Received: 14 August 2022 | Revised: 12 January 2023 | Accepted: 16 February 2023

DOI: 10.1002/cae.22622

RESEARCH ARTICLE

WILEY

Utilizing programming traces to explore and model the dimensions of novices' code-writing skill

Yingbin Zhang[1] | **Luc Paquette**[2] | **Juan D. Pinto**[2] | **Aysa Xuemo Fan**[2]

[1]Institute of Artificial Intelligence in Education, South China Normal University, Guangzhou, China

[2]Department of Curriculum & Instruction, University of Illinois at Urbana-Champaign, Champaign, Illinois, USA

Correspondence
Luc Paquette, Department of Curriculum & Instruction, University of Illinois at Urbana-Champaign, 383 Education Bldg, 1310 S Sixth St, Champaign, IL 61820, USA.
Email: lpaq@illinois.edu

Funding information
China Scholarship Council, Grant/Award Number: 201806040180; National Science Foundation, Grant/Award Number: DRL-1942962

Abstract

Studies have found that most novice programmers have low proficiency in writing code. However, it is unclear what subskills compose code writing and which subskills novice programmers struggle with. This study utilizes programming traces to identify latent subskills that constitute code writing so that teachers can offer specific instruction on the weak subskills. Data were collected from an undergraduate course teaching introductory computer science in Java. Six hundred and fourteen students made submissions to homework programming questions in a web-based learning system. Based on the submission traces, we computed 11 features related to correctness and time students spent on their submissions. We conducted an exploratory factor analysis on two-thirds of students selected randomly and identified four factors. The first factor, code style proficiency, was mainly related to code style errors. The second, syntactic proficiency, concerned compiler errors. The third is semantic proficiency, which concerns runtime and logic errors. The fourth, syntactic debugging proficiency, concerned the success rate and time required for fixing compiler and code style errors. A confirmatory factor analysis conducted on the remaining one-third of the data supported the four-factor structure. The factor model showed measurement invariance between the data set where the model was developed and two new datasets, one from the same sample but collected at a different time point and another from a different sample and context (onsite course vs. online course). The factors were related to prior programming abilities, programming language familiarity, and future exam performance. These associations provided validity evidence for the factor model.

KEYWORDS
code writing, CS1, factor analysis, novice programmer, programming trace analysis

1 | INTRODUCTION

An early multinational study (conducted in 2001) found that most undergraduates in introductory computer science (CS1) courses did not know how to program by the end of the course [41] and raised concerns about the students' code-writing skills. While a subsequent repeat study (conducted in 2013) found that students' performance at the end of a CS1 course matched teachers' expectations better than in the 2001 study, the reason might be that teachers lowered their expectations [62]. Even in this 2013 study, many students still had difficulty in programming tasks by the end of their first CS1 course. Many studies have investigated explanations for this incapacity and found that students were weak at the prerequisites of code writing, such as code tracing and reading [37, 39, 42, 63]. Correspondingly, researchers have proposed instructions, particularly on the prerequisites and the writing skill itself [43, 71]. However, investigations on what skills novice programmers' code writing consists of are still scarce. Such effort is valuable because it enhances our understanding of the code-writing process of novice programmers, which in turn could lead to the design of more targeted instruction to address weaknesses in specific skills.

Programming traces contain detailed programming problem-solving information, such as edits on the submitted code, errors generated by the submitted code, and timestamp of submissions. That is, they capture the process through which students develop the final solution [64]. Thus, programming trace analysis can contribute to the understanding of programming behaviors and learning and has attracted increasing interest [7, 9, 30, 64, 65]. Nevertheless, there is limited work on linking programming trace analysis and novices' code-writing skills. This study aims to address this gap.

Specifically, this study asks the question: how can programming traces bring insights into the dimensions of novices' code writing? To answer this question, we extracted features related to code-writing from undergraduates' programming trace data in a CS1 course and conducted factor analyses to discover the underlying factors accounting for these features. We regarded these factors as code-writing skills and linked them with existing frameworks of programming skills. We further examined the relationships between these factors and students' prior experiences as well as exam performance to check the validity of the factors. Overall, this study investigated four research questions:

- Research question 1: What latent factor model can we identify underlying the features?
- Research question 2: What code-writing skills do the factors represent?
- Research question 3: Do the factors relate to exam performance?
- Research question 4: Do students' prior experiences relate to the factors?

2 | LITERATURE REVIEW

2.1 | Novice programming skills

Successfully completing programming activities requires both declarative knowledge and procedural skills [71]. Declarative knowledge in programming may include a basic understanding of programming constructs [60] and programming language syntax [20, 56]. Procedural skills enable individuals to apply their declarative knowledge to solve problems [19]. Studies have identified three distinct but related skills in novice programmers: code tracing, explaining (or reading), and writing [16, 37, 41–43, 51]. Tracing is the skill of manually compiling and executing a program and forecasting state changes and outputs [15, 43]. Explaining refers to the ability to explain the function or purpose of a program in plain language [42]. Writing is the ability to generate a program to perform a task [51]. Earlier work has suggested that there is a hierarchy among these procedural skills [39, 59]. For instance, the tracing skill is necessary for reading code correctly [17], and both tracing and explaining skills are strong predictors of performance on code writing tasks [39, 42, 63]. However, a recent study argued against this hierarchy and that existent evidence could only support that these procedural skills are related [24].

Many studies have focused on supporting novice programmers' learning of code-writing skills, but it is unclear what constitutes code writing [18]. Can it be decomposed into subskills, and what are they? Xie et al. [71] proposed a framework that decomposes reading and writing across the dimensions of semantics and template. Writing semantics is the skill of translating an unambiguous description to code, while writing a template is the skill of identifying the objective of an ambiguous problem description and creating a plan that uses a template (reusable abstraction of programming knowledge) to solve the problem. However, the framework was developed from the perspective of instruction and may not align with the structure of code writing produced by novice programmers.

Researchers have used Bloom's taxonomy [10] and the Structure of Observed Learning Outcomes (SOLO) taxonomy [8] to understand programming proficiencies [34, 61, 71]. The current study focuses on the SOLO

taxonomy because it has been widely used for evaluating the responses to problems requiring different programming skills [16, 38, 68]. The SOLO taxonomy typically classifies responses to a programming problem into four sequential levels [71]: (1) prestructural responses lack understanding of the problem or show knowledge unrelated to the problem; (2) unistructural responses provide a description for a small part of the code; (3) multistructural responses provide a line by line description of most of the code; (4) relational responses summarize the code in terms of its purpose. A general agreement is that solving an unfamiliar problem requires relational responses, but there is disagreement on responses for more specific problems [34]. One primary reason may be that the level of responses to a problem relies on students' prior knowledge and abilities [45]. The more advanced prior knowledge and abilities are, the lower levels the response may be. Another possible reason is that the responses to problems for the same skill may fall into different levels if the exact subskills of these problems differ. For example, writing semantics is at the unistructural or multistructural level, but writing template falls into the relational level [71].

2.2 | Programming trace analysis

2.2.1 | Understanding and supporting learning

Programming traces can provide fine-grained information about how problem-solving and learning processes unfold [7, 9, 12] and help build personalized and adaptive programming learning environments [50]. For example, Berland et al. [7] presented high school students with a task that required them to program virtual robots to play soccer. The programming environment logged each of the students' changes to their programs. Based on the programming traces, they investigated how students' program states evolved over time. The results showed that, on average, program states experienced three phases: exploration, tinkering, and refinement. Throughout these phases, the quality of programs gradually increased. Overall, Berland's study revealed how programming trace analysis could inform us about novice programmers' progress toward the final solution and the role of tinkering in their learning.

Similarly, Blikstein et al. [9] analyzed students' assignment submission traces in a programming methodology course. They found that across assignments, the average size and distribution of code updates (the number of lines added, deleted, and modified) were weakly related to course performance. However, the change in the code update patterns of two assignments strongly predicted course performance. The researchers further honed in on the first assignment and found three meaningful programming pathways, which had better power in predicting midterm grades than students' scores on the first assignment.

2.2.2 | Exploring programming behaviors and predicting course performance

Researchers have used programming traces to explore programming behaviors and predict future course performance. Two well-known metrics based on programming traces are the error quotient (EQ) [32] and the Watwin Score [66]. EQ characterizes the extent to which a student struggles with syntax errors while solving a programming problem [32]. It is computed via four steps: (1) generate pairs of consecutive compilation events on a problem; (2) assign a score to each pair based on an algorithm; (3) divide the score by 11, which is the maximum possible value (the maximum value is nine in a more complex version) [31]; (4) compute the average of the normalized scores of all pairs. The algorithm in step two penalizes the student if they repeat the same errors consecutively. A pair of compilation events gets a score of zero if at least one event does not end with any error, 8 if the two events have different types of errors, and 11 if both events have the same error type. Thus, a student with a high EQ struggles with the problem more than a student with a low EQ. Jadud [32] and Tabanao et al. [57] have found that EQ explained a substantial proportion of exam score variances (25%–29%).

