Analyzing Student Coding Practices using Fine-grained Edits

Dr. Clinton Andrew Staley, California Polytechnic State University

Dr Staley is a professor of Computer Science at California Polytechnic State University in San Luis Obispo, CA. His research interest is in building novel tools for instruction, particularly of Computer Science.

Mr. Corey Ford, California Polytechnic State University

Corey Ford is pursuing a blended B.S. and M.S. in Computer Science at California Polytechnic State University, San Luis Obispo. His research interests include distributed systems and social software. He is a Junior Software Engineer at Software Inventions.

Analyzing Student Coding Practices Using Fine-grained Edits

Abstract

In this paper, we gather data from three groups of students doing three different assigned programming labs. For this, we use an online IDE for introductory programming that records student code editing, compiling, and testing activities down to the individual keystroke. The IDE also gathers periodic student feedback on frustration levels during the coding process.

We report on patterns of student work, including working sessions, total time spent, how far ahead of deadline students start, and time of day worked. We compare work patterns between students who completed the assignments on time, and those who did not. We also compare statistics such as recent numbers of good and bad test runs and editing activity against reported student frustration levels. Finally, we review a sample of student compile errors in two different C language projects, one by beginning programmers, and the other by upperdivision programmers, and report the types of errors made in each group.

We find several interesting results from these comparisons: students often work in short stints, they work fewer late hours than might be expected, and early starts on a project, while useful, are not as critical to success as might be expected. We also find that patterns of compile errors and bugs do not correlate closely to student frustration—different students have different emotional responses to the same situations. And we find that even among advanced students, compile errors are skewed toward simple mistakes, and the majority of errors are of just a few different types.

Introduction

Interesting information regarding the learning experience of introductory Computer Science students can be obtained by analyzing their coding activity at a fine-grained level, In this paper, we report results from analyses of student coding patterns using an online IDE for introductory programming that collects keystroke-level information.

The ultimate goal of our research is to improve online IDEs of this sort by adding means of detecting student difficulties, improving compile and runtime error reporting, and identifying successful patterns of code development.

Prior Work

The behavior of student programmers has been the subject of substantial research. An early instance of such work[7] compared successive program submissions in a batch-processing environment, finding that most changes affected only one or two lines of the source. A later study[9] observed and tracked high-level behaviors of high school students in a Pascal programming class, and noted that the students spent most of their time editing and running their program rather than planning or reformulating code.

Most recent work extracts detailed data directly from program editors, and is usually considered to fall under the umbrella of *learning analytics* or *educational data mining*. Berland[1] analyzed

data collected in the IPRO mobile programming environment to describe general programming processes. Using the BlueJ programming environment, Jadud[6] recorded snapshots at each compilation and analyzed students' process towards creating syntactically correct programs, finding a correlation between assignment grades and a measure of syntax-error struggles. Helminen et al.[4, 5] studied behavior in solving Parsons problems. One of the most fine-grained analyses was performed by Blikstein[2], who analyzed students' development strategies through code snapshots and event logs. Blikstein and coworkers[3, 8] have also used machine learning techniques to understand student pathways to completing a program.

Our work adds the elements of requesting live student feedback regarding their level of frustration during the development process, and an ability to play student work back in time-lapse form, keystroke by keystroke, at any point in the development process.

Methods

Methods: LearningIDE Tool

We studied student programming assignments conducted using LearningIDE (www.LearningIDE.com), a web-based integrated development environment (IDE). This IDE, whose user interface is shown in figure 1, provides typical facilities for editing a set of source code files, compiling these to executable form, and interactively executing the result or automatically running it against test cases. All project state is stored, and all code compilation and execution is performed, on the LearningIDE server. The server keeps detailed records of individual text edits and document scrolls, compilation events, and execution events, allowing for realtime or time-lapse playback of a student's work, as well as automated analysis.

For this study, we extended the user interface of LearningIDE to include a small panel (upper left) for collecting self-reported, qualitative feedback on students' development processes.

In comparison to tools used in prior work, notable features of LearningIDE include recording at the keystroke level rather than the file-save or compile level, the ability to visually check student work in time-lapse playback, and the use of a web-based user interface with a centralized database for easy collection of this data.

