# Towards Generalizing Expert Programmers' Suggestions for Novice Programmers

Michelle Ichinco, Aaron Zemach, Caitlin Kelleher
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
michelle.ichinco@wustl.edu, zemacha@seas.wustl.edu, ckelleher@cse.wustl.edu

***Abstract*—Novice programmers may lack the experience to recognize opportunities to either improve their code or apply unfamiliar programming constructs. Yet, these opportunities are often clear to an experienced programmer. In this paper, we describe an exploratory study investigating 1) the potential value of the suggestions experienced programmers make to novice programmers and 2) the ways experienced programmers envision identifying other programs that would benefit from the same suggestion. The results of our study suggest that experienced programmers make suggestions that can introduce new programming constructs to novice programmers. The participants in our study most commonly made suggestions that improve the code quality of novice programs, rather than changing their output. Furthermore, experienced programmers could often state a simple heuristic rule to use in identifying other novice programs that would benefit from their suggestion. Participants were able to author the rules in pseudocode, mostly using combinations of iteration and comparison to find patterns of problematic code. However, based on a test implementation of a selected set of rules for these suggestions, we conclude that support for improving rules through review and community input will be valuable.**

***Keywords— novice programming; independent learning; static code analysis; crowdsourcing***

## I. Introduction

Estimates predict that low enrollment in computing degree programs will leave almost 40% of American computing jobs unfulfilled in 2018 [1], with a similar outlook in Europe [2] and the Asia-Pacific region [3]. Contributing to this problem, many students opt out of studying computer science long before they reach college [4]. Yet, between 2005 and 2009, the number of secondary schools offering introductory and AP computer science classes decreased by 17% and 35%, respectively [5]. These numbers are unlikely to improve in the near future, due to a lack of certified computer science teachers and the absence of computer science in many state curriculum standards [5]. Currently, there are three common approaches to learning programming without formal computer science education: after-school programming classes, reusing code from the internet, and free online programming classes.

Programming opportunities outside of schools, like Scratch workshops in the Computer Clubhouse [6], are one way for students to gain exposure to programming. Yet, camps and workshops like this are extremely limited and require a time commitment from in-person teaching support to provide the necessary feedback to students.

Without formal education, novice programmers often reuse code examples from the internet to fill in the gaps. However, end-user non-programmers often do not understand example code [7], which can lead to incorrect code usages and errors. Further, novice programmers cannot seek out techniques that they do not know exist. Therefore, learning programming from reusing code cannot replace the guidance and individualized feedback students would likely receive in school.

Novice programmers who do not have access to programming classes in school can also take free online computer science courses [8], [9]. Yet, retention rates for online classes are low compared to US colleges [10], indicating that self-motivation is necessary to take these classes.

A less readily available alternative to learning programming in school is for a novice programmer to get tips from an expert programmer they already know, such as a friend or family member. Imagine two hypothetical middle school students named Beth and Carl, who are learning programming independently and do not have access to programming classes.

*Beth is programming a synchronized swimming animation and finds a backflip procedure online. She reuses it in her program, but the backflip procedure is specific to a certain object. Not realizing she can generalize the existing method, she recreates an identical backflip method for each of the synchronized swimmers. Carl, on the other hand, shows his uncle, who is a software engineer, a program where he has created an identical swim method for each of five fish. Carl's uncle suggests that he generalize the method for all fish.*

Currently, there is no way for this suggestion to reach Beth. If we can capture the suggestion and determine that Beth's program would also benefit from it, Beth might learn how to generalize her backflip method.

We ran an exploratory study to investigate crowdsourcing expert programmer suggestions to enable mentoring novice programmers at a large scale. We hypothesize that expert programmers can make suggestions to novice programmers' code and write rules for those suggestions. We define a rule as a simple program that evaluates whether a novice program

This material is based upon work supported by the National Science Foundation under Grant No. 1054587.

should receive a suggestion. If possible, this would enable a large population of novice programmers to benefit from a relatively small number of suggestions. In this study, we asked participants to suggest an improvement for a novice program and author a rule to identify more general patterns in code that indicate that their suggestion would be appropriate. This paper addresses the following questions, 1) do participants make suggestions that have the potential to teach a novice how to improve their program, 2) what does the rule pseudocode tell us about how to design support for authoring rules, and 3) in practice, how well do rules generalize whether a program is a potential "target," meaning it should receive the suggestion?

## II. Related Work

This study explores whether expert programmers can identify problematic code, provide suggestions to improve novice code, and then generalize those suggestions. While no one has addressed this question directly, our work builds on a variety of research areas involving code analysis, which define and evaluate either code correctness or code quality.

### A. Code Correctness

Researchers have made progress on evaluating code correctness in the following areas: automatic grading systems, finding bugs and errors in professional code automatically, and in novice code using crowdsourcing.

Automated tests that check code correctness in programming assignments provide quick feedback to a large number of students and allow professors to assign more assignments. A variety of automated grading systems [11]–[17] allow teachers to specify assignments and tests that evaluate the correctness of code output. While these systems require assignments with defined answers, independent learners using novice programming systems often work on open ended programs that do not have right or wrong answers.

Evaluating source code can locate bugs and errors, which often affect whether code executes correctly. PREfix and PREfast are systems that successfully determine the density of defects by analyzing code [18]. Static code analysis can also find issues in large-scale multi-threaded programs [19] and detect security vulnerabilities [20], [21]. However, novice programmers, focused on learning programming constructs, are mainly shielded from these complex bugs.

