Education and Information Technologies (2023) 28:4673–4701
https://doi.org/10.1007/s10639-022-11343-7

Combining latent profile analysis and programming traces to understand novices' differences in debugging

Yingbin Zhang[1] · **Luc Paquette**[1] · **Juan D. Pinto**[1] · **Qianhui Liu**[1] · **Aysa Xuemo Fan**[1]

Received: 12 May 2022 / Accepted: 8 September 2022 / Published online: 22 October 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

It is widely recognized that debugging is challenging for novice programmers and, as such, computing educators and researchers have called for explicit debugging instruction. Debugging requires various knowledge and skills, and different students may show different strengths and weaknesses. An understanding of such individual differences is important as it may guide personalized instruction. The current study investigated individual differences in debugging in an undergraduate introductory computer science course. We extracted variables related to debugging from students' submission traces to programming problems in the first month of the course. We applied latent profile analysis to these variables and identified three distinctive profiles. Profile A showed higher debugging accuracy and speed. Profile B showed lower debugging performance in runtime and logic errors, while profile C had lower performance in syntactic errors and tended to make large code edit every submission. Students' gender and self-rated programming ability predicted profile membership. Moreover, profile A got higher scores than the others in the first exam, and this difference persisted in the second and third exam, even controlling for background variables and score on the first exam. We investigated how students transitioned across debugging profiles over the duration of the course. From the beginning to the end of the course, a large part of students stayed in lower performance profiles. Overall, these findings support the call that debugging should be taught at an early stage and suggest that different groups may need different debugging instructions or support.

Keywords Debugging · Novice programmer · CS1 education · Programming trace · Person-centered study · Latent profile analysis

Extended author information available on the last page of the article

1 Introduction

Debugging is an indispensable but difficult part of programming, with 73% of professional software developers reporting that they spend 20–60% of their work time debugging (Perscheid et al., 2017). However, it can be frustrating and time-consuming for novice programmers, who have an inadequate understanding of programming constructs (e.g., loop and recursion) and lack experts' knowledge base of stereotypical bugs and their symptoms (Li et al., 2019; McCauley et al., 2008; Pea, 1986). Moreover, even experienced programmers may have a low level of debugging knowledge (Beller et al., 2018). Some of the main reasons for this phenomenon may be that debugging is hardly emphasized in introductory computer science (CS1; Malik & Coldwell-Neilson 2017) and most programmers receive little formal debugging training (Perscheid et al., 2017).

Therefore, increasingly, computing educators and researchers have called for teaching debugging explicitly at an early stage (Chmiel & Loui, 2004; Li et al., 2019; McCauley et al., 2008; Rich et al., 2019). Empirical evidence supports this initiative. Studies have shown that explicit instruction on debugging may foster students' skill to hypothesize the bug cause (Whalley et al., 2021a), improve debugging performance (Michaeli & Romeike, 2019), and enhance the ability to read programs and solve programming tasks (Eranki & Moudgalya, 2014). The benefits may extend to non-cognitive factors, such as self-efficacy toward debugging (Michaeli & Romeike, 2019) and critical self-reflection (DeLiema et al., 2019).

Novices in general may face different challenges, given that debugging requires various skills and knowledge (Decasse & Emde, 1988; Li et al., 2019). One group may be good at debugging some types of errors and applying some strategies, while another group may be good at the others. Identifying latent groups with heterogeneous patterns of debugging performance and strategies may allow for more personalized instruction. Thus, the person-centered approach, which aims to model heterogeneous patterns in variables of interest (Hickendorff et al., 2018), has the potential for understanding individual differences in debugging and providing actionable insights for instruction. Indeed, empirical studies have found groups of novices with qualitative differences in programing (Blikstein et al., 2014; Jiang et al., 2019; Perkins et al., 1986). For example, when analyzing programming logs on an assignment, Blikstein et al., (2014) found three groups with different programming pathways. Group membership had better predictive power on exam scores than assignment scores. Nevertheless, these person-centered studies have focused on how novices wrote a correct program rather than on how they handled the task of debugging. There is a lack of person-centered analysis on novices' debugging. This paper aims to address this gap.

Specifically, the current study extracted variables related to debugging from undergraduates' submission traces to programming problems in a CS1 course. We conducted a latent profile analysis (LPA) on these variables and identified three profiles with distinctive debugging patterns. Profile A showed higher debugging accuracy and speed. Profile B showed lower debugging performance in runtime and logic errors, while profile C had lower performance in syntactic errors and tended to make large code edit every submission. Students' demographics, prior programming experiences, and exam scores were related to profile membership. Latent transition analysis

found that a large part of students stayed in the lower performance profiles from the beginning to the end of the course. To the best of our knowledge, this work is the first person-centered study that utilizes programing traces to understand novices' differences in debugging. The results bring insights about individual differences in debugging at an early stage and how these differences are related to students' backgrounds and influence subsequent exam scores. The finding supports the call that debugging should be taught early, and the revealed distinctive debugging profiles provide guidance for personalized debugging instructions.

2 Literature review

2.1 Programming errors and debugging process

Debugging is the activity of finding and fixing errors or bugs (or faults, defects) in a program (McCauley et al., 2008; Zeller, 2009). It is a special case of troubleshooting (Katz & Anderson, 1987; Li et al., 2019). Some researchers distinguish between errors and bugs: errors are externally observable symptoms of bugs, while bugs are the cause of errors (Zeller, 2009). Errors can be classified into three general categories (Hristova et al., 2003; McCauley et al., 2008): (1) syntax or compiler errors, which occur when the program violates the syntax rules of the language and cannot be compiled, such as missing a "}"; (2) runtime errors, which occur when the program can be compiled but not run or executed, such as "index out of bounds"; (3) logic or semantic errors, which occur when the program can be executed but does not generate the expected output. For example, a logic error would occur if the expected output for a given program is the difference between two integers, but the actual output is the sum of the two integers instead. Prior studies have investigated novices' debugging on runtime and logic errors together without making an explicit distinction (e.g., Alqadi & Maletic 2017; Fitzgerald et al., 2008), possibly because these errors are more general than syntax errors and relatively independent of language (Pea, 1986).

McCauley et al., (2008) reviewed studies around programming errors and concluded that most errors or bugs occur because of fragile knowledge or a superbug. Fragile knowledge is the knowledge that students partially know and cannot apply correctly (Perkins & Martin, 1986). The superbug refers to the misconception that ordering the computer or system via programming is analogous to human communication, and the system has "intelligent interpretive powers" of understanding programming language (Pea, 1986). Misconceptions about language-related constructs also result in some errors. However, McCauley et al., (2008) noted that why bugs occur has no simple answer.

Novices tend to find errors randomly, while experts use a systematic debugging process (Whalley et al., 2021a). Systematic debugging is an essential computational thinking practice (Rich et al., 2020). It can assist programmers in efficiently fixing even tractable errors (Spinellis, 2018). There are a few frameworks of systematic debugging (Klahr & Carver, 1988; Li et al., 2019; Zeller, 2009). For instance, Li et al., (2019) adapted a four-step framework of systematic troubleshooting for debugging. The first step is constructing a mental model of the purpose of a program and

how it achieves the purpose. The second step is identifying the discrepancy between the expected and actual behaviors of the program, i.e., the errors. The third step, which is the core step, is developing a hypothesis about the cause of errors and evaluating the hypotheses. This step is iterative. A hypothesis evaluated as correct can help debuggers to narrow down the range of bug locations and develop finer hypotheses. By contrast, an incorrect hypothesis is used to generate alternative hypotheses. Novices are particularly weak at this step. They have difficulty in generating correct or precise hypotheses (Whalley et al., 2021a) and often stick with initial hypotheses without considering alternatives (Fitzgerald et al., 2010; Vessey, 1985). The final step is generating and verifying solutions. The other frameworks of systematic debugging can be aligned with Li et al.'s four steps (Whalley et al., 2021a).

2.2 Debugging knowledge and strategies

Debugging is a complex skill and entails various knowledge (Decasse & Emde, 1988). Based on a troubleshooting framework, Li et al., (2019) classified debugging knowledge into five types: domain, system and procedural knowledge, debugging strategies, and programming experiences. (1) Domain knowledge refers to the understanding of the implementation language of a program. (2) System knowledge concerns the components, interactions between components, structure and purpose of the program. (3) Procedural knowledge concerns utilizing debugging variables of an IDE, such as breakpoints and memory inspection. (4) Debugging strategies include global and local strategies. Global strategies are independent of programs and can be applied across contexts, such as forward and backward reasoning (Katz & Anderson, 1987). Local strategies are for special programs, such as using print statements and test cases. (5) Programming experiences form the knowledge base of stereotyped bugs and their symptoms. This knowledge base may be useful for developing rational hypotheses about bugs (Gugerty & Olson, 1986).

Different knowledge is interdependent. Without an understanding of implementation language and program behaviors, procedural knowledge and debugging strategies may not be applied effectively (Li et al., 2019). Thus, good debuggers are also good programmers (Ahmadzadeh et al., 2005; Fitzgerald et al., 2008). On the other hand, debugging is challenging without mastering procedural knowledge and debugging strategies. Consequently, good programmers are not necessarily good at debugging (Ahmadzadeh et al., 2005; Fitzgerald et al., 2008).

Researchers have investigated novices' debugging strategies to understand their debugging processes (McCauley et al., 2008). Studies have found that novices used various strategies to locate and fix errors (Fitzgerald et al., 2010; Murphy et al., 2008). Some strategies were usually effective, such as rereading the problem specification and reexamining the program output to improve the understanding of the problem and program. Some strategies were less effective, such as working around problems (e.g., replacing the code with completely new code) and tinkering (editing the program randomly and unproductively). The others were used both effectively and ineffectively. For example, students used print statements a lot to trace the program, but these statements are often only useful for following the flow of control.

