Quantifying Incremental Development Practices and Their Relationship to Procrastination

Ayaan M. Kazerouni, Stephen H. Edwards, and Clifford A. Shaffer
Department of Computer Science, Virginia Tech
Blacksburg, Virginia 24061
ayaan|s.edwards|shaffer@vt.edu

ABSTRACT

We present quantitative analyses performed on character-level program edit and execution data, collected in a junior-level data structures and algorithms course. The goal of this research is to determine whether proposed measures of student behaviors such as incremental development and procrastination during their program development process are significantly related to the correctness of final solutions, the time when work is completed, or the total time spent working on a solution. A dataset of 6.3 million fine-grained events collected from each student's local Eclipse environment is analyzed, including the edits made and events such as running the program or executing software tests. We examine four primary metrics proposed as part of previous work, and also examine variants and refinements that may be more effective. We quantify behaviors such as working early and often, frequency of program and test executions, and incremental writing of software tests. Projects where the author had an earlier mean time of edits were more likely to submit their projects earlier and to earn higher scores for correctness. Similarly earlier median time of edits to software tests was also associated with higher correctness scores. No significant relationships were found with incremental test writing or incremental checking of work using either interactive program launches or running of software tests, contrary to expectations. A preliminary prediction model with 69% accuracy suggests that the underlying metrics may support early prediction of student success on projects. Such metrics also can be used to give targeted feedback to help students improve their development practices.

KEYWORDS

Educational data mining, incremental development, metrics, software development process, IDE, Eclipse, plugin, DevEventTracker

1 INTRODUCTION

Every CS student eventually reaches a point in their coursework where they must begin using good program development practices if they are going to successfully complete their programming assignments. When this happens may depend on the individual's ability or prior experience. For some students, this may happen in a traditional CS1 or CS2 course. Others successfully pass through CS1 and CS2 without developing good project management practices, but then reach the limits of undisciplined development in a later course with larger programming assignments. At our University, students are required to take a junior-level Data Structures and Algorithms course, which we will call "CS3". Unfortunately, it is typical to see 25-30% of students each semester who either drop this course, or fail to earn a grade of C or better so they can progress to later courses.

Students in our CS3 course typically complete four significant programming assignments, each with a 3-4 week life cycle. While the raw code size of these projects is not hugely greater than those found in CS2, they are generally considered to be far more difficult. Possible reasons include less scaffolding in terms of design constraints, significant use of programming techniques such as recursion, dynamic memory allocation, and pointer manipulation, and file-based data access. Typically, these projects involve far more complicated design choices, and far greater need for a rigorous testing process than projects in earlier classes.

We believe that a lack of good project management skills may be a key contributing factor to poor outcomes on major programming projects such as these. Necessary skills include incremental development (writing, testing, and debugging small chunks of code at a time), effective time management, and effective software testing. Unfortunately, poor testing ability is common at many US universities [6, 18], and students often display a disinclination to practice regular testing as they work towards project completion [3].

We believe that changing student behavior in this regard will require changing the way this material is taught, practiced, and assessed. Learning any skill requires practice [17]. But without a mechanism to capture necessary details about each student's personal development process (in contrast to outcomes in terms of successful completion of projects on time), it is not possible to assess or give feedback on that process.

The goal of this research is to capture and analyze the information needed for interventions related to improved learning of project management skills. This requires that we both collect data, and use it to deduce behavior related to processes such as incremental development, testing, and time management. Eventually, we seek to "close the loop" by providing feedback in the form of carefully designed interventions that provide timely and effective guidance. But to accurately assess incremental development and procrastination, sufficient information about the detailed behavior of students during the development process is required. This level of information is not available if one only examines work students

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICER '17, August 18-20, 2017, Tacoma, WA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-4968-0/17/08...$15.00
DOI: http://dx.doi.org/10.1145/3105726.3106180

elect to submit for assessment as they near completion of an assignment, even when students are making frequent submissions in order to determine if their program passes instructor unit tests, as happens in our programming courses.

In previous work [15], we presented *DevEventTracker*, a plugin for the Eclipse Integrated Development Environment (IDE) that captures programming events in real time as students develop. The DevEventTracker plugin is based on the HackyStat project [14]. We employed this plugin to collect data in one semester of our CS 3 course involving 166 students over four assignments, producing a total of 546 final programs. The result was a dataset of nearly 6.3 million events capturing the details of student editing, compiling, execution, and testing activities. These data were used to calculate four metrics designed to cover the various dimensions of incremental development—working early and often, incrementally checking work via either interactive program launches or software test execution, and incrementally writing software tests. Such metrics allow instructors to assess aspects of a student's programming process and make it possible to provide guidance based on that assessment.

