On How Novices Approach Programming Exercises Before and During Coding

On How Novices Approach Programming Exercises Before and During Coding

JUAN PABLO SÁENZ, Politecnico di Torino, Italy
LUIGI DE RUSSIS, Politecnico di Torino, Italy

Various tools and approaches are available to support undergraduate students learning to program. Most of them concentrate on the code and aim to ease the visualization of data structures or guide the debugging. However, in undergraduate introductory courses, students are typically given exercises in the form of a natural language problem. Deriving a correct solution largely depends on the problem-solving strategy they adopt rather than on their proficiency in dealing with the syntax and semantics of the code. Indeed, they face various challenges (apart from the coding), such as identifying the relevant information, stating the algorithmic steps to solve it, breaking it into smaller parts, and evaluating the implemented solution. To our knowledge, almost no attention has been paid to supporting such problem-solving strategies before and during the coding. This paper reports an interview and a sketching exercise with 10 participants exploring how the novices approach the programming exercises from a problem-solving perspective and how they imagine a tool to support their cognitive process. Findings show that students intuitively perform various actions over the exercise text, and they would appreciate having support from the development environment. Accordingly, based on these findings, we provide implications for designing tools to support problem-solving strategies.

CCS Concepts: • **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → *Development frameworks and environments.*

Additional Key Words and Phrases: programming, problem-solving strategies, novices, development environment

ACM Reference Format:
Juan Pablo Sáenz and Luigi De Russis. 2022. On How Novices Approach Programming Exercises Before and During Coding. In *CHI '22: ACM CHI Conference on Human Factors in Computing Systems, April 30–May 6, 2022, New Orleans, LA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND BACKGROUND

Programming demands problem-solving skills that go well beyond the knowledge of the language syntax and semantics [17, 23]. It concerns the iterative process of refining mental representations of computational problems and solutions and expressing those representations as code [16]. Indeed, the benefits of learning to program should result in problem-solving skills useful in diverse domains. From a broader Computational Thinking (CT) perspective, programming and coding are meant to foster analytical thinking helpful in solving complex real-world—not necessarily computational—problems [14].

In practice, in introductory programming courses at university level, undergraduate students are typically given exercises in the form of a problem written in natural language; and they must implement a solution expressed as a computer program. The first challenges novices face do not concern just the coding but the application of problem-solving skills such as: identifying the relevant information, stating the algorithmic steps to solve it, breaking it into

Manuscript submitted to ACM

smaller parts, and evaluating the implemented solution [3, 10, 30]. They often struggle in understanding the task and decomposing the problem [15, 25].

Nevertheless, introductory courses typically do not teach about the cognitive aspects of programming, including problem-solving strategies and programming practices [16], and neither the development environments explicitly support those aspects. Consequently, when solving programming exercises, novices usually lack metacognitive awareness—the ability to think about and reflect on their problem-solving process—and fail to make progress to a working solution [22]. What is worse, some novices come to consider that spending time planning is indicative of lower programming intelligence [9] even though planning is widely documented as a good practice [8].

The number of tools, languages, and environments used in computing education has proliferated in the last few years [19]. Nevertheless, traditional IDEs treat code as the primary artifact [1]. The IDEs for novice beginners principally consist of: visualization tools that make the data structures and the code execution visible, *e.g.,* [24, 28]; graphical programming tools that enable to drag and drop the code to build programs without making syntax errors, *e.g.,* [29]; automated assessment systems that can automatically grade students' code and provide immediate feedback, *e.g.,* [2, 12, 21]; or tools for supporting debug by providing enhanced error messages, *e.g.,* [4, 7]. However, less attention has been paid to supporting problem-solving strategies, even though they are closely tied to Computational Thinking and are crucial for deriving a correct solution [18].

This paper explores how the novices approach the programming exercises from a problem-solving perspective and how they imagine a tool to support their cognitive process. To that end, we report an interview and a co-design and sketching exercise with 10 participants. Findings show that beginners intuitively perform various actions over the exercise text, and they would appreciate having support from the development environment. Accordingly, based on these findings, we provide implications for designing a tool to support problem-solving strategies before and during the coding.

2 USER STUDY

We conducted an interview and a co-design and sketching exercise with ten novice programmers. The goal was to understand, directly from the users' perspective, how they approach programming exercises analysis and comprehension, even before writing the first line of code, and during the coding, and how they would imagine a tool targeted at supporting that problem-solving step. The goals of this observation were to (i) identify the problem-solving strategies commonly used by the novices to **analyze and understand** the programming exercises problem text; (ii) recognize the problem-solving strategies adopted to **structure and map** that text requirements into code; (iii) comprehend how do the novices **cope with the difficulties** they find during the implementation of the solution.