Studies have used crowdsourcing to assist new programmers in understanding compilation errors and bugs. HelpMeOut, BlueFix and Crowd::Debug [22]–[24] utilize example code from a database of users' error fixes, in conjunction with expert explanations, to assist novice programmers in understanding and fixing bugs. Our study extends crowdsourcing and expert programmer advice to present novice programmers with new programming concepts.

### B. Code Quality

Though Anderson and Shneiderman claim that peer review is an appropriate method for evaluating code quality [25], three types of systems define and assess code quality more formally: tools for code quality assessment, a subset of automatic grading systems, and code smell detection methods.

A number of tools allow programmers to check the quality and style of their code: PMD, Klokwork, SourceMonitor, QJ-Pro, and StyleCop [26]–[30]. These tools provide standard metrics and allow customization of metrics, but are aimed at professional programmers and do not have support for suggesting new programming concepts or skills.

A number of automatic grading systems measure the quality of student code using standards and structural properties. Some systems use metrics as code quality measures for student assignments, like the ISO/IEC 9126 standard [31], [32] or Berry and Meekings' style metrics [33], [34]. Other automatic grading systems consider the structure of code. For example, one framework employs cyclomatic complexity [35], [36], while another study uses LOGISCOPE to find knots [37]. Cyclomatic complexity and knots find problematic code by looking at the paths through a program. These evaluation techniques focus on complex structural and style issues or aspects of code such as whitespace, all of which novices can often ignore while learning basic programming concepts.

Fowler and Beck [38] developed categorizations and definitions of "code smells." Code smells are patterns of code, such as long methods or duplicated code, which may indicate a problem. Based on these definitions, a body of research has investigated humans and metrics as detectors of code smells. Several studies [39]–[41] explore human evaluations of code smells, finding low agreement for detection of complex code smells. Mantyla, Vanhanen and Lassenius developed a taxonomy [42] to enable better understanding of code smells for human detection. They found that their taxonomy aligns with correlations software developers noted between code smells. DECOR and "detection strategies" [43], [44] enable humans to operationalize code smells using software metrics, while another set of systems [45]–[47] use metrics to automatically detect code smells [38]. Based on code smells, Cunha, Fernandes, Ribeiro and Saraiva identified smells in spreadsheet code and created a tool to find these smells [48]. Similarly, indications of issues in novice programs will be different from those in software systems. Yet, work on code smells supports the idea that expert programmers may be able to specify opportunities for improvement in novice code.

Current systems test correctness of student code, find bugs and errors, check quality of student and professional programmer code, and find code smells. We address the feasibility of expert programmers making suggestions and then generalizing which programs should receive the suggestions.

## III. Mentoring in Looking Glass

Looking Glass [49] is a drag and drop integrated development environment (IDE) designed to help middle school students learn to program independent of formal education. We chose to explore mentoring in Looking Glass partially because of its integration with an online community, where users can upload their programs. One way for experts to mentor novices is to open a program from the online community, edit it, and then re-submit it, which can be easily streamlined in Looking Glass.

To create a program in Looking Glass, users first make a scene and then drag and drop code tiles to create an animation.

The drag and drop tiles prevent users from making syntax errors. Looking Glass has a set of standard methods for characters, such as *walk*, *say*, and *move*, as well as a set of programming constructs, like the *DoTogether, DoInOrder, If/Else* and *CountLoop*. Users can also create custom methods with sequences of methods and programming constructs.

In Looking Glass, the realization of a mentoring system will involve an interface for mentors to make suggestions and author rules. A suggestion is a code improvement a mentor makes to a novice's program, while a rule is a short program that generalizes whether the suggestion is likely to improve another program. One aim of this study is to inform the design of an Application Programming Interface (API) that will enable experts to author rules without deep knowledge of how Looking Glass stores programs.

## IV. METHODS

We are interested in the suggestions expert programmers make, how expert programmers write rule pseudocode, and whether the rules find appropriate target programs. The study involved an introduction to Looking Glass, followed by four rule authoring tasks, one in which the participant was asked to make their own suggestion.

### A. Study Procedures

This study collects two kinds of information: the types of suggestions programming experts make and rule pseudocode that captures how programming experts envision finding target programs. Due to time constraints, we asked participants to make one suggestion and author a rule that specifies when to offer that suggestion, and then write three additional rules based on premade suggestions. The study has three parts: an introduction to Looking Glass, a "Suggestion/Rule Pair" task, and three "Rule Authoring with Premade Suggestion" tasks. We randomly assigned each participant an example program for each task, such that each participant saw one example program from each of four skill groups in Fig. 1. The four selected skill levels range from beginner to intermediate, which we describe in more detail in the Materials section. Due to a lack of student programs in the advanced skill group, we did not include advanced example programs in this study.

#### 1) Looking Glass Introduction

To enable participants to make a suggestion by editing a program, we introduced participants to the components of the Looking Glass IDE. After demonstrating a completed program created by a researcher, we asked participants to create a simple program to familiarize themselves with Looking Glass.

#### 2) Suggestion/Rule Pair Task

To investigate the types of suggestions domain experts make for novice programs and how they write the corresponding rules, we asked participants to create a suggestion and then author a rule. We first played an example program animation in Looking Glass and provided the participant with a skill group diagram (Fig. 1) to guide them toward suggestions at an appropriate skill level. We then asked participants to make a suggestion by editing the program.