Methods: Projects Studied

We arranged for three different groups of students to use LearningIDE for programming assignments. The first was a section of ACM CS1 students, coding a Python project. The second was two sections of an introductory C programming course intended for engineering students, and the third was an upperdivision C systems programming class, from which one large section's worth of students participated. All programs were reasonably small exercises that might be completed in one or two hours.

Methods: Data Studied

Once student work was done, we cleaned the data, dropping students who had not agreed to participate in the study, removing minor "test runs" that were clearly not full efforts, and discluding from the frustration-level studies certain students (see Frustration Level Analysis section).

After this, we had 145 participating students, who collectively performed about 190 hours of work using the IDE, in 1053 separate working sessions, averaging approximately 10.7 minutes each. Deciding what constitutes a "working session" based on student activity is a judgment call —we assumed that a student who had shown no activity (not even scrolling, which the IDE records) for 5 minutes had ended a working session. This working time included 5800 attempts to compile, of which about 4200 were successful and 1600 had errors. It also included 5700 tests of a program, either hand-runs or automated test runs.

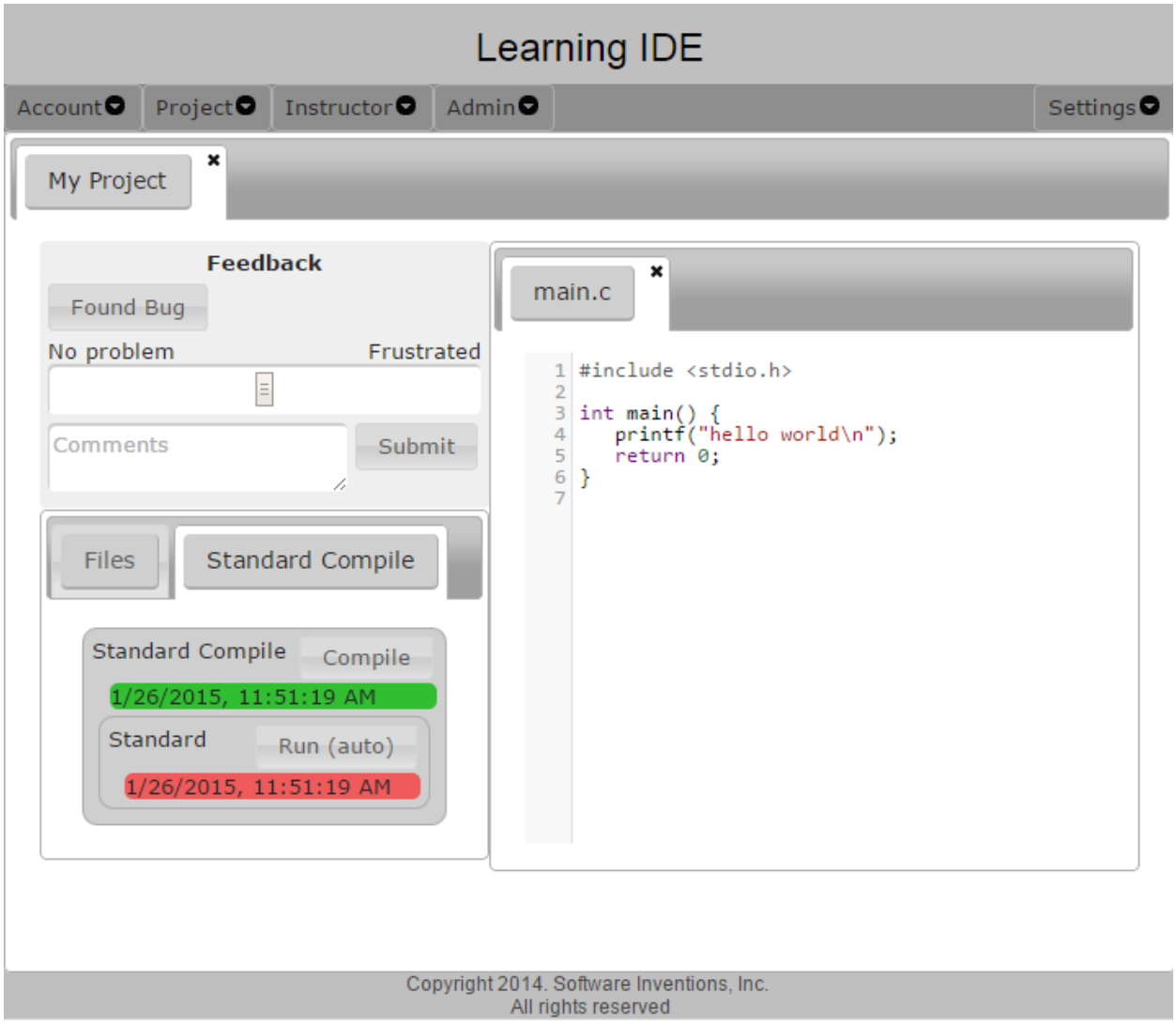

Figure 1: Screenshot of Learning IDE

Results

Results: Working Behavior

We ran several statistical studies on students' work patterns, to determine whether students worked in long or short stints, whether they started early or near the deadline, and what times of day they typically worked. We were also interested in whether student success, as measured by completion of the projects by deadline (as occurred for 82% of projects, excluding entirely projects from one of the groups of students for which we had no measure of completion), correlated with any of these.

Even for these relatively short projects, students put in remarkably short working stints, as shown below (mean ± standard deviation, all time units in minutes). Playback of a few samples of very short sessions showed that some comprised only one brief edit.

Student Success	Session Length	Min	Max	Number of Sessions
Successful	9.3 ± 15.0	0	101	576
Unsuccessful	7.0 ± 14.3	0	75	100

The average session length of unsuccessful projects was modestly shorter than that of successful ones. Taking an average and total session length for each project, and aggregating these per-project statistics, shows the following (mean ± standard deviation, all time units in minutes):

Student Success	Session Length	Total Length
Successful	10.9 ± 7.0	59.5 ± 32.0
Unsuccessful	6.3 ± 5.3	35.2 ± 34.6

