Evidence for programming strategies in university coding exercises

No Author Given

No Institute Given

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 patterns of 600 students from an introductory university course in object-oriented programming. The students were provided unit tests for the assessment of their codes, 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 the 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 the success in the coding exercises, we used indicators from students' testing behaviour reflecting the time and the effort differences between two successive unit test runs. The results show a clear difference in the strategies employed by the students within different success levels. Moreover, the results also highlight the potential ways of providing actionable feedback to the students in a timely and appropriate manner.

Keywords: programming strategies · personalized feedback · computer science education

1 Introduction Katerina

Programming is considered to be a problem-solving skill. 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 [37]. Soloway et al. managed to show that students' sensitivity to strategies while learning to program has significant effect on their performance [37]. However, first year students lack variety of skills and also ability to read code [24]. Therefore, besides choosing the most appropriate programming approach, programming environment and tools, the educators should consider conveying and teaching problem-solving strategies (e.g. hill climbing, trial and error, top down, and bottom up) that students could exploit and apply while learning coding [2, 22]. In addition, Felder says, the 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) [17]. But all strategies are not equally good, so students need feedback from educators in order to learn and improve. Moreover, the strategy that students employ

to solve a coding problem cannot be observed directly but must be inferred. Therefore, this study aims to analyze the program submission patterns of 600 students from an Introductory Java University course. Consequently, we want to investigate the early behavior of first year students learning to program by utilizing more fine-grained data to reveal their strategies in the coding activities, so that the educators could offer actionable real-time feedback [4, 35]. Enhancing the learning experience of students with carefully designed coding assignments and support in assessing the required knowledge, should aid first year students to face the difficulties with syntax and semantics, and understand error messages and control flow.

To get more insight into the students' problem-solving strategies, the authors extended the Eclipse IDE (Integrated Development Environment) used in the course with a data collection facility (i.e. Eclipse plug-in) that allows us to identify students' strategies that lead to success in learning to program. Furthermore, the widespread collection of data with the use of technology, unlock opportunities for educators to engage learning analytics and derive models of student behavior to develop early indicators when students struggle to successfully finish a coding task. The predictive models need to offer educators a rich set of relevant measures and more insight into the learning process considering students' individual differences (e.g. prior knowledge, meta-cognitive skills, learning strategies, etc.) and predict the student performance. Thus, the ultimate goal of this study is to enhance the learning and programming environment with relevant learning analytics and predictive models that improve the feedback by considering personalization and adaptation. This way, the educators could forward a meaningful personalized feedback to promote reflection and support students to improve their programming strategies. Consequently, the study addresses the following research questions:

- **RQ1:** What type of programming strategies do first year students employ to succeed in assignments?
- **RQ2:** What variables could predict students' programming behavior and support educators to early detect struggling students?

2 Related Work

2.1 Student academic performance and success Serena

Previous research has shown a multitude of individual factors influence academic achievement at various educational levels (e.g. primary, secondary, university). Some of these factors include self-efficacy [15, 39], personality traits (e.g. conscientiousness) [32, 3], cognitive ability [7], prior knowledge and experience [15, 39], motivational and strategic (e.g. learning strategies) aspects [34].

Consciousness has been shown to be the personality trait that is most influential on academic achievement according to past studies [14, 32, 3, 9]. Moreover it is the dimension most closely linked to the will to achieve [14]. Another key predictor of student learning and academic performance is the self-regulated learning (SRL) [12, 13, 31, 25]. SRL leads to a deeper cognitive engagement with the learning resources [12] which in turn transitions the extrinsic motivational

behaviour to behaviour that is driven by intrinsic motivation [13]. This path of deeper cognitive engagement to high levels of intrinsic motivation was found to be correlated with student learning and academic achievement [44]. Another behavioural factor found to be correlated with student learning (e.g. mastering the content) and academic achievement is performance approach [15] or deep strategy [34]. Deep learning strategies result in mastering the content [15] which might lead to higher examination success [34]. 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 [8, 5], respectively. Finally, previous research had shown that intellectual (cognitive/mental) ability have an influence on academic performance. Intellectual abilities can be measured in different ways such as IQ [1], general mental ability (American College Test scores) [39] and logical reasoning [10]. Although multiple 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 [23, 30]. This correlation does not consider multiple choice assessment but constructed response assessment which fits appropriately in measuring thinking and problem solving capacities.