To explore how expert programmers author rules, we provided participants with a rule-authoring template, which is a

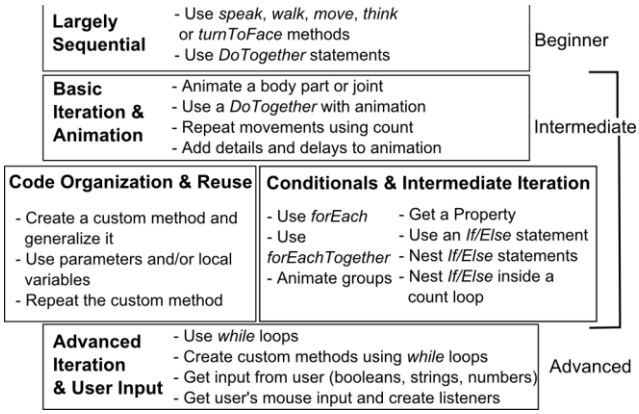


Fig. 1. Skill group diagram

Word document with instructions for the task and space for typing the rule. We asked participants to describe the rule in English, explaining what pattern of code indicates that their suggestion might be useful. We then asked participants to write the rule as a program in pseudocode that, when run on any novice program, determines whether the novice program would benefit from the suggestion they created. Similar to the natural programming approach [50], we asked participants to use their own vocabulary and format and did not provide any information about terms or structure that they should use.

#### 3) Rule Authoring with Premade Suggestion Task

The purpose of the rule authoring tasks with premade suggestions was to explore how participants wrote rules in pseudocode, in order to inform the design of an API. We asked participants to author three rules based on premade suggestions designed by the experimenters. For these tasks, we showed the participant an example program beside a premade suggestion and asked them to author a rule, as in the Suggestion/Rule Pair task, that finds target programs for the provided suggestion.

### B. Participants

This study involved 21 participants, five of whom were female, ranging in age from 19 to 68. We recruited participants through the Academy of Science of St. Louis mailing list because these members represent the type of programming experts who may be interested in mentoring students in the future. Participants' programming backgrounds range from being self-taught to having a Ph.D in Computer Science. Most participants listed their occupation as software developer, software engineer or programmer. Fifteen participants described themselves as an expert in at least one programming language and all had experience programming.

## V. MATERIALS

We created the example programs, premade suggestions, skillset diagrams, and the rule-authoring template used in this study to simulate the experience of a mentor.

### A. Example Programs

Participants made suggestions and wrote rules for "example programs," which are slight alterations of Alice programs [51] and Looking Glass programs created in a 2010 study. In the 2010 Looking Glass study, participants had never programmed before. Alice is a sister project of Looking Glass that has

college-aged users with more advanced skills. The programs were selected to represent a variety of skill levels and such that we could point to at least one potential suggestion per program, but there were often many. The example programs maintain the same structure and content as student programs, with changes mainly in the characters and props. We created example programs due to a lack of student programs compatible with the current system. The example programs also remove information that might identify the original authors.

### B. Premade Suggestions

Each example program has an associated premade suggestion, created by the experimenters and used in the "Rule with Premade Suggestion" tasks. Suggestions were one of two types: code-based or animation-based. Code changes improve the coding style of a program, while animation changes improve the output of the program. An example premade suggestion for an animation change is "A more complex animation with body movements was added where Tami previously danced. She now moves her left leg and arms to be in a dancing position before she turns, instead of just spinning in a standing pose." An example premade suggestion for a code change is "A list of characters all doing the same action was replaced with a *ForEach* loop with the array of characters." To show a participant a premade suggestion, we provided them with a sheet of paper that contained screenshots of the original program and the premade suggestion. The English descriptions of the suggestions were printed above the screenshots.

### C. Skill group diagrams

To help participants make an appropriate suggestion, we provided them with a skill group diagram like Fig. 1. These skill groups are based on a hierarchy of novice programming skills currently being developed, similar to groupings used by the Computer Science Teachers Association [52]. For participants, the diagrams indicated which of these skills the novice programmer likely already knows, which might be appropriate to present next, and which might be too advanced, based on the structures present in the example program.

### D. Rule-authoring template

We chose to use a Word document for rule authoring to minimize the influence of the IDE on the participants' coding style choices. The document contains instructions asking participants to write the rule in a sentence and then in pseudocode, such that it returns "True" if the rule has found a target program and otherwise returns "False."

## VI. Analysis and Results

The results of this study answer three questions: 1) what types of suggestions do programmers make, 2) what does the pseudocode tell us about designing a tool for authoring rules, and 3) what do the target programs tell us about the rules?

### A. Suggestions

To explore the types of suggestions participants made, we used a grounded theory approach [53], which is an iterative process of labeling possibly important features of the data to develop categories and theories. We labeled suggestions based on the suggestion as a whole. This process resulted in hierarchies of categories for Suggestion Type and Suggestion Novelty, based on common labels and relationships between labels. The Suggestion Types group suggestions by the concept or idea the improvement presents. The Suggestion Novelty categories consider whether the suggestion utilizes new concepts. For categorization purposes, new concepts are programming constructs, structures, or methods that the example program does not contain. After developing the categories using grounded theory, two researchers then individually selected categories for the 21 suggestions. This set of suggestions resulted from each of the 21 participants creating a suggestion in the "Suggestion/Rule Pair" task. The categorizations had inter-rater reliabilities of 95% for Suggestion Types and 86% for Suggestion Novelty categories.

