Improving Assessment of Programming Pattern Knowledge through Code Editing and Revision

Sara Nurollahian
School of Computing
University of Utah
Salt Lake City, Utah
sara.nurollahian@utah.edu

Anna N. Rafferty
Computer Science
Carleton College
Northfield, MN
arafferty@carleton.edu

Eliane Wiese
School of Computing
University of Utah
Salt Lake City, Utah
eliane.wiese@utah.edu

***Abstract*—How well do code-writing tasks measure students' knowledge of programming patterns and anti-patterns? How can we assess this knowledge more accurately? To explore these questions, we surveyed 328 intermediate CS students and measured their performance on different types of tasks, including writing code, editing someone else's code, and, if applicable, revising their own alternatively-structured code. Our tasks targeted returning a Boolean expression and using unique code within an `if` and `else`.**

We found that code writing sometimes under-estimated student knowledge. For tasks targeting returning a Boolean expression, over 55% of students who initially wrote with non-expert structure successfully revised to expert structure when prompted – even though the prompt did not include guidance on how to improve their code. Further, over 25% of students who initially wrote non-expert code could properly edit someone else's non-expert code to expert structure. These results show that non-expert code is not a reliable indicator of deep misconceptions about the structure of expert code.

Finally, although code writing is correlated with code editing, the relationship is weak: a model with code writing as the sole predictor of code editing explains less than 15% of the variance. Model accuracy improves when we include additional predictors that reflect other facets of knowledge, namely the identification of expert code and selection of expert code as more readable than non-expert code. Together, these results indicate that a combination of code writing, revising, editing, and identification tasks can provide a more accurate assessment of student knowledge of programming patterns than code writing alone.

***Index Terms*—Code writing, Code Revising, Code Editing, programming patterns and anti-patterns, code refactoring, code readability, Code quality, code structure**

I. Introduction: Code Structure

Well-structured code uses control flows, idioms, and programming patterns that are suited to the task at hand [1]–[3]. While well- and alternatively-structured code can have the same functionality, well-structured code is easier for others to understand and maintain (e.g., [4]). Code structure is an important element of code readability, and is distinct from other elements such as comments [5], proper naming of variables [6], and formatting [7]. Selecting an appropriate control structure for a task requires programmers to follow conventions that are often implicit – what Soloway and Ehrlich [8] termed discourse rules. For example, repeating code within an if-block and its corresponding else-block violates the general discourse rule of using appropriate language constructs since the code is within a conditional statement but executes in all cases [9].

Ideally, we want students to write well-structured code in regular IDEs by (1) coding with appropriate structures initially, and (2) improving the structure of existing code (their own and others'). However, as a necessary prior step, our community needs more studies on understanding students' knowledge of discourse rules and supports they need to modify alternatively-structured code.

Although our field has taken steps toward understanding students' knowledge of discourse rules, code writing is often the only metric used to measure this knowledge [10]–[13], with anti-patterns interpreted as indicating larger issues or misunderstanding of patterns [1], [14]. However, using the prevalence of anti-patterns as the only metric may underestimate student knowledge. First, students may not be incentivized to attend to code structure. Second, students may use the anti-pattern even when they can use the correct pattern — many do both within the same assignment [15]. Learning science theories explain that deliberate attention can impact performance [16], and that students have knowledge bases that are not always activated when they produce an answer (e.g., write code), but may be activated when they evaluate one (e.g., revise code) [17].

Recent work has begun exploring code editing as another lens for assessing students' knowledge of code structure, where students are given alternatively-structured but functional code and asked to improve its style [18]. Our work builds on the idea of using alternative lenses for evaluating students' knowledge in order to gain a clearer picture of their mastery of code structure choices. We examine whether students' performance on tasks other than code writing indicates gaps in knowledge about code structure, as well as the degree of similarity in performance between writing code, editing someone else's code, and revising one's own code. We explore these through three research questions:

RQ1 **To what extent do anti-patterns indicate knowledge gaps regarding the target code structure?** We explore what amount of information students need to revise their non-expert code by providing progressive hints. Hints start by flagging the code as non-expert and building

to an isomorphic worked example. Our pre-registered hypothesis (https://osf.io/32wuv) is that at least 20% of students who were asked to revise their code would correctly revise it to expert structure at the first prompt. At least 30% would correctly revise without being given the worked example, and at least 50% would successfully revise overall. Note that we did not pre-register hypotheses for the other research questions.

RQ2 **How useful is code writing as a predictor of successful editing and revision?** Specifically, is writing with correct functionality and/or expert style positively correlated with successfully editing someone else's code? For code flagged as non-expert, is code-writing functionality positively correlated with successful revision?

RQ3 **What facets of knowledge (beyond those assessed by code writing) impact students' success in revising their own code and in editing someone else's?** Specifically, considering models that use code writing to predict successful editing and/or revision, how useful is it to additionally include other measures of knowledge (e.g., successful identification of the expert pattern and/or preference for the expert pattern). For predicting successful revision and editing, how predictive is success on the other task?

II. Prior Work

The considerable research on code structure and anti-patterns highlights the challenges in developing approaches for teaching code structure that are effective and scalable.

A. Writing Well-Structured Code is Difficult

Well-structured code uses language constructs and idioms that best suit the task, and follows programming conventions (discourse rules [8]) that make the code easier for others to read and understand. However, it is difficult for students to write well-structured code: Discourse rules are often implicit, and violating them will not always affect overall functionality. Therefore, students may not recognize these violations in their code. Feedback on the correct usage of discourse rules can help, but in-depth feedback requires hand-inspecting student code, which is difficult and time-consuming [19], [20]. Understandably, instructors may focus on code functionality , which can be auto-graded. Consequently, students who are only being graded for functionality (and do not receive feedback on structure) undervalue the importance of writing well-structured code [21].

Scalable ways to help students write well-structured code are needed. Prior work has focused on two general approaches: (1) using static code analyzers for automated feedback on coding assignments, and (2) directly teaching students programming patterns and refactoring.

B. Code Analyzers Are Not Sufficient

To provide automated feedback to students about their code structure, prior work has either (1) leveraged professional static code analyzers or (2) developed pedagogical analyzers specifically for students. In the first approach, researchers use professional static code analyzers like PMD [22], FindBugs [23], and SonarQube [24] to identify and flag the location of anti-patterns. However, professional analyzers flag many issues that are not relevant to novice learning, while ignoring other pedagogically important structural issues common in student code [10]. The messages from these analyzers are difficult to interpret and act on, both for students [25] and also for professional software developers [26], [27]. Further, students' use of static analyzers does not seem to impact their code structure, when compared with students who do not use these tools [10].

