DevEventTracker: Tracking Development Events to Assess Incremental Development and Procrastination

Ayaan M. Kazerouni, Stephen H. Edwards, T. Simin Hall, and Clifford A. Shaffer
Dept. of Computer Science, Virginia Tech, Blacksburg, VA 24061
ayaan|s.edwards|simin.hall|shaffer@vt.edu

ABSTRACT

Good project management practices are hard to teach, and hard for novices to learn. Procrastination and bad project management practice occur frequently, and may interfere with successfully completing major programming projects in mid-level programming courses. Students often see these as abstract concepts that do not need to be actively applied in practice. Changing student behavior requires changing how this material is taught, and more importantly, changing how learning and practice are assessed. To provide proper assessment, we need to collect detailed data about how each student conducts their project development as they work on solutions. We present DevEventTracker, a system that continuously collects data from the Eclipse IDE as students program, giving us in-depth insight into students' programming habits. We report on data collected using DevEventTracker over the course of four programming projects involving 370 students in five sections of a Data Structures and Algorithms course over two semesters. These data support a new measure for how well students apply "incremental development" practices. We present a detailed description of the system, our methodology, and an initial evaluation of our ability to accurately assess incremental development on the part of the students. The goal is to help students improve their programming habits, with an emphasis on incremental development and time management.

CCS Concepts

•**Social and professional topics** → Computer science education; •**Software and its engineering** → *Software creation and management;*

Keywords

Incremental development; procrastination; interactive development environment; educational data mining; project management practice

ITiCSE '17, July 03 - 05, 2017, Bologna, Italy
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4704-4/17/07...$15.00
DOI: http://dx.doi.org/10.1145/3059009.3059050

1. INTRODUCTION

Tools for automatic assessment of programming assignments enable students to gain more programming practice with less active grading effort for instructors [11]. These tools are able to automatically grade on metrics like code style and thoroughness of testing in addition to correctness [3, 9]. Many such systems are designed to support small-scale programming exercises. At that scale, there is little concern for the development process, and procrastination generally relates only to getting started with the assignment. Mid-level computer science courses often involve major programming projects, in which students are writing many hundreds or even thousands of lines of code, with life cycles measured in weeks. In this situation, support systems such as Web-CAT [3] can help with managing project submissions, evaluating code style, automated grading through unit testing, and assessment of artifacts such as student tests in terms of code coverage.

None of these aspects directly address a major concern that too many students are unable to complete programming projects at this scale. This often is a result of inadequate skill by the student in good project management techniques, including time management and fundamental development processes such as incremental development. Procrastination has proven to be a pervasive problem among students working toward project completion. Previous work has shown that students who start their projects early and practice good time management receive better grades than students who start late [4].

Unfortunately, to date there have not been tools to help instructors assess and evaluate student adherence to good development practice. This is in large part because the necessary data about the details of the student's development process have not been available. For example, incremental development with regular testing is a known best practice of software development [2]. Existing tools are generally unable to assess incremental development and time management as students work on their solutions. In this paper we present *DevEventTracker*, a system designed to collect fine-grained data about the student development process. With these data in hand, the next step is to analyze the data to detect procrastination and whether students employ good development practices like incremental development and effective testing procedures. If that could be done, then suitable interventions to encourage good practices could be devised.

Section 2 presents related work in data tracking and procrastination assessment systems and strategies. A detailed description of DevEventTracker's functionality is given in

Section 3. Sections 4, 5, and 6 present preliminary findings obtained from using the system over the course of two semesters, and the results of a series of student interviews.

2. RELATED WORK

Web-based Center for Automated Testing (Web-CAT) [3] is a web-based automated grading system that allows students to make multiple submissions to an assignment and receive immediate feedback. This feedback can be about correctness, code style, or code coverage by student-written tests. Web-CAT interacts with a custom Eclipse plugin that allows students to make submissions and download starter projects directly from within the IDE.

