Evidence for Programming Strategies in University Coding Exercises

Kshitij Sharma(✉), Katerina Mangaroska, Halvard Trætteberg, Serena Lee-Cultura, and Michail Giannakos

Norwegian University of Science and Technology, Trondheim, Norway
{kshitij.sharma,katerina.mangaroska,hal,serena.leecultura, michailg}@ntnu.no

Abstract. Success in coding exercises is deeply related to the strategy employed by the students to solve coding tasks. In this contribution, we analyze the programming assignments of 600 students from an introductory university course in object-oriented programming. The students were provided unit tests for the assessment of their code, and their editing and testing actions were recorded using an Eclipse plug-in. The primary motivation for this study is to discover the programming strategies used by students for coding exercises with different difficulty levels, and find out if any relation exists between these strategies and the success in solving the coding tasks. More insights into this process will enable educators to provide future students timely, appropriate and constructive feedback on their coding process. Thus, to predict success in the coding exercises, we used indicators from students' testing behaviour reflecting the time and effort differences between two successive unit test runs. The results show a clear difference in the strategies employed by students within different success levels. The results also highlight ways of providing actionable feedback to the students in a timely and appropriate manner.

Keywords: Programming strategies · Personalized feedback
Computer science education

1 Introduction

Programming involves the process of generating a solution to a problem, thus one of the main learning outcomes of a programming course is to develop a student's ability to solve problems [31]. Therefore, it is important for educators to be responsive to "the problem-solving skills students bring to programming, and to those required by programming" because students are influenced by the facilitated strategies [33]. Soloway et al. managed to show that students' sensitivity to strategies while learning to program has significant effect on their performance [33]. However, first year students have a small skill set and the ability to read code [22]. Therefore, besides choosing the most appropriate programming approach,

V. Pammer-Schindler et al. (Eds.): EC-TEL 2018, LNCS 11082, pp. 326–339, 2018.
https://doi.org/10.1007/978-3-319-98572-5_25

programming environment and tools, the educators should consider conveying and teaching problem-solving strategies (e.g. hill climbing, trial and error, divide and conquer, top down, and bottom up) that students could exploit and apply while learning coding [2]. In addition, Felder says, that students "should be given the freedom to devise their own methods of solving problems rather than being forced to adopt the teacher's strategy" (p. 679) [16]. But all strategies are not equally good, thus students need feedback from educators in order to learn and improve. Moreover, the strategies that students employ to solve coding problems cannot be observed directly and must be inferred. Therefore, this study aims to analyze the program assignments of 600 students from an introductory Java university course. Consequently, we aim to investigate the programming behavior of freshmen while learning how to program, by utilizing data generated when solving their programming assignments. This allows us to ascertain the strategies students employ during coding activities and understand the efficiency of these different strategies, so educators can offer actionable feedback to nurture good programming habits and strategies [4]. Enhancing the learning experience of students with carefully designed coding exercises and support in assessing the required knowledge, should assist freshmen when faced with the difficulties of syntax and semantics, as well as understand error messages and control flow.

To capture students' programming behavior and identify their strategies, the authors extended the Eclipse programming tool with a plug-in for data collection. The goal of this study is to identify successful students' programming strategies. This will allow educators to provide meaningful personalized feedback promoting reflection and support, allowing students to improve the way they program. Consequently, the study addresses the following research questions:

RQ1: What programming strategies do freshmen employ to succeed in their assignments?

RQ2: Which actions can predict students' programming behavior and support educators in early detection of difficulties and misconceptions?

2 Related Work

Previous research has shown a multitude of individual factors influencing academic achievement at various educational levels (e.g. primary, secondary, university). Some of these factors include self-efficacy [14,35], personality traits (e.g. conscientiousness) [3,28], cognitive ability [6], prior knowledge and experience [14,35], and motivational and strategic (e.g. learning strategies) aspects [30].

