2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated

Eliane S. Wiese
University of Utah

Anna N. Rafferty
Carleton College

Armando Fox
University of California, Berkeley

***Abstract*—Novices' functionally-correct code is often redundant, verbose, or un-idiomatic. Such code could indicate shallow understanding of the programming language, or unfamiliarity with experts' preferences for code structure. Understanding why novices write poorly is important for designing instruction and tools to help novices write elegantly. 231 novices judged style and readability for sets of code snippets targeting seven topics. Within each set, functionality was the same, but the writing followed either common novice patterns or a more elegant, "expert" pattern. Overall, 76% of novices thought the "expert" snippets had the best style, but only 64% said those snippets were most readable. However, comprehension was similar for both "expert" and novice patterns, regardless of readability preferences. This suggests that students who prefer novice patterns do not necessarily have deep misunderstandings about the programming language. One topic included a code-writing task, and students' readability preferences were predictive of their code-writing patterns, suggesting that readability preferences reflect writing choices rather than comprehension. Thus, novices may benefit from lightweight tools that identify common patterns and suggest an "expert" solution, while helping them see that the "expert" solution is more readable than they think.**

***Index Terms*—computer science education, code readability, novice code comprehension**

I. Introduction: Teaching Elegant Coding

Code must not only be functionally correct, but also readable to other humans. Readability is crucial in professional settings: the dominant cost incurred during the lifecycle of a long-lived software system is not bug fixing, but rather maintenance and enhancement of legacy code [7]. Further, in a case study, [13] found that when writing new code that would interact with existing code, 10% of the time was spent on writing and 90% on reading. Ideally, code should be *elegant*: it should be readable, concise, and use language constructs and control flows that are suited to the problem [3], [5].

For style issues like proper indentation and informative variable names, experts and novices may agree that improving style also improves readability [15]. However, novices and experts may disagree on the readability of control flows and built-in language features. If a novice is not fluent with these language features, the novice may not agree that incorporating them improves the code's readability. Determining how novices think about style and readability is crucial for teaching programming effectively and for designing tools to support novice programmers. If well-styled code does not improve readability for novices, instruction to encourage good style (e.g., [28]) may inadvertently lead to code that is less readable for the code's author. While students may write code with good style for a class, in the long term, students are unlikely to continue using good style if doing so makes their own code less readable to them. Programming style includes many facets (e.g., typographical conventions, variable names, efficiency). We focus on code structure and control flow, and our use of the word *style* in this paper refers to those facets.

For experts, readability is an important factor in determining if code has good style. However, novices may differ from experts in what they find easy to read and understand. Students are also likely to spend much less time than experts on reading, modifying, or extending existing code. Thus, the importance of good style for readability and maintenance may be less apparent to students. If novices do not understand the benefits of good style, and if good style makes it harder for them to read the code, then style is likely to seem opaque and arbitrary. Designs for instruction and tools to support students in writing well-styled code will depend on whether they find well-styled code to be more readable than poorly-styled code. If students understand well-styled code and find it readable, detecting errors and suggesting solutions may be sufficient. If not, students may need to be taught how the well-styled code works. Therefore, this paper investigates, empirically, how well novices' perceptions of style and readability match those of experts, and how novices perceptions of readability match actual code comprehension.

II. Prior Work: Metrics, Tools, Assumptions

Experts expect code to follow certain conventions, and deviating from that expected style can reduce readability for them [23]. For experts, there are several metrics for code readability and tools to assist in improving code structure. However, in drawing conclusions from the literature, we face three challenges: first, even among experts there is variability in the accuracy of readability metrics; second, tools to support experts in structural improvements support refactoring rather than the lower-level errors typical of novices; and third, the difference between how experts and novices read and perceive code likely influences their opinions of readability, so tools to promote readability for experts may not help novices.

With respect to the first challenge, various metrics exist for measuring readability, including LOC counting (lines-of-code), cyclomatic complexity [14], and Assignment–Branch–Condition (ABC) score [6]. However, empirical studies of experts suggest that code readability may be influenced by

978-1-7281-1000-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-SEET.2019.00017

neighboring code [10], and that (for example) not all instances of a construct (e.g. a branch) affect readability equally [1]. Accordingly, researchers are examining additional contextual features to develop more precise readability metrics [22].

Addressing the second challenge, *refactoring*, a process during which code is modified to improve its structure without changing its behavior, is closely tied to good code style. One type of refactoring involves identifying and remediating *code smells*—(anti-)patterns in source code that create unnecessary complexity and make code more difficult to comprehend and/or modify [9], [16]. Automatic detection of such smells [17], [19], [24] comes closer to our goal, but smells often occur only in considerably more complex code than novices are called upon to produce. Further, tools that aid in refactoring and in detecting code smells focus on pointing out problems that an expert might otherwise miss because the problem stretches across several different areas of code [17], [19], [24]. Poor style within a single function may be assumed to be purposeful, and therefore, not something that the author would want to change. One tool focuses on lower-level problems that occur within a single function: inadvertent duplication of code fragments [25]. However, the tool is specifically designed to avoid flagging a typical novice error: repeating code inside an `if` and its corresponding `else`. Even in cases where the entire body of the `if` and `else` were identical, the tool does not flag this as an error, under the assumption that this structure was purposefully written by the programmer to match their mental model of the program's execution [25].

The third challenge in drawing conclusions about novices from literature about experts is that, like in other domains, novices and experts perceive and process code differently. Experts recognize larger perceptual chunks than novices when reading code, and they focus more readily on the code's overall purpose [11]. Since novices have difficulty perceiving and understanding large code chunks, they may rely more heavily on line-level details to comprehend code, and may be more distracted by low-level differences [27]. Thus, empirical studies with novice programmers are necessary.

The existing literature on explicitly teaching better coding style to novices is sparse, as is the repertoire of systems designed to give actionable feedback to students on code structure. Some tools provide checks on "coding style" narrowly defined in terms of following typographical standards (e.g. placement of semicolons and braces) [4], [12], [21], and many tools, such as Eclipse, will auto-format code according to defined standards. However, typographical standards do not address code structure and use of built-in functions (i.e. idioms). FrenchPress begins to address some of novices' common, non-typographical errors, such as incorrectly using the modifier `public` in Object-Oriented coding, or comparing a boolean to `true` [2]. FrenchPress detects these problems and offers clear, simple solutions, such as to declare certain variables or methods as `private`, or to replace `condition == true` with `condition` [2]. Submitty [20] offers the potential for checking a much larger and more complex range of errors, but must be programmed by the instructor (e.g., in a few lines of code, an instructor could check if submissions contain the correct number of `if` statements or nested `for` loops). Feedback messages from Submitty are written by the instructor, and could include hints and suggestions, or simply alert the student to the error. An Eclipse plugin, PMD [8], alerts programmers to style issues including collapsible `if`-statements and dead code. AutoStyle [28] automatically analyzes a pre-existing corpus of submissions for a particular assignment and suggests that students use or not use particular idioms so that their assignments can become more similar to the best solutions. However, we have not found studies examining students' perspectives on code readability for code that follows or does not follow style suggestions. Since we don't know if implementing these style suggestions makes the code feel more or less readable to novices, we don't know if these systems provide the optimal support for long-term change in students' coding.