2.1 Method

2.1.1 Participants. We recruited participants among the former university students of the last version of an introductory programming course. The course comprises a theoretical and a practical part. The practical part consists of an hour and a half-long lab, during which students are given a set of exercises where they can put into practice what they have studied in the theoretical part. Due to the COVID-19 pandemic, that version of the course was held entirely remotely using the Zoom video conferencing software, and the communication with the professor and the teaching assistants was managed using the Slack platform. Python is the chosen programming language.

The invitations to participate were sent by mail and through messages on the course's Slack channel. To minimize selection bias, all the participants had already completed the course by the recruiting time, meaning that the answers

(which, in any case, were anonymous) would not have any repercussion on their final grade. Our final sample included 8 participants who self-identified as male and 2 participants who self-identified as female, with an average age of 19.30 years (SD = 0.46). All participants currently live in Italy, and the study was conducted in Italian. Additionally, a brief questionnaire was applied to the participants before the interview to gather demographic data and information concerning the course's students' performance. The male/female ratio was 80/20, compared to the 69.2/30.8 reported by the university for its first-year bachelor students in 2020-2021. Similarly, the average age of participants was 19.3 years, compared to 19.5 years reported by the university. Finally, the middle grade on the final exam was 27.7/30, compared to the 25.9/30 average for the analyzed course in the 2020-2021 edition.

2.1.2 Procedure. The participants completed a two-part study session: an interview, and a co-design and sketching exercise. Both sessions were conducted online, on Zoom, in the spring of 2021. On average, each study session lasted 21 m 01 s (SD = 5 m 0 s).

Interview In the first part, we conducted a semi-structured interview to understand, in as much detail as possible, how students approach the programming exercises text before starting coding. To that end, the participants were asked to complete an introductory questionnaire. By using a Likert scale, the introductory questionnaire aimed at exploring:

- Which strategies do the students adopt to understand the text? In particular, are they prone to draw flow diagrams, graphically represent the data structures, or manually trace the variable values throughout the program execution?
- What is the students' *modus operandi* while coding? Do they consult the problem text, the notes they wrote next to the text, or the diagrams they drew? Do they read and implement a section of the text before coding the next one?
- Which difficulties do the students find? Do they find it challenging to understand the meaning of the text, identify the keywords or relevant information, map the text requirements to a code construct, determine which data structure to use, choosing the correct algorithm and solution strategy?
- How do the students tend to solve text comprehension problems? By searching online, taking a look at the course notes, comparing with the classmates, addressing the questions to the professor or the teaching assistant?

Then, upon their responses, the participants were asked to deepen their answers. They were encouraged to provide qualitative justifications and specific anecdotes to supplement their scores. Additionally, they were asked to reflect on the positive and negative aspects of each one's approach to face the programming exercises. Data were audio-recorded, transcribed, and subject to qualitative thematic analysis [5] for commonly recurring themes.

Co-design and sketching exercise Once completed the interview, we conducted a co-design and sketching activity to elicit, directly from the participants' perspective, how they imagined a tool that would help them solve the programming exercises. However, this activity focused on the understanding, interpretation, and abstraction of the programming problem text to map it into code. The sketch did not have to be very elaborate but to evidence its components.

3 RESULTS

3.1 Interview

Through a qualitative thematic analysis, we derived four main themes from our user study: (i) the code-focused assessment and time constraints, (ii) the relevancy of identifying, choosing, and manipulating the data structures, (iii) the approach to ask and search for advice and solve punctual doubts, and (iv) the strategies to structure the coding of the solution.

3.1.1 Code-focused assessment and time constraints. Time constraints are a notable factor that significantly influences the problem-solving strategies adopted by the novices. Although most of them reported being conscious of the benefits that a deep analysis of the text, a preliminary sketch of the code structure, and the data structures might bring, the time constraints during the labs and the exam prevent them from completing these problem-solving steps. Indeed, some participants acknowledged that they could clearly recognize a logical process and code it straightforwardly when they devoted time to reflect on their solution strategy before implementing it. For instance, P6 and P4 said:

> *"At the beginning of the course, I was able to complete the labs on time. Then, I began to work them at home, before the lab, because I was not being able to complete all the programming exercises during the given time."* (P6)

