Assessing Incremental Testing Practices and Their Impact on Project Outcomes

Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant
Virginia Tech
Blacksburg, VA
ayaan|shaffer|s.edwards|fservant@vt.edu

ABSTRACT

Software testing is an important aspect of the development process, one that has proven to be a challenge to formally introduce into the typical undergraduate CS curriculum. Unfortunately, existing assessment of testing in student software projects tends to focus on evaluation of metrics like code coverage *over the finished software product*, thus eliminating the possibility of giving students early feedback as they work on the project. Furthermore, assessing and teaching the process of writing and executing software tests is also important, as shown by the multiple variants proposed and disseminated by the software engineering community, *e.g.,* test-driven development (TDD) or incremental test-last (ITL). We present a family of novel metrics for assessment of testing practices *for increments of software development work*, thus allowing early feedback before the software project is finished. Our metrics measure the *balance* and *sequence* of effort spent writing software tests in a work increment. We performed an empirical study using our metrics to evaluate the test-writing practices of 157 advanced undergraduate students, and their relationships with project outcomes over multiple projects for a whole semester. We found that projects where more testing effort was spent per work session tended to be more semantically correct and have higher code coverage. The percentage of method-specific testing effort spent before production code did not contribute to semantic correctness, and had a negative relationship with code coverage. These novel metrics will enable educators to give students early, incremental feedback about their testing practices as they work on their software projects.

KEYWORDS

incremental development, process measurement, software repository mining

ACM Reference Format:
Ayaan M. Kazerouni, Clifford A. Shaffer, Stephen H. Edwards, and Francisco Servant. 2019. Assessing Incremental Testing Practices and Their Impact on Project Outcomes. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287366

SIGCSE '19, Feb 27-Mar 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287366

1 INTRODUCTION

Software testing is an important aspect of software development, and one that is widely recognized to contribute to software quality [25, 32]. Unfortunately, teaching effective software testing often runs into many difficulties, *e.g.*, students in many US universities display a disinclination to practice regular software testing as they work toward project completion [13], and they often show poor testing ability [16, 37]. Furthermore, software testing is not a formal part of the typical CS undergraduate curriculum [25, 35].

In this paper, we focus on two major challenges for teaching software testing. First, existing approaches to assess student software testing can typically only be applied after students finish their project [15, 37]. As a consequence, by the time that these approaches provide students with feedback, they cannot really apply it to correct their practice for the same assignment. Examples of such measures include code coverage [15, 16, 37], bugs uncovered by running each student's test suite against every other student's implementation [17, 22], and coverage from mutation testing [1, 34]. In order to be able to provide students with early and continuous feedback, new approaches are necessary — to continuously assess the process that students follow *as they work on a software project.*

Second, there is a lack of strong understanding of *how specifically* software testing should be practiced [3, 28]. Existing studies measure the effects of testing methodologies [11, 19] by studying scenarios of complete adherence to a given well-known technique, *e.g.,* test-driven development (TDD) [4, 7], often with conflicting results [28, 31] [1]. However, the findings from these studies may not generalize to the context of students learning testing practices. The testing behaviors of students working on projects in uncontrolled environments (*e.g.,* at home) typically cannot be cleanly dichotomized into *fully test-first* or *fully test-last* styles of development. In our experience, since students are learning the practice, they typically follow various testing practices with varying rigor at different points in time.

To address these challenges, we propose a novel set of metrics that will enable educators to provide students with early, continuous feedback about their testing practices — as they are working on their software project. Our metrics quantitatively characterize how students write test code and production code *for work increments*, enabling the incremental assessment of their testing practices. These novel metrics measure the two testing behaviors that practitioners and software engineering researchers consider beneficial: the **balance of testing effort** (a common idea behind TDD and ITL [4, 7, 18, 19]), and the **sequence of testing effort** (the main idea behind test-first development [7]). In addition to

[1] Most industrial case studies lean toward an increase in program quality (external and internal), with a decrease in developer productivity

measuring work increments, our metrics are also *continuous* in nature: they measure the extent to which students followed beneficial practices within a work increment — instead of measuring *whether or not* they followed them. As a result, our metrics more faithfully represent (and thus allow us to study) the varying levels of student engagement with various testing practices at different times in the project life-cycle — without needing to classify their testing practices as any particular well-known technique (for neither work increments nor the whole project).

