ABSTRACT

SILVA DE OLIVEIRA, GABRIEL. Understanding Coding Struggle in a Large Project-Based CS2 Course. (Under the direction of Dr. Collin Lynch and Dr. Sarah Heckman).

Computer Science enrollment has grown over the past few years, with some times having 200 to 300 students enrolled in introductory courses. On the other hand, several students may have a hard time learning programming, since it requires skills not commonly taught in early education. With a large amount of students struggling, it becomes impossible for instructors to offer individual, timely feedback for each student. Additionally, research on novice programmers and Computer Science students tends to focus on CS1 courses, leaving a research gap on the exploration of CS2 courses. Seeing the challenges students were facing, combined with the difficulty to offer timely feedback, we decided to use research to aid students' success in CS2 courses. The goal of this research is to find a way to identify struggle through submission data, particularly submission data from large projects. We use submission data because it is the most readily available data for the teaching staff, but also because courses are generally centered on assignments. We create a definition of struggle based on persistent errors. We separate students into low and high struggle groups and analyze coding metrics, such as code line changes, to understand their coding behavior before and during struggle. We find meaningful differences between groups, such as students in high struggle groups tend to focus on more code files simultaneously while focusing less on test code development. Additionally, we explore the relationship between our definition of struggle and external factors, such as demographics and prior-experience. We believe this research can lead to interventions and timely feedback that can be given to the students during the period where they are working on a programming assignment, which could potentially increase student success and retention in Computer Science courses.

Understanding Coding Struggle in a Large Project-Based CS2 Course

by
Gabriel Silva de Oliveira

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Dr. Lina Battestilli	Dr. Kanton Reynolds
Dr. Collin Lynch Co-chair of Advisory Committee	Dr. Sarah Heckman Co-chair of Advisory Committee

DEDICATION

To my parents, for showing me the value of education, and to my wife, for always loving and supporting me through every step of this journey.

ACKNOWLEDGEMENTS

I would like to thank Dr. Collin Lynch and Dr. Sarah Heckman for guiding me through my years in graduate school, and my committee members, Dr. Lina Battestilli and Dr. Kanton Reynolds, for the interest in my research. I would also like to thank Dr. Amy Ort for the incredible support, and Kathy Luca, for being the first person who saw when I was not ok.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Motivations

The overarching motivation for the research presented here is to find new ways to support student success in Computer Science, particularly novice programmers. With student success, we mean not only achieving high grades and succeeding in a course, but to succeed in the course regardless of their background, and personal and social situations.

Challenges novice programmers face have been well studied, as we demonstrate in Section 1.2. However, as shown in our literature review, research on novice programmers is generally focused on introductory Computer Science courses, which we will call CS1 courses. Furthermore, there is a gap in research on efficient ways to identify students struggling in large classes. An additional research gap is when the course involves more advanced concepts or combines concepts into large projects. These complex projects that involve the integration of multiple concepts are typically seen in the second and/or third courses in an introductory Computer Science course sequence. We will call these courses CS2. Generally, research studies on introductory Computer Science students refer to students from CS1 (Izu et al. 2019; Duran et al. 2019; Kaczmarczyk et al. 2010; Karvelas et al. 2020) courses, which is another reason we chose to study a CS2 course.

Additionally, through experiences as teaching assistants and instructors, we have seen

several students facing different levels of struggle. Several students were unable to overcome challenges after working on the same programming assignment for extremely long periods. Those occurrences, combined with the struggles I experienced while I was a novice programmer, have drawn me to use research to explore ways to improve student success, particularly in environments where instructors do not have the resources or the time to offer personalized feedback for each student, such as large introductory Computer Science courses. This interest led to the first research question, *How can we identify struggle using submission data? (RQ1)*. The focus on submission data is because it is the most readily available data for instructors since they cannot monitor students' actions 27/4. Doing well on assignments is also generally the students' goal when taking a course, and most courses are designed with schedules and lectures functioning around assignments.

With the gap in research focused on struggling students in CS2 courses, we decided to focus this research work on one particular CS2 course, "Software Development Fundamentals". I have been a teaching assistant and lab instructor for the course, which covers advanced object-oriented programming and combines programming concepts in large project assignments. RQ1 is further subdivided into two sub-questions: *How to define and quantify struggle? (RQ1.1)* and *What kinds of different struggle patterns appear in the data? (RQ1.2)*. However, defining and locating struggle would be ineffective without also defining methods to identify student behaviors during struggle. Therefore, we try to identify those behaviors through our second research question, *How do students respond to struggle? (RQ2)*, which has the goal of identifying changes in coding behavior that happen when students reach a struggling point.

Aside from programming-related challenges, students may have additional difficulties due to reasons external to the courses, which I have seen affecting my students. For instance, students from less fortunate backgrounds might have to hold a job and/or help their families financially, or their previous academic experiences might have not provided them with the cognitive and academic tools needed to succeed. Furthermore, several less privileged students who face those challenges are part of groups historically underrepresented in Computer Science[1]. For example, the National Science Foundation (NSF) reports that less than 30% of Computer Science degree recipients are women, and in 2023 only 9% of science and engineering degree recipients were black students. (Stevens Institute of Technology 2024; National Science Foundation 2023).

Through my graduate education, I have learned that increasing the success of diverse student populations also increases the chances of creating new solutions in both industry and academia, as diverse groups contribute with different points of view (Milem 2003). The differences in prior experience we saw between different groups of students led us to investigate

[1]By historically underrepresented in Computer Science we mean women and non-white students.

how socio-economical differences influence the challenges faced by the students, which is the topic explored in our third research question *What are the relationships between struggle and factors external to the course? (RQ3)*. We aim to answer through survey responses and information from the US Census Bureau data.

We have evaluated our data through progress indicators, changes in feedback, and an exploration of student effort. The results of those studies led to the current research questions. The overarching goal of our research is not only to aid in student success, but also to improve accessibility to Computer Science education. We believe finding ways to identify struggling students would give them a better chance to succeed. Through this research, we hope to contribute to the field by creating a definition of struggle that can be used by other instructions, along with a better understanding of how programming students behave when facing struggle. This knowledge could potentially be used to create new assessments and new interventions that would help increase retention in introductory programming courses. We also hope to increase our understanding of how underrepresented students are affected by their background, which can hopefully lead to better ways to help them.

1.2 Background

As this research's goal is to aid in introductory programming students success, we start by understanding what challenges novice programming students face. Computer programming is a widely applicable skill in many domains, and the demand for skilled programmers grows with each passing year (Loten 2019; U.S. Bureau of Labor Statistics 2020). The Wall Street Journal reports in 2019 that "in the first half of 2019, tech job postings in the U.S. rose 32% from a year earlier", according to federal employment data analyzed by IT trade group CompTIA (Loten 2019; U.S. Bureau of Labor Statistics 2020). In the past three months, according to the same data, "U.S. employers had about 918,000 unfilled IT jobs". Programming, however, is a complex skill that students often struggle with, and many students turn away from otherwise promising STEM careers due in part to poor experiences with coding. Students who struggle during the first semesters of Computer Science courses might end up dropping courses, changing majors, or even graduating but changing fields after graduation (Kinnunen and Malmi 2006; Pappas et al. 2016; Bucciarelli 2018; National Center for Education Statistics 2017). To address this, we must understand the possible challenges faced by students. Students may present low performance for different reasons; however, the observable effects, such as low grades and a failure to complete work, can be the same. Therefore, our goal in this work is to create ways to narrow down and identify different ways novice programmers show the effects of those issues through submission data, to provide instructors with quantifiable indicators that will

enable them to offer better help to students in need. This chapter presents a literature review on students' challenges, effort, performance, progress, and code behavior and struggle in Computer Science.

1.2.1 Challenges from Cognitive Skills

While programming is a unique skill, students' success in programming is affected by more general cognitive and metacognitive skills such as self-regulation, or the lack thereof (Prather et al. 2018, 2020). These general skills are not typically addressed in CS1 course outcomes, but they are important points of consideration for instruction.

Key to a student's ability to learn in any domain or solve any problem is comprehension. This includes the students' ability to comprehend both the task and the criteria for it. Problem comprehension (Lawanto et al. 2018; Rivera-Reyes et al. 2017; Wrenn and Krishnamurthi 2019) is the process of understanding what needs to be solved when reading a problem. It depends greatly on the student's general reading comprehension skills (Cartwright et al. 2020), as well as their task and course-specific knowledge. Assessing and scaffolding initial comprehension is key for any instructional process, in part because task comprehension is foundational.

Task comprehension is closely related to the far more challenging problem of Evaluative Misalignment (EM), which arises when students do not understand the assessment criteria used for assignments or when they perceive them differently from the instructors. This phenomenon was first identified by Roscoe et al. (2016) in the domain of writing, where they noted that students often failed to understand directives to make their writing "clear" or to "citep evidence supporting their argument". This, in turn, will cause them to repeat themselves, or simply add excess words, rather than making the changes the instructor sought. This phenomenon is problematic both because it leads students to flounder by making changes to their work that do not fix the problems, or make it worse, but also because it can lead them to misinterpret the feedback that they receive. According to Roscoe et al. (2016), on writing assignments evaluative misalignment occurs when the assessment criteria of the writer differs from that from their audience. In our case, evaluative misalignment occurs when assessment criteria and/or expectations created by the instructor for an assignment is different than that perceived by the student. That could potentially result in either students misjudging how much time and/or effort they will need to apply to finish an assignment, or students completing assignments and doing worse than they expected to.

In addition to being able to comprehend tasks, students must also be able to comprehend, evaluate, and manage their own thinking. Metacognition, term created by Flavell (1976), "generally refers to an individual's awareness of their own knowledge and the ability to un-

derstand, control, and manipulate individual cognitive process" (Osman and Hannafin 1992). This includes their ability to evaluate their own understanding, and ideas, their general self-assessment, and explicit goal setting and self-regulation (Falkner et al. 2014; Prather et al. 2020; Loksa 2017). In each case these represent skills that are essential to effective learning and problem-solving in Computer Science but which are generally not taught explicitly, nor are they evaluated, particularly in the context of introductory courses.

Metacomprehension and metamemory govern an individual's ability to evaluate and manage their knowledge of concepts (Martinez 2006). Such skills are essential to problem-learning and problem-solving in all domains. This is particularly true in a technical domain like programming where students are introduced to new concepts (e.g. the use of interfaces in Java, code execution paths, or memory management) and are required to apply them constructively. Failure to understand a key coding concept, along with an incorrect belief that concept was understood correctly, could potentially make students more prone to produce incorrect code, and yet less prone to seek guidance as they believe their knowledge is sound. Avoiding these errors may require instructors to seek concept reviews on students, rather than rely on them to seek out review opportunities.

Self-evaluation is the process of analyzing one's performance, and identifying their weaknesses and strengths (Duran et al. 2019). Self-efficacy is the perceived notion one person has about their capacity to perform tasks (Bandura and Adams 2005), and it has been correlated in previous research with positive learning outcomes (Duran et al. 2019). Although not part of the categories described in (Martinez 2006), self-evaluation and self-efficacy can also be considered forms of metacognition. As Duran et al. (Duran et al. 2019) pointed out, self-evaluation analyses past and current work, while self-efficacy attempts to assess one's ability to perform future tasks. Self-efficacy and self-evaluation work together. Without a realistic process of self-evaluation, a student can potentially believe their performance is better or worse than in reality. This in turn might make it difficult for them to regulate their own learning or benefit from feedback. Kallia and Sentance (Kallia and Sentance 2019) aim to understand the impact programming misconceptions have on students' self-efficacy. The authors chose to focus on functions since it is a hard subject for students, based on other research studies. The students first answered a self-efficacy scale, then had to complete six programming tasks with increasing difficulty, and at the end they answered two 11-point Likert-scale questions. They show self-efficacy levels for both programming and Computer Science are lower in students who presented misconceptions in their work; also, students who do not present misconceptions tend to underestimate their performance, while students who do present misconceptions tend to overestimate their performance.

Self-regulation arises out of self-reflection, and it refers to the process of proactively taking

actions to improve one's learning on their own (Zimmerman and Kitsantas 2014). For instance, self-regulation is related to students creating a schedule, asking questions, creating time to work on assignments, participating in class and office hours, and any other action taken toward learning that was not required by external influence. This form of goal setting (Falkner et al. 2014; Prather et al. 2020; Loksa 2017) is distinct from the goals of programming problems specifically, and includes things like time management or learning goals. Self-regulation has a high impact on student performance (Sahranavard et al. 2018). When students are motivated to study and learn by their efforts, they are more successful in class. Students tend to demonstrate a high level of self-regulation when the subject is important to them in some form rather than just trying to achieve an extrinsic goal (Bouffard et al. 2005). Therefore, students' participation and performance depends on how invested they are in learning rather than wanting to achieve a passing grade.

Coupled with a student's ability to evaluate themselves is their ability to evaluate their ideas for their quality or critical thinking (Martinez 2006). Critical thinking can directly influence evaluation misalignment and self-efficacy. By using critical thinking, students should be able to decide whether their ideas and their methodologies are adequate to solve a problem and whether they follow logically from what was taught in the lecture. Students will need to be able to evaluate whether their work needs improvement. If students have a misunderstanding they are not aware of, they will likely evaluate the quality of their work incorrectly. Also, if they misunderstand the assessment criteria, they might decide to use methodologies that will lead to incomplete work. Like many of the previously mentioned skills, critical thinking is developed through practice. One way to help students improve their critical thinking skills is to ask students to explain the problems they are trying to solve when they ask questions about the problem (Loksa 2017).

Flores and Rodrigo (2020) attempted to develop 5 models of wheel-spinning for novice programmers and to identify wheel-spinning indicators. According to the authors, "wheel-spinning refers to the failure to master a skill in a timely manner or after a considerable number of practice opportunities" (Flores and Rodrigo 2020). When a student gets stuck in a problem and does not know why, they might never learn a skill and continuously fail that kind of problem. They created a "Meaningful Gamification System" that provided students with feedback and allowed the students to fail and learn from their mistakes. The students also had partial control of the game, choosing extra elements to answer. The control, "freedom to fail", and feedback all led to students to achieve better scores. The use of the "freedom to fail" feature implied the occurrence of wheel-spinning. Wheel-spinning models were generated by RapidMiner (Hofmann and Klinkenberg 2016) using a logistic regression classifier. Features with high correlation were considered indicators of wheel-spinning. The selection of relevant features

was performed through logistic regression and validation, and the creation of the equation afterward is how the models were created. All models had high accuracy or percentage of correct recognition.

"Problem-solving" means the explicit application of strategies (Mayer 2013). This includes general strategies, such as planning, and more programming-centric strategies such as decomposition and subgoaling, which are central to programming practice (Goldsby et al. 2009; Barr and Stephenson 2011; Hmelo-Silver 2004). While some explicit strategies are taught in CS courses, often under the heading of software engineering, it is still the case that many students enter CS1 courses without explicit strategies in hand. It is also the case that effective execution of these strategies relies on the metacognitive skills outlined above. A student cannot decompose a problem they do not understand, nor can they effectively target their work if they have an incorrect understanding of the evaluation criteria, or an overestimate of their skill. For this reason, we regard problem-solving as arising out of metacognition.

Loksa and Ko (2016) created a problem-solving methodology that describes six stages of problem-solving: (1) reinterpret the prompt, (2) search for analogous problems, (3) search for solutions, (4) evaluate a potential solution, (5) implement a solution, (6) evaluate implemented solution. Prather et al. (2018) use that methodology to understand what difficulties students who lack metacognitive awareness face when using an automated assessment tool on online learning, trying to find ways for the tool to help introductory programmers develop metacognitive skills. They performed a think-aloud in which CS1 students completed a quiz. Students who completed the quiz successfully went through the first 5 stages of problem-solving and read most of the error messages that appeared during development. The students who started the implementation without reinterpreting the problem first struggled with finding the solution, some of them specifically trying to solve a different problem that was previously given in class. All the students who did not complete the quiz failed at least one stage of the "learning how to solve the problem" theory. The authors came to the conclusion that the automated tool presents the problem, but there is no measure between viewing the problems and submitting the solution that tells whether the student understood what they were supposed to do. The basic conclusion is that students who present metacognitive awareness perform better when learning how to program. The attempted solution suggested by both Loksa et al. (2016) and Prather et al. (2018) was to explicitly teach the students about problem-solving steps, and when a student asked for help from an instructor, they had to explain which step of the problem-solving process they were working on.

de Raadt (2007) performed a literature review on papers from Australasia on CS problem-solving and novice programming. In 85 introductory programming courses in Australia and New Zealand, instructors did not agree on whether to teach problem-solving strategies and

did not uniformly distinguish teaching programming strategies from teaching programming knowledge. Studies in comprehension/performance showed that students are generally not performing at the expected level; they describe code line by line instead of giving an overall purpose and have difficulty applying programming strategies. Traditional introductory courses teach problem-solving through implicit learning, while explicit learning is more effective. One of the studies claims that an "explicit approach is more likely to produce novices who understand and apply strategies". A different study found no significant difference between teaching variables traditionally and teaching variables by using roles, however, students being taught roles of variables gave "data level descriptions" while the group taught traditionally gave "operation level descriptions". The authors say "studies have shown that novices perform very poorly on standardized program generation tests. In program comprehension tests novice performance is better, but still poorer than expected by the instructors".

1.2.2 Challenges from CS Conceptual Skills

Computer Science conceptual skills come from understanding Computer Science topics, differently than cognitive skills which come from students' life and experiences. Kaczmarczyk et al. (2010) explored how students understood 10 core concepts to assess student misconceptions: Memory Model, References, and Pointers, Primitive and Reference Type Variables, Control Flow, Iteration and Loops, Types, Conditionals, Assignment Statements, Arrays, and Operator Precedence. They conducted interviews with students, during which the students attempted to solve 18 small coding problems covering the concepts. The authors noted that students often faced problems due to using real-world semantics for variables and array lengths, and had difficulty understanding objects and computer memory management when using variables.

Swidan et al. (2018) conducted similar work on misconceptions, where they used multiple-choice exams to assess undergraduate students' incoming knowledge of code concepts. Among other things, they found that students routinely failed to understand how sequencing worked and how values are held in variables. Taken together, this work suggests that CS1 instructors should focus on addressing these misconceptions early, in part because students who fail to understand these basic concepts were unable to comprehend assignments or feedback. Interestingly, Swidan et al. (2018) also found that students who learned Scratch (Maloney et al. 2010), a block-based programming language, early along with other languages had fewer misconceptions and were able to work with the knowledge more than students who studied one or more other textual languages, or who studied Scratch alone. This suggests that early language diversity, an uncommon approach in CS1 courses, may help to build a stronger foundation for future learning.

Smith and Rixner (2019) further studied runtime errors in Python language. They focus on the evolution process of the errors. They found runtime errors were around 36% of total errors, while syntax errors were around 12%. Runtime errors took, on average, approximately two consecutive code versions to be solved. The authors claim frequent or persistent types of errors, – like type, index, and key errors – could indicate students need a stronger theory base on those topics. Izu et al. (2019) explored whether students who have completed an introductory programming course will know which direction to use when creating an iterative code, and whether the CS1 explored that concept, by performing a think-aloud study. The authors identified a lack of materials that explored the iteration direction as an explicit part of algorithm design. Most students chose to use upward loops unless the task explicitly stated otherwise, and those who used downward loops did it incorrectly. Taken together, this work shows a need for early interventions on misconceptions to head off future problems.

In contrast to work that focused on conceptual and code misunderstanding, Kolikant and Mussai (2008) focused on students' understanding of algorithmic output and debugging. More specifically, they explored whether students would detect errors in code and understand how their score relates to students' understanding. They gave 159 students a questionnaire covering six algorithmic problems, only two of which were correct. The others had logic and secondary errors. Only 26% of students tested the program to find errors. The majority of students said they "tested it in their head". In interviewing the students, the authors found that most did not consider extensive testing worthwhile, as many were focused on the grade. This led to the study of how our students use Test-Driven Development (TDD) strategies as part of RQ1. They also found that many students held the belief that code could be "partly right" because they received partial credit for code that ran, even if the algorithm was wrong. Thus, the students had internalized an extrinsic evaluative model that limited their effort and understanding.

The primary mechanism by which students receive feedback while working is the compiler. To that end, some researchers have focused on studying when and how students use compiler messages and the challenges they face with them. Karvelas et al. (2020), for example, explored how changes in the way compile error messages are displayed affect how students respond to them. They compared two versions of the BlueJ Java compiler: one that compiles code on command and shows one error at a time; and the other that compiles code automatically with every code change and shows all errors. With more compiling error messages, the number of compilations per hour decreased, while the percentage of successful compilations increased. Jadud (2005), similarly, extracted information from BlueJ in order to understand student behavior around compiling messages and what types of errors generated compilation error messages. As expected, the most common errors on compilation were syntax errors (58%) with the remainder being distributed among other types. They also found that 51% of all compilation events

happened within 30 seconds of one another, and roughly 20% was separated by more than two minutes. In general, they found that students prioritized solving bugs through short fixes and recompilation, with students working for extended periods when they had no compiling errors and then making lots of small changes and compilations until they could clear the errors. While this work provides no clear guidance on how students should use the compiler, it makes clear that more research is needed on whether interventions geared towards compilation use, or even adaptive compilers can support learning.

In light of the importance of these messages, other researchers have studied whether they can be used to provide guidance. One such approach was taken by Gusukuma et al. (2018) who replaced existing compiler messages with ones written by the instructors to target the root misconceptions across four categories: runtime, syntax, logical correctness, and generic. They evaluated their approach by comparing performance on quizzes and programming problems when using regular automated feedback with when using MDF. While they found an improvement in student performance, it was not statistically significant. Becker et al. (2016), likewise, noted that compiling error messages are essential for feedback, but they can make it more difficult to solve bugs if badly written/formatted. The authors tested enhanced compiler messages on an IDE they developed specifically for their research. The enhanced messages contained an explanation of the possible error and a suggestion of how to fix it. They attempted to understand whether enhanced messages could reduce the overall number of errors, number of errors per student, and incidence of repeated errors, and whether students found the enhanced messages beneficial. As a result, the experimental group presented a significantly reduced number of errors overall and errors per student. When answering a final survey, students found enhanced messages to be less frustrating and less of a barrier. As this work makes clear, the compiler messages provide an ideal opportunity to scaffold good practice by providing student-focused feedback, thus blurring the line between a tool and a tutor. This, however, could raise other concerns about the transfer of knowledge once students leave the supportive space.

Debugging in response to compiler or runtime errors is a fundamental skill in programming. To be effective programmers students must learn not only to read and process error messages but to localize and diagnose the causes of errors. Gaber and Kirsh (2018) conducted a study of student problem-solvers to examine whether self-reporting coding issues after submission would support better programming comprehension. To do this, they added self-reporting and attitude surveys to three assignments in a CS1 course. They reported a significant decrease in bugs and missing features in the code. However, they also noted that the fact that they gave students points for self-reporting the issues may have changed their normal behavior. It is also worth noting that, in contrast to many courses, the instructors gave a higher priority for code quality than other factors.

Fitzgerald et al. (2008) studied novice programmers' debugging quality by watching students attempt to debug and fix a buggy program, which was followed by student's self-evaluations. On average, students found 70% and fixed 68% of the bugs, and spent approximately six times longer on finding bugs than on fixing them. Interestingly, when they compared debugging scores to self-assessments, they found that students were generally inaccurate as the better performing students downplayed their debugging skills. They also found that debugging skills did not mirror other programming skills. Better-performing students were not always good at debugging, while the poorer performing students were consistently bad debuggers. A similar phenomenon was noted by Wrenn and Krishnamurthi (2019) who found that asking students to write their own test cases was not always an effective mechanism to assess problem comprehension or improve code quality as students tended to propagate fundamental misconceptions to their tests, thus making them uninformative for debugging. These results highlight the importance of teaching good debugging skills explicitly and not relying on other related tasks such as testing to teach these skills.

Each of the CS conceptual challenges discussed here — misconceptions, feedback, and debugging — led to a portion of the exploration done through my research work and contributed to the topics explored through our research questions.

1.2.3 Student Effort, Progress, Performance, Coding Behavior, and Struggle

"Effort" is largely used as a broad term, which is partially the reason we explore effort in our research.

Edwards and Li (2016) tried to understand where students apply their effort by creating *progress indicators*, which relate to code writing. Half of their indicators correspond to writing code for their solution, while the other half corresponds to writing code to test their solution. As a result, their students applied a large amount of effort to writing tests, but at the same time, students did not use those tests to increase the correctness of their solution. The same authors then apply their indicators on historical CS2 course data in addition to using a student performance predictive model (Li and Edwards 2018). They achieved high accuracy in the prediction of which indicators would be triggered. Some of the typically triggered indicators were "reducing average method size", increase in statement coverage", and "increase in conditional coverage".

To understand how students apply their effort when learning programming, it is crucial to understand their coding behavior. According to Kinnunen and Simon (2012), programming assignments take students as much time as all their other courses combined in their current course. They attempt to explore students' experiences when completing assignments to help

with introductory course retention. The authors identify 6 stages of doing a programming assignment and provide a descriptive model of "how CS majors build their self-efficacy perceptions". The six stages were getting started, encountering difficulties, dealing with difficulties, submitting, and stopping (not necessarily in this order). Each stage has a series of phenomena and emotions related to it, from despair of being lost to relief and happiness of being done with the assignment. Emotional reactions were a theme the students kept mentioning during the first interviews, and emotions were essential in the theory of how students "experience the process of doing programming assignments and assess their programming related self-efficacy". Students expressed, explicitly or implicitly, that their feelings about the experience of doing an assignment impacted their potential for success in the course. Azcona and Smeaton (2017) created a predictive model through machine learning that identifies at-risk students by using engagement and effort predictors. As "effort and engagement" predictors, they use interactions from the students in a Virtual Learning Environment and use those to predict laboratory exam grades. As a result, their model helped identify which students would likely pass the next laboratory exam. Although the previous studies aim to understand student effort, their definition of effort revolves around whether the student made progress.

Students' progress and success when solving programming problems depends on self-regulation and knowing where they are in the problem-solving process (Bergin et al. 2005; Prather et al. 2018). Applying self-regulation skills might lead to a decrease in procrastination and an increase in student success (Bergin et al. 2005; Tuckman 2005).

Also attempting to understand progress in programming courses, (Kazerouni et al. 2017b) developed four metrics to assess student progress: working early and often, writing test cases, launching test cases, and launching the program manually. They investigate which metrics are significantly related to project success, which metrics are significantly related to finishing solutions on time, and which metrics are significantly related to how much time is spent working. Data came from collection on projects from three sections of CS3 Data Structures and Algorithms course at Virginia Tech. They found that students who started working on projects earlier were more successful and finished earlier, with no significant difference in total work time. Working longer on projects did not show a significant improvement in grades. The results in our effort exploration, later described, are in agreement with their previous work (Kazerouni et al. 2017a).

While it is difficult to make every single student feel comfortable – especially with as many students as some Computer Science introductory courses have – we believe it is important to know how assignments impact students. Many of them might feel too overwhelmed to reach out for help. Coordinating schedules with professors of other courses, letting students know they can talk to their instructors, and giving them access to mental-health related resources

are a few ways to help them. One of the goals of our feedback and project milestone structure is to reduce student procrastination. We use duration metrics to compare the two feedback strategies. We also consider commits and when students reach particular milestones as they develop their projects.

Two groups of authors attempted to understand the differences in student programming behaviors in Java when comparing low, average, and high-achieving students.

Rodrigo et al. (Rodrigo et al. 2013) explored the students' error quotient (Jadud 2006) (student's capacity to cope with syntax errors), confusion rate (LEE 2011) (function of compilations with errors, time between them, and number of pairs of compilations with the same errors), self-reports on difficulty to learn Computer Science, and what errors/misconceptions each group is more prone to make. The authors also used midterm grades in their analysis. Interestingly, debugging and common Java errors were explicitly taught to students. Dividing low, average, and high-achiever by midterm grade, there was a significant difference both between grades and between the self-perception of average-high vs low-achievers.

Jegede et al. (Jegede et al. 2019) explored the difference in types of errors between low, average, and high-achieving students, grouping them by grades. Student's code from exams was analyzed and errors were classified into invalid symbols, mismatched symbols, inappropriate naming, and excessive symbols. Most errors were related to missing symbols, fewer were related to invalid symbols. Missing symbols were also the most common error by student category, while invalid symbols were the least often both in low and high-achieving students. Most errors of high-achieving students were in methods and classes, while for low-achieving it was other object concepts. Correlation between errors was found: missing symbol was related to invalid symbol, inappropriate naming, and mismatch symbol; loops and decision-making, and other object concepts; other object concepts and methods and classes. Students also participated in an interview and answered why they thought they made the mistakes. Students said, among other responses, that "convention does not matter as long as the code can run", "IDE for practice gave hints the editor for [the] exam did not", and "programming is like math with precise formula and memorized steps". Methods and classes were the concepts with the most errors, which led to issues in object-oriented programming. The authors concluded that courses should be taught based on achievement level and use unintelligent programming editors.

1.2.4 Discussion

As we can see from our survey, students' metacognition skills, as well as problem-solving strategies, influence how they learn and perceive several programming concepts. Although it is

difficult to understand whether, or which, cognitive issues are a cause of student struggle while programming, the effects of these issues have been observed by instructors. Instructors know their students enough to understand whether their issues are related to misunderstanding concepts or something else. Teaching or giving students resources on the necessary metacognitive skills, like how to break down programming problems into smaller pieces, at the beginning of the semester could improve their chances of succeeding in the course by providing them with adequate interventions to deal with their struggle. As students have different educational backgrounds, they will have different levels of cognitive skills when entering an introductory course. Our exploration of how previous computing experience affects coding struggle, since the available studies generally focus on CS1, is a starting point in understanding how much impact these skills have in a CS2 course.

With the growing enrollments in Computer Science programs (Zweben and Bizot 2022; National Academies of Sciences, Engineering, and Medicine 2018), in addition to the challenges discussed through this literature review, identifying struggling students early is important for supporting student success. Teaching staff will help students synchronously during office hours (Smith et al. 2017; Gao et al. 2022a) and asynchronously via message boards (Vellukunnel et al. 2017). Students may also seek help from peers, online resources, or through tutorial centers (Doebling and Kazerouni 2021). However, some students never seek help either because they do not need help or because of challenges with attending office hours, wait times, or approachability of the teaching staff member (Doebling and Kazerouni 2021). Although there are studies on coding and programming struggle (Lowe 2019; Yarmish and Kopec 2007; McCall and Kölling 2019), as explained previously they focus on CS1 courses and small assignments. The difficulty to pinpoint whether students are facing challenges due to metacognitive issues led us to the path of identifying how struggle is demonstrated through coding behavior based on conceptual CS challenges that can be identified, in hopes to offer instructors with information that will allow them to offer earlier interventions to offset the lack of metacognitive skills presented by novice programmers. Specifically, the lack of studies on student challenges on CS2 courses led us to explore the challenges students face when coding for large, project-based CS2 courses.

Our research ideas started by exploring why students were struggling as a way to understand how to decrease student struggle. That led us to the exploration of cognitive issues and skills in novice programmers, which led us to part of our third research question, along with experiences while being instructors. The exploration, in turn, led us to search for tangible ideas of struggle that could be used in the classroom, with specifics metrics that could indicate where and how students were struggling when learning programming. That led us to our exploration of struggle, which is the center of this dissertation, and our first research question. On the road

from cognitive issues to the exploration of struggle, we notice during office hours that students that applied a large amount of effort during assignments were not necessarily seeing the expected results on their programming, as they struggled with persistent errors after trying to fix them for a long time. That led us to the exploration of effort and progress, which compose the foundational research piece of this dissertation, and to the metrics collected from submission data, which eventually formed the metrics used to answer our research questions.

1.3 Definitions

The following terms are used often through this document:

- Automated Acceptance Testing (AAT): a set of tests that runs automatically to assess whether a particular implementation follows the guidelines.
- Teaching Staff (TS): teaching staff is comprised of instructors and teaching assistants. The term is used to differentiate tests written by instructors from those written by students.
- System tests: global tests that assess the full functionality of the students' implementation. System tests were written by the instructors who created the project.
- Unit tests: Java tests that execute each method developed by the student. Unit Tests were written by the instructors who created the project.
- Source code/Solution code: Code from to implementation of a programming assignment solution.
- Test code: Code from tests used to assess the implementation of an assignment solution. Student tests are developed by students, and TS tests are developed by the teaching staff.
- Commit: a commit is a partial implementation of a programming project submitted by a student. The student's last commit is their final submission.

CHAPTER

2

FOUNDATIONAL RESEARCH

2.1 Course Context

Given that CS2 courses are less studied than CS1 courses, we explore submission data from a CS2 course. All data used in this research comes from submissions of the same course, "Software Development Fundamentals". The course covers advanced object-oriented programming, the software engineering life-cycle, software development practices and skills, finite-state machines, linear data structures (array lists, linked lists, stacks, queues, iterators), and recursion. It uses Java as the programming language of instruction to demonstrate implementation of the key learning concepts. The course is required for computer science majors and computer programming minors. The programming assignments considered in this research vary each semester, and are graded in part by using Automated Acceptance Testing (AAT) Prather et al. (2018).

2.1.1 Subjects

The data comprised in this research comes from 10 different semesters, spanning from 2019 to 2024. On average, there were approximately 263 students enrolled in each Spring semester, and approximately 325 students in each Fall semester. Fall 2021 had the largest enrollment with 370 students. In total, there were 2944 students enrolled, from which 1065 (36.2%) consented to participate in the research studies involved in this dissertation. There were two on-campus sections and one distance education (online asynchronous) section each semester. Our research studies are covered by IRB approval from our institution and only data from consenting students is used. A summary of demographics on the students from the relevant semesters can be seen in Tables 2.1 and 2.2.

Table 2.1: Students demographics from Spring semesters, excluding Spring of 2018.

	# Students	213	254	274	272	302
Category	**Response**	**S20%**	**S21%**	**S22%**	**S23%**	**S24%**
1st Attempt	Yes	86	86	78	79	82
	No	14	14	22	21	18
Gender	Male	73	76	75	76	70
	Female	24	20	18	19	23
Ethnicity	Non-Hispanic White	55	53	54	45	43
	Asian or Asian American	28	24	28	37	31
	Black or African American	3	5	6	4	6
	Other/Multiple	14	18	12	15	20
Standing	Freshman	31	27	22	28	25
	Sophomore	41	38	43	36	30
	Junior	22	18	23	20	30
	Senior	2	3	3	6	4
	Other	1	13	7	7	8

Table 2.2: Students demographics from Fall semesters.

	# Students	271	287	370	351	350
Category	**Response**	**F19%**	**F20%**	**F21%**	**F22%**	**F23%**
1st Attempt	Yes	92	93	91	86	88
	No	8	7	9	14	12
Gender	Male	79	73	73	73	68
	Female	18	24	23	24	24
Ethnicity	Non-Hispanic White	59	53	40	48	38
	Asian or Asian American	22	31	17	33	35
	Black or African American	5	4	4	5	6
	Other/Multiple	15	13	39	14	21
Standing	Freshman	5	5	3	5	4
	Sophomore	60	56	63	49	56
	Junior	21	23	22	32	27
	Senior	2	6	2	4	4
	Other	0	10	9	8	5

2.1.2 Course Assignments

The main assignments in the course are programming projects, which are described below. Besides projects, students are assessed with three exams (two midterms and one final exam), which are beyond the scope of this research.

Guided Projects

The first three assignments are small projects, called Guided Projects, which have the goal of getting the students familiar with combining different programming concepts into a project, and with how to use the tools necessary for development in the course. For the programming tasks that must be completed in the Guided Projects, students receive specific instructions that guide them step-by-step to the completion of the exercise.

Development Projects

In each semester, students also complete two or three large development projects. In the F18 semester, there were three 3-week development projects focused on object-oriented concepts (P0), finite state machines (P1), and linear data structures (P2); students had to complete P0 individually, while P1 could be complete in optional pairs, and P2 needed to be completed in mandatory pairs. In the remaining semesters, the projects were changed to two 4-week projects that focused on finite state machines (P1), and linear data structures (P2); students completed P1 individually, and P2 in optional pairs.

Development projects are broken into two parts: Part 1 requires students to propose a class-level design and black box test plans given a set of requirements. The requirements are composed by user cases of how the software being developed in the project would be used. The goal of Part 1 is for students to learn to design software based on clients specifications. Part 2 requires that students implement and test a design created by the teaching staff, which is given to the students in the form of a UML class diagram. This research focus on student code, which means we focus on the data from the second part of each project. All the projects had a built-in 48-hour late window, with an increasing penalty over the late period. We use the late deadline as the zero hour for our time calculations.

Since in this research we only explore development projects, we use the terms "development project" and "project" interchangeably.

2.1.3 Software Engineering Practices & Tools

Our CS2 course emphasizes software engineering practices and tools. Students are introduced to the Eclipse integrated development environment (IDE). We emphasize verification and validation of software through unit testing (using JUnit), coverage (using EclEmma and Jacoco), and static analysis tools (CheckStyle, PMD, and SpotBugs). Course tools are also described on Heckman and King (2018). Students submit versions of their implementation to an enterprise GitHub system. GitHub is a version control system in which each student or student-pair had a repository - a storage unit to store code files - specific for the project they were working on. The continuous integration system (Jenkins) will pull student code and build it to provide feedback to students through Automated Acceptance Testing (AAT). Automated feedback supports student metacognitive skills and can scale to large classes, where instructors do not have enough resources or time to give personal feedback to every single student. To generate feedback, Jenkins executes unit tests created by the teaching-staff against each version of the student implementation, and the feedback is based on the unit tests that failed. Feedback strategies for teaching staff test failures varied by the semesters studied, as explained in the next section.

Each version of the student implementation submitted to GitHub is called a commit. A commit contains the student's code and a record of how the code was changed based on the previous commit. Each commit also contains a message ("commit message") written by the student, that should explain what changes were made in the code. After each commit, we utilized a continuous integration system, Jenkins, to automatically grade student work and provide feedback. The automated grader completes the following steps: 1) compiles the student's source and test code, 2) compiles the teaching staff test code, 3) runs static analysis tools, 4) runs the student's test code instrumented for coverage, 5) given sufficient coverage, and runs the teaching staff tests. A failure at any of the steps will stop the build. The system provides feedback on compilation errors, static analysis notifications, test failures (students and teaching staff), and code coverage. Over 60% of the project's grade is pulled from the automated grading system. The remaining portion of the grade is derived from acceptance tests, student system tests, and the contents of their Javadoc.

