Measuring the Software Development Process to Enable Formative Feedback

Ayaan M. Kazerouni

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

March 23, 2020
Blacksburg, Virginia

Keywords: Computing education, software engineering, software testing, self-regulation, procrastination

Measuring the Software Development Process to Enable Formative Feedback

Ayaan M. Kazerouni

(ABSTRACT)

Graduating CS students face well-documented difficulties upon entering the workforce, with reports of a gap between what they learn and what is expected of them in industry. Project management, software testing, and debugging have been repeatedly listed as common "knowledge deficiencies" among newly hired CS graduates. Similar difficulties manifest themselves on a smaller scale in upper-level CS courses, like the Data Structures & Algorithms course at Virginia Tech: students are required to develop large and complex projects over a three to four week lifecycle, and it is common to see close to a quarter of the students drop or fail the course, largely due to the difficult and time-consuming nature of the projects. My research is driven by the hypothesis that regular feedback about the software development process, delivered *during development*, will help ameliorate these difficulties. Assessment of software currently tends to focus on qualities like correctness, code coverage from test suites, and code style. Little attention or tooling has been developed for the assessment of the software development process. I use empirical software engineering methods like IDE-log analysis, software repository mining, and semi-structured interviews with students to identify effective and ineffective software practices to formulate. Using the results of these analyses, I have worked on assessing students' development in terms of time management, test writing, test quality, and other "self-checking" behaviours like running the program locally or submitting to an oracle of instructor-written test cases. The goal is to use this information to formulate formative feedback about the software development process. In addition to educators, this research is relevant to software engineering researchers and practitioners, since the results from these experiments are based on the work of upper-level students who grapple with issues of design and work-flow that are not far removed from those faced by professionals in industry.

Measuring the Software Development Process to Enable Formative Feedback

Ayaan M. Kazerouni

(GENERAL AUDIENCE ABSTRACT)

Graduating CS students face well-documented difficulties upon entering the workforce, with reports of a gap between what they learn and what is expected of them as professional software developers. Project management, software testing, and debugging have been repeatedly listed as common "knowledge deficiencies" among newly hired CS graduates. Similar difficulties manifest themselves on a smaller scale in upper-level CS courses, like the Data Structures & Algorithms course at Virginia Tech: students are required to develop large and complex software projects over a three to four week lifecycle, and it is common to see close to a quarter of the students drop or fail the course, largely due to the difficult and time-consuming nature of the projects. The development of these projects necessitates adherence to disciplined software process, i.e., incremental development, testing, and debugging of small pieces of functionality. My research is driven by the hypothesis that regular feedback about the software development process, delivered *during development*, will help ameliorate these difficulties. However, in educational contexts, assessment of software currently tends to focus on properties of the final product like correctness, quality of automated software tests, and adherence to code style requirements. Little attention or tooling has been developed for the assessment of the software development process. In this dissertation, I quantitatively characterise students' software development habits, using data from numerous sources: usage logs from students' software development environments, detailed sequences of snapshots showing the project's evolution over time, and interviews with the students themselves. I analyse the relationships between students' development behaviours and their project outcomes, and use the results of these analyses to determine the effectiveness or ineffectiveness of students' software development processes. I have worked on assessing students' development in terms of time management, test writing, test quality, and other "self-checking" behaviours like running their programs locally or submitting them to an online system that uses instructor-written tests to generate a correctness score. The goal is to use this information to assess the quality of one's software development process in a way that is formative instead of summative, i.e., it can be done while students work toward project completion as opposed to after they are finished. For example, if we can identify procrastinating students early in the project timeline, we could intervene as needed and possibly help them to avoid the consequences of bad project management (e.g., unfinished or late project submissions). In addition to educators, this research is relevant to software engineering researchers and practitioners, since the results from these experiments are based on the work of upper-level students who grapple with issues of design and work-flow that are not far removed from those faced by professionals in industry.

Dedication

To my family, friends, and mentors.

Acknowledgments

It may not look like it, but a PhD is a group effort. The friendship, influence, and support of several people and institutions are what got me to this finish line, and I am eternally grateful. I attempt to name them here, and I am certain that I have forgotten a few. If you are among that number, accept my apologies!

First, I owe thanks to my advisors and advisory committee. I am eternally grateful to my advisor Cliff Shaffer for taking me on as a doctoral student in my first semester at Virginia Tech. I could not have asked for a better advisor—his work ethic and careful attention to detail have been monumentally important in my development as a researcher, science communicator, and mentor to future researchers. I owe thanks to my co-advisor, Steve Edwards, for countless brainstorming sessions that led to much of the research presented herein. Both Dr. Shaffer and Dr. Edwards have an incredible commitment to impactful research, and everything we worked on (within and without this dissertation) was done with that impact in mind. This was a huge motivating factor during the inevitably hard parts of a PhD. I'd like to thank my committee member Francisco Servant for the many discussions in his office that led to more complete research, and for his Software Engineering course, which fundamentally changed the way I thought about my work in particular and research in general. I am grateful to the other members of my committee, Dennis Kafura and Jaime Spacco, for asking important questions that forced me to question my assumptions about computing education.

A PhD would be a bleak affair without an army of friends. Five years is a long time, and there are many names to name. First, I would like to thank Jamie Davis for his friendship, sage advice about life, research, and the pursuit of happiness, and one racquetball championship. I owe enormous thanks to Chandani Shrestha and Nidhi Menon for becoming my family here in Blacksburg and for making the hard part of this ride feel like a breeze. Thanks also to Thomas Lux and Tyler Chang who made dreary work days seem, well, less dreary; Moe Mondays will always hold a special place in my heart. Other friends I owe gratitude to are Tuna Önder, Prashant Chandrasekar, Daniel Chiba, Aakash Gautam, Satyajit Upasani, Anika Tabassum, Apratim Mukherjee, and my fellow members of Panda Bag—Ethan Smith, Wenhui Lee, Greg Lambert, Han Chen, Saikat Mukherjee, and Kirk Broadwell. I am thankful for the Torgersen 2000 crowd, past and present, for their companionship and impromptu discussions about various topics. Additionally, I am grateful to Cory Bart for providing helpful perspectives from having experienced each PhD milestone a few years ahead of me.

Last and certainly not the least, I thank my family, whose support is unequivocally what got me through this journey. I would like to thank my parents, Sharmeela and Mehdi Kazerouni. I am forever grateful for their courage and foresight in making the impossible decision to part with their children as they left to pursue higher education on the other side of the world. Their love and support can move mountains, and has continued to do so from a few thousand miles away.

I would like to thank my uncle and aunt, Beheruz and Madhavi Sethna, for becoming like a second set of parents to my brothers and me, and for accepting us into the fabric of their daily thoughts and lives. This was immensely important when we left home behind and I count myself incredibly fortunate to be a continued recipient of their love. I am also grateful for the monthly cookie deliveries.

I owe huge thanks to my virtual graduate school support group: my brothers Amaan and Ameen, and my soon-to-be sister-in-law Anum. Hard days were softened and good days brightened, and I will always be grateful. Amaan and Ameen have been my best friends from my first moments, and everything I do is owed in huge part to them. In no uncertain terms, none of this would have been possible without them.

Finally, many institutions have played a role in supporting this effort. I thank the faculty and staff at the CS Department at the University of West Georgia for welcoming me into the computing community and whose teaching was instrumental in moulding my research interests. I thank the now defunct Advanced Academy of Georgia, where I obtained a willingness to admit ignorance that is essential for an endeavour such as this. I also owe thanks to the National Science Foundation and the Department of Computer Science at Virginia Tech for providing financial support that allowed me to focus on academics, and the students and instructors of CS 3114 at Virginia Tech for being such a good test bed for research.

This material is based upon work supported by the National Science Foundation under Grants DUE-1245334, DUE-1625425, and DLR-1740765. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graduating CS students face numerous difficulties in their first software development jobs, with reports of gaps between what they learn and what is expected of them in industry [16, 17, 141, 142]. In addition to difficulties with the human aspects of professional software engineering—e.g., effective written and oral communication, collaboration and multidisciplinary teamwork, and knowing when to seek help [16, 28]—reports indicate that students struggle with elements of the personal software development process, such as project management, software testing, and debugging [141, 142]. These knowledge deficiencies are not a recent phenomenon: based on the results of a 1998 survey of practitioners, Lethbridge examined the differences between *knowledge after education* and *current knowledge* [110]. A positive difference indicates *on-the-job* learning, and a large difference might suggest a need for formal education to improve its coverage of that topic. Analysis indicated that on-the-job learning was commonly reported for project management, software testing, and debugging.

Perhaps an early manifestation of these difficulties is the challenges faced by intermediate Computer Science students while trying to complete large and complex programming projects. Every CS student eventually reaches a point in their coursework where they must begin using good program development practices if they are going to successfully complete these assignments. *When* this happens may depend on the individual's ability or prior experience. Some may face these difficulties in a typical undergraduate CS1 or CS2 course. Others successfully pass CS1 and CS2 without developing the necessary software development skills, but reach the limits of undisciplined development practices when they are faced with large and complex software projects like those found in mid-to-upper level CS courses. For example, the Data Structures & Algorithms course at Virginia Tech involves the development of major software projects over a three to four week lifecycle. Relative to projects encountered in previous courses, these projects involve far more complicated design choices, and far greater need for a rigorous testing and development process. As a possible consequence, it is common to see large numbers of unsuccessful attempts at the course: in Fall 2016, 22% of enrolled students ended up withdrawing from or failing the course. For Fall 2018, that number is 28%. Typical prerequisite coursework does not prepare students to develop software of this scale or complexity. In other words, they are expected to learn these skills *on the job*, as it were, as part of this course.

This work is driven by the hypothesis that a stronger pedagogical focus on the software development process will help ameliorate the challenges faced during and after undergraduate

CS education. Learning any skill requires practice, and to maximise learning gains and skill acquisition, this practice must be accompanied by formative feedback [23]. Effective feedback about the software development process could improve students' self-awareness about their development habits, promoting meta-cognition and self-regulation, both of which have been linked to numerous benefits such as increased productivity and improved project outcomes [68, 114]. Changing student behaviour relating to software development practices will require changing the way this material is taught, practised, and assessed. However, without a mechanism to capture the necessary details about a developer's personal development process (as opposed to outcomes of the final product), such feedback is impossible.

I propose the following thesis:

> *Measurable differences in students' software development habits can explain differences in their project outcomes.*

The focus of this dissertation is therefore to capture, characterise, and determine the effectiveness of the software development processes undertaken by students, with the goal of providing them with feedback about their process during development. I use data from numerous sources to arrive at quantitative characterisations of students' software development habits, focusing on two specific aspects: time management and software testing. Both have commonly been cited as deficiencies displayed by newly graduated software professionals [110] as well as introductory and experienced undergraduate students [71]. Data sources include click-stream data from students' integrated development environments (IDE), qualitative data from interviews with students, and information about outcomes like correctness and code quality from an automated assessment tool (AAT). With these data in hand, I determine the effectiveness or ineffectiveness of development habits by empirically investigating their relationships with eventual project outcomes like correctness, test suite quality, total time taken, and on-time/late submission status.

Procrastination is a pervasive problem in undergraduate education, and we believe it is a contributing factor to the difficulties faced by intermediate-to-advanced software developers. I developed metrics to understand when students tend to work on software projects, and I used these metrics to investigate the degree to which they procrastinated while completing these tasks. Software testing, an important aspect of software development, has long been denied the attention it is due in the typical undergraduate CS curriculum. To effectively teach this skill, we need to identify and encourage effective software testing and test quality. I developed methods to measure students' engagement with software testing during their project lifecycles at varied levels of granularity, and measured their relationships with project correctness and test suite quality.

Using the metrics described above, we can conduct observational studies to learn the current software development behaviour of students. Empirical studies help us determine how these habits relate with outcomes like project correctness, test suite quality, time taken to complete projects, and the likelihood of late submissions. In this dissertation, I describe observational and empirical studies, whose results allow us to understand the specific characteristics of student software development habits we ought to be encouraging or discouraging. In the future, the metrics I describe in this dissertation will help us to measure the impact we might have on student software development habits through improved pedagogy, interventions, or feedback mechanisms.

In addition to measuring students' time management and software testing processes, I worked on improving the assessment of software tests using *mutation analysis*, a more robust measurement of test adequacy. I did this by addressing its primary limitation—its computational cost—in the common educational context of using AATs to provide feedback to students about their software tests. This work enables the deployment of improved feedback about software test suites, enabling students to reason more robustly about deficiencies in their testing and to produce more reliable project implementations.

The studies described above enable us to automatically identify effective or ineffective elements of students software development. With this information in hand, we are set up to design and deploy interventions that can provide developers with formative feedback about their development habits, i.e., *as they work on software projects.*

My research methods and findings are relevant to software engineering researchers, practitioners, and managers. My dataset comprises IDE click-streams and program snapshots for hundreds of implementations of relatively large and complex software projects, along with "ground truth" outcome data about correctness, style, test coverage, and time-to-completion. The size and scale of this corpus has allowed for the design of robust experiments, controlling for differences between students and programming tasks (projects). Experiments are based on the work of students who are only two or three semesters removed from professionals entering the workforce, who deal with design and work-flow issues that are comparable to what professionals face in the industry.

This dissertation follows the outline below:

- **Chapter 2**: I situate my work in its broader context, reviewing and connecting literature related to self-regulation, procrastination, software development, and testing.
- **Chapter 3**: I give an overview of infrastructure that was deployed at Virginia Tech to collect the data necessary for experiments.
- **Chapters 4 and 5**: I describe metrics we developed to measure students' time management and software testing practices, and experiments we conducted to learn to their relationships with project outcomes.
- **Chapter 6**: I explore various test adequacy criteria, investigate the feasibility of using mutation analysis to give students automated feedback about their software tests, and propose new mutation approaches that are more feasible for the autograding context.
- **Chapter 7**: I conclude with a summary of our findings and explore avenues for future work that are revealed by the work presented herein.

Statement of attribution. Following the "manuscript" dissertation format, the chapters that follow draw material from published or in-preparation manuscripts on which I am the main contributor and author. I have reorganised some content to form a coherent document.

The papers are:

- DevEventTracker: Tracking Development Events to Assess Incremental Development and Procrastination [98]
- Quantifying Incremental Development Practices and Their Relationship to Procrastination [99]
- Assessing Incremental Testing Practices and Their Impact on Project Outcomes [100]
- Scaling Automated Feedback for Students' Software Tests Using Selective Mutation Analysis (in preparation)

Chapter 2

Literature Review

In this section, I review some related work in self-regulation, software testing, and data mining efforts as they pertain to the practice and assessment of software development. I start in §2.1 with a review of literature relating to self-regulation and its connections with disciplined personal software process. In addition, I briefly summarise Piers Steel's meta-analysis on the nature of procrastination [161], focusing on its causes, correlates, and suggested intervention types. In §2.2 I give a brief overview of literature on tools for educational data mining, focusing on systems developed to track the programming process and metrics that researchers have developed from the collected data. Next, I explore issues relating to the pedagogy of software testing (§2.3), including empirical studies on effective and ineffective software testing practices (§2.3.1), methods to evaluate the quality of software tests (§2.3.2), and efforts to include software testing in the undergraduate CS curriculum (§2.3.3). Finally, I briefly touch upon the reported gaps between what students learn in university and what is expected of them in industry (§2.4).

2.1 Self-Regulation in Software Development

Robillard posits that the development of software involves the use of *topic knowledge*, e.g., the meanings of words or concepts [145]. It also involves *episodic knowledge*—knowledge based on experience and practice, typically applied once the topic knowledge has been acquired. Software development methodology can then be viewed as application of episodic knowledge, after gaining the required topic knowledge (e.g., the syntax of a programming language).[1] It follows that practice is important for developing good episodic knowledge of effective software development process. For practice to be constructive, it needs to be accompanied by feedback [24] that can be reflected upon and reacted to.

A *self-regulating* learner is able to observe, critique, and adjust their own practice. Social cognitive theorists have codified this cycle of *self-observation*, *self-judgement*, and *self-reaction* as sub-processes involved in self-regulation [177]. A student's ability to self-regulate is highly predictive of their academic performance [65]. To be self-regulated, a student's learning must "involve the use of specified strategies to achieve academic goals on the basis of self-efficacy

[1]These knowledge categories are dependent on the learner, e.g., a *for* loop is episodic knowledge to a novice programmer who has just learned about *variables* (topic knowledge).

perceptions" [177]. The development of self-regulatory skills is important for novices to achieve expertise within a specified domain, and programming and software development are no exception [70, 113, 114].

Self-regulatory failures in software development. Unfortunately, novice programmers (typically, those in first-year CS classes) lack meta-cognitive awareness. That is, they are unable to think about their problem-solving process [140], and are therefore unable to "plan, set goals, organise, self-monitor, and evaluate" [177]. Novices in any discipline tend to lack this ability, often because they hold a limited store of discipline-specific episodic knowledge [145] that they can draw from. As a result, considerable effort has been devoted to scaffolding meta-cognition and self-regulation for novices as they work on programming problems [113, 114, 139, 140, 170]. Falkner et al. found that training software engineering students to follow specified strategies improved their task performance [70]. Loksa et al. found that providing students with explicit strategies to solve programming problems improved their productivity, self-efficacy, and performance [114]. Prather et al. [139] and Wrenn et al. [170] found that scaffolding to ensure that students had correctly interpreted problem prompts improved their ability to produce correct solutions, and to produce a more thorough set of test cases.

Developing self-regulatory skills is difficult for novices in any discipline. Novice software engineers (as opposed to novice *programmers*), when they first encounter larger and more complex software development tasks, are faced with self-regulatory challenges that manifest in different ways. For example, they engage in "opportunistic and arbitrary design and implementation" [69], or procrastination [120]. Falkner states that while expert student developers are more likely than novices to use discipline-specific self-regulatory strategies while developing software, they are still only 50% likely to engage in software testing, and they nearly universally reported problems with time management [71].

One might look at software testing as a type of self-regulation in software development. Composing and writing tests offers the developer the opportunity to reflect on the functionality they are about to implement or have just implemented, allowing them to come up with explicit task-specific strategies that they can follow. Indeed, test-oriented development methodologies like test-driven development (TDD) or incremental test last development (ITL) prescribe sub-processes that align with the stated sub-processes involved in self-regulation [177]: a cycle of writing a little solution code and little test code (*self-observation*), running the test (*self-judgement*), and fixing or refactoring the code (*self-feedback*). Edwards argues that software testing will encourage students to reflect on their development, help improve their analytical skills, and lead them away from trial-and-error methods of software development [59].

Procrastination. Procrastination is a pervasive and well-studied phenomenon that Steel called the "quintessential self-regulatory failure" [161]. In his meta-analysis of the nature of procrastination, he defines procrastination by stating that "to procrastinate is to voluntarily delay an intended course of action despite expecting to be worse off for the delay".

Researchers have investigated various causes and correlates for procrastination, summarised in full in Steel's study.

Sufficient evidence exists to suggest that a person's proclivity to procrastinate may be driven, to an extent, by personality traits. In particular, individuals' procrastination tendencies are considerably stable across long periods of time as well as in different situations [8]. In terms of individual differences, low self-efficacy and self-esteem have been identified as common causes for procrastination [10], a relationship confirmed by meta-analysis of the literature [161]. If an individual does not feel prepared or able to complete a task, they are more likely to delay starting it or avoid continuing work on it. Additionally, impulsiveness and distractibility were identified as personality traits that contribute to procrastination.

Task characteristics have also been determined to be a cause of procrastination. In particular, tasks whose outcomes are farther in the future are more likely to invite procrastination than tasks with more immediate value [2]. Additionally, the more averse one is to a task, the more likely one is to avoid it (i.e., to procrastinate) [124]. Tasks that offer more frequent choices or opportunities for decision-making are also more likely to invite procrastination [156].

From the lens of task-related procrastination, we can look at the failure to practise software testing as a manifestation of procrastination on the perceived low-value task of software testing. After all, novice developers are often unable to see the benefits of testing [11, 35, 158]. The frequent use of AATs in undergraduate CS education [136] complicates the situation further: the incentive for students to test their own work is largely removed by the availability of an oracle of instructor-written test cases (e.g., in a system like Web-CAT [60], ASSYST [88], or others.). This leads to software testing losing its perceived value, which could (in theory [8]) lead to students procrastinating on testing.

Interventions to encourage self-regulation. Steel suggests flavours of interventions that might reduce procrastination by specifically targeting a given cause or correlate. For example, he describes *expectancy-related* interventions as those that increase the procrastinator's expectancy of success. Bandura has argued that one's efficacy expectancy is susceptible to persuasive measures [10] like visual or verbal feedback. Indeed, this was borne out in Buffardi's study of influences on undergraduate students' software testing habits [35]. Of the feedback students received, they indicated that the red–green bars showing the thoroughness of their testing were the most important and effective at swaying their testing practices.

Steel also describes *value-related* interventions that could be used to reduce procrastination when an individual believes that a task has low value, either because they are averse to the task or because the task's outcomes are not immediately apparent. In this situation, one might reduce the time until the individual sees value from completing the task, e.g., by setting sub-goals and rewarding their completion. The Data Structures & Algorithms course at Virginia Tech has used this strategy to reduce procrastination in its month-long software development assignments, with some success. Starting in the Fall semester of 2016, projects included three *milestones*—increments of functionality that had to be completed by specified deadlines. Failure to complete milestones by their specified due dates resulted in penalties applied to the project grade. Additionally, the project increments represented an explicit strategy that students could use to complete project requirements. Since many students were encountering projects of this size for the first time, this may have helped reduce instances of procrastination stemming from students not knowing where to start [156]. It also may have increased students' sense of self-efficacy regarding the completion of smaller milestones (as opposed to the entire project) [10]. Similarly, procrastination interventions could be deployed to address students' disinclination to write and run their own software tests.

CS educators have deployed other interventions to address procrastination [86, 120], the lack of software testing [58, 158], and students' meta-cognitive difficulties [114, 140]. Subsequent challenges have led researchers to recognise the need for more granular interventions, leading to the development of numerous data mining tools to better observe students' development processes [84]. This dissertation in particular is concerned with using data from such systems to model aspects of students' software development habits—particularly their time management and testing practices. Once a student's development process can be characterised at a low level of granularity, the next step is to devise effective, theoretically grounded feedback mechanisms that can act at the right time for the right student.

2.2 Programming Process Tracking Systems

In their ITiCSE working group report on educational data mining and learning analytics in programming, Ihantola et al. describe a spectrum of granularity at which programming data can be collected, defined roughly in terms of the size or frequency of data points [84]:

- Key strokes—smallest/most frequent
- Line-level edits
- File saves
- Compilations
- Executions
- Submissions—largest/least frequent

AATs typically collect data at the lowest level of granularity, i.e., submissions. Web-CAT [60] is an AAT that allows students to make multiple submissions to an assignment and receive immediate feedback. This feedback can be about correctness, code style, or code coverage by student-written tests. Web-CAT interacts with a custom Eclipse plugin that allows students to make submissions and download starter projects directly from within the IDE. This model of multiple submissions affords the ability to gather information about the student's development process, such as when a student started submitting it to get feedback and when they finished. It also provides an opportunity for analysis of the differences between submissions, giving a rough idea of a project's development trajectory. What it does *not* do is provide enough insight to answer questions such as *are students practising incremental development?* The frequency with which students make submissions can vary considerably. For example, in Fall 2016, students made an average of 54 submissions per assignment ($\sigma = 39$), with an average of 9 hours ($\sigma = 17$) passing between submissions. Using submission-level data for assessments of development process would require reliance on a data stream of relative sparseness, depending in large part on a given student's propensity to make frequent submissions. To obtain an accurate assessment, we need smaller and more frequent data points collected *during development*, rather than at submission time.

Numerous systems have been developed to collect programming process data at various levels of granularity. Vihavainen et al. developed the *Test My Code* (TMC) plugin for the NetBeans programming environment in order to capture how students go about developing software [165]. It records events whenever the student saves, runs, or tests code using instructor-provided tests. Hosseini et al. used TMC to explore the common problem-solving paths undertaken by novice Java programmers at the University of Helsinki [82]. Hackystat [94] is an open-source project from the University of Hawaii that provides product and process measurements in software engineering situations in education and industry. Marmoset [159] is an automated grading system developed at the University of Maryland. It uses an Eclipse plugin to collect student code and store it in a Concurrent Versioning System (CVS) repository each time a file is saved. Spacco et al. have used Marmoset to analyse students' software testing habits in an effort to better understand the challenges associated with teaching test-driven development [158]. Blackbox is an ongoing data-collection project [31] that collects data about Java compilations by worldwide users of the BlueJ IDE—a programming environment designed for novice programmers. Blackbox remains one of the largest corpuses of programming snapshot data available to computing education researchers today. It has been used in numerous studies [29], including to study novice compilation activity [3], the effects, occurrences, and affordances of compiler errors and error messages [30, 123, 144], and code quality issues in student-written programs [101].

The computing education and educational data mining communities have frequently used data from these systems to develop quantitative metrics for various aspects of programming Most of these systems and their data are used to measure novice propensity for syntactic or semantic errors and their proficiency at recovering from them. Jadud used the compilation behaviours of users of BlueJ to gain a "rough sketch" of novice programming behaviour in the classroom, describing the errors they commonly run into, the time they typically spend programming before re-compiling, and the ways in which they respond to error messages from the IDE [89]. He also developed the Error Quotient, which examines consecutive pairs of compilation events and assigns a score to each pair [90]. The score increases if both compilations include an error, and again if those errors are of the same type. Watson et al. built on the Error Quotient with the Watwin Score, which evaluates a student based on their ability to resolve a specific type of error, compared to the time taken by their peers. It does this by taking into account the time taken to resolve errors, error locations, and the complete error message [167]. Evaluation showed it to be a good predictor of performance, and an improvement over Jadud's Error Quotient. To further improve on the Error Quotient, Becker developed the Repeated Error Density metric as a less context-dependent alternative, particularly for shorter programming sessions where the Error Quotient might be less accurate [15].

A significant portion of previous work attempts to model the programming process—based on compilation (syntactic) errors, semantic errors, or both—for novice programmers. The Normalized Programming State Model (NPSM), developed by Carter et al., models the programming process in much finer detail than the work described above, and has been used to model the programming process of CS2-level students [39]. The NPSM forms a holistic representation of the programming process, using sequences of state transitions and time spent in certain states (editing, test editing, debugging, etc.). It has been used to develop predictive models for various outcomes like assignment performance and overall course performance, that were shown to be improvements over previous measures like the Error Quotient or the Watwin score. Carter et al. also used the NPSM to determine patterns of developer activity in the four days leading up to an assignment deadline [41], finding differences in editing and debugging behaviours between A-, B-, and C-level students. Blending NPSM with models of students' interactions with an online social learning environment revealed that the combination provided higher predictive power than either of its parts [40]. A broader literature review on tools and techniques for educational data mining and learning analytics in programming may be found in [84].

Targeting professional and open-source software developers, Beller et al. developed WatchDog, a family of IDE plugins that collect fine-grained programming snapshots in order to capture and characterise software testing habits [20]. Research regarding measurements of software testing practices, software test quality, and the pedagogy of software testing are presented in §2.3.

2.3 Software Testing

I use Andreas Zeller's definition of testing [174]: *Testing is the process of executing a program with the intent of producing some problem.*

In other words, testing attempts to uncover an as-yet-unknown problem. This is in contrast to *debugging*, where the goal is to uncover a known problem, or "testing for debugging" where software tests are used as a method of debugging, e.g., to reproduce and localise a reported bug.

Software testing is an important aspect of software development, and one that is widely recognised to contribute to software quality [95, 126]. Unfortunately, students in many US universities display a disinclination to practice regular software testing as they work toward project completion [35], and they often show poor testing ability both in the classroom [60, 158] and in their first jobs in industry [42, 142]. Still, testing is not a formal part of the typical CS undergraduate curriculum [95, 154] due in part to the numerous challenges that hinder the pedagogy of software testing. These include a lack of focus on process (Chapter 1), a lack of consensus in the software engineering community on what that process should look like (§2.3.1), and widespread use of weak test adequacy criteria to drive assessments (§2.3.2). We cannot give students effective feedback without validated assessments of their software testing process and the thoroughness of their software tests. In this section, I review the literature related to different test-oriented development methodologies (§2.3.1), various methods of evaluating the quality of software tests (§2.3.2), and efforts and challenges related to integrating software testing into CS and software engineering education (§2.3.3).

2.3.1 Empirical Studies on Software Testing Practices

Test-driven development (TDD) [9, 14] is an agile software development technique popularised in the early 2000s, with a key focus on the writing and running of automated unit tests. As TDD grew in popularity with practitioners, it garnered interest from academia as well, inviting empirical studies into its effectiveness (e.g., [21, 66, 75, 122]). Studies investigating the effectiveness of TDD tend to focus on its "key aspect", *test-first development* [66, 75]. Test-first development involves writing a failing test case before writing the relevant solution code to make it pass [14]. This is in contrast to the traditional *incremental test-last* (ITL) style of development, which involves writing a small amount of solution code and then testing it [67]. This dichotomy between TDD and ITL is prevalent in both industry and academia [7, 75, 81, 106].

The software engineering research community has long disagreed about the effectiveness of TDD, particularly in comparison to ITL methods of development. Over the years, there have been numerous case studies [21, 118, 122, 168] and experiments [66, 75, 76, 83] observing the effectiveness of TDD. Numerous meta-analyses and literature reviews discuss the fact that these studies tend to produce conflicting results [57, 76, 83, 106, 143]. There are a number of possible reasons for this. Hammond describes TDD as not having a clear definition in practice [81], and Aniche et al. report evidence of misconceptions about and non-conformance to TDD in practice [7]. Kollanus [106] describes the interesting trend that controlled experiments—particularly those in an academic setting—tend to find that TDD has no effect on product quality, while industrial case studies tend to find that practising TDD improves software quality. Differing study contexts could be another contributing factor to these conflicting results. Case studies tend to focus on a single implementation of a large project that is difficult to compare to other projects, since implementations are rarely replicated. Findings based on such small sample sizes may not be widely generalizable. On the other hand, experiments involve many implementations of the same programming task, using multiple prescribed development methods in a heavily controlled setting. However, process conformance in an experimental setting is difficult to enforce ([66, 118, 157]), and artificially controlling the environment in which software is developed could introduce threats to validity.

Case studies [21, 47, 81, 122] have found TDD to be an effective method of software development, demonstrating significant defect-reduction in final products. In an industrial case study at IBM, TDD was found to reduce the defect rate by about 50% as compared to a “similar” system built using a different testing approach [122]. Bhat et al. [21] found similar results in two case studies at Microsoft, though they also found that TDD decreased productivity (measured by speed of development). Of course, there is an exception to this trend [118], i.e., a case study that showed no difference in quality after implementing TDD.