We performed an empirical study to learn the relationship between our metrics and the outcome of the software projects that students wrote. We analyzed the natural programming habits of 157 students working on complex projects with month-long life-cycles, with a median of 1.4K lines of code. We found that balancing the amount of test code and production code written during each work session (§3.2.1) — both project-wide and for specific pieces of the project — was related to better project outcomes. Additionally, we found that writing more test code *before* finalizing the relevant production code was not related to project correctness, and negatively related to test suite coverage. These findings show that our metrics can be used to incrementally provide students with feedback about their testing practices — as they work on their projects — to lead them to project success.

This paper provides the following contributions:

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

2 PROPOSED METRICS OF TESTING EFFORT

In this section, we propose a new family of metrics to measure the balance and sequence of test and production code writing effort in a continuous fashion. Figure 1 shows an example sequence of developer activity, created from synthetic data. Each group of blocks represents a *work session*, and each individual block represents test code (shaded) or production code (solid) written for a given method. Figure 2 depicts the the metrics we describe in the following sections, in terms of the example activity sequence from Figure 1. Metrics are defined in terms of **the balance of testing effort** and the **sequence of testing effort**.

2.1 Balance of Testing Effort

We examine commits in the raw event stream in terms of *time* and *location*. Commits may be bucketed based on the *work session* in which they took place, or the *methods* they were related to.

Project-wide Overall Balance of Testing Effort (POB). This metric is depicted in the first row of Figure 2. It represents the test effort in the entire project regardless of the production methods being tested or the session in which test code was written. That is, notice how the visual dimensions for color (method), ordering, and block-group (sessions) have been eliminated for this metric.

"Effort" is captured as the number of line-level code-changes (additions, removals, or inline modifications) as captured by the `git diff` command (see §3.2). We define "**testing effort**" as the percentage of effort that was devoted to writing test code. By walking the code changes for a project, we compute the total size of all changes to test code T and the total size of all changes to production code P. Then we can calculate testing effort as a percentage of overall effort spent writing test code:

$$POB = \frac{T}{P + T} \tag{1}$$

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

Method-specific Overall Balance of Testing Effort (MOB). This metric is depicted in the second row of Figure 2. In this metric, we determine testing effort while taking *location* into account, i.e., we measure the testing effort devoted to individual methods. We use the method-modification stream described in §3.2.2, and apply Equation (1) to each method in the project to compute the testing effort devoted to each method. Let this set of method-level test effort values be POB_m. Then we calculate the *MOB* metric as the median of the distribution of testing effort measured for all methods.

$$MOB = \widetilde{POB}_m \tag{2}$$

Project-wide per-Session Balance of Testing Effort (PSB). This metric is depicted in the third row of Figure 2. To determine testing effort while taking *time* into account, we compute *project-wide per-session balance* (PSB) of production code and test code. We grouped snapshots into work sessions as described in §3.2.1, and calculated the testing effort devoted to each work session using Equation (1).

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

$$PSB = \widetilde{POB}_w \tag{3}$$

Why use the median? Each project will *certainly* include some testing effort, because subjects were required to include test suites. Previous work has found that test code and production code do not co-evolve gracefully, whether written by students or by professionals [8, 9]. Using the mean testing effort, there would be no way to gauge whether this testing effort is put in gracefully over time. That is, a dearth of testing effort early on would be counteracted by an increase in testing effort at the end of the project, or vice versa.

Figure 3 shows the distribution of this metric across all projects. The distribution of medians indicates that a majority of students put in less than 20% testing effort in at least half of their work sessions. This could mean that subjects are not testing as much as they should. This would be in keeping with other work that finds that test code and production code do not co-evolve gracefully for students or for professionals [8, 9].

Method-specific per-Session Balance of Testing Effort (MSB). This metric is depicted in the fourth row of Figure 2. *MSB* determines testing effort while taking both *time* and *location* into account. That is, for the production code that is written in a given work session, we would like to know if the student tends to write *related* test code. We compute *method-specific balance over time* (*MSB*) of production

Figure 1: An example sequence of developer activity.