2.2 Assessment in computer science education Katerina

The evaluation process in computer science education is still following the traditional outcome-based assessment [11]. However, programming is a problem-solving skill and not just a capability to generate code. Past research has shown that this assumption has been neglected, leading to a gap in students' ability to apply core programming concepts in real-world computing tasks [36, 41]. Moreover, without educators' support to teach students what is the right strategy and when to abandon an inefficient one, students are not able to efficiently identify problems [18]. Thus, researchers collected more fine-grained data and explored the processes by which students arrived at the final solution [38]. This idea has become reality with the increase in popularity and usage of automated assessment in computer science education. Automated assessment systems aid educators to assess various features of coding assignments and scale it up for large courses [16]. However, even with the vast amounts of collected data, not until recently were researchers looking into understanding how humans actually solve coding tasks. Thus, 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 [20]. Following this idea, Blikstein utilized code snapshots to uncover differences between novices and experts' programming strategies [4]. Other studies also looked at students programming behaviour [19]. Expanding on these past research studies, we extended the Eclipse IDE used in the course with a data collection facility (i.e. Eclipse plug-in) to collect data about students' programming activity. The goal is to explore students strategies when solving coding tasks and their success in doing so. Moreover, exploring students

strategies in combination with their personality traits and motivational choices as described in the previous section, could be used to develop models of student behavior that will assist educators in predicting students' programming steps so that they can intervene by offering actionable and personalized feedback to struggling students.

2.3 Personalized feedback in learning programming Katerina

Learning is interchangeably connected with feedback, as processes that support acquisition of knowledge. However, feedback does not help much if it conveys a message of right or wrong. It needs to me meaningful and actionable in order to help the learning process. Traditionally, in computer science education, students receive basic level of feedback, usually from the compiler [33]. But this feedback carries limitations. Compiler messages are not helping students understand why they fail to solve the coding task. Moreover, every coding task can have multiple paths a student can take to arrive at a solution. And every student can apply strategies that build on their previous knowledge [21]. 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 [28]. Turkle and Papers proposed two categories, "tinkerers" and "planners" [40], while Bruce et al. identified five: "followers", "coders", "understanders", "problem solvers", and "participators" [6]. Turkle and Papert's idea was not only related to categorizing the novice programmers, but also wanted to convey the epistemological pluralism. Epistemological pluralism highlights that students can have different approaches to the same problem and communicate different behavior (e.g. "tinkerer" or "planner") without achieving better results. Consequently, educators recognize the importance of the students learning process how to program, and developed tools and systems that support their learning progress [43, 33, 26]. However, researchers still work to personalize and automate the feedback when learning programming [27, 33, 26]. Thus, this study contributes to a data-driven development of timely and actionable feedback in programming by using the writing and testing behavioural indicators of the students while they attempt to solve coding exercises. Our aim, for this contribution, is to keep the behavioural indicators as semantic-less as possible to attain a greater generalizability and reproducibility of results.

3 Methodology

3.1 Research objectives Hallvard

The context of this research is a bachelor course in object-oriented programming (with Java). The course has 600+ students and is a mandatory second semester course for most of them, while some choose it in later semesters. There is a substantial variation in motivation, skills and talent, and since this course is the basis for later software development courses, 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 (is this aligned with what has been written earlier?):

1. identify coding strategies that lead to success in solving exercises
2. find ways of detecting struggling students early

Note that the coding strategies we look for are not necessarily the same as those employed by mature developers, as the learning process is different from professional development and the exercises are different from real-world programming problems.

3.2 Assignment structure Hallvard

The course has 10 assignments with a reward of 100 points for completing each successfully, and the students need 750 points to qualify for the exam. 7 of the assignments (1-3, 5-6 and 8-9) are composed of smaller exercises with specific requirements about what to code that allow us to use unit tests for automatic grading and collect detailed data about what they do and their progress. The other 3 (4, 7 and 10) give the students more freedom, and are not considered here, since they are less suited for the kind of data collection we need.

The size (number of Java classes and methods) and difficulty level vary, and the students are to a certain degree allowed to choose based on their (self-assessed) skill level. Statistics indicate that they spread out pretty evenly and use approximately the same amount of time each week.

