An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior

Austin Z. Henley, Julian Ball, Benjamin Klein, Aiden Rutter, Dylan Lee
University of Tennessee
Knoxville, Tennessee
azh@utk.edu, {jball16, bklein3, arutter1, dlee97}@vols.utk.edu

***Abstract*—Novice programmers face numerous barriers while attempting to learn how to code that may deter them from pursuing a computer science degree or career in software development. In this work, we propose a tool concept to address the particularly challenging barrier of novice programmers holding misconceptions about how their code behaves. Specifically, the concept involves an inquisitive code editor that: (1) identifies misconceptions by periodically prompting the novice programmer with questions about their program's behavior, (2) corrects the misconceptions by generating explanations based on the program's actual behavior, and (3) prevents further misconceptions by inserting test code and utilizing other educational resources. We have implemented portions of the concept as plugins for the Atom code editor and conducted informal surveys with students and instructors. Next steps include deploying the tool prototype to students enrolled in introductory programming courses.**

***Index Terms*—Code editor, program comprehension, novice programmers**

I. Introduction

Novice programmers face numerous barriers while attempting to learn how to code that may deter them from pursuing a computer science degree or career in software development. In this work, we propose a tool concept to address the particularly challenging barrier of novice programmers holding misconceptions about how their code behaves. A key reason that novice programmers have this issue is they often have a different definition of program correctness than professional programmers [34]. In fact, one study found that many students believe a program is correct when there are no compiler errors, but the students indicated little regard to the program's behavior [50]. These *semantic errors*, when the program compiles but behaves differently than the programmer expects, have vexed many students, and in another study, it took students substantially longer to fix semantic errors than syntax errors (if they were fixed at all) [1].

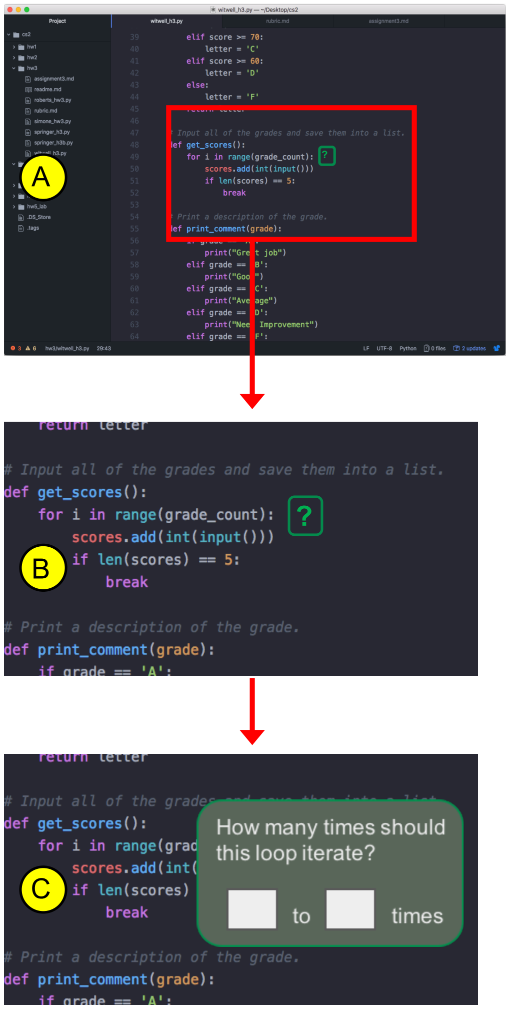
Towards addressing these issues, we propose an inquisitive code editor that elicits information about program behavior from novice programmers to address their misconceptions. The editor will prompt them with questions about the program's behavior that can be verified using static analysis and utilize the response as a learning opportunity for the programmer. More specifically, the editor will do this by detecting potentially buggy code using a code smell detector that is attuned to mistakes made by novices. Then it will prompt the programmer with a question about the program behavior that can be validated with static analysis. If the programmer answers incorrectly, the tool will provide an explanation about the program behavior and insert assertions and comments into the code. Given these features, the goals of the inquisitive code editor are to (1) identify when a programmer has a misconception about the program behavior, (2) correct the misconception by explaining how the program will actually behave, and (3) prevent the programmer from having similar misconceptions again.