Consciousness has been shown to be the personality trait that is most influential on academic achievement according to past studies [3,8,13,28]. Moreover it is the dimension most closely linked to the will to achieve [13]. Another key predictor of student learning and academic performance is self-regulated learning (SRL) [11,12,23,27]. SRL leads to deep cognitive engagement with the learning resources [11] which in turn transitions the extrinsic motivational behavior to behavior that is driven by intrinsic motivation [12]. This path from deep cognitive engagement to high levels of intrinsic motivation was found to be correlated

with student learning and academic achievement [40]. Another behavioral factor correlated with student learning (e.g. mastering the content) and academic achievement is performance approach [14] or deep strategy [30]. Deep learning strategies (when the student's focus is to attain understanding of the content and not merely obtaining a higher grade) result in mastering the content [14] which may lead to higher examination success [30]. In past studies, researchers show the difference between strategies (deep vs. surface) and their relation to academic achievement, and concluded that deep and surface strategies were positively and negatively correlated with academic achievement [7], respectively. Finally, previous research has shown that intellectual (cognitive/mental) ability influences academic performance. Intellectual abilities can be measured in different ways such as IQ [1], general mental ability (American College Test scores) [35] and logical reasoning [9]. Although several different factors can influence student academic achievement, when it comes to programming, problem solving ability demonstrates the most significant correlation with student performance in solving coding tasks [21]. In this contribution we will focus on the behaviour of the students rather than the above mentioned constructs. These previous contribution are to give reader a brief summary of which factors affect the academic achievement.

In computer science education, student assessment still abides by traditional outcome-based assessment [10]. However, programming is more than just the capability to generate code. It is a problem solving skill. Past research has shown that this assumption has been neglected, leading to a gap in students' ability to apply core programming concepts to real-world problems [32,37]. To address this issue, educators must be able to guide students in determining correct strategy, and identifying the appropriate time to abandon an inefficient approach [17]. Thus, researchers need to collect more authentic data and explore the processes by which students arrive at their final solutions [34]. This idea has become reality with the increase in popularity and usage of automated code testing and assessment in computer science education. Existent systems aid educators in assessing various features of coding assignments and scale the assessment up for large courses [15]. For instance, Jadud introduced the idea of researching students' compilation behaviour (i.e. "the programming behaviour students engage in while repeatedly editing and compiling their programs"), to better understand how students progress through a programming task, so that appropriate interventions can be applied [19]. Following this idea, Blikstein et al. utilized code snapshots to uncover differences between novices and experts' programming strategies [4]. Expanding on these past research studies, we extended the Eclipse tool to collect data portraying students' programming behaviour; with a goal to explore students strategies when solving coding tasks and their success in doing so.

Feedback is one of the most powerful variables influencing learning [18]. However, feedback is of little use if it only conveys a message of right or wrong. Feedback must be meaningful and actionable in order to help the learning process. Traditionally, in computer science education, students receive basic level

of feedback presented by the compiler [29]. Compiler messages are not always helpful, as they do not allow students to understand why they fail in solving the coding task. In most cases, coding tasks have multiple ways of achieving multiple solutions. To complete programming tasks, students apply strategies that build on their previous knowledge [20]. This led researchers to categorize students based on their programming behavior and employed strategies. Perkins et al. classify novice programmers as "stoppers" and "movers" based on the strategy they choose when facing a problem [25]. Turkle and Papers proposed two categories, "tinkerers" and "planners" [36], while Bruce et al. identified five: "followers", "coders", "understanders", "problem solvers", and "participators" [5]. Turkle and Papert's idea was not only related to categorizing the novice programmers, but also conveying epistemological pluralism. Epistemological pluralism highlights that students can have separate approaches to the same problem and communicate different behavior (e.g. "tinkerer" or "planner") while achieving similar results. Consequently, educators recognized the importance of the students learning process when learning how to program, and developed tools and systems to support this progress [24,29,39]. This study contributes to a data-driven development of personalized feedback in programming by using the writing and testing behavioral indicators of the students as they attempt to solve coding exercises. Our aim for this contribution is to keep the behavioral indicators as semantic-less as possible to attain greater generalizability and reproducibility of results.

3 Methodology

3.1 Research Objectives

The context of this research is a compulsory course in object-oriented programming (OOP). This course is offered to second semester CS-majors (600 students) in Java. As an introductory to OOP, there is a substantial variation in motivation and skills. This course is the basis for later software development courses, thus, it is important to identify struggling students early, provide appropriate feedback and help them develop good strategies for solving programming problems. Hence, the goal of the research is twofold: (1) identify programming strategies that lead to success in solving coding exercises; and (2) find ways to quickly detect student difficulties and misconceptions.