3.3 Data collection Hallvard

We have limited the data collection to the latest 4 assignments, since the students needed some time to get used to the Eclipse programming tool. For each of these exercises we provided Eclipse with detailed instructions about which files and activities it should collect data about. Currently, we can collect 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 and whether they pass or fail
4. the use of certain commands and panels, typically those used for debugging

All data are time-stamped and most of them are limited to the relevant files of a specific exercise, both for practical and privacy reasons. A special "Exercise panel" shows in detail what data has been collected, hence, the students may track their progress and review their process. When the exercise is finished or the deadline reached, they submit the collected data the e-learning system.

There are some issues with last year's data set used in this research. First, data is only recorded when the Exercise panel is open, and if students forget to do so, data will be missing. Also, code snapshots are only available when the students save it, so the time between snapshots will vary. At least, the code will be saved when the launch their code or the provided tests. Finally, if the students develop on several machines and don't share files across, data may be lost.

Before being used for research, the data is made anonymous, but with identifiers corresponding to exam result, so they can be correlated at a later stage.

3.4 Hypothesis

Although the collected data may support research in many directions, this research focuses on categorizing and identifying students' programming strategies. From observing students and manually inspecting the data, we have noticed distinct modes of working which we believe can be used for categorization:

- **Productive:** characterized by code growth and increase in passed tests in large chunks, as if the programmer is good and confident
- **Incremental:** also characterized by code growth and increase in passed tests, but in smaller chunks, as if the programmer is good, but cautious
- **Debugging:** characterized by little or no code growth and slow increase in passed tests, as if the programmer is fixing bugs
- **Struggling:** similar to debugging, but more based on trial and error than a plan

During the work on an exercise, a student will typically switch between modes, and the pattern will depend on several characteristics of both the student and exercise, e.g. how the skilled and confident the programmer is, and how tricky and big the exercise is. E.g. a skilled programmer will in general be productive, but if the exercise is tricky s/he may nevertheless have periods of debugging. A less skilled but still confident student, may write code in large chunks, but end up struggling a lot. The characteristics of these modes overlap, e.g. frequent testing is typical of all but the first mode. Here we are not interested in the modes the themselves, but our hypothesis is that we may understand more about programming strategies, by looking at variables related to these modes.

3.5 Variables Kshitij

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

1. **Number of test runs:** is the total number of times students ran the unit tests to check their code. This is counted for each exercise in every assignment.
2. **Improvement in unit test success:** every time students ran the unit tests, they passed and/or failed a certain 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 the students' programming behaviour in terms of the above mentioned measures, the authors also computed the following variables from the student unit test running time series:

1. **Time difference launch:** is the average time difference between two consecutive student program launches before students run 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 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 attempt score:** is the unit test success score for the very first time students ran a unit test for each exercise in every assignment.

4 Results Kshitij

In this section, we first present the prediction results and then the behavioural analysis based on the student categorization using an explanatory model.

Prediction results. To predict the dependent variables, improvement in unit test success and the number of test runs, we used four different predictor variables: time difference launch, time difference edit, size difference, and improvement in errors utilizing a Generalized Additive Model (GAM). On one side, considering the improvements in the unit test success, in Table 1 we can see that the overall prediction error considering the combined data of the four assignments is 0.11; and the average prediction error considering data from each assignment separately is 0.18 (SD = 0.03). On the other side, in the same table, considering the number of attempts, we can see that the overall prediction error is 0.18 and the average prediction is 0.24 (SD = 0.04). Tables 2 and 3 show the coefficients of the explanatory variables.

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

Data used	RMSE for final score	RMSE for number of attempts
Assignment 5	0.13	0.21
Assignment 6	0.20	0.26
Assignment 8	0.20	0.21
Assignment 9	0.18	0.28
Overall	0.11	0.18

Regarding the number of test runs for each individual assignment, we explored the fact *how early can we predict*. Figure 1 shows that the prediction results from as early as the 4th attempt show Root Mean Square Error (RMSE) of 0.10.