Rich et al., (2019) reviewed K-8 CS education studies and identified debugging strategies that students should learn. While these strategies are derived from K-8 studies, they may be helpful for novices regardless of the grade. For instance, iterative refinement refers to making a small change on the code and testing it to check if the error is fixed. This is a special case of incremental development, a "code-a-little, test-a-little" method for software development that is recommended for both novices and experts (Baumstark & Orsega, 2016; Larman & Basili, 2003). The strategy of addressing compiler errors by their appearance order has also been investigated by studies in higher education, known as "fix the first, ignore the rest" (Becker et al., 2018). It is useful for debugging because compiler errors in the earlier part of a program may cause errors in the subsequent part. Fixing the former automatically fixes the latter. This strategy is a special form of correcting one bug at a time (Liu et al., 2017), which emphasizes decomposing the task of debugging compiler errors by focusing on and editing one error at a time because compiler errors may be mutually related.

2.3 Novices' debugging behaviors and programming traces

Studies have observed how novices debug their own and others' buggy code. In the latter situation, researchers can focus on debugging on particularly designed errors and obtain a deep understanding of debugging performance and difficulties on these errors (Ahmadzadeh et al., 2005; Alqadi & Maletic, 2017; Fitzgerald et al., 2008). However, findings under this situation may have limited generalizability to the real setting where participants usually troubleshoot their own buggy code because the familiarity toward their code may differ from the familiarity toward others' code (Lewis, 2012). Consequently, debugging behavior and strategy may be different. Indeed, novices were more likely to use forward reasoning when debugging others' code but backward reasoning when debugging their code (Katz & Anderson, 1987). The current study focused on novices' behaviors while debugging their own code.

In either situation, the process data is critical for understanding debugging. Studies have used process data from various channels. Think-aloud data and human observation capture rich information about debugging progression and strategies (Fitzgerald et al., 2008; Liu et al., 2017; Murphy et al., 2008), but data collection and processing are labor-consuming and not scalable. By contrast, it is easier to collect a large sample's programming traces over many problems because the collection can be automated (Ihantola et al., 2015). Programming traces have been used to extract indicators of debugging performance, such as debugging success rate and time (Ahmadzadeh et al., 2005; Alqadi & Maletic, 2017; Fitzgerald et al., 2008). Based on programming traces, Fitzgerald et al., (2008) found that novices with high programming ability might show low debugging success rates. Debugging success rate and time were related to years of programming experience (Alqadi & Maletic, 2017). Some debugging strategies can also be inferred from programming traces, such as correcting one bug at a time (Becker et al., 2018; Liu et al., 2017) and iterative refinement (Kazerouni et al., 2017).

2.4 Novices' differences in debugging

Studies have used process data to understand novices' differences in debugging. Lin et al., (2016) undergraduates' differences in eye-movement sequences while fixing buggy programs. They found that participants with high-debugging performance tended to trace the program in an order matching the logic of the code, while participants with low performance more likely traced the program in a line-by-line order, without considering the logic. Alqadi & Maletic (2017) asked novices to fix a program with eight errors and grouped novices manually into three clusters based on the number of errors fixed. Stoppers fixed six or fewer errors, while movers fixed seven or eight errors. Tinkerers generated more errors or did not fix any error. Jemmali et al., (2020) visualized programming traces in a puzzle game to investigate how novices corrected errors toward a correct solution. They identified three participants with distinct debugging progression. Participant A showed an efficient progression: errors in their code linearly and quickly decreased to zero. By contrast, participant B struggled with the errors at the beginning but fixed them through a trial-and-error approach. Participant C made a lot of submissions with medium to large edits on code, but the same errors persisted.

These studies have brought different insights into novices' differences in debugging. However, they shared a methodological drawback: a small sample of novices was grouped manually. Such manual grouping may not be robust due to the arbitrary cut-off value or the subjectivity of the decision (Hickendorff et al., 2018). Moreover, the grouping in Lin et al., (2016) as well as Alqadi & Maletic (2017) is based on a single indicator of debugging performance, assuming that differences in this single indicator sufficiently represent differences in debugging. This assumption may not hold because debugging entails various knowledge and skills. Two novices' differences may be inconsistent across these knowledge and skills.

3 The current study

The current study applied the person-centered method, which can uncover latent subpopulations in a statistically sound way, to address the above methodological limitation. The main person-centered approach is clustering (Hickendorff et al., 2018), including traditional clustering algorithms, such as *K*-means and hierarchical clustering, and model-based clustering, such as latent profile analysis (LPA; Oberski 2016). LPA estimates the membership probabilities that a subject belongs to latent subpopulations and assigns this subject to the subpopulation with the highest membership probability. Some traditional clustering algorithms also estimate the membership probability, such as fuzzy c-means (Bezdek et al., 1984), but LPA allows controlling the assignment error in the subsequent analysis of the latent membership with external variables (Asparouhov & Muthén, 2014). In addition, researchers can quantify how well the LPA model fits the data. Thus, this study applied LPA to debugging-related variables to investigate novices' differences in debugging.

Figure 1 displays the conceptual model and research questions of this study. Phases one to three represent earlier to latter parts of the course, respectively. The focus

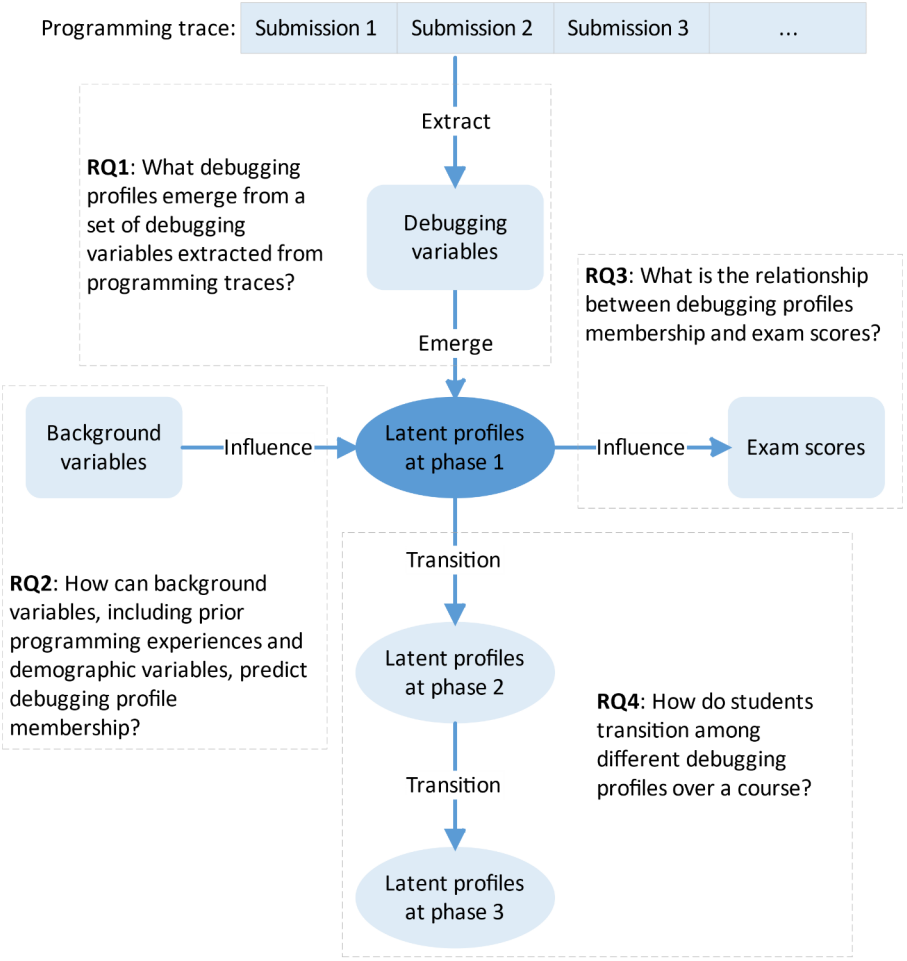

Fig. 1 Conceptual model of the current study

is on the latent profiles at phase one because researchers have suggested teaching debugging as early as possible (Chmiel & Loui, 2004; Li et al., 2019; Murphy et al., 2008). Firstly, we ask what debugging profiles emerge from students' programming traces in phase one of the course (research question 1; RQ1). Based on the result of RQ1, we further investigate the relationship between debugging profile membership and students' backgrounds (RQ2) as well as exam performance (RQ3). Students in a certain debugging profile at phase one might not stay in the same profile in subsequent phases because they might learn about debugging knowledge and skills over the course of this study. Thus, for RQ4, we utilize students' programming traces in phases two and three to explore how students transit among debugging profiles over the duration of the course.

4 Method

4.1 Participants and data

Data were collected from a CS1 course at a public university in the US during the Fall 2019 semester. The course lasted 16 weeks. It taught Java programming and was hosted on PrairieLearn, a web-based, problem-driven learning system (West, Herman, & Zilles, 2015). Students used the system to submit solutions to programming homework and lab problems as well as take weekly quizzes and three exams. Students could submit solutions to a programming problem as many times as they wanted until the deadline of homework and labs (or before the quizzes and exams ran out of time) to obtain full credit. The course instructor expected that a programming problem should take no more than 10–15 min to complete.

The learning system automatically graded each submission and generated feedback about mistakes. It first tried compiling a submission to check if the submission contained checkstyle or syntax errors. A checkstyle error occurred when the submitted code did not match the required style (e.g., an operator was not surrounded by whitespace). If a submission could not be compiled, i.e., it contained checkstyle or syntax errors, the system displayed an error message, which mainly contained the errors and corresponding lines of code. If a submission could be compiled, the system would run problem-specific tests on it. These tests checked whether the submission could implement the problem's requirement. For instance, if the problem asked students to write a program to compute the sum of two numbers, the test could randomly generate two numbers, input the numbers to the submitted code, and examine the equality between the sum and the output of the submission. The test could repeat this procedure a few times to avoid coincidence. If the submission could output a number equal to the sum every time, it implemented the requirement. Otherwise, the code contained test errors (including runtime or logic errors), and the system would display a test error message. Different from the checkstyle and syntax error message, the test error message mainly contained test errors without corresponding lines of code because a specific line of code rarely caused a test error.