2.1.4 Projects and Automated Feedback

Each project has two milestones intended to help students make progress towards project completion, and one milestone that indicates completion of the automated grading portion of the project. Except for the F18 semester, the first two milestones have an associated *Process Points* deadline with associated points that are part of the total project grade.

Milestone 1 requires students to complete a compiling code skeleton, including at least one test class, given a UML class diagram for the project. A compiling skeleton would be recognized by a yellow ball on the Jenkins continuous integration system. Additionally, students were expected to fully Javadoc their classes and methods. Students earned 5 points on the project for meeting the compilation expectations and 5 points on the project for commenting their code well enough to avoid CheckStyle notifications by *Process Points Deadline 1*.

Milestone 2 requires students to meet the expectations for one of the feedback strategies. Feedback was generated for students when the unit tests written by them achieved a certain coverage threshold for all non-UI classes in the project.

- In the **layered feedback** strategy, when the coverage threshold is achieved students see feedback for *all* teaching staff test cases related to the whole project. Students would meet Milestone 2 once the coverage threshold was achieved.

- In the **incremental feedback** strategy, students received feedback on a class-by-class basis. Once a student achieved the coverage threshold for a single class, they see feedback about teaching staff tests cases for that *single class*. Students would meet Milestone 2 expectations when they had sufficient statement coverage, and were passing teaching staff tests, for roughly half the classes in the project, as determined by the teaching staff.

Strategies were used with different coverage thresholds in each semester. Between Spring of 2018 and Fall of 2018, and also for Project 1 of Spring of 2019, we used the *layered feedback* with a coverage threshold of 80%. From Project 2 in the Spring of 2019 to the Spring of 2020, the coverage threshold was also 80%, but we used the *incremental feedback* strategy. Starting in Fall of 2020, we returned to using the *layered feedback*, but the coverage threshold was lower to 60%.

In Fall 2018, Milestone 2 did not have an associated deadline where students could earn points for achieving the goal. In Fall 2019 and Fall 2020, we added the *Process Points Deadline 2* for students to meet Milestone 2. Students earned 10 points for meeting the expectations for Milestone 2 by the deadline. Partial credit was given for work towards the Milestone 2 goals.

Milestone 3, which signifies project completion, required students to pass their own tests with at least 80% statement coverage, pass all the teaching staff unit tests, and have no static analysis notifications. Completing Milestone 3 would lead to a green ball on Jenkins and represents the *done criteria* for the portion of the project that can be automatically evaluated using an Automated Acceptance Testing (AAT). There are several other tasks that students complete for manual evaluation including, run their system tests and recording the results of execution and generating Javadoc. Those changes are outside the scope of this research.

2.1.5 Help-Seeking

Help-seeking resources in the course we are exploring comes in two different main formats. One format is office-hours. Each teaching staff member completes two-hours of office hours per week. With large classes, office-hours are often full close to deadlines. A better description of how students use our office-hours can be seen in Gao et al. (2022b). To keep track of if and when students attend office hours, as well as what are the topics of their questions, we use a system called MyDigitalHand (MDH) Smith et al. (2017). Students are enrolled in the MDH system through a course ID, meaning if students are enrolled in two different courses that use MDH, their records will be kept separate. For each office-hours session, students log into MDH and "raise their hand", which alerts the teaching staff leading the session that a student is waiting for help. When students raise their hand in the system, they also input what assignment they need help with and what their questions are. MDH creates a line of students, ordered by the time students raised their hand. The teaching staff will then call each student by order, and at the end of the interaction the staff can input into MDH what was done in the interaction and whether the problem was solved or what the necessary follow-up is. MDH records the times the interaction started and ended, and the students' descriptions of their questions.

The second format of help-seeking in our course is through an online forum set-up on Piazza, which is an online forum widely used in higher education. Students are also registered on Piazza using a unique code per course. On Piazza, students can write questions virtually at any time. For each question, they can select which assignment their question relates to. Once a question is posted, both teaching staff members and students can write a response. Those two responses are viewed by students separately, and teaching staff members can validate students' answers. Once a question is responded, students have the opportunity to add follow-up questions. A question can be marked as resolved by teaching staff members. Also, at least one teaching staff member has the responsibility of monitoring the forum at given times through during business hours. Piazza records the time and contents of each post, as well as whether that post was an original question, an answer, or a follow-up question.

2.1.6 Data Description and Collection

Our data is composed by submission data from the course's development projects. Currently, our data covers Spring and Fall semesters from 2018 to 2023. The distinction between Fall and Spring semesters is due to Spring semesters having lower enrollment rates, as it can be seen on 2.1. Overall, there were 2944 students enrolled within the semesters, from which 1065 agreed to participate in the studies that compose this dissertation.

For each commit submitted by a student, we collect code files, commit messages, and

time and date of commit, collect number of commits, differences in number of code lines between commits for both source code and test code, and unit test cases results from both student-written and TS test cases. We also collect all the progress indicators described in Edwards and Li (2016) and use them not to establish progress but to explore differences between partial submissions. In addition, we collect milestone metrics, also described in Section 2.3, and Chapters 3 and 4, which include completion rates and time distribution for when each milestone was achieved. All this data is collected per repository, meaning we have data for each project submitted in each of the semesters. When exploring how students attempt to fix failures in TS test cases, we focus on data between milestones M2 and M3 — which represent the period where students had access to TS test cases feedback.

We run post-hoc automated data collection on the projects associated with the repositories of students consenting to participate in our studies. Some projects may have been optional or required pairs; we only included repositories where both students consented to participate. Our process consisted of repository mining to obtain the build history for each repository. Then we collected metrics to help address our research questions. We collect data through two systems, CanaryMiner and JenkinsDataMiner.

We use the CanaryMiner data mining tool, which is part of the Canary Miner framework Heckman and King (2018), to gather data from the build history on analyzed repositories. CanaryMiner uses Jenkins, following the same build steps as described in Section 2.1.3, to build each commit, i.e. student implementation version, while removing any blocking actions to ensure full data collection. For example, during the semester, insufficient coverage would block the presentation of any teaching staff test failure, as described in 2.1.4. However, the TS tests are running on every build even if students do not have access to them. For our analysis, we collected TS test failure information even if there was insufficient coverage. By doing so, we can measure students' *actual* progress toward project solution rather than what the student may have observed during the instructional semester.

CanaryMiner is a combination of bash scripts and Java programs that support the historical repository mining, meaning it will build each commit from the most recent one to the first commit in a given repository. The process consisted of the following steps, after identifying the repositories of consenting students:

1. Clone the repository
2. Save the log file for the repository
3. For each commit in the history
    (a) Revert the repository to the earlier commit

(b) Build the project using the standard ant build script

(c) Save the output files for the build into a directory for that repository and commit

(d) Repeat for the next commit

The data is saved in a file structure, where each repository directory has subdirectories for the data of each commit. The first commit, made by the instructor when creating the repository, was not considered in the analysis.

The JenkinsDataMiner tool was created by Ericsson et al. Erickson et al. (2022). The tool works similarly to CanaryMiner, by building each commit and collecting the information resulting from the build. However, JenkinsDataMiner focus on collecting information in code changes between pairs of commits rather than raw submission data. It collects the number of lines changed for both source code files and test code files, date and time the changes were committed to the repository, the commit message written by the student, the number of methods, asserts, comment lines, and blank lines on each file. It also collects the number of TS test failures and student test failures, which lines were covered or missed by the students' unit tests, and the amount of static errors present in the implementation. It then calculates all progress indicators created by Edwards and Li (2016). All data is saved to a database.

2.2 Progress Exploration

2.2.1 Progress Indicators Study Replication

Our research started by exploring how students make progress in our programming assignments through progress indicators. Progress and/or performance indicators have been studied in programming courses (Alzubaidi 2017; Abdelhadi et al. 2022; Yang and Li 2018; Shana and Venkatachalam 2011; Veerasamy et al. 2021), including combining programming behavior and demographics (Azcona et al. 2019), and mining students' repositories (Mierle et al. 2005). However, the studies found do not include assessment of student code by automated tests, and most do not describe specifically which indicators were used, e.g. machine learning procedures select the indicators automatically. Therefore, we decided to use the indicators created by Edwards and Li (2016). Although their study uses smaller programming exercises, the structure of their assignments and grading are similar to ours, as their assignments also include both solution code and tests written by students. Their grading, like ours, is also partially done using Automated Acceptance Testing (AAT) tools. We perform a partial replication of their study, by following the same procedures to calculate each progress indicator, and to obtain a composite measure of indicators that can be used to determine whether a commit was productive. This

work is part of what led to RQ1.1, by exploring how to assess progress in CS courses. Differently than the original work, which focuses on understanding student effort, we use the progress indicators as a way to analyze coding behavior during struggle.

Given the multitude of challenges faced by novice programmers, Edwards and Li (2016) divide their indicators into two groups, which relate to source code development and test code development, respectively. Their indicators are described in their work as follows:

General Purpose Indicators:

1. Adding New Solution Method(s): the student's solution contains more methods than before.
2. Removing Static Analysis Errors: problems in the layout, formatting, commenting, or style of the code have been resolved.
3. Reducing Cyclomatic Complexity: the student's solution contains fewer branches and/or methods than before.
4. Reducing Average Method Size: the average number of statements per method has been reduced.
5. Increasing Comments Density: the proportion of comment lines, relative to the number of source code lines, has increased.
6. Increasing Solution Classes: the student's solution contains more classes than before.
7. Increasing Correctness: the number of instructor provided reference tests passed by the student's solution is more than before (the solution's behavioral correctness has increased).

Software Testing Indicators:

8. Adding New Test Method(s): the student's software test suite contains more tests than before, while the number of solution methods remains the same.
9. Adding to Existing Tests: the number of statements in the student's test suite is greater than before, while the solution remains approximately the same size.
10. Increasing Number of Tests per Method: the number of test methods increases, relative to the number of methods in the solution.

11. Increasing Statement Coverage: the student's own software tests execute a larger number of statements in the solution than before.

12. Increasing Method Coverage: the student's own software tests execute a larger number of methods in the solution than before.

13. Increasing Conditional Coverage: the student's own software tests execute a larger number of conditional branches in the solution than before.

14. Increasing Assertion Density: the student's own software test execute a larger number of assertions per methods in test than before.

15. Increasing Test Classes: the number of test classes increases in the solution than before.

Methods and Results

Data was collected using the CanaryMiner data mining system, as described in Section 2.1. Since compiling code is necessary to collect progress indicators, we exclude repositories with no compiling commits and consider only compiling commits in our calculations. We collect metrics for each semester, across assignments as was done by Edwards and Li (2016). We collect the following metrics for every compiling commit in each repository:

- Solution Methods: number of solution methods on the student source code

- Static Analysis: total number of static analysis errors, which are obtained from PMD, Spotbugs and Checkstyle plugins on Eclipse (if students run the plugins) and Jenkins (at each commit submission). Static analysis errors include errors from both source code and test code written by students. This indicator represents forward progress when its delta is negative.

- Cyclomatic Complexity: Number of decision points on source code plus 1, automatically calculated by CanaryMiner when checking code coverage.

- Average Method Size: average number of lines per method on source code.

- Comment Density: number of comments in the solution divided by number of lines in the solution

- Solution Classes: number of classes in the source code

- Failing Tests (Correctness): number of failing teaching and staff test cases. This indicator represents forward progress when its delta is negative.

- Test Methods: number of methods on students' test code.
- Test Lines: number of lines on students' test code.
- Tests per Method: number of tests divided by number of solution methods.
- Statement Coverage: percentage of solution statements covered by students' test cases.
- Method Coverage: percentage of solution methods covered by students' test cases.
- Branch (Conditional) Coverage: percentage of decision points on source code covered by students' test cases.
- Assertion Density: total number of asserts divided by total number of test methods written by students.
- Test Classes: number of test classes written by students.

Following Edwards and Li (2016) work, we first calculate indicators by getting the difference for each metric between pairs of consecutive compiling commits within a repository. Each indicator is a delta that shows whether there was progress for that specific metric. Progress in this context means either an increase or a reduction in the value of the indicator, depending on the indicator (e.g. an increase in statement coverage or a reduction of average method size).

After calculating indicators, we generate the proportion of commits that triggered each indicator, and average those proportions per semester. Proportions across semesters can be seen in Figure 2.1. Proportions for each Spring semester can be seen in Figure 2.2, and proportions for each Fall semester can be seen in Figure 2.3. We can see outliers in Fall 2020 and Fall 2019 for the increase in statement and conditional coverage, where those indicators were triggered at a higher rate when compared to the other semesters. Due to the structure of our projects, i.e. grades depend on correctness based on unit tests, the indicators triggered more often differ from the indicators in the original work. For example, in the original work *increase in correctness* is triggered in 7% of partial submissions, while in our work the same indicator is triggered, on average across semesters, in 32% of partial submissions (commits). Since both in both courses students had to write their own tests, *increase in statement coverage* is triggered often in both studies, being triggered in 25% of partial submissions in the original study, and in 34% of partial submissions in our study; the higher frequency of that indicator in our study is likely because students need to achieve a certain level of statement coverage in order to see feedback from teaching-staff (TS) unit tests.

We then evaluate whether the commits in the submissions were productive or not productive. Since several indicators can be triggered individually by adding poorly written code,

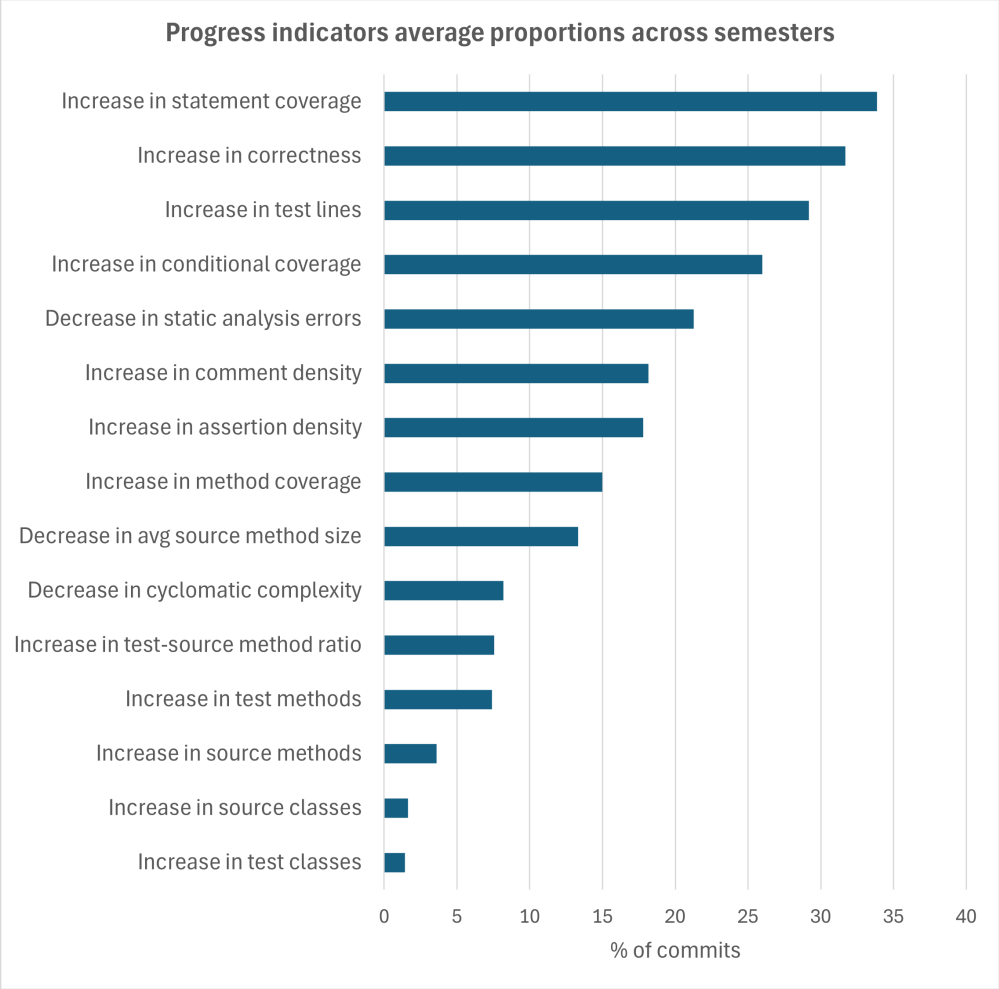


Figure 2.1: Proportion of commits triggering each indicator across semesters.

one indicator being triggered does not indicate progress. On the other hand, triggering a high number of indicators might indicate that students are making code modifications without focusing on a specific area of the project or without a goal in mind; additionally, some indicators contradict each other. Therefore, triggering "too many" indicators at a time might also not indicate progress. To find how many indicators would cause a commit to be considered productive, we calculate what Edwards and Li (2016) called a "composite measure", by plotting the percentage of commits that trigger at least K of indicators, from the commits that trigger at least zero indicators, to the proportion of commits that trigger all indicators. We follow the original work's methodology, which follows methods by O'Rourke et al. (2015, 2014). In Edwards and Li (2016), a productive commit has 5 to 8 indicators triggered when considering all indicators, or 3 to 4 indicators triggered when considering only general indicators. The composite measure for all indicators and general indicators can be seen in Figures 2.4 and 2.5,

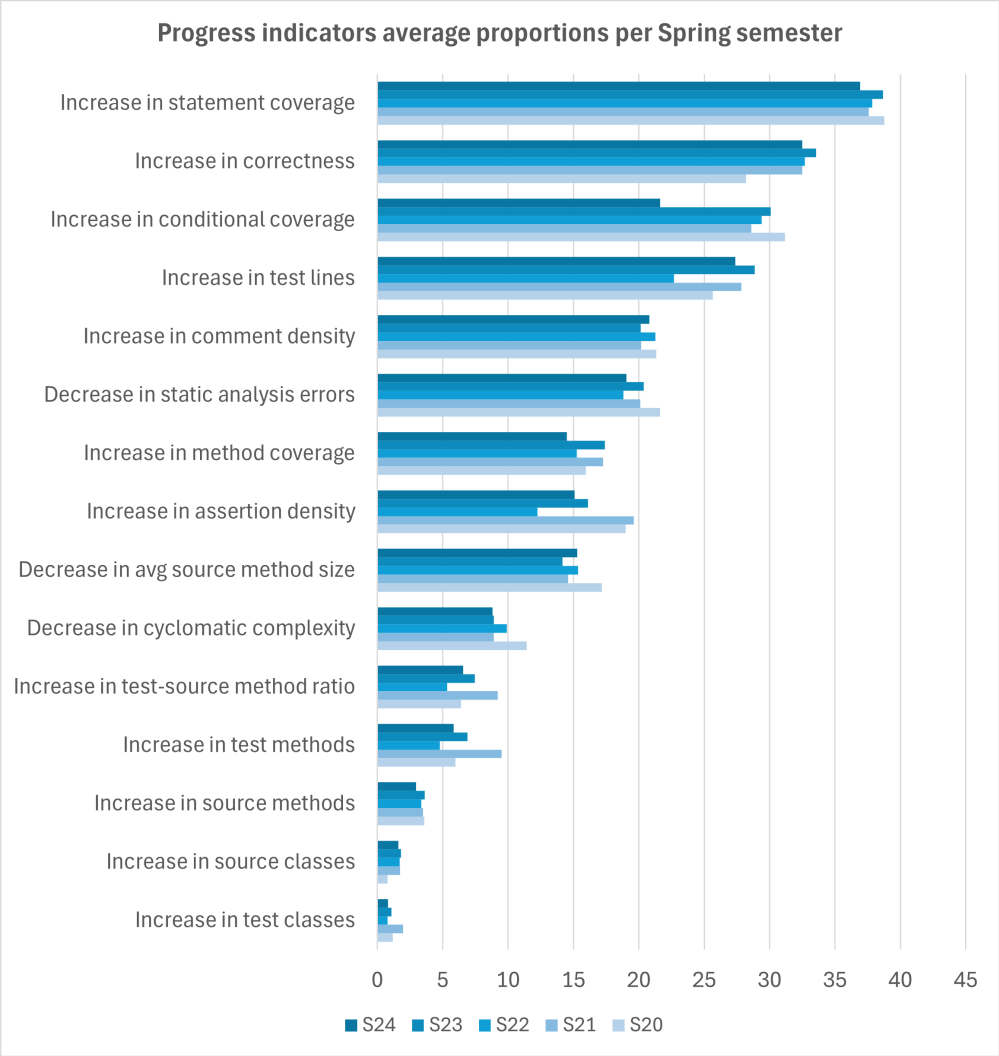


Figure 2.2: Proportion of commits triggering each indicator in Spring semesters.

respectively. As we can see in the figures, we find in our data that productive commits are the ones that trigger between 5 and 7 indicators when considering all of them, and between 2 and 3 indicators when considering only general purpose indicators. Both measures fit well with a polynomial function of degree 4 and are similar to the original work.

In addition to the original work, we also calculate the composite measure considering only test indicators, as test metrics are part of a large portion of our research. Composite measures for test indicators can be seen in Figure 2.6, and productive commits would trigger between 3 and 5 test indicators. Different than the other indicators, the composite measure for test indicators did not fit a polynomial function well. The trendline shown in the picture is a polynomial function of degree 2, and it was the closest fit.

We then calculate the proportion of compiling productive commits. Differently from the

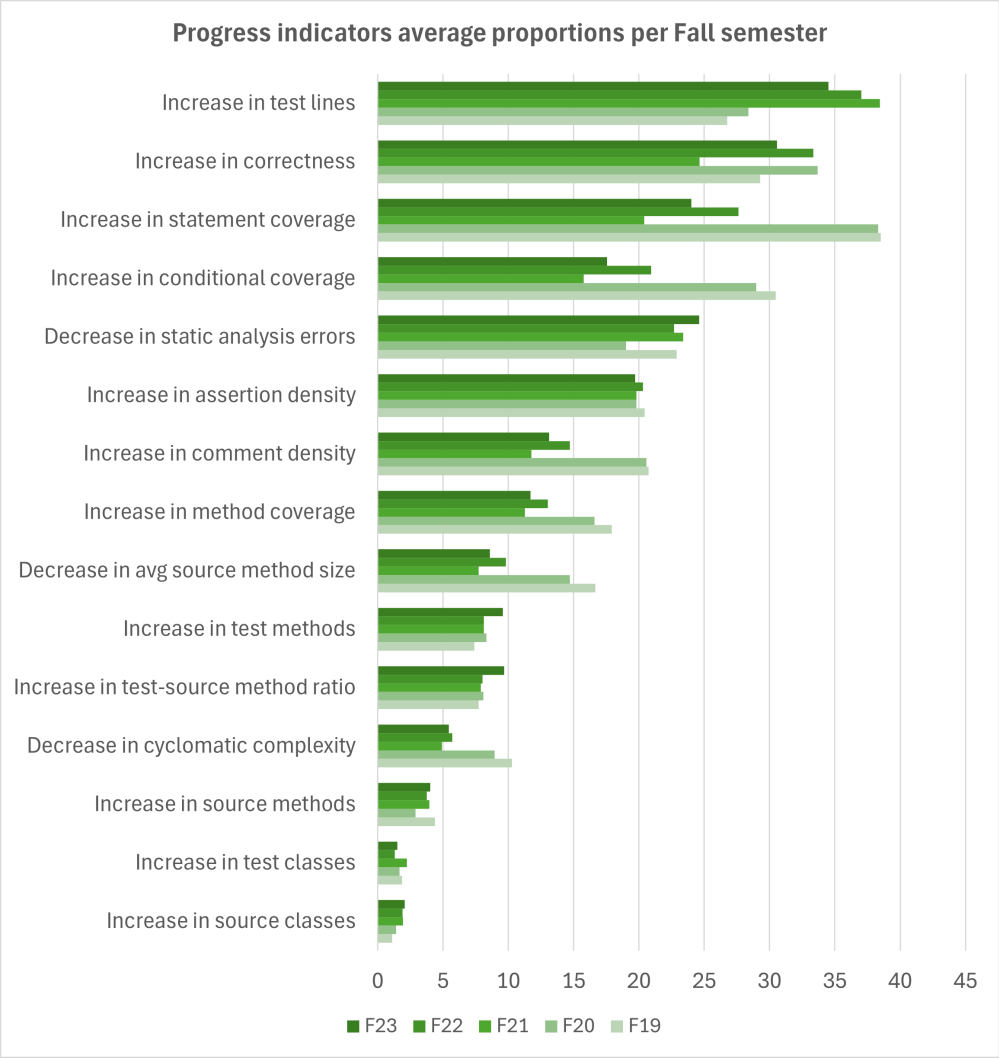


Figure 2.3: Proportion of commits triggering each indicator in Fall semesters.

original work, we use all indicators, since overall our research includes both coding and testing metrics. The average of those proportions for each semester can be seen in Table 2.3. In each semester, less than 20% of the commits were productive.

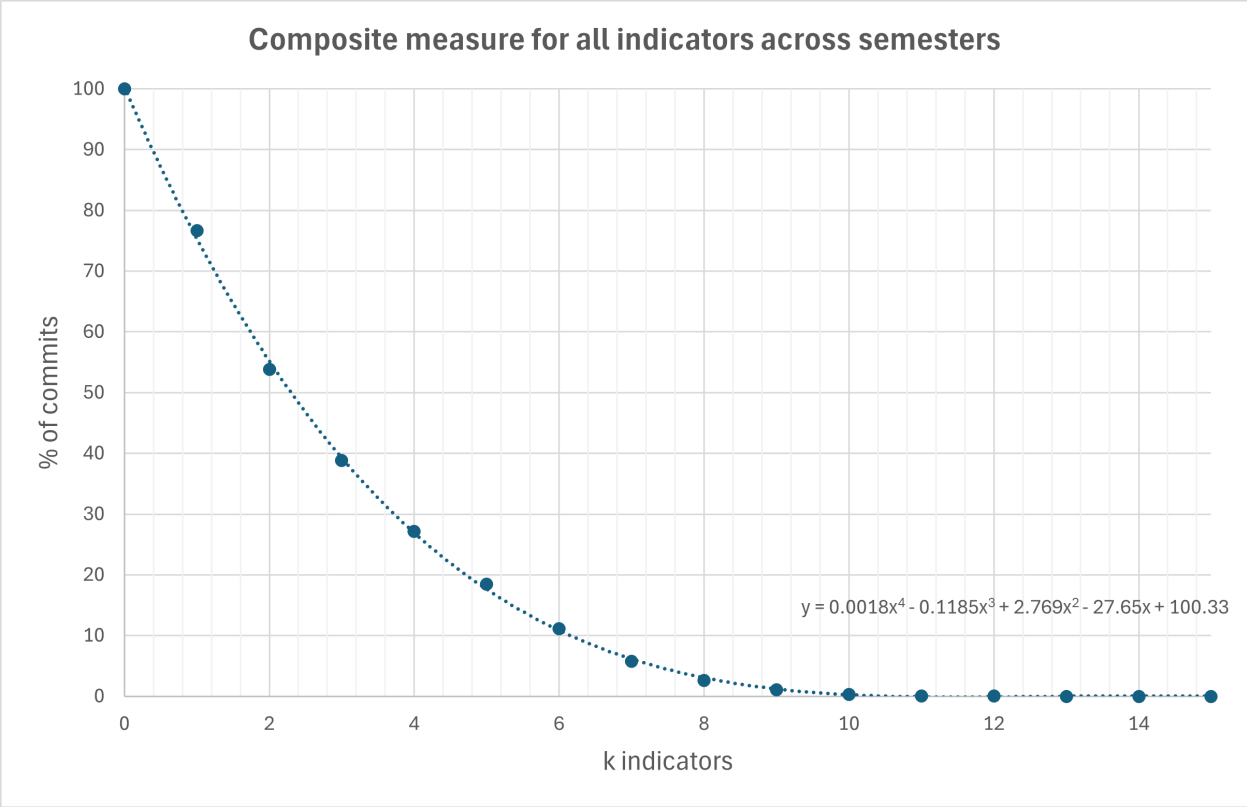


Figure 2.4: Composite measure for all indicators

Table 2.3: Average of compiling and productive commits per semester.

Semester	Commit count	% Compiling commits	% Productive commits
F19	127	82.6	16.7
S20	164	80.6	15.3
F20	94	78.2	16.0
S21	80	79.2	16.1
F21	74	75.6	11.5
S22	80	74.3	12.2
F22	72	78.0	12.0
S23	78	77.7	13.7
F23	71	76.3	10.1
S24	80	78.0	12.6

Additionally, we also calculate averages of compiling and productive commits by grade group, dividing project grades into quarters (i.e. 0-25%, 25-50%, 50-75%, and 75-100% groups),

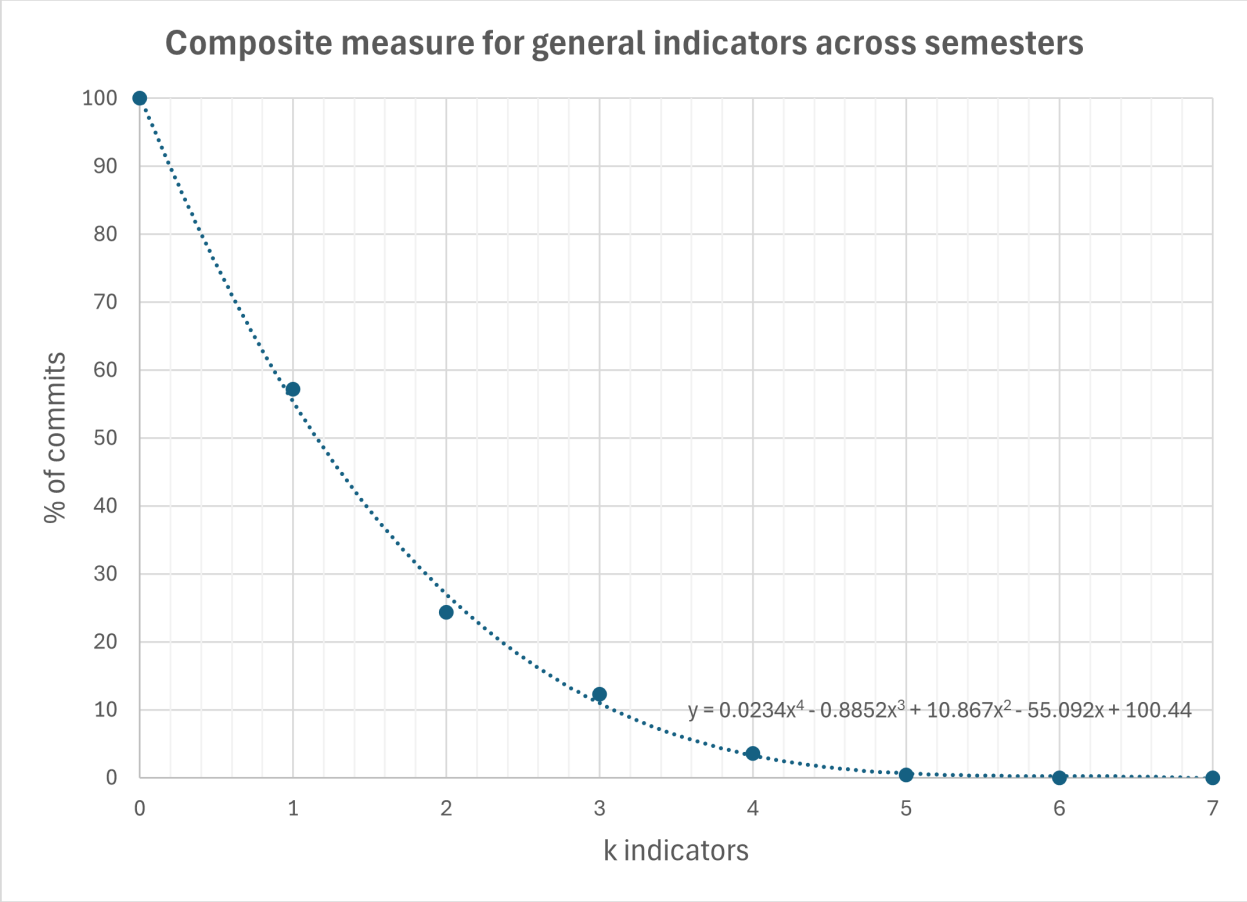


Figure 2.5: Composite measure for general indicators

which can be seen in Table 2.4.

Table 2.4: Averages of productivity and grades per grade group.

Grade Group	Commit count	% Compiling commits	% Productive commits	Avg. Grade
0-25	25	33.0	6.4	18.04
25-50	43	44.5	11.8	39.21
50-75	81	66.3	12.9	63.95
75-100	97	81.8	13.9	94.25

Discussion

The proportions of each indicator being triggered differ from the original work in our course context. Compared to the proportion shown in the original work, "Increasing Conditional

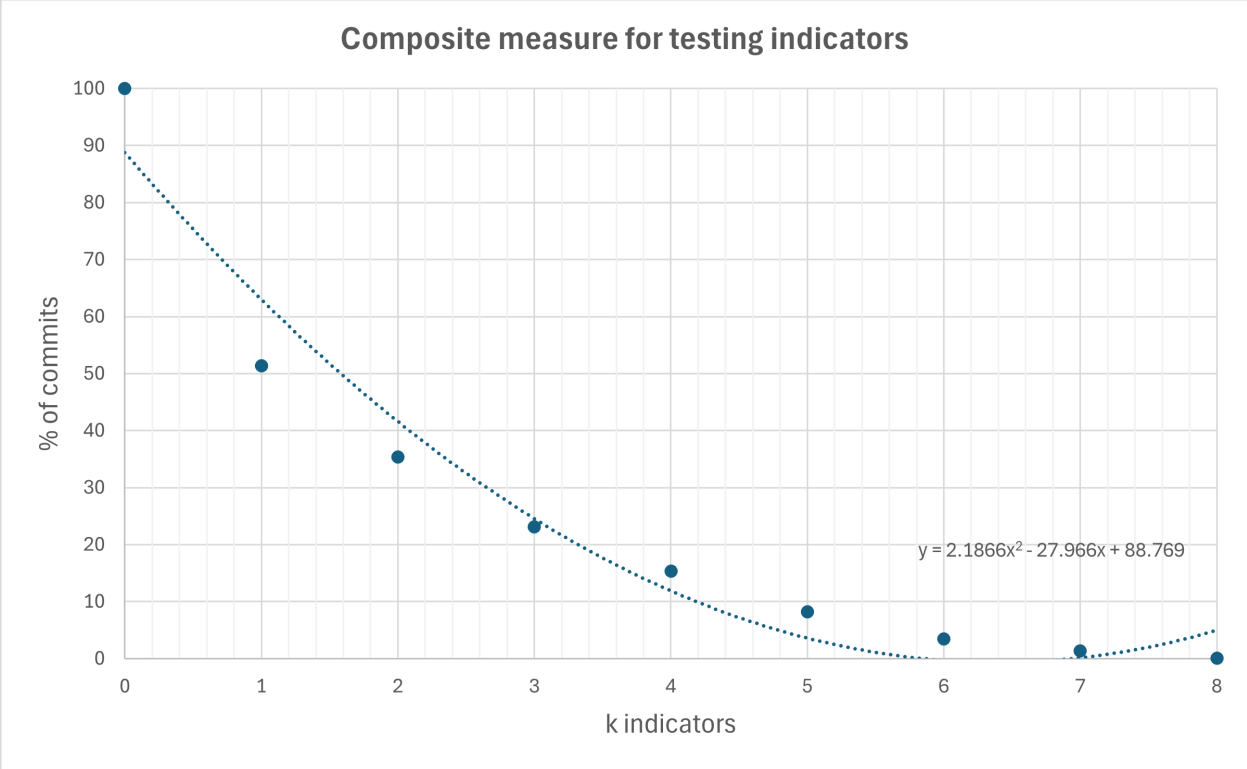


Figure 2.6: Composite measure for test indicators.

Coverage" is triggered in significantly fewer commits, which likely happens because the course's focus on statement coverage as a graded element. However, both of those occurrences align with our grading methods: students receive feedback from TS unit tests once they reach a certain statement coverage level, as explained on 2.1. An increase in test lines and an increase in correctness also align with our grading, since most of the project grades come from passing TS unit tests. Indicators that refer to the number of classes and methods are seldom triggered, since students have the UML diagrams of the required classes and methods. Those indicators are triggered at the beginning of the project when students are creating their code skeleton.

Productive commits represent less than a fifth of the total compiling commits, which indicates students might have difficulties making progress through the project. When indicators are divided by grade groups, students with grades in the 0-25 range generally did not complete the project. They do have fewer commits overall, while also having a lower proportion of compiling and productive commits. When looking into differences between the two higher grade groups, the proportion of productive commits is similar, and so is the overall commit count. However, the proportion of compiling commits is highly different between each group. That shows students in those groups are working at similar rates, but students with lower grades are struggling to create an implementation that functions.

These findings proportionate insights into the overarching research questions on this dissertation, as working at the same rate but achieving different results led to the need to define what students were struggling with. It also adds to our research the possibility of using the progress indicators that fit with our course structure to differentiate between productive and unproductive struggle.

Even though results were different between grade groups, this exploration led to finding out there is a ceiling effect on the project grades in our course. An example of project grade distribution, from Fall 2019, can be seen in Figure 2.7. The distributions from other semesters were similar. This finding led us to remove the use of grades when attempting to identify struggle.