The foundation of our tool concept is based on prior research on debugging tools, empirical evidence of mistakes that novice programmers make, and tutoring systems. A notable research tool for debugging is Whyline, which enables programmers to ask *why* and *why not* questions about their program's behavior [31], [32]. However, this may not be suitable for novices since Whyline requires that the programmer already knows that a bug exists and is able to replicate the buggy behavior. The aim of our inquisitive code editor is to instead have the code editor asking the questions, such that novice programmers have a more efficient feedback loop regarding their program's behavior and can effectively address their misconceptions. Moreover, our concept makes use of recent research on tutoring systems for learning to code (e.g., PLTutor [40] and Ludwig [48]).

In this paper, we present a detailed description of our novel tool concept along with a mock implementation in the form of a plugin for the popular Atom code editor (Section II). We also share our preliminary results and efforts towards researching the effectiveness of our technique (Section III). Following that, we provide a detailed literature review of relevant background information and related works (Section IV). Finally, we conclude with the next steps in our research towards investigating how to better address the misconceptions of novice programmers (Section V).

II. Idea: An Inquisitive Code Editor

To address the misconceptions that novice programmers have about their code, we propose a novel concept of an inquisitive code editor that elicits information about program behavior from novice programmers to address their misconceptions. The editor concept does this through a three step process. First, it can identify misconceptions by periodically



```
        elif score >= 70:
            letter = 'C'
        elif score >= 60:
            letter = 'D'
        else:
            letter = 'F'
        return letter

# Input all of the grades and save them into a list.
def get_scores():
    for i in range(grade_count):
        scores.add(int(input()))
        if len(scores) == 5:
            break

# Print a description of the grade.
def print_comment(grade):
    if grade == 'A':
        print("Great job")
    elif grade == 'B':
        print("Good")
    elif grade == 'C':
        print("Average")
    elif grade == 'D':
        print("Need Improvement")
    elif grade == 'F':
```

How many times should this loop iterate?

Fig. 1. (A) A mockup of the inquistive code editor for addressing misconceptions as a plugin for the Atom code editor. (B) Code that may be misunderstood is automatically annotated with a green question mark. (C) Clicking the annotation displays a question for the programmer to answer about how they think the program will behave.

prompting the programmer with questions about the program's behavior. Second, it can correct the misconceptions by generating explanations based on the program's actual behavior. Third, it can prevent further misconceptions by inserting test code (e.g., assertions or unit tests) and documentation, providing external help for more in-depth explanations, and send aggregate reports to the instructor.

A. Identifying Misconceptions

The technique that we propose is for the editor to *ask* the programmer questions about the behavior of specific portions of code while the programmer is writing the code. For example, it can ask, "How many times will this loop iterate?", along with a textbox or slider for the programmer to input the range they believe is correct. Using *program analysis*, the tool can compare the programmer's answer to how the analysis determined that the program will actually behave. This example question could identify common loop misconceptions, such as a loop that will only ever iterate one time. Fig. 1-A depicts a mockup of a tool that extends a code editor to annotate potentially problematic code (Fig. 1-B) and pose questions to the programmer about the code (Fig. 1-C).

There are a number of design dimensions that need to be studied in order to identify misconceptions effectively. First, which misconceptions and code issues should the code editor ask questions about (and how does it know this is a potential bug)? There are existing databases of bugs that students have encountered in various course settings, such as Blackbox [5]. The tool could combine such a database with existing code smell detector techniques. Second, how does the tool determine the correct answer to how the program will behave? There are a variety of program analysis techniques that could be used (e.g., data-flow analysis) based on the misconceptions targeted. These static and dynamic program analysis techniques allow the editor to infer specific properties about the program behavior (e.g., a loop will never terminate). Third, how should programmers interact with the editor without finding it distracting? From our prior work, we have found that it is vital for the tool to work without upfront configuration and without disruption to their workflow. For example, Yestercode and CodeDeviant record code changes automatically and provide visualizations that are available if the programmer needs assistance, without upfront configuration [21], [22]. If our inquisitive editor is too disruptive, the programmer will be less likely to use it and it may interfere with their task, and thus contradict the goals we are trying to achieve. Fourth, what other techniques could be used in conjunction with program analysis for identifying misconceptions? Other researchers have began looking into ways to identify when a programmer is frustrated (e.g., [9], [39], [47]), which could also be indicative of a misconception. Additionally, other signals could be investigated to see if they correlate with misconceptions (e.g., keyboard and mouse activity, inserting breakpoints, or running the program repeatedly).