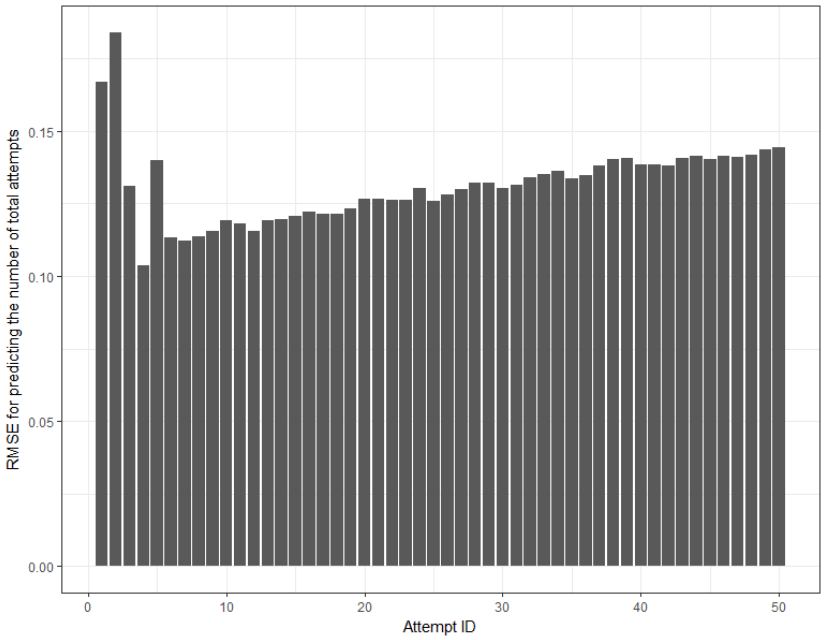

Fig. 1. Caption

Table 2. Linear model for average improvement all the exercises combined in one data set

	Estimate	**Std. error**	**t value**	**p value**
intercept	1.786e-01	1.520e-02	11.75	.00001
Time diff launch	1.737e-06	2.958e-07	5.82	.0001
Diff size	5.300e-07	1.797e-07	2.95	.003
Time diff edit	1.928e-04	1.415e-03	0.13	0.89
Diff error	-3.740e-02	2.089e-02	-1.79	0.07
Diff warning	-4.743e-02	5.008e-02	-0.94	0.34

Explanatory models. Table 2 shows the linear model fitted over the complete data set for the improvement in unit test success. We can 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 the students who made larger and less frequent changes in the code showed more improvement in the unit test success. Furthermore, Table 3 shows the linear model fitted over the the complete data set for the number of attempts. Here we can observe that the time difference launch and the difference in code size are negatively correlated to the number of attempts. These results support the

assumption that the students who made larger and less frequent changes in the code had fewer number of attempts.

Table 3. Linear model for number of attempts with all the exercises combined in one data set

	Estimate	**Std. Error**	**t value**	**p value**
intercept	4.764e+01	4.279e-01	111.33	.00001
Time diff launch	-2.945e-05	8.320e-06	-3.54	.0004
Time diff edit	-4.511e-05	5.063e-06	-8.91	.00001
Diff size	-2.975e-01	3.990e-02	-7.45	.0001
Diff error	3.694e-01	5.890e-01	0.62	.53
Diff warning	1.491e+00	1.413e+00	1.05	.29

4.1 Categorization

In order to explain the coding behaviour of the students in more details, we categorized the whole student population into three categories (i.e. intellects, thinkers, and probers) based on the total number of unit test runs from every student. Table 4 presents the number of students in each category for every assignment and Figure 4 shows the change in category between two consecutive assignments.

Assumptions for the suggested three categories of students:

1. Intellects: run tests less frequently, because they are skilled and confident.
2. Thinkers: run tests more frequently, to get an early feedback about progress.
3. Probers: run tests even more frequently, because they struggle.

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

Data used	**Thresholds**	**Intellects**	**Thinkers**	**Probers**
Assignment 5	5, 14	131	129	131
Assignment 6	5, 10	224	163	193
Assignment 8	8, 19	123	106	109
Assignment 9	7, 13	119	90	105

The difference from the perspective of the three categories. The following are the differences between the three categories in terms of the explanatory and dependent variables (Figures 2 and 3; Tables 6,7,8). These results hold for the individual assignments as well (barring a few exceptions) as shown in Table 5.

– Significant difference on time between two student program launches (F [1,383] = 70.27, p = .00001); post-hoc pairwise comparisons show that intellects have higher time difference than thinkers; and thinkers and probers

have no significant difference based on time between two student program launches (Figure 2 top-right).

- Significant difference on change in code between two tests (F [1,383] = 198.85, $p = .00001$) post-hoc pairwise comparisons show that intellects have more code changed than thinkers; and thinkers have more code change than the probers (Figure 2 bottom-left).
- Significant difference on the average improvement in success (F [1,383] = 121.51, $p = .00001$); post-hoc pairwise comparisons show that intellects have more success improvements than thinkers; and thinkers have more success improvements than the probers (Figure 2 top-left).
- Significant difference on average change in number of errors and warnings (F [1,383] = 5.79, $p = .01$); post-hoc pairwise comparisons show that intellects reduce more errors than thinkers; and thinkers and probers have no significant difference based on reducing the number of errors in the code (Figure 2 bottom-right).
- Significant difference on average success in first attempt (F [1,383] = 16.60, $p = .001$); post-hoc pairwise comparisons show that intellects score more in the first attempt than thinkers; and thinkers and probers have no significant difference based on first attempt scores.

