SIGCSE: G: Quantifying the Programming Process to Help Teach Incremental Development

Ayaan M. Kazerouni
Virginia Tech
Blacksburg, VA
ayaan@vt.edu

ABSTRACT

Assessment of software tends to focus on postmortem evaluation of metrics like correctness, mergeability, and code coverage. This is evidenced in the current practices of continuous integration and deployment that focus on software's ability to pass unit tests before it can be merged into a deployment pipeline. However, little attention or tooling is given to the assessment of the software development process itself. Good process becomes both more challenging and more critical as software complexity increases. Real-time evaluation and feedback about a software developer's skills, such as incremental development, testing, and time management, could greatly increase productivity and improve the ability to write *tested* and *correct* code. My work focuses on the collection and analysis of fine-grained programming process data to help quantitatively model the programming process in terms of these metrics. I report on my research problem, presenting past work involving the collection and analysis of IDE event data from junior level students working on large and complex projects. The goal is to quantify the programming process in terms of incremental development and procrastination. I also present a long-term vision for my research and present work planned in the short term as a step toward that vision.

1 MOTIVATION

Writing working software is a crucial skill both for students in programming-intensive courses and professional developers in the field. We believe that the process used to develop a complex piece of software must also be important to its success. The main focus of my research is to quantitatively characterize a developer's programming process. By measuring the programming process, I can empirically evaluate its adherence to known best practices in software engineering. Once I can characterize the observed process, I hope to build tools that provide developers with intelligent and timely feedback when they are in danger of straying from those practices. Changing developer behavior requires changing how this material is taught, and more importantly, changing how learning and practice are assessed. To provide proper assessment, we need information about how developers conduct their project development as they work on solutions. To do this, one must collect detailed data about the programming process.

We use as our primary test-bed students in a junior level Data Structures and Algorithms course at Virginia Tech, and how they go about developing large and complex software projects. While the raw code size of these projects is not much greater than those found in a typical CS2 course, they are generally considered to be far more difficult. Typically, these projects include less scaffolding in terms of design constraints; the use of complex programming techniques such as recursion, dynamic memory allocation and pointer manipulation, and file-based data access; and far more complicated design choices. Consequently, there is a greater need for rigorous testing process in these projects than for projects given in earlier classes.

We believe that a lack of good project management skills may be a 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 [8, 23], and students often display a disinclination to practice regular testing as they work towards project completion [4]. In spite of this, software testing is not a formal part of the typical undergraduate CS curriculum [16, 22]. Desai, et al. [7] conducted a survey on the teaching and use of test-driven development (TDD) in academia. They concluded that TDD helps students with complex projects and increases student confidence, but also acknowledged the challenge in knowing when to introduce software testing in the curriculum. They note that regular reinforced learning could be important to successfully introducing students to TDD; when students were briefly introduced to it at the start of the semester, TDD was not preferred [19] and not widely used [1].

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 collect data and use it to deduce behavior related to processes such as incremental development, testing, and time management. Using the results of these analyses, we seek to "close the loop" by providing feedback in the form of carefully designed interventions that provide timely and effective guidance. The students under study are advanced undergraduates, typically only two or three semesters removed from professional software developers entering the workforce. The tools and techniques developed in this research are designed to be applicable in an industrial setting as well as an academic one. In the long term, this work will contribute to the standardization and adoption of continuous software assessment techniques that include not only the final product, but also the process undertaken to produce it. If this can demonstrably improve development practices, the benefits would include a sped-up product life-cycle and an improvement in code quality.

2 RELATED WORK

The Web-Center for Automated Testing (Web-CAT) [9] is a web-based automated grading system that allows students to make multiple submissions to an assignment and receive immediate feedback. Feedback can be about correctness, code style, or code coverage

by student-written tests. While Web-CAT's model of multiple submissions affords us the ability to get a rough idea of the project's trajectory over time, submission level data is considered the least granular form of student-data [12]. To assess incremental development and testing, we would need to collect data *during development*, rather than at submission.

