Modeling Student Software Testing Processes: Attitudes, Behaviors, Interventions, and Their Effects

Kevin John Buffardi

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 and Applications

June 16, 2014
Blacksburg, Virginia

Keywords: Computer Science Education, Software Testing, Test-driven development, eLearning, Adaptive Feedback

Modeling Student Software Testing Processes: Attitudes, Behaviors, Interventions, and Their Effects

Kevin John Buffardi

(ABSTRACT)

Effective software testing identifies potential bugs and helps correct them, producing more reliable and maintainable software. As software development processes have evolved, incremental testing techniques have grown in popularity, particularly with introduction of test-driven development (TDD). However, many programmers struggle to adopt TDD's "test a little, code a little" approach and conventional computer science classrooms neglect evaluating software development as a *process*. In response, we explore influences on students' testing behaviors, effects of incremental testing strategies, and describe approaches to help computer science students adopt good testing practices.

First, to understand students' perspectives and adoption of testing strategies, we investigated their attitudes toward different aspects of TDD. In addition, we observed trends in when and how thoroughly students tested their code and how these choices impacted the quality of their assignments. However, with insight into why students struggle to adopt incremental testing, we identified a need to assess their behaviors during the software development process as a departure from traditional product-oriented evaluation.

By building upon an existing automated grading system, we developed an adaptive feedback system to provide customized incentives to reinforce incremental testing behaviors while students solved programming assignments. We investigated how students react to concrete testing goals and hint reward mechanisms and found approaches for identifying testing behaviors and influencing short-term behavioral change. Moreover, we discovered how students incorporate automated feedback systems into their software development strategies.

Finally, we compared testing strategies students exhibited through analyzing five years and thousands of snapshots of students' code during development. Even when accounting for factors such as procrastinating on assignments, we found that testing early and consistently maintaining testing throughout development helps produce better quality code and tests. By applying our findings of student software development behaviors to effective testing strategies and teaching techniques, we developed a framework for adaptively scaffolding feedback to empower students to critically reflect over their code and adopt incremental testing approaches.

Dedicated to my parents,
Lou and Jean Buffardi

Acknowledgements

Foremost, I would like to thank my committee for their support and guidance. In particular, I owe a particular debt of gratitude to my adviser, Stephen Edwards. His brilliance as a software engineer and his dedication to furthering computer science education will continue to inspire me both as a computer scientist and as an educator. I cannot imagine this body of work without his valuable mentorship and collaboration. When my research in educational technology was merely a distant aspiration, Manuel Pérez-Quiñones convinced me that Virginia Tech’s Computer Science department was the best home for my doctoral studies. True to his word, he provided continual guidance and encouragement as that dream came to fruition.

I thank Eli Tilevich for his collaborative teaching and for inspiring many reflections over our roles as educators during our late-night conversations at the office. Meanwhile, Cliff Shaffer persistently challenged me with unique perspectives on computer science education and helped me reexamine my research from different vantage points. Finally, I am grateful for Shelli Fowler’s unparalleled expertise and enthusiasm for contemporary pedagogy.

I extend a special thanks to my colleagues in the Innovative Digital Education Apps and Systems (IDEAS) group: Tony Allevato, Jason Snyder, Zalia Shams, Mohammed Al-Kandari, and Ellen Boyd—I treasured their company and camaraderie. Likewise, without the support of my Blacksburg friends, my experience at Virginia Tech would have been incomplete. I also thank my students who served as a muse—and as participants—for my research.

For their roles in nurturing my passion for computer science, I also thank my teachers from high school and undergraduate studies: Linda Gattis, Karen Anewalt, Jennifer Polack, Marsha Zaidman, Ernie Ackermann, John Reynolds, and Rita D’Arcangelis. A special thanks to Peter Hastings for advising my master’s studies and for igniting my passion for eLearning and learning sciences.

Lastly, I thank my family. My sister, Anne, always reminded me that there was a light at the end of the doctoral tunnel and motivated me with her dedication and accomplishments in research. My brother, Scott, offered steadfast support and kept my academic visions of software engineering grounded with his real-world experience. Most of all, I thank my parents, Lou and Jean Buffardi—any accomplishment I achieve is inspired and empowered by their unconditional love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivations

Software development involves problem solving to design, implement, and validate complex algorithms and systems. Consequently, formal software engineering methods propose various strategies to control for quality during software development. Meanwhile, educational accreditation agencies acknowledge the need for students to gain proficiency in these methods by requiring computer science departments to prepare students with "An ability to use current techniques, skills, and tools necessary for computing practice" [ABET2013]. Likewise, a collaborative task force of leaders in industry and professional computing organizations—including the Association for Computing Machinery (ACM) and Institute of Electrical and Electronics Engineers Computer Society (IEEE-CS)—reviews and recommends guidelines for computer science curricula. Their report [ACM2013] identifies *software engineering* as a knowledge area with particular need for attention. Specifically, it recognizes a growing need to concentrate on *professional practices* in computing, with particular consideration for *testing* and *software engineering techniques*. Accordingly, computer science education should prepare students with experience following software engineering methods and techniques for improving their problem solving skills.

In addition, teaching and assessing *application* of software engineering methods and techniques is considerably more complicated than explaining the subjects and evaluating students' comprehension. Students should learn how to apply techniques in programming assignments to demonstrate *procedural knowledge* of software engineering. To the contrary, traditional evaluation of programming assignments considers only a single submission of each student's work. However, an individual submission does not indicate the *process* a student follows, only the result thereof. Thus, gathering data on how a student's work develops over time provides better insight into the process she followed. With newfound understanding of the student's software development process, one can evaluate how closely that process demonstrates adherence to software engineering conventions.

For instance, software testing is a common practice in software engineering where programmers write tests for automated verification and validation of the software's functionality. However, different software engineering paradigms can take various approaches to software testing. test-driven development (TDD), for example, is a technique whereby writing tests precedes writing production code in deliberately small, incremental steps [BECK1999]. When evaluating TDD in computer science courses, an individual deliverable of a student's program is insufficient because it, alone, cannot demonstrate an incremental testing process (or lack thereof). While a single submission may indicate the *result* or outcome of a student's testing and development, multiple snapshots are necessary to assess how well he adhered to TDD as a *process*.

In addition, adherence to software development methods is not typically measured on an absolute scale. That is, students may adopt different aspects or techniques involved in the intended method to varying degrees. Consequently, students may demonstrate various adaptations of a formal method. For example, one student may strictly adhere to TDD's test-first, incremental process while a second student works in larger increments and writes tests after her code while a third student disregards TDD and tests only sporadically throughout development. Evaluation of only their finalized work may not draw any distinctions between the first two students. However, process-oriented analysis of each of the three students' development can distinguish behaviors with varying degrees of adherence to multiple dimensions of TDD. Furthermore, by identifying distinct testing behaviors, we can then compare their effects on the outcomes of students' work. As a result, analyzing students' development processes can provide insight into individuals' adherence to software engineering methods, common variations of those conventions, as well as evaluations and comparisons between different techniques.

In addition to improving assessment of development methods and student outcomes, eLearning tools can use process-oriented analysis to help students learn while working on assignments. While collecting periodic snapshots of students' work, eLearning technology can utilize automated approaches to collecting and persistently analyzing data about students' code. For example, for each snapshot of work, an eLearning tool could use quantitative measurements of the code to measure adherence to a software development method. As a result, the tool could give students individualized feedback providing insights into evaluation of their work. Moreover, with insight into students' work-in-progress, the tool has the opportunity to use educational interventions to help the students learn, improve their work, and follow methods. Opposed to only providing feedback on learning objectives *after* an assignment is complete, continual process-oriented analysis and feedback can help students *while* working on each assignment.

1.2 Research Goals

This work addresses the primary research question: **how can computer science education use eLearning tools to assess and influence students' software testing behaviors?** To attend to this question, we take a process-oriented approach to investigating students' adherence to test-driven development as a formal software development method. Consequently, this work concentrates on three main objectives in addressing the primary research question:

1. **Describe student affect (emotions and valence) and opinions with regard to their influence on adherence to test-driven development (TDD).**

2. **Design an eLearning intervention for encouraging TDD adherence and evaluate its impact on student affect, behaviors, and outcomes.**

3. **Characterize software testing strategies demonstrated by students and evaluate their consequential outcomes.**

For a comprehensive synopsis of students' experience learning TDD in higher education, we examine their attitudes about and adherence to TDD (*chapter 3*). Furthermore, this work describes an eLearning tool—motivated by pedagogical principles—that assesses students' testing behaviors and provides adaptive feedback to reinforce incremental testing. We evaluate the effects of students' attitudes and behaviors on their TDD adherence and the consequential outcomes of their work process. Likewise, we evaluate our adaptive eLearning intervention for its influence on students' attitudes and behaviors (*chapter 4*). Finally, we discuss effective and ineffective testing strategies backed by our comprehensive quantitative research and provide insights to designing adaptive eLearning tools to support procedural learning (*chapters 5 & 6*).

1.3 Organization and Contributions of This Work

In this section, we describe the remaining chapters. Following a literature review, chapters 3, 4, and 5 describe completed research as it addresses three primary objectives: (i) describing students' emotions, attitudes, and outcomes of their adherence to test-driven development (TDD), (ii) detailing an eLearning intervention and evaluating its influence on student behavior, and (iii) characterizing software testing strategies and their associated consequences.

Chapter 2

In the following chapter, we outline principles of test-driven development and reasons for teaching TDD in computer science higher education curricula. We survey implementations and findings from studies of TDD use in industry. Likewise, we synthesize these conclusions with those from existing research on TDD in education and identify remaining issues to research. Additionally, this chapter outlines pedagogy and learning science with the purpose of reviewing existing eLearning tools and motivating design of an adaptive feedback system.

Chapter 3

Chapter 3 describes a study of influences on student adherence of TDD. In particular, the study uses quantitative instruments for measuring students' attitudes toward individual principles of TDD as well as establishing a baseline measurement of anxiety while students develop programming assignments. In addition, we propose metrics for analyzing adherence to incremental testing and provide a preliminary investigation of relationships between these metrics and outcomes of software quality. Finally, we discuss findings from this study and their implications for designing educational interventions to encourage TDD.

Chapter 4

In the next chapter, we describe how pedagogy motivated our design of an adaptive feedback system with the aim to reinforce incremental testing behaviors. We describe three experiments for investigating the influence on the adaptive feedback system on student testing behavior. The first study compares student behaviors and outcomes from two semesters: a control semester, where students did not use the automated feedback system, and an experimental semester where they did. Next, the second study investigates the impact of different reinforcement strategies implemented in multiple treatments of the adaptive feedback system. Finally, the third study concentrates on short-term responses students exhibit after receiving rewards. In addition to reporting findings from both studies, we discuss findings from interviewing students about their software development processes and experiences with the eLearning tool.

Chapter 5

Chapter 5 describes a comprehensive investigation into identifying testing strategies that students exhibit on programming assignments. We outline a study of testing strategies identified using process-oriented analysis of a large-scale data set of snapshots of programming assignment development. In this chapter, we characterize effective and ineffective testing strategies and explain how they represent different adaptations of (and varied adherence to) TDD principles. Finally, we describe an approach for scaffolding feedback to students by promoting reflection through software testing.

Chapter 6

The final chapter of this document summarizes the contributions of the findings in this dissertation and identifies avenues for future work.

Chapter 2

Literature Review

In this chapter, we provide background and context for our research in two sections. The first section (§2.1) provides an overview of test-driven development (TDD). We describe TDD's purpose, procedure, application, and findings from its use in both industry and academia. In Section 2.2, we summarize pedagogical principles and explain their application in a survey of relevant tools for computer science education (and TDD in particular).

2.1 Test-Driven Development (TDD)

Kent Beck first introduced test-driven development (TDD) as an integral part of the larger eXtreme Programming (XP) software development method [Beck1999]. He describes:

> Development is driven by tests. You test first, then code. Until all the tests run, you aren't done. When all the tests run, and you can't think of any more tests that would break, you are done adding functionality.

This approach entails two inherent practices. Firstly, developers write tests in small parts. This practice describes *unit testing*. Unit testing involves writing each test with a focus on the smallest segment of code, or in most cases, testing a method or function. The purpose of a unit test is to verify that a specific method works correctly. When a unit test fails, the developer should know immediately what is broken in the code. Conversely, when a unit test passes, she should be confident that the code works as expected.

TDD's second inherent practice entails developing with a *test-first* approach. That is, writing a unit test should precede writing the corresponding method. This test-first approach represents a notable shift from previous testing conventions, where testing often followed writing code with a *test-last* approach. However, since developers write unit tests using testing frameworks—such as JUnit [JUni2013] for testing Java—unit tests can run automatically and provide developers with immediate feedback on whether their code works.

By using unit testing with a test-first approach, TDD specifies an incremental, “test a little, code a little” approach [Beck2003]. Specifically, developers should: (i) identify a test case and write it as a unit test, (ii) write the corresponding code, (iii) run all unit tests, and then (iv) correct the code if tests fail or clean up the code and repeat from the beginning (with new features) if all tests pass. Consequentially, the incremental, test-first approach resembles the iterative cycle represented in **Figure 2.1:**

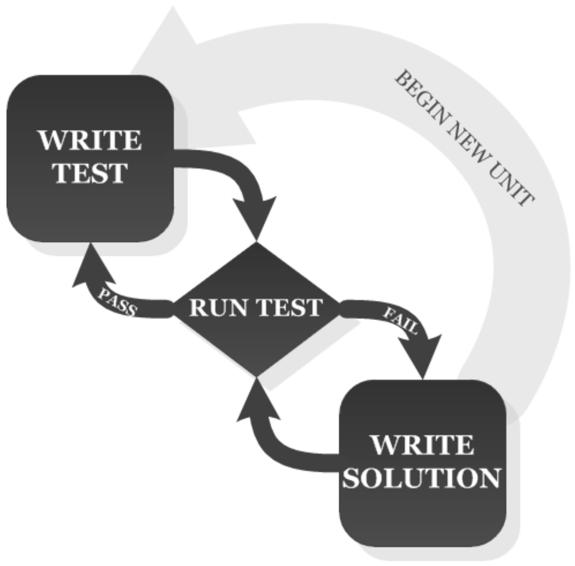


Figure 2.1. An Iteration of test-driven development's Test-First Approach.

Philosophically, following TDD should promote confidence and maintainability of code. With immediate feedback after writing solution code, developers should gain confidence that their code performs according to their expectations. In addition, unit testing advocates that each test is independent and has one specific purpose. Consequentially, when refactoring code or implementing new features, one can identify if the changes broke existing functionality by executing the unit tests. A failed test should identify specifically what broke, which enables the developer to focus on what to fix or revert to previous versions of the code until all the tests pass. Accordingly, Beck suggests that TDD should assist in refactoring, improve productivity, and consequentially reduce defects in code [Beck2003].

The potential for developing software with greater confidence and improved quality is appealing. As a result, several leading software manufacturers have adopted TDD as a key ingredient to the Agile software development methodology [FAB+2003]. However, a mix of reports from use of TDD in industry presents both enthusiasm and caution for adopting TDD.

Bhat and Nagappan [BN2006] provide a generally favorable perspective of TDD from two case studies from Microsoft. They warned that the overall development time for TDD projects took 25-35% longer. However, they also found that projects using TDD produced higher quality code, with less than half the defects-per-thousand-lines-of-code (defects/KLOC) than comparable projects that did not use TDD. They also acclaimed that TDD's unit tests documented code specifications well.

Canfora, et al. [CCG+2006] conducted a controlled experiment using software professionals to compare TDD to the "test after coding (TAC)," test-last approach. Similarly, they found that TDD is more time consuming. Nevertheless, they suggest that productivity is not necessarily hurt since TDD improves unit testing. Moreover, they also found that the workload involved is more predictable when using TDD since incremental testing sets clearer expectations than TAC, which can vary more depending on the developer's will and care.

Meanwhile, Sanchez, et al. [SWM2007] performed post-hoc longitudinal analysis on five years of developing an IBM product while using TDD. They compared the results to another product's development that did not use TDD and concluded that TDD aided the quality of the software. Like the previous studies, they found TDD requires more time. However, they suggest that TDD produced code with fewer defects. While the cyclomatic complexity tends to increase over time as software grows and matures, they suggest that TDD may decrease the rate at which it becomes more complex. Consequentially, they concluded that the improved quality of the software compensated for the initial impressions of less productivity when using TDD.

Nevertheless, in a comprehensive meta-analysis of 22 reports of 32 unique clinical trials, Turhan et al. [TLD2010] hedge possible benefits of TDD with reports of inconsistent effects and possible side effects. They evaluated TDD on external quality (regarding defects and effort correcting defects), internal quality (regarding simplicity and maintainability of designs), productivity, and test quality. For each category, they noted mixed results and drew conclusions from trends between studies.

They found that TDD only benefitted external quality if considering pilot and other less-rigorous studies. Otherwise, there was no consensus on the effect on external quality. While considering internal quality, they found that TDD may benefit some aspects (such as the previously mentioned effect on complexity) but detriment others (such as design cohesion). Likewise, they found that reports on productivity were inconsistent. Generally, TDD decreases productivity initially because of a steep learning curve but it is not clear whether the long-term effect is positive, negative, or neutral. To the contrary, Turhan et al. found that TDD's emphasis on incremental testing tends to yield better (or at least no worse) quality tests. However, they noted that most of the positive reports came from pilot studies and that more rigorous experiments found no significant difference in test quality when following TDD.

Consequentially, they offer a perspective that tempers expectations for TDD's effectiveness. In doing so, they posit that TDD may be appropriate in some contexts but not in others. Moreover, they recognize that TDD is not used in a vacuum and that other factors—such as pair versus solo programming, design paradigms, or developing for specific domains like embedded or distributed systems—may determine TDD's impact. Furthermore, they suggest:

> […] TDD is difficult to learn. It involves a steep learning curve that requires skill, maturity, and time, particularly when developers are entrenched in the code-then-test paradigm. Better tool support for test-case generation and early exposure in the classroom to a test-then-code mentality may encourage TDD adoption.

Finally, they also recognize that most research does not strictly enforce TDD. That is, the experiments usually involved directing (after sometimes teaching) the developers to use TDD, but did not actively enforce its adoption. Consequently, we do not know how closely study subjects adhered to the strict definition of TDD.

Nevertheless, given TDD's popularity in industry, some computer science programs include TDD in their curricula. However, academia faces unique challenges in teaching TDD. Primarily, students in computer science courses often work on software projects with smaller scope and duration. In addition, for individualized assessment, programming assignments often require students to work entirely independent of each other instead of as part of larger teams. Potentially increasing students' workload on programming assignments can be problematic because of the short timespan they have to complete an assignment – adding undue workload may be burdensome to students' studies. Finally, some reports caution that TDD may be difficult for novices, which would include most students.

Edwards, [Edwa2003] introduced TDD to computer science classes along with an automated grading system that included assessment of students' test quality. At the conclusion of multiple semesters of a junior-level programming languages course, he surveyed students and found general enthusiasm about using TDD. Students produced programs with 28% fewer defects/KLOC. Survey responses showed that generally students believed TDD improved their confidence and ability to write tests, but also had a moderately poor opinion of TDD's impact on how long assignments took.

Melnik and Maurer [MM2005] evaluated TDD, as used in Agile development, over three academic years. By surveying both graduate and undergraduate students after experiencing TDD in class, students reported positive perspectives of TDD's impact on software quality, speed of testing, and software design. However, only a small majority (55%) reported that they used TDD on a team assignment. Consequently, some students' beliefs may reflect what they were taught in lectures regarding TDD rather than appraising it from their first-hand experiences.

Janzen and Saiedian [JS2008] compared test-first to test-after-coding incremental unit testing. They found that test-first programmers consistently wrote more concise (fewer lines of code) and simpler (cyclomatic complexity) code. They did not find conclusive differences in the coupling or cohesion. In addition, they identified an issue of less-experienced students testing after coding even when instructed to follow a test-first approach.

In another study [JS2007], they also reported that students' experience as developers and amount of exposure to TDD influenced their willingness to adopt TDD. Likewise, Barriocanal, et al. [BUM+2002] identified a need to motivate students to write tests when they are first introduced to TDD. Spacco and Pugh [SP2006] specifically suggest that students require incentives to adopt a “test-first mentality.” They warned that otherwise, students resort to test-after-coding approaches.

As anticipated, concerns about students’ willingness to adopt TDD are prevalent; however, learning TDD has multiple benefits. Foremost, with exposure to TDD, students gain experience with professional techniques. Secondly, some reports suggest that students’ work will consequently improve with better tests and code with fewer defects. Finally, following TDD’s test-first approach may promote improved problem solving.

Edwards [Edwa2004] suggests that by testing their own code, students may be more likely to experiment and reflect over their solutions. Consequently, students may abandon weak “trial-and-error” problem solving strategies by thinking analytically and exercising “reflection-in-action.” Therefore, in addition to TDD’s practical applications, Edwards advocates that it may also have pedagogical advantages by encouraging reflection and better problem solving strategies.

2.2 Applications of Pedagogy in eLearning Contexts

A loose definition of eLearning tools includes any digital technology used in an educational setting. Such a broad definition encompasses a wide variety of tools from presentation software and word processors to educational games and intelligent tutoring systems. However, by narrowing the scope of the definition to include only tools created with the *specific* purpose of supporting students’ learning, pedagogy deliberately motivates use and implementation of eLearning tools. Accordingly, popular pedagogical philosophies and empirically vetted teaching techniques provide a foundation for designing and evaluating eLearning technology [CM2003].

Learning Taxonomies

Bloom’s Taxonomy [Bloom1969] classified three principle domains of learning: cognitive, affective, and psychomotor. The psychomotor domain relates to physical skills, which are not particularly relevant to fundamental computer science education. On the other hand, the cognitive domain addresses how students construct knowledge while the affective domain concentrates on their emotions, attitudes, and motivations. Both the cognitive and affective domains can help frame what students comprehend and how they learn and adopt new skills in computer science.

An updated revision of Bloom’s Taxonomy [AKA+2001] classifies the cognitive domain as levels from lower- to higher-order thinking skills: remember, understand, apply, analyze, evaluate, and create. *Remembering* involves identifying and retrieving information from memory. *Understanding* includes explaining, interpreting, comparing and other methods of constructing meaning. *Applying* is the use of procedures to undertake tasks. Together, remembering, understanding, and applying represent lower-order thinking skills (LOTS). Meanwhile, analyzing, evaluating, and creating represent higher-order thinking skills (HOTS), which demonstrate critical thought. For example, *analyzing* often involves dissecting material into parts and considering their relationships and purposes. *Evaluating* incorporates checking and making judgments on material and finally *creation* forms new structures from distinct parts through hypothesis, design, and construction. Computer science education should combine both LOTS and HOTS by enabling students to perform simple tasks like identifying and implementing common programming conventions as well as preparing them to demonstrate complex cognition such as deconstructing complicated problems and constructing novel solutions.

Meanwhile, the affective domain of Bloom's Taxonomy presents a context for framing students' feelings and motivations for learning. It includes five levels of affective processes: receiving, responding, valuing, organizing, and characterizing. While *receiving* only reveals a passive attention to learning, *responding* demonstrates participation as a learner. Going further, by *valuing*, students assign worth to behaviors and phenomena. Moreover, students demonstrate *organization* of those values by differentiating them from other values and adapting them. Finally, students exemplify *characterizing* a value by internalizing it so that it directs their independent behavior. The affective domain is particularly important in explaining motivations behind students' behaviors. Consequently, assessing student affect helps evaluate their adherence to software development methods, such as test-driven development (TDD). The following table outlines each level of the cognitive and affective domains with corresponding learning objectives for TDD:

Table 2.1. Learning Objectives for test-driven development (TDD) as Classified by Bloom's Taxonomy

Cognitive Process	TDD Learning Objective
Create	Develop personal problem solving strategy using TDD
Evaluate	Appraise TDD's effectiveness in software development
Analyze	Review TDD's outcomes compared to other approaches
Apply	Demonstrate TDD behaviors when developing software
Understand	Generalize TDD's approach as development method
Remember	Recall test-first and incremental approach to TDD

Affective Process	TDD Learning Objective
Characterize	Adopt TDD as a personal problem solving strategy
Organize	Differentiate advantages of TDD vs. of other strategies
Value	Recognize benefits of TDD on software development
Respond	Attempt approaches described in TDD
Receive	Read/listen to instructions on how to follow TDD

Tomei also proposes a technology domain [Tome2005] to supplement the traditional domains in Bloom's Taxonomy. In particular, he defines a six-tiered technology taxonomy: *literacy* for understanding technology, *collaboration* for sharing ideas through mediated interpersonal interaction, *decision-making* for leveraging technology to problem-solve, *infusion* for identifying existing technology to apply to learning, *integration* for developing new technology to aid learning, and *tech-ology* for judging technology's social impacts.

Put broadly, computer science education implicitly addresses each of these levels since technology is inseparable from the study of computing. However, to support learning of software development methods, learning objectives must particularly emphasize *decision-making*, *infusion*, and *integration*. As we introduce systems for providing students with feedback, that technology should influence students' *decision-making* in adopting effective problem-solving techniques. Likewise, we need to *infuse* existing educational and software engineering tools by adapting and *integrating* them into interventions specific to incorporating TDD in software development. Accordingly, while Tomei's Taxonomy helps characterize *tangible application* of technology as educational media, Bloom's Taxonomy emphasizes the psychological aspects of learning.

Scaffolding

In addition to gauging students' cognition and affect, educational interventions often aim to guide students' knowledge and behaviors to meet specific learning objectives. In particular, educational constructivism emphasizes generating new knowledge based on existing knowledge. For example, Vygotsky defines the *Zone of Proximal Development* (ZPD) [Vygo1978] as:

> …the distance between the actual development level as determined by independent problem solving and the level of potential development as determined through problem solving under (instructor) guidance, or in collaboration with more capable peers.

Consequentially, educational interventions may be most effective when they gauge the student's actual development level, predict what the student can achieve with help and provide just enough guidance until the student can demonstrate that development level without assistance. *Scaffolding* describes a similar approach where the teacher gradually withdraws guidance as the student attains higher goals and greater independence. Scaffolding can serve as a general teaching philosophy, but eLearning implementations of scaffolding show particular potential [Bran2000]. For example, intelligent tutoring systems (ITS) use cognitive models to track student comprehension and adaptively help them learn, such as when solving algebra problems using the ITS, Practical Algebra Tutor [KS1996].

To bridge cognition with technological tools and interventions, Kaptelinin [Kapt1995] describes a*ctivity theory* as an extension of Vygotsky's work for contextualizing human-computer interaction. Put broadly, activity theory explains acquisition of new knowledge and abilities as the process of mentally internalizing interactions. Meanwhile, externalization represents manifesting mental processes through external actions. Since external actions are directly observable while internal processes are not, mental processes are verified through behavioral observation. Consequently, correcting mental processes first requires observation of an incorrect action, followed by mediation that shapes new interactions to internalize. Accordingly, Kaptelinin describes computers as tools for mediation within the framework of activity theory.

Applied eLearning Techniques

Promoting learning through human-computer interaction is challenging. Moreover, mediating behavioral change through cognitive and affective development poses particularly unique challenges in eLearning. However, Fogg introduces principles of *captology*, the "design, research, and analysis of interactive computing products created for the purpose of changing people's attitudes or behaviors" [Fogg2003]. In particular, he identifies distinctions computers have from humans that give computers unique advantages in persuading change. For example, while humans may become fatigued or frustrated in persuading others, computers can persistently provide interventions without giving up. In addition, computers can take advantage of massive data storage with quick calculation to store and access relevant information. Moreover, computers can also scale those capabilities well to larger groups of users (e.g. many students) while human instructors' one-on-one guidance does not scale to large groups of learners.

Fogg also outlines several types of empirically vetted, persuasive techniques that can apply—either alone or in conjunction—to shaping complex behaviors. *Reduction* involves simplifying target behaviors to simple, clear steps for the user to follow. *Tunneling* requires a user to relinquish some self-determination and self-regulation by using technology that dictates a sequence of steps. Meanwhile, *tailoring* takes an alternate approach by adapting relevant information to match the user's state (such as comprehension level or attitude) to the target behavior. Similarly, *suggestion* technology persuades behavior by providing recommendations for target behavior. However, Fogg emphasizes that effective *suggestion* requires keen use of kairos—identifying an opportune time to deliver the message.