In the second approach, researchers develop educational code analyzers specifically for students. The main advantage of using these tools over professional analyzers is that they exclusively check for pedagogically important violations [14], [28]–[31]. However, even these tools have many limitations: current educational code analyzers only capture a small subset of important structural violations. Their hint messages are often very detailed, based on the assumption that students' violations are indicators of significant knowledge gaps or misunderstandings [14]. Yet, because there is limited literature on what level of supports students need, these comprehensive messages could be more information than is needed, adding significant cognitive load for students who interact with the tool. The literature on student interaction with educational analyzer messages is very sparse. One such study, on Autostyle, found that many students had trouble following the hint messages, taking on average four tries to do so correctly [32].

C. Students Can Practice Code Structure Directly

While code analyzers can add support for existing programming assignments, their instructional support is limited by the types of patterns that the analyzers can detect. Another approach is to create dedicated activities for teaching code structure, which can also target anti-patterns that can't yet be detected automatically. Weinman et al. found that faded Parsons problems were effective and time-efficient for teaching specific programming patterns [33]. (Faded Parsons problems present mis-ordered lines of code to re-arrange, where some lines include blanks for the students to fill in.) Keuning et al.'s *Refactoring Tutor* teaches students code structure at the method level. The Refactoring Tutor gives students functional but alternatively-structured code to revise. At any time, students can check their code functionality against test cases, and get progressively informative hints on how to improve the code's structure [30].

A key difference between the two approaches — detecting anti-patterns in programming assignments vs. creating standalone opportunities to learn code structure — is that in the detection approach, only students who use anti-patterns get instructional support, while in the standalone approach, all students get this support. This difference prompts two questions: (1) do students who write alternatively-structured code require extra instruction and practice with code structure, and (2) do students who avoid anti-patterns in their writing not

need further support? The answer depends on the extent to which *usage* of patterns and anti-patterns indicates *knowledge* of patterns and anti-patterns. Here, we focus on the first question.

Most approaches to assessing students' knowledge of code structure in our field rely on counting violations detected by professional or educational code analyzers. It makes sense that usage of an anti-pattern indicates misunderstandings of the correct pattern, and that code writing tasks are crucial for assessing knowledge of code structure [1], [13], [14]. Yet, some studies have found that other methods of measuring understanding of code structure, such as asking students to select the best structured or most readable code, can lead to different conclusions about students' knowledge than only looking at their code writing [34], [35]. Therefore, this study examines the extent of code structure knowledge gaps in students who use anti-patterns, and the effectiveness of using code writing as a predictor for success in code editing and revision tasks.

D. Students May Know More Than They Show

According to Ohlsson's theory of learning from errors, less knowledge is available when performing actions than when evaluating them [17]. This theory explains why checking your work is useful: people can often correct mistakes without new information, because additional knowledge becomes activated. This theory suggests that students who use anti-patterns when writing may be capable of revising them correctly, without instruction on the anti-pattern, since they may have some knowledge of the code structure that was not available at first.

The dual process theory describes another pathway for correct revision without additional instruction. This theory proposes two types of cognitive processes: *autonomous judgement* that is fast, does not require deliberate attention, and executes with little effort, and *deliberate process* that is slower, more reflective, and requires deliberate effort and concentration [16]. Since writing code with readable structures is a deliberate process, greater attention to the task should result in more success. Students may not pay attention to code structure if they believe the task only requires correct functionality. Therefore, alerting students to the expectation of good code structure may provide the necessary motivation for revision. Consistent with these hypotheses, some recent research [18] found that students do not always need extra information on how to edit an alternatively-structured code to use the expert structure. They gave students functional (but non-expert) code to revise, and in their analyses examining what level of progressive series of hints students needed, they found that the amount of needed support varied across different types of control structures. While their main intention was instruction for students, their findings also provides a finer-grained assessment of student knowledge of code quality and structure. In this study, we build on this approach of providing progressively more information. We first start with code that students wrote and we focus explicitly on the amount of support they need to revise errors across several different structures. We also examine how students edit a given piece of alternatively-structured code.

III. Methods

We surveyed 328 CS students across two courses (CS2 and the following course in the major sequence). Students had one week to complete the self-paced online survey. In this paper we mainly focus on the survey's code writing, editing and revising tasks.

A. Survey Overview and Topics

We build on the Readability and Intelligibility of Code Examples (RICE) survey, which consists of five sections: code writing, style and readability preferences, comprehension, code editing, and code revising [35].

Code writing tasks provided a description of the expected behavior and the method signature, and asked students to fill in the method body. All methods could be written with fewer than ten lines of code. For readability and style preferences, students were shown 3-4 code blocks: the readability prompt asked which code block was most readable, and the style prompt asked which had the best style, as an expert would view it. Code editing tasks asked students to improve the style of a given method without changing its functionality. Finally, code revision tasks showed students their own non-expert code from the writing section and asked if they could improve the code's style. The survey questions are available on github[1]. For the tasks discussed in this paper, all students saw the same questions. To control for ordering effects, students were randomly assigned to forward or reversed question order for writing and preference tasks. Editing and revising tasks were ordered by expected difficulty from low to high. Detailed descriptions of these tasks are in section III-B.

The survey tasks targeted 7 control structures topics (patterns). For each topic, the *appropriate structure* indicates the most readable structure (unanimous agreement from three instructors) and *alternative structures* reflects common novice implementations. Each topic involved method-level structures; all are taught in the first semester of CS and can be accomplished with only a few lines of code.

Although other sections of the survey targeted 7 control structures, code revising only targeted 3 topics where alternative code structures could be detected most simply, with regular expressions (allowing them to be checked within our survey software):

- T1 *Returning a Boolean expression with an operator vs. literals*: Returning the expression is appropriate (`return` $x > 7$). An if statement returning `true` or `false` is an example of alternative structures.
- T2 *Returning a Boolean expression with method call vs. literals*: Similar to topic 1, an example with appropriate structure is `return s.equals("a")`.
- T3 *Unique vs. repeated code within `if` and `else`*: When some code is shared across all `if` and `else` branches, it

[1]https://github.com/SaraNrl/ICSE2023

could be written once, outside the `if-else` (appropriate), or repeatedly inside all branches (alternative).