#### 1) Suggestion Type

The data labeling and categorization process resulted in two major types of suggestions: code changes and animation changes, as shown in Fig. 2. In a code change, the participant modified the style of the program's code, such as adding a variable when a value is used multiple times for the same purpose. In animation changes, participants modified the animation output of the program, such as creating a new method *fallDown* where the character flails and falls realistically to replace a method that makes a character turn backward without bending any joints. Of 21 participants, 16 improved the code, 4 modified the animation, and 1 changed both the code and the animation. At the most detailed level, there are eight suggestion types with no more than 20% of suggestions in any one category. Further, the majority of suggestions were different than those made by the researcher for the "premade suggestions," indicating that the example programs had a wide variety of possible improvements.

As seen in Fig. 2, two common code change types are creating a new method (4) and restructuring repeated code (4). One new method suggestion is for a program where a man pushes another man into the ocean. The suggestion extracts the animation to a custom method, which makes the action more easily reusable. In another example program, a set of kids each turn to face the camera sequentially. The suggestion, which restructures repeated code, replaces the list of repeated statements with a *ForEach* loop, improving the code style and introducing or reinforcing *ForEach* loops.

Six participants improved the usability of code by generalizing a method (3) or returning the animation to a default state (3). Generalizing a method can be, for example,

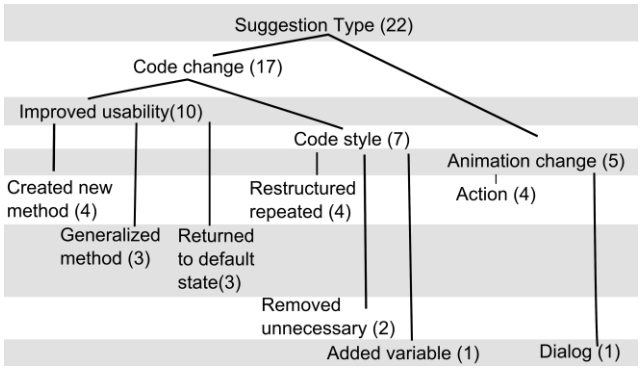


Fig. 2. Suggestion type categories

adding parameters or making a method accessible to a class of objects, rather than a single character. Returning code to a default state involves bringing a character to a position where the animation can continue or is more easily reusable. For instance, one example program has a dancer jump into the air and the suggestion returns the dancer to the ground.

Although we encouraged both code and animation changes, most participants made code changes, for several reasons: lack of familiarity with Looking Glass, fear of changing a child's creation, and difficulty generalizing animation changes. Since participants only had a short introduction to Looking Glass, changing programming constructs or restructuring code was easier than creating animations. Several participants commented that they did not want to change the animation because they were unsure of the original intentions. Other participants considered changing the animations in a program, but stated that they did not believe it would apply to other programs. These results suggest that experts will be more likely to improve novice code than animations.

*2) Suggestion Novelty*

Thirteen of the 21 suggestions utilized explicitly new concepts or constructs, while 10 reinforced constructs or concepts already present in the example program. The majority of suggestions are either in the category "Method creation or abstraction" (9) or "Added method calls" (6). In the suggestion novelty classification in Fig. 3, a method call refers to a provided method, like *walk* or *move* and programming constructs refer to loops and conditional logic. Method creation involves restructuring a sequence of methods into a custom method. Data storage refers to variables and parameters.

Both new concepts and reinforcement can be valuable feedback for novice programmers, as using a concept once does not imply mastery. Yet, it may be useful for the system to keep track of the suggestions given to users in order to prevent overwhelming repetition of suggestions. Further, when a suggestion does not introduce something new, providing an explanation may help students understand why a suggestion will improve their code and what they might learn from it.

## B. Rule Pseudocode

We analyzed the rule pseudocode to inform the design of an API for rule authoring. To enable any programming expert to write rules, the API must provide mentors with access to the information in the novice programs that they need. The rules' pseudocode provides insight into the types of functionality domain experts expect to have while writing rules.

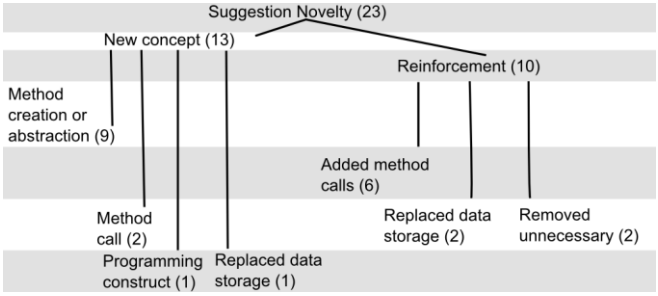


Fig. 3. Suggestion novelty categories

The 21 participants wrote 72 rules, averaging 3.5 of 4 completed tasks, due to time. This set of rules results from each participant completing a "Suggestion/Rule Pair" task and up to three "Rule Authoring with Premade Suggestion" tasks. We analyzed pseudocode with a similar grounded theory approach as used for Suggestion Type. For pseudocode, we labeled each line of code and categorized them individually. The rules contained 287 total lines, not counting lines that were braces or the required return statements. On average, rules contain four lines of code, with a standard deviation of 1.8. Consistent with the method for categorizing suggestions, two researchers independently categorized pseudocode lines, with a 94% inter-rater agreement rate. We will not discuss the 6% of disputed lines, as they fit into multiple categories or were ambiguous. The remaining 94% of lines fall into three overarching categories: iteration, comparison, and "other."