Empirically determining the impact of novice code patterns on novice readability is important, especially in light of the hypothesis that novices produces poorly-structured code because they do not fully understand the task the program is intended to accomplish or the language features best matched to that task [26]. This hypothesis is echoed in tools, such as Style Avatar, which is explicitly restricted to typographical issues because of the assumption that novices who made idiomatic or structural errors would not have the knowledge needed to fix them [12]. Therefore, our studies examine empirically the relationship between novice and expert code structures on readability for novices. Understanding these relationships will support the design of tools to help novices write elegant code.

III. Research Questions: Readability, Style, and Comprehension

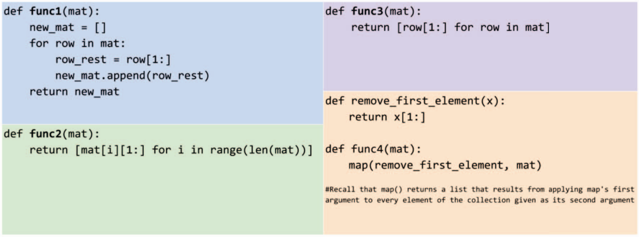
Tools intended to help novices improve their coding embody pedagogical perspectives on what novices are capable of understanding. This study aims to collect empirical evidence for how novices think about readability and style, as these concepts pertain to common novice coding patterns. Specifically, our three research questions are:

RQ1. Do novice programmers find common novice patterns to be more readable than expert patterns?

RQ2. Do novice programmers believe that what is most readable to them also matches experts' style preferences?

RQ3. Do novice programmers' stated preferences for readability match their actual code comprehension?

Here, *readability* is a self-reported opinion, while *comprehension* is objective, based on finding the output for a code sample with a given input. To investigate these questions we ran a pilot study and a follow-up assessment study with undergraduate computer science students. Both studies used surveys to present code with the same functionality but written with different structures or idioms. Both studies showed all styles of code to all participants, and were analyzed using within-subject measures.



```
def func1(mat):
    new_mat = []
    for row in mat:
        row_rest = row[1:]
        new_mat.append(row_rest)
    return new_mat
```

```
def func2(mat):
    return [mat[i][1:] for i in range(len(mat))]
```

```
def func3(mat):
    return [row[1:] for row in mat]
```

```
def remove_first_element(x):
    return x[1:]

def func4(mat):
    map(remove_first_element, mat)

#Recall that map() returns a list that results from applying map's first
argument to every element of the collection given as its second argument
```

Fig. 1. Sample readability/style question on list comprehensions in Python. Questions included the note that all code samples did the same thing, followed by the Style or Readability prompt. The Style prompt was: "How would a programming expert rank them from best to worst for style? Style is the tasteful use of language that makes code elegant, efficient, and revealing of design intent." The Readability prompt was: "Rank them from best to worst for readability: how easy it is for YOU to figure out what the code does?"

IV. Pilot Study: Built-In Language Features

We administered a self-paced, respond-anytime, online survey to 27 undergraduate computer science students at Institution C. 37% of these students (10/27) had previously completed a rigorous introductory Python programming course on which the survey questions were based, and an additional 59% (16/27) were enrolled in that course during the study.

The survey asked students to self-report their familiarity with specific idioms (list comprehensions and sets) in the Python language. It then presented short code examples (5–15 lines) designed to solve problems similar to those encountered in the course assignments. In the *code comprehension* questions, students were shown functional code and were asked what the output would be, given some input. In the *style and readability* questions, students were shown four code blocks with the same functionality, and were asked to rank them first in order of style and then in order of readability (see figure 1 for prompts).

A domain expert (an instructor at Institution C) had previously classified each correct implementation as Optimal (most-stylistic solution), Naïve (avoids stylistically-appropriate idioms in favor of longer code using only primitive language features), Suboptimal (includes appropriate idiom(s) but *misuses* them in a way that makes the solution needlessly complex), or Helper Function (structurally identical to either the Optimal or Suboptimal solution, but references a helper function that duplicates the functionality of the built-in language feature an expert would have used). Each target idiom had one set of code blocks for the style/readability ranking, and one or two comprehension questions with code implemented in both the Optimal and Naïve styles. Key findings included:

1. For each readability question, a large minority of students (40%) did not find the Optimal code to be the most readable. Across both questions together, only 11 students (41%) ranked both Optimal solutions as most readable. If we assume that our domain expert's choices of "most readable" are representative of other Python experts, this finding suggests that most students do not consistently find "expert style" to be the most readable implementation of a piece of code.
2. Students were more likely to be correct on comprehension questions for Naïve code compared to Optimal code (on average, 89% and 66%, respectively). Even students who ranked the Optimal solutions as most readable were more accurate with Naïve code compared to Optimal code.

These findings suggest that novices may prefer to read non-expert code, and that expert code may reduce their comprehension. Even novices who claim to prefer expert language features may be overestimating their own understanding. These results suggest that tools for supporting elegant code among novices may first need to teach novices what that code does. Yet, a majority of novices did understand the expert code, suggesting that they possess sufficient comprehension skills to be receptive to instruction on advanced idioms.

Results from the pilot were not conclusive because of several threats to validity (beyond the small sample size, likely a result of running the study near final exams). We only had one expert's opinions on which code exhibited best style. Also, the target idioms were not emphasized equally across the course sessions from which we had recruited participants, so all participants may not have had the same level of exposure to them. This was evidenced by some students indicating that they did not have experience using the target idioms. Further, while comprehension and style/readability questions were matched by topic, the topics were likely too broad. For example, the style/readability questions for `set` included the data structure but no built-in functions; some comprehension questions included a particular syntax for a built-in `set` function that many students found confusing (`&` for intersection). Finally, the pilot targeted specific Python idioms that may not be relevant to other languages. The follow-up study continues to examine the relationship between style, readability, and comprehension, with a study design that responds to these threats.