The Watwin score is an extension of the EQ [66]. It can be calculated by the beforementioned procedure with a different algorithm in step two. This algorithm penalizes students based on the amount of time between compilation events, in addition to consecutive compiler errors. Watwin score has been found to have predictive power on course grades higher than EQ [66, 67].

Both EQ and the Watwin score focus on students' compilation behaviors and do not consider how students handle the semantic errors of a program [13]. The Normalized Programing State Model (NPSM) addresses this limitation by modeling students' programming activities in terms of changes in both syntactic and semantic correctness [13]. It divides students' program states into 11 categories based on syntactic and semantic correctness. For example, a program with unknown semantic correctness might be in a state that was edited to be syntactically correct by the student. This state might switch to a state of execution outside or inside the debug

mode. The results showed that NPSM had a better predictive power on assignment scores and final course grades than EQ and the Watwin score. For a detailed overview of these and other related metrics, see Villamor [64].

From among these different approaches, this study is closer to NPSM because it considers both syntactic and semantic correctness. It differentiates from NPSM, as well as EQ and the Watwin score, by focusing on the underlying structure among metrics that characterize programming trajectories rather than on finding metrics that have the best predictive power on grades. Our assumption is that these metrics are the manifestation of students' programming proficiencies. Factors that account for the covariances among these metrics may represent programming proficiencies, and different factors represent different subskills. It is critical to disentangle various subskills and identify those where a student is weakest. Such understanding can help to build better learner models and provides insights for personalized intervention. This effort, together with developing accurate prediction models, may move us toward closing the loop of learning analytics [70].

2.2.3 | Programming trace features

Programming trace analysis generally utilizes three types of features. The first type of feature aims to capture the characteristics of individual computer programs. Examples are the number of variables, loops, uncommented lines of code [58], the cyclomatic or McCabe complexity, and the programming constructs used by a program [27]. The second category quantifies the change, similarity, or dissimilarity between two programs. Examples are the number of added, deleted, and modified lines [9], the bag of word differences [9], and the edit distance between the abstract syntax trees (AST) of the two programs [50]. The third category provides aggregated information about the programming process. Examples are the number of submissions [46], the time on the task [46], and the total number of compiler errors made on a problem [5]. This category also includes such metrics as EQ [32] and NPSM [13]. This study mainly uses the second and third types and aggregates them over problems because the analysis was done at the student level.

It is noteworthy that many programming trace features may be related to the code writing skill, but some are not suitable for revealing the dimensions of this skill. Some features, especially those designed to capture characteristics of individual computer programs, are not applicable to all problems. For instance, some programming tasks do not involve loops, and the number of loops is not appropriate in these tasks. Similarly, cyclomatic complexity is not useful in programming tasks without complex control flow. Some features are too coarse to bring insights into the dimensions of code writing, for example, the number of submissions and the total time on a task. This study identified 12 features that might be proper for investigating the dimensions of code writing. We detailed the features in Section 3.3.

3 | METHODS

3.1 | Participants

Participants were undergraduates in a CS1 course at a public university in the United States during the Fall 2019 semesters. This study only included those who completed the course, finished more than half of the homework problems, and approved the use of their data for research purposes. In total, 612 students were included, among which 30.08% were female, and 81.36% were freshmen. A total of 16.53% of students majored in CS, 27.39% were in a program that combined CS and another discipline (e.g., CS and mathematics), and the others were in a program unrelated to CS. Participants completed a survey about their CS experiences before the course started.

3.2 | The course, learning system, and data

The course taught basic computing concepts and techniques in Java over a period of 16 weeks. Coursework included 69 homework problems, 12 weekly quizzes, and 3 exams. All coursework was required and counted toward the course grade. One of the 69 homework problems was released every weekday on most days of the semester. Each problem presented a programming task that required students to write code to solve and was estimated to take 10–15 min to complete. Topics taught included: (before exam one) variables, conditionals, loops, arrays, functions, and strings; (between exams one and two) objects, polymorphism, references, and interfaces, (before exam three) algorithm analysis, lists, trees, recursion, sorting, internet, exceptions, and hashing after exam two.

The coursework was hosted on a web-based, problem-driven learning system. Students used the system to submit solutions to homework problems, quizzes, and exams. Students could submit solutions to homework problems as many times as they wanted until the deadline. The system automatically graded a

submission based on a set of tests and generated feedback about mistakes. First, the learning system checked whether the style of the submitted code met the course requirement and whether the code could be compiled. Code that did not match the required style had check style errors (e.g., an operator was not surrounded by whitespace, the submission had incorrect indentation, etc.), and code that could not be compiled had compiler errors. If the submitted code had check style or compiler errors, the system delivered a message containing the errors and corresponding lines of code where the errors occurred.

If there were neither check style nor compiler errors, the system compiled the solution and ran problem-specific tests to check whether the code fulfilled the problem's requirements. For example, if the problem asked students to write a program to compute the mean of three numbers, the test could randomly generate three numbers, use the generated numbers as input for the code, and examine the equality between the mean and the output of the code. This step might repeat multiple times to avoid coincident equality between the mean and the output. If the code fulfilled the requirements, it was correct, and students obtained full credits for this problem. Otherwise, the code contained test errors, and the system would display a message related to the first encountered test error.

Test errors were a mix of runtime errors (the system could compile but not run the solution) and logic errors (the system could run the solution, but the solution did not generate the expected output) [28, 40]. This study did not distinguish runtime and logic errors because, from the students' perspective, the presentation of the error feedback looked the same. Unlike check style and compiler error feedback, runtime and logic error messages could not indicate the error line because they were rarely caused by a specific line of code. In addition, prior studies have investigated runtime and logic errors together [3, 13, 22], possibly because they are more general than check style and compiler errors and relatively independent of language [44].

The learning system automatically recorded the information of each submission, including the code, date, correctness, and error feedback. These submission traces were stored in a secure database outside the learning system. This study only utilized submission traces on homework programming problems because these problems were closer to real-world programming tasks, where individuals can manage progress at their own pace and use any necessary resources [36]. Since students' code-writing skills were expected to develop during the course, it was decided that aggregating homework submissions over the complete duration of the course would be unreasonable. Thus, we split the homework submissions into three datasets: before exam one, between exams one and two, and after exam two. Section 3.5 described the detail of how we used these datasets. Table 1 presents the characteristics of each data set.

3.3 | Code writing-related features

We utilized the homework submission traces to compute features related to code writing for factor analyses. The error feedback generated by the learning system provided rich information about errors in each submission. We used this error feedback to extract the number and type of errors in each submission. We further compared consecutive submissions on the same problem to compute features that measured changes in errors, such as the number of syntax errors present in one submission

TABLE 1 The characteristics of submission trace datasets.

Data set	Before exam one	Between exams one and two	After exam two
# problems	22	23	24
# submissions	81,070	94,863	105,219
% submissions with at least one checkstyle error	28.10	16.19	16.16
% submissions with at least one syntax errors	39.00	51.42	37.70
% submissions with at least one test errors	24.08	27.18	42.32
Average # submissions per problem per student	5.15	7.28	8.36
Average # checkstyle errors per problem per student	1.59	0.95	1.33
Average # syntax errors per problem per student	2.98	5.68	5.59
Average # test errors per problem per student	0.48	0.63	0.97

Note: #, number of; %, the proportion of.

that were not present in the next consecutive submission. These features were then aggregated across submissions on the same problem. We computed 12 problem-level features, described below.

(1–3) The number of each type of error: the total number of checkstyle errors, compiler errors, and test errors students made in a problem. The number of errors has previously been used as a negative indicator of learning and proficiency [5, 6, 71].

(4–6) The time required for fixing each type of error: The time required for fixing check style errors is the average of the time between the submission where a check style error appeared and the submission where the error was fixed. The same operationalization applied to the time for fixing syntax errors and for fixing test errors. The time spent fixing errors may reflect the extent to which students struggled with those errors [66]. A student might solve an error with multiple submissions across sessions, and the sum of the time across sessions was used as the time for fixing a single error.

