TextCode: A Tool to Support Problem Solving Among Novice Programmers

Fulvio Corno
Dip. di Automatica e Informatica
Politecnico di Torino
Turin, Italy
fulvio.corno@polito.it

Luigi De Russis
Dip. di Automatica e Informatica
Politecnico di Torino
Turin, Italy
luigi.derussis@polito.it

Juan Pablo Sáenz
Dip. di Automatica e Informatica
Politecnico di Torino
Turin, Italy
juan.saenz@polito.it

***Abstract*—Several tools have been developed to support novices learning to program. Most of them focus on the code and provide features regarding the visualization of the data structures or the debugging. However, in introductory programming courses, students are typically given exercises in the form of a problem written in natural language; and the first challenge they face is understanding the problem, identifying the relevant information, and then translating that information into code. To our knowledge, little attention has been paid to proposing tools targeted at supporting this problem-solving step, even though it is crucial for deriving a correct solution. In this paper, we present an IDE to encourage novices to understand the problem before start coding, decompose it down into subproblems, explore alternative implementations for each subproblem, and arrange these implementations to build a general solution. Finally, the adopted problem-solving approach is discussed.**

***Index Terms*—novices, programming, problem solving**

I. Introduction

Getting started with programming is difficult. It requires motivation, practice, and, most importantly, abstract thinking and deep understanding instead of surface memorization [1]. Contrary to the common perception, learning to program does not only concerns the syntax and semantics of the programming languages but, more fundamentally, the iterative process of refining mental representations of computational problems and solutions, as well as mapping those representations to code [2]. Programming entails skills that go far beyond becoming familiar with the language [3], [4].

In introductory programming courses, 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. In this context, the first challenge novices face is understanding the problem, identifying the relevant information, and then translating it into code. However, they often struggle in understanding the assignment and decomposing the problem [5], [6]. Furthermore, the introductory courses generally do not address the cognitive aspects of programming, including problem-solving strategies and programming practices [2], and neither the development environments explicitly support such cognitive aspects. Thus, when solving programming problems, 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 [7]. What is worse, some novices come to consider that devoting time to planning is suggestive of lower programming intelligence [8] even though planning is extensively recognized as a good practice [9].

During the last few years, the number of languages, tools, and development environments for computing education has flourished [10]. Nevertheless, traditional IDEs treat code as the primary artifact [11], and the IDEs for novice beginners mainly concern: visualization tools that make the data structures and the code execution visible, *e.g.,* [12], [13]; graphical programming tools that enable to drag and drop the code to produce syntax error-free programs, *e.g.,* [14]; or automated assessment systems that can automatically grade students' code and provide immediate feedback, *e.g.,* [15], [16]. However, less attention has been paid to supporting problem-solving strategies, even though they are crucial for deriving a correct solution.

In this paper, we present TextCode, an IDE specifically designed to support problem solving among novice programmers in an introductory university course. In our proposed IDE, the graphical interface arrangement and the interaction design is aimed at encouraging the developers to follow a set of steps to reach the final solution. Namely, these steps refer to reading and understanding the programming problem statement before start coding, decomposing it down into subproblems, exploring alternative implementations for each subproblem, and arranging these implementations to build a general solution.

II. Background & Related Work

Medeiros *et al.* [17] conducted a systematic literature review on teaching and learning introductory programming in higher education. Upon analyzing a significant number of research works, they identified the crucial skills for a novice learning to program, the main challenges faced by novices, and the challenges faced by the teachers. Most of the literature reviewed remarked that problem solving was a crucial skill for novices learning to program and, at the same time, the main challenge they face. Indeed, various authors and the Computer Science Curricula [18] agree that programming courses' primary goal should be fostering problem-solving abilities and algorithmic

978-1-6654-4592-4/21/$31.00 ©2021 IEEE

thinking rather than teaching the syntactic specificities of a given programming language.

Although problem solving refers to understanding the context of a problem, identifying critical information, and creating a plan to solve it [19], the reasons behind students' weaknesses in problem solving are not reported in detail in the literature. It is uncertain whether those limitations regard the lack of understanding of the programming problem statement, or the lack of knowledge of a strategy to solve them, or the absence of a step-by-step plan to implement the strategy [17]. Against this backdrop, and taking into account that the main challenge faced by educators is the necessity for proper approaches and tools for teaching programming at the introductory level, the review shows the need to design tools specifically targeted at the problem formulation stage.