Although the guidelines for expert code in T1, T2, and T3 apply to simple contexts (as targeted by our survey and other prior work [18], [36]), they do not apply in all contexts. For example, sometimes it is more readable to return an intermediate variable or a Boolean literal (rather than an expression) [37], [38]. Exploring how students' structure choices differ across related contexts is an important avenue for future work.

B. Code Revision and Editing

In this paper, we primarily focus on students' code revising performance, support they need to revise correctly, and other facets of knowledge that affect their revising success (RQ1 and RQ2). Thus, while code revising was the final section of the survey, within our methods and results sections, we discuss revising first and editing afterwards.

1) Code Revising: Revision tasks asked students to revise their non-expert code from the code writing section of the survey. For two topics T1 and T2 (returning a Boolean expression), student responses from the code writing section were flagged for revision if they included an `if` statement. Students' writing responses for T3 (Unique vs. repeated code) were flagged if they included an `else` or had more than one `return` statement, (both likely indicators of repeated code for our context). While this strategy for flagging had high accuracy for T1 and T2, it was less accurate for T3. See Table IV for the details.

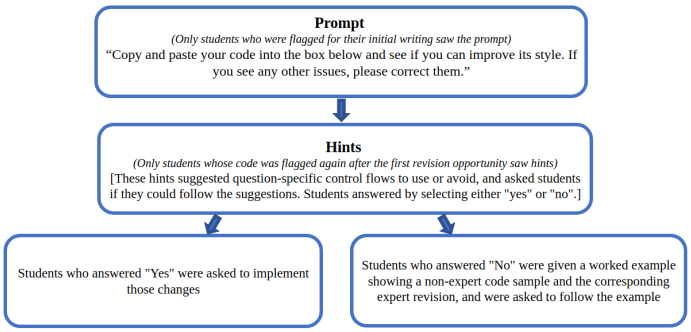

Figure 1: Code revision steps.

Revision opportunities had three steps: (1) Students were shown the first prompt and their own code writing response, and asked if they could improve the style of their code (and correct any other errors they noticed). (2) If that revision was also flagged, the next question offered a hint (e.g., "Can you improve the style of your code by re-writing it without an `if` statement?"). Students indicated if they could follow the hint, and (3) they were given a second chance to revise. Students who said they could not follow the hint were also given a worked example. The worked example showed two code samples matched to the topic, one having the expert-structure and one using an alternative structure. Students were told that the code samples had the same functionality. Fig. 1 indicates a flowchart of code revising steps. The code writing tasks, prompts, hints, and worked examples are shown in Table I (T1 and T2) and Table III (T3).

Return a Bool. expression w/operator	**Return Bool. expression w/method call**
Code Writing Task: Write a function that takes an **int** as input and **returns a boolean.** • when input is 7 return true. • otherwise return false.	**Code Writing Task:** Write a function that takes a **String** as input and **returns a boolean.** • when input starts with "A", return true. • otherwise return false. Hint: `word.startsWith("x")` returns `true` when the String word starts with "x".
Prompt for 1st Revision: Copy and paste your code into the box below, and see if you can improve its style. If you see any other errors, please correct them. **Student responses:**	

`if (num == 7) {`
`  return true;`
`} else {`
`  return false;`
`}`
	`if (word.startsWith("A"))`
	`  return true;`
	`return false;`

Hint for 2nd Revision: Can you improve the style of your code by re-writing it without an if statement?	
Selected: "Yes, I could do that." **Final revision:** `return num == 7;`	Selected: "No, I don't know how to do that."
Worked Example (for students who said they could not follow the hint)	

Check out this example, these two code blocks do the same thing.
 	
Ex. 1:
`if (num > 9 )`
`  return true;`
`return false;`
 	
Ex. 2:
`  return num > 9;`
	Check out this example, these two code blocks do the same thing.
	
	Ex. 1:
	`if (word.equals("ABC"))`
	`  return true;`
	`return false;`
	
	Ex. 2:
	`  return word.equals("ABC");`

[this student said they could follow the hint, so they were not shown the worked example]	**Final revision:** `return word.startsWith("A");`

Table I: Sequence of prompts and hints for T1 and T2 (returning a Boolean expression with either operators or method calls). Two students' answers show their progressive revisions in response to each opportunity. For T2, since the student said they could not follow the hint, they were shown the worked example.

2) Code Editing: The survey included five editing tasks, targeting six topics. In this section, students were given functional but poorly structured code blocks and were asked to edit them for the style without changing the code functionality (with the same inputs, the edited and original code should produce the same outputs). If students thought the code already had the best possible style, they could leave it unchanged. In this study, we only discuss students' performance on the editing tasks that targeted the three topics that were also targeted in the code revising tasks (Table II).

Please copy and paste the function into the box below, and then edit the function so that **it has good style, as an expert would view it**. **Do not change the functionality** (that is, if your new code and original code were called with the same input, they should have the same output.) If the function is already written with the best possible style, simply copy and paste it without editing. Note: you do not need to add comments.

```
public static boolean ending(String word) {
 if (word.endsWith("ing")||word.endsWith("ed")) {
  return true;
 } else {
  return false;
 }
}
```

```
public static boolean between(int max, int min,
 int num) {
 if (min < num) {
  if (max > num) {
   return true;
  }
 }
 return false;
}
```

Table II: Editing questions targeting returning a Boolean expression with operator (top code) and with method call (bottom code). The text prompt was the same for both topics.

C. Participants

Participants were recruited from two intermediate courses in the CS major: data structures and algorithms, and introduction to software engineering (the next course in the major sequence, taught by the same instructor). The instructor publicized the survey to all students and offered extra credit for completion. Students could access the survey and receive extra credit without participating in the research. The study was approved by our IRB (protocol number 00124175). 328 participants consented to the research and skipped no more than one question per section.

D. Data Coding for Non-Multiple Choice Questions

Data from the code writing, editing, and revising items were evaluated in two ways. First, each code block was compiled and evaluated using automated tests to determine if it exhibited the correct functionality. For code that did not compile, small manual edits were made in some cases to fix compile issues while attempting to preserve intent. For example, for an array variable, `arr.length()` could be modified to `arr.length`; allowed changes were included in the pre-registration and refined as we found new errors.