As such, our initial design is to annotate the potentially problematic code in the code editor (shown in Fig. 1-B as a green question mark), much like a syntax error. Annotating the code is a deliberate alternative to interrupting the programmer with a dialog window whenever the issue is identified. These annotations are based on *negotiable interruptions*, notifications that allow the user to attend to when they choose, and studies have found them to be preferred in long, continuous tasks [37]. Fig. 1-C shows an example of a question that is posed to the programmer after clicking the green question mark, indicating a portion of code that may be indicative of misconceptions. There are a variety of different question types and input forms (e.g., fill in the blank, drag-and-drop, natural language, etc.) that should be investigated.

B. Correcting Misconceptions

After identifying a misconception, our inquisitive code editor aims to then *correct* programmers' misconceptions. To do so, the editor will present an explanation as to how the program will actually behave and why the program will behave that way. Using the programmer's answer to the question (e.g., Fig. 1-C), the editor will generate a customized explanation based on the specific misconception. More specific designs can utilize findings from *computing education research*, such as tutoring systems, hint generation, and enhanced compiler errors.

There are two open questions regarding the features for correcting programmers' misconceptions: *what* information to provide and *how* to present it. It is likely not sufficient to just inform the programmer how the program will behave (e.g., "This loop will never iterate more than 5 times."). An existing tool, the Whyline, was designed to help programmers interrogate their program to understand why it is not behaving as expected [31]. However, our goal is different in that we want to assist the novice programmer in overcoming a misconception that they do not even know they have yet. The information needs of novice programmers is likely dependent on the situation and context, and may need to be adapted based on the environment.

Furthermore, an interactive explanation may lead to more engagement and better understanding for the programmer. For example, the editor could tell the programmer to test the program with specific inputs that would cause it to fail, and thus, give the programmer a first-hand experience of the program not behaving as they expected. A similar approach, known as *surprise-explain-reward*, has been applied to programming environments with notable benefits [6], [29]. For example, it has been used to entice end-user programmers in testing spreadsheets by inputting arbitrary values into the spreadsheet to see if it "surprises" the programmer with the result [54]. Seeing the program behave in an unexpected manner could have a positive effect on the programmer's understanding rather than just reading an explanation, and also encourage the programmer to continue testing the program. This approach should prove to be an improvement over just providing textual explanations, such as enhanced compiler error messages, which have yielded mixed results [2], [7], [12], [41], [44], [45].

C. Preventing Misconceptions

Once the programmer's misconception has been identified and corrected, our proposed code editor will attempt to prevent the programmer from having similar misconceptions again. There are three mechanisms that can be utilized to prevent programmers from having similar misconceptions in the future about their program's behavior. First, the editor could generate test code (i.e., assertions or unit tests) and documentation (i.e., code comments). The goal of the test code is to inform the programmer if and when the program behavior changes, such that their understanding of the program's behavior changes accordingly. For example, an assertion could be inserted that will cause an immediate error if a variable does not always fall within a specified range. For more advanced students, the tools could automatically generate unit tests, a more sophisticated means of testing than assertions, but are often not taught in introductory computer science courses. The tools will also automatically generate code comments to provide the programmer a reminder of how the code behaves and an explanation as to why it behaves that way. To avoid causing additional frustration and cognitive load, the tools will need to provide features to efficiently hide the inserted test code and comments, such as a toggle button. A concern is that inserting too many lines of code or comments could make navigating the code tedious and difficult, so this will require further investigation on how to do so efficiently and succinctly.

Second, the editor could monitor which types of issues the programmer is struggling with (e.g., loops or bitwise operators) and use it as an opportunity to teach that topic to the programmer. For example, if multiple misconceptions have been identified regarding the number of iterations in a loop, the tool could walk through each piece of a loop with explanations on the syntax and semantics. If more help is needed, the tools could provide excerpts from or links to relevant code documentation, programming tutorials, and examples.