V. Study: Novice vs. Expert Patterns

This follow-up study was designed to further explore the readability of expert code for novice programmers. This follow-up study was also designed to address shortcomings in the pilot. In contrast to the pilot which examined particular idioms that novices may not be familiar with, this study examined coding patterns with foundational elements such as returns, if-statements, and loops, which all participants were expected to understand. Additionally, we worked closely with the course instructor to examine novice patterns thought to be prevalent and important, and we verified which code blocks exhibited the best style with five other CS instructors. Finally, by running the study earlier in the semester, we recruited more students, ending up with data from 231 participants in our analyses.

Based on the preliminary results from the pilot, we pre-registered our hypotheses and analysis plan with the Open Science Framework (https://osf.io/b53zv/register/564d31db8c5e4a7c9694b2be). We hypothesized that:

H1. Prevalence: At least 20% of the population of students will: choose non-expert code as most readable (for all topics); choose at least 2 non-expert code blocks as most readable (across all topics); and use a novice pattern on the code-writing task.

H2. Style vs. Readability: For each topic, students will be more likely to choose the expert code as being the best styled rather than being the most readable.

H3. Comprehension: Overall, students will be more accurate at comprehension questions for code written with novice patterns. Within each topic, this effect will be more pronounced for students who selected the novice code as most readable.

A. Assessment Design

We identified topics to investigate by examining final homework submissions from past students in their second programming intensive course. We chose second courses because we wanted to examine patterns that persisted beyond students' first encounters with the target concepts. One of the authors teaches a second programming class in Java at Institution A, and randomly selected 20 submissions from the past four offerings of the course (from a total of 81 submissions; each submission was completed by 1-2 students). Hand-inspection revealed eight sub-optimal patterns in those 20 submissions. Another author examined submissions from the previous year's second programming class at Institution B (also taught in Java). All 316 submissions from that assignment (completed individually) were searched for the presence of `while` loops since `for` loops were most appropriate for those tasks. Since the assignment at Institution B targeted loops while the assignment at Institution A did not, this search revealed an additional novice pattern. Hand-inspection of several assignments at Institution B revealed that many patterns identified at Institution A were also present at Institution B. The current instructor of the second- and third-semester courses at Institution B (from which we recruited participants) reviewed the patterns and ordered them by importance for his courses. The instructor verified that for these courses, there were no other topics which he thought were more important for writing well-structured code within a single function.

An undergraduate student teaching assistant completed a draft version of the survey to check that the instructions made sense and to see how long the survey might take our target students to finish. The student indicated that the instructions were clear, but some questions with long code examples were time-consuming to answer. The draft survey took longer than our target time of 1 hour, so we simplified the longer code examples and eliminated two novice patterns from the final survey: explicitly checking within an `if` statement if something is `true`, and unnecessarily assigning a value to an intermediate variable instead of assigning the value directly to the ultimate target variable. The final seven topics in our study had these novice patterns:

1. For exclusive cases, writing a series of `if` statements rather than using `else if`.
2. Conjoining conditions using nested `if` statements rather than the `&&` operator.
3. Repeating code inside an `if` block and its corresponding `else` block.
4. Including extraneous cases, with a general solution that covers those special cases.
5. Returning a boolean value by checking a condition with an operator (e.g., `==, >, <`) by using an `if` statement and explicitly returning `true` or `false` rather than simply returning the condition.
6. The previous pattern for conditions with method calls instead of operators.
7. Using a `while` loop when a `for` loop is more appropriate.

The first four were identified by the course instructor as the most important of the original set.

1) Prevalence of Target Patterns in Students' Code: Several patterns were identified in prior years' homework submissions with automated checks using regular expressions. Comments were either removed before the checking was done, or results were hand-inspected to ensure that matches within comments were not included in the final count.

Automated checking for the boolean return patterns found them in 75% (61/81) of the submissions from Institution A and 13% (42/316) of the submissions from Institution B. The automated check searched for returning `true` or `false` if the condition was true, and returning the opposite value if the condition was false. The patterns optionally allowed for brackets enclosing the body of the `if` and/or `else`; and optionally allowed for an explicit `else` statement. Automated checking for `while` loops found them in 5% (17/316) of the submissions from Institution B, which targeted iterating through arrays (the assignment from Institution A did not target loops). Though using `while` instead of `for` was not widespread, for some students it appeared to be pernicious: of the students who used `while` loops, 4 students used them exclusively. The remaining 13 also used `for` loops (sometimes nested in combination with `while`). Using nested `if` statements rather than `&&` was found in 5% (4/81) of the submissions from Institution A, and 3% (9/316) from B. Nested `if` statements were identified with regular expressions, and then hand-inspected to verify that using `&&` would have been appropriate.

The remaining patterns were identified by hand inspection of the original 20 randomly-selected submissions from Institution A (completed by 1-2 students), and of 30 randomly-selected submissions from Institution B (completed individually). The *Extraneous Cases* pattern was found in 7/20 submissions from Institution A and 1/30 from B. *Repeated `if` Statements for Exclusive Cases* was found in 4/20 assignments at Institution A, and 0/30 at B. *Repeating Code Within an `if` and its `else`* occurred in 11/20 submissions at Institution A and 10/30 at B.

Overall, these issues as a set were prevalent in student code. Of the 20 random assignments from Institution A (where the patterns were first identified), 19 contained at least one of the first six novice patterns, as did 13 of the 30 random assignments from Institution B. The specific tasks in the different assignments account for some of the differences in errors and error rates. Our goal in this analysis is not to estimate the overall rates of these errors among novices, but simply to demonstrate that evidence suggests they are prevalent enough to warrant examination.

2) Instruction on the Target Patterns: The instructor for the first-semester programming course at Institution B (also taught in Java) verified that students would have received instruction on the expert patters for six of the seven topics on the survey. One assignment in that first-semester course explicitly targets returning boolean values, and requires students to do so without using `if` statements. Other patterns are addressed in live code demonstrates during lectures. Students are also explicitly told to use `for` loops rather than `while` loops when iterating through arrays; to avoid duplicating code (which pertains to repeating code within an `if` and its `else`); and to use conjunctions rather than nested `if` statements. The one pattern that is not addressed in that course is *Extraneous Cases*.

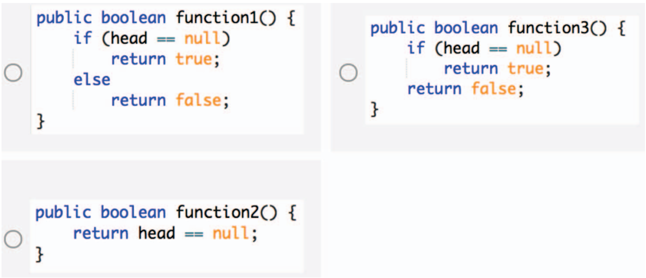
The instructor for the second-semester programming course at Institution B noted that no assignments or lectures in that course specifically target these patterns. Students would lose points on assignments for duplicate code, which somewhat addresses patterns 3 and 5. Teaching Assistants may also deduct points for the other patterns under a rubric category "general code quality," but the instructor also noted that this category is used subjectively, and that it is too time-consuming to hand-inspect code for all novice patterns. Therefore, while students may be penalized for using these patterns in their second semester and beyond, the high cost of identifying these patterns prevents consistent detection and feedback.