This work builds on [15], which described DevEventTracker's functionality, defined the calculation of the four metrics, and performed a preliminary qualitative evaluation of the validity of the measures using manual inspection of code snapshots, and using student interviews to compare metrics against student subjective experience. In this paper, we develop a quantitative evaluation of the four metrics with respect to three separate outcome measures:

(1) Which metrics are significantly related to project success, in terms of producing a solution that behaves correctly?
(2) Which metrics are significantly related to finishing solutions on time?
(3) Which metrics are significantly related to how much time is spent working?

By examining how the original four metrics relate to these outcomes, we also develop and evaluate refinements with the aim of better characterizing intuitive notions of good development practices and time management practices. By using an analysis of covariance with repeated measures, we perform within-subjects comparisons to account for varying performance traits of individual students. Finally, we consider how these metrics might be used to produce a predictive model indicating whether students might be successful on a project vs. struggling.

We begin the discussion in Section 2, which presents related work on capturing and quantifying the programming process. Our method is described in Section 3, with the corresponding analysis in Section 4. Section 5 discusses our results, and their impact is presented in Section 6.

2 RELATED WORK

There is a sizable body of literature on the modeling and assessment of the student programming process. Studies done in this area range in size from a few students working on small assignments in a controlled environment to several hundred students working on several projects over the course of a semester or year. The bulk of this work focuses on the needs and behaviors of students who are just learning to program, while our work is focused on more advanced programmers.

Jadud [12] focuses on modeling the programming process of novices in the BlueJ programming environment using their compilation behaviors. He uses this information to gain a 'rough sketch' of novice programming behavior in the classroom, describing the errors novices commonly run into, the time they typically spend programming before re-compiling, and the ways in which they respond to error messages from the IDE. Jadud also developed the Error Quotient [13], a $0 \rightarrow 1$ metric developed by taking into account the type, location, and frequency of syntax errors, used to characterize the novice programming process.

Watson, et al. [20] developed the Watwin Algorithm to score a student based on their programming process in an introductory programming course. Specifically, the Watwin Algorithm scores a student based on their ability to resolve a specific type of error, compared to the time taken by their peers. Evaluation of the score showed it to be a good predictor of performance, and an improvement from Jadud's Error Quotient.

Blackbox [2] is a perpetual data-collection project that collects programming process data about Java code written by worldwide users of the BlueJ IDE—a programming environment designed for novice programmers. Altadmri and Brown [1] use this dataset to gain an understanding of the common syntactic and semantic errors encountered by novice programmers and the times taken to address them.

More closely related to our work, Carter, et al. model the student programming process of CS2-level students using the Normalized Programming State Model (NPSM) [5]. The NPSM focuses on knowing the state of the program at key points in the development process (when the program is being edited, launched, or debugged). The NPSM was used to develop predictors and explain variance for various outcome variables like assignment performance and overall course performance, making use of a holistic representation of the programming process as well as sequences of transitions between states [4].

A significant portion of previous work attempts to model the programming process based on compilation (syntactic) errors, semantic errors, or both. The NPSM models the programming process in much finer detail than other work described above, but does not model incremental development. Attempts have been made to assess and reward students based on their software testing practices [6]. However, students are typically scored *after* a significant portion of work has been done (for example, when they make a submission to Web-CAT) [7]. This means that it cannot be used as the basis for a just-in-time intervention to nudge students back on track or change their ongoing behavior.

Helminen et al. [9] capture and analyze the programming process of students using an in-browser Python editor-and-console environment in a Web Software Development (WSD) course. Data is collected in the form of 'interaction traces'. In addition to exploring syntactic and semantic errors, they explore student testing behaviors, but focus on ad-hoc testing in the console, rather than formal unit or functional test writing. They also analyze and visualize problem-solving paths taken by students [10]. Backgrounds of students varied greatly: some had previous experience with both Python and web programming, and some had little prior programming experience. As such, the course seems to be at the CS1 level or below, though this is not explicitly stated. Some data were collected

from a CS2 course, but the analysis was focused on data from the WSD course.

The NPSM work in particular is the most similar to our work discussed here. NPSM is focused on metrics of state transitions, which are abstracted from the clickstream. In contrast, our work is focused on metrics of activity type and dispersion. That is, we look at how much time or other definition of effort is being devoted to various aspects (solution development, test writing, testing, debugging), and how that effort is dispersed over time. we model the student programming process with a focus on deducing high-order behavior such as testing, incremental development, and procrastination. Further, we focus on post-CS2 students working on more complex projects. As such, our results present potential for application in professional settings as well.

3 METHOD

The data presented in this paper was obtained by administering data-collection to three sections of CS3 at Virginia Tech, a junior-level Data Structures and Algorithms course. Students programmed in Java using the Eclipse IDE. The programming projects were relatively large, with lifecycles typically 3 or 4 weeks long. We include data from all completed project submissions, including data from students who might have withdrawn from the course after completing a project. Data from students who did not give consent for their data to be used (less than 4%) were excluded from the analysis reported here. According to the points of granularity as defined in [11], our data are a combination of character-level edits, executions, and submissions. Further information about the data collection process can be found in [15].

