From Bugs to Breakthroughs: Novice Errors in CS2

Nadja Just
Professorship of Software Engineering
University of Technology Chemnitz
Chemnitz, Germany
https://orcid.org/0009-0006-9076-0362

Belinda Schantong
Professorship of Software Engineering
University of Technology Chemnitz
Chemnitz, Germany
https://orcid.org/0009-0005-7091-0880

Janet Siegmund
Professorship of Software Engineering
University of Technology Chemnitz
Chemnitz, Germany
siegj@hrz.tu-chemntiz.de

***Abstract*—Background: Programming is a fundamental skill in computer science and software engineering specifically. Mastering it is a challenge for novices, which is evidenced by numerous errors that students make during programming assignments.**

Objective: In our study, we want to identify common programming errors in CS2 courses and understand how students evolve over time.

Method: To this end, we conducted a longitudinal study of errors that students of a CS2 course made in subsequent programming assignments. Specifically, we manually categorized 710 errors based on a modified version of an established error framework.

Result: We could observe a learning curve of students, such that they start out with only few syntactical errors, but with a high number of semantic errors. During the course, the syntax and semantic errors almost completely vanish, but logical errors remain consistently present.

Conclusion: Thus, students have only little trouble with learning the programming language, but need more time to understand and express concepts in a programming language.

***Index Terms*—CS2, Java, Computer science education, Novice Programmers.**

I. Introduction

Programming is a fundamental skill for software development. It is also a task with which many students struggle, as evidenced by the high failure rates in programming courses [1]. For beginners, it is especially challenging to grasp the linguistic nuances of programming languages, while also learning computational thinking [2]. This is one reason why it is particularly difficult for students to create programs without syntactic, semantic, or logical errors [3]–[5].

Syntactic programming errors refer to the basic rules of a programming language, such as incorrectly declaring a variable or additional closing parenthesis, while semantic errors go one step further, such as using a variable that has not been declared or incorrectly assigning a floating point value to a variable that can only hold natural numbers [3], [6]. Syntactic and many semantic errors can be detected by compiler error messages [3], [7]–[10]. However, some semantic errors, such as using == for comparison of objects instead of the `equals` method in Java [4], are not detected on compile time, but still depend on the context of the language.

Logical errors describe that a program does not behave according to its specification, but adheres to the syntactic and semantic rules of a programming language. An example would be a program that is supposed to identify prime numbers, but for some reason identifies 4 as a prime number. [3], [5]

Studies suggest that students still make numerous errors after completing a CS1 course, which even counts for basic syntactic errors [4], [11]–[13]. However, there is only little dedicated research on the specific kinds of errors students make in a CS2 course or how they evolve within one semester. This is a missed opportunity, because it would help educators understand the learning curves of students and how they could help them to overcome the threshold from novice programmer to professional programmer.

With our work, we want to seize that opportunity. To shed more light on student's errors and how they evolve in this particular phase of learning, we replicate a study from McCall and Kölling [11]. Specifically, we use their error categorization framework as a tool to explore and understand student errors. One advantage of this framework is its modular extensibility, which we exploited by adding categories that contain logical errors.

To this end, we analyze student submissions to programming assignments during a CS2 course, and run unit tests on them, so that we can automatically detect logical errors, in addition to compiler errors. We found that students start the course with a high number of semantic errors, and, when faced with new concepts, logical errors as well. This changes during the course, such that semantic errors only play a minor role. Thus, students struggle only at first with learning the semantic rules of the new programming language, and quickly adapt, so that they can focus on implementing a concept, less on how a programming language works.