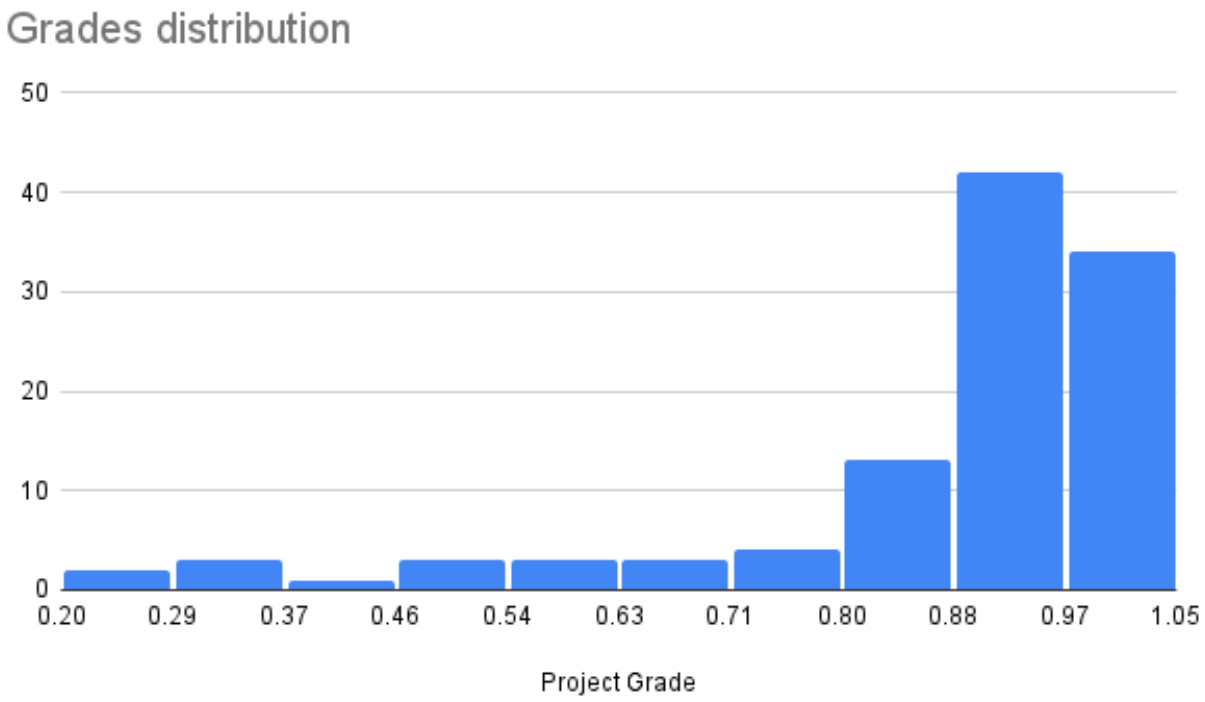


Figure 2.7: Project 1 Grades Distribution - Fall 2019

2.2.2 Characterization of Progress in our Course

In our course, project completion depends on the number of TS test case failures, so we define progress as a *decrease in TS test failures*. Based on the progress indicators study replication described in Edwards and Li (2016), we select coding metrics that are relevant for our course: changes in source code over project time and commits, changes in test code over project time and commits, project completion, and TS test failure rates.

In most semesters, over 80% of the students reach project completion. Students tend to pass around 75% of their test cases by the end of the project. On average, they modify approximately four code classes per commit, although over half of those are source code classes. As expected, progress tends to increase with each commit. However, a little over half of the students tend not to reach the milestones by the deadlines. On average, students needed between 3 and 4 commits to pass a given TS test case. Fall of 2021, however, presents outliers in the data. The project completion rate was around 34%, as opposed to over 80% on the other semesters, and only 1% of the students reached M2 before the deadline on Project 2. Fall of 2021 was when classes were returning to campus after the COVID-19 pandemic, which we believe is the reason for the decrease in those rates.

Students' progress generally increased over time, which was expected. A few commits per repository present regression in progress; however, that regression was generally due to errors in the expected output format, which needed to match the format in the TS test case exactly, and were corrected in the next commit. When analyzing progress over time, students performed several code changes over short spurts of time in the beginning and middle of projects, which are generally representative of students trying to reach M1 and M2. Between milestones M2 and M3, the students perform smaller code changes that focus on increasing progress to pass the TS tests. Examples of progress over commits and time can be seen along with source code changes in Figure 2.9, while examples of progress over commit and time along with test code changes can be seen in Figure 2.11. The figures also show coverage variation over the period of development.

Figure 2.8: Source code changes and progress over time. Commits that happened on the same day are shown on the stacked bars.

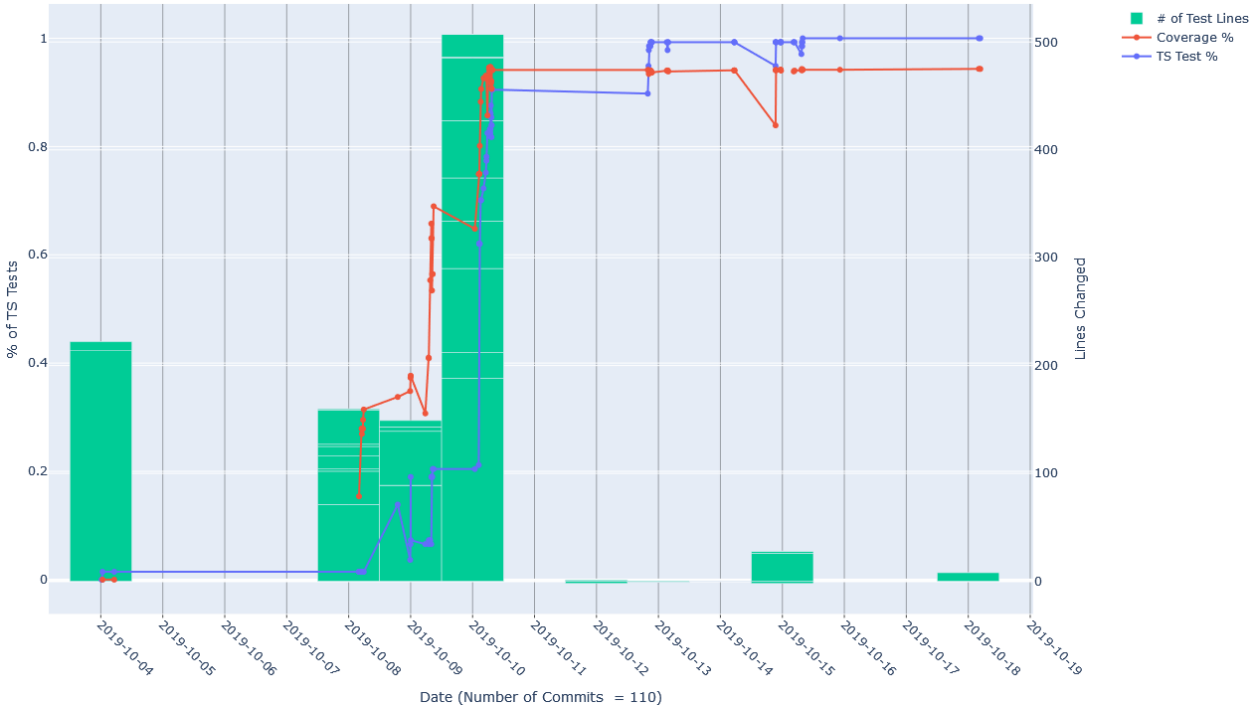


Figure 2.10: Test code changes and progress over time. Commits that happened on the same day are shown on the stacked bars.

Figure 2.9: Source code changes and progress over time per commit.

Figure 2.11: Test code changes and progress per commit.

2.3 Feedback Comparison Study

As discussed in Section 2.1, different Automated Feedback strategies were used throughout the semesters to give students feedback on their assignments: in the Fall 2018 semester, we used *layered feedback* with a coverage threshold of 80%; in the Fall of 2019, we used *incremental feedback* also with a coverage threshold of 80%. In the Fall of 2020, we returned to using *layered feedback*, but with a coverage threshold of 60%. This work led us to observe differences in milestone completion, which in turn led to the first part of the work done to answer RQ1. This work was published at SIGCSE Technical Symposium '24.

Automated Feedback is used because it can scale to large classes, where instructors do not have enough resources or time to give individual student feedback. Providing students with relevant, timely, feedback on their work, preferably before project deadlines, is one way of helping programming students successfully complete their assignments. Also, novice programmers are in the process of developing metacognitive skills to efficiently solve programming problems, and correct, timely feedback supports that skill development.

Our exploration of student coding using study sessions, described in Section 2.1, showed that students with lower grades also might alter class files simultaneously. Object-oriented programming involves a hierarchy structure between classes, which means there are dependencies between them. For a class to work properly, its dependencies must be developed first. Prior to this research, the instructor had observed students trying to fix errors that in the feedback appeared in one class but were actually caused by programming or logic mistakes made in another, typically dependent, class. Although the *incremental feedback* strategy was created specifically to increase scaffolding by guiding students to develop one class at a time, students were not required to develop classes in a certain order. As shown in **??**, some students were still developing different classes simultaneously. With the possible connection between feedback and the order in which students were working on development, we explore completion-related metrics across semesters with different feedback strategies.

To analyze the differences between semesters, we focus on completion of milestones. Each project, as described in Section 2.1, has three milestones (M1, M2, and M3). M1 refers to building a correct skeleton of the necessary classes, M2 refers to sufficiently testing their implementation, and M3 indicates project completion. We analyzed data from 1) the repository's commit history, 2) student test case coverage reports for each build; 3) the failing teaching staff test cases for each build, and 4) student grades.

2.3.1 Metrics

For feedback comparison, we only use data from Fall of 2018, Fall of 2019, and Fall of 2020. That is because those were the semesters where each type of feedback was implemented. We collected the following repository-level metrics: grade metrics, milestone metrics, deadline metrics, time metrics, and test failure metrics. The metrics are then aggregated by average across repositories and semesters, and used to calculate milestone completion rates on each project. The metrics are defined as follows:

Project average grades provide a measure of student performance on each of the development projects. The students had a period of 48 hours after the deadline to submit their implementations with a "late penalty". This metric is divided into two:

- *General grades*, which include the late penalty, represent the grades students received.
- *Late submission grades*, which are grades from late submissions with the late penalty removed, represent grades students would have received if they had submitted their implementations on time.

Deadline Metrics describe the number of commits before the Process Points 1 (PP1), Process Points 2 (PP2), Project Deadline, and Late Deadline. By comparing these metrics with Milestone metrics, we can determine if students met the Process Points Deadline. These metrics also show progress through the project in relation to deadlines. These values are cumulative from the first commit. We do not collect PP2 Deadline commits for Fall 2018 since there was no PP2 deadline that semester.

- **PP1 Deadline Commits**: number of commits made before the Process Points 1 deadline
- **PP2 Deadline Commits**: number of commits made before the Process Points 2 deadline (if applicable to the project)
- **Project Deadline Commits**: number of commits made before the Project deadline
- **Late Deadline Commits**: number of commits made before the Late deadline, which is the same as the total commits in the project

Time Metrics describe the number of hours before the late deadline when students started the project; reached Milestone 1, Milestone 2, and Milestone 3 criteria; and finished the project. Since the late deadline was the common stop point where repository access was removed, all time metrics are measured relative to the late deadline. If any of these metrics are equal to or less than 48 hours, that means that the student made a submission during the late window and incurred a penalty on their project grade.

Table 2.5: Project Summary Data

Semester	Project	# of Repos	Repos w/ Submissions	Avg. Commits	Avg. Duration (hours)	Proportion of Time Available
F18	P0 (OO)	274	263	55.7	228.3	73.17%
	P1 (FSM)	262	193	43.7	156.4	49.97%
	P2 (Linear DS)	151	120	112	288.9	92.60%
F19	P1 (FSM)	116	111	97.3	367.2	76.50%
	P2 (Linear DS)	116	93	142	340	70.83%
F20	P1 (FSM)	119	116	70.1	364.5	75.94%
	P2 (Linear DS)	71	65	83.7	213.7	44.43%

- **Start Time**: hours before the late deadline of the first commit
- **M1 Time**: hours before the late deadline of the commit where the student achieved Milestone 1 criteria
- **M2 Time**: hours before the late deadline of the commit where the student achieved Milestone 2 criteria
- **M3 Time**: hours before the late deadline of the commit where the student achieved Milestone 3 criteria
- **End Time**: hours before the late deadline of the last commit

Test Failures Metric summarizes the number of teaching staff test failures that remain in the last submission of the project.

We present results on every project. Even though F18 P0 was conceptually different from the other projects, we assume the concepts alone do not influence our metrics enough to exclude that project from the analysis. We characterize the projects in Table 2.5, which shows the total number of repositories, the number of repositories that had submissions, the average number of commits in each repository, the average duration (e.g., time between start time and end time, with the understanding that students were not constantly working during this time frame), and the proportion of average duration to the available time for easier comparison between semesters, due to differences in project length.

2.3.2 Milestone Achievement Rates

The average progress for each project in terms of milestones achievement is shown in Table 2.6. Most students tend to make quick progress toward M1, with fewer than 15 commits on average.

For project completion, students in semesters with *layered feedback* tended to reach M3 in fewer commits than the semester with *incremental feedback*. In Fall 2018, P0 had an achievement rate of 35.7%, P1 had an achievement rate of 72.0%, and P2 had an achievement rate of 53%. In Fall 2019 and Fall 2020, the achievement rates increased between P1 and P2; however, the achievement rate on P2 in Fall 2020 is not as high as in Fall 2019. This may suggest that *incremental feedback* leads to higher achievement. However, Fall 2020 was a pandemic semester, so the lower achievement rates may be for other reasons than just feedback strategy. Also, during the semester of Fall 2019, we were able to notice that the *incremental feedback* led students to try to pass every single test related to each class they were working on before moving to the next class. That caused a number of students to spend too much time in one single class, which was an unexpected outcome and led to the use of the modified *layered feedback* strategy during Fall 2020.

Figures 2.12 and 2.13 show the proportion of students who reached each milestone by its deadline for P1 and P2, respectively, across semesters. Figures 2.14 and 2.15 show the proportion of students who reached each milestone over the full course of the project.

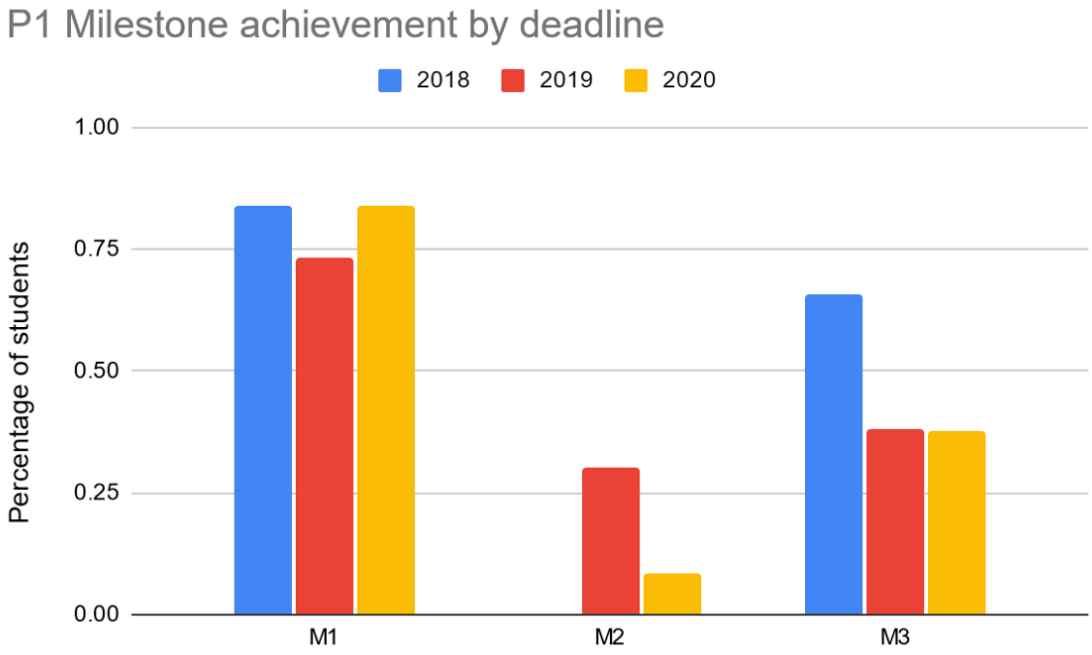


Figure 2.12: Proportion of students reaching milestones by the deadline on P1

Table 2.6: Milestone achievement summary

Semester	Project	Milestone	Achievement Rate	Avg. Commits	Time Before Late Deadline	Proportion Before Late Deadline
Fall 2018	P0	M1	97.30%	8.2	241.5	77.4%
		M2	74.10%	31.4	53.4	17.1%
		M3	35.70%	67.2	40.8	13.1%
	P1	M1	95.90%	9.5	264.6	84.5%
		M2	88.10%	31.3	89.9	28.7%
		M3	72.00%	61.7	78	24.9%
	P2	M1	83.20%	12.6	300.1	96.2%
		M2	74.10%	47.5	105.7	33.9%
		M3	53.20%	112.3	84.4	27.1%
Fall 2019	P1	M1	91.90%	11.4	379.1	79.0%
		M2	67.60%	72.3	160.7	33.5%
		M3	39.60%	107.8	77.6	16.2%
	P2	M1	98.90%	16.6	412.3	85.9%
		M2	79.60%	95.2	158.4	33.0%
		M3	64.50%	138.1	91.8	19.1%
Fall 2020	P1	M1	94.00%	8.0	399.9	83.3%
		M2	44.80%	59.2	111.2	23.2%
		M3	38.80%	77.9	69.3	14.4%
	P2	M1	95.40%	14.9	409	85.0%
		M2	75.40%	65.8	149.4	31.1%
		M3	53.90%	94.4	109.3	22.7%

2.3.3 Milestone Achievement Timing

We can see in Table 2.6 the number of hours that passed between reaching a milestone and the project's late deadline, along with the proportion those hours represent when related to the full time available for the project. Within every semester, students tend to reach each milestone earlier when comparing projects. This behavior likely means students learned how much work was necessary to complete the project and started earlier. Since the behavior spans across semesters, it is likely not related to the feedback change. The only exception is for M2 on Fall 2019. However, the difference in the time left once M2 was reached between Fall 19 projects is not large enough to conclude that the reaching M2 later on P2 was related to the feedback strategy change. We can also see that the differences in timing for each milestone when comparing P1 and P2 are much smaller during Fall 2019, which might indicate that the *incremental feedback* strategy supported students in developing a more stable work routine.

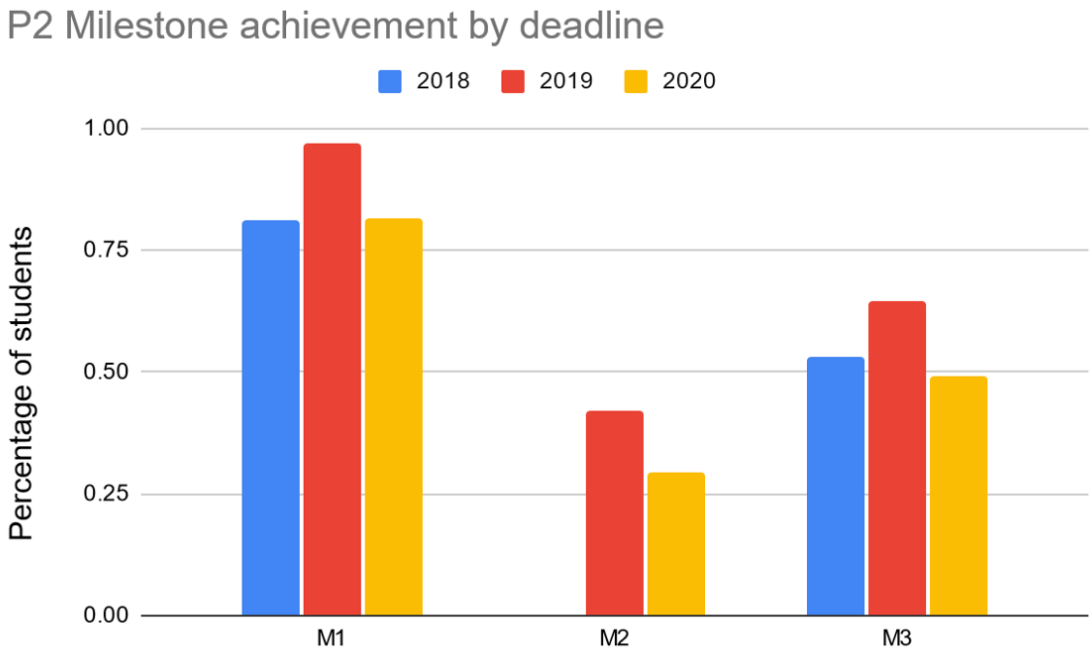


Figure 2.13: Proportion of students reaching milestones by the deadline on P2

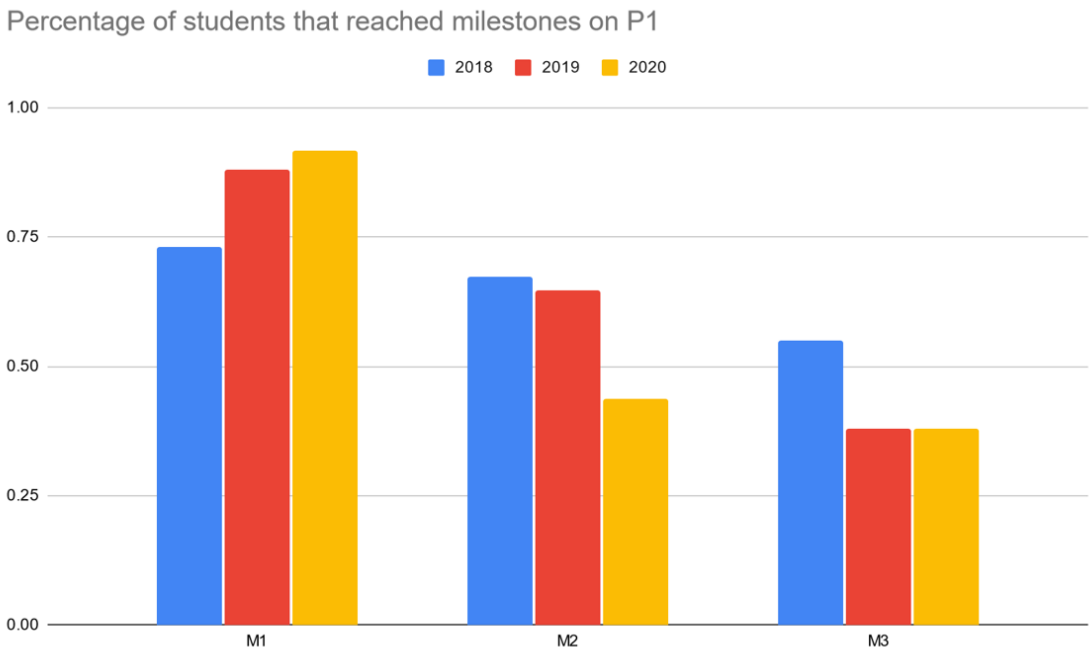


Figure 2.14: Proportion of students reaching milestones by the deadline on P1

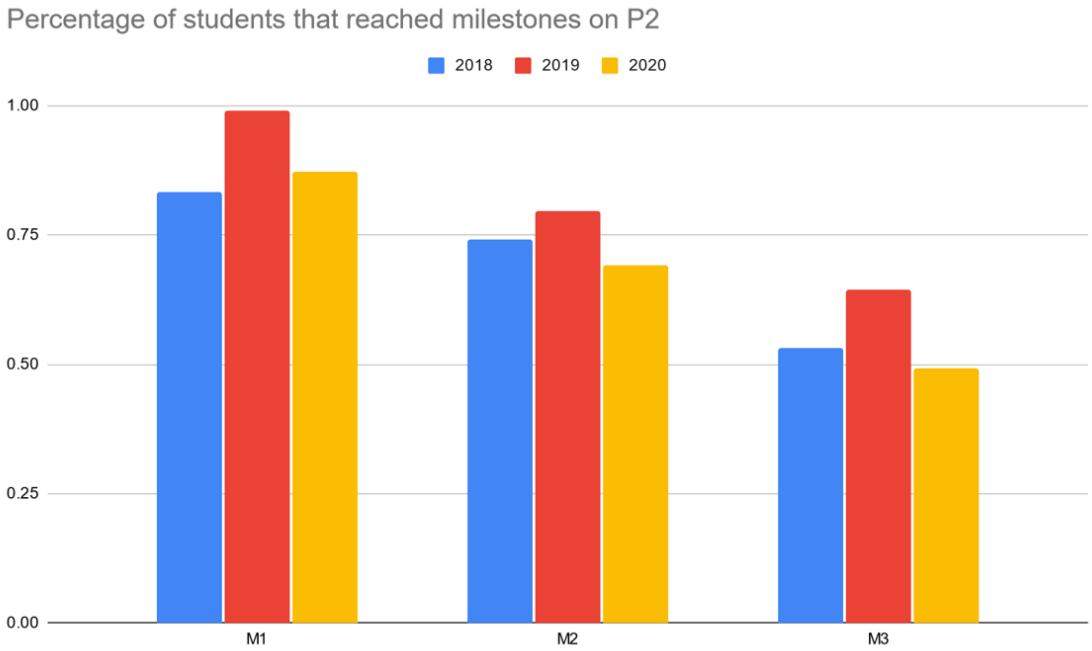


Figure 2.15: Proportion of students reaching milestones by the deadline on P2

2.3.4 Test Failures

Passing all teaching staff tests represents a complete implementation. We can understand how close the students were to done by considering the percentage of failing tests on their last commit. If students failed fewer teaching staff tests, that might suggest which feedback strategy supports students getting closer to the M3 expectations. Table 2.7 shows the number of teaching-staff test cases that were failing on the last commit, on average, per project for each semester. When comparing P1 and P1 among semesters, the percentage of submissions with failures at the deadline is similar. However, the amount of failures is higher for Project P1 in Fall of 2019, and higher for Project P2 in Fall of 2020.

When taking into consideration how many test cases existed for each project, we see that students had a similar percentage of test failures in Fall of 2018 and Fall of 2020 for projects P1 and P2. For Fall of 2019, the percentage of test failures was higher for Project P1, but lower for Project P2. Project P0 in Fall of 2018 had the highest rate of failed tests at the end of the project. The results do not enable us to define a clear difference when comparing the two feedback modes when looking at teaching staff test failures.

Table 2.7: Test failures on last commit

Semester	Project	Average # Failures	% Submissions	Total # of Tests	% Failures
Fall 2018	P0	10.4	58.9%	41	25.4%
	P1	5.2	23.3%	79	6.6%
	P2	3.2	34.2%	37	8.6%
Fall 2019	P1	13.0	22.5%	137	9.5%
	P2	2.1	32.3%	59	3.6%
Fall 2020	P1	6.2	26.7%	105	5.9%
	P2	6.6	35.4%	83	8.0%

2.3.5 Project Grades

Average grades are shown in Table 2.8. Fall of 2018, grades were higher compared to other semesters for project P2, and the lowest among all semesters for Project P1. Fall 2020, grades were the highest among semesters for project P1, and the lowest for project P2. This is likely related to difficulties introduced by the COVID-19 pandemic. Also, for the first two projects of 2018, students who submitted late work would have slightly higher grades if the late penalty was not applied. For all other projects and semesters, the grades for on time submissions were higher. From the grades, we have no strong indication of which strategy best supports student success.

Table 2.8: Average grades with no late penalty

Semester	Project	General	On Time Submission	Late Submission
Fall 2018	P0	73.3	73.0	73.9
	P1	60.1	60.1	62.7
	P2	82.6	85.4	60.9
Fall 2019	P1	80.9	83.6	58.4
	P2	74.5	89.8	20.1
Fall 2020	P1	82.0	84.5	62.2
	P2	53.6	82.9	15.4

2.3.6 Discussion

The results of our investigation are mixed; there isn't a clear answer to which feedback strategy best supports student learning. Project completion metrics suggest that *incremental feedback*

supports student achievement on reaching Milestone 3. However, milestone metrics suggest that *layered feedback* supports student efficiency through earlier completion in fewer commits. The results derived from the remaining metrics are mixed.

While the focus of this study was on feedback strategies, there are additional questions to consider from the results. A higher percentage of students achieved Milestone 2 in Fall 2018 when there was not a formal deadline for Process Points 2. Achievement of Milestone 2 was lower in Fall 2019 and 2020 when there was a formal deadline. While achievement as reported in Table 2.6 is independent of deadlines, there is a potential concern that a formal and evaluated Process Points deadline stalled student progress when they were unable to achieve the expected evaluation criteria.

The findings, however, support instructor observations from the semesters studied. The motivation for creating the incremental feedback was to provide earlier feedback to students a little bit at a time. With layered feedback, students receive all the test failures at once and sometimes they will approach fixing the failures in a suboptimal order. Incremental feedback was intended to provide failures a class at a time. However, the instructional staff noticed during office hours that students were still working on the early classes a few days before the deadline due to their focus on passing all the teaching staff tests before moving on.

We do tend to see an increase in achievement over a given semester, suggesting that students are improving their strategies for project completion. Part of that might be because Project 2 was required pairs in Fall 2018 and optional pairs in Fall 2019 and 2020; peer pressure may have contributed to Milestone 3 achievement. Fall 2020 was a pandemic semester, which likely impacted student engagement in the course independent of the feedback strategy.

Although the feedback changes influenced the students in different ways, at times creating new challenges, this study demonstrated the importance of the milestones used in the project. Therefore, it led to the milestone exploration done as part of our RQ1 question.

2.4 Effort Exploration

Students may present high levels of struggle for different reasons, including lack of previous computing experience, lack of self-regulation skills, or not comprehending the material. While those are different issues, the observable effect, low grades or a failure to complete work, can be the same. In this study, we aim to understand how students apply their efforts as a step to understanding the challenges they are facing.

Although understanding the way students apply their effort can help understand how they struggle, there are a few issues. The first issue with trying to identify effort is that there is no clear or consistent definition of what effort is, or how to measure it, especially in the context

of computer science education. Each student might show effort differently, i.e. one student may spend time planning all of their classes before starting implementation, while a different student may start implementation early by working on one class at a time. Another issue is that even if we knew what effort meant for each student, we did not have a way to measure effort that was applied outside of observable interactions with course resources. For example, we have no access to how much time a student spends thinking about the project before starting their implementation. A third issue is that, within the context of our course structure, we have no access to how much time the students spent working on the project, even for implementation tasks; all we have is the time each commit was submitted. This last issue led to building a plugin for the Eclipse integrated development environment that would capture student actions during implementation and data generated by the plugin is later explored in this dissertation. Without a way to understand what effort means for each student, we focused on defining forms of tangible, demonstrated effort that could be detected through the available data, while also focusing on more than just programming tasks and coding behavior. We also explore if our effort metrics lead to a way to know whether a submission will be submitted during a late window.

To our knowledge, at this time there are no studies that aim to create a measurable definition of effort in introductory computer science classes in the same way we are, although similar metrics have been used to understand student performance.

2.4.1 Understanding Student Effort

The overarching goal of this research is to find ways to identify struggling students. Since we have the issue of a ceiling effect on project grades, as explained in the previous sections, we decided to explore student coding behavior from a different angle. We try to understand where students were applying their effort since the progress indicators replication study showed that the number of commits a student submits is not necessarily related to their grade or their progress through the project. We were particularly interested in the idea of being able to detect effort because, as a former teaching assistant, I had seen too many students spend hours trying to fix a problem on their implementation only to remain stuck.

2.4.2 Methods and Results

Without a way to understand what effort means for each student, we focused on constructing a form of tangible, demonstrated effort that could be detected through the available data, while also focusing on more than just programming tasks and coding behavior. Furthermore, we also explore the relationships between effort metrics, between effort metrics and grades, and

between effort metrics and the lateness of a submission. Grades are a measure of performance, which may or may not be directly linked to the effort applied by the student. Late submissions can signify low effort, poor self-regulation, or struggle to complete the assignment. Either low grades or late submissions can happen for various reasons, such as lack of time-management skills, self-evaluation issues, evaluation misalignment, or personal issues the student may be dealing with.

Effort definition

We define effort as *actions taken towards project completion* in two ways. Our first definition of effort, *coding effort*, consists of coding actions like lines of code changed and amount of commits submitted. From my experience as a teaching assistant, I noticed that, unless close to a deadline, office hours were generally frequented by the same students. Therefore, our second definition of effort representation includes the frequency of use of the help-seeking resources available on the course. As described in 2.1, the course offers two forms of help-seeking resources: in-person office hours, which are organized and documented through MyDigitalHand (MDH); and an online forum for questions, which is hosted through Piazza. This definition of non-coding effort also included students' lecture attendance frequency. During a lecture, students completed 1-3 exercises related to the lecture's topic. The completion of those exercises was recorded and used as an attendance record. That attendance record is what we refer to as attendance in this paper.

Coding Effort Metrics - Study Sessions

Since there is no limit to how many commits a student can make, and no limit on commit size, we first divide the commits into *study sessions*, which identify periods where students are likely working on the project for a block of time. Following the procedure from Sheshadri et al. (2019), we measure how much time passed between commits. The idea behind measuring time between commits is that successive commits that happen with short periods of time between them are likely to represent a period where students were focused on working in the project, and longer times between commits likely represent pauses in work where students may have engaged in unrelated activities. We plot the distribution of time between commits for every semester, an example distribution of time between commits can be seen in Figure 2.16. In every semester, the majority of commits happen with 150 minutes or less in between them. Therefore, we define a session as a sequence of commits in which the time gap between subsequent commits is 150 minutes or less. Once 150 minutes have passed with no commits, the previous commit is considered the end of the previous session, and the next session starts

on the next commit. This means that sequential commits with less than 150 minutes, or 2.5 hours, between them will belong to the same session. This process does not create sessions of 2.5 hours, it divides sessions using gaps of 2.5 hours between commits. That means a session does not have a length limit, i.e. a student could be in the same session for 8 hours as long as there is not gap larger than 150 minutes between commits. That also means there is no limit of commits per session, i.e. a session could have only one commit if the next commit was done more than 2.5 hours after, or it could have 100 commits if there were only 5 minutes between successive commits. This process does capture when students stop working continuously on the project. We choose this approach because by using gaps between commits we can explore times students were working on the project independently of how many commits they make, and independently of how long each individual session is. However, this procedure does miss non-coding effort, e.g. a student could stop commits for longer than 2.5 hours to look into course materials, therefore we only capture continuous coding effort.

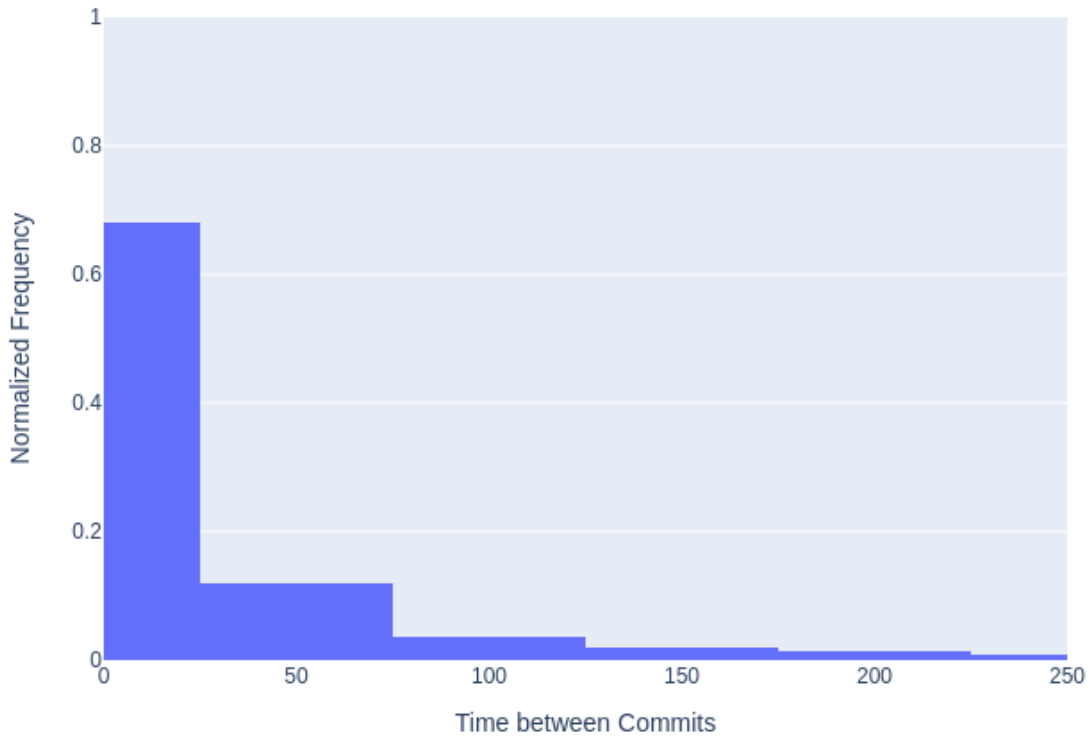


Figure 2.16: Distribution of time gaps (in minutes) between commits.

For each repository, we collect the number of sessions, and the medians of 1) number of commits per session, 2) session length, 3) number of lines changed per session, 4) number of files changed per session, 5) session start time, 6) session end time, and 7) time between each session and the project deadline. Session start and end times were in timestamp format, which allowed us to calculate medians. Medians were used instead of averages to lessen the weight of outliers.

Grading Metrics

For each repository, we also collect grades for unit and system teaching staff (TS) tests and overall project grades. Unit TS tests test each method of the student's implementation, while system TS tests test the functionality of the implementation and whether it completes project requirements. We collect unit and system TS tests grades because they relate directly to the student's code and cannot be improved without improving the implementation. The overall project grade includes grade items that students can earn credit for even if their solution is incomplete (e.g., high-quality code comments). We explore whether effort metrics have any relationships with grades, however we do see a ceiling effect on the grades where most students achieve high grades on the projects. The grades used in this study do not have late submission penalties applied to them.

Non-coding Metrics

Our second effort definition revolves around tangible student actions beyond coding behavior. For each repository, we collect the 1) frequency of each student attending office hours, 2) posting Piazza questions, and 3) attending lectures. Office hours and Piazza question frequency are collected per project; we do not consider help-seeking behavior related to other parts of the course.

Metrics aggregation and comparison

We first separate effort metrics per semester, and calculate the averages of each effort metric for each semester, to explore any differences that may happen between semesters. We also separate and calculate the averages of each effort metric, for each semester and across semesters, between projects that were completed individually and projects that were completed by student pairs and explore those differences. We then aggregate all effort metrics by creating letter grade groups, also for each semester and across semesters. For each letter grade group, we collect the average for each effort metric from repositories that received that grade. This process is

intended to separate effort metric values into groups that could show meaningful differences in the students' efforts and whether that effort relates to project grades despite the ceiling effect.