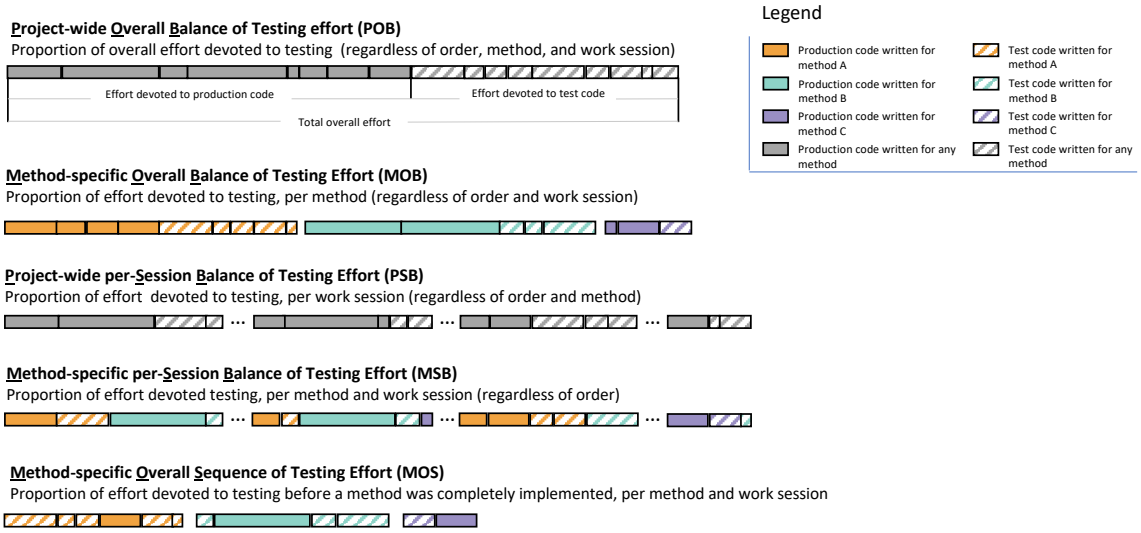

Figure 2: Measures to be derived from a programming activity event stream. Each row depicts a different method of aggregating the programming events from Figure 1.

Figure 3: Distribution of median testing effort across work sessions

code and test code. To do this, we use *method-modification events* (§3.2.2) and divide them into *work sessions* based on their timestamps (§3.2.1).

Therefore, for all changes related to a given method that were made in a given work session, we may measure the testing effort using Equation (1). We are left with test effort values at two levels of grouping: **work session** and **method**. To compute *MSB*, we first find the project-wide per-session testing balance *for individual methods*, and call it PSB_m. Then we find the median per-session testing balance *across all methods*. Therefore, this metric can be represented as a "median of medians":

$$MSB = \widetilde{PSB}_m \tag{4}$$

2.2 Sequence of Testing Effort

We are interested in the balance of test code written *before* and *after* the relevant production code, and the relationship of that balance with project outcomes.

Only a small number of methods across the entire data set were invoked in a test before being declared themselves. This could be because test-first was not practiced at all, or because test-first developers declared incomplete stubs before writing tests, or because of some other reason. We can characterize this behavior by observing the central tendency of the size of changes to test code that take place before the relevant production code has been finalized.

Method-specific Overall Sequence of Testing Effort (MOS). This metric is depicted in the last row of Figure 2. We measure *sequence* as the percentage of test code that was written for a production method before the method was *finalized*[2].

Therefore, if m is a production method, then T_b is the total size of changes to test code before m was finalized, and T_a is the total size of changes to test code after m was finalized. Note than T_b and T_a only include changes to test methods that directly invoke m. Then we compute the percentage of test code for m that was

[2]A production method is 'finalized' when it is changed for the last time.

written before m was finalized as:

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

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

$$MOS = \widetilde{MOS}_m \quad (5)$$

3 RESEARCH METHOD

We performed an empirical study to understand the impact that **balance** and **sequence** of testing effort have on the quality of projects. We are driven by the following research questions:

RQ1: How do software product outcomes relate to the **balance** of effort devoted to writing test code and production code?