Third, the code editor could provide aggregate feedback to course instructors on what misconceptions their students are often facing. To do so, the editor could be configured to anonymously collect data on misconceptions in a course setting (e.g., homeworks or in-class activities) and upload the data to a web server that is accessible by the course instructor. The course instructor could then view reports of what programming constructs are students having difficulty in understanding and adapt the course to provide additional instruction on the problematic topics.

III. Preliminary Results

We have been investigating our tool concept from a number of different perspectives. We have implemented a plugin for Atom that supports annotating potentially problematic code snippets, presenting a question and allowing the programmer to enter an answer, and presenting an explanation and inserting an optional code fix or assertion. Additionally, we have created a framework for logging interaction data in the code editor that is submitted to a remote server in a confidential manner, such that we can later analyze the effectiveness of our technique.

To identify the potentially problematic code snippets, we first investigated the viability of off-the-shelf code smell detectors and linters (e.g., SonarSource[1]). However, these tools are predominately designed for professional software developers. By examining the rules employed by such tools and showing them to undergraduate students, we found that the majority of them are of little to no value to novice programmers and could actually cause further confusion. Furthermore, these tools do not report many of the semantic errors that seem prevalent among novices.

We are exploring the design of code smell detectors that are specific to mistakes that novices make. While it is trivial to detect individual mistakes reported by instructors and researchers (e.g., [1]), this may not generalize to other programming languages or programming assignments. In an effort to overcome this limitation, we have combined several

[1]https://rules.sonarsource.com/

large data sets, including Blackbox [5], Stack Overflow, and GitHub. Using this data, we are currently training various classifiers to detect patterns in code that has indicators of being buggy or not buggy.

Our informal surveys on the usefulness of our inquisitive code editor have provided us positive feedback. A primary concern is that students would ignore or be annoyed by the tool's questions, but due to the different workflow and mindset of novices, they seem to be eager for any opportunity to receive help with their code. Additionally, we asked how students felt about a tool that would insert code comments or assertions into their code as they worked, and they responded favorably. When speaking with instructors for introductory programming courses, they believe such a tool could greatly benefit students, so long as the tool does not solve the bugs for them or let the students become dependent on it.

IV. Related Work

A. *Mistakes Novice Programmers Make*

Novice programmers make a lot of mistakes in their code [1], [3], [4], [10], [24], [28], [49]. Although the majority of the mistakes are syntax errors (e.g., unbalanced parentheses), these errors are often fixed in a relatively short time [1]. One common approach to aiding students with syntax errors is for tools to provide *enhanced compiler error messages* that are easy to comprehend and may provide suggestions as to how to address the error. However, studies so far have had inconsistent or inconclusive results as to how effective they are in benefiting students [2], [7], [12], [41], [44], [45] Another approach that is growing in popularity is blocks-based languages (e.g., Scratch [36] and App Inventor [55]), which mitigate syntax errors completely by using a drag-and-drop interface that does not require memorizing syntax, but these languages have not yet been adopted in industry.

Semantic errors—when the program behaves differently than expected—are also common [1], [4] and even more challenging to fix since compilers often do not provide any warnings about them. In fact, one study found that students took far longer to fix semantic errors than syntax errors, if they were fixed at all [1]. The underlying cause of why novices introduce semantic errors is complex, but may stem from a lack of conceptual knowledge, inadequate problem solving strategies, or extraneous cognitive load [46]. Consider the following snippet of Python code based on a student's homework submission:

```
response = input(''Please enter (y)es or (n)o'')
while response != 'y' or response != 'n':
    response = input(''Please enter (y)es or (n)o'')
```

From the student's perspective, this code should work. The program will ask the user to input 'y' or 'n', and if anything else is typed in, the program will continue to ask for the correct input. However, this contains a semantic error since the logical "or" in the loop condition will cause the loop to iterate forever.

Researchers have proposed other techniques and tools that may alleviate various errors that novices make. One such approach is to provide visualizations of how the program will behave to aid programmers in understanding. A notable example of such a tool is Python Tutor, which displays visualizations of the program's data structures at each step of the code [16]. Other tools include Omnicode, which displays a scatterplot matrix of all run-time values for every variable in the program [30], Theseus, which annotates functions in the code editor with the number of times it was called during the current execution [35], and a tool extension that displays a small graph of how each variable changes over time during execution [23]. Another approach is to generate content to help programmers, such as hints [13], [20], [27], [43], examples [25], [26], [42], tutorials [18], and recommendations [11], [19], [38], [51], [53]. Recently, researchers have began exploring how to enable users to efficiently tutor students over the web [15], [17], [52]. These techniques could all benefit our proposed work in overcoming misconceptions.