We compare effort metrics between semesters and also analyze whether effort metrics show differences between students developing projects individually and students developing the projects in pairs. To explore relationships between effort metrics, we calculate correlations for each effort metric pair using Spearman's correlation (Chambers 1989), since our data does not follow a normal distribution. We analyze the validity of those correlations by calculating corrected p-values using the Holm-Bonferroni method (Abdi 2010); the correction to the p-value is necessary because we stack multiple statistical inferences, e.g. multiple medians are collected per repository, which are first averaged to create aggregate values for each repository, and then averages are collected per semester and across semesters.

We show the averages of effort metrics per repository for each semester in Table 2.9, and average of effort metrics per repository across semesters, per group type, and per grade group in Table 2.10. Grades were analyzed in more detail, including grade groups with minus and plus grades – i.e. A+ and A-. We show groups aggregated by average for space purposes, i.e. A-, A, and A+ grades were averaged to make the A-grade group.

When observing the correlations across semesters, we find few significant relationships. Project grades largely depend on TS unit and system test grades, and do present highly, significant (p-values equal zero) correlation values, with the correlation of project grades being 0.784 and 0.746 to TS system tests and TS unit tests, respectively. TS unit tests and TS system tests have a statistically significant relationship (p-value equal zero) of 0.986. Additionally, lines changed per session and files changed per session have a significant (p-value equal zero) correlation of 0.728. Across semesters, non-coding effort metrics did not present any significant relationships with the other effort metrics or grades, no effort metric had a relationship with lateness status.

When aggregating effort metrics by grade groups, we see that students who earned D- and F grades on the project tend to have fewer help-seeking interactions and lower attendance rates. Students in the D-grade group (D+, D, and D-) tend to start their projects later than students in the A-grade group, but earlier than students in the C-grade group. Students with lower grades tend to have fewer study sessions, but similar project length, which indicates they work for longer without as many breaks. Students earning D- and B+ tend to change more lines per session than students in the A-grade group. As seen with the correlations, TS unit tests and TS system tests grades are lower in the lower grade groups, and students in the A-grade group tend to submit their projects on-time more often than students in other grade groups. We compare effort metric correlations for each semester. In Fall 2019 and Spring 2020, lateness status and time between study session and deadline have a significant correlation of -0.503 and -0.514,

which did not happen in other semesters. In Fall 2020, there exists a significant relationship between attendance and project grade, with a correlation of 0.530. The largest difference is in Spring 2020, where there are significant relationships between TS unit tests, TS system tests, and grade effort metrics with project start and end, and session time effort metrics. During Spring and Fall 2020, students seem to have used help-seeking resources more often.

We see that teams tend to start the project earlier and work on the project for longer. Session length for teams is also shorter. Teams also tend to have more commits per session, and more sessions, but fewer code lines and files changed in each session, When working in teams, students have a lower rate of late submissions while also having higher grades.

Table 2.9: Effort metrics averages per semester.

Semester	F19	S20	F20	S21	F21	S22	F22	S23	F23	S24
MDH Interactions	2.4	5.1	6.9	4.5	4.2	3.7	4.8	5.9	4.8	3.3
Piazza Posts	3.5	5.1	5.9	3.0	3.3	3.5	2.6	4.2	4.7	3.9
Attendance (%)	93.8	91.3	84.7	86.9	86.1	84.0	90.5	77.1	82.2	83.5
Hours between first commit and project deadline	396.0	384.7	386.2	387.3	386.5	381.8	376.9	387.5	384.1	388.2
Hours between first commit and last commit	396.4	392.5	378.0	379.0	379.1	377.9	370.2	379.2	399.7	397.0
Session Length (min)	267.4	238.6	165.7	193.4	220.3	119.8	234.9	192.2	195.3	204.3
Hours between median session start and project deadline	181.8	163.2	177.7	188.0	181.1	170.5	173.3	178.0	180.6	181.9
Commits per session	8.2	10.7	7.1	6.8	6.3	6.6	5.4	6.4	5.6	6.4
Session Count	15.4	14.6	13.4	12.3	12.3	12.4	13.2	13.0	12.6	12.1
Lines Changed per Session	171.5	266.3	224.1	168.7	246.3	199.7	205.8	132.6	199.4	150.0
Files Changed per Session	3.3	3.6	3.7	3.9	4.0	3.7	4.1	3.1	4.1	3.4
Late Submissions (%)	67.0	79.2	61.1	55.6	66.2	66.3	61.2	55.0	57.0	53.2
TS Unit Tests Grade	90.9	84.8	90.3	94.5	84.2	77.5	91.9	91.1	88.4	84.8
TS System Tests Grade	81.6	71.0	84.1	82.2	74.9	76.8	78.2	87.4	78.4	79.1
Project Grade	86.4	80.1	87.0	86.9	84.2	83.1	89.2	89.4	86.7	85.9

Table 2.10: Effort metrics averages by group types and grade groups, across semesters.

		Avg. by group type		Avg. by grade				
Metrics	**Avg. across semesters**	**Individual**	**Teams**	**A**	**B**	**C**	**D**	**F**
MDH Interactions	4.6	4.9	4.1	4.8	5.1	6.5	4.0	3.9
Piazza Posts	4.0	4.2	3.6	4.2	3.9	4.9	3.5	4.0
Attendance (%)	86.0	85.1	86.8	88.3	86.5	76.8	80.3	68.7
Hours between first commit and project deadline	385.9	375.1	429.1	397.8	364.2	357.0	367.6	339.2
Hours between first commit and last commit	384.9	374.1	423.5	392.9	374.7	377.9	408.5	327.5
Session Length (min)	203.2	215.7	153.2	147.7	130.6	502.7	166.0	538.1
Hours between median session start and project deadline	177.6	169.3	211.4	186.6	152.9	145.8	135.0	158.4
Commits per session	6.9	6.7	8.0	7.3	8.2	7.6	7.3	4.8
Session Count	13.1	11.8	18.5	14.2	12.1	13.1	11.3	7.9
Lines Changed per Session	196.4	230.5	89.5	160.0	204.6	193.6	273.0	409.3
Files Changed per Session	3.7	4.0	2.7	3.4	3.9	3.6	4.4	5.4
Late Submissions (%)	62.2	63.9	59.0	55.7	84.4	92.2	92.2	76.7
TS Unit Tests Grade	87.8	85.7	96.5	98.7	91.8	72.8	64.3	31.2
TS System Tests Grade	79.4	77.2	87.2	95.4	74.5	53.7	29.2	9.5
Project Grade	85.9	83.9	93.5	95.3	85.1	75.0	65.0	35.2

2.4.3 Using study session metrics as predictors of grade

We attempted to create a grade prediction model using the study session metrics. We run the task for each semester. First we implement a multivariate regression model, using a stratified 70/30 data split, which means 70% of the data was used as training data, and 30% of the data was used as testing data, and the proportion of grades between data groups was equivalent. However, we ran into a ceiling-effect issue in our data; most students achieved high grades on the projects, so our model ended up predicting mostly "A" grades. The model statistics and a confusion matrix showing the prediction results can be seen in 2.17 and 2.18. Accuracy was above 50%, which means it was better than a guess. However, the precision score is less than 50%, which means that our accuracy was above 50% by predicting "A" for most grades, but most of the grades predicted were incorrect.

We also used a second approach for a prediction model, also using a multivariate regression but splitting the data using the deadline for Milestone 2. Data from sessions before students achieved Milestone 2 was used as training data, and data from sessions after students achieved Milestone 2 was used as testing data. The idea was to see if it would be possible to predict which students would fail the project by using early development data. However, this prediction model was extremely inaccurate, as it can be seen in the example on figure 2.18, which shows data from Fall 2020 as an example.

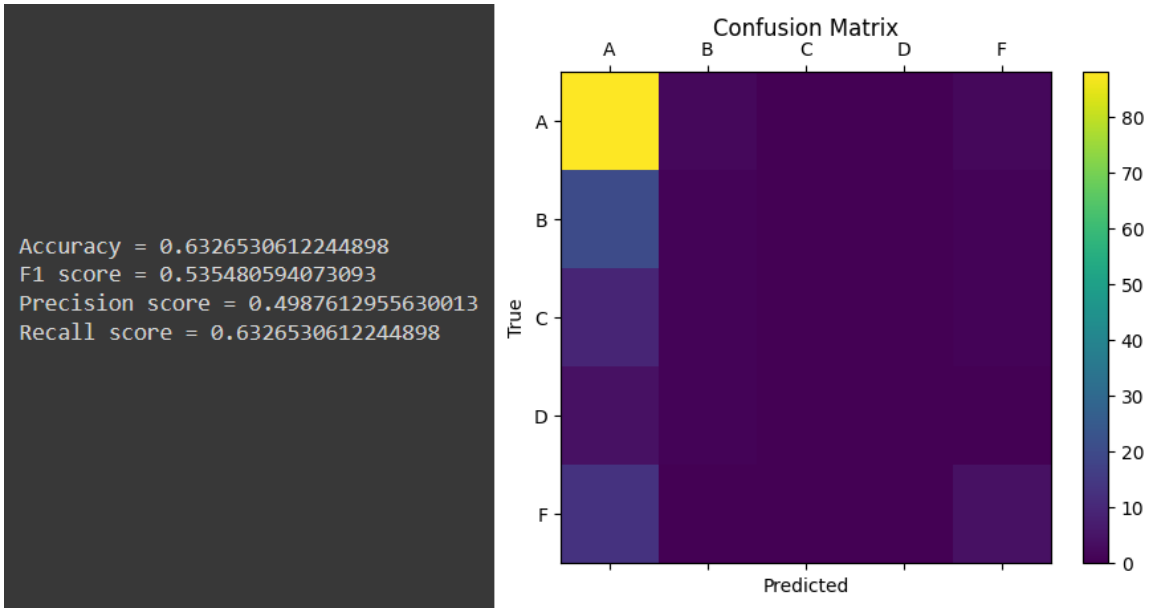


Figure 2.17: Prediction of grades for Fall 2020 using 70/30 data split

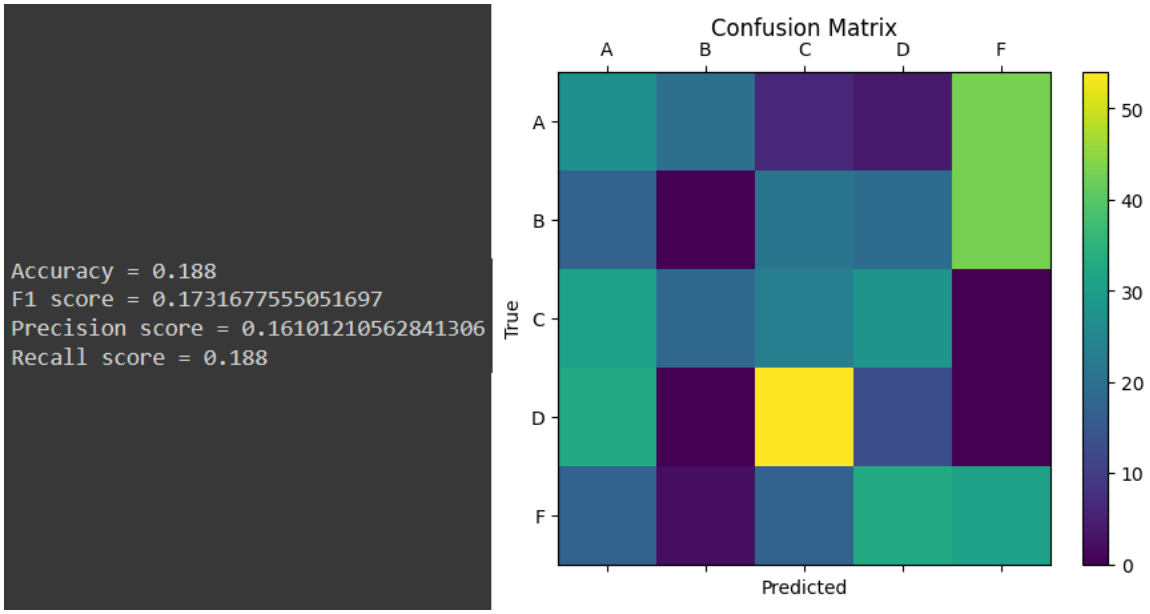


Figure 2.18: Prediction of grades for Fall 2020 using time data split

Considering that we had the ceiling effect on grades on the other semesters and that the correlation values between metrics and grades were low for every semester, we also explored the same relationships using grades lower than 83%, however, the results were the same.

To try and isolate behaviors that could lead to high grades versus low grades, we then

focus on comparing data from sessions only from students that had a grade either equal to "A" (including "A-", "A", and "A+" grades) or "F". Although the correlations demonstrated no strong relationship between the session metrics, we did see a difference in the way students were working on their projects when comparing those grade groups. Agreeing with the existent literature (Williams et al. 2021; Leinonen et al. 2021), the students with higher grades started working on the projects earlier. Also, there seems to be a difference in the order students in each group work on files. As an example, Figure 2.19 show box plots for each file required for completion of P1 in Fall of 2019. Each green box plot refers to the students with grade "A", and each red box plot refers to the students with grade "F". Files are in the same order on both plots. "A" students concentrate on altering some files earlier than others while "F" students alter all files with no defined order. This observation has led to one of the sub-questions on our current RQ1, where we explore how the order of implementation changes depending on struggle rates.

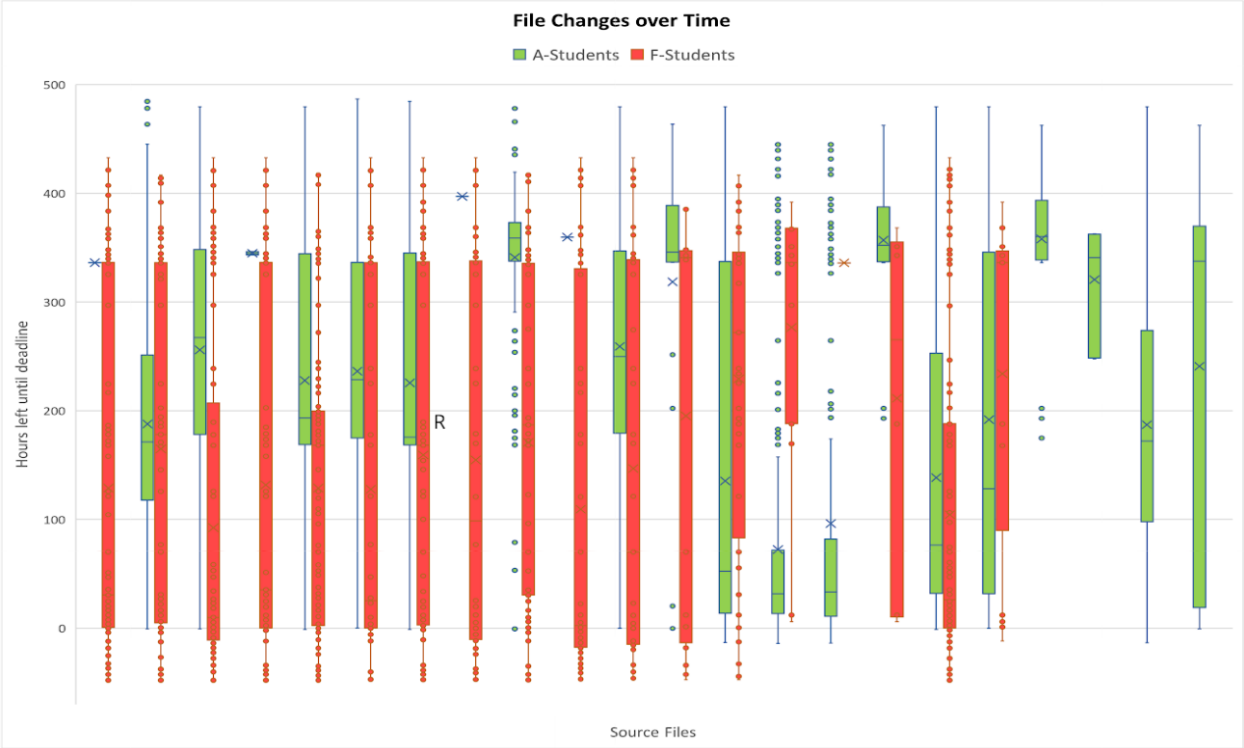


Figure 2.19: Average time of file changes by students that received an A on the project, in Fall 2019

2.4.4 Discussion

Through our exploration of relationships between metrics, we can see that non-coding effort and coding effort did not correlate to each other. That means that, in general, how students work on their implementation does not depend on their help-seeking behavior. On average, help-seeking interactions during project implementation were low; finding no relationship between help-seeking behavior and grades agrees with results in other studies (Gao et al. 2022a; Battestilli et al. 2022). TS unit tests and TS system tests do correlate with each other, meaning that students should be able to achieve high grades on system tests if they work on passing all unit tests. Students can access feedback from TS unit tests, but not from TS system tests. This creates an issue, as students could achieve high grades on system tests without thinking of the project as a system just by correcting their implementation based on TS test feedback. That is corroborated by the relationship between lines changed and files changed, as students tend to modify few files during sessions; this signifies they are working on a few files at a time and modifying lines in only those files, missing the perspective that the files they are working might depend on other files.

Even though there was no significant correlation between grades and non-coding effort, students with low grades did show lower help-seeking rates. Students in the C-grade group start their development later than students in the D-grade group; that might indicate a self-evaluation issue, where students in the C-grade group thought development would be easier for them than it was – pushing them to delay their development, and might also indicate students in the C-grade group struggle with time-management skills. Students in the D- and B+ grade groups tend to change more lines than students in the A-grade group, which might indicate they are struggling to know how or where to fix their mistakes leading to possibly unproductive code churn. Additionally, even though the grades in this study did not have late submission penalties, we could see that students in the A-grade group submitted their assignments on time more often. Additionally, D-grade and A-grade groups present similarities, which may indicate one of the main factors in achieving high grades is early development start.

On average, students in the A-grade group also start their development earlier than other grade groups, working for longer on the project with a higher session count. That could indicate students in the A-grade group have better time management and self-efficacy skills, knowing when they need to start based on their programming skills and subdividing their tasks better than students in the other grade groups.

When exploring changes between semesters, we did not see many differences when analyzing effort metrics between semesters as opposed to across semesters. Most differences happened in Spring 2020, when grades and early work/submission had a significant relationship, meaning students that started their projects earlier achieved higher grades. However, that

might have been impacted by the selection of students who consented to the study in that semester, i.e. students who consented might have been students who were more engaged in the course. Additionally, we do see an increase in the use of help-seeking resources during Spring and Fall 2020. In those semesters, students were working from home due to the COVID-19 pandemic. Online office hours Gao et al. (2022a) increase accessibility; additionally, students likely had fewer opportunities to converse with teaching staff outside of office hours. Those factors explain why students were frequenting office hours and posting questions more often during those semesters.

When looking at differences in effort by individual students versus student pairs, there are differences in effort. Teams start the projects earlier and work for longer, using more, shorter, focused sessions. That shows that students tend to divide their work better when working in teams. Teams also tend to achieve higher grades and submit late less often. Those differences can indicate working in teams might help with self-efficacy and evaluation misalignment issues as students have a partner to discuss the project. These results agree with studies on pair-pressure in programming Williams and Kessler (2000); Jun et al. (2007). We saw that students presented on average the same results across semesters, even though changes happened to both course structure and course delivery. For our course structure, we concluded that exploring study sessions is insufficient for identifying struggling students. Part of the challenge is the unbalanced data sets since there is a ceiling effect where most students achieve high grades.

When observing the calculated correlations, there were no significant relationships between non-coding and coding metrics. However, when aggregating metrics by grade groups, students who got a B on the project tend to work on longer sessions, and have higher commit and files changed per session count than students with higher grades, while students who got C+, C, C-, D+ and D as their grade had a higher session count than students with higher grades. Students who got a D- also had a much higher rate of lines changed per session than all other students, and a higher rate of files changed per session.

Predicting grades through effort metrics was not successful, as the prediction tasks were impacted by the ceiling-effect in our grades.

CHAPTER

3

IDENTIFYING AND QUANTIFYING STRUGGLE (RQ1)

Our first research question, *"How can we identify struggle using submission data?"*, explores the creation of a definition of struggle using the data from our course projects. The question is divided into two sub-questions:

RQ1.1 "How to define and quantify struggle?" (RQ1.1)

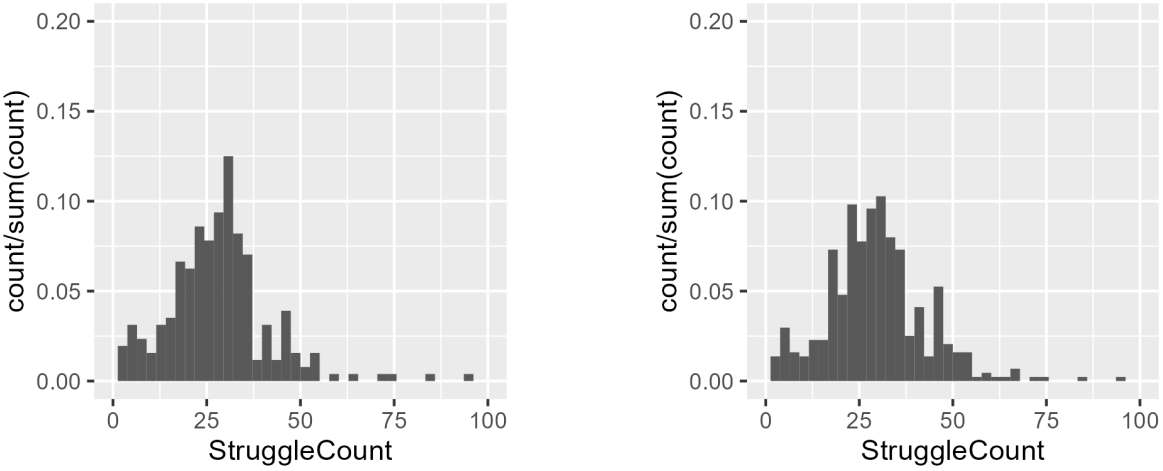
RQ1.2 "What kinds of different struggle patterns appear in the data?" (RQ1.2)

Through our progress analysis, as shown in 2.2, we see that the largest variance of code changes happens after students reach M2 in the projects. That is the moment when they start receiving feedback from teaching-staff (TS) test cases and start to fix the bugs in their code. Therefore, to answer these research questions, we focus on data from after milestone M2. In this research, we define progress as the decrease in TS unit test cases failures between consecutive commits.

3.1 Initial definition of struggle

When analyzing progress, we identify that across all semesters, and all projects, students take between 3 and 4 commits to solve an error triggered by a specific TS test case. Therefore, we define struggle as having an error on the same TS test case for 4 or more commits, in which case we call that error a *persistent error*. The notion of persistent errors have been used similarly in the literature before (Liu and Paquette 2023). The set of commits containing a persistent error is defined as a *struggle sequence*. A struggle sequence starts once an error has persisted for 4 commits and ends at the first subsequent commit where that error does not happen, which means there is no limit for the length of a struggle sequence. Although this is a straightforward definition of struggle that can be generalized to other courses that use unit test cases, we propose a new definition and explain the changes and reasons in Section 3.2. When starting this research, the data available was from Spring semesters of 2020 and 2021, and Fall semesters of 2019, 2020, and 2021. We were able to collect data from other semesters, as described in 2.1, and use the complete dataset on the later phases of this research.

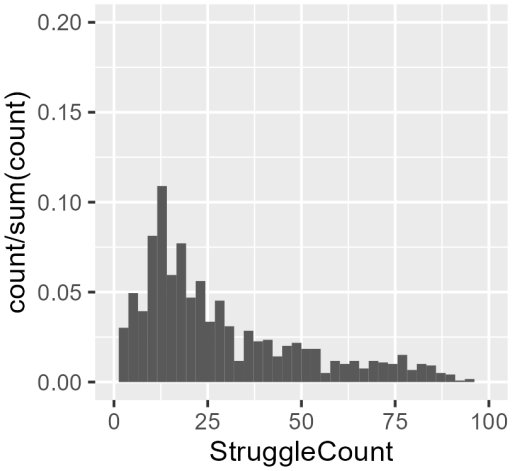
We collect, per project and across repositories, all occurrences of struggle sequences based on our first definition of struggle previously described. Distributions of struggle sequences can be seen in Figures 3.1 through 3.5. Although distributions are similar, there were fewer moments of struggle in Spring 2020, Spring 2021, and Fall 2021. Those results were surprising, given those semesters were during the COVID-19 pandemic and students did not have access to in-person assistance.



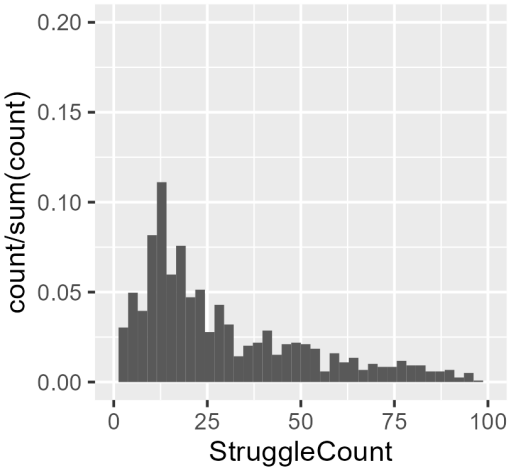
(a) F19 - P1

(b) F19 - P2

Figure 3.1: Distribution of struggle sequences through each project - Fall 2019.

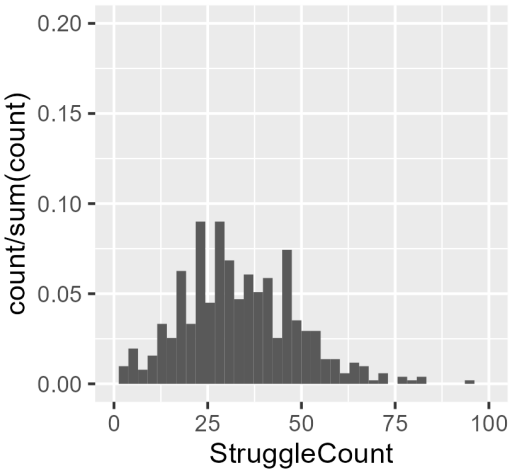


(a) S20 - P1

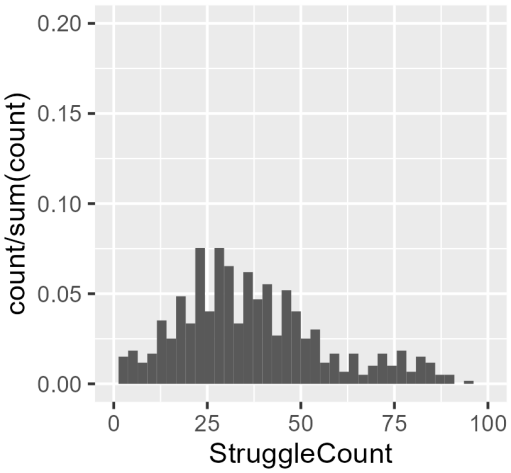


(b) S20 - P2

Figure 3.2: Distribution of struggle sequences through each project - Spring 2020.

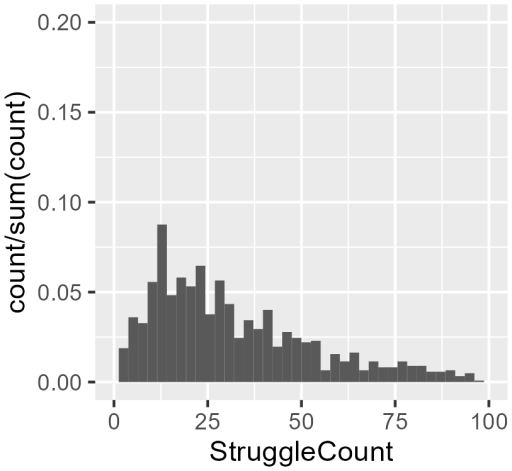


(a) F20 - P1

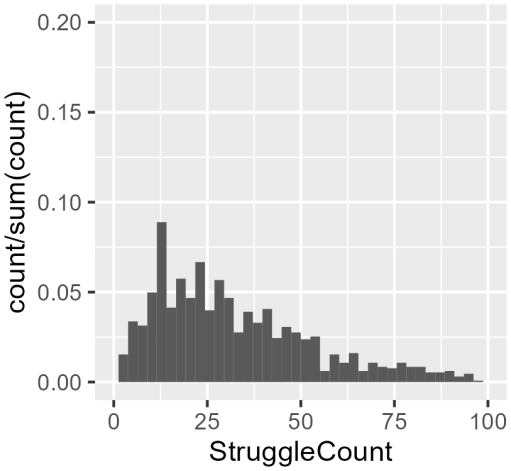


(b) F20 - P2

Figure 3.3: Distribution of struggle sequences through each project - Fall 2020.

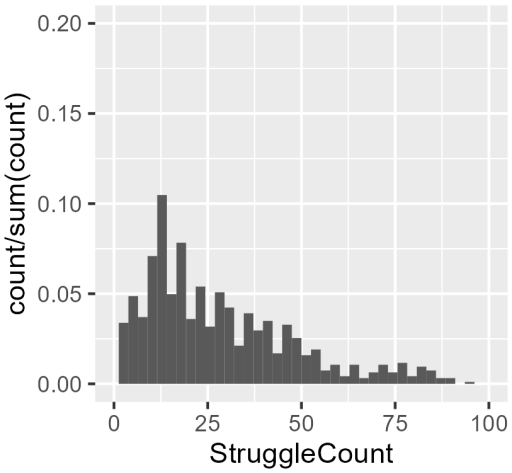


(a) S21 - P1

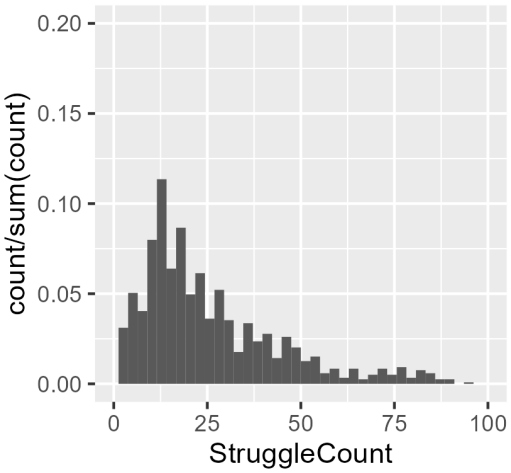


(b) S21 - P2

Figure 3.4: Distribution of struggle sequences through each project - Spring 2021.



(a) F21 - P1



(b) F21 - P2

Figure 3.5: Distribution of struggle sequences through each project - Fall 2021.

3.2 New Definition of Struggle (RQ1.1)

Although our previous definition of struggle led to a meaningful exploration of how students respond to persistent errors, each commit can contain multiple persistent errors. Therefore, when counting struggle sequences, each commit that had more than one persistent error was being counted multiple times. Due to that problem, we decide to develop a new definition of struggle by using the set of persistent errors contained in each commit. Since the multiple persistent errors contained in a struggle commit relate to different classes, we also develop the concepts of *global struggle* and *local struggle*.

Global struggle includes all persistent errors present in a commit, which is defined as a global struggle commit if the containing set of persistent errors is equal or a superset of the persistent errors contained in the previous commit.

Local struggle is a subdivision of global struggle, and refers to persistent errors relating to the same class. A commit is a local struggle commit if the set of persistent errors related to a given class is equal or a superset of the set of persistent errors related to the same class in the previous commit.

The notion of persistent error is still the same, with being a test case failure that persists for 4 or more consecutive commits. Therefore, both global and local struggle commits are only defined once the same set of persistent errors appears for 4 or more consecutive commits.

With that, we also create the definition of *progress commit*. A progress commit is a commit in which its set of persistent errors are a *subset* of the set of the persistent errors present in the previous commit. Even if the number of TS test failures increases, the change in the set of failures indicates progress, as high as the new set is not a superset of the persistent errors present in the previous commit, i.e. at least one TS test failure was fixed. A progress commit indicates an increase in progress, as we define progress as a decrease in TS test failures, and signifies the ending of a particular struggle sequence.

Along the notion of a progress commit, we are also creating the definition of a *debugging commit*. A debugging commit is a commit within a struggle sequence that presented either source class or test class changes related to an error that was fixed at the ending of that struggle sequence. The idea is that students were either modifying the source class to try to fix the error, or modifying the test class in order to reproduce the error given by TS tests.

Additionally, we use the progress indicators Edwards and Li (2016) that fit with our course structure to separate productive struggle from unproductive struggle. Using the same definition of productive commits that we have arrived at during our progress exploration, described in Section 2.2. We classify a struggle sequence as productive struggle if at least 60% of the commits in the struggle sequence were productive commits.

We collect global and local struggle sequences for all repositories from consenting students within the semesters included in this dissertation. First we analyze the relationship between struggle and grades. We then explore differences on count and length of struggle sequences between semesters, projects, and student group types.

3.2.1 Struggle count and length

Descriptive statistics for the count of struggle sequences, e.g. the number of struggle sequences per repository in each semester can be seen in Table 3.1. Descriptive statistics for the length of local struggle sequences, e.g. the number of commits in each struggle sequence per repository, can be seen in Table 3.2.

In general, local struggle shows larger variances than global struggle, especially for the local struggle counts. Values are similar between semesters. However, for Spring of 2020 the standard deviation for both global and local struggle count was larger than the average, meaning that semester presented a larger variance in struggle counts than the other semesters. As Spring of 2020 also show higher averages and maximum values for global and local struggle, students in that semester appear to have struggled more than students in other semesters. Fall of 2020 shows the second highest maximum value for both global and struggle counts, however Fall of 2023 also show a high maximum relatively to other semesters.

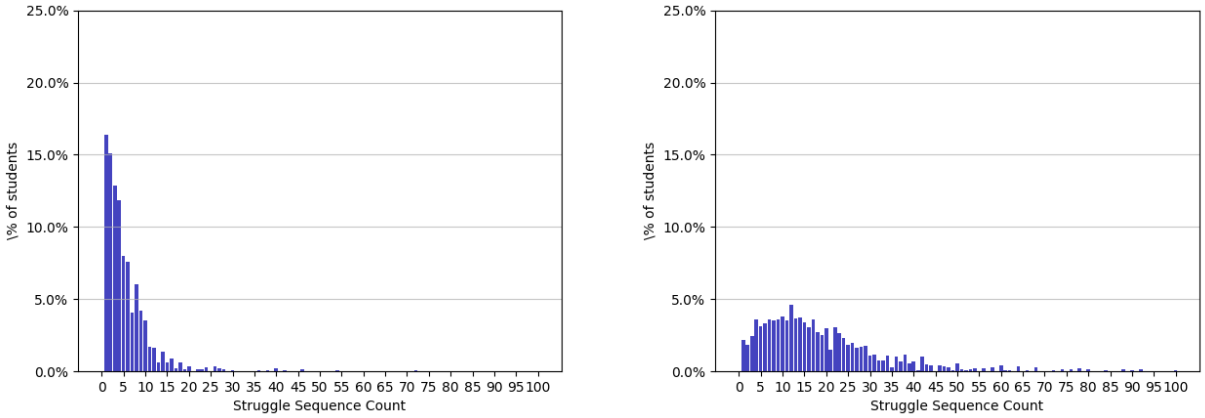
Table 3.1: Count of struggle sequences per repository in each semester.

	Global Struggle Count					Local Struggle Count				
Semester	**Avg**	**Std. Dev.**	**Median**	**Min**	**Max**	**Avg**	**Std. Dev.**	**Median**	**Min**	**Max**
F19	5	5	4	1	26	15	14	11	1	84
S20	9	10	6	1	72	24	26	17	1	228
F20	6	6	5	1	54	24	17	20	1	146
S21	5	4	4	1	28	18	12	16	2	92
F21	4	4	3	1	30	14	11	12	1	80
S22	7	7	5	1	46	26	17	24	1	92
F22	4	4	3	1	18	19	12	16	2	72
S23	4	3	3	1	18	17	13	14	1	48
F23	5	4	4	1	24	17	15	14	1	100
S24	4	3	4	1	9	14	11	11	1	42

Struggle sequence count distributions for global and local struggle can be seen on Figure 3.6, and struggle sequence length distributions for global and local struggle can be seen on Figure 3.7.

Table 3.2: Struggle sequence length per repository in each semester.

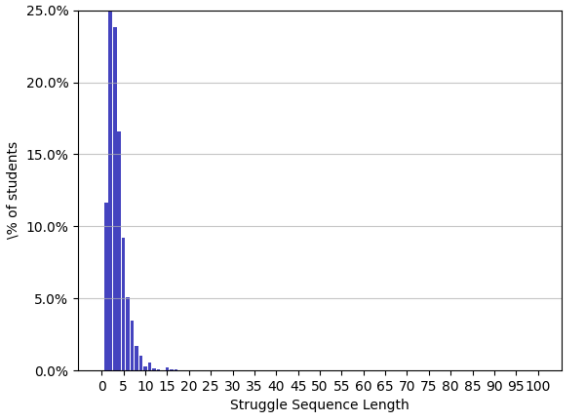
	Global Struggle Length					Local Struggle Length				
Semester	**Avg**	**Std. Dev.**	**Median**	**Min**	**Max**	**Avg**	**Std. Dev.**	**Median**	**Min**	**Max**
F19	3	3	3	1	27	5	10	3	1	120
S20	3	2	3	1	44	6	13	3	1	164
F20	2	2	2	1	39	5	9	3	1	167
S21	2	2	2	1	21	5	8	3	1	162
F21	3	3	3	1	38	5	9	3	1	125
S22	2	2	2	1	29	5	9	3	1	170
F22	2	2	2	1	33	5	8	3	1	114
S23	2	2	2	1	17	4	7	3	1	122
F23	3	3	2	1	31	5	8	3	1	102
S24	2	1	2	1	12	5	8	3	1	82



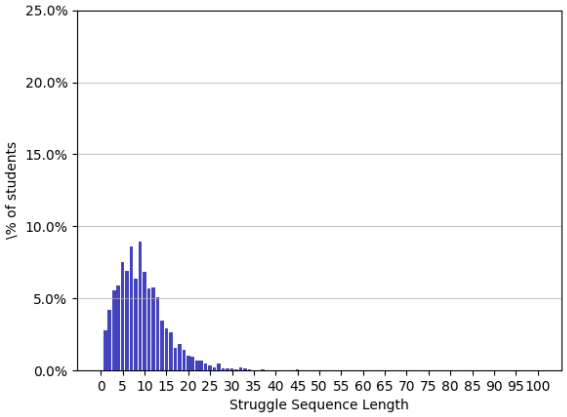
(a) Global struggle count distribution.