We make the following contributions:

- A replication of the study by McCall and Kölling [11] to describe programming errors in a CS2 course.
- The addition of logical error categories to the framework.
- A longitudinal view to observe the development of students over the course of a semester.
- A publicly available replication package: https://github.com/CSEEandT-25-ErrorPaper/CS2-Error-Categorization

II. Related Work

The framework we based our work on was developed by McCall and Kölling based on thematic analysis of student

submissions that were automatically collected [14]. In subsequent studies, this framework was evaluated and refined [11]. Thus, it is a well-tested and established framework, making it the best choice for describing errors of students. It consists of 11 categories of errors, of which 4 describe syntactical errors and 6 semantic errors (one additional category contains unspecified errors). Each of the 11 categories is described by several subcategories. For example, the category *9.0 Simple syntactical errors* consists of subcategories, such as *9.8 Extraneous closing curly brace* and *9.3 ; missing*. In its original version, it does not contain categories for logical errors.

Several other frameworks exist, as well. Hristova and others were the first to propose a framework to describe students errors beyond compiler-based analysis [3]. Based on interviews with educators and students, they described the most common errors, which they divide into syntax, semantic, and logical errors, each of which are refined in several subcategories. This framework was tested and extended by Brown and Altamdri as well as Mase and Nel [5], [15]. However, it lacks the modular extensibility of the framework developed by McCall and Kölling, due to its flat hierarchy and more rigid categorization. Additionally, while McCall and Kölling's error categorization encompasses all error types identified by Brown and Altamdri, the reverse does not hold true [4], [11]. Their framework has fewer specific subcategories, while the super categories are not meant to contain unspecified programming errors.

There is considerable further research on how to categorize student errors, focusing on logical errors [16], static analysis errors [17], or compiler messages [8], [9]. However, none of these frameworks are sufficiently detailed or focused on concrete programming errors to describe the variety of student errors that can be observed.

III. Study Design

In this section, we present the study set up and research questions. All material is available at the project's Web site: https://github.com/CSEEandT-25-ErrorPaper/CS2-Error-Categorization.

A. Research Questions

Our overarching goal with this work is to understand students' errors and how they evolve during a CS2 course. This way, we can understand the obstacles face to become proficient programmers. To guide our study, we define the following research questions:

- RQ_1: What syntactic and semantic errors do students make?
- RQ_2: What logical errors do students make?

The error categories follow the framework by McCall and Kölling [11]. We added two further categories to describe logical errors, with 7 subcategories. Note that the framework contains the category *8.0 Semantic errors* and *9.0 Simple syntactical errors*. To clearly differentiate these categories from the type of errors, we set categories of the framework in *italics*, and when we talk of the broader categories of syntax, semantic, and logical errors, we do not set them in italics.

B. Participants

Of all the students enrolled in the CS2 course, we received the permission of 13 to analyze their solutions to mandatory programming assignments. Most have a background in C, Python, or both, from a preceding CS1 course.

C. Tasks

We defined seven tasks, which covered arrays, classes, lists, binary trees, and graphs. For example, in Task 2, students should implement five classes that interacted with each other to simulate a university in a small program.

The students received a task sheet with descriptions of the classes and methods they needed to implement, which included interfaces for the mandatory classes, and a source code file for each class that already implemented the interface and also provided the constructor declaration. This ensured the compatibility of the submissions with unit tests, which we used to assess the submissions to automatically detect logical errors. Students were encouraged to create additional classes and methods to structure their code.

The students had two weeks to solve each task and had to use the Java API, no third party libraries, but could use code from online or generative AI sources. The students were free to work with an editor of their choice, but we recommended an integrated development environment (IDE), specifically Visual Studio Code. Only compilable code would be graded.

D. Analysis protocol

We anonymized the submissions by assigning each student an ID. The first author proceeded to categorize the errors in each submission. If an error did not fit into any of the existing subcategories from [11], we sorted it into an *unspecified* category. If one of these appeared more than two times, we extended the original framework by a new subcategory.

To find the errors, we ran unit tests. If a unit test failed, we documented the cause, fixed it, and ran the tests again. We repeated this process until the submission passed all tests. To not miss any potential errors not captured by the unit tests, we manually evaluated each submission for further errors.