On the other hand, several controlled experiments found TDD to be less effective than traditional test-last approaches at reducing defects. Erdogmus et al. [66] designed a controlled experiment in an academic setting to evaluate the effectiveness of the Test-First approach. They found that subjects practising Test-First development showed increased productivity, but no significant difference in average quality of the code produced. Fucci et al. [75] conducted an experiment with professional programmers to determine how external quality and developer productivity are impacted by certain process characteristics. They found that consistently shorter programming cycles were correlated with higher product quality. Notably, the order in which test code and solution-code were written was irrelevant to product quality and developer productivity. Multiple rigorous replication studies also did not show any significant effect of TDD on external quality [74, 76, 146] compared with ITL. For example, in 2020 Santos et al. ran a replication of a 2005 study by Erdogmus et al. ([66]) in which they accounted for various threats to validity from previous studies evaluating TDD (e.g., they had participants solve four different tasks instead of one, and had them apply both TDD and ITL) [146]. Results suggested that the interaction between development approach (TDD or ITL) and the specific programming task had a significant effect on external quality, with the development approach showing no effects by itself. External quality was positively related to TDD and ITL on two each out of four tasks.

Other empirical studies have simply characterised how much and when developers tend to write software tests (as opposed to determining how effective these practices are). Beller et al. instrumented IDEs to collect fine-grained event data, and used the data to answer some useful questions about how software developers go about writing software tests. Their findings suggest that most professional and student developers do not practice testing actively, they spend less time testing than they think they do, and that solution code and test code do not co-evolve gracefully [18, 19]. Levin et al. [111] conducted repository mining and found that solution code fixes are often unaccompanied by complementary test code maintenance. Lubsen [115] and Marsavina [119] have conducted case studies to quantitatively and qualitatively characterise the co-evolution of test and solution code using association rule mining techniques. Others have leveraged software visualisation in order to characterise the evolution of software tests in a project [46, 150, 173]. Zaidman et al. attempt to understand how test code and solution code co-evolve by mining software repositories and creating visualisations of version history [173]. In a series of case studies, they characterised different approaches to testing: 1) periodic phases of heavy testing, 2) periods of synchrony between test code and solution code, and 3) consistent co-evolution of test code and solution code.

2.3.2 Evaluating Software Test Quality

In addition to the process used to develop a test suite, it is also important to assess its quality. By quality one typically means the test suite's defect-detection capability, i.e., its *adequacy*. Defined by Goodenough & Gerhart in 1975, a *test adequacy criterion* is a predicate that defines "what properties of a program must be exercised to constitute a 'thorough' test, i.e., one whose successful execution implies no errors in a tested program" [79]. For example, a test is considered "adequate" by the statement coverage adequacy criterion when all statements in the program are exercised by the test [126]. Zhu et al. provide a thorough treatment of available test adequacy criteria and comparisons between them [176].

Test adequacy criteria help guide software testers in three ways:

- to know which elements of the program to execute (e.g., statements, conditions);
- to know when to terminate testing (e.g., when all statements are executed by tests);
- to quantify test suite thoroughness (e.g., what percentage of statements were executed?)

Numerous test adequacy criteria have been proposed for use in education [1, 58, 78], but they have been limited in terms of their effectiveness [61, 85], their ability to provide incremental feedback [63, 78], or their running time cost [1, 61]. They are briefly described below, with a more thorough treatment in §6.1.1.

Code coverage is a test adequacy criterion that measures the proportion of solution code that was executed at least once by a test suite [126]. This is measured based on different kinds of program constructs (e.g., conditions, statements, methods), with some methods being stronger than others.

An advantage of code coverage criteria is that they are easy to reason about and fast to compute. This is particularly true for its simpler formulations, like statement or decision coverage. However, code coverage measures tend to be generally weaker than other available test adequacy criteria. This can be attributed to the fact that they are only sensitive to the *execution* of testable code constructs, but not to the surfacing of output or program state from those constructs to test outcomes. As a result, code coverage is not correlated with the test suite's actual defect-detection capability [85] and is easily game-able by students [1], with classrooms-full of students routinely achieving code coverage near the 100% mark, regardless of the adequacy of their test suites [151].

All-pairs execution [78] involves running all students' tests against all other students' code and vice-versa. Students' tests are evaluated on their ability to detect known defects in other student's implementations. It is not amenable to incremental feedback, since it would require students to have completed their work before feedback can be computed [63]; it depends on students writing "correct" tests (i.e., those with high positive verification ability) [37, 171]; and relies on each student's code compiling against every other student's tests [63].

Mutation analysis [51] is a fault-based test assessment technique in which small changes (*mutations*) are made to the target program, creating incorrect variants of the original, called *mutants*. For example, a mutant might be created by negating a conditional expression (e.g., changing **`a > b`** to **`a <= b`**). Alternatively, a mutant might be created by simply replacing **`a > b`** with a Boolean literal (i.e., **`true`** or **`false`**). The different kinds of mutations that can be applied are called *mutation operators*.

A primary challenge associated with mutation analysis is its considerable computational cost. The process could involve running a test suite possibly hundreds of times, depending on the number of mutants that are generated. Significant scholarly effort has been devoted to reducing the cost of mutation analysis [54, 121, 131, 132, 155, 162]. A detailed review of these works may be found in §6.1.2.2.

There have been few efforts to apply mutation analysis in an educational setting [1, 61, 152]. Studies have generally found it to be a much stronger method of test evaluation, but one that exacts an intolerable computational cost.

Other work has attempted to judge test suite quality along axes other than defect-detection and positive verification capabilities. Bowes et al. worked with industry partners and compiled a core list of testing principles [25], e.g., the requirement for a test to have a *single point of failure*. They also discuss possible methods for quantifying adherence to these principles for the purpose of assessing test quality.

2.3.3 Software Testing in the Undergraduate CS Curriculum

There have been significant efforts to integrate software testing into all or parts of the undergraduate Computer Science curriculum [43, 58, 95, 154, 159]. Unfortunately, most prior work in this area tends to focus on novice programmers working on small projects, where the benefits of testing are not readily apparent [11], and on “after-the-fact” feedback that focusses on assessing test quality but mostly ignores the testing process [1, 59, 78].

Many have argued against the notion of treating software testing as an isolated topic in the CS curriculum [43, 57, 95]. Jones calls for a holistic inclusion of software testing in the curriculum [95], stating that students absorb testing concepts more readily when they are presented in small doses. Echoing this recommendation, Desai notes that regular, reinforced learning of testing might be better than only an introduction to it at the start of the semester [57]. Christensen argues further that “Systematic testing is not a goal in itself. *Reliable software* is the goal” [43]. He argues that software engineering education consists of “core knowledge” (e.g., *programming*), and “topics” (e.g., *concurrency*), and argues that systematic software testing, though generally treated as an independent topic, should be treated as “core knowledge” in the curriculum.

The benefits of testing are often not readily apparent to students [11]. Buffardi & Edwards suggest that the frequent conflation of testing and debugging might contribute to students’ view that testing is a process for *reactively* fixing faults, rather than for *proactively* avoiding them [35]. Spacco & Pugh argue that requiring students to submit software tests along with their solutions could be counter-productive [158]. For example, in their study, it led students to produce tests toward the end of their project lifecycles, only to fulfil what they perceived as an arbitrary requirement. There is a need for pedagogical methods that make clear to students the value added by software testing. Aniche et al. have made strides in this regard by appealing to authenticity: in their dedicated software testing course at the Delft University of Technology [5], industry experts talk to students about the "real-world" importance of testing and their experiences with it. Other educators have led students in studying high profile software faults—like those connected to the 2009 and 2019 Boeing 737 plane crashes—and how or if software testing techniques might have avoided them.

The automated assessment tool Web-CAT [58, 60] has been used extensively to include testing in the curriculum [136] by requiring that students submit software tests along with their solutions. Students are rewarded or penalised based on the strength of their tests, using metrics such as code coverage. Edwards argued that an emphasis on software testing would help CS students to develop strong comprehension, analysis, and hypothesis-testing skills [59]. It would also lead novice programmers away from trial-and-error (or "impasses and local fixes" [164]) to arrive at correct solutions. Web-CAT's submission model allows students to continue making submissions until they arrive at a working solution, and allows instructors to reveal or withhold hints based on the thoroughness of the student's own software testing. This encourages students to conduct their own testing before using an instructor-provided oracle of test cases to reveal faults. A pilot study showed that, when students were encouraged to practise testing in this way, they produced code with fewer defects per source line of code. Since these effects were observed in an upper-level CS course, Web-CAT has been used throughout the CS curriculum at Virginia Tech [61, 62]. Web-CAT's model of multiple submissions enforces that students include criterion-adequate test suites with each submission.

However, this model has two limitations. First, the test adequacy criteria currently used by Web-CAT are code coverage-based; the tests submitted by students might only serve to fulfil the criterion used, without actually checking the solution code it accompanies. In theory, this would allow a student to write a test that invokes *most* of the codebase, e.g., by using a battery of inputs but not making assertions about their expected outputs. This is possible because of the weak nature of coverage-based adequacy criteria (see §2.3.2). Second, submission-level data gives only a rough idea of the evolution of project's tests, since submissions might be few and far between. More granular data is required to make an accurate determination of the student's adherence to incremental software testing. Spacco & Pugh [158] and Baumstark & Orsega [13] have made progress in this regard by examining students' version control histories for evidence of iterative or test-driven development. In the next chapter, I describe software developed at Virginia Tech to collect such data about students' software development practices, and in Chapter 5 I describe how we used it to measure adherence to incremental testing.

2.4 The Academia–Industry Gap

Numerous researchers have studied scope and possible causes of a perceived "gap" between what students learn during an undergraduate CS degree and what is expected of them in industry (e.g., [16, 17, 42, 110, 141, 142, 163]). Researchers have found that novices struggle with non-technical and collaborative aspects of software engineering [16], testing and debugging [42, 137], and tools for modern software engineering [142]. Others have investigated this from the lens of students' objectives for obtaining a computer science degree and faculties' perspectives on the purpose of the degree [110, 163].

An early investigation into this gap (as it relates to software engineering) was conducted by Lethbridge in 1998 [110]. Reporting on a survey of 181 software practitioners, Lethbridge investigated *knowledge gained through education*, *current knowledge*, and the *importance* of various topics. Initially, Lethbridge measured importance as an average of the *usefulness* of a topic's details and the amount of *influence* that the topic had on the respondent's career or life. Additionally, he considered the positive difference between current knowledge and knowledge gained through education to be a measure of *on-the-job knowledge change*. If a topic was not learned during education, but the practitioner was forced to learn it during employment (i.e., on-the-job knowledge change was high), this suggests that *forced learning* took place, and might suggest that the topic is of high importance. Results indicated that—on a list of 75 topics ordered by the mean amount of forced learning—the topics *Testing, verification, and quality assurance* and *Project management* placed 4th and 5th, respectively.

More recent gaps in new hires' testing and project management abilities have also been frequently reported. This is perhaps not surprising when one considers the numerous challenges associated with the pedagogy of software testing (see §2.3). Radermacher reports that hiring managers have lamented new engineers' lack of software testing ability and their inability to estimate task size and the time required to complete tasks [142]. Unfortunately, in addition to presenting difficulties for newly hired software engineers, these skill deficiencies may be influencing hiring practices and decisions [137, 141].

Begel & Simon have reported new hires' difficulties with non-technical aspects of software engineering [16]. They conducted a case study during which they shadowed eight new software developers at Microsoft during their first six months on the job. They found that students struggled with inter-disciplinary teamwork and knowing when to ask for help.

In a 2020 survey of CS faculty, Valstar et al. explored faculties' perspectives on the academia–industry skills gap and the role of an undergraduate CS degree. They report that a possible reason for the skills gap is the lack of authenticity of undergraduate project experiences [163], which results in students graduating without appropriate project experience. This is a persistent problem because industry practices change much faster than pedagogy [16], and faculty tend to be wary of overtaxing students or course staff with elaborate project experiences [163]. Further, some faculty confided that they had limited industry experience, and that their ideas of industrial software engineering may not align completely with reality, or may have become outdated.

In this dissertation, I propose methods to mitigate the issues described above that relate to technical software development skills. In particular, I explore ways to improve the way we impart time management and software testing skills during a CS education, by enabling formative feedback mechanisms that help students adhere to good practices as they work on assigned projects. The primary hypothesis is that good software process leads to good software outcomes, and that formative feedback about this process can help to improve it.

Chapter 3

Data Collection Infrastructure: *DevEventTracker*

3.1 Description of Collected Data

In this section I describe the data collected by *DevEventTracker*, an Eclipse plugin built on Hackystat [94]. To provide the reader with context for the rest of this dissertation, I briefly describe all aspects of the system. However, my contributions to the DevEventTracker codebase are a portion of a larger effort. See §3.3 for a listing of my contributions.

At Virginia Tech, students use a custom Eclipse plugin to make submissions to Web-CAT and to download starter projects provided by the instructor. We added functionality to the plugin that continuously collects data from (consenting) students' IDEs, not limited by when they make submissions. This continuous data-stream enables us to develop deeper insights about a typical student's programming process. DevEventTracker collects timestamped development events as well as automated snapshots of the project using the Git version control system. Data collected by DevEventTracker are described in detail below (key events are described in Table 3.1).

Edit events. DevEventTracker collects Edit events in real time as a student programs. An Edit event is recorded each time a student saves their work. For each event, some meta-data is included. Each event is accompanied by the current size of the edited source file, in terms of the number of characters, statements, and methods. We also determine if the edit was made to solution code or test code, using a number of static analysis heuristics. For example, a source file is identified as a test file if it contains a Java class whose name ends in `Test`. This naming scheme is enforced by Web-CAT to enable identification of student-written tests, and is therefore a reliable heuristic for our context. For more general cases, a file is deemed a test file if it contains test methods which can be found by their method signatures, the `@Test` JUnit annotation, or the fact that they contain JUnit assertions.

Launch events. Software testing typically involves executions of automated tests, or interactive launches of a program followed by manual examination of its output or resulting program state. DevEventTracker monitors Launches within Eclipse, collecting and recording meta-data about each launch. It records the type of the Launch (execution of test cases vs. interactive execution of the program); whether the Launch terminated normally or with

Table 3.1: Key event types collected by DevEventTracker, the actions that trigger them, and the accompanying meta-data.

Event Type	**Triggering Action**	**Meta-data Included**
Edit events	Save a file in the IDE	File size (in characters, statements, and methods)
		On test code? (Yes | No)
Launch events	Any program execution (normal, test, or debug)	Termination state (Normal | Error)
		Standard output
	Execute tests in the IDE	Test names
		Test outcomes (Pass | Fail | Error)
Debugger events	Set/unset debugger breakpoints	Line number + file
	Step into/step over	Program constructs being stepped into or over
Program snapshots	Save a file in the IDE	Git commit

an error code; and for unit test executions, the names of the tests that were run and their outcomes (passed, failed, or errored out). Finally, Launch events include any printed output that might have been sent to `stdout` during execution, truncated to 80 characters per line and 500 lines to avoid transmission of giant events.

The sequence of tests passing or failing over the course of development provides a representation for how testing aids the successful implementation of a project. This information can be provided to students, who will benefit from an external view of how regular testing would help them successfully complete projects.

Debugger events. DevEventTracker collects data about a student's use of the Eclipse debugger. It records when breakpoints are added or removed, when a debug session is started, a student's actions during that session (*step into*, *step over*, etc.), and when the debugger session is terminated.

Git snapshots. Each student's project has a Git repository associated with it, with the remote repository residing on the Web-CAT server. When a student saves a file, a Git snapshot is automatically captured and sent to the server. This provides us with the ability to make further fine-grained observations about the changes to a project over time, allowing static or dynamic analyses on versions of the source code at arbitrary points in time. Git repositories are maintained separately from the project's working directory, so as to not hinder the student's ability to use Git version control as part of their own development process.

The fact that Git snapshots are saved automatically and invisibly mitigates some of the "perils" of software repository mining [22]. That is, project histories cannot be "re-written" using **`git rebase`** or **`amend`** commands.

Work sessions. We defined *work sessions* based on DevEventTracker data. Work sessions are sequences of activity that are delimited by one hour or more of inactivity. In this context, 'activity' refers to automatically captured events, and they are grouped based on their timestamps. Grouping DevEventTracker into work sessions helps give a better idea of the actual time spent on task (i.e., actively working on the project).

3.2 DevEventTracker in Virginia Tech Coursework

DevEventTracker has been used in numerous CS courses at Virginia Tech, and has collected millions of development events from thousands of students over several semesters. Data collected by DevEventTracker is fine-grained—using the spectrum of granularity from Ihantola et al. [84], it falls between keystroke-level and line-level edit events. This has enabled the quantitative measurements of software development processes described in subsequent sections.

The experiments described in this dissertation are based on the work of students in the Data Structures & Algorithms course at Virginia Tech (CS 3114), which I will refer to hereafter as CS3. For the semesters in which we collected data, the class syllabus required students to program all of their projects in the Eclipse IDE, using the Web-CAT submission plug-in to submit assignments and download starter code. This has been the standard setup for many programming courses at Virginia Tech for several years. The only difference for this research was that the plugin was augmented with data-collection functionality, and students were prompted for consent when they first installed it.

The act of downloading starter code or making a submission creates a link between Web-CAT and a specific project in a student's Eclipse workspace, and this allows Web-CAT to begin receiving data for that project. However, we did not wait until the first submission to begin receiving data. To ensure that we received data from the moment work began on a project, we provided starter code for each project. The starter file did not provide students with any scaffolding in the form of method or class stubs. It only contained a driver method that printed the text 'Hello world!', used to invoke the project. This ensured that we obtained information about how students approached assigned projects, from start to finish.

3.3 My Contributions to DevEventTracker

The following functionality existed before I started working on DevEventTracker:

- Capturing Edit, Debug, and Build events
- Capturing Git snapshots
- Other plugin functionality (submitting assignments to Web-CAT, downloading starter projects, etc.)

Implementation details are described in Joseph Luke's master's thesis [116].

I contributed the following:

- Capturing launch and termination events, including detailed information about printed output, test cases, and their outcomes
- A mechanism for maintaining Git histories that did not hinder students who wished to use Git as part of their independent work flow
- Data reporting was batched and moved to background tasks, so that students would not experience slowdowns while using Eclipse when the Web-CAT server was slow to respond
- Optimised and secured offline storage of data when server connections were unavailable
- Various minor modifications to peripheral functionality

The source code can be found at the plugin's GitHub repository[1], on the *DevEventTracker-Addition* branch. My contributions may be found starting on September 5, 2015.

[1] `http://github.com/web-cat/eclipse-plugins-importer-exporter/tree/DevEventTrackerAddition`

Chapter 4

Time Management in Intermediate Programming Projects

In a previous study, Edwards et al. [62] provided convincing evidence of the expected effects of procrastination on task outcomes ([161])—it led to lower project scores, and increased rates of late submissions. They analysed five years of submissions from the first three programming courses at Virginia Tech. Assignment results were partitioned into two groups: scores above 80% (A/B), and scores below 80% (C/D/F). Students who consistently received either A/B scores or C/D/F scores were excluded from the study, leaving students with projects with both outcomes. Analysis yielded important results. When students received A/B scores, they started earlier and finished earlier than when the same students received C/D/F scores. After normalizing for program length, there was no significant difference in the amount of time spent on each project stemming from starting earlier vs. later.

These findings led to another study [120] that administered three different types of interventions to prevent procrastination, driven in part by Steel's temporal motivation theory [161]. The interventions were short reflection essays after each project, a requirement to set and track scheduling information and progress throughout the assignment, and e-mail alerts regarding progress toward completion. Only e-mail alerts were associated with significantly reduced rates of late program submissions and significantly increased rates of early program submissions. The promise shown by this method was credited to the fact that the emails were relevant to individual students, generated using data from that student's latest submission to Web-CAT. These studies provided encouraging evidence that: 1) Procrastination has a causal effect on lower project scores, and 2) Regular, automatic, and adaptive feedback helps reduce procrastination, and might help change other programming practices.

The studies above, while providing valuable evidence about the effects of various time management strategies, were carried out using submission-level data. With data from DevEventTracker in hand, we are in a position to build upon these results using more fine-grained analysis of development behaviours to determine effective and ineffective time management behaviours. In this chapter, I describe the context in which I conducted experiments (§4.1.2); I define my research questions (§4.1.1); and I describe metrics developed to help answer them (§4.2). Then I describe my analyses and results (§4.3–§4.5) and close with a discussion of results and their implications (§4.6).

Future work can build upon these results to design interventions that are grounded in both empiricism and theory, i.e., by targeting the specific self-regulatory software development habits that are associated with improved project outcomes.

This chapter is based on two research papers that were published at the ITiCSE ([98]) and ICER ([99]) conferences in 2017. I was the main contributor on both papers.

4.1 Research Method

4.1.1 Research Questions

Students procrastinate for a variety of reasons (§2.1), and it would be a worthwhile endeavour to determine common reasons for procrastination on software development projects. However, to enable formative feedback, it is important to first identify when undesirable behaviours (like procrastination) are actually taking place. Therefore, we focused on quantifying the extent to which procrastination manifests, and on determining how this relates with project outcomes. We measure procrastination on various (observable) aspects of software development—writing and executing solution code and test code.

We are driven by the following research questions.

RQ1: How accurate are our measurements of students' development habits? Before making inferences based on our measurements of students' time management habits, it is necessary to evaluate the accuracy with which we quantified them. We qualitatively evaluated our metrics by triangulating our measurements with data from interviews with students about their development habits, and manual inspection of automatically collected Git snapshots (§4.3).

RQ2: When do students tend to work on a project, relative to its deadline? Before attempting to change the behaviours of a population, it is necessary to understand the population's current behaviours. We use descriptive statistics to characterise how and when students tend to go about working on their software projects (§4.4).

RQ3: How are time management behaviours related to project outcomes? Before designing interventions or instructional improvements based on student development habits, it is important to empirically determine their effectiveness or ineffectiveness. We use quantitative analyses to characterise the relationship between students' incremental software development habits and their project outcomes (as they pertain to time management) (§4.5).

4.1.2 Study Context

The data analysed in this chapter were collected from two sections of the CS3 course at Virginia Tech. Students developed their projects in Java using the Eclipse IDE, and were given 3–4 weeks to complete each one. Projects were relatively large, with a median 892 lines of solution code and test code. We include data from all completed project submissions, including data from students who might have withdrawn from the course after completing a project. On the first day of class, we collected informed consent from students. Data from students who did not give consent (less than 4% of the total) were excluded from the analysis. The consent form can be found in Appendix A.

Processing and filtering. Some preprocessing and filtering were necessary to only include data generated while students were actually working toward project completion. During preliminary interviews with students, we found that some students tend to open Eclipse and make changes to their projects several days after final work had been submitted and graded. While there might be educational value to such activities, they are not considered part of the development of a student's final solution. We excluded these edits from our analyses. We also excluded projects that were worked on for less than 1 hour (that is, projects started by students but with no meaningful attempt to finish). After filtering, the dataset consists of the work of 162 students working on 545 programming projects turned in to four assignments. Not every student completed every assignment, since some students dropped the course, and others may have missed an assignment for personal reasons.

4.2 Proposed Metrics of Time Management

4.2.1 Working Early and Often

Before measuring how early and often students work on projects, we must define what it means to "work" on a software project. Software development involves a mixture of cognitive tasks (e.g., designing a solution) and physical actions (e.g., typing code). For example, a student might work on a project by designing a solution on a whiteboard or discussing the problem with peers. To observe all aspects of development, one would need to observe the student in a controlled lab environment. However, it is infeasible to observe hundreds of students in this way. Therefore, the students we study work in uncontrolled conditions, e.g., in their homes or lab spaces. The only actions we can observe at this scale are those that are captured by DevEventTracker. Therefore, we say that a student is working on the project when they are actively taking actions in the IDE. It is important to note that this is an approximation of the actual time the student truly spends working on the project.

With this definition in mind, we can develop measures for when students tend to work on assigned projects. We can consider the student's work as a collection of uniformly-sized edits.[1] Each edit is represented as a timestamp—the time at which it occurred. The mean

[1]In practice, Edit Events as collected by DevEventTracker are typically not uniformly-sized. For ease of conceptualisation, we "expand" each event into a series of single-character Edit events. This is mathematically equivalent to weighting an edit event by its size.

of this distribution of edit times gives us a sense of when the student tended to work on the project. If we represent each edit as the number of days before the deadline it took place, then the mean tells us how many days before the deadline the student tended to work on the project. Mathematically, if E is the set of all edits, then:

$$\text{earlyOften(E)} = \frac{\sum_{e \in E} daysToDeadline(e)}{|E|} \tag{4.1}$$

Therefore, if a student tends to work several days or weeks before the deadline, this metric will have a larger value; and if a student tends to procrastinate until the project deadline is close, this metric will have a smaller value (or possibly negative, if work was done after the project deadline). We use this metric as a quantitative assessment of time management on a software project, and **a larger value indicates less procrastination**.

For example, consider Figure 4.1, which shows how a (real) student distributed their work across the days on which they worked on a given project. This student's Early/Often "score" is 6, because their mean edit time was September 8, which is 6 days before the project deadline on September 14. A sizeable portion of work was done within the period of September 1 to September 8, and daily work was much higher during the last three days of the project lifecycle. This leads the mean edit time to be roughly in the middle of those time periods. The student's score is therefore sensitive to not only the days on which was done, but also to the *amount* of work that was done on those days.

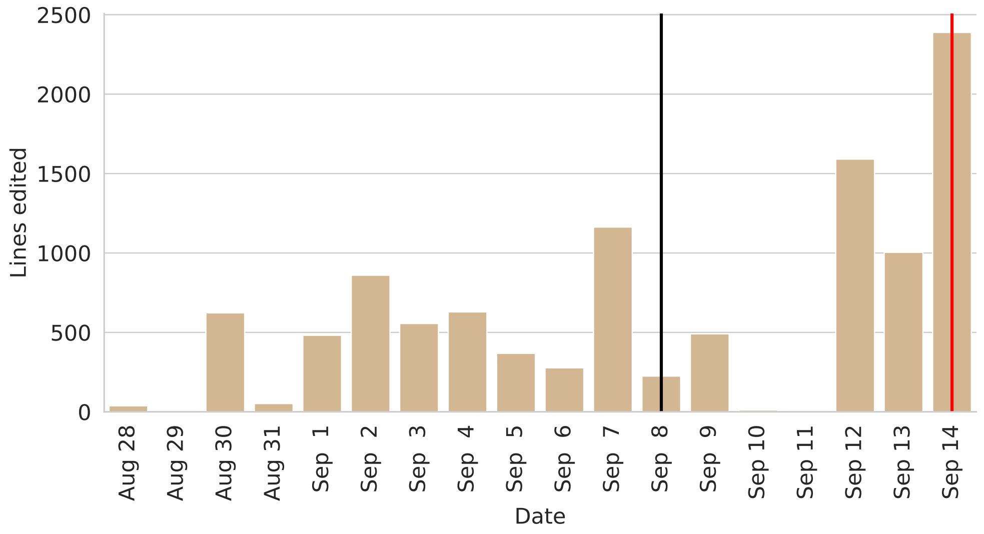


Figure 4.1: Distribution of work from a student on the first CS3 project in the Fall 2018 semester. The red vertical line on September 14 indicates the project deadline. The black vertical line on September 8 indicates the student's mean edit time for this project.

4.2.2 Test Writing

Using Equation 4.1, we can also measure how early and often students conduct software testing by examining when students tend to write and run their software tests. One can measure this by implementing a modified Early/Often Index, applied specifically to Edit events made to test code, or to Launch events involving test cases. Going further, we can measure the difference between the mean time of solution edits and the mean time of test edits. A small number indicates that the central tendency for test editing somewhat closely follows the central tendency for solution editing, while a larger value indicates that the majority of test writing occurred after the bulk of the solution code was written. If $SE \subset E$ is the set of all solution edits, and $TE \subset E$ is the set of all test edits, then this can be measured as:

$$\text{testWriting} = earlyOften(SE) - earlyOften(TE) \tag{4.2}$$

This gives a coarse-grained idea of when students tended to conduct their software testing. A more detailed analysis of how that testing is interspersed with solution implementation is presented in Chapter 5.

4.2.3 Incremental Program Executions

Another key notion of working incrementally is self-checking one's work periodically, as each small chunk nears completion. This might be done by writing and running software tests as one develops, for students who practice incremental testing. Alternatively, it might also involve interactively running a program to confirm its behaviour manually. While proponents of test-driven development argue persuasively that interactive execution is not as effective for checking behaviours, in designing our incremental development metrics we chose to include both possibilities.

The DevEventTracker plugin tracks both interactive program launches and software test executions, and also records the pass/fail outcomes of software tests, providing all of this information in the logged data for analysis. Using these data, we defined the *Incremental Checking* metric as a measure of how often a student self-checks their code by launching it. Here, "launches" are defined as either regular program executions or test executions. First, we represent each uniformly-sized program edit as the number of hours until the next program execution takes place. Then we find the mean of this distribution. So, if E is the set of all single-character edits:

$$\text{programExecutions} = \frac{\sum_{e \in E} hoursToNextLaunch(e)}{|E|} \tag{4.3}$$

One could apply Equation 4.3 to approximate the students' reliance on their own software tests for program checking. A **smaller result would indicate a higher reliance on program or test executions**. Equation 4.3 could also be applied to different combinations of solution or test edits, and program or test executions.

In the next section, I evaluate the signal obtained from these measurements in terms of its accuracy (measured qualitatively) and their relationships with various project outcomes (measured quantitatively).

4.3 Research Question 1

RQ1: How accurate are our measures of students' development habits?

Quantifying software development habits is a non-trivial problem. A primary concern is that there is no readily available "ground truth" against which one can test such a metric. We qualitatively evaluated our measurements by interviewing students and by manually inspecting the snapshot histories captured by DevEventTracker. This work was published and presented at the 2017 conference on Innovation and Technology in Computer Science Education (ITiCSE) [98].

4.3.1 Interviews With Students

In order to evaluate the validity of our measurements, we interviewed students to gather their opinions about their software development habits. We implemented Equations (4.1)–(4.3), and used them to calculate scores for students on the projects they had worked on so far. Ten students representing a range of scores were selected and invited to participate in interview sessions. Of those ten, seven agreed to participate.

We followed a semi-structured interview script (see Appendix A). We asked students in-depth questions about their development habits, focusing on when they tended to work on the project, and when and how they went about testing their programs (e.g., using JUnit tests, manually executing and checking the program, or instructor-written tests). The interviewers were not involved in grading the students in any way, to avoid the possibility of students believing that their answers would affect their course grades. At the end of the interview, students were shown our assessment of their development practice, and were asked what they thought about its accuracy and potential utility in helping them to change their software development habits in the future.

Six of the seven students found our assessment to be **accurate**. The descriptions that follow use feminine pronouns for students 1–7 (S1–S7), regardless of the gender of the participant.

- **S1** stated that she found the model's assessment to be accurate.