The learning system automatically recorded the code, date, correctness, and error feedback of submissions. Together with other information such as exam scores, these submission traces were stored in a secure database outside the learning system. Because students' debugging skills developed during the course, aggregating the submissions over the whole semester was unreasonable. Thus, we split the semester into three phases: phase one was before the first exam (occurring in the middle of the fifth week), phase two was between the first and second exams (occurring in the middle of the tenth week), and phase three was after the second exam. For RQs 1 to 3, we only used the submissions in phase one because researchers have suggested teaching debugging as early as possible (Chmiel & Loui, 2004). Early identification of latent debugging profiles allows earlier personalized instruction or intervention. For RQ 4, we used data from all phases and conducted latent transition analysis (Collins & Lanza, 2009) to investigate students' transitions among different debugging profiles across phases (see Sect. 3.3 Analyses for details).

Table 1 Demographic information

Gender	Female	Male	Withheld [a]	
Count	185	428	4	
Percent	29.98%	69.37%	0.65%	
Major	CS	CS+[b]	Others	
Count	102	169	346	
Percent	16.53%	27.39%	56.08%	
Grade	1st year	2nd year	3rd year	4th year
Count	502	75	19	21
Percent	81.36%	12.16%	3.08%	3.40%

a: The student preferred not to report their gender.

b: The student was in a CS program plus another discipline, e.g., CS plus chemistry.

Six hundred and seventeen students completed the course and approved the use of their data for research purposes. Before the course started, they completed a survey about their prior CS experiences and their expected time commitment for out-of-class work. Table 1 presents the demographic information. Note that the same data have been used in prior work to investigate novice programmers' productive and unproductive persistence (Pinto et al., 2021), which is distinct from the current study conceptually and methodologically.

4.2 Measurements

4.2.1 Debugging variables

We extracted debugging variables from submission traces of homework problems, lab exercises, and quizzes. Exam submissions were excluded to avoid confounding the effect of debugging profiles on exam scores with students' programming behaviors during exams. Two sets of variables were related to debugging proficiencies: the debugging success rate and time. These variables have been used as debugging performance indicators in prior studies (Alqadi & Maletic, 2017; Chmiel & Loui, 2004; Fitzgerald et al., 2008).

(1–3) *Debugging success rate for checkstyle errors, syntax errors, and test errors*. The success rate for fixing checkstyle errors was calculated as the proportion of pairs of successive submissions where at least one checkstyle error in the first submission disappeared in the second submission (among all pairs where the first submission had checkstyle errors). The success rates for fixing syntax errors and fixing test errors followed the same operationalization.

Note that the feedback message only indicated one test error at a time for a submission (the first error encountered), but it might indicate multiple checkstyle and compiler errors. As such, we only considered whether the next submission fixed at least one error rather than counting how many errors the next submission fixed. We argue that both fixing one error and multiple errors indicated that the student was making progress toward a correct solution. The difference between fixing one error and multiple errors might reflect more differences in debugging strategies than in

debugging proficiencies. The seventh debugging variable considered the strategy of editing one versus multiple errors at a time.

(4–6) *Debugging time for checkstyle errors, syntax errors, and test errors.* The time for fixing checkstyle errors was the average time students spent fixing a checkstyle error. The time for fixing syntax errors and test errors follows the same operationalization.

We also computed two variables related to global debugging strategies but not necessarily related to debugging performance.

(7) *Multiple-edit rate for syntax errors.* This variable was a proportion computed via Eq. 1.

$$multiple - editrate = \frac{N1}{N2} \tag{1}$$

N2 is the number of consecutive submission pairs where the first submission had multiple syntax errors. *N1* is the number of consecutive submission pairs where the second submission edited more than one place associated with the syntax errors of the first submission. This variable was related to but distinct from the debugging success rate for syntax errors. A student might edit multiple compiler errors in one submission but fix none of them, one of them, or all of them. Moreover, they might edit one error place but fix multiple errors because errors in the earlier part might cause errors in the latter part of a program. This variable is related to the strategy of correcting one bug at a time (Liu et al., 2017). We did not compute the multiple-edit rate for checkstyle errors because checkstyle errors were independent of each other. We did not compute this rate for test errors because the feedback message only indicated one test error (the first one that occurred) for a submission.

(8) *Change rate in code size.* This variable quantified the relative magnitude of the code change between two consecutive submissions. It was the average normalized absolute difference between the number of tokens of two consecutive submissions on a problem. We computed the normalized absolute difference via Eq. 2:

$$changerate = \frac{|T1 - T2|}{Tf} \tag{2}$$

T1 and *T2* are the number of tokens in the first and second submissions of two consecutive submissions on a problem[1]. *Tf* is the number of tokens in the student's final submission on the problem. We divided $|T1 - T2|$ by *Tf* to normalize the difference since the length of a correct solution varied across problems. After obtaining the normalized absolute differences of all pairs of successive submissions, we computed the mean as the change rate in code size. This variable might be related to the strategy of iterative refinement (Baumstark & Orsega, 2016; Rich et al., 2019). Specifically, in the problem-solving context of this study, a small edit at each submission had two advantages in comparison with a large edit. First, it was easier to understand why the

[1] Note that T1, T2, and Tf do not include the number of tokens in the starter code provided to the student for a problem.

small edit fixed an error. Second, when the edit introduced new errors, it was easier to identify which part of the edit caused the new errors.

Debugging strategies that were impossible to compute from programming traces were not included in our analysis. For example, programming traces cannot provide information about whether a student used forward or backward reasoning to locate the cause of an error (Katz & Anderson, 1987). Some strategies were rarely used by students in this study, such as using print statements to check the intermediate values of a variable. We were unable to compute the exact frequency that students used print statements for debugging due to the large size of submissions, but on average, only 9.05% of students used print statements at least once in a problem, and students used print statements in only 9.27% of problems. The few uses caused these strategies to have little variation across students and were not useful for distinguishing students.

4.2.2 Programming experiences

The survey conducted before the first class 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 one to five 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 class. 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 recoded students' responses to this question into four categories: (1) none (9.89%), students selected "I've never programmed before!"; (2) Java (17.83%), students only selected Java; (3) others (22.53%), students selected one or more languages but not Java; (4) Java and others (49.75%), 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 experiences than students only familiar with Java.

4.2.3 Time commitment

The survey contained one question asking students the time that they expected to devote each week to the course outside of class. Alternatives were *1 to 5 h*, *5 to 10 h*, and *10+hours*, with a response proportion of 33.50%, 51.16%, and 15.45%, respectively.

4.2.4 Exam scores

The course of this study contained three exams, which occurred in the 5th ,10th, and 16th weeks, respectively. An exam might involve anything covered up to the time, with emphasis on the material covered since the last exam. Each exam was a mix of multiple-choice problems and small programming problems. All problems were automatically graded. The maximum point possible on an exam was 100, with

programming problems accounting for over half of the points. The multiple-choice questions allowed one or two attempts, while the programming problems allowed unlimited attempts. Students had one hour to complete an exam.

4.3 Analyses

We preprocessed the data before the formal analysis. One student made no more than 20 submissions in phase one, far fewer than the others (≥78 submissions). This student was removed from the analysis. Three students had extreme variable values (outside of mean ± five standard deviations) and were removed from the analysis because extreme outliers may undesirably influence the estimation of LPA (Vermunt & Magidson, 2002). The multiple-edit rate for syntax errors, the second and third exam scores, and the time commitment had missing data with a proportion from 0.3 to 4.1%. The Little's test indicated that the missing data were missing completely at random ($\chi^2 = 99.13$, $df = 79$, $p = 0.062$). Missing values were filled via the full information maximum likelihood imputation (Enders & Bandalos, 2001).

Based on the eight debugging variables, we conducted LPA to identify debugging profiles in phase one using Mplus 8. The analysis followed the procedure recommended by Spurk et al., (2020). The variables did not obey a multivariate normal distribution, so we used maximum likelihood estimation with sandwich estimator standard errors that are robust to non-normality. We examined models including one to six profiles with 3,000 random sets of start values, 100 iterations for each random start, and the 200 best solutions retained for final stage optimization. The variable variances were freely estimated across profiles because findings in prior studies have suggested that groups with different programming trajectories might have different variances in relevant variables (Blikstein et al., 2014; Jiang et al., 2019). Covariances or correlations were equally estimated across profiles as no theory or empirical evidence suggests that the correlations between debugging variables vary across groups. The best log-likelihood for the final stage solution was replicated in all models. We then decided on the appropriate number of profiles based on statistical fit values and content-related considerations. The statistical fit indices included the consistent Akaike information criterion (CAIC), the Bayesian information criterion (BIC), sample-size adjusted BIC (SABIC), the adjusted Lo-Mendell- Rubin likelihood ratio test (aLMR), and the bootstrap likelihood ratio test (BLRT) (Spurk et al., 2020). aLMR and BLRT compare the model of k profiles with a model of k-1 profiles. A significant result in these tests indicates that the model of k profiles is better than the model of k-1 profiles. We did not use AIC and entropy because they are not suitable for selecting the number of profiles (Tein et al., 2013). We reported entropy but only used it to evaluate the classification quality of a model. Entropy ranges from 0 to 1, with 1 representing perfect classification.