(b) Local struggle count distributions.

Figure 3.6: Struggle count distribution including all repositories from all semesters.



(a) Global struggle length distribution.

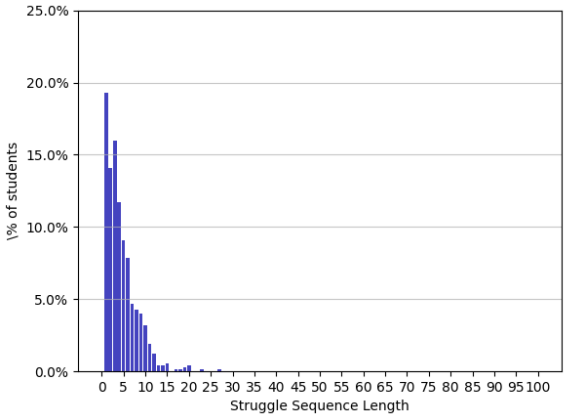


(b) Local struggle length distribution.

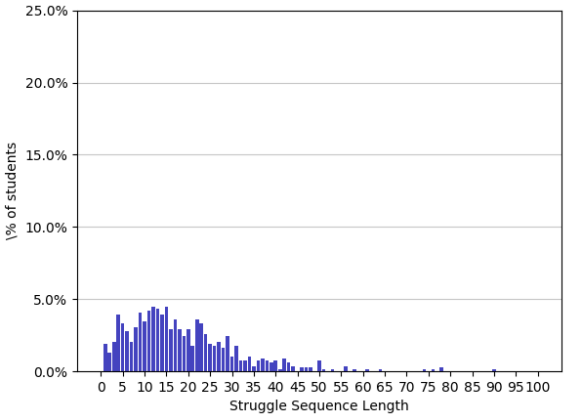
Figure 3.7: Struggle length distribution including all repositories from all semesters.

Differences between projects

Distributions for each project are presented across semesters. For Project 1, global and local struggle count distributions can be seen on Figure 3.8, and global and local struggle length distributions can be seen on Figure 3.9. For Project 2, global and local struggle count distributions can be seen on Figure 3.10, and global and local struggle length distributions can be seen on Figure 3.11.

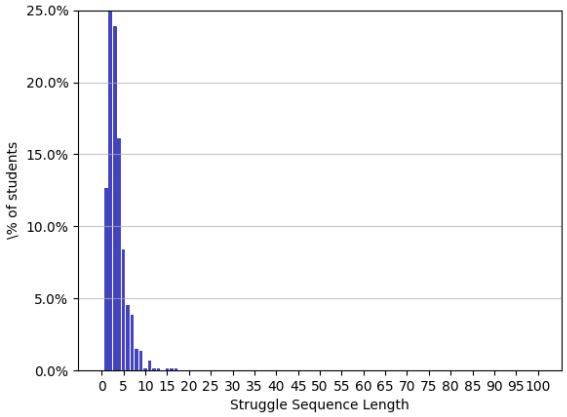


(a) Global struggle count distribution for Project 1.

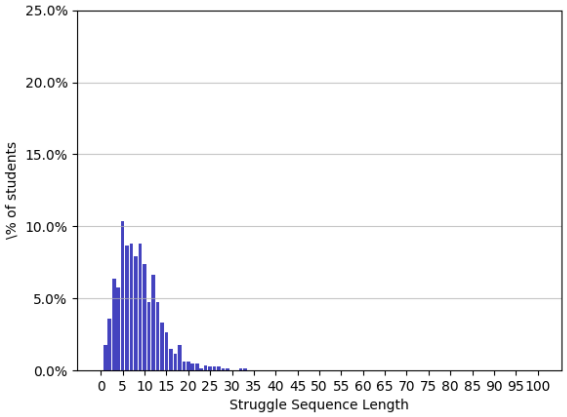


(b) Local struggle count distribution for Project 1.

Figure 3.8: Global and local struggle count distributions for Project 1 across semesters.

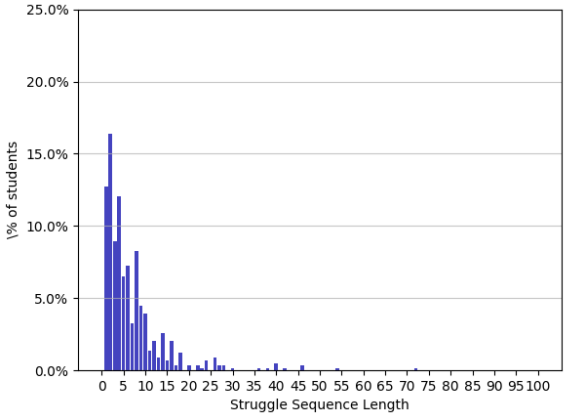


(a) Global struggle length distribution for Project 1.

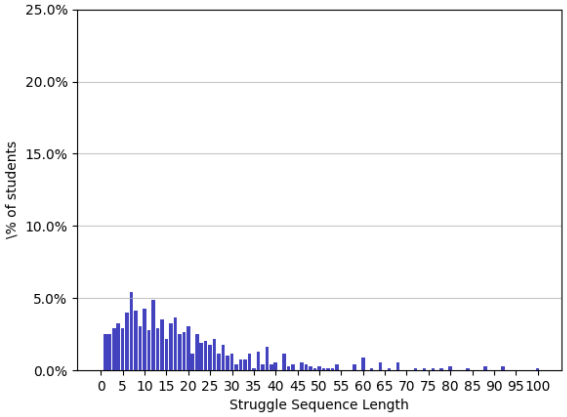


(b) Local struggle length distribution for Project 1.

Figure 3.9: Global and local struggle length distributions for Project 1 across semesters.

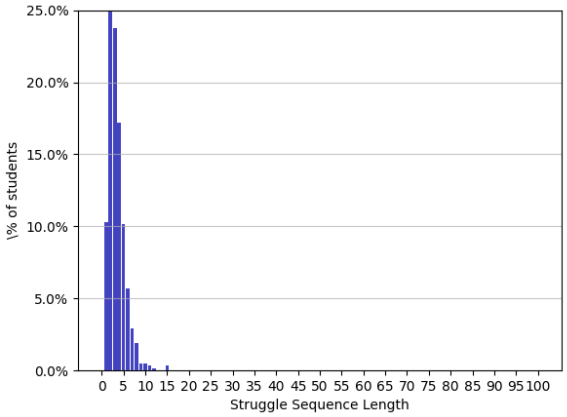


(a) Global struggle count distribution for Project 2.

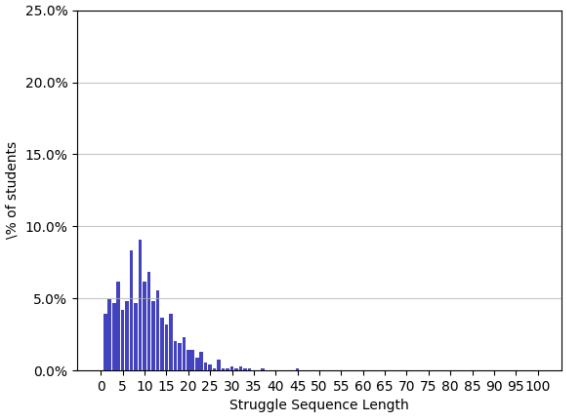


(b) Local struggle count distribution for Project 2.

Figure 3.10: Global and local struggle count distributions for Project 2 across semesters.



(a) Global struggle length distribution for Project 2.



(b) Local struggle length distribution for Project 2.

Figure 3.11: Global and local struggle length distributions for Project 2 across semesters.

Global and local struggle sequence count differences per project can be seen on Table 3.3. With the exception of Spring of 2023 and Spring of 2024, where the highest struggle count happened in Project 1, global struggle counts tend to be higher on Project 2. We can see the large global and local struggle count values on Spring 2020 and Fall 2020 happened during Project 2, as in Project 1 the results were similar to other semesters.

Differences in global and local struggle sequence length per project can be seen on Table 3.4. Local struggle sequences tend to be higher than global struggle sequences. Global struggle sequence lengths tend to be similar between projects, while local struggle sequence lengths tend to be higher during Project 2. Spring and Fall of 2020 present higher struggle sequences than most semesters, however the highest struggle sequence happened on Project 1 of Spring of 2022. The largest variances appear in local struggle sequence lengths from Spring and Fall of 2020.

Table 3.3: Count of struggle sequences per repository in each project.

		Global struggle count					Local struggle count				
Semester	**Project**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**
F19	**P1**	3.87	3.66	3	1	20	15.73	14.01	12	1	78
S20		5.23	3.92	5	1	20	16.75	13.31	14	1	74
F20		5.15	3.72	4	1	27	19.09	10.64	18	1	64
S21		5.61	3.89	5	1	20	19.32	10.43	18	2	56
F21		3.31	2.42	3	1	12	12.04	7.66	12	1	40
S22		5.11	3.98	4	1	23	26.55	15.71	24	1	90
F22		4.03	2.94	3	1	13	17.65	9.64	15	5	50
S23		4.56	3.48	4	1	18	24.45	11.73	25	1	48
F23		3.51	2.70	3	1	12	16.09	11.08	14	2	61
S24		4.70	3.27	4	1	9	22.36	10.85	18	10	42
F19	**P2**	6.09	5.12	5	1	26	14.26	13.09	9	1	84
S20		13.08	13.45	9	1	72	32.78	34.43	20	2	228
F20		7.01	7.05	5	1	54	29.45	21.25	23.5	1	146
S21		5.35	4.74	4	1	28	17.40	13.65	14	2	92
F21		5.22	4.39	4	1	30	16.81	12.83	14	1	80
S22		10.41	10.57	6	1	46	25.49	20.27	20	3	92
F22		5.07	4.42	3	1	18	19.66	13.72	16	2	72
S23		3.28	2.87	2	1	12	8.90	7.81	6.5	1	36
F23		6.45	5.01	6	1	24	17.22	18.57	13	1	100
S24		2.25	1.26	2	1	4	6.08	4.50	6	1	15

Table 3.4: Struggle sequence length per repository in each project.

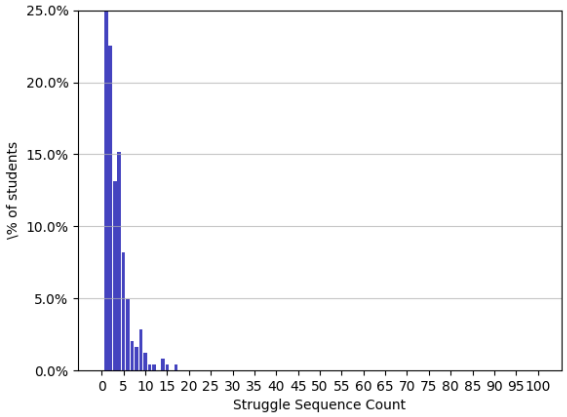
		Global struggle length					Local struggle length				
Semester	**Project**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**
F19	**P1**	2.36	2.02	2	1	23	4.72	8.02	3	1	75
S20		2.51	2.26	2	1	33	5.26	10.61	3	1	161
F20		2.52	2.30	2	1	28	4.59	7.83	3	1	137
S21		2.49	2.07	2	1	21	4.47	7.62	3	1	112
F21		3.07	3.52	3	1	33	4.56	7.64	3	1	125
S22		2.39	2.02	2	1	29	5.00	9.30	3	1	170
F22		2.44	2.11	2	1	33	4.36	7.09	3	1	113
S23		2.27	1.78	2	1	17	4.80	8.15	3	1	122
F23		2.48	2.27	2	1	23	4.25	6.27	3	1	60
S24		2.28	1.54	2	1	12	5.20	9.04	3	1	82
F19	**P2**	2.82	2.83	3	1	27	6.19	11.38	3	1	120
S20		2.60	2.44	3	1	44	6.76	13.58	3	1	164
F20		2.39	2.10	2	1	39	5.48	10.26	3	1	167
S21		2.32	1.92	2	1	21	4.85	8.15	3	1	162
F21		2.87	3.13	3	1	38	5.45	9.56	3	1	115
S22		2.49	2.27	2	1	26	5.13	9.75	3	1	155
F22		2.44	2.10	2	1	28	5.02	8.97	3	1	114
S23		2.24	1.67	2	1	16	3.35	4.35	3	1	44
F23		2.58	2.74	2	1	31	5.07	9.16	3	1	102
S24		2.15	1.10	2	1	5	3.38	4.78	3	1	35

Differences between student group types

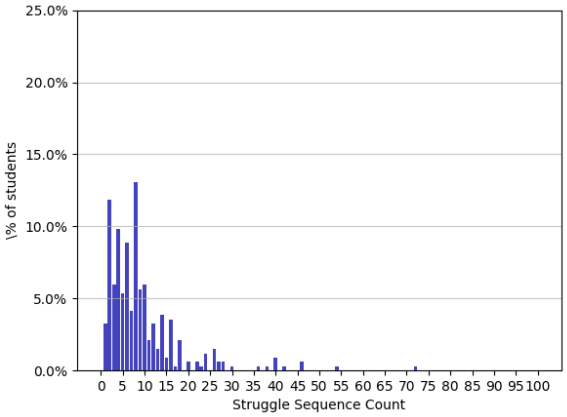
In each semester, students could choose to implement Project 2 individually or in pairs. Therefore a *student group type* can be either *individual* or *pair*. Since students can choose to be individual or in pairs only for Project 2, we only consider Project 2 when comparing struggle sequences between pair and individually implemented projects. Distributions of struggle sequence counts of projects completed in teams and projects completed individually, across semesters, can be seen in Figures 3.12 and 3.13.

Differences in global and local struggle sequences count for teams and individually implemented projects can be seen on Table 3.5, and differences in struggle sequence lengths between team and individually implemented projects can be seen in Table 3.6.

Pairs tend to struggle more often and present higher struggle sequences, however their results present a larger variance than individual students. Spring and Fall of 2020 present higher count and length of local struggle sequences when comparing student groups between semesters.

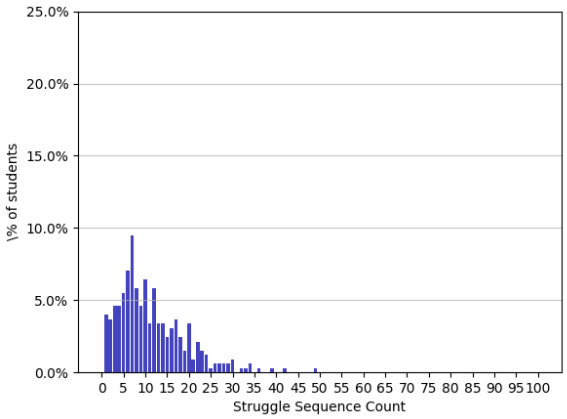


(a) Individual Global Struggle

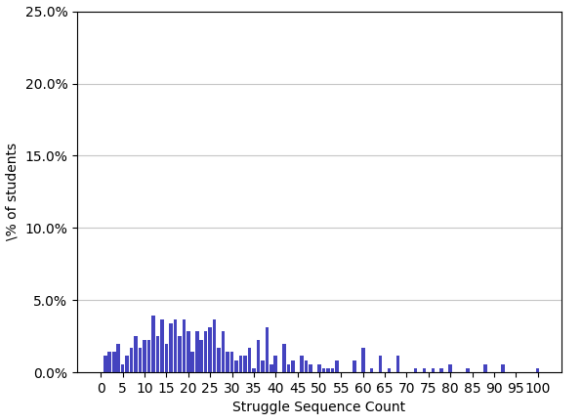


(b) Teams Global Struggle

Figure 3.12: Global struggle sequence distributions for projects completed individually and projects completed in teams across semesters.



(a) Individual Local Struggle



(b) Teams Local Struggle

Figure 3.13: Local struggle sequence distributions for projects completed individually and projects completed in teams across semesters.

Table 3.5: Counts of struggle sequence in each repository per group type in each semester.

		Global struggle count					Local struggle count				
Semester	**Project**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**
F19	**Individual**	3.29	2.00	3	1	9	9.20	6.32	7	1	28
S20		4.97	3.47	4	1	17	15.24	9.16	13.5	2	49
F20		3.90	3.47	3	1	14	19.26	7.98	19	5	42
S21		3.67	3.14	2.5	1	15	11.18	7.00	10	2	36
F21		2.96	1.75	3	1	9	10.35	6.17	9	1	30
S22		3.00	2.22	2	1	9	10.60	6.75	8	3	32
F22		2.67	1.92	2	1	10	12.71	6.81	12	2	39
S23		1.33	0.71	1	1	3	5.20	3.48	5	1	16
F23		1.00	0.00	1	1	1	3.93	3.41	3	1	12
S24		1.00	0.00	1	1	1	3.50	3.33	2	1	9
F19	**Pair**	8.27	5.73	7	2	26	18.39	15.58	13	3	84
S20		20.26	14.90	16	2	72	49.83	41.02	34	4	228
F20		9.78	8.22	8	1	54	39.20	25.16	34	1	146
S21		6.83	5.41	6	1	28	24.22	15.83	20	9	92
F21		6.87	4.97	6	1	30	22.14	14.41	19	3	80
S22		14.26	11.15	10	2	46	38.79	19.04	38	6	92
F22		7.16	4.92	6	2	18	27.91	15.30	23	10	72
S23		4.38	3.05	3.5	1	12	13.95	9.23	12	2	36
F23		6.85	4.96	6	1	24	22.54	19.50	19	1	100
S24		2.67	1.15	2	2	4	8.29	4.35	8	3	15

Table 3.6: Struggle sequence length per repository by grouping type in each semester.

		Global struggle length					Local struggle length				
Semester	**Project**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**	**Average**	**Std. Dev**	**Median**	**Min**	**Max**
F19	**Individual**	2.73	2.51	3	1	18	5.06	7.76	3	1	78
S20		2.54	2.95	2	1	44	5.47	10.28	3	1	109
F20		2.31	1.82	2	1	20	5.24	10.86	3	1	167
S21		2.23	1.82	2	1	21	4.99	9.27	3	1	162
F21		2.44	2.10	2	1	15	4.36	6.28	3	1	61
S22		2.15	1.48	1	1	8	3.54	5.22	3	1	48
F22		2.19	1.52	2	1	9	4.32	6.86	3	1	83
S23		1.92	1.02	1	1	4	2.57	2.76	2	1	24
F23		1.67	1.03	1	1	3	1.94	1.10	1	1	9
S24		2.00	1.00	2	1	3	2.57	2.00	3	1	12
F19	**Pair**	2.85	2.92	3	1	27	6.65	12.53	3	1	120
S20		2.61	2.31	3	1	28	7.14	14.39	3	1	164
F20		2.42	2.19	2	1	39	5.60	9.96	3	1	142
S21		2.36	1.97	2	1	17	4.78	7.52	3	1	93
F21		3.01	3.38	3	1	38	5.87	10.53	3	1	115
S22		2.53	2.34	2	1	26	5.52	10.53	3	1	155
F22		2.52	2.25	2	1	28	5.40	9.91	3	1	114
S23		2.29	1.75	2	1	16	3.74	4.92	3	1	44
F23		2.59	2.75	2	1	31	5.28	9.43	3	1	102
S24		2.17	1.13	2	1	5	3.67	5.42	3	1	35

3.2.2 Productive and Unproductive Struggle

To classify struggle between productive and unproductive, we first classify the commits in the struggle sequences as productive or unproductive using the progress indicators created by Edwards and Li (2016), and replicated in our first investigation of progress shown in Section 2.2. However, since not all progress indicators fit well with our course structure, we first eliminate the irrelevant indicators and then create new composite measures to arrive at the number of indicators a productive commit would trigger. The indicators eliminated, along with an explanation of why each indicator was eliminated, are:

General Purpose Indicators:

1. Adding New Solution Method(s): Solution methods are determined by the TS project design, and therefore this indicator is only triggered when students are constructing their implementation skeleton on the first few commits.

2. Removing Static Analysis Errors: Static Analysis Errors also occur mostly in the beginning of project development.

3. Increasing Solution Classes: Just like solution methods, solution classes are determined by the TS project design, and therefore this indicator is also only triggered when students are constructing their implementation skeleton on the first few commits.

Software Testing Indicators:

4. Adding New Test Method(s): as with solution methods, test methods are pre-determined by the TS project design. The number of test methods changes mostly during the first phase of project development, which is not a part of the struggle exploration.

5. Increasing Number of Tests per Method: as the TS project design dictates, there is one test method per class method, and this indicator is seldom triggered at this phase of the project.

6. Increasing Test Classes: test classes are also pre-determined by the TS project design.

With those indicators eliminated, we now consider the following nine indicators – since distributions did not show significant differences between semesters, we collect indicators across semesters:

General Purpose Indicators:

1. Reducing Cyclomatic Complexity: the student's solution contains fewer branches and/or methods than before.

2. Reducing Average Method Size: the average number of statements per method has been reduced.

3. Increasing Comments Density: the proportion of comment lines, relative to the number of source code lines, has increased.

4. Increasing Correctness: the number of instructor provided reference tests passed by the student's solution is more than before (the solution's behavioral correctness has increased).

Software Testing Indicators:

5. Adding to Existing Tests: the number of statements in the student's test suite is greater than before, while the solution remains approximately the same size.

6. Increasing Statement Coverage: the student's own software tests execute a larger number of statements in the solution than before.

7. Increasing Method Coverage: the student's own software tests execute a larger number of methods in the solution than before.

8. Increasing Conditional Coverage: the student's own software tests execute a larger number of conditional branches in the solution than before.

9. Increasing Assertion Density: the student's own software test execute a larger number of assertions per methods in test than before.

As our exploration consider both solution implementation and test implementation metrics, we consider general and test purpose indicators together, differently than the original work by Edwards and Li (2016), when calculating the new composite measure and determining how many indicators a productive commit would trigger. The composite measure of the selected indicators can be seen in Figure 3.14.

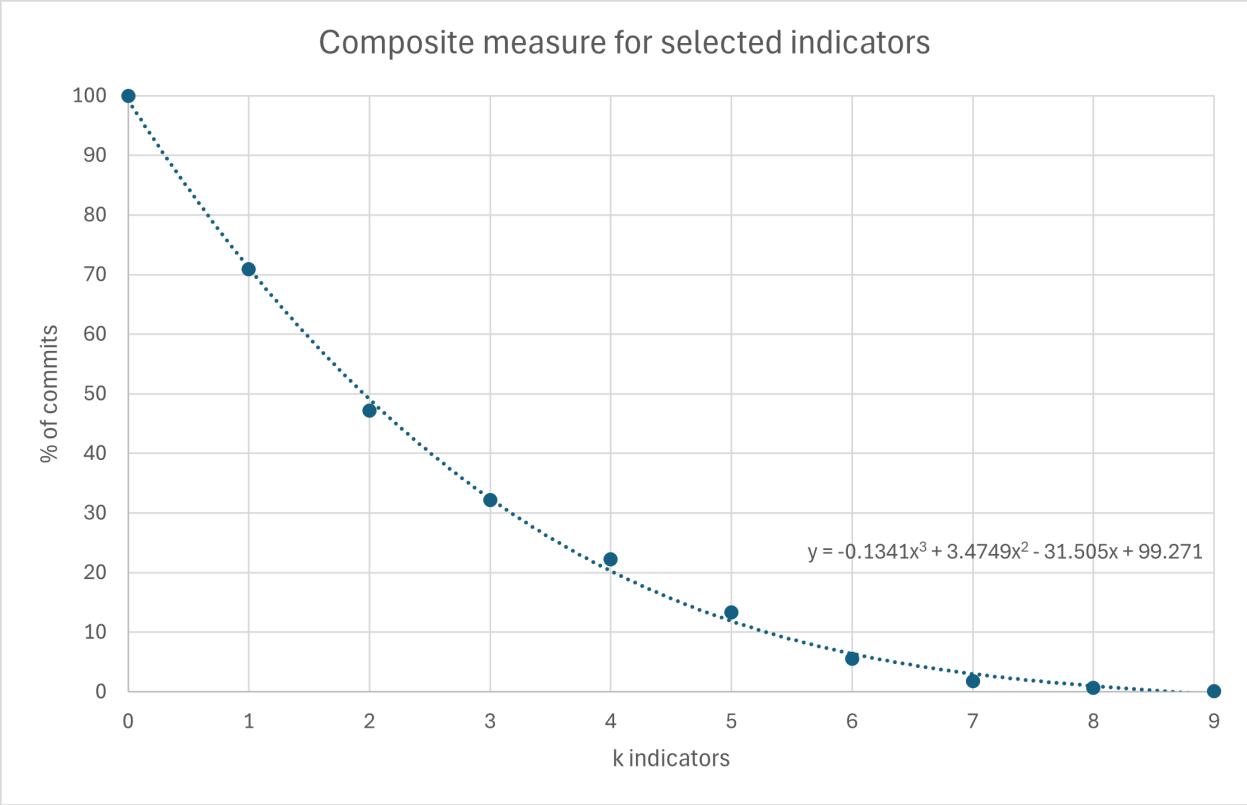


Figure 3.14: Composite measure of indicators selected for struggle exploration.

Using the selected indicators, a productive commit would trigger 3 to 5 indicators. The average percentage of productive commits per repository for each semester when using those values can be seen in Table 3.7. When using only indicators that align with our course structure, the percentage of productive commits is higher compared to using all the original indicators.

Table 3.7: Average percentage of productive commits per repository based on indicators selected for struggle exploration.

Semester	Commit count	% compiling commits	% productive commits
F19	127	82.6%	23.8%
S20	164	80.6%	24.0%
F20	94	78.2%	22.8%
S21	80	79.2%	22.0%
F21	74	75.6%	15.7%
S22	80	74.3%	23.1%
F22	72	78.0%	20.8%
S23	78	77.7%	24.0%
F23	71	76.3%	17.0%
S24	80	78.0%	23.6%

As mentioned previously, we classify a struggle sequence as productive if at least 60% of the commits in the struggle sequence were productive. With that boundary, we show the average percentage of productive struggle sequences in each semester, and the average percentage of struggle commits per struggle sequence, in Table 3.8, for both global and local struggle. Even though, on average, the number of productive commits per sequence is higher when considering local struggle sequences, the number of productive sequences, on average, is higher when considering global struggle sequences.

Table 3.8: Average percentage of productive commits per struggle sequence and productive struggle sequences per repository in each semester.

	Averages - Global Struggle		**Averages - Local Struggle**	
Semester	**% of productive commits per struggle sequence**	**% of productive struggle sequences**	**% of productive commits per struggle sequence**	**% of productive struggle sequences**
F19	25.2%	13.4%	28.4%	8.8%
S20	27.0%	15.5%	29.5%	10.2%
F20	24.7%	13.8%	30.9%	10.2%
S21	26.5%	15.8%	29.2%	10.8%
F21	13.1%	6.5%	22.0%	8.0%
S22	23.6%	12.2%	30.0%	12.0%
F22	17.3%	11.1%	27.9%	11.7%
S23	25.9%	16.8%	31.7%	13.2%
F23	14.1%	8.0%	21.5%	8.2%
S24	25.2%	17.9%	29.8%	14.2%

3.2.3 Struggle and grades

To determine whether our definition of struggle could offer any insight on the grade a submission was going to achieve, we calculate Spearman's correlation (Chambers 1989) between each type of struggle and project grades. Correlations-p-value pairs can be seen in Table 3.9, and statistically significant – when considering alpha equal 0.05 – correlations are in red. All correlations are small. However, all statistically significant values have negative correlations, which could signify that students that struggle more often or for longer achieve lower grades.

Table 3.9: Correlation and p-value pairs for each type of struggle and project grades, per project and semester.

Semester	Project	Global Count	Local Count	Global Length	Local Length
F19	P1	(-0.1126, 0.3295)	(-0.0722, 0.5163)	(0.0932, 0.4201)	(0.0519, 0.6413)
	P2	(0.019, 0.8537)	(-0.0859, 0.398)	**(-0.2345, 0.0208)**	(-0.0084, 0.9345)
S20	P1	(0.0538, 0.6336)	(-0.0127, 0.9083)	(-0.0219, 0.8458)	(0.1815, 0.0965)
	P2	(0.1238, 0.271)	(0.0488, 0.6593)	(-0.0903, 0.4227)	(0.1058, 0.3381)
F20	P1	(-0.1868, 0.085)	(-0.0719, 0.4936)	(-0.1011, 0.3541)	**(-0.2099, 0.0434)**
	P2	(0.0079, 0.9395)	(0.1814, 0.0754)	(-0.0756, 0.4691)	(-0.0692, 0.5007)
S21	P1	(0.0835, 0.4262)	(-0.0722, 0.4711)	(-0.1262, 0.2281)	(-0.1247, 0.2118)
	P2	(-0.0062, 0.9531)	(-0.0407, 0.6843)	(-0.0285, 0.7861)	(-0.1242, 0.2136)
F21	P1	(-0.0587, 0.5369)	(-0.107, 0.2426)	(0.0501, 0.5983)	(-0.0489, 0.5942)
	P2	(0.0464, 0.6076)	(0.2114, 0.0154)	(-0.0077, 0.932)	(0.0173, 0.8441)
S22	P1	(0.1313, 0.2397)	(0.1806, 0.0903)	(-0.0774, 0.4894)	(0.0102, 0.9241)
	P2	(-0.1881, 0.1218)	(0.0357, 0.7501)	**(-0.2894, 0.0159)**	(0.0844, 0.4507)
F22	P1	(-0.0792, 0.4686)	(-0.0184, 0.8623)	**(-0.2204, 0.0414)**	(-0.1657, 0.1165)
	P2	(-0.0515, 0.6767)	(0.0903, 0.4259)	**(-0.2478, 0.0416)**	**(-0.3481, 0.0016)**
S23	P1	**(-0.3319, 0.0186)**	(-0.1366, 0.3199)	**(-0.3208, 0.0231)**	(-0.2147, 0.1154)
	P2	(0.2208, 0.241)	(0.0439, 0.7413)	(-0.1231, 0.5169)	**(-0.2672, 0.0408)**
F23	P1	(-0.1901, 0.2006)	(-0.12, 0.3875)	**(-0.3148, 0.0312)**	(-0.2366, 0.0849)
	P2	(-0.0997, 0.5748)	(-0.0334, 0.8086)	(-0.0633, 0.722)	(-0.0682, 0.6209)
S24	P1	(-0.2119, 0.5567)	(-0.2517, 0.4552)	(0.1216, 0.7379)	(0.2818, 0.4011)
	P2	(-0.6325, 0.3675)	(0.1765, 0.5462)	(0.4, 0.6)	(0.4074, 0.1483)

3.2.4 Discussion

Our definition of struggle is based on how often and for how long students struggle with persistent errors when implementing programming projects. Since global struggle is collected across implementation classes, and local struggle is collected for each implementation class, there are more local struggle sequences than global struggle sequences per repository, as expected. As a result, larger portion of students present higher counts of local struggle sequences than global struggle sequences. Local struggle sequences tend to be longer than global struggle sequences; given the way our project is structured, with one manager class using all other classes, it is likely that the local struggle sequences for manager classes would last until students are further in the project development. However, as we discuss in Section 3.3.7, some students attempt to work on the manager classes early on.

Although distributions were similar between semesters, Spring and Fall of 2020 show higher struggles than other semesters. Due to the COVID-19 pandemic, students moved to hybrid/online classes in the middle of the Spring 2020 semesters, which explains why struggle was higher on Project 2 in that semester. During Fall of 2020 students were still online. This seems to indicate not only the transition between in-person and online classes affected the students, but also full-online classes seem to be difficult for students. However, the standard deviation on those semesters was also high, which could mean some students did better in the online environment. As shown in our effort exploration, students frequented help-seeking more often during semesters where classes were online. That was likely because online office-hours add accessibility, but also students were likely dealing with personal issues brought by the pandemic and needed the extra help. Overall, Fall of 2019 and Fall of 2021 presented less struggle sequences than students in the other semesters.

For both struggle sequences count and length, median values are similar between semesters. Project 2 presents more sparse distributions in every semester, with a larger amount of students showing struggle more often and in longer sequences, likely because the project is more complex than the first one. That could also be caused by fatigue and time-management issues since Project 2 is later in the semester. Max values generally happen during Project 2, with the exception of the max number of struggle sequences during Spring of 2022, which happened in Project 1.

Different than expected, in every semester teams tend to have a more sparse distribution of struggle, with high struggle sequences counts, and higher struggle sequences – however, students in teams also produce more code changes, which is discussed in Chapter 4. Higher struggle in teams could come from having two students with different coding and communication styles learning to work together, but it could also come from the fact Project 2 tends to be more complex than Project 1.

Additionally, productive commits did not change during struggle, as the percentage of productive commits within struggle sequences was relatively similar to productive commits over the entire project. Less than a quarter of commits tend to be productive commits, which might contribute for the occurrences of struggle. Since project instructions are well detailed, this might mean students are not following given instructions as expected.

We calculate correlations between struggle count and length, and despite the ceiling-effect present in the data, small negative correlations appear between struggle length and grades. Therefore, long occurrences of struggle could be an indicator that a student would achieve a lower grade. It would be necessary to do the same struggle analysis on a different data set where grades present a less skewed distribution.

3.3 Different patterns during struggle (RQ1.2)

To examine patterns during struggle, we analyze different implementation patterns within *struggle groups*. Struggle groups are sets of repository grouped by the count and length of their struggle sequences. By grouping repositories with similar metrics, we could be able to find coding behaviors that might be helpful or unhelpful when dealing with coding struggle.

3.3.1 Creating Struggle Groups

In order to compare student actions, we separate repositories according to the count and length of struggle sequences. For each semester, we calculate the range of struggle sequence counts, and the range of struggle sequence lengths and divide the range into approximately equal thirds – range being the maximum count or length value minus the minimum count or length value for the struggle sequences in a semester. That process generated three groups of struggle counts – low count, medium count, and high count –, and three groups of struggle lengths – low length, medium length, and high length. We explore global and local struggle, and counts and lengths, separately, so any student will be in four different groups depending on the count and length of their global and local struggle sequences. The percentage of students in each struggle group can be seen on Table 3.10. As struggle distributions showed previously, most students present low counts of struggle, and low length struggle sequences. Groups with high counts and lengths of struggle have fewer students, in most semesters, when compared with low count and length groups.

Table 3.10: Percentage of students in each struggle group in each semester.

	Global Struggle (%)						Local Struggle (%)					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	85	11	4	89	9	3	77	18	5	72	25	3
S20	83	15	3	89	8	3	66	31	3	58	33	9
F20	94	5	1	81	17	2	76	19	6	76	22	2
S21	86	11	3	79	18	3	60	19	21	66	27	7
F21	90	7	3	85	12	3	80	19	2	75	22	3
S22	89	9	2	80	18	2	88	8	4	72	23	4
F22	81	10	9	79	16	5	85	12	3	74	21	5
S23	93	5	3	81	12	7	90	6	4	83	13	4
F23	94	4	2	93	6	1	90	7	4	88	8	3
S24	94	2	4	91	5	5	91	2	7	91	5	4

3.3.2 Source and test code implementation during struggle

We explore source and test code implementation during struggle by collecting, per commit, numbers of code lines and code files changed, number of commits that present source and test development, and number of commits that present *related* source and test code development. Additionally, we collect the order in which students implemented their classes – as shown in Section 2.4.1, students with lower grades tend to modify files in a different order than students with higher grades. We also explore struggle productivity by using progress indicators introduced in 2.2.

Differences in coding behavior between struggle groups

We first explore the source and test code differences in each commit between groups. For each group, we collect the percentage of commits per repository that show source code implementation, test code implementation, source and test code implementation, and *related* source and test code implementation. Related source and test code implementation means the commit shows modifications on a source class and its test(s). For each commit per repository, we also collect number of source classes altered, number of test classes altered, and number of lines inserted and deleted for source and test code. The average of those metrics per repository for each group in each semester can be seen on Tables 3.11 through 3.18. We show one table for each metric to facilitate comparison between groups.

Percentage of commits containing source code (Table 3.11) is similar between groups. High struggle count groups show slightly more commits containing source code changes than low struggle count groups, while high struggle length groups show slightly less commits containing source code changes than low struggle length groups. Percentage of commits containing test code (Table 3.12) presents a larger variation between high and low struggle groups. Additionally, all low struggle groups present a higher percentage of commits with test code than high struggle groups.

Table 3.11: Average percentages of commits per repository that contain source code changes for each struggle group.

	Global Struggle (%)						Local Struggle (%)					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	79	86	92	80	81	76	86	90	95	87	86	75
S20	82	85	78	81	84	89	89	88	88	89	89	89
F20	80	76	88	80	78	81	88	87	91	88	86	86
S21	81	87	81	81	79	85	88	91	90	89	88	90
F21	79	83	86	79	83	80	88	86	88	88	86	84
S22	81	86	95	82	85	76	90	90	93	90	90	93
F22	77	78	78	79	74	60	85	87	89	87	86	79
S23	77	80	85	78	87	63	88	87	87	89	85	90
F23	76	80	78	78	76	76	88	88	82	90	85	81
S24	80	74	78	84	90	73	91	92	84	92	84	88
Total Average	**79**	**81**	**84**	**80**	**82**	**76**	**88**	**89**	**89**	**89**	**86**	**86**

Table 3.12: Average percentages of commits per repository that contain test code changes for each struggle group.

	Global Struggle (%)						Local Struggle (%)					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	45	32	31	44	38	45	43	35	32	44	38	42
S20	45	42	43	47	42	31	43	41	46	44	44	40
F20	49	46	25	50	47	40	48	45	41	49	42	43
S21	45	37	44	50	46	32	48	36	32	47	41	43
F21	48	39	36	48	43	51	47	43	40	48	44	41
S22	45	36	23	43	38	49	47	37	24	48	40	24
F22	51	47	47	50	48	65	58	45	41	58	48	45
S23	42	40	43	44	28	51	47	45	45	49	43	41
F23	53	40	53	52	49	49	49	42	44	49	45	46
S24	56	52	47	66	38	49	54	39	40	57	43	41
Total Average	**48**	**41**	**39**	**49**	**42**	**46**	**48**	**41**	**38**	**49**	**43**	**41**

Percentages of commits containing both source and test code (Table 3.13) are higher for low struggle groups, following the same pattern of commits containing test code. When looking at percentages of commits with related source and test code *within* commits containing source and test code (Table 3.14), the pattern reverses, as low struggle groups tend to present less commits where their source and test changes are related.

Table 3.13: Average percentages of commits per repository that contain source and test code changes for each struggle group.