To this end, we developed DevEventTracker [17, 20], an Eclipse plugin that collects event-level data. According to the data types described in [12], DevEventTracker collects data for: *executions, compilations, file saves, line-level edits*. It also captures periodic Git snapshots, providing a rich representation of a project's evolution over time. Coupled with submission data from Web-CAT (and the associated results from instructors' unit tests), this provides for more robust analysis of a student's programming process.

The Test My Code (TMC) plugin [24] for NetBeans records events whenever the student saves, runs, or tests code using tests provided by the instructor. Hosseini, et al [11] make use of this plugin in an attempt to achieve goals similar to ours, but with some differences in the type of data collected.

Hackystat [15] is an open-source project from the University of Hawaii. It consists of sensors integrated into the user's development environment of choice that trigger when certain actions are taken. DevEventTracker builds upon Hackystat, using its client-side protocols and preexisting sensors in conjunction with our own to send data to the server.

Carter, et al. [5, 6] use the Normalized Programming State Model to represent the programming process as a series of state transitions. They perform predictive analysis for project scores based on the time spent in each state. Other student data-tracking research includes [3, 14, 25].

3 APPROACH

Continuously collecting development events as students program gives us a unique insight into the development process of the typical student. The major benefit from this augmentation is twofold: 1) Our data are no longer limited by when a student decides to make a submission. 2) Since we are collecting data from the IDE itself, we have access to events that are not available through Web-CAT alone. With the data in hand, the next step is to use it to empirically determine if a student is practicing *incremental development*. We unpack the notion of incremental development into three aspects: 1) time management, 2) periodically launching software tests or the program itself, and 3) regularly writing software tests.

We collected data from three sections of a post-CS2, junior-level Data Structures and Algorithms course at Virginia Tech. After filtering, the dataset consists of the work of 162 students on 545 programming projects turned in across four assignments (approximately 6.3M events). Using this data, we developed the following measures to assess adherence to incremental development [17]:

3.1 Early/Often Index

This metric is a quantification of procrastination on a programming project. If E is the set of all edit events, then the Early/Often Index is defined as:

$$\text{earlyOften}(E) = \frac{\sum_{e \in E} \text{size}(e) \cdot \text{daysToDeadline(e)}}{\sum_{e \in E} \text{size}(e)} \quad (1)$$

This definition amounts to a measure of how early and how often a student works on a project, defined in relation to the due date for a project. Equation 1 gives the mean time at which a given character was edited, measured relative to the assignment deadline. A larger value would indicate that more work was done more days before the deadline, while a smaller (possible negative) value would indicate that work was done close to or even after the deadline. Therefore, a **a larger value is better**.

3.2 Incremental Checking and Test Checking

Another key notion of working incrementally is self-checking one's work periodically, as each small chunk nears completion. Alternatively, it might also involve interactively running a program to manually check functionality. If E is the set of edits, then Incremental Checking and Incremental Test Checking may be defined as:

$$\text{incChecking}(E) = \frac{\sum_{e \in E} \text{size}(e) \cdot \text{hoursToNextLaunch}(e)}{\sum_{e \in E} \text{size}(e)} \quad (2)$$

$$\text{incTestChecking}(E) = \frac{\sum_{e \in E} \text{size}(e) \cdot \text{hoursToNextTestLaunch}(e)}{\sum_{e \in E} \text{size}(e)} \quad (3)$$

Equations 2 and 3 give us values governed by the amount of code written and the time that passes before the program (or tests) are launched next. Therefore, if a developer writes a lot of code before checking that it works, this would return larger values, and vice-versa, so **smaller values are better**.