We had expected to find a number of late night working sessions, and to compare relative success rates of projects completed in daylight vs those done in the wee morning hours. The reputation of CS students notwithstanding, we found that there were so few late working sessions that no statistical analysis was worthwhile. In all, only 8 students in the study put in any work between midnight and 6am. Three night owls were at work steadily between 3a and 6a, but the others worked only until 1a or so. One student, somewhat mysteriously, put in just a single edit at 1a, evidently a quick thought before bed. Further statistical study would of course be needed, but our modest data suggest that perhaps the sleepless reputation of students in coding classes is exaggerated.

We were interested in whether early starts on a project correlated with success, so we computed, for each student, the average distance between starting time of each working session and the project deadline, weighted by length of working session, dubbing this "lead time". We also wanted to know how much working time students put in before doing their first compile (dubbed Time To Compile) and before doing their first run (dubbed Time To Run). For successful and unsuccessful student work, these results are (mean ± standard deviation):

Student Success	Lead Time (hours)	Time To Compile (min)	Time To Run (min)
Successful	56.3 ± 37.2	8.8 ± 9.7	9.7 ± 12.3
Unsuccessful	46.8 ± 38.4	14.0 ± 16.7	20.7 ± 23.7

These data support the usual teaching advice to start early on assignments and to practice incremental development, though it's worth noting that the difference is well under a standard deviation, and four students succeeded with a lead time of under 3 hours—one student with a lead time of 100 minutes. Starting early is not necessarily essential, if these data are to be believed.

Results: Frustration Level Analysis

One of our research goals was to measure student frustration levels during the programming process, and explore ways of automatically identifying "pain points" where the help of a TA or instructor might be of value. Importantly, our intent was to measure not merely student difficulty, as would be reasonably evident, e.g. from a series of failed tests, or repeated attempts to resolve the same compile error. We were interested in the students' affective experience, since a student who is challenged *and* frustrated is the one in most need of help.

To this end, we augmented the LearningIDE interface with a frustration-level slider, and a text field for free comments (see earlier screenshot). We explained the use of this to the participating students, and urged them to adjust the slider periodically to indicate how they were feeling about the work. We also programmed the slider to pulse red every five minutes as a reminder to do this. Students were generally willing and helpful about offering this response, giving us over 1600 individual data points, around 11 per student, by adjusting the slider.

We removed any student with average frustration level below 20% (against a frustration-level scale of 0-100%) and with no response above 50% (the initial default value), because such students may have lowered the frustration threshold initially, and then just perfunctorily tweaked it to silence the red pulse reminder. We also dropped students who offered no feedback at all, and those whose feedback had a standard deviation of under 10%, for the same reason. This dropped 33 "low responders", leaving a pool of 112 responding students.