This model of multiple submissions affords us the ability to gather information about the student's development process, such as when a student started submitting it to get feedback and when they finished. It also provides an opportunity for analysis of the differences between submissions, giving a rough idea of a project's development trajectory. What it does *not* do is give us enough insight to assess whether or not students are practicing incremental development. To accurately assess this, we would need more granular data collected *during development*, rather than after submissions; in the latter state, we would invariably receive data about projects already in varied degrees of completion.

To this end, an addition was made to Web-CAT's Eclipse plugin, called the DevEventTracker Addition. The plugin continuously collects development event data as students program, giving us unique insight into the development process of the typical student. Benefits are twofold: 1) Our data are no longer limited by when a student decides to make a submission. 2) Since we are collecting data directly from the IDE, we have access to events that were not available through Web-CAT alone.

The Test My Code (TMC) plugin [10] for NetBeans behaves in a similar fashion to our Eclipse plugin. It records events whenever the student saves, runs, or tests code using instructor-provided tests. Hosseini, et al [5] make use of this plugin in an attempt to achieve goals similar to ours, but with some differences in the type of data collected. For example, in terms of detecting student testing, the TMC plugin collects data on runs of pre-written tests provided by the instructor, while DevEventTracker collects data about students writing and running their own tests. This provides information about the student's autonomous software development habits, which is ultimately what we wish to assess.

Hackystat [6] is an open-source project from the University of Hawaii that provides product and process measurements in software engineering situations in education and industry. DevEventTracker builds upon Hackystat, using Hackystat's client-side protocols and preexisting sensors in conjunction with our own extensions to send data to the server. Unlike Hackystat, DevEventTracker also collects event data for program and test launches from within the IDE. These additional events provide valuable information for assessing development patterns.

Marmoset [9] is an automated grading system developed at the University of Maryland. It uses an Eclipse plugin to collect student code and store it in a Concurrent Versioning System (CVS) repository each time a file is saved. Marmoset inspired us to capture repositories of student code, since event data alone might not capture enough of a student's work-flow to properly assess incremental development.

Researchers at the University of West Georgia [1] mined student revision histories from the Mercurial version control system to assess the state of incremental development in a CS2 course. Incremental development was defined in terms of the size and scope of commits, including checking for commits related to integrated test-writing. In answer to the question *Are students properly incorporating testing as part of their iterative development?*, the researchers found that there was room for improvement.

A previous study [4] collected five years of data from the first three programming courses at Virginia Tech. Assignment results were partitioned into two groups: scores above 80% (A/B), and scores below 80% (C/D/F). Analysis yielded important results. When students received A/B scores, they started earlier and finished earlier than when the same students received C/D/F scores. After normalizing for program length, there was no significant difference in the amount of time spent on each project stemming from starting earlier vs. later. This study provided convincing evidence that procrastination was correlated with lower project scores.

These findings led to another study [8] that administered three different types of interventions to prevent procrastination. That study found that of the three interventions (short reflection essays after each project, a requirement to set and track scheduling information and progress throughout the assignment, and e-mail alerts regarding progress toward completion), only e-mail alerts were associated with significantly reduced rates of late program submissions and significantly increased rates of early program submissions. The promise shown by this method was credited to the fact that the emails were relevant to individual students, generated using data from that student's latest submission to Web-CAT. However, we note that the emails were quite non-specific as to how much progress the student had actually made. More detailed feedback about progress and class standing could potentially have a greater impact.

We believe that interventions that provide students with feedback about their own programming practices should encourage the practice of incremental development and self-checking behavior. When a student makes a submission to Web-CAT, Web-CAT is able to give the student a percentage of code covered by their test cases, but is unable to tell with accuracy *when* the testing takes place: students could be practicing regular testing and development like they are supposed to, or they could be churning out tests right before the deadline. The reality is probably somewhere in the middle, and a rough idea can be gleaned from looking at the differences between submissions. Unfortunately, the information is not granular enough to accurately assess incremental development, especially since some students submit frequently to Web-CAT, while others prefer to do a lot of work between submissions. Some students make their first Web-CAT submission early in the development cycle with the goal of initially passing only a few tests, while others wait until much later in the development cycle to make their first submission. While early submission might seem to be more indicative of incremental development, we can't know that those whose initial submissions are made relatively late are not doing a lot of incremental development and testing without feedback from Web-CAT. We should ideally generate feedback from data that is generated directly from the student's edit-and-run process within their IDE.