- **S2** mentioned that she had been ill and started Project 1 late and worked past the deadline. When the assessment was revealed, we saw that our model had been able to detect this and had given her a low Early/Often score. When the interviewee saw the scores, she agreed with the overall assessment.
- **S3** acknowledged that she and her partner had gotten a late start on Project 1, but that she had worked alone on Project 2 and started relatively earlier. Our model was able to detect this—the interviewee was given a low Early/Often score for one project, and a higher score for the next.
  The student also mentioned that she "didn't write the best tests during the beginning of [Project 1]"; she relied mostly on simple diagnostic print statements for testing and "wrote tests at the end". This is in contrast to Project 2, where she "[brought] in formal testing", since she now had some experience with it. The model's assessment recognized this difference—the student received a poor score for Test Writing (Equation 4.2) on the first project, but a much better score for the second project.
- **S4** received a much lower score for Test Writing on Project 2 than she did on Project 1. Project 2 was almost universally cited as the hardest project that the students worked on this semester (at the time of the focus group, they were starting work on Project 4). The student mentioned that, because the project was so hard, she found herself getting caught up in trying to implement it correctly and ended up writing "more code before testing" than she did on Project 1. This was reflected in our assessment. Also seen was a lower score for Program Executions, which intuitively makes sense—if she was not writing tests until the end, she was not running them, either. Curiously, on a hard project where testing would be most useful, the student brushed it aside in favour of going straight ahead with the implementation.
- **S5** thought the model was *mostly* accurate. Her answers to questions about writing tests did not agree with her failing score for Incremental Test Writing on Project 2. After initially expressing surprise, she backtracked on what she had said earlier by saying that the project involved a lot of recursion, and she tends to test recursive algorithms using iterative print statements rather than formalized testing strategies. This was the only occurrence of a student volunteering new information to explain a score provided by our metric.
- **S6** was the only student who did not find the metrics accurate. The interview revealed disconnects between our assessment and the student's description of her programming practice. However, further investigation revealed transient issues with this student's data reaching the server, which would lead to inaccuracies during metric calculation.
- **S7** received high scores on all metrics, except Test Writing for Project 2. She expressed surprise at her low score for this. The remaining scores were in keeping with her description of her programming practices.

4.3.2 Manual Inspection of Snapshot Histories

While our focus group provided student validation of the measures, we also wanted to directly investigate the edits students were performing in their projects. A second type of evaluation was carried out by manually inspecting the Git repositories maintained by DevEventTracker. Twelve projects were randomly sampled from the pool of submissions. The inspection focused on checking whether our assessment of incremental development matched the "actual programming process" of the student (as seen in the Git revision histories).

Working early and often. Eight of the twelve projects had low scores (< 80 on a normalized 100-point scale) for working early and often. Stepping through commit histories showed that seven of these projects had multiple breaks of several days where no work was done, leading to the project being completed within the last few days before the due date. The remaining project with a low Early/Often score was worked on the day before the project deadline, in a marathon session taking up most of the day. One out of the remaining four projects received a surprisingly high Early/Often score, since the project was started within the last two days. However, it was worked on steadily without breaks, possibly contributing to its high Early/Often score.

Program executions. Most projects received middling or good scores (> 80 on a normalised 100-point scale) for these metrics, but one project received a low score. This project was not launched for the first 10 days in its lifecycle, and launches took place after large amounts of code were written. To evaluate the metrics, we examined a combination of raw DevEvent data and Git snapshots. For two consecutive Launch events, we stepped through revisions for commits made between the two launches. Doing this for several random pairs of launches gives an idea of the usual amount of work done by that student before the program is launched.

Test writing. Five projects received failing scores (< 70) for test writing. Inspecting the file changes over time showed that a majority of testing was done on the last few days of work. Three projects received middling scores (70 to 90) for this metric. Inspection of their commit histories showed that regular testing began after a few days of regular work on the project, but was fairly regular for the rest of the project. The remaining four projects received high scores (≥ 90) for this metric. Their commit histories showed that testing began on the first day of work and continued consistently until the end of the project. Also clear was the fact that test classes were usually created and edited within a few minutes of their corresponding solution class.

This method of validation, carried out on a separate set of projects by manually inspecting the code edits of students through Git snapshots, produced a similar result as the interviews: our metrics do track the incremental development behaviours they were designed to capture, although there is certainly room for improvement of accuracy.

4.4 Research Question 2

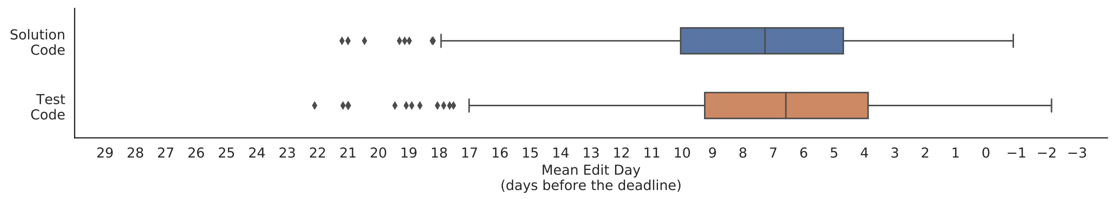
RQ2: When do students tend to work on software projects, relative to their deadlines?

Changing aspects of a population's behaviour necessitates knowing the current behaviours of that population. In addressing this question, I establish what the software development habits of the current population of CS3 students look like. The work for this and the next research question were published and presented at the 2017 International Computing Education Research (ICER) conference [99].

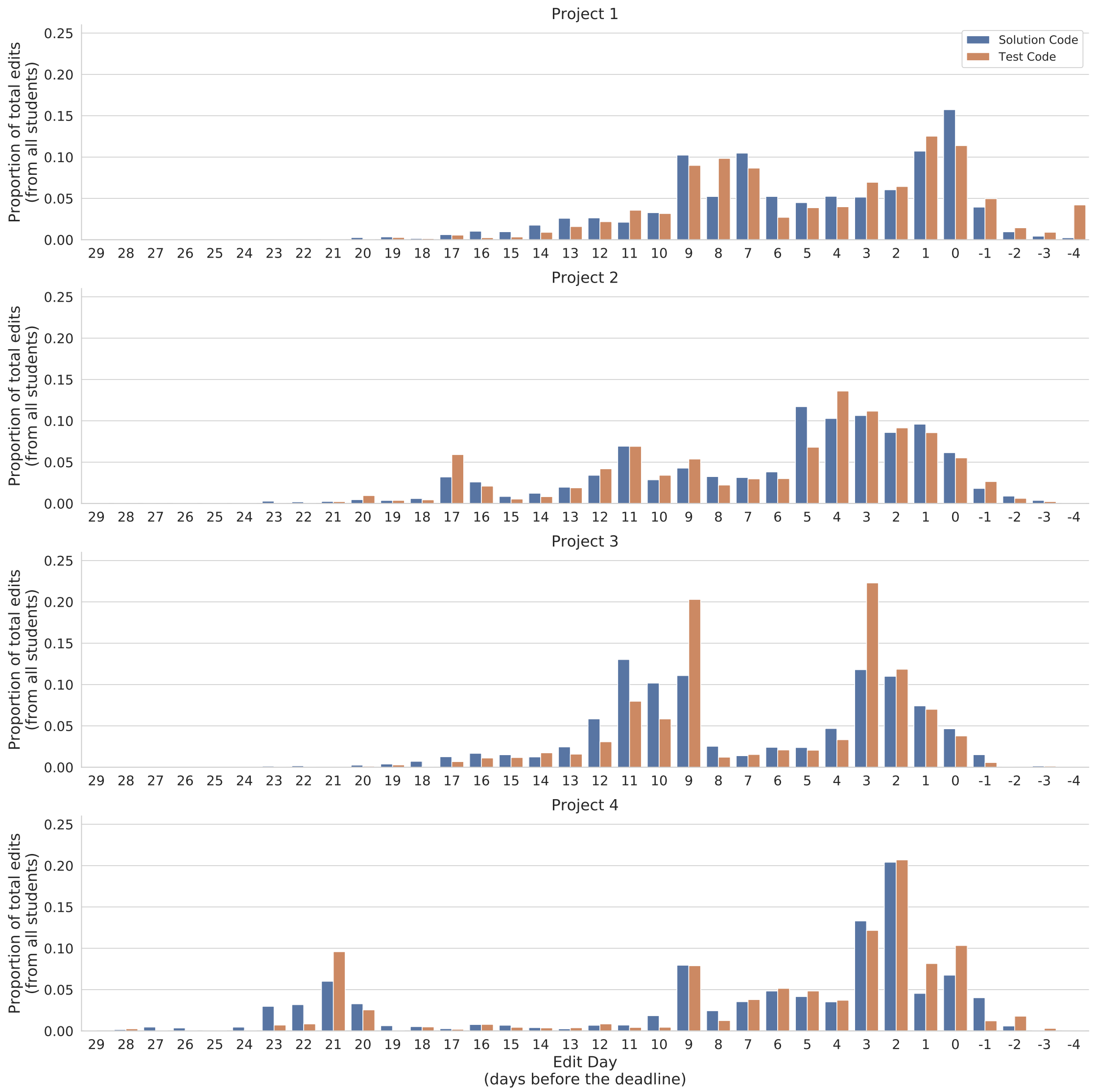
Students tend to work on projects less than 10 days before the deadline, even though they are given nearly 30 days to work on projects (see Figure 4.2). Figure 4.2a depicts the distribution of students' mean edit times for solution code ($\mu = 8.48, \sigma = 6.44$) and test code ($\mu = 7.78, \sigma = 7.04$). Test code tends to be edited slightly closer to the project deadline, but this difference seems negligible. We can explore this further by comparing the actual distributions of work in Figure 4.2b. The horizontal axes represent days in the project timeline in terms of the number of days before the deadline, and the vertical axes represent the work done by *all* students on the given work day. Blue bars represent edits made to solution code and orange bars represent edits made to test code. Naturally, each work day involves more solution code editing than test code editing. To facilitate comparisons between their edit times, Figure 4.2b depicts standardised distributions: the bars represent the proportion of the total amount of (solution or test) code editing that took place on that day. Like Figure 4.2a, Figure 4.2b also suggests that test code and solution code tended to be edited on the same days, a trend which is more or less consistent across all four assignments.

Using Equation 4.2, we confirmed that the "edit day" difference between solution code editing and test code editing was insignificant ($\mu = 0.68, \sigma = 2.96$). Indeed, analysis revealed a high correlation between the two distributions ($R = 0.91$). Recall that this is at the "day" level of granularity—a more detailed analysis of students' incremental test writing habits can be found in Chapter 5. Similar distributions were observed for program executions ($\mu = 8.86, \sigma = 8.82$) and test executions ($\mu = 7.09, \sigma = 7.10$).

Experience led us to expect that students would tend to use interactive program executions (as opposed to test executions) to check their work. However, we were surprised to find that students used software test executions much more commonly than interactive program executions. Test executions were significantly more frequent than normal solution executions ($t = 13.977, p < 0.0001, \text{test} = 229.23, \text{normal} = 55.66$). 83% of projects had more test executions than solution executions, and test executions made up approximately 80% of all executions across projects. Using Equation 4.3 with respect to solution code edits and *any* kind of program launch, we found that, on average, 1.37 hours ($\sigma = 3.31$) passed between edits and executions (largely dominated by test executions due to their much higher frequency).



(a) On average, students tend to write code less than 10 days before the project deadline. Note that the horizontal axis decreases from left to right, to facilitate comparison with Figure 4.2b.



(b) The proportion of work done (edits made) by students on each day of the project timeline, from the start of the timeline to the last acceptable late day.

Figure 4.2: Aggregated and actual distribution of work days. *Edit Day 0* is the due date.

4.5 Research Question 3

RQ3: How are time management behaviours related to project outcomes?

In previous sections, I defined measures of when different programming behaviours were undertaken (§4.2), evaluated them qualitatively (§4.3), and examined the behaviour of our student sample using descriptive statistics (§4.4). In this section, I describe quantitative analyses we conducted to learn the relationships between measured behaviours and the following project outcomes:

- **Project correctness**: The percentage of instructor-written reference tests passed by a student-written solution. This excludes point bonuses or penalties due to early or late submissions, and points gained for the strength of the student's test suite.
- **Finish time**: The time when the project was completed (i.e., the timestamp of the last event transmitted before the project was submitted for grading).
- **Time spent**: The amount of time spent on the project. To measure this, we used the total length of all work sessions (as defined in §3.1). This ensured that idle time spent (i.e., not interacting with the IDE) did not inflate time on task estimates.

Results for all statistical tests presented herein use $\alpha = 0.05$ to determine significance.

We do not make inferences by comparing the behaviours of different students with each other. Different student's behaviours *and* outcomes could be symptoms of some other unknown factor (e.g., differing course loads or prior experience), making such inferences weaker. To test for relationships with the outcome variables, we used a linear mixed-effects ANCOVA [12]. Students were subjects, and assignments served as repeated measures (with unequal variances), allowing within-subjects comparisons in the ANCOVA. In other words, each student's software development habits were measured repeatedly (assignments), and differences in outcomes for the same student were analysed.

4.5.1 Working Early and Often

Project correctness. We found that solution mean edit times were significantly related to project correctness ($F = 16.2, p < 0.0001$). In other words, students who worked on their solution earlier were more likely to produce more correct programs. This is consistent with the earlier result from [62], but now using actual development log data instead of just submitted work.

We illustrate this relationship below. First, notice in Figure 4.3—which shows the distribution of correctness scores—that there is a clear trough just under a perfect score, with approximately half (47%) of the class scoring very close to perfect, and the remainder (53%) scoring noticeably lower. By choosing a cutoff of 95%, we can partition the class into projects that have successfully "solved" the behaviour required for an assignment, and those that have

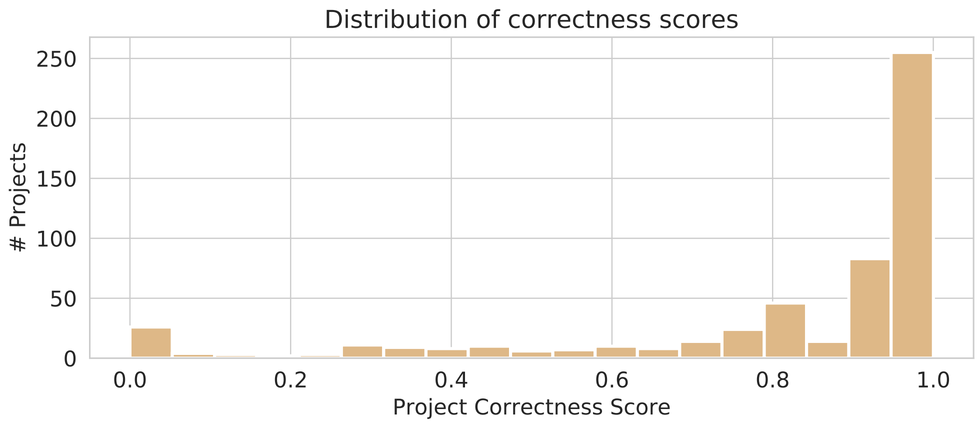


Figure 4.3: The 95% correctness mark splits the class roughly into half.

imperfect solutions. As a result, we will examine differences in key metrics between projects that achieve this threshold and those that do not. Figure 4.4 shows the distribution of solution mean edit times for projects with greater than 95% correctness scores versus those with lower scores.

Note however, that Figure 4.4 includes *all* students, and is therefore affected by inherent student traits, unlike our mixed-effect model. To address this, we consider the within-subjects difference in Early/Often scores (edit mean time) between when students solved the project (scored $\geq$ 95%) and when they didn't. Students who consistently fell into one of the groups—i.e., those that scored $\geq$ 95% on all assignments, and those that scored $<$ 95% on all assignments—are excluded, leaving behind 84 students, each of whom produced submissions that appear in each group at least once. For each of these students, the average Early/Often score for their "unsolved" projects is subtracted from their average Early/Often score for their "solved" projects This gives us an idea of the typical difference between *a given student's* mean edit time between successful and unsuccessful project attempts.

The distribution of these differences is in Figure 4.5. The figure indicates that when students successfully solved an assignment, they tended to work approximately 2 days earlier than they did on other assignments that they did not successfully solve (median difference = 2.11 days). Note also that there are the exceptions to this trend: 19 students (22.62% of these 84) showed a *negative* difference in mean time between solved and unsolved projects, i.e., they worked *later* on solved projects than they did on unsolved ones.

Though §4.2.1 uses only the mean to characterise the distribution of a student's work, we can also use other measures of central tendency, like the median. However, the mean is more sensitive to potential skew in the data, which can play an important role in cases where procrastination leads to more work being done late in the lifecycle.

When both the solution edit mean and median times were considered, the median was not

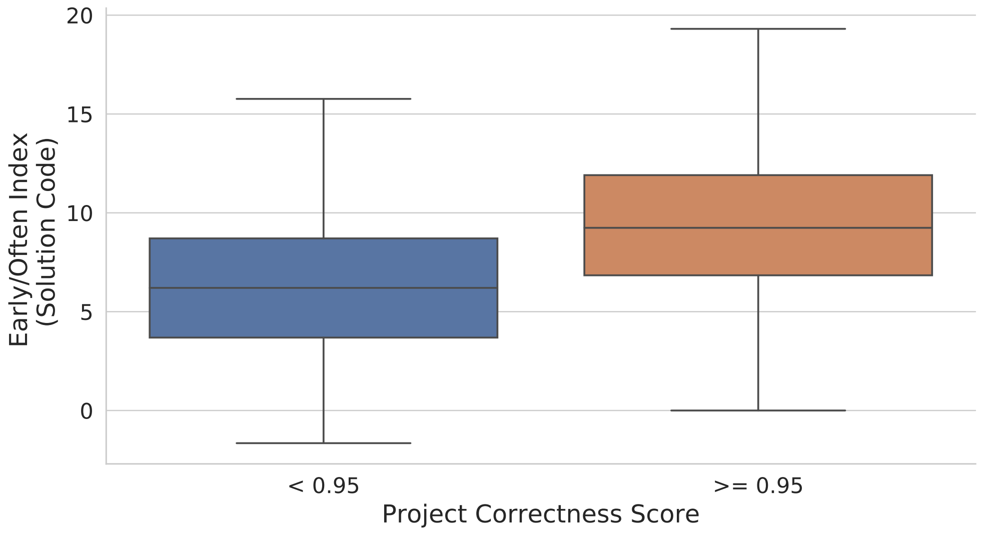


Figure 4.4: Comparison of solution edit times between projects that correctly solved an assignment, and those that did not.

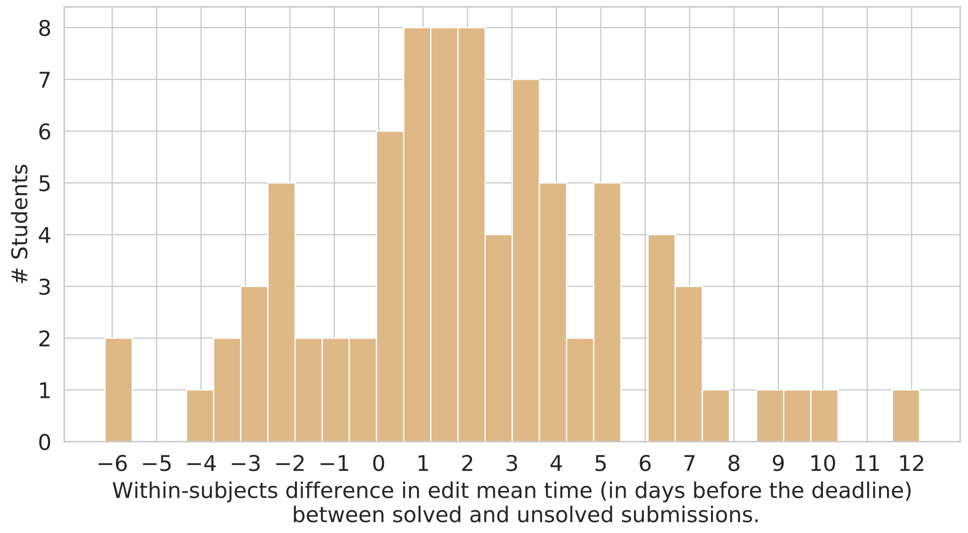


Figure 4.5: When students successfully solved projects, they worked approximately 2 days earlier than they did when they were unable to solve projects.

significant ($F = 0.73, p = 0.39$). However, the same ANCOVA indicated that the test edit median time was also significantly related to project correctness ($F = 10.0, p = 0.0018$; the mean was not significant: $F = 0.06, p = 0.80$). These differences were present even when controlling for student variability using a within-subjects test, indicating that they are not simply due to individual student traits.

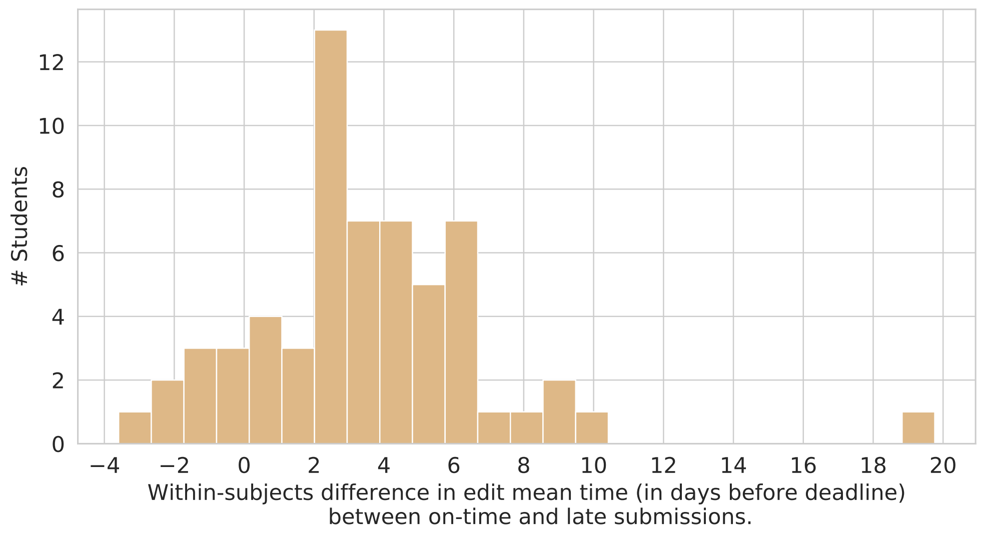


Figure 4.6: Within-subjects difference between edit mean times between early and late submissions.

Finish time. We used the number of hours before the deadline when the student's final work was submitted as a dependent variable in a mixed-effects ANCOVA. We found that both solution mean edit times ($F = 55.9, p < 0.0001$) and solution median edit times ($F = 28.7, p < 0.0001$) were significantly related to finish time, with earlier early/often scores corresponding to earlier finish times. This is as one would expect, since working earlier does allow a greater opportunity to finish earlier. This is also similar to the results in [62], where earlier submission times were associated with earlier completion times.

Figure 4.6 illustrates this relationship by showing the distribution of differences in Early/Often index between projects that were completed on time and those that were completed late. Like Figure 4.5, this figure only includes the 61 students who produced projects that that were on time *and* projects that were late, and the differences depicted are within subjects. The figure indicates that when students ended up submitting their projects on time (i.e., before the deadline), they tended to work about 3 days earlier than when they submitted their projects late (median difference = 3.09 days). Again there are the exceptions: 9 students (14.75% of these 61) worked *later* on the projects that they were able to submit on time.

Total time spent. Using the number of hours spent as the dependent variable in the ANCOVA, we found that only the test edit time median ($F = 10.8, p = 0.001$) was significantly related to total time spent, with earlier edit times associated with slightly longer total time spent.

To sum up, when projects had higher Early/Often scores (i.e, earlier solution edit mean times), 1) they tended to be more correct, 2) they tended to be finished earlier, 3) with no difference in the amount of time spent on the project. A similar repeated measures ANCOVA revealed no evidence of a significant relationship between time spent and project correctness ($F = 1.9, p = 0.17$).

4.5.2 Test Writing

As with the early/often means and medians, for incremental test writing we used the same mixed model ANCOVA with assignments as repeated measures over students as subjects. With the incremental test writing metric as a continuous independent variable, we found no evidence for a relationship with project correctness ($F = 2.54, p = 0.11$), finish time ($F = 0.17, p = 0.68$), or time spent ($F = 0.29, p = 0.59$). It appears that the median time of test edits is more important than that test edits be "close" to solution edits, since test edit median time is significantly associated with project correctness.

At the same time, the DevEventTracker data can be used to provide visual analysis of a student's programming process. Helping students to visualize their own programming process and to compare that against their peers might encourage them to introspectively consider where improvements could be made. This could provide useful feedback during project life cycles. Figures 4.7, 4.8 and 4.9 show "skyline plots" of the programming process for projects with varying levels of incremental test writing and procrastination. The plots depict step-functions for the amount of test code and solution code written over time. The width of each step is the length of the work session, and the height (from the x-axis) is the amount of code written in each work session. Each work session is separated by at least 3 hours of inactivity.[2] Therefore, work sessions that look as though they lasted multiple days (particularly in Figures 4.7 and 4.8), appear because the student settled down to work on the project multiple times, without stopping for a period of at least 3 hours.

The dashed vertical lines represent project milestone due dates (**M1**, **M2**, and **M3**). These milestones are intermediate due dates, with minor grade penalties attached if a given milestone's requirements are not met by the due date. Typically, milestones are defined in terms of some number of reference tests passed, and percentage of solution code lines covered by student unit tests. Dashed vertical lines are also shown for (**E**), the "early bonus deadline" (students who make their final submission by this deadline are given a bonus in their total project score), and the actual project deadline (**F**).

[2]This is a larger threshold than the threshold of 1 hour used to calculate time spent on projects, which results in more dense skyline plots that are harder to understand.

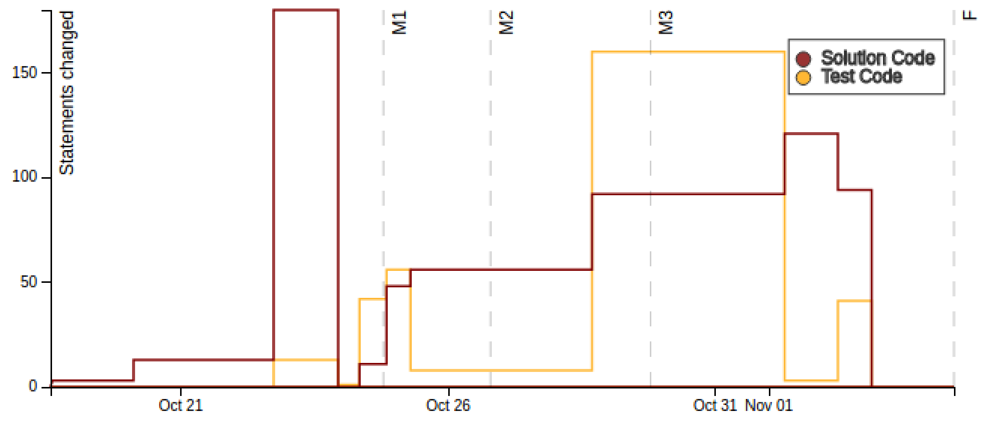


Figure 4.7: An example of a project with unsatisfactory test writing—notice the spike in the amount of test code written as the due date approaches. This project was in the 45th percentile for Incremental Test Writing, and in the 49th percentile for the Early/Often Index.

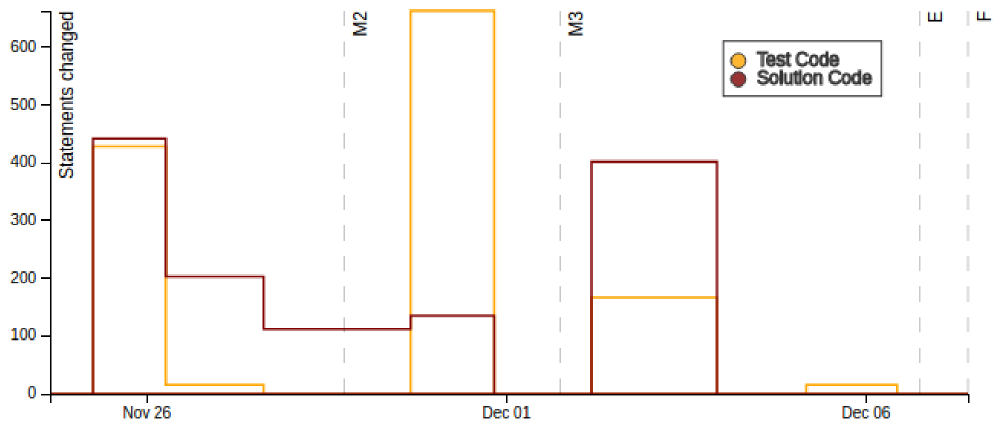


Figure 4.8: An example of a project with an intermediate metric score for test writing, with room for improvement. Notice the irregular bursts of test code writing, and that work started after the first Milestone was due. This project was in the 64th percentile for Incremental Test Writing, and in the 69th percentile for the Early/Often Index.

4.5.3 Program and Test Executions

When examining the relationships between the program executions and the identified outcome variables, we found no evidence for a significant relationship with project correctness, finish time, or time spent. We explored alternative measures, including mean and median times for both interactive program launches and software test executions relative to the due date, and also found no significant relationships.

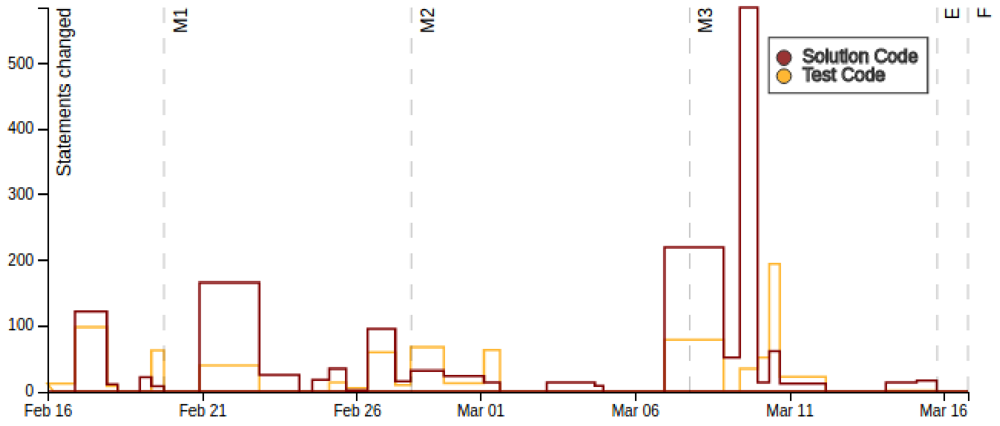


Figure 4.9: An example of a project with model scores under the test writing metric—notice how the test code and solution code follow similar patterns over time. This project was in 90+ percentile for both Incremental Test Writing and the Early/Often Index.

4.6 Discussion

If students are working on projects primarily during the last third of a 30 day timeline, do they really need all 30 days? Figure 4.2 indicated that students tend to do their work within the last 10 days of a typically 30-day project timeline. While it is possible that students do not *need* 30 days to complete projects, we must consider the fact that students are putting in roughly 35 hours of work on each project (see Figure 4.10). I believe that we should encourage students to spread out these 35 hours of work. In other words, I think a 30 day timeline may still be appropriate, and we as educators should work to encourage more productive usage of that timeline. This is particularly important since results presented in §4.5 suggested that working early and often may lead to improved project correctness.

Why are earlier median test edit times associated with higher amounts of time spent on projects? In §4.5, we found that when students had an earlier test edit median time (as described in §4.2.1), they tended to spend more total time on the project. It is notable that the median (not mean) was significant in this case, since the median is less sensitive to skewing when there are outliers very early in the development process but more editing occurs in a smaller time frame closer to the deadline. The median edit time marks the point at which half of the edit activity has already been completed, regardless of its distribution over time. This might mean that students who do a significant portion of the work earlier have more opportunities to invest time on the project later. Or, instead, it may be that students who start very early have to spend more time figuring out details that are only clarified in the assignment specification for everyone else at a later date. e.g., due to frequently asked questions on a class forum.