IV. Results

In total, we evaluated 67 submissions from 13 different students. We excluded one submission (Task 5 from S_4) from analysis, since the student ignored the provided interface and strayed too far from the intended task, making meaningful error categorization impossible.

Table I contains an overview of the number of errors per task. Most programming errors occurred in the first three tasks, with Task 2 being the most difficult task with 295 errors. A considerable drop in the total number of errors occurred after Task 3. This downward trend continued, except for the last task, which has slightly more errors than Task 6.

TABLE I
PROGRAMMING ERRORS PER CATEGORY AND TASK

Error category	Error type	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Sum
1.0 Incorrect attempt to use variable	Semantic	5	33	9	0	0	0	1	48
2.0 Incorrect variable declaration	Syntactic	2	1	0	0	0	0	0	3
3.0 Incorrect method call	Semantic	1	46	1	0	0	0	0	48
4.0 Incorrect method declaration	Syntactic	5	15	0	0	0	0	0	20
5.0 Incorrect constructor call	Semantic	0	4	0	0	0	0	0	4
6.0 Incorrect constructor declaration	Semantic	0	0	0	0	0	0	0	0
7.0 Incorrect/attempted use of class or type	Semantic	0	68	6	5	8	0	2	89
8.0 Semantic error	Semantic	105	66	31	5	1	4	5	217
9.0 Simple syntactical error	Syntactic	7	12	0	4	0	1	0	24
10.0 Statement outside method/block	Syntactic	1	1	0	0	0	0	0	2
11.0 Uncategorized	-	1	4	0	0	0	0	0	5
12.0 Logical error in working with data structures	Logic	0	5	83	28	13	11	1	141
13.0 Other logical error	Logic	0	40	22	23	9	1	14	109
Sum		127	295	152	70	31	17	23	710

A. RQ_1: What syntactic and semantic errors do students make?

Semantic errors are the most frequent error type. The number of errors was especially high for Tasks 1 and 2, before dropping in Task 3. From Task 4 onwards, they occurred only rarely. The majority of the *8.0 semantic errors* in Task 1 referred to an incorrect use of indexes (63 errors). A regularly occurring *8.0 semantic error* was using comparison operators (==) to compare strings instead of the method `equals`, which was one of the categories which we added to the framework.

The most frequent semantic category in Task 2 is *7.0 incorrect/attempted use of a class or type* (68 errors). The errors are mostly related to inheritance or interfaces, such as passing a class instead of an interface as parameter (13 errors), incorrect use of the `@Override` annotation (11 errors), changing the return type of an inherited method (9 errors), and general interface or inheritance implementation errors (9 errors).

Notably, students did not struggle with basic syntactic rules, such as *2.0 Incorrect variable declaration*, or *10.0 Statement outside method/block*, each of which occurred only rarely.

Looking at each student over time (cf. Table II), we observe two patterns: (1) students who continuously made only few errors, such as S_6 and S_9, and (2), students who made a lot of errors in the beginning and then either improved, such as S_3, S_5, and S_8, or stopped submitting, for example, S_2 and S_{11}. Notably, the high error count in the first three tasks is caused by few students. In Task 1, three students caused 58 % of the errors. This is even more pronounced in Task 2, in which two students made 70% of errors. And for Task 3, one student caused 63 % of the errors. Noticeably, these are never the same students, so each student only had one submission with a high error count.

Answer RQ1: Students seem to struggle mostly with semantic errors, especially when using classes. Basic syntactic structures do not seem to pose problems. Midway through the course, the error rate drops considerably, but some errors persist during the course.

TABLE II
PROGRAMMING ERRORS PER STUDENT. MEDIAN IN BRACES DENOTE VALUES WITHOUT S1, S7, AND S11.