Koulouri *et al.* [20] conducted a quantitative evaluation over four years to assess the impacts of three factors —choice of programming language, problem-solving training, and the use of formative assessment— on learning to program in CS1 courses. The outcomes indicated that problem-solving training positively affects the students' programming ability. According to the authors, novice programmers usually start writing the code without examining and analyzing the problem. However, it is essential to realize that a programming exercise at its core is a problem that can be solved by first decomposing it into steps and then expressing them as code. In the same vein, Lishinski *et al.* [21] used various kinds of course assignments (projects and tests) to evaluate programming outcomes and determined that problem-solving ability significantly correlates with performance on programming assignments.

In the optics of tools that aim at increasing the novice's understanding of the code, Adeli *et al.* [11] conducted an empirical study to determine how the right information at the right time and right place enables novices to overcome the difficulties associated to understanding the code. By combining relevant information and code fragments on a canvas-based IDE, the cognitive load of the participants was decreased, allowing them to efficiently seek critical information and develop consistent mental models of the codebase. Similarly, Lichtschlag *et al.* [22] present a prototype to increase programmers' software comprehension by integrating hand-drawn sketches into a development environment and linking these sketches to source code. For their part, Biegel *et al.* [23] created a prototype tool for visualizing and exploring the developers' mental model through a two-dimensional canvas in which visual representations of source code entities could be freely arranged.

In this context, while the currently available tools —whether they are visualization, graphical programming, automated assessment, or even tutoring tools— are directly focused on the solution expression, TextCode aims at supporting problem solving starting from the text of the programming assignment. Specifically, the IDE is intended to encourage the developers to follow a set of steps to reach the final solution, and to that extent, to raise awareness among the novices about the problem-solving process they bear in mind when solving a programming problem.

III. Usage scenario

We implemented the IDE as a web application using the React framework. Programmers can easily access and use it potentially from anywhere and without any platform restriction or prerequired software components. To show how the TextCode IDE looks and works, Figure 1 illustrates a usage scenario:

(a) The student starts with a programming problem statement written in natural language.

(b) After reading the text, the student clicks the "Code it" button, and both the "Canvas" panel and the "Sortable list" panel (B and C) appear. Before clicking the button, only the "Programming problem statement" panel (A) is visible.

(c) The student selects a text fragment in the problem statement that they retains relevant (information that impacts the coding) and drags and drops it into the Canvas (B). Once dropped, the selected text fragment highlights with a given color and an empty code snippet (bordered in the same color) is created on the Canvas. Each text fragment is assigned a different color.

(d) The student writes into the corresponding snippet the code to solve the subproblem that the associated text fragment pose. They is free to create as many code snippets as they wants for the same text fragment. To do it, the user has to select and drag the text fragment again into the canvas (B). At the moment, Python is the programming language that TextCode supports.

(e) When a code snippet is ready, the student drags it to the Sortable list panel (C). In that area, the snippets are also vertically draggable so that the user can sort them in an ordered list. After the first snippet is dropped into the Sortable list panel (C), "Code preview and execution output" panel (D) immediately becomes visible. As the user changes the code within the snippets in the sortable list or rearranges them, the Code preview and execution output updates accordingly, to display the consolidated code. However, in this panel, the code is read-only. Additionally, since there might be various code snippets to solve the same subproblem (linked to the same text fragment), the user can experiment with these alternative implementations by exchanging them on the sortable list.

(f) When the student has finished composing the code, they can run it, and the results display in the lower part of the Code preview and execution output panel (D). Finally, the programmer clicks the "Verify snippets" button (E) to check whether the text fragments upon which they built their solution match those indicated by the instructor who prepared the programming problem statement. To that end, a modal window appears, displaying the code fragments that on the instructor's criteria should have been selected along with an explanatory remark in the form of comment for each one of them.