> *"By devoting some time, in the beginning, to analyze the text and identify the requirements, I never find myself in the situation of realizing that I have to restart from scratch the code that I had already implemented. Instead, I can modify the code very easily with this approach. However, this way of proceeding seems very slow to me, and I have the sensation of being wasting time."* (P4)

Consistent with the literature [9], the implementation steps different from coding are regarded as time-consuming (in the best case) or time-wasting for a few participants:

> *"When I analyze the exercise before starting to code, I make fewer mistakes, and the logic process is clear. However, I lose time."* (P7)

While the assessment focuses exclusively on the code, all the other artifacts that the students might have produced, which account for their underlying reasoning, remain ignored. Nevertheless, although all the evaluation is typically based on the code, several implementation decisions can not be inferred from it. In fact, in some cases, participants try to embrace a problem-solving strategy when starting to solve the problem. Nevertheless, when they notice that some amount of time has passed and no code has been produced, they adopt a try-error approach. In our opinion, there is a need for a tool able to make more comprehensive artifacts that can account for the design decisions and the reasoning behind the resulting code. Our findings suggest that the decision to start coding without devoting time to the text analysis does not stem from the lack of awareness on the good practices but due to the impression that artifacts different from code do not produce value.

3.1.2 Identifying, choosing, and manipulating the data structures. Data structures seem to be a primary concern among most of the participants, specifically, choosing the proper data structure to satisfy the problem requirements. The primary outcome that students expect to get from the text analysis is identifying the needed data structures. The data structure determines the design of the algorithms and, in a broader sense, the overall implementation strategy. In this regard, in an attempt to understand if a given structure might work, some participants represent it graphically on a

piece of paper to simulate how it would behave: which data would be stored; how that data would change during the program execution; and how the algorithms can access the data.

> *"I implement the algorithms imagining the data structures that fill up. At the exam, I draw them on a piece of card"* (P8). *"I do not usually do graphical schemes. Occasionally I did them when dealing with complex list manipulation exercises."* (P9).

Therefore, for instance, if the data structure were a two-dimensional array, participants drew a table and inserted the corresponding data chunk within the cells. To some extent, this practice represents a manual debug process of the chosen data for an algorithm. Additionally, the drawing of the data structure was also made before sketching the algorithms to analyze if the data could fit on it. As approached by previous research [11, 20, 27], providing graphical resources to represent their structure, data, and behavior was identified as a promising alternative:

> *"I read the text and highlight the key points to understand which data structures to use. Additionally, I comment on the code concerning complex data structures. I write why I have created them and which data they contain. In this manner, I do not forget my choices and explain them to others (...) sometimes I draw the data structures."* (P1)

> *"In the second read, I highlight keywords, associate data structures to the text, and set up the pseudo-code"* (P2). *"I read the text, decompose it into subproblems, and I aim at understanding the proper data structures to use. Additionally, I draw them (the data structures) with the values that they store during the program execution."* (P6)

Nevertheless, although the students were introduced to Python Tutor[1], they found it helpful to draw the structures by hand on a piece of paper, probably because they can freely sketch various data structures without thinking about the code.

3.1.3 Asking and searching for advice and solving punctual doubts. While students search online to check syntax, they turn to the professor to check their reasoning. Concerning the reasoning, participants are not very prone to search online: they find it difficult to search in English; the amount of information they find is vast; it does not usually solve their specific doubts.

> *"If I cannot solve it by myself, I take a look at the cheat sheet given by the professor. I do not usually search online; it is difficult to search in English and translate the answers commonly written in very technical English. I turn to my colleagues mainly for the code; I turn to the professor or the teaching assistant for clarifications regarding the understanding of the text and for advice on where to start and how to address the problem."* (P7)

We have also found that they might be influenced to use, by default, the code constructs and data structures studied in the last part of the course. For instance, during the course's final exam, the proposed exercise could be easily solved using a two-dimensional rectangular array. However, since the last data structure studied was the linked lists, all the students used them, although they were more challenging to manipulate. Additionally, we determined that during the labs, the students regularly get back to their notes, the course slides, and previous exercises to get inspiration on how to proceed.