Why does performance on the Test Writing not reveal a significant relationship with project

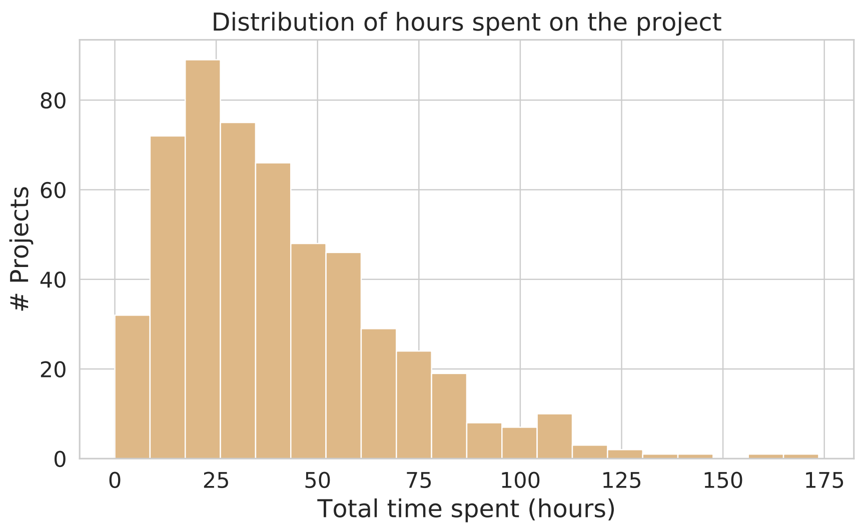


Figure 4.10: Students spent a median 34.45 hours on each project.

outcomes? The Test Writing metric (§4.2.2) measures the difference between the solution edit mean time and the test edit mean time, in terms of days before the deadline. Scores on this metric were not significantly associated with better or worse project outcomes. My conjecture is that this metric is too coarse to paint an accurate picture of a student's test writing habits. The metric might be missing students' cycles of development and testing, which could be much shorter than complete days. It is also possible that the "day distance" between development and test writing activities does not explain variance in project correctness because students might be testing using other methods, e.g., manual executions or submissions to Web-CAT. However, in §4.5 we found that the mean time between code-writing and program executions was not a significant predictor of project outcomes, either. I explore students' test writing activities at a more fine-grained level in Chapter 5.

How might we use these metrics to operationalise feedback about incremental development and time management? Using the metrics described above to change students' development habits requires the development of interventions or feedback mechanisms based on them. For example, a predictive model based on these metrics could aid in early identification of a procrastinating student during a project lifecycle.

As a first step, we conducted a response surface using the project correctness score as the continuous dependent variable to be predicted. Because both solution edit mean times and test edit median times were related to project performance, we used them as continuous independent variables. This model was statistically significantly related to correctness scores. We used its prediction equation as the input to generate a partition model to classify students' solutions as either "solved" (in the group scoring greater than 95% correctness) or not. This predictive model was 69% accurate at classifying the projects in our sample (where SE

represents all solution edits and TE represents all test edits):

$$(0.733 + 0.022 * \text{earlyOften}(SE) - 0.007 * \text{medianTime}(TE)) > 0.83$$

While this is by no means a validated prediction model, it suggests that such models can achieve some degree of accuracy. Developing and validating appropriate feedback mechanisms is important future work, and this chapter lays the necessary groundwork by identifying the most appropriate measures on which to base feedback.

4.7 Threats to Validity

Internal. Without control and experimental groups, we do not claim causality between our process measurements and project outcomes. However, our sample of students and assignments is large and does not suffer from selection bias, since the course is a required part of the CS curriculum and we included all consenting students in the study (>96%). Additionally, our within-subjects experimental makes it unlikely that our findings were due to traits inherent to students, like prior computing experience, course loads, or other demands.

External. Our findings may not be generalisable to all junior level student programmers. However, we take advantage of the intuitive and well-known nature of the effects of procrastination to lend legitimacy to our findings. We observe that the CS 3 course and the sample of students it provided seemed typical of our long experience with the course.

Construct. Interviews with students and data anomalies (e.g., projects whose solution edit median time was earlier than their start date, or entire projects that were completed in one or two edits) indicated that there were transient data transmission issues during the semester. They seemed to have affected fewer than 10 students out of 157, so we do not believe this to be a serious threat to validity.

4.8 Summary

Working earlier and more often was related to earlier final submission times, reduced likelihood of late submissions, and improved project performance. There was no significant relationship with the total time spent on the project. Together, these findings strongly suggest that working earlier and more often, while not resulting in *more* time spent on a project, can result in *more constructive* time spent on a project. Our experimental design further strengthens this conclusion, since factors unique to individual students (like work habits, competing demands on time, or others) were controlled for by the within-subjects analysis. Finally, earlier median test edit times were associated with more time spent on the project in total, and the time or frequency of program or test executions had no significant relationships with any project outcomes.

Chapter 5

Incremental Testing in Intermediate Programming Projects

In the previous chapter, I described analysis into students' time management habits on software projects and their relationships with project outcomes. This included understanding *when* students engaged in different development activities: when they programmed, when they tested, and when they launched their programs and their tests. In this chapter, we dive deeper into students' test writing habits, drawing inspiration from the literature on software engineering and testing to help form our hypotheses and research questions.

Ideal incremental development requires that small chunks of solution code are written, tested, and debugged in an iterative fashion. The order in which these take place largely determines whether the development is categorised as test-driven development (TDD) or incremental test last (ITL) development. In §2.3.1, I described conflicting results from case studies and controlled experiments that presented arguments for, against, or ambivalent toward test-driven development (TDD)—the general trend being that case studies tend to fall in favour of TDD (e.g., [21, 118, 122, 168]) and controlled experiments tend to be inconclusive (e.g., [66, 75, 76]). Differences and challenges in study contexts have been cited as possible reasons for these conflicting findings.

Controlled experiments often take the following steps: 1) develop a large project using TDD, or some other test-oriented development methodology, 2) gather product measures such software quality and developer productivity, and 3) compare product measures with a "comparable" project.

This approach is not without limitations. Sample sizes are necessarily small, since implementations are rarely replicated for the sake of experimentation. Additionally, it is difficult or nearly impossible to accurately compare work done on different projects, possibly worked on by different teams, under different managers. Finally, it is sometimes unclear what TDD is being compared to (for example, "a more traditional, ad hoc unit testing approach" in [122]), making it difficult to interpret results.

Case studies typically include: 1) A controlled lab setting of some kind, 2) A control group developing a piece of software using a "traditional" development approach, 3) An experimental group developing the same piece of software using the development methodology that is being evaluated (often TDD).

There are numerous challenges associated with observing and measuring testing process in a controlled lab or workshop setting. It is difficult to enforce process conformance [67, 118, 157] in such a setting, which casts a certain level of doubt on conclusions. Programming tasks are often "toy" problems which can be completed in a few hours, in which the effects of TDD (or any other test-oriented development methodology) may not become apparent. Finally, measurements could be affected by the *observer effect*—subjects' testing processes may change if they know that they are being observed.

If we are to teach software testing process to students, and we need to assess their testing process in order to formulate feedback, then we need more solid findings on which to base our hypotheses. In this chapter, I describe an empirical study we conducted to examine the testing habits of intermediate-to-advanced undergraduate students and their impacts on project quality. The study avoids many of the challenges faced by both controlled experiments and case studies. Like many controlled experiments, our sample size is large—over 400 project implementations from over 150 students. We avoid the pitfalls of trying to enforce process conformance by not assigning specific development approaches to groups of students (aside from the requirement that they practice *some* testing). Instead, we examine the natural and uncontrolled programming behaviours of our students. Like most case studies, the projects we study are relatively large, complex, and long-running. Finally, we make "apples-to-apples" comparisons, since we compare multiple implementations of each project, and multiple projects implemented by the same developers, by using a crossed random effect experimental design.

In our study, we found that both project correctness and code coverage were positively related to higher testing effort in each work session for a project, and to higher total testing effort on the project. Additionally, we found that when more test code was written for recently changed solution code (i.e., in the same work session), the project tended to have higher code coverage. Both findings provide evidence that supports the conventional wisdom about continuously writing tests alongside solution code. While we do not strictly measure adherence to *test-first* development, we found that writing more test code before finalizing the relevant solution code was irrelevant to project correctness, and negatively related to code coverage. Finally, these behaviours explained a statistically significant but ultimately small percentage of variance in project quality, with inherent variation among students and programming tasks accounting for a larger proportion.

This chapter provides the following contributions:

- A family of metrics to faithfully capture the extent to which students achieve balance and sequence of testing and solution coding effort
- An empirical study that measures the extent to which these metrics are related to successful project outcomes (correctness and test suite coverage)

This chapter is based on a research paper that was published the SIGCSE Technical Symposium in 2019 ([100]). I was the main contributor on the paper.

5.1 Research Method

5.1.1 Research Questions

In our empirical study, we study the relationship of eventual software quality with the balance *balance* of effort devoted to writing solution and test code (a key tenet of test-oriented development [67, 75]), and the *sequence* of effort devoted to solution code and test code (a key concern of researchers and practitioners interested in test-first development [14]).

We study the following research questions:

RQ1: To what extent are students practising incremental test writing? We use descriptive statistics to characterise students' test writing practices in terms of how it is spread out over time (work sessions) and space (portions of the project).

RQ2: How do software project outcomes relate to the *balance* and *sequence* of effort devoted to writing test code and solution code? A virtually universally agreed upon tenet of effective software engineering is *incremental* or *test-oriented development* [67]. Where existing work presented evidence for or against TDD, we are primarily interested in finding concrete evidence for or against *incremental* test-oriented development, be it test-first or test-last. Additionally, test-first development, a key aspect of TDD, has frequently been extolled as a beneficial way to practice test-oriented development. However, experiments and case studies have found conflicting evidence regarding how exactly testing first affects software quality and project outcomes.

In §5.2 I describe the set of metrics we developed to help answer these research questions.

5.1.2 Study Context

Similar to the studies presented in Chapter 4, we studied the software development habits of students enrolled in the CS3 course at Virginia Tech. Students in this course have taken several prerequisite programming courses, most of them in Java. Subjects became acquainted with the JUnit testing framework in a previous course, and in the data structures course were taught material about project management and incrementally writing and testing code by one of the authors of the paper described in this chapter ([100]). We examined 157 students' programming habits as they work on four course projects over the course of the semester, for a total of 415 observations. Our design is unbalanced because not all students completed all 4 projects, typically because they withdrew from the course.[1]

Subjects were given about four weeks for each project. Each project asked the subjects to implement one or more data structures, along with its standard operations. Students were required to write JUnit tests for their code. In addition to correctness, students were graded

[1]A manifestation of the problem that this dissertation aims to address.

on the percentage of conditions covered by their test-suites, providing strong incentive to eventually write test suites with near-total code coverage.

5.1.3 Data Collection and Preprocessing

We collected and preprocessed data about the code-writing activities that students performed while working on their projects. We used DevEventTracker (see Chapter 3) for data collection. As described in Table 3.1, a Git repository is automatically created for each student's project. A **`commit`** is saved each time the student clicks the "save" button in the IDE. This rich data set captures the detail and nuance of test-writing behaviour that is necessary to assess the student's engagement with software testing over the project lifecycle. Repositories were mined using the open-source tool RepoDriller [6].[2]

Work sessions. Similar to the work in Chapter 4, we split the event stream for a subject working on a project into *work sessions*.

Capturing test code and solution code modifications. To determine when changes affected test code vs. solution code, and when test code changes were related to solution code changes, we extracted developer activity on a per-method basis, and call these events *method-modification events*.

Recall that project snapshots were captured as Git commits, which can be looked at as a sequence of software patches or snapshots. Using **`git diff`** output and an abstract syntax tree (AST) of the current patch, we determine which solution methods and which tests for such solution methods were modified in each commit. This was done by 1) using **`git checkout`** to restore the project to a given historical state; 2) comparing the line numbers in the patch's **`diff`** output with line number information provided in the AST for the given patch; and 3) determining which solution methods and test methods were modified in the given patch. A test for a solution method m is modified in a commit if the test directly invokes m. Therefore, we magnify the commit history into a series of method-specific events, meaning we expand commits into a series of method-specific `MODIFY_SELF` or `MODIFY_TEST_FOR_SELF` events.

Data filtering. We filtered out some solution methods from analysis. We excluded getters, setters, and printing methods. Exploratory analysis showed that 93% of projects in our dataset directly invoked at most 60% of all solution methods in test code.[3] To illustrate the challenge this poses, consider Figure 5.1 which shows the distribution of testing effort on all methods in a representative subject's implementation of the fourth project assigned during the semester. The horizontal shows the percentage of changes related to a method that were test changes. Notice that most solution-methods saw zero testing effort. Both the mean

[2]Now PyDriller [160]

[3]Note that this is not the same as code coverage, which includes constructs that are invoked further down the call stack.

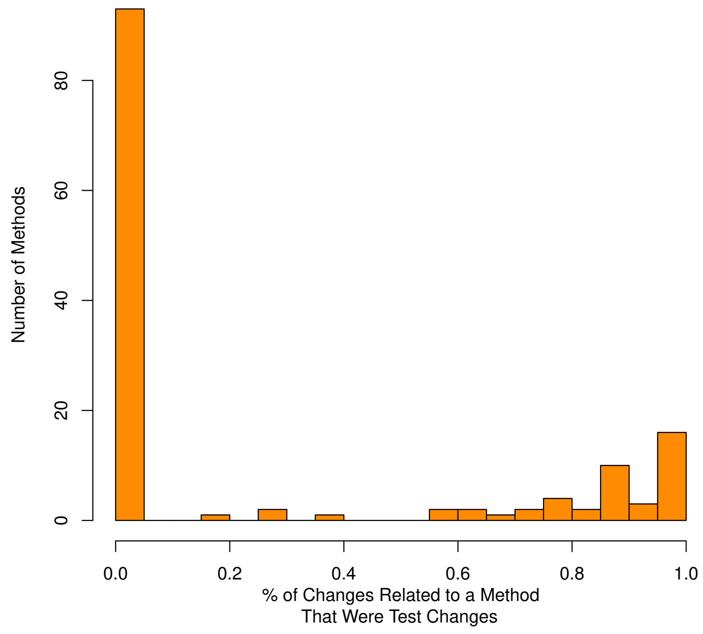


Figure 5.1: Example: Testing effort on individual methods in a representative project.

(0.27) and median (0) for this example are highly influenced by the skewed distribution, and would result in a misleading 'score' of method-specific testing effort. To account for this, we only examine the *top 60% most tested* solution-methods in each project, where methods were ordered by the testing effort they were given. By examining only these most tested methods, we are robust to the highly skewed distribution of testing effort across methods.

5.2 Proposed Metrics of Testing Effort

Consider Figures 5.2 and 5.3. Figure 5.2 shows an example sequence of developer activity, created from synthetic data. Each group of blocks represents a *work session*, and each individual block represents test code (shaded) or solution code (solid) written for a given method. Notice that the work done on individual methods or during individual work sessions cannot be clearly characterised as TDD or ITL. That is, current notions of TDD and ITL would not be able to characterise this example.

This non-conformance is common among practitioners [7] and students [34]. In a series of case studies, Zaidman et al. found that projects can have varying amounts of effort devoted to testing at different points in the lifecycle (for example, periods of heavy testing before a release, or non-contiguous periods of synchrony between test code and solution code) [173].

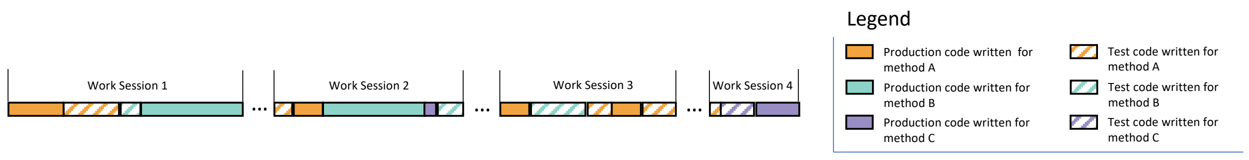

Work Session 1
Work Session 2
Work Session 3
Work Session 4
Legend
Production code written for method A
Production code written for method B
Production code written for method C
Test code written for method A
Test code written for method B
Test code written for method C

Figure 5.2: An example sequence of developer activity.

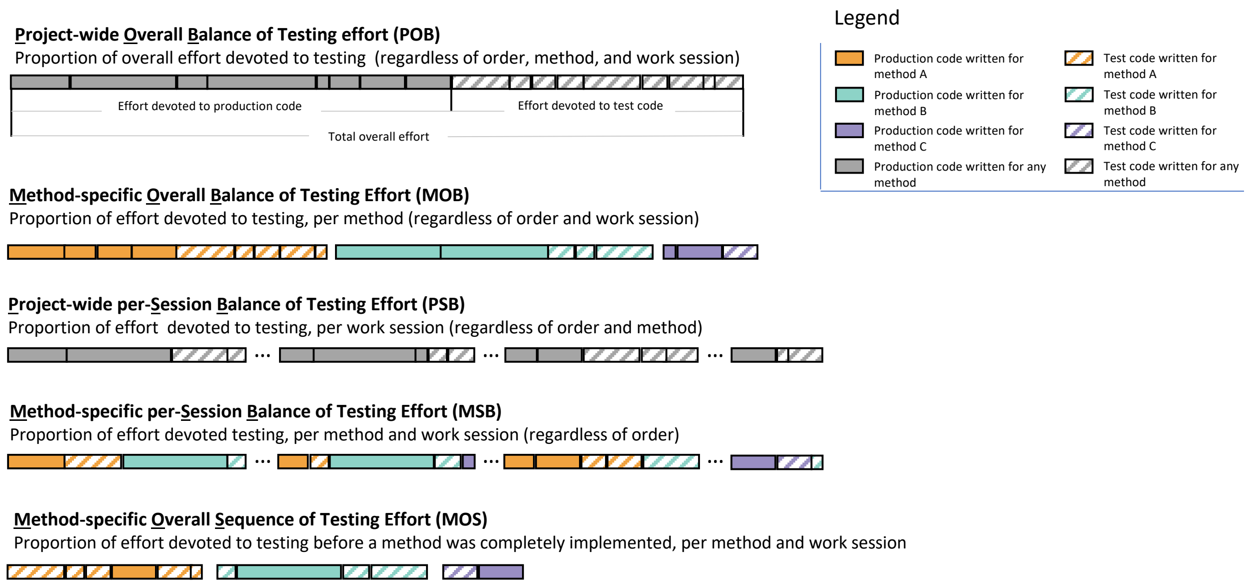

Project-wide Overall Balance of Testing effort (POB)
Proportion of overall effort devoted to testing (regardless of order, method, and work session)
Effort devoted to production code
Effort devoted to test code
Total overall effort
Method-specific Overall Balance of Testing Effort (MOB)
Proportion of effort devoted to testing, per method (regardless of order and work session)
Project-wide per-Session Balance of Testing Effort (PSB)
Proportion of effort devoted to testing, per work session (regardless of order and method)
Method-specific per-Session Balance of Testing Effort (MSB)
Proportion of effort devoted testing, per method and work session (regardless of order)
Method-specific Overall Sequence of Testing Effort (MOS)
Proportion of effort devoted to testing before a method was completely implemented, per method and work session
Legend
Production code written for method A
Production code written for method B
Production code written for method C
Production code written for any method
Test code written for method A
Test code written for method B
Test code written for method C
Test code written for any method

Figure 5.3: Measures to be derived from a programming activity event stream. Each row depicts a different method of aggregating the programming events from Figure 5.2.

This is seemingly at odds with the notion that software development behaviour can be cleanly bucketed into TDD or ITL.

We would like to avoid this dichotomy. In this section, we propose a new family of metrics to measure the balance and sequence of test and solution code writing effort in a continuous fashion. That is, we measure the extent to which students followed beneficial practices within a work increment—as opposed to measuring *whether or not* they followed them. As a result, we measure more faithfully the varying levels of student engagement with testing at different times in the project lifecycle. These conceptualisations are useful “in the wild”, where it is difficult to automatically infer adherence to TDD or other development methodologies without artificially controlling the programming environment (for example in a lab or workshop setting).

We define our metrics of testing effort according to the two factors with which practitioners seem most concerned with when recommending testing practices: the **balance of testing effort** (a common idea behind TDD and ITL [67]), and the **sequence of testing effort** (the main idea behind test-first development [14]). Figure 5.3 depicts the metrics we describe in the following sections, in terms of the example activity from Figure 5.2. Metrics are defined in terms of the **balance of testing effort** and the **sequence of testing effort**. Our raw data were Git histories of project snapshots captured using DevEventTracker (see §5.1.3).

5.2.1 Balance of Testing Effort

We examine project snapshots in the raw event stream in terms of *time* and *location* in the source code. That is, snapshots may be bucketed based on the *work session* in which they took place, or the *methods* they were related to.

Project-wide Overall Balance of Testing Effort (POB). This metric is depicted in the first row of Figure 5.3. It represents the test effort in the entire project regardless of the solution methods being tested or the session in which test code was written. That is, notice how the visual dimensions for colour (method), ordering, and block-group (sessions) have been eliminated for this metric.

"Effort" is measured as the number of line-level code-changes (additions, removals, or inline modifications) as captured by the **`git diff`** command (see §5.1.3). We define "**testing effort**" as the percentage of effort that was devoted to writing test code. By traversing the code changes for a project, we compute the total size of all changes to test code T and the total size of all changes to solution code S. Then we can calculate testing effort as a percentage of overall effort spent writing test code:

$$POB = \frac{T}{S + T} \tag{5.1}$$

Note that this is not equivalent to the percentage of source code that is test code. Additions, removals, and line-changes are included in this measure, as opposed to simple line counts in each snapshot or in the final product. Therefore, this is a measure of *testing effort*, rather than a measure of *amount of test code*.

Method-specific Overall Balance of Testing Effort (MOB). This metric is depicted in the second row of Figure 5.3. In this metric, we determine testing effort while taking *location* into account, i.e., we measure the testing effort devoted to individual methods. We use the method-modification stream described in §5.1.3, and apply Equation 5.1 to each method in the project to compute the testing effort devoted to each method. Let this set of method-level test effort values be POB_m. Then we calculate the MOB metric as the median of the distribution of testing effort measured for all methods (after filtering as described in §5.1.3).

$$MOB = \widetilde{POB_m} \tag{5.2}$$

Project-wide per-Session Balance of Testing Effort (PSB). This metric is depicted in the third row of Figure 5.3. To determine testing effort while taking *time* into account, we compute *project-wide per-session balance* (PSB) of solution code and test code. We grouped snapshots into work sessions as described in §4.5, and calculated the testing effort devoted to each work session using Equation 5.1.

Therefore, if POB_w is a set of values of testing effort devoted in each work session, this measure of testing effort over time (PSB) can be defined as the median testing effort across all work sessions:

$$PSB = \widetilde{POB}_w \tag{5.3}$$

Why use the median? Each project will certainly include *some* testing effort, because subjects were required to include test suites. Using the mean testing effort, there would be no way to gauge whether this testing effort is put in gracefully over time. That is, the paucity of testing effort early on would be counteracted by an increase in testing effort at the end of the project, or vice versa.

Method-specific per-Session Balance of Testing Effort (MSB). This metric is depicted in the fourth row of Figure 5.3. MSB determines testing effort while taking both *time* and *location* into account. That is, for the solution code that is written in a given work session, we would like to know if the student tends to write *related* test code. We compute *method-specific balance over time* (MSB) of solution code and test code. To do this, we use *method-modification events* and divide them into *work sessions* based on their timestamps (see §5.1.3).

Therefore, for all changes related to a given method that were made in a given work session, we may measure the testing effort using Equation 5.1. We are left with test effort values at two levels of grouping: **work session** and **method**. To compute MSB, we first find the project-wide per-session testing balance *for individual methods*, and call it PSB_m. Then we find the median per-session testing balance *across all methods*. Therefore, this metric can be represented as a "median of medians":

$$MSB = \widetilde{PSB}_m \tag{5.4}$$

5.2.2 Sequence of Testing Effort

Method-specific Overall Sequence of Testing Effort (MOS). Students did not practice test-first development. In our entire dataset, only one method was invoked in a test before it was defined in the solution. It is possible that students defined method stubs or increments of functionality before conducting relevant testing, but it is difficult or impossible to automatically infer this using static analysis and repository mining with any degree of certainty. However, what we can tell for certain is when work was completed for a given solution method, i.e., when it was modified for the last time. That is the basis for the *MOS* metric.

This metric is depicted in the last row of Figure 5.3. We measure *sequence* as the percentage of test code that was written for a solution method before the method was *finalised*. This gives an idea of the extent to which the student tended to write tests before completing the solution code under test. Therefore, if m is a solution method, then T_b is the total size of changes to test code before m was finalised, and T_a is the total size of changes to test code after m was finalised. Note than T_b and T_a only include changes to test methods that directly invoke m. Then we compute the percentage of test code for m that was written before m was finalised as:

$$MOS_m = \frac{T_b}{T_b + T_a}$$

We compute the median MOS_m over all methods to get an overall measure for a student working on a project (MOS):

$$MOS = \widetilde{MOS_m} \tag{5.5}$$

5.3 Research Question 1

RQ1: To what extent did students practise incremental test writing?

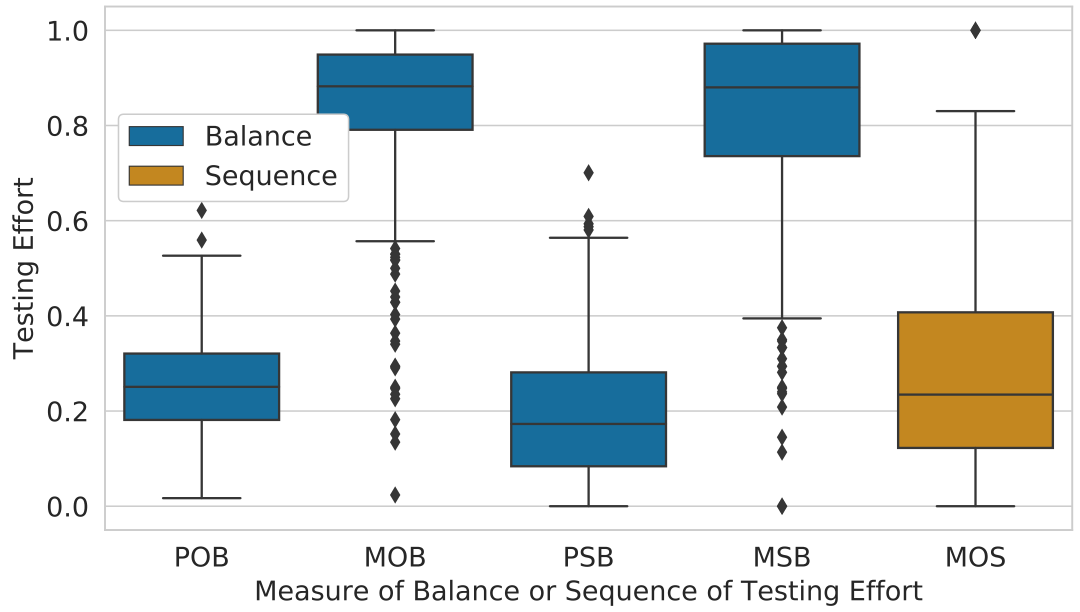


Figure 5.4: Distributions of testing effort metrics as defined in §5.2.

Figure 5.4 depicts distributions for all of the metrics described in §5.2. Since all the metrics are proportions and therefore on the same scale, they are plotted together. The blue boxplots represent measurements of the *balance* of test writing effort, and the orange boxplot represents the measurement of the *sequence* of test writing effort.

Balance of testing effort. The figure indicates that, over the course of the entire project, students tended to devote about 25% of their effort to writing test code (median $POB = 25.08\%$). The situation seems to be a bit worse when one considers the amount of testing effort applied in individual work sessions. The boxplot indicates that a majority of students devoted less than 20% of their effort toward writing tests in most of their work sessions (median $PSB = 17.29\%$). This could mean that students are not testing as much as they should. This would be in keeping with other work that finds that test code and solution code do not co-evolve gracefully for students or for professionals [18, 19].

The higher values for method-specific balance of testing effort indicate that most methods received a considerable amount of testing effort. This is evident across the entire project timeline (median $MOB = 88.24\%$) as well as in individual work sessions (median $MSB = 88.01\%$). These high proportions are perhaps not surprising when one recalls testing effort was considered devoted to a given method when that method was directly invoked in the test (see §5.1.3). Therefore, for a given unit of testing effort, the event stream possibly contained **`MODIFY_TEST_FOR_SELF`** events for several solution methods, driving up the proportion of "effort" devoted to testing specific methods (see §5.6).

Sequence of testing effort. In terms of the sequence of testing effort, the figure indicates that in a majority of project implementations (85%), students put in less than 50% of their testing effort before the relevant production code was finalised (median $MOS = 23.45\%$). This suggests that subjects in our study tended to put in more testing effort for a given method *after* it was finalised.

Whether the behaviours outlined above can be considered effective or ineffective is explored in the next research question.

5.4 Research Question 2

RQ2: How do software project outcomes relate to the balance *and* sequence *of effort devoted to writing test code and solution code?*

Once we characterised the balance of writing test code and solution code as a series of continuous independent variables, we used linear models to determine their effects on the following project outcomes:

- **Semantic Correctness (C)**: Measured as the percentage of tests passed from a suite of automated acceptance tests written by the course teaching staff.

- **Test Suite Coverage (*T*)**: Measured as the percentage of condition coverage achieved by the student's own test suite, measured using the JaCoCo[4] plugin. While subjects were required to submit test suites with nearly 100% code coverage, that might not happen until near the end of the development cycle.

We use linear mixed-effects ANCOVAs [12], with *students* and *projects* as crossed random effects. Experience suggests that no two students are the same, and this inherent variation might confound the model. Further, it is difficult to compare the programming process on different projects, even if they are assignments in the same course. Mixed-effects models allow us to control for the variation from these unaccounted-for effects.

Results presented herein use $\alpha = 0.05$ to determine significance.

We present *marginal* and *conditional* R^2 values [127] that describe the amount of variance in project outcomes explained by our models. Marginal R^2 values refer to the amount of variance in the outcome variable described by *fixed effects only* (in this case, our process measurements). Conditional R^2 values refer to the amount of variance in the outcome variable described by *the entire model* (in this case, the process measurements as well as individual students and assignments).

We fit the following linear mixed models for **semantic correctness** (C) and **test suite coverage** (T), using our metrics as fixed effects.[5]

$$C \sim POB + MOB + PSB + MSB + MOS + (1|student) + (1|project)$$
$$T \sim POB + MOB + PSB + MSB + MOS + (1|student) + (1|project)$$

Table 5.1: ANCOVA model summary for overall testing effort.

	C		**T**	
Measure	Estimate	p	Estimate	p
POB	0.30	$< 0.001*$	0.23	$< 0.001*$
MOB	*NA*	0.12	*NA*	0.41
PSB	*NA*	0.83	*NA*	0.97
MSB	*NA*	0.97	0.08	0.01*
MOS	*NA*	0.74	-0.06	0.03*
	Residual Std. Err. = 0.23 Marginal $R^2 = 0.05$ Conditional $R^2 = 0.39$		Residual Std. Err. = 0.11 Marginal $R^2 = 0.10$ Conditional $R^2 = 0.17$	

[4]https://www.eclemma.org/jacoco/
[5]The notation $(1|factor)$ means that *factor* was used as a random effect.