We then identified the most dramatic indications of frustration—those where the slider was adjusted upward by at least 40% in a single input. In all, these comprised 107 "frustration points" of special interest. We performed a visual inspection on approximately half of these, spending several minutes using LearningIDE to play back the student's editing, compiling, and execution activities surrounding the frustration point.

Various patterns seemed evident, some expected, others perhaps less so. Frustration points were often preceded by 5 or more compilations, or more often, execution failures, within the past 5 minutes. But they were also often surrounded, both before and after, by a period of scrolling up and down, or simple inactivity, perhaps indicating student puzzlement over the code. We had thought that a combination of repeated, recent, failures, with only minor edits (e.g. to insert a

print statement or to try a fix) between them would indicate student frustration, but the visual analysis did not often show this pattern.

Based on our informal inspection, we then performed a number of small statistical studies. In each, we computed some statistic combining such factors as the number of failed recent compilation attempts, the number of failed recent compilation attempts with no progress on errors (similar to a previously proposed metric[6]), the number of recent execution attempts, the number of failed recent execution attempts, or the number of editing actions both before and after a given point. For each, we computed an average and standard deviation value for the statistic across a large randomly-selected set of points from the entire body of working sessions, and for our identified frustration points (with a level above 70%).

Only one or two showed even a slightly useful "signal". Periods of baffled-looking light editing, for instance, proved to be about as common when students weren't frustrated as when they were. The same went for blocks of unsuccessful execution or compilation attempts. Averages for these statistics were almost always modestly higher for the frustration points, to be sure, but never more so than a fraction of a standard deviation. Some examples follow, giving (mean ± standard deviation):

Statistic	High Frustration	Random Time
# run attempts, prior 5 min	2.84 ± 4.05	1.86 ± 2.91
# failed run attempts, prior 5 min	1.90 ± 3.09	0.95 ± 1.98
% failed run attempts, prior 5 min	66.7% ± 39.4%	49.1% ± 42.5%
(# run attempts + 2(# failed run attempts)), prior 5 min	6.65 ± 9.98	3.75 ± 6.55
# text edits, prior 5 min	33.4 ± 38.3	42.4 ± 41.7
# text edits, following 5 min	29.3 ± 36.3	41.7 ± 41.8

After viewing scatter plot after scatter plot that resembled white noise with perhaps a vague central line, we decided that the best statistical proxy we had seen for frustration was a weighted combination of the number of recent execution attempts, and the number of recent failed execution attempts, with the latter weighed twice as much as the former. We adjusted this statistic to a sensitivity threshold that would reject 90% of the random points, leaving only 10% false positives. Under these conditions, 75% of the frustration points were rejected, leaving only 25% correct positives. Clearly not a useful result.

So, even with a prefiltering process on low-responding students that could arguably be said to spin for better results in automatically identifying student frustration, we did not discover a good statistic for doing so.

We emphasize that this was one small study, and we hope that others may try similar work and find better outcomes. But, we do offer two pieces of advice drawn from this experience. First, it is our informal impression, and one that agrees with most teaching experience, that student frustration level is very individual, even given the same intellectual challenge. One student's infuriating blind alley is another's fascinating puzzle, and this works against deriving a general

statistic for student frustration. In future work, we plan to look at automated identification of frustration points using statistics and patterns customized per student.

Second, to that end, an easy and even attractive way for students to indicate frustration in the UI is essential. A periodically pulsing slider is workable only for a short study with cooperative students. Our future work will incorporate a better UI to allow long-term gathering of frustration data in sufficient volume for statistics to be customized per student (e.g a big red "Dammit!" button is likely to get more attention than an awkward slider.)

Results: Analysis and Improvement of Compilation Errors

LearningIDE tracks every compilation error each student encounters. We are interested in improving compiler error messages for beginning and more advanced students, so we conducted a survey of errors, for C code, encountered by the beginning C engineering students and the upper division C systems programming students. We used only first errors per compile, to avoid the effect of "ripple errors".