(7–9) The failure rate for fixing each type of error: Studies have considered consecutive submissions with the same errors as an important indicator of struggling [32, 66]. The failure rate for fixing check style errors was calculated as the proportion of pairs of successive submissions where all check style errors in the first submission also existed in the second submission among all pairs where the first submission had check style errors. The failure rates for fixing compiler errors and fixing test errors follow the same operation. Note that a submission might have both checkstyle errors and compiler errors, and students might decide to focus on one type of error for a given submission, for example, fixing the check style errors and ignoring the compiler errors. Such pairs of consecutive submissions were not counted when computing the failure rate for fixing compiler errors. This was the same as the failure rate for fixing check style errors, that is, the pairs where students only fixed the compiler errors and did not address the check style errors in the next submission were not counted.

(10–11) The rate of making new errors: The rate of making new check style errors is the proportion of pairs of consecutive submissions where a check style error did not exist in the first submission but appeared in the second submission. The rate of making new compiler errors follows the same operation. Code changes that cause new errors are known as fix-inducing changes [54]. Fix-inducing changes may be the result of quick attempts to fix errors without a deep understanding of the problem and the current code. Indeed, experienced programmers are less likely to make fix-inducing changes [21, 48]. We did not compute the rate of making new test errors because the learning system only showed one test error at a time. When a test error did not appear in the first submission of a pair but appeared in the second, this could mean that either this was a new error for the student or that the error already existed in the first submission, but the system did not present it to the student. We could not distinguish between the two situations.

(12) The number of uncommented lines of code in the final submission: We assumed that students with good code-writing skills would write a few unnecessary lines of code while solving a problem.

Problem difficulties varied, and thus, the raw features were not directly comparable across problems. We used the median of a feature on a problem as an approximation of the problem's difficulty regarding the feature and used the difference between the median and the feature's value to make it more comparable across problems. Features were then aggregated at the student level by computing the mean of each feature over problems completed by the student. We used the student-level features for subsequent analyses.

Note that most of the features were related to submissions with errors, but we did not discard correct submissions. Features 4–12 used all submissions. For example, if a submission with a test error was followed by a correct submission, it meant that the second submission fixed the test error in the first submission. This pair of submissions was used in the computation of the failure rate for fixing test errors.

3.4 | External criteria

We used exam scores, self-rated programming abilities, and self-reported programming language familiarity to examine the criterion validity of the factors suggested by factor analysis. Our rationale was that if the factors capture students' code-writing skills, they should be related to these criteria. We did not use the course grade because students' performance on homework problems was a part of the course grade. Thus, it would be difficult to interpret the association between the factors and course grades because both variables were related to homework performance.

There were three exams in the course, which occurred in the 5th, 10th, and 16th weeks, respectively.

An exam might involve anything covered up to the time, with emphasis on the topics covered since the last exam. The exam contained multiple-choice and small programming problems. All problems were automatically graded. The multiple-choice questions allowed one or two attempts, while the programming problems allowed unlimited attempts. Students had 1 h to complete an exam. The maximum number of points possible was 100, with programming problems accounting for over half of the points. The average scores of exams one, two, and three were 84.89 (SD = 13.88), 84.05 (SD = 18.61), and 84.15 (SD = 17.39), respectively.

The survey conducted before the beginning of the course asked students about their programming experiences. One survey question asked students to rate their current programming abilities on a five-point scale, with five representing the highest level. The proportions of students in levels 1–5 were 11.51%, 32.90%, 38.57%, 12.97%, and 4.05%, respectively.

Another question asked students which programming languages they were already familiar with before taking the course. Students could select "I've never programmed before!" or one or more of the following options: C, C#, C++, Java, JavaScript, MatLab, PHP, Python, and Swift. We coded students' responses to this question into four categories: (1) *none* (9.89%), students selected "I've never programmed before!"; (2) *Java* (17.67%), students only selected Java; (3) *others* (22.53%), students selected languages other than Java; (4) *Java and others* (49.92%), students selected Java and at least one other language. We distinguished Java from other languages because the course specifically taught Java. We distinguished *Java* from *Java and others* because students familiar with Java and at least one other language might have more programming experience and better code-writing skills than students only familiar with Java.

3.5 | Analyses

As aforementioned, we split the homework submissions into three datasets by submission dates because students' code-writing skills were expected to develop during the course. We conducted factor analysis only in the submission data set after exam two to develop a factor model (see Figure 1). We used this data set rather than the data set before exam one because of the concern that students might be unfamiliar with the learning system at the beginning of the course. This lack of familiarity might influence their behaviors in the system and the code writing-related features, which in turn influenced the factor analysis. However, after obtaining the factor model, we conducted a measurement invariance test to examine whether the factor model holds for the submission data set before exam one. Readers may wonder why we did apply a longitudinal factor analysis. We argue that the longitudinal analysis is not applicable to the data of this study and explain the reason in detail in the last paragraph of Section 5.2.

3.5.1 | Factor analyses

We applied factor analyses to code writing-related features to discover the underlying factors accounting for these features. We randomly selected two-thirds of the students for exploratory factor analysis (EFA) [4]. We implemented EFA with principal factor analysis and Promax rotation via the *fa()* function of the *psych* 2.1.6 library in R 4.1.1 [49]. We used promax rotation to allow correlations between the factors.

We then conducted confirmatory factor analysis (CFA) on the remaining one-third of students to test the factor model suggested by EFA. We implement CFA via the *cfa()* function of the *lavaan* 0.6-9 library in R 4.1.1

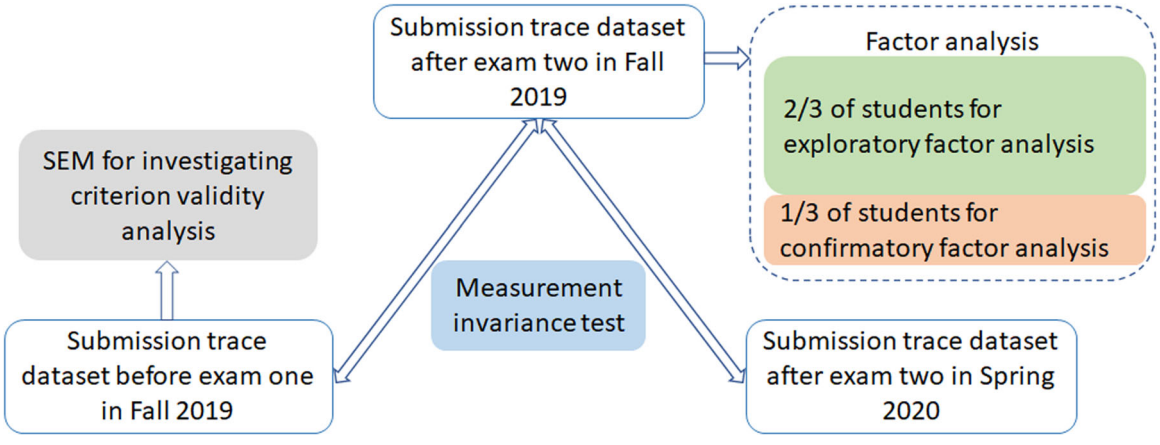


FIGURE 1 The datasets for different analyses.

[52]. For the goodness of fit, we considered the comparative fit index (CFI), Tucker–Lewis index (TLI), root mean square error of approximation (RMSEA), and standardized root mean square residuals (SRMR). CFI and TLI contrast the specified model with a model fitted terribly and suggest an acceptable model fit with values > 0.90 [29]. RMSEA represents the difference between the specified model and a model fitted reasonably and suggests an acceptable model fit with values < 0.08 [11]. SRMR is a normalized summary of the average covariance residuals and suggests a satisfactory fit with values near 0.08 [29]. The initial model fit was poor, so we modified the factor model until finding an acceptable fit based on the modification index [55].

3.5.2 | Measurement invariance test

Since the final CFA model was derived from the data set after exam two, it was unclear whether it would generalize to other datasets. To investigate this issue, we tested the measurement invariance of the final model between the datasets before exam one and after exam two (see Figure 1). Given that these two datasets were from the same students, we also examined the generalizability of the final model by conducting the measurement invariance test between the data set after exam two and a data set collected in the Spring 2020 semester (also after exam two). Note that in the Spring 2020 semester, the course switched to an entirely online course 2 weeks before exam two due to coronavirus disease 2019 (COVID-19). Thus, to some extent, the Spring 2020 data set was from a different sample in a different context. Typically, the measurement invariance test includes four sequential steps [47]: (1) the configural invariance model for the model structure equivalence; (2) the metric invariance model for factor loading equivalences; (3) the scalar invariance model for item intercept equivalences; and (4) the residual invariance model for item residual equivalences. Each model is more stringent than the last one. The difference in model fit indices between more stringent and less stringent can inform us whether the more stringent model is acceptable.