3.2 Assignment Structure

The course has 10 assignments with a reward of 100 points for completing each successfully. A student needs 750 points to qualify for the exam. Seven of the assignments (1-3, 5-6 and 8-9) are composed of smaller coding exercises with specific requirements indicating what to code. This allows us to use unit tests for automatic grading, as well as collect rich data regarding student progression. Students are encouraged to test by writing and launching their own testing

code. Due to the open nature of the remaining assignments (4, 7, 10), they have been excluded from this part of the study. The size (number of Java classes and methods) and difficulty level of exercises vary; thus, the students are granted a certain degree of freedom in selecting exercises based on their (self-assessed) skill level. Statistics indicate that exercise choice is evenly spread. As well, exercises use approximately the same amount of time each week.

3.3 Data Collection

We focus our data collection to the last 4 assignments, as the first three assignments were relatively basic for students to develop concrete strategy. For each of these exercises we provided Eclipse with detailed instructions about which files and activities to track. In particular, we collected the following data: (1) snapshots of files when they are saved, with compiler errors and warnings (2) student programs that are launched, typically for testing their own code (3) unit tests that are run, with information as to whether they pass or fail, and (4) the use of certain commands and panels, typically those used for debugging

All data is time-stamped and most are limited to the relevant files of a specific exercise, for both practical and privacy reasons. A special "Exercise panel" shows the details of which data has been collected, allowing the students to track their progress and review their process. The data is anonymized, but with identifiers corresponding to exam result, prior to its use in our research such that it can be correlated at a later stage.

3.4 Measurements

To analyze the behavior and predict the outcome of each assignment, we captured the following measures:

1. **Number of test runs:** is the total number of times a student ran the unit tests to check their code. This is counted for each exercise in every assignment.
2. **Improvement in unit test success:** each time a student ran the unit tests, they passed and/or failed a specific number of tests. The score they obtained is the number of passed tests divided by the total number of tests. As a result, the authors computed the improvement (or lack thereof) in this score between two consecutive test runs.

To predict and analyze a student's programming behavior in terms of the above mentioned measures, the authors also computed the following variables from the student's unit test running time series:

1. **Time difference launch:** is the average time difference between two consecutive launches of their own test code, before the students runs another unit test.
2. **Time difference edit:** is the average time difference between two consecutive logs of saving the file(s).

3. **Size difference:** is the difference in the number of lines of code between two consecutive unit test runs, i.e. code growth.
4. **Improvement in errors:** is the reduction in number of errors and warnings between two consecutive unit test runs.
5. **First test run score:** is the unit test success score of the first time a student ran a unit test for each exercise in every assignment.

4 Results

In this section, we present the prediction results followed by the behavioral analysis based on student categorization using an explanatory model.

Prediction Results. To predict the dependent variables: (1) improvement in unit test success and (2) the number of test runs, we used four different independent (also termed predictor) variables: (1) time difference launch, (2) time difference edit, (3) size difference, and (4) improvement in errors fitting a Generalized Additive Model (GAM). We divided the data set into 80% training and 20% testing set. We performed 5-fold cross-validation for both the training and testing. On one side, considering the improvements in the unit test success, in Table 1 we can see that the overall prediction error using the combined data of the four assignments is 0.11; and the average prediction error using data from each assignment separately is 0.18 (SD = 0.03). On the other side, in the same table, considering the number of test runs, we can see that the overall prediction error is 0.18 and the average prediction is 0.24 (SD = 0.04). Table 2 show the coefficients of the explanatory variables.

Table 1. Prediction results for the final score in a given assignment and the total number of test runs using data from individual assignments and the complete data sets.

Assignement ID	5	6	8	9	Overall
RMSE improvement in score	0.13	0.20	0.20	0.18	0.11
RMSE number of attempts	0.21	0.26	0.21	0.28	0.18

Relative to the number of test runs per individual assignment, we explore the question *how early can we predict?* Figure 1 demonstrates Root Mean Square Error (RMSE) of 0.10 from as early as the fourth test run. We can see that most of RMSE values are between 0.12 and 0.16, however the lowest value is observed at the 4^{th} test run. This facts can be seen as a "proof of concept" for the hypothesis regarding early prediction of the total number of test runs.

Explanatory Models. Table 2 shows the linear model fitted over the complete data set for the improvement of unit test success. We observe that the time difference launch and the difference in size are positively correlated with the improvement in unit tests success. These results support the assumption that students

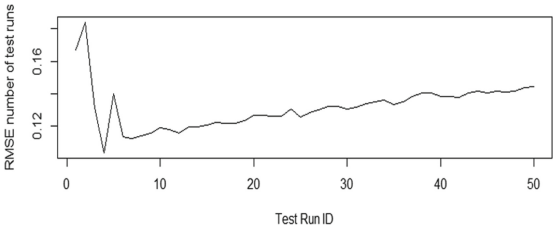

Fig. 1. RMSE values for predicting the total number of test runs using the data up to a given test runs ID.