3) Survey Introduction and Code Writing: All code on the survey was in Java, the language taught in the first two programming courses at Institution B. The survey first presented a short list of instructions (e.g, to complete the survey individually without outside resources, to note that answers could not be changed after moving on to the next question). Then the survey presented the consent document and asked participants if they wished to participate in the research. Next, the survey included one code writing task, with the prompt "Fill in the function so that it returns true if the input is 7, and false otherwise (the first line and last line are provided for you)." The first line of the function was "`public boolean func(int num)`{", and the last line was the closing bracket.

4) Readability and Style Questions: Next, the survey presented one pair of readability and style questions for each topic. These questions showed three code samples with the same functionality (see figures 2 and 3). One code sample demonstrated a common novice pattern, and another demonstrated a more expert pattern. The third code sample either presented another version of the novice pattern (for the *Returning Booleans* topics, figure 2, and for *Repeating Code within an `if` and its `else`*), another version of the expert pattern (for *Extraneous Cases*, figure 3, and for *A Series of `if` statements for Exclusive Cases*), or a mix of the novice and expert patterns (with nested loops for *`for` vs. `while` Loops*, and for *Conjoining Conditions with Nested `if` Statements*). For all readability and style questions, the readability question was presented first, and then the style question. Readability questions asked what was most readable for the student, and style questions asked which they thought an expert would pick as having the best style (see figure 2 for prompts). The style questions additionally gave students the option to select that an expert would say all the code samples had equal style.

5) Comprehension Questions: After the readability/style questions, the survey presented 14 comprehension questions (two per topic), one with code written with the common novice pattern, and one with matched code written with a more expert structure. Each comprehension question asked what the output of the code would be for two or three different inputs, with the code sample shown after the specific questions. To have the same level of difficulty between the matched questions while also encouraging students to read both code samples, matched questions either varied the inputs while keeping the functionality of the code the same (2 topics, see figure 6), or kept the inputs the same while slightly changing the functionality of the code (2 topics, e.g., checking if the last character in a string is 'e' or 'a'), or did both (3 topics, see figures 4 and 5). All comprehension questions were multiple-choice, and all included the options "the code will not compile" and "the code will throw a runtime exception."

All code samples on the survey which demonstrated novice patterns were closely based on real novice code submitted as homework at either Institution A or B. All code samples were between 3 and 18 lines. Within each topic, variable names and typographic conventions (e.g., whitespace and placement of brackets) was kept consistent. Novice and Expert patterns on the comprehension questions were tightly matched to those on the readability/style questions.

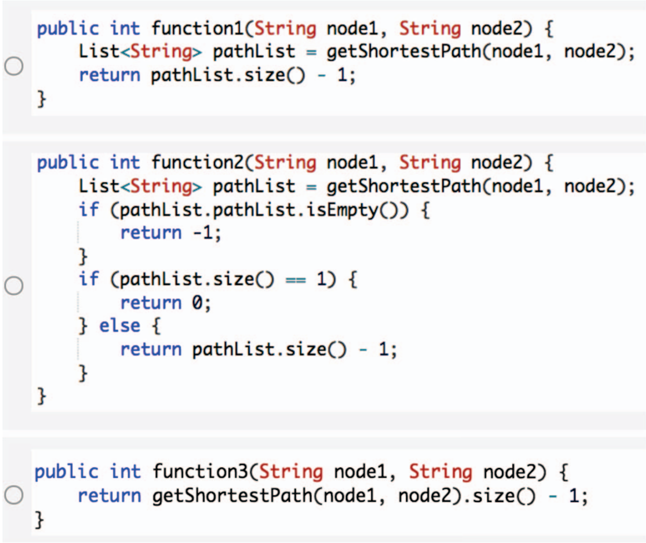


○
```
public boolean function1() {
    if (head == null)
        return true;
    else
        return false;
}
```

○
```
public boolean function3() {
    if (head == null)
        return true;
    return false;
}
```

○
```
public boolean function2() {
    return head == null;
}
```

Fig. 2. Sample readability/style question: returning a boolean value with an operator (==). The Readability prompt was: "All of these code samples do the same thing. **Which one is most readable to you**: which one makes it easiest for YOU to figure out what the code does?" The Style question immediately followed the Readability question, with this prompt: "These are the same code samples as the previous question. All of these code samples do the same thing. **Which one would an expert say has the best style?** Style is the tasteful use of language that makes code elegant, efficient, and revealing of design intent." The Style questions all had one additional option, "An expert would say they all have equal style".

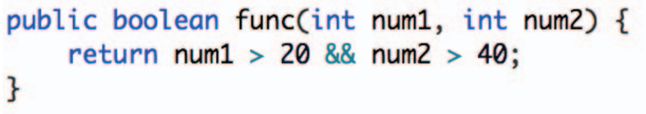


```
○ public int function1(String node1, String node2) {
      List<String> pathList = getShortestPath(node1, node2);
      return pathList.size() - 1;
  }
```

```
  public int function2(String node1, String node2) {
      List<String> pathList = getShortestPath(node1, node2);
      if (pathList.pathList.isEmpty()) {
          return -1;
      }
○     if (pathList.size() == 1) {
          return 0;
      } else {
          return pathList.size() - 1;
      }
  }
```

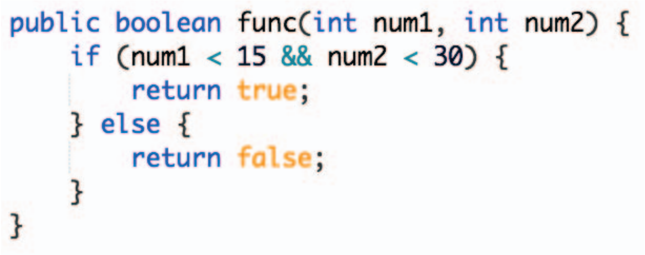
```
  public int function3(String node1, String node2) {
○     return getShortestPath(node1, node2).size() - 1;
  }
```

Fig. 3. Sample readability/style question: splitting a task into special cases while also including a general solution.



