptions and concerns from the end user. We study student perspectives on this logging, discussed later in Section 4.4.5.

4.4 Survey And Results

A survey was conducted in Fall 2023 to gauge student opinions about the Shell Tutor, in accordance with our institution’s IRB protocols. There were 74 respondents from the Fall 2023 semester, and 87 from students who took the course in prior semesters. This was 60% of Fall 2023 Shell Tutor users, and 15% of all Shell Tutor users at the time of the survey.

Many questions were included on this survey to build a profile of the student, including their prior use of the command shell. Students were asked to share their perceptions about

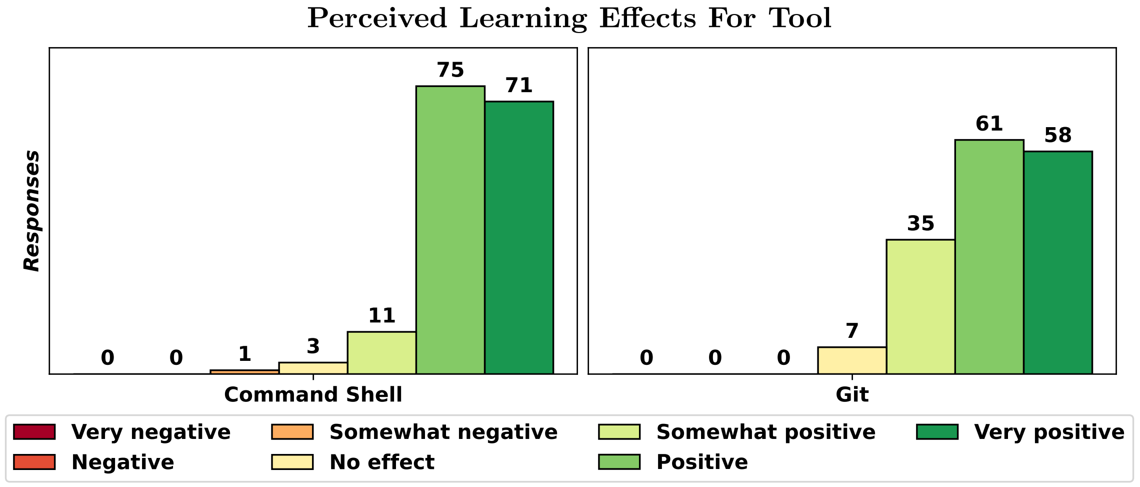

Fig. 4.3: Results from all students regarding the Shell Tutor's learning effects on the various tools it teaches. Results are overwhelmingly positive learning effects for both tools, even more so for the command shell. More info can be found in Section 4.4.1.

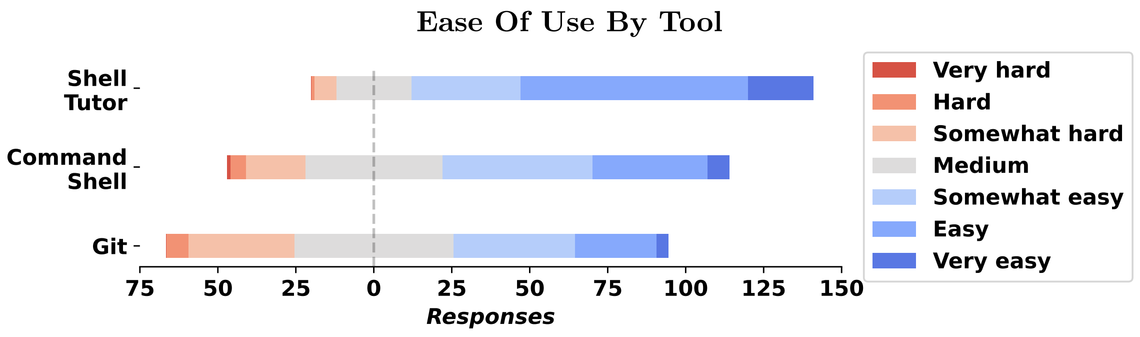

Fig. 4.4: Results from all students regarding the difficulty to use each of the noted tools. Students found the Shell Tutor significantly easier to use than the command shell and Git, with Git being the most difficult to use. More info can be found in Section 4.4.2.

