Towards Crowdsourced Large-Scale Feedback for Novice Programmers

Michelle Ichinco
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri, USA
ichincom@seas.wustl.edu

I. Introduction

Often, students do not have access to computer science classes in school. Novices learning programming independently can choose to use online resources like tutorials, books, and forums, but they have no way of receiving unsolicited feedback from an experienced programmer or teacher. This feedback is especially important for novices, as they may not always know what questions to ask in order to gain new skills.

I propose a crowdsourced large-scale feedback system for novice programmers powered by experienced programmers, or code reviewers (who I will refer to as "reviewers"). Reviewers have two jobs: making *suggestions* to improve novice programs and authoring *rules* that generalize when a program should receive their suggestion. A rule is a heuristic program that can be run on a novice program to determine whether the system should present the suggestion to the novice programmer. For example, imagine a novice program that contains a certain method call repeated three times in a row. A reviewer might suggest to improve the program by replacing the three identical method calls with a loop. The reviewer would then author a rule that checks whether code in other novice programs also contains repeated lines of code. If the rule determines that a novice program does have repeated code, the system would present the suggestion to the novice programmer to use a loop by showing an example of correct loop usage.

Existing code evaluation tools for professional programmers and students do not currently provide unsolicited feedback with the goal of introducing new programming skills. Static code analysis tools, such as FindBugs [1], and code smell detection methods [2] can automatically evaluate code and locate potential issues, but these tools focus on coding issues much more complex than those in novice programs. Automatic grading systems, such as those described in Ihantola's review [3], and crowdsourced bug fix assistance [4], can aid novices in fixing code problems, but these systems focus on correctness of code output, as opposed to finding opportunities to teach better programming practices.

In this work, I discuss 1) our prior work that supports leveraging experienced programmers as reviewers, 2) a prototype tool for reviewers to make suggestions, author rules, and evaluate the quality of suggestions, and 3) my proposal for presenting the crowdsourced feedback to novices and enabling reviewers to improve each others' feedback.

II. Experienced Programmers as Reviewers

Our previous work toward providing large-scale feedback for novices includes an exploratory study of experienced programmer suggestions and rules and a prototype of a tool that enables reviewers to make suggestions, author rules, and evaluate suggestions and rules [5].

A. *Reviewers Can Create Suggestions and Rules*

We ran an exploratory study to investigate the types of suggestions experienced programmers make to novices' programs and whether reviewers can author pseudocode rules that indicate whether their suggestion applies to any other novice program [5]. In this study, participants first modified novice code to make improvements. Participants then authored rules for their suggestions using pseudocode in a text document. We qualitatively categorized the suggestions and rules participants created to analyze 1) types of suggestions, 2) rule quality, and 3) types of rule pseudocode.

We found that the majority of suggestions improved the quality of the novice code, as opposed to improving the output of the program. In this case, the output of a program is a 3D animation, as we propose this system within the novice programming environment Looking Glass. Since the aim of this feedback system is to enable novices to learn valuable programming skills, we believe having suggestions focus on improving code quality, as opposed to improving output animations, will help novices improve their programming skills.

I used the categorizations of rule quality and rule pseudocode to inform the design of a prototype for suggestion and rule creation. The types of pseudocode participants used to author these rules shaped the design of the rule authoring interface, while the quality of rules indicate that having reviewers evaluate each others' suggestions and rules could be valuable. The quality of a rule is determined by whether the rule correctly finds novice programs that benefit from its associated suggestion. The majority of rules were either appropriate without editing or easily fixable, which is a promising result. However, a crowdsourced feedback system needs to ensure that poorly written rules do not present feedback inappropriately to novice programmers. Thus, a tool should have reviewers evaluate each others' suggestions and rules in order to determine which feedback needs improvement.

B. A Crowdsourced Reviewer Tool

Using an iterative design process, I implemented a prototype tool to enable experienced programmers to complete the following objectives: make a suggestion, author a rule, and evaluate suggestions and rules other reviewers have created. In the study, participants completed each objective for up to two tasks. Participants, on average, took less than 22 minutes to complete each of the objectives for their first task, with rule authoring taking the longest. On their second use of the system, they were able to complete each of the objectives in less than 12 minutes on average, which is promising for a crowdsourced system where users will be volunteering their time.