Student ID	Task 1	2	3	4	5	6	7	Sum
S1	7	/	/	/	/	/	/	7
S2	18	95	16	/	/	/	/	129
S3	14	110	20	10	5	3	3	165
S4	2	25	7	10	n/a	2	6	52
S5	28	18	5	8	8	2	3	72
S6	2	5	2	8	2	1	4	24
S7	2	/	/	/	/	/	/	2
S8	22	0	3	1	1	0	1	28
S9	0	3	9	2	1	1	0	16
S10	9	12	13	14	4	3	4	59
S11	23	/	/	/	/	/	/	23
S12	/	10	66	6	3	2	0	87
S13	/	17	11	6	7	3	2	46
Median	9 (11.5)	14.5	10	8	3.5	2	3	46 (55.5)

B. RQ_2: What logical errors do students make?

Logical errors started appearing in Task 2. We show an overview of the logical errors in Table III. Overall, we observed 7 subcategories of logical errors. In Task 3, students made the most logical errors (105 errors). In the subsequent tasks, the amount of logical errors decreased by about half each task until Task 6 with 12 errors. There is no further drop to Task 7, but a slight increase, with most of the 15 errors relating to incomplete implementation of logic (12 errors). Starting from Task 3, logical errors were consistently the most frequent ones.

The majority of errors relate to the concept that was part of a task. Linked lists especially caused troubles for students, followed by working with classes. Typically, the amount of programming errors decreases the second time a concept was part of the implementation. One exception were classes, which caused the most trouble in the second task they occurred in.

TABLE III
LOGICAL ERRORS PER SUB CATEGORY AND TASK

Logical error	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Sum
12.0 Logical error in working with data structures							
12.1 ... in working with classes	5	42	5	0	0	0	52
12.3 ... in working with custom implemented linked lists	0	41	23	0	0	0	64
12.5 ... in working with a queue or a stack	0	0	0	2	0	0	2
12.2 ... in working with graphs	0	0	0	0	11	1	12
12.2.1 ... in working with a binary tree	0	0	0	11	0	0	11
13.0 Other logical error							
13.1 Incomplete logic in a function	11	10	14	2	0	12	49
13.2 Unspecified logical error	29	12	9	7	1	2	60
Sum	45	105	51	22	12	15	250

Answer RQ2: Students tend to make logical errors when implementing new data structures or concepts, but these errors decrease with repeated practice.

V. DISCUSSION

A. Learning Curve

The learning curve we observed can be summarized as follows: Students face only few problems when learning basic Java syntax. Instead, they start with a high rate of semantic errors, which relate to concepts that have no explicit syntactic equivalent in C (e.g., structs vs. classes) or behave a bit differently than in Java (e.g., type casting). This takes about the first half of the course, and, while students begin to make fewer semantic errors, logical errors are increasing. This learning curve is in line with studies demonstrating that students transfer their knowledge from one programming language to another [18], [19].

In other words, students quickly capture basic syntactic rules of Java, based on their knowledge of C, which they learned in the preceding CS1 course. Basic rules can be directly transferred from C to Java, such as variable declaration or that statements have to be within one block. Since these basic rules pose no troubles, students focus on learning how to transfer concepts that they already know, such as structs and static types, to how these concepts are expressed in Java. This takes some effort, because these concepts do not have the same syntactical representation. Especially the first tasks that introduce classes pose difficulties for students. In Task 2, the focus is on the newly introduced concept of classes, and we observe a peak of errors in the *incorrect/attempted use of class or type* category. This ripples through to Task 3, in which *logical errors in working with classes* occur most often. But from Task 4, there are no errors related to classes. Thus, once students grasped that structs and classes are comparable, they do not have to learn something entirely new, but learn how to express something that they already know in a new programming language. Then, they can start focusing on understanding and implementing new concepts.