	Global Struggle (%)						Local Struggle (%)					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	25	21	24	26	22	23	31	26	28	33	27	21
S20	30	28	23	31	28	21	34	30	35	34	34	29
F20	31	26	14	32	28	22	37	33	33	39	30	30
S21	28	24	25	32	27	20	37	27	23	37	31	33
F21	31	24	24	30	29	32	37	31	29	38	32	28
S22	28	23	18	27	26	29	38	28	18	40	31	19
F22	31	27	28	31	26	27	45	34	32	47	35	26
S23	23	24	31	27	17	22	36	35	35	39	30	34
F23	31	22	35	31	28	30	39	32	26	40	32	32
S24	36	27	30	49	28	25	48	31	26	53	29	30
Total Average	**29**	**25**	**25**	**32**	**26**	**25**	**38**	**31**	**28**	**40**	**31**	**28**

Table 3.14: Average percentages of commits that contain both source and test code where the source and test code is *related*.

	Global Struggle (%)						Local Struggle (%)					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	63	78	75	58	79	90	80	84	79	79	84	79
S20	72	76	78	66	81	83	82	82	81	80	85	81
F20	72	77	86	71	75	74	82	86	88	83	82	89
S21	65	79	83	62	73	73	85	82	85	86	82	85
F21	36	49	67	37	43	42	47	65	66	54	44	51
S22	65	81	77	63	76	90	80	85	84	78	85	92
F22	48	61	49	53	40	24	55	59	65	56	58	54
S23	53	89	91	59	56	83	61	82	87	60	85	86
F23	45	50	55	47	43	51	45	60	46	46	51	51
S24	50	58	100	63	100	65	59	76	84	60	82	83
Total Average	**57**	**70**	**76**	**58**	**67**	**67**	**68**	**76**	**76**	**68**	**74**	**75**

Number of source and test code classes altered in each commit (Tables 3.15 and 3.16) are similar for all groups, with each commit modifying around 1 source class and/or 1 test class.

Table 3.15: Average number of source classes altered in each commit per repository for each struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	1.16	1.19	1.20	1.15	1.24	1.14	1.22	1.20	1.12	1.24	1.18	1.14
S20	1.27	1.22	1.14	1.23	1.32	1.14	1.34	1.20	1.16	1.44	1.22	1.19
F20	1.26	1.19	1.13	1.28	1.21	1.18	1.37	1.22	1.19	1.39	1.21	1.17
S21	1.23	1.16	1.20	1.22	1.28	1.16	1.34	1.20	1.16	1.36	1.24	1.25
F21	1.30	1.24	1.19	1.30	1.27	1.21	1.41	1.37	1.24	1.45	1.32	1.23
S22	1.22	1.17	1.16	1.17	1.19	1.64	1.31	1.21	1.18	1.33	1.24	1.16
F22	1.25	1.37	1.30	1.25	1.36	1.46	1.50	1.37	1.25	1.54	1.34	1.22
S23	1.11	1.24	1.19	1.09	1.27	1.20	1.25	1.26	1.22	1.29	1.21	1.20
F23	1.31	1.28	1.42	1.31	1.36	1.24	1.35	1.31	1.32	1.33	1.34	1.43
S24	1.68	1.22	1.11	2.00	1.20	1.16	1.44	1.33	1.11	1.47	1.22	1.19
Total Average	**1.28**	**1.23**	**1.20**	**1.30**	**1.27**	**1.25**	**1.35**	**1.27**	**1.20**	**1.38**	**1.25**	**1.22**

Table 3.16: Average number of test classes altered in each commit per repository for each struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	1.15	1.00	1.00	1.09	1.21	1.33	1.12	1.00	1.00	1.16	1.00	1.00
S20	1.03	1.00	1.00	1.01	1.06	1.00	1.07	1.00	1.00	1.14	1.00	1.00
F20	1.17	1.00	1.00	1.22	0.99	1.12	1.13	1.03	1.00	1.15	1.02	1.00
S21	1.09	1.04	1.10	1.15	1.09	0.99	1.10	1.04	1.20	1.13	1.03	1.00
F21	1.15	1.23	1.00	1.16	1.13	1.00	1.12	1.07	1.00	1.16	1.01	1.00
S22	1.07	1.00	1.00	1.04	1.14	1.10	1.14	1.12	1.00	1.17	1.09	1.00
F22	1.26	1.05	1.00	1.21	1.20	1.00	1.16	1.03	1.00	1.17	1.04	1.00
S23	0.93	1.00	1.00	0.92	1.00	1.00	0.97	1.13	1.00	1.00	1.07	1.04
F23	1.11	1.00	1.00	1.05	1.19	1.00	1.01	1.00	1.00	1.00	1.00	1.05
S24	1.29	1.00	1.00	1.50	1.00	1.00	1.07	1.00	1.00	1.08	1.17	1.00
Total Average	**1.13**	**1.03**	**1.01**	**1.13**	**1.10**	**1.05**	**1.09**	**1.04**	**1.02**	**1.12**	**1.04**	**1.01**

Low count struggle groups tend to change more source code lines per commit than low struggle groups, with the exception of the global struggle length groups where the global struggle high length group changed more source lines per commit than the global struggle low length group. However, the differences between groups are small, and all groups modify a similar amount of source code lines per commit.

Table 3.17: Average number of source code lines changed in each commit per repository, for each struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	14.7	18.2	17.7	14.6	18.0	11.3	20.6	20.7	19.5	20.8	20.1	21.8
S20	24.0	14.5	16.3	15.8	31.7	17.4	21.1	13.2	23.3	21.3	17.1	29.3
F20	17.8	22.8	13.5	17.7	20.0	15.1	19.0	16.7	17.6	19.5	15.8	19.1
S21	19.9	12.2	17.3	19.9	19.7	15.7	20.8	13.2	12.9	21.0	16.5	13.7
F21	20.1	16.1	22.4	20.8	17.8	13.2	23.0	20.7	28.5	26.1	16.8	16.1
S22	23.5	14.8	10.2	13.2	15.4	114.7	20.3	15.0	15.8	16.0	23.0	13.5
F22	26.4	18.4	20.8	26.4	18.2	13.4	28.1	20.4	18.7	29.7	21.1	12.7
S23	12.9	16.1	12.4	13.1	14.7	12.7	15.0	22.2	22.7	17.1	21.8	16.6
F23	14.4	16.3	20.0	13.8	17.2	16.8	22.0	17.1	11.4	20.8	20.3	22.5
S24	7.1	16.7	10.4	5.8	13.2	10.7	15.1	11.0	13.0	14.4	16.5	14.8
Total Average	**18.1**	**16.6**	**16.1**	**16.1**	**18.6**	**24.1**	**20.5**	**17.0**	**18.4**	**20.7**	**18.9**	**18.0**

Table 3.18: Average number of test code lines changed in each commit per repository, for each struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	27.7	27.6	25.4	26.5	31.7	24.5	40.1	36.1	22.3	40.3	37.0	41.9
S20	27.0	24.7	31.3	25.4	28.6	27.6	31.9	23.7	28.7	32.7	29.2	31.1
F20	31.9	20.4	27.2	31.3	29.7	35.2	35.5	31.8	30.0	38.5	26.7	23.1
S21	30.8	22.2	32.6	32.2	30.9	24.4	38.6	31.9	24.5	41.1	31.8	28.8
F21	32.7	39.5	25.2	35.4	28.8	16.8	38.2	30.5	35.5	42.6	26.0	26.6
S22	31.4	16.6	12.7	27.8	21.9	51.3	36.2	25.3	23.2	36.6	29.8	21.2
F22	26.6	34.0	27.6	27.1	27.9	40.4	40.9	29.9	27.1	42.2	30.2	29.2
S23	20.0	22.3	22.0	18.0	23.7	28.2	22.0	29.6	27.4	24.3	24.6	32.9
F23	31.3	24.3	22.9	28.2	29.9	33.5	33.5	27.3	25.9	30.4	32.5	42.8
S24	4.2	28.5	39.9	19.5	52.8	12.8	14.3	33.1	13.5	12.3	33.3	25.5
Total Average	**26.4**	**26.0**	**26.7**	**27.1**	**30.6**	**29.5**	**33.1**	**29.9**	**25.8**	**34.1**	**30.1**	**30.3**

Percentages of insertions and deletions within source code line changes were similar, with around 60% of the lines changes being insertions, and 40% of the lines changes being deletions for every struggle group. However, when looking the insertions and deletions within test code line changes, there was a consistent difference between groups. Test code line deletions can be seen in Table 3.19. Although test line insertions present a higher rate than test line deletions, all low struggle groups delete test code lines less often than high struggle groups. Conversely, high struggle groups present less test code line insertions than high struggle groups,

Table 3.19: Percentage of deletions among test line changes for each group.

	Global Struggle (%)						Local Struggle (%)					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	24	27	25	23	28	34	22	25	26	21	26	20
S20	30	33	40	29	33	32	26	32	39	25	29	28
F20	24	32	40	26	24	21	23	31	36	25	26	31
S21	26	36	46	29	21	33	22	28	41	21	27	33
F21	29	35	40	29	31	31	26	32	36	26	29	36
S22	30	32	35	29	24	39	26	31	33	22	34	34
F22	30	38	34	31	33	45	24	33	32	25	28	35
S23	29	37	21	28	29	36	20	26	27	20	26	29
F23	29	44	41	26	37	46	24	32	33	21	29	37
S24	44	31	23	25	15	38	33	22	38	32	24	35
Total Average	**30**	**34**	**35**	**27**	**27**	**36**	**25**	**29**	**34**	**24**	**28**	**32**

Classes implemented together

As students can alter multiple classes in the same commit, we explore how often pairs of classes are altered together, and compare paired-classes frequency between struggle groups. The recommendation is for students to follow Test-Driven Development (TDD) guidelines and alter a source class and the related test class together. Since classes are specific for each project, and we are analyzing how they work together, we do this exploration for each project.

The classes in each project are specific for the course. To increase generalizability we collect paired-classes frequency by creating *class types*. Class types are a representation of the main purpose of the class, and types are created based on classes used in the projects on every semester. Each class type is subdivided into source or test class. From our data, we extract the following types:

- Manager: the main class of the project, it uses all other classes/objects.

- Writer: class to write files, which can be used internally or to save information.
- Reader: class to read external files, either created by the writer class or independently created as input for the project.
- Data Structure: class to create a functional data structure, for example a linked list. Students are required to create their own data structures when implementing projects.
- State: class to create a finite state machine, one of the states used in a finite state machine, or to transition between the states of a finite state machine. This type is only used in Project 1.
- Object: Any object that will be used by the other classes but not general enough to be its own type. These object classes are generally specific for each project's theme.

As shown earlier in this section, students tend to implement one class at a time, so the percentages for each class pair are low for all groups. In Project 1 (Table 3.20), the class pair that appears the most in all groups is composed of a source state class and a source manager class. That shows students, when developing different classes simultaneously, tend to work on both sides of the project structure hierarchic at the same time. Additionally, there are more pairs with two source classes than pairs formed by one source and one test class. All pairs not shown on the table had percentages of 1% or less. In Project 2 (Table 3.21) students seem to implement related test and source classes more often, as those pairs are present in a higher percentage of commits for all struggle groups.

Table 3.20: Percentage of commits that each pair of class types was implemented together during Project 1.

	Global Struggle (%)						Local Struggle (%)					
Class type pair	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
src_state-src_manager	4	5	4	5	5	3	6	6	10	6	7	8
src_state-src_reader	4	5	4	4	4	5	5	6	4	7	5	5
src_state-src_state	2	2	2	3	2	1	2	2	2	2	2	2
src_state-src_writer	2	2	2	2	2	2	3	2	2	3	2	2
src_object-src_object	1	1	1	2	1	0	1	1	2	2	1	1
src_manager-src_writer	1	1	1	2	1	1	2	1	1	2	2	1
src_reader-src_writer	1	1	1	1	1	0	1	1	1	1	1	1
src_object-src_writer	2	1	2	2	2	1	2	2	1	2	1	1
src_manager-src_reader	1	1	1	1	1	1	2	1	1	2	1	1
src_manager-src_object	3	1	2	2	3	2	2	2	1	3	1	1
src_manager-test_manager	2	2	2	2	2	3	2	2	1	2	2	2
test_object-test_manager	2	1	1	1	1	1	1	1	1	1	1	1
test_manager-test_object	2	1	1	2	1	1	1	1	1	1	1	1

Table 3.21: Percentage of commits that each pair of class types was implemented together during Project 2.

Class type pair	Global Struggle (%)						Local Struggle (%)					
	Low Count	Medium Count	High Count	Low Length	Medium Length	High Length	Low Count	Medium Count	High Count	Low Length	Medium Length	High Length
src_object-test_object	9%	16%	9%	13%	9%	14%	15%	14%	11%	8%	15%	13%
src_data_structure-src_object	17%	6%	5%	19%	8%	7%	9%	8%	6%	7%	5%	22%
src_data_structure-test_data_structure	4%	5%	4%	3%	5%	6%	5%	8%	6%	4%	5%	7%
test_manager-src_manager	5%	3%	2%	4%	2%	5%	5%	3%	2%	4%	3%	2%
src_data_structure-test_object	3%	4%	2%	3%	3%	5%	3%	5%	2%	3%	3%	4%
src_object-src_reader	4%	1%	2%	4%	2%	2%	2%	2%	1%	2%	1%	4%
src_reader-test_reader	2%	1%	2%	2%	3%	2%	2%	1%	1%	1%	2%	1%
src_object-test_data_structure	2%	2%	2%	2%	2%	3%	2%	3%	2%	2%	2%	2%
test_object-test_data_structure	4%	4%	4%	4%	5%	5%	4%	4%	3%	4%	2%	4%
src_object-src_manager	4%	2%	2%	5%	2%	4%	4%	2%	2%	3%	2%	4%
test_manager-src_object	2%	1%	1%	1%	1%	2%	2%	1%	1%	2%	1%	1%
test_reader-src_reader	2%	1%	2%	2%	2%	2%	2%	2%	1%	2%	2%	1%
test_manager-test_object	2%	1%	1%	1%	1%	2%	1%	1%	1%	2%	1%	1%

3.3.3 Productivity during struggle

In addition to the previous metrics, we collect the number of productive commits and productive struggle sequences, and the number of struggle sequences for each repository per struggle group. We also collect the percentage of struggle sequences that end with a progress commits and the struggle sequences that end with non-compiling commits. To determine whether a productive commit or sequence was productive, we use the process discussed in Section 3.2.2. Average percentage of productive struggle sequences, and percentage of productive commits in each sequence, are shown in Tables 3.22 and 3.23, respectively. Average percentage of productive commits is shown in Table 3.24 – struggle sequences only end with a progress commit, or with a non-compiling commit, therefore the complementary values are the percentage of struggle sequences that ended with non-compiling commits.

When comparing groups of high and low struggle, both for struggle counts and struggle length, students in the low struggle groups present both a higher count of productive struggle sequences, and a higher count of productive commits within struggle sequences. That said, the percentage of productive struggle sequences is low for all groups, as in most semesters each group had less than a fifth of their struggle sequences being productive. Additionally, for most groups in most semesters, less than a quarter of the commits in their struggle sequences was considered productive.

Table 3.22: Average percentage of productive struggle sequences per struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	12	12	10	14	9	1	10	7	4	12	6	0
S20	13	12	12	14	12	6	7	10	13	8	8	6
F20	13	9	11	15	10	8	10	10	8	11	6	6
S21	15	15	12	19	16	7	11	10	9	13	8	7
F21	8	6	7	9	6	0	9	7	5	11	3	2
S22	14	10	3	16	9	0	17	9	3	20	8	3
F22	12	9	7	12	11	2	11	8	14	14	6	4
S23	14	18	4	17	9	9	17	12	10	20	10	4
F23	16	7	4	18	9	5	9	6	5	10	7	2
S24	0	13	9	3	6	6	30	11	5	30	11	2
Total Average	**12**	**11**	**8**	**13**	**10**	**4**	**13**	**9**	**8**	**15**	**7**	**3**

Table 3.23: Average percentage of productive commits within struggle sequences per struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	22.1	22	21.9	22.7	21.6	15.7	27.8	26.9	25	29.1	25.5	20.6
S20	22.4	23.6	22.1	21.8	23.9	21.8	24.9	27.9	31.5	24	27.4	23.4
F20	23.4	21.6	20.7	23.3	23.4	21.9	28.6	29.7	28.4	29.3	28	24.6
S21	24.2	25	20.6	25.9	27.6	18.8	28.2	26.5	24.1	28.7	25.8	28.6
F21	14.4	13.4	15.7	14.5	14.6	6.8	22.2	22.9	16.6	25	17.2	14.1
S22	24.2	20.7	15.8	24.1	22.3	19.4	32.1	25.9	19.7	33	27.6	19.9
F22	17.6	15.2	12.9	17	17.7	7.6	27.4	25.7	26.8	29.6	22.4	22.8
S23	20.9	28	14.1	22.2	18.4	23.7	25.8	32.3	28.3	28.8	28.6	26
F23	22.4	15.1	9.2	20.8	17.8	21.6	21.3	19.2	16.4	20.9	20.7	20.1
S24	14.9	24	18.6	17.2	11.7	19.1	39.1	20.8	23.1	37.4	25.3	19
Total Average	**20.7**	**20.9**	**17.2**	**21.0**	**19.9**	**17.6**	**27.7**	**25.8**	**24.0**	**28.6**	**24.9**	**21.9**

Most of the struggle sequences end with progress commits for every struggle group. When looking at local struggle, the high struggle length group have a higher rate of struggle sequences that end in progress than the low struggle length group; additionally, the high struggle count group has much lower rate of struggle sequences ending in progress than the low struggle count group.

Table 3.24: Average percentage of struggle sequences that lead to progress per struggle group.

	Global Struggle						Local Struggle					
Semester	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
F19	93	94	96	93	95	97	84	73	56	78	88	90
S20	95	94	96	96	92	98	81	73	72	79	80	85
F20	96	96	94	95	99	98	89	80	55	84	93	91
S21	98	99	97	97	99	99	93	88	85	90	94	95
F21	97	93	87	96	96	93	88	74	60	85	84	88
S22	95	97	95	96	94	89	88	87	76	85	91	90
F22	97	96	96	97	98	92	91	85	66	88	88	92
S23	95	92	93	96	91	91	94	87	83	92	87	90
F23	98	93	94	98	95	94	93	76	57	92	86	79
S24	100	100	98	100	100	99	100	100	92	100	96	95
Total Average	**96**	**95**	**95**	**96**	**96**	**95**	**90**	**82**	**70**	**87**	**89**	**89**

3.3.4 Different error patterns per struggle group

Error patterns are patterns that appear during struggle regarding which teaching-staff (TS) tests are failing and which TS tests are passing. For each struggle group, we collect the frequency of TS classes with tests that pass and fail together, and the frequency of TS classes with tests that alternate, e.g. one passes when the other fails, for each struggle group. We collect the pairs for each project, as Project 1 and Project 2 use different types of classes. There were no classes that never fail simultaneously, e.g. every class had at least one test failing with tests from another classes. "Together" in this context means they are failing/passing in the same struggle commit. There was also no class that fails by itself. By looking at these patterns, we might see differences between struggle groups that offer insight into why one group of students is struggling for longer or more often than another group. As when exploring pairs of classes implemented together, we use class types to increase the generalizability of the work.

TS classes that fail together

The average proportion of class pairs failing in the same struggle sequence during Project 1 can be seen on Table 3.25. The class pair that fails together more often is formed by two state classes. That makes sense given that in a finite state machine states are sequential, so one state class depends on the other. Pairs of object classes also fail together at a higher rate than the other pairs.

Table 3.25: Average proportion of class types failing together on Project 1.

	Global Struggle						Local Struggle					
Class type pair	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
state-writer	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%
state-state	38%	40%	41%	48%	38%	39%	39%	37%	42%	34%	43%	38%
object-object	4%	4%	4%	8%	3%	3%	4%	3%	4%	4%	4%	3%
state-reader	8%	11%	11%	7%	11%	11%	10%	11%	10%	10%	10%	11%
state-manager	0%	0%	0%	0%	1%	0%	0%	2%	0%	0%	0%	1%
manager-reader	3%	3%	3%	2%	3%	3%	3%	3%	3%	4%	2%	3%
writer-reader	3%	2%	3%	1%	3%	3%	2%	3%	2%	4%	2%	3%
writer-manager	4%	2%	3%	1%	3%	3%	3%	3%	2%	4%	2%	3%
writer-object	18%	14%	14%	7%	16%	16%	16%	16%	13%	17%	13%	16%
manager-object	18%	17%	17%	20%	16%	17%	18%	16%	18%	17%	18%	16%
reader-object	5%	7%	5%	5%	5%	5%	6%	5%	5%	8%	5%	5%

Table 3.26: Average proportion of class types failing together on Project 2.

	Global Struggle						Local Struggle					
Class type pair	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	Medium Length	**High Length**
object-object	4%	6%	2%	1%	7%	4%	8%	11%	3%	9%	5%	3%
data_structure-data_structure	19%	7%	15%	11%	6%	18%	6%	11%	18%	4%	10%	19%
manager-reader	0%	5%	0%	0%	6%	0%	5%	1%	0%	6%	0%	0%
data_structure-reader	13%	10%	15%	21%	8%	13%	8%	1%	15%	8%	10%	15%
data_structure-manager	0%	11%	0%	1%	9%	0%	10%	1%	0%	8%	1%	0%
writer-manager	0%	6%	0%	0%	6%	0%	4%	0%	0%	5%	0%	0%
writer-reader	4%	5%	9%	10%	5%	5%	4%	0%	6%	5%	5%	6%
manager-object	0%	8%	0%	1%	11%	0%	11%	1%	0%	10%	1%	0%
reader-object	7%	8%	11%	11%	10%	8%	9%	1%	9%	10%	5%	9%
data_structure-object	29%	18%	24%	22%	17%	28%	21%	42%	26%	18%	31%	26%
writer-data_structure	16%	10%	15%	21%	7%	15%	7%	21%	15%	8%	20%	15%
writer-object	9%	7%	11%	11%	9%	10%	8%	21%	9%	10%	15%	9%

TS classes that pass together

For both Project 1 (Table 3.27), and Project 2 (Table 3.26, the pairs that tend to fail together also tend to pass together. However, the pair between writer class type and data-structure class type do not pass together often even though it fails often. Classes tend to pass together less often for high struggle groups.

Table 3.27: Average proportion of class types passing together on Project 1.

	Global Struggle						Local Struggle					
Class type pair	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
state-state	36%	35%	44%	31%	45%	45%	36%	42%	45%	43%	41%	43%
object-object	9%	9%	11%	9%	9%	12%	9%	9%	12%	12%	11%	9%
state-writer	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
state-reader	10%	7%	9%	11%	12%	4%	9%	13%	6%	9%	7%	10%
state-manager	0%	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%
writer-manager	3%	4%	2%	4%	2%	2%	3%	2%	2%	2%	3%	2%
writer-reader	2%	1%	2%	3%	2%	1%	2%	2%	2%	0%	2%	1%
manager-reader	2%	1%	2%	3%	2%	1%	2%	2%	2%	2%	2%	2%
writer-object	12%	21%	11%	15%	12%	11%	17%	14%	9%	6%	12%	14%
manager-object	21%	18%	17%	20%	13%	20%	19%	12%	19%	25%	20%	14%
reader-object	4%	3%	3%	5%	4%	2%	4%	4%	3%	4%	3%	4%
writer-object	9%	7%	11%	11%	9%	10%	8%	21%	9%	10%	15%	9%

Table 3.28: Average proportion of class types passing together on Project 2.

	Global Struggle						Local Struggle					
Class type pair	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**	**Low Count**	**Medium Count**	**High Count**	**Low Length**	**Medium Length**	**High Length**
data_structure-data_structure	30%	8%	44%	7%	29%	36%	25%	8%	40%	8%	26%	42%
object-object	3%	17%	6%	16%	6%	4%	4%	16%	4%	17%	5%	4%
manager-manager	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
writer-manager	0%	0%	0%	1%	0%	0%	0%	1%	0%	1%	0%	0%
writer-reader	1%	0%	0%	1%	0%	1%	2%	1%	0%	1%	1%	0%
manager-reader	0%	3%	0%	2%	0%	0%	0%	3%	0%	2%	0%	0%
data_structure-reader	20%	8%	0%	7%	19%	12%	16%	8%	11%	7%	17%	10%
data_structure-manager	0%	7%	0%	7%	0%	0%	0%	8%	0%	7%	0%	0%
manager-object	0%	10%	0%	9%	0%	0%	0%	10%	0%	10%	0%	0%
writer-data_structure	4%	1%	0%	3%	0%	3%	8%	2%	0%	2%	6%	0%
writer-object	2%	1%	0%	5%	0%	2%	4%	2%	0%	3%	3%	0%
data_structure-object	32%	33%	0%	31%	40%	39%	33%	32%	42%	33%	35%	43%
reader-object	8%	12%	0%	10%	10%	5%	8%	11%	4%	10%	9%	2%

3.3.5 Implementation order beyond struggle commits

We explore the order in which classes were implemented beyond struggle commits, considering all commits after milestone M2 to see what classes are modified more often once students have feedback from TS tests, and differences in implementation order between struggle groups. The order of class modification is important due to the structure of the projects in our course. The project structure is in the form of a hierarchic, which is created by which classes use other classes. For example, manager type classes utilize all other types, and belong on the top of the hierarchic. Object type classes *are used* by other classes, and they are at the bottom of the hierarchic. Students should start their development by implementing/changing classes at the bottom of the hierarchic, as failing tests related to classes on the top of the hierarchic will likely be impacted by fixing failing tests related to classes on the bottom of the hierarchic. Therefore, we focus on understanding which class types student develop during the start of their post-M2 development by analyzing the first five commits after milestone M2 from each repository, with the goal of analyzing the students' code once they are trying to debug TS unit tests.

To obtain the order in which classes were altered, we calculate the percentage of repositories that altered each class type in each commit. For example, a class type with percentage of 25% on commit 1 means 25% of the repositories made changes in a class of that type on their first commit. Since classes are different in Project 1 and Project 2, we analyze both projects separately, for each struggle group.

We show the average percentage of repositories that altered each class t

Table 3.29: Percentage of repositories that on average altered each class type over the 5 first commits after milestones M2 when considering global struggle groups.

	Global Struggle - Project 1									
	Manager - Source	**Reader - Source**	**Writer - Source**	**Object - Source**	**State - Source**	**Manager - Test**	**Reader - Test**	**Writer - Test**	**Object - Test**	**State - Test**
Low count	87	11	6	21	5	79	9	7	20	5
Medium count	76	12	9	23	5	65	7	9	26	5
High count	71	13	7	21	2	57	8	10	22	2
Low length	92	10	6	18	4	81	8	6	18	4
Medium length	77	9	7	23	4	65	9	9	19	8
High length	57	19	10	36	11	62	12	13	38	5

Table 3.30: Percentage of repositories that on average altered each class type over the 5 first commits after milestones M2 when considering local struggle groups.

	Local Struggle - Project 1									
	Manager - Source	**Reader - Source**	**Writer - Source**	**Object - Source**	**State - Source**	**Manager - Test**	**Reader - Test**	**Writer - Test**	**Object - Test**	**State - Test**
Low count	89	11	6	19	5	81	8	7	18	4
Medium count	76	13	9	26	6	67	9	10	28	6
High count	69	10	5	23	5	61	7	8	20	7
Low length	85	12	8	24	6	77	9	8	22	5
Medium length	87	10	6	18	4	77	7	7	18	5
High length	73	11	7	21	3	70	11	10	20	2

Table 3.31: Percentage of repositories that on average altered each class type over the 5 first commits after milestones M2 on Project 2 when considering global struggle groups.

	Global Struggle - Project 2									
	Manager - Source	**Reader - Source**	**Writer - Source**	**Data Structure- Source**	**Object - Source**	**Manager - Test**	**Reader - Test**	**Writer - Test**	**Data Structure - Test**	**Object- Test**
Low count	15	15	9	43	69	13	9	7	31	50
Medium count	3	6	4	38	75	3	3	4	30	48
High count	8	8	5	58	74	8	5	5	46	51
Low length	16	14	9	46	67	14	9	7	33	52
Medium length	9	13	8	38	73	7	7	7	27	47
High length	7	13	7	34	78	8	7	8	31	47

Table 3.32: Percentage of repositories that on average altered each class type over the 5 first commits after milestones M2 on Project 2 when considering local struggle groups.

	Local Struggle - Project 2									
	Manager - Source	**Reader - Source**	**Writer - Source**	**Data Structure- Source**	**Object - Source**	**Manager - Test**	**Reader - Test**	**Writer - Test**	**Data Structure - Test**	**Object- Test**
Low count	14	16	10	44	72	12	10	9	32	54
Medium count	8	10	8	53	71	7	4	4	40	52
High count	8	6	5	65	61	7	3	5	45	40
Low length	16	16	10	47	72	14	11	9	35	58
Medium length	9	12	10	41	72	9	7	7	28	46
High length	4	15	6	52	70	5	6	9	40	44

3.3.6 Differences related to the COVID-19 pandemic

When observing the differences between semesters, students from Spring and Fall of 2020 presented higher struggle counts than all other semesters, especially on Project 2 of Spring of 2020m and more sparse distributions of struggle. The COVID-19 pandemic may have had a large participation on those results, as students were moved from in-person classes to online classes in the middle of the Spring 2020 semester. Students in Fall 2021 had less productive commits and less productive struggle sequences than other semesters.

In Spring of 2022, students have a high rate of struggle as well, although not as high as during the two previous semesters. Students that took our course in Spring of 2022 had their previous introductory courses in a hybrid environment, which may explain their difficulties when returning to campus full time. They also had high school courses likely online during the pandemic, so their struggles during high school might have impact their learning during this semester. The pandemic did not seem to affect the students' coding behavior, as those metrics are similar between all semesters.

3.3.7 Discussion

In this phase of the research, we compare implementation patterns within struggle sequences. We first divide students into struggle groups according to the count and length of their struggle sequences. We then divide students between struggle groups and analyze, within each group, the amount commits that contain source changes, test changes, source and test changes, and

source and test related changed. Source and test related changes happen when source code is modified along with its tests in the same commit. We then analyze the order in which students work on their files once they reach the point in implementation where they have access to TS tests. As the project classes are specific to our course, we create class types that can be used beyond our course.

Struggle groups were skewed, with the majority of the students presenting low count of struggle sequences and short struggle sequences. This affects the results as the analyzed behavior from high struggle groups comes from small sets of students.

Percentage of commits that show source class modifications is similar between groups. However, low struggle groups tend to develop test code at higher rates. This might indicate that low struggle groups present low struggle because they test their code more thoroughly or more often, not necessarily because they know what tests to write or what specific parts of the code to test. Additionally, high struggle groups present higher rates of test code deletion and higher rates of source-test related code changes, showing more test code churning, which might indicate they are trying to change their test code often to reproduce TS errors instead of writing their test code based on the projects descriptions.

Students tend to develop one class at a time. Class types that appear together more often are dependent on the project. In Project 1 classes being developed together are generally source classes, while in Project 2 students tend to develop more source and test classes together. That shows that by Project 2 they have understood better how to use test development. Classes that tend to fail and pass together also depend on each project. However, classes that tend to fail together also tend to pass together. That shows students tend to work on the classes they see from the TS feedback. Longer struggle sequences might mean they are trying to work on the classes showing the failure instead of looking at the classes the failures depend on.

As for the order of development, in Project 1 students tend to develop manager classes first. That is not a good approach, since the manager classes depend on all the other classes. In Project 2 students develop other types of classes first, showing they understand better how the projects' structure works; low struggle groups still implement manager source and test classes more often when starting development on Project 2, perhaps because that approach did not cause them to struggle as much during Project 1. High struggle groups tend to develop tests less often when starting development. That along with the other results shows high struggle students tend to work on tests once they get feedback from TS tests.

CHAPTER

4

STUDENTS' RESPONSE TO STRUGGLE (RQ2)

Having struggle defined, we moved to understand how students' coding behavior changes when dealing with persistent errors. Our second question, "How do students respond to struggle?" (RQ2), is divided into two sub-questions:

RQ2.1 "What are the differences in source and test code implementation during struggle?" (RQ2.1)

RQ2.2 "How do students use their own test cases during struggle?" (RQ2.2)

4.1 Differences in implementation (RQ2.1)

To capture the coding behavior, we use metrics based on our initial study on progress indicators. We focus on the metrics that seemed relevant in our assignments' structure: source lines count, test lines count, test failures, and the number of lines that were covered or missed by the students' test code. We explore differences in implementation per struggle group.

4.1.1 Comparing students' behavior before and during struggle

To explore whether students change their code behavior when facing struggle moments, we collect metrics that compare both their code behavior and their help-seeking behavior in different moments. Data used in these comparisons come from the period between milestones M2 and M3, to understand students' development once they have access to TS feedback. We collect coding metrics: number of struggle moments, total number of lines inserted and deleted before and after struggle for both source and test code, and number of source and test files altered. We also collect help-seeking metrics: number of piazza posts and average office hours attendance before and during moments of struggle. Coding metrics can be seen in Tables 4.1 through 4.8. Help-seeking metrics can be seen in Tables A.1 through A.8. We present each variable for low and high struggle groups defined on Chapter 3 with the goal of observing meaningful differences between groups.

Code lines and file changes for each struggle group

Students in the global low count struggle group (Table 4.1) tend to decrease source and test lines changes, and increase file changes during struggle. The ratio between source and test line changes decreases during struggle, showing they are working more on their tests and less on their source code while working on fixing errors.

Table 4.1: Line and file changes before and during struggle for repositories in the global low count struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	3	145	75	200	47	100	64	113	40	11	7	7	12
S20	6	168	95	169	62	164	123	151	63	18	9	10	21
F20	5	192	91	279	81	131	84	163	60	17	12	10	16
S21	4	118	56	166	52	78	52	104	34	13	8	6	11
F21	3	136	77	203	68	125	86	137	51	12	8	9	16
S22	4	132	66	181	53	86	68	112	46	13	8	6	11
F22	3	132	66	166	62	58	35	92	36	10	7	6	8
S23	3	101	46	128	46	49	28	49	19	9	6	3	6
F23	3	121	61	138	51	64	52	106	60	10	6	6	9
S24	2	23	15	21	11	17	9	4	3	7	4	2	5
Avg	4	127	65	165	53	87	60	103	41	12	7	7	11

During struggle students in the global high count struggle group (Table 4.2) increase their source and test line insertions and deletions, while decrease the number of files being changed,

showing they might be struggling with unproductive code churning.

Table 4.2: Line and files changes before and during struggle for repositories in the global high count struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	21	534	495	231	89	621	384	400	135	44	15	67	24
S20	32	686	582	402	234	579	540	658	304	60	32	74	46
F20	41	577	377	696	434	797	547	454	307	81	30	125	31
S21	20	389	276	333	145	433	290	355	286	47	24	54	32
F21	13	427	279	273	129	500	367	294	170	31	16	56	24
S22	36	430	219	319	68	510	401	180	96	74	21	102	28
F22	13	299	193	304	143	258	171	320	176	35	18	38	23
S23	12	281	128	186	44	172	126	152	58	30	13	28	14
F23	13	281	201	259	134	265	196	353	207	35	18	42	31
S24	7	170	102	213	109	79	59	216	63	21	14	16	11
Avg	21	407	285	321	153	421	308	338	180	46	20	60	26

Students in the global low length struggle group (Table 4.3) decrease both their source and test line changes, and their source and test file changes during struggle. Additionally, their test line insertions is higher than their source line insertions both before and during struggle, showing they focus on test development.

Table 4.3: Line and files changes before and during struggle for repositories in the global low length struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	4	162	92	168	37	84	65	93	27	13	7	11	6
S20	6	141	80	160	56	95	67	101	41	17	9	14	7
F20	6	197	101	294	87	114	71	137	53	18	12	14	9
S21	4	101	47	147	45	47	33	75	34	12	8	7	4
F21	4	143	80	200	62	111	78	120	47	13	9	13	7
S22	6	159	89	211	58	91	63	98	42	16	9	12	6
F22	4	164	87	180	70	91	58	114	53	14	9	11	8
S23	4	130	62	159	58	54	34	52	22	11	7	7	4
F23	4	117	67	151	54	59	37	76	37	12	8	8	5
S24	3	68	36	103	38	6	13	78	23	11	6	5	4
Avg	4	138	74	177	57	75	52	94	38	14	8	10	6

Students in the global high length struggle group (Table 4.4) tend to increase their source and test line insertions and deletions during struggle, while also increasing their source and test file changes. This shows they likely do not know which classes to work on in order to fix the errors they are getting. However, the difference between source and test code line changes diminishes during struggle.

Table 4.4: Line and files changes before and during struggle for repositories in the global high length struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	5	95	58	271	69	212	119	223	113	10	9	31	19
S20	12	314	170	235	55	704	552	435	134	38	12	82	27
F20	6	170	95	185	76	264	194	288	104	20	11	37	16
S21	7	184	126	202	77	225	165	193	108	25	11	32	14
F21	4	86	80	91	21	270	187	291	127	13	6	44	27
S22	6	265	108	215	95	372	373	323	126	20	10	37	18
F22	4	91	61	269	143	100	55	335	265	12	9	17	20
S23	4	142	45	131	48	76	46	110	72	13	8	13	11
F23	6	153	110	210	115	221	196	368	276	20	10	33	23
S24	4	63	44	134	50	72	38	59	33	11	9	10	7
Avg	6	156	90	194	75	252	193	263	136	18	10	33	18

Students in the local low count struggle (Table 4.5) group increase their source line insertions and deletions during struggle, while decreasing their test line insertions and increasing their test line deletions. They also increase file changes for both source and test files. They might be attempting to change their tests to reproduce TS feedback errors, as the increase in source line and file changes indicates they might not know where the errors are being generated.