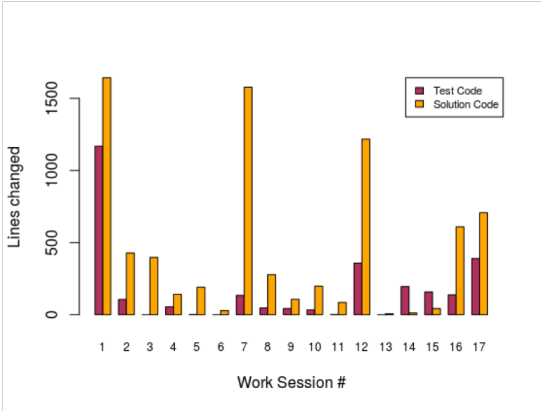
3.3 Incremental Test Writing

First, we sought to measure the difference in central tendency of edit times for solution code and test code. We calculate Early/Often indices separately for *solution code* and *test code*. Then we find the difference between the two values. The result is a number whose value is governed by the average amount of time that passes between the writing of solution code and test code, and by the amount of code written for each. A smaller number indicates that the central tendency for test editing somewhat closely follows the central tendency for solution editing, while a large value would indicate that test editing on average took place after the bulk of solution code was written. Therefore, **smaller values are better**. 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 is defined as:

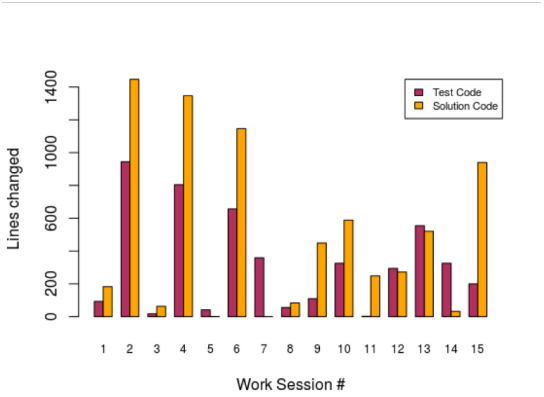
$$\text{incTestWriting}(E) = \text{earlyOften}(SE) - \text{earlyOften}(TE) \quad (4)$$

Note that Equation 4 is not related to procrastination. However, this measure of test writing did not display much ability to explain variance in program correctness, indicating that the difference in central tendency between the time of writing solution code and test code may not be of high importance for this population.

As a next step, we quantified the *balance* of writing test code and solution code over time. Using the heuristic described in [18], we divided the programming process into *work sessions*, where work sessions are sequences of events divided by an hour or more of inactivity. Then, for each work session, if S the total size of changes to solution code, and T is the total size of changes to test code, we



(a) Most of the testing is done in the first work session, and then testing becomes relatively sparse until the end of the project.



(b) Test code seems to co-evolve gracefully with solution code.

Figure 1: The work of two students on the same project in Fall 2016.

calculate *test effort* as:

$$TE = \frac{T}{S+T} \tag{5}$$

By applying Equation 5 to the work done in individual work sessions, we obtain a series of test effort values TE_W. The median of the values in TE_W gives us a measure of central tendency of *how much code a student writes each time they work on the project.*

In addition to this "project-wide" balance of testing effort over time, we used repository mining and static analysis techniques to determine the balance of testing effort *devoted to individual methods*, and empirically determine its relationship with project outcomes.

4 RESULTS

4.1 Qualitative Student Interviews

We decided to use individual interviews to gather student opinions. As we neared the end of the Fall 2016 semester, we generated incremental development scores for students on the projects they had worked on until that point. Scores were generated by running the raw event data through in-house Python processing scripts to calculate the metrics [1]. Then we interviewed students in depth about their programming practices.

Six of the seven students found our assessment **accurate**. The model showed the ability to detect differences between the *same student's* programming process on different assignments. The Early/Often Index showed the most accuracy, according to interviewees. Students consistently received high scores on Incremental Checking and Test Checking except in one case that was determined to be because of transient issues with data transmission. Two students expressed surprise at their low scores on Incremental Test Writing, but other students found the model's assessment to agree with their self-perceived testing habits. Only one of the seven students did not find the model **useful**. This was the student who had received an inaccurate assessment due to data transmission errors.

4.2 Manual Examination of Git Snapshots

We carried out a second type of evaluation by manually inspecting Git repositories maintained by DevEventTracker. Twelve projects were randomly sampled from the pool of submissions. The inspection focused on checking whether our assessment of incremental development matched the "actual programming process" of the student (as seen in the Git revision histories).