Most of the 24% of pseudocode lines identified as neither iteration nor comparison were either matching functions, attempts to access dynamic information, or count functions. A few participants created template matching functions that defined a set of constraints and then checked whether any lines met those constraints. Other participants checked information only available at runtime, such as the location of characters. However, rules with access only to the static code cannot check runtime information. Several participants used functions to count the number of times a line occurs. Because participants rarely used these types of functions or we deemed the functions not viable, we focus on iteration and comparison.

The remaining 70% of pseudocode lines are either iteration or comparison, indicating that the API should focus on allowing mentors to iterate through programs and perform comparisons to find code patterns. Fig. 4 shows an example of a rule that has the typical pseudocode structure: line 1 iterates through each of the methods and line 2 compares the name of the current method call to the name of the next method called.

*1) Iteration*

The iteration lines fall into six categories: three ways of iterating through lines of code, and iterating through parameters, sets of objects, and programming constructs. The number of pseudocode lines for each of the categories is shown in Fig. 5. When iterating through lines of code, some participants assumed access to a set of lines, in a *ForEach(line)* style, for the whole program or within a certain scope. Surprisingly, a number of participants parsed the program as strings in a *While!(EndofDocument)* style. However, it will be less work for mentors to iterate through elements in the program than to write complex regular expressions to parse strings. By far, the most used iteration style was through each line in the program, but providing support for checking conditions in a certain construct or in a custom method is also likely to be useful.

*2) Comparison*

Participants often used comparison to determine whether a line or group of lines contain a certain issue. Comparisons fell

```
1: foreach(allMethods as k => method)
2:   if(method->name == allMethods[k +1]->name)
3:     return true;
```

Fig. 4. Example rule pseudocode

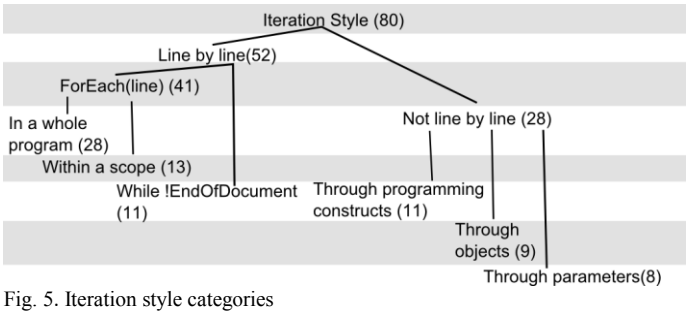

Fig. 5. Iteration style categories

into three high-level groups: comparison of multiple lines of code, whole lines or methods, and part of a line or method, as shown in Fig. 6. The differences result from participants either envisioning the code as a string or as a structure. Eight participants treated programs as strings, parsing and pattern matching with functions like *line.contains("methodName")*. The other 13 participants assumed that programs have a structure that they could query for information, such as accessing the method name with a function like *line.getName()*. These categories imply that the API design should focus on accessing parts of a method and enabling comparison of entire lines of code. Only 5 pseudocode lines compared multiple lines of code, but the amount of code required to accomplish this manually suggests that API support would be useful.

We used the pseudocode categories to develop a preliminary API, which enables iterating through lines of code, iterating through programming constructs, comparing consecutive lines of code, comparing multiple lines of code, comparing non-consecutive lines of code, checking values of parameters, and comparing parameters of multiple methods.

### C. Rule Implementation

To evaluate whether rules are likely to work in practice, we implemented each of the rules from the "Suggestion/Rule Pair" task using the preliminary API. We chose to implement these rules because they are based on suggestions participants' made, rather than premade suggestions. Two pairs had identical implementations, so we tested the remaining 19 rules.

We tested each rule on the 165 programs currently uploaded to the Looking Glass Community. We will refer to the Looking Glass Community programs as the "mixed group," since Looking Glass Lab members, ranging from college sophomores to a professor, contributed 92 of these programs. We also tested the rules on 55 programs created by middle school children, the "novice group," in an unrelated 2013 study. The novice group did not receive instruction on how to

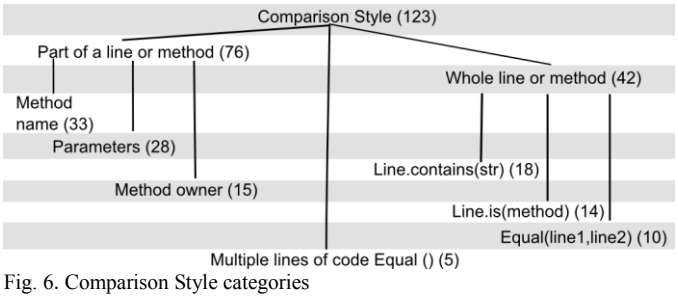

Fig. 6. Comparison Style categories

program. Although this distribution of test programs is not ideal, program authors do range from novice to advanced.