65% of beginning student errors were one of missing variable declaration, missing semicolon, and errors in user-declared function prototype. 15% were also pretty routine: expression syntax, missing brace/paren, etc. 20% were rarer, e.g. missing opening paren (*scanf"xx"*)), missing a variable name in a declaration (*double = 2.3;*), using an int as a pointer (*myexp(*x)*), etc.

Compiler error messages (LearningIDE uses GCC as the serverside compiler) ranged, predictably, from acceptable to utterly baffling to a beginner. Missing semicolon messages of course fell on the line after the error, never on it. Missing closing braces result in *Expected declaration or statement at end of input*. Even missed variable declaration errors (*x undefined, first use this function*) could have been better.

One would hope that upper division students would have learned where to put semicolons, but even their errors, while much less numerous than the beginners', were 50% missing variables or semicolons. Rather more of their errors had to do with brace and parentheses matching, missing subexpressions, and use of rvalues to the left of assignments. These collectively comprised 30% of errors. This may have been partly due to the fact that their assignment involved complex expressions, however.

Remarkably, a very large majority of error messages that students typically encounter could be made friendlier with modest regular-expression based postprocessing of the compiler's output, either rephrasing or augmenting it. Some examples:

GCC error message	Friendlier error message
<varname> undefined, first use this function	You forgot to declare <varname>
Expected declaration or statement at end of input	Did you miss a closing bracket?
Line <x> semicolon expected after <whatever>	Missing semicolon near line <x-1>
lvalue required before assignment operator	You are trying to assign into an expression

LearningIDE already filters compiler output, and we will consider adding such error message rewrite rules as future work. The next version of the LearningIDE will also change from gcc to clang, which offers superior error messages.

Further Work

Planned further work includes, as indicated above, incorporation of better UI for student frustration feedback, efforts to customize automated frustration identification per student, incorporation of more programming languages and a simple debugging interface into the LearningIDE, and possibly a study on the impact of improved compile error messages. We also hope to use LearningIDE in a wider range of classes and possibly institutions to gather a broader statistical base of student performance.

Summary

We used an online, keystroke-recording, IDE to study various aspects of student programming behavior and frustration levels in several different small programming projects, with varying levels of students.

We discovered some interesting data regarding work patterns, including the fact that students often work in short stints, and that fewer work late hours than might be expected. We also found a modest correlation between longer stints and student success. And we found that while starting a project early is useful, it's not as closely correlated with success as might be expected.

We gathered direct data on student frustration points, and attempted to find statistics that would automatically identify such points, but finding a general such statistic was difficult, probably due to varying student affective responses to programming challenges. We suggest that it may be more constructive to customize such statistics per-student, and to present an easy and attractive UI for students to express their frustration levels.

We reviewed typical compilation errors for beginning and experienced students, and found that a relatively small set accounted for a large majority of errors. The number of errors dropped significantly for more advanced students, but the distribution of errors remained skewed toward simple mistakes, with somewhat more errors in expression and statement structure due to more complex code. We also found that almost all of the most common error messages could be improved by simple regular-expression style substitutions.

Bibliography

1. Berland, M., & Martin, T. (2011). Clusters and patterns of novice programmers. In The meeting of the American Educational Research Association. New Orleans, LA.
2. Blikstein, P. (2011, February). Using learning analytics to assess students' behavior in open-ended programming tasks. In Proceedings of the 1st international conference on learning analytics and knowledge (pp. 110-116). ACM.
3. Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. Journal of the Learning Sciences, 23(4), 561-599.
4. Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012, September). 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 (pp. 119-126). ACM.
5. Helminen, J., Ihantola, P., Karavirta, V., & Alaoutinen, S. (2013). How do students solve parsons programming problems?--execution-based vs. line-based feedback. Learning and Teaching in Computing and Engineering (LaTiCE), 2013, 55-61.
6. Jadud, M. C. (2006, September). Methods and tools for exploring novice compilation behaviour. In Proceedings of the second international workshop on Computing education research (pp. 73-84). ACM.
7. Nagy, G., & Pennebaker, M. C. (1974). A step toward automatic analysis of student programming errors in a batch environment. International Journal of Man-Machine Studies, 6(5), 563-578.
8. Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012, February). Modeling how students learn to program. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 153-160). ACM.
9. Pintrich, P. R., Berger, C. F., & Stemmer, P. M. (1987). Students' programming behavior in a Pascal course. Journal of Research in Science Teaching, 24(5), 451-466.