The models are summarised in Table 5.1. Estimates suggest that project-wide overall balance (POB) of test code and solution code is positively related with semantic correctness and test coverage. Additionally, test coverage has a positive relationship with method-specific per-session balance (and MSB) of test and solution code, and a significant but weak negative relationship with the amount of testing effort put in before finalizing solution code (MOS).

To gain a more fine-grained understanding of our measures and their effects, we fit another "process-based" model for each outcome, only including the time-based measures:

$$C \sim PSB + MSB + MOS + (1|student) + (1|project)$$
$$T \sim PSB + MSB + MOS + (1|student) + (1|project)$$

The models are summarised in Table 5.2. Notice that PSB is significantly related to both outcomes in the process-based model, but not in the overall model in Table 5.1.

Table 5.2: ANCOVA model summary for process-based testing effort (only including metrics that take time into account.)

	C		**T**	
Measure	Estimate	p	Estimate	p
PSB	0.30	0.005*	0.12	0.008*
MSB	0.11	0.10	0.09	0.002*
MOS	-0.03	0.62	-0.06	0.02*

Semantic correctness (C) and test coverage (T) were positively related to project-wide per-session balance of test and solution code (PSB). That is, project implementations tended to be more semantically correct, and their test-suites tended to have higher condition coverage, when students wrote more test code each time they sat down to work on the project. Test coverage was also positively related to the balance of testing effort devoted to individual methods in each work session (MSB), i.e., condition coverage tended to be higher when more testing effort was devoted to methods that were edited during the same work session.

The MOS metric measures the percentage of testing effort that tends to be put in before the relevant solution method has been finalised. Whether testing effort took place more predominantly before or after the relevant solution code was finalised was irrelevant to semantic correctness. On the other hand, test coverage had a negative relationship with MOS, suggesting that implementations tended to have stronger tests when a higher proportion of the testing effort devoted to a method was expended *after* the method was finalised.

5.5 Discussion

Why is semantic correctness not associated with writing test code for relevant solution code? Notice that method-specific per-session balance of test code and solution code (*MSB*) was not significantly related to semantic correctness (Table 5.2). Does this mean that it does not matter *what* test code a developer writes, only that they write *some* test code? It would be surprising and counter-intuitive to think so. However, it may be that, for this population, the variance in correctness explained by the specificity of test code written is subsumed by the variance explained by simply writing test code consistently.

Why is PSB significantly related to outcomes in the process-based model (Table 5.2), but not in the overall model (Table 5.1)? It may be that the variance explained by consistently writing software tests in each work session is subsumed by the variance in explained by maintaining an overall balance of testing effort.

How does one balance test code and solution code over time, while also writing more test code after the relevant solution methods have been finalised? In §5.4, we saw that a better balance of solution code and test code over time (*PSB*) was related to better project outcomes, and that writing a higher proportion of test code before the relevant solution is finalised led to *less thorough* test suites. These findings together suggest that while regular, balanced testing is important to test coverage, it is also important to put in testing effort after pieces of functionality have been completed.

Why are higher amounts of "test last code" related to test suites with higher code coverage scores? It is possible that students were able to better compose tests for existing ("finalised") functionality—that is, they may have found it easier to identify untested conditions or statements after the relevant code had already been written. It is also possible that students, being incentivised to maximise the code coverage achieved by their test suites, went about systematically increasing their code coverage scores *after* they had finalised the work for a project. If this happened, they would be driving up their code coverage with test code written *after* the relevant code had been finalised, contributing to the *MOS* metric's negative relationship with test suite thoroughness.

What do the low marginal R^2 values suggest? The process measurements show significant relationships with semantic correctness and test suite coverage. Marginal (fixed effects only) R^2 values [127] for both models suggest that this effect size is small (5% for C and 10% for T). There are likely to be numerous unexplained sources of variation when measuring human behaviour. In other words, any number of unaccounted-for factors could affect the quality of software produced by student developers. Those factors are not under study. The goal of this study was to determine the impact of balancing solution and test coding effort on project quality, and the models are able to answer our research questions with statistical confidence. It could also be that the assignments in this CS3 course do not "hit the right switches", i.e., success did not demand strict incremental testing. After all, they were not explicitly designed to do so. Future work should involve a similar study using semester-long projects from software engineering courses, more closely imitating real-world scenarios.

5.6 Threats to Validity

Internal. Since we do not have strictly defined experimental and control groups, we do not claim direct causality between process measurements and project outcomes. Our sample of student developers is sufficiently large and does not suffer from a selection bias.[6] Therefore, we do not believe that this is a serious threat to validity. Subjects had mechanical experience with the JUnit testing framework from a previous course. Multiple class periods were devoted to teaching material about project management skills, including incremental software testing, before the first project was assigned, possibly mitigating threats from differential experience.

External. Findings based on this particular population of students and assignments might not generalise to all junior level student programmers. Further, student behaviour is often motivated by a number of unknown external factors (for example, deadlines and responsibilities from other courses). It is unclear if or how this might have affected our findings, other than to observe that this semester seemed typical of our long experience with the course.

Construct. The largest threat to construct validity is related to the computation of the event stream described in §5.1.3. Specifically, we link test methods and solution methods if a given test method directly invokes a given solution method. However, it could be that the solution method was not the 'focal point' [77] of the test method, and was only being invoked to set up the test case, or to gain access to the method that was actually being tested. This could have increased the number of `MODIFY_TEST_FOR_SELF` events for some solution methods, affecting our process measurements. However, if a solution method is directly invoked in a test, it is reasonable to claim that the method is being tested, regardless of developer intent. Indeed, this is the basis for code coverage, which treats methods (or statements, branches, etc.) as "covered" upon invocation "somewhere in the call stack".

5.7 Summary

All in all, our findings support the conventional wisdom about the virtues of incremental testing. We can tell with some confidence that students do *not* consistently practice testing as they work on projects, and that this behaviour has the potential to negatively affect their project outcomes. We did not find support for the notion that writing tests first leads to improved project outcomes. The balance metrics described in §5.2.1— particularly *PSB*— put us in a position where we should be able to determine *during development* whether a student is engaging in these counter-productive behaviours.

[6]The course we study is a required part of the CS undergraduate curriculum, and we included all consenting students (>96%) enrolled in the course as subjects.

Chapter 6

Improving the Assessment of Software Test Quality

Software testing is an important aspect of software engineering, but students struggle with writing effective tests [60, 158]. Too often, students seem disinclined to practice software testing [35] and display poor testing abilities [60, 158]. As incorporating software testing into programming courses has become routine practice [5, 95, 158], educators have considered how best to evaluate student-written software tests [61, 78, 171]. Computer science departments are increasingly interested in providing incremental feedback about students' test suites using automated assessment tools (AATs) [136]. Examples of these systems [135, 158, 166] include Web-CAT [60] and ASSYST [88]. Of these, Web-CAT is the most widely used [136], serving over 120 institutions in the US. Feedback provided by these systems is usually expected to be immediate.

Numerous strategies have been proposed [1, 60, 78] for giving feedback on software test suites, but they are limited in terms of their strength as test adequacy criteria [1, 85], their ability to give incremental feedback on intermediate submissions [63, 78], or their high computational cost [1, 152]. We consider one such approach, *mutation analysis*, as a candidate for evaluating the quality of students' software tests. Mutation analysis is widely considered a strong test adequacy criterion whose use is limited largely by its computational cost. For our uses in particular, it may be untenable to enable automated mutation-based feedback in an AAT like Web-CAT. In this chapter, we evaluate various mutation analysis approaches for their suitability for deployment into an AAT. We propose our own approaches that appear to drastically reduce the cost of mutation analysis while preserving its effectiveness as a strong test adequacy criterion.

I begin with a general discussion of test adequacy criteria (§6.1.1), mutation analysis and its related literature (§6.1.2), and the use of test adequacy criteria in educational contexts (§6.1.3). Following this, I motivate and state my research questions (§6.2.1), context (§6.2.2) and results (§6.3–§6.6). I close with a discussion of results (§6.7) and threats to validity (§6.8), and conclude (§6.9).

The work presented in this chapter is based on an as-yet-unpublished manuscript. I am the main contributor and will be the first author on the resulting research paper.

6.1 Background

In the following section, I discuss various test adequacy criteria and their limitations, and how mutation analysis addresses these limitations (§6.1.1). I then discuss the primary challenge associated with mutation analysis—its computational cost—and existing efforts to mitigate it (§6.1.2.2).

6.1.1 Test Adequacy Criteria

A test adequacy criterion is a predicate that defines the program constructs that should be executed and the conditions that must be met for a given test or set of tests to be considered "adequate". Numerous forms of testing criteria are in use today. In this section, I discuss various methods of testing a program and their related adequacy criteria. For the purposes of this discussion, I partially conform to the categorisation from Zhu et al. [176], which describes *structural testing* and *fault-based testing* criteria. Zhu also defines *error-based testing*, which is somewhat limited to numerical software or software with limited input domains. Further, there is a distinction between test adequacy based on a *program* and based on a *specification*; this discussion focuses on the testing of a program.

6.1.1.1 Structural Testing

Structural testing aims to satisfy the requirement that certain elements or constructs of a program or specification have been exercised. A program can typically undergo structural testing using two types of test-adequacy criteria (independently or in concert with one another): *control-flow* or *data-flow* adequacy criteria.

Control-flow adequacy criteria. First, a control-flow graph (CFG) for a program is computed. A CFG is a directed graph in which each node is a linear sequence of computations [176] (or a *basic block*, i.e., "a maximal sequence of simple statements with one entry point such that if the first statement is executed, all statements in the block will be executed" [130]). Each directed edge represents a transfer of control from one basic block to another. Possible execution paths through the program can be computed by starting at the CFG's entry point and following edges until a termination.

The CFG forms the basis of *code-coverage* test adequacy criteria [126]. *Statement coverage* is the most basic form of code coverage, and requires that all statements in the program (or basic blocks in the graph) be executed by the test suite at least once. This is known to be a weak test adequacy criterion.

Decision coverage is a stronger control-flow adequacy criterion that requires that each edge in the CFG be executed at least once. For the program, this translates to the requirement that all decisions (one or more conditions joined by `AND` or `OR` operators) be made to evaluate to both **`true`** and **`false`** at least once. This can be strengthened further by taking into account the (possibly multiple) conditions that make up decisions in the programs. *Condition coverage* requires that all conditions in all decisions evaluate to **`true`** and **`false`** at least once. *Decision / condition coverage (DCC)* requires that both condition coverage and decision coverage are satisfied. *Modified condition / decision coverage (MCDC)* requires that DCC is satisfied and that every condition has been shown to affect its decision's outcome. MCDC is the criterion used by the FAA to ensure test adequacy for safety critical software in aircraft [72]. *Multiple condition coverage* requires that the entire truth table for all decisions be satisfied. This is typically not a practically feasible option due to the high redundancy between execution paths that is likely to occur.

Data-flow adequacy criteria. In data-flow adequacy [73], information about the flow of data is introduced into the CFG for a program. That is, nodes are characterised in terms of the variables that are defined and used in them. The *definition* of a variable occurs when a value is assigned to it. The *use* of a variable occurs when an already-defined variable's value is used during execution. Uses can be *computation uses*—those that affect a computation or output—or they can be *predicate uses*—those that affect control flow of the program. Incorporating this into the CFG, computation uses exist on the nodes of the CFG, and predicate uses exist on the edges. A *def-use pair* for some variable x is a pair of nodes in which the first contains a definition of x, and second contains a use of x.

Data-flow adequacy is satisfied when all def-use pairs for all variables are executed by the tests. Like control-flow adequacy, data-flow adequacy can be measured in different ways, i.e., by considering only computation uses, predicate uses, or both.

6.1.1.2 Fault-Based Testing

In fault-based testing, test suites are evaluated on their ability to detect defects that manifest in program state or output. This is typically done by seeding artificial errors into otherwise "correct" programs and evaluating the test suite's ability to detect them. Fundamentally, this is advantageous over structural testing methods since they are satisfied only by *exercising* specific constructs, whereas fault-based testing is sensitive to the propagation of faulty program state or output to the tests. In terms of automated tests, fault-based criteria are sensitive to the *assertions* in the tests.

All-pairs execution. In an educational setting, we can take advantage of the fact that we have access to multiple "real" implementations for the same program specification (i.e., assignments completed by multiple students). The all-pairs approach, proposed by Goldwasser as a "fun way to incorporate aspects of software testing" in the classroom [78], involves running each student's test cases against every other student's implementation. The adequacy score of the student's test cases is determined by measuring the percentage of known faults (in other students' programs) the test cases are able to uncover. The approach has been implemented in multiple contexts and flavours [27, 36, 61, 171], and evaluations have been found it to be effective at revealing deficiencies test suites. It affords a diverse set of "real" edge cases available in tests and solutions from classrooms-full of students.

Mutation analysis. In mutation analysis, artificial defects are injected into a program, creating faulty variants of the original. Test adequacy is measured by the percentage of faulty variants that are detected by the test suite, i.e., by a failing test. Ignoring the fact that mutation analysis was first developed many years before Goldwasser's experiment, one can look at it as an automated and autonomous way of implementing all-pairs execution.

In the following section, I describe mutation analysis in some detail and discuss its strengths and its primary weakness—its computational cost.

6.1.2 Mutation Analysis—A Silver Bullet?

Proposed in the 1970s by DeMillo et al. [51], mutation analysis is a fault-based test adequacy criterion in which small defects (*mutations*) are injected into a program, creating faulty versions of the original—called *mutants*. Mutations are typically small syntactic changes. The types of mutations that can be made are called *mutation operators*. While the number of possible mutation operators is essentially infinite in principle, mutation operators have more or less been designed based on years of knowledge of and experience with human-made errors in programming [102]. If a mutant is detected by the test suite, i.e., by a test that fails after introducing the mutant, it is said to be *killed*. Likewise, if a mutant is not killed, it is said to have *survived*. The test suite's *mutation adequacy score* can then be measured as the percentage of mutants that were killed. Mutation analysis can be used to assess the adequacy of a test suite, or to guide a tester such that they know when a program is "tested enough".

Mutation analysis is the strongest test adequacy criterion currently available. Offutt showed that it subsumes the structural testing criteria described in §6.1.1.1. That is, a test set that satisfies the mutation adequacy criterion is theoretically proven to also satisfy strong condition coverage and data-flow testing criteria [130]. Separately, Wong & Mathur showed that mutation analysis subsumes data-flow testing criteria [169]. Mutation analysis is also *flexible*, i.e., it can be and has been applied in a variety of programming languages (e.g., Java [45, 96, 117, 147], C [50], C# [55], JavaScript [125], and FORTRAN [102]), paradigms (e.g., object-oriented [172], functional [109]), and can be used for other testable artefacts like design specifications.

6.1.2.1 Underlying Assumptions

The validity of mutation analysis as a test-adequacy criterion relies on two fundamental assumptions: *the competent programmer hypothesis* and *the coupling effect* [33]. The soundness of these assumptions is not self-evident, and researchers have theoretically and empirically evaluated their veracity.

The Competent Programmer Hypothesis. Proposed by DeMillo et al., this hypothesis states that developers tend to produce programs that are *mostly correct*, i.e., their programs require a few syntactic changes to reach correctness [51]. Therefore, if we generate mutants by making small syntactic changes to the program, we are producing faulty versions that are close to those that are likely to occur "naturally". This hypothesis has been tested with varying levels of rigour (e.g., [53][1], [4, 80, 97]). Andrews et al. found that the ease with which real faults and hand-seeded faults were detected were approximately similar [4]. Gopinath et al. note that since this study is based only on eight C programs, and since the conclusion was drawn from a single program, the findings are not quite convincing [80]. In their larger study involving projects in C ($n = 1850$), Java ($n = 1128$), Python ($n = 1000$), and Haskell ($n = 1393$), Gopinath et al. measured the edit distance from "real" faults and the patches that fixed them. They found that real faults had a mean token distance of three to four, i.e., it would take 3–4 character replacements to turn a fault into a correct program or vice-versa. Papadakis et al. interpret this to be partial confirmation of the Competent Programmer Hypothesis [134]. However, Gopinath stresses that, since the vast majority of mutation operators make single-edit mutations, their findings increase the dependence of mutation analysis on the Coupling Effect.

The Coupling Effect. Also proposed by DeMillo et al., the coupling effect states that "test data that distinguishes all programs from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [51]. Offutt further distinguished between the "Coupling Effect" and the "Mutation Coupling Effect" [129]. Say that simple faults (or mutants) are made by making a single syntactical change, and complex faults (or mutants) are created by making more than one change. Then the Coupling Effect states that "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of complex faults". And the Mutation Coupling Effect says "complex *mutants* are coupled to simple *mutants* in such a way that a test data set that detects all simple *mutants* in a program will detect a high percentage of complex *mutants*". If this holds true, then Gopinath's fears regarding the prevalence of single-character mutation operators can be alleviated. Theoretically and empirically, the Coupling Effect and the Mutation Coupling Effect have been found to be true. Offutt found that test data designed to kill all first order (single-edit) mutants killed over 99% of second and third order mutants [129].

Other work investigating these two underlying assumptions is summarised in the survey of mutation testing research between 1970 and 2008 from Jia & Harman [93].

[1]Technical report, no evidence of peer review.

6.1.2.2 Efforts to Reduce the Cost of Mutation Analysis

While the exact number of mutants generated is dependent on the mutation operators used, it is also proportional to elements of program size. In 1979, Acree et al. claimed that the number of mutants is in $O(Lines \times Refs)$, where *Lines* is the number of lines of code, and *Refs* is the number of data references, going on to say that this is $O(Lines \times Lines)$ for most programs. In 1980, Budd claimed that it is in $O(Vals \times Refs)$, where *Vals* is the number of data objects. Offutt et al. tested these claims statistically, finding that *Lines* was only a modest predictor of the number of mutants, but that *Vals* and *Refs* were strong predictors.

The computational cost of mutation analysis comes from the fact that it requires the execution of a test suite potentially hundreds or thousands of times, i.e., for each generated mutant. Exacerbating this situation is the likelihood of generating *trivial mutants*, *equivalent mutants* [132], and *subsumed mutants* [133], which can artificially drive up or down a test suite's mutation score, while simultaneously increasing cost:

- *Trivial mutants* are trivially easy to kill, and are nearly always killed by a test suite.
- *Equivalent mutants* [132] are functionally equivalent to the source code, and therefore impossible to kill. These mutants tend to artificially drive down a test suite's mutation adequacy score.
- *Subsumed mutants* [133] are those mutants that are killed by a superset of the tests that kill some other mutant. That is, when the *subsuming* mutant is killed, the *subsumed* mutant is also killed.
- *Duplicate mutants* are functionally equivalent to other killable mutants.

Considerable effort has been devoted to reducing the cost of mutation testing (e.g., [49, 51, 54, 107, 131, 132, 155, 175]). Jia & Harman's survey [93] clusters these efforts into two categories:

- *Do fewer approaches*, those that reduce the number of generated mutants, and
- *Do faster approaches*, those that reduce the execution cost using other system or compiler level optimisations, e.g., by mutating byte code in memory instead of source code on disk [45]

In this chapter, we are concerned with mutant reduction techniques. Specifically, we investigate the practicality of *selective mutation* approaches [121, 131], i.e., approaches that generate a subset of all possible mutation operators that significantly reduce cost without incurring a significant loss in effectiveness.

In a seminal example of selective mutation, Offutt et al. experimentally determined a *sufficient* set of operators [131]. They found that of the 22 operators originally used in Mothra [52, 102], one of the first mutation analysis systems, a subset of just 5 key operators performed nearly as well as the complete set (99.5% mutation adequacy), with considerable cost savings (77.56% fewer mutants generated). Namin et al. [155] later used a statistical procedure to extract a subset of 28 sufficient operators out of the 108 operators available in a newer alternative system called Proteum [50].

Reducing the number of mutants further, Untch [162] proposed producing mutants simply by systematically deleting statements from the target program, instead of applying a (full or selective) set of several mutation operators. This initial evaluation of statement deletion (SDL) yielded positive results, prompting Offutt et al. to carry out an empirical evaluation of SDL [54] and other deletion operators [49]. SDL eliminated 81% of mutants that would otherwise have been generated using Offutt's sufficient set of 5 operators, with only a small but acceptable loss in accuracy. At the same time, SDL was found to be much less likely to produce equivalent mutants (mutants that are functionally the same as the original program, and thus impossible to kill) or duplicate mutants (mutants that are functionally the same as other mutants). These studies provide convincing evidence that mutation by deletion offers a possible path toward practical and scalable mutation testing.

In an educational context, Shams & Edwards [152] identified and partially solved key problems associated with introducing mutation analysis into a CS2 course. Selective mutation was identified as a practical choice for applying mutation testing in the classroom. They compared the effectiveness of various selective mutation approaches with other measures of test quality [61]. Of the mutator subsets evaluated, mutation by deletion was identified as the most effective in terms of cost and effectiveness as a test adequacy criterion.

6.1.3 Test Adequacy Criteria in Education and Their Limitations

Many strategies have been proposed [1, 60, 78] for giving feedback on software test suites, but they have multiple limitations [1, 61, 63]. There are primarily three approaches for giving students feedback about the quality of their software tests: code coverage, all-pairs execution, and mutation analysis (see Table 6.1).

Code coverage. Code coverage measures are the most widely used test adequacy criteria, in both education [60, 88, 159] and industry [87, 138]. They are inexpensive to compute and easy to reason about, and so are a popular choice in AATs that aim to provide students with rapid feedback. However, code coverage measures have been shown to be weak test adequacy criteria [1, 61, 85]. A key reason is that criterion satisfaction is not sensitive to the propagation of output or program state to the test output (i.e., through the use of assertions); they are satisfied as long as code is executed ("covered") by the test suite. Inozemtseva et al. showed that code coverage does not have a high correlation with test suite effectiveness after test suite size has been controlled for [85]. Elbaum et al. studied the impact of software evolution on code coverage [64] They found that small changes to the code base lead to relatively large deteriorations in code coverage, and that consequently code coverage is not likely to stay stable throughout a project's evolution. In the classroom, code coverage has been shown to be an easily "game-able" measurement, in the sense that students can directly manipulate it without substantively changing the quality of their tests [1]. Indeed, including coverage measures in project grading encourages students to maximise their adequacy of their test suites *as measured by the criterion*, rather than focusing on the actual quality of their test suites (i.e., its defect detection capability). The result is that code coverage distributions tend to cluster around the 100% mark [1, 151], regardless of the actual quality of students' test cases.

All-pairs execution. Edwards observed that a naive implementation of all-pairs approaches in the classroom would either require students to exclusively practice a form of black-box testing, or it would depend on students' solutions and tests compiling together. This assumption cannot be relied upon when students are given the freedom to create alternative designs, as is the case in most upper-level CS courses. The all-pairs approach also depends on all students having completed their projects and tests, precluding the possibility of feedback for intermediate submissions. Additionally, others have considered measuring the positive verification ability of a test, i.e., its ability to accurately identify correct solutions. Wrenn et al. [171] describe the problem of *over-zealous tests*, which make assertions about implementation details that are not part of the requirement specification. In all-pairs execution as described above, it is important to eliminate such tests from the corpus of available test cases, or to help students produce correct test cases as they work toward project completion [27, 170].

Buffardi highlights the importance of *unit test accuracy*, which considers both defect detection and positive verification ability when assessing student-written test suites [36, 37]. However, measuring unit test accuracy requires the use of a known-good reference solution against which student-written tests must compile. Like other all-pairs methods, these assessments cannot easily be applied in assignments where students are free to design their own solutions.

Mutation analysis. Mutation analysis mitigates the limitations of both code coverage and all-pairs approaches. As a test adequacy criterion, it subsumes strong forms of structural testing criteria described in §6.1.1.1 ([130]). It also does not require students to have complete solutions before they can receive feedback, mitigating the limitations associated with all-pairs methods. Aaltonen et al. compared code coverage and mutation testing in terms of their ability to effectively evaluate student-written tests [1]. Students found it easier to fool code coverage tools, and more difficult to achieve a high mutation score. However, this work only examined the accuracy of the scores, without exploring the cost of deployment or the feasibility of using the approach for providing incremental feedback to students.

Mutation analysis is well known for its high run-time cost (e.g., Section 3 in Jia & Harman's survey [93]). The process may involve creating potentially hundreds of mutants from the original program and then running the test suite against all of them. Cost is a significant concern in an educational context, since it reduces the opportunity to provide incremental feedback to students as they work, say, using an AAT like Web-CAT. In this chapter, we explore approaches to reduce this cost using selective mutation.

Table 6.1: Test evaluation techniques used in CS education and their strengths and weaknesses. $\star$ = Addressed in this chapter.

Technique	Strong test adequacy criterion?	Supports incremental feedback?	Fast response?
Code coverage [60, 158]	✗	✓	✓
All-pairs execution [37, 63, 78, 171]	✓	✗	✗
Mutation analysis [1, 152]	✓	✓	✗*

6.2 Research Method

6.2.1 Research Questions

We performed an empirical study to provide educators with an efficient, scalable mechanism to evaluate student test suites using mutation analysis. We study the following research questions:

RQ1: How efficient is mutation analysis at providing automated feedback on test suites? We study whether it is necessary to improve the efficiency of mutation analysis at all for student code. It may be that the smaller size of these projects allows mutation analysis to scale "as is". We evaluate the efficiency of using mutation analysis for automated feedback, in terms of the time taken for individual submissions to generate mutation results. We interpret results in terms of running time under typical and peak submission loads faced by the AAT server at our institution, which runs Web-CAT [60].

RQ2: Is mutation by deletion [49, 54] a cost-effective alternative to comprehensive mutation for educational software projects of varying complexities? Shams' evaluation [151] found statement deletion to be a cost-effective approach for evaluating the test suites produced by novice CS students. We believe this result is promising, so we conduct an evaluation of deletion operators set on a more general corpus of student-produced codebases.

RQ3: Can the cost of mutation by deletion be reduced further? Even though mutation by deletion represents notable runtime savings over comprehensive mutation, it may be possible to reduce this cost further. We evaluate whether cost-effective subsets of the deletion set of operators can perform comparably well at evaluating the thoroughness of a test suite.

RQ4: Are the benefits of different mutation strategies project dependent? Our analyses were conducted on a diverse set of programs, based on size and complexity (and therefore in terms of the mutants produced). We investigate whether our chosen selective mutation strategies vary in terms of cost-effectiveness based on the size of the projects under test. This would allow educators to make a more informed choice of operator subset to use for test suite evaluation.

Table 6.2: Programming tasks undertaken by students in our sample, and descriptions of their implementations. *# Mutants* indicates the number of mutants generated under the `FULL` set. Projects 1–4 are CS2 projects, and 5–7 are CS3 projects.

#	**Description**	**n**	**LoC**		**Cyc. Comp.**		**# Mutants**	
	(data structures implemented)		μ	σ	μ	σ	μ	σ
1	Bag	350	139.20	13.84	26.87	1.61	470.16	32.60
2	Linked stack	321	204.29	22.85	38.50	2.94	421.44	38.05
3	Array-based queue	259	448.00	39.71	104.00	7.58	1651.54	126.83
4	Linked list	89	718.02	221.53	147.38	53.78	2988.94	1491.91
5	Hash table, doubly-linked list	128	724.26	142.33	152.38	32.40	3377.76	776.02
6	Hash table, sparse matrix	133	946.56	178.69	202.17	38.77	3244.17	785.65
7	Bintree, skip-list	109	1263.18	303.22	261.96	73.89	6095.79	1952.25
	Total	1389	650,515		136,763		2,521,871	

6.2.2 Study Context

We studied Java projects developed by students enrolled in the CS2 and CS3 courses at Virginia Tech. These students have taken several prerequisite programming courses, most of them in Java, and are acquainted with the JUnit testing framework and writing their own unit tests. Students were required to write unit tests for their projects, and part of their grade depended on the thoroughness of their test suites (as measured by code coverage criteria).

The **CS2 corpus** included submissions to four programming assignments in which students were asked to implement and test a simple data structure, including: 1) an array-based bag; 2) an array-based stack; 3) an array-based queue; and 4) a linked list. Students in this course produced 1019 distinct submissions. In total, these submissions comprised 294,230 source lines of code (SLoC) and led to the generation of 993,602 mutants.

The **CS3 corpus** included submissions to three large and complex programming assignments. Students were given four weeks to work on each assignment. Each project asked the students to implement one or more data structures, including: 1) a memory management package with a hash table; 2) a sparse matrix; and 3) a Bintree, which is a kind of binary tree designed for storing and querying spatial data. Students in the CS3 course produced 370 distinct submissions. These submissions together made up 356,285 SLoC, and led to the generation of 1,528,269 mutants.

Together these comprise 1389 submissions to seven programming assignments of increasing size (Figure 6.1). Descriptions of the assignments and their submissions are presented in Table 6.2.

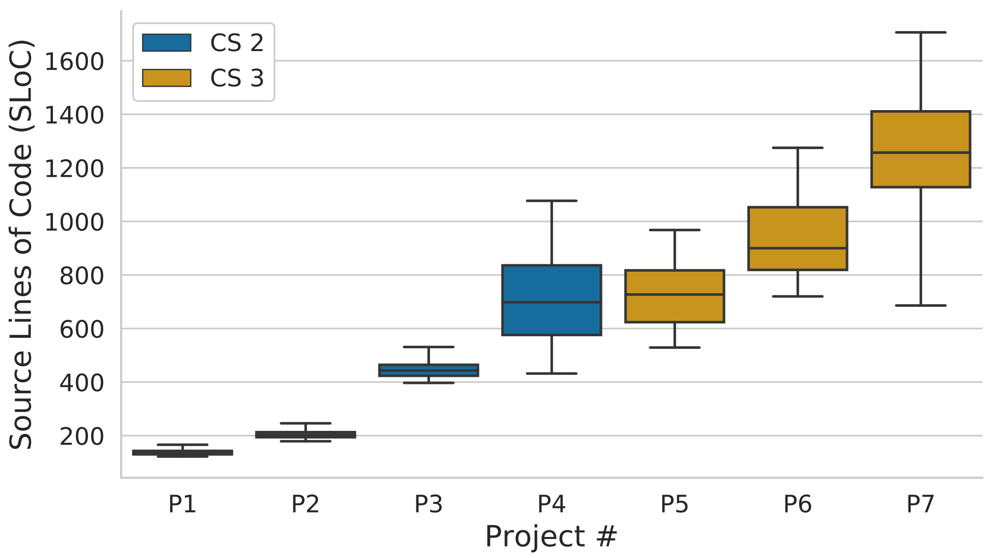


Figure 6.1: Our corpus contains submissions to assignments of increasing sizes (source lines of code). Whiskers indicate the 5th and 95th percentiles.