RQ2: How do software product outcomes relate to the **sequence** of effort devoted to writing test code and production code?

3.1 Study Setting

We studied software systems that were developed by Computer Science students enrolled in a third year Data Structures & Algorithms (DSA) course at a large R1 public university in the US in the Fall 2016 semester. Students in this course have taken several prerequisite programming courses, most of them in Java. Subjects became acquainted with the JUnit[3] testing framework in a previous course, and in the DSA course were taught material about project management and incrementally writing and testing code by one of the authors of this paper. We examine 157 students' programming habits as they work on four course projects over the course of the semester, for a total of 415 observations. Our design is unbalanced because not all students completed all 4 projects, typically because they withdrew from the course.

Subjects were given about four weeks for each project, and their solutions had a median size of 1.4K lines of source code, including tests. Each project asked the subjects to implement one or more data structures, along with its standard operations. Students were required to write JUnit tests for their code. In addition to correctness, students were graded on the percentage of conditions covered by their test-suites, providing strong incentive to eventually write test suites with near-total code coverage.

3.2 Data Collection

To operationalize the specific concepts that will soon be described, we collected and preprocessed the activities that our studied developers performed. We developed a custom Eclipse plugin to capture developer activity in the IDE [26, 29]. Our plugin creates a git repository and creates a new commit every time the developers click the "save" button in the IDE. Because the commits are maintained by our plugin and are not under the control of subjects, we avoid some "perils" of mining these snapshots [12]. That is, histories were not "re-written" using `git rebase` or `amend` commands. This rich data set captures the detail and nuance of test-writing behavior that is necessary to calculate the metrics to be described below. Repositories were mined using the open-source tool RepoDriller [2] [4].

[3] http://junit.org

[4] Now PyDriller [38]

3.2.1 Work sessions. Following Kazerouni et al. [27], we split the event stream for a subject working on a project into *work sessions*. Work sessions are sequences of activity, and are delimited by an hour or more of inactivity. In this context, 'activity' refers to automatically captured commits (driven by file save events), and they were grouped based on their timestamps.

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

Using `git diff` output and an AST of the current snapshot, we determine which production methods and which tests for such production methods were modified in each commit. A test for a production method m is modified in a commit if the test directly invokes m. Therefore, we magnify the commit history into a series of method-specific events, meaning we expand commits into a series of method-specific `MODIFY_SELF` or `MODIFY_TEST_FOR_SELF` events.

3.2.3 Data Preprocessing. We filtered out some production methods from analysis. We excluded getters, setters, and printing methods. Exploratory analysis showed that 93% of projects in our dataset directly invoked at most 60% of all production methods in test code[5]. In other words, the distribution of testing effort across production methods is highly skewed. Both the mean and median values would be highly influenced by the skewed distribution, and would lead to a misleading 'score' for method-specific testing effort.

To account for this, we only examine the *top 60% most-tested* production methods in each project, where methods were ordered by the testing effort they were given. By examining only these most-tested methods, we are robust to the highly skewed distribution of testing effort across methods.

3.3 Data Analysis

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

- **Semantic Correctness (*C*)**: Measured as the percentage of tests passed from a suite of automated acceptance tests written by the course teaching staff.
- **Test Suite Coverage (*T*)**: Measured as the percentage of condition coverage achieved by the student's own test suite, measured using the JaCoCo[6] plugin. While subjects were required to submit test suites with nearly 100% code coverage, that might not happen until near the end of the development cycle.

We use linear mixed-effects ANCOVAs [6], with *students* and *projects* as crossed random effects. Experience suggests that no two students are the same, and this inherent variation might confound the model. Further, it is difficult to compare the programming process on different projects, even if they are assignments in the same

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

[6] https://www.eclemma.org/jacoco/

Table 1: Final ANCOVA Summary – Overall Testing Effort

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

course. Mixed-effects models allow us to control for the variation from these unaccounted-for effects.

We present *marginal* and *conditional* R^2 values [33] that describe the amount of variance in project outcomes explained by our models. Marginal R^2 values refer to the amount of variance in the outcome variable described by *fixed effects only* (in this case, our process measurements). Conditional R^2 values refer to the amount of variance in the outcome variable described by *the entire model* (in this case, the process measurements as well as individual students and assignments).