Such an interpretation is supported by the development of the logical errors, which are the most frequent ones starting from Task 3. From that task on, students had to work with a new data structure every time, and for each task, the logical errors are related to the concrete data structure. Other studies have demonstrated similar peaks in errors around certain newly introduced concepts [5], [20], possibly hinting at them being threshold concepts, which are difficult to learn, but once grasped, cannot be unlearned [21], [22]. Concepts of object orientation and data structures have also been suggested as threshold concepts [23]. Thus, after having expressed data structures and related algorithms in Java, students may have overcome the threshold concept.

B. Nudging Students to Submit Compilable Code

The setup of the course explicitly fostered students submitting working code, thus, avoiding errors that can be detected by a compiler. First, we explicitly encouraged students to implement the assignments with an IDE, which alerts students of such errors during programming, supporting them in spotting and possibly also fixing errors based on the compiler error messages. Although compiler error messages are often confusing for students [3], [24], they might have familiarized themselves in the previous CS1 course with the interpretation of compiler error messages to fix errors. Second, our submission procedure supported and encouraged students to submit solutions that compile, due to the templates we provided. Thus, with dedicated support to learn the syntactic rules of a new programming language, we can help students focus on learning the intended concepts and data structures and how to express related problems in a programming language, thereby fostering computational thinking [2].

C. Student Development

Some students made a high number of errors in the beginning, but then either improved or stopped submitting. Interestingly, it was never the same students with the high error count, so all students improved quickly. This might indicate that the feedback they receive from one error-loaded submission leads to them paying more attention to details and produce more correct code. The students who considerably improved might have mastered one or two threshold concepts, lifting them to a higher level of programming skill.

All students who finished the course did so with low error counts in their final submissions, suggesting that it is important for students to push through these early obstacles, as it is still

possible to learn and improve considerably. This is interesting to interpret in the context of the work on predicting student success in programming courses, which found that students who fall behind early in programming courses tend to stay behind [25]. These studies focused on CS1 students, and showed that, if students fail to grasp basic concepts, such as how variables work, they have trouble catching up. For our CS2 course, we could not observe such a pattern, since none of the students stopped submitting with high error rates. Instead, we interpret that motivation is an important driving factor to succeed in a programming course, which is also in line with previous studies [26], [27]. Possibly, the students could have also learned from the preceding course that pushing through obstacles leads to eventual success.

TABLE IV
PERCENTAGE OF ERRORS PER CATEGORY ACROSS DIFFERENT STUDIES

Error category	**McCall and Kölling [11]**	**Current Study**
1.0 Incorrect attempt to use variable	19.9 %	6.8 %
2.0 Incorrect variable declaration	5.2 %	0.4 %
3.0 Incorrect method call	18 %	6.8 %
4.0 Incorrect method declaration	7.9 %	2.8 %
5.0 Incorrect constructor call	0.8 %	0.6 %
6.0 Incorrect constructor declaration	0.4 %	0 %
7.0 Incorrect/attempted use of class or type	6 %	12.5 %
8.0 Semantic error	12.7 %	30.6 %
9.0 Simple syntactical error	20.5 %	3.4 %
10.0 Statement outside method/block	0.2 %	0.3 %
11.0 Uncategorized	8.5 %	0.7 %
12.0 Logical error in working with data structures	n/a	19.9 %
13.0 Other logical error	n/a	15.4 %
Sum syntax errors	(34 %) 373	(8 %) 55
Sum semantic errors	(58 %) 638	(58 %) 408
Sum logic errors	n/a	(35 %) 250
Amount of investigated errors	1105	710