Language and tooling. We focus on testing Java programs, since Java is widely used in introductory and advanced programming courses at the secondary and post-secondary levels. We used **PIT** [45], the state-of-the-art mutation testing system for the JVM, to conduct mutation analysis. The Java language has attracted the development of a mature collection of mutation testing tools [45, 96, 117, 147]. Of these, μJava [117] has received the most attention in the software testing literature [48, 93, 103]. PIT [45] and MAJOR [96] are newer tools, still under active development. PIT was found to be the most robust and easy-to-use mutation tool for Java programs [48]. It is designed to be highly scalable, which makes it suitable for practical mutation testing to drive incremental feedback. It has seen much academic scrutiny in recent years [50, 103, 104, 108, 175], and the current version has undergone empirically motivated improvements. Laurent et al. empirically compared PIT with two well-known mutation analysis systems for Java—μJava and MAJOR—in terms of their defect-detecting capabilities [104, 108], using all possible mutation operators for each system. PIT was extended with additional operators based in research on mutation testing [45] (added in v14.4.4). Subsequent evaluation established that, as of this writing, mutation adequacy under the full set of PIT operators is the most stringent mutation-based measure of test suite thoroughness for Java programs [104, 108].

Building on the past performance of deletion operators (§6.1.2.2) in other languages and tools, and on PIT's current dominance of the Java mutation testing space in terms of effectiveness [104] and performance [61], we have analysed selective mutation using PIT with the goal of reducing the cost of mutation testing while maintaining performance on par with the comprehensive set of PIT operators.

Table 6.3: Selective mutation approaches evaluated for use in an AAT in this chapter, including the incremental subsets evaluated in §6.6. The *Ref.* column refers to the first proposal of the specified subset. ⋆ = Proposed in this chapter.

Approach	**PIT Operators**	**Ref.**	**Practical for an AAT?**
`FULL`	See Table 2 in [108]	[52]	RQ1
`SUFFICIENT`	*AOR*, *ROR*, *ABS*, *UOI*	[131]	RQ2
`DELETION`	*RemoveConditionals*, *AOD*, *NonVoidMethodCalls*, *VoidMethodCalls*, *MemberVariable*, *ConstructorCalls*	[49]	RQ2, RQ3
2-op Subset	*RemoveConditionals*, *AOD*	⋆	RQ4
1-op Subset	*RemoveConditionals*	⋆	RQ4

6.3 Research Question 1

RQ1: How efficient is mutation analysis at providing automated feedback on test suites?

AATs tend to handle substantial throughput, particularly when they provide students with intermediate feedback on incremental submissions. This typically results in many dozens of students making submissions in close temporal proximity to each other. It is imperative that any mutation analysis strategy used in an AAT supports a reasonable response time so that students can make appropriate use of intermediate feedback. To provide context for this study, I provide concrete running times for mutation analysis on our AAT server, which runs Web-CAT. During peak submission times, the Web-CAT server at Virginia Tech receives 39 submissions per minute, or a submission every 1.5 seconds. The sustained throughput is 652 submissions per hour, or a submission about every 5.5 seconds. With these numbers in mind, we would ideally like to produce feedback from mutation analysis in under 5 or 10 seconds.[2] It is worth noting that: 1) Virginia Tech researchers created Web-CAT and deploy its updates; 2) the Virginia Tech Web-CAT server serves 20–30 institutions in the US; and

[2]Of course, it is possible but expensive to scale horizontally and reduce running time. However, selective mutation helps to reduce the *asymptotic* cost of mutation analysis.

3) 100 or so other institutions with their own installations of Web-CAT would presumably adopt any improvements we made as a result of this research. As such, our motivations are driven by more than just the needs of Virginia Tech educators.

In this section we evaluate the efficiency of applying a comprehensive set of mutations to our corpus of target programs, in terms of the time taken to generate feedback on their tests.

6.3.1 Method

We define the `FULL` set of PIT operators to be all those used in the comparison of PIT with μJava and MAJOR by Laurent et al. [108] (see Table 2 in the reference), with some improvements. We omitted operators that would, by definition, duplicate mutants created by other operators. The *ROR* operator, which was added to PIT by Laurent et al., replaces occurrences of comparison operators with all other comparison operators. For example, the $<$ operator would be systematically replaced by <=, >, >=, ==, and !=, for a total of 5 mutants. On the other hand, the *ConditionalsBoundary* operator would only replace it with <=, and *NegateConditionals* would only replace it with its negation (>=). Clearly, the mutants produced by these operators are duplicates of those created by *ROR*. We omitted the *PrimitiveReturns* and *FalseReturns* operators, which duplicate mutants that are produced by *NonVoidMethodCalls*. We also omitted the *InvertNegatives* operator, which duplicates mutants that are produced by the *ABS* operator. Finally, we omitted a subclass of the *AOR* operator (*AOR1*, according to PIT's nomenclature), which would duplicate mutants produced by the *Math* operator. The `FULL` set of operators as described here is currently the strongest set of mutation operators available for Java programs [103, 108].

Mutation analysis was run on 1389 submissions, and results and running times were collected for each submission. Analysis was run on a machine with two 16-core 2.60 GHz Intel Xeon Gold 6142 CPUs and 256GB RAM running CentOS 7. This machine specification is similar to what is used to run our Web-CAT server.

6.3.2 Result

Mutation analysis using the `FULL` set is not efficient enough for incremental feedback. Mutation analysis took a median of 0.51 minutes to run per submission in the CS2 course, and 5.04 minutes per submission in the CS3 course. Feedback times that are this slow might result in intolerable slowdowns on the Web-CAT server that would be experienced by a sizeable community of users. Effects would be exacerbated during peak submissions times, when the server handles up to 40 submissions a minute. Additionally, consider the fact that students receive feedback about correctness in 5–10 seconds. To a student that already displays a disinclination to practice regular testing [95], testing feedback that is this much slower may not motivate or help the student to actively identify weaknesses in their test suites as they work on projects.

6.4 Research Question 2

RQ2: Is mutation by deletion a cost-effective alternative to comprehensive mutation for educational software projects of varying complexities?

Having found that the `FULL` set of PIT operators is not efficient enough for incremental feedback in AATs, in this section we evaluate the cost-effectiveness of mutation by deletion, i.e., mutating the program by systematically removing statements or constructs. Statement deletion been found to be an efficient mutation analysis approach in other educational contexts [151]. However, the efficiency of the deletion set has so far not been evaluated for the common educational context of automated assessment systems, and its effectiveness has only been evaluated on codebases produced by novice CS students. We compare the performance of deletion operators with that of the `FULL` set of operators (§6.3) and a key operator subset from the literature, the `SUFFICIENT` set.

6.4.1 Method

Mutation results for each operator can be determined from output generated by PIT. A single PIT run emits rich data for each generated mutant, including the type of operator used and its survival status. This allows information—like mutation coverage and number of mutants generated—to be determined for arbitrary subsets after a single run using all PIT operators.

We define the `DELETION` set to be a subset of PIT operators that approximates the statement deletion (SDL) and operator deletion (ODL) mutation operators for Java [49, 54]. Since PIT operates on Java bytecode, a precise replication of those deletion operators is not practical. We use 6 operators as our `DELETION` set (Table 6.3).[3]

We define the `SUFFICIENT` set of PIT operators as a research-based subset intended to produce significantly fewer mutants while maintaining effectiveness. In 2017, Laurent et al. [108] extended PIT with operators from the literature, including most of Offutt's experimentally determined sufficient set [131].[4] The `SUFFICIENT` set of mutation operators in PIT is Table 6.3.

We define two dependent variables:

[3]We did not use the OBBN mutation operator—variants of which mutate by deleting bitwise operators and operands—because only 5.7% of submissions contained bitwise operations at all.

[4]The Logical Connector Replacement (*LCR*) operator does not exist in PIT. The && and || logical connectors do not translate to single bytecode instructions, but instead to branching instructions which are mutated by *ROR*.

- **Cost**: The number of mutants per thousand lines of code (KSLoC)
- **Accuracy**: A measure of deviation of a subset's coverage from coverage that would be achieved using the FULL set, defined as follows:

$$\begin{aligned}\text{Accuracy} &= 1 - \text{squared err.}\\ &= 1 - (\texttt{FULL}\ \text{cov.} - \text{Subset cov.})^2\end{aligned}$$

Accuracy is measured against the FULL set because it is the strongest form of mutation testing currently available for Java projects (§6.2.2).

We measured the cost and accuracy for the FULL, SUFFICIENT, and DELETION subsets of operators, for our corpus of 1389 submissions. We used an ANOVA followed by posthoc pairwise comparisons to determine the differences between the cost and accuracy of individual subsets. Results are summarised in Table 6.4 and Figure 6.2.

6.4.2 Result

Mutation by DELETION is a cost-effective alternative for smaller codebases, but its running time still presents challenges for more larger codebases. A one-way ANOVA indicated a statistically significant difference in *cost* measured in mutants per KSLoC (F(2, 4164)=5110.50, $p < 0.01$) between the DELETION ($\mu = 678.09$), SUFFICIENT ($\mu = 1594.68$) and FULL ($\mu = 3430.56$) operator sets. Similarly, a one-way ANOVA also indicated a statistically significant difference in *accuracy* (F(2, 4164)=630.46, $p < 0.001$) between the DELETION ($\mu = 0.995$), SUFFICIENT ($\mu = 0.997$), and FULL ($\mu = 1.00$) sets. Posthoc analysis using Tukey's HSD test showed statistically significant pairwise differences between the subset groups in terms of both cost and accuracy ($p < 0.01$ for all pairs), and indicated that both cost and accuracy decreased in the order FULL $\rightarrow$ SUFFICIENT $\rightarrow$ DELETION. Effect sizes calculated using Cohen's d [44] showed that cost decreased much more precipitously than accuracy (Table 6.4).

This is visually apparent in Figure 6.2, which shows the accuracy of coverage and the cost of using the DELETION, SUFFICIENT, and FULL subsets. Note that there is no distribution for the accuracy of the FULL set, only a line, since accuracy was measured against the FULL set.

Although accuracy seems to be slightly lower for DELETION operators, observe that the accuracy axis is *lower-bound* at 0.94. This suggests that, though weaker than the FULL or SUFFICIENT sets, the DELETION set still provides an effective evaluation of a test suite's thoroughness. The difference in accuracy could possibly be attributed to the FULL and SUFFICIENT sets producing more *equivalent mutants* (mutants that are functionally identical to the original program, and thus impossible to kill) than the DELETION set.[5] This would

[5]We do not omit equivalent mutants in this analysis. Automatic identification of equivalent mutants is an impossible task [32], and manual inspection would render incremental feedback impossible in practice. See §6.8.

Table 6.4: Difference effect sizes in accuracy and cost between operator subsets. Accuracy and cost both decrease in the order `FULL` → `SUFFICIENT` → `DELETION`. The decrease in cost is more pronounced.

Subset Pair	**Accuracy**		**Cost**	
	μ	Cohen's d (w.r.t. `FULL`)	μ	Cohen's d (w.r.t. `FULL`)
`FULL`	1.00	—	3430.56	—
`SUFFICIENT`	0.997	1.14	1594.68	2.07
`DELETION`	0.995	1.25	678.09	3.63

be consistent with previous findings that the SDL operator tends to generate few equivalent mutants ($< 4\%$) [54]. It is also intuitive, since equivalent mutants under the `DELETION` set would indicate redundant code.

The running time cost of the `DELETION` set shows notable savings over the `FULL` set (§6.3). However, there is still a need for improvement if one were to use the `DELETION` set to offer automated feedback in an AAT. This is particularly true for larger and more complex projects, i.e., those in the CS3 corpus. The `DELETION` set took a median of 4.75 seconds to run per submission in the CS2 course, and 1.11 *minutes* per submission in the CS3 course.

Using the `DELETION` set is considerably cheaper than using the `SUFFICIENT` set, which in turn is cheaper than using the `FULL` set. These sizeable differences in cost coupled with relatively smaller differences in accuracy suggest that the PIT `DELETION` set is a promising path forward for cost-effective mutation analysis.

This analysis can be seen as a replication study that gathers more support for previous work evaluating deletion operators. We found support for findings from Untch [162], Offutt et al. [49, 54], and Dereziǹzka et al. [56] that found mutation by deletion to be highly cost-effective, and partial support for Shams [151], who found that statement deletion (SDL) represented a promising path toward the use of mutation testing in an educational context.

6.5 Research Question 3

RQ3: Can the cost of mutation by deletion be reduced further?

In this section, we investigate whether a subset of the `DELETION` set performs comparably well at approximating `FULL` coverage.

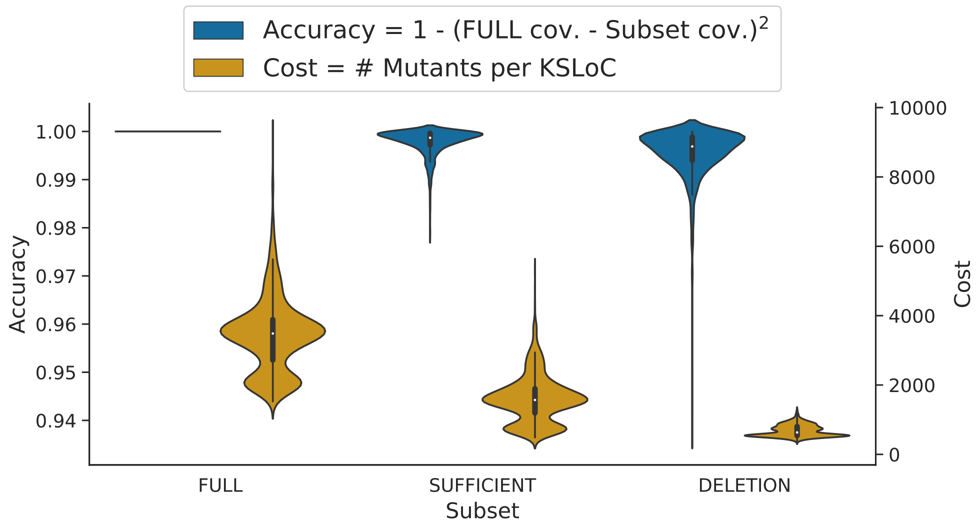


Figure 6.2: Accuracy and cost of the `DELETION`, `SUFFICIENT`, and `FULL` subsets of mutation operators ($n = 1389$). All three subsets have high accuracy, but the `DELETION` set is considerably cheaper than the `FULL` and `SUFFICIENT` sets.

6.5.1 Method

We formulate our problem as a regression problem:

- **Independent variables:** Coverage percentage achieved under individual operators
- **Dependent variable:** Coverage percentage achieved under the `FULL` set of PIT operators.
- **Data points:** Project submissions (Table 6.2)

We used a forward-selection procedure to select a subset of mutation operators out of an initial superset. We fit linear models in each step using the `statsmodels` [149] Python module. The goal is to produce a subset of operators while incurring acceptable losses in effectiveness, which is a form of *selective mutation*.

Forward selection [26] is a statistical model selection method. Starting with an empty model, (i.e., with no features), we consider features one at a time, checking to see how much each one improves the model. The best-performing feature is added and the procedure is repeated for all remaining features. This process repeats until the model stops improving, or until there are no more features. Forward selection is generally used when the initial number of features is large, and one wishes to select a small subset.

Our features are individual mutation operators. However, each feature carries with it a considerable computational cost. Therefore, forward selection is an appropriate feature selection strategy since it will (theoretically) help reduce the number of operators while maintaining overall effectiveness.

Table 6.5: Forward selection on the entire corpus of submissions, choosing `DELETION` operators. Highlighted cells contain values from the final model. Other cells contain cumulative values for intermediate models, after adding each operator.

Step	**Operator Added**	**# Mutants Generated**			**Adj. R^2**	**Coeff.**	**Std. Error**
		Median	% of `DELETION`	% of `FULL`			
	(intercept)	—	—	—	—	0.03	0.008
1	*RemoveConditionals*	102	36.04%	7.04%	0.78	0.35	0.011
2	*AOD*	140	49.47%	9.67%	0.88	0.19	0.007
3	*NonVoidMethodCalls*	236	83.39%	16.30%	0.91	0.28	0.012
4	*VoidMethodCalls*	240	84.81%	16.57%	0.92	-0.04	0.005
5	*MemberVariable*	271	95.76%	18.72%	0.92	0.06	0.009
6	*ConstructorCalls*	283	100.00%	19.54%	0.92	0.04	0.007

We start with no operators, and at each step we add the operator that minimises the Bayesian Information Criterion (BIC) [148]. If two operators perform equally well when added to the model, we select the one with lower cost, i.e., the one that produces fewer mutants. BIC was chosen over R^2 since it is better at predicting model performance on future, unseen data. It was chosen over the closely related Akaike Information Criterion (AIC) [26] because BIC penalises additional features more heavily than AIC and might result in a simpler model. This benefits our aim of reducing the number of mutation operators. The procedure stops when none of the remaining operators reduce BIC any further.

We used the procedure described above to incrementally choose operators in order of decreasing value-added. Since our goal is to minimise cost, we chose operators from the cheapest known-good subset of mutation operators, the `DELETION` set. At each step, we add the next best operator that further improves the model according to BIC.[6]

6.5.2 Result

A small subset of `DELETION` operators is responsible for most of the `DELETION` set's value, indicating that its cost can be reduced further. Applying this process to the entire corpus of 1389 submissions yielded `DELETION` operators in the order described in Table 6.5. Highlighted cells indicate estimates, errors, and cost from the final model, and other cells indicate cost and performance from intermediate models considered during

[6]We report adjusted R^2 for the sake of interpretability.

forward selection. Notice that all the `DELETION` operators were included in the final model, suggesting that each of them brings some additional explanatory power to the model.

The `DELETION` operators explained 92% of the variance in mutation coverage achieved under the `FULL` set (see the highlighted R^2 value in Table 6.5), while doing just under 20% of the work. This is in keeping with previous work [49, 54, 56, 162] that found mutation by deletion to be highly effective, and lends further support to our findings regarding **RQ2** (§6.4).

Additionally, a small subset of `DELETION` operators is responsible for most of its effectiveness. Model improvement tended to plateau after the first three operators were selected. The *RemoveConditionals* and *AOD* operators alone performed reasonably well at predicting coverage under the `FULL` set (adj. $R^2 = 0.88$). *NonVoidMethodCalls* was selected next, bringing with it a slight increase in effectiveness: R^2 goes from 0.88 to 0.91. The addition of subsequent operators resulted in moderate successive increases in cost, and the model never improved beyond adj. $R^2 = 0.92$. These diminishing returns suggest that, after a certain point, additional `DELETION` operators are not worth the cost they incur.

6.6 Research Question 4

RQ4: Are the benefits of different mutation strategies project dependent?

Recall that our goal is to select a cheaper subset of the `DELETION` set that provides a good approximation of coverage achieved under the `FULL` set. The models presented in Table 6.5 are based on the entire corpus of 1389 submissions. Observe that this is a considerably heterogeneous corpus of programs, in terms of both size and complexity (see Table 6.2). We hypothesise that the choice of `DELETION` operators differs based on the actual programs under test. Specifically, we hypothesise that the *larger* a program is, the *fewer* operators it requires to make this approximation. If this is true, then we may be able to avoid doing unnecessary work to evaluate test suites present in larger programming projects (e.g., those found in upper-level CS courses).

6.6.1 Method

Clustering projects by size. To test this hypothesis, we split the corpus of submissions based on the number of source lines of code (SLoC). Splitting was performed using Jenks natural breaks optimization [92].[7] The main idea behind this splitting technique is to 1) maximise the variance between groups, and 2) minimise the variance within groups.

We used goodness of variance fit (GVF) [91] to determine the appropriate number of splits. GVF is a $0 \rightarrow 1$ measure that is directly proportional to between-group variance, and

[7]Can be seen as a variation of K-Means clustering [112] applied to 1-dimensional data.

inversely proportional to within-group variance. Therefore, we would like to maximise it. To determine the appropriate number of splits k, we applied the Jenks algorithm for increasing values of k from 2 to 7 and plotted the GVF for each splitting. The diminishing improvements in GVF (Figure 6.3) indicated $k = 4$ to be an appropriate number of splits for this dataset. The four submission groups SG1–SG4 and their intervals are depicted in Figure 6.4.

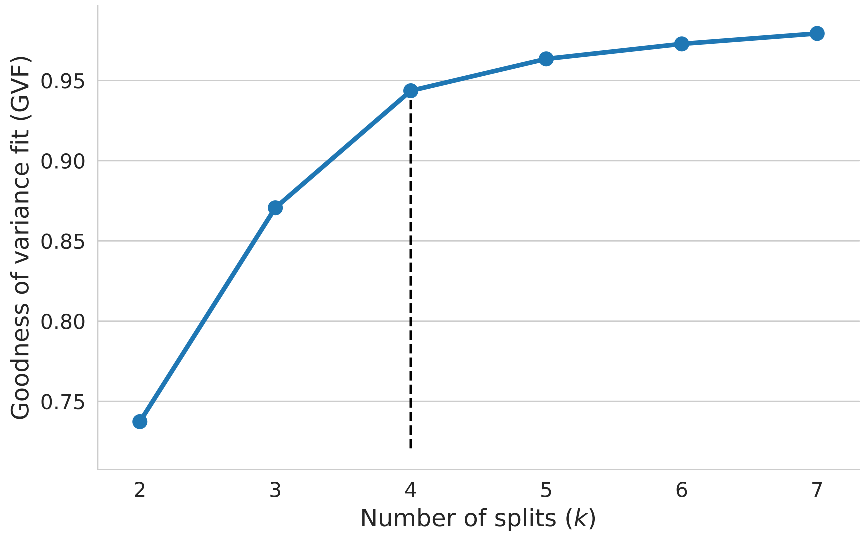


Figure 6.3: Goodness of variance fit (GVF) for increasing values of k using Jenks natural breaks optimization.

Incremental subsets. We incrementally built "n-operator" subsets of `DELETION` operators for increasing values of n. Operators were selected one at a time in the order obtained through forward selection (Table 6.5), and the resulting subset was evaluated separately against all groups of submissions using linear regressions.

The running time for each incremental subset was approximated by obtaining a "time-per-mutant" multiplier TPM for each submission. This was obtained using the following ratio (where $m_{\texttt{FULL}}$ is the number of mutants produced under the `FULL` set, and $t_{\texttt{FULL}}$ is the total running time for the `FULL` set):

$$TPM = \frac{t_{\texttt{FULL}}}{m_{\texttt{FULL}}}$$

Then for a given subset of operators S, its running time t_S for a submission was approximated as:

$$t_S = TPM \times m_S$$

Normalizing by program size, we obtain a composite measure of running time cost for a given subset of operators, in seconds-per-KSLoC (Table 6.6).

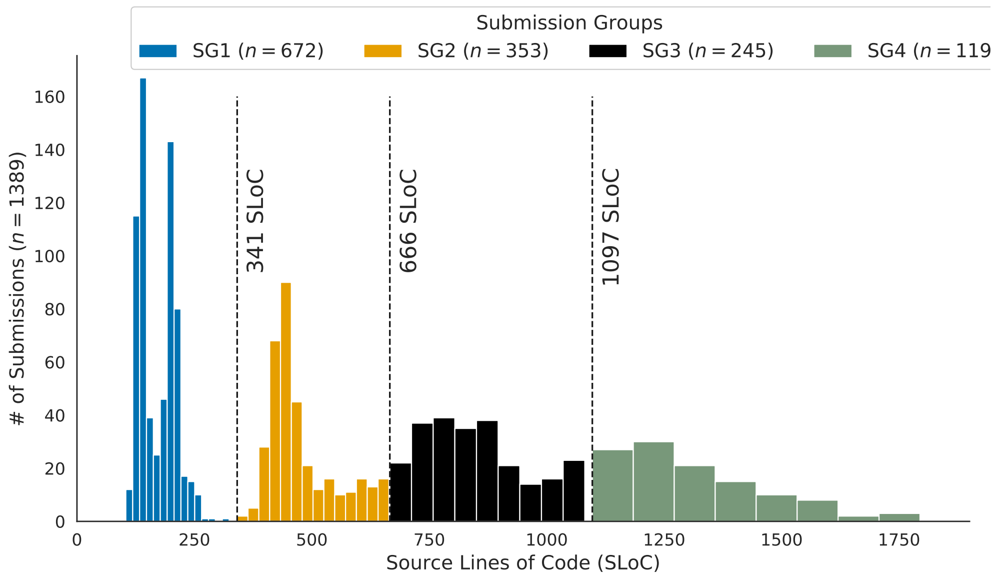


Figure 6.4: Groups of submissions based on SLoC. Dashed lines indicate group boundaries.

6.6.2 Result

Mutation adequacy on larger projects can be approximated with fewer mutation operators. Results are summarised in Figure 6.5. Each bar indicates the submission group under test, and each group of bars indicates the incremental subset being evaluated. The y-axis indicates the percentage of variance in `FULL` coverage that can be explained by the specified subset, for the given submission group. The incremental subsets proposed are included in Table 6.3. Median subset running times for each submission group and for the entire corpus are presented in Table 6.6.

1-op Subset. The first subset comprises only the *RemoveConditionals* operator, which removes conditionals by replacing them with boolean literals (**`true`** or **`false`**).

The 1-op Subset shows poor performance for SG1, the group of small submissions (see the first bar in the first group in Figure 6.5). For the groups in the middle, SG2 and SG3, *RemoveConditionals* is able to explain 88% and 86% of the variance in `FULL` coverage, respectively. It is able to explain 90% of the variance in `FULL` coverage for group SG4 (the group containing the largest submissions).

2-op Subset. This subset contains the 1-op Subset plus the *AOD* operator, with removes arithmetic operators from statements by systematically removing each operand.

This subset does better at predicting `FULL` coverage for all submission groups, with a to-be-expected increase in cost. Two operators—*RemoveConditionals* and *AOD*—are able to explain over 92% of the variance in `FULL` coverage for groups SG2–SG4. The subset still

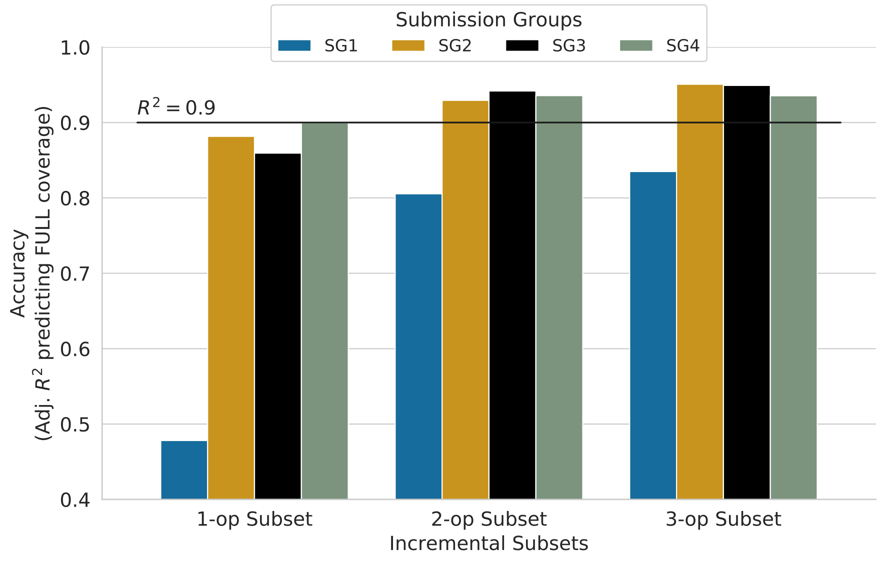


Figure 6.5: Effectiveness of incremental subsets used to predict `FULL` coverage for submission groups.

performs relatively poorly for SG1, with adjusted $R^2 = 0.80$.

3-op Subset. This subset contains the 2-op Subset plus the *NonVoidMethodCalls* operator. It removes calls to non-void methods by replacing their return values the with given type's default value.

The inclusion of *NonVoidMethodCalls* results in negligible improvements in model performance for all submission groups. The model continues to perform well for groups SG2–SG4 (adj. $R^2 > 0.94$), and it continues to perform poorly for group SG1 (adj. $R^2 = 0.84$). Note that the addition of the *NonVoidMethodCalls* operator adds nearly 50% to the costs incurred by the previous subset for each submission group (see Table 6.6).

6-op Subset. For the sake of brevity, we jump to results for the entire available set of `DELETION` operators, i.e., containing all 6 deletion operators listed in Table 6.3.

With the entire `DELETION` set included, models are able to explain a high amount of variance in `FULL` coverage (94% or higher) for submission groups SG2–SG4. For group SG1, the model is only able to explain 85% of the variance in `FULL` coverage. For all groups, this represents a small improvement from the 3-op subset.

6.7 Discussion

In this section, we discuss the implications of our results.

Table 6.6: Median running times (in seconds) using incremental subsets of operators, for each submission group, and for the entire corpus normalised by program size.

Group-wise and overall running time medians (seconds).

	Per group				All groups
	SG1	SG2	SG3	SG4	Time / KSLoC
`FULL`	16.11	83.81	283.42	325.06	205.64
`DELETION`	3.93	15.84	63.39	66.78	34.77
3-op Subset	2.76	13.50	50.80	54.58	27.91
2-op Subset	1.63	8.40	29.52	31.25	19.52
1-op Subset	1.19	6.16	20.08	20.83	13.95

6.7.1 Choosing a Subset of Operators

The ***`DELETION`*** *set, though cheaper than the* ***`FULL`*** *and* ***`SUFFICIENT`*** *sets (Figure 6.2), incurs unproductive cost.* Consider the mutation operators chosen through the lens of cost-effectiveness. We have seen in sections 6.5 and 6.6 that `DELETION` operators' ability to approximate `FULL` coverage improves and then tapers off after a few mutation operators have been chosen. Based on the changing R^2 values in Table 6.5, one might conclude that the critical point is after the second (*AOD*) or third (*NonVoidMethodCalls*) operator is added to the model. However, consider the cost incurred by this third operator. The first two operators chosen—*RemoveConditionals* and *AOD*—together account for just under half of all `DELETION` mutants, and just under 10% of all mutants under the `FULL` set, but are able to explain 88% of variance in `FULL` coverage. Including the *NonVoidMethodCalls* operator increases the total cost of the previous two operators by nearly 50%, (3-op Subset in Table 6.6), but only explains an additional 3% (91%) of the variance, which is a relatively small improvement over the previous subset.

Figure 6.6 is a "zoomed in" version of Figure 6.2, with the `FULL` and `SUFFICIENT` sets excluded, and the 1-op and 2-op subsets included. For extremely small losses in accuracy, subsets of the `DELETION` set are able to bring huge cost savings. Taking cost and accuracy with respect to `FULL` coverage into account, we conclude that in general the **2-operator subset is the most practical set for fast and effective mutation analysis**.