After deciding on the number of debugging profiles for phase one, we examined relations between background variables and debugging profiles via the R3STEP option in Mplus (Asparouhov & Muthén, 2014). The main function of this option is a latent variable multinomial logistic regression where the profile membership was regressed on background variables, controlling classification errors. The background variables include gender, major, grade, self-rated programming ability, language

Table 2 Fit statistics for models with one to six profiles

Model	# FP	LL	CAIC	BIC	SABIC	aLMR p and meaning		BLRT p and meaning		Entropy
1	44	-6472.65	13040.27	13227.71	13088.02	-		-		-
2	61	-6228.24	12592.20	12847.99	12654.33	0.000	2>1	0.000	2>1	0.687
3	78	-6127.51	12434.10	12755.65	12508.02	0.001	3>2	0.000	3>2	0.702
4	95	-6055.87	12337.02	12721.48	12419.88	0.336	4<3	0.000	4>3	0.677
5	112	-6005.15	12284.93	12729.16	12373.59	0.201	5<4	0.000	5>4	0.695
6	129	-5955.60	12238.65	12739.18	12329.63	0.115	6<5	0.000	6<5	0.736

Note. #FP, number of free parameters; LL, log-likelihood; CAIC, consistent AIC; BIC, Bayesian information criterion; SABIC, sample-size adjusted BIC; aLMR, adjusted Lo-Mendell-Rubin likelihood ratio test; BLRT, bootstrap likelihood ratio test. 2>1 means the model with two profiles fitted the data better than the model with one profile.

familiarity, and time commitment. Except for self-rated programming ability, the other variables were dummy coded. For gender, the female group was the reference group. Students who preferred not to report their gender were excluded from the current analysis because the sample size (four) was too small to obtain reliable estimates. For major, students who were not pursuing a CS-related degree were the reference group. For grade, we only coded whether a student was in the 1st year because the number of students in the 3rd and 4th years was small. For language familiarity, the group unfamiliar with any programing language was the reference group. For time commitment, the group that expected themselves to invest *1 to 5 h* per week was the reference group.

To investigate how exam scores differ across debugging profiles of phase one, we applied the Block-Croon-Hagenaars approach (BCH; Bakk & Vermunt 2016). Its main function is a weighted analysis of variance (ANOVA). The weights are inversely related to classification errors. We controlled the background variables because students' backgrounds might influence exam scores.

We conducted latent transition analysis to investigate how students transitioned among debugging profiles. The problem complexity and difficulty increased over the semester, so debugging performance on earlier problems was not directly comparable with later problems (e.g., problems in phases one and two). Thus, we could not test the number of latent profiles simultaneously for all phases. Instead, we determined the profile number in phases two and three separately based on the aforementioned procedure for identifying the best number of debugging profiles. This procedure is suggested by researchers of latent transition analysis (Asparouhov & Muthén, 2014; Nylund-Gibson et al., 2014). We then used a three-step approach to estimate latent transition probabilities (Nylund-Gibson et al., 2014).

5 Results

5.1 RQ1: Identification of debugging profiles

Table 2 presents the statistical fit indices and tests for LPA models with one to six profiles. CAIC and SABIC decreased as the number of profiles increased. BIC reached

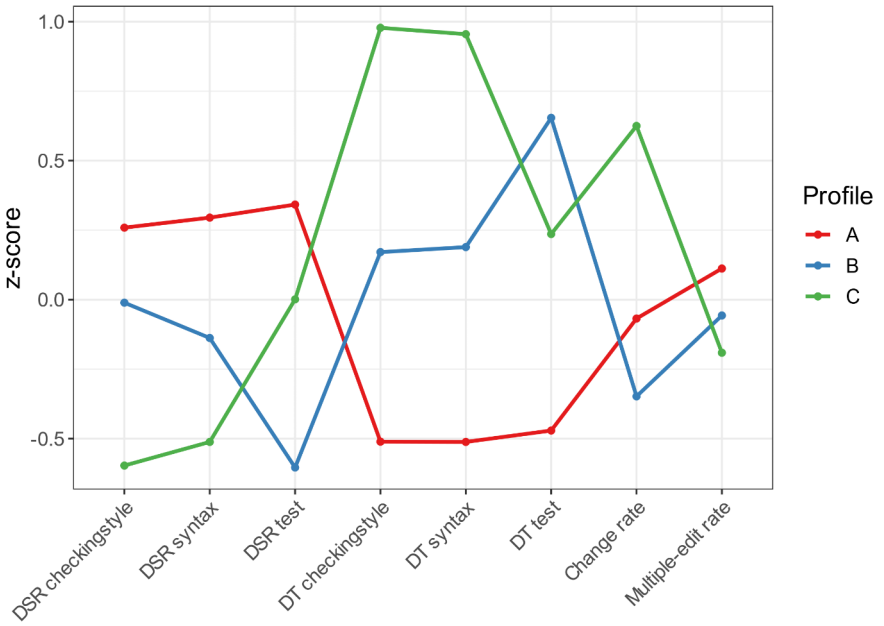

Fig. 2 Average variable z-scores of the three profiles in phase one. *Note*. A point represents the average z-score per variable for each profile. DSR: debugging success rate for a type of error. DT: debugging time for one error. Change rate: mean absolute change rate in code size. Multiple-edit rate: when the first submission has more than one syntax error, the probability that a student edited multiple error places, no matter fixing them or not.

the minimum at the model with four profiles. aLMR was not statistically significant when the number of profiles was greater than three, indicating that the model with four profiles was not superior to the model with three profiles, which was superior to the model with two profiles. BLMRT was significant for all models and not useful for determining the appropriate number of profiles. Thus, a model with three or four profiles might be the best solution. Further inspection found that the model with four profiles arbitrarily split a profile in the three-profile model into two smaller profiles that were close in all debugging variables. This indicated that the four-profile model did not provide more insights than the three-profile model. The entropy of the three-profile model was 0.702, indicating an acceptable classification quality (Muthén, 2004). Based on these methodological and content considerations, we adopted the three-profile model.

Figure 2 displays the *z*-standardized mean scores of debugging variables to illustrate the three debugging profiles. Profile A was the largest one ($N=323$; 52.69% of the sample) and was characterized by high debugging success rates and short debugging time in all types of errors. Thus, this profile was labeled Higher Performance. Profile B ($N=173$; 28.22% of the sample) was characterized by low debugging success rates and long debugging time for test errors. We labeled this profile Lower Test Performance. Profile C ($N=117$; 19.09% of the sample) was characterized by low debugging success rates and long debugging time for checkstyle and compiler errors

Table 3 Profiles' means and standard deviations at the raw scale of each variable

Profile	DSR Check	DSR Syntax	DSR Test	DT Check	DT Syntax	DT Test	Change rate	Multiple-edit rate
A	0.88 (0.08)	0.85 (0.09)	0.54 (0.14)	0.50 (0.17)	0.56 (0.21)	2.82 (0.97)	0.08 (0.03)	0.44 (0.19)
B	0.85 (0.06)	0.81 (0.07)	0.40 (0.08)	0.80 (0.29)	0.87 (0.28)	4.64 (1.61)	0.07 (0.02)	0.42 (0.11)
C	0.80 (0.11)	0.77 (0.10)	0.49 (0.12)	1.12 (0.53)	1.20 (0.55)	3.92 (1.48)	0.10 (0.04)	0.38 (0.19)

Note. The unit of time is minutes. DSR: debugging success rate for a type of error. DT: debugging time for one error. Profiles A, B, and C: Higher Performance, Lower Test Performance, as well as Lower Syntactic Performance and Higher Change Rate.

as well as high change rates in code size. As both checkstyle and compiler errors were about syntax, we labeled this profile Lower Syntactic Performance and Higher Change Rate.

Note that we used *lower* and *higher* rather than *low* and *high* in profile names. The reason was that all students in this study were novices in debugging. The characteristics of each profile were relative to the other profiles. Using *low* and *high* in profile names may overstate the meanings of profiles. Indeed, the Higher Performance profile's variable means at the raw scale (Table 3) suggested that students in this profile had much room for improvement. For instance, this profile's debugging success rate for test errors was 0.54, indicating that, on average, students needed two submissions to fix a test error.

At the raw scale, profiles' differences in debugging success rates were moderate. However, the differences in debugging time were large. On average, the time for fixing a checkstyle or syntax error in profile A was half of that in profile C. The time for fixing a test error in profile A was three fifths of that in profile B. Such differences suggest that all profiles might be able to fix errors that they encountered, but profiles B and C might use a time-inefficient approach.

5.2 RQ2: Prediction of debugging profiles by background variables

Table 4 presents the results of the latent variable multinomial logistic regression that examined the relationship between background variables and debugging profile membership. Three effects were statistically significant and are shown in bold in the table. The odds of belonging to profiles A vs. B for male students were 2.73 times the odds for female students. The odds of belonging to profiles A vs. B increased 1.60 times when the self-rated programming ability increased by one unit. The odds of belonging to profiles A vs. C increased 2.04 times when the self-rated programming ability increased by one unit.

Major, grade, and time commitment did not predict debugging profile membership. Language familiarity had a seemingly strong effect on debugging profile membership. For example, the odds of belonging to profiles A vs. B for students familiar with Java were 24.30 times the odds for students unfamiliar with any programming language. However, these effects had large standard errors and were not statistically significant. Thus, we considered the effect of language familiarity null.