We evaluated responses for structure if they met a threshold for completion. For T1 and T2, the code needed to compile after the allowed manual edits. For T3, the response needed to include code that addressed a specified sub-set of the method requirements (e.g., checking if the ending of the inputted String matched a pre-determined String), but did not need to compile or pass any test cases. Coding guidelines (for categorizing the responses) were developed based on the appropriate structure for each task, refined through examination of student responses to each item, and verified by the course instructor. To evaluate whether responses met the coding guidelines, a combination of human and automated coding was used. T1 and T2 were evaluated with regular expressions looking for any `if` statement, and double-checked by the second and third authors, who specifically looked for ternary operators and insufficient functionality (e.g., always returning `false`). For T3, the second and third authors coded each response independently and resolved any disagreements through discussion[2].

Unique vs. repeated code within an `if` and `else`

Code Writing Task: Write a function that takes a **String** as input and **returns a String.** For input Strings that ends in "sh", concatenate the ending "-ishness" and return a message saying how long the new word is. Follow the format of the examples:

- **Input:** Fish
  - **Output:** Your word was Fish. The length of Fish-ishness is 12.
- **Input:** Hat
  - **Output:** Your word was Hat. The length of Hat is 3.

Write the function so that it would be easy for someone else to modify. Hint: `word.endsWith("sh")` will return `true` if the word ends with ("sh"). The message should be returned not printed. You may use `+` to concatenate strings. For example:

- `String word2 = word1 + " there";`
- If word1 is `"hello"`, word2 is `"hello there"`

Prompt for 1st Revision:

Copy and paste your code into the box below, and see if you can improve its style. If you see any other errors, please correct them.

Hint for 2nd Revision:

We detected at least one of these issues in your code:

- The code has an `else`.
- The code has more than one `return`.
- The code has a `print` statement.

For the correct functionality, there shouldn't be any `print` statements. For optimal style, there should be no `else`, and only one `return`.

Worked Example (for students who said they could not follow the hint):

Check out this example, these two code blocks do the same thing.
Ex. 1:

```
 if (word.startsWith("b"))
  return word +"BBB" + " is a fun word." + word +
  " has"+ (word.length() + 3) +" letters.";
 else
  return word + " is a fun word." + word + " has"
  + (word.length()) + " letters.";
```

Ex. 2:

```
 String secondWord = word;
 if (word.startsWith("b"))
  secondWord = "BBB"+ word;
 return word + "is a fun word." + secondWord +
 " has" + secondWord.length() + " letters.";
```

Table III: Code writing and code revising prompts for Unique vs. repeated code within an `if` and `else`.

Topics	Expert code correctly not flagged	Non-expert code correctly flagged	Expert code incorrectly flagged	Non-expert code incorrectly not flagged
Returning a Boolean expression w/operators	178	148	0	2
Returning a Boolean expression w/method call	193	133	1	1
Unique vs. repeated code within `if` and `else`	101	169	25	33

Table IV: Survey flagging accuracy for the initial writing. Flagging accuracy was lower for unique vs. repeated code.

Topics	Code Writing	Prompt	Hint	Hint + Worked Example
Returning a Boolean expression with operators	54% (178/328)	57% (84/148)	89% (39/44)	71% (12/17)
Returning a Boolean expression w/method call	59% (194/328)	69% (92/133)	88% (23/26)	69% (9/13)
Unique vs. repeated code within `if` and `else`	38% (126/328)	6.5% (11/169)	26% (27/105)	37% (14/38)

Table V: Percentage of students who used expert structure, by stage and topic. For the revision stages, percentages are based on the number of students whose attempts at the prior stage were correctly flagged by the survey software as non-expert.

IV. Findings

A. Anti-Patterns Don't Always Mean Big Knowledge Gaps

1) Returning a Boolean Expression with Operator:

For this topic, 178 students wrote well-structured code. The survey flagged 148 student code writing responses for using alternative structures and missed flagging two student code that used ternary operators. (See Table IV for flagging accuracy.) Of students who wrote alternatively-structured code, 57% successfully revised their structure at the first revision opportunity. Table I indicates revision prompts for the first two topics. Aside from incentivizing the style, the first prompt provided no additional information on how students should revise the code. This implies that the non-expert structured code of these students should not be associated with a lack of knowledge of correct structure. These students only required an incentive or additional time to reflect on their understanding of the structure.

However, 43% of students (64 students) who were flagged for writing with alternative structures could not succeed at the first revising opportunity and were flagged again. These students were shown the second revising prompt and were asked if they can revise the code. Of those, 44 students answered "yes" and were given a final chance of revision, 17 students answered "no" and observed the worked examples and 3 did not answer this question. Finally, 89% of students who only saw the hint and 71% of students who also saw the worked example could properly revise their code.

Overall, 91% of students who were flagged for revision ended up revising successfully (135/148), resulting in 95% students using appropriate structure by the end (313/328). (See Fig. 2, bool w/Operator.)

Examining the code by students who could not properly revise after given opportunities revealed three patterns that may reflect conceptual gaps. These few students may need further support or brief instruction on this topic:

[2]Full coding guidelines in here: https://github.com/SaraNrl/ICSE2023

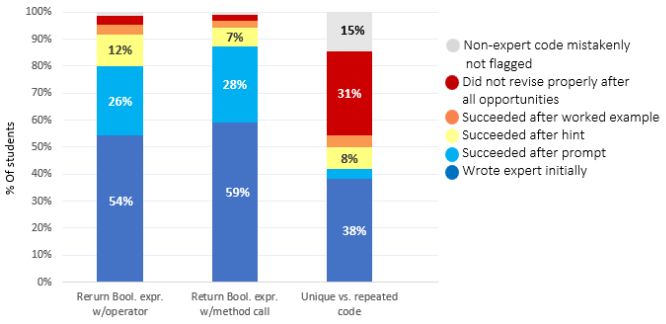

Figure 2: Percentage of students who succeeded in using the expert structure after each revision opportunity. Percentages are shown only for groups with at least 5%. For returning Boolean expressions, many students revised successfully after the first prompt, which simply asked students to improve their code (but did not say what was wrong or how to fix it).

(1) changing `if` statement into a `while` loop (two students); (2) `return int == 7` or `return 7` rather than `return number == 7` (three students); (3) copying and pasting worked examples or inappropriately using it (four students).