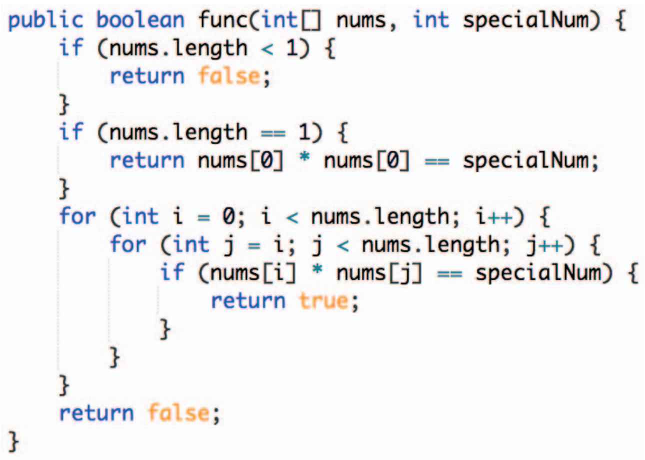
```
public boolean func(int num1, int num2) {
    return num1 > 20 && num2 > 40;
}
```

Fig. 4. Expert code for two comprehension questions on returning boolean values with operators. Prompts asked "For the code below, what will be the output for this input:" with inputs 28, 81, and 4, 60. Options for each question were: true, false, the code will not compile, the code will throw a runtime exception.



```
public boolean func(int num1, int num2) {
    if (num1 < 15 && num2 < 30) {
        return true;
    } else {
        return false;
    }
}
```

Fig. 5. Novice code for two comprehension questions on returning boolean values with operators. Prompts asked "For the code below, what will be the output for this input:" with inputs 19, 74, and 12, 8. Options for each question were: true, false, the code will not compile, the code will throw a runtime exception.



```
public boolean func(int[] nums, int specialNum) {
    if (nums.length < 1) {
        return false;
    }
    if (nums.length == 1) {
        return nums[0] * nums[0] == specialNum;
    }
    for (int i = 0; i < nums.length; i++) {
        for (int j = i; j < nums.length; j++) {
            if (nums[i] * nums[j] == specialNum) {
                return true;
            }
        }
    }
    return false;
}
```

Fig. 6. Novice code for three comprehension questions: this code splits out two special cases with the `if` statements, but these special cases are already covered by the general solution at the end. Prompts asked "For the code below, what will be the output for this input:" with inputs [],16 and [4], 16 and [1, 2, 8], 16. Options for each question were: true, false, the code will not compile, the code will throw a runtime exception. The expert version of this code was the same, except the two leading `if` blocks were not included. Inputs for the expert version were [], 36 and [6], 36 and [3, 12, 19], 36.

6) Question Order: Within the readability/style section, topics were ordered so that the boolean return questions would be first and last, with the other topics ordered randomly between them. The survey was set up so that half of the participants would be randomly assigned to see the topics of the readability/style questions in one order, and the other half would see them in the reverse order. For the comprehension section, questions were ordered randomly, but blocked such that each topic would appear once within the first seven questions, and once within the last seven questions. The survey was set up so that half of the participants would be randomly assigned to see the comprehension questions in one order, and the other half would see them in the reverse order. The random assignment to question order was independent for each section. Random assignment for question order was done to mitigate ordering effects, especially if seeing one version of code in the comprehension section made it easier for students to answer the matched question later on.

7) Five CS Instructors' Review of Questions: The course instructor reviewed the survey questions and confirmed that the proposed "more expert" code samples were preferable to the ones demonstrating the novice patterns. Five other instructors who taught programming-intensive courses (two from Institution A and three from Institution B) also reviewed the code samples. The instructors saw a modified version of the survey that included the same style questions that were on the regular survey, but not the code writing, readability, or comprehension questions. The instructors also saw seven additional style questions asking them to compare the style of the matched code samples from the comprehension questions. In total, the five instructors saw 14 questions, two for each topic, which asked them to select the best-styled code from sets of two or three code blocks which had the same or similar functionality. The instructors could also say that the code blocks had equal style. For four topics, all five instructors were unanimous in their choices for which code block exhibited the best style.

For *Extraneous Cases*, instructors were unanimous in selecting code samples that only included the general solution. However, for the readability/style question, the instructors were split between two of the three code samples which both followed this expert pattern (the top and bottom code samples in figure 3). Therefore, we consider both of these code samples to be "expert."

For *Repeating Code within an `if` and its `else`*, all five instructors were unanimous on the code samples from the comprehension questions: the novice pattern included repeated code, while the more expert pattern eliminated the repeated code and, in doing so, eliminated the entire `else` block. However, only three instructors chose the similarly-styled code from the readability/style set. One instructor thought the novice pattern was best, and one thought that all the options were equal. We followed the majority opinion in determining which was the "expert" code.

For *A Series of `if` statements for Exclusive Cases*, all code samples in the readability/style set included four conditional statements, and the first `if` statement included a `return`. The novice pattern used `if` statements for all four conditionals, one sample used an `if` statement following the `return` and the rest `else if` statements, and the last sample only used `else if` statements following the initial `if`. Instructors were split between the last two samples, with four instructors indicating that `else if` was not necessary if the preceding `if` statement included a `return`, and one indicating that it was. Therefore, for the readability/style question, we consider both code samples to be "expert." For the comprehension questions for that topic, there were only two conditionals and neither included a `return`. Three instructors preferred the `else if` version, one preferred two `if` statements, and one thought both were equal.

We were surprised by the disagreement among instructors for the latter two topics, since some instructors preferred the novice pattern in some cases. The disagreement suggests that the severity of a novice pattern may be context-dependent. We discuss this further in the Threats to Validity section.

B. Participants and Method

We recruited participants from Institution B who were taking the second or third semester course of the programming-intensive sequence. Students could not take those two courses concurrently. Instructors at Institution B explained that for all of the topics on the survey, students would have been taught to avoid the novice patterns in the first course of the programming-intensive sequence. Students would have learned the expert patterns from examples and from lectures, and would get feedback and/or point deductions if TAs or instructors saw the novice patterns in their homework. However, instructors also noted that detecting these patterns was time-consuming and could not be done thoroughly for all assignments. Further, some students persisted in using the novice patterns in their second and third semester of programming courses.

The instructor of the two courses offered students extra credit for completing the survey (available at https://tinyurl.com/RICE-Survey-pdf) At the beginning of the online survey, students were invited to participate in the research, but they could still complete the survey for extra credit if they declined. Approximately 500 students are enrolled across the two courses. We include data from 231 students in our analyses: these students consented to participate in the research, were 18 or over, and answered all readability/style questions and at least 80% of the comprehension questions by the deadline announced for receiving extra credit. Of these 231 students, 75 were enrolled in the second-semester course and 156 were enrolled in the third-semester course.