Eight of the twelve projects had unsurprisingly low scores (< 80 on a normalized 100-point scale) for **working early and often**, since they either had breaks of several days where no work was done, or they were worked on late in the project lifecycle. One out of the remaining four projects received a surprisingly high Early/Often score, since the project was started within the last two days. The final three projects received expected high scores.

We evaluated **program launching behavior** using a combination of raw DevEvent data and Git snapshots. For two consecutive 'Launch' events, we stepped through revisions for commits made between the two launches. Doing this for several random pairs of launches gives an idea of the usual amount of work done by that student before the program is launched. Only one project received a low score for this metric; it had not been launched for the first 10 days of its lifecycle.

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

[1] See https://github.com/ayaankazerouni/sensordata

Further discussion of our ability to accurately identify aspects of the development process, and their relationship to project outcomes may be found in the next section.

4.3 Quantitative Evaluation

Qualitative evaluation using student interviews and manual examination of Git snapshots showed the assessment model to be *mostly accurate* with room for improvement. To further investigate the measures, we conducted quantitative evaluation to determine which measures, if any, were significantly related to:

- **Project correctness**, measured as the percentage of instructor-written software tests passed.
- **Finishing solutions on time**, determined by the time of the student's final submission to Web-CAT and the assignment due date.
- **Time spent on the project**. 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 and preprocessing, the dataset consists of the work of 162 students working on 545 programming projects turned in across four assignments. To test for relationships with the outcome variables, we used a mixed model ANCOVA. 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. Results for all statistical tests use $\alpha = 0.05$ to determine significance unless otherwise noted.

Working Early and Often. The Early/Often index represents a mean edit time for code written for a project. Using a similar calculation, one can also calculate the median edit time.

With respect to *project correctness*, we used the percentage of instructor-written software tests that the student's final submission could pass as the dependent variable in the ANCOVA. We found that the solution mean edit times (Early/Often Index) were significantly related to project correctness ($F = 16.2,\ p < 0.0001$). In other words, students tended to produce more correct programs when they worked on their solutions earlier. The same ANCOVA indicated that the test edit median time was also significantly related to project correctness ($F = 0.06,\ p = 0.0018$).

With respect to *finish times* we used the number of hours before the deadline when the 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 times.

In summary, projects with better early/often scores (more specifically, solution edit mean times) tended to have more correct code, and were more likely to be finished on time. These results align with a separate study that measured procrastination and its effect on project performance, using a different data source, suggesting that the Early/Often Index is an accurate measure of procrastination [10].

Finally, with respect to *total time spent working*, we used the numbers of hours spent working as the dependent variable in the ANCOVA. The data suggests that the time of test-writing is significant, but the nature of this relationship is unclear. We found that the test edit time median ($F = 10.8,\ p = 0.001$) was significantly related to total time spent. Earlier edit times were associated with slightly longer time spent. Since there is no evidence for a relation between time spent and project correctness, this extra time does not appear to translate into improved grades. Instead, it may simply mean more time to work at a slower pace under less stressful conditions, and more time for reflection while working.

Test Writing. With Incremental Test Writing (Equation 4) as a continuous independent variable, the mixed model ANCOVA showed no evidence for a relationship with project correctness ($F = 2.45,\ p = 0.11$), finish time ($F = 0.17,\ p = 0.68$), or time spent ($F = 0.29,\ p = 0.59$).

Measures for the *balance* of test writing over time showed more explanatory power for project outcomes. These results are currently under review.

Data collected by DevEventTracker also lends itself to visual analysis and feedback. Consider Figures 1a and 1b. In Figure 1a, notice how most of the test code was written in the first work session, and then testing was sparse until the last few work sessions. This is presumably because *some* testing is required as part of the project specifications. However, this figure indicates that the student did not practice regular testing. This is contrast to Figure 1b, where the test code and solution code tend to co-evolve gracefully over time. These figures could help students introspectively consider their own programming practices.

5 PROPOSED WORK

Assessing Test-Suite Quality. Historically, code coverage has been a popular metric for test quality, but it has been shown to not be strongly correlated with test suite effectiveness [13]. Currently, we score students on the code coverage achieved by their test-suites, but this allows some bad practices to slip through the cracks. For example, we see projects whose tests violate the *single point of failure* requirement as described in [2] by invoking large swathes of functionality and making assertions about the generated output. A key idea here is that code coverage treats *invocation* and *verification* of code as the same thing.