D. Comparison to McCall and Kölling [11]

To set our results into context, we compare the errors that we found with the ones from McCall and Kölling in Table IV (the comparison with other frameworks is available on the project's Web site). The original framework does not contain logical errors, likely because the authors did not automatically detect them [11]. Our analysis procedure based on automatic unit testing allowed us to also detect and categorize logical errors. Thus, for the comparison, we only focus on syntax and semantic errors. We found 55 *syntax errors*, while in the original work, there were 373 syntax errors. Especially the 21 % *9.0 Simple syntactical errors* are a difference, as we did observe only few of them (3 %). Regarding semantic errors, we observed 408, in contrast to 638 in the original study. The most noteworthy difference is the category of *8.0 Semantic errors*, in which we found 31 %, compared to 13 % in the original study. Thus, our CS2 students seem to be further along in their learning process, compared to the students from McCall and Kölling. Unfortunately, the Blackbox data prohibits us from directly comparing their results to ours. Given that, in the original study, the Blackbox data set is a snapshot sometime during the process of programming, it gives a more general overview of types of errors, while in our sample, we observed the last snapshot of a submitted solution. Thus, our data points more toward the potential of students to provide an error-free, compilable program, while in the original framework, it gives information about all the types of errors during the process. Thus, our data hints more toward the difficulties that students cannot overcome themselves and that might be especially important to address in CS2 courses to support them on their way to become professional programmers.

VI. Threats to Validity

We selected the most suitable error categorization for our data set. Although a different framework may have produced a different outcome, thus posing a threat to *construct validity*, this framework nevertheless gives a useful overview of students' programming errors and how they evolve.

Our submission procedure encouraged students to provide compilable solutions, which might threaten *internal validity*, because it might hide some of the more basic errors that students may have made during programming. With a different procedure, for example, including regular snapshots, we likely would have observed a different distribution of errors. Nevertheless, our data points toward the potential of CS2 students to avoid errors.

Unfortunately, our sample was rather small, so it is difficult to generalize our findings, which threatens *external validity*. Nevertheless, we reviewed a large body of code for different tasks and over the course of an entire semester, giving us valuable insights into the evolution of students.

VII. Conclusion

In general, we could replicate the errors that students make during programming according to the framework by McCall and Kölling, but with some nuances. Specifically, the concrete data structures and related algorithms seem to be a struggle for students in a CS2 course, which may hint at the data structures being threshold concepts. Semantic errors seem to be a hindrance at the beginning of the course that students overcome with some time. Syntactic errors, which are considered typical for novice programmers, do not seem to pose a problem for CS2 students.

Data Availability

All data and supplementary material is available at:
https://github.com/CSEEandT-25-ErrorPaper/CS2-Error-Categorization

References

[1] A. V. Robins, *Novice Programmers and Introductory Programming*. Cambridge University Press, 2019.

[2] J. Wing, "Computational Thinking," vol. 49, no. 3, pp. 33–35, 2006.

[3] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," *SIGCSE Bull.*, vol. 35, no. 1, p. 153–156, jan 2003. [Online]. Available: https://doi.org/10.1145/792548.611956

[4] N. C. C. Brown and A. Altadmri, "Novice java programming mistakes: Large-scale data vs. educator beliefs," *ACM Trans. Comput. Educ.*, vol. 17, no. 2, may 2017. [Online]. Available: https://doi.org/10.1145/2994154

[5] B. Mase and L. Nel, *Common Code Writing Errors Made by Novice Programmers: Implications for the Teaching of Introductory Programming*. Springer Nature Switzerland AG, 01 2022, pp. 102–117.

[6] P. Linz, *An Introduction to Formal Languages and Automata, Fifth Edition*, 5th ed. USA: Jones and Bartlett Publishers, Inc., 2011.

[7] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the Second International Workshop on Computing Education Research*, ser. ICER '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 73–84. [Online]. Available: https://doi.org/10.1145/1151588.1151600

[8] E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud, "Predicting at-risk novice java programmers through the analysis of online protocols," in *Proceedings of the Seventh International Workshop on Computing Education Research*, ser. ICER '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 85–92. [Online]. Available: https://doi.org/10.1145/2016911.2016930

[9] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 75–80. [Online]. Available: https://doi.org/10.1145/2325296.2325318

[10] T. Dy and M. M. Rodrigo, "A detector for non-literal java errors," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 118–122. [Online]. Available: https://doi.org/10.1145/1930464.1930485