Table 5. My caption

	Assignment5		**Assignment6**		**Assignment8**		**Assignment9**	
	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

Table 6. Linear model for improvement with all the exercises combined in one data set – intellects

	Estimate	**Std. error**	**t value**	**p value**
intercept	2.952e-01	2.869e-02	10.29	.00001
Time diff launch	1.727e-06	4.592e-07	3.76	.0001
Time diff edit	8.786e-07	2.855e-07	3.07	.002
Diff size	-2.303e-03	2.241e-03	-1.02	0.30
Diff warning	-6.971e-02	3.796e-02	-1.83	0.06
Diff error	-1.145e-02	9.524e-02	-0.12	0.90

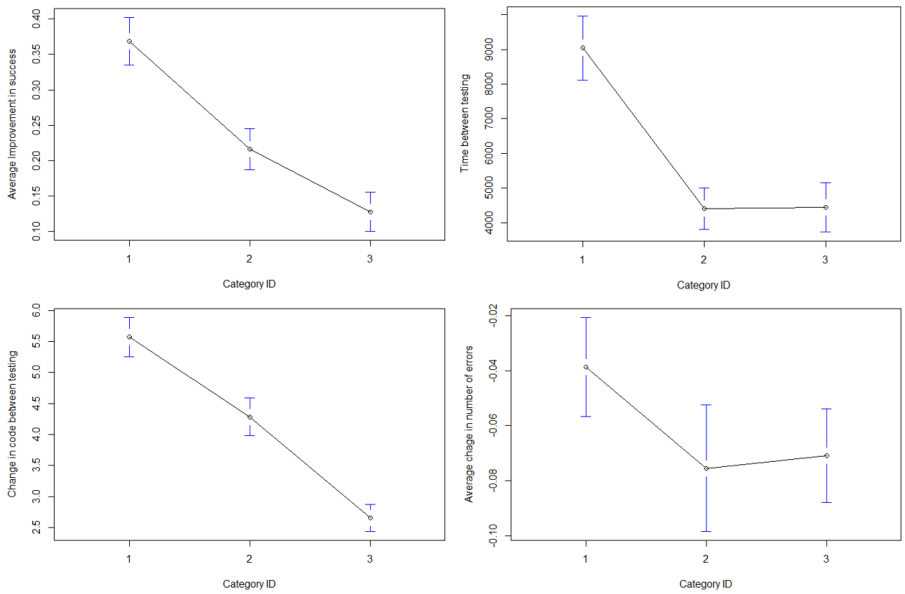

Fig. 2. Caption

Figure 3 shows the explanatory variables for the three categories as the progress based on the number of test runs. What is evident from the Figure 3 (left panels) is a clear difference in the time between two student program method launches and the average improvement between the intellects (shown with red) and the other two categories for the attempts 5–15 (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 (Tables 6,7,8), we observe that the behaviour 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 tests with main methods and the change in code. That means 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 tests using the main method. That means the thinkers take time to test, but nothing clearly can be said about the other parameters. Finally, the probers have change in code as a negative and significant coefficient, meaning that they make smaller changes to the code between two unit tests.

Finally, we can say that it is expected students to belong to more than one category while attempting to solve the programming assignments. Figure 4 shows how the students move between intellects, thinkers and probers for different assignments. For example, one can observe that intellects are a larger

group (233) than the thinkers (54) and probers (23) for the assignment 5 ($a5$); for the next assignment (i.e., $a6$) we see that 66 students did not attempt to solve $a6$; similar to $a5$, the largest category is intellect followed by thinkers and then probers, also the largest part of intellects did not change; most of the thinkers and probers either stayed the same or they interchanged categories.