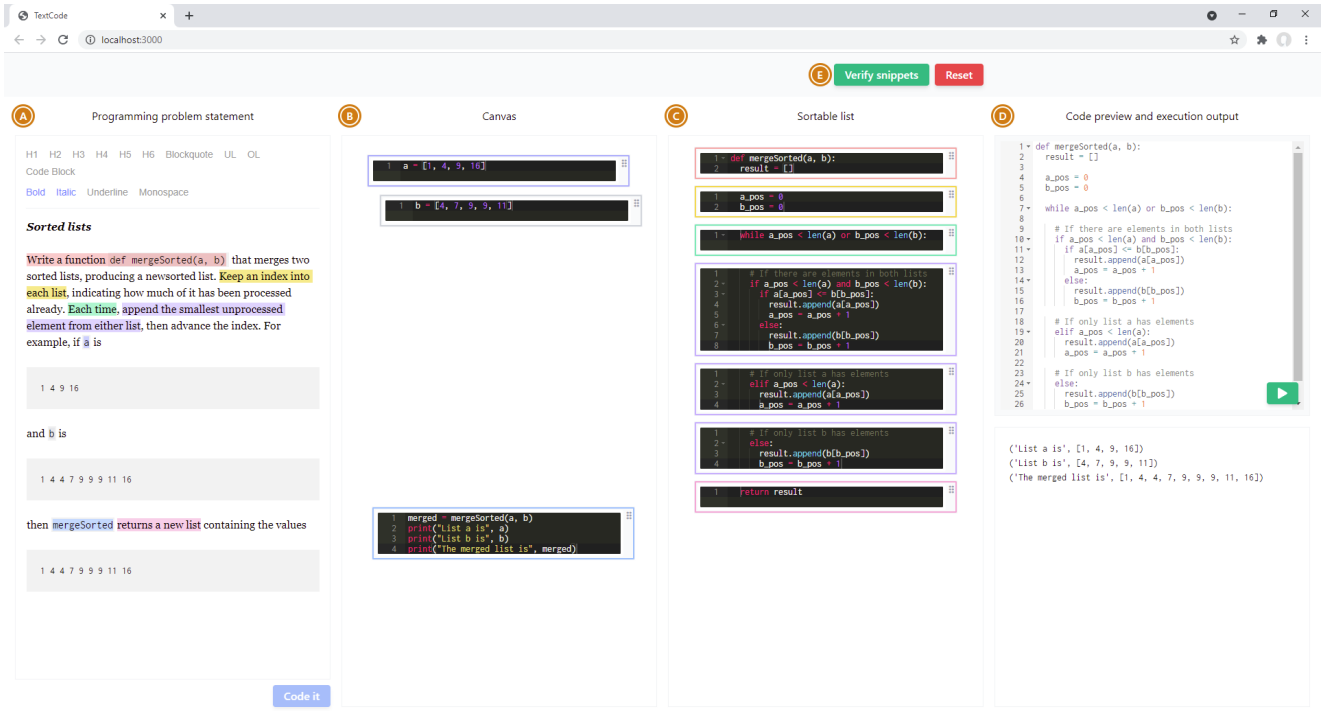

Fig. 1. TextCode is an IDE that allows the user to (i) visualize the programming problem statements; (ii) select text fragments and link them to code snippets where they writes the code to solve the subproblem that the associated fragment pose; (iii) arrange the code snippets to build a general solution; (iv) experiment with alternative solutions to the subproblems by exchanging code snippets associated with the same text fragment; (v) execute the final solution.

IV. Design and Features of TextCode

We now revisit the features of Figure 1 with a focus on design motivations and implementation details. We refined TextCode interface design over several rounds of discussions with *four experts* (a full professor and three postdocs with experience preparing and delivering introductory programming courses in different universities and various languages) that during the last year were involved in developing and assessing an introductory undergraduate programming course with approximately 200 students.

The resulting graphical interface is structured around four panels, one for each of the problem-solving steps outlined by the experts during our discussions: i) understanding the programming problem statement before start coding, ii) decomposing it down into subproblems, iii) exploring alternative implementations for each subproblem, and iv) arranging these implementations to build a general working solution.

A. Programming problem statement

The first panel to the left (A) corresponds to the text editor. In this area, the programming problem statements are written in natural language by the instructors, who prepare them, and are displayed to the developers, who will work on their solution. As can be observed in Figure 1, the text editor provides various formatting options. In this manner, the examples of the inputs, the expected outputs, the name of the variables that the program should manipulate, and eventual subtitles, listings, or keywords, remain as identifiable and easy to understand as possible to the novices. These formatting options become important because if students misinterpret the problem statement, they will probably not create a valid conceptual model of the assignment and will fail to develop a working solution [7]. Naturally, the formatting options are just available for those who prepare the programming problem. Instead, for the students, the text is not editable.

From the instructors' perspective, they prepare a set of programming problem statements using the text editor. They also highlight and save the text fragments in the problem description that, in their opinion, might correspond to relevant information. They might also add an explanatory remark in the form of comment for each text fragment highlighted. Later, the developer sees that comment when they finishes implementing their solution and wants to obtain feedback on their decomposition into subproblems (described in detail below). From the novices' point of view, the first and only thing they see is the text editor with a programming problem statement. After reading the text, they highlight text fragments that, on their criteria, are relevant and drag and drops them into the second panel.