Before undertaking analysis, some preprocessing and filtering was necessary to only include data generated while students were actually working toward project completion. During our qualitative evaluation, we found that some students tend to open their Eclipse projects and make some edits several days after projects had been graded and their final work had been submitted. While there is educational value to such activities, they are not part of the process of developing their final solution. We excluded these edits from our analysis since they are clearly not part of the organic development process for a project, and tend to corrupt the incremental development scores due to the large amount of time between project completion and these after-the-fact activities. We also excluded projects that were worked on for less than 1 hour (that is, projects started by students but with no meaningful attempt to finish).

We use intervals between event timestamps to calculate the time spent on projects. We break up the project into *work sessions* that are separated by at least 1 hour of inactivity, and add up the times for each work session. This ensures that time calculations are not inflated by long periods of inactivity where no actual work is being done.

After filtering, the dataset consists of the work of 162 students working on 545 programming projects turned in to four assignments. Not every student completed every assignment, since some students dropped the course, and others may have missed an assignment for personal reasons. Project correctness was measured as the percentage of instructor-written software tests used as the reference for grading correctness in each assignment. We do not

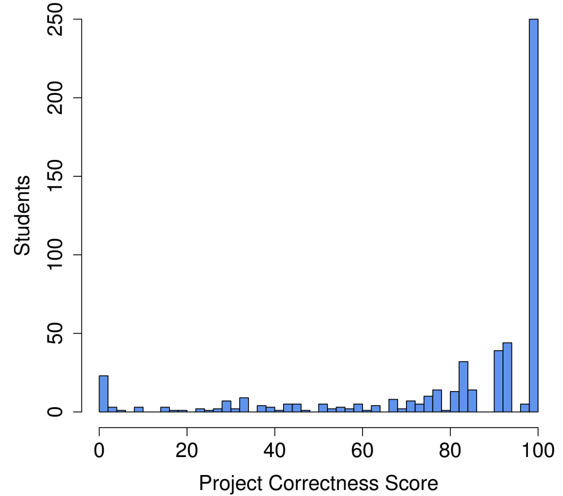

Figure 1: Distribution of correctness scores.

include bonuses/penalties due to early or late completion, manual grading criteria, or conformance to coding style standards when examining correctness. Results for all statistical tests use $\alpha = 0.05$ to determine significance unless otherwise noted.

To assess program correctness, a key outcome under consideration, we measure the percentage of instructor-written reference tests that a student's final solution passes. By examining the distribution of correctness scores over the class, there is a clear separation between students who are able to successfully "solve" a problem by creating a working solution, and those who create buggy or incorrect solutions.

Figure 1 shows the distribution of correctness scores. There is a clear trough just under a perfect score, with approximately half (47%) of the class scoring very close to perfect, and the remainder (53%) scoring noticeably lower. By choosing a cutoff of 95%, we can partition the class into projects that have successfully "solved" the behavior required for an assignment, and those that have imperfect solutions. As a result, we will examine differences in key metrics between projects that achieve this threshold and those that do not.

4 ANALYSIS

In this section, we analyze the various metrics derived from the extensive log data collected by DevEventTracker. We investigate the relationships our metrics have with the key outcome variables of project correctness, time spent on the project, and time of completion. The analyses are presented as a method of evaluating our metrics' relationships with various aspects of the programming process, particularly those that would be affected by the practice of incremental development.

4.1 Working Early and Often

A previous study by Edwards et al. [8] found that students who began submitting their work earlier also tended to score better

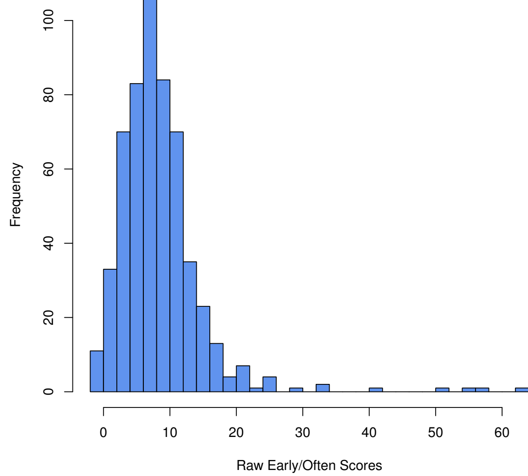

Figure 2: Distribution of Early/Often scores—the mean number of days before the deadline when editing occurred.