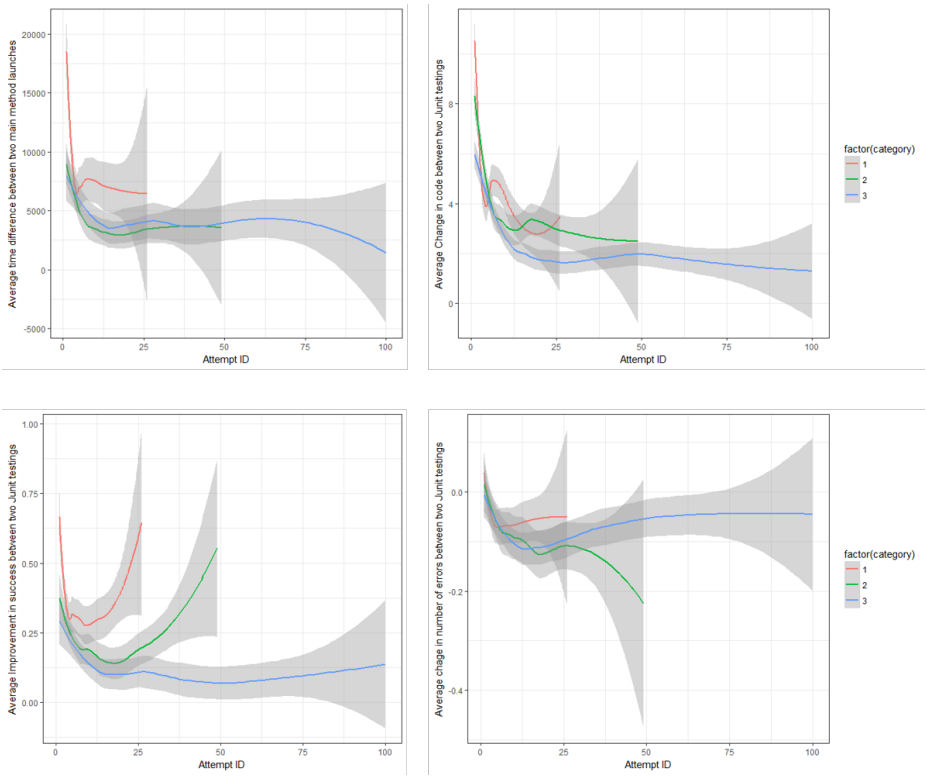

Fig. 3. Caption

Table 7. Linear model for improvement with all the exercises combined in one data set – thinkers

	Estimate	Std. error	t value	p value
intercept	1.688e-01	2.648e-02	6.36	.00001
Time diff launch	1.412e-06	6.433e-07	2.19	.02
Time diff edit	-7.058e-08	3.311e-07	-0.21	.83
Diff size	-1.277e-03	2.420e-03	-0.52	.59
Diff error	-7.698e-04	3.169e-02	-0.02	.98
Diff warning	4.564e-02	8.672e-02	0.52	.59

Table 8. Linear model for improvement with all the exercises combined in one data set – probers

	Estimate	Std. error	t value	p value
intercept	8.504e-02	2.293e-02	3.70	.0002
Time diff launch	1.510e-06	4.822e-07	3.13	.001
Time diff edit	-4.872e-08	3.328e-07	-0.14	.88
Diff size	-6.289e-03	3.033e-03	2.07	.03
Diff error	-5.213e-02	3.989e-02	-1.30	.19
Diff warning	-1.501e-01	7.507e-02	-2.00	.04