Table 4 Results of multinomial logistic regression for the effects of predictors on debugging profile membership

	Profiles A vs. B[a]		Profiles A vs. C[a]		Profiles B vs. C[a]	
Predictor	β (SE)	OR	β (SE)	OR	β (SE)	OR
Gender[b]	**1 (0.33)****	2.73	0.5 (0.34)	1.64	-0.51 (0.35)	0.60
CS[c]	0.28 (0.41)	1.32	0.7 (0.52)	2.02	0.43 (0.56)	1.54
CS+[c]	0.01 (0.36)	1.01	-0.32 (0.36)	0.72	-0.33 (0.4)	0.72
Grade[d]	0.52 (0.47)	1.68	0.21 (0.42)	1.23	-0.31 (0.44)	0.73
Ability	**0.47 (0.18)****	1.60	**0.71 (0.21)****	2.04	0.24 (0.25)	1.28
Others[e]	2.21 (2.32)	9.14	2.48 (2.22)	11.88	0.26 (0.53)	1.30
Java[e]	3.19 (2.3)	24.30	2.47 (2.21)	11.81	-0.72 (0.62)	0.49
Java and others[e]	3.38 (2.31)	29.26	3.13 (2.22)	22.87	-0.25 (0.62)	0.78
Commitment: 5 to 10 hours[c]	-0.36 (0.46)	0.70	0.21 (0.44)	1.24	0.57 (0.5)	1.77
Commitment: > 10 hours[c]	-0.25 (0.34)	0.78	0.15 (0.32)	1.16	0.4 (0.39)	1.49

Note. Profiles A, B, and C: Higher Performance, Lower Test Performance, as well as Lower Syntactic Performance and Higher Change Rate. β: unstandardized coefficient. SE: standard error. OR: odds ratio of another profile to the reference profile. Positive coefficients (or OR > 1) indicates that the higher the value of the variable, the less likely it is to be in the reference profile. a: The reference profile. b: Dummy variables with female = 0. c: Dummy variables with the group of non-CS-related majors = 0. d: Dummy variables with first-year students = 0. e: Dummy variables with the group unfamiliar with any language = 0. f: Dummy variables with the group that committed less than 5 h per week = 0.

5.3 RQ3: Exam performance by debugging profiles

Figure 3 depicts the model that we used to test the differences in exam scores between different debugging profiles. The effect of background variables on exam scores was controlled and not of interest. The focus was the effect of debugging profiles on exam scores. First, we applied model 1, where the three exam scores were only correlated. Table 5 presents the results. Profiles B and C had no difference in any exam. Profile A had higher scores in all exams than the other profiles, with medium to large effect sizes (Cohen's $d = 0.55 \sim 1.02$). The differences between profile A and the other profiles were larger in the second exam than in the first exam. This raised the question of whether their score differences grew larger between the first and second exams.

To investigate this question, we applied model 2 in Fig. 3, where the second and third exam scores were regressed on the first exam. Table 6 displays the results. With the first exam score being controlled, profile A and the others still had differences in the second and third exams, with a small to medium effect size (Cohen's $d = 0.42 \sim 0.76$). The results suggest that profiles' differences in exam performance persisted over the course and might enlarge after the first exam.

5.4 RQ4: Debugging profile transitions

Results in this section are about the debugging profiles in phase two (between the first and second exams) and phase three (after the second exam) as well as profile transitions from phases one to three. Based on the same procedure in Sect. 4.1, we adopted

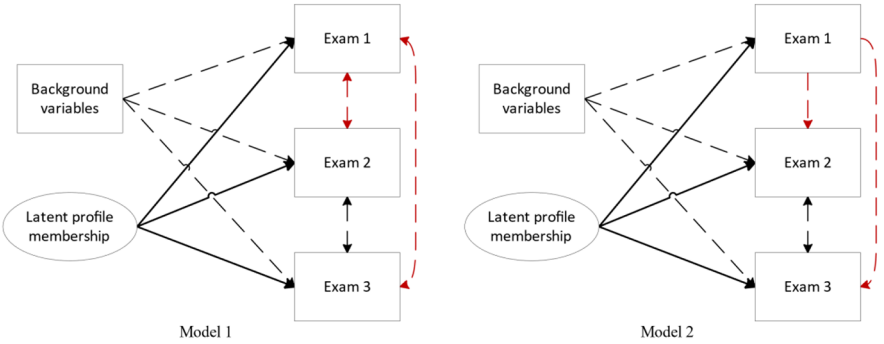

Fig. 3 Models 1 (left) and 2 (right) for the equality test of exam scores by debugging profiles. *Note*. Dashed lines are only for the purpose of controlling variables and not of interest. Single-headed arrows represent regression, while double-headed arrows represent correlations. The arrows from exam 1 to exams 2 and 3 (in red) are single-headed in model 2, representing that the influence of exam 1 on exams 2 and 3 is controlled. These arrows are double-headed in model 1, so the influence is not controlled.

Table 5 Equality tests of exam scores by debugging profiles in model 1

Exam	Adjusted mean (SD)			Difference & Cohen's *d*		
	Profile A	Profile B	Profile C	Profiles A vs. B	Profiles A vs. C	Profiles B vs. C
E1	80.56 (9.79)	71.94 (13.92)	74.68 (12.75)	**8.62*******0.75**	**5.88******0.55**	-2.74 -0.20
E2	87.33 (13.38)	71.64 (18.22)	76.42 (16.57)	**15.69*******1.02**	**10.92*******0.76**	-4.77 -0.27
E3	88.52 (13.98)	76.44 (17.38)	79.65 (16.56)	**12.09*******0.79**	**8.87******0.60**	-3.22 -0.19

Note. The means of exam scores were under the condition of adjusting the effect of background variables. Profiles A, B, and C: Higher Performance, Lower Test Performance, as well as Lower Syntactic Performance and Higher Change Rate.

Table 6 Equality tests of exam scores by debugging profiles in model 2

Exam	Adjusted mean (SD)			Difference & Cohen's *d*		
	Profile A	Profile B	Profile C	Profiles A vs. B	Profiles A vs. C	Profiles B vs. C
E2	33.81 (10.72)	23.73 (16.69)	26.76 (14.43)	**10.09*******0.76**	**7.06******0.60**	-3.03 -0.19
E3	37.88 (11.46)	31.09 (15.92)	32.62 (14.94)	**6.80******0.51**	**5.27*****0.42**	-1.53 -0.1

Note. The means of exam scores were under the condition of adjusting the effect of background variables and the first exam. Profiles A, B, and C: Higher Performance, Lower Test Performance, as well as Lower Syntactic Performance and Higher Change Rate.

a three-profile solution for the data of phase two and a two-profile solution for the data of phase three (the Appendix provides a detailed justification for using these solutions). The three debugging profiles in phase two were close to the three profiles in phase one (the left plot in Fig. 4). Profile A (35.90% of the sample) in phase two showed high debugging success rates and short debugging time in all types of errors,

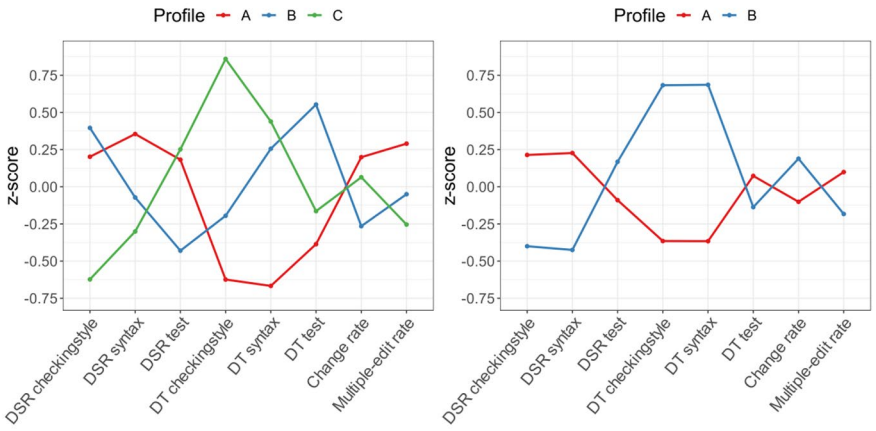

Fig. 4 Average variable z-scores of profiles in phase two (left plot) and phase three (right plot). *Note*. DSR: debugging success rate for a type of error. DT: debugging time for one error. Change rate: mean absolute change rate in code size. Multiple-edit rate: when the first submission has more than one syntax error, the probability that a student edited multiple error places, no matter fixing them or not.

same as profile A in phase one. This profile also had a higher multiple-edit rate for compiler errors than the others, but the difference was smaller than the differences in debugging success rates and time. Thus, we labeled this profile Higher Performance, same as profile A in phase one. We labeled profile B in phase two Lower Test Performance, the same as profile B in phase one, because profile B in phase two (34.60% of the sample) was characterized by low debugging success rates and long debugging time for test errors. Profile C (29.51% of the sample) of phase two was characterized by low debugging success rates and long debugging time for checkstyle and compiler errors and labeled Lower Syntactic Performance. In phase two, profile A had higher scores in the second exam than profiles B and C (score difference = 5.06 and 9.18, Cohen's $d = 0.41$ and 0.65, $p = 0.047$ and 0.001), controlling for background variables and the first exam scores. Profiles B and C had no statistically significant difference (score difference = 4.12, Cohen's $d = 0.27$, $p = 0.064$).

Profiles A (67.05% of the sample) and B (32.95% of the sample) in phase three had a small difference in debugging test errors, change rate in code sizes, and multiple-edit rate for syntax errors (right plot in Fig. 4). Their main differences were in debugging checkstyle and syntax errors. Profile A had much higher debugging success rates and shorter debugging time for these errors than profile B. Thus, we labeled Profiles A and B as Higher and Lower Syntactic Performance, respectively. Profile A of phase three had higher scores in the third exam than profile B (difference = 4.78, Cohen's $d = 0.35$, $p = 0.013$), controlling for background variables and the first exam scores.