> *"At first, I try to solve exercises by myself, and if I take too long, I tend to ask my colleagues by sharing with them the code, or I send the code to the professor to understand if they are due to a wrong reasoning."* (P3)

[1] https://pythontutor.com/

> *"I usually refer to my notes and the course slides to find algorithms or code constructs that I had not considered in my solution. I do not search on the internet: it is too vast for me. With some colleagues, I discuss the data structure and algorithms choice, with some others, I compare the syntax and the resulting code."* (P8)

3.1.4 Strategies to structure the coding of the solution. Various participants reported structuring their implementation around empty functions defined at the beginning of the coding. In this approach, they first decompose the problem posed by the programming exercise into a set of subproblems. Then, produce a list of candidate functions typically associated with a subproblem. After that, they write just the function headers with their parameters. Later, in the main function, the empty functions are invoked to define their execution order, and finally, they proceed to implement each function. According to their experience, this approach allows getting more clarity of the logical process behind the resolution strategy:

> *"I decompose the text into empty functions. Then I start to implement them."* (P4)

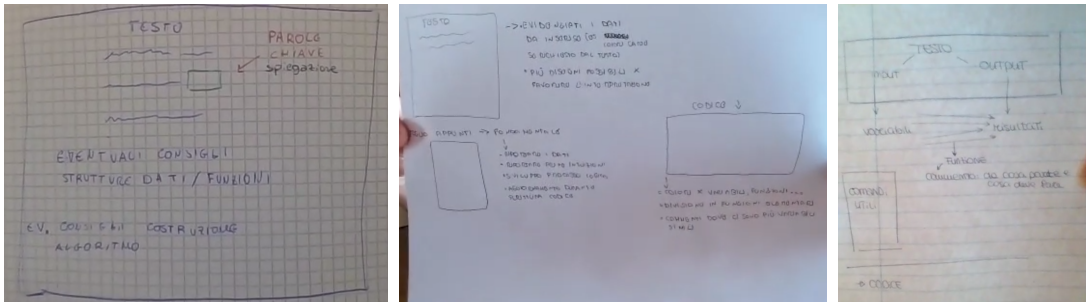
> *"I first write the main, then the functions. When I have finished implementing a function, I link it to the main (by invoking it) and check if it works."* (P6)

> *"When I worked on the exercises, I started by writing the comments, then the main, and finally the functions. The comments on each function helped me explain what they did and clarify complex chunks of code."* (P2)

The participants have constantly mentioned pseudocoding to reflect on and structure their solution. Although this pseudocoding accounts for the students' reasoning process, it is not considered in the evaluation of the exercises and the exams. All the assessment focuses on the code and is based on one artifact: code. Consequently, other artifacts students have produced to foster their understanding (diagrams, pseudocode, annotations, drawings) remain ignored. From our perspective, the code-centric way of evaluating novice programmers discourages them from adopting problem-solving strategies and devoting time to analyze the text.

> *"I follow a bottom-up approach in which I start writing the simplest functions (...) When I have already prepared the pseudocode, I write the empty functions, and then I start to fill them in order from the simplest to the most complex ones."* (P4)

3.2 Co-design and sketching exercise



(a) (b) (c)

Fig. 1. Some graphical interfaces proposed by the participants

As can be observed in Figure 1, all the participants sketched a graphical user interface indicating the features they would like to find in a tool for helping them to solve programming exercises. First of all, participants proposed integrating the text of the programming exercises directly in the IDE. In this manner, providing a holistic view of the statement and the code aimed at solving it would help programmers not lose sight of the requirements and specificities of the exercise and, consequently, take consistent implementation decisions. Various participants try to emulate this arrangement, whether by copying and pasting the programming exercise text into the IDE as a comment or by opening two windows side by side on the screen: one with the IDE and the other with the PDF of the exercise text.

They have also suggested introducing a feature to highlight and comment text fragments over the programming exercise text. Indeed, they have proposed various formatting options, both on the exercise text and the code. In this manner, the font's color and style could help to give conceptual meaning to specific pieces of the exercise statement. Adding comments to the text was also mentioned as a feature. Similar to integrating the exercise text into the IDE, these suggestions aim to support, within the development environment, the already adopted novices' tactics.

They would also appreciate some guidance on the choice of the data structure, specifically by listing the available ones with brief documentation on how and when to use them. In the same vein, P3 suggested an automatical refactoring feature that detects when the code might be incomprehensible to others (*e.g.,* very long or overlapping conditionals, unnecessary nested loops) and suggests replacements, similar to writing assistants that review spelling, grammar, and engagement.

On the other hand, P4 proposed having an area to set the inputs and expected outputs apart. This way, the programmer is expected to structure her code as steps that drive him from the initial information to achieving the specified outcomes. This alternative would be somehow similar to computational notebooks. However, in the context of introductory programming, we must consider that the code solutions do not necessarily have (and not usually do) to follow such linear and incremental order. Additionally, although several programming exercises provide sample input and output, that information cannot be taken for granted.