3.5.3 | Reliability and validity analysis

We examined whether the final factor model had acceptable reliability and validity. Composite reliability (CR), similar to coefficient alpha, evaluates the internal consistency among features, which is the ratio of the true score variance to the observed score variance [23]. A CR larger than 0.7 is acceptable [26]. Three types of validity were considered: convergent, discriminant, and criterion validity. Convergent validity combines factor loadings, their significance, and factors' average variance extracted (AVE) to assess the degree that features converge on a factor [25]. Good convergent validity is indicated by factor loadings and AVE larger than 0.5 with $p < .05$. Discriminant validity evaluates the degree to which the factors are distinct [23]. If the square root of each factor's AVE is larger than correlations between any factors, factors are discriminant.

Criterion validity is the extent to which a factor relates to a theoretically relevant variable, that is, a criterion [1]. This study assessed criterion validity via the associations between the factors and the aforementioned external criteria: exam scores, self-rated programming ability, and self-reported language familiarity. Given that students' code writing proficiency developed as the course went on, submission traces before exam one might preserve more differences in code writing proficiency caused by prior experiences than submission traces after exam one or two. Thus, we applied structural equation modeling (SEM) to the submission traces before exam one for the criterion validity analysis. Specifically, self-rated programming ability and self-reported language familiarity were predictors of the latent factors, which were predictors of exam scores. We implement SEM via the *sem()* function of the *lavaan* 0.6-9 library in R 4.1.1 [52].

4 | RESULTS AND DISCUSSION

4.1 | RQ1: What latent factor model can we identify underlying the features?

4.1.1 | EFA

We first examined whether the data were suitable for factor analysis using Bartlett's test of sphericity and the Kaiser–Meyer–Olkin (KMO) test [4]. Bartlett's test indicated sufficient correlations among the features ($\chi^2/df = 36.59$, $p < .001$). The overall KMO was 0.65, indicating that a relatively high proportion of variance in the features might be caused by underlying factors, and all single KMO values were greater than 0.55. Overall, these results suggested that EFA was applicable to the data. We determined the number of factors based on the parallel analysis, which suggested four factors (Figure 2). Thus, we conducted EFA with a four-factor solution.

The initial EFA returned a pattern matrix where the loadings of the number of uncommented lines of code on all factors were smaller than 0.30. Further investigation showed that the number of lines was weakly related to the other features (only one correlation greater than

FIGURE 2 Plot of parallel analysis.

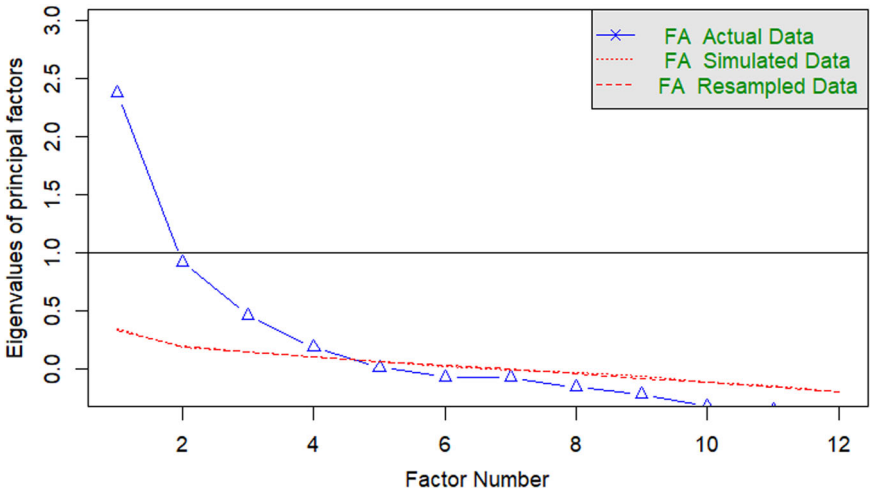


TABLE 2 Pattern matrix with four factors.

Factor	1	2	3	4
Number of checkstyle errors	**0.61**	0.11	0.10	−0.00
Making new checkstyle errors	**0.56**	0.00	0.00	0.00
Number of compiler errors	0.00	**0.98**	0.10	0.00
Making new compiler errors	0.00	**0.58**	0.00	0.10
Number of test errors	−0.15	**0.59**	0.16	−0.23
Time for solving test errors	0.11	0.00	**0.77**	0.00
The failure rate of fixing test error	−0.00	0.26	**0.65**	0.00
Time for solving checkstyle errors	0.00	0.00	0.12	**0.36**
Time for solving compiler errors	0.00	0.00	0.12	**0.55**
The failure rate of fixing checkstyle errors	0.13	0.00	−0.12	**0.35**
The failure rate of fixing compiler errors	0.00	0.26	−0.18	**0.62**
Proportion variation	0.07	0.16	0.10	0.09
Cumulative variation	0.07	0.23	0.33	0.43

0.30). Thus, this feature was discarded. The second EFA returned a pattern matrix where all features have a loading larger than 0.35 on one of the four factors (Table 2, in bold). Table 2 shows that each factor was strongly loaded by at least two features. We then conducted CFA to test the factor structure.

4.1.2 | CFA

We began CFA with the four-factor model suggested by Table 2. Each factor was loaded by the features whose loadings on the factor are in bold in Table 2. However, the fit indices indicated a poor fit (see Table 3). The modification index suggested allowing the number of test errors to load on factor 3. We revised the model accordingly and reconducted CFA. The model fit became acceptable (model 2). However, the number of test errors cross-loaded on factors 2 and 3. We wondered whether we could remove the loading of the number of test errors on factor 2 as this would make factor 2 more interpretable. However, this also greatly decreased the model's fit (model 3). The modification index suggested an error correlation between the number of test errors and compiler errors. After adding this correlation, the model fit became acceptable again. CFI and TLI reached 0.90. RMSEA and SRMR were 0.07. All factor loadings were significant ($p < .01$; see Figure 3) [1]. Thus, we selected this model as the final model.

Table 4 displays each factor's composite reliabilities, AVEs, and the square roots of AVEs. The composite reliabilities of the first three factors were larger than or close to 0.7, indicating acceptable internal consistency. Their AVEs were larger or equal to 0.5, indicating acceptable convergent validity. The square roots of their AVEs were larger than their correlations with other factors, suggesting good discriminant validity. For the fourth factor, its CR was 0.46, indicating poor internal consistency. Its AVE was 0.22, indicating poor convergent validity. The square root of its AVE was only larger than one of the correlations between it and other factors, indicating poor discriminant validity. We evaluated the criterion validity in Section 4.3.

[1]In Figure 2, the factor loading of the failure rate for fixing test errors was 1.002, slightly greater than 1. This does not suggest that the model is wrong. Factor loadings can be larger than one in models where factors are related since they represent regression coefficients rather than correlations [33].

4.1.3 | Measurement invariance

Table 5 presents the results of the measurement invariance test. Between the datasets before exam one and after exam two of Fall 2019, the configural model had an acceptable model fit, suggesting that the pattern of feature loadings was invariant across the two datasets. Thus, more stringent models were fitted to the data. The decreases in CFI and TLI as well as the increases in RMSEA and SRMR were smaller than the 0.01 cutoff [14, 53] between configural and metric models, metric and scalar models, and scalar and residual invariance models. Similarly, for the measurement invariance test between semesters, the fit statistics of configural model as well as the change in fit statistics were also acceptable. Overall, the results of the test between the two Fall 2019 datasets suggest that the factor model had measurement invariance over the course, while the results of the test between Fall 2019 and Spring 2020 datasets suggest that the factor model had measurement invariance between different samples and contexts.

4.2 | RQ2: What code-writing skills do the factors represent?

The first factor in the final model contained the number of check style errors and the rate of making new check style errors. Check style errors occurred when students' code did not match the desired style, such as putting whitespace between tokens and using the correct number of spaces for indentation. Thus, we labeled the first factor as code style proficiency. A correct code style usually required students to consider one line at a time because most style requirements were about a few tokens (e.g., whitespace between tokens). The correct indentation level required students to consider a few lines simultaneously, but these lines were a small part of the code.

TABLE 3 Fit indices for the four-factor models in CFA.

Model	Modification	CFI	TLI	RMSEA	SRMR
Initial model	-	0.84	0.77	0.11	0.08
Model 2	Let the number of test errors load on factor 3	0.92	0.88	0.08	0.08
Model 3	Let the number of test errors not load on factor 2	0.79	0.69	0.13	0.08
Final model	Add an error correlation between the number of test and compiler errors	0.93	0.90	0.07	0.07

Abbreviations: CFI, comparative fit index; RMSEA, root mean square error of approximation; TLI, Tucker–Lewis index; SRMR, standardized root mean square residual.