As an example, Fogg illustrates that technology meant to persuade drivers to carpool requires the suggestion to precede leaving to commute for immediate behavior change. Alternately, in this context, persuasive technology could interject while the driver is stuck in traffic while a carpool lane is empty to influence the driver's attitude and future decisions. However, he also points out that studies suggest that people are persuaded more often when they can take immediate action on the decision.

Fogg also describes how *self-monitoring* technology can influence behavior by empowering people to deliberately track their progress toward a goal and develop intrinsic motivation. Alternately, incentive systems use *surveillance* to monitor peoples' behaviors and then offer rewards to reinforce target behaviors. *Conditioning* technology specifically uses positive and/or negative reinforcement—often in the form of sounds and images as digital rewards or punishments—to encourage or discourage associated behaviors. However, influencing complex behaviors is difficult so designing reinforcement needs to address each aspect of the behavior strategically.

Linehan, et al. [LKL+2011] describe a framework of reinforcement schedules and how they relate to prompting different types of behaviors. As alternatives to constant reinforcement, *ratio* or *interval* schedules can establish intermittent reinforcement. More specifically, either schedule can also be *fixed* or *variable*. A *fixed ratio* schedule offers reinforcement after the target behavior is demonstrated a *fixed* number of times while a *variable ratio* schedule changes the number of times a behavior has to be demonstrated between each reinforcement. Both schedules produce high and steady rates of response while the latter may be more economical by using fewer instances of reinforcement. On the other hand, *fixed interval* schedules only reinforce a behavior after a given amount of time has elapsed, while *variable interval* oscillates the amount of elapsed time required. *Variable interval* scheduling is particularly difficult to predict and consequently, students may be more surprised by unanticipated reinforcement. All four schedules are used regularly in educational games.

In addition to strategies for scheduling reinforcement, research has also identified tactics for engaging students by leveraging motivations from games. *Gamification* is the technique of applying basic elements of games—such as point systems, leaderboards and badges—to promote engagement outside of gaming contexts [DSN+2011]. For instance, Mekler, et al. [MBO+2013] found that establishing a point system motivated increased intrinsic motivation to perform simple but otherwise mundane tasks.

However, Bruckman [Bruck1999] also warns that many attempts to merge education with entertainment fail. She uses "chocolate-dipped broccoli" as a metaphor to explain how educational games sometimes adopt the less-favorable aspects of education and entertainment without effectively delivering their respective *appealing* aspects. That is, without proper game and educational design, "edutainment" may neither be fun nor promote learning. Instead, she advises that following some basic principles help establish engagement in educational contexts, such as: "Make the learning inherently fun—don't sugar-coat an unpleasant educational core" and "Whenever possible, provide social support for learning."

Software Testing Tools for eLearning

There are several tools for supporting test-driven development (TDD), some of which are designed specifically for educational purposes. However, there are currently no eLearning systems that use persuasive technology, gamification, or other overt, pedagogically-driven techniques to assess and encourage TDD as a software development process. Instead, we describe the following technologies to identify their assets as well as opportunities for advancing TDD eLearning tools.

JUnit [JUni2013] is a testing framework for the Java programming language and similar (xUnit) frameworks exist for other popular languages. In addition, TDD extensions for Integrated Development Environments (IDE's) are common to execute test cases in JUnit and automatically generate reports on whether each test passed or failed. While the TDD extensions visually identify passes and failures (usually with green and red progress bars), they do not require specific testing behaviors and do not deliberately persuade students' testing behaviors.

Alternatively, Lappalainen, et al.[LII+2010] developed ComTest, a system to allow students to write test cases in a simplified format. While they found that ComTest was popular among novices and promoted students to write more tests, using ComTest diverts students from gaining practical experience using JUnit, which is the *de facto* standard for unit testing in industry. Furthermore, no notable research suggests that JUnit's syntax is a significant obstacle to students adopting TDD. Finally, Lappalainen, et al., also acknowledge that ComTest requires unit testing but does not enforce either test-first nor incremental approaches to testing (as required in TDD).

Meanwhile, automated testing tools, such as Clover [Atla2013], provide more insight into tests. Specifically, it calculates test *coverage*, or the percentage of the code that is exercised by executing all of the tests. Consequently, students can use these tools to appraise how thoroughly their tests exercise their code. Similarly to JUnit extensions for IDE's, these tools often produce visual indications to help monitor coverage, but neither enforce or suggest particular testing methodology.

On the other hand, Kou, et al. [KJE2010] developed Zorro, a tool for automatically inferring whether software development done within the IDE exhibits TDD. It records operations in the IDE such as creating a test method, creating solution methods, running tests, and refactoring code. Based on the order and pattern of these "micro-processes," Zorro determines if the development behavior matches the operational definition of TDD. In preliminary pilot studies, they found that Zorro inferred TDD adherence with up to 90% accuracy (albeit with very small sample size). Kou, et al. also recognize some issues and limitations to the tool's inferences.

For example, Zorro's inference may be inaccurate as a result of misleading behaviors. For instance, if a student creates the signature for a test method but leaves the body of the method blank (with no actual testing operations) and then completes the corresponding code before returning and writing its tests. This behavior does not exhibit a test-first approach, but may be inferred as TDD adherence because the test method creation preceded that of the solution code. As they continue to develop and evaluate Zorro, it shows potential for supporting learning TDD. In particular, by using Zorro as *self-monitoring* or *surveillance persuasive technology* [Fogg2003], it could provide students with indications for tracking their adherence to TDD and/or offer positive reinforcement to overtly encourage adherence.

Meanwhile, automated programming assessment tools can provide students with informative *self-monitoring* and track their testing behaviors over time. Both Marmoset [Spac2013] and Web-CAT [Edwa2013] allow students to submit their work—both code and tests—for automatic evaluation. Both tools provide prompt evaluation of the code's correctness (as determined by its performance against a set of instructor's tests) and allow the student to make revisions and resubmit. Web-CAT particularly emphasizes test evaluation as well by providing aforementioned coverage assessment. Consequently, students can analyze the results of the automated evaluation to identify ways to improve their code and tests. However, the automated assessment tools do not overtly evaluate or encourage particular *processes* of testing. Nevertheless, by leveraging the pedagogical approaches outlined in this chapter, we can adapt Web-CAT to analyze students' behaviors (as indicated by their testing across multiple submissions to the system), use eLearning techniques to influence students' cognition and affect, and persuade TDD behaviors.

Chapter 3

Student Affect, Attitudes, and Testing Behaviors

This chapter depicts two studies we conducted as initial investigations into students' attitudes about and adherence to test-driven development (TDD). Specifically, we concentrate on the first objective of the research goals (see section 1.2): "**Describe student affect (emotions and valence) and opinions with regard to their influence on adherence to test-driven development (TDD).**" The first study (*3.1 Exploring Influences*) explores measurements of students' feelings and attitudes and their implications for adhering to TDD. It also provides a baseline for assessing student anxiety when working on programming assignments. The second study (*3.2 Impacts of Teaching test-driven development*) investigates metrics for observing adherence to TDD and studies relationships between those metrics and consequential outcomes. Overall, the research in this chapter provides insight into students' experiences as context for planning TDD assessment and teaching interventions.

3.1 Exploring Influences

We have published the work described in this section in the article, "Exploring influences on student adherence to test-driven development" and presented it at the conference on Innovation and Technology in Computer Science Education (ITiCSE) [BE2012ii].

3.1.1 Background

At Virginia Tech, we teach test-driven development (TDD) with Java in our introductory programming courses. TDD is introduced early using the JUnit framework [JUni2013] and is reinforced throughout the course. Programming project grades include assessment on design as well as on the comprehensiveness of student-written tests. However, project grades only evaluate individual, completed submissions. They cannot definitively reveal whether or not students followed a test-first approach when writing their solutions.

Grading test code in programming projects may offer insight to students' abilities to produce adequate tests. However, by incorporating TDD into the computer science curriculum, we intend to not only expose students to testing, but to also prepare them in software development methods applied in industry. Therefore, we are interested in both the software they produce and the processes to which they adhere.

In our CS2 course, Software Design and Data Structures, students submit their code to Web-CAT [Edwa2013], an on-line automated grading tool that provides students feedback including static analysis, test coverage, and solution correctness evaluated by undisclosed instructor-written reference tests. Students may submit their programming projects as many times as they wish without penalty and only their final submission is considered for grading.

By allowing students to submit their work-in-progress, we also gain access to snapshots of students' progress as they develop their software. These series of snapshots provide some insight into students' behaviors and the development methods they follow. Consequently, this study addresses student adherence to TDD and investigates reasons for their adherence or lack thereof.

3.1.2 Method

In order to understand student behavior and affect—including their attitudes and emotional valence—with regards to adhering to TDD, we collected data from a semester of CS2 assignments via Web-CAT. We collected both project submissions and attitudinal surveys. Participation was voluntary and no compensation was offered. Consent was acquired for Web-CAT assignment data analysis and for survey analysis separately.

The survey consisted of 5-point interval scale items. First, students were asked to "Rate each item individually on the 5-point scale from 1 (Very Unimportant) to 5 (Very Important) on *how important the skill is in Computer Science*" for: time management, problem solving, attention to detail, writing solution code, and writing test code. They then repeated ratings for the same skills according to "*how strong you are in the skill*" from "Very Poor" to "Very Good."

Next, students rated "*the impact of the following behaviors have on developing programs*" from "Very Harmful" to "Very Helpful," for the following behaviors: beginning work *as soon as it is assigned*, beginning work *near its deadline*, developing thorough test code, developing code and corresponding tests in *small units* at a time, developing code and corresponding tests in *large portions* at a time, developing tests *before* writing solution code, and developing tests *after* writing solution code. Correspondingly, they also rated these behaviors according to "*how often you practice the behavior*" from "Very Rarely" to "Very Often."

Students then rated their agreement with the following statements from "Strongly Disagree" to "Strongly Agree," "*based on your experience with test-driven development (TDD)* by incrementally developing tests and then solution code one unit at a time:" *I consistently followed TDD in my programming projects during this course*; *TDD helped me write better test code*; *TDD helped me write better solution code*; *TDD helped me better design my programs*; and *In the future, I will choose to follow TDD when developing programs outside of this course*.

On the same scale, the students then rated the following items "based on *your experience with Web-CAT automated results* (NOT TA/instructor feedback):" *Web-CAT helped me improve writing test code*; *Web-CAT helped me improve writing solution code*; *Web-CAT helped me improve designing my programs*; *Web-CAT helped me follow test-driven* *development*; *Web-CAT helped improve my time management*; and *Web-CAT helped improve my attention to detail*.

To explore relationships between psychological affect and TDD and instructional technology interaction, the survey included the Brief Fear of Negative Evaluation Scale (BFNES) [RWT+2004] and the Westside Test Anxiety Scale (WTAS) [Dris2007]. Both are brief (5-point scale, 8- and 10-item, respectively), validated questionnaires. The adaptation of WTAS specifically addressed working on programming projects with the following items:

- *The closer I am to a programming project deadline, the harder it is for me to concentrate on it*
- *When I prepare for programming projects, I worry I will not understand the necessary material*
- *While working on programming projects, I think that I am doing awful or that I may fail*
- *I lose focus on programming projects, and I cannot understand material that I knew before the project*
- *I finally understand solutions to programming projects after the deadline passes*
- *I worry so much before a programming project deadline that I am too worn out to do my best on the project*
- *I feel out of sorts or not really myself when I work on programming projects*
- *I find that my mind sometimes wanders when I am working on important programming projects*
- *After the project deadline, I worry about whether I did well enough*
- *I struggle with developing programming projects, or avoid them as long as I can. I feel that what I do will not be good enough*.

Statistical analysis concentrated on evaluating the following hypotheses:

(1) Students will rate importance of skills and their corresponding strengths with a positive correlation

(2) Students will rate helpfulness of and their adherence to behaviors with a positive correlation

(3) Students more likely to adhere to TDD principles will rate TDD’s helpfulness more positively

(4) Students with higher programming anxiety (according to WTAS) will adhere less to starting work early and to principles of TDD

(5) Students with higher programming anxiety will rate Web-CAT as more helpful

(6) Students with higher evaluation anxiety (according to BFNES) will rate Web-CAT as less helpful.

See **Appendix A** for the full survey. All participants were students in CS2 in the Fall 2011 semester. 87 students consented (66% of 131 enrolled) to release their project data on Web-CAT for analysis and 61 students (47%) voluntarily completed the survey. Of the survey respondents, 84% were computer science majors and others either belonged to a different department or had no declared major. Before the course began, 48% of respondents had previously used Web-CAT, 61% had previously written test code, and 52% had previously followed TDD.

3.1.3 Results

Ratings on skill strength and importance to computer science were generally positive for each skill. The mean and standard deviation for each item is shown in **Figure 3.1**, below. Analysis of paired ratings using Pearson's product-moment correlation coefficient did not reveal any significant relationships except for a small, positive correlation between the importance of writing test code and personal strength at it (r^2=0.26, p=0.02).

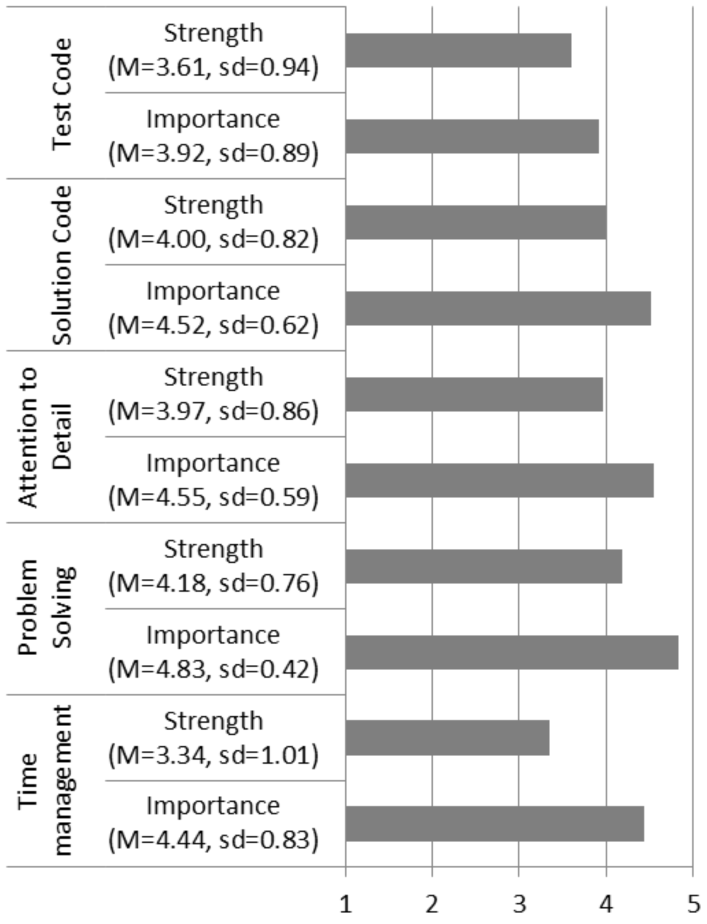
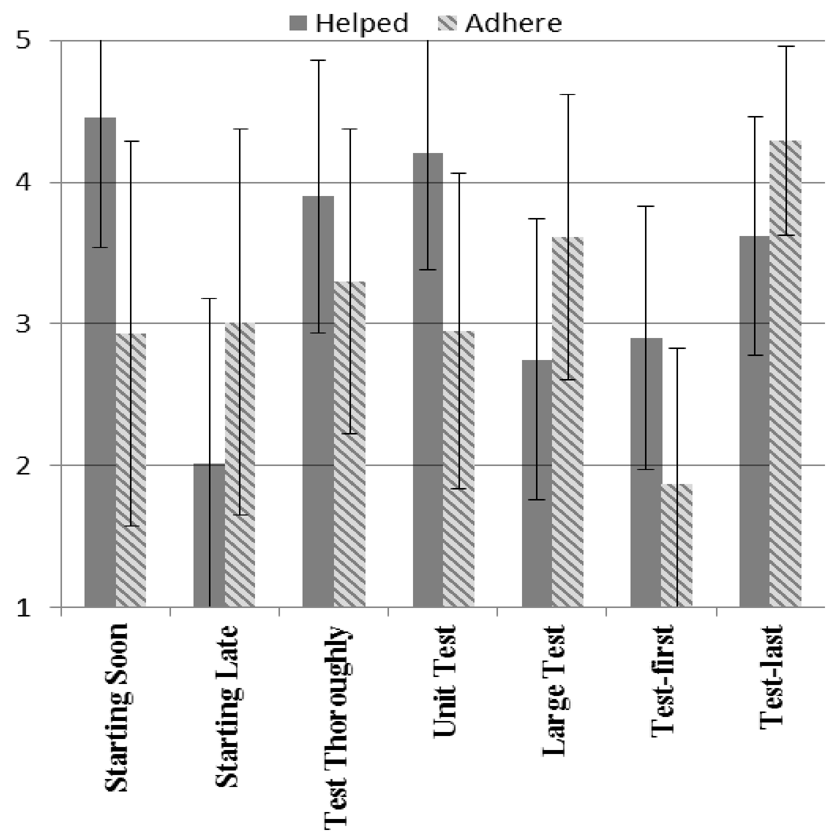


Figure 3.1. Personal Strength and Importance of Skills in Computer Science

Hypothesis 1 (positively correlated importance and strengths) cannot be supported given the lack of either correlation or statistical significance for time management (r^2=0.08, p=0.54), problem solving (r^2=0.00, p=1.00), attention to detail (r^2=0.07, p=0.60), and writing source code (r^2=0.03, p=0.80). On the other hand, positive correlations between ratings of helpfulness and adherence support Hypothesis 2 (positively correlated helpfulness and adherence). **Figure 3.2** shows a summary of the ratings.



(1) Very Harmful to **(5) Very Helpful**

Help (M, sd)	4.46, 0.92	2.02, 1.16	3.90, 0.96	4.20, 0.82	2.75, 0.99	2.90, 0.93	3.62, 0.84
Adhere (M, sd)	2.93, 1.35	3.01, 1.36	3.30, 1.08	2.95, 1.12	3.61, 1.00	1.87, 0.96	4.30, 0.67
r^2, p	**0.42, <0.01**	**0.33, =0.01**	**0.55, <0.01**	**0.57, <0.01**	**0.39, <0.01**	**0.49, <0.01**	0.23, =0.07

Figure 3.2. Behavior Adherence and Helpfulness Ratings Graph (above) with Correlations (below)

All items demonstrate positive correlations that are statistically significant except for *test-last*, which has a correlation approaching significance.

Likewise, Hypothesis 3 (TDD adherence leads to higher helpfulness rating) was supported with significant correlations between self-reported adherence to TDD and perceived helpfulness of TDD. Often adhering to TDD has a strong, positive correlation with agreement to: “TDD helped me write better test code” (r^2=0.67, p<0.01), “TDD helped me write better solution code” (r^2=0.65, p<0.01), “TDD helped me better design my programs” (r^2=0.60, p<0.01), and “In the future, I will choose to follow TDD when developing programs outside of this course” (r^2=0.57, p<0.01). Figure 2 illustrates the ratings and correlations.

For both the Brief Fear of Negative Evaluation Scale (BFNES) and Westside Test Anxiety Scale (WTAS), ratings were averaged to score each participant where higher scores represent higher anxiety. On a 5-point scale, WTAS found moderately-low project anxiety (M=2.16, sd=0.81) and BFNES found slightly higher evaluation anxiety (M=2.7, sd=1.08). Cronbach’s coefficient confirmed that BFNES (a=0.96) and WTAS (a=0.88) were internally consistent and the Shapiro-Wilk test found the results were not normally distributed for either BFNES (W<0.01) or WTAS (W=0.02).

Hypothesis 4 (higher anxiety will adhere less) suggested that project anxiety (WTAS) would correlate with avoiding TDD, in starting work late, and in not testing thoroughly. However, no conclusion can be made because no correlations were found between project anxiety and: “Beginning work as soon as it is assigned” (r^2=-0.08, p=0.54), “Developing thorough test code” (r^2=0.08, p=0.54), “Developing code and corresponding tests in small units at a time” (r^2=-0.02, p=0.87), nor “Developing tests before writing solution code” (r^2=0.03, p=0.82).

Moderate positive correlations between project anxiety and some ratings of Web-CAT’s helpfulness suggest some support for Hypothesis 5 (higher anxiety will rate Web-CAT higher) with significance for: “Web-CAT helped me improve writing solution code” (r^2=0.26, p<0.05), “Web-CAT helped me improve designing my programs” (r^2=0.36, p<0.01), and “Web-CAT helped me follow test-driven development” (r^2=0.45, p<0.01). However, correlations with some other items were not significant: “Web-CAT helped me improve writing test code” (r^2=0.17, p=0.19), “Web-CAT helped improve my time management” (r^2=0.15, p=0.27), and “Web-CAT helped improve my attention to detail” (r^2=0.21, p=0.11).

In addition, programming project submissions to Web-CAT allowed for analysis of patterns in student software development. It should be noted, however, that submissions to Web-CAT did not equally represent the code in all stages of development. On average, the first submission by a student already represented nearly 65% of the number of the statements in that student’s final submission. Therefore, the snapshots of work-in-progress are sometimes limited in what they reveal about early work habits.

However, when comparing the relationship between the time of first submission and eventual final test coverage of statements, there was a significant difference between those who achieved 100% coverage and those who did not. The median time of first submission for students who achieved 100% coverage was 3.18 days before the deadline, while students who did not achieve 100% coverage had a median time of first submission 1.22 days before the deadline. According to the Mann-Whitney U test, the time remaining before the project deadline had a significant effect on final test coverage achieved ($Z=-4.37$, $p<0.01$).

We also compared the test coverage of each snapshot in the series of submissions for each student's project. We investigated relationships between the following metrics:

- **Average test coverage** — Percent of statements covered by tests at time of each submission to Web-CAT, averaged for each student on each project
- **Total coverage rate** — Percent of submissions that achieved 100% statement coverage at time of submission to Web-CAT, measured for each student on each project
- **Final coverage** — Percent of statements covered by tests on student's final submission for a project
- **Final correctness** — Correctness of solution code on student's final submission for a project, as determined by instructor-written tests
- **Final Test NCLOC** — Amount of student-written test code, in terms of the number of non-comment, non-blank lines of code
- **Final Solution NCLOC** — Amount of student-written solution code, in terms of the number of non-comment, non-blank lines of code

The relationships between these metrics are summarized in **Table 3.1**, below. We found that the average test coverage (M=82%, sd=19.08) had a strong, positive correlation with both the project's final correctness and final coverage. It also had moderately positive correlations with final test and solution non-comment lines of code (NCLOC). Similarly, the total coverage rate positively correlated with final coverage and final correctness. A combination of high average test coverage and consistent total coverage rate would suggest incremental, thorough testing on behalf of the student.

Table 3.1. Correlations Between Test Coverage and the Quality and Quantity of the Final Submission

	Mean, sd	**Avg Test Coverage (r^2, p)**	**Total Coverage Rate (r^2, p)**
Final Coverage	97.49%, 0.09	***0.72, <0.0001***	***0.28, <0.0001***
Final Correctness	88.06%, 20.60	***0.55, <0.0001***	***0.39, <0.0001***
Final Test NCLOC	235.43, 108.01	***0.38, =0.0001***	-0.06, 0.3097
Final Solution NCLOC	355.37, 120.61	***0.21, <0.0001***	***-0.24, <0.0001***

3.1.4 Discussion

While Hypothesis 1's prediction—that students' perceived importance of skills would be positively correlated to their self-assessments—was not supported, the results may be interpreted with a positive outlook. Students' ability to separate their own ability from what they consider important skills in computer science may allow them to recognize their strengths and identify important skills they may lack.

Additionally, the strong relationship between what students characterize as helpful practices and their adherence illustrates reciprocity between affect and behavior in learning. A positive attitude toward good programming practices will encourage a student to act accordingly, which should reinforce that attitude with positive outcomes. The supportive result for Hypothesis 3 (TDD adherents will rate its helpfulness higher) strengthens this reciprocal model of learning, where we observe students who have adhered to test-driven development (TDD) characterize the method as helpful.

Students who adhered to TDD also benefited with significantly better test coverage and solution correctness. The observations of high average coverage per submission and percentage of submissions with 100% coverage associate with incrementally testing and developing a solution in tandem. While one cannot irrefutably conclude that these students were following the test-first principle of TDD, the pattern strongly suggests behavior consistent with unit testing. The adherents' improved outcomes also corroborate with other studies' findings that TDD improves quality of code and testing.

Interestingly, in post-hoc analysis we found that adherence to TDD principles of test-first and unit testing also correlated with adherence to the good practice of starting work early. Survey data showed modest, positive correlations between starting work early with both test-first ($r2=0.28$, $p<0.05$) and unit testing ($r2=0.33$, $p<0.01$).

Despite these encouraging results, our conclusions must also acknowledge two concerning findings. Foremost, both survey results and Web-CAT data showed trends in procrastinating before starting work. Procrastination should be discouraged and the data show negative effects on test coverage. This finding is consistent with other studies that found that students who earn A and B grades start work significantly earlier than those who earn lower grades [ESP+2009]. There is opportunity for research in dissuading this counterproductive behavior, perhaps through pedagogical or instructional technology interventions.

Secondly, survey results showed discouragingly low ratings for both helpfulness and adherence to test-first development (Figure 2). In fact, the Wilcoxon signed-rank test found that the mean score for helpfulness (M=3.62, SD=0.84) and adherence (M=4.29, SD=0.67) to test-last is significantly greater ($p<0.01$ and $p<0.01$, respectively) than the helpfulness (M=2.90, SD=0.93) and adherence (M=1.87, SD=0.96) to test-first. Snapshots of project submissions to Web-CAT may indicate whether students tested early or not, but they cannot concretely determine whether a unit test was written before or after its corresponding solution code. Consequently, we can only rely on the survey data to evaluate student adherence to test-first.

It is not entirely surprising that the test-first aspect of TDD garners more resistance than unit testing. For students with experience programming but not with TDD, the behavior of designing tests first is a considerable departure from immediately developing the solution. As mentioned previously, studies in both academia and industry found programmers who were younger and less experienced were less accepting of TDD. Our study suggests that student attitudes towards test-first development may be the primary hindrance for new novices accepting and adhering to TDD. Accordingly, therein lies an opportunity for computer science educators to explore ways to persuade students to use test-first development. We plan to investigate this opportunity in future research.

Finally, our research investigated possible relationships between psychological anxiety and the adherence to TDD and attitudes toward Web-CAT. Both Brief Fear of Negative Evaluation Scale (BFNES) and Westside Test Anxiety Scale (WTAS) were internally consistent. There was no observable relationship between project anxiety and adherence to TDD principles.

According to the Yerkes-Dodson law [TD1908], stress has a positive effect on performance until surpassing threshold where stress becomes too great and produces a negative effect on performance. Likewise, stress may motivate students to follow recommended methods unless the stress is so great that it encourages bad habits. Further investigation into individual differences in response to project anxiety is necessary to conclude how stress may influence software development behaviors. Meanwhile, the positive correlation found between project anxiety and Web-CAT helpfulness was modest, but indicates opportunity for easing disruptive affect with instructional technology.

On the contrary, we hypothesized (6) that instructional technology that includes feedback on test failures and other negative results would have an adverse effect on those with fear of evaluation (BFNES). However, the results of evaluation anxiety relating to impression on Web-CAT's helpfulness were inconclusive. It is important though to design feedback with sensitivity to possible negative student affect.

Our research investigated adherence to test-driven development (TDD) with particular attention to the role of student affect, or attitudes and emotions. As we observed, attitudes toward TDD influenced both negative and positive outcomes in students new to the method. Understanding the reciprocal nature of affect in learning appears to be especially important when teaching methods. Methods like TDD are taught but not necessarily assessed directly. Consequently, it is vital to evaluate whether students are adhering to the methods as well as understand their attitudes that will likely inform their behavior.

This study found significant results that emphasize beneficial outcomes of following TDD. However, it also identified challenges to persuade both desired affect and behavior. We hope this research will mark grounds for progress in designing instructional technology and computer science curriculum to improve TDD education.

3.2 Impacts of Teaching Test-Driven Development

We have published the work described in this section in the article, "Impacts of Teaching test-driven development to Novice Programmers" in the International Journal of Information and Computer Science [BE2012i].

3.2.1 Background

At Virginia Tech, introductory computer science courses began including TDD in 2003. These courses introduce TDD early and continue to reinforce it throughout the semester by requiring students to write their own software tests for all of their assignment solutions. Despite teaching students how to use TDD and encouraging them to apply its principles in class, we anecdotally observed some students not adhering to its core principles. These observations draw obvious concern.