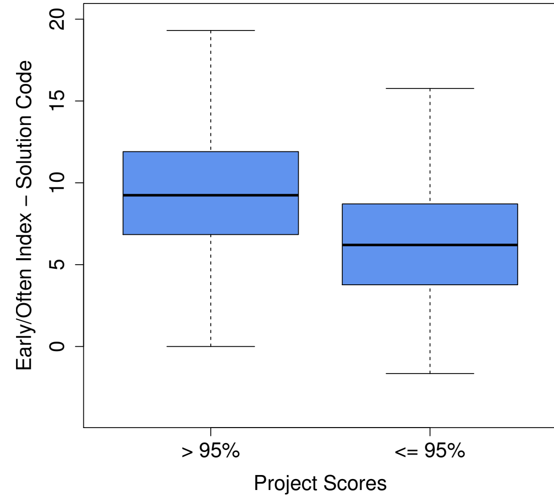

Figure 3: Comparison of solution edit times between projects that correctly solved an assignment, and those that did not.

on the project, with no significant difference in the time spent on the project. This is consistent with research on procrastination that indicates procrastination can lower scores. However, with DevEventTracker data, we have the ability to examine all programming activity, back to the initial creation of the student's project. In previous work [15], we defined the *Early/Often Index* as a way of using this data to capture the intuitive notion of procrastination. If E is the set of all edits events, then the Early/Often Index is defined as:

$$\text{earlyOften}(E) = \frac{\sum_{e \in E} \text{size}(e) * \text{daysToDeadline}(e)}{\sum_{e \in E} size(e)}$$

This definition amounts to the mean edit time, across all individual character-level changes in the project, with time measured relative to the assignment deadline—the average time at which a given character was edited. Since this measure is a time-based average, we present it as a (real) number of days, representing the mean number of days before the deadline across all character-level edits.

For this measure, we chose the mean because it is a common measure of central tendency, and it can be more sensitive to potential skew in the data. In this case, because skew can play an important role, where procrastination leads to larger edits late in the development period, the mean may provide greater discrimination. Students who work early and often will receive higher scores for this metric (representing more days in advance of the deadline) than students who tend to do more work close to the project deadline. Figure 2 shows the distribution of early-often scores across the data set.

While [15] define only the mean, following this strategy, we also can calculate an edit median in a similar way. Because skewness is an important consideration in the distribution of times for student development actions, we also considered using *Pearson's second coefficient of skewness* to characterize lopsided spread. Pearson's coefficient is defined as $(mean - median) * 3/\sigma$, and gives a measure of skew normalized into units based on the standard deviation, with the sign of the measure indicating the direction of the skew. However, since this coefficient is a linear combination of the mean and median, it does not add explanatory power to any linear regression models. Instead, we opt to use both mean and median together, which captures this same notion of skewness.

Further, while both means and medians can be calculated across all edits, we also can calculate these measures separately for edits to software tests and for edits to the solution code. These measures help give an idea of *when* code is typically written for a project. Nevertheless, they are highly correlated, with R = 0.87 between the solution edit time mean and median, R = 0.91 between the solution edit time mean and the test edit time mean, and R = 0.84 between the test edit time mean and median. Still, we investigated all four for completeness.

To test for relationships with the outcome variables, we used a mixed model ANCOVA. Means and medians for both solution editing and software test editing were used as continuous independent variables, and were all simultaneously treated as covariates with the dependent variable of interest. Students served as subjects, and assignments were treated as repeated measures (with unequal variances) on the same subject, to perform within-subjects comparisons in the ANCOVA.

With respect to project correctness, we used the percentage of instructor-written software tests that the student's final solution could pass as the dependent variable in the ANCOVA. We found that solution mean edit times were significantly related to project correctness ($F = 16.2$, $p < 0.0001$). In other words, students who worked on their solution earlier were more likely to produce more correct programs. This is consistent with the earlier result from

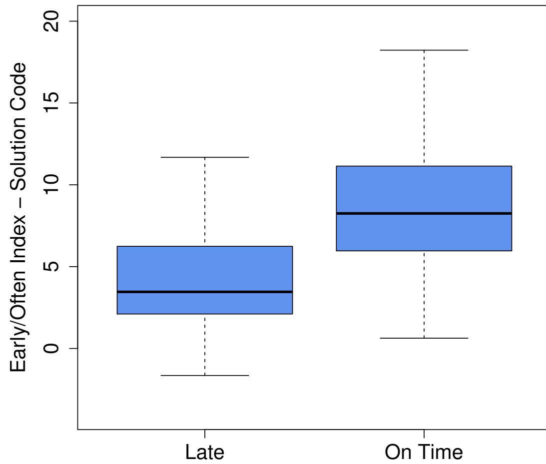

Figure 4: Comparison of solution edit times between projects that were on time versus late.