Figure 5 depicts how students' debugging profile memberships changed across phases. From phases one to two, students were likely to stay in the same profiles. Students in profiles A, B, and C of phase one had 0.59, 0.57, and 0.59 probabilities of staying in profiles A, B, and C in phase two, respectively. Students in profile A of phase one were more likely to stay in profile A in phase two than transitioning into profile B (compared with profile B of phase one, odds ratio = 29.93, $p = 0.002$; com-

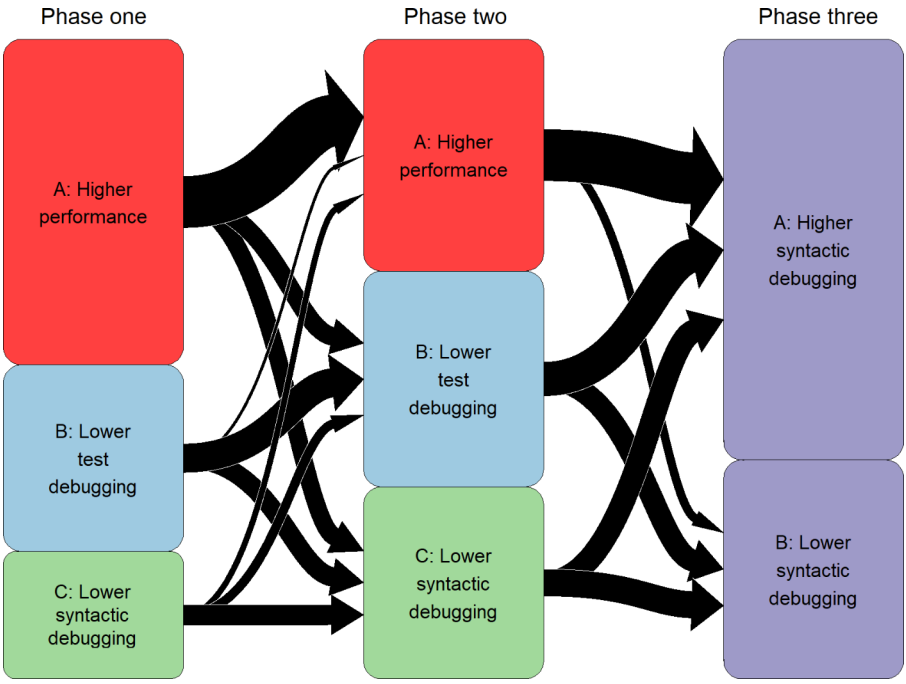

Fig. 5 Transitions among debugging profiles across phases. *Note*. The height of a box is proportional to the percentage of students in the corresponding profile. The thickness of an arrow is proportional to the percentage of students in the corresponding transition. For example, the arrow from profile A of phase one to profile A of phase two is much thicker than the others, indicating that a large proportion of students in profile A at phase one stayed in profile A at phase two.

pared with profile C of phase one, odds ratio = 4.72, $p = 0.02$) or profile C (compared with profile B of phase one, odds ratio = 20.09, $p = 0.004$; compared with profile C of phase one, odds ratio = 10.79, $p < 0.001$). Students in profiles B and C of phase one had no difference in the probability of transitioning into profile A in phase two. Compared with students in profile C, students in profile B of phase one were more likely to stay in profile B in phase two than transitioning into profile C (odds ratio = 3.40, $p = 0.016$).

Students in all profiles of phase two had a good probability of transitioning into profile A of phase three (0.87, 0.61, and 0.46 for profiles A, B, and C, respectively), possibly because this profile accounted for 67.05% of the sample. Students in profile A of phase two were more likely to transition into profile A in phase three (compared with profile B of phase two, odds ratio = 4.09, $p = 0.003$; compared with profile C of phase two, odds ratio = 7.50, $p < 0.001$). Profiles B and C of phase two had no statistically significant difference in the probability of transitioning into profiles A vs. B in phase three (odds ratio = 1.83, $p = 0.087$).

Overall, students with higher debugging performance in phase one were likely to stay in the higher performance profiles over the duration of the course, while those with lower performance (no matter whether checkstyle, syntax, or test errors) in phase one were likely to stay in the lower performance profiles. Indeed, 60.48% of

students in profile A of phase three were from profile A of phase one, while 72.25% of students in profile B of phase three were from profiles B and C of phase one.

6 Discussion

6.1 Debugging profiles, transitions, and exam performance

This study found three distinctive debugging profiles at the beginning of a CS1 course (phase one), two of which showed lower debugging performance. Profile B was weaker at debugging runtime and logic errors (i.e., test errors), while profile C was weaker at debugging checkstyle and syntax errors. Prior work found that novices had more difficulty in debugging runtime and logic errors than syntax errors (Fitzgerald et al., 2008). Thus, we expected that students in profile B would outperform those in profile C on exams. However, the two profiles had the same performance on all exams. One possible explanation is that language-independent constructs, the cause of runtime and logic errors (Pea, 1986), were relatively simple at phase one, the beginning of the CS1 course of this study. The proficiency of debugging errors related to such simple constructs may not be more critical than the proficiency of debugging syntax errors. Indeed, at the end of the course (i.e., phase three), where language-independent constructs were relatively complicated, the difference in debugging runtime and logic errors between profiles was small (see the right plot in Fig. 4).

In phase one, profile A had higher debugging performance in all types of errors than the other profiles. Profile A's higher debugging proficiency contributed to higher performance in the first exam. This was expected because better debugging success rate and speed helped students solve problems more quickly and effectively. What is more interesting is that the difference in the first exam between profile A and the others persisted in the second and third exams, even when the first exam's performance was controlled. Results about debugging profile transitions over the duration of the course may provide an explanation for this result. Compared with others, students in profile A of phase one were more likely to transition into profiles with higher debugging performance in phases two and three. These higher performance profiles also outperformed the other profiles on exams of the corresponding phase.

Students in different profiles showed differences in debugging strategies. In phase one, profile C had higher change rates in code size. In phase two, profile A showed higher multiple-edit rate for syntax errors. Nevertheless, the differences in debugging strategies were smaller than differences in debugging success rates and time, indicating that the debugging strategies captured from the programming trace data were less useful in distinguishing novices in the current study. The reason might be that debugging strategies have weak relationships with debugging performance (i.e., debugging success rates and time). For instance, in phase one, the average absolute Pearson correlation between change rates in code size and debugging performance variables was 0.06, and the range of the absolute correlation was 0.02 to 0.13. The average absolute correlation for multiple-edit rate for syntax errors was 0.09, and the range was 0.04 to 0.21. By contrast, the average absolute correlation among debugging performance variables was 0.28, and the range was 0.13 to 0.61. The weak cor-

relation between debugging strategies and performance suggests that novices may not apply these strategies effectively or that they may not intentionally apply these strategies, given that these strategies were not emphasized in the course.

Students in this study rarely used print statements. This is inconsistent with prior studies (Fitzgerald et al., 2010; Murphy et al., 2008; Whalley et al., 2021b). The reason may be that the task in the current study was different from prior studies, where participants located and fixed errors of particularly designed buggy programs. In the current study, students debugged their own programs. Because students might already be familiar with their programs, they may have been less likely to use print statements to trace their codes. The difference between current and prior studies matches prior findings that novices used different approaches when debugging their own programs versus others' programs (Katz & Anderson, 1987). These results together call for caution on generalizing findings between participants debugging their own programs and debugging others' programs.

6.2 Background variables and debugging profiles

Self-rated programming ability predicted debugging profiles in phase one. The higher a student self-rated their programming ability, the more likely they were to belong to profile A (Higher Performance). This is expected because students with high programming ability might have good domain and system knowledge and programming experiences, which are helpful for debugging (Li et al., 2019). However, this result was unexpectedly strong. The odds of belonging to profiles 1 vs. 2 for students with the highest programming ability (i.e., 5) was 10.49 times the odds for students with the lowest programming ability (i.e., 1), and the odds ratio of belonging to profiles 1 vs. 3 between the two groups was even larger, increasing to 35.33. The strong effect was partially inconsistent with prior findings that good programmers are not necessarily good debuggers (Ahmadzadeh et al., 2005; Fitzgerald et al., 2008). A possible explanation is that debugging performance was evaluated differently in prior studies compared to the current study. Prior work evaluated debugging performance based on students' final solutions for buggy programs and human observations of the processes generating the solution, which provided finer and more comprehensive information about debugging ability than the current study. Thus, the current study may not fully capture students' debugging ability due to the limited information of submission traces.

Gender predicted novices' debugging performance at phase one. This is unsurprising because gendered disaprities are known in CS1 (Lewis et al., 2019). Women's underrepresentation in CS has long existed, and gendered CS stereotypes is prevelantent. Girls may be less likely to participate in CS learning opportunities due to lower expectations from teachers and parents. What was unexpected is that the gendered difference in debugging performance mainly existed in test errors. Compared with female students, male students were more likely in profile A (Higher Performance) versus profile B (Lower Test Performance). However, when profile C (Lower Syntactic Performance and Higher Change Rate) was the reference group, female and male students had the same probability of belonging to profile A. Prior work found that male and female students applied different strategies during debugging (Subrah-

maniyan et al., 2008). In the current study, male and female students might employ debugging strategies differentially in ways not captured by our analyses, resulting in different performance in debugging test errors. Nevertheless, further investigation needs to replicate this result and, if it can be replicated, explore what kinds of debugging strategies cause the difference in debugging performance.

It is interesting that language familiarity did not predict debugging profiles. Students who reported familiarity with Java (groups of *Java* as well as *Java and others*) should have more domain knowledge of debugging than those unfamiliar with any programming language in the current study (group *none*). We expected that groups of *Java* as well as *Java and others* would be more likely to belong to profile A than group *none*. One possible explanation is that debugging entails various knowledge and skills (Decasse & Emde, 1988). Groups with prior expertise with *Java*, as well as *Java and others*, might not be better than group *none* in the system knowledge, procedural knowledge, and debugging strategies (Li et al., 2019). Overall, the results suggest that it may be better to not adapt debugging instruction to language familiarity, current grade level, or major, as none of these factors predicted profile memberships.

Students' expected time commitment was also unrelated to debugging profiles. The reason may be that their actual time investment did not match their expected commitment. Alternatively, most of their time effort might be spent on learning about computing concepts and Java language, given that debugging was not emphasized in the course.