Running each rule identifies the target programs that could benefit from the associated suggestion. Table 1 summarizes the rules, grouped by quality, and reports the percentages of programs in the mixed and novice groups that fit the rules. We initially hypothesized that these percentages might indicate whether a rule is too general or too specific. In practice, other, unrelated factors such as skill level of the user, affect the percentage, making it only a weak indicator of rule quality. Our analysis suggests that rules range in quality from unfixable to immediately applicable. Thus, we focus our discussion of rule quality on the following four types of rules: Unfixable (UF), Bad Code (BC), Fixable Code (FC) or Good Code (GC).

#### *1) Unfixable Code (3 rules)*

Unfixable rules have issues such as presenting suggestions that are either a poor example or irrelevant to the user's program. For example, UF2 looks for sequences of *say* statements and creates a new method, passing in the text strings as parameters. In an object-based context, this presents an unrepresentative example of using a new method. In most cases, Looking Glass programmers use new methods to create and name a cohesive set of actions for a character.

#### *2) Bad Code (2 rules)*

Based on their English descriptions, the two rules in this category are inspired by valid observations. However, the pseudocode rules do not match these descriptions. BC1, for example, intended to look for cases in which a character performs two actions that can be condensed into a single action. For example, a new user might not realize that parameter values can be modified and use a string of *move* statements to position a character, rather than changing the distance moved. In practice, the rule searches for a sequence of two identical lines. While two *move* forward statements can be easily combined, not all actions have that characteristic.

#### *3) Fixable Code (6 rules)*

Fixable code rules are very close to making appropriate suggestions, but the code neglects to check one or more conditions that could improve their results. To get a sense for how these rules might perform if corrected, we also created fixed versions of them. For example, FC1 suggested creating a new method if a character performs 3 actions in a row. A sequence of actions by the same character is often a reasonable place to suggest creating a new method. We modified this rule to check for at 6 actions instead of 3 and to ensure that those actions are not already in a custom method. This dramatically reduced the percentages of target programs for the mixed group from 68 to 8 and for the novice group from 70 to 0. In Table 1, we report the original matching percentage followed by the fixed matching percentage for the mixed and novice groups.

#### *4) Good Code (8 rules)*

Eight of the rules present suggestions to an appropriate set of programs with no improvement necessary. Some of the rules with good code are straightforward programming concepts. For example, GC2 looks for unnecessary *DoTogether* statements, those that contain fewer than two statements. GC3 replaces repeated actions by a set of characters with a *ForEach* loop.

TABLE 1 RULES, RULE ISSUES, AND PERCENTAGES OF PROGRAMS RECEIVING SUGGESTIONS FOR PARTICIPANT RULES

| Rule Quality | Short Description of Rule Intent | Fig.2 Category | Rule Quality Issue | Mixed (%) | Novice (%) |
|---|---|---|---|---|---|
| **Un-fixable Code** | UF1. Program contains a reversible action | Default Pos. | Too Aggressive | 83% | 71% |
| | UF2. Find a sequence of single character *say* statements | New Method | Poor Example | 26% | 36% |
| | UF3. *DoTogether* contains several items | New Method | Poor Example | 63% | 9% |
| **Bad Code** | BC1. Two consecutive identical lines | Remove Rep. | Idea-Code Mismatch | 11% | 0% |
| | BC2. Single character performs an action multiple times | Improve Action | Idea-Code Mismatch | 82% | 95% |
| **Fixable Code** | FC1. Character performs three actions in a row | New Method | Too Aggressive/Incomplete | 68% → 8% | 70% → 0% |
| | FC2. My First Method has *DoInOrder* child | Remove Unnec. | Incomplete | 8% → 13% | 0% → 0% |
| | FC3. Object *moves* more than once in sequence | Remove Rep | Incomplete | 27% → 2% | 7% → 2% |
| | FC4. *Say* statement contains more than eight words | Dialog | Incomplete | 46% → 39% | 56% → 9% |
| | FC5. Custom method for a specific character | Generalize | Incomplete | 36%→ 33% | 7% → 7% |
| | FC6. Repeated use of same duration value in a method | Local Var. | Incomplete | 64% → 64% | 18%→15% |
| **Good Code** | GC1. Sequence of color changes | New Method | N/A | 2% | 2% |
| | GC2. *DoTogether* with fewer than two statements | Remove Unnec. | N/A | 7% | 4% |
| | GC3. Characters perform same statement in sequence | Remove Rep. | N/A | 16% | 15% |
| | GC4. Custom method called more than once contains *move* | Default Pos. | N/A | 14% | 0% |
| | GC5. A character *turns* backward ¼ | Improve Action | N/A | 0% | 0% |
| | GC6. Program contains *moveTo* statement | Improve Action | N/A | 21% | 40% |
| | GC7. Something moves up, never moves down | Default Pos. | N/A | 26% | 6% |
| | GC8. Repeated use of a property value | Generalize | N/A | 41% | 0% |

Other rules in this category were less straightforward. GC1 looks for a sequence of color changes. Initially, we were skeptical that this would be generally useful. However, the cases in which programs repeatedly change colors are mostly implementing a flash behavior. While this suggestion may not be offered with great frequency (only 2% of programs matched), it is likely to be a good suggestion when it is offered.