Table 2. Linear model for score improvement and total number of tests run, all the exercises combined in one data set, bold t-values are significant ($p < 0.01$). Unbiased risk estimation for score improvement = 0.01 and for number of attempts = 0.03

	Improvement in score			Number of test runs		
	Estimate	Std. err.	t-val	Estimate	Std. err.	t-val
Intercept	1.78e-01	1.520e-02	**11.75**	4.764e+01	4.279e-01	**11.33**
Time diff launch	1.737e-06	2.958e-07	**5.82**	−2.945e-05	8.320e-06	**−3.54**
Time diff edit	1.928e-04	1.797e-07	0.13	−4.511e-05	5.063e-06	**−8.91**
Diff size	5.300e-02	1.415e-03	**2.95**	−2.975e-01	3.990e-02	**−7.45**
Diff error	−3.740e-02	2.089e-02	−1.79	13.694e-01	5.890e-01	0.62
Diff warning	−4.743e-02	5.008e-02	−0.94	1.491e+00	1.413e+00	1.05

who made larger and less frequent changes in their code showed greater improvement in unit test success. Furthermore, Table 2 also shows the linear model fitted over the complete data set for the number of tests run. Here we observe that the time difference launch and the difference in code size are negatively correlated to the number of test run. These results support the assumption that students who made larger and less frequent changes in code had fewer number of test runs. The average marginal effects are shown in Table 3.

Table 3. Average marginal effects for the models shown in Table 2

Dependent variable	Time diff launch	Time diff edit	Diff size	Diff error	Diff warning
Score improvement	1.701e-06	5.3e-06	0.0009	−0.03	−0.04
Number of test runs	−2.945e-05	−4.511e-05	−0.29	0.36	1.49

4.1 Categorization

In order to explain the coding behavior of the students in more details, we categorized the student population into three categories (i.e. intellects, thinkers,

and probers) based on the total number of unit test runs by each student. Table 4 presents the number of students belonging to each category for every assignment and Fig. 3 shows the change in category between two consecutive assignments. Assumptions for the suggested three categories of students, we would like to point out here that the pragmatic sense of the category labels might be different from our interpretation in the paper:

1. **Intellects:** run tests less frequently, as they are skilled and confident.
2. **Thinkers:** run tests more frequently, to receive early feedback regarding progress.
3. **Probers:** run tests most frequently, as they experience difficulty.

We would like to point out here that the categories are for each assignment and could change student to student and even for one student from one assignment to other.

Table 4. Number of students in the different categories for the separate assignments.

Data used	Thresholds	Intellects	Thinkers	Probers
Assignment 5	5, 14	163	131	160
Assignment 6	5, 10	173	140	141
Assignment 8	8, 19	138	132	126
Assignment 9	7, 13	88	85	62

The Difference from the Perspective of the Three Categories. We present the differences between the three categories with respect to the explanatory and dependent variables (Table 6). These results hold for individual assignments as well (barring a few exceptions) as shown in Table 5.

1. Significant difference on time between two student program launches (F [2,383] = 70.27, p = .00001): post-hoc pairwise comparisons show that intellects have higher time difference than thinkers; and thinkers have higher time difference than probers.
2. Significant difference on change in code between two tests (F [2,383] = 198.85, p = .00001): post-hoc pairwise comparisons show that intellects have greater code change than thinkers; and thinkers have greater code change than probers.
3. Significant difference on the average improvement in success (F [2,383] = 121.51, p = .00001): post-hoc pairwise comparisons show that intellects have greater success improvements than thinkers; while thinkers have greater success improvements than probers.
4. Significant difference on average change in number of errors and warnings (F [2,383] = 5.79, p = .01): post-hoc pairwise comparisons deptict intellects reduce more errors than thinkers; while thinkers and probers have no significant difference based on reducing the number of errors in the code.

5. Significant difference on average success in first test run (F [2,383] = 16.60, $p = .001$): post-hoc pairwise comparisons show that intellects score higher in the first attempt than thinkers; while thinkers and probers have no significant difference based on first test run scores.

Table 5. ANOVA results for difference measures for the three categories.