Why does RemoveConditionals perform so effectively by itself? For the groups of larger submissions SG2–SG4, *RemoveConditionals* alone proved to be tremendously effective at approximating `FULL` coverage (§6.6). This operator replaces conditionals with Boolean literals, effectively excluding (or ensuring the execution of) all statements controlled by a condition. Mutation analysis using this operator has strong ties to *object branch coverage* (OBC) [151], one of the strongest forms of code coverage for Java programs. OBC requires students to write tests that exercise every Boolean condition generated in their solution's compiled bytecode. *RemoveConditionals* can be seen as a stronger form of this measure,

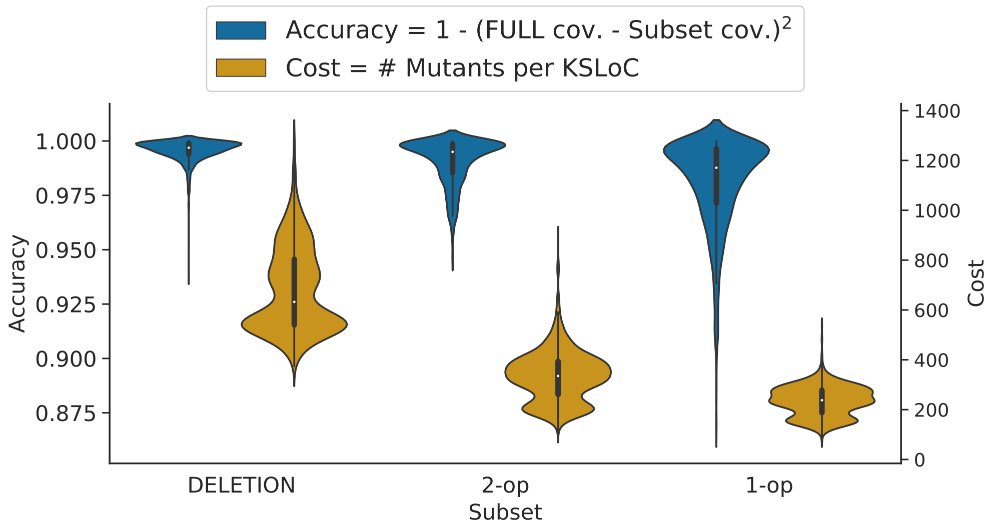


Figure 6.6: Accuracy and cost of the `DELETION`, 2-op, and 1-op subsets of mutation operators. This figure is a "zoomed in" version of Figure 6.2, with the `FULL` and `SUFFICIENT` sets no longer included.

since it is sensitive to not only the *execution* of logical branches, but also to the *propagation of program state or output* from those logical branches to the tests, i.e., through assertions.

Also consider the kinds of programs under test in this study. We included submissions from an upper-level Data Structures & Algorithms (CS3) course, nearly all of which were clustered in submission groups SG2–SG4. These projects require significant control flow components to implement complex behaviors, so it is plausible that focusing on conditions in the control flow logic would imply the most critical aspects of quality testing. We recommend that AATs **use the 1-op subset for larger projects (SLoC** > 341**)**.

How does one evaluate tests for smaller submissions (group SG1)? In §6.6, notice that the 3-op subset—or indeed, the entire `DELETION` set—is unable to achieve a good approximation of coverage under the `FULL` set for smaller submissions. This throws into question whether selective mutation is an effective approach for these projects. In fact, the time to run mutation analysis on these submissions is so low ($\mu = 0.37$ minutes, $\sigma = 0.28$ minutes) that one might consider simply using the `FULL` set of PIT operators instead of a cheaper approximation.

Submissions in SG1 overwhelmingly belong to early assignments in the CS2 course. They are small and simple codebases that present comparatively fewer opportunities for mutation, even when normalizing by program size. They generate an order of magnitude fewer mutants per KSLoC than projects in SG2–SG4. Submissions in SG1 generate an average of 2772 ($\sigma = 719$) mutants per KSLoC, while submissions in SG2, SG3, and SG4 generate an average of 3896 ($\sigma = 738$), 4076 ($\sigma = 1104$), and 4443 ($\sigma = 1034$) mutants per KSLoC, respectively.

```java
1 public int probeSquare(int i) {
2   return i * i;
3   // Tests did not check the use of this arithmetic expression.
4 }
```

Listing 1: A snippet highlighting a line that contained a surviving mutant (similar to reports emitted by PIT.

An analysis of variance confirmed that the number of mutants per KSLoC is significantly different for different submission groups. Post-hoc analysis using Tukey's HSD test showed that the pairwise differences in mutants per KSLoC between groups SG2–SG4 is at least an order of magnitude less than the difference between SG1 and each of the other submission groups ($p < 0.05$ for all pairs). In light of these differences in the effectiveness and cost of mutation testing on our data set, we recommend that AATs **use the `FULL` set of mutation operators for small submissions (SLoC ≤ 341)**.

It is also worth considering whether mutation testing is an over-engineered test evaluation strategy for programs of such minimal complexity. When cyclomatic complexity is low, infected program state is much more likely to propagate to test output, assuming that the faulty code is executed by the test suite. This assumption is usually satisfied by AAT requirements for complete condition coverage. These facts mitigate much of the threat associated with code coverage measures for small and simple software projects.

6.7.2 Operationalising Feedback

What might feedback based on mutation analysis look like? Ultimately, the goal of our research is to improve the quality of student-written test suites. Mutation analysis only furthers this goal if the students get feedback about the process in some way. Similar to code coverage, it is easy to generate feedback for students by highlighting the lines of code that contain undetected mutations. Consider the code snippet in Listing 1. The *AOD* mutation operator was applied to the highlighted line (line 2), changing it to **`return i`**. The highlight indicates that *all tests passed* even with the specified mutation in place. In other words, no test behaves differently whether the output is **`i`** or **`i * i`**. A combination of information—the highlighted line and the exact mutation that was applied—gives the student an explicit strategy for improving the test suite based on the provided feedback, i.e., write a test that makes an assertion about the function's return value. Similar feedback may be devised for other mutation operators.

Is this any better than code coverage? Listing 1 depicts a real function from our corpus of submissions, that achieved complete object-branch coverage but zero mutation coverage. It is a simple probing function that helps determine a record's position in a hash table. The student's tests only verified that the size of the hash table increased after each insertion, but they never verified that records were inserted at the right positions. Code coverage measures were unable to detect this deficiency, since the **`probeSquare`** function was executed ("covered") during the insertion process.

Indeed, this discrepancy was reflected in submissions across our entire corpus. Object branch coverage (a strong form of code coverage) scores tended to cluster close to the 100% mark ($\mu = 0.98, \sigma = 0.03$), while mutation coverage using only the 2-op Subset (*RemoveConditionals* and *AOD*) tended to be lower ($\mu = 0.81, \sigma = 0.18$). We observed an insignificant Pearson correlation between the two measures ($r = -0.01, p = 0.58$).

An intuitive explanation for the distributions above is that students naturally try to score highly on the measures for which they are incentivised to do so. Whether idealistically or to maximise incentives, students in our dataset attained near-perfect code coverage scores. Our hope is that this tendency will remain when mutation analysis is used for feedback instead, resulting in test suites with much better guarantees of thoroughness due to its strength as a test adequacy criterion (§6.1.2).

6.8 Threats to Validity

Internal. We did not exclude equivalent or duplicate mutants from those generated by PIT. In general, equivalent mutants manifest in mutation analysis by artificially driving *down* the mutation coverage score (since it creates unkillable mutants), and duplicate mutants may manifest by artificially driving *up* the score (since it means multiple mutants are killed by the same set of tests). The problem of identifying these mutants automatically is undecidable [32], but can be done heuristically by manual inspection of programs. This was infeasible due to the size of our submission corpus, and is impossible to operationalise in an incremental feedback context. The size of our corpus may have helped to mitigate this threat. Additionally, in a study that aims to reduce the cost of mutation analysis in AATs, excluding this inevitable cost could lead to inaccurate results. Where possible, we reduced the occurrence of duplicate mutants such that they could also be omitted when deployed in an AAT (§6.3).

External. As with any educational research, the generalizability of our results may be threatened by our sample of students and their submissions. We tried to mitigate this by studying 1389 distinct submissions from 7 programming assignments in 2 CS courses. Submissions between and within assignments were heterogeneous in terms of size and complexity of both source code and test code. Testing the generalizability of these results on open-source or industry projects would be interesting future work.

Construct. We studied PIT, a mature mutation testing tool available for Java. As described in §6.2.2, it is currently the most robust, easy-to-use, and practical mutation testing tool for the JVM, making it the most practical choice for fast feedback based on mutation analysis. Nevertheless, we do not use any direct measures of software test quality here, such as measuring bug revealing capability. Instead, we use coverage on the FULL set of PIT operators as a proxy for measuring test quality, relying on existing theory [51] and research on the validity of mutation analysis [4, 97]. We are encouraged by the fact that Shams [151] performed an assessment of deletion mutators in terms of measuring test suite bug detection ability and found them to be more effective than code coverage measures, but these results still depend on the validity of the relationship between mutation analysis and test quality.

6.9 Summary

We have built on the current state of research to devise a cost-effective mutation strategy to produce accurate, incremental, and scalable feedback on the quality of student-written software tests. This approach provides a better assessment of how well software tests check expected behaviours, and can be used to generate feedback for students. We improved upon the most efficient mutation operator set previously proposed in the literature, the DELETION set. For the projects we studied, the *RemoveConditionals* and *AOD* operators produced results comparable to the most stringent set of operators at 1/10th the cost, achieving *less than half the cost* of the best mutation approach proposed so far. Cost savings such as these would go a long way toward being able to apply mutation analysis in practice in auto-graders such as ASSYST or Web-CAT.

Chapter 7

Conclusions and Future Work

7.1 Summary of Findings and Conclusions

This dissertation presents methods to quantitatively measure aspects of the software development process as followed by students. The eventual goal is to use these measurements to provide students with formative feedback about their software development habits as they work on projects, though no intervention experiments were conducted as part of this work. In addition, to address challenges with evaluating the strength of student-written software tests, we investigated the utility of *mutation analysis* as a test adequacy criterion. We addressed its primary limitation: its computational cost.

This dissertation provides a number of contributions to computing education.

7.1.1 Process Metrics

Procrastination. In Chapter 4, I defined the *Early/Often Index*, which measures the mean number of days before the project deadline when the student tends to write their code. It represents a quantification of procrastination on software projects: a smaller number indicates that work tended to be done closer to the deadline (i.e., procrastination occurred).

To validate the measure, I conducted semi-structured interviews with CS3 students in which we talked about their development habits, loosely following the script in Appendix A. Additionally, I manually inspected snapshot histories that were captured by DevEventTracker. Details of this qualitative validation are presented in §4.3. Results indicated that the measure was generally accurate at characterising when a student tended to work on a given project, showing discriminatory power between early and late work from different students, and early and late work from the same student on different projects.

We found that students tended to work on projects less than 10 days before the deadline on average, even though they were 3–4 weeks to work on each project. Relationships between this metric and project outcomes aligned with experiential and theoretical expectations of the effects of procrastination. When students worked earlier and more often:

- They were more likely to solve the assigned problem (i.e, achieve $> 95\%$ correctness)
- They were more likely to finish their projects earlier and on time

- They did not spend any more or less time on projects

Additionally, we considered the edit median time (calculated just like the Early/Often Index, but as a median instead of mean). We found that when students had earlier test edit median times (i.e., when they finished half of their testing earlier in the project lifecycle), they tended to spend more time on their projects. This relationship was not present for the test edit mean time. It is possible that when students did the bulk of their testing earlier, they were more comfortable trying different approaches to the problem, or they had to re-design solutions based on clarifications that were made available later in the project lifecycle.

Analyses were conducted using a within-subjects repeated measures design, meaning that inferences were made by comparing the work of the same students on different projects with each other. This helps us avoid making inferences based on traits or factors that are inherent to individual students. Therefore, while our findings are not strictly causal, but we are confident acting upon them to design interventions (proposed in §7.2).

Writing and running software. Also in Chapter 4, I described metrics to measure the amount of time that passed between writing the project and executing it. This can be measured with varying combinations of solution code or test code and normal executions and test executions. A smaller result would indicate that, on average, less time passes between when the student writes code and when they execute code (i.e., they engage in more incremental "self-checking" behaviours).

Similar to the Early/Often Index, these metrics were considered generally accurate after interviews with students and manual inspection of Git snapshots. We were unsurprised to find that students launch their programs very often, with an average of 1.37 hours passing between code edits and executions. However, we were surprised to find that the overwhelming majority of launches were test launches as opposed to normal program launches (solution $\mu = 55.66$ and test $\mu = 229.23$). In any case, we did not find any relationships between students' program or test launching practices and any project outcomes.

Incremental test writing. In Chapter 5, I presented metrics to measure the degree to which students engaged with software testing during the project lifecycle. In particular, I measured the *balance* and *sequence* of effort devoted to writing solution code and test code.

We defined a measure of *testing effort*: the proportion of code writing effort that was devoted to writing test code. This was then calculated and aggregated (as a median) across multiple axes: work sessions devoted to the project, individual methods in the project, and both. Additionally, we measured the amount of testing effort devoted to a given method before and after the method was completed, i.e., modified for the last time.

We found that, on average, around 20% of code writing effort in individual work sessions was devoted to writing test code. This might be indicative of students' disinclination to practise testing that other researchers have reported. In terms of individual methods, after filtering out getters, setters, and printing methods, we found that students were likely to directly invoke no more than 60% of their projects' methods in their test suites. Finally, we observed that students were more likely to practice incremental test-last development than they were to practice test-first development: in 85% of projects, students put in less than half of their testing effort before the code under test had been finalised.

We measured the metrics' relationships with two project outcomes: *correctness*, as measured by an oracle of instructor-written test cases, and *test suite strength*, as measured by condition coverage. We found that students were likely to produce higher quality software and tests when they devoted a higher proportion of their coding effort to testing during each work session in which they worked on their projects. Whether this testing occurred before or after the solution code under test was written was irrelevant to project correctness. This indicates that the *incremental* nature of testing was more important than whether the student practised TDD or ITL.

Additionally, students were likely to produce test suites with higher condition coverage when they:

- Conducted more testing *after* the code under test had been completed
- Devoted more testing effort to individual solution methods

The primary contribution of this work is that we are able to identify these behaviours with lead time before project deadlines. For example, at or before the halfway point of a project timeline, one could use metrics from §5.2 to determine the extent to which students are engaging with incremental testing. This would help set the stage for interventions or feedback mechanisms that could encourage effective software development habits and discourage ineffective ones.

7.1.2 Mutation Analysis

In addition to measuring incremental test writing process, it is important to appropriately assess test quality. We discussed the fact that previous options for measuring test adequacy suffer from a number of limitations, and that a more robust solution is required. To this end, we examined the feasibility of using *mutation analysis* as an automated assessment mechanism for evaluating student-written software tests. Experiments were conducted on a server setup that is similar to that of the Virginia Tech Web-CAT server, which serves many thousands of users. Upon finding that current approaches to mutation analysis were too expensive for the autograding context, we devised and evaluated new ones. Analyses and results are presented in Chapter 6.

We investigated the feasibility of using mutation analysis "as is" for feedback on CS2 and CS3 software projects. Comprehensive mutation took a median of approximately 30 seconds to run for CS2 projects, and 5 minutes to run for CS3 projects. These running times are too slow for providing incremental feedback and might result in community-wide slowdowns for the 20–30 institutions that use the Virginia Tech Web-CAT servers. We also found that mutation by deletion—which Shams found to be more cost-effective [151]—was infeasible for automated feedback, particularly for CS3 projects.

To evaluate less expensive subsets of mutation operators, we used linear models to measure the percentage of variance in comprehensive mutation coverage that was explained by a smaller subset of mutation operators. We confirmed that mutation by deletion is effective at approximating test adequacy: it 92% of the variance in comprehensive mutation coverage.

Reducing the cost mutation by deletion further, we found that a single mutation operator—replacing conditional statements with **`true`** and **`false`**—was tremendously effective at measuring test adequacy in larger projects. It explained approximately 90% of the variance and incurred only 8% of the cost, taking a median of 20.83 seconds to run for the largest codebases in our corpus ($\geq$ 1097 KSLoC). For smaller projects, an additional mutation operator—deleting arithmetic expressions—is required to make this approximation. This subset of two operators explained $>$ 90% of the variance for all but the smallest codebases ($\leq$ 341 KSLoC). For small codebases (like those submitted to early assignments in CS2), comprehensive mutation analysis is a feasible assessment mechanism, taking a median of only 16.11 seconds for these projects.

7.2 Future Work

7.2.1 Development Process Interventions

How can we use our metrics to change student development behaviours? In §4.6, I discussed a prototypical predictive model that achieved some accuracy at classifying projects as "solved" a project or not, based on their solution edit mean times and their test edit median times. A model like this might be used to aid in early identification of students who are following suboptimal development practices like procrastination or sporadic testing, thereby enabling formative assessments. In this section I discuss possible uses of the metrics described in Chapters 4 and 5 to power such interventions.

Ideally, we would like to support interventions *in real time*, i.e., while students are still working on projects. However, a challenge is that the Early/Often Index—as a measure of central tendency—is "backwards facing" (see §4.2.1). The mean or median of a student's distribution of work days can only be calculated (or approximated to a reasonable degree of accuracy) when all or most of the work has already been done. Unfortunately, this would mean that the student will already have faced the consequences of procrastinating on the project, which is precisely what we would like to avoid.

However, it should be possible to approximate a projection of a student's *future* work plans, based on the work of thousands of students from previous semesters on similar projects. Given the work done by a student *so far*, one might use time series forecasting methods to project what the rest of the student's work distribution will look like. In theory, we could obtain a distribution of work days that is part real data and part projected data, with the proportion that is real data increasing in size as time passes. This distribution can then be used to make increasingly accurate approximations of the mean edit time (i.e., the Early/Often Index). Since successful (solved) project solutions have mean edit times more than a week ahead of deadlines, it should be possible to predict performance with some degree of accuracy with some degree of lead time before the deadline.

The measurements of test writing described in §5.2 are readily amenable to incremental feedback. The *PSB* metric measures the testing effort being applied during each work session. Unlike the solution edit mean or median time, *PSB* begins to provide useful insights much earlier in the project lifecycle. For example, during the first or second week, we can tell how much testing effort the student is applying per work session, on average. This is information that can be acted on to provide formative feedback.

There are many potential interventions that could be driven by early identification of ineffective development practices.

Project grade. A portion of the project grade is already allocated based on properties other than correctness, such as the percentage of code covered by students' own tests, and the quality of the comments and program style. A natural step is to base a portion of the grade on an assessment of incremental development and time management practices. A grade-based intervention could work under Web-CAT's model of multiple submissions, by encouraging students to respond to rewards or penalties based on their development practice between intermediate submissions. It is unclear how this might affect development behaviours. There is research that shows that students performed better on project correctness when they were graded on the strength of their test suites [58]. However, students have also been shown to maximise incentives (i.e., points) by driving up their testing "scores" without actually improving the strength of their test suites [1, 153]. Would such interventions cause students to improve their software development behaviours? Or would it incentivise them to find ways to maximise scores without substantively changing behaviours?

A learning dashboard or a leader-board. Perhaps the solution need not be grade-based at all. Instead, it may be fruitful to encourage students to reflect on their behaviours over the course of a project, perhaps in comparison with their peers. Graphs showing the progression of solution code and test code over time (e.g., Figures 4.7–4.9) be automatically generated for each student, providing visual feedback on their programming process. One might also consider a leader-board that relates the individual's performance to that of the rest of the class. Making students aware of their standing in the class could provide more incentive for self-improvement than simply informing them of their own programming practices.

Adaptive emails. Previous work has discussed the effects of interventions with adaptive feedback on students' procrastination behaviours and project performance [120]. Students were sent emails with feedback generated from data about their last submitted work, and the effects were positive when compared to a control group. Our suite of metrics could be applied in a similar fashion. The feedback generated from data made available by DevEventTracker could be far more adaptive than that reported in [120].

Visual cues in the IDE. We have considered creating an Eclipse plugin that provides students with visual cues about their development process *in situ*. Bandura reports that self-regulatory behaviours are malleable by visual, verbal, and aural cues [10]. In a study conducted by Buffardi & Edwards, students reported that they found the red–green bars indicating the thoroughness of their testing to be the most important points of feedback (about testing) provided in the Web-CAT interface [34]. Using the metrics described in Chapters 4 and 5, we could provide more detailed, real-time feedback in students' IDEs themselves. For example, an unobtrusive widget in the IDE's toolbar might transition along a red–green gradient, indicating the balance of the student's testing and implementation effort during the current work session. It is important for an intervention like this to maintain a balance between unobtrusiveness and noticeability to the student.

Project milestones. Steel's meta-analysis [161] suggested that procrastination occurs when a task presents many junctures of choice [156] and when a person feels low self-efficacy about their ability to complete a task [10]. As a preliminary response to high rates of late or incomplete submissions, the CS3 course at Virginia Tech instituted *project milestones*—increments of project requirements that were due before the final deadline—that appeared to reduce rates of late submission and improve overall project performance. It is possible that this intervention succeeded because it counteracted the two procrastination correlates described above. The data analysed in Chapters 4 and 5 were not collected during semesters in which milestones were instituted. Future work will need to quantitatively evaluate the impact of project milestones, perhaps by examining its effect on, say, the Early/Often Index.

Other considerations. It is possible for the approaches described above to have negative effects on student performance, self-efficacy, or motivation. For example, if the student starts testing late in their project lifecycle, it might be difficult for them to counteract the low score they garnered from early work sessions. Should we still penalise bad practices from earlier in the lifecycle if the student is now engaging in better practices? Moving up a leader-board might supplant learning or project performance as the student's goal, potentially leading them to optimise for incentives rather than substantively changing their development habits. Alternatively, a student who is sufficiently behind may feel like no action they take is good enough to move them up the leader-board, leading to feelings of inadequacy or a lack of self-efficacy. We need to be cognizant of these potential side effects while deploying interventions.

7.2.2 Mutation Analysis

Feedback based on mutation analysis. In Chapter 6, we devised and evaluated a scalable approach to deploying mutation analysis in an AAT like Web-CAT. We are considering deploying feedback based on our results at Virginia Tech, starting with the CS3 course. In §6.7.2, I described a possible presentation of feedback based on mutation analysis. The feedback mode is quite similar to that of code coverage, except it focuses on undetected defects instead of unexecuted statements or conditions (see Listing 1). Before deploying this feedback, it is important to evaluate the pedagogical effectiveness of mutation analysis. As a preliminary step, we held 9 interviews with CS3 students, who indicated that they found feedback based on mutation analysis to be useful and actionable. We conducted 2-op mutation analysis (using *RemoveConditionals* and *AOD*) on their project submissions, and showed them instances of live mutants in their otherwise "covered" code. They were able to identify specific test cases that would have killed the mutants we showed them.

Evaluating the pedagogical value of mutants and mutators. Further work is needed to determine the degree to which mutation feedback is useful to CS students, and the factors that affect this usefulness. I define *useful* mutation feedback to mean feedback that guides the student to make targeted improvements to their test adequacy based on live mutants. How does the number of mutation operators used or mutants generated affect the student's ability to improve their testing? It is possible that too many reported live mutants could overwhelm students into thinking that sufficient testing is a mammoth task that they are not equipped to undertake. Are some mutation operators more useful than others? It is possible, for example, that students will more easily respond to feedback based on *RemoveConditionals*, since it has more direct ties to code condition coverage measures, with which they are familiar. We could borrow from program comprehension research to aid in answering this question.

Mutant selection based on program type. Another promising avenue for future work is in static determination of appropriate mutation operators for a given program. One might analyse programs to determine various characteristics (cyclomatic complexity, use of looping constructs, or others) that might impact mutation analysis. From those characteristics, it might be possible to derive a context-specific subset of practical mutation operators. While the space for such program characteristics may be intractably large for industry-grade codebases, it might be manageable for student projects. The question is then whether we can identify clusters of program types for which different subsets of operators perform better.

Closing the loop between Chapters 4–6. In Chapters 4 and 5, we used *project correctness* as an outcome variable measured by an oracle of tests written by course staff. But who tests the tests themselves? Future work should involve strengthening our oracles using mutation analysis to ensure that our measurement of correctness is itself correct and complete. Similarly, in Chapter 5 we used *test suite quality* as an outcome variable, measured by condition coverage. It would be interesting to augment the analysis by examining the relationships between incremental testing practices and the different mutation analysis approaches described in Table 6.3.

7.2.3 Long-Term Research Plans

Instructional improvements. We know the following about procrastination and students' test writing habits:

1. People procrastinate on tasks that they perceive to provide little value ([161])
2. Students often do not see the value in software testing ([11, 35])
3. Students are not practising regular software testing (see Chapter 5 in this dissertation)

These three findings together provide a fairly clear path forward: design instruction that makes clear to students the value of software testing. For example, a software engineering or software testing course might involve discussions of high-profile software failures and how they might have been avoided by testing methodologies [5].

Additionally, we should be designing software assignments that more closely mimic real-world software development, where the value of testing is more apparent. In §5.5, we discussed the fact that students' incremental test writing practices explained a significant but ultimately small proportion of variance in eventual project outcomes. My conjecture is that students are able to "test" their programs using the readily available instructor-created oracle (i.e., Web-CAT), and therefore do not see value in conducting their own systematic testing. Preliminary interviews held with students during the Fall semester of 2019 have mostly confirmed this. Removing or reducing the ability to rely on such an oracle may be a good first step. Irwin & Edwards have made some progress toward this goal by using gamification methods to throttle the frequency with which students can make submissions to Web-CAT [86].

Longitudinal studies. Once interventions, feedback mechanisms, or instructional improvements have been deployed, it would be interesting to conduct longitudinal studies to investigate subjects' efficacy in their first engineering jobs. This would give us a real measure of the impact we are having at mitigating the difficulties we discussed at the beginning of this dissertation.

Good development process for end-user software developers. The development of software—historically the sole purview of trained software professionals—is increasingly being carried out in a professional capacity by people with varying intents and motivations [128]. There are far more *end-user programmers* today than professional software developers [38]. For example, data analysts often maintain computational notebooks or spreadsheets to help make sense of data. Information visualization specialists might follow more exploratory or reactive development cycles than typical software engineers. Ko et al. have done important work identifying the barriers faced by end-user programmers in today's tools and technologies [105]. What does effective programming process look like for people in these roles? How is this similar to or different from "traditional" software engineering best practices? How can we design instruction for students preparing for these roles, keeping in mind their motivations and intents for practising computing?

7.3 Final Remarks

Software development is a skill. Like any skill, it requires practice and feedback in order to develop. In this dissertation, I have contributed methods to empirically characterise software development habits as effective or ineffective in real time, i.e., as students work toward project completion. In doing this, I have set the stage for instructors to provide formative feedback on various aspects of the software development process. The primary hypothesis that remains to be tested is that this kind of feedback will enable to students to achieve improved project outcomes (e.g., higher project quality, reduced likelihood of incomplete or abandoned submissions, and reduced rates of late submissions).

Findings from this research are also relevant to software engineering researchers and practitioners. The intermediate-to-advanced undergraduates that were studied are only a year away from entering the professional workforce, and the research methods used (developer interviews, IDE log analysis, and software repository mining) were borrowed from the software engineering research community. Professional engineers often follow project-based workflows with deadlines, deliverables, and demands on their time that are not unlike the ones that students grapple with in university.

Bibliography

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 153–160, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869567. URL http://doi.acm.org/10.1145/1869542.1869567.

[2] George Ainslie. Specious reward: a behavioral theory of impulsiveness and impulse control. *Psychological bulletin*, 82(4):463, 1975. doi: 10.1037/h0076860.

[3] Amjad Altadmri and Neil C.C. Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, page 522–527, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450329668. doi: 10.1145/2676723.2677258. URL https://doi.org/10.1145/2676723.2677258.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062530. URL http://doi.acm.org/10.1145/1062455.1062530.

[5] Maurício Aniche, Felienne Hermans, and Arie van Deursen. Pragmatic software testing education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 414–420, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287461. URL http://doi.acm.org/10.1145/3287324.3287461.

[6] Maurício Finavaro Aniche. Repodriller. https://github.com/ayaankazerouni/repodriller, 2018.

[7] Maurício Finavaro Aniche and Marco Aurélio Gerosa. Most common mistakes in test-driven development practice: Results from an online survey with developers. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 469–478. IEEE, 2010. doi: 10.1109/ICSTW.2010.16.

[8] Richard D. Arvey, Maria Rotundo, Wendy Johnson, Zhen Zhang, and Matt McGue. The determinants of leadership role occupancy: Genetic and personality factors. *The Leadership Quarterly*, 17(1):1 – 20, 2006. ISSN 1048-9843. doi: https://doi.

org/10.1016/j.leaqua.2005.10.009. URL http://www.sciencedirect.com/science/article/pii/S1048984305001232.

[9] Dave Astels. *Test driven development: A practical guide.* Prentice Hall Professional Technical Reference, 2003.

[10] Albert Bandura. *Self-Efficacy: The Exercise of Control.* Macmillan, 1997. ISBN 978-0716728504.

[11] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. An experience in integrating automated unit testing practices in an introductory programming course. *SIGCSE Bull.*, 34(4):125–128, December 2002. ISSN 0097-8418. doi: 10.1145/820127.820183.

[12] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. Fitting linear mixed-effects models using lme4. *Journal of Statistical Software, Articles*, 67(1):1–48, 2015. ISSN 1548-7660. doi: 10.18637/jss.v067.i01.

[13] Lewis Baumstark and Michael Orsega. Quantifying introductory cs students' iterative software process by mining version control system repositories. *J. Comput. Sci. Coll.*, 31(6):97–104, June 2016. ISSN 1937-4771.

[14] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[15] Brett A. Becker. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 296–301, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342315. doi: 10.1145/2899415.2899463. URL https://doi.org/10.1145/2899415.2899463.

[16] Andrew Begel and Beth Simon. Novice software developers, all over again. In *Proceedings of the Fourth International Workshop on Computing Education Research*, ICER '08, page 3–14, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582160. doi: 10.1145/1404520.1404522. URL https://doi.org/10.1145/1404520.1404522.

[17] Andrew Begel and Beth Simon. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, page 226–230, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595937995. doi: 10.1145/1352135.1352218. URL https://doi.org/10.1145/1352135.1352218.

[18] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190,

New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786843.

[19] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 559–562, Piscataway, NJ, USA, 2015. IEEE Press.

[20] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. How to catch 'em all: Watchdog, a family of ide plug-ins to assess testing. In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice*, SER&#38;IP '16, pages 53–56, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4170-7. doi: 10.1145/2897022.2897027. URL http://doi.acm.org/10.1145/2897022.2897027.

[21] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: Industrial case studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 356–363, New York, NY, USA, 2006. ACM. ISBN 1-59593-218-6. doi: 10.1145/1159733.1159787.

[22] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, May 2009. doi: 10.1109/MSR.2009.5069475.

[23] Paul Black and Dylan Wiliam. Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice*, 5(1):7–74, 1998. doi: 10.1080/0969595980050102. URL https://doi.org/10.1080/0969595980050102.

[24] Paul Black and Dylan Wiliam. Developing the theory of formative assessment. *Educational Assessment, Evaluation and Accountability*, 21(1):5–31, Feb 2009. ISSN 1874-8597, 1874-8600. doi: 10.1007/s11092-008-9068-5.

[25] David Bowes, Tracy Hall, Jean Petrić, Thomas Shippey, and Burak Turhan. How good are my tests? In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, pages 9–14. IEEE Press, 2017.

[26] Hamparsum Bozdogan. Model selection and akaike's information criterion (aic): The general theory and its analytical extensions. *Psychometrika*, 52(3):345–370, 1987.

[27] Michael K. Bradshaw. Ante up: A framework to strengthen student-based testing of assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 488–493, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2966-8. doi: 10.1145/2676723.2677247. URL http://doi.acm.org/10.1145/2676723.2677247.

[28] Eric Brechner. Things they would not teach me of in college: What microsoft developers learn later. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, page 134–136, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137516. doi: 10.1145/949344.949387. URL https://doi.org/10.1145/949344.949387.

[29] Neil C. C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. Blackbox, five years on: An evaluation of a large-scale programming data collection project. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, page 196–204, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356282. doi: 10.1145/3230977.3230991. URL https://doi.org/10.1145/3230977.3230991.