[8], but now using actual development log data instead of just submitted work. Figure 3 illustrates this relationship by showing the distribution of solution mean edit times for projects with greater than 95% correctness scores versus those with lower scores. Note that when both the solution edit mean and median times were considered, the median was not significant (F = 0.73, p = 0.39). The same ANCOVA indicated that the test edit median time was also significantly related to project correctness (F = 10.0, p = 0.0018; the mean was not significant, F = 0.06, p = 0.80). These differences were present, even when controlling for student variability using a within-subjects test, indicating that these differences were not simply due to individual student traits.

With respect to finish times, we used the number of hours before the deadline when the student's final work was submitted as the dependent variable in the ANCOVA. We found that both solution mean edit times (F = 55.9, p < 0.0001) and solution median edit times (F = 28.7, p < 0.0001) were significantly related to finish time, with earlier early/often scores corresponding to earlier finish times. This is as one would expect, since working earlier does allow a greater opportunity to finish earlier. This is also similar to the results in [8], where earlier submission times were associated with earlier completion times. Figure 4 illustrates this relationship by showing the distribution of solution mean edit times for projects that were completed on time versus late.

Finally, with respect to total time spent working, we used the number of hours spent as the dependent variable in the ANCOVA. We found that only the test edit time median (F = 10.8, p = 0.001) was significantly related to total time spent, with earlier edit times associated with slightly longer total time spent. It is notable that the median (not mean) was significant in this case, since the median is less sensitive to skewing when there are outliers very early in the development process but more editing occurs in a smaller time frame closer to the deadline. The median edit time marks the point at which half of the edit activity has already been completed, regardless of its distribution over time. One might interpret this to mean that students who do a significant portion of the work earlier have more opportunities to invest time on the project later. Or, instead, it may be that students who start very early have to spend more time figuring out details that are only clarified in the assignment specification for everyone else at a later date. By performing a similar repeated measures ANCOVA to examine the relationship between time spent and program correctness, we find no evidence of a significant relationship (F = 1.9, p = 0.17).

In summary, projects with high early/often scores (more specifically, solution edit mean times) tended to be more correct and to be finished earlier. While earlier median edit times for software tests were associated with students who spent more time on their projects, this was not directly associated with higher scores. Our calculation of time spent on a project is more accurate than the previous study. Instead of using first and last submission times as proxies for beginning and completing a project, DevEventTracker allows us to get the *actual time spent* developing the project, by giving us work session information directly from students' local Eclipse environments.

4.2 Test Writing

Incremental test writing—that is, writing software tests to check your own work as you go—is another development practice we wished to examine. One aspect of test writing has already been discussed in Section 4.1: the test edit time mean (and median). Those measures capture part of what it means to "test early", but do not directly capture how close in time the writing of code and tests happen.

It is also worth noting here the relationship with *test-driven development* (TDD). In current practice, most developers interpret TDD to require writing the software tests for a feature *first*, before writing the corresponding piece of the solution. Here we are using a less stringent notion for incremental testing: whether you write the test or the solution first is less important than whether you do them together, in small chunks. In other words, we are more interested in indicating when students practice a "code a little, test a little" style of programming, regardless of whether they strictly write tests before writing solution code.

To capture this notion, the *Incremental Test Writing* measure is defined [15] as the difference between the mean time of solution edits and the mean time of test edits. A small number indicates that the central tendency for test editing somewhat closely follows the central tendency for solution editing, while a much larger value indicates that test editing on average occurs closer to the end of development—that is, noticeably after the bulk of the solution code was written. While this metric is calculated as a combination of early/often indices for test code and solution code, it is important to note that it has nothing to with procrastination. It is only concerned with assessing how regularly the student writes tests during the project life cycle, regardless of when in the life cycle this occurs. If $SE \subset E$ is the set of all solution edits and $TE \subset E$ is the set of all test edits, then Incremental Test Writing can be calculated as:

$$\text{incTestWriting}(E) = \text{earlyOften}(SE) - \text{earlyOften}(TE)$$

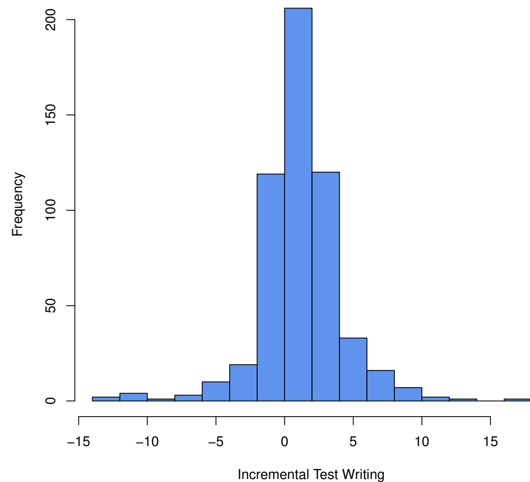

Figure 5: Distribution of Incremental Test Writing scores—the number of days that test writing occurs after solution writing, on average.