4 IMPLICATIONS FOR DESIGN

Stemming from the findings that emerged from the study, we propose the following implications for designing a tool aimed at supporting novice programmers, from a problem-solving perspective, before and during the coding.

Supporting artifacts different from code. As mentioned before, time is a significant concern. The use of problem-solving strategies highly depends on the available time. However, in our opinion, which gives rise to the time concerns is the perception that the final code is the only artifact that will account for the solution of the programming problem. Consequently, novices do not devote time doing diagrams, highlighting the text, structuring the possible solution around pseudocode, or commenting on the programming exercise text. Instead, students feel the impulse to start writing code immediately, even if they do not plan to complete the problem, because at least they will get some points for the chunk of code they could produce. This observation is consistent with the literature. Particularly with Carbone *et al.* [6], who stated that initially, students wanted to understand the problem at hand; however, once this became an excessively time-consuming activity with limited success, students opted for strategies that limited their learning instead of reevaluating their plan of attack. Due to the above, we envision a tool that must enable, other than writing the code, creating and capturing complementary artifacts that account for the reasoning behind the final code. In particular, given its wide use, we would propose a feature to create blocks of pseudocode linkable to code fragments.

This way, students can efficiently structure their implementation without worrying from the beginning about the syntax's specificities, and after having completed their solution, the pseudocode blocks would enrich the code.

Integrate and link the programming exercise text to the code constructs. As the majority of the participants stated, they keep the exercise text next to the code. This way, they could constantly re-read the text, check that their code is consistent with the requirements, and associate their data structure and algorithmic decisions to a piece of given information. Furthermore, as shown in Figure 1, in the graphical interfaces that they proposed, the text is placed next to the code. Nevertheless, this feature goes beyond merely arranging the components. The design implication would be to visually enable the student to link text fragments to code pieces and comment on those associations. In this commenting and annotations concern, the work by Adeli *et al.* [1] provides a working basis. Additionally, this approach contributes to identifying the necessary algorithmic steps to complete the solution and breaking up the programming exercise into smaller pieces.

Support the choice of the algorithms and the data structure. Various participants commented during the interview that their main challenges when facing the programming exercises were to identify which data structure to use and the proper algorithm accordingly. Similarly, in the co-design and sketching exercise, an area of contextual documentation was proposed to suggest, based on the code written by the student, a possible data structure, information on how to use such data structure, documentation on the available code constructs, or a code completion feature with the skeleton of a specific data structure or code construct. One of the participants pointed out that the documentation and the autocompletion feature are available in several IDEs. Indeed, after the decomposition and structuration of the problem, understanding the data structures is perceived as particularly challenging [13, 26].

Formatting and commenting features for the text and the code. The interview let us determine that when analyzing the exercise, the most common tactic among the students was to highlight the keywords or the critical text fragments. In various cases, students use different colors to distinguish the semantics of the highlighted fragment, whether it defines the set of functions to be declared, the data structure, and the algorithmic approach. Consequently, formatting the text in different ways (*e.g.,* highlighting, underlining, bolding, or adding comments) is vital to support the text analysis. In this manner, if the text is integrated into an IDE, the formatting options would help students carry their text analysis directly on the development environment and easily link it to the executable code. Furthermore, after completing the exercise, the tool will be able to save a trace of how the novices approached the text and how they mapped the textual information written in natural language into code. This feature aligns with our emphasis on capturing the problem-solving strategies adopted by the students by diversifying the artifacts from solving the programming exercises.

5 CONCLUSIONS

Getting proficient in programming demands problem-solving skills. While most of the available approaches focus on code-centric activities, less attention is paid to supporting novices' cognitive processes. This paper explores how novice programmers approach the programming exercises from a problem-solving perspective. In this vein, we conducted an interview and a co-design and sketching exercise with 10 first-year undergraduate students. Findings show that novices intuitively perform various actions over the exercise text, such as decomposing the problem and highlighting key points. They would appreciate having support from the development environment to link the programming exercise text to the code or a guide in choosing the data structures. Future work concerns the implementation of tools based upon the results and their assessment with the students in the context of introductory programming courses.