	Assignment 5		Assignment 6		Assignment 8		Assignment 9	
	F	p	F	p	F	p	F	p
Time diff launch	37.95	.0001	24.41	.0001	66.28	.0001	2.6	.10
Diff size	17.95	.0001	56.00	.0001	50.01	.0001	45.41	.0001
Diff success	94.87	.0001	39.99	.0001	60.93	.0001	31.00	.0001
Diff error	4.7	.03	2.13	.14	0.61	.43	0.65	.41
Score 1st attempt	2.4	.11	4.65	.03	10.46	.001	5.07	.02

Figure 2 shows the explanatory variables corresponding to the three categories with progress based on the number of test runs. Upon inspection of Fig. 2, (left panels) it is evident that there exists a clear difference in the time between two student program launches and the average improvement between the intellects (shown with red) and the remaining two categories for the test runs 5–10 (i.e. time between main method launches) and 15–25 (i.e. improvement). However, the other differences are not as pronounced.

From the explanatory models for each category (Table 6), we observe that the behavior of the students in each category is subtly different than the other two categories. The intellects have two positively significant coefficients: the wait between two student program launches and the change in code size. This indicates that intellects take their time to alter the code and remove errors and bugs. The thinkers have only one positively significant coefficient: the wait between two student program launches. That means the thinkers take time to test, but nothing clearly can be said about the other parameters. The probers have change in code as a negative and significant coefficient, meaning that they make smaller code changes between two successive unit tests runs.

Table 6. Linear model for improvement with all the exercises combined in three data sets, one each for intellects, thinkers, probers, bold t-values are significant ($p < 0.01$).

	Intellects			Thinkers			Probers		
	Estimate	std. err.	t-val	Estimate	std.err	t.val	Estimate	std.err	t-val
Intercept	2.9e-01	2.8e-02	**10.29**	1.6e-01	2.6e-02	**6.36**	8.5e-02	2.2e-02	**3.70**
Time diff launch	1.7e-06	4.5e-07	**3.76**	1.4e-06	6.4e-07	**2.19**	1.5e-06	4.8e-07	**3.13**
Time diff edit	8.7e-07	2.8e-07	**3.07**	−7.0e-08	3.3e-07	−0.21	−4.8e-08	3.3e-07	−0.14
Diff size	−2.3e-03	2.2e-03	−1.02	−1.2e-03	2.4e-03	−0.52	−6.2e-03	3.0e-03	**2.07**
Diff error	−6.9e-02	3.7e-02	−1.83	−7.6e-04	3.1e-02	−0.02	−5.2e-02	3.9e-02	−1.30
Diff warning	−1.1e-02	9.5e-02	−0.12	4.5e-02	8.6e-02	0.52	−1.5e-01	7.5e-02	**−2.00**

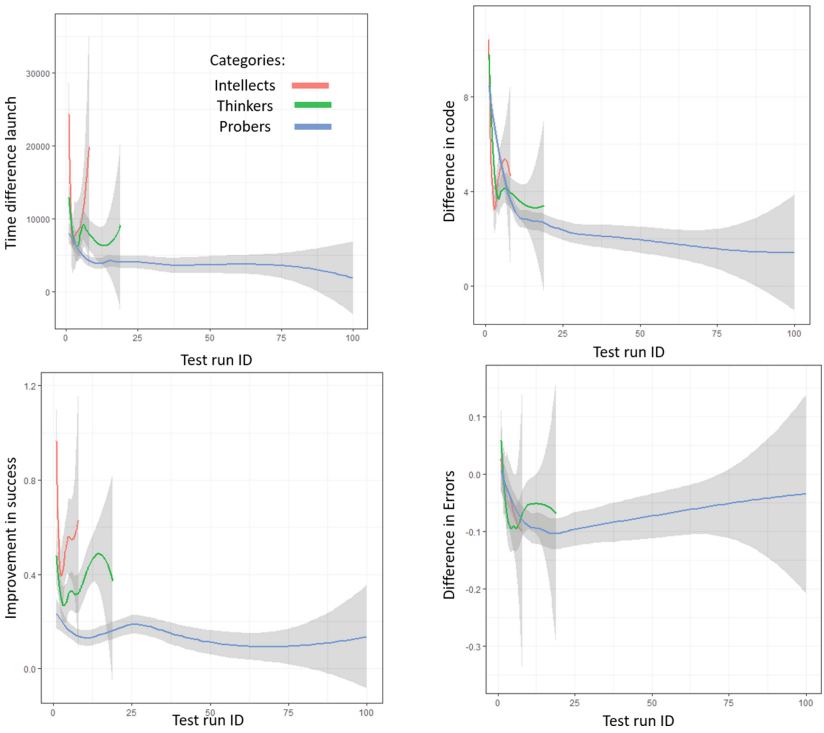

Fig. 2. Different measures for the three categories for each test run ID. (Color figure online)