Figure 5 shows the distribution of Incremental Test Writing scores across all projects.

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

At the same time, the DevEventTracker data can be used to provide visual analysis of a student's programming process. Helping students to visualize their own programming process and to compare that against their peers might encourage them to introspectively consider where improvements could be made. This could provide useful feedback during project life cycles. Figures 6, 7 and 8 show "skyline plots" of the programming process for projects with varying levels of incremental test writing and procrastination. The plots depict step-functions for the amount of test code and solution code written over time. The width of each step is the length of the work session, and the height (from the x-axis) is the amount of code written in each work session. Each work session is separated by at least 3 hours of inactivity [1]. Therefore, work sessions that look as though they lasted multiple days (particularly in Figures 6 and 7), appear because the student settled down to work on the project multiple times, without stopping for a period of at least 3 hours.

[1]This is less granular than the threshold of 1 hour used to calculate time spent on projects, which results in more cluttered visualizations that are harder to understand.

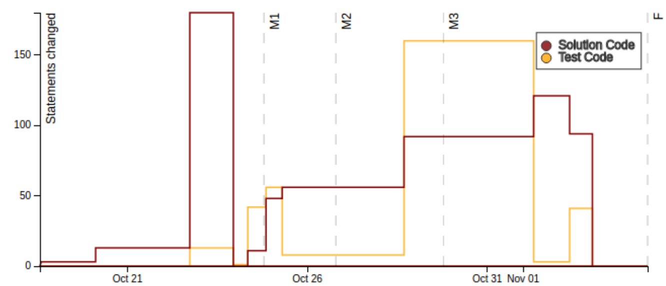

Figure 6: An example of a project with unsatisfactory test writing—notice the spike in the amount of test code written as the due date approaches. This project was in the 45th percentile for Incremental Test Writing, and in the 49th percentile for the Early/Often Index.

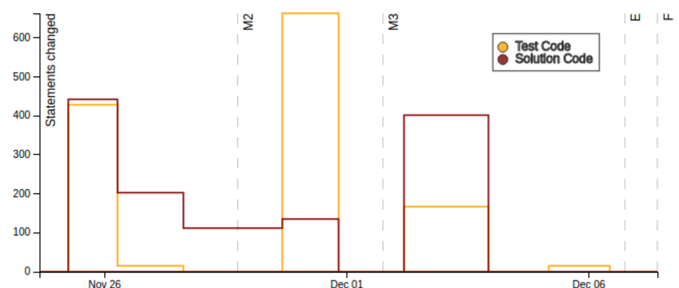

Figure 7: An example of a project with an intermediate metric score for test writing, with room for improvement. Notice the irregular bursts of test code writing, and that work started after the first Milestone was due. This project was in the 64th percentile for Incremental Test Writing, and in the 69th percentile for the Early/Often Index.

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

4.3 Program and Test Launches

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

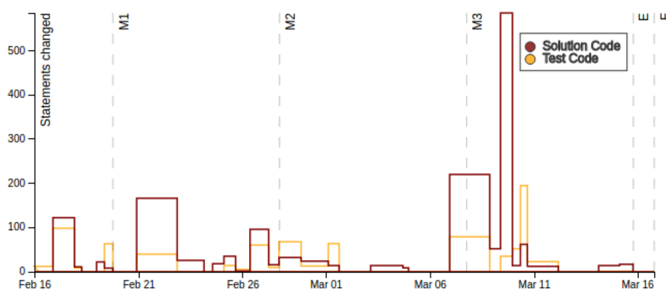

Figure 8: An example of a project with model scores under the test writing metric—notice how the test code and solution code follow similar patterns over time. This project was in 90+ percentile for both Incremental Test Writing and the Early/Often Index.

The DevEventTracker plugin tracks both interactive program launches and software test executions, and also records the pass/fail outcomes of software tests, providing all of this information in the logged data for analysis. The metrics originally presented in [15] included two aimed at capturing the amount of code students typically write before either launching the program interactively or running software tests on it.

The incrementalChecking metric [15] considers the time between an individual character-level edit action and the next subsequent program launch or test execution. It is the mean of these times over all edits, representing the average amount of time that passes between a code edit and the next program launch or test execution. Using the same ANCOVA analysis procedure, we found no evidence for a significant relationship between this measure and project correctness, time of completion, or time spent.

The incrementalTestChecking metric [15] is almost identical, except that it only considers software test executions and does not count any interactive program launches. While we advocate software testing in CS3 (and also in CS1 and CS2), experience suggests that at least some students do not follow it, and in proposing measures we wanted to account for students who self-checked their work without using software tests. However, it turned out that the event data showed students used software test executions much more commonly than interactive program launches. Test launches were significantly more frequent than normal program launches ($t = 13.977, p$¡0.0001, test = 229.23, normal = 55.66). 83% of projects had more test launches than solution launches, and test launches made up approximately 80% of all launches across projects.