Supporting and following TDD as supplemental material to the usual introductory courses requires both instructors and students to assume additional workload. To justify the extra burden, it is necessary to investigate the effects of TDD and weigh its merits. In this section, we assess the impact of using TDD in an academic setting and demonstrate its outcomes. In addition, we identify challenges to improving student adherence to TDD principles.

3.2.2 Method

Current literature on test-driven development (TDD) education is often limited to anecdotal observations and findings from a single semester. Likewise, assessment of students' programming assignments has focused on evaluating their completed code. However, since TDD is a process and not just an outcome, only considering the completed code turned in by a student does not provide adequate insight into the process the student followed to get there.

We used Web-CAT as an automated grader for programming assignments. Web-CAT provides rapid feedback to students with assessments of the quality of their solutions, tests, and style. In addition, students are allowed to submit their code to Web-CAT as many times as they like, without penalty—an average of 15 times for each assignment in our study. With each submission, students may review the provided feedback and try to improve their scores. Since each submission is assessed and archived, we gain the unique insight of multiple works-in-progress as students complete their assignments. By considering each submission as a momentary snapshot of the students' development, their history of submissions allows a deeper picture of their activities while developing their solution, and we gain a richer understanding of the processes they follow.

In addition, we have compiled submissions from five years of introductory computer science classes, each including multiple assignments. Every assignment required students to submit test code along with their solution code for assessing test thoroughness. With data from 59678 submissions to Web-CAT, the scale of our analysis is significantly larger than earlier investigations of student testing behaviors.

Over five years (ten academic semesters) of introductory Java courses, we taught TDD and collected students' work on programming assignments. All assignments required unit testing, written in JUnit [JUni2013]. As mentioned previously, students submitted their work to Web-CAT for automated feedback and grading. Some assignments included additional, manual evaluation from instructors or teaching assistants. However, only Web-CAT's automated assessment was included to control for individual differences in instructor grading. Scores were normalized (0 to 100) for maximum possible score from automatic grading alone.

Web-CAT evaluates students' solutions based on results of instructor-written reference tests. Web-CAT obscures instructor reference tests from the students. Instead, students receive feedback based on the results from the reference tests. Solution correctness was calculated based on the percent of instructor's reference tests passed. Web-CAT also uses Clover [Atla2013] to evaluate the thoroughness of students' unit tests. Test validity is determined by unit tests passing or failing the solution. Test completeness is assessed by code coverage: the percent of solution statements run by the unit tests passed. We concentrated on correctness and coverage to measure student outcomes in our analysis. **Figure 3.3** illustrates the evaluation of a student's submission for correctness and coverage.

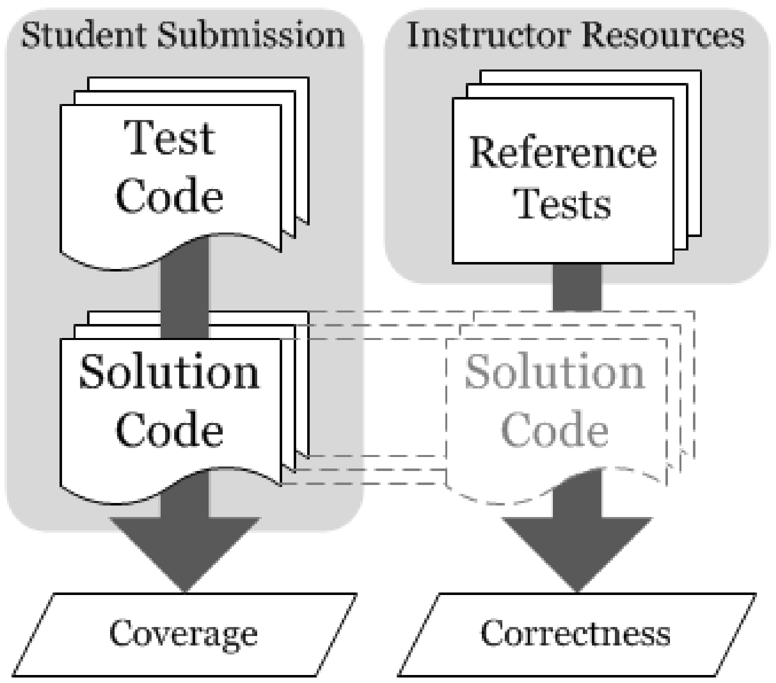


Figure 3.3. Evaluation of Outcome Metrics Based on Student Code and Obscured Reference Tests

For the purpose of our discussion, a submission refers to an individual snapshot of a student's work sent by the student to Web-CAT for assessment. For each assignment, each student averaged approximately 15 submissions. Although each submission is assessed, only their final submissions were considered when assigning grades. Accordingly, when describing students' outcomes, we refer to the correctness and coverage of this final submission, unless noted otherwise. To discern students' adherence to TDD principles, we concentrated on the following metrics:

- **Test Statements per Solution Statement (TSSS)**: the number of programming statements in student-written test classes relative to the number of statements in their solution classes.
- **Test Methods per Solution Method (TMSM)**: the number of student-written test methods relative to the number of methods in their solution.

Both TSSS and TMSM measure the amount of test code relative to the amount of solution code. Measuring the absolute amount of code (say, in terms of non-commented source lines of code) would make it difficult to control for the differing size and complexity of various assignments, and also differences between the early or late stages of development of a given student's solution. For instance, in early submissions, having few test statements is not necessarily a concern if the solution also is relatively small. Accordingly, TSSS and TMSM normalize values according to the relative amount of code written by the student so far.

TSSS provides an overall indication of how much work a student has devoted to testing the solution, compared to writing the solution. A low TSSS value (approaching 0) indicates that a student is not dedicating much effort to testing. However, since JUnit test cases typically are less complex than the solution algorithms they test, a TSSS below 1 is expected in most cases.

TDD also emphasizes testing in small units. When incrementally testing and developing units, a JUnit test method corresponds with a solution method that it validates. Accordingly, TMSM provides insight into how well students develop unit tests along with their solution methods.

To observe progress of students' development processes, we recorded these metrics for both the initial and final submissions on each assignment for every student. However, TDD is an incremental process so we also calculated average TSSS and TMSM across all submissions for each student on each assignment. In doing so, we can distinguish students who consistently make progress on developing unit tests from those who may neglect testing for periods only to catch up later.

3.2.3 Results

We used the Kolmogorov-Smirnov-Lilliefors test [CF2009] to determine whether each code analysis metric belonged to a statistical normal distribution. **Table 3.2** summarizes the results, where low p-values support rejecting the null hypothesis that the values are normally distributed.

Table 3.2. Mean, Standard Deviation, and KSL P-Values for Correctness, Coverage, TSSS, and TMSM

Metric		Mean	S.D.	p
Initial				
	TSSS	0.68	0.78	< 0.01
	TMSM	0.92	0.62	< 0.01
	Coverage	0.61	0.37	< 0.01
Average				
	TSSS	0.81	0.67	< 0.01
	TMSM	1.09	0.58	< 0.01
	Coverage	0.77	0.24	< 0.01
Final				
	TSSS	0.88	0.49	< 0.01
	TMSM	1.20	0.68	< 0.01
	Coverage	0.87	0.21	< 0.01
	Correctness	0.79	0.30	< 0.01

Initial, final, and average measurements for several metrics were tested independently. As the table shows, each metric's p-value is less than 0.01, which indicates all distributions are non-parametric. Spearman's rho (ρ) [CF2009] determines correlation relationships between non-parametric distributions. Likewise, we used Wilcoxon's 2-sample test (p) [CF2009] to compare means since distributions are non-parametric.

To investigate the relationship between testing early in development and final outcomes, we first tested correlations between Test Statements per Solution Statement (TSSS) on a student's very first submission (M=0.68, sd=0.78), as well as correctness (M=0.79, sd=0.30) and coverage (M=0.87, sd=0.21) achieved on that student's final submission for the same assignment. Results show a nominally positive correlation with both correctness ($\rho = 0.0929$, $p < 0.0001$) and coverage ($\rho = 0.1529$, $p < 0.0001$). The Test Methods per Solution Method (TMSM) of a student's initial submission (M=0.92, sd=0.62) is also positively correlated with correctness ($\rho = 0.2189$, $p < 0.0001$) and coverage ($\rho = 0.1931$, $p < 0.0001$) on the corresponding final submission. In addition, both average (M=0.81, sd=0.67) and final (M=0.88, sd=0.49) TSSS and TMSM (M=1.08, sd=0.58; M=1.20, sd=0.68) values demonstrate increasing strength of correlation with correctness and coverage. All correlations are statistically significant, as shown by low p-values. These correlations are summarized in **Table 3.3**.

While all the results are statistically significant ($p < 0.0001$), we considered the possibility that the correlations could be explained by unattributed phenomena. For instance, particularly diligent students may be more likely to follow directions from the instructor. Consequently, they may have produced high quality code regardless of whether they adhered to TDD. However, since they follow directions more closely than less diligent students, a positive correlation for TDD and code quality would emerge.

Table 3.3. Summary of Correlations Between TSSS, TMSM, and Correctness and Coverage Outcomes

				Correctness (M=0.79, sd=0.30)		**Coverage** (M=0.87, sd=0.21)	
Metric		**Mean**	**S.D.**	**ρ**	**p**	**ρ**	**p**
Initial							
	TSSS	0.67	0.78	0.0925	< .0001	0.1529	< .0001
	TMSM	0.92	0.62	0.2189	< .0001	0.1931	< .0001
Average							
	TSSS	0.81	0.67	0.1515	< .0001	0.2762	< .0001
	TMSM	1.08	0.58	0.2886	< .0001	0.2690	< .0001
Final							
	TSSS	0.88	0.49	0.2800	< .0001	0.4086	< .0001
	TMSM	1.20	0.68	0.3156	< .0001	0.3049	< .0001

Likewise, negligent students may write poor code regardless of their adherence to TDD. Since they may not follow directions as closely, they could be less likely to adhere to TDD and would therefore reinforce the positive correlation. Both plausible student stereotypes would strengthen correlation but would not suggest that TDD positively affects code quality.

To investigate the possibility of such confounding variables, we categorized students based on performance. Each group was determined based on students' correctness across all of their assignments. Students who achieved at least 80% correctness (which usually corresponds with A and B grades in American schools) on every programming project were designated as "high achieving." On the other hand, students who consistently achieved less than 80% correctness (usually designated as C, D, and F grades) on every assignment were assigned to the "low achieving" group. Lastly, students with at least one assignment under 80% correctness and at least one assignment above 80% were "mixed achieving." Approximately 14% of students qualified as low achieving, while 54% were mixed achieving and 32% were high achieving.

We controlled for potential effects of particularly motivated and unmotivated students by concentrating on the mixed achieving group. This group accounts for the majority of students and each student in the group demonstrated that they have the ability and motivation to write at least one quality programming project, but have also faltered on at least one other programming project as well. Consequently, this group provides an opportunity for investigating the difference in behaviors exhibited by a student when she succeeds in comparison to those exhibited when she fails another assignment. Moreover, a statistically significant relationship between TDD adherence and project correctness would provide compelling evidence of TDD's effects.

To examine within-subject relationships in the mixed achieving group, we performed a Mann–Whitney–Wilcoxon test for repeated measures [CF2009] comparing the average TSSS, TMSM, and coverage between assignments with high- (≥80%) and low- (<80%) scoring correctness. Average TSSS on high-scoring assignments (M=0.87, sd=0.71) was significantly greater ($p < 0.0001$) than that of low-scoring assignments (M=0.67, sd=0.66). Average TMSM was also significantly greater ($p < 0.0001$) on high-scoring assignments (M=1.18, sd=0.62) than low-scoring assignments (M=0.93, sd=0.50). Likewise, average coverage was significantly greater ($p < 0.0001$) for high-scoring assignments (M=0.83, sd=0.17) than for low-scoring assignments (M=0.63, sd=0.29).

For due diligence, we examined the trends of these same metrics across all groups. A post-hoc Tukey test [CF2009] identified differences in average TSSS, TMSM, and coverage between each group. The high achieving group had statistically greater ($p < 0.0001$) average TSSS (M=0.89, sd=0.64) than the mixed achieving (M=0.79, sd=0.69) and low achieving (M=0.61, sd=0.46) groups. The mixed group was also significantly greater ($p < 0.0001$) than the low group. Likewise, the high group's average TMSM (M=1.16, sd=0.59) was significantly greater ($p < 0.001$) than the mixed group (M=1.08, sd=0.58) and ($p < 0.0001$) the low group (M=0.79, sd=0.39). The mixed group was also significantly greater ($p < 0.0001$) than the low group. Average coverage was also significantly greater ($p < 0.0001$) for high (M=0.83, sd=0.18) than mixed (M=0.75, sd=0.25) and low (M=0.59, sd=0.29) groups. Correspondingly, the mixed group's average coverage was significantly greater than that of the low group with a p-value below 0.0001.

Spearman rank tests [CF2009] also indicated correlations between average TSSS, TMSM, coverage and the final correctness and coverage within each of the three groups.

Table 3.4 comprehensively presents the correlations for each metric, separated by group. Again, all correlations are statistically significant with p-values less than 0.001. Positive correlations with correctness and coverage persist in all three groups, with only one exception: average TSSS for high achievers demonstrates a trivially small negative correlation with correctness. Since high achievers already average over 96% correctness, it is difficult to improve much on that score. In addition, high performing students who are just short of achieving 100% correctness may add extraneous test assertions to identify their last remaining defects, which may explain the slightly negative correlation. Otherwise, all three groups demonstrated small- to moderately-positive correlations between each TDD-indicating metric and the final correctness and coverage. Overall, average coverage had the strongest positive correlations with both correctness and coverage.

Anecdotal observations by course instructors suggest that some students completely ignore TDD and only test en masse at the end of their work, after completing a working solution. They may do so only to satisfy the assignment requirements where part of the grade is dependent on Web-CAT's automated testing assessment. To investigate the consequences of this test-last strategy, we identified all situations where the student's initial submission contained no test code at all. We compared the correctness and coverage outcomes of these late-testers to the rest of the students who demonstrated at least some early testing. Early-testers (M=0.81, sd=0.29) achieved significantly better correctness in their final submissions ($p < 0.0001$) than late-testers (M=0.70, sd=0.38). Likewise, the early-testers (M=0.88, sd=0.18) achieved significantly better coverage ($p < 0.05$) than late-testers (M=0.78, sd=0.33).

Table 3.4. Correlations Grouped by High-, Mixed-, and Low-Achieving Students

			Correctness (M=0.79, sd=0.30)		Coverage (M=0.87, sd=0.21)	
Metric	**Mean**	**S.D.**	**ρ**	**p**	**ρ**	**p**
Average TSSS						
High Achieving	0.89	0.64	-0.1002	< 0.001	0.1457	< 0.0001
Mixed Achieving	0.79	0.69	0.1670	< 0.0001	0.2863	< 0.0001
Low Achieving	0.61	0.46	0.2930	< 0.0001	0.5015	< 0.0001
Average TMSM						
High Achieving	1.16	0.59	0.2110	< 0.0001	0.1361	< 0.0001
Mixed Achieving	1.08	0.58	0.2680	< 0.0001	0.2796	< 0.0001
Low Achieving	0.79	0.39	0.2135	< 0.0001	0.3712	< 0.0001
Average Coverage						
High Achieving	0.83	0.18	0.2116	< 0.0001	*0.4741*[A]	*< 0.0001*
Mixed Achieving	0.75	0.25	0.4624	< 0.0001	*0.6623*[A]	*< 0.0001*
Low Achieving	0.59	0.29	0.4243	< 0.0001	*0.8305*[A]	*< 0.0001*

A - Final Coverage data are subsets of Average Coverage calculations, so strong positive correlations are expected

Lastly, we wanted to identify what early behaviors students demonstrate when they eventually achieve complete (100%) coverage. We compared the initial coverage, TSSS, and TMSM of students who achieved complete coverage to those with incomplete (< 100%) coverage. Complete testers started with higher ($p < 0.0001$) initial coverage (M=0.68, sd=0.39) than incomplete testers (M=0.61, sd=0.35). Complete testers also began with a greater ($p < 0.0001$) initial TMSM (M=1.05, sd=0.73) than those with incomplete final coverage (M=0.89, sd=0.57). While complete initial TSSS (M=0.74, sd=0.82) was greater than that of the incomplete group (M=0.68, sd=0.71), the difference was not statistically significant (p=0.1761).

3.2.4 Discussion

The measures for Test Statements per Solution Statements (TSSS), Test Methods per Solution Methods (TMSM), and coverage on a student's initial submission all correlated with positive outcomes in terms of final correctness and coverage of the completed solution. This suggests that early testing yields both higher quality tests and solutions. It is also encouraging to find that 87% of first submissions for an assignment include at least some test code. We also see that those who do not test early are less likely to achieve complete coverage by their final submissions, even if they employ extensive testing later in their development. Together, these findings support the claim that test-driven development (TDD) promotes code that is easier to debug and maintain.

TDD also emphasizes testing in small units. We believe that TMSM provides an estimate of how well students comply with unit testing principles by developing corresponding test and solution methods. Meanwhile, TSSS more broadly describes the work put into testing compared to that put into developing a solution. TMSM often had stronger positive correlations with correctness and coverage than TSSS, which suggests that unit testing is particularly beneficial.

However, findings also suggest that success does not only depend on unit testing early, but also on following TDD consistently and incrementally. Higher average TMSM over all submissions for an assignment reflects this behavior. Average TMSM shows an even stronger positive correlation with correctness and coverage than TMSM on only the initial submission. It is also a strong endorsement of consistently following TDD since these correlations hold true for all types of students, regardless of their final correctness. That is to say, even if a poorly performing student does not demonstrate the knowledge or effort required to produce a high-quality program, he will still likely improve his solution and testing by following TDD.

While submissions to Web-CAT cannot conclusively indicate whether a student has written software tests first or written solution code first, survey responses confirmed an overall concern with motivating students to adapt a test-first mentality. Overall, students do not consistently practice the principle of testing first. However, as a consolation, they do generally appear to start testing early, even if they “code a little, test a little” [26] instead of “test a little, code a little.”

Our study of test-driven development (TDD) featured a previously-unmatched volume of empirical data, spanning ten academic semesters of programming projects. With such extensive insight into student coding practices, we contribute unique findings to the practice and teaching of TDD. Our study presents: novel metrics for assessing adherence to TDD, strong evidence for positive outcomes from following TDD, and direction for improving TDD education.

We developed new metrics for analyzing code that richly describe TDD. Although measuring tests’ coverage was not new, it provides a depiction of the quality or completeness of tests. To complement measurements of test completeness, we focused on adherence to the TDD philosophy of unit testing incrementally. Average Test Methods per Solution Methods (TMSM) in particular helps indicate how consistently a programmer develops unit tests paired with their solution methods.

Previously, evaluating student code was typically restricted to reviewing only the final product of their work. Instead, our analysis leveraged Web-CAT to capture multiple snapshots of students’ code while they developed their programs. As a result, we acquired a uniquely rich depiction of the behaviors and processes students follow. With enhanced detail of their processes, project-specific averages of TMSM and coverage revealed how well students demonstrated consistent adherence to TDD over time.

Using these novel metrics on our exceptionally large data set, we add strong evidence that TDD advocates improvements in code and testing quality. We support this validation of TDD with consistent, positive correlations and statistically significant differences in resulting outcomes. Now with improved confidence in the positive impact of following TDD, we can continue advocating and refining TDD education.

Chapter 4

Interventions for Reinforcing Testing Methods

Chapter 4 concentrates on the second main objective of our research: "**Design an eLearning intervention for encouraging TDD adherence and evaluate its impact on student affect, behaviors, and outcomes.**" Accordingly, in the first section (*4.1 Adaptive Feedback*), we describe the design for an automated, adaptive feedback system for encouraging TDD. We also provide initial evaluation with an experiment comparing students' attitudes, behaviors, and outcomes from one semester using the adaptive feedback system to another semester without it.

The second section (*4.2 A Formative Study of Influences on Testing Behaviors*) concentrates on data collected from interviews with students. In this study, we gathered feedback from students on their general software development strategies as well as their interaction with eLearning technology during development. We also discuss an experiment using variations of the adaptive feedback system with different schedules for reinforcement. Finally, the third section (*4.3 Responses to Adaptive Feedback*) describes short-term responses to rewards and punishments on consecutive submissions.

4.1 Adaptive Feedback

We have published the work described in this section in the article, "Impacts of Adaptive Feedback on Teaching test-driven development" and presented it at the symposium for Computer Science Education (SIGCSE) [BE2013ii].

4.1.1 Background

At our university, we teach test-driven development (TDD) in introductory CS1 and CS2 courses. However, to weigh the possible benefits and shortcomings of TDD in an academic setting, we focused on investigating how well students follow TDD and assessing how it affected their code quality.

In several instances, we observed anecdotal evidence of students resisting TDD. For example, while working on a programming project, one student explained, "*I will worry about my testing […] later; I just want to get it working first*." Another student worried about failing software tests and lamented, "*[The program] is my baby and I don't want to be told that my baby is bad!*" Both cases demonstrate attitudes that discourage TDD adoption.

Other studies have also identified challenges to motivating TDD adherence. Researchers identified students' reluctance to accept TDD especially from less experienced programmers [JS2007][MM2005][SP2006]. Our preliminary investigation confirmed claims that following TDD contributes to code and test quality. We also supplemented our anecdotal observations with evidence of some students' reluctance to follow TDD. Students' project code identified some behaviors that violate TDD principles. Furthermore, our survey linked poor attitudes toward some aspects of TDD as a reason for non-adherence [BE2012ii].

Consequentially, while we recognize educational value in continuing to teach TDD, we also acknowledge a need to improve TDD adherence. We sought pedagogical interventions to foster better attitudes toward TDD and encourage its use. As a result, we developed an adaptive feedback system to track students' programming behaviors and to reinforce TDD. This paper describes the system's design and a preliminary evaluation of its effects.

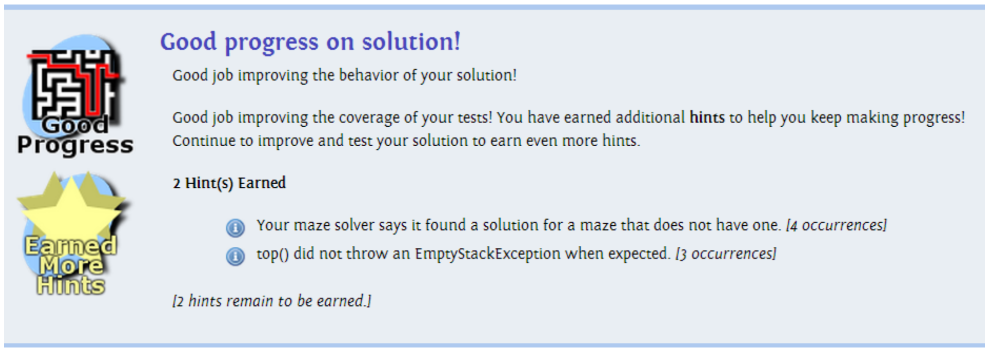
4.1.2 Method

An Adaptive Feedback System

We designed and evaluated a pedagogical intervention for improving student adherence to test-driven development (TDD). The intervention intends to monitor students' development habits and persuade them to follow TDD's incremental process. We implemented this intervention by developing a plugin for Web-CAT that provides adaptive feedback based on how well the student is adhering to incremental unit testing. Since Web-CAT allows students to resubmit multiple times to improve their scores, the plugin monitors how a student's code changes from one submission to the next. The plugin particularly takes note of changes in: the amount of solution code, the amount of test code, the correctness of the solution (as determined by the instructor's reference tests), and the test coverage achieved (as determined by the student's tests run against her own code).

As mentioned previously, Web-CAT displays hints to the student when a submitted solution fails any instructor reference tests to help guide the student in the right direction. These hints have obvious value to the students because they provide some guidance on where to look for problems in their programs. By default, Web-CAT shows three hints to the student (the exact number can be configured by the instructor) and obscures the rest until one or more of the displayed hints are resolved. However, since hints help students, we decided to use the hints as incentives to follow TDD.

The plugin monitors progress on correctness and coverage as indicators of whether a student is testing incrementally while developing a solution. When students satisfy a threshold in making progress on testing, they earn a credit for a hint. With the plugin, students no longer receive three "free" hints automatically. Instead, with each credit they earn, they receive a hint. If a problem that triggers a hint is resolved, the credit remains and a new (previously obscured) hint displays instead. As students continue to make progress on their solution and testing, they can accumulate multiple hint credits. **Figure 4.1** shows an example of adaptive feedback where the student has earned two hints by following TDD practices. See **Appendix B** for a complete guide for the adaptive feedback.



Good progress on solution!

Good job improving the behavior of your solution!

Good job improving the coverage of your tests! You have earned additional **hints** to help you keep making progress! Continue to improve and test your solution to earn even more hints.

2 Hint(s) Earned

- Your maze solver says it found a solution for a maze that does not have one. *[4 occurrences]*
- top() did not throw an EmptyStackException when expected. *[3 occurrences]*

[2 hints remain to be earned.]

Figure 4.1. The Web-CAT Plugin Displays Adaptive Feedback with Hints Earned

However, if a student's submission pattern does not indicate sufficient adherence to TDD, he will not earn any hint credits. Instead of displaying hints, the plugin displays an image and message that encourages the student to improve their testing in order to earn hints.

In addition to earning and showing hints, the plugin also displays affirming messages to acknowledge good habits, even when they are insufficient to warrant earning another hint. For example, if the plugin detects that the student has increased the amount of test code but has not improved coverage, the message praises the work done and suggests continuing to improve test coverage. All adaptive messages and accompanying images purposely use only positive reinforcement to encourage and gently persuade students to follow TDD. The choice of wording was influenced by the need to promote and maintain a more positive mood regarding the feedback received.

We also engineered the adaptive feedback system to account for students who may try to manipulate or cheat in hopes to gain more (unearned) hints. First, students may only earn hints by making improvements to their solution and testing. While the feedback messages acknowledge added solution and test code, simply adding lines of code without improving test coverage or solution correctness does not earn hint credit.

Second, a student might remove code in between submissions with the hope of earning more hints on the following submission by adding the code back and "faking" an impression of improving coverage and/or correctness. However, the plugin compares the latest submission to whichever previous submission had the highest coverage and correctness, rather than simply comparing to the preceding submission. Therefore, attempts to manipulate indicators of progress in this way will not earn hints.

Evaluation Method

To evaluate the impact of this adaptive feedback intervention, we collected data from student submissions to Web-CAT for two semesters of our CS2 course, Software Design and Data Structures. The programming projects for the first semester did not use our intervention, so Web-CAT feedback always showed (up to a maximum of) three hints to each student, whenever hints were available. The second semester used the adaptive feedback system to provide encouragement for following TDD, but required students to earn hints as incentives for doing so. Both semesters used the same four project assignments.

Students signed consent forms to allow their submission data to be included in the analysis at the conclusion of each semester. Participation was voluntary and participants did not receive any form of compensation. 87 of 130 (66%) students enrolled in the first (control) semester and 78 of 129 (60%) students enrolled in the second (experimental) semester opted to participate. Code analysis excluded faulty Web-CAT submissions (typically, submissions that were cancelled before they were processed, or where a student accidentally submitted the wrong files). Students who did not submit at least once for each of the four programming projects were omitted from analysis. Students averaged roughly 22 submissions per project.

If the pedagogical intervention succeeds in persuading students to adhere to TDD, we may observe multiple effects. Primarily, increased average TMSM and increased average coverage both indicate closer adherence to unit testing. Since we previously found positive correlations for both metrics with final correctness and coverage, we may also observe improved outcomes because of TDD adherence. Persuading TDD adherence may produce longer-term results as well. As students follow TDD, we may see increasing average TMSM and increasing average coverage over the span of the semester for the experimental group as they adopt TDD habits and mature in their practice. Statistical analysis of the data will evaluate the following hypotheses:

H1) The experimental group will have significantly greater average TMSM and average coverage than the control group.

H2) The experimental group will have significantly greater project correctness and coverage scores than the control group.

H3) The experimental group's average TMSM and average coverage will increase over time relative to the control group's average TMSM and average coverage trends.