We fit the following linear mixed models for **semantic correctness** (C) and **test suite coverage** (T), using our metrics as fixed effects[7]:

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

The models are summarized in Table 1. Estimates suggest that project-wide overall balance (*POB*) of test code and production code is positively related with semantic correctness and test coverage. Additionally, test coverage has a positive relationship with method-specific per-session balance (and *MSB*) of test and production code, and a significant but weak negative relationship with the amount of testing effort put in before finalizing production code (*MOS*).

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

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

The models are summarized in Table 2. Notice that *PSB* is significantly related to both outcomes under the process-based model, while it was not significant under the overall model in Table 1.

4 RESULTS

We now present our findings regarding **RQ1** and **RQ2**.

RQ1 – Balance: Both semantic correctness (C) and test coverage (T) showed positive relationships with project-wide per-session balance of test and production code (*PSB*). That is, project implementations tend to be more semantically correct, and their test-suites tend to have higher condition coverage, when students more often write test code when they sit down to work on the project.

Test coverage was also positively related to the balance of testing effort devoted to individual methods in each work session (*MSB*) That is, condition coverage tended to be higher when more testing effort was devoted to individual methods each time the project was worked on.

Table 2: ANCOVA Summary – Process-Based Testing Effort

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

RQ2 – Sequence: Whether testing effort took place more predominantly before or after the relevant production code was finalized was irrelevant to semantic correctness. On the other hand, test coverage had a negative relationship with *MOS*, indicating that implementations tended to have higher condition coverage when a higher proportion of the testing effort devoted to a method was expended *after* the method was finalized.

5 DISCUSSION

Why is semantic correctness not associated with writing test code for relevant production code? Notice that method-specific per-session balance of test code and production code (*MSB*) was not significantly related to semantic correctness (Table 2). Does this mean that it does not matter *what* test code a developer writes, only that they write *some* test code? It would be surprising and counterintuitive to think so. However, it may be that, for this population, the variance in correctness explained by the specificity of test code written is subsumed by the variance explained by simply writing test code consistently.

Why is PSB significantly related to outcomes in the process-based model (Table 2), but not in the overall model (Table 1)? It may be that the variance explained by consistently writing software tests in each work session is subsumed by the variance in explained by maintaining an overall balance of testing effort.

How does one balance test code and production code over time, while also writing more test code after the relevant production methods have been finalized? The quantitative measure *MOS* measures the percentage of testing effort that tends to be put in before the relevant production method has been finalized, and is negatively related with the percentage of conditions covered by the test suite. These findings together suggest that while regular, balanced testing is important to test coverage, it is also important to put in testing effort after pieces of functionality have been completed.

What do the low marginal R^2 values suggest? The process measurements show significant relationships with semantic correctness and test suite coverage. Marginal (fixed effects only) R^2 values [33] for both models suggest that this effect size is small (5% for C and 10% for T). There are always likely to be numerous unexplained sources of variation when measuring human behavior. In other

[7] The notation $(1|factor)$ means that *factor* was used as a random effect.

words, there could be any number of unaccounted factors that affect the quality of software produced by developers (particularly students). Those factors are not under study. The goal of this study was to determine the impact of balancing production and test coding effort on project quality, and the models are able to answer our research questions with statistical confidence.

It could also be that the assignments in this DSA course do not "hit the right switches", that is, success did not demand a high adherence to incremental testing. After all, they were not explicitly designed to do so. It would be interesting to conduct a similar study using semester-long projects from dedicated software engineering courses, more closely imitating real-world scenarios.

6 THREATS TO VALIDITY

Internal Validity. Since we do not have strictly defined experimental and control groups, we do not claim direct causality between process measurements and project outcomes. Our sample of student developers is sufficiently large and does not suffer from a selection bias [8]. Therefore, we do not believe that this is a serious threat to validity.

Subjects had mechanical experience with the JUnit testing framework from a previous course. Multiple class periods were devoted to teaching material about project management skills, including incremental software testing, before the first project was assigned. This fact might mitigate threats from differential experience.