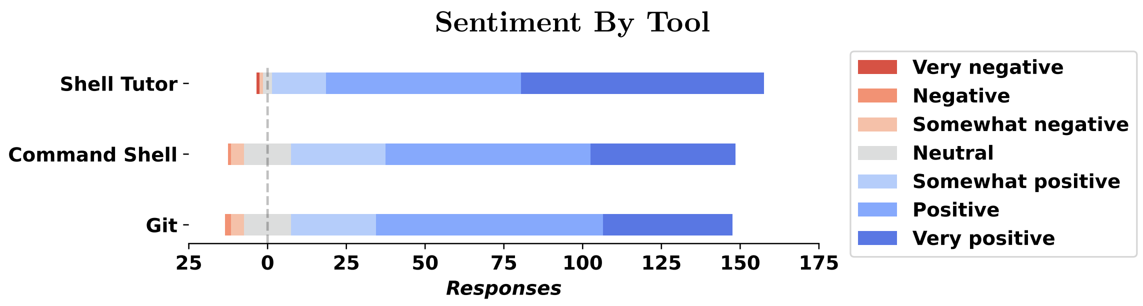

Fig. 4.5: Results from all students regarding feelings about each the Shell Tutor, command shell, and Git. Students felt overwhelmingly positive about each tool, even more so about the Shell Tutor. More info can be found in Section 4.4.3.

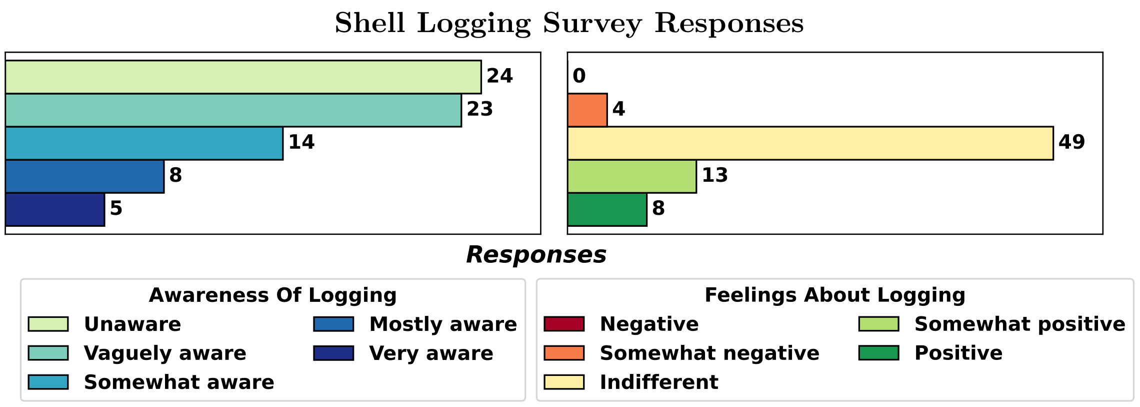

Fig. 4.6: Perspectives on shell logging from the Fall 2023 students. In general, students were vaguely aware of the shell logging and felt indifferent about its existence. More info can be found in Section 4.4.5.

various tools related to the Shell Tutor, responding on a Likert scale. Additional free response questions were provided to allow students to provide more detailed information as needed.

4.4.1 Learning Of Command Shell And Git

All students were asked about their feelings on the Shell Tutors impact on their learning. One question focused on learning the command shell, the other on learning Git. Responses were on a 7-point Likert scale, as follows: *(1) Very negative effect, (2) Negative effect, (3) Somewhat negative effect, (4) No effect, (5) Somewhat positive effect, (6) Positive effect, (7) Very positive effect.*

Overall, students felt very positively about the Shell Tutors effects on their learning of the command shell, as seen in Figure 4.3 (Mean: 6.32, Median 6). Students also felt very positively about the Shell Tutor's effects on their learning of the Git version control system (Mean: 6.06, Median: 6).