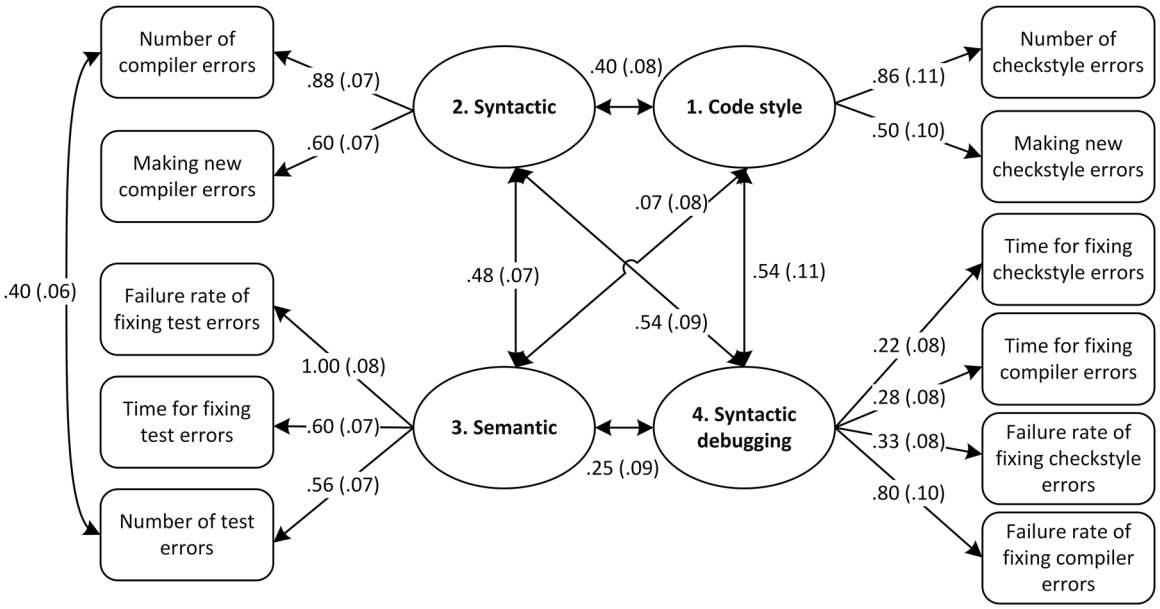


FIGURE 3 The final confirmatory factor analysis model. Single-headed arrows represent factor loadings, while double-headed arrows represent correlations. Values in the parentheses are standard errors. All coefficients are standardized and significant ($p < .01$).

TABLE 4 Composite reliability (CR), average variance extracted (AVE), and discriminant validity.

Factor	CR	AVE	Discriminant validity			
			Code style	Syntactic	Semantic	Syntactic debugging
1. Code style	0.65	0.50	(0.71)			
2. Syntactic	0.73	0.59	0.40	(0.77)		
3. Semantic	0.78	0.56	0.07	0.48	(0.75)	
4. Syntactic debugging	0.46	0.22	0.54	0.54	0.25	(0.47)

Note: Diagonal in parentheses: the square root of AVE. Off-diagonal: correlations between factors.

TABLE 5 Model fit statistics for the measurement invariance test.

Model	Datasets before exam one and after exam two of Fall 2019			Datasets after exam two of Fall 2019 and Spring 2020				
	CFI	TLI	RMSEA	SRMR	CFI	TLI	RMSEA	SRMR
Configural model	0.892	0.840	0.093	0.057	0.894	0.843	0.087	0.063
Metric model	0.884	0.842	0.092	0.063	0.892	0.853	0.084	0.065
Scalar model	0.886	0.857	0.088	0.063	0.895	0.868	0.080	0.065
Residual	0.878	0.864	0.085	0.064	0.898	0.886	0.074	0.064

Abbreviations: CFI, comparative fit index; RMSEA, root mean square error of approximation; TLI, Tucker–Lewis index; SRMR, standardized root mean square residual.

Thus, the response reflecting code style proficiency is at the unistructural level of the SOLO taxonomy [38, 68].

The number of compiler errors and the rate of making new compiler errors loaded on the second factor. A compiler error occurred when the code was syntactically incorrect. We expect that a student familiar with the Java syntax would make fewer compiler errors. Thus, we labeled the second factor as syntactic proficiency. Some compiler errors were as simple as code style errors, such as missing a ")" or ";". Students can avoid such errors by simply focusing on the current line. Other compiler errors were more complex, such as "constructor Parent in class Parent cannot be applied to given types." Students need to consider multiple lines and sometimes even most of the code in mind to avoid such errors. Thus, the response to a problem that requires syntactic proficiency may be at the unistructural and multistructural levels of the SOLO taxonomy [38, 68]. Syntactic proficiency may be related to the skill of writing semantics in Xie et al's framework of novice programming skills [71]. If there is a format requirement, code style proficiency may be also a part of this skill.

The number of test errors, the time for fixing test errors, and the failure rate for fixing test errors loaded on the third factor. Test errors occurred when the code was free of check style and compiler errors but could not run or generate the expected output given certain inputs. It meant that the code was syntactically correct but semantically incorrect, that is, failing to fulfill the problem requirements. The loadings of three test-error-relevant features suggest that this factor relates to students' ability to write semantically correct code to solve a problem. Thus, we labeled the third-factor semantic proficiency. Writing semantically correct code for a problem required students to identify the problem objective and create an unambiguous plan or pseudo code. Thus, the response that reflects high semantic proficiency is at the relational level of the SOLO taxonomy [38, 68]. Semantic proficiency may be related to the skill of writing templates in Xie et al's framework [71].

The failure rate for fixing compiler errors, the failure rate for fixing check style errors, the time for fixing compiler errors, and the time for fixing check style errors loaded on the fourth factor. The first two features concerned the accuracy of debugging compiler and check style errors, while the last two features concerned the speed of debugging. Compiler and check style errors relate to syntax and style. Thus, we labeled the fourth factor as syntactic debugging proficiency. Debugging requires

students to locate the error cause and repair the error [40]. The compiler message contained the line number of check style and compiler errors. For all check style errors and some compiler errors, the given line number provides direct information about where to find the error. In such cases, it is easy to locate the cause of these errors and repair them. However, for some compiler errors, locating the cause of the error requires students to understand multiple lines or even most of the code. Thus, high syntactic debugging proficiency may be at the multi-structural level of the SOLO taxonomy [38, 68]. Xie et al's framework does not explicitly consider debugging skills [71], but novices are generally weak in this skill [40]. How debugging skills develop along with other skills is an important gap awaiting further research.

4.3 | RQ3: Do the factors relate to exam performance?

We applied SEM to the submission traces before exam one of Fall 2019 to investigate the associations between the factors and students' exam performance as well as prior experiences (Section 4.4). Features loaded positively on the factors, but higher feature values meant worse performance, such as a high number of compiler and test errors. Thus, higher factor scores indicated lower proficiencies. To facilitate the interpretation of the relationships between these factors and exam performance as well as prior experiences, we inverted the sign of the coefficients in Tables 6 and 7 so that higher factor scores mean higher proficiencies. The factors positively predicted all exams and together explained 34%, 27%, and 16% of score variances in exams one, two, and three, respectively (see Table 6). The decrease in explained variances from exams one to three was expected because code writing factors were extracted from submission traces before exam one. Exams two and three were months later than the time that students made these submissions. Overall, these correlations provided validity evidence for the code-writing factors.

TABLE 6 The standardized coefficients of code writing factors on exam scores.

β (SE)	Exam 1	Exam 2	Exam 3
Code style	3.30 (0.98)***	1.99 (1.33)	4.66 (1.41)***
Syntactic	6.41 (0.68)***	6.82 (0.93)***	3.67 (0.90)***
Semantic	1.43 (0.79)	4.88 (1.15)***	3.45 (1.14)**
Syntactic debugging	4.84 (0.85)***	5.82 (1.19)***	6.16 (1.21)***
R^2	.34	.27	.17

**$p < .01$; ***$p < .001$.

4.4 | RQ4: Do students' prior experiences relate to the factors?

The self-rated programming abilities positively predicted code writing factors except for code style (see the bottom row of Table 7). The lack of correlation between code style and self-rated programming abilities was expected as the code style requirement was specific to the course. Students with high self-rated abilities might not have experience with this requirement. Overall, these correlations matched the definitions of the factors.

