A Tool for Authoring Programs that Automatically Distribute Feedback to Novice Programmers

Michelle Ichinco[1], Yoanna Dosouto[2], Caitlin Kelleher[1]

[1]Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
{ichincom, ckelleher}@seas.wustl.edu

[2]School of Computing and Information Sciences
Florida International University
Miami, FL, USA
ydoso001@fiu.edu

***Abstract*— One way to provide feedback to independent novice programmers is by leveraging experienced programmers as code reviewers. To provide this feedback at a large scale, experienced programmers can author heuristic programs, or rules, that automatically determine whether a novice program should receive certain feedback. This work presents the lessons learned from designing a tool to enable rule authoring.**

Keywords—crowdsourcing; code review; novice programming

I. Introduction

While most middle and high schools lack computer science classes, novice programming environments can provide a motivating context for students to learn programming [1]. However, novices programming independently have no way to receive the type of unsolicited feedback a teacher provides. While systems exist to help novices fix syntax errors and bugs [2], they cannot help novices recognize missed opportunities for improving their code. Prior work proposed crowdsourcing experienced programmers as code reviewers for novice programs [3]. In this system, reviewers make *suggestions* to novice programs, and then author heuristic programs, called *rules*, that determine when another novice program should receive a suggestion. For example, if a suggestion condensed multiple identical consecutive lines of code into a loop, the rule would iterate over the lines of code in a program, checking whether there are multiple identical lines of code, such that the system should suggest a loop. This concept of a rule is similar to static code analysis used in industry, but in a crowdsourced system, the novice code and rule programming language will likely be unfamiliar to reviewers. In this poster, we discuss work towards a tool to enable a diverse population of reviewers to author rules.

We ran an iterative, formative study to design and evaluate a rule authoring tool. We recruited 31 participants from Washington University in St. Louis and through the Academy of Science of St. Louis mailing list who had at least two years of programming experience. Participants made suggestions by improving code and used the reviewer tool to author rules. This study aimed to answer two questions based on tool usage:

1. What barriers exist in rule authoring?
2. How can we enable reviewers to access information about novice code naturally by proving API methods?

II. Rule Authoring Tool

The rule authoring tool is composed of: a suggestion display (Fig. 1), and a rule authoring pane (Fig. 2), both of which expose the Application Programing Interface, or API. The API provides the reviewer with methods for accessing aspects of the novice program, such as the method names or parameter values of a line of code. We initially hypothesized that presenting the API in the context of the novice code would provide easy and meaningful access to our API. Since reviewers focus on the structure of novice code in writing rules, we chose to enable reviewers to click directly on novice code to reveal the API methods associated with the clicked method, construct, or parameter (Fig. 1). For ease of typing complex API methods, we also included autocomplete in the rule editor, shown in Fig. 2, but expected participants to use the suggestion display to understand how to use the API methods.

A. Suggestion display

The code suggestion display, shown in Fig. 1, provides a view of the changes in the code that the reviewer made to improve the novice program. In addition to providing access to the API methods, this display has two other purposes: 1) it can be used to remind users of the suggestion they made by showing the code changes through red highlight of changed code, and 2) it displays the rule results by outlining code in green that the rule finds could benefit from improvement.

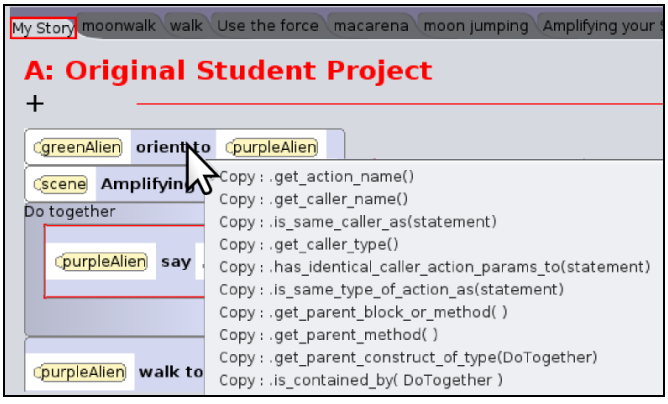

Fig. 1. The suggestion display highlights that code has been inserted before "orientTo" and that "purpleAlien say" has been modified. When a reviewer clicks on "orientTo," a list of API methods appears for that type of code. Selecting one will copy it to the clipboard.