4.4.2 Perceived Difficulty Of Shell Tutor, Command Shell, And Git

All students were asked to rank the difficulty of working with the Shell Tutor system, the command shell, and the Git version control system. These responses were on a 7-point

Likert scale, as follows: *(1) Very easy to work with, (2) Easy to work with, (3) Somewhat easy to work with, (4) Medium difficulty to work with, (5) Somewhat hard to work with, (6) Hard to work with, (7) Very hard to work with.*

Over all sample groups, students on average found the Shell Tutor somewhere between "easy to use" and "somewhat easy to use," as seen in Figure 4.4 (Mean: 2.54, Median: 2). The same students also had an average ranking of the difficulty of working with the command shell and Git as somewhere between "somewhat easy to use" and "medium difficulty to use" (Command Shell Mean: 3.31, Median: 3; Git Mean: 3.66, Median 4).

Comparing these rankings of difficulty revealed a statistically significant difference, indicating that students found the Shell Tutor far easier to use than the command shell (Paired t-test $t(161) = -7.06$, $p = 4.78 * 10^{-11}$). Similarly, comparing the rankings of difficulty between the Shell Tutor and Git reveals a statistically significant difference, indicating students found the Shell Tutor easier to use than the Git version control system (Paired t-test $t(161) = -10.1$, $p = 5.47 * 10^{-19}$).

When restricting to only students of the Fall 2023 semester who had prior experience with the shell, with a sample size of 20 students, we find similar results. Comparing these students rankings of difficulty also revealed a statistically significant difference, indicating that even students with prior shell knowledge found the Shell Tutor easier to work with than the command shell (Paired t-test $t(20) = -3.13$, $p = .005$). Similarly, comparing the rankings of difficulty between the Shell Tutor and Git reveals a statistically significant difference, indicating the students who took the course most recently and have prior shell knowledge found the Shell Tutor easier to use than the Git version control system (Paired t-test $t(20) = -4.59$, $p = .0002$).

4.4.3 Feelings About Shell Tutor Vs. Other Tools

Students responded to the questions, "How do you feel about the Shell Tutor in general?", "Please describe your feelings on the command shell and command line interface (not the Shell Tutor)," and "Please describe your feelings on the Git version control system." To compare the feelings about these tools, paired t-tests were conducted between feelings on the Shell Tutor and the other tools. All responses were on the the same 7-point Likert scale, as follows: *(1) Very negative, (2) Negative, (3) Somewhat negative, (4) Neutral, (5) Somewhat positive, (6) Positive, (7) Very positive.*

General feelings on the Shell Tutor were also significantly positive, As seen in Figure 4.5 (Mean: 6.29, Median: 6). Feelings on the command shell were slightly less positive, but still had a tendency towards positivity (Mean: 5.81, Median 6). Feelings on the Git version control system were comparable to feelings on the command shell (Mean 5.78, Median 6). Statistical significance was found indicating that students have more positive feelings on the Shell Tutor than the command shell (Paired t-test $t(161) = 4.61$, $p = 8.20*10^{-6}$). Similarly, statistical significance was found indicating that students have more positive feelings on the Shell Tutor than the Git version control system (Paired t-test $t(161) = 4.89$, $p = 2.51*10^{-6}$).

4.4.4 Negative Perceptions

To better understand the negative perceptions about the tool, we directly analyzed all responses that believed the Shell Tutor had a negative impact on their learning or had general negative opinions about the Shell Tutor. We analyzed these survey responses as a whole and built a model of the students with these experiences to better understand these perceptions and identify what may contribute to a students poor perceptions of the Shell Tutor.

One student's response indicated that they had very negative opinions about the Shell Tutor. This student had taken the course in a prior semester, and indicated that they had significant prior proficiency with the shell. They indicated that the Shell Tutor did not have any impact on their learning of the command shell or Git, they have very positive feelings of the command shell and Git, and they frequently use the shell. In free response questions, this student indicated feelings that the Shell Tutor system was *too* restrictive for a student with shell proficiency of their caliber. They did not like that it forced them to accomplish certain tasks, felt that there was too much text, and wished there were clearer summaries of the lessons provided.