Finally, it could be expected that students belong to more than one category while attempting to solve programming assignments. Figure 3 shows students changing across the categories intellects, thinkers and probers, for different assignments. For example, the intellects are a larger group (163) than the

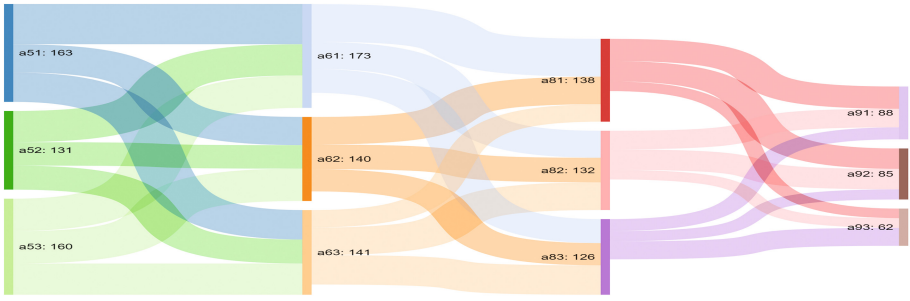

Fig. 3. Students changing their strategies across the different assignments. $a51 : 233$ shows that in assignment a5, there were 233 students in category 1. Category labels: 1 = intellects; 2 = thinkers; 3 = probers.

thinkers (131) for assignment 5 ($a5$); for the next assignment (i.e., $a6$) we see that similar to $a5$, the largest category is intellects followed by similar numbers of thinkers and probers. Also, a large majority of intellects did not change category, while most thinkers and probers either stayed the same or interchanged categories.

5 Conclusion and Discussion

In this study we analyzed the programming patterns of 600 students from an introductory university course in object-oriented programming using an Eclipse plug-in to collect data. Results from the analyses supported our two assumptions: (1) there are different programming strategies that lead students to success when attempting to solve coding exercises, and (2) we can early identify low performers. Using semantic-less measures from students' coding and debugging behavior (e.g. time difference launch, time difference edit) and one code-base measure (i.e. growth in size), we managed very early (fourth attempt) to predict improvement in unit test success at a low granularity level of one student with one assignment. Our focus on semantic-less-ness lead to better reproducibility and generalizability of the results, because we can not, at least with current state-of-art, know without explicitly asking students if they are experiencing difficulty with the coding constructs (e.g. loops, recursion) or in the domain (e.g. Fibonacci numbers). Moreover, our study also adds to the growing body of research utilizing low granularity data compared to previous studies that have successfully provided predictive models that either looked at the students' level as a whole class, or focused only on code-based variables [4,26,38]. In addition, none of the previous studies attempted early prediction.

Furthermore, we also presented behavioral analysis of students practicing different programming strategies. Thus, we can say that *intellects* as a group are characterized by having the highest first test run score; the highest improvement in unit test success; the lowest total number of test runs among the three categories; the longest wait time between two student program launches; and finally, the most changes in the code between two unit tests. *Thinkers* are characterized as follows: a low first unit test score; a short wait time between two successive student program launches; a lower change in code size than the intellects but higher than the probers; and unit test success that is higher than the probers but lower than the intellects. Finally, *probers* are characterized by having low first unit test score; the shortest wait time between two successive student program launches; the least code size change between two successive tests; and finally, the least improvement in unit test success. The key difference between thinkers and probers is the modifications they make to the code in a similar duration of time. The thinkers appear to have a strategy to fix errors and bugs in the code, while the probers appear to employ a trial and error approach. This is also evident from Fig. 2 (bottom-left), where we can see that for a large number of attempts, the probers have slow growth (close to 0.25, that is, 4 unit test runs for passing one unit test); where as, after certain test runs students from the remaining two

categories require one or two test runs to pass one unit test. This exponential improvement is demonstrated earlier by the intellects than the thinkers, indicating that intellects initially make fewer mistakes and hence require fewer test runs to pass the complete set of unit tests. However, thinkers show more regulated and informed behaviour of testing the code than probers, and this might be a plausible explanation for why probers require more tests run to pass all of the unit tests. Consequently, from past studies we know that the weaker students have less understanding of what is tested by each test, and that makes them more likely to use a trial and error approach [25].