2) Returning a Boolean Expression with Method Call: For this topic, 194 students wrote with the expert structure. The survey correctly flagged 133 student code writing responses. (See Table IV for flagging accuracy.) Two students left the survey at that point, and 92 students could properly revise their code at the first prompt (see Table V). Therefore, 69% of the students who wrote with alternative structures could revise correctly without further information, supporting our hypothesis that many students who use alternative structures do not have a deep knowledge gap and code writing alone may not provide an accurate means for measuring student knowledge of code structure. However, 31% could not revise their code to follow the appropriate structure and thus, were flagged again by the survey software and saw the second prompt (41 students, including the two students who left

the revising questions blank). After the second prompt, 26 students answered they know how to modify their code, and 13 answered they don't know how to improve the code. Similar to T1, a few students overestimated their revising capability and could not achieve the expert structure. Of 13 students who saw the worked examples, four students could not revise correctly.

Overall, 93% of students who were flagged for revision ended up revising successfully (124/133), resulting in 97% of students using the expert structure by the end (318/328). (See Fig. 2, T2.) Among students who could not revise after all revision opportunities, common patterns were: (1) changing explicit `else` to an implicit by removing the `else` (two students); (2) copying and pasting of worked examples (two students); (3) copying and pasting their alternatively-structured code from a former revising step (three students).

3) Unique vs. Repeated Code within an `if` and `else`: Although our simple survey detector was accurate for the first two topics, it was less accurate for this topic, resulting in more false positives and negatives (see Table IV). Therefore, interpretation of the results is more complicated for this topic.

For this topic 38% of students wrote with expert structure. The survey flagged 194 students including three students who left the corresponding code writing task blank. Hand inspection of the flagged cases indicated that 169 non-expert code were flagged correctly (including the 3 students who left the code writing tasks blank). However, 25 expert code were flagged incorrectly. Of 169 correctly flagged cases, we couldn't evaluate the structure of 23 code as they did not have the minimum required functionality to be assessed for the structure, and we wanted them flagged.

After the first revision opportunity, only 6.5% of students could correctly remove all of the repeated code (See Table V). Besides those three students who left the code writing task blank, the rest (155 code) either lacked the minimum functionality to be evaluated for the structure or used alternative structures. Of these 158 students who did not revise properly, 71% either copied and pasted their code from the code writing section of the survey or just changed the formatting elements like white space or brackets.

After the first revision, 143 student code got flagged correctly and those students saw the hint message. The prompts for this topic can be observed in Table III. Of students who saw the hint, 105 students answered they know how to revise their code and were given their final revising opportunity. 38 students answered they don't know how to revise, and saw the worked examples. Of students who said they could revise, only 26% succeeded (See Table V), and 78 students overestimated their revision capability.

Of students who saw the worked examples, only 37% could revise correctly, 26% copied the worked examples, 29% either used alternative structures or their code lacked the minimum required functionality to be evaluated, and the rest made no substantial changes to their code from the prior revision opportunity. This suggests that our progressive prompts were not sufficient for this topic.

Overall, only 31% of students who were flagged for revision could revise successfully (52/169), resulting in 54% of students using appropriate structure by the end (178/328). This lower number of students who could successfully revise the code structure indicates that this topic was much more challenging for students compared to the first two.

Among students who did not succeed in revision task for this topic, common patterns were: making explicit `else` implicit, or vice versa; removing one of the `return` expressions that impaired the full functionality of the code; improving the code functionality but not using the appropriate structure like changing `print` statements into `return` statements; and fixing semicolons or parenthesis.

RQ1: ***To what extent do anti-patterns indicate knowledge gaps regarding the target code structure?*** If usage of an anti-pattern were associated with missing knowledge of what the correct pattern is, or how to implement the correct pattern, we would expect that students who used the anti-pattern would need informative hints or other supports to revise. Instead, for the returning Boolean topics (T1 and T2), the majority of students who used the anti-pattern could revise correctly at the first prompt (which asked them to improve their code but gave no information on what to change or how to implement the revision). For Unique vs. Repeated code (T3), very few students revised correctly at the first prompt. These findings show that usage of an anti-pattern may indicate important knowledge gaps in some cases, but not in all cases.

B. Code Writing Is Correlated with Editing but Not Revising

We examined and compared performance on code editing tasks for students who wrote well-structured vs. alternatively structured code. Although students who wrote well-structured code were better at code editing, many students who wrote non-expert code were also successful in code editing, suggesting that students who write with alternatively-structured code can still use the expert code in other contexts.

Code writing style and functionality can weakly predict students' success on code editing and code revising tasks. Overall, we found students' code writing style across all topics is significantly correlated with their editing success. Code functionality, on the other hand, only had a significant correlation with editing success in T2. There was no reliable correlation between code writing functionality and student success in code revision opportunities.

1) Code Writing Is a Weak Predictor of Code Editing: In the code editing section of the survey, students were given functional but alternatively-structured code blocks and asked to edit the style. To observe how students' code writing structure affects their code editing, we separately report code editing results for students who wrote expert code vs. non-expert code for the same topic. Overall, we found that for T1 and T2 (returning a Boolean expression) more than 65% of the students who wrote well-structured code, could also edit their code to follow the appropriate structure. While the percentage of students who edited successfully was lower for students who wrote with non-expert structures, a large

minority of them (more than 25%) were still successful. This finding supports our hypothesis that students' alternatively-structured code writing in many cases is not an indicator of deep conceptual gaps. Similar to our findings for revising, the percentage of students who could edit successfully was much lower for T3 (unique vs. repeated code) than for T1 and T2, indicating that T3 was a more difficult topic.

a) Returning a Boolean Expression with Operator: Table VI shows students' code editing performance for each topic. For T1, of 178 students who wrote well-structured code, 69% could also edit properly for this topic. However, of students who used non-expert structures (150 students), only 27% edited the code to return the Boolean expression directly.