3. THE *DEV EVENT TRACKER*

In this section we present a detailed description of the DevEventTracker subsystem. We use a custom Eclipse plugin to allow students to make submissions to Web-CAT and to download starter projects provided by the instructor. An addition was made to the plugin that allows the continuous collection of data from the IDE, not limited by when a student decides to make a submission. This continuous stream of data will provide instructors and researchers with a real-time understanding of a typical student's programming process. We collect timestamped development events as well as Git snapshots of the project as the student develops it. The development events capture a number of activities from within the student's IDE, and are used in the development of automatic assessment. The Git snapshots are used primarily for verification and evaluation of these assessments.

Edit Events: DevEventTracker collects Edit events in real time as a student programs. An Edit event is recorded each time a student saves their work. For each event, some meta-data is included. We know the size of the edit in statements or methods added or removed, and we know if the edit was within *solution code* or *test code.* Analysis of an ordered sequence of edits containing this information yields an understanding of how and when a student approaches writing tests for their program. These time-stamped events also provide insight into a student's procrastination habits, using a method described in Section 5.

Launch Events: Often, especially in the early stages of a project and for small changes, testing mainly consists of launching a program and examining its behavior. DevEventTracker monitors launches within Eclipse, collecting and recording meta-data about each launch. It records the type of the launch (execution of test cases vs. a regular interactive execution of the program); whether the launch terminated normally or with an error code; and for unit test runs, how many test case successes, failures, and errors resulted. This functionality does not limit our knowledge of a student's testing behavior to when they create *new* tests; it also records when they are running *existing* tests.

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

Other events: When students encounter build errors, we receive events containing the error message and information about the Java file that caused it. Data such as this can be used to analyze the types of problems most commonly encountered by students with different levels of programming expertise.

DevEventTracker also saves data about a student's use of the Eclipse debugger. It records when breakpoints are added or removed, when a debug session is started, and a student's actions during that session (*step into*, *step over*, etc.). The plugin collects data about code-refactoring activities within Eclipse. It records renaming and moving activities, with the plugin collecting information about old and new names and locations of units as they are refactored. Refactoring data do not star in the initial analysis we present in this paper, but they open up avenues for further research that depends on this type of analysis.

Git snapshots: Every student's project has a git repository associated with it, with the remote repository residing on the Web-CAT server. Whenever a student makes a change and saves a file, a Git snapshot is captured and sent to the server. This provides us with the ability to make further fine-grained observations about the changes to a project over time. More importantly, it allows us to evaluate the assessments made using development events.

4. METHOD

DevEventTracker was used by students in five sections of a post-CS2, junior level Data Structures and Algorithms course over two semesters. On the first day of class, we collected informed consent from the students. Students who did not give consent (less than 4% of the total) were excluded from data analysis. Once the number of students enrolled had stabilized, this made for a total of 370 students, generating data each time they worked on their projects. The course had four assigned projects, and data were collected for each one.

The class syllabus required students to program all of their projects in the Eclipse IDE, using the Web-CAT submission plug-in to submit assignments and download starter code. This setup has been the standard for many programming courses at our university for several years. The only difference for this project is that the plugin was augmented with data-collection functionality. The act of downloading starter code or making a submission creates a link between Web-CAT and a specific project in a student's Eclipse workspace, and this allows Web-CAT to begin receiving data for that project. To ensure that we received data from the moment work began on a project, we provided starter code for each project. In other words, we did not wait until the first submission to begin receiving data. The starter files provided for each project did not contain any stubbed out code except a `main()` method (with the correct file and class names) that printed the string 'Hello world!' and a test method invoking it. This is because we wanted unadulterated information about how students approach assigned projects, from start to finish. An added benefit of this approach was that students did not have to worry about the semantics of structuring and naming the project so that Web-CAT would accept it; that was already provided via the starter project. While we attempted to minimize the impact our data collection had on the students' programming experience, students were aware that it was taking place, and they sometimes experienced some delays due to data transmission.