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

5 DISCUSSION

Quantifying the programming process in terms of incremental development and procrastination is a non-trivial task, primarily because of the lack of ground truth against which to judge any metrics. However, our suite of metrics have provided some encouraging qualitative as well as quantitative results. However, based on this study, we can formulate answers to the research questions posed in Section 1.

(1) Which metrics are significantly related to project success, in terms of producing a solution that behaves correctly?

In this study, we measured project success using the percentage of instructor-written reference tests passed by a student's final submission for an assignment. We found a statistically significant relationship between project correctness and the mean edit times for solution edits, and also the median edit times for test edits. These metrics provide a quantified representation of procrastination, and these findings are in keeping with well-known effects of procrastination while working toward project completion [19]. However, we did not find significant relationships with the incremental test writing metric, or with either incremental checking metric based on when students launched their programs (or tests).

(2) Which metrics are significantly related to finishing solutions on time?

In this study, we measured finish times using the time of submission by a student of their final work on an assignment. Both solution mean and median edit times were significantly related to finish times. Again, these results are consistent with the known effects of procrastination. Better performance on the Early/Often Index is related with a higher likelihood of completing a project on time, regardless of the amount of time spent on a project. These findings align with a separate study that measured procrastination and its impact on project performance, using a different data source, suggesting that the Early/Often Index is an accurate measure of procrastination [8]. At the same time, we did not find significant relationships with the incremental test writing metric, or with either incremental checking metric.

(3) Which metrics are significantly related to how much time is spent working?

Although there was no evidence for a relationship between time spent and project correctness, we did find a significant relationship between the median time for test edits and total time spent. This result was not observed in prior work, although it is based on more accurate, finer-grained data collected over the whole development cycle, rather than only timestamps of submission attempts made by students as they near completion of their work. It is plausible that this effect may be related to the larger span of opportunities available over the longer period of time between project initiation and the deadline for students who start earlier, although more work would be needed to confirm this. Yet, since there is no evidence for a relation between time spent and project correctness, this extra time does not appear to translate directly into a grade advantage. Instead, it may simply mean more time to work at a slower pace under less stressful conditions, and more time for reflection while working.

6 APPLICABILITY

In this section we discuss methods by which our metrics could be used to support class interventions. To support possible interventions it is necessary to develop a predictive model based on these metrics. While complete predictive model development is outside the scope of this paper, we did explore predicting program correctness prediction in particular. Because both solution edit mean times and test edit median times were significant, we constructed a response surface model using these two as continuous independent variables, with project correctness score as the continous dependent variable to be predicted. This model was statistically significantly related to correctness scores, and we used its prediction equation as the input to generate a partition model used to classify program solutions as either "solved" (in the group scoring greater than 95% correctness) or not. This predictive model was 69% accurate at classifying the students in our sample (where SE represents all solution edits and TE represents all test edits):

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

While this is by no means a validated prediction model, it suggests that such models can achieve some degree of accuracy. Further, if project size can be estimated, medians (or approximations of means) can also be estimated. Since successful (solved) project solutions have mean edit times more than a week ahead of deadlines, it should be possible to predict performance with some degree of accuracy with some degree of lead time before the deadline. Developing and validating an appropriate model is important future work, although this paper lays the necessary groundwork by identifying the measures most appropriate for use in such a model. There are also many potential interventions that could be driven by such a predictive model.

Adaptive emails: Previous work [16] has discussed the effects of interventions with adaptive feedback on students' procrastination behaviors and project performance. Students were sent emails with feedback generated from data about their last submitted work, and the effects were positive when compared to a control group. Our suite of metrics could be applied in a similar fashion. The feedback generated from data made available by DevEventTracker could be far more specific than that reported in [16].

A learning dashboard: Visualizations such as those seen in Figures 6, 7, and 8 could be part of a web-based learning dashboard. Graphs showing the progression of solution code and test code over time could be automatically generated for each student, providing visual feedback of their programming process.

A leaderboard: The information described above could also be presented in a way that relates the individual's performance to the rest of the class. Making students aware of their standing in the class could provide more incentive for self-improvement than simply informing them of their own programming practices.

Project grade: A portion of the project grade is already allocated based on things other than correctness, such as the percentage of code covered by students' own tests, and the quality of the comments and program style. A natural step is to allocate a portion of the project grade based on an assessment of incremental development and time management practices. However, this opens up some possibilities for gaming the system to artificially raise the metrics, so work would need to be done to make the metrics robust to these activities.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we built on previous work [15] that enabled us to collect fine-grained programming process data from students' local Eclipse environments. In order to accurately assess abstract concepts like incremental development and procrastination, we developed a set of four metrics that we believe cover different dimensions of both concepts. We built on previous qualitative evaluations by conducting a quantitative analysis to investigate the relationships our metrics have with three identified outcome variables, and we used these results to make a case for the accuracy and correctness of our calculations.