We conducted two chi-square tests for code editing: one on the relationship with code *structure* on the initial writing task and the other on the relationship with code *functionality* on the initial writing task (if the code passed all test cases or not). We found a significant positive correlation between writing structure and editing success ($p < 0.0001$). The correlation between code writing functionality and code editing, on the other hand, was not significant ($p = 0.817$). To determine the extent to which writing style can predict students' editing success, we ran a logistic regression and found that, while code writing is a significant predictor of code editing, it is not a strong predictor (writing style $\beta = 1.9073, p < 0.001, pseudo - R^2 = .14$).

b) Returning a Boolean Expression with Method Call: For this topic, 67% of students who wrote well-structured code and 33% of students who wrote with alternative structures could edit the code block to follow this pattern. Again, we ran two chi-square tests between code writing structure/functionality and code editing and we found that, for this topic, both code writing style and functionality are significantly correlated with editing (writing style vs. editing $p < .0001$); writing functionality vs. editing $p = .040$). Logistic regression with writing style and functionality as independent variables and code editing as dependent variable again reveals that code writing is not a strong predictor of code editing (writing functionality $\beta = .7034$, writing style $\beta = 1.2764, pseudo - R^2 = .15$).

c) Unique vs. Repeated Code within an `if` and `else`: The question targeting this topic was longer and more complicated than the first two. It also targeted two more topics of *conjoined conditions rather than nested `if`s* and *`if-else-if` instead of sequential `if` statements with exclusive conditions* which we do not discuss in this paper. Therefore, for this topic only 14% of students who wrote well-structured code and only 6% of students who wrote with alternative structures could remove repeated code. Chi-square tests for this topic indicates a significant correlation between code editing and code writing style ($p = .00258$) but the correlation between code editing and code writing functionality was not significant ($p = .158$). The Logistic regression with code writing style as the independent variable and code editing as the dependent variable resulted in: ($\beta = 1.4428. p = .012, pseudo - R^2 = 0.08$)

2) Code Writing Functionality is not Significantly Correlated with Code Revising Success: Since the code revising tasks were at the continuation of code writing questions and only students who wrote with less readable code structures were flagged and saw at least one of the revising prompts, we cannot examine the effect of writing style on students' success in code revising opportunities. Thus, we only examined the effect of code writing functionality on students' revising success. Running a series of chi-square tests between students success at each revising opportunity vs. their code writing functionality we could not find any reliable significant correlation for any of the topics.

Readability Prompt: All of these code samples do the same thing. Which one is most readable to you: which one makes it easiest for YOU to understand what the code does?

Style 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 use of language that makes code elegant, efficient, and revealing of design intent.

```
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;
}
```

```
An expert would say they
all have equal style.
```

Figure 3: Prompts for the readability and style questions for *returning a Boolean expression with operators.*

RQ2: ***How useful is code writing as a predictor of successful editing and revision?*** For all three topics, students who used the expected pattern (rather than an anti-pattern) in their own code were more likely to edit another's code correctly. However, this relationship is weak. While logistic regressions show that writing style is a significant predictor for each topic, using it as the only predictor results in models explain a very small portion of the variance (pseudo-R^2 for all three models was $\leq .15$). Code functionality was not a useful predictor for either editing or revising success.

C. What Predicts Students' Success in Code Editing and Code Writing?

As discussed in section III-A, the survey included 5 tasks, including perception of normative style and readability. This allowed us to examine what other aspects of knowledge can predict students' success in editing and revision. Fig. 3 shows the readability/style prompts for T1, *Returning a Boolean expression with operator*.

1) Students Who Correctly Identified the expert Styled Code and Selected that as Most Readable, are More Likely to Edit Properly: To examine the relationship between code editing and other facets of knowledge, we ran three logistic regressions (one per topic). Each regression used code editing success as the dependent variable, with five independent variables: structure at the initial writing task, functionality at the

Topics	Code writing	Copied the given code with no edits	Edited to expert style	Edits remained non-expert
T1: Returning a Boolean expression w/operator	Expert: 178	4 (2%)	123 (69%)	51 (29%)
	Non-expert: 150	16 (11%)	40 (27%)	94 (63%)
T2: Returning a Boolean expression w/method call	Expert: 194	23 (12%)	131 (67%)	40 (21%)
	Non-expert: 134	41 (31%)	44 (33%)	49 (36%)
T3: Unique vs. repeated code within `if` and `else`	Expert: 126	25 (20%)	14 (11%)	87 (69%)
	Non-expert: 202	58 (29%)	6 (3%)	135 (67%)

Table VI: Students' performance on code editing tasks for each topic, separated by their initial code writing structure. More than 25% of students who wrote non-expert code still edited successfully for returning Boolean topics. In contrast, for all topics, more than 30% of students who wrote expert code could not successfully edit.

initial writing task, revision success, and students' selection of expert code on the style and readability questions.

For T1 (*Returning a Boolean expression with operator*), regression result indicates only students' choice of expert style as the best-styled code ($p = .044, \beta = 0.6342$) and selection of the expert code as the most readable one ($p = .005, \beta = 0.9859$) are significant predictors for students' code editing success ($pseudo - R^2 = 0.2594$).

For T2 (*Returning a Boolean expression with method call*) the regression indicates that code writing structure ($p < .001, \beta = 1.2311$), perception of exert style ($p < .001, \beta = 1.4967$) and readability ($p = .001, \beta = 0.9845$) all are significant predictors of students' success in editing ($pseudo - R^2 = 0.2029$).

For T3 (*Unique vs. repeated code within an* `if` *and* `else`) as explained in section IV-B1, very few students could edit the code block targeting this topic. Consequently, the regression result for this topic also shows no significant relationship between students' success in code editing and our predictors.

2) Selection of Expert Code as Most Readable Predicts Success in Writing, and Revising: We used logistic regression to investigate how students' success in code editing, code writing functionality, and their style and readability preferences can predict (1) their initial code writing style, and (2) their revising success. All students are included in the first model (initial writing style). However, for the second model (first and second revision opportunities) only students who were flagged for those revisions are included in the respective models.

For T1 (*Returning a Boolean expression with operators*), the significant predictors of writing with the appropriate structure were: the selection of expert-styled code as most readable ($p = .001, \beta = 1.4712$), writing functionality, ($p < .001, \beta = 2.7587$) and editing success ($p < .001$, $\beta = 1.5613$, $pseudo - R^2 = 0.2313$). For the first revising opportunity, only students' selection of expert styled code as most readable ($p < .001, \beta = 1.6297$) and their success in editing others' code ($p < .001, \beta = 3.2566$, $pseudo - R^2 = 0.2985$) were significant predictors. None of the predictors were significant for success on the second revising opportunity.

For T2, (*Returning a Boolean expression with method call*) the significant predictors of writing with the appropriate structure were: the selection of expert-styled code as most readable ($p < 0.001, \beta = 1.0152$), writing functionality ($p = 0.039, \beta = 0.8275$), and editing success ($p < 0.001, \beta = 1.2632$, $pseudo - R^2 = 0.1260$). However, only code editing was a significant predictor of success on the first revision opportunity ($p < 0.001, \beta = 2.7007$, $pseudo - R^2 = 0.1534$). Again, none of the predictors were significant for success on the second revision opportunity.