Survey

We also collected surveys at the conclusion of each semester to investigate student attitudes and perceptions of TDD. With the survey, we gathered information about attitudes and perceptions of TDD, self-reported adherence to TDD principles, and anxieties. The survey also asked students to indicate how frequently they practiced different development behaviors. Since we are concerned about how negative feedback may affect student attitudes and behaviors, we also used two questionnaires to measure anxiety: the Brief Fear of Negative Evaluation Scale (BFNES) [RWT+2004] and the Westside Test Anxiety Scale (WTAS) [Dris2007]. Both are brief, validated questionnaires. The survey instruments are described in detail in section 3.1.2 and the complete survey is shown in **Appendix A**.

Students completed the survey at the end of each semester. Participation was voluntary and participants did not receive compensation. 61 students from the control and 54 students from the experimental group completed the survey. Using the results from the surveys, we evaluate the following hypotheses:

H4) Students' perceptions of the helpfulness of test-first and unit testing will have a positive correlation with their self-reported adherence to the same behaviors.

H5) The experimental group will value the helpfulness of test-first and unit testing behaviors significantly higher than the control group.

H6) The experimental group will score significantly lower on WTAS (project anxiety) scale relative to their BFNES (fear of negative evaluation) scale when compared to the control group.

H7) The experimental group will respond more positively to following TDD in the future than the control group.

4.1.3 Results

Before testing hypotheses H1-H3 (experimental group producing better correctness, coverage, and TMSM) on code analysis, we used the Shapiro-Wilk test to check for normal distributions. None of the following code metrics fit normal distributions, where low p-values reject the null hypothesis that the data is normally distributed: correctness (M=0.84, sd=0.27, p<0.0001), coverage (M=0.93, sd=0.17, p<0.0001), average TMSM (M=0.68, sd=0.27, p<0.0001), and average coverage (M=0.79, sd=0.20, p<0.0001). Consequentially, we used the Mann-Whitney-Wilcoxon test to check for significant differences between independent, non-parametric samples.

H1 hypothesizes that the experimental group will have greater average coverage and average TMSM than the control group. However, the experimental group did not have significantly different (p=0.80) average coverage (M=0.79, sd=0.20) than the control (M=0.79, sd=0.19). Likewise, there was no significant difference (p=0.81) between the average TMSM for the experimental group (M=0.69, sd=0.28) and the control (M=0.68, sd=0.26). Results do not provide any support for H1.

H2 hypothesizes that the experimental group's correctness will be greater than that of the control. However, no significant difference (p=0.41) was found between the experimental (M=0.83, sd=0.27) and the control (M=0.84, sd=0.26) groups. Furthermore, the coverage for the control (M=0.94, sd=0.17) was significantly greater (p<0.001) than the experimental group (M=0.92, sd=0.17), contradicting H2.

The third hypothesis (experimental group will have greater TMSM and average coverage rate of increase) compares the trends of average TMSM and average coverage over the entire semester. First, we calculated the difference in average TMSM and average coverage between the first and last projects for each subject. The Mann-Whitney-Wilcoxon test showed no significant differences (p=0.73) in the change in average

TMSM between the control (M=-0.12, sd=0.48) and experimental group (M=-0.11, sd=0.48).

The average TMSM decreased over the course of the semester for both groups, but this decrease may be attributed to assignment difficulty. The first project only involved developing a business logic model, while latter projects additionally required developing user interfaces and event handling, which are typically less straight-forward to test. Both groups showed a similar decrease in average TMSM from the first project to the last. No significant difference between the two groups indicates that there is insufficient evidence to support H3 that the experimental group would mature over the semester to increase TMSM when compared to the control.

On the other hand, the experimental group's change in average coverage (M=0.10, sd=0.19) is greater than that of the control group (M=0.05, sd=0.20) and is approaching significance (p=0.10). Since this result suggests there might be an effect between groups, we performed a multivariate analysis of variance testing effects for PROJECT (representing the repeated measure for four projects) and PROJECT*TREATMENT interaction. We found an effect for PROJECT (p<0.0001), but no significant interaction for PROJECT*TREATMENT (p=0.22), so there was no observed treatment effect. **Table 4.1** shows the average coverage for each project.

Table 4.1. The Average Coverage on Each Project, Separated by Treatment

	Project 1	**Project 2**	**Project 3**	**Project 4**
Control	M=0.82	M=0.68	M=0.79	M=0.87
	sd=0.16	sd=0.23	sd=0.18	sd=0.17
Experimental	M=0.79	M=0.69	M=0.78	M=0.90
	sd=0.18	sd=0.22	sd=0.18	sd=0.11

The final four hypotheses address attitudes observed from survey results. We used the Shapiro-Wilk test to find that unit testing helpfulness (M=4.09, sd=0.83, p<0.0001) and adherence (M=3.07, sd=1.14, p<0.0001) responses are not normally distributed. Likewise neither test-first helpfulness (M=3.04, sd=0.99, p<0.0001) nor adherence (M=2.04, sd=1.05, p<0.0001) are normally distributed. We used Spearman's ρ to investigate relationships between non-parametric samples. There is a moderately strong positive correlation (ρ=0.57, p<0.0001) between unit testing helpfulness and adherence ratings. Similarly, there is a moderately positive correlation (ρ=0.50, p<0.0001) between test-first helpfulness and adherence. These correlations verify the relationship between perceived helpfulness and adherence found in our previous study and supports H4 (positively correlated helpfulness and adherence).

As described in H5, we expected the experimental group to rate the helpfulness of test-first and unit testing behaviors higher than the control group. To the contrary, a Mann-Whitney-Wilcoxon test found the control group (M=4.20, sd=0.82) rated unit testing helpfulness greater than the experimental group (M=3.96, sd=0.82) at a difference approaching significance (p=0.09). On the other hand, the experimental group (M=3.20, sd=1.05) rated test-first helpfulness greater than the control group (M=2.90, sd=0.93) at a difference approaching significance (p=0.06). Neither difference appears to translate into a meaningful distinction at the level of the response scale used, however.

H6 (experimental group will have lower anxiety) addresses comparing scores from the WTAS and BFNES scales. Cronbach's coefficient for WTAS (α=0.90) and BFNES (α=0.95) confirmed each scale's internal reliability. Students in the experimental group scored slightly lower for project anxiety (M=2.10, sd=0.91) than the control group (M=2.2, sd=0.81) but this difference was not significant (p=0.48). When WTAS scores were adjusted relative to each subject's BFNES score, there was still no significant difference (p=0.68) between the experimental (M=1.12, sd=0.58) and control groups (M=1.07, sd=0.41).

Finally, to test H7 (experimental group will be more likely to use TDD in the future), we compared each group's reported likelihood of using TDD in the future. The experimental group (M=3.59, sd=1.09) provided higher ratings than the control group (M=3.38, sd=1.21) but the difference is not statistically significant (p=0.38).

4.1.4 Discussion

We based each of the seven hypotheses on the expectation that the adaptive feedback system would persuade adherence to test-driven development (TDD). However, none of the statistical analyses provided sufficient evidence to claim it did so. Nevertheless, the results provide insight into the plugin's strengths and weaknesses and can serve as a formative assessment of its use as an educational and persuasive tool.

Although the differences were not quite significant, the experimental group's higher rating of test-first's helpfulness is worth noting since our previous study identified attitudes toward the test-first approach as the primary factor hindering acceptance of TDD. This insight into opinions of test-first approach is particularly useful in this research since the programming metrics concentrate on incremental unit testing, but cannot explicitly observe test-first adherence. Continuing to improve upon that impression of a test-first approach has potential for persuading changes in student behavior.

Improving attitudes toward test-first development may be a matter of better demonstrating its immediate value. Benefits of early testing—such as improved confidence—are difficult to portray concretely. Consequently, it would be valuable to investigate students' expectations and appraisals to build a mental model and make the effects of test-first development more salient.

Likewise, the trends in changing adherence to TDD over the course of a semester show promise for potential long-term persuasion. Several factors may contribute to the apparent lack of immediate changes in student behavior. Primarily, by the time a student receives adaptive feedback, it may be too late to persuade considerable change. Our studies have found that students' first submission for any given assignment often has over half the number of lines of code of their final submission [BE2012][ESP+2009]. Instead, students may be more inclined to change their behavior if persuasive interventions begin before they have established a pattern of behavior. For this reason, it would be worthwhile to explore ways to increase how soon and how often students receive adaptive feedback.

4.2 A Formative Study of Influences on Testing Behaviors

We have published the work described in this section in the article, “Impacts of Adaptive Feedback on Teaching Test-Driven Development” and presented it at the symposium for Computer Science Education (SIGCSE) [BE2013ii].

4.2.1 Background

With the emergence of automated grading systems that allow students to submit their work throughout development, instructors gain improved granularity of students’ work patterns. While exams may evaluate how well students understand a development technique, snapshots of students’ development over time may reveal how they apply the technique. Furthermore, when students receive automated feedback while they are still working, there are opportunities to influence their behavior.

At our university, we teach test-driven development (TDD) in introductory CS1 and CS2 courses. TDD involves an incremental “test a little, code a little” process meant to encourage confidence and improve maintainability [Beck1999]. However, students may find the test-first approach unnatural or intimidating. Consequently, they may benefit from repeated reinforcement during programming assignments.

In this paper, we describe a system we designed to encourage students’ adherence to TDD. We compare the results of implementing different schedules of reinforcement and the presence or absence of specific goals in automated, adaptive feedback. In addition, we explore influences on students’ testing behaviors through interviewing them about their experiences developing programming assignments and interacting with the adaptive feedback system.

4.2.2 Method

By introducing our adaptive feedback system to students, we aimed to observe common testing behaviors and investigate how the feedback influences those behaviors. Using students’ submissions to Web-CAT provides insight into the changes in their code over time. However, this approach has two principal limitations in discerning test behaviors. Firstly, the data only reflects changes in students’ code after they first submit to the automated grading system. We found that the average first submission already had 71% (sd=25%) of non-comment lines of code (NCLOC) when compared to their respective final submission NCLOC. Consequently, the data is blind to the beginning stages of development.

Secondly, there is no benefit for students to submit tests before writing the corresponding solution code (as encouraged by TDD) since without the solution, correctness will not improve. Likewise, Web-CAT cannot evaluate test coverage in the absence of solution code. Therefore, we cannot differentiate the incremental patterns "test a little, code a little" and "code a little, test a little" since students will only submit test code along with solution code. Continuous data collection would be necessary to determine the fine-grain order of testing versus coding the solution.

Consequently, we conducted interviews with students so they could explain their development processes and testing behaviors. This mixed-method approach benefits from quantitative, empirical analysis from code collection supplemented by explanations and clarifications from the students themselves.

Qualitative Investigation

While quantitative assessment can offer empirical data demonstrating testing behaviors, it lacks insight into explaining why students adopt particular processes and strategies. Previous studies have speculated into reasons for students' attitudes and behaviors from anecdotes and surveys [BE2012][MM2005]. However, we recognized a need to speak directly to the students in depth. Consequently, at the conclusion of the Spring 2013 academic term, we recruited students to participate in group interviews to explain their personal strategies, procedures, and motivations behind their behavior.

Twelve students volunteered to participate in the interviews, of whom seven showed and consented to participate (5.5% of 128 who sat the final exam). Volunteers were provided a meal (valued approximately $10 USD/person) to participate in group interviews for about one hour.

While this sample size may sound small, one should remember that the intent of the interviews is to elicit formative feedback rather than to empirically test hypotheses. By comparison, user experience studies often report that the majority of findings are discovered within the first five participants [Nielsen1993]. Likewise, in the interviews, we concentrated on eliciting feedback on students' experience with the adaptive feedback system and generating broad motivations and testing strategies.

Interviews were scheduled in two groups (n=4 and n=3) so that students could compare and contrast their experiences and strategies to generate discussion. Unlike focus groups, each participant was specifically prompted to respond to every one of the interviewer's questions and while there was discussion between participants, there was no attempt to collaborate on—or agree to—a consensus opinion.

After providing written consent and doing quick introductions, the group interviews began with an opportunity for each participant to explain their top priorities and objectives when a programming project is first assigned. Then, the participants were asked to each share and discuss their personal testing strategies—including when they began testing, testing incrementally or periodically in large portions, and testing first or after corresponding solution code.

Next, participants were shown an example screen capture of Web-CAT feedback and they discussed the different parts of the feedback. Without leading them to specific parts of the page, participants were asked to explain what they paid attention to and what was most important to them after their first submission to Web-CAT. They were also asked to explain what then influenced their next step in development. This series of discussions was repeated for screens representing resubmissions where they first earned hints and then again, once they achieved high (but not quite perfect) correctness and coverage scores.

Lastly, they were specifically probed on parts of Web-CAT's feedback to discuss what is or is not helpful and how different elements of feedback may play a part in development going forward. The screen elements discussed include (visually from top to bottom): grade summary, individual file details, adaptive feedback, results from running student's tests, code coverage from student's tests, estimation problem correctness, and interpreting correctness/testing scores. The interviewer particularly encouraged explanations about the adaptive feedback's goals and hints.

Next, each participant was asked if (and how) their individual testing strategies evolved over the duration of the academic term. The group discussions concluded with each participant completing a brief questionnaire on paper to reflect on and summarize their testing habits. each individual's reflection over their personal testing habits. The questionnaire asked participants to rate how often (from 1, rarely, to 5, often) they did the following:

- Started writing tests early in development
- Started writing tests late in development
- Wrote one test at a time to test a small part of the solution code
- Wrote many tests at a time to test large portions of the solution code
- Wrote tests before writing its corresponding solution code
- Wrote tests after writing its corresponding solution code

Likewise, the questionnaire asked to rate the same six behaviors by "how likely you are to do the following in the future, from: (1 Very Unlikely) to (5 Very Likely)." At the bottom of the questionnaire, participants had empty space to "Please describe how your testing and development strategies may change or stay the same, depending on the size of the program you are working on." Comparing differences between current behavior and projected future behavior could reveal influences on long-term testing strategies.

Finally, each participant was given a blank sheet of paper and a pencil and were asked to draw a timeline that depicted their typical project development from when it is assigned to the participant's last submission to Web-CAT. They were given freedom to express their development as best they could, but were asked to also identify when they worked on the solution, when they tested, and when they submitted to Web-CAT.

Quantitative Evaluation

While Web-CAT provides students with feedback upon each submission, it also collects a snapshot of their work that we can later analyze. Since we are particularly interested in students' testing behaviors, we analyzed each submission for both quantity and quality of test code as well as that of solution code. Web-CAT allows for part of students' grades to be manually entered by instructional staff by considering aspects such as design and documentation. However, to eliminate subjective judgments and potential inconsistency between graders, we only included Web-CAT's automated measurements in our quantitative assessment.

As previously mentioned, correctness describes how well students' solution code performs, as measured against the instructors' obscured tests. On the other hand, the test coverage describes the thoroughness of testing. Coverage represents the percent of statements, conditionals, and branches in the student's code exercised when run against her own tests. Influences of feedback on students' testing will be illustrated in Δ coverage, or how coverage on the final submission compares to that of the first submission. Positive values represent improvements in coverage while negative values suggest complacency or neglecting testing.

In addition to these measurements of code quality, we examined the quantity of code written. Non-comment lines of code (NCLOC) is the self-explanatory amount of semantic code. We maintained records of test and solution NCLOC separately to compare the relationship between amount of test and solution code. Similarly to coverage, we concentrated on how test NCLOC might change in response to adaptive feedback. Δ test NCLOC reveals increases or decreases in the amount of test code from the first to last submission within an assignment.

To check for normality of their distributions, we performed the Shapiro-Wilk test on measurements used for these metrics. All of the following measurements (with significantly small p-values) rejected the null hypothesis of normal distribution: correctness ($M=0.91$, $sd=0.16$, $p<0.0001$), first submission test NCLOC ($M=129.83$, $sd=125.63$, $p<0.0001$) and coverage ($M=0.69$, $sd=0.32$, $p<0.0001$), and final submission test NCLOC ($M=247.15$, $sd=162.20$, $p<0.0001$) and coverage ($M=0.98$, $sd=0.08$, $p<0.0001$). Consequently, all comparisons between treatments used Wilcoxon tests for non-parametric data.

Experimental Design

In Spring 2012, students in a CS2 (Software Design and Data Structures) course received feedback for their programming assignments using our adaptive feedback system with its initial design with constant (C) reinforcement schedule and hidden (H) coverage goals. Students received feedback according to this treatment for the entire term, encompassing four independent programming assignments. Of the 129 students who sat the final exam, 78 (60%) provided written consent to include their data in our analysis.

Following the preliminary semester, we implemented additional treatments of the adaptive feedback system, including delayed (D) and random (R) reinforcement schedules, each with separate shown (S) and hidden (H) goal combinations. Of the 128 students who sat the final exam in 2013, 84 (66%) consented to include their work in the data analysis. We planned a 3x2 factorial design of the treatments, treatments would be assigned to the five different course sections for each assignment to enable both within- and between-group comparisons.

However, constraints prevented us from implementing our planned experimental design. Firstly, the CS2 instructor developed new projects so that the Spring 2013 offering would not replicate the assignments completed in Spring 2012. In good faith, we could not ask the instructor and his students to forgo innovative course material for the sake of experimental validity. Secondly, we recognized a technical flaw in the adaptive feedback when applied to Spring 2013's first assignment. As a result, we had to exclude Project 1's data and use the semester's three remaining assignments to make comparisons between the five alternate treatments.

Consequently, we resolved to randomly assign each of the five course sections a unique experimental group. While this design excludes the possibility of within-subject comparisons, it controls for potential ordering effects of being exposed to different treatments on subsequent projects.

To see if the assignments between the two terms were of comparable size and difficulty, we compared the size and outcome of assignments between the two terms. Using the Mann-Whitney-Wilcoxon test, we found assignments from 2012 (M=325.79, sd=134.33) required significantly more solution NCLOC ($p<0.0001$) code than those assignments given in 2013 (M=290.20, sd=194.85). Moreover, the students performed better ($p<0.05$) on correctness for assignments in 2013 (M=0.91, sd=0.16) than for assignments in 2012 (M=0.84, sd=0.27).

Our previous study also found that the CH adaptive feedback treatment in Spring 2012 failed to improve either testing quality or quantity when compared to the same assignments in Fall 2011 (without the adaptive feedback system) [BE2013ii]. Consequently, with too many confounding variables in comparing students with different outcomes on different assignments with different scopes and difficulties in different years, we decided to compare only the remaining five treatments in this study: CS, DH, DS, RH, and RS.

Although the consequential design excludes the 2012 experimental treatment combination (CH), CS, RH, and DH provide insight into the individual dimensions of CH. While a balanced, full 3x2 factorial design would be ideal for within-group comparisons, it was impractical to execute such a controlled design within the constraints of the academic term. Since the study's investigation of influences on student testing behaviors is for formative design purposes, between-treatment comparisons should still offer valuable insights.

4.2.3 Results

Qualitative Investigation

When describing their initial motivations and objectives once a project is assigned, the participants concentrated mostly on reviewing the assignment's specifications and devising broad outlines and plans for completing the solution. However, their opinions diverged once discussion lead to testing strategies.

Four of the seven participants explained their habits of waiting to test until a substantial amount of the solution is complete. Three explicitly stated with confident tone that while they understood that the instructor encourages testing early, it only makes sense to them to test at the very end of their development process.

The remaining participants explained that their testing strategy varied between assignments. Meanwhile, one participant explained that she felt overwhelmed and dissatisfied with her work early in the term until she adopted an incremental testing strategy. She reported that she consequently grew more confident in her work and improved her grades considerably.

The responses on the participants' questionnaires reflected similar behaviors and attitudes. All participants reported that they more often start testing late than early in development and test after writing solution rather than before. However, reports on incremental testing were split. Four participants usually tested in large portions while two usually tested in smaller increments and one remaining participant does both equally often.

Nevertheless, on average, participants reported that in the future they expect to increasingly test early, test in small increments, and test before coding the solution. Correspondingly, they also expect to decrease testing late and testing after coding the solution. In other words, participants predicted that in the future, they will draw closer to being equally likely to exhibit TDD practices as they are to avoid them. **Table 4.2** summarizes the average ratings for all of the ratings.

Table 4.2. Summary of Questionnaire Responses

	Current		**Future**	
	Rare 1 - Often 5		Very Unlikely 1 – Very Likely 5	
	M	**sd**	**M**	**sd**
Test Early	1.86	0.63	2.43	0.90
Test Late	4.29	1.03	3.71	1.16
Small Increments	2.57	1.29	3.29	1.28
Large Portions	3.86	0.64	4.14	0.64
Test First	1.43	0.49	1.71	0.45
Test After	4.86	0.35	4.14	0.64

The participants who were confident in testing last suggested that their testing strategies were not likely to change regardless of the size of the project. One such participant wrote: "*While we've been encouraged to 'test as we go', I still feel like it makes the most sense to test a method only AFTER it's completely written.*" Others anticipated that larger or more challenging projects will require more thought and effort to devote to testing and consequently they will need to begin testing earlier.

Participants provided more granularity into their development process by drawing timelines. None of the participants illustrated any test-first behavior. Two participants represented strictly test-last strategies where the only testing took place as the last activity on the timeline. The first timeline in **Figure 4.2** provides an example of this test-last approach. However, the other five participants all demonstrated some degree of incremental testing with at least two iterations of “code a little, test a little.” Notably, three of the seven participants illustrated at least one event on their timeline where testing and either “debugging” or “fixing” problems or issues were combined as a single marker. The lower timeline in **Figure 4.2** shows a common “write tests / fix issues” activity near the end of the development process.

In addition to discussing development and testing strategies, participants also provided their opinions and experiences with Web-CAT and particularly with the adaptive feedback system. Although students expressed appreciation for Web-CAT, there were general notions that the feedback page could be more concise. When asked about what was most important, what first caught their attention, and what influenced their future behavior, participants nearly always identified the red bars that indicate incomplete correctness and coverage as focal points.

When asked specifically about the adaptive feedback, participants explained that they understood the intent to encourage more testing. However, as one participant explained when talking about the target and coverage goal: “I know why it’s there. But I know I’m supposed to get 100% coverage because that is part of the grade. I don’t need this to tell me to improve my coverage because I already know that.” Likewise, the participants described that the hints were usually important but that they did not usually purposely add tests to earn more hints; instead, when they added more tests, they did so to improve their coverage score.

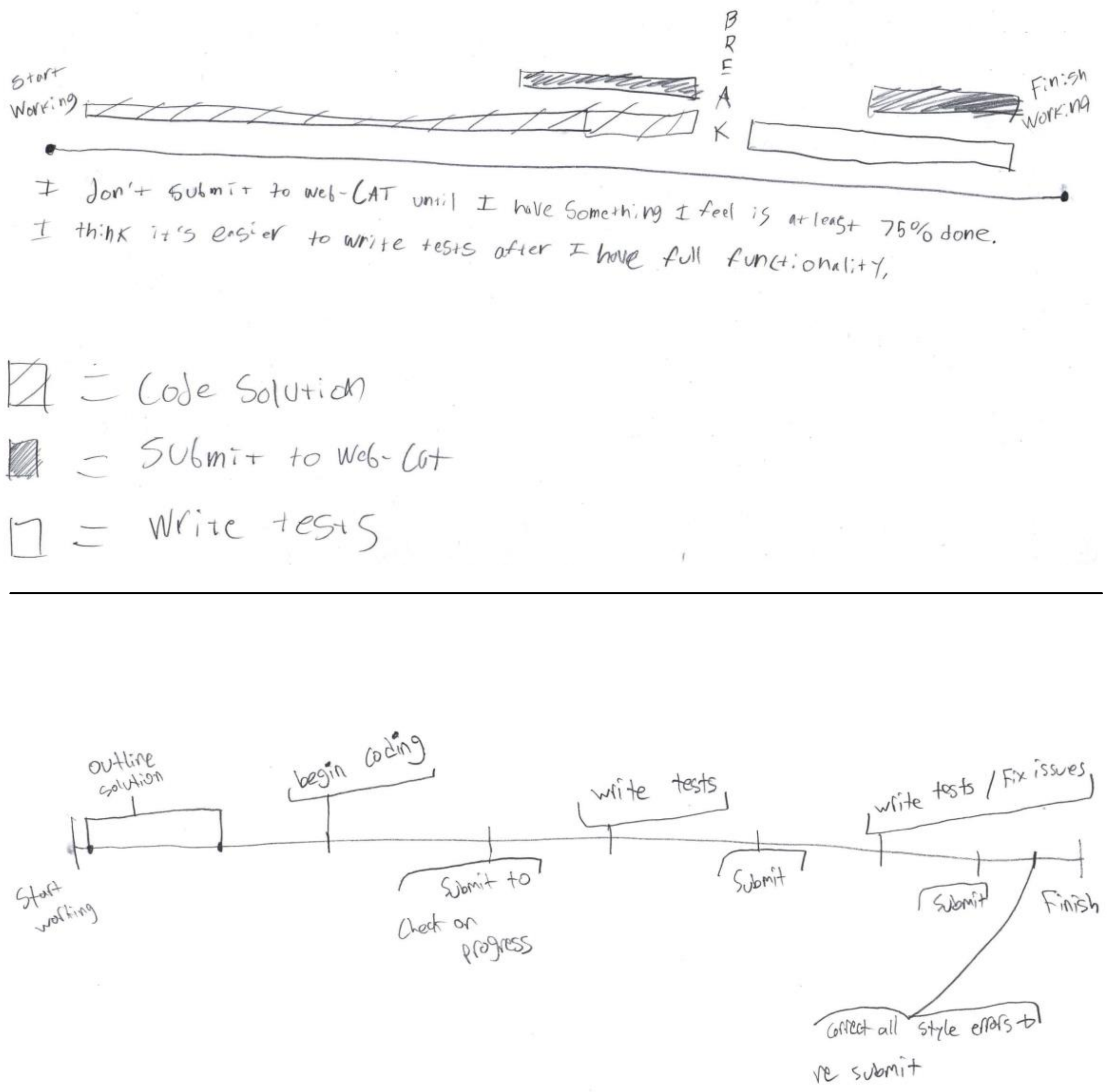

Start Working
BREAK
Finish Working
I don't submit to web-CAT until I have something I feel is at least 75% done.
I think it's easier to write tests after I have full functionality.
= Code Solution
= Submit to Web-Cat
= Write tests
outline solution
begin coding
write tests
write tests / Fix issues
Start working
Submit to Check on progress
Submit
Submit
Finish
Correct all style errors to re submit

Figure 4.2. Development Timelines Contrasting Test-Last Approach (Above) and Periodic Testing (Below)

Quantitative Evaluation

Since each of the five treatments includes a combination of two dimensions for each research question, we adjusted the significance level using Bonferonni correction [WJT1999] to counteract testing multiple hypotheses. A Wilcoxon each-pairs test compared each of the ten combinations of experimental groups (α=0.008). Likewise, a Wilcoxon each-pairs test compared each of the three different treatments (C, D, R) of the reinforcement schedule dimension (α=0.0333) independently. Finally, a Wilcoxon-Mann-Whitney test compared the two treatments (S, H) of the goal dimension (α=0.10) independently.

Since each project required a different amount of work but we want to consider each with equal weight, Δ test NCLOC was adjusted to account for difference in project sizes. Its values were divided by the mean solution NCLOC for each respective project: Project 2 (M=215.67, sd=39.66), Project 3 (M=159.18, sd=21.16), and Project 4 (M=495.75, sd=216.87). Larger projects would typically require a greater amount of test code and time spent developing. However, when making comparisons across all projects, we are concerned with the amount of code and amount of time spent relative to the size of the project. Otherwise, large projects (such as Project 4) would unduly influence the results due to their considerably larger absolute measurements, before adjustment.

First, we compared the Δ test NCLOC between all experimental groups and found no significant differences, as shown by insignificant p-values in **Table 4.3**. Likewise, similar constant (M=0.25, sd=0.25), delayed (M=0.24, sd=0.27), and random (M=0.23, sd=0.17) Δ test NCLOC demonstrated no significant difference between constant and delayed (p=0.86), constant and random (p=0.89), and delayed and random (p=0.68) pairwise comparisons. In addition, the shown (M=0.25, sd=0.27) and hidden (M=0.22, sd=0.19) goal treatments yielded no significant difference (p=0.39).

Table 4.3. Wilcoxon Each-Pairs Comparison of Δ Test NCLOC

(M,sd)	**CS**	**DH**	**DS**	**RH**	**RS**
CS(.25,.25)		p=.73	p=.49	p=.73	p=.85
DH(.21,.22)			p=.13	p=.17	p=.40
DS(.27,.33)				p=.56	p=.41
RH(.23,.16)					p=.67
RS(.22,.18)					

After finding no differences in quantity of test code, we compared the test code quality. **Table 4.4** shows the pairwise comparisons between each experimental group.