B. *Asking Questions about Program Behavior*

A particularly relevant tool to our proposed research is the Whyline [31]–[33], which enables programmers to record program executions and then ask "why" and "why not" questions about the program's behavior. For example, a programmer can use the tool to click on a specific part of their program's interface and ask the question, "why is this button red?" The Whyline will then provide an answer (i.e., the relevant code) and allow the programmer to ask follow-up questions, such as "why did this event not occur?" To provide these answers, Whyline records an extensive amount of information from a program's execution and uses a combination of static and dynamic program slicing to derive the answers about the program behavior [32]. In an empirical study, participants using Whyline were able to fix bugs in half of the time than those using a traditional debugger [33].

Despite the great benefits that the Whyline provides, there a number of ways in which it does not solve the misconceptions that we are striving to overcome for novice programmers. One potential barrier is that using the Whyline requires the programmer to have already acknowledged that there is a bug and to have recorded the executing program while displaying the buggy behavior. This is insufficient since novices may never even realize that there is a bug in their code. Additionally, novices may find it difficult to find a location in the program that is relevant, then form a question to ask. Beyond preemptively catching misconceptions, our work also aims to correct and prevent misconceptions. Although the Whyline enables the programmer to trace the lines of code that may be relevant to a bug, it does not provide any additional explanation that may be needed for a novice to understand. Moreover, the Whyline does not provide any features that prevent the programmer from having similar misconceptions in the future. Even though the Whyline has made tremendous progress in improving the debugging process, it is clear that there is still a gap in tools that support novices in overcoming semantic errors.

C. Tutoring Systems

To provide explanations and feedback about the program behavior, we will utilize findings from research on *tutoring systems*. These systems provide some form of instruction or feedback to students as if it was an automated teacher. One such system, PLTutor, provides short lessons along with a code editor and visualization of the program state [40]. Another tutoring system is Ludwig [48], which provides students feedback on code style and compares the student's program output to the instructor's program output. Furthermore, a number of programming tools have augmented features of tutoring systems into code editors, such as providing hints to novices in open-ended programming tasks [27]. AutoTutor, a well-studied tutoring system for many subjects other than programming, has shown great promise in keeping students engaged and identifying students' emotions using mixed-initiative dialog in natural language [8], [14]. These techniques could be essential for the tool to be used effectively by novice programmers.

V. Conclusion

The next major steps in our research towards making this idea a reality include conducting a laboratory study and a field study. With a lab study involving our inquisitive code editor, we can understand how users interact with the questions and the effectiveness in how it corrects misconceptions in a controlled environment. To better understand the usefulness of our tool in a more ecologically valid setting, we will deploy it to students enrolled in introductory programming courses to be used while completing their homework assignments. The logging framework that we developed will provide us data on what bugs the students have, the changes they make to fix the bug, how they debug, and how they interact with our tool.

By proposing this initial tool concept and conducting the research necessary to implement the inquisitive code editor, we strive to advance knowledge in the areas of *software engineering*, *human-computer interaction*, and *computing education* in order to help novice programmers understand what their code is really doing. With the ever growing need for more people with computer programming skills, it is of paramount concern to reduce the barriers that may deter people from obtaining these skills. Understanding code is an already arduous process that students go through, thus our work aims to alleviate these difficulties and to support a diverse population of students in learning how to program.

VI. Acknowledgments

This material is based upon work supported by the National Science Foundation under grants 1850027 and 2008408.

References

[1] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 522–527.

[2] B. A. Becker, "An effective approach to enhancing compiler error messages," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, ser. SIGCSE '16. New York, NY, USA: ACM, 2016, pp. 126–131.

[3] B. A. Becker, "A new metric to quantify repeated compiler errors for novice programmers," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '16. New York, NY, USA: ACM, 2016, pp. 296–301.