6.3 Implications

In this study, novice programmers had both quantitative and qualitative differences in debugging performance and strategies at the beginning of the course. The difference in debugging extended to exam performance, and the difference in exam performance increased from the first to second exam. These results support the call for teaching debugging explicitly and early (Chmiel & Loui, 2004; Li et al., 2019; Michaeli & Romeike, 2019). Debugging is a frustrating process for novices (Whalley et al., 2021b). Frequent unsuccessful debugging may damage self-efficacy, causing persisting and enlarged programming and debugging disparities. At the early stage of CS1, explicitly teaching some debugging strategies may help novices locate and fix programming errors, reduce negative affective experiences, and facilitate learning.

Nevertheless, debugging is a complex skill. At the early stage of CS1, novices are weak at programming language knowledge and language-independent constructs and lack the knowledge base of stereotyped bugs and their symptoms. Thus, even equipped with debugging strategies, novices may still find debugging challenging. Researchers may consider using scaffolding to reduce the difficulty of debugging. For instance, programming error messages are usually vague, unclear, and hard to read (Becker et al., 2019). Improving the usability and readability of error messages may facilitate efficient and successful debugging (Denny et al., 2020). In addition, the source of test errors is difficult to identify. As such, it may be useful to provide students with explicit guidance about how to apply debugging strategies to help them identify the source of an error. Of course, as novices' knowledge and skills grow, it is

necessary to fade the scaffolding so that the novices can practice debugging opportunities in a more authentic environment.

Additionally, novices often find error messages frustrating and discouraging (Becker et al., 2016). Researchers and instructors may consider adding positive information to error messages to motivate students, such as praising achievements when errors are fixed and providing motivation when the same errors persist (Marwan et al., 2020). Such attempts may not only support the debugging process but also cultivate students' belief of "debugging as productive failure" (Kafai et al., 2019).

6.4 Limitations and future directions

The web-based learning system in this study did not have debugging functions common in IDEs, such as break point and memory inspection. With such functions and instruction on utilizing them, students might have better debugging performance. Additionally, the programming language was Java, which typically has no code style requirement. However, the instructor required a certain code style, and violating the requirement in this study would result in checkstyle errors. Such errors may not be common in Java programming courses. Thus, the current result may not generalize to an environment with debugging functions and without code style requirements. Future work should be undertaken to investigate novices' debugging differences in different environments and whether the differences are consistent across environments.

This study examined the transitions among debugging profiles during the course. However, it was not a direct investigation of how students' debugging abilities grew over time. We were unable to do this because the programming assignments became increasingly difficult and complex. The debugging performance measures we calculated based on the assignment-solving process would be uncomparable across assignments at different phases of the course. Further studies might design debugging tasks and present them to novices at different phases of CS1 courses to investigate the development of novices' debugging abilities. Such investigation may not only improve conceptual understanding of the development but also assist in developing best practice for instruction. For example, without instruction on debugging strategies, the debugging performance may still grow at the early phase of CS1 because of the growth in programming abilities. However, without explicit instruction, the growth in debugging may plateau. Teaching debugging strategies early fits students' timely needs, and thus, it is worth identifying when the plateau arrives in future studies.

7 Conclusion

The current study conducted a person-centered investigation of debugging in a CS1 course. Combining programming traces and LPA, we identified three distinctive debugging profiles at the beginning of the course. Profile A showed higher debugging accuracy and speed. Profile B showed lower debugging performance in runtime and logic errors, while profile C had lower performance in syntactic errors and tended to make large code edit every submission. Students' gender and self-rated program-

ming ability predicted profile membership. Moreover, profile A got higher scores than the others in the first exam, and this difference persisted in the second and third exams. Latent transition analysis found that a large part of students stayed in lower debugging performance profiles over time. Overall, these findings support the call that debugging should be taught at an early stage and provide guidance for personalized debugging instructions.

8 Appendix

Table A1 presents the statistical fit indices and tests for LPA models for the data of phase two. Models with four and six profiles did not converge, and their fit indices could not be obtained. CAIC and SABIC decreased as the number of profiles increased. BIC reached the minimum at the model with three profiles. aLMR was statistically insignificant when the number of profiles was greater than two, indicating that the model with three profiles was not superior to the model with two profiles. BLMRT was significant for all models and not useful for determining the appropriate number of profiles. Thus, a model with two or three profiles might be the best solution. Further inspection found that the model with three profiles included three distinctive profiles. This indicates that the model with three profiles provided more insights than the model with two profiles. Besides, the three profiles in phase two were close the three profiles in phase one. Thus, we adopted the three-profile solution for the data of phase two.

Table A1 Model fit statistics for the data of phase two (between the first and second exams)

Model	# FP	LL	CAIC	BIC	SABIC	aLMR p and meaning		BLRT p and meaning		Entropy
1	44	-6488.61	13072.23	13259.42	13119.73	NA		-		-
2	61	-6322.82	12781.45	13036.87	12843.20	0.024	2>1	0.000	2>1	0.647
3	78	-6247.55	12674.30	12995.34	12747.71	0.119	3>2	0.000	3>2	0.606
5	112	-6147.51	12569.94	13013.32	12657.74	0.126	5<4	0.000	5>4	0.695

Table A2 Model fit statistics for the data of phase three (after the second exams)

Model	# FP	LL	CAIC	BIC	SABIC	aLMR p and meaning		BLRT p and meaning		Entropy
1	44	-6541.07	13177.27	13363.69	13224.00	-		-		-
2	61	-6274.34	12684.72	12939.00	12745.34	0.000	2>1	0.000	2>1	0.706
3	78	-6170.01	12519.62	12839.10	12591.47	0.146	3>2	0.000	3>2	0.727
4	95	-6101.41	12428.93	12810.68	12509.08	0.538	4<3	0.000	4>3	0.692
5	112	-6054.85	12385.57	12826.34	12470.77	0.334	5<4	0.030	5>4	0.695
6	129	-6018.75	12366.71	12862.91	12453.37	0.002	6<5	0.148	6<5	0.729

Table A2 presents the statistical fit indices and tests for LPA models for the data of phase three. Like Table A1, Table A2 suggests that a model with two or three profiles might be the best solution. However, further inspection found that the model with three profiles split a profile in the model with two profiles into one large (59.4%)

and one small profiles (7.32%), which only had statistically significant difference in debugging time for test errors (the smaller profile showed shorter time). Given the greatly unbalanced profile sizes, we think that the two profiles were not distinctive in a meaningful way. The model with three profiles might just select those with short debugging time for test errors and group them into a profile. Thus, we adopted the two-profile solution for the data of phase three.

Authors' contributions Yingbin Zhang: Conceptualization, methodology, formal analysis, writing - original draft preparation, review, and editing. **Luc Paquette**: Conceptualization, methodology, writing - review and editing, supervision. **Juan Pinto**: Conceptualization, writing - review and editing. **Sophie Qianhui Liu**: Conceptualization, writing - review and editing. **Aysa Xuemo Fan**: Conceptualization, writing - review and editing.

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

Data Availability 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.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. (pp. 84–88), Caparica, Portugal. https://doi.org/10.1145/1067445.1067472

Alqadi, B. S., & Maletic, J. I. (2017). An empirical study of debugging patterns among novices programmers. In *Proceedings of the 48th ACM Technical Symposium on Computer Science Education-SIGCSE '17*. (pp. 15–20). http://doi.org/10/ghqjp4

Asparouhov, T., & Muthén, B. (2014). Auxiliary variables in mixture modeling: Three-step approaches using mplus. *Structural Equation Modeling: A Multidisciplinary Journal*, *21*(3), 329–341. https://doi.org/10.1080/10705511.2014.915181

Bakk, Z., & Vermunt, J. K. (2016). Robustness of stepwise latent class modeling with continuous distal outcomes. *Structural Equation Modeling: A Multidisciplinary Journal*, *23*(1), 20–31. https://doi.org/10.1080/10705511.2014.955104

Baumstark, L., & Orsega, M. (2016). Quantifying introductory CS students' iterative software process by mining version control system repositories. *Journal of Computing Sciences in Colleges*, *31*(6), 97–104. https://doi.org/10.5555/2904446.2904470

Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, C. (2016). Effective compiler error message enhancement for novice programming students. *Computer Science Education*, *26*(2–3), 148–175. https://doi.org/10.1080/08993408.2016.1225464

Becker, B. A., Murray, C., Tao, T., Song, C., McCartney, R., & Sanders, K. (2018). Fix the first, ignore the rest: Dealing with multiple compiler error messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. (pp. 634–639). http://doi.org/10/gg7fkb

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., Osera, P., Pearce, J. L., & Prather, J. (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. (pp. 177–210), Aberdeen, Scotland Uk. https://doi.org/10.1145/3344429.3372508

Beller, M., Spruit, N., Spinellis, D., & Zaidman, A. (2018). On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. (pp. 572–583), Gothenburg, Sweden. https://doi.org/10.1145/3180155.3180175

Bezdek, J. C., Ehrlich, R., & Full, W. (1984). FCM: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, *10*(2), 191–203. https://doi.org/10.1016/0098-3004(84)90020-7

Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, *23*(4), 561–599. http://doi.org/10/gfz7xk

Chmiel, R., & Loui, M. C. (2004). Debugging: From novice to expert. *SIGCSE Bulletin*, *36*(1), 17–21. https://doi.org/10.1145/1028174.971310

Collins, L. M., & Lanza, S. T. (2009). *Latent class and latent transition analysis: With applications in the social, behavioral, and health sciences*. John Wiley & Sons. https://doi.org/10.1002/9780470567333

Decasse, M., & Emde, A. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings of the 11th International Conference on Software Engineering (ICSE '88)*. (pp. 162–171). https://doi.org/10.1109/ICSE.1988.93698

DeLiema, D., Dahn, M., Flood, V., Asuncion, A., Abrahamson, D., Enyedy, N., & Steen, F. (2019). Debugging as a context for fostering reflection on critical thinking and emotion *Deeper Learning, Dialogic Learning, and Critical Thinking* (pp. 209–228). https://doi.org/10.4324/9780429323058-13