C. Results

Across all topics, students identified the expert code as the most readable in 65% of responses. Agreement varied widely by topic, from 24% of students agreeing that code should not be repeated within an `if` and its `else`, to 90% agreeing that using `&&` was more readable than nested `if` statements (Table I).

1) H1, Prevalence - Partially Supported: Evidence from our sample of students indicates that for the first four patterns in Table I, at least 20% of the population of students would find a non-expert code snippet to be most readable (z-tests, with Bonferroni-corrected $p < .01$, Table I). For three of these topics, a *majority* of students chose one of the novice code snippets as more readable than the expert code. 74% of students found a non-expert code style most readable for at least two of the seven topics (z-test for a population mean of $\geq$20% is significant, $p < .0001$), and students chose an average of 2.49 non-expert code blocks as most readable, supporting our hypothesis that novice programmers would choose at least two non-expert solutions as most readable.

Students in the third-semester class were more likely to select the expert code (for both best style and most readable). A logistic regression with fixed effects for question type (readability, style), and random effects for student, on selection of expert code, found a significant effect for course ($t(3231) = 3.37$, $p < .001$). Overall, 57% of responses from second-semester students chose expert code as most readable, compared with 68% for third-semester students. Considering only third-semester students, z-tests for population proportions for three of the first four topics in Table I are still significant at a Bonferroni-corrected $p < .01$ for at least 20% of students (all $p < .0001$ for both Boolean Return topics and *Repeating Code within an `if` and `else`*).

On the code writing question, 59% of students wrote correct code that used an `if` statement to check the condition and explicitly return `true`, compared to 34% of students who gave the correct expert pattern answer of returning the boolean condition (7% of students wrote non-functional code). 11 of the 16 students who wrote non-functional code used an `if` statement incorrectly, including returning `true` when the the condition was true but not having a return statement for when the condition was false. Overall, 64% of students used an `if` (148/231), and, of students who wrote functional code, 64% (137/215) used `if`. Both ways of measuring usage of `if`

TABLE I
SURVEY RESULTS: SELECTION OF EXPERT CODE AS MOST READABLE AND BEST STYLED, AND COMPREHENSION ACCURACY

Topic	Agreed Expert Code Was:			z-test	McNemar		Comprehension	
	Most Readable	Best Styled	Both				Good Style	Poor Style
Boolean Returns with Operator	39%	71%	36%	$p < .0001$*	$p < .0001$*	$\chi^2 = 62$	91%	93%
Boolean Returns without Operator	58%	86%	52%	$p < .0001$*	$p < .0001$*	$\chi^2 = 47$	95%	94%
Special Cases with General Solution	71%	94%	70%	$p = .0008$*	$p < .0001$*	$\chi^2 = 50$	72%	81%
Repeating Code within `if` and `else`	24%	25%	10%	$p < .0001$*	$p = .80$	$\chi^2 = 0.1$	86%	86%
`For` vs. `While` loop	84%	81%	74%	$p = .92$	$p = .31$	$\chi^2 = 1.4$	67%	76%
`if` statements vs. `else if` (exclusive cases)	86%	92%	82%	$p = .99$	$p = .020$	$\chi^2 = 6.1$	92%	86%
Nested `if` statements vs. `&&`	90%	89%	84%	$p > .99$	$p = .86$	$\chi^2 = 0.2$	75%	77%

*Bonferroni-corrected $p < .01$ (that is, the raw p-value $< .0014$). p-values given above are raw.
z-tests examined if the population proportion was $\geq$ 20% for students thinking that novice code was more readable than expert code.
McNemar tests examined if students' agreement with expert choices were different for style vs. readability.

support Hypothesis 1 for code writing with novice patterns (both z-tests for a population mean of $\geq$20% are significant, $p < .0001$). The third-semester students were more likely to use expert style than the second-semester students (42% and 16%, respectively), and were less likely to use `if` (56% and 81%, respectively). However, repeating the corresponding z-tests with the third-semester students alone yields the same results (both z-tests for a population mean of $\geq$20% are significant, $p < .0001$).

The pattern of code that students identified as most readable for *Returning Boolean Values with an Operator* was predictive of whether or not they wrote expert style code on the matched writing task (logistic regression; main effect for identifying the expert pattern as most readable: $t(212) = 7.28$, $p < .001$): 68% of students who marked expert code as most readable for this topic actually produced code that matched the expert pattern, and 85% of those who marked the novice code as most readable for this topic produced code that matched the novice pattern. Thus, there was high agreement between students' readability preferences and style of coding.

2) H2, Differences in Readability vs. Style - Supported: When asked which code snippets had better style, students identified an expert code snippet in 75% of responses. Using logistic regression, we found that students were significantly more likely to identify expert code as best styled than to identify it as more readable (model also included a random effect for student; effect for style versus readability: $t(3232) = 8.07$, $p < .001$). This effect varied by topic: McNemar tests comparing agreement with expert choices for style vs. readability were significant for three topics, with more students selecting the expert choice for style than for readability for: *Returning Boolean Values with Operators, Returning Boolean Values without Operators,* and *Extraneous Cases* (all Bonferroni-corrected $p < .01$, see Table I for test statistics).

Regardless of their agreement with expert choices, students frequently selected different code blocks as being most readable versus best styled. Only 39% of students made the same choice across the readability and style questions for *Repeating Code within an `if` and its `else`*, while 50–75% made the same choices across both questions for the two Returning Booleans topics, *A Series of `if` statements for Exclusive Cases*, and *Extraneous Cases*. Students made the same choice for style and readability more than 80% of the time for each of the final two topics (*Conjoining Conditions with Nested `if` Statements*, and *`for` vs. `while` Loops*). Overall, these results demonstrate that students do not always consider their own sense of readability to be indicative of what an expert would call well-styled. We consider the implications of this disconnect in the discussion.

3) H3, Comprehension - Not Supported: Although many students selected non-expert code as more readable than the expert code, we did not find evidence that accuracy on the comprehension questions was affected by whether the code used an expert or non-expert pattern (logistic regession, with random effects for question and student, and fixed effects for non-expert style, question topic, and interaction between non-expert style and topics; $t(7287) = 0.379$, $p = .70$ for non-expert style). There was also no evidence that students were more accurate on the type of code snippet for each topic that they had selected as most readable (logistic regression on comprehension scores, with random effects for question and student; fixed effect of matching one's choice of most readable: $t(7299) = 0.212$, $p = .83$). These results do not support hypothesis 3. We did not separate students by course because course was not significant in a logistic regression on comprehension scores ($t(7286) = 1.281$, $p = .20$).