Table 4.4. Wilcoxon Each-Pairs Comparison of Δ Coverage

(M,sd)	**CS**	**DH**	**DS**	**RH**	**RS**
CS(.35,.36)		p=.02	p=.96	p=.64	p=.22
DH(.20,.26)			p=.01	p=.03	p=.21
DS(.35,.35)				p=.66	p=.20
RH(.31,.32)					p=.34
RS(.24,.29)					

While no pairwise comparisons were significantly different at the α=0.008 level, DS (M=0.35, sd=0.35), CS (M=0.35, sd=0.36), and RH (M=0.31, sd=0.32) approached significantly greater improvement in coverage (p=0.01, 0.02, and 0.03, respectively) than DH (M=0.20, sd=0.26). When comparing the Δ coverage for the reinforcement schedules, there was no difference ($p=0.16$) between constant ($M=0.35$, $sd=0.36$) and delayed ($M=0.27$, $sd=0.31$) reinforcement. There was also no significant difference between constant and random ($M=0.28$, $sd=0.31$; $p=0.37$) or between delayed and random ($p=0.42$). However, shown goals ($M=0.32$, $sd=0.34$) nears significantly greater ($p=0.10$) Δ coverage than hidden goals treatment ($M=0.26$, $sd=0.30$).

4.2.4 Discussion

Both quantitative and qualitative investigations verify suggestions from previous studies that students are hesitant to adhere to test-first procedures. However, the group interviews in particular revealed unique insight into students' reasons for their behaviors. The students who expressed particularly strong reluctance to follow test-driven development (TDD) also happened to be students who were especially confident in their programming skills. Still early in their computer science education, they did not seem to have been challenged enough by programming assignments to warrant changing their habits of concentrating on coding the solution alone.

While it is encouraging to hear one participant find her confidence by using TDD, it was even more revealing to examine students' development timelines. The proximity of testing and its frequent association with debugging suggest that within students' mental models, testing is a process for fixing problems rather than proactively avoiding them. In order for students to adopt different behaviors, this mental model would have to change.

Quantitative analysis of the adaptive feedback system does not reveal significant differences between reinforcement schedules on influencing student testing. However, with the conservative Bonferonni correction and the constrictions of only leveraging between-group comparisons, it may be worthwhile to continue exploring reinforcement schedules.

On the other hand, while students commented that the salient testing goal did not change their behavior, the change in coverage after seeing the coverage goal was greater than in its absence at a nearly significant level. While it is unlikely that any rewards or gamification techniques will ever influence students' behaviors as much as the expectations set by graded requirements, we observed a strong impact that simple red-to-green status bars have on students' attention.

Dissonant visual cues such as error icons, incomplete progress bars, or unattended notifications may implicitly change student behaviors as they are compelled to resolve the dissonance. Correspondingly, introducing similar visual cues of incomplete testing coverage within students' integrated development environment has potential for motivating change. Future work needs to explore such implicit interventions, particularly if they can be delivered directly to the students' development environment instead of relying on them to actively seek feedback.

4.3 Responses to Adaptive Feedback

We have published the work described in this section in the article, “Responses to adaptive feedback for software testing” and presented it at the conference on Innovation and Technology in Computer Science Education (ITiCSE) [BE2014i].

4.3.1 Background

Melnik and Maurer acknowledged that adopting Agile methods may be more challenging in an academic setting than it is in industry. Consequentially, they surveyed students’ opinions of different aspects of eXtreme Programming (XP), including test-driven development (TDD). They found generally positive views of all aspects of eXtreme Programming from 240 respondents, representing a variety of demographics with differing degrees of experience and exposure to XP [MM2005]. In addition, they discovered a weak positive correlation between attitudes toward TDD and students’ ages.

However, they also found that some students struggled to think with a test-first approach. They reasoned this difficulty may be due to TDD “almost like working backwards” by drawing attention to documenting design early through writing unit tests [FAB+2003]. Similarly, Janzen and Saiedian compared the opinions and acceptance of TDD between novice and mature developers in computing courses. They found that mature developers are more willing to accept TDD. Furthermore, students were significantly more likely to choose to follow TDD in the future after having tried it [JS2007].

To aide teaching TDD, Spacco and Pugh leveraged Marmoset [Spac2013], a rich submission and automated grading system. Students benefit from receiving prompt, online feedback including how their code performs against the tests. Despite the emphasis and prompt feedback on testing, Spacco and Pugh recognize that many students still favor a “test-late mentality of writing their implementation and then testing it at the very end” and call for a need to design incentives to motivate students to test early [SP2006].

Positive reinforcement can be a powerful tool in motivating new behaviors. Schedules of reinforcement have shown to encourage target behaviors in both games and learning environments [LKL+2011]. Linehan suggests a model for leveraging Applied Behavior Analysis to motivate target behaviors through a process of measuring performance, analyzing performance, presenting feedback, and defining a rewards schedule that coordinates with the target behavior. Likewise, we have the unique opportunity to measure, analyze, and use reinforcement and punishment to influence students’ development processes.

To modify human behavior, operant conditioning usually depends on four mechanisms: stating the goal, tracking the behavior, rewarding target behavior with positive reinforcement, and discouraging deviations from the target behavior [PC2004]. For example, providing rewards as incentives for demonstrating the target behavior provides positive reinforcement. Then, when the subject does not demonstrate the target behavior, the rewards can be removed (negative punishment) and the subject reprimanded (positive punishment).

Meanwhile, we have been teaching TDD in CS1 and CS2 courses. In a preliminary study of a five-year data set of snapshots of students' work, we found positive correlations between indicators of incremental testing and consequential outcomes. Specifically, we identified two measurements of quantity of testing average test statements per solution statements (TSSS) and average test methods per solution method (TMSM) with small but statistically significant correlations with functional correctness and test coverage. Similarly, average test coverage across all snapshots for an assignment was positively correlated with final functional correctness [BE2012i]. However, despite its advantages, we also witnessed some students who resisted adhering to TDD.

In a separate study, we investigated students' attitudes toward the test-first and incremental unit testing aspects of TDD [BE2012ii]. Similar to reports in related literature, we discovered that students generally appreciated the value of testing but were apprehensive to adopt test-first habits. While students valued an incremental unit-testing approach, most students did not follow strict test-first procedures. Likewise, we identified a close relationship between students' perception of how helpful these aspects of TDD are and how likely they are to adhere to them.

This relationship is likely reciprocal in that expecting TDD to help should make students more likely to adhere; likewise, adhering to TDD should advocate students' appreciation of its benefits. However, most students reported that did not persistently write tests in small increments nor did they test first. For these reasons, we recognized a need to better understand students' development processes and investigate approaches to encouraging adherence to TDD.

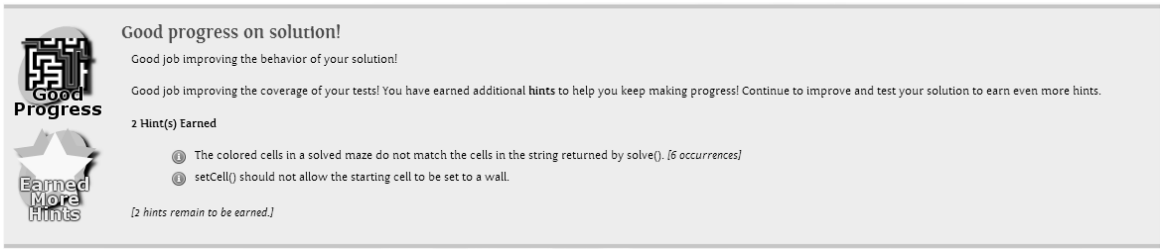
4.3.2 Method

A Model for Adaptive Feedback

To observe students' development processes, it is necessary to gain insight into changes in their work over time. By using Web-CAT [9]—an automated grading system—to collect student submissions and provide rapid evaluation of their performance, students are encouraged to submit several versions of their work as they refine their assignments. Among other features, Web-CAT evaluates students' code on its correctness and coverage. Correctness is determined by the percent of instructor-provided tests (obscured from students) successfully passed by the student's solution. Coverage is determined by the amount of solution code evaluated by the students' own unit tests. Upon submitting their work, students promptly receive results of their correctness and coverage scores. Students may submit their work unlimited times, without penalty, until the assignment deadline.

Each time students submit and receive feedback, there are opportunities to assess their adherence to incremental testing methods and trigger interventions to encourage the desired behavior. Ideally, incremental testing would be demonstrated by maintaining high (at or near 100%) coverage while the correctness gradually increases. Consequently, we designed the system to reinforce this behavior and to correct students who deviate from it.

Our adaptive feedback system supplements Web-CAT's correctness and coverage assessment by monitoring progress from one submission to the next. Students receive positive reinforcement through images and brief messages acknowledging improvements in their solution and/or testing, as shown in **Figure 4.3**. When their submissions do not demonstrate improvement, the feedback encourages them to improve their testing by offering additional incentives. In particular, students receive hints about how to improve their solution as a reward for improving the thoroughness of their testing. **Figure 4.3** illustrates reinforcing feedback with two hints displayed.



Good progress on solution!

Good job improving the behavior of your solution!

Good job improving the coverage of your tests! You have earned additional **hints** to help you keep making progress! Continue to improve and test your solution to earn even more hints.

2 Hint(s) Earned

- The colored cells in a solved maze do not match the cells in the string returned by solve(). *[6 occurrences]*
- setCell() should not allow the starting cell to be set to a wall.

[2 hints remain to be earned.]

Figure 4.3. The Web-CAT Plugin Displays Adaptive Feedback with Positive Reinforcement and Hints.

The system adaptively caters hints to help students correct problems with their solutions that are identified by failed instructor tests. Each instructor unit test includes a hint message to provide some guidance about why it failed. Closely related instructor tests may generate identical hint messages, but duplicates are combined and the collection of hints is sorted so precedence is given to hints with more occurrences.

Students earn their first hint from their first submission that demonstrates some progress on both the solution and testing with non-zero correctness and coverage scores. To earn additional hints on subsequent submissions, students have to meet a minimum threshold of coverage (initially ≥85%) to demonstrate they are testing their solution substantially. Given the minimum coverage is met, students can earn hints by either maintaining 100% coverage and making changes to their solution code, or by improving their coverage over the previous submissions. Following this model, students are required to begin testing in early submissions and continue to meet progressively higher coverage requirements as they progress.

If a student earns hints on sequential submissions, she may receive the same hints in both submissions' feedback. On its face, this approach may not seem to reward the student for earning hints. However, it allows the student to track the flaws in her assignment. For example, if a previously seen hint is replaced by a new hint, one may falsely conclude that the bug that generated the hint has been resolved. Instead, earned hints are only dismissed once their corresponding instructor test passes.

If students deviate from the incremental testing process, they do not receive hints on their submissions until they demonstrate sufficient coverage again. The system also controls for potential attempts to get additional hints by artificially manipulating measurements of progress. For example, the system records the highest correctness score achieved so far to prevent students from deleting or sabotaging their solution in one (worse) submission only to give the illusion of improvement by reverting to the previous (better) solution in the next submission.

Intervention Design

We integrated our adaptive feedback model into Web-CAT for students to submit their programming assignments. During the Spring 2013 academic semester, we introduced CS2 (Software Design and Data Structures) students to the adaptive feedback system. Of the 128 students who sat the final exam, 84 (66%) provided written consent to include their data in our analysis. Assignments included: (1) maintaining a list of digital photo metadata, (2) solving a maze with two-dimensional arrays, and (3) implementing both array- and link-based queues. In addition, students from each of the five sections of the course used different variations of the adaptive feedback system. Treatments included combinations of consistent or intermittent reinforcement and either with- or without- a visible goal for improving test coverage. The consistent reinforcement treatment rewarded students with hints every time they met the incremental testing criteria while intermittent reinforcement treatments only rewarded the students with hints at most once per hour or at random chance (after meeting the same criteria). The goal treatments either always showed the coverage threshold necessary to earn a hint or always obscured it. In a separate study, we analyzed the different treatments with inconclusive long-term outcomes [BE2013ii][BE2014ii]. However, in this study we concentrate on the immediate responses students demonstrate after receiving hints.

Evaluation

To observe students' short-term behavioral responses, we identified whether the feedback for each submission: received additional hints, maintained the same number of hints (but earned no additional hints), or received no hints. Accordingly, we inspected changes in students' work in their successive submissions. In particular, we considered the difference in the amount of code between the latter and former submissions. We recorded changes (Δ) in non-comment lines of code (NCLOC) for the solution and the tests independently. Likewise, we measured the change in test coverage as well as changes in the number of assertions in the students' test code.

Since the adaptive feedback system follows protocol for operant conditioning to encourage incremental testing, we suspected students to increase their testing efforts after receiving positive reinforcement. Specifically, we made the following hypotheses:

1. After receiving additional hints, students will be more likely to make any changes in their test code than after they do not receive additional hints.
2. After receiving additional hints, students will add significantly more test code than after they do not receive additional hints.
3. After receiving additional hints, students will add significantly more test assertions than after they do not receive additional hints.
4. After receiving additional hints, students will improve coverage significantly more than after they do not receive additional hints.

To address hypothesis 1, we first categorized types of change to test code. Adding new test methods constituted major changes, adding only new test NCLOC signified moderate changes, and any changes that did not require changing the number of test NCLOC were minor changes. Using chi-squared test, we compared the distribution of these types of change after receiving: additional hints, same hints, and no hints. To test hypothesis 2, we used a Wilcoxon signed-rank test to compare Δ test NCLOC in submissions after incidents of additional, same, and no hint reinforcement. To control for potential differences between individuals' behaviors, we also performed a within-subject analysis of variance using Friedman's test for non-parametric repeated measures. We followed the same Wilcoxon and Friedman test design to compare affects to the Δ test assertions for hypothesis 3 and the Δ coverage for hypothesis 4.

4.3.3 Results

Across three programming assignments during the semester, we analyzed 3115 submissions. Since students' first submissions (170 out of 3115 submissions) to each assignment do not represent responses to the adaptive feedback intervention, their data was not included in comparing changes in code. 2169 (~74%) of the submissions followed submissions with no hints, while 524 (~18%) received the same number of hints and 252 (~9%) received additional hints. As described in the previous section, we categorized types of changes to test code as minor (1429, ~49%), moderate (856, ~29%), and major (650, ~22%). **Figure 4.4** illustrates the distribution of how additional, same, and no hint groups responded with minor, moderate, and major test changes.

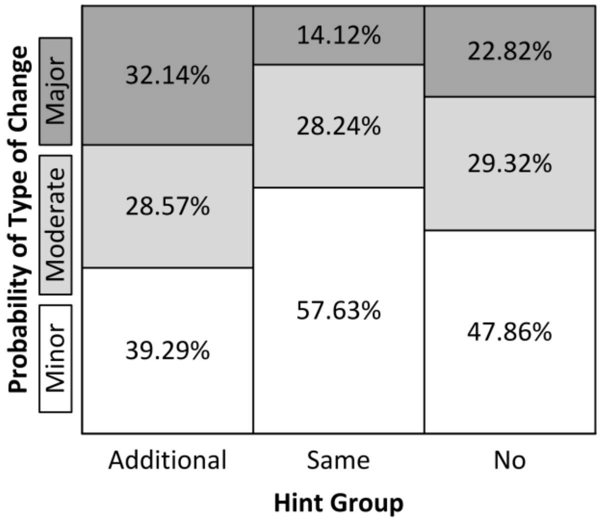


Figure 4.4. Probability of Types of Response to Hints

We performed a chi-squared test to examine the relation between hint group and the type of change response. The relation between these variables was significant X2(4, N=2945)=41.45, p<.0001 and there was no significant interaction with the subject variable X2(2, N=2945)=3.53, p=.15. These results support hypothesis 1 that the greater likelihood of making major test changes after receiving additional hints is not due to chance.

We performed a Friedman test and found that the hint group variable had a significant effect on Δ test NCLOC F(2,1)=15.48, p<.0001. We used a Wilcoxon test to compare Δ test NCLOC between each pair of hint groups. We also used the Bonferroni correction to adjust the critical p value (α=0.0167) for multiple comparisons. The additional hints (M=9.52, sd=23.80) group responded with significantly greater Δ test NCLOC (p<.0001) than same hints (M=1.89, sd=18.51) and no hints (M=5.31, sd=17.59, p<.01). No hints responded with significantly greater Δ test NCLOC (p<.0001) than same hints. To test hypothesis 3, we performed a Friedman test and found that while the hint group variable had no significant effect on Δ test assertions F(2,1)=7.79, p=.67. The Δ test assertions were low for additional hints (M=0.54, sd=3.26), same hints (M=0.35, sd=3.01), and no hints (M=0.36, sd=3.12).

However, we tested hypothesis 4 by performing a Friedman test on Δ coverage and found a significant effect of hint group on Δ coverage F(2,1)=13.39, p<.0001. To compare differences between each pair, we performed a Wilcoxon test (α=0.0167) and found that additional hints (M=0.03, sd=0.13) responded with greater Δ coverage (p<.0001) than same hints (M=-0.01, sd=0.05) but was not significantly greater than no hints (M=0.02, sd=0.11); moreover, no hints responded with greater Δ coverage (p<.0001) than same hints.

The initial statistical analysis provided support for hypotheses that students respond to additional hints as reinforcement with greater likelihood of adding test code and with greater amounts of test code. While tests did not support the hypotheses that test assertions would also increase, they did find that coverage increases. We also observed a recurring pattern across all the measurements where additional hints had the highest means while some hints had the lowest. Curious from this observation, we also investigated comparisons in other changes in students' code with post-hoc tests.

We performed a Wilcoxon test and found that after receiving additional hints (M=5.94, sd=23.44), students responded with greater (p<.01) Δ solution NCLOC than same hints (M=1.42, sd=13.81) and no hints (M=1.70, sd=17.09, p<.0001) while no hints approached significantly greater Δ solution NCLOC (p=.0293, α=0.0167). Similarly, we compared the elapsed time (in minutes) from the previous (intervention stimulus) to subsequent (response) submissions and found that additional hints (M=307.21, sd=1120.52) responded after more time (p<.0001) than same hints (M=111.45, sd=485.15) and no hints (M=227.73, sd=931.13). There was no significant difference (p=.07) between the elapsed time of no hints and some hints.

Consequently, we see that after additional hints, students produce more test code and more solution code but also take more time to do so. Therefore, it is possible that the additional code written was an artifact of taking more time than rather than as a response to positive reinforcement. Moreover, we found that elapsed time between submissions varied greatly. As a result, we categorized elapsed time by quartiles: quick included elapsed time under 3.47 minutes, short included longer elapsed time but within 8.85 minutes, medium included elapsed time greater than 8.85 but less than 31.93 minutes, and long included the remaining quartile of submissions with greater elapsed time. **Table 4.5** shows the Δ test NCLOC (with mean and standard deviation) for hint groups with subgroups for elapsed time.

Table 4.5. Change in Test NCLOC by Elapsed Time

	Hint Group (M,sd)		
	Additional	**Same**	**No**
Quick	-0.15, 20.75	-0.28, 7.70	-0.16, 14.89
Short	2.67, 10.51	0.16, 11.28	*3.23, 12.80*
Medium	*7.20, 15.06*	3.58, 9.84	*5.87, 11.66*
Long	*21.70, 33.34*	5.12, 36.41	*12.34, 24.87*

We see that in quick resubmissions, the average Δ test NCLOC for each hint group is less than one line of code changed. That might be expected since the minimal time elapsed between submissions suggests that students gave little to no time to read the Web-CAT results before making small changes and resubmitting.

We performed Wilcoxon each-pairs comparisons between hint groups with each elapsed time interval independently. After quick time elapsed there is no significant difference between additional hints and same hints (p=.91) or no hints (p=.63) and no significant difference between same hints and no hints (p=.18). After short time elapsed, there is no significant difference between additional hints and no hints (p=.99) but approaches greater Δ test NCLOC than same hints (p=.04). No hints demonstrates significantly greater (p<.01) Δ test NCLOC than same hints. Likewise, after medium time elapsed, there is no significant difference between additional hints and no hints (p=.93) but approaches greater Δ test NCLOC than same hints (p=.04). No hints demonstrates significantly greater (p<.0001) Δ test NCLOC than same hints. After long time elapsed, additional hints has greater Δ test NCLOC than no hints (p<.01) and same hints (p<.0001) and no hints also has greater Δ test NCLOC than some hints (p<.0001).

One should also note that since the incidents of additional and same hints groups are considerably smaller when considering each interval independently (quick: n=33 and 134, short: n=70 and 154, medium: n=70 and 132, long: 79 and 104, respectively) so it is more difficult to find significant results. Nevertheless, we continue to observe a trend where same hints group responds with the fewest changes in their test code.

One possible explanation for most of the tests showing the greatest changes in test code after receiving additional hints is that students may be in a period of their development where they are mostly concentrating on testing. By improving their test coverage, they earn the additional hints and then may submit to observe their progress and then continue adding test code until they are satisfied with their test coverage. Such a pattern would suggest that receiving hints may have no influence on their behavior.

To investigate this case, looked exclusively at instances when students exhibit the target behavior (as identified by the adaptive feedback system) and compare their responses after being rewarded with new hints to those when no additional hints are received. Students may not receive additional hints despite earning them either because the system had no more hints to provide (because the student's solution did not have sufficient flaws) or because the reinforcement schedule treatment (described in section 3.2) withheld reinforcement on that particular submission. We identified these instances as unacknowledged (n=153, no hints shown despite earning some) or unrewarded (n=179, some hints shown but not additional, as earned) groups. By comparing these groups, we can observe the impact a hint reward has on students continuing the target behavior on subsequent submissions after exhibiting it for the previous submission.

We compared responses after additional hints to those after unacknowledged and unrewarded groups. While correcting the threshold for significance when making multiple comparisons (α=0.0167), we performed each-pairs comparison using the Wilcoxon test. We found that additional hints (M=9.52, sd=23.80) responded with significantly greater Δ test NCLOC (p<.0001) than unacknowledged (M=1.57, sd=13.99, p<.0001) and unrewarded (M=0.55, sd=8.06) submissions. There was no significant difference (p=.36) between unacknowledged and unrewarded groups. In these cases, students exhibited similar behavior by initially satisfying the adaptive feedback system's criteria for incremental testing, but responded differently depending on whether or not they received additional hints.

4.3.4 Discussion

The statistical analysis supported three hypotheses:

1. After receiving additional hints, students will be more likely to make any in their test code than after they do not receive additional hints.
2. After receiving additional hints, students will add significantly more test code than after they do not receive additional hints.
4. After receiving additional hints, students will improve coverage significantly more than after they do not receive additional hints while tests for one hypothesis was inconclusive:
3. After receiving additional hints, students will add significantly more test assertions than after they do not receive additional hints.

In general, the changes in the number of test assertions from one submission to the next were small (M=0.38, sd=3.11) so change in test code NCLOC (M=5.06, sd=18.45) offers more substantial alterations and consequently may be a better indicator of the short-term responses to adaptive feedback stimuli. While there are many factors that may influence changes students' code from one submission to the next, we found that after receiving rewards reinforcing incremental testing, their responses usually exhibited greater increases in test code. Even when comparing only submissions following demonstrations of incremental testing, those who received positive reinforcement responded by continuing to add more test code than those who did not receive rewards.

Likewise, when controlling for the amount of time elapsed between submission, we found that responses to receiving additional hints often increased test code more than those after receiving no hints. In turn, those receiving no hints often increased test code more than those who received hints, but no more than they had previously earned. Upon initial consideration, this pattern may seem counter-intuitive. To the contrary, the responses are consistent with principles of operant conditioning.

Receiving additional hints is an example of positive reinforcement—adding a stimulus that rewards a behavior and consequently increases the frequency of that behavior. Receiving no hints is an example of negative punishment—removing a pleasing stimulus (hints) to decrease the behavior. However, by neither adding nor removing hints, the behavior is neither incentivized nor discouraged. Consequently, receiving hints may encourage students to continue their incremental testing while removing hints may dissuade them from behaviors that do not exhibit incremental testing.

We considered that the insight provided by the contents within hints might have a greater impact on changes in students' test code than whether or not their behavior received reinforcement. However, it should be noted that hints only describe flaws in the students' solution code. Students may be able to extrapolate missing test case(s) from a hint, but both hint selection and wording concentrate on students' solution code and disregard their test code (e.g. "setCell() should not allow the starting cell to be set to a wall." for the maze solver assignment). Specific feedback on test coverage—both by line of code and as a percentage of all the code—is available regardless of whether a hint is provided.

More importantly, at the moment of the resubmission, those who responded to additional hint stimuli did not necessarily have greater insight from more hints. For example, a student who previously earned three hints and satisfied the criteria to earn a fourth—but did not because of the reinforcement strategy—would have seen more hints than another student who only just received his first new hint. If we consider every unique hint that a student had received for an assignment up until the stimuli response, responses to receiving additional hint (M=1.81, sd=1.25) had actually seen fewer hints (p<.0001) than those who earned an additional hint but did not receive one (M=2.30, sd=1.29).

Nevertheless, we should also acknowledge that the presence (or absence) of hints were not the only stimuli received upon submission to Web-CAT. Since Web-CAT is an automated grading system, it also provides indicators of the student's current score, along with some other analysis of their code. While hints appear to play a role in influencing students' immediate behavior, the scores that determine their grades also motivate them considerably [BE2014ii]. Grade calculation for the programming assignments in our study included students' test coverage, but did not necessitate adherence to an incremental testing process.

Conclusion

In this study, students received rewards from an adaptive feedback system after exhibiting behaviors conducive to an incremental testing process. Rewards included social acknowledgement via digital "badges" and an encouraging message along with hints that guided students toward fixing flaws in their code. From 3115 submissions to the adaptive feedback system over the duration of an academic term, we analyzed changes in students' code from one submission to their subsequent submission.

We compared changes in test and solution non-comment lines of code (NCLOC), number of test assertions, and test coverage. Statistical analysis suggested that after receiving additional hints as rewards, students responded with significant increases in the amount of test code and coverage. Observations of increases in the amount of test code persisted when considering within-subject comparisons as well as when accounting for time elapsed between submissions and for behaviors exhibited before the previous submission.

Our findings are consistent with operant conditioning techniques of increasing target behavior with positive reinforcement and discouraging undesired behaviors with negative punishment. Meanwhile, lack of either overt reinforcement or punishment often resulted in fewer tests added than either positive reinforcement or negative punishment. Consequently, the study provides evidence for significant differences in short-term responses to rewards offered by adaptive feedback systems. However, despite invoking short-term responses, our accompanying studies found no impact on long-term behavioral change nor final outcomes of software quality [BE2013ii][BE2014ii]. Future work is necessary to build upon the short-term effects of adaptive reinforcement to support the ambitious goal of affecting behavioral change on the complex task of as software development.

Chapter 5

Models of Testing Strategies and Outcomes

This chapter concentrates on the third research objective: "**Characterize software testing strategies demonstrated by students and evaluate their consequential outcomes.**" The first section (*5.1 Effective and Ineffective Software Testing Behaviors*) describes a study that analyzes a large dataset of students' programming assignments to identify trends in testing behaviors that result in fewer bugs and better tests. The subsequent section synthesizes findings from previous chapters with the general trends identified in Section 5.1 to propose a new framework for scaffolding testing behaviors.

5.1 Effective and Ineffective Software Testing Behaviors

We have published the work described in this section in the article, "Effective and Ineffective Software Testing Behaviors by Novice Programmers" and presented it at the International Computing Education Research (ICER) conference [BE2013i].

5.1.1 Background

This paper describes a study conducted on a 5-year (10-semester) dataset involving introductory programming assignments completed by 883 unique students. After being taught TDD, students were required to write software tests as part of each solution, and an automated grading system was used to collect their work. Students were allowed to make multiple submissions to receive feedback and refine their work as they developed their solution. As a result, the dataset includes a total of 49,980 separate attempts at the programming assignments given to the students in the study, representing a series of snapshots of the work-in-progress of each student. By examining relationships between when students add software tests to their projects and how thoroughly they test their own code, this study shows that there is a positive relationship between early testing and more positive student outcomes on programming assignments, including better scores and reduced likelihood of turning in work late.

5.1.2 Method

Data Collection

In introductory computer science courses at our university, students submit their programming assignments to Web-CAT. After submitting, students receive prompt feedback, including scores for the correctness of their code and their test coverage. The percentage of instructor-provided reference tests that pass when run against a student's code determines the correctness score. Meanwhile, test coverage is calculated based on how much of the student's code—a composite of methods, statements, and branches—is executed by their tests.