## VII. CONCLUSIONS AND FUTURE WORK

Overall, nearly three quarters of the rules experts wrote were either strong or readily fixable, which provides some support for the approach of crowdsourcing suggestion-based help for novice programmers. In fact, our study may underestimate the likely percentage of good and readily fixable rules. Participants wrote rules in a Word document to prevent any influence of the IDE on their pseudocode. However, this also removes feedback participants would normally receive when programming. We intend to develop an interface and API to enable rule authoring, editing and testing within Looking Glass. We expect that the ability to test rules prior to submission could substantially improve their quality.

In addition to the lack of feedback, we hypothesize that varying experience could also explain some of the variety in rule quality. Our study attracted an enthusiastic response from computing professionals with a wide range of experience: in addition to software engineers with decades of experience, participants included students, self-taught web developers, and people who used to program but do not currently. However, there appeared to be little relationship between rule quality and experience. This suggests that there will always be variance in rule quality and that we will need to design crowdsourcing approaches with this in mind. One can imagine a collaboration system where a rule requires vetting before "going live." The system would not recommend suggestions to novices until a number of other mentors certify that the rule is appropriate. Mentors could either edit inappropriate rules or vote them Unfixable, in which case they would eventually disappear from the system. In future work, we plan to test the effectiveness of this type of crowdsourced rule evaluation system.

Finally, the nature of the rules implies that we will need to use care when offering suggestions to novice programmers. While some rules identify matches with 100% certainty, others do not. *DoTogethers* with fewer than two statements (i.e. GC2) can always be simplified. In contrast, GC4 is inspired by the observation that methods called repeatedly should be free of side effects, such as ending in a location different from the start position. This rule finds methods called multiple times that contain *move* animations. While this may be appropriate for certain programs, some methods appropriately need the ending position to differ from the starting position. Additionally, this rule looks only for a *move* method, which is not a perfect signal that a method has side effects. Consequently, systems that offer suggestions based on mentor rules will need to be designed such that suggestions are unobtrusive and are not offered endlessly. The development of this suggestion system will also enable evaluation of whether rules and suggestions actually teach novice programmers new programming concepts.

## REFERENCES


[1] National Center for Women & Information Technology, "Computing Education and Future Jobs: A Look at National, State & Congressional District Data," Dec. 2011.

[2] R. Schneiderman, "Competition Heats Up To Fill European Tech Jobs," 2012. [Online]. Available:

http://careers.ieee.org/article/European_Job_Outlook_0312.php. [24-Feb-2013].

[3] R. Schneiderman, "Economic Strains Are Having Little Impact On Tech Job Opportunities In The Asia-Pacific," *IEEE JobSite*, 2012. [Online]. Available: http://careers.ieee.org/article/asiapacific_0812.php. [24-Feb-2013].

[4] J. H. Pryor, S. Hurtado, L. DeAngelo, L. P. Blake, and S. Tran, *The American freshman: national norms, fall 2009*. Amer Coun On Educ., 2005.

[5] "Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age," Assoc. for computing Machinery and The Comput. Sci. Teachers Assoc., 2010.

[6] J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk, "Programming by choice: urban youth learning programming with scratch," in *Proc. 39th SIGCSE technical symp.on Computer sci. educ.*, New York, NY, USA, 2008, pp. 367–371.

[7] P. Gross and C. Kelleher, "Non-programmers identifying functionality in unfamiliar code: strategies and barriers," *Journal of Visual Languages & Computing*, vol. 21, no. 5, pp. 263–276, 2010.

[8] "Coursera," *Coursera*. [Online]. Available: https://www.coursera.org/. [09-Mar-2013].

[9] "Khan Academy," *Khan Academy*. [Online]. Available: http://www.khanacademy.org. [Accessed: 09-Mar-2013].

[10] J. Daniel, "Making Sense of MOOCs: Musings in a Maze of Myth, Paradox and Possibility," *J. Interactive Media in Education*, vol. 3, no. 0, Dec. 2012.

[11] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT: automatically grading programming assignments," in *Proc. 13th annual conf. on ITiCSE*, New York, NY, USA, 2008, pp. 328–328.

[12] M. J. Hull, D. Powell, and E. Klein, "Infandango: automated grading for student programming," in *Proc. 16th annu. joint conf. ITiCSE*, New York, NY, USA, 2011, pp. 330–330.

[13] D. Arnow and O. Barshay, "WebToTeach: An interactive focused programming exercise system," in *Frontiers in Education Conf. FIE'99. 29th Annual*, 1999, vol. 1, pp. 12A9–39.

[14] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu, "APOGEE: automated project grading and instant feedback system for web based computing," *SIGCSE Bull.*, vol. 40, no. 1, pp. 77–81, Mar. 2008.

[15] E. L. Jones, "Grading student programs-a software testing approach," *J. of Computing Sci. in Colleges*, vol. 16, no. 2, pp. 185–192, 2001.

[16] M. Blumenstein, S. Green, A. Nguyen, and V. Muthukkumarasamy, "An experimental analysis of GAME: a generic automated marking environment," *SIGCSE Bull.*, vol. 36, no. 3, pp. 67–71, Jun. 2004.

[17] R. Saikkonen, L. Malmi, and A. Korhonen, "Fully automatic assessment of programming exercises," in *Proc. 6th ann. conf. on ITiCSE*, New York, NY, USA, 2001, pp. 133–136.

[18] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proc. 27th int. conf. on Software eng.*, 2005, pp. 580–586.