However, not all reviewers completed each of the objectives. For example, 84% of participants evaluated suggestions and rules for both tasks, but only 48% of participants authored rules for both tasks. This discrepancy in completion between tasks results from different programming experience levels in participants. Thus, some reviewers will only evaluate suggestions, while others will create suggestions or rules. This distribution of reviewers could be very effective in producing a small set of suggestions and rules that have been evaluated and vetted by a large crowd of reviewers. Further, participants' evaluations of reviewer rules reinforce the need for further improvement of rules before presenting them to novice programmers.

III. Improving and Presenting Feedback

My prior work indicates that a crowdsourced reviewer tool would benefit from having a way for reviewers to improve each others' suggestions and rules. The system then needs to effectively present the resulting suggestions to novice programmers.

I hypothesize that A) the iterative process of a group of reviewers creating, editing and evaluating suggestions and rules will produce valuable suggestions that are presented to programs where they apply, B) presenting reviewer suggestions as worked examples will enable novices to learn new programming concepts, and that C) we can motivate novices to use suggestions by providing achievements, varying the type of suggestion based on skill level, and by presenting suggestions when novices are actively working on the code that the feedback applies to.

A. Evaluating and Improving Feedback

We have demonstrated that experienced programmers have the potential to act as reviewers to novice programmers by creating suggestions and authoring rules, but unfortunately, not all feedback created by experienced programmers will necessarily benefit novices. For example, reviewers sometimes want to split long methods into smaller ones, but their choices of how to split the methods are not always beneficial. Our prototype tool enables reviewers to evaluate whether a a rule correctly suggests an improvement to a novice programmer, but does not yet allow reviewers to modify problematic rules. I plan to expand the current reviewer tool to enable reviewers to view evaluations of existing suggestions and rules and improve suggestions and rules that receive poor evaluations. The evaluations will indicate specific lines of code where reviewers either believe that a suggestion should or should not apply. We can then supply the evaluations of specific lines of code as test cases for improving a rule.

B. Presenting Suggestions

I propose presenting the changes reviewers make to novice programs as worked examples in the context of the programs novices are working on. A worked example shows the steps to solving a problem similar to a problem a student needs to solve. Worked examples have been shown to be highly effective for teaching problem solving in a variety of topics, such as Algebra [6], but are commonly created by teaching professionals for specific assignments. In order to test whether novices can learn from suggestions in the form of worked examples, I will design a worked example presentation within the novice programming environment that notifies users that they have suggestions and indicates which code may be improved with the worked example. The worked examples will show the suggestion code, include a description of the goal of the feedback, and also enable a user to run the worked example code.

C. Motivating Use of Unsolicited Feedback

When students receive unsolicited feedback in a classroom setting, they are motivated by the impact of the feedback on their grades. However, independent novice programmers may not have intrinsic motivation to use a suggestion that improves the quality of their code without a visible impact on the output. I propose three ways to motivate novice programmers to use unsolicited feedback: provide achievements for completing suggestions, vary the type of feedback based on the user's skill level, and present suggestions at times when users are most likely to find them helpful. Achievements may be one way to motivate users to complete tasks they would not have otherwise chosen to complete. However, when a novice is just beginning to learn programming, they may not want to use a suggestion that has no visible impact, such as one that refactors code. I propose presenting animation suggestions in the beginning to make the feedback more appealing. Additionally, timing the presentation of suggestions to align with the novice's current task will likely provide more motivation than if the suggestion applies to code the user has already decided is complete.

References

[1] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[2] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999, 474 pp.

[3] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proc. 10th Koli Calling Int. Conf. on Computing Ed. Research*, ser. Koli Calling '10, ACM, 2010, pp. 86–93.

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

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

[6] J. Sweller and G. A. Cooper, "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, no. 1, pp. 59–89, Mar. 1985.