If a connection to the Web-CAT server was unavailable, event data was logged locally until a connection became available. This ensured that we did not miss out on event data generated when students programmed without an internet connection, or if the Web-CAT server went down for a period of time.

5. ASSESSMENT MODEL

Our modeling process involved taking in a large volume of data for each student's project and reducing it to a vector of four metrics that we designed to cover the various dimensions of incremental development.[1] We focused on having each metric represent an item that—when presented to the

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

student—provides a concrete course of action aimed at improving their programming practice.

At this stage of our project, each metric is intentionally kept separate from the project grade. Students were not informed about any of these results (except for the students interviewed as described in Section 6.1), since at this stage we were only trying to judge our ability to recognize the level of incremental development. Note that the metrics as described below are *raw indices*, meaning that lower numbers might be better for some of them.

Early/Often Index: A measure of how early and how often a student works on a project, defined in relation to the due date for the project. We add up each edit's size in statements, with each size being weighted by the number of days until the project deadline. Then we divide this total by the total edit size. If E is the set of all edits, then the early/often index is defined as:

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

This produces an average that is weighted by how many days from the deadline—and how often—a student tends to work on their project. Therefore, if a student tends to work several weeks or days before the deadline, this metric will have a larger value; and if a student tends to procrastinate until the project deadline is close, this metric will have a smaller value (or possibly negative, depending on course policies). This metric is used as a quantitative assessment of procrastination, and **a larger value is better**.

Incremental Checking: A measure of how well a student self-checks their code by launching it. Here, 'launches' are defined as either regular program executions or test executions. For many students, simple launching with diagnostic print statements is a valuable method of testing and debugging. To focus on unit test launches only would be to ignore a common testing strategy for many. We add up each edit's size, with each size being weighted by the number of hours until the next project launch. Then we divide this total by the total number of edits:

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

This gives a value governed by the amount of the code a student writes and the time that passes before they next launch their code. For this metric, **smaller values are better**.

Incremental Test Checking: A measure of how well a student self-checks their code using automated tests. Unlike the previous metric, here we focus only on test executions. The metric is calculated in a similar way: we add up each edit's size, with each size being weighted by the number of hours until the next test launch. Then we divide this total by the number of edits:

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

The main benefit from this metric over the previous one is that it gives us an indication of whether the student is practicing progressive regression testing [7] or not. It also gives an indication of the role that formalized testing plays in the student's development process. For these first two metrics, if a student tends to write a lot of code before checking that it works, they would have larger values, and vice-versa, so **smaller values are better**.

Incremental Test Writing: A measure of how regularly a student writes tests. To successfully practice incremental development, students should regularly write unit tests to verify the correctness of the functionality they have recently implemented. We calculate Early/Often indices separately for *solution code* and *test code*. Then we find the difference between these two metrics. The result is a metric 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. Let SE be the set of all solution edits, and let TE be the set of all test edits. Then we calculate this as:

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

Therefore, if a student writes a lot of code before writing tests for it, this metric would have a larger value. Similarly, if a student writes test code a long time after writing solution code, a larger value would be produced; for example, students who do their testing at the end of the project life cycle will receive a higher value for this metric. Therefore, **smaller values are better**.

It is important to note that this score is not related to procrastination. A student can do the entire the project on the last day and still receive a good score for this metric; it depends on when solution code and test code were written in relation *to each other*, rather than in relation to the deadline.

6. EVALUATION

With the data collection process in hand, our main concern now is whether we can accurately determine if the student is using good time management practice, and if the student is using incremental development practice. Assessing these are difficult. The four measures proposed in the previous section are models, and they might or might not, singly or in combination, provide reliable results. A primary concern is that there is no readily available 'ground truth' against which we can test our metrics. Validating our models against grades could potentially lead to inaccuracies. It is possible for students with good grades to follow poor incremental development, and vice-versa.