For T3 (*Unique vs. repeated code within an* `if` *and* `else`) the significant predictors of writing with the appropriate structure were: the selection of expert-styled code as most readable ($p = .005, \beta = 1.0226$, $pseudo - R^2 = 0.1284$), writing functionality ($p < .001, \beta = 0.5171$), and editing success ($p = .012, \beta = 1.4573$). Again, only code editing was a significant predictor of success on the first revision opportunity ($p = .010, \beta = 2.3963$, $pseudo - R^2 = 0.07839$). None of the predictors were significant for success on the second revision opportunity.

RQ3: ***What facets of knowledge (beyond those assessed by code writing) impact students' success in revising their own code and in editing someone else's?*** Students' identification of expert style and selection of that code as most readable for them were both significant predictors of editing success. Logistic regressions for the two returning Boolean topics show that these predictors are significant even when code writing structure is included in the models. While the inclusion of these additional predictors improved the explanatory power of the models, the models still have a lot of unexplained variance (pseudo-$R^2 \leq .26$). For code writing, selection of expert style as most readable was a significant predictor of success, but correct identification of expert style was not.

V. Discussion

To provide students with appropriate feedback and instruction on code structure, we must first assess their knowledge to determine the level of support they need. Current approaches to assessing students' knowledge have primarily focused on counting structural violations (anti-patterns) in students' code writing assignments and viewing them as knowledge gaps that require remediation through instruction (as in [1], [13], [39]) or comprehensive feedback [14]. However, there are other explanations for students' alternatively-structured code. One is that when students are graded primarily or entirely on code

functionality, attending to code structure may be a low priority, especially when they are short on time [21]. Consistent with the dual process learning theory [16], such students do not need further information to revise their code. They may only need to be prompted to pay attention to structural choices or given time to reflect on their prior knowledge.

Assessing students' code structure knowledge through code editing and revising revealed that code writing did not provide a complete picture of student knowledge: many students who wrote non-expert structured code were able to correctly edit and revise these same structures without receiving any additional instruction or information about expert structure. For return Boolean topics, more than 55% of students who wrote with non-expert code structures were able to successfully revise their code without receiving additional guidance on how to revise. More than 25% of students who wrote with these non-expert structures were able to successfully edit others' code for the same topic. This is consistent with Ohlsson's work that suggests more knowledge is available when evaluating a task compared to when performing it [17]. As a whole, these findings show that for the return Boolean topics, students' usage of anti-patterns does not necessarily indicate a deep conceptual gap in their knowledge, echoing Nurollahian et al.'s finding that students who wrote with an anti-pattern were also capable of using the pattern correctly [15]. Researchers thus must find additional ways to measure student knowledge that go beyond measure anti-pattern usage in student code. A better approach to understanding student knowledge is to employ multiple modes of assessment that include comprehension and evaluation tasks in addition to code writing performance.

We found that students' performance on code writing, editing other's code, and revising their own code was lower on *unique vs. repeated code within an* `if` *and* `else` compared to other topics, suggesting less understanding. One reason for students' greater struggle on this topic could be that correctly editing or revising *unique vs. repeated code* requires working with longer sequences of code and considering more possible modifications. In contrast, converting from non-expert to expert structure for the returning Boolean topics can be done in a few rote steps, with minimal adjustment to apply these steps to a particular snippet of code. This is consistent with previous examinations of students' code editing performance that found simplifying complex control structures to be the most challenging task [30]. Further study is needed to explore gradations of difficulty across other control structures topics in order to determine where students need the most support. Such research could help instructors decide which control structures to prioritize when allocating instructional time, and could help researchers to develop educational code analyzers that provide students with sufficient feedback.

Using multiple modes of assessment reveals significant variation across students' knowledge. For instance, the majority of students wrote expert-structured code or revised correctly after a single prompt for the return Boolean items. However, some students required further support to revise properly, and for a few students our hints were not sufficient. Students who wrote with non-expert structured code are thus not homogeneous, and we cannot view students' knowledge of code structure as a single binary skill, even when focused on a particular topic.

Our findings also indicate that while most students could identify the expert style, the correct identification did not significantly correlate with students' success in writing well-structured code or revising their code. This is in line with Ohlsson's learning theory that makes a distinction between knowledge bases required for performing (writing with expert structure) and evaluating (judging expert style) [17]. While these students can identify the expert style, they may not feel comfortable using it or may not have an appropriate understanding of it. Therefore, only showing the correct usage of the pattern may not be sufficient for students to gain mastery on implementing it in a correct manner.

A final finding was that students exhibited better performance when revising their own code compared to editing someone else's code. One possible explanation is that to edit others' code, students must first understand what the code does. This increases the cognitive load of editing compared to revising. A confounding factor is that the editing questions in our study typically targeted multiple control structures, rather than focusing on each topic in isolation. Overall, our results indicate that code writing, editing, and revising are not strong predictors of one another. Therefore, researchers should be cautious about using one task to measure or predict students' performance in other tasks.

VI. Limitations and Threats to Validity

This study only examined three control structure topics where instances of anti-patterns in student code could be easily identified with regular expressions. Therefore, the results may not generalize to other topics. The appropriate structure for each topic was selected based on unanimous agreement from three instructors. However, this selection is to some degree subjective, and instructors' opinions may not reflect professional software developers' preferences.

Regular expressions were not very accurate for flagging non-expert code for *unique vs. repeated code within an* `if` *and* `else`, resulting in false positives (in which normatively-structured code was erroneously flagged as non-expert) and false negatives (in which non-expert structured code was not detected). Further, we cannot distinguish between students who lacked the knowledge to correctly complete the tasks and those who could have done so but chose not to put in the effort. Those students who did not revise correctly after all prompts may have needed further support, but some may simply not have read the provided prompts.

Although our first revising prompts did not include information on *how* students should revise the code, they did alert students that something was wrong. The student-generated methods were short, simplifying the task of locating the problems. Thus, students' performance in code editing and revising tasks does not necessarily predict how well students could identify issues in their own code in other settings.

Ecological validity is an important concern for our survey. The survey setting lacks some features of the environments in which students typically write code: students could not compile or run their code to identify bugs or syntactic issues. Some students may have felt less comfortable changing non-expert structured (but still functional) code because they lacked the ability to run the code and verify that they had not changed its functionality. The survey was online, self paced, and optional, which may have offered a more relaxed atmosphere than typical course assignments. Further, students received the extra credit regardless of whether the answers were correct, thus, students may have expended less effort than they would on regular course assignments. Since the survey tasks were short and relatively simple, they may not predict performance on longer or more complex tasks. Nevertheless, this study shows the importance of assessing students' knowledge of code structure through measures of code revising and editing. It also indicates how common approaches to measuring students' knowledge, such as counting students' violations, can give an incomplete picture of students' capabilities.