An additional student noted somewhat negative opinions about the Shell Tutor. This student had a similar profile and responses to the prior student and had prior experience with the shell. They did note the Shell Tutor had a positive effect on their learning of Git and noted that the Shell Tutor and its associated lessons were very easy to work with. Unfortunately, this student did not leave any free responses allowing us to analyze their perspectives on the tool more. From their other survey responses, we can infer that their negative opinions about the Shell Tutor do not come from the Git related tasks or any perceived difficulty of using the system. We infer their opinions about the Shell Tutor are due to similar reasons as the prior student studied. However, this is only speculation, as we lacked any elaboration in this student's response.

Another student indicated that they believed the Shell Tutor to have had a *negative* effect on their learning of the command shell. Interestingly, this student listed that they have *positive* opinions of the Shell Tutor. They had positive opinions of both Git and the command shell, and found the Shell Tutor to have "medium difficulty" to work with. However, this student indicated that they are "unsure" if they understand the differences between the command shell and Git. Investigating their free responses, the student indicated that they believed the *Shell Tutor* made the other course assignments harder: assignments which were independent of the Shell Tutor, but *did* require the use of Git and the command shell. They provided yet another fascinating response in the free response, indicating that using the Shell Tutor allowed them to feel as if they were learning and that it was helpful outside of the class. This whole response seemed to have many contradictory and unclear responses. This leads us to suspect that the student may have at times been confused between the Shell Tutor, the command shell, and the Git version control system.

4.4.5 Feelings About Shell Logging

Students from the Fall 2023 cohort responded to questions about their feelings on shell logging. The first question being, "Until this survey and any associated materials for this survey, how aware were you of the command logging feature of the Shell Tutor (for Assignment 1.1 and Assignment 2.1)?" These responses were on a 5-point Likert scale, as follows: *(1) I was entirely unaware of this, (2) I vaguely remember this feature, (3) I* *somewhat remember this feature, (4) I was mostly aware of this feature, (5) I was very aware of this feature.*

These students were also asked the question, “What are your feelings about the existence of this shell logging feature and the sharing of these logs with your instructor?” These responses were on a 5-point Likert scale as follows: *(1) Negative feelings, (2) Somewhat negative feelings, (3) I feel indifferent about this feature, (4) Somewhat positive feelings, (5) Positive feelings.*

As seen in Figure 4.6, students tended to somewhat be aware of the existence of the shell logging feature (Mean: 2.28, Median: 2). Regarding feelings about the logging, students felt mostly indifferent about the feature (Mean: 3.34, Median: 3).

For the question of awareness, it is important to note that the survey was conducted about a month after the last Shell Tutor assignment was complete. The instructor did note that there was logging of the shell commands at the time, and students were required to submit a compressed file of these logs with their submission, which required them to manually `git add` the file, clearly named `shell-logs`.

In the free responses, many students requested to know more about the logging feature. Some seemed confused and believed that these logs were purely the reported timer at the end of a Shell Tutor session. Many students shared the opinion that the logging features were good, and some suggested they be used to create a “report” that can be shared with students to allow them to see their performance and struggles with the Shell Tutor. However, these same students added the caveat that these logs should never be used to influence a students grade on the Shell Tutor assignments. They suggested that the grades remain entirely based on completion, and that the logs and observed behaviors do not influence grade outcomes. One student expressed concern that if they were more aware of the command logging feature, they would have been less likely to experiment in the Shell Tutor, out of fear that the instructor may “judge” their performance.

4.5 Conclusions, Reflection, And Future Work

We found that students believed there to be a significant positive effect on their learning of both the command shell and Git, supporting the belief that the Shell Tutor is helpful for student learning. We also found that students believe the Shell Tutor to be easier to work with than the tools they are designed to teach. We believe this to indicate that our tool provides a lower barrier to entry for learning both the command shell and the Git version control system. Students are able to use this tool effectively as a first step to learn the command shell and Git. We also found that students generally have better feelings about our tool than the tools they teach. We believe these findings to be yet another indicator that the tools development is in the right direction, and supports the belief that this tool assists student learning.

Some negative perspectives exist about the tool. Notably, some students who have more experience with the command shell can feel burdened by being required to use this tool. Reflecting on this, we will work to identify ways to make select steps less rigid in the solutions it accepts, and may even explore ways to "test out" of some Shell Tutor assignments.