[11] D. McCall and M. Kölling, "A New Look at Novice Programmer Errors," *ACM Trans. Comput. Educ.*, vol. 19, no. 4, jul 2019. [Online]. Available: https://doi.org/10.1145/3335814

[12] E. Albrecht and J. Grabowski, "Sometimes it's just sloppiness - studying students' programming errors and misconceptions," in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 340–345. [Online]. Available: https://doi.org/10.1145/3328778.3366862

[13] Y.-T. Chuang and H.-Y. Chang, "Analyzing novice and competent programmers' problem-solving behaviors using an automated evaluation system," *Science of Computer Programming*, vol. 237, p. 103138, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642324000613

[14] D. Mccall and M. Kölling, "Meaningful Categorisation of Novice Programmer Errors," in *Conference: Frontiers In Education Conference (FIE)*, vol. 2015, 10 2014.

[15] N. C. Brown and A. Altadmri, "Investigating novice programming mistakes: educator beliefs vs. student data," in *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ser. ICER '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 43–50. [Online]. Available: https://doi.org/10.1145/2632320.2632343

[16] A. Ettles, A. Luxton-Reilly, and P. Denny, "Common logic errors made by novice programmers," in *Proceedings of the 20th Australasian Computing Education Conference*, ser. ACE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 83–89. [Online]. Available: https://doi.org/10.1145/3160489.3160493

[17] S. H. Edwards, N. Kandru, and M. B. Rajagopal, "Investigating static analysis errors in student java programs," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 65–73. [Online]. Available: https://doi.org/10.1145/3105726.3106182

[18] E. Tshukudu and Q. Cutts, "Understanding conceptual transfer for students learning new programming languages," in *Proceedings of the 2020 ACM conference on international computing education research*, 2020, pp. 227–237.

[19] E. Tshukudu, Q. Cutts, and M. E. Foster, "Evaluating a Pedagogy for Improving Conceptual Transfer and Understanding in a Second Programming Language Learning Context," in *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3488042.3488050

[20] R. C. Bryce, A. Cooley, A. Hansen, and N. Hayrapetyan, "A one year empirical study of student programming bugs," in *2010 IEEE Frontiers in Education Conference (FIE)*, 2010, pp. F1G–1–F1G–7.

[21] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, K. Sanders, and C. Zander, "Threshold concepts in computer science: do they exist and are they useful?" *ACM SIGCSE Bulletin*, vol. 39, no. 1, pp. 504–508, Mar. 2007.

[22] K. Sanders and R. McCartney, "Threshold concepts in computing: past, present, and future," in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling 2016, vol. 57. ACM, Nov. 2016, pp. 91–100.

[23] J. E. Moström, J. Boustedt, A. Eckerdal, R. McCartney, K. Sanders, L. Thomas, and C. Zander, "Concrete examples of abstraction as manifested in students' transformative experiences," in *Proceedings of the Fourth international Workshop on Computing Education Research*, ser. ICER '08, vol. 57. ACM, Sep. 2008, pp. 125–136.

[24] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera, J. L. Pearce, and J. Prather, "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*, ser. ITiCSE '19. ACM, Dec. 2019.

[25] A. Ahadi, R. Lister, and D. Teague, "Falling Behind Early and Staying Behind When Learning to Program," in *PPIG*, vol. 14, 2014, pp. 77 – 88.

[26] A. Lishinski and A. Yadav, "Motivation, attitudes, and dispositions," in *The Cambridge Handbook of Computing Education Research*, ser. Cambridge Handbooks in Psychology, A. V. Robins and S. A. Fincher, Eds. Cambridge: Cambridge University Press, 2019, pp. 801–826.

[27] B. Schantong, D. Gorgosch, and J. Siegmund, "Toward finding and supporting struggling students in a programming course with an early warning system," 2024. [Online]. Available: https://arxiv.org/abs/2402.01709