After receiving feedback from Web-CAT, students may revise their code and resubmit as many times as they like without penalty. Their assignment grade is determined by their final submission. The score calculation includes: correctness, test coverage, and style. Instructors and/or assistants often assign a portion of each program grade by hand by evaluating documentation and design; however, we exclude this aspect of assignment scores in our analysis to avoid issues with subjectivity and consistency between human graders. In addition, assignments are sometimes submitted after the deadline, which may involve a score penalty depending on a course's policies regarding late work. However, late submission penalties are disregarded in our analysis of student program scores to avoid misrepresenting the actual quality of the solution and software tests produced.

To perform a comprehensive analysis of student testing behavior, we used a five-year dataset of programming assignment submissions from our CS1 course: Introduction to Object-oriented Development I. The dataset includes each full-length semester from Spring 2004 to Fall 2008. We excluded data from summer semesters because summer schedules are abbreviated and consequently the courses differ considerably in pace and structure.

The ten-semester data set includes 49,980 submissions for 3,715 scored assignments, with an average of over 12 submissions by each individual student completing each separate assignment. The data reflects the work of 883 unique students. While several assignments were used for more than one semester, both the assignment instructions and number of assignments varied between semesters.

To observe students' behaviors during development, we analyzed each of their submissions for each assignment. In addition to quality of code—as measured by correctness and test coverage—each submission also included data on the time of submission, amount of solution code, and amount of test code. In particular, we concentrated on the following metrics:

- NCLOC: Non-comment lines of code, separated into lines that are part of the student's solution and lines that are part of the student's software tests.

- Time Remaining: The amount of time between when a submission was made and the assignment deadline. Negative values represent submissions made after the deadline.

- Time Elapsed: The amount of time between the students' first submission for that assignment and the current submission in question.

- Relative Worktime: The amount of time elapsed, expressed as a percentage of the total duration over all of the student's submissions for an assignment. Zero- and one-values represent the first and final submissions by that individual, respectively. This metric disregards the relationship between submission time and the assignment deadline. Instead, it represents the progression of time within the workflow of development.

Since assignments vary in scale and complexity between courses and semesters, the amount of code varies greatly between assignments; the final submission NCLOC mean is 587.6 and standard deviation is 729.6. In addition, this study concentrates on *testing* behaviors so we are more concerned with the amount of test code compared to the size of the solution at any given time.

Accordingly, we concentrate on the test NCLOC *relative* to the amount of solution NCLOC. Low relative NCLOC (nearing zero) indicates a lack of test code relative to the amount of solution code that has been written. Meanwhile, high relative NCLOC indicates that the student has developed comparable amounts of both test and solution code. In some cases, students produce more test code than solution code and consequently have a relative NCLOC value greater than one.

By capturing multiple submissions per assignment, snapshots of students' development provide insight into their behaviors over time. We analyzed student submissions over time and identified significant milestones in their development processes. The **first submission** within an assignment offers a preliminary glimpse of a student's code. While students often write a considerable amount of code before their first submission, it is the earliest available snapshot of their work and comparisons to later submissions indicate changes and behaviors that occur as the student continues to refine his or her work.

Since students were taught test-driven development (TDD), which encourages early and incremental testing, we identified when in their development they first achieved substantial test coverage (at least 85% of their code at that moment is covered) on each assignment. In this paper, we refer to this submission as the **test threshold milestone**. When students follow TDD strictly and write quality tests, their test threshold milestone should take place very early in development. However, postponing testing pushes the test threshold milestone to later submissions, perhaps even to a student's final submission.

Students sometimes continue to submit their work after they have achieved their highest correctness score. This occurs because their score reflects more than just correctness, and other aspects of their solution (or testing) may still need improvement. We refer to the first time they achieve their highest correctness as the **maximum score milestone**. Some submissions following this milestone may represent attempts (but failures) to raise their correctness score. However, some students continue to submit after achieving perfect correctness scores, usually in an attempt to improve another aspect of their grade beyond correctness.

Finally, a student's **final submission** marks the milestone of completing the assignment. This milestone is particularly valuable in observing students' qualitative and quantitative outcomes—final correctness and coverage in particular. Additionally, by comparing an earlier submission to the final milestone, we can observe students' progress relative to their completed work.

Grouping Data

To extract effective and ineffective behaviors, we must first identify assignments with positive and negative outcomes. First, we investigate the final correctness score of each assignment to distinguish good- and poor-quality solutions. Our courses use a 10-point graduated letter-grade scale, where A/B/C/D/F grades are designated by 90/80/70/60% and below, respectively. We identified A/B correctness scores (80% and above) as good outcomes and C/D/F (below 80%) as poorer outcomes.

Often, researchers compare behaviors of higher-performing students with those of lower-performing students. However, it would be hasty to make conclusions about the effects of behaviors based only on correlations with outcomes. Outcomes may reflect differences in undetected behaviors or other external factors. For example, students who score well on assignments may devote more time to their projects or demonstrate other "good work habits." Meanwhile, students with worse scores may exhibit "bad work habits" such as procrastinating or neglecting the amount of time a project requires. However, an entirely different type of behavior (such as the adherence to effective software engineering strategies) may result in the differences in their outcomes as well.

To address this concern, we grouped students by how they performed on all of their assignments. The A/B group includes students who earned correctness scores of 80% or better on each of their assignments. Likewise, the C/D/F group consists of students who never scored 80% or better on any of their assignments. Lastly, the varied-performance group submitted at least one assignment earning 80% or better and at least one assignment below 80%. **Figure 5.1** shows the distribution of assignment correctness scores.

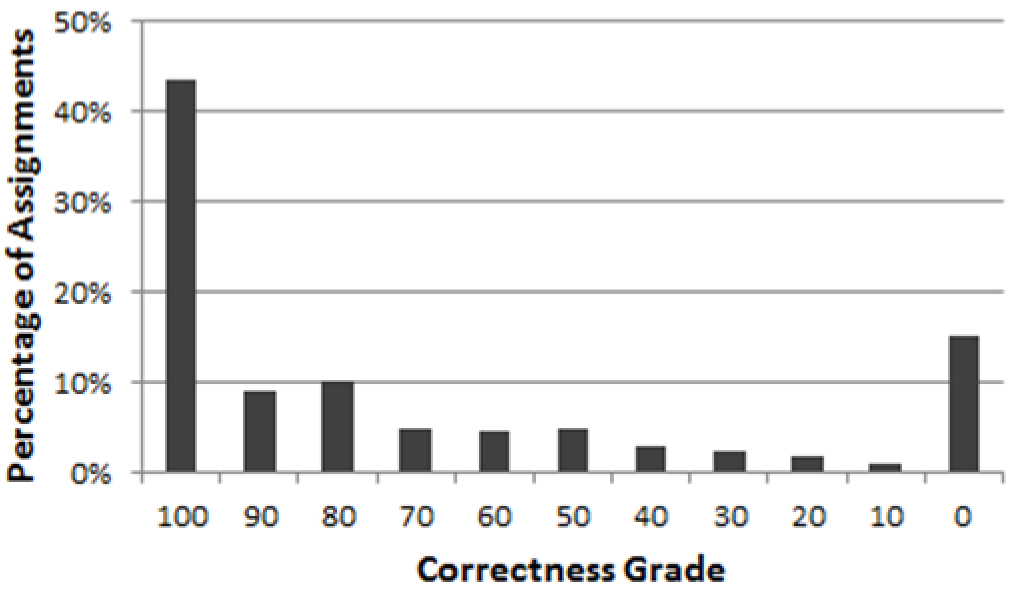


Figure 5.1. Distribution of Final Correctness Scores for All Assignments

Of the 883 unique students, 307 (35%) belong to A/B group, 170 (19%) to C/D/F group, and 406 (46%) belong to varied-performance group. **Table 5.1** summarizes mean outcomes (correctness, coverage, relative NCLOC, time remaining, and time elapsed) of each of these groups. By concentrating on the varied-performance group, we can perform within-subject comparisons to identify behavioral differences between when a student scored well and when he/she performed poorly.

Table 5.1. Summarized Final Outcomes, With Students Grouped By Their Performance on All Their Assignments

	Students	Completed Programs	Correctness (M, sd)	Coverage (M, sd)	Test:Solution Relative NCLOC (M,sd)	Time Remaining (Hrs) (M, sd)	Time Elapsed (Hrs) (M, sd)
A/B only	307	1324	0.97, 0.06	0.95, 0.06	0.92, 0.45	30.72, 64.15	37.82, 48.71
Varied	406	1891	0.76, 0.33	0.87, 0.20	0.77, 0.40	18.97, 61.28	28.56, 42.65
C/D/F only	170	500	0.27, 0.29	0.64, 0.31	0.53, 0.35	6.74, 46.48	21.92, 34.94
Total	883	3715	0.76, 0.34	0.87, 0.21	0.79, 0.43	21.51, 61.07	30.97, 44.33

We compared behaviors of assignments earning A/B and C/D/F correctness outcomes using an unbalanced, partial-factorial design. Assignment correctness group (A/B vs. C/D/F), semester, and subject (individual student) were the factors in a 2x10x406 experimental model. Analysis of variance was performed between groups, but within-subjects. Since each semester used a unique set of assignments, students only enrolled in one semester, and students had varying numbers of A/B or C/D/F scores, subjects could not appear in each combination of factors. Although the unbalanced, partial-factorial design weakens the power of the statistical results, the statistical tests also benefit from the dataset's large size.

After investigating varied-performance, within-subject differences, we also compared trends of the consistently scoring A/B and C/D/F groups. By comparing when these groups start and finish their development and how their testing changes over that time, we can isolate behaviors specific to well- and poor-performing students.

Furthermore, we examined the testing metrics at the development milestones of the varied-performance group. There is a clear bimodal distribution of the relative worktime of the testing threshold milestone. In other words, there are distinct periods within an assignment's development when students achieve substantial coverage. We partitioned test development time into four groups: early, intermediate, late, and neglectful. The "early testers" reached the testing threshold within the first 20% of their relative worktime. "Late testers" reached the same milestone within the last 20% of their relative worktime. "Intermediate testers" reached the milestone between those groups, while "neglectful testers" never achieved 85% coverage. **Figure 5.2** shows the distribution and partitioning of the testing threshold relative worktime.

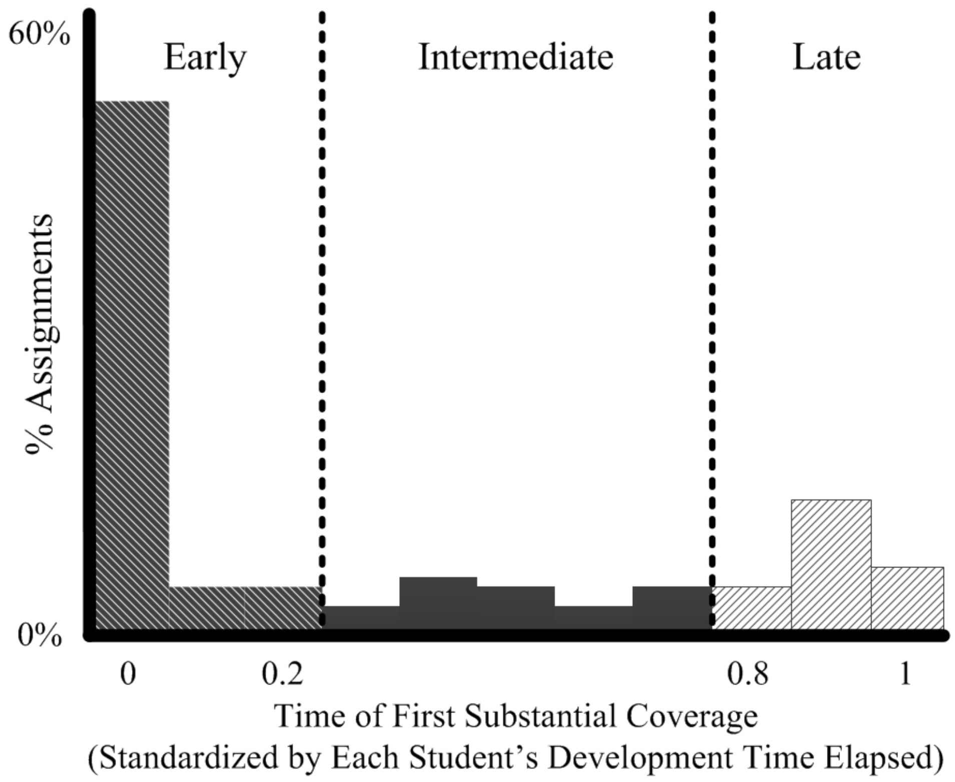


Figure 5.2. Distribution and Grouping of First Moment Reaching 85% Coverage by Standardized Time Elapsed within Assignment Submissions

5.1.3 Results

The Friedman's test for investigating within-subjects differences of the varied-performance group compared timing of development milestones (first submission, testing threshold, maximum scoring, and final submission), quality of code (correctness and coverage), as well as quantity of test code (relative NCLOC). Likewise, we compared the same metrics between exclusively A/B and C/D/F groups using the Wilcoxon signed-rank test. The following subsections describe the findings for each area of analysis, before detailing the results of a post-hoc analysis of testing strategy.

Time of Development Milestones

A within-subjects comparison reveals that the first submission time for A/B assignments (M=62.26, sd=7.29) is earlier (F(1,1)=36.96, p<0.0001) before the deadline than for the same students' C/D/F assignments (M=32.11, sd=58.52) from the varied-performance group. Likewise, the first submissions of consistent A/B students (M=68.54, sd=75.08) takes place earlier (p<0.001) than that of consistent C/D/F students (M=28.66, sd=51.11) according to the Wilcoxon test.

We compared the time remaining for the final submissions on well- and poor-performing assignments. In the within-subjects comparison, well-performing assignments were finished (M=28.02 sd=63.73) significantly earlier (F(1,1)=32.56, p<0.0001) than poor-performing assignments (M=3.88, sd=52.68). Consistent A/B students submitted their final submissions (M=30.72, sd=64.15) with significantly more time remaining (p<0.0001) than C/D/F students (M=6.74, sd=46.49).

To compare the timing of the testing threshold and maximum scoring milestones, we examined their relative worktime, or the percentage of the time elapsed between first and last submissions. Varied-performing students reached the testing threshold earlier in development (F(1,1)=1.26, p<0.01) on poor-performing assignments (M=0.55, sd=0.45) than on well-performing assignments (M=0.65, sd=0.43). The C/D/F students who attained the testing threshold (M=0.40, sd=0.39) did so earlier in their development process (p<0.0001) than A/B students (M=0.53, sd=0.37). However, interpretations of testing threshold relative worktime comparisons should use particular caution because these tests exclude all assignments that failed to achieve 85% coverage.

Varied-performing students reached their maximum scoring milestone earlier (F(1,1)=55.78, p<0.0001) within their development on poor-performing assignments (M=0.55, sd= 0.45) than on well-performing assignments (M=0.65, sd=0.43). Correspondingly, A/B students reached their maximum scoring milestone (M=0.70, sd=0.40) later (p<0.0001) than C/D/F students (M=0.41, sd=0.45).

Both of these last findings may contradict expectations: poorer assignments reached the testing threshold milestone earlier in development than better assignments, and poorer assignments reached the maximum scoring milestone earlier in development as well. Since testing early is supposed to benefit students, one would hypothesize that better scoring assignments would be associated with demonstrating substantial correctness and test coverage earlier during development compared to weaker assignments. Consequently, we investigated this issue further in ***Testing Strategies***, later in this section.

Lastly, with concern to time spent in development, we examined the overall time elapsed from first to last submission for each assignment. The within-subjects comparison of varied-performing students shows that the amount of time a student spent (in hours) on his or her well-performing assignments (M=27.31, sd=43.22) was slightly less than (F(1,1)=7.51, p<0.01) the time spent by the same individual on poor-performing assignments (M=30.66, sd=41.61). However, students who consistently achieved A/B scores (M=37.82, sd=48.71) spent nearly twice as much time (p<0.001) as students who consistently achieved C/D/F scores (M=21.92, sd=34.94).

Code Quality and Test Quality

We measured quality of solution code by tracking correctness during development. Meanwhile, we evaluated quality of testing with testing coverage. Final correctness is used to identify assignment groups in the within-subject Friedman’s test and to identify student groups for the Wilcoxon between-groups comparison. Therefore, for the final submission, we only compared test quality within-subjects and between groups.

The within-subject comparison of the varied-performance group, demonstrated significantly higher coverage (F(1,1)=684.44, p<0.0001) for assignments with high correctness (M=0.94, sd=0.07) than those with low correctness (M=0.75, sd=0.28). In other words, when students in the varied group achieved higher test coverage scores, they also produced solutions with fewer bugs (Spearman's ρ=0.67, p<0.0001). As expected, the A/B students' final submissions also had significantly (p<0.0001) higher coverage (M=0.94, sd=0.06) than those of C/D/F students (M=0.64, sd=0.31). Over the entire dataset, there was a strong positive correlation between higher test coverage and higher correctness scores (Spearman's ρ=0.71, p<0.0001).

For the first submission, varied-performance students had significantly higher correctness (F(1,1)=46.45, p<0.0001) and coverage (F(1,1)=93.88, p<0.0001) on their well-performing assignments (M=0.45, sd=0.42; M=0.71, sd=0.33) than on poor-performing assignments (M=0.14, sd=0.24; M=0.46, sd=0.24, respectively). The between group analysis also found that the first submissions for A/B students had higher (p<0.0001) correctness (M=0.40, sd=0.34) than on C/D/F student assignments (M=0.08, sd=0.18). Furthermore, the A/B's also had higher (p<0.0001) initial coverage (M=0.68, sd=0.35) than the C/D/F group (M=0.49, sd=0.34).

We also investigated the quality of code at the point of reaching the testing threshold milestone. Satisfying our definition for the testing threshold milestone already requires that coverage is at least 85%. However, not all assignments achieve this milestone. Varied-performance students reach the testing threshold on 63% of their well-performing assignments but on only 43% of their poor-performing assignments.

When considering only those assignments that reach the milestone, a within-subject comparison reveals that the solution correctness at the time of the testing threshold milestone is significantly higher (F(1,1)=36.51, p<0.0001) on well-performing assignments (M=0.74, sd=0.32) than on poor-performing assignments (M=0.42, sd=0.26). Similarly, assignments from A/B students achieve the testing threshold 65% of the time, while C/D/F students only achieve it 32% of the time. Moreover, A/B assignments (M=0.76, sd=0.39) also have higher (p<0.0001) correctness at the testing threshold milestone than do C/D/F assignments (M=0.39, sd=0.24).

Finally, we compared code quality at the time of achieving the highest correctness within each assignment. Since we already know by definition that well-performing assignments have higher correctness than poor-performing assignments, we only compare test coverage at the maximum score milestone. At the time of the maximum score milestone, varied-performance students have higher (F(1,1)=542.15, p<0.0001) coverage on well-performing assignments (M=0.94, sd=0.07) than on poor-performing assignments (M=0.71, sd=0.31). Accordingly, assignments from consistent A/B students (M=0.95, sd=0.06) have higher (p<0.0001) coverage than those from consistent C/D/F students (M=0.61, sd=0.34). It should not be surprising that throughout development, consistent A/B students regularly demonstrate better solutions and testing than consistent C/D/F students do. However, it is more telling that varied-performance students also demonstrate a positive relationship between maintaining good coverage throughout development and producing higher-quality results. This within-subject comparison rejects the hypothesis that the relationship between high-quality testing and high-quality solutions only reflects inherent study/work habits of "good" and "bad" students.

Quantity of Code

In addition to code quality, we are concerned with how much test code students write at different stages of development. By comparing the amount of test NCLOC to solution NCLOC, we can observe—at any given moment in development—the amount of work on testing relative to that of the solution. To inspect this amount of testing throughout submissions to Web-CAT, we compared relative NCLOC at the times of: first submission, testing threshold milestone (when available), maximum scoring milestone, and final submission.

In within-subject comparisons, the fit for each threshold moment was calculated independently since not all assignments reach the testing threshold milestone. On the first submission, varied-performance students had significantly higher (F(1,1)=35.25, p<0.0001) relative NCLOC (more test code per line of solution code) on well-performing assignments (M=0.61, sd=0.41) than on poor-performing assignments (M=0.47, sd=0.38).

Well-performing assignments (M=0.81, sd=0.39) also had significantly higher relative NCLOC (F(1,1)=35.81, p<0.0001) at the testing threshold milestone than poor-performing assignments (M=0.76, sd=0.32). Likewise, well-performing assignments (M=0.80, sd=0.38) also had significantly higher relative NCLOC (F(1,1)=109.73, p<0.0001) at the maximum scoring milestone than poor-performing assignments (M=0.64, sd=0.42).

However, there is no significant difference (F(1,1)=1.37, p=0.24) in relative NCLOC between consistently well-performing assignments (M=0.81, sd=0.38) and consistently poor-performing assignments (M=0.76, sd=0.33) at the time of testing threshold milestone. Since the milestone requires substantial testing, poor assignments with less test code (and that never reach the milestone at all) are excluded, consequently artificially inflating the mean. To further support this explanation, within varied-performing students, fewer poor-performing assignments (35%) achieved testing threshold than well-performing assignments (57%).

Figure 5.3 illustrates the change in relative NCLOC during development for well-performing and poor-performing assignments of varied-performing students. Both groups show a relative increase in testing between first submission and the submission where the assignment first achieves 85% coverage. However, the well-performing assignments tend to maintain (or slightly increase) amount of test code through the rest of development while poor-performing assignments do not.

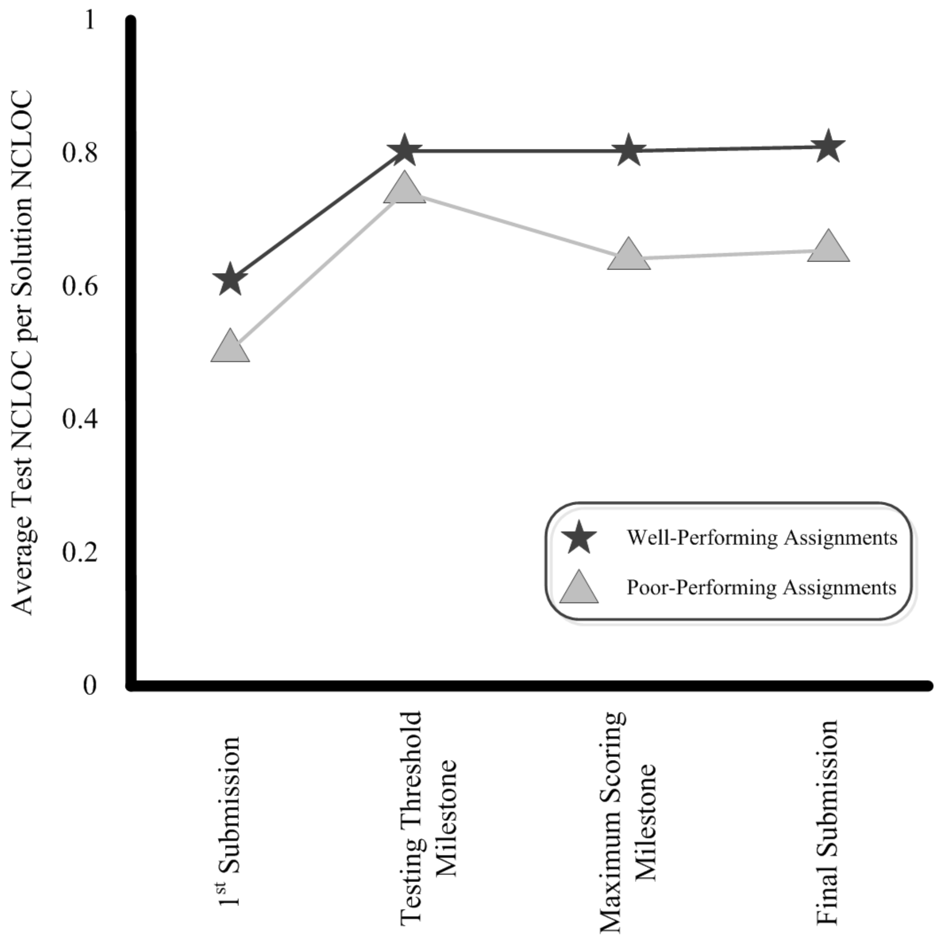


Figure 5.3. Relative NCLOC Through Development, Grouped by Final Correctness

Testing Strategies

While the majority of analyses in this study support the argument that early, incremental testing scaffolds higher-quality tests and solutions, aberrations emerge when inspecting relative worktime (between first and last submissions) of the testing threshold milestone. These peculiarities are highlighted by the results of a post-hoc Wilcoxon each pairs test comparing quantity and quality of code between early, intermediate, late, and neglectful testing groups (see: Figure 5.2). To account for chances of type I error with multiple comparisons, we used the Bonferroni correction to set a stricter ($\alpha=0.008$) confidence interval for post-hoc tests.

When comparing the final correctness between each group, early testing (M=0.80, sd=0.34) and intermediate testing groups (M=0.87, sd=0.20) performed comparably well (p=0.55), but both worse ($p<0.001$ and $p<0.003$) than the late testing group (M=0.88, sd=0.21). All three of these groups performed better (each $p<0.0001$) than the neglectful group (M=0.65, sd=0.39). For final coverage, the late (M=0.95, sd=0.06) and intermediate (M=0.95, sd=0.05) groups scored comparably (p=0.11). While the late group's mean coverage is greater than the early group's (M=0.88, sd=0.24), this difference is not significant (p=0.018). Likewise, there is no significant difference (p=0.44) between the intermediate and early groups. All three groups performed better (p<0.0001) than the neglectful group (M=0.78, sd=0.25) in terms of final test coverage.

Similarly, late (M=0.88, sd=0.40) and intermediate (M=0.87, sd=0.42) groups had higher (but not significant) final relative NCLOC (p=0.02 and p=0.21) than the early testing group (M=0.83, sd=0.44). There was no significant difference between the intermediate and late groups (p=0.23). All three groups had higher final relative NCLOC (p<0.0001) than the neglectful group (M=0.70, sd=0.42). From these comparisons alone, one might conclude that either early testing really is no better than later testing strategies, or that there is a previously undetected phenomenon affecting the early testing group scores.

To investigate the relationship between testing threshold relative worktime—which the testing groups are based on—with other timing factors, we performed a Spearman's correlation with: time remaining at the first and last submissions, final time elapsed, and maximum scoring relative worktime. **Table 5.2** summarizes the correlations with the testing threshold relative worktime. Given the moderately positive correlation between the relative worktime of the test threshold and maximum score threshold, it is not surprising to find that the maximum score milestone also has a bimodal distribution, as demonstrated in **Figure 5.4** (on the following page).