In an exploratory analysis, we compared style on the code-writing task to comprehension of the matched, expert-styled code for *Returning Booleans with Operators*, restricting the analysis to students who had written functional code. A chi-square test on scores (0-2) and writing style (optimal vs. using an `if`) was significant ($\chi^2 = 6.3$, $p = .042$). However, comprehension was high for both groups, with average scores of 1.89/2 for students with optimal writing style, and 1.80/2 for students who used `if` (92% and 82% of each group, respectively, were correct on both comprehension questions).

Accuracy overall on the comprehension questions was high (Table I). Overall accuracy was above 85% for three of the seven topics, and only one topic (*`for` vs. `while` Loops*) had less than 70% accuracy for one of the code blocks. Accuracy was especially high for all code blocks for the two topics on Returning Boolean Values ($> 90\%$), which is notable because many students selected the novice code as more readable for those topics. We note that three code blocks in the comprehension questions inadvertently had missing semicolons that would have prevented the code from compiling (the `while` loop block and both blocks for *Repeating Content within an `if` and `else`*). Most students did not notice these errors, with a maximum of 20 students per question answering that the code would not compile. We removed from our corresponding means and analyses all such students.

To attempt to assess whether students might be having more difficulty comprehending some of the code despite choosing the correct answer, we examined the amount of time students spent on the questions with the expert code patterns versus the non-expert code patterns. Comprehension questions asked what the output would be for 2-3 inputs for a particular code bock. Time spent was aggregated across all inputs involving a particular topic and code block. Since the survey was self-paced, students could take breaks if they wished. Therefore, as pre-specified in the registered analysis plan, only students who answered each question were included; times that exceeded 20 minutes were removed; of the remaining data, times that exceeded three standard deviations above the mean time for a single code block were also removed. A repeated measures analysis of variance predicting time spent, with a random factor for student, did not find a main effect of the expert versus non-expert code patterns ($F(1, 3058) = 0.414$, $p = .52$), although there was a main effect of question topic ($F(6, 3058) = 258$, $p < .001$) and an interaction between question topic and code pattern type ($F(6, 3058) = 9.31$, $p < .001$). We note that while the timing data was not normally distributed, a repeated measures ANOVA is still appropriate, under the assumptions of the Central Limit Theorem, given our large sample size. The lack of a significant main effect indicates that any differences in timing were not in a consistent direction. However, the significant interaction indicates that there were differences in time spent on novice vs. expert code for some topics. Logistic regressions found significant differences in timing for two topics (using $\alpha = .007$ as the Bonferroni-corrected .05). For *Returning Boolean Values with Operators*, students responded more quickly to expert code than to non-expert code ($29s$ vs. $37s$; $t(444) = 2.82$, $p = .005$). In that case, the expert code was shorter. However, for *Repeating Code within an `if` and `else`*, students responded more quickly to the novice code, which was longer ($91s$ vs. $116s$; $t(430) = 3.86$, $p = .0001$). Overall, the timing data do not provide support for the hypothesis that students would more easily comprehend the non-expert code patterns.

D. Threats to Validity

While our study attempts to identify connections between novices' readability preferences, their comprehension, and their ability to identify expert-style code, there are several reasons why this experiment may not have fully assessed these connections or why the results may not generalize outside of the experimental context. First, the survey context may have impacted students' responses. Extra credit was given for completing the survey regardless of whether the answers were correct, potentially leading students to spend less effort than they would in a higher-stakes context. While the high rates of accuracy on the comprehension questions suggest some reasonable level of student engagement, lack of effort could manifest differently in the readability and style questions as opposed to the comprehension questions. Additionally, the existence of two separate questions, one about readability and one about style, addressing the same code, may have lead students to believe they should give different responses to each question. This would lead to an overestimate of students' true beliefs about the disparities between style and readability.

Our survey focused on code snippets, rather than full programs, which may also lead to lack of generalization outside of the survey context. We used short code examples to examine many topics without over-burdening students' time. However, the classification of something as exhibiting the best style or being the most readable may be dependent on the broader design of the program, meaning that choices based on a snippet do not fully capture the novice-expert pattern distinction. We also saw disagreements among the eperts for some of the topics. This could mean that for simple code blocks, these distinctions are not as relevant or salient, or that in the specific options we gave, there were multiple choices that exhibited expert-like features.

The other significant threats to validity center on our comprehension measures and the limitations of the data we collected. We collected code writing data for only one topic, and thus the results may not generalize to other topics. Our comprehension questions asked only to give the output of code based on a specific input, which ignores a number of other measures of comprehension. Specifically, we did not assess any overall understanding of the code, nor did we measure ability to successfully modify or debug code that followed an expert versus a novice pattern. The topics that we focused on were also limited to code within a single function, and we acknowledge that other structural issues are also important, such as how to break a problem into individual functions and object-oriented design patterns. Finally, for some comprehension questions, accuracy was near ceiling. It is possible that the impact of readability on comprehension is only noticeable when the code is more difficult to understand. Examining response time data was intended to address this issue to some extent, but does not address possibilities like lower readability only interfering with comprehension in contexts with higher working memory load.

VI. Discussion

Results partially support the Prevalence hypothesis (H1), indicating that at least 20% of second- and third-semester students do not find expert structures to be most readable for four of the topics on the survey. This suggests a problem for instruction that aims to improve the elegance of novices' code:

doing so may cause students to write in a way that they find less readable. However, results also support the Readability vs. Style hypothesis (H2), indicating that students are more likely to agree that expert code is best styled, compared to most readable. Further, choosing expert code as best styled was common (over 80% for five of the seven topics). This suggests that even when students perceive expert code as less readable, they may still be inclined to learn it because they believe it is preferable to others. Further, the lack of support for the Comprehension hypothesis (H3) and the high scores for comprehension of expert code (over 80% for five of the seven topics) indicates that there is no reason to assume that students will have difficulty comprehending expert code just because they think it is less readable.

These findings challenge the assumption, implicit in prior work (e.g., [25], [26]), that code preferences are tightly coupled to comprehension. We note that [25], [26] dealt with preferences in writing code, not reading code, and further that our exploratory analysis suggests that code-writing patterns are predictive of code comprehension. However, readability preference was predictive of writing style on the matched task for one topic, suggesting that readability preferences on other topics may likewise be correlated with writing preferences. Further, we found that while code-writing patterns are predictive of code comprehension, the two are not tightly coupled: 81% of students who wrote correct but poorly-styled code were correct on both comprehension questions for the matched task with code written in expert style. While students who write poorly may be less likely to understand expert code, we should not assume that they can't. The act of evaluating one's own work activates knowledge that may not have been available when the work was done [18]. Therefore, novices may start out by writing code that reflects their mental model in that moment, but may be willing and able to revise it, and may learn through that process. Future work should continue to explore the links between novices' code writing, code comprehension, code revision, and readability preferences.