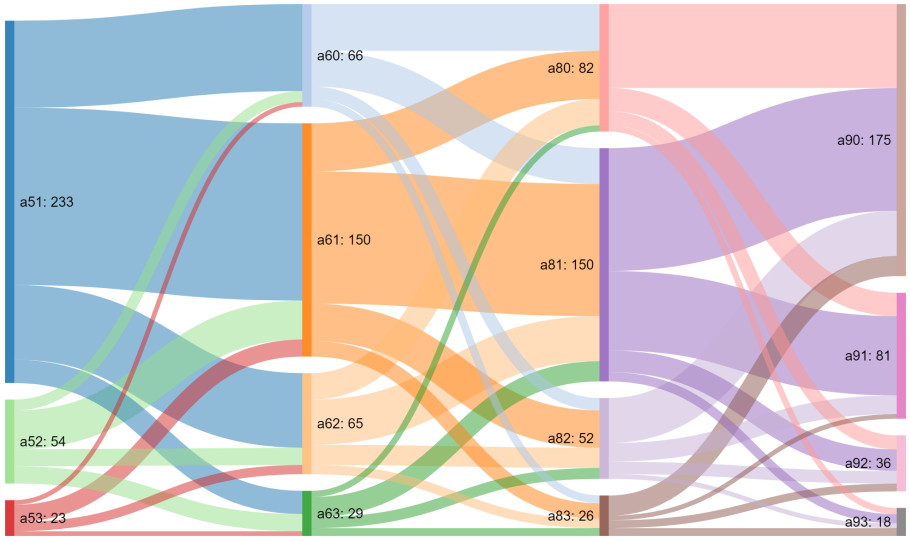

Fig. 4. Caption

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. The results from the performed analyses supported our two assumptions, 1) there are different programming strategies that lead students to success when they attempt to solve coding exercises, and 2) we can early identify struggling students. 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 (from the 4th attempt) to predict improvement in unit test success at a low granularity level of one student one assignment. Our focus on semantic-less-ness lead to better reproducibility and generalizability of the results, because we can not, at lest with the current state-of-art, know without explicitly asking students if they are having troubles with the coding constructs (e.g. loops or recursion) or they are having difficulty 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 done good job with predictive models that either looked at the students' level as a whole class, or focused only into code-based variables [4, 29, 42]. In addition, none of the previous studies managed to do an 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 attempt score; the highest improvement in unit test success; the lowest total number of attempts among the three categories; the longest wait time between two student program method launches; and finally, most changes in the code between two JUnit tests. The *thinkers* characteristics are: low first attempt score; low waiting time between two student program method launches; lower number of changes in code than those of the intellects but higher than those of the probers; and not a higher improvement in unit test success than the intellects but also not a lower than the probers. Finally, the *probers* as a group are characterized by having low first attempt score; low waiting time between two student program method launches; least changes to code 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 amount of time. The thikers appear to have a strategy to fix the errors and the bugs in the code, while the probers appear to have a trial and error approach to tackle the same problem. This is also evident from the Figure 3 (bottom-left), where we can see that for a big set of attempts, the probers have a slow growth (close to 0.25, that is, 4 attempts for passing one unit test); where as, after certain attempts students from the other two categories require one or two attempts to pass one unit test. This exponential improvement can be seen earlier for the intellects than for the thinkers, meaning that intellects make fewer mistakes to begin with and hence they need fewer attempts to pass all of the unit tests. However, thinkers show more regulated and informed behaviour of testing the code than probers, and this might be a plausible reason why probers require more attempts 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 make them more likely to use a trial and error approach [28].

Finally, the prediction results presented in this study could support educators in providing motivational feedback, that might be an incentive to students to test the code a few more times before they give up. For example, we can predict the number of attempts a student would carryout at an early stage and we can also predict the improvement in the unit test success at each attempt. Given the current attemptID and unit test score of the student, we could provide him/her with a target number of attempts at his/her given pace of improvement which might motivate the student to change the strategy (from probing to thinking) or to keep testing the code (if he/she is too close to the target attempt number).

5.1 Limitations and Future work Kshitij

The approach in this study carries a few limitations that we plan to overcome in the next studies. First of all, this is a black box approach because we do not look at the actual code at all. We look into the students behavioral patterns when they solve coding tasks. In future, we plan to analyze the mistakes made by the students and observe what category makes what kind of mistakes. Next, we also did not consider any semantic features computed from the code; however adding features from abstract syntax tree into the analysis could improve the prediction results. Finally, we do not have any additional information about students personality traits (e.g. consciousness, SRL) or their motivation during the course, which hinders us into providing personalized feedback yet. Thus, we plan to incorporate this information in the next study to be able to provide not just timely and actionable feedback as we can now, 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. Journal of experimental child psychology **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. Journal of Applied Psychology **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. Journal of the Learning Sciences **23**(4), 561–599 (2014)
5. Booth, P., Luckett, P., Mladenovic, R.: The quality of learning in accounting education: the impact of approaches to learning on academic performance. Accounting Education **8**(4), 277–300 (1999)
6. 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. Transforming IT education: Promoting a culture of excellence pp. 301–325 (2006)
7. Busato, V.V., Prins, F.J., Elshout, J.J., Hamaker, C.: Intellectual ability, learning style, personality, achievement motivation and academic success of psychology stu-