REFERENCES

[1] M. Adeli, N. Nelson, S. Chattopadhyay, H. Coffey, A. Henley, and A. Sarma. 2020. Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10. https://doi.org/10.1109/VL/HCC50065.2020.9127264

[2] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 139–150. https://doi.org/10.1145/3377814.3381703

[3] Valerie Barr and Chris Stephenson. 2011. Bringing Computational Thinking to K-12: What is Involved and What is the Role of the Computer Science Education Community? *ACM Inroads* 2, 1 (feb 2011), 48–54. https://doi.org/10.1145/1929887.1929905

[4] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 126–131. https://doi.org/10.1145/2839509.2844584

[5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[6] Angela Carbone, John Hurst, Ian Mitchell, and Dick Gunstone. 2009. An Exploration of Internal Factors Influencing Student Learning of Programming. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., AUS, 25–34.

[7] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B. Powell. 2021. *On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors*. Association for Computing Machinery, New York, NY, USA.

[8] Alireza Ebrahimi. 1994. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies* 41, 4 (1994), 457–480. https://doi.org/10.1006/ijhc.1994.1069

[9] Jamie Gorson and Eleanor O'Rourke. 2019. How Do Students Talk About Intelligence? An Investigation of Motivation, Self-Efficacy, and Mindsets in Computer Science. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/3291279.3339413

[10] Shuchi Grover and Roy Pea. 2013. Computational Thinking in K–12: A Review of the State of the Field. *Educational Researcher* 42, 1 (2013), 38–43. https://doi.org/10.3102/0013189X12463051

[11] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[12] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (Sept. 2018), 43 pages. https://doi.org/10.1145/3231711

[13] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A Study of the Difficulties of Novice Programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Caparica, Portugal) *(ITiCSE '05)*. Association for Computing Machinery, New York, NY, USA, 14–18. https://doi.org/10.1145/1067445.1067453

[14] Rina P. Y. Lai. 2021. Beyond Programming: A Computer-Based Assessment of Computational Thinking Competency. *ACM Trans. Comput. Educ.* 22, 2, Article 14 (nov 2021), 27 pages. https://doi.org/10.1145/3486598

[15] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '06)*. Association for Computing Machinery, New York, NY, USA, 118–122. https://doi.org/10.1145/1140124.1140157

[16] Dastyni Loksa and Andrew J. Ko. 2016. The Role of Self-Regulation in Programming Problem Solving Process and Success. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/2960310.2960334

[17] Dastyni Loksa, Andrew J. Ko, Will Jernigan, Alannah Oleson, Christopher J. Mendez, and Margaret M. Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1449–1461. https://doi.org/10.1145/2858036.2858252

[18] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory Programming: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[19] Monica M. McGill and Adrienne Decker. 2020. Construction of a Taxonomy for Tools, Languages, and Environments across Computing Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 124–135. https://doi.org/10.1145/3372782.3406258

[20] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2019. A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education* 62, 2 (2019), 77–90. https://doi.org/10.1109/TE.2018.2864133

[21] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education* (Dundee, Scotland) *(ITiCSE-WGR '07)*. Association for Computing Machinery, New York, NY, USA, 204–223. https://doi.org/10.1145/1345443.1345441

[22] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 531–537. https://doi.org/10.1145/3287324.3287374

[23] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. https://doi.org/10.1145/3077618

[24] Jonathan C. Roberts, Panagiotis D. Ritsos, James R. Jackson, and Christopher Headleand. 2018. The Explanatory Visualization Framework: An Active Learning Framework for Teaching Creative Computing Using Explanatory Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 791–801. https://doi.org/10.1109/TVCG.2017.2745878

[25] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education* 13, 2 (2003), 137–172. https://doi.org/10.1076/csed.13.2.137.14200 arXiv:https://doi.org/10.1076/csed.13.2.137.14200

[26] Neil Smith, Mike Richards, and Daniel G. Cabrero. 2018. Summer of Code: Assisting Distance-Learning Students with Open-Ended Programming Tasks. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018)*. Association for Computing Machinery, New York, NY, USA, 224–229. https://doi.org/10.1145/3197091.3197119

[27] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 4, Article 15 (Nov. 2013), 64 pages. https://doi.org/10.1145/2490822

[28] J. Ángel Velázquez-Iturbide, Isidoro Hernán-Losada, and Maximiliano Paredes-Velasco. 2017. Evaluating the Effect of Program Visualization on Student Motivation. *IEEE Transactions on Education* 60, 3 (2017), 238–245. https://doi.org/10.1109/TE.2017.2648781

[29] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 283–295. https://doi.org/10.1145/3426425.3426948

[30] Jeannette M. Wing. 2006. Computational Thinking. *Commun. ACM* 49, 3 (mar 2006), 33–35. https://doi.org/10.1145/1118178.1118215