Despite our concerns that shell logging may be a controversial feature, we found that students generally have a positive opinion about shell logging. Though, we believe there should be more transparency about the logging, and assurance to the students that these logs are not going to be used to "judge" their performance or effect their scores on the assignments. We intend to keep the shell logging features of this tool going forward, enhancing the research capabilities of this tool.

Overall, we are very pleased with the Shell Tutor's impacts on our students. We believe our findings indicate that the Shell Tutor has been a resounding success for our students, and we suspect it has the potential to assist many other students learning the command shell and Git.

Future work will be done to better understand the log data collected, understanding more of how students utilize the tool. We hope to use these logs to enable artificial intelligence in the development of the tool and identification of student struggles. We also wish to better understand perceptions of these logs and find the "sweet spot" between awareness of the logging tool–for transparency–and “over-awareness” that may influence behaviors. Tools to generate better reviews of student performance on the Shell Tutor will be explored as well. We hope to expand the suite of lessons the Shell Tutor provides to more advanced topics in the future, and guide others in creating their own lessons with the Shell Tutor library.

CHAPTER 5

CONCLUSIONS

In this thesis, we have seen a sequence of research projects focused on computer science classroom interventions. Chapter 2 detailed the successful design, deployment, and analysis of an intervention for student procrastination. Chapter 3 researches the generally accepted "good practice" of incremental development, and provides evidence indicating its correlation with efficient work for CS1 students. Chapter 4 analyzes the design and deployment of the novel intelligent tutoring system, the Shell Tutor, in the USU CS1440 course. This chapter will recap the conclusions of each research project detailed in these chapters.

When analyzing the correlation between early project analysis and better performance on the project, Chapter 2 did find correlation between early project analysis and project performance. As can be seen in Section 2.6.1, we see statistical significance [1] when all assignment performances were aggregated together. We also see statistical significance when restricting the analysis to only Assignment 5.0, the most difficult assignment of the 3 studied. For Assignment 3 and Assignment 4, we do not see individual significance according to our chosen p-value threshold, though we do see trends indicating better performance for those who analyze their projects earlier, with relatively low p-values[2]. With this research, we feel that it is conclusive to state that early project analysis correlates with better project performance, and the effect is more pronounced on more difficult assignments.

Looking towards the question of grace points affecting student behaviors in Chapter 2, we also found significant evidence supporting the introduction of grace points as causation for students starting assignments earlier. We see in section 2.6.2 that when looking at student behaviors before introducing grace points, the mean time students started on the assignment was with 35.5% of the assignment period remaining, roughly 4.3 days before the assignment due date. However, in the semester grace points were introduced, the mean

[1] Significance is determined when following the conventional threshold of significance where $p < .05$.

[2] $p = 0.0849$ for Assignment 3, and $p = 0.0922$ for Assignment 4.

time students started on the assignment was with 52.3% of the assignment period remaining, roughly 6.3 days before the assignment due date. Additionally, within the semester the grace points intervention was deployed, we compared the assignment starting behaviors before the introduction of grace points and after. We see similar results, with the mean start time of assignments with grace points as having 52.3% of the assignment period remaining (≈ 6.3 days remaining) and the mean start time of the assignments before grace points being introduced as 44.1% of the assignment time remaining (≈ 5.3 days remaining). All of these comparisons showed statistically significant differences between the groups, indicating that the grace point intervention caused students to start on their assignments earlier.

Chapter 2 clearly shows that the intervention of grace points for early project analysis in CS1440 was a success. It positively influenced student behaviors, encouraging them to start on their projects earlier. We also found that earlier starting behaviors correlated with success on the assignment, indicating that this intervention helped encourage student success on their assignments in the course.