Table 4.5: Line and files changes before and during struggle for repositories in the local low count struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	11	220	99	371	82	441	222	465	124	15	9	37	19
S20	18	304	175	339	107	612	375	572	200	30	15	70	31
F20	19	372	165	562	135	484	276	538	170	29	19	55	30
S21	13	316	127	530	109	295	156	377	107	23	15	36	20
F21	11	242	102	353	99	304	172	354	112	16	11	34	19
S22	17	285	134	358	105	271	184	284	119	22	14	34	17
F22	13	379	150	534	153	354	172	458	145	23	17	35	22
S23	6	157	69	199	53	80	42	83	27	10	7	11	6
F23	11	229	106	333	83	260	156	308	118	16	11	26	15
S24	8	82	75	128	83	100	60	88	59	11	9	14	8
Avg	13	259	120	371	101	320	182	353	118	19	13	35	19

During struggle students in the local high count struggle group (Table 4.6) increase their source and test line insertions, decrease their source line deletions, and increase their test line deletions. However, test insertions increase at a higher rate. They also increase both source and file changes. Similar to the students in the local low count struggle group, they might be attempting to reproduce TS tests in order to find the errors. Particularly in this group, students in Spring of 2020 show higher rates of source line insertions, and Spring and Fall of 2020 students show higher struggle counts.

Table 4.6: Line and files changes before and during struggle for repositories in the local high count struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	70	595	542	312	68	1149	252	673	233	63	19	150	52
S20	93	1141	950	804	436	1597	365	1308	733	86	48	171	99
F20	94	750	545	786	439	850	534	1021	583	78	38	163	67
S21	62	630	374	702	294	638	81	580	403	72	39	137	62
F21	48	640	428	434	182	845	242	749	402	40	21	101	45
S22	82	868	575	460	192	1128	105	700	346	99	27	218	64
F22	50	560	294	499	167	500	204	617	297	52	29	95	47
S23	38	453	269	413	164	382	295	505	209	41	24	73	38
F23	81	345	178	671	200	469	177	1280	500	53	29	139	84
S24	25	270	171	193	100	249	556	245	144	31	19	55	30
Avg	64	625	433	527	224	781	281	768	385	62	29	130	59

Students in the local low length struggle group (Table 4.7) increase their source line insertions and deletions during struggle, but decrease test line insertions and deletions, while

increasing file changes for both source and test files. However, their test line changes are higher than their source line changes before and during struggle.

Table 4.7: Line and files changes before and during struggle for repositories in the local low length struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	12	226	107	329	67	367	188	380	88	15	9	30	15
S20	14	237	127	292	86	377	194	340	108	23	12	39	18
F20	24	387	182	625	164	471	264	570	194	32	21	52	29
S21	16	307	130	543	108	279	145	340	94	22	15	32	16
F21	12	258	111	376	110	288	166	342	112	17	12	29	16
S22	18	239	106	365	91	184	109	220	77	21	13	28	14
F22	15	399	162	542	160	345	168	427	139	23	18	32	21
S23	10	208	93	257	76	120	69	121	38	13	9	14	7
F23	10	187	100	295	74	187	113	209	70	14	10	19	11
S24	7	65	51	68	55	34	17	32	28	8	6	8	4
Avg	14	251	117	369	99	265	143	298	95	19	12	28	15

In the local high length struggle group (Table 4.8), students increase both source line insertions and deletions, and test line insertions and deletions during struggle, as well as increasing the number of source and test file changes. Their long struggle sequences might be cause by unproductive code churning cause by not knowing where the error is being generated.

Table 4.8: Line and files changes before and during struggle for repositories in the local high length struggle group.

		Line Changes								File Changes			
		Before Struggle				During Struggle				Before Struggle		During Struggle	
Semester	**Struggle Count**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source Insertions**	**Source Deletions**	**Test Insertions**	**Test Deletions**	**Source**	**Test**	**Source**	**Test**
F19	11	207	59	438	53	690	292	838	193	13	9	60	35
S20	31	423	275	410	142	1214	844	1041	378	50	22	148	66
F20	32	441	266	466	176	929	585	791	353	49	27	136	64
S21	24	451	279	540	213	719	546	784	419	45	28	108	58
F21	22	263	119	262	142	646	400	691	338	28	14	95	53
S22	49	516	310	464	203	915	648	443	204	62	22	145	38
F22	22	312	204	446	218	580	371	846	480	38	24	92	55
S23	27	504	223	641	203	652	376	671	269	34	22	76	41
F23	30	384	200	355	95	715	488	897	445	30	17	80	48
S24	21	294	185	329	175	391	284	443	234	33	21	60	31
Avg	27	379	212	435	162	745	483	744	331	38	21	100	49

Help-seeking behavior

Students help-seeking rates are generally low overall. We do see long lines during officer hours, but that happens close to deadlines and generally with the same group of students. However, we can see, as shown previously, that during Spring and Fall of 2020 help-seeking rates were higher. Particularly for local high count groups, MDH rates were higher for Fall of 2020 were higher than any other semesters. Piazza and MDH rates tend to be higher for the local high count and length groups during struggle, indicating students tend to look for help after dealing with the errors in too many different classes or errors in the same class for a long number of commits. MDH and Piazza frequencies before and during struggle can be seenin A on Tables A.1 through A.8.

4.1.2 Discussion

For every struggle group, ratios between test and source code development diminishes during struggle, meaning they do tend to increase their focus on test development once they start to see persistent errors. However, each groups behaves in a different way when responding to struggle.

Students in the global low count struggle group present less modifications in a wider range of files, which can indicate they understand that errors do not necessarily come from the same file the error is being triggered, but do not necessarily understand which files depend on each other. Students in the global low length struggle group present less line and file changes, showing they might have a better understanding of which files to modify than the global low count struggle group. Students in the local low count and length groups present more source insertions and deletions, while also working on more files, which indicates those groups might struggle with unproductive code churning. However, students in the low count group present less test line insertions and more test line deletions, which shows they might be trying to reproduce TS tests, while students in the low length group present less test line insertions and deletions, which might show they are relying on TS tests to debug their code and not on their own tests.

Students in the global high count struggle group present more source and test insertions and deletions during struggle, but work on less files. That might show they are trying to reproduce TS errors and changing their source files accordingly. Students in the global high length struggle group also increase their source and test line insertions and deletions, but also work on more files during struggle; although they might also be trying to reproduce TS tests, it seems they have more difficulty in understanding which files to work on. Students in the local high count struggle group show more source line and test insertions, but less source line deletions and

more test line deletions and more file changes; that might indicate they are struggling with code churning when trying to fix persistent errors. Students in the local high length struggle group present more source and test insertions, deletions, and files, which might show they are struggling with understanding which classes to fix.

In conclusion, coding and help-seeking behavior was not well-divided between struggle groups, with low struggle groups also presenting indications of code churning and a difficulty to know which classes were triggering the errors they were seeing, and all students show the tendency to only seek help once they have been struggling with persistent error.

4.2 Students use of their own test cases (RQ2.2)

To analyze the students' use of their own test cases, we collect the following metrics for each repository: number of test cases that never pass or never fail, number of student test cases that change status from either failing to passing or passing to failing, number of failing student tests cases that were followed by a commit with related source code changes — meaning the changes were done on the relevant source code class, which would indicate the student was using their tests to fix their implementation errors, a desirable behavior —, number of failing test cases that were never followed by relevant source code changes — meaning the student waiting until having access to TS test cases to fix their implementation —, and total number of non-implemented test methods on the last commit before reaching milestone M2. These metrics were collected from the period between milestones M1 and M2, which means students were working on their code without having access to TS test cases and relied solely on their own tests for their implementation.

4.2.1 Test-Driven Development

Since we are using TS tests results we also explore the students' use of Test-Driven Development (TDD) strategies, which is encouraged in the course, and we analyze whether students behavior regarding help-seeking changes when facing struggle moments. We show here adherence to TDD strategies for low and high struggle groups.

TDD is a programming strategy in which tests are developed prior to source code, and it was first introduced by Kent's "eXtreme Programming" (XP) work Beck (1999). There has been studies exploring the impact of TDD on novice programmers Buffardi and Edwards (2012); Desai et al. (2008). In our course, we use a slightly different definition of TDD, where students can also write test code simultaneously with source code. As explained in Section 2.1, students develop tests for each one of the source code methods. To explore TDD, we collect the

number of: commits with more source code than test code, commits with no test development, commits with non-implemented tests, tests that were not passing once students reached M2, and commits done before any test implementation. An average summary of those metrics per project can be seen in Tables 4.9 through **??**. Although these metrics are similar to the ones on Section 4.1.1, here we explore student code before they had TS tests feedback, which means they had not reached a point of struggle yet.

More meaningful differences are shown when exploring metrics between projects, which is why they are presented for each project. When exploring data from before milestone M2 between struggle groups, their behavior was similar. Therefore struggle groups might not be an effective way to separate students to analyze data before they reach struggle. However, there were two meaningful differences: global and local low length groups presented less total commits, while global high count and length, and local high count length groups presented a higher number of commits before test implementation. This might give an insight at how they would present higher rates of struggle later on, by not starting test development early.

Table 4.9: Average per commit TDD metrics for each project.

			Average Per commit				
Semester	**Project**	**#Commits**	**Source Files**	**Test Files**	**Lines Changed**	**Tests not Passing**	**Coverage**
F19	**P1**	37	7%	5%	113.2	16.9	42%
	P2	38	7%	5%	106.1	32.1	45%
S20	**P1**	37	6%	4%	67.3	13.8	50%
	P2	42	6%	4%	67.7	26.7	51%
F20	**P1**	8	46%	42%	233.1	21.0	30%
	P2	6	77%	66%	238.6	25.6	35%
S21	**P1**	9	45%	32%	223.6	14.5	32%
	P2	5	92%	67%	226.8	20.8	33%
F21	**P1**	10	37%	29%	205.5	21.5	30%
	P2	9	46%	33%	176.9	28.8	33%
S22	**P1**	8	54%	40%	214.4	13.9	32%
	P2	8	57%	41%	212.7	19.6	34%
F22	**P1**	9	45%	33%	214.7	17.6	31%
	P2	7	64%	48%	216.8	25.2	34%
S23	**P1**	9	39%	28%	170.5	16.3	30%
	P2	9	44%	32%	179.4	20.5	34%
F23	**P1**	12	35%	28%	246.2	16.9	29%
	P2	8	62%	48%	239.8	24.4	36%
S24	**P1**	9	34%	32%	168.7	17.9	30%
	P2	8	42%	37%	170.4	24.2	32%

Table 4.10: TDD source and test changes metrics for each project.

			Commits with			
Semester	**Project**	**#Commits**	**Non-implemented tests**	**More Source than Test changes**	**Source changes and no Test changes**	**Source Changes and Non-implemented Tests**
F19	**P1**	37	2%	92%	86%	23%
	P2	38	2%	89%	82%	23%
S20	**P1**	37	3%	75%	71%	29%
	P2	42	3%	95%	84%	38%
F20	**P1**	8	7%	63%	48%	10%
	P2	6	15%	100%	79%	19%
S21	**P1**	9	7%	69%	52%	14%
	P2	5	18%	100%	94%	30%
F21	**P1**	10	6%	73%	57%	14%
	P2	9	16%	80%	64%	23%
S22	**P1**	8	11%	64%	47%	19%
	P2	8	14%	61%	47%	20%
F22	**P1**	9	12%	55%	42%	18%
	P2	7	18%	69%	56%	23%
S23	**P1**	9	10%	61%	49%	14%
	P2	9	13%	60%	50%	16%
F23	**P1**	12	8%	59%	48%	19%
	P2	8	18%	77%	62%	25%
S24	**P1**	9	4%	49%	40%	9%
	P2	8	11%	58%	49%	14%

There was a drop on the number of commits after Fall of 2020. There were no structural changes in the course or in the project other than students working from home during the COVID-19 pandemic. Working from home in a more relaxed environment may have had a long-lasting influence on how students work through programming courses.

Additionally, we can see that for every project, commits focus on source code development rather than on test development, including high rates of commits in Spring of 2020 that had at least one test not implemented. There is also a high number of commits with student tests failing when students reach M2, even though most students do start the implementation of tests early.

Table 4.11: Test implementation TDD metrics for each project.

Semester	Project	#Commits	Non-implemented tests at M2	Tests not Passing at M2	Commits Before Test Implementation
F19	**P1**	37	1	11	17%
	P2	38	1	15	13%
S20	**P1**	37	1	9	18%
	P2	42	1	16	19%
F20	**P1**	8	0	16	47%
	P2	6	1	19	64%
S21	**P1**	9	0	11	48%
	P2	5	1	16	72%
F21	**P1**	10	0	15	38%
	P2	9	1	23	44%
S22	**P1**	8	1	12	51%
	P2	8	1	15	42%
F22	**P1**	9	1	13	37%
	P2	7	1	20	34%
S23	**P1**	9	0	12	42%
	P2	9	1	14	35%
F23	**P1**	12	0	10	42%
	P2	8	1	18	50%
S24	**P1**	9	0	13	44%
	P2	8	1	20	44%

4.2.2 Overall student test use

We collect metrics on student tests that show how students use their own tests on average before milestone M2. Exploring these metrics per student group showed no meaningful differences. Metrics per project can be seen on Table 4.12. We collect total number of student tests, tests that never fail or never pass, tests that change status once or more than one time, and the percentage of tests not implemented and no passing at milestone M2.

One interesting finding was that students write tests that become obsolete with the evolution of the project, and later delete them, which is why the percentage of tests that never pass is higher than the percentage of tests not passing at M2. The percentage of tests not passing at M2 is generally higher than 25%. There is also a high percentage of tests that never fail, while a lower percentage of tests that change status more than once, which indicates students might be writing low quality tests, agreeing with our hypothesis that they write tests with the goal of

achieving coverage.

Table 4.12: Overall student test use per project.

Semester	Project	# Tests	# Tests never fail	# Tests never pass	# Pass to fail	# Fail to pass	# Change more than once	Not Implemented at M2	Not passing at M2
F19	**P1**	47	25%	39%	3%	2%	9%	1%	23%
	P2	99	15%	37%	1%	0%	14%	1%	15%
S20	**P1**	39	22%	40%	5%	3%	14%	2%	24%
	P2	90	11%	44%	2%	1%	12%	2%	17%
F20	**P1**	42	20%	54%	2%	0%	2%	1%	37%
	P2	58	17%	49%	2%	0%	2%	1%	32%
S21	**P1**	37	28%	43%	3%	0%	1%	1%	30%
	P2	48	22%	54%	2%	1%	2%	2%	34%
F21	**P1**	45	23%	48%	2%	0%	2%	1%	35%
	P2	53	13%	65%	2%	0%	1%	3%	44%
S22	**P1**	35	30%	42%	3%	0%	2%	1%	33%
	P2	44	20%	55%	2%	0%	0%	3%	35%
F22	**P1**	40	30%	43%	2%	0%	1%	2%	34%
	P2	48	17%	66%	2%	0%	0%	3%	42%
S23	**P1**	35	34%	47%	2%	0%	0%	1%	34%
	P2	41	22%	51%	2%	0%	1%	2%	35%
F23	**P1**	47	37%	35%	2%	0%	1%	1%	22%
	P2	52	21%	58%	2%	0%	0%	3%	35%
S24	**P1**	40	28%	50%	2%	1%	2%	0%	33%
	P2	51	16%	64%	2%	0%	0%	3%	38%

4.2.3 Test use before and during struggle

Lastly, we analyze how students use their own tests after milestone M2. We collect coverage and student test failures before and during struggle. We also introduce the *debugging commit*. A *source debugging commit* is a commit during a struggle sequence that had source code changes related to the TS test fixed at the end of the struggle sequence, if the struggle sequence did not end with a non-compiling commit. A *test debugging commit* follows the same logic, but with a modification on the students' test class related to the fixed TS test. Debugging commits only happen during struggle.

Metrics can be seen in Tables 4.13 through 4.20. Students test use in general does not change between before and during struggle moments. Coverage levels are similar between groups and semesters. Student test failures tend to go down during struggle for all groups, but

not significantly. High struggle groups present higher numbers of debugging source and test commits, which agrees with the previous findings of source and test changes metrics during struggle, and shows students in high struggle groups try to alter the class showing the error in the TS feedback which might not be the correct class to modify.

Table 4.13: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the global low count struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	3	86%	9	86%	8	6	2
S20	6	85%	9	85%	8	10	4
F20	5	85%	5	86%	3	7	3
S21	4	86%	5	87%	4	6	2
F21	3	82%	11	82%	10	2	2
S22	4	86%	5	87%	5	6	2
F22	3	89%	5	90%	4	2	1
S23	3	85%	4	86%	3	3	1
F23	3	88%	4	89%	4	2	2
S24	2	87%	2	87%	2	2	1
Avg	4	86%	6	87%	5	5	2

Table 4.14: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the global high count struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	21	83%	19	83%	12	41	14
S20	32	83%	28	83%	11	37	13
F20	41	84%	28	83%	2	98	20
S21	20	84%	24	83%	5	29	19
F21	13	73%	25	73%	12	17	8
S22	36	86%	31	87%	1	61	19
F22	13	87%	26	88%	4	11	9
S23	12	89%	16	89%	2	13	6
F23	13	87%	17	88%	3	11	14
S24	7	84%	56	86%	1	7	5
Avg	21	84%	27	84%	5	33	13

Table 4.15: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the global low length struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	4	85%	9	86%	9	5	2
S20	6	84%	7	85%	7	7	4
F20	6	85%	4	86%	3	6	3
S21	4	87%	5	87%	4	4	2
F21	4	81%	10	81%	10	3	2
S22	6	86%	5	87%	5	8	3
F22	4	88%	5	89%	4	4	2
S23	4	86%	4	86%	3	3	2
F23	4	87%	3	88%	5	3	2
S24	3	83%	1	84%	4	1	1
Avg	4	85%	5	86%	5	4	2

Table 4.16: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the global high length struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	5	88%	7	88%	5	18	8
S20	12	84%	14	83%	15	42	10
F20	6	83%	5	85%	4	20	5
S21	7	85%	5	85%	5	16	5
F21	4	83%	15	84%	14	6	5
S22	6	82%	4	84%	4	17	7
F22	4	91%	3	91%	2	2	4
S23	4	86%	7	89%	3	5	3
F23	6	87%	1	87%	1	9	8
S24	4	89%	0	89%	1	3	3
Avg	6	86%	6	86%	5	14	6

Table 4.17: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the local low count struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	11	84%	11	86%	9	9	4
S20	18	84%	10	85%	9	16	7
F20	19	84%	5	87%	3	10	4
S21	13	85%	6	88%	4	7	3
F21	11	81%	14	83%	10	3	3
S22	17	85%	7	88%	5	6	2
F22	13	88%	7	90%	4	3	2
S23	6	85%	6	88%	2	1	1
F23	11	88%	6	90%	4	3	2
S24	8	84%	3	88%	2	1	1
Avg	13	85%	7	87%	5	6	3

Table 4.18: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the local high count struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	70	80%	15	81%	14	47	20
S20	93	81%	7	81%	6	41	16
F20	94	84%	4	85%	3	70	15
S21	62	82%	8	82%	7	42	18
F21	48	77%	14	78%	12	22	11
S22	82	85%	2	85%	2	62	18
F22	50	86%	6	88%	5	15	8
S23	38	85%	4	86%	3	11	5
F23	81	88%	1	88%	1	15	31
S24	25	83%	1	85%	1	7	4
Avg	64	83%	6	84%	5	33	15

Table 4.19: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the local low length struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	12	84%	11	86%	9	6	3
S20	14	84%	10	84%	8	7	3
F20	24	85%	6	87%	3	9	3
S21	16	86%	6	88%	4	6	3
F21	12	79%	13	81%	10	3	2
S22	18	85%	7	88%	4	4	1
F22	15	88%	7	90%	4	3	2
S23	10	85%	5	88%	2	1	1
F23	10	88%	6	89%	4	2	2
S24	7	83%	3	87%	1	0	0
Avg	14	85%	7	87%	5	4	2

Table 4.20: Coverage and student test failures, and number of debugging commits, before and during struggle for repositories in the local high length struggle group.

		Before Struggle		During Struggle		Debugging commits	
Semester	**Struggle Count**	**Coverage**	**Student test failures**	**Coverage**	**Student test failures**	**Source**	**Test**
F19	11	86%	8	86%	9	23	8
S20	31	83%	11	84%	10	41	15
F20	32	85%	2	85%	2	49	14
S21	24	84%	8	85%	6	25	12
F21	22	82%	16	83%	11	11	9
S22	49	85%	5	85%	5	53	14
F22	22	84%	5	85%	4	13	7
S23	27	85%	4	87%	2	12	6
F23	30	87%	3	89%	2	11	12
S24	21	83%	2	85%	1	8	4
Avg	27	84%	6	85%	5	25	10

4.2.4 Discussion

Although changes in source and test changes during struggle varies among struggle groups, the ratio between test code changes and source code changes is lower during moments of struggle. That indicates that during struggle students do focus more on test development.

TDD metrics indicate that students do not follow the principles of TDD. We hypothesize that students develop tests just enough to achieve project marks and milestones, so they can then test their implementations through the TS tests. That aligns with the need a student have to complete an assignment for the purpose of achieving a grade, and it demonstrates students might not understand how testing can be a helpful tool to guide development. In the following section, we present TDD metrics for each struggle group.

In general, at least a third of the tests developed by the students seem to never change status between passing and failing within milestones, however that is impacted by students deleting tests before they pass or fail. That also indicates that students are writing tests with the goal of achieving milestones, and not to guide their development. However, students tend to write more test cases for the second project of the semester, which would indicate that there is an improvement in testing behavior through the semester, as the projects differ in topics but not necessarily in complexity — although complexity is relative, because some concepts are "hard" for some students, while a different set of concepts might be considered "harder" to a

different set of students.

We can also see that students do increase their development in moments of struggle, which indicates they do attempt to use their tests to fix errors in their development or reproduce TS tests, which are both desired behaviors. However, their source and test code changes during struggle indicates they might be face code churning when not knowing which classes to work on to fix persistent errors. In addition, the amount of non-implemented tests before students reached M2 seems to be negligible when compared to the total of student tests, but around 35% of those tests are not passing at M2, which aligns with the hypothesis that students write tests to achieve coverage and not necessarily to guide their development. Furthermore, students do not seem to change the use of their own tests during moments of struggle, but the debugging commits rate shows they do try to modify the source and test classes related to the persistent error they are trying to fix.

CHAPTER

5

RELATIONSHIPS BETWEEN STRUGGLE AND FACTORS EXTERNAL TO THE COURSE (RQ3)

Our third research question, *"What are the relationships between struggle and factors external to the course?" (RQ3)*, aims to study the relationship between the occurrence of struggling moments and factors external to the course, particularly demographic and cognitive factors. The idea for this exploration comes from anecdotal observations of previous computing experience affecting the success of students in introductory programming courses combined with the influence demographic factors have on opportunities to achieve computing experience. This research question is divided into three sub-questions:

RQ3.1 "Is there a relationship between demographics and struggle occurrence?" (RQ3.1)

RQ3.2 "Is there a relationship between struggle and CS general misconceptions, self-efficacy, and interest?" (RQ3.2)

RQ3.3 "Is there a relationship between struggle and economic well-being?"(RQ3.3)

5.1 Demographics and previous experience (RQ3.1)

Through all semester, as a normal part of the course, we have collected students' gender and race. We exclude students that did not give consent to participate in the study. Along with demographics, we also collect information on students' previous computer science experience before enrolling in the course.

5.1.1 Race analysis

When answering which race(s) students belong to, they had the options of choosing between Asian or Asian-American, Non-Hispanic White, American Indian or Alaska Native, Black or African-American, Hawaiian or Other Pacific Islander, and Hispanic or Latino. They could select all races they believed to belong too, or deny to respond. To protect students' privacy, we group the race options with fewer students as Other/Multiple, so students cannot be identified through this work.

The majority of our students either selected Non-Hispanic White, or Asian/Asian-American, so the distributions were skewed towards those races, and similar among semesters with over 70% of the students selecting one of those two races. Race distribution can be seen on Figure 5.1.

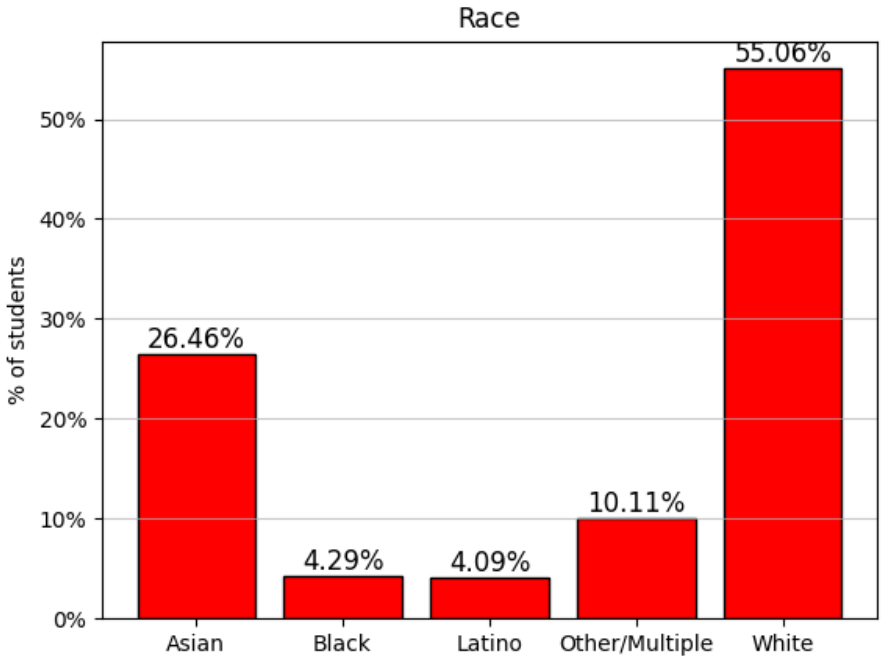


Figure 5.1: Race distribution across all semesters.

To analyze whether struggle and race had any relationships, we calculate the correlations between students race selection and their struggle group, struggle count, and struggle length. Due to the skewed nature of our students' race distribution, there was no significant relationship shown in the correlations, as the correlation value was 0.0079 when calculating the correlation across semesters, and correlations for each semester presented similar results, as race distribution had approximately the same shape in every semester. Distributions were also similar when analyzing the race distribution of each struggle group. An example can be seen in Figure 5.2, which shows the race distribution for the local struggle low count group in Fall of 2019.

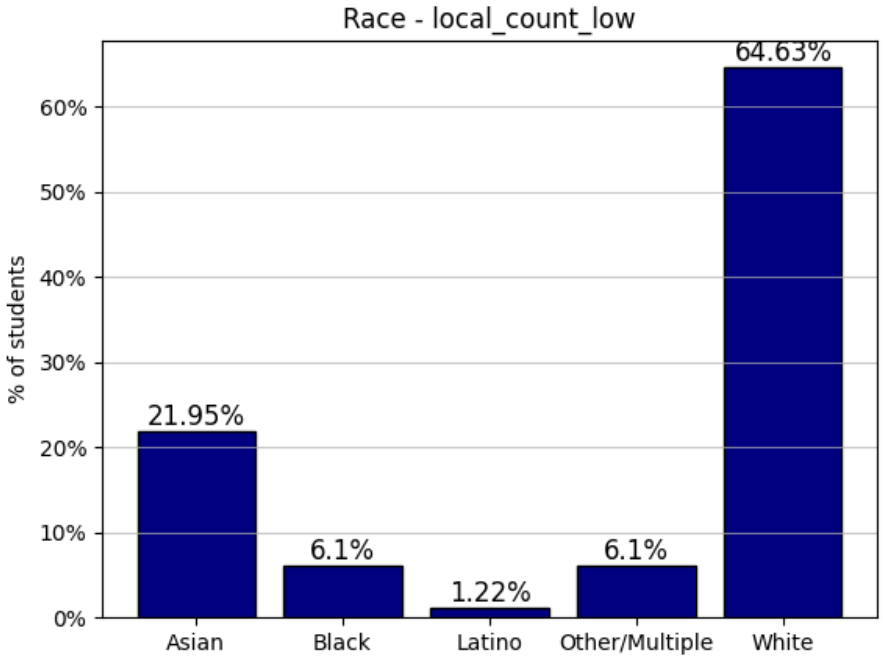


Figure 5.2: Race distribution for the local struggle low count group in Fall of 2019

When calculating correlations between race and struggle count and length, the correlation values were not significant, both being lower than 0.02 for global struggle, and lower than 0.04 for local struggle.

5.1.2 Gender Analysis

When analyzing gender demographics, most of our students selected "male" as their answer. They could choose "male", "female", "other", or they could input their own gender or decline to answer. Distributions of gender between semesters were similar, and the gender distribution across semesters can be seen in Figure 5.3.

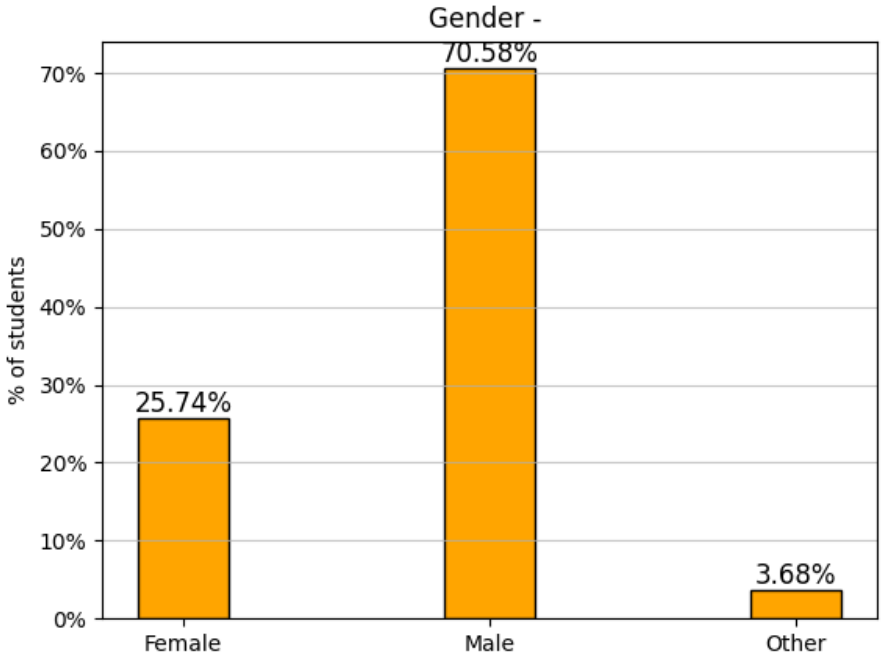


Figure 5.3: Gender distribution across semesters.

Due to the skewed distribution, gender and struggle groups did not present a significant correlation, with a value of approximately 0.005. When calculating correlations between gender and struggle counts and length, struggle count and gender presented a correlation of 0.11 with global struggle, which although is higher than the other correlations is still not significant. When considering local struggle, values were lower than 0.09.

5.1.3 Prior Experience Analysis

We explore whether students' prior Computer Science experiences have a relation to struggle. Prior experience was collected along with demographics, and students could choose from

the following options: CSC116 (the CS1 course that is generally taken before CSC116 at our institution), AP and high-school (HS) Computer Science classes, intro Computer Science course from Community College (CC) or a different university, CSC116 transfer credits, online courses (MOOCs), professional experience, and self-learning. They could chose one or more options when responding.

Most of our students have taken CSC116 before. For example, Figure 5.4 shows prior experience distribution, and over 60% of the students have taken CSC116. Distributions for other semesters are similar. Figure 5.5 shows the distribution of previous experiences disregarding CSC116 across semesters.

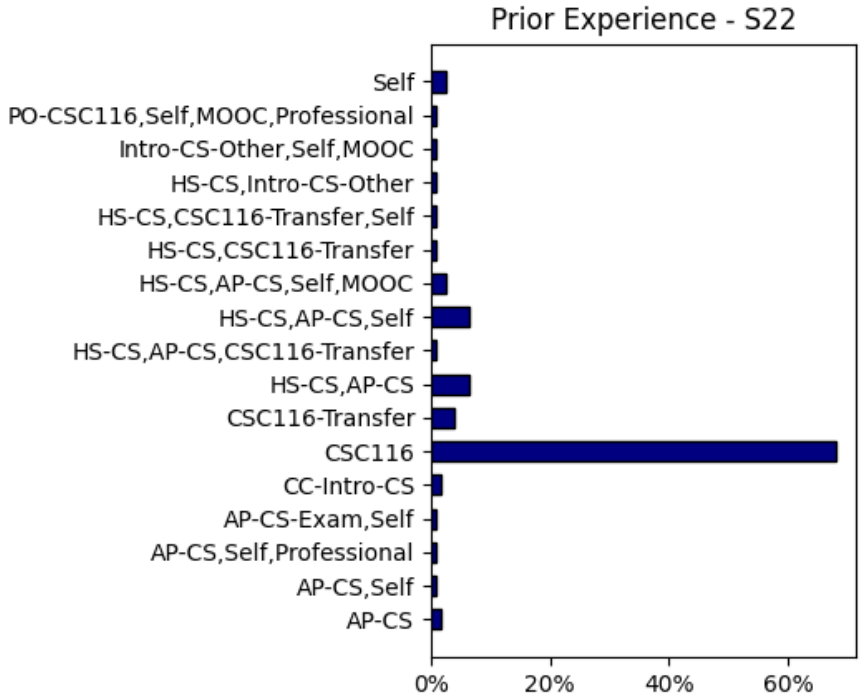


Figure 5.4: Prior experience distribution across semesters.

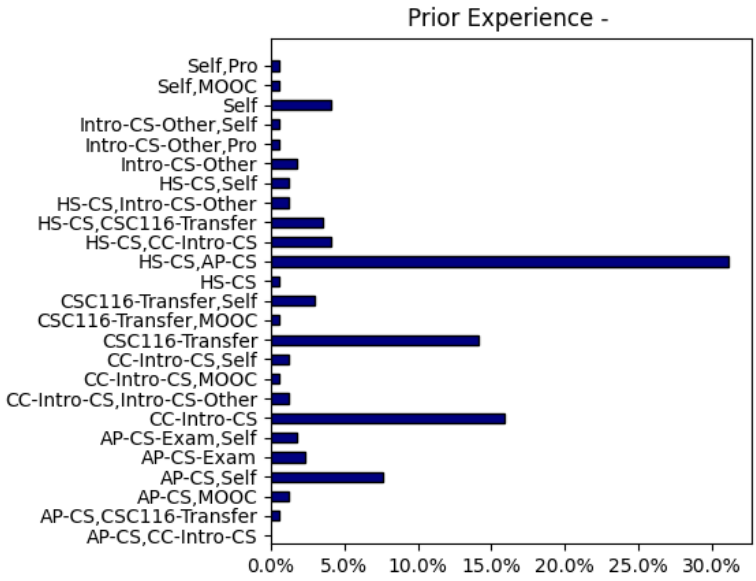


Figure 5.5: Prior experience distribution across semesters.

5.2 Qualitative Surveys (RQ3.2)

Students answer the Computing Attitude Survey (CAS) every semester, at the beginning of the semester. CAS contains questions referring to the cognitive factors we are interested in exploring, and have been thoroughly validated in educational settings Elliott Tew et al. (2012); Bockmon et al. (2019); Lee (2019); Leifheit et al. (2019); Zavaleta Bernuy and Harrington (2020); Jain et al. (2023).

Along with CAS, the students respond to 26 additional questions at the beginning of the semester that cover their interest in Computer Science, their confidence in the possibilities of achieving high grades and passing the course, their reasons to take the course, and other self-belief questions. CAS together with the 26 additional questions form Survey A, which can be seen on Appendix B. In Spring of 2023, we started a new survey – Survey B – that covers similar topics, but that was administered twice every semester, after each project, to analyze whether their answers change through the course. Questions from Survey B can be seen in C. In total, students responded to 52 questions over each semester from Spring of 2019 to Fall of 2022, and 71 survey questions over each semester from Spring of 2023 to Spring of 2024. All questions are answered through a Likert scale from 1 to 5.

Although the CAS survey has pre-defined question groups, the groups did not work with our students. We calculate correlations between questions for each semester, and there was no

significant correlation between questions in pre-defined groups. All correlation values were less than 0.5. We also did not have pre-defined groups for the questions not from CAS. Therefore, instead of using groups, we calculate the correlations between struggle and individual questions. Survey response distributions for both Survey A and Survey B were similar between semesters, with each semester having the same average and median for each question, and therefore we calculate correlations across semesters.

We first calculate correlations between questions and struggle groups, meaning the correlation between the responses and the chance of a student being in a given struggle group. There were no significant correlations, as values for all questions were less than 0.05.

We then calculate the correlations between questions within each struggle group, meaning calculating the correlation between questions using only responses from students from a given struggle group. The idea is that students in the same group might not have similar responses for a specific question, but they might have similar responses for a group of questions.

For Survey A, that exploration yielded a few questions with higher correlations when considering each struggle group. In fact, the same questions correlated in every struggle group, with 90% of the students in the group choosing 4 or 5 as a response. Correlation values were between 0.6 and 0.7, with the exception of questions 33 and 34 which had a higher correlation of above 0.8. The questions that formed groups when calculating correlations within struggle groups are:

- Questions 27, 28, 30, 31, and 32:
  - Generally I have felt secure about attempting computer programming problems.
  - I am sure I could do advanced work in computer science.
  - I think I could handle more difficult programming problems.
  - I can get good grades in computer science.
  - I have a lot of self-confidence when it comes to programming.
- Questions 33 and 34:
  - I like writing computer programs.
  - Programming is enjoyable and stimulating to me.
- Questions 11 and 13:
  - I find the challenge of solving computer science problems motivating.
  - I enjoy solving computer science problems.

For Survey B, since that survey was given once after every project, we first check whether the answers changed over time. The questions did not significantly change over time. Since average and medians for answers were the same across semesters for each question, we also analyze this survey across semesters. When calculating correlations between the questions ans struggle groups, correlations were not significant, following the same patter as Survey A. We then follow the same process of calculating correlation withing each struggle group. Questions 1 and 2 presented correlation above 0.65 for every struggle group with students answer those questions with mostly 4 and 5. Interestingly, questions 1, 2, 9, 11, and 12 correlated for every *high struggle* group, but with slightly lower answers, as the average answer to question 10 was closer to 3, and averages for other questions were closer to 4. Questions are presented below:

- Question 1: I am confident I can learn the concepts taught in this course.
- Question 2: I am confident I can apply the skills being taught in this class.
- Question 9: I find the challenge of solving CS problems motivating.
- Question 11: Before I work on assignments, I divide my tasks into smaller sub tasks.
- Question 12: I set a specific goal for each study/work section.