Finally, the prediction results presented in this study could support educators in providing motivational feedback to act as incentive to students to test their code a few more times before giving up. For example, we can predict the number of tests run a student would carryout at an early stage and we can also predict their projected improvement in unit test success at each test run. Given the current *TestRunID* and unit test score of the student, we could provide him/her with a target number of test runs at his/her given pace of improvement which might motivate the student to change their strategy (from probing to thinking) or to continue testing the code (if he/she is relatively close to the target number of tests run).

Limitations and Future Work. Our approach carries a few limitations that we plan to overcome in the next studies. First, this is a "black box" approach because we do not examine the code, instead we look into behavioral patterns when coding. In future work, we plan to analyze the mistakes made by the students and observe the corresponding strategic category. Next, we also did not consider any semantic features computed from the code; incorporating code metrics into the analysis could improve the prediction results. Finally, we do not gather or utilize data about students (e.g. consciousness, SRL, exam performance) or their motivation during the course, which hinders us in providing personalized feedback at this stage. Thus, we plan to incorporate this information in future studies in order to provide feedback that is not only timely and actionable, but personalized and adaptive as well.

References

1. Alloway, T.P., Alloway, R.G.: Investigating the predictive roles of working memory and IQ in academic attainment. J. Exp. Child Psychol. **106**(1), 20–29 (2010)
2. Barnes, D.J., Fincher, S., Thompson, S.: Introductory problem solving in computer science. In: 5th Annual Conference on the Teaching of Computing, pp. 36–39 (1997)
3. Barrick, M.R., Mount, M.K., Strauss, J.P.: Conscientiousness and performance of sales representatives: test of the mediating effects of goal setting. J. Appl. Psychol. **78**(5), 715 (1993)
4. Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., Koller, D.: Programming pluralism: using learning analytics to detect patterns in the learning of computer programming. J. Learn. Sci. **23**(4), 561–599 (2014)

5. Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., Stoodley, I.: Ways of experiencing the act of learning to program: a phenomenographic study of introductory programming students at university. In: Transforming IT Education: Promoting a Culture of Excellence, pp. 301–325 (2006)
6. Busato, V.V., Prins, F.J., Elshout, J.J., Hamaker, C.: Intellectual ability, learning style, personality, achievement motivation and academic success of psychology students in higher education. Pers. Individ. Differ. **29**(6), 1057–1068 (2000)
7. Cano, F.: Epistemological beliefs and approaches to learning: their change through secondary school and their influence on academic performance. Br. J. Educ. Psychol. **75**(2), 203–221 (2005)
8. Chamorro-Premuzic, T., Furnham, A.: Personality traits and academic examination performance. Eur. J. Pers. **17**(3), 237–250 (2003)
9. Chamorro-Premuzic, T., Furnham, A.: Personality, intelligence and approaches to learning as predictors of academic performance. Pers. Individ. Differ. **44**(7), 1596–1603 (2008)
10. Cooper, S., Cassel, L., Moskal, B., Cunningham, S.: Outcomes-based computer science education. In: ACM SIGCSE Bulletin, vol. 37, pp. 260–261. ACM (2005)
11. Corno, L., Mandinach, E.B.: The role of cognitive engagement in classroom learning and motivation. Educ. Psychol. **18**(2), 88–108 (1983)
12. Corno, L., Rohrkemper, M.: The intrinsic motivation to learn in classrooms. Res. Motiv. Educ. **2**, 53–90 (1985)
13. Digman, J.M.: Five robust trait dimensions: development, stability, and utility. J. Pers. **57**(2), 195–214 (1989)
14. Diseth, Å.: Self-efficacy, goal orientations and learning strategies as mediators between preceding and subsequent academic achievement. Learn. Individ. Differ. **21**(2), 191–195 (2011)
15. Edwards, S.H., Perez-Quinones, M.A.: Web-CAT: automatically grading programming assignments. In: ACM SIGCSE Bulletin, vol. 40, pp. 328–328. ACM (2008)
16. Felder, R.M., Silverman, L.K., et al.: Learning and teaching styles in engineering education. Eng. Educ. **78**(7), 674–681 (1988)
17. Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., Zander, C.: Debugging from the student perspective. IEEE Trans. Educ. **53**(3), 390–396 (2010)
18. Hattie, J., Timperley, H.: The power of feedback. Rev. Educ. Res. **77**(1), 81–112 (2007)
19. Jadud, M.C.: Methods and tools for exploring novice compilation behaviour. In: Proceedings of the Second International Workshop on Computing Education Research, pp. 73–84. ACM (2006)
20. Kiesmüller, U.: Diagnosing learners problem-solving strategies using learning environments with algorithmic problems in secondary education. ACM Trans. Comput. Educ. **9**(3), 17 (2009)
21. Lishinski, A., Yadav, A., Enbody, R., Good, J.: The influence of problem solving abilities on students' performance on different assessment tasks in CS1. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education, pp. 329–334. ACM (2016)
22. Lister, R., et al.: A multi-national study of reading and tracing skills in novice programmers. In: ACM SIGCSE Bulletin, vol. 36, pp. 119–150. ACM (2004)
23. Maldonado-Mahauad, J., Pérez-Sanagustín, M., Kizilcec, R.F., Morales, N., Munoz-Gama, J.: Mining theory-based patterns from big data: identifying self-regulated learning strategies in massive open online courses. Comput. Hum. Behav. **80**, 179–196 (2018)