In Chapter 3, we study the effects that incremental development has on students. We have two main research questions: **1) does incremental development correlate with better assignment performance** and **2) what trends are found in student behaviors when they perform incremental development as measured by the MID scores?**

Within this chapter, we find that there was no evidence correlating incremental development with better assignment performance, with two notable caveats. First, we only utilize the MID score metric defined by Shah et al. [10] for our analysis, and this metric has its own definition of incremental development coded into its creation, focused on student edit behaviors. The definition of incremental development is not standardized, and other works utilize slightly different definitions of incremental development, such as [11]. It is possible that repeating our study with other analyses for incremental development, using slightly different definitions, may produce different results. Additionally, our dataset had a significant ceiling effect on the scores. A significant portion of the assignments in our study had scores above 90%, so a replication of our study using a dataset which contains more discrimination between performances would enable better conclusions to be drawn regarding this question. The study detailed in Chapter 3 found no correlation between incremental development and assignment performance, using the MID score metric and the USU CS1 student keystroke dataset [20].

On the topic of student behavioral trends correlating with incremental development, we found in Chapter 3 that incremental development correlates with select assignment creation behaviors. We found correlation between incremental development and a fewer number of interaction events[3], a lower average assignment time[4], fewer number of times students run their assignments[5], and a higher percentage of IDE events happening before running their code in a given coding session[6]. We did not find any correlation between incremental development average time between interaction events, number of coding sessions, and amount of time between running code.

When paired with our findings of no discernible differences in assignment scores between assignments developed incrementally and those that were not, we conclude that incremental development correlates with more *efficient* work on assignments. The work in Chapter 3 does not establish any causation, and only establishes correlation. However, we strongly suspect that there is causation, and this should be studied in future work. The work in Chapter 3 establishes baseline research for future computing education researchers to continue on, supporting the development of future informed interventions that may one day encourage students to complete their assignments more efficiently.

Within Chapter 4, we detail the design, deployment, and analysis of the novel intelligent tutoring system, the Shell Tutor. This tool was developed and deployed to help students in the USU CS1440 class learn command shell and Git basics. We cover technical aspects of the Shell Tutor and its design in a manner that could enable other computing educators

[3]Nearly half the number of interaction events, when compared to non-incremental assignments.

[4]Nearly half the time spent on assignments, when compared to non-incremental assignments.

[5]Nearly half the number of assignment runs, when compared to non-incremental assignments.

[6]An 11 percentage point difference. We suspect this metric correlates with more "forward progress" in a given coding session, as we theorize that coding sessions with a higher percentage of code written *after* code has been ran becomes primarily focused on debugging. This metric has not been validated, and the results related to this metric should be interpreted with caution.

to adopt it into their classroom. The way the course has changed in the wake of the Shell Tutor's deployment is also briefly discussed.

Lastly, we analyze student perceptions of the tool, which are overwhelmingly positive. Students generally felt the Shell Tutor positively affected their learning of both the command shell and Git, viewed the tool positively, and generally found it to be between "easy to use" and "somewhat easy to use." Additionally, we isolated the few responses of students who had negative perceptions or experiences with the tool to find trends. We found that students who had negative perceptions of the tool were commonly students who had prior significant proficiency with the command shell, and felt that the Shell Tutor limited them in their ability to complete exercises and disliked the requirement of using it to complete select tasks. One of the negative responses had indications that a student may have been confused as to what exactly the tool is, and we believe that the tool could be improved by drawing clearer distinctions between itself, the command shell, and Git. We also observe that students had primarily neutral-towards-somewhat-positive opinions about the Shell Tutor logging features.

With our analysis of the Shell Tutor in Chapter 4, we see significantly positive opinions and perceived impact on students of the CS1440 course. We also identify some specific ways that the Shell Tutor could be improved in the future, and use this analysis as a means to adjust the future development directions of the Shell Tutor. We find the results of this analysis to be very pleasing, indicating that the development of the Shell Tutor has been on the right track so far, and has been assisting the students of the USU CS1440 course in their learning of the command shell and Git.

This thesis finds overwhelming success in the design, deployment, and analysis of the various course interventions detailed. With this research, we have seen significant improvements to the computer science classroom at Utah State University, and hope to take these improvements beyond this institution. We also detail research that supports future work towards classroom interventions, and hope that future computing education researchers may build upon these foundations to continue improving the computer science classroom.