[4] 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: ACM, 2014, pp. 43–50.

[5] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: ACM, 2014, pp. 223–228.

[6] J. Cao, S. D. Fleming, M. Burnett, and C. Scaffidi, "Idea Garden: Situated support for problem solving by end-user programmers," *Interacting with Computers*, vol. 27, no. 6, pp. 640–660, Nov. 2015.

[7] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *Proceedings of the 2014 Conference on Innovation &#38; Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: ACM, 2014, pp. 273–278.

[8] S. D'mello and A. Graesser, "Autotutor and affective autotutor: Learning by talking with cognitively and emotionally intelligent computers that talk back," *ACM Trans. Interact. Intell. Syst.*, vol. 2, no. 4, pp. 23:1–23:39, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2395123.2395128

[9] I. Drosos, P. J. Guo, and C. Parnin, "Happyface: Identifying and predicting frustrating obstacles for learning programming at scale," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2017, pp. 171–179.

[10] A. Ebrahimi, "Novice programmer errors: Language constructs and plan composition," *International Journal of Human Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994.

[11] E. Fast, D. Steffee, L. Wang, J. R. Brandt, and M. S. Bernstein, "Emergent, crowd-scale programming practice in the ide," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2491–2500.

[12] T. Flowers, J. Jackson, and C. Carver, "Empowering students and building confidence in novice programmers through gauntlet," in *34th Annual Frontiers in Education, 2004. FIE 2004.(FIE)*, vol. 00, 10 2004, pp. T3H/10–T3H/13 Vol. 1.

[13] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "Codehint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 653–663.

[14] A. C. Graesser, P. Chipman, B. C. Haynes, and A. Olney, "Autotutor: an intelligent tutoring system with mixed-initiative dialogue," *IEEE Transactions on Education*, vol. 48, no. 4, pp. 612–618, Nov 2005.

[15] P. J. Guo, J. White, and R. Zanelatto, "Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 79–87.

[16] P. J. Guo, "Online python tutor: Embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 579–584.

[17] P. J. Guo, "Codeopticon: Real-time, one-to-many human tutoring for computer programming," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, ser. UIST '15. New York, NY, USA: ACM, 2015, pp. 599–608.

[18] K. J. Harms, D. Cosgrove, S. Gray, and C. Kelleher, "Automatically generating tutorials to enable middle school children to learn programming independently," in *Proceedings of the 12th International Conference on Interaction Design and Children*, ser. IDC '13. New York, NY, USA: ACM, 2013, pp. 11–19.

[19] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1019–1028. [Online]. Available: http://doi.acm.org/10.1145/1753326.1753478

[20] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations

of online code," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2015, pp. 3–12.

[21] A. Z. Henley and S. D. Fleming, "Yestercode: Improving code-change support in visual dataflow programming environments," in *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '16, 2016, pp. 106–114.

[22] A. Z. Henley and S. D. Fleming, "CodeDeviant: Helping programmers detect edits that accidentally alter program behavior," in *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '18, 2018.

[23] J. Hoffswell, A. Satyanarayan, and J. Heer, "Augmenting code with in situ visualizations to aid program understanding," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: ACM, 2018, pp. 532:1–532:12. [Online]. Available: http://doi.acm.org/10.1145/3173574.3174106

[24] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '03. New York, NY, USA: ACM, 2003, pp. 153–156.

[25] M. Ichinco, W. Hnin, and C. Kelleher, "Suggesting examples to novice programmers in an open-ended context with the example guru," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sept 2016, pp. 230–231.

[26] M. Ichinco, W. Y. Hnin, and C. L. Kelleher, "Suggesting api usage to novice programmers with the example guru," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: ACM, 2017, pp. 1105–1117.

[27] M. Ichinco and C. Kelleher, "Semi-automatic suggestion generation for young novice programmers in an open-ended context," in *Proceedings of the 17th ACM Conference on Interaction Design and Children*, ser. IDC '18. New York, NY, USA: ACM, 2018, pp. 405–412.

[28] J. Jackson, M. Cobb, and C. Carver, "Identifying top java errors for novice programmers," in *Proceedings Frontiers in Education 35th Annual Conference*, Oct 2005, pp. T4C–T4C.