Table 5.2. Spearman Rho Correlation with Testing Threshold Relative Worktime

			Testing Threshold Relative Worktime (M=0.53, sd=0.39)	
	M	sd	Spearman Correlation (ρ)	p
Time Remaining (1st Submission)	52.48	70.01	-0.01	0.52
Relative Worktime (Maximum Scoring Milestone)	*0.62*	*0.44*	*0.34*	*<0.0001*
Time Remaining (Last Submission)	*21.51*	*61.07*	-0.04	0.09
Time Elapsed (Last Submission)	30.97	44.33	*0.07*	*<0.01*

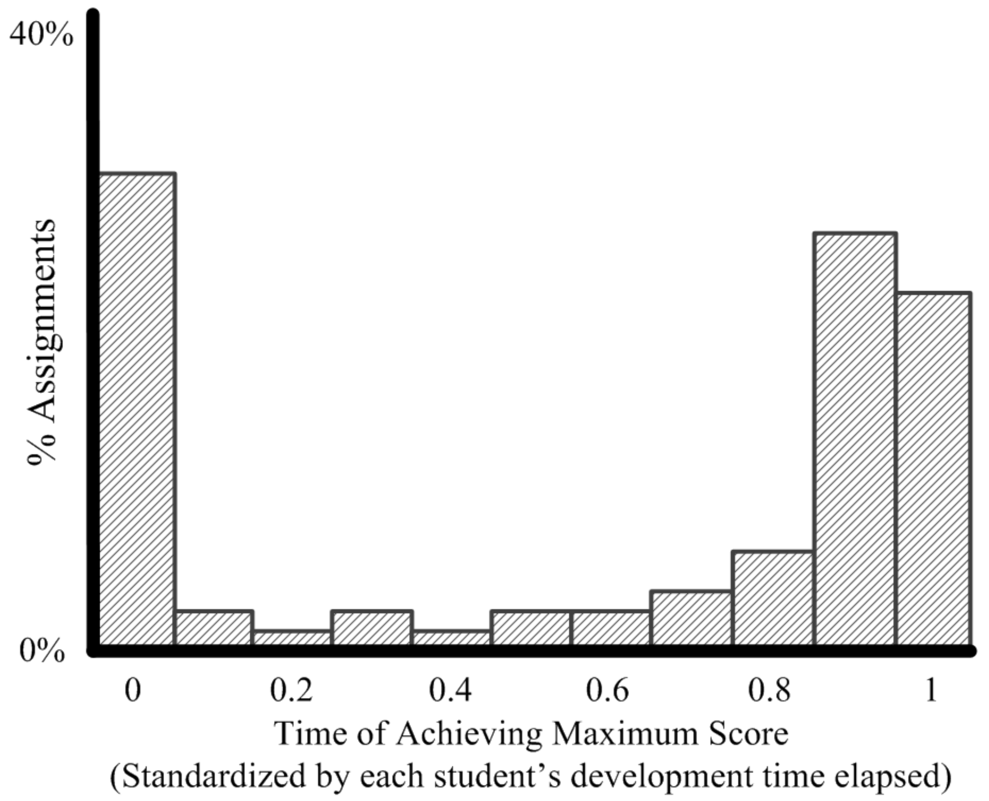


Figure 5.4. Distribution of Time of Maximum Scoring Milestone, by Relative Worktime

A quarter of relative worktimes for maximum score threshold are 0.0, or the very first submissions to Web-CAT. These assignments reflect instances where students never improve upon their correctness after their first submission. We will refer to these assignments as the complacent group. As one might expect from assignments without correctness improvement, these assignments averaged very few submissions (M=4.69, sd=6.31), and little time elapsed between first and last submissions (M=11.85, sd=30.83). The correctness (M=0.52, sd=0.47) and coverage (M=0.69, sd=0.37) on average are also quite poor. However, the wide standard deviations reflect divergent trends within this segment.

Half of the complacent assignments stagnate with either 100% or 0% correctness. The 25% who began with 0% correctness and never improved reflect students who either neglected the effort necessary for the assignment or seem to have given up on it. Meanwhile, the 25% of the complacent group who achieved 100% correctness on their first submission clearly demonstrated substantial work before submitting to Web-CAT and were nearly—if not totally—finished. Unfortunately, the data reflecting these assignments do not expose much of their development process.

Accordingly, with poor granularity of the complacent group's development process, it would not be appropriate to categorize their testing strategies with those assignments with finer granularity. For instance, if a student only submits once but achieved at least 85%

coverage, does that indicate early testing? Likewise, imagine a student facing an impending deadline and submitting four times over the span of an hour. Is there a discernible difference in testing strategies between taking 10 and 60 minutes to achieve 85% coverage? Even in non-complacent assignments with few submissions or short duration, we can compare development patterns through differences in changes to their work. However, complacent assignments suffer from a lack of both granularity and change.

These complacent assignments appear to reflect at least two distinct habits. In some situations, students complete the vast majority of work on well-performing assignments before submitting. In other situations, students seem to give up shortly after their first submission. It is unlikely that testing strategies influenced the outcomes of those complacent with poor correctness. Instead, their outcomes likely reflect personal characteristics or study habits such as lack of motivation, poor time management, or inadequate preparation. Meanwhile, it would be intriguing to investigate the development habits of those in the complacent group who started with perfect correctness. However, since our data are limited to their submissions to Web-CAT that began at the tail end of the development process, it would not be fair to categorize their testing habits as either early or late with so little information.

If we prune complacent assignments from the dataset and repeat the Wilcoxon test, we observe a notable change in results. Now, when comparing the final correctness, the early group (M=0.90, sd=0.18) performs comparably (p=0.48) to the late (M=0.89, sd=0.19) group, better (p<0.001) than the intermediate group (M=0.87, sd=0.19), and still considerably better (p<0.0001) than the neglectful group (M=0.79, sd=0.26). However, testing quality effects are even more salient.

When comparing the final coverage of the pruned dataset, the early group (M=0.96, sd=0.08) performs significantly better than the intermediate (M=0.95, sd=0.05, p<0.0001) and late (M=0.95, sd=0.06, p<0.001) groups. There is no significant difference (p=0.43) between coverage of intermediate and late groups. All three groups produced significantly higher coverage (p<0.0001) than the neglectful group (M=0.84, sd=0.16).

The final relative NCLOC for early (M=0.92, sd=0.42) is insignificantly higher than intermediate (M=0.86, sd=0.42, p=0.02), and late (M=0.87, sd=0.42, p=0.17) groups. All three groups produce higher (p<0.0001) relative NCLOC than the neglectful group (M=0.75, sd=0.42).

With these improved groupings, **Figure 5.5** illustrates the timing and coverage at each milestone for each group (excluding pruned submissions). The milestones follow, in order: first submission, testing threshold milestone, maximum scoring milestone, and final submission. Note that the neglectful group, by definition, does not have a second (testing threshold) milestone.

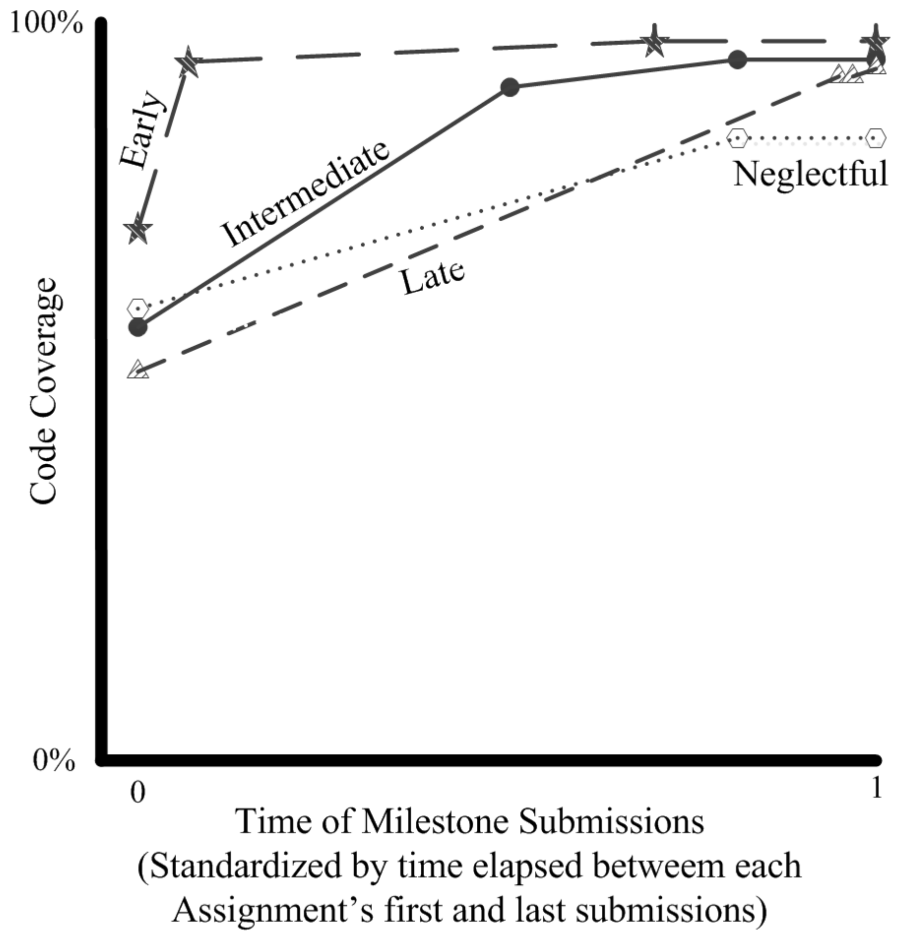


Figure 5.5 Relative Worktime of Milestone Submissions, Grouped by Test Behavior

Finally, to investigate the relationship between testing strategy and outcome, independent of time management behaviors, we performed post-hoc analysis excluding submissions with poor habits. Specifically, we included only assignments with the following characteristics: one (or more) submissions preceding the deadline by at least 48 hours; multiple submissions spanning development over 24 hours; and final submission before the deadline. This pruned dataset included 2,727 assignments from 790 unique students.

Using the Wilcoxon test on each pair of groups (early, intermediate, late, neglectful), we found early-tested assignments (M=0.92, sd=0.17) had higher correctness ($p<0.01$) than intermediate-tested assignments (M=0.90, sd=0.15) approaching significance ($\alpha=0.008$). The intermediate group also approached higher ($p=0.09$) correctness than the late group (M=0.89, sd=0.15). Accordingly, the aforementioned groups also had higher correctness ($p<0.0001$) than the neglectful group (M=0.76, sd=0.32).

The final test coverage comparison between groups revealed similar results. The early group (M=0.96, sd=0.08) had significantly higher coverage ($p<0.0001$) than both the intermediate (M=0.94, sd=0.05) and late (M=0.94, sd=0.05) groups. The intermediate and late groups had no significant difference in coverage (p=0.64). Finally, the intermediate and late groups also had significantly higher coverage (p<0.0001) than the neglectful group (M=0.84, sd=0.15).

Finally, we reviewed students' testing behaviors across assignments. Approximately 18% of students demonstrated neglectful testing practices on all of their assignments. Meanwhile, 2% consistently tested late, 3% consistently tested early, and 1% consistently tested in-between ("intermediate"). The remaining 76% of students followed different testing behaviors on different projects.

5.1.4 Discussion

Results showing strong positive correlations between early test quality and quantity and the consequential assignment outcomes corroborate with previous studies suggesting benefits of test-first and test-early strategies [BUM+2002][BE2012i][DJS2008]. However, we focused on within-subject comparisons of students who performed well on some assignments and poorly on others. By contrasting their testing behaviors on well-performing assignments with those on poorer-performances, we control for students' personal traits and isolate the impact of differences of behaviors.

We characterized divergent testing strategies based on analysis of how early within an assignment's development a student first achieves substantial (>85%) test coverage of their code. By considering timing relative to individuals' development time—opposed to in absolute time before the deadline—we were able to compare testing strategies independently of general time management skills. Furthermore, to investigate the effects of testing independently of time management behaviors, we compared a subset of only assignments that started early, continued to develop for substantial time, and submitted their final work on time.

We found that on assignments where students reached this testing threshold milestone early in their submissions (<20% within relative worktime), they usually produced higher quality code and tests than those who demonstrated later testing strategies. Surprisingly, we also found that assignments achieving the testing threshold late (>80% within relative worktime) produced higher quality code and tests than those who did so in the middle of their development.

This study benefits from the advantage of observing students' testing behaviors with finer granularity than reviewing a solitary assignment deliverable. However, we also identify small segments of submission behaviors with inadequate detail to infer testing strategies. Approximately one-in-every-eight varied-performance assignments had very few submissions with no measurable progress from their first submission. For the purpose of this study, these complacent assignments lacked enough detail to categorize their testing strategies confidently. Consequently, we excluded these outliers as we found that they were unduly influencing our statistical results.

However, there is potential value in learning more about these assignments with unknown testing and development strategies. Extending the snapshots of students' work into earlier stages of development (i.e. before first submissions) will promote more detail in models of testing behavior. Therefore, we hope this study represents a step in the progression of techniques and tools that enable increasing granularity in observing software development behaviors.

5.2 Scaffolding Software Testing

Computer science curricula have begun to add software testing requirements to programming assignments and automated grading systems such as Web-CAT [Edwa2013] and Marmoset [Spac2013] have incorporated features to evaluate the quality of students' tests. In addition, these grading systems provide feedback to students to identify shortcomings in both solution correctness and thoroughness of testing. However, concerns arise about whether automated feedback systems empower students with effective software testing practices. In particular, multiple instructors have independently expressed worries—based on their anecdotal observations—that their students rely upon Web-CAT's automated testing feedback to supplant the need for thorough testing.

Since the point of testing is to verify that software behaves as expected, the practice of writing tests should involve reflecting over a variety of different test cases and confirming that each produces the expected outcome. In performing due diligence in testing, a student should consider not only the mainstream cases, but also which unusual cases may cause bugs, or unexpected outcomes. To the contrary, we have found that students write "happy path" tests—only those mainstream cases unlikely to cause unexpected bugs [ESB2014].

After students submit their work to Web-CAT, they receive feedback on the correctness of their solutions—as determined by a set of obscured reference tests provided by the instructor—as well as measurements of how much of their own code they have exercised with their own tests. In addition, failed reference test cases generate hints that instructors use to give students general direction in identifying what is malfunctioning without revealing the specific details of the reference test. Web-CAT's correctness score provides students with confirmation of how well their solution performs according to the project specifications. While the hints do not necessarily indicate exactly which test cases are not working, they also provide students with insight into which features they need to fix in their solution code. Both types of feedback may be well intended to help students make progress on their programming assignments; however, they also relieve at least some of the burden of software testing.

Ideally, students would test their own code thoroughly enough that without *any* feedback from the instructor, they would have strong confidence that their code behaves correctly by its performance against their tests alone. In non-academic settings, software developers do not usually have the benefit of an instructor's thorough checks to make sure their software is sound and robust. Instead, they would have to rely on their own estimates of the code quality to determine if it is ready to deploy. Nevertheless, it may be unrealistic to expect students to know how to test effectively without assistance.

Consequently, programming courses should find a compromise between alleviating the burden of quality assurance from the students and demanding professional-level testing proficiency immediately. In this section, we reconsider students' interaction with automated grading systems. We describe a model for scaffolding feedback in Web-CAT to encouraging students to reflect over their code and concentrate on writing effective software tests to reveal bugs instead of relying on external sources.

5.2.1 Background

When students submit programming assignments to Web-CAT, test coverage estimates the quality of their tests. Traditionally, Web-CAT evaluates the percentage of methods, lines of code, and conditions within a student's solution code are executed by her unit tests. This combination of method, line, and condition coverage provides a general perspective of the breadth of code exercised by the student's tests. However, even 100% code coverage does not necessarily indicate that tests have thoroughly checked the program's behavior. For example, consider the following code segment:

```
if ( a && (b || c) )
{
    return true;
}
else
{
    return false;
}
```

Figure 5.6. Example of IF-ELSE Control with a Compound Condition

The above code can achieve full coverage in two test cases: one where the variables `a` `b` and `c` are true (which returns true) and one where they are all false (which returns false). With these two test cases, we have executed each line in the code and have executed both outcomes of the `if` condition (`true` and `false`). However, with these tests, we have not considered other cases with different combinations of values for the variables. While the compound condition has been both true and false, variations of the atomic decisions are not thoroughly tested. For example, if **`a`** and **`b`** are true but **`c`** is false, does the code's outcome (true) match the expected behavior of the program?

There are alternate approaches for estimating test quality beyond traditional coverage measurements. Another study compared coverage to two other techniques--mutation testing and all-pairs testing—to determine how well they predict bugs in students' actual code [ES2014]. Mutation testing involves making multiple versions of a student's source code with minor modifications—each called a "mutant"—and identifying whether the student's tests "kill" mutants by producing different outcomes from the test with its original code. Meanwhile, all-pairs testing involves running a student's tests against all other students' implementations of the same assignment and measures how many bugs it identifies. The study found that all-pairs testing was most effective at determining a test suite's ability to identify bugs while mutation testing was no better than coverage at doing so.

However, all-pairs testing has caveats that limit the practicality of using it in automated grading systems like Web-CAT. First, as the number of students in the class (and the number of tests they write) grows the computational expense of running tests against all other students' code increases exponentially. Perhaps more poignantly, Web-CAT allows students to submit their assignment multiple times throughout development so each student's code and all-pairs score will change over time (until submissions are no longer accepted). Consequently, the first students to submit their code would have less comprehensive all-pairs analysis and the results may change as others submit their work.

Modified Condition / Decision Coverage (MC/DC) is another approach for measuring comprehensiveness of software tests [CM1994]. Unlike conventional coverage where a decision only requires an overall true and false evaluation, MC/DC requires showing that each *atomic* condition affects a "decision's outcome by varying just that condition while holding fixed all other possible conditions" [Chile2001]. As one form of MC/DC, *masking MC/DC* takes advantage of setting one operand of an operator so that the other operand cannot affect the value of the operator. Reconsidering the example in **Figure 5.6**, the operand **`b`** can be masked by holding **`c`** true so that it will evaluate as true regardless of **`b`**'s value; likewise, **`a`** can be masked by holding **`(b || c)`** false so that the expression will evaluate to false regardless of **`a`**'s value. Because masking MC/DC examines conditions with more detail, it subsumes Web-CAT's approach of measuring coverage for each method, line, and control decision.

In addition to challenges of measuring the thoroughness of students' tests, automated grading systems need to encourage students to test their code more thoroughly rather than depend on automated feedback to tell them whether their code works. Instead, it should provoke students to reflect, "What other test cases might possibly break my code?" However, as alluded to in our previous study, *4.2 A Formative Study of Influences on Testing Behaviors*, students' responses to Web-CAT's feedback concentrate predominantly on fixing errors in their code to improve their correctness score. While students also aim to earn full credit for testing by obtaining 100% coverage, their approach does not reflect the mentality of trying to expose weaknesses in their code. Rather, their testing behaviors more closely demonstrate a desire to extend the least amount of work to obtain 100% coverage, a relatively superficial objective for testing.

Nevertheless, students' responses to Web-CAT's feedback is understandable since they are driven to achieve good grades (which 100% coverage allows), and the feedback already provides some insight into how acceptable their code is without having to test it sufficiently themselves. Our previous attempts with adaptive feedback with overt encouragement to test (see Chapter 4, *Interventions for Reinforcing Methods*) had no long-term impacts on their testing behaviors. However, the adaptive feedback interventions supplemented other feedback that still gave students insight into the quality of their code, perhaps undermining the efforts to encourage testing. Consequently, we designed a model for scaffolding feedback that still provides students with help on improving their code, but with a strategic approach emphasizing testing.

5.2.2 Method

A Framework for Scaffolding Testing

Previous approaches considered overall coverage of the student's entire programming project. Of course, students' projects usually include many methods and often even multiple classes. Consider a situation where a student submits a project that is not behaving perfectly (according to the instructor's specifications) and has some (but not exhaustive) testing. Web-CAT detects some bugs in the student's code from some of the instructor's tests failing and gathers the hints associated with each failed test. For one feature of the project, the student thoroughly tested her code but the feature was not implemented precisely to the instructor's specifications: the text output in her implementation uses commas to separate values but the instructor expected line breaks instead. The problem with this feature is not one caused by a lack of testing, but rather by a misunderstanding of the requirements. It is unlikely that more testing will expose the problem and without some insight from the hints, it may be frustrating to overcome this relatively superficial bug.

However, in the same submission to Web-CAT, the student may have another feature that is incorrectly implemented but has not discovered the problem because she has not tested that feature as thoroughly. Unlike the previous example, revealing a hint to correct this bug would be inappropriate because the student should first try to reflect over what test cases she neglected in her test suite. Nevertheless, by only considering the coverage of the entire project, the automated grader cannot determine which feature has been thoroughly tested or which hints are appropriate to show.

Consequently, in order to choose appropriate guidance for students, automated feedback must first identify testing quality feature-by-feature. To do so, we developed a custom runner for JaCoCo [Jaco2014]—an open source coverage tool for Java—that produced a coverage report for each instructor reference test independently. After identifying which reference tests fail against a student's submission, the system then reviews the report for each test and identifies which method(s) within which class(es) of the student's code the test executes. As a result, the system can associate failed tests with particular features within the student's submission. While these features cannot always be localized to a single method, they narrow the scope considerably because unit tests by definition concentrate on small portions of code.

A reference test suite may include many test cases for a single feature and, as a result, create duplicate hints and execute the same methods in the student's code. As Web-CAT traditionally eliminates redundant hints, our system records the number of times a hint occurs for a single submission and considers it one unique hint. After recording the test results, for each unique hint, it determines whether the student has tested the feature associated with that hint enough to warrant revealing the hint. To do so, the system calculates the masking MC/DC of the student's own tests against the method(s) associated with that hint.

If the student has not sufficiently tested the atomic decisions within the methods' control structures, the system will withhold the hint and instead direct the student to the method(s) that need more testing. However, if masking MC/DC is fully satisfied on each of the methods associated with the reference test failure, the system reveals the hint. Accordingly, this framework requires students to reflect over each feature of their work and test it thoroughly before receiving hints. In that manner, students cannot rely upon hints to supplant the burden of testing.

Evaluation of Existing Technology

The aforementioned framework emphasizes offering hints only on features of the student's code that have been adequately tested. The precision of this feature-by-feature feedback would be demonstrated by a lack of both false positives and false negatives in providing hints. A false positive occurs when a hint is provided to a student when the feature it tests has not been tested well by the student. Meanwhile, a false negative occurs when a student has sufficiently tested a feature but does not receive a hint generated by a reference test that failed the same feature. To compare the framework's approach to hint precision to existing approaches in automated testing, we empirically tested the occurrences of false positives and false negatives using Web-CAT's traditional hint mechanism.

Web-CAT allows instructors to customize how many hints students receive for any given assignment submission. However, its default setting is to show a maximum of three hints, where hints with multiple occurrences (multiple failed reference tests generating the same hint) are given precedence. If additional hints are generated by failed tests, they are obscured until later submissions resolve the bugs responsible for the revealed hints. Consequently, hints shown may include those generated by reference tests that fail on features that the student has not tested (a false positive). Likewise, if a student has tested a feature well but a reference test's hint is not within the top three hints, this represents a false negative. Since traditional automated graders do not follow our feature-by-feature scaffolding framework, we hypothesized that Web-CAT does not discriminate between earned (true positives) and unearned hints (false positives) any better than it discriminates obscuring unearned hints (true negatives) from earned hints (false negatives).

To test our hypothesis, we analyzed existing student submissions to Web-CAT's traditional automated feedback system. Using the procedures described in the test scaffolding framework in this section, we performed post-hoc analysis of which features each reference test exercised, and whether the student's earned each hint (both revealed and withheld) according to masking MC/DC. The student submissions were from a programming assignment from the Fall 2011 term of a CS2 course that involved students implementing (and testing) two queue data structures: one as a linked structure and the other delegating an array. We previously collected consent from students to analyze the results of this assignment. After excluding submissions from students who chose not to consent and removing errant submissions (such as non-compiling code), we analyzed 2,083 submissions from 72 individual students.

Students averaged submitting to Web-CAT approximately 29 times (M=28.93, sd=24.32) each. For each failed test, we evaluated the masking MC/DC score for the methods it exercised and recorded whether it earned the hint with complete masking MC/DC on each method. After analyzing each unique hint generated for a submission, we also recorded how many unique methods in the student's code needed improved MC/DC to earn all hints. On average, each submission required additional testing on 3.09 (sd=1.74) unique methods, or 8.68% (sd=5.16%) of its solution methods; **Figure 5.7** shows the distribution across all submissions.

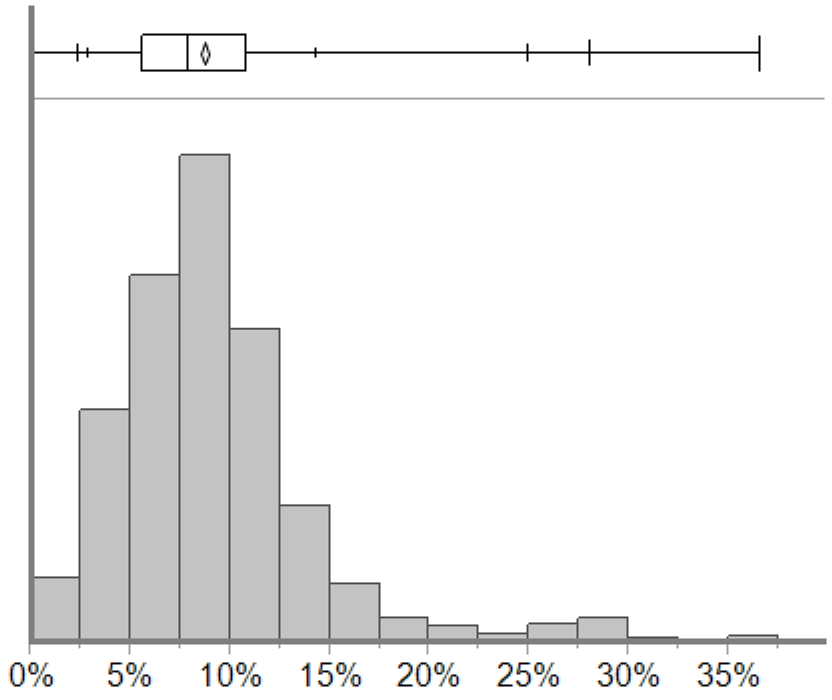


Figure 5.7. Distribution and Box-plot of Methods Requiring More Testing

Meanwhile, the failed reference tests produced an average of 22.24 (sd=15.99) unique hints per submission. **Figures 5.8** and **5.9** shows the distribution of earned (full MC/DC for applicable methods) and unearned hints (incomplete MC/DC for applicable methods) that were revealed and obscured, respectively.

We used the Shapiro-Wilk test to check for normality of these distributions. The test rejected the null hypothesis (that the data are normally distributed) for both the unique methods (p<.01) and unique hints (p<.01). Consequently, to test our hypotheses by comparing the incidents of false positives and true positives for revealed hints to false negatives to true negatives of withheld hints, we used the Friedman's test for within-subject comparisons of non-parametric distributions. Furthermore, we performed Wilcoxon signed-rank test to compare the overall percentage of hints earned of those revealed to that of those withheld.

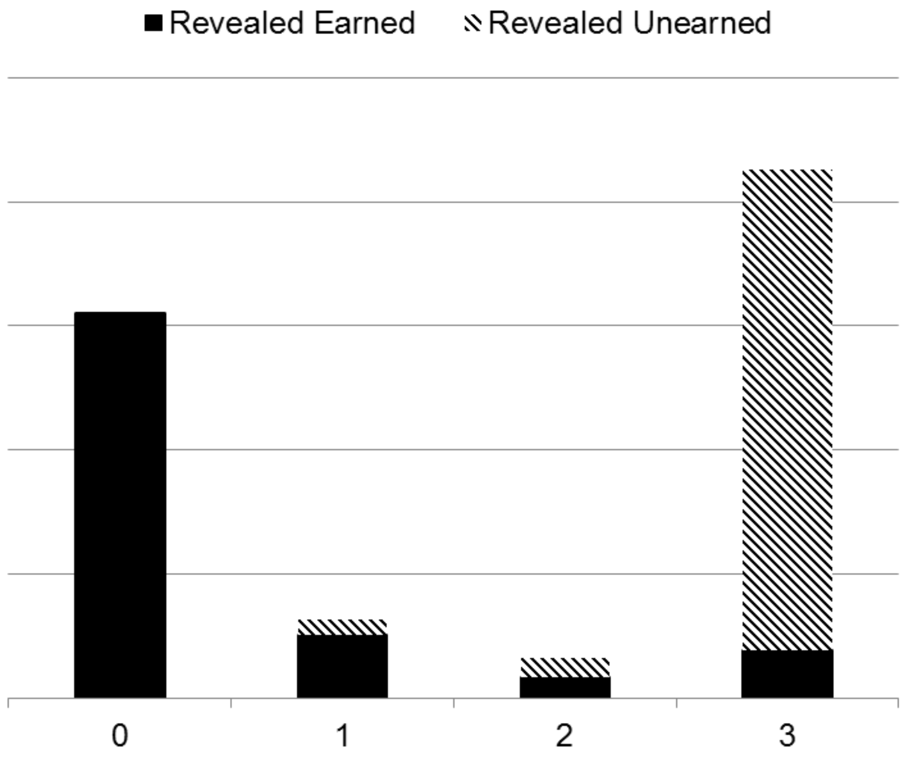


Figure 5.8. Distribution of Revealed Hints, Earned Versus Unearned

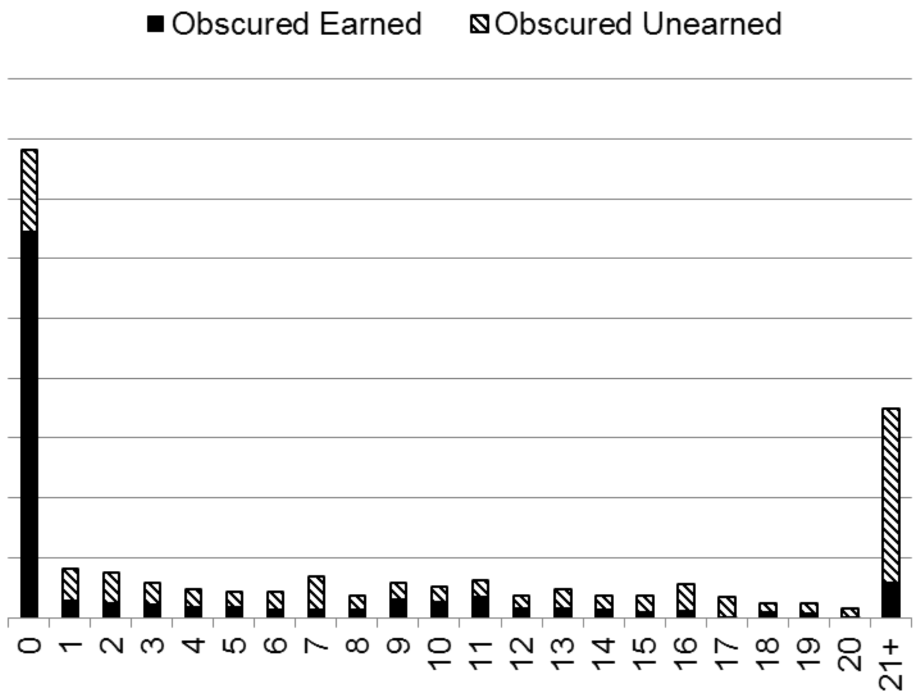


Figure 5.9. Distribution of Obscured Hints, Earned Versus Unearned

5.2.3 Results

To test our hypothesis, we performed a Friedman test comparing the rate of earned-to-unearned hints between those revealed and withheld, when averaged across multiple submissions within-subject. The test found that there was no significant difference (F(2,1)=-0.67, p=0.49) between the rate at which revealed hints were earned (M=0.20, sd=0.09) and withheld hints were earned (M=0.20, sd=0.09).