While both the pilot and the follow-up study revealed that many students do not find expert code to be most readable, this effect was more pronounced in the pilot. Further, while students in the pilot seemed to overestimate their comprehension of expert code, students in the follow-up seemed to underestimate it (in comparisons of readability choices and comprehension scores). The follow-up study was designed to be confirmatory, unlike the pilot study, so we give greater weight to those results. However, we caution that our findings on novice patterns may not generalize to novice/expert differences in usage and comfort with built-in functions.

A. Implications for Instructional Design

Students in the third-semester course were more likely to select expert code as most readable and best styled, compared to students in the second-semester course. This indicates that, with normal instruction, students are learning to recognize and feel comfortable with expert patterns. However, many students will still find expert patterns to be less readable than novice patterns. Helping all students feel comfortable with expert patterns, and helping them use those patterns, will likely require targeted interventions.

One barrier to promoting good style is the difficulty of detecting poor style. Without automated detection, students in large courses cannot get consistent feedback, and students are unlikely to improve their style unless their grades depend on it. While existing tools like Submitty [20] and PMD [8] can detect some of these patterns, they are not in widespread use in programming courses, perhaps because of the set-up and customization required from the instructor. Therefore, there is a need for easy-to-use tools that detect common novice patterns.

B. Implications for Programming Environment Design

The novice patterns presented here are likely to be familiar to many instructors of programming classes. While static analysis tools exist to detect some of these patterns, others are not currently detected, and some, such as *extraneous cases* may require dynamic analysis. Our findings suggest that many students who make these errors would be able to understand the expert patterns. Therefore, it would be worthwhile to try lightweight solutions that detect novice patterns and suggest an expert pattern instead.

C. Conclusion

This paper presented a within-subject experiment that assessed 231 novices' perceptions of code readability and style, and their ability to comprehend code written with expert and novice patterns. This work codifies topics where students are likely to use or prefer novice patterns. While this paper examined Java, these topics are relevant across programming languages. Further, this work, like [28], shows that coding style is multi-faceted, and is composed of several skills. Therefore, students may have an intermediate level of style knowledge (e.g., a student may recognize good style without producing it).

Our results indicate that, for four topics (*Returning Boolean Values with and without Operators, Extraneous Cases*, and *Repeating Code within an `if` and its `else`*), at least 20% of novice programmers in their second or third semester of programming-intensive courses will find the expert pattern to be less readable than the novice pattern. Still, many novices will recognize which patterns are preferred by experts. And finally, for these seven patterns, comprehension was generally high and did not differ by expert/novice pattern, or by students' stated preferences for readability. Two reasonable hypotheses for why students use poor style are (1) they don't know what good style is and (2) they can't understand code that is written with good style. Our results do not support either of these hypotheses. Therefore, for these novice patterns, many students may benefit from tools which detect the patterns and offer suggestions. While some students may need support to understand the expert patterns, many students do not, even those who perceive expert patterns to be less readable.

References

[1] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. Syntax, predicates, idioms what really affects code complexity? *Empirical Software Engineering*, pages 1–42, 2018.

[2] Hannah Blau and J. Eliot B. Moss. Frenchpress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 15–20, New York, NY, USA, 2015. ACM.

[3] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd Edition).* Addison-Wesley Professional, 2007.

[4] Filippo Corbo, Concettina Del Grosso, and Massimiliano Di Penta. Smart formatter: Learning coding style from existing source code. In *IEEE International Conference on Software Maintenance, ICSM*, pages 525–526, 2007.

[5] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[6] Jerry Fitzpatrick. *More C++ Gems*, volume 17 of *SIGS Reference Library*, chapter Applying the ABC metric to C, C++, and Java, pages 245–264. Cambridge University Press, 2000.

[7] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002.

[8] Philip Graph. eclipse-pmd. https://marketplace.eclipse.org/content/eclipse-pmdgroup-details, 2019.

[9] Felienne Hermans and Efthimia Aivaloglou. Do code smells hamper novice programming? A controlled experiment on Scratch programs. *IEEE International Conference on Program Comprehension*, 2016-July(Section IV):1–10, 2016.

[10] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: a controlled experiment using eye tracking. *Empirical Software Engineering*, 22(3):1440–1477, 2017.

[11] Douglas A. Kranch. Teaching the novice programmer: A study of instructional sequences and perception. *Education and Information Technologies*, 17(3):291–313, September 2012.

[12] Jin-Su Lim, Jeong-Hoon Ji, Yun-Jung Lee, and Gyun Woo. Style Avatar: A Visualization System for Teaching C Coding Style. *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1210–1211, 2011.

[13] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[14] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[15] K McMaster, S Sambasivam, and Stuart Wolthuis. Teaching programming style with ugly code. In *Information Systems Educators Conference*, volume 30, San Antonio, TX, 2013.

[16] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[17] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 5–14, New York, NY, USA, 2010. ACM.

[18] Stellan Ohlsson. Learning from Performance Errors. *Psychological Review*, 103(2):241–262, 1996.

[19] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 77–86, New York, NY, USA, 2008. ACM.

[20] Matthew Peveler, Jeramey Tyler, Samuel Breese, Barbara Cutler, and Ana Milanova. Submitty: An open source, highly-configurable platform for grading of programming assignments (abstract only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 641–641, New York, NY, USA, 2017. ACM.

[21] Steven P. Reiss. Automatic code stylizing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 74–83, New York, NY, USA, 2007. ACM.

[22] S. Scalabrino, M. Linares-Vsquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016.

[23] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[24] Bruno L. Sousa, Priscila P. Souza, Eduardo M. Fernandes, Kecia A.M. Ferreira, and Mariza A.S. Bigonha. FindSmells: Flexible Composition of Bad Smell Detection Strategies. *IEEE International Conference on Program Comprehension*, pages 360–363, 2017.

[25] R. van Tonder and C. Le Goues. Defending against the attack of the micro-clones. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, May 2016.

[26] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. Salient elements in novice solutions to code writing problems. *Conferences in Research and Practice in Information Technology Series*, 114:37–45, 2011.

[27] Susan Wiedenbeck and Vennila Ramalingam. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human Computer Studies*, 51(1):71–87, 1999.

[28] Eliane S Wiese, Michael Yen, Antares Chen, Lucas A Santos, and Armando Fox. Teaching Students to Recognize and Implement Good Coding Style. In *Proceedings of the 4th ACM conference on Learning at Scale*, pages 41–50, Cambridge, MA, 2017. ACM.