REFERENCES

[1] R. Cerezo, M. Sánchez-Santillán, M. P. Paule-Ruiz, and J. C. Núñez, "Students' lms interaction patterns and their relationship with achievement," *Comput. Educ.*, vol. 96, no. C, p. 42–54, may 2016. [Online]. Available: https://doi.org/10.1016/j.compedu.2016.02.006

[2] Akinsola, Mojeed Kolawole, Tella, Adedeji, and Tella, Adeyinka, "Correlates of academic procrastination and mathematics achievement of university undergraduate students," *Eurasia Journal Of Mathematics, Science and Technology Education*, vol. 3, no. 4, pp. 363–370, 2007.

[3] C. Grunschel, M. Schwinger, R. Steinmayr, and S. Fries, "Effects of using motivational regulation strategies on students' academic procrastination, academic performance, and well-being," *Learning and Individual Differences*, vol. 49, pp. 162–170, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1041608016300905

[4] M. Kokoç, G. Akçapınar, and M. N. Hasnine, "Unfolding students' online assignment submission behavioral patterns using temporal learning analytics," *Educational Technology & Society*, vol. 24, no. 1, pp. 223–235, 2021. [Online]. Available: https://www.jstor.org/stable/26977869

[5] S. H. Edwards, J. Martin, and C. A. Shaffer, "Examining classroom interventions to reduce procrastination," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 254–259. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/2729094.2742632

[6] Z. Felker and Z. Chen, "The impact of extra credit incentives on students' work habits when completing online homework assignments," in *Physics Education Research Conference 2020*, ser. PER Conference, Virtual Conference, July 22-23 2020, pp. 143–148.

[7] R. Włodarski, A. Poniszewska-Marańda, and J.-R. Falleri, "Comparative case study of plan-driven and agile approaches in student computing projects," in *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2020, pp. 1–6.

[8] D. Umphress, T. Hendrix, and J. Cross, "Software process in the classroom: the capstone project experience," *IEEE Software*, vol. 19, no. 5, pp. 78–81, 2002.

[9] L. Beus-Dukic and I. Alexander, "Learning how to discover requirements," in *2008 Requirements Engineering Education and Training*, 2008, pp. 12–14.

[10] A. Shah, M. Granado, M. Sharma, J. Driscoll, L. Porter, W. G. Griswold, and A. G. Soosai Raj, "Understanding and measuring incremental development in cs1," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023. New York, NY, USA: Association

for Computing Machinery, 2023, pp. 722–728. [Online]. Available: https://dl.acm.org/doi/10.1145/3545945.3569880

[11] A. M. Kazerouni, S. H. Edwards, and C. A. Shaffer, "Quantifying incremental development practices and their relationship to procrastination," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 191–199. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3105726.3106180

[12] A. Shah. (2022) Measure incremental development : Python package index - pypi. PyPI. [Online]. Available: https://pypi.org/project/measure-incremental-development/

[13] L. Baumstark and M. Orsega, "Quantifying introductory cs students' iterative software process by mining version control system repositories," *J. Comput. Sci. Coll.*, vol. 31, no. 6, p. 97–104, jun 2016.

[14] L. Glassy, "Using version control to observe student software development processes," *J. Comput. Sci. Coll.*, vol. 21, no. 3, p. 99–106, feb 2006.

[15] P. Mäkiaho, T. Poranen, and A. Seppi, "Version control usage in students' software development projects," in *Proceedings of the 15th International Conference on Computer Systems and Technologies*, ser. CompSysTech '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 452–459. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/2659532.2659646

[16] A. M. Kazerouni, S. H. Edwards, T. S. Hall, and C. A. Shaffer, "Deveventtracker: Tracking development events to assess incremental development and procrastination," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 104–109. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3059009.3059050

[17] R. Moazeni, D. Link, and B. Boehm, "Incremental development productivity decline," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/2499393.2499403

[18] R. Moazeni, D. Link, C. Chen, and B. Boehm, "Software domains in incremental development productivity decline," in *Proceedings of the 2014 International Conference on Software and System Process*, ser. ICSSP 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–83. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/2600821.2600830

[19] M. McGregor, L. Azzopardi, and M. Halvey, "Untangling cost, effort, and load in information seeking and retrieval," in *Proceedings of the 2021 Conference on Human Information Interaction and Retrieval*, ser. CHIIR '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 151–161. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3406522.3446026

[20] J. Edwards, "2021 CS1 Keystroke Data," 2022. [Online]. Available: https://doi.org/10.7910/DVN/BVOF7S

[21] J. Edwards, J. Ditton, B. Sainju, and J. Dawson, "Different assignments as different contexts: predictors across assignments and outcome measures in cs1," in *2020 Intermountain Engineering, Technology and Computing (IETC)*. New York, NY, USA: IEEE, 2020, pp. 1–6.