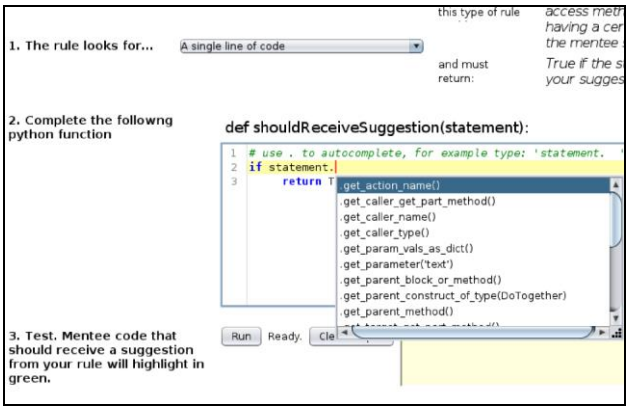

Fig. 2. The API methods are available through autocomplete in the rule authoring editor.

B. *Rule authoring pane*

Rule authoring has three elements: rule type selection, rule authoring editor, and rule testing, shown in Fig. 2. The user first has to choose what type of rule they want to write, which determines how the rule will iterate over the lines of code in the program. These types were created based on common pseudocode rules written in a prior study [3]. For example, one rule type provides access to each line of code in the program, while another enables checking the conditions and body of loops in a program. Reviewers then author their rule using Python in an editor. At any point, users can choose to run their rule, which provides two forms of feedback: console output displayed below the code editor and rule results, shown on the suggestion display.

III. Lessons Learned

We tested four iterations of this tool, which I will refer to as RTs (reviewer tools), shown in Table I. We found that A) encouraging reviewers to test rules early could offset the effects of using an unfamiliar language, and B) providing standard API documentation may better enable rule authoring.

A. *Testing early may reduce the barrier of a new language*

Since there is no language that all reviewers are guaranteed to know, this tool needs to provide affordances to enable users to participate regardless of their language-specific knowledge. In this study, 61% of participants had never used Python. Initially, the coding and testing panes were minimized to reduce the complexity of the interface, requiring the reviewer to expand each pane separately. This discouraged participants from running their rules early and caused Python syntax errors to propagate throughout the rule, increasing the time it took to author rules. From RT2 on, we combined the rule authoring and testing panes to encourage users to test earlier. Although based on a small sample size, the average amount of time participants authoring their first rule for RT1 before pressing the run button was about 20 minutes, while the average for RT2-4 was about 6 minutes, showing a decreasing trend. In addition to this change, we also provided a python overview sheet to help participants acclimate to Python syntax more quickly, which may have also impacted rule authoring times.

TABLE I. FOUR ITERATIONS OF RULE AUTHORING TASK

Iteration (Participants)	Resource Changes	Interface Changes
RT1 (4)	Original	Original
RT2 (10)	Python overview sheet Hint if API not found within 2 min.	Improved instructions pointint to API. Combined rule authoring editor and testing panes
RT3 (7)	None	Rule type descriptions immediately visible.
RT4 (10)	None	Examples added to API descriptions in autocomplete. Emphasized autocomplete in API instructions

B. *Reviewers gravitate toward standard IDE and API support*

We found three main issues in the initial API presentation: 1) participants did not expect to access the API by clicking on novice code, 2) splitting up the API methods makes it difficult to understand the API as a whole, and 3) the presentation lacked details and examples for the API methods. To direct users to the API presentation shown when a user clicks on novice code, we added hints in RT2, which may have been a factor in the decrease in time to author a first rule from an average of 37 minutes in RT1 to 18 minutes in RT2-4. However, participants still struggled to choose the API method they needed and to understand exactly how the methods functioned. For example, participant 21 stated, "I think I was just confused between the difference of the two," when deciding between two comparison methods that did not specifically indicate which parameters they compared. In RT4, we added more complete explanations and examples of the API methods to help reviewers better understand exactly what each method does. Additionally, we changed instructions in the interface to focus users on autocomplete, which allowed participants to see all of the available API methods at once. These changes aligned the interface more with standard API documentation, but further work is still needed to improve presentation of the API so that reviewers unfamiliar with the programming environment can quickly acclimate to the API.

IV. Future Work

These findings indicate that a priority for a rule authoring tool in a novice programming environment is conveying how to access information about the novice code in order to enable reviewers to write programs that check for certain issues. Since experienced programmers do not have a vocabulary for referring to the novice code, we need to integrate documentation with a clear way to understand which novice code the API methods access.

References

[1] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in *Proc. SIGCHI conf.on Human factors in comput. systems*, 2007, pp. 1455–1464.

[2] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proc. 28th int. CHI*, 2010, pp. 1019–1028.

[3] M. Ichinco, A. Zemach, and C. Kelleher, "Towards generalizing expert programmers' suggestions for novice programmers," in *VL/HCC, 2013 IEEE Symp. on*, 2013, pp. 143–150.