[30] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, page 43–50, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327558. doi: 10.1145/2632320.2632343. URL https://doi.org/10.1145/2632320.2632343.

[31] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 223–228, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2605-6. doi: 10.1145/2538862.2538924.

[32] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, Mar 1982. ISSN 1432-0525. doi: 10.1007/BF00625279. URL https://doi.org/10.1007/BF00625279.

[33] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, page 220–233, New York, NY, USA, 1980. Association for Computing Machinery. ISBN 0897910117. doi: 10.1145/567446.567468. URL https://doi.org/10.1145/567446.567468.

[34] Kevin Buffardi and Stephen H. Edwards. 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, 2012. Association for Computing Machinery. ISBN 9781450312462. doi: 10.1145/2325296.2325324. URL https://doi.org/10.1145/2325296.2325324.

[35] Kevin Buffardi and Stephen H. Edwards. A formative study of influences on student testing behaviors. In *Proceedings of the 45th ACM Technical Symposium on Computer*

Science Education, SIGCSE '14, pages 597–602, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2605-6. doi: 10.1145/2538862.2538982.

[36] Kevin Buffardi and Pedro Valdivia. The significance of positive verification in unit test assessment. In *52nd Hawaii International Conference on System Sciences, HICSS 2019, Grand Wailea, Maui, Hawaii, USA, January 8-11, 2019*, pages 1–10. ScholarSpace / AIS Electronic Library AISeL, 2019. URL http://hdl.handle.net/10125/60199.

[37] Kevin Buffardi, Pedro Valdivia, and Destiny Rogers. Measuring unit test accuracy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 578–584, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5890-3. doi: 10.1145/3287324.3287351. URL http://doi.acm.org/10.1145/3287324.3287351.

[38] Margaret M. Burnett and Brad A. Myers. Future of end-user software engineering: Beyond the silos. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, page 201–211, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328654. doi: 10.1145/2593882.2593896. URL https://doi.org/10.1145/2593882.2593896.

[39] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 141–150, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3630-7. doi: 10.1145/2787622.2787710. URL http://doi.acm.org/10.1145/2787622.2787710.

[40] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. Blending measures of programming and social behavior into predictive models of student achievement in early computing courses. *ACM Trans. Comput. Educ.*, 17(3), August 2017. doi: 10.1145/3120259. URL https://doi.org/10.1145/3120259.

[41] Adam Scott Carter and Christopher David Hundhausen. Using programming process data to detect differences in students' patterns of programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 105–110, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346986. doi: 10.1145/3017680.3017785. URL https://doi.org/10.1145/3017680.3017785.

[42] J. C. Carver and N. A. Kraft. Evaluating the testing ability of senior-level computer science students. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T)*, pages 169–178, May 2011. doi: 10.1109/CSEET.2011.5876084.

[43] Henrik Bundefinedrbak Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, page 7–10, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136722. doi: 10.1145/961511.961517. URL https://doi.org/10.1145/961511.961517.

[44] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[45] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2948707. URL http://doi.acm.org/10.1145/2931037.2948707.

[46] Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5): 720–735, 2009.

[47] Lars-Ola Damm and Lars Lundberg. Quality impact of introducing component-level test automation and test-driven development. In Pekka Abrahamsson, Nathan Baddoo, Tiziana Margaria, and Richard Messnarz, editors, *Software Process Improvement*, pages 187–199, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75381-0.

[48] M. Delahaye and L. du Bousquet. A comparison of mutation analysis tools for java. In *2013 13th International Conference on Quality Software*, pages 187–195. IEEE, July 2013. doi: 10.1109/QSIC.2013.47.

[49] M. E. Delamaro, J. Offutt, and P. Ammann. Designing deletion mutation operators. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 11–20. IEEE, March 2014. doi: 10.1109/ICST.2014.12.

[50] Márcio Eduardo Delamaro, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. *Proteum/IM 2.0: An Integrated Mutation Testing Environment*, pages 91–101. Springer US, Boston, MA, 2001. ISBN 978-1-4757-5939-6. doi: 10.1007/978-1-4757-5939-6_17. URL https://doi.org/10.1007/978-1-4757-5939-6_17.

[51] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978. ISSN 0018-9162. doi: 10.1109/C-M.1978.218136.

[52] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. An extended overview of the mothra software testing environment. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151. IEEE, July 1988. doi: 10.1109/WST.1988.5369.

[53] Richard A DeMillo and Aditya P Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. *Software Engineering Research Center, Purdue University, West Lafayette, IN„ Tech. Rep. SERC-TR92-P*, 1991.

[54] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 84–93. IEEE, March 2013. doi: 10.1109/ICST.2013.20.

[55] A. Derezinska and K. Kowalski. Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 342–350, March 2011. doi: 10.1109/ICSTW.2011.54. ISSN: null.

[56] Anna Derezińska. Evaluation of deletion mutation operators in mutation testing of c# programs. In Wojciech Zamojski, Jacek Mazurkiewicz, Jarosław Sugier, Tomasz Walkowiak, and Janusz Kacprzyk, editors, *Dependability Engineering and Complex Systems*, pages 97–108, Cham, 2016. Springer International Publishing. ISBN 978-3-319-39639-2.

[57] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *SIGCSE Bull.*, 40(2):97–101, June 2008. ISSN 0097-8418. doi: 10.1145/1383602.1383644.

[58] Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3), September 2003. ISSN 1531-4278. doi: 10.1145/1029994.1029995.

[59] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, page 26–30, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137982. doi: 10.1145/971300.971312. URL https://doi.org/10.1145/971300.971312.

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

[61] Stephen H. Edwards and Zalia Shams. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 354–363, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591164. URL http://doi.acm.org/10.1145/2591062.2591164.

[62] Stephen H. Edwards, Jason Snyder, Manuel A. Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-615-1. doi: 10.1145/1584322.1584325.

[63] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. Running students' software tests against each others' code: New life for an old "gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 221–226, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1098-7. doi: 10.1145/2157136.2157202. URL http://doi.acm.org/10.1145/2157136.2157202.

[64] Sebastian Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 170. IEEE Computer Society, 2001.

[65] Mary C English and Anastasia Kitsantas. Supporting student self-regulated learning in problem-and project-based learning. *Interdisciplinary journal of problem-based learning*, 7(2):6, 2013. doi: 10.7771/1541-5015.1339.

[66] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31 (3):226–237, 2005.

[67] Hakan Erdogmus, Grigori Melnik, and Ron Jeffries. Test-driven development., 2010.

[68] Anneli Eteläpelto. Metacognition and the expertise of computer program comprehension. *Scandinavian Journal of Educational Research*, 37(3):243–254, 1993. doi: 10.1080/0031383930370305. URL https://doi.org/10.1080/0031383930370305.

[69] Katrina Falkner, Rebecca Vivian, and Nickolas J. G. Falkner. Neo-piagetian forms of reasoning in software development process construction. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering*, LATICE '13, page 31–38, USA, 2013. IEEE Computer Society. ISBN 9780769549606. doi: 10.1109/LaTiCE.2013.23. URL https://doi.org/10.1109/LaTiCE.2013.23.

[70] Katrina Falkner, Rebecca Vivian, and Nickolas J.G. Falkner. 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, 2014. Association for Computing Machinery. ISBN 9781450328333. doi: 10.1145/2591708.2591715. URL https://doi.org/10.1145/2591708.2591715.

[71] Katrina Falkner, Claudia Szabo, Rebecca Vivian, and Nickolas Falkner. Evolution of Software Development Strategies. In *Proceedings of the 37th International Conference*

on Software Engineering - Volume 2, ICSE '15, pages 243–252. IEEE Press, 2015. doi: 10.1109/ICSE.2015.153. event-place: Florence, Italy.

[72] Radio Technical Commission for Aeronautics (U.S.), Software Considerations RTCA, Inc. SC 205, RTCA (Firm). SC-205, and EUROCAE (Agency). Working Group 71. *Software Considerations in Airborne Systems and Equipment Certification*. Document: RTCA, Inc. RTCA, 2011.

[73] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988. ISSN 2326-3881. doi: 10.1109/32.6194.

[74] D. Fucci and B. Turhan. A replicated experiment on the effectiveness of test-first development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 103–112, Oct 2013. doi: 10.1109/ESEM.2013.15.

[75] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, July 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2616877.

[76] Davide Fucci, Giuseppe Scanniello, Simone Romano, Martin Shepperd, Boyce Sigweni, Fernando Uyaguari, Burak Turhan, Natalia Juristo, and Markku Oivo. An external replication on the effects of test-driven development using a multi-site blind analysis approach. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344272. doi: 10.1145/2961111.2962592. URL https://doi.org/10.1145/2961111.2962592.

[77] M. Ghafari, C. Ghezzi, and K. Rubinov. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70, Sept 2015. doi: 10.1109/SCAM.2015.7335402.

[78] Michael H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 271–275, New York, NY, USA, 2002. ACM. ISBN 1-58113-473-8. doi: 10.1145/563340.563446. URL http://doi.acm.org/10.1145/563340.563446.

[79] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975. ISSN 2326-3881. doi: 10.1109/TSE.1975.6312836.

[80] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014. doi: 10.1109/ISSRE.2014.40.

[81] Susan Hammond and David Umphress. Test driven development: the state of the practice. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 158–163. ACM, 2012.

[82] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. Exploring problem solving paths in a java programming course. In *Psychology of Programming Interest Group Conference, PPIG 2014*, pages 65 – 76, 2014. URL http://d-scholarship.pitt.edu/21832/.

[83] Liang Huang and Mike Holcombe. Empirical investigation towards the effectiveness of test first programming. *Information and Software Technology*, 51(1):182–194, 2009.

[84] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, ITICSE-WGR '15, pages 41–63, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4146-2. doi: 10.1145/2858796.2858798.

[85] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568271. URL http://doi.acm.org/10.1145/2568225.2568271.

[86] Michael S. Irwin and Stephen H. Edwards. Can mobile gaming psychology be used to improve time management on programming assignments? In *Proceedings of the ACM Conference on Global Computing Education*, CompEd '19, page 208–214, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362597. doi: 10.1145/3300115.3309517. URL https://doi.org/10.1145/3300115.3309517.

[87] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 955–963, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3340459. URL http://doi.acm.org/10.1145/3338906.3340459.

[88] David Jackson and Michelle Usher. Grading student programs using assyst. *SIGCSE Bull.*, 29(1):335–339, March 1997. ISSN 0097-8418. doi: 10.1145/268085.268210. URL https://doi.org/10.1145/268085.268210.

[89] Matthew C Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[90] Matthew C Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[91] George F Jenks. The data model concept in statistical mapping. *International yearbook of cartography*, 7:186–190, 1967.

[92] GF Jenks. Optimal data classification for choropleth maps occasional paper no 2. *University of Kansas, Department of Geography*, 1977.

[93] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sep. 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.62.

[94] P. M. Johnson, Hongbing Kou, J. M. Agustin, Qin Zhang, A. Kagawa, and T. Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: lessons learned from hackystat-uh. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, pages 136–144, Aug 2004. doi: 10.1109/ISESE.2004.1334901.

[95] Edward L. Jones. Software testing in the computer science curriculum – a holistic approach. In *Proceedings of the Australasian Conference on Computing Education*, ACSE '00, pages 153–157, New York, NY, USA, 2000. ACM. ISBN 1-58113-271-9. doi: 10.1145/359369.359392.

[96] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 612–615, Nov 2011. doi: 10.1109/ASE.2011.6100138.

[97] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635929. URL `http://doi.acm.org/10.1145/2635868.2635929`.

[98] Ayaan M. Kazerouni, Stephen H. Edwards, T. Simin Hall, and Clifford A. Shaffer. DevEventTracker: Tracking development events to assess incremental development and procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 104–109, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4704-4. doi: 10.1145/3059009.3059050.

[99] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. Quantifying incremental development practices and their relationship to procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 191–199, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4968-0. doi: 10.1145/3105726.3106180.

[100] Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. Assessing incremental testing practices and their impact on project outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 407–413, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287366. URL https://doi.org/10.1145/3287324.3287366.

[101] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 110–115, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450347044. doi: 10.1145/3059009.3059061. URL https://doi.org/10.1145/3059009.3059061.

[102] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Softw. Pract. Exper.*, 21(7):685–718, June 1991. ISSN 0038-0644. doi: 10.1002/spe.4380210704. URL http://dx.doi.org/10.1002/spe.4380210704.

[103] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156, Oct 2016. doi: 10.1109/SCAM.2016.28.

[104] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, Aug 2018. ISSN 1573-7616. doi: 10.1007/s10664-017-9582-5. URL https://doi.org/10.1007/s10664-017-9582-5.

[105] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, page 199–206, USA, 2004. IEEE Computer Society. ISBN 0780386965. doi: 10.1109/VLHCC.2004.47. URL https://doi.org/10.1109/VLHCC.2004.47.

[106] Sami Kollanus. Test-driven development-still a promising approach? In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 403–408. IEEE, 2010. doi: 10.1109/QUATIC.2010.73.

[107] B. Kurtz, P. Ammann, and J. Offutt. Static analysis of mutant subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, April 2015. doi: 10.1109/ICSTW.2015.7107454.

[108] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, March 2017. doi: 10.1109/ICST.2017.47.

[109] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mucheck: An extensible tool for mutation testing of haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 429–432, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628052. URL https://doi.org/10.1145/2610384.2628052.

[110] Timothy C. Lethbridge. Priorities for the education and training of software engineers. *Journal of Systems and Software*, 53(1):53 – 71, 2000. ISSN 0164-1212. doi: https://doi.org/10.1016/S0164-1212(00)00009-1. URL http://www.sciencedirect.com/science/article/pii/S0164121200000091.

[111] S. Levin and A. Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–46, Sep. 2017. doi: 10.1109/ICSME.2017.9.

[112] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982. doi: 10.1109/TIT.1982.1056489.

[113] Dastyni Loksa and Amy J. Ko. 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, 2016. Association for Computing Machinery. ISBN 9781450344494. doi: 10.1145/2960310.2960334. URL https://doi.org/10.1145/2960310.2960334.

[114] Dastyni Loksa, Amy J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, page 1449–1461, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450333627. doi: 10.1145/2858036.2858252. URL https://doi.org/10.1145/2858036.2858252.

[115] Z. Lubsen, A. Zaidman, and M. Pinzger. Using association rules to study the co-evolution of production test code. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 151–154, May 2009. doi: 10.1109/MSR.2009.5069493.

[116] Joseph Abraham Luke. Continuously collecting software development event data as students program. Master's thesis, Virginia Tech, 2015.

[117] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005. doi: 10.1002/stvr.308.

[118] Lech Madeyski and Łukasz Szała. The impact of test-driven development on software development productivity — an empirical study. In Pekka Abrahamsson, Nathan Baddoo, Tiziana Margaria, and Richard Messnarz, editors, *Software Process Improvement*, pages 200–211, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75381-0.

[119] C. Marsavina, D. Romano, and A. Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204, Sep. 2014. doi: 10.1109/SCAM.2014.28.

[120] Joshua Martin, Stephen H. Edwards, and Clfford A. Shaffer. The effects of procrastination interventions on programming project success. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 3–11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3630-7. doi: 10.1145/2787622.2787730.

[121] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference*, pages 604–605, Sep. 1991. doi: 10.1109/CMPSAC.1991.170248.

[122] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569. IEEE, 2003.

[123] D. McCall and M. Kölling. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8, Oct 2014. doi: 10.1109/FIE.2014.7044420.

[124] Norman A Milgram, Barry Sroloff, and Michael Rosenbaum. The procrastination of everyday life. *Journal of Research in Personality*, 22(2):197–212, 1988. doi: 10.1016/0092-6566(88)90015-3.

[125] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient javascript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 74–83, March 2013. doi: 10.1109/ICST.2013.23.

[126] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011. ISBN 978-111803196.

[127] Shinichi Nakagawa and Holger Schielzeth. A general and simple method for obtaining r2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution*, 4 (2):133–142. doi: 10.1111/j.2041-210x.2012.00261.x.

[128] Bonnie A Nardi. *A small matter of programming: perspectives on end user computing.* MIT press, 1993.

[129] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992. ISSN 1049-331X. doi: 10.1145/125489.125473. URL https://doi.org/10.1145/125489.125473.

[130] A Jefferson Offutt and Jeffrey M Voas. Subsumption of condition coverage techniques by mutation testing. *Department of Information and Software Systems Engineering, George Mason University, Tech. Rep. ISSE-TR-96-100*, 1996.

[131] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996. ISSN 1049-331X. doi: 10.1145/227607.227610. URL http://doi.acm.org/10.1145/227607.227610.

[132] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 936–946, May 2015. doi: 10.1109/ICSE.2015.103.

[133] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 354–365, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931040. URL http://doi.acm.org/10.1145/2931037.2931040.

[134] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019. ISBN 978-0-12-815121-1. doi: 10.1016/bs.adcom.2018.03.015. URL https://linkinghub.elsevier.com/retrieve/pii/S0065245818300305.

[135] Andrei Papancea, Jaime Spacco, and David Hovemeyer. An open platform for managing short programming exercises. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 47–52, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2243-0. doi: 10.1145/2493394.2493401. URL http://doi.acm.org/10.1145/2493394.2493401.

[136] Raymond Pettit and James Prather. Automated assessment tools: Too many cooks, not enough collaboration. *J. Comput. Sci. Coll.*, 32(4):113–121, April 2017. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=3055338.3079060.

[137] Raphael Pham, Stephan Kiesling, Leif Singer, and Kurt Schneider. Onboarding inexperienced developers: struggles and perceptions regarding automated testing. *Software Quality Journal*, 25(4):1239–1268, December 2017. ISSN 0963-9314, 1573-1367. doi: 10.1007/s11219-016-9333-7. URL http://link.springer.com/10.1007/s11219-016-9333-7.

[138] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of 1993 15th International Conference on Software Engineering*, pages 287–301, May 1993. doi: 10.1109/ICSE.1993.346035.

[139] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 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, 2018. Association for Computing Machinery. ISBN 9781450356282. doi: 10.1145/3230977.3230981. URL https://doi.org/10.1145/3230977.3230981.

[140] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 531–537, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287374. URL https://doi.org/10.1145/3287324.3287374.

[141] Alex Radermacher and Gursimran Walia. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 525–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318686. doi: 10.1145/2445196.2445351. URL https://doi.org/10.1145/2445196.2445351.

[142] Alex Radermacher, Gursimran Walia, and Dean Knudson. Investigating the skill gap between graduating students and industry expectations. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 291–300, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327688. doi: 10.1145/2591062.2591159. URL https://doi.org/10.1145/2591062.2591159.

[143] Yahya Rafique and Vojislav B Mišić. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2013.

[144] Kyle Reestman and Brian Dorn. Native language's effect on java compiler errors. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, page 249–257, New York, NY, USA, 2019. Association for

Computing Machinery. ISBN 9781450361859. doi: 10.1145/3291279.3339423. URL https://doi.org/10.1145/3291279.3339423.

[145] Pierre N. Robillard. The role of knowledge in software development. *Commun. ACM*, 42(1):87–92, January 1999. ISSN 0001-0782. doi: 10.1145/291469.291476. URL https://doi.org/10.1145/291469.291476.

[146] Adrian Santos, Sira Vegas, Fernando Uyaguari, Oscar Dieste, Burak Turhan, and Natalia Juristo. Increasing validity through replication: an illustrative TDD case. *Software Quality Journal*, March 2020. ISSN 0963-9314, 1573-1367. doi: 10.1007/s11219-020-09512-3. URL http://link.springer.com/10.1007/s11219-020-09512-3.

[147] David Schuler and Andreas Zeller. Javalanche: Efficient mutation testing for java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595750. URL http://doi.acm.org/10.1145/1595696.1595750.

[148] Gideon Schwarz. Estimating the dimension of a model. *Ann. Statist.*, 6(2):461–464, 03 1978. doi: 10.1214/aos/1176344136. URL https://doi.org/10.1214/aos/1176344136.

[149] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

[150] Francisco Servant and James A Jones. History slicing: assisting code-evolution tasks. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 43. ACM, 2012.

[151] Zalia Shams. *Automated Assessment of Student-written Tests Based on Defect-detection Capability*. PhD thesis, Virginia Tech, Blacksburg, VA, 2015. URL http://hdl.handle.net/10919/52024.

[152] Zalia Shams and Stephen H. Edwards. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 53–58, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2243-0. doi: 10.1145/2493394.2493402. URL http://doi.acm.org/10.1145/2493394.2493402.

[153] Zalia Shams and Stephen H. Edwards. Checked coverage and object branch coverage: New alternatives for assessing student-written tests. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 534–539, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2966-8. doi: 10.1145/2676723.2677300. URL http://doi.acm.org/10.1145/2676723.2677300.

[154] Terry Shepard, Margaret Lamb, and Diane Kelly. More testing should be taught. *Commun. ACM*, 44(6):103–108, June 2001. ISSN 0001-0782. doi: 10.1145/376134.376180.

[155] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 351–360, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368136. URL http://doi.acm.org/10.1145/1368088.1368136.

[156] Maury Silver. Procrastination. *Centerpoint*, 1974.

[157] Sivert Sørumgård. *Verification of process conformance in empirical studies of software development.* PhD thesis, Ph. D. thesis, Norwegian University of Science and Technology, 1997.

[158] Jaime Spacco and William Pugh. Helping students appreciate test-driven development (tdd). In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 907–913, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176743.

[159] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 13–17, New York, NY, USA, 2006. ACM. ISBN 1-59593-055-8. doi: 10.1145/1140124.1140131. URL http://doi.acm.org/10.1145/1140124.1140131.

[160] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. *PyDriller: Python Framework for Mining Software Repositories.* 2018. doi: 10.1145/3236024.3264598.

[161] Piers Steel. The nature of procrastination: A meta-analytic and theoretical review of quintessential self-regulatory failure. *Psychological bulletin*, 133(1):65, 2007. doi: 10.1037/0033-2909.133.1.65.

[162] Roland H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-421-8. doi: 10.1145/1566445.1566540.

[163] Sander Valstar, Sophia Krause-Levy, Alexandra Macedo, William G. Griswold, and Leo Porter. Faculty views on the goals of an undergraduate cs education and the academia-industry gap. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 577–583, New York, NY, USA, 2020. Association

for Computing Machinery. ISBN 9781450367936. doi: 10.1145/3328778.3366834. URL https://doi.org/10.1145/3328778.3366834.

[164] Marcel VJ Veenman, Jan J Elshout, and Joost Meijer. The generality vs domain-specificity of metacognitive skills in novice learning across domains. *Learning and instruction*, 7(2):187–209, 1997. doi: 10.1016/S0959-4752(96)00025-4.

[165] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2078-8. doi: 10.1145/2462476.2462501.

[166] Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. Ability-training-oriented automated assessment in introductory programming course. *Comput. Educ.*, 56(1):220–226, January 2011. ISSN 0360-1315. doi: 10.1016/j.compedu.2010.08.003. URL http://dx.doi.org/10.1016/j.compedu.2010.08.003.

[167] Christopher Watson, Frederick WB Li, and Jamie L Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Advanced learning Technologies (ICALT), 2013 IEEE 13th international conference on*, pages 319–323. IEEE, 2013.

[168] Laurie Williams, E Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 34–45. IEEE, 2003.

[169] W. Eric Wong and Aditya P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995. ISSN 0963-9314, 1573-1367. doi: 10.1007/BF00404650. URL http://link.springer.com/10.1007/BF00404650.

[170] John Wrenn and Shriram Krishnamurthi. Executable examples for programming problem comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, pages 131–139, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6185-9. doi: 10.1145/3291279.3339416. URL http://doi.acm.org/10.1145/3291279.3339416.

[171] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. Who tests the testers? In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 51–59, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5628-2. doi: 10.1145/3230977.3230999. URL http://doi.acm.org/10.1145/3230977.3230999.

[172] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 352–363, Nov 2002. doi: 10.1109/ISSRE.2002.1173287.

[173] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, Jun 2011. ISSN 1573-7616. doi: 10.1007/s10664-010-9143-7.

[174] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009. ISBN 9780123745156.

[175] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 342–353, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931038. URL http://doi.acm.org/10.1145/2931037.2931038.

[176] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL https://doi.org/10.1145/267580.267590. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[177] Barry J Zimmerman. A social cognitive view of self-regulated academic learning. *Journal of educational psychology*, 81(3):329, 1989. doi: 10.1037/0022-0663.81.3.329.

Appendices

Appendix A

Research Materials

This appendix contains the materials used for qualitative interviews with students, held during Fall semester of Fall 2016. The following materials are included:

- The consent form given to students on the first day of class in the Spring and Fall 2016 semesters.
- The script followed by the interviewers (both co-authors on [98]) during the interviews described in §4.3.1.

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Informed Consent for Participants in Research Projects Involving Human Subjects

Title of Project:

Classroom Interventions to Reduce Procrastination, and Assessing and Expanding the Impact of OpenDSA

Investigator(s):

Dr. Clifford Shaffer, Dr. Stephen Edwards, and Dr. T. Simin Hall

I. Purpose of this Research/Project

This project investigates several classroom interventions that are intended to help students taking Computer Science couses, including CS2114 and CS3114. OpenDSA is an eTextbook system intended to improve learning of course content including proficiency with a range of data structures and algorithms, and to understand algorithm analysis. OpenDSA's goal is to improve student performance on exams and similar learning outcomes measures. Other interventions attempt to address deficiencies in project management skills and procrastination on programming assignments. The overall goal for this research is to come up with strategies that measurably increase student performance on programming projects, but that can be implemented without excessive manpower costs on course staff.

II. Procedures

This study relies on examining student work and student outcomes in CS 3114 at the level of user interactions with course software, surveys, exam scores, etc. This includes user interactions in OpenDSA, and for programming projects it includes file editing activity, program execution, assignment scores, test results, cumulative scores, and final grades. Normally, this information is regularly collected as part of these courses, both from your software development environment as you work, and on the electronic grading system to which you submit your work for grading. As a result, there is nothing extra you have to do to participate. **This consent form is simply your way of giving us permission to use your coursework and scores in the research project.**

III. Risks

Participation in this research will not place you at more than minimal risk of physical or psychological harm.

IV. Benefits

We expect that this research will lead to improvements in CS education. These improvements are expected to include better strategies related to project management and learning content, so that students can achieve their full potential on their coursework. Study participants, including you, may directly benefit from the ongoing improvements to the courses that are the subject of investigation.

V. Extent of Anonymity and Confidentiality

The data we collect as part of this project will be kept strictly confidential. The data collected will not be anonymous, since we wish to examine relationships between scores throughout the course and when students start and stop working on assignments. However, all data will have your name removed and only a subject number will identify you during analyses and any written or oral reports of the results. Any personally identifiable information will be stored securely, where only the investigators can access it.

The investigators will use the collected data in publications, and they may make such data available to other researchers outside the project who are investigating the learning of programming. **In no case will personally identifiable information be divulged to any party outside the project team.**

VI. Compensation

No compensation will be provided to participants. Your choice to participate will have no effect on your course grade.

VII. Freedom to Withdraw

You are free to withdraw from this study at any time, for any reason, and without penalty.

VIII. Approval of Research

This research has been approved, as required, by the Institutional Review Board for projects involving human subjects at Virginia Polytechnic Institute and State University, and by the Department of Computer Science.

IX. Subject's Responsibilities

I voluntarily agree to participate in this study, and am not a minor (am not under 18).

X. Subject's Permission

I have read the Consent Form and conditions of this project. I have had all my questions answered. I hereby acknowledge the above and give my voluntary consent for participation in this project. If I participate, I may withdraw at any time without penalty.

____ ____________

Subject Signature Date

____

Name (Please Print)

XI. Demographic Information

A) Gender: Male Female

B) Ethnicity

1. Hispanic
2. White
3. African-American
4. Asian
5. Native Hawaiian or Other Pacific Islander
6. American Indian or Alaska Native
7. Other

C) I am a first generation college student: 1. Yes 2. No

Should I have any questions about this research or its conduct, I may contact:

Dr. Clifford Shaffer Investigator	Email: shaffer@vt.edu Phone: 540-231-4354
Dr. Stephen Edwards Investigator	Email: s.edwards@vt.edu Phone: 540-231-5723
Dr. T. Simin Hall Investigator	Email: thall57@vt.edu
David M. Moore Chair, IRB	Email: moored@vt.edu Phone: 540-231-4991

Programming Project Interview Script

Welcome

The interviewers introduce themselves.

Purpose

We are conducting an educational research project on programming project management skills.

This interview provides a way for us to learn what students actually do when they develop a programming project for a class such as CS3114. You were invited because you've experienced these projects first-hand in class, and we value your experience.

Participation in this interview is purely voluntary—you will not receive any course credit, and your comments will not affect any course grade in any way.

While we hold our discussion, I'll take notes on the comments that are made. However, I will not use any names or write down any personally identifying information for any one of you, so everything is kept anonymous.

Please read this consent form completely and decide if you wish to participate. If you agree to participate, you're giving us permission to use the comments made here as part of our research. Remember that the information will always be anonymous.

If you don't wish to participate, that is no problem—you are free to leave for any reason you choose, at any time. Also, if you are a minor, we cannot include you in our research, so please decline to participate now.

[Give time to read/sign consent forms, or for participants to elect to leave.]

Questions

1. How did you do on your first project from CS3114?
    - Did you find it to be hard? If so, why?
    - About how much time did you spend on Project 1?
    - Did you think that the difficulty of Project 1 was appropriate for this course?
    - Did you learn much from doing the project?

2. Describe to me how you went about doing the project. Please explain the design-implement-test strategy that you used. That is, tell us about things like how much you designed before you wrote code, how much code you typically wrote before you did testing and debugging. [If this is not clear in the response, ask again explicitly: How much code did you typically write before you did testing and debugging for that code?]
3. For Project 1, how many "Parts" did you break the project into, where a "part" means that you completed testing and debugging it before implementing any more code.
4. What is your primary method of doing debugging?
5. What do you think of using JUnit testing? How do you use it? Would you use it if it were not required?
6. Please explain how you developed JUnit test cases in relationship to the rest of your work. Did you write test cases along with the code? Did you do it after the code but as an integrated part of your testing strategy? Or did you write the JUnit tests only afterward so as to satisfy Web-CAT's requirements?
7. How helpful do you find the requirement to have good code coverage from your JUnit tests?
8. Generally speaking, there are two extremes to program development. One is to write all of the program, and then test and debug it. The other is "incremental development", where you write just the smallest possible working functionality, and then test and debug it before adding more functionality. To what extent do you write everything and then test/debug, versus do incremental development?
9. Did you have a partner on this project?
    - If so, describe how you worked together. Did you do things together, or did different tasks get done by each of you
    - How did having a partner go?
10. [Aside from your partner if you had one] Who did you talk to about the project?
    - Did you discuss it with classmates? If so, how much/how often? How important was that?
    - Did you go to see the TAs or Professors? If so, how much/how often? How important was that?
11. [At this point, interviewers will share the report produced by the model software with the interviewee.] We have written a program that analyzes the edit logs from your project, and then gives scores regarding how well it thinks that you followed an incremental development process. What we are really trying to do today is determine how well the software is able to estimate your use of incremental development. [Interviewers explain the report results for the interviewee.] How well do you feel that this report accurately reflects your use of incremental development for this project?
12. Of all the points that have been made during our discussion, which do you think is most important?
13. Have we missed anything?

Thank you for participating!