Denny, P., Prather, J., & Becker, B. A. (2020). Error message readability and novice debugging performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. (pp. 480–486), Trondheim, Norway. https://doi.org/10.1145/3341525.3387384

Enders, C. K., & Bandalos, D. L. (2001). The relative performance of full information maximum likelihood estimation for missing data in structural equation models. *Structural Equation Modeling: A Multidisciplinary Journal*, *8*(3), 430–457. https://doi.org/10.1207/S15328007SEM0803_5

Eranki, K. L. N., & Moudgalya, K. M. (2014). Application of program slicing technique to improve novice programming competency in spoken tutorial workshops. In *2014 IEEE Sixth International Conference on Technology for Education*. (pp. 32–35). https://doi.org/10.1109/T4E.2014.1

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging from the student perspective. *Ieee Transactions On Education*, *53*(3), 390–396. http://doi.org/10/b7nbm5

Gugerty, L., & Olson, G. (1986). Debugging by Skilled and Novice Programmers. *SIGCHI Bull.*, *17*(4), 171–174. http://doi.org/10/bdzt57

Hickendorff, M., Edelsbrunner, P. A., McMullen, J., Schneider, M., & Trezise, K. (2018). Informative tools for characterizing individual differences in learning: Latent class, latent profile, and latent transition analysis. *Learning and Individual Differences*, *66*, 4–15. https://doi.org/10.1016/j.lindif.2017.11.001

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull*, *35*(1), 153–156. https://doi.org/10.1145/792548.611956

Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., Edwards, S. H., Isohanni, E., Korhonen, A., Petersen, A., Rivers, K., Rubio, M., Sheard, J., Skupas, B., Spacco, J., Szabo, C., & Toll, D. (2015). Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*. (pp. 41–63), Vilnius, Lithuania. https://doi.org/10.1145/2858796.2858798

Jemmali, C., Kleinman, E., Bunian, S., Almeda, M. V., Rowe, E., & El-Nasr, M. S. (2020). MAADS: Mixed-methods approach for the analysis of debugging sequences of beginner programmers. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* (pp. 86–92). Portland, OR, USA. https://doi.org/10.1145/3328778.3366824

Jiang, B., Zhao, W., Zhang, N., & Qiu, F. (2019). Programming trajectories analytics in block-based programming language learning. *Interactive Learning Environments*, 1–14. https://doi.org/10.1080/10494820.2019.1643741

Kafai, Y. B., DeLiema, D., Fields, D. A., Lewandowski, G., & Lewis, C. (2019). Rethinking debugging as productive failure for CS education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. (pp. 169–170), Minneapolis, MN, USA. https://doi.org/10.1145/3287324.3287333

Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, *3*(4), 351–399. http://doi.org/10/bf6gh8

Kazerouni, A. M., Edwards, S. H., Hall, T. S., & Shaffer, C. A. (2017). DevEventTracker: Tracking development events to assess incremental development and procrastination. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. (pp. 104–109), Bologna, Italy. https://doi.org/10.1145/3059009.3059050

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, *20*(3), 362–404. https://doi.org/10.1016/0010-0285(88)90004-7

Larman, C., & Basili, V. R. (2003). Iterative and incremental developments: A brief history. *Computer*, *36*(6), 47–56. https://doi.org/10.1109/MC.2003.1204375

Lewis, C. M. (2012). The importance of students' attention to program state: A case study of debugging behavior. In *Proceedings of the ninth annual international conference on International computing education research*. (pp. 127–134), Auckland, New Zealand. https://doi.org/10.1145/2361276.2361301

Lewis, C. M., Shah, N., & Falkner, K. (2019). Equity and diversity. In A. V. Robins, & S. A. Fincher (Eds.), *The Cambridge handbook of computing education research* (pp. 481–510). Cambridge University Press. http://doi.org/. DOI

Li, C., Chan, E., Denny, P., Luxton-Reilly, A., & Tempero, E. (2019). Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*. (pp. 79–86), Sydney, NSW, Australia. https://doi.org/10.1145/3286960.3286970

Lin, Y., Wu, C., Hou, T., Lin, Y., Yang, F., & Chang, C. (2016). Tracking students' cognitive processes during program debugging—An eye-movement approach. *IEEE Transactions on Education*, *59*(3), 175–186. https://doi.org/10.1109/TE.2015.2487341

Liu, Z., Zhi, R., Hicks, A., & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*, *27*(1), 1–29. https://doi.org/10.1080/08993408.2017.1308651

Malik, S. I., & Coldwell-Neilson, J. (2017). A model for teaching an introductory programming course using ADRI. *Education and Information Technologies*, *22*(3), 1089–1120. https://doi.org/10.1007/s10639-016-9474-0

Marwan, S., Dombe, A., & Price, T. W. (2020). Unproductive help-seeking in programming: What it is and how to address it. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. (pp. 54–60), Trondheim, Norway. https://doi.org/10.1145/3341525.3387394

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

Michaeli, T., & Romeike, R. (2019). Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*. (pp. 15), Glasgow, Scotland, Uk. https://doi.org/10.1145/3361721.3361724

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: The good, the bad, and the quirky–a qualitative analysis of novices' strategies. In. (pp.163–167)

Muthén, B. (2004). Latent variable analysis: Growth mixture modeling and related techniques for longitudinal data. In D. Kaplan (Ed.), *The SAGE handbook of quantitative methodology for the social sciences* (pp. 345–368)

Nylund-Gibson, K., Grimm, R., Quirk, M., & Furlong, M. (2014). A latent transition mixture model using the three-step specification. *Structural Equation Modeling: A Multidisciplinary Journal*, *21*(3), 439–454. https://doi.org/10.1080/10705511.2014.915375

Oberski, D. (2016). Mixture models: Latent profile and latent class analysis. In J. Robertson, & M. Kaptein (Eds.), *Modern Statistical Methods for HCI* (pp. 275–287). Springer International Publishing. https://doi.org/10.1007/978-3-319-26633-6_12

Pea, R. D. (1986). Language-independent conceptual "Bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. (pp. 213–229), Washington, D.C., USA

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, *2*(1), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

Perscheid, M., Siegmund, B., Taeumel, M., & Hirschfeld, R. (2017). Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, *25*(1), 83–110. https://doi.org/10.1007/s11219-015-9294-2

Pinto, J. D., Zhang, Y., Paquette, L., & Fan, A. X. (2021). Investigating elements of student persistence in an introductory computer science course. In *Joint Proceedings of the CSEDM Workshops at the 14th International Conference on Educational Data Mining*. Virtual, Online. Retrieved from http://ceur-ws.org/Vol-3051/CSEDM_2.pdf

Rich, K. M., Strickland, C., Binkowski, T. A., & Franklin, D. (2019). A K-8 debugging learning trajectory derived from research literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. (pp. 745–751), Minneapolis, MN, USA. https://doi.org/10.1145/3287324.3287396

Rich, K. M., Yadav, A., & Larimore, R. A. (2020). Teacher implementation profiles for integrating computational thinking into elementary mathematics and science instruction. *Education and Information Technologies*, *25*(4), 3161–3188. https://doi.org/10.1007/s10639-020-10115-5

Spinellis, D. (2018). Modern debugging: the art of finding a needle in a haystack. *Communications of the Acm*, *61*(11), 124–134. https://doi.org/10.1145/3186278

Spurk, D., Hirschi, A., Wang, M., Valero, D., & Kauffeld, S. (2020). Latent profile analysis: A review and "how to" guide of its application within vocational behavior research. *Journal of Vocational Behavior*, *120*, 103445. https://doi.org/10.1016/j.jvb.2020.103445

Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R., & Fern, X. (2008). Testing vs. code inspection vs. what else? male and female end users' debugging strategies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. (pp. 617–626), Florence, Italy. https://doi.org/10.1145/1357054.1357153

Tein, J., Coxe, S., & Cham, H. (2013). Statistical power to detect the correct number of classes in latent profile analysis. *Structural Equation Modeling: A Multidisciplinary Journal*, *20*(4), 640–657. https://doi.org/10.1080/10705511.2013.824781

Vermunt, J. K., & Magidson, J. (2002). Latent class cluster analysis. In A. L. McCutcheon, & J. A. Hagenaars (Eds.), *Applied Latent Class Analysis* (pp. 89–106). Cambridge University Press. https://doi.org/10.1017/CBO9780511499531.004

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494. http://doi.org/10/dmg64q

Whalley, J., Settle, A., & Luxton-Reilly, A. (2021a). Analysis of a process for introductory debugging. In *Australasian Computing Education Conference*. (pp. 11–20), Virtual, SA, Australia. https://doi.org/10.1145/3441636.3442300

Whalley, J., Settle, A., & Luxton-Reilly, A. (2021b). Novice reflections on debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. (pp. 73–79). https://doi.org/10.1145/3408877.3432374

West, M., Herman, G. L., & Zilles, C. (2015). PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. (pp. 26.1238.1-26.1238.14), Seattle, WA, USA. http://doi.org/10.18260/p.24575

Zeller, A. (2009). *Why programs fail: A guide to systematic debugging*. Elsevier

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Yingbin Zhang[1] · Luc Paquette[1] · Juan D. Pinto[1] · Qianhui Liu[1] · Aysa Xuemo Fan[1]

✉ Luc Paquette
lpaq@illinois.edu

Yingbin Zhang
yingbinzhang25@hotmail.com

Juan D. Pinto
jdpinto2@illinois.edu

Qianhui Liu
ql29@illinois.edu

Aysa Xuemo Fan
xuemof2@illinois.edu

[1] Department of Curriculum and Instruction, College of Education, University of Illinois at Urbana-Champaign, 383 Education Building, 1310 S Sixth Street, 61820 Champaign, IL, USA