6.1 Interviews

In order to evaluate the validity of our models, we decided to use individual interviews to gather student opinions. As we neared the end of the 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 written to calculate the metrics described in Section 5. Ten students representing a range of scores on the different metrics were selected and invited to participate in interview sessions. Of those ten, seven agreed to participate.

The students were interviewed in depth about their programming practices. Specifically, we asked about their testing habits: How often did they write/run tests on the specified projects? What is their preferred method of testing? What do they think of Web-CAT's testing requirements? The interviewers were not involved in grading students in any way, to avoid the possibility of students thinking that their answers would somehow affect their grade. The students were shown our model's assessment of their programming practice, and were asked what they thought about its accuracy, usefulness, and potential efficacy in helping them change their programming habits in the future.

Accuracy: Six of the seven students found our assessment accurate. The descriptions that follow use feminine pronouns, regardless of the gender of the participant.

- **Interviewee 1** stated that she found the model's assessment to be accurate.
- **Interviewee 2** mentioned that she had been ill and started Project 1 late and worked past the deadline. When the assessment was revealed, we saw that our model had been able to detect this and had given her a low Early/Often score. When the interviewee saw the scores, she agreed with the overall assessment.
- **Interviewee 3** acknowledged that she and her partner had gotten a late start on Project 1, but that she had worked alone on Project 2 and started relatively earlier. Our model was able to detect this—the interviewee was given a low Early/Often score for one project, and a higher score for the next.
  The student also mentioned that she "didn't write the best tests during the beginning of [Project 1]"; she relied mostly on simple diagnostic print statements for testing and "wrote tests at the end". This is in contrast to Project 2, where she "[brought] in formal testing", since she now had some experience with it. The model's assessment recognized this difference—the student received a poor score for Incremental Test Writing on the first project, but a much better score for the second project.
- **Interviewee 4** received a much lower score for Incremental Test Writing on Project 2 than she did on Project 1. Project 2 was almost universally cited as the hardest project that the students worked on this semester (at the time of the focus group, they were starting work on Project 4). The student mentioned that, because the project was so hard, she found herself getting caught up in trying to implement it correctly and ended up writing "more code before testing" than she did on Project 1. This was reflected in our assessment. Also seen was a lower score for Incremental Test Checking, which intuitively makes sense—if she was not writing tests until the end, she was not running them, either. An interesting thing to note here is that, on a hard project where testing would be most useful, the student brushed it aside in favor of going straight ahead with the implementation.
- **Interviewee 5** thought the model was *mostly* accurate. Her answers to questions about writing tests did not agree with her failing score for Incremental Test Writing on Project 2. After initially expressing surprise, she backtracked on what she had said earlier by saying that the project involved a lot of recursion, and she tends to test recursive algorithms using iterative print statements rather than formalized testing strategies. This was the only occurrence of a student volunteering new information to explain a score provided by our metric.
- **Interviewee 6** was the only student who did not find the metrics accurate. The interview revealed disconnects between our assessment and the student's description of her programming practice. However, further investigation revealed transient issues with this student's data reaching the server, which would lead to inaccuracies during metric calculation.
- **Interviewee 7** received high scores on all metrics, except Incremental Test Writing for Project 2. She expressed surprise at her low score for this. The remaining scores were in keeping with her description of her programming practices.

Usefulness and future efficacy: Only one of the seven students did not see value in the model. This was the student who had received an inaccurate assessment due to data transmission errors. Of the remaining six students, five found the model to be useful and stated unconditionally that they would try to change their programming practice if they were given this feedback between projects. The final student said that while the model could be useful to "somebody coming into their best practices", it was not particularly useful to her since she already knows about and follows incremental development. This student received high scores on nearly all metrics.