B. Canvas

The second panel is a canvas (B) where the code snippets are created and can be moved freely by dragging and dropping them. Specifically, each time the user drops a selected text fragment from the text editor into the canvas, a new empty code snippet is created. Additionally, after adding the code snippet, a highlighting color is assigned to the text fragment

and the snippet's border to visually represent the link between them. One text fragment may have associated several code snippets, and novices can create as many code snippets as needed. This cardinality one-to-many enables the programmer to propose various implementations of the same subproblem. For instance, a text fragment that requires the implementation of a cycle can be solved by at least two code snippets: one of them using `for` and the other using `while`. Similarly, as illustrated in Figure 1, the text fragment *"append the smallest unprocessed element from either list"* (highlighted in purple) is structured around three code snippets, one for each condition that must be verified in the context of the sample exercise. Nevertheless, our approach does not rely on direct *translation* of the problem statement into implementation constructs or prose elements converted to formal representations. Instead, TextCode relies on *mapping*, aiding developers in considering whether their programming decisions are consistently linked to some information in the text. Finally, once the developer has written the code in a snippet, they drags the snippet and drops it into the next panel: the code snippets' sortable list. Furthermore, the user can delete the snippets by dragging and dropping them to the lower part of the canvas. When all the snippets linked with a highlighted text fragment are deleted, such highlighting deletes as well.

C. Sortable list

In the third panel (C), the snippets are arranged into a sortable list. By dragging and dropping them upwards or downwards, the novice can set the snippets' order and compose the final solution to the programming problem statement. In this sortable list panel, the snippets are still editable. Additionally, the user can remove the snippet from the list by dragging and dropping it into the canvas panel. In this way, for instance, if there were two code snippets with alternative implementations of the same text fragment, users could easily experiment by exchanging them from the canvas to the list and vice-versa. This feature, in our opinion, has a pedagogical value since it allows novices to experiment with various ways to solve the same sub-problem and get proficient in the use of several code constructs.

D. Code preview and execution output

The last panel (D) encompasses two components: the code previewer in the upper half and the execution output in the lower half. The previewer displays the concatenated code from all the snippets' sortable list into a single container. It is not editable. Instead, it updates every time a snippet in the sortable list is modified. The previewer's purpose is to provide the novice a consolidated view of the code composed through the snippets in the sortable list. The lower part of the panel displays the outputs resulting from the previewer's code execution. The output shown in Figure 1 results from executing all the code snippets. However, after running them, the two first snippets and the last one were dragged and dropped into the canvas for demonstrative purposes. Lastly, in the upper part of the interface (E), there is the "Verify snippets" button. It enables the novice to see, over the programming problem statement, the text fragments that the instructor considered relevant and a comment written by them (in natural language), explaining their choice and pointing out aspects to be considered for the implementation.

V. Design considerations and conclusions

In this paper, we presented TextCode, an IDE to support problem-solving skills among university students learning to program. The tool strives to encourage developers to read and understand the problem before starting coding, decomposing it down into subproblems, exploring alternative implementations for each subproblem, and arranging these implementations to build a general solution. We believe that providing a tool whose design emphasizes the problem-solving steps, starting from understanding the programming problem statement, can raise metacognitive awareness among novices and help them build a working solution. Furthermore, we consider that our tool counteracts the novices' misconceptions about the problem-solving steps as time-spending, non-useful activities from a practical point of view.

Moreover, stemming from the discussions with the four experts, the design decisions of the tool were oriented toward supporting the cognitive aspects of programming. In particular, our chosen approach concerns subgoal learning, a strategy widely used in STEM areas to help students break down problem-solving procedures into subgoals (functional parts of the overall procedure) and distinguish the primary elements of the problem-solving process [24]. We consider that encouraging novices, directly from the IDE, to decompose the problem, solve the subproblems individually, and then arrange them in the general solution can improve their confidence in two ways: first, by noticing that programming is a step-by-step, incremental process in which, on every completed step, no matter how simple it appears, they are making progress; and secondly, by realizing that they can deal with a problem that at first glance seems overwhelming if, instead of immediately thinking into coding, they first understand what the problem is demanding and can associate each their programming decisions to the problems' requirements and specificities.

The upcoming version of the tool will include semantics into the colors used to highlight the text fragments. In this manner, depending on the color, the developer can also recognize whether the text fragment implies a variable's declaration, the creation of a function, or a constructor's definition. This feature might become particularly relevant in an Object-Oriented Programming context, where developers are also required to identify the entities present in the problem description, along with their attributes and methods. Similarly, while Python is the programming language that the IDE currently supports, it might be extended to support other languages. Finally, future work will regard conducting a full-scale evaluation in the context of an undergraduate introductory programming course to evaluate the IDE's usability and assess to what extent it effectively supports the novices' problem-solving process.