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

Construct Validity. The largest threat to construct validity is related to the computation of the event stream described in §3.2.2. Specifically, we link test methods and production methods if a given test method directly invokes a given production method. However, it could be that the production method was not the 'focal point' [21] of the test method, and was only being invoked to set up the test case, or to gain access to the method that was actually being tested. This could have increased the number of `MODIFY_TEST_FOR_SELF` events for some production methods, affecting our process measurements.

However, if a production method is directly invoked in a test, it is reasonable to claim that the method is being tested, regardless of developer intent. Indeed, this is the basis for code coverage, which treats methods (or statements, branches, etc.) as "covered" as long as they are invoked "somewhere in the call stack".

7 RELATED WORK

In this section we review some related work on teaching, observing, and assessing software testing.

Teaching Testing. In the classroom, Desai [14] acknowledges the challenges associated with introducing testing into the curriculum, and notes that regular, reinforced learning of testing might be better than only an introduction to it at the start of the semester. Jones [25], Edwards [15, 16] and others [35, 36] have worked toward introducing testing into the curriculum. Unfortunately, most prior work in this area tends to focus on novice programmers working on small projects, where the benefits of testing are not readily apparent [5], and on 'after-the-fact' feedback. In this paper, we study advanced students working on larger, more complex projects, and we work toward a long-term goal of continuous feedback and self-correction on the testing process.

Observing Testing Behavior. The Test My Code (TMC) plugin [39] for NetBeans records events whenever the student saves, runs, or tests code using instructor-provided tests. Hosseini et al. [23] make use of this plugin in an attempt to reason about how students check their work as they program. In contrast, this work focuses on data about students' testing habits, i.e., how they go about writing and running their own software tests.

Assessing Testing. Beller et al. [8–10] developed WatchDog, a family of IDE plugins to capture and assess software testing. Using these data, they found that most professional and student developers do not practice testing actively and that solution code and test code do not co-evolve gracefully. WatchDog assesses how closely developers followed a given testing methodology. Our metrics allow feedback for how students balance and sequence testing effort regardless of whether they follow (or not) a specific methodology.

Industrial case studies measuring the effects of TDD [11, 30, 40] tend to focus on a single implementation of a large project. As a consequence, comparisons to other projects are difficult and these studies are rarely replicated. Other existing studies do controlled experiments, which tend to involve many implementations, but they focus on learning about a specific testing methodology which is used by the developers for the whole duration of the study [18–20, 24]. In this paper, we study a different setting, in which students may follow various testing methodologies with varying rigor at different times. In addition, we strengthen the external validity of our results by comparing implementations of the same project from over 100 students (for multiple projects).

8 CONCLUSION

In this paper, we presented a novel family of metrics to assess students' incremental testing practices. We also conducted an observational study examining the relationships of test-writing behaviors, as measured by these metrics, on project outcomes. Our findings support the conventional wisdom about continuously writing tests alongside production code, but we did not find evidence to support the notion that writing tests first leads to project success.

This work is a step toward enabling continuous feedback on students' programming process. Future work will include formulating and deploying feedback based on these assessments, helping instructors to better teach incremental development.

9 ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grant DUE-1245334. The authors would like to thank Jamie Davis for valuable feedback on drafts of the paper.

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

REFERENCES

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*. ACM, New York, NY, USA, 153–160. https://doi.org/10.1145/1869542.1869567

[2] Maurìcio Finavaro Aniche. 2018. RepoDriller. https://github.com/ayaankazerouni/repodriller.

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

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

[5] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. 2002. An Experience in Integrating Automated Unit Testing Practices in an Introductory Programming Course. *SIGCSE Bull.* 34, 4 (Dec. 2002), 125–128. https://doi.org/10.1145/820127.820183

[6] Douglas Bates, Martin MÃďchler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software, Articles* 67, 1 (2015), 1–48. https://doi.org/10.18637/jss.v067.i01

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

[8] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (Much) Do Developers Test?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 559–562.