5.3 Struggle and Economic well-being (RQ3.3)

We also investigate the relationship between struggle and economic-well being by calculating correlations between struggle and median-income per zip code. Students' home area zip code was collected with IRB authorization, and only zip codes from consenting students were used. Median-income per zip code data was collected from the American Community Survey 2005-2022 Bureau (2024), created by the United States Census Bureau. Income distribution can be seen in Figure 5.6.

Although income approaches a more normal distribution than race and gender demographic data, there was no significant relationship between income and struggle groups, with a correlation of less than 0.001. When calculating the relationship between struggle counts and lengths and income across semesters, there also was not a significant relationship.

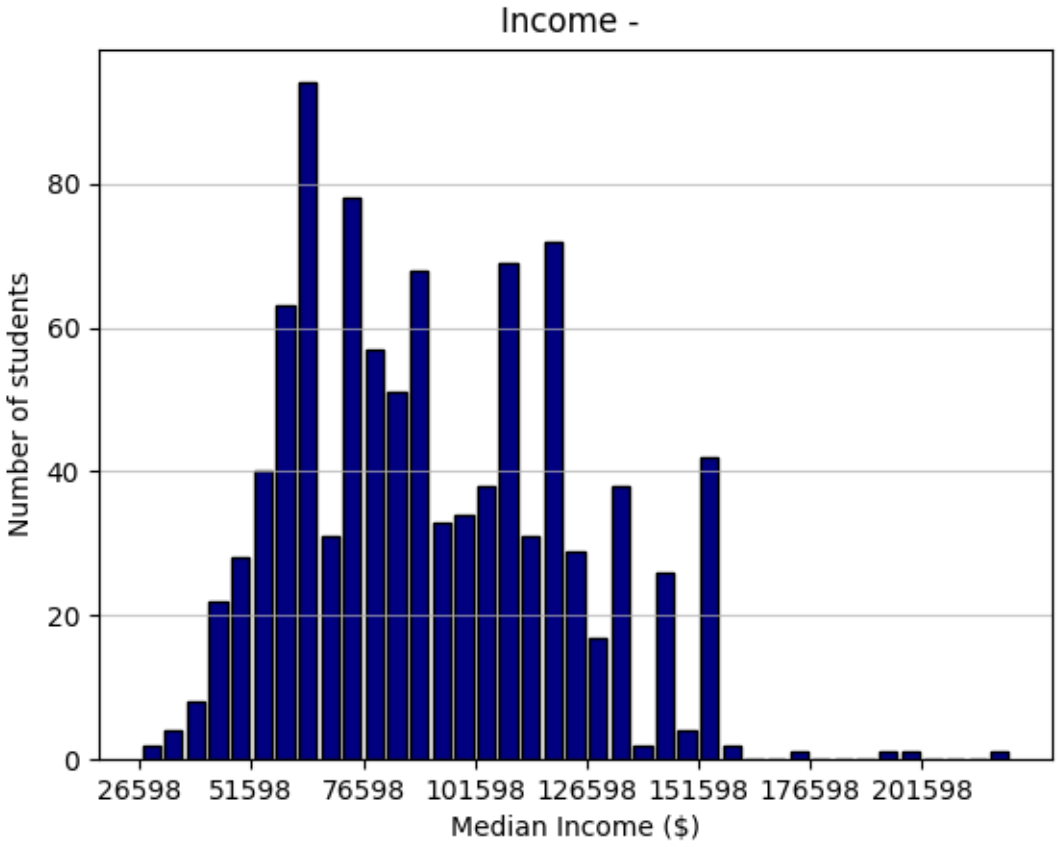


Figure 5.6: Income distribution across semesters.

5.4 Discussion

Our hypothesis for this research question was that higher rates of struggle would correlate with less advantaged students: students from community colleges, students with no prior experience, students from minorities, and/or students from less fortunate economic backgrounds.

We were not able to find any relationships between demographics and economic well-being, and struggle. With demographic data, that result was likely because our data distribution was skewed, since our students selected mostly white and male as their race and gender, so we did not have enough representation of all populations to arrive at a conclusion on whether there is a relationship between those variables and struggle. However, that does show that Computer Science student populations are still not diverse. Income distribution among our students approached a more normal distribution, but there was still not relationship between ecnomic well-bein and struggle.

When analyzing qualitative data from our surveys, there was no relationship between questions, and no single question presented a relationship with struggle, so we were not able to arrive at a conclusion on the relationship between self-efficacy, CS misconceptions, interest, and struggle. However, when grouping questions by struggle groups, were able to identify a relationship between questions that related to interest in Computer Science, although there was no significant difference in the responses between struggle groups.

CHAPTER

6

CONCLUSIONS

6.1 Conclusions

The goal of this research was to define struggle in programming courses in a way that could be used by instructors to help students, and to understand the coding behavior of students from a large, project-based CS2 course, during struggle. We chose to study a CS2 course because they are less studied in the current literature; however, as the CS2 course studied in this research has a CS1 course as a pre-requisite, students participating in this research already successfully completed one programming course and chose to continue in the CS path; students that never participated in a college-level programming course before would likely present different coding behavior during struggle. This research was motivated by our search for ways to improve student success. We start by exploring progress through progress indicators Edwards and Li (2016), the types of feedback offered in our course's assignments, and student effort. We see the the progress indicators available in the literature do not completely align with our course structure, and that non-coding and coding effort did not show a significant relationship, as help-seeking did not have correlations with grades or coding behavior. However, we did see that the order students implemented their code files changed between students with different grades.

We then pursue a more tangible way of helping students by exploring their behavior once

they start to struggle during programming assignments. We use data available in assignment submissions to define struggle (RQ1), which is defined as sets of persistent errors, that persist for 4 or more commits. We also arrive at two new definitions of struggle, *global struggle*, which refers to all test failures happening in a given commit, and *local struggle*, which refers to errors from specific classes. A struggle sequence is a sequence of commits presenting the same struggle set, and the struggle sequence ends when the student either fix one of the errors happening or commits non-compiling code. We find small correlations between struggle types and grades, despite the ceiling-effect in our grade data. To separate productive struggle from unproductive struggle, we use the productive indicators from the literature, selecting only the indicators that align with our course structure. We first compare student behaviors between programming projects, as there are two projects per semester, and between student group types, as students could choose to complete the second project individually or in pairs. Students in pairs tend to present higher levels of struggle; however, levels of struggle are generally higher on the second project.

To understand how students respond to struggle (RQ2), we use metrics generated by our previous explorations to analyze the students behavior before and during struggle. We see that most students present low levels of struggle, as they had a low number of struggle sequences and/or short struggle sequences. To locate meaningful differences in coding behavior, we divide students into struggle groups; global struggle groups – global high count, global low count, global high length, global low length –, and local struggle groups – local high count, local low count, local high length, local low length. We analyze implementation differences for each group during struggle. To analyze implementation behavior, we use coding metrics as source and test code line changes per struggle commit, file changes per struggle commit, and pairs of class tests that are implements, and pass or fail together. We also analyze the order in which students implement their project files. We see that students in low struggle groups show a higher rate of productive commits and productive struggle sequences, as well as a higher focus on test code development. We also see that some metrics, for example the order in which students implement their files, did not change between groups as much as it changed between projects. We then compare students' coding behavior between before and during struggle using similar metrics. We see that all students tend to increase their test code implementation when compared to their source code implementation. However, students from all groups show indications of undesired behaviors such as unproductive code churning. When we explore how students use their own tests, we conclude that they mainly use their tests as a way to achieve project marks, rather than to guide their development. However, we do see, through debugging commits, that students on high struggle groups tend to alter their code more frequently during struggle.

When analyzing whether external factors, such as demographics and economic well-being, have a relationship with struggle (RQ3), we were unable to get conclusive answers as distributions were skewed towards specific demographics. This does raise the point that underrepresented populations in Computer Science continue to be underrepresented even through new generations of students. Additionally, we do see an increase in struggle during the COVID-19 pandemic, and a decrease in metrics afterwards.

In conclusion, we were able to create a definition of struggle that can locate students that face persistent errors more often or for longer periods of time, and identify students that fit those categories. We increase generalizability by using programming class types instead of the specific classes used in our projects, as a way to give other instructors the possibility of using our definition of struggle with their own student populations. Furthermore, by using our definition of struggle, we were able to identify meaningful differences in student coding behavior that if addressed in time might increase retention and grades in CS2 courses.

6.2 Limitations

The assignments and student data used are all from the same course. As much as we tried to increase generalizability, it might be difficult to apply our methodology in a course with different structure, and results might be different when trying to apply this struggle definition into a new course with a different student population. Additionally, qualitative data came from course surveys, which may be biased by the students trying to give questions the "right answer".

Furthermore, this research was done on data from a CS2 course which has a CS1 course as a pre-requisite. Students that participated in this research were able to previously complete a CS1 course and chose to continue on the CS path by taking a CS2 course. That narrows the student population to students likely interested in CS, who already dealt with coding struggle in the past, which likely influenced their coding behavior. Analyzing submission data also means that we analyzed coding behavior from students that had submissions; we do not look into students that dropped out of the course. The bias introduced by this narrowing of the student population might be akin to "suvivorship bias", since students already "survived" one programming course in the past and the assignments of the current course. Future work is necessary to: 1) analyze coding struggle with a student population that had no previous college-level programming experience, and 2) look into whether the amount of coding struggle faced by students that dropped out of the course influenced their decision in dropping out, both for CS1 and CS2 courses.

Additionally, this research and its results are dependent on course structure. All data used

was dependent on how submissions were made, and each part of the research used a subset of data dependent on course milestones. The dependency on course structure reduces the reproducibility of the research studies covered in this dissertation. To be able to reproduce this research, the course structure should allow for the collection of the same metrics. It is not necessary for the course to cover the same content, or to follow test-driven development patterns; however, it is necessary to allow partial submissions, and to have testing code to assert the correctness of the submissions. Furthermore, although it is not necessary to match the exact course milestones present in this research, the milestones do influence the results. For instance, if students had access to teaching-staff test feedback throughout the whole project, it would be possible to collect struggle sequences throughout the whole project. Therefore, to replicate the definition of struggle in this research, it is also necessary to observe and separate collected data between periods when students have access to correct testing code.

Finally, our student population presented skewed distributions in all demographics. The distributions shown correlate to the distributions from the student population at the college level in our institution. As a result, we were unable to find statistically significant correlations between students' gender, race, previous experience, and economic well-being, and the amount of struggle they face when considering our definition of struggle. As noted, that does not rule out the possibility that a student's background and demographics does affect they way they struggle in programming and CS courses. However, the best way to create more balanced demographics is to increase the participation of underrepresented populations in CS, which could possibly be achieved through K-12 programs. Selecting groups of students to participate in research based on demographics would create a more balanced distribution; however, it would exclude a portion of students and introduce bias into the results, since having a balanced distribution in one variable does not guarantee a balanced distribution in another variable. For example, suppose we select students to create a group of students with balanced racial distribution; all selected students from one race group might have taken university CS classes, while all students from another race group might have been self-taught. In that situation, although race distribution would be balanced, the comparison between students levels of struggle would become unreliable since we excluded students from race groups that may have a different background.

6.3 Future Work

One piece of information lacking from this research is the work students do between commits, or partial submissions, since we do all the work with submission data. Submission data is readily available, but most of the information might be contained in what students do outside

of partial submissions. To address this issue, we are developing a plugin that captures students' coding actions while they are working on the project. We plan to apply our definition of struggle on the data generated by the plugin.

REFERENCES

Abdelhadi, A., Zainudin, S., and Sani, N. S. (2022). A regression model to predict key performance indicators in higher education enrollments. *International Journal of Advanced Computer Science and Applications*, 13(1).

Abdi, H. (2010). Holm's sequential bonferroni procedure. *Encyclopedia of research design*, 1(8):1–8.

Alzubaidi, L. (2017). Program outcomes assessment using key performance indicators.

Azcona, D., Hsiao, I.-H., and Smeaton, A. F. (2019). Detecting students-at-risk in computer programming classes with learning analytics from students' digital footprints. *User Modeling and User-Adapted Interaction*, 29(4):759–788.

Azcona, D. and Smeaton, A. F. (2017). Targeting at-risk students using engagement and effort predictors in an introductory computer programming course. In Lavoué, É., Drachsler, H., Verbert, K., Broisin, J., and Pérez-Sanagustín, M., editors, *Data Driven Approaches in Digital Education*, pages 361–366, Cham. Springer International Publishing.

Bandura, A. and Adams, N. E. (2005). Analysis of self-efficacy theory of behavioral change. *Cognitive Therapy and Research*, 1:287–310.

Barr, V. and Stephenson, C. (2011). Bringing computational thinking to k-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, 2(1):48–54.

Battestilli, L., Zahn, M., and Heckman, S. (2022). Academic help seeking patterns in introductory computer science courses. In *2022 ASEE Annual Conference & Exposition*, number 10.18260/1-2–41526, Minneapolis, MN. ASEE Conferences. https://peer.asee.org/41526.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77.

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., and Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, 26(2-3):148–175.

Bergin, S., Reilly, R., and Traynor, D. (2005). Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, page 81–86, New York, NY, USA. Association for Computing Machinery.

Bockmon, R., Cooper, S., Gratch, J., and Dorodchi, M. (2019). (re)validating cognitive introductory computing instruments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 552–557, New York, NY, USA. Association for Computing Machinery.

Bouffard, T., Bouchard, M., Goulet, G., Denoncourt, I., and Couture, N. (2005). Influence of achievement goals and self-efficacy on students' self-regulation and performance. *International Journal of Psychology*, 40(6):373–384.

Bucciarelli, L. (2018). Engineering enrollment data at mit. *MIT Faculty Newsletter*, XXX(5):14–16.

Buffardi, K. and Edwards, S. H. (2012). Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, page 105–110, New York, NY, USA. Association for Computing Machinery.

Bureau, U. C. (2024). American community survey 1-year data (2005-2022).

Cartwright, K. B., Lee, S. A., Taboada Barber, A., DeWyngaert, L. U., Lane, A. B., and Singleton, T. (2020). Contributions of executive function and cognitive intrinsic motivation to university students' reading comprehension. *Reading Research Quarterly*, 55(3):345–369.

Chambers, L. G. (1989). 73.52 spearman's rank correlation coefficient. *The Mathematical Gazette*, 73(466):331–332.

de Raadt, M. (2007). A review of australasian investigations into problem solving and the novice programmer. *Computer Science Education*, 17(3):201–213.

Desai, C., Janzen, D., and Savage, K. (2008). A survey of evidence for test-driven development in academia. *SIGCSE Bull.*, 40(2):97–101.

Doebling, A. and Kazerouni, A. M. (2021). Patterns of academic help-seeking in undergraduate computing students. In *21st Koli Calling International Conference on Computing Education Research*, Koli Calling '21, New York, NY, USA. Association for Computing Machinery.

Duran, R., Rybicki, J.-M., Sorva, J., and Hellas, A. (2019). Exploring the value of student self-evaluation in introductory programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 121–130, New York, NY, USA. Association for Computing Machinery.

Edwards, S. and Li, Z. (2016). Towards progress indicators for measuring student programming effort during solution development. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, page 31–40, New York, NY, USA. Association for Computing Machinery.

Elliott Tew, A., Dorn, B., and Schneider, O. (2012). Toward a validated computing attitudes survey. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, page 135–142, New York, NY, USA. Association for Computing Machinery.

Erickson, B., Heckman, S., and Lynch, C. F. (2022). Characterizing student development progress: Validating student adherence to project milestones. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, page 15–21, New York, NY, USA. Association for Computing Machinery.

Falkner, K., Vivian, R., and Falkner, N. J. (2014). Identifying computer science self-regulated learning strategies. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, page 291–296, New York, NY, USA. Association for Computing Machinery.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116.

Flavell, J. H. (1976). Metacognitive aspects of problem solving. In Resnick, L. B., editor, *The nature of Intelligence*, pages 231–236. Erlbaum, Hillsdale, NJ.

Flores, R. M. and Rodrigo, M. M. T. (2020). Wheel-spinning models in a novice programming context. *Journal of Educational Computing Research*, 58(6):1101–1120.

Gaber, I. and Kirsh, A. (2018). The effect of reporting known issues on students' work. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 74–79, New York, NY, USA. Association for Computing Machinery.

Gao, Z., Heckman, S., and Lynch, C. (2022a). Who uses office hours? a comparison of in-person and virtual office hours utilization. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 300–306, New York, NY, USA. Association for Computing Machinery.

Gao, Z., Heckman, S., and Lynch, C. (2022b). Who uses office hours? a comparison of in-person and virtual office hours utilization. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*, SIGCSE 2022, page 300–306, New York, NY, USA. Association for Computing Machinery.

Goldsby, H. J., Goings, S., Clune, J., and Ofria, C. (2009). Problem decomposition using indirect reciprocity in evolved populations. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 105–112, New York, NY, USA. Association for Computing Machinery.

Gusukuma, L., Bart, A. C., Kafura, D., and Ernst, J. (2018). Misconception-driven feedback: Results from an experimental study. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 160–168, New York, NY, USA. Association for Computing Machinery.

Heckman, S. and King, J. (2018). Developing software engineering skills using real tools for automated grading. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, page 794–799, New York, NY, USA. Association for Computing Machinery.

Hmelo-Silver, C. E. (2004). Problem-based learning: What and how do students learn? *Educational psychology review*, 16(3):235–266.

Hofmann, M. and Klinkenberg, R. (2016). *RapidMiner: Data mining use cases and business analytics applications*. CRC Press.

Izu, C., Pope, C., and Weerasinghe, A. (2019). Up or down? an insight into programmer's acquisition of iteration skills. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 941–947, New York, NY, USA. Association for Computing Machinery.

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1):25–40.

Jadud, M. C. (2006). An exploration of novice compilation behaviour in BlueJ.

Jain, A., Bockmon, R., Bourke, C., and Cooper, S. (2023). Validating a language-independent CS1 learning outcomes assessment. In *Proceedings of the ACM Conference on Global Computing Education Vol 1*, CompEd 2023, page 78–83, New York, NY, USA. Association for Computing Machinery.

Jegede, P. O., Olajubu, E. A., Ejidokun, A. O., and Elesemoyo, I. O. (2019). Concept–based analysis of java programming errors among low, average and high achieving novice programmers. *Journal of Information Technology Education: Innovations in Practice*, 18:049–059.

Jun, S., Kim, S., and Lee, W.-G. (2007). Online pair-programming for learning programming of novices.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. (2010). Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 107–111, New York, NY, USA. Association for Computing Machinery.

Kallia, M. and Sentance, S. (2019). Learning to use functions: The relationship between misconceptions and self-efficacy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 752–758, New York, NY, USA. Association for Computing Machinery.

Karvelas, I., Li, A., and Becker, B. A. (2020). The effects of compilation mechanisms and error message presentation on novice programmer behavior. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 759–765, New York, NY, USA. Association for Computing Machinery.

Kazerouni, A. M., Edwards, S. H., Hall, T. S., and Shaffer, C. A. (2017a). 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*, ITiCSE '17, page 104–109, New York, NY, USA. Association for Computing Machinery.

Kazerouni, A. M., Edwards, S. H., and Shaffer, C. A. (2017b). *Quantifying Incremental Development Practices and Their Relationship to Procrastination*, page 191–199. Association for Computing Machinery, New York, NY, USA.

Kinnunen, P. and Malmi, L. (2006). Why students drop out CS1 course? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, page 97–108, New York, NY, USA. Association for Computing Machinery.

Kinnunen, P. and Simon, B. (2012). My program is ok – am i? computing freshmen's experiences of doing programming assignments. *Computer Science Education*, 22(1):1–28.

Kolikant, Y. B.-D. and Mussai, M. (2008). "so my program doesn't run!" definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education*, 18(2):135–151.

Lawanto, O., Minichiello, A., Uziak, J., and Febrian, A. (2018). Students' task understanding during engineering problem solving in an introductory thermodynamics course. *International education studies*, 11(7):43.

LEE, D. M. (2011). Detecting confusion among novice programmers using BlueJ compile logs.

Lee, M. J. (2019). Exploring differences in minority students' attitudes towards computing after a one-day coding workshop. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 409–415, New York, NY, USA. Association for Computing Machinery.

Leifheit, L., Tsarava, K., Moeller, K., Ostermann, K., Golle, J., Trautwein, U., and Ninaus, M. (2019). Development of a questionnaire on self-concept, motivational beliefs, and attitude towards programming. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, WiPSCE '19, New York, NY, USA. Association for Computing Machinery.

Leinonen, J., Castro, F. E. V., and Hellas, A. (2021). Does the early bird catch the worm? earliness of students' work and its relationship with course outcomes. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, page 373–379, New York, NY, USA. Association for Computing Machinery.

Li, Z. and Edwards, S. (2018). Applying recent-performance factors analysis to explore student effort invested in programming assignments. Copyright - Copyright The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) 2018; Last updated - 2019-02-27.

Liu, Q. and Paquette, L. (2023). Using submission log data to investigate novice programmers' employment of debugging strategies. In *LAK23: 13th International Learning Analytics and Knowledge Conference*, LAK2023, page 637–643, New York, NY, USA. Association for Computing Machinery.

Loksa, D. (2017). Explicitly teaching metacognitive and self-regulation skills in computing. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 289–290, New York, NY, USA. Association for Computing Machinery.

Loksa, D. and Ko, A. J. (2016). The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, page 83–91, New York, NY, USA. Association for Computing Machinery.

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., and Burnett, M. M. (2016). *Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance*, page 1449–1461. Association for Computing Machinery, New York, NY, USA.

Loten, A. (2019). America's got talent, just not enough in it. *The Wall Street Journal*.

Lowe, T. (2019). Explaining novice programmer's struggles, in two parts: Revisiting the iticse 2004 working group's study using dual process theory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, page 30–36, New York, NY, USA. Association for Computing Machinery.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. (2010). The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4).

Martinez, M. E. (2006). What is metacognition? *Phi Delta Kappan*, 87(9):696–699.

Mayer, R. E. (2013). 769 Problem Solving. In *The Oxford Handbook of Cognitive Psychology*. Oxford University Press.

McCall, D. and Kölling, M. (2019). A new look at novice programmer errors. *ACM Trans. Comput. Educ.*, 19(4).

Mierle, K., Laven, K., Roweis, S., and Wilson, G. (2005). Mining student cvs repositories for performance indicators. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA. Association for Computing Machinery.

Milem, J. F. (2003). The educational benefits of diversity: Evidence from multiple sectors. *Compelling interest: Examining the evidence on racial dynamics in higher education*, pages 126–169.

National Academies of Sciences, Engineering, and Medicine (2018). *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments*. The National Academies Press, Washington, DC.

National Center for Education Statistics (2017). Summary of source data for code.org infographics and stats.

National Science Foundation (2023). Report - science and engineering degrees earned.

O'Rourke, E., Chen, Y., Haimovitz, K., Dweck, C. S., and Popović, Z. (2015). Demographic differences in a growth mindset incentive structure for educational games. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, page 331–334, New York, NY, USA. Association for Computing Machinery.

O'Rourke, E., Haimovitz, K., Ballweber, C., Dweck, C., and Popović, Z. (2014). Brain points: A growth mindset incentive structure boosts persistence in an educational game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 3339–3348, New York, NY, USA. Association for Computing Machinery.

Osman, M. and Hannafin, M. (1992). Metacognition research and theory: Analysis and implications for instructional design. *Educational Technology Research and Development*, 40:83–99.

Pappas, I. O., Giannakos, M. N., and Jaccheri, L. (2016). Investigating factors influencing students' intention to dropout computer science studies. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 198–203, New York, NY, USA. Association for Computing Machinery.

Prather, J., Becker, B. A., Craig, M., Denny, P., Loksa, D., and Margulieux, L. (2020). What do we think we think we are doing? metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 2–13, New York, NY, USA. Association for Computing Machinery.

Prather, J., Pettit, R., McMurry, K., Peters, A., Homer, J., and Cohen, M. (2018). Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 41–50, New York, NY, USA. Association for Computing Machinery.

Rivera-Reyes, P., Lawanto, O., and Pate, M. L. (2017). Students' task interpretation and conceptual understanding in an electronics laboratory. *IEEE Transactions on Education*, 60(4):265–272.

Rodrigo, M. M. T., Andallaza, T. C. S., Castro, F. E. V. G., Armenta, M. L. V., Dy, T. T., and Jadud, M. C. (2013). An analysis of java programming behaviors, affect, perceptions, and syntax errors among low-achieving, average, and high-achieving novice programmers. *Journal of Educational Computing Research*, 49(3):293–325.

Roscoe, R., Allen, L., and Patchan, M. (2016). Evaluative misalignment of students' writing assessment criteria: Hypothesized consequences and causes.

Sahranavard, S., Miri, M., and Salehiniya, H. (2018). The relationship between self-regulation and educational performance in students. *Journal of Education and Health Promotion*, 7.

Shana, J. and Venkatachalam, T. (2011). Identifying key performance indicators and predicting the result from student data. *International Journal of Computer Applications*, 25.

Sheshadri, A., Gitinabard, N., Lynch, C. F., Barnes, T., and Heckman, S. (2019). Predicting student performance based on online study habits: A study of blended courses.

Smith, A. J., Boyer, K. E., Forbes, J., Heckman, S., and Mayer-Patel, K. (2017). My digital hand: A tool for scaling up one-to-one peer teaching in support of computer science learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 549–554, New York, NY, USA. Association for Computing Machinery.

Smith, R. and Rixner, S. (2019). The error landscape: Characterizing the mistakes of novice programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 538–544, New York, NY, USA. Association for Computing Machinery.

Stevens Institute of Technology (2024). Closing gaps in representation among historically underrepresented groups in computer science.

Swidan, A., Hermans, F., and Smit, M. (2018). Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 151–159, New York, NY, USA. Association for Computing Machinery.

Tuckman, B. W. (2005). Relations of academic procrastination, rationalizations, and performance in a web course with deadlines. *Psychological Reports*, 96(3_suppl):1015–1021. PMID: 16173372.

U.S. Bureau of Labor Statistics (2020). Computer and information technology occupations.

Veerasamy, A. K., Laakso, M.-J., and D'Souza, D. (2021). Formative assessment tasks as indicators of student engagement for predicting at-risk students in programming courses. *Informatics in Education*, 21.

Vellukunnel, M., Buffum, P., Boyer, K. E., Forbes, J., Heckman, S., and Mayer-Patel, K. (2017). Deconstructing the discussion forum: Student questions and computer science learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 603–608, New York, NY, USA. Association for Computing Machinery.

Williams, L. and Kessler, R. (2000). The effects of "pair-pressure" and "pair-learning" on software engineering education. In *Thirteenth Conference on Software Engineering Education and Training*, pages 59–65.

Williams, L., Titus, K. J., and Pittman, J. M. (2021). How early is early enough: Correlating student performance with final grades. In *Proceedings of 5th Conference on Computing Education Practice*, CEP '21, page 13–16, New York, NY, USA. Association for Computing Machinery.

Wrenn, J. and Krishnamurthi, S. (2019). Executable examples for programming problem comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 131–139, New York, NY, USA. Association for Computing Machinery.

Yang, F. and Li, F. W. (2018). Study on student performance estimation, student progress analysis, and student potential prediction based on data mining. *Computers Education*, 123:97–108.

Yarmish, G. and Kopec, D. (2007). Revisiting novice programmer errors. *SIGCSE Bull.*, 39(2):131–137.

Zavaleta Bernuy, A. and Harrington, B. (2020). What are we asking our students? a literature map of student surveys in computer science education. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 418–424, New York, NY, USA. Association for Computing Machinery.

Zimmerman, B. J. and Kitsantas, A. (2014). Comparing students' self-discipline and self-regulation measures and their prediction of academic achievement. *Contemporary Educational Psychology*, 39(2):145–155.

Zweben, S. and Bizot, B. (2022). 2021 taulbee survey. *Computing Research News*, 34.

APPENDICES

APPENDIX

A

TABLES SHOWING HELP-SEEKING RATES BEFORE AND AFTER STRUGGLE

Table A.1: Help-seeking before and during struggle for repositories in the global low count struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	3	0.04	0.09	0.06	0.12
S20	6	0.02	0.17	0.00	0.36
F20	5	0.36	0.28	0.30	0.36
S21	4	0.07	0.07	0.04	0.05
F21	3	0.11	0.14	0.38	0.32
S22	4	0.04	0.11	0.07	0.09
F22	3	0.07	0.04	0.06	0.03
S23	3	0.35	0.30	0.11	0.37
F23	3	0.04	0.16	0.05	0.30
S24	2	0.00	0.00	0.00	0.00
Avg	4	0.11	0.14	0.11	0.20

Table A.2: Help-seeking before and during struggle for repositories in the global high count struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	21	0.5	0.3	0.0	1.0
S20	32	0.5	0.5	0.8	5.0
F20	41	5.0	1.0	8.0	3.0
S21	20	0.4	0.6	0.2	0.2
F21	13	0.1	0.3	1.5	1.3
S22	36	0.0	0.0	0.0	2.7
F22	13	0.5	0.1	1.6	0.3
S23	12	2.4	0.0	1.4	0.0
F23	13	1.2	1.5	1.0	1.5
S24	7	0.6	0.0	0.0	0.4
Avg	21	1.1	0.4	1.4	1.5

Table A.3: Help-seeking before and during struggle for repositories in the global low length struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	4	0.1	0.1	0.0	0.1
S20	6	0.2	0.3	0.0	0.4
F20	6	0.5	0.2	0.3	0.3
S21	4	0.0	0.1	0.1	0.0
F21	4	0.1	0.2	0.3	0.2
S22	6	0.1	0.1	0.1	0.1
F22	4	0.1	0.1	0.2	0.1
S23	4	0.4	0.3	0.0	0.1
F23	4	0.2	0.2	0.0	0.1
S24	3	0.0	0.0	0.0	0.0
Avg	4	0.2	0.2	0.1	0.1

Table A.4: Help-seeking before and during struggle for repositories in the global high length struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	5	0.0	0.0	0.4	0.4
S20	12	0.0	0.4	0.0	1.1
F20	6	0.3	1.0	0.7	0.9
S21	7	0.1	0.1	0.6	0.3
F21	4	0.7	0.2	2.2	0.8
S22	6	0.1	0.0	0.5	0.1
F22	4	0.2	0.3	0.5	0.3
S23	4	0.5	0.0	0.5	0.3
F23	6	0.2	0.9	0.9	2.0
S24	4	0.4	0.0	0.0	0.1
Avg	4	0.2	0.2	0.1	0.1

Table A.5: Help-seeking before and during struggle for repositories in the local low count struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	11	0.1	0.1	0.7	1.1
S20	18	0.0	0.5	0.7	4.3
F20	19	1.3	1.7	5.4	6.1
S21	13	0.5	0.4	1.0	0.9
F21	11	0.3	0.4	2.9	1.9
S22	17	0.6	0.3	1.5	2.0
F22	13	0.3	0.2	2.7	1.2
S23	6	1.3	0.6	0.9	1.4
F23	11	0.4	1.0	2.0	4.0
S24	8	0.0	0.0	0.0	0.8
Avg	13	0.5	0.5	1.8	2.4

Table A.6: Help-seeking before and during struggle for repositories in the local high count struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	70	0.5	5.0	0.8	25.5
S20	93	0.4	9.2	2.0	37.8
F20	94	6.5	62.5	3.8	43.5
S21	62	4.8	2.2	3.6	0.6
F21	48	0.0	4.6	1.4	2.8
S22	82	1.0	24.0	0.8	19.6
F22	50	1.4	10.3	2.0	5.4
S23	38	1.7	16.9	0.8	5.8
F23	81	1.0	27.5	0.0	20.0
S24	25	0.0	2.0	1.7	4.0
Avg	64	1.7	16.4	1.7	16.5

Table A.7: Help-seeking before and during struggle for repositories in the local low length struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	12	0	0.1	0.4	0.8
S20	14	0	0.5	0.4	2.6
F20	24	2	1.6	7.8	6.5
S21	16	0.3	0.4	0.5	0.5
F21	12	0.2	0.4	1.4	1.7
S22	18	0.7	0.1	1.5	0.8
F22	15	0.5	0.4	1.1	1.4
S23	10	1.7	0.6	1.1	1.1
F23	10	0.3	0.5	2.3	2.2
S24	7	0	0.3	1	1.4
Avg	14	0.6	0.5	1.8	1.9

Table A.8: Help-seeking before and during struggle for repositories in the local high length struggle group.

		Before Struggle		During Struggle	
Semester	**Struggle Count**	**MDH**	**Piazza**	**MDH**	**Piazza**
F19	11	0.0	0.0	0.0	2.3
S20	31	0.1	0.7	4.8	14.8
F20	32	0.3	1.5	10.2	20.7
S21	24	0.2	1.1	9.6	4.5
F21	22	0.2	0.0	13.0	6.7
S22	49	0.6	1.1	7.2	4.6
F22	22	0.8	0.4	20.8	8.1
S23	27	2.8	0.8	26.3	16.3
F23	30	0.4	4.2	9.1	25.3
S24	21	0.0	2.0	2.4	3.8
Avg	27	0.5	1.2	10.3	10.7

APPENDIX

B

SURVEY A

Survey administered at the beginning of every semester, composed of the CAS survey (26 questions) with the additional questions from our course (25 questions). All questions are Likert scale questions with possible answers of 1 to 5.

Q.1 After I study a topic in computer science and feel that I understand it, I have difficulty solving problems on the same topic.

Q.2 Errors generated by computers are random, and when they happen there's not much I can do to understand why.

Q.3 If I want to apply a method used for solving one computer science problem to another problem, the problems must involve very similar situations.

Q.4 I can usually figure out a way to solve computer science problems.

Q.5 When I solve a computer science problem, I break it into smaller parts and solve them one at a time.

Q.6 I do not spend more than five minutes stuck on a computer science problem before giving up or seeking help from someone else.

Q.7 There are times I solve a computer science problem more than one way to help my understanding.

Q.8 I think about the computer science I experience in everyday life.

Q.9 Tools and techniques from computer science can be useful in the study of other disciplines (e.g., biology , art, business).

Q.10 When working on a computer science problem I find it useful to brainstorm about solution strategies before writing code.

Q.11 I find the challenge of solving computer science problems motivating.

Q.12 When studying computer science, I relate the important information to what I already know rather than just memorizing it the way it is presented.

Q.13 I enjoy solving computer science problems.

Q.14 Reasoning skills used to understand computer science can be helpful to me in my everyday life.

Q.15 Learning computer science is just about learning how to program in different languages.

Q.16 When I am working on a computer science program, I try to decide what reasonable output values would be.

Q.17 When I'm trying to learn something new in computer science, I find it useful to write a small program to see how it works.

Q.18 A significant problem in learning computer science is being able to memorize all the information I need to know.

Q.19 We use this statement to discard the surveys of people who are not reading the questions. Please select "Agree" for this question to preserve your answers.

Q.20 Understanding computer science basically means being able to recall something you've read or been shown.

Q.21 If I get stuck on a computer science problem, there is no chance I'll figure it out on my own.

Q.22 The subject of computer science has little relation to what I experience in the real world.

Q.23 There is usually only one correct approach to solving a computer science problem.

Q.24 To learn computer science, I only need to memorize solutions to sample problems.

Q.25 I worry that mistakes I make when writing a program may damage my computer.

Q.26 I am interested in learning more about computer science.

Q.27 Generally I have felt secure about attempting computer programming problems.

Q.28 I am sure I could do advanced work in computer science.

Q.29 I am sure that I can learn programming.

Q.30 I think I could handle more difficult programming problems.

Q.31 I can get good grades in computer science.

Q.32 I have a lot of self-confidence when it comes to programming.

Q.33 I like writing computer programs.

Q.34 Programming is enjoyable and stimulating to me.

Q.35 When a programming problem arises that I can't immediately solve, I stick with it until I have the solution.

Q.36 Once I start trying to work on a program, I find it hard to stop.

Q.37 When a question is left unanswered in computer science class, I continue to think about it afterward.

Q.38 I am challenged by programming problems I can't understand immediately.

Q.39 I am taking CSC216 because I'm interested in learning how to program.

Q.40 I am taking CSC216 because I have to.

Q.41 I expect to pass CSC216 with a C or higher.

Q.42 I expect to get an A in CSC216.

Q.43 I am taking CSC216 because I expect it to prepare me for a career in industry.

Q.44 I am taking CSC216 because I expect it to prepare me for graduate school

Q.45 I intend to work in the computer science or software engineering profession after completion of my degree or certificate.

Q.46 I intend to go to graduate school in computer science or software engineering after completion of my degree or certificate.

Q.47 I currently have no concrete plans for after completion of my degree or certificate.

Q.48 How confident are you that you can, by yourself: Identify the first action to take when I run into a program or test failure.

Q.49 How confident are you that you can, by yourself: Use print statement or some other form or output to document program execution to find a fault in my program.

Q.50 How confident are you that you can, by yourself: Step through program execution using a debugging tool to find the fault in my program

Q.51 How confident are you that you can, by yourself: Given feedback about a problem with my program, write a test to help with successfully debugging the fault.

APPENDIX

C

SURVEY B

Survey administered at the end of every project semester, composed of some questions from the CAS survey with additional questions created based on this research. All questions are Likert scale questions with possible answers of 1 to 5.

Q.1 I am confident I can learn the concepts taught in this course.

Q.2 I am confident I can apply the skills being taught in this class.

Q.3 I believe that the teaching staff provides adequate support for me.

Q.4 When I start to struggle with the material or assignments, I generally switch tasks.

Q.5 I feel that Computer Science comes easily to me.

Q.6 I believe working on this course can improve my aptitude for Computer Science.

Q.7 If I get stuck on a problem or assignment, I should be able to figure it out on my own.

Q.8 I believe that I can complete the classwork best on my own.

Q.9 I find the challenge of solving CS problems motivating.

Q.10 I schedule specific times weekly to work on assignments from this course.

Q.11 Before I work on assignments, I divide my tasks into smaller sub tasks.

Q.12 I set a specific goal for each study/work section.

Q.13 I attend this class regularly.

Q.14 When I need help with this course, I often study with classmates or friends.

Q.15 It is my responsibility to understand the course material.

Q.16 It is the teaching staff responsibility to make sure I understand the material.

Q.17 When I seek help from the teaching staff, I expect them to help me until my problem is solved.

Q.18 I tend to seek help as soon as the assignment gets challenging.

Q.19 Do you attend office hours when you need help with assignments?