References

[1] F. B. Tek, K. S. Benli, and E. Deveci, "Implicit theories and self-efficacy in an introductory programming course," *IEEE Transactions on Education*, vol. 61, no. 3, pp. 218–225, 2018.

[2] D. Loksa and A. J. Ko, "The role of self-regulation in programming problem solving process and success," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ser. ICER '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 83–91. [Online]. Available: https://doi.org/10.1145/2960310.2960334

[3] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett, "Programming, problem solving, and self-awareness: Effects of explicit guidance," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1449–1461. [Online]. Available: https://doi.org/10.1145/2858036.2858252

[4] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3077618

[5] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003. [Online]. Available: https://doi.org/10.1076/csed.13.2.137.14200

[6] R. Lister, B. Simon, E. Thompson, J. L. Whalley, and C. Prasad, "Not seeing the forest for the trees: Novice programmers and the solo taxonomy," in *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ser. ITICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 118–122. [Online]. Available: https://doi.org/10.1145/1140124.1140157

[7] J. Prather, R. Pettit, B. A. Becker, P. Denny, D. Loksa, A. Peters, Z. Albrecht, and K. Masci, "First things first: Providing metacognitive scaffolding for interpreting problem prompts," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 531–537. [Online]. Available: https://doi.org/10.1145/3287324.3287374

[8] J. Gorson and E. O'Rourke, "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*, ser. ICER '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 21–29. [Online]. Available: https://doi.org/10.1145/3291279.3339413

[9] A. Ebrahimi, "Novice programmer errors: language constructs and plan composition," *International Journal of Human-Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S107158198471069X

[10] M. M. McGill and A. Decker, "Construction of a taxonomy for tools, languages, and environments across computing education," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ser. ICER '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 124–135. [Online]. Available: https://doi.org/10.1145/3372782.3406258

[11] M. Adeli, N. Nelson, S. Chattopadhyay, H. Coffey, A. Henley, and A. Sarma, "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)*, 2020, pp. 1–10.

[12] J. n. Velázquez-Iturbide, I. Hernán-Losada, and M. Paredes-Velasco, "Evaluating the effect of program visualization on student motivation," *IEEE Transactions on Education*, vol. 60, no. 3, pp. 238–245, 2017.

[13] J. C. Roberts, P. D. Ritsos, J. R. Jackson, and C. Headleand, "The explanatory visualization framework: An active learning framework for teaching creative computing using explanatory visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 791–801, 2018.

[14] M. Verano Merino and T. van der Storm, "Block-based syntax from context-free grammars," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–295. [Online]. Available: https://doi.org/10.1145/3426425.3426948

[15] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3231711

[16] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," in *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 204–223. [Online]. Available: https://doi.org/10.1145/1345443.1345441

[17] R. P. Medeiros, G. L. Ramalho, and T. P. Falcão, "A systematic literature review on teaching and learning introductory programming in higher education," *IEEE Transactions on Education*, vol. 62, no. 2, pp. 77–90, 2019.

[18] Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) and IEEE Computer Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013.

[19] S. Lehoczky and R. Rusczyk, *The Art of Problem Solving, Volume 1: the Basics*. AoPS Inc., 2006.

[20] T. Koulouri, S. Lauria, and R. D. Macredie, "Teaching introductory programming: A quantitative evaluation of different approaches," *ACM Trans. Comput. Educ.*, vol. 14, no. 4, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2662412

[21] A. Lishinski, A. Yadav, R. Enbody, and J. Good, "The influence of problem solving abilities on students' performance on different assessment tasks in cs1," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 329–334. [Online]. Available: https://doi.org/10.1145/2839509.2844596

[22] L. Lichtschlag, L. Spychalski, and J. Bochers, "Codegraffiti: Using hand-drawn sketches connected to code bases in navigation tasks," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 65–68.

[23] B. Biegel, S. Baltes, I. Scarpellini, and S. Diehl, "Code basket: Making developers' mental model visible and explorable," in *2015 IEEE/ACM 2nd International Workshop on Context for Software Development*, 2015, pp. 20–24.

[24] R. K. Atkinson, R. Catrambone, and M. M. Merrill, "Aiding transfer in statistics: Examining the use of conceptually oriented equations and elaborations during subgoal learning," *Journal of Educational Psychology*, vol. 95, no. 4, pp. 762–773, 2003. [Online]. Available: https://doi.org/10.1037/0022-0663.95.4.762