[29] W. Jernigan, A. Horvath, M. Lee, M. Burnett, T. Cuilty, S. Kuttal, A. Peters, I. Kwan, F. Bahmani, and A. Ko, "A principled evaluation for a principled Idea Garden," in *Proc. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '15)*, Oct. 2015, pp. 235–243.

[30] H. Kang and P. J. Guo, "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*, ser. UIST '17. New York, NY, USA: ACM, 2017, pp. 737–745.

[31] A. J. Ko and B. A. Myers, "Designing the whyline: A debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '04. New York, NY, USA: ACM, 2004, pp. 151–158.

[32] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 301–310. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368130

[33] A. J. Ko and B. A. Myers, "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 1569–1578.

[34] Y. B.-D. Kolikant, "Students' alternative standards for correctness," in *Proceedings of the First International Workshop on Computing Education Research*, ser. ICER '05. New York, NY, USA: ACM, 2005, pp. 37–43.

[35] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2481–2490.

[36] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[37] D. C. McFarlane, "Comparison of four primary methods for coordinating the interruption of people in human-computer interaction," *Human-Computer Interaction*, vol. 17, no. 1, pp. 63–139, 2002.

[38] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann, "Crowdsourcing suggestions to programming problems for dynamic web development languages," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '11. New York, NY, USA: ACM, 2011, pp. 1525–1530.

[39] S. C. Müller and T. Fritz, "Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 688–699.

[40] G. L. Nelson, B. Xie, and A. J. Ko, "Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in cs1," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: ACM, 2017, pp. 2–11.

[41] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 168–172.

[42] S. Oney and J. Brandt, "Codelets: Linking interactive documentation and example code in the editor," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12. New York, NY, USA: ACM, 2012, pp. 2697–2706.

[43] B. Peddycord III, A. Hicks, and T. Barnes, "Generating hints for programming problems using intermediate output," in *Educational Data Mining 2014*. Citeseer, 2014.

[44] R. S. Pettit, J. Homer, and R. Gee, "Do enhanced compiler error messages help students?: Results inconclusive." in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '17. New York, NY, USA: ACM, 2017, pp. 465–470.

[45] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. Cohen, "On novices' interaction with compiler error messages: A human factors approach," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17. New York, NY, USA: ACM, 2017, pp. 74–82.

[46] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, pp. 1:1–1:24, Oct. 2017.

[47] M. M. T. Rodrigo and R. S. Baker, "Coarse-grained detection of student frustration in an introductory programming course," in *Proceedings of the fifth international workshop on Computing education research workshop*. ACM, 2009, pp. 75–80.

[48] S. C. Shaffer, "Ludwig: An online programming tutoring and assessment system," *SIGCSE Bull.*, vol. 37, no. 2, pp. 56–60, Jun. 2005.

[49] J. C. Spohrer and E. Soloway, "Novice mistakes: Are the folk wisdoms correct?" *Commun. ACM*, vol. 29, no. 7, pp. 624–632, Jul. 1986. [Online]. Available: http://doi.acm.org/10.1145/6138.6145

[50] I. Stamouli and M. Huggard, "Object oriented programming and program correctness: The students' perspective," in *Proceedings of the Second International Workshop on Computing Education Research*, ser. ICER '06. New York, NY, USA: ACM, 2006, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/1151588.1151605

[51] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, and B. Hartmann, "Tracediff: Debugging unexpected code behavior using trace divergences," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct 2017, pp. 107–115.

[52] J. Warner and P. J. Guo, "Codepilot: Scaffolding end-to-end collaborative software development for novice programmers," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17. New York, NY, USA: ACM, 2017, pp. 1136–1141.

[53] C. Watson, F. W. Li, and J. L. Godwin, "Bluefix: using crowd-sourced feedback to support programming students in error diagnosis and repair," in *International Conference on Web-Based Learning*. Springer, 2012, pp. 228–239.

[54] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel, "Harnessing curiosity to increase correctness in end-user programming," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '03. New York, NY, USA: ACM, 2003, pp. 305–312. [Online]. Available: http://doi.acm.org/10.1145/642611.642665

[55] D. Wolber, H. Abelson, E. Spertus, and L. Looney, *App Inventor*. " O'Reilly Media, Inc.", 2011.