dents in higher education. Personality and Individual differences **29**(6), 1057–1068 (2000)
8. Cano, F.: Epistemological beliefs and approaches to learning: Their change through secondary school and their influence on academic performance. British Journal of Educational Psychology **75**(2), 203–221 (2005)
9. Chamorro-Premuzic, T., Furnham, A.: Personality traits and academic examination performance. European journal of Personality **17**(3), 237–250 (2003)
10. Chamorro-Premuzic, T., Furnham, A.: Personality, intelligence and approaches to learning as predictors of academic performance. Personality and individual differences **44**(7), 1596–1603 (2008)
11. Cooper, S., Cassel, L., Moskal, B., Cunningham, S.: Outcomes-based computer science education. In: ACM SIGCSE Bulletin. vol. 37, pp. 260–261. ACM (2005)
12. Corno, L., Mandinach, E.B.: The role of cognitive engagement in classroom learning and motivation. Educational psychologist **18**(2), 88–108 (1983)
13. Corno, L., Rohrkemper, M.: The intrinsic motivation to learn in classrooms. Research on motivation in education **2**, 53–90 (1985)
14. Digman, J.M.: Five robust trait dimensions: Development, stability, and utility. Journal of Personality **57**(2), 195–214 (1989)
15. Diseth, Å.: Self-efficacy, goal orientations and learning strategies as mediators between preceding and subsequent academic achievement. Learning and Individual Differences **21**(2), 191–195 (2011)
16. Edwards, S.H., Perez-Quinones, M.A.: Web-cat: automatically grading programming assignments. In: ACM SIGCSE Bulletin. vol. 40, pp. 328–328. ACM (2008)
17. Felder, R.M., Silverman, L.K., et al.: Learning and teaching styles in engineering education. Engineering education **78**(7), 674–681 (1988)
18. Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., Zander, C.: Debugging from the student perspective. IEEE Transactions on Education **53**(3), 390–396 (2010)
19. Hosseini, R., Vihavainen, A., Brusilovsky, P.: Exploring problem solving paths in a java programming course (2014)
20. 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)
21. Kiesmüller, U.: Diagnosing learners problem-solving strategies using learning environments with algorithmic problems in secondary education. ACM Transactions on Computing Education (TOCE) **9**(3), 17 (2009)
22. Koenemann, J., Robertson, S.P.: Expert problem solving strategies for program comprehension. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 125–130. ACM (1991)
23. 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)
24. Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., 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)
25. 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. Computers in Human Behavior **80**, 179–196 (2018)

26. 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 (AIEDCS 2013). p. 40 (2013)
27. Nussbaumer, A., Hillemann, E.C., Gütl, C., Albert, D.: A competence-based service for supporting self-regulated learning in virtual environments. Journal of Learning Analytics **2**(1), 101–133 (2015)
28. Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., Simmons, R.: Conditions of learning in novice programmers. Journal of Educational Computing Research **2**(1), 37–55 (1986)
29. 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)
30. Pillay, N., Jugoo, V.R.: An investigation into student characteristics affecting novice programming performance. ACM SIGCSE Bulletin **37**(4), 107–110 (2005)
31. Pintrich, P.R.: A conceptual framework for assessing motivation and self-regulated learning in college students. Educational psychology review **16**(4), 385–407 (2004)
32. Poropat, A.E.: A meta-analysis of the five-factor model of personality and academic performance. Psychological bulletin **135**(2), 322 (2009)
33. 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 (AIEDCS 2013). vol. 50 (2013)
34. Rodriguez, C.M.: The impact of academic self-concept, expectations and the choice of learning strategy on academic achievement: the case of business students. Higher Education Research & Development **28**(5), 523–539 (2009)
35. Roll, I., Aleven, V., McLaren, B.M., Koedinger, K.R.: Metacognitive practice makes perfect: Improving students self-assessment skills with an intelligent tutoring system. In: International Conference on Artificial Intelligence in Education. pp. 288–295. Springer (2011)
36. 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)
37. Soloway, E., Bonar, J., Ehrlich, K.: Cognitive strategies and looping constructs: An empirical study. Communications of the ACM **26**(11), 853–860 (1983)
38. Soloway, E., Ehrlich, K.: Empirical studies of programming knowledge. In: Readings in artificial intelligence and software engineering, pp. 507–521. Elsevier (1986)
39. 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. Personality and Individual Differences **120**, 238–245 (2018)
40. Turkle, S., Papert, S.: Epistemological pluralism and the revaluation of the concrete. Journal of Mathematical Behavior **11**(1), 3–33 (1992)
41. 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)
42. 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 (ITS 2006). pp. 13–20 (2006)

43. 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)
44. 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)