[22] K. Hart, C. M. Warren, and J. Edwards, "Accurate estimation of time-on-task while programming," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 708–714. [Online]. Available: https://doi.org/10.1145/3545945.3569804

[23] J. Ditton, H. Swanson, and J. Edwards, "External imagery in computer programming," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1226–1231. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3408877.3432426

[24] S. Overflow, "Stack Overflow developer survey 2022." [Online]. Available: https://survey.stackoverflow.co/2022

[25] "Git," 2023, [Online; accessed 08-December-2023]. [Online]. Available: https://git-scm.com/

[26] A. Mitrovic, *Modeling Domains and Students with Constraint-Based Modeling*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 63–80. [Online]. Available: https://doi.org/10.1007/978-3-642-14363-2_4

[27] V. Berry, A. Castelltort, C. Pelissier, M. Rousseau, and C. Tibermacine, "Shellonyou: Learning by doing unix command line," in *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, ser. ITiCSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 379–385. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3502718.3524753

[28] M. Šuppa, O. Jariabka, A. Matejov, and M. Nagy, "Termadventure: Interactively teaching unix command line, text adventure style," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 108–114. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3430665.3456387

[29] J. Bailey and C. Zilles, "Uassign: Scalable interactive activities for teaching the unix terminal," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 70–76. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3287324.3287458

[30] "Tuxlab," 2023, [Online; accessed 10-November-2023]. [Online]. Available: https://tuxlab.org

[31] A. Solomon, D. Santamaria, and R. Lister, "Automated testing of unix command-line and scripting skills," in *2006 7th International Conference on Information Technology Based Higher Education and Training*, 2006, pp. 120–125.

[32] J. Mirkovic, A. Aggarwal, D. Weinman, P. Lepe, J. Mache, and R. Weiss, "Using terminal histories to monitor student progress on hands-on exercises," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 866–872. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/3328778.3366935

[33] T. Greening, "Command-line usage in a programming exam," *SIGCSE Bull.*, vol. 28, no. 3, p. 55–59, sep 1996. [Online]. Available: https://doi-org.dist.lib.usu.edu/10.1145/234867.234879

[34] V. Švábenský, J. Vykopal, D. Tovarňák, and P. Čeleda, "Toolset for collecting shell commands and its application in hands-on cybersecurity training," in *2021 IEEE Frontiers in Education Conference (FIE)*, 2021, pp. 1–9.

[35] K. VanLehn, "The behavior of tutoring systems," *Int. J. Artif. Intell. Ed.*, vol. 16, no. 3, p. 227–265, aug 2006.

[36] "Gnu bash," 2023, [Online; accessed 08-December-2023]. [Online]. Available: https://www.gnu.org/software/bash/

[37] "Z shell," 2023, [Online; accessed 08-December-2023]. [Online]. Available: https://www.zsh.org/

CURRICULUM VITAE

Jaxton Winder

Published Conference Papers

- The Shell Tutor: An Intelligent Tutoring System For The UNIX Command Shell And Git, Jaxton Winder, Erik Falor, Seth Poulsen, and John Edwards in *ITiCSE 2024: Proceedings of the 2024 Conference on Innovation and Technology in Computer Science Education*, 2024.

- Incremental Development and CS1 Student Outcomes And Behaviors, Jaxton Winder, Elise Francis, Bridget Staley, and John Edwards in *ACE 2024: Australian Computing Education Conference*, 2024.

- Early Submission of Project Analysis Milestones Correlates Positively With Student Project Performance; Incentives for This Early Project Analysis Positively Changes Student Behaviors, Jaxton Winder, Erik Falor, and John Edwards in *Intermountain Engineering, Technology and Computing (IETC)*, 2023.