Overall, the model was mostly accurate with students generally finding the information interesting and useful. The interviews confirmed the model's ability to detect differences in programming practices *between students* as well as *within students and between projects*. The model also has benefits in that it is clear what kinds of actions students should pursue to increase their scores on any of the metrics.

6.2 Git Snapshots

While our focus group provided student validation of the measures, we also wanted to directly investigate the edits students were performing in their projects. A second type of evaluation was carried out by manually inspecting the Git repositories maintained by DevEventTracker. Twelve projects were randomly sampled from the pool of submissions. The inspection focused on checking whether our assessment of incremental development matched the "actual programming process" of the student (as seen in the Git revision histories). Eight of the twelve projects had low scores (< 80 on a normalized 100-point scale) for **working early and often**. Stepping through commit histories showed that seven of these projects had multiple breaks of several days where no work was done, leading to the project being completed within the last few days before the due date. The remaining project with a low Early/Often score was worked on the day before the project deadline, in a marathon session taking up most of the day. One out of the remaining four projects received a surprisingly high Early/Often score, since the project was started within the last two days. However, it was worked on steadily without breaks, possibly contributing to its high Early/Often score.

Incremental checking and **incremental test checking** were evaluated 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. Most projects received middling or good scores (> 80 on a normalized 100-point scale) for these metrics, but one project received a low score. This project was not launched for the first 10 days in its life-cycle, and launches took place after large amounts of code were written.

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 ($\geq$ 90) for this metric. Their commit histories showed that testing began on the first day of work and continued consistently until the end of the project. Also clear was the fact that test classes were usually created and edited within a few minutes of their corresponding solution class.

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

7. CONCLUSIONS AND FUTURE WORK

In this study, we analyzed data for 370 students working on four large projects in a Data Structures and Algorithms course over the course of two semesters. We collected detailed development event data using custom event-tracking software integrated into an Eclipse plugin. In order to accurately assess abstract concepts like incremental development and procrastination, we developed a set of four metrics we believe cover the different dimensions of both concepts. Development of these easy-to-understand metrics allowed us to gather feedback from students on the accuracy of our model.

Six out of seven students interviewed found the model accurate and useful, stating that the feedback provided might encourage them to change their programming behavior from one project to the next. Further, a separate manual investigation of code snapshots in student projects confirmed the measures track intended behaviors. Together, these represent encouraging results in an effort to turn qualitative ideas that are difficult to assess into quantitative measures that can be systematically and repeatedly applied.

An important benefit of DevEventTracker is that it affords us the ability to conduct further work toward answering questions about student programming behaviors. Such questions include:

- How does incremental development (as measured by our metrics) affect project grades?
- How does providing students with their incremental development scores affect their programming practice from one project to the next?
- Are there meaningful patterns in the ways partners work together on projects?
- Are there meaningful patterns of behavior related to early vs. late Web-CAT submission practices?

8. ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation under grant DUE-1245334. 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.

9. REFERENCES

[1] L. Baumstark, Jr. and M. Orsega. Quantifying introductory cs students' iterative software process by mining version control system repositories. *J. Comput. Sci. Coll.*, 31(6):97–104, June 2016.

[2] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. *The Agile Manifesto*, 2001.

[3] S. H. Edwards and M. A. Perez-Quinones. Web-cat: Automatically grading programming assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, pages 328–328, 2008.

[4] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 3–14, 2009.

[5] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a java programming course. In *Psychology of Programming Interest Group Conference, PPIG 2014*, pages 65–76, 2014.

[6] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering, ISESE'04*, pages 136–144, 2004.

[7] H. K. N. Leung and L. White. Insights into regression testing [software testing]. In *Proceedings. Conference on Software Maintenance - 1989*, pages 60–69, Oct 1989.

[8] J. Martin, S. H. Edwards, and C. A. Shaffer. The effects of procrastination interventions on programming project success. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 3–11, 2015.

[9] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with Marmoset: an automated programming project snapshot and testing system. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5, 2005.

[10] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122, 2013.

[11] C. Wilcox. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 90–95, 2015.