We found a number of significant relationships between our metrics and project correctness, time of completion, and total time spent working on the project. Although we hoped to characterize the effects of incremental development actions, it appears that the most significant effects come from effective time management practices—that is, working on a project early and often, as characterized by mean and median edit times for solution code and for test code, and thus avoiding the pitfalls of procrastination. Unfortunately, other metrics regarding incremental test writing, or incremental self-checking of work using interactive program launches or execution of software tests were not significant. Those implications are discussed in Section 5. These relationships are not novel ones uncovered by our metrics. Rather, we take advantage of their intuitive and well-known nature to provide legitimacy to our metrics. The key issue is that these metrics provide an opportunity for meaningful feedback to students, either in an on-going basis during a project's development cycle, or as assessment feedback.

This research is a work in progress, and naturally there is room for improvement and future work. The most important next step is to develop and validate a predictive model that can be used for applying interventions. This can be followed by evaluation of the effectiveness of interventions, which can be measured in terms of the metrics found to be significant in this paper.

In addition, the DevEventTracker data is rich enough that this work barely touches the surface. It offers the possibility to examine the effects of debugger use, on its own or in relation to testing activities; examine issues regarding test quality, and what role it plays in self-checking or incremental development; examine predicting time to completion, to keep students informed of when they are likely to finish or whether they are likely to be late, with as much advance notice as possible; and perform deeper examinations of code changes captured in the git snapshots that track the event stream collected by DevEventTracker. All these and more are enabled by this style of data collection, which helps to open new avenues of data-driven research about student programming activities.

8 ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation under grants DUE-1245334 and DUE-1625425. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 522–527. DOI:https://doi.org/10.1145/2676723.2677258

[2] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 223–228. DOI:https://doi.org/10.1145/2538862.2538924

[3] 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. DOI:https://doi.org/10.1145/2538862.2538982

[4] Adam Scott Carter and Christopher David Hundhausen. 2017. Using Programming Process Data to Detect Differences in Students' Patterns of Programming. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 105–110. DOI:https://doi.org/10.1145/3017680.3017785

[5] Adam S. Carter, Christopher D. Hundhausen, and Olusola Adesope. 2015. The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 141–150. DOI:https://doi.org/10.1145/2787622.2787710

[6] 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). DOI:https://doi.org/10.1145/1029994.1029995

[7] 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. DOI:https://doi.org/10.1145/1384271.1384371

[8] Stephen H. Edwards, Jason Snyder, Manuel A. Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. 2009. Comparing Effective and Ineffective Behaviors of Student Programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09)*. ACM, New York, NY, USA, 3–14. DOI:https://doi.org/10.1145/1584322.1584325

[9] Juha Helminen, Petri Ihantola, and Ville Karavirta. 2013. Recording and Analyzing In-browser Programming Sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling '13)*. ACM, New York, NY, USA, 13–22. DOI:https://doi.org/10.1145/2526968.2526970

[10] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How Do Students Solve Parsons Programming Problems?: An Analysis of Interaction Traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. ACM, New York, NY, USA, 119–126. DOI:https://doi.org/10.1145/2361276.2361300

[11] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR '15)*. ACM, New York, NY, USA, 41–63. DOI:https://doi.org/10.1145/2858796.2858798

[12] Matthew C Jadud. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40.

[13] Matthew C. Jadud. 2006. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research (ICER '06)*. ACM, New York, NY, USA, 73–84. DOI:https://doi.org/10.1145/1151588.1151600

[14] Philip M Johnson, Hongbing Kou, Joy M Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. 2004. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering, ISESE'04*. 136–144.

[15] 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 (ITiSCE '17)*. ACM, New York, NY, USA. DOI:https://doi.org/10.1145/2361276.2361300

[16] Joshua Martin, Stephen H. Edwards, and Clfford A. Shaffer. 2015. The Effects of Procrastination Interventions on Programming Project Success. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*. ACM, New York, NY, USA, 3–11. DOI:https://doi.org/10.1145/2787622.2787730

[17] Allen Newell, Paul S Rosenbloom, and JR Anderson. 1981. Mechanisms of skill acquisition and the law of practice. *Cognitive skills and their acquisition* 1 (1981), 1–55.

[18] 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. DOI:https://doi.org/10.1145/1176617.1176743

[19] Piers Steel. 2007. The nature of procrastination: a meta-analytic and theoretical review of quintessential self-regulatory failure. (2007).

[20] C. Watson, F. W. B. Li, and J. L. Godwin. 2013. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*. 319–323. DOI:https://doi.org/10.1109/ICALT.2013.99