[10] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. 2016. How to Catch 'Em All: WatchDog, a Family of IDE Plug-ins to Assess Testing. In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP '16)*. ACM, New York, NY, USA, 53–56. https://doi.org/10.1145/2897022.2897027

[11] Thirumalesh Bhat and Nachiappan Nagappan. 2006. Evaluating the Efficacy of Test-driven Development: Industrial Case Studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*. ACM, New York, NY, USA, 356–363. https://doi.org/10.1145/1159733.1159787

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

[13] Kevin Buffardi and Stephen H. Edwards. 2014. A Formative Study of Influences on Student Testing Behaviors. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 597–602. https://doi.org/10.1145/2538862.2538982

[14] Chetan Desai, David Janzen, and Kyle Savage. 2008. A Survey of Evidence for Test-driven Development in Academia. *SIGCSE Bull.* 40, 2 (June 2008), 97–101. https://doi.org/10.1145/1383602.1383644

[15] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *J. Educ. Resour. Comput.* 3, 3, Article 1 (Sept. 2003). https://doi.org/10.1145/1029994.1029995

[16] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)*. ACM, New York, NY, USA, 328–328. https://doi.org/10.1145/1384271.1384371

[17] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. 2012. Running Students' Software Tests Against Each Others' Code: New Life for an Old "Gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*. ACM, New York, NY, USA, 221–226. https://doi.org/10.1145/2157136.2157202

[18] Hakan Erdogmus, Grigori Melnik, and Ron Jeffries. 2010. Test-Driven Development.

[19] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 43, 7 (July 2017), 597–614. https://doi.org/10.1109/TSE.2016.2616877

[20] Davide Fucci, Giuseppe Scanniello, Simone Romano, Martin Shepperd, Boyce Sigweni, Fernando Uyaguari, Burak Turhan, Natalia Juristo, and Markku Oivo. 2016. An external replication on the effects of test-driven development using a multi-site blind analysis approach. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 3.

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

[22] Michael H. Goldwasser. 2002. A Gimmick to Integrate Software Testing Throughout the Curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*. ACM, New York, NY, USA, 271–275. https://doi.org/10.1145/563340.563446

[23] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring problem solving paths in a Java programming course. (2014).

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

[25] Edward L. Jones. 2000. Software Testing in the Computer Science Curriculum – a Holistic Approach. In *Proceedings of the Australasian Conference on Computing Education (ACSE '00)*. ACM, New York, NY, USA, 153–157. https://doi.org/10.1145/359369.359392

[26] Ayaan M. Kazerouni, Stephen H. Edwards, T. Simin Hall, and Clifford A. Shaffer. 2017. DevEventTracker: Tracking Development Events to Assess Incremental Development and Procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 104–109. https://doi.org/10.1145/3059009.3059050

[27] Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer. 2017. Quantifying Incremental Development Practices and Their Relationship to Procrastination. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 191–199. https://doi.org/10.1145/3105726.3106180

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

[29] Joseph Abraham Luke. 2015. *Continuously Collecting Software Development Event Data As Students Program*. Master's thesis. Virginia Tech.

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

[31] Hussan Munir, Misagh Moayyed, and Kai Petersen. 2014. Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology* 56, 4 (2014), 375 – 394. https://doi.org/10.1016/j.infsof.2014.01.002

[32] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[33] Shinichi Nakagawa and Holger Schielzeth. [n. d.]. A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution* 4, 2 ([n. d.]), 133–142. https://doi.org/10.1111/j.2041-210x.2012.00261.x

[34] Zalia Shams and Stephen H. Edwards. 2013. Toward Practical Mutation Analysis for Evaluating the Quality of Student-written Software Tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA, 53–58. https://doi.org/10.1145/2493394.2493402

[35] Terry Shepard, Margaret Lamb, and Diane Kelly. 2001. More Testing Should Be Taught. *Commun. ACM* 44, 6 (June 2001), 103–108. https://doi.org/10.1145/376134.376180

[36] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, New York, NY, USA, 13–17. https://doi.org/10.1145/1140124.1140131

[37] Jaime Spacco and William Pugh. 2006. Helping Students Appreciate Test-driven Development (TDD). In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 907–913. https://doi.org/10.1145/1176617.1176743

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

[39] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding Students' Learning Using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*. ACM, New York, NY, USA, 117–122. https://doi.org/10.1145/2462476.2462501

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