[19] C. Artho and A. Biere, "Applying static analysis to large-scale, multi-threaded Java programs," in *Software Engineering. Proc. Australian*, 2001, pp. 68–75.

[20] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, IEEE Symp. on*, 2006, p. 6–pp.

[21] M. S. Ware and C. J. Fox, "Securing Java code: heuristics and an evaluation of static analysis tools," in *Proc. workshop on Static analysis*, New York, NY, USA, 2008, pp. 12–21.

[22] 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*, New York, NY, USA, 2010, pp. 1019–1028.

[23] C. Watson, F. Li, and J. Godwin, "BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair," *Advances in Web-Based Learning-ICWL*, pp. 228–239, 2012.

[24] D. Mujumdar, M. Kallenbach, B. Liu, and B. Hartmann, "Crowdsourcing suggestions to programming problems for dynamic web development languages," in *Proc. 2011 annu. conf. extended abstracts on Human factors in comput. systems*, 2011, pp. 1525–1530.

[25] N. Anderson and B. Shneiderman, "Use of peer ratings in evaluating computer program quality," in *Proc. 15th annu.SIGCPR conf.*, New York, NY, USA, 1977, pp. 218–226.

[26] "PMD." [Online]. Available: http://pmd.sourceforge.net/. [15-Mar-2013].

[27] "Klocwork." [Online]. Available: http://www.klocwork.com/index-v3.php. [Accessed: 16-Jan-2013].

[28] "SourceMonitor." [Online]. Available: http://www.campwoodsw.com/sourcemonitor.html. [21-Feb-2013].

[29] "Code Analyzer for Java." [Online]. Available: http://qjpro.sourceforge.net/index.html. [Accessed: 21-Feb-2013].

[30] "StyleCop." [Online]. Available: http://stylecop.codeplex.com/. [Accessed: 21-Feb-2013].

[31] D. M. Breuker, J. Derriks, and J. Brunekreef, "Measuring static quality of student code," in *Proc. 16th annu. joint conf. on ITiCSE*, New York, NY, USA, 2011, pp. 13–17.

[32] ISO/IEC 9126-1 2001, "Software Engineering- Product Quality – Part 1: Quality Model." Geneva, 2001.

[33] D. Jackson and M. Usher, "Grading student programs using ASSYST," in *Proc. 28th SIGCSE technical symp. on Computer science education*, New York, NY, USA, 1997, pp. 335–339.

[34] R. E. Berry and B. A. E. Meekings, "A style analysis of C programs," *Commun. ACM*, vol. 28, no. 1, pp. 80–88, Jan. 1985.

[35] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' Java programs," in *Proc. 6th Australasian Conf. on Computing Educ.-Volume 30*, Darlinghurst, Australia, Australia, 2004, pp. 317–325.

[36] T. J. McCabe, "A Complexity Measure," *IEEE Trans. on Softw. Eng.*, vol. SE-2, no. 4, pp. 308 – 320, Dec. 1976.

[37] S. A. Mengel and J. Ulans, "Using Verilog LOGISCOPE to analyze student programs," in *Frontiers in Education Conf., 1998. FIE '98. 28th Annual*, 1998, vol. 3, pp. 1213 –1218 vol.3.

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

[39] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proc. ACM-IEEE Int. Symp. on Empirical Software Eng. and Measurement*, New York, NY, USA, 2010, pp. 8:1–8:10.

[40] M. V. Mantyla, J. Vanhanen, and C. Lassenius, "Bad smells - humans as code critics," in *20th IEEE Int. Conf. on Software Maintenance. Proc.*, 2004, pp. 399 – 408.

[41] M. V. Mantyla, "An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement," in *Int.Symp. on Empirical Software Eng.*, 2005, p. 10 pp.

[42] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Int. Conf. on Software Maintenance, 2003. ICSM 2003. Proc.*, 2003, pp. 381 – 384.

[43] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance. Proc. 20th IEEE Int. Conf.*, 2004, pp. 350–359.

[44] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. on Softw. Eng.*, vol. 36, no. 1, pp. 20 – 36, Feb. 2010.

[45] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design Defects Detection and Correction by Example," in *Program Comprehension, IEEE 19th Int.Conf.*, 2011, pp. 81–90.

[46] M. J. Munro, "Product Metrics for Automatic Identification of," in *Software Metrics, 2005. 11th IEEE Int. Symp.*, 2005, pp. 15–15.

[47] J. Kreimer, "Adaptive Detection of Design Flaws," *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 4, pp. 117–136, Dec. 2005.

[48] J. Cunha, J. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," *ICCSA 2012*, pp. 202–216, 2012.

[49] "Looking Glass Community." [Online]. Available: https://lookingglass.wustl.edu/. [24-Feb-2013].

[50] J. F. Pane, B. A. Myers, and C. A. Ratanamahatana, "Studying the language and structure in non-programmers' solutions to programming problems," *Int. J. Hum.-Comput. Stud.*, vol. 54, no. 2, pp. 237–264, Feb. 2001.

[51] "Alice Community." [Online]. Available: http://www.alice.org/community/. [12-Mar-2013].

[52] The CSTA Standards Task Force, "CSTA K-12 Computer Science Standards," CSTA, 2011.

[53] B. G. Glaser and A. L. Strauss, *The discovery of grounded theory: Strategies for qualitative research*. Aldine de Gruyter, 1967.