The language familiarity groups had no difference in code style proficiency. This is expected because students familiar with programming languages before the class might not be familiar with the code style requirement of

TABLE 7 The standardized coefficients of prior experiences on code writing factors.

β (SE)		Code style	Syntactic proficiency	Semantic proficiency	Syntactic debugging
Self-rated ability		0.02 (0.02)	0.21 (0.04)***	0.12 (0.04)**	0.23 (0.04)***
Reference group: None[a]	Others	0.06 (0.06)	0.60 (0.14)***	0.14 (0.13)	0.23 (0.13)
	Java	0.05 (0.06)	1.02 (0.15)***	0.37 (0.14)*	0.25 (0.14)
	Java and others	0.08 (0.07)	0.98 (0.15)***	0.35 (0.14)*	0.51 (0.14)***
Reference group: Others[a]	Java	0.00 (0.00)	0.43 (0.11)***	0.23 (0.10)	0.01 (0.10)
	Java and others	0.00 (0.02)	0.39 (0.09)***	0.21 (0.09)	0.27 (0.09)*
Reference group: Java[a]	Java and others	0.03 (0.03)	−0.04 (0.09)	−0.02 (0.09)	0.26 (0.09)*

Note: *, **, ***$p < .05, .01, .001$ after BY correction.

[a]Language familiarity was dummy coded, and the reference group was coded as 0. For example, the coefficient in the second cell of the code style column was 0.06, which meant that the *Others* group on average scored 0.06 higher than the *None* group in the code style factor. Three structural equation models were fitted, each time with a different language familiarity group as the reference. Redundant coefficients were not shown.

the course in this study. The group of *none* scored lower than the other groups in syntactic proficiency, and the group of *others* scored lower than groups of *Java* as well as *Java and others*, which in turn had no difference in syntactic proficiency. This result is also expected because students in this study used Java to solve programming problems. In terms of semantic proficiency, the group of *none* had no difference from the group of *others* but scored lower than groups of *Java* as well as *Java and others* in semantic proficiency. This difference is partially expected and suggests that semantic proficiency might capture some information about students' familiarity with Java. The group of *Java and others* scored higher in syntactic debugging than the other groups. This result matches expectations. The group *Java and others* might have more programming experience than the group of *Java*. In comparison with the group of *none* and *others*, the group of *Java and others* knew more about Java and might process Java-related errors more accurately and quickly.

Overall, the associations between code writing factors and exam scores as well as prior experience were in line with the factors' interpretation. These results support the definitions of the factors and suggest acceptable criterion validity.

5 | CONCLUSION

This study utilized programming traces and factor analysis to explore the latent factors of novice programmers' code-writing skills. We identified four factors, and the factor model showed measurement invariance between the data set from which the model was developed and two new datasets, one from the same sample but collected at a different time point and another from a different sample and context (onsite course vs. online course). Three of the four factors had acceptable composite, convergent, and discriminant validity. The four factors predicted the scores in exams months later and were related to self-rated programming abilities and language familiarity in a way matching the definitions of these factors, indicating acceptable criterion validity. We linked these factors to the SOLO taxonomy [8], a popular framework for evaluating novice programmers' responses in the field of CS education [16, 38, 68]. Code style and syntactic proficiencies may be related to the skill of writing semantics in a framework of novice programming skills [71], while semantic proficiency may be related to the skill of writing templates in the same framework. The framework lacks skills explicitly related to debugging, suggesting research opportunities.

5.1 | Limitations and future research

Although the syntactic debugging factor showed good criterion validity, it had poor internal consistency, CR, and discriminative validity. The fit of the four-factor model was acceptable but far from satisfactory. Overall, the final factor model has rooms for improvement.

The course for this study was taught in Java and hosted on a web-based, problem-driven learning system. The four-factor model may not generalize to other learning environments and languages. Another limitation is that students in this study might not be a representative sample of CS1 course students because 90.11% of students in this study reported familiarity with at least one programming language before taking the course. We argue that participants could still be considered novice programmers as even a 4-year CS program may only turn a student into a competent programmer [69]. However, we acknowledge that further research may be necessary to validate whether the model is applicable to a group lacking any prior programming experience.

The four factors may not fully capture all the skills involved in code writing as they explained only 43% of feature variances. The large proportion of unexplained variance suggests other underlying factors. The four-factor model did not account for the number of uncommented lines of code, a feature that was removed because of small loadings. In addition, the characteristics of the data set prevented us from computing more features. For instance, the existence of compiler errors prevented us from converting the code in each submission to ASTs and computing the AST edit distance between two submissions [50]. The data set did not contain individual student keystrokes or code edits, only code submissions, preventing us from computing more fine-grained features. With more features, more code-writing skills may be uncovered.

We selected the code-writing features based on prior studies and extracted them from students' programming traces in real-world programming tasks. Thus, the content validity of the features might be appropriate. Nevertheless, a formal evaluation of the content validity entails experts' judgment of the relationship between the features and the code-writing skill [1]. Future research may invite experts to formally evaluate the content validity of the code-writing skill model developed from programming trace analysis.

We were also unable to use our four-factor model to investigate how code-writing skills develop over time throughout the duration of the course. This is because the difficulties of homework programming problems gradually increased over the duration of the course, as

evidenced by Table 1, where students made more syntactic and test errors on a problem after exam two than before exam one. We deducted the median from the raw feature on a problem to account for its difficulty and to make features more comparable across problems. However, after such processing, feature values represented students' relative standings in the course, and so did the code writing factor scores. Thus, changes in a student's factor scores between different stages of the course (e.g., before exam one and after exam two) represented their change in relative standings rather than their development in code-writing skills. As such, the increase in some students' relative standings implies a decrease in others, even though it is possible that proficiency in code writing skills increases for all students. Other studies that want to investigate students' development based on programming traces may face the same limitation. As such, it is critical for further work to address this issue.

5.2 | Implications

Our factor analyses identified four latent factors that may represent code-writing skills. Most pairs of skills had weak to moderate associations (see Figure 3), indicating that they are quite distinctive. CS1 instructors may wish to differentiate their instruction based on students' varying levels of proficiency in these skills. For instance, the correlation between syntactic proficiency and syntactic debugging was 0.54, implying that some students might have a good understanding of Java syntax but not know how to debug Java syntactic errors. This is in line with prior research [2, 40] and suggests that CS instructors should explicitly teach novices debugging strategies. At the beginning of a course, it may be challenging to know which skills a student is weak in due to the scarcity of programming traces, and thus, it is difficult to adapt instructions directly to students' code-writing skills. In this case, instructors may differentiate instructions based on students' self-reported language familiarity and programming ability because these variables were related to code-writing skills. For example, students familiar with either Java or other programming languages showed lower syntactic debugging proficiency than those familiar with both Java and at least one other language. As such, it may be beneficial to differentiate debugging instruction between the two groups.

The four factors explained 17%–34% of the variance in exam scores. Large unexplained variances may indicate that the factors do not fully capture code writing proficiency. However, it may also be the opposite—final performance in programming tests or tasks may not accurately represent programming proficiency. Indeed, researchers have argued that a grade based on a final program may not be a valid indicator of programming proficiency because it does not consider the process that leads to the final program [12, 35, 64]. In this regard, assessing programming proficiency should consider both the programming processes and outcomes. Compared with typical exams focusing on programming outcomes, which are summative [32], an assessment combining processes and outcomes is more formative because it may provide more accurate and comprehensive diagnosis information about programming skills. For example, for the CS1 course in this study, combining the factor model and exam performance may bring insights about which code-writing skills students at a particular performance level are weak in. Such understanding may help instructors in designing more individualized instruction to foster weak skills. For a programming course different from this study, the factor model may not be directly applicable, but the same approach could be used to develop a new factor model fitting the course. That is, defining features theoretically related to code writing, extracting the features from the programming traces, and applying factor analysis to obtain the factor model.

This study also found that students unfamiliar with any programming language had lower proficiency in three of the four code-writing skills compared with their peers in the early stage of a CS1 course. This result suggests that CS1 instructors should pay particular attention to these students. Moreover, learning code style, syntax, semantics, and debugging together may be too overwhelming for individuals without prior programming experience. It may be difficult to disentangle and sequence instructions in syntax, semantics, and debugging, but code style can be learned after these other skills have begun to be developed. Thus, if writing code in a good style is one of the instructional goals, it may be better to put it aside at the beginning of a CS1 course and introduce it after students have experience with other subskills.

Overall, the results are promising and call for more investigation into programming trace analysis and code-writing skills. A valid theory of novice programmers' code-writing skills would go a long way toward guiding the instruction of CS1 courses.