24. Mitchell, C.M., Boyer, K.E., Lester, J.C.: When to intervene: toward a Markov decision process dialogue policy for computer science tutoring. In: The First Workshop on AI-supported Education for Computer Science, p. 40 (2013)
25. Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., Simmons, R.: Conditions of learning in novice programmers. J. Educ. Comput. Res. **2**(1), 37–55 (1986)
26. Piech, C., Sahami, M., Koller, D., Cooper, S., Blikstein, P.: Modeling how students learn to program. In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, pp. 153–160. ACM (2012)
27. Pintrich, P.R.: A conceptual framework for assessing motivation and self-regulated learning in college students. Educ. Psychol. Rev. **16**(4), 385–407 (2004)
28. Poropat, A.E.: A meta-analysis of the five-factor model of personality and academic performance. Psychol. Bull. **135**(2), 322 (2009)
29. Rivers, K., Koedinger, K.R.: Automatic generation of programming feedback: a data-driven approach. In: The First Workshop on AI-Supported Education for Computer Science, vol. 50 (2013)
30. Rodriguez, C.M.: The impact of academic self-concept, expectations and the choice of learning strategy on academic achievement: the case of business students. High. Educ. Res. Dev. **28**(5), 523–539 (2009)
31. Saeli, M., Perrenet, J., Jochems, W.M., Zwaneveld, B.: Teaching programming in secondary school: a pedagogical content knowledge perspective. Inform. Educ. **10**(1), 73–88 (2011)
32. Simon, B., Chen, T.Y., Lewandowski, G., McCartney, R., Sanders, K.: Commonsense computing: what students know before we teach (episode 1: sorting). In: Proceedings of the Second International Workshop on Computing Education Research, pp. 29–40. ACM (2006)
33. Soloway, E., Bonar, J., Ehrlich, K.: Cognitive strategies and looping constructs: an empirical study. Commun. ACM **26**(11), 853–860 (1983)
34. Soloway, E., Ehrlich, K.: Empirical studies of programming knowledge. In: Readings in Artificial Intelligence and Software Engineering, pp. 507–521. Elsevier (1986)
35. Stajkovic, A.D., Bandura, A., Locke, E.A., Lee, D., Sergent, K.: Test of three conceptual models of influence of the big five personality traits and self-efficacy on academic performance: a meta-analytic path-analysis. Pers. Individ. Differ. **120**, 238–245 (2018)
36. Turkle, S., Papert, S.: Epistemological pluralism and the revaluation of the concrete. J. Math. Behav. **11**(1), 3–33 (1992)
37. VanDeGrift, T., Bouvier, D., Chen, T.Y., Lewandowski, G., McCartney, R., Simon, B.: Commonsense computing (episode 6): logic is harder than pie. In: Proceedings of the 10th Koli Calling International Conference on Computing Education Research, pp. 76–85. ACM (2010)
38. Vee, M., Meyer, B., Mannock, K.L.: Understanding novice errors and error paths in object-oriented programming through log analysis. In: Proceedings of Workshop on Educational Data Mining at the 8th International Conference on Intelligent Tutoring Systems, pp. 13–20 (2006)
39. Vihavainen, A., Vikberg, T., Luukkainen, M., Pärtel, M.: Scaffolding students' learning using test my code. In: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, pp. 117–122. ACM (2013)
40. Zimmerman, B.J., Schunk, D.H.: Reflections on theories of self-regulated learning and academic achievement. In: Self-Regulated Learning and Academic Achievement, pp. 282–301. Routledge (2013)