I plan to characterize what the 'anti-tests' described above look like, so we can give students feedback about them. I hope to do this by statically analyzing the methods invoked in test methods. If a test invokes a 'large' amount of code based on some heuristic, I can characterize the test as being more of a functional test and having less of an effect. It is important to note that methods developed under this research question should ideally focus on assessing *test-suite as a whole*, as opposed to single tests. Functional tests are not inherently bad; they are only rendered less effective when unaccompanied by lower-level unit tests.

Closing the Loop. A stated goal of this research is to act on the measures I develop by giving students feedback on their software development practices in the form of adaptive interventions. This question investigates what the best way to give this feedback is. Ranging from least intrusive to most, some potential methods of doing this are:

- A **learning dashboard** that can be visited by students when they choose to. Such a dashboard could also contain class overview information for an instructor to see.
- **Regular emails** were relatively successful at curbing procrastination in previous work [21]. It may be that *adaptive* emails sent at critical moments might become more effective at changing other programming habits. An advantage of this method is that it would likely include the least mechanical development work. There would be little need for a user-facing interface to be developed or maintained, and some infrastructure for sending customised emails is already in place from work done in [21].
- We could also use a plugin to give feedback **directly in the IDE**. The disadvantage here is that students might find this invasive and/or distracting, and it would increase computational load on the student's machine.

Naturally, we want to do this in the way that effects the most positive change. Having collected log data from semesters without any interventions, we already have a control group. New methods for delivering feedback could be devised and administered, allowing us to design robust experiments to determine the most effective one. It could be that different groups of students respond differently to different interventions.

6 CONTRIBUTION

The most obvious contribution is to students taking programming courses. Regular interventions during project completion will keep students on track to complete assignments and follow best practices during project development. The students under study are junior-level students working on large and complex programming assignments. They are typically only two or three semesters removed from professional developers entering the workforce. The metrics developed in this research could easily be applied in an industrial setting. A primary vision of this research is to deploy an end-to-end pipeline that receives an incoming event stream and responds with timely and effective feedback to the developer.

REFERENCES

[1] 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. *ACM SIGCSE Bulletin* 34, 4 (2002), 125–128.

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

[3] 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. https://doi.org/10.1145/2538862.2538924

[4] 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*. ACM, 597–602.

[5] 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. https://doi.org/10.1145/3017680.3017785

[6] 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. https://doi.org/10.1145/2787622.2787710

[7] Chetan Desai, David Janzen, and Kyle Savage. 2008. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin* 40, 2 (2008), 97–101.

[8] Stephen H Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing (JERIC)* 3, 3 (2003), 1.

[9] 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

[10] 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*. ACM, 3–14.

[11] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring Problem Solving Paths in a Java Programming Course. In *Psychology of Programming Interest Group Conference, PPIG 2014*. 65–76. http://d-scholarship.pitt.edu/21832/

[12] 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. https://doi.org/10.1145/2858796.2858798

[13] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 435–445.

[14] 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. https://doi.org/10.1145/1151588.1151600

[15] 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 *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 136–144.

[16] Edward L Jones. 2000. Software testing in the computer science curriculum–a holistic approach. In *Proceedings of the Australasian conference on Computing education*. ACM, 153–157.

[17] 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

[18] 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*. ACM, 191–199.

[19] Karen Keefe, Judithe Sheard, and Martin Dick. 2006. Adopting XP practices for teaching object oriented programming. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. Australian Computer Society, Inc., 91–100.

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

[21] 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*. ACM, 3–11.

[22] Terry Shepard, Margaret Lamb, and Diane Kelly. 2001. More testing should be taught. *Commun. ACM* 44, 6 (2001), 103–108.

[23] 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*. ACM, 907–913.

[24] 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

[25] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2013. Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior. In *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies (ICALT '13)*. IEEE Computer Society, Washington, DC, USA, 319–323. https://doi.org/10.1109/ICALT.2013.99