AUTHOR CONTRIBUTIONS

Yingbin Zhang: Conceptualization, methodology, formal analysis, writing—original draft preparation, review, and editing. **Luc Paquette**: Conceptualization, methodology, funding acquisition, writing—review and editing, supervision. **Juan Pinto**: Conceptualization, writing—review,

and editing. **Aysa Xuemo Fan**: Conceptualization, writing—review, and editing.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation (grant numbers DRL-1942962) and the China Scholarship Council (grant numbers 201806040180).

CONFLICT OF INTEREST STATEMENT

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are not openly available due to sensitive private information about students. The data belongs to the corresponding author's institution and are available from the corresponding author upon reasonable request and with permission of the corresponding author's institution.

ORCID

Yingbin Zhang http://orcid.org/0000-0002-2664-3093
Luc Paquette http://orcid.org/0000-0002-2738-3190
Juan D. Pinto http://orcid.org/0000-0002-2972-485X

REFERENCES

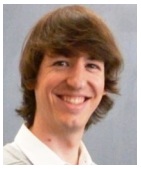
1. AERA, APA and NCME, *Standards for educational and psychological testing*, American Educational Research Association, Washington, DC, 2014.
2. M. Ahmadzadeh, D. Elliman and C. Higgins, *An analysis of patterns of debugging among novice computer science students*, Association for Computing Machinery (Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education), 2005, pp. 84–88.
3. B. S. Alqadi and J. I. Maletic, An empirical study of debugging patterns among novices programmers Association for Computing Machinery (Proceedings of the 48th ACM Technical Symposium on Computer Science Education), 2017, pp. 15–20.
4. A. S. Beavers, J. W. Lounsbury, J. K. Richards, S. W. Huck, G. J. Skolits and S. L. Esquivel, *Practical considerations for using exploratory factor analysis in educational research*, Pract. Assess. Res. Eval. **18** (2013), 6.
5. B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, *Effective compiler error message enhancement for novice programming students*, Comput. Sci. Educ. **26** (2016), no. 2-3, 148–175.
6. M. Berges, M. Striewe, P. Shah, M. Goedicke and P. Hubwieser, Towards deriving programming competencies from student errors, (Proceedings of the 2016 International Conference on Learning and Teaching in Computing and Engineering) 2016, pp. 19–23.
7. M. Berland, T. Martin, T. Benton, C. Petrick Smith, and D., Davis, *Using learning analytics to understand the learning pathways of novice programmers*, J. Learn. Sci. **22** (2013), no. 4, 564–599.
8. J. B. Biggs and K. F. Collis, *Evaluating the quality of learning*: The SOLO taxonomy (Structure of the Observed Learning Outcome), Academic Press, Cambridge, MA, 2014.
9. P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper and D. Koller, *Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming*, J. Learn. Sci. **23** (2014), no. 4, 561–599.
10. B. S. Bloom, *Taxonomy of educational objectives*: The classification of educational goals, Longman Group, Harlow, Essex, England, 1956.
11. M. W. Browne and R. Cudeck, *Alternative ways of assessing model fit*, Sociol. Methods. Res. **21** (1992), no. 2, 230–258.
12. R. Cardell-Oliver, How can software metrics help novice programmers?, Australian Computer Society (Proceedings of the 13th Australasian Computing Education Conference), 2011, pp. 55–62.
13. A. S. Carter, C. D. Hundhausen and O. Adesope, *The normalized programming state model: Predicting student performance in computing courses based on programming behavior*, Association for Computing Machinery (Proceedings of the Eleventh Annual International Conference on International Computing Education Research), 2015, pp. 141–150.
14. G. W. Cheung and R. B. Rensvold, *Evaluating goodness-of-fit indexes for testing measurement invariance*, Struct. Equ. Modeling Multidiscip. J. **9** (2002), 233–255.
15. C.Y. Chou and P.F. Sun, *An educational tool for visualizing students' program tracing processes*, Comput. Appl. Eng. Educ. **21** (2013), no. 3, 432–438.
16. T. Clear, J. Whalley, P. Robbins, A. Philpott, A. Eckerdal, M. Laakso, and R. Lister, *Report on the final BRACElet workshop: Auckland University of Technology, September 2010*, J. Appl. Comput. Inform. Technol. **15** (2011), no. 1, T1.
17. K. Cunningham, S. Blanchard, B. Ericson, and M. Guzdial, Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw, Association for Computing Machinery (Proceedings of the 13th ACM Conference on International Computing Education Research), 2017, pp. 164–172.
18. M. J. Davidson, Towards an understanding of program writing as a cognitive process: Analysis of keystroke logs, Association for Computing Machinery (Proceedings of the 17th ACM Conference on International Computing Education Research), 2021, pp. 407–408.
19. S. P. Davies, *Models and theories of programming strategy*, Int. J. Man-Mach. Stud. **39** (1993), no. 2, 237–267.
20. J. Edwards, J. Ditton, D. Trninic, H. Swanson, S. Sullivan, and C. Mano, Syntax exercises in CS1, Association for Computing Machinery (the 16th ACM Conference on International Computing Education Research), 2020, pp. 216–226.
21. J. Eyolfson, L. Tan, and P. Lam, Do time of day and developer experience affect commit bugginess?, Association for Computing Machinery (Proceedings of the 8th Working Conference on Mining Software Repositories), 2011, pp. 153–162.
22. S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander, *Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers*, Comput. Sci. Educ. **18** (2008), no. 2, 93–116.

23. C. Fornell and D. F. Larcker, *Evaluating structural equation models with unobservable variables and measurement error*, J. Mark. Res. **18** (1981), no. 1, 39–50.
24. M. Fowler, D. H. Smith IV, M. Hassan, S. Poulsen, M. West, and C. Zilles, *Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study*, Comput. Sci. Educ. **32** (2022), 355–383.
25. J. Hair, W. Black, B. Babin and R. Anderson, *Multivariate data analysis*: A global perspective, Prentice Hall, 2009.
26. J. F. Hair Jr., M. Sarstedt, L. Hopkins, and V. G. Kuppelwieser, *Partial least squares structural equation modeling (PLS-SEM): An emerging tool in business research*, Eur. Bus. Rev. **26** (2014), no. 2, 106–121.
27. R. Hosseini, A. Vihavainen, and P. Brusilovsky, Exploring problem solving paths in a Java programming course, (Psychology of Programming Interest Group Annual Conference 2014), 2014, pp. 65–76.
28. M. Hristova, A. Misra, M. Rutter, and R. Mercuri, *Identifying and correcting java programming errors for introductory computer science students*, ACM SIGCSE Bull. **35** (2003), no. 1, 153–156.
29. L. Hu and P. M. Bentler, *Cutoff criteria for fit indexes in covariance structure analysis: Conventional criteria versus new alternatives*, Struct. Equ. Modeling Multidiscip. J. **6** (1999), no. 1, 1–55.
30. P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. Á. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll, Educational data mining and learning analytics in programming: Literature review and case studies, Association for Computing Machinery (Proceedings of the 2015 Innovation and Technology in Computer Science Education Conference), 2015, pp. 41–63.
31. M. C. Jadud, An exploration of novice compilation behaviour in BlueJ, vol., Doctoral, University of Kent, Canterbury, United Kingdom 2006.
32. M. C. Jadud, *Methods and tools for exploring novice compilation behaviour*, Association for Computing Machinery, (Proceedings of the 2nd International Workshop on Computing Education Research) 2006, pp. 73–84.
33. K. G. Jöreskog, How large can a standardized coefficient be, 1999. Available at http://www.statmodel.com/download/Joreskog.pdf
34. T. Lancaster, A. V. Robins, and S. A. Fincher, *Assessment and plagiarism*, The Cambridge Handbook of Computing Education Research (A. V. Robins and S. A. Fincher, eds.), Cambridge University Press, Cambridge, 2019, pp. 414–444.
35. H. C. Lane and K. VanLehn, *Intention-based scoring: an approach to measuring success at solving the composition problem*, ACM SIGCSE Bull. **37** (2005), no. 1, 373–377.
36. R. S. Lemos, *Measuring programming language proficiency*, AEDS J. **13** (1980), no. 4, 261–273.
37. R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, *A multi-national study of Reading and tracing skills in novice programmers*, ACM SIGCSE Bull. **36** (2004), no. 4, 119–150.
38. R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, *Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy*, ACM SIGCSE Bull. **38** (2006), no. 3, 118–122.
39. M. Lopez, J. Whalley, P. Robbins, and R. Lister, *Relationships between reading, tracing and writing skills in introductory programming*, Association for Computing Machinery, (Proceedings of the Fourth International Workshop on Computing Education Research) 2008, pp. 101–112.
40. R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, Debugging: A review of the literature from an educational perspective, *Computer*, Sci. Educ. **18** (2008), no. 2, 67–92.
41. M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y.B.D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students*, ACM SIGCSE Bull. **33** (2001), no. 4, 125–180.
42. L. Murphy, S. Fitzgerald, R. Lister, and R. McCauley, Ability to 'explain in plain english' linked to proficiency in computer-based programming, Association for Computing Machinery (Proceedings of the 9th Annual International Conference on International Computing Education Research), 2012, pp. 111–118.
43. G. L. Nelson, B. Xie, and A. J. Ko, Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in CS1, Association for Computing Machinery (Proceedings of 14th ACM Conference on International Computing Education Research), 2017, pp. 2–11.
44. R. D. Pea, *Language-independent conceptual "bugs" in novice programming*, Journal of Educational Computing Research. **2** (1986), no. 1, 25–36.
45. A. Petersen, M. Craig, and D. Zingaro, *Reviewing CS1 exam question content*, Association for Computing Machinery, (Proceedings of the 42nd ACM technical Symposium on Computer Science Education) 2011, pp. 631–636.
46. J. Pinto, Y. Zhang, L. Paquette, and A. Fan, *Investigating elements of student persistence in an introductory computer science course*, CEUR-WS (Joint Proceedings of the Workshops at the International Conference on Educational Data Mining 2021) 2021, pp. 1–10.
47. D. L. Putnick and M. H. Bornstein, *Measurement invariance conventions and reporting: the state of the art and future directions for psychological research*, Dev. Rev. **41** (2016), 71–90.
48. F. Rahman and P. Devanbu, *Ownership, experience and defects: A fine-grained study of authorship*, Association for Computing Machinery (Proceedings of the 33rd International Conference on Software Engineering), 2011, pp. 491–500.
49. W. Revelle, *Psych*: Procedures for personality and psychological research, Northwestern University, Evanston, Illinois, 2017.
50. K. Rivers and K. R. Koedinger, *Data-driven hint generation in vast solution spaces: A self-improving python programming tutor*, Int. J. Artif. Intell. Educ. **27** (2017), no. 1, 37–64.
51. A. Robins, J. Rountree, and N. Rountree, *Learning and teaching programming: A review and discussion*, Computer Science Education. **13** (2003), no. 2, 137–172.
52. Y. Rosseel, *Lavaan: An R package for structural equation modeling*, J. Stat. Softw. **48** (2012), no. 2, 1–36.
53. L. Rutkowski and D. Svetina, *Assessing the hypothesis of measurement invariance in the context of large-scale international surveys*, Educ. Psychol. Meas. **74** (2013), no. 1, 31–57.
54. J. Śliwerski, T. Zimmermann, and A. Zeller, *When do changes induce fixes?*, ACM SIGSOFT Softw. Eng. Notes. **30** (2005), no. 4, 1–5.