VII. Conclusion

We assessed 328 CS students on their knowledge of code structure. We present results from three tasks (code writing, editing others' code, and revising their own code) on three structures (returning a Boolean expression with and without method calls, and unique code within an `if` and its corresponding `else`). Our results show that code writing is not a very accurate measure of students' knowledge of code structure. Writing expert code is only weakly predictive of editing someone else's code correctly. Further, many students who initially wrote alternatively-structured code were able to revise successfully without additional information on how to do so. A constellation of tasks, including writing, editing, revising, and even selecting which style of code is most readable can provide a more informative assessment of students' knowledge than code writing alone.

VIII. Acknowledgements

This work was supported by the National Science Foundation through the award SHF 1948519.

References

[1] J. Whalley, T. Clear, P. Robbins, and E. Thompson, "Salient elements in novice solutions to code writing problems," in *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, 2011.

[2] M. Guzdial, "Centralized mindset: A student problem with object-oriented programming," *ACM SIGCSE Bulletin*, vol. 27, no. 1, pp. 182–185, 1995.

[3] S. D. Smith, N. Zemljic, and A. Petersen, "Modern goto: Novice programmer usage of non-standard control flow," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, 2015, pp. 171–172.

[4] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" In *2012 28th IEEE international conference on software maintenance (ICSM)*, IEEE, 2012, pp. 306–315.

[5] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, 2005, pp. 68–75.

[6] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension: An empirical study," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, IEEE, 2017, pp. 55–65.

[7] K. McMaster, S. Sambasivam, and S. Wolthuis, "Teaching programming style with ugly code," in *Proceedings of the Information Systems Educators Conference ISSN*, Citeseer, vol. 2167, 2013, p. 1435.

[8] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on software engineering*, no. 5, pp. 595–609, 1984.

[9] S.-N. A. Joni and E. Soloway, "But my program runs! discourse rules for novice programmers," *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 95–125, 1986.

[10] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 110–115.

[11] T. Delev and D. Gjorgjevikj, "Static analysis of source code written by novice programmers," in *2017 IEEE Global Engineering Education Conference (EDUCON)*, IEEE, 2017, pp. 825–830.

[12] D. M. Breuker, J. Derriks, and J. Brunekreef, "Measuring static quality of student code," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 13–17.

[13] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code," in *Proceedings of the 20th Australasian Computing Education Conference*, 2018, pp. 73–82.

[14] H. Blau and J. E. 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*, 2015, pp. 15–20.

[15] S. Nurollahian, M. Hooper, A. Salazar, and E. Wiese, "Use of an anti-pattern in cs2: Sequential if statements with exclusive conditions," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 542–548.

[16] S. Frederick, "Cognitive reflection and decision making," *Journal of Economic perspectives*, vol. 19, no. 4, pp. 25–42, 2005.

[17] S. Ohlsson, "Learning from performance errors.," *Psychological review*, vol. 103, no. 2, p. 241, 1996.

[18] H. Keuning, B. Heeren, and J. Jeuring, "Student refactoring behaviour in a programming tutor," in *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, 2020, pp. 1–10.

[19] C. Hundhausen, A. Agrawal, D. Fairbrother, and M. Trevisan, "Integrating pedagogical code reviews into a cs 1 course: An empirical study," *ACM SIGCSE Bulletin*, vol. 41, no. 1, pp. 291–295, 2009.

[20] M. Woodley and S. N. Kamin, "Programming studio: A course for improving programming skills in undergraduates," in *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, 2007, pp. 531–535.

[21] R. Pettit, J. Homer, R. Gee, A. Starbuck, and S. Mengel, "An empirical study of iterative improvement in programming assignments," in *SIGCSE 2015 - Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Association for Computing Machinery, 2015, pp. 410–415.

[22] *Pmd source code analyzer*, 2022. [Online]. Available: https://pmd.github.io/latest/.

[23] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[24] *Code quality and code security*, 2022. [Online]. Available: https://www.sonarqube.org/.

[25] J. Anderson, "Addressing novice coding patterns: Evaluating and improving a tool for code analysis and feedback," Report UUCS-20-002, University of Utah, Tech. Rep., 2020.

[26] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 672–681.

[27] X. Yang, J. Chen, R. Yedida, Z. Yu, and T. Menzies, "Learning to recognize actionable static code warnings (is intrinsically easy)," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–24, 2021.

[28] J. B. Moghadam, R. R. Choudhury, H. Yin, and A. Fox, "Autostyle: Toward coding style feedback at scale," in *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, 2015, pp. 261–266.

[29] K. Ala-Mutka, T. Uimonen, and H.-M. Jarvinen, "Supporting students in c++ programming courses with automatic program style assessment," *Journal of Information Technology Education: Research*, vol. 3, no. 1, pp. 245–262, 2004.

[30] H. Keuning, B. Heeren, and J. Jeuring, "A tutoring system to learn code refactoring," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 2021, pp. 562–568.

[31] L. C. Ureel II and C. Wallace, "Automated critique of early programming antipatterns," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 738–744.

[32] E. S. Wiese, M. Yen, A. Chen, L. A. Santos, and A. Fox, "Teaching students to recognize and implement good coding style," in *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*, 2017, pp. 41–50.

[33] N. Weinman, A. Fox, and M. A. Hearst, "Improving instruction of programming patterns with faded parsons problems," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–4.

[34] E. S. Wiese, A. N. Rafferty, and A. Fox, "Linking code readability, structure, and comprehension among novices: It's complicated," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, IEEE, 2019, pp. 84–94.

[35] E. Wiese, A. N. Rafferty, and J. Pyper, "Readable vs. writable code: A survey of intermediate students' structure choices," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, 2022, pp. 321–327.

[36] C. Izu, P. Denny, and S. Roy, "A resource to support novices refactoring conditional statements," in *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, 2022, pp. 344–350.

[37] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[38] S. McConnell, *Code complete*. Pearson Education, 2004.

[39] T. Effenberger and R. Pelánek, "Code quality defects across introductory programming topics," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, 2022, pp. 941–947.