We also compared the overall rates of earned hints—considering each submission independent of who submitted it—using a Wilcoxon signed-rank test and found that withheld hints had a higher rate (p<.0001) of earned hints (M=0.21, sd=0.30) than did revealed hints (M=0.16, sd=0.32). Both tests reject the null hypothesis that revealed hints have a higher rate of earned-to-unearned hints than withheld hints.

5.2.4 Discussion

The post-hoc analysis of hints that Web-CAT natively reveal and obscure supported our hypothesis that the automated feedback often results in both false positives and false negatives. False positives inadvertently relieve some of the burden of testing thoroughly by providing insight into failing tests for features that have not been thoroughly tested. In fact, false positives (revealing unearned hints) were roughly four times more common than true positives (earned hints revealed). Withheld hints had comparable rate of earn-to-unearned hints, resulting in roughly half of earned hints being withheld. With such high false positive and false negative rates, it should not be surprising that the current hint mechanism does not promote students to practice reflective testing before trial-and-error in response to failed hints.

One might consider configuring Web-CAT to remove the limitation to a maximum of three hints per submission. Doing so would eliminate all false negatives. However, the false positive rate would continue to overwhelm the number of earned hints and continue to undermine attempts to encourage students to rely on their own testing instead of using Web-CAT as a crutch. While the interventions described in Chapter 4 avoid static restrictions on how many tests Web-CAT reveals, their designs do not account for precision of hints served on a feature-by-feature basis. Consequently, there is no reason to believe that they would have any better success at preventing false positives and false negatives than the default Web-CAT settings did.

To the contrary, our proposed framework offers a unique approach to scaffolding feedback to guide students to revisit their extent of their testing without depending on Web-CAT to determine behavioral acceptability for them. For example, without any test cases, our framework could first indicate only that features in the student's assignment have not been tested—no indication of the program's correctness or results from reference tests. Then, as the student adds test cases and resubmits, the feedback scaffold would give more direction by identifying features that are not working properly and need more thorough testing. Finally, when the student has thoroughly tested a feature but has not resolved its disparity from the project specifications, the framework would provide detailed hints about why the feature fails reference tests.

This framework's approach to scaffolding feedback is modeled after Vygotsky's Zone of Proximal Development [Vygo1978]. Before a student has written tests, her immediate need in effective software development and problem solving is to reflect over her solution and consider its possible flaws. As she continues to reflect and test thoroughly, she will gradually approach a mastery of the problem where she can use hints to reconcile differences between her implementation and the expected solution.

In addition, it is worth acknowledging that there is no *de facto* standard for measuring test effectiveness. Therefore, while our use of masking MC/DC in our application of the framework should provide more accurate estimations of test thoroughness than conventional coverage, the framework is not explicitly tied to masking MC/DC. As we continue to study methods of evaluating student testing on programming assignments, other measurements could replace masking MC/DC without upsetting the framework's infrastructure. In conclusion, we hope to evaluate how interaction with our framework influences student testing behaviors while continuing to investigate options for assessing test quality.

Chapter 6

Conclusion

There is growing pressure to prepare students as proficient software testers. This dissertation concentrates on the contemporary software development process of test-driven development (TDD), which incorporates a test-first approach to incrementally producing code with confidence that adheres to the behaviors as specified by its corresponding unit tests. While TDD's popularity in industry makes it a practical method to learn in computer science curricula, some programmers hesitate to adopt the technique and avoid testing thoroughly and incrementally. Through six studies, we described comprehensive investigations into influences on novices' adoption of incremental testing as well as empirical analysis of the effectiveness of incremental testing and educational interventions to help develop good testing practices. Specifically, we addressed the following three objectives:

1. **Describe student affect (emotions and valence) and opinions with regard to their influence on adherence to test-driven development (TDD).**

Because TDD involves a stark contrast to novice programmers' "code now, test later" mentality, they often have poor opinions of test-first and incremental testing strategies. Previous research provided general opinions of TDD, but did not dissect programmers' opinions on the two main components of TDD: testing first, and testing in small increments. As we described in Chapter 3, while early testing behaviors showed modest improvements in code and testing quality, students resisted strict TDD. Our surveys revealed that while they generally accepted principles of unit testing, their prevailing opinion was that testing first did not help. Consequently, we found that students did not entirely object to TDD, but primarily to its test-first approach. The surveys also revealed that students did not typically test incrementally; this discovery exposed a need for educational intervention that motivated our second objective:

2. **Design an eLearning intervention for encouraging TDD adherence and evaluate its impact on student affect, behaviors, and outcomes.**

Detecting incremental testing requires students' software development process to be observed. However, it is impractical to expect instructors to watch all their students while they work on individual assignments. In addition, assessment for programming assignments traditionally focuses on the product of students' work, which does not consider the *process* they followed. Consequently, we leveraged Web-CAT—an automated grading system—to collect snapshots of students' coding while they worked on programming assignments and evaluate their software development processes.

Since students did not readily incorporate testing throughout their development processes, we developed an adaptive system to encourage them to begin testing earlier. The adaptive feedback systems described in Chapter 4 introduces novel approaches to identifying testing behaviors while students are working on programming assignments. Experimentation with rewards and reinforcement techniques proved unsuccessful in affecting long-term behavioral change. However, influencing changes in complex behaviors such as those involved in software development and problem solving is an ambitious aspiration.

Nevertheless, our experiments with adaptive feedback showed potential for using concrete testing goals to complement reward systems that can promote short-term behavioral change. Meanwhile, qualitative evaluation of students' programming strategies and interaction with automated grading systems helped reframe the role that eLearning systems take during programming assignments. To supplement these findings with quantitative evaluation of students' behaviors, we focused on our third objective:

3. **Characterize software testing strategies demonstrated by students and evaluate their consequential outcomes.**

Finally, Chapter 5 of the dissertation evaluated the impact of when students test on the quality of the code they produce. By analyzing a comprehensive, five-year data set of students' programming assignments, we found that even when controlling for external variables—such as procrastination and varying learner diligence—students who tested thoroughly earlier in their work produced higher-quality code and tests. Consequently, we synthesized our findings and developed a framework for scaffolding feedback to guide students through effective, incremental testing strategies. By identifying individual features within a student's solution, our framework uses higher testing standards to detect whether each feature needs more testing and guides feedback accordingly.

Future Work

Our studies used student-initiated submissions to an automated grading system to capture snapshots of their work. While this approach department from traditional *product-oriented* assessment and introduced novel *process-oriented* analysis, there were two principal limiting factors. First, the submissions could not indicate whether students followed TDD's "test-first" approach. Since coverage scores were dependent on students submitting both solution and test code together, we do not know the order in which they were developed: "test a little, code a little," or vice-versa. Consequently, our quantitative analysis concentrated on incremental testing and relied on students to report their adherence to "test-first" in surveys.

Secondly, students typically did not submit their work to the automated grader until they had written at least half of the code they would ultimately write for the assignment. As a result, snapshots of students' code lacks data from the beginning stages of their work. Since TDD advocates testing throughout development and our studies suggest testing early produces better code, it would be advantageous to provide educational interventions to promote testing as students begin their assignments. Furthermore, it may be more difficult for students to change their testing behaviors after they have completed a substantial amount of the development. Instead, early interventions may be more effective in promoting good testing habits.

As our *process-oriented* approach improved upon traditional *product-oriented* assessment, we hope to continue to increase the granularity of data representing students' development. By integrating test evaluation and feedback into students' integrated development environments (IDE), we can address both of the limiting factors of this research. Continuous test evaluation within an IDE would gather whether students' develop tests or their solution code first. In addition, integrating educational interventions within a student's IDE provides the opportunity for adaptive feedback to promote good testing practices before students adopt bad habits. Feedback inside an IDE may apply our findings, particularly by taking advantage of salient goals and self-monitoring to empower students to monitor and improve their testing behaviors.

The automated grading system used for the studies in this dissertation provided feedback only upon students' request (by submitting their work). If adaptive feedback is provided persistently within the IDE instead, other questions arise. For instance, how can feedback provide useful information and encouragement without disrupting students or overburdening their cognitive load while programming? Likewise, how do reinforcement methods such as salient goals and rewards affect student behavior when provided immediately? Finally, how does persistent, immediate feedback affect students' anxiety on programming assignments?

In addition to improving the granularity of data collection for observing software development processes, there is a need to improve techniques for evaluating software test quality. As we described in Section 5.2, Modified Condition/Decision Coverage provides more rigorous test evaluation than traditional coverage metrics. However, even high masking MC/DC scores do not necessarily indicate that tests are comprehensive and assertions are robust. While our study [ESB2014] found that mutation analysis was no more effective in identifying faults in students' code than masking MC/DC, it is worthwhile to continue studying mutation analysis because it provides better insight into test assertion quality than coverage and masking MC/DC. Consequently, a combination of both (and/or other) techniques may provide better evaluation of students' tests.

Finally, our investigation of adaptive feedback techniques may have practical application beyond incremental testing reinforcement. When teaching other software engineering practices, our tools and findings can offer guidance on reinforcing other target behaviors as well. In addition, as we continue to study students' proficiency at testing, it is becoming increasingly evident that students need more practice at writing software tests with a critical "how can I break this code?" mentality. Consequently, there is a unique opportunity to leverage what we have learned about adaptive feedback and evaluation of test quality to build an eLearning drill-and-practice system that scaffolds students' mastery of testing on small exercises.

Bibliography

[ABET2013] ABET (2013). "Criteria for Accrediting Computing Programs, 2013-2014." Retrieved November, 2013, from http://www.abet.org/DisplayTemplates/DocsHandbook.aspx?id=3148.

[ACM2013] ACM (2013). "Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science." Retrieved June, 2014, from http://www.acm.org/education/curricula-recommendations.

[AKA+2001] Anderson, L. W., Krathwohl, D.R, Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, J. & Wittrock, W.C. (2001). A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives, Addison Wesley Longman.

[Atla2013] Atlassian. "Clover: Java Code Coverage." 2013, from http://www.atlassian.com/software/clover/overview.

[BUM+2002] Barriocanal, E.G., Urbán, M.-Á.S., Cuevas, I.A., & Pérez, P.D. (2002). "An experience in integrating automated unit testing practices in an introductory programming course." SIGCSE Bull. 34(4): 125-128.

[Beck2003] Beck, K. (2003). test-driven development by Example, Addison Wesley.

[Beck1999] Beck, K. (1999). "Embracing change with extreme programming." Computer 32(10): 70-77.

[BN2006] Bhat, T. and N. Nagappan (2006). Evaluating the efficacy of test-driven development: industrial case studies. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering. Rio de Janeiro, Brazil, ACM: 356-363.

[Bloom1969] Bloom, B. S. (1969). Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook I: Cognitive Domain. United Kingdom, Longman Group.

[Brans2000] Bransford, J. D. (2000). How People Learn. Washington, D.C., National Academy Press.

[Bruck1999] Bruckman, A. (1999). Can educational be fun? Game Developer's Conference. San Jose, California.

[BCJ+2010] Buffardi, K., Churbanau, D., Jayaraman, R.K.N., Edwards, S.H. (2010). CoPractice: An Adaptive and Versatile Practice Tool. ASEE Southeast Section Conference.

[BE2014i] Buffardi, K. and S.H. Edwards (2014). "Responses to Adaptive Feedback for Software Testing." Proceedings of the 19th ACM annual conference on Innovation and technology in computer science education. Uppsala, Sweden

[BE2014ii] Buffardi, K. and S.H. Edwards (2014). "A Formative Study of Influences on Student Testing Behaviors." Proc. of Computer Science Education Symposium

[BE2013i] Buffardi, K. and S. H. Edwards (2013). Effective and ineffective software testing behaviors by novice programmers. Proceedings of the ninth annual international ACM conference on International computing education research, ACM.

[BE2013ii] Buffardi, K. and S. H. Edwards (2013). Impacts of adaptive feedback on teaching test-driven development. Proceeding of the 44th ACM technical symposium on Computer science education. Denver, Colorado, USA, ACM.

[BE2012i] Buffardi, K. and S. H. Edwards (2012). "Impacts of Teaching test-driven development to Novice Programmers." International Journal of Information and Computer Science 1(6): 9.

[BE2012ii] Buffardi, K. and S. H. Edwards (2012). Exploring influences on student adherence to test-driven development. Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. Haifa, Israel, ACM: 105-110.

[CCG+2006] Canfora, G., Cimitile, A., Garcia, F., Piattini, M., & Visaggio, C.A. (2006). Evaluating advantages of test driven development: a controlled experiment with professionals. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering. Rio de Janeiro, Brazil, ACM: 364-371.

[CM1994] Chilenski, J. J. and S. P. Miller (1994). Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 193-200.

[Chile2001] Chilenski, J. J. (2001). "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion."

[CM2003] Clark, R. C. and R. E. Mayer (2003). E-Learning and the science of instruction: proven guidelines for consumers and designers of multimedia learning. San Francisco, Pfeiffer.

[CF2009] Corder, G.W. and D.I. Foreman (2009). Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach, Wiley.

[DJC2009] Desai, C., Janzen, D.S., & Clements, J. (2009). “Implications of integrating test-driven development into CS1/CS2 curricula.” SIGCSE Bull. 41(1): 148-152.

[DJS2008] Desai, C., Janzen, D.S., & Savage, K. (2008). “A survey of evidence for test-driven development in academia.” SIGCSE Bull. 40(2): 97-101.

[DSN+2011] Deterding, S., Sicart, M., Nacke, L., O’Hara, K., & Dixon, D. (2011). Gamification. using game-design elements in non-gaming contexts. CHI ‘11 Extended Abstracts on Human Factors in Computing Systems. Vancouver, BC, Canada, ACM.

[Dris2007] Driscoll, R. (2007). “Westside Test Anxiety Scale Validation.” Education Resources Information Center: 6.

[ES2014] Edwards, S. H. and Z. Shams (2014). Comparing test quality measures for assessing student-written tests. Companion Proceedings of the 36th International Conference on Software Engineering. Hyderabad, India, ACM.

[Edwa2013] Edwards, S. H. “Web-CAT.”, from https://web-cat.cs.vt.edu.

[ESB2014] Edwards, S. H., Shams, Z. & Buffardi, K. (2014). Do Student Programmers All Tend to Write the Same Software Tests? Proceedings of the 19th ACM annual conference on Innovation and technology in computer science education. Uppsala, Sweden.

[ESC+2012] Edwards, S. H., Shams, Z., Cogswell, M., & Senkbeil, R.C. (2012). Running students’ software tests against each others’ code: new life for an old “gimmick”. Proceedings of the 43rd ACM technical symposium on Computer Science Education. Raleigh, North Carolina, USA, ACM.

[ESP+2009] Edwards, S. H., Snyder, J., Perez-Quinones, M.A., Allevato, A., Kim, D., & Tretola, B. (2009). Comparing effective and ineffective behaviors of student programmers. Proceedings of the fifth international workshop on Computing education research workshop. Berkeley, CA, USA, ACM: 3-14

[Edwa2004] Edwards, S. H. (2004). "Using software testing to move students from trial-and-error to reflection-in-action." SIGCSE Bull. 36(1): 26-30.

[Edwa2003] Edwards, S. H. (2003). "Improving student performance by evaluating how well students test their own programs." J. Educ. Resour. Comput. 3(3): 1.

[Fogg2003] Fogg, B. J. (2003). Persuasive Technology: Using Computers to Change What We Think and Do. San Francisco, Morgan Kaufmann.

[FAB+2003] Fraser, S., Astels, D., Beck, K., Boehm, B., McGregor, J., Newkirk, J., & Poole, C. (2003). Discipline and practices of TDD: (test driven development). Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Anaheim, CA, USA, ACM: 268-270

[Jaco2014] "JaCoCo Java Code Coverage Library." 2014, from http://www.eclemma.org/jacoco/

[JS2008] Janzen, D. and H. Saiedian (2008). "Does test-driven development Really Improve Software Design Quality?" IEEE Softw. 25(2): 77-84

[JS2007] Janzen, D. S. and H. Saiedian (2007). A Leveled Examination of test-driven development Acceptance. Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society: 719-722.

[JUni2013] "JUnit." 2013, from http://www.junit.org/.

[Kapt1995] Kaptelinin, V. (1995). Activity theory: implications for human-computer interaction. Context and consciousness: activity theory and human-computer interaction, Massachusetts Institute of Technology: 103-116.

[KJE2010] Kou, H., Johnson, P.M., & Erdogmus, H. (2010). "Operational definition and automated inference of test-driven development with Zorro." Automated Software Engineering 17(1): 57-85.

[LII+2010] Lappalainen, V., Itkonen, J., Isomöttönen, V., & Kollanus, S. (2010). ComTest: a tool to impart TDD and unit testing to introductory level programming. Proceedings of the fifteenth annual conference on Innovation and technology in computer science education. Bilkent, Ankara, Turkey, ACM: 63-67.

[LKL+2011] Linehan, C., B. Kirman, et al. (2011). Practical, appropriate, empirically-validated guidelines for designing educational games. Proceedings of the 2011 annual conference on Human factors in computing systems. Vancouver, BC, Canada, ACM.

[MBO+2013] Mekler, E. D., Brühlmann, F., Opwis, K., & Tuch, A.N. (2013). Disassembling gamification: the effects of points and meaning on user motivation and performance. CHI '13 Extended Abstracts on Human Factors in Computing Systems. Paris, France, ACM.

[MM2005] Melnik, G. and F. Maurer (2005). A cross-program investigation of students' perceptions of agile methods. Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA, ACM: 481-488.

[PC2004] Pierce, W.D. and C. D. Cheney (2004). Behavior Analysis and Learning. Psychology Press.

[RWT+2004] Rodebaugh, T. L., Woods, C.M., Thissen, D.M., Heimberg, R.G., Chambless, D.L., & Rapee, R.M. (2004). "More Information From Fewer Questions: The Factor Structure and Item Properties of the Original and Brief Fear of Negative Evaluation Scale." Psychological Assessment 16(2): 169-181.

[SWM2007] Sanchez, J. C., Williams, L., & Maximilien, E.M. (2007). On the Sustained Use of a test-driven development Practice at IBM. Agile Conference (AGILE), 2007.

[Schw2000] Schwarz, N. (2000). "Emotion, cognition, and decision making." Cognition & Emotion 14(4): 433-440.

[Spac2013] Spacco, J. "Marmoset." 2013, from http://marmoset.cs.umd.edu/.

[SP2006] Spacco, J. and W. Pugh (2006). Helping students appreciate test-driven development (TDD). Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. Portland, Oregon, USA, ACM: 907-913.

[Tome2005] Tomei, L. A. (2005). Taxonomy for the Technology Domain. Hershey, Idea Group Inc.

[TLD+2010] Turhan, B., Layman, L., Diep, M., Erdogmus, H., & Shull, F. (2010). How Effective Is test-driven development. Making software: what really works, and why we believe it. A. Oram and G. Wilson, O'Reilly.

[Vygo1978] Vygotsky, L. S. (1978). Mind in Society: Development of Higher Psychological Processes, Harvard University Press.

[WJT1999] Williams, V.S.L., Jones, L.V., & Tukey, J.W. (1999). "Controlling Error in Multiple Comparisons, with Examples from State-to-State Differences in Educational Achievement." Journal of Educational and Behavioral Statistics 24(1): 42-69.

[YD1908] Yerkes, R. M. and J. D. Dodson (1908). "The relation of strength of stimulus to rapidity of habit-formation." Journal of Comparative Neurology and Psychology 18(5): 459-482.

Appendix A: TDD Affect and Anxiety Survey

This voluntary survey includes questions designed to gauge your attitudes towards computers. We will use the combined information to help understand how students view computers. The results will be used for research purposes and course improvement.
Please complete all items, even if you feel that some are redundant. This may require 15-20 minutes of your time.
Usually it is best to respond with your first impression, without giving a question much thought. Your answers will remain confidential, and will not affect your grade in any way.
Your responses (without your name, of course--your identity is always strictly confidential) will be made available for use by other education researchers outside the course who are conducting studies on the learning of computer programming.
If you are a minor (under 18 years old), please skip this survey.
Are you majoring in Computer Science?
Yes No

Before enrolling in this course, had you previously used Web-CAT?
Yes No
Before enrolling in this course, had you previously written test code?
Yes No
Before enrolling in this course, had you previously followed TestDriven Development (TDD)?
Yes No
Rate each item individually on the 5-point scale from
1 (Very Unimportant) to 5 (Very Important)
on how important the skill is in Computer Science.

Time management
1 2 3 4 5

Problem solving
1 2 3 4 5

Attention to detail
1 2 3 4 5
Writing solution code
1 2 3 4 5

Writing test code
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Very Poor) to 5 (Very Good)
on how strong you are in the skill.

Time management
1 2 3 4 5

Problem solving
1 2 3 4 5

Attention to detail
1 2 3 4 5

Writing solution code
1 2 3 4 5

Writing test code
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Very Harmful) to 5 (Very Helpful)
on the impact of the following behaviors have on developing programs.

Beginning work as soon as it is assigned
1 2 3 4 5

Beginning work near its deadline
1 2 3 4 5

Developing thorough test code
1 2 3 4 5

Developing code and corresponding tests in small units at a time
1 2 3 4 5

Developing code and corresponding tests in large portions at a time
1 2 3 4 5

Developing tests before writing solution code
1 2 3 4 5

Developing tests after writing solution code
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Very Rarely) to 5 (Very Often)
on how often you practice the behavior.

Beginning work as soon as it is assigned
1 2 3 4 5

Beginning work near its deadline
1 2 3 4 5

Developing thorough test code
1 2 3 4 5

Developing code and corresponding tests in small units at a time
1 2 3 4 5

Developing code and corresponding tests in large portions at a time
1 2 3 4 5

Developing tests before writing solution code
1 2 3 4 5

Developing tests after writing solution code
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Strongly Disagree) to 5 (Strongly Agree)
based on your experience with test-driven development (TDD)
by incrementally developing tests and then solution code one unit at a time.

I consistently followed TDD in my programming projects during this course
1 2 3 4 5

TDD helped me write better test code
1 2 3 4 5

TDD helped me write better solution code
1 2 3 4 5

TDD helped me better design my programs
1 2 3 4 5

In the future, I will choose to follow TDD when developing programs outside of this course
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Strongly Disagree) to 5 (Strongly Agree)
based on your experience with Web-CAT automated results
(NOT TA/instructor feedback).

Web-CAT helped me improve writing test code
1 2 3 4 5

Web-CAT helped me improve writing solution code
1 2 3 4 5

Web-CAT helped me improve designing my programs
1 2 3 4 5

Web-CAT helped me follow test-driven development
1 2 3 4 5

Web-CAT helped improve my time management
1 2 3 4 5

Web-CAT helped improve my attention to detail
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Not at all characteristic or true of me) to
5 (Extremely characteristic or true of me)
on how well they describe you.

I worry about what people will think of me even when I know it
doesn't make any difference
1 2 3 4 5

I am frequently afraid of other people noticing my shortcomings
1 2 3 4 5

I am afraid that others will not approve of me
1 2 3 4 5

I am afraid that people will find fault with me
1 2 3 4 5

When I am talking to someone, I worry about what they may be
thinking about me
1 2 3 4 5

I am usually worried about what kind of impression I make
1 2 3 4 5

Sometimes I think I am too concerned with what other people think
of me
1 2 3 4 5

I often worry that I will say or do the wrong things
1 2 3 4 5

Rate each item individually on the 5-point scale
from 1 (Not at all or never true) to 5 (Extremely or always true)
on how well they describe you during your work on programming projects
for this course.

The closer I am to a programming project deadline, the harder it is
for me to concentrate on it
1 2 3 4 5

When I prepare for programming projects, I worry I will not
understand the necessary material
1 2 3 4 5

While working on programming projects, I think that I am doing
awful or that I may fail
1 2 3 4 5

I lose focus on programming projects, and I cannot understand
material that I knew before the project
1 2 3 4 5

I finally understand solutions to programming projects after the
deadline passes
1 2 3 4 5

I worry so much before a programming project deadline that I am
too worn out to do my best on the project
1 2 3 4 5

I feel out of sorts or not really myself when I work on programming
projects
1 2 3 4 5

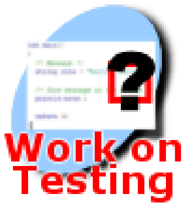
I find that my mind sometimes wanders when I am working on
important programming projects

1 2 3 4 5

After the project deadline, I worry about whether I did well enough
1 2 3 4 5

I struggle with developing programming projects, or avoid them as long as I can. I feel that what I do will not be good enough.
1 2 3 4 5

Appendix B: Adaptive Feedback Guide

Badge	Prompt	Description
	Good Start! Good start on your solution and it is good that you have started testing early, so you have earned a hint! Continue to write tests while you program to earn more hints on future submissions.	Received upon first submission that attains both correctness and coverage scores greater than zero (0).
	No progress on solution You can earn more hints by improving the behavior of your solution.	Displayed when the correctness is less than 100% and has not increased since the highest correctness score from all previous submissions on the assignment.
	**No progress on testing** You can earn hints by testing your solution more thoroughly. Consider which cases you have not yet tested.	Displayed when the coverage is less than 100% and has not increased since the highest coverage score from all previous submissions on the assignment.
	Good progress on solution! Good job improving the behavior of your solution!	Displayed when the correctness score either has maintained 100% or has increased from the previous highest correctness score.
	Good job adding more tests Good job writing more tests!	Displayed when the coverage score either has maintained 100% or has increased from the previous highest coverage score.

Badge	Prompt	Description
Keep Testing	**Small progress** You have made changes to your solution but have not improved the thoroughness of your software tests. To earn more hints you need to improve the coverage of your tests.	Displayed when the number of test methods or lines has changed, but coverage has not increased.
	Keeping pace Keep improving your test coverage to earn additional hints.	Displayed when the coverage score has increased from the highest coverage score so far, but has not met the goal (see below).
Earned More Hints	**You earned extra hints!** Good job improving the coverage of your tests! You have earned additional hints to help you keep making progress! Continue to improve and test your solution to earn even more hints.	Displayed when the coverage score has increased to or beyond the goal, with one additional hint for each multiple of 10% change in the remaining coverage. For example, an increase from 90% to 92% coverage increased 20% of the remaining coverage (2 of remaining 10%) and earns 2 hints.
	Goal: Improve to __% code coverage from your tests Current code coverage from your tests: __%	Shown in *shown goals* treatment, where the goal is initially 85%. After 85% coverage is first reached, the goal increases to covering 10% more of the remaining coverage. For example, after achieving 90% coverage, the subsequent goal will be 91% (1 of remaining 10%). If a previous submission has reached 100% coverage, the goal will continue to be 100% for the rest of the assignment submissions.