55. D. Sörbom, *Model modification*, Psychometrika. **54** (1989), no. 3, 371–384.
56. A. Stefik and S. Siebert, *An empirical investigation into programming language syntax*, ACM Trans. Comput. Educ. **13** (2013), no. 4, 1–40.
57. E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, *Predicting at-risk novice Java programmers through the analysis of online protocols*, Association for Computing Machinery, (Proceedings of the 7th International Workshop on Computing Education Research) 2011, pp. 85–92.
58. A. Taherkhani and L. Malmi, *Beacon-and schema-based method for recognizing algorithms from students' source code*, J. Educ. Data Min. **5** (2013), no. 2, 69–101.
59. D. Teague, M. Corney, A. Ahadi, and R. Lister, Swapping as the 'Hello World' of relational reasoning: Replications, reflections and extensions, Australian Computer Society (Proceedings of the 14th Australasian Computing Education Conference), Australia, 2012, pp. 87–94.
60. A. E. Tew and M. Guzdial, *Developing a validated assessment of fundamental CS1 concepts*, Association for Computing Machinery, (Proceedings of the 41st ACM Technical Symposium on Computer Science Education), 2010, pp. 97–101.
61. Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, and F. Saleem, *Bloom's taxonomy: A beneficial tool for learning and assessing students' competency levels in computer programming using empirical analysis*, Computer Applications in Engineering Education. **28** (2020), no. 6, 1628–1640.
62. I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. B. Kolikant, J. Sorva, and T. Wilusz, A fresh look at novice programmers' performance and their teachers' expectations, Association for Computing Machinery (Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education), 2013, pp. 15–32.
63. A. Venables, G. Tan, and R. Lister, *A closer look at tracing, explaining and code writing skills in the novice programmer*, Association for Computing Machinery, (Proceedings of the 5th International Workshop on Computing Education Research Workshop), 2009, pp. 117–128.
64. M. M. Villamor, *A review on process-oriented approaches for analyzing novice solutions to programming problems*, Res. Pract. Technol. Enhanc. Learn. **15** (2020), 8.
65. E. Vinker and A. Rubinstein, Mining code submissions to elucidate disengagement in a computer science MOOC, Association for Computing Machinery (Proceedings of the 12th International Learning Analytics and Knowledge Conference), 2022, pp. 142–151.
66. C. Watson, F. W. B. Li, and J. L. Godwin, Predicting performance in an introductory programming course by logging and analyzing student programming behavior, (Proceedings of 2013 IEEE the 13th International Conference on Advanced Learning Technologies), 2013,pp. 319–323.
67. C. Watson, F. W. B. Li, and J. L. Godwin, No tests required: Comparing traditional and dynamic predictors of programming success, Association for Computing Machinery (Proceedings of the 45th ACM Technical Symposium on Computer Science Education), 2014, pp. 469–474.
68. J. L. Whalley, R. Lister, E. Thompson, T. Clear, P. Robbins, P. K. A. Kumar, and C. Prasad, An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies, Australian Computer Society (Proceedings of the 8th Australasian Conference on Computing Education), 2006, pp. 243–252.
69. L. E. Winslow, *Programming pedagogy—a psychological overview*, ACM SIGCSE Bull. **28** (1996), no. 3, 17–22.
70. A. F. Wise, S. Knight, and X. Ochoa, *What makes learning analytics research matter*, J. Learn. Anal. **8** (2021), no. 3, 1–9.
71. B. Xie, D. Loksa, G. L. Nelson, M. J. Davidson, D. Dong, H. Kwik, A. H. Tan, L. Hwa, M. Li, and A. J. Ko, *A theory of instruction for introductory programming skills*, Comput. Sci. Educ. **29** (2019), no. 2-3, 205–253.

AUTHOR BIOGRAPHIES

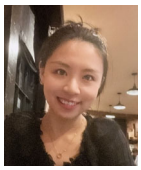
Yingbin Zhang is a research associate at the Institute of Artificial Intelligence in Education, South China Normal University. He received PhD in the department of Curriculum and Instruction, the University of Illinois at Urbana-Champaign, where he was a member of the Human-centered Educational Data Science (HEDS) Lab. His research interests are how AI and digital learning environments can support children in becoming self-regulated learners and make data science accessible for all. He is also interested in how sequential analysis methods (e.g., sequential pattern mining and process mining) can help us understand learning process.



Luc Paquette is an associate professor in the department of curriculum & instruction at the University of Illinois Urbana-Champaign. His research focuses on the usage of machine learning, data mining and knowledge engineering approaches to analyze and build predictive models of the behavior of students as they interact with digital learning environments such as MOOCs, intelligent tutoring systems, and educational games. He is interested in studying how those behaviors are related to learning outcomes and how predictive models of those behaviors can be used to better support the students' learning experience.

Juan D. Pinto is a PhD student at the University of Illinois Urbana-Champaign. His research interests revolve around applications of data science methods in educational settings, including the analysis of big data to inform learning theory and the use of statistical and machine learning models to improve teaching and learning. His current work as a member of the Human-centered Educational Data

Science (HEDS) Lab deals largely with the application of these methods in computer science education.



Aysa Xuemo Fan Aysa is a PhD student at the University of Illinois Urbana-Champaign specializing in educational data mining, focusing on educational technology, design, and natural language processing. Previously, she was a member of CoLearnLab, where she analyzed collaborative learning behaviors using clustering and video analysis. Her current research involves analyzing programming data from an introductory CS course to uncover students' debugging strategies.

How to cite this article: Y. Zhang, L. Paquette, J. D. Pinto, and A. X. Fan, *Utilizing programming traces to explore and model the dimensions of novices' code-writing skill*, Comput. Appl. Eng. Educ. (2023), 